MADHYA PRADESH BHOJ(OPEN) UNIVERSITY BHOPAL

Data Structure

BSDS-204

PROGRAMMING IN C AND DATA STRUCTURES

मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल

MADHYA PRADESH BHOJ (OPEN) UNIVERSITY - BHOPAL

Reviewer Committee

1. Dr. Sharad Gangele
   Professor
   R.K.D.F. University, Bhopal (M.P.)

2. Dr. Romsha Sharma
   Professor
   Sri Sathya Sai College for Women,
   Bhopal (M.P.)

3. Dr. K. Mani Kandan Nair
   Department of Computer Science
   Makhanlal Chaturvedi National University of
   Journalism & Communication, Bhopal (M.P.)

• •

Advisory Committee

1. Dr. Jayant Sonwalkar
   Hon'ble Vice Chancellor
   Madhya Pradesh Bhoj (Open) University,
   Bhopal (M.P.)

2. Dr. L.S. Solanki
   Registrar
   Madhya Pradesh Bhoj (Open) University,
   Bhopal (M.P.)

3. Dr. Kishor John
   Director
   Madhya Pradesh Bhoj (Open) University,
   Bhopal (M.P.)

4. Dr. Sharad Gangele
   Professor
   R.K.D.F. University, Bhopal (M.P.)

5. Dr. Romsha Sharma
   Professor
   Sri Sathya Sai College for Women,
   Bhopal (M.P.)

6. Dr. K. Mani Kandan Nair
   Department of Computer Science
   Makhanlal Chaturvedi National University of
   Journalism & Communication, Bhopal (M.P.)

• •

COURSE WRITERS

Rohit Khurana, Faculty and Head, I.T.L. Education Solutions Ltd., New Delhi
Units (1.0-1.2.1, 1.2.3, 1.2.5, 1.3-1.3.2, 1.6-1.6.8, 2.2-2.2.5, 3.0-3.1, 3.2, 3.2.1-3.2.3, 3.2.4,, 3.3, 3.3.1-3.3.2, 3.4, 3.4.1-3.4.3, 3.5.4-3.5.7, 3.5.8-3.5.10, 3.6-3.13, 4.0-4.2, 4.4,4.7-4.14, 5.0-5.3, 5.6-5.13)

Dr. Subburaj Ramasami, Former Professor, S.R.M. University, Chennai
Units (1.2.2, 1.2.4, 1.2.6, 1.4-1.4.6, 1.5, 1.6.9-1.14, 2.0-2.1, 2.3-2.3.7, 2.3.8-2.11)

Munesh Chandra Trivedi, Assistant Professor, Institute of Management Studies, Ghaziabad
Units (4.3, 4.5-4.6, 5.4-5.5)

SYLLABI-BOOK MAPPING TABLE
Programming in C and Data Structures

CONTENTS

INTRODUCTION

A data structure is defined as a set of data elements that represents operations, such as insertion, deletion, modification and traversal of the values present in the data elements. Data elements are the data entries that are stored in the memory for organizing and storing data in an ordered and controlled way. The commonly used data structures in various programming languages, such as C, are arrays, linked lists, stacks and trees. Data structures are of two types, linear and non-linear.

This book, *Programming in C and Data Structures*, introduces you to the basic concepts of data structures. It explains arrays, which can be used to store lists of elements and discusses stacks – a linear data structure, which includes memory representation and the various applications of stacks. Queues, including their representation and trees; linked list and the various types of linked lists; trees including the binary tree, binary search tree and threaded binary tree, along with the various applications of trees; graphs – their properties and applications; searching and sorting data; and hashing are some of the other topics explained in this book.

This book, *Programming in C and Data Structures* is divided into five units that follow the self-instruction mode with each unit beginning with an Introduction to the unit, followed by an outline of the Objectives. The detailed content is then presented in a simple but structured manner interspersed with Check Your Progress Questions to test the student's understanding of the topic. A Summary along with a list of Key Terms and a set of Self-Assessment Questions and Exercises is also provided at the end of each unit for recapitulation.`

UNIT 1 INTRODUCTION TO 'C' LANGUAGE

Structure

1.0 INTRODUCTION

In any programming language, writing even an elementary program requires the knowledge and clear understanding of various data types, variables, constants and operators provided by that language. All these constitute the most basic elements

of a language which are combined to form an instruction. A set of these instructions constitute a program. Generally, the instructions are executed in the sequence in which they appear in the program. However, to make a program more flexible and efficient, the flow of execution can be altered using various control statements. In this unit, you will learn about the concept of structured programming, evolution of C language, advantages and basics of C. You will also learn about flow of control, arrays, structures, pointers and functions. You will also study the history of C language development, tokens, operators and expressions used in C programming. Operators are of arithmetic or unary type. Arithmetic operators include addition, subtraction, multiplication, division and modulus operators. However, in an expression, there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. To avoid ambiguity, there are defined precedence rules for operators in C. Precedence means the evaluation order of operators. In addition, more than one operator may have the same precedence. For instance * and / have the same precedence. To avoid ambiguity in such cases, there is a rule called associativity, i.e., either left to right or vice versa. This means that when operators of the same precedence are encountered, operators of the same precedence have to be evaluated from left to right, if the associativity is left to right. You will learn that C has special operators called unary operators which operate on a single operand. The increment operator ++ (plus plus) and decrement operator – – (minus minus) are unary operators. This unit will discuss relational, logical, assignment and conditional operators. You will also study the use of statements like if and if…else statements.

1.1   OBJECTIVES

After going through this unit, you will be able to:

- Describe the evolution and advantages of the 'C' programming language
- Discuss the concept of structured programming
- Know how C language was developed
- Develop a C program
- Describe the various categories of data types
- Use `if` and `if...else` statements in your C programs
- Explain `break`, `continue` and `goto` statements

1.2   BASIC CONCEPTS OF C LANGUAGE

Designed and developed by Brian Kernighan and Dennis Ritchie, at The Bell Research Labs in 1972, the 'C' programming language is one of the most popular computer languages in today's computer world. It was created so as to allow the programmer access to almost all of the machine's internals—registers, I/O slots

and absolute addresses. In addition to this, 'C' allows for as much data handling and programmed text modularization as is needed to allow very complex multi-programmer projects to be constructed in an organized and timely fashion.

Although this language was originally intended to run under UNIX, there has been a great interest in running it under the MS-DOS operating system on the IBM PC and compatibles. It is an excellent language for this environment because of the simplicity of expression, the compactness of the code, and the wide range of applicability.

Also, due to the simplicity and ease of writing a C compiler, it is usually the first high level language available on any new computer, including microcomputers, minicomputers, and mainframes.

Programming in C can be a great help in the areas where you need to use Assembly Language but would prefer to keep it simple to write and easy to maintain the program. The time saved in coding of C can be quite rewarding in such cases.

Even though the C language has a good record when programs are transported from one implementation to another, there are differences in compilers that you will find anytime you try to use another compiler. You come to know of many differences when you use nonstandard extensions such as calls to the DOS BIOS when using MS-DOS. Nevertheless, these differences can be minimized by careful choice of programming constructs.

When the C programming language gained popularity among users, using a wide range of computers, representatives of the software sector met to propose a standard set of rules for the use of the C programming language. The group represented all sectors of the software industry and after many meetings, and many preliminary drafts, they finally wrote an acceptable standard for the C language. It has been accepted by the American National Standards Institute (ANSI), and by the International Standards Organization (ISO). Although implementation of the program is not compulsory, it would be the loss of the individual/organization not opting for it.

1.2.1 Basic Constructs of Structured Programming

In late 1960s, the high-level languages such as C and Pascal were developed, which provided a structured way of writing programs. Structured programming (also known as procedural programming) was a powerful and an easy approach of writing complex programs. In **procedural programming**, programs are divided into different procedures (also known as functions, routines or subroutines) and each procedure contains a set of instructions that performs a specific task. This approach follows the top-down approach for designing the program. That is, first the entire program is divided into a number of subroutines. These subroutines are again divided into small subroutines and so on, until each subroutine becomes an indivisible unit (Figure 1.1).
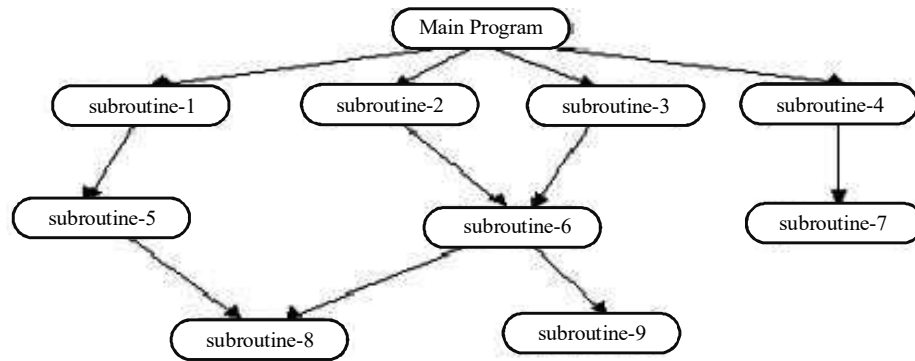
Fig. 1.1 *Top-down Approach*

Programs in procedural programming consist of a controlling procedure known as the **main**, which controls the execution of other procedures. When a call to a procedure is made, the program control is passed to that procedure and all the instructions in that procedure are executed one after another. After executing all the instructions, the program control returns to the procedure, from where the call is made.

For example, to write a program that can add, modify, find and delete students' records from a database using the procedural programming, the entire program can be divided into four different procedures, namely, `add ()`, `find ()`, `delete ()` and `modify ()` (Figure 1.2). In addition to these procedures, the program consists of a main procedure. If a user wants to add a record of a student in a database, the control is passed from the main procedure to the *add* procedure. Once all the instructions are executed in *add* procedure, the control is transferred back to the main procedure. The same steps are followed while calling other procedures.
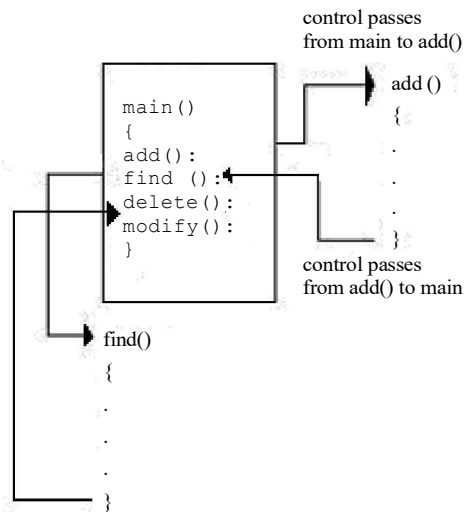
Fig. 1.2 *Procedural Programming*

1.2.2 History of C Language

Martin Richards developed a language called *BCPL* in the mid-1960s. It had many good features. In the year 1970, Ken Thompson, also working at AT&T Bell Labs USA, developed a language called B, which was influenced by *BCPL*. He developed *B* as a compact language for system programming. Subsequently, in 1972, Dennis Ritchie, also working at Bell Labs, designed and implemented C language on the UNIX operating system on a Digital Equipment Corporation (DEC) PDP-11 computer. The C language was, therefore, developed under the influence of BCPL and B. The C language is, however, rich in data types unlike the other two languages. The B language can be considered as the C language without types. The C language added data types such as *char* and *float* to B. By 1973, the C language was ready. The C language further added new features between 1973 and 1980. The C language was used to write UNIX kernel for PDP-11 computer.

The languages BCPL, B and C are procedure-oriented languages similar to FORTRAN. However, they are more compact having few keywords. Simple compilers translate them. They use programs in the standard library for input/output and other interactions with the operating system. During the 1980s, the C language compilers became available for most computer architectures and operating systems. It became a programming tool in PC, which increased its popularity.

Ritchie along with Kernighan published a book *The C Programming Language* in 1978. The Kernighan and Ritchie (K&R) description of the language became a kind of industry standard, popularly called K&R C. Since then, as many C compilers came into existence and there were minor compatibility problems due to variations in implementation, the American National Standards Institute (ANSI) formed a committee (X3JII) for bringing out a standard for the C language. This committee brought out a standardized definition of the C language in 1989. The international standard on the C language, namely ISO/IEC 9899 was released in 1990 by adopting the ANSI standard on the C language. This standard is called C90. Subsequently it was revised in 1999. The revised standard is called C99. Thus, there are three forms of C, namely K&R C, C90 and C99. Even thereafter, minor amendments to the standard are taking place. You will now learn the C language that conforms to the ISO/IEC 9899: 1990 standard, hereinafter called the Standard.

Note:

ISO—International Organization for Standardization

IEC—International Electro-technical Commission

Developing a C Program

A number of integrated development environment (IDE) for developing and executing C programs are available in the market. Freeware compilers for C and a host of other languages are available on the Internet from an organization called The Free Software Foundation. This is called gcc C of GNU. It can be downloaded from the following URLs:

http://www.gnu.ai.mit.edu/software/gec to work with
UNIX operating system http://www.delorie.com/djgpp/ to work
under DOS/Windows environment

A typical methodology for the development of a C program using GNU C
in the Windows environment in the DOS prompt, is given in Figure 1.3.
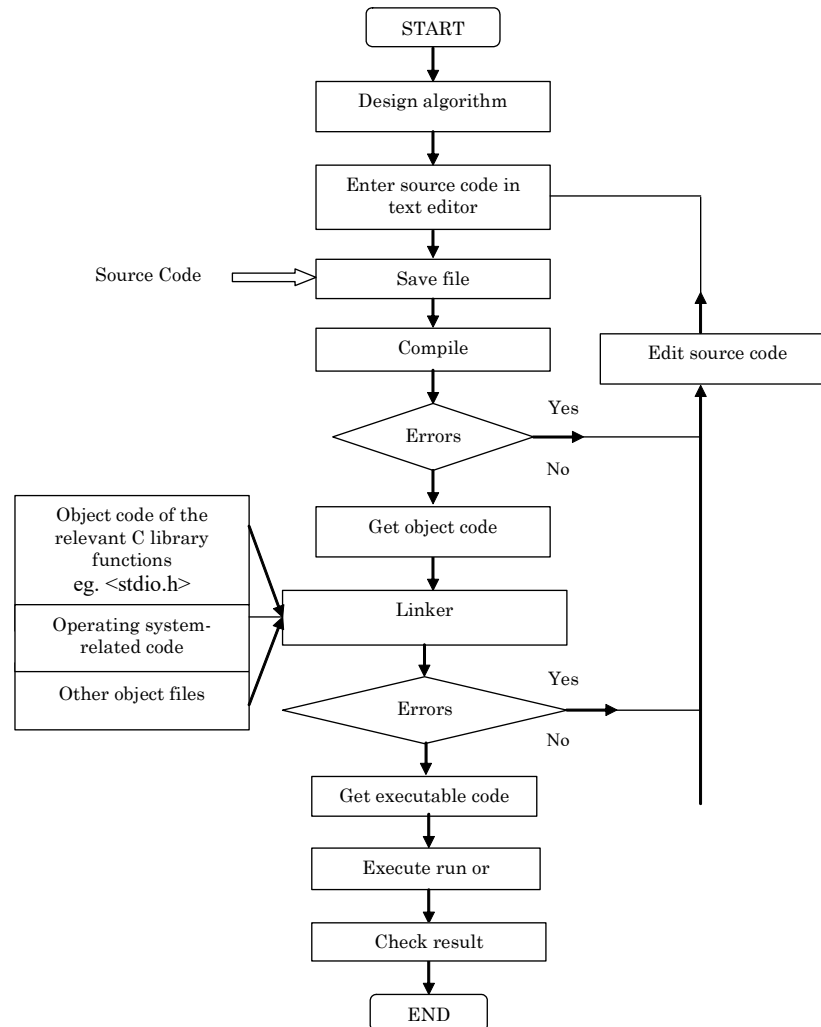
Fig. 1.3 Steps involved in developing a C Program

The various steps involved in program development as given in Figure 1.3 are
briefly discussed below.

Source Code

The program statements written in a high-level language are called source code.
The source code is entered in a text editor. Before attempting to enter the source
code, we should formulate the algorithm and document it in pseudo code. The
pseudo code may be converted into the *C* language code and typed in the text
editor available in the system. After entering the code, it may be saved in a file with
file name .c extension. We can save the file as Example 1.1.c

Object Code

Now we are ready to compile the source code. We have to call the compiler and submit our source code for compilation. If we are using a GNU compiler, we type it in the DOS Command for compilation as follows:

```
gcc-1.1.c
```

If we are using Turbo C++, we select compile and click. After the compilation process is over, there are two possible outcomes:

- There are errors in the source code.

- There are no errors.

If there are no errors, we get an object code file with the same filename with `.o` extension automatically. In case of the above-mentioned example, we will get Ex 1.1.c.

If there are errors, the compiler will give error messages. The error messages will give the line numbers in the program along with the type of error. The programmer should note the line numbers and the error messages and then edit the source file. The statements in the line numbers should be checked for errors. Therefore, the programmer should carefully examine the program and correct all the errors and recompile. When there are no errors, the compiler will automatically generate the object code and save it with a `.o` extension with the file name same as that of the corresponding source code.

The object code is in the machine language. However, it is not yet ready for execution. If your program contains more than one source code file, all the source files are to be compiled separately to get the corresponding object code. Assume that you have another source file called `px.c`. You will get `px.o` when the compilation is successful.

Linking and Loading

During this process, you combine all the object files. In this case, Ex 1.1.c and `px.o`. Additionally, you will add the object code corresponding to the header files included in the source code or program under execution. For instance, you will combine the object code corresponding to the `printf()` function in the `<stdio.h>` file.

Another important code is that corresponding to the particular operating system of the computer system. This is essential for executing the program in the given hardware and the operating system. The linker to get an executable file for the source file will combine all these. The executable file will be in the machine code. It will be generated automatically on successful linking and loaded for execution. You get Ex 1.1.c.exe in this case after successful linking.

1.2.3   Advantages of C Language

Some of the advantages of C language are as follows.

- **Readability**: It is easier to learn and understand because it is very close to human languages.

- **Machine independent**: Programs written in C language are portable as codes. Once written they can be used on different platforms (machines) also.

- **Easy debugging**: Programs written in this language are easy for debugging. Debugging means to rectify the errors in a program. High-level languages require an interpreter or a compiler to detect error(s) and helps programmers to correct errors.

- **Easy maintenance**: Programs written in C are modified easily as they are similar to human languages.

- **Reusability**: C codes are reusable, that is, once the programs are loaded into a library they can be used by other applications also.

- **Structured programming**: Code written in C can be broken down into several functional programs. This can be developed independently and then combined into a single program.

- **Low development cost**: Programs written in C language increase programmers' productivity (number of lines of code generated per hour).

- **Easy documentation**: Programs written in C provide easy documentation for future maintenance.

1.2.4 Components of C Language

Standard defines the six classes of tokens in C programming language:

(i) Keyword

(ii) Identifier

(iii) Constant

(iv) String literal

(v) Operator

(vi) Punctuators

Tokens are similar to atomic elements or building blocks of a program. A C program is constructed using tokens. There are certain other building blocks of a program that do not form part of any of the above. They are as follows:

- Blanks
- Horizontal tabs
- Vertical tabs
- New line characters
- Form feed
- Comments

C Character Set

The C language supports and implements the American Standard Code for Information Interchange (ASCII) for representing characters. The ASCII uses 7 bits for representing each character or digit. The characters are coded from 0000000 (decimal 0) to 1111111 (decimal 127). Therefore, the ASCII consists of code for 128 characters in all. The ASCII values (decimal equivalent of the 7 bits) of some alphabets and digits are given in Table 1.1.

Table 1.1 *ASCII Values of Selected Alphabets*

ASCII Value	Character or Digit
48	0
49	1
57	9
65	A
66	B
67	C
89	Y
90	Z
97	a
98	b
121	y
122	z

The digits and alphabets are organized sequentially and hence, it is easy to get the ASCII value; for instance, the ASCII value of D is 68, E is 69, 8 is 56, *x* is 120 and so on. The ASCII table is given in Annexure 1.

Identifiers

Any name is an identifier. Just as the name of a person, street or city helps in the identification of a person or a street or a city, the identifier in the C language assigns names to files, functions, constants, variables, etc. An identifier in the C language is defined as a sequence of alphanumeric characters, i.e., alphabets or digits. The first character of an identifier has to be an alphabet. In the C language, lowercase alphabets and uppercase alphabets are considered to be different. For instance, VAR and var represent different names in the C language.

VALID IDENTIFIERS

```
C1
PROC1
P34
VAR_1
EX1
a
bc
Ual1
Aa
```

INVALID IDENTIFIERS

```
1PROGA
4.3
A-B
```

Any function name is also an identifier. For instance, 'printf' is the name of function available with the C language system. The function helps in printing. Therefore, identifiers can be constructed with alphabets (A...Z), (a...z), (0...9). In

addition, underscore can also be used in identifiers. Unless otherwise specified, small letters are usually used for identifiers.

Keywords

These are also known as reserved words in C. In the first program, 'int' is a reserved word or keyword. They have a specific meaning to the compiler. They should be used for giving specific instructions to the computer. These words cannot be used for any other purpose such as naming a variable. C is a very concise language containing only 32 reserved words and this is one of its strengths. Common statements, such as print, input, etc., are implemented through library functions in C, giving relief to programmers and reducing the size of code as compared to other programming languages. This makes the task of programming simple.

Now take a look at the keywords given in Table 1.2. You will come across most of them in the book. Their meaning will become clear as you read the book.

Table 1.2 C Keywords

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

1.2.5   Structure of a C Program

Programs are a sequence of instructions or statements. These statements form the structure of a C program. To understand the structure of a program, consider Figure 1.4.
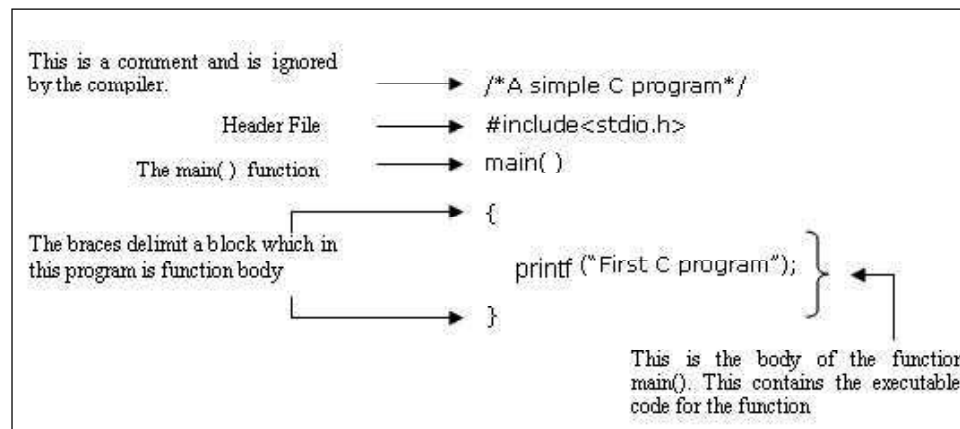
Fig. 1.4   Structure of a C Program

The structure of a C program can be broadly classified into *header files* and the *main() function*. They can be described as follows:

- **Header files**: While writing programs, programmers use various programming elements defined in the Standard C Library. These pre-

defined programming elements include library functions, variables, constants, etc. In order to use such pre-defined elements in a program, appropriate standard header files must be included in the program. The standard header files are files that contain the information (like the return type of library functions, data type of constants, etc.) required by C compilers in order to use the pre-defined elements. In other words, they contain declarations of library functions, type definitions and so on. As a result, programmers do not need to explicitly declare (or define) the programming elements. Standard header files are specified in a program through the preprocessor directive, `# include`

In Figure 1.4, the `stdio.h` header file is used. When the compiler processes the instruction `#include <stdio.h>`, it includes the contents of `stdio.h` in the program. Once the contents of the `stdio.h` header file are included, programmers can work with the *standard input*, *output* and *file system* facilities. The name `stdio.h` file stands for standard input-output header file. The `stdio.h` file contains numerous prototypes and macros to perform input or output (I/O) operations for C program. The standard I/O functions defined in `stdio.h` are discussed here.

- **printf()**: It is the standard library function used to display any message or values on screen. It takes as arguments a format string and an optional list of variables or literals to output. The `printf()` takes the following form:

```
printf("format-control-string",variable-list);
```

The `format-control-string` controls the format of the output and it can consist of the following:

1. Sequence of characters

2. Conversion specifications that always begin with a `%` sign and end with a conversion specifier

3. Escape sequences that always begin with a `\` sign.

The `variable-list` (which is optional) consists of variable names separated by commas whose values are to be printed. This list should include a variable corresponding to each conversion specifier.

- **scanf()**: It is also a standard library function used to receive formatted input from keyboard. The `scanf()` takes the following form:

```
scanf("format-control-string", variable-address-
list);
```

The `format-control-string` describes the format of the data to be input. It can consist of conversion specifications and character literals. The `variable-address-list` is a list of addresses of the variables in which the input values are to be stored. This list should include an address for each conversion specification. The address of variable is obtained by using ampersand (`&`) followed by the variable name.

- **main()**: The first executable instruction in all C programs is the `main()` function. That is, programs begin their execution from this instruction.

Once all the instructions in the `main()` function are executed, the control passes out of `main()`, terminating the entire program. Generally, large and complex programs are divided into smaller subprograms known as functions. Out of all the functions defined in a program, the `main()` function is one of the most important functions.

In Figure 1.4, `main()` is the beginning of its function definition. The word `main` refers to the name of function and the following '`()`' indicates that it is a function. The opening and closing curly braces '`{ }`' enclose the `main()` *function's body*. All C statements that need to be executed are written within the `main()` function's body. The statement `printf("First C Program";` written within the `main()` function's body, displays the message `First C Program` with the help of the `printf( )` function.

Note: Every C program must have one and only one `main()` function.

1.2.6 A Sample C Language Program

A program in the C language is given below, which displays a text. Have a close look at the program.

```
/* Example 1.1*/
/* program for displaying a text */
#include <stdio.h>
int main()
{
printf ("Om Vinayaga") ;
}
```

Create a new file in the text editor. Then type in the program exactly as given above. Save this program as Example 1.1. Next compile the program. The compiler after compilation will give a message. Look at the message box. If the C program was compiled in a C++ compiler, there may be warnings, but you can safely ignore them. However, the errors, if any, should not be ignored. The compiler will give the errors and line numbers. Sometimes, even an experienced programmer will find it difficult to understand the error messages. If you encounter the same difficulty in understanding the error messages, you need not worry. Open the program file again and check whether you have typed it exactly as in the book including the semicolons, quotation marks and brackets and the program is a working program. Keep checking and compiling till the compiler says 'success', meaning that there is no error in the program after compilation. Now you have to link the object code to get the executable code. When there are no errors even after linking, you have to execute or run the program. Use the right command for 'run'. On execution, you will get the result as given below:

Result of the program

```
Om Vinayaga
```

You have succeeded in establishing a communication link with the computer. You can now talk to it regularly by learning new commands and using better communication methods. You have now become familiar with the methodology for learning a programming language. The steps involved are summarized as follows:

- Type the program in a text editor
- Save and give a name to the program
- Compile the program
- Look for errors and correct them
- Link the object code to produce executable code
- Execute/Run the program and analyse the result

The methodology for the preparation of the source code, compilation, linking and execution may vary from system to system. Learn the correct operations of the IDE/system being used for the various steps mentioned above.

Variation in the Main Function

Most compilers have implemented K&R C. Some have implemented C90. Not many have implemented the new features introduced in C99. Hence, it is important that the reader checks what standard his compiler supports. In the Example 1.1, we did not explicitly return anything. On successful execution, the program returns zero. Return of any other integer means that the program execution was not successful. Hence, you can add a statement `return 0` at the end of the program.

The main function did not receive any value, which was indicated by the empty parentheses. We can indicate nothing by the keyword `void`. The modified program including the above two features is given below:

```
/* Example 1.2*/
/* program for displaying a text */
#include <stdio.h>
int main(void)
{
printf ("Om Vinayaga") ;
return 0;
}
```

Result of the program

```
Om Vinayaga
```

Thus, both these programs have carried out the task successfully. However, in the rest of the book, the style given in the previous program will be used. You may adopt the style that works in your compiler.

Check Your Progress

1. What do you mean by `main` in procedural programming?
2. What is a source code?

1.3 DATA TYPES

A data type determines the value that a variable can take and the operations that can be performed on that variable. C provides several primitive data types. It also provides the facility of defining new data types according to the requirements of the programmer. In this section, the **primitive data types** provided by C, and **composite data types** such as arrays and pointers that are derived from the primitive data types will be discussed. Structures and union which are **user-defined data types** will be discussed here.

1.3.1 Primitive Data Types

The basic (fundamental) data types provided by C are `int`, `float`, `char` and `double`. The data types `char` and `int` are collectively known as integral data types. The `char` data type occupies 1 byte of memory (that is, it holds only one character at a time). It can be preceded by `signed` and `unsigned` modifiers. The `int` data type is used to store integers such as `41, 12, 521, –321`, etc. Like `char` data type, `int` can also be qualified with `signed` and `unsigned` modifiers. In addition to these modifiers, it can be also be qualified with `short` and `long` modifiers.

The `float` and `double` data types are used to store numbers with decimal point. The `float` specifies single-precision floating-point numbers and `double` specifies double-precision floating-point numbers. In addition to these data types, C also provide a special data type known as `void`, which is used for specifying an empty parameter list to a function and return type for a function. When `void` is used to specify an empty parameter list, it indicates that function does not take any arguments and when it is used as a return type for a function, it indicates that a function does not return any value. The various primitive data types with their size and range are listed in Table 1.3.

Table 1.3 Primitive Data Types

Type	Size (in bytes)	Range
int	2	–32,768 to 32,767
signed int	2	–32,768 to 32,767
unsigned int	2	0 to 65,535
short int	2	–32,768 to 32,767
signed short int	2	–32,768 to 32,767
unsigned short int	2	0 to 65,535
long int	4	–2,147,483,648 to 2,147,483,647
signed long int	4	–2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
char	1	-128 to 127
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	$3.4*10^{-38}$ to $3.4*10^{38}$
double	8	$1.7 * 10^{-308}$ to $1.7*10^{308}$
long double	10	$3.4 * 10^{-4932}$ to $1.1*10^{4932}$

1.3.2 Composite Data Types

The two most commonly used composite data types in C are *arrays* and *pointers*.

Arrays: Arrays are a set of sequentially indexed elements of the same data type.

All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique *index* or *subscript* value. The subscript value indicates the position of the element in an array. For example, the array `a[5]` is used to store five values and the individual elements are referred to as `a[0]`, `a[1]` and so on.

Pointers: Pointers are variables that store the memory address of another variable. For example, the instruction `int marks=24` creates the variable `marks` and stores it in memory. However, to store the address of marks, a pointer variable needs to be declared. The instruction int `*pmarks` declares a pointer variable and the instruction, `pmarks=&marks` stores the address of `marks` in `pmarks.`

User-Defined Data Types

The various user-defined data types provided by C are *structures, unions* and *enumerations*.

Structures: Structures are the collection of data items referred to by a common name. A structure provides a convenient way of keeping related information together. For example, in the library management software, a record for a book consists of information such as `book name`, `price` and `pages`. This information can be stored and referenced as individual data types or stored and referenced using a common name by declaring a structure.

Unions: Unions, like structures, consist of members of the same data type or different data types. The only difference is that all the members of a union share a common storage area, whereas each member in a structure is assigned its own unique storage area. Hence, unions are used to conserve memory.

Enumerations: An enumeration is a set of integer constants, written as identifiers, specifying the possible values that can be assigned to the enumeration variables. These set of possible values are known as **enumerators**. By default, the first enumerator in the enumeration data type is assigned the value zero. The values of subsequent enumerators are increased by one. For example, an enumeration for days in a week is declared as given here.

```
enum {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday}days;
```

This statement declares an enumeration variable days, consisting of enumerators `Sunday, Monday, Tuesday, Wednesday, Thursday, Friday` and `Saturday`. These enumerators represent integer constants `0, 1, 2, 3, 4, 5` and `6`, and can be used anywhere in the program like an integer.

Check Your Progress

3. What is the function of data type?

4. Name the two most commonly used composite data types in C.

5. Define pointers.

1.4 CONSTANTS

The following are the types of constants:

- Integer constant
- Character constant
- Float constant
- Enumeration constant
- String constant
- Symbolic constant

All these types are explained below.

1.4.1 Integer Constants

The following are the types of integers:

```
int
unsigned int
long
unsigned long
```

Short, long and unsigned

Unsigned

We can use the sign bit also for holding the value. In such cases, the variable will be called `unsigned int`. The maximum value of an `unsigned int` will be equal to 65535 because we are using the Most Significant Bit (MSB) also for storing the value. The minimum value will obviously be 0.

Long

A long integer is represented as `long int` or simply `long`. The maximum and minimum values of `long` are given below:

| LONG | MAX | + | 2147483647 |
| LONG | MIN | – | 2147483647 |

Unless otherwise specified, integers or long integers will be signed, i.e., the first bit will be reserved for the sign. The `long int` obviously uses 4 bytes or 32 bits.

The magnitudes of `long` can also be doubled by using an unsigned long integer denoted as `unsigned long`.

However, integers are not suitable for very low values and very large values. This can be overcome by floating point or real numbers.

An integer constant may be suffixed by the letter `u` or `U` to specify that it is an `unsigned integer`. Similarly, if the integer is suffixed with `l` or `L`,

it signifies a `long integer`. If we specify `unsigned long integer` we suffix the constant with `ul` or `UL`.

The following are the examples of valid and invalid integers:

Valid Integers

```
+345      /* integer */
 345      /* integer */
-345      /* integer */
729u      /* unsigned integer */
729U      /* unsigned integer */
-112345L  /* Long integer */
112345UL  /* Unsigned Long integer */
+112345l  /* Long integer */
112345l   /* Long integer - if no sign precedes, it is a
positive number */
```

Invalid Integers

```
345.0       /* decimal point not allowed */
112, 345L   /* no comma allowed */
112 345UL   /* = blank not allowed */
112890345L  /* exceeds the maximum */
+112 345UL  /* unsigned cannot have + */
(345l       /* ( not allowed */
-345s       /* illegal characters */
```

Short

There is another integer type called `short` or `short int`. The storage space allocation for this modified type of integer is implementation dependent. But, it will occupy the same or less space than `int` type.

We have so far considered only decimal numbers. The C language, however, entertains other type of numbers as well. The octal numbers will be preceded by 0 (zero).

The following are the examples of valid and invalid octal numbers:

Valid Octal Number

```
0346
0547
0120
```

Invalid Octal Number

```
0394    /* 8 or 9 are not allowed in an octal number */
0 x 345  /* prefix has to be zero only */
```

The C language also supports hexadecimal numbers. Here, since the base is 16, we use alphabet also in the numbers as given in Table 1.4.

Table 1.4 *Alphabet for Hexadecimal Numbers*

a	or	A	for	10
b	or	B	for	11
c	or	C	for	12
d	or	D	for	13
e	or	E	for	14
f	or	F	for	15

Additionally, hexadecimal numbers will be preceded by `0X` or `0x`, i.e., zero followed by `x`.

The following are the examples of valid and invalid hexadecimal numbers:

Valid Hexadecimal Numbers

```
0x345
0xA132
0x100
0x20B
```

Invalid Hexadecimal Numbers

```
0x, 123   /* no comma */
0x        /* cannot be empty */
0A00      /* x is missing */
```

1.4.2 Character Constants

A character constant is a single character enclosed in single quotes as in `'x'`. Characters can be alphabets, digits or special symbols.

The following are the examples of valid and invalid character constants:

Valid Character Constants

```
'A'
'Z'
'C'
'c'
```

Invalid Character Constants

```
'\n'
'\t'
'\u'
'\b'
AA
'AA'
"AA"
'1a'
```

A character constant represents its integer value as defined in the character set of the machine. Therefore, you can add 2 characters. For example, the ASCII values of digit 1 = 49 and C = 67. When we add these values we get code 116 whose

equivalent character is t.

Let us verify this with the following example:

```
/* Example 1.3
demonstrates that chars can be treated like integers*/
#include <stdio.h>
int main()
{
const char ALPHA1='1';
char alpha2='C';
char alpha3;
alpha3=ALPHA1+alpha2;
putchar(alpha3);
return o;
}
```

Result of the program

```
t
```

Therefore, characters can be treated like integers as well, although they are declared as character variables and constants. Since characters are of type int, we could add them. Characters can also be defined as integers as given in the following example:

```
/* Example 1.4
Demonstrates that a char can also be declared as int*/
#include <stdio.h>
int main()
{
int x;
x='1'+'C';
printf("x as integer=%d\n", x);/*x printed as integer*/
printf("x as character=%c\n", x);/*x printed as character*/
return 0;
}
```

Result of the program

```
x as integer=116
x as character=t
```

1.4.3   Floating Point or Real Numbers

Let us enumerate the difference between floating point and integer numbers.

- Integers are whole numbers without decimal points but a float has always a decimal point. Even if the number is a whole number, it is written with a decimal point. For instance, 42 is an integer, while 42.0 is floating point number.

- Floating point numbers occupy more space for storage as we have already seen.

A real number in the simple form consists of a whole number followed by the decimal point and also one or more decimal numbers following the decimal point, which makes the fractional part. This form of representation is known as fractional form. It must have a decimal point. It could be either positive or negative. As usual the default sign is positive. No commas or blanks or special characters are allowed in between.

The following are the examples of valid and invalid float types:

Valid Floats

```
144.00
226.012
```

Invalid Floats

```
+144   /* no decimal point  */
1,44.0 /* comma not allowed */
```

Scientific notation: Floating point numbers can also be expressed in scientific notation. For example, `3.0 E`$_2$ is a floating point number. The value of the number will be equal to $3.0 \times 10^2 = 300.0$

Instead of the upper case `E`, the lower case `e` can be used as,

`0.453 e + 05,` which will be equal to `0.453` $\times$ `10`^{5} `= 45300`

There are two parts in the scientific notation of a real number, which are as follows:

- Mantissa (before e)
- Exponent (after e)

In the scientific form the following rules are to be observed:

- The mantissa part can be positive or negative.
- The exponent must have at least one digit, which can be a positive or negative integer. Remember the exponent cannot be a floating point number.

  `type float` is a single precision number occupying a storage space of 4 bytes.

  `type double` represents floating point numbers of double precision and hence occupies 8 bytes.

  If you look at the file `<float.h>` you will find the maximum and minimum floating point numbers as given below.

FLT – MAX	1E	+ 37	maximum floating point number
FLT – MIN	1E	– 37	minimum floating point number

Floating point constants: The constants are suffixed as given below:

`F or f`	–	`float`
no suffix	–	`double`
`L or l`	–	`long double`

If an integer is suffixed with `L` or `l`, then it is a `long integer`.

If a float is suffixed with `L` or `l`, then it is a `long double` floating point number.

Examples

Valid Floating Point Constants

```
1.0 e 5
123.0 f        /* float          */
11123.05       /* double         */
23467.34 e 5 l /* long double    */
```

Invalid Real Constants

```
245.0      /* invalid float, but valid double    */
456        /* It is an integer                   */
1.0 e 5.0  /* exponent cannot be a real number   */
```

When they are declared as variables, they can be declared as follows:

```
float  a = 3.12;
float  a, b, c;
float val1;
float val2;
long double val3;
```

The values of constants cannot be altered in programs. They can be defined as follows:

```
const int PRINC = 1000;
const float INT_RATE = 0.12 f;
```

The values of `PRINC` and `INT_RATE` cannot be changed in the program even by introducing another assignment statement when they are declared as constants using const.

1.4.4 Enumeration Constant

The keyword for this data type is `enum`. We can define `guardian` as follows:

```
enum guardian
{
   father,
   husband,
   guardian
};
```

Note that there are no semicolons, but only a comma after the members except the last member. There is not even a comma after the last member. There is no conflict between both guardians. The top one is `enum` and the bottom one is a member of `enum` guardian. See the similarity between `struct`, `union` and `enum`. The `enum` variables can be declared as follows:

```
enum guardian          emp1, emp2, emp3;
```

This is similar to structures and unions. The first part is the declaration of the data type. The second part is the declaration of the variable.

The initial values can be assigned in a simple manner as given below:

```
emp1 = father;
emp2 = husband;
emp3 = guardian;
```

We have to assign only those declared as part of the `enum` declaration. Assigning constants not declared will cause error. The compiler treats the enumerators given within the braces as constants. The first enumerator father will be treated as 0, the husband as 1 and the guardian as 2. Therefore it is strictly as per the natural order starting from 0.

The enumerated data type is never used alone. It is used in conjunction with other data types. We can write a program using `enum` and `struct`. It is given below:

```
/*Example 1.5
enum within structure*/
#include <stdio.h>
void main()
{
    enum guardian
    {
        father,
        husband,
        relative
    };
    struct employee
    {
        char *name;
        float basic;
        char *birthdate;
        enum guardian guard;
    }emp[2];
    int i;
    emp[0].name="RAM";
    emp[0].basic= 20000.00;
    emp[0].birthdate= "19/11/1948";
    emp[0].guard= father;
    emp[1].name="SITA";
    emp[1].basic= 12000.00;
    emp[1].birthdate= "19/11/1958";
    emp[1].guard=husband;
    for(i=0;i<2;i++)
        {
            if( emp[i].basic ==12000)
            {
                printf("Name:%s\nbirthdate:%s\nguardian:
%d\n",
```

```
              emp[i].name,  emp[i].  birthdate,
    emp[i].guard);
          }
      }
}
```

Result of the program

```
Name:SITA
birthdate:19/11/1958
guardian: 1
```

The program clearly assigns the relationships between the `employee` and the `guardian`. `enum guardian` is the data type, and `guard` is a variable of this type.

However, when you are printing `emp [i]. guard`, you are printing an integer. Hence 0, 1 or 2 will only be printed for the status, and this is a limitation. This can be overcome by modifying the program.

Even though the conversion of an integer to actual name is additional work, `enum` is an useful construct since it improves readability in addition to number of other advantages.

For example, we can define `boolean` as follows:

```
  enum boolean     {
          false,
          true
      };
```

Here 'false' will contain an integer value 0 and 'true' 1. This can be used to assign values for 'found' in our sorting programs. We can define `found` as follows after declaring `boolean`.

```
      enum boolean found;
```

We have allowed the system to assign integer values to the members of `enum`, but we can also assign specific values to the various members of `enum`. When values are assigned, then it takes precedence over what the system assigns.

We can define `boolean` as:

```
    enum boolean
    {  yes = 1,
       no = 0
    };
```

This is similar to defining using `#define`. The `#define` equivalent for this will be:

```
  #define yes 1
  #define no 0
```

Although `#define` and `enum` provide a way to associate symbolic names such as boolean with constants, there are difference between them. The differences are:

(a) enum can generate values itself unlike #define where you have to specify the replacement constant.

(b) The compilers need not check the validity of what is stored in the enum variable, but the #define replacement constant will be checked for valildity.

(c) It is possible to print out the values of enum variables in symbolic form, but this is not possible with #define. Anyway, either of them can be used depending on the context.

1.4.5 String Constants

A character constant is a single character enclosed within single quotes. A string constant is a number of characters, arranged consecutively and enclosed within double quotes.

Examples of valid strings:

```
"God"
"is within me"
"  "
```

You may be surprised about the third string constant, which has no characters. This is called a NULL or empty string and is allowed in C.

The string constant can contain blanks, special characters, quotation marks, etc. within the string. In order to distinguish the end of a string constant, the compiler places a null character \0 (back slash followed by zero) at the end of each string before the quotation mark. The null character is not visible when the string appears on the screen. The programmer does not include the null character either. It is the compiler, which automatically inserts the null character at the end of every string.

Invalid string:

```
'Yoga'     /* should be enclosed in double quotes */
```

Symbolic Constants

The format for symbolic constant is as follows:

```
#define name constant
```

For example, we can define:

```
#define INITIAL 1
```

It defines INITIAL as 1.

The INITIAL type of definition is called symbolic constants. They are not variables and hence, they are not defined as part of the declarations of variables. They are specified on top of the program before the main() function. The symbolic constants are to be written in capital or upper case letters. Wherever the symbolic constant names appear in the program, the compiler will replace them with the corresponding replacement constants defined in the #define statement. In this case, 1 will be substituted wherever INITIAL appears in the program. Note that there is no semicolon at the end of the #define statement.

1.4.6 Logical Constants

In the above examples, you have been checking one condition at a time. It would be nice if you could check more than one condition at a time. 'C' provides three logical operators for combining more than one condition. These are as follows:

Logical `AND` represented as `&&`

Logical `OR` represented as **||**

`Negation` or `NOT` represented as `!` (exclamation).

Take a look at the following example.

if x `>` y and if x `>` z, then x is the greatest.

You will represent the same as,

```
if ((x > y) && (x > y))
printf ("x is the greatest");
```

You will see that the program has become much more elegant.

The syntax for `&&` is,

```
if ((condition1) && (condition2))
   {
   statements-s1;
   }
```

`Statements-s1` will be executed only if both the conditions are true.

The syntax for 'OR' is as follows:

```
if ((condition 1 ) ||(condition 2))
   {
   statements-s2;
   }
```

In this case, even if one of the conditions is true, the `statements-s2` will be executed. At least one condition has to be true for the execution of `s2`. However, if both are false, `s2` will not be executed.

The NOT operator with symbol `!` can be used along with any other relational or logical operator or as a stand-alone operator. It simply negates the operator following it. The syntax of `'!'` is as follows:

```
if ! (condition) statement s3;
```

`s3` will be executed only when the condition is not true or the condition is false.

Given below is the Algorithm 1 using the logical operators.

Algorithm 1

The revised Algorithm to find the greatest of 3 integers

Step 1: Print a message to enter 3 integers.

Step 2: Get three numbers and store them at `&x`, `&y` and `&z`.

Step 3: If (x > y) and (x > z), x is the greatest.

Step 4: Else if (x<y) and (y>z), y is the greatest.

Step 5: Else print z is the greatest.

The complete program is given in Example 1.6.

```
/*Example 1.6*/
*/ This Example demonstrates the use of logical operators*/
#include <stdio.h>
main()
{
    int x,y,z;
    printf ("Enter three unequal integers\n");
    scanf("%d%d%d", &x, &y, &z);
    if ((x>y) && (x>z))
        printf("x is greatest\n");
    else
    {
        if((x<y) && (y>z))
            printf("y is greatest\n");
        else
            printf("z is greatest\n");
    }
}
```

The output of the program is:

```
Enter three unequal integers
12 23 78
z is greatest
```

Now write a program to convert a lower case letter typed into an upper case letter. For this purpose you may have to refer to the ASCII table in Annexure 1.

It is obvious that if you subtract 32 from the ASCII value of a lower case alphabet, you will get the ASCII value of the corresponding upper case letter. Now write an algorithm for the conversion of lower case to an upper case letter. It is given in Algorithm 2.

Algorithm 2

Algorithm for the conversion of lower case to an upper case letter.

Step 1: Send a message for getting a character

Step 2: Get a character

Step 3: Check whether the character typed is >=a and <=z

(This is essential since you can only convert a lower case alphabet into upper case.)

Step 4: If so, subtract 32 from the value of the character; if not, go to Step 6

Step 5: Output the character with the revised ASCII value; END

Step 6: Print 'an invalid character' END

The algorithm is implemented in Example 1.7.

/*Example 1.7*/

```
/* Conversion of lower case letter to upper case*/
#include <stdio.h>
main()
{
    char alpha;
    printf ("Enter lower case alphabet\n");
    alpha=getchar();
    if (( alpha >='a')&& (alpha<='z'))
    {
        alpha=  (alpha-32);
        putchar (alpha);
    }
    else
        printf("Invalid entry; retry");
}
```

Now you can test the program by giving both the valid and invalid inputs; valid inputs are the lower case letters and invalid inputs are all other characters.

The output of the program is:

The result for the invalid input is as follows:

```
Enter lower case alphabet
8
Invalid entry; retry
```

The result when tried with a valid input is given below:

```
Enter lower case alphabet
n
N
```

The programs should be executed, i.e., tested with both the valid and invalid inputs.

Check Your Progress

6. Name the types of constants.

7. Name the two parts of real numbers.

1.5 VARIABLES

The names of variables and constants are identifiers. The names are made up of characters, digits and underscore, but the first character of an identifier must be an alphabetic character. C allows up to 31 characters for the identifier (names) and

therefore, the naming of the variables should be carried out in an easily understandable manner. For example, in the program for the calculation of,

```
Simple interest, I = (p×n×r)/100,
```

you can declare them with actual names,

```
p = principal, r = rate_of_interest, n =
number_of_years
```

Naturally, programmers may not like typing long names for fear of making mistakes elsewhere in the program apart from being reluctant to increase their typing workload. Meaningful names can, however, be assigned to data types even with few letters. For example,

```
 p = princ; r = intrate; n = years
```

Some compilers may allow names with up to 31 (thirty-one) characters, but may consider the first eight characters for all purposes. Hence, a programmer could coin shorter yet meaningful names, instead of using single alphabets for variable names. One should, however, be careful not to use the reserved words, such as 32 keywords, for variable names as they have a specific meaning to the compiler. If they are used as variable names, then the compiler will get confused. Be careful not to use the reserved words as identifiers.

A program to find out the square of integer 5 is given as follows:

```
/*Example 1.8*/
/*Program to find square of 5*/
#include <stdio.h>
int main()
{
printf("square of %d= %d", 5, 5*5);
return.;
}
```

Result of the program

```
square of 5= 25
```

You have now achieved the objective of finding the square of 5. Later on, you may want to find out the square of another number, say 7, for example. We would have to write the same program again replacing 5 by 7 and then compile and run it. This would waste a lot of time. To save time, we can, therefore, write a general-purpose program as given below:

```
/*Example 1.9*/
/*Program to find square of any given number*/
#include <stdio.h>
int main()
{
int num;
```

```
printf("Enter the integer whose square is to be found\n");
scanf("%d", &num);
printf("square of %d= %d", num, num*num);
return.;
}
```

Here, we define `num` as an integer variable. When '`&`' precedes `num`, it indicates the memory address of `num`.

At the first `printf`, the message appears as it is and the cursor goes to the next line because of the new line character `\n` at the end of the message, but before the closing quotation mark. The next statement is new to you. It is used to receive an integer typed on the console. You can type in any integer number, and the number typed will be stored in the memory at the memory location named '`num`'. The purpose of the statement is, therefore, to get the integer (because of the appearance of `%d` within quotes) and it is stored at the memory address '`num`'.

The next statement prints the number typed and its square.

When you execute the program, the following message appears on the monitor:

Enter the integer whose square is to be found.

Since we want to find out the square of 25 type:

```
25
```

Promptly, the reply will be as shown as follows:

```
 square of 25 = 625
```

The next time you may want to find out the square of another number, say 121. Then simply run the program and when prompted, type 121 to get the answer.

Here the number whose square has to be found out has been declared as a variable. The variable has a name and is stored at the location pointed to by the variable name. Therefore, the execution of the program for finding out the square of any number is easy with the above modification as in Example 1.9.

Variables and constants are fundamental data types. A variable can be assigned only one value at a time, but can change value during program execution. A constant, as the name indicates, cannot be assigned a different value during program execution. For example, `PI`, if declared as a constant, cannot have its value varied in a given program. If `PI` has been declared as a constant = 3.14, it cannot be reassigned any other value in the program. Programs may declare a number of constants. Variables are similarly useful for any programming language. If `PI` has been declared as a variable, then it can be changed in the program to any value. This is one difference between a variable and a constant. Whether an identifier is constant or variable depends on how it is declared. Both variables and constants belong to one of the data types like `int, float`, etc. The convention in 'C' is to indicate the names of constants by the upper case letters.

```
PI
SIGMA
```

Variable names are, on the other hand, indicated by the lower case letters.

```
int a
float xy
```

Size of Variables

The C programmer should understand how much memory storage each variable type occupies in the IDE used by him. The following example will help us to find the size of each variable type. The result will be in terms of bytes occupied by the variable type. A new keyword sizeof is used to find out the size. The syntax for using the keyword is as follows:

```
sizeof (<data type>)
```

or

```
sizeof (<expression>)
```

Consider the following example:

```
/*Example 1.10*/
/*Program to find out the sizes of various types of
integers*/
#include <stdio.h>
int main()
{
printf("size of char =%d\n", sizeof(char));
printf("size of short=%d\n", sizeof(short));
printf("size of int =%d\n", sizeof(int));
printf("size of unsigned int=%d\n", sizeof(unsigned));
printf("size of long int=%d\n", sizeof(long));
printf("size of unsigned long int=%d\n", sizeof(unsigned
long));
printf("size of float =%d\n", sizeof(float));
printf("size of double=%d\n", sizeof(double));
printf("size of long double%d\n", sizeof(long double));
return 0;
}
```

Result of the program

```
size of char = 1
size of short = 2
size of int = 2
size of unsigned int = 2
size of long int = 4
size of unsigned long int = 4
size of float = 4
size of double = 8
size of long double = 10
```

Therefore, it is obvious that a long double occupies 10 bytes and stores long floating point numbers with double precision.

Note that the size of `short int` will be either equal to or less than the size of an integer variable.

Variables, which require more than 1 byte for storage, will be stored consecutively in the memory.

1.6 OPERATORS AND EXPRESSIONS

The symbols which represent various computations (such as addition, subtraction, etc.) performed on various variables or constants are known as **operators**. The variables and constants on which the operators act are known as **operands**. For example, in `a + b`, `a` and `b` are operands and `+` is an operator. Note that to perform an operation, operators and operands are combined together forming an **expression**. For example, to perform an addition operation on operands `a` and `b`, the addition (+) operator is combined with the operands `a` and `b` forming an expression.

The points that should be noted about expressions are as follows:

- A single variable or constant is also considered as an expression.
- Two expressions connected by an operator also form an expression.
- Two operands in an expression should not occur in continuation.
- Two operators in an expression should not occur in continuation.

An expression consisting of more than one operator leads to a problem as to which operator is to be evaluated first. For example, consider this expression.

```
a + b * c - d
```

In this expression, the compiler needs to know which operator is evaluated first. For this, it is important to determine the precedence and associativity of operators as follows:

- **Precedence:** The order or priority in which various operators in an expression are evaluated is known as **precedence**. Every operator in C has a precedence associated with it. The operators with a higher precedence are evaluated before the operators with a lower precedence. For example, multiplication is performed before addition as the multiplication operator has a higher precedence than the addition operator.

- **Associativity:** The order or priority in which operators of the same precedence are evaluated is known as **associativity**. For example, the addition and subtraction operators have the same precedence. However, addition or subtraction may be performed on the expression depending upon the order of their occurrence. The associativity of an operator can be either from the left to the right or from the right to the left. The operators with the left to right associativity are evaluated from the left hand side while the operators with the right to left associativity are evaluated from the right hand side.

Depending on the function performed, the C operators can be classified into various categories. This includes *arithmetic operators*, *relational operators*, *logical operators*, *conditional operator assignment operators bitwise operators and other operators* These categories are further classified into *unary operators*, *binary operators* and *ternary operators* (Figure 1.5).
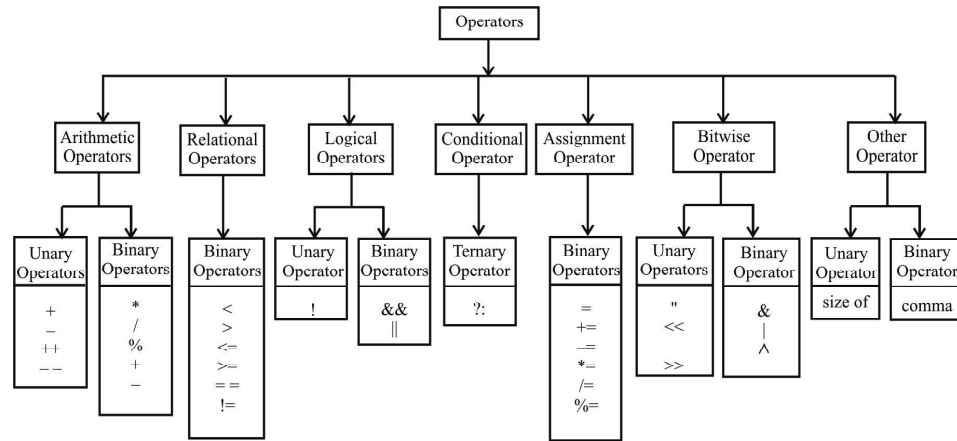
Fig. 1.5 *Types of Operators*

1.6.1 Arithmetic Operators

Arithmetic operators perform the basic arithmetic operations on operands. Arithmetic operators work on integer, floating-point and character data types. As characters are internally represented as ASCII codes (integers), thus arithmetic operators perform the arithmetic operations on the ASCII values of the characters and produce the result accordingly.

The various arithmetic operators used in C are the unary plus '+', unary minus '–', addition '+', subtraction '–', multiplication '*', division '/' and modulus '%' operators.

Note that the modulus operator is also known as the remainder operator and works only on integer and character data types.

1.6.2 Increment and Decrement Operators

C provides two special unary arithmetic operators known as the **increment operator** (represented by ++) and the **decrement operator** (represented by—).

To understand the importance of the increment and decrement operators, consider example 1.11.

Example 1.11: Evaluate these statements

```
x = 5;
y = x+1;
z = ++x;
```

In this example, the variable x is assigned the value 5. In the statement y = x+1, the value of x is incremented by 1 using the binary addition operator

and the value is stored in the variable y. However, the value of x remains 5, that is, inspite of adding 1 to x, the value of x remains the same.

To change the value of x, the increment operator is used. That is, in the statement z = ++x, the value of x is incremented by 1 using the prefix increment operator and then stored in the variable z. Similarly, the decrement operator can be used to decrement the value of x by 1.

The increment and decrement operators can be used in two forms which are as follows:

- **Prefix form:** In this form, the increment or the decrement operator precedes its operand. The prefix increment operator is represented as ++operand and the prefix decrement operator is represented as – operand. The prefix increment or the prefix decrement operator increments or decrements the value of an operand respectively before its value is used in an expression.

  To understand prefix operators, consider example 1.12.

  Example 1.12: Evaluate these statements

  ```
  x = 10;
  y = ++x;
  z = –x;
  ```

  In this example, the statement y = ++x uses the prefix increment operator to first increment the value of x by 1 and then assign the incremented value to y. Similarly, the decrement operator is used to decrement the value of x by 1 and then assign the decremented value to z.

- **Postfix form:** In this form, the increment or the decrement operator succeeds its operand. The postfix increment operator is represented as operand++, and the postfix decrement operator is represented as operand–. The postfix increment or the decrement operator increments or decrements the value of an operand respectively *after* using it in the expression.

  To understand postfix operators, consider example 1.13.

  Example 1.13: Evaluate these statements

  ```
  x = 10;
  y = x++;
  z = x–;
  ```

  In this example, the statement y = x++ first assigns the value of x to y and then increments the value of x by 1. Similarly, the decrement operator is used to assign the value of x to z and then decrement the value of x by 1.

1.6.3 Logical Operators

Logical operators combine expressions and then return True or False where True represents 1 and False represents 0.

The various logical operators provided by C are as follows:

- **AND (&&) operator**: It returns True only if all the expressions evaluate to True, otherwise it returns False.

- **OR (||) operator**: It returns True if any one or all the expressions evaluate to True and returns False only if all the expressions evaluate to False.

- **Negation (!) operator**: It returns True if the expression on which it is operating is False and vice versa.

1.6.4 Relational Operators

Relational operators compare two values or expressions and then return True or False where True represents 1 and False represents 0. All relational operators can work on integer, floating-point and character data types.

The various relational operators provided by C are less than '<', less than or equal to '<=', greater than '>', greater than or equal to '>=', equal to '==' and not equal to '! =' operator.

The operators == and ! = are also known as equality operators as they are used for checking the equality of operands. However, the equality operators should not be used with floating-point operands. This is because arithmetic operations performed on floating-point numbers are not exact and accurate as compared to arithmetic operations performed on integer numbers.

1.6.5 Conditional Operator

The conditional operator selects a value based on some condition. Note that the conditional operator is a ternary operator, that is, this operator involves three operands.

The syntax of the conditional operator is given here.

```
expression1 ? expression2 : expression3
```

If expression1 is True, then expression2 is evaluated, otherwise expression3 is evaluated.

To understand the conditional operator, consider example 1.14.

Example 1.14: Evaluate this statement

```
(x==5) ? 8 : 9;
```

In this example, if the value of x is equal to 5, then the expression returns 8, otherwise the expression returns 9.

1.6.6 Assignment Operator

Assignment operators assign values to variables. Assignment operators are of two types, namely, the *simple assignment operator* and *compound assignment operators*.

(i) Simple Assignment Operator

The simple assignment operator assigns the value on its right hand side to the variable on its left hand side. Note that the left hand side of an assignment expression

should be a variable. It cannot be a constant or an expression. However, the right hand side of an assignment expression can be a variable, a constant or an expression.

To understand the simple assignment operator, consider example 1.15.

Example 1.15: Evaluate this statement

```
x = 8;
```

In this example, the value 8 is assigned to the variable x. With the help of the assignment operator, several variables can be assigned a common value. This is accomplished by using *multiple assignments* in a single statement. For example, in the statement x=y=z=5, the value 5 is assigned to the three variables x, y and z.

(ii) Compound Assignment Operators

C provides compound assignment operators that can be used to simplify the coding process of writing certain type of assignment statement and other binary operators. For example, the expression x=x+6 can be written as x+=6. In this expression, x is incremented by 6 and then the result is assigned to x. The various compound assignment operators used in C are '+=', '-=', '*=', '/=' and '%='.

1.6.7 Bitwise Operators

The bitwise operator is used to perform bit manipulation on data. The various bit-manipulation operators provided by C are '&' bitwise AND, '|' bitwise OR, '^' bitwise exclusive OR, '~' bitwise complement, '>>' shift left, and '<<' shift right. These operators are used to tests the bits, or shift them right or left.

1.6.8 Special Operators

In addition to the operators discussed, there are some special operators used for performing particular tasks. This includes *sizeof operator* and *comma operator*.

The size of Operator: The sizeof operator returns a value (treated as integer) that determines the amount of memory occupied by an operand or a data type in the memory.

The syntax of the sizeof operator for data types is

```
sizeof(type)
```

The syntax of the sizeof operator for an operand is

```
 sizeof var
```

Note that type is enclosed in parentheses, whereas var may or may not be enclosed in parentheses.

To understand the sizeof operator, consider example 1.16.

Example 1.16: Evaluate these statements

```
sizeof(int);
float i;
sizeof i;
```

In this example, the statement sizeof(int) returns the size of the int data type and the statement sizeof i returns the size of the variable i.

Comma Operator: The comma operator is used for combining two or more expressions into a single expression. All the expressions, except the right-most expression, are always evaluated as void. The value of the right most expression forms the result of the combined expression. For example, in the statement, x = (1+2, 3+4), the expression 1+2 is evaluated as void, while the output of the expression 3+4 is assigned to x.

1.6.9 Operators and Associativity

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate value to the proper type so that the expression can be evaluated without losing any significance. Each operator in C has a precedence associated with it. The **precedence** is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to any one of these levels. The operators of higher precedence are evaluated first. The operators of same precedence are evaluated from right to left or from left to right depending on the level. This is known as associativity property of an operator. Operator precedence describes the order in which C reads expressions. For example, the expression a=4+b*2 contains two operations, an addition and a multiplication. Questions are raised like does the C compiler evaluate 4+b first, then multiply the result by 2 or does it evaluate b*2 first, then add 4 to the result? The given operator precedence chart will help you to get the correct answers. Operators higher in the chart have a higher precedence, meaning that the C compiler evaluates them first. Operators on the same line in the chart have the same precedence and the associativity column on the right gives their evaluation order. Two operator characteristics determine how operands group with operators: precedence and associativity. Precedence is the priority for grouping different types of operators with their operands. **Associativity** is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses. For example, in the following statements, the value of 5 is assigned to both a and b because of the right-to-left associativity of the = operator. The value of c is assigned to b first and then the value of b is assigned to a.

```
b = 9;
c = 5;
a = b = c;
```

The above statements only assign value to the expression. To evaluate these expressions we have to use arithmetic operators. The * and / operations are performed before + because of precedence. The variable b is multiplied by c before it is divided by d because of associativity rule. The order of precedence and associativity is given in the given table that C uses for evaluating declared expressions.

```
a + b * c / d
```

You can explicitly force the grouping of operands with operators by using parentheses to specify the order of precedence.

Table 3.3 *Operators Precedence and Associativity*

Operator Name	Associativity	Operators
Primary	left to right	`()   [ ]   .  ->`
Unary	right to left	`++   --   +   -` `!  ~  &  *  (type_name)  sizeof`
Multiplicative	left to right	`*  /  %`
Additive	left to right	`+  -`
Bitwise Shift	left to right	`<<  >>`
Relational	left to right	`<  >  <=  >=`
Equality	left to right	`==  !=`
Bitwise AND	left to right	`&`
Bitwise Exclusive OR	left to right	`^`
Bitwise Inclusive OR	left to right	`|`
Logical AND	left to right	`&&`
Logical OR	left to right	`||`
Conditional	right to left	`?  :`
Assignment	right to left	`=  +=  -=  *=   /=  <<=  >>=` `%=   &=  ^=  |=`
Comma	left to right	`,`

1.6.10 Type Modifiers

To summarize, there are four basic data types as given below:

```
char      int         float        void
```

The basic data types could be modified and a summary of all modified data types is given below:

Basic data type	Modified	No. of bytes occupied
`char`	`signed char`	1
	`unsigned char`	1
`int`	`short`	`<=2`
	`unsigned int`	2
	`long`	4
	`long unsigned`	4
`float`	`double`	8
	`long double`	10

We have discussed these modifiers and how their range gets affected by modifying the basic data types.

1.6.11 Type Definitions Using `Typedf`

The keyword `typedef` is used to create a new data type name. It does not really create a new type but only gives a new name to an existing type.

For example,

```
typedef   long double    LD;
                  LD x , y;
```

Here, `long double` is given a new name `LD`. Wherever `LD` appears, the compiler will take it as a `long double`. For example,

```
typedef void (FN) (int , int);
```

Here, `FN` represents a function with two integer arguments and returns nothing. This can be used for any other function having the same property, i.e., it passes two integer arguments and nothing is returned from the function. Later on, in the same program you may declare,

```
FN   f1 , f2;
```

This serves as a declarator for `f1` and `f2`, which are of the same type. The program below explains the concept.

```
/*Example 1.17*/
Program to demonstrate calling
muliple functions*/
#include <stdio.h>
int main()
{
    long nummul=0;
    long num=0, rev=0, add_digit=0;
        /*good practice to inialize all variables*/
    typedef long FN(long );
    FN reverse; /*see simplification*/
    FN mult;
    FN sum_digit;
    printf("enter unsigned number\n");
    scanf("%lu", &num);
    if (num%2)  /*remainder 1*/
    {
        rev =reverse(num);
        printf("number is odd\n");
        printf("number    entered=%lu\n   number
reversed=%lu\n", num, rev);
    }
    else
    {
        nummul=mult(num);
```

```
        printf("number is even\n");
        printf("number=%lu\n its multiple=%lu\n", num,
    nummul);
        }
        if (num%3 ==0)
        {
            add_digit= sum_digit(num);
            printf("number evenly divisible by 3\n");
            printf("sum of digits =%lu", add_digit);
        }
        return 0;
    }
    long reverse(long n)
    {
        long r=0;
        while (n>0)
        {
            r=r*10+(n%10);
            n=n/10;
        }
        return r;
    }
    long mult(long p)
    {
        long sq;
        sq=2*p;
        return sq;
    }
    long sum_digit(long num)
    {
        long sum=0;
        while (num >0)
        {
            sum=sum+(num%10);
            num=num/10;
        }
        return sum;
    }
```

Result of the program

```
enter unsigned number
234
number is even
```

```
number=234
its multiple=468
number evenly divisible by 3
sum of digits =9
```

In the `union guardian` declaration we can add the following :

```
typedef    union    guardian    UN;
  UN    u1,  u2;
```

Here, `UN` will be substituted by `union guardian` at the time of compilation. Thus, the usage of `typedef` helps in reducing typing and considered to improve readability. This can be used to make short-hand notations for the existing data types as well as user-defined variables, such as structures.

Check Your Progress

8. Define operators and operands.

9. What do arithmetic operators perform?

10. Name the types of assignment operators.

11. What is the use of `typedef`?

1.7 CONDITIONAL STATEMENTS

Real-life application programs do not merely consist of simple multiplication or addition. They call for solving complex problems. Depending on the occurrence of a particular situation, we may follow different paths; the `if` and `else` keywords are quite handy in branching to different segments of the program. 'C' is ideal for handling branching because the syntax is clear and unambiguous. We will now discuss the branching constructs. Relational operators are used in conjunction with branching constructs. Hence, we look at them first.

`If` Statement

The syntax of the `if` statement is given below.

```
if (condition)
    {statements}
```

If the condition is true, then a single statement or group of statements following the `if` will be executed. If more than one statement is to be executed, then the statements are grouped within braces. If it is a single statement, then curly braces are not required.

If the condition turns out to be false, then the next statement after those belonging to the `if` will be executed. Example 1.18 will make the concept clear.

Input two integers from the keyboard. If they are equal, then the program will print, 'you typed equal numbers'; otherwise, it will print nothing.

```
/*Example 1.18
To demonstrate the use of if*/
```

```
#include<stdio.h>
main()
{
unsigned int a,b;
printf ("enter two integers\n");
scanf("%u%u", &a, &b);
if (a==b)
    {
    printf("you typed equal numbers\n");
    }
}
```

Each open curly brace has to have a matching closing curly brace. The first closing curly brace corresponds to the `if` statement and the second one to the main function. Execute the program by first keying in two equal valued unsigned integers.

Result of the program

```
enter two integers
56 56
you typed equal numbers
```

After you are satisfied, you can try the program with unequal numbers. You will not get any message.

If...else Statement

We did not get any message when the numbers were unequal and this can be avoided by using the else statement.

The usage of `if .. else` is shown below.

```
if(condition true)
{
    statements s1
}
else
{
    statements s2
}
statements s3;
```

The statement `else` is always associated with an `if`.

If the condition is true, then statements `s1` will be executed. After executing them, the program will skip the `else` block and control goes to statement `s3` that follows the `else` block.

If the condition is false, then the statements in the `else` block, i.e., `s2` will be executed followed by statement `s3`.

Statements s1 will not be executed at all when the condition becomes false. The usage of braces clearly brings out which statements belong to the if block and which to the else block.

Example 1.19 brings out the usage of if... else.

```
/*Example 1.19
To demonstrate the use of if.. else*/
#include <stdio.h>
main()
{
    unsigned int a,b;
    printf ("enter two integers\n");
    scanf ("%u%u", &a, &b);
    if (a==b)
    {
        printf ("you typed equal numbers\n");
    }
    else
    {
        printf ("numbers not equal\n");
    }
}
```

The output of the program when unequal numbers were keyed in is as follows.

Result of the program

```
enter two integers
17 13
numbers not equal
```

Nesting of the `if...else` Statements

We witnessed the usage of a single if statement in Example 1.18. We saw if followed by else in Example 1.19. There is no restriction to the number of if, which can be used in a program. This applies to else as well, but else can only follow an if statement.

We can have the following in a program:

```
{
if (condition1)
    {
        if (condition2)
            {statements-s1}
        else
            if (condition3)
                {statements-s2,}
    }
```

```
else
     {statements-s4}
statements-s5
}
```

This is called a nested `if` and `else` statement. As the level of nesting increases, it will be difficult to analyse and logical mistakes will be made more easily.

In the above example, when `condition1` is false, `statements-s4` will be executed. If `condition1` is true and `condition2` is also true, then `statements-s1` will be executed.

If `condition1` is true and `condition2` and `condition3` are false, `statements-s5` will be executed directly.

To execute statements–s2

`Condition1` has to be true;

`Condition2` has to be false,

And `condition3` has to be true.

Try to analyse this yourself. There are better methods to solve the above problem, which will be discussed later.

For example, three unequal integers are keyed in and are called x, y and z. Write a program to find the greatest of the three numbers.

Before writing a program, we must write the algorithm. We should not straight away get down to programming.

Algorithm 3

Algorithm for finding the largest of 3 integers.

Step 1: Print a message to enter 3 integers.

Step 2: Get three numbers and store them at &x, &y and &z.

Step 3: Check if x > y

Step 4: If false, go to step 9

Step 5: If true

Step 6: Check if x > z

Step 7: If true, write x is the largest; End

Step 8: If false, write z is the largest; End

Step 9: Check if y>z

Step 10: If true, write y is the largest; End

Step 11: If not, write z is the largest.

End

Let us now code these steps into a 'C' program, which is shown in Example 1.20.

```
/*Example 1.20
To demonstrate the use of the nested if.. else*/
#include <stdio.h>
main()
{
    int x,y,z;
    printf ("enter three unequal integers\n");
    scanf("%d%d%d", &x, &y, &z);
    if(x>y)
    {
      if(x>z)
        {
            printf("x is largest\n");
        }
      else
        {
            printf("z is largest\n");
        }
    }
    else
    {
        if(y>z)
        {
            printf("y is largest");
        }
        else
        {
            printf("z is largest");
        }
    }
}
```

Test the correctness of the program by giving a different set of values for x, y and z.

Result of the program

```
enter three unequal integers
908 231 907
x is largest
```

Look at the example. It uses multiple nesting of if ... else.

Take care to see that every opening brace has a corresponding closing brace. It is better to indent the braces as shown in the example so that no mistake is committed. Take care to see that else matches with the corresponding if and each opening brace {matches with a corresponding closing brace}; if either an opening {or closing} is extra, then an error will result.

Logical Operators and Branching

In the above examples we have been checking one condition at a time. It would be nice if we could check more than one condition at a time. 'C' provides three logical operators for combining more than one condition. These are as follows:

Logical `and` represented as `&&`

Logical `or` represented as ||

`Negation` or `not` represented as `!` (exclamation).

Let us see some examples of usage of the logical operators. In Example 1.20 we concluded that,

if $x > y$ and if $x > z$, then x is the largest.

We will represent the same as,

```
if ((x > y) && (x > y))
printf ("x is the largest");
```

You will see that the program has become much more elegant.

The syntax for `&&` is,

```
if ((condition1) && (condition2))
    {
    statements-s1
    }
```

Statements–s1 will be executed only if both the conditions are true.

The syntax for 'or' is as follows:

```
if ((condition 1 ) ||(condition 2))
    {
    statements-s2
    }
```

In this case, even if one of the conditions is true, the `statements-s2` will be executed. At least one condition has to be true for the execution of `s2`. However, if both are false, `s2` will not be executed.

The ***NOT operator*** with symbol **!** can be used along with any other relational or logical operator or as a stand-alone operator. It simply negates the operator following it. The syntax of '`!`' is as follows:

```
if ! (condition) statement s3;
```

`s3` will be executed only when the condition is not true or the condition is false.

Let us rewrite Algorithm 3 by using the logical operators. The revised Algorithm 4 is given below:

Algorithm 4

Step 1: `If` $(x > y)$ and $(x > z)$, x is the largest.

Step 2: `Else if` $(x < y)$ and $(y > z)$, y is the largest.

Step 3: `Else` print z is the greatest.

The complete program is given in Example 1.21.

```
/*Example 1.21
To demonstrate the use of logical operators*/
#include <stdio.h>
main()
{
    int x,y,z;
    printf ("enter three unequal integers\n");
    scanf("%d%d%d", &x, &y, &z);
    if ((x>y) && (x>z))
        printf("x is largest\n");
    else
    {
        if((x<y) && (y>z))
            printf("y is largest\n");
        else
            printf("z is largest\n");
    }
}
```

Result of the program

```
enter three unequal integers
12 23 78
z is greatest
```

Let us now write a program to convert a lower case letter typed into an upper case letter.

It is obvious that if we subtract 32 from the ASCII value of a lower case alphabet we will get the ASCII value of the corresponding upper case letter. Let us write an algorithm for the conversion of lower case to an upper case letter. It is given in Algorithm 5.

Algorithm 5

Step 1: Send a message for getting a character

Step 2: Get a character

Step 3: Check whether the character typed is >=a and <=z

(This is essential since we can only convert a lower case alphabet into upper case.)

Step 4: If so, subtract 32 from the value of the character; if not, go to step 6

Step 5: Output the character with the revised ASCII value; END

Step 6: Print "an invalid character" END

The algorithm is implemented in Example 1.22.

```
/*Example 1.22
Conversion of lower case letter to upper case*/
#include <stdio.h>
```

```
main()
{
    char alpha;
    printf ("enter lower case alphabet\n");
    alpha=getchar();
    if ((alpha >='a')&&(alpha<='z'))
    {
        alpha= (alpha-32);
        putchar (alpha);
    }
    else
        printf("invalid entry; retry");
}
```

Now you can test the program by giving both the valid and invalid inputs; valid inputs are the lower case letters and invalid inputs are all other characters.

Result of the program

The result for the invalid input is given below:

```
enter lower case alphabet
8
invalid entry; retry
```

The result when tried with a valid input is given below:

```
enter lower case alphabet
n
N
```

The programs should be executed, i.e., tested with both the valid and invalid inputs.

Conditional Operator and `if...else`

The syntax for the conditional operator is given below:

(Condition) ? statement1: statement2;

What does it mean? If the condition is true, execute `statement1`; else, execute `statement2`. Here nesting is not possible. The `if...else` statement is more readable than the `conditional(?)` operator. However, the conditional operator is quite handy in simple situations as given below:

```
(a > b) ?  print a greater
        :  print b greater;
```

Thus, the operator has to be used in simple situations. If nothing is written in the position corresponding `to else,` then it means nothing is to be done when the condition is false.

Example 1.19 is rewritten using the ? operator in Example 1.23.

/*Example 1.23

To demonstrate the use of the ? operator*/

```
#include <stdio.h>
```

```
main()
{
    unsigned int a,b;
    printf ("enter two integers\n");
    scanf("%u%u", &a, &b);
    (a==b)?printf("you typed equal numbers\n"):
    printf("numbers not equal\n");
}
```

Result of the program

```
enter two integers
123 456
numbers not equal
```

1.5   SWITCH STATEMENT

Switch statements allow clear and easy implementation of multiway decision-making. Assuming that a number is received from the keyboard and that depending on the value, we want to carry out some operations, the switch statement can be used effectively in this situation. In simpler situations if...else could be used, and in complex situations, switch can be used. For example, if we get numbers starting from 1 to 4 and print their values in words, we can use the if...else statement as given in Example 1.24:

```
/*Example 1.24
Converts the digits 1-4 in words using if*/
#include<stdio.h>
#include<conio.h>
main()
{
    int a;
    char ch='c';
    while (ch=='c')
        {
        printf("\nEnter a digit 1 to 4\n");
        scanf("%d",&a);
        if(a==1)
            printf("One\n");
        else
            if (a==2)
                printf("Two\n");
            else
                if(a==3)
                    printf("Three\n");
                else
                    if(a==4)
```

```
                        printf("Four\n");
                else
        printf("Illegal character\n");
        printf("enter 'c' if you want to continue\n");
        printf("or any other character to end\n");
                             ch=getche();
                             if(ch!='c')
                                  printf("End     of
    Session");
            }
    }
```

Result of the program

```
Enter a digit 1 to 4

3

Three

enter 'c' if you want to continue

or any other character to end

c

Enter a digit 1 to 4

1

One

enter 'c' if you want to continue

or any other character to end

c

Enter a digit 1 to 4

2

Two

enter 'c' if you want to continue

or any other character to end

nEnd of Session
```

We have struggled hard to do this exercise. Assuming that we want to get a one digit number up to 9 and print its value in words, it would become a more complex task. The switch statement comes in handy in such situations. The syntax of the switch statement is as follows:

```
    switch (expression)
    {
        case  constant or expression : statements
        case  constant or expression : statements
        ..
     default : statements }
```

When the **switch** keyword is encountered, the associated expression is evaluated. The program now looks for the **case**, which matches with the value of the expression. Execution then starts from the statement corresponding to the

case which matches. Each **case** has to be accompanied by integer expressions, which must be unique, as otherwise the program will not know where to start. For example, if the first two cases are as given below:

```
case 10 : s1;
case 10 : s2;
```

In this case the program would not know whether to execute s1 or s2 when the expression of switch evaluates to 10. Therefore, the constant expressions following the case keyword should all be unique. There may be occasions when none of the constant expressions matches the switch expression in which case the default statements will be executed. Thus switch allows branching of the program execution to appropriate place. A program to print the values of the digits in words is given below:

```
/*Example 1.25
converts the digits 0-9 in words*/
#include <stdio.h>
#include <conio.h>
main()
{
    int a;
    char ch ='c';
    while (ch=='c')
        {
            printf("\nEnter a digit 0 t0 9\n");
            scanf("%d",&a);
            switch(a)
            {
                case 0:printf("Zero\n");
                    break;
                case 1:printf("One\n");
                    break;
                case 2:printf("Two\n");
                    break;
                case 3:printf("Three\n");
                    break;
                case 4:printf("Four\n");
                    break;
                case 5:printf("Five\n");
                    break;
                case 6:printf("Six\n");
                    break;
                case 7:printf("Seven\n");
                    break;
                case 8:printf("Eight\n");
                    break;
```

```
            case 9:printf("Nine\n");
                break;
            default:printf("Illegal character\n");
        }
        printf("enter 'c' if you want to continue\n");
        printf("or any other character to end\n");
        ch=getche();
        if (ch!='c')
            printf("End of Session");
    }
}
```

Result of the program

```
Enter a digit 0 t0 9
6
Six
enter 'c' if you want to continue
or any other character to end
c
Enter a digit 0 t0 9
7
Seven
enter 'c' if you want to continue
or any other character to end
nEnd of Session
```

The do...while concept has been used in this example also. The program first enters the do loop. After the execution of the loop, the program asks you to enter c if you want to continue or any other character to end the program. If you enter c, the program will continue. Thus, if you want to exit, you can press any other letter, say 'n'. If you press 'n' the program stops instantly. This is the right way of using the do...while loop.

Now, the switch is analysed. The program asks for entry of a digit 0 to 9. The entered digit is stored in variable a. If a = 5, then the program goes to case 5. It is followed by printing the value Five and then break. If you press 8, then case 8 matches and Eight will be printed.

What is a break statement ?

Assume that all the break statements are removed from the program. Then if you press 1, 'One' will be printed and all the statements following that will be executed. This means that Two, Three, ... till Nine will be printed. If you enter 7, it will print all the numbers starting from Seven, which in not desirable. The break statement has, therefore, been introduced. After printing the value of the number, the break statement takes the program to the end of the switch statement. The end is just the closing brace corresponding to switch after the default printf() statement. Therefore, the combination of switch and break does the trick.

Assuming that a character other than 0 to 9 is entered, none of the cases match. Therefore, default is executed. The program will print 'illegal character'. The **default** is optional and even in its absence, when no match is found, the program will come out of the switch statement without any action.

The `case` statements need not be in any specified order. The default can be on top before `case` 0 and the `case` can occur in any order. The program is made to execute, as long as you want, by the `do` statement. If `do` is absent, then the program will be executed only once. The `do...while` is necessary to make it an iterative program.

Every `switch` statement, therefore, contains a condition in the form of an expression. The expression could also be a single variable as in this case. The expression will be evaluated at the time of program execution and must be an integer. Then, depending on the value it goes to a `case` label, which is like a label in a `goto` statement. The label should match with the value of the expression.

Unless the program is made to exit by statements such as `break` after executing the group of statements corresponding to a particular case, the program will execute all the statements in the program from then on. This should be noted. Therefore, the programmer has to specify where to end. In the case of `if...else`, where to begin and where to end is clear as also in the case of `switch`. The program starts at the beginning of the `case` that meets the condition, but ends at the bottom of the `switch` unless otherwise specified. It will also execute the statements following default. It will of course not bother about the keywords. That is the reason for the `break` statement, since we do not want the program to execute irrelevant statements.

It is a good programming practice to include a break statement after the default as well. If this is not done at the inception, at a later stage when more case statements are added after default, this would lead to problems. Whenever default is executed all the statements following it, even if they belong to some other case, will also be executed if `break` is not included after default.

Now the program can be extended to print the value of the number up to 99 in words. This makes it more complicated since the words are unique up to nineteen. A program is given below for achieving the task. This is made to loop using `do...while`.

```
/*Example 1.26
to print out in words the value
of a number typed in the range 1-99*/
#include <stdio.h>
#include<conio.h>
main()
{
    int num,m,ch='y';
    do
    {
        printf("Type a number 1 to 99\n");
        scanf("%d",&num);
```

```
    if ((num>0)&&(num<100))

    /*only if the number is within the valid range 0 to 99
the following will be executed*/

    {
        if (num>=20)
        {
            m=num/10;
            switch(m)
            {
                case 2:printf("TWENTY ");
                    break;
                case 3:printf("THIRTY ");
                    break;
                case 4:printf("FORTY ");
                    break;
                case 5:printf("FIFTY ");
                    break;
                case 6:printf("SIXTY ");
                    break;
                case 7:printf("SEVENTY ");
                    break;
                case 8:printf("EIGHTY ");
                    break;
                case 9:printf("NINETY ");
                    break;
            }
        }
        if (num>20)
            num= num%10;
            switch(num)
            {
                case 1:printf("ONE\n");
                    break;
                case 2:printf("TWO\n");
                    break;
                case 3:printf("THREE\n");
                    break;
                case 4:printf("FOUR\n");
                    break;
                case 5:printf("FIVE\n");
                    break;
                case 6:printf("SIX\n");
                    break;
```

```
                        case 7:printf("SEVEN\n");
                                break;
                        case 8:printf("EIGHT\n");
                                break;
                        case 9:printf("NINE\n");
                                break;
                        case 10:printf("TEN\n");
                                break;
                        case 11:printf("ELEVEN\n");
                                break;
                        case 12:printf("TWELVE\n");
                                break;
                        case 13:printf("THIRTEEN\n");
                                break;
                        case 14:printf("FOURTEEN\n");
                                break;
                        case 15:printf("FIFTEEN\n");
                                break;
                        case 16:printf("SIXTEEN\n");
                                break;
                        case 17:printf("SEVENTEEN\n");
                                break;
                        case 18:printf("EIGHTEEN\n");
                                break;
                        case 19:printf("NINETEEN\n");
                                break;
                    }
                }
                else
                    printf("number outside range\n");
                    printf("\nenter y if you want to continue\n");
                    ch=getche();
                if (ch!='y')
                    printf("End of session");
                }
            while (ch=='y');
        }
```

Result of the program

```
Type a number 1 to 99
123
number outside range
enter y if you want to continue
yType a number 1 to 99
```

```
78
SEVENTY EIGHT
enter y if you want to continue
yType a number 1 to 99
6
SIX
enter y if you want to continue
nEnd of session
```

How does it work ?

The entered number is checked. If it is 20 or above, the following action takes place. For example, assume that a number 64 is entered. It is $>= 20$.

Therefore, m = num/10 will give m as 6. Hence, SIXTY will be printed (with a space at the end), corresponding to case 6.

Now since num > 20, the second if condition will be true.

Num = num % 10 will assign the remainder equal to 4 to num. Therefore, case 4 in the second switch will be true and four will be printed.

Assume you have expressed your desire to continue and enter 16.

Since num $>= 20$ will be false and switch (m) will be ignored. Hence, switch (num) will be executed. Case 16 matches with switch (16). Therefore, SIXTEEN will be printed. The correctness of the program can be checked for any number in the range of 0 – 99.

The expression associated with the switch can be of type char since char can also be considered an integer. For example, the switch statement can be of the form,

```
char m ;
switch (m)
{
    case 'a' : s1 ;
        break ;
    case 'b' : s2 ;
        break ;
}
```

If m evaluates to 'a', s1 will be executed and if it is 'b', s2 will be executed. Remember the label of case has to be a constant expression.

Other characters such as +, –, etc. can be used as case labels since their integer values are known from the ASCII table.

```
Int op ;
switch (op)
{
    case '+' : s1; break;
    case '/' : s2 ; break;
}
```

Here when `op` is +, `s1` will be executed and when it is /, `s2` will be executed.

1.9 CONTROL PROGRAM

The keywords `while`, `for` and `switch` test the condition on top, while `do...while` checks at the bottom for quitting the loop. The `break` statement helps immediate exit from any part of the loop as demonstrated with the `switch` statement. It can be used with any other loop construct or anywhere in the program. When the `break` statement is executed it goes to the bottom of the block. Recall that a block is a group of statements enclosed between an opening brace and the corresponding closing brace.

The **continue statement** is related to `break`. When `continue` is executed, it causes the next iteration of the corresponding `for, do...while` or `while` loop to begin. Therefore, `continue` takes the program to the top of the block and in the `for` loop, it will cause the next increment operation, followed by checking whether the condition is true or false in order to decide the next course of action. This is similar to skipping the current execution and continuing with the next operation after incrementing. The statement `continue` skips the rest of the statements in the loop for that iteration, whereas `break` terminates the loop.

Write a program to check whether a given number is positive or negative. If it is zero, the program should terminate after printing the value. If it is a positive integer above zero and <=20000, the value will be printed; if negative, it will go to fetch the next number. If the number is >20000, the program terminates. The program is given below:

```c
/*Example 1.27
/*program to demonstrate continue*/
#include <stdio.h>
main()
{
    int a;
    do
    {
        printf("enter a number-enter 0 to end session\n");
        scanf("%d", &a);
        if(a > 20000)
        {
            printf("you entered a high value-going out of
range\n");
            break;
        }
        else
        if(a>=0)
            printf("you entered %d\n", a);
```

```
        if (a < 0)
        {
            printf ("you entered a negative number\n");
        continue;
        }
    }
    while (a !=0);
    printf (" End of session\n");
}
```

Result of the program

```
enter a number-enter 0 to end session
33
you entered 33
enter a number-enter 0 to end session
-60
you entered a negative number
enter a number-enter 0 to end session
45
you entered 45
enter a number-enter 0 to end session
25000
you entered a high value-going out of range
End of session
```

If the number typed > 20000, or if it is equal to zero, the program comes out of the loop and prints "End of session". If the number is negative, $a < 0$ and hence, continue will be executed. It will go to the top of the loop. The next integer will be received. The program, therefore, terminates when $a = 0$ as well as $a > 20000$, but there is a difference. If the number entered is zero, the program checks whether $a > 20000$. Since the condition fails, it checks whether $a > = 0$ and since it is true, 0 will be printed and then the while condition is checked. The program terminates after the while condition is checked.

However, if the number entered is >20000, the loop terminates instantly without transacting any business except printing messages as given above.

The return statement can appear anywhere in a function and when it is encountered a value is returned to the called function. The return may also not return a value in statements as given below:

```
return ;
return (0) ;
```

The return statement may appear anywhere in the function and not necessarily at the end of the function. Whenever return is executed, the program returns to the function called the current function. The program returns to the place from where it called the function. Thus return is also used to suddenly exit from a function or a loop in a function.

Exit Function

The library function **exit()** causes the termination of the current program. Note that, exit() terminates the execution of the program itself, and not the block. The statement break enables coming out of the block or loop in which it is executed but exit terminates the program at whatever stage the program may be. exit is a powerful function.

Check Your Progress

12. Give the syntax of if statement.

13. Why is goto not used in a program? What can be used instead of goto?

14. What is a break statement?

15. How are continue and break statements related?

16. Define the exit() function.

1.10 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Programs in procedural programming consist of a controlling procedure known as the **main**, which controls the execution of other procedures.

2. The program statements written in a high-level language are called source code.

3. A data type determines the value that a variable can take and the operations that can be performed on that variable.

4. The two most commonly used composite data types in C are arrays and pointers.

5. Pointers are variables that store the memory address of another variable.

6. The following are the types of constants:
   - Integer constant
   - Character constant
   - Float constant
   - Enumeration constant
   - String constant
   - Symbolic constant

7. There are two parts in the scientific notation of a real number, which are as follows:
   - Mantissa (before e)
   - Exponent (after e)

8. The symbols which represent various computations (such as addition, subtraction, etc.) performed on various variables or constants are known as **operators**. The variables and constants on which the operators act are known as **operands**.

9. Arithmetic operators perform the basic arithmetic operations on operands. Arithmetic operators work on integer, floating-point and character data types.

10. Assignment operators are of two types, namely, the simple assignment operator and compound assignment operators.

11. The keyword `typedef` is used to create a new data type name.

12. The syntax of the `if` statement is given below.

```
if (condition)
    {statements}
```

If the condition is True, then a single statement or group of statements following the `if` will be executed. If more than one statement is to be executed, then the statements are grouped within braces. If it is a single statement, then curly braces are not required.

13. Usage of `goto` is considered to be bad programming practice since it leads to errors when changes are made in the program and also affects readability. It is always possible to write a program without using `goto`. The program can be rewritten without `goto` by using a `for` statement.

14. The `break` statement takes the program to the end of the `switch` statement. The end is just the closing brace corresponding to `switch` after the default `printf()` statement.

15. The `continue` statement is related to `break`. When `continue` is executed, it causes the next iteration of the corresponding `for`, `do...while` or `while` loop to begin. Therefore, `continue` takes the program to the top of the block and in the `for` loop, it will cause the next increment operation, followed by checking whether the condition is true or false in order to decide the next course of action. This is similar to skipping the current execution and continuing with the next operation after incrementing. The statement `continue` skips the rest of the statements in the loop for that iteration, whereas `break` terminates the loop.

16. There is a library function `exit()`, which causes the termination of the current program. Note that, `exit()` terminates the execution of the program itself, and not the block. The statement `break` enables coming out of the block or loop in which it is executed but `exit()` terminates the program at whatever stage the program may be. The `exit()` is a powerful function.

1.11 SUMMARY

- Designed and developed by Brian Kernighan and Dennis Ritchie, at The Bell Research Labs in 1972, the 'C' programming language is one of the most popular computer languages in today's computer world. It was created so as to allow the programmer access to almost all of the machine's internals—registers, I/O slots and absolute addresses.

- Structured programming (also known as procedural programming) was a powerful and an easy approach of writing complex programs. In **procedural**

programming, programs are divided into different procedures (also known as functions, routines or subroutines) and each procedure contains a set of instructions that performs a specific task. This approach follows the top-down approach for designing the program.

- Programs in procedural programming consist of a controlling procedure known as the **main**, which controls the execution of other procedures.

- Ritchie along with Kernighan published a book *The C Programming Language* in 1978. The Kernighan and Ritchie (K&R) description of the language became a kind of industry standard, popularly called K&R C.

- The international standard on the C language, namely ISO/IEC 9899 was released in 1990 by adopting the ANSI standard on the C language. This standard is called C90.

- Freeware compilers for C and a host of other languages are available on the Internet from an organization called The Free Software Foundation.

- The program statements written in a high-level language are called source code. The source code is entered in a text editor. Before attempting to enter the source code, we should formulate the algorithm and document it in pseudo code.

- Tokens are similar to atomic elements or building blocks of a program. A C program is constructed using tokens.

- The C language supports and implements the American Standard Code for Information Interchange (ASCII) for representing characters. The ASCII uses 7 bits for representing each character or digit.

- Any function name is also an identifier. For instance, `printf` is the name of function available with the C language system. The function helps in printing.

- A data type determines the value that a variable can take and the operations that can be performed on that variable. C provides several primitive data types. It also provides the facility of defining new data types according to the requirements of the programmer.

- The basic (fundamental) data types provided by C are `int`, `float`, `char` and `double`. The data types `char` and `int` are collectively known as integral data types. The `char` data type occupies 1 byte of memory (that is, it holds only one character at a time).

- The `float` and `double` data types are used to store numbers with decimal point. The `float` specifies single-precision floating-point numbers and `double` specifies double-precision floating-point numbers.

- The two most commonly used composite data types in C are *arrays* and *pointers*.

- The various user-defined data types provided by C are *structures, unions* and *enumerations*.

- Structures are the collection of data items referred to by a common name. A structure provides a convenient way of keeping related information together.

- An enumeration is a set of integer constants, written as identifiers, specifying the possible values that can be assigned to the enumeration variables. These set of possible values are known as **enumerators**.

- We can use the sign bit also for holding the value. In such cases, the variable will be called `unsigned int`. The maximum value of an `unsigned int` will be equal to 65535 because we are using the Most Significant Bit (MSB) also for storing the value.

- There is another integer type called `short` or `short int`. The storage space allocation for this modified type of integer is implementation dependent. But, it will occupy the same or less space than `int` type.

- A character constant is a single character enclosed in single quotes as in `'x'`. Characters can be alphabets, digits or special symbols.

- Integers are whole numbers without decimal points but a float has always a decimal point.

- A character constant is a single character enclosed within single quotes. A string constant is a number of characters, arranged consecutively and enclosed within double quotes.

- The names of variables and constants are identifiers. The names are made up of characters, digits and underscore, but the first character of an identifier must be an alphabetic character.

- Variables and constants are fundamental data types. A variable can be assigned only one value at a time, but can change value during program execution. A constant, as the name indicates, cannot be assigned a different value during program execution.

- The symbols which represent various computations (such as addition, subtraction, etc.) performed on various variables or constants are known as **operators**. The variables and constants on which the operators act are known as **operands**.

- An expression consisting of more than one operator leads to a problem as to which operator is to be evaluated first.

- The order or priority in which operators of the same precedence are evaluated is known as **associativity**.

- Depending on the function performed, the C operators can be classified into various categories. This includes arithmetic operators, relational operators, logical operators, conditional operator assignment operators bitwise operators and other operators These categories are further classified into unary operators, binary operators and ternary operators.

- Arithmetic operators perform the basic arithmetic operations on operands. Arithmetic operators work on integer, floating-point and character data types.

- C provides two special unary arithmetic operators known as the **increment operator** (represented by ++) and the **decrement operator** (represented by –).

- The postfix increment operator is represented as `operand++`, and the postfix decrement operator is represented as `operand−`. The postfix increment or the decrement operator increments or decrements the value of an operand respectively *after* using it in the expression.

- Relational operators compare two values or expressions and then return True or False where True represents `1` and False represents `0`. All relational operators can work on integer, floating-point and character data types.

- The operators `==` and `!=` are also known as equality operators as they are used for checking the equality of operands.

- Assignment operators assign values to variables. Assignment operators are of two types, namely, the *simple assignment operator* and *compound assignment operators*.

- The keyword `typedef` is used to create a new data type name. It does not really create a new type but only gives a new name to an existing type.

- The syntax of the `if` statement is given below.

```
if (condition)
    {statements}
```

If the condition is true, then a single statement or group of statements following the `if` will be executed. If more than one statement is to be executed, then the statements are grouped within braces.

- The statement `else` is always associated with an `if`.

- The *NOT* **operator** with symbol `!` can be used along with any other relational or logical operator or as a stand-alone operator. It simply negates the operator following it.

1.13 KEY TERMS

- **Compiler:** A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.

- **Constants:** A constant is a value that cannot be altered by the program during normal execution, i.e., the value is constant.

- **Operand:** Operands are expressions or values on which an operator acts or works.

1.13 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write a short note on structured programming.

2. What do you understand by linking and loading?

3. What are tokens and keywords?

4. Write in brief about logical constants.

5. What are types modifiers?

6. What do you mean by nested `if` and `else` statement?

NOTES

Long-Answer Questions

1. What are data types? Explain with examples.

2. Describe the types of constants with examples.

3. Illustrate the types of operators.

1.14 FURTHER READING

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures.* New Delhi: Galgotia Publications, 2003.

Tanenbaum, A.M., Yedidyah Langsam and Moshe J. Augenstein. *Data Structures using C and C++*. New Delhi: Prentice-Hall of India, 1995.

UNIT 2 ARRAYS, FUNCTIONS, STRUCTURES AND POINTERS

Structure

2.0 INTRODUCTION

Data plays an important role in programming. Thus, it must be represented, organized, stored, processed and managed in a way that facilitates easy access and retrieval. This can be accomplished by using arrays. An array is a collection of data elements of similar data types. The data elements grouped in an array can be of any basic data type like integer, float or character. This unit discusses different types of arrays in detail. In this unit, you will also learn about user-defined functions, and about function declaration, function call and how to define a function. A function, when declared in a program, is a function prototype which may be declared at the beginning of a main program. A function, on execution, is supposed to do something: either return a value as integer, character or float; or perform some operation. It consists of two parts, declarator and declaration. A function declaration has the format in which the type of data returned, name of the function and arguments on which the function is to operate, are mentioned. Arguments declared as part of the function prototype are called formal parameters, which are enclosed in a pair of parentheses. A function may not contain any parameter, in which case an empty pair of parentheses should follow the name of the function. A function may not return a value, in which case `void` is written as the return data type. A function may be called directly or indirectly by another function. For this, there should be one-to-one correspondence between formal arguments declared and actual arguments sent and should be of the same data type. A function declarator is a replica of a function declaration; the difference lies in the way they are written inside a program body. A declaration in a calling function will end with a semicolon and a declarator in a called function will not end with a semicolon. The pointer is a powerful concept of C and is closely associated with memory addressing. It is an integer variable, which contains the address of a variable. These variables are stored in the memory and each location in the memory has an address. It can point to any data type. The storage capacity varies from machine to machine. You will learn how a pointer to `void` can be declared. `Void` is a keyword and it means 'nothing'. Therefore, the `void` pointer points to nothing but it is very useful when we assign address of any data type to a pointer to `void`. NULL pointers can take any pointer type, but do not point to any valid reference or memory address. You can call a function and pass actual values to them. The function call using pointers is known as call by reference where the address of the variable is passed. Functions can also return reference or pointers. The `**` indicates pointer to pointer. You will learn that array manipulation becomes easier using pointers. Both the array of numbers and array of characters, i.e., strings are explained. Static allocation of memory in case of arrays through dimension leads either to excessive allocation or short allocation. This can be managed by dynamically allocating memory at run-time through the functions, namely `malloc()` and `calloc()`. The memory allocated using these functions can be freed, when it is not required using the function `free()`. In this unit, you will also learn about structures and `union`. A structure is a user-defined data type like an array. While an array contains elements of the same data type, a structure contains members of varying data types. Thus, it is useful to represent real life examples. The structure declaration ends with a semicolon and the keyword is `struct`. The tag for

structure is optional. The structure elements can be referenced or copied individually through the dot operator. A structure can also be straightaway assigned to another structure. An array of structures is a replication of structures like an array of any other variable(s) and can be passed to a function either entirely or element wise. Pointers to structures are similar to other pointers. Structures can be passed by reference and can be nested. A structure is useful in database management and graphics and is the forerunner for classes in C++.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain array data structures
- Describe initializing and accessing single-dimensional array and multi-dimensional arrays along with their operations
- Discuss user-defined functions
- State the method of calling and declaring a function
- Discuss about storage classes and their uses
- Explain the use of recursion in writing more compact programs
- Describe the basic concept of pointers
- Analyse why a pointer is a variable
- Explain the pointer to void and use of NULL pointer
- State pointer notations for arrays and array of pointers
- Discuss dynamic allocation of memory
- Describe structure pointers and the basic concept of a structure
- Explain the process of passing of structures to function

2.2 ARRAYS

One of the data types, which can be used for storing a list of elements, is an array. Whenever programmers want to store a list of elements under a single variable name but still want to access and manipulate an individual element of the list, arrays are used. Arrays are defined as a fixed-size sequence of elements of the same data type. These elements are stored at contiguous memory locations and can be accessed sequentially or randomly. The programmer can access a particular element of an array by using one or more indices or subscripts. If only one subscript is used, an array is known as the single-dimensional array. If more than one subscript is used, an array is known as the multi-dimensional array.

2.2.1 Single-Dimensional Arrays

A single-dimensional array is defined as an array in which only one subscript value is used to access its elements. It is the simplest form of an array. Generally, a single- dimensional array is denoted as:

array_name[L:U]

where,

array_name = the name of the array

L = the lower bound of the array

U = the upper bound of the array

Before using an array in a program, it needs to be declared. The syntax of declaring a single-dimensional array in C is:

data_type array_name[size];

where,

data_type = data type of elements to be stored in array

array_name = name of the array

size = the size of the array indicating that the lower bound of the array is 0 and the upper bound is size-1. Hence, the value of the subscript ranges from 0 to size-1.

For example, in the statement int abc[5], an integer array of five elements is declared and the array elements are indexed from 0 to 4. Once the compiler reads a single-dimensional array declaration, it allocates a specific amount of memory for the array. Memory is allocated to the array at the compile-time before the program is executed.

Initializing and Accessing Single-Dimensional Array

An array can be initialized in two ways: by declaring and initializing it simultaneously or by accepting elements for the already declared array from the user. Once an array is declared and initialized, the elements stored in it can be accessed any time. These elements can be accessed by using the combination of the array name and the subscript value.

Program 2.1: A program to illustrate initialization of two arrays and display their elements.

```
#include<stdio.h>
#include<conio.h>
#define MAX 5

void main()
{
    int A[MAX]={1,2,3,4,5};
    int B[MAX],i;
    clrscr();
    printf("Enter the elements of array B:\n");
    for (i=0;i<MAX;i++)
    {
        printf("Enter the element: ");
        scanf("%d", &B[i]);
    }
```

```
    printf("Elements of array A: \n");
    for (i=0;i<MAX;i++)
        printf("%d\t", A[i]);
    printf("\nElements of array B: \n");
    for (i=0;i<MAX;i++)
        printf("%d\t", B[i]);
    getch();
}
```

The output of the program is

```
Enter the elements of array b:
Enter a value: 6
Enter a value: 7
Enter a value: 8
Enter a value: 9
Enter a value: 10
Elements of array a:
1    2    3    4    5
Elements of array b:
6    7    8    9    10
```

In this example, an array A is declared and initialized, simultaneously, and the elements for the array B are accepted from the user. Then the elements of both the arrays are displayed.

Once an array is declared and initialized, various operations such as traversing, searching, insertion, deletion, sorting and merging can be performed on an array. To perform any operation on an array, the elements of the array need to be accessed. The process of accessing each element of an array is known as traversal. Generally, the traversal of an array is performed from the element at position 0 to element at position size-1.

Algorithm 2.1 Traversing an Array

```
traverse (ARR, size)

1.  Set i = 0, sum = 0
2.  Print "The elements of the array are: "
3.  While i < size            //size indicates number of elements in the array
        Print ARR[i]
        Set sum = sum + ARR[i]
        Set i = i + 1
    End While
4.  Print "Sum of elements of an array: ", sum
5.  End
```

Program 2.2: A program to illustrate the traversal of an array:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
```

```
/*Function prototype*/
void traverse(int [], int);

void main()
{
    int ARR[MAX];
    int i, size;
    clrscr();
    printf("Enter the number of elements in array:\n");
    scanf("%d", &size);
    printf("Enter the elements of the array:\n");
    for (i=0;i<size;i++)
    {
        scanf("%d", &ARR[i]);
    }
    traverse(ARR, size);
    getch();
}


/*Function to find the sum of the elements of matrix*/
void traverse(int ARR[], int size)
{
    int i, sum=0;
    printf("The elements of the array are:\n");
    for (i=0;i<size;i++)
    {
        printf("%d ", ARR[i]);
        sum+=ARR[i];
    }
    printf("\nSum of elements of an array: %d", sum);
}
```

The output of the program is

```
Enter the number of elements in array: 5
Enter the elements of the array:
12 23 34 45 56
The elements of the array are:
12 23 34 45 56
Sum of elements of an array: 170
```

2.2.2 Multi-Dimensional Arrays

Multi-dimensional arrays can be described as 'arrays of arrays'. A multi-dimensional array of dimension n is a collection of elements which are accessed with the help of n subscript values. Most of the high-level languages including C

support arrays with more than one dimension. However, the maximum limit of an array dimension is compiler dependent.

The syntax of declaring a multi-dimensional array in C is

```
element_type array_name[a][b][c]......[n];
```

where,

`element_type` = the data type of array

`array_name` = name of the array

`[a][b][c]......[n]` = array subscripts

The arrays of three or more dimensions are not often used because of their huge memory requirements and the complexity involved in their manipulation. Hence, only two-dimensional and three-dimensional arrays are discussed in brief in the following section.

2.2.3 Two-dimensional Arrays

A two-dimensional array is defined as an array in which two subscript values are used to access an array element. Two-dimensional arrays are useful when the elements being processed are to be arranged in the form of rows and columns (matrix form).

Generally, a two-dimensional array is represented as:

```
A[Lr : Ur, Lc : Uc]
```

where,

`Lr` and `Lc` = the lower bounds of the row and the column, respectively.

`Ur` and `Uc` = the upper bounds of the row and the column, rspectively.

The number of the rows in a two-dimensional array can be calculated as (`Ur-Lr+1`) and the number of columns can be calculated as (`Uc-Lc+1`).

Like single-dimensional array, a two-dimensional array also needs to be declared first. The syntax of declaring a two-dimensional array in C is:

```
data_type array_name [row_size][column_size];
```

For example, in the statement `int a[3][3]` an integer array of three rows and three columns is declared. Once a compiler reads a two-dimensional array declaration, it allocates a specific amount of memory for this array.

Initializing and Accessing Two-dimensional Arrays

Like a single-dimensional array, a two-dimensional array can also be initialized in two ways: by declaring and initializing the array simultaneously and by accepting array elements from the user.

Once a two-dimensional array is declared and initialized, the array elements can be accessed anytime. Like a one-dimensional array, two-dimensional array elements are also accessed by using the combination of the name of the array and subscript values. The only difference is that instead of one subscript value, two subscript values are used. First subscript indicates the row number and the second subscript indicates the column number of two-dimensional array.

Algorithm 2.2 Traversing Two-Dimensional Array

```
traverse(ARR, m, n)

1.  Set i = 0, sum = 0 //sum stores sum of elements of two-dimensional array
2.  While i < m          //m is number of rows in two-dimensional array
        Set j = 0
        While j < n      //n is number of columns in two-dimensional array
                Print ARR[i][j]
                Set sum = sum + ARR[i][j]
                Set j = j + 1
        End While
        Set i = i + 1
    End While
3.  Print "Sum of the elements of a matrix is : ", sum
4.  End
```

Program 2.3: A program to illustrate traversal of a matrix (two-dimensional array) and finding the sum of its elements.

```c
#include<stdio.h>
#include<conio.h>
#define MAX 10

/*Function prototype*/
void traverse(int [][MAX], int, int);

void main()
{
    int ARR[MAX][MAX], i, j, m, n;
    clrscr();
    printf("Enter the number of rows and columns of a matrix
A: ");
    scanf("%d%d", &m, &n);
    printf("Enter the elements of matrix A: \n");
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            scanf("%d", &ARR[i][j]);
    traverse(ARR, m, n);
    getch();
}

/*Function to find sum of elements of the matrix*/
void traverse(int ARR[][MAX], int m, int n)
{
    int i, j, sum=0;
    printf("Matrix A is: ");
    for(i=0;i<m;i++)
    {
        printf("\n");
```

```
        for(j=0;j<n;j++)
        {
            printf("%d ",ARR[i][j]);
            sum=sum+ARR[i][j];
        }
    }
    printf("\nSum of elements of a matrix is: %d", sum);
}
```

The output of the program is

```
Enter the number of rows and columns of a matrix A: 3 3
Enter the elements of matrix A:
1 2 3
4 5 6
7 8 9
Matrix A is:
1 2 3
4 5 6
7 8 9
Sum of elements of a matrix is: 45
```

2.2.4 Three-dimensional Arrays

A three-dimensional array is defined as an array in which three subscript values are used to access an individual array element. The three-dimensional array can be declared as:

```
int A[3][3][3];
```

Algorithm 2.3 Traversing Three-Dimensional Array

```
traverse(ARR)

1.  Set i = 0, count = 0        //count is used to count the number of zeroes in
                                //three-dimensional array
2.  While i < MAX               //MAX is the size of three-dimensional array
        Set j = 0
        While j < MAX
            Set k = 0
            While k < MAX
                If ARR[i][j][k] = 0
                        Set count = count + 1
                End If
                Set k = k + 1
            End While
            Set j = j + 1
        End While
        Set i = i + 1
    End While
3.  Print "Number of zeroes in given array are: ", count
4.  End
```

Program 2.4: A program to illustrate the traversal of a three-dimensional array and find the number of zeroes in it.

```
#include<stdio.h>
#include<conio.h>
```

```
#define MAX 3

/*Function prototype*/
void traverse(int [][MAX][MAX]);

void main()
{
    int ARR[MAX][MAX][MAX], i, j, k;
    clrscr();
    printf("Enter the elements of an array A(3x3x3):\n");
    for(i=0;i<MAX;i++)
        for(j=0;j<MAX;j++)
            for(k=0;k<MAX;k++)
                scanf("%d", &ARR[i][j][k]);
    traverse(ARR);
    getch();
}

void traverse(int ARR[][MAX][MAX])
{
    int i, j, k, count=0;
    for(i=0;i<MAX;i++)
        for(j=0;j<MAX;j++)
            for(k=0;k<MAX;k++)
                if(ARR[i][j][k]==0)
                    count++;
    printf("Number of zeroes in given array are: %d", count);
}
```

The output of the program is

```
Enter the elements of an array A(3x3x3):
1 2 3 0 4 5 0 6 7
0 7 3 2 0 5 6 0 8
2 0 4 5 0 7 6 9 0
Number of zeroes in given array are: 8
```

2.2.5 String

Each programming language keeps a character set, which are used to communicate with the computer. The character sets are the following types:

1. Alphabets: A, B, C, D, E, F, G, H, I, J , K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

2. Digits: 0 1 2 3 4 5 6 7 8 9

3. Special Characters: ~ ! @ # $ % ^ & * ( )

A NULL character (`'\0'`) or string terminator is automatically appended after the last character of the string. It (`'\0'`) marks the end of the string. This does not happen in integer or float arrays in C. The set of special charters are denoted by string. Length represents the number of characters. If string has zero character then it is known as empty string or even NULL string. Specific strings are denoted by enclosing the characters using single quotation marks. Sometimes, quotation marks are served as *string delimiter*, for example,

`'Information Technology', 'High Level Language', ''`

represent strings with length 21, 19, 0 respectively. The blank space is also a character because it contributes to the length of the string; for example, `S1` and `S2` are the two variables of the given string. The string consisting of the characters of S, followed by the character of `S2` is called the concatenation of `S1` and `S2` and will be denoted by `S1//S2`. For example,

`'Information' // 'Technology' = 'InformationTechnology'`

`But, 'Information' // '' // 'Technology' = 'Information Technology'`

The length of `S1//S2` refers to the sum of lengths of the given strings `S1` and `S2`. A string `Y` is called a substring of `S`. For example,

`'Language'` is the substring of `'High Level Language'`

`'Information'` is the initial substring of `'Information Technology'`.

Therefore, if Y is a substring of S, then the length of Y cannot exceed the length of S. Strings in the form of characters, are stored in the computer using 6-bit, 7-bit or even 8-bit code. The number equal to the number of bits needed to represent a character is called a *byte*. A byte usually refers to 8-bit. A computer which can access an individual byte is known as a *byte-addressable machine*.

String Representation

Various types of string operations are handled in data structure in which you will learn about string representation and handling, how string is represented, end string notation, initializing char array, character representation, string literals, example program to understand the character array concept and the accessing of character array in detail and How to represent array of strings. Strings, sequences of characters, are represented as arrays of char elements; for example, when a programmer wants to represent a char array name having 30 elements it is declared:

`char name[30];`

This means the character array has 30 storage location allocated where it can store up to a maximum of 30 characters. The character array can store the specified maximum of array limit. This means it can store less than the allocated array limit up to the maximum. It is essential that a notation exist to mark the end of character array or string representation. This end of character array is denoted using backslash zero '\0'. In data structure, string is represented by NULL-terminated sequence of characters as shown in the Figure 2.1 (a).

'a'	' '	's'	't'	'r'	'n'	'g'	'\0'
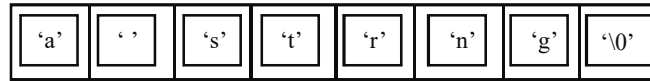
(a) C-style string with a NULL as the end of string

size:8	'a'	' '		't'	'r'	'i'	'n'	'g'
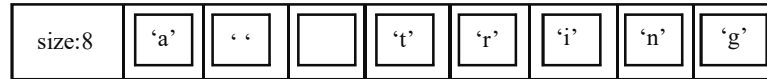
(b) string with the size of data in the first byte

size:7	bufsize:8	'a'	' '	's'	't'	'r'	'i'	'n'	'g'	
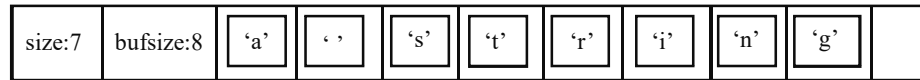
(c) string structure with the size of string data and buffer

Fig. 2.1 *Internal Representation of String*

All string manipulations functions are defined in the header file `string.h`. The limitations of string representation are as follows:

1. Maximum string size can be 255 character due to one byte reserved for size parameter,

2. The string cannot have larger than current size.

Figure 2.1(c) shows the different type of representation, in which a string can accompany with the size of actual string and size of buffer that holds the string. Suppose the programmer wants to initialize the array *char name[30]* with value Information. This initialization of char array can be performed using two methods:

1. Character representation

2. String Literals

The character representation is notated as follows:

```
char name[30]={'I', 'n', 'f', 'o', 'r', 'm', 'a', 't', 'i',
'o', 'n',\0'};
```

The string literal representation is notated as follows:

```
char name[30]= "Information";
```

Character representation uses single quotes to represent each character while string literal representation uses double quotes for the entire string literal. Another main difference of string literals from character representation is the end of character denoted by '\0'. String literal representation uses double quotes, there is no need for representing the end of character. The end of character task is automatically performed by programming language. It is also possible to initialize array as follows:

```
char name[]="Technology";
```

In the above case, the char array `name` is not mentioned with the size and is initialized with the string literal `Technology`. The character array `name` takes the size of the initialized string literal. It is possible to declare two dimensional arrays in character arrays or array of strings. The general syntax for this as follows:

```
char name[num1][num2];
```

In this `num1` denotes the number of string literals defined in the array and `num2` defines the maximum length of characters of the strings; for example, if the programmer wants to represent a `char` array name with string literals as:

```
Information
Technology
Programs
```

Then `num1` takes the `value 3` and `num2` takes the value `11` because `Information` has the maximum character `11`. The representation is as follows:

```
char name[3][8];
```

The access of the character array can be performed by using `for` loop. You can find in the following code, how string operation is performed:

Program 2.5: A program to input a string then check whether string is palindrome or not.

```c
*A palindrome string is same as its reverse like RADAR, DALDA,
MADAM */
#include<stdio.h>  //for printf, scanf,gets  and other
functions
#include<conio.h> //for clrscr() and getch() functions
void main()
{
clrscr();

char arr[10];
int i,j,flag=1;
printf("\n Enter a String\n");
gets(arr);
for(i=0;arr[i]!='\0';i++);
//empty loop which does not have a body
 for(j=0,i—;i>=0;i—,j++)
//i moves backward and j moves forward
 {
  if(arr[i]!=arr[j])
   {
     flag=0;
     break;
   }
}
if(flag==1)
 printf("String is palindrome");
else
 printf("String is not palindrome");
getch();
}
```

The output of the program is:

```
Enter a String
MADAM
String is palindrome
```

String Operations

A string is a collection of characters and terminated with NULL characters and it is denoted by \0. The four string handling functions are as follows:

- strlen() – returns the length of the given string as argument.
- strcpy() – to copy the string from source to destination variable.
- strcmp() – to compare the two given string.
- strcat() – to concatenate (combine or join) the both given strings into one.

For all the functions mentioned in the preceding paragraph, we require the header file string.h.

1. **strlen():** The function strlen() takes string as argument and returns the length of the integers as shown below in the code.

```
Syntax: int strlen(char*);
int len;
char *str = "Sreenivasa Rao";
len = strlen(str);
printf("%d", len);
```

Here the len variable contains the 14 value since the number of characters is fourteen including the space character.

2. **strcpy():** The function strcpy() copies the content of source string to destination string, stopping after the terminating NULL character. The syntax is as follows:

```
char * strcpy(char *dest, char *source);
```

This function returns the pointer to the character.

```
char *s=" Parthiv Sai";
char *s1;
strcpy(s1, s);
printf("the copied string is %s", s1);
```

3. **strcmp():** The function strcmp() compares the two given strings character by character, and returns a value 0 if both the strings are equal, and value < 0 if the string one is less than string two and > 1 if string one is greater than string two. The code is written as follows:

```
int strcmp(char *, char*);
int val;
char *str1 = "Apple";
char *str2 = "apple";
val = strcmp("Apple", "apple");
```

Here, the first string is less than the second string; hence, the value of val is less than zero (0), since the ASCII value of character 'A' differs from character 'a'.

4. `strcat()`: This function concatenates two strings resulting in a single string. It takes two arguments, namely, the pointers to the two strings. The resultant string is stored in the first string specified in the argument list.

```
char destination[25];
char *blank = " ", *c = "Bandaru", *t = "Parthiv";
strcpy(destination, t);
strcat(destination, blank);
strcat(destination, c);
printf("%s\n", destination);
```

Check Your Progress

1. What is an array?

2. What do you mean by traversing an array?

3. Define multi-dimensional array.

4. What is the difference between two-dimensional and three-dimensional arrays?

5. How are strings, sequences of characters represented?

6. How is the initialization of an array performed?

2.3 FUNCTIONS

A function in a program consists of three characteristics:

(a) Function prototype

(b) Function call

(c) Function definition

2.3.1 Function Declaration and Prototype

A function prototype is called a function declaration. A function may be declared at the beginning of the main function. Function declaration is of the following type:

```
return data – type function name (formal argument 1, argument
2, ............. );
```

A function after execution may return a value to the function, which called it. It may not return a value at all but may perform some operations instead. It may return an integer, character, or float. If it returns a float, we may declare the function as

```
float f1(float arg 1, int arg 2);
```

If it does not return any value we may write the above as
```
void fun2(float arg1, int arg2); /*void means nothing*.
```

If it returns a character, we may write
```
char fun3(float arg1, int arg2);
```

If no arguments are passed into a function, an empty pair of parentheses must follow the function name. For example,

```
char fun4();
```

The arguments declared as part of the prototype are also known as formal parameters. The formal arguments indicate the type of data to be transferred from the calling function.

2.3.2   Function Call – Passing Arguments to a Function

We may call a function either directly or indirectly. When we call the function, we pass the actual arguments or values. Calling a function is also known as function reference.

There must be a one-to-one correspondence between formal arguments declared and the actual arguments sent. They should be of the same data type and in the same order. For example,

```
sum=f1(20.5,10); fun4();
```

You will learn more about calling a function by value and by reference in further section.

2.3.3   Function Definition

Function definition can be written anywhere in the file with a proper declarator, followed by the declaration of local variables and statements. **Function definition** consists of two parts, namely function declarator or heading and function declarations. The function heading is similar to function declaration but will not terminate with a semicolon.

The use of functions will be demonstrated with simple programs in this unit.

Suppose you wish to get two integers. Pass them to a function `add`. Add them in the `add` function. Return the value to the main function and print it. The algorithm for solving the problem will be as follows:

<div align="center">Main Function</div>

Step 1:   Define function add

Step 2:   Get 2 integers

Step 3:   Call add and pass the 2 values

Step 4:   Get the sum

Step 5:   Print the value

function `add`

Step 1:   Get the value

Step 2:   Add them

Step 3:   Return the value to main

Thus, you have divided the problem. The program is as follows:

```
/*Example 2.1*

  /* use of function*/

#include <stdio.h>

int main()
```

```
    {
        int a=0, b=0, sum=0;
        int add(int a, int b); /*function declaration*/
        printf("enter 2 integers\n");
        scanf("%d%d", &a, &b);
        sum=add(a, b);  /*function call*/
        printf("sum of %d and %d =%d", a, b, sum);
    }
    /*function definition*/
    int add (int c, int d)  /*function declarator*/
    {
        int e;
        e= c+d;
        return e;
    }
```

Result of the program

```
enter 2 integers
6667 4445
sum of 6667 and 4445 =11112
```

The explanation as to how the program works is given below:

On the fifth statement (seventh line), the declaration of the function `add` is given. Note that the function will return an integer. Hence, the return type is defined as `int`. The formal arguments are defined as `int a` and `int b`. The function name is `add`. You cannot use a variable without declaring it, as also a function without telling the compiler about it. Note also that function declaration ends with a semicolon, similar to the declaration of any other variable. Function declaration should appear at the beginning of the calling function. It hints to the compiler that the function is going to call the function `add`, later in the program. If the calling function is not going to pass any arguments, then empty parentheses are to be written after the function name. The parentheses must be present in the function declaration. This happens when the function is called to perform an operation without passing arguments. In this case, if `a` and `b` are part of the called function `(add)` itself, then we need not pass any parameters. In such a case, the function declaration will be as follows assuming that the called function returns an integer:

```
    int add ();
```

In Example 8.1, you get the values of a and b. After that you call the function `add` and assign the value returned by the function to an already defined `int` variable `sum` as follows:

```
    sum = add ( a , b );
```

Note that `add(a, b)` is the function call or function reference. Here, the return type is not to be given. The type of arguments are also not to be given. It is a simple statement without all the elements of the function declaration. However, the function name and the names of the arguments passed, if any, should be present

in the function call. When the program sees a function reference or function call, it looks for and calls the function and transfers the arguments.

The function definition consists of two parts, i.e., the function declarator and function declarations.

The **function declarator** is a replica of the function declaration. The only difference is that while the declaration in the calling function will end with a semicolon, the declarator in the called function will not end with a semicolon. As in main(), the entire functions body will be enclosed within braces. The whole function can be assumed to be one program statement. This means that all the statements within the body will be executed one after another before the program execution returns to the place in the main function from where it was called.

The important points to be noted are:

(a) The declarator must agree totally with the declaration in the called function, i.e., the return data type, the function name, the argument type should all appear in the same order. The declarator will not end with a semicolon.

(b) You can also give the same name as in the calling function—in declaration statement or function call—or different names to the arguments in the function declarator. Here, we have given the names c and d. What is important; however, is that the type of arguments should appear, as it is in the declaration in the calling program. They must also appear in the same order.

(c) At the time of execution, when the function encounters the closing brace }, it returns control to the calling program and returns to the same place at which the function was called.

In this program, you have a specific statement return (e) before the closing brace. Therefore, the program will go back to the main function with the value of e. This value will be substituted as

```
sum = (returned value)
```

Therefore, sum gets the value which is printed in the next statement. This is how the function works.

Assume now that the program gets a and b values, gets their sum1, gets c and d and gets their sum2 and then both the sums are passed to the function to get their total. The program for doing this is as follows:

```
/*Example 2.2*
  /* A function called many times */
  #include <stdio.h>
  int main()
  {
      float a, b, c, d, sum1, sum2, sum3;
      float add(float a, float b); /*function declaration*/
      printf("enter 2 float numbers\n");
      scanf("%f%f", &a, &b);
      sum1 = add(a, b);  /*function call*/
```

```
        printf("enter 2 more float numbers\n");
        scanf("%f%f", &c, &d);
        sum2 =add(c, d); /*function call*/
        sum3 =add(sum1, sum2); /*function call*/
        printf("sum of %f and %f =%f\n", a, b, sum1);
        printf("sum of %f and %f =%f\n", c, d, sum2);
        printf("sum of %f and %f =%f\n", sum1, sum2, sum3);
}
/*function definition*/
float add (float c, float d)  /*function declarator*/
{
        float e;
        e= c+d;
        return e;
}
```

Result of the program

```
enter 2 float numbers
1.5   3.7
enter 2 more float numbers
5.6   8.9
sum of 1.500000 and 3.700000 =5.200000
sum of 5.600000 and 8.900000 =14.500000
sum of 5.200000 and 14.500000 =19.70000
```

You have defined sum1, sum2 and sum3 as float variables.

You are calling the function add three times with the following assignment statements:

```
        sum1 = add( a, b );
        sum2 = add( c, d);
        sum3 = add( sum1 , sum2 );
```

Thus, the program goes back and forth between main() and add as given below:

```
        int main()
        add (a, b)
        int main()
        add (c, d)
        int main()
        add (sum 1, sum 2)
        int main()
```

Had you not used the function add, you would have to write statements pertaining to add 3 times in the main program. Such a program would be large and difficult to read. In this method, you have to code for add only once and hence, the program size is small. This is one of the reasons for the usage of functions.

In Example 2.2, you could add another function call by add (10.005, 3.1125); This statement will also work perfectly. After the function is executed, the sum will be returned to the main () function. Therefore, both variables and constants can be passed to a function by making use of the same function declaration.

You have seen a program, which calls a function thrice. You will now discuss a problem, which calls three functions.

Problem

The user gives a four-digit number. If the number is odd, then the number has to be reversed. If it is even, then the number is to be doubled. If it is evenly divisible by three, then the digits are to be added. Now, let us write the algorithm for solving the problem.

Step 1:   Get the number.

Step 2:   If the number is odd call, the reverse function.

Step 3:   Else multiply the number by 2 and hence call multiply.

Step 4:   If the number is evenly divisible by 3, call add-digits.

These are the steps. The first one, namely writing a function to multiply by 2 is simple. We will look at the other two steps now.

Reverse

The following steps show the reversing of number.

```
Step 1:   reverse = 0
step 2:   while ( number > 0 )
          reverse = reverse * 10 + (number % 10 )
          number = number/10
Step 3:   return (reverse).
```

add–digits function

```
Step 1:   sum = 0
Step 2:   while number > 0
          sum = sum + (number % 10)
          number = number / 10
Step 3:   return (sum)
```

Let us see how the above algorithm add–digits works.

Let us give 4321 as the number.

```
Step 1:   sum = 0
Step 2:   Iteration 1
          sum = 0 + modulus of (4321/10)
              = 0 + 1 = 1
       number = 4321/10 = 432
    Iteration 2
          sum = 1 + modulus of (432/10)
              = 1 + 2
```

After 4 iterations:

    sum = 1+2+3+4

  Step 3: sum is returned.

  The program is given below:

```
/*Example 2.3*/
/*program to demonstrate calling
multiple functions*/
#include<stdio.h>
int main()
{
    long nummul=0;
    long num=0, rev=0, add_digit=0;
        /*good practice to inialize all variables*/
    long reverse(long num);
    long mult(long num);
    int sum_digit(long num);
    printf("enter unsigned number\n");
    scanf("%lu", &num);
    if (num%2)    /*remainder 1*/
    {
        rev = reverse(num);
        printf("number is odd\n");
        printf("number entered=%lu\n number reversed=%lu\n",
    num, rev);
    }
    else
    {
        nummul=mult(num);
        printf("number is even\n");
        printf("number=%lu\n its multiple=%lu\n", num,
    nummul);
    }
    if (num%3 ==0)
    {
        add_digit= sum_digit(num);
        printf("number evenly divisible by 3\n");
        printf("sum of digits =%lu", add_digit);
    }
}
long reverse(long n)
{
    long r=0;
    while (n>0)
    {
```

```
        r=r*10+(n%10);
        n=n/10;
    }
    return r;
}
long mult(long p)
{
    long sq;
    sq=2*p;
    return sq;
}
int sum_digit(long num)
{
    long sum=0;
    while (num >0)
    {
        sum=sum+(num%10);
        num=num/10;
    }
    return sum;
}
```

Result of the program

```
enter unsigned number
4321
number is odd
number entered=4321
number reversed=1234
```

Look at the program. After getting the number from the user, it evaluates if the remainder of $(num/2) = true$; i.e., if the remainder is 1, then it is true. If the remainder $= 1$, then the number is odd and hence the reverse function is called. The returned value is assigned to rev and printed.

If the number is even, the number is doubled. Since the doubled value may exceed the maximum of the unsigned integer, we have declared it as a long integer.

Next, we check if the number is evenly divisible by 3.

If it is so, then we add the digits. Thus, the main function of Example 2.3 calls three functions for carrying out specific tasks. All the three functions are supplied with the same arguments but return different values.

Function Arguments

You know now that an argument is a parameter or value. It could be of any of the valid types, such as all forms of integers or a float or char. You come across two types of arguments when you deal with functions:

```
formal arguments
actual arguments
```

Formal arguments are defined in the function declaration in the calling function. What is actual argument? Data, which is passed from the calling function to the called function, is called the actual argument. The actual arguments are passed to the called function through a function call.

Each actual argument supplied by the calling function should correspond to the formal arguments in the same order. The new ANSI standard permits declaration of the data types within the function declaration to be followed by the argument name. You have used only this type of declaration as it will help students follow the C++ program easily. This helps in understanding one to one correspondence between the actual arguments supplied and those received in the function and facilitates the compiler to verify that one to one correspondence exists and that the right number of parameters have been passed. It may be noted that formal arguments cannot be used for any other purposes. They only give a prototype for the function. Thus, the names of the formal arguments are dummy and will not be recognized elsewhere, even in the functions in which they are defined.

Although, the types of variables in the function declaration, also known as prototype and function call are to be the same, the names need not be the same. You have already used this concept in Example 8.2 after defining `float a` and `float b` in the functions prototype, you first called `add (a, b)`, `add (c, d)` and then `add (sum1, sum2)`. Thus, the formal arguments defined in the prototype and the actual arguments were not the same in two of the above cases.

When the actual arguments are passed to a function, the function notes the order in which they are received and appropriately stores them in different locations. You must note that even if you use `a` and `b` in the `add` function, they will be stored in different locations. They will have no relationship with `a` and `b` of the main function. Therefore, even if `a` and `b` are assigned different values in the called function, the corresponding values in the calling function would not have changed. You will verify this point in the program in the next section.

2.3.4 Need of User Defined Functions

A User-Defined Function (UDF) is a common feature in any programming languages and is referred as the most important tool of programmers for creating specific applications with reusable code. Because, the computer programs are primarily composed of programming codes that are developed by the computer programmers, hence it is composed of user-defined functions sometimes punctuated by built-in functions.

Using the user-defined functions, the programmers create or develop their own computer programs using the routines and procedures that the computer can follow. Basically, it is the basic building block of any computer program and also very significant for modularity and code reuse because a programmer could create or develop a user-defined function which can perform a specific process and can simply call the routines and procedures whenever it is required. The program

syntax depends entirely on the programming language or application in which these are created.

The simplest form of a user-defined function is one that does not require any parameters to complete its task. These same functions also do not return any values to the calling script or user-defined function. This type of function is often referred to as a 'void' function.

In C programming, the user can define functions according to their need. These functions are known as user-defined functions. For example, to create a circle and to colour it the user will depend upon the radius and colour parameters. Following two user-defined functions can be created to solve this problem:

```
createCircle()  Function
color()  Function
```

Example on User-Defined Function

Following is an example code to add two integers. To perform the addition of integers the user have to create an user-defined `addNumbers()` as shown below.

Example 2.4

```c
#include <stdio.h>
int addNumbers(int a, int b);     // function prototype
int main()
{
  int n1,n2,sum;
 printf("Enters two numbers: ");
 scanf("%d %d",&n1,&n2);
  sum = addNumbers(n1, n2);   // function call
 printf("sum = %d",sum);
  return 0;
}
int addNumbers(int a, int b) // function definition
{
  int result;
  result = a+b;
  return result;       // return statement
}
```

Scope Rules for Local Variable

The scope of the variable is local to the function, unless it is a global variable. For instance,

```c
int function1(int I)
{ int j=100;
   double function2(int j);
   function2(j);
}
```

```
double function2 (int p)
{ double m;
    return m;
}
```

The variable `j` in `function1` is not known to `function2`. You pass it to `function2` through the argument `j`. This will be assigned as equal to `int p`. Similarly, `m` in `function2` is not known to `function1`. It can be made known to `function1` through the return statement. This makes the scope rules of variables in function quite clear. The scope of variables is local to the function where defined. However, global variables are accessible by all the functions in the program if they are defined above all functions.

/*Example 2.5*

```
/* To demonstrate that the scope of a
variable is local to the function*/
#include <stdio.h>
int main ()
{
    float a=100.250, b=200.50;
    void change (float a, float b);
    change (a, b);
    printf ("a= %f b= %f\n ", a, b);
    printf ("these are the original values");
}
/*function definition*/
void change (float a, float b) /*function declarator*/
{
    a +=1000;
    b-=200.5;
}
```

Result of the program

```
a= 100.250000 b= 200.500000
these are the original values
```

We passed `a` = `100.25` and `b` = `200.5` to the function. In the function, you modified `a` as `1100.25` and `b` as `zero`. However, when you print `a` and `b` in the main function, you get the same old values. This confirms that variables are local to the function unless otherwise specified.

Notice; however, that in the calling function, the type declaration of formal parameters is symbolic and used only to indicate the format. You will notice, for example in Example 2.1, that the `int a` has been declared and assigned a value of 0. This has no relationship with `int a` in the function declaration. You could even omit the variable name and declare as `int add(int, int)`. It will still work. Here `a` and `b` have been given for better readability.

This is the reverse in the case of a called function. In the same program, `int c` and `int d` are explicitly defined in function `add`, in the declarator. The variables are used further in the function `add`. This is not the case with the variables in the declaration statement or prototype of the calling function, which will never be used further.

This method of invoking a function is called call by value, i.e., you call the functions with values as arguments.

2.3.5 Scope and Lifetime Declaration of Variables

In everyday programming, it is not necessary to explicitly remove variables from the workspace. All local variables of a function die on exit from that function anyhow and the variables in the global name space usually do not need special treatment. Scope refers to the scope and lifetime of the variables. The scope and lifetime depends on the storage class of the variable in C language. Variables can belong to any one of the four storage classes, i.e., automatic variables, external variable, static variable or register variable. The scope determines over which part or parts of the program the variable is available. Variables can also be categorized as local or global. **Local variables** are the variables that are declared within the function and are accessible to all functions in a program while global variables can be declared both within a function and outside the function. The following are the types of scope of declaring variables:

Block Scope: Block refers to any set of statements enclosed in curly braces ({and}). A variable declared within a block has block scope. Thus, the variable is active and accessible from its declaration point to the end of the block. Block scope is also called local scope. For example, the variable i declared within the block of the following main function has block scope:

```
int main()
{
    int i;   /* block scope */
    .
    .
    .
    return 0;
}
```

A variable with block scope is called a local variable.

Function Scope: Indicates that a variable is active and visible from the beginning to the end of a function. In C, only the goto label has function scope start. For example, the goto start; in the following code shows function scope:

```
int main()
{
    int i;   /* block scope */
    .
    .
    .
    start:   /* A goto label has function scope */
    .
    .
    .
    goto  start; /* the goto statement */
```

```
        .
        .
        .
    return 0;
}
```

Here, the label `start` is visible from the beginning to the end of the `main()` function. Therefore, there should not be more than one label having the same name within the `main()` function.

Program Scope: A variable is said to have program scope when it is declared outside a function. The following is an example for this code:

```
int x = 0;          /* program scope */
float y = 0.0;     /* program scope */
int main()
{
    int i;   /* block scope */
    .
    .
    .
    return 0;
}
```

Here, the int variable x and the float variable y have program scope. Variables with program scope are also called global variables which are visible among different files. These files are the entire source files that make up an executable program. A global variable is declared with an initializer outside a function. The following program shows the relationship between variables with program scope and variables with block scope.

```
/*Example 2.6
/* Program for relationship between program scope and
block scope
#include <stdio.h>
int x = 1234;          /* program scope */
double y = 1.234567;  /* program scope */
void function_1()
{
    printf("From function_1:\n  x=%d, y=%f\n", x, y);
}
int main()
{
    int x = 4321;    /* block scope 1*/
    function_1();
    printf("Within the main block:\n  x=%d, y=%f\n", x,
y);
    /* a nested block */
    {
```

```
      double y = 7.654321;  /* block scope 2 */
      function_1();
     printf("Within the nested block:\n  x=%d, y=%f\n",
x, y);
     }
  return 0;
}
```

Result of the Program

```
From function_1:
  x=1234, y=1.234567
Within the main block:
  x=4321, y=1.234567
From function_1:
  x=1234, y=1.234567
Within the nested block:
  x=4321, y=7.654321
C:\app>
```

As you can see in this program that there are two global variables x and y with program scope and they are declared in lines 4 and 5. A function called `function_1()` is declared. The `function_1()` function contains only one statement. It prints out the values held by both x and y. Because there is no variable declaration made for x or y within the function block, hence the values of the global variables x and y are used for the statement inside the function. To prove this, the `function_1()` function is called twice in the above program, respectively, from two nested blocks. The output shows that the values of the two global variables x and y are passed to `printf()` enclosed in the `function_1()` function body. Then, another integer variable, x, is defined with block scope, which can replace the global variable x within the block of the `main()` function. The result made by the statement in line 17 shows that the value of x is the value of the local variable x with block scope while the value of y is still that of the global variable y.

The lifetime of the variable retains a given value during the execution of the program. **Lifetime of variable** is the period of time during which that variable exists during execution. Some variables exist briefly. Some variables are repeatedly created and destroyed while others exist for the entire execution of a program. The automatic variables and static variables decide the lifetime phenomena in the program.

An automatic variable's memory location is created when the block in which it is declared is entered. An automatic variable exists while the block is active and then it is destroyed when the block is exited. Since a local variable is created when the block in which it is declared is entered and is destroyed when the block is left. One can see that a local variable is an automatic variable.

A **static variable** is a variable that exists from the point at which the program begins execution and continues to exist during the duration of the program. Storage

for a static variable is allocated and initialized once when the program begins execution. A global variable is similar to a static variable since a global variable exists during the duration of the program. Storage for the global variable is allocated and is initialized once when the declaration for the global variable is encountered during execution. Thus, both the global variable and the static variable have a history preserving feature and they continue to exist and their contents are preserved throughout the lifetime of the program. A programmer can also declare a local variable to be static by using the keyword static in the variable's declaration as in the following example:

```
static int num=0;
```

For most applications, the use of automatic variables works just fine. Sometimes, however, programmers want a function to remember values between function calls. This is the purpose of a static variable. A local variable that is declared as static causes the program to keep the variable and its latest value even when the function that declared it is executing. It is usually better to declare a local variable as static as to use a global variable. A static variable is similar to a global variable in that its memory remains for the lifetime of the entire program. However, a static variable is different from a global variable because a static variable's scope is local to the function in which it is defined. Thus, other functions in the program cannot modify a static variable's value because only the function in which it is declared can access the variable.

2.3.6 Return Values

The return data type is declared in the function declaration in the `main()` function or the calling function and the declarator is indicated in the first line of the function definition. If no value is to be returned, the return data type `void` is specified. `Void` simply means NULL or nothing. Therefore, it does not fall in any other data types, such as `integer` or `float` or `char`.

The return value as you have seen is the result of computation in the called function. You return a value, which is stored in a data type in the called function. The return value means that the value, thus stored in the called function is assigned or copied to a variable in the `main()` or calling function. Therefore, to receive the result, a data type should have been declared and preferably initialized in the calling function.

The return statement can be any of the following types:

```
return (sum);
return V1;
return "true";
return 'Z';
return 0;
return 4.0 + 3.0;
```

In some examples, you have returned variables whose values are known when they are returned and in other examples, you return constants. You can even return expressions. If the return statement is not present, it means the return data type is `void`.

You can also have multiple return statements in a function. However, for every call, only one of the return statements will be active and only one value will be returned.

Arrays and Functions

There is no restriction in passing any number of values to a function; the restriction is only in the return of values from a function. Therefore, arrays can be passed to a function without any difficulty, one element at a time, as follows:

```c
#include <stdio.h>
int main()
{
    int a[]={1,2,3,4,5};
    int j;
    int func(int a);
    for (j=0; j<=4; j++)
        func (a[j]);
    ..........
}
int func(int c)
{
    ......
}
```

Here, `func` has been declared as a function passing a single integer. Note here that the declaration or the prototype gives only the format of the parameters passed. The values are only indicative and are not actual values. They are the formal values. Therefore, the parameters declared inside the parentheses act only as a checklist. They cannot be used in the main function elsewhere without actually declaring them on top of the function. But, for this rule, there would have been a conflict between `a[]` which is an array and `a` which is a simple variable. Here, no conflict arises because `a` is not recognized in the main function. It is only a checklist to see that whenever the function calls `func`, an integer has to be passed. If we try to pass a `float`, the compiler will detect an error. This is not so in the case of variables defined in the function declarator above the functions body, as they are recognized as actual names. In this case, `int c` is declared as a variable in `func`. The initial value will be the same as passed by the calling function. Thus, since `a` is used in the function declaration, only one integer can be passed to the function `func`. Actually, the entire array can be passed to a function irrespective of its size, by suitable declaration, as the following example indicates.

/*Example 2.7*/

/* To find the greatest number in an array*/

```c
#include <stdio.h>
int main()
{
    int array[]= {8, 45, 5, 911, 2};
    int size=5, max;
```

```
        int fung(int array[], int size);
        max=fung(array, size);
        printf("%d\n", max);
}
int fung(int a1[], int size)
{
        int i, j, maxp=0;
        for (j=0; j<size; j++)
        {
           if (a1[j] > maxp)
           {
              maxp=a1[j];
           }
        }
        return maxp;
}
```

Result of the program

```
911
```

The objective of Example 2.7 is to find the greatest number in an array. In the program, an array called array is initialized with 5 values as given below:

int array[] = {8, 45, 5, 911, 2}; size is declared as 5 and a function called fung has been declared. It will pass an array and an integer to the called function. The array size has been kept open and the called function will return an integer. The next statement calls fung and passes all elements of the array and an integer 5 equal to size. The function gets the actual values and size=5. The maximum value in the array is found in the for loop and stored in maxp. The value maxp is returned to the main function and printed there. Thus, the function is called by value.

Call by Value

In this section, you have been calling functions by passing values. For example, function calls in some of the above programs are as follows:

```
change(a, b);
rev = reverse(num);
```

The values passed to the function change are a & b which are known. Similarly, while calling function reverse, we pass num. This is called call by value. When you call functions by value, the called functions can return only one value.

Call by Reference

The call by reference method of parring agreements to a function copies the address of an argument into the formal parameter. So, it results into the modification in the values of actual parameters.

2.3.7 Storage Classes

A variable has two specifiers, namely data type and storage class.

- Data type (e.g., `int, float, char`, etc.)
- Storage class (e.g., `auto, static`, etc.)

Data type specifies the types of data stored in a variable. Storage class specifies the segments of the program where the variable is recognized and how long the storage of the value of the variable will last.

There are four types of storage class specifications as follows:

- `automatic`
- `register`
- `static`
- `extern`

You have so far been defining only the data type of the variables but not the storage class. You may wonder then, how your programs worked! You as a programmer have to specify the type when it is required to operate in a particular manner. If the storage class is not specified, the compiler will assume the type on its own. The storage class is applicable to all types of variables and is prefixed to the data type declaration as given below:

```
auto char  z ;
extern int a, b, c ;
static float x ;
register char  y ;
```

The basic characteristics of each storage class are discussed in the following sections.

Automatic Variables

Any variable declared in a function is assumed to be an automatic variable by default. The following are automatic variables:

(i) **Storage Location:** Except for `register` variables, the other three types will be stored in memory.

(ii) **Scope:** `auto` variables are declared within a function and are local to the function. This means the value will not be available in other functions.

Any variable declared within a function is interpreted as an `auto` variable, unless a different type of storage class is specified.

`auto` variables defined in different functions will be independent of each other, even if they have the same name.

`auto` variables are local to the block in a function. If an `auto` variable is defined on the top of the function after the opening brace, then it is available for the entire function. If it is defined later in a block after another opening brace, it will be valid only till the end of the block, i.e., up to the corresponding closing brace.

The following program illustrates the concept.

Program 2.6: Program to Demonstrate the use of `Auto` Variable

```c
#include <stdio.h>
int main()
{
    auto int x=10;
    void f1(int x);
    int f2(int x);
    {
        auto int x =20;
        printf("x = %d in the first block\n", x);
        x=f2(x); /*20 is passed to f2 and returned value
assigned to x*/
        printf("x = %d after the return from f2\n", x);
    }
    printf("x = %d after the first block\n", x);
    {
        auto int x=30;
        printf("x = %d in the second block\n", x);
    }
    printf("x = %d after the second block\n", x);
    f1(x); /*x=10 is passed to the function f1*/
    printf("x = %d after return from function will be 10\n",
x);
}
void f1(int a)
{
    auto float x=5.555; /*integer x will be lost*/
    printf("x = %f in the function\n", x);
}
int f2(int x)
{
    auto int y=100;
    y+=x; /*y will be 120*/
    printf("y = %d in the function\n", y);
    return y;
}
}
```

Execute the program and you will get the following results:

Result of the program:

```
x = 20 in the first block
y = 120 in the function
x = 120 after the return from f2
x = 10 after the first block
x = 30 in the second block
x = 10 after the second block
x = 5.555000 in the function
x = 10 after return from function will be 10
```

This gives a clear idea about the scope of the `auto` variables.

(iii) **Initial Values:** The `auto` variable will contain some garbage values unless initialized. Therefore, they must be initialized before use.

(iv) **Life:** How long will the value stored in the `auto` variable last?

It will last as long as the function or block in which it is defined is active. If the entire function has been executed and the value has been returned, the value of the `auto` variables of the functions will be lost. Remember that you cannot call it later.

`register` Variables

`register` variables have characteristics similar to `auto` variables. The only difference between them is that while `auto` variables are stored in memory, `register` variables are stored in the register of the CPU. The initial value will be an unpredictable or garbage value; the variables are local to the block and they will be available as long as the blocks are active.

Why then do you need to declare one more storage class? The CPU registers respond much faster than the memory. After all, you want to access, store and retrieve the stored variables faster so that the computing time is reduced. Registers are faster than memory. Therefore, those variables which are used frequently can be declared as `register` variables.

They are declared as,

```
register int i ;
```

A memory's basic unit may be 1 byte but depending on the size of the variable even 10 contiguous bytes of memory can be used to store a `long double` variable. Such an extension of size is not however possible in the case of registers. The registers are of fixed length like 2 bytes or 4 bytes and therefore, only `integer` or `char` type variables can be stored as `register` variables. Since registers have many other tasks to do, `register` variables may be defined sparingly. If a `register` variable is declared and if it is not possible to accept it as a `register` variable for whatever reasons, the computer will treat it as an `auto` variable. Therefore, the programmer may specify a frequently used variable in a program as a `register` variable in order to speed up the execution of the program.

External (Global) Variables

External variables are also known as global variables. What is global? The scope of external variables extends to all the functions of a program. You have so far created all the functions in a single file. You can create functions in more than one file. However, for the sake of simplicity, assume that all the functions are in one file. Global variables will be declared like other variables with a storage class specifier extern. For example,

```
extern  int a, b

extern  float c, d
```

The scope of the variables starts from the point of declaration to the end of the program. The value of the external variable at any point of time is that of the last assignment. Assume that the main function may assign a = 10. The function z may then use it and perform a calculation and at the end assign a value 20. If printed at that point of time, the value will be 20. It may be called by another function p where its value may become zero. If, at this point of time, the main() function calls or z calls it, the value will be 0. Thus, external variable is accessible and transparent to all the functions below it. We will write a program to demonstrate this concept.

Program 2.7: Program to Demonstrate use of External Variable

```c
#include<stdio.h>
extern int ext_a=10;
int main()
{
    int f1(int a);
    void f2(int a);
    void f3();
    printf("ext_a = %d in the main function\n", ext_a);
    f1(ext_a);
    printf("ext_a = %d after the return from f1\n", ext_a);
    f2(ext_a);
    printf("ext_a = %d after the return from f2\n", ext_a);
    ext_a*=ext_a;
    printf("ext_a = %d \n", ext_a);
    f3();
    printf("ext_a = %d after return from f3\n", ext_a);
}
int f1(int x)
{
    ext_a-=10;
    return ext_a;
}
```

```
void f2(int x)
{
    ext_a+=20;
}
void f3()
{
    ext_a/=100;
}
```

Result of the program:

```
ext_a = 10 in the main function
ext_a = 0 after the return from f1
ext_a = 20 after the return from f2
ext_a = 400
ext_a = 4 after the return from f3
```

How does the program work?

ext_a is declared as an external variable with value 10 before main(). Therefore, ext_a will be recognized all through the program.

f1, f2 and f3 are functions.

f1 returns an integer and f2 returns void, i.e., it does not return a value.

f3 neither receives nor returns any value.

In the first printf(), we get ext_a = 10.

Now f1 is called. In f1 ext_a = 10 is passed as an argument.

The value of ext_a is 0 now in function f1. The second printf() in main prints ext_a = 0

Now f2 is called.

ext_a becomes 20 now. It does not return any value. However, the third printf() prints the value as 20. How does this happen? It is because the current value of ext_a is known to main() even without f2 passing it.

We discussed that a function can return only one value. However, by using a global variable, you can overcome this limitation as follows:

You square ext_a, i.e., ext_a = 400 now.

The 4th printf() statement confirms this. Now, you call f3. In spite of the fact that you neither passed an argument nor returned any value from f3, ext_a is known to f3 as 400. Then, 100 divides ext_a. Therefore, ext_a will be 4 as confirmed by the fifth printf() statement.

This program illustrates the concept of external variables in simpler situations where the name of the global variable is not assigned to the function's local variables.

It is perfectly legal to use the same name for different local variables in a function. We can even use the name and declare it as another data type.

For example, you can define `ext_a` as a `float` in another function `f1`. Then how is the conflict to be resolved? You will reserve the answer to the question for a few minutes.

You should be careful while handling external variables because the variables may be disturbed in a remote corner inadvertently. Global variables when declared on top of `main()` can be identified easily and therefore, the storage class specifier `extern` need not be specified in such situations. If it cannot be easily recognized by declaration elsewhere in the program, it should be specified clearly.

The initial value of an external variable is zero, if not assigned. The life of the variable is till the termination of program execution. The scope extends from the point of declaration till the end. It will be stored in memory.

`static` Variables

The initial value of `static` variables is zero unless otherwise specified. This is also stored in memory. `static` variables are declared as follows:

```
static int x, y, z ;
static char a ; etc.
```

`static` variables are local to the functions and exist till the termination of the program. Therefore, when the program leaves the function, the value of the `static` variable is not lost. If the program calls the function again, the `static` variable will execute the function with the value it already possesses. Assume that `f1` is a function containing a `static` variable as given below:

```
main ()
{
    int f1 (-);
    f1 (-);
}
int f1 (-)
{
    static int var = 0 ;
}
```

When `f1` is called the first time, `var` will be initialized to zero. If `var` is finally assigned the value 10 at the end of `f1`, then `var = 10` will remain till the program stops execution and if main calls `f1` again, the value of `var` will not be initialized to 0 again but will remain as 10. The initialization `var = 0` will not have any effect. However, `var` can further be modified depending on the statements in `f1`. Had it been an `auto` variable, `var` would have been initialized each time to zero. This is essentially due to the value of `auto` variable being lost immediately after the program leaves the block. This is one of the differences between `static` and `auto` variables. `static` variables; however, will not be known outside the function, i.e., in other functions, such as `main()` or any other functions called by `main()`.

Now, consider the conflict arising out of external variables and local variables (`auto/register, static`) having the same names. In such cases, the local variables take precedence over the external variables. This means that in a function the local variable of the same name would only be recognized. The function will be blind to the external variable of the same name. However, the global variables of other names will be recognized as explained already. All other conditions remain the same. The value of a local variable does not affect the global variable and vice versa. The initial value of the local variable will be dependent upon whether it is `static` or `auto`. The program below explains the concept of the working of the different storage classes.

Program 2.8: Program to Demonstrate Scope of Variables

```c
#include <stdio.h>
char chara,charb,charc; /*global variables*/
int main()
{
    int disp();
    char prn(char m);
    chara='x';
    charb='y';
    charc='z';
    prn(charc);
    putchar(chara);
    disp();
    putchar(charb);
    disp();
}
int disp()
{
    static int charb;
    charb=charb+1;
    printf("%d\n",charb);
    return charb;
}
char prn(char charn)
{
    auto char chara;
    putchar(charc);
    chara=charn;
    putchar(chara);
    return chara;
}
```

Result of the program:

```
zzx1

y2
```

Let us understand how the program functions.

You declare `chara, charb` and `charc` as global variables and in `main()`, you assign `chara = x, charb = y, charc = z`.

You have declared `disp()` as a function passing no variable but returning an integer.

You have declared `prn()` as a function passing and returning a character.

You call function `prn()`. You have a `chara` of type `auto` in `prn()`. The statement `putchar (charc)` will display the value of `charc` in the `main` function, which is z.

The next `putchar (chara)` in `prn()` will display z because of `chara = charc`. z is returned to `main()`.

Now `putchar (chara)` will print x and not z since in the `main()` function, `chara` refers to the global variable.

Now you call `disp ()`. Although, you have not initialized it, the initial value of b will be zero and therefore, it will print 1. Now the program returns to `main()`.

The next `putchar (charb)` will print y because the global variable is active in the `main()` function. Now, you call `disp()` again. Since the old value of b is not lost, the next time, the program prints 2. Thus, the value of a `static` variable is maintained between function calls. Local variables get precedence over global variables.

External (Global) Static Variable

A `static` variable can be placed outside all functions. Then, it is called external `static` variable. An ordinary external variable is accessible by all functions in any file. But, the external `static` variable is accessible by functions in the same file where the variable was declared. An external `static` variable can be defined outside all functions as follows:

```
#include <stdio.h>

static int ext_a=10;

int main()

{
```

Initialization

The importance of initialization was stressed a number of times. It is formally discussed here.

External and `static` variables are automatically initialized to zero. On the other hand, `auto` and `register` variables get initialized to garbage values and should, therefore, be initialized with constants or by expressions. For example,

```
auto int a = 10;
auto char ch = 'z';
```

When it is an expression, the contents of the expression should have been defined previously. For example,

```
auto int a = 5;
auto int b = a + a * 5;
auto int c = a * b;
```

`static` and external variables can only be initialized with a constant. For example,

```
extern char z = 'A';
static double = 343.25;
```

Note also that `static` and external or global variables are initialized only once, i.e., before program execution. However, in the case of `auto` variables, the initialization is carried out every time the function or block is entered. Arrays can also be initialized with statements like:

```
int Z [ ] = { 1, 2, 3, 4, 5, 6 } ;
```

Similarly, a string can be initialized as follows:

```
char string1 [ ] = " peter ";
```

Multi-File Program

Programs so far seen were contained in one file. Large programs may reside in more than one file. You may need a global variable to be accessible in all functions in all files. If such a need arises, you have to write the defining declaration in one file and referencing declaration in all other files. The defining declaration of the variable should not use the `extern` keyword. It can be declared on top of the file without using the `extern` keyword as shown below:

/* File 1 Defining Declaration*/

```
#include <stdio.h>
int ext_a=10;
int main()
{
```

The other file should also declare the variable by prefixing the **extern** keyword as shown below.

/* File 2 Referencing Declaration*/

```
#include <stdio.h>
extern int ext_a;
sort()
{
/* File 3 Referencing Declaration*/
#include <stdio.h>
extern int ext_a;
```

```
int arrange()
{
```

It should be noted that the initial value can be assigned in the defining declaration and nowhere else as shown in the above program segments.

2.3.8   Command Line Arguments

These are called command line arguments. You have so far been passing arguments to functions other than the main function. You can also pass arguments to main function, but since this is the first function to be executed, the arguments are to be supplied before calling it. When do you call main? You call main when you execute the program itself. If the program is executed under a DOS environment, i.e., at C> prompt, then you can run the executable file of the program and pass the arguments. For example, you can create `prg. exe`. When you execute this `prg`, the main function of `prg` will be executed. If you want to pass arguments then you can pass it to main along with the command itself. For example,

```
C > prg. exe 100, 200, 300
```

You type the above mentioned command at the C prompt in DOS shell. The name of file containing the program is to be followed by the arguments or values to be passed to the program, i.e., the `main` function. In this way you can pass arguments to the main function in the command line. The main function is always called with two arguments, neither more nor less. The first is called `argc` and the second argument is called `agrv`. It is the convention to call the command line arguments such as these as `argc` and `agrv`. The `argc` stands for argument count. It is an integer type argument and contains the total number of arguments passed. The second is called the argument vector, conventionally called `agrv`, and is a pointer to an array of strings that contain the arguments. Each argument will be a string. If you want to pass 4 argument then `argc` will contain 4 and `agrv` will point to the array of 4 strings, i.e., the address of the first string. `agrv[0]` will always point to the name of the program. In the above example, `prg. exe` will be stored in `agrv[0]`. 100, 200 and 300 will be stored as strings `agrv [1]`, `agrv [2]` and `argv [3]` respectively. If `argc = 1`, there is only one string, i.e., the program file. Therefore there are no other command line arguments. The following program prints the contents of the `agrv`. For this to happen the program has to be executed in DOS mode and arguments are to be supplied in the command line. An example of the program is given below:

```
/*Example 2.8*/
/* To demonstrate argc, agrv*/
#include <stdio.h>
int main(int argc, char *agrv[])
{
    int j;
    for (j=0; j<argc; j++)
        printf("%s\n", agrv[j]);
        return 0;
}
```

The names `argc` and `agrv` are conventional, but any other name could be used as well. Compile the program. When successful, go to DOS Shell. Type the program name, followed by whatever strings you want. It will reproduce whatever was typed in the C prompt.

Output of the Program when the file name followed by the arguments.

Result of the program

```
D:\CPROG1\BX62.EXE
100
200
300
400
```

We did not give `argc` explicitly, and the compiler counted the number of arguments and assigned the value for `argc`. Do not look for the same type of result, since this will depend on the arguments and file name actually given.

This demonstrates the command line arguments passed to the `main` function at run time.

This can be used to copy a file to another file. Assume that the first named file is to be copied to the second named file. We may write a program and convert it into an executable file, specifying the argument in the DOS command line.

We may specify as follows at the C> prompt:

```
C > prgname . exe f1.cpp f2.cpp
```

This means that we want to copy the contents of `f1.cpp` to `f2.cpp.` Here, the number of arguments are 3, and therefore `argc` will contain value 3.

```
*agrv[0]   =      prgname.exe
*agrv[1]   =      f1.cpp – source to copy from
*agrv[2]   =      f2.cpp – file where to be copied
```

A character at a time is to be fetched from `f1.cpp` and put into `f2.cpp`.

Creation of a Utility to Search for a given String in a File

Let us write a program `sfind`. The program has to be stored in this name after compilation, and executed in the command line. The argument `agrv[0]` is the name of the file. The string to be searched in the file is given as `agrv[1]`. The file where to be searched is given as `agrv[2]`. Thus there will be three arguments as given below:

```
sfind exe        swamy ws.doc
```

When this is typed the program should find whether `swamy` is in `ws.doc`; if so it should return the string, and if not, say so. The program `sfind.exe` is given below:

```
/* Example 2.9 – Finding given string in file */
#include <stdio.h>
#include<string.h>
int main (int argc, char *agrv[])
```

```
    {
        char alpha[80];
        FILE *fr;
        fr=fopen(agrv[2], "rb");
        while
        (   fgets(alpha,79, fr)!=NULL)
        {
            if(strstr(alpha, agrv[1])!=NULL)
            {   printf("String %s found", agrv[1]);    }
            else
                puts("string not found");
        }
        fclose (fr);
        return 0;
    }
```

Result of the program

```
C:\BORLANDC\BIN>sfind.exe Delhi ws.doc
String Delhi found
```

The main() is passed the arguments. The file ws.doc is opened. The contents are brought, line after line to the buffer and the matching of strings checked by using the library function strstr. If the result is NULL, the string is not found. If it is not NULL, then the string supplied as agrv[1] is found in the file.

When this program is available we need not specify the actual names of the string and file to be scanned in the program file. The file name and string name can be typed from the keyboard at run time. Thus this program has become a general purpose program.

2.3.9 Recursion in Functions

Basic Concepts

The previous section dealt with the concept of a function calling another function, as well as multiple functions, being called by a number of functions. A function calling itself is called **recursion** and the function may call itself either directly or indirectly. This concept is difficult to understand unless explained through examples. Every program can be written without using recursion but the reverse is not true. Some problems; however, are suitable for recursion. For instance, the factorial problem can be solved using recursion as shown in program below:

```
/* Example 2.10*
    To find the factorial of a given number*/
    #include <stdio.h>
    int main()
    {
        int n;
        long int result;
        long int fact(int n);
```

```
        printf("Enter the number whose ");
        printf("factorial is to be found\n");
        scanf("%d", &n);
        result=fact(n);
        printf("result=%ld", result);
    }
    long int fact(int n)
    {
        if (n<1) return 0;
        else
            if (n==1) return 1;
            else
                return (n*fact(n-1));
    }
```

Result of the program
```
Enter the number whose factorial is to be found
10
result=3628800
```

Now, let us analyse how the program proceeds. You get an integer n from the keyboard. In order to find factorial n, you call fact(n), where fact is the function for finding the factorial of number n. The recursion takes place in function fact. Assume that n=1. The main function calls fact(1), which will be assigned to result in the main function after return from the function. In the function, since n is equal to 1, 1 is returned and printed in main().

Next, assume you want to find out the factorial of say, 2 and fact(2) is called. In the function fact, since n is not equal to 1, n * fact(n-1) is returned, i.e., 2 * fact(1) is returned to result. Result = 2 * fact(1). This intermediate result is stored somewhere and can be called stack. Stack is an array which stores values and gives the last element first. The writing into stack is popularly called push and getting information from stack is called pop. You have not defined any stack and therefore, you can assume that the system does this for you. After pushing the intermediate result into stack, the program calls fact(1), which returns 1. Now, the intermediate result is popped and the value of fact 1 is substituted to get the factorial of 2 as 2.

Now call factorial 5. You call fact and get back,
    result = 5 * fact(4)   [1]

Now, fact(4) is called to get 4 * fact(3.) Substituting this in equation [1], we get
    result = 5 * 4 * fact(3)

fact(3) again is called to get 3 * fact(2) and so on till we get fact(1) which will be returned as 1. Therefore, we get factorial 5 as 5 x 4 x 3 x 2 x 1. Such repetitive calling of the same function is called recursion. The calls are recursive calls. The result and function fact has been declared to be of

type `long` so as to take care of large numbers. If the `fact` function were not called repeatedly, the program size would have become very large. Thus, recursion keeps the program size small but understanding recursion is not easy. If the program can be visualized as recursive, it will result in compact code. Recursive functions can easily become infinite loops. Therefore, the functions should have a statement with `if` so that the program will definitely terminate. In the factorial program, assume for a moment that the first statement in fact function is absent. Then we have to end the program only when n becomes 1. What will happen, if by chance, n is entered as a negative number? The program will get into an endless loop. Therefore, to avoid such eventualities, you can either have a statement as follows:

```
If (n<1)        exit().
```

Or you could do this by the statement `if (n<1) return 0`, as has been done here.

This will ensure that if either 0 or a negative number is entered, you get the factorial as 0 and the program will terminate gracefully.

You should also note that recursive programs simulate the use of stack. You write to the stack and retrieve information from the stack. Writing to stack is called `push` and retrieving or reading or getting value from the stack is known as `pop`. The feature of stack is last in, first out and therefore, you get the value of the data entered last first, as illustrated in the following example.

You `push` or `pop` only through the top of the stack. Assume that you want to `push a, b & c,` one at a time. You `push 'a'`. It will be on top of the stack. When you push b, a will be pushed to the next lower location and b will occupy the top of the stack. Next, when you `push 'c', 'c'` will occupy the top of the stack, `'b'` one location before and `'a'` one location before `'b'`. Thus, you can push data till the stack becomes full. If you pop the stack now, it will eject `'c'`, then `'b'` and so on. Thus, you `pop` on a last-in-first-out basis.

Reconsider the factorial program in which you were storing intermediate results in a stack-like manner and popping LIFO. Take the example of finding the factorial of 4. On the first call, you get `4 * fact(3)`. You push this into the stack and in the next call, you get `3 * fact(2)`. Again, you push the intermediate result to stack and evaluate `fact(2)` to get `2 * fact(1)`. This is also put to stack. Now, you pass `fact 1` and `get fact(1)` as 1, after which you get the value of `fact(2)` by popping `fact(2)` as `2 * fact(1)`. The popping continues till the result is obtained. Such a conceptualization will help you to understand recursion easily.

2.3.10 Implementation of Euclid's `gcd` Algorithm

Euclid's `gcd` algorithm is quite suitable for recursion. The modified algorithm, which uses recursion, is as follows:

Algorithm using recursion

Main function

```
Step 1  Read two integers m and n
Step 2  Call function gcd (m, n)
Step 3  Print gcd
```

Function `gcd (m, n)`
```
Step 1  if (n==0) return m;
else
return (gcd(n, m%n));
Step 3  End
```

When two integers are received, the main function calls `gcd` function. In the `gcd` function, it is checked whether `n` equals zero. If so, `m` is the `gcd` if not, the function calls `gcd` again by changing the arguments as follows:

```
m=n
n=m%n
```

See the clarity in the above function.

You can easily observe that by using recursion, even the number of steps in the program has gone down. But, it requires a little skill to convert the program into a recursive function. The program implementing the above algorithm is as follows:

```
/*Example 2.11*
Euclid's GCD*/
#include<stdio.h>
int main()
{
int m, n;
int gcd(int m, int n);
printf("Enter 2 integers\n");
scanf("%d %d", &m, &n);
printf("GCD of %d and %d=%d", m, n, gcd(m,n));
}
int gcd(int m, int n)
{
if (n==0) return m;
else
return (gcd(n, m%n));
}
```

The program was executed twice and the result of the program is as follows:

Result of the program

```
First Time
Enter 2 integers
12 256
GCD of 12 and 256=4
Second Time
Enter 2 integers
1225 625
GCD of 1225 and 625=25
```

You can even enter the first number to be lower than the second number as executed during the first time. The program works all right because in one iteration, the numbers get reversed namely the first number is larger than the second number after iteration. Thus, recursion is quite suitable for solving this problem.

Check Your Progress

7. Where a function definition should be written in a program?

8. What does a function definition consist of?

9. How does the use of pointers economize memory space?

10. What is a void pointer? How is it useful?

11. How a function returns a pointer?

12. How can we pass pointers to structures? Give an example.

13. Name the functions that are useful as arguments to functions.

14. Which symbol is used to terminate the label?

15. What is recursion?

2.4 STRUCTURES

A structure is synonymous with a record. A structure, similar to a record, contains a number of fields or variables. The variables can be of any of the valid data types.

2.4.1 Structure Initailization

In order to use structures, you have to first define a unique structure. The definition of the record `book`, which you call a structure, is as follows:

```
struct book
{
    char title [25];
    char author [15];
    char publisher [25];
    float price;
    unsigned year;
};
```

`struct` is a keyword or a reserved word of 'C'. A structure tag or name follows which is `book` in this case. This is not a must but giving a tag to structure will improve the reader's understanding. The beginning of the structure is indicated by an opening brace. Thereafter, the fields of the record or data elements are declared one by one. The variables or fields declared are also called members of the structure. Structure consists of different types of data elements which is different from an array. Let us now look at the members of `struct book`.

The `title` of the book is declared as a string with width 25; similarly, the `author` and publisher are arrays of characters or strings of the specified width. The price is defined as a `float` to take care of the fractional part of the currency. The `year` is defined as an `unsigned integer`.

Note carefully the appearance of a semicolon after the closing brace. This is unlike other blocks. The semicolon indicates the end of the declaration of the structure. Thus, you have to understand the following when you want to declare a structure:

(a) `struct` is the header of a structure definition.

(b) It can be followed by an optional name for the structure.

(c) Then the members of the structure are declared one by one within a block.

(d) The block starts with an opening brace but ends with a closing brace followed by a semicolon.

(e) The members can be of any data type.

(f) The members have a relation with the structure; i.e., all of them belong to the defined structure and they have identity only as members of the structure and not; otherwise. Therefore, if you assign a name to the author, it will not be accepted. You can only assign values to `book.author`.

2.4.2 Declaration: Assigning Values to Members

The structure definition given above is similar to the prototype in a function in so far as memory allocation is considered. The system does not allocate memory as soon as it finds the structure definition, which is for information and checking consistency later on. The allocation of memory takes place only when structure variables are declared. What is a structure variable? It is similar to other variables. For example, `int I` means that `I` is an integer variable. Similarly, the following is a structure variable declaration:

```
struct book s1;
```

Here `s1` is a variable of type structure book. Suppose, you define,

```
struct book s1, s2 ;
```

This means that there are two variables `s1` and `s2` of type `struct book`. These variables can hold different values for their members.

Another point to be noted is that the structure declaration appears above all other declarations. An example which does nothing but defines structure and declares structure variables is as follows:

```
main()
{
    struct book
    {
    char title [25];
    char author [15];
    char publisher [25];
    float price;
    unsigned year;
    };
struct book s1, s2, s3;
}
```

If you want to define a large number of books, then how will you modify the structure variable declaration? It will be as follows:

```
struct book s[1000];
```

This will allocate space for storing 1000 structures or records of books. However, how much storage space will be allocated for each element of the array? It will be the sum of storage spaces required for each member. In `struct book`, the storage space required will be as follow:

```
title 25 + 1 (for NULL to indicate end of string)
author 15 + 1
publisher 25 + 1
price 4
year 2
```

Therefore, the system allots space for 1000 structure variables each with the above requirement. Space is allocated only after seeing the structure variable declaration.

Look at another example to make the concept clear. You know that the bank account of each account holder is a record. Now, define a structure for it.

```
struct account
{
unsigned number;
char name [15];
int balance ;
} a1, a2;
```

Instead of declaring separate structure variables, such as `struct account a1, a2;` we can use coding as in the example given above. Here, the variables are declared just after the closing brace of the structure declaration and terminated with a semicolon. This is perfectly correct. The declaration of the members of the structure is clear; the balance has been declared as an integer instead of a `float` to make it simple. This means that the minimum transaction is a rupee.

2.4.3 Processing a Structure Variable

The structure variable declaration is of no use unless the variables are assigned values. Here, each member has to be specifically accessed for each structure variable. For example, to assign the account number for variable `a1`, you have to specify as follows:

```
a1.number = 0001;
```

There is a dot operator between the structure variable name and the member name or tag.

Suppose, you then want to assign account no.2 to `a2`, it can be assigned as follows:

```
a2.number = 2;
```

If you want to know the address where `a2` number is stored, you can use

```
printf ("%u", &a2.number);
```

This is similar to other data types. The structure is a complex data type and therefore, you have to indicate which structure variable to which the member belongs as; otherwise, the number is common to all the structure variables, such as `a1, a2, a3`, etc., Therefore, it is necessary to be specific. Assuming that you want to get the value from the keyboard, you can use `scanf()` as follows:

```
scanf ("%u", &a1.number);
```

You can also assign initial values directly as follows:

```
struct account a1 = {0001, "Vasu", 1000};
struct account a2 = {0002, "Ram", 1500};
```

All the members are specified. This is similar to the declaration of initial values for arrays. However, note the semicolon after the closing brace. The `struct a1` will, therefore, receive the values for the members in the order in which they appear. Therefore, you must give the values in the right order and they will be accepted automatically as follows:

```
a1 . number = 0001
a1 . name   = Vasu
a1 . balance = 1000
```

Note too that if the initial values are assigned as above, inside a function, they will be treated as `static` variables. If they are declared before `main`, they will be treated as global variables.

Let us write a program to create a structure account, open 2 accounts with initial deposits in the accounts. Deposit Rs 1000 to Vasu's account, withdraw 500 from Ram's account and print the balance. The following example demonstrates the above.

/*Example 2.12 - structures*

```
#include<stdio.h>
int main()
{
    struct account
    {
        unsigned number;
        char name[15];
        int balance;
    };
    static struct account a1= {001, "VASU", 1000};
    static struct account a2= {002, "RAM", 2000};
    a1.balance+=1000;
    a2.balance-=500;
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a1.number, a1.name, a1.balance);
    printf("A/c No:=%u\t Name:=%s\t balance:=%d\n",
```

```
      a2.number, a2.name, a2.balance);
      return 0;
  }
```

Result of the program

```
A/c No:=1    Name:=VASU   Balance:=2000
A/c No:=2    Name:=RAM    balance:=1500
```

A simple program was written for a bank transaction. For a deposit, we write

```
     a1 . balance = a1 . balance + 1000 ;
```

Therefore, the balance is updated. Similarly, when an amount is withdrawn, the balance is adjusted. However, in practice, the user cannot write a program for each credit and deposit. You will develop a program soon which will not require the user to do programming.

User Defined Data Type

A structure is also a data type. It is not a basic type defined in C language like `int, float`, etc. But, it is defined by the programmer. Hence, it is called a user defined data type.

2.4.4 Comparison of Structure Variables

In C programming language, a structure is a collection of different datatype variables, which are grouped together under a single name. The general form of a structure declaration and initialization is as follows:

```
datatype member1;
struct tagname{
   datatype member2;
   datatype member n;
};
```

In the above program code, `struct` is a keyword. The `tagname` specifies the name of the structure.

`member1, member2` specifies the data items that are used to create the structure.

For example,

```
struct book{
   int pages;
   char author [30];
   float price;
};
```

Structure Variables

Following are three methods for declaring structure variables.

First Method

```
struct book{
    int pages;
    char author[30];
    float price;
}b;
```

Second Method

```
struct{
    int pages;
    char author[30];
    float price;
}b;
```

Third Method

```
struct book{
    int pages;
    char author[30];
    float price;
};
struct book b;
```

Initialization and Accessing of Structures: The link between a member and a structure variable can be established by using a member operator or a dot operator. Initialization can be done using the following methods:

First Method

```
struct book{
    int pages;
    char author[30];
    float price;
} b = {100, "balu", 325.75};
```

Second Method

```
struct book{
    int pages;
    char author[30];
    float price;
};
struct book b = {100, "balu", 325.75};
```

Third Method: It is done by using a member operator.

```
struct book{
    int pages;
    char author[30];
    float price;
} ;
```

```
struct book b;
b. pages = 100;
strcpy (b.author, "balu");
b.price = 325.75;
```

Example of the C program for the comparison of structure variables.

Example 2.13

```
struct class{
    int number;
    char name[20];
    float marks;
};
main(){
    int x;
    struct class student1 = {001,"Hari",172.50};
    struct class student2 = {002,"Bobby", 167.00};
    struct class student3;
    student3 = student2;
    x = ((student3.number == student2.number) &&
    (student3.marks == student2.marks)) ? 1 : 0;
    if(x == 1){
      printf("\nstudent2 and student3 are same\n\n");
      printf("%d %s %f\n", student3.number,
       student3.name,
       student3.marks);
    }
    else
    printf("\nstudent2 and student3 are different\n\n");
}
```

Output of the program

When the above program is executed, it gives the following output after doing comparison of structure variables.

```
student2 and student3 are same
2 Bobby 167.000000
```

2.4.5 Array of Structures

Let us now create an array of structures for the account. This is nothing but an array of accounts, declared with size. Let us restrict the size to 5. The records will be created by using keyboard entry. The program is given below:

/*Example 2.14 – To demonstrate structures*

```
#include<stdio.h>
int main()
{
```

```
struct account
{
    unsigned number;
    char name[15];
    int balance;
}a[5];
  int i;
  for(i=0; i<=4; i++)
  {
    printf("A/c No:=\t Name:=\t Balance:=\n");
    scanf("%u%s%d", &a[i].number, a[i].name,
&a[i].balance);
  }
  for(i=0; i<=4; i++)
  {
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a[i].number, a[i].name, a[i].balance);
  }
  return 0;
}
```

Result of the program

```
A/c No:=   Name:=              Balance:=
1          suresh              5000
A/c No:=   Name:=              Balance:=
2          Lesley              3000
A/c No:=   Name:=              Balance:=
3          Ahmed        5500
A/c No:=   Name:=      Balance:=
4          Lakshmi             10900
A/c No:=   Name:=              Balance:=
5          Thomas              29000
A/c No:=1   Name:=suresh  Balance:=5000
A/c No:=2   Name:=Lesley  Balance:=3000
A/c No:=3   Name:=Ahmed   Balance:=5500
A/c No:=4   Name:=Lakshmi Balance:=10900
A/c No:=5   Name:=Thomas  Balance:=29000
```

The structure array has been declared as a part of structure declaration as a[5]. The individual elements of the 5 accounts are scanned and printed in the same order. Note that when you scan a name, you do not give the address but the actual name of the variable as in a[I].name, since it is a string variable. Remember this uniqueness. This program basically gets the 5 structures or records pertaining to 5 account holders. Thereafter, the details of the 5 accounts are printed using the for statement. The first half of the result was typed by the user and the last 5 lines are the output of the program.

2.4.6 Structure Elements passing to Functions

Structures can be copied individually, member wise as well as at one go.

For example, let a3 and a1 be struct account. The following are valid:

```
a3.number = a1.number;
a3.balance = a1.balance;
```

Here, the members of a1 are copied into a3, one by one.

You can also write a3=a1; when all the elements of a1 will be copied to a3. The latter coding can be used if all the elements are to be copied and the former, if some members are to be copied selectively.

Note that structures cannot be compared as: for example if (a4 == a2). This is not a valid operation.

2.4.7 Structure Passing to Functions

Passing each member of the structure is a tedious job. The entire structure can instead be passed to the function making for easy handling. The above example can be altered by passing an entire structure, as follows:

/*Example 2.15 – Passing entire structure to function*/

```c
#include<stdio.h>
struct account
{
  unsigned number;
  char name[15];
  int balance;
};
int main()
{
    static struct account a1={001, "Vasu", 1000};
    struct account credit(struct account x);
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a1.number, a1.name, a1.balance);
    a1=credit(a1);
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a1.number, a1.name, a1.balance);
    return 0;
}
struct account credit(struct account y)
{
    int x;
    printf("enter deposit made");
    scanf("%d", &x);
    y.balance+=x;
    return y;
}
```

Result of the program

```
A/c No:=1    Name:=Vasu   Balance:=1000
enter deposit made 6700
A/c No:=1    Name:=Vasu   Balance:=7700
```

If you want to pass a structure, the called function should also know the structure and hence, the structure has to be declared before the `main` function. Therefore, structure `account` has been declared as a global structure. The function `credit` is declared with return data type `structure` as follows:

```
struct account credit(struct account x);
```

Thus, you pass and `return struct account`. Then, credit is called by simply passing structure `a1`. In the called program, the deposit is added to the balance and updated. This is returned to the `main()` where the updated record is printed.

2.4.8 Structure within Structure

You have been nesting `if` statement and loops so far. You can now create structures within structures. Here, a structure defined earlier can become a member of another structure. For example, you can create a structure called deposit using other data types and structure account. The declaration of the basic structure should precede the desired structure as follows:

```
struct account
{
    Unsigned number ;
    char name [15];
    int balance ;
};
struct deposit
{
    struct account ac ;
    int amount;
    int years;
};
```

You can write a program to demonstrate the concept.

/*Example 2.16 – Structure within structure*/

```
#include <stdio.h>
int main()
{
    struct account
    {
        unsigned number;
        char name[15];
        int balance;
    };
    struct deposit
```

```
   {
      struct account ac;
      unsigned amount;
      int years;
   }d2;
   static struct deposit d1={001, "VASU", 1000, 50000, 3};
   d2=d1; /*structure copy*/
   printf("A/c      No:=%u\t      Name:=%s\t
Balance:=%d\tdeposit=%u\tterm=%d\n",
   d2.ac.number, d2.ac.name, d2.ac.balance, d2.amount,
d2.years);
return 0;
}
A/c No:=1    Name:=VASU    Balance:=1000 deposit=50000
term=3
```

You have created the structure `account`. Then, you have created another structure `deposit`. In the deposit structure, `struct account ac` is one of the members and 2 more members, amount and years have been declared.

Next structure `deposit d1` is initialized. The first 3 elements pertain to the members of `struct account` and the last two for `amount` and `years`, respectively.

Now `d1` is copied to `d2` in a simple manner and the deposit details of `d2` are printed.

Whenever members of included structures are accessed, you will find two dots, instead of the usual one dot. This is essential since `d1.name` is invalid. Since name is in `ac`, you have to access it as `d1.ac.name`.

However, nesting can be up to any level. You can create one more level of nesting as shown:

```
struct account
{
   unsigned number;
   char name [15];
   int balance;
};
struct deposit
{
   struct account ac;
   int amount;
   int years;
};
struct loan
{
   struct deposit dep;
```

```
int amount;
char date[10];
};
```

You see that `struct deposit` is included as a member of loan.

Let us declare:
```
struct loan loan1;
```

Now, to access loan amount, we have to specify:
```
loan1.amount
```

To access deposit amount, we have to specify:
```
oan1.dep.amount
```

To access the balance, we have to specify:
```
loan1.dep.ac.balance
```

Therefore, usage of the same variable name `amount` has not resulted in a conflict since it has to be seen in which context it is defined. Thus, structures can be used within structures without difficulty.

2.4.9 Structure Containing Arrays

You saw some members of structures being arrays. You used them to represent an array of characters. You can also have integer, `float` arrays as members of structures. For example,
```
struct fixed_deposit
{
unsigned Ac_Number;
char name[25];
double deposit[4];
}rama;
```

Here, you have defined a structure `fixed_deposit` and declared a variable `rama` of the same type. In the structure definition, you have a member `deposit` as an array of double of size 4. This is called array within structure.

You can access the first deposit of `rama` by the following:
```
rama.deposit[0]
```

The last deposit of `rama` can be accessed by:
```
rama.deposit[3]
```

Application of Structures

Structures as you know, can be used for database management. They can be used in several applications, such as libraries, departmental stores, banking, etc. Structures are also used in C++. The syntax of structures is similar to classes in C++. Structures contain data but classes contain data and functions.

Structures are also used in a variety of other applications, such as:

- Graphics
- Formatting floppy discs

- Mouse movement

- Payroll

Thus, a structure is a very useful construct.

2.4.10 Union

Union is a variable, which holds at a common assigned area different data types of varying sizes at different points in time. Assume that a program, at different points in time of execution uses a `double,` a `float,` an `integer` and a `string.` In the normal course, you would have to allocate memory space for each data type. Assuming that you want to use only one of them at any time and if you do not mind losing the values, you can save a lot of memory space by declaring a common area for storing them. If you use dedicated memory for each variable, the space would remain unutilized most of the time during program execution. This common storage area can be declared as a `union` as shown here:

```
union uname
    {
        double d;
        float f;
        char s[];
        int i;
    } un1;
```

See the resemblance between `structure` declaration and `union` declaration. The common properties are :

(i) They can have a name optionally, such as `uname.`

(ii) They can contain members of varying data types.

(iii) The declarations end with a semicolon.

(iv) These `union` members can be accessed in the same way as structure members, as shown:

```
union_name.member or union_pointer->member
```

(v) A `union` can be assigned to another `union,` such as

```
un2=un1;
```

where the structure of `un1` along with its members are copied to `un2.`

However, the difference is:

(i) The memory size of the structure variable is the sum of the sizes of its members, whereas in `union,` it is the largest size of its members.

(ii) It is the programmer's responsibility to keep track of which type is currently in use unlike in structure where no member is lost.

(iii) In structures, all members can be initialized, whereas a `union` can only be initialized with a value of the type of its first member.

A program using `union` to store either an `int` value or `float` value is given as follows:

```
/* Example 2.17 – Demonstrate union */
#include <stdio.h>
#include <conio.h>
union sel
{    int n;
     float f;
};
int main()
{    union sel m1;
     void printval(union sel *m1,char type);
     char type = 'p';
     char cont = 'y';
     while(cont=='y')
     {
         printf("\nWant to enter Integer Or Float (i/f):");
         type=getche();
         if(type=='i')
         {
             printf("\nEnter Integer Value :");
             scanf("%d",&m1.n);
         }
         else
         {
             printf("\nEnter Float Value :");
             scanf("%f",&m1.f);
         }
     printval(&m1,type);
     printf("\nWant to continue- enter (y/n):");
     cont=getche();
   }
     return 0;
}
void printval(union sel *m1,char type)
{
     if(type=='i')
         printf("Integer Value Is %d\n",m1->n);
     else
         printf("Float Value Is %f\n",m1->f);
}
```

Result of the program

```
Want to enter Integer Or Float (i/f):i
Enter Integer Value :456
Integer Value Is 456
Want to continue- enter (y/n):y
Want to enter Integer Or Float (i/f):3
Enter Float Value :300.30
Float Value Is 300.299988
Want to continue- enter (y/n):n
```

The union `sel` is declared before `main`, with two members `int n` and `float f`. A function `printval` is also declared. Depending upon whether the user wants an `integer` value or `float` value, type is set to `i` or `f`. If type is `i`, integer value is received and if it is `f`, a `float` value is received. The value received is printed in the function `printval`. Note carefully how the pointer to `union` is declared in the function prototype and header. You have to declare `union` whenever you pass a `union` to a function. It is declared as `union sel * m1`. As you would have declared `int * i`, the type here is `union sel`.

If you have perused the result, you would find that the program asks for `float` value, although 3 has been typed instead of `f`. It is because of the design of the program. It will ask for `float` value when a character other than `i` is typed. You can take this as an exercise to correct the program to ask for float value only when `'f'` is typed.

Now, consider another example.

In the case of an employee database, you would like to store either the father's name (in the case of men and unmarried women), the husband's name in the case of married women or the guardian's name. This can be represented as a `union` as shown below:

```
Union guardian
{
    char father [10];
    char husband [10];
    char guardian [10];
} e ;
```

You have defined a `union guardian` of employees. By using this, you can save memory space. The memory required will be 10 bytes. Had you considered this as a structure, you would have used 30 bytes and all three variables will be used. This is not required since only one value will be present at any time and `union` is useful in such cases.

You can now define a structure for an employee database with `union` embedded. You have to declare `union` above structure since `union` will be one of the members of the structure. You can define it as follows :

```
union guardian
{   char father [10];
    char husband [10];
    char guardian [10];
} u;
struct employee
{   char name [15];
    float basic;
    char birthdate [10];
    union guardian u;
} emp [2] ;
```

You know how to refer to members of structures, for example, you can refer to the name of the employee's father as: `emp[0].u.father`.

2.4.11 Structure Pointers

You know how to declare pointers to various data types and arrays. Similarly, pointers to structures can also be declared. For instance, in the case of the structure `account,` you can declare a pointer as shown:

```
struct account a1 = { 1, "Vasu", 1000 };
struct account * sp;
struct sp = &a1 ;
```

Now `sp` is a pointer to a structure. Therefore, if you assume structure as another basic data type, declaring it as an array and declaring it as a pointer, etc., follow the same rules. Structure is in fact a user-defined data type.

Access to individual elements of a structure defined in the form of a pointer is similar, but instead of dot we use an arrow pointer `'-' >`. Arrow pointer is formed by typing minus followed by the greater sign. However, on to the left of the arrow operator there must be a pointer to a structure. The following example will clarify the point:

```
/*Example 2.18.
/*Structure pointers */
#include<stdio.h>
main()
{
    struct account
    {
        unsigned number;
        char name[15];
        int balance;
    }a5;
    static struct account a1={001, "VASU", 1000};
    struct account *sp;
    sp=&a1;
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    sp->number, sp->name, sp->balance);
    a5=*sp;
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a5.number, a5.name, a5.balance);
}
```

The output of the program is:

```
A/c No:=1  Name:=VASU Balance:=1000
A/c No:=1  Name:=VASU Balance:=1000
```

In this program, `sp` is a pointer to `structure account,` and therefore `sp` is assigned the address of structure `a1`. Then the contents of structure `*sp`

are printed. The elements of `*sp` are copied to `a5` and then printed (to demonstrate copying of structures). Note the difference between the notations when accessing elements of a structure and a structure pointer.

Check Your Progress

16. Define a structure.
17. What is a union?

2.5 POINTERS: DECLARATION AND INITIALIZATION

A pointer is a variable that contains the address of another variable. As you know, any variable has the following four properties:

(a) Name

(b) Value

(c) Address

(d) Data Type

For example, consider the following declaration of a simple integer:

```
int var = 10;
```

The name here is `var`, and its value is `10`. Its address is not declared here, since we want to give flexibility to the compiler to store it wherever it wants. If we specify an address, then the compiler must store the value at the same address. Specifying the actual address is carried out during machine language programming. However, this is not required in High Level Language (HLL) programming, and by printing the value of `&var`, we can find out the address of the variable. When the statement to find the address is executed at different times, different addresses will be printed. What is important is that the compiler allocates an address at run-time for each variable and retains this till program execution is completed. This is not strictly so in the case of `auto` variables. The compiler forgets the address of a variable when the program comes out of the block in which the variable is declared. At this point you may also recall that in the case of function declarations in the calling function, the compiler does not allocate memory to the variables in the declaration. That is the reason why the parameters in the declaration part are not recognized in the calling function. It is only a prototype.

The fourth feature of a variable is its data type. In the above example, `var` is an integer. A pointer has all the four properties of variables. However, the data type of every pointer is always an integer because it is the value of the memory address. Memory addresses are integers. They cannot be floats or any other data types. They may point to an integer or a float or a character or a function, etc. They have a name. They have a value. For example, the following is a valid declaration of a pointer to an integer.

```
int * ip;
```

Here `ip` is the name of a `pointer`. It points to or contains the address of an integer, which is the value. It will also be stored in another location in the memory like any other variables. The pointer itself is an integer even though it is not declared as such.

Pointers and Variable Length Arguments List

To understand the concept of pointer and variable length argument list, let us implement a `minprintf()` function which is a version of `printf()` library function. In the following example, `...` is used in `void minprintf()` function to indicate the number and types of arguments. These arguments may vary. The declaration `...` only appears at the end of an argument list. In the following example, `va_list` declares a variable that will refer to each argument in turn. Function `va_arg` returns one argument to the `va_start(ap)`. Macro `va_start` initializes `ap` which points to the first unnamed argument. Function `va_arg` uses a type name to determine the value and its data type. Function `va_end` is used when cleanup operation is necessary. It must be called before the program returns.

Example 2.19

```
#include <stdarg.h>
void minprintf(char *format, ...)
{
 va_list ap; // points to each unnamed argument in turn
 char *p, *sval;
 int ival;
 va_start(ap, format);
 // Make ap point to first unnamed argument
for(p = format; *p; ++p)
{
  if (*p != '%')
   {
    putchar(*p);
    continue;
   }
switch (*++p)
{
 case 'd':
 ival = va_arg(ap, int);
 printf("%d", ival);
   break;
 case 's':
for (sval = va_arg(ap, char *); *sval; ++sval)
 {
 putchar(*sval);
 }
```

```
   break;
    default:
   putchar(*p);
    }
    }
 va_end(ap); // Cleanup process
 }
```

Header file <stdarg.h> handles following variable argument list.

```
 #include <stdarg.h>
 void va_start(va_list ap, argN);
 void va_copy(va_list dest, va_list src);
 type va_arg(va_list ap, type);
 void va_end(va_list ap);
```

The <stdarg.h> header file contains a set of macros which allows portable functions that accept variable argument lists to be written. The argument va_list is defined for variables used to traverse the list. The va_start() macro is invoked to initialize ap to the beginning of the list before any calls to va_arg().The va_copy() macro initializes dest (destination) as a copy of src (source) when the va_start() macro had been applied to dest followed by the va_arg() macro as had previously been used to reach the present state of src.

2.5.1 Pointer Notation and Accessing Variable

At this point you must note another way of representing the elements of the array.

The address of the 0th element is stored in at location p2 or address p2 + 0. The element with a subscript 1 of the array will be at a location one above or at p2 + 1. Thus, the address of the nth element of this array *p2 will be at address p2 + n. If you know the value of the pointer variable, i.e., the address of pointer, then it would be easy to express its value. The value of the integer stored at the nth location can be represented as * (p2 + n) just as the value at address (p2 + 0) is * (p2 + 0) or * p2. This notation is, therefore, quite handy.

The if statement compares maxp with * (p2 + j) or p2 [j] or array [j]. You will easily understand the logic as to how we get the maximum or greatest number in * (p2 + j). At the end of the iterations, * (p2 + j), which is stored at location p2 + j, contains the maximum value in the array and therefore, we are returning an address or reference to the called function. In this example, (p2 + 3) is the address of the greatest number in the array. After return from the function, max gets the value of p2 + j. In the printf *max, which is the value stored in max, is printed. We have already defined max as a pointer to an integer in the main function. Thus, the function fung() returns the address of a value or a pointer, and by returning the address, the value is retrieved automatically by the main function.

2.5.2 Arrays and Pointers

Examples above involved arrays and pointers. The usage of a pointer made the passing of an array to a function a simple task. You define array as an integer. However, in order to pass an address, the prototype was defined with a pointer argument int * p1. This means an address will be passed to the function while calling it. No distinction was made between a simple integer variable and an integer array. int * p1 can be a single valued integer or an array. This is possible because arrays are stored contiguously in the memory. If the address of the 0th element is known, and the data type is known, it would not be difficult to calculate the address of any element in the array. Therefore, it is enough if the address of the 0th element is passed to a function. *p1 refers to the address of a variable or to the 0th element of an array. Note the flexibility of arrays in 'C' and how intelligently the language uses pointers.

While calling a function, the address has to be passed and this is achieved in the above example by passing & array[0].

In the called program, *p2 is treated as an array without any additional efforts. By adding the index to p2, you get the address of the various elements in the array. Getting value is achieved by placing * before the address. Now, look at another example using arrays and pointers as follows:

```
/*Example 2.20
/*Passing an array of integers to function – method2*/
#include<stdio.h>
main()
{
    int array[]={10, 20, 30, 40, 50};
    int *a;
    void pass(int *a, int k);
    a=&array[0];
    pass(a, 4);
}
void pass(int *b, int j)
{
    int k=0;
    while (k <= j)
    {
        (*b)=(*b)/2;
        printf("Value %d @ address %d\n", *b, b);
        k++;
        b++;
    }
}
```

The output of the program is:

```
Value 5 @ address 8694
Value 10 @ address 8696
Value 15 @ address 8698
```

```
Value 20 @ address 8700
Value 25 @ address 8702
```

Here, `*a` has been declared as a pointer to integer. The variable `a` has been assigned the address of the `0th` element of the array. Now the function call is made using `pass(a, 4)`; again `a` is the address of the `variable array [0]`.

In the function pass, `b` received the address of `a`, and the next element of the array is accessed each time by incrementing the address `b`. Note that the values of elements in the array are divided by 2 in the called function. As you become familiar with pointers you can write programs very easily.

2.5.3 Pointer Expressions

An integer operand can be used with a pointer to move it to a point/refer to some other address in the memory. Consider an int type pointer as follows.

```
                            65494
    int *mptr;         mptr ┌──────────┐
                            └──────────┘
```

Assume that it is allotted the memory address 65494. Increment its value by 1 as follows:

```
    mptr++; or ++mptr;
    mptr = mptr + 1; or mptr += 1;
```

Note: ++ and −− operators are used to increment and decrement a pointer and are commonly used to move the pointer the next location.

Now the pointer will refer to the next location in memory with the address 65496. C language automatically adds the size of the **int** type (2 bytes) to move the pointer to the next memory location as follows.

```
    mptr   = mptr + 1;
           = 65494 + 1 × sizeof(int)
           = 65494 + 1 × 2
           = 65496
```

Similarly any integer can be added or subtracted to move the pointer to any location in Ramdom Access Memory (RAM). But the resultant address is dependant on the size of the data type of the pointer. The step in which the pointer is increased or reduced is called **scale factor**. Scale factor is nothing but the size of the data type used in a computer. We know that in a PC, the size of **float** is 4, **char** is 1, **double** is 8, and so on. Consider another example.

```
        float *xp;
```

(Assume its address is 63498)

```
        xp = xp + 5;
```

It will move the pointer to the address 63518 which is obtained as follows.

```
        = 63498 + 5 × sizeof(float)
        = 63498 + 5 × 4
        = 63518
```

Note that any arithmetic operation, such as addition or subtraction can be performed on a pointer. A great care is required to understand the resultant memory address obtained in the operation.

Pointer Arithemtic

The following are the examples of pointers:

Example 2.21

```
int dat = 100;
int * var;
var = &dat;
```

Here `dat` is an integer variable. Its value is 100; its name is `dat`; it will be stored in the memory in a location with an address.

The next declaration means that `var` is a pointer to an integer and is also a variable. It is an integer and will be stored at a location in the memory with an address. The value of `var` is the address of the integer variable it points to. We do not know as yet which integer it points to. It can, however, be made to point to any integer we like by a proper declaration.

Now look at the next assignment. The variable `var` is assigned the value of `&dat`. This means `var` has the same value as the address of `dat`. By taking into consideration the previous statements we can conclude that `var` is a pointer and it points to `dat`.

Now if you specify `dat` or `* var`, they point to the same value 100. Similarly, if you specify `&dat` or `var`, it is the address, or to be precise, the starting address of `dat` or `* var`.

Example 2.22

```
int * var
* var = 100
```

The above statements declare `var` as a pointer to an integer and later propose to assign 100 to the integer variable. We do not know or do not want to make public the name of the variable. However, we can always access the variable as `* var`. This works well in Borland C++ compilers, but could lead to run-time errors in other compilers.

Both the statements cannot be combined into one as int `* var = 100`. This will be flagged as an error even in the Borland C++ compiler. Therefore, it is not possible to combine both the declaration and assignment so far as a pointer variable and an integer constant are concerned. It would be safer to make `var` point to another variable as given in Example 2.21.

Example 2.23

```
int * var;
* var = 100;
```

Note: If this gives an error while compiling or running the program, modify this as in Example 4.

What will be the value of the output of the following statements after the execution of the following statements?

```
printf ("%d", * var);
printf ("%d", (* var) ++);
printf ("%d", * var);
printf ("%d", var);
```

You can easily guess that the first `printf()` will give the value of `* var` as `100`.

What is the significance of the parentheses and the increment operation in the second statement? As the bracket or parentheses has precedence over other operators, the value of `* var` will be printed as `100`. After printing, it will be incremented as `101` and because the increment is postfix, the value of `* var` after the execution of the statement will be `101`. The next statement will confirm this when it prints `101`.

The fourth `printf()` case prints the address of `var`. It will print `1192` when the program is executed. You are unlikely to get the same address on execution. The location where the variable is stored will not vary till the execution of the program is completed. It you try the program again, you will get a different address. Do not worry. It does not affect our work. However, you may note down the value and substitute it for the values mentioned here for understanding the concept of pointers.

Remember to enclose the pointer variable within parenthesis as given in the example. The postfix of the increment operator enables increment of the variable after printing.

Example 2.24

After the execution of the above fourth `printf()` statement, we execute the following statements:

```
printf ("%d", * var ++);
printf ("%d", * case);
```

What happens? The value of `* var`, i.e., `101`, is the first to be printed out and then `var` will be incremented, i.e., instead of incrementing the value as desired, the address is incremented, and therefore, `var` now points to the next location. Remember `var` was pointing to 1192. Will it go to 1193? No. Since `var` is an integer, `1192` and `1193` (2 bytes) are already occupied. Hence, `var` points to 1194. The next statement prints the value, of `* var`. We had stored 101 in location 1192. We do not know what is contained in 1194. Hence, the next `printf()` will print garbage value. Note carefully what had happened. We wanted to increment `* var` in the first statement of Example 2.22. However, the compiler has assumed that we wanted to increment the pointer, which underlines the importance of parenthesis. Also note that whenever you use the postfix notation, the postfix operation is effective only after the execution of the statement.

Is everything lost now? Can we not go back to address 1192? Yes, you can, as the following indicates. Note that the following statements are executed in continuation of all the above statements.

Example 2.25

```
printf ("%d",var);
printf ("%d", --var);
printf ("%d", *var);
```

The first statement in this example prints the address of the location in the memory pointed to by var. As expected, the pointer is at the location 1194. The second statement carries out two operations in the sequence given below:

(a) Decrement the pointer.

(b) Print the new address.

Decrementing takes place before printing because it is a prefix operator. Now it prints 1192 and the original address is restored. Now the third statement prints the value of *var or the value stored in location 1192, i.e., 101.

Note that prefix carries out the increment or decrement before printing or any desired operation, whereas postfix does that after printing. Note that we are discussing the fundamentals of pointers and they should be understood clearly before you proceed further.

We now have 101 stored in address 1192 and pointed to by var. Let us see what happens on execution of the following statements in continuation.

Example 2.26

```
printf ("%d", ++ (*var));
printf ("%d",var);
```

These statements are perfectly correct. *var is incremented and the new value printed in the first statement. Therefore, 102 will be printed. Has the address been changed? No. Hence, the second statement will print the address as 1192.

Can we increment and decrement addresses? Yes, as the following indicates:

Example 2.27

```
printf ("%d", var ++);
printf ("%d",var);
```

What will be printed in the first statement above, 1192 or 1194? It will be 1192, because incrementing var will take place after the first printf(). Obviously, the second statement will print the address incremented after the previous printf(), viz. 1194. Let us not lose track, but get back to the old address, and try prefixing increment/decrement operators to the address.

Example 2.28

```
var --;                      (a)
printf ("%d",var);           (b)
printf ("%d", ++var);        (c)
printf ("%d", --var);    (d)
printf ("%d",var);           (e)
printf ("%d", *var)          (f)
```

Before the execution of the first statement in this example, you have:

`var = 1194`

Location 1192 contains 102.

Let us now analyse the execution statement-wise.

- Decrements `var` to 1192.

- Confirms that `var` is 1192 indeed.

- `++ var`, increments `var` and then prints as 1194.

- `-- var` decrements `var` and then prints as 1192.

- Confirms `var` is 1192.

- The value in `var` is 102.

    Now the concepts are becoming clearer. Let us carry out one more example. Assume that all the above statements were executed and the following are now executed:

Example 2.29

```
printf ("%d", * (var ++));                    (g)
printf ("%d", * (- - var));                   (h)
printf ("%d", var);                           (i)
printf ("%d", * var);                         (j)
```

- Here `* var` is printed as 102 because of the postfix operator. Then `var`, i.e., the address is incremented to 1194.

- Here because of the prefix, `var` is decremented to 1192 and then the value at 1192, i.e., 102 is printed.

The next two statements confirm this.

Now you are familiar with the intricacies of pointers, prefix, suffix and parenthesis.

For your convenience, the program involving all these statements and the output is given below:

```
/*Example 2.30 */
/* Program on pointers*/
#include <stdio.h>
main()
{
    int * var;
    int a =100;
    var = &a;
    printf("Value of * var=%d\n", *var);
    printf("Value of (*var)++=%d\n", (*var)++);
    printf("Value of * var=%d\n", *var);
    printf("Address var=%d\n", var);
    printf("Value of *var++=%d\n", *var++);
    printf("Value of * var=%d\n", *var);
    printf("Address var=%d\n", var);
```

```
printf("Original address var again=%d\n", —var);
/*original address restored*/
printf("Value of * var=%d\n", *var);
printf("Value of ++(*var)=%d\n", ++(*var));
printf("Address var=%d\n", var);
printf("Address var++=%d\n", var++);
printf("Address var=%d\n", var);
var—;
printf("Address var after decrementing=%d\n", var);
printf("Address ++var=%d\n", ++var);
printf("Address – –var=%d\n", – –var);
printf("Address var=%d\n", var);
printf("Value of * var=%d\n", *var);
printf("Value of * (var++)=%d\n", * (var++));
printf("Value of * (– –var) = %d\n", * (– –var));
printf("Address var=%d\n", var);
printf("Value of * var=%d\n", *var);
}
```

The output of the program is:

```
Value of * var=100
Value of (*var)++=100
Value of * var=101
Address var=9106
Value of *var++=101
Value of * var=9108
Address var=9108
Original address var again=9106
Value of * var=101
Value of ++(*var)=102
Address var=9106
Address var++=9106
Address var=9108
Address var after decrementing=9106
Address ++var=9108
Address – –var=9106
Address var=9106
Value of * var=102
Value of * (var++)=102
Value of * (– –var) = 102
Address var=9106
Value of * var=102
```

2.5.4 Pointers and One Dimensional Arrays

If you now declare,

```
int a [ 5 ];
ip = &a[0];
```

You have defined an array of integer a with 5 elements. When you assign the address of a[0], i.e., the 0th element of a to ip, ip will point to the array. The system will automatically assign addresses for the other elements in the array by noting the data type of the array and the size occupied by each element.

Finding the Greatest Number in an Array

The example below also explains the concept of a function returning a pointer.

```
/* Example 2.31

/*To find the greatest number in an array*/
#include <stdio.h>
main()
{
    int array[]= {8, 45, 5, 131, 2};
    int size=5, * max;
    int* fung(int *p1, int size);/*function returns pointer
to int*/
    max=fung(&array[0], size); /*max is a pointer*/
    printf("%d\n", *max);
}
int * fung(int *p2, int size)
{
    int i, j, maxp=0;
    for (j=0; j<size; j++)
    {
        if (* (p2+j) > maxp)
        {
            maxp=* (p2+j);
            i=j;
        }
    }
    return (p2+i); /*pointer returned*/
}
```

The output of the program is:

```
131
```

The called function returns the address of the greatest number in the array. Look at the function declaration. The function returning a pointer is indicated by the following (a star mark between the return data type and function name).

```
int * fung(...)
```

The address of `array [0]` is received by `fung ()` and stored in `*p2`, or in other words, `p2` points to the 0th location of the array.

2.5.5 Malloc Library Function and Calloc Library Function

The dimension or array size declaration of an array is an important subject. The array dimension is to be declared before compiling. For example, some valid declarations are:

```
char name[25];
int mark[40];
```

You have not come across any problem since you were initializing the arrays and hence the array size was known. If you were to get the array elements at run-time, sometimes you could give either a lesser number of elements or more elements. In the former case, garbage values will be stored in the empty spaces in the array misleading the user. In the latter case, the elements will be lost. To avoid this problem, you may think that you can specify `marks [n]` and give the value of `n` later at run-time. However, this will not work, and the compiler will force you to give the actual dimension. Hence, dynamic memory allocation is useful. The functions `malloc ()` and `calloc ()` serve to specify the actual dimension at run-time and hence enable memory allocation dynamically. Next time when you execute the program, you can specify any value for `n`. It will definitely work.

The functions `malloc ()` and `calloc ()` allot memory dynamically. The specifications are given below in the following statement.

```
void * malloc (n*size n );
```

It returns the pointer to n bytes of the memory or NULL if allotment is not possible. Since it returns a pointer you have to specify the array as a pointer variable as given below:

```
int *b ;  / * array declared * /
b = (int *) malloc (x * 2) ;    / * x is the array size * /
```

Similarly the `calloc ()` is defined as `void * calloc (sizen, size_size)`. Here the number of arguments is two. The first one is the array size and the second one is the bytes occupied per element, i.e., the storage space required for the data type. The function returns a pointer allocating space for an array of size `sizen`, of data type size `size`, but the array contents will be initialized to zero. In `malloc ()`, the initial contents will be garbage values. You can use it as follows:

```
int * b;
b = (int *) calloc (x, 4);
```

Here 4 indicates the array is of type `float` and space is allocated to store x `floats`. When you use `calloc` or `malloc`, you must include `alloc.h`.

A program to demonstrate `malloc` and `calloc` is as follows:

```
/* Example 2.32.
/*To do string concatenation by using dynamic allocation of
memory*/
```

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
main()
{
    char *newstrcat(char *dest, char *src);
    char *name1, *name2;
    int n1, n2;
    printf("Enter size of 2 names:\n");
    scanf("%d%d", &n1, &n2);
    name1 = (char *) calloc(n1, 1);
    name2 = (char *) calloc(n2, 1);
    printf("Enter 2 names\n");
    scanf("%s%s", name1, name2);
    printf("New name:%s\n",newstrcat(name1,name2));
}
char *newstrcat(char *dest, char *src)
{
    char *w;
    int i,len,len1,cnt=0;
    len=strlen(dest);
    len1=strlen(src);
    w=(char *)malloc(len+len1+1);
    for(i=0;i<len;i++)
        w[cnt++]=dest[i];
    for(i=0;i<len1;i++)
        w[cnt++]=src[i];
        w[cnt]=0;
    return(w);
}
```

The output of the program is:

```
Enter size of 2 names:
4 4
Enter 2 names
Rama samy
New name:Ramasamy
```

calloc is used to allot memory to name1 and name2. The function newstrcat is called while printing. In the called function, malloc is used to allot space equal to the size of name1, name2 +1. The additional space is for placing NULL. The string is concatenated by copying one character at a time using two for statements. Finally, the concatenated string is returned to the main function where it is printed.

The memory allocated using `malloc`, `calloc` can also be freed, when it is no longer necessary by using the function `free ()`. For example, in the `main ()` of the above example, after the last statement you can add the following statements:

```
free(name1);
free(name2);
```

These statements will deallocate the memory space allocated to them after the job of printing the concatenated string is over. Thus, dynamic allocation of the memory according to the exact need and freeing it after use is quite useful for conserving the memory. This concept is used by professional programmers when they develop commercial software products. Memory saved is more than money saved in such product development.

2.5.6 Pointers and Multi-dimensional Arrays

We may represent a two-dimensional array conventionally as `a[i][j]`.

The elements of a two-dimensional array of size 3×3 can be represented symbolically as given below:

```
        a00     a01         a02
        a10     a11         a12
        a20     a21         a22
```

We can then visualize this as an array of arrays. Do not get puzzled. Each row is an array and we have three such arrays. Thus, a two-dimensional array can be considered as an array of one-dimensional arrays. Therefore, we can expect the following type of memory allocation by the system:

```
   0 1 2       0 1 2        0 1 2
 a[0][0]     a[1][0]   a[2][0]
```

This means all the elements of the `0th` row are stored contiguously. `a[0][0]` is stored first, followed by all the elements of the `0th` row. Next, the first row starts with the first element `a[1][0]`. At the end of the first row, the second row starts with `a[2][0]`. Try to visualize how the elements of a two-dimensional array are stored contiguously.

You know that

```
a[i] = * (a + i)
```

Where `a + i` is the address of the element.

You also know that

`a [0] = * a` and its address is stored in location `a`.

You can now transform a two-dimensional array into this convenient form. You can visualize `**b[0]`, `**b[1]` and `**b[2]` to represent the values of row first elements, which are stored at locations `*b[0]`, `*b[1]` and `*b[2]` as shown:

```
        0 1 2  0 1 2     0 12
 Address  *b[0]    *b[1]     *b[2]
```

`**` indicates `pointer` to `pointer`. This is acceptable since the row pointers in turn point to the column pointers. If you assume the row first elements

as pointers, then the addresses of elements of the two-dimensional arrays can be addressed by simple arithmetic.

`*b` points to the value in the one-dimensional array. Then `**b` refers to the value in the two-dimensional array. All elements of the two-dimensional array are stored contiguously. From the storage pattern we declare the following:

`**b` refers to the value at the 0th row of a two-dimensional array. Naturally the address of the 0th row will be `*b`.

`**(b + 1)` refers to the value of the 0th element in the 1st row. Its address will be `*(b + 1)`.

In general, the 0th element in ith row contains value `**(b + i)` and its address is `*(b + i)`.

You have only discussed about the 0th element in each row, i.e., where the row starts conceptually. How do you address the other elements? You know that an element `b[i][j]` is the `j`th element in the `i`th row.

You know the address of the 0th element. It is `*(b + i)`. Therefore, the address of the `j`th element in `i`th row will be `(*(b + i) + j)`. Note `j` is the offset. Therefore, the value at this location can be expressed as `*(*(b + i) + j)`. You have just added a star outside the address.

Now the address of the 2nd element in the 2nd row will be,

```
(*(b + 2) + 2)
```

Its value will be,

```
*(*(b + 2) + 2)
```

The address of the 0th element in the second row will be,

```
(*(b + 2))
```

The value will be,

```
*(*(b + 2))
```

Note the parentheses and `*`. To understand this clearly, execute the following program:

```
/*Example 2.33.

/*To understand pointers to two-dimensional.
array*/
#include<stdio.h>
main()
{
int i,j;
int a[3][3];
    printf("Enter the values of 3x3 matrix\n");
    for (i=0; i<=2; i++)
    for(j=0; j<=2; j++)
        scanf("%d", (*(a+i)+j));
        for (i=0; i<=2; i++)
        {
```

```
                              for (j=0; j<=2; j++)
                              {
                                  printf("Address=%d\n", (*(a+i)+j));
                                  printf("Value=%d\n", *(*(a+i)+j));
                              }
                          }
                      }
```

The output of the program is:

```
Enter the values of 3x3 matrix
00 01 02
10 11 12
20 21 22
Address=9098
Value=00
Address=9100
Value=01
Address=9102
Value=02
Address=9104
Value=10
Address=9106
Value=11
Address=9108
Value=12
Address=9110
Value=20
Address=9112
Value=21
Address=9114
Value=22
```

This example helps you to understand how a two-dimensional array is stored in the memory. The array has been declared in the normal way without pointers, but the array elements have been received, stored and printed using pointer notation. When you execute the above program, you can type the numbers in any one of the ways given below:

- Enter one integer at a time followed by Return.
- Enter all integers in a row with spaces between them and finally hit Return.
- Enter three integers in a row as the result of the program indicates.

Since it is a two-dimensional array you would like to input the value in the form of rows and columns and get the output in the same form. It would be better if the computer accepts the inputs at the given places and prompts us to do so as in the program given below:

Receiving Inputs at Chosen Points

```c
/*Example 2.34.
/*To accept and print in matrix form*/
#include <stdio.h>
#include <conio.h>
main()
{
    int a[3][2];
    int i,j;
    printf("Enter values of 3x2 matrix-\n");
    printf("Press Enter after each value");
    for (i=0; i<=2; i++)
    {
        for (j=0; j<=1; j++)
        {
            gotoxy(j*10+10,4+i*2);
            scanf("%d",(*(a+i)+j));
        }
    }
    for (i=0; i<=2; i++)
    {
        for (j=0; j<=1; j++)
        {
            gotoxy(j*10+40, 4+i*2);
            printf("%d", *(*(a+i)+j));
        }
    }
}
```

The output of the program is:

```
Enter values of 3x2 matrix
Press Enter after each value
  00   01      0   1
  10   11      10   11
  20   21      20   21
```

A library function `gotoxy(x, y)` has been used in the program. This function makes the cursor go to the point `(x, y)` on the screen. At the first iteration when both `j` and `i` are `0`, you will call `goto (10, 4)`, while scanning the values and call `goto (40, 4)`, while printing the values. Therefore, the first value will be scanned on the 4th line from the top and at the 10th character position. The first value will be printed at the same line and the 40th character position as the Result of the Program indicates. Therefore, you can direct the cursor to any position you like. Thus, `gotoxy()` will be quite useful in advanced programming.

This has been used both for `scanf()` and `printf()`. The system will scan inputs from those points and print outputs at the points specified. You can specify any point on the screen to take inputs and print outputs. When you execute the program the cursor will go to the specified position. After you have typed one value and entered the return key, the cursor will blink at the next point, giving the impression of inputting a matrix. After the values are given, the output also appears in the form of a matrix. The left pair of numbers is what was entered. Since the same vertical position, but a different horizontal position, has been specified in the program, the printout appears on the same lines, but to the right of the input values. Try the program and see for yourself.

Look at the address specification along with the `scanf()`. It is,

`(*(a + i) + j)`, which is equivalent to `&a [i][j]`.

Putting a star before this converts this to the value `a [i][j]`.

This program clearly illustrates the way to handle a two-dimensional array with pointers.

Although the array could have been declared as `**a`, it has been declared with the dimensions of the array. Since it does not know exactly the total number of elements in the former definition, garbage values may be stored in some more locations adjacent to the array elements, which can create problems. Some compilers will cause a run-time error, if the array has been declared as `**a`. Therefore, it would be better to specify the dimension of the array as given in the above two examples.

2.5.7 Arrays of Pointers

A pointer to a one-dimensional array can be denoted as `*b` and two-dimensional array can be denoted as `**b`. In this section, an array of pointers will be discussed.

Sorting Character Strings

As you know, names of students in a class can be denoted by a two-dimensional array like `b[50][20]`, with the second subscript denoting the width of the names and the first 50 denoting the number of students. There may be insufficient space for names longer than 20 characters, leading to truncation. If the name is short, it will lead to the wastage of space. By using an array of pointers, we can declare `char * name [50]` as an array to store the names of 50 students. The number of students in a class is definitely known. Here `name` is an array and points to character. Thus, it is an array of pointers. Actually what happens is that 50 addresses are stored in the `array name [50]`, `name [0]` corresponds to the address of the first name, `name [1]` to the address of the second name, and so on. Using this concept a program is developed to sort names. It is given below:

```
/* Example 2.35.
/*For sorting character strings */
#include <stdio.h>
#include <string.h>
#include <alloc.h>
```

```
#define N 5 /*5 names sorted*/
main()
{
    int ret,i,j,p=0;
    char *name[N], t[50];
    for(i=0;i<N;i++)
    {
        printf("Enter name size\n");
        scanf("%d", &p);
        name[i]=(char *)(malloc(p*1));
        printf("Enter name\n");
        scanf("%s",name[i]);
    }
    for(i=0;i<N-1;i++)
    for(j=i+1;j<N;j++)
    {
        ret=strcmp(name[j],name[i]);
        if (ret<0)
        {
            strcpy(t,name[i]);
            strcpy(name[i],name[j]);
            strcpy(name[j],t);
        }
    }
    printf("\nSorted Names Are :\n");
    for(i=0;i<N;i++)
        printf("%s\n",name[i]);
}
```

The output of the program is:

```
Enter name size
5
Enter name
Raman
Enter name size
5
Enter name
Gopal
Enter name size
6
Enter name
Sharma
Enter name size
7
```

```
Enter name
Ramanan
Enter name size
4
Enter name
Basu
Sorted Names Are:
Basu
Gopal
Raman
Ramanan
Sharma
```

Here the user is asked to first enter the number of characters in a name to be entered. After that the name is entered. This helps in allocating the right size to each name, no space more or less.

You call `strcmp()` and pass references to names being compared. The function `strcmp()` returns 0 if strings are equal; <0 if string 1 is less than string 2 in the ASCII value; > 0 if string 1 is greater than string 2. Thus, names will be exchanged if `*name[j]` is less than `*name[i]`. The string comparison starts from the first character of each word. If they are equal, it will go to the next character and so on till they are unequal or NULL is reached in either of them. Then the difference in the ASCII values of the characters compared last is returned. This is what you wanted in arranging the names alphabetically. Key in the program and you will find that it works correctly.

2.5.8 Pointer to Pointers

You know that pointer is an integer variable that contains the address of any variable of any valid datatype. The variable can itself be another pointer. In this case, you have a pointer which contains the address of another pointer. For instance, let us declare the following:

```
double dv=3.6;
int *p;
int **pp;
```

Now, we assign the following:

```
P=&dv;
pp=&p;
```

Assume that `p =1000`. Then, address of `dv` denoted by `&dv` will also be 1000. Then, the value of `*pp` will also be 1000.

Assume that the address of `p`, i.e. `&p` is 3000. Then, the address of `p` denoted by `*pp` will also be 3000.

The value of `dv`, denoted by `* (&dv)` equals 3.6 in the above.

The same can also be obtained by stating `*p` which is also 3.6.

The same can be obtained by double dereferencing of pp as `**pp` i& also equal to 3.6. Thus, `pp` is a pointer to pointer `p`.

2.5.9 Pointers and Functions

The function call using pointers is known as call by reference. Here, the address of the variable is passed. Note that calling by reference has to be indicated in the function declaration, such as these:

```
fun (int *p, char *cp, float *fp, int *array);
```

This is an indication that the function is to be called by reference for those parameters, which are pointers. A mixed declaration could also be used as given below:

```
fun1(int a, char *cp);
```

Here, you are indicating that an integer is passed by value and a character variable is passed by reference. In the above example, while you can either pass a character array (string) or a character through the second declaration, you can only pass one integer variable through the first parameter.

The function declarator above the function body has to match the declaration. Hence, when `fun1` is called, an integer followed by an address of character will be passed. In the function `fun1`, you may have a declarator as follows:

```
fun1(int d, char * ch)
```

Here, the value of the integer variable will automatically get assigned to d and the `ch` will be assigned the address of the character variable. This means both `ch` and the address of the character variable in the calling function will point to the same location. Thus, both in the calling function and in the called function, the variable is accessible, although under different names. Any modification made to the character variable either in the calling function or the called function affects both.

While calling by reference, you have to pass the address of the variable, if the variable is declared by value, such as `int a`, then you have to pass `&a`. If it was declared as a pointer to say, an integer, such as `int * ip`, then `ip` has to be passed.

So far, we have seen functions returning only a value, irrespective of whether they were called by value or by reference. Functions can also return references or pointers.

Whenever a function is to return a pointer, this has to be indicated by the following:

Declare the function as returning pointers. For instance,

```
int * fun1();
char * fun2();
float * fun3 (int a, float * b, char * c);
```

The difference is the insertion of pointer(`*`) between the return datatype and the function name.

This will be in the same format as the prototype or function declaration. For instance,

```
float * fun3 (int a, float * b, char * c)
```

to match the third function declaration in the above example.

You may call,

```
fun3 (x, &y, z);
```

Here, x is a variable and &y is an address. Although z looks like a value, it is in fact, a reference to character since prototype has the declaration char * c.

The program will obviously return an address or pointer.

2.6 FUNCTION WITH A VARIABLE NUMBER OF ARGUMENTS

To understand the concept of pointer and variable length argument list, let us implement a minprintf() function which is a version of printf() library function. In the following example, ... is used in void minprintf() function to indicate the number and types of arguments. These arguments may vary. The declaration ... only appears at the end of an argument list. In the following example, va_list declares a variable that will refer to each argument in turn. Function va_arg returns one argument to the va_start(ap). Macro va_start initializes ap which points to the first unnamed argument. Function va_arg uses a type name to determine the value and its data type. Function va_end is used when cleanup operation is necessary. It must be called before the program returns.

Example 2.36

```
#include <stdarg.h>
void minprintf(char *format, ...)
{
 va_list ap; // points to each unnamed argument in turn
 char *p, *sval;
 int ival;
 va_start(ap, format);
 // Make ap point to first unnamed argument
for(p = format; *p; ++p)
{
  if (*p != '%')
   {
    putchar(*p);
    continue;
   }
switch (*++p)
{
  case 'd':
```

```
 ival = va_arg(ap, int);
 printf("%d", ival);
   break;
 case 's':
for (sval = va_arg(ap, char *); *sval; ++sval)
 {
putchar(*sval);
 }
 break;
 default:
 putchar(*p);
 }
 }
va_end(ap); // Cleanup process
 }
```

Header file `<stdarg.h>` handles following variable argument list.

```
#include <stdarg.h>
void va_start(va_list ap, argN);
void va_copy(va_list dest, va_list src);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

The `<stdarg.h>` header file contains a set of macros which allows portable functions that accept variable argument lists to be written. The argument `va_list` is defined for variables used to traverse the list. The `va_start()` macro is invoked to initialize `ap` to the beginning of the list before any calls to `va_arg()`. The `va_copy()` macro initializes `dest` (destination) as a copy of `src` (source) when the `va_start()` macro had been applied to `dest` followed by the `va_arg()` macro as had previously been used to reach the present state of `src`.

Check Your Progress

18. Define pointer.

19. What are memory locations? How are values stored in each location?

20. How is a pointer to void declared?

21. How does void pointer function?

22. Define NULL pointer.

23. Differentiate between void pointer and NULL pointer.

24. What will an increment operator do?

25. Define decrementing.

26. What is call by reference in a defined function?

27. What condition is to be kept in mind while calling by reference?

2.7 ANSWERS TO 'CHECK YOUR PROGRESS'

1. An array is a fixed-size sequence of elements of similar data types.

2. To perform any operation on an array, the elements of the array need to be accessed. The process of accessing each element of an array is known as traversal. Generally, the traversal of an array is performed from the element at position 0 to element at position `size`-1.

3. Multi-dimensional arrays can be described as 'arrays of arrays'. A multi-dimensional array of dimension `n` is a collection of elements which are accessed with the help of `n` subscript values.

4. An array in which two subscript values are used to access an array element is known as two-dimensional array, whereas, an array in which three subscript values are used is known as three-dimensional array.

5. Strings, sequences of characters, are represented as arrays of char elements.

6. The initialization of char array can be performed using two methods:
   (a) Character representation
   (b) String Literals

7. A function definition can be written anywhere in the file with a proper declarator, followed by the declaration of local variables and statements.

8. Function definition consists of function declarator and function declarations. Function declaration is terminated by a semicolon but function declarators are not terminated with colon.

9. Pointers contain the addresses of variables. A compiler allocates an address at runtime for each variable and retains this till program execution is completed. Thus, entire memory is not used at a time and only that part of memory is used that is required for execution.

10. The keyword `void` means 'nothing'. A pointer named `void_pointer` also points to a memory address and is an integer like any other pointer. Though it points to nothing but it is useful. In case you assign the address of an integer variable to a float pointer or any other type of pointer other than a pointer to an integer, it is an error. But you can assign address, of any data type, to the pointer of `void` and it is valid. There are situations when a programmer does not know the data type of return value and we can use a pointer of `void` type.

11. Whenever, a function is to return a pointer, this has to be indicated in the function declaration as follows:
    (a) `int * fun1() ;`
    (b) `char * fun2() ;`
    (c) `float * fun3 (int a, float * b, char * c) ;`

    The difference is the insertion of pointer (`*`) between the return data type and the function name.

12. The structure can be assumed as basic datatype and can be declared as an array and as a pointer. Pointers to one or more structures can be declared. For example, in the case of a structure having name `account`, we can declare a pointer to this structure as follows:

    struct account a1 = { 1, "Vasu", 1000 };

    struct account * sp;

    struct sp = &a1 ;

    Here, `sp` is a pointer to structure.

13. Some functions, such as `getchar` and `putchar`, `stdin` and `stdout` are useful as arguments to functions.

14. Label is terminated by colon (:) symbol.

15. When a function calls itself, directly or indirectly, it is known as recursion.

16. A structure, similar to a record, contains a number of fields or variables.

17. Union is a variable, which holds at a common assigned area different data types of varying sizes at different points in time.

18. The pointer is a powerful concept of C. It is powerful because of the following two reasons:

    • It helps in achieving results, which could not otherwise be achieved, such as an indirect method for returning more than one value from a function.

    • It results in a compact code.

    Pointers are closely associated with memory addressing.

19. Memory locations are available in groups of 8 bits or a byte. Each byte in memory has an address. Therefore, each location has an address and stores a value. The value stored can correspond to any data type, such as `float` or `char` or `int`, and their type modifiers.

20. A `pointer` to `void` can be declared in the same way we declare any other pointer, such as pointer to integer, float, etc.

    `void*  void_pointer;`

21. `void` is a keyword and it means 'nothing'. The `void pointer` points to a memory address and hence it is also an integer like any other pointer. The `void pointer` points to nothing.

22. NULL pointer is a type of pointer of any data type and generally takes a value as zero. This denotes that a NULL pointer does not point to any valid memory address.

23. A `void pointer` is a special type of pointer of `void` and denotes that it can point to any data type. NULL pointers can take any pointer type, but do not point to any valid reference or memory address. It is important to note that a NULL pointer is different from a pointer that is not initialized.

24. The postfix of the increment operator enables increment of the variable after printing.

25. Decrementing takes place before printing because it is a prefix operator.

26. The function call using pointers is known as call by reference.

27. While calling by reference, you have to pass the address of the variable, if the variable is declared by value, such as `int a`, then you have to pass `&a`.

2.8 SUMMARY

- Arrays are defined as a fixed-size sequence of elements of the same data type.

- Arrays are stored at contiguous memory locations and can be accessed sequentially or randomly.

- A single-dimensional array is defined as an array in which only one subscript value is used to access its elements.

- An array can be initialized in two ways: by declaring and initializing it simultaneously or by accepting elements for the already declared array from the user.

- To perform an operation on an array, the elements of the array need to be accessed.

- The process of accessing each element of an array is known as traversal.

- Multi-dimensional arrays can be described as 'arrays of arrays.' A multi-dimensional array of dimension `n` is a collection of elements which are accessed with the help of `n` subscript values.

- A two-dimensional array is defined as an array in which two subscript values are used to access an array element.

- Two-dimensional arrays are useful when the elements being processed are to be arranged in the form of rows and columns (matrix form).

- A three-dimensional array is defined as an array in which three subscript values are used to access an individual array element.

- Each programming language keeps a character set, which are used to communicate with the computer.

- Strings, sequences of characters, are represented as arrays of char elements.

- Character representation uses single quotes to represent each character while string literal representation uses double quotes for the entire string literal. Another main difference of string literals from character representation is the end of character denoted by '\0'.

- A function consists of two parts, declarator and declaration.

- A function declaration has the format in which the type of data returned, name of the function and arguments on which the function is to operate, are mentioned.

- Arguments declared as part of the function prototype are called formal parameters, which are enclosed in a pair of parentheses.

- A function prototype is called a function declaration. A function may be declared at the beginning of the main function.

- We may call a function either directly or indirectly. When we call the function, we pass the actual arguments or values. Calling a function is also known as function reference.

- The function declarator is a replica of the function declaration. The only difference is that while the declaration in the calling function will end with a semicolon, the declarator in the called function will not end with a semicolon.

- A function calling itself is called **recursion** and the function may call itself either directly or indirectly.

- A structure, similar to a record, contains a number of fields or variables. The variables can be of any of the valid data types. In order to use structures, you have to first define a unique structure.

- The structure variable declaration is of no use unless the variables are assigned values. Here, each member has to be specifically accessed for each structure variable.

- A structure is also a data type. It is not a basic type defined in C language like `int, float`, etc. But, it is defined by the programmer. Hence, it is called a user defined data type.

- Union is a variable, which holds at a common assigned area different data types of varying sizes at different points in time. Assume that a program, at different points in time of execution uses a `double`, a `float`, an `integer` and a `string`.

- Pointers are closely associated with memory addressing. They are variables, which contain the address of another variable.

- Memory locations are available in groups of 8 bits or a byte. Each byte in memory has an address. The memory locations are arranged in an increasing order of addresses starting from 0000, which increases one by one.

- A pointer is declared like a variable with appropriate data type. The pointer variable in the declaration is preceded by the * (asterisk) symbol.

- The data type of every pointer is always an integer because it is the value of the memory address. Memory addresses are integers.

- An integer operand can be used with a pointer to move it to a point/refer to some other address in the memory. ++ and – –operators are used to increment and decrement a pointer and are commonly used to move the pointers the next location.

- When a function is called, the actual arguments are passed according to the list provided in the declaration. These are all values, which are passed to a function and this method is called call by value. In call by value, only one value is returned from a function.

- A two-dimensional array is conventionally represented as `[i][j]` and can be considered as an array of one-dimensional arrays.

- `**` indicates `pointer` to `pointer`.

- A pointer to a one-dimensional array can be denoted as `*b` and two-dimensional array can be denoted as `**b`.

- A string is an array of characters. A library function `strlen()` is used to find the length of the string.

- A set of library functions are used for testing characters. The prototype for these functions is defined in `<ctype.h>`. The argument for all functions must be an unsigned character.

- `vows` array contains the five vowels in the upper case and in the lower case.

- Alphabets other than vowels are called consonants and are checked using the function `iscons()`.

- `malloc` and `calloc` are used to specify the actual dimension at run-time and hence enable memory allocation dynamically. The functions `malloc()` and `calloc()` allot memory dynamically.

- The memory allocated using `malloc()`, `calloc()` can also be freed when it is no longer necessary by using the function `free ( )`.

- The function call using pointers is known as call by reference. Here, the address of the variable is passed.

- While calling by reference, you have to pass the address of the variable, if the variable is declared by value, such as `int a`, then you have to pass `&a`. If it was declared as a pointer to say, an integer, such as `int * ip`, then `ip` has to be passed.

2.9   KEY TERMS

- **Arrays:** It is the fixed-size sequence of elements of the same data type.

- **Traversal:** It is the process of accessing each element of an array.

- **Multi-Dimensional Array:** It is a collection of elements which are accessed with the help of n subscript values.

- **Two-Dimensional Array:** It is defined as an array in which two subscript values are used to access an array element.

- **External Variables:** They are also known as global variables and the scope of external variables extends to all the functions of a program.

- **Pointer:** It is a variable that contains the address of another variable and is closely associated with memory addressing.

- **Memory Locations:** These are available in groups of 8 bits or a byte which has an address and stores a value that can be any data type, such as `float` or `char` or `int`.

- **Call by Reference:** The function call using pointers is known as call by reference.

2.10 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Why is an array needed?
2. Write a program that accepts a single-dimensional integer array of size N, and reverses the contents of the array without using any second array.
3. Write a program that accepts an integer array of size N, and exchanges the elements of the first half with the elements of the second half of the array. For example, if the array is {8, 10, 1, 3, 17, 90, 13, 60}, then the output should be {17, 90, 13, 60, 8, 10, 1, 3}.
4. Write a program to find the sum of the elements below the main diagonal and sum of the elements above the main diagonal.
5. Write a program to check whether a given matrix is singular matrix or not.
6. What is a function declarator?
7. How do you call a function?
8. Write a short note on function arguments.
9. What is the difference between structure declaration and union declaration?
10. Write the syntax to define structure within structure.
11. What is a pointer?
12. What is dynamic allocation of memory?

Long-Answer Questions

1. Write a program to compute the inverse of a square matrix.
2. Write a program to check the orthogonality of a given matrix.
3. Write a program to find the rank of a matrix.
4. An array `Arr[35][15]` is stored in the memory along the row with each of its element occupying 4 bytes. Find out the base address and the address of an element `Arr[20][5]`, if the element `Arr[2][2]` is stored at the address 3000.
5. An array `S[40][30]` is stored in the memory along the row with each of the element occupying 2 bytes. Find out the memory location for the element `S[15][5]`, if an element `S[20][10]` is stored at the memory location 5500.
6. An array `ARR[5][5]` is stored in the memory with each element occupying 4 bytes of space. Assuming the base address of `ARR` to be 1000, compute the address of `ARR[2][4]`, when the array is stored: (i) row wise (ii) column wise.
7. An array `VAL[1..15][1..10]` is stored in the memory with each element requiring 4 bytes of storage. If the base address of array `VAL` is

1500, determine the location of VAL [12] [19] when the array VAL is stored: (i) row wise   (ii) column wise

8. Find the output of the following program and then confirm your findings by executing the same.

```
#include<stdio.h>
    int main()
    {
    float z=0.0;
    int y;
`   float div(int x);
        for (y=0; y<=10; y++)
    {
        z=div(y+2);
    printf("result=%f\n", z);
    }
}
    float div(int n)
{
    float b, c=2.0;
    b=n/c;
    return b;
}
```

9. Explain the following:
   (i) Function
   (ii) Return statement
   (iii) A function calling multiple functions

10. Write a program to that passes argc, and argv arguments in the main() function.

11. Explain the recursive factorial algorithm for finding the factorial of 7.

12. Write a program using the struct keyword to find out the name and account balance amount as per values that have been input.

13. Describe the following with the help of examples:
   (i) Pointer arithmetic.
   (ii) Pointers and two-dimensional arrays.
   (iii) Advantage and functions of:
      (a) malloc
      (b) calloc
   (iv) String functions.
   (v) Call by reference and similarity with global variables.

14. Write programs for the following:
   (i) A function nstrcmp to compare string cs to string ct; return <0 if cs <ct, 0 if cs = = c or >0 if cs > ct.

(ii) A function to check whether characters +, −, ∗, / present in a string.

(iii) To find the kth smallest element in an array using pointers.

(iv) To check whether a string is palindrome.

2.11 FURTHER READING

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures.* New Delhi: Galgotia Publications, 2003.

Tanenbaum, A.M., Yedidyah Langsam and Moshe J. Augenstein. *Data Structures using C and C++*. New Delhi: Prentice-Hall of India, 1995.

UNIT 3 DATA STRUCTURES

Structure

3.0 INTRODUCTION

Data plays an important role in programming and all computer programs involve applying operations on the data. Data can be described as a value or a set of values such as name and age of a person, grade of a student, salary of an employee, and so on. It is just a collection of values and no conclusion can be drawn from it, however, after processing it becomes information that can be helpful in making some decisions. For processing data, it should be available in the main memory, since a processor only operates on data that is in main memory. In order to represent the data in main memory, some model is needed so that it can be processed efficiently. This model is termed as *data structure*. There are various models available for this, and in this unit, we will introduce you to various data structures

and different operations that can be performed on them. An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and any automated reasoning tasks. It is an efficient method that can be expressed within finite amount of time and space and is the best way to represent the solution of a particular problem in a very simple and efficient manner. If we have an algorithm for a specific problem, then we can implement it in any programming language. This means that the algorithm is independent from any programming languages. The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. In this unit, you will learn about how algorithms work. A stack is a linear data structure in which an element can be added or removed only at one end called the top of the stack. In stack terminology, the insert and delete operations are known as push and pop operations, respectively. The last element added to a stack is the first element to be removed, that is, the elements are removed in the opposite order in which they are added to the stack. Hence, a stack works on the principle of last in first out and is known as a last-in-first-out (LIFO) list. A pile of books is one of the common examples of a stack. A new book to be added to the pile is placed at the top and a book to be removed is also taken off from the top. You will also learn about queues and the operations on queues. A queue is a linear data structure in which a new element is inserted at one end and an element is deleted from the other end. The end of the queue from which the element is deleted is known as the front and the end at which a new element is added is known as the rear. A common example of a queue is people waiting in line at a bus stop. The first person in the queue is the first person to take the bus. Whenever a new person comes, he joins at the end of the queue. That is, the order in which the people take the bus is the order in which they have joined the queue. The size of the queue keeps varying according to the number of people joining and leaving the queue. In simple terms, a list refers to a collection of data items of similar type that are arranged in a sequence. A list of students' names or addresses is an example of lists. One way to store such lists in the computer's memory is to use an array data structure. However, arrays have certain problems associated with them. As array elements are stored in adjacent memory locations, a sufficient block of memory is allocated to an array at compile time. Once the memory space is allocated to an array, it cannot be expanded or contracted. That is why an array is called a static data structure. If the number of elements to be stored in an array increases or decreases significantly at runtime, it may require more memory space or result in wastage of memory, both of which are unacceptable. Another problem is that the insertion and deletion of an element into an array are expensive operations. This is because they may require a number of elements to be shifted. As a result of these problems, arrays are not generally used to implement linear lists. Therefore, a new data structure is required which eliminates the problems of static data structures. This unit introduces you to dynamic data structure used to implement linear lists. In addition, it also discusses the various operations that can be performed such as creation, traversal, searching, insertion and deletion on linked lists.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the meaning and terminology of data structure
- Discuss how algorithms work
- Analyze the algorithm efficiency
- Describe the meaning and applications of stacks along with their implementation with pointers
- State the meaning of queue along with its implementation and types
- Explain circular queue, priority queue, deque and their implementation
- Discuss static and dynamic memory allocation
- Describe the concept of linked lists and static and dynamic variables

3.2 DATA STRUCTURE

The logical or mathematical model used to organize data in the main memory is called **data structure**. Various data structures are available, each with its special features. These features should be kept in mind while choosing a data structure for a particular situation. Generally, the choice of a data structure depends on its simplicity and effectiveness in processing data. In addition, we also consider how well it represents the actual relationship of data in the real world. Data structures are divided into two categories, namely *linear data structure* and *non-linear data structure*.

Linear Data Structures

A linear data structure is one in which the elements form a sequence. This means each element in the structure has a unique predecessor and a unique successor. An array is the simplest linear data structure. Various other linear data structures are linked lists, stacks and queues.

Arrays

A finite collection of homogenous elements is termed an **array**. Here, the word 'homogenous' indicates that the data type of all the elements in the collection should be the same, that is, int, char, float or any other built-in or user-defined data type. However, an array cannot have elements of two or more data types together.

Elements of an array are always stored in contiguous memory locations irrespective of the array size. The elements of an array can be referred to by using one or more indices or subscripts. An index or subscript is a positive integer value, which indicates the position of a particular element in the array. If the number of subscripts required to access any particular element of an array is one, then it is a single-dimensional array. Otherwise, it is a multidimensional array. A multidimensional array may be two dimensional or even more.

Consider a single-dimensional array `Arr` with size n, where n is the maximum number of elements that `Arr` can store. Mathematically, the elements of `Arr` are denoted as Arr_1, Arr_2, Arr_3,..., Arr_n. In different programming languages, array elements are denoted by different notations, such as parenthesis notation or bracket notation. Table 3.1 shows the notation of elements of a single-dimensional array `Arr` with size n in different programming languages.

Table 3.1 Different Notations of a Single-dimensional Array

S. Number	Notation	Programming Language(s)
1	`Arr(1)`, `Arr(2)`, `Arr(3)`,…, `Arr(n)`	BASIC and FORTRAN
2	`Arr(1)`, `Arr(2)`, `Arr(3)`,…, `Arr(n)`	PASCAL
3	`Arr(0)`, `Arr(1)`, `Arr(2)`,…,`Arr(n-1)`	C, C++ and Java

Note that in languages like BASIC, PASCAL and FORTRAN, the smallest subscript value is 1 and the largest subscript value is n. On the other hand, in languages like C, C++ and Java, the smallest subscript value is 0 and the largest subscript value is n−1. In general, the smallest subscript value used to access an array element is the lower bound (L_b) and the largest subscript value is upper bound (U_b).

In two-dimensional arrays, the elements can be viewed as arranged in the form of rows and columns (matrix form). To access an element of a two-dimensional array, two subscripts are used—the first one represents the row number and the second one represents the column number. Consider a two-dimensional array `Arr` with size m*n, where m and n represent the number of rows and columns, respectively. Mathematically, the array `Arr` is denoted as Arr_{ij}, where i and j indicate the row number and the column number with i<=m and j<=n. Table 3.2 shows the notation of elements of a two-dimensional array `Arr` in different programming languages.

Table 3.2 Different Notations of a Two-dimensional Array

| S. Number	Notation	Programming Language(s)
1	`Arr(i,j)` with `0<i<=m and 0<j<=n`	BASIC and FORTRAN
2	`Arr[i,j]` with `0<i<=m and 0<j<=n`	PASCAL
3	`Arr[i][j] with 0<=i<m and 0<=j<n`	C, C++ and Java

Stacks and Queues

A **stack** is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end. This end is called **Top** of the

stack. The operations of adding a new element in the stack and deleting an element from the stack are called **push** and **pop,** respectively. Since the addition and deletion of elements always occurs at one end of the stack, the last element that is pushed onto the stack is the first one to come out. Therefore, a stack is also called a **Last-In-First-Out** (**LIFO**) list. Figure 3.1 shows a stack with five elements and the position of top.
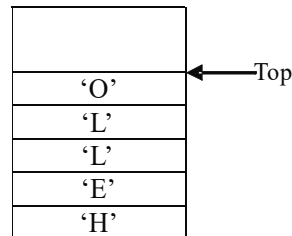
Fig. 3.1 *A Stack with Five Elements*

A **queue** is a linear data structure in which the addition or insertion of a new element occurs at one end called **Rear**, and the deletion of an element occurs at the other end called **Front**. Since insertion and deletion occur at opposite ends of a queue, the first element that is inserted in the queue is the first one to come out. Therefore, a queue is also called a **First-In-First-Out** (**FIFO**) list. Figure 3.2 shows a queue with five elements and the position of Front and Rear.
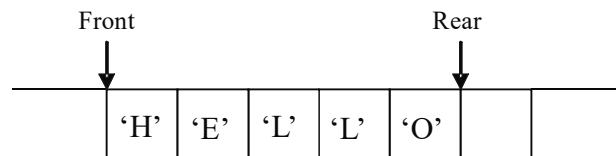
Fig. 3.2 *A Queue with Five Elements*

### Non-Linear Data Structures

A non-linear data structure is one in which the elements do not form a sequence. It means, unlike a linear data structure, each element is not constrained to have a unique predecessor and a unique successor. Trees and Graphs are classified under non-linear data structures.

Trees

Many times we observe a hierarchical relationship between various data elements. This hierarchical relationship between data elements can be easily represented using a non-linear data structure called a **tree**. A tree consists of multiple nodes with each node containing zero, one or more pointers to other nodes called **child nodes**. Each node of a tree has exactly one parent except a special node at the top of the tree called the **root node**. A sample tree with A as the root of the tree is shown in Figure 3.3.
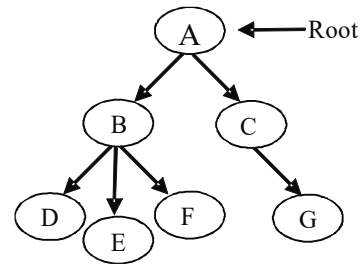
***Fig. 3.3*** *A Sample Tree*

In this tree, the root node has two child nodes B and C. In turn, the node B has three child nodes D, E and F, and the node C has one child node G. The nodes D, E, F and G have no child nodes. The nodes without any child node are called **external nodes** or **leaf nodes**, whereas the nodes having one or more child nodes are called **internal nodes**.

Graphs

Formally, a graph G(V, E) consists of a pair of two non-empty sets V and E, where V is a set of vertices or nodes and E is a set of edges. A graph is used to represent the non-hierarchical relationship among the pairs of data elements. The data elements become the vertices of the graph and the relationship is shown by edges between the two vertices. For example, assume four places W, X, Y and Z, such that:

$\Rightarrow$ There exists some path from X to Y, X to W, Y to W, Y to Z, and Z to W.

$\Rightarrow$ There is no direct path from X to Z.

We can simply represent this situation using a graph where the places W, X, Y and Z are represented as the nodes of the graph and a path from one place to another place is represented by an edge between them (Figure 3.4).
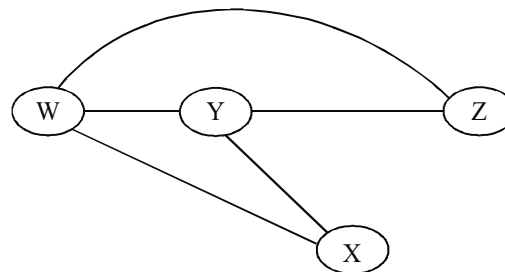
Fig. 3.4 *A Sample Graph*

It is clear from Figure 3.4 that each node can have links with multiple other nodes. This analogy suggests that it is similar to a tree; however, unlike trees, there is no root node in a graph. Further, graphs show relationships, which may be non-hierarchical in nature. It means that there is no parent and child relationship. A tree can be considered a variant or a special type of graph.

3.2.1 Primitive and Composite Data Types

Each programming language provides various data types and each data type is represented differently within the computer's memory. The memory requirement

of a data type determines the permissible range of values for that data type. The data types can be classified into several categories, including primitive data types and composite data types.

The data types provided by a programming language are known as *primitive data types* or *in-built data types*. Different programming languages provide different set of primitive data types. For example, the primitive data types in the C language are int (for integers), char (for characters), and float (for floating point numbers). The data types that are derived from primitive data types of the programming language are known as **composite data types** or **derived data types**. Various data types in the C language include arrays, functions, and pointers.

In addition to primitive and composite data types, programming languages allow the user to define new data types (or user-defined data types) as per his requirements. For example, the various user-defined data types as provided by C are structures, unions, and enumerations.

3.2.2 Abstract Data Type

Generally, handling small problems is much easier than handling comparatively larger problems. The same rule is applicable to programming also. Therefore, a large program is decomposed into small logical units or modules, each of which does a well-defined sub task of the whole program. The size of each module is kept as small as possible and if required, other modules are invoked from it. This modular design provides several advantages. First, several people can be employed to work on a single program, which will increase the speed of completing the given task. Second, a well-designed modular program has modules independent of each others, implementation, which will make the program easily modifiable.

An abstract data type (ADT) is an extension of a modular design in a way that the set of operations of an ADT are defined at a formal, logical level, and nowhere in ADT's definition, it is mentioned how these operations are implemented. The data type integer is an example the abstract data type. We frequently perform operations on integers that are associated with them like addition, subtraction, division, multiplication, modulus, etc. However, we do not know how these operations are actually performed on integers. We only know the syntax of how to perform these operations in some programming language. For example, C language defines +, –, /, *, % to perform some basic arithmetic operations on integers.

The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT accomplishes this by invoking the required operation. If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programs using it. The data structures, namely, *linked lists*, *stacks*, and *queues* are some examples of ADTs.

3.2.3 Algorithm Design

There are various techniques which can be used for designing algorithms. Some commonly used algorithm design techniques have been discussed in this section.

Brute Force Algorithm

Brute force is a general technique which is used for finding solutions to various problems. In this technique, all possible candidates for the solution are listed and then examined to check whether each candidate satisfies the problem. For example, to find the factors of a natural number n, it will determine the number of integers from 1 to n and then check all possible combinations of integers from 1 to n, that will form a product (equivalent to number n) when multiplied together.

There are various algorithms where brute force technique can be applied; some of which are selection sort, pattern matching in strings, knapsack problem, and travelling salesman problem. Let us discuss how this technique is used in pattern matching. Pattern matching is the process of determining whether a given pattern of string (say, **P**) occurs in another string (say, **S**) or not, provided the length of string **P** is not greater than that of **S**. In other words, pattern matching determines whether or not **P** is a sub-string of **S**. Using this algorithm, the string **S** is scanned character by character. Starting from the first character, each character of **S** is compared with the first character of **P**. When the match for the first character is found, the next character from the pattern string **P** is compared with the character adjacent to the searched character in string **S**. This process continues till the complete pattern string is found. If the next character does not match, string **S** is searched again for other occurrences of the first character and also the subsequent characters of the pattern string **P** in the similar manner. This process continues until a match is found or the end of pattern string **P** is reached. If a match is found, this algorithm returns the position in **S** where the pattern string **P** occurs. For example, consider a pattern string **P=“ways”** that is to be searched in string **S = “hard work always pays”**, then this algorithm will return 13 as result.

The main advantage of brute force is that it is simple to implement. The algorithm will definitely find a solution if it exists, since it examines all possible solutions to a problem. However, the execution time of this algorithm is directly proportional to the number of solutions, that is, it increases rapidly with an increase in the size of the problem. Therefore, it is used in situations where the size of the problem is small or when some problem-specific heuristics are available that can be used to limit the number of possible solutions to a controllable size. It is also used as a baseline (an imaginary standard by which things are measured or compared) method to develop heuristics for other search algorithms.

Divide and Conquer

The divide and conquer technique is one of the widely used technique is to develop algorithms will for problems which can be divided into sub-problems (smaller in size but similar to the actual problem) so that they can be solved efficiently. The technique follows a top-down approach. To solve the problem, it recursively divides the problem into a number of sub-problems, to the extent where they cannot be sub-divided any further into more sub-problems. It then solves the sub-problems to find solutions that are then combined together to form a solution to the actual problem.

Some of the algorithms based on this technique are sorting, multiplying large numbers, syntactic analysis, etc. For example, consider the merge sort algorithm

that uses the divide and conquer technique. The algorithm is composed of steps, which are as follows:

Step 1: Divide the n-element list, into two sub-lists of n/2 elements each, such that both the sub-lists hold half of the element in the list.

Step 2: Recursively sort the sub-lists using merge sort.

Step 3: Merge the sorted sub-lists to generate the sorted list.

Note that the merging of sub-lists starts, only when the length of sorted sub-lists (through recursive application) reach to 1. At this point, two sub-lists each of length 1 are merged (combined) by placing all the elements of the list in a sorted order.

Dynamic Programming

Dynamic programming is a technique that is generally used for solving optimization problems where the best (optimal) solution out of the available possible solutions is to be found. One example of such a problem is the shortest path problem where, if a person in city X has to reach city Z there would be many possible routes to reach the city Z. The aim is to select the shortest route from all the available routes so as to reach the destination in minimum possible time. Note that in the given problem, all possible routes represent different solutions to the problem.

Using dynamic programming, when the problem is solved, there is a possibility that sub-problems of the same type may arise. The basic idea behind the technique is that it stores the solutions to such sub-problems. This helps in repeated calculation and hence, improves the efficiency of the algorithm. The algorithms that are designed using this technique consist of three steps, which are as follows:

- **Dividing the problem into simpler sub-problems**: The problem is divided into sub-problems, such that each sub-problem has a similar structure to the original problem.

- **Finding optimal solutions to sub-problems**: The solution to an original problem is computed by combining the solutions of the sub-problems. Therefore, for finding optimal solution to the original problem, the solutions to sub-problems should also be optimal.

- **Storing solutions to overlapping sub-problems**: The identified sub-problems consist of either unrelated sub-problems (each having an independent solution) or common sub-problems (having similar optimal solution). The solutions to these sub-problems are stored in a table. So, while finding optimal solutions, if any overlapping (recurring) sub-problems are found, the solutions stored in the table can be used. This increases the efficiency of the dynamic programming algorithm.

Note that dynamic programming applies a bottom-up approach to solve the problem. That is, it first finds a solution to the simplest sub-instances of the problem and then solves the more complex instances, using the results of earlier computed (sub) instances. Some of the well-known optimization problems where the dynamic programming technique is used are knapsack problem, problem of making change, shortest path problem, and chained matrix multiplication problem.

Greedy Algorithm

We have discussed the use of dynamic programming to solve optimization problems. However, there are many optimization problems such as the famous minimal spanning tree problem by Kruskal, minimum number of notes problem, and activity-selection problem that can be solved more efficiently using a greedy algorithm. As we know that the algorithm for optimization problems consist of stages, having a set of choices at each stage. The basic idea of the greedy technique is to make locally optimal choices (i.e., the best choice at a particular stage) assuming that these choices will result in a globally optimal solution.

The technique follows a top-down approach. To find solution to a problem, sequence of choices is made recursively (based on minimum or maximum value criterion) from the set of given choices at each stage. After a choice is made, the problem reduces to a sub-problem (similar to the actual problem). The solution to the actual problem is found by combining the sequence of choices that are made.

The greedy technique does not always lead to an optimal solution. However, the problems that are solvable using this technique are said to possess the greedy-choice property.

3.2.4 Program Analysis

A computer accepts instructions from a user in order to execute the same on a machine. In order to instruct a computer to perform a task, a program is required which enables a computer to perform a specified task. These computer programs include set of steps which explain a computer what needs to be done, how to implement instructions and in case of any errors how to inform the end user about the same. A computer executes the set of statements also known as a program in order to achieve the desired objective. A program can be written in different languages like COBOL, FORTRAN, PASCAL, C, C++, JAVA, Python etc. The statements written in a program differ from one language to another. However, one of the easiest technique to write a program is to use algorithms. The computer algorithm is a structure which explains the set of instructions that need to be executed in order to accomplish the objective of a program. These computer algorithms can be translated very easily into computer programs written using a programming language as per the requirements. For example an algorithm can be written to prepare a cup of tea and the same is given below:

Algorithm to prepare a cup of tea
Line 1: start
Line 2: collect ingredients vessel, water, tea leaves, sugar, milk
Line 3: switch on the Heater
Line 4: Put the empty vessel on heater
Line 5: pour water as per the requirement in the vessel
Line 6: wait till the water is boiled
Line 7: Add milk, tea leaves and sugar as per the requirement
Line 8: wait till the mixture is boiled
Line 9: pour the boiling mixture in a cup
Line 10: serve the tea

The algorithm given above to prepare a cup of tea is one of the methods to prepare tea. However, different methods are used to prepare a cup of tea and the same may vary from one place to another. Therefore, the algorithm written is not the only algorithm to prepare tea rather 'n' number of algorithms can be written to prepare tea. Similarly a computer program can have different methods to write a computer program.

Notion of Algorithm

An algorithm is a systematic sequence of instructions which are required to be executed in order to achieve the objective of an algorithm. The basic characteristics of an algorithm are given below:

(i) Finite

(ii) Single Entry and Single Exit point

(iii) Achieve Desired Objective

(iv) Systematic Sequence

Finite: The algorithm should be finite in nature, for example, an algorithm with a condition which is always true will lead to execution of a program or an algorithm infinite number of times.

```
void main()
{
    while(1)
    {
        printf("i am stuck in the loop as the
condition will never terminate");
    }
}
```

The program given above will not terminate as the condition mentioned in loop is always true. The program will result in displaying the text "I am stuck in the loop as the condition will never terminate" continuously on screen. Therefore, the need of the hour in this case is to write a program or an algorithm which will allow a user to terminate a program as and when required.

Single Entry and Single Exit point: Every algorithm should have a single entry point and a single exit point. In case a program is having multiple entry and exit points will lead to improper execution of an algorithm.

Table 3.3 Single entry and single exit point

Algorithm	Program using C language

Line 1: start
Line 2: declare a,b,c
Line 3: read a,b
Line 4: c=a+b
Line 5: display c
Line 6: stop
	``void main()
	{
	    int a,b,c;
	    printf("Enter two numbers:");
	    scanf("%d,%d", a,b);
	    c=a+b;
	    printf("the sum of two numbers is %d",c);
	}```

The algorithm shown above is written to automate the process of addition of two numbers. An equivalent program in C language is written for the same for the benefit of the reader. The algorithm should start from line No. 1 and should end at line number 6 only. In case the algorithm starts from line No. 4 will generate error as the information about the initialization of the variable is not available at line No. 4. Therefore, the algorithm needs to be executed from a single point of entry and exit from a single point in order to maintain the integrity and accuracy of an algorithm.

Achieve Desired Objective: An algorithm should be able to achieve the desired objective for which the same has been designed. For example the algorithm given above is designed to add two numbers and display the result to the user. In case the algorithm is not generating the summation of two numbers then the same cannot be treated as an algorithm which is fulfilling the requirement of an end-user.

Systematic Sequence: An algorithm needs to written in a sequence of instructions which are executed one after another. The sequence will always be implemented from line number first till the line number last.

The algorithm written to add two numbers is not necessarily the only method which can be used to automate the process of adding two numbers. As an example a simple algorithm can be written in "n" number of ways and to explain the same an algorithm to add two numbers is written in three different ways and the same is given below:

*Table 3.4 An algorithm to add two numbers in three different ways*

Algorithm-I	Algorithm-II	Algorithm-III
Line 1: start	Line 1: start	Line 1: start
Line 2: declare a,b,c	Line 2: declare a,b	Line 2: declare a,b
Line 3: read a,b	Line 3: read a,b	Line 3: read a,b
Line 4: c=a+b	Line 4: a=a+b	Line 4: b=a+b
Line 5: display c	Line 5: display a	Line 5: display b
Line 6: stop	Line 6: stop	Line 6: stop

The number of variables used in the above given algorithms varies which results in optimal usage of resources. The same algorithm can be written in different ways in order to achieve the desired objective. This helps a user in differentiating

between a good algorithm and a bad algorithm. The comparison of the algorithms based on the usage of resources like memory-usage, time-taken, etc., motivates the reader to study the domain of algorithm analysis. Keeping in mind the proper utilization of memory-usage and time taken to solve a problem a good algorithm will always efficiently use the resources of a system.

Fundamentals of Algorithmic Problem Solving

The problem solving is the main initiator of writing algorithms or computer programs. Everyone around you has different problems or requirements on daily basis. Every individual would like to develop automated tools for specific problem solving methods or procedures. Therefore, a computer program or an algorithm is written for automating the process of developing a set of instructions to fulfill the requirement of a user by automating the procedure of a problem solving technique. The process of writing an algorithm for a problem solving methods includes different steps and the same are shown in Figure 3.5.
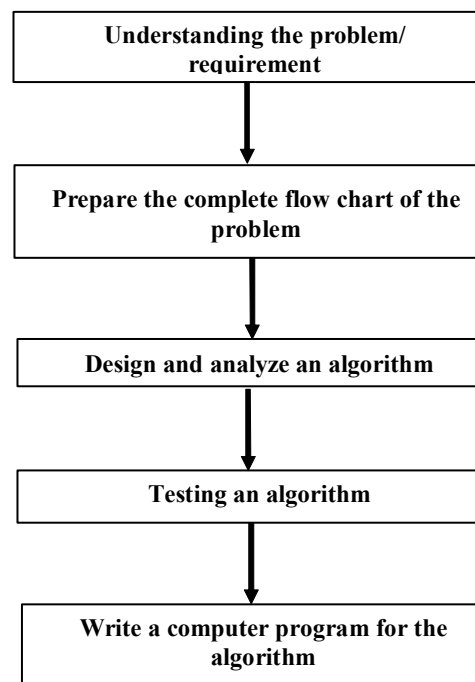
Fig. 3.5 Steps of problem solving

(i) Understanding the problem/requirement

Understand the requirement of a user in order to develop and design an algorithm. The requirement of a user can be an overview but as a designer of an algorithm the developer needs to understand all the processes and sub processes within the system. The problem understanding is one of the most important activity for designing an algorithm as if a developer skips even a small sub process within the problem definition will result is design of wrong algorithm. As discussed in the precious section one of the characteristics of an algorithm is to fulfill the desired objective which cannot be achieved if a developer has not understood the problem accurately and completely.

An algorithm developer needs to discuss all the confusions in order to understand the problem without any ambiguities. At the same time the developed should try to consider multiple alternatives which can enhance the process of achieving the final objective of a problem solution process. The evaluation of multiple alternatives will help a developer to prepare a plan for developing an algorithm which is designed to automate the process of problem solution effectively and efficiently.

(ii) Prepare the complete flow chart of the problem

Flow chart is a graphical representation to display the sequence of instruction within an algorithm or a computer program. The flow-chart tool helps the developer in preparing a graphical representation of the problem solving process which results in efficient design of an algorithm. This graphical representation helps the end-user to understand the accuracy of the problem definition which is very essential before designing an algorithm.

(iii) Design and analyze an algorithm

Based on the understanding of the problem solving process and graphical representation of the same an algorithm is written. The reader here can again get an idea of the importance of step (i) of the problem solving process.

Once the algorithm has been designed, the developer has to ensure that the resources like space and time of a computer are used by an algorithm effectively and efficiently. The process of analyzing an algorithm plays an important role while writing a program and execution of the same on a computer. An algorithm designed can develop alternative algorithms for the same problem solving process in order to analyze the same for selection of an optimal algorithm. The main objective of the analyzing an algorithm is to ensure that the algorithm uses the resources of a computer optimally.

(iv) Test the algorithm

Once the algorithm is designed the developer needs to ensure the correctness, completeness and accuracy of the algorithm. The process of checking the correctness of the algorithm is known as testing where the algorithm is tested for dummy situations and dummy data. This method of executing the algorithm is also known as dry run.

(v) Write a computer program for the algorithm

The algorithm written at step (iv) can be written in any programming language as per the requirement of the problem solving process in order to write a computer program for the problem solving process.

Fundamentals of Analysis of Algorithm Efficiency

The analysis of an algorithm for efficiency is a method to evaluate the efficiency of an algorithm for different parameters. The main objective of evaluating different algorithms for efficiency is to find out an algorithm which uses the system resources optimally and achieving the desired objective of an algorithm. As discussed earlier in this chapter for every problem the possible solutions are more than one which

justifies the importance of evaluation of algorithms for finding out an optimal one based on utilization of resources. The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time. The time efficiency of an algorithm demonstrates how fast an algorithm will be executed and space efficiency of an algorithm demonstrates the optimal units of memory required to execute an algorithm. The two parameters memory and time are two basic parameters used to find the efficiency of an algorithm. However, some other parameters are also used to find out efficiency of an algorithm and the list of the parameters is given below:

- Size of input
- Running time
- Worst, Best and Average scenarios
- Asymptotic Notations

Size of input: The efficiency of algorithm is also evaluated based on the input size of an algorithm. For example the size of input in case of a word count algorithm will be the number of words and for an alphabet count algorithm the size of the input will be the number of alphabets given as input to an algorithm.

Running time: The efficiency of an algorithm based on running depends on the measuring unit used to measure running time. For example if the unit used for measurement in terms of second then the efficiency will be evaluated in seconds. In case the measuring unit is nano-seconds then the evaluation parameter unit will be in nano-seconds. The running time for an algorithm also is directly dependent on the speed of the computer, compiler used and quality of an algorithm. However, for uniformity the basic operations are identified within an algorithm and the time taken to complete the same is evaluated in order to analyze the efficiency of an algorithm.

Worst, Best and Average scenarios: The algorithm efficiency is also evaluated based on the different possibilities of the input size. In case the size of the input is worst fit where the size of the inputs is taken as extreme higher value and the efficiency of an algorithm based on the same is evaluated and the same is used to compare for a good algorithm. In case the size of the input is best fit where the size of the input is ideal and the efficiency of an algorithm is evaluated and the same criterion is used as one of the benchmarks in finding out an optimal algorithm. Similarly in case the size of the input is average then the efficiency of the algorithm is evaluated in order to find out the best selection of an algorithm.

Order of Growth: The efficiency of an algorithm is evaluated based on order of the growth in which an algorithm is performing. The order of growth is about how an algorithm is performing when the system used to execute the same very fast and what is the efficiency of an algorithm when the input size is doubled. The efficiency can be evaluated using a basic equation as given below:

$T(n) = Cop \times Cn$

where $T(n)$ is running time of an algorithm

Cop is the time take for single basic operation

Cn is the number of basic operations within an algorithm.

Asymptotic Notations: The efficiency of an algorithm is measure using asymptotic notations where three notations O, W, Θ are used. The notation O is known as big "Oh", the W notation is known as omega and the notation Θ is known as big theta.

For big O notation a function t(n) is said to be in O(g(n)), for big W notation a function t(n) is said to be in W (g(n)) and for big Θ notation a function t(n) is said to be in Θ (g(n)).

Check Your Progress

1. Define the term data structure.
2. Differentiate between primitive and composite data types.
3. Define ADT.
4. In what languages can a program be written?
5. What is the main initiator of writing algorithms or computer programs?
6. What is the analysis of an algorithm for efficiency?
7. What is the definition of efficiency of an algorithm?

3.3 STACKS AND THEIR REPRESENTATION

In computer science, a **stack** is a last in first out (LIFO) data structure. It is characterized by two fundamental operations: *push* and *pop*. While the 'push' operation adds an element to the top of the list, the 'pop' operation removes an item from there and returns the value to the caller. A push operation either hides items already on the stack or initializes the stack if it is empty. A 'pop' operation reveals previously concealed items, or results in an empty list.

A stack is a *restricted data structure*, because only a small number of operations are performed on it. The nature of the pop and push operations mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition; therefore, the lower elements are typically those that have been in the list the longest. Figure 3.6 shows a representation of a stack.
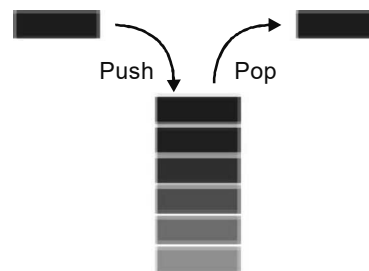
Fig. 3.6 *Simple Representation of a Stack*

Implementation of Stacks with Pointers

A stack pointer, usually in the form of a hardware register, is an address that identifies the location of the most recent item placed on the stack. Typically, a stack has a fixed origin and a variable size. Initially, when the size of a stack is zero the stack pointer points to its origin.

In case of a *push* operation, a data item is placed at the location pointed to by the stack pointer and the address in the stack pointer is adjusted by the size of the data item. On the other hand, in case of a *pop* operation, the data item at the current location pointed to by the stack pointer is removed, and the stack pointer is adjusted by the size of the data item. As data items are added to the stack, the stack pointer is displaced to indicate the current extent of the stack, which expands away from the origin. For example, a stack might start at a memory location of 1000, and expand towards lower addresses. In that case new data items are stored at locations ranging below 1000, and the stack pointer is decremented each time a new item is added. When an item is removed from the stack, the stack pointer is incremented.

Stack pointers may point to the origin of a stack or to a limited range of addresses, either above or below the origin; however, the stack pointer cannot cross the origin of the stack. In other words, if the origin of the stack is at address 1000 and the stack grows downwards (towards addresses 999, 998, and so on), the stack pointer must never be incremented beyond 1000 (to 1001, 1002, etc.).

3.3.1 Applications of Stacks

Stacks are used where the last-in-first-out principle is required like reversing strings, checking whether the arithmetic expression is properly parenthesized, converting infix notation to postfix and prefix notations, evaluating postfix expressions, implementing recursion and function calls, etc. This section discusses some of these applications.
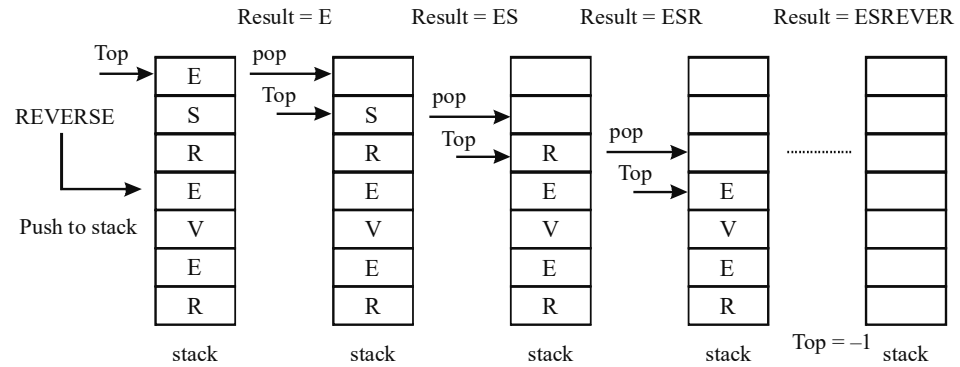
Reversing Strings

A simple application of stacks is reversing strings. To reverse a string, the characters of the string are pushed onto the stack one by one as the string is read from left to right. Once all the characters of the string are pushed onto the stack, they are popped one by one. Since the character last pushed in comes out first, subsequent pop operations result in the reversal of the string.

For example, to reverse the string 'REVERSE', the string is read from left to right and its characters are pushed onto a stack, starting from the letter R, then E, V, E, and so on as shown in Figure 3.7.

Fig. 3.7 Reversing a String Using a Stack

Once all the letters are stored in the stack, they are popped one by one. Since the letter at the top of the stack is E, it is the first letter to be popped. The subsequent pop operations take out the letters S, R, E, and so on. Thus, the resultant string is the reverse of the original one (see Figure 3.7).

Algorithm 3.1 String Reversal using Stack

```
reversal(s, str)

1. Set i = 0
2. While(i < length_of_str)
       Push str[i] onto the stack
       Set i = i + 1
   End While
3. Set i = 0
4. While(i < length_of_str)
       Pop the top element of the stack and store it in str[i]
       Set i = i + 1
   End While
5. Print "The reversed string is: ", str
6. End
```

Program 3.1: Program to reverse a given string using stacks

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 101
typedef struct stack
{
    char item[MAX];
    int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
void reversal(stk *, char *);
void push(stk *, char);
char pop(stk *);
```

```
void main()
{
    stk s;
    char str[MAX];
    int i;
    createstack(&s);
    clrscr();
    do
    {
        printf("Enter any string (max %d characters): ",
MAX-1);
        for(i=0;i<MAX;i++)
        {
            scanf("%c", &str[i]);
            if(str[i]=='\n')
                break;
        }
        str[i]='\0';
    }while(strlen(str)==0);
    reversal(&s, str);
    getch();
}
/*Function definitions*/
void createstack(stk *s)
{
    s->Top=-1;
}
void reversal(stk *s, char *str)
{
    int i;
    for (i=0;i<strlen(str);i++)
        push(s, str[i]);
    for(i=0;i<strlen(str);i++)
        str[i]=pop(s);
    printf("\nThe reversed string is: %s", str);
}
void push(stk *s, char item)
{
    s->Top++;
    s->item[s->Top]=item;
}
char pop(stk *s)
{
    char popped;
```

```
                            popped=s->item[s->Top];

                            s->Top--;

                            return popped;

                     }
```

Output of the program
```
Enter any string (max 100 characters): Hello World
The reversed string is: dlroW olleH
```

Converting Infix Notation to Postfix and Prefix Notations

Another important application of stacks is the conversion of expressions from the infix notation to the postfix and prefix notations. The general way of writing arithmetic expressions is known as the **infix notation** where the binary operator is placed between two operands on which it operates. (For simplicity, we have ignored expressions containing unary operators.) For example, the expressions `a+b` and `(a-c)*d`, `((a+b)*(d/f)-f)` are in the infix notation. The order of evaluation in these expressions depends on the parentheses and the precedence of operators. For example, the order of evaluation of the expression `(a+b)*c` is different from that of `a+(b*c)`. As a result, it is difficult to evaluate an expression in the infix notation. Thus, the arithmetic expressions in the infix notation are converted to another notation, which can be easily evaluated by a computer system to produce the correct result.

Jan Lukasiewicz, A Polish mathematician, suggested two alternative notations to represent an arithmetic expression. In these notations, the operators are written either before or after the operands on which they operate. The notation in which an operator occurs before its operands is known as the **prefix notation** (also known as **Polish notation**). For example, the expressions `+ab` and `*-acd` are in the prefix notation. On the other hand, the notation in which an operator occurs after its operands is known as the **postfix notation** (also known as the **Reverse Polish** or **suffix notation**). For example, the expressions `ab+` and `ac-d*` are in the postfix notation.

A characteristic feature of the prefix and postfix notations is that the order of evaluation of the expression is determined by the position of the operator and operands in the expression. That is, the operations are performed in the order in which the operators are encountered in the expression. Hence, parentheses are not required for the prefix and postfix notations. Moreover, while evaluating the expression, the precedence of the operators is insignificant. As a result, they are compiled faster than the expressions in the infix notation. Note that the expressions in the infix notation can be converted to both the prefix and postfix notation. This section discusses both types of conversions.

Conversion of Infix to Postfix Notation

To convert an arithmetic expression from an infix notation to a postfix notation, the precedence and associativity rules of operators are always kept in mind. The operators of the same precedence are evaluated from left to right. This conversion can be performed either manually (without using stacks) or by using stacks. The

steps for converting the expression manually are as follows:

1. The actual order of evaluation of the expression in the infix notation is determined by inserting parentheses in the expression according to the precedence and associativity of operators.

2. The expression in the innermost parentheses is converted into the postfix notation by placing the operator after the operands on which it operates.

3. Step 2 is repeated until the entire expression is converted into a postfix notation.

For example, to convert the expression `a+b*c` into an equivalent postfix notation, these steps are followed:

1. Since the precedence of `*` is higher than +, the expression `b*c` has to be evaluated first. Hence, the expression is written as

   `(a+(b*c))`

2. The expression in the innermost parentheses, that is, `b*c` is converted into its postfix notation. Hence, it is written as `bc*`. The expression now becomes

   `(a+bc*)`

3. Now the operator + has to be placed after its operands. The two operands for + operator are `a` and the expression `bc*`. The expression now becomes

   `(abc*+)`

Hence, the equivalent postfix expression is

   `abc*+`

When expressions are complex, manual conversion becomes difficult. On the other hand, the conversion of an infix expression into a postfix expression is simple when it is implemented through stacks. In this method, the infix expression is read from left to right, and a stack is used to store the operators and the left parenthesis temporarily. The order in which the operators are pushed onto and popped from the stack depends on the precedence of operators and the occurrence of parenthesis in the infix expression. The operands in the infix expression are not pushed onto the stack; rather they are directly placed in the postfix expression. Note that the operands maintain the same order as the original infix notation.

Algorithm 3.2 Infix to Postfix Conversion
```
infixtopostfix(s, infix, postfix)

1. Set i = 0
2. While (i < number of symbols in infix)
      If infix[i] is a whitespace or comma
           Set i = i + 1 and go to step 2
      If infix[i] is an operand, add it to postfix
      Else If infix[i] = '(', push it onto the stack
      Else If infix[i] is an operator, follow these steps:
        i. For each operator on the top of stack whose precedence is greater
           than or equal to the precedence of the current operator, pop the
           operator from stack and add it to postfix
        ii.  Push the current operator onto the stack
      Else If infix[i] = ')', follow these steps:
        i. Pop each operator from top of the stack and add it to postfix
           until '(' is encountered in the stack
        ii. Remove '(' from the stack and do not add it to postfix
      End If
      Set i = i + 1
   End While
3. End
```
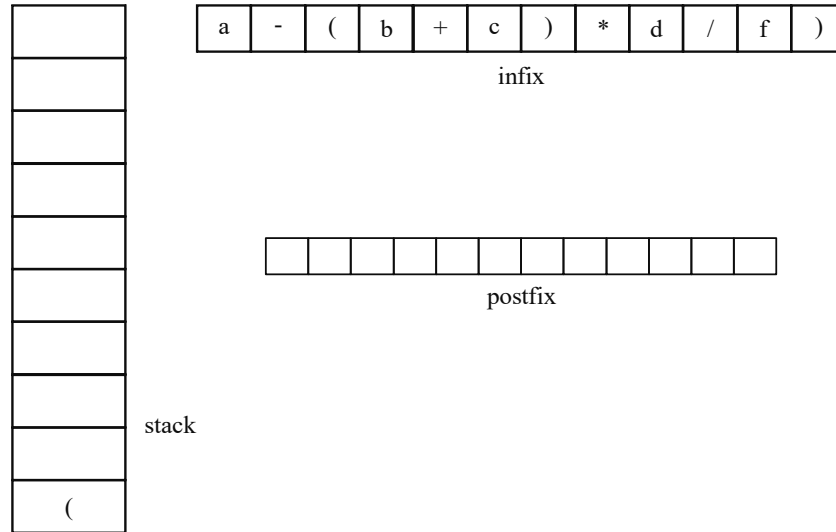
For example, consider the conversion of the following infix expression to the postfix expression:

```
a-(b+c)*d/f
```

Initially, a left parenthesis ' (' is pushed onto the stack, and the infix expression is appended with a right parenthesis ') '. The initial states of the stack, infix expression and postfix expression are shown in Figure 3.8.

Fig. 3.8 *Initial States of the Stack, Infix Expression and Postfix Expression*

infix is read from left to right and the following steps are performed:

1. The operand a is encountered, which is directly put to postfix.

2. The operator – is pushed onto the stack.

3. The left parenthesis ' (' is pushed onto the stack.

4. The next element is b, which being an operand is directly put to postfix.

5. + being an operator is pushed onto the stack.

6. Next, c is put to postfix.

7. The next element is the right parenthesis ') ' and hence, the operators on the top of stack are popped until ' (' is encountered in stack. Till now, the only operator in the stack above the ' (' is +, which is popped and put to postfix. ' (' is popped and removed from the stack [Figure 3.9 (a)]. Figure 3.9(b) shows the current position of the stack.
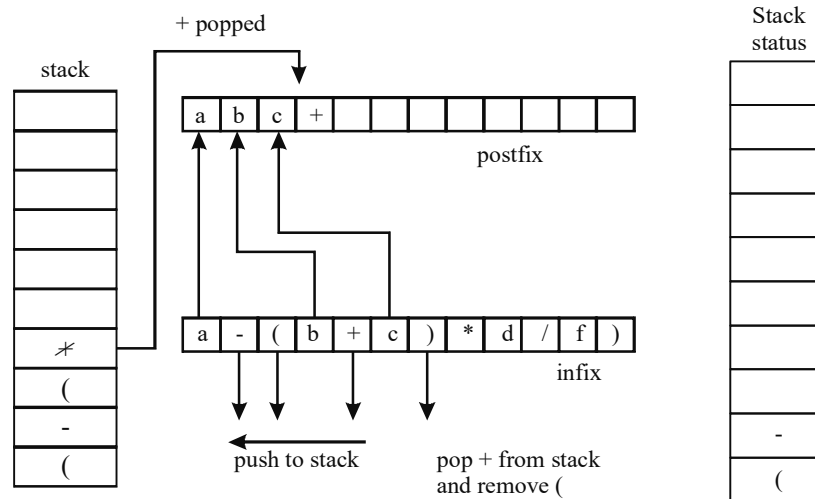
(a) *Postfix Expression when + is popped* **(b)** *State of the Stack*

Fig. 3.9 *Intermediate States of Postfix and Infix Expressions and the Stack*

8. After this, the next element $\ast$ is an operator and hence, it is pushed onto the stack.

9. Then, d is put to postfix.

10. The next element is /. Since the precedence of / is the same as the precedence of $\ast$, the operator $\ast$ is popped from the stack and / is pushed onto the stack (Figure 3.10).

11. The operand f is directly put to postfix after which ') ' is encountered.

12. On reaching ') ', the operators in the stack before the next ' (' is reached are popped. Hence, / and – are popped and put to postfix as shown in Figure 3.10.
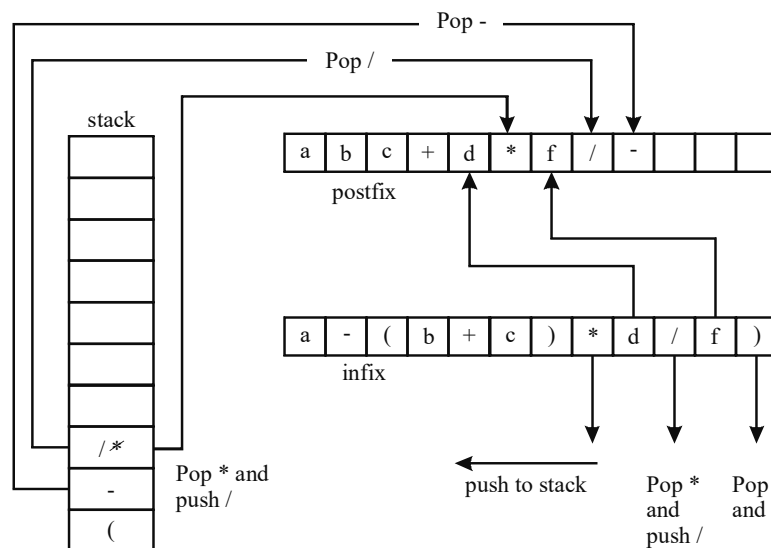
Fig. 3.10 *States When – and / are Popped*

13. ' (' is removed from the stack. Since the stack is empty, the algorithm is terminated and postfix is printed.

The step-wise conversion of the infix expression a- (b+c) *d/f into its equivalent postfix expression is shown in Table 3.5.

Table 3.5 *Conversion of Infix Expression into Postfix Expression*

Element	Action performed	Stack status	Postfix expression
a	Put to postfix	(	A
-	Push	(-	a
(	Push	(- (	a
b	Put to postfix	(- (	ab
+	Push	(- (+	ab
c	Put to postfix	(- (+	abc
)	Pop +, put to postfix, pop (	(-	abc+
*	Push	(-*	abc+
d	Put to postfix	(-*	abc+d
/	Pop *, put to postfix, push /	(-/	abc+d*
f	Put to postfix	(-/	abc+d*f
)	Pop / and -	Empty	abc+d*f/-

Program 3.2: Program to convert an expression from the infix notation to the postfix notation

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 102
typedef struct stack
{
    char item[MAX];
    int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
void infixtopostfix(stk *, char *, char *);
int precedence(char);
int isOperator(char);
void push(stk *, char);
char pop(stk *);
void main()
{
    stk s;
    char infix[MAX], *postfix;
    int i, len;
    clrscr();
    createstack(&s);
    do
    {
```

```
        printf("\nEnter expression in infix notation (max
%d characters): ", MAX-2);
        for(i=0;i<MAX-1;i++)
        {
            scanf("%c", &infix[i]);
            if(infix[i]=='\n')
                break;
        }
        infix[i]=')';
        infix[i+1]='\0';
    }while(strlen(infix)==0);
    push(&s, '(');
    len=strlen(infix);
    postfix=(char*)malloc(len+1);
    infixtopostfix(&s, infix, postfix);
    printf("\nThe equivalent postfix expression is %s",
postfix);
    getch();
}
/*Function definitions*/

void createstack(stk *s)
{
    s->Top=-1;
}
void infixtopostfix(stk *s, char *in, char *po)
{
    int i,j,len, preStack, preOp;
    char popped;
    len=strlen(in);
    i=j=0;
    while(i<len)
    {
if (in[i]==' '||in[i]=='\t' || in[i]==',')
{
        i++;
        continue;
    }
    if(in[i]=='(')
        push(s, in[i]);
    else if(isOperator(in[i]))
    {
        preStack=precedence(s->item[s->Top]);
        preOp=precedence(in[i]);
        while(preStack>=preOp)
```

```
            {
                po[j++]=pop(s);
                preStack=precedence(s->item[s->Top]);
            }
            push(s, in[i]);
        }
        else if(in[i]==')')
        {
            while((popped=pop(s))!='(')
            {
                po[j++]=popped;
            }
        }
        else
            po[j++]=in[i];
        i++;
    }
    po[j]='\0';
}
void push(stk *s, char item)
{
    s->Top++;
    s->item[s->Top]=item;
}

char pop(stk *s)
{
    char popped;
    popped=s->item[s->Top];
    s->Top--;
    return popped;
}
int isOperator(char op)
{
    switch(op)
    {
        case '^':
        case '+':
        case '-':
        case '*':
        case '/': return 1;
    }
    return 0;
}
```

```
int precedence(char op)
{
    switch(op)
    {
        case '^': return 3;
        case '/':
        case '*':
        case '%': return 2;
        case '+':
        case '-': return 1;
    }
    return 0;
}
```

Output of the program

```
Enter expression in infix notation (max 100 characters):
A+(B*C-(D/E^F)*)*H
The equivalent postfix expression is ABC*DEF^/*-H*+
```

Conversion of Infix to Orefix Notation

The conversion of an infix expression to a prefix expression is similar to the conversion of an infix expression to a postfix expression. The only difference is that the expression in the infix notation is scanned in the reverse order, that is, from right to left. Therefore, the stack in this case stores the operators and the closing (right) parenthesis.

Algorithm 3.3 Infix to Prefix Conversion

```
infixtoprefix(s, infix, prefix)

1. Set i = 0
2. While (i < number_of_symbols_in_infix)
       If infix[i] is a whitespace or comma
             Set i = i + 1 go to step 2
       If infix[i] is an operand, add it to prefix
       Else If infix[i] = ')', push it onto the stack
       Else If infix[i] is an operator, follow these steps:
       i. For each operator on the top of stack whose
          precedence is greater than or equal to the
          precedence of the current operator, pop the
          operator from stack and add it to prefix
       ii. Push the current operator onto the stack
       Else If infix[i] = '(', follow these steps:
       i. Pop each operator from top of the stack and add it
          to prefix until ')' is encountered in the stack
       ii.   Remove ')' from the stack and do not add it to
          prefix
       End If
       Set i = i + 1
   End While
3. Reverse the prefix expression
4. End
```

For example, consider the conversion of the following infix expression to a prefix expression:

```
a-(b+c)*d/f
```

The step-wise conversion of the expression `a-(b+c)*d/f` into its equivalent prefix expression is shown in Table 3.6. Note that initially, ') ' is pushed onto the stack, and ' (' is inserted in the beginning of the infix expression. Moreover, since the infix expression is scanned from right to left and the elements are inserted in the resultant expression from left to right, the prefix expression needs to be reversed.

Table 3.6 *Conversion of Infix Expression into Prefix*

Element	Action performed	Stack status	Prefix expression
f	Put to expression	)	f
/	Push	)/	f
d	Put to expression	)/	fd
*	Push	)/*	fd
)	Push	)/*)	fd
c	Put to expression	)/*)	fdc
+	Push	)/*)+	fdc
b	Put to expression	)/*)+	fdcb
(	Pop and + and put to expression, pop)	)/*	fdcb+
-		)-	fdcb+*/
a	Pop *, / and push -	)/*-	fdcb+a
(	Put to expression	Empty	fdcb+*/a-
	Pop - and put to expression, pop (		-a/*+bcdf
	Reverse the resultant expression		

The equivalent prefix expression is −a/*+bcdf.

Evaluation of Postfix Expression

In a computer system, when an arithmetic expression in an infix notation needs to be evaluated, it is first converted into its postfix notation. The equivalent postfix expression is then evaluated. The evaluation of postfix expressions is also implemented through stacks. Since the postfix expression is evaluated in the order of appearance of operators, parentheses are not required in the postfix expression. During the evaluation, a stack is used to store the intermediate results of evaluation.

Since an operator appears after its operands in a postfix expression, the expression is evaluated from left to right. Each element in the expression is checked whether it is an operator or an operand. If the element is an operand, it is pushed onto the stack. On the other hand, if the element is an operator, the first two operands are popped from the stack and the operation is performed on them. The result of the operation is then pushed back to the stack. This process is repeated until the entire expression is evaluated.

Algorithm 3.4 Evaluation of a Postfix Expression

```
evaluationofpostfix(s, postfix)

1. Set i = 0, RES=0.0
2. While (i < number_of_characters_in_postfix)
      If postfix[i] is a whitespace or comma
           Set i = i + 1 and continue
      If postfix[i] is an operand, push it onto the stack
      If postfix[i] is an operator, follow these steps:
           i. Pop the top element from stack and store it in operand2
           ii.     Pop the next top element from stack and store it
              in operand1
           iii.    Evaluate operand2 op operand1, and store the
              result in RES (op is the current operator)
           iv.     Push RES back to stack
      End If
      Set i = i + 1
   End While
3. Pop the top element and store it in RES
4. Return RES
5. End
```

For example, consider the evaluation of the following postfix expression using stacks:
    abc+d*f/-

where, a=6, b=3, c=6, d=5, f=9

After substituting the values of a, b, c, d and f, the postfix expression becomes
    636+5*9/-

The expression is evaluated as follows:

1. The expression is read from left to right and each element is checked to see whether it is an operand or an operator.

2. The first element is 6, which being an operand is pushed onto the stack.

3. Similarly, the operands 3 and 6 are pushed onto the stack.

4. The next element is +, which is an operator. Hence, the element at the top of the stack (6) and the next top element (3) are popped from the stack as shown in Figure 3.11.
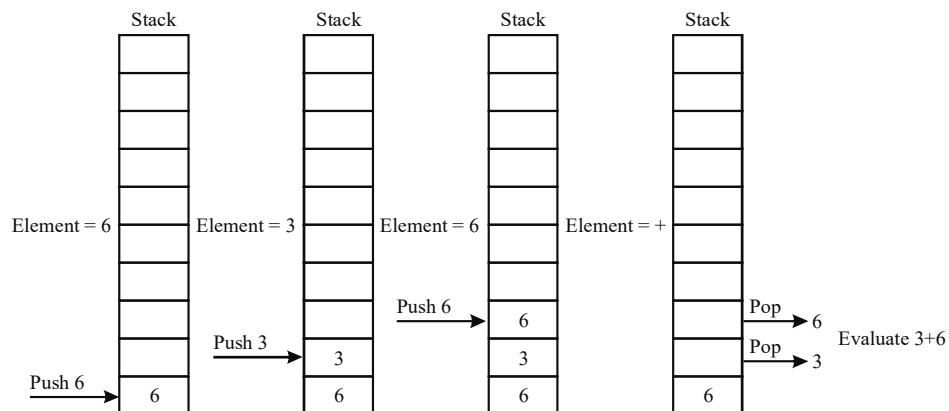
Fig. 3.11 *Evaluation of the Expression Using Stacks*

5. The expression 3+6 is evaluated and the result (that is, 9) is pushed back to the stack as shown in Figure 3.12.

6. The next element in the expression, that is 5, is pushed to the stack.

7. The next element is *, which is a binary operator. Hence, the stack is popped twice and the elements 5 and 9 are taken off from the stack as shown in Figure 3.12.
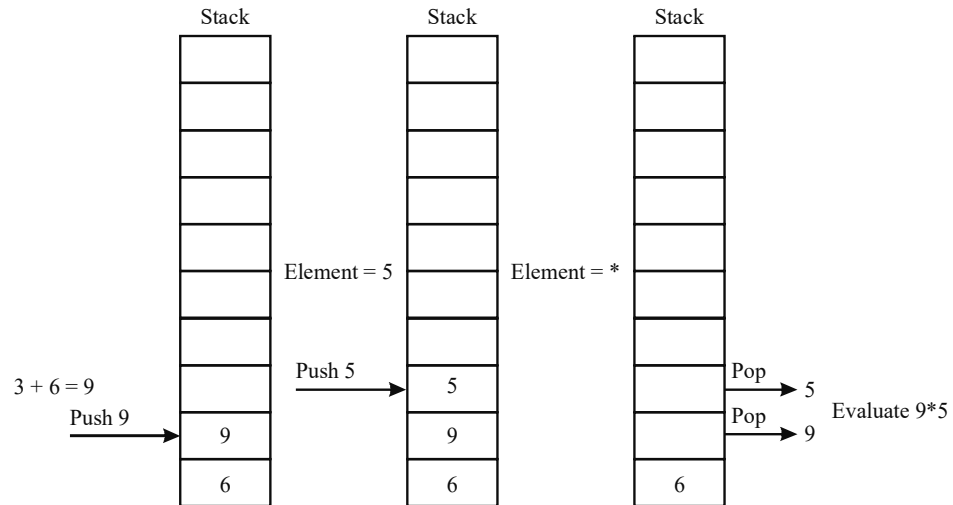

Fig. 3.12 *Popping 9 and 5 from Stack*

8. The expression 9*5 is evaluated and the result, that is, 45, is pushed back to the stack.

9. The next element in the postfix expression is 9, which is pushed onto the stack.

10. The next element is the operator /. Therefore, the two operands from the top of the stack, that is, 9 and 45, are popped from the stack, and the operation 45/9 is performed. The result 5 is again pushed to the stack.

11. The next element in the expression is −. Hence, 5 and 6 are popped from the stack and operation 6−5 is performed. The resulting value, that is, 1, is pushed to the stack (Figure 3.13).
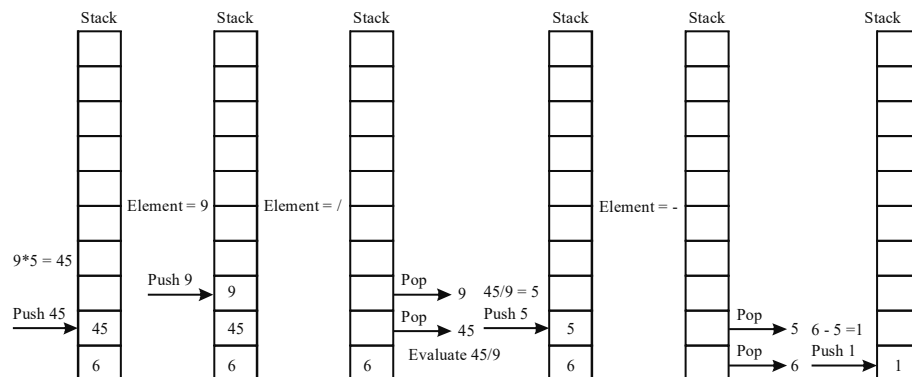

Fig. 3.13 *Final State of Stack with the Result*

12. There are no more elements to be processed in the expression. The element at the top of the stack is popped, which is the result of the evaluation of the postfix expression. Thus, the result of the expression is 1.

The step-wise evaluation of the expression 636+5*9/− is shown in Table 3.7.

Table 3.7 *Evaluation of the Postfix Expression*

Element	Action Performed	Stack Status
6	Push to stack	6
3	Push to stack	6 3
6	Push to stack	6 3 6
+	Pop 6	6 3
	Pop 3	6
	Evaluate 3+6=9	6
	Push 9 to stack	6 9
5	Push to stack	6 9 5
*	Pop 5	6 9
	Pop 9	6
	Evaluate 9*5=45	6
	Push 45 to stack	6 45
9	Push to stack	6 45 9
/	Pop 9	6 45
	Pop 45	6
	Evaluate 45/9=5	6
	Push 5 to stack	6 5
−	Pop 5	6
	Pop 6	EMPTY
	Evaluate 6-5=1	EMPTY
	Push 1 to stack	1
	Pop VALUE=1	EMPTY

Program 3.3: Program to evaluate a postfix expression

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<math.h>
#define MAX 102
typedef struct stack
{
    float item[MAX];
    int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
float evaluationofpostfix(stk *, char *);
void push(stk *, float);
float pop(stk *);
void main()
{
    stk s;
    char postfix[MAX];
    int i;
```

```
    float result;
    clrscr();
    createstack(&s);
    do
    {
        printf("\nEnter expression in postfix notation (max
%d characters): ", MAX-2);
        for(i=0;i<MAX-1;i++)
        {
            scanf("%c", &postfix[i]);
            if(postfix[i]=='\n')
                break;
        }
        postfix[i]='\0';
    }while(strlen(postfix)==0);
    result=evaluationofpostfix(&s, postfix);
    printf("\nThe result of postfix expression is %7.2f",
result);
    getch();
}
/*Function definitions*/
void createstack(stk *s)
{
    s->Top=-1;
}
float evaluationofpostfix(stk *s, char *po)
{
    int i, len;
    int number;
    float operand1, operand2;
    float res=0.0;
    len=strlen(po);
    i=0;
    while(i<len)
    {
if (po[i]==' '||po[i]=='\t' || po[i]==',')
{
        i++;
        continue;
    }
    if(isdigit(po[i]))
    {
        number=(int)(po[i]-'0');
        i++;
```

```
        while (isdigit(po[i]))
        {
            po[i]=(int)(po[i]-'0');    /*converting
char to int*/
            number=number*10;
            number+=po[i];
            i++;
        }
        push(s, number);
    }
    else
    {
        operand2=pop(s);
        operand1=pop(s);
        switch(po[i])
        {
            case '+': res=operand1+operand2;
                break;
            case '-': res=operand1-operand2;
                break;
            case '*': res=operand1*operand2;
                break;
            case '/': res=(float)operand1/operand2;
                break;
            case '%': res=(int)operand1%(int)operand2;
                break;
            case '^': res=pow(operand1, operand2);
                break;
            default: printf("\nIllegal expression...");
                getch();
                exit();
        }
        push(s, res);
    }
    i++;
    }
    res=pop(s);
    return res;
}
void push(stk *s, float item)
{
    s->Top++;
    s->item[s->Top]=item;
}
```

```
float pop(stk *s)
{
    float popped;
    popped=s->item[s->Top];
    s->Top--;
    return popped;
}
```

Output of the program

```
Enter expression in postfix notation (max 100 characters):
7 5 - 9 2 / *
The result of postfix expression is 9.00
```

3.3.2   Simulating Recursive Function using Stack

So far, we have discussed programs containing a single stack. If we need two or more stacks in a program, then it can be accomplished in two ways. One way is to have a separate array for each stack in the program. This approach has a disadvantage that if one stack needs to store a larger number of elements than the specified size and another stack has fewer elements, then it is not possible to store the elements of the first stack in the second stack in spite of the vacant space. This problem can be solved by having a single array of sufficient size to hold two or more stacks. For simplicity, we will discuss only the representation of two stacks in a single array. The two stacks can be represented efficiently in the same array if one stack grows from left to right and the other grows from right to left. The memory is also utilized more efficiently in this case.

For example, consider an array `stack[MAX]` to hold two stacks, say `stack1` and `stack2`. Two top variables `Top1` and `Top2` are required to represent the top of two stacks. Initially, `Top1` is set to "1 and `Top2` is set to `MAX` to represent empty stacks. The condition of overflow in this case occurs when the combined size of both the stacks exceeds `MAX`. For this, we use a variable `count` that keeps track of the number of elements stored in the array. Initially, `count` is set to 0. If the value of `count` exceeds the value of `MAX` and an attempt is made to insert a new element, overflow occurs.

To push an element in `stack1`, `Top1` is incremented by one and the element is inserted at that position. On the other hand, to push an element in `stack2`, `Top2` is decremented by one and the element is inserted at that position. To pop an element from `stack1`, the element at the position indicated by `Top1` is assigned to a local variable and then `Top1` is decremented by one. On the other hand, to pop an element from `stack2`, the element at the position indicated by `Top2` is assigned to a local variable and then `Top2` is incremented by one. Figure 3.14 shows an array of size `MAX` to hold two stacks, `stack1` and `stack2`, where `stack1` grows from left to right and `stack2` grows from right to left.
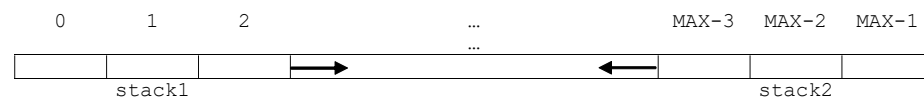
Fig. 3.14 *Representing two Stacks by an Array of Size MAX*

To represent two stacks in the same array, the following structure named `multistack` needs to be defined in C language:

```
struct multistack
{
    int item[MAX];
    int Top1, Top2;
    int count;
    int sno;    /*sno indicates the stack number (1 or
2)*/
};
```

Algorithm 3.5 Push Operation on Multi-stack

```
push(s, element)                 //s is a pointer to multi-stack

1. If (s->count == MAX)
      Print "Overflow: Stack is full!" and go to step 4
   End If
2. If(s->sno == 1)      //if element is to be inserted in stack1
      Set s->Top1 = s->Top1 + 1
      Set s->item[s->Top1] = element
      Print "Value is pushed onto stack1..."
      Set s->count = s->count + 1
   End If
3. If(s->sno == 2)
      Set s->Top2 = s->Top2 – 1
      Set s->item[s->Top2] = element
      Print "Value is pushed onto stack2..."
      Set s->count = s->count + 1
   End If
4. End
```

Algorithm 3.6 Pop Operation on Multi-stack

```
pop(s)                  //s is a pointer to multi-stack

1. If(s->sno == 1)
      If (s->Top1 == -1)
            Print "Underflow! Stack1 is empty"
            Return 0 and go to step 4
      Else
            Set popped = s->item[s->Top1]
            Set s->Top1 = s->Top1 - 1
            Set s->count = s->count - 1
      End If
   End If
2. If(s->sno == 2)
      If (s->Top2 == MAX)
            Print "Underflow! Stack2 is empty"
            Return 0 and go to step 4
      Else
            Set popped = s->item[s->Top2]
            Set s->Top2 = s->Top2 + 1
            Set s->count = s->count - 1
      End If
   End If
3. Return popped
4. End
```

Program 3.4: Program to implement multi-stacks using a single array

```c
#include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0
typedef struct multistack
{
    int item[MAX];
    int Top1, Top2;
    int count;
    int sno;    /*sno is the stack number (1 or 2)*/
}mstk;
/*Function prototypes*/
void createstack(mstk *);
void push(mstk *, int);
int pop(mstk *);
int isempty(mstk *);
int isfull(mstk *);
void main()
{
    int choice;
    int value;
    mstk s;
    createstack(&s);
    do{
        clrscr();
        printf("\n\tMain Menu");
        printf("\n1. Push");
        printf("\n2. Pop");
        printf("\n3. Exit\n");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: printf("\nEnter the stack number (1 or
2): ");
                scanf("%d", &s.sno);
                printf("\nEnter the value to be inserted:
");
                scanf("%d", &value);
                push(&s, value);
                getch();
                break;
```

```
            case 2: printf("\nEnter the stack number (1 or
2): ");
                    scanf("%d", &s.sno);
                    value=pop(&s);
                    if (value==0)
                    {
                        if(s.sno==1)
                            printf("\nUnderflow: Stack1 is
empty!");
                        else if(s.sno==2)
                            printf("\nUnderflow: Stack2 is
empty!");
                    }
                    else
                        printf("\nPopped element is: %d", value);
                    getch();
                    break;
            case 3: exit();
            default: printf("\nInvalid choice!");
        }
    }while(1);
}
/*Function definitions*/
void createstack(mstk *s)
{
    s->Top1=-1;
    s->Top2=MAX;
    s->count=0;
}
void push(mstk *s, int item)
{
    if (isfull(s))
    {
        printf("\nOverflow: Stack is full!");
        return;
    }
    if(s->sno==1)
    {
        s->Top1++;
        s->item[s->Top1]=item;
        printf("\nValue is pushed onto stack1...");
        s->count++;
    }
    if(s->sno==2)
```

```
        {
            s->Top--;
            s->item[s->Top2]=item;
            printf("\nValue is pushed onto stack2...");
            s->count++;
        }
    }
    int pop(mstk *s)
    {
        int popped;
        if (isempty(s))
            return 0;
        if(s->sno==1)
        {
            popped=s->item[s->Top1];
            s->Top--;
            s->count—;
        }
        if(s->sno==2)
        {
            popped=s->item[s->Top2];
            s->Top2++;
            s->count--;
        }
        return popped;
    }
    int isempty(mstk *s)
    {
        int r;
        if(s->sno==1)
        {
            if (s->Top1==-1)
                r=True;
            else
                r=False;
        }
        if(s->sno==2)
        {
            if (s->Top2==MAX)
                r=True;
            else
                r=False;
        }
```

```
        return r;
    }


    int isfull(mstk *s)
    {
        if (s->count==MAX)
            return True;
        else return False;
    }
```

Output of the program

```
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the stack number (1 or 2): 1
Enter the value to be inserted: 34
Value is pushed onto stack1...
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the stack number (1 or 2): 2
Enter the value to be inserted: 45
Value is pushed onto stack2...
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the stack number (1 or 2): 1
Enter the value to be inserted: 23
Value is pushed onto stack1...
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 1
Popped element is: 23
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
```

```
Enter the stack number (1 or 2): 2
Popped element is: 45
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 1
Popped element is: 34
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 1
Underflow: Stack1 is empty!
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 2
Underflow: Stack2 is empty!
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 3
```

Check Your Progress

8. What do you understand by the 'push' operation in a stack?

9. What do you understand by the 'pop' operation in a stack?

10. How can a stack be implemented?

11. State some applications of stacks.

12. In a stack, what does the condition Top=-1 indicate?

13. How can an infix expression be converted to a postfix expression?

3.4 QUEUES

A queue is a linear data structure that is used to represent a linear list and permits deletion to be performed at one end of the list (known as front of the queue) and the insertion at the other (known as rear of the queue). The information in such a

list is processed in the same order as it was received, that by the FIFO or FCFS method.

Therefore, a queue is a type of collection in which the entities contained in it are kept in order and the principal operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. Queues provide services in computer science, transport and operations research where various entities such as data, objects, persons or events are stored in order to be processed later. In such a scenario, the queue performs the function of a buffer.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.

Representing a Queue

The primary characteristic of a queue data structure is that it allows access only to the front and back of the structure. In this manner, an appropriate metaphor often used to represent queues is the idea of a checkout line.

Consider the example of a queue in a departmental store to understand this concept. The customer or object at the front of the line was the first one to enter, while the one at the end of the line is the last to have entered. Every time one customer finishes paying for their items, he leaves the queue from the front. This represents the queue–dequeue function. Every time another customer enters the line to wait, they join the end of the line and represent the 'enqueue' function (Figure 3.15). The 'queue size' function returns the length of the line and the 'empty' function would turn out to be true only if there was nothing in the line.
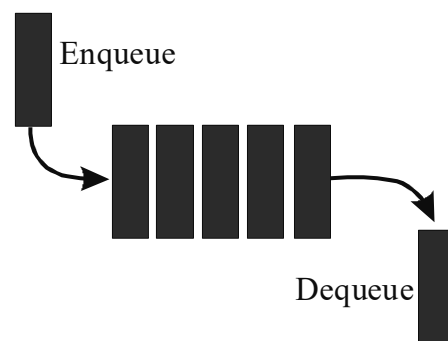
Fig. 3.15   Representation of a FIFO Queue

Primitive Operations on Queue

Enqueue (*new-item*:item-type)**:** It adds an item onto the end of the queue.

front():item-type**:** It returns the item at the front of the queue.

dequeue()**:** It removes the item from the front of the queue.

is-empty():Boolean**:** It is true if no more items can be de-queued and there is no front item.

is-full():Boolean**:** It is true if no more items can be enqueued.

get-size():Integer: It returns the number of elements in the queue.

All operations except get-size() can be performed in $O(1)$ time. get-size() runs in at worst $O(N)$.

Implementation of Queues

A priority queue is a variation on the queue which does not qualify for the name FIFO because it is not accurately descriptive of that data structure's behavior. Queueing theory encompasses the more general concept of queue, as well as interactions between strict-FIFO queues

Theoretically, one characteristic of a queue is that it does not have a specific capacity. Inspite of the number of elements already contained, there is always scope to add a new one. It can also be empty, at which point removing an element will not be possible until one has been added again. A practical implementation of a queue, like with pointers, does have some capacity limit, that is dependent on the concrete situation it is used in. For a data structure the executing computer will eventually run out of memory, thus, limiting the queue size. A queue overflow results from an attempt to add an element onto a full queue and queue underflow occurs when trying to remove an element from an empty queue. A bounded queue is a queue limited to a fixed number of items.

Two common ways in which queues may be implemented are as follows:

- Arrays
- Pointers (one-way linear-linked list)

Linked-List Implementation

The basic linked-list implementation uses a singly-linked list with a tail pointer to keep track of the back of the queue.

```
type Queue<item_type>
  data list:Singly Linked List<item_type>
  data tail:List Iterator<item_type>

  constructor()
    list := new Singly-Linked-List()
    tail := list.get-begin() # null
  end constructor
```

Enqueue

The action 'enqueue' means to add something to a queue or a list of to-do processes for a program.

When you want to enqueue an object, you add it to the back of the item pointed to by the tail pointer. So the previous tail is taken to be the next one as compared to the item being added and the tail pointer points to the new item. If the list is empty, this doesn't work as the tail iterator doesn't refer to anything.

```
method enqueue(new_item:item_type)
  if is-empty()
```

```
      list.prepend(new_item)
       tail := list.get-begin()
     else
      list.insert_after(new_item, tail)
      tail.move-next()
     end if
   end method
```

The item in the front of the queue is the one referred to by the linked list's head pointer.

```
method front():item_type
    return list.get-begin().get-value()
  end method
```

Double-Ended Queue (Dequeue)

According to Donald Knuth, in computing, a double-ended queue is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

When an item needs to be dequeued off the list, the head pointer is pointed to the previous item from the head. The old head item is the one that has been removed of the list. If the list is now empty, you have to fix the tail iterator.

```
method dequeue()
   list.remove-first()
   if is-empty()
     tail := list.get-begin()
   end if
  end method
```

Empty Queue

An empty list contains no data records. It is usually the same as believing that it has zero nodes. A list is usually said to be empty when it has only sentinel nodes. A check for emptiness is quite easy.

```
method is-empty():Boolean
    return list.is-empty()
  end method
```

Full Queue

A check for a full list is simple. Linked lists are considered to be limitless in size.

```
method is-full():Boolean
    return False
  end method
```

This is different from the queue abstract data type or FIFO, where elements can only be added to one end and removed from the other. This data class has the following possible sub-types:

- An **input-restricted dequeue** where removal can be made from both ends but input can only be made at one end.

- An **output-restricted dequeue** where input can be made at both ends but output can be made from one end only.

3.4.1 Circular Queue

As discussed earlier, in case of a queue represented as an array, once the value of the rear reaches the maximum size of the queue, no more elements can be inserted. However, there may be the possibility that space on the left of the front index is vacant. Hence, in spite of space on the left of front being empty, the queue is considered to be full. This wastage of space in the array implementation of a queue can be avoided by shifting the elements to the beginning of an array, if the space is available. In order to do this, the values of Rear and Front indices have to be changed accordingly. However, this is a complex process and difficult to implement. An alternative solution to this problem is to implement a queue as a circular queue.

The array implementation of a circular queue is similar to the array implementation of a queue. The only difference is that as soon as the rear index of the queue reaches the maximum size of the array, Rear is reset to the beginning of the queue, provided it is free. A circular queue is full only when all the locations in the array are occupied. A circular queue is shown in Figure 3.16.
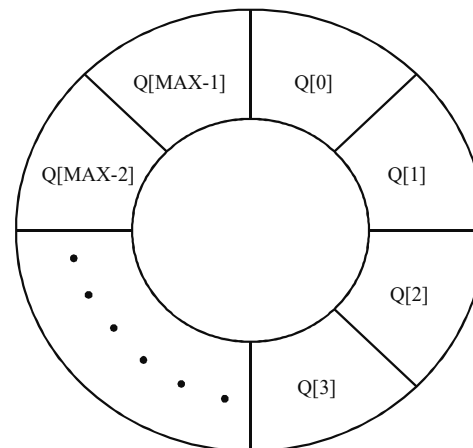
Fig. 3.16 *A Circular Queue*

Note: A circular queue is generally implemented as an array, though it can also be implemented as a circular linked list.

To understand the operations on a circular queue, consider a circular queue represented in the memory by the array CQueue[MAX]. Rear and Front are used to store the indices of the rear and front elements of CQueue, respectively. Initially, both Rear and Front are set to "1 to indicate an empty queue.

Whenever an element is to be inserted in a circular queue, Rear is incremented by one. However, if the value of the Rear index is MAX"1, instead of incrementing Rear, it is reset to the first index of the array if space is available in the beginning. Hence, if any location to the left of the Front index is empty, an

element can be added to the queue at an index starting from 0. The queue is considered full in the following cases:

- When the value of `Rear` equals the maximum size of the array and `Front` is at the beginning of the array
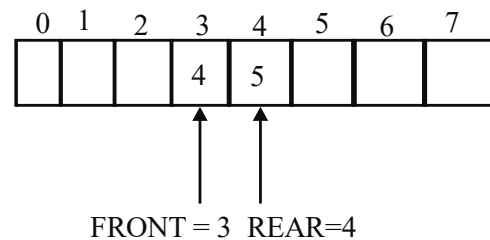- When the value of `Front` is one more than the value of `Rear`

Whenever an element is to be deleted from the queue, `Front` is incremented by one. However, if the value of `Front` is MAX"1, it is reset to the 0th position in the array. When the value of `Front` equals the value of `Rear` (other than "1), it indicates that there is only one element in the queue. On deleting the last element, both `Rear` and `Front` are reset to "1 to indicate an empty queue. Figure 3.17 shows the various states of a queue after some insert and delete operations.

FRONT = –1 REAR = –1

(a) *An Empty Queue*

FRONT = 0   REAR=4
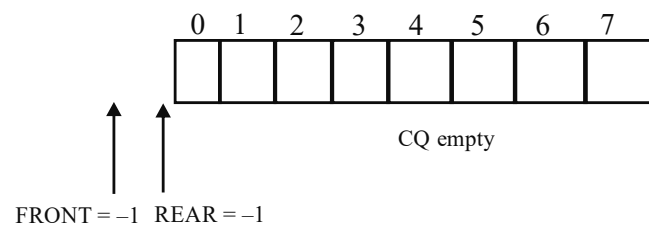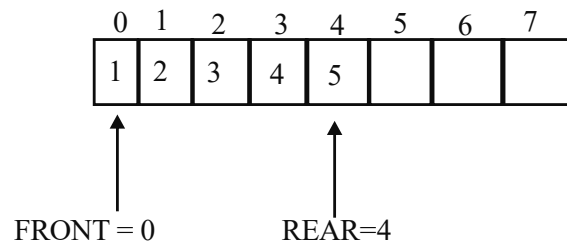
(b) *Queue after Inserting a Few Elements*

FRONT = 3 REAR=4

(c) *Queue after Deleting a Few Elements*

FRONT = 3   REAR = 7
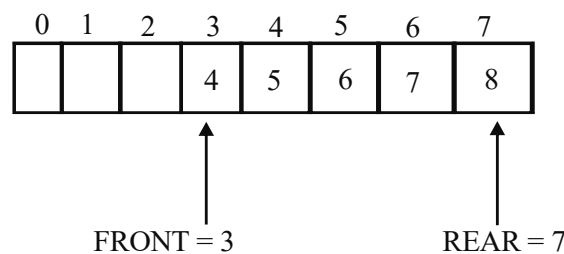
(d) *Queue when Rear = MAX"1*

0	1	2	3	4	5	6	7
9			4	5	6	7	8

REAR = 0   FRONT = 3

(e) *Rear is reset to Zero*

0	1	2	3	4	5	6	7
9	10	11	4	5	6	7	8

CQ full

FRONT = 2   REAR = 3

(f) *Queue Full*

0	1	2	3	4	5	6	7
9	10	11					

FRONT = 0   REAR = 2

(g) *Front is reset to Zero*

0	1	2	3	4	5	6	7
		11					

Delete →

0	1	2	3	4	5	6	7

CQ empty

FRONT = 2   REAR = 2   FRONT = –1   REAR = –1
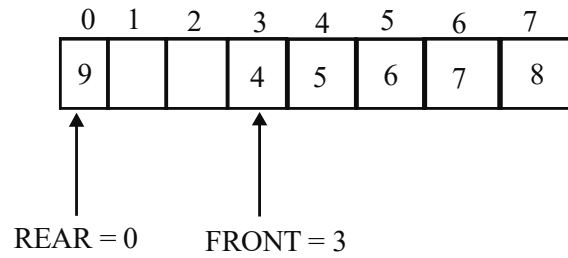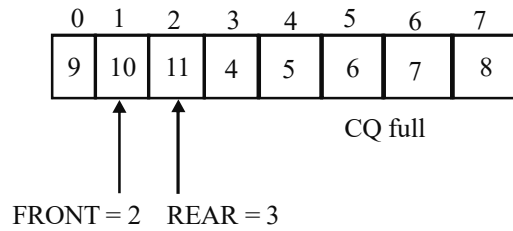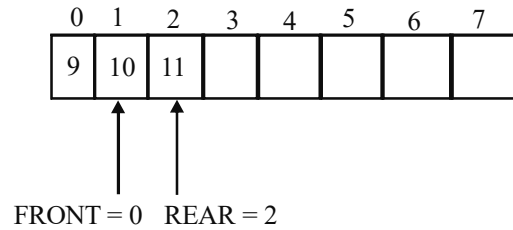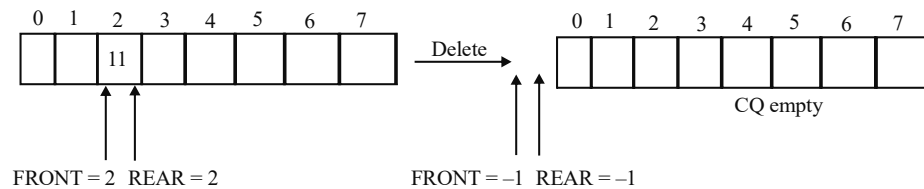
(h) *Queue after Deleting the Only Element*

Fig. 3.17 *Various States of a Circular Queue after the Insert and Delete Operations*

The total number of elements in a circular queue at any point of time can be calculated from the current values of the Rear and the Front indices of the queue. In case, Front<Rear, the total number of elements = Rear – Front + 1. For instance, in Figure 3.18(a), Front = 3 and Rear = 7. Hence, the total number of elements in CQueue at this point of time is 7 " 3 + 1 = 5. In case, Front > Rear, the total number of elements = Max + (Rear " Front) + 1. For instance, in Figure 3.18(b), Front = 3 and Rear = 0. Hence, the total number of elements in CQueue is 8 + (0 " 3) + 1.
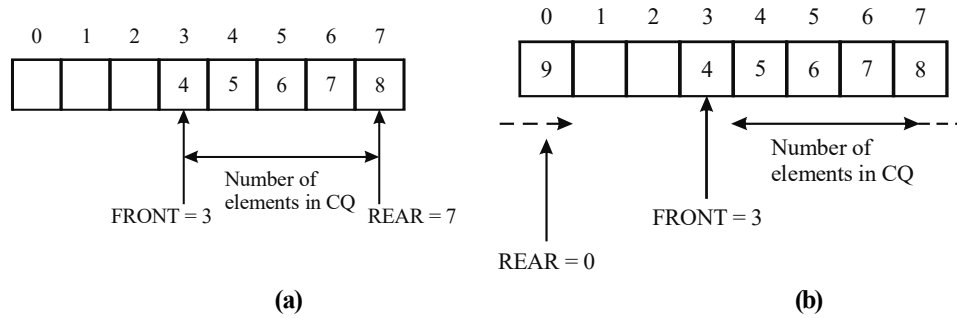
Fig. 3.18 Number of Elements in a Circular Queue

Algorithm 3.7 Insert Operation on a Circular Queue

```
qinsert(q, val)

1. If ((q->Rear = MAX-1 AND q->Front = 0) OR (q->Rear + 1 = q-
                      >Front))
     Print "Overflow: Queue is full!" and go to step 5
    End If //check if circular queue is full
2. If q->Rear = MAX-1    // check if rear is MAX-1
     Set q->Rear = 0
    Else
     Set q->Rear = q->Rear + 1     //increment rear by one
    End If
3. Set q->CQueue[q->Rear] = val     //val  is the value to be
                      inserted in the queue
4. If q->Front = -1     //check if queue is empty
     Set q->Front = 0
    End If
5. End
```

Algorithm 3.8 Delete Operation on a Circular Queue

```
qdelete(q)

1. If q->Front = -1
     Print "Underflow: Queue is empty!"
     Return 0 and go to step 5
    End If
2. Set del_val = q->CQueue[q->Front] //del_val is the value to be
                      deleted
3. If  q->Front = q->Rear     // check if there is one element
                      in the queue
     Set q->Front = q->Rear = -1
    Else
     If q->Front = MAX-1
          Set q->Front = 0
     Else
          Set q->Front = q->Front +1
     End If
    End If
4. Return del_val
5. End
```

Program 3.5: A program to implement a circular queue:

```
#include<stdio.h>
#include<conio.h>
#define MAX 4
#define True 1
```

```
#define False 0
typedef struct queue
{
    int CQueue[MAX];
    int Front;
    int Rear;
}que;
/* Function prototypes */
void createqueue(que *);
void qinsert(que *,int);
int qdelete(que *);
void qdisplay(que);
int isempty(que);
int isfull(que);
void main()
{
    que q;
    int choice,element,val;
    createqueue(&q);
    do
    {
        clrscr();
        printf("\n\n\tMain Menu");
        printf("\n1. Insert");
        printf("\n2. Delete");
        printf("\n3. Exit\n");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: printf("\nEnter the value to be inserted:
");
                scanf("%d", &element);
                qinsert(&q, element);
                getch();
                break;
            case 2: val=qdelete(&q);
                if(val==0)
                    printf("\nUnderflow: Queue is empty!");
                else
printf("\nThe value of deleted item is: %d\n", val);
                getch();
                break;
            case 3: exit();
```

```
            default: printf("Invalid choice");
        }
    } while(1);
 }
void createqueue(que *q)
{
    q->Front=q->Rear=-1;
}
void qinsert(que *q, int val)
{
    if(isfull(*q))
    {
       printf("\nOverflow: Queue is full!");
       return;
    }
    if(q->Rear==MAX-1)
       q->Rear=0;
    else
        (q->Rear)++;
    q->CQueue[q->Rear]=val;
    if(isempty(*q))
        q->Front=0;
    qdisplay(*q);
}
int qdelete(que *q)
{
    int del_val;
    if(isempty(*q))
        return 0;
    del_val=q->CQueue[q->Front];
    if(q->Front==q->Rear)
       q->Front=q->Rear=-1;
    else
    {
       if(q->Front==MAX-1)
           q->Front=0;
       else
           (q->Front)++;
    }
        return del_val;
 }
void qdisplay(que q)
{
```

```
            int i;
            printf("\nFront: %d, Rear: %d", q.Front, q.Rear);
            printf("\n\nQueue is: ");
            if(q.Front<=q.Rear)
                for(i=q.Front; i<=q.Rear; i++)
                    printf("%d ", q.CQueue[i]);
            else
            {
                for(i=0; i<=q.Rear; i++)
                    printf("%d ", q.CQueue[i]);
                for(i=q.Front; i<MAX; i++)
                    printf("%d ", q.CQueue[i]);
            }
    }
    int isempty(que q)
    {
    if(q.Front==-1)
            return True;
    else
            return False;
     }


    int isfull(que q)
    {
        if((q.Rear==MAX-1 && q.Front==0)||(q.Rear+1==q.Front))
            return True;
        else
            return False;
    }
```

The output of the program is:

```
Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 1
Front: 0, Rear: 0
Queue is: 1

Main Menu
1. Insert
2. Delete
```

```
3. Exit
Enter your choice: 1
Enter the value to be inserted: 2
Front: 0, Rear: 1
Queue is: 1 2

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 3
Front: 0, Rear: 2
Queue is: 1 2 3

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted item is: 1

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 7
Front: 1, Rear: 3
Queue is: 2 3 7

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 4
Front: 1, Rear: 0
Queue is: 4 2 3 7

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 3
```

3.4.2 Deques

A deque (short form of double-ended queue) is a linear list in which elements can be inserted or deleted at either end but not in the middle. That is, elements can be inserted/deleted to/from the rear end or the front end. Figure 3.19 shows the representation of a deque.
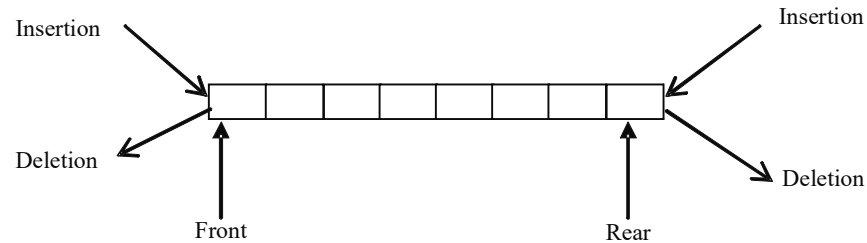
Fig. 3.19 *A Deque*

Like a queue, a deque can be represented as an array or a singly-linked list. Here, we will discuss the array implementation of a deque.

Algorithm 3.9 Insert Operation in the Beginning of a Deque

```
qinsert_beg(q, val)

1. If (q->Rear = MAX-1 AND q->Front = 0)
       Print "Overflow: Queue is full!" and go to step 4
     End If
2. If q->Front = -1
       Set q->Front = q->Rear = 0
       Set q->DQue[q->Front] = val
     End If
3. If q->Rear != MAX -1        //check if last position is
     occupied
       Set num_item = q->Rear - q->Front + 1    // num_item is
total number of elements
       Set i = q->Rear +1
             Set j = 1
       While j <= num_item
             Set q->DQue[i] = q->DQue[i-1]  //shift elements one
space to the right
             Set i = i - 1
             Set j = j + 1
       End While
       Set q->DQue[i] = val
       Set q->Front = i
       Set q->Rear = q->Rear + 1
     Else
       Set q->Front = q->Front  - 1
       Set q->DQue[q->Front] = val
     End If
4. End
```

Algorithm 3.10 Insert Operation at the End of a Deque

```
qinsert_end(q, val)

1. If  (q->Rear = MAX-1 And q->Front = 0) //check if queue is
                        full
       Print  "Overflow: Queue is full!" and go to step 4
     End If
2. If q->Front = -1     //check if queue is empty
     Set q->Front = q->Rear =0
     Set q->DQue[q->Front] = val and go to step 4
     End If
3. If q->Rear = MAX-1   // check if last position is occupied
     Set i = q->Front -1
     While i < q->Rear       //shift elements one place to the
left of queue
           Set q->DQue[i] = q->DQue[i+1]
           Set i = i +1
     End While
     Set q->DQue[q->Rear] = val
     Set q->Front = q->Front - 1
     Else
     Set q->Rear = q->Rear +1
     Set q->DQue[q->Rear] = val
     End If
 4. End
```

Algorithm 3.11 Delete Operation in the Beginning of a Deque

```
qdelete_beg(q)

1. If q->Front = -1
     Print "Underflow: Queue is empty!"
     Return 0 and go to step 5
     End If
2. Set del_val = q->DQue[q->Front]
3. If q->Front = q->Rear
     Set q->Front = q->Rear = -1
     Else
     Set q->Front = q->Front  + 1
     End If
4. Return del_val
5. End
```

Algorithm 3.12 Delete Operation from the End of a Deque

```
qdelete_end(q)

1. If q->Front = -1
     Print "Underflow: Queue is empty!"
     Return 0 and go to step 5
     End If
2. Set del_val = q->DQue[q->Rear]
3. If q->Front = q->Rear
     Set q->Front = q->Rear = -1
      Else
     Set q->Rear = q->Rear - 1
     If q->Rear = -1
           Set q->Front = -1
     End If
     End If
4. Return del_val
5. End
```

The two variations of a deque are as follows:

- **Input restricted deque:** It allows insertion of elements only at one end, but deletion can be done at both ends.

- **Output restricted deque:** It allows deletion of elements only at one end, but insertion can be done at both ends.

The implementation of both these queues is similar to the implementation of a deque. The only difference is that in an input restricted queue, the function for insertion in the beginning is not needed, whereas in an output restricted queue, the function for deletion in the beginning is not needed.
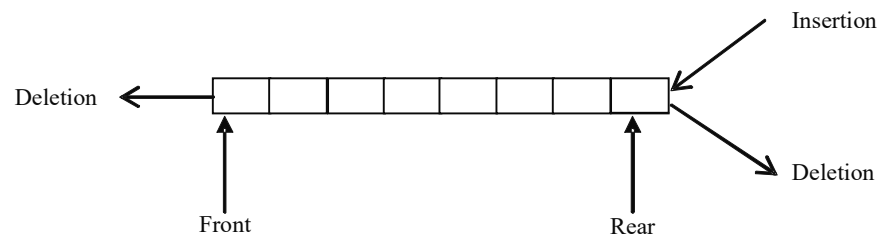
Fig. 3.20 Input Restricted Deque
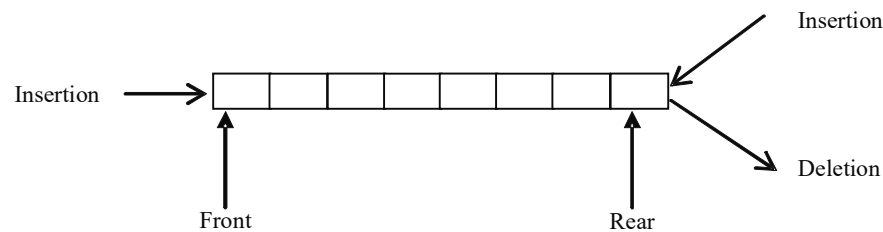
Fig. 3.21 Output Restricted Deque

3.4.3 Priority Queue

A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority. While implementing a priority queue, the following two rules are applied:

- The element with higher priority is processed before any element of lower priority.
- The elements with the same priority are processed according to the order in which they were added to the queue.

A priority queue can be represented in many ways. Here we will discuss the multi-queue implementation of a priority queue.

Multi-Queue Implementation

In a multi-queue representation of a priority queue, for each priority, a queue is maintained. The queue corresponding to each priority can be represented in the same array of sufficient size. For each queue, two variables `Fronti` and `Reari` are maintained (see Figure 3.22). Observe that `Fronti` and `Reari` correspond to the front and rear positions of the queue for priority `Priorityi`.
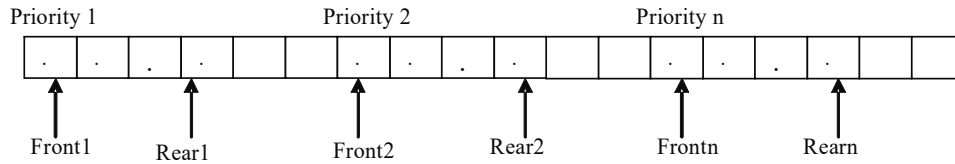
Fig. 3.22 Multi-queue Representation of a Priority Queue

Clearly, in this representation, shifting is required to make space for an element to be inserted. To avoid this shifting, an alternative representation can be used (see Figure 3.23). In this representation, instead of representing a queue corresponding to each priority using a single array, a separate array for each priority is maintained. Each queue is implemented as a circular array and has its own two variables, `Front` and `Rear`. The element with the given priority number is inserted in the corresponding queue. Similarly, whenever an element is to be deleted from a queue, it must be the element from the highest priority queue. Note that the lower priority number indicates the higher priority.
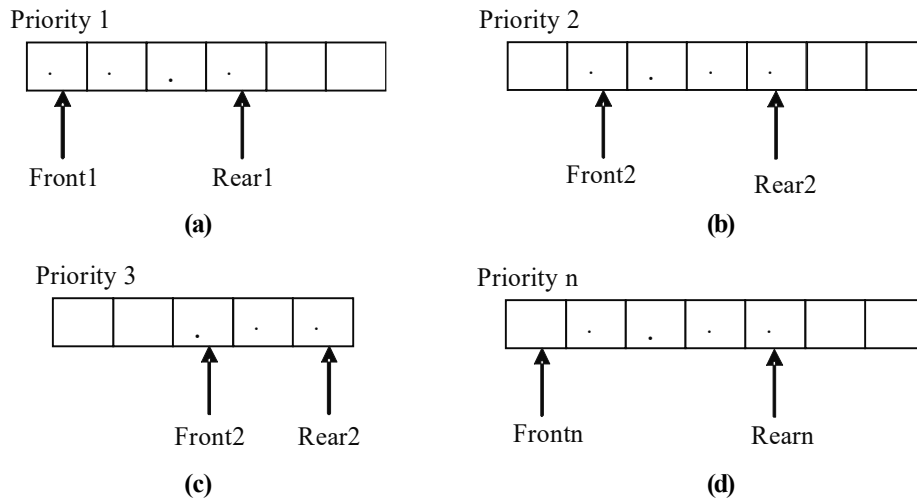
Fig. 3.23 Multiple Queues according to the Priority

If the size of each queue is the same, then instead of multiple one-dimensional arrays, a single two-dimensional array can be used where row `i` corresponds to the queue of priority `i`. In addition, two single-dimensional arrays are used. One is used to keep track of the front position and another to keep track of the rear position of each queue (see Figure 3.24).
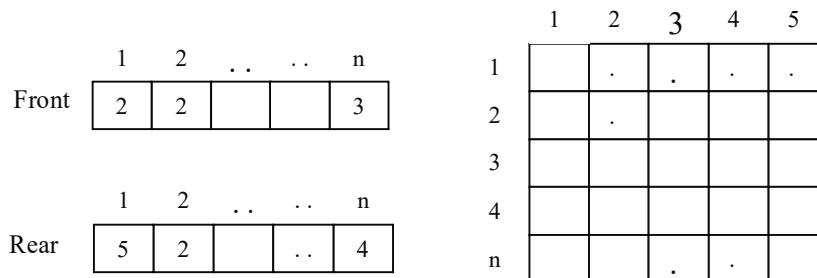
Fig. 3.24 Priority Queue as a Two-dimensional Array

Algorithm 3.13 Insert Operation in a Priority Queue

```
qinsert(q, val, prno)   //prno is the priority of val

1. If (q->Rear[prno] = MAX-1 AND q->Front[prno] = 0) OR (q-
   >Rear[prno]+1 = q->Front[prno])
            Print "Overflow: Queue full!" and go to step 5
        End If
2. If q->Rear[prno-1] = MAX-1
            Set q->Rear[prno-1] = 0
     Else
     Set q->Rear[prno-1] = q->Rear[prno-1] + 1
    End If
3. Set q->CQueue[prno-1][q->Rear[prno-1]] = val
4. If q->Front[prno-1] = -1
     Set q->Front[prno-1] = 0
     End If
5. End
```

Algorithm 3.14 Delete Operation in a Priority Queue

```
qdelete(q)

1. Set flag = 0, i = 0
2. While i <= MAX-1
            If NOT (q->Front[prno]) = -1  //check if queue is not
empty
                 Set flag = 1
                 Set del_val = q->CQueue[i][q->Front[i]]

                 If q->Front[i] = q->Rear[i]
                       Set q->Front[i] = q->Rear[i] = -1

                       Else If q->Front[i] = MAX-1

                            Set q->Front[i] = 0

                              Else
                                  Set q->Front[i] = q->Front[i]
                   + 1                 End If
                 End If
                 Break //jump out of while loop
            End If
            Set i = i +1
        End While
3. If flag = 0
     Print "Underflow: Queue is empty!"
     Return 0 and go to step 4
      Else
     Return del_val
      End If
4. End
```

Program 3.6: A program to implement a priority queue using a two-dimensional array:

```c
#include<stdio.h>
#include<conio.h>
#define MAX 5
#define True 1
#define False 0
typedef struct queue
{
```

```
    int CQueue[MAX][MAX];
    int Front[MAX];
    int Rear[MAX];
}que;
void createqueue(que *);
void qinsert(que *, int, int);
int qdelete(que *);
void qdisplay(que, int);
int isempty(que, int);
int isfull(que, int);
void main()
{
    que q;
    int choice,element,pno,val;
    createqueue(&q);
    do
    {
      clrscr();
      printf("\n\n\tMain Menu");
      printf("\n1. Insert");
      printf("\n2. Delete");
      printf("\n3. Exit\n");
      printf("\nEnter your choice: ");
      scanf("%d", &choice);
      switch(choice)
      {
          case 1: printf("\nEnter the value and its
priority: ");
              scanf("%d%d", &element, &pno);
              qinsert(&q, element, pno);
              getch();
              break;
          case 2: val=qdelete(&q);
              if(val==0)
                 printf("\nUnderflow: Queue is empty!");
              else
printf("\nThe Deleted item is: %d\n",val);
              getch();
              break;
          case 3: exit();
          default: printf("Invalid choice");
      }
    } while(1);
}
```

```
void createqueue(que *q)
{
int i;
for(i=0;i<MAX;i++)
q->Front[i]=q->Rear[i]=-1;
}
int isempty(que q, int prno)
{
if(q.Front[prno]==-1)
        return True;
else
return False;
}
int isfull(que q, int prno)
{
if((q.Rear[prno]==MAX-1                    &&
q.Front[prno]==0)||(q.Rear[prno]+1==q.Front[prno]))

return True;
else
return False;
}
void qinsert(que *q,int val,int prno)
{
if(isfull(*q, prno))
{
printf("\nOverflow: Queue is full!");
return;
}
if(q->Rear[prno-1]==MAX-1)
q->Rear[prno-1]=0;
else
(q->Rear[prno-1])++;
q->CQueue[prno-1][q->Rear[prno-1]]=val;
    if(isempty(*q, prno))
        q->Front[prno-1]=0;
qdisplay(*q, prno);
}
int qdelete(que *q)
{
int del_val, i, prno,flag=0;
for(i=0;i<= MAX-1;i++)
{
if(!isempty(*q,i))
```

```
{
flag=1;
del_val=q->CQueue[i][q->Front[i]];
if(q->Front[i]==q->Rear[i])
q->Front[i]=q->Rear[i]=-1;
else if(q->Front[i]==MAX-1)
q->Front[i]=0;
else
q->Front[i]++;
prno =i+1;
break;
}
}
if(flag==0)
return 0;
else
{
printf("\nPriority of deleted item is: %d\n",prno);
return del_val;
}
}
void qdisplay(que q, int prno)
{
int i;
printf("\nFront: %d,  Rear: %d", q.Front[prno-1],
q.Rear[prno-1]);
printf("\n\nQueue for prno %d is: ", prno);
if(q.Front[prno-1]<=q.Rear[prno-1])
{
for(i=q.Front[prno-1];
          i<=q.Rear[prno-1]; i++)
printf("%d ",q.CQueue[prno-1][i]);
}
else
{
for(i=0; i<=q.Rear[prno-1]; i++)
printf("%d ", q.CQueue[prno-1][i]);
for(i=q.Front[prno-1]; i<MAX; i++)
printf("%d ", q.CQueue[prno-1][i]);
}
}
```

The output of the program is:
```
Main Menu
1. Insert
```

```
2. Delete
3. Exit
Enter your choice: 2
Underflow: Queue is empty!
Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value and its priority: 8 3
Front: 0, Rear: 0
Queue for prno 3 is: 8

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value and its priority: 9 4
Front: 0, Rear: 0
Queue for prno 4 is: 9

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 2
The priority of deleted item is: 3
The Deleted item is: 8

Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 3
```

Check Your Progress

14. What is the role played by a queue?

15. State the main characteristic of a queue data structure.

16. What does the Enqueue operation do?

17. State two common ways in which queues are implemented?

18. What do you mean by a priority queue?

3.5 LINKED LIST

A **linked list** is a linear collection of homogeneous elements called **nodes**. The successive nodes of a linked list need not occupy adjacent memory locations. The linear order between the nodes is maintained by the means of pointers. In linked lists, the insertion or deletion of nodes does not require shifting of the existing nodes as it is in the case of arrays; nodes can be inserted or deleted merely by adjusting the pointers or links.

Depending on the number of pointers in a node or the purpose for which the pointers are maintained, a linked list can be classified into various types such as singly linked list, circular linked list and doubly linked list.

3.5.1 Static and Dynamic Memory Allocation

Allocation of memory is a unique and special task performed by the compiler when any program is compiled and run. The specifications for static and dynamic type of allocation are defined in the program module. Both static and dynamic memory allocation are discussed as follows:

Static Memory Allocation

In static memory allocation, memory is allocated at compilation time. For example, if we initialize as int arr[] {1 2 3}, then the compiler will automatically allocate the required memory space for declared variables, in this case 3 integers, at the time of compilation. The address of operator is used to acquire the reserved address which may be assigned to a pointer variable. As most of the declared variables contain static memory address, hence assigning pointer value to a pointer variable is termed as static memory allocation. Static memory is pre-allocated at the time of mapping process into the main memory.

This technique's application includes a program module (e.g., function or subroutine), which declares a static data locally. This is done in a manner in which these data are inaccessible in other modules till it receives references as parameters or returns them. A single copy of static data is held back and is accessible through many calls to the function in which it is declared. Static memory allocation, therefore, has the advantage of modularizing data within a program design in the situation where these data must be retained through the runtime of the program.

The use of static variables within a class in object-oriented programming enables a single copy of such data to be shared between all the objects of that class.

Dynamic Memory Allocation

While in dynamic memory allocation, memory is assigned at run time, malloc( )compiler will merely view it as a function and an argument.

It makes use of functions such as malloc( ) or calloc( ) to get memory dynamically. In case these functions are made use of to get memory dynamically

and the values returned by these functions are assigned to pointer variables, then they are called dynamic memory allocation. Memory is assigned during run time.

Dynamic memory is allocated on the heap space of the process map. The pointer reference helps in the visibility throughout the process.

In computer science, dynamic memory allocation (also called heap-based memory allocation) is the allocation of memory storage that can be used in a computer program during the runtime of that program. It can also be viewed as a method of distributing ownership of limited memory resources among several pieces of data and code.

Dynamically allocated memory exists till it is release either explicitly by the programmer, or by the garbage collector. On the other hand, static memory allocation has a fixed duration. It is said that an object so allocated has a *dynamic lifetime*.

Implementations

- **Fixed-size-blocks allocation:** Fixed-size-blocks allocation, also known as memory pool allocation, makes use of a free list of fixed-size blocks of memory (often all of the same size). This is known to work well for simple embedded systems.

- **Buddy blocks:** In this system, memory is allocated from a large block in memory that is a power of two in size. If the block is more than twice of the desirable size, it is split into two. One of the halves is selected, and the process repeats (checking the size again and splitting if needed) until the block is just large enough.

  All the blocks of a particular size are kept in a sorted linked list or tree. When a block is freed, it is compared with its buddy. In case they are both free, they are combined and placed in the next largest size buddy-block list. (When a block is allocated, the allocator will start with the smallest sufficiently large block avoiding needlessly breaking blocks.)

3.5.2 Static and Dynamic Variables

A static variable upholds the same data all through the execution of a program while a dynamic variable can have different values during the course of a program.

Dynamic variables are allocated on the stack, so they are by default 'thread-local,' assuming a thread-safe implementation. By specifying them in a set statement in the thread's initial function, it makes them available (via use statements) to all the functions called in the same thread, as if they were global variables.

Theoretically, the set statement creates a set of dynamic variables and pushes them on the global stack of such sets. Reaching the end of the set statement pops the stack. The use statement searches the sets from the top of the stack down for each identifier listed in use statement and, if the identifier is found, stores the address of the dynamic variable in a local variable of the same name. Within the use statement, values of the dynamic variables are referenced indirectly. The set statement can be implemented with no allocation overhead, as all of the allocations can be done at compile time as local variables. The stack of sets is a list of structures defined by the following pseudo-C code, one for each dynamic variable.

```
struct dVariable {
struct dVariable *link;
const char *name;
Type *type;
void *address;
}
```

dVariable instances are connected via the link field. The name field points towards the name of the variable, the type field points towards the type descriptor sufficient for testing the subtype relation and the address field contains the address of the variable.

Dynamic variables are those that can have the space allocated to them at some point during the execution of a program or procedure and that can also be disposed of and have their space given back to the system by the program.

Dynamic and Static Variables in C

- Variable declarations can be done outside all functions or inside a function.
- Declarations made outside the functions are global and in fixed memory locations.
  - o The static declaration declares a variable outside a function to be a 'file global' (cannot be referenced by code in other source files).
- Declarations within a block statement {} (function body or block statement nested within a function body) have the following features:
  - o They are dynamically allocated, unless declared static.
  - o They are allocated in memory when program execution enters the block.
  - o They occur when memory is released and when the execution exits the block.
  - o They occur if a function calls itself (directly or indirectly) and it gets a new set of dynamic variables (called a stack frame).
  - o They are handled like any other call to the function.

Static Memory Use of Static Variable

When a program begins to execute, there must be some specific blocks of memory set aside for use that cannot be trespassed upon by any other program or even by the system for instance, the memory containing the program's own code. While it is possible (in machine language) to write a program that can modify its own code, it is very dangerous practice and must never be done.

Moreover, any variables named in the declaration section must have a specific memory set aside for their contents, and this action can not be controlled or changed in any way by the programmer, except by declaring more or fewer variables in the first place. The memory in question can not itself be relocated to some other place or expanded or contracted.

Static variables are those that are created in the declaration section of a program and continue to exist (whether visible or not) and require space until its

conclusion. Their space is allocated at the beginning of the program run.

The only change that can take place in static variable is their content and this change is done by assignment statements.

Items of all the data types, that have been taken into account till now, are of the static kind–once declared, they will be at a fixed location and consume a specific amount of memory during the running of the program. Both the size and location (relative to the start of the code) are predetermined at the time the program is compiled. Its location is relative and not absolute as there is no way for the compiler to determine ahead of time how many programs will be running and what memory will already be in use when the new program is loaded. However, with respect to program starting address, it is fixed and cannot be changed by the program.

Figure 3.25 illustrates a popular method of allocating memory. A block is a memory set aside for the program's use, and within this, the code is placed first (at the lowest address) and this is followed by the static variable space.
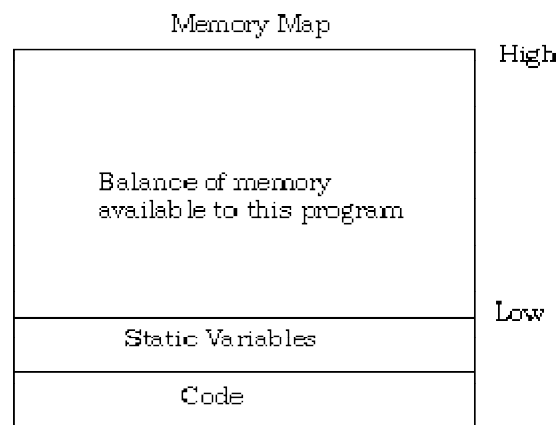
Fig. 3.25 *Allocation of Memory for Static Variable*

The area of memory into which the procedure activation records are dynamically and automatically placed is called stack and the marker that delimits the top end of the currently allocated stack is termed as the stack pointer.

3.5.3 Linked Lists: Pointers

The most commonly used linear data structure is a linked list. A linked list is a linear collection of similar data elements, called nodes, with each node containing some data and pointer(s) pointing to other node(s) in the list. Nodes of a linked list are not constrained to be at contiguous memory locations; instead they can be stored anywhere in the memory.

The linear order of the list is maintained by the **pointer field(s)** in each node. Depending on the **pointer field(s)** in each node, **linked lists** can be of different types. If each node of a linked list contains only **one pointer** and it points to the next node, then it is called a linear linked list or singly linked list. In such type of lists, the pointer field in the last node contains NULL. However, if the pointer in the last node is modified to point to the first node of the list, then it is called a circular linked list.

In addition to the pointer to the next node, each node of a linked list can also contain a pointer to its previous node. This type of a linked list is called doubly linked list. Figure 3.26 shows a singly, circular, and doubly linked list with five nodes each and pointers as follows:
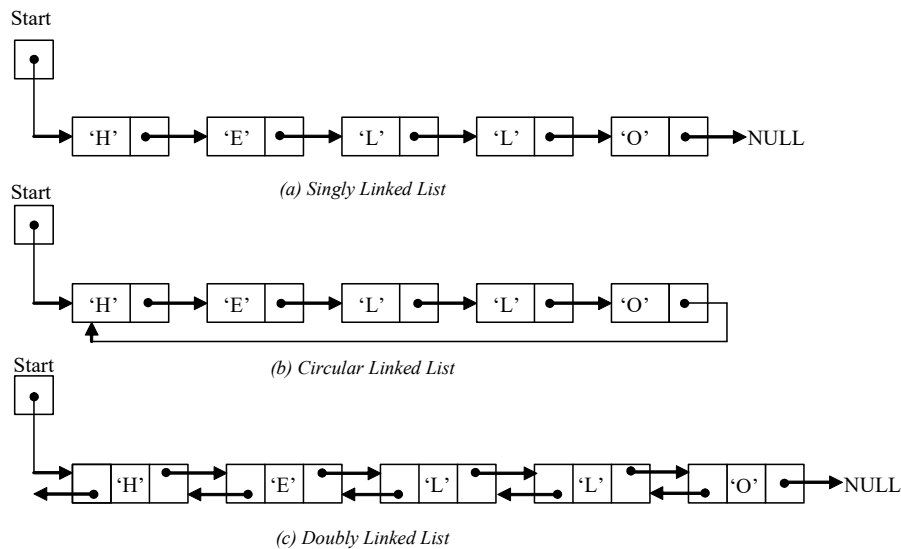
Start

'H' → 'E' → 'L' → 'L' → 'O' → NULL

(a) Singly Linked List

Start

'H' → 'E' → 'L' → 'L' → 'O'

(b) Circular Linked List

Start

'H' ⇄ 'E' ⇄ 'L' ⇄ 'L' ⇄ 'O' → NULL

(c) Doubly Linked List

Fig. 3.26 *Various Types of Linked Lists with Pointers*

Note: One major reason behind the popularity of linked lists is that it can expand or shrink during its life.

The linked lists allow the elements of the list to be stored non-contiguously. Thus, in order to access an element in the list, one has to start with the first node and follow the pointers in the nodes until the required element is found or till the end of list. In other words, linked lists allow only sequential access to their elements, meaning, in order to access the **n**th node of the list, the **n-1** preceding nodes need to be traversed.

3.5.4 Singly Linked Lists.

In a singly linked lists (also called **linear linked list**), each node consists of two fields: `info` and `next` (Figure 3.27). The `info` field contains the data and the `next` field contains the address of memory location where the next node is stored. The last node of the singly linked lists contains NULL in its `next` field that indicates the end of the list.

info   **next**

| data | ● | → pointer to next node
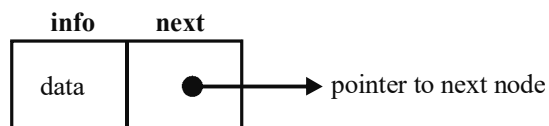
Fig. 3.27 *Node of a Linked List*

Note: The data stored in the info field may be a single data item of any data type or a complete record representing a student, or an employee, or any other entity. In this unit, however, it is assumed that the info field contains an integer data.

A linked list contains a list pointer variable `Start`, which stores the address of the first node of the list. In case, `Start` contains NULL, then the list is called an **empty list** or a **null list**. Since each node of the list contains only a single pointer pointing to the next node (not to the previous node), therefore, allows traversing only in one direction. It is also referred to as a **one-way list**. Figure 3.28 shows a singly linked lists with four nodes.
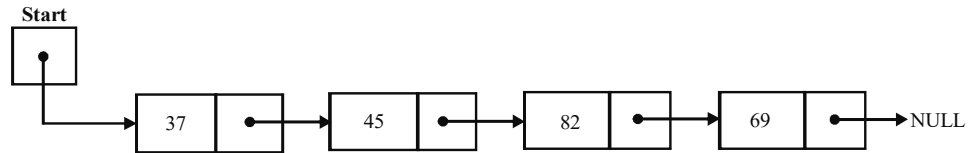
Fig. 3.28 *A Singly Linked Lists with Four Nodes*

3.5.5 Representation of Linked List

To maintain a linked list in the memory, two parallel arrays of equal size are used. One array (for example INFO) is used for the `info` field and another array (say, NEXT), for the `next` field of the nodes of the list. The values in the arrays are stored such that the ith locations in arrays INFO and NEXT, contain the `info` and `next` fields of a node of the list, respectively. In addition, a pointer variable `Start` is maintained in the memory that stores the starting address of the list. Figure 3.29 shows the memory representation of a linked list where each node contains an integer.
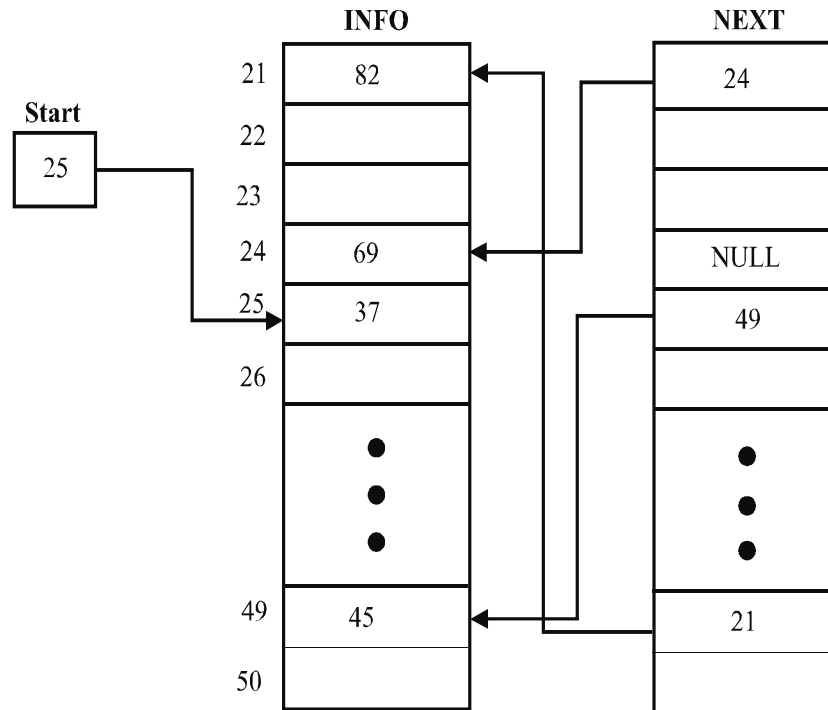
Fig. 3.29 *Memory Representation of a Linked List*

In Figure 3.29, the pointer variable `Start` contains 25, that is, the address of the first node of the list. This node stores the value 37 in the array INFO, and its corresponding element in the array NEXT stores 49. 49 the address of the

next node in the list. Similarly it is for other nodes. Finally, the node at address 24 stores value 69 in the array `INFO` and `NULL` in the array `NEXT`, thus, it is the last node of the list. Note that the values in the array `INFO` are stored randomly and the array `NEXT` is used to keep track of the values in the list.

Memory Allocation

As memory is allocated dynamically to a linked list, a new node can be inserted anytime in the list. For this, the memory manager maintains a special linked list known as **free storage list** or **memory bank** or **free pool** that consists of unused memory cells. This list keeps track of the free space available in the memory and a pointer to this list is stored in a pointer variable `Avail` (Figure 3.30). Note that the end of a free storage list is also denoted by storing `NULL` in the last available block of memory.
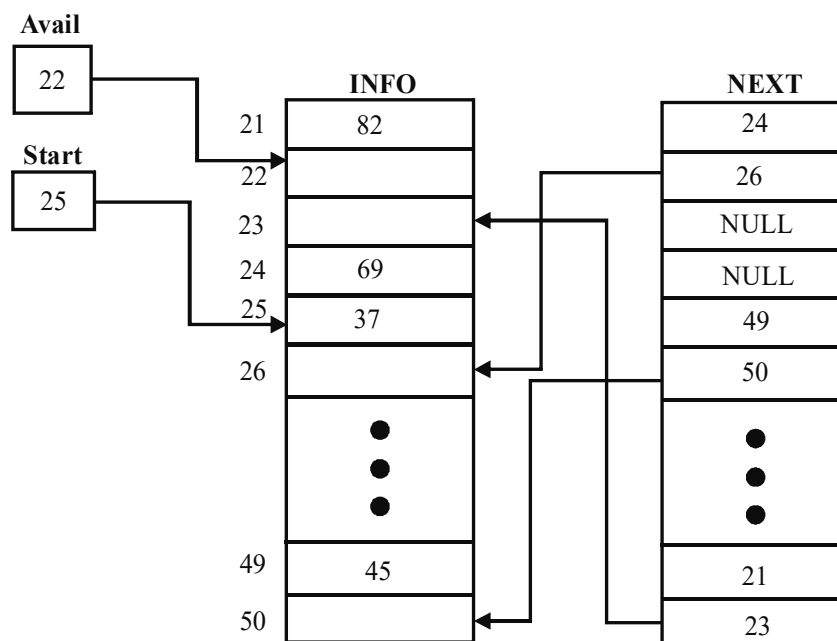
Fig. 3.30 *Free Storage List*

In Figure 3.30, `Avail` contains 22, hence, `INFO[22]` is the starting point of the free storage list. Since `NEXT[22]` contains 26, `INFO[26]` is the next free memory location. Similarly, other free spaces can also be accessed and `NULL` in `NEXT[23]` indicates the end of the free storage list.

While creating a linked list or inserting an element into a linked list, whenever a request for the new node arrives, the memory manager searches through the free storage list for the block of desired size. If the block of desired size is found, it returns a pointer to that block. However, sometimes there is no space available, that is, the free storage list is empty; this situation is termed as **overflow**. In this situation, the memory manager replies accordingly.

3.5.6 Implementation of Linked Lists

A number of operations can be performed on singly linked listss. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting and

merging linked lists. Before implementing these operations, you first need to understand how a node of a linked list is created.

Creating a node means defining its structure, allocating memory to it and its initialization. As discussed earlier, the node of a linked list consists of data and a pointer to the next node. To define a node containing an integer data and a pointer to the next node in 'C', you can use a self-referential structure whose definition is as follows:

```
typedef struct node
{
    int info;                /*to store integer type data*/
    struct node *next;       /*to store a pointer to next
node*/
}Node;
Node *nptr;           /*nptr is a pointer to node*/
```

After declaring a pointer `nptr` to the new node, memory needs to be dynamically allocated to it. If the memory is allocated successfully (that is, there is no overflow), the node is initialized. The `info` field is initialized with a valid value and the `next` field is initialized with `NULL`.

Algorithm 3.15 Creation of a Node

```
create_node()

1. Allocate memory for nptr        //nptr is a pointer to new node
2. If nptr = NULL
      Print "Overflow: Memory not allocated!" and go to step 7
   End If
3. Read item                       //item  is  the  value  to  be
   inserted in the new node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Return nptr                     //returning pointer nptr
7. End
```

Now, the linked list can be formed by creating several nodes of type `Node` and inserting them either in the beginning or at the end or at a specified position in the list.

1. Traversing

Traversing a list means accessing its elements one by one, in order to process all or some of the elements. For example, if you need to display the values of the nodes, count the number of nodes or search a particular item in the list, then traversing is required. You can traverse the list by using a temporary pointer variable (say, `temp`), which points to the node currently being processed. Initially, you make `temp` to point to the first node, process that element, then move `temp` to point to the next node using the statement `temp=temp->next`, process that element and move on as long as the last node is not reached, that is, until `temp` becomes `NULL`.

Algorithm 3.16 Traversing a List

```
display(Start)

1.  If Start = NULL                    //Start points to the first
    node of list
            Print "List is empty!!" and go to step 4
    End If
2.  Set temp = Start                    //initializing temp with Start

3.  While temp != NULL
        Print temp->info         //displaying value of each node
        Set temp = temp->next    //moving temp to point to next node
    End While
4.  End
```

Another example of traversing a linked list is counting the number of nodes in the linked list, which is as follows:

Algorithm 3.17 Counting the Number of Nodes

```
count_node(Start)

1.  Set count = 0
2.  Set temp = Start                    //initializing temp with Start
3.  While temp != NULL          //traversing the list
        Set count = count + 1          //incrementing count
        Set temp = temp->next
    End While
4.  Return count                //returning total number of nodes in
    the list
5.  End
```

2. Insertion

To insert a node in a linked list, a new node is created (as explained in Algorithm 3.15) and then placed at the desired position by adjusting the pointers. Nodes can be inserted either at the beginning or at the end or at any specified position in the list.

  (i) **Insertion in the beginning**: To insert a node at the beginning of the list, the next field of the new node (pointed to by nptr) is made to point to the existing first node and the Start pointer is modified to point to the new node (Figure 3.31).
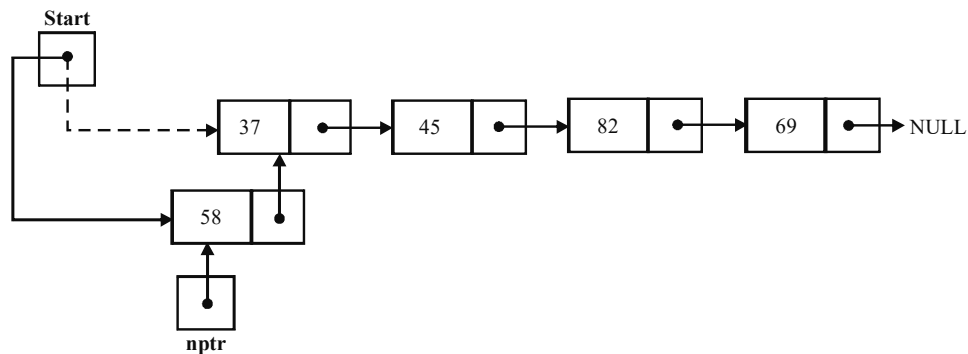
Fig. 3.31 *Insertion in the Beginning of a Linked List*

Algorithm 3.18 Insertion in the Beginning

```
insert_beg(Start)

1. Call create_node()          //creating a new node pointed to by
   nptr
2. Set nptr->next = Start
3. Set Start = nptr            //Start pointing to new node
4. End
```

(ii) **Insertion at the end:** To insert a node at the end of a linked list, the list is traversed up to the last node and the `next` field of this node is modified to point to the new node. However, if the linked list is initially empty, then the new node becomes the first node and `Start` points to it. Figure 3.32(a) shows a linked list with a pointer variable `temp` pointing to its first node and Figure 3.32(b) shows `temp` pointing to the last node and the `next` field of the last node pointing to the new node.

(a)

(b)
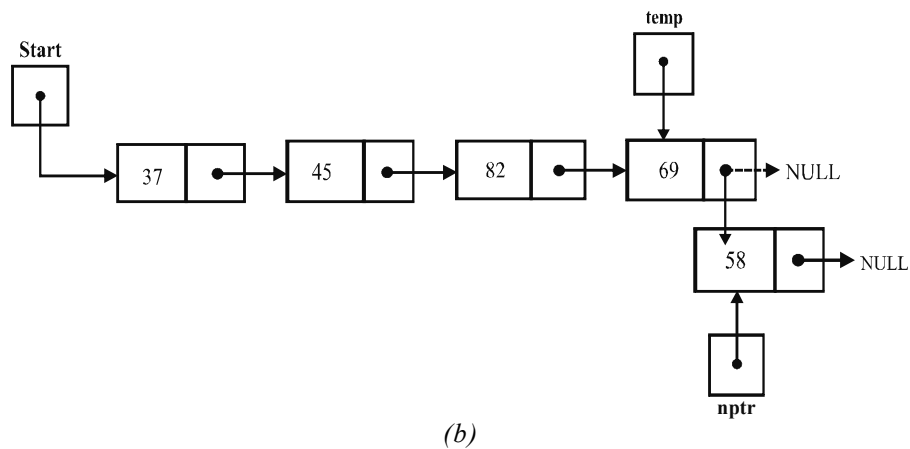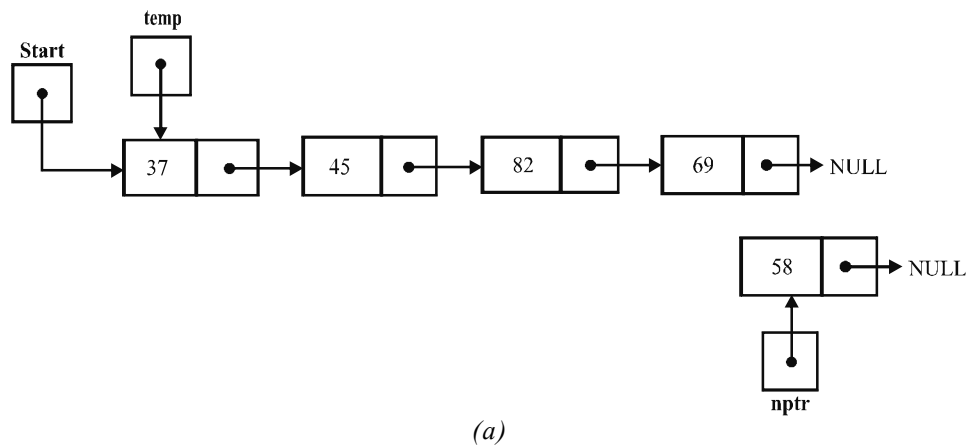
Fig. 3.32 *Insertion at the End of a Linked List*

Algorithm 3.19 Insertion at the End

```
insert_end(Start)

1. Call create_node()              //creating a new node pointed
   to by nptr
2. If Start = NULL                 //checking for empty list
      Set Start = nptr        //inserting new node as the first
node
   Else
      Set temp = Start
      While temp->next  != NULL     //traversing  up  to  the  last
node
            Set temp = temp->next
      End While
      Set temp->next = nptr        //appending  new  node  at  the
end
   End If
3. End
```

(iii) **Insertion at a specified position:** To insert a node at a position (say, `pos`) specified by the user, the list is traversed up to `pos-1` position. Then, the next field of the new node is made to point to the node that is already at `pos` position and the next field of the node at `pos-1` position is made to point to the new node. Figure 3.33 shows the insertion of the new node pointed to by the `nptr` at the third position.
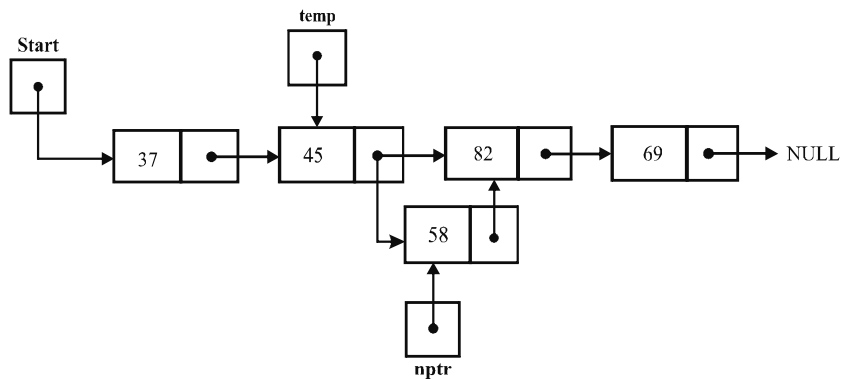
Fig. 3.33 *Insertion at a Specified Position in a Linked List*

Algorithm 3.20 Insertion at a Specified Position

```
insert_pos(Start)

1. Call create node()              //creating a new node pointed
   to by nptr
2. Set temp = Start
3. Read pos                        //position at which the new node is
   to be inserted
4. Call count node(temp)           //counting  total  number  of
   nodes in count variable
5. If (pos > count + 1 OR pos = 0)
       Print "Invalid position!" and go to step 7
   End If
6. If pos = 1
       Set nptr->next = Start
       Set Start = nptr            //inserting  new  node  as  the
first node
   Else
       Set i = 1
       While i < pos - 1           //traversing up to the node at
pos-1 position
             Set temp = temp->next
             Set i = i + 1
       End While
       Set nptr->next = temp->next       //inserting new node at
pos position
       Set temp->next = nptr
   End If
7. End
```

3. Deletion

Like insertion, nodes can be deleted from a linked list at any point of time and from any position. Whenever a node is deleted, the memory occupied by the node is de-allocated. Note that while performing deletions, you need to keep track of the node, that is, the immediate predecessor of the node to be deleted. Thus, while traversing the list, two temporary pointer variables are used (except in case of deletion from the beginning).

Note: The situation where the user tries to delete a node from an empty linked list, is termed as **Underflow**.

(i) **Deletion from the beginning:** To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable temp and Start is modified to point to the second node in the linked list. After that, the memory occupied by the node pointed to by temp is de-allocated. Figure 3.34 shows the deletion of a node from the beginning of a linked list.
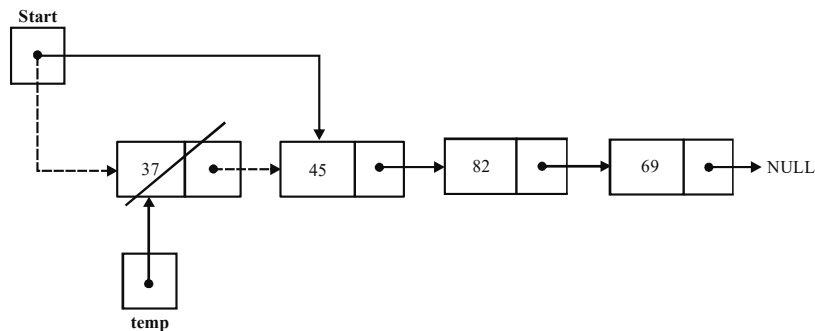
Fig. 3.34 Deletion from the Beginning of a Linked List

Algorithm 3.21 Deletion from the Beginning
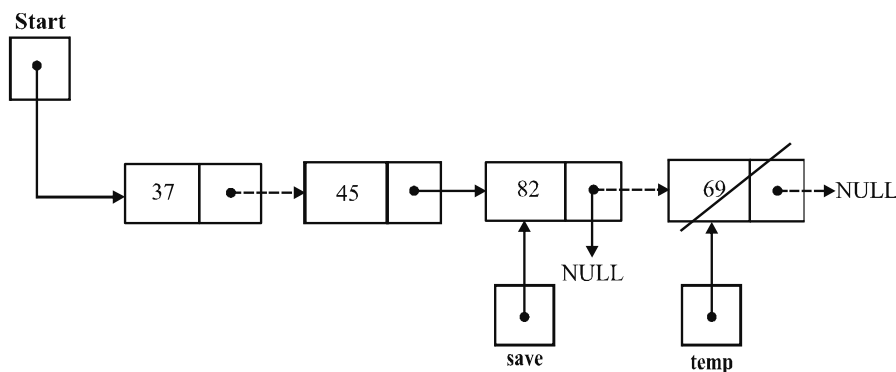
```
delete_beg(Start)

1. If Start = NULL                    //checking for underflow
      Print "Underflow: List is empty!" and go to step 5
   End If
2. Set temp = Start                   //temp pointing to the first
   node
3. Set Start = Start->next            //moving Start to point to the
   second node
4. De-allocate    temp                //de-allocating memory
5. End
```

(ii) **Deletion from the end**: To delete a node from the end of a linked list, the list is traversed up to the last node. Two pointer variables save and temp are used to traverse the list, where save points to the node previously pointed to by temp. At the end of traversing, temp points to the last node and save points to the second last node. Then, the next field of the node pointed to by save is made to point to NULL, and the memory occupied by the node pointed to by temp is de-allocated. Figure 3.35 shows the deletion of a node from the end of a linked list.
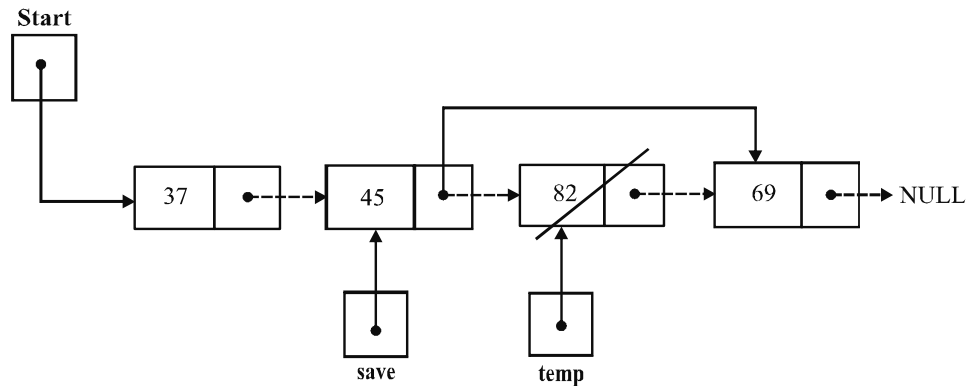

***Fig. 3.35** Deletion from the End of a Linked List*

Algorithm 3.22 Deletion from the End

```
delete_end(Start)

1. If Start = NULL              //checking for underflow
      Print "Underflow: List is empty!" and go to step 6
   End If
2. Set temp = Start            //temp pointing to the first node
3. If temp->next = NULL         //deleting the only node of the list
      Set Start = NULL
   Else
      While (temp->next) != NULL    //traversing  up  to  the  last
node
            Set save = temp   //save pointing to node previously
                       //pointed to by temp
      Set temp = temp->next   //moving temp to point to next node
      End While
   End If
4. Set save->next = NULL       //making new last node to point to
   NULL
5. De-allocate temp            //de-allocating memory
6. End
```

(iii) **Deletion from a specified position**: To delete a node from the position (say, pos) specified by the user, the list is traversed up to the pos position using pointer variables temp and save. At the end of traversing, temp points to the node at pos position and save points to the node at pos-1 position. Then, the next field of the node pointed to by save is made to point to the node at pos+1 position and, the memory occupied by the node pointed to by temp is de-allocated. Figure 3.36 shows the deletion of a node at the third position.

Fig. 3.36 *Deletion from a Specified Position in a Linked List*

Algorithm 3.23 Deletion from a Specified Position

```
delete_pos(Start)

1. If Start = NULL            //checking for underflow
      Print "Underflow: List is empty!" and go to step 8
   End If
2. Set temp = Start
3. Read pos              //position of the node to be deleted
4. Call count_node(Start)    //counting total number of nodes in
   count  variable
5. If pos > count OR pos = 0
      Print "Invalid position!" and go to step 8
   End If
6. If pos = 1
      Set Start = temp->next  //deleting the first node
   Else
      Set i = 1
      While i < pos           //traversing  up  to  the  node  at
position pos
         Set save = temp
         Set temp = temp->next
         Set i = i + 1
      End While
      Set save->next = temp->next   //deleting    the    node    at
position pos
   End If
7. De-allocate temp                    //de-allocating memory
8. End
```

Program 3.7: The following program illustrates the implementation of a singly linked list:

```
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
    int info;
    struct node *next;
}Node;
/* Function prototypes */
Node * create_node();
int isempty(Node *);
void display(Node *);
int count_node(Node *);
void insert_beg(Node **);
void insert_end(Node **);
void insert_pos(Node **);
void delete_beg(Node **);
void delete_end(Node **);
void delete_pos(Node **);
void main()
{
    int item,ch,ch1;
    Node *Start=NULL;
    do
    {
       clrscr();
       printf("\n\n\tMain Menu");
       printf("\n1. Insert");
       printf("\n2. Delete");
       printf("\n3. Display");
       printf("\n4. Exit\n");
       printf("\nEnter your choice: ");
       scanf("%d",&ch);
       switch(ch)
       {
          case 1: printf("\n1. Insert in the beginning");
             printf("\n2. Insert at the end");
                           printf("\n3.  Insert  at  a
specified position");
printf("\n4. Back to main menu\n");
printf("\nEnter your choice: ");
scanf("%d",&ch1);
```

```
switch(ch1)
{
case 1:   insert_beg(&Start);
    break;
case 2: insert_end(&Start);
break;
case 3: insert_pos(&Start);
break;
case 4: break;
default: printf("\nInvalid choice!");
}
break;
case 2: printf("\n1. Delete from the beginning");
        printf("\n2. Delete from the end");
        printf("\n3. Delete from a specified position");
            printf("\n4. Back to main menu\n");
            printf("\nEnter your choice: ");
            scanf("%d",&ch1);
    switch(ch1)
    {
        case 1: delete_beg(&Start);
            break;
        case 2: delete_end(&Start);
            break;
        case 3: delete_pos(&Start);
            break;
        case 4: break;
        default: printf("\nInvalid choice!");
        }
    break;
case 3: display(Start);
    break;
case 4: exit();
default: printf("\nInvalid choice!");
}
getch();
}while(1);
}
Node * create_node()
{
    Node *nptr;
    int item;
    nptr=(Node *)malloc(sizeof(Node));
    if(nptr==NULL)
    {
        printf("\nOverflow: Memory not allocated!");
```

```
        exit();
    }
    printf("\nEnter the value to be inserted: ");
    scanf("%d",&item);
    nptr->info=item;
    nptr->next=NULL;
    return(nptr);
}
int isempty(Node *Start)
{
    if(Start==NULL)
        return True;
    else
        return False;
}
void display(Node *Start)
{
    Node *temp=Start;
    if(isempty(temp))
    {
        printf("\nList is empty!!");
        return;
    }
    printf("\nThe linked list is: ");
    while(temp != NULL)
    {
        printf("%d ",temp->info);
        temp=temp->next;
    }
}
int count_node(Node *Start)
{
    Node *temp=Start;
    int count=0;
    while(temp != NULL)
    {
        count++;
    temp=temp->next;
    }
    return(count);
}
void insert_beg(Node **Start)
{
    Node *nptr=create_node();
```

```
                                   nptr->next=*Start;
                                   *Start=nptr;
                                   printf("\nNode inserted.");
```

NOTES

```
                               }
                               void insert_end(Node **Start)
                               {
                                   Node *temp=*Start;
                                   Node *nptr=create_node();
                                   if(isempty(temp))
                                       *Start=nptr;
                                   else
                                   {
                                       while(temp->next != NULL)
                                           temp=temp->next;
                                       temp->next=nptr;
                                   }
                                   printf("\nNode inserted.");
                               }


                               void insert_pos(Node **Start)
                               {
                                   int i,pos,count;
                                   Node *nptr=create_node();
                                   Node *temp=*Start;
                                   printf("\nEnter the position at which you want to
                               insert: ");
                                   scanf("%d",&pos);
                                   count=count_node(temp);
                                   if(pos>count+1 || pos==0)
                                   {
                                       printf("\nInvalid position!");
                                       return;
                                   }
                                   if(pos==1)
                                   {
                                       nptr->next=*Start;
                                       *Start=nptr;
                                   }
                                   else
                                   {
                               for(i=1;i<pos-1;i++)
                                   temp=temp->next;
                               nptr->next=temp->next;
                               temp->next=nptr;
```

```
}
printf("\nNode inserted.");
}
void delete_beg(Node **Start)
{
    Node *temp=*Start;
    if(isempty(temp))
    {
       printf("\nUnderflow: List is empty!");
       return;
    }
    *Start=temp->next;
    free(temp);
    printf("\nNode deleted.");
}
void delete_end(Node **Start)
{
    Node *temp=*Start;
    Node *save;
    if(isempty(temp))
    {
       printf("\nUnderflow: List is empty!");
       return;
    }
    if(temp->next==NULL)
       *Start=NULL;
    else
    {
       while(temp->next != NULL)
       {
           save=temp;
           temp=temp->next;
       }
       save->next=NULL;
    }
    free(temp);
    printf("\nNode deleted.");
}
void delete_pos(Node **Start)
{
     Node *temp=*Start,*save;
     int i,pos,count;
     if(isempty(temp))
     {
```

```
        printf("\nUnderflow: List is empty!");
        return;
    }
    printf("\nEnter the position of the node to be deleted: ");
    scanf("%d",&pos);
    count=count_node(temp);
    if(pos>count || pos==0)
    {
      printf("\nInvalid position!");
      return;
    }
    if(pos==1)
      *Start=temp->next;
    else
    {
      for(i=1;i<pos;i++)
      {
          save=temp;
          temp=temp->next;
      }
      save->next=temp->next;
    }
    free(temp);
    printf("\nNode deleted.");
}
```

The output of the Program is as follows:

```
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu
Enter your choice: 1
Enter the value to be inserted: 1
Node inserted.
Main Menu
1. Insert
2. Delete
3. Display
```

```
4. Exit
Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu
Enter your choice: 2
Enter the value to be inserted: 3
Node inserted
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu
Enter your choice: 3
Enter the value to be inserted: 2
Enter the position at which you want to insert: 2
Node inserted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
The linked list is: 1 2 3
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu
Enter your choice: 1
Node deleted.
Main Menu
1. Insert
```

```
2. Delete
3. Display
4. Exit
Enter your choice: 3
The linked list is: 2 3
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu
Enter your choice: 3
Enter the position of the node to be deleted: 2
Node deleted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu
Enter your choice: 2
Node deleted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
List is empty!!
Main Menu
1. Insert
```

```
2. Delete
3. Display
4. Exit
Enter your choice: 4
```

4. Searching

Searching a value (say, `item`) in a linked list means finding the position of the node, which stores `item` as its value. If `item` is found in the list, the search is successful and the position of that node is displayed. However, if `item` is not found till the end of the list, then the search is unsuccessful and an appropriate message is displayed. Note that the linked list may be in a sorted or an unsorted order. Therefore, here, two searching algorithms will be discussed: one for the sorted and another for the unsorted linked list.

Note: Only linear search can be performed on linked lists.

   (i) **Searching in an unsorted list:** If data in a linked list is not arranged in a specific order, the list is traversed completely starting from the first node towards the last node and the value of each node (that is, `node->info`) is compared with the value to be searched.

Algorithm 3.24 Searching in an Unsorted List

```
search_unsort(Start)

1. If Start = NULL
       Print "List is empty!!" and go to step 7
   End If
2. Set ptr = Start            //ptr pointing to the first node
3. Set pos = 1
4. Read item                  //item is the value to be searched
5. While ptr != NULL          //traversing up to the last node
       If item = ptr->info
             Print "Value found at position", pos and go to step 7
       Else
             Set ptr = ptr->next//moving ptr to point to next node
             Set pos = pos + 1
       End If
   End While
6. Print "Value not found"     //search unsuccessful
7. End
```

  (ii) **Searching in a sorted list:** The process of searching an item in a sorted (ascending order) linked list is similar to that of an unsorted linked list. However, while comparing, once the value of any node exceeds `item` (the value to be searched), the search is stopped immediately. In other word, the list is not required to be traversed completely.

Algorithm 3.25 Searching in a Sorted List

```
search_sort(Start)

1. If Start = NULL
      Print "List is empty!!" and go to step 7
   End If
2. Set ptr = Start                //ptr pointing to the first
   node
3. Set pos = 1
4. Read item
5. While ptr->next != NULL         //traversing up to the last
   node
      If item < ptr->info          //comparing item with the
value of current node
            Print "Value not found" and go to step 7
      Else If item = ptr->info
            Print "Value found at position", pos and go to step 7
      Else
            Set ptr = ptr->next    //moving ptr to point to next
node
            Set pos = pos + 1
      End If
   End While
6. Print "Value not found"          //search unsuccessful
7. End
```

3.5.7 Reversing of Linked List

To reverse a singly linked list, the list is traversed up to the last node and the links of the nodes are reversed such that the first node of the list becomes the last node and the last node becomes the first node. For this, three pointer variables (say, save, ptr and temp) are used. Initially, temp points to Start, and both ptr and save point to NULL. While traversing the list, temp points to the current node, ptr points to the node previously pointed to by temp and save points to the node previously pointed to by ptr. The links between the nodes are reversed by making the next field of the node pointed to by ptr to point to the node pointed to by save. At the end of traversing, temp points to NULL, ptr points to the last node and save points to the second last node of the list. Then Start is made to point to the node pointed to by ptr in order to make the last node as the first node of the list. Figure 3.37 (a-f) show the process of reversing a linked list.

(a)

(b)

(c)

(d)

(e)

(f)
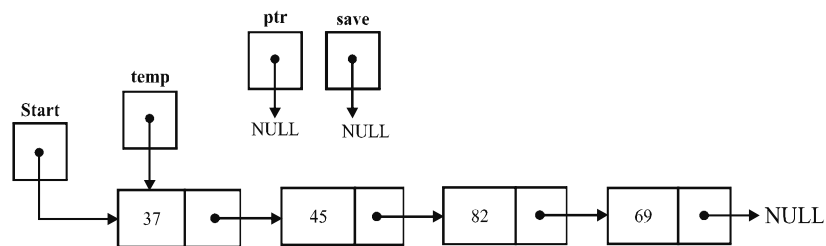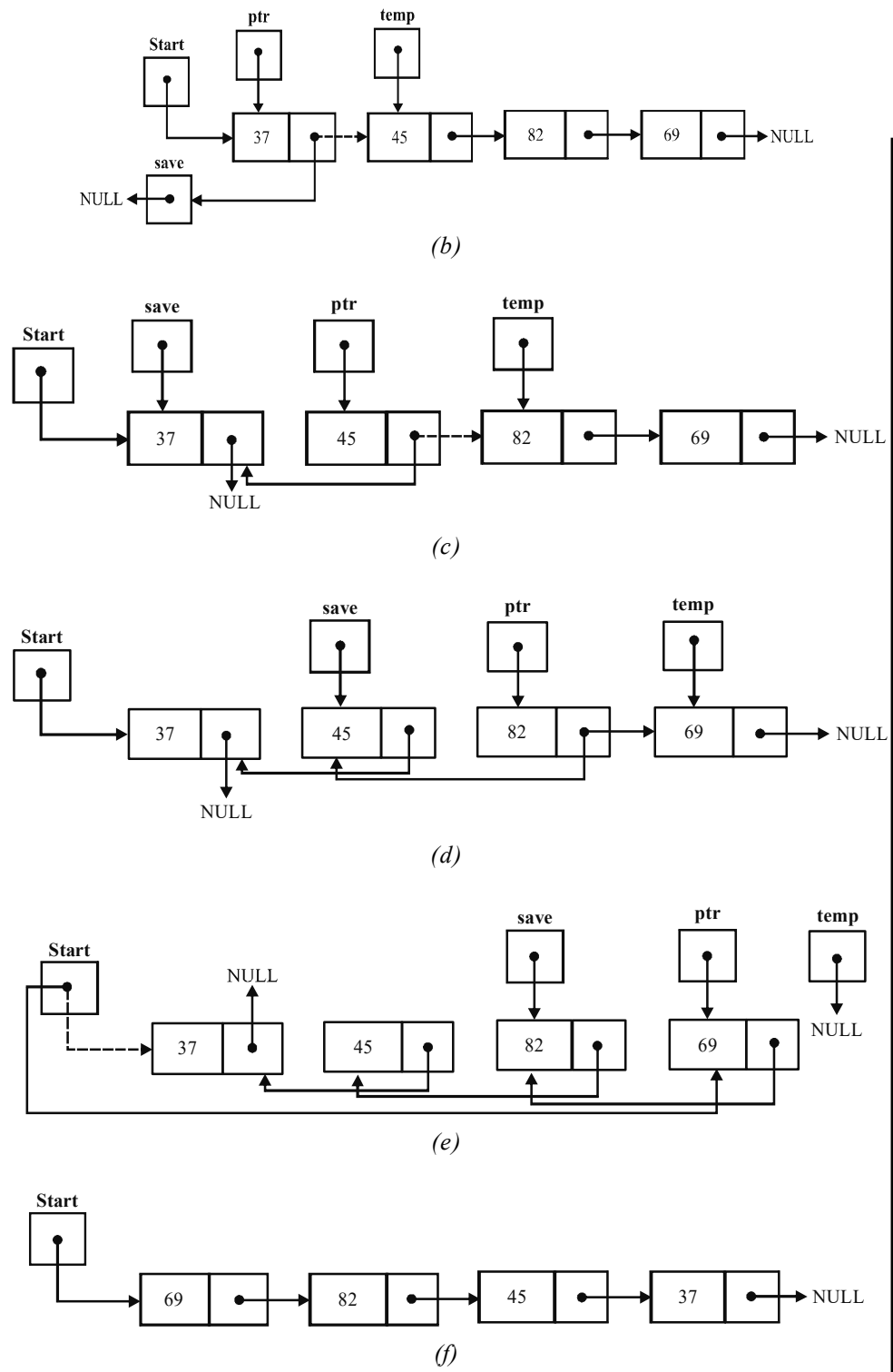
Fig. 3.37 *Reversing a Linked List*

Algorithm 3.26 Reversing a Singly-linked List

```
reverse(Start)

1. Set temp = Start
2. Set ptr = NULL
3. Set save = NULL
4. While temp != NULL                  //traversing up to the last
   node
       Set save = ptr
       Set ptr = temp
       Set temp = temp->next
       Set ptr->next = save
   End While
5. Set Start = ptr
6. End
```

3.5.8 Concatenation of Linked List

In computer programming, string concatenation is the process of joining two character strings from one end to the other. For example, the strings, 'snow' and 'ball' can be concatenated to give 'snowball'. In several programming languages, string concatenation is a binary infix operator.

As an example, the following expression uses the '+' symbol as the concatenation operator:

```
print "Hello " + "World";
```

The concatenation operation on strings can be generalized to an operation on a set of strings as follows:

For two sets of strings S_1 and S_2, the *concatenation S_1S_2* consists of all strings of the form *vw* where *v* is a string from S_1 and *w* is a string from S_2.

In this definition, the string *vw* is the ordinary concatenation of strings *v* and *w* as defined in the introductory section. In this context, sets of strings are often referred to as formal languages. Note that you do not use an explicit operator symbol to represent the concatenation.

Program on linked list concatenate operations

```
# include<stdio.h>
# include<conio.h>
# include "malloc.h"
struct node
{
int data;
struct node *link;
};

void main()
{
int a=111,b=2,c=3,will,wish,num;
struct node *ptr,*ptr2,*result,*temp;
void add(struct node **,int );
```

```
struct node * search(struct node *);
void display(struct node *);
void invert(struct node *);
void del(struct node *,int);
struct node * concat(struct node *,struct node *);
ptr=NULL;
ptr2=NULL;
result=NULL;   //result  for  storing  the  result  of
concatenation
clrscr();
will=1;


while(will==1)
{
printf("


      Main Menu
1. Add element
2.Delete element
3.Search element
4Linked List concatenation
5.Invert linked list
6. Display elements
      Please enter the choice");
scanf("%d",&wish);
switch(wish)
{
case 1:
    printf("
Enter the element you want to add  ");
    scanf("%d",&num);
    add(&ptr,num);
    display(ptr);
    break;
case 2:
    printf("
Enter the element to delete ");
    scanf("%d",&num);
    del(ptr,num);
    break;
case 3:
    printf("
 Now demonstrating search ");
```

```
                              temp = search(ptr);
                                  printf("
Address of first occurence is  %u ",temp);
                                  break;
case 4:
                                  /* Inputs given internally for demo only */
                                  printf(" Now demonstrating linked list concatenation
 Press any key to continue...");
                                  add(&ptr2,2);
                                  add(&ptr2,4);
                                  add(&ptr2,6);
                                  getch();
                                  printf("
 Displaying second Linked List
");
                                  display(ptr2);
                                  getch();
                                  result = concat(ptr,ptr2);
                                  clrscr();
                                  printf("


Now Displaying the result of concatenation");
                                  display(result);
                                  getch();
                                  break;
case 5:

                                  printf("
 Inverting the list ...
Press any key to continue...");
                                  invert(ptr);
                                  break;
case 6:
                                  display(ptr);
                                  break;
default:
                                  printf("


 Illegal choice

");
}
printf("
 DO you want to continue ( press 1 for yes ");
```

```
scanf("%d",&will);
}   //end of while
}
void add(struct node **q,int num)
{
struct node *temp;
temp = *q;
if(*q==NULL)
{
    *q=malloc(sizeof(struct node));
    temp = *q;
}
else
{
    while((temp->link)!=NULL)
    {
        temp=temp->link;
    }
    temp->link = malloc(sizeof(struct node));
    temp=temp->link;
}
temp->data = num;
temp->link  = NULL;
}

void display(struct node *pt)
{

while(pt!=NULL)
{

    printf("

 Data : %d",pt->data);
    printf("
 Link : %d",pt->link);
    pt=pt->link;
}
}
void invert(struct node *ptr)
{
struct node  *p,*q,*r;
p=ptr;
q=NULL;
```

```
while(p!=NULL)
{
    r=q;
    q=p;
    p=p->link;
    q->link=r;
}
ptr = q;
display(ptr);
}
//  CONCATENATION OF LINKED LISTS

struct node * concat(struct node *p,struct node *q)
{
struct node *x,*r;
if (p==NULL)
r=q;

if (q==NULL)
r=p;
else
{
    x=p;
    r=x;
    while(x->link!=NULL)
x=x->link;
    x->link=q;
}
    return(r);
}
```

3.5.9 Merging Linked List Using Merge Sort

Merge sort is an $O(n \log n)$ comparison-based algorithm which sorts comparison-based sorting algorithm. In maximum implementations it remains steady, which means that it maintains the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm. The credit for its invention goes to John von Neumann, who invented it in 1945.

Conceptually, a merge sort works as follows:

1. In case the list is of length 0 or 1, then it is already sorted. Otherwise, the following steps should be followed.

2. Categorize the unsorted list into two sublists of approximately half the size.

3. Sort each sublist recursively by re-applying merge sort.

4. Merge the two sublists back into one sorted list.

Two major ideas are incorporated by merge sort in order to improvize its runtime. These ideas are as follows:

1. A small list will take fewer steps to sort than a large list.

2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once, in case they are already sorted. An example of this kind of implementation is as follows:

Example 3.1: Using merge sort to sort a list of integers contained in an array:

Suppose we have an array A with n indices ranging from A_0 to A_{n-1}. We apply merge sort to $A(A_0..A_{c-1})$ and $A(A_c..A_{n-1})$ where c is the integer part of $n / 2$. Once the two halves are returned they would have been sorted. It is now possible to merge them together to form a sorted array.

In a simple pseudo code form, the algorithm could look something like this:

```
function merge_sort(m)
    if length(m) d" 1
        return m
    var list left, right, result

    var integer middle = length(m) / 2
    for each x in m up to middle
        add x to left
    for each x in m after middle
        add x to right
    left = merge_sort(left)
    right = merge_sort(right)
    if left.last_item > right.first_item
        result = merge(left, right)
    else
        result = append(left, right)
    return result
```

merge_sort function is required to merge both the left and right lists created above. There are several variants for the merge() function, the simplest variant could be as follows:

```
function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) d" first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
```

```
end while
if length(left) > 0
    append left to result
else
    append right to result
return result
```

The following is the program for merging two linked lists:

```
LIST Merge_List(LIST P,LIST Q)
{
   HEADER h = P;
    NODEPTR ptr2 =Q;
     NODEPTR ptr1 = P;
     NODEPTR prev=P;
     NODEPTR temp=NULL;
     /* Second List Pointer */
     while(ptr2 != NULL)
      {
           /* First List Pointer */
           while(ptr1 != NULL)
            {
                while(ptr2->Age>ptr1->Age)
                 {
                     prev = ptr1;
                     ptr1 = ptr1->Next;
                    }
                  if(prev == ptr1)
                   {
                        // This means the first node of
   the header needs to be changed
                        temp = New_Node();
                        temp->Age = ptr2->Age;
                        temp->Next = prev;
                        prev = temp;
                        h = temp;
                    }
                    else
                    {
                        temp = New_Node();
                        temp->Age = ptr2->Age;

                        temp->Next = prev->Next;
                        prev->Next = temp;
                    }
                    break;
               }
             ptr2 = ptr2->Next;
          }
        return h;
```

3.5.10 Applications of Linked List

Given below are the applications of linked list:

1. Linked list is used to implement stacks and queues which are like fundamental needs throughout computer science.

2. A singly linked list is used to prevent the collision between the data in the hash map.

3. While working in the notepad, undo, redo and deleting functions are performed by the linked list.

4. Linked list can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.

5. Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap.

6. It is also used by the operating system to share time for different users. Generally a linked list uses a Round-Robin time-sharing mechanism. This algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking.

7. Multiplayer games use a circular list to swap between players in a loop.

8. Doubly linked list is used in navigation systems, as it needs front and back navigation.

9. In the browser when we want to use the back or next function to change the tab, it is done through a doubly-linked list.

10. Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.

11. It is used in music playing system where one can easily play the previous or next one song as many times one wants to.

12. In many operating systems, the thread scheduler (which chooses what processes need to run at what times) maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue (say, the list of active processes that need a turn to run) into another queue (say, the list of processes that are blocked and waiting for something to release them).

Check Your Progress

19. What is a linked list?

20. What are the fields of the node of a singly linked lists?

21. What is a dynamic data structure?

22. How can a linked list be created to contain roll numbers and names of students having structure definition in C language?

23. What do you understand by the term underflow?

24. How is searching in a sorted linked list different from searching in an unsorted linked list?

25. How can a linked list be reversed?

3.6 CIRCULAR LINKED LIST

A linear linked list in which the next field of the last node points back to the first node instead of containing NULL is termed a circular linked list. The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, you can reach any of its predecessor nodes. This is because when you traverse a circular linked list starting with a particular node, you come back to the same node at the end. Figure 3.38 shows an example of a circular linked list.
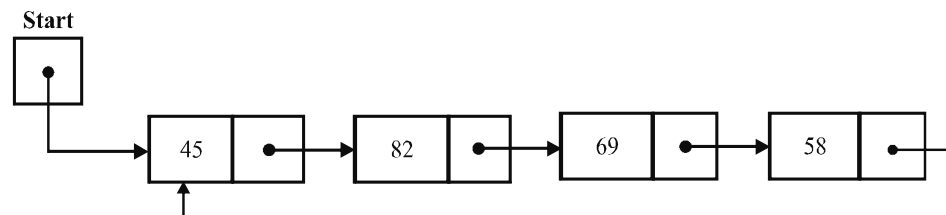
Fig. 3.38 Example of a Circular Linked List

All the operations that can be performed on linear linked lists can be easily performed on circular linked lists, but with some modifications.

Note: The process of creating a node of a circular linked list is the same as that of a linear linked list.

Traversing

You can traverse a circular linked list in the same way as a linear linked list except the condition for checking the end of the list. Here, the list is traversed until you reach a node in the list that contains the address of the first node in its next field rather than NULL, as in the case of a linear linked list.

Algorithm 3.27 Traversing a Circular Linked List

```
display(Start)

1. If Start = NULL
          Print "List is empty!!" and go to step 4
   End If
2. Set temp = Start                //initializing temp with Start
3. Do
      Print temp->info        //displaying value of each node
      Set temp = temp->next
   While temp != Start
4. End
```

Insertion

Like linear linked lists, nodes can be inserted at any position in a circular linked list. To insert a new node (pointed to by nptr) at the beginning of a circular linked list (Figure 3.39(a)), the next field of the new node is made to point to the existing first node and the Start pointer is modified to point to the new node. Now, since the first node of the list is changed, the next field of the last node also needs to be modified to point to the new node. However, if the list is initially empty, a new node is inserted as the first node and its next field is made to point to itself (Figure 3.39(b)).

(a)

(b)
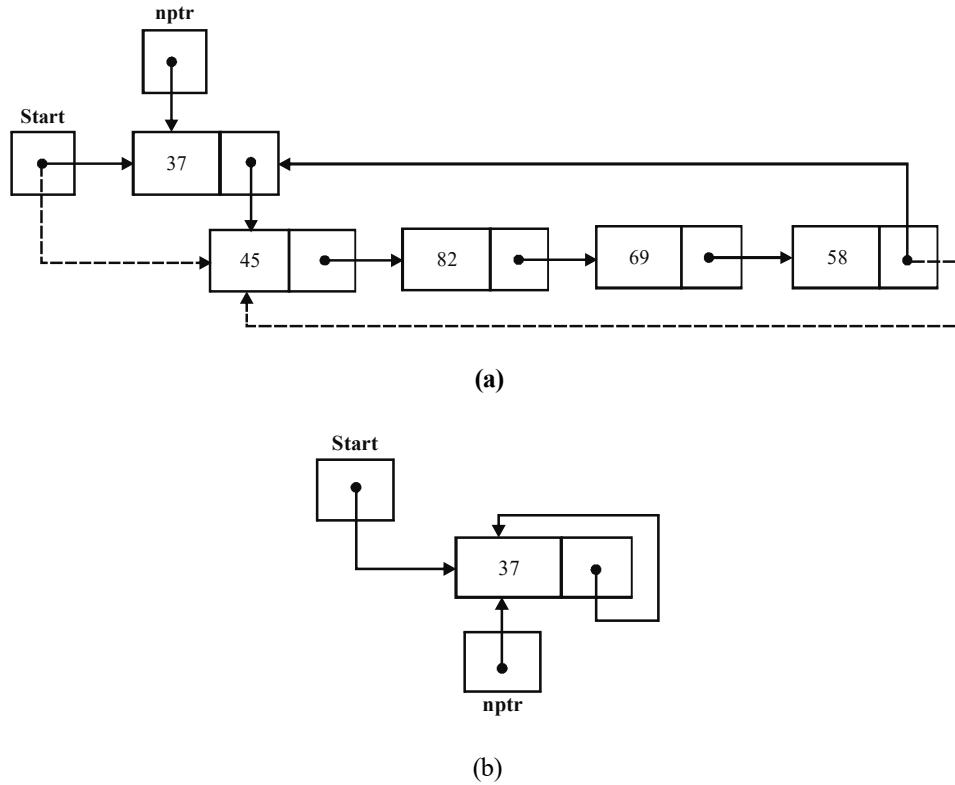
Fig. 3.39 *Insertion in the Beginning*

Algorithm 3.28 Insertion in the Beginning

```
insert_beg(Start)

1. Call create_node()              //creating a new node pointed
   to by nptr
2. If Start = NULL                 //checking for empty list
          Set Start = nptr                 //inserting new node as
the first node
     Set Start->next = Start
     Else
     Set temp = Start
     While temp->next != Start     //traversing  up  to  the  last
node
          Set temp = temp->next
     End While
     Set nptr->next = Start        //inserting  new  node  in  the
beginning
     Set Start = nptr       //Start pointing to new node
     Set temp->next = Start  //next field of last node pointing
to new node
     End If
3. End
```

While inserting a new node (pointed to by nptr) at the end of a circular linked list, the list is traversed up to the last node. The next field of the last node is made to point to the new node and the next field of the new node is made to point to Start. However, if the circular linked list is empty, the new node becomes the first node and Start points to it. In addition, the next field of the new

node points to itself, as it is the single node in the list. Figure 3.40 shows the insertion of a new node pointed to by `nptr` at the end of a circular linked list.
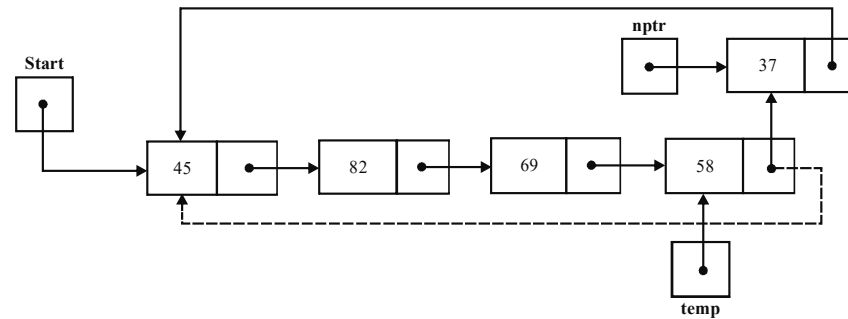
***Fig. 3.40** Insertion at the End*

Algorithm 3.29 Insertion at the End

```
insert_end(Start)

1. Call create_node()                  //creating a new node pointed
   to by nptr
2. If Start = NULL                     //checking for empty list
           Set Start = nptr                    //inserting new node in
the empty linked list
     Set Start->next = Start //next field of first node pointing
to itself
     Else
     Set temp = Start
     While temp->next != Start      //traversing  up  to  the  last
node
           Set temp = temp->next
     End While
     Set temp->next = nptr          //next   field   of   last   node
pointing to new node
     Set nptr->next = Start         //next   field   of   new   node
pointing to Start
   End If
3. End
```

Deletion

To delete a node from the beginning of a circular linked list, `Start` is modified to point to the second node and the `next` field of the last node is made to point to the new first node. For this, two pointer variables `temp` and `ptr` are used. The pointer `temp` stores the address of the node to be deleted (that is, the address of the first node) and `Start` is modified to point to the second node. The pointer `ptr` is used for traversing the list and at the end of traversing, it stores the address of the last node. Then the `next` field of the last node is made to point to the new first node. Also, the memory occupied by the node pointed to by `temp` is de-allocated. Figure 3.41 shows the deletion of a node from the beginning of a circular linked list.
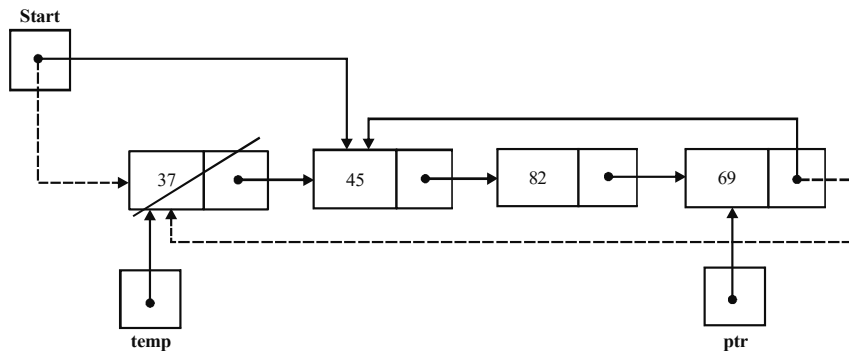
Fig. 3.41 Deletion from the Beginning

Algorithm 3.30 Deletion from the Beginning

```
delete_beg(Start)

1. If Start = NULL
     Print "Underflow: List is empty!" and go to step 8
   End If
2. Set temp = Start
3. Set ptr = temp
4. While ptr->next != Start          //traversing  up  to  the  last
   node
     Set ptr = ptr->next
   End While
5. Set Start = Start->next            //Start  pointing  to  the  next
   node
6. Set ptr->next = Start              //last  node  pointing  to  new
   first node
7. De-allocate temp                   //de-allocating memory
8. End
```

To delete a node from the end of a circular linked list, two pointer variables save and temp are used. The pointer variable temp is used to traverse the list and save points to the node previously pointed to by temp. At the end of traversing, temp points to the last node and save points to the second last node. Then the next field of save is made to point to Start, and the memory occupied by the last node (that is, pointed to by temp) is de-allocated. Figure 3.42 shows the deletion of a node from the end of a circular linked list.
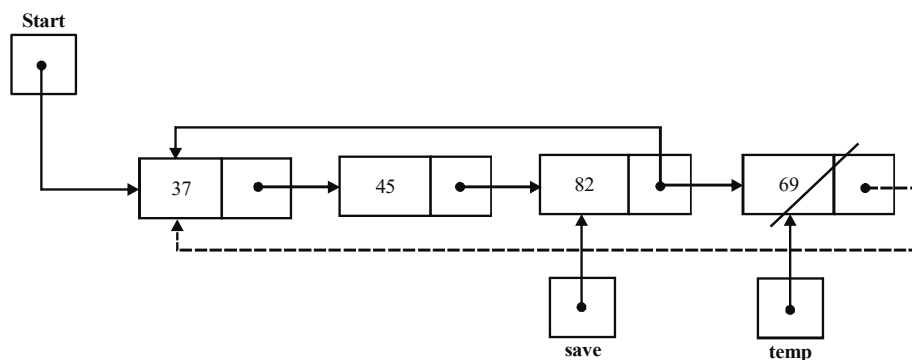
Fig.3.42 Deletion from the End

Algorithm 3.31 Deletion from the End

```
delete_end(Start)

1. If Start = NULL                    //checking for underflow
      Print "Underflow: List is empty!" and go to step 5
   End If
2. Set temp = Start
3. If temp->next = Start              //deleting the only node
   of the list
      Set Start = NULL
   Else
      While temp->next != Start       //traversing  up  to  the
last node
           Set save = temp
           Set temp = temp->next
      End While
      Set save->next = Start       //second last node becomes the
last node
   End If
4. De-allocate temp                   //de-allocating memory
5. End
```

Note: The process of deleting a node from a specified position in a circular linked list is the same as that of a singly linked list.

Program 3.8: The following program illustrates the implementation of a circular linked list:

```
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
    int info;
    struct node *next;
}Node;
/* Function prototypes */
Node * create_node();
int isempty(Node *);
void display(Node *);
void insert_beg(Node **);
void insert_end(Node **);
void delete_beg(Node **);
void delete_end(Node **);
void main()
{
    int item,ch,ch1;
    Node *Start=NULL;
    do
    {
        clrscr();
        printf("\n\n\tMain Menu");
```

```
        printf("\n1. Insert");
        printf("\n2. Delete");
        printf("\n3. Display");
        printf("\n4. Exit\n");
        printf("\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\n1. Insert in the beginning");
                            printf("\n2. Insert  at  the
end");
            printf("\n3. Back to main menu\n");
                    printf("\nEnter your choice: ");
                            scanf("%d",&ch1);
            switch(ch1)
            {
                case 1:     insert_beg(&Start);

                    break;
                case 2: insert_end(&Start);
                            break;
                case 3: break;
                default: printf("\nInvalid choice!");
                    }
            break;
            case  2:  printf("\n1.  Delete  from  the
beginning");
                printf("\n2. Delete from the end");
                printf("\n3. Back to main menu\n");
                printf("\nEnter your choice: ");
                scanf("%d",&ch1);
                switch(ch1)
                {
                        case 1: delete_beg(&Start);
                            break;
                        case 2: delete_end(&Start);
                                    break;
                        case 3: break;
                            default:  printf("\nInvalid
choice!");
                }
                break;
            case 3: display(Start);
                break;
            case 4: exit();
            default: printf("\nInvalid choice!");
```

```
            }
            getch();
        }while(1);
}
Node * create_node()
{
    Node *nptr;
    int item;
    nptr=(Node *)malloc(sizeof(Node));
    if(nptr==NULL)
    {
        printf("\nOverflow: Memory not allocated!");
        exit();
    }
    printf("\nEnter the value to be inserted: ");
    scanf("%d",&item);
    nptr->info=item;
    nptr->next=NULL;
    return(nptr);
}
int isempty(Node *Start)
{
    if(Start==NULL)
        return True;
    else
        return False;
}
void display(Node *Start)
{
    Node *temp=Start;
    if(isempty(temp))
        printf("\nList is empty!!");
    else
    {
        printf("\nThe linked list is: ");
        do
        {
            printf("%d ",temp->info);
            temp=temp->next;
        }while(temp != Start);
    }
}
void insert_beg(Node **Start)
{
```

```
        Node *nptr=create_node();
        Node *temp=*Start;
        if(isempty(temp))
        {
           *Start=nptr;
           (*Start)->next=*Start;
        }
        else
        {
           while(temp->next != *Start)
               temp=temp->next;
           nptr->next=*Start;
           *Start=nptr;
           temp->next=*Start;
        }
        printf("\nNode inserted.");
   }
   void insert_end(Node **Start)
   {
        Node *temp=*Start;
        Node *nptr=create_node();
        if(isempty(temp))
        {
           *Start=nptr;
           (*Start)->next=*Start;
        }
        else
        {
           while(temp->next != *Start)
               temp=temp->next;
           temp->next=nptr;
           nptr->next=*Start;
        }
        printf("\nNode inserted.");
   }
   void delete_beg(Node **Start)
   {
        Node *temp=*Start;
        Node *ptr=temp;
        if(isempty(temp))
        {
           printf("\nUnderflow: List is empty!");
           return;
        }
```

```
        while(ptr->next != *Start)
            ptr=ptr->next;
        *Start=(*Start)->next;
        ptr->next=*Start;
        free(temp);
        printf("\nNode deleted.");
    }
    void delete_end(Node **Start)
    {
        Node *temp=*Start;
        Node *save;
        if(isempty(temp))
        {
            printf("\nUnderflow: List is empty!");
            return;
        }
        if(temp->next==*Start)
            *Start=NULL;
        else
        {
            while(temp->next != *Start)
            {
                save=temp;
                temp=temp->next;
            }
            save->next=*Start;
        }
        free(temp);
        printf("\nNode deleted.");
    }
```

The output of Program is as follows:

```
Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Back to main menu
Enter your choice: 1
Enter the value to be inserted: 5
Node inserted.
```

```
Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Back to main menu
Enter your choice: 1
Enter the value to be inserted: 4
Node inserted.

Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Back to main menu
Enter your choice: 2
Enter the value to be inserted: 6
Node inserted.

Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
The linked list is: 4 5 6

Main Menu
1. Insert
2. Delete
3. Display
4. Exit
```

```
Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Back to main menu
Enter your choice: 1
Node deleted.

Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3
The linked list is: 5 6

Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Back to main menu
Enter your choice: 2
Node deleted.

Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
The linked list is: 5

Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4
```

3.7 DOUBLY LINKED LIST

In a singly linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, it can traverse only in one direction, that is, from beginning to end. However, sometimes it is required to traverse in the backward direction, that is, from the end to the beginning. This can be implemented by maintaining an additional pointer in each node of the list that points to the previous node. Such a linked list is called a doubly linked list.

Each node of a doubly linked list consists of three fields: `prev`, `info` and `next` (Figure 3.43). The `info` field contains the data, the `prev` field contains the address of the previous node and the `next` field contains the address of the next node.
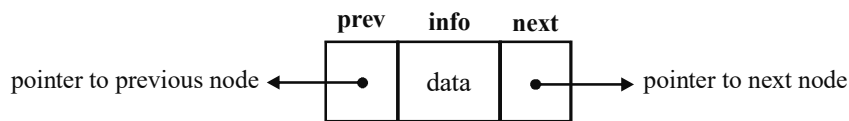
Fig. 3.43 *Node of a Doubly Linked List*

Since a doubly linked list allows traversing in both the forward and backward directions, it is also referred to as a **two-way list**. Figure 3.44 shows an example of a doubly linked list having four nodes. Note that the `prev` field of the first node and the `next` field of the last node in a doubly linked list points to `NULL`.
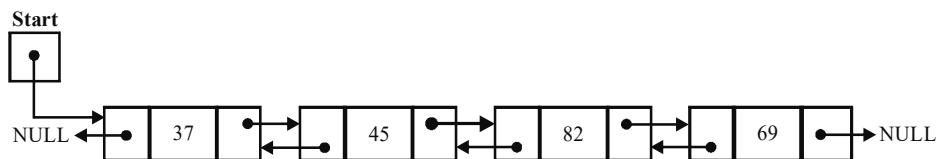
Fig. 3.44 *An Example of a Doubly Linked List with Four Nodes*

To define the node of a doubly linked list in 'C', the structure used to represent the node of a singly linked list is extended to have an extra pointer, which points to the previous node. The structure of a node of a doubly linked list is shown as follows:

```
typedef struct node
{
    int info;              /*to  store  integer  type
data*/
    struct node *next;     /*to store a pointer to next
node*/
    struct node *prev;     /*to  store  a  pointer  to
previous node*/
}Node;
Node *nptr;          /*nptr is a pointer to node*/
```

When memory is allocated successfully to a node, that is, there is no overflow, the node is initialized. The `info` field is initialized with a valid value and the `prev` and `next` fields are initialized with `NULL`.

Algorithm 3.32 Creating a Node of a Doubly-linked List

```
create_node()

1. Allocate memory for nptr    //nptr is a pointer to new node
2. If nptr = NULL
      Print "Overflow: Memory not allocated!" and go to step 8
3. Read item                   //item is the value stored in the
   node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Set nptr->prev = NULL
7. Return nptr
8. End
```

Note that all the operations that are performed on singly linked lists can also be performed on doubly linked lists.

Insertion

To insert a new node in the beginning of a doubly linked list, a pointer (say, nptr) to the new node is created. The next field of the new node is made to point to the existing first node and the prev field of the existing first node (that has become the second node now) is made to point to the new node. After that, Start is modified to point to the new node. Figure 3.45 shows the insertion of a node at the beginning of a doubly linked list.
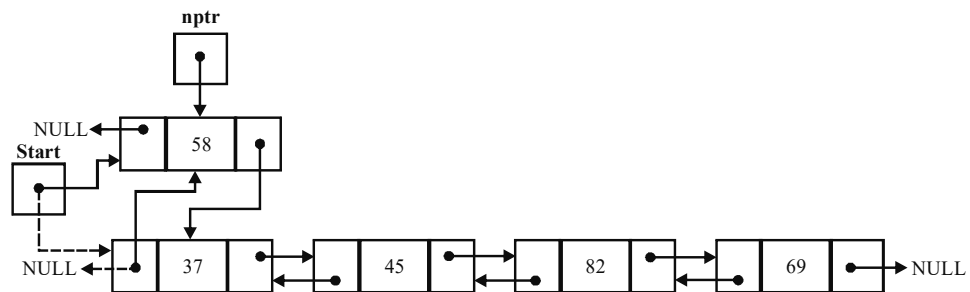
Fig. 3.45 *Insertion in the Beginning of Doubly Linked List*

Algorithm 3.33 Insertion in the Beginning

```
insert_beg(Start)

1. Call create_node()                  //creating a new node pointed
   to by nptr
2. If Start != NULL
           Set nptr->next = Start       //inserting node in the
beginning
      Set Start->prev = nptr
      End If
3. Set Start = nptr                  //making Start to point to new
   node
4. End
```

To insert a new node at the end of a doubly linked list, the list is traversed up to the last node using some pointer variable (say, temp). At the end of traversing, temp points to the last node. Then, the next field of the last node (pointed to by

temp) is made to point to the new node and the `prev` field of the new node is made to point to the node pointed to by `temp`. However, if the list is empty, the new node is inserted as the first node in the list. Figure 3.46 shows the insertion of a new node at the end of a doubly linked list.

NOTES
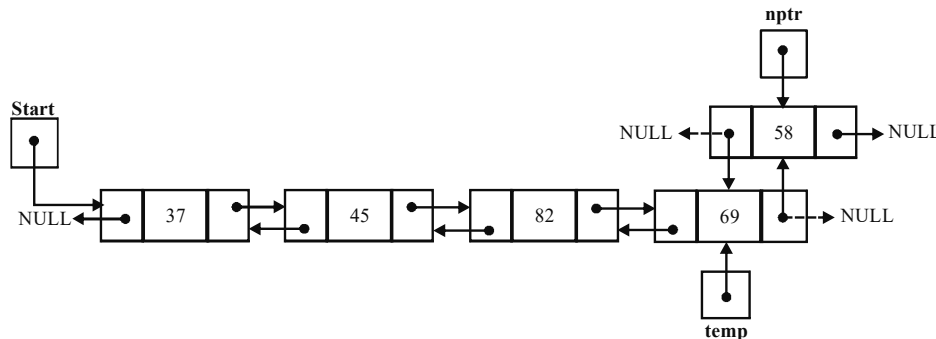
Fig. 3.46 Insertion at the End

Algorithm 3.34 Insertion at the End

```
insert_end(Start)

1. Call create_node()          //creating a new node pointed to by
   nptr
2. If Start = NULL
           Set Start = nptr  //inserting new node as the first
node
   Else
      Set temp = Start        //pointer temp used for traversing
      While temp->next != NULL
           Set temp = temp->next
      End While
      Set temp->next = nptr
      Set nptr->prev = temp
   End If
3. End
```

To insert a new node (pointed to by `nptr`) at a specified position (say, `pos`) in a doubly linked list, the list is traversed up to `pos-1` position. At the end of traversing, `temp` points to the node at `pos-1` position. For simplicity, another pointer variable (say, `ptr`) is used to point to the node that is already at `pos` position. Then, the `prev` field of the node pointed to by `ptr` is made to point to the new node and the `next` field of the new node is made to point to the node pointed to by `ptr`. Also, the `prev` field of the new node is made to point to the node pointed to by `temp` and the `next` field of the node pointed to by `temp` is made to point to the new node. Figure 3.47 shows the insertion of a new node at the third position in a doubly linked list.
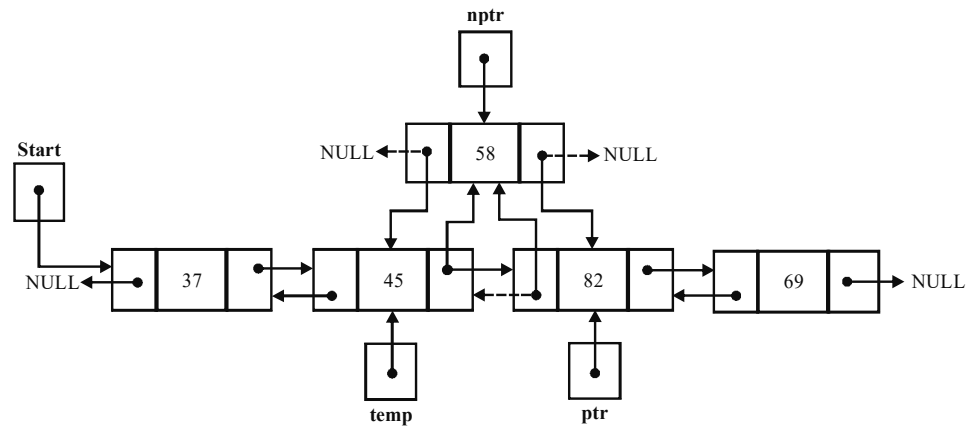
Fig. 3.47 Insertion at a Specified Position

Algorithm 3.35 Insertion at a Specified Position

```
insert_pos(Start)

1. Call create_node()    //creating a new node pointed to by nptr
2. Set temp = Start
3. Read pos
4. Call count_node(temp)//counting number of nodes in count
5. If pos = 0 OR pos > count + 1
      Print "Invalid position!" and go to step 7
   End If
6. If pos = 1
           Set nptr->next = Start //inserting node at the beginning
      Set Start = nptr             //Start pointing to new node
   Else
      Set i = 1
      While i < pos-1             //traversing up to the node at pos-
1 position
           Set temp = temp->next
           Set i = i + 1
      End While
      Set ptr = temp->next
      Set ptr->prev = nptr
      Set nptr->next = ptr
      Set nptr->prev = temp
      Set temp->next = nptr
   End If
7. End
```

Deletion

To delete a node from the beginning of a doubly linked list, a pointer variable (say, temp) is used to point to the first node. Then, Start is modified to point to the next node and the prev field of this node is made to point to NULL. After that, the memory occupied by the node pointed to by temp is de-allocated. Figure 3.48 shows the deletion of a node from the beginning of a doubly linked list.
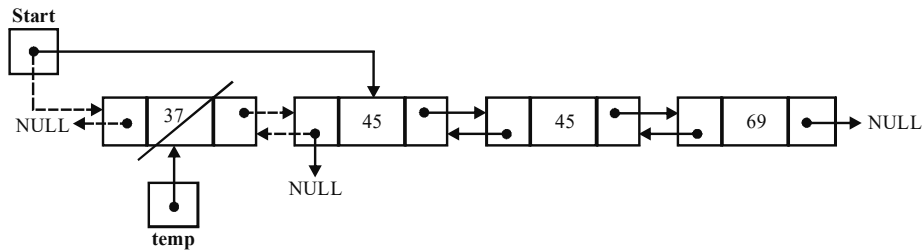
Fig. 3.48 *Deletion from the Beginning in a Doubly Linked List*

Algorithm 3.36 Deletion from the Beginning

```
delete_beg(Start)

1. If Start = NULL
      Print "Underflow: List is empty!" and go to step 6
   End If
2. Set temp = Start              //temp points to the node to
   be deleted
3. Set Start = Start->next       //making Start to point to
   next node
4. Set Start->prev = NULL
5. De-allocate temp              //de-allocating memory
6. End
```

Note: The process of deleting a node from the end of a doubly linked list is the same as that of a singly linked list.

To delete a node from a position (say, `pos`) specified by the user, the list is traversed up to the `pos` position using the pointer variables `temp` and `save`. At the end of traversing, `temp` points to the node at `pos` position and `save` points to the node at `pos-1` position. Here, for simplicity, another pointer variable `ptr` is used to point to the node at `pos+1` position. Then, the `next` field of the node at `pos-1` position (pointed to by `save`) is made to point to the node at `pos+1` position (pointed to by `ptr`). In addition, the `prev` field of the node at `pos+1` position (pointed to by `ptr`) is made to point to the nod e at `pos-1` position (pointed to by `save`). After that, the memory occupied by the node pointed to by `temp` is de-allocated. Figure 3.49 shows the deletion of a node at the third position from a doubly linked list.
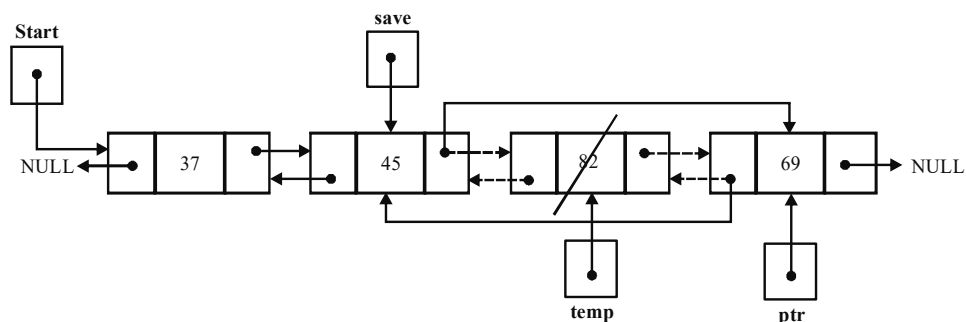
Fig. 3.49 *Deletion from a Specified Position in a Doubly Linked List*

Algorithm 3.37 Deletion from a Specified Position

```
delete_pos(Start)

1. If Start = NULL
      Print "Underflow: List is empty!" and go to step 8
   End If
2. Set temp = Start
3. Read pos
4. Call count_node(temp)          //counting  total  number  of
   nodes in count variable
5. If pos > count OR pos = 0
      Print "Invalid position!" and go to step 6
   End If
6. If pos = 1
      Set Start = Start->next      //deleting the first node
      Start->prev = NULL
   Else
      Set i = 1
      While i < pos                //traversing up to the node at
pos position
            Set save = temp        //save pointing to the node at
pos-1 position
            Set temp = temp->next  //making temp to point to next
node
            Set i = i + 1
      End While
      Set ptr = temp->next
      Set save->next = ptr
      Set ptr->prev = save
   End If
7. De-allocate temp                //de-allocating memory
8. End
```

Note: A doubly linked list, in which the next field of the last node points to the first node instead of NULL, is termed as a **doubly-circular linked list**.

Program 3.9: The following program illustrates the implementation of a doubly linked list:

```c
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
    int info;
    struct node *next;
    struct node *prev;
}Node;          /* node of a doubly-linked list */
/* Function prototypes */
Node * create_node();
int isempty(Node *);
void display(Node *);
int count_node(Node *);
void insert_beg(Node **);
void insert_end(Node **);
```

```
void insert_pos(Node **);
void delete_beg(Node **);
void delete_end(Node **);
void delete_pos(Node **);
/*Main Function*/
void main()
{
int item,ch,ch1;
Node *Start=NULL;
do
{
clrscr();
printf("\n\n\tMain Menu");
printf("\n1. Insert");
printf("\n2. Delete");
printf("\n3. Display");
printf("\n4. Exit\n");
printf("\nEnter your choice: ");
scanf("%d",&ch);
switch(ch)
{
    case 1: printf("\n1. Insert in the beginning");
                printf("\n2. Insert at the end");
                    printf("\n3. Insert at a specified
position");                        printf("\n4.
Back to main menu\n");
    printf("\nEnter your choice: ");
            scanf("%d",&ch1);
switch(ch1)
{
case 1:insert_beg(&Start);
break;
case 2:insert_end(&Start);
break;
case 3:insert_pos(&Start);
break;
case 4:   break;
default:printf("\nInvalid choice!");
}
break;
case 2: printf("\n1. Delete from the beginning");
        printf("\n2. Delete from the end");
        printf("\n3. Delete from a specified position");
            printf("\n4. Back to main menu\n");
            printf("\nEnter your choice: ");
            scanf("%d",&ch1);
```

```
switch(ch1)
{
case 1:delete_beg(&Start);
break;
case 2:delete_end(&Start);
break;
case 3:delete_pos(&Start);
break;
case 4:break;
default: printf("\nInvalid choice!");
}
break;
case 3:display(Start);
    break;
        case 4: exit();
        default: printf("\nInvalid choice!");
    }
    getch();
}while(1);
}
Node * create_node()
{
    Node *nptr;
    int item;
    nptr=(Node *)malloc(sizeof(Node));
    if(nptr==NULL)
    {
        printf("\nOverflow: Memory not allocated!");
        exit();
    }
    printf("\nEnter the value to be inserted: ");
    scanf("%d",&item);
    nptr->info=item;
    nptr->next=NULL;
    nptr->prev=NULL;
    return(nptr);
}
int isempty(Node *Start)
{
    if(Start==NULL)
        return True;
    else
        return False;
}
```

```
void display(Node *Start)
{
    Node *temp=Start;
    if(temp==NULL)
        printf("\nList is empty!!");
    else
    {
        printf("\nThe linked list is: ");
        while(temp != NULL)
        {
            printf("%d ",temp->info);
            temp=temp->next;
        }
    }
}
int count_node(Node *Start)
{
    Node *temp=Start;
    int count=0;
    while(temp != NULL)
    {
        count++;
        temp=temp->next;
    }
    return(count);
}
void insert_beg(Node **Start)
{
    Node *nptr=create_node();
    if (*Start != NULL)
    {
        nptr->next=*Start;
        (*Start)->prev=nptr;
    }
    *Start=nptr;
    printf("\nNode inserted.");
}
void insert_end(Node **Start)
{
    Node *temp;
    Node *nptr=create_node();
    if(*Start==NULL)
        *Start=nptr;
    else
```

```
    {
        temp=*Start;
        while(temp->next != NULL)
            temp=temp->next;
        temp->next=nptr;
        nptr->prev=temp;
    }
    printf("\nNode inserted.");
}
void insert_pos(Node **Start)
{
    int i,pos,count;
    Node *nptr=create_node();
    Node *temp=*Start,*ptr;
    printf("\nEnter the position at which you want to
insert: ");
    scanf("%d",&pos);
    count=count_node(temp);
    if(pos==0 || pos>count+1)
{
printf("\nInvalid position!");
return;
}
if(pos==1)
{
    nptr->next=*Start;
    *Start=nptr;
}
else
{
    for(i=1;i<pos-1;i++)
    temp=temp->next;
    ptr=temp->next;
ptr->prev=nptr;
nptr->next=ptr;
nptr->prev=temp;
temp->next=nptr;
}
printf("\nNode inserted.");
}
void delete_beg(Node **Start)
{
    Node *temp=*Start;
    *Start=(*Start)->next;
```

```
    (*Start)->prev=NULL;
    free(temp);
    printf("\nNode deleted.");
}
void delete_end(Node **Start)
{
    Node *temp=*Start;
    Node *save;
    if(isempty(temp))
    {
        printf("\nUnderflow: List is empty!");
        return;
    }
    if(temp->next==NULL)
        *Start=NULL;
    else
    {
        while(temp->next != NULL)
        {
            save=temp;
            temp=temp->next;
        }
        save->next=NULL;
    }
    free(temp);
    printf("\nNode deleted.");
}
void delete_pos(Node **Start)
{
    Node *temp=*Start,*save,*ptr;
    int i,pos,count;
    printf("\nEnter the position of the node to be deleted:
");
    scanf("%d",&pos);
count=count_node(temp);
if(pos>count)
{
    printf("\nInvalid position!\n");
    return;
}
if(pos==1)
{
    *Start=temp->next;
    (*Start)->prev=NULL;
```

```
            }
else
{
    for(i=1;i<pos;i++)
    {
save=temp;
temp=temp->next;
}
ptr=temp->next;
save->next=ptr;
ptr->prev=save;
}
free(temp);
printf("\nNode deleted.\n");
}
```

The output of the Program is as follows:

```
Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu
Enter your choice: 1
Enter the value to be inserted: 6
Node inserted.

Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu
```

```
Enter your choice: 2
Enter the value to be inserted: 5
Node inserted.


Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu
Enter your choice: 3
Enter the value to be inserted: 8
Enter the position at which you want to insert: 2
Node inserted.


Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
The linked list is: 6 8 5


Main Menu
1. Insert
2. Delete
3. Display
4. Exit


Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu
Enter your choice: 3
Enter the position of the node to be deleted: 4
Invalid position!
```

```
Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu
Enter your choice: 1
Node deleted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3
The linked list is: 8 5

Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu
Enter your choice: 2
Node deleted.

Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3
The linked list is: 8
```

```
Main Menu
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 4
```

3.8 GENERALIZED LIST

A generalized list object is an ordered arrangement of zero or more elements of a certain type, including generalized lists.

- Allows *sublists*
    - o A list consists of a *head* and a *tail*
    - o The *empty list* is without any elements
    - o The head and tail functions
    - o Lists may occur more than once, possibly recursively
- Representation
- Example application: Polynomials in several variables
    - o Factor out each variable with each exponent
    - o Top list is the first variable, with a head node giving the variable name, then nodes for each exponent
    - o Sublists for each node, of the same nature for subsequent variables
- Recursive algorithms
    - o Copying a list
    - o Recursive algorithm
    - o Time complexity = O(NumberOfNodes)
- Space complexity = O(Longest branch)
    - o List equality
    - o List depth
- Shared and recursive lists
    - o Sharing helps save space
    - o Sharing requires naming of lists, implemented as head nodes with reference counts (useful for deleteing)
    - o Deleting shared lists, and recursive lists

Check Your Progress

26. What is a circular linked list?

27. What is the advantage of a circular linked list over a linear linked list?

28. Where is the address of the first node of a circular linked list contained?

29. Write a function in C to count the number of nodes in a circular linked list.

30. What is a doubly linked list?

31. What is a generalized list?

3.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The logical or mathematical model used to organize data in the main memory is called data structure.

2. The data types provided by a programming language are known as primitive data types or in-built data types and the data types that are derived from primitive data types of the programming language are known as composite data types or derived data types.

3. An abstract data type (ADT) is an extension of modular design in a way that the set of operations of an ADT are defined at a formal and logical level and nowhere in an ADT's definition it is mentioned how these operations are implemented. The data type integer is an example of abstract data type.

4. A program can be written in different languages like COBOL, FORTRAN, PASCAL, C, C++, JAVA, Python etc.

5. The problem solving is the main initiator of writing algorithms or computer programs.

6. The analysis of an algorithm for efficiency is a method to evaluate the efficiency of an algorithm for different parameters.

7. The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time.

8. The insertion of an element in a stack is known as a 'push' operation.

9. The deletion of an element from a stack is known as a 'pop' operation.

10. A stack can be implemented as an array as well as a linked list.

11. Stacks can be used for reversing strings, checking whether the arithmetic expression is properly parenthesized, converting infix notation to postfix and prefix notations, evaluating postfix expressions, and implementing recursion and function calls.

12. In a stack, the condition Top=-1 indicates that the stack is empty.

13. An infix expression can be converted to a postfix expression using stacks.

14. A queue is a type of collection in which the entities contained in it are kept in order and the principal operations on the collection are the addition of

entities to the rear terminal position and removal of entities from the front terminal position. Queues are useful in computer science, transport and operations research where various entities such as data, objects, persons or events are stored in order to be processed later. In such cases, the queue plays the role of a buffer.

15. The primary characteristic of a queue data structure is that it allows access only to the front and back of the structure.

16. The Enqueue operation adds an item onto the end of the queue.

17. Queues are implemented using arrays and pointers.

18. A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.

19. A linked list is a linear collection of homogeneous elements called **nodes**.

20. Each node of a singly linked list consists of two fields: info and next. The info field contains the data and the next field contains the address of memory location where the next node is stored. The last node of the linked list contains NULL in its next field that indicates the end of the list.

21. A dynamic data structure is one in which the memory for elements is allocated dynamically at runtime.

22. The structure definition to define a node of a linked list containing the roll numbers and names of students is as follows:

```
struct stu_node
{
int roll_no;         //to store roll number of                student
char name[25];          //to store name of student
struct stu_node *next;   //to store pointer to next node
};
```

23. A situation where the user tries to delete a node from an empty linked list is termed as underflow.

24. While searching in an unsorted list, the list is traversed completely starting from the first node towards the last node and the value of each node (that is, node->info) is compared with the value to be searched. On the other hand, in a sorted linked list, while comparing, once the value of any node exceeds item (the value to be searched), the search is stopped immediately. In that case, the list is not required to be traversed completely.

25. To reverse a linked list, the list is traversed up to the last node and the links of the nodes are reversed such that the first node of the list becomes the last node and the last node becomes the first node. For this, three pointer variables (say, save, ptr and temp) are used. Initially, temp points to Start, and both ptr and save point to NULL. While traversing the list, temp points to the current node, ptr points to the node previously pointed to by temp and save points to the node previously pointed to by ptr. The links between nodes are reversed by making the next field of the node pointed to by ptr to

point to the node pointed to by save. At the end of traversing, temp points to NULL, ptr points to the last node and save points to the second last node of the list. Then Start is made to point to the node pointed to by ptr in order to make the last node as the first node of the list.

26. A linear linked list, in which the next field of the last node points back to the first node instead of containing NULL, is termed a circular linked list.

27. The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, you can reach any of its predecessor nodes.

28. The address of the first node is contained by the last node of the circular linked list.

29. The function to count the number of nodes in a circular linked list is as follows:

```
int count_node(Node *Start)
{
        Node *ptr=Start;
        int count=0;
        do
    {
        count++;
        ptr=ptr->next;
    }while(ptr != Start);
        return count;
}
```

30. A doubly linked list is a linear collection of homogeneous data elements, called nodes, where each node contains three fields: prev, info and next. The info field stores the data, the prev field stores the address of the previous node, and the next field stores the address of the next node.

31. A generalized list object is an ordered arrangement of zero or more elements of a certain type, including generalized lists.

3.10 SUMMARY

- The logical or mathematical model used to organize data in the main memory is called **data structure**. Various data structures are available, each with its special features. These features should be kept in mind while choosing a data structure for a particular situation.

- A linear data structure is one in which the elements form a sequence. This means each element in the structure has a unique predecessor and a unique successor. An array is the simplest linear data structure.

- A **linked list** is a linear collection of similar data elements called **nodes**, with each node containing some data and pointer(s) pointing to other node(s) in the list. Nodes of a linked list are not constrained to be at contiguous memory locations; instead they can be stored anywhere in the memory. The linear order of the list is maintained by the pointer field(s) in each node.

- A **queue** is a linear data structure in which the addition or insertion of a new element occurs at one end called **Rear**, and the deletion of an element occurs at the other end called **Front**.

- A non-linear data structure is one in which the elements do not form a sequence. It means, unlike a linear data structure, each element is not constrained to have a unique predecessor and a unique successor.

- The data types provided by a programming language are known as *primitive data types* or *in-built data types*.

- An abstract data type (ADT) is an extension of a modular design in a way that the set of operations of an ADT are defined at a formal, logical level, and nowhere in ADT's definition, it is mentioned how these operations are implemented. The data type integer is an example the abstract data type.

- Brute force is a general technique which is used for finding solutions to various problems. In this technique, all possible candidates for the solution are listed and then examined to check whether each candidate satisfies the problem.

- The divide and conquer technique is one of the widely used technique is to develop algorithms will for problems which can be divided into sub-problems (smaller in size but similar to the actual problem) so that they can be solved efficiently. The technique follows a top-down approach.

- Dynamic programming is a technique that is generally used for solving optimization problems where the best (optimal) solution out of the available possible solutions is to be found.

- A computer executes the set of statements also known as a program in order to achieve the desired objective.

- A program can be written in different languages like COBOL, FORTRAN, PASCAL, C, C++, JAVA, Python etc.

- The statements written in a program differ from one language to another.

- An algorithm is a systematic sequence of instructions which are required to be executed in order to achieve the objective of an algorithm.

- An algorithm needs to written in a sequence of instructions which are executed one after another. The sequence will always be implemented from line number first till the line number last.

- The algorithm written to add two numbers is not necessarily the only method which can be used to automate the process of adding two numbers.

- The number of variables used in the above given algorithms varies which results in optimal usage of resources.

- The comparison of the algorithms based on the usage of resources like memory-usage, time-taken etc motivates the reader to study the domain of algorithm analysis.

- The problem solving is the main initiator of writing algorithms or computer programs.

- Every individual would like to develop automated tools for specific problem solving methods or procedures.

- The process of writing an algorithm for a problem solving methods includes different steps.

- The requirement of a user can be an overview but as a designer of an algorithm the developer needs to understand all the processes and sub processes within the system.

- An algorithm developer needs to discuss all the confusions in order to understand the problem without any ambiguities.

- The evaluation of multiple alternatives will help a developer to prepare a plan for developing an algorithm which is designed to automate the process of problem solution effectively and efficiently.

- Flow chart is a graphical representation to display the sequence of instruction within an algorithm or a computer program.

- The searching problem is related to searching an item from a list of items. The item searched can be an integer or a character which helps an end-user in searching for an attribute as and when required based on the requirements.

- A graph is a collection of nodes also known as vertices and the vertices are connected with one another using edges.

- The analysis of an algorithm for efficiency is a method to evaluate the efficiency of an algorithm for different parameters

- The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time.

- The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time.

- A stack is a linear data structure in which an element can be added or removed only at one end called top of the stack. In stack terminology, the insert and delete operations are known as push and pop operations, respectively.

- A stack works on the principle of last-in-first-out (LIFO) list. A stack can be organized (represented) in memory either as an array or as a singly linked list. In both the cases, the insertion and deletion of elements are allowed only at one end.

- An array representation of a stack is static and a linked list representation is dynamic in nature. Before we insert a new element onto a stack, it is necessary to test the condition of overflow. Similarly, before we remove an item from a stack, it is necessary to test the condition of underflow.

- Stacks are used where the last-in-first-out principle is required like in the case of reversing strings, checking whether the arithmetic expression is properly parenthesized, converting infix notation to postfix and prefix notations, evaluating postfix expressions, implementing recursion and function calls, etc.

- If a program needs two stacks, then it is advisable to represent both of them in the same array as this leads to most efficient use of memory.

- A queue is a kind of collection wherein the entities it contains are kept in an order and the principal operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position.

- Queues allow access only to the front and back of the structure.

- Enqueue, front(), dequeue(), is-empty(), is-full() and get-size() are the primitive operations performed on queues.

- A queue does not have a specific capacity. Inspite of the number of elements already contained, there is always scope to add a new one.

- A queue overflow results from an attempt to add an element onto a full queue and queue underflow occurs when trying to remove an element from an empty queue

- A queue may be implemented as arrays or as pointers.

- When you want to enqueue an object, you add it to the back of the item pointed to by the tail pointer.

- When an item needs to be dequeued off the list, the head pointer is pointed to the previous item from the head. The old head item is the one that has been removed of the list.

- In case of a queue represented as an array, once the value of the rear reaches the maximum size of the queue, no more elements can be inserted.

- A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.

- A deque (short form of double-ended queue) is a linear list in which elements can be inserted or deleted at either end but not in the middle. That is, elements can be inserted/deleted to/from the rear end or the front end.

- There are various types linked lists such as singly linked lists, circular linked lists and doubly-linked lists.

- A dynamic data structure is one in which the memory for elements is allocated dynamically at runtime. Linked list is an example of a dynamic structure.

- A linked list is a linear collection of homogeneous elements called nodes.

- The successive nodes of a linked list need not occupy adjacent memory locations and the linear order between nodes is maintained by means of pointers.

- In a singly linked list (also called linear linked list), each node consists of two fields: info and next, which contain the data and the address of the memory location where the next node is stored, respectively.

- Each node of a singly linked list contains only a single pointer pointing to the next node, thereby allowing traversing in only one direction; therefore, it is referred to as a one-way list.

- To maintain a linked list in memory, two parallel arrays of equal size are used.

- The memory manager also maintains a special list known as free storage list or memory bank or free pool that consists of unused memory cells. This special list keeps track of the free space available in the memory and a pointer to this list is stored in a pointer variable Avail. When there is no space available, that is, the free storage list is empty, then this situation is termed as overflow.

- A number of operations can be performed on singly linked lists which include creating, traversing, searching, inserting and deleting nodes, reversing, sorting and merging linked lists.

- A situation where the user tries to delete a node from an empty linked list is termed as underflow.

- To reverse a linked list, the list is traversed up to the last node and the links of the nodes are reversed such that the first node of the list becomes the last node and the last node becomes the first node.

- A linear linked list in which the next field of the last node points back to the first node instead of containing NULL is termed a circular linked list.

- The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, it is possible to reach any of its predecessor nodes.

- A linked list that maintains an additional pointer pointing to the previous node in each node of the list is termed a doubly linked list.

- Each node of a doubly linked list consists of three fields: prev, info and next, where in the info field contains the data, the prev field contains the address of the previous node and the next field contains the address of the next node.

- Since a doubly linked list allows traversing in both forward and backward directions, it is also referred to as a two-way list.

3.11 KEY TERMS

- **Data Structure:** Data structure is the logical or mathematical model used to organize data in the main memory.

- **Algorithm:** An algorithm is a systematic sequence of instructions which are required to be executed in order to achieve the objective of an algorithm.

- **Searching Problem***:* It is related to searching an item from a list of items. The item searched can be an integer or a character which helps an end-user in searching for an attribute as and when required based on the requirements.

- **Linked List:** It refers to a linear collection of similar data elements, called nodes, with each node containing some data and pointer(s) pointing to other node(s) in the list.

- **Linear Linked List:** When each node of a linked list contains only one pointer and it points to the next node, it is known as linear linked list.

- **Stack:** It is a linear list of data elements in which addition of a new element or deletion of an existing element occurs only at one end.

- **Queue:** It is a linear data structure in which addition or insertion of a new element occurs at one end, called 'rear' and deletion of an element occurs at other end, called 'front'.

- **Non-Linear Data Structure:** It is a structure in which the elements do not form a sequence.

- **Stack:** It is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end.

- **Enqueue:** It is the process of adding something to a queue or list of to-do processes for a program.

- **Double-Ended Queue:** It is an abstract data structure that implements a queue for which elements can only be added to or removed from the front or back.

- **Input-Restricted Dequeue:** It is a data class sub-type wherein removal can be made from both ends but input can only be at one end.

- **Out-put Restricted Dequeue:** It is a data class sub-type wherein input can be made at both ends but output can be made from one end only.

- **Priority Queue:** It is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.

- **Deque:** It is a linear list in which elements can be inserted or deleted at either end but not in the middle.

- **Dynamic Data Structure:** It is a data structure in which the memory for elements is allocated dynamically at runtime.

- **Linked List:** It is a linear collection of homogeneous elements called nodes.

- **Singly linked List:** It is a sequence of dynamically allocated objects, each of which refers to its successor in the list.

- **Circular linked List:** It is a linear linked list in which the next field of the last node points back to the first node instead of containing NULL.

- **Doubly Linked List:** It is a data structure in which each element contains pointers to the next and previous elements in the list, thus forming a bidirectional linear list.

3.12 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write short notes on the following

   (a) Arrays   (b) Trees

   (c) Linked lists   (d) Graphs

2. What is a non-linear data structure?

3. List the steps used in dynamic programming.

4. Write in brief about the greedy algorithm design technique.

5. What is the complexity of algorithms?

6. How can one understand the problem?

7. What are the objectives of problem identification?

8. What are the two ways of implementing stacks? Which one is preferred over the other and why?

9. Differentiate between infix, postfix and prefix expressions.

10. State the difference between a stack and a queue.

11. What are the advantages of a circular queue?

12. What is meant by a dynamic data structure? What are its advantages over a static data structure?

13. What is a linked list? How is it represented in memory?

14. Mention the applications of linked list.

15. What is the significance of the NULL pointer in a linked list?

16. Write an algorithm to insert a new node after a specified node in a singly linked list.

17. Consider a linked list containing some even and odd integer values. Write an algorithm to split this list into two lists—one containing even numbers and the other containing odd numbers.

18. Write a function in C to make the first node of the list the last node of the list without changing any values.

19. Write a function in C to merge two sorted linked lists into a single sorted linked list with elimination of duplicates.

20. Write a short note on static and dynamic memory allocation of linked list.

Long-Answer Questions

1. Explain why the analysis of an algorithm is required.

2. Discuss the different evaluation criterions used in evaluating the efficiency of an algorithm.

3. For a program to find out summation of 10 numbers find out the different methods to write an algorithm.

4. Describe the applications of stacks. Write a C program to implement any one of them.

5. Write a program in C to determine whether a given string is a palindrome or not with the help of a stack.

6. Write a C function to convert an infix expression into a postfix notation.

7. Convert the following infix expressions into equivalent postfix and prefix expressions:
   a. A*B+(C-D/F)
   b. A*(B+D)/E-F-(G+H/K)
   c. A-B)*(C/D)+E
   d. A+(B*C-(D/E^F)*G)*H

8. Describe the ways in which queues work.

9. Discuss the benefits and drawbacks of using linked lists.

10. Write a note on the various types of queues. Differentiate between queue overflow and queue underflow.

11. Consider a linked list containing integer values and write a function in C to find the average MEAN of the values.

12. Write a program in C to create a linked list storing the names, ages, and salaries of ten employees. Then arrange the list in the descending order of salaries.

13. Write a program in C to maintain a doubly-circular linked list.

14. Write an algorithm to insert a new node at a specified position in a circular linked list.

15. Write a program in C to create an ascending order doubly linked list.

16. Describe the procedure of deleting a node from the end of a circular linked list.

3.13 FURTHER READING

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures.* New Delhi: Galgotia Publications, 2003.

Tanenbaum, A.M., Yedidyah Langsam and Moshe J. Augenstein. *Data Structures using C and C++*. New Delhi: Prentice-Hall of India, 1995.

UNIT 4 TREES AND GRAPHS

Structure

4.0 INTRODUCTION

In computer science, a tree is a widely-used data structure that resembles a tree structure with a set of linked nodes. These trees grow downwards contrary to the trees in nature. A node that has a child is called the child's parent node (or ancestor node or superior). A tree is known as a binary tree if each of its nodes gives rise to only most two branches, that is, two child nodes. Binary tree are maintained by sequential or linked list representations. The process of visiting each node in a tree data structure, exactly once, in a systematic way is called tree-traversal. A tree is a special type of non-linear data structure called graph, which differs from the tree in the sense that it represents the relationship that is not necessarily hierarchical in nature, that is, a node in a graph may have more than one parent. The graph is widely used in real-life applications, like finding the shortest routes, statistical analysis, etc. This unit will introduce you to the various aspects of graphs such as representing and traversing a graph. Various applications of graphs are also discussed in this unit.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the basic terminologies of trees

- Describe the meaning of binary trees

- Explain the theorems associated with trees

- Discuss the process of deletion from a binary tree

- State the terminology associated with graph

- Illustrate the graphs using adjacency matrix and path matrix
- Describe the traversing of graphs using depth-first and breadth-first search
- Discuss topological sorting and minimal spanning tree

4.2 TREES: BASIC TERMINOLOGY

The data structure of 'trees' has the components root (for a base), branches (for growth) and leaves (for existence). Tree with some or all branches contain no leaves. The roots of the computer trees will be at the top while the leaves will be at the bottom. The direction from root to the leaves is 'downward' while the opposite direction is 'upward'. Going up from the leaves to the root is known as 'climbing' the tree and going from the root to the leaves is known as 'descending' the tree.

Trees are bi-dimensional data structures with hierarchical relationship between data items. A tree and its components can be described as follows:

Tree: A tree is a non-empty collection of vertices (nodes) and edges that satisfy certain requirements.

Path: A path in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree.

Root: One node in the tree is designated as the root.

Graph: A graph is the structure, which has more than one path between the root and some node or no path at all.

Forest: A set of trees is called a forest.

Recursive tree: A tree is either a single node or a root node connected to a forest.

Ordered tree: An ordered tree is a tree in which the order of children is specified.

Depth: The depth of a node is the number of nodes on the path from that node to the root. The depth of a node is the length of the path to its root (i.e., its root path).

Height: The height (maximum distance) of a tree is the maximum level among all of the nodes in the tree. The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree.

Path length: The path length of a tree is the sum of the levels of all the nodes in the tree.

Multiway tree: A tree where each node has a specific number of children appearing in a specific order is call a multiway tree. The simplest type of a multiway tree is the binary tree. Each node in a binary tree has exactly two children one of which is designated as a left child, and the other is designated as a right child.

Arborescence: It is an acyclic connected graph where each node has zero or more children nodes and at most one parent node. Furthermore, the children of each node have a specific order.

Node: A node is a structure that may contain a value, a condition or represent a separate data structure (which could be a tree of its own). Each node in a tree has

zero or more child nodes, which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent.

Leaf nodes: Nodes that do not have any children are called leaf nodes. They are also referred to as terminal nodes.

Root node: The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links.

Internal node: An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

Subtree: A subtree of a tree T is a tree consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is a proper subtree.

Check Your Progress

1. What does the data structure of tree have?
2. How would you define path as a component of a tree?
3. What does the first subset in a binary tree contain?

4.3 BINARY TREES

A tree is known as a binary tree if each node of it can have a maximum of two branches (two child nodes). In other words, it can be said that if every node of a tree can have a maximum of degree 2, then it is known as a binary tree. Typically, the child node is known as leaf and its general name as per its position is left and right.

Figure 4.1 shows five possible binary trees all with three nodes.
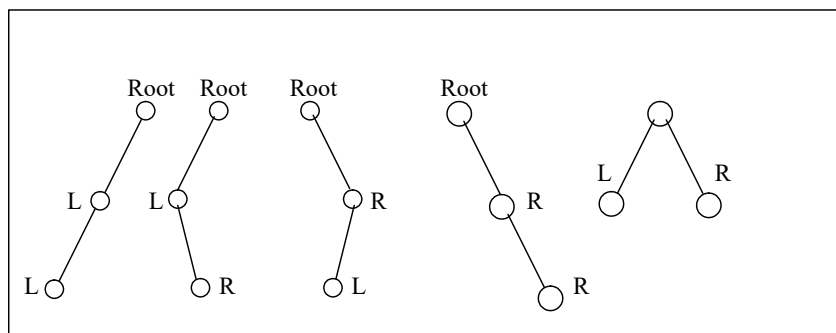
Fig. 4.1 *Five Possible Binary Trees*

Leaf and Non-Leaf Nodes

Consider a binary tree shown in Figure 4.2. The node A is a root. The nodes D, G, H and I have no branches and are known as leaf nodes. The nodes A, B, C, E and F have a branch to other nodes and are called non-leaf nodes.
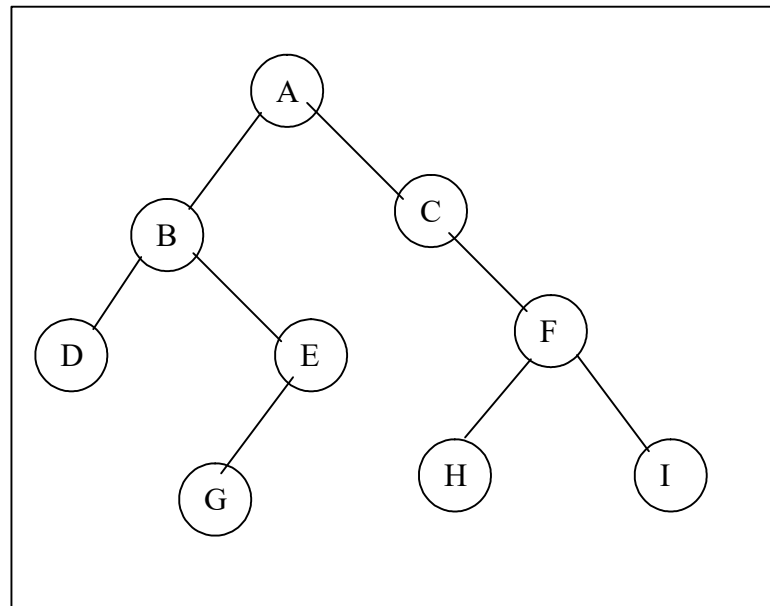
Fig. 4.2 Binary Tree

Sub trees

In Figure 4.2, node 'B' is a left subtree of root A and node 'C' is a right subtree of root A. Similarly, the node D is left subtree of node B and E is right subtree of node B. Any number of left and right subtrees are rooted at some node other than the original root node A.

Ancestor and descendant nodes

In Figure 4.2, node A is the ancestor of node E and node H is a descendant of node C. However, node E is neither an ancestor nor a descendant of node C.

A binary tree is a finite set of element. It means that it is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called root of the tree. The other two subsets are themselves binary trees called left and right subtree of the original tree. A left and right subtree can be empty. Each element is called a node of tree.
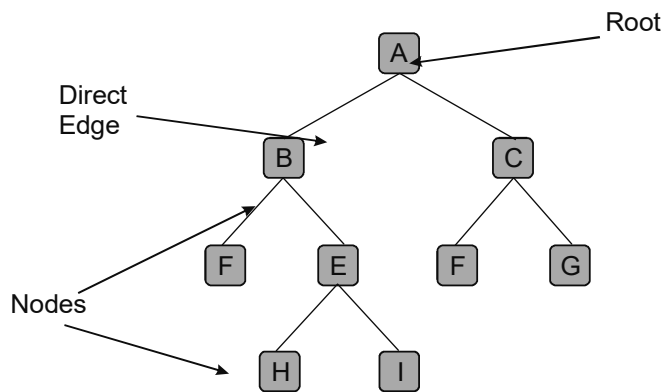
(a) Letter Representation
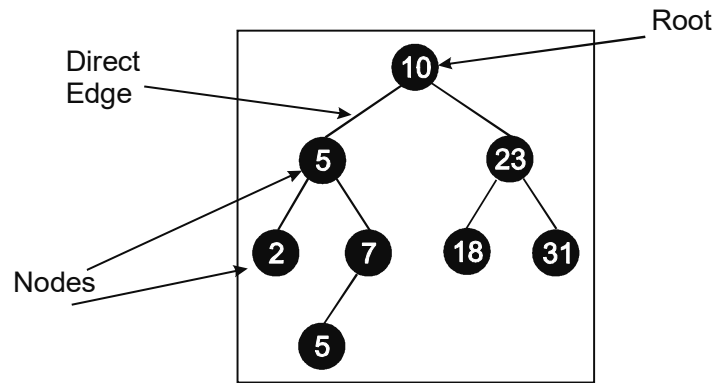
(b) Number Represenation

Fig. 4.3 *Binary Tree Representation*

Methodology

Binary tree works in the following fashion:

- The root node of a tree is the node with no parents. There is at the most one root node in a rooted tree.

- A leaf node has no children.

- A directed edge refers to the link from the parent to the child (the arrows in the picture of the tree)

- All the elements having value less than root of tree (subtree) go to the left of the root.

- All the elements having value greater than root of tree (subtree) go to right of the root.

- As shown in Figure 4.3, it is clear that 10 is the root of the main tree and 5, 7 and 23 are the roots of the different subtrees with more than 0 child.

- Considering the main tree, it is clear that 5 is less than 10 (root) and, hence, it is on the left hand side. Similarly, 23 is greater than 10(root) and, hence, it is on the right hand side of the root.

- For subtree with the root 23, you can observe that since 18 is smaller than 23 hence it is on left, and 31 is greater than 23, hence, it is on right of the root.

- The depth of a node *n* is the length of the path from the root to the node. The set of all the nodes at a given depth is sometimes known as a level of the tree. The root node is at depth zero. A tree with only a root node has a height of zero.

- We can also find out the level of a node in a binary tree. For example, root is at level '0' and level of any other node is one or more than the level of its ancestor.

- The nodes 5 and 23 are at level '1'. The nodes 2, 7, 18 and 31 are at level '2' and the nodes 5 is at level '3'. The depth of binary tree is the maximum level of any leaf in the tree

- Siblings are nodes that share the same parent node.
- In-degree of a vertex is the number of edges arriving at that vertex.
- Root is the only node in the tree with in-degree=0.

4.4 THROREMS ASSOCIATED WITH BINARY TREES

Theorem 4.1. A complete binary tree having height $h \geq 0$ includes at least 2^h and at most $2^{h+1} - 1$ nodes.

Proof. First prove the lower-bound using induction method. For a complete binary tree having height h take m_h as the minimum number of nodes. Prove the lower bound by using $m_h = 2^h$.

In this case, the base case is exactly one node in a tree having height zero (0).

Hence, $m_0 = 1 = 2^0$.

Consider that $m_h = 2^h$ for $h = 0, 1, 2, \ldots, k$ for some $k \geq 0$. Assume that a complete binary tree is having height $k + 1$ and possesses the smallest number of nodes. In it, the left subtree is a complete tree having height k with the smallest number of nodes and the right subtree is an ideal tree with height $k - 1$.

According to induction hypothesis, the left subtree contains 2^k nodes and the ideal right subtree contains the $2^{(k-1)+1} - 1$ nodes. Hence,

$$
\begin{aligned}
m_{k+1} &= 1 + 2^k + 2^{(k-1)+1} - 1 \\
&= 2^{k+1}
\end{aligned}
$$

Thus, by induction $m_h = 2^h$ for all $h \geq 0$. It proves the lower bound.

Now, prove the upper bound using induction method. Similarly, in a complete binary tree having height h take M_h as the maximum number of nodes. Prove the upper bound by using $M_h = 2^{h+1} - 1$.

In this case also the base case is exactly one node in a tree having height zero (0).

Hence, $M_0 = 1 = 2^1 - 1$.

Consider that $M_h = 2^{h+1} - 1$ for $h = 0, 1, 2, \ldots, k$ for some $k \geq 0$. Assume that a complete binary tree is having height $k + 1$ and possesses the largest number of nodes. In it, the left subtree is an ideal tree with height k and the right subtree is a complete tree having height k with the largest number of nodes.

In this case, there are accurately $2^{h+1} - 1$ nodes in an ideal left subtree. According to induction hypothesis, the right subtree contains $2^{h+1} - 1$ nodes. Hence,

$$
\begin{aligned}
M_{k+1} &= 1 + 2^{k+1} - 1 + 2^{k+1} - 1 \\
&= 2^{(k+1)+1} - 1.
\end{aligned}
$$

Thus, by induction $M_h = 2^{h+1}$ for all $h \geq 0$. It proves the upper bound.

As per the theorem, there exists exactly one complete binary tree which contains n exact number of internal nodes for every integer $n \geq 0$. Theorem also

specifies that the height of a complete binary tree having *n* internal nodes is $h =$ ëlog$_2$ *n*û. Thus, a complete binary tree holds the property of having the smallest possible internal path length.

Theorem 4.2. The internal path length of a binary tree having *n* nodes is in any case as big as the internal path length of a *complete* binary tree having *n* nodes.

Proof. Assume a binary tree having *n* nodes which has the smallest possible internal path length. Obviously, the root node is the only node at depth zero. Also, at the maximum two nodes are possible at depth one, four nodes at depth two, and so on. Hence, the internal path length of a tree having *n* nodes will be always at least as large as the sum of the first *n* terms in the series as follows:

$$\underbrace{0}_{1}, \underbrace{1, 1}_{2}, \underbrace{2, 2, 2, 2}_{4}, \underbrace{3, 3, 3, 3, 3, 3, 3, 3}_{8}, 4, \ldots$$

This summation explains the exact internal path length of a complete binary tree, because in a tree the depth of the average node is acquired by dividing the internal path length of the tree by *n*. Theorem explains that complete binary trees are the best possible as the average depth of a node in a complete tree is the smallest possible.

Theorem 4.3. The internal path length of a complete binary tree having *n* nodes is:

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = (n+1)\lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2.$$

Proof. Using the above Theorems we conclude that the internal path length of a complete tree is O (*n* log *n*). Accordingly, the depth of the average node in a complete tree will be O (log *n*).

Theorem 4.4. There are *n* - 1 edges in a tree having *n* nodes.

Proof: It can be proved using the concept that each edge is connected to a node with its parent. Except the root node every node has a parent.

Theorem 4.5. If *n* is the number of nodes in a complete binary tree having height *h*, then $2^h \pounds = n < 2^{h+1}$.

Proof. A complete binary tree having height *h* has at least one and at the most 2^h nodes at level *h*. *T*he final level may not be empty and may be filled. In any occurrence, level

h –1 for *h* > 0 must be full as per the definition of a complete binary tree. The number of nodes in a complete binary tree having height *h* - 1 is $2^{(h-1)+1} - 1$, as discussed in Theorem 1. Thus, a complete binary tree having height *h* has at least one more node than a tree having height *h* -1. Therefore, the number of nodes *n* in a complete binary tree is bounded by,

$$2^{(h-1)+1} - 1 + 1 \leq n \leq 2^{h+1} - 1$$

Therefore,

$$2^h \quad n < 2^{h+1}$$

Consistently, $h \pounds \log n < h + 1$.

4.5 TREE TRAVERSAL

Tree-traversal refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. Compared to linear data structures like lined lists and one dimensional arrays, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the orders in which they are performed define the traversal type. This is a procedure by which each node in the tree is processed exactly once.

There are three main methods of traversing a binary tree. These methods are as follows:

1. **Preorder traversal:** In this traversal, the root is processed first, followed by the left subtree and then the right subtree. To traverse a non-empty binary tree in a preorder, use the following steps:

   (i) Visit the node

   (ii) Traverse the left sub tree

   (iii) Traverse the right sub tree

(This is also called depth-first traversal.)

The designation of traversal uses (D) for root node, (L) for left subtree and (R) for right subtree. Figure 4.4 shows the preorder traversal sequence, i.e. D-L-R.
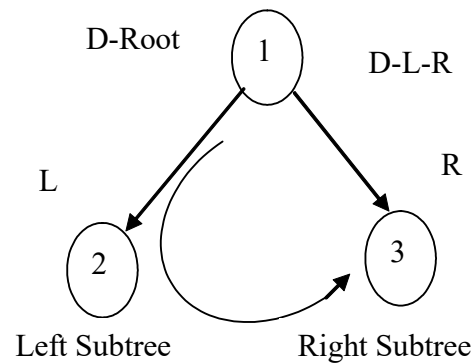
Fig. 4.4 *Preorder Traversal Sequence*

Figure 4.5 shows a binary tree with seven nodes. The processor sequence for a preorder traversal processes this tree as follows: first the root A is processed. After the root, the left subtree is processed. To process the left subtree, you first need to process its root B, then its left subtree and right subtree 's' in order. When B's left and right subtree have been processed in order, A's right subtree E is ready to be processed.
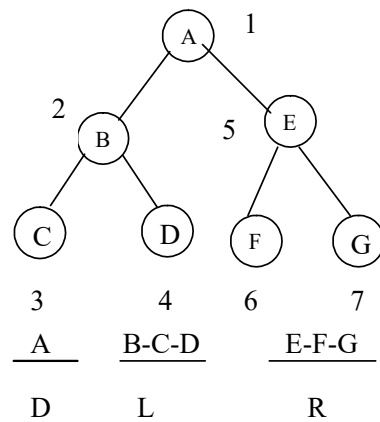
Fig. 4.5 *Binary Tree with Seven Nodes*

To process the subtree E, you first process the root and then the left subtree F and right subtree G.

Algorithm 4.1 Implementation of Preorder Traversal Recursively

```
if (root is not null)
    process (root)
    preorder (root→left)
    preorder (root→right)
return
```

2. **Inorder traversal:** In this traversal, the root node is visited in between the nodes in left and right subtrees. To traverse a non-empty binary tree in inorder, use the following steps:
   (i) Traverse the left subtree.
   (ii) Visit the node.
   (iii) Traverse the right subtree.

Figure 4.6 shows the inorder traversal sequence, i.e. L-D-R.
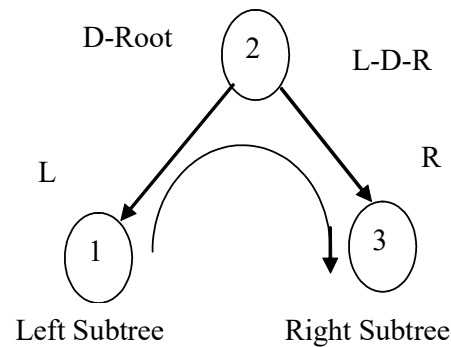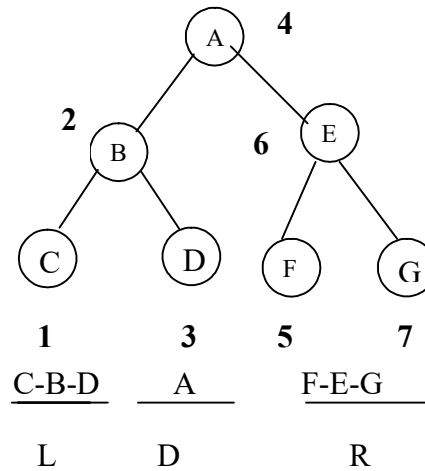
Fig. 4.6 Inorder Traversal Sequence

Figure 4.6 shows a binary tree with seven nodes. The processing sequence for an inorder traversal processes this tree as follows: left subtree must be processed first. You trace from the root to the leftmost leaf node before processing any nodes. After processing the leftmost subtree C, you process its parent root node B. You are now ready to process the right subtree D. Processing D completes the processing of root A's left subtree completely. Now you have to process the root node A, followed by its right subtree. In right subtree also you have to follow the L-D-R rule. In the right subtree of root A there are three nodes. For root E, left subtree is F and the right subtree is G as pre inorder traversal rule for processing of nodes E, F and G. left subtree of E i.e. F is processed first then root E followed by right subtree G.

Algorithm 4.2 Implementation of inorder traversal recursively
```
if (root is not null)
inorder (root→left)
process (root)
inorder (root→right)
return
```

3. **Postorder traversal:** In this traversal, the root node is visited after the nodes in the left and right and subtrees. To traverse a non-empty binary tree in postorder, use the following steps:

    (i) Traverse the left sub tree

    (ii) Traverse the right sub tree

    (iii) Visit the node

Figure 4.7 shows the postorder traversal sequence, i.e. L-R-D. Figure 4.7 also shows a binary tree with seven nodes. The processing sequence for a postorder traversal processes this tree as follows: left subtree of root must be processed first. You trace from the root to the leftmost leaf node before processing any nodes.
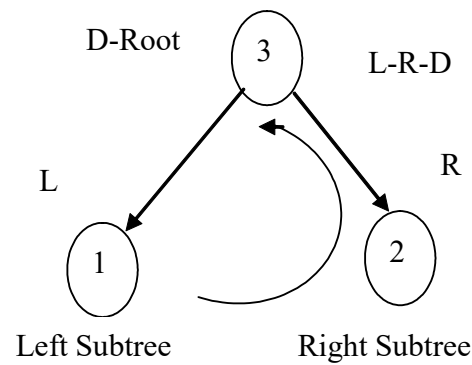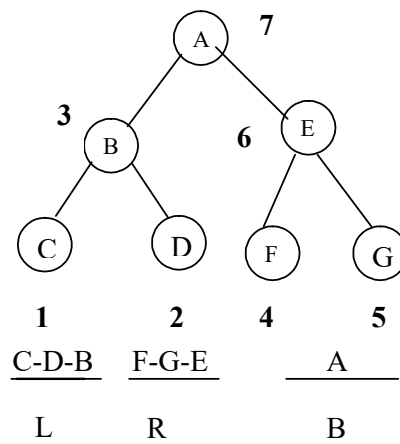
Fig. 4.7 *Postorder Traversal*

As per L-R-D rule, you first traverse left subtree of root node a. In left subtree, you follow the L-R-D rule. Initially, after processing the left most subtree node C, you process the right subtree D and then root node B. Here the processing of the left part of the binary tree is over. Now, you process the right part of root node A. In the right subtree of root node A, you follow L-R-D rule for processing nodes E, F and G. First, you process the left subtree F, then you process the right subtree G and then the root node E. Finally, you process the root A of binary tree.

Algorithm 4.3 Implementation of Postorder Traversal Recursively

```
if (root is not null)
postorder (root→left)
postorder (root→right)
process (root)
return
```

```
/* Binary tree traversal by using recursive method* */
#include<stdio.h>
#include<conio.h>
#define TRUE 1

struct node
{
```

```
                          int data;
                          struct node *left;
                          struct node *right;
                     }*root,*p,*q;

                     void setleft(struct node *,int);
                     void setright(struct node *,int);
                     void searchtree(struct node *,int);
                     void preorder(struct node *);
                     void inorder(struct node *);
                     void postorder(struct node *);

                     void main()
                     {
                         struct node *q,*root;
                         struct node *maketree(int);
                         int n,ch;
                         clrscr();
                         printf("\n Enter Root of Binary Tree(for exit input -
                     1):\t");
                         scanf("%d",&n);
                         root=maketree(n);
                         p=root;
                         while(TRUE)
                         {
                             printf("\n Enter Another Number:\t");
                             scanf("%d",&n);
                             if(n==-1)
                                 break;
                             p=root;
                             q=root;

                             while(n!=p->data && q!=NULL)
                             {
                                 p=q;
                                 if(n<p->data)
                                     q=p->left;
                                 else
                                     q=p->right;
                             }
                             if(n<p->data)
                             {
                                 setleft(p,n);
                                 printf("\n left branch of:\t%d\n",p->data);
                             else
                             {
                                 setright(p,n);
```

```
          printf("\n right branch of:\t%d\n",p->data);
       }
    }
    printf("\n Binary tree traversal\n");
    printf("\n  1-Preorder,2-Inorder,3-Postorder,4-
Quit\n");
    while(TRUE)
    {
       printf("\n Enter your choice:\t");
       scanf("%d",&ch);
       switch(ch)
       {
          case 1: preorder(struct node *root);
             break:
          case 2: inorder(struct node *root);
             break:
          case 3: postorder(struct node *root);
             break:
          case 4: exit();
       }
    }
}
struct node *maketree(int y)
{
    struct node *r;
    r=(struct node *)malloc(sizeof(struct node));
    if(r==NULL)
       exit(1);
    r->left=NULL;
    r->data=y;
    r->right=NULL;
    return(r);
}
void setleft(int p,int k)
{
    if(r==NULL)
       printf("\n void insertion");
    else if(r->left!=NULL)
       printf("\n Invalid Insertion");
    else
       r->left=maketree(x);
}
void setright(int p,int k)
{
    if(r==NULL)
       printf("\n void insertion");
    else if(r->right!=NULL)
```

```
                    printf("\n Invalid Insertion");
            else
                r->right=maketree(x);
}
void preorder(struct node *r)
{
    if(r!=NULL)
    {
        printf(" %d",r->data);
        preorder(r->left);
        preorder(r->right);
    }
}
void inorder(struct node *r)
{
    if(r!=NULL)
    {
        inorder(r->left);
        printf(" %d",r->data);
        inorder(r->right);
    }
}
void postorder(struct node *r)
{
    if(r!=NULL)
    {
        postorder(r->left);
        postorder(r->right);
        printf(" %d",r->data);
    }
}
```

These methods can be summed up as shown in Table 4.1.

Table 4.1 Methods of Traversing a Binary Tree

Preorder	Display info	Go to left	Go to right
Inorder	Go to left	Display info	Go to right
Postorder	Go to left	Go to right	Display info

4.6 IMPLEMENTATION OF BINARY TREES

There are two traditional popular techniques that are used to maintain binary tree in the memory. These are as follows:

(1) Sequential representation (linear)

(2) Linked list representation (non-linear)

1. Sequential Representation

In this representation, nodes are stored in one-dimensional array and given level-number to every node in tree.

Consider the binary tree shown in Figure 4.8. Here, numbers are assigned for all the nodes. You can assign numbers in such a way that the root is assigned the number 1; a left subtree is assigned twice the number assigned to its root.
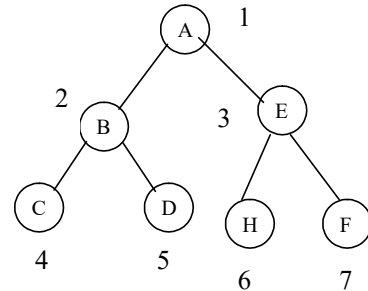
Fig. 4.8 *Binary Tree with Numbers Assigned*

You can keep the nodes of a binary tree in array. The use of root, left or right fields is not necessary. This is because the individual nodes are accessed using their position in the array and the position of the node is calculated on the basis of the arrangement for nodes in the tree. Since the array in *c* starts at position 0, it may assign numbers to the nodes from 0 to 8 instead of 1 to 9. The nodes kept in an array are as follows:
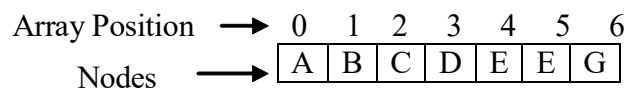
By this conversion, the node in position p is the implicit root of nodes 2p+1 and 2p+2 the left subtree of node p is node 2p+1; right subtree of node p is node 2p+2. For example, node (A) in array position 0 is the root of nodes (B) and (C) in array position 1 and 2. The sequential representation can be extended to general binary trees.

For example, consider the binary tree shown in Figure 4.9.
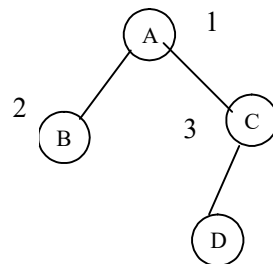
Fig. 4.9 *Sequential Representation of Binary Tree*

The complete binary tree diagram is shown in Figure 4.10.
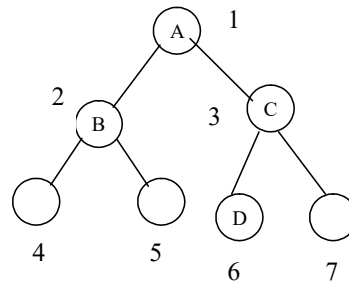
Fig. 4.10 Complete Binary Tree

The node of the binary tree given in Figure 4.10 is stored in the array as follows:

```
Array Position ──→    0  1 2  3 4 5 6
                     ┌─┬─┬─┬─┬─┬─┬─┐
          Nodes ───→ │A│B│C│ │ │D│ │
                     └─┴─┴─┴─┴─┴─┴─┘
```
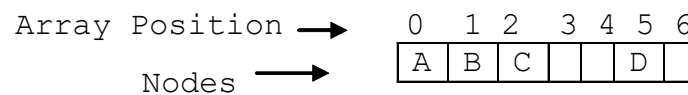
Note that some of the array positions are empty. The following program describes how to create binary tree using an array.

```c
/* creation of a binary tree by using sequential (array)
representation */
    #include<stdio.h>
#include<conio.h>
#define true 1
#define false 0
#define MAX 300

struct node
{
    int data;
    int used;
}x[MAX];

void setleft(int,int);
void setright(int,int);
void maketree(int);

void main()
{
    int q=0,p,n,pos;
    clrscr();
    printf("\n Enter Root of Binary Tree(for exit input -
1):\t");
    scanf("%d",&n);
    if(n==-1)
        exit(1);
```

```
    maketree(n);
    printf("\n This number is at position:\t%d\n\n",q);
    while(true)
    {
        printf("\n Enter Another Number:\t");
        scanf("%d",&n);
        if(n==-1)
            break;
        p=q=0;
                        while(q<MAX && x[q].used &&
n!=x[p].data)
        {
            p=q;
            if(n<x[p].data)
                q=2*p+1;
            else
                q=2*p+2;
        }
        if(n==x[p].data)
            printf("\n %d is a duplicate number\n",n);
        else if(n<x[p].data)
        {
            setleft(p,n);
            printf("\n  Number  is  stored  at
position:\t%d\n",q);
        }
        else
        {
            setright(p,n);
            printf("\n  Number  is  stored  at
position:\t%d\n",q);
        }
    }
    while(true)
    {
        printf("\n To access node enter its array
position");
        printf("\n For exit enter:\t-1\n");
        scanf("%d",&pos);
        if(pos==-1)
            exit(1);
        if(x[pos].used==true)
            printf("The         node         content
is:"\t%d\n",x[pos].data);
        else
```

```
                        printf("Null Tree node\n");
               }
               getch();
}
void maketree(int y)
{
       int p;
       x[0].data=y;
       x[0].used=true;
       for(p=1;p<MAX;p++)
           x[p].used=false;
}
void setleft(int p,int k)
{
       int q;
       q=2*p+1;
       if(q>MAX)
           printf("\n ARRAY IS FULL");
       else if(x[q].used)
           printf("\n Invalid Insertion");
       else
       {
           x[q].data=k;
           x[q].used=true;
       }
}
void setright(int p,int k)
{
       int q;
       q=2*p+2;
       if(q>MAX)
           printf("\n ARRAY IS FULL");
       else if(x[q].used)
           printf("\n Invalid Insertion");
       else
       {
           x[q].data=k;
           x[q].used=true;
       }
}
```

2. Linked List Representation

In linked list representation every node is stored separately and keeps the address of other nodes (Figure 4.11).
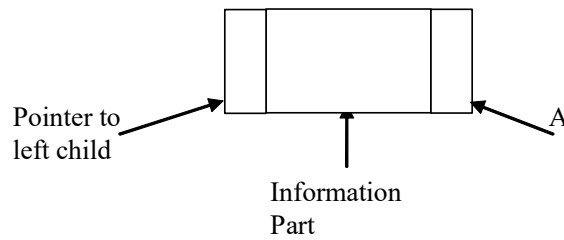
Fig. 4.11 Linked List Representation

In the linked list, you take three structure members for data; the first member is the root pointer; second member is a pointer for left subtree and the third member is also a pointer for the right subtree. The left and right pointers are necessary for binary representation for the same binary tree (Figure 4.12).
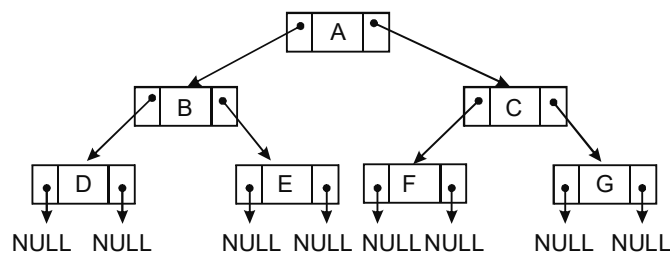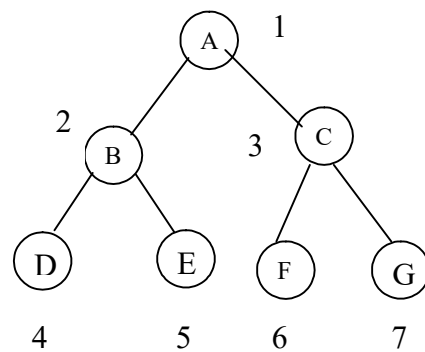
Fig. 4.12 Linked List Representation for Binary Tree

```
struct node
{
            int data;
            struct node *left;
            struct node *right;
}*p;
```

Here p, is a pointer to access tree nodes. p->left as a pointer to the left subtree and p-> right as a pointer to the right subtree.

Under this implementation, the system will take care of allocating and freeing nodes using malloc and free functions from dynamic memory allocation.

```
/* Binary Tree Representation in Linked list*/
#include<stdio.h>
#include<conio.h>
```

```
#define TRUE 1

struct node
{
    int data;
    struct node *left;
    struct node *right;
}*proot,*p,*q;

void setleft(struct node *,int);
void setright(struct node *,int);

void main()
{
    struct node *q,*proot;
    struct node *maketree(int);
    int number;
    clrscr();
    printf("\n Enter Root of Binary Tree(for exit input -
1):\t");
    scanf("%d",&number);
    proot=maketree(number);
    p=proot;
    printf("\n This number is at position:\t%d\n\n",q);
    while(TRUE)
    {
       printf("\n Enter Another Number:\t");
       scanf("%d",&number);
       if(number==-1)
          break;
       p=proot;
       q=proot;

       while(number!=p->data && q!=NULL)
       {
          p=q;
          if(number<p->data)
             q=p->left;
          else
             q=p->right;
       }
       if(number<p->data)
       {
          setleft(p,number);
```

```
                    printf("\n left branch of:\t%d\n",p->data);
             else
             {
                    setright(p,number);
                    printf("\n right branch of:\t%d\n",p->data);
             }
        }
    getch();
}
struct node *maketree(int y)
{
    struct node *r;
    r=(struct node *)malloc(sizeof(struct node));
    if(r==NULL)
        exit(1);
    r->left=NULL;
    r->data=y;
    r->right=NULL;
    return(r);
}
void setleft(int p,int k)
{
    if(r==NULL)
        printf("\n void insertion");
    else if(r->left!=NULL)
        printf("\n Invalid Insertion");
    else
        r->left=maketree(x);
}
void setright(int p,int k)
{
    if(r==NULL)
        printf("\n void insertion");
    else if(r->right!=NULL)
        printf("\n Invalid Insertion");
    else
        r->right=maketree(x);
}
```

4.6.1 Deleting from a Binary Tree

Deletion of nodes from a BST is a little more complicated than inserting a node. This is so, because, deletion of a node that has children needs some other node to be chosen to fill the empty space that the deleted node created. In case, care is not taken while replacing the gap the property of the binary search tree can be violated.

For example, consider the BST in Figure 4.13. In case the node 150 is deleted, some node should be placed in the gap created by node 150's deletion. If you arbitrarily choose to move, say node 92 there, the BST property will be deleted since 92's new left subtree will have nodes 95 and 111, both of which are greater than 92 therefore resulting in the violation of the binary search tree property.

The first step in the algorithm to delete a node is to first locate the node to be deleted. This can be performed with the help of the searching algorithm, and therefore has a $\log_2 n$ running time. Next, a node from the BST must be selected to take the deleted node's position. Three cases need to be considered while the choice for a replacement node is being made, all of which are illustrated in Figure 4.14.
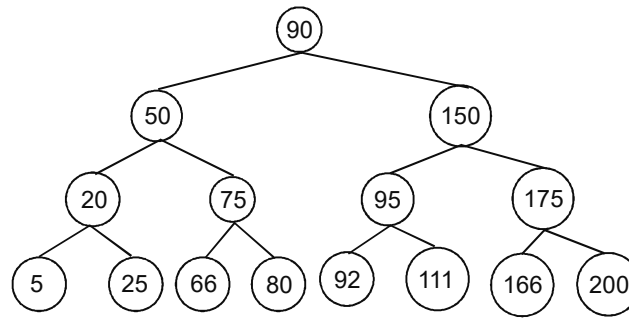
Fig. 4.13 *Deleting Node from a Binary Tree*

Case 1: If the node being deleted has no right child, then the node's left child can be used as the replacement. The binary search tree property is maintained because we know that the deleted node's left subtree itself maintains the binary search tree property, and that the values in the left subtree are all less than or all greater than the deleted node's parent, depending on whether the deleted node is a left or right child. Therefore, replacing the deleted node with its left subtree will maintain the binary search tree property.

Case 2: If the deleted node's right child has no left child, then the deleted node's right child can replace the deleted node. The binary search tree property is maintained because the deleted node's right child is greater than all nodes in the deleted node's left subtree and is either greater than or less than the deleted node's parent, depending on whether the deleted node was a right or left child. Therefore, replacing the deleted node with its right child will maintain the binary search tree property.

Case 3: Finally, if the deleted node's right child does have a left child, then the deleted node needs to be replaced by the deleted node's right child's left-most descendant. That is, we replace the deleted node with the deleted node's right subtree's *smallest* value.

This choice of replacement keeps the binary search tree property since it chooses the smallest node from the right subtree of the deleted node, which is assured to be larger as compared to all the nodes in the deleted node's left subtree.

Besides, since it is the smallest node from the deleted node's right subtree, if you place it at the position of the deleted node, all the nodes in its right subtree will be greater.

Figure 4.14 illustrates the node to choose for replacement for each of the three cases.

Case 1
Imagine we wanted to delete node 50. Since 50 has no right children we could simply repalce it with 20.

Case 2
Imagine we wanted to delete 150. Since 150's right child has no left child, we simply replace 150 with its right child

Case 3
Imagine that we wanted to delete 50. Since 50's right child contains a left child, we need to choose the left-most descendant of 50's right child-this left-most child contains the smallest value in 50's right subtree. This left-most node is 66.

Fig. 4.14 *Three Cases for Replacement Node*

Check Your Progress

4. What is a rooted binary tree also known as?

5. State the main advantage of a complete binary tree.

6. What does tree traversal refer to?

7. When is a tree known as a binary tree?

8. Name the two popular techniques used to maintain binary tree in the memory.

9. What happens in linked list representation?

4.7 GRAPH: DEFINITION AND TERMINOLOGY

A graph denoted by G(V, E) consists of a pair of two non-empty sets V and E, where V is a set of **vertices** (or **nodes**) and E is a set of edges. Each edge is identified with a unique pair of vertices, and is denoted by E = (x, y), where x and y are two vertices. Figure 4.15(a) illustrates a graph in which the set of vertices is {1,2,3,4,5,6} and the set of edges is {(1,2), (2,3), (1,3), (2,5), (3, 5), (3, 4), (4,5) and (4,6)}. This graph is **undirected graph** since there is no direction associated with its edges. In undirected graph, each edge is represented as an unordered pair of vertices, that is (x, y) and (y, x) represent the same edge. For example, in Figure 4.15(a), the pair of vertices (1,2) and (2,1) represent the same edge. On the other hand, if direction is associated with each edge, the graph is known as **directed graph** (also known as **digraph**). In directed graph, each edge is represented by an ordered pair of vertices, that is, <x, y> and <y, x> represent two different edges. For example, Figure 4.15(b) shows a directed graph in which directions are indicated using arrows. Here, the pair of vertices (1, 2) and (2, 1) represents the different edges.
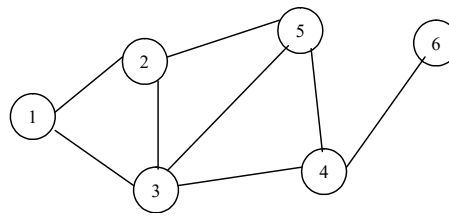
(a) Undirected Graph
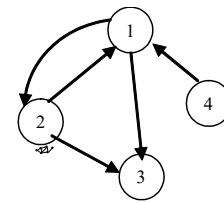
(b) Directed Graph
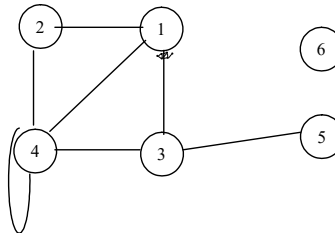
(c) Disconnected Graph

(d) Weighted Graph
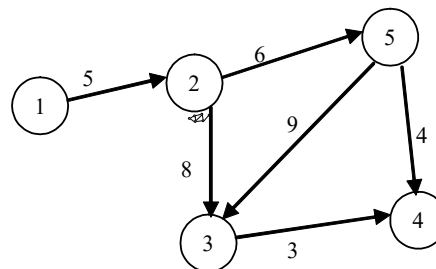
(e) Strongly Connected Graph   *(f) weakly Connected Graph*
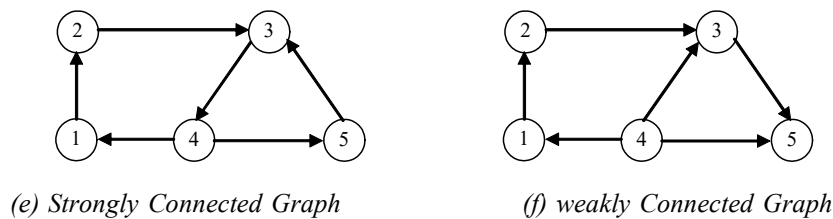
Fig. 4.15 *Examples of Graphs*

Before discussing graph in detail, let us discuss some of the terminologies associated with it.

- **Adjacent vertices:** Two vertices x and y are said to be adjacent, if there is an edge from x to y or from y to x. For example, in Figure 4.15(a), vertices 1 and 3, and 2 and 5 are adjacent, but 1 and 5 are not

- **Degree:** The degree of a vertex x, denoted as degree(x), is the number of edges containing x. For example, in Figure 4.15(a), the degree of vertex 2 is three. Note that the degree(x) can be zero also which represents that vertex x does not belong to any edge. In this case, vertex is called **isolated vertex**. For example, in Figure 4.15(c), vertex 6 is isolated.

- **Path:** A path of length k from vertex x to vertex y is defined as a sequence of k+1 vertices $V_1, V_2, \ldots, V_{k+1}$ such that $V_1=x$, $V_{k+1}=y$ and V_i is adjacent to V_{i+1} for i=1, 2,..,k+1. For example, in graph shown in Figure 4.15(a), there is a path of length 2 from vertex 1 to vertex 5 and a path of 4 from vertex 1 to vertex 6. If $V_1=V_{k+1}$ then path is said to be **closed**. If all the vertices are distinct except V_1 and V_{k+1}, which may be same, the path is said to be **simple**.

- **Cycle:** A cycle is a closed simple path. If a graph contains a cycle, it is called **cyclic graph**; otherwise, **acyclic graph**.

- **Loop:** An edge is a loop if its starting and ending vertex is same. For example, In Figure 4.15(c), edge (4,4) is a loop.

- **Connected graph:** A graph is said to be connected if there is a path between any two of its vertices, that is, there is no isolated vertex. For example, the graph shown in Figure 4.15(a) is connected whereas the graph shown in Figure 4.15(c) is not connected.

- **Weighted graph:** A graph is said to be weighted if a non-negative number (called weight) is associated with each edge. This weight may represent distance between two vertices or the cost of moving along that edge. Figure 4.15(d) illustrates a weighted graph.

Note: A tree is a connected graph without any cycle.

Following are the terminologies that are associated with the directed graphs only.

- **Indegree of a vertex:** Indegree of a vertex x, denoted by indeg(x) refers to the number of edges terminating at x. For example, in directed graph shown in Figure 4.15(b), indeg(2) = 1.

- **Outdegree of a vertex:** Outdegree of a vertex x, denoted by outdeg(x) refers to the number of edges originating from x. For example, in directed graph shown in Figure 4.15(b), the outdeg(2)=2.

- **Strongly connected:** A directed graph is said to be connected or strongly connected if for every pair of vertices <x, y>, there exists a directed path from x to y and from y to x. However, if there exists any pair <x, y>, such that there is a path either from x to y or from y to x, the graph is said to be **weakly connected**. For example the graphs shown in Figure 4.15(e) and 4.15(f) are strongly and weakly connected, respectively.

Note: The concept of path, cycle remains same for directed graphs. The only difference is that the direction of each edge is considered.

4.8   REPRESENTATION OF GRAPHS

Two types of representations can be used to maintain a graph in memory: sequential and linked representations. In sequential representation, a graph is represented by means of adjacency matrix while in linked representation, it is represented by means of adjacency lists. In this section, both these representations are discussed.

Adjacency Matrix Representation

Given a graph `G(V,E)` with n vertices, the adjacency matrix `A` for this graph is a `n x n` matrix such that

    Aij =

        1, if there is an edge from `Vi` to `Vj`   where, 1<=i<=n

        0, if there is no edge   1<=j<=n

This matrix is also known as bit matrix or boolean matrix since it contains only two values 0 and 1. The adjacency matrices for the graphs 4.15(a) and 4.15(b) are shown in Figure 4.16.

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	0	1	0
3	1	1	0	1	1	0
4	0	0	1	0	1	1
5	0	1	1	1	0	0
6	0	0	0	1	0	0

(a) Adjacency Matrix for Graph of Figure 8.1(a)

	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	0	0	0	0
4	1	0	0	0

(b) Adjacency Matrix for Graph of Figure 8.1(b)

Fig. 4.16 *Adjacency Matrix Representation of Graphs*

Observe that the adjacency matrix for an undirected graph is symmetric, that is, the upper and lower triangles of matrix are same. However, it does not always hold true for the directed graph. Note that in the adjacency matrix of directed graph, the total number of 1's tells the number of edges, and the number of 1's in a row tells the outdegree of corresponding vertex.

Although adjacency matrix is a simple way to represent a graph, it needs n2 bits to represent a graph with n vertices. In case of undirected graph, by storing only upper or lower triangle of the matrix, this space can be reduced to half. However, directed graph is not always symmetric, so it takes n2 space for most of the graph problems.

Adjacency Lists

In adjacency list representation, the graph is stored as a linked structure. In this representation, for each vertex, a linked list (known as adjacency list) is maintained that stores its adjacent vertices. That is, for n vertices in a graph, there will be n adjacency lists. Each list has a header node that points to its corresponding adjacency list. The header nodes are stored sequentially to provide easy access to adjacency list for each vertex. The adjacency lists for the graphs 4.15(a) and 4.15(b), are shown in Figure 4.17(a) and 4.17(b) respectively.
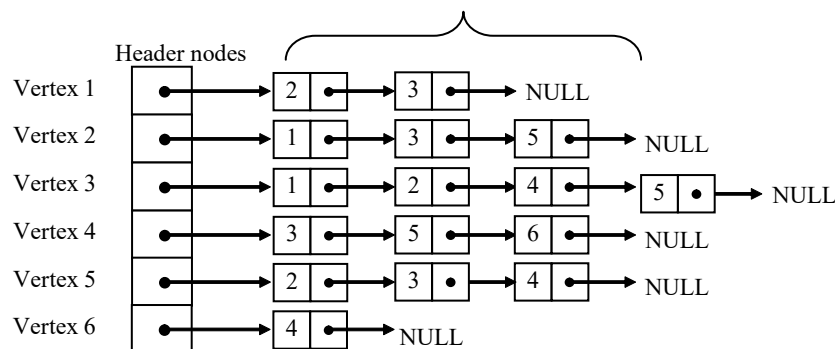
(a) Adjacency List for Graph of Figure 4.17(a)
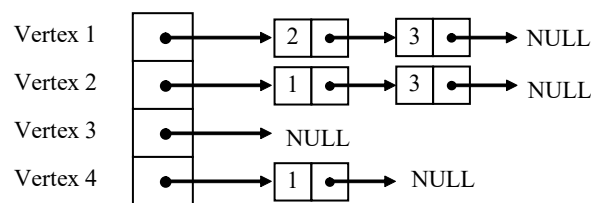
(b) Adjacency List for Graph of Figure 4.17(b)

Fig. 4.17   *Adjacency List Representation of Graphs*

4.8.1   Path Matrix

Given a directed graph G(V,E) with n vertices, the **path matrix** P for this graph is an n × n matrix, such that:

Such that:

$$P_{ij} = \begin{cases} 1, \text{ if there is a path of any length from } V_i \text{ to } V_j \\ 0, \text{ if there is no path from } V_i \text{ to } V_j \end{cases}$$

The path matrix of a given graph G can be computed using its adjacency matrix A. From an adjacency matrix A, it can be determined whether there is an edge between V_i to V_j. Further, the number of paths of any length L from vertex V_i to V_j can be determined from by computing the powers of A.

Now, if matrix A and its powers, A^2, A^3, …, A^L, are added to obtain a matrix BL as follows:

$$BL = A + A^2 + A^3 + … + A^L$$

The number of paths, less than or equal to L, between any two vertices could be determined. In other words, it could be determined whether there exists a path of any length between any two vertices.

By using this matrix B_L, one can compute the path matrix by replacing the nonzero entries of B_L by 1. For example, consider the graph shown in Figure 7.3. The path matrix for this graph can be computed by adding the matrices A, A^2, A^3, and A^4 and replacing the nonzero entries in the resultant matrix by 1 as follows:

$$B_4 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \begin{bmatrix} 5 & 5 & 6 & 3 \\ 5 & 5 & 6 & 3 \\ 3 & 3 & 4 & 3 \\ 3 & 3 & 4 & 3 \end{bmatrix} \end{array}$$

$$P = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

It can be observed from the path matrix P that each node of the given graph is reachable which means that there exists a path between any two nodes.

Check Your Progress

10. Is a graph a linear structure?

11. What does a graph consist of?

12. What is the difference between an undirected graph and a directed graph?

4.9 TRAVERSAL OF GRAPH

One of the common operations that can be performed on graphs is traversing, that is, visiting all vertices that are reachable from a given vertex. The most commonly used methods for traversing a graph are depth-first search and breadth-first search.

Depth-First Search

In depth-first search, starting from any vertex, a single path P of the graph is traversed until a vertex is found whose all-adjacent vertices have already been visited. The search then backtracks on the path P until a vertex with unvisited adjacent vertices is found and then begins traversing a new path P' starting from that vertex and so on. This process continues until all the vertices of the graph are visited. Note that there is always a possibility that a vertex can be traversed more than one time. Thus, it is required to keep track whether the vertex is already visited.

For example, the depth-first search for the graph shown in Figure 4.18(a) results in the sequence of vertices 1, 2, 4, 3, 5, 6 obtained as follows:

1. Visit a vertex, say, 1.

2. Its adjacent vertices are 2, 3, 4 and 6. Select any unvisited vertex, say, 2.

3. Its adjacent vertices are 1 and 4. Since vertex 1 is already visited, select unvisited vertex 4.

4. Its adjacent vertices are 1, 2 and 3. Since vertices 1 and 2 are already visited, select unvisited vertex 3.

5. Its adjacent vertices are 1, 4, 5 and 6. Since vertices 1 and 4 are already visited, select an unvisited vertex, say, 5.

6. Its adjacent vertices are 3 and 6. Select the unvisited vertex 6. Since all the adjacent vertices of vertex 6 are already visited. Backtrack to the visited adjacent vertices to find if there is any unvisited vertex. Since, all the vertices have been visited, algorithm terminates.
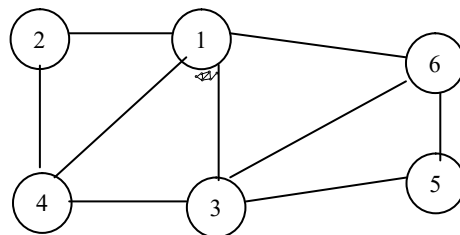
(a) Graph
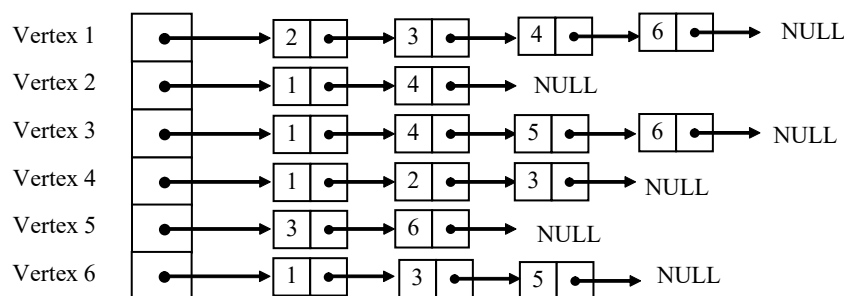
(b) Adjacency list of Graph

Fig. 4.18 *Graph and its Adjacency-list*

To implement depth-first search, an array of pointers `arr_ptr` to store the address of the first vertex in each adjacency list, and a boolean-valued array `visited` to keep track of the visited vertices are maintained. Initially, all the values of `visited` array are initialized to `False` to indicate that no vertex has yet been visited. As soon as a vertex is visited, its value is changed to `True` in the array `visited`. Following is the recursive algorithm for depth-first search.

Algorithm 4.4 Depth First Search (Recursive)

```
void depth_first_ search(v, arr_ptr) //v is the vertex of the graph

1.  Set visited[v] = True    //mark first vertex as visited
2.  Print v
3.  Set ptr = *(arr_ptr+v)
                //assign address of adjacency list of vertex v to ptr
4.  While ptr != NULL
    If visited[ptr->info] = False  //check if vertex is not visited
        Call depth_first_search(ptr->info, arr_ptr)
                            //call depth_first_search
    Else
        ptr = ptr->next;
    End If
    End While
5.  End
```

Program 4.1: A program to illustrate the depth first search algorithm.

```c
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
#define MAX 10

typedef struct node
{
    int info;
    struct node *next;
}Node;

int visited[MAX];      /* global variable; all the values
are initialized to 0 */

void create_graph(Node *[], int);
void input(Node *[], int);
void depth_first_search(int, Node *[]);
void display(Node *[], int);

void main()
{
    Node *arr_ptr[MAX];            /* array of pointers to
node type structures */
    int nvertex;
```

```
    clrscr();
    printf("\nEnter the number of vertices in Graph: ");
    scanf("%d", &nvertex);
    create_graph(arr_ptr, nvertex);
    input(arr_ptr, nvertex);
    printf("\nValues are inputted in the graph");
    display(arr_ptr, nvertex);
    printf("\n\nDepth First Search is:\t");
    depth_first_search(1, arr_ptr);
    getch();
  }


void create_graph(Node *arr_ptr[], int num)      /*  to
create an empty graph, the entire */
  {                            /*  adjacency  list  is
initilialised with NULL */
    int i;
    for(i=1; i<=num; i++)
    arr_ptr[i]=NULL;
  }


  void input(Node *arr_ptr[], int num)
  {
    Node *nptr,*save;
    int i,j,num_vertex,item;
    for(i=1; i<=num; i++)
    {
       printf("Enter the no. of vertices in adjacency
list a[%d] : ",i);
       scanf("%d", &num_vertex);
       for(j=1; j<=num_vertex; j++)
       {
         printf("Enter the value of vertex : ");
         scanf("%d", &item);
         nptr=(Node*)malloc(sizeof(Node));
         nptr->info=item;
         nptr->next=NULL;
         if(arr_ptr[i]==NULL)
            arr_ptr[i]=last=nptr;
         else
         {
            save->next=nptr;
            save=nptr;
         }
```

```
                    }
                 }
              }
```

```
 void display(Node *arr_ptr[], int num)
 {
    int i;
    Node *ptr;
    printf("\n\nGraph is:\n");
    for(i=1;i<=num;i++)
    {
       ptr=arr_ptr[i];
       printf("\na[%d]   ",i);
       while(ptr != NULL)
       {
        printf(" -> %d", ptr->info);
        ptr=ptr->next;
        }
    }
 }


void depth_first_search(int v, Node *arr_ptr[])
{
    Node *ptr;
    visited[v]=True;  /* mark first vertex as visited */
printf("\%d\t", v);
ptr=*(arr_ptr+v);      / * assign address of adjacency
list to ptr */
 while(ptr!=NULL)
    {
       if(visited[ptr->info]==False)
          depth_first_search(ptr->info, arr_ptr);
       else
          ptr=ptr->next;
    }
}
```

The output of the program is:

```
Enter the number of vertices in Graph: 6

Enter the no. of vertices in adjacency list a[1] : 4
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the value of vertex : 4
Enter the value of vertex : 6
```

```
Enter the no. of vertices in adjacency list a[2] : 2
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the no. of vertices in adjacency list a[3] : 4
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the value of vertex : 5
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[4] : 3
Enter the value of vertex : 1
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the no. of vertices in adjacency list a[5] : 2
Enter the value of vertex : 3
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[6] : 3
Enter the value of vertex : 1
Enter the value of vertex : 3
Enter the value of vertex : 5


Values are inputted in the graph


Graph is:

a[1]    -> 2 -> 3 -> 4 -> 6
a[2]    -> 1 -> 4
a[3]    -> 1 -> 4 -> 5 -> 6
a[4]    -> 1 -> 2 -> 3
a[5]    -> 3 -> 6
a[6]    -> 1 -> 3 -> 5


Depth First Search is:  1     2     4     3     5     6
```

Note that the depth-first search can also be implemented non-recursively by using a stack explicitly. In this implementation, all unvisited vertices adjacent to the one being visited are placed onto a stack and then the top element of the stack is popped to find the next vertex to visit. This process is repeated until the stack is empty.

Breadth-First Search

In the breadth-first search, starting from any vertex, all its adjacent vertices are traversed. Then any one of adjacent vertices is selected and all its adjacent vertices that have not been visited yet are traversed. This process continues until all the vertices have been visited. For example, the breadth-first search for the graph shown in Figure 4.18(a) results in sequence 1, 2, 3, 4, 6, 5 which is obtained as follows:

1. Visit a vertex, say, 1.

2. Its adjacent vertices are 2, 3, 4 and 6. Visit all these vertices one by one. Select any vertex, say, 2 and find its adjacent vertices.

3. Since all the adjacent vertices for 2, that is, 1 and 4 are already visited. Select vertex 3.

4. Its adjacent vertices are 1, 5 and 6. Visit 5 as 1 and 6 are already visited.

5. Since all the vertices are visited, the process is terminated.

The implementation of breadth-first search is quite similar to the implementation of depth-first search. The difference is that the former uses a queue `queue` instead of stack (either implicitly via recursion or explicitly) to store the vertices of each level as they are visited. These vertices are then taken one by one and their adjacent vertices are visited and so on until all the vertices have been visited. The algorithm terminates when the queue becomes empty.

Algorithm 4.5 Breadth First Search

```
void breadth_first_search(arr_ptr)

1.  Set v = 1
2.  Set visited[v] = True   //mark first vertex as visited
3.  Print v
4.  call qinsert(v)         //insert this vertex in queue
5.  While isqempty() = False   // check if there is element in queue
        Call qdelete()    // returning an integer value v
        Set ptr=*(arr_ptr+v)
                //assign address of adjacency list to ptr, ptr is a pointer of type node
        While(ptr != NULL)
                If visited[ptr->info] = False
                        Call qinsert(ptr->info)
                        Set visited[ptr->info] = True
                        Print ptr->info
                End If
        End While
        Set ptr = ptr->next
    End While
6.  End
```

Program 4.2: A program to illustrate the breadth first search algorithm.

```c
#include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0

typedef struct node
{
    int info;
    struct node *next;
}Node;

int visited[MAX];
int queue[MAX];
```

```
int Front, Rear;

void create_graph(Node *[], int num);
void input(Node *[], int num);
void breadth_first_search(Node *[]);
void qinsert(int);
int qdelete();
int isqempty();
void display(Node *[], int num);

void main()
{
    Node *arr_ptr[MAX];
    int nvertex;
    clrscr();
    printf("\nEnter the number of vertices in Graph: ");
    scanf("%d", &nvertex);
    create_graph(arr_ptr, nvertex);
    input(arr_ptr, nvertex);
    printf("\nValues are inputted in the graph");
    display(arr_ptr, nvertex);
    Front=Rear=-1;
    breadth_first_search(arr_ptr);
    getch();
  }

 void create_graph(Node *arr_ptr[], int num)
 {
      int i;
      for(i=1; i<=num; i++)
         arr_ptr[i]=NULL;
 }

 void input(Node *arr_ptr[], int num)
 {
    Node *nptr,*save;
    int i,j,num_vertex,item;
    for(i=1; i<=num; i++)
    {
        printf("Enter the no. of vertices in adjacency
list a[%d] : ",i);
        scanf("%d", &num_vertex);
        for(j=1; j<=num_vertex; j++)
        {
```

```
                    printf("Enter the value of vertex : ");
                    scanf("%d", &item);
                    nptr=(Node*)malloc(sizeof(Node));
                    nptr->info=item;
                    nptr->next=NULL;
                    if(arr_ptr[i]==NULL)
                        arr_ptr[i]=save=nptr;
                    else
                    {
                     save->next= nptr;
                     save=nptr;
                    }
                }
            }
        }

 void display(Node *arr_ptr[], int num)
 {
    int i;
    Node *ptr;
    printf("\n\nGraph is:\n");
    for(i=1;i<=num;i++)
    {
        ptr=arr_ptr[i];
        printf("\na[%d]   ",i);
        while(ptr != NULL)
        {
            printf(" -> %d", ptr->info);
            ptr=ptr->next;
        }
    }
 }

void breadth_first_search(Node *arr_ptr[])
{
    Node *ptr;
    int v=1;
    visited[v]=True; /*mark first vertex as visited*/
    printf("\nBreadth First Search: %d\t", v);
    qinsert(v);  /*insert this vertex in queue*/
    while(isqempty()==False)
    {
        v=qdelete();
        ptr=*(arr_ptr+v);  /8assign address of adjacency
```

```
list to ptr*/
     while(ptr != NULL)
     {
         if(visited[ptr->info]==False)
         {
             qinsert(ptr->info);
             visited[ptr->info]=True;
             printf("%d\t", ptr->info);
         }
     ptr=ptr->next;
     }
}
}

void qinsert(int vertex)
{
    if (Rear==6)
    {
       printf("Overflow! Queue is Full");
       exit();
    }
     queue[++Rear]=vertex;
     if (Front==-1)
        Front=0;
 }

int qdelete()
{
    int item;
    if (Front==-1)
    {
       printf("Underflow! Queue is empty");
       exit();
    }
    item=queue[Front];
    if (Front==Rear)
       Front=Rear=-1;
    else
       Front++;
    return item;
}

int isqempty()
{
```

```
                             if (Front==-1)
                                 return True;
                             return False;
```

```
                         }
```

The output of the program is:

```
Enter the number of vertices in Graph: 6
Enter the no. of vertices in adjacency list a[1] : 4
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the value of vertex : 4
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[2] : 2
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the no. of vertices in adjacency list a[3] : 4
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the value of vertex : 5
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[4] : 3
Enter the value of vertex : 1
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the no. of vertices in adjacency list a[5] : 2
Enter the value of vertex : 3
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[6] : 3
Enter the value of vertex : 1
Enter the value of vertex : 3
Enter the value of vertex : 5


Values are inputted in the graph


Graph is:


a[1]    -> 2 -> 3 -> 4 -> 6
a[2]    -> 1 -> 4
a[3]    -> 1 -> 4 -> 5 -> 6
a[4]    -> 1 -> 2 -> 3
a[5]    -> 3 -> 6
a[6]    -> 1 -> 3 -> 5


Breadth First Search: 1   2       3       4       6       5
```

4.9.1 Spanning Trees

The graphs have found application in diverse areas. Various real-life situations like traffic flow, analysis of electrical circuits, finding shortest routes, applications related with computation, etc. can be easily managed by the graphs. Some of the applications of graphs like topological sorting and minimum spanning trees are discussed here.

Topological Sorting

The topological sort of a directed acyclic graph is a linear ordering of the vertices such that if there exists a path from vertex x to y, then x appears before y in the topological sort. Formally, for a directed acyclic graph $G = (V, E)$, where $V = \{V1, V2, V3, . . . , Vn\}$, if there exists a path from any Vi to V_j, then V_i appears before V_j in the topological sort. An acyclic directed graph can have more than one topological sorts. For example, two different topological sorts for the graph illustrated in Figure 4.19(a) are (1, 4, 2, 3) and (1, 2, 4, 3).
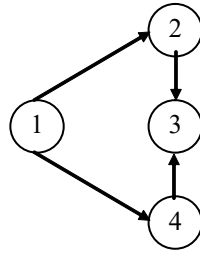
Fig. 4.19 *Acyclic Directed Graph*

Clearly, if a directed graph contains a cycle, topological ordering of vertices is not possible. It is because for any two vertices Vi and V_j in the cycle, Vi precedes Vj as well as Vj precedes Vi. To examplify this, consider a simple cyclic directed graph shown in Figure 4.20. The topological sort for this graph is (1, 2, 3, 4) (assuming the vertex 1 as starting vertex). Now, since there exists a path from the vertex 4 to 1, then according to the definition of topological sort, the vertex 4 must appear before the vertex 1, which contradicts the topological sort generated for this graph. Hence, topological sort can exist only for the acyclic graph.
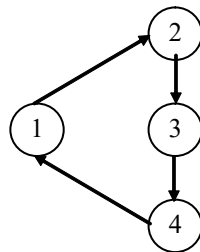
Fig. 4.20 *Cyclic Directed Graph*

In an algorithm to find the topological sort of an acyclic directed graph, the indegree of the vertices is considered. Following are the steps that are repeated until the graph is empty.

1. Select any vertex `Vi` with 0 indegree.
2. Add vertex `Vi` to the topological sort (initially the topological sort is empty).
3. Remove the vertex `Vi` along with its edges from the graph and decrease the indegree of each adjacent vertex of `Vi` by one.

To illustrate this algorithm, consider an acyclic directed graph shown in Figure 4.21.
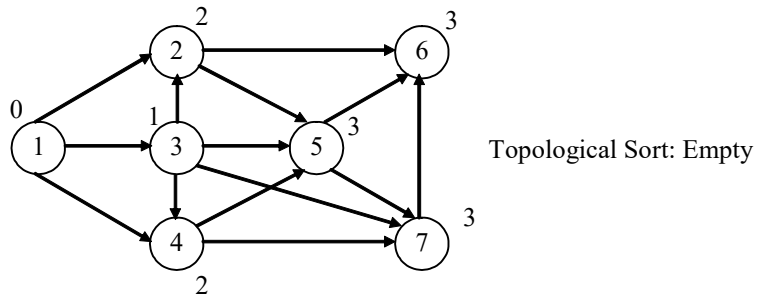
Fig. 4.21 *Acyclic Directed Graph*

The steps for finding topological sort for this graph are shown in Figure 4.22.

(a) Removing vertex 1 with 0 indegree

(b) Removing vertex 3 with 0 indegree

(c) Removing vertex 2 with 0 indegree

Topological Sort: 1, 3, 2, 4

(d) Removing vertex 4 with 0 indegree

Topological Sort: 1, 3, 2, 4, 5

(e) Removing vertex 5 with 0 indegree

Topological Sort: 1, 3, 2, 4, 5, 7   Topological Sort: 1, 3, 2, 4, 5, 7, 6

(f) Removing node 7 with 0 in-degree
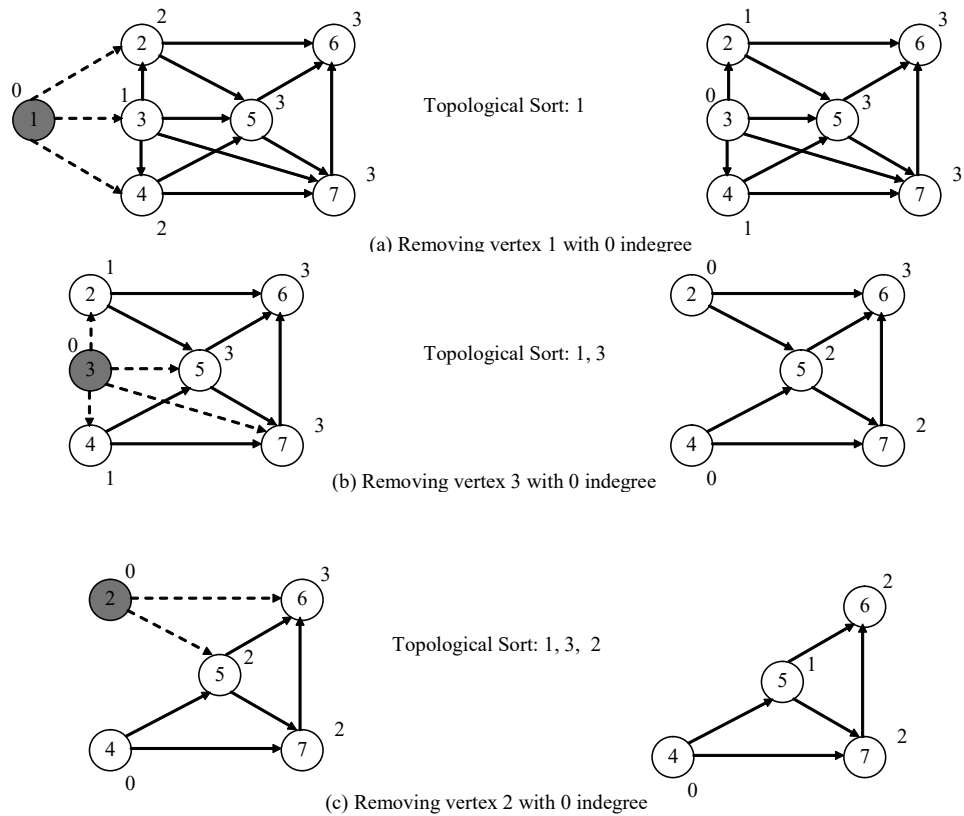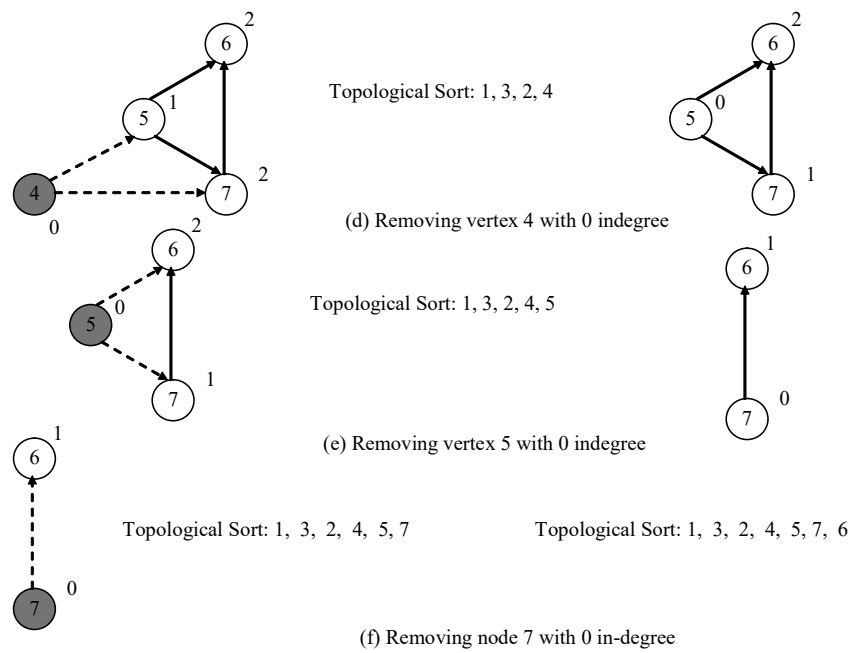
Fig. 4.22 *Steps for Finding Topological Sort*

Another possible topological sort for this graph is (1, 3, 4, 2, 5, 7, 6). Hence, it can be concluded that the topological sort for an acyclic graph is not unique. Topological ordering can be represented graphically. In this representation, edges are also included to justify the ordering of vertices (see Figure 4.23).

(a)

(b)
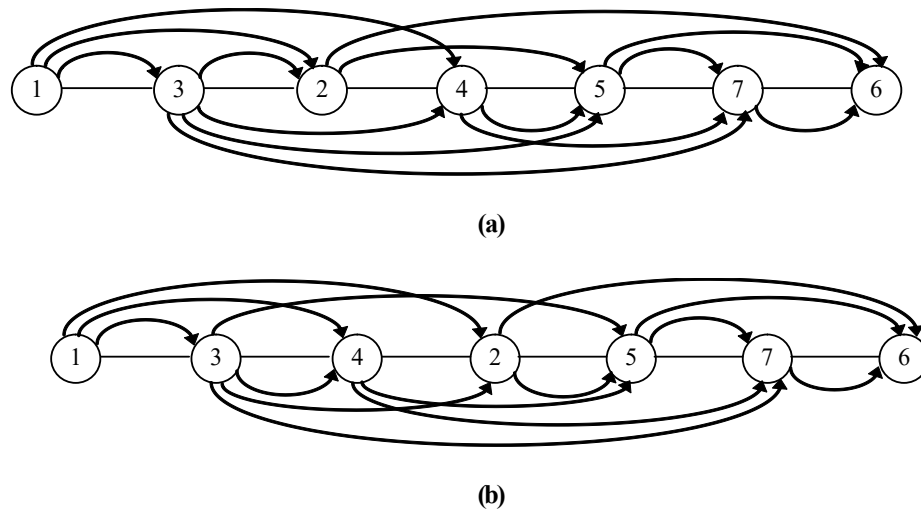
Fig. 4.23   *Graphical Representation of Topological Sort*

Topological sort is useful for the proper scheduling of various sub-tasks to be executed for completing a particular task. In computer field, it is used for scheduling the instructions. For example, consider a task in which a smaller number is to be subtracted from the larger one. The set of instructions for this task is as follows:

```
1.  If A>B then goto step 2, else goto step 3
2.  C = A-B, goto step 4
3.  C = B-A, goto step 4
4.  Print C
5.  End
```

The two possible scheduling orders to accomplish this task are (1, 2, 4, 5) and (1, 3, 4, 5). From this, it can be concluded that the instruction 2 cannot be executed unless instruction 1 is executed before it. Moreover, these instructions are non-repetitive, hence acyclic in nature.

Minimum Spanning Trees

A spanning tree of a connected graph G is a tree that covers all the vertices and the edges required to connect those vertices in the graph. Formally, a tree T is called a spanning tree of a connected graph G if the following two conditions hold.

1. T contains all the vertices of G.

2. All the edges of T are subsets of edges of G.

For a given graph G with n vertices, there can be many spanning trees and each tree will have n-1 edges. For example, consider a graph as shown in Figure 4.24. Since this graph has 4 vertices, each spanning tree must have 4 –1 = 3 edges. Some of the spanning trees for this graph are shown in Figure 4.25. Observe that in spanning trees, there exists only one path between any two vertices and insertion of any other edge in the spanning tree results in a cycle.

Note: The weight of a spanning tree is the sum of the weight of edges in that tree.
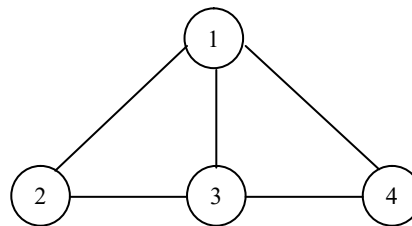
Fig. 4.24 *A Graph G*
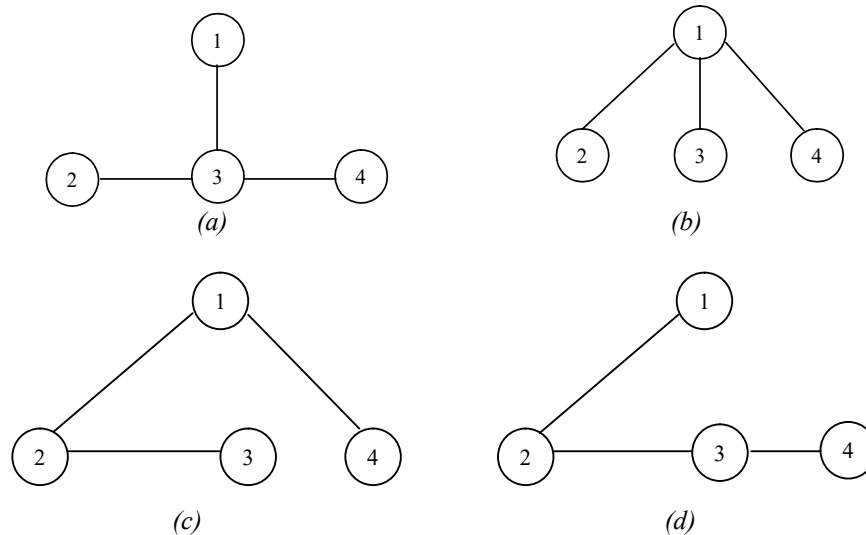
Fig. 4.25 *Spanning Trees of Graph G*

For a connected weighted graph `G`, it is required to construct a spanning tree `T` such that the sum of weights of the edges in `T` must be minimum. Such a tree is called a minimum spanning tree. There are various approaches for constructing a minimum spanning tree. One such approach has been given by Kruskal. In this approach, initially, all the vertices `n` of the graph are considered as distinct partial tree having one vertex. The minimum spanning tree is constructed by repeatedly inserting one edge at a time until exactly `n-1` edges are inserted. The edges are inserted in the increasing order of their costs. Further, an edge is inserted in the tree only if its inclusion does not form a cycle.

Consider an undirected weighted connected graph shown in Figure 8.12. In order to construct the minimum spanning tree `T` for this graph, the edges must be included in the order $(1, 3), (1, 2), (1, 6), (4, 6), (3, 6), (2, 6), (3, 4), (1, 5), (3, 5)$ and $(4, 5)$. This sequence of edges corresponds to the increasing order of weights (3, 4, 5, 6, 7, 8, 9, 12, 14 and 15). The first four edges $(1, 3), (1, 2), (1, 6)$ and $(4, 6)$ are included in `T`. The next edge that is to be inserted in the order of cost is $(3, 6)$. Since, its inclusion in the tree forms a cycle, it is rejected. For the same reason, the edges $(2, 6)$ and $(3, 4)$ are rejected. Finally, on the insertion of the edge $(1, 5)$, `T` has `n-1` edges. Thus, the algorithm terminates and the generated tree is a minimum spanning tree. The weight of this minimum spanning tree is 30. All the steps are shown in Figure 4.27.
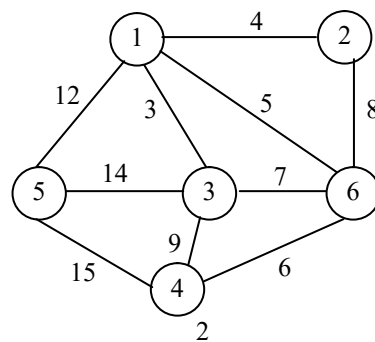
Fig. 4.26 *An Undirected Connected Graph*

(a) Distinct Vertices

(b) Insertion of Edge (1, 3) with Minimum Weight 3

(c) Insertion of Edge (1,2) with weight 4
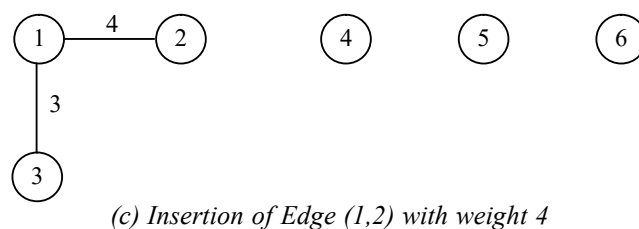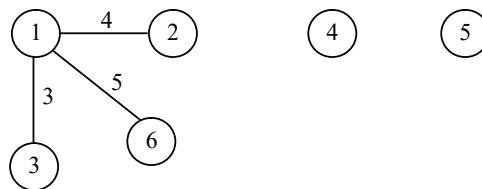
(d) Insertion of Edge (1,6) with weight 5
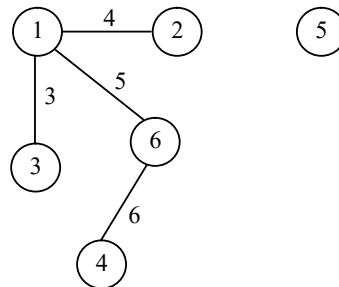
(e) Insertion of Edge (6,4) with weight 6
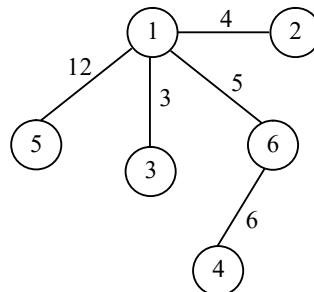
(e) Insertion of Edge (1,5) with weight 12

Fig. 4.27 *Constructing Minimum Spanning Tree*

Minimum spanning trees have many applications such as:

- Analysing the electrical circuits.
- Finding the shortest path

Check Your Progress

13. Which data structures are used to keep track of the nodes to be visited in depth-first and breadth-first search?

14. For which type of graphs is topological sorting defined?

4.10 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The data structure of 'trees' has the components root (for a base), branches (for growth) and leaves (for existence).

2. A path as a component of a tree can be defined as a list of distinctive vertices with its following vertices connected by edges in the tree.

3. The first subset in the binary tree contains a single element called a root of the tree.

4. A rooted binary tree is a rooted tree in which every node has at most two children.

5. The main advantage with the complete binary tree structure is that we can easily find the left and right subtree of node and root of any subtree.

6. Tree traversal refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way.

7. A tree is known as a binary tree, when it is possible for each of its node to have a maximum of two branches (two child nodes).

8. The two traditional, popular techniques that are used to maintain binary tree in the memory are sequential representation (linear), linked list representation (non-linear).

9. In linked list representation, every node is stored separately and keeps the address of other nodes.

10. No, a graph is a non-linear data structure.

11. A graph denoted by G(V, E) consists of a pair of two non-empty sets V and E, where V is a set vertices (or nodes) and E is a set of edges.

12. An undirected graph is a graph in which there is no direction associated with the edges whereas a directed graph (also known as digraph) is a graph in which direction is associated with each edge. In a directed graph, each edge is represented by an ordered pair <x,y> of vertices.

13. Stack and queue are used to keep track of the waiting vertices in depth-first and breadth–first searches, respectively.

14. Topological sorting is defined for directed acyclic graphs.

4.11 SUMMARY

- The data structure of 'trees' consists of root (for a base), branches (for growth) and leaves (for existence). Trees with some or all branches contain no leaves. The roots of the trees will be at the top while the leaves will be at the bottom.

- A node is a structure which may have a value, a condition or represent a separate data structure (which could be a tree of its own).

- The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e. its root path).

- The node at the highest portion of a tree is known as the root node. Since it is the topmost node, the root node will be without any parents. It is the node at which operations on the tree commonly begin.

- A binary tree is a fixed set of elements that is either empty or is divided into three disjoint subsets. The first subset contains a single element called a root of the tree.

- A tree is known as a binary tree if each of its node has a maximum of two branches (two child nodes).

- There are two traditional popular techniques that are used to maintain binary tree in the memory, namely, sequential representation (linear) and linked list representation (non-linear).

- In sequential representation, nodes are stored in one-dimensional array and given level-number to every node in tree.

- In linked-list representation, every node is stored separately and keeps the address of other nodes.

- There are two types of threading in inorder traversal, namely, one way inorder threading and two way inorder threading.

- A graph denoted by G(V, E) consists of a pair of two non-empty sets V and E, where V is a set of vertices (or nodes) and E is a set of edges. Each edge is identified with a unique pair of vertices and is denoted by E = (x, y), where x and y are two vertices.

- An undirected graph is a graph with which no direction associated with its edges. In undirected graph, each edge is represented as an unordered pair of vertices, that is (x, y) and (y, x) represent the same edge.

- A directed graph (also known as digraph) is a graph in which direction is associated with each edge. In directed graph, each edge is represented by an ordered pair of vertices, that is, <x, y> and <y, x> represent two different edges.

- Two types of representations can be used to maintain a graph in memory which are sequential and linked representation. In sequential representation, a graph is represented by means of adjacency matrix while in linked representation, it is represented by means of adjacency lists.

- One of the common operations that can be performed on graphs is traversing, that is, visiting all vertices that are reachable from a given vertex. The most commonly used methods for traversing a graph are depth-first search and breadth-first search.

- The graphs have found application in diverse areas. Various real-life situations like traffic flow, analysis of electrical circuits, finding shortest routes, applications related with computation, etc. can be easily managed by using graphs.

- The topological sort of a directed acyclic graph is a linear ordering of the vertices such that if there exists a path from vertex x to y, then x appears before y in the topological sort.

- A spanning tree of a connected graph G is a tree that covers all the vertices and the edges required to connect those vertices in the graph.

- For a connected weighted graph G, it is required to construct a spanning tree T such that the sum of weights of the edges in T must be minimum. Such a tree is called a minimum spanning tree.

4.12 KEY TERMS

- **Trees**: These are bi-dimensional data structures with hierarchical relationship between data items.

- **Node:** It is a structure that may have a value, a condition or represent a separate data structure (which could be a tree of its own).

- **Binary Tree:** It is a fixed set of elements that is either empty or is divided into three disjoint subsets.

- **Tree Traversal:** It is the process of visiting each node in a tree data structure, exactly once, in a systematic way.

- **Cycle:** A cycle is a closed simple path.

- **Loop:** An edge is a loop if its starting and ending vertices are same.

- **Connected Graph:** A graph is said to be connected if there is a path between any two of its vertices, that is, there is no isolated vertex.

- **Weighted Graph:** A graph is said to be weighted if a non-negative number (called weight) is associated with each edge.

4.13 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Define a tree and list its components.

2. List the methodology of the working of a binary tree.

3. What is the difference between the height and depth of a node?

4. Consider the graph shown in here.
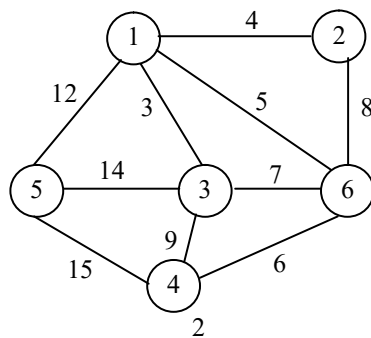
   (i) Find the adjacency matrix for this graph.
   (ii) Find the adjacency list for this graph.

5. How will a graph look like if a row of its adjacency matrix consists of only zeroes?

6. How is a breadth-first search different from a depth-first search? Find the sequence in which the vertices of the following graph will be visited during the breadth-first and the depth-first searches.
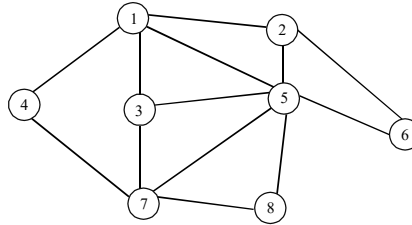
7. Find all possible spanning trees for the following graph.
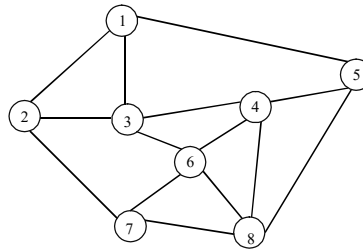
8. Draw the minimal spanning tree for the following graph and also find its weight.
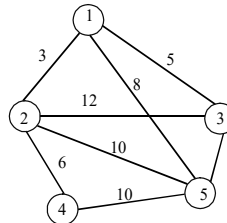
9. Find the topological sort for the following graph.
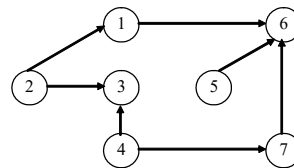
Long-Answer Questions

1. Discuss the theorems associated with binary trees.

2. Write a program that describes how to create a binary tree using an array.

3. Write short notes on
   (i) directed graph,
   (ii) undirected graph.

4. Describe adjacency matrix representation using appropriate diagrams.

5. Explain various applications of graph.

4.14 FURTHER READING

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures.* New Delhi: Galgotia Publications, 2003.

Tanenbaum, A.M., Yedidyah Langsam and Moshe J. Augenstein. *Data Structures using C and C++*. New Delhi: Prentice-Hall of India, 1995.

UNIT 5 HASHING, SEARCHING AND SORTING

Structure

5.0 INTRODUCTION

Hash functions are well-defined procedures or mathematical functions that convert a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array. A hash table is a way of creating a table with operations like Empty, Insert and Retrieve. It offers fast lookup, insertion and deletion of pairs. The basis of a hash table is hash function which takes an element of whatever data type you are storing and outputs an integer in a certain range. Some of the basic terms associated with hash tables are collisions, buckets and slots. In this unit, you will study about the hash table, hashing functions and the basics of bucket handling and overflow. In real life, various things need to be arranged in a particular order so that they can be referred to quickly and easily. For example, the words in the dictionary are arranged in alphabetical order for easy and fast searching of words. In the same way, the large amount of data stored in the computer systems also needs to be organized (sorted) in some logical manner so that individual records can be searched easily and efficiently. If the data is kept in an unordered manner then searching becomes a tedious task. Therefore, sorting and searching are the most basic and commonly performed operations in the computer systems. In this unit, you will also learn about ISAM, searching and sorting.

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the concept of hash table and hashing functions
- Describe the terms associated with hash tables

- Discuss the term bucket overflow and the process of handling it
- Discuss various search techniques
- Discuss different types of sorting
- Analyse various sorting algorithms in terms of their time complexities

5.2 HASH TABLE

A hash table is also known as an associative array or scatter table. It is a simple data structure that offers fast lookup, insertion and deletion of pairs. In this type of data structure, keys are mapped to array positions by a hash function. The easiest way to conceptualize a hash table is to think of it as an array. When a program stores an element in the array, the element key is transformed by a hash function that produces array indexes for that array.

In a well-designed implementation of hash tables, all of these operations have a time complexity of O (1). It means that search times can be independent of the number of entries in a table rather than O (n) which links association lists and vectors. Hashing was first discovered, published and implemented on computers in the early 1950s. It is based on a simple idea of converting the key to a number and then using that number to index a table. It means that the position of a particular entry in the table is determined by the value of the key for that entry. In order to keep the table size manageable, reduce the computed number module to the table size.

You can create a table either on disk or in main memory with the use of a hash table. Hash table can be used for storing records of data, in either an array (when main memory is used) or a file (if disk space is used).
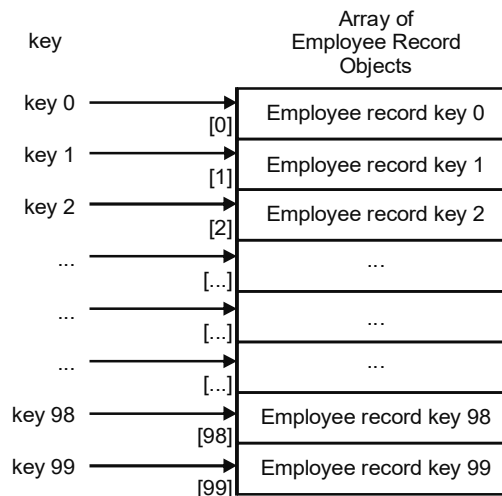
Fig. 5.1 Records on Main Memory

Now, try to understand how hash table works with the help of the following example: you have a list of employees of a small company. Each of the hundred employees has an ID number in the range 0–99. If you store the elements (employee

records) in the array, then each employee's ID number will be an index to the array element where his record is stored.

In this situation, once you know the ID number of the employee, you can directly access his record through the array index. There is a one-to-one correspondence between the element's key and the array's index. However, in practice, this perfect relationship is not easy to establish or to maintain. For example, the same company might use employee's five-digit ID number as the primary key. In this case, key values run from 00,000 to 99,999 and you need to set up an array of size 100,000, of which only 100 elements will be used.
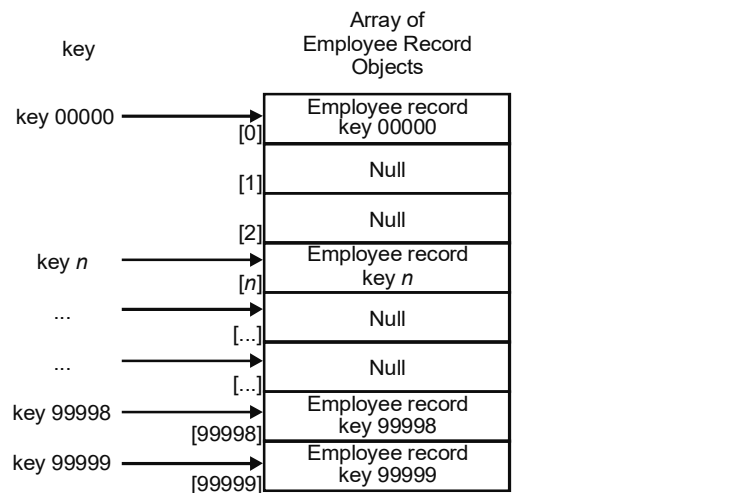
Fig. 5.2 *Array of Employee Record Objects*

It will obviously be very impractical to waste that much storage space just for the purpose of ensuring that each employee' record is safe in a unique and non-predictable location.

But, what if the array size is kept down to the size which is actually used (100 elements) and just the last two digits of the key are used for identifying every employee? For example, the employee with the key number 54,876 will be stored in the element of the array with index 76, and the employee with the key 98,759 will be stored in the element of the array with index 59. In this situation, the elements are not stored according to the value of the key. In fact, the record of the employee with key number 54,876 precedes the record of the employee with key number 98,759, although the value of its key is larger. Moreover, you need a way to convert a five-digit key number to a two-digit array index. You need some functions that will do the conversion. The technique that you use in array is called hash table and the function that is used is called hash function.

Check Your Progress

1. What is a linear search?
2. What is a hash table?
3. What is a hash table used for?

5.3 HASHING FUNCTION

The basis of a hash table is the hash function. A hash function is a function that takes an element of whatever data type you are storing (integer, string, etc.) and outputs an integer in a certain range. This integer is used for determining the location of that element in the table. The hash table itself consists of an array whose indices are the range of the hash function. Each record has a key field and an associated data field and is stored in a location which is based on its key. The function that produces this location for each given key is called hash function.

There are two types of hashing functions as follows:

- Distribution-independent function
- Distribution-dependent function

A distribution independent function means that this function does not use the distribution of the keys of a table in computing the position of a record. It is also known as static hashing because its table has a fixed size. A distribution dependent function refers to examining the subset of keys corresponding to their known records. A hashing function includes speed and generation of addresses regularly. Hash function, h, is a function which transforms a key from a set, K, into an index in a table of size n:

$$h: K \to \{0, 1, ..., n\text{-}2, n\text{-}1\}$$

A key can be a number, a string, a record, alphanumeric characters, etc. They may be arithmetically or logically difficult to manipulate. The process of converting such keys into a more easily manipulative form by using a hashing function is known as preconditioning. The size of the set of keys, |K|, is relatively very large. It is possible for different keys to hash to the same array location. This situation is called collision.

Characteristics of good hash functions are:

- They minimize collisions.
- They are quick and easy to calculate.
- They distribute the key values evenly in the hash table.
- They use all the information provided in the key.

Now, consider an example, for the preconditioning of the key RATE1. Assume that that the possible encode values for the letters are numbers like 11, 12…36 and the set of special symbols like +,-,*,/…are 37, 38, 39… and so on. With the help of this approach the letters RATE1 are encoded as 2811301501. It means that R, A, T, E and 1 are replaced by the integers 28, 11, 30, 15, 01, respectively.

This preconditioning is very efficiently performed by using the numerically coded internal representation of each character in the key. For example, in ASCII the key A1 is binary encoded as 1000001 0110001. This 14-digit binary number is interpreted as 8369 which is equivalent to decimal. Similarly, EBCDIC representation of A1 is binary encoded as 11000001 11110001 or 49,649. So, it

is concluded that the preconditioned result of a key may not fit into a word of memory.

Following are the basic hashing functions:

1. **Division remainder (using the table size as the divisor):** Divison remainder is one of the most widely accepted method in hashing function and it uses % operator to compute the hash value from the key. Its general form is given as follows:

   $H(x) = x \bmod m + 1$

   The identifier x is divided by some number m and the remainder is used as the hash address of x. For example, if $x=35$ and $m=11$(table size) then:

   $H(35) = 35 \bmod(\%) 11 + 1 = 2 + 1 = 3$

The division method yields a 'hash value' which belongs to the set $\{1, 2\ldots m\}$. This method preserves the uniformity that exists in a key set in mapping keys to various addresses. For example, for a divisor m=31, the keys 1,000, 10,001…1,010 are mapped to the addresses 9, 10…and 19. This preservation of uniformity is a disadvantage if two or more clusters are mapped to the same addresses. For example, if another cluster of keys is 2,300, 2,301…and 2,313 then these keys are mapped to addresses 7, 8… and 20 and there are many collisions with keys from the cluster starting at 1,000. That is why the keys in the two clusters yield the same remainder when divided by m=31.

The choice of m is critical. When m is a large prime number at that time it is impossible for a key to yield the same remainder. In practical, it has been found that odd divisors without factors less than 20 are also satisfactory and even divisors are to be avoided since even and odd keys would be mapped to odd and even addresses, respectively.

2. **Mid-square:** This is the easiest and the most widely used method. In it, the key is squared (means it multiplicities by itself) and the middle part of the result is taken as the hash value. Number of bits or digits chosen depends on the table's size and they can fit into one computer word of memory. For example, to map the key 3,121 into a hash table of size 1,000, you square it, i.e., 31,212 = 9,740,641 and extract 406 as the hash value. It works well if the keys do not contain a lot of leading or trailing zeros. Here, non-integer keys have to be preprocessed to obtain corresponding integer values.

3. **Folding:** This technique involves splitting of keys into two or more parts. Each of which has the same length as the required address with the possible exception of the last part and then combining the parts (means added together and ignore finale carry) for forming the hash addresses.

   For example, to map the key 25,936,715 to a range between 0 and 9,999, you can:

   - Split the number into two as 2,593 and 6,715.

   - Add these two to get 9,308 as the hash value.

Two types of transformations are possible as follows:

- Fold shifting method
- Fold boundary method

In the fold shifting method the key is as it is and then it is partitioned into its two or three parts as per the requirement. It involves the reversal of the digits in the outermost partitions. In the earlier example, imagine 3,952 and 5,176 numbers obtain 8,922 as the hash value. It is a hashing function which is useful in converting multiword keys into a single word so that other hashing functions can also be used.

It is very useful if you have keys that are very large. It is very fast and simple especially with bit patterns. A great advantage is its ability to transform non-integer keys into integer values.

4. **Truncation or digit/character extraction:** It forms the addresses by selecting and shifting digits or bits of the original key. It is in a sense distribution dependent. For example, a key 7,546,123 is transformed to the address 2,164 by selecting digits in position 3 to 6 and reversing their order. For a given key set, the same positions in the key and the same rearrangement pattern must be used consistently. Initially, an analysis on a sample of the key set is performed to determine which key positions should be used in forming an address. Its work is based on the distribution of digits or characters in the key.

   In this technique, more evenly distributed digit positions are extracted and used for hashing purposes. For instance, student IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision. It is very fast but digits/characters distribution in keys may not be very even.

5. **Length dependent:** This technique is used for table handling applications. In this method the length of the key is used along with some portion of the key to produce either a table address directly or an intermediate key which is used. For example, with the division method to produce a final table addresses. One function which has produced good results sums the internal binary representation of the first and the last characters and the length of the key shifted left four binary places. Consider one example, the key PARTNO becomes $215+214+(6*16) = 525$ assuming EBCDIC representation. If you treat 525 as an intermediate key and apply the division method with a divisor of 49 then the resulting address is 36.

6. **Radix conversion:** This technique transforms a key into another number base to obtain the hash value. It typically uses the number base other than base 10 and base 2 to calculate the hash addresses. To map the key 55,354 in the range of 0 to 9,999 by using base 11 you have:

$$55354_{10} = 38652_{11}$$

You may truncate the high-order 3 to yield 8,652 as your hash address within 0 to 9,999.

7. **Use of a random-number generator:** In this technique, if you give a seed as a parameter, the method generates a random number.

The algorithm must ensure that:

- It always generates the same random value for a given key.
- No two keys yield the same random value.

The random number produced can be transformed to produce a valid hash value.

Check Your Progress

4. What is a hash function?

5. Name the two types of hashing functions.

6. What are the two considerations for an algorithm?

5.4 TERMS ASSOCIATED WITH HASH TABLES BUCKET OVERFLOW

Collisions, buckets and slots are some of the basic terms associated with hash tables. Ideal hashing means that no two values can be mapped into one table position. For hash function f, each key, k, maps into position f (k). While searching for an element, you compute its expected position: if there is an element present then you operate on it, if not, then the element was not present in the table. If you try to use the same position to store the data for both of them, then you will have a collision.

You will now describe each position in the table as a bucket with position f (k) being the home bucket (the number of buckets in a table equals to the table length). As several keys can map into one bucket, it is possible to design buckets with several slots (this is common for disk-stored hash trees), each slot holding a different value. You will generally have just one slot per bucket, although the linked list approach is different again.

People solve the 'collision' problem on a regular basis while visiting the cinema or sporting event. Like, if you are allocated seat number 47 in a half-full stadium and, on arrival, you discover someone sitting there (but the next seat is free), what would you do? You may ask the user to move…or you could take the next available seat. But, what if a complete stranger was looking for you and knew only your ticket number? If he knew you to be a placid and reasonable person and discovered that seat 47 was otherwise occupied then he/she might keep walking by the higher-numbered seats asking the occupants their ticket numbers until he found you (or discovered that you were not in the stadium, after also checking seats 1… 46).

For example, you have already stored several employees' records, and your table looks some what like this:

Imagine that, your next employee's key is 57,879. Then, your hash function will produce an array index 79. But, the array element with index 79 already has a

value. As the array begins to fill, keys will inevitably produce the same array index when transformed by the hash function. When more than one element tries to occupy the same array position, then you have a collision.

A 'collision' is a condition which occurs when two or more keys produce the same hash location.

A function which generates no collision is known as a 'perfect hash function.'

Perfect hash function is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such functions produce no collisions.
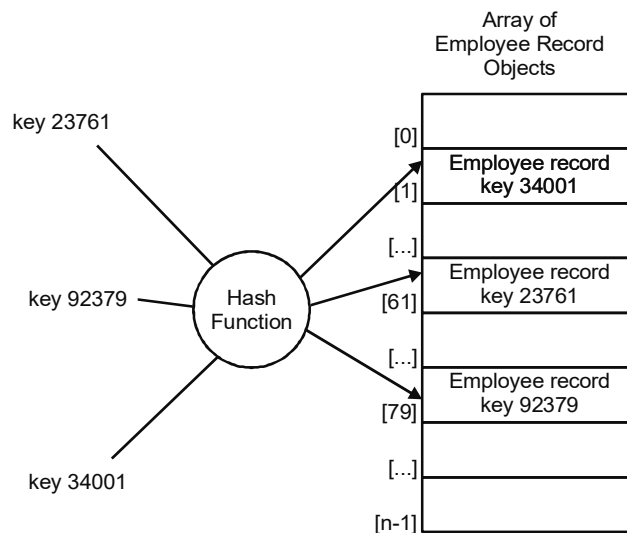
Fig. 5.3 *Hash Function*

Perfect hash function is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such functions produce no collisions.

Overflow Handling (Collision Resolution Techniques)

When collision situation occurs colliding records must be sorted and accessed as determined by a collision-resolution technique. There are two methods for detecting collisions and overflows in a static hash table, but each method uses different data structure for representing the hash table. They are as follows:

- Linear open addressing (linear probing)
- Chaining

Linear Open Addressing

The simplest collision resolution strategy in open addressing is called linear probing. Let us take the same example of cinema/stadium a bit further, what if you sold 1,000 tickets (numbered 0...999) and then discovered that no more than 400 people were going to attend. You might want to move the event to a 500-seat capacity venue. If you take the ticket number (t) and find its value mod 400 ($t\%$ 400) then you will have a new ticket number (n) in the range 0...399. Up to three people (e.g., tickets 73, 473 and 873) may be given the same value of n but they

will still find a seat somewhere by using linear probing. A simple numerical example would be where you wished to insert data with keys 28, 19, 59, 68, 89 into a table of size 10.

These entries are made in the table with the following decisions, as shown in Figure 5.4.

(i) Enter $28 \to 28 \% 10 = 8$. Position is free, 28 placed in element 8

(ii) Enter $19 \to 19 \% 10 = 9$. Position is free, 19 placed in element 9

(iii) Enter $59 \to 59 \% 10 = 9$. Position is occupied, try next place in table (using wraparound), i.e., 0, 59 placed in element 0

(iv) Enter $68 \to 68 \% 10 = 8$. Position is occupied, try next place in table (0) – occupied, try next (1), 68 placed in element 1

(v) Enter $89 \to 89 \% 10 = 9$. Position is occupied, next two places in table are occupied, try next (2), 89 placed in element 2.

Linear probing

Fig. 5.4 *Linear Probing*

Finding and Deleting Data

Finding data is not very difficult. A particular key, k, is given, you look for an element in the home bucket, f (k). If the data is not there then you keep looking further up the table (by using wraparound) until you find it or have ended up back at the home bucket (i.e., the data is not in the table).

Deleting the data is more difficult and trickier. By having first finding the data as described, you cannot simply reinitialize the table element. Reordering the table is an expensive option. To avoid this problem, a process of lazy deletion may be used whereby each element of the array has an additional boolean data member which may be marked used or unused.

The component type of the array is a structure that contains at least a key field. Since, the keys are usually words, you use a string to denote them. Creating the hash table with one slot per bucket is as follows:

```
#define  MAX_CHAR  10 /* max number of characters in
an identifier */
#define  TABLE_SIZE  13 /*max table size = prime
number */
struct  element
```

```
{
 char  key[MAX_CHAR];  /*   other fields   */
};
element  hash_table[TABLE_SIZE];
```

Before inserting any elements into this table, you must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when you insert the elements into the table. The obvious choice for an empty slot is the empty string since it will never have a valid key in any application.

Initialization of a hash table is as follows:

```
void  init_table ( element  ht[ ] )
{
 short  i;
 for ( i = 0; i < TABLE_SIZE; i ++ )
    ht [ i ].key[0] = NULL;
}
```

To insert a new element into the hash table you convert the key field into a natural number, and then apply one of the hash functions. You can transform a key into a number if you convert each character into a number and then add these numbers together. The function transform (below) uses this simplistic approach. For finding the hash address of the transformed key, hash (below) uses the division method.

```
short   transform (char   *key )
{/* simple additive approach to create a natural number
that is within the integer range */
 short   number = 0;
 while (*key)
    number += *key++;
 return   number;
}
short   hash (char   *key)
{
/*transform key to a natural number, and return this
result modulus the table size */
return  (transform (key) % TABLE_SIZE);
}
```

For implementing the linear probing strategy, you first compute f (x) for identifier x and them examine the hash table buckets ht[(f(x) + j) % TABLE_SIZE], $0 \leq j \leq$ TABLE_SIZE in this order. Four outcomes result from the examination of a hash table bucket as follows:

- The bucket contains x. In this case, x is already in the table. Depending on the application, you may either simply report a duplicate identifier, or you may update the information in the other fields of the element.

- The bucket contains the empty string. In this case, the bucket is empty, and you may insert a new element into it.

- The bucket contains a non-empty string other than x. In this case you proceed to examine the next bucket.

- You return to the home bucket ht [f (x)] (j = TABLE_SIZE). In this case, the home bucket is being examined for the second time and all the remaining buckets have been examined. The table is full and you report an error condition and exit.

Implementation of the Insertion Strategy

void linear_insert (element item, element ht [ ])

```
{  /*  insert the key into the table using the linear
probing technique, exit the function if the table is full
*/
    short  i, hash_value;
    hash_value = hash (item.key);
    i = hash_value;
    while (strlen (ht [i].key)
    {
       if (! strcmp (ht [i].key, item.key)
       {
          printf ( "Duplicate entry !\n");
          exit (1);
       }
       i = (i+1) % TABLE_SIZE;
       if (i = = hash_value)
       {
          printf ("The table is full !\n");
          exit (1);
       }
    }
   ht [i] = item;
}
```

In particular, you may wish to minimize the effects of clustering.

Chaining

The term slots allow several keys to map into the same bucket. These are commonly used in disk storage and they can be implemented by using arrays, as long as dynamic array expansion (i.e., array doubling) is allowed. Take the example where you were trying to place the following ten values into a hash table with divisor 13: 42, 39, 57, 3, 18, 5, 67, 13, 70 and 26.

With the linear probing, this would result in the table shown in Figure 5.5.

0	1	2	3	4	5	6	7	8	9	10	11	12
39	26	67	42	<u>3</u>	57	<u>18</u>	<u>5</u>	<u>13</u>	<u>70</u>			

Table using Linear Probing
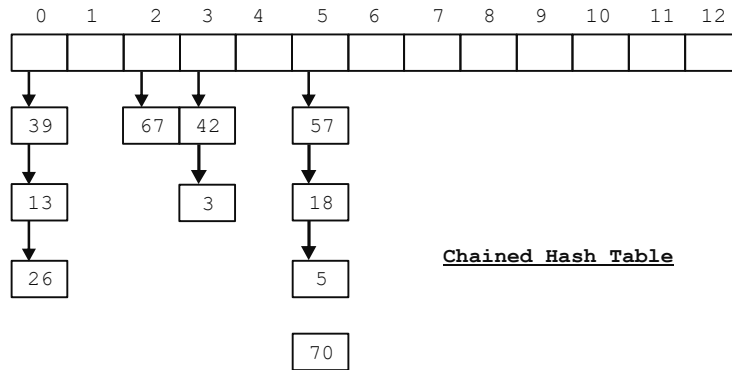
Chained Hash Table

Fig. 5.5 *Table Using Linear Probing and Chained Hash Table*

Now, instead, you will make each bucket simply the starting point of a linked list (initialized to null). Strictly, this means zero slots but effectively you can have a large and a variable number of slots. The first element placed there is put in a node pointed to by the bucket. Each time you have a collision at that bucket, you simply add another node, where elements underlined have been placed outside their home bucket.

A little more reasonable way of implementing this would be to have the list sorted by key values by using structures. To find an element with key k, you would calculate the home bucket (f(k) = k % d) and then search along the chain until you find the element.

Linear probing and its variations perform very badly as inserting an identifier requires the comparison of identifiers with different hash values. To insert a new element you would only have to compute the hash address f (x) and examine the identifiers in the list for f (x). As, you do not know have any idea about the size of the lists in advance, you should think them of them as linked chains. You now require additional space for a link field. Since, you will have M lists, where M is the desired table size, you employ a head node for each chain. These head nodes only need a link field, so they are smaller than the other nodes. You maintain the head nodes in ascending order, 0… M-1 so that you may access the lists at random. The C++ declarations that are required to create the chained hash table are as follows:

```
#define   MAX_CHAR   10 /* maximum identifier size*/
#define   TABLE_SIZE   13 /* prime number */
#define   IS_FULL (ptr) (!(ptr))
struct element
{
    char   key[MAX_CHAR];
    /* other fields */
};
typedef   struct   list   *lis_pointer;
```

```
struct   list
{
   element   item;
   list_pointer   link;
};
list_pointer   hash_table[TABLE_SIZE];
```

The function chain_insert implements the chaining strategy. The function first computes the hash address for the identifier. It then examines the identifiers in the list for the selected bucket. If the identifier is found, you print an error message and exit. If the identifier is not in the list, you insert it at the end of the list. If the list was empty, you change the head node to point to the new entry.

Implementation of the function chain_insert:

```
void   chain_insert (element   item,   list_pointer
ht[])
{ /* insert the key into the table using chaining */
  short   hash_value = hash (item.key);
  list_pointer   ptr, trail = NULL, lead = ht [hash_value];
  for( ; lead; trail = lead, lead = lead->link)
  {
     if (!strcmp(lead->item.key, item.key))
     {
        Printf( "The key is in the table \n");
        exit (1);
     }
  }
  ptr = new struct list;
  if (IS_FULL (ptr))
  {
     Printf( "The memory is full \n");
     exit (1);
  }
  ptr->item = item;
  ptr->link = NULL;
  if (trail)
     trail->link = ptr;
  else
     ht [hash_value] = ptr;
}
```

Clustering

Going back to the same stadium example, where you switched to a venue with a capacity of 500 as only 400 people were expected. It could get even trickier if the number turning up is 40 and you have to move to a venue that has a capacity of 50. Up to 20 people could be seeking the same seat and 19 of them would need to find an alternative seat (i.e. the next higher numbered one which is free). A

situation often arises where several keys map into the same home bucket and, through linear probing, have to seek another higher- numbered bucket. The resultant bunching together of elements is one example of what is called clustering and you would wish to avoid the multiple collisions that occur due to it– especially as one cluster easily merges with another cluster.

Secondary Clustering

As an exercise, consider entering the values 3, 13, 23, 33 and 43 into Figure 11.6 by using linear probing and then quadric probing. Although, the places where collisions will differ, at those places the amount of collisions will be the same. This process is termed as 'secondary clustering.' Fortunately, simulations show that the average cost of this effect is only about half of an extra probe for each search and then only for high load factors. One method of resolving secondary clustering is 'double hashing' which calculates the distance away of the next probe by using another hash function (instead of the linear `+1` or quadratic `+i2`).

The only major drawback of quadratic hashing is the need to ensure that the table is no more than half full (i.e., $l < 0.5$). This is usually a reasonable price to pay.

Choice of Divisor

Not all hashing techniques result in an even distribution of keys and it is quite common to have an imbalance between odd and even keys. If you are using the division-remainder method with $f(k) = k\% b$ then consider the following:

If b is even and there are more even than odd values of k then $f(k)$ will produce an excess of even values. Similarly, more odd than even values of k would produce an excess of odd results for $f(k)$.

If b is odd then either kind of excess of k values would still give a balanced distribution of even/odd results.

Clearly, the divisor should be an odd number. Also, if b itself is divisible by a small odd number (e.g., 3, 5, 7) then the results are less well distributed. Ideally, b should be a prime number but, if no such prime is available near your table size then you should use an odd number which has no small factors (in the range, say, of 3…19).

The type hash entry is yet to be defined elsewhere but it will contain a key, an object and a field which is used for lazy deletion. Similar methods are needed for implementing `search`, `get` and `put` operations.

Analysis of Linear Probing

Assume that you are using a large hash table and that each probe (`search, get` or `put`) is independent from the earlier one.

The proportion of the table which is full is called a load factor, usually represented by the symbol ë (lambda). So, a 40 element table holding only 10 elements has a ë of 0.25–which is also the probability of finding an occupied cell. Therefore, the probability of finding an empty cell during a probe is (1–l) and the expected number of probes needed for finding an empty element is 1/(1 - l).

For example, if your table is a fifth full then you would expect an average of 1.25 probes (= 1/(1-0.20)) to find an empty element. If it is nine-tenths full, then you would expect ten probes on an average. But, this is not true. Clustering means that one failed insertion in a bucket (causing an insertion further up the table) has a knock-on effect on other attempted insertions. Knuth2 has shown that the actual number of probes needed is as follows:

$$(1 + 1/(1 - l)2)/2)$$

By recalculating the above given examples, you have the following:

- For a fifth-full table, you expect an average of 1.28 probes (not 1.25, but close)

- For a half-full table, you expect an average of 2.5 probes (not 2.0, but close)

- For a nine-tenths full table, you expect an average of 50.5 probes (nothing like the 10.0 probes that were predicted earlier)

Most of these problems occur due to clustering. The type which you have studied about till now is actually called primary clustering. You can alleviate many of its problems by moving from linear probing to quadratic probing.

Quadratic Probing

Having calculated your bucket with $f(k) = k \% b$ then, with linear probing, you check position $f(k) + 1$ for availability, then $f(k) + 2$ and so on, until you find a free position.

Quadratic probing uses the search $f(k) + 12$, then $f(k) + 22$, then $f(k) + 32$, etc. (hence the term 'quadratic'). Stepping on distances of 1, 4, 9… rather than 1, 2, 3…greatly reduces the incidence of primary clustering.

Let us try a numerical example similar to that of linear probing. Figure 5.6 shows where the values end up with quadratic probing and the decisions involved are as follows:

Fig. 5.6 Quadratic Probing

(i) Enter 28→28% 10 = 8, position is free, 28 placed in element 8 (result is same as before)

(ii) Enter 19→19% 10 = 9, position is free, 19 placed in element 9 (result is same as before)

(iii) Enter 59→59% 10 = 9, position is occupied, try 1 (=12) places along in table (using wraparound), i.e., 0, 59 placed in element 0 (result is same as before)

(iv) Enter 68→68% 10 = 8, position is occupied, try next place in table (0) – occupied, try next (1), 68 placed in element 1 (result is same as before)

(v) Enter 89→89% 10 = 9, position is occupied, previously you tried positions 0 and 1 then put data in 2. This time, you try 0 then go directly to 3, jumping over the existing cluster.

Effectiveness of Quadratic Probing

This small example–that saves only one collision–might not look like a proof of effectiveness but a full statistical analysis of quadratic probing shows its benefits over linear probing.

It can be proven that, if the table is less than half full and its size is prime, then quadratic probing guarantees that a new element can always be inserted into the table and no cell is probed twice in the process.

The fact that you are adding squares to location numbers instead of just incrementing them might suggest that there is a much larger computational cost from squaring.

This is not the case, as simpler processes can be used.

In a table of size **S**, if your home bucket is called **P0** then your **i**-th probe after that will be as follows:

`Pi = P0 + i2 (MOD S)` and the one prior to that will be:

`Pi-1 = P0 + (i-1)2 (MOD S)`

Subtracting the second equation from the first gives:

`Pi - Pi-1 = i2 - (i-1)2 = (2i-1) (MOD S)`

Therefore,

`Pi = Pi-1 + (2i-1) (MOD S)`

Now, you are dealing with multiplication by 2 and adding 1. Multiplication by 2 is a simple shift-left of a binary number and an increment is a very fast operation for a microprocessor. The (MOD S) is still needed for dealing with wraparound, but it is simply dealt with by subtracting S as and when the element index overshoots the table's last element.

Rehashing

Multiple pseudorandom probes generated by a sequence of transformations are used either to locate a free slot for an overflow record or to find that record for subsequent retrieval. As this method generates random probes, it can have the greatest access time if records are stored on a secondary storage device such as a disk. The non locality of the random accesses can result in large rotational delays.

In general, the problem is that as more and more overflows occur, more records are not where the hash function indicates they should be. Searching for those records can increase the access time beyond the ideal O (l), possibly to the undesirable O (n), if an open addressing method is employed, where n is the total number of buckets used.

External Hashing

Hashing for disk files is called external hashing. For good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a threshold.

It aims to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only one bucket.

The objective of external hashing is to provide acceptable hash table performance on a per operation basis. It has h(k, p)–the integer formed by the p least significant bits of h(k).

Several dynamic hashing schemes for external files have been developed over the last few years. These schemes allow the file size to grow and shrink gracefully according to the number of records actually stored in the file. Any one of the schemes can be used for internal hash tables as well. However, the two methods best suited for internal tables are linear hashing and spiral storage. They are easy to implement and use very less storage space.

The performance is also matched with more traditional solutions for handling dynamic key sets. Both the methods are bound to have efficient techniques for applications where the cardinality of the key set is not known in advance. Amongst the two, linear hashing is faster and also easier to implement. An inherent characteristic of hashing techniques is that a higher load on the table increases the cost of all basic operations: insertion, retrieval and deletion. If the performance of a hash table is to remain within the acceptable limits when the number of records increases, additional storage must somehow be allocated to the table. The traditional solution is to create a new, larger hash table and rehash all the records into the new table. The details of how and when this is done can vary. Linear hashing and spiral storage allows a smooth growth. As the number of records increase, the table grows gradually, one bucket at a time. When a new bucket is added to the address space, a limited local reorganization is performed. There is never any total reorganization of the table.

Linear Hashing

Linear hashing was developed by W. Litwin in 1980. The original scheme was intended for external files. However, for internal hash tables their more complicated address calculation is likely to outweigh their benefits.

Imagine, a hash table consisting of N buckets with addresses 0, 1. . . N–1. Linear hashing increases the address space slowly by splitting the buckets in a predetermined order: first bucket 0, then bucket 1 and so on, up to and including bucket N–1. Splitting a bucket involves moving approximately half of the records from the bucket to a new bucket at the end of the table. The splitting process is illustrated in Figure for an example file with N = 5. A pointer p keeps the track of

the next bucket to be split. When all N buckets have been split and the table size has doubled to 2N, the pointer is reset to zero and the splitting process starts over again. This time the pointer travels from 0 to 2N–1, doubling the table size to 4N. This expansion process can continue as long as required.
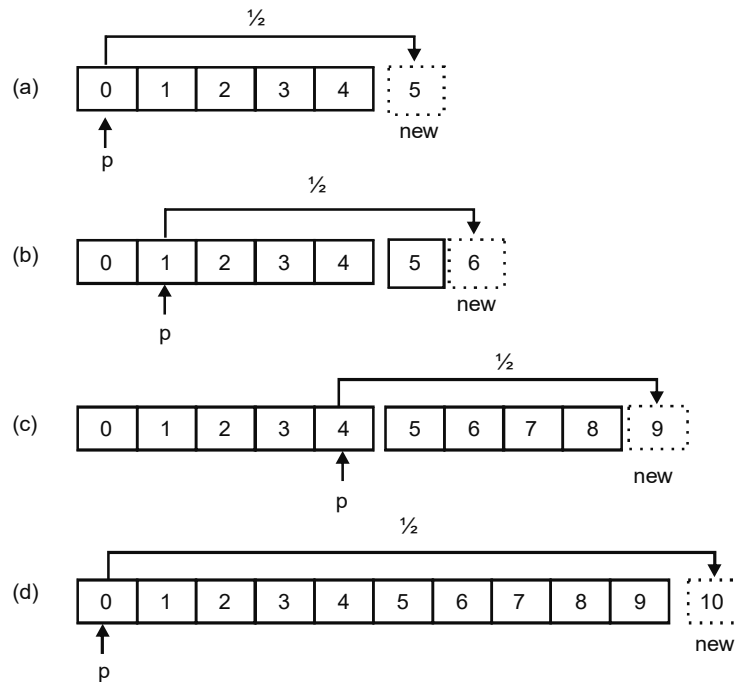
***Fig. 5.7** Splitting of Bucket 0*

Figure 5.7 illustrates the splitting of bucket 0 for an example table with N = 5. Each entry in the hash table contains a single pointer, which is the head of a link containing all the records hashing to that address. When the table is of size 5, all the records are hashed by ho(K) = K mod 5. When the table size has doubled to 10, all the records will be addressed by hi(K) = K mod 10. However, instead of doubling the table size immediately, you expand the table one bucket at a time as required. Consider the keys hashing to bucket 0 under ho(K) = K mod 5. To hash to 0 under ho(K) = K mod 5, the last digit of the key must be either 0 or 5. Under the hashing function h(K) = K mod 10, keys with a last digit of 0 still hash to bucket 0, while those with last digit of 5 hash to bucket 5. Notice that none of the keys hashing to buckets 1, 2, 3 or 4 under ho(K) can possibly hash to bucket 5 under h1. Hence, to expand the table, you allocate a new bucket (with address 5) at the end of the table, increase the pointer p by one, and scan through the records of bucket 0, relocating to the new bucket those hashing to 5 under h(K) = K mod 10.

The current address of a record can be found as follows:

Given a key K, you first compute ho(K). If ho(K) is less than the current value of p, the corresponding bucket has already been split, otherwise not. If the bucket has been split, the correct address of the record is given by hi(K). Note that when all the original buckets (buckets 0–4) have been split and the table size has increased to 10, all records are addressed by h_1.

Returning to the general case, the address computation can be implemented in several ways. For internal hash tables the solution shown in Figure 5.8 appears

to be the simplest. Let g be a normal hashing function producing addresses in some interval [0, M]. M should be sufficiently large, say $M > 2^{20}$. To compute the address of a record you use the following sequence of hashing functions:

```
h,(K) =g(K)mod[N x 2j), j = 0, 1, . . .
```

Hash functions: $h_0(K) = K \bmod 5$
 $h_1(K) = K \bmod 10$
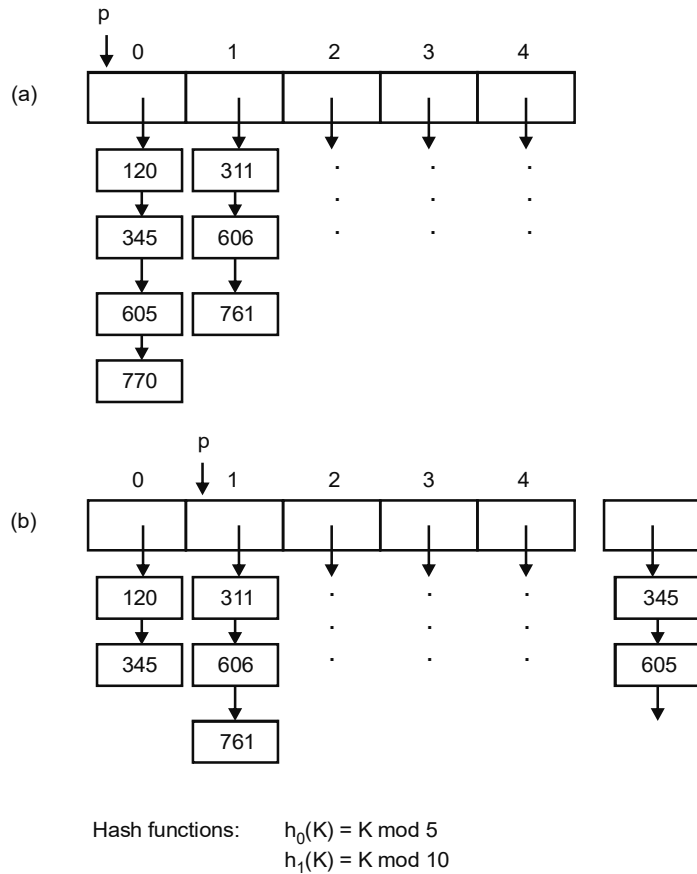
Fig. 5.8 *Normal Hashing Functions*

Where N is the minimum size of the hash table. If N is of the form 2^k, the modulo operation reduces to extracting the last `k + j` bits of `g(K)`. The hashing function `g(K)` can also be implemented in several ways. Functions of the type

```
g(K) = (cK)mod M
```

where c is a constant and M is a large prime. They have been found to perform well experimentally.

Different hashing functions are easily achieved by choosing different values for c and M, thus providing a 'tuning' capability. By choosing c = 1, the classical division remainder function is obtained. You must also keep the track of the current state of the hash table. This can be done with the help of two variables.

Different hashing functions are easily obtained by choosing different values for c and M, thus providing a 'tuning' capability. By choosing c = 1, the classical division remainder function is obtained. You must also keep track of the current state of the hash table. This can be done by two variables:

- L: Number of times the table size has doubled (from its minimum size N), L >= 0

- P: Pointer to the next bucket to be split, $0 <= p < N \times 2^L$

The key idea is to keep the overall load factor bounded. The overall load factor is defined as the number of records in the table divided by the (current) number of buckets. In this case, the overall load factor equals the average chain length. You fix a lower and an upper bound on the overall load factor and expand (contract) the table whenever the overall load factor goes above (below) the upper (lower) bound. This requires that you keep track of the current number of records in the table, in addition to the state variables L and p.

Spiral Storage

While using linear hashing the expected cost of retrieving, inserting or deleting a record varies cyclically. Spiral storage overcomes this undesirable feature and exhibits uniform performance regardless of the table's size. In other words, the table may grow or shrink by any factor but the expected performance always remains the same. Spiral storage intentionally distributes the records unevenly over the table. The load is high at the beginning of the (active) address space and tapers off towards the end as shown in Figure 5.9. To expand the table additional space is allocated at the end of the address space, and at the same time a smaller amount of space is freed at the beginning. The records stored in the bucket that disappear are distributed over the new buckets.

Address space
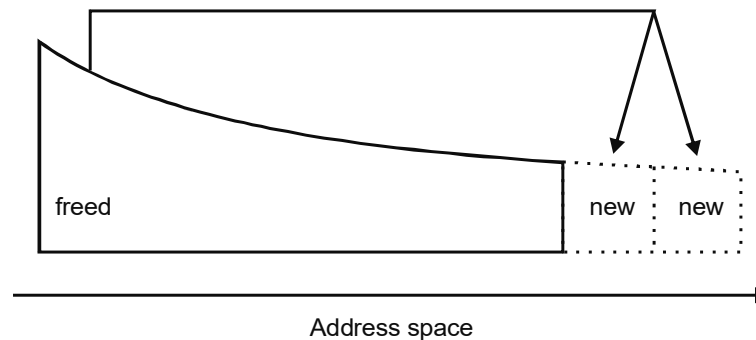
Illustrating the Load Distribution and the Expansion Process of Spiral Storage

Table 5.9 *Load Distribution and Expansion Process*

Check Your Progress

7. What is a collision?

8. Name the methods for detecting collisions and overflows in a static hash table.

9. Which is the only major drawback of quadratic hashing?

5.5 HANDLING BUCKET OVERFLOWS

A trie, an ordered tree data structure, is a tree in which branching is calculated not by the entire key value but only by a portion of it. In addition, branching is based on consideration of that key alone, not on the comparison of a search key with a key stored inside the node. The keys can be from any character or number set. Any other key representation can be easily converted to binary. Let the key under consideration be a binary number with `n` bits. The bits may be chosen in any order, but for simplicity you will let the branching at level `i` be determined by the ith most significant bit. As an example consider a key with three bits. Figure 5.10 shows a trie that addresses three pages A, B and C. Records beginning with the bit sequence 00... are placed in page A, those with bit sequence 01…are placed in page B and those with bit sequence 1... are placed in page C.
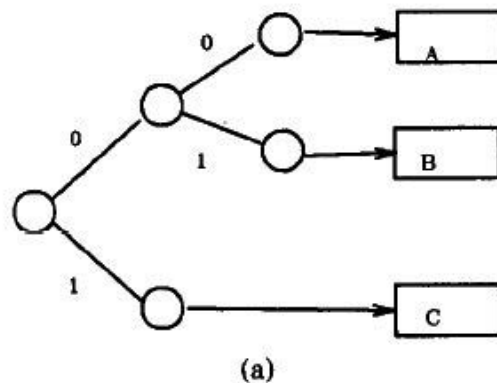
Fig. 5.10 *Trie Showing Three Pages A, B and C*

If page A overflows, it can be split into two pages, as shown in Figure 5.11. An additional bit is needed to divide the records that were all in A between A and the new page D. Now page A will contain records beginning with bit sequence 000..., and page D will contain the records with bit sequence 001. The trie shown in both the figures is very flexible and can grow and shrink as the number of records grow or shrink. The trie must, however, be searched in order to find a desired record. If the trie is skewed due to a skewed key distribution, it can lead to long access time. The trie can be balanced by choosing a different order of bits. The search for the order of bits is shortened by collapsing the trie into a directory. This directory can then be addressed by a hash function. If the chosen hash function is uniform, the implied, collapsed trie will be balanced, which will tend to keep the directory small.
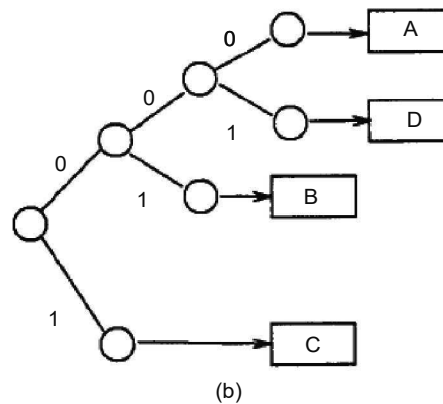
Fig. 5.11 Page A Being Split into Two

Collapsing a trie to a directory is shown in Figure 5.11. The depth of the trie
as shown is two as two bits are needed for deciding how to distribute the records
among the pages. The second bit is not needed for page C in the trie, but still it is
shown to exemplify how the complete directory is formed. Once a complete trie
has been formed, the bits on the search path can become indexes to the directory.
The first two bits of the keys are now an index into the directory, so that there is no
tree to be searched. There are, however, redundant directory entries since two
directory entries are used for page C. This problem of redundant directory entries
is exacerbated by a skewed trie. If a uniform key distribution can be found, the
trie will be balanced, which will minimize the redundancy.
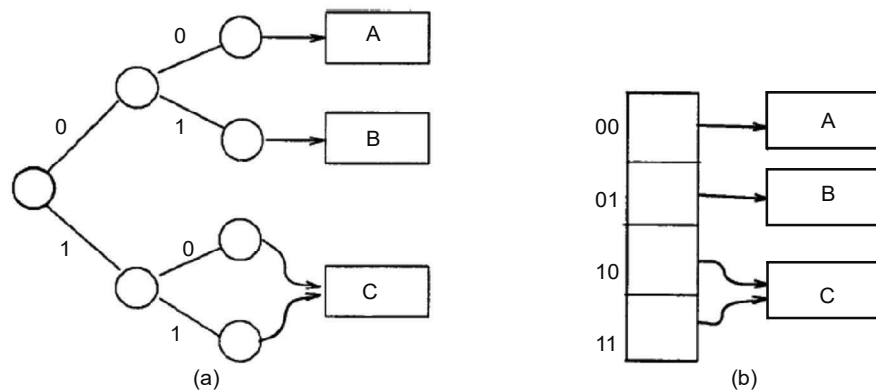
Fig. 5.12 Collapsing a Trie to a Directory

Imagine the situation in which page A overflows and an additional page D is
added. This solution is similar to what is shown in Figure 5.12. As an extra bit was
needed to differentiate between pages A and D, three bits and hence three levels
were added. These extra bits result in a directory with eight entries (2^3). Of those,
many were redundant.

There is a problem with having a directory: It adds a potentially undesirable
level of indirection that can increase access time. A different approach to collapsing
the trie, as shown in Figure 5.13, eliminates the directory.
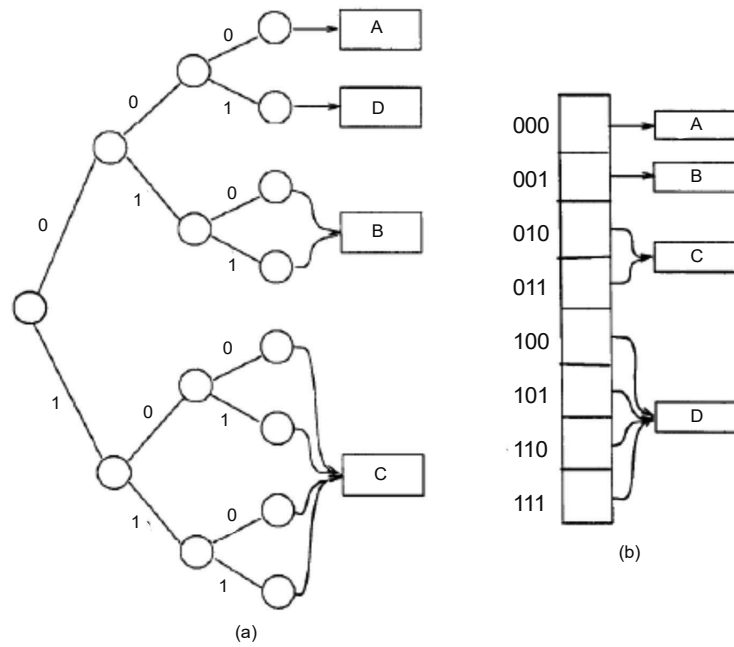
Doubling the directory.

Fig. 5.13 *Doubling the Directory*
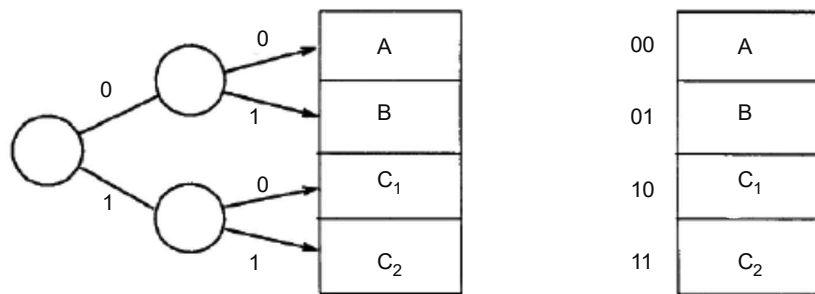
Collapsing without a directory.

Fig. 5.14 *Collapsing Without a Directory*

The directory can be avoided by maintaining the pages in a contiguous address space. Eliminating the directory also eliminates the indirection allowed by the pointers in the directory. Again, consider page C in the directory scheme in Figure 5.14. The two pointers to page C were reproduced in the directory from the lowest level of the trie. With the directory removed, the search path in the trie forms the address of the page rather than the index to the directory. This addressing method requires that there be as many pages in the contiguous address space as there were entries in the directory. The outcome will be addresses 10 and 11, which through the indirection of the directory point to page C. They must now point to two different pages. Thus, the contents of page C must be split across two pages, as shown in Figure 5.14.

Now, when the trie is collapsed, no directory is necessary. The 2 bits that previously indexed the directory can now be used to address the pages themselves, as shown in Figure 5.14. However, when a page overflows in the directory less scheme, the file expands one page at a time rather than doubling.

Basic Dynamic Hashing Schemes

The dynamic hashing systems can be categorized as either directory schemes or directoryless schemes that can correspond directly with the trie models. Classifying the systems will give you a basis for describing the important issues for dynamic schemes. In order to have a hashed file system that dynamically expands and contracts, you need a hash function whose range of values can change dynamically. Most of the dynamic schemes use a function that generates more key bits as the file expands and fewer key bits as the file contracts. One such hash function can be constructed by using a series of functions $h_i(k)$, $i = 0, 1...$ which map the key space K into the integers $(0, 1... 2^i - 1)$ such that for any $k \in K$, either

$$h_i(k) = h_{i-1}(k)$$

or

$$h_i(k) = h_{i-1}(k) + 2^{i-1}$$

There are many ways to obtain the functions $h_i(k)$. YOu might typically use a function

$H(k)$, which maps the key space into random bit patterns of length m, for m is sufficiently large. Then, $h_i(k)$ may be defined as the integers formed by the last i bits of $H(k)$. Thus, if

$$H(k) = b_{m-1} \ldots b_2 b_1 b_0,$$

where, each b_i is either 0 or 1, then

$$h_i(k) = b_{i-1} \ldots b_2 b_1 b_0$$
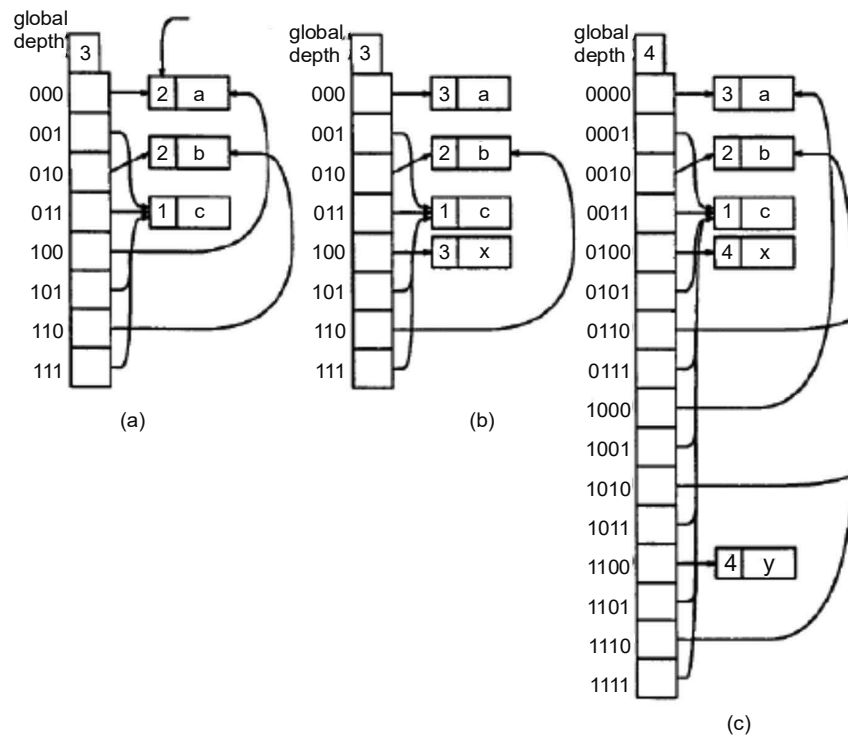
Extendible hashing scheme. (a) Before split. (b) Split a into a and x, (c) Split x into x and y.

Fig. 5.15 *Hashing Schemes*

Using directories

In a directory, d, of pointers, the size of 'd' depends on the directory depth, i.e., # of bits of $h(k)$ is used to index d. It gives the directory depth t. The size of d is 2^t and the # of buckets is almost equal to the size of d.

k	h(k)
A0	100.000
A1	100.001
B0	101.000
B1	101.001
C1	110.001
C2	110.010
C3	110.011
C5	110.101

To search for a key k, examine the bucket d[h(k, t)], where t is the directory depth.

Dynamic Hashing Using Directories–Overflow Resolution

It determines the least u such that $h(k, u)$ is not the same for all the keys in the overflowed bucket.

Case 1: If the least u is greater than the directory depth then it increases the directory depth to this least u value. This requires for us to increase the directory size but not the # of buckets. In the following screen shot try to insert C5 into C1.
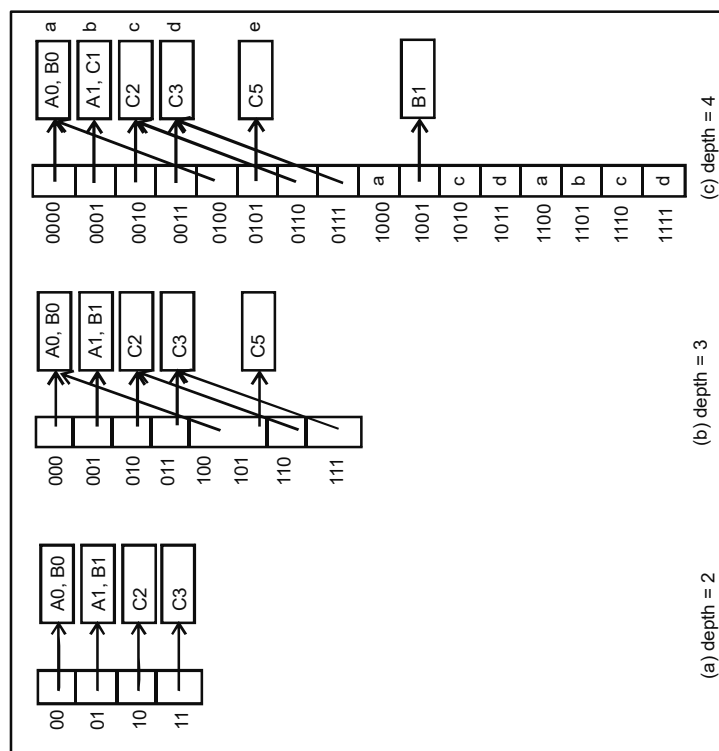
Case 2: If the current directory depth is greater than or equal to u then

Some of the other pointers to the split bucket must be updated to point to the new bucket.

Deletion from a dynamic hash table with a directory is similar to insertion.

Using Directoryless Dynamic Structure

It is also known as linear dynamic hashing in this dynamically size is not increase.

Two variables q and r to keep track of active buckets

Its range is $0 \le q < 2^r$

In which only buckets 0 through 2r+ q–1 are active

The remaining on a chain are overflow buckets

This technique is used indexed like:

(1) h(k, r+1): buckets 0 through q–1 as well as buckets 2rthrough 2r+ q–1

(2) h(k, r): the remaining active buckets

DirectorylessDynamic Hashing –Overflow Resolution

Some steps available for it describe below:

Step 1: Activate bucket 2r+ q

Step 2: Reallocate the entries in the chain q between q and the newly activated bucket $2^r+ q$ and increment q by 1.

If $q= 2^r$, increment r by 1 and reset q to 0. The reallocation is done by using h (k, r+ 1).

Ex. To insert C5 into the table of given below figure (a)

Ex. To insert C1 into the table of given below figure (b)
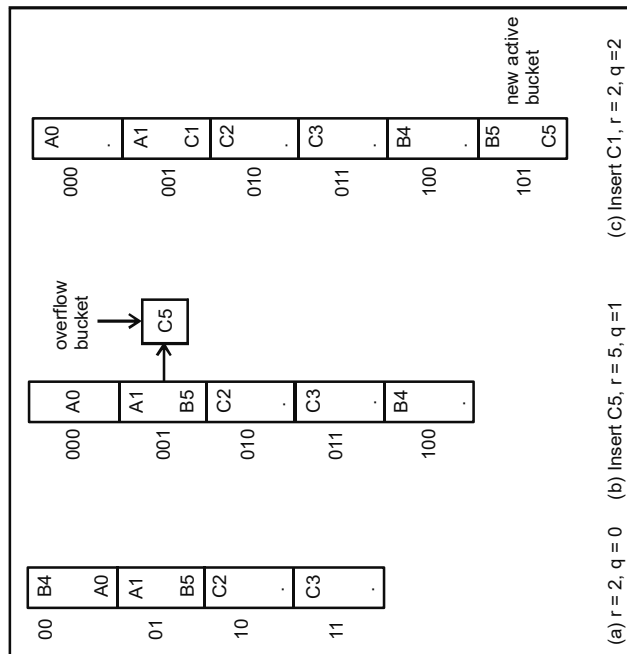
Extendible Hashing

Extendible hashing has been designed to retrieve the data associated with a given key with two disk accesses, i.e., one access to retrieve the desired part of the directory and another to retrieve the data bucket that contains the key. Extendible hashing achieves its goal by merging the concepts of a radix search tree and hashing. An intuitive description of how these two file schemes have been merged into extendible hashing is now presented. Radix search trees, which are naturally extendible, can be modified to yield optimum speed merely by having a pointer for each leaf node. Having a pointer for each leaf node has the effect of flattening the tree forcing it to be degenerate. Next, as for hashing schemes, in order to prevent their performance from deteriorating in a dynamic environment, a mechanism which ensures that the hash table remains balanced under insertions and deletions of records is desired. In extendible hashing, this is achieved by introducing the concept of a degenerate radix search tree to the hash table. In order to make the hash table extendible while the hash function is kept unchanged, the partition of the hash address space must have a variable number of variable sized blocks. It starts with a small number of blocks, and if a data bucket overflows, the partition is changed by introducing new boundaries so that in regions where keys cluster the partition is finer than in regions where keys are sparse. Among various partition techniques, the buddy system partition is recommended for its simplicity. Then, when a data bucket overflows, the corresponding block in the address space is halved, a new data bucket is added, and only those keys in the halved block are reorganized. If bucket underflows as a result of deletions and its buddy is under filled as well, then the two buckets can be coalesced into one, decreasing the number of blocks in the partition by one. The buddy system partition of the hash address space results in the same effect as a radix search tree with radix $r = 2$ over the alphabet $\{0,1\}$. Thus, the buddy system partition can be implemented in a hash table using a directory with 2^d entries, where d is the depth of the buddy system partition, or equivalently, the depth of the radix search tree. Then, the directory allows the d leftmost bits of a key to be taken as an index to the directory. If the depth of the partition increases, the directory doubles in size. Although the directory may become large and have to be kept in external storage, it is normally accessible with one disk I/O operation without using a special algorithm since the bit pattern of a key is used as an index to the directory. Therefore, each record in a file can be retrieved in two disk accesses with extendible hashing.

Bucket Hashing

At times every location in the table is a bucket which will hold a fixed number of items, all of which will be hashed to one location. This speeds up lookups as there is no need to go look at another location. However, a bucket could fill up so that new insertions to this location will need some other collision handler, such as one of the following methods. It means that to store colliding elements in the same position in the table can be achieved by associating a bucket with each address.

A bucket is either one disk block or a cluster of many blocks. The hashing function h maps a key into a relative bucket number. A table maintained in the file header converts the bucket number into the corresponding disk block address.

Linked List Approach (Separate Chaining)

All the items that hash to the same location can be inserted in a linked list whose front pointer is stored in the given location. If the data is on disk, the linked list can be imitated by using record numbers instead of actual pointers. On disk, the hash table file would be a file of fake pointers (record numbers). Each record number would be for a record in another file, i.e., a file of nodes, where each node has a field containing the number of the next record in the list.

Linked List Inside the Table (Coalesced Hashing)

Here, every table entry is a record containing one item that hashed to this location and a record number for another item in the table that hashed to this location, which may have the record number of another that hashed to the same spot, etc. You use a linked list, but this list is stored within the table, whereas in the earlier method the list was external to the table, which only stored the front pointer for each list. Items beyond the first may be stored in an overflow area of the table (sometimes called the cellar) so as not to take up the regular table space. It is recommended that the overflow area take only about 15% of the space available for the table.

Some characteristics are described as below:

1. Slots are grouped into buckets.
2. The hash function transforms the key into a bucket number.
3. Each bucket contains B slots and no collision occurs until the bucket is full. At that point you need to apply a collision processing strategy to find another bucket.
4. The size of a bucket should correspond to disk block address.

Bucket Addressing

To store colliding elements in the same position in the table can be achieved by associating a bucket with each address

A bucket is a block of space large enough to store multiple items
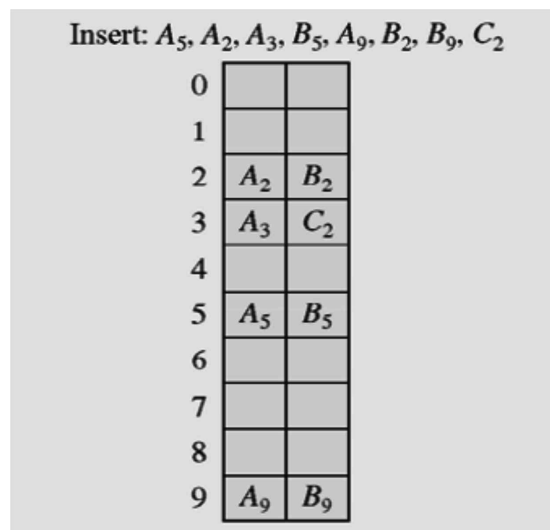
Fig. 5.16 *Collision resolution with bucket and linear probing method.*
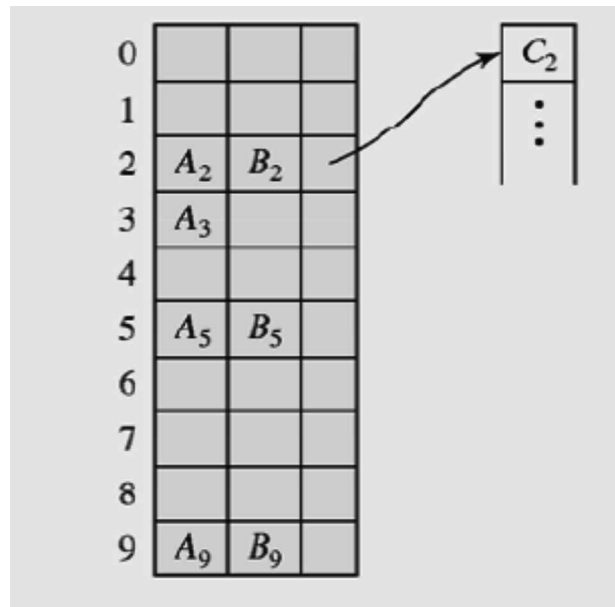
Fig. 5.17 *Collision resolution with bucket and overflow area*

Bucket Hashing Example

The following is the example of Bucket-hashing. In this, buckets, each having four records, will be located by hash function that leads the key to the home bucket.

order of insertion	key	home bucket	# accesses for bucket hashing
1	42	4	
2	20	1	
3	23	4	
4	99	4	
5	58	1	
6	137	4	
7	172	1	
8	153	1	

Applications Related to Hash Tables:

1. **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

2. **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols

frequently. Therefore, it is important that symbol tables be implemented very efficiently.

3. **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

4. **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

5. **Browser Cashes**: Hash tables are used to implement browser cashes.

Disadvantages

If the hash function is poor, then there will be excessive collisions and performance will quickly approach O (N). Some hash functions are slower than the others, and a slow hash function could increase the constant for O (1) due to expensive operations and the hash table could be slower than other data structures which are theoretically slower. Only separate chaining can easily handle deletion of a key with the use of collision resolution. Separate chaining is also the only collision resolution strategy that can easily handle duplicate keys. A few moments thought will show this to be true because to search for duplicates in a chain, the entire chain must be searched, whereas with duplicates in an open addressed table, the entire cluster must be searched for linear probing, and the entire table for non-linear probing.

The keys in a table need to be rehashed when the size is altered because each insertion and subsequent search depends on the table size to give the correct result. If the table size changes, the indices for all of the previously hashed keys are invalidated because future searches will no longer work correctly.

As a hash table is an unordered data structure, certain operations are difficult and expensive. Range queries, selection and sorted traversals are possible only if the keys are copied into a sorted data structure. There are hash table implementations that keep the keys in order, but they are far from being efficient.

A solution to this problem is a hybrid data structure called a hashlist. The idea is to use a sorted data structure such as a binary search tree or sorted linked list as the primary storage unit. Then the keys in the primary structure are hashed and references to them are stored in the hash table. This has excellent performance properties. However, in practice, the hashlist can be more work than it is worth.

The keys in a hash table are ideally spread in a uniform, pseudorandom pattern throughout the table. This OK in theory, but in practice, processors will create a cache of frequently accessed memory under the assumption that adjacent memory is more likely to be accessed than non-adjacent memory. As it is, unless a fix like the hashlist is used, searching in a hash table may result in to many cache misses for medium to large tables. This is because the actions of the hash table do not fit into the usage pattern for efficient caching. For smaller tables which fit completely in a cache line, any performance problems are unlikely to be noticeable.

<table>
<tr><td colspan="2">Check Your Progress</td></tr>
<tr><td>10.</td><td>What is a trie?</td></tr>
<tr><td>11.</td><td>What is extendible hashing?</td></tr>
</table>

5.6 ISAM

ISAM stands for Indexed Sequential Access Method. It is a method for creating, maintaining, and manipulating computer files of data so that records can be retrieved sequentially or randomly by one or more keys. Indexes of key fields are maintained to achieve fast retrieval of required file records in Indexed files.

File Organization

Arrangement of the records in a file plays a significant role in accessing them. Moreover, proper organization of files on the disk helps in accessing the file records efficiently. There are various methods, known as file organization of organizing the records in a file while storing a file on the disk.

(i) Sequential File Organization

Often, the records of a file are required to be processed in the sorted order based on the value of one of its fields. If the records of the file are not physically placed in the required order, it consumes time to fulfill this request. However, if the records of that file are placed in the sorted order based on that field, we would be able to efficiently fulfill this request. A file organization in which the records are sorted, based on the value of one of its fields is called sequential file organization, and such a file is called sequential file. In a sequential file, the field on which the records are sorted is called ordered field. This field may or may not be the key field. In case, the file is ordered on the basis of a key, then the field is called the ordering key. The sequential file does not make any improvement in processing the records in random order.

Operation like searching of records is more efficient in a sequential file if the search condition is specified on the ordering field because a binary search is applicable instead of a linear search. Moreover, retrieval of records with the range condition specified, on the ordering field is also very efficient. In this operation, all the records, starting with the first record that satisfies the range selection condition, till the first record that does not satisfy the condition are retrieved. However, handling deletion and insertion operations are complicated. Deletion of a record leaves a blank space in between the two records. This situation can be handled by using a deletion marker. When a new record is to be inserted in a sequential file, there are two possibilities. Firstly, the record needs to be inserted at its actual position in the file. Obviously, it requires locating the first record that has to come after the new record and making space for the new record. Making space for a record may require shifting a large number of records and this is very costly in terms of disk access. Secondly, we can insert that record in an overflow area allocated to the file instead of its correct position in the original file. The records in the overflow

area are unordered. Periodically, the records in the overflow area are sorted and merged with the records in the original file.

The second approach of insertion makes the insertion process efficient; however, it may affect the search operation. This is because the required record needs to be searched in the overflow area using linear search if it is not found in the original file using binary search.

(ii) Random File Organization

Unlike sequential file, records in this file organization are not stored sequentially. Instead, each record is mapped to an address on the disk on the basis of its key value. One such technique used for this mapping of record to an address is called hashing. Hashing consists of two parts—a hash function and a collision-resolution technique.

When a record is to be inserted in a direct (or random) access file, a hash function is applied on the key value of the record that gives the page address where the record is to be placed. If a record is mapped to page, which is already full then another page address is computed for the record using the collision-resolution technique.

Since each record is placed at the page indicated by the hash function, searching a record is simple. The same hash function is applied on the key value of the record to be searched, which gives the address of the page where the desired record may be found. Then, all the records of that page are examined to locate the desired record. If the desired record is not found in that page, the address of another page is computed according to the method employed in the collision-resolution technique.

Using ISAM with Indexed Sequential File Organization

The file organization, using the indexed sequential access method (ISAM) provides the benefits of both the sequential and random file organization methods. This type of organization allows accessing the records both sequentially, based on some key value and also directly, by using the same key. The records are stored sequentially in the disk blocks with each block containing a specified number of records. The records can be accessed through an index. An index is a file having two fields—one stores the key value and contains a pointer to the record in the original file. Each entry in the index file stores the key value of the first record in each disk block and a pointer to the disk block containing that record.

To understand this, consider the file shown in Figure 5.18, which contains information about the various books. Now if an index is created on the field Book_Id, the index file will be as shown in Figure 5.19.

Book_Id	Book_title	Category	Page_count	
354	Ransack	Novel	200	Record 1
456	Introduction to German language	Language Book	450	Record 2
489	C++	Textbook	800	Record 3
556	Differential Calculus	Textbook	200	Record 4
678	Call Away	Novel	200	Record 5
887	Learning French language	Language Book	500	Record 6

Fig. 5.18 *A Sample of Books File*
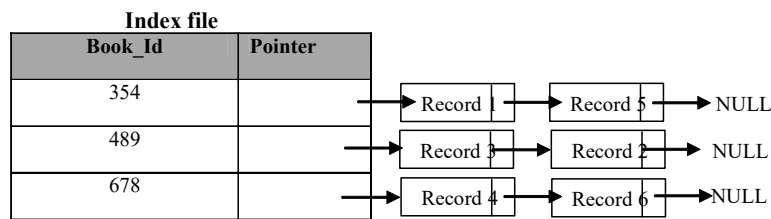
Fig. 5.19 *Indexed Sequential Organization*

Check Your Progress

12. What do you mean by sequential file organization?

13. Define hashing.

5.7 SEARCHING

While solving a problem, a programmer may need to search a value in an array. The process of finding the occurrence of a particular data item in a list is known as searching. Search is said to be successful or unsuccessful depending on whether the data item is found or not. The various search techniques are linear binary, Fibonacci and interpolation searches.

Linear Search

Linear search is one of the simplest searching techniques. In this technique, the array is traversed sequentially from the first element until the value is found or the end of the array is reached. While traversing, each element of the array is compared with the value to be searched, and if the value is found the search is said to be successful. This technique is suitable for performing a search in a small array or in an unsorted array.

Algorithm 5.1 Linear Search

```
linear_search (ARR, size, item)
1.  Set i = 0
2.  While i < size
        If ARR [i] = item        //item is the value to be searched
                Return i and go to step 4
        End If
        Set i = i  + 1
    End While
3.  Return -1            //search unsuccessful
4.  End
```

Program 5.1: A program to perform linear search

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
int linear_search(int ARR[] , int size , int item );

void main()
{
int ARR[MAX];
    int item, size, pos, i;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
    scanf("%d", &item);
    pos=linear_search(ARR, size, item);
    if (pos==-1)
        printf("\nElement not found");
    else
        printf("\nElement found at location: %d", pos+1);
    getch();
}

int linear_search(int ARR[], int size, int item)
{
    int i;
    for (i=0;i<size;i++)
```

```
    {
        if(ARR[i]==item)
            return (i);
    }
    return (-1);
}
```

The output of the program is
```
Enter the size of the array (max 20): 5

Enter elements of the array:
1
4
3
6
7

Enter the element to be searched: 4

Element found at location: 2
```

Binary Search

The binary search technique is used to search for a particular element in a sorted (in ascending or descending order) array. In this technique, the element to be searched (say, `item`) is compared with the middle element of the array. If `item` is equal to the middle element, then the search is successful. If `item` is smaller than the middle element, `item` is searched in the segment of the array before the middle element. However, if `item` is greater than the middle element, `item` is searched in the array segment after the middle element. This process is repeated until the element is found or the array segment is reduced to a single element that is not equal to `item`.

At every stage of the binary search technique, the array is reduced to a smaller segment. It searches a particular element in the lowest possible number of comparisons. Hence, the binary search technique is used for larger and sorted arrays, as it is faster compared to linear search. For example, consider an array `ARR` shown here:

1	2	3	4	5	6	7
0	1	2	3	4	5	6

To search an `item` (say, 7) using binary search in the array `ARR` with `size=7`, the following steps are performed.

1. Initially, set `LOW=0` and `HIGH=size-1`. The middle of the array is determined using the formula `MID=(LOW+HIGH)/2`, that is, `MID=(0+6)/2`, which is equal to 3. Thus, `ARR [MID]=4`.

LOW   MID   HIGH

1	2	3	4	5	6	7
0	1	2	3	4	5	6
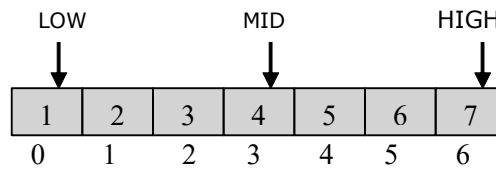
2. Since the value stored at ARR [3] is less than the value to be searched, that is 7, the search process is now restricted from ARR[4] to ARR[6]. Now LOW is 4 and HIGH is 6. The middle element of this segment of the array is calculated as MID=(4+6)/2, that is, 5. Thus, ARR[MID]=6.

LOW   MID HIGH

5	6	7
4	5	6
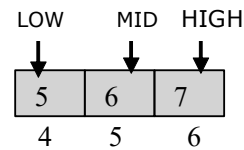
3. The value stored at ARR[5] is less than the value to be searched, hence the search process begins from the subscript 6. As ARR[6] is the last element, the item to be searched is compared with this value. Since ARR[6] is the value to be searched, the search is successful.

Algorithm 5.2: Binary Search

```
binary_search(ARR, size, item)
1.  Set LOW = 0
2.  Set HIGH = size - 1
3.  While LOW <= HIGH
            Set MID = (LOW + HIGH) / 2
            If ITEM = ARR[MID]
                    Return MID and go to step 5
            Else If item < ARR[MID]
                    Set HIGH = MID – 1
            Else
                    Set LOW = MID + 1
            End If
    End While
4.  Return -1
5.  End
```

Program 5.2: A program to perform binary search

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
int binary_search(int ARR[], int size, int item);

void main()
{
    int ARR[MAX];
    int item, size, pos, i;
    do
    {
```

```
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter elements in sorted order:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
    scanf("%d", &item);
    pos=binary_search(ARR, size, item);
    if (pos==-1)
        printf("\nElement not found");
    else
        printf("\nElement found at location: %d", pos+1);
    getch();
}

int binary_search(int ARR[], int size, int item)
{
    int LOW=0;
    int HIGH=size-1;
    int MID;
    while(LOW<=HIGH)
    {
        MID=(HIGH+LOW)/2;
        if(ARR[MID]==item)
            return MID;
        else
            if(item<ARR[MID])
                HIGH=MID-1;
            else
                LOW=MID+1;
    }
    return (-1);
}
```

The output of the program is

```
Enter the size of the array (max 20): 5

Enter elements in sorted order:
11
22
33
44
55
```

```
Enter the element to be searched: 33

Element found at location: 3
```

Fibonacci Search

Like binary search, Fibonacci search is also applied on sorted arrays. However, unlike, binary search it does not compare the element to be searched (say, `item`) with the middle element of the list. Instead, it uses Fibonacci series to determine the index, say `pos`, where to look in the array.

The Fibonacci series up to `n` terms (generally denoted by `F0, F1, F2, …, Fn),` is generated as follows:

```
0, 1, 1, 2, 3, 5, 8, 13, 21...
That is, each successive term is the sum of its two
preceding terms. That is,
    If n=0 or n=1
Then
Fn = n,
    Else
        Fn = Fn-2 + Fn-1
```

Initially, the search takes a smallest number, say `Fm`, of Fibonacci series that is greater than or equal to the `size`, where `size` is the number of elements in the array. Then, it compares the `item` with the element at `Fm-1` position in the array. The result is as follows:

- If they are equal, search is successful, element found at position `Fm-1+1`.
- If the `item` is smaller, it is searched in the sub-list left to `Fm-1`.
- If the `item` is greater, it is searched in the sub-list right to `Fm-1`.

If `item` is not found, again a smallest number of Fibonacci series that is greater than or equal to the size of the sub-list (left or right) to be searched is taken, and the whole process is repeated until the desired element is found or the sub-list is reduced to a single element that is not equal to `item`.

To understand the Fibonacci search, consider the array `ARR` shown here. Suppose you need to search the element 13.

1	7	13	19	25	31	36
0	1	2	3	4	5	6

Since the size of the array is 7, the initial value of `Fm` will be 8. The search first compares the element 13 with element at index 5 (`Fm-1`), that is, 26. Since 13 is less than 26, the sub-list left to index 5 is considered. The size of this sub-list is 5, therefore, new `Fm` will be 5. Now, the element 13 is compared with the element at index 3 (`Fm-1`), that is, 13. Since, it is the desired element, the search is successful, and the element is found at position 4.

Algorithm 5.3 Fibonacci Search

```
fibonacci_search(ARR, size, item)

1.  Set flag = 0, first = 0, pos = -1, last = size
2.  While (flag != 1 AND first <= last)
            Set index = retfib(size)            //calling retfib() function
            If item = ARR[index+first]
                    Set flag = 1
                    Set pos = index
                    Break                        //jump out of the loop
            Else if item > ARR[index+first]
                    Set first = index + 1
                    Set size = last – first
            Else
                    Set last = index – 1
                    Set size = last – first + 1
            End If
    End While
3.  If flag = 1
        Return (pos + first + 1) and go to step 4
    Else
        Return -1
    End If
4.  End
```

```
retfib(n)                   //function to generate a Fibonacci number

1.  Set a = 1, b = 1, c = 1
2.  If (n = 0 OR n = 1)
        Return 0 and go to step 3
    Else
        While c < n
                Set c = a + b
                Set a = b
                Set b = c
        End While
        Return a
    End If
3.  End
```

Program 5.3: A program to implement Fibonacci search

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototypes*/
int fibonacci_search(int [], int, int);
int retfib(int n);

void main()
{
    int ARR[MAX], size, item, pos, i;
do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
```

```
printf("\nEnter elements in sorted order:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
    scanf("%d", &item);
    pos=fibonacci_search(ARR, size, item);
    if (pos==-1)
        printf("\nElement not found");
    else
        printf("\nElement found at location: %d ", pos);
    getch();
}


int retfib(int n)
{
    int a=1, b=1, c=1, i;
    if(n==0 || n==1)
        return 0;
    else
    {
        while(c<n)
        {
            c=a+b;
            a=b;
            b=c;
        }
        return a;
    }
}


int fibonacci_search(int ARR[], int size, int item)
{
    int flag=0, first=0, index, pos=-1, last=size;
    while(flag!=1 && first<=last)
    {
        index=retfib(size);
        if(item==ARR[index+first])
        {
            flag=1;
            pos=index;
            break;
        }
        else if(item>ARR[index+first])
        {
```

```
            first=index+1;
            size=last-first;
        }
        else
        {
            last=index-1;
            size=last-first+1;
        }
    }
    if(flag==1)
        return (pos+first+1);
    else
        return -1;
}
```

The output of the program is

```
Enter the size of the array (max 20): 5

Enter elements in sorted order:
45
67
89
100
120

Enter the element to be searched: 67

Element found at location: 2
```

Interpolation Search

Interpolation search is similar to binary search in the sense that it also applies on sorted arrays. However, it is based on the assumption that the elements in the list are uniformly distributed. This is done is a manner, that the probability of an element being in a particular range equals its probability of being in any other range of the same length. Thus, instead of determining the middle element using the formula (low+high)/2, interpolation search determines the location of the item to be searched. This is done according to the magnitude of the item relative to the first and last elements of the array.

For example, if an array contains 10 elements ranging from 1 to 100 which are uniformly distributed, then according to the interpolation search, the element 50 lies in between the list.

Therefore, it uses the following formula for calculating the mid:
```
mid = low+(high-low)*((item-ARR[low])/(ARR[high]-
ARR[low]))
```

To understand the interpolation search, consider the array ARR. Suppose you have to search the element 13. Note that the elements in this array are uniformly distributed.

1	7	13	19	25	31	36
0	1	2	3	4	5	6

Initially, `low=0, high=6, ARR[low]=ARR[0]=1`, and `ARR[high]=ARR[6]=37`, therefore the value of `mid` can be calculated as follows:

```
mid = 0+(6-0)*((13-1)/(37-1))
    = 6*(12/36)
    = 6*0.33
    = 2
```

Since `ARR[mid]=ARR[2]=13`, which is the desired element, the search terminates successfully.

Algorithm 5.4 Interpolation Search

```
interpol_search(ARR, size, item)

1.  Set low = 0, high = size – 1, flag = -1
2.  While low <= high
        If high = low
                If ARR[low] = item
                        Set flag = low
                End If
                Break           //jump out of the loop
        End If
        Set mid = low + (high - low) * ((item - ARR[low]) / (ARR[high] - ARR[low]))
        If ARR[mid] = item
                Set flag = mid
                Break
        Else if ARR[mid] > item
                Set high = mid - 1
        Else
                Set low = mid + 1
        End If
3.  Return (flag)
4.  End
```

Program 5.4: A program to implement interpolation search

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

int interpol_search(int [], int, int);

void main()
{
    int ARR[MAX];
    int item, size, pos, i;
    do
    {
        clrscr();
```

```
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter elements in sorted order:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
        scanf("%d", &item);
    pos=interpol_search(ARR, size, item);
    if (pos==-1)
        printf("\nElement not found");
    else
        printf("\nElement found at location: %d", pos+1);
    getch();
}


int interpol_search(int ARR[], int size, int item)
{
    int mid, low, high, flag=-1;
    low=0;
    high=size-1;
    while(low<=high)
    {
        if(high==low)
        {
            if(ARR[low]==item)
                flag=low;
            break;
        }
        mid=low+(high-low)*(float)(item-ARR[low])/
(ARR[high]-ARR[low]);
        if(ARR[mid]==item)
        {
            flag=mid;
            break;
        }
        else if(ARR[mid]>item)
            high=mid-1;
        else
            low=mid+1;
    }
    return (flag);
}
```

The output of the program is

```
Enter the size of the array (max 20): 8

Enter elements in sorted order:
20
40
65
80
100
120
180
200

Enter the element to be searched: 180

Element found at location: 7
```

Comparison of Different Search Algorithms

This section gives the analysis of different search algorithms and determines their time complexities.

Analysis of Linear Search

In the best case, when the item is found at first position, the search operation terminates successfully with only one comparison. Thus, in this case the complexity of the algorithm is `O(1)`. In the worst case, when the item to be searched appears at the end of the list or is not present in the list, linear search requires n comparisons. In both the cases, the average complexity of linear search is `O(n)`.

Analysis of Binary Search

In each iteration, binary search algorithm reduces the array to one half. Therefore, for an array containing n elements, there will be `log2n` iterations. Thus, the complexity of binary search algorithm is `O(log2n)`. This complexity will be same irrespective of the position of the element, even if the element is not present in the list.

Analysis of Fibonacci Search

The complexity of Fibonacci search is similar to that of binary search, that is, `O(log2 n)`. However in general, the performance of Fibonacci search is always worse than binary search. Finding the middle element in binary search involves division operation `[(low+high)/2]`, but in Fibonacci search only addition or subtraction operation is involved. Therefore, the average performance of Fibonacci search may be better than binary search on computers where division is more time consuming than addition or subtraction.

Analysis of Interpolation Search

The performance of interpolation search is highly dependent on the distribution of elements in the list. In the best case when the elements are uniformly distributed,

the interpolation search requires `log2n(log2n)` comparisons, as compared to binary search which requires `log2n` comparisons. Thus, in this case interpolation search has a very poor performance. On the other hand, if the elements are not uniformly distributed, then interpolation search gives a very poor performance. In the worst case, the value of `mid` can consistently be equal to `low+1` or `high-1`. In this case, the performance of interpolation search deteriorates to linear search. However, the performance of binary search can never exceed `log2n`.

Check Your Progress

14. What is searching? Name the various searching techniques.

15. Which searching technique is suitable for unsorted arrays?

16. When is linear search preferred?

5.8 SORTING

The process of arranging the data in some logical order is known as sorting. The order can be ascending or descending for numeric data, and alphabetically for character data. There are two types of sorting, namely, internal sorting and external sorting. If all the data that is to be sorted fits entirely in the main memory, then internal (in-memory) sorting is used.

On the other hand, if all the data that is to be sorted do not fit entirely in the main memory, external sorting is required. An external sorting requires the use of external memory such as disks or tapes during sorting. In external sorting, some part of the data is loaded into the main memory, sorted using any internal sorting technique and written back to the disk in some intermediate file. This process continues until all the data is sorted.

Internal Sorting

There are different internal sorting algorithms such as insertion sort, bubble sort, selection sort, heap sort, merge sort, quick sort and bucket sort. The choice of a particular algorithm depends on the properties of the data and the operations to be performed on the data. For all these algorithms, we will consider an array `ARR` containing `n` elements, which are to be sorted in an ascending order.

Insertion Sort

The insertion sort algorithm selects each element and inserts it at its proper position in the earlier sorted sub-list. In the first pass, the element `ARR[1]` is compared with `ARR[0]`, and if `ARR[1]` and `ARR[0]` are not sorted, they are swapped. In the second pass, the element `ARR[2]` is compared with `ARR[0]` and `ARR[1]`, and it is inserted at its proper position in the sorted sub-list containing the elements `ARR[0]`, `ARR[1]`. Similarly, during `i`th iteration, the element `ARR[i]` is placed at its proper position in the sorted sub-list containing the elements `ARR[0]`, `ARR[1]`, `ARR[2]`,…, `ARR[i-1]`.

In order to determine the actual position of the element (say, `ARR[i]`) in the sorted sub-list containing the elements `ARR[0]`, `ARR[1]`, …, `ARR[i-`

1], the element ARR[i] is compared with all other elements to its left, until an element ARR[j] is found such that ARR[j]<=ARR[i]. Now, to insert the element at its actual position, all the elements ARR[i-1], ARR[i-2], ARR[i-3],…, ARR[j+1] are shifted one position towards the right to create the space for ARR[i], and then ARR[i] is inserted at (j+1)st position.

To understand the insertion sort algorithm, consider an unsorted array shown here. The steps to sort the values stored in the array in ascending order using insertion sort are given here.

| 7 | 33 | 20 | 11 | 6 |
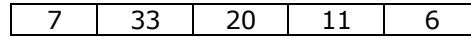
- The first value, that is, 7 is trivially sorted by itself.
- Then the second value 33 is compared with the first value 7. Since 33 is greater than 7, no changes are made.
- Next, the third element 20 is compared with its previous elements (elements towards its left). Since 20 is smaller than 33 but greater than 7, it is inserted at second position. For this, the element 33 is shifted one position towards right and 20 is inserted at its appropriate (second) position.

| 7 | 33 | 20 | 11 | 6 |

| 7 | 20 | 33 | 11 | 6 |
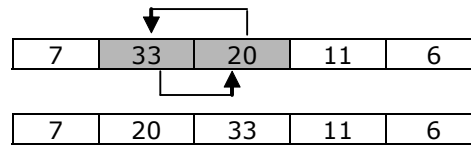
- Then, the fourth element 11 is compared with its previous elements. Since 11 is greater than 7 and less than 20 and 33, it is placed between 7 and 20. For this, the elements 20 and 33 need to be shifted one position towards the right.

| 7 | 20 | 33 | 11 | 6 |

| 7 | 11 | 20 | 33 | 6 |
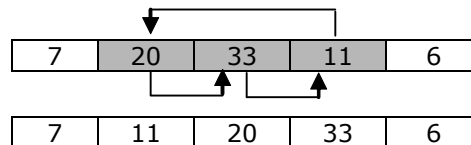
- Finally, the last element 6 is compared with all the elements preceding it. Since it is smaller than all the other elements, they are shifted one position towards right and 6 is inserted at the first position in the array. After this pass, the array is sorted.

| 7 | 11 | 20 | 33 | 6 |

| 6 | 7 | 11 | 20 | 33 |
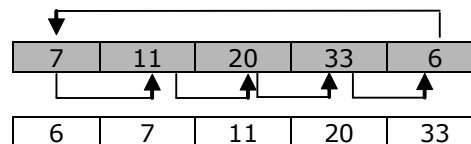
The final sorted array is as follows:

| 6 | 7 | 11 | 20 | 33 |
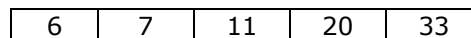
Algorithm 5.5 Insertion Sort

```
insertion_sort(ARR, size)

1.  Set i = 1
2.  While (i < size)
        Set temp = ARR[i]
        j = i - 1
        While (temp < ARR[j] AND j >= 0)
                Set ARR[j+1] = ARR[j]
                Set j = j - 1
        End While
        Set ARR[j+1] = temp
        Print ARR after i^th pass
        Set i = i + 1
    End While
3.  Print "No. of passes: ", i-1
4.  End
```

Program 5.5: A program to show sorting of an array using insertion sort

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20


/*Function prototype*/
void insertion_sort(int [], int);


void main()
{
    int ARR[MAX],i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    insertion_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d  ", ARR[i]);
    getch();
}
void insertion_sort(int ARR[], int size)
{
    int i, j, k, temp, count=0;;
    for (i=1;i<size;i++)
    {
```

```
            temp=ARR[i];
            j=i-1;
            if(temp<ARR[j])
            {
                while(temp<ARR[j] && j>=0)
                {
                    ARR[j+1]=ARR[j];
                    j—;
                }
            }
            ARR[j+1]=temp;
            printf("\nArray after pass %d: ", i);
            for(k=0;k<size;k++)
                printf("%d  ", ARR[k]);
        }
        printf("\nNo. of passes: %d", i-1);
    }
```

The output of the program is

```
Enter the size of the array (max 20): 5
Enter the elements of the array:
35
20
4
10
5
Array after pass 1: 20  35  4  10  5
Array after pass 2: 4  20  35  10  5
Array after pass 3: 4  10  20  35  5
Array after pass 4: 4  5  10  20  35
No. of passes: 4
The sorted array is: 4  5  10  20  35
```

Bubble Sort

The bubble sort algorithm requires n−1 passes to sort an array. In the first pass, each element (except the last) in the list is compared with the element next to it, and if one element is greater than the other then both the elements are swapped. After the first pass, the largest element in the list is placed at the last position. Similarly, in the second pass the second largest element is placed at its appropriate position. Thus, in each subsequent pass, the next largest element is placed at its appropriate position. Since this algorithm makes the larger values to 'bubble up' to the end of the list, it is named bubble sort.

The bubble sort algorithm possesses an important property. This property is that if a particular pass is made through the list without swapping any items, then

there will be no further swapping of elements in the subsequent passes. This property can be used to eliminate the unnecessary passes once the list is sorted in the desired order. For this, a flag variable can be used to detect if any interchange has been made during the pass. We use flag = 0 to indicate that no swaps have occurred in a particluar pass, therefore, no further passes are required.

To understand the bubble sort technique, consider an unsorted array shown here.

8	7	65	5	43

The steps to sort the values stored in the array in ascending order using bubble sort are as follows:

First pass:

1. The values 8 and 7 are compared with each other. Since 7 is smaller than 8, both the values are swapped with each other.

2. **No swapping:** Next, the values 8 and 65 are compared with each other. Since 8 is less than 65, means they are in proper order and, hence, no swapping is required. The list remains unchanged.

No swapping

7	8	65	5	43

3. **Elements compared:** Then the values 65 and 5 are compared with each other. Since 5 is less than 65, both the values are swapped.

Elements compared

7	8	65	5	43

7	8	5	65	43

Elements swapped

4. **Elements compared:** Next, the values 65 and 43 are compared with each other. Since 43 is less than 65, both the values are swapped.

Elements compared

7	8	5	65	43

7	8	5	43	65

Elements swapped

After the first pass, the largest value of the array (here, 65) is placed at last position.

Second pass

1. The values 7 and 8 are compared with each other. Since 7 is smaller than 8, no swapping is required.

2. Then the values 8 and 5 are compared. Since 8 is greater than 5, both are swapped.

3. Next, the elements 8 and 43, and 43 and 65 are compared. Since they are already in ascending order, they need not be swapped.

| 7 | 5 | 8 | 43 | 65 |

Third Pass

1. The values 7 and 5 are compared with each other. Since 7 is greater than 5, both are swapped.

2. Since the remaining elements are already in ascending order, they are not swapped.

| 5 | 7 | 8 | 43 | 65 |

Fourth Pass

1. In the fourth pass, no swapping is required as all the elements are already in ascending order. Thus, at the end of this pass, the list is sorted in ascending order as follows:

| 5 | 7 | 8 | 43 | 65 |

Algorithm 5.6 Bubble Sort

```
bubble_sort(ARR, size)

1.  Set i = 0, flag = 1
2.  While (i < size-1 AND flag = 1)
        Set j = 0
        Set flag = 0
        While (j < size-i-1)
            If (ARR[j] > ARR[j+1])
                    Set flag = 1            //swap will occur, hence set flag = 1
                    Set temp = ARR[j]       //temp is temporary variable used to swap
                                            //two values
                    Set ARR[j] = ARR[j+1]
                    Set ARR[j+1] = temp
            End If
            Set j = j + 1
        End While
        Print ARR after (i+1)th pass
        Set i = i + 1
    End While
3.  Print "No. of passes: ", i
4.  End
```

Program 5.6: A program to show sorting of an array using bubble sort

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
void bubble_sort(int [], int);

void main()
{
    int ARR[MAX],i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
```

```
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    bubble_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d  ", ARR[i]);
    getch();
}


void bubble_sort(int ARR[], int size)
{
    int i, j, k, temp, flag=1;
    i=0;
    while (i<size-1 && flag==1)
    {
        flag=0;
        for(j=0;j<size-i-1; j++)
        {
            if (ARR[j]>ARR[j+1])
            {
                flag=1;
                temp=ARR[j];
                ARR[j]=ARR[j+1];
                ARR[j+1]=temp;
            }
        }
        printf("\nArray after pass %d: ", i+1);
        for(k=0;k<size;k++)
            printf("%d  ", ARR[k]);
        i++;
    }
    printf("\nNo. of passes: %d", i);
}
```

The output of the program is

```
Enter the size of the array (max 20): 5
Enter the elements of the array:
8
7
65
5
```

43

```
Array after pass 1: 7  8  5  43  65
Array after pass 2: 7  5  8  43  65
Array after pass 3: 5  7  8  43  65
Array after pass 4: 5  7  8  43  65
No. of passes: 4
The sorted array is: 5  7  8  43  65
```

Selection Sort

In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position). Then, the second smallest element is searched and swapped with the second element in the list (that is, it is placed at the second position), and so on.

Like bubble sort algorithm, the selection sort also requires `n-1` passes to sort an array containing `n` elements. However, there is a slight difference between the selection sort and the bubble sort algorithms. In selection sort, the smallest element is the first one to be placed at its correct position, then the second smallest element takes its position, and so on. Whereas, in bubble sort, the largest element is the first one to be placed at its appropriate position, then the second largest element, and so on.

To understand the selection sort algorithm, consider an unsorted array shown here.

8	33	6	21	4
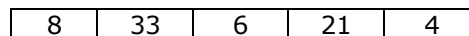
The steps to sort the values stored in the array in ascending order using selection sort are as follows:

1. In the first pass, the entire array is scanned for the smallest element, which is 4 in this list. It is swapped with the first element, that is, 8. Thus, 4 is placed at its correct position and is not used for any further comparisons.

2. In the second pass, the smallest element is searched from the last four elements, which is 6. It is swapped with the second element, that is, 33.

4	33	6	21	8

4	6	33	21	8
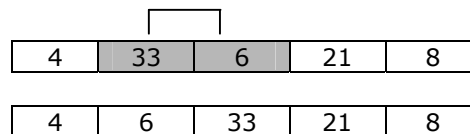
3. In the third pass, the smallest element is searched from the last three elements, which is 8. This value is swapped with the third element, that is, 33.

4	6	33	21	8

4	6	8	21	33
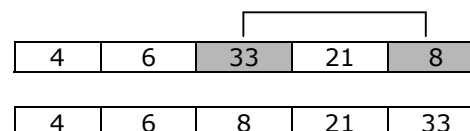
4. In the fourth pass, the smallest element is searched from the last two elements. Since 21 is smaller than 33, therefore, no changes are made in the list obtained after the third pass, and the list is sorted in ascending order. The sorted list is as follows.

4	6	8	21	33

Algorithm 5.7 Selection Sort

```
selection_sort(ARR, size)

1.  Set i = 0
2.  While (i < size-1)
        Set small = ARR[i]
        Set pos = i
        Set j = i + 1
        While (j < size)          //searching the smallest element in unsorted list
                If (ARR[j]<small)
                        Set small = ARR[j]
                        Set pos = j
                End If
                Set j = j + 1
        End While
        Set ARR[pos] = ARR[i]    //placing the smallest element at its correct position
        Set ARR[i] = small
        Print ARR after (i+1)ᵗʰ pass
        Set i = i + 1
    End While
3.  Print "No. of passes: ", i
4.  End
```

Program 5.7: A program to show sorting of an array using selection sort

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
void selection_sort(int [], int);

void main()
{
    int ARR[MAX],i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    selection_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
```

```
                        printf("%d  ", ARR[i]);
                    getch();
                }
```

```
            void selection_sort(int ARR[], int size)
            {
                int i, j, k, small, pos, count=0;
                for (i=0;i<(size-1);i++)
                {
                    small=ARR[i];
                    pos=i;
                    for (j=i+1;j<size;j++)
                    {
                        if (ARR[j]<small)
                        {
                            small=ARR[j];
                            pos=j;
                        }
                    }
                    ARR[pos]=ARR[i];
                    ARR[i]=small;
                    printf("\nArray after pass %d: ", i+1);
                    for(k=0;k<size;k++)
                        printf("%d  ", ARR[k]);
                }
                printf("\nNo. of passes: %d", i);
            }
```

The output of the program is

```
Enter the size of the array (max 20): 5

Enter the elements of the array:
8
6
33
21
5

Array after pass 1: 5  6  33  21  8
Array after pass 2: 5  6  33  21  8
Array after pass 3: 5  6  8  21  33
Array after pass 4: 5  6  8  21  33
No. of passes: 4
The sorted array is: 5  6  8  21  33
```

Heap Sort

Heap sort is a much more efficient version of selection sort. Like selection sort, it also first determines the largest (or smallest) element of the list. Then it places it at the end (or beginning) of the list, and continues with the rest of the list. However, it accomplishes this task efficiently by using a different data structure called heap, which can be visualized as a complete binary tree. Recall that a complete binary tree is completely filled, with the possible exception of the last level which is filled from left to right (Figure 5.20).
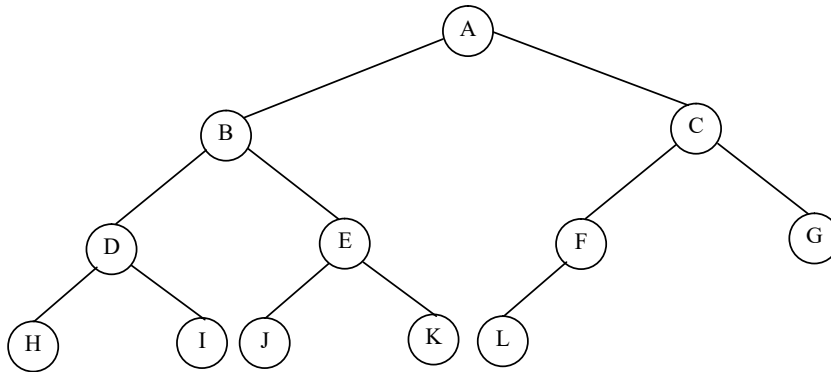
Fig. 5.20   *A Complete Binary Tree*

Heaps can be of two types, namely, *max-heap* and *min-heap*. A max-heap (or descending heap) is a kind of heap in which the value present at any node is greater than or equal to the value of each of its child nodes. On the other hand, a min-heap (or ascending heap) is a kind of heap in which the value present at any node is smaller than or equal to the value of each of its child nodes. The max-heap and min-heap with 12 nodes are shown in Figure 5.21. Note that, the values in the child nodes of a node must not be in order. This mean that sometimes the value in the right child may be more than the value in the left child. At some other times it may be less than the value in left child.
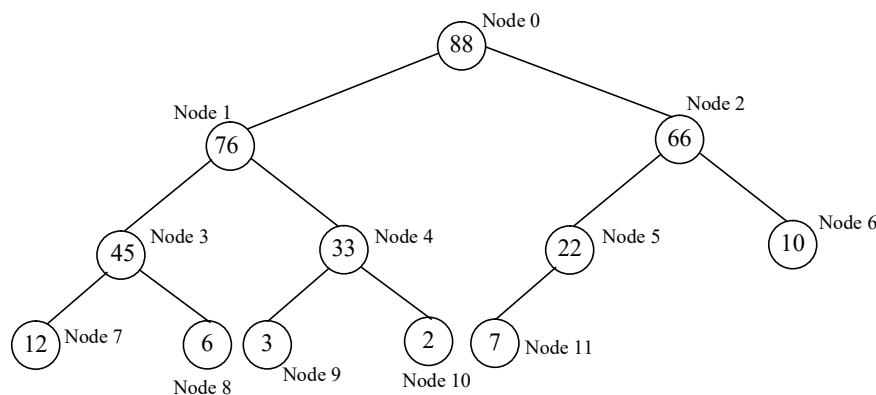
(a) Max-heap
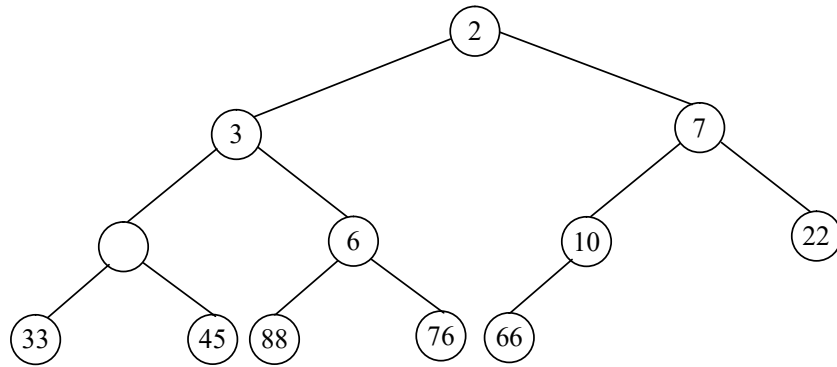
(b) Min-heap

Fig. 5.21   *Types of Heap*

A complete binary tree can be stored most efficiently as a single-dimensional array. In this the root node is stored at `0th` position and its left and right child nodes are stored at `1st` and `2nd` position, respectively. For each `ith` node, the left and right child exist at `(2i+1)th` and `(2i+2)th` position, respectively. The parent node of `ith` node is stored at `((i-1)/2)th` node. For example, in Figure 5.21(a) the left and right child nodes of the `4th` node are stored at `9th` `(2*4+1)` and `10th` `(2*4+2)` position, respectively. The parent node of the `4th` node is stored at `1st` `((4-1)/2)` position. The array representation of the heap shown in Figure 5.21 (a) is as follows:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
88	76	66	45	33	22	10	12	6	3	2	7

To sort an array of size `n` in ascending order using heap sort, the following steps are performed:

1. The initial max-heap is built from the given array.
2. The root element is swapped with the last element in the array.
3. The heap of remaining elements is restored.
4. Steps 2 and 3 are repeated until there are no more elements.

To understand the heap sort, consider an unsorted array `ARR`.

9	11	6	45	22	10	12	90	67	17

First the steps to build a heap out of the given array will be discussed. The elements in the given array are considered one by one. If an element `ARR[i]` is greater than its parent which is stored at location `(i-1)/2`, then the element is swapped with its parent. The element is then compared with its new parent and a swap occurs, if it is greater than its parent. This process continues until no more swapping is needed, or we are at the root node. For example, the steps to construct a heap out of the array `ARR` are as follows:

1. The first element, that is, 9, is stored at position `ARR[0]`.

2. The second element `11` is compared with its parent node which is at location `(i-1)/2`, that is, `(1-1)/2 = 0`. Since `ARR[0]<ARR[1]`, they are swapped. Now `ARR` is as follows:

11	9	6	45	22	10	12	90	67	17

3. The third element 6 is compared with its parent node which is at location `(2-1)/2=0`. Since `ARR[0]>ARR[2]`, they are not swapped. `ARR` remains the same.

4. The fourth element `45` is compared with its parent node which is at location `(3-1)/2=1`. Since `ARR[1]<ARR[3]`, they are swapped. It is again compared with its parent node which is at location `(1-1)/2=0`. Since `ARR[0]<ARR[1]`, they are swapped. Now `ARR` is as follows:

| 45 | 11 | 6 | 9 | 22 | 10 | 12 | 90 | 67 | 17 |

5. The next element 22 is compared with its parent node which is at location `(4-1)/2=1`. Since `ARR[1]<ARR[4]`, they are swapped. It is again compared with its parent node which is at location `(1-1)/2=0`. Since `ARR[0]>ARR[1]`, they are not swapped. `ARR` at this point is as follows:

| 45 | 22 | 6 | 9 | 11 | 10 | 12 | 90 | 67 | 17 |

6. The next element 10 is compared with its parent node which is at location `(5-1)/2=2`. Since `ARR[2]<ARR[5]`, they are swapped. It is again compared with its parent node which is at location `(2-1)/2=0`. Since `ARR[0]>ARR[1]`, they are not swapped. `ARR` at this point is as follows:

| 45 | 22 | 10 | 9 | 11 | 6 | 12 | 90 | 67 | 17 |

7. The seventh element 12 is compared with its parent node which is at location `(6-1)/2=2`. Since `ARR[2]<ARR[6]`, they are swapped. It is again compared with its parent node which is at location `(2-1)/2=0`. Since `ARR[0]>ARR[1]`, they are not swapped. `ARR` at this point is as follows:

| 45 | 22 | 12 | 9 | 11 | 6 | 10 | 90 | 67 | 17 |

8. The next element 90 is compared with its parent node which is at location `(7-1)/2=3`. Since `ARR[3]<ARR[7]`, they are swapped. It is again compared with its parent node which is at location `(3-1)/2=1`. Since `ARR[1]<ARR[3]`, they are swapped. It is again compared with its parent node which is at location `(1-1)/2=0`. Since `ARR[0]<ARR[1]`, they are swapped. `ARR` at this point is as follows:

| 90 | 45 | 12 | 22 | 11 | 6 | 10 | 9 | 67 | 17 |

9. The next element 67 is compared with its parent node which is at location `(8-1)/2=3`. Since `ARR[3]<ARR[8]`, they are swapped. It is again compared with its parent node which is at location `(3-1)/2=1`. Since `ARR[1]<ARR[3]`, they are swapped. It is again compared with its parent node which is at location `(1-1)/2=0`. Since `ARR[0]>ARR[1]`, they are not swapped. `ARR` at this point is as follows:

| 90 | 67 | 12 | 45 | 11 | 6 | 10 | 9 | 22 | 17 |

10. The last element 17 is compared with its parent node which is at location `(9-1)/2=4`. Since `ARR[4]<ARR[9]`, they are swapped. It is again compared with its parent node which is at location `(4-1)/2=1`. Since `ARR[1]>ARR[4]`, they are not swapped. `ARR` at this point is as follows:

90	67	12	45	17	6	10	9	22	11

The initial max-heap for the array `ARR` is shown in Figure 5.22.
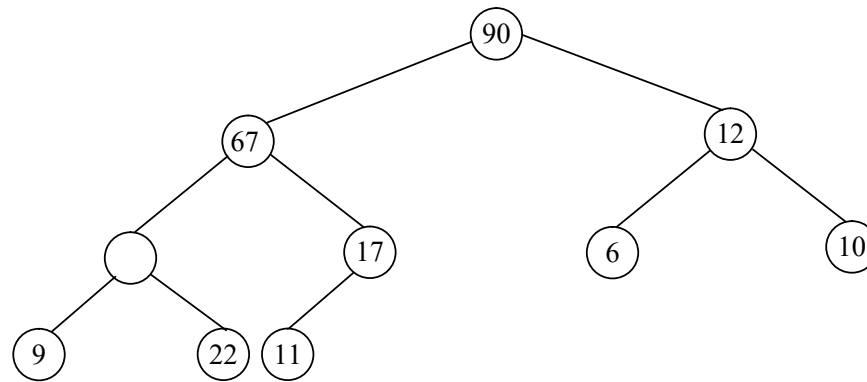
Fig. 5.22 *Initial Max-Heap for ARR*

Once the max-heap is created, the root node is guaranteed to contain the largest element of the list. This element is swapped with the last element in the list. It means the largest element of the list is placed at its proper position. Now, the maximum index value of the array is reduced by one. The array at this point may not be satisfying the properties of heap. Therefore, the heap needs to be restored with remaining elements.

During restoration, the element at the root node is compared with its child node(s), and if it is smaller than its child nodes then it is swapped with the greater of the two child nodes. Now, this element is compared with its current child nodes. It is then, again, swapped with the greater of the two child nodes if it is smaller than its child nodes. This process is repeated until this element is placed at its proper position. At this time, the second largest element is placed at the root node. It is swapped with the second last element in the list. This means that the second largest element is placed at its proper position. Now again, the maximum index value of the array is reduced by one and the process continues until there are no more elements in the heap.

For example, to sort the given array using the heap sort, first the root element (which is 90) is swapped with the last element (which is 11). This moves the largest element to the end of the list. Now the heap is restored with the remaining elements. Since the element at the root node, that is, 11 is smaller than its child nodes, it is swapped with the greater of the two child nodes. Here, it is swapped with 67. The element 11 is still smaller than its two child nodes, it is again swapped with largest of the two child nodes, which is 45. Finally, the element 11 is swapped with 22. The heap after this point is shown in Figure 5.23.
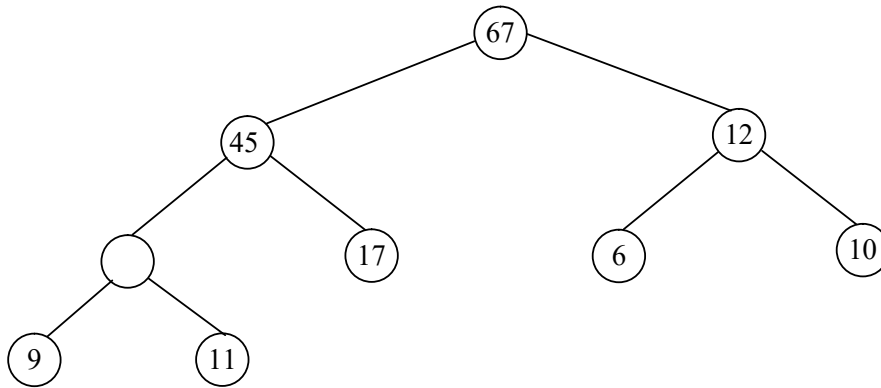
Fig. 5.23 *Heap after Eliminating 90*

At this point, the second largest element (that is, 67) is at the root node. Again 67 is swapped with the second last element in the array (which is, 11). The heap is again restored with the remaining elements. This process is repeated until the array is sorted.

Algorithm 5.8 Heap Sort

```
heap_sort(ARR, size)

1.  Set i = size-1, j = 1
2.  Call make_heap(ARR, size)
3.  Print the initial heap
4.  While (i > 0)
        Set temp = ARR[i]
        Set ARR[i] = ARR[0]
        Set ARR[0] = temp
        Call restore(ARR, i)        //calling restore function to restore the heap
                                    //with remaining elements
        Print Heap after jth pass
        Set j = j + 1
        Set i = i - 1
    End While
5.  Print "No. of passes: ", j-1
6.  End
```

```
make_heap(ARR, size)

//make_heap builds the heap of array ARR

1.  Set i = 1
2.  While (i < size)
        Set child = ARR[i]
        Set k = i
        Set parent = (k-1) / 2;
        While (k>0 AND ARR[parent]<child)
                Set ARR[k] = ARR[parent];
                Set k = parent;
                Set parent = (k-1) / 2;
        End While
        Set ARR[k] = child;
        Set i = i + 1
    End While
3.  End
```

```
restore(ARR, size)

//restoring the heap with remaining elements

1.  Set i = 0
2.  Do
        Set lchild = (2*i+1)
        Set rchild = (2*i+2)
        If (rchild >= size)
                If (lchild < size AND ARR[i] < ARR[lchild])
                        Set temp = ARR[i]
                        Set ARR[i] = ARR[lchild]
                        Set ARR[lchild] = temp
```

```
                End If
                go to step 3
        Else If (ARR[i] < ARR[lchild] OR ARR[i] < ARR[rchild])
                If (ARR[lchild] > ARR[rchild])
                        Set temp = ARR[i]
                        Set ARR[i] = ARR[lchild]
                        Set ARR[lchild] = temp
                        Set i = lchild
                Else
                        Set temp = ARR[i]
                        Set ARR[i] = ARR[rchild]
                        Set ARR[rchild] = temp
                        Set i = rchild
                End If
        Else
                go to step 3
        End If
    While(1)
3.  End
```

Program 5.8: A program to show sorting of an array using heap sort

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototypes*/
void make_heap(int [], int);
void heap_sort(int [], int);
void restore(int [], int);

void main()
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    heap_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d  ", ARR[i]);
    getch();
}

void make_heap(int ARR[], int size)
{
```

```
        int i, k, parent, child;
        for(i=1;i<size;i++)
        {
            child=ARR[i];
            k=i;
            parent=(k-1)/2;
            while(k>0 && ARR[parent]<child)
            {
                ARR[k]=ARR[parent];
                k=parent;
                parent=(k-1)/2;
            }
            ARR[k]=child;
        }
}

void restore(int ARR[], int size)
{
    int i=0, lchild, rchild, temp;
    do
    {
        lchild=(2*i+1);
        rchild=(2*i+2);
        if(rchild>=size)
        {
            if(lchild<size && ARR[i]<ARR[lchild])
            {
                temp=ARR[i];
                ARR[i]=ARR[lchild];
                ARR[lchild]=temp;
            }
            break;
        }
        else if(ARR[i]<ARR[lchild] || ARR[i]<ARR[rchild])
        {
            if(ARR[lchild]>ARR[rchild])
            {
                temp=ARR[i];
                ARR[i]=ARR[lchild];
                ARR[lchild]=temp;
                i=lchild;
            }
            else
            {
```

```
                temp=ARR[i];
                ARR[i]=ARR[rchild];
                ARR[rchild]=temp;
                i=rchild;
            }
        }
        else
            break;
    }while(1);
}


void heap_sort(int ARR[], int size)
{
    int i, j=1, k, temp;
    make_heap(ARR, size);
    printf("\nInitial heap: ");
    for(i=0;i<size;i++)
        printf("%d  ", ARR[i]);
    for(i=size-1;i>0;i—)
    {
        temp=ARR[i];
        ARR[i]=ARR[0];
        ARR[0]=temp;
        restore(ARR, i);             /*rebuilding heap with
remaining elements*/
        printf("\nHeap after %d pass: ", j);
        for(k=0;k<size;k++)
            printf("%d  ", ARR[k]);
        j++;
    }
    printf("\nNo. of passes: %d", j-1);
}



void heap_sort(int ARR[], int size)
{
    int i, j=1, k, temp;
make_heap(ARR, size);
    printf("\nInitial heap: ");
    for(i=0;i<size;i++)
        printf("%d  ", ARR[i]);
    for(i=size-1;i>0;i—)
    {
        temp=ARR[i];
        ARR[i]=ARR[0];
```

```
        ARR[0]=temp;
        restore(ARR, i);    /*rebuilding heap with
    remaining elements*/
        printf("\nHeap after %d pass: ", j);
        for(k=0;k<size;k++)
            printf("%d ", ARR[k]);
        j++;
    }
    printf("\nNo. of passes: %d", j-1);
}
```

The output of the program is

```
Enter the size of the array (max 20): 10

Enter the elements of the array:
9
11
6
45
22
10
12
90
67
17

Initial heap: 90 67 12 45 17 6 10 9 22 11
Heap after 1 pass: 67 45 12 22 17 6 10 9 11 90
Heap after 2 pass: 45 22 12 11 17 6 10 9 67 90
Heap after 3 pass: 22 17 12 11 9 6 10 45 67 90
Heap after 4 pass: 17 11 12 10 9 6 22 45 67 90
Heap after 5 pass: 12 11 6 10 9 17 22 45 67 90
Heap after 6 pass: 11 10 6 9 12 17 22 45 67 90
Heap after 7 pass: 10 9 6 11 12 17 22 45 67 90
Heap after 8 pass: 9 6 10 11 12 17 22 45 67 90
Heap after 9 pass: 6 9 10 11 12 17 22 45 67 90
No of passes: 9
The sorted array is: 6 9 10 11 12 17 22 45 67 90
```

Merge Sort

Merge sort algorithm is based on the fact that it is easier and faster to sort two smaller arrays than one larger array. Therefore, it follows the principle of *divide-and-conquer*. In this sorting, the list is first divided into two halves. The left and right sub-lists obtained are recursively divided into two sub-lists until each sub-list contains not more than one element. The sub-lists containing only one element do

not require any sorting. Therefore, start merging the sub-lists of size one to obtain the sorted sub-list of size two. Similarly, the sub-lists of size two are merged to obtain the sorted sub-list of size four. This process is repeated until we get the final sorted array.

To understand the merge sort algorithm, consider an unsorted array shown here.

18	13	5	20	55	89	4	14

The steps to sort the values stored in the array in ascending order using merge sort are as follows:

1. Initially, `low=0` and `high=7`, therefore, `mid=(0+7)/2=3`. Thus, the given list is divided into two halves from the fourth element. The sub-lists are as follows:

18	13	5	20

55	89	4	14

2. The left sub-list is considered first, and it is again divided into two sub-lists. Now, `low=0` and `high=3`, therefore, `mid=(0+3)/2=1`. Thus, the left sub-list is divided into two halves from the `2nd` element. The sub-lists are as follows:

18	13

5	20

3. These two sub-lists are again divided into sub-lists such that all of them contain one element. Now the sub-lists are as follows:

18

13

5

20

4. Since each sub-list now contains one element, they are first merged to produce the two arrays of size 2. First, the sub-lists containing the elements 18 and 13 are merged to give one sorted sub array, and the sub-lists containing the elements 5 and 20 are merged to give another sorted sub array. The two sorted sub arrays are as follows:

13	18

5	20

5. Now these two sub arrays are again merged to give the following sorted sub array of size 4:

5	13	18	20

6. After sorting the left half of the array, perform the same steps for the right half. The sorted right half of the array is as follows:

4	14	55	89

7. Finally, the left and right halves of the array are merged to give the sorted array as follows:

4	5	13	14	18	20	55	89

Algorithm 5.9 Merge Sort

```
merge_sort(ARR, low, high)

1.  If (low < high)
        Set mid = (low + high) / 2
        Call merge_sort(ARR, low, mid)  //calling merge_sort recursively for left sub list
        Call merge_sort(ARR, mid+1, high) //calling merge_sort for right sub list
        Call merging(ARR, low, mid, mid+1, high)
    End If
2.  End
```

```
merging(ARR, ll, lr, ul, ur)

//merging merges the two sub arrays to produce a sorted array named merged

//ll and ul are the lower bounds of left and right sub list respectively.

//ul and ur the upper bounds of left and right sub list respectively.

1.  Set i = ll, j = ul, k = ll
2.  While(i <= lr AND j <= ur)
        If(ARR[i] <= ARR[j])
                Set merged[k] = ARR[i]
                Set i = i + 1
        Else
                Set merged[k] = ARR[j]
                Set j = j + 1
        End If
        Set k= k + 1
    End While
    If(i <= lr)
        While(i <= lr)
                Set merged[k] = ARR[i]
                Set i = i + 1
                Set k = k + 1
        End While
    End If
    If(j <= ur)
        While(j <= ur)
                Set merged[k] = ARR[j]
                Set j = j + 1
                Set k = k+ 1
        End While
    End If
    Set k = ll
    While (k <= ur)
        Set ARR[k] = merged[k]
        Set k = k + 1
    End While
3.  End
```

Program 5.9: A program to show sorting of an array using merge sort

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20


/*Function prototypes*/
void merging(int [], int, int, int, int);
void merge_sort(int [], int, int);


void main()
{
    int ARR[MAX], i, size;
    do
    {
```

```
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    merge_sort(ARR, 0, size-1);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d  ", ARR[i]);
    getch();
}


void merge_sort(int ARR[], int low, int high)
{
    int mid, k;
    if(low<high)
    {
        mid=(low+high)/2;
        merge_sort(ARR, low, mid);
        merge_sort(ARR, mid+1, high);
        merging(ARR, low, mid, mid+1, high);
    }
}


void merging(int ARR[], int ll, int lr, int ul, int ur)
{
    int i, j, k, merged[MAX];
    i=ll;
    j=ul;
    k=ll;
    while(i<=lr && j<=ur)
    {
        if(ARR[i]<=ARR[j])
        {
            merged[k]=ARR[i];
            i++;
        }
        else
        {
            merged[k]=ARR[j];
            j++;
```

```
        }
        k++;
    }
    if(i<=lr)
        while(i<=lr)
        {
            merged[k]=ARR[i];
            i++;
            k++;
        }
    if(j<=ur)
        while(j<=ur)
        {
            merged[k]=ARR[j];
            j++;
            k++;
        }
    for(k=ll;k<=ur;k++)
        ARR[k]=merged[k];
}
```

The output of the program is

```
Enter the size of the array (max 20): 10

Enter the elements of the array:
65
12
45
78
96
32
56
44
25
11

The sorted array is: 11  12  25  32  44  45  56  65  78  96
```

Quick Sort

Quick sort algorithm also follows the principle of *divide-and-conquer*. However, it does not simply divide the list into halves. Rather it first picks up a partitioning element, called pivot that divides the list into two sub-lists. This is done in a never that all the elements in the left sub-list are smaller than the pivot, and all the elements in the right sub-list are greater than the pivot. The same process is applied on the

left and right sub-lists separately. This process is repeated recursively until each sub-list contains not more than one element.

The main task in quick sort is to find the pivot that partitions the given list into two halves so that the pivot is placed at its appropriate location in the array. The choice of pivot has a significant effect on the efficiency of quick sort algorithm. The simplest way is to choose the first element as pivot. However, the first element is not a good choice, especially if the given list is already or nearly ordered. For better efficiency, the middle element can be chosen as a pivot. Note that, the first element is as taken as pivot for simplicity.

The steps involved in quick sort algorithm are as follows:

1. Initially, three variables `pivot`, `beg` and `end` are taken, such that both `pivot` and `beg` refer to the `0th` position, and `end` refers to `(n-1)th` position in the list.

2. Starting with the element referred to by `end`, the array is scanned from right to left, and each element on the way is compared with the element referred to by `pivot`. If the element referred to by `pivot` is greater than the element referred to by `end`, they are swapped and step 3 is performed. Otherwise, `end` is decremented by 1 and step 2 is continued.

3. Starting with the element referred to by `beg`, the array is scanned from left to right, and each element on the way is compared with the element referred to by `pivot`. If the element referred to by `pivot` is smaller than the element referred to by `end`, they are swapped and step 2 is performed. Otherwise, `beg` is incremented by 1 and step 3 is continued.

The first pass terminates when `pivot`, `beg` and `end` all refer to the same array element. This indicates that the pivot element is placed at its final position. The elements to the left of this element are smaller than this element, and elements to its right are greater.

To understand the quick sort algorithm, consider an unsorted array:

8	33	6	21	4

The steps to sort the values stored in the array in ascending order using quick sort are as follows:

1. Initially, the index 0 in the list is chosen as the pivot, and the index variables beg and end are initialized with index 0 and *n*–1 respectively.

8	33	6	21	4

2. The scanning of elements is started from the end of the list. ARR[pivot] (that is, 8) is greater than ARR[end] (that is, 4). Therefore, they are swapped.

8	33	6	21	4

↑ pivot beg   ↑ end

3. Now, the scanning of elements is started from the beginning of the list. Since ARR[pivot] (that is, 8) is greater than ARR[beg] (that is 33), therefore beg is incremented by 1, and the list remains unchanged.

4	33	6	21	8

beg   pivot
end

4. Next, the element ARR[pivot] is smaller than ARR[beg], they are swapped.

4	8	6	21	33

pivot   end
beg

5. Again, the list is scanned from right to left. Since, ARR[pivot] is smaller than ARR[end], therefore the value of end is decremented by 1, and the list remains unchanged.

4	8	6	21	33

pivot   end
beg
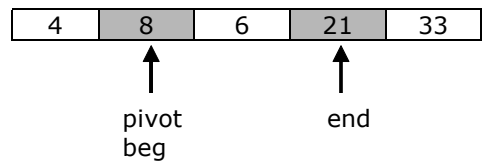
6. Next, the element ARR[pivot] is smaller than ARR[end], value of end is decremented by 1, and the list remains unchanged.

4	8	6	21	33

pivot end
beg
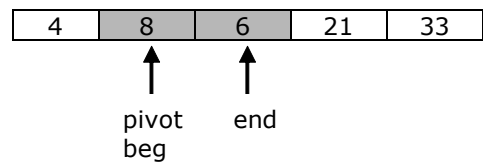
7. Now, ARR[pivot] is greater than ARR[end], they are swapped.

4	6	8	21	33

beg   pivot
end
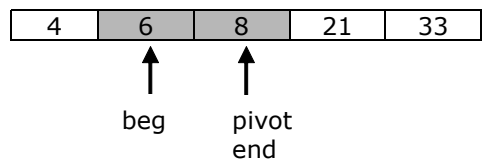
8. Now, the list is scanned from left to right. Since, ARR[pivot] is greater than ARR[beg], value of beg is incremented by 1. The list remains unchanged.

4	6	8	21	33
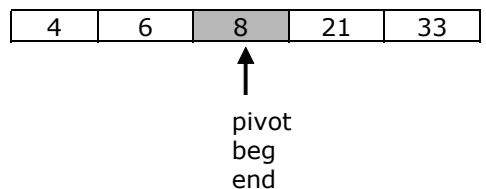
pivot
beg
end

At this point since the variables `pivot`, `beg` and `end` all refer to the same element, the first pass is terminated and the value 8 is placed at its

appropriate position. The elements to its left are smaller than 8, and elements to its right are greater than 8. The same process is applied on the left and right sub-lists.

Algorithm 5.10 Quick Sort

```
quick_sort(ARR, size, lb, ub)

1.  Set i = 1              //i is a static integer variable
2.  If lb < ub
        Call splitarray(ARR, lb, ub)      //returning an integer value pivot
        Print ARR after iᵗʰ pass
        Set i = i + 1
        Call quick_sort(ARR, size, lb, pivot – 1)      //recursive call to quick_sort() to
                                                       //sort left sub list

        Call quick_sort(ARR, size, pivot + 1, ub);     //recursive call to quick_sort() to
                                                       //sort right sub list

    Else if (ub=size-1)
        Print "No. of passes: ", i
    End If
3.  End
```

```
splitarray(ARR, lb, ub)

//spiltarray partitions the list into two sub lists such that the elements in left sub list are
smaller than ARR[pivot], and elements in the right sub list are greater than ARR[pivot]

1.  Set flag = 0, beg = pivot = lb, end = ub
2.  While (flag != 1)
        While (ARR[pivot] <= ARR[end] AND pivot != end)
                Set end =  end – 1
        End While
        If pivot = end
                Set flag = 1
        Else
                Set temp = ARR[pivot]
                Set ARR[pivot] = ARR[end]
                Set ARR[end] = temp
                Set pivot = end
        End If
        If flag != 1
                While (ARR[pivot] >= ARR[beg] AND pivot != beg)
                        Set beg =  beg + 1
                End While
                If pivot = beg
                        Set flag = 1
                Else
                        Set temp = ARR[pivot]
                        Set ARR[pivot] = ARR[beg]
                        Set ARR[beg] = temp
                        Set pivot = beg
                End If
        End If
    End While
3.  Return pivot
4.  End
```

Program 5.10: A program to show sorting of an array using quick sort

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20


/*Function prototypes*/
void quick_sort(int [], int, int, int);
int splitarray(int [],  int, int);


void main()
```

```
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    quick_sort(ARR, size, 0, size-1);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d  ", ARR[i]);
    getch();
}

void quick_sort(int ARR[], int size, int lb, int ub)
{
    int pivot, k;
    static int i=0;
    if (lb<ub)
    {
        pivot=splitarray(ARR, lb, ub);
        printf("\nArray after pass %d: ", i+1);
        for (k=0;k<size;k++)
            printf("%d ", ARR[k]);
        i++;
        quick_sort(ARR, size, lb, pivot-1);
/*recursive call to function to sort left sub-list*/
        quick_sort(ARR, size, pivot+1, ub);
/*recursive call to function to sort right sub-list*/
    }
    else if (ub==(size-1))
        printf("\nNo. of passes: %d", i);

}
int splitarray(int ARR[], int lb, int ub)
{
    int pivot, beg, end, temp, flag=0;
    beg=pivot=lb;
    end=ub;
    while(!flag)
```

```
        {
            while ((ARR[pivot]<=ARR[end]) && (pivot!=end))
                end—;
            if (pivot==end)
                flag=1;
            else
            {
                temp=ARR[pivot];
                ARR[pivot]=ARR[end];
                ARR[end]=temp;
                pivot=end;
            }
            if (!flag)
            {
                while ((ARR[pivot]>=ARR[beg]) && (pivot!=beg))
                    beg++;
                if (pivot==beg)
                    flag=1;
                else
                {
                    temp=ARR[pivot];
                    ARR[pivot]=ARR[beg];
                    ARR[beg]=temp;
                    pivot=beg;
                }
            }
        }
        return pivot;
    }
```

The output of the program is

```
Enter the size of the array (max 20): 5

Enter the elements of the array:
6
5
4
3
2

Array after pass 1: 2 5 4 3 6
Array after pass 2: 2 5 4 3 6
Array after pass 3: 2 3 4 5 6
Array after pass 4: 2 3 4 5 6
No. of passes: 4
The sorted array is: 2  3  4  5  6
```

Bucket/Radix Sort

The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left. In the first pass, the numbers are sorted according to the digits at units place. In the second pass, the numbers are sorted according to the digits at tens place, and so on. Since the base of decimal numbers is 10, the radix sort requires ten buckets, numbered 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 to store the array elements in each pass. The number of passes in the algorithm is equal to the number of digits in the largest number. Therefore, the algorithm first determines the largest number in the list and counts the number of digits in it.

In the first pass, the numbers having 0 at units place are placed in bucket 0. Numbers having 1 at their units place are placed in bucket 1, numbers having 2 at their units place are placed in bucket 2 and so on. The elements are then retrieved from these buckets (starting from bucket 0 till bucket 9) and copied in the original array. At this point, the numbers are sorted based on the digits at units place. This list becomes the input for the second pass. In the second pass, the numbers having 0 at their tens place are placed in bucket 0. Numbers having 1 at their tens place are placed in bucket 1, numbers having 2 at their tens place are placed in bucket 2 and so on. The elements are then retrieved from these buckets (starting from bucket 0 till bucket 9) and copied in the original array. At this point, the numbers are sorted based on the digits at tens place. Now, this array becomes the input for the third pass. This process is repeated d times, where d represents the number of digits in the largest number of the list.

To understand the radix sort algorithm, consider an unsorted array.

318	233	56	899	912	674	555	110	21	746

Since the largest element (that is, 899) consists of three digits, the array will be sorted in three passes.

First pass

In the first pass, the digits at the units place are considered, and the elements are placed in the corresponding buckets as follows:

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
110	21	912	233	674	555	56 746		318	899

Now, the elements are retrieved from each bucket and copied into the original array. The array now becomes:

110	21	912	233	674	555	56	746	318	899

Second pass

In the second pass, the digits at the tens place are considered, and the elements are placed in the corresponding buckets as follows:

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
	1**1**0 9**1**2 3**1**8	**2**1	2**3**3	7**4**6	55**5** **5**6		6**7**4		8**9**9

Now, the elements are retrieved from each bucket and copied into the original array. The array now becomes:

110	912	318	21	233	746	555	56	674	899

Third pass

In the third pass, the digits at the hundredth place are considered, and the elements are placed in the corresponding buckets as follows:

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
021 **0**56	**1**10	**2**33	**3**18		**5**55	**6**74	**7**46	**8**99	**9**12

Now, the elements are retrieved from each bucket and copied into the original array. After this pass, the array is sorted as follows:

21	56	110	233	318	555	674	746	899	912

Algorithm 5.11 Radix Sort

```
bucket_sort(ARR, size)

1.  Set i = 1, digitcount = 0, divisor = 1, t = 0, largest = ARR[0]
2.  While (i < size)
        If (ARR[i] > largest)
                Set largest = ARR[i]
        End If
        Set i = i + 1
    End While
3.  While (largest > 0)
        Set largest = largest / 10
        Set digitcount = digitcount + 1
    End While
4.  Set i = 0
5.  While (i < digitcount)
        Set k = 0
        While (k < 10)
                Set buckcount[k] = 0
                Set k = k + 1
        End While
        Set j = 0
        While (j < size)
                Set r = (ARR[j] / divisor) % 10
                Set bucket[buckcount[r]][r] = ARR[j]
                Set buckcount[r] = buckcount[r] + 1
                Set j = j + 1
        End While
        Set t = 0, j = 0
        While (j < 10)
                Set k =0
                While (k < buckcount[j])
                        Set ARR[t] = bucket[k][j]
                        Set t = t + 1
                        Set k = k + 1
                End While
                Set j = j + 1
        End While
        Print ARR after i^th pass
        Set divisor = divisor * 10
        Set i = i + 1
    End While
6.  Print "No. of passes: ", digitcount
7.  End
```

Program 5.11: A program to show sorting of an array using radix sort

```c
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
void bucket_sort(int [], int);

void main()
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    bucket_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d ", ARR[i]);
    getch();
}

void bucket_sort(int ARR[], int size)
{
    int bucket[MAX][10], buckcount[10];
    int i, j, k, r, digitcount=0, divisor=1, largest,
t=0;
    largest=ARR[0];
    for(i=1;i<size;i++)
    {
        if(ARR[i]>largest)
            largest=ARR[i];
    }
    while(largest>0)
    {
        largest/=10;
        digitcount++;
    }
    for(i=0;i<digitcount;i++)
    {
```

```
        for(k=0;k<10;k++)
            buckcount[k]=0;
        for(j=0;j<size;j++)
        {
            r=(ARR[j]/divisor)%10;
            bucket[buckcount[r]++][r]=ARR[j];
        }
        t=0;
        for(j=0;j<10;j++)
            for(k=0;k<buckcount[j];k++)
            {
                ARR[t++]=bucket[k][j];
            }
        printf("\nArray after pass %d: ",i+1);
        for(j=0;j<size;j++)
            printf("%d ", ARR[j]);
        divisor*=10;
    }
    printf("\nNo. of passes: %d ", digitcount);
}
```

The output of the program is

```
Enter the size of the array (max 20): 10

Enter the elements of the array:
318
233
56
899
912
674
555
110
21
746

Array after pass 1: 110 21 912 233 674 555 56 746 318 899
Array after pass 2: 110 912 318 21 233 746 555 56 674 899
Array after pass 3: 21 56 110 233 318 555 674 746 899 912
No. of passes: 3
The sorted array is: 21 56 110 233 318 555 674 746 899 912
```

Comparison of Various Sorting Algorithms

In order to choose an appropriate algorithm that suits a particular problem, an analysis of algorithms is necessary. For this, a number of efficiency parameters are

considered. Two most important efficiency parameters are the time required to execute the algorithm and the amount of memory space it requires.

Here, the efficiency of an algorithm in terms of the amount of time required in its execution will be discussed. The estimated amount of time required in executing an algorithm is referred to as the time complexity (or time efficiency) of the algorithm. Time does not mean the number of time units (seconds or minutes) required to execute the algorithm. Rather, the time efficiency in terms of the number of significant operations performed by the algorithm is measured.

In sorting algorithms, several operations are performed, such as comparison operation (that compares two values to determine which is smaller or larger), interchange (swap) operation, increment of an index variable in a loop, etc. However, comparison and interchange operations are the most significant operations as they require much more time than any other simple operation. Moreover, the number of interchanges cannot be greater than the number of comparisons. Therefore, a number of comparisons is considered as a useful measure of a sort's time efficiency.

In determining the time complexity of an algorithm, the size of an instance (input) also plays an important role. If the size of an instance is `n`, then the time complexity of the algorithm is some function of `n`. Thus, we need to determine a function `f(n)` that relates the number of operations to be performed to the size of the input. While comparing any two sorting algorithms, the algorithm whose function grows slowly than the other is considered to be better. In mathematical terms, this relation is represented in `Big Oh` notation. In this notation, a function `f(n)` is `O(g(n))`, if there exists positive integers `k` and `c` such that `f(n)<=c*g(n)`, for all `n>=k` (where, `f(n)` and `g(n)` are the functions of two different algorithms). The expression `O` is also called Landau's symbol.

Using the concept of `Big Oh` notation, we can compare various sorting algorithms and classify them as *good* or *bad* in general terms. If the algorithm has `O(n)` complexity, it implies that it has a linear complexity and it grows linearly with the size of the input. For example, consider an algorithm that takes `t` time units to sort an array of `10` elements. Then it will take `10*t` time units for `100` elements (`10*10=100`). Unfortunately, no such sorting algorithm exists. Generally, the complexities of most of the sorting algorithms range between `O(n2)` and `O(n log n)`. `O(n2)` is known as the polynomial complexity and `O(n log n)` is known as the logarithmic complexity. If the size of input is `n`, then `O(n log n)` is significantly faster than `O(n2)` as shown in Table 5.1.

Table 5.1   *Comparing O(n log n) with O(n²)*

N	n²	N log n
10	100	33
100	10000	665
1000	10^6	10^4
10^6	10^{12}	$2*10^7$
10^9	10^{18}	$3*10^{10}$

Analysis of Insertion Sort

In the worst case, when the input list is in descending order, the first pass of insertion sort requires one comparison, second pass requires two comparisons,…, `ith` pass requires `i` comparisons, and the last pass requires `(n-1)` comparisons. Therefore, complexity of insertion sort algorithm is as follows:

```
f(n)   = 1 + 2 + 3 + … + (n-i) + … + (n-3) + (n-2) + (n-
1)
    = n(n-1)/2
    = (n2-n)/2
    = O(n2)
```

Analysis of Bubble Sort

To sort a list containing `n` elements atmost `n-1` passes are required. The first pass requires `n-1` comparisons, second pass requires `n-2` comparisons, …, `ith` pass requires `n-i` comparisons. Therefore, average complexity of bubble sort algorithm is:

```
f(n)   = (n-1) + (n-2) + (n-3) + … + (n-i) + … +3 + 2 + 1
    = n(n-1)/2
= (n2-n)/2
```

Since for all n, $(n^2-n)/2$ is always less than n^2, the time complexity of bubble sort algorithm is $O(n^2)$. Note that under best-case conditions (when the list is almost or completely sorted), the bubble sort can approach the `O(n)` level of complexity. In other cases, the complexity level is `O(n2)`.

Analysis of Selection Sort

Selection sort also requires `n-1` passes to sort an array of `n` elements. The first pass requires `n-1` comparisons, second pass requires `n-2` comparisons,…, `ith` pass requires `n-i` comparisons, and the last pass requires only one comparison. Therefore, average complexity of selection sort algorithm is as follows:

```
f(n)   = (n-1) + (n-2) + (n-3) + … + (n-i) + … +3 + 2 + 1
    = n(n-1)/2
= (n2-n)/2
    = O(n2)
```

Analysis of Heap Sort

A complete binary tree with `n` nodes has depth of `log n`. Therefore, building the initial heap of `n` elements requires `n* log n` comparisons, since inserting each element requires at the most `log n` comparisons. After the creation of initial heap, the element at the root node is swapped with the last element and the heap is restored. During restoration, moving an element one level down requires at the most four comparisons, and you know that tree levels cannot exceed `log n`. Therefore, restoring requires at the most `4*log n` comparisons to find out the position of the element in the tree. This process is repeated n times, which means that at the most `4*n*log n` comparisons are required. Therefore, the complexity of heap sort is `O(n log n)`.

Analysis of Merge Sort

In the first pass of merge sort algorithm, the given array is divided into two halves and each half is sorted separately. In each of the recursive calls to the `merge_sort()`, one for left half and one for right half, the array is further divided into two halves. This results in four segments of the array. Thus, in each pass, the number of segments of the array gets doubled until each segment contains not more than one element. Therefore, the total number of divisions is `log n`. Moreover, in any pass, at the most `n` comparisons are required. Hence, the complexity of the merge sort algorithm is `O(n log n)`.

Analysis of Quick Sort

The quick sort algorithm gives worst case performance when the list is already sorted. In this case, the first element requires `n` comparisons to determine that it remains in the first position, second element requires `n-1` comparisons to determine that it remains in the second position and so on. Therefore, the total number of comparisons in this case is as follows:

```
f(n)    = n + (n-1) + … + 3 + 2 +1
        = n(n+1)/2
        = O(n2)
```

Note that in the worst case, the complexity of quick sort algorithm is equal to the complexity of bubble sort algorithm. In the best case when pivot is chosen in such a way that it partitions the list approximately in half, then there will be `log n` partitions. Each pass performs a maximum of `n` comparisons. Therefore, complexity of quick sort algorithm in this case is as follows:

```
f(n)    = n * log n
        = O(n log n)
```

Analysis of Bucket/Radix Sort

The radix sort requires `d` passes, where `d` is the number of digits in the largest element in the given array. Thus, the total number of comparisons is as follows:

```
f(n)    = d * n * 10
```

where, `n` = number of elements in the array and constant `10` denotes the base of decimal number system.

Though the number of passes `(d)` in the radix sort is independent of the number of elements, still if `d=n`, then in the worst case, time complexity of radix sort is as follows:

```
f(n)    = n * n * 10
        = 10n2
        = O(n2)
```

On the other hand, if `d = log n` then in the best case, the time complexity of radix sort is as follows:

```
f(n)    = (log n) * n * 10
        = (n log n) * 10
        = O(n log n)
```

External Sorting

Many applications require the data stored in files to be sorted. However, if the file is too large and cannot fit in the main memory at once, no internal sorting technique can be applied directly. In this situation, external sorting also known as file sort is useful. In external sorting, some part of the file is read into the main memory, sorted using any internal sort technique and written back to the disk in a separate file (sub-file). This process continues till all the data in the original file gets sorted. After this, these sub-files are merged together to produce a single sorted file. For example, assume that there is a file of N records that must be sorted and only $N/2$ records can be loaded in the memory at one time. All the records of this file are read in the main memory in two turns, and in each turn, $N/2$ records are sorted and a separate sub-file is created. These two sub-files are then merged to form a single sorted file (Figure 5.24).
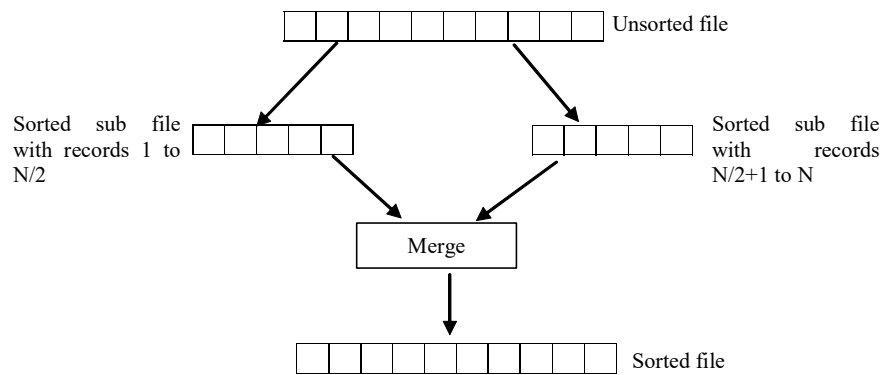
Fig. 5.24 *External Sorting*

Run Generation

In this phase, the records in the file to be sorted are divided into several groups, called runs such that each run can fit in the available buffer space. An internal sort is applied to each run and the resulting sorted runs are written back to the disk as temporarily sorted runs. The number of initial runs (ni) is computed as ébF/ Mù, where bF is the number of blocks containing records of file F and M is the size of available buffer in blocks. If M = 5 blocks and bF = 20 blocks, then ni=4 each of size 5 blocks. Thus, after the sorting phase, 4 runs are stored on the disk as temporarily sorted runs.

K-Way Merge

In this phase, the sorted runs created during the sort phase are merged into larger runs of sorted records. The merge continues until all the records are in one large run. The output of merge phase is the sorted file. If n_i>M, it is not possible to allocate a buffer for each sorted run in one pass. In this case, the merge operation is carried out in multiple passes. In each pass, M–1 buffer blocks are used as input buffers which are allocated to M–1 input runs and one buffer block is kept for holding the merge result. The number of runs that can be merged together in each pass is known as degree of merging (d_M). Thus, the value of d_M is the smaller of

(M–1) or n_i. The output of merge phase is the sorted relation. If two runs are merged in each pass, it is known as two-way merging. In general, if k runs are merged in each pass, it is known as k-way merging.

An example of external sort–merge is given in Figure 5.25.
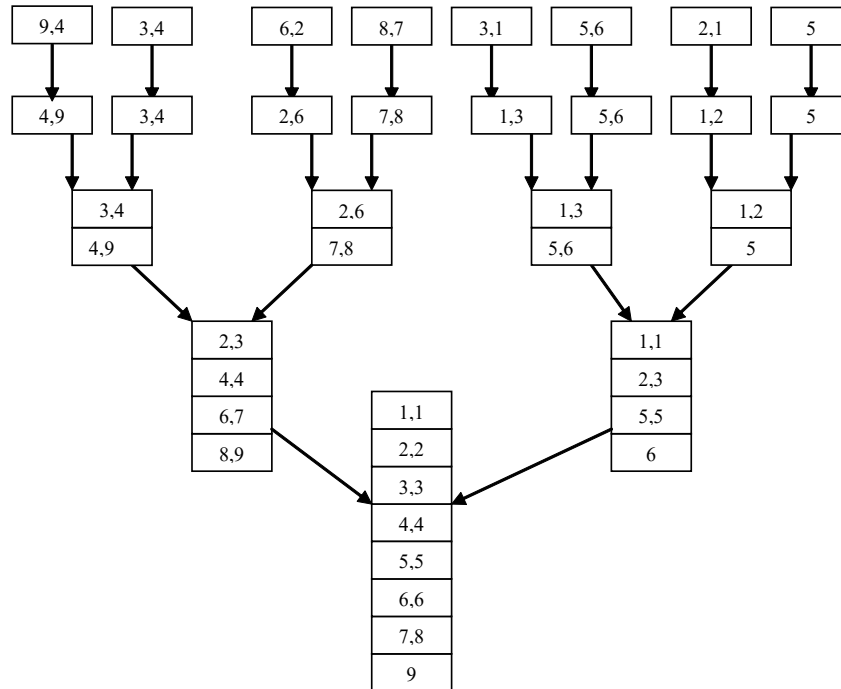
Fig. 5.25   *An Example of External Sort with k=2*

In Figure 5.25, two input buffers and one output buffer. Records from input buffer have been taken must be sorted before they can be merged to the output buffer. The output buffer is written to create larger runs out of the given runs. Merging requires several passes as shown in Figure 5.25. Merging of all input files into a single file requires four passes.

Algorithm 5.12: External Sort

```
createrun (ARR, str, size)
1.  Set i= 0
2.  Set FP = fopen(str, "w")      // open file in write mode using file pointer FP
3.  While (i < size)
            Store elements of ARR in FP
4.  End While
5.  End
```

```
mergerun (fnames, countrun)
1.  Set j= n1= n2 = 0
2.  Set str = "tempA.dat"
3.  If (countrun = 1)
        Set FP=fopen(fnames[0],"r")    //open file in read using FILE pointer FP
        While (FP! = EOF)
                Print element
        End while
        Fclose(FP)                          // close file
        End If
4.  While (countrun > 1)
        Set flag1 = flag2 = 1
        Set FP = fopen(str, "w")
        Set F1 = fopen(fnames[0], "r")
        Set F2 = fopen(fnames[1], "r")
```

```
                    While (F1! = NULL && F2! = NULL)
                            If (flag1= 1)
                                    If (F1= EOF)
                                            Store all elements of F2 into FP
                                    End If
                            End If
                            If (flag2 = 1)
                                    If (F2= EOF)
                                            Store all elements of F1 into FP
                                    End If
                            End If
                            If (n1 < n2)
                                    Store element of F1 into FP
                                    Set flag1 =1
                                    Set flag2 =0
                            End If
                            Else
                                    Store elements of F2 into FP
                                    Set flag1=0
                                    Set flag2=1
                            End If
                    End While
            fcloseall()                            //close all files
            remove (fnames[0])                     // remove F1
            remove (fnames[1])                     // remove F2
            Place str at 1st location of fnames
            Shift elements of fnames to one position left from 2nd element
            Set countrun = countrun – 1
            End While
    5.  fcloseall();
    6.  Set FP= fopen(fnames[0], "r")        // open merged file stored at fnames
    7.  Print all sorted elements
    8.  fclose(FP)
    9.  End
```

Program 5.12: A program to perform external sort

```c
#include<stdio.h>

#include<string.h>

#include<conio.h>

#define MAX 10


/*Function prototypes*/

void mergerun(char [][10], int);

void internalsort(int [], int);

void createrun(int [], char [], int);

void display(char []);


void main()

{

    int i=0, flag=0, n, countrun;

    int ARR[MAX];

    char fnames[5][10];

    char str[]="Filex.dat", t='A';

    char str1[]= "File1.dat";

    FILE *FP, *FP1;
```

```
    clrscr();
    printf("File before sorting:\n");
    display(str1);
    FP=fopen("File1.dat","r");
    while(1)
    {
        for(i=0;i<MAX;i++)
        {
            if(fscanf(FP,"%d",&ARR[i])==EOF)
            {
                flag=1;
                break;
            }
        }
        if(flag==1 && i==0)
            break;
        internalsort(ARR, i);
        str[4]=t;
        createrun(ARR,str,i);
        t=t+1;
    }
    countrun=t-65;
    t='A';
    for(i=0;i<countrun;i++)
    {
        str[4]=t;
        strcpy(fnames[i],str);
        t=t+1;
    }
    mergerun(fnames, countrun);
    getch();
}

void internalsort(int ARR[], int size)
{
    int i=0,j,temp, flag=1;
    while(i<size && flag)
    {
        flag= 0;
        for(j=0;j<size-i-1;j++)
            if(ARR[j]>ARR[j+1])
```

```
            {
                flag=1;
                temp=ARR[j];
                ARR[j]=ARR[j+1];
                ARR[j+1]=temp;
            }
        i++;


    }
}


void createrun(int ARR[], char str[], int size)
{
    int i;
    FILE *FP;
    FP=fopen(str,"w");
    for(i=0;i<size;i++)
    {
        fprintf(FP,"%d\n",ARR[i]);
    }
    fclose(FP);
}


void display(char str1[])
{
    int n1;
    FILE *FP;
    FP=fopen(str1,"r");
    while(fscanf(FP,"%d",&n1)!=EOF)
        printf("%d\t",n1);


}


void mergerun(char fnames[][10], int countrun)
{
    int j;
    int n1, n2, flag1, flag2;
    char str[]="tempx.dat",t='A';
    FILE *FP, *F1, *F2;
    if(countrun==1)
        {
```

```
        FP=fopen(fnames[0],"r");

    while(fscanf(FP,"%d",&n1)!=EOF)

        printf("%d\n",n1);

    fclose(FP);

    return;

}

while(countrun>1)

{

    flag1=1;

    flag2=1;

    str[4]=t;

    t=t+1;

    FP=fopen(str,"w");

    F1=fopen(fnames[0],"r");

    F2=fopen(fnames[1],"r");

        while(F1!=NULL || F2!=NULL)

        {

            if(flag1)

                if(fscanf(F1,"%d",&n1)==EOF)

                {

                        fprintf(FP,"%d\n",n2);

                        while(fscanf(F2,"%d",&n2)!=EOF)

                        {

                                fprintf(FP,"%d\n",n2);

                        }

                        break;

                }

            if (flag2)

                if(fscanf(F2,"%d",&n2)==EOF)

                {

                        fprintf(FP,"%d\n",n1);

                        while(fscanf(F1,"%d",&n1)!=EOF)

                        {

                                fprintf(FP,"%d\n",n1);

                        }

                        break;

                }

            if(n1<n2)

            {

                fprintf(FP,"%d\n",n1);

                flag1=1;
```

```
                    flag2=0;
                }
                else
                {
                    flag1=0;
                    flag2=1;
                    fprintf(FP,"%d\n",n2);
                }
            }
            fcloseall();
            remove(fnames[0]);
            remove(fnames[1]);
            strcpy(fnames[0],str);
            for(j=1;j<countrun-1;j++)
            {
                strcpy(fnames[j],fnames[j+1]);
            }
            countrun-=1;
        }
        fcloseall();
        FP=fopen(fnames[0],"r");
        printf("\n");
        printf("\nFile after sorting:\n");
        while(fscanf(FP,"%d",&n1)!=EOF)
            printf("%d\t",n1);
        fclose(FP);
    }
```

The output of the program is

```
File before sorting:
14      19      11      16      24      17      34      23
27      21
37      15      18      26      56      45      47      29
33      47
42      67


File after sorting:
11      14      15      16      17      18      19      21
23      24
26      27      29      33      34      37      42      45
47      47
56      67
```

The preceding program uses k-way merge for k= 2. It means two runs are being merged in each pass. Instead we can also merge more than two runs in a single pass. In case, four runs are merged in a single pass, it is a k-way merge with k=4. The example given in Figure 5.26 illustrates this.
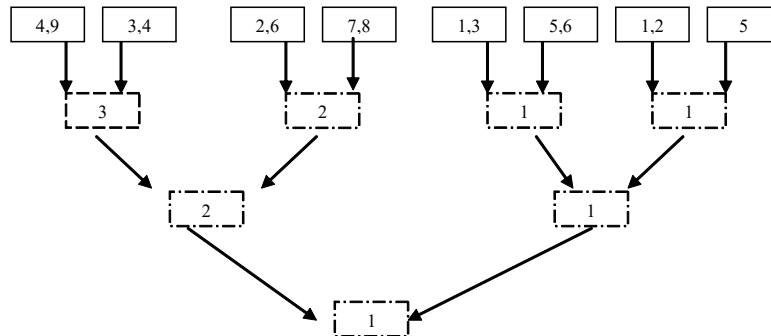
Fig. 5.26 *An Example of k-way Merge with k=4*

Check Your Progress

17. Define sorting.

18. Differentiate between internal sorting and external sorting.

19. What is an important property of bubble sort algorithm?

20. What steps are required to sort an array of size n in ascending order using heap sort?

21. What do you know about restoration?

22. What principle is the merge cost based on?

23. What is the partitioning element in quick sort known as?

24. What does the radix or bucket sort algorithm do?

25. What are the two parameters that are considered while analysing the algorithms?

26. Define time complexity of an algorithm.

27. What is the first pass of merge sort algorithm?

5.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A linear search is an algorithm which searches for any particular target key in a data set.

2. A hash table is a simple data structure that offers fast lookup, insertion and deletion of pairs.

3. A hash table is used for storing records of data, in either an array or a file.

4. A hash function is a function that takes an element of whatever data type you are storing (integer, string, etc.) and outputs an integer in a certain range.

5. The two types of hashing functions are as follows:

a. Distribution-independent function

b. Distribution dependent function

6. An algorithm must ensure that:

    a. It always generates the same random value for a given key.

    b. No two keys yield the same random value.

7. A 'collision' is a condition which occurs when two or more keys produce the same hash location.

8. The two methods for detecting collisions and overflows in a static hash table are as follows:

a. Linear open addressing

b. Chaining

9. The only major drawback of quadratic hashing is the need to ensure that the table is no more than half full (i.e., $1 < 0.5$).

10. A trie is a tree in which branching is calculated not by the entire key value but by a portion of it.

11. Extendible hashing is designed to retrieve the data associated with a given key with two disk accesses.

12. A file organization in which the records are sorted, based on the value of one of its fields is called sequential file organization.

13. Records in random file organization are not stored sequentially. Instead, each record is mapped to an address on the disk on the basis of its key value. One such technique used for this mapping of record to an address is called hashing.

14. The process of finding the occurrence of a particular data item in a list is known as searching. The various search techniques are linear search, binary search, Fibonacci search and interpolation search.

15. The searching technique suitable for unsorted arrays is linear search.

16. Linear search is preferred over binary search when the array is small or the array is unsorted.

17. The process of arranging the data in some logical order is known as sorting.

18. If all the data that is to be sorted fits entirely in the main memory, then internal (in-memory) sorting is used. On the other hand, if all the data that is to be sorted does not fit entirely in the main memory, external sorting is required.

19. If a particular pass is made through the list without swapping any items, then there will be no further swapping of elements in the subsequent passes.

20. To sort an array of size n in ascending order using heap sort, the following steps are required:

    a. The initial max-heap is built from the given array.

    b. The root element is swapped with the last element in the array.

    c. The heap of remaining elements is restored.

d. Steps ii and iii are repeated until there are no more elements.

21. During restoration, the element at the root node is compared with its child node(s), and if it is smaller than its child nodes, then it is swapped with the greatest of the two child nodes.

22. Merge sort is based on the principle of divide-and-conquer.

23. The partitioning element in quick sort is known as a pivot.

24. The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left.

25. Two most important efficiency parameters are the time required to execute the algorithm and the amount of memory space it requires.

26. The estimated amount of time required in executing an algorithm is referred to as the time complexity (or time efficiency) of the algorithm.

27. In the first pass of merge sort algorithm, the given array is divided into two halves and each half is sorted separately.

5.10 SUMMARY

- The time taken for a search depends on the size of its elements.

- Hashing was first discovered, published and implemented on computers in the early 1950s.

- Hashing is based on a simple idea of converting the key to a number and then using that number to index a table.

- A hash function is a function that takes an element of whatever data type you are storing (integer, string, etc.) and outputs an integer.

- A distribution independent function means that this function does not use the distribution of the keys of a table in computing the position of a record.

- The folding technique involves the splitting of keys into two or more parts.

- Collisions, buckets and slots are some of the basic terms associated with hash tables.

- Perfect hash function is a function which produces a unique set of integers within some suitable range.

- Slot refers to several keys being mapped into one bucket.

- External hashing aims to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only one bucket.

- A file organization in which the records are sorted, based on the value of one of its fields is called sequential file organization, and such a file is called sequential file. In a sequential file, the field on which the records are sorted is called ordered field.

- Unlike sequential file, records in this file organization are not stored sequentially. Instead, each record is mapped to an address on the disk on the basis of its key value. One such technique used for this mapping of

record to an address is called hashing.

- The file organization, using the indexed sequential access method (ISAM) provides the benefits of both the sequential and random file organization methods. This type of organization allows accessing the records both sequentially, based on some key value and also directly, by using the same key.

- The process of finding the occurrence of a particular data item in an array is known as searching. Search is said to be successful or unsuccessful depending on whether the data item is found or not. The various search techniques are linear search, binary search, Fibonacci search and interpolation search.

- The average complexity of linear search is `O(n)`, and the complexity of binary and Fibonacci search algorithm is `O(log2n)`. However, in general, the performance of Fibonacci search is always worse than the binary search. In the best case when the elements are uniformly distributed, the interpolation search requires `log2n(log2n)` comparisons. In the worst case, when elements are not uniformly distributed, the performance of interpolation search deteriorates to linear search.

- The process of arranging the data in some logical order is known as sorting. There are two types of sorting, namely, internal sorting and external sorting.

- If all the data that is to be sorted fits entirely in the main memory, then internal (in-memory) sorting is used. On the other hand, if all the data that is to be sorted do not fit entirely in the main memory, external sorting is required.

- The various internal sorting techniques are insertion, bubble, selection, heap, merge, quick and bucket sort.

- Heap sort is a much more efficient version of selection sort. It uses a different data structure called heap, which can be visualized as a complete binary tree. Heaps can be of two types, namely, max-heap and min-heap.

- Merge sort and quick sort algorithms are based on the fact that it is easier and faster to sort two smaller arrays than one larger array.

- The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left. The number of passes in the algorithm is equal to the number of digits in the largest number. Therefore, the algorithm first determines the largest number in the list and counts the number of digits in it.

- In order to choose an appropriate algorithm that suits a particular problem, the analysis of algorithms is necessary. Two most important efficiency parameters are time required to execute the algorithm and the amount of memory space it requires.

- Using the concept of `Big Oh` notation, you can compare various sorting algorithms and classify them as good or bad, in general terms. Generally, the complexities of most of the sorting algorithms range between `O(n2)` and `O(n log n)`.

5.11 KEY TERMS

- **Hash Table:** It is a simple data structure that offers fast lookup, insertion and deletion of pairs.

- **Hash Function:** It is a function that takes an element of whatever data type you are storing and outputs an integer in a certain range.

- **Radix Conversion:** It transforms a key into another number base to obtain the hash value.

- **Collision:** It is a condition which occurs when two or more keys produce the same hash location.

- **Perfect Hash Function:** It is a function which generates no collision.

- **Searching:** It is the process of finding the occurrence of a particular data item in a list.

- **Binary Search:** It is a technique used to search for a particular element in a sorted array.

- **Sorting:** It is the process of arranging data in a logical order.

- **Bubble Sort:** It is an algorithm that makes the larger values to bubble up to the end of the list.

- **Heap:** It is a data structure which can be visualized as a complete binary tree.

5.12 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Describe a hash table with an example.

2. What are the characteristics of a good hash function?

3. What is linear open addressing? What are its types?

4. What is the process of spiral storage?

5. What do you mean by ISAM?

6. Give a brief comparison of binary and Fibonacci search in terms of their performance.

7. Consider the following array:

   10, 15, 32, 40, 48, 52, 78, 80

   Write the steps to search the element 52 using Fibonacci search. Determine the number of comparisons required.

8. Write a program to sort a linked list of integers using selection sort.

9. Sort the following array in descending order using heap sort.

   1, 3, 24, 17, 5, 32, 6, 99

10. Write a program that accepts an integer array of size N and exchanges the elements of the first half with the elements of the second half of the array. For example, if the array is *{8, 10, 1, 3, 17, 90, 13, 60}*, then the output should be *{17, 90, 13, 60, 8, 10, 1, 3}*.

Long-Answer Questions

1. Explain the major hashing functions.

2. Describe truncation or digit/character extraction in your own words.

3. Write notes on:
   a. External hashing
   b. Linear hashing

4. Discuss the basics of bucket handling.

5. Explain different searching techniques. Give the advantages and disadvantages of these techniques.

6. Write an algorithm to sort an array of 10 elements using bubble sort.

7. What is time complexity of an algorithm? How does the size of input play an important role in determining the time complexity of the algorithm?

8. Determine the time complexity of quick sort and heap sort algorithm.

9. Write a program that accepts a single-dimensional integer array of size N, and reverses the contents of the array without using any second array. For example, if the array is *{2.15, 3, 14, 7, 9, 19, 6, 1, 10}*, then the output should be *{10, 1, 6, 19, 9, 7, 14, 3, 15, 2}*

5.13 FURTHER READING

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures.* New Delhi: Galgotia Publications, 2003.

Tanenbaum, A.M., Yedidyah Langsam and Moshe J. Augenstein. *Data Structures using C and C++*. New Delhi: Prentice-Hall of India, 1995.