WILEY
PRECISE
TEXTBOOK

WILEY-
INDIA

Data Structures
Using C & C++

Rajesh K. Shukla

Data Structu
Using C and C+

Data Structu
Using C and C+

Rajesh K Shukla
Vice Principal and Head
Department of Computer Science and Engineering
Corporate Institute of Science & Technology
Bhopal, Madhya Pradesh

WILEY-
INDIA

Data Structures Using C++ and C

Other Wiley Editorial Offices:

John Wiley & Sons, Inc. 111 River Street, Hoboken, NJ 07030, USA

Wiley-VCH Verlag GmbH, Pappellaee 3, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop # 02-01, Jin Xing Distripark, Singapore 1298 09

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada, M9W 1L1

First Edition: 2009

ISBN: 978-81-265-1997-2

ISBN 978-81-265-0194 (ebk)

www.wileyindia.com

To my mother Mrs. Sushila Shukla
and my son Tanishque Shukla

Preface

Data Structures using C and C++ is designed to serve as a textbook for a single-semester undergraduate course on data structures and algorithms. It provides an insight into the fundamentals of data structures as delineated by the syllabi of various reputed Indian universities. This book introduces the concept of data structures through both the programming languages C and C++ in a very systematic manner under one umbrella with numerous illustrative examples. This book does not assume any basic knowledge of C or C++ on the part of readers; it covers the fundamentals of both the programming languages so it is easy to the student to write programs for different types of data structures.

The book has been written in a simple and comprehensible manner to enable the students to grasp important concepts easily. The student-friendly nature of the book adds to its value and the presence of a large number of examples and illustrations increases its utility. End-of-chapter exercises, containing objective-type problems as well as long-answer type questions, have been included to test the students' understanding of the topics discussed. A number of model question papers have also been included at the end of the book to help students gain practice in problem solving. The book is organized in such a manner that students and programmers will find it easy to read and understand.

Chapter 1 introduces the fundamentals of data structures that will help the students in understanding the concept of data structures. Chapter 2 deals with the simplest linear data structure called an array in a very effective manner so after reading this chapter student can understand the sequential representation of various data structures. Chapter 3 covers fundamentals like structures, class and template of the C/C++ programming languages that are used in the later chapters to program various kinds of data structures. Chapter 4 explores the dynamic memory management and pointers that will be used to dynamically represent the different kinds of data structures. Chapter 5 covers another type of linear data structures – stacks and queues which put restriction on the insertion and deletion of elements. Chapter 6 deals with one of the most difficult part of the data structures called linked list in a very simple and easy-to-understand manner. Chapter 7 focuses on the first non-linear data structures called trees. Chapter 8 explores another non-linear data structure called graph. Chapter 9 introduces the searching and hashing techniques used to search an element in the list and resolve the any collision that occurs. Chapter 10 covers different types of sorting and their complexities. Chapter 11 deals with the advanced topics of data structure – file handling.

The key features of the book are as follows:

1. It provides a lucid introduction to data structures.
2. It provides implementation in both C and C++ programming languages.
3. It includes a large number of solved examples.
4. It contains a number of illustrations to enhance understanding of the concepts.
5. It incorporates extensive end-of-chapter exercises, including objective-type questions.

Acknowledgments

First of all, I would like to thank my "guru", guide, philosopher and mentor Dr P.K. Chande and Dr Sanjay Silakari for continuously motivating and guiding me during the writing and completion of this book in all respects.

I have immense pleasure in expressing my whole hearted gratitude to all my teachers Dr G.P. Basal, Dr J.L. Rana, Dr R.C. Jain, Dr R.K. Pandey, Dr K.R. Pardasani, Dr S. Tokekar, Dr P. Mitra, Dr Sanjeev

Jain, Dr Manish Manoria, Dr Mahesh Motwati, Dr A K Sachan, Dr Bhupendra Verma, Prof. Sanjeev Sharma, Prof Rajesh Pateria and Dr O.B.V. Ramanaiah for their continuous encouragement during the preparation of this book.

I cannot forget my students and my friend and all my colleagues of Corporate Institute of Science and Technology for their invaluable suggestions and critical review of the manuscript and for inspiring me to write this book? Thanks to all of you.

I am grateful to Paras Bansal, Associate Publisher, at Wiley India Pvt. Ltd. who was very accessible and helped to keep my spirit up. My deep sense of appreciation to the editorial group of Wiley India Pvt. Ltd. especially to Meenakshi Sehrawat and Sruthi Guru who have been enthusiastic and helpful in giving fruitful comments and suggestions while reviewing the manuscript.

I would be failing in my duty, if I do not say a word of thanks to my father, mother, wife, daughter, son and brother for all educative, effective, pleasurable and sincere advice.

Last but not the least, I would like to thank the whole management of Corporate Institute of Science and Technology, Bhopal, especially to Er G.C. Shah (Chairman), Er Sunil Kumar Gupta (Secretary), Er Satish Parihar (Chief Operating Officer) and Dr N.K. Gupta (Director) for continuously encouraging me for writing this book.

Rajesh K. Shukla

About the Author

Rajesh K. Shukla is currently the Vice Principal and Head of the Department of Computer Science and Engineering at Corporate Institute of Science and Technology, Bhopal. He has done his BE and MTech (CSE) from Samrat Ashok Technological Institute, Vidisha and is currently pursuing PhD from Rajiv Gandhi Proudhyogiki Vishwavidyalaya, Bhopal in the area of Web Mining and Web Personalisation. He has been teaching data structures and algorithms over the last 10 years at undergraduate and postgraduate levels. With over a decade of teaching experience in the engineering institutions, he has exemplified his work at various designations.

Prof Shukla is a life member of CSI, ISTE, ISCA and a member of ACM IEEE, IAENG and IACSIT, Singapore. His areas of interests include object oriented programming, analysis of algorithms, data structures, database management system, theory of computation, artificial neural network and web mining. He has presented some papers at national and international conferences and authored two other popular titles "Object Oriented Programming in C++" and "Theory of Computation". He has also organized and attended several workshops and summer/winter schools. He is also a reviewer of some of the international journals and member of technical committees of various international conferences.

Contents

1 Introduction to Data Structures

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Introduction to data structures.
- Definition of data structures.
- Different operations on data structures.
- Introduction to algorithms.
- Complexity and their types.
- Big Oh notation.

The computer nowadays is involved in our day-to-day lives. The computer has limited memory so there is need for efficient utilization of memory. We all know that the computer works on a binary system, comprising 0 and 1 (true or false). Computer memory is divided into the smallest unit called bits, and memory has some addresses. Since the computer stores and manipulates large amount of data, the type of the data and storage of the data are very important. We also know that different programming languages support different data types, and that different operations are possible on them, for example, remainder (%) operation is not valid with float values. Different data types take different amount of storage as shown in the following table provided for C and C++.

S. No.	*Data type*	*Size (in bytes)*
1	`int`	2
2	`char`	1
3	`float`	4
4	`double`	8
5	`long int`	4

Now, the question arises: Apart from the single values, is it possible to store and process the collection of the same data types or of different data types? If possible, then how are the data stored in the memory and how are they processed? The answer lies in the study of data structure. Therefore, data structure is the study of the organization of data in the main memory and their processing. In this chapter, we will discuss the various types of data structures and algorithms as well as analysis of the algorithms.

1.1 Definition

A data structure may be defined as the logical or mathematical model of the organization of data elements in the computer memory, where some specific method exists to access and process the data elements. This specific method is called an algorithm. The mathematical model to organize the data in the computer memory (main or secondary) and the methods to process them are collectively called data structures. Data structures are the basic building blocks of the program; therefore, they should efficiently process the elements. These can be classified into two different categories as shown in Figure 1.

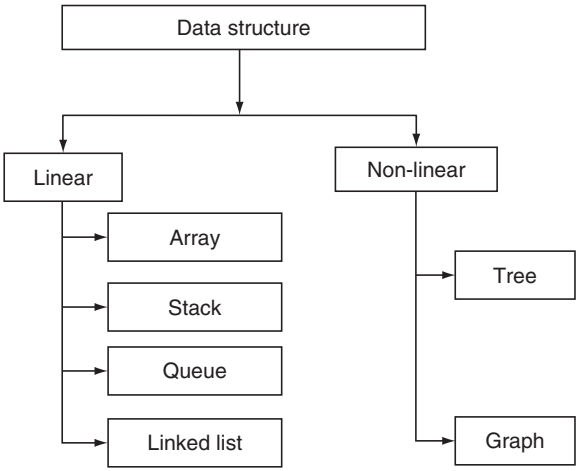


Figure 1 Categories of data structures.

1. **Linear data structure:** Those data structures in which data elements are organized in some sequence are called linear data structures, for example, arrays, stacks, queues and linked lists. In this category, various operations are only possible in a sequence, that is, we cannot insert the element directly into some location without traversing the previous element of the data structures.
2. **Non-linear data structure:** Those data structures in which data elements are organized in any arbitrary order or they are not organized in any sequence are called non-linear data structures. The data stored in the various data structures can be processed using various operations.

Common Operations on Data Structures

The various operations that can be performed on data structures are explained briefly here; and explained in detail, in the subsequent chapters.

1. **Creating:** The first operation used to define the data structure for the data elements is called create operation on the data structures. This operation reserves the memory for the data elements.
2. **Traversal:** The elements of data structure are accessed in order to process them for various operations. The accessing or visiting of each element of the data structure exactly once is called traversal operation on the data structure.
3. **Searching:** The operation used to find the location of a particular element in the data structure is called the search operation on the data structure.
4. **Insertion:** The operation used to add some element into the data structure is called insertion operation on the data structure.
5. **Deletion:** The operation used to delete an existing element from the data structure is called deletion operation on the data structure.
6. **Updating:** The operation used to change the existing value of the data element is called update operation on the data structures.
7. **Sorting:** Some applications require arranging the data elements of the data structure in some specific order (either ascending or descending). This arrangement is called sorting operation on the data structure.
8. **Merging:** The operation used when elements of two different sorted lists, composed of similar elements, are combined to form a new sorted list is called merging operation on the data structure.

1.2 Algorithms

The computer is used to solve real-life problems which are first identified and designed before being programmed for a computer. Algorithms are used to design the solution of a problem. Therefore, we can say that an algorithm is a tool for solving any computational problem. It may be defined as a sequence of instructions which is applied to solve a computational problem. These instructions are applied on some raw data called the input, and the solution of the problem produced is called the output. There are some characteristics that each algorithm should possess.

1. Some input (raw value) must be supplied to the algorithm externally to solve the computational problem.
2. The algorithms process the data by applying some operations on them, and then the desired output is generated.
3. The algorithms must reach their concluding state; that is, they must terminate after executing certain number of instructions.

Algorithms are expressed in an English-like language called pseudocode; and designed by using basic programming constructs, such as sequence, branching and looping. The algorithms are graphically represented by using flowcharts, where different symbols are used to represent different programming constructs as shown in Figure 2.

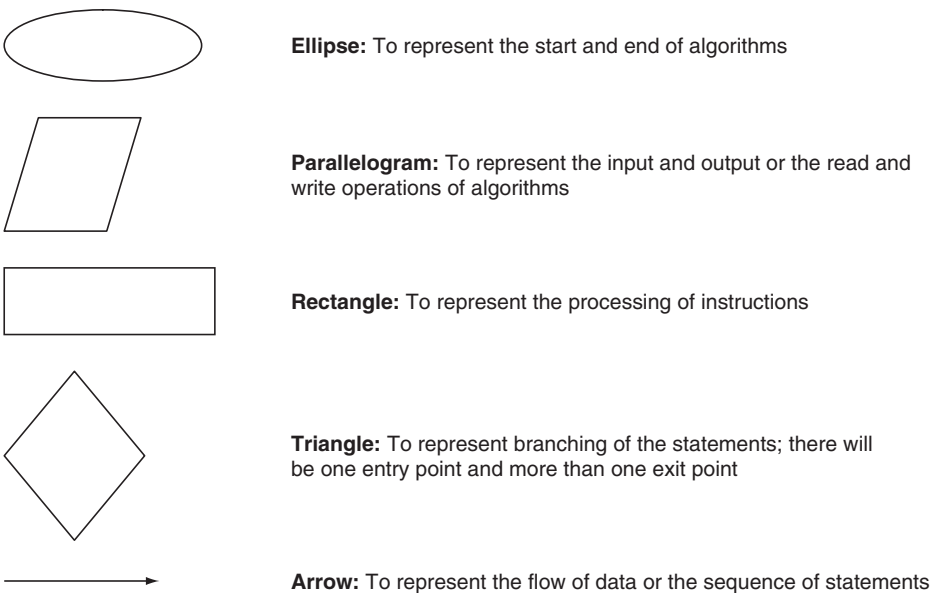


Figure 2 Different symbols used to represent different programming constructs in algorithms.

1.3 Complexities

In order to solve a computational problem, it is necessary to consider the various factors involved in solving the problem. Since computer has a limited memory, the first requirement is the consideration of memory space and its efficient utilization, called the space complexity of the algorithm. The second factor that must be taken into consideration is the time required to solve the computational problem,

called time complexity of the algorithm (program). The third factor depends on the selection of programming languages in which the algorithm is implemented. The computational problem can be solved by using the different algorithms; however, the selection of efficient algorithms is very important so that it reduces the space and time utilization for that problem. The selection of an algorithm depends mainly on the following factors:

1. space required to solve the computational problem;
2. time required to solve the computational problem;
3. programming language to implement the problem.

These factors are discussed in detail below.

1. **Space complexity:** We have been discussing the space requirement for the algorithm or the program, but the question is: What is the space required for? The space is required to store and process the instructions, variables, constants, formal parameters, actual parameters, dynamically allocated memory, etc. used in the program.
2. **Time complexity:** The amount of time required to execute the complete program is called time complexity of the program. Since it is difficult to measure the time complexity directly, it can be expressed as a function of a number of key operations performed.

Since the space and time complexity depend on the instruction executed to solve the computational problem, the space and time for the instruction execution depends on the size of the input. Normally, the time complexity depends on the size of input supplied to the program. If N is the number of the elements to be processed by an algorithm (i.e., N is the size of the input) then the number of operations can be represented as a function of N, that is, $f(N)$. Two algorithms have the same complexity if functions representing the number of operations have the same rate of growth. The type of input also affects the running time, for example

1. **Worst-case analysis:** For instance, if the input is in descending order then sorting it into an ascending order would be the worst case for sorting. The worst-case analysis is based on ideal input. Hence, the worst-case complexity of algorithms is defined by a function which takes the maximum number of steps to process the input data in the desired form.
2. **Best-case analysis:** For instance, if the input is in ascending order then sorting it into an ascending order is the best case for sorting. The best-case analysis is based on best possible input. Hence, the best-case complexity of algorithms is defined by a function which takes the minimum number of steps to process the input data in the desired form.
3. **Average-case analysis:** For instance, if the input data is scattered evenly then sorting it into an ascending or a descending order is an average case for sorting. The average-case analysis is based on the average outcome of execution of the instruction. Hence, the average-case complexity of algorithms is defined by a function which takes the average number of steps to process the input data in the desired form.

The time efficiency of almost all the algorithms can be characterized by a few growth-rate functions. If n is the number of the elements to be processed by an algorithm (i.e., n is the size of the input) then the number of operations can be represented as a function of n, that is, $f(n)$. Given a function $f(n)$, there are some notations which are used to represent the time complexity of the program in terms of $f(n)$. Some of these notations are discussed as follows.

Big Oh (O) Notation

Suppose the time required by some algorithms to solve the computational problem is described by $f(n) = 3n^2 + 5n - 6$. When we analyzed this function $f(n)$, we observed that as n grows, n^2 term will begin

to dominate and $5n - 6$ part becomes insignificant for the large values of n. Clearly, the complexity function $f(n)$ of an algorithm increases as n increases. It is the rate of increase of $f(n)$ that needs to be examined, because the constant terms will depend on the implementation details and on the hardware the algorithm runs on. So the algorithm's complexity can be determined ignoring implementation-depending constant factors. The rate of growth of a function is called asymptotic growth. Since we are interested in the asymptotic behavior of the function growth, the constant factor can be ignored when analyzing algorithms and this is allowed by the Big Oh (O) notation. This notation is used to describe the asymptotic upper bound for the magnitude of a function in terms of another, usually simpler, function; and it tells us that a certain function will never exceed another (simpler) function beyond a constant multiple for large enough values of n.

The Big Oh is represented by an uppercase O and never by a digit zero. We write $f(n) = 3n^2 + 5n - 6 = O(n^2)$ and say that the algorithm has an order of n^2 time complexity. Hence $O(n^2)$ is a set, or a family of functions, such as $4n^2 + 3n$, $7n^2 - n - 3$, $n^2/5 + 3n - 5$ and so on. Let us assume that f(n) and g(n) are two functions defined on some subset of the real numbers.

$$f(n) = O(g(n)) \quad \text{as } n \to \infty \quad \text{iff } \lim_{n\to\infty}\left|\frac{f(n)}{g(n)}\right| \leq C$$

where C is a constant. This definition implies that the function f does not grow faster than g.
$O(g(n))$ = {The set of all functions f that do not grow faster than $g(n)$, that is, $0 <= f(n) <= Cg(n)$}

The complexity of some algorithms is said to be $O(n)$ if it requires at most $C \times n$ instructions for some constant C (if n is large enough). For example, $25n - 10$ is $O(n)$ where $C = 15$ and $n >= 1$.

Common Orders of Functions

The algorithms may also be classified according to the classes of functions that are commonly encountered while analyzing algorithms. Based on the Big Oh notations, the algorithms can be categorized as follows.

S. No.	*Big Oh notation*	*Name of algorithm*	*Remarks*
1	O(1)	Constant time	Operations are fixed regardless of the size of the input.
2	O(log n)	Logarithmic time	Number of operations are proportional to the logarithm of the size of input.
3	O(n)	Linear time	Number of operations are proportional to the size of input.
4	O(n^k) for $n > 1$	Polynomial time	Number of operations are proportional to the power k of the size of input.
5	O(k^n) for $k > 1$	Exponential time	Number of operations are proportional to the exponent of the size of input.
6	O(n!)	Factorial	Number of operations are proportional to the factorial of the size of input.
7	O(n log n)	Linear logarithmetic time	Number of operations are proportional to the product of logarithms and size of the input.

(*Continued*)

S. No.	*Big Oh notation*	*Name of algorithm*	*Remarks*
8	$O(n^2)$	Quadratic	Number of operations are proportional to the square of the size of input.
9	$O(n^3)$	Cubic	Number of operations are proportional to the cube of the size of input.
10	$O(\log^2 n)$	Logarithmic squared time	Number of operations are proportional to the square of the logarithm of the size of input.

Limitations of Big Oh Notations

1. It is based on the size of the input and it does not consider the implementation language.
2. It is difficult to analyze every algorithm mathematically.
3. It tells about the growth of the function with respect to the simpler function, but it does not tell anything about the efficiency of the program.
4. Finding the order of many useful functions can be a challenging task.
5. Finding the Big Oh notation of many functions is simply a matter of finding a known result for a similar function and using it.

1.4 Program Design

Programming methodology deals with the analysis, design and implementation of programs. There are two methods of designing a program (algorithm) in software development process:

1. top-down and
2. bottom-up.

Top-Down Design

Top-down design is a method of program design that leads from generalization to specialization. Top-down design essentially starts with the description of the overall system and breaks down the system into subsystems. Each subsystem is further broken down into more subsystems until we reach a refined subsystem, which cannot be broken down further. Top-down design consists of a hierarchical structure which contains the description of the subsystem at each lower level. The hierarchical or tree-like structure is shown in Figure 3.

The top level of the hierarchical structure contains the description of the overall system, and each sublevel consists of a kind of system module. As shown in the schematic diagram, the top-down design starts with a specification and definition of the main procedure (i.e., the complex piece of the main program), which is then recursively divided into successively smaller pieces of subprograms that can be coded. The main program consists of all the major functions and the subprogram consists of the requirement of all these functions, and this process is repeated. The top-down design supports the modular approach of the program design by separating the lower level program from the higher level. This makes the software design easy and less time consuming because each module may be prepared by different persons and the errors can be easily identified. The top-down design is preferred by the software designer because of the above-mentioned reasons; and it is also not possible to design lower level in advance as required by the bottom-up approach. However, the use of top-down design requires that

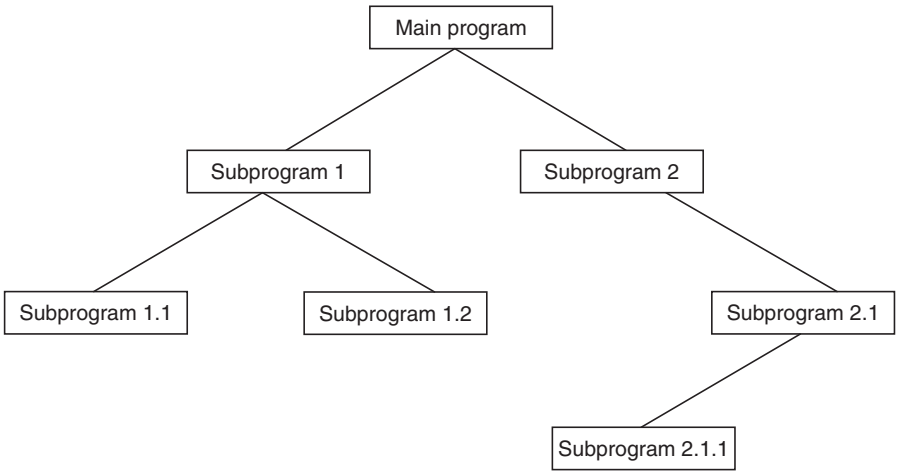


Figure 3 Schematic diagram of top-down approach.

the designer or the system analyst should have the detailed understanding of the system. Top-down design is usually possible only when one has a clear idea of the problem and the method to solve it.

Bottom-Up Design

The bottom-up approach is a reversal of the top-down approach. In this approach, the design methodology starts with the description of the lowest level procedures. These procedures are used to define the more complex procedure at the next higher level and the process repeatedly continues until the complete system is designed. Hence, this approach starts with the specific details and moves up to the general. This design methodology continues until all levels of the system are connected in a hierarchical structure to form the entire system. This is similar to the case when the doctors examine specific symptoms and then infer the general disease that causes the symptoms. The object-oriented programming also goes for the real-world entities called objects to design the application.

These two approaches (top-down and bottom-up design) play a key role in the software development process. Although the top-down and bottom-up design are two different methodologies of the program design, but the real-world software design approaches usually implement a combination of the two. This is because both of them define the system by describing all the interactions possible between the procedures.

1.5 Abstract Data Types

An abstract data type (ADT) is a mathematical tool used to define the concept of data types. We all know that a data type is a collection of values and the operations possible on those values. For example, a float data type can hold values having the fraction parts and the possible operations for this data type are addition, multiplication, etc. This concept of data type is extended to the real-world object and user-defined data types, such as array, structures, class, etc. because large programs deal with collections of data types, which may be organized in many ways and various program structures can be used to represent them. However, there will be a few common operations on any collection. For example, any real-world entity, say, car may have some properties, such as color, gears, speed, company, etc., represented in

different data types and there are certain operations that can be performed on these data types so that their values are changed like driving will change the speed value. Similarly, other operations, such as sale, buy, park can also be associated with the car. However, remember that these properties may belong to other real-world entities also, such as bus, truck, motorcycle, etc., and the same set of operations can be performed on these entities also.

Now the question arises: Is there any method to logically define the common properties and their associated functions for all the common entities in a single unit? Here lies the importance of ADT. The ADT can be defined as a collection of data values and the functions which operate on these data values. It is implementation-independent and isolates the programs from the representation.

Advantages of ADT

The advantages of ADT are as follows:

1. It is language independent.
2. It prevents the direct manipulation of data values, except through the defined functions.
3. It is very important for organizing big software projects because it partitions the solution of a problem into independent parts or modules on which different persons can work.
4. The different modules can be used by different persons for different purposes with small changes.
5. ADTs encourage modularity of the programming.
6. By using ADTs, we can change implementations quickly.

The ADT is only concerned with the data or the properties which are affected and the operations which are identified by the problem. For example, students of an Engineering institution can be characterized with many properties, such as

1. roll number;
2. student name;
3. date of birth;
4. marks in different subjects;
5. sex;
6. city;
7. place of birth;
8. father's occupation;
9. age;
10. previous qualification;
11. height and so on.

All the above information may not be needed to carry out some projects on the students, but some are required to solve the problem. There may be different possible operations which can be performed on these properties of the students. For example, we can display the information, edit the information, print the result sheet of the students and so on. So the ADT "Student" is defined as follows.

```
ADT Student
{
Data:
    Roll_Number;
    Student_name;
    Marks_in_different subjects;
```

```
    Sex;
    Age;
    Height;
Operations:
    Read_information();
    Display_information();
    Print_mark_sheet();
}
```

The object-oriented programming paradigm implements the ADT through classes. Hence the class definition in C++ is very similar to the definition of ADT because all objects of the class will have the same properties and the common operations on the properties.

```
class Student
{
private:
    int Roll_Number;
    char Student_name[10];
    int Marks_in_different subjects[5];
    char Sex;
    int Age;
    float Height;
public:
    void Read_information();
    void Display_information();
    void Print_mark_sheet();
}
```

One of the simplest ADTs is the stack. The ADT "Stack" is defined as follows.

```
ADT Stack
{
Data:
    Item;
    Top;
    Size;
Operations:
    Isemptystack();
    Isfullstack();
    Createstack();
    Pushitem();
    FindTop();
    Popitem();
}
```

The set of operations is also called interface. Some pre-conditions are also defined in the ADT. For example, in stack we cannot push an element if the stack is already full, similarly we cannot delete

element from the stack if stack is already empty. These are the pre-conditions for the push and pop operations of the stack. So the ADT data is defined with the following items:

1. Data items/properties and
2. operations/interfaces.

For each operation, the pre-conditions and post-conditions are given where pre-conditions are required to perform the operations and post-conditions are the result of the operation. For example, for `Push item operation()`, the pre-condition is that stack should not be full means `! Isfullstack()` and the post-condition is that item is inserted and top is changed.

1.6 Generic Abstract Data Types

The same ADTs may also be used for different data types, and such ADTs are called generic data types. These generic data types are supported by C++ with the help of template. In the generic ADT, the data type of the data elements are changed according to the data type supplied externally, which are also called generic parameters. Since the generic ADT is defined for general data types and the actual ADT is created according to the generic parameter supplied externally; therefore, we can say that ADT is an instance of a generic ADT.

```
ADT<Data type> T
{
Data:
    T       STK[MAX];
    T       ITEMI, ITEMD;
    int     TOP;
Operations:
    Initialize()
    {
      To initialize the stack top
    }
    push()
    {
      Insert the data item
    }
    void pop()
    {
      Delete the top element
    }
    Isempty()
    {
      To check whether the stack is empty
    }
    Isfull()
    {
      To check whether the stack is full
    }
```

```
    display()
    {
      To display the contents of the stack
    }
}
```

Summary

1. A data type is a collection of values and the set of operations possible on those values.
2. Data structure is a mathematical model to organize the data and the allowed operations on those data.
3. The data structure should be simple and capable to represent the relationship so that it can be efficiently processed when required.
4. Data structures can be classified into two categories: Linear data structures and non-linear data structures.
5. In linear data structures, such as stacks, queues and linked lists, the elements are organized into a sequence.
6. In non-linear data structures like tree and graphs, the elements do not form a sequence.
7. An algorithm is a well-defined sequence of instructions to solve a particular problem.
8. The complexity depends on the space, time and the programming language implemented.
9. There are two types of complexities: time and space complexity.
10. The Big Oh (O) notation is the most commonly used asymptotic notation for comparing functions.
11. The worst-case complexity of algorithms is defined by a function which takes the maximum number of steps to process the input data in the desired form.
12. The best-case analysis is based on the best possible input and is defined by a function which takes the minimum number of steps to process the input data in the desired form.
13. The average-case analysis is based on the average outcome of execution of the instruction and is defined by a function which takes the average number of steps to process the input data in the desired form.
14. An abstract data type (ADT) is a special case of generic ADT.

Key Terms

Data
Structure
Algorithms
Complexity
Time complexity
Space complexity
ADT
Top-down design
Bottom-up design

Multiple-Choice Questions

1. In the top-down design,
   a. the complexity of a programming task is decreased.
   b. each subtask is coded by separate functions.
   c. black-box principle is important.
   d. all the above.

2. Hierarchical tree-like structure is used in
   a. top-down design.
   b. bottom-up design.
   c. modular design.
   d. all the above.

3. Which of the following is more efficient?
   a. $O(2^n)$
   b. $O(\log n)$
   c. $O(n^2)$
   d. $O(n \log n)$

4. Which of the following is a linear data structure?
   a. Array
   b. Stack
   c. Linked list
   d. All of the above

5. Which of the following is a non-linear data structure?
   a. Tree
   b. Queue
   c. Doubly linked list
   d. Circular linked list

6. A good algorithm must possess which of the following property?
   a. There must be some values which are externally supplied.
   b. Each step must be clear and unambiguous.
   c. The algorithm must terminate after a finite number of steps.
   d. All of the above.

7. The complexity of an algorithm depends on
   a. the space.
   b. the time.
   c. the programming language implemented.
   d. all of the above.

8. Which of the following cases defined the complexities for best possible input?
   a. The worst-case complexity
   b. The best-case complexity
   c. The average-case complexity
   d. None of the above

9. Which of the following is not true?
   a. $O(f) + O(f) \in O(f)$
   b. $O(f)\, O(g) \in O(fg)$
   c. $O(O(f)) = O(f)$
   d. $O(f) \in O(f)$

10. Which of the following is a limitation of the Big Oh notations?
    a. It is based on the size of the input and it does not consider the implementation language.
    b. It is difficult to analyze every algorithm mathematically.
    c. It tells about the growth of the function with respect to the simpler function, but it does not tell anything about the efficiency of the program.
    d. All of the above.

11. The Big Oh is the
    a. upper bound of a function.
    b. lower bound of a function.
    c. average bound of a function.
    d. none of the above.

12. Which of the following statements is true?
    a. The average-case complexity of algorithms is defined by a function which takes the maximum number of steps to process the input data in the desired form.
    b. The worst-case complexity of algorithms is defined by a function which takes the maximum number of steps to process the input data in the desired form.

c. The best-case complexity of algorithms is defined by a function which takes the maximum number of steps to process the input data in the desired form.
d. None of the above.

13. To represent the linear logarithmetic time algorithms, which of the following notations is used
a. O(1)
b. O(n)
c. O(log n)
d. O(n log n)

14. To represent the logarithmic time algorithms, which of the following notations is used?
a. O(log n^2)
b. O(n)
c. O(log n)
d. O(n log n)

15. To represent the linear time algorithms, which of the following notations is used?
a. O(k^n)
b. O(n^k)
c. O(n)
d. O(1)

16. To represent the constant time algorithms, which of the following is used?
a. O(k^n)
b. O(n^k)
c. O(n)
d. O(1)

17. Which of the following is an advantage of ADT?
a. It is language independent.
b. It prevents the direct manipulation of data values, except through the defined functions.
c. It is very important for organizing big software projects because it partitions the solution of a problem into independent parts or modules on which different persons can work.
d. All of the above.

18. Which of the following is a disadvantage of ADT?
a. The different modules can be used by different persons for different purposes with small changes.
b. ADTs encourage modularity of the programming.
c. By using abstract data types, we can change implementations quickly.
d. None of the above.

Review Questions

1. What are the properties of abstract data types?
2. Why is it necessary to include pre-conditions to the definition of an abstract data type?
3. What is a generic abstract data type, and how it is different from the abstract data type?
4. What do you understand by the sentence "the running time of an algorithm is O(N log N)"?
5. Compare n log n and n^2.
6. Arrange the following functions in the order of increasing rate of growth:

   n log n, n^2, $3n$, $n!$, n
7. What do you understand by data structure and how is it different from algorithms?
8. What is the Big Oh notation? Define data type.
9. What is abstract data type (ADT)? Explain the two parts of ADT.
10. What is an algorithm? What the different types of program designing techniques?

11. How are the generic ADT and ADTs implemented in C++?
12. What are the different types of data structures?
13. What are the different operations possible with data structures?
14. How is the data structure different from the data types?
15. Which algorithm should we implement if we have more than one algorithm?
16. What is the importance of Big Oh notations?
17. Define algorithm and its importance.
18. What is the difference between a flowchart and an algorithm?
19. What are the different approaches to the program design?
20. What is the difference between the top-down design and bottom-up design?
21. Which of the design methods lead to specialization from generalization?

Answers

Multiple-Choice Questions

1. (d)
2. (d)
3. (b)
4. (d)
5. (a)
6. (d)
7. (d)
8. (b)
9. (c)
10. (d)
11. (a)
12. (b)
13. (d)
14. (c)
15. (c)
16. (d)
17. (d)
18. (d)

2 Arrays

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Definition of array.
- Types of arrays.
- Initialization and operation on one-dimensional array.
- Initialization and operation on two-dimensional array.
- Representation of arrays.
- Calculation of address of the element in arrays.
- Strings.
- Sparse array.

Simple variables are used to store single value of a particular data type. For example

```
int X;
```

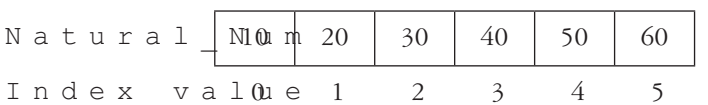
The variable X can hold value of one data item of integer type at a time, for example, we can store the age of a person into X. Now, suppose we want to store many data items of the same type (such as age of 10 people) which can be referred by the single name. In this case, array comes into the picture. We can consider an array as a single folder with many compartments. A folder may have some name and it contains similar types of items. For example, a folder named "Personnel Record" may contain personnel files (say resume, supporting documents, leave application and so on) of many employees of an organization. Therefore, folder is an array named "Personnel Record" that contains similar items (personnel files). Similarly, an array of integer numbers, float numbers, etc. may be defined.

2.1 Defining an Array

An array is a collection of similar data items which may be of int type, char type, float type or any user-defined type such as structure or class. The array elements are stored in consecutive memory locations; therefore, they are linear data structures. In other words, an array is a collection of similar items which are referred by the common name, that is, the array name.

Example 1

Consider the structure in the figure below.



`Natural` is a collection of six similar data items that are of integer type. Here, the collection of natural numbers is referred by a single name that is `Natural`. The elements are stored in different but contiguous locations. The array elements are accessed by their index value which is also called subscript value. So the 0th element (i.e., 10) of an array "`Natural`" is referred to as `Natural[0]`.

There are two types of arrays:

1. **One-dimensional array:** It is a collection of similar data items. It is sometimes called a list.
2. **Multidimensional array:** It is also a collection of similar data items where each data item is also an array of fixed size. Two-dimensional array is a simple case of multidimensional array where each element of the array is a one-dimensional array itself. So we can say that two-dimensional array is a collection of one-dimensional arrays.

2.2 Types of Arrays

The simplest form of an array is a one-dimensional array.

One-Dimensional Array

The one-dimensional array is defined by specifying its name, size and data types of the item. The general format in C or C++ for defining one-dimensional array is

Syntax

```
Storage_class  Data_type  array_name[
Example
int  arr[5];
```

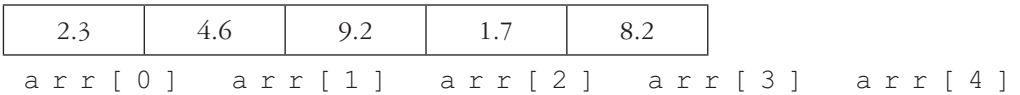
Here, the storage class specifies the scope of the array variables. The storage class can be external, static or auto. The storage class specifier is optional. If it is not used then compiler automatically takes it as auto. The array name is the name of the array which follows the same naming conventions as for a variable. The size must be an integer value or integer constant (without any sign) that specifies the number of elements which are to be stored into the array. The size can be directly given as integer number or an expression may be given that evaluates into an integer. We cannot use the variable name in place of size. It must be constant. The array must be declared before using it in the program like other variables.

```
float  arr[5];
```

Here `arr` is an array which can hold five data items of `float` type. These items are stored into consecutive memory locations. Array elements are accessed by their index or subscript. The first index starts from 0. So the five elements are stored in the following manner.

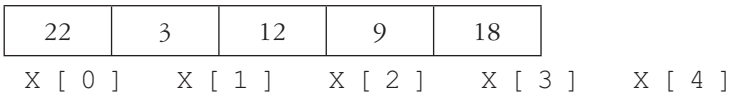
2.3	`arr[0]`
4.6	`arr[1]`
9.2	`arr[2]`
1.7	`arr[3]`
8.2	`arr[4]`

or



In a similar manner we can declare an integer array

```
int X[5];
```



Here X is an array of five integer elements stored in contiguous memory locations starting from the array index 0 up to array index 4. We can find the number of elements in one-dimensional array by using the formula

$$\text{Size} = \text{UB} - \text{LB} + 1$$

where UB is greatest index value and LB is least index value. Therefore

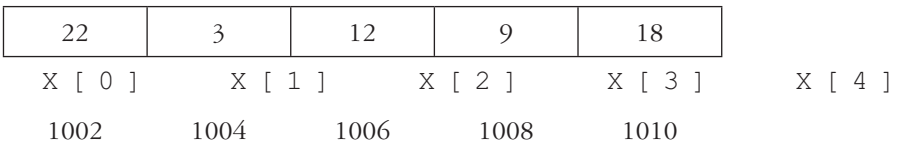
$$\text{Size} = 4 - 0 + 1 = 5$$

We are now aware that the array elements are stored in contiguous location in memory. What does it actually mean? Example 2 explains this in detail.

Example 2

Let us take the example of `int X.[5]`

```
int X[5];
```



Let us assume that the array is stored at the memory address 1002. This means that the first element of the array (i.e., X[0]) is stored at the address 1002. Since the array is declared to be int type, every element of the array takes two bytes of memory. Therefore the next element (i.e., X[1]) is stored in memory location 1004. In the same way X[2] at 1006, X[3] at 1008 and X[4] at 1010. The address of the first element is also known as the base address and array name itself represents the base address. The address of the element in a one-dimensional array can be found by using the formula

```
Position (Arr[K])
Address of Arr[K]=Base(Arr)+(Size)*
```

where Arr is an array name, "Bas" refers to the base address of the array Arr and "Siz" refers to the size of the data type or we can say that it refers to the number of bytes required for one element of the array. We can verify the formula by finding the address of X[3]

```
Address of X[3] = Base(X) + 2 * 3 = 1002 + 6 =
```

Here X is an array of integer type and every integer number takes two bytes of storage so we have taken Siz=2. *We should note that integer element takes two bytes, float takes four bytes and character element takes one byte of memory storage*. We can also find the total number of bytes required for an array by using the formula

Bytes required for an array = Size of an array*bytes for an element of the array

Problem 1

Consider an array X[8] of characters and let its base address be 1002. What is the address of X[5] and the total number of bytes required for this array?

Solution
We know that

```
Address of Arr[K] = Base address + Siz
```

Substituting the values we have

```
Address of X[5] = 1002 + 1 * 5 = 1007
```

Bytes required for an array = Size of an array × bytes for an element of the array

$= 8 \times 1 = 8$

Problem 2

Consider an array X{10} of float values and let its base address be 1002. What is the address of X{5} and total number of bytes required for this array?

Solution
We know that

```
Address of Arr[K] = Base address + Siz
```

Substituting the values we have

```
Address of X[5] = 1002 + 4 * 5 = 1022
```

Bytes required for an array = Size of an array × bytes for an element of the array

$= 10 \times 4 = 40$

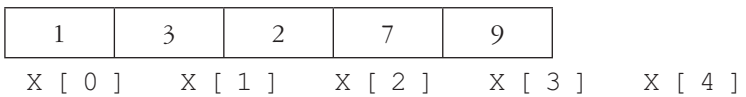
Initialization of One-Dimensional Array

Array elements can also be initialized in the same way as simple variables. The general format for initialization of one-dimensional array elements is

Syntax

```
storage_class  data_type  array_name[size]={val
```

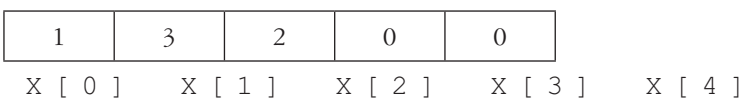
The initial values may be enclosed into the braces and they are separated by comma. For example, `int X[5]={1, 3, 2, 7, 9};`
The elements of array X are initialized as follows:



We know that the index of an array always starts with 0, so the first element is represented by `X[0]`, second by `X[1]` and so on. It is not necessary that all the elements of the array should be initialized.

```
int  X[5]={1, 3, 2};
```

The elements of array X are initialized as follows:



Though only three elements of array X are initialized, the memory is allocated to five elements, therefore the remaining two elements are initialized to 0.

Program 1 A program to initialize the array elements and display them.

```
#include<iostream.h>
#include<conio.h>
#define  max  5;
void  main()
{
clrscr();
int  X[5]={2,5,8,1,6};
float  Y[5]={1.2,3.1,2.0,4.9,3.1};
int  A[5]={12,25,38};
float  B[5]={11.2,23.1,2.05};
int  i;
cout<<"**ARRAY  X***";
for(i=0;i<5;i++)
```

```
{
cout<<"\nDisplay value of X["<<i<<"]:"<<X
}
cout<<"\n**ARRAY Y***";
for(i=0;i<5;i++)
{
cout<<"\nDisplay value of Y["<<i<<"]:"<<Y
}
cout<<"\n**ARRAY A***";
for(i=0;i<5;i++)
{
cout<<"\nDisplay value of A["<<i<<"]:"<<A
}
cout<<"\n**ARRAY B***";
for(i=0;i<5;i++)
{
cout<<"\nDisplay value of B["<<i<<"]:"<<B
}
getch();
}
```

Output

```
**ARRAY X***
Display value of X[0]: 2
Display value of X[1]: 5
Display value of X[2]: 8
Display value of X[3]: 1
Display value of X[4]: 6
**ARRAY Y***
Display value of Y[0]: 1.2
Display value of Y[1]: 3.1
Display value of Y[2]: 2
Display value of Y[3]: 4.9
Display value of Y[4]: 3.1
**ARRAY A***
Display value of A[0]: 12
Display value of A[1]: 25
Display value of A[2]: 38
Display value of A[3]: 0
Display value of A[4]: 0
**ARRAY B***
Display value of B[0]: 11.2
Display value of B[1]: 23.1
Display value of B[2]: 2.05
Display value of B[3]: 0
Display value of B[4]: 0
```

Operations on One-Dimensional Array

There are different operations that can be performed on a one-dimensional array. These are as follows:

1. insertion;
2. deletion;
3. traversal.

Insertion and deletion can be performed from any position: at the beginning, at the end or in the middle. If insertion is required at the beginning or in the middle then the array elements are shifted so that the element can be inserted in a given position. An element can be inserted at the end of the array if sufficient memory space is allocated to accommodate the element. Similarly, when the element is deleted from the middle or beginning of the array, the elements of the array are shifted. When element is deleted from the last, then element shifting is not required. The different situation for insertion and deletion is shown in Figure 1.

Shifting of array elements can be done by using

```
A [ i + 1 ] = A [ i ]
```

where `i` starts from the last location of the array which is gradually decreased up to the position where element is to be inserted. The traversal of an array starts from lower bound, accessing each element and proceeds to the upper bound. The algorithm for insertion at a particular position is given here.

We can see that these algorithms are very simple to write. Algorithms for other operations are left as an exercise for the students. They can refer to the programs given at the end of this chapter for the same.

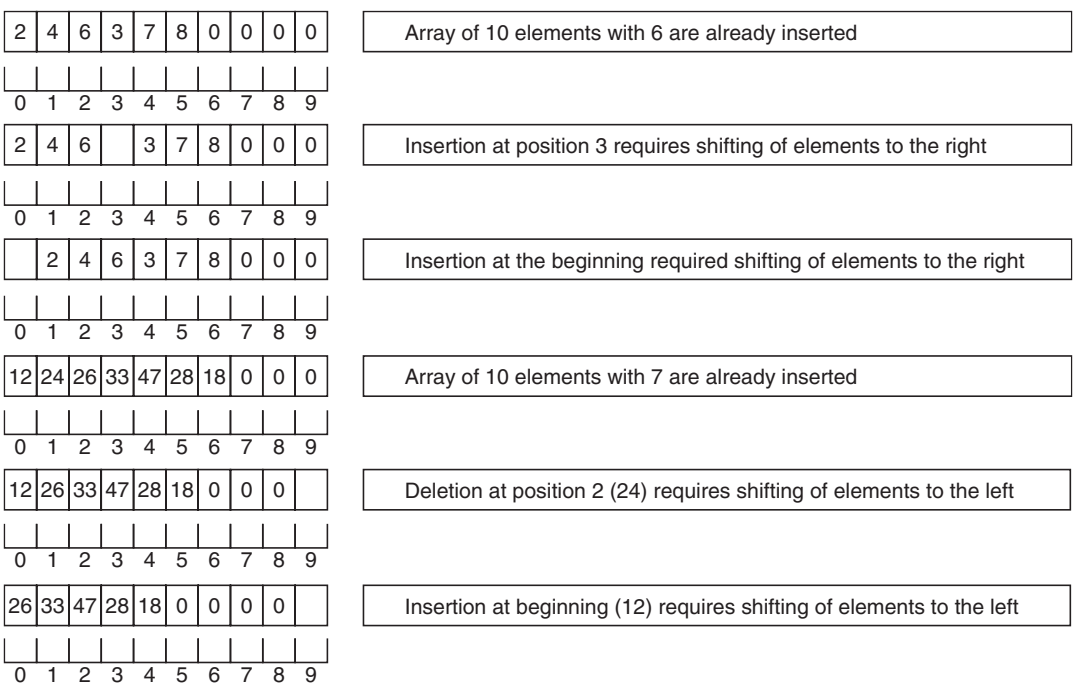


Figure 1 Representation of insertion and deletion in an array.

Algorithm 1 Insertatposition(Arr, Max, Pos, E

```
1. Set=Max - 1
2. Repeat
      Arr=A[i - 1];
      Set=i - 1;
3. Until>=Pos
4. Set Arr[i] Element
5. Return
```

Algorithm 2 Deleteatposition(Arr, Max, Pos, E

```
1. Set=pos
2. Repeat
      Arr[=A[i]];
      Set=i+1;
3. Until<Max
4. Set Arr=[i - 1]
5. Return
```

Multidimensional Array

Multidimensional arrays are defined in a similar way as one-dimensional arrays are defined, although the number of square brackets are more (a pair for each subscript is required) in multidimensional array. There will be two pairs of square brackets for two-dimensional arrays, three pairs for three-dimensional arrays and so on. The general syntax for declaration of multidimensional array is

Syntax

Storage_class data_type array_name [Dim$_1$][Dim$_2$]...[Dim$_n$]

where Dim$_1$, Dim$_2$, ... Dim$_n$ are the dimensions of the array.

Two-Dimensional Arrays

It is the simplest form of the multidimensional array where there are two dimensions only. The syntax to define two-dimensional arrays in C or C++ is

Syntax

Storage class data type array name [

The syntax is same as for multidimensional arrays. Here only two dimensions are required so first dimension refers to the number of rows and the second dimension refers to the number of columns. It is just like a matrix in mathematics. For example

```
char student[10][5];
```

Here `stud` is a two-dimensional array of `char` type. `stud` array contains names of 10 students where each name can have maximum 5 characters. We can say that it is a one-dimensional array (of names); that is, each element itself is a one-dimensional array (of maximum 5 characters). So there will be total 50 characters in the array `stud`. If a two-dimensional array `X[m][n]` is there then this array can contain `m*n` elements. For example

`int X[3][3];` contains $3 \times 3 = 9$ elements as shown below.

1	2	3
4	9	5
6	8	7

Processing a Two-Dimensional Array

The multidimensional array is usually processed in row major order discussed later in this section. First, all the columns of row 1 are processed, then the columns of row 2 and so on. In C/C++ usually two loops are required to process the elements of two-dimensional arrays.

Program 2 A program to read and display two-dimensional array elements.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int X[3][4],i,j;
cout<<"Enter array elements\n";
for(i=0;i<3;i++)
{
   for(j=0;j<4;j++)
   {
   cout<<"Enter X["<<i<<"]"<<"["<<j<<"]:";
   cin>>X[i][j];
   }
}
cout<<"The array elements are:\n";
for(i=0;i<3;i++)
{
   for(j=0;j<4;j++)
   {
     cout<<X[i][j]<<"    ";
   }
   cout<<endl;
}
getch();
}
```

Output

```
Enter array elements
Enter X[0][0]: 1
Enter X[0][1]: 3
Enter X[0][2]: 4
Enter X[0][3]: 6
Enter X[1][0]: 9
Enter X[1][1]: 5
Enter X[1][2]: 2
Enter X[1][3]: 7
Enter X[2][0]: 8
Enter X[2][1]: 3
Enter X[2][2]: 2
Enter X[2][3]: 7
The array elements are
1     3     4     6
9     5     2     7
8     3     2     7
```

Initialization of Two-Dimensional Array Elements

We can initialize the two-dimensional array in the same manner as one-dimensional array is initialized. For example

```
int X[3][4]={1, 3, 5, 7, 9, 11, 32, 45, 4
```

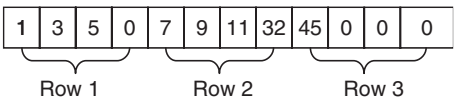
Array X contains 12 (3×4) elements and so all the 12 elements are assigned the initial values. It is not necessary to initialize all the elements. For example

```
int X[3][4]={1, 3, 5, 7, 9, 11, 32, 45};
```

Here 8 elements are assigned the initial values. Remaining elements are initialized to zero by default. The assignment is always done in row major order, that is, initially first row is initialized then second row is initialized, then third and so on. If we want to change the order of initialization, we can use extra braces. For example

```
int X[3][4]={{1,3,5}, {7,9,11,32}, {45}};
```

In this example, the first three elements of row one are initialized; the last element is initialized to zero by default. All the four elements of the second row are initialized. Only one element of the third row is initialized and remaining three elements are initialized to zero by default.



Program 3 A program to initialize the two-dimensional array elements and display them.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int X[3][2]={1,2,3,4,5,6};
int i,j;
int Y[3][2]={1,2,3,6};
int Z[3][2]={{1,2},{3},{6}};
cout<<"Display elements of X \n";
for(i=0;i<3;i++)
{
   for(j=0;j<2;j++)
   {
   cout<<X[i][j]<<"   ";
   }
cout<<endl;
}
cout<<"\nDisplay elements of Y \n";
for(i=0;i<3;i++)
{
   for(j=0;j<2;j++)
   {
   cout<<Y[i][j]<<"   ";
   }
cout<<endl;
}
cout<<"\nDisplay elements of Z\n";
for(i=0;i<3;i++)
{
   for(j=0;j<2;j++)
   {
   cout<<Z[i][j]<<"   ";
   }
cout<<endl;
}
getch();
}
```

Output

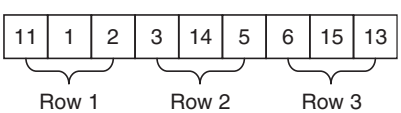
```
Display elements of X
1    2
3    4
5    6
Display elements of Y
1    2
3    6
0    0
Display elements of Z
1    2
3    0
6    0
```

Implementation of Two-Dimensional Arrays

Since all elements of two-dimensional array are stored sequentially, so there are two ways of representing a two-dimensional array:

1. row major order;
2. column major order.

Row Major Order In this case, the elements are arranged row-wise, that is, first all the elements of first row are arranged, then all elements of second row and so on. This is the default representation of two-dimensional arrays. For example, `int X[3][3]={11, 1, 2, 3, 14, 5, 6, 15, 13};` is represented in row major order as shown below.



This is the physical representation of a two-dimensional array. This can be clearly understood by arranging `X[3][3]` in a logical form of two-dimensional array:

$$\begin{pmatrix} 11 & 1 & 2 \\ 3 & 14 & 5 \\ 6 & 15 & 13 \end{pmatrix} \begin{matrix} \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \end{matrix}$$

Here all elements of Row 1 are arranged, then all elements of Row 2 and so on. Since all elements are stored in contiguous memory locations we can find the address of the element at the given position if we know the base address and the dimension of the array.

Address of Elements in Row Major Order The address of the element in two-dimensional array can be calculated by using the following formula:

```
X[i][j]=Base address+Size*(Number of co
```

If rows and columns are not given in the range but their dimensions are given in a range then we cannot find the address using the above formula. The general formula for calculating the address of the element in a two-dimensional array if the elements are arranged in row major order is

$$X[i][j] = \text{Base address} + \text{Size} * [\text{Number of rows before } (i-L_1) * \text{Number of columns} + \text{Number of columns before } (j-L_2)] \quad (2.2)$$

where "Base address" is the address of the first element (X[0][0]) of the array, "Size" is the number of bytes required for an element of the array, L_1 is the lower bound of the rows and L_2 is the lower bound of the columns. The number of columns can be calculated by finding the difference between the upper bound and lower bound of the second dimension $[U_2 - L_2]$.

Problem 3

Consider an array X{4}{3} of integer elements with base address 1002 arranged in row major order. Find the address of X{1}{3}.

Solution

Since dimensions are not given in range so we can use Eq. (2.1):

$$X[i][j] = \text{Base address} + \text{Size} * (\text{Number of columns before } (i-L_1) * \text{Number of columns} + \text{Number of columns before } (j-L_2))$$

Therefore

$$X[1][3] = 1002 + 2 * (3 * 1 + 3) = 1014$$

We can use Eq. (2.2) also to find the address of X[1][3]. Here we can write the dimension in the form of a range. For example, X[4][3] can be written as X[0···4][0···3], where $L_1 = 0$, $U_1 = 4$ (lower and upper bounds for first dimension) and $L_2 = 0$, $U_2 = 3$(lower and upper bounds for second dimension). Number of columns can be found using $U_2 - L_2 = 3 - 0 = 3$. Therefore

$$X[i][j] = \text{Base address} + \text{Size} * [\text{Number of rows before } (i-L_1) * \text{Number of columns} + \text{Number of columns before } (j-L_2)]$$

Substituting the values we get

$$X[1][3] = 1002 + 2 * (3 * (1 - 0) + (3 - 0)) = 1002 + 1$$

Note that we obtain the same address using both the formulas.

The array can be of any dimension, so we can generalize the method for calculating the address of *n*-dimensional array where each dimension is given in a range. Let us take an array $X[L_1 \cdots U_1][L_2 \cdots U_2][L_3 \cdots U_3] \cdots [L_n \cdots U_n]$ where L stands for the lower bound and U for the upper bound. The length of each dimension can be calculated using U – L + 1. So the length Len_1 of first dimension is $Len_1 = U_1 - L_1 + 1$. Similarly the length of other dimensions can be calculated and the total number of elements in the array can be found by multiplying the length of all the dimensions ($Len_1 * Len_2 * Len_3 * \cdots * Len_n$). Position of an element in *n*-dimensional array can be found by generalizing the formula of two-dimensional array as

$$X[i][\cdots j][k] = \text{Base address} + \text{Size} * [(i - L_1) * Len_2 * Len_3 * \cdots * Len_n + \cdots + (j - L_2) * Len_n + \cdots + (k - L_n)]$$

where "Base address" is the address of the first element (X[0][0]) of the array and "Size" is the number of bytes required for each element of the array.

Problem 4

Let X{1···3}{4···8}{–3···5} be a three-dimensional array of integer elements. The base address is 1002. Find the address of X{2}{5}{4}.

Solution
We have the formula

$$X[i][j][k] = \text{Base address} + \text{Size} * [(i - L_1) * Len_2 * Len_3 + (j - L_2) * Len_3 + (k - L_3)]$$

for *n*-dimensional array. For three-dimensional array, it will become

$$X[i][j][k] = \text{Base address} + \text{Size} * [(i - L_1) * Len_2 * Len_3 + (j - L_2) * Len_3 + (k - L_3)]$$

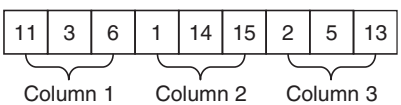
It requires the length of each dimension. Let us find length of each dimension first.

$$Len_2 = U_2 - L_2 = 8 - 4 = 4$$
$$Len_3 = U_3 - L_3 = 5 - (-3) = 8$$

Therefore

$$X[2][5][4] = 1002 + 2 * (4 * 8 * (2 - 1) + 8 * (5 - 4) + (4 - (-3)))$$
$$= 1002 + 2 * (32 + 8 + 7) = 1098$$

Column Major Order. In column major order, the elements are arranged column-wise, that is, first all the elements of first column are arranged, then all elements of second column and so on. This is another representation of two-dimensional arrays. For example, `int X[3][3] = {11, 3, 6, 1, 14, 15, 2, 5, 13};` is represented in column major order as shown below



This is the physical representation of a two-dimensional array in column major order. Here first all elements of column 1 are arranged, then all elements of column 2 and so on. Since all elements are stored in contiguous memory location, we can find the address of the element at a given position if we know the base address and the dimension of the array.

Address of Elements in Column Major Order. The address of an element in a two-dimensional array can be calculated using the following formula

$$X[i][j] = \text{Base address} + \text{Size} * (i + j * \text{Number of rows}) \quad (2.3)$$

If rows and columns are not given in the range but their dimensions are given in a range then we cannot find the address by this formula. The general formula for calculating the address of the element in a two-dimensional array if the elements are arranged in column major order is

$$X[i][j] = \text{Base address} + \text{Size} * ((i - L_1) + (j - L_2) * \text{Number of rows}) \quad (2.4)$$

where "Base address" is the address of the first element (X[0][0]) of the array, "Size" is the number of bytes required for an element of the array, L_1 is the lower bound of the rows and L_2 is the lower bound of the columns. Number of rows can be calculated by finding the difference between the upper bound and lower bound of the first dimension $[U_1 - L_1]$.

Problem 5

Consider an array X{4}{3} of integer elements with base address 1002 arranged in column major order. Find the address of X{1}{3}.

Solution

Since dimensions are not given in range, we can use Eq. (2.3):

```
X [ i ] [ j ] = B a s e   a d d r e s s + S i z e * ( i + j * N u m b e r
```

Substituting the values we get

```
X [ 1 ] [ 3 ] = 1 0 0 2 + 2 * ( 1 + 3 * 4 ) = 1 0 1 4
```

We can use Eq. (2.4) also to find the address of X[1][3]. We can write the dimensions in the form of range such as X[4][3] can be written as X[0···4][0···3], where $L_1 = 0$, $U_1 = 4$ (lower and upper bounds for first dimension) and $L_2 = 0$, $U_2 = 3$ (lower and upper bounds for second dimension). We can find the number of rows using $U_1 - L_1 = 4 - 0 = 4$. Therefore

```
X [ i ] [ j ] = B a s e   a d d r e s s + S i z e * [ ( i - L 1 ) + ( j - L 2 ) * N u m b e r   o f   r o w s ]
```

Substituting the values we get

```
X [ 1 ] [ 3 ] = 1 0 0 2 + 2 * [ ( 1 - 0 ) + ( 3 - 0 ) * 4 ] = 1 0 0 2
              =1 0 1 4
```

Note that we find the same address by using both the formulas.

The address of the elements in a three-dimensional array can also be found in a similar fashion by using a formula that is slightly different from the corresponding formula for the row major order representation. Position of an element in *n*-dimensional array can be found by generalizing the formula of two-dimensional arrays.

```
X [ i ] [ j ] [ k ] = B a s e   a d d r e s s + S i z e * ( ( i - L 1 ) * L e n 2 * L e n 3 + ( j - L 2 ) * L e n 3 + ( k - L 3 ) ]
```

(2.5)

where "Base address" is the address of the first element (X[0][0]) of the array and "Size" is the number of bytes required for each element of the array.

Problem 6

Let X{1···3}{4···8}{–3..5} be a three-dimensional array of integer elements arranged in column major order. The base address is 1002. Find the address of X{2}{5}{4}.

Solution
We have the formula

$$X[i][j]...[m_k] = Base\ address_1 + Size * (j - L_1 + i - L_1 Len_2 n ... Len_n * (m_3 - L_3) + ... + Len_n * (k_n - L_n)$$

for the *n*-dimensional array. For three-dimensional array it becomes

$$X[i][j][k] = Base\ address_1 + Size * (j - L_2 + Len_2 * (i - L_1 Len_2 * (k_3 - L_3)$$

Now to evaluate this we require the length of each dimension. Let us find length of each dimension first.

$$Len_1 = U_1 - L_1 = 3 - 1 = 2$$
$$Len_2 = U_2 - L_2 = 8 - 4 = 4$$

Therefore

$$X[2][5][4] = 1002 + 2 * [(2 - 1) + 2 * (5 - 4) + 2 * 4$$
$$= 1002 + 2 * (2 + 2 + 5 6) = 1122$$

Character Array (String)

The character array is a collection of characters. Character arrays are declared in the same manner as integer or float arrays. The general syntax for declaring a character arrayin C or C++ is as follows:

Syntax

```
Storage_class  data_type  array_name[
```

The difference between character array and other type of array is that the character array is always terminated by a null character ("\ 0") in C or C++. The null character is represented by a backslash followed by zero. Character arrays are also called strings. The null character is very important because it refers to the end of a string. For example

```
char  name[10];
```

Here "name" is an array of 10 characters. Actually we can store only 9 characters in this array because last location (space) is required to store the null character. Therefore we must declare the size of the character array one more than the actual number of characters we want to store.

An array of string is a two-dimensional array where the first dimension determines the number of strings and the second dimension determines the number of characters in each string. For example

```
char  empname[5][10];
```

Here empn acan have 5 elements (name) of length equal to 10 characters.

Initialization of Character Array

The character array can be initialized in same manner as integer or float array. The only thing we must remember is that the character arrays always terminate with the null character so we must keep an extra space for null character. For example

```
char name[7]={'R', 'A', 'J', 'E', 'S', 'H',
```

When we initialize the character array it is not necessary to mention the size. The C/C++ compiler automatically takes it equal to the number of characters assigned. So we can initialize the above array name as follows:

```
char name[]={'R', 'A', 'J', 'E', 'S', 'H',
```

Program 4 A program to demonstrate the initialization the character array.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
void main()
{
clrscr();
char name[15];
char stuname[]="Neha";
char junname[]={'S','m','r','i','t','i','\0
int i,len=0,lenR=0;
cout<<"Enter the name:";
gets(name);
for(i=0;stuname[i]!='\0';i++)
len=len+1;
for(i=0;name[i]!='\0';i++)
lenR=lenR+1;
int lenj=strlen(junname);
cout<<"\nName initialized string: "<<stunam
cout<<"\nThe number of characters in "<<stu
cout<<"\nName read: "<<name;
cout<<"\nThe number of characters in "<<na
cout<<"\nName initialized string (characte
cout<<"\nThe number of characters in "<<jun
getch();
}
```

Output

```
Enter the name: Prine jain
Name initialized string: Neha
The number of characters in Neha are: 4
Name read: Prince jain
The number of characters in Prince jain are
Name initialized string (character): Smriti
The number of characters in Smriti are: 6
```

We know that the character constants are always represented by using single quotes and the characters must be separated by a comma. Character arrays may also be initialized in the following manner.

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

We can also initialized the two-dimensional array in the same manner

```
char month[5][10]={"January", "February", "
```

Built-In Function for Strings

There are various built-in functions which are frequently used with the character array, one of which (strl) has been used in Program 4. These functions are defined in string.h in C or in C++ so we must include that file whenever we want to use such functions. Some of the functions are given in Table 1.

Table 1 String manipulator functions

Function	*Syntax*	*Description*	*Example*
strcpy	strcpy (dest,sour	This function copies the source string into destination string, where source and destination both must be character arrays.	char dest; char source[]="A strcpy(dest,sour
strcmp	strcmp(S1,S2);	The comparison starts with the first character of the string and continues until the corresponding character differs or end of the string is reached.	strcmp("Rajesh", "Kumar") Produce less tha if s1<s2 Produce 0 if s1= Produce more tha if s1>s2
strlen	strlen(string);	It counts the number of characters not including the null character.	strlen("Rajesh")

Program 5 A program in C++ to find the length of strings, copy and compare the strings using string library function.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
clrscr();
char S1[]="RAJESH";
char S2[]="KUMAR";
cout<<"copy KUMAR into RAJESH: "<<strcpy("
cout<<"The length of string is: "<<strlen
cout<<"Comparison of two strings: "<<str
<<endl;
```

```
cout<<"copy S2 into S1: "<<strcpy(S1,S2)<<e
cout<<"The length of S2 string is: "<<strle
cout<<"Comparison of two strings S1 and S1
getch();
}
```

Output

```
Copy KUMAR into RAJESH: KUMAR
The length of string is: 6
Comparison of two strings: 7
Copy S2 into S1: KUMAR
The length of S2 string is: 5
Comparison of two strings S1 and S1: 0
```

Sparse Array

Sparse array is a special array that contains the higher number of elements having zero values than those having non-zero values. If most of the elements are zero then the occurrence of zero elements in a large array is both a computational and storage inconvenience. It is wastage of the memory if we store all elements instead of dealing only with non-zero elements because most of the space will be occupied by zero elements of the array. Main objective of using arrays is to minimize the space requirement and improving the execution speed of a program. A two-dimensional sparse array is called the sparse matrix. A matrix is said to be sparse matrix if many of its elements are zero. There are various situations in which such sparse arrays are used with different names. A few such situations are:

1. The diagonal matrix in which all non-zero elements lie on the diagonal and rest of the elements are having zero values.
2. Upper triangular matrix in which all the elements on or above the diagonal have non-zero values and rest of the elements have zero values.
3. Lower triangular matrix in which all the elements on or below the diagonal have non-zero values and rest of the elements have zero values.

These are the cases of structured sparse matrices where all non-zero elements are arranged in a particular structure. A matrix that is not sparse is called dense matrix. The density of a matrix is the percentage of entries that are non-zero. In many fields of mathematics and engineering, sparse arrays are quite common. Therefore, it is worthwhile to develop efficient methods for handling such arrays. Instead of storing all the elements of sparse matrix it may be faster to represent the matrix compactly as a list of non-zero entries. There are two alternative representations that will explicitly store only the non-zero elements:

1. array representation;
2. dynamic representation.

Array Representation

In array representation, all non-zero elements are stored in another array of triplet where triplet contains the row number, the column number of the non-zero element and the value of the non-zero element. So the triplet can be represented by <Row, Col, Element>, where Row represents the row position, Col represents the column position of the non-zero element. The array is defined for the MAX number of non-zero elements.

```
struct triplet
{
int Row;
int Col;
float Element;
};
  typedef struct triplet Sparse[MAX]
```

The limitations of the array representations are as follows:

1. Though an array is very simple and efficient data structure but we need to know the number of non-zero elements in advance.
2. Insertion and deletion is not an easy task with array representation.

Dynamic Representation

All the non-zero elements are represented by the triplet where a triplet contains the row number, the column number of the non-zero element and the value of the non-zero element. In the dynamic representation, the triplet can be represented using nodes where each node contains four parts: one each for

Problem 7

Represent the following sparse matrix X as an array of triplet:

$$X = \begin{pmatrix} 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 7 & 0 \end{pmatrix}$$

Solution

The triplet contains <Row, Col, Element>. So X can be represented by another array say Y of six elements because there are only six non-zero elements in the given sparse matrix.

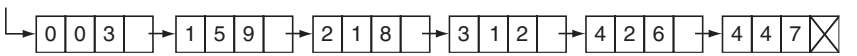
$$Y\ [\ 6\ ] = \{<0, 3, 3>, <1, 5, 9>, <2, 1, 8>, <3, 1, 2>, <4, 2, 6>, <4, 4, 7>\}$$

As we can see the first non-zero element 5 occurs in the 0th row and 3rd column. Similarly other triplets can be found. They are shown in Table 2.

Table 2 The triplet representation of sparse array

Row	*Column*	*Element*
0	0	3
1	5	9
2	1	8
3	1	2
4	2	6
4	4	7

Row, Col, Element and next, where next contains the address of the next node or triplet as shown below.



Special Ty pes of Matrices

1. **Square matrix:** A matrix in which the numbers of rows equals the number of columns is called square matrix. For example,

1	4	7
2	5	8
3	6	9

2. **Diagonal matrix:** A matrix in which only diagonal elements are non-zero is called a diagonal matrix. For example,

1	0	0
0	5	0
0	0	9

3. **Row matrix:** If the matrix consists of only one row then it is called row matrix. For example,

1	4	7

4. **Zero matrix:** A matrix in which all the elements have zero value is called zero matrix. For example,

0	0	0
0	0	0
0	0	0

5. **Upper triangular matrix:** A matrix in which all the non-zero elements occur only on or above the diagonal is called upper triangular matrix. For example,

1	2	3
0	7	5
0	0	6

6. **Lower triangular matrix:** A matrix in which all the non-zero elements occur on or below the diagonal is called lower triangular matrix. For example,

0	0	0
0	0	0
0	0	0

7. **Scalar matrix:** A diagonal matrix in which all diagonal elements are same is called scalar matrix. For example,

7	0	0
0	7	0
0	0	7

8. **Identity or unit matrix:** A diagonal matrix in which all diagonal elements are 1 is called unit matrix. For example,

1	0	0
0	1	0
0	0	1

Solved Examples

Example 1 *Write a program in C to perform addition, subtraction, multiplication, transpose and merging operations on a matrix.*

Solution

```
#  i n c l u d e  < s t d i o . h >
#  i n c l u d e  < s t d l i b . h >
#  i n c l u d e  < c o n i o . h >
f l o a t  A [ 3 ] [ 3 ] ;
f l o a t  B [ 3 ] [ 3 ] ;
f l o a t  C [ 3 ] [ 3 ] ;
i n t  i , j , k ;
f l o a t  t e m p [ 3 ] [ 6 ] = { 0 } ;
```

```
void display(float X[][3])
{
printf("The Resultant Matrix is \n");/*
                                    cout<< in C++*/
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("%f",X[i][j]);
printf("  ");
}
printf("\n");
}
}
void ReadMatrixA()
{
printf(" Enter the values of the matrix A
      :\n");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%f",&A[i][j]); /*Replacement for ci
}
}
}
void ReadMatrixB()
{
printf(" Enter the values of the matrix B
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%f",&B[i][j]);
}
}
}
void Add()
{
ReadMatrixA();
ReadMatrixB();
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
C[i][j]=A[i][j]+B[i][j];
}
```

```
display(C);
}
void Subtract()
{
ReadMatrixA();
ReadMatrixB();
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
C[i][j]=A[i][j]-B[i][j];
}
display(C);
}
void Multiply()
{
ReadMatrixA();
ReadMatrixB();
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
C[i][j]=0.0;
for(k=0;k<3;k++)
{
C[i][j]=C[i][j]+A[i][k]*B[k][j];
}
}
}
display(C);
}
void Transpose()
{
ReadMatrixA();
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
C[j][i]=A[i][j];
}
display(C);
}
void Merge()
{
ReadMatrixA();
ReadMatrixB();
printf("\n The merging matrix_c of matrix_
```

```
printf("  (  A,B  )=");
for(i=0;i<3;i++)
{
for(j=0;j<6;j++)
{
if(j<3)
temp[i][j]=A[i][j];
else
temp[i][j]=B[i][j-3];
}
}
for(i=0;i<3;i++)
{
for(j=0;j<6;j++)
{
printf("%f",temp[i][j]);
printf("      ");
}
printf("\n");
}
}
void main()
{
int operation;
clrscr( );
do
{
printf(" Select one of the following :\n "
printf("1. Addition  \n");
printf("2. Subtraction \n");
printf("3. Multiplication \n");
printf("4. Transpose \n");
printf("5. Merging Metrices  \n");
printf("0  Exit\n");
printf("Enter your Choice=");
scanf("%d",&operation);
clrscr();
switch(operation)
{
case 0:
exit(0);
case 1 :
printf("  A+B  =\n");
Add();
break;
```

```
case 2 :
printf(" A - B =\n");
Subtract();
break;
case 3 :
printf(" A*B =\n");
Multiply();
break;
case 4 :
printf(" A'=\n");
Transpose();
break;
case 5 :
printf("(A,B)=\n");
Merge();
break;
}
getch();
clrscr();
}while(operation>6||operation!=0);
getch();
}
```

Output

```
Select one of the following :
1. Addition
2. Subtraction
3. Multiplication
4. Transpose
5. Merging Metrices
0  Exit
Enter your Choice=1
Enter the values of the matrix A in row r
2 3 4 5 6 1 2 3 4
Enter the values of the matrix B row by r
5 6 7 8 2 1 3 4 5
The Resultant Matrix is
 7.000000     9.000000     11.000000
13.000000     8.000000      2.000000
 5.000000     7.000000      9.000000
Select one of the following :
1. Addition
2. Subtraction
3. Multiplication
4. Transpose
5. Merging Metrices
0  Exit
Enter your Choice=0
```

Example 2 *Write a program in C++ using array to calculate the mean, standard deviation and the variance of the given elements.*

Solution

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
void main()
{
clrscr();
float X[20], mean,variance,stdev;
int i,num;
float sumsquare=0,sum=0;
cout<<"Enter size of the array:";
cin>>num;
cout<<"**Enter array elements**\n";
for(i=0;i<num;i++)
{
cin>>X[i];
sum=sum+X[i];
}
mean=sum/num;
clrscr();//Clear screen before displaying t
cout<<"Array elements enterd are:\n";
for(i=0;i<num;i++)
{
cout<<X[i]<<"    ";
}
cout<<"\n\nThe mean calculated is:"<<mean;
sumsquare=sumsquare+pow((mean-X[i]),2);
variance=sumsquare/num;
cout<<"\n\nThe varaince calculated is:"<<va
stdev=sqrt(variance);
cout<<"\n\nThe standard deviation calculate
getch();
}
```

Example 3 *Write a program to find the sum of diagonal elements, upper triangular elements and the lower triangular elements.*

Solution

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
```

```
void main()
{
clrscr();
float X[3][3];
int i,j,num;
float sumdgl=0,sumupper=0,sumlower=0;
cout<<"Enter size of the square matrix:";
cin>>num;
cout<<"**Enter matrix elements**\n";
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
{
cin>>X[i][j];
if(i==j)
{
sumdgl=sumdgl+X[i][j];
}
else if(i<j)
{
sumupper=sumupper+X[i][j];
}
else
{
sumlower=sumlower+X[i][j];
}
}
}
clrscr();//Clear screen before displaying
cout<<"**matix elements enterd are**\n";
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
cout<<X[i][j]<<"    ";
cout<<endl;
}
cout<<"\n\nThe sum of diagonal elements
cout<<"\n\nThe sum of upper tringular mat
cout<<"\n\nThe sum of lower traingular ma
getch();
}
```

Example 4 *Write a program to check whether a given matrix is orthogonal or not. Use the formula* $A*A^T = I$.

Solution

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
void main()
{
clrscr();
float X[3][3],Y[3][3],Z[3][3];
int i,j,num,flag=0;
cout<<"Enter size of the square matrix:";
cin>>num;
cout<<"**Enter elements for matrix X**\n";
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
{
cin>>X[i][j];
}
}
cout<<"Find the transpose of X\n";
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
{
Y[i][j]=X[j][i];
}
}
//Find the multiplication of X and Y and st
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
{
Z[i][j]=0;
for(int k=0;k<num;k++)
{
Z[i][j]=Z[i][j]+X[i][k]*Y[k][j];
}
}
}
//Check for identity matrix
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
```

```
{
if(Z[i][i]==1 && Z[i][j]==0)
{
flag=1;
}
}
}
clrscr();//Clear screen before displaying
cout<<"**matix elements entered for X are
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
cout<<X[i][j]<<"    ";
cout<<endl;
}
cout<<"**Transpose of matrix X is**\n";
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
cout<<Y[i][j]<<"    ";
cout<<endl;
}
cout<<"**The mulplication of X and Y is*
for(i=0;i<num;i++)
{
for(j=0;j<num;j++)
cout<<Z[i][j]<<"    ";
cout<<endl;
}
if(flag!=1)
cout<"\n The given matrix is not orthogon
else
cout<<"\The given matrix is orthogonal";
getch();
}
```

Example 5 *Write a program in C++ to find the number of duplicate elements and remove the duplicate elements from the list.*

Solution

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

```
void main()
{
clrscr();
int X[10],size,num,i,j,flag=0;
cout<<"Enter the size of the array:";
cin>>num;
size=num;
cout<<"Enter the elements of the array\n";
for(i=0;i<num;i++)
cin>>X[i];
clrscr();
cout<<"The elements entered are ";
for(i=0;i<num;i++)
cout<<X[i]<<"   ";
for(i=0;i<num-1;i++)
{
for(j=i+1;j<num;j++)
{
if(X[i]==X[i+1])
{
num=num-1;
for(int k=j;k<num;k++)
X[k]=X[k+1];
flag=1;
j=j-1;
}
}
}
if(flag==0)
cout<<"\n\nNo duplicates found in the arra
else
{
cout<<"\n\nThere are "<<size-num<<" duplic
in the array";
cout<<"\n\nAfter deleting duplicate elemer
array is\n";
for(i=0;i<num;i++)
cout<<X[i]<<"   ";
cout<<endl;
}
getch();
}
```

Example 6 *Write a program to concatenate two given strings.*

Solution

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
void main()
{
clrscr();
char A[10],B[20],C[40];
int i,j;
cout<<"**Enter the first string**\n";
gets(A);
cout<<"**Enter the second string**\n";
gets(B);
for(i=0;A[i]!='\0';i++)
C[i]=A[i];
C[i]=' ';
for(j=0;B[j]!='\0';j++)
C[j+i+1]=B[j];
C[j+i+1]='\0';
cout<<"**The concatenated string is**\n";
puts(C);
getch();
}
```

Example 7 *Write a program to count the number of characters, number of words in a string and to check whether the given string is a palindrome or not.*

Solution

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<process.h>
void main()
{
clrscr();
char A[10];
int i,j,k,word=1;
cout<<"**Enter the first string**\n";
gets(A);
for(i=0;A[i]!='\0';i++);
//The last value of i will be equal to the
```

```
cout<<"The length(No of characters)in the
for(i=0;A[i]!='\0';i++)
{
if(A[i]==' ')
{
word=word+1;
}
}
cout<<"\nThe number of words in the string
for(k=i-1,j=0;k>i/2;k--,j++)
{
if(A[k]!=A[j])
{
cout<<"\nString is not palindrome";
getch();
exit(1);
}
}
cout<<"\nString is palindrome";
getch();
}
```

Summary

1. An array is a collection of similar items that are referenced by a common name.
2. Array may be categorized into two categories: one-dimensional and multidimensional.
3. A sparse array is an array which contains majority of non-zero elements.
4. The array elements may be arranged in row-major order or column-major order.
5. In C/C++, the index value for first element is always 0; it means that the array index will vary between 0 and n–1, where n is the size of the array.
6. All strings in C/C++ terminate with a null character so we must make sure that the array size be enough to include the null.
7. An array name always refers to the base address.
8. Array elements are stored in contiguous memory locations.
9. Array elements can be initialized by providing the values at the time of declaration.

Key Terms

Array
String
Null character
One-dimensional array
Two-dimensional array
Functions and arrays
Initialization of array
Unsized array initialization
Row-major order
Column-major order
Sparse arrays

Multiple-Choice Questions

1. Array is a collection of
   a. integer data items.
   b. similar data items.
   c. dissimilar data items.
   d. built-in data types.

2. Array is a
   a. derived data type.
   b. built-in data type.
   c. enumerated data type.
   d. user-defined data type.

3. Array elements are accessed by using
   a. subscript.
   b. dot operator.
   c. arrow operator.
   d. elements name.

4. Array elements are stored in
   a. contiguous memory locations.
   b. randomly.
   c. integer array.
   d. structure.

5. The null character is represent by
   a. \ 0 (zero).
   b. /0 (zero).
   c. \ o (o alphabet).
   d. 0 (zero).

6. Which of the array declarations is valid?
   a. `int a[3];`
   b. `int [3];`
   c. `a[3] of int`
   d. `int a[0-3];`

7. An array is passed as a parameter to a function using
   a. always call by value.
   b. always call by reference.
   c. call by value or call by reference.
   d. none of the above.

8. ASCII value for null character is
   a. 0 (zero)
   b. 48
   c. 97
   d. 68

9. To read a string that consists of spaces, we use
   a. `cin>>`
   b. `gets()`
   c. `cout<<`
   d. `puts()`

10. Which header file is required to use `exit?()`
    a. `stdio.h`
    b. `ctype.h`
    c. `iostream.h`
    d. `process.h`

11. Which header file is required to use `get?s`
    a. `stdio.h`
    b. `string.h`
    c. `iostream.h`
    d. `process.h`

12. `int A={51], 4 , W`hat is value of `A[3?]`
    a. 0
    b. garbage
    c. 6
    d. 4

13. `int A[3] = {.1W`hat is value of `A[3?]`
    a. 0
    b. garbage
    c. 6
    d. 4

14. Which of the following is not correct?
    a. `char A[] = "Umang";`
    b. `char A[6] = "Umang";`
    c. `char A{6U= , 'm', 'a'`
       `'n', 'g', '\0'}`
    d. `char A[6] = 'Umang';`

15. For the command `char` wh`ich of the following is correct?
    a. A = "A";
    b. A = "X";
    c. A = ' A';
    d. A = X;

16. What is the purpose of the `cin` object?
    a. To represent an input stream.
    b. To represent an output stream.
    c. To represent the process of input.
    d. To act as a receptacle to put values in.

17. The number of elements in a one-dimensional array can be calculated by
    a. Lower bound – Upper bound + 1.
    b. Lower bound + Upper bound – 1.
    c. Upper bound + Lower bound – 1.
    d. Upper bound – Lower bound + 1.

18. The address of X[i][j] in a one-dimensional array can be calculated by
    a. `Base + Size * [No. of rows * (i - 1) + (j - 1)]`
    b. `Base + Size * [No. of columns * (i - 1) + (j - 1)]`
    c. `Base + Size * [No. of rows * (j - 1) + (i - 1)]`
    d. cannot be calculated.

19. The address of X[i][j] in a one-dimensional array arranged in row-major order can be calculated by
    a. `Base + S*[No. of rows * (i - 1) + (j - 1)]`
    b. `Base + S*[No. of columns * (i - 1) + (j - 1)]`
    c. `Base + Size * [No. of r * (j - 1) + (i - 1)]`
    d. cannot be calculated

20. The array which contains majority of non-zero elements is
    a. scalar array.
    b. sparse array.
    c. orthogonal array.
    d. none of the above.

Review Questions

1. Describe the various operations associated with arrays.
2. Explain the memory representation of one-dimensional array.
3. Explain the memory representation of two-dimensional array.
4. Write the pros and cons of using arrays.
5. What do you mean by sparse array?
6. What does the array name refer to?
7. What is an array and how is it declared and initialized? What are multidimensional arrays? [VTU – Sep 1999]
8. What are the rules to be followed while using arrays? [Anna – 2006–07]
9. In which order the two-dimensional array elements are initialized in the array?
10. How is the address of an element calculated in an *n*-dimensional array arranged in row major order?
11. What is the ASCII value for the null character?
12. Find the number of elements in X[7···19] and the address of fifth element.
13. Explain the significance of array. What are the different types of arrays?
14. Write a program to find the inverse of a given matrix.
15. How is character array different from other type of arrays?
16. Can a multidimensional array be passed as an argument to a function?
17. Can a group of strings be stored in a two-dimensional array?

18. What is the purpose of subscript in an array?
19. Write an algorithm to find the sum of the elements above the diagonal of the given matrix say X.
20. What is the character string? Define the major operations on the string.
21. Let an array X[11][8] be stored in column major order and X[2][2] be stored at 1024 and X[3][3] at 1084. Find the addresses of X[5][3] and X[1][1].
22. Let X[10][20] be an array of float elements whose base address is 1002. Determine the address of X[3][6] when the array elements are arranged in
    a. row major order.
    b. column major order.
23. Let X[–3···5][4···10][2···6] be an array of float elements whose base address is 1002. Determine the address of X[3][6][4] when the array elements are arranged in
    a. row major order.
    b. column major order.
24. What is the procedure for calculating the address of an element in any two-dimensional array?
25. Describe various operations that can be performed on arrays.
26. Calculate the address of X[4][3] in a two-dimensional array X[1···5][1···4] stored in row major order. Assume the base address to be 1000 and that each element requires 4 words of storage.
27. Draw a linked and a vector representation of the sparse matrix of the following matrix.

$$\begin{bmatrix} 0 & 12 & 0 & 0 \\ 0 & 11 & 0 & 0 \\ 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

28. Implement transposition of a sparse matrix in C language.
29. Explain memory addressing scheme(s) for two-dimensional arrays with suitable example(s). [Anna – 2006–07]

Programming Assignments

1. Write a program in C++ to search an element in a given two-dimensional array.
2. Write a program in C to search and display the position of an element in a one-dimensional array.
3. Write a program in C to calculate the mean, standard deviation and the variance of the given elements using array.
4. Write a program in C to find the sum of diagonal elements, upper triangular elements and lower triangular elements.
5. Write a program in C to check whether the given matrix is orthogonal or not. Use the formula $A*A^T = I$.
6. Write a program in C++ to multiply two matrices if multiplication is possible.
7. Write a program in C++ to store and display sparse array.
8. Write a program in C to find the number of duplicate elements and remove the duplicate elements from the list.
9. Write a program in C to add two sparse arrays.
10. Write a program in C++ to count the number of characters, number of words in the string and to check whether the given string is a palindrome or not.
11. Write a program in C++ to reverse the string.

Answers

Multiple-Choice Questions

1. (b)
2. (a)
3. (a)
4. (a)
5. (a)
6. (a)
7. (b)
8. (a)
9. (b)
10. (d)
11. (a)
12. (a)
13. (b)
14. (d)
15. (c)
16. (a)
17. (d)
18. (d)
19. (a)
20. (b)

3 Introduction to Structure, Class and Template

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Concept of structures.
- Dot operator.
- Array and structures.
- Concept of union.
- Difference between structure and union.
- Class and objects.
- Difference between structure and class.
- Constructor and destructor.
- Class and object templates.
- Different types of template and class template.

3.1 Introduction to Structure

As discussed in Chapter 2, an array is a collection of similar items. We cannot keep different kinds of data items in an array. Sometimes, it is required to process together the data elements of different types which are logically related. Structure is used to handle such a situation. A structure is a collection of different types of data elements which are related together. The data items in a structure are called the members of the structure. In other words, we can say that similar or different data types can be grouped to form a structure to represent a single entity. For example, suppose we want to maintain the following information about each book in the library:

1. Title.
2. Author.
3. Publication.
4. Publication year.
5. Price.

A group of these five elements is required to define each book in the library where *Title*, *Author* and *Publication* can be character arrays, *Publication year* can be of integer type and *Price* can be of float type. The collection of these five elements may be defined as a structure named "book". Here book is a collection of different types of data elements. Similarly, distance can be measured in both meter and centimeter. These units may be of integer type but these two elements will measure distance. Therefore, we should represent distance as a structure of two elements – meter and centimeter. Table 1 highlights the differences between arrays and structures.

Table 1 Comparison of arrays and structures

Array	*Structure*
Array is a collection of similar data items.	Structure is a collection of similar or different data items.
Array elements are accessed by their position in the array.	Each structure element has a different name.
Array name is not used as a data type.	Structure name is used to define the variable of that type.
Memory is allocated at the time of array declaration.	Memory is not allocated when the structure is declared but rather when the structure variable is declared.
Arrays have finite number of elements of similar type.	Structures also have finite number of members of different type.
The entire array can be referred using the array name because array name refers to the base address.	The entire collection of elements can be referred using the structure name.
An array of structure type may be defined.	A structure may contain an array as a member.

Declaration of a Structure

The general format for structure declaration in C or C++ is as follows:

```
struct
structure_name
{
data type field 1;
data type field 2;
  .
  .
  .
data type field n;
};
```

```
struct book                    keyword
{                              structure tag
char title[20];                first member, a char array
char author[10];               second member, a char array
char publication[20];          third member, a char array
int pubyear;                   fourth member, an int
float price                    fifth member, float
};                             end of definition
```

A structure definition is specified by the keyword `struct` which is followed by the name for the structure, and then the members of the structure with their data type specification enclosed in a set of braces. The definition is finally terminated by a semicolon. The structure name is also called "tag of structure". Sometimes the keyword `struct` is preceded by storage class specifier which is optional. The structure definition or the tag does not allocate memory. Memory is allocated when the variables of the structure tag type are created. To define a variable of structure type, we use the structure tag followed by the variable name. For example

`book X;` (in C++) or `struct book X;` (in C)

Here, `X` is a structure variable of `book` type. Therefore, `X` will have `title`, `author`, `publication`, `pubyear` and `price`. We can define the variable along with the structure definition. For example, consider cases (A) and (B).

```
struct date
{
int day;
int month;
int year;
}jdate,bdate;
```

Case (A)

```
struct
{
char title[20];
char author[10];
char publication[20];
int pubyear;
float price;
}X;
```

Case (B)

In the definition in Case (A), `date` is a structure tag and `jdate` and `bdate` are variables (instances) of `date` type. Since two structure variables have been declared, the memory space has been reserved for these two variables. Sometimes, the structure tag is skipped when only one variable is declared as shown in Case (B).

In C language we have to use the keyword `struct` with the structure tag so

```
struct book X;
```

is a correct declaration in C. We can use `typedef` to avoid repetitive use of `struct` keyword in declaring variables:

```
typedef struct book sample;
```

Now onwards we can use 'sample' to declare book type of structures.

The Dot Operator

Each member of a structure is accessed by using three components: a variable name followed by a dot and then the member name. The format for the dot operator is

```
structure variable.member name;
```

Once the structure variable is defined, its members can be accessed by using dot operator. For example, consider

```
bdate.month=11;
```

where structure variable `bdate` is followed by a dot and the field name or member name (`month`).

Initialization of a Structure

The structure variable can be initialized by assigning the values to the members in the same order in which they are declared in the structure. Therefore, whenever structure variable is initialized, first value is assigned to the first member, second value to the second member and so on. The initialization of a structure must be enclosed within a pair of braces. For example

```
book X={"OOPS","R K Shukla","WILEY",2008,25
```

Hence a compiler not only allocates the memory but also stores value in the structure members (Figure 1).

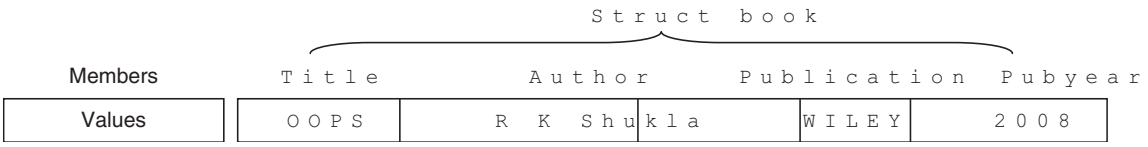


Figure 1 Memory allocation for a structure variable.

Array and Structures

Like the simple data type, an array can also be a member of a structure and array of structures can also be formed. When a structure consists of array elements, the array elements are treated in the same way as normal arrays are treated.

Program 1 A program in C++ to initialize the structure elements and display its contents using dot operator.

```
#include<iostream.h
#include<conio.h
#include<stdio.h
struct book
{
char title[20];
char author[10];
char publication[5];
int pubyear;
float price;
}Y;
void main()
{
clrscr();
book X={"OOPS", "R K Shukla", "WILEY", 20
cout<<"\n\nEnter information for Y";
cout<<"\nTitle: ";gets(Y.title);
cout<<"Author: ";gets(Y.author);
cout<<"Publication>>Y.publication;
cout<<"Publication ye>>Yr.:pub;yceianr;
cout<<"Price: >>Y.price;
clrscr();
cout<<"\n\n**The information initialisez for
cout<<"\nTitl<<X:.t"itle;
cout<<"\nAuth<<X:.a"uthor;
cout<<"\nPublica<<Xo.pub"lication;
cout<<"\nPublicatio<<Xy.epaub:ye"ar;
cout<<"\nPric<<X:.p"rice;
cout<<"\n\n**The information entered for Y i
cout<<"\nTitl<<Y:.t"itle;
```

```
cout<<"\nAuthor:".author;
cout<<"\nPublication:".publication;
cout<<"\nPublication year:".pubyear;
cout<<"\nPrice:".price;
getch();
}
```

Output

```
Enter information for Y
Title: Data structure
Author: Rajesh K Shukla
Publication: Wiley
Publication year: 2008
Price: 345.27
**The information initializes for X is **
Title: OOPS
Author: R K Shukla
Publication: WILEY
Publication year: 2008
Price: 250.949997
**The information entered for Y is **
Title: Data structure
Author: Rajesh K Shukla
Publication: Wiley
Publication year: 2008
Price: 345.269989
```

The array elements of a structure are also accessed by using the dot operator. For example

```
struct student
{
int rollno;
char name[20];
char branch[10];
int marks[5];
int age;
} X;
```

Here, the structure `student` contains three arrays: `name`, `branch` and `marks`. Arrays `name` and `branch` are of character type and array `marks` is of integer type. `X` is defined as a variable of `student` type. Now, suppose we want to access the marks in different subjects scored by `X`. In such cases, we have to write

```
X.marks[i]
```

where `i` is an index value that can vary from 1 to 5 or 0 to 4.

To declare an array of structure, we have to first define the structure and then we can define the array of that structure type. For example

```
struct student
{
int rollno;
char name[20];
char branch[10];
int marks[5];
int age;
};
student X[3]; or struct student X[3];
```

Here X is an array of three student type data. This implies that each element of X will have roll, name, branch, marks in five subjects and age. The elements of this array – X [0], X [1] and X [2] will be structure variables of type student. To access the marks of the first student in the third subject, we have to write X [0] . mark s [2]

Initialization of Array of Structure

An array of structures can be initialized in the same manner as normal array is initialized. For example, we can define an array of the following structure:

```
struct book
{
char title[20];
char author[10];
char publication[20];
int pubyear;
float price;
};
book X[2]={
        {"OOPS", "R K SHUKLA", "WILEY", 200
        {"DSA", "RAJESH SHUKLA", "WILEY", 2
             };
```

Or in C as

```
struct book X[2]={
        {"OOPS", "R K SHUKLA", "WILEY",    2
        {"DSA", "RAJESH SHUKLA", "WILEY",   2
                  };
```

Nested Structures

When a structure is declared as a member of another structure, then it is called a nested structure. Structures can be nested to any depth. Let us take the previous structure declaration of student. Now one more information is required, date of birth for each student. Consider as an example date of birth as 10 February, 2008. We can see that day (10) is an integer value, month (February) is a string and year (2008) is again an integer value; therefore, dat is a collection of three different types of elements, namely, day, month and year. The structure of dat can be defined as follows.

```
struct date
{
int day;
char month[10];
int year;
};
```

```
struct student
{
int rollno;
char name[20];
char branch[10];
int marks[5];
int age;
date birthdate
};
student X;
```

Accessing Nested Structure Members

Here `birthdate` (structure variable) is a member of `student` and so `student` is called a nested structure. The members of the nested structure are also accessed using the dot operator. To access the element of this nested structure we have to use the dot operator twice:

```
X.birthdate.day=04;          //04 is assigned to
gets(X.birthdate.month);     //To read month o
cout<<X.birthdate.year;      //To display year of
```

Function and Structure

The structure variables can also be passed to the function in two ways (call by value and call by reference) just like other variables. When the structure variable is passed by using call by value, the actual parameters do not change even if formal parameters are changed. When the structure variables are passed by using call by reference, the changes made in the formal parameter are also reflected in the actual parameter. The function can return a structure also. The structures are generally defined as global declaration so that structure tag can be used in any function. Apart from passing the whole structure variable, we can also pass the individual structure elements in two ways: call by value and call by reference.

Program 2 A program to read and display the students' information using array within structure, array of structure, nested structure and function.

```
#include<iostream.h
#include<conio.h>
#include<stdio.h>
struct date
{
int day;
char month[10];
int year;
};
struct student
{
```

```
int rollno;
char name[20];
char branch[20];
int marks[5];
date bdate;
int age;char result;
};
void main()
{
clrscr();
student X[2];
int total=0;
void display(student [ ]);
for(int j=0;j<2;j++)
{
cout<<"  **  Enter information of student "<<j+1<<"\n";
cout<<"\nEnter roll number: ";
cin>>X[j].rollno;
cout<<"Enter the name: ";
gets(X[j].name);
cout<<"Enter the branch: ";
gets(X[j].branch);
cout<<"Enter marks scored in different subje
for(int i=0;i<5;i++)
{
cin>>X[j].marks[i];
}
cout<<"\n*  Enter date of birth like 10 feb 2
cin>>X[j].bdate.day;
gets(X[j].bdate.month);
cin>>X[j].bdate.year;
cout<<"Enter the age: ";
cin>>X[j].age;
for(i=0;i<5;i++)
{
total=total+X[j].marks[i];
}
float avg=total/5;
if(avg>=75)
X[j].result='H';
else
    if(avg>=65)
      X[j].result='F';
```

```
    else
    X[j].result='S';
}
display(X);
getch();
}
void display(student X[2])
{
clrscr();
for(int j=0;j++)
{
cout<<"\n** RESULT FOR STUDENT ";
cout<<"\nRoll number:"<<X[j].rollno;
cout<<"\nName:"<<X[j].name;
cout<<"\nThe branch:"<<X[j].branch;
cout<<"\nMarks scored in different subjects:
for(int i=0;i++)
{
cout<<X[j].mark[i]<<"\t";
}
cout<<"\nDate of birth:";
cout<<X[j].bdate.day<<"-"<<X[j].bdate.month;
cout<<"-"<<X[j].bdate.year;
cout<<"\nAge:"<<X[j].age;
cout<<"\nOverall result:"<<X[j].result;
}
cout<<"\n(H: For honors, F: first division, S:
}
```

Output

```
** RESULT FOR STUDENT 1 **
Roll number: 111
Name: Anshu
The branch: CSE
Marks scored in different subjects:
88        89        85        78        76
Date of birth: 12-October-1988
Age: 20
Overall result: H
**RESULT FOR STUDENT 2 **
Roll number: 112
Name: Smriti
The branch: CSE
Marks scored in different subjects:
78        91        88        86        85
```

```
Date of birth: 25-November-1989
Age: 19
Overall result: H
(H: For honors, F: first division, S: sec
```

3.2 Introduction to Unions

A union is similar to structure in the sense that it is also a collection of different types of data. The difference between structure and union is that union stores values of different types in a single location, whereas each member of the structure has its own storage location. In union, space is reserved for the largest data type only. Therefore union provides a way to manipulate different kinds of data in a single area of storage. If a new assignment is made, the previous value is automatically erased. The general syntax for union declaration is as follows.

Syntax

```
Storage class union union_name
{
datatype member 1;
datatype member 2;
:::
datatype member n;
};
```

It can be noted that "union" is a keyword and "union_name" is any valid identifier. This is called the union tag. We can access the members of the union by using dot operator in the same manner as is used with the structure members. A union has only one active member at any given time, therefore it not possible to initialize the members of the union. A union can be a member of a structure, array and vice versa. The following should always be kept in mind: *When a union is declared as the member of a structure, it should not be the first member but the last one because the union only holds a value for one data type which requires a larger storage among their members*. If we define a union without a name or tag it is called an *anonymous union*.

3.3 Introduction to Class and Objects

Class is a special feature of C++ which is not supported by C. It is a user-defined data type like structure. It is an extension of structure of C. Where structure of C only consists of the data members of different types, a class consists of the data members and the member functions which perform some action on the data members. The data members can be of any type including built-in types (int, float, char), derived type (array, pointer) and user-defined type (structure or class). Member functions change the data members, create and destroy class variables. The variable of the class is called an *object*. The data members of the class are also called the *properties or state of an object* and the member functions are also called the *behavior or service of an object*. Let us take an example of vehicle class with properties color, price, gear, and tray

speed and no. of gears. An objects of class vehicle may be a car (color = red, price = 4.5 lakhs, brand = Maruti, category = LMV, max speed = 200 km, no. of gears = 6), truck (color = mixed, price = 8.5 lakhs, brand = TATA, category = HMV, max speed = 200 km, no. of gears = 6), motorcycle (color = black, price = 0.5 lakhs, brand = Bajaj, category = Two wheeler, max speed = 160 km, no. of gears = 4). The possible operations on these objects may be selling, purchasing and driving. The class supports various features like data abstraction, encapsulation, data hiding, modularity, inheritance and polymorphism of object-oriented programming.

The class is defined by a unique name, a collection of data members (called the attributes of the class) and the member functions (called the method of the class). An object is an instance or a variable of a class which is defined by the unique name (called the identity of an object), holding some value for corresponding data members (called states of an object) and the member functions (called the behavior of an object).

Declaration of a Class

The general syntax for declaration of a class is as follows.

Syntax

```
class classname
{
private:
  data members;
  member functions;
public:
  data members;
  member functions;
protected:
  data members;
  member functions;
};
```

The class declaration is similar to the structure declaration. Here `class` is a keyword followed by the `classname` which will be used to create the instances (object) of the class. There are two types of elements in the class. The data members are similar to the variables in the structures and the member functions act on the data members. The data members and member functions can be defined in three accessibility modes: `private`, `public` and `protected` in the class. The accessibility modes are followed by a colon. The data members and the member functions are enclosed within braces and the declaration is terminated by the semicolon. The data member or the member functions defined in `private` mode can be accessed only from within the class and the `public` members can be accessed from outside the class. The use of keyword `private` is optional. By default all members of a class are `private`. The `classname` is the tag name of the class that acts as a type specifier. Using the `class` tag name we can create the variables (objects) of the `class` tag types. It is a normal practice to put data members in the `private` access mode of the class, where they can only be accessed via member functions (methods) of the class. Methods or functions themselves appear in the `public` part of the class, so they can be accessed externally. They provide the interface to the class. A data member cannot be defined twice in the class, not even in two different modes.

Defining the Member Functions

The member functions are also called the methods of a class. The member functions can be defined in two ways:

1. **Inside the class:** The member functions may be defined inside the class like any normal function. Only small functions must be defined inside the class.
2. **Outside the class:** The member functions may also be defined outside the class by using scope resolution operator. Large functions must be defined outside the class.

The variables (objects) of the class may be declared in the same manner as the variables of the structures are declared. The `class` tag acts as a data type specifier. The general syntax for defining the object of a class is as follows.

Syntax

```
classname obj1, obj2;
```

where `obj1` and `obj2` are the instances of the `classname`

The `public` members of the class are accessed by the objects directly; the `private` and the `protected` members can only be accessed by the member function of the same class. To access the `public` members, the dot operator is used with the object. The objects can be declared at the same time when the class is defined by writing the object name before the semicolon in the class definition. An object of the unnamed class may also be defined just by writing the object name before the semicolon in the class definition.

Program 3 A program in C++ to read and display the distance in meter and centimeter using class and object.

```
#include<iostream.h
#include<conio.h>
class distance
{
int meter;
int cmeter;
public:
void inputdata()  //function is defined in
{
cout<<"Enter the distance (in meter<<and1;cent
cin>>meter>>cmeter; //private members are acce
the member function
}
void display();                    //function prot
};Y;                                //object Y d
```

```
void distance:: display() //member functio
the class
{
meter=meter+cmeter/100;
cmeter=cmeter%100;
cout<<"\nThe distance entered is:";
cout<<meter<<"M"<<cmeter<<"CM";
}
void main()
{
clrscr();
distance X;                    //object X decla
cout<<"\nObject X\n";
X.inputdata();                 //public member
X.display();
cout<<"\nObject Y\n";
Y.inputdata();
Y.display();
getch();
}
```

Output

```
Object X
Enter the distance (in meter and centimeter
2 3
7 8 1 2 3
The distance entered is: 148M 87CM
Object Y
Enter the distance (in meter and centimeter
2 4 5
3 4 5 6
The distance entered is: 279M 56CM
```

Array and Class

Class is a binding of data members and member functions. The data members can be of any type. They can be derived data types also such as arrays. Arrays can be `private` or `public` members of a class. If they are defined in `private` mode then only member functions can access them. If they are declared in the `public` mode then they can directly be accessed by the object of that class.

An object is an instance of a class. The collection of objects of the same class is called an array of the objects. An array of objects can be defined in the same manner as it is defined for any other type of data such as built-in data type or structure data type. We can use the usual array accessing methods to access the individual array elements and then the dot operator can be used to access the member of a class.

Program 4 A program in C++ to print the score card of two students using array of objects.

```
#include<iostream.h
#include<conio.h
#include<stdio.h
class student
{
int rollno;
char stuname[20];
char subname[5][20];
char branch[20];
int marks[5];
char grade;
public:
void inputdata()     // function is defined
{
cout<<"Enter the student's information";
cout<<"\nRoll no.: ";
cin>>rollno;
cout<<"\nStudent's name: ";
gets(stuname);
cout<<"\nBranch: ";
gets(branch);
cout<<"Enter five subjects name";
for(int j=0;j<5;j++)
{
cout<<"\nSubject name: ";
gets(subname[j]);
}
for(int k=0;k<5;k++)
{
cout<<"\nMarks in"<<k<<": "subject:";
cin>>marks[k];
}
}
void calculate();
void display()//function is defined insid
{
cout<<"\nRoll no.: ";
cout<<rollno;
cout<<"\nStudent's name: ";
puts(stuname);
cout<<"\nBranch: ";
puts(branch);
```

```
cout<<"** Score Card **\n";
for(int j=0;j<5;j++)
{
cout<<"\nSubject name: ";
cout<<subname[j];
cout<<"   Marks:";
cout<<marks[j];
}
cout<<"\nGrade:"<<grade;
}
};                                        //class declaratio
void student::calculate()//function is defi
{
int i, sum=0;
float avg;
for(i=0;i<5;i++)
{
sum=sum+marks[i];
}
avg=sum/5;
if(avg>=75)
{
grade='H';
}
else if(avg>=60&&avg<75)
{
grade='G';
}
else if(avg>=50&&avg<60)
{
grade='A';
}
else grade='F';
}
void main()
{
clrscr();
student X[2];
int i;
for(i=0;i<2;i++)
{
X[i].inputdata();
X[i].calculate();
}
```

```
clrscr();     //after clearing the screen o
for(i=0;i<2;i++)  //  function is shown
{
X[i].display();
}
getch();
}
```

Output

```
Roll no.: 1
Student's name: Arunima
Branch: CSE
** Score Card **
Subject name: ADC Marks: 88
Subject name: DCS Marks: 77
Subject name: DSA Marks: 86
Subject name: CSO Marks: 79
Subject name: DS Marks: 89
Grade: H
Roll no.: 2
Student's name: Suresh Kumar
Branch: IT
** Score Card **
Subject name: CSO Marks: 55
Subject name: DCS Marks: 56
Subject name: DS Marks: 58
Subject name: DSA Marks: 54
Subject name: ITC Marks: 51
Grade: A
```

Initialization of Members of a Class

We can initialize the object (variable of class type) by using constructor and destroy the object by using destructor. The constructor and destructor are special member functions because their name should be the same as the class name.

Constructors

Whenever an object is created, the constructor will be executed automatically. Constructors are used to allocate the memory for the newly created object and they can be overloaded so that different forms of initialization can be accommodated. If a class has a constructor, each object of that class will be initialized. The constructor (function) name must be same as the class name. It does not return any value (not even `void`). The parameters are optional. Constructors are defined in `public` access mode. They cannot be virtual or static. Constructors are defined like other member functions of a class. They may be defined inside the class or outside the class using scope resolution operator. The syntax used for constructors is as follows.

Syntax

```
class sername
{
private:
   ⋮ //Private data members and member f
protected:
   ⋮ //Protected data members and member
public:
   ⋮ //Public data members and member fu
username (parameters);//constructor
};
username :: username (parameters)
{       //Constructor is defined outside
body of the constructor;
}
```

There are different kinds of constructors such as

1. **Default constructor:** It is the constructor without any arguments (parameters).
2. **Parameterized constructor:** A constructor where parameters are passed is called parameterized constructor. When parameterized constructors are used in a program, it is necessary to pass the appropriate arguments when objects are created.
3. **Default argument constructors:** These are constructors where we can use the default arguments. The order of assignment should be from rightmost variable to leftmost variable. For example
   `Time(int hh, int mm, int ss=20);`
   When an object is created with some missing arguments, the default value is set for the missing parameters.
4. **Copy constructor:** When the initialization of one object is done by another object, this is called copy constructor. An argument for a copy constructor is a reference to an object of the same class.
5. **Dynamic constructor:** The constructors which allocate memory at run time are called dynamic constructors. They deal with the dynamically initialized of objects. The `new` operator (described in the Chapter 4 pointer and dynamic memory allocation) is used to allocate memory at run time.

Program 5 A program to initialize the objects using parameterized constructor.

```
#include<iostream.h>
#include<conio.h>
class distance
{
int meter;
int centme;
```

```
public:
//distance()//default constructor may be
//{ }                  otherwise it is supplied
distance(int met,int cen)//constructor wi
{
cout<<"Object is initialized and assigned the va
meter=met;
centme=cen;
}
distance(int m) // parameterized construct
{
cout<<"\nOne data member is initialized
required";
cout<<" for other\n";
meter=m;
cout<<"Enter the distance in centimeter:";
cin>>centme;
}
void display()
{
cout<<"\nThe distance is:";
cout<<meter<<", "<<centme;
}
};
void main()
{
clrscr();
int dm,dcm;
distance X=distance(3,8);   //explicit cal
X.display();
cout<<"\nEnter distance in meter and centime
cin>>dm>>dcm;
distance Y(dm,dcm); //implicit call
Y.display();
distance Z(6);//implicit call
Z.display();
getch();
}
```

Output

```
Object is initialized and assigned the va
The distance is: 3, 8
Enter distance in meter and centimeter: 4
Object is initialized and assigned the va
```

```
The distance is: 41, 56
One data member is initialized and input is
Enter the distance in centimeter: 99
The distance is: 6, 99
```

Destructors

When the variables of built-in data types are not needed, they are automatically destroyed by the compiler. Similarly we can destroy the objects created by the user-defined data type 'class'. To destroy the object, a special member function is used called the destructor. This function is automatically called whenever an object of the class to which it belongs goes out of existence. The destructor function releases the memory occupied by the object in the heap. Destructors have the same name as their class name. The destructor name is preceded by the *tilde* sign. It does not take any argument and it does not return any value (not even void). It is automatically called by the compiler whenever the object is destroyed. If the object is defined inside the block (local) then destructor is called when the block gets over, and if the object is globally defined then destructor is called when program terminates. Destructors are used for efficient memory utilization.

Program 6 A program to demonstrate the use of destructor.

```
#include<iostream.h
#include<conio.h
class distance
{
int meter;
int centme;
public:
distance()                          //default constru
{}
distance(int met,int cen)// constructor wit
{
cout<<"Object is initialized and assigned the
meter=met;
centme=cen;
}
~distance()                        //Destructor
{
cout<<"Destructor is invoked";
getch();
}
distance(int m) // parameterized constructo
{
cout<<"\none data member is initialized and
for oth<<endl;
```

```
meter=m;
cout<<"Enter the distance in centimeter:";
cin>>centme;
}
void display()
{
cout<<"\nThe distance is:";
cout<<meter<<", "<<centme;
}
};
void main()
{
clrscr();
int dm,dcm;
distance X=distance(3,8);   //explicit cal
X.display();
cout<<"\nEnter distance in meter and centimeter:";
cin>>dm>>dcm;
distance Y(dm,dcm); //implicit call
Y.display();
distance Z(6);//implicit call
Z.display();
getch();
}
```

Output

```
Object is initialized and assigned the va
The distance is: 3, 8
Enter distance in meter and centimeter: 2
Object is initialized and assigned the va
The distance is: 2, 3
One data member is initialized and input
Enter the distance in centimeter: 4
The distance is: 6, 4
Destructor is invoked
Destructor is invoked
Destructor is invoked
```

3.4 Introduction to Template

Object-oriented programming languages like C++ provide another very important feature in which same definition is executed for the different range of parameters. This feature is called generacity in which a single generic function or a class is able to process all data types including user-defined data types. In C++ the generacity is implemented by using template. Template in C++ allows us to operate on any data type for which the internal implementation is appropriate. Templates are very useful for

small general-purpose functions especially inline functions. They are also useful for the generalized implementation of algorithms. A very important application of template is that it is used to represent abstract data types.

Templates can be defined for both classes and functions. There are two types of template in C++:

1. function template;
2. class template.

Function Template

In function template, only one function signature needs to be created for a family of similar functions. The C++ compiler automatically generates the required functions for handling the individual data types. Function template avoids the unnecessary repetition of the source code and it is easy to use than class template. It does not specify the actual data types of the arguments that a function accepts but it uses a generic or parameterized data type. In the functions template at least one formal argument must be generic. The general syntax for function template is as follows:

Syntax

```
template <class data type (T)
return_type function_name (parameters
{
function body ;
}
```

where `template` and `class` are keywords.

The generic data type is generally represented as `T`. The generic data type (`T`) and the keyword class must be enclosed in pointed or angular brackets (<…>).

Program 7 A program to find out the largest value among the three given numbers using function template.

```
#include<iostream.h
#include<conio.h>
template<class T
T greatest(Tx, Ty, Tz)
{
if (x>y)
if (x>z)
return(x);
else
return(z);
else
if (y>z)
return(y);
else
```

```
return(z);
}
void main()
{
clrscr();
int ch;
char choice;
do
{
cout<<"Find the greatest among numbers"
cout<<"1. Among integers"
cout<<"2. Among floats"
cout<<"3. Among doubles"
cout<<"Enter your choice: ";
cin>>ch;
switch(ch)
{
case 1:
cout<<"Enter integer numbers: "
int a,b,c,maxI;
cin>>a>>b>>c;
maxI=greatest(a,b,c);
cout<<"The greatest number is: "<<maxI;
break;
case 2:
cout<<"Enter float numbers: "
float x,y,z,maxF;
cin>>x>>y>>z;
maxF=greatest(x,y,z);
cout<<"The greatest number is: "<<maxF;
break;
case 3:
cout<<"Enter double numbers: "
double m,n,q,maxD;
cin>>m>>n>>q;
maxD=greatest(m,n,q);
cout<<"The greatest number is: "<<maxD;
}
cout<<"\nWant to continue (y/n): ";
cin>>choice;
}while(choice=='y'||choice=='Y');
getch();
}
```

Output

```
Find the greatest numbers
1. Among integer numbers:
2. Among float numbers:
3. Among double numbers:
Enter your choice: 1
Enter integer numbers: 2 4 6
The greatest number is: 6
Want to continue (y/n): y
   Find the greatest numbers
1. Among integer numbers:
2. Among float numbers:
3. Among double numbers:
Enter your choice: 2
Enter float numbers: 3.4 5.8 2.1
The greatest number is: 5.8
Want to continue (y/n): n
```

Program 8 To find the largest among the three given numbers using normal function and the template function in the same program.

```
#include<iostream.h
#include<conio.h
template<class T
T greatest(Tx, Ty, Tz)     // template functi
{
if(x>y)
if(x>z)
return(x);
else
return(z);
else
if(y>z)
return(y);
else
return(z);
}
void greatest(float x, float y, float z)
{
if(x>y)
if(x>z)
```

```
cout<<"\nThe greatest number is: "
else
cout<<"\nThe greatest number is: "
else
if (y>z)
cout<<"\nThe greatest number is: "
else
cout<<"\nThe greatest number is: "
}
void main()
{
clrscr();
int ch;
char choice;
do
{
cout<<" Find the greatest among numbers"
cout<<"1. Among integer numbers:"
cout<<"2. Among float numbers:"
cout<<"3. Among double numbers:"
cout<<"Enter your choice: ";
cin>>ch;
switch(ch)
{
case 1:
cout<<"Enter integer numbers: "
int a,b,c,maxI;
cin>>a>>b>>c;
maxI=greatest(a,b,c);
cout<<"The greatest number is: "<<maxI;
break;
case 2:
cout<<"Enter float numbers: "
float x,y,z,maxF;
cin>>x>>y>>z;
greatest(x,y,z);
break;
case 3:
cout<<"Enter double numbers: "
double m,n,q,maxD;
cin>>m>>n>>q;
maxD=greatest(m,n,q);
cout<<"The greatest number is: "
}
```

```
cout<<"\nWant to continue (y/n): ";
cin>>choice;
}while(choice=='y'||choice=='Y');
getch();
}
```

Output

```
Find the greatest numbers
1. Among integer numbers:
2. Among float numbers:
3. Among double numbers:
Enter your choice: 2
Enter float numbers: 3.4 5.7 3.9
The greatest number is: 5.7
Want to continue (y/n): y
 Find the greatest numbers
1. Among integer numbers:
2. Among float numbers:
3. Among double numbers:
Enter your choice: 1
Enter integer numbers: 4 5 9
The greatest number is: 9
Want to continue (y/n): n
```

Template Function with Multiple Parameters

More than one generic type parameter may also be used in the template function. These parameters are separated using a comma. The general syntax for such template function is given as follows.

Syntax

```
template<class T1, class T2
return_type function_name(parameters of
{
function body;
}
```

Program 9 A program to display the name and salary or employee no. of the employee by using multitype parameters in the function template.

```
#include<iostream.h>
#include<conio.h>
template<class T1,class T2
```

```
void display(T1 *x,T2 y)
{
cout<<"\nName of the emp: "<<x;
cout<<"\nEno or salary: "<<y;
}
void main()
{
clrscr();
display("Rajesh",101);
display("Suresh",42102.34);
getch();
}
```

Output

```
Name of the emp: Rajesh
Eno or salary: 101
Name of the emp: Suresh
Eno or salary: 42102.34
```

Class Template

When a single class is written for a family of similar classes then such class is called the template class. C++ class templates are used where we have multiple copies of code for different data types with the same logic. The general syntax for class template is as follows.

Syntax

```
template<class datatype(T)
class classname
{
body of the class with generic type
appropriate
};
```

where `template` and `class` are keywords and the data type (`T`) is a generic data type.

The definition of class template is very similar to normal class definition except for prefix `template <class>`. `T` may be substituted by any data type including the user-defined types. This process of creating a specific class from a class `template` is called *instantiation*. The syntax for defining an object of a `template` class is as follows.

Syntax

```
class <atype> object name (parameters);
```

Program 10 A program to find the greatest number among the three given numbers by using the class template.

```
#include<iostream.h>
#include<conio.h>
template<class T>
class sample
{
T first,second,third;
public:
void inputdata()
{
cout<<"\nEnter three values";
cin>>first>>second>>third;
}
T greatest()
{
if(first>second && first>third)
cout<<"\n The greatest number is: "
else if(second>first && second>third)
cout<<"\n The greatest number is: "
else if(third>first && third>second)
cout<<"\n The greatest number is: "
}
};
void main()
{
clrscr();
sample<int>X;
X.inputdata();
X.greatest();
sample<float>Y;
Y.inputdata();
Y.greatest();
getch();
}
```

Output

```
Enter three values 3 4 5
The greatest number is: 5
Enter three values 3.5 5.8 9.2
The greatest number is: 9.2
```

Member Function Defined Outside of the Class Template

The member functions of the class template may be defined either within the class or outside the class by giving the full template definition. When the member function of a class is declared outside the class, the function definition contains the keyword template.

Constructor and Class Template

Constructor is a special function which has the same name as of the class name. We can use the generic parameters with the constructor also in a similar manner as it is done with the other functions of the class template. The only difference is that we do not need to call the constructor function. If a parameterized constructor is used then we pass the parameters with the object.

Program 11 A program to find out the greatest among the three given numbers using the class template and parameterized constructor.

```
#include<iostream.h>
#include<conio.h>
template<class T>
class example
{
T a,b,c;
public:
example(Tx, Ty, Tz)
{
a=x;
b=y;
c=z;
}
void inputdata()
{
cout<<"\nEnter values";
cin>>a>>b>>c;
}
T greatest()
{
if(>b)
if(>a)
return(a);
else
return(c);
else if(b
return(b);
else
return(c);
}
};
```

```
void main()
{
clrscr();
example<int> obj1(3,8,2);
int maxI=obj1.greatest();
cout<<"Maximum="<<maxI;
obj1.inputdata();
maxI=obj1.greatest();
cout<<"Maximum="<<maxI;
example<float> obj2(3.3,1.8,18.2);
float maxF=obj2.greatest();
cout<<"Maximum="<<maxF;
obj2.inputdata();
maxF=obj2.greatest();
cout<<"Maximum="<<maxF;
getch();
}
```

Output

```
Maximum = 8
Enter values 15 17 82
Maximum = 82
Maximum = 18.200001
Enter values 12.8 54.1 1.2
Maximum = 54.099998
```

Class Template and Multitype Parameters

Multitype parameters may also be used with the class in a similar manner as they are used with function template.

Non-Type Template Arguments

We have seen that a template can have multiple parameters of generic type. In addition to the generic type T parameters, we can also use other parameters such as built-in types (int, char and float), strings, function names and constant expressions with the class template. These parameters are called non-type parameters. The value supplied for a non-type template parameter must be a constant expression.

Abstract Data Type Using Template

The abstract data type basically concerns with the data and the operation that can be performed on those data. It is similar to the class definition in C++. The abstract data type requires the operation name along with the precondition and post-condition. Precondition means those conditions that are required to perform the operation and post-condition would be the result after performing the operation. For example the precondition for searching an element in an array is that we should have an array the post-condition is that we search an element and display the result.

Solved Examples

Example 1 *Write a menu-driven program in C to add, subtract, multiply and divide complex numbers.*

Solution

```
#incl<stdio.h>
#incl<conio.h>
struct comp
{
int real;
int imag;
};
typedef struct comp complex;
/* Assigns the name 'complex' to the data
complex addition(complex X, complex Y)
{
complex Z;
clrscr();
Z.real=X.real+Y.real;
Z.imag=X.imag+Y.imag;
return(Z);
}
void subtraction(complex X,complex Y)
{
complex Z;
clrscr();
Z.real=X.real-Y.real;
Z.imag=X.imag-Y.imag;
printf("The difference is:");
print%d(+%di",Z.real, Z.imag);
}
complex multiplication(complex X, complex
{
complex Z;
clrscr();
Z.real=X.real*Y.real-X.imag*Y.imag;
Z.imag=X.real*Y.imag+X.imag*Y.real;
return(Z);
}
void main()
{
complex X,Y,Z,sum, prod, div;
int ch;
/* Variable declaration must be the first
```

```
clrscr();
do
{
printf("\n1. For addition");
printf("\n2. For subtraction");
printf("\n3. For multiplication");
printf("\n4. For division");
printf("\nEnter your option ");
printf("\n(0 to quit");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("Enter value for first complex numbe
scanf("%d%d",&X.real,&X.imag);
printf("Enter value for second complex numb
scanf("%d%d",&Y.real,&Y.imag);
sum=addition(X,Y);
printf("Sum=%d+%di",sum.real,sum.imag);
break;
case 2:
printf("Enter value for first complex numbe
scanf("%d%d",&X.real,&X.imag);
printf("Enter value for second complex numb
scanf("%d%d",&Y.real,&Y.imag);
subtraction(X,Y);
break;
case 3:
printf("Enter value for first complex numbe
scanf("%d%d",&X.real,&X.imag);
printf("Enter value for second complex numb
scanf("%d%d",&Y.real,&Y.imag);
prod=multiplication(X,Y);
printf("The product is %d+%di",prod.real,prod.imag)
break;
case 4:
printf("Enter value for first complex numbe
scanf("%d%d",&X.real,&X.imag);
printf("Enter value for second complex numb
scanf("%d%d",&Y.real,&Y.imag);
div.real=(X.real*Y.imag+X.imag*Y.real)/
Y.imag*Y.imag);
div.imag=(X.imag*Y.real-X.real*Y.imag)/
Y.imag*Y.imag);
```

```
printf("The div%d+i%dn ii"s,:div.real,div.imag)
}
}while(ch!=40);||ch
getch();
}
```

Output

```
1. For addition
2. For subtraction
3. For multiplication
4. For division
Enter your option
(0 to quit) 1
Enter value for first complex number X
1 2
Enter value for second complex number Y
4 5
Sum is: 5 + 7 i
1. For addition
2. For subtraction
3. For multiplication
4. For division
Enter your option
(0 to quit) 0
```

Example 2 *Write a program in C to print the bill details of 10 customers with the following data: meter number, customer name, no. of units consumed, bill date, last date to deposit and city. The bill is to be calculated according to the following condition:*

No. of units	*Charges*
For first 100 units	Rs 0.75 per unit
For the next 200 units	Rs 1.80
For the next 300 units	Rs 2.75

Solution

```
#incl<sdtedio>.h
#incl<cobenio>.h
#incl<sdtedio>.h
#incl<sdterin>g.h
struct bill
{
int meterno;
```

```
char cname[10];
int units;
char bdate[10];
char ldate[10];
char city[10];
double amount;
};
typedef struct bill BILL;
/* Onwards we can use BILL in place of str
the variable*/
void main()
{
BILL X[2];
double amt1, amt2, amt3;
int i;
clrscr();
for(i=0;i<2;i++)
{
printf("\nInformation mno,name,unit,bd,ld,m
printf("Meter no.: \n");
scanf("%d",&X[i].meterno);
printf("Customer name: ");
scanf("%s",&X[i].cname);
printf("No. of units consumed: ");
scanf("%d",&X[i].units);
printf("Date of billing");
scanf("%s",X[i].bdate);
printf("Last date for payment: ");
scanf("%s",X[i].ldate);
printf("City: ");
scanf("%s",X[i].city);
}
clrscr();// clear screen before display the
for(i=0;i<2;i++)
{
printf("\n** Bill Information ** \n");
printf("Meter no.: ");
printf("%d",X[i].meterno);
printf("\nCustomer name: ");
printf("%s",X[i].cname);
printf("\nNo. of units consumed: ");
printf("%d",X[i].units);
if(X[i].units<=100)
amt1=X[i].units*.75;
else if(X[i].units<=300)
```

```
amt2=X[i].units*1.80;
else if(X[i<=.6u0n0i)ts
amt3=X[i].units*2.75;
X[i].amount=amt1+amt2+amt3;
printf("\nTotal payment due is Rs=");
print%f(",X[i].amount);
printf("\nDate of billing");
print%s(",X[i].bdate);
printf("\nLast date for payment: ");
print%s(",X[i].ldate);
printf("\nCity: ");
print%s(",X[i].city);
}
getch();
}
```

Output

```
** Bill Information **
Meter no.: 1
Customer name: S KUMAR
No. of units consumed: 113
Total payment due is Rs = 203.400000
Date of billing 22/02/08
Last date for payment: 28/02/08
City: Bhopal
** Bill Information **
Meter no.: 3
Customer name: Umang S
No. of units consumed: 444
Total payment due is Rs = 1424.400000
Date of billing 12/01/09
Last date for payment: 22/01/09
City: Bhopal
```

Example 3 *Write a menu-driven program for deposition, withdrawal and display balance for the bank account of a particular customer.*

Solution

```
#incl<uidoes tre>am.h
#incl<ccdoeni o>.h
#incl<sdtedi o>.h
class bank
{
int acno;
```

```
char name[10];
float amount;
public:
void inpudata()
{
cout<<"\nEnter details of the customer "
cout<<"Account number ";
cin>>acno;
cout<<"\nCustomer's name ";
gets(name);
cout<<"\nBalance ";
cin>>amount;
}
friend void display(bank X)
{
cout<<"Details of the customer "
cout<<"Account number ";
cout<<X.acno;
cout<<"\nCustomer's name ";
puts(X.name);
cout<<"\nBalance ";
cout<<X.amount;
}
void withdrawl(float amt)
{
if(amount<1000)
cout<<"\nSorry!please maintain minimum balanc
else
amount=amount-amt;
}
void deposit()
{
float amt;
cout<<"\nEnter the amount to deposit ";
cin>>amt;
amount=amount+amt;
}
int retacno()
{
return(acno);
}
};
void main()
{
clrscr();
```

```
bank customers[10];
for(int ii=1;ii<10;ii++)
{
customers[i].inpudata();
}
int op;
int cno,custno;
do
{
cout<<"\nEnter customer no. on which accoun
is to be done";
cin>>cno;
for(i=1;i<10;i++)
{
if(custno==customers[i].retacno())
cno=custno;
}
cout<<"**Options end**";
cout<<"1. To deposit";
cout<<"2. To withdrawl";
cout<<"3. To show the details";
cout<<"Enter your option ";
cin>>op;
switch(op)
{
case 1:
customers[cno].deposit();
break;
case 2:
float value;
cout<<"\nEnter amount for withdrawl ";
cin>>value;
customers[cno].withdrawl(value);
break;
case 3:
display(customers[cno]);
}
}while(op!=0);
getch();
}
```

Output

```
Enter details of the customer
Account Number 1
Customer's name Rajesh N
```

```
Balance 2000
Enter details of the customer
Account Number 102
Customer's name Umang S
Balance 10000
Enter customer no. on which account number
done 1
** Options **
1. To deposit
2. To withdraw
3. To show the details
Enter your option 1
Enter the amount to deposit 1000
Enter customer no. on which account number
done 1
** Options **
1. To deposit
2. To withdraw
3. To show the details
Enter your option 3
Details of the customer
Account Number 1
Customer's name Rajesh N
Balance 3000
Enter customer no. on which account number
done 1
** Options **
1. To deposit
2. To withdraw
3. To show the details
Enter your option 0
```

Example 4 *Write a program in C++ to find the greatest among the three given numbers of different types using the normal function and the template function.*

Solution

```
#incl<uidoestrea>m.h
#incl<ucdoeni>.h
templ<actlea s> T
T greatest(Tx, Ty, Tz)
{
cou<<"\n Template functio<<einsdlc;alled: "
if (>x
```

```
if (x>z)
return(x);
else
return(z);
else
if (y>z)
return(y);
else
return(z);
}
float greatest(float x, float y, float z)
{
cout<<"Normal function definition is called
if (x>y)
if (x>z)
return(x);
else
return(z);
else
if (y>z)
return(y);
else
return(z);
}
void main()
{
clrscr();
int ch;
char choice;
do
{
cout<<"Find the greatest numbers"<<endl;
cout<<"1. Among integer numbers:"<<endl;
cout<<"2. Among float numbers:"<<endl;
cout<<"3. Among double numbers:"<<endl;
cout<<"Enter your choice: ";
cin>>ch;
switch(ch)
{
case 1:
cout<<"Enter integer numbers: "
int a,b,c,maxI;
cin>>a>>b>>c;
maxI=greatest(a,b,c);
```

```
cout<<"The greatest number is:"<<maxF;
break;
case 2:
cout<<"Enter float numbers: ";
float x,y,z,maxF;
cin>>x>>y>>z;
maxF=greatest(x,y,z);// normal function get
cout<<"The greatest number is:"<<maxF;
break;
case 3:
cout<<"Enter double numbers: ";
double m,n,q,maxD;
cin>>m>>n>>q;
maxD=greatest(m,n,q);
cout<<"The greatest number is:"<<maxD;
}
cout<<"\nWant to continue (y/n): ";
cin>>choice;
}while(choice=='y'||choice=='Y');
getch();
}
```

Output

```
Find the greatest numbers
1. Among integer numbers:
2. Among float numbers:
3. Among double numbers:
Enter your choice: 2
Enter float numbers: 2.3 4.2 8.3
Normal function definition is called
Want to continue (y/n): n
```

Example 5 *Write a program to find the greatest element in a given array of a given size by using class template with non-type parameters.*

Solution

```
#include<iostream.h>
#include<conio.h>
template<class T,int size
class sample
{
T array[size];
```

```
public:
void inputdata()
{
cout<<"\nEnter values ";
for(int i=0;i<5;i++)
cin>>array[i];
}
void display()
{
cout<<"\nData members of an array are ";
for(int i=0;i<5;i++)
cout<<array[i]<<" ";
}
T greatest(T max)
{
for(int i=0;i<5;i++)
{
if(array[i]>max)
max=array[i];
}
return(max);
}
};
void main()
{
clrscr();
sample<int>obj1;
cout<<"Integer values";
obj1.inputdata();
obj1.display();
int maxI=obj1.greatest(0);
cout<<"\n The greatest number is: "<<maxI;
sample<float>obj2;
cout<<"\nFloat values";
obj2.inputdata();
obj2.display();
float maxF=obj2.greatest(0.0);
cout<<"\n The greatest number is: "<<maxF;
getch();
```

Output

```
Integer values
Enter values 1 2 7 2 9
Data members of an array are 1 2 7 2 9
The greatest number is: 9
```

```
Float values
Enter values 1.2 3.6 8.9 2.1
Data members of an array are 1.2   3.6   8.9
The greatest number is: 8.9
```

Example 6 *Write a program to use member function as template function for the addition of two numbers of different types.*

Solution

```
#include<iostream.h>
#include<conio.h>
template<class T>
class example
{
T a;
T b;
public:
example(Tx, Ty)
{
a=x;
b=y;
}
void inputdata()
{
cout<<"\n Enter values for a and b ";
cin>>a>>b;
}
T sum(T, T);
void display()
{
cout<<"\n First value: "<<a;
cout<<"\n Second value: "<<b;
}
};
template<class T>
T example<T>::sum(Tx, Ty)
{
return(x+y);
}
void main()
{
clrscr();
example<int>*obj1(2,8);
```

```
obj1.display();
int totI=obj1.sum(2,8);
cout<<"\nThe total I";
obj1.inputdata();
totI=obj1.sum();
cout<<"\nThe total I";
example of obj2(2.7,8.6);
obj2.display();
float totF=obj2.sum(2.7,8.6);
cout<<"\nThe total F";
getch();
}
```

Output

```
First value: 2
Second value: 8
The total: 10
Enter values for a and b 1 3
The total: 4
First value: 2.7
Second value: 8.6
The total: 11.3
```

Example 7 *Write a program in C++ to demonstrate the use of union.*

Solution

```
#include<iostream.h
#include<conio.h>
#include<stdio.h>
void main()
{
clrscr();
union example
{
int rollno;
float percentage;
char name[10];
};
example X;
cout<<"Memory occupied by X is:";
cout<<sizeof(X);
cout<<"\n**Enter information** ";
cout<<"\n\nEnter roll no.:";
```

```
cin>>X.rollno;
cout<<"\nEnter name:";
gets(X.name);
cout<<"\nEnter percentage marks:";
cin>>X.percentage;
cout<<"\nThe result after executuion **\n";
cout<<"\nThe roll no.:";
cout<<X.rollno;
cout<<"\n\nName:";
puts(X.name);
cout<<"\n\nThe percentage marks:";
cout<<X.percentage;
cout<<"\n\nSize of X is still "<<sizeof(X);
getch();
}
```

Output

```
Memory occupied X is: 10
**Enter information**
Enter roll no.: 101
Enter name: Himanshu
Enter percentage marks: 78.29
The result after execution **
The roll no.: 27525
Name: Himanshu
The percentage marks: 78.290001
Size of X is still 10
```

Summary

1. A structure is a collection of dissimilar data items. Members of structures are accessed using dot operator.
2. The data declared within the structure is called member and the structure name is called the structure tag which is a new data type. Memory is not reserved for the structure tags but the memory is reserved when the variable of structure tag type is defined.
3. Structure variables can be initialized. There must be one-to-one correspondence between the members and the initializing values. If some of the structure members are not initialized, the C++ compilers will automatically initialize them to zero.
4. One structure variable can be assigned to another variable of the same type but the other operations, such as comparison or not equal to, are not defined for similar structures.
5. A structure can be a member of another structure. Such structures are called nested structures.
6. Union is only memory management feature which provides a way to manipulate different kinds of data in a single area of storage.

7. An abstract data type defines the attributes and the operations of all objects belonging to a particular class.
8. Member functions of a class may also be defined outside the class by using scope resolution operator.
9. The `private` and `protected` members can only be accessed by the member functions of the same class while `public` members can be accessed outside the class by using object of this class type.
10. When a class contains object of some other class as data members, such class is called container class.
11. All members of a structure are `public` by default and all members of a class are `private` by default.
12. The memory for the data members of an object is allocated when the object is declared and the memory for the member function is allocated at the time of class declaration.
13. Constructors are used to initialize objects and destructors are used to destroy objects.
14. Constructors can be classified into five categories: default constructor, parameterized constructors, default argument constructors, copy constructor and dynamic constructors.
15. Template in C++ supports the concept of generic programming.
16. Template allows us to generate a family of classes or a family of functions to handle different data types. Therefore it eliminates the code duplication for different types.
17. The process of creating a template class is known as instantiation.
18. Member function of a class template is defined as function template outside the class.
19. We may also use non-type parameters such as basic or derived data types as arguments templates.

Key Terms

Structure	`Public`	Class template
Array and structure	`Protected`	Function template
Nested structure	Constructor	Generic programming
Union	Default argument	Member function template
Dot operator	Parameterized constructor	Template class
Class	Copy constructor	
`Private`	Destructor	

Multiple-Choice Questions

1. For the following code how much memory is reserved?

```
union example
{
float a;
char b;
} X;
```

   a. 2 bytes
   b. 4 bytes
   c. 1 byte
   d. 8 bytes

2. `struct` is
   a. an identifier.
   b. constant.
   c. keyword.
   d. user-defined variable.

3. Structure members are accessed
   a. by using dot operator.
   b. by using index.
   c. by using subscript.
   d. directly.

4. Structure is a collection of
   a. similar data items.
   b. dissimilar data items.
   c. either similar or dissimilar data items.
   d. special characters.

5. For the following code, how much memory is allocated?
```
struct example
 {
 int a;
 float b;
 char c;
 };
```
   a. 4 bytes
   b. 7 bytes
   c. No allocation
   d. 8 bytes

6. While accessing the structure members, left side of dot operator must be a
   a. structure variable.
   b. structure pointer.
   c. keyword struct.
   d. index value.

7. Class is a
   a. collection of similar elements.
   b. collection of dissimilar elements.
   c. combination of data members and member functions.
   d. none of the above.

8. By default all the members of a class are
   a. private
   b. public
   c. protected
   d. all of the above.

9. Which of the following cannot be accessed by the outside world?
   a. Private members of a class
   b. Public members of a class
   c. Protected members of a class
   d. Both a and c

10. Which operator is used to access the members of a class using object?
    a. Arrow operator
    b. Dot operator
    c. Bitwise operator
    d. All of the above

11. Which operator is used to define the outline member function?
    a. Dot operator
    b. Scope resolution operator
    c. Arrow operator
    d. Bitwise operator

12. A class is called a container class if
    a. it contains all the member functions in public mode.
    b. it contains the objects of another class as data members.
    c. it contains another class inside.
    d. it contains both the friend and the inline functions.

13. The only difference between a class and structure is that
    a. members are private in class and public in structure by default.
    b. class consists of data and member functions, and structure consists of data only.
    c. we can specify different access mode in class but in structure it is not possible.
    d. all the above.

14. The class without a tag name is called
    a. nested class.
    b. container class.
    c. anonymous class.
    d. friend class.

15. A constructor is called automatically when an object is
    a. created.
    b. initialized.
    c. destroyed.
    d. a member of another class.

16. Constructor is
    a. a function which is automatically called when object is created.
    b. called explicitly when object is created.
    c. a function which destroys the object.
    d. a special operator like `new` or `delete`.

17. Constructor can return
    a. `void` data.
    b. any data of user-defined type.
    c. any data of built-in type.
    d. no value (not even `void`).

18. The constructor which does not take any parameter is called
    a. parameterized constructor.
    b. default constructor.
    c. default argument constructor.
    d. dynamic constructor.

19. Constructor generally declared in
    a. `public` mode.
    b. `private` mode.
    c. `protected` mode.
    d. none of the above.

20. Destructor is
    a. a function which is automatically called when object is created.
    b. called explicitly when object is created.
    c. a function which destroys the object.
    d. a special operator like `new` or `delete`.

21. Template is used for
    a. virtual base class.
    b. friend class.
    c. container class.
    d. nested class.

22. Generacity is defined as the method in which
    a. special property of a class is defined.
    b. the data structures and functions are defined without knowing the details of the data types on which they operate.
    c. pure virtual function is used in the class.
    d. all of the above.

23. Template supports
    a. generacity.
    b. polymorphism.
    c. inheritance.
    d. all of the above.

24. Which keyword is used to define a function template in C++?
    a. Template
    b. Template function
    c. Friend
    d. Virtual

25. Templates are useful because
    a. the same logic need not be repeatedly written for different data types.
    b. they provide a way of defining the behavior of the class without actually knowing the data types.
    c. they provide polymorphic behavior using generacity.
    d. all of the above.

Review Questions

1. How is an array different from a structure?
2. Can you assign a structure variable to another? If yes, when?
3. How is a structure initialized?
4. Can an array be a member of a structure?
5. What are the rules for declaration of the constructor function?
6. What are the differences between a class and an object?
7. What are the differences between a structure and a class?
8. What is a parameterized constructor? How it is different from the default argument constructor?

9. What is the difference between default constructor and default argument constructor?
10. What should be the sequence of default value assignments for the default argument constructor?
11. What are the different ways to define the member functions of a class? What is a template?
12. What are the advantages and disadvantages of templates?
13. What is a function template?
14. How is the function template defined in a program in C++? What are the rules to follow in order to define a function template?
15. What is a class template? What are the rules to follow in order to define a class template?
16. What is the difference between an overloaded function and a function template?
17. What is the difference between a class template and a template class?
18. What is generic programming and how it is implemented in C++?
19. Can we have a more than one constructor in a class? If yes, explain the need for such a situation.
20. How is an array different from a structure?
21. Can an array be a member of a structure?
22. Define structure in C. [Anna – 2006–2007]
23. Differentiate structure and union. [Anna – 2006–2007]
24. What is the importance of a constructor and a destructor?

Programming Assignments

1. Write a program in C++ that prints the Fibonacci series using parameterized constructor and a destructor member function.
2. Write a program using member function of a class as template function to calculate the sum, the smallest element in the list and to check whether the given number is present in the list. If present, count the frequency (how many times is it present).
3. Write a menu-driven program for deposition, withdrawal and display balance for the bank account of a particular customer.

Answers

Multiple-Choice Questions

1. (b)
2. (c)
3. (a)
4. (c)
5. (b)
6. (a)
7. (c)
8. (a)
9. (a)
10. (b)
11. (b)
12. (b)
13. (a)
14. (c)
15. (b)
16. (a)
17. (d)
18. (b)
19. (a)
20. (c)
21. (c)
22. (b)
23. (d)
24. (b)
25. (a)

4 Pointers and Dynamic Memory Management

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Pointer declaration.
- Pointer arithmetic.
- Pointer operators.
- Pointer to function.
- Pointer to arrays.
- Pointer to pointer.
- Dynamic memory allocation.

As we already know, variables are used to store values that are kept in the memory. We also know that every byte in the memory has an address; therefore, the values stored in the memory must have some address. A variable which holds the address of the value is called *pointer*. In other words, we can say that the pointer is a variable that can hold the address of other variables or address of value. Therefore, using pointers we can access the memory locations and the data stored in those memory locations by indirect reference. Hence, pointers not only provide the access to the memory location but also provide the method for dynamic memory allocation. They are the most useful and strongest features of C and C++. In fact, a pointer can point to any type of data item or function.

4.1 Declaration of a Pointer

Since pointer is also a variable, therefore, the declaration of a pointer variable is also similar to that of a normal variable except for the addition of the unary * character. This symbol informs the compiler that the variable is a pointer variable. The * (asterisk) means "pointer to". The syntax for pointer declaration is as follows.

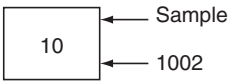
Syntax

```
Data type *pointer variable;
```

Here, `pointer variable` is the name of the pointer and the data type may be any C/C++ data type such as `int`, `float`, `char`, etc. The asterisk symbol may be placed anywhere between the data type and the pointer variable. Therefore, the pointer declaration is based on the data type of the variable it points to.

Example 1

```
int *ptr;
int sample=10;
ptr=&sample;
```



In Example 1, `pt` is a pointer to the integer, so it can hold the address of the integer type of data; `sample` is an integer type of variable which contains the value 10. Therefore, `pt` can hold the address of `sample` (let it be `1000`). With pointers, two types of operators are used:

1. address of operator (&);
2. value at operator (*).

Address of Operator

"Address of operator" is a unary operator that returns the address of the variable following it. Since it is a unary operator; therefore, it requires only one operand and so it is represented by using ampersand sign (&) followed by the variable name. For example

```
pt=&sample;
```

Here `pt` will hold the address of sample, that is, `1000`.

Value at Operator/ Indirection Operator

"Indirection operator" is a unary operator that returns the contents stored at the memory location (address). It is also called "value at operator" or "dereference operator". This operator is represented by a combination of * with a variable. For example, if `pt` holds the address of `sample`

```
pt=&sample;
```

If we now write `cout<<*pt;` in C++ or `printf("%u", *pt);` in C, then it will display the content at `pt`, that is 10. The `(*pt)` is read as "the value at the address contained in `pt`" or the "value at `pt`". Here, we must remember the following points when dealing with pointers:

1. The pointer variable must be initialized. If the address of base data type variable is not assigned to the pointer variable, it may be initialized with null.
2. Both the pointer operators & and * have a higher precedence over all other arithmetic operators except the unary minus, with which they have equal precedence.
3. The variable whose address is assigned to a pointer variable is declared first.
4. Only two operations are possible with pointer variables:
   - addition of value (you may not add two pointers);
   - subtraction of value (you may not subtract two pointers).
5. The addition of value to the pointer variable is performed according to the size of the base data type. For example

   ```
   pt=pt+3
   ```

   Since `pt` is an integer pointer and integer number usually takes 2 bytes of storage, so `pt=pt+3 * (size of base data type)`

   ```
   pt=pt+3 * 2
   pt=pt+6
   pt=1000+6
   pt=1008
   ```

6. Whenever the pointer is incremented by one, it points to the memory location of the next element of its base type. Similarly, when we decrement the pointer variable by 1 (subtracting by 1), it will point to the location of the previous element of its base type. The pointer arithmetic is compiled in Table 1.

Table 1 Arithmetic operations with pointer variables

Pointer arithmetic	*Equivalent to*	*Explanation*
`ptr++`	`ptr=ptr+sizeof(data type)`	We are using post-increment operator which means "use then increment". So this statement will use original value of `ptr` and then `ptr` is incremented after statement execution.
`++ptr`	`ptr=ptr+sizeof(data type)`	We are using pre-increment operator which means "increment then use". So this statement will increment the original value of `ptr` and will be used in the statement.
`ptr--`	`ptr=ptr-sizeof(data type)`	We are using post-decrement operator which means "use then decrement". So this statement will use original value of `ptr` and then `ptr` is decremented after statement execution.
`--ptr`	`ptr=ptr-sizeof(data type)`	We are using pre-decrement operator which means "decrement then use". So this statement will decrement the original value of `ptr` and will use it in the statement.
`(*ptr++)`	`*ptr then *ptr=*ptr+1`	Although ++ has higher precedence then *, this precedence is altered by using parenthesis. So this statement will cause first the retrieval of the content at `ptr` and then the content will be incremented (post) by one after statement execution.
`*ptr++`	`*ptr then ptr=ptr+1`	++ operator has higher precedence than * so this statement will retrieve the content of the location pointed to by pointer and then increment (post) `ptr` after statement execution. It is not similar to `(*ptr++)`.
`*++ptr`	`ptr=ptr+1 then *ptr`	++ operator has higher precedence than * so this statement `*(++ptr)` will increment pointer and then retrieve the content of the new location pointed to by `ptr`.
`++*ptr`	`*ptr then *ptr=*ptr+1`	This statement will cause first the retrieval of the content at `ptr` and then the content will be incremented (pre) by one before statement execution.

Program 1 A program in C++ to demonstrate the pointer arithmetic operations.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int x=10;
int *ptr;
```

```
pt=&x;                                    //pointer init
cout<<"Applying the pointer arithmetic"<<endl;
cout<<"pt="<<pt<<endl;      //display ptr i.e. &x
cout<<"++pt="<<++pt<<endl//increment ptr then dis
pt=&x;
pt++;                    //use ptr then after exec
increment ptr
cout<<"pt++="<<pt<<endl;  //display incremented p
pt=&x;
cout<<"*pt="<<*pt<<endl;  //value at ptr i.e. 10
pt=&x;
cout<<"pt++="<<pt++<<endl;//use ptr then incremen
cout<<"pt="<<pt<<endl;     //it shows incremented
cout<<"++*pt="<<++*pt<<endl;//increment the conten
pt=&x;
cout<<"*++pt="<<*++pt<<endl;  //increment ptr then
content
pt=&x;                                    //at increm
cout<<"*pt++="<<*pt++<<endl;//display the content
increment ptr
cout<<"pt++="<<pt<<endl;//display the address in
previous statement
pt=&x;
cout<<"(*pt++)="<<(*pt++)<<endl;//display content o
content is incremented
cout<<"*ptr="<<*ptr<<endl;//displays the incremen
cout<<"pt++ ="<<pt<<endl;//ptr will remain same
getch();
}
```

Output

```
Applying the pointer arithmetic
pt=0x8fbffff4
++pt=0x8fbffff6
pt++=0x8fbffff6
*pt=10
pt++=0x8fbffff4
pt++=0x8fbffff6
++*pt=11
*++pt=1
*pt++=10
pt++=0x8fbffff6
(*pt++)=10
*ptr:11
pt++=0x8fbffff4
```

Program 2 A program in C to demonstrate the pointer arithmetic operations.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x=10;
int *ptr;
clrscr();
ptr=&x;                                          //pointer i
printf("Applying the pointer arithmetic\n");
printf("ptr=%u\n",ptr);                  //display ptr i.e.
printf("++ptr=%u",++ptr);    //increment ptr then d
ptr=&x;
ptr++;      //use ptr then after execution incre
printf("ptr=%u",ptr);    //display incremented
ptr=&x;
printf("*ptr=%u",*ptr);          //value at ptr i.e.
ptr=&x;
printf("ptr=%u",ptr++);          //use ptr then incre
printf("ptr=%u",ptr);  //it shows incremented
printf("++*ptr=%u",++*ptr);  //increment the conten
ptr=&x;
printf("++ptr=%u",++*ptr);//increment ptr then d
content
ptr=&x;                                          //at i
printf("\n*ptr=%u",*ptr++);      //display the content
increment ptr
printf("ptr=%u",ptr);  //display the address i
previous statement
ptr=&x;
printf("\n++(*ptr)=%u",++(*ptr));    //display content of
content is incremented
printf("*ptr=%u",*ptr);                  //displays
content
printf("ptr=%u",ptr);                    //ptr will
getch();
}
```

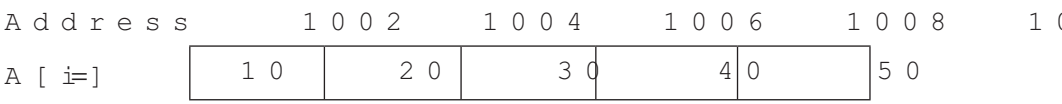
Output

```
Applying the pointer arithmetic
ptr=65524
++ptr=65526
ptr++=65526
```

```
*ptr=10
ptr=6552 4
ptr=6552 6
++*ptr=11
*++ptr=11
*ptr++=10
ptr=6552 6
(*ptr++)=10
*ptr=11
ptr=6552 4
```

4.2 Pointers and One-Dimensional Arrays

Arrays and pointers are closely related in C++. In fact the versatility of pointers is established due to their ability to handle arrays. In C++ the array name acts as a pointer because the array name always refers to the address of first element of the array. So we can say that array name is a pointer which holds the address of its first element. For example




```
int A[5];
int *ptr;
```

ptr=&A[0]; true because ptr is a pointer to integer so ptr will hold 1002
ptr=A; true because array name refers to the address of the first element, that is, ptr=1002
So, ptr=&A[0]; or ptr=A; are two equivalent statements. We generalize the statements as

```
A[i]=*(A+i)=*(&(A)[0]+(i))
```

Now, we can say that pointer is one of the efficient tools to access elements of an array. Summarizing, the relation between array and pointer as given below where array subscripting is defined in terms of pointer arithmetic.

```
ptr=1002                    =content of pointer start the address
                          contained in ptr=A (a=1010002)
ptr+1=1002+1*2=1002       +1 is content at address 1002+2=1004
ptr+2=1002+2*2=1004       +2 is content at address 1002+4=1006
ptr+3=1002+2*3=1006       +3 is content at address 1002+6=1008
ptr+4=1002+2*4=1008       +4 is content at address 1002+8=1010
```

Program 3 A program in C++ to use pointers with one-dimensional array.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int A[]={2, 4, 6, 8, 1};
cout<<"Base address:"<<A<<endl;
cout<<"A+1:"<<A+1<<endl;
cout<<"A+2:"<<A+2<<endl;
cout<<"A+3:"<<A+3<<endl;
cout<<"A+4:"<<A+4<<endl;
cout<<endl;
cout<<"Value at A:"<<*A<<endl;
cout<<"Value at A+1:"<<*(A+1)<<endl;
cout<<"Value at A+2:"<<*(A+2)<<endl;
cout<<"Value at A+3:"<<*(A+3)<<endl;
cout<<"Value at A+4:"<<*(A+4)<<endl;
cout<<"Value at A[4]:"<<4[A]<<endl;
cout<<"Value at 4[A]:"<<4[A]<<endl;
cout<<endl;
cout<<"(Value at A+1:"<<*A+1<<endl;
cout<<"(Value at A+2:"<<*A+2<<endl;
cout<<"(Value at A+3:"<<*A+3<<endl;
cout<<"(Value at A+4:"<<*A+4<<endl;
getch();
}
```

Output

```
Base address: 0x8fc2ffec
A+1: 0x8fc2ffee
A+2: 0x8fc2fff0
A+3: 0x8fc2fff2
A+4: 0x8fc2fff4
Value at A: 2
Value at A+1: A4
Value at A+2: A6
Value at A+3: A8
Value at A+4: A1
Value at A[4]: 1
Value at 4[A]: 1
(Value at A+1: A3)
(Value at A+2: A4)
(Value at A+3: A5)
(Value at A+4: A6)
```

4.3 Pointers and Two-Dimensional Arrays

Array name always refers to the address of the first element; in one-dimensional array, the address of 0th element. In two-dimensional, the 0th element is the element present in the 0th row and 0th column. We know that two-dimensional array is a collection of one-dimensional array. For example

```
int A[2][3] = { {1, 2, 4}, {5, 8, 3} };
```

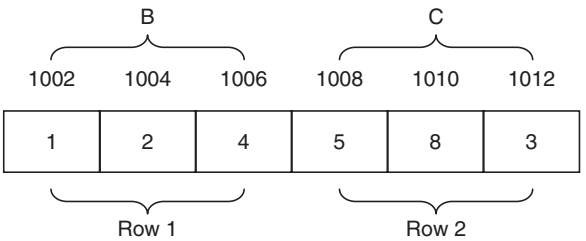
We can say that A is a collection of two elements where these two elements themselves are arrays of three elements. Therefore, array A is a collection of two one-dimensional arrays. Let us take

```
B[3] = {1, 2, 4}
C[3] = {5, 8, 3}
```

Then we can say that A is a collection of B and C. So

```
A[2] = {B, C}
```

This array can be represented in row major order in the memory as



Now

```
cout<<A;              will display 1002.
cout<<A+1;            will display the address of the next element of array A, that is, address of
                      A[1], 1008
cout<<*(A+1);         will display content at 1008, that is a one-dimensional array C of
                      three elements 5, 8, and 3.
cout<<*(A+1)+1;       will display the address of the next element of array C, that is, 1010.
cout<<*(*(A+1)+1);    will display the content at 1010, that is 8
cout<<A[1][1];        will display the content of 1st row and 1st column, that is 8
```

We can now establish the formula to access the element of two-dimensional array using pointers as

```
A[i][j] = *(*(A+i)+j)
```

If `int *ptr;`
`ptr = A;` or `ptr = &A[0][0];`
Then to access the contents A[i][j] we can use

```
A[i][j] = *(*(ptr+i)+j)
```

Following the same strategy we can say that the three-dimensional array A is a collection of two-dimensional arrays. So for three-dimensional arrays we can also give the formula to access the contents as

```
A [ i ] [ j ]=*[(k*]( *+i( )Aj )+k )
```

Similarly we can form the formula for multidimensional arrays also.

Program 4 A program in C to use pointers with two-dimensional array.

```
# i n c l <s d t e d i o > . h
# i n c l <c c o n i o > . h
v o i d   m a i n ( )
{
i n t   A [ 3 = ] { 2 , 4 , 4 , 8 , 1 , 5 , 7 , 8 , 9 , 1 1 , 1 3 , 6 , 3 } ;
c l r s c r ( ) ;
p r i n t f ( " B a s e   a %d d r e s s :
p r i n t f ( + 1 : %u A " , + A ) ;
p r i n t f ( + 2 : %u A " , + 2 A ) ;
p r i n t f ( " \ n %u ( " A , ) * : A ) ;
p r i n t f ( " + 1 n ) * : % u A " , * + 1 A ) ;
p r i n t f ( " + 2 n ) * : % u A " , * + 2 A ) ;
p r i n t f ( " \ + n : * % u A " ) , * ( + A ) ) ;
p r i n t f ( " + 1 n ) + 1 ( % u " , * + 1 A + 1 ) ;
p r i n t f ( " + 2 n ) + 1 ( % u " , * + 2 A + 1 ) ;
p r i n t f ( " \ n + 1 ( * % u A " ) , * ( + A ) ) ;
p r i n t f ( " \ n 1 * ) + ( 1 * ) ( : % u " , * ( + 1 ( ) + A ) ) ;
p r i n t f ( " \ n 2 * ) + ( 1 * ) ( : % u " , * ( + 2 ( ) + A ) ) ;
p r i n t f ( " \ n V a l u e   a % u " A , [ A 0 [ ] 0 [ ] 1 [ ] 1 : ] ) ;
p r i n t f ( " \ n V a l u e   a % u " A , [ A 1 [ ] 1 [ ] 1 [ ] 1 : ] ) ;
p r i n t f ( " V a l u e   a t % u A " [ , 2 A ] [ [ 2 1 ] ] [ : 1 ] ) ;
g e t c h ( ) ;
}
```

Output

```
B a s e   a d d r e s s :   2
A + 1 :   6 5 5 1 0
A + 2 :   6 5 5 1 8
* ( A ) :   6 5 5 0 2
* ( + A 1 ) :   6 5 5 1 0
* ( + A 2 ) :   6 5 5 1 8
* ( A + 1 ) :   6 5 5 0 4
* ( + A 1 ) + 1 :   6 5 5 1 2
* ( + A 2 ) + 1 :   6 5 5 2 0
* ( * ( + A 1 ) ) :   4
* ( * ( + 1 A ) + 1 ) :   7
* ( * ( + 2 A ) + 1 ) :   1 3
V a l u e   a t   A [ 0 ] [ 1 ] :   4
V a l u e   a t   A [ 1 ] [ 1 ] :   7
V a l u e   a t   A [ 2 ] [ 1 ] :   1 3
```

4.4 Arrays of Pointers

Just like the array of built-in data type we can also declare the array of pointers. The collection of pointers is called array of pointers. Each element of such array is a pointer and hence it can hold the address of specified data type. Here we must remember that the pointers must be of same type. Array of pointers may be declared as follows:

Syntax

```
Data type *array name[size];
```

For example, `int *ptr[3];` means each element of `ptr` array can hold the address of `int` variable. So if we have

```
int A[] = {1, 2, 3};
```

Then we can assign the addresses of each element of A to the different elements of `ptr`.

```
ptr[0] = &A[0];
ptr[1] = &A[1];
ptr[2] = &A[2];
```

We can assign the address of any integer to any element of `ptr` array.

```
int a, b, c, d, e;
ptr[0] = &a;
ptr[1] = &b;
ptr[2] = &c;
```

The declaration for array of pointers to characters may be given as

```
char *ptr[row][col];
```

This is very useful when we deal with array of strings.

4.5 Pointers and Strings

Pointers are very useful for string manipulations because a pointer makes the manipulation of the strings much easier. Pointers are used to perform various string operations such as string copy, string length, string concatenate, string compare, etc. In fact, we can easily exchange the position of strings in the array using pointers. There are two ways to initialize the character string.

1. `char A[] = "Sample string";`

   Here the array contains 14 characters including null character (`\0`). The array name refers to the base address, that is, the address of the first element. Note that the array name is a constant pointer, so we cannot the change the value of A.

2. `char *ptr = "Sample string";`

   Here, character pointer `ptr` is declared which points to the first character of the string. Since `ptr` is a pointer variable, we may change the value in it by using operators likes `ptr++` or `ptr--`. Table 2 compares pointers and arrays.

Table 2 Comparison of pointers and arrays

Pointer	*Array*
Pointer is a variable that holds the address.	Array is a collection of similar items.
Since pointer is a variable so its value may be changed using ++ or –– operators.	Array name is constant so its value cannot be changed, that is array name++ or array name–– is not allowed.
The pointer variables generally appear on the left side of an assignment operator.	The array name cannot appear on the left side of an assignment operator because it is a constant.
It is not necessary that pointer should be an array.	Array name refers to the base address so it is a pointer to its first element.
Pointer does not allocate continuous memory blocks.	Array allocates continuous memory blocks.

Program 5 A program in C++ to swap two strings using pointers.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
char *S1="Pankaj Pandey";
char *S2="M.S. Sisodia";
cout<<"First str="<<S1<<endl;
cout<<"Second str="<<S2<<endl;
cout<<"Swap the strings";
char *temp;
temp=S1;
S1=S2;
S2=temp;
cout<<"First str="<<S1<<endl;
cout<<"Second str="<<S2<<endl;
getch();
}
```

Output

```
First str=Pankaj Pandey
Second str=M.S. Sisodia
Swap the strings
First str=M.S. Sisodia
Second str=Pankaj Pandey
```

4.6 Pointers, Structures and Class

We have seen that a pointer can point to `int`, `char`, `float` and `string`. In this section, we will learn how a pointer is used to hold the address of a structure variable. We already know the declaration for structure variable. Let us take the structure named `student`

```
struct student
{
int rollno;
char name[20];
int age;
} A;
```

Here A is a variable of structure "student" type. Now let us take a pointer to structure

```
student *ptr;
```

This declaration is similar to the pointer declaration for other data type. The ptr can hold the address of A:

```
pt=& A;
```

The members of student A can be accessed by pointer to structure pt using arrow operator. For example, if we want to access the age of student A then we can access it in two ways.

1. pt->age or
2. A.age

Here we must note that when arrow operator is used, on the left side of the arrow operator there must be a pointer to structure and when dot operator is used, on the left side of the dot operator there must be a structure variable. The structure pointers are very useful to create dynamic data structures such as linked list, stacks, queues, trees, etc. The arrow operator is made up of a minus sign and a greater than sign. We should also note that the member operator (.) has a higher precedence than the indirection operator (*).

A pointer can be used to point to the object in C++ a similar way as it is used to point to the structure variable. There are two ways to access the members of a class using pointers:

(a) using the arrow operator in the similar way as it is used for the structure variables.

```
pointer variable->member name
```

(b) The other way is

```
(*pointer variable).member name
```

Here the parentheses are essential since the dot has a higher precedence over the indirection operator (*).

Program 6 A program in C++ to read and display the employee information using pointer to object.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class date
{
```

```
int mm,dd,yy;
public:
void inputdate()
{
cin>>dd>>mm>>yy;
}
void displaydate()
{
cout<<dd<<"/"<<mm<<"/"<<yy;
}
};
class emp
{
private:
char name[20];
char desig[20];
int age;
date dob;//object of date class is a priv
class
public:
date doj;//object of date class is a pub
class
void inputdata()
{
cout<<"\nEnter employee details ";
cout<<"\nName:";
gets(name);
cout<<"\nDesignation:";
gets(desig);
cout<<"\nEnter age:";
cin>>age;
cout<<"\nEnter date of birth:";
dob.inputdate();
}
void display()
{
cout<<"\nDisplay employee details ";
cout<<"\nName:";
puts(name);
cout<<"\nDesignation:";
puts(desig);
cout<<"\nAge:";
cout<<age;
cout<<"\nThe date of birth: ";
dob.displaydate();
```

```
cout<<"\nDate of joining: ";
doj.displaydate();
}
};
void main()
{
clrscr();
emp X, *ptr1;
ptr=&X;
ptr->inputdata();          //first method of acc
cout<<"\nEnter date of joining:";  // memb
operator
ptr->doj.inputdate();
(*ptr1).display(); //second method of acc
getch();
}
```

Output

```
Enter employee details
Name: Prerna
Designation: Manager
Enter age: 26
Enter date of birth: 11 11 1982
Enter date of joining: 11 11 2006
Display employee details
Name: Prerna
Designation: Manager
Age: 26
The date of birth: 11:11:1982
Date of joining: 11:11:2006
```

"*this*" Pointer

this is a special type of pointer in C++ (not available with C) which holds the address of the objects that invoke the member function of the class. For example, if we write `X.inputdata()` then the function call statement will set the address of the object `X` to *this* pointer. Whenever a member function is called, an address of the object that calls it is passed to the function and this address is collected in the *this* pointer. Hence *this* pointer acts as an implicit argument to all the member functions of the class and it is automatically passed to the member function when member function of a class is called. The *this* pointer can only be used for non-static member functions of a class. One important application of the pointer *this* is to return the objects it points to. Another application is in the overloading of a binary operator where only one argument is explicitly passed and another argument is automatically passed using *this* pointer.

Program 7 A program in C++ to show the use of `this` pointer.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class date
{
int mm,dd,yy;
public:
date* inputdate()
{
cout<<"\nEnter the date in DD/MM/YY format:";
cin>>dd>>mm>>yy;
return this;//returns the address of an ob
function
}
void displaydate()
{
cout<<"\nThe entered date is: ";
cout<<this->dd<<":"<<this->mm<<":"<<this->yy;
}
void displaydate1()
{
cout<<"\nThe entered date is: ";
cout<<(*this).dd<<":"<<(*this).mm<<":"<<(*this).yy;
}
};
void main()
{
clrscr();
date X,Y;
date *ptr;
X.inputdate();
X.displaydate();
Y.inputdate();
Y.displaydate1();
ptr=X.inputdate();
cout<<"\nThe address "<<ptr<<" is:"
ptr=Y.inputdate();
cout<<"\nThe address "<<ptr<<" is:"
getch();
}
```

Output

```
Enter the date in DD/MM/YY format: 1 2 20
The entered date is: 1:2:2003
```

```
Enter the date in DD/MM/YY format: 2 4 20
The entered date is: 2:4:2006
Enter the date in DD/MM/YY format: 2 4 20
The address of X is: 0x8fc7fff0
Enter the date in DD/MM/YY format: 1 4 20
The address of Y is: 0x8fc7ffea
```

4.7 Pointers and Functions

Pointer as Function Argument

Just like the normal arguments, functions can also have pointers as arguments. The general syntax for functions with pointers as arguments is as follows.

Syntax

```
return type function name(datatype *va
*variable2);
```

When we call a function, we have to pass the address of the variable which can be stored in the corresponding pointer variables.

Function Returns Pointer

Since functions can return any type of data, they can also return a pointer variable. The general syntax for pointer to function is as follows.

Syntax

```
return type *function name(paramet
```

Here, `return type` is the data type of the value which is returned by the function and `function name` follows the usual rules of naming the identifiers. The list of parameters may consist of any types of parameters, for example, `int`, `float`, `array`, `structure`, `reference`, `class`, etc.

Pointer to Function

We have pointer to `int`, `float`, `char`, etc. Can we define a pointer to function? The answer to this question is yes. A pointer to function is just like a normal pointer. It contains the address of the function, that is, the value of pointer to function is the address of the function name. Since the name itself is a pointer, therefore it is a constant pointer. A pointer to a function is declared as a pointer to the data type returned by the function, such as `void`, `int`, `float`, etc. The general syntax for pointer to function is as follows.

Syntax

```
return type (*pointer variable)(list o
```

The pointer variable is always enclosed in parenthesis; otherwise C/C++ compiler interprets it as a function which returns a pointer type value. Using pointer to function, the function is called as follows:

```
sum=(*ptr)(var1,var2)
```

where var1 and var2 are actual parameters and sum is a variable in which the function returns value. We must note here that declaring a pointer only creates a pointer. It does not create actual function. We have to define the function separately for the task to be performed by the function. The pointers to functions can be used to refer to the function, and they can also be used to pass a function as an argument to another function. Once a pointer to function is declared, the address of the function is assigned to the pointer.

Passing a Function to Another Function

By using pointer to function technique, we can pass the function as argument to another function. The general syntax for passing a function to another function is as follows.

Syntax

```
return type function name((*pointer to parameters));
```

Program 8 A program in C++ to concatenate the strings using function pointers.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>
void main()
{
clrscr();
char *concatenate(char *,char *);
char first[10],second[10],*final;
cout<<"Enter the first string:";
gets(first);
cout<<"Enter the second string:";
gets(second);
final=concatenate(first,second);
cout<<final<<endl;
getch();
}
char *concatenate(char *s,char *t)
{
int i=0,j=0;
char final[100];
```

```
strcpy(final,s);
len=strlen(final);
final[++l]='n';  //to put the space between tw
for(i=len+1;*t!='\0';++i)
final=*i++;
final='i\]0';        //to mark the end of strin
return(final);       //Array name always r
address
}
```

Output

```
Enter the first string: Manish
Enter the second string: Tiwari
Manish Tiwari
```

Program 9 A program in C++ to find the summation of different parts of two complex numbers using pointer to function.

```
#incl<iostream.h
#incl<conio.h
struct complex
{
float real;
float imag;
};
void main()
{
clrscr();
complex A,B,C;
cout<<"Enter complex number A ";
cin>>A.re>>A.imag;
cout<<"Enter complex number B ";
cin>>B.re>>B.imag;
float (*ptr)(complex,complex); //pointer
float sum(complex,complex); //function pr
pt=&sum;       //address of function is assi
function
float sum1=(u*mptr)(A,B); //funcntion is called
cout<<"Sum="<<sum1;
getch();
}
float sum(complex A,complex B)
```

```
{
float C;
C=A.real+B.real+A.img+B.img;
return(C);
}
```

Output

```
Enter complex number A 2 3
Enter complex number B 4 5
Sum= 14
```

Program 10 A program in C++ to find the sum and average of given numbers by using function as argument to another function.

```
#include<iostream.h>
#include<conio.h>
float first,second,final;
void main()
{
float sum(float,float);
float (*ptr)(float,float);//pointer to func
float average(float(*)(float,float),float,
cout<<"Enter two numbers ";
cin>>first>>second;
ptr=&sum;
final=average(ptr,first,second);
cout<<"Average="<<final;
getch();
}
float sum(float a,float b)
{
return(a+b);
}
float average(float (*ptr)(float,float),flo
{
float temp;
temp=(*ptr)(first+second)/2.0;
return(temp);
}
```

Output

```
Enter two numbers 2 3
Average=3.333333
Enter two numbers 2 3
Average=2.5
```

4.8 Pointer to a Pointer

Pointer is a variable which can hold the address of other variable that may also be of pointer type.

Example 2

```
int  *ptr;
int  X;
pt=& X;
```

Here pt is also a variable. We can also hold the address of pt. Let it be 2000. So we can define a pointer to hold the address of pt as

```
ptr*  ptr1
```

where pt itself is a pointer to integer. We can write the above statement as

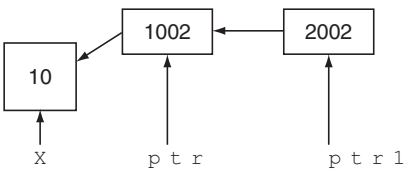
```
int  **ptr1;
```

The pointer to a pointer is a form of multiple indirections and this indirection can be carried out onto whatever extent desired.

```
ptr=& ptr; will assign  2000 to  ptr1.
pt=& X; will assign  1000 to  ptr
```

Now the statement

```
cout<<ptr1; will display  2002
cout<<* ptr1; will display the value at  2002 that is,  1002
cout<<* (*ptr1); will display the value at  1002 that is,  10
```



Program 11 A program in C to demonstrate pointer to pointer.

```
#include<stdio.h>
#include<conio.h>
void  main()
{
int=2x0;
int  *ptr,**ptr1;
clrscr();
pt=& x;
ptr=& ptr;
```

```
printf("\n=%u",ptr);
printf("\n=%d",*ptr);
printf("\n=%u",ptr1);
printf("\n=*%u",*ptr1);
printf("\n=**%u",**ptr1);
getch();
}
```

Output

```
ptr= 65524
*pt= 20
ptr= 65522
*ptr=1 65524
**ptr=120
```

4.9 Dynamic Memory Management

When the memory is allocated at compile time, it is called the static memory allocation. The declarations of variables, array structures, class, etc. that we have done so far are the example of static memory allocation. The global and local variables are allocated memory during compile time. We can allocate memory according to our requirement at the run time and this method of memory allocation is called dynamic memory allocation. We cannot add the global and local variables at run time. This technique provides the way both to allocate the memory at run time and deallocate the unused memory at run time. The dynamic memory allocation provides more flexibility to the C and C++ programmer so that the use of memory space may be optimized. The following functions in C dynamically manage the memory:

1. **malloc()** The syntax is as follows.

Syntax

```
void *ptr=(datatype*)malloc(sizeof(datatype))
```

This function is used to allocate the memory from the free memory space (called heap) of the specified *size* at run time and returns the void type pointer to the first byte of the allocated memory. Since it returns the void pointer so pointer is required to be cast to the appropriate data type and the address of the first byte is kept in the same type of pointer. "void" pointer is a special type of pointer which can point to any data type. The void pointer is declared as void *ptr. If there is no free space in the heap than it returns Null. For example

```
struct student
{
int rollnum;
char name[10];
int age;
float marks;
};
```

```
struct student *ptr, X;
```

(Note: The `struct` keyword is used in C in declaration of the structure variable)

```
ptr=(struct student*)malloc(sizeof( struc
```

Here `struct`, `malloc` and `sizeof` are the keywords. In this case the function `malloc` allocates the memory of 2 + 10 + 2 + 4 = 18 bytes corresponding to the `int roll-num, char name[10], int` of structures, `float` and returns the `void` pointer. Since we need to store it in a structure pointer, typecasting (from `void` to structure pointer) is done and the address is stored in the structure pointer variable `ptr`. Now we can access the members of the structure by using the arrow operator. To access the `rollnum` we use

```
ptr->rollnum
```

The `malloc` function can also be used in C++ to dynamically allocate the memory.

2. **`calloc`** The syntax is as follows.

Syntax

```
void pointer=calloc(number of blocks, size o
```

This is same as `malloc` except that

- It initializes the allocated memory space with zero.
- It requires two parameters.

3. **`free`** The syntax is as follows.

Syntax

```
free(pointer to the block which is t
```

The `free` function is used to release the dynamically allocated memory. Therefore, it is very useful in managing the memory dynamically (i.e., at run time).

4. **`realloc`** The syntax is as follows.

Syntax

```
void pointer=realloc(pointer to already
                     memory, change in size)
```

As the name itself implies this function is used to reallocate the memory. This means that the memory that is already allocated may be reallocated for changing the size of the block. The content of the original block is transferred to the newly reallocated block without any change.

C++ provides some more operators in addition to the functions provided by C. These operators are discussed in the following subsections.

The new Operator

This operator is used to dynamically allocate the memory in C++. It allocates the memory from the free pool of memory called heap and returns a pointer of the appropriate data type, which points to the reserved space. It returns the pointer to the starting point. The syntax for new is as follows.

Syntax

```
pointer_variable_name = new data_type(initial_value);
```

where data_type is any valid data type of C++ and the pointer variable is a variable which follows the usual rules of naming.

The allocated memory can also be initialized at the same time by giving the value within the parenthesis. This initial value is optional. The new operator allocates the memory equal to the size of specified data type and returns a pointer of the same data type for which memory is allocated. Therefore, the pointer variable points to the newly allocated memory. The memory allocated through new remains allocated until it is deleted through delete operator. The new operator fails if the program runs out of memory. In that case, it returns a null pointer. The memory allocation at run time is also called dynamic memory allocation. The memory allocated by new operator remains until it is deleted using the delete operator.

Example 3

```
int *ptr = new int;
```

This statement will allocate the memory (two bytes) for an integer and address of the location is stored in ptr.

```
int *ptr = new int(20);
```

This statement will allocate the memory for an integer and initialize it with 20. The address of the location is stored in ptr.

The new operator can be used to allocate memory for all types of objects like int, float, arrays, structures and classes. The syntax is as follows.

Syntax

```
pointer_variable_name = new data_type[size];
```
(The syntax for allocating memory for one-dimensional array.)
```
pointer_variable_name = new data_type[size1][size2];
```
(size1 may be omitted)
(The syntax for allocating memory for two-dimensional arrays.)

Example 4

```
int *ptr=new int[5];
```

Memory is allocated for one-dimensional array and base address is stored in ptr.

```
char *ptr=new a[2][3];
```

Memory is allocated for two-dimensional array and base address is stored in ptr.
We can access the elements of an array using pointers like

```
cin>>*(ptr+1);to read second element of the array.
cout<<*(*(ptr+2)+1);to display the element of third row and second column, that
is, Array[2][1].
```

The memory may be allocated dynamically for the object using the operator new or malloc() function in C++. The arrow operator is used to access the members of a class when memory is dynamically allocated using the new operator or malloc() function. If malloc() function is used then we need to include alloc.h file.

The delete Operator

The delete operator is used to release the memory allocated through the new operator. It returns the memory pointed to by a pointer to the free store called memory heap. It takes no arguments for deleting a single instance of a memory variable created by a new operator. The general form of delete is as follows.

```
delete pointer variable;
```

Here pointer variable is a pointer that holds the address of the object created with new. The delete operator must define a pointer name only but not with the data type. To release the memory allocated through new for the array, we use the following form of delete

```
delete[size]pointer variable;
```

When the objects are created dynamically, they can be deleted using the delete operator. The delete operator releases the memory pointed by the pointer.

Program 12 A program in C++ to dynamically allocating and freeing the memory for an object.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class date
{
int mm,dd,yy;
public:
void inputdate()
```

```
{
cout<<"\nEnter the date in DD/MM/YY format";
cin>>dd>>mm>>yy;
}
void displaydate()
{
cout<<"\nThe entered date is: ";
cout<<dd<<":"<<mm<<":"<<yy;
}
};
void main()
{
clrscr();
date *ptr;      //ptr is a pointer to class d
ptr=new date; //dynamically creating space fo
ptr->inputdate();   //we can access the membe
using pointers-playdate();//arrow operator
cout<<"\nAllocated space is freed";
delete ptr;          //free the memory pointed
getch();
}
```

Output

```
Enter the date in DD/MM/YY format
11 3 2007
The entered date is: 11:3:2007
Allocated space is freed
```

Solved Examples

Example 1 *Write a program in C++ to find the sum of array elements using pointer to function.*

Solution

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int a[]={1,2,3,4,5};
int n;
int (*ptr)(int[],int); //pointer to funtion
int sum(int a[],int); // function prototype
ptr=&sum;      // address of function is assig
```

```
function
int s=a(m*1ptr)(a,5); //funcntion is called
cou<<" sum=m"<<sum1;
getch();
}
int sum(int a[],int n) // function is de
{int=0s;
for(i=0;<5;++)
{
s=s+a[i];
}
return(s);
}
```

Output

```
Sum=15
```

Example 2 *Write a program in C to find the sum of all odd or all even natural numbers upto a given limit.*

Solution

```
#incl<stdlib.h>
#incl<conio.h>
int *sumodd(int num)
{
int i;
int o=0;m
for=1i;<=num;i+=2)
osum=osum+m;
return &osum;
}
int *sumeven(int num)
{
int i;
int e=0;m
for=0i;<=num;i+=2)
esum=esum+m;
return &esum;
}
void main()
{
int limit,op;
int *sumodd(int n);// function which retu
int *sumeven(int n);
```

```
int *ptr;
clrscr();
printf("Enter the end number ");
scanf("%d",&limit);
printf("\n1. Sum of odd natural numbers ")
printf("\n2. sum of even natural numbers "
printf("\nYour option: ");
scanf("%d",&op);
if(op==1)
{
pt=sumodd(limit);
printf("\nThe sum of all odd natural numb
%d",limit,*ptr);
}
else if(op==2)
{
pt=sumeven(limit);
printf("\nThe sum of all odd natural numb
%d",limit,*ptr);
}
else
printf("\nOption is not 1 or 2 ");
getch();
}
```

Output

```
Enter the end number 10
1. Sum of odd natural numbers
2. sum of even natural numbers
Your option: 2
The sum of all odd natural numbers up to 1
```

Summary

1. Pointer is a variable which can hold the address of another variable.
2. Only addition and subtraction are possible with pointer variable.
3. When each element of an array is a pointer, such arrays are called arrays of pointers.
4. A pointer supports dynamic allocation and deallocation of memory at run time.
5. The functions `malloc()` and `calloc()` are used in C to dynamically allocate the memory.
6. The functions `new` and `malloc` are used in C++ to dynamically allocate memory.
7. There are two special operators which are used with pointers: address of operator and value at operator.

8. We can use pointers with arrays, structures, classes, functions, etc.
9. The `free` function is used release the memory allocated by using `malloc` and `calloc` and `delete` is used to release the memory allocated through `new` operator.
10. With the help of pointers memory can be utilized very efficiently.
11. Null pointers are also called zero pointers.

Key Terms

`void` pointer	`new` operator	Pointer to array
Address of operator	`delete` operator	`malloc()`
Value at operator	Pointer to function	`calloc()`
Null pointer	Pointer to pointer	`realloc()`
Indirection operator	Array of pointers	`free()`

Multiple-Choice Questions

1. `int A={1, 2, 3, 4};` Suppose the base address is 1000 then `cout<<(A+1)` will display
   a. 2
   b. 1
   c. 1000
   d. 1002
2. `char A="Umang";` Suppose the base address is 1000 then `printf("%u",A+1)` will display
   a. 1002
   b. 1001
   c. Umang
   d. mang
3. `char A="Umang";` Suppose the base address is 1000 then `cout<<*(A+1)` will display
   a. m
   b. u
   c. garbage value.
   d. none of the above.
4. `struct complex{int real; float imag}X; complex *ptr=&X;` Which of the following is correct?
   a. `ptr->real;`
   b. `ptr.real;`
   c. `X->imag;`
   d. All of the above
5. `int A[2];` then `A[1]` is equivalent to
   a. `*(A+1)+2;`
   b. `*(*(A+1)+2);`
   c. `*(A[1]+2);`
   d. both (b) and (c).
6. `int A[2][4];` then `A[3][0][0][0]` is equivalent to
   a. `*(*(*(A+0)+0)+0);`
   b. `*(*(A+0)+0);`
   c. `*(A[0]+0);`
   d. all of the above.
7. `char X='X'; char Y="YY"[1];` then `cout<<sizeof("X")<<sizeof(Y)` will display
   a. 2 1
   b. 1 1
   c. 1 2
   d. 0 0
8. `float A[]={1.2, 2.3, 3.5, 4.1};` then `cout<<sizeof(A)` will display
   a. 20
   b. 10

c. `5`
d. `0`

9. Which operator is used to access the structure member through pointer to structure?
   a. Arrow operator
   b. Dot operator
   c. Scope resolution operator
   d. Ternary operator

10. Which of the following is a valid declaration for pointer to function in C++?
    a. `(float *) example();`
    b. `float (*)example();`
    c. `float *example();`
    d. All of the above

11. Which of the following is a valid function declaration which returns a pointer?
    a. `(double) (example*(double, double));`
    b. `double (*) example(double, double);`
    c. `double (example(*double, *double));`
    d. `double *example(double, double)`

12. Which of the following operator is used for dynamically allocating the memory?
    a. `new`
    b. `malloc`
    c. `calloc`
    d. All of the above

13. Which of the following operator is used for dynamically deallocating the memory?
    a. `delete`
    b. `destructor`
    c. `destroy`
    d. `void`

14. Which of the following operator is not used for dynamically allocating the memory in C?
    a. `new`
    b. `malloc`
    c. `calloc`
    d. All of the above

15. Which of the following operator is used for dynamically deallocating the memory in C?
    a. `delete`
    b. `free`
    c. `destroy`
    d. `void`

Review Questions

1. What is a pointer?
2. What is the relationship between an array and a pointer?
3. What is the difference between `&` and `*` operator?
4. Which operations are possible with pointer variables?
5. How are arrays and pointers related?
6. What do you understand by pointer to pointer?
7. What are the differences between array of pointers and pointer to array?
8. What are the differences between array of pointers and pointer to pointer?
9. What are the differences between `malloc()` and `new` operator?
10. What are the differences between `free()` and `delete`?
11. If `A` is an array then can you use `A++` in the program? If not, why?
12. What is the difference between `A[i]` and `i[A]`?
13. Write `A[I][J]` in pointer notation.
14. If `A` is a two-dimensional array than what will `*(A)` give?
15. What are the advantages of using pointers with string?

16. Which operator is used to access the members of a structure and a class using pointer?
17. Which pointer holds the address of an object?

Programming Assignments

1. Write a program in C to find the smallest and largest element of an array using dynamic memory management technique.
2. Write a program in C++ to read and display the employee details (ename, designation, age, dob in dd/mm/yyyy format) using dynamic memory allocation and pointers.

Answers

Multiple-Choice Questions

1. (d)
2. (b)
3. (a)
4. (a)
5. (b)
6. (d)
7. (c)
8. (a)
9. (a)
10. (b)
11. (d)
12. (d)
13. (a)
14. (a)
15. (b)

5 Stack and Queues

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Concept of stacks and queues.
- Primitive operations on the stack.
- Applications of stack.
- Polish notations.
- Recursion.
- Operations on the queue.
- Different types of queues.

In Chapter 2, we discussed the linear list called an array in which the insertion and deletion of an element can take place at any position of the array. In this chapter, we will discuss a different kind of linear list, called the restricted list, in which addition and deletion of items is restricted to the ends of the list. We describe two such most commonly used linear list structures: the *stack* and the *queue*.

5.1 Stack

Stack is a linear data structure used for temporary storage where elements can be inserted or deleted at only one end, called the "Top" of the stack; that is, the addition and deletion of elements is restricted to only one end (Top) in the case of stack. The Top of the stack is the only "enter and exit" end. The element inserted last will be the first element which can be deleted from the stack. This is the reason why the stack is also called last in first out (LIFO). Stack of dishes, stack of coins, stack of plates in a restaurant, stack of books, stack of folded towels are a few examples of stack. Figure 1 illustrates the concept of stack using the example "stack of books".

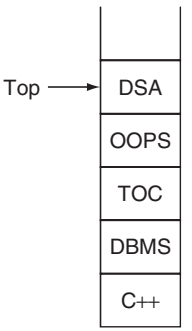


Figure 1 A stack of books.

In Figure 1, one can observe that the first book that can be taken out from the stack is "DSA", the topmost book on the stack as indicated by the arrow "Top". Then we can withdraw the book "OOPS" and so on. We cannot withdraw the book "C++" unless all the books above it have already been withdrawn.

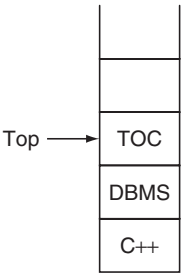


Figure 2 Removal of two books from the stack.

Similarly, if we now want to put another book "JAVA" on the stack of books shown in Figure 2, we have to first move to the location above the "TOC" (Figure 2) and only then we can put the book on that location as shown in Figure 3.

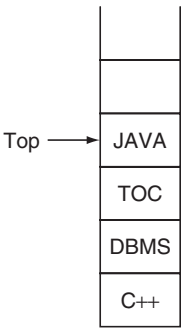


Figure 3 Addition of one book in the stack.

Stacks are also known as *pushdown lists*. They have many applications in computer science especially in temporary storage. We will discuss some of them in this chapter.

5.2 Stack Representation and Implementation

There are two common ways to implement most of the data structures. Stack can also be represented in these two ways:

1. Static or sequential or array representation (implementation).
2. Dynamic or pointer or linked representation (implementation).

Static or Array Representation (Implementation)

As in the case of arrays, stack is also a collection of homogenous elements. Therefore, stack can be easily implemented using an array. A stack can be implemented as an array named Stack of Max elements, where Max is the maximum size of the array; that is, the maximum number of elements an array named Stack can hold. One end of the array is fixed to represent the bottom of the stack and the other end is kept open to perform insertion and deletion operations. The stack usually grows at this open end where the top of the stack constantly shifts as the items are popped and pushed. The array Stack has elements stored from Stack[0] to Stack[Top], where Stack[0] represents the bottom of the Stack and Top is an

integer which holds the index number of the topmost element of the stack. As already discussed, the elements can be inserted or deleted from the "Top" end only. Initially the stack is empty and it is represented by assigning Top = –1 as shown in Figure 4(a). We are not using Top = 0 because array index starts from 0. Let us define an array named Stack having seven elements, that is, Stack[7] index ranging from 0 to 6. So Max in this case is 7. Figure 4(b) illustrates the situation when some of the elements are inserted into the stack and Figure 4(c) shows the situation when the stack is full (i.e., all the elements have been inserted into the array named Stack).

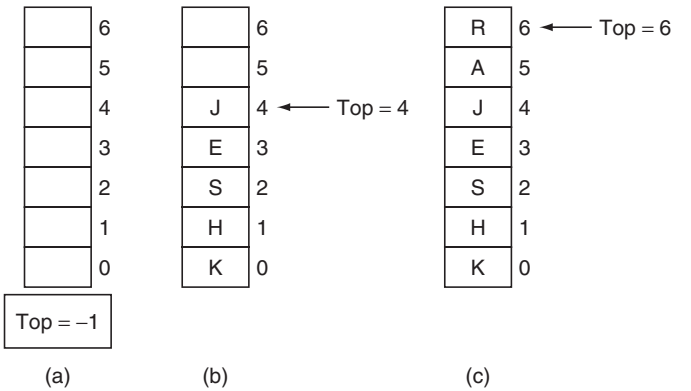


Figure 4 (a) Empty array Stack; (b) some elements inserted into the array Stack; (c) array Stack is full.

Therefore, in the static implementation we must take care that the array must be large enough to hold all stack elements. Though the array implementation is very simple and efficient, still it requires the ultimate size of the stack to be declared beforehand. This sometimes results in wastage of memory and sometimes in crashing of the application. Once the size of the array is declared, the size cannot be varied during program execution in the static implementation. For example, if we declare an array stack of size 7 before execution of the program then the memory is allocated for the array of size 7; however, an application may actually need much less space (let us say for 4 elements). This results in wastage (for 3 elements extra) of memory. Similarly, an application may need more number of elements (let us say for 10 more elements) to be stored in the stack. In such a case, we cannot change the size of array to increase its capacity. This may crash the application as soon as it tries to push its 8th element on the stack because array allocates memory in contiguous location.

Dynamic or Pointer Representation (Implementation)

The array implementation of stack is very simple and efficient. However, it requires the ultimate size of the stack to be declared in advance. The advance declaration of Max size is not possible all the time; therefore, we can say that array representation of stack is not flexible. The size of the stack may increase or decrease according to the requirement of the application because the stack size is constantly changing as the items are popped and pushed. In the static implementation, once the size of the array is declared, the size cannot be varied during program execution; therefore, static implementation is not an efficient method when resource optimization is concerned. Fortunately, dynamic implementations of the stack do not have size limitations and use space proportional to the actual number of elements stored in the stack.

The dynamic representation of a stack is also called linked representation of stack. The linked representation, commonly termed as linked stack, is more suitable if an accurate estimate of the

stack's size cannot be made in advance and the data elements are large records. In the linked representation, the stack is a collection of nodes where each node is divided into two parts. The first half (*info* field) of the node contains the element of the stack, and the second half (*next* field) holds the pointer to the neighboring element in the stack as shown in Figure 5(a).

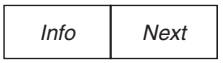


Figure 5 Node representation.

The Top pointer of the stack is represented by the *start* pointer of the linked list. If the *start* pointer is null then it shows that the stack is empty. It can be indicated by Top = Null. If the last node link field contains Null then it shows the bottom of the stack. Figure 6 shows the linked or dynamic representation of the stack.

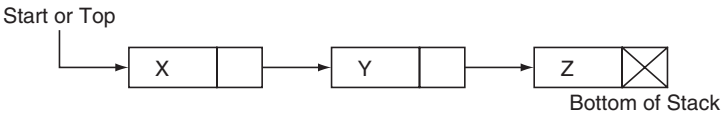


Figure 6 Dynamic representation of stack with three elements.

In this figure, the stack contains three elements Z , Y and X, where Z is at the bottom of the stack. We can see that the *info* field of the node contains items Z , Y and X and the next pointer of the node contains the address of the next element (node). Figure 7 illustrates a stack in the form of linked list.

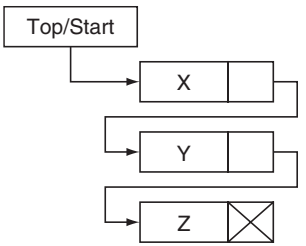


Figure 7 Physical dynamic stack implementation.

Irrespective of the way a stack is described, its underlying property is that insertions and deletions can occur only at the top of the stack. The implementation of the linked stack is given in Chapter 6.

5.3 Stack Operations

There are two basic operations that are usually performed with stacks:

1. Push;
2. Pop.

Let us discuss each one in detail.

P ush Operation

The Push operation is basically used to insert the elements into the stack. When we say "push an element", we mean insert an element into a stack. Each time a new element is inserted into the stack, the value of the Top is incremented by one before placing this new element. Let us suppose, we want to insert five elements A, B, C, D and E into the stack. The graphical representation of a stack containing these elements is shown in Figure 8.

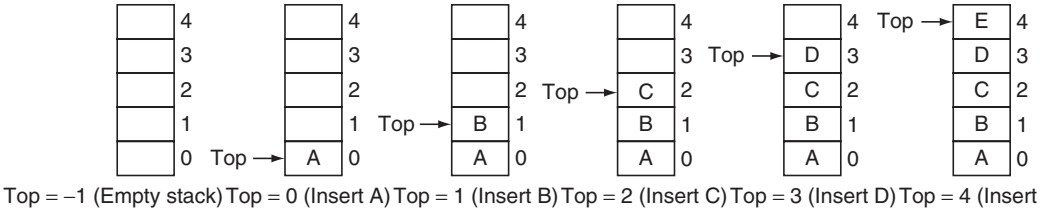


Figure 8 A stack Push operation.

We can see from Figure 8 that the elements are inserted only from the top. This implies that pushing an element into a stack will add the new element at the top. The position "Top" always points to the recently inserted item. *We should note here that each time a new element is inserted into the stack, the value of the Top is incremented by one before placing the new element into the stack accordingly to keep track of the current Top of the stack.* Therefore, after every Push operation the Top is incremented by one and the new item becomes the Top. Empty stack is shown by Top = –1 (in array representation) and Top = Null (in dynamic representation). When we insert A into the stack, the Top is increased by 1 so it becomes 0 (in array representation) and at this location of the stack, element A is inserted. In the end, E is inserted so now the pointer Top is pointing to the position where E is placed. In this case Top = 4, that is, Max – 1. We have reached to the maximum index so we cannot add more items to the stack. This situation is called *overflow condition*. If the memory is full and no new element can be accommodated, then the stack overflow condition occurs. We can perform the push operation only when the stack has sufficient memory space to accommodate the new element. The algorithm for push operation is given in Algorithm 1. Here STACK is an array of MAX size and we want to insert an ELEMENT

Algorithm 1 PUSH(STACK, MAX, TOP, ELEMENT)

1. Check for Overflow

```
If TOP = MAX - 1 then Print "Overflow" and R
Else
a. TOP = TOP + 1          // Increment the Top by O
b. STACK[TOP] = ELEMENT   // Insert the ELEMENT
   location
End if
```

2. Return

P op Operation

Pop operation is basically used to remove/delete the elements from the stack. So when we say "pop an element" we mean delete an element from the stack. Let us suppose we want to remove all the five elements from the stack shown in Figure 8. The graphical representation of such removal from the stack is shown in Figure 9.

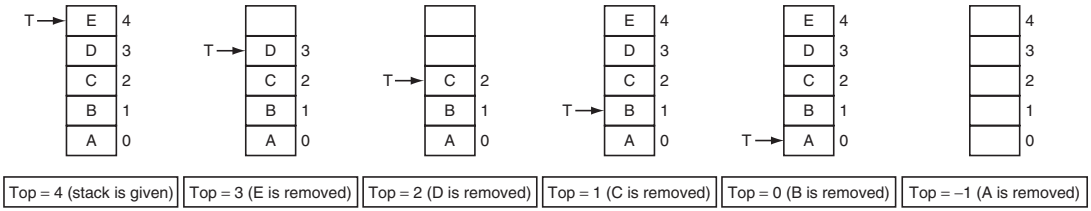


Figure 9 Graphical representation of stack pop operation.

The most recently inserted element (i.e., E) is at the top so it is deleted first from the stack. The element below the Top element (i.e., D) is deleted next because once the previous Top element E is removed, the next item D in the stack becomes the new Top element. The value of Top changes after the deletion in stack takes place. In this way, the element inserted first (i.e., A) is deleted last from the stack. When the last item in the stack is deleted, the stack is set to an empty state. The elements from the stack are always retrieved in the reverse order. As we can see from Figure 9, the elements are removed in the reverse order (E, D, C, B, A) in which they were pushed (A, B, C, D, E) into the stack (Figure 8). Removal of an element depends upon its position in the stack; you have to first remove all the elements above it. Therefore, you cannot delete B unless all the elements above it (i.e., C, D and E) are removed. We can see in Figure 9 that after every pop operation, the stack Top is decremented by one. The most frequently accessible element in the stack is the topmost element whereas the least accessible element is at the bottom of the stack. When all the elements from the stack are removed (i.e., stack is empty), we cannot delete anything from the stack. This situation is called ' underflow'. If the pop operation is performed in the empty stack then the stack underflow condition occurs, so one must test whether there is an element in the stack to be deleted. We can perform the pop operation only when the stack is not empty. The algorithm for pop operation is given in Algorithm 2. Here S T A C K is an array of M A X size and we want to delete the E L E M E N T from the S T A C K

Algorithm 2 P O P (S T A C K , M A X , T O P , E L E M E N T)

1. Check for Underflow
   I f T O P = 0 t h e n P r i n t " U n d e r f l o w " a n d R e t u
   E l s e
   a. E L E M E N T = S T A C K [T o p] // D e l e t e a n E L E
      t h e S T A C K
   b. T O P = T o p - 1 // D e c r e m e n t t h e T o p b y O n e
   E n d i f
2. R e t u r n

Program 1 **A program in C++ to implement push and pop operations on the stack.**

```
#include<iostream.h>
#include<conio.h>
#include<process.h> To use cerr
#define SIZE 10
class LIFO
```

```
{
int STACK[SIZE];
int TOP;
public:
LIFO()
{
TOP=-1;
}
void push(int NUM)
{
if(TOP==SIZE-1)
cerr<<"Stack is full-<<endlr;flow"
else
{
TOP++;
STACK[TOP]=NUM;
clrscr();
display();
}
}
void pop()
{
if(TOP==-1)
{
cerr<<"Stack is empty-<<endlr;flow"
}
else
{
cout<<"\n Number popped="<<STACK[TOP];
TOP--;
display();
}
}
void display() // To display the contents o
{
cout<<endl;
cout<<"Stack contents-<<endl;"
for(i=0;i<=TOP;i++)
cout<<STACK[i]<<"  ";
}
};
void main()
{
clrscr();
LIFO STK;
```

```
int ch=1, num;
while(ch!=0)
{
cout<<"\n** Stack Operation Menu **";
cout<<"1. PUSH"<<endl;
cout<<"2. POP"<<endl;
cout<<"Enter the choice (0 for exit): ";
cin>>ch;
switch(ch)
{
case 1:
cout<<" Enter the number of push";
cin>>num;
STK.push(num);
break;
case 2:
STK.pop();
break;
default:
cout<<"illegal option";
}
}
getch();
}
```

Output

```
Stack contains
11    12    13
** Stack Operation Menu **
1. PUSH
2. POP
Enter the choice (0 for exit): 2
 Number popped is 13
Stack contains
11    12
** Stack Operation Menu **
1. PUSH
2. POP
Enter the choice (0 for exit): 2
 Number popped is 12
Stack contains
11
** Stack Operation Menu **
1. PUSH
2. POP
```

```
Enter the choice (0 for exit): 2
 Number popped is
Stack contains
** Stack Operation Menu **
1. PUSH
2. POP
Enter the choice (0 for exit): 2
Stack is empty-Underflow
** Stack Operation Menu **
1. PUSH
2. POP
Enter the choice (0 for exit): 0
```

Program 2 **A program in C++ to implement stack operations using template.**

```
#include<iostream.h
#include<conio.h>
#define MAX 3
template<class T
class stack
{
T STK[MAX];
public:
T ITEMI,ITEMD;
int TOP;
stack()
{
TOP=-1;          //TOP indicates an empty stack
}
void push()
{
if(TOP==MAX-1)                              //Check for
cerr<<"Stack is full-Overflow"
else
{
TOP++;
cout<<"Enter the value to push?";
cin>>ITEMI;
STK[TOP]=ITEMI;          //Place the recently read
stack
clrscr();
display();                    //To display the cont
}
}
```

```
void pop()
{
if(TOP==-1)
{
cout<<"STACK IS EMPTY: UNDERFLOW";
}
else
{
ITEM=STK[TOP];          //Keep the Topmost Value
cout<<"The Popped Item from the STACK is: "<<ITEM;
STK[TOP]=NULL;          //Set Original Location
TOP--;                  //Decrement Top by
}
display();
}
void display()
{
cout<<endl;
cout<<"\nNow maximum elements =>"<<(MAX-TOP-1);
cout<<"\nStack contents: "
for(i=0;i<=TOP;i++)
cout<<STK[i]<<" ";
}
};
void main()
{
clrscr();
stack<int>obj;   //Create object 'obj' of cla
Template
int ch;
do
{
cout<<"\n***Menu***";
cout<<"\n(1) Push Item";
cout<<"\n(2) Pop Item";
cout<<"\nEnter your choice (0 to exit)";
cin>>ch;
switch(ch)
{
case 1:
obj.push();
break;
case 2:
obj.pop();
break;
}
```

```
}while(ch!=0);
getch();
}
```

Output

```
Now maximum elements you can push is 0
Stack contains
11   12   13
***Menu***
(1) Push Item
(2) Pop Item
Enter your choice (0 to exit): 1
Stack is full-Overflow
***Menu***
(1) Push Item
(2) Pop Item
Enter your choice (0 to exit): 2
The Popped Item from the Stack is: 13
Now Maximum elements you can push is 1
Stack contains
11   12
***Menu***
(1) Push Item
(2) Pop Item
Enter your choice (0 to exit): 0
```

5.4 Applications of Stack

Stacks are basically used as a temporary storage especially for dealing with nested structures or processes; so they are more useful in indicating the order of the processing of data. Therefore, they are useful when the application requires that the usage of data be postponed for a while. For example, in finding the factorial of a given number, say 5!, we have to postpone the calculation of 5! because we first need to execute the steps to find the value of 4!. In this case, first the procedure for calculating the factorial of 5 is kept in the stack, and the procedure for calculating the factorial of 4 is executed. However, for calculating the factorial of 4, the factorial of 3 is needed and so procedure of 4 is also kept in the stack. This procedure continues.

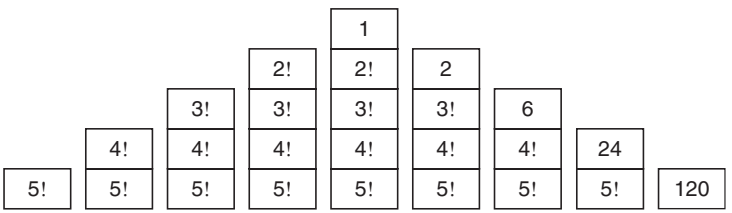


Figure 10 An application of stack.

We know that 1! = 1. Therefore, the execution of 1! does not require any subprogram. It is popped and executed and returns its value. The value of 2! is calculated after popping it from the stack and following the same procedure. Continuing this way, we find the factorial of 5! as shown in Figure 10. It (5!) was the first element inserted in the stack but last one to be popped from the stack. There are some other applications such as reversal of the string, parsing, recursion, matching the nested parenthesis in arithmetic expression, backtracking and infix to postfix transformation as well as postfix expression evaluation in which stacks are used. We will discuss some of the applications in this chapter.

R epresentation of Arithmetic E x pression

An arithmetic expression is made up of operands and operators. The binary operations may have different levels of precedence. Sometimes the precedence is changed by using parentheses. An arithmetic expression can be represented in three different formats according to the relative position of the operator with respect to the two operands. The different categories are

1. Infix;
2. postfix;
3. prefix.

Next we discuss each one in detail.

Infix Notation

This notation of arithmetic expression is used in most arithmetic operations. In this method, the operator comes in between two operands; note that *only unary operators precede their operand*.

Example 1

1. `A+B`
2. `A+B * C`
3. `X+Y * Z ^ M`
4. `X+Y * Z – M / N`

To determine the final value of the expression, the precedence of operators is determined by the BODMAS rule. We consider the operations (a) addition, (b) subtraction, (c) multiplication, (d) division and (e) exponentiation. We assume the following three levels of precedence for these five binary operations in such notations:

1. **Highest:** Exponentiation.
2. **Next highest:** Multiplication and division.
3. **Lowest:** Addition and subtraction.

The expression may use parenthesis to change the usual order of precedence otherwise operations on the same level precedence are performed from left to right except in the case of exponentiation where operation are performed from right to left if they appear on the same level of precedence.

Example 2

1. 2+3 * 4 2+1 2=1 4
2. 2 * 3 ^ 2=2 5 9 =1 5 8 −=5 1 3
3. 2 * 3 * 2 ^ 2=2 * 3 * 2 2=2 4 3 * 1 6 * 1=8 6

Polish Notations

Other than infix notations, there are other notations of arithmetic expression called the Polish notations, in which the operators are used either before or after their operands. These representations were given by the Polish mathematician, Jan Luksiewicz. The fundamental property of Polish notations is that the order in which the operations are performed is determined by the positions of the operators and operands in the expression. Compared to infix notations, parentheses are not required in these notations to determine the order of operations in any arithmetic expression. The priority of operators is no longer relevant.

Any infix expression can be converted into Polish notation. The only rule to remember during this conversion is that

1. operators with higher precedence are converted first.
2. after a portion of the expression has been converted to Polish notation, it must be treated as a single operand in the rest of the expression.

Note that the order of the operands in all the forms is same. The Polish notations enable easy evaluation of expressions. Two such notations are the prefix and the postfix notations.

Prefix Notation (Polish Notation) When the operators are written before their operands, the expression is said to be in prefix notation. For example $+XY$; $-+XY * YZ$; $-+X * YZ/MN$ are in prefix notation.

Conversion from Infix to Prefix

Let us take the infix expression

$$X + Y * Z - M/N$$

We know that multiplication and division have the same precedence and they are on the same level, so the conversion should be performed from left to right. Therefore multiplication is performed first and so $Y * Z$ becomes $*YZ$. Now set $T_1 = *YZ$ where T_1 is a single argument. The given expression becomes

$$X + T_1 - M/N$$

In the next step division is performed, that is, M/N is converted into $/MN$. Set it as T_2

$$X + T_1 - T_2$$

We know that addition and subtraction are on the same precedence so they should be evaluated from left to right. Therefore $X + T_1$ becomes $+XT_1$. Set it as T_3. The expression is now converted into $T_3 - T_2$ which is equivalent to $-T_3T_2$ in prefix notation. Therefore, the final expression in prefix notation is

$$-T_3T_2$$

Now, replace the variables T_1, T_2 and T_3 with the corresponding values

$$-+XT_1T_2 \text{ (substituting } T_3 = +XT_1)$$
$$-+XT_1/MN \text{ (substituting } T_2 = /MN)$$
$$-+X * YZ/MN \text{ (substituting } T_1 = *YZ)$$

Postfix Notation (Reverse Polish Notation)

In this notation, the operator comes after the operands in the arithmetic expression. For example $XY+$; $XY + YZ*-$; $XYZ*+MN/-$ are in postfix notation. This notation is very important because the common evaluation technique used by the computer is that it accepts an infix expression and produces the correct code by converting it into postfix expression. Then the evaluation of the expression is done.

Conversion from Infix to Postfix

Let us take the infix expression

$$X + Y * Z - M/N$$

We know that multiplication and division have the same precedence and they are on the same level so these operations should be performed from left to right. Therefore, the multiplication is performed first. $Y*Z$ becomes $YZ*$. Set it as T_1, a single argument. The given expression now becomes

$$X + T_1 - M/N$$

Next, division is performed. So M/N is converted into $MN/$. Set it as T_2; therefore

$$X + T_1 - T_2$$

We know that addition and subtraction are on the same precedence so they should be evaluated from left to right because they are on the same level. Therefore $X + T_1$ becomes XT_1+. Set it as T_3. Therefore, the expression is now converted into $T_3 - T_2$ which is equivalent to T_3T_2- in prefix notation. Therefore, the final expression in postfix notation is

$$T_3T_2-$$

Now, replace the variable T_1, T_2 and T_3 with the corresponding values

$$XT_1 + T_2- \text{ (substituting } T_3 = XT_1+)$$
$$XT_1 + MN/- \text{ (substituting } T_2 = MN/)$$
$$XYZ* + MN/- \text{ (substituting } T_1 = YZ*)$$

Here we can see that the prefix form is not the mirror image of the postfix form. The following algorithm (Algorithm 3) is used to transform the infix expression into the postfix expression, where in, each step of the stack is the main tool that is used to accomplish the given task. Let us assume that the given infix expression is I. The infix expression is scanned from left to right until we reach the end of the expression which may be marked by putting some sentinel or delimiter #.

Algorithm 3 `Postfix (I, P)`

Take the empty stack, Infix expression and put the special character # as a sentinel to mark the end of infix expression. Place the # in the stack to mark empty stack. Here P represents the postfix expression which is initially empty

1. Read the infix expression (string) I from left to right one character at a time.
2. Repeat Step 3.
3. Read Symbol=next input character from the infix expression.
   a. If the scanned symbol is an operand then put it in the postfix string.
   b. Else if the scanned symbol is an open parenthesis then *push* it onto the stack.
   c. Else if the scanned symbol is a closing parenthesis then *pop* from the top of the stack until open parenthesis is encountered; concatenate the popped elements into postfix expression in their order of popping.
   d. Else if the scanned symbol is an operator then
      (1) If the incoming operator has the same or less precedence than the operators already available at the top of stack, then *pop* all such operator from the top of the stack and concatenate them to postfix string.
      (2) Push the incoming operator to the stack.
4. Until sentinel # is encountered.
5. Pop all the elements from the stack to make the stack empty.

Problem 1

Convert the following infix expression into postfix expression:

$$((A + (B - C))/D)$$

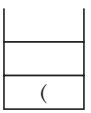
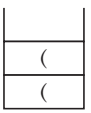
Solution

The sentinel # is placed at the end of the infix expression as

$$((A + (B - C))/D)\ \#$$

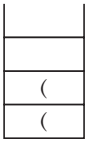
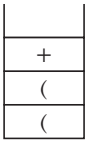
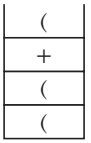
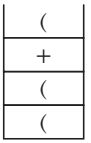
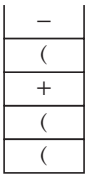
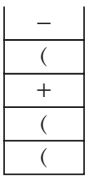
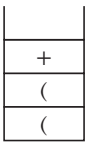
The conversion from infix to postfix notation is shown in Table 1.

Table 1 Representation and status of the stack in converting an infix expression to a postfix expression

S. No.	*Symbol scanned*	*Postfix expression*	*Stack*	*Pictorial representation*
1	(		(	
2	(		((	

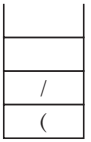
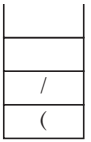
(*Continued*)

Table 1 (Continued)

S. No.	*Symbol scanned*	*Postfix expression*	*Stack*	*Pictorial representation*
3	A	A	((	
4	+	A	((+	
5	(	A	((+(	
6	B	AB	((+(	
7	–	AB	((+(–	
8	C	ABC	((+(–	
9	)	ABC–	((+	

(*Continued*)

Table 1 (Continued)

S. No.	*Symbol scanned*	*Postfix expression*	*Stack*	*Pictorial representation*
10	)	ABC–+	(	
11	/	ABC–+	(/	
12	D	ABC–+D	(/	
13	)	ABC–+D/		
14	#	Stop		

Evaluation of Postfix Expression

Once the infix expression is converted into the postfix expression, it is required to be evaluated. In the evaluation process, once again the stack plays an important role of temporary storage. To evaluate the postfix expression we start reading the postfix expression from its left. If an operand is encountered then it is placed into the stack and if an operator is encountered, then we pop the top two elements to perform the indicated operations on them. Since each operator refers to the previous two operands in the postfix string, we pop the two elements from the stack and apply the operator between the popped elements. We should note the order of operands when the operator is applied between them. If the Top operand is Var1 and next to Top is Var2, then the operator (op) is applied between the two operands as Var2(op)Var1. The result of this expression is again pushed into the stack which will be used as an operand for another operator if any. Algorithm 4 gives the algorithm for evaluation of postfix expression.

Algorithm 4 `Evalpostfix(P, STACK, ANSWER)`

1. Add a sentinel # at the end of P.
2. Scan the given postfix expression P from left to right reading one character at a time.
3. Undertake Steps 1 and 2.
4. Read next character from P.

5. If next character is an operand then push it into the stack
   Else if next character is an operator (op) then
   (a) Pop top element from the stack let it be in Var1.
   (b) Again Pop top(Originally next to top) element from the stack let it be Var2.
   (c) Answer = Var2(op)Var1.
   (d) Push the result Answer to the stack.
   End if
6. Until # is encountered.
7. Pop the top element from the stack let it be `Var3`
8. Set Answer = Var3.

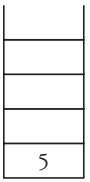
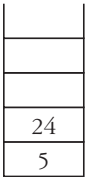
Problem 2

Evaluate the postfix expression 5 12 3 2 2 ^ */–.

Solution

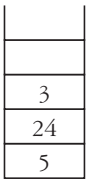
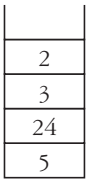
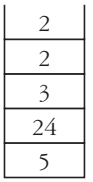
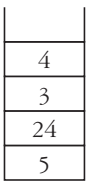
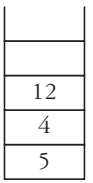
Let the given postfix expression be *P*. Place the sentinel # at the end of *P*. Therefore, *P* = 5 12 3 2 2 ^ */–#. The steps for evaluation are shown in Table 2.

Table 2 Representation and status of stack in evaluating a postfix expression

S. No.	*Character Read*	*Stack*	*Pictorial Representation*	*Remarks*
1	5	5		
2	24	5 24		

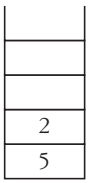
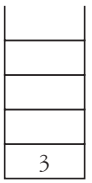
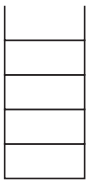
(*Continued*)

Table 2 (Continued)

S. No.	*Character Read*	*Stack*	*Pictorial representation*	*Remarks*
3	3	5 24 3		
4	2	5 24 3 2		
5	2	5 24 3 2 2		
6	^	5 24 3 4		Var1 = 2, Var2 = 2 So 2^ 2= 4
7	*	5 24 12		Var1 = 4, Var2 = 3 So 4*3 = 12

(*Continued*)

Table 2 (Continued)

S. No.	*Character Read*	*Stack*	*Pictorial Representation*	*Remarks*
8	/	5 2		Var1 = 12, Var2 = 24 So 24/12 = 2
9	–	3		Var1 = 2, Var2 = 5 So 5 – 2 = 3
10	#			Pop the Top element of the stack and assign it to Answer = 3

Program 3 **A program in C to evaluate the given postfix expression.**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<ctype.h>
#define MAX 10
char stack[MAX],postfix[MAX];
int top=-1;
void push(char);
char pop(void);
int eval(int x,int y,char op);
int calculate(void);
void main()
{
      char ch;
      do
  {
```

```
top=-1;
clrscr();
printf("\nEnter a postfix expression\n");
gets(postfix);
printf("\nThe Given post fix expression is
%s\n",postfix);
   printf("The result of the given expressi
%d\n",calculate());
   printf("\nDo you wish to continue: ");
   ch=getche();
      }while(ch=='y'||ch=='Y');
}
void push(char sym)
{
      if(top==MAX-1)
      {
     printf("\nStack is full\n");
     exit(0);
      }
      else
     stack[++top]=sym;
}
char pop(void)
{
      if(top==-1)
      {
       printf("\nStack is empty\n");
       exit(0);
      }
      else
     return(stack[top--]);
}
int eval(int x,int y,char op)
{
if(op=='+')
return(x+y);
else if(op=='-')
return(x-y);
else if(op=='*')
return(x*y);
else if(op=='/')
return(x/y);
else if(op=='^')
return(pow(x,y));
```

```
}
int calculate()
{
int i=0;
int x;
int opn1,opn2,ans;
while(postfix[i]!='\0')
return(pow(x,y));
}
int calculate()
{
int i=0;
int x;
int opn1,opn2,ans;
while(postfix[i]!='\0')
{
if(x>='0' && x<='9')
push((int)(x-'0'));
else
{
  opn2=pop();
  opn1=pop();
  ans=eval(opn1,opn2,x);
  push(ans);
  }
  i++;
  }
return ans;
}
```

Output

```
Enter a postfix expression
24^4/32*-
Given postfix expression is:24^4/32*-
The result of the given expression is: -2
Do you wish to continue n
```

Recursion

Another important application of stack is in recursion, where it is used to save the parameters, local variables and return address temporarily. *Recursion is an ability of a function or procedure or algorithm to repeatedly call itself until a certain condition is met*. Such condition is called the *base condition*. The function which calls itself is called a *recursive function*. A recursive function is said to be well-defined if the base condition is defined. Each time when the function calls itself it must be closed to the base condition otherwise recursion will not terminate and would run indefinitely.

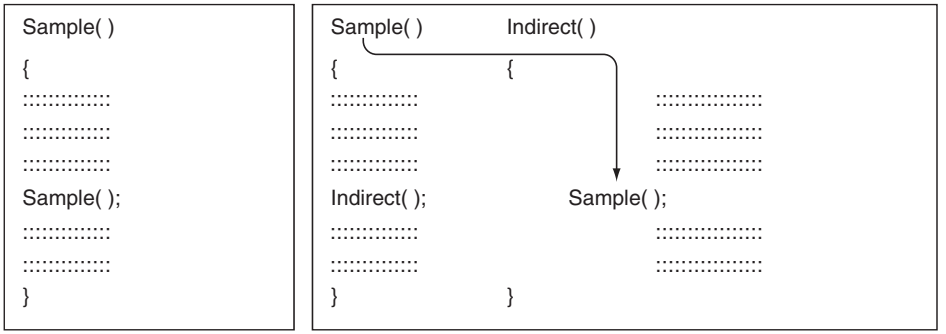


Figure 11 (a) Direct recursion; (b) indirect recursion.

Recursion can be classified into two categories: (a) Direct recursion and (b) indirect recursion. Direct recursion occurs when the function calls itself directly [Figure 11(a)] and indirect recursion occurs when the function calls another function which may eventually call the original function [Figure 11(b)].

Recursion is especially useful in dealing with nested structures. However, it requires more memory and time to implement. Therefore, the function implemented without recursion may work faster.

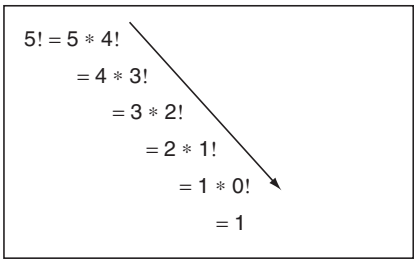
Recursion is implemented using stack which aids in keeping the function arguments, return addresses and local variables.

Problem 3

Calculate factorial of 5.

Solution

The product of the positive integers from 1 to N, both inclusive, is called "factorial of N" and is usually denoted by $N!$ (i.e., $N! = 1 \times 2 \times 3 \times 4 \times 5 \times \ldots \times (N-2) \times (N-1) \times N$ or $N \times (N-1) \times (N-2) \times \ldots \times 4 \times 3 \times 2 \times 1$ where $0! = 1$. To find out the value of 5! we follow the sequence as



The value of 5! is the product of 5 and 4! So to find the factorial of 5, we first require the value of factorial of 4. Observe that for every positive integer N, the value of $N! = N \times (N-1)!$ So the definition is recursive as shown in Figure 12. Since it refers to itself, therefore, the factorial function calls itself with different parameter values, first 5 then 4 then 3, 2, 1 and at last 0. For the value 0 the factorial does not call itself because the default value for 0! is 1. This is the base condition, $0! = 1$.

We can see in the solution of Problem 3 for calculating 5! that every time a factorial is calculated with the values that lead to the base value as shown by the arrows. The algorithm for calculating $N!$ is given in Algorithm 5.

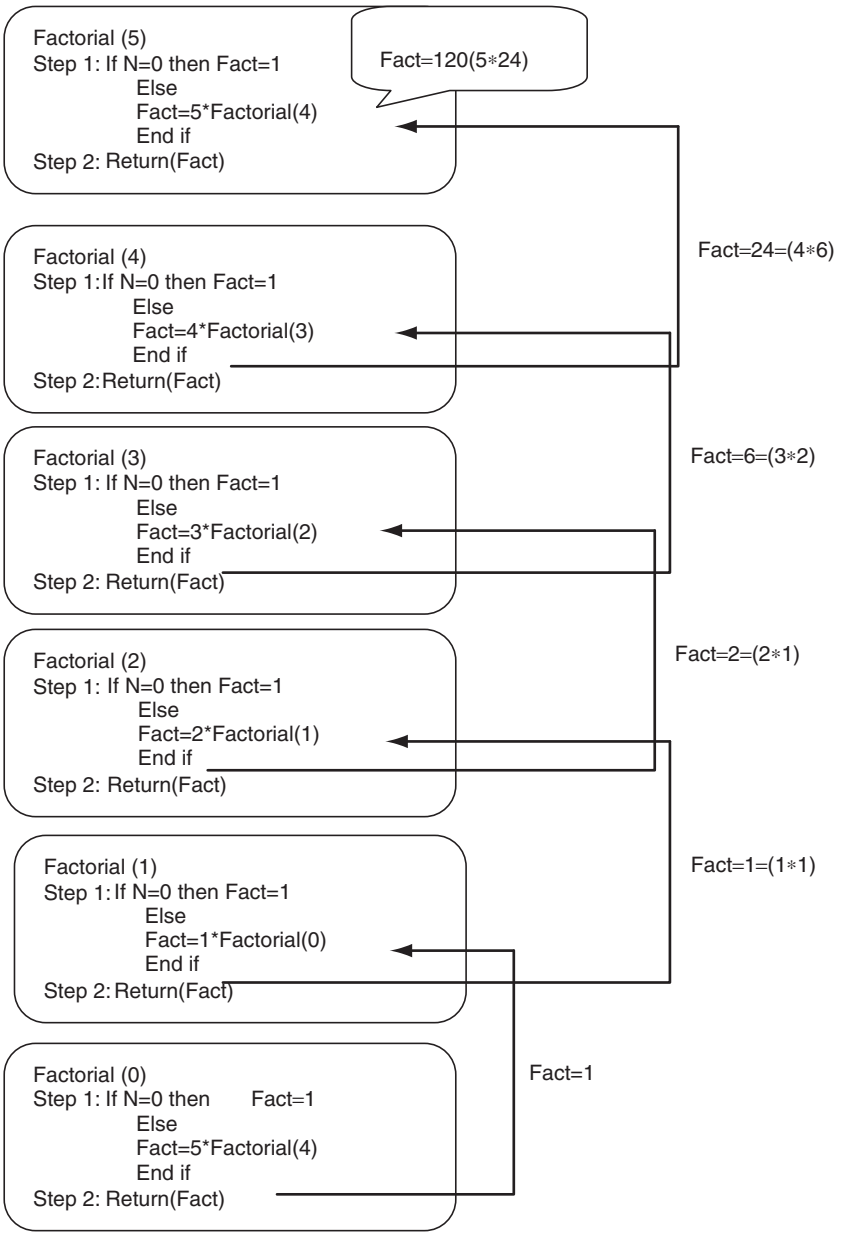


Figure 12 Use of recursion in finding the factorial of a number.

Algorithm 5 `Factorial(N)`

Here `Factorial` is a function to find out the factorial of a given number `N` stored in `Fact`.

1. `If N=0 then Fact=1 // Base condition`

```
    Else
     Fact = N * Factorial(N - 1)    //Leading towards
    the base condition
    End if
 2. Return(Fact)
```

Program 4 **A program in C++ to find the factorial of a given number using recursion.**

```
#include<iostream.h>
#include<conio.h>
  int factorial(int N)
  {
  int fact;
  if(N==0)
  fact=1;
  else
  fact=N*factorial(N-1);
  return(fact);
  }
void main()
{
clrscr();
int Num;
cout<<"Enter the number ";
cin>>Num;
cout<<"\nThe factorial of "<<factorial(Num);
getch();
}
```

Output

```
Enter the number 8
The factorial of 8 is 40320
```

Examples of Recursion: Tower of Hanoi

The Tower of Hanoi is a very interesting problem that can be efficiently solved by using recursion. In this, there are three pegs labeled X, Y and Z with several disks on the peg X. Let peg X be placed with *N* disks in the order of decreasing diameter, from largest disks at the bottom and the smallest on the top. We have to transfer all the *N* disks from peg X to peg Z using the peg Y as an intermediate peg with the following rules:

1. Only one disk from the top of any peg may be moved at a time to another peg.
2. A larger disk cannot be placed over a smaller disk at any point of time.

Let us represent the movement of topmost disk from peg A to peg B by A–>B where A and B may be any pegs. The recursive algorithm is given in Algorithm 6.

Algorithm 6 `TowerofHanoi(N, X, Y, Z)`

This algorithm is used to move the entire ($N>1$) disks from `peg` to `X peg` using `Y` as an intermediate `peg`.

1. `If =1N` then move the disk from `X>Z` and `return End if`
2.
   a. `Tower(N-1, X, Z, Y)`
      Move the top `N-1` disks from `peg` to `X peg` using `peg` as an intermediate `peg`
   b. `Tower(1, X, Y, >ZZ)` or `X-`
      Move the remaining top disk from `peg` to `X peg Z`
   c. `Tower(N-1, Y, X, Z)`
      Move the top `N-1` disks from `peg Y` to using `Z peg` as an intermediate `peg`
3. `Return`

In general this recursive solution requires $f(N) = 2^N - 1$ moves for N disks. Let us implement this algorithm for Tower of Hanoi with peg X having four disks.

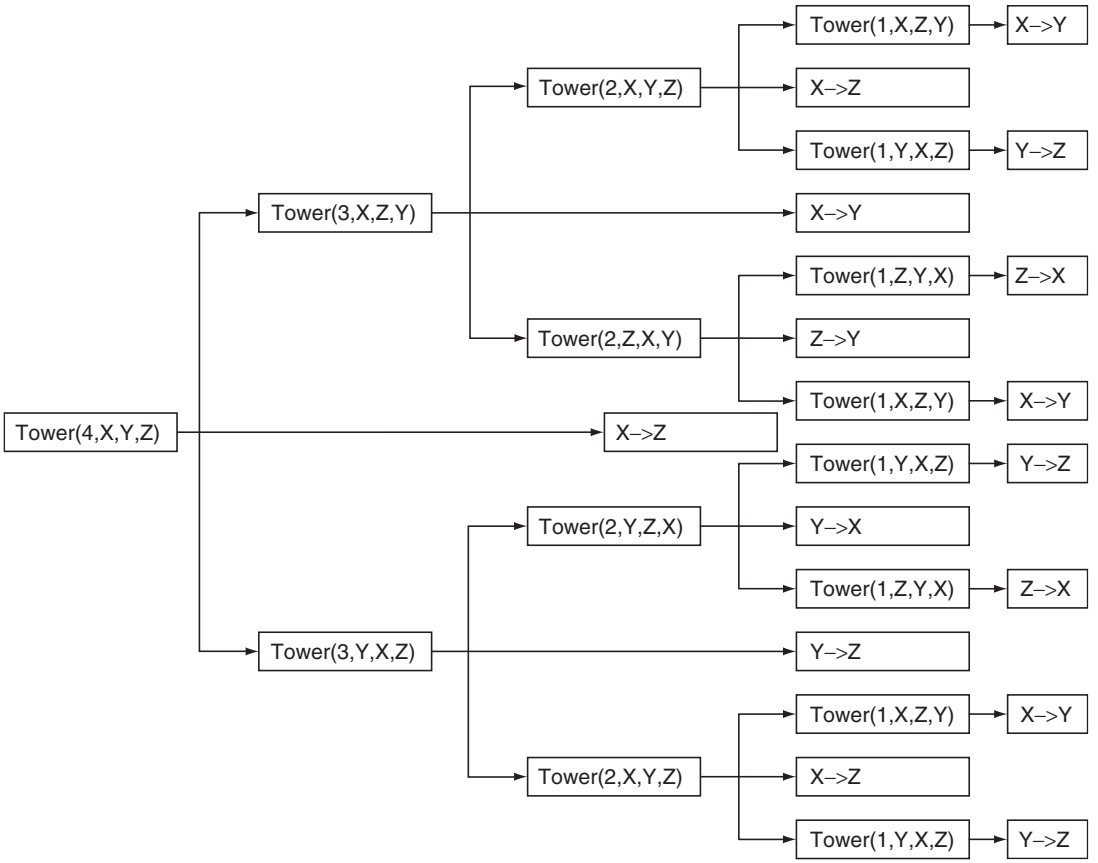


Figure 13 Implementation of Tower of Hanoi using recursion.

We have seen that the recursive solution for $N = 4$ disks consists of the following 15 moves:

X–>Y , X->Z , Y>Z , X->Y , Z>X, Z ->Y , X->Y , X->Z , Y>Z , Y>X, Z ->X, Y ->Z ,
X–>Y , X->Z , Y>Z

Recursion is very important concept in Computer Science. A recursive function needs more storage and more time. Many algorithms can be best described in terms of recursion but at the cost of space and time. Stack is used internally by the compiler when we implement the recursive function. Recursion is generally used for repetitive computation in which each action is defined in terms of the previous result. Recursion looks just like iteration, but this is not true. There are some differences between recursion and iterations as compiled in Table 3.

Table 3 Comparison of recursion and iteration

S. No.	*Recursion*	*Iteration*
1	When a function calls itself, it is called recursion.	In iteration, a set of statements is executed repeatedly until some specified condition is satisfied.
2	Recursion is based on the base condition; execution of statements moving towards the base condition.	Iteration is based on the initialization, termination condition, updation and execution.
3	Recursion is a top-down approach.	Iteration is bottom-up approach.
4	Recursion runs out of memory if base condition is not checked.	Iteration results in infinite loop if termination condition is not met.
5	Recursion takes more space to store new set of local variables.	Iteration does not take much time as compared to recursion.
6	Recursion is not efficient.	Iteration is efficient.
7	Every recursive algorithm can be converted into its iterative version.	Every iterative method cannot be converted into its recursive version.
8	For proper function recursion saves the return address.	Iterative method does not save the return address.
9	Recursion uses stack for temporary storage.	Stacks are not required in implementing iterative algorithms.

5.5 Multiple Stack

We have discussed the representation of a single stack in the memory. Now the question is: Is it possible to implement more than one stack in the memory? The answer is yes. Let us restrict ourselves to the array representation of the stack. In a single stack, array is restricted at one end and it keeps on growing at other end (Top of the stack). Now, let us restrict both the ends of the array in a manner that the array cannot grow below the starting index and after the end index. If this array is used to represent two stacks then these two stacks can only grow in the opposite direction as shown in Figure 14. The two ends are the two tops of two different stacks. If an array contains Max elements, then Stack-1 may have the starting value for the Top = –1 when it is empty. The starting value of the Top of Stack-2 may be Max [because array index starts from 0] when it is empty. The Top value of the stack is incremented when some item is inserted in Stack-1. The Top of the Stack-2 is decremented when some item is

inserted in Stack-2. The array index for Stack-1 starts from 0 and for Stack-2 from Max – 1. In this way we can represent two stacks in a single array. Note that

1. The overflow condition occurs when any of the stacks reaches the Top of other stack.
2. The underflow condition for Stack-1 is Top 1 = –1 and the underflow condition for Stack-2 is Top 2 = Max.

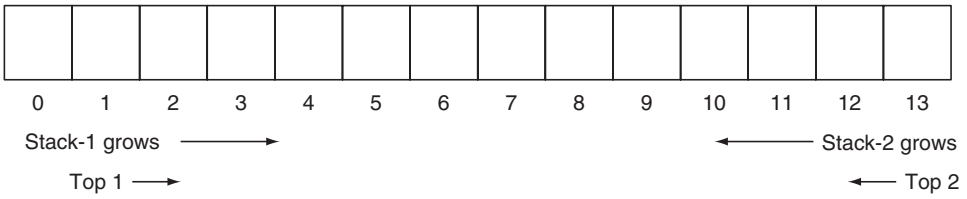


Figure 14 Representation of two stacks in an array.

Two stacks may also be implemented by dividing the array into two parts of fixed number of elements in which the Top of the Stack-1 may start from 0 and the Top of the Stack-2 may start from M as shown in Figure 15. In this figure, Stack-1 contains the elements of array index ranging from 0 to $M - 1$ and Stack-2 contains the elements of array index ranging from M to Max − 1. In this case overflow for Stack-1 occurs when Top − 1 becomes $M - 1$ and overflow for Stack-2 occurs when Top − 2 becomes Max − 1. The underflow condition for Stack-1 is Top 1 = −1 and for Stack-2 is Top 2 = $M - 1$.

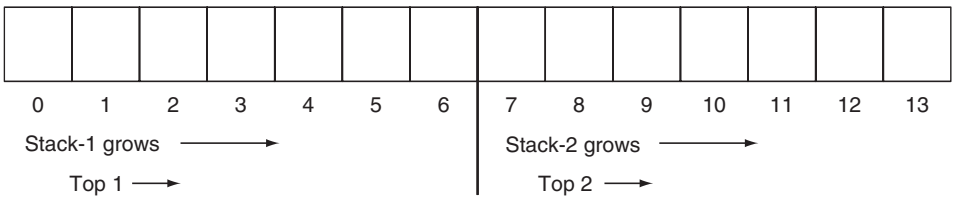


Figure 15 Representation of two stacks in an array.

This strategy may be extended to implement any number of stacks (obviously not more than the size of an array) using an array as shown in Figure 16. We can divide the array into N parts where the size of each part depends on the expected size of the stack. A stack becomes full when $T(i - 1) = B(i)$, that is, when Top of previous {$(i - 1)$th} stack becomes equal to bottom of the next (ith) stack.

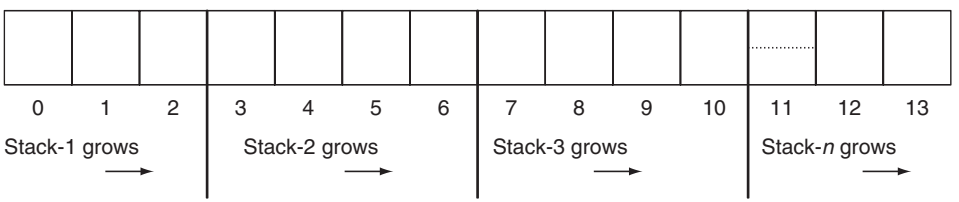


Figure 16 Representation of many stacks in an array.

5.6 Queue

Queue is a linear data structure in which elements can be inserted from one end (called the Rear end) and deleted from the other end (called the Front end). The elements of the *queue* are processed in the order in which they are inserted, so queue is basically useful for processing elements in their order of arrival. In other words, *queue* is based on the principle of first come first serve, that is, the order in which elements enter a queue is the order in which they leave. This is in contrast with the principle of stack, which is a LIFO list. For the reason stated above a *queue* is also called as first in first out (FIFO). Few examples of queue are: a line of people waiting for the reservation at the reservation counter; a line of people waiting for the bus at the bus stop; *queue* of jobs waiting for execution in the computer. As another example consider the reservation counter; a new person who comes takes his or her place at the end of the line and the person at the front of the line gets the reservation ticket first and leaves the *queue* first.

Queue finds many applications in Computer Science, particularly in system programming and application programming for buffering commands or data between processes and devices. The data or processes wait in the queue for resources. The use of queue is an integral part of the operating system where *queue* is basically used for process management and implementing the multiprogramming concepts.

5.7 Queue Representation and Implementation

A *queue* can be represented in two ways in a manner similar to the other data structures. These are:

1. Static or sequentially or array representation (implementation).
2. Dynamic or pointer or linked representation (implementation).

Static or Array Representation (Implementation)

As *queue* is a collection of homogenous elements like an array therefore it can be easily implemented by using an array. A *queue* is implemented with Max elements array named Queue with elements stored from queue[*i*] to queue[*j*] where Max is the maximum size of the array, that is, the maximum number of elements an array named Queue can hold and *i* and *j* are integers that hold the index number of the front and rear elements of the *queue*. Array implementation of a queue requires two pointer variables: Front containing the location of the front element of the queue and Rear containing the location of the rear element of the queue. In the rest of the chapter we will use F and R for Front and Rear, respectively.

The condition Front = Null or Front = –1 will indicate that the queue is empty as shown in Figure 17(a). We are not using Front = Rear = 0 because array index starts from 0. When the first element is inserted into the queue the value of Front and Rear becomes 0. The array implementation is very simple and efficient but it requires the ultimate size of the Queue to be declared beforehand, which results sometimes in wastage of memory and sometimes in crashing of the application. Once the size of the array is declared, the size cannot be varied during program execution in the static implementation because arrays allocate memory in contiguous location. One end of the array represents the Front and other end represents the Rear end of the Queue. The Queue usually grows at Rear end and shrinks from the Front end. Each time a new element is inserted into the Queue, the value of the Rear is incremented by one before placing the new element into the queue and the value of Front is incremented after deleting the element from the Queue accordingly to keep track of the current Rear and Front of the Queue. Different situations of Queue are shown in Figures 17(a)–(d).

Therefore, in the static implementation we must take care that the array must be large enough to hold all queue elements. However, implementation of queue using linear arrays is not as straightforward as that of stack.

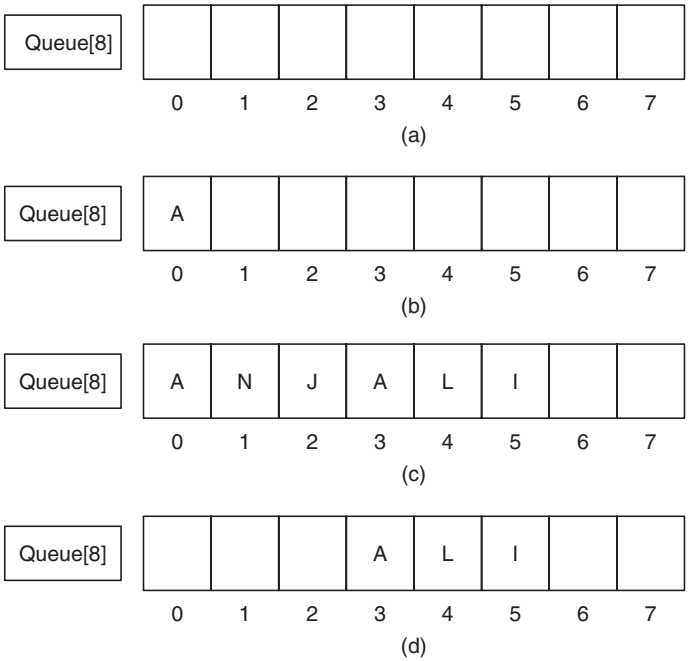


Figure 17 (a) Queue is empty so $F = R = -1$; (b) one element "A" is inserted into the Queue so $F = R = 0$; (c) some elements are inserted into the Queue so $F = 0$, $R = 5$; (d) other view of Queue where $F = 3$ and $R = 5$.

Dynamic or Pointer Representation (Implementation)

The advance declaration of Max size is not possible all the time. The size of the queue may increase or decrease according to the requirement of the application because fundamentally queue is a dynamic object whose size is constantly changing as the items are deleted and inserted. In the static implementation, once the size of the array is declared, it cannot be varied during program execution. The other representation of a queue is dynamic representation, which is also called as linked representation of queue. The linked representation commonly termed as linked queue is more suitable if an accurate estimate of the queue's size cannot be made in advance and the data elements are large records. A linked queue is a queue implemented as a linked list with two pointers variable, Front and Rear, pointing to the nodes at the front and rear of the queue. Linked queue is a collection of nodes where each node of the queue is divided into two parts as shown in Figure 18.

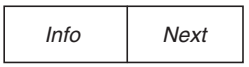


Figure 18 Node representation.

The first half (*info* field) of the node contains the element of the queue and the second half (*next* field) holds the pointer to the neighboring element in the queue. Two pointers are also required to maintain front and rear position of the queue. If the pointer Front is null then it shows that the queue is empty and can be indicated by Rear = Front = Null. If the *next* field of the last node contains Null then it

shows the end of the queue. The linked queue is not limited in capacity and therefore as many nodes as per the requirement may be inserted into the queue. Figure 19 shows the linked or dynamic representation of a queue. The implementation of linked queue is discussed in the next chapter. The comparison between stacks and queues is given in Table 4.

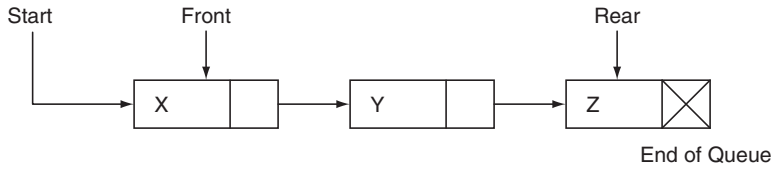


Figure 19 Dynamic representation of queue.

Table 4 Comparison between stacks and queues

S. No.	*Stacks*	*Queues*
1	Stacks can be implemented sequentially and dynamically.	Queues can also be implemented sequentially and dynamically.
2	Stacks are also called LIFO.	Queues are also called FIFO.
3	There is only one entry and exit called the Top in the stack	There is a separate entry called Rear and exit point called Front in the queue.
4	There are no different forms of stacks.	There are different types of queues such as circular, priority, deque, etc.

5.8 Queue Operations

The two basic operations that are performed on the queue are insertion and deletion. Queue implementation needs to keep track of the front and the rear of the queue whereas in the case of stack one only needs to worry about one end, the top.

Insertion in the Queue

We know that insertion is only possible from Rear end in the queue, so when we want to insert the element into the queue, its Rear is incremented and the new item is inserted into the new location. After the element is inserted into the queue, the new element becomes the rear. To perform the insertion operation we have to check the current status of the available memory space as we do in the case of stack also. If we reach the last location of the queue then we cannot insert the element and the condition is called overflow. Algorithm 7 is used to insert an "Element" in to the queue, where QUEUE is an array of MAX size and FRONT and REAR are the two ends of the QUEUE

Algorithm 7 QINSERT(QUEUE, MAX, FRONT, REAR, EL

1. Check for Overflow
   If REAR = MAX - 1 or FRONT = REAR + 1 then Print "Overflo
   and Return
   End if

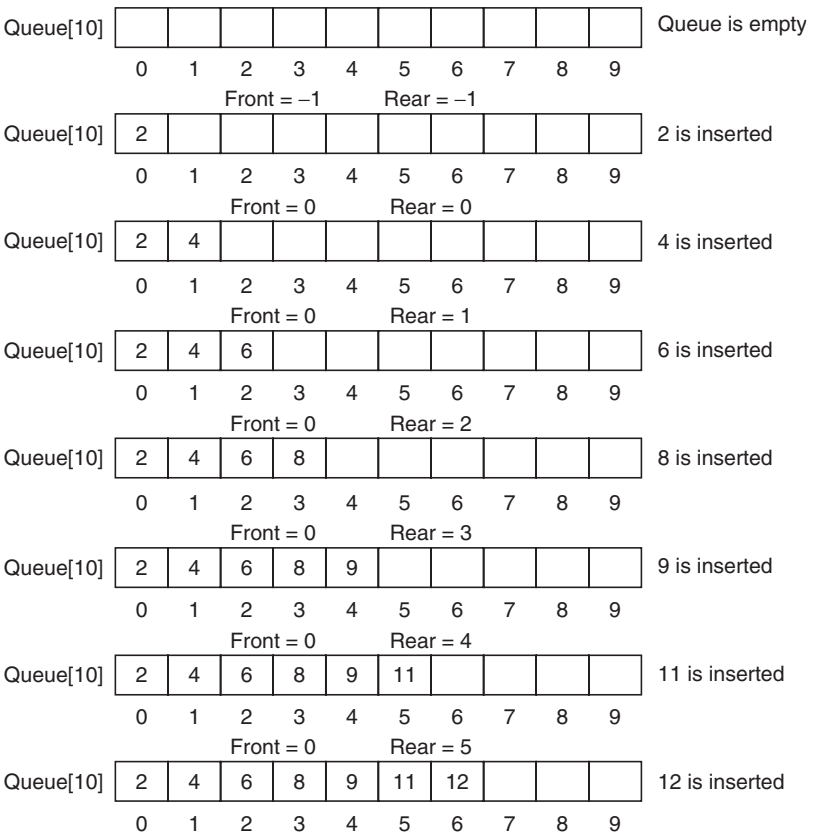
```
2. If FRONT = 0 and REAR = MAX-1 then Set FRONT = 0 and REAR = 0
   Else if REAR = MAX-1 then Set REAR = 0
   Else Set REAR = REAR + 1 //Increment the REAR by
   End if
   End if
3. QUEUE[REAR] = ELEMENT //Insert the Element a
   tion
4. Return
```

Problem 4

Insert 2, 4, 6, 8, 9, 11, 12, 14, 15 and 16 into a queue of 10 elements.

Solution

Figure 20 explains the concept diagrammatically.



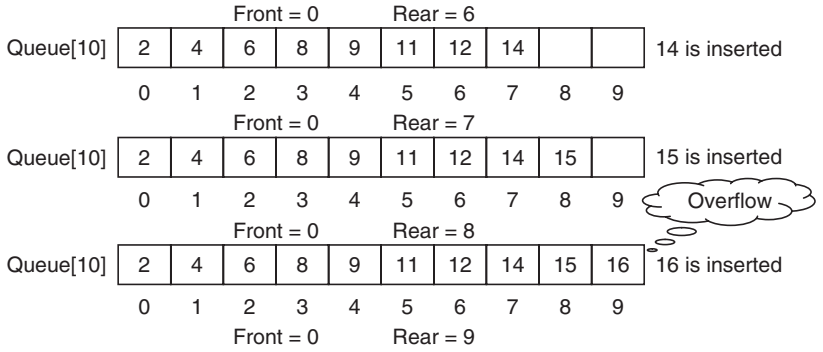


Figure 20 Solution to Problem 4.

Deletion from a Queue (Using Array)

The elements from the queue are deleted only from the front end. After the deletion of an element the front is incremented by one, the deleted element of the queue is returned to the user and removed from the queue. The delete operation cannot be attempted in the empty queue. The queue is called in an underflow state when we try to delete an element from the empty queue. This algorithm (Algorithm 8) is used to delete an "Element" from the Queue, where QUEUE is an array of MAX size and FRONT and REAR are the two ends of the QUEUE.

Algorithm 8 QDELETE(QUEUE, MAX, FRONT, REAR, EL

1. Check for Underflow
   If FRONT = NULL then Print "Underflow" and Retu
2. ELEMENT = QUEUE[FRONT] //Delete the ele
3. If FRONT = REAR then set FRONT = REAR = NULL
   Else
   FRONT = FRONT + 1 //Increment the FRONT by On
   End if
4. Return

Problem 5

Perform the deletion operation on the queue given

2	4	6	8	9	11	12	14	15	16

Solution

Figure 21 illustrates the solution diagrammatically.

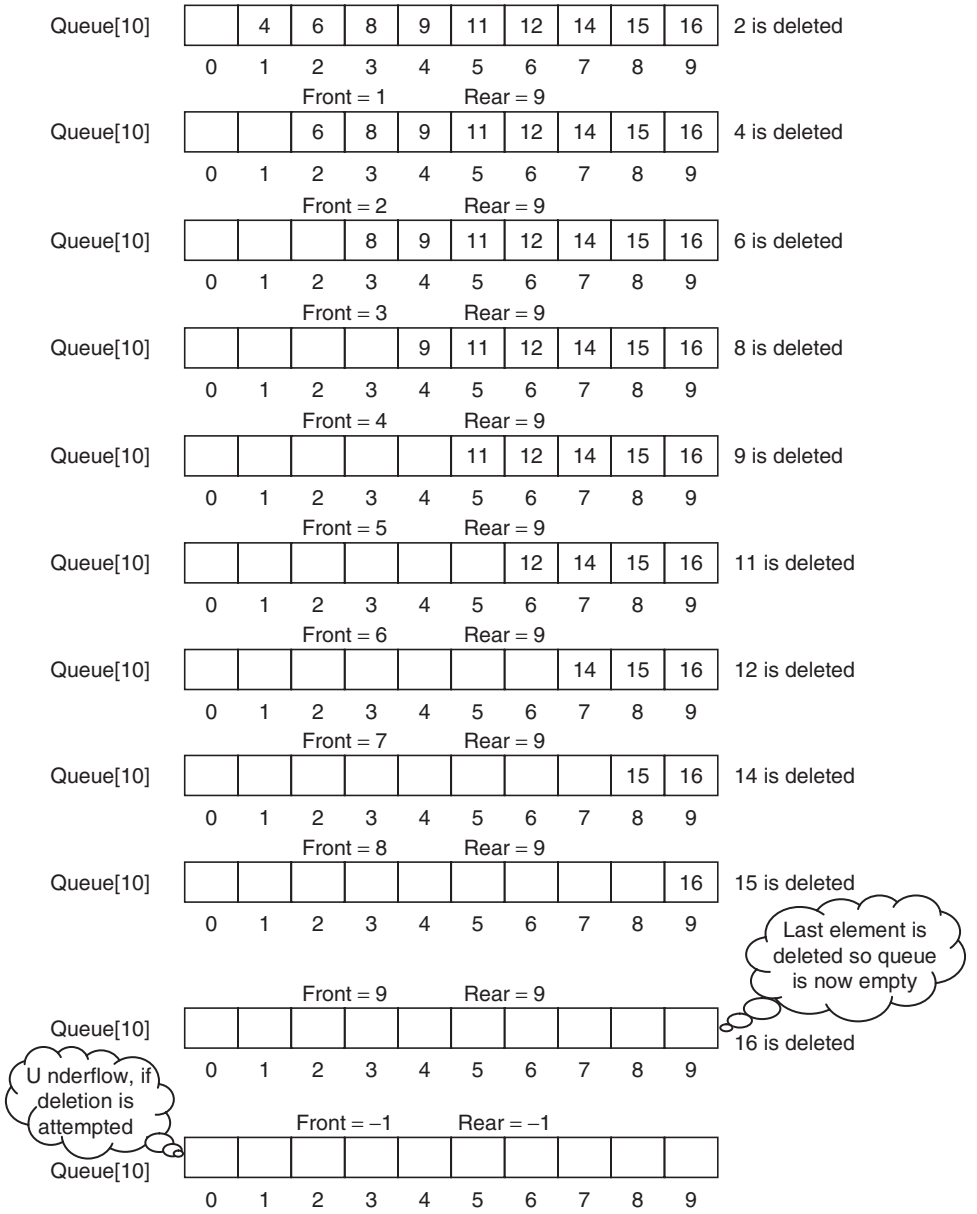


Figure 21 Solution to Problem 5.

Program 5 **A program in C++ to implement insertion and deletion operations on a simple queue.**

```
#include<iostream.h>
#include<conio.h>
#include<process.h> // to use cerr
#define SIZE 3
class FIFO
{
int QUEUE[SIZE];
int FRONT;
int REAR;
public:
FIFO()
{
FRONT=-1;
REAR=-1;
}
void Insert()
{
int NUM;
if(REAR==SIZE-1)
{
cerr<<"QUEUE is Full-Overflow";
exit;
}
else
{
if(FRONT==-1)
{
FRONT=0;
REAR=0;
}
else
{
REAR=REAR+1;
}
cout<<"Enter the number to Insert";
cin>>NUM;
QUEUE[REAR]=NUM;
clrscr();
display();
}
}
void qdelete()
```

```
{
if ( FRONT == -1 )
{
cout << " Queue is empty - Underflow "
exit ;
}
else
{
cout << " ITEM deleted from QUEUE " << QUEUE [ FRONT ] ;
if ( FRONT == REAR )
{
REAR = -1 ;
FRONT = -1 ;
}
else
{
FRONT = FRONT + 1;
display ( ) ;
}
}
}
void display ( )
{
cout << endl ;
cout << " QUEUE contents " << endl ;"
for ( int i = FRONT ; i <= REAR ; i++)
cout << QUEUE [ i ] << " " ;
}
} ;
void main ( )
{
clrscr ( ) ;
FIFO QUE;
int ch;
while ( ch != 0 )
{
cout << " \n ** QUEUE Operations Menu ** "
cout << " 1. INSERT " << endl ;
cout << " 2. DELETE " << endl ;
cout << " Enter the choice ( 0 for exit ) : " ;
cin >> ch ;
switch ( ch )
{
case 1 :
QUE . Insert ( ) ;
```

```
break;
case 2:
clrscr();
QUE.qdelete();
break;
default:
cout<<"illegal option";
}
}
getch();
}
```

Output

```
QUEUE contains
11    12    13
** QUEUE Operation Menu **
1. INSERT
2. DELETE
Enter the choice (0 for exit):1
QUEUE is Full-Overflow
** QUEUE Operation Menu **
1. INSERT
2. DELETE
Enter the choice (0 for exit):2
ITEM deleted is 11
QUEUE contains
12    13
** QUEUE Operation Menu **
1. INSERT
2. DELETE
Enter the choice(0 for exit):0
illegal option
```

Program 6 **A program in C++ to perform insertion and deletion operation on queue using templates.**

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
#define MAX 3
template<class T>
class queue
```

```
{
T  Q[MAX];
T  Front;
T  Rear;
T  Item;
public:
queue()
{
Front=-1;//Queue is empty
}
void Insert()
{
if(Rear==MAX-1)  //Check for overflow
{
cout<<"\n***Queue is full-Overflow!***\n";
}
else
{
if(Front==-1)  //If Queue is empty
{
Front=0;
Rear=0;
}
else
{
Rear=Rear+1;
}
cout<<"\nEnter the character to Insert";
cin>>Item;
Q[Rear]=Item;//Place the Item to the Rear of
cout<<"\nYou have Inserted an Item to the qu
clrscr();
display();                    //To display the co
}
}
void Delete()
{
if(Front==-1)
{
cout<<"\n***Queue is empty-Underflow!***\n";
}
else
{
Item=Q[Front];   //Keep the Front Value to I
```

```
if(Rear==Front)
{
Front=-1;
Rear=-1;
}
else
{
Front=Front+1;  //Increment Front by 1
}
cout<<"\n"<<Item<<" is deleted from the queue!\n";
}
display();
}
void display()
{
cout<<endl;
cout<<"\nNow Maximum elements you can insert="<<(MAX-Rear+Front)-
1);
if(Front==-1)
{
cout<<"\nThe queue has no data!\n";
}
else
{
cout<<"\nQueue contains: ";
for(int i=Front;i<=Rear;i++)
cout<<Q[i]<<" ";
}
}
};
void main()
{
clrscr();
queue obj;//Create object 'obj' of class
Template
int ch;
do
{
cout<<"\n***Menu***";
cout<<"\n(1) Insert Item";
cout<<"\n(2) Delete Item";
cout<<"\nEnter your Choice (0 to exit)";
cin>>ch;
switch(ch)
```

```
{
case 1:
obj.Insert();
break;
case 2:
obj.Delete();
break;
}
}while(0ch!=0);
getch();
}
```

Output

```
Now Maximum elements you can insert are
Queue contains
A     B     C
***Menu***
(1) Insert Item
(2) Delete Item
Enter your Choice (0 to exit) 1
***Queue is full-Overflow!***
***Menu***
(1) Insert Item
(2) Delete Item
Enter your Choice (0 to exit) 2
  A is deleted from the queue!
Now Maximum elements you can insert are
Queue contains
B     C
***Menu***
(1) Insert Item
(2) Delete Item
Enter your Choice( 0 to exit) 0
```

5.9 Types of Queue

We have discussed the simple queue in the previous section. In a simple queue, we cannot insert elements into the queue when we reach to the end of queue in spite of the free space made available in the memory by the elements deleted from the queue. Another drawback is that the insertion of elements is restricted to the rear end and deletion of elements is restricted to the front end only. There are different types of queue which do not have these drawbacks. Such queues are

1. circular queue;
2. deque (double ended queue);
3. priority queue.

We will discuss each of them in this section.

Circular Queue

The circular queue is used to remove the drawback of the simple queue in which we cannot insert the elements even if memory space is available due to the deletion of elements from the queue. In this queue, the last element follows the first element of the queue as shown in Figure 22. The last element I at CSTACK[8] follows the first element A at CSTACK[0] where CSTACK is a circular array of nine elements.

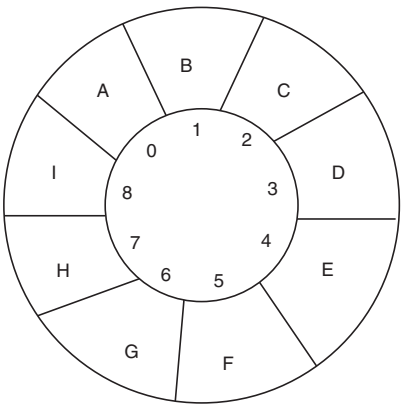


Figure 22 Circular queue.

The circular queue follows the same rule of the simple queue for insertion and deletion of elements except that after the end of queue it reaches the start position of the queue.

Insertion and Deletion in the Circular Queue

Insertion and deletion from the circular queue are performed in a manner similar to that in simple queue except for the condition of overflow. The condition for overflow is

$$\text{Front} = (\text{Rear} + 1)\%\text{Max}$$

and the for underflow it remains the same as in simple queue, that is,

$$\text{Front} = -1$$

The rear is incremented by (Rear + 1)%Max in insertion operation and front is increased by (Front + 1)%Max in deletion operations. The algorithm for insertion of elements in the circular queue is given in Algorithm 9 and Figure 23 shows the insertion process in the circular queue with example.

Algorithm 9 CQINSERT (QUEUE, MAX, FRONT, REAR,

```
1. Check for Overflow
       If FRONT = (REAR + 1) % MAX then Print "Overflow
       Return
          End if
2. If FRONT = -1 then Set FRONT = REAR = 0
          Else Set REAR = (REAR + 1) % MAX //Increment the RE
       One
          End if
3. QUEUE [REAR] = ELEMENT //Insert the Element at ne
4. Return
```

Problem 6

Perform the following operations in sequence on the circular queue:
(*a*) *Insert A, B, C, D, E, F, G, H and I*
(*b*) *Delete A and B*
(*c*) *Insert M, N and P*

Solution
See Figure 23.

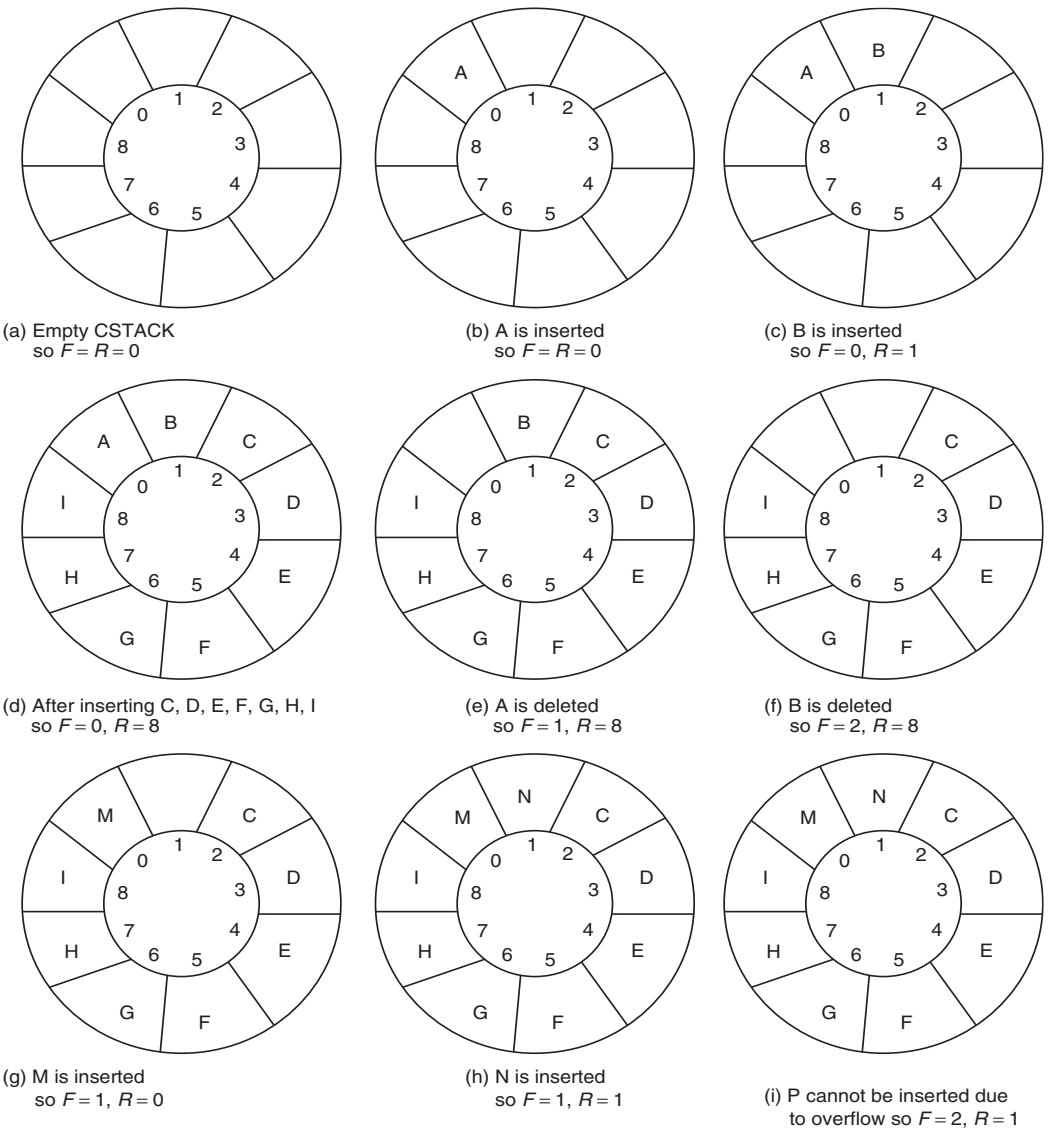


Figure 23 Insertion and deletion in the circular queue.

We can see from Figure 23(a) that overflow condition occurs when

$$\text{Front} = (\text{Rear} + 1)\%\text{Max}$$

where F is used for Front, R is used for Rear and Max is the size of the circular array.

Algorithm 10 is the algorithm for deletion of elements from the circular queue and Figure 24 shows the deletion process in the circular queue with example.

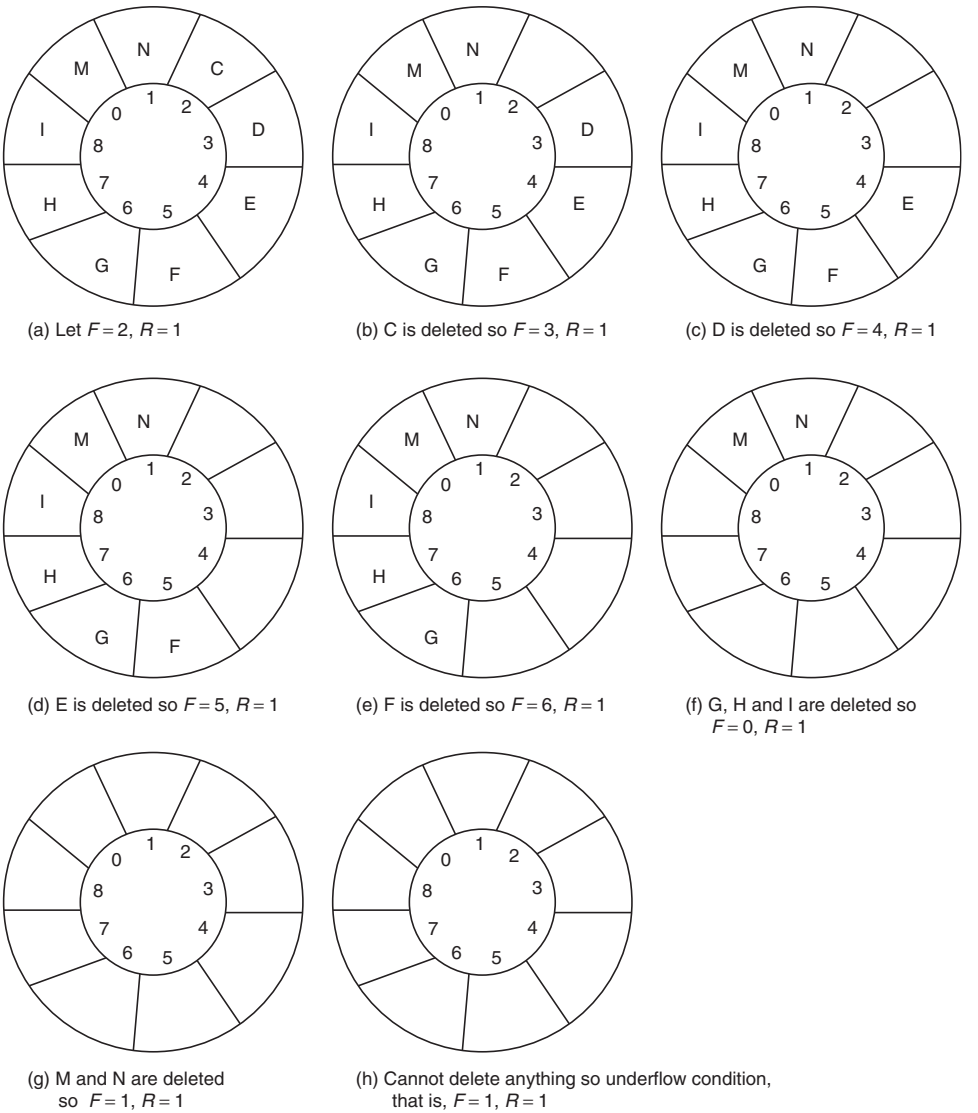


Figure 24 Deletion from the circular queue.

Algorithm 10 CQDELETE(QUEUE, MAX, FRONT, REAR,

1. Check for Overflow
   If FRONT Then Print "Underflow" and Re
   End if
2. If ELEMENT QUEUE [FRONT]
   If (FRONT REAR) the set REAR FRONT
   Else FRONT (FRONT 1) % MAX // Incremen
   FRONT by One
   End if
3. Return

Program 7 **A program in C++ to implement insertion and deletion operation in circular queue.**

```
#include<process.h>
#include<iostream.h>
#include<conio.h>
#define max 3
 class Queue
 {
 private:
 int queue[max];
 int rear;
 int front;
 public:
 Queue()
 {
 rear=-1;
 front=-1;
 }
 int Insert()
 {
 int item;
 if(front==(rear+1)%max)
 {
 cout<<"\n **Overflow : Queue is full **";
 return 0;
 }
 else
 {
 if(front==-1)
 {
 rear=0;
 front=0;
 }
```

```
else
  {
  rear=(rear+1) % max;
  }
  cout<<"\n Enter the item to insert into the
  cin>>item;
  queue[rear]=item;
  cout<<item<<"  is inserted into the Queue";
  }
  }
  void show();
  int Delete();
  };
  int Queue::Delete()
  {
  if (front==-1)
  {
  cout<<"\n** Underflow : Queue is empty **";
  return 0;
  }
  else
  {
  cout<<"\n"<<"queue[front]is deleted from the Queu
  if (front==rear)
  {
  front=-1;
  rear=-1;
  }
  else
  {
  front=(front+1) % max;
  }
  }
  }
  void Queue::show()
  {
  int i;
  if (front==-1)
  cout<<"\n*** The Queue is empty ***";
  else
{
  if (rear<front)
  {
  for(i=front;i<max;i++)
  cout<<queue[i]  ";
  for(i=0;i<=rear;i++)
```

```
    cout<<queue[i]<<" ";
    }
    else
    {
    for(i=ftr+1;i<=rear;i++)
    cout<<queue[i]<<" ";
    }
    }
    }
void main()
{
    Queue obj;
     char ch;
    do
    {
    clrscr();
    cout<<"\n\n\t\t*** Implementation of Circular Queue
    cout<<"\n\t\t I: To Insert an item";
    cout<<"\n\t\t D: To Delete an item";
    cout<<"\n\t\t S: To show the item in the Circular
    cout<<"\n\t\t E: To Exit";
    cout<<"\n\t\t Enter your Choice : ";
    ch=getche();
    switch(ch)
    {
    case "i":
    case "I":
            obj.Insert();
            break;
    case "d":
    case "D":
            obj.Delete();
            break;
    case "s":
    case "S":
            obj.show();
            break;
    case "e":
    case "E":
    exit(0);
    }
    cout<<"\n\n\t\t Press Y to continue";
    ch=getch();
    }while(ch=='y'||ch=='Y');
    getch();
    }
```

Output

```
*** Implementation of Circular Queue ***
  I: To Insert an item
  D: To Delete an item
  S: To show the item in the Circular Queue
  E: To Exit
Enter your Choice : i
  Enter the item to insert into the Queue :
2   is inserted into the Queue
Press Y to continue
::::::::::
::::::::::
*** Implementation of Circular Queue ***
  I: To Insert an item
  D: To Delete an item
  S: To show the item in the Circular Queue
  E: To Exit
Enter your Choice : i
  ** Overflow : Queue is full **
Press Y to continue
```

Deq ue

Deque is a double ended queue in which insertion and deletion can be performed from both ends, rear and front. It follows the same strategy as simple queue when insertion is done from the rear end and deletion is done from the front end. The rear is incremented by 1 before inserting the element. After this, the element is inserted into the new location. The new inserted element becomes the rear element of the deque. Similarly, when the element is deleted from the front end, the front is incremented by 1 and this deleted element is returned to the user.

The interesting point to note is that the elements can be inserted from the front end in deque. When elements are inserted from the front end, the front is decremented by 1 before insertion of element. The element is then inserted into the new front location. If front of deque is 0 then overflow occurs and we cannot insert more items into the deque.

Similarly, elements can be deleted from the rear end also in the deque. When elements are deleted from the rear end, the rear is decremented by 1 after the deletion. The deleted element is returned to the user. We can see that this is the reversal of the insertion and deletion performed on the simple queue. When an element is inserted in a simple queue, the rear is incremented by 1 but when it is inserted from the front in the deque it is decremented (reverse) by 1. Similarly, when an element is deleted from the front in a simple queue, the front is incremented by 1 but when deletion is performed from the rear end in the deque the rear in decremented by 1.

There are various types of deques, namely,

1. **Input restricted:** This is the type of deque in which insertion is restricted to one end (normally rear end) and deletion can be performed from both ends, front and rear.
2. **Output restricted:** This is the type of deque in which deletion is restricted to one end (normally front end) and insertion can be performed from both ends, front and rear.

3. **Circular deque:** This deque is also implemented in a circular form so that last element follows the first element in the queue. The term circular comes from the fact that we assume that deque[0] comes after deque[$N-1$] in the array. The condition Front = –1 is used to indicate an empty deque.

The input restricted and output restricted deques are considered to be the intermediate between a deque and a queue.

Problem 7

Perform the following operations on the deque:

(***a***) *Insert A B C D and E from the rear end.*
(***b***) *Delete A B C from the front end.*
(***c***) *Insert F, G and H from the front end.*
(***d***) *Delete D E F from the rear end.*

Solution

Let deque be an array of 14 elements. Figure 25 diagramatically gives the solution.

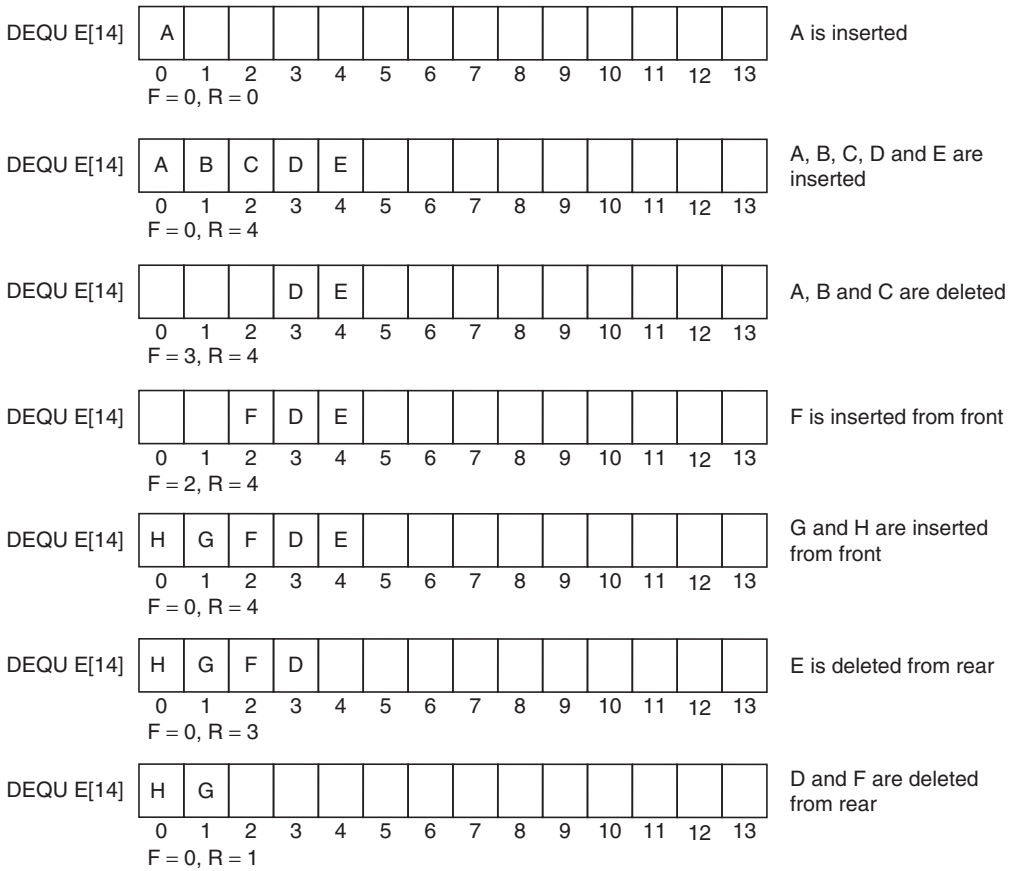


Figure 25 Solution to Problem 7.

Program 8 **A menu- driven program in C++ to implement insertion and deletion operations in a double ended queue.**

```
#include<process.h>
#include<iostream.h>
#include<conio.h>
#define max 5
 class DeQue
 {
 private:
 int queue[max];
 int rear;
 int front;
 public:
 DeQue()
 {
 rear=-1;
 front=-1;
 }
 int InsertR();
 int InsertF();
 int DeleteR();
 int DeleteF();
 void show();
 };
 int DeQue::InsertR()
 {
 int item;
 if(rear>=max-1)
 {
 cout<<"\n **Overflow : Queue is full **";
 return 0;
 }
 else
 {
 if(front==-1)
 {
 rear=0;
 front=0;
 }
 else
 {
 rear=rear+1;
 }
```

```
cout<<"\n Enter the item to insert into the
cin>>item;
queue[rear]=item;
cout<<item<<" is inserted into the Queue";
}
}
int DeQue::InsertF()
{
int item;
if(front==0)
{
cout<<"\n ** Overflow : Queue is full **";
return 0;
}
else
{
front=front-1;
}
cout<<"\n Enter the item to insert into the
cin>>item;
queue[front]=item;
cout<<item<<" is inserted into the Queue";
 }
void DeQue::show()
{
int i;
if(front==0)
{
cout<<"\n*** The Queue is empty ***";
cout<<"Front="<<front<<"Rear="<<rear;
}
else
{
for(i=front;i<=rear;i++)
cout<<queue[i]<<" ";
 cout<<"\nFront="<<front<<"Rear="<<rear;
}
}
int DeQue::DeleteF()
{
if(front==0)
{
cout<<"\n** Underflow : Queue is empty **";
return 0;
}
```

```
    else
    {
    cout<<"\n<<"queue[front]is deleted from the Queu
    if (front==rear)
    {
    front=-1;
    rear=-1;
    }
    else
    {
    front=front+1;
    }
    }
    }
    int DeQue::DeleteR()
    {
    if (rear==-1)
    {
    cout<<"\n** Underflow : Queue is empty **";
    return 0;
    }
    else
    {
    cout<<"\n<<"queue[rear]is deleted from the Queu
    if (front==rear)
    {
    front=-1;
    rear=-1;
    }
    else
    {
    rear=rear-1;
    }
    }
    }
  void main()
  {
    DeQue obj;
     int ch;
    char ch1;
    do
    {
    clrscr();
    obj.show();
  cout<<"\n*** Implementation of Circular Queue
```

```
  cout<< " 1: To Insert from rear end "
  cout<< " 2: To Delete from rear end "
  cout<< " 3: To Insert from front end "
  cout<< " 4: To Delete from front end "
  cout<< " 5: To show the item in the Circular
  cout<< " 0: To Exit";
  cout<<"Enter your Choice : ";
  cin>>ch;
  switch(ch)
  {
  case 1:
    obj.InsertR();
    break;
  case 2:
    obj.DeleteR();
    break;
   case 3:
    obj.InsertF();
    break;
  case 4:
    obj.DeleteF();
    break;
  case 5:
    obj.show();
    break;
  case 0:
  exit(0);
  }
  cout<<"\nPress Y to continue";
  ch=getch();
  }while(ch=='y'||ch=='Y');
  getch();
  }
```

Output

```
DeQue Contains 3 4
Front 1 Rear 2
*** Implementation of Circular Queue ***
 1: To Insert from rear end
 2: To Delete from rear end
 3: To Insert from front end
 4: To Delete from front end
 5: To show the item in the Circular Queu
 0: To Exit
Enter your Choice: 0
```

Priority Queue

A priority queue is a special queue in which insertion and deletion of elements are performed according to priority assigned to them. It does not follow the FIFO principle as in other queues. The elements of high priorities are processed before the elements of lower priorities. If the elements have same priorities then they are processed in the order in which they were inserted in the queue. Priority queue is used in operating system to implement time sharing property. Insertion of an element in the priority queue is more difficult than deletion of an element because we have to find the correct location according to the priority of the element to insert it in the priority queue. Since highest priority element is in the front of the queue, therefore it is easy to delete. Priority queues can be represented in two ways:

1. array or sequential representation;
2. dynamic or linked representation.

In the priority queue, each node is divided into three parts: The first part contains the information, second part contains the priority number and third part contains pointer to the neighboring node.

Multiqueue

We have discussed the representation of a single queue in the memory. Now the question is: "Is it possible to implement more than one queue in the memory?" The answer is yes. Let us restrict ourselves to the array representation of a queue. In the single queue, elements are inserted from the rear end and deleted from the front end. Now consider an array in which two queues are represented, growing in opposite direction as shown in Figure 26. The two ends are the front of the two different queues. If an array contains Max elements then Queue-1 may have the starting value for the Front = –1 when it is empty. The starting value of the front of Queue-2 may be Max (since array index starts from 0) when it is empty. Queue-1 grows in the opposite direction of Queue-2. The rear and front values of the Queue-1 are incremented when some item is inserted or deleted, respectively into Queue-1. The rear and front values of the Queue-2 are decremented when some item is inserted or deleted, respectively into Queue-2. The rear and front values of Queue-2 are decremented when some item is inserted or deleted, respectively, into Queue-2. The array indices for Queue-1 start from 0 and for Queue-2 from Max – 1. In this way we can represent two queues in a single array. The overflow condition occurs when any of the stacks reaches to the rear of other queue. The underflow condition for Queue-1 is Front = –1 and the underflow condition for Queue-2 is Front = Max.

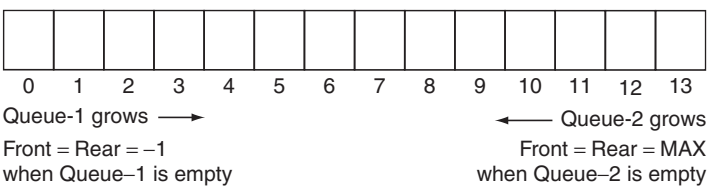


Figure 26 Two queues in an array.

Two queues may also be implemented by dividing the array into two parts of fixed number of elements in which the front and rear of Queue-1 may start from 0 and the front and rear of Queue-2 may start from M as shown in Figure 27, where Queue-1 contains elements of array index ranging from 0 to $M - 1$ and Queue-2 contains elements of array index ranging from M to Max – 1. In this case overflow for Queue-1 occurs when rear-1 becomes $M - 1$ and the overflow for Queue-2 occurs when rear-2 becomes Max – 1. The underflow condition for Queue-1 is Front = –1 and for Queue-2 is Front = $M - 1$.

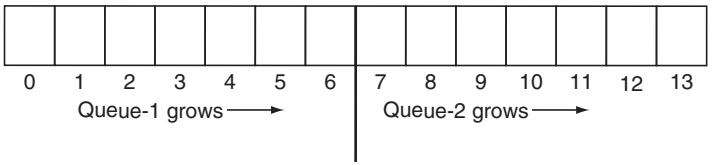


Figure 27 Two queues in an array.

This strategy may be extended to implement any number of queues (obviously not more than the size of an array) using an array as shown in Figure 28. We can divide the array into N parts where the size of each part depends on the expected size of the queue. A queue becomes full when rear$(i-1)$ = front(i) when rear of previous $\{(i-1)$th$\}$ queue becomes equal to the front of the next (ith) queue.

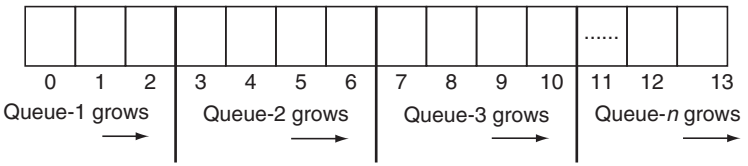


Figure 28 Many queues in an array.

Solved Examples

Example 1 *Write a program in C to implement insertion and deletion operations on a circular queue.*

Solution

```
#include<process.h>
#include<stdio.h>
#include<conio.h>
#define max 3
  int queue[max];
  int rear;
  int front=-1;
  int Insert()
  {
  int item;
  if (front==(rear+1) % max)
  {
  printf("\n ** Overflow : Queue is full *
  return 0;
  }
else
  {
  if (front==-1)
```

```
{
rear=0;
front=0;
}
else
{
rear=(rear+1)%max;
}
printf("\n Enter the item to insert into t
scanf("%d",&item);
queue[rear]=item;
printf("%d  is inserted into the Queue",it
}
}
Delete()
{
if(front==-1)
{
printf("\n** Underflow : Queue is empty **
return 0;
}
else
{
printf("\n %d  is deleted from the Queue",
if(front==rear)
{
front=-1;
rear=-1;
}
else
{
front=(front+1)%max;
}
}
}
void show()
{
int i;
if(front==-1)
printf("\n*** The Queue is empty ***");
else
{
if(rear<front)
{
for(i=front;i<max;i++)
printf("\n%d  ",queue[i]);
```

```
    for(i=0;i<=rear;i++)
    printf("%d  ",queue[i]);
    }
    else
    {
    for(i=front;i<=rear;i++)
    printf("\n%d  ",queue[i]);
    }
    }
  void main()
  {
    char ch;
    do
    {
    clrscr();
    printf("*** Implementation of Circular Qu
    printf("  I: To Insert an item \n");
    printf("  D: To Delete an item \n");
    printf("  S: To show the item in the Circu
    printf("  E: To Exit\n");
    printf("Enter your Choice : ");
    ch=getche();
    switch(ch)
    {
    case 'i':
    case 'I':
      Insert();
      break;
    case 'd':
    case 'D':
      Delete();
      break;
    case 's':
    case 'S':
      show();
      break;
    case 'e':
    case 'E':
    exit(0);
    }
    printf("\nPress Y to continue");
    ch=getch();
    }while(ch=='y' || ch=='Y');
    getch();
    }
```

Example 2 *Write a program in C to convert the infix expression into postfix expression.*

Solution

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>
#define MAX 10
static char infix[MAX], postfix[MAX];
static int top;
void convert(void);
void push(char);
char pop(void);
int priority(char);
void main()
{
      char ch;
      do
      {
    top=-1;
    clrscr();
    printf("\nEnter an infix expression\n");
    fflush(stdin);
    gets(infix);
    convert();
    printf("\nInfix expression is %s",infix);
    printf("\nPostfix expression is %s",postfix);
    printf("\nDo you wish to continue\n");
    ch=getche();
      }while(ch=='y'||ch=='Y');
}
void convert(void)
{
int iindex,pindex,length;
char next;
iind=0;pind=0;
leng=strlen(infix);
while(iind<length)
{
if(infix[iind]=='(')
push(infix[iind]);
else if(infix[iind]==')')
while((next=pop())!='(')
```

```
postfix[p++]=index++;
else if(isalpha(infix[iindex]) ||isdigit(
postfix[p++]=infix[iindex];
else
{
if(infix[i==i=='+'||infix[i==i=='
'||infix[i==i=='*'||infix[i==i=='/']
'||infix[i==i=='^'||infix[i==i=='^']
{
while(top>=0 && priority(infix[iindex])<=priority(stack[to
postfix[p++]=pop();
}
push(infix[iindex]);
}
iindex++;
}
while(top>=0)
postfix[p++]=pop();
postfix[p++]='\0';
}
void push(char sym)
{
    if(top==MAX-1)
    {
   printf("\nStack is full\n");
   exit(0);
    }
    else
    stack[++top]=sym;
}
char pop(void)
{
    if(top==-1)
    {
    printf("\nStack is empty\n");
    exit(0);
    }
    else
    return(stack[top--]);
}
int priority(char sym)
{
     switch(sym)
```

```
        {
        case '(':
            return(0);
        case '+':
        case '-':
            return(1);
        case '*':
        case '/':
        case '%':
            return(2);
        case '^':
            return (3);
        default :
            return(9);
        }
}
```

Output

```
Enter an infix expression
2+(4-5)/6
infix=2+(4-5)/6
postfix=245-+6/
Do you wish to continue: n
```

Example 3 *Write a program in C to implement insertion and deletion operations on a circular double ended queue.*

Solution

```
#include<process.h>
#include<stdio.h>
#include<conio.h>
#define max 5
  int queue[max];
  int rear=-1;
  int front=-1;
  int InsertR();
  int InsertF();
  int DeleteR();
  int DeleteF();
  void show();
  int InsertR()
```

```
{
int item;
if (front==(rear+1) % max)
{
printf("\n ** Overflow : Queue is full *
return 0;
}
else
{
if (front==-1)
{
rear=0;
front=0;
}
else
{
rear=(rear+1) % max;
}
printf("\n Enter the item to insert into
scanf("%d", &item);
queue[rear]=item;
printf("%d is inserted into the Queue",
}
}
int InsertF()
{
int item;
if (front==(rear+1) % max)
{
printf("\n ** Overflow : Queue is full *
return 0;
}
else
{
   if (front==-1)
   {
   front=0;
   rear=0;
   }
   else if(front==0)
   {
   front=max - 1;
   }
```

```
      else
      {
        f r=f r o n t - 1 ;
      }
  p r i n t f ( " \ n  E n t e r  t h e  i t e m  t o  i n s e r t  i n t o  t
  s c a n f ( " % d " , & i t e m ) ;
  q u e u e [ f r=r e a r + 1 ] = i t e m ;
  p r i n t f ( " % d     i s  i n s e r t e d  i n t o  t h e  Q u e u e " ) ;
    }
    }
  v o i d  s h o w ( )
  {
  i n t  i ;
  i f ( f r o n t = = - 1 )
  p r i n t f ( " \ n * * *  T h e  Q u e u e  i s  e m p t y  * * * " ) ;
  e l s e
  {
p r i n t f ( " D e q u e  c o n t a i n s  " ) ;
  i f ( r e a r < f r o n t )
  {
  f o r ( i = f r o n t ; i < = m a x ; i + + )
  p r i n t f ( " % d  " , q u e u e [ i ] ) ;
  f o r ( i = 0 ; i < = r e a r ; i + + )
  p r i n t f ( " % d  " , q u e u e [ i ] ) ;
  }
  e l s e
  {
  f o r ( i = f r o n t ; i < = r e a r ; i + + )
  p r i n t f ( " % d  " , q u e u e [ i ] ) ;
  }
  }
  }
  i n t  D e l e t e F ( )
  {
  i f ( f r o n t = = - 1 )
  {
  p r i n t f ( " \ n * *  U n d e r f l o w  :  Q u e u e  i s  e m p t y  * *
  r e t u r n  0 ;
  }
  e l s e
  {
  p r i n t f ( " \ n % d  i s  d e l e t e d  f r o m  t h e  Q u e u e " ,
  i f ( f r o n t = = r e a r )
  {
  f r o n t = r e a r = - 1 ;
  r e a r = r e a r - 1 ;
  }
```

```
    else
    {
    front=(front+1) % max;
    }
    }
    }
    int DeleteR()
    {
    if(front==-1)
    {
    printf("\n** Underflow : Queue is empty
    return 0;
    }
    else
    {
    printf("\n%d  is deleted from the Queue",
    if(front==rear)
    {
    front=-1;
    rear=-1;
    }
    else if(rear==0)
    rear=max-1;
    else
    rear=rear-1;
    }
    }
void main()
{
    int ch;
    char ch1;
    do
    {
    clrscr();
    show();
    printf("\n*** Implementation of Circular
    printf("  1: To Insert from rear end \n");
    printf("  2: To Delete from rear end \n");
    printf("  3: To Insert from front end \n"
    printf("  4: To Delete from front end \n"
    printf("  5: To show the item in the Circu
    printf("  0: To Exit\n");
    printf("Enter your Choice : ");
    scanf("%d",&ch);
    switch(ch)
```

```
{
case 1:
  InsertR();
  break;
case 2:
  DeleteR();
  break;
case 3:
  InsertF();
  break;
case 4:
  DeleteF();
  break;
case 5:
  show();
  break;
case 0:
exit(0);
}
printf("\nPress Y to continue");
ch=getch();
}while((ch=='y'||ch=='Y');
getch();
}
```

Output

```
Deque contains 4    6
*** Implementation of Circular Queue ***
 1: To Insert from rear end
 2: To Delete from rear end
 3: To Insert from front end
 4: To Delete from front end
 5: To show the item in the Circular Queue
 0: To Exit
Enter your Choice :0
```

Example 4 *Consider the following sequentially implemented linear queue of characters, which is allocated 5 characters with Front (F) = 1 and Rear (R) = 2. Now, perform the following operations.*

Queue

	B	C		

$F = 1, R = 2$

(*a*) *Add D to the queue.*
(*b*) *Add E to the queue.*
(*c*) *Delete B from the queue.*

Solution

(a) We know that the element can be inserted only from the rear end of the queue, so first increase the value of rear and then add D at the new index so after addition of D, the front will remain same and rear is changed to 3 and the queue will look like

Queue

	B	C	D	

$F = 1, R = 3$

(b) After addition of E again rear is changed and it will be 4 and front will remain same. The resultant queue is shown here

Queue

	B	C	D	E

$F = 1, R = 4$

(c) We know that the element can be deleted only from the front end of the queue so first delete the value at the front of the queue and then increase the value of front so after deletion of D, the rear will remain same and front is changed to 2 and the resultant queue is shown below

Queue

		C	D	

$F = 2, R = 3$

Example 5 *Consider the following sequentially implemented double ended queue of characters which is allocated 5 characters with Front = 1 and Rear = 2.*

DQueue

	B	C		

$F = 1, R = 2$

Now, perform the following operations on the deque

(***a***) *Add D from the front end.*
(***b***) *Add E from the rear end.*
(***c***) *Delete E from the rear end.*
(***d***) *Delete D from the front end.*

Solution

(a) When the element is added from the front end of the deque, the front is decremented by 1 and then the element is inserted at the new location of front so after insertion of D in the deque the front will become 0 and rear will remain same. The resultant queue is shown here

DQueue

D	B	C		

$F = 0, R = 2$

(b) When the element is added from the rear end of the deque, it behaves like a simple queue so the rear is incremented by 1 and then the element is inserted at the new location of rear so after insertion of E in the deque the front will remain same and rear will become 3. The resultant queue is shown here

DQueue

D	B	C	E	

$F = 0, R = 3$

(c) When the element is deleted from the rear end of the deque, the element is deleted from the rear end and then the rear is decremented by 1 so after deletion of E from the deque the front will remain same and rear will become 2. The resultant queue is shown here

DQueue

D	B	C		

$F = 0, R = 2$

(d) When the element is deleted from the front end of the deque, the element is deleted from the front and then the front is incremented by 1 so after deletion of D from the deque the front will become 1 and rear will remain same. The resultant queue is shown here

DQueue

	B	C		

$F = 1, R = 2$

Example 6 *Consider the following sequentially implemented input restricted double ended queue of characters which is allocated maximum 5 characters with Front = 1 and Rear = 2.*

DQueue

	B	C		

$F = 1, R = 2$

Perform the following operations

(***a***) *Add D to the front end.*
(***b***) *Add E to the rear end.*
(***c***) *Delete E from the rear end.*
(***d***) *Delete B from the front end.*

Solution

(a) D is added from the front (not possible because input restricted deque allows insertion from one end, that is, rear end only)

DQueue

	B	C		

$F = 1, R = 2$

(b) E is added to the rear end

DQueue

	B	C	E	

$F = 1, R = 3$

(c) E is deleted from the rear end

DQueue

	B	C		

$F = 1, R = 2$

(d) B is deleted from the front end

DQueue

		C		

$F = 2, R = 2$

Example 7 *Show the steps to convert the following infix expression to postfix form using Stack:*

$$A + (B * C - (D/E\hat{}F)*E)*H$$

Solution

Put the sentinel in the stack to mark empty stack and at the end of the given expression so it becomes A + (B * C – (D/E^ F)*E)*H#

S. No.	*Symbol Scanned*	*Postfix Expression*	*Stack*
			#
1	A	A	#
2	+	A	#+
3	(	A	#+(
4	B	AB	#+(
5	*	AB	#+(*
6	C	ABC	#+(*
7	–	ABC*	#+(–
8	(	ABC*	#+(–(
9	D	ABC*D	#+(–(
10	/	ABC*D	#+(–(/
11	E	ABC*DE	#+(–(/
12	^	ABC*DE	#+(–(/^
13	F	ABC*DEF	#+(–(/^
14	)	ABC*DEF^ /	#+(–
15	*	ABC*DEF^ /	#+(–*
16	E	ABC*DEF^ /E	#+(–*
17	)	ABC*DEF^ /E*–	#+
18	*	ABC*DEF^ /E*–	#+*
19	H	ABC*DEF^ /E*–H	#+*
20	#	ABC*DEF^ /E*–H*+	#

Example 8 *Evaluate the following postfix expression 16 4 2 – * 12 5 2 – / +.*

Solution

Put the sentinel # at the end of the postfix expression so it becomes
16 4 2 – * 12 5 2 – / + #

Scan the expression from left to right. If an operand is encountered then put it on the stack and if operator is encountered then evaluate the top two elements of the stack.

S. No.	*Symbol Scanned*	*Stack*
1	16	16
2	4	16 4
3	2	16 4 2
		Evaluate 4 − 2 = 2
4	−	16 2
		Evaluate 16 * 2 = 32
5	*	32
6	12	32 12
7	5	32 12 5
	2	32 12 5 2
		Evaluate 5 − 2 = 3
8	−	32 12 3
		Evaluate 12/ 3 = 4
9	/	32 4
		Evaluate 32 + 2 = 36
10	+	36
11	#	Stop

Summary

1. A stack is a linear list implemented in last in first out, in which all additions are restricted to one end called the Top of the stack.
2. Push operation is used to add an item to the Top of the stack and recently added new item becomes the new Top of the stack.
3. A push operation cannot be performed if there is no sufficient room for new item. This condition is called overflow state.
4. Pop operation is used to remove the item from the Top of the stack. After the pop operation, the next item (if any) becomes the new Top.
5. A *pop* operation cannot be performed in underflow condition, so each pop must ensure that there is at least one item in the stack.
6. A stack is used to facilitate recursive calls.
7. A queue is a linear list in which data can only be inserted at one end called the rear end and deleted from the other end called the front end.
8. A queue is a first in first out (FIFO) data structure.
9. Stacks and queues can be implement using linked lists or arrays.
10. Deque is a special queue in which the insertion and deletion can be performed at either end.
11. Priority queue is a queue in which priorities are assigned to the elements.
12. Recursion is a method in which function calls itself.
13. There are two types of recursion: direct and indirect recursions.

Key Terms

Stack	Last in first out (LIFO)	Rear
Pop	Top	Circular queue
Push	Stack Top	Priority queue
Empty stack	Deque	Recursion
Overflow	First in first out (FIFO)	Indirect recursion
Underflow	Front	Direct recursion

Multiple-Choice Questions

1. A linear list in which elements can be added or removed at either end but not in the middle is known as
   a. stack.
   b. queue.
   c. linked list.
   d. array.

2. Which data structure would you use for recursive call
   a. stack.
   b. queue.
   c. linked list.
   d. array.

3. The time required to insert an element in a stack with linked implementation is
   a. O(1).
   b. O($\log_2 n$).
   c. O(n).
   d. O($n\log_2 n$).

4. The time required to insert an element in a queue of n elements with sequential implementation is
   a. O(1).
   b. O($\log_2 n$).
   c. O(n).
   d. O($n\log_2 n$).

5. Which of the following statement is false?
   a. The postfix expression is merely a mirror image of the prefix expression.
   b. A stack can be used in evaluating a prefix expression.
   c. Parenthesis are never needed in prefix and postfix expressions.
   d. None of the above.

6. Which of the following is/are a Polish notation?
   a. Postfix
   b. Prefix
   c. Infix
   d. None of the above

7. What can be said about the array representation of a circular queue when it contains only one element?
   a. $F = R =$ Null
   b. $F = R + 1$
   c. $F = R - 1$
   d. $F = R = 0$

8. Which of the following data structure may give overflow errors even though the memory space is available?
   a. Linear queue
   b. Circular queue
   c. Stack
   d. Linked list

9. The postfix form of the A + B * C is
   a. ABC+*
   b. ABC*+
   c. AB+C*
   d. AB*C+

10. The time required to insert an element in a stack with linked implementation is
    a. O(1).
    b. O($\log_2 n$).

c. $O(n)$.
d. $O(n\log_2 n)$.

11. A circular list can be used to represent
a. deque.
b. queue.
c. circular queue.
d. all of the above.

12. Which of the following statements is not correct?
a. Stacks can be implemented by using arrays or linked lists.
b. A linear list is a sequence of one or more data items.
c. All insertions and deletions take place at one end in the stack.
d. Stacks are used in recursion.

13. An element of a stack is read in O(1) time only if
a. stack is sequentially implemented.
b. stack is dynamically implemented.
c. It does not matter how the stack is implemented.
d. None of the above.

14. In input-restricted deque,
a. insertion is possible from one end and deletion from both the ends.
b. deletion is possible from one end and insertion from both ends.
c. insertion and deletion are possible only from one end.
d. insertion and deletion are possible from different ends.

15. In output-restricted deque,
a. insertion is possible from one end and deletion from both ends.
b. deletion is possible from one end and insertion from both ends.
c. insertion and deletion are possible only from one end.
d. insertion and deletion is possible from different ends.

16. In deque,
a. insertion is possible from one end and deletion from both the ends.
b. deletion is possible from one end and insertion from both ends.
c. insertion and deletion is possible only from one end.
d. insertion and deletion is possible from both ends.

17. The postfix expression corresponding to (A – B)*(C/D) + E is
a. AB–CD/*E+
b. AB*–CD/E+
c. AB–*CD+E/
d. AB*–CD+/E

18. For the statically implemented circular queue of six elements if Front = 1 and Rear = 2, what will be the value of front and rear after inserting two more elements?
a. $F = 1, R = 2$
b. $F = 1, R = 4$
c. $F = 4, R = 1$
d. $F = 4, R = 4$

19. The prefix notation for the expression A/B^ C+D is
a. +/A^ BCD
b. /+AB^ CD
c. AB/C^ D+
d. ABC^ /D+

20. The position of the element to be inserted in a circular queue will be calculated by
a. Rear = (Rear + 1)% Size of the Queue
b. Front = (Front + 1)% Size of the Queue
c. Rear = (Rear – 1)% Size of the Queue
d. Front = (Front – 1)% Size of the Queue

21. The position after the deletion of an element in a circular queue will be calculated by
a. Rear = (Rear + 1)% Size of the Queue
b. Front = (Front + 1)% Size of the Queue
c. Rear = (Rear – 1)% Size of the Queue
d. Front = (Front – 1)% Size of the queue

22. When the element is deleted from the front end of the deque then
a. Front is increased by 1.
b. Rear is increase by 1.
c. Front is decreased by 1.
d. Rear is decreased by 1.

23. When the element is inserted from the front end of the deque then
    a. Front is increased by 1.
    b. Rear is increased by 1.
    c. Front is decreased by 1.
    d. Rear is decreased by 1.
24. When the element is deleted from the rear end of the deque then
    a. Front is increased by 1.
    b. Rear is increase by 1.
    c. Front is decreased by 1.
    d. Rear is decreased by 1.
25. When the element is inserted from the rear end of the deque then
    a. Front is increased by 1.
    b. Rear is increased by 1.
    c. Front is decreased by 1.
    d. Rear is decreased by 1.

Review Questions

1. Define stack and also describe the different operations that can be performed on the stack.
2. Explain the static and dynamic implementation of a stack.
3. Explain push and pop operations on a stack.
4. Explain the underflow and overflow condition of a stack.
5. Compare the array and dynamic representation of stack.
6. How is the queue similar to a stack?
7. How is a single queue implemented using two stacks? Write a program for it.
8. Write down the procedure to implement two stacks using only one array.
9. Define the following:
   (a) Infix.
   (b) Polish notation.
   (c) Reverse Polish notation.
10. Write a program to convert an infix expression to postfix notation for the following expression: (3**2*5)/(3*2–3)+5
11. How can we convert infix notation to postfix notation using stack?
12. Convert the following infix notation into postfix notation using stack:
    A–B – (C*D–F/G)*E
13. Write down a short note on recursion. Give a suitable example.
14. What do you mean by direct and indirect recursion? Write a recursive coding for "Tower of Hanoi".
15. What is a queue? What are the different operations that can be applied to a queue?
16. What are the limitations of a linear queue and how are they removed in a circular queue?
17. Write a procedure that implements circular queue.
18. Write short notes on deque and priority queue.
19. Explain the dynamic implementation of circular queue.
20. Convert the following infix expression to postfix notation and give various steps in involved in evaluating using stacks: (3^ 2*5)/(3*2–3)+5.
21. Write different applications of a stack.
22. What are the different ways to represent a queue in the memory?

23. What is deque? Explain different types of deques with example.
24. Explain circular queue with an example.
25. What is a priority queue?
26. Explain how a postfix expression is evaluated using stack with suitable examples.
27. Write the procedure to convert infix expression into postfix expression.
28. Discuss the use of a stack in implementing recursive procedure.
29. Translate the following expressions into their equivalent postfix expressions:
    (a) (A+B)*C–D*E+C
    (b) A*B–C*D
30. What is LIFO?
31. How is recursion different from iteration?
32. Is it possible to represent more than one stack in an array? If yes, explain the process.
33. What is FIFO?
34. How is a stack different from a queue?
35. How can more than one queue be represented in an array?

Programming Assignments

1. Write a program in C/C++ to convert a decimal number into binary number using stack.
2. Write a program in C/C++ to find whether a string is a palindrome or not using stack.
3. Write a recursive program in C for the Fibonacci series.
4. Write a program in C/C++ to insert an element in an input restricted deque.
5. Write a program in C++ using class to convert infix expression into postfix expression and then evaluate the postfix expression.
6. Write a program in C/C++ to implement a queue using two stacks.

Answers

Multiple-Choice Questions

1. (b)
2. (a)
3. (a)
4. (a)
5. (a)
6. (b)
7. (d)
8. (a)
9. (b)
10. (a)
11. (d)
12. (b)
13. (a)
14. (a)
15. (b)
16. (d)
17. (a)
18. (b)
19. (a)
20. (a)
21. (b)
22. (a)
23. (c)
24. (d)
25. (b)

6 Linked List

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Definition of single linked list, doubly linked list and circular linked list.
- Definition of static and dynamic representation of single linked list, doubly linked list and circular linked list.
- Comparison of two methods of representation.
- Operations on single linked list, doubly linked list and circular linked list.
- How to write algorithms to perform various operations on the linked list.
- Implementation of the linked list in C and C++.

In Chapter 2, we had learnt about the array, which is a linear collection of similar elements. The main drawback of using arrays is that the size of the array must be defined in advance. Here, the size of the array is fixed at compile time which cannot be changed and it is not always possible to define the maximum size of the array in advance. If we want to insert an item in the middle of the array, then half of the array elements would have to be shifted. Similarly, if we want to delete the middle element from the array then again half of the array elements need to be shifted. Suppose we declare an array of size 10. Now if we use it for five elements only then this would result in the wastage of memory and if we want to use it for 12 elements then resizing of the array is required. Therefore, although arrays are easy to create and use, they are not flexible and are inefficient for insertion and deletion of sequential data items. Hence we must go for an efficient alternative in which the size of the list can be defined as and when required. In this chapter we will discuss the more flexible data structure called the linked lists, their concepts, their types, different operations on those types and their implementation.

6.1 Linked List as Data Structure

Linked list is a collection of similar elements where each element points to the `next` element. A linked list is a linear list of specially designed nodes where each node is divided into two parts:

1. **`Info` field:** It contains the necessary information about the items of the list to be stored and processed.
2. **`Next` field:** It contains the address of the `next` (adjacent) node. This field is used to access the `next` data item maintained in the other node. The typical node is shown in Figure 1.

Figure 1 A node.

The `next` field contains a pointer giving the address of the `next` item in the list. This field is used to maintain the linking of the data items of the list. Here we must note that data items are maintained at a fixed distance in the array, which has fixed blocks and size that cannot be changed. The data items of the linked list are not necessarily maintained in the contiguous memory locations; they may be placed anywhere in the memory. So the correct order of the data items of the linked list is maintained by the logical addresses maintained in the `next` field of each node. The `next` field contains the address of the next node so it makes possible the insertion and deletion of data items in a constant time.

We can access the data items of the linked list sequentially only. The linked list does not allow the random access of the data items because they are linked to each other through pointers. A schematic diagram of a linked list with five nodes where each node is divided into two parts is shown in Figure 2.

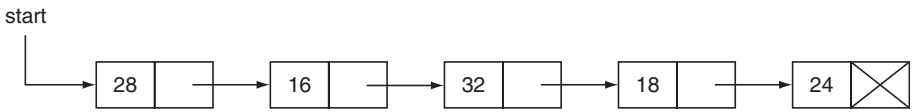


Figure 2 A linked list.

Here the linked list (28, 16, 32, 18, 24) is a collection of similar kind of data items where we can see that the first node contains the `info`=28 and `next`=address of the next node containing `info`=16, which itself contains the address of the next node containing `info`=32 and so on. The left part of each node contains the actual data items and the right part of each node contains the address of the next node represented by an arrow drawn from it to the next node. The last node contains `info`=24 and the `next` field of the next node does not contain any address, so the list ends here. The address of the first node is taken into a variable called "`start`"; hence there is an arrow drawn from `start` to the first node of the linked list. If we assume different addresses in the memory for different data items as shown in Table 1 then the linked list would appear as shown in Figure 3.

Table 1 Data item with assumed memory addresses

Data item	*Address in the memory*
28	2008
16	2099
32	1236
18	3456
24	4520

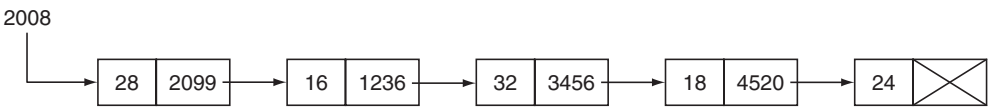


Figure 3 Linked list for the example.

6.2 Representation of Linked List

There are two common ways to implement most of the data structures. Linked list can also be represented in the following two ways:

1. Static or sequential or array representation (implementation).
2. Dynamic or pointer or linked representation (implementation).

We have so far discussed the concept of a simple linked list, also called single linked list, in which each node contains one `info` field and one `next` field. This `next` field points to the next node in the list. The `next` field in the last node points to a null value. There are many other types of linked lists that will be discussed later in this chapter. We will now discuss the representation of single linked list.

Static or Array Representation (Implementation)

Linked list is also a collection of similar data items like an array and so it can easily be implemented using an array. The data items of the linked list are maintained in the form of nodes divided into two parts; therefore, the linked list may be maintained or represented by two parallel arrays of the same sizes. One array for the actual data items is called the *Info* field of the nodes and the other array for the addresses is called the `next` field of the nodes. These two fields are shown in Figure 4. As an example let us take the array of 12 elements.

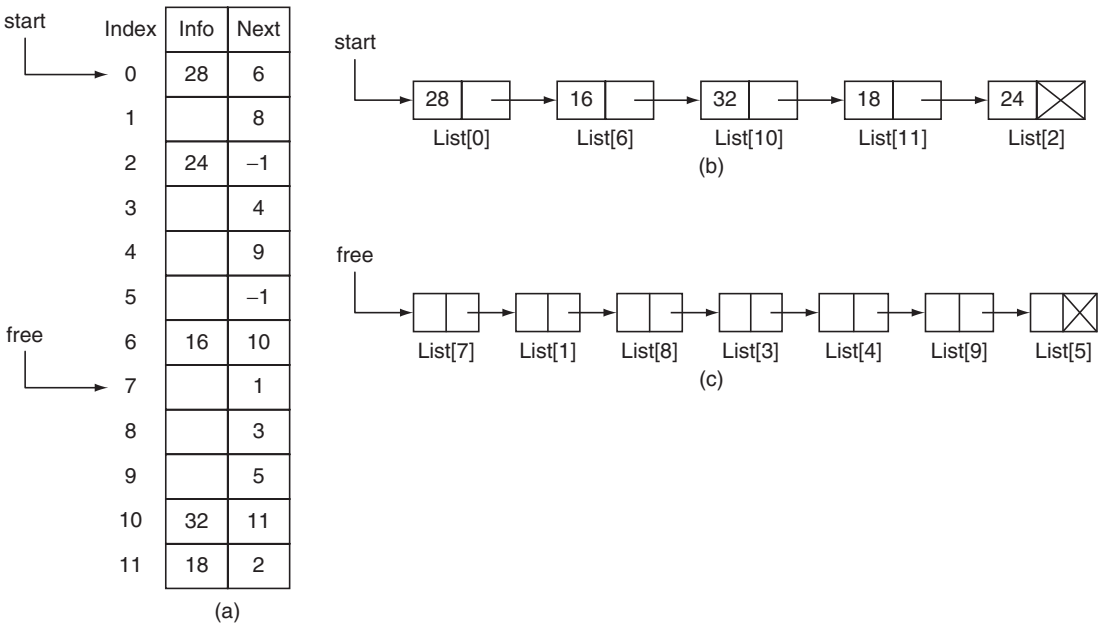


Figure 4 (a) An array representation of a linked list; (b) pictorial representation of linked list using an array; (c) pictorial representation of free blocks of the array.

As we can see the elements are not placed into the contiguous memory location but they are scattered into different locations of the array named "`List`" [Figure 4(a)]. `List[0]` contains the first element 28 of the list pointed by name "`start`" and the `next` field of `List[0]` contains 6, which means the next element is at List[6]. Similarly, the `next` field of `List[6]` contains the index value 10 of the third element 32 and so on. The list of elements is shown in Figure 4(b). Since array has a fixed size declared in advance, so we have to maintain other linked lists of free locations. Let us take the first

free location at index value 7 pointer by the name "Free". The list of free locations is maintained as shown in Figure 4(c). Linked list using array may be implemented in C/C++ by using array of structure defined as follows.

In C

```
struct node
{
int info;
int next;
};
struct node List[12];
```

Remember to use struct keyword

In C++

```
struct node
{
int info;
int next;
};
node List[12];
```

struct keyword is not required

Therefore, List[12] is a collection of 12 nodes each containing integer type of info and integer type of next because array index is also an integer and we use array index as a pointer to a node. For example, List[6] points to the sixth element of the array that contains data item 16 and next address = 10 as shown in Figure 4(a). In the array representation (Table 2), array index starts from 0, so the null value may be represented by –1. Hence, if the next field of any node of the linked list contains –1; it means there is no next element in the list. When the linked list is represented using an array as explained earlier and shown in Figure 4 then

List[n] represents the node of the linked list pointed by index value *n*. For example, List[6] is the node of the list pointed by index value 6, that is

16	10

List[n].info represents the info part of the node of the linked list pointed by index value *n*. For example, List[6].info = 16

List[n].next represents the next field of the node of the linked list pointed by index value *n*. For example List[6].next = 10

Dynamic or Pointer Representation (Implementation)

The array implementation of the linked list is very simple and efficient in randomly accessing the elements of the list, but it requires the ultimate size of the linked list to be declared in advance.

Table 2 Representation of info and next field by an index of the array "List"

Index	List{index}.info	List{index].next
0	28	6
6	16	10
10	32	11
11	18	2
2	24	– 1

The advance declaration of size is not possible all the time; therefore, we can say that array representation of linked list is not flexible. The size of the linked list may increase or decrease according to the requirement of the application because the size of the linked list constantly changes as the items are inserted or deleted. If the item is inserted at a specified position, say, at the middle of the linked list then it requires the shifting of elements (half of the list in this case) which is not an efficient way because it consumes extra time and memory. Similarly, the deletion of any element also leaves an empty space. Array implementation is also called static implementation, which is not an efficient method when resource optimization is concerned. Fortunately, dynamic implementations of linked list do not have size limitations and use space proportional to the actual number of elements of the list. This is represented in computer memory with the help of following structure in C/C++.

In C

```
struct node
{
int info;
struct node *next;
};
struct node *start;
```

Remember to use `struct` keyword

In C++

```
struct node
{
int info;
node *next;
};
node *start;
```

The `struct` keyword is not required

A schematic diagram of a linked list with five nodes where each node is divided into two parts is shown in Figure 5.

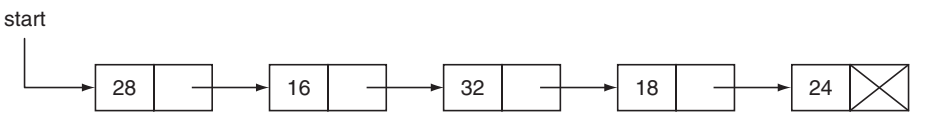


Figure 5 Linked list with five nodes.

Here we can see that

```
List[start].info = 28
List[start].next = 16
```

In the dynamic representation of the linked list, the nodes are created as and when required by using the dynamic memory allocation methods of C or C++ as already discussed in Chapter 4.

6.3 Operations on the Linked List

The various operations that can be performed on the linked list are discussed in the following subsections.

Create Linked List

This operation is used to create the linked list. In the array implementation of the linked list, the numbers of nodes are declared in advance and the elements are entered into the contiguous memory locations. They are placed sequentially into the array. We have to maintain the list of free nodes in array implementation. Initially, all the nodes are empty and linked to one another in sequence. The `next` field of the last node contains –1 to indicate the end of the list. All the nodes are maintained as a list of free nodes.

```
free=0;
for(i=0;i<=Size;i
List[i].next=i
List[i].next=-1;
```

In the dynamic implementation, the list is created by creating the nodes of the list as and when required. There are various functions in C/C++ to allocate the memory dynamically as already explained.

In C	In C++
`struct node`	`struct node`
`{`	`{`
`int info;`	`int info;`
`struct node *ptr;`	`struct node *p`
`};`	`};`
`struct node *ptr;`	`node *ptr;`
`ptr=(struct node*)` `malloc(sizeof` `(struct node));`	`node=new node;`

Algorithm 1 `Createlist()`
To create a linked list "List" where starting node is pointed by the *start* pointer. Here *ptr* is a pointer pointing to the current node and *value* is the item to insert into the node.

1. Declare the structure of a node and define "start" pointer to the node type.
2. Set `Start=NULL` to indicate the list is initially empty.
3. Get the free new node from the memory heap assign its address into pointer "ptr"
4. Assign the value `ptr->info=value` to the *info* part of the new node pointed by `ptr`.
5. Assign the `NULL` to `ptr->next` by using `ptr->next;` the next part of the new node.
6. Assign the address of the first node to the start variable by `start=ptr`.
7. Return.

Using the programming in C we can further elaborate the algorithms of `Createlist()`

Step 1: Declare the structure of a node and define *start* pointer to the node type.

```
struct node
{
int info;
node *next;
};
struct node *start;
```

Step 2: Set `start = NULL` to indicate the list is initially empty.

```
start = NULL;
```

Step 3: Get the free new node from the memory heap assign its address into pointer `ptr`.

```
struct node *ptr=(struct node *)malloc(sizeof(
                 node));
```

Step 4: Assign the value to be inserted to the *info* part of the new node pointed by `ptr`

```
ptr->info = (First supply the value)
```

Step 5: Assign `NULL` to the *next* part of the new node

```
ptr->next = start (ptr->next becomes NULL if start = NULL
```

Step 6: Assign the address of the first node to the `start` variable.

```
start = ptr
```

Step 7: Return.

Traversing a Linked List

This operation is used to visit each node of the list exactly once in order to access the information stored in the nodes. The algorithm to traverse the linked list is as follows.

Algorithm 2 `Traverse(start, info, next, ptr)`

Here *List* is a linked list of nodes, *start* is the pointer which points to the first node of the linked list and *ptr* is a pointer which points to the current node of the linked list.

1. `If start = NULL`
   a. `Print "List is empty"`
   b. `Exit`
   `End if`
2. Set `ptr = start`
3. Repeat Steps 4 and 5 until `ptr != NULL`
4. Access and apply some operation to `ptr->.info`
5. Set `ptr = ptr->next`
   `End Repeat`
6. `Exit`

Program 1 A menu-driven program in C++ to create, display and search an element in a linked list of user's choice.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
```

```
template <class T>
class linked_list
{
struct node
{
T info;
node *next;
};
node *start, *ptr;
T item;
public:
linked_list()
{
start=NULL;
}
creatlist()
{
char ch;
do
{
ptr=getnode();
cout<<"Enter value to insert: ";
cin>>item;
ptr->info=item;
ptr->next=start;
start=ptr;
cout<<"\nWant to insert more nodes: ";
cin>>ch;
}while(ch=='Y'||ch=='y');
}
node* getnode()
{
ptr=new node;
return ptr;
}
search()
{
int flag=0;
cout<<"\nEnter the item to search: ";
cin>>item;
ptr=start;
while(ptr!=NULL)
{
if(ptr->info==item)
{
flag=1;
```

```
}
ptr=ptr->next;
}
if(flag==1)
{
cout<<"item found ";
getch();
exit(1);
}
else
{
cout<<" item not found";
}
}
display()
{
ptr=start;
cout<<"start";
while(ptr!=NULL)
{
cout<<"-->"<<ptr->info;
ptr=ptr->next;
}
}
};
void main()
{
clrscr();
linked_list<int>obj;
linked_list<char>obj1;
linked_list<float>obj2;
int choice;
cout<<"** MENU **"<<endl;
cout<<"1. To create, display and search lis
cout<<"2. To create, display and sea
characters"<<endl;
cout<<"3. To create, display and search lis
cout<<"Enter your choice";
cin>>choice;
switch(choice)
{
case 1:
obj.creatlist();
obj.display();
obj.search();
break;
```

```
case 2:
obj1.creatlist();
obj1.display();
obj1.search();
break;
case 3:
obj2.creatlist();
obj2.display();
obj2.search();
}
getch();
}
```

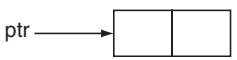
Output

```
** MENU **
1. To create, display and search list of
2. To create, display and search list of
3. To create, display and search list of
Enter your choice2
Enter value to insert: a
Want to insert more nodes: y
Enter value to insert: b
Want to insert more nodes: y
Enter value to insert: c
Want to insert more nodes: y
Enter value to insert: d
Want to insert more nodes: n
start-->d-->c-->b-->a
Enter the item to search: a
item found
```

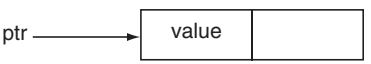
Insertion of an Element into the Linked List at Various Positions

Insertion of a node into a linked list requires a free node in which the information can be inserted and then the node can be inserted into the linked list. The pointers are then adjusted. Insertion of any element into the linked list requires following steps:

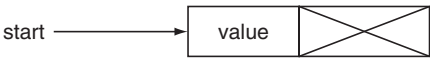
1. Get the free node from the memory pool and assign its address to the `ptr`



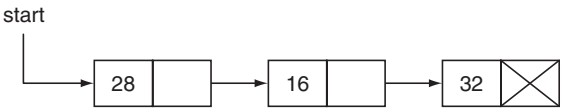
2. Assign the value `ptr->info = value` to the info part of the new node pointed by `ptr`



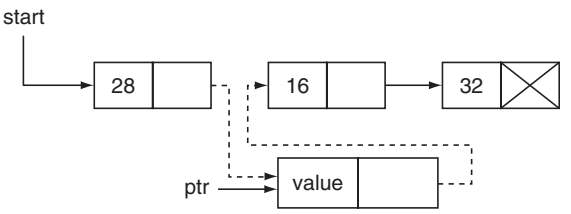
3. If the linked list is empty, that is, `start = NULL` then assign `ptr->next = NULL` and `start = ptr`



4. Otherwise adjust the pointers according to the place of insertion. A typical example is shown below.



Before insertion

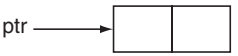


After insertion

Insertion at the Beginning

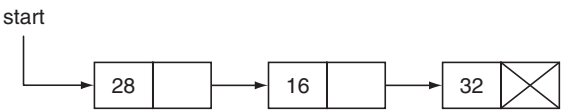
The process is described here with the help of the diagram.

1. The first requirement is to get a free node. Let it be pointed by `ptr`

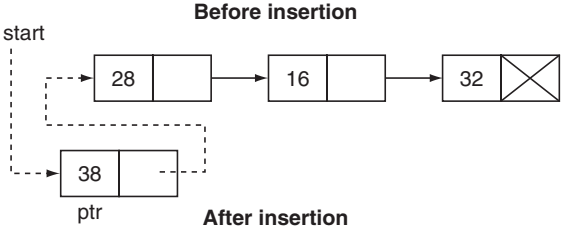


   Now the question is how to get the free node. It depends on the implementation of the linked list whether we use array (static) representation or dynamic (pointer) representation. If we are using the array implementation then the node can be obtained from the free list and if the list is implemented using dynamic implementation than the node can be created dynamically using `malloc` of C or `new` operator of C++. If `ptr` returns null (or –1 in the case of array representation) then we cannot insert the element because of overflow condition.
2. Set the `info` (say 38) field of the newly created node.

3. Now adjust the pointers. Therefore the `start` pointer will point to the new node pointed by `ptr` and the `next` pointer of the new node `ptr` will point to the previous `start` node. The algorithm is given in Algorithm 3.



Before insertion



After insertion

Algorithm 3 InsertBeg(List, value, start, ptr)
Here "List" is a linked list of nodes, *start* is the pointer which points to the first node of the linked list or contains null if the List is empty and *ptr* is a pointer which points to the new node. We want to insert the value at the beginning of the List.

1. If ptr = NULL
   a. Print "Overflow"
   b. Exit
   End if
2. Set ptr->info = value
3. Set ptr->next = start and start = ptr
4. Exit

The insertion in the beginning of the linked list implemented using an array can be performed similarly as shown in the following algorithm.

Algorithm 4 InsertBeg(List, value, start, ptr)
This algorithm is to insert a value in the linked list implemented sequentially (using array). Here List is a linked list of nodes, start is the variable which holds the starting index of the list or –1 if the List is empty. Here pt is a variable which holds the index of the first free node or –1 if list of available free nodes is empty. We want to insert value at the beginning of the List.

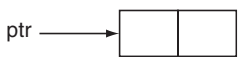
1. If ptr = -1
   a. Print "Overflow"
   b. Exit
   End if
2. Set List[ptr].info = value
3. Set List[ptr].next = start and start = ptr
4. Set ptr = List[ptr].next
5. Exit

Other algorithms of this chapter can also be changed to the equivalent array implementation of the linked list by taking the above algorithm into consideration. From now onwards, we would discuss the algorithms for the dynamic implementation of a linked list.

Insertion at the End of the List

The process is described here with the help of the diagram.

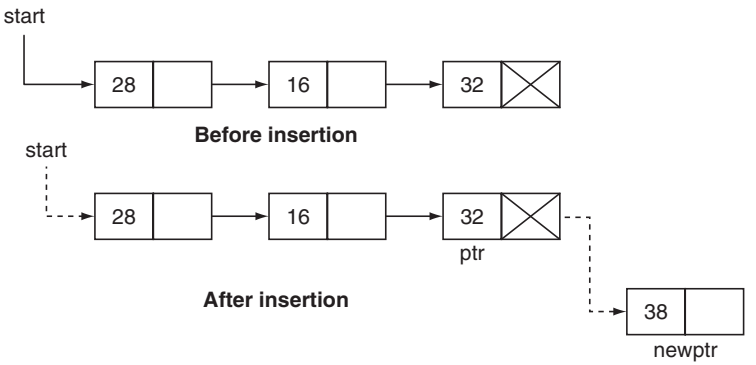
1. The first requirement is to get a free node. Let it be pointed by pt.r



If pt returns null (or –1 in the case of array representation) then we cannot insert the element because of overflow condition.

2. If overflow does not occur then set the `info` (say 38) field of new node pointed by `ptr`

3. Now adjust the pointers. Therefore, the `next` pointer of the last node will point to the new node pointed by `ptr` and the `next` pointer of the new node `ptr` will point to the `next` pointer (null) of the previous last node. The algorithm is given in Algorithm 5.



Algorithm 5 `Insertend(List, value, start, ptr,`

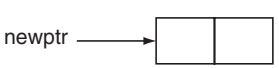
Here `List` is a linked list of nodes, *`start`* is the pointer which points to the first node of the linked list or null if the `List` is empty and *`newptr`* is a pointer which points to the new node. *`ptr`* is a pointer variable of `newptr` type to temporarily keep the address. We want to insert value at the end of the `List`.

1. `If newptr = NULL`
   a. `Print "Overflow"`
   b. `Exit`
   `End if`
2. Set `newptr->info = value`
3. Set `ptr = start`
4. Repeat `while ptr->next != NULL` (to find the last node).
5. Set `newptr->next = ptr->next`
6. Set `ptr->next = newptr`
7. `Exit`

Insertion at a Specified Position in the Linked List

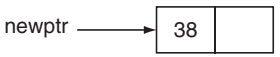
To insert a new node at the specified position, we have to search for that specified position in the list. Then we can insert the node after that position. The process is described here with the help of the diagram.

1. The first requirement is to get a free node. Let it be pointed by `newptr`

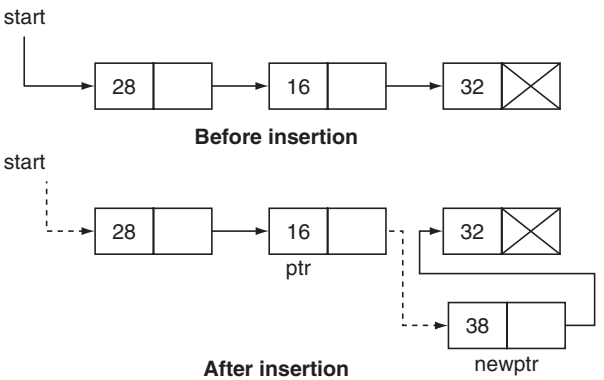


If `newptr` returns null (or –1 in the case of array representation) then we cannot insert the element because of overflow condition.

2. If overflow does not occur then set the `info` (say 38) field of new node pointer by `newptr`



3. Now search for the specified position in the linked list. Let the address of the specified node be kept in `ptr`. Adjust the pointer so that the `next` pointer of the `newptr` node will point to the `next` pointer of the specified node pointed by `ptr` and the `next` pointer of the specified node `ptr` will point to the `next` pointer of the node pointed by `newptr`



Algorithm 6 `Insertafter(List, value, ptr, newptr)`

Here `List` is a linked list of nodes, `start` is the pointer which points to the first node of the linked list or null if the List is empty and `newptr` is a pointer which points to the new node. `ptr` is a pointer variable of `newptr` type to temporarily keep the address. We want to insert value after the given position of the node in the list.

1. `If newptr = NULL`
   a. `Print "Overflow"`
   b. `Exit`
   `End if`
2. Set `ptr = start, count = 1`
3. Perform Steps 4 and 5 `while (count < pos)`
4. `ptr = ptr -> next`
5. `count = count + 1`
   End of Step 3.
6. Set `newptr -> info = value`
7. Set `newptr -> next = ptr -> next`
8. `ptr -> next = newptr`
9. `Exit`

Program 2 A program in C++ to perform insert operations at various position on the linked list.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template <class T>
class linked_list
{
struct node
{
T info;
node *next;
};
node *start,*ptr,*newptr;
T item;
public:
linked_list()
{
start=NULL;
}
creatlist()
{
char ch;
ptr=getnode();
cout<<"Enter value to insert";
cin>>item;
ptr->info=item;
ptr->next=start;
start=ptr;
cout<<"Want to insert more nodes: ";
cin>>ch;
while(ch=='Y'||ch=='y')
{
newptr=getnode();
cout<<"Enter value to insert";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
ptr=ptr->next;
cout<<"\nWant to insert more nodes: ";
cin>>ch;
}
}
node* getnode()
{
```

```
node *newptr1=new node;
return newptr1;
}
void display()
{
node *ptr1=start;
cout<<"The linked list is: start";
while(ptr1!=NULL)
{
cout<<"-->"<<ptr1->info;
ptr1=ptr1->next;
}
cout<<endl;
}
void insbeg()
{
node *newptr2=new node;
cout<<"\nEnter the item to insert";
cin>>item;
newptr2->info=item;
newptr2->next=start;
start=newptr2;
}
void insatend()
{
node *newptr=new node;
cout<<"\nEnter the item to insert";
cin>>item;
newptr->info=item;
ptr=start;
while(ptr->next!=NULL)
{
ptr=ptr->next;
}
newptr->next=ptr->next;
ptr->next=newptr;
}
void insafter()
{
int choice;
cout<<"1. To insert after the given posit
cout<<"2. To insert after given value "<<
cout<<"Enter your choice: ";
cin>>choice;
switch(choice)
```

```
{
case 1:
int pos;
cout<<"Enter the position "<<endl;
cin>>pos;
int i=1;
ptr=start;
while(i<pos)
{
ptr=ptr->next;
i=i+1;
}
node *newptr=new node;
cout<<"Enter the item to insert: ";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
break;
case 2:
cout<<"Enter the value after which you want
cin>>item;
ptr=start;
while(ptr->info!=item)
{
ptr=ptr->next;
}
newptr=new node;
cout<<"Enter the item to insert: ";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
}
}
};
void main()
{
clrscr();
linked_list<int>obj;
char ch;
int choice;
cout<<"*** Create and display the list of
obj.creatlist();
obj.display();
```

```
do
{
cout<<"1. Insertion at beginning"<<endl;
cout<<"2. Insertion at end"<<endl;
cout<<"3. Insertion at after specified el
cout<<"Your choice is: ";
cin>>choice;
switch(choice)
{
case 1:
obj.insbeg();
obj.display();
break;
case 2:
obj.insatend();
obj.display();
break;
case 3:
obj.insafter();
obj.display();
break;
}
cout<<"Do you want more insert operations
cin>>ch;
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** Create and display the list of intege
Enter value to insert 2
Want to insert more nodes: y
Enter value to insert 4
Want to insert more nodes: n
The Linked list is: start-->2-->4
1. Insertion at beginning
2. Insertion at end
3. Insertion at after specified element
Your choice is: 1
Enter the item to insert 3
The linked list is: start-->3-->2-->4
Do you want more insert operations: y
1. Insertion at beginning
2. Insertion at end
3. Insertion at after specified element
```

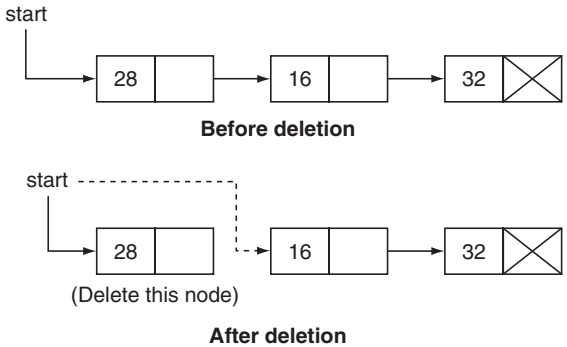
```
Your choice is: 2
Enter the item to insert5
The linked list is: start-->3-->2-->4-->5
Do you want more insert operations: y
1. Insertion at beginning
2. Insertion at end
3. Insertion at after specified element
Your choice is: 3
1. To insert after the given position
2. To insert after given value
Enter your choice: 1
Enter the position: 2
Enter the item to insert: 45
The linked list is: start-->3-->2-->45-->4
Do you want more insert operations: y
1. Insertion at beginning
2. Insertion at end
3. Insertion at after specified element
Your choice is: 3
1. To insert after the given position
2. To insert after given value
Enter your choice: 2
Enter the value after which you want to ins
Enter the item to insert: 23
The linked list is: start-->3-->2-->45-->2
Do you want more insert operations: n
```

Deletion of an Element from the Linked List

There may be some nodes in the list which are no longer required and so they may be deleted from the list. Deleting a node in a linked list is as simple as pointing the previous node to the following node. Deletion from the linked list is an important operation which can be performed at various positions in the linked list just like the insertion operation. In order to delete a node from the linked list it is necessary to search for position of a node to delete. When deletion operation is to be performed we have to check for the underflow condition, that is, if the linked list is empty then we cannot delete the node from the list. In the underflow condition `start = NULL`. In this section we will present the algorithms to delete the nodes from the linked list. When it comes to deleting nodes, we have three choices.

Deletion at the Beginning of the List

Deleting a node at the beginning of the list is a very simple operation but first we should check for the underflow condition. If the first node is to be deleted then the `start` pointer will point to the second node of the list, that is, the list will now start from the second node of the list as shown by the dotted arrow. Hence we have to simply make the pointer `start` equal to the `next` field of the first node in the list. Before deleting the node we should save the value into some variable. After deleting the node we must make it available for future use, so we should return it into the memory heap by *`free()`* function in C and *`delete`* operator in C++.



The algorithm for deletion of an element from the beginning of the list is as follows.

Algorithm 7 Deletebeg(List, value, start, ptr)

Here List is a linked list of nodes, start is the pointer which points to the first node of the linked list or null if the List is empty and ptr is a pointer variable to temporarily keep the address. We want to delete value from the beginning of the List.

1. If start = NULL
   a. Print "Overflow" (linked list is empty)
   b. Return
   End if
2. Set ptr = start
3. Assign value = start -> info
4. Set start = start -> next (second node becomes the first node).
5. Release the node pointed by ptr to the memory heap.
6. Exit

Program 3 A program in C++ to delete the elements from the beginning of the linked list.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template <class T>
class linked_list
{
struct node
{
T info;
node *next;
};
node *start,*ptr,*prev,*newptr;
T item, value;
```

```
public:
linked_list()
{
start=NULL;
}
creatlist()
{
char ch;
ptr=getnode();
cout<<"Enter value to insert ";
cin>>item;
ptr->info=item;
ptr->next=start;
start=ptr;
cout<<"Want to insert more nodes: ";
cin>>ch;
while(ch=='Y'||ch=='y')
{
newptr=getnode();
cout<<"Enter value to insert";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
ptr=ptr->next;
cout<<"\nWant to insert more nodes: ";
cin>>ch;
}
}
node* getnode()
{
node *newptr1=new node;
return newptr1;
}
void display()
{
node *ptr1=start;
cout<<"\nThe linked list is: start";
while(ptr1!=NULL)
{
cout<<"-->"<<ptr1->info;
ptr1=ptr1->next;
}
cout<<endl;
}
```

```
int delbeg()
{
if(start==NULL)
{
cout<<"underflow";
cout<<"The list is empty";
}
else
{
ptr=start;
if(ptr->next==NULL)
{
value=start->info;
cout<<"\nThe deleted item is: "<<value;
start=start->next;
delete ptr;
cout<<"\nList is empty now";
}
else
{
value=start->info;
cout<<"\nThe deleted item is: "<<value;
start=start->next;
delete ptr;
display();
}
}
}
};
void main()
{
clrscr();
linked_list<int>obj;
char ch;
int choice;
cout<<"*** Create and display the list of
obj.creatlist();
obj.display();
cout<<" \n** Deletion from beginning **"<
do
{
obj.delbeg();
cout<<"\nDo you want more deletion?: ";
cin>>ch;
}while(ch=='Y'||ch=='y');
```

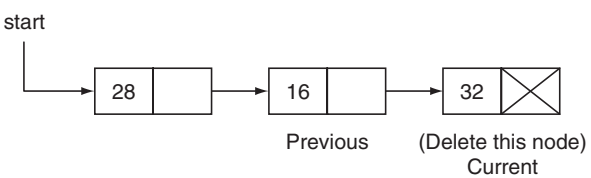
```
getch();
}
```

Output

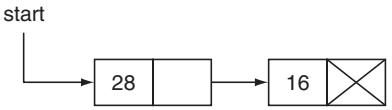
```
*** Create and display the list of integers
Enter value to insert 1
Want to insert more nodes: y
Enter value to insert 3
Want to insert more nodes: y
Enter value to insert 5
Want to insert more nodes: y
Enter value to insert 6
Want to insert more nodes: n
The linked list is: start-->1-->3-->5-->6
** Deletion from beginning **
 The deleted item is: 1
The linked list is: start-->3-->5-->6
Do you want more deletion?: y
 The deleted item is: 3
The linked list is: start-->5-->6
Do you want more deletion?: y
 The deleted item is: 5
The linked list is: start-->6
Do you want more deletion?: y
 The deleted item is: 6
List is empty now
Do you want more deletion?: n
```

Deletion at the End of the List

If the node to be deleted is the last node then traverse the entire list to find the address of the second last node so that we can adjust the pointers. Now the `next` field of the second last node will point to the `next` field of the last node. Hence the `next` field of the second last node will now contain null. Deleting a node from the end of the list is harder, because it requires a temporary pointer to hold the address of the preceding node, that is, of the node to be deleted.



After deleting the last node from the linked list, now the `next` field of the second last node will contain null as shown below.



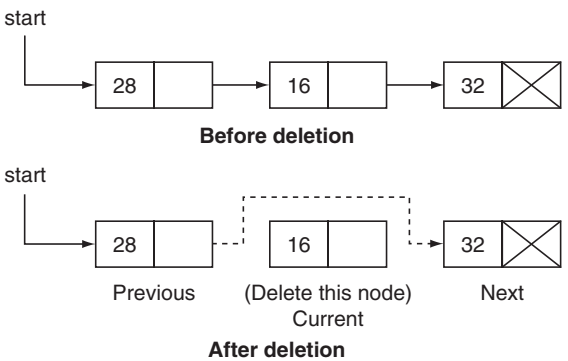
The algorithm for deletion of element at the end of the list is as follows.

Algorithm 8 Deleteend(List, value, start, ptr, pre)

Here List is a linked list of nodes, start is the pointer which points to the first node of the linked list or null if the List is empty, ptr is a pointer variable to temporarily keep the address, pre is a pointer used to hold the address of the previous node of the node we want to delete from the List.

1. If start = NULL
   a. Print "Overflow (linked list is empty).
   b. Return
   End if
2. Set ptr = start
3. If ptr->next = NULL (the list has only one node).
   a. Assign value = ptr->info
   b. Start = NULL
   c. Release the deleted node pointed by ptr to the memory heap.
   Else
4. Repeat a and b while (ptr->next != NULL)
   a. prev = ptr
   b. ptr = ptr->next
5. Assign value = ptr->info
6. Release the deleted node pointed by ptr to the memory heap.
   End if
7. Exit

Now if we want to delete the node containing some specific information or the node at a particular position then we have to traverse the list to find that node and adjust the pointer so that the previous node will now point to the following node as shown by the dotted arrow in the following figure. If the desire node is not found in the list then display the message "Node not found". So the important task is to find the address of the node preceding the node to be deleted.



The algorithm for deletion of an element from the specified position of the list is as follows.

Algorithm 9 Delafter(List, value, item, start,
Here List is a linked list of nodes, *start* is the pointer which points to the first node of the linked list or null if the List is empty, *ptr* is a pointer variable to temporarily keep the address, *prev* is a pointer used to hold the address of the previous node of the node we want to delete from the List containing the item.

1. If start = NULL
   a. Print "Over (linked list is empty).
   b. Return
   End if
2. Set ptr = start (make ptr point to whatever the start pointer is pointing to).
3. If ptr->info (the list has only one node).
   a. Assign value = ptr->info
   b. start = NULL
   c. Release the deleted node pointed by ptr to the memory heap.
   Else
4. Repeat a and b while (ptr->next and !ptr=NULL->info!!=item)
   a. prev = ptr
   b. ptr = ptr->next
5. Assign value = ptr->info
6. Release the deleted node pointed by ptr to the memory heap.
7. Exit

Program 4 A program in C++ to delete an element either from the end of the list or from the specified position.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template <class T>
class linked_list
{
struct node
{
T info;
node *next;
};
node *start, *ptr,*prev,*newptr;
T item, value;
public:
linked_list()
```

```
{
start=NULL;
}
creatlist()
{
char ch;
ptr=getnode();
cout<<"Enter value to insert";
cin>>item;
ptr->info=item;
ptr->next=start;
start=ptr;
cout<<"Want to insert more nodes: ";
cin>>ch;
while(ch=='Y'||ch=='y')
{
newptr=getnode();
cout<<"Enter value to insert";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
ptr=ptr->next;
cout<<"\nWant to insert more nodes: ";
cin>>ch;
}
}
node* getnode()
{
node *newptr1=new node;
return newptr1;
}
void display()
{
node *ptr1=start;
cout<<"The linked list is: start";
while(ptr1!=NULL)
{
cout<<"-->"<<ptr1->info;
ptr1=ptr1->next;
}
cout<<endl;
}
int delend()
{
```

```
ptr=start;
if(ptr->next==NULL)
{
value=ptr->info;
cout<<"\nThe deleted item is: "<<value;
delete ptr;
return 0;
}
while(ptr->next!=NULL)
{
prev=ptr;
ptr=ptr->next;
}
value=ptr->info;
cout<<"\nThe deleted item is: "<<value;
prev->next=ptr->next;
delete ptr;
}
void delafter()
{
node *ptr3,*ptr4;
int choice;
int pos;
cout<<"Enter the position: ";
cin>>pos;
int i=1;
ptr3=start;
while(i<=pos)
{
prev=ptr3;
ptr3=ptr3->next;
i=i+1;
}
value=ptr3->info;
cout<<"\nThe deleted item is: "<<value;
prev->next=ptr3->next;
delete ptr3;              //Release the memory
}
};
void main()
{
clrscr();
linked_list<int>obj;
char ch;
```

```
int choice;
cout<<"*** Create and display the list of
obj.creatlist();
obj.display();
do
{
cout<<"1. Deletion from the beginning"<<e
cout<<"2. Deletion after specified positi
cout<<"Your choice is: ";
cin>>choice;
switch(choice)
{
case 1:
obj.delend();
obj.display();
break;
case 2:
obj.delafter();
obj.display();
break;
}
cout<<"Do you want more deletion:  ";
cin>>ch;
}
while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** Create and display the list of intege
Enter value to insert1
Want to insert more nodes: y
Enter value to insert5
Want to insert more nodes: y
Enter value to insert6
Want to insert more nodes: y
Enter value to insert7
Want to insert more nodes: n
The linked list is:  start-->1-->5-->6-->
1. Deletion from the beginning
2. Deletion after specified position
Your choice is: 2
Enter the position: 2
The deleted item is:  6
The linked list is:  start-->1-->5-->7
Do you want more deletion:  n
```

6.4 Comparison between Array and Pointer Representation of Linked List

An array requires less memory because it stores the information only where the linked list requires more memory in the computer to store nodes consisting of the information and the reference to the next node of the list. We have seen that the linked list can be implemented either using array or using pointers. There are some advantages and disadvantages of each implementation method (see Table 3). Here we are providing comparisons between both the methods of linked list representation based on comparisons between an array and a linked list.

Table 3 Comparisons between dynamic and static representation of linked list

S. No.	*Linked list using dynamic representation (linked list)*	*Linked list using array representation (array)*
1	The elements can be added to the linked list indefinitely, so we can say that linked lists can grow during the execution of a program.	Array has a fixed size, so it will eventually get filled. It cannot grow according to the need. It is hard to predict the size of the array every time.
2	It is dynamic in nature and does not require predicting the size in advance.	It is static in nature and has a fixed size, although it is easy to use.
3	The elements can be deleted from the linked list. so we can say that linked list can shrink during the execution of a program.	Deletion of element from the array leaves an empty space that is a wastage of memory and it requires the shifting of elements which is a costly operation.
4	Insertion and deletion are very fast because it does not require the shifting of elements.	Insertion and deletion in the array are slow because it needs to shift elements.
5	It efficiently utilizes the memory because memory is allocated at run time according to the need.	The memory is allocated at compile time and so it does not utilize the memory efficiently.
6	Linked list allows insertion and deletion of element at specified position by adjusting two pointers and inserting or deleting an element easily. Therefore it is flexible.	Insertion and deletion in the array at specified position require resizing of the array which is not always possible and is a costlier operation.
7	The linked list only allows sequential access to the elements.	Array allows random access to the elements.
8	Access to random element in the list is time consuming and cumbersome because it allows sequential access only.	The random element can be accessed easily in the array because accessing of element is based on the index of array element. Therefore direct access of the elements in the array is very fast.
9	Linked list is used for the larger list in which insertion and deletion are performed frequently.	Arrays, on the other hand, are better suited to small lists of elements, where the maximum number of items that could be in the list is known.
10	In the linked list we need to pass through the first $(k - 1)$ elements in the list to reach the kth element.	We can access the kth element in a single operation by using the index value k of the list.

6.5 Stack as a Linked List

We have already discussed the array implementation of stack in Chapter 5. The array implementation is very simple and efficient, but it requires the ultimate size of the stack to be declared in advance. The advance declaration of maximum size is not possible all the time. The size of the stack may increase or decrease according to the requirement of the application because the stack size is constantly changing as the items are popped and pushed. The dynamic representation of a stack, commonly termed as linked stack, is more suitable if an accurate estimate of the stack's size cannot be made in advance and the data elements are large records. In the linked representation, the stack is represented as a linked list where each node contains *info* and *next* part.

The Top pointer of the stack is represented by the *start* pointer of the linked list. If the *start* pointer is Null then it shows that the stack is empty. If the last node link field contains Null then it shows the bottom of the stack. Figure 6 shows the linked or dynamic representation of the stack.

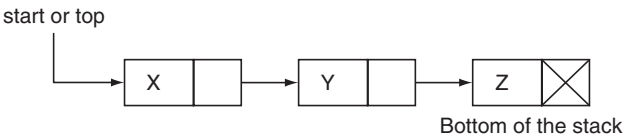


Figure 6 Dynamic representation of stack with three elements.

Here the stack contains three elements Z, Y and X where Z is at the bottom of the stack. We can see that the *info* field of the node contains the items Z, Y and X and the *next* pointer of the node contains the address of the next element (node). Figure 7 shows a linked list in the form of a stack in a more understandable manner.

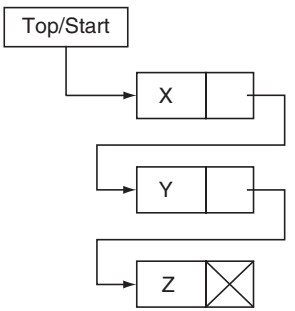


Figure 7 Physical dynamic stack implementation

Regardless of the way a stack is described, its underlying property is that insertions and deletions can occur only at the top of the stack. The algorithms for Push and Pop are as follows:

Algorithm 10 PUSH(STACK, PTR, TOP, ELEMENT)

Here TOP is a pointer pointing to the topmost element of the STACK and we want to push ELEMENT into the STACK

1. Create a new node pointed by PTR
2. If PTR=NULL then Print "OVERFLOW" and Ret
   Set PTR->info=ELEMENT
3. PTR->next=TOP
4. Set TOP=PTR
5. Return

Algorithm 11 POP(STACK, TOP, ITEM, PTR)

1. If TOP=NULL then Print "Underflow" and R
2. PTR=TOP
3. ELEMENT=PTR->info
4. TOP=TOP->next
5. Make deleted node pointed by PTR free
6. Return

Program 5 A program in C++ to implement push and pop operations of the linked stack.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template <class T>
class linked_stack
{
struct node
{
T info;
node *next;
};
node *ptr,*top;
T item;
public:
linked_stack()
{
top=NULL;
}
void push()
{
char ch;
ptr=getnode();
if(ptr==NULL)
{
```

```
cout<<"OVERFLOW\n";
getch();
exit(0);
}
else
{
cout<<"Enter value to insert";
cin>>item;
ptr->info=item;
ptr->next=top;
top=ptr;
}
}
void pop()
{
if(top==NULL)
{
cout<<"UNDERFLOW\n";
getch();
exit(0);
}
else
{
ptr=top;
item=ptr->info;
cout<<"\nThe deleted item is: "<<item;
top=top->next;
}
delete ptr;
}
node* getnode()
{
node *newptr1=new node;
return newptr1;
}
void display()
{
node *ptr1=top;
cout<<"\nThe linked list is: Top ";
while(ptr1!=NULL)
{
cout<<"-->"<<ptr1->info;
ptr1=ptr1->next;
}
cout<<endl;
```

```
}
};
void main()
{
clrscr();
linked_stack<int>obj;
char ch;
int choice;
do
{
cout<<"*** MENU ***"<<endl;
cout<<"1. PUSH"<<endl;
cout<<"2. POP"<<endl;
cout<<"Enter your choice: ";
cin>>choice;
switch(choice)
{
case 1:
obj.push();
obj.display();
break;
case 2:
obj.pop();
obj.display();
}
cout<<"\nDo you want to continue:   ";
cin>>ch;
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** MENU ***
1. PUSH
2. POP
Enter your choice: 1
Enter value to insert 2
The linked list is:  Top -->2
Do you want to continue: y
*** MENU ***
1. PUSH
2. POP
Enter your choice: 1
Enter value to insert 5
The linked list is:  Top -->5-->2
Do you want to continue:  y
```

```
***  MENU  ***
1.  PUSH
2.  POP
Enter  your  choice:  2
The  deleted  item  is:    5
The  linked  list  is:    Top  -->2
Do  you  want  to  continue:    y
***  MENU  ***
1.  PUSH
2.  POP
Enter  your  choice:  2
The  deleted  item  is:    2
The  linked  list  is:    Top
Do  you  want  to  continue:    y
***  MENU  ***
1.  PUSH
2.  POP
Enter  your  choice:  2
UNDERFLOW
```

6.6 Queue as a Linked List

The size of the queue may increase or decrease according to the requirement of the application because fundamentally queue is a dynamic object whose size is constantly changing as the items are deleted and inserted. In the static implementation, once the size of the array is declared, the size cannot be varied during program execution. The other representation of a queue is dynamic representation, commonly termed as linked queue, which is more suitable if an accurate estimate of the queue's size cannot be made in advance and the data elements are large records. A linked queue is a queue implemented as a linked list with two pointer variables Front and Rear pointing to the nodes which are in the front and rear, respectively, of the queue. Linked queue is a collection of nodes where each node of the queue is divided into two parts as shown in Figure 8.

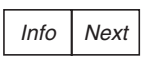


Figure 8 Node representation.

The first half (`info` field) of the nodes contains the element of the queue and the second half (`next` field) holds the pointer to the neighboring element in the queue. Two pointers are also required to maintain front and rear position of the queue. If the pointer Front is Null then it shows that the queue is empty because the empty queue does not contain any element and can be indicated by Rear = Front = Null. Figure 9 shows the linked or dynamic representation of the queue.

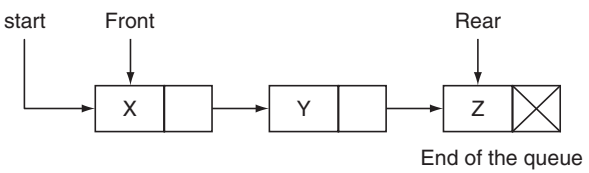


Figure 9 Dynamic representation of queue.

The algorithms for insertion and deletion from a queue are as follows.

Algorithm 12 INSERT(QUEUE, PTR, FRONT, REAR, ELE

Here FRONT and REAR are pointers pointing to the front and rear elements, respectively. Front = Null indicates that the queue is empty. We want to push ELEMENT into the QUEUE

1. Create a new node pointed by PTR
2. If PTR = NULL then Print "OVERFLOW" and Re
3. Set PTR->info = ELEMENT
4. REAR->next = PTR
5. Set REAR = PTR
6. Return

Algorithm 13 DELETE(QUEUE, PTR, FRONT, REAR, ELE

Here FRONT and REAR are the pointers pointing to the front and rear elements, respectively. Front = Null indicates that the queue is empty. We want to push ELEMENT into the QUEUE.

1. If FRONT = NULL then Print "UNDERFLOW" and
2. Set ELEMENT = FRONT->info
3. If (FRONT = REAR) then
   Set FRONT = REAR = NULL
   Else
   FRONT = FRONT->next
4. Return

6.7 Doubly Linked List

The restriction of the single linked list is that we can access the nodes of the list in forward direction only; accessing the previous node of the current node is not possible so we cannot traverse in the backward direction. This drawback is removed in doubly linked list in which each node contains one *info* field and two links; one forward and one backward. These links point to the two nodes adjacent to the current node. The forward pointer points to the next node of the current node and the backward pointer points to the preceding node of the current node. So these pointers help in accessing the successor and predecessor nodes of any arbitrary node within the list. Each node of the doubly linked list is divided into three parts: the first part contains the address of the previous node in the pointer *pre*, the second part contains the actual information into the info variable and the third part contains the address of the next node into the *next* pointer. The forward link in the last node and the backward link in the first node point to a null value. The sample node of the doubly linked list is shown in Figure 10 and an example is shown in Figure 11.

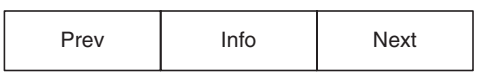


Figure 10 A node of doubly linked list.

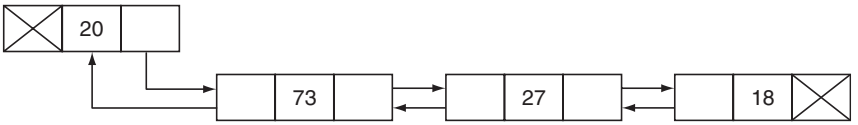


Figure 11 Example of a doubly linked list.

The comparisons of single linked list and doubly linked list are shown in Table 2.

Table 4 Comparisons between single and doubly linked list

S. No.	*Single Linked List*	*Doubly Linked List*
1	In single linked list we can only move in forward direction.	In doubly linked list we can move both in forward and backward direction.
2	We cannot reach to the preceding node from any arbitrary node.	We can reach to the preceding node from any arbitrary node.
3	Insertion and deletion of preceding node will require traversing the linked again from the first node.	Insertion and deletion of preceding node can easily be performed because every node contains the pointer to the preceding node.

Representation of Doubly Linked List

Doubly linked list may also be represented in two ways like other data structures as:

1. Static or sequential or array representation (implementation).
2. Dynamic or pointer or linked representation (implementation).

We have so far discussed the concept of simple linked list, also called as single linked list, in which each node contains one `info` field and one `next` field. The doubly linked list contains the node which is divided into three parts: one part for `info` and two parts for keeping the addresses.

Static or Array Representation (Implementation)

Doubly linked list is also a collection of similar data items like an array, so it can easily be implemented using an array. The data items of the linked list are maintained in the form of nodes divided into three parts, so the linked list may be maintained or represented by three parallel arrays of the same sizes. One array for the actual data items called the *Info* field of the node, second to keep the addresses (index) of the predecessor nodes and third to keep the addresses (index) of the successor nodes as shown in Figure 12. Let us take the array of 12 elements.

The doubly linked list is shown in Figure 12(a). `List[0]` contains the first element 28 of the list pointed by name "`start`" and the `next` field of the `List[0]` contains 6 which means the next element is at `List[6]`. Similarly, the `pre` field of `List[0]` contains –1 which means there is no preceding node to `List[0]`. The `next` field of `List[6]` contains the index value 10 of the third element 32 and `pre` field of `List[6]` contains 0 which means the preceding node of `List[6]` is `List[0]` and so on. The list of elements is shown in Figure 12(b). Since array has the fixed size declared in advance, we have to maintain other linked list of free locations. The list of free locations may be maintained by using the single pointer `next` so that the nodes remain connected. Let us take the first free location at index value 7. Let us take this index value 7 into the pointer named free. The list of free locations is maintained as shown in Figure 12(c). The linked list using array may be implemented in C/C++ by using array of structure as defined here.

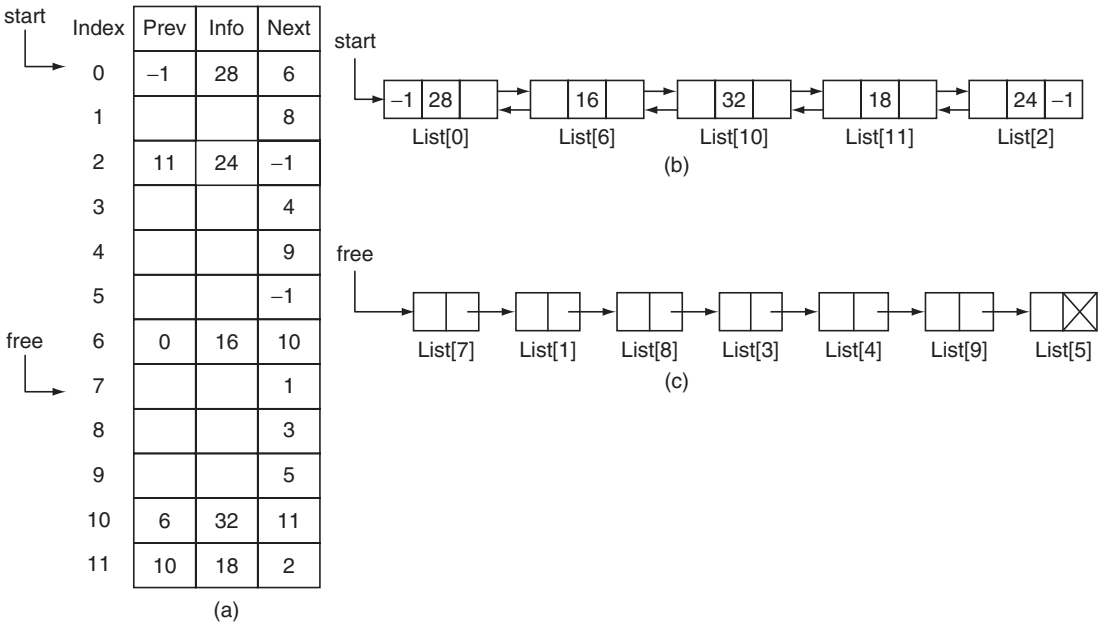


Figure 12 (a) An array representation of a doubly linked list; (b) pictorial representation of doubly linked list using an array; (c) a pictorial representation of free blocks of the array.

In C

```
struct node
{
int prev
int info;
int next;
};
struct node List[12];
```

Remember to use `struct` keyword

In C++

```
struct node
{
    int prev;
    int info;
    int next;
};
node List[12];
```

The `struct` keyword is not required

So `List[12]` is a collection of 12 nodes each containing integer type of `info` and integer type of `prev` and `next` because array index is also an integer and we use array index as a pointer to a node. For example, `List[6]` points to the sixth element of the array that contains data item 16 and `next` address = 10 and `prev` address = 0 as shown in Figure 12(a). In the array representation, the array index starts from 0 so the null value may be represented by –1. Hence if `next` or `prev` field of any node of the linked list contains –1, it means there is no successor or predecessor element in the list, respectively. When the linked list is represented using an array as explained earlier and shown in Figure 12 then `List[n]` represents the node of the linked list pointed by index value *n*. For example `List[6]` is the node of the list pointed by index value 6. That is

0	16	10

`List[n].info` represents the `info` part of the node of the linked list pointed by index value *n*. For example,

```
List[6].info=16
```

`List[n].next` represents the `next` field of the node of the linked list pointed by index value *n*. For example,

```
List[6].next=10
```

`List[n].prev` represents the `prev` field of the node of the linked list pointed by index value *n*. For example,

```
List[6].prev=0
```

Table 5 Representation of `info` and `next` field by an index of the array `List`

Index	`List[index].info`	`List[index].prev`	`Li`
0	28	–1	6
6	16	0	10
10	32	6	11
11	18	10	2
2	24	11	–1

Dynamic or Pointer Representation (Implementation)

The array implementation of a doubly linked list has the same limitations as array implementation of single linked list. The dynamic implementations of linked list do not have size limitations and use space proportional to the actual number of elements of the list. This is represented in computer memory with the help of following structure in C/C++.

In C

```
struct node
{
struct node *prev;
int info;
struct node *next;
};
struct node *start;
```

Remember to use `struct` keyword

In C++

```
struct node
{
node *prev;
int info;
node *next;
};
node *start;
```

The `struct` keyword is not required

A schematic diagram of a linked list with five nodes, where each node is divided into three parts, is shown in Figure 13. In the dynamic representation of the linked list, the nodes are created as and when required by using the dynamic memory allocation methods of C or C++ as already discussed in Chapter 4.

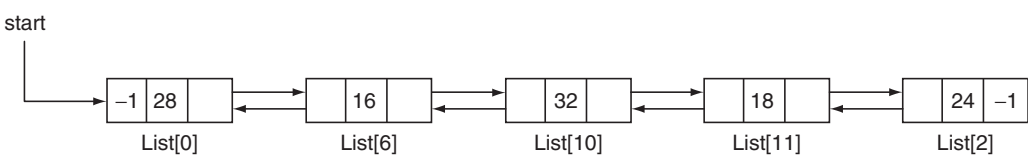


Figure 13 A linked representation of doubly linked list.

Operations on a Doubly Linked List

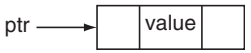
The operations that can be performed on the doubly linked list are same as on the single linked list. We would discuss each of them in detail.

Insertion of the element into the Doubly Linked List at Various Positions

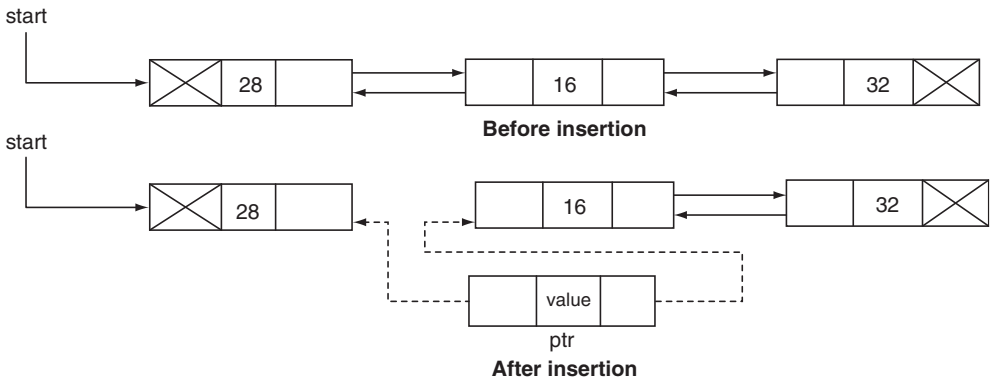
Insertion of node into the doubly linked list requires a free node in which the information can be inserted and then the node can be inserted into the doubly linked list and the pointers are adjusted. Insertion of any element into the linked list requires following steps.

1. Get the free node from the memory pool and assign its address to the `ptr`

2. Assign the value `ptr->info` to the `info` part of the new node pointed by `ptr`



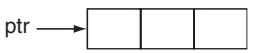
3. If the linked list is empty means `start = NULL` then assign `ptr->next = start`, `start->prev = ptr` and `start = ptr`
4. Otherwise adjust the pointers according to the place of insertion as shown below.



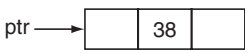
Inserting Node at the Beginning of Doubly Linked List

The process is described here with the help of the following diagrams.

1. Get the free node from the memory pool and assign its address to the `ptr`

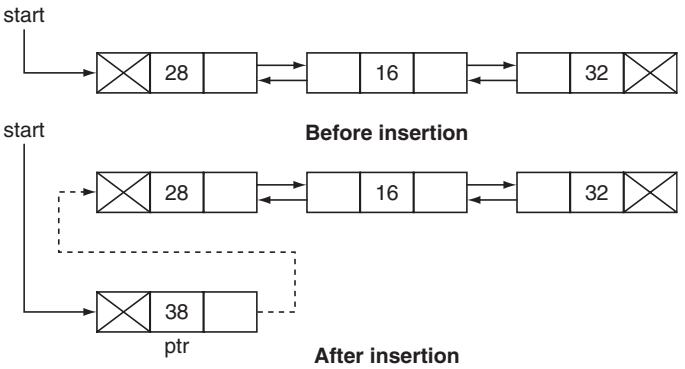


2. Assign the value `ptr->info` (say 38) to the *info* part of the new node pointed by `ptr`



3. If the linked list is empty means `start = NULL` then assign `ptr->next = start`, `start->prev = ptr` and `start = ptr`

4. Otherwise adjust the pointers according to the place of insertion as shown below.



5. Now adjust the pointers so that now the `start` pointer will point to the new node pointed by `ptr` and the `next` pointer of the new node `ptr` will point to the previous `start` node and the `prev` pointer of the new node will contain null. The algorithm is as follows.

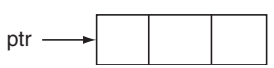
Algorithm 14 `InsertBeg(List, value, start, ptr)`

This algorithm is to insert a *value* into the doubly linked list implemented dynamically. Here `List` is a linked list of nodes; `start` is a variable which holds the address of the first node of the list or null if the `List` is empty. Here, `ptr` is a variable which holds the address of the new node or null if free memory is not sufficient. We want to insert value at the beginning of the `List`.

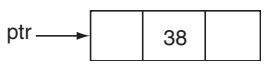
1. Get the first free node into pointer `ptr`
2. `If ptr = NULL`
   a. `Print "Overflow"`
   b. `Exit`
   `End if`
3. Set `ptr->prev = NULL`
4. Set `ptr>info = value`
5. Set `start->prev = ptr`
6. Set `start = ptr`
7. `Exit`

Inserting Node at the End of Doubly Linked List The process is described here with the help of the following diagrams.

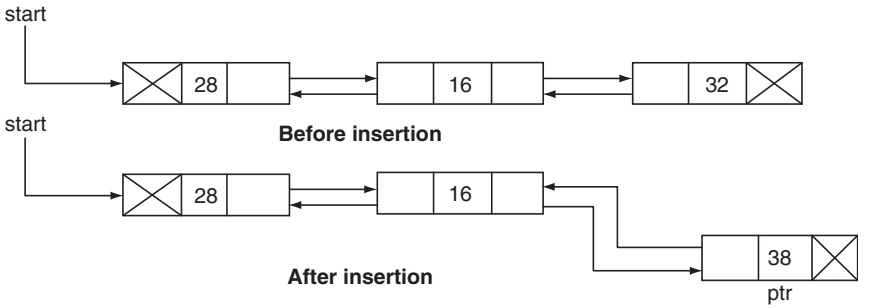
1. Get the free node from the memory pool and assign its address to the `ptr`.



2. Assign the value `ptr->info` (say 38) to the *info* part of the new node pointed by `ptr`.



3. If the linked list is empty means `start = NULL`. Then assign `ptr->next = ptr->prev = NULL` and `start = ptr`.
4. Otherwise traverse the list to find the last node. Now adjust the pointers; so now the `next` pointer of new node will contain null and the `next` field of the second last node will point to the new node and `prev` field of the new node will point to the second last node of the list. The algorithm is given in Algorithm 13.



Algorithm 15 `Insertend(List, value, start, ptr,`

This algorithm is to insert a *value* into the doubly linked list implemented dynamically. Here `List` is a linked list of nodes; *`start`* is the variable which holds the address of the first node of the list or null if the `List` is empty; *`ptr`* is a variable which holds the address of the new node or null if free memory is not sufficient; *`temp`* is a pointer variable to hold the addresses temporarily. We want to insert value at the beginning of the `List`.

1. Get the first free node into pointer `ptr`
2. `If ptr=NULL`
   a. `Print "Overflow"`
   b. `Exit`
   `End if`
3. Set `temp=start`
4. Repeat `while(temp->next!=NULL)`
   a. Set `prev=temp`
   b. `temp=temp->next`
5. Set `ptr>info=value`
6. Set `ptr->prev=prev`
7. Set `prev->next=ptr`
8. Set `ptr->next=NULL`
9. `Exit`

Inserting Node at the Specified Position The node may also be inserted at a specified position in the doubly linked list. We have to traverse the linked list to reach to the specified position; and at the same time we have to keep the address of the previous node so that the pointer can be adjusted. The pictorial representation is shown in Figure 14 for insertion after the specified position.

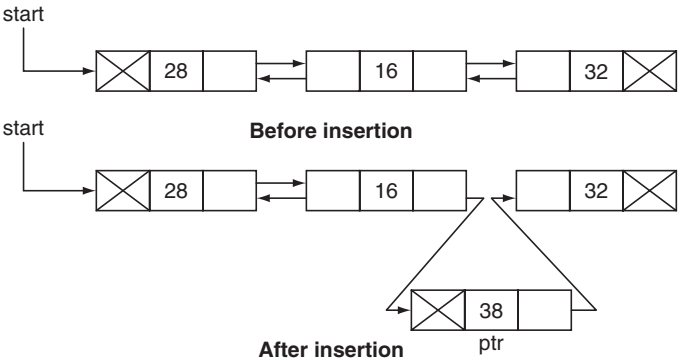


Figure 14 Pictorial representation for insertion after the specified position.

Algorithm 16 `Insertafter(List, value, start, temp)`

This algorithm is used to insert a *value* into the doubly linked list implemented dynamically. Here `List` is a linked list of nodes; `start` is the variable which holds the address of the first node of the list or null if the `List` is empty, `ptr` is a variable which holds the address of the new node or null if free memory is not sufficient, `temp` is a pointer variable to hold the addresses temporarily. We want to insert value after the node containing the given item.

1. Get the first free node into pointer `ptr`
2. `If ptr = NULL`
   a. `Print "Overflow"`
   b. `Exit`
   `End if`
3. Set `temp = start`
4. Repeat `while (temp->info != item)`
   a. Set `prev = temp`
   b. `temp = temp->value`
5. Set `ptr>info = value`
6. Set `ptr->next = prev->next`
7. Set `ptr->prev = prev`

Program 6 A program in C++ to perform insertion at various position in the doubly linked list.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template <class T>
class linked_list
{
struct node
```

```
{
node *prev;
T info;
node *next;
};
node *start, *ptr,*prev,*newptr;
T item;
public:
linked_list()
{
start=NULL;
}
creatlist()
{
char ch;
ptr=getnode();
cout<<"\nEnter value to insert";
cin>>item;
ptr->info=item;
ptr->next=NULL;
ptr->prev=start;
start=ptr;
cout<<"\nWant to insert more nodes: ";
cin>>ch;
while(ch== 'Y'||ch=='y')
{
newptr=getnode();
cout<<"\nEnter value to insert";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
newptr->prev=ptr;
ptr->next=newptr;
ptr=ptr->next;
cout<<"\nWant to insert more nodes: ";
cin>>ch;
}
}
node* getnode()
{
node *newptr1=new node;
return newptr1;
}
void display()
{
```

```
node *ptr1=start;
cout<<"The linked list is: start";
while(ptr1!=NULL)
{
cout<<"<-->"<<ptr1->info;
ptr1=ptr1->next;
}
cout<<endl;
}
void insbeg()
{
node *newptr2=new node;
if(newptr2==NULL)
{
cout<<"\nOVERFLOW";
exit(0);
}
cout<<"\nEnter the item to insert";
cin>>item;
newptr2->info=item;
if(start==NULL)
{
newptr2->next=NULL;
newptr2->prev=NULL;
start=newptr2;
}
else
{
newptr2->next=start;
newptr2->prev=NULL;
start=newptr2;
}
}
void insatend()
{
node *newptr=new node;
if(newptr==NULL)
{
cout<<"\nOVERFLOW";
exit(0);
}
cout<<"\nEnter the item to insert";
cin>>item;
newptr->info=item;
ptr=start;
```

```
while(ptr->next!=NULL)
{
prev=ptr;
ptr=ptr->next;
}
newptr->next=NULL;
ptr->next=newptr;
newptr->prev=ptr;
}
void insafter()
{
int choice;
cout<<"1. To insert at the given position"<
cout<<"2. To replace the given value "
cout<<"Enter your choice: ";
cin>>choice;
switch(choice)
 {
case 1:
int pos;
cout<<"Enter the position ";
cin>>pos;
int i=1;
ptr=start;
while(i<pos)
{
prev=ptr;
ptr=ptr->next;
i=i+1;
}
node *newptr=new node;
if(newptr==NULL)
{
cout<<"\nOVERFLOW";
exit(0);
}
cout<<"Enter the item to insert: ";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
prev->next=newptr;
newptr->prev=prev;
break;
case 2:
cout<<"Enter the value which is to be replace
```

```
cin>>item;
ptr=start;
while(ptr->info!=item)
{
prev=ptr;
ptr=ptr->next;
}
newptr=new node;
if(newptr==NULL)
{
cout<<"OVERFLOW";
exit(0);
}
cout<<"Enter the item to insert:";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
prev->next=newptr;
newptr->prev=prev;
}
}
};
void main()
{
clrscr();
linked_list obj;
char ch;
int choice;
cout<<"*** First create the list of integers<<endl;
obj.creatlist();
do
{
cout<<"1. Display the list of integers ***"<<endl;
cout<<"2. Insertion at the beginning"<<endl;
cout<<"3. Insertion at the end"<<endl;
cout<<"4. Insertion at after specified element"<<endl;
cout<<"Your choice is: ";
cin>>choice;
switch(choice)
{
case 1:
obj.display();
break;
case 2:
obj.insbeg();
```

```
obj.display();
break;
case 3:
obj.insatend();
obj.display();
break;
case 4:
obj.insafter();
obj.display();
break;
}
cout<<"Do you want more insert operations:   "
cin>>ch;
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** First create the list of integers ***
Enter value to insert 1
Want to insert more nodes: y
Enter value to insert 5
Want to insert more nodes: y
Enter value to insert 7
Want to insert more nodes: n
1. Display the list of integers ***
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
Your choice is: 1
The linked list is -> 1 -> 5 -> 7
Do you want more insert operations:   y
1. Display the list of integers ***
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
Your choice is: 2
Enter the item to insert 33
The linked list is -> 33 -> 1 -> 5 -> 7
Do you want more insert operations:   y
1. Display the list of integers ***
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
```

```
Your choice is: 3
Enter the item to insert 55
The linked list is : 3->1->5->7->55
Do you want more insert operations:
Do you want more insert operations:  y
1. Display the list of integers ***
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
Your choice is: 4
1. To insert at the given position
2. To Replace the given value
Enter your choice: 2
Enter the value which is to be replaced:
Enter the item to insert: 88
The Linked list is : 3->1->5->7->88
Do you want more insert operations:  n
```

Deletion of the element into the Doubly Linked List at Various Positions

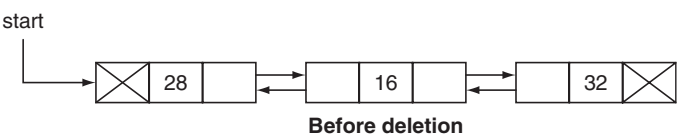
*Deleting Node at the Beginning of Doubly L*The nodes from the doubly linked may also be deleted in a similar manner as they can be deleted from the single linked list. Deletion from the linked list can also be performed in three different positions as:

1. deletion from the beginning of the doubly linked list;
2. deletion from the end;
3. deletion from the specified position.

In this section we would discuss deletion at the beginning of the list. When deletion operation is to be performed we have to check for the underflow condition, that is, if the linked list is empty then we cannot delete the node from the list. This condition is called underflow condition, where `start = NULL`. So deletion from the list requires following operations to be performed:

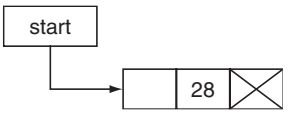
1. Check for the underflow, that is, `start = NULL`
2. Traverse the list to reach to the desired node.
3. Save the value of the deleting node to some variable.
4. Adjust the pointers of the preceding and next nodes.
5. Delete the node and release it back to the memory heap.

Let the list be as given below. We will explain the process in a stepwise manner.



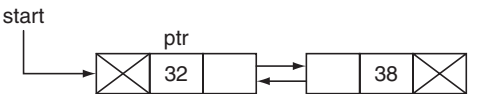
Before deletion

1. If there is only one element in the list, that is, `start->next` then after deleting the node set `start = NULL`



2. If there are more elements in the list, then set `ptr = start->next`

```
ptr->prev = start
start = ptr
```



If the first node is to delete then the `start` pointer will point to the second node of the list, that is, the list will now start from the second node of the list. Hence we have to simply make the pointer `start` equal to the `next` field of the first node in the list and `prev` field of the next node to `start`. Before delting the node we should save the value into some variable. After deleting the node we must make it available for future use so we should return it into the memory heap by `free()` function in C and delete operator in C++.

Algorithm 17 `Delatbeg(List, value, start, ptr)`

Here `List` is a doubly linked list of nodes, *`start`* is the pointer which points to the first node of the linked list or null if the `List` is empty, *`ptr`* is a pointer variable to temporarily keep the address of the node. We want to delete value from the beginning of the doubly linked list.

1. `If start = NULL`
   a. `Print "Underflow (linked list is empty)."`
   b. `Return`
   `End if`
2. `If start->next = NULL` then assign `value = start->info` and set `start = NULL` `Return`
3. Assign `value = start->info`
4. Set `start = start->next` (second node becomes the first node).
5. Set `start->prev = NULL` (release the node pointed by `ptr` to the memory heap).
6. `Exit`

Deleting Node at the End of Doubly Linked List

The algorithm is as follows.

Algorithm 18 Delend(List, value, start, ptr, p

Here List is a doubly linked list of nodes, *sta* is the pointer which points to the first node of the linked list or null if the List is empty, *ptr* is a pointer variable to temporarily keep the address of the node. We want to delete value from the beginning of the doubly linked list.

1. If start = NULL
   a. Print "Underflow (linked list is empty)."
   b. Return
   End if
2. If start -> next = NULL then assign value = start -> info and set start = NULL Return
3. Set ptr = start
4. Repeat while ptr -> next != NULL
   a. prev = ptr
   b. ptr = ptr -> next
5. Assign value = ptr -> info
6. Set prev -> next = NULL
7. Exit

Deleting Node at the Specified Position of Doubly Linked List

The algorithm is as follows.

Algorithm 19 Delpos(List, value, start, ptr, p

Here List is a doubly linked list of nodes, *sta* is the pointer which points to the first node of the linked list or null if the List is empty, *ptr* is a pointer variable to temporarily keep the address of the node. We want to delete value from the specified position *pos* of the doubly linked list.

1. If start = NULL
   a. Print "Underflow (linked list is empty)."
   b. Return
   End if
2. If start -> next = NULL then assign value = start -> info and set start = NULL Return
3. Set ptr = start, i = 1
4. Repeat while (i <= pos)
   a. Set prev = ptr
   b. ptr = ptr -> next
5. Assign value = ptr -> info
6. Set ptr1 = ptr -> next
7. prev -> next = ptr -> next
8. ptr1 -> prev = prev
9. Exit

Program 7 A program in C++ to perform various deletion operations in doubly linked list.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template<class T>
class linked_list
{
struct node
{
node *prev;
T info;
node *next;
};
node *start, *ptr,*prev,*newptr;
T item, value;
public:
linked_list()
{
start=NULL;
}
creatlist()
{
char ch;
ptr=getnode();
cout<<"Enter value to insert";
cin>>item;
ptr->info=item;
ptr->next=NULL;
ptr->prev=NULL;
start=ptr;
cout<<"Want to insert more nodes: ";
cin>>ch;
while(ch=='Y'||ch=='y')
{
newptr=getnode();
cout<<"Enter value to insert";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
newptr->prev=ptr;
ptr->next=newptr;
ptr=ptr->next;
```

```
cout<<"\nWant to insert more nodes:";
cin>>ch;
}
}
node* getnode()
{
node *newptr1=new node;
return newptr1;
}
void display()
{
ptr=start;
cout<<"\nThe linked list is: start";
while(ptr!=NULL)
{
cout<<"<--"<<ptr->info;
ptr=ptr->next;
}
cout<<endl;
}
int delbeg()
{
if(start==NULL)
{
cout<<"underflow";
return 0;
}
ptr=start;
value=start->info;
cout<<"\nThe deleted element is:"
start=start->next;
start->prev=NULL;
delete(ptr);
}
int delend()
{
ptr=start;
if(start->next==NULL)
{
value=start->info;
cout<<"\nThe deleted element is:  "
start=NULL;
delete(ptr);
return 0;
}
```

```
while(ptr->next!=NULL)
{
prev=ptr;
ptr=ptr->next;
}
value=ptr->info;
cout<<"\nThe deleted element is:"<<value;
prev->next=ptr->next;
delete(ptr);
}
void delpos()
{
int choice;
cout<<"1. To delete at the given position"<<endl;
cout<<"2. To delete the given value"<<endl;
cout<<"Enter your choice:";
cin>>choice;
switch(choice)
{
case 1:
int pos;
cout<<"Enter the position"<<endl;
cin>>pos;
int i=1;
ptr=start;
while(i<pos)
{
prev=ptr;
ptr=ptr->next;
i=i+1;
}
value=ptr->info;
cout<<"\nThe deleted element is:"<<value;
node *ptr1=ptr->next;
prev->next=ptr->next;
ptr1->prev=prev;
delete(ptr);
break;
case 2:
cout<<"Enter the value after which you want t
cin>>item;
ptr=start;
while(ptr->info!=item)
{
```

```
prev=ptr;
ptr=ptr->next;
}
value=ptr->info;
cout<<"\nThe deleted element is:"<<value;
prev->next=ptr->next;
ptr1->prev=prev;
delete(ptr);
}
}
};
void main()
{
clrscr();
linked_list obj;
char ch;
int choice;
cout<<"***Create and delete from the doub
integers"<<endl;
obj.creatlist();
obj.display();
do
{
cout<<"1. Deletion from the beginning"<<endl;
cout<<"2. Deletion at the end"<<endl;
cout<<"3. Deletion at the specified posit
element"<<endl;
cout<<"Your choice is:";
cin>>choice;
switch(choice)
{
case 1:
obj.delbeg();
obj.display();
break;
case 2:
obj.delend();
obj.display();
break;
case 3:
obj.delpos();
obj.display();
break;
}
```

```
cout<<"Do you want more insert operations:";
cin>>ch;
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** Create and delete from the doubly link
***
Enter value to insert 1
Want to insert more nodes: y
Enter value to insert 4
Want to insert more nodes: y
Enter value to insert 2
Want to insert more nodes: y
Enter value to insert 8
Want to insert more nodes: n
The linked list is 1 -> 4 -> 2 -> 8

1. Deletion from the beginning
2. Deletion at the end
3. Deletion at the specified position or o

Your choice is: 1
 The deleted item is:  1
The linked list is 4 -> 2 -> 8
Do you want more insert operations:  y

1. Deletion from the beginning
2. Deletion at the end
3. Deletion at the specified position or o
Your choice is: 2
 The deleted item is:  8
The linked list is 4 -> 2 start
Do you want more insert operations:  y

1. Deletion from the beginning
2. Deletion at the end
3. Deletion at the specified position or o
Your choice is: 3
1. To delete at the given position
2. To delete the given value

Enter your choice: 2
Enter the value after which you want to del
 The deleted item is:  2
The linked list is 4 start
Do you want more insert operations:  n
```

6.8 Circular Linked List

The drawback of both simple linked list and doubly linked list is that we cannot move directly to the first node after reaching the last node of the list. In the circular-linked list the first and last nodes are linked together, that is, the *next* pointer of the last node points to the first node instead of containing null. Hence circular linked list contains valid addresses instead of null. The circular linked list may be a simple circular linked list or it may be a doubly circular linked list.

In the doubly circular linked list the *prev* pointer of the first node points to the last node and *next* pointer of the last node points to the first node. Hence circular linked list works for both single and doubly linked lists. A typical example of simple circular linked list is shown in Figure 15 and an example of double circular linked list is shown in Figure 16. The end of the list is checked by comparing the next pointer field of any node with the address of the first node.

Circular linked list is used to implement the circular queue which is used to maintain the process for execution in the time sharing operating system. In the time sharing operating system, each process is allotted a time stamp for execution or keeping the CPU busy. They then have to wait for next turn in a circular queue. Since all the nodes in a circular linked list are connected in a circle so any node can be accessed from any other node.

S. No.	*Single Linked List*	*Circular Linked List*
1	In single linked list, we can only move in forward direction.	In circular linked list, we can move in forward direction only with the exception that last node points to the first node.
2	We cannot reach to the first node directly of the linked list.	We can reach to the first node directly from the last node.
3	Insertion and deletion of preceding node will require traversing the linked list afresh again.	Insertion and deletion of preceding node can be performed because last node follows first node and so we can traverse the list once again by reaching to the last node.
4	`struct node { int info; struct node *next; }; struct node *start;` where `start->next` in the beginning is `NULL`	`struct node { int info; struct node *next; }; struct node *start;` where `start->next` in the beginning is `start`

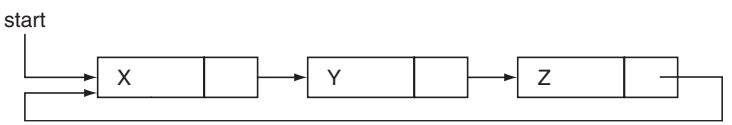


Figure 15 Simple circular linked list.

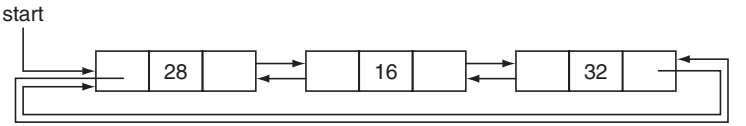


Figure 16 Circular doubly linked list.

6.9 Representation of Circular Linked List

Circular linked list may also be represented in two ways:

1. Static or sequential or array representation (implementation).
2. Dynamic or pointer or linked representation (implementation).

We are giving the representation of circular doubly linked list here and the representation of simple circular linked list is left as an exercise for the students.

Static or Array Representation (Implementation)

Circular doubly linked list is also a collection of similar data items like an array, so it can easily be implemented using an array. The data items of the linked list are maintained in the form of nodes divided into three parts and so the linked list may be maintained or represented by three parallel arrays of the same sizes as shown below in Figure 17. Let us take an array of 12 elements. The only difference is that the `next` field of the last node does not contain –1 (to indicate null). Rather it contains the index of the first node. Similarly, the `pref` field of the first node does not contain –1. Rather it contains the index value of the last node as shown in bold numbers.

The circular doubly linked list is shown in Figure 17(a). `List[0]` contains the first element 28 of the list pointed by name 'start' and the `next` field of the `List[0]` contains 6 which means the next element is at `List[6]`. Similarly, the `pref` field of `List[0]` contain 2 which means node[0]

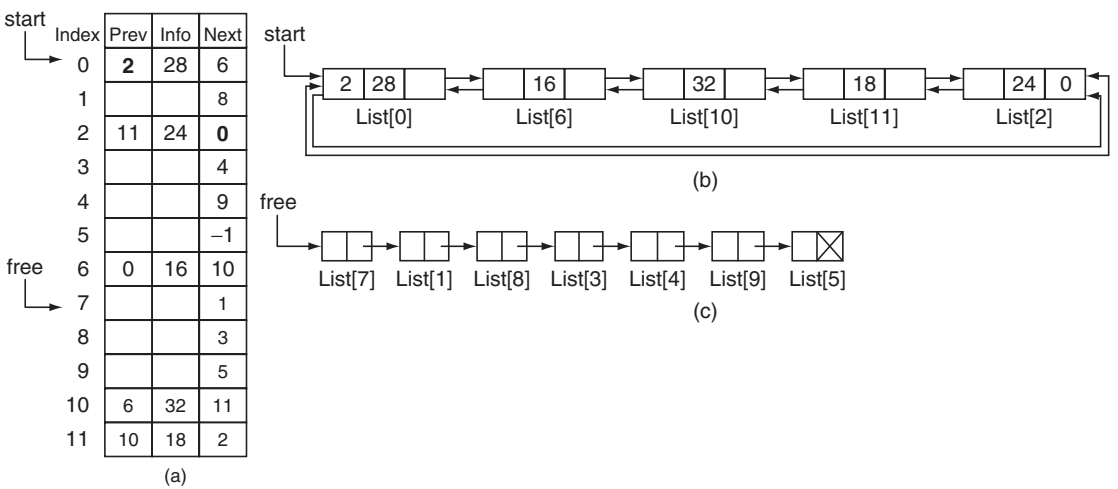


Figure 17 (a) An array representation of a circular doubly linked list; (b) pictorial representation of circular doubly linked list using an array; (c) a pictorial representation of free blocks of the array.

is preceded by node[2]. Similarly, we can see that the `next` field of node[2], the last node, contains 0 means last node is followed by first node.

Dynamic or Pointer Representation (Implementation)

This is represented in computer memory just like the doubly linked list with the help of the following structure in C/C++.

In C

```
struct node
{
struct node *prev;
int info;
struct node *next;
};
struct node *start;
```

Remember to use `struct` keyword

In C++

```
struct node
{
node *prev;
int info;
node *next;
};
node *start;
```

The `struct` keyword is not required

During implementation we have take care that the *`next`* field of the last node points to the first node and the *`prev`* field of the first node points to the last node as already discussed.

Operations on a Circular Linked List

The different operations that can be performed on the simple linked list or doubly linked can also be performed on the circular linked list with one change that the last node follows the first node and first node is preceded by last node. We are simply giving the algorithms for different operations on the circular linked list here.

Program 8 A program to perform various operations on the circular simple linked list.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template<class T>
class linked_list
{
struct node
{
T info;
node *next;
};
node *start, *ptr,*ptr1,*newptr;
T item;
public:
linked_list()
```

```
{
start=NULL;
}
void creatlist() // just like inserting an
beginning
{
char ch;
if(start==NULL)
{
ptr=getnode();
cou<<"Enter value to insert";
cin>>item;
ptr->info=item;
ptr->next=ptr;
start=ptr;
}
else
{
ptr=start;
while(ptr->next!=start)
{
ptr=ptr->next;
}
newptr=getnode();
cou<<"Enter value to insert";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
}
cou<<"Want to insert more nodes:";
cin>>ch;
if(ch=='Y'||ch=='y')
{
creatlist();
}
}
node* getnode()
{
node *newptr1=new node;
return newptr1;
}
void display()
{
node *ptr1=start;
cou<<"The linked list is:start";
```

```
while(ptr1->next!=start)
{
cout<<"--"<<ptr1->info;
ptr1=ptr1->next;
}
cout<<"--"<<ptr1->info<<"--"<<ptr1->next->info;
cout<<endl;
}
void insatend()
{
node *newptr=new node;
cout<<"\nEnter the item to insert";
cin>>item;
newptr->info=item;
ptr=start;
while(ptr->next!=start) // instead of NULL
list
{
ptr=ptr->next;
}
newptr->next=ptr->next;
ptr->next=newptr;
}
void insafter()
{
int choice;
cout<<"1. To insert after the given position
cout<<"2. To insert after given value"
cout<<"Enter your choice:";
cin>>choice;
switch(choice)
{
case 1:
int pos;
cout<<"Enter the position
cin>>pos;
int i=1;
ptr=start;
while(i<pos)
{
ptr=ptr->next;
i=i+1;
}
node *newptr=new node;
cout<<"Enter the item to insert:";
```

```
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
break;
case 2:
cout<<"Enter the value after which you want t
cin>>item;
ptr=start;
while(ptr->info!=item)
{
ptr=ptr->next;
}
newptr=new node;
cout<<"Enter the item to insert:";
cin>>item;
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
}
}
};
void main()
{
clrscr();
linked_list obj;
char ch;
int choice;
cout<<"*** Create and display the li
obj.creatlist();
obj.display();
do
{
cout<<"1. Insertion at end";
cout<<"2. Insertion after specified e
cout<<"Your choice is:";
cin>>choice;
switch(choice)
{
case 1:
obj.insatend();
obj.display();
break;
```

```
case 2:
obj.insafter();
obj.display();
break;
}
cout<<"Do you want more insert operations:";
cin>>ch;
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** Create and display the list of intege
Enter value to insert1
Want to insert more nodes: y
Enter value to insert2
Want to insert more nodes: y
Enter value to insert3
Want to insert more nodes: n
The linked list is: start-->1-->2-->3-->
1. Insertion at end
2. Insertion at after specified element
Your choice is: 1
Enter the item to insert5
The linked list is: start-->1-->2-->3-->
Do you want more insert operations: y
1. Insertion at end
2. Insertion at after specified element
Your choice is: 2
1. To insert after the given position
2. To insert after given value
Enter your choice: 1
Enter the position 2
Enter the item to insert: 8
The linked list is: start-->1-->2-->8-->
Do you want more insert operations: y
1. Insertion at end
2. Insertion at after specified element
Your choice is: 2
1. To insert after the given position
2. To insert after given value
Enter your choice: 2
Enter the value after which you want to i
Enter the item to insert: 4
The linked list is: start-->1-->2-->8-->
Do you want more insert operations: n
```

6.10 Josephus Problem

This is an application of queue using circular linked list. In such a queue one can quickly reach from one node to another. In this problem a circular queue is used for the solution. The problem is that the *A group of soldiers is surrounded by overwhelming enemy force and enemy is coming nearer and can attack the soldiers any time and only one horse was available for escape. The soldiers have to escape from that place, so they played a trick. The soldiers formed a circle and each soldier was given a token in his hand. Each soldier was supposed to pass the token to his neighbor. The passing of the token starts from the soldier whose name is picked up from the sack and it should be in clockwise manner around the circle.* The idea is that the one having a token can catch the horse and escape. Thus the token is moving in a circle and one by one the lucky soldiers escape.

6.11 Applications of Linked List

There are various applications of the linked list including implementation of sparse arrays, manipulation of sparse arrays, manipulation of polynomials, implementation of circular queue, implementation of hashing techniques, etc. We will discuss some of them in this section.

Manipulation of Polynomials

Different operations like addition, multiplication, subtraction and division of polynomials can easily be performed using the linked list. First let us see the representation of the polynomial $f(x)$ and $g(x)$ in Figure 18, where

$$f(x) = ax^3 + bx^2 + c$$
$$g(x) = mx^4 + nx^2 + px + q$$

They can easily be represented as a collection of nodes of the linked list, where each node is divided into three parts: coefficient, exponent and `next` (the address of the next node).

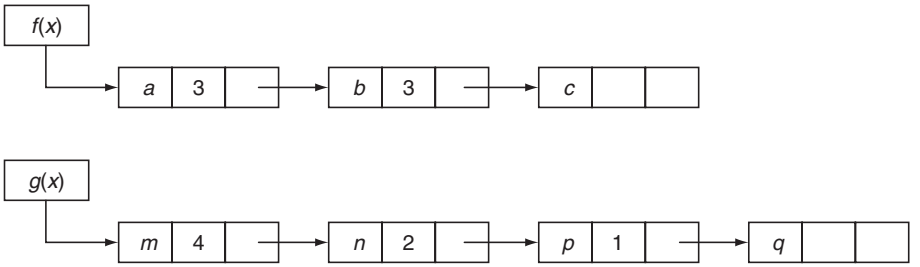


Figure 18 Polynomial representation.

Array Representation

```
struct term
{
int info;
int coff;
int next;
}
term Polynomial[3];
```

Dynamic Representation

```
struct term
{
    int info;
    int coff;
    term *next;
};
struct term *sta
```

If the polynomial consists of more than one variable, then the node will have separate part for different variables along with `i n f o` and `n e x t` part of the node as shown in Figure 19 for $f(x) = ax^3 + by^2 + c$ and $g(x) = mx^4y + nyz^2 + px + q$.

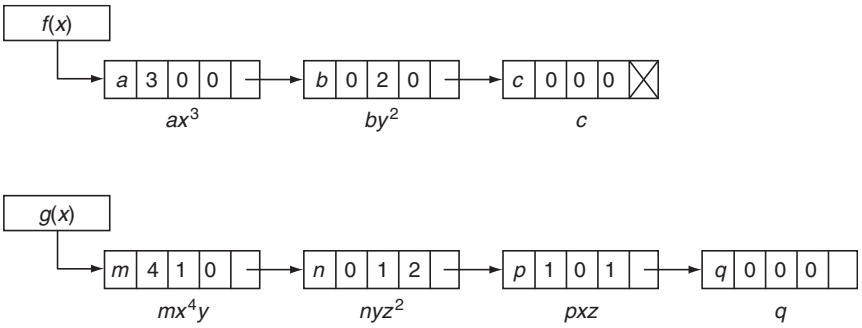


Figure 19 Representation of polynomial consisting of more than one variable.

They can easily be represented as a collection of nodes of the linked list, that is, where each node is divided into five parts: coefficient, exponentx, exponent, exponentz and `n e x t` (the address of the next node).

Array Representation	Dynamic Representation
`s t r u c t  t e r m`	`s t r u c t  t e r m`
`{`	`{`
`i n t  i n f o ;`	`i n t  i n f o ;`
`i n t  c o f f x ;`	`i n t  c o f f ;`
`i n t  c o f f y ;`	`i n t  c o f f y ;`
`i n t  c o f f z ;`	`i n t  c o f f z ;`
`i n t  n e x t ;`	`t e r m  * n e x t ;`
`}`	`} ;`
`t e r m  P o l y n o m i a l [ 3 ] ;`	`s t r u c t  t e r m  * s`

Operations on P ol y nomial s

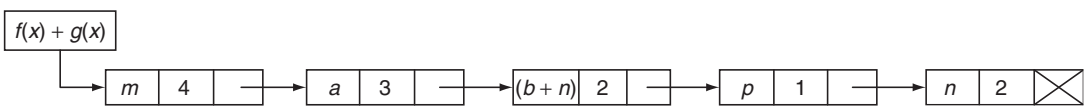
Let us assume four polynomials:

1. $f(x) = ax^3 + bx^2 + c$
2. $g(x) = mx^4 + nx^2 + px + q$
3. $f1(xyz) = ax^3 + by^2 + c$
4. $g1(xyz) = mx^4y + nyz^2 + px + q$

Addition

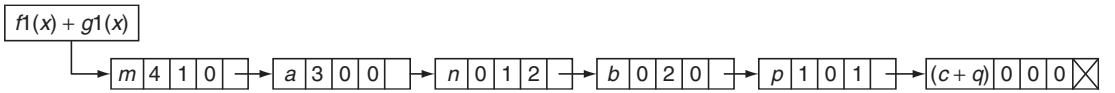
If two polynomials are defined on the same variable then the addition is performed as follows:

$$f(x) + g(x) = mx^4 + ax^3 + (b + n)x^2 + px + (c + q)$$



If two polynomials are defined on more than one variable then the addition is performed as given below:

$$f1(xyz) + g1(xyz) = mx^4y + ax^3 + nyz^2 + by^2 + pxz + (q + c)$$



So the addition of the polynomial is performed by comparing the exponent of the terms of first polynomial with the exponent of the terms of the second polynomial. If they match then add their coefficient, otherwise compare it with the other term of the second polynomial and so on.
The polynomial can be represented using linked list by following these steps:

1. First get the free node from the memory heap.
2. Assign the values to the different parts of the newly created node.
3. Now adjust the pointer so that the newly created node can be placed into its proper position.

Program 9 Write a program in C++ to add two polynomials of single variables.

```
#include<iostream.h>
#include<malloc.h>
#include<conio.h>
#include<process.h>
struct node
{
int coff;
int exp;
node *next;
};
main()
{
clrscr();
node *first,*second,*result;
void create(node *);
void display(node *);
void sum(node *, node *, node *);
char ch;
do
{
cout<<"\Enter the first term number "
first=new node;
if(first==NULL)
{
cout<<"unable to create node";
```

```
exit(1);
}
create(first);
cout<<"\Enter the second number";
second=new node;
if(second==NULL)
{
cout<<"unable to create node";
exit(1);
}
create(second);
cout<<"\n*** Display the numbers ***";
cout<<"\nThe First polynomial is :";
display(first);
cout<<"\nThe second polynomial is :";
display(second);
sum(first,second, result);
cout<<"\nThe sum of two polynomial is:";
display(result);
cout<<"\nDo you want to add more polynomials
ch=getch();
}
while(ch=='y'||ch=='Y');
}
void create(node *ptr)
{
char ch;
do
{
cout<<"\nEnter the coefficient : ";
cin>>ptr->coff;
cout<<endl;
cout<<"Enter the exponent:";
cin>>ptr->exp;
ptr->next=new node;
ptr=ptr->next;
ptr->next=NULL;
cout<<" Do you want to insert more coefficie
ch=getch();
cout<<endl;
}
while(ch=='y'||ch=='Y');
}
```

```
void display(node *ptr)
{
cout<<endl;
while(ptr->next!=NULL)
{
cout<<ptr->coff<<"fxf^"<<ptr->exp;
ptr=ptr->next;
if(ptr->next!=NULL)
cout<<"+";
}
}
void sum(node *first,node *second, node *re
{
while(first->next && second->next)
{
if(first->exp>second->exp)
{
result->exp=first->exp;
result->coff=first->coff;
first=first ->next;
}
else if(first->exp<second->exp)
{
result->exp=second->exp;
result->coff=second->coff;
second=second->next;
}
else
{
result->exp=first->exp;
result->coff=first->coff+second->coff;
first=first->next;
second=second->next;
}
result->next=new node;
result=result->next;
result->next=NULL;
}
while(first->next||second->next)
{
if(first->next)
{
result->exp=first->exp;
```

```
result->coff=first->coff;
first=first->next;
}
if(second->next)
{
result->exp=second->exp;
result->coff=second->coff;
second=second->next;
}
result->next=new node;
result=result->next;
result->next=NULL;
}
}
```

Output

```
Enter the first number
Enter the coefficient :  2
Enter the exponent:3
Do you want to insert more coefficients(y
Enter the coefficient :  2
Enter the exponent:4
Do you want to insert more coefficients(y
Enter the coefficient :  2
Enter the exponent:  5
Do you want to insert more coefficients(y
Enter the second number
Enter the coefficient :  3
Enter the exponent:2
Do you want to insert more coefficients(y
*** Display the numbers ***
The First polynomial is :
2x^+3x^+4x^5
The second polynomial is :
3x^2
The sum of two polynomial is :
2x^+3x^+4x^+3x^2
Do you want to add more polynomials: n
```

Solved Examples

Example 1 *Write a program in C to perform deletion at various positions in the single linked list.*

Solution

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
char info;
struct node *next;
};
struct node *start=NULL, *ptr,*prev,*newptr
char item, value;
struct node* getnode()
{
struct node *newptr1=(struct node *)malloc(
node));
return newptr1;
}
void creatlist()
{
char ch;
ptr=getnode();
printf("\nEnter value to insert:");
item=getche();
ptr->info=item;
ptr->next=start;
start=ptr;
printf("\nWant to insert more nodes:");
ch=getche();
while(ch=='Y'||ch=='y')
{
newptr=getnode();
printf("\nEnter value to insert:");
item=getche();
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
ptr=ptr->next;
printf("\nWant to insert more nodes:");
ch=getche();
}
}
```

```
void display()
{
struct node *ptr1=start;
printf("\nThe linked list is:  start");
while(ptr1!=NULL)
{
printf("%c",ptr1->info);
ptr1=ptr1->next;
}
printf("\n");
}
int delbeg()
{
if(start==NULL)
{
printf("\nunderflow");
return 0;
}
ptr=start;
value=start->info;
printf("\nThe deleted item is: %c",value);
start=start->next;
free(ptr);
}
int delend()
{
ptr=start;
if(ptr->next==NULL)
{
value=ptr->info;
printf("\nThe deleted item is:  %c",value);
free(ptr);
return 0;
}
while(ptr->next!=NULL)
{
prev=ptr;
ptr=ptr->next;
}
value=ptr->info;
printf("\nThe deleted item is: %c",value);
prev->next=ptr->next;
free(ptr);
}
```

```
void delafter()
{
struct node *ptr3,*ptr4;
int choice;
int pos;
int i=1;
printf("\nEnter the position:");
scanf("%d",&pos);
ptr3=start;
while(i<=pos)
{
prev=ptr3;
ptr3=ptr3->next;
i=i+1;
}
value=ptr3->info;
printf("\nThe deleted item is:  %c",value);
prev->next=ptr3->next;
free(ptr3);
}
void main()
{
char ch;
int choice;
clrscr();
printf("\n*** Create and display the list
creatlist();
display();
do
{
printf("\n1. Deletion at beginning\n");
printf("2. Deletion at end\n");
printf("3. Deletion after specified element
printf("Your choice is:");
scanf("%d",&choice);
switch(choice)
{
case 1:
delbeg();
display();
break;
case 2:
delend();
display();
break;
```

```
case 3:
delafter();
display();
break;
}
printf("Do you want more deletions:");
ch=getche();
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** Create and display the list of integer
Enter value to insert:  a
Want to insert more nodes:  y
Enter value to insert:  b
Want to insert more nodes:  y
Enter value to insert:  c
Want to insert more nodes:  y
Enter value to insert:  d
Want to insert more nodes:  n
The linked list is:  start--> a--> b--> c
1. Deletion at beginning
2. Deletion at end
3. Deletion after specified element
Your choice is:  1
The deleted item is: a
The linked list is:  start--> b--> c--> d
Do you want more deletions:  y
1. Deletion at beginning
2. Deletion at end
3. Deletion after specified element
Your choice is:  3
Enter the position: 1
The deleted item is:  c
The linked list is:  start--> b--> d
Do you want more deletions:  y
1. Deletion at beginning
2. Deletion at end
3. Deletion after specified element
Your choice is:  2
The deleted item is:  d
The linked list is:  start--> b
```

```
Do you want more deletions:   y
1. Deletion at beginning
2. Deletion at end
3. Deletion after specified element
Your choice is:   1
The deleted item is: b
The linked list is:   start
Do you want more deletions:   y
1. Deletion at beginning
2. Deletion at end
3. Deletion after specified element
Your choice is: 1
underflow
The linked list is:   start
Do you want more deletions n:
```

Example 2 *Write a program in C to perform insertion operations at various positions in the single linked list.*

Solution

```
#incl<stdio.h>
#incl<conio.h>
#incl<process.h>
struct node
{
int info;
struct node *next;
};
typedef struct node *nodeptr;
nodeptr start=NULL, ptr,newptr;
int item;
nodeptr getnode()
{
nodeptr newptr1=(nodeptr)malloc(sizeof(stru
return newptr1;
}
void creatlist()
{
char ch;
ptr=getnode();
printf("Enter value to insert");
scanf("%d",&item);
ptr->info=item;
```

```
ptr->next=start;
start=ptr;
printf("Want to insert more nodes:");
ch=getche();
while(ch=='Y'||ch=='y')

{
newptr=getnode();
printf("\nEnter value to insert");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
ptr=ptr->next;
printf("\nWant to insert more nodes:");
ch=getche();
}
}
void display()
{
nodeptr ptr1=start;
printf("\nThe linked list is:  start");
while(ptr1!=NULL)
{
printf("-->  %d",ptr1->info);
ptr1=ptr1->next;
}
printf("\n");
}
void insbeg()
{
nodeptr newptr2=(nodeptr)malloc(sizeof(st
printf("\nEnter the item to insert");
scanf("%d",&item);
newptr2->info=item;
newptr2->next=start;
start=newptr2;
}
void insatend()
{
nodeptr newptr=(nodeptr)malloc(sizeof(str
printf("\nEnter the item to insert");
scanf("%d",&item);
newptr->info=item;
ptr=start;
```

```
while(ptr->next!=NULL)
{
ptr=ptr->next;
}
newptr->next=ptr->next;
ptr->next=newptr;
}
void insafter()
{
int pos,i=1;
nodeptr newptr=(nodeptr)malloc(sizeof(struc
printf("\nTo insert after the given positio
printf("Enter the position:");
scanf("%d",&pos);
ptr=start;
while(i<pos)
{
ptr=ptr->next;
i=i+1;
}
printf("\nEnter the item to insert:");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
}
void main()
{
char ch;
int choice;
clrscr();
printf("*** Create and display the list of
creatlist();
display();
do
{
printf("1. Insertion at beginning\n");
printf("2. Insertion at end\n");
printf("3. Insertion at after specified ele
printf("Your choice is:");
scanf("%d",&choice);
switch(choice)
{
case 1:
insbeg();
```

```
display();
break;
case 2:
insatend();
display();
break;
case 3:
insafter();
display();
break;
}
printf("Do you want more insert operations
ch=getche();
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** Create and display the list of intege
Enter value to insert: 3
Want to insert more nodes: y
Enter value to insert: 7
Want to insert more nodes: y
Enter value to insert: 9
Want to insert more nodes: n
The linked list is:  start-->  3-->  7-->
1. Insertion at beginning
2. Insertion at end
3. Insertion at after specified element
Your choice is: 1
Enter the item to insert: 22
The linked list is:  start-->  22-->  3--
Do you want more insert operations: y
1. Insertion at beginning
2. Insertion at end
3. Insertion at after specified element
Your choice is: 2
Enter the item to insert: 25
The linked list is:  start-->  22-->  3--
Do you want more insert operations: y
1. Insertion at beginning
2. Insertion at end
3. Insertion at after specified element
```

```
Your choice is: 3
To insert after the given position
Enter the position : 2
Enter the item to insert: 44
The linked list is: start-->  22-->  3-->
Do you want more insert operations: n
```

Example 3 *Write a program in C to perform insertion at various positions in the doubly linked list.*

Solution

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
struct node *prev;
char info;
struct node *next;
};
typedef struct node *nodeptr;
nodeptr start=NULL, ptr,ptr1,prev,newptr;
char item;
nodeptr getnode()
{
nodeptr newptr1=(nodeptr)malloc(sizeof(stru
return newptr1;
}
void creatlist()
{
char ch;
ptr=getnode();
printf("\nEnter value to insert");
scanf("%d",&item);
ptr->info=item;
ptr->next=NULL;
ptr->prev=start;
start=ptr;
printf("\nWant to insert more nodes:");
ch=getche();
while(ch=='Y'||ch=='y')
```

```
{
newptr=getnode();
printf("\nEnter value to insert");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
newptr->prev=ptr;
ptr->next=newptr;
ptr=ptr->next;
printf("\nWant to insert more nodes:");
ch=getche();
}
}
void display()
{
nodeptr ptr1=start;
printf("The linked list is: start");
while(ptr1!=NULL)
{
printf("<-%d",ptr1->info);
ptr1=ptr1->next;
}
printf("\n");
}
void insbeg()
{
nodeptr newptr2=(nodeptr)malloc(sizeof(st
if(newptr2==NULL)
{
printf("\nOVERFLOW");
exit(0);
}
printf("\nEnter the item to insert");
scanf("%d",&item);
newptr2->info=item;
if(start==NULL)
{
newptr2->next=NULL;
newptr2->prev=NULL;
start=newptr2;
}
else
```

```
{
newptr2->next=start;
newptr2->prev=NULL;
start=newptr2;
}
}

void insatend()
{
nodeptr newptr=(nodeptr)malloc(sizeof(struc
if(newptr==NULL)
{
printf("\nOVERFLOW");
exit(0);
}
printf("\nEnter the item to insert");
scanf("%d",&item);
newptr->info=item;
ptr=start;
while(ptr->next!=NULL)
{
prev=ptr;
ptr=ptr->next;
}
newptr->next=NULL;
ptr->next=newptr;
newptr->prev=ptr;
}
void insafter()
{
int pos,i=1,choice;
nodeptr newptr=(nodeptr)malloc(sizeof(struc
printf("1. To insert at the given position\
printf("2. To replace the given value \n");
printf("Enter your choice:");
scanf("%d",&choice);
switch(choice)
 {
case 1:
printf("Enter the position \n");
scanf("%d",&pos);
ptr=start;
while(i<pos)
{
```

```
prev=ptr;
ptr=ptr->next;
i=i+1;
}
if(newptr==NULL)
{
printf("\nOVERFLOW");
exit(0);
}
printf("Enter the item to insert:");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
prev->next=newptr;
newptr->prev=prev;
break;
case 2:
printf("Enter the value which is to be rep
scanf("%d",&item);
ptr=start;
while(ptr->info!=item)
{
prev=ptr;
ptr=ptr->next;
}
newptr=(nodeptr)malloc(sizeof(struct node)
if(newptr==NULL)
{
printf("\nOVERFLOW");
exit(0);
}
printf("\nEnter the item to insert:");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
prev->next=newptr;
newptr->prev=prev;
}
}
void main()
{
char ch;
int choice;
clrscr();
```

```
printf("*** First create the list of intege
creatlist();
do

{
printf("1. Display the list of integers \n"
printf("2. Insertion at beginning\n");
printf("3. Insertion at end\n");
printf("4. Insertion at after specified ele
printf("Your choice is:");
scanf("%d",&choice);
switch(choice)
{
case 1:
display();
break;
case 2:
insbeg();
display();
break;
case 3:
insatend();
display();
break;
case 4:
insafter();
display();
break;
}
printf("\nDo you want more insert operation
ch=getche();
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** First create the list of integers ***
Enter value to insert 1
Want to insert more nodes: y
Enter value to insert 4
Want to insert more nodes: n
1. Display the list of integers
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
```

```
Your choice is: 1
The linked list is : 1 4
Do you want more insert operations: y
1. Display the list of integers
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
Your choice is: 2
Enter the item to insert 44
The linked list is : 44 1 4
Do you want more insert operations:
Your choice is: 3
Enter the item to insert 66
The linked list is : 44 1 4 -> 66
Do you want more insert operations: y
1. Display the list of integers
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
Your choice is: 4
1. To insert at the given position
2. To replace the given value
Enter your choice:1
Enter the position 2
Enter the item to insert: 55
The linked list is : 44 55 1 4 -> 66
Do you want more insert operations: y
1. Display the list of integers
2. Insertion at beginning
3. Insertion at end
4. Insertion at after specified element
Your choice is: 4
1. To insert at the given position
2. To replace the given value
Enter your choice: 2
Enter the value which is to be replaced :
Enter the item to insert: 88
The linked list is : 44 55 1 4 -> 88 -> 66
Do you want more insert operations: n
```

Example 4 *Write a program in C to create, insert at end, insert after specified position, insert after specified value and search an element in a circular linked list.*

Solution

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
int info;
struct node *next;
};
typedef struct node *nodeptr;
nodeptr start=NULL, ptr,ptr1,newptr;
int item;
nodeptr getnode()
{
nodeptr newptr1=(nodeptr)malloc(sizeof (str
return newptr1;
}
void creatlist()
{
char ch;
if(start==NULL)
{
ptr=getnode();
printf("\nEnter value to insert");
scanf("%d",&item);
ptr->info=item;
ptr->next=ptr;
start=ptr;
}
else
{
ptr=start;
while(ptr->next!=start)
{
ptr=ptr->next;
}
newptr=getnode();
printf("\nEnter value to insert");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
```

```
ptr->next=newptr;
}
printf("\nDo you want to insert more nodes
ch=getche();
if(ch=='Y'||ch=='y')
{
creatlist();
}
}
void display()
{
nodeptr ptr1=start;
printf("\nThe linked list is:  start");
while(ptr1->next!=start)
{
printf("-->  %d",ptr1->info);
ptr1=ptr1->next;
}
printf("-->  %d-->%d",ptr1->info,ptr1->next
printf("\n");
}
void insatend()
{
nodeptr newptr=getnode();
printf("\nEnter the item to insert");
scanf("%d",&item);
newptr->info=item;
ptr=start;
while(ptr->next!=start) // instead of NULL
list
{
ptr=ptr->next;
}
newptr->next=ptr->next;
ptr->next=newptr;
}
void insafter()
{
int pos;int i=1;
int choice;
nodeptr newptr=getnode();
printf("1. To insert after the given posit
printf("2. To insert after given value \n"
printf("Enter your choice:");
```

```
scanf("%d",&choice);
switch(choice)
  {
case 1:
printf("\nEnter the position \n");
scanf("%d",&pos);
ptr=start;
while(i<pos)
{
ptr=ptr->next;
i=i+1;
}
printf("\nEnter the item to insert:");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
break;
case 2:
printf("\nEnter the value after which you wa
scanf("%d",&item);
ptr=start;
while(ptr->info!=item)
{
ptr=ptr->next;
}
newptr=getnode();
printf("\nEnter the item to insert:");
scanf("%d",&item);
newptr->info=item;
newptr->next=ptr->next;
ptr->next=newptr;
}
}
void search()
{
int i=1,flag=0;
printf("\nEnter an element to search:");
scanf("%d",&item);
ptr=start;
while(ptr->next!=start)
{
if(ptr->info==item)
{
flag=1;
```

```
break;
}
i++;
ptr=ptr->next;
}
if(flag==1)
{
printf("\n%d:   is found at   : %d", item,i
}
}
void main()
{
char ch;
int choice;
clrscr();
printf("\n*** Create and display the list
creatlist();
display();
do
{
printf("\n1. Insertion at end\n");
printf("2. Insertion after specified eleme
printf("3. Search an element \n");
printf("Your choice is:");
scanf("%d",&choice);
switch(choice)
{
case 1:
insatend();
display();
break;
case 2:
insafter();
display();
break;
case 3:
search();
}
printf("\nDo you want more insert operatio
ch=getche();
}while(ch=='Y'||ch=='y');
getch();
}
```

Output

```
*** Create and display the list of integers
Enter value to insert 1
Do you want to insert more nodes: y
Enter value to insert 2
Do you want to insert more nodes: y
Enter value to insert 3
Do you want to insert more nodes: n
The linked list is:  start--> 1--> 2--> 3--
1. Insertion at end
2. Insertion after specified element
3. Search an element
Your choice is: 1
Enter the item to insert 6
The linked list is:  start--> 1--> 2--> 3--
Do you want more insert operations:  y
1. Insertion at end
2. Insertion at after specified element
3. Search an element
Your choice is: 3
Enter an element to search: 6
6:  is found at  : 4
Do you want more insert operations:  n
```

Summary

1. A linked list is a collection of similar data item connected through the pointers.
2. The linked list may be categorized into single linked list, doubly linked list and circular linked list.
3. The nodes of single linked lists are divided into two parts, the first part contains `info` and the second part contains the `next` pointer containing the address of the next node.
4. In a single linked list only forward links are used to connect the nodes.
5. In doubly linked list, we can move to forward and backward, that is, in both directions.
6. The linked list is created using dynamic memory allocation so nodes can be allocated and deallocated at run time.
7. The linked lists can be used for polynomial manipulation, circular queue implementation, stack implementations and hashing implementation.
8. In doubly linked list adjacent nodes are connected both by forward links as well as backward links.
9. In circular linked lists the last node follows the first node.
10. We can insert an element at any position in any kind of linked list.

11. The linked list can be implemented either using static memory allocation or using dynamic memory allocation.
12. In a circular linked list we can traverse from any node to any other node in forward direction only.
13. In static implementation of linked list the size is declared in advance, while in dynamic implementation of linked list the nodes can be created according to the need.
14. List is a flexible data structure that can grow or shrink according to the need.
15. The doubly linked list may also be circular.

Key Terms

Node
`Info`
`Next`
Linked list
Single linked list
Doubly linked list
Circular linked list
Linked stack
Linked queue
Sparse array

Multiple-Choice Questions

1. The linked list is not suitable data structure for
   a. binary search.
   b. linear search.
   c. bubble sort.
   d. circular queue.
2. A linked list in which item can be inserted or deleted from either end is called
   a. stack.
   b. queue.
   c. array.
   d. none of the above.
3. Time require to insert an element into the linked queue of n elements is proportional to
   a. O(1).
   b. O($\log_2 n$).
   c. O(n).
   d. O($n \log_2 n$).
4. Time require to insert an element into the linked stack of n elements is proportional to
   a. O(1).
   b. O($\log_2 n$).
   c. O(n).
   d. O($n \log_2 n$).
5. Linked list is a better choice to implement
   a. circular queue.
   b. deque.
   c. sparse arrays.
   d. arrays.
6. A circular linked can be used to represent
   a. stack.
   b. array.
   c. queue.
   d. none of the above.
7. Which of the following is used in ‘ C’ to release the memory?
   a. `malloc()`
   b. `new`
   c. `free()`
   d. `delete`
8. Which of the following provides two-way traversal?
   a. Doubly linked list
   b. Single linked list

c. Circular linked list
d. None of the above

9. To represent the empty linked list we use
a. `Start=NULL`
b. `Ptr=NULL`
c. `Start->next=NULL`
d. `Ptr->next=NULL`

10. Which of the following is used to represent the overflow condition in a single linked list?
a. `Start=NULL`
b. `Ptr=NULL`
c. `Ptr->next=NULL`
d. `Start->next=NULL`

11. Which of the following statements is correct?
a. The contiguous memory allocation is done is array.
b. The contiguous memory allocation is done is linked list.
c. The contiguous memory allocation is not done is array.
d. The contiguous memory allocation is note done is array.

12. Which of the following statement is not correct?
a. Stacks can be implemented using linked list.
b. Circular queue can be implemented using linked list.
c. Sparse array can be implemented using linked list.
d. All of the above.

13. In a double linked list,
a. each node contains `inf`, `pre` and `next` field.
b. the last node of the list follows first node.
c. we can move in to the forward direction only.
d. each node contains `inf` and `next` field.

14. Which of the following is used in ' C' to dynamically allocate the memory?
a. `malloc()`
b. `new`
c. `free`
d. `delete`

15. The function `malloc()` returns
a. `void` pointer.
b. `int` pointer.
c. pointer of `struct` type.
d. It depends on the program.

16. Which of the following statements is correct?
a. Run time memory allocation is called dynamic memory allocation.
b. Compile time memory allocation is called dynamic memory allocation.
c. Memory size is declared in advance in dynamic memory allocation.
d. All of the above.

17. Searching in a linked list can be done
a. if linked list is in sorted order.
b. if the linked list is a single linked list.
c. if the linked list is a doubly linked list.
d. all of the above.

18. Inserting a node in a doubly linked list requires
a. one pointer exchanges.
b. two pointer exchanges.
c. three pointer exchanges.
d. four pointer exchanges.

19. Which of the following is not a correct statement?
a. We cannot insert an element at any position in a linked list.
b. The linked list can be implemented using static memory allocation.
c. The linked list can be implemented using dynamic memory allocation.
d. In a circular linked list we can traverse from any node to any other node in forward direction only.

Review Questions

1. Define linked list and compare it with the array.
2. What are the different types of linked lists?
3. Can polynomials be represented as linked list?
4. Explain the advantages and disadvantages of linked lists over arrays.
5. What are the differences between doubly linked lists and circular linked lists?
6. What is a doubly linked list? Write algorithms to add and delete a node from it.
7. Explain how a polynomial can be represented using single linked lists.
8. Write an algorithm to add two polynomials.
9. Write an algorithm to insert and delete element from the specified position in the doubly linked list.
10. Write an algorithm to reverse the single linked list.
11. Explain the representation of the linked list in the memory.
12. Differentiate the following: single linked list, doubly linked list and circular linked list.
13. What is a priority queue? Which data structure is best suited for the implementation of priority queue and why?
14. Explain dynamic memory allocation and its advantages.
15. How can a linked list be implemented using arrays?
16. Explain and write an algorithm to insert a node into a linked list at various positions.
17. Write an algorithm to delete an element from the single linked list.
18. Write an algorithm to
    (a) append an element to the end of the single linked list.
    (b) delete the last element from the double linked list.
    (c) return the sum of integers in a single linked list.
19. What is a sparse array? How it is represented using the linked list?
20. What are the applications of linked lists? Describe the different types of linked lists.
21. Why do we need linked list?
22. What are the differences between array representation and dynamic representation?
23. What are the operations that can be performed on a linked list?
24. Is it possible to insert or delete the first node of the linked list?
25. What are the advantages of representing a stack using linked list?
26. What is a doubly linked list? How is it different from the single linked list?
27. What is a circular linked list? How is it different from the single linked list?
28. What is the value of the `next` field of the first node of a circular linked list?

Programming Assignments

1. Write a program in C/C++ to count the numbers of elements in a circular linked list.
2. Write a program in C/C++ to arrange the data elements of a linked list in ascending order.
3. Write a program in C/C++ to concatenate two circular linked lists producing one circular linked list.
4. Write a program in C/C++ that will split a circularly linked list into two circular linked lists.
5. Write a program in C/C++ to multiply two long integers represented by doubly circular linked list.
6. Write a program in C/C++ to remove all the duplicate elements from the linked list.
7. Write a program in C/C++ for linear search using single linked list with recursive and non-recursive methods.
8. Write a program in C/C++ to revert the doubly linked list without creating a new node.
9. Write a program in C/C++ to insert an element in a sorted single linked list.
10. Write a menu driven program in C/C++ to sort, add, display and multiply the polynomial.

Answers

Multiple-Choice Questions

1. (a)
2. (b)
3. (a)
4. (a)
5. (c)
6. (c)
7. (c)
8. (a)
9. (a)
10. (b)
11. (d)
12. (d)
13. (a)
14. (a)
15. (a)
16. (a)
17. (d)
18. (d)
19. (a)

7 Tre e

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Non-linear data structure called tree.
- Representation of trees.
- Different types of traversal methods.
- Construction of different types of trees.
- Operations on different types of trees.
- Different types of trees, such as AVL, B-tree, general tree, expression tree, etc.
- How to implement the tree in the programming language.

We know that data structures are divided into two categories: Linear and non-linear data structures. In the preceding chapters, we have discussed various linear data structures such as arrays, stacks, queues and linked lists. Now we will discuss the first non-linear data structure called tree in this chapter, and the second non-linear data structures called the graph in the next chapter. We have seen that the linear data structures process the data elements only in a sequential order. Tree is very useful in computer science applications because it is a non-linear data structure used to represent hierarchical relationships among the data elements. Also, it may be used in various applications of computer science, for example, to represent and easily solve the algebraic expression, in information retrieval, in data mining, in artificial intelligence, in natural language processing, etc. In this chapter, we will discuss the basic concept of tree, their types, operations and implementation.

7.1 Definition of Tree

In the non-linear data structure, data elements do not form a sequence. The tree may be defined as a finite collection of special data items called the nodes. These nodes of the tree are arranged in the hierarchical structures to represent the parent–child relationship. The first node is linked to some other nodes of the tree which in turn are linked to some other nodes as shown in Figure 1.

The parent is connected through an edge to its children or descendents. The first node which does not have any parent is called the root (node) of the tree. For example, the director does not have any parent or senior to him, or he does not report to any other person above him; therefore, the director is the root of the tree in Figure 1. There is only one descendent to the director, that is, Dean (academics) which in turn has four descendents: HOD (CSE), HOD (IT), HOD (EC) and HOD (EX). Each HOD has two descendents: Teaching (staff) and Non-teaching (staff). The teaching has two descendents F1 and F2, which do not have any descendents so they are the leaf nodes of the tree. All the nodes of the tree are called internal nodes except the root and the leaf nodes. The leaf nodes are sometimes called as external nodes. The binary tree with n nodes can have maximum $n - 1$ branches. We can see that each node of the tree has zero or more descendents. The nodes with zero descendents are called leaf nodes.

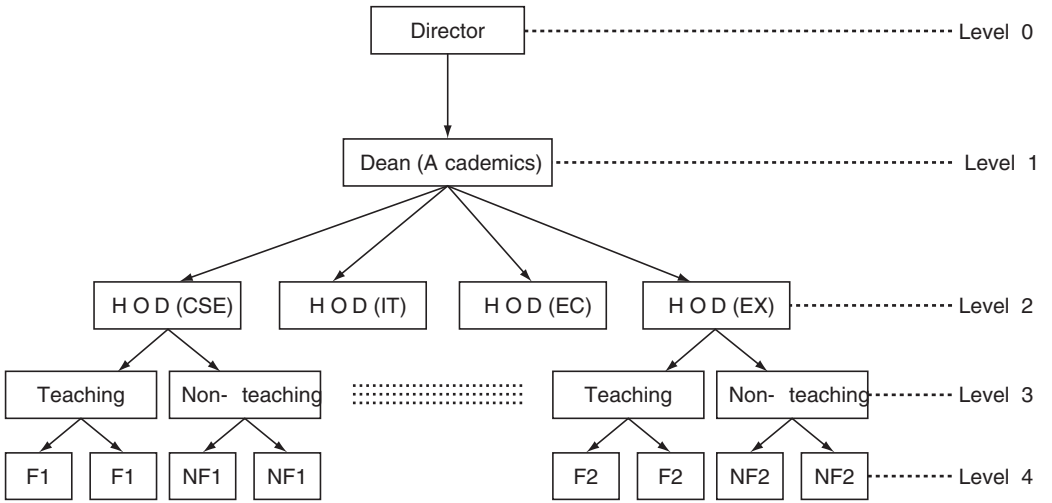


Figure 1 A typical example of a tree.

A very good example of a tree data structure is a directory system in our computers. We can now define the tree data structure as follows

Tree is a finite set of data elements called nodes such that there is a special node called the root of the tree; and remaining nodes are partitioned into a number of mutually exclusive subsets, which are themselves trees. The edges that connect the nodes are called branches, and the number of branches connected to a particular node is called the degree of that node.

Some of the key terms and their definition are as follows:

1. **Root:** A node of the tree which does not have a parent or the first node of the tree within-degree zero is called the root of the tree.
2. **Leaf node:** The last node of the tree which does not have any descendents or the node with zero out-degree is called leaf node or terminal node.
3. **Degree of a node:** The number of nodes connected to any node is called the degree of that node, or we can say that the number of children any node contains is the degree of that node.
4. **In-degree of a node:** The number of edges that ends at a particular node is called the in-degree of the node.
5. **Out-degree of a node:** The number of edges that yields from a particular node is called the out-degree of that node. The sum of in-degree and out-degree branches is the degree of the node.
6. **Path:** A sequence of edges between any two nodes is called the path between those two nodes.
7. **Depth of the tree:** The length of the longest path from the root to any terminal node is called the depth of the tree. Each node of the tree may be assigned a level as shown in Figure 1, which represents the distance of the node from the root. The root is always assigned a level zero. The level of the node is called the depth of the node. The largest level of the tree is called the depth of the tree. The minimum depth of the tree is $(\log_2 n + 1)$ where n is the number of nodes.
8. **Siblings:** The children of the same parent node are called siblings.
9. **Degree of a tree:** The maximum degree of any node is called the degree of a tree.

7.2 Binary Tree

In a general tree there is no restriction on the number of children associated with any node, but the general tree does not make the task easier. To begin with, we will discuss a special tree called binary tree which is easier than the general tree in processing and implementation. In the binary tree, each node can have either 0 children, 1 child or 2 children. Therefore, the binary tree is a tree which is either empty or each node can have maximum two children. The node on the left of any node is called its left child and that on its right is called its right child. The left and right children are also trees and they are said to be left subtree and right subtree as shown in Figure 2.

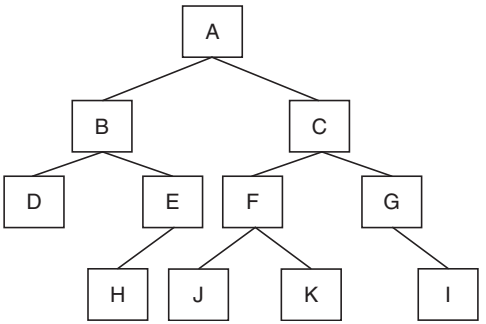


Figure 2 A binary tree.

As we can see in Figure 2, the left subtree of node A consists of D, B, F, H and the right subtree of node A consists of C, F, G, J, K and I. Similarly, the left subtree of node B consists D only and its right subtree consists E and H. The left child (left subtree) of the node E is H and its right child (right subtree) is nothing; therefore, we can see that node E has only one descendent and H has zero descendent. We are aware that in the binary tree, each node can have maximum two descendents. Therefore, the binary tree can also be defined as a finite collection of nodes where each node is divided into three parts containing left child address, information and right child address as shown in Figure 3.

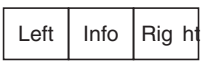


Figure 3 A node of binary tree.

The definition of a binary tree is recursive in the sense that left and right child of any node are themselves trees as shown in Figure 4. The empty subtree is represented by null as shown in the following figure by a cross(×).

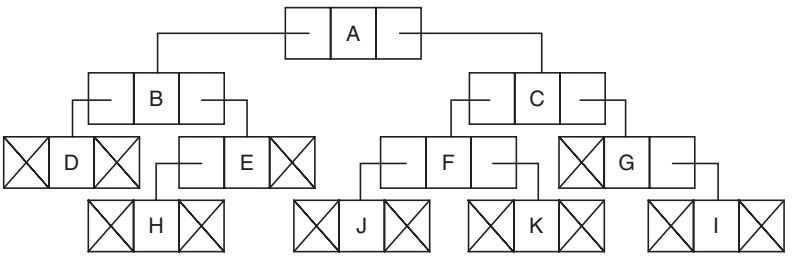


Figure 4 A descriptive representation of binary tree.

There are various variations of a binary tree.

Classification of Binary tree

Complete Binary Tree

If all the terminal nodes of a tree are at level d, where d is the depth of the tree, then such trees are called strictly complete binary trees. The maximum possible nodes at any level are 2^n, where n is the level number. The complete binary tree contains the maximum number of nodes. So there will be only one node (root) in the strictly complete binary tree of depth zero, three nodes for the strictly complete binary tree of depth 1 and seven nodes for the strictly complete binary tree of depth 2 as shown in Figure 5(a–c).

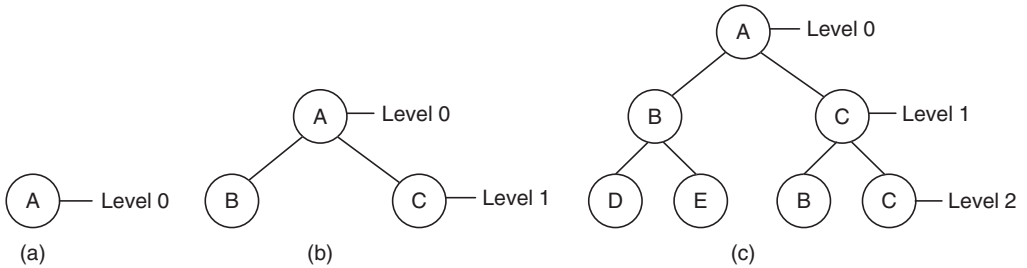


Figure 5 A strictly complete binary tree of (a) depth 0; (b) depth 1; (c) depth 2.

Almost Complete Binary Tree

If all the terminal nodes of a tree are at level d or $d - 1$, where d is the depth of the tree, then such trees are called almost complete binary trees. All the nodes at the last level are filled from left to right direction. If the right child of any node is at level d, then its left child must also be at level d. The almost complete binary tree is shown in Figure 6(a–b).

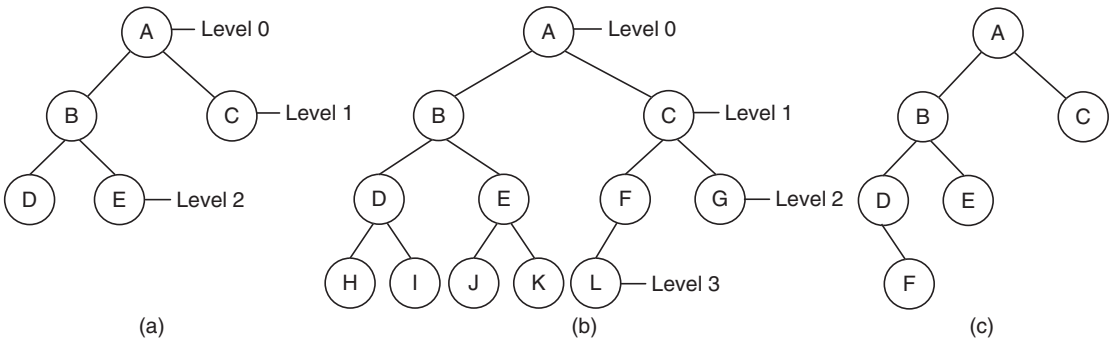


Figure 6 Almost complete binary tree in (a) and (b), and not an almost complete binary tree in (c).

Here in Figure 6(a) the leaf nodes are either at level 2 or at level 1. In Figure 6(b), node F has only one child, but it is a left child and all the terminals nodes are at either level 3 or level 2. However in Figure 6(c), D has the right child but it has no left child while in almost complete binary tree the left to right association is required, hence it is not an almost complete binary tree.

7.3 Representation of a Tree

Like the other data structures, the tree can also be represented in different ways.

Linked Representation Using an Array

A tree is a collection of data items that can easily be implemented using an array. The data items of the tree are maintained in the form of nodes divided into three parts. Therefore, the tree may be maintained or represented by three parallel arrays of the same sizes. One array for the actual data items is called the *Info* field of the node. Another array is for the addresses of its left child and the third array to keep the address of the right child of every node as shown in Figure 7. Let us take the array of 12 elements.

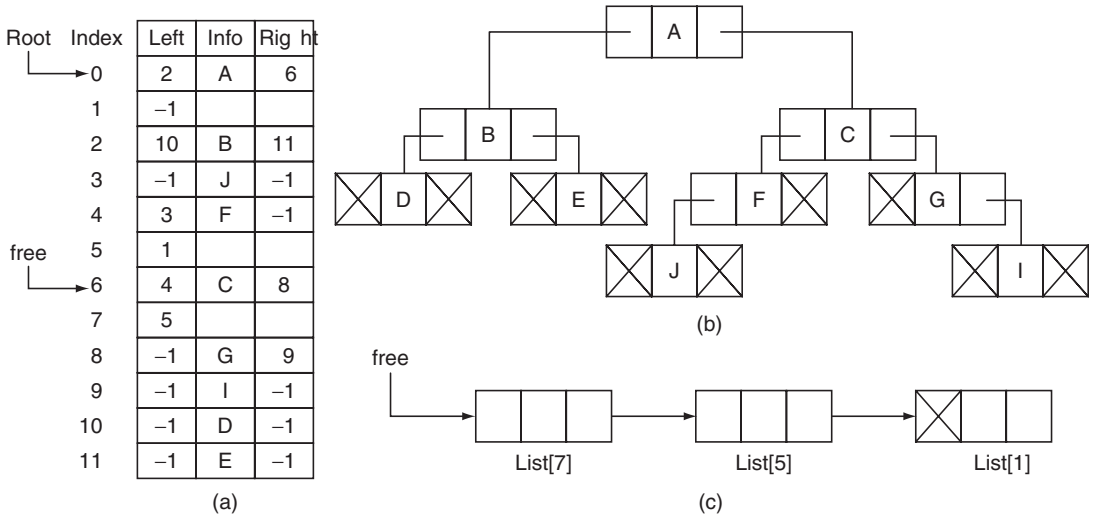


Figure 7 (a) Array representation of a tree; (b) pictorial representation of tree using an array; (c) pictorial representation of free blocks of the array.

As we can see that the elements are not placed into the contiguous memory location, but they are scattered into different locations of the array named "Tree" as shown in Figure 7(a). Tree[0] contains the root element A of the tree pointed by name "Root". The left field of the Tree[0] contains 2 which means the left subtree is at Tree[2], and the right field of the Tree[0] contains 6 which means that the right subtree is at Tree[6]. Other nodes are also represented in a similar manner. The tree of the elements is shown in Figure 7(b). Since array has a fixed size declared in advance, so we have to maintain other trees of free locations. Let us take the first free location at index value 7 pointer by the name "Free" and the tree of free locations is maintained as shown in Figure 7(c). The tree using array may be implemented in C/C++ by using array of structure as defined here.

In C	In C++

`struct node`
`{`
`int Left;`
`int Info;`
`int Right;`
`};`
`struct node Tree[12];`
	`struct node`
	`{`
	`int Left;`
	`int Info;`
	`int Right;`
	`};`
	`node Tree[12];`

Remember to use `struct` keyword or `typedef`	The `struct` keyword is not required

Therefore, Tree[12] is a collection of 12 nodes each containing integer type of info and integer type of left and right fields because array index is also an integer and we use array index as a pointer to a node. For example, Tree[6] points to the 6th element of the array that contains data item C and left field address = 4 as shown in Figure 7(a). In the array representation, array index starts from 0 so the null value may be represented by – 1. Hence, if the left or right field of any node of the tree contains – 1, it means there is no left subtree or right subtree, respectively. When the tree is represented using an array as explained earlier and shown in Figure 7 then

`Tree[n]` represents the node of the tree pointed by index value *n*. For example, Tree[6] is the node of the tree pointed by index value 6, that is,

4	C	8

`Tree[n].info` represents the info part of the node of the tree pointed by index value *n*. For example, `List[6].info` = C

`Tree[n].left` represents the left field of the node of the tree pointed by index value *n*. For example, `List[6].left` = 4. It represents the address of the left subtree of the node pointed by index 6.

`Tree[n].right` represents the right field of the node of the tree pointed by index value *n*. For example, `List[6].right` = 8. It represents the address of the right subtree of the node pointed by index 6.

The above data is summarized in Table 1.

Table 1 The representation of info and next field by an index of the array `List`

Index	`List[index].info`	`List[index].left`	`Lis`
0	A	2	6
2	B	10	11
6	C	4	8
4	F	3	– 1
10	D	– 1	– 1
11	E	– 1	– 1
8	G	– 1	9
3	J	– 1	– 1
9	I	– 1	– 1

Sequential Representation of Binary Tree

Instead of taking the three parallel arrays, the complete or almost complete binary tree can be represented using a single array. In this method, the numbers are assigned to each node in a sequence from leftmost nodes to rightmost nodes, and with the condition that the root is always assigned to 1. Consider the almost complete binary tree shown in Figure 8. In this method the left child of any node say k is placed at [2*k] and right child is placed at [2*k + 1]. The size of the array depends on the depth of the tree and it is given by 2^{d+1}. So for the example given below, the depth of the tree is 3 and therefore the number of elements in the array should be $2^{3+1} = 16$.

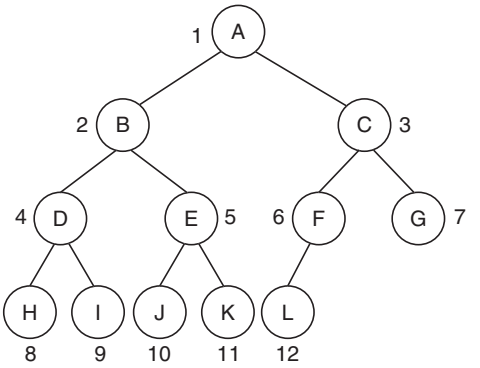


Figure 8 Almost complete binary tree.

Root (A) = 1
Left child of Root (A) = B = ⌊2*k⌋ = ⌊2*1⌋ = 2
Right child of Root (A) = C = ⌊2*k + 1⌋ =⌊2*1 + 1⌋ = 3

Consider B as the root of the subtree and to be placed at position 2, so
Left child of B = ⌊2*k⌋ = ⌊2*2⌋ =4
Right child of B = ⌊2*k + 1⌋ = ⌊2*2 + 1⌋ = 5

Consider C as the Root of the subtree and to be placed at position 3 so
Left child of C = ⌊2*k⌋ = ⌊2*3⌋ = 6
Right child of C = ⌊2*k + 1⌋ = ⌊2*3 + 1⌋ = 7

Similarly, the other node's position can be calculated and the given tree can be stored sequentially in a single array of 16 elements as follows

A	B	C	D	E	F	G	H	I	J	K	L				
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The father of any node k can be determined by k/2 so the father of node j at position 10 is 10/2 = 5.

If the root is placed at number 0 then left child's position will be ⌊2*k + 1⌋ and right child's position will be ⌊2*k + 2⌋. The given tree can be represented by

A	B	C	D	E	F	G	H	I	J	K	L				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Linked Representation Using Pointer (Dynamic Representation)

The array implementation of tree is very simple but it requires the ultimate size of the tree to be declared in advance. The advance declaration of size is not possible all the time, therefore, we can say that array representation of a tree is not flexible. The dynamic implementations of tree do not have size limitations and use space proportional to the actual number of elements of the tree. This is represented in computer memory with the help of following structure in C/C++.

In C	In C++

`struct node`
`{`
`node *left;`
`int info;`
`node *right;`
`};`
`struct node *root;`
	`struct node`
	`{`
	`node *left;`
	`int info;`
	`node *right;`
	`};`
	`node *root;`

Remember to use `struct` keyword or `typedef`	The `struct` keyword is not required

In the dynamic representation of the tree, the nodes are created as and when required by using the dynamic memory allocation methods of C or C++ as already discussed in Chapter 4. We may use `p` as a pointer to a tree node, then the left child of `p` may be accessed by using `p->left` and the right child of `p` may be accessed by using `p->right`

7.4 Operations on the Binary Tree

Some of the basic operations that can be performed on the tree are discussed in this section.

Construction of a Binary Tree

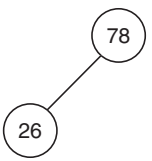
Before going into the operation on the tree, let us discuss the construction of a binary tree. The binary tree is constructed with an example as follows.

Example 1

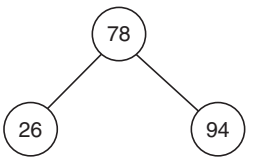
Let us take the example 78, 26, 94, 43, 23, 14, 53, 76, 76, 29.

Step 1: Initially the tree is empty so place the first number at the root.

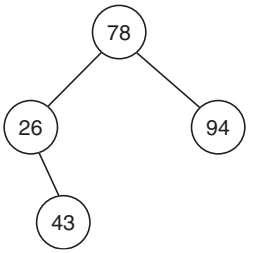
Step 2: Compare the next number (26) with the root, if the incoming number is greater than or equal to the root then place it in the right child position otherwise place it into the left child position.



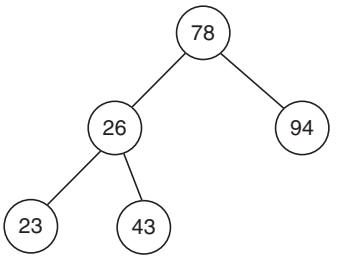
Step 3: Compare the next number (94) with the root, if the incoming number is greater than or equal to the root then go through the right subtree otherwise examine the left subtree.



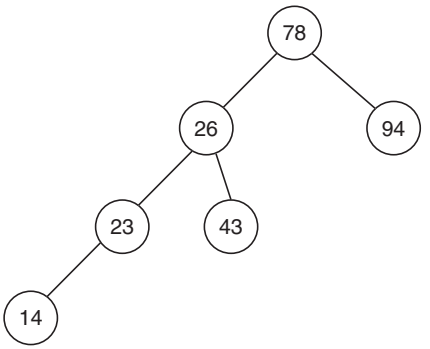
Step 4: Repeat the process for the next incoming number 43 (43 is smaller than 78 so go to the left subtree where compare it with 26 since 43 > 26, and so place it in the right child's position).



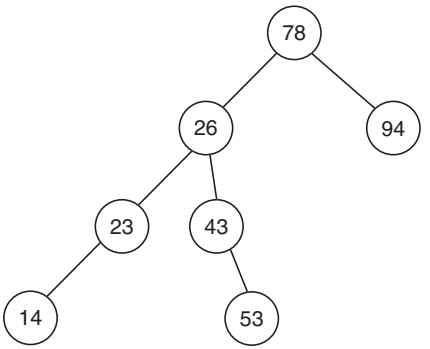
Step 5: Repeat the process for the next number 23.



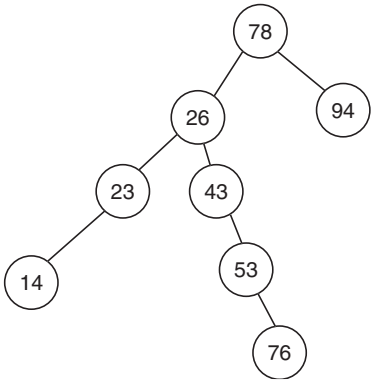
Step 6: Repeat the process for the next number 14.



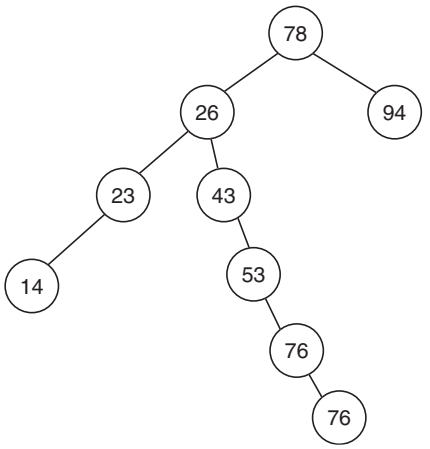
Step 7: Repeat the process for the next number 53.



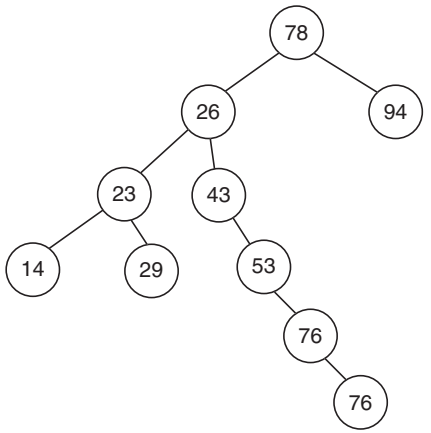
Step 8: Repeat the process for the next number 76.



Step 9: Repeat the process for the next number 76.



Step 10: Repeat the process for the next number 29.



Trav ersal of Binary Tree

Traversing is a process in which each node is visited exactly once. We can perform any operation on the node traversed; the operation depends on the application. Like other data structures, the elements (nodes) of a tree can also be traversed and there are three different methods of traversal of a tree. All these methods perform the same steps but the sequences of these steps differ. The steps are as follows:

1. Visit (traverse) the root denoted by N.
2. Visit (traverse) the left subtree denoted by L.
3. Visit (traverse) the right subtree denoted by R.

The different traversal methods are as follows:

1. In-order traversal.
2. Pre-order traversal.
3. Post-order traversal.

In-Order Traversal

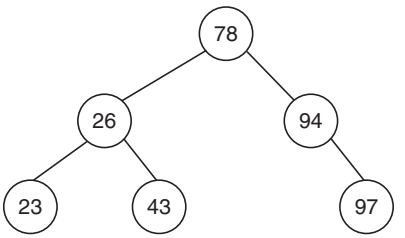
In in-order traversal, the sequence of the above-mentioned steps is as follows:

1. Visit the left subtree in the in-order (L).
2. Visit the node (N).
3. Visit the right subtree in the in-order (R).

In in-order traversal, the left subtree is traversed in the in-order first followed by the root and then the right subtree. We can write the in-order traversal as LNR, and we can see that this method is a recursive method in which left subtree or right subtree are again traversed in the in-order.

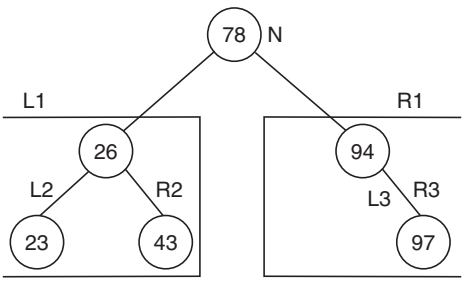
Problem 1

Traverse the following tree using in-order traversal method.



Solution

As we are aware that in-order traversal first traverses the left subtree in in-order followed by the root and then the right subtree, we can draw the above tree as



Here L1 is a left subtree of Node $N = 78$ and R_1 is a subtree of N. The in-order traversal of the given tree will be L1NR1 (left subtree, node, right subtree) where L1 is again traversed in-order and it results in L2 26 R2. The in-order traversal of right subtree will be R1 = L3 94 R3, where L3 is empty because there is no left subtree of node containing 94. Therefore, the in-order traversal of R1 results in 94 R3. Placing the in-order traversal of L1 and R1 into L1NR1, we will get L2 26 R2 N 94 R3. Now, replacing the variables L2 R2 N R3 with the corresponding values, we get 23 26 43 78 94 97.

Algorithm 1 `Inorder(ptr)`

1. `If ptr != NULL`
   a. `Inorder(ptr->left)`
   b. `Print ptr->info`
   c. `Inorder(ptr->right)`
2. `Exit`

Pre-Order Traversal

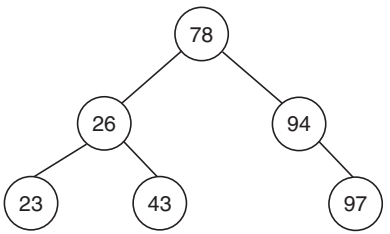
In pre-order traversal, the sequence of the steps is as follows:

1. Visit the node (N).
2. Visit the left subtree in in-order (L).
3. Visit the right subtree in in-order (R).

In pre-order traversal, the node is traversed first followed by the left subtree in pre-order and then the right subtree. We can write the pre-order traversal as NLR. This is again a recursive method.

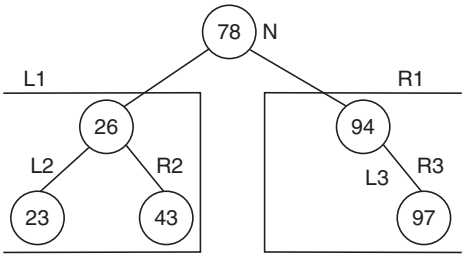
Problem 2

Traverse the following tree using pre-order traversal method.



Solution

As we are aware that pre-order traversal first traverses the node followed by left subtree in pre-order and then right subtree, we can draw the above tree as follows:



Here L1 is a left subtree of node $N = 78$ and R1 is a subtree of N. The pre-order traversal of the given tree will be NL1R1 (node, left subtree, right subtree), where L1 is again traversed in pre-order and it results in 26 L2 R2. The pre-order traversal of right subtree will be R1 = 94 L3 R3, where L3 is empty because there is no left subtree of node containing 94. Therefore, the pre-order traversal of R1 results in 94 R3. Placing the pre-order traversal of L1 and R1 in to NL1R1, we will get **N** 26 L2 R2 94 R3. Now, replacing the variables L2 R2 N R3 with the corresponding values, we get 78 26 23 43 94 97.

Algorithm 2 `Preorder(ptr)`

1. `If ptr!=NULL`
   a. `Print ptr->info`
   b. `Preorder(ptr>left)`
   c. `Preorder(ptr->right)`
2. `Exit`

Post-Order Traversal

In post-order traversal, the left subtree is traversed in post-order first, followed by the right subtree in post-order and then the root. We can write the post-order traversal as LRN and we can see that this method is also a recursive method in which the left subtree or right subtree are again traversed in post-order. The sequence of the steps is as follows:

1. Visit the left subtree in in-order (L).
2. Visit the right subtree in in-order (R).
3. Visit the node (N).

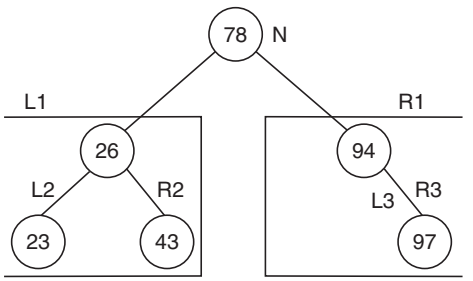
Here we can see that in all traversal methods, the left subtree is traversed before the right subtree, the difference lies in the sequence when the root (node) is traversed. If the root is traversed first then it is pre-order traversal. If it is traversed in between left and right subtrees then it is in-order traversal. And if it is traversed at the last then it is post-order traversal.

Problem 3

Traverse the tree given in Problem 1 in post-order traversal.

Solution

As we are aware that post-order traversal first traverses the left subtree in post-order followed by the right subtree in post-order then the root of the tree. Therefore, we can draw the above tree as



Here L1 is a left subtree of node *N* = 78 and R1 is a subtree of N. The post-order traversal of the given tree will be L1R1N (left subtree, right subtree, node) where L1 is again traversed in post-order and it results in L2 R2 **26**. The post-order traversal of right subtree R1 = L3 R3 **94** where L3 is empty because there is no left subtree of node containing 94. Therefore, the post-order traversal of R1 results in R3 94. Placing the post-order traversal of L1 and R1 into L1R1**N**, we will get L2 R2 26 R3 94 **N**. Now, replacing the variables L2 R2 N R3 with the corresponding values, we get 23 43 26 97 94 78.

Algorithm 3 `Postorder(ptr)`

```
1. If ptr!=NULL
   a. Postorder(ptr>left)
   b. Postorder(ptr->right)
   c. Print ptr->info
2. Exit
```

We have seen the different traversal techniques for tree traversal, but they are recursive in nature. Therefore, the question is: Are the traversals performed non-recursively? The algorithms for non-recursive traversal of a tree are as follows.

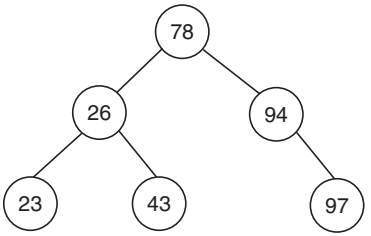
Algorithm 4 `Preorder(Root)`

Here the `Ptr` contains the adderss of the current node being scanned and the stack is used to temporarily hold the addresses to be processed.

```
1. Set Stack = empty by assigning Top = NULL
2. Set Ptr = Root
3. If Ptr==NULL then
   a. "Display the message - Tree is empt
   b. Exit
4. Repeat while ptr!=NULL
   a. Print Ptr->info (Process the node)
      if (any right child is there for Ptr then keep its address into the stack)
   b. If Ptr->right!=NULL then
        Top=Top++
        Stack[Top]=Ptr->right
   c. If Ptr->left!=NULL then
        Ptr=Ptr->left (Follow the left
        Else
        {
         Ptr=Top (Back track the tree a
      elements from the Top of the stack)
        Top=Top-1
        }
5. Exit
```

Problem 4

Traverse the following tree in pre-order using non-recursive traversal.

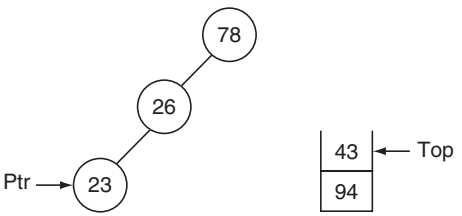


Solution

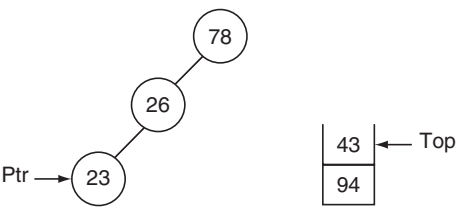
Step 1: `Top = Null` and `Ptr = Root` so `Ptr` now points to the node containing 78.

Step 2: Process the node at `Ptr` (containing 78). There is a right child to `Ptr` so push the right child into the stack and move the `Ptr` to next element in the leftmost path.

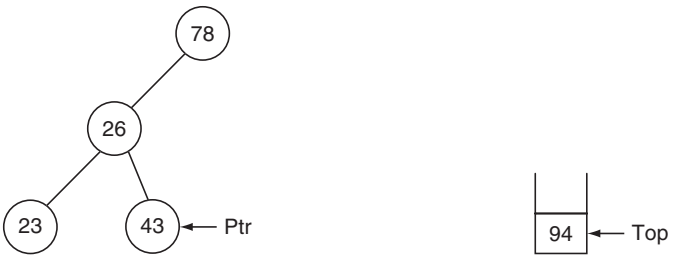
Step 3: Process the node at `Ptr` (containing 26). There is a right child to `Ptr`, so push the right child into the stack and move the `Ptr` to next element in the leftmost path.



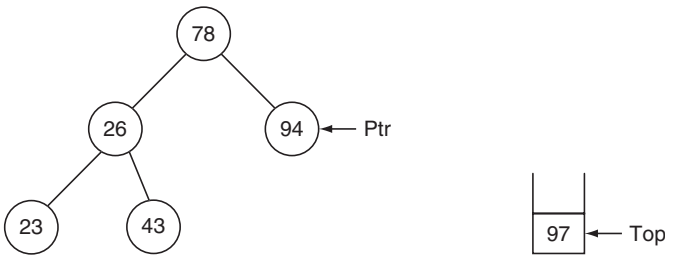
Step 4: Process the node at `Ptr` (containing 23). There is no right child to `Ptr`, so no need to push anything into the stack and move the `Ptr` to next element in the leftmost path. Since the next element does not exist so `Ptr = Null`



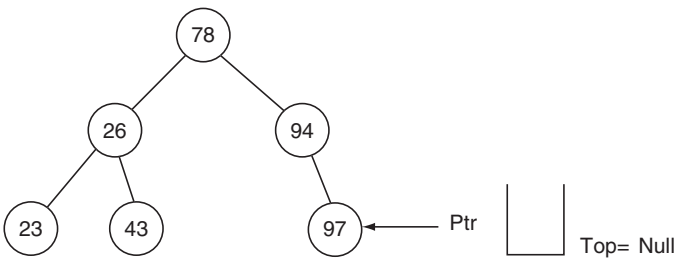
Step 5: Pop the topmost element (43) from the stack and assign its address to `Ptr`. Process the node at `Ptr` (containing 43). There is no right child to `Ptr`, so no need to push anything into the stack and move the `Ptr` to next element in the leftmost path. Since the next element does not exist so `Ptr = Null`



Step 6: Pop the topmost element (94) from the stack and assign its address to `Ptr`. Process the node at `Ptr` (containing 94). There is a right child to `Ptr`, so push it into the stack and move the `Ptr` to next element in the leftmost path. Since the next element does not exist so `Ptr = Null`



Step 7: Pop the topmost element (97) from the stack and assign its address to `Ptr`. Process the node at `Ptr` (containing 97). There is no right child to `Ptr`, so no need to push anything into the stack and move the `Ptr` to next element in the leftmost path, Since the next element does not exist so `Ptr = Null`. Therefore, now stop the process since `Ptr = Null` and `Top = Null`



Therefore, the elements 78 26 23 43 94 97 are processed in pre-order.

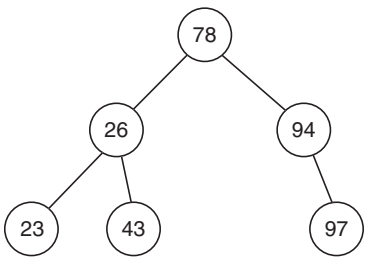
Algorithm 5 `Inorder(Root)`

Here the `Ptr` contains the adderss of the current node being scanned, and the stack is used to temporarily hold the addresses to be processed.

1. Set `Stack` = empty by assigning `Top = Null`
2. Set `Ptr = Root`
3. If `Ptr == Null` then
   a. "`Display the message - Tree is empty`"
   b. `Exit`
4. Repeat the following steps a, b and c `while Ptr != Null` (Pushing each node into the stack coming in the leftmost path)
   a. `Top++`
   b. `Stack[Top] = Ptr`
   c. `Ptr = Ptr->left`
5. Set `Ptr = Top`
6. `Top = Top - 1`
7. Repeat the following steps a and b `while Ptr != Null` (Popping the elements from the stack until Null is reached)
   a. Print `Ptr->info` (Process the node)
      (if any right child is there for `Ptr` then make the `Ptr` to its right child)
   b. `If Ptr->right != NULL then`
      a. `Set Ptr = Ptr->right`
      b. Go to Step 5
8. `Ptr = Top`
9. `Top = Top - 1`
10. `Exit`

Problem 5

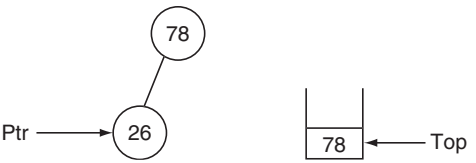
Traverse the following tree in in-order using non-recursive traversal.



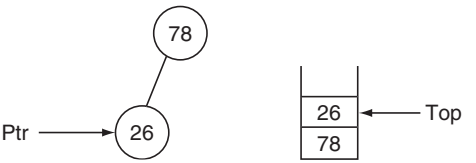
Solution

Step 1: `Top = Null` and `Ptr = Root` so `Ptr` now points to the node containing 78.

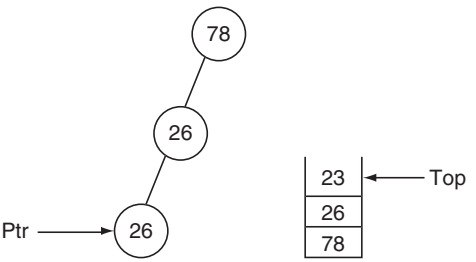
Step 2: Push the current node at `Ptr` into the stack and move to the next node in the leftmost path by assigning `Ptr = Ptr->left`. Therefore, `Ptr` now points to the node containing 26.



Step 3: Push the current node at `Ptr` into the stack, and move to the next node in the leftmost path by assigning `Ptr = Ptr->left`. Therefore, `Ptr` now points to the node containing 23.

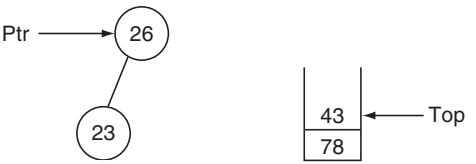


Step 4: Push the current node at `Ptr` into the stack, and move to the next node in the leftmost path by assigning `Ptr = Ptr->left`. Therefore, `Ptr` now points to Null.



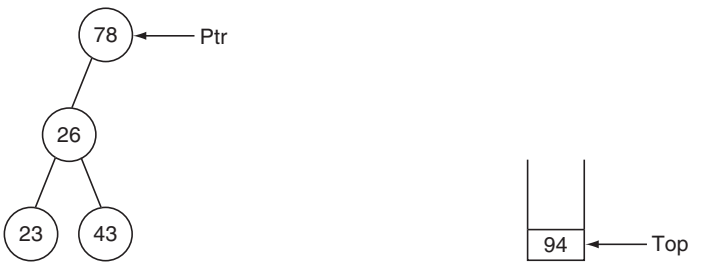
Step 5: Now pop the elements from the stack and assign it to `Ptr` and process it (23). If there is a right child to `Ptr`, assign `Ptr = Ptr->right` and again repeat the procedure by pushing the elements encountered in the leftmost path at `Ptr.r`

Step 6: Now pop the elements from the stack, assign it to `Ptr` and process it (26). Since there is a right child to `Ptr`, again repeat the procedure by pushing the elements encountered in the leftmost path at `Ptr.r`

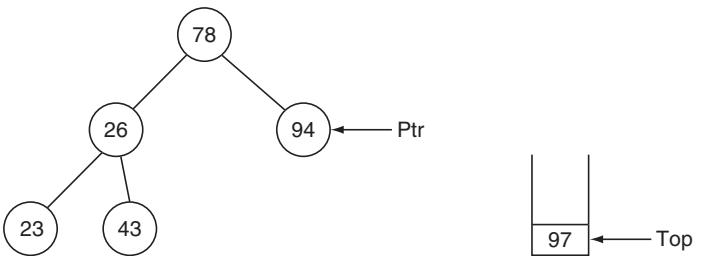


Step 7: Now pop the elements from the stack, assign it to `Ptr` and process it (43). Since there is no right child to `Ptr` so there is no need to push anything to the stack.

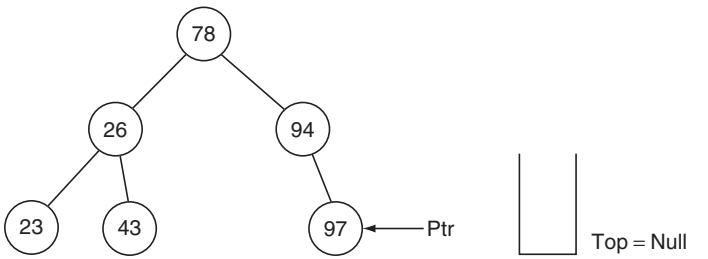
Step 8: Now pop the elements from the stack, assign it to `Ptr` and process it (78). Since there is a right child to `Ptr`, assign `Ptr = Ptr->right` and again repeat the procedure by pushing the elements encountered in the leftmost path at `Ptr.r`



Step 9: Now pop the elements from the stack, assign it to `Ptr` and process it (94). Since there is a right child to `Ptr`, assign `Ptr = Ptr->right` and again repeat the procedure by pushing the elements encountered in the leftmost path at `Ptr.r`



Step 10: Now pop the elements from the stack, assign it to `Ptr` and process it (97). Since there is no right child to `Ptr`, there is no need to push anything to the stack.



Step 11: Now pop the elements from the stack, assign it to `Ptr`. However, since there is nothing in the stack, so `Ptr = NULL`. Therefore, we must stop the process here. The in-order traversal of the tree results in the sequence 23 26 43 78 94 97.

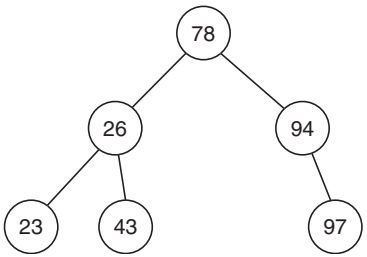
Algorithm 6 `Postorder(Root)`

Here the `Ptr` contains the address of the current node being scanned, and stack is used to temporarily hold the addresses to be processed.

1. Set `Stack = Empty` by assigning `Top = Null`
2. Set `Ptr = Root`
3. If `Ptr = Null` then
   a. "`Display the message - Tree is empt`
   b. `Exit`
4. Repeat a, b, c and d `while Ptr !=Null` (Pushing each node into the stack coming in the leftmost path)
   a. `Top++`
   b. `Stack[Top] = Ptr`
   c. `If Ptr->right != Null`
      `{`
      `Top++;`
      `Stack[Top] = -Ptr ->right`
      `}`
   d. `Set Ptr = Ptr->left`
5. `Ptr = Top`
6. `Top = Top - 1`
7. Repeat steps a and b `while Ptr > 0`
   (Pop the elements from the stack until Null is reached)
   a. `Print Ptr` (Process the node)
   b. `If Ptr = Top`
   c. `Top = Top - 1`
8. `If Ptr < 0`
   a. `Set Ptr = -Ptr`
   b. Goto Step 4
9. `Exit`

Problem 6

Traverse the following tree in post-order using non-recursive traversal.

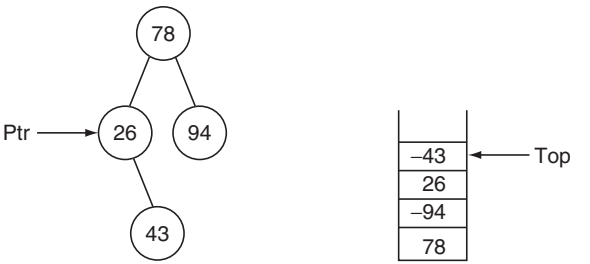


Solution

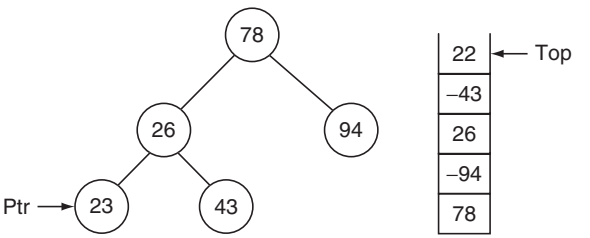
Step 1: `Top = Null` and `Ptr = Root` so `Ptr` now points to the node containing 78.

Step 2: Push the current node at `Ptr` into the stack. Since `Ptr` has a right child, push its negative to the stack and then move to the next node in the leftmost path by assigning `Ptr = Ptr->left`. Therefore, `Ptr` now points to the node containing 26.

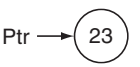
Step 3: Push the current node at `Ptr` and negative of its right child into the stack. Move to the next node in the leftmost path by assigning `Ptr = Ptr->left`. Therefore, `Ptr` now points to the node containing 23.



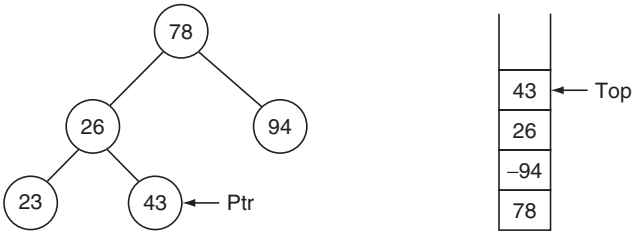
Step 4: Push the current node at `Ptr` into the stack since there is no right child of this node. There is no left child also, so now start popping the nodes from the stack and process them.



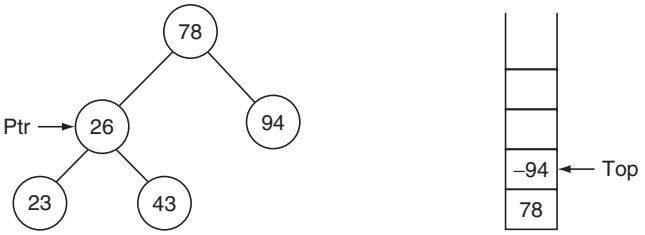
Step 5: Now pop the elements from the stack and assign it to `Ptr`. If it is greater than zero then process it (23).



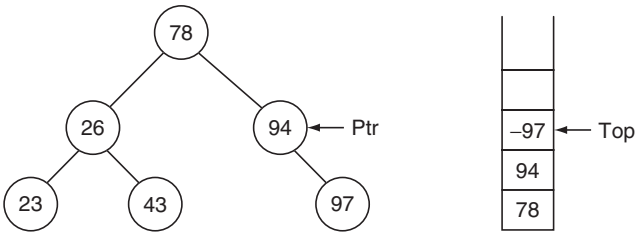
Step 6: Now pop the elements from the stack and assign it to `P t .r`. Since it is less than zero, so convert it into positive and repeat the process by following the leftmost path from `P t .r`



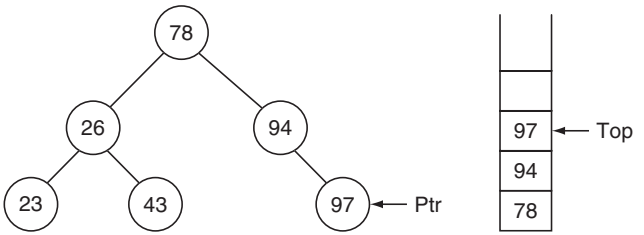
Step 7: Now pop the elements (43 and 26) since they are greater than zero so process them.



Step 8: Now pop the elements from the stack and assign it to `P t .r`. Since it is less than zero, so convert it into positive and repeat the process by following the leftmost path from `P t .r`



Step 9: Now pop the elements from the stack and assign it to `P t .r`. Since it is less than zero, so convert it into positive and repeat the process by following the leftmost path from `P t .r`



Step 10: Now pop the elements (97, 94 and 78) since they are greater than zero. So process them and stop the process since `P t r = N` because `T o p = N`. Therefore, the post-order traversal of the given tree follows the sequence 23 43 26 97 94 78.

Program 1 A program in C++ to create, insert and perform various traversal techniques in the binary tree.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
class tree
{
private:
struct node
{
int info;
node *left;
node *right;
};
node *ptr,*root,*pre;
char ch;
int item;
public:
tree();
~tree();
node *getnode()
{
ptr=new node;
return ptr;
}
void add(int n);
void traversal();
void inorder(node *ptr);
void preorder(node *ptr);
void postorder(node *ptr);
void create(node *);
};
tree::tree()
{
root=NULL;
}
tree::~tree()
{
delete root;
}
void tree::create(node *ptr)
{
```

```
ptr=root;
pre=root;
cout<<"\nEnter item to insert : ";
cin>>item;
do
{
if(item<ptr->info)
{
pre=ptr;
ptr=ptr->left;
}
else if(item>ptr->info)
{
pre=ptr;
ptr=ptr->right;
}
else
{
pre=NULL;
break;
}
}while(ptr);
if(pre)
{
ptr=getnode();
ptr->info=item;
ptr->left=ptr->right=NULL;
if(ptr->info<pre->info)
{
pre->left=ptr;
if(pre==root)
root=pre;
}
if(ptr->info>=pre->info)
{
pre->right=ptr;
ptr->left=ptr->right=NULL;
if(pre==root)
root=pre;
}
}
else
cout<<"Duplicate entry"<<endl;
cout<<"Do you want to insert more elements
```

```
ch=getche();
}
void tree::add(int n)
{
ptr=root;
if(ptr==NULL)
{
ptr=getnode();
root=ptr;
root->info=n;
root->left=root->right=NULL;
cout<<"Do you want to insert more elements
ch=getche();
while(ch=='y')
create(root);
}
}
void tree::traversal()
{
int option;
do
{
cout<<"\n1.In order traversal";
cout<<"\n2.Pre order Traversal";
cout<<"\n3.Post order Traversal\n";
cout<<"Enter 0 to exit\n";
cout<<"Enter your choice: ";
cin>>option;
switch(option)
{
case 1:
cout<<"The in order traversal = ";
inorder(root);
getch();
clrscr();
break;
case 2:
cout<<"The pre order traversal = ";
preorder(root);
getch();
clrscr();
break;
case 3:
cout<<"The post order traversal = ";
```

```
postorder(root);
getch();
clrscr();
break;
}
}while(option);
}
void tree::inorder(node *q)
{
if(q!=NULL)
{
inorder(q->left);
cout<<"\t"<<q->info;
inorder(q->right);
}
}
void tree::preorder(node *q)
{
if(q!=NULL)
{
cout<<"\t"<<q->info;
preorder(q->left);
preorder(q->right);
}
}
void tree::postorder(node *q)
{
if(q!=NULL)
{
postorder(q->left);
postorder(q->right);
cout<<"\t"<<q->info;
}
}
void main()
{
clrscr();
tree obj;
int ele;
cout<<"Enter the Root element to insert "
cin>>ele;
obj.add(ele);
obj.traversal();
getch();
}
```

Output

```
Enter the Root element to insert  12
Do you want to insert more elements(y/n)?
Enter item to insert : 67
Do you want to insert more elements(y/n)?
Enter item to insert : 43
Do you want to insert more elements(y/n)?  y
Enter item to insert : 2
Do you want to insert more elements(y/n)?  y
Enter item to insert : 4
Do you want to insert more elements(y/n)?  y
Enter item to insert : 12
Duplicate entry
Do you want to insert more elements(y/n)?  y
Enter item to insert : 65
Do you want to insert more elements(y/n)?  y
Enter item to insert : 2
Duplicate entry
Do you want to insert more elements(y/n)?  n
1.In order traversal
2.Pre order Traversal
3.Post order Traversal
Enter 0 to exit
Enter your choice: 1
The in order traversal =          2          4
671.Inorder traversal
2.Preorder Traversal
3.Postorder Traversal
Enter 0 to exit
Enter your choice: 2
The preorder traversal =         12          2
651.Inorder traversal
2.Preorder Traversal
3.Postorder Traversal
Enter 0 to exit 0
```

Reconstruction of the Tree

We have learned the different tree traversal techniques in the last section. Now, we will discuss how to construct the tree if the different traversals of the tree are given. There are two possibilities of tree construction if

1. in-order and pre-order traversals are given;
2. in-order and post-order traversals are given.

We know that in pre-order traversal, the root node is traversed first and in the post-order traversal the root node is traversed last. Therefore, we can easily find the root of the tree. In in-order traversal, left subtree is traversed first, followed by the root node and then the right subtree. After finding the root of the tree, we can find the left subtree and right subtree of the root node by looking into the in-order traversal. The root comes in between the elements forming the left subtree and the elements forming right subtree. Therefore, all the elements which come before the root node in the in-order traversal will form the left subtree and the elements which come after the root node in the in-order traversal will form the right subtree. By repeating the process with the left subtree and the right subtree, we can easily construct the tree.

Problem 7

Traverse the following tree:

In-order:	23	26	43	78	94	97
Pre-order:	**78**	26	23	43	94	97

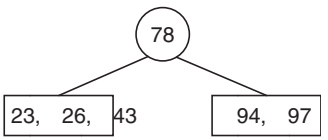
Solution

Step 1: The first element in the pre-order traversal is the root of the tree as shown below in boldface.

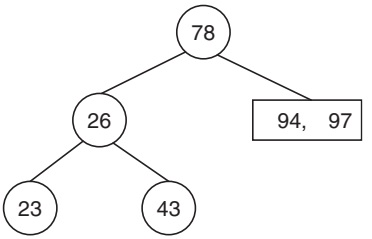
In-order:	23	26	43	78	94	97
Pre-order:	**78**	26	23	43	94	97

Step 2: Now, find the position of root (shown in boldface) in the in-order traversal. All the elements coming before the root (23, 26, 43 shown in italics) will form the left subtree, and the elements coming after root (94, 97 shown in italics) will form the right subtree.

In-order:	23	26	43	**78**	94	97
Pre-order:	78	26	23	43	94	97

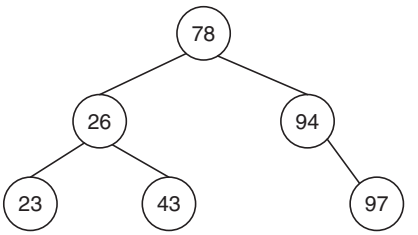


Step 3: The next element that comes in pre-order traversal is 26, so it will form the root of the left subtree. Now, find the elements (out of 23, 26, 43) that come before 26 (that is, 23 shown in italics) in in-order traversal to form the left subtree of 26 and that comes after the 26 (that is 43 shown in italics) to form the right subtree of 26.



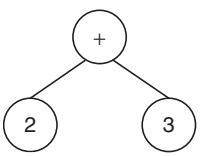
Step 4: Now, take the right subtree and check which element among the elements of the right subtree comes first in the pre-order traversal. Here we can see that 94 comes first, so it will be the root of the right subtree. Now check into the in-order traversal, which of the elements of the right subtree comes before 94 to form the left subtree of 94 and which of them comes after 94 to form the right subtree of 94.

In-order:	23	26	43	78	**94**
Pre-order:	78	26	23	43	94



7.5 Expression Tree

The mathematical expressions consists of operands and the operators. The binary tree corresponding to the mathematical expressions is called expression tree. The expression trees are constructed by using the operands; the operators (logical or arithmetic) and their order of precedence and associative law of operators. For example, the parentheses are evaluated first then the exponent, followed by multiplication or division and then addition or subtraction. The multiplication and division is associated from left to right so whichever comes first will be evaluated first. Similarly the addition and subtraction is also associative from left to right. The expression tree can easily be constructed by dividing the expression into subexpression and then combining them according to the order of precedence. Although the expression may contain parentheses, but these are not included into the expression tree. The root and the internal nodes of the expression tree will always contain the operators and the leaf node will contain the operands. For example, the expression tree corresponding to the $a + b$ is



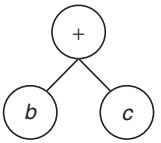
Problem 8

Construct the expression tree for the expression

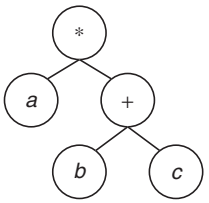
$$a*(b+c)/d - e$$

Solution

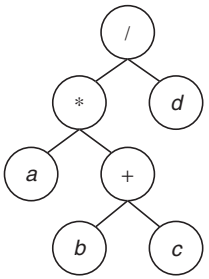
Step 1: First the parenthesis is evaluated so construct the tree for $(b + c)$.



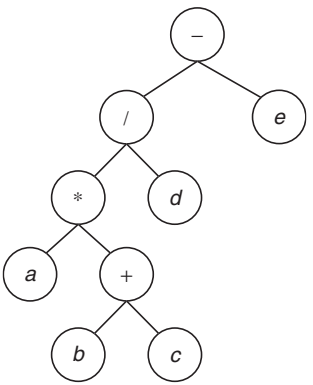
Step 2: The result of $(b + c)$ will be multiplied by a so now construct the expression tree for $a*(b + c)$.



Step 3: The result of $a*(b + c)$ will be divided by d so now construct the expression tree for $a*(b + c)/d$.



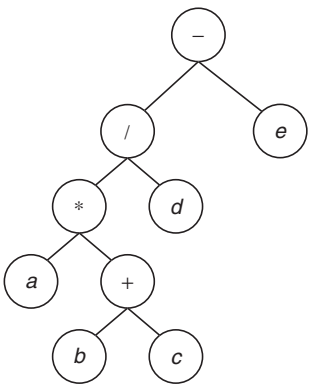
Step 4: Operand e is subtracted from the result of $a*(b + c)/d$ so now construct the expression tree for $a*(b + c)/d - e$.



Since expression is also a binary tree so it can also be traversed in three different ways:

1. in-order;
2. pre-order;
3. post-order.

These traversal techniques of the expression tree produce the different polish forms of the arithmetic expression. When the expression tree is traversed in the in-order way, it produces the infix form of the expression. When it is traversed in pre-order way, it produces the prefix form of the expression in which every operator is written before its operand. And when it is traversed in post-order way, it produces the postfix form of the expression in which every operator is written after its operand. We can see the result by traversing the following tree:



In-order traversal: $((a*(b+c))/d)-e$
Pre-order traversal: $-/*a+bcde$
Post-order traversal: $abc+*d/e-$

7.6 General Tree

The binary tree is defined as a tree, which can have maximum two children. A general tree is defined as a tree, which can have unlimited children so any node of the general tree can have unlimited out-degree as shown in Figure 9.

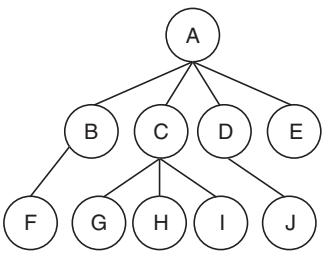


Figure 9 A general tree.

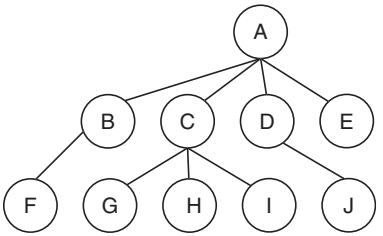
Therefore, it is unpredictable to guess the number of children associated with each node. For example, in binary tree a node can have maximum two children so it is better to represent the general tree in the

form of binary tree. All the nodes of a binary tree will be the same as the node of the general tree. The general tree can be converted into binary tree as follows:

1. Keep the link of the leftmost child of parent node intact and remove all other links with the parent node.
2. Now add the links between the siblings or immediate right child of any node.

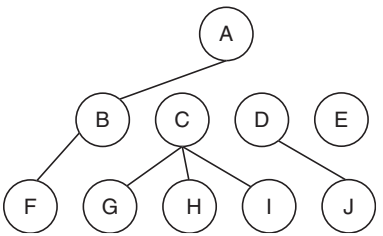
Problem 9

Convert the following general tree into the binary tree.

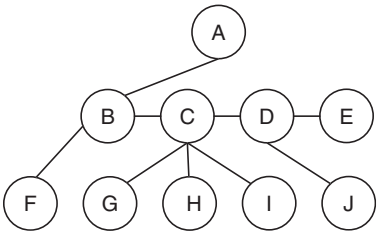


Solution

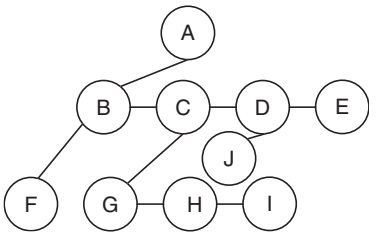
Step 1: Keep the link of the leftmost child of parent node intact and remove all other links with the parent node.



Step 2: Now add the links between the siblings or immediate right child of any node.



Step 3: Repeat the process with the left subtrees and right subtrees so the final binary tree is as follows:



7.7 Threaded Binary Tree

We can see in the binary tree as shown below that many of the entries are null entries (i.e., ×) which can be considered as the wastage of the memory space. This memory space can be utilized more efficiently by replacing the null pointer entries with some special pointers which point to the nodes higher in the tree.

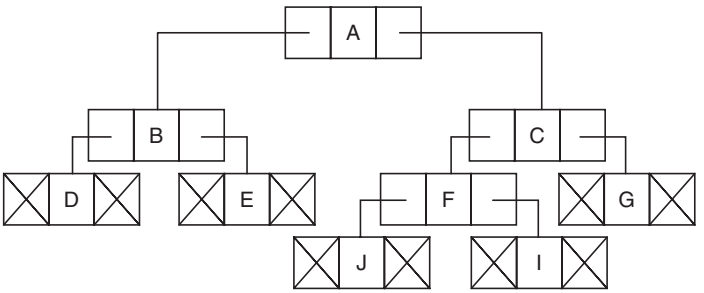


Figure 10 A binary tree.

Threaded binary tree is a special kind of binary tree in which these null entries of the left and right subtrees are replaced by a special pointer called thread. A thread is a special link that not only avoids the null entries, but it also simplifies the traversal on binary trees. We have seen the use of stack in the non-recursive traversal of a tree which requires extra memory and pointers to keep the addresses. This problem can also be avoided by the threaded binary tree and it makes the in-order traversal of a tree easier without the use of stack. When the threaded binary tree is stored into the computer memory, an extra field is used to represent the thread so that it can be differentiated with the normal pointers also. A binary tree is threaded according to the particular traversal order in two ways.

1. **One-way threading:** The null entry of the right field of the node is replaced by the successor of the node using any of the traversal (in-order, pre-order or post-order) of the tree. The node of the tree in one-way threading will contain an extra field to represent the thread as follows

Left	Info	Right	Rthread

   The Rthread field is 0 if there is a thread, otherwise it will be 1.
2. **Two-way threading:** The null entry of the right field of the node is replaced by the successor of the node in any of the traversal (in-order, pre-order or post-order) ordering of the tree. Null entry of the left field of the node is also replaced by the predecessor of the node using any of the traversal ordering of the tree.

Lthread	Left	Info	Right	Rthread

   The Lthread and Rthread field is 0 if there is a thread otherwise it will be 1.

Problem 10

Construct the one-way threaded binary tree for the binary tree given in Figure 10.

Solution

The threaded binary tree can be constructed according to the traversal applied. Let us construct the one-way threaded tree for the in-order traversal and pre-order traversal. The traversal sequence of the tree is given as follows.

In-order:	D	B	E	A	J	F	I	C	G
Pre-order:	A	B	D	E	C	F	J	I	G

The threads are represented by the dotted line. In one-way threading, the right null field of the node is replaced by the successor of the node given in a traversal sequence.

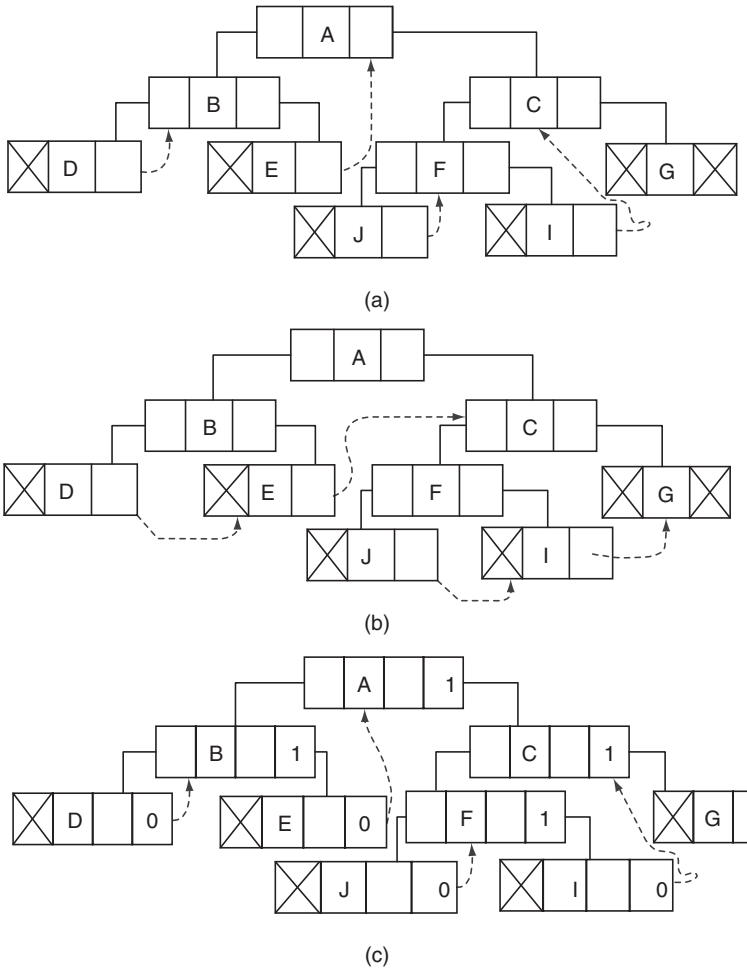


Figure 11 (a) One-way in-order threading; (b) one-way pre-order threading; (c) one-way in-order threading.

In-order:	D	B	E	A	J	F	I	C	G
Pre-order:	A	B	D	E	C	F	J	I	G

The right null field of the node *D* is replaced by the thread pointing to node *B*; the in-order successor of node *D* in Figure 11(a). Since there is no in-order successor of node *G*, hence its right field contains Null. Similarly, we can construct the pre-order one-way threaded binary tree as shown in Figure 11(b). The tree can be represented by using the nodes for treaded binary tree as shown in Figure 11 (c) for one-way in-order threading.

Problem 11

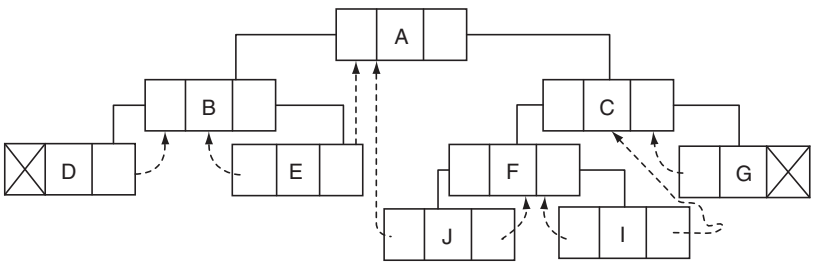
Construct the two-way threaded binary tree for the binary tree given in Figure 10.

Solution

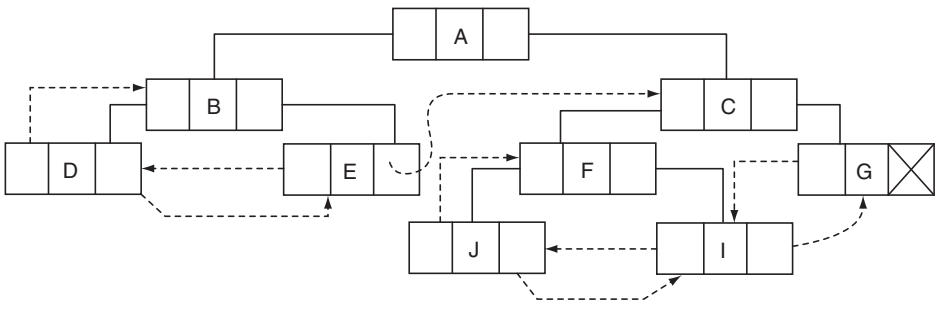
Let us construct the in-order two-way threaded binary tree and the pre-order two-way threaded binary tree. The traversal sequence of the tree is given as follows.

In-order:	D	B	E	A	J	F	I	C	G
Pre-order:	A	B	D	E	C	F	J	I	G

The threads are represented by the dotted line. In two-way threading, the right null field of the node is replaced by the successor of the node and the left null field of the node is replaced by the predecessor of the node given in a traversal sequence.



(a)



(b)

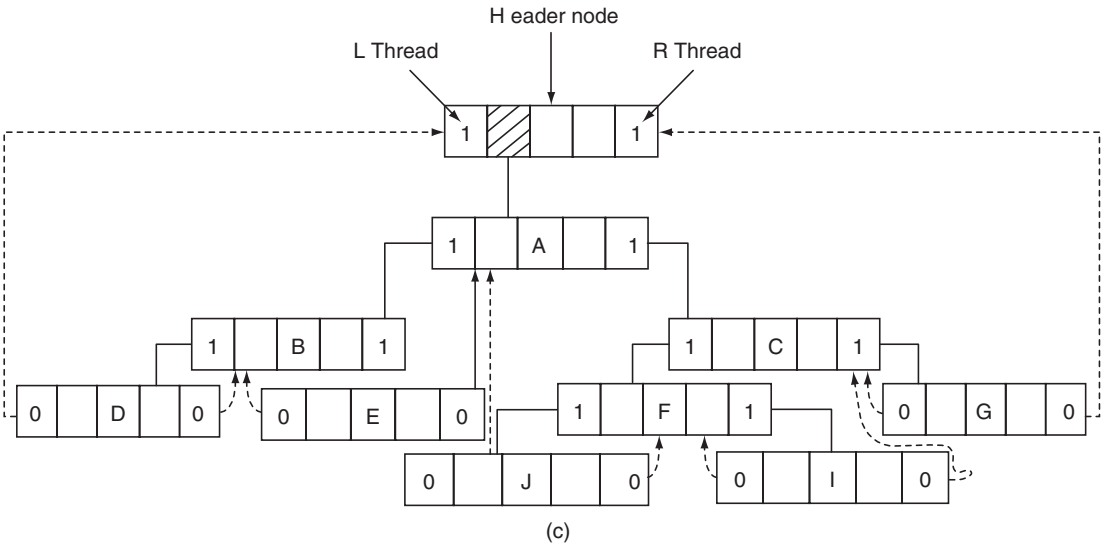


Figure 12 (a) One-way in-order threading; (b) one-way pre-order threading; (c) two-way in-order threading.

In-order:	D	B	E	A	J	F	I	C	G
Pre-order:	A	B	D	E	C	F	J	I	G

The right null field of the node *E* is replaced by the thread pointing to node *A*; the in-order successor of node *E* and the left null field is replaced by *B*; the predecessor of node *E* is shown in Figure 12(a). Since there is no in-order successor of node *G*, hence its right field contains Null; and similarly there is no predecessor of node *D* so its left field contains Null. Similarly, we can construct the pre-order one-way threaded binary tree as shown in Figure 12(b).The null entries of the First (*D*) and Last (*G*) leaf node can also be removed by using a special node called header node, and such threaded trees are called header threaded tree. The header threaded tree is shown in Figure 12(c) with the nodes represented as the threaded node containing the parts for Rthread and Lthread.

7.8 Binary Search Tree

A binary search tree is a special kind of binary tree that satisfies the following conditions.

1. The data elements of the left subtree are smaller than the root of the tree.
2. The data elements of the right subtree are greater than or equal to the root of the tree.
3. The left subtree and right subtree are also the binary search trees, that is, they must also follow the above two rules.

Construction of a Binary Search Tree (BST)

In fact, we have already discussed the method to construct the BST in the construction of the binary tree. This is the common method of constructing the binary tree.

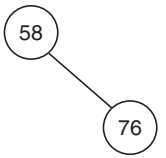
Problem 12

Construct the BST of 58, 7 6, 14, 63, 63, 7 8, 43, 6, 11, 61.

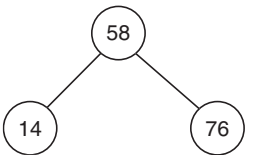
Solution

Step 1: Initially, the tree is empty so place the first number at the root.

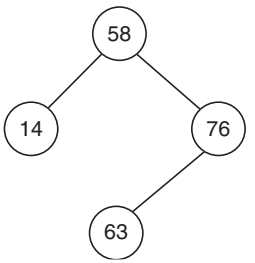
Step 2: Compare the next number (76) with the root. If the incoming number is greater than or equal to the root then place it in the right child position, otherwise place it into the left child position.



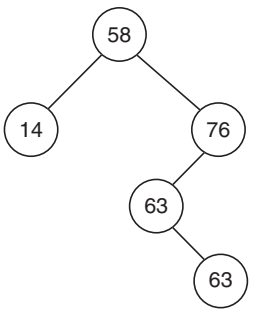
Step 3: Compare the next number (14) with the root. This is less than the root so examine the left subtree.



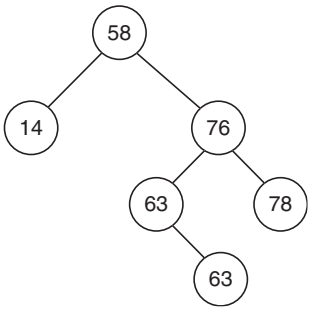
Step 4: Now, the next incoming number 63 is greater than 58 so go to the right subtree where compare it with 76. Since 63 < 76 so place it in the left child's position of 76.



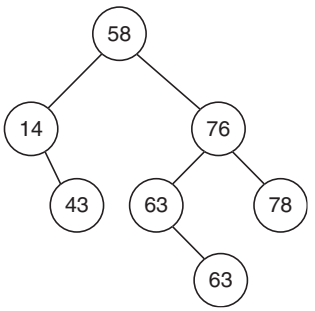
Step 5: Repeat the process for the next number 63.



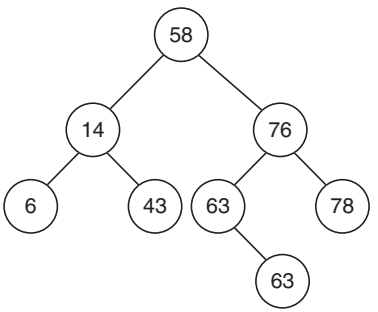
Step 6: Repeat the process for the next number 78.



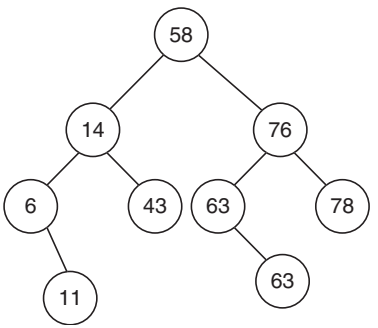
Step 7: Repeat the process for the next number 43.



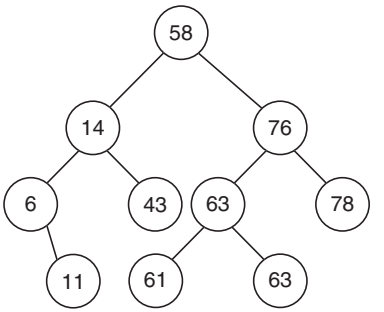
Step 8: Repeat the process for the next number 6.



Step 9: Repeat the process for the next number 11.



Step 10: Repeat the process for the next number 61.



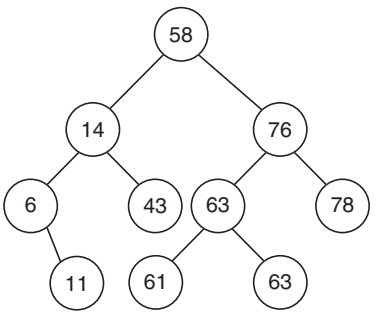
Now the in-order traversal (LNR) of the BST will result in 6, 11, 14, 43, 58, 61, 63, 63, 76, 78.

Operations on the BST

Traversal of BST

BST is also a binary tree so it can be traversed in three ways:

1. Pre-order;
2. post-order;
3. in-order.



The different traversals of the tree shown in Problem 12 and given above are as follows:

1.	**Pre-order:**	58	14	6	11	43	76	63	61	63	78
2.	**Post-order:**	11	6	43	14	61	63	63	78	76	58
3.	**In-order:**	6	11	14	43	58	61	63	63	76	78

We can see that the in-order traversal of BST produces the list of tree elements in ascending order. This is an important property of BST that in-order traversal of it arranges the tree elements in ascending order.

Insertion into the BST

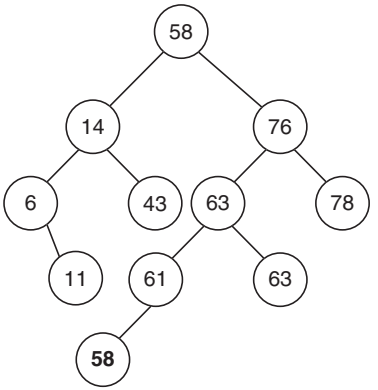
This operation has already been explained in the previous example. The insertion of an element into BST follows the process as given below:

1. Compare the number with the root of the BST.
2. If the number is less than the root value then follow the left subtree.

3. If the number is greater than the root value then follow the right subtree.
4. Apply the same process (1, 2 and 3) to the left subtree and right subtree if required.

Let us insert 58 into the tree given in Problem 12.

Compare 58 with the root (58); both are equal. So proceed to the right subtree. Now, compare the number 58 with the root of the right subtree that is 76, now 58 < 76. Therefore, proceed to the left subtree at 76. Again compare the number 58 with root of the left subtree, that is, 63; now 58 < 63. Therefore, again proceed to the left subtree at root 63. Now compare 58 with 61; since 58 < 61, so proceed again to the left subtree and insert the node at the left of the root 61 so the tree will be as shown below.



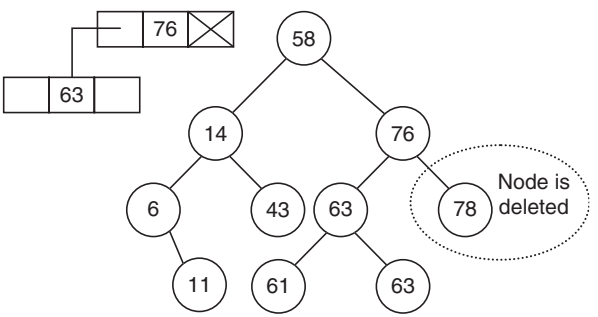
Deletion from the BST

The deletion of data element from the BST can be performed in three different ways as follows:

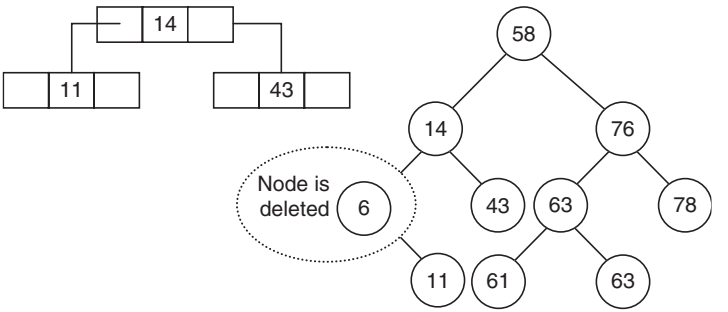
1. Deletion of the leaf (terminal) node.
2. Deletion of the node having only one child.
3. Deletion of the node having two children.

Now, lets us discuss each of the case in detail.

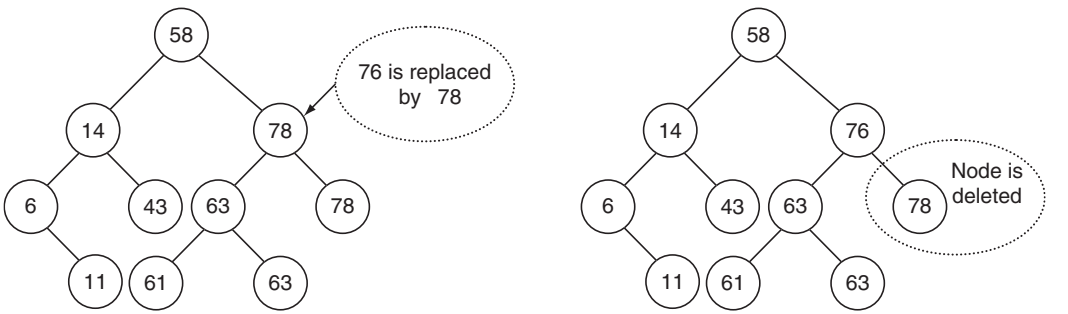
1. **Deletion of the leaf node:** In this case, we have to change the deleted node's entry in the parent node to Null. For example, if from the tree given in Problem 12, a node containing 78 is to be deleted then we have to change the right field of its parent node containing 76 to null. Therefore, the parent node will now change as shown below. The BST after deleted element is also shown below.



2. **Deletion of a node having only one child:** In this case, there are two possibilities the node may have the left child only or the right child only. If there is left child only then we need to attach the node's left child to the node's parent in place of the deleted node. And if there is a right child only then we need to attach the right child to the node's parents in place of the deleted node. For example, if the node containing 6 is to be deleted from the tree shown in Problem 12 then we have to change the left field of node's (6) parent to point to the left child of the deleted node so tree will be changed into as shown below



3. **Deletion of a node having two children:** In this case, the data element can be deleted from the middle of the tree also, but the structure and integrity of the BST will not be maintained. Therefore, in this we replace the node (to be deleted) by the in-order successor of that node. Therefore, first find the in-order successor of the node then replace the selected node with the in-order successor and then delete the in-order successor of the node from the tree. For example, if we want to delete the node containing 76 from the tree shown in Problem 12 then first we have to find the in-order traversal sequence of the tree that is 6 11 14 43 58 61 63 63 76 78. In the in-order sequence, we can see that the in-order successor of the selected node (76) is 78 so first replace 76 by 78 and then delete the node containing 78.



7.9 Balanced Tree

In the preceding section, we have discussed the binary search tree and it makes searching easier if it contains half of the elements in the left subtree and almost half of the elements in its right subtree. The left and right subtrees themselves follow the same composition. However, it is not possible all the time because it depends on the elements inserted. For example, consider the following BST.

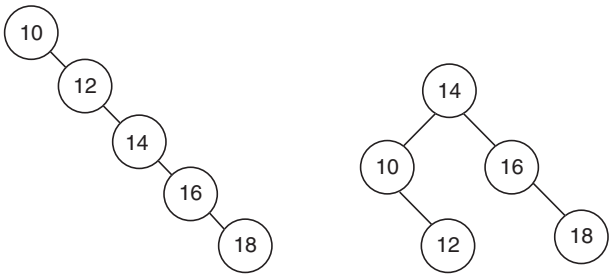


Figure 13 (a) Unbalanced BST; (b) balanced BST.

This BST in Figure 13(a) requires 5 comparisons to search 16 in the tree and 4 comparisons to search 14. However, if this is converted into the tree as shown in Figure 13(b) it will require only 1 comparison to search 14 and 2 comparisons to search 16. Here, the tree is balanced. Before discussing the balanced tree, lets us first discuss the balance factor.

Balance Factor (BF) = Height of the left subtree − Height of the right subtree

$$BF = H_L - H_R$$

For example, the height of the different nodes are assigned in Figures 14(a), (b) and (c).

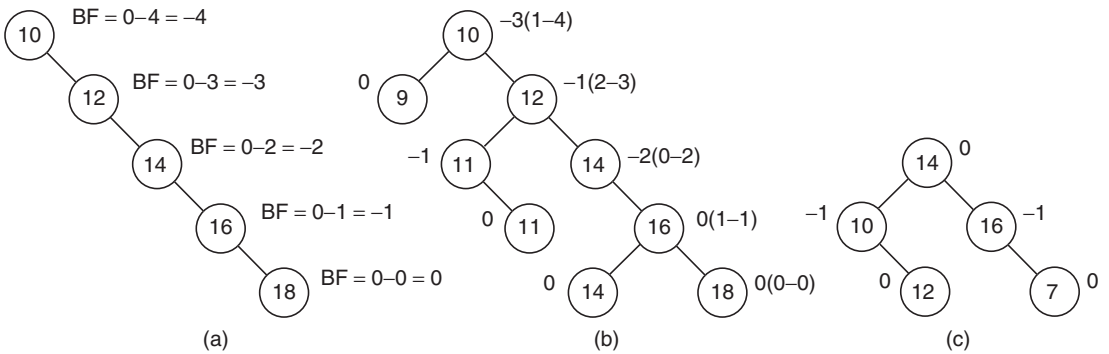


Figure 14 (a) Unbalanced BST; (b) unbalanced BST; (c) balanced BST.

Therefore, the searching depends on the height of the tree in the BST, and the height changes as the insertion or deletion takes place. The insertion and deletion of nodes makes the tree unbalanced. There are different kinds of balanced trees which are balanced by working with their heights. Some of such trees are as follows:

1. AVL tree.
2. B-tree.
3. 2-3 tree
4. Red black tree.
5. Splay tree.

Two of these are explained in the following subsections.

AVL Tree

AVL tree is a special kind of balance tree in which the balance factor of each node cannot be other than 0, – 1 or 1. The AVL tree was named after the Russian scientists G.M. Adelson, Velski and E.M. Landis. It is also called as a height-balanced tree. In other words, we can say that the height of the left subtree and right subtree of the node differs at the most by 1 where left and right subtrees are again AVL. If the balance factor of the node is −1, then the right subtree is said to be higher than the left subtree. It is called right heavy and if the balance factor is 1, then left subtree is said to be higher than the right subtree. It is called left heavy and if the balance factor is 0, then both subtrees are on same height.

Representation of AVL Tree

The AVL tree can easily be implemented in the memory like other trees with an additional field to keep track of balance factor. The balance factor of a node represents the difference of heights between the left and right subtrees of the node. Therefore, each node of the AVL tree will be represented as follows.

The nodes in the AVL trees are divided into four fields, the left, right pointer, info and the balance factor. It can be represented by the programming languages C and C++ as follows:

In C

```
struct node
{
node *left;
int info;
node *right;
int BF;
};
struct node * ro
```

Remember to use `struct` keyword

In C++

```
struct node
{
node *left;
int info;
node *right;
int BF;
};
onto;de *root;
```

The `struct` keyword is not required

Building a Height Balanced Tree

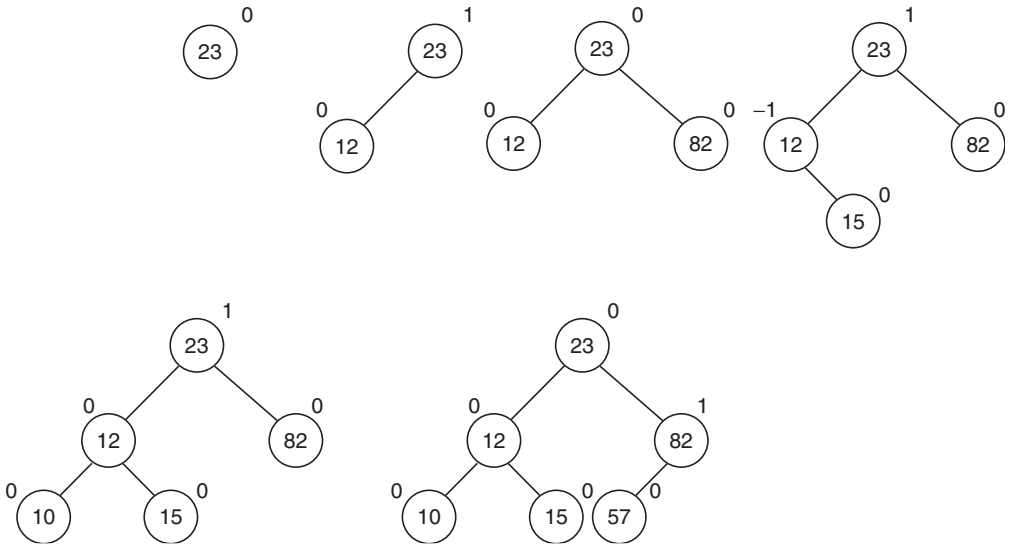
The AVL tree is constructed by using the same procedure as applied to construct the BST. If the new element is smaller than the root value then it examines the left subtree; and if it is greater than or equal to the root then it examines the right subtree. The only thing we must remember while constructing the AVL (height balanced tree) tree is the property of the height balanced tree. The balance factor of each node can be 0, −1 or +1. While the insertion or deletion is performed, the tree becomes unbalanced and violates the property of AVL tree. There are different rotation methods to balance the tree. We will discuss each rotation in this section. Let us start with the simple example of constructing the AVL tree.

Problem 13

Construct the AVL tree for the following list.
23, 12, 82, 15, 10, 57

Solution
The nodes of the tree along with the balance factor are shown below. The balance factor (BF) is calculated as follows:

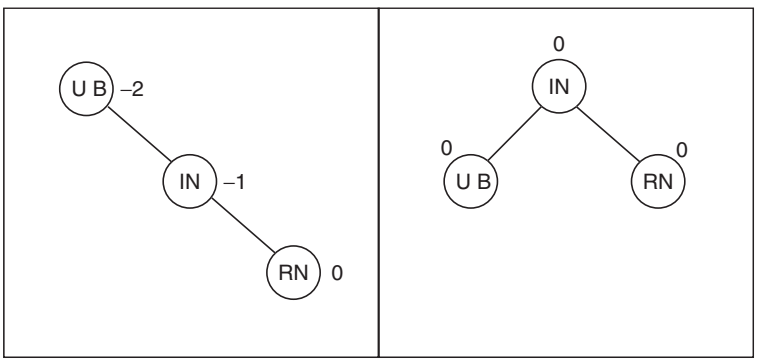
BF = Height of left subtree − Height of right subtree



Rotations

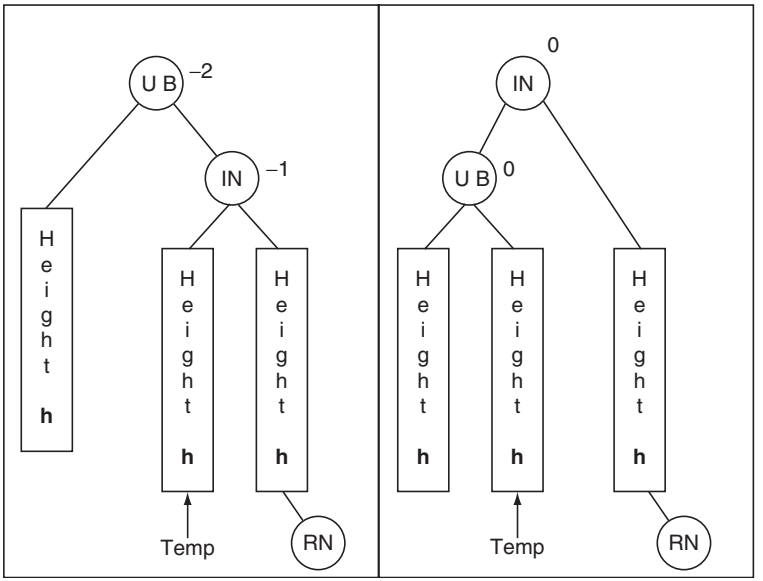
There are two kinds of rotations, which are applied to balance the tree and they are

1. **Left rotation:** If the balance factor of the unbalanced node is −2 and the most recently inserted node lies in the straight line along with the unbalanced node, then single left rotation is applied at the closest parent to the newly inserted item. The left rotation is performed in counter clockwise direction.



Here we can see that the recently inserted node RN and the unbalanced node UB lies in the straight line along with the intermediate node IN in the tree. The balance factor of UB is −2 so we need to apply the left rotation. In other words, we can say that if the tree becomes unbalanced due to the recently added item at the right subtree of the right subtree of the unbalanced node, then we apply

the left rotation. Now this is a very simple example, but if tree is already there with some left or right subtrees attached with the nodes as shown below and insertion of new element creates the unbalanced tree then how to perform the left rotation.

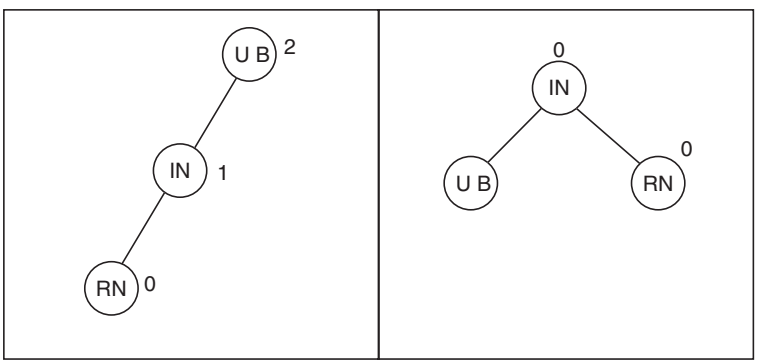


Therefore, the left rotation is performed as follows:

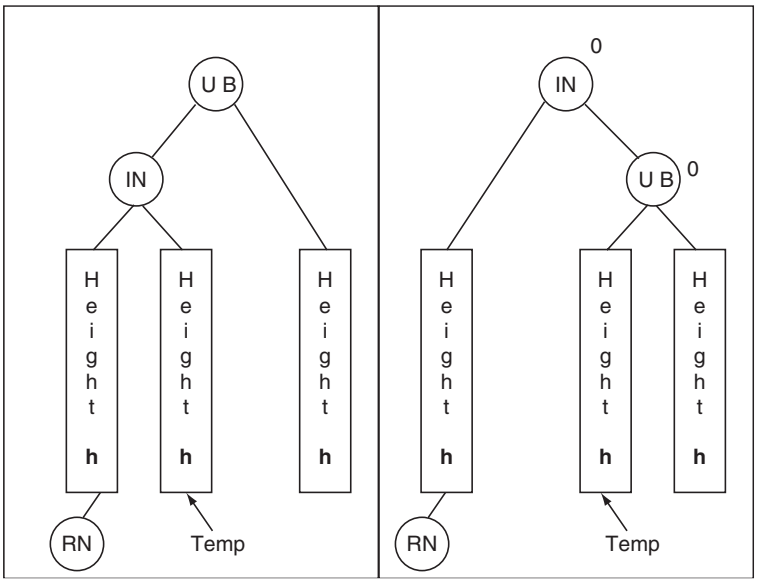
```
IN = Right (UB)
Temp = Left (IN)
Left (IN) = UB
Right (UB) = Temp
```

Here we can see that the unbalanced node loses its right subtree because the right subtree becomes the root. Therefore, at least one leaf node (`Temp`) from the right subtree is shifted to left subtree so that it preserves the search tree relationship.

2. **Right rotation:** If the balance factor of the unbalanced node is +2 and the most recently inserted node lies in the straight line along with the unbalanced node, then single right rotation is applied at the closest parent to the newly inserted item. The right rotation is performed in the clockwise direction.



Here we can see that the recently inserted node RN and the unbalanced node UB lie in the straight line along with the intermediate node IN in the tree, and the balance factor of UB is +2. Therefore, we need to apply the right rotation. In other words, we can say that if the tree becomes unbalanced due to the recently added item at the left subtree of the left subtree of the unbalanced node, then we apply the right rotation. Now, this is a very simple example but if tree is already there with some left or right subtrees attached with the nodes as shown below and insertion of new element creates the unbalanced tree, then the question of arises: How to perform the left rotation?



Therefore, the right rotation is performed as follows

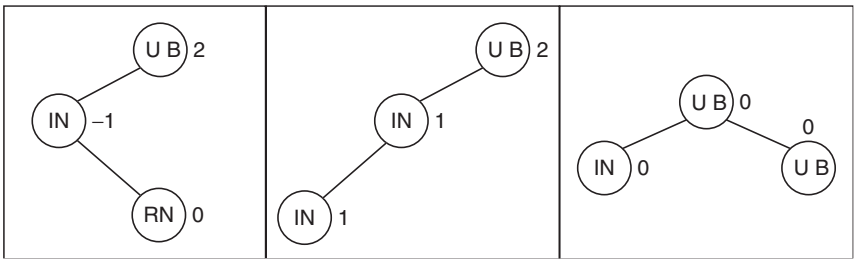
```
IN = left(UB)
Temp = right(IN)
right(IN) = UB
left(UB) = Temp
```

Here we can see that if a tree needs right rotation for balancing, then at least one leaf node (Temp) from left subtree is shifted to right subtree.
When the rotations are performed, we have to consider the following things:

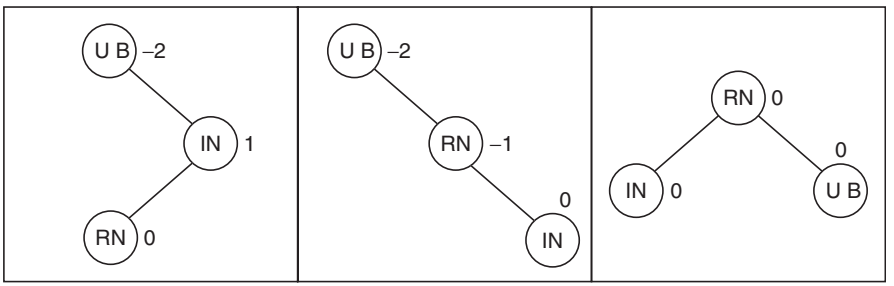
1. Consider the path from the recently inserted node RN to the root UB.
2. Consider only two layers below the unbalanced node UB.
3. The rotation is performed on the immediate ancestor IN of the newly inserted node.
4. If RN, IN and UB are in a single straight line and they are at the right subtree of the root (UB); that is, the BF of UB is – 2 then we need to perform left rotation.
5. If RN, IN and UB are in a single straight line and they are at the left subtree of the root; that is, the BF of UB is +2 then we need to perform right rotation.
6. If they are not in the straight line or they form the dog legs like structure or bend, then we need to perform the double rotation. Double rotation is a combination of two single rotations in opposite direction. The first rotation is applied to the node which creates the bend, and then

second rotation is applied with including the node creating imbalance. These situations are described as follows:

a. If the recently added item (RN) is at the right subtree of the left subtree of the root (UB), then we need to perform the left rotation at (IN) first followed by the right rotation. In other words, we can say that if the closest parent of the newly inserted item has the BF – 1 and the BF of the unbalanced node is +2, then we need to perform the left–right (LR) rotation. The situation is shown below



b. If the recently added item (RN) is at the left subtree of the right subtree of the root (UB), then we need to perform the right rotation at (IN) first then left rotation at RN. In other words, we can say that if closest parent of the newly inserted item has the BF as 1 and the BF of the unbalanced node is – 2, then we need to perform the right–left (RL) rotation. The situation is shown below.

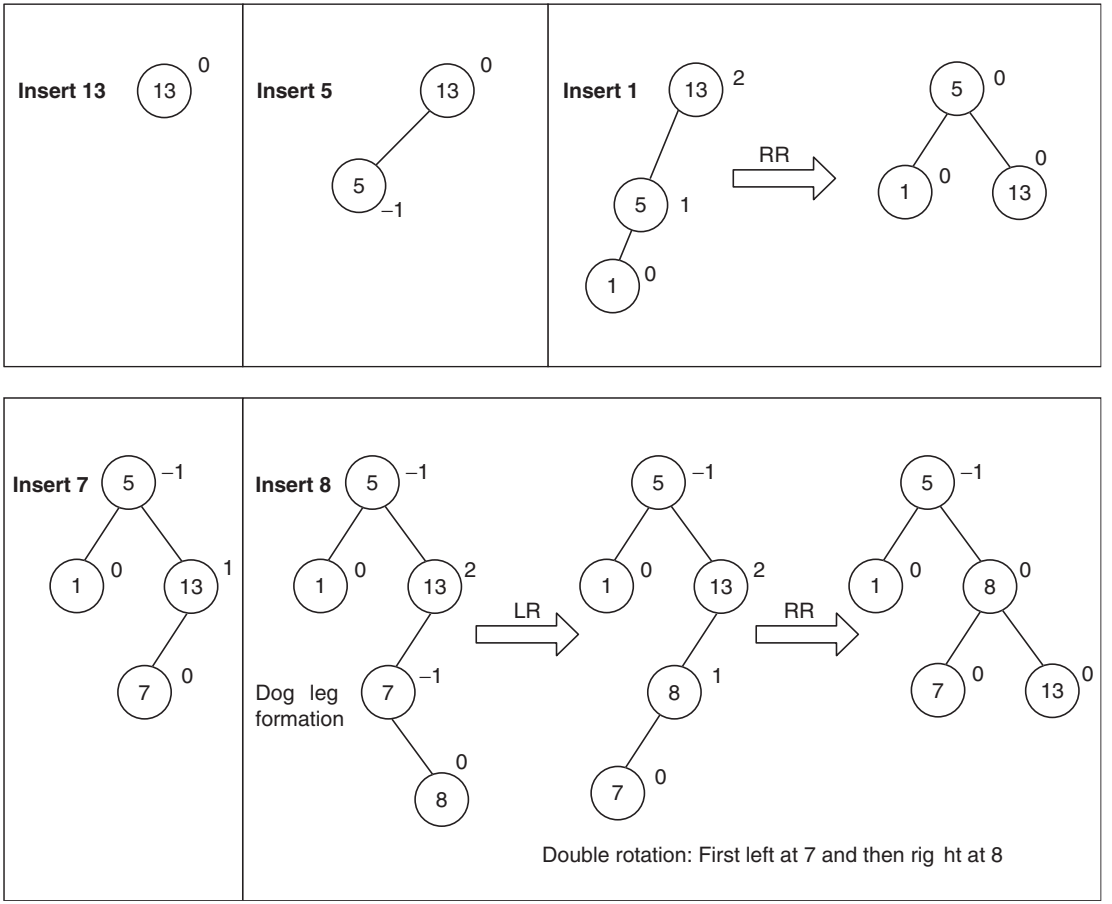


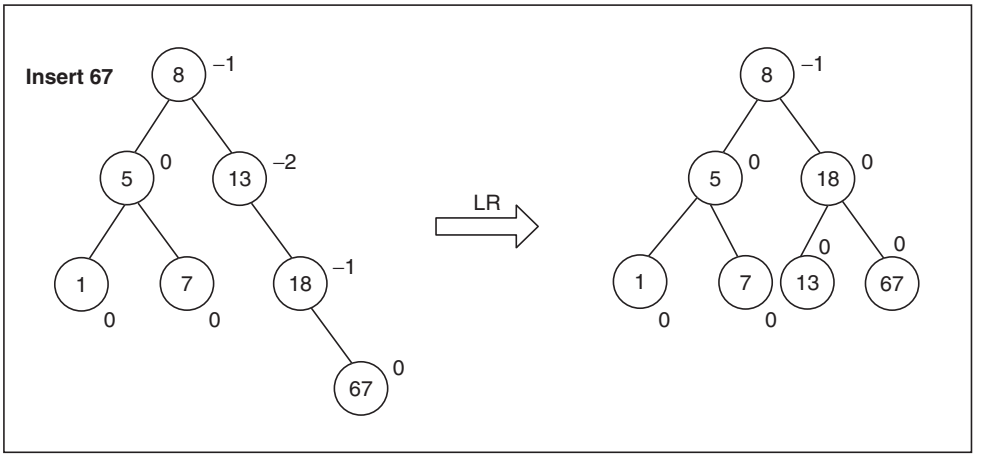
7. A very important fact is that the balancing of the tree does not change the sequence of elements in the in-order traversal. Hence the in-order traversal of the tree after balancing it will produce the same sequence as produced by the in-order traversal of the unbalanced tree. Therefore, we can also verify the correctness of the tree after balancing it.
8. The unbalanced tree has the worst-case complexity O(n) for searching an element in the tree, but the balanced tree takes only O(log n) time complexity in the worst case.
9. The balancing of the tree preserves the search tree relationship; that is, all the nodes in the right subtree of the node must have greater or equal values to the node's value and the left subtree must have smaller value than the node's value.

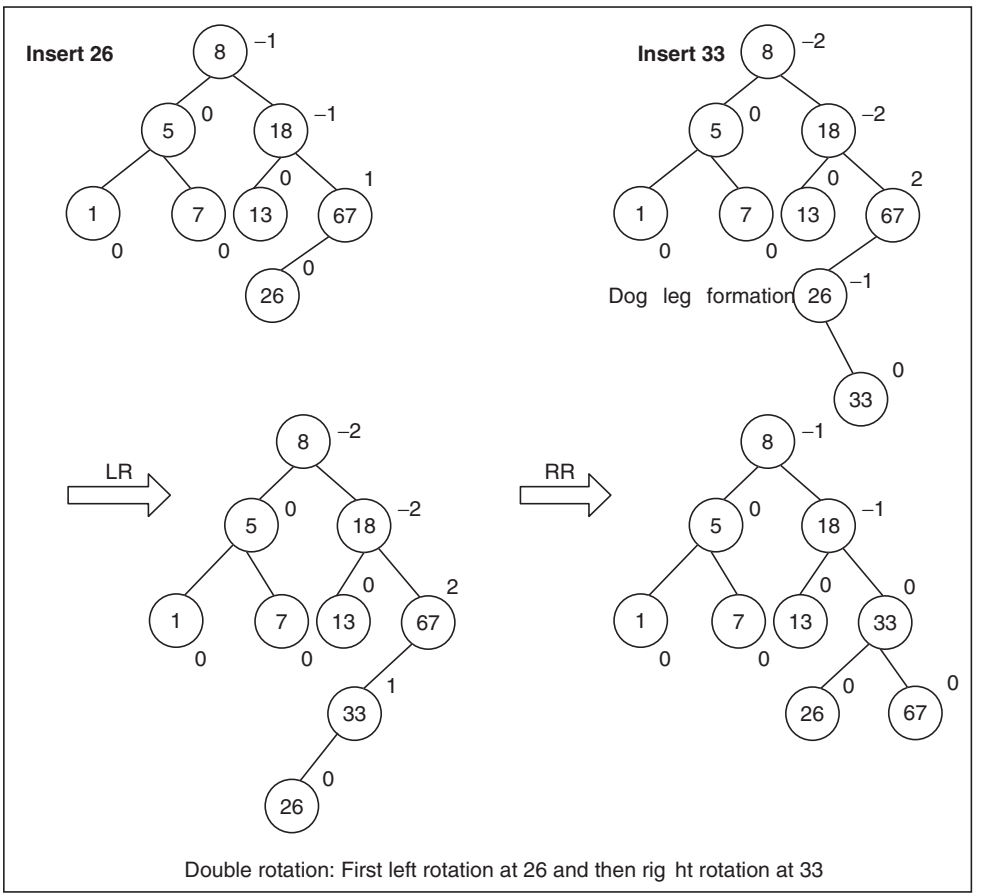
Problem 14

Construct the AVL tree for the following set of elements.
13, 5, 1, 7 , 8, 98, 67 , 26, 33, 12, 6, 7 , 8

Solution


Insert 13
13
0
Insert 5
13
0
5
–1
Insert 1
13
2
5
1
1
0
RR
5
0
1
0
0
13
Insert 7
5
–1
1
0
13
1
7
0
Insert 8
5
–1
1
0
13
2
7
–1
8
0
Dog leg formation
LR
5
–1
1
0
13
2
8
1
7
0
RR
5
–1
1
0
8
0
7
0
13
0
Double rotation: First left at 7 and then rig ht at 8
Insert 18
5
–2
1
0
8
–1
7
0
13
–1
18
LR
8
0
–1
5
0
13
–1
1
0
7
0
18
0





Double rotation: First left rotation at 26 and then rig ht rotation at 33

B-Tree

In the preceding section, we have studied the different kinds of binary trees such as BST, AVL and expression tree. All these trees have the limitations on the number of children they can have (at the most 2). Therefore, as the number of elements increase, the height of the tree also increases which makes different operations on the tree time consuming. Another point is that the leaves of such tree may not be on the same level, so searching takes different time duration for different elements depending on their position in the tree. If the element is positioned at higher height, it will take more time and if it is positioned at lower height it will take less time. Therefore, we can say that height is the major concern for efficient operations on the tree.

We know that BST makes the searching easy because the left subtree of the node contains all the elements which are less than the node value, and right subtree contains all the elements which are greater than or equal to the node value. Hence, searching is done by following a single path depending on the value of the key element which we want to search. However, the limitation of BST is that it cannot have more than two branches at each node, so the height grows as the number of elements grows. The AVL tree has the disadvantages of disk access proportional to the depth of AVL tree and performance of AVL tree is not good for the large volume of data. Therefore the alternative solution is M-way tree in which every node has multiple children (M-way means *M* branches). Multiway (M-way) search tree is an extension of the concept of BST that removes all these drawbacks and is defined as follows:

The M-way search tree is a search tree with the following properties:

1. Each node can contain $N-1$ key elements, where N is the order of the tree.
2. Each node can have N branches.
3. It satisfies the property of BST, that is, the elements with smaller value than the key are positioned into the left subtree and the elements with greater or equal values are positioned into the right subtree of the key.
4. All subtrees are themselves M-way trees.

1. The M-way search tree reduces the height of the tree substantially so it can provide shorter search path length as compared to BST for same set of keys. The depth of the M-way tree is $O(\log_m n)$ instead of $O(\log_2 n)$ therefore number of disk access also decreases. The disadvantage of M-way tree is that it is not balanced. R. Bayer and E. McCreight in 1970 at the Boeing Scientific Research Lab proposed the new data structure called B-tree. Here the B is not defined, but it is believed that it may stand for Balanced, Bayer or Boeing. The B-tree is defined as follows:

   - A B-tree is an M-way search tree.
   - The root is either a leaf or it has 2 to N subtrees, where N is the order of the B-tree.
   - All nodes can contain maximum $N-1$ elements.
   - All nodes except the root and the leaves can have minimum $N/2$ and maximum N branches (subtrees).
   - All leaf nodes are on the same level so the tree is balanced.
   - It retains the general property of BST; that is, all elements in the left subtree of a key element are predecessors of the key element and that on the right are successors of the key element.
   - Each node contains the key elements and the pointers to the children.
   - When a new key is inserted into full node (with $N-1$ elements), then the node is split into two new nodes at the same level and the key with the median value is inserted in the parent node in case the parent node is the root, a new root will be created. The full node condition is called the overflow.

- There must be no empty subtrees above the leaves of the tree.
- All the key elements are arranged in a defined order within the node.

Since the leaf nodes in a B-tree are on the same level, so searching of any element in any branch requires equal time or it requires the same number of access. The B-tree is a very popular data structure for organizing the index structure in the files, and it is most commonly used in databases and the file system. It is used for placing and locating the information in a file. It minimizes file access because height of the tree is kept as small as possible, by keeping all the leaf nodes on the same level. It finds its use in external sorting. The B-tree of order n is also called $N - N - 1$) tree, where the first entry N) represents the maximum number of branches and the second entry ($N - 1$) represents the maximum number of elements in each node.

Insertion into the B-Tree

1. First, find the position in the tree where the element can be inserted. If the node does not violate the B-tree condition, that is, if the position is found at non-full node then insert the element into that position in proper order.
2. If the node is a full node then the node splits at the median element into two nodes on the same level, each with the minimum number of elements. This process continues recursively up the tree until the root is reached; if the root is spilt then a new root is created.
3. The median key is not put into either of the two new nodes, but is instead sent up the tree to become a new root and to be inserted into the parent node.
4. If the order of the B-tree is an odd number, then the median is found by using $N/2 + 1$. And if the order is an even number then median is found by using $N/2$.

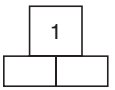
The process of insertion into the B-tree is explained with the help of an example.

Problem 15

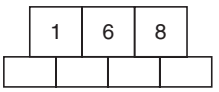
Construct the B-tree of order 4 for the following elements.
1, 6, 8, 2, 9, 12, 15, 7 , 18, 3, 4, 20

Solution
Since the order of the B-tree is 4, so each node can contain maximum 3 elements and can have maximum 4 children. Each node contains the elements and the pointer to their children.

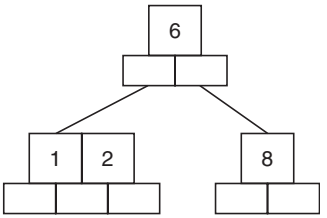
Step 1: Insert the first element (1) into the empty new node, root can have minimum two branches as shown below



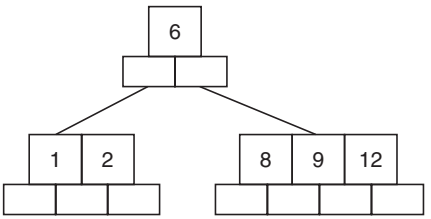
Step 2: Insert the next two elements (6 and 8) into the root node. These elements can be inserted into the root node because each node can contain maximum 3 elements as shown below



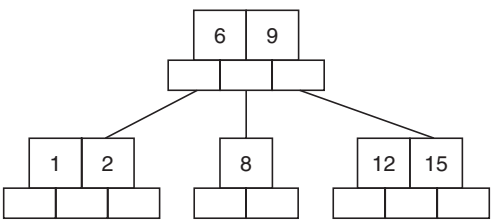
Step 3: Insert the next elements (2) into the root node. However, this node is already full, so split this node into two new nodes at the same level at the medial element (4/2 = 2; the element at position 2 is 6). And then insert the element into its proper position by using BST property. Compare the next element with root 6 since 2 < 6, so it will follow the left subtree.



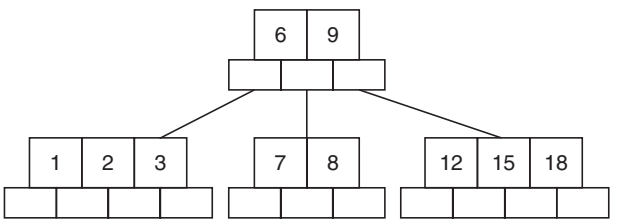
Step 4: Insert the next two elements (9 and 12) into the tree. These elements will follow the path of the right subtree of node 6 since 9 > 6 and 12 > 6.



Step 5: Insert the next elements (15) into the tree. It will follow the right subtree but the node at the right subtree is already full, so it splits into two new nodes at the same level at the medial element (4/2 = 2; the element at position 2 is 9) and the median is sent upward to the parent node. And then insert the element into its proper position by using BST property. Compare the next element (15) with root 9 since 15 > 9, so it will follow the right subtree.

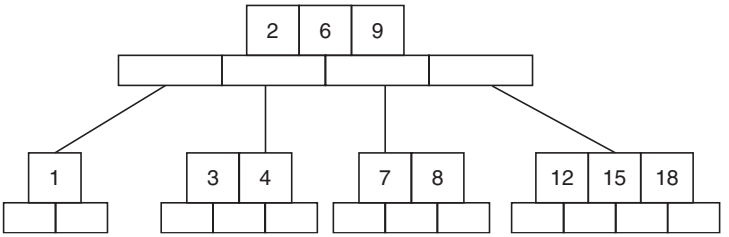


Step 6: Insert the next three elements (7, 18 and 3) into the tree.

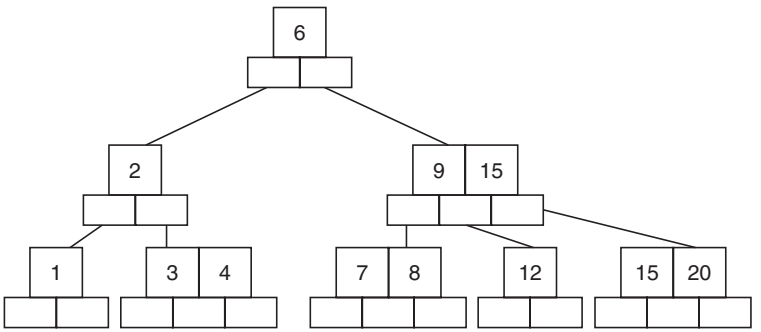


Step 7: Insert the next elements (4) into the tree. It will follow the leftmost subtree but the node at the leftmost subtree is already full, so it splits into two new nodes at the same level at the medial element (4/2 = 2; the element at position 2 is 2) and the median is sent upward

to the parent node. And then insert the element into its proper position by using BST property.



Step 8: Insert the next elements (20) into the tree. It will follow the rightmost subtree but the node at the rightmost subtree is already full, so it splits into two new nodes at the same level at the medial element (4/2 = 2; the element at position 2 is 15) and the median is sent upward to the parent node which is already full. So it also splits into two new nodes at the same level and the median (4/2 = 2; the element at position 2 is 6) is sent upward to create the new root node and then insert the element into its proper position by using BST property.



Deletion from the B-Tree

Let the tree be given to us as shown in Figure 15.

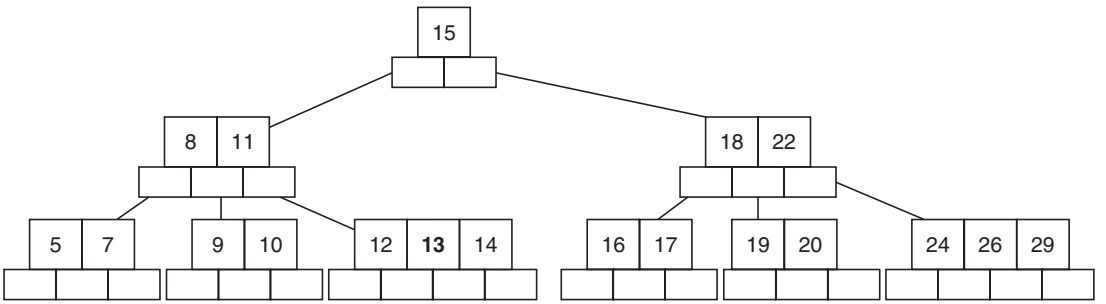
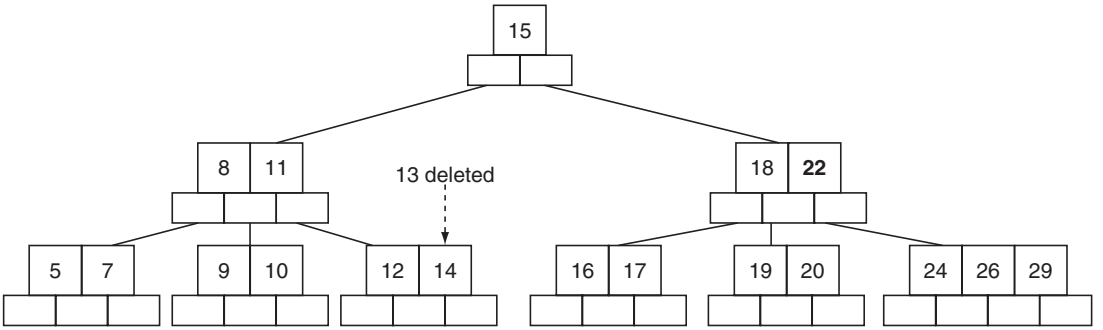


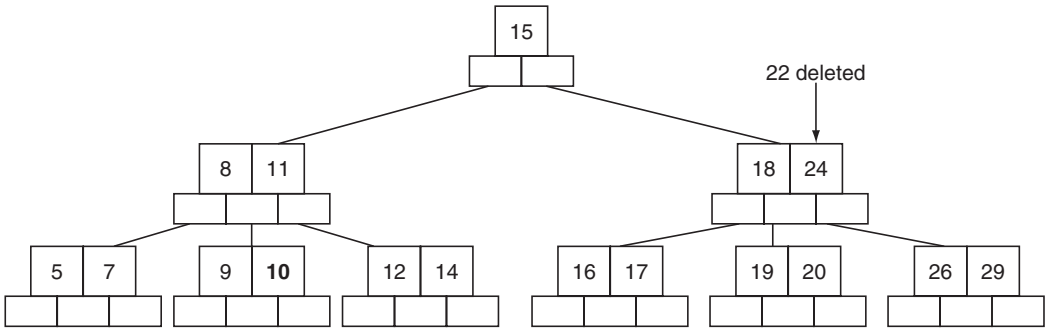
Figure 15 An example of B-tree.

The deletion from the B-tree can be performed as follows.

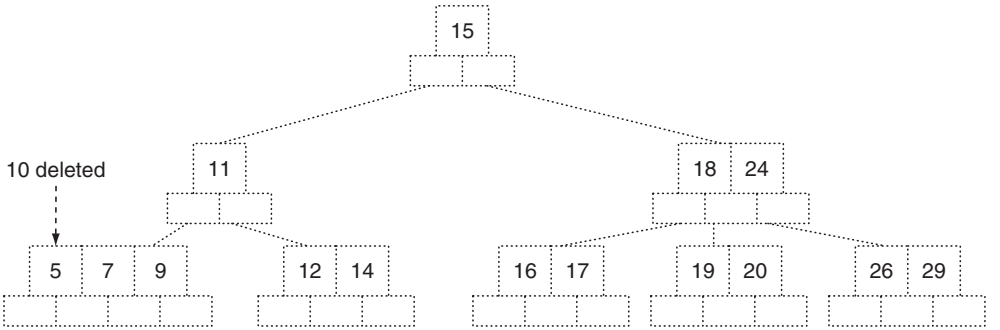
1. If the element is deleted from the leaf node, and its deletion does not make the node with less than the minimum number of elements then simply delete the node. For example, deletion of 13 (shown in bold face in Figure 15) from the tree is simple so the tree will now change as



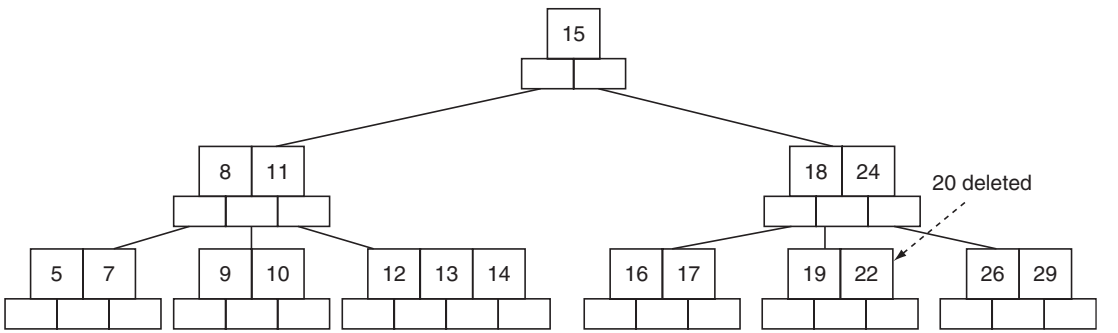
2. Now if the element is deleted from the internal node and its deletion does make the node with less than the minimum number of elements, then the element is replaced by the immediate successor of the element. For example, deletion of 22 (shown in boldface in the previous figure) from the tree then it will be replaced by the successor in the tree, that is, 24 whose deletion does not affect "the minimum number of elements a node can have" property so the tree will now change as



3. Now if the element is deleted from the leaf node and its deletion makes the node with less than the minimum number of elements, then the element is replaced by the immediate successor of the element. However, if it is not possible then the node is merged with the parent node and one of its siblings. For example, if there is deletion of 10 (shown in boldface in the previous figure) from the tree, then it cannot be replaced by the successor (11) because it will become the node with less elements than the minimum number of elements required. Hence, the node containing 10 and the parent node containing 8 along with left children containing 5 and 7 are grouped into one node with parent node 11 and right child as a node containing 12 and 14. The tree will now change as



4. Now if the element is deleted from the leaf node and its deletion does make the node with less than the minimum number of elements, then the element is replaced by the immediate successor of the element. For example, if there is deletion of 20 (from Figure 15) from the tree then it will be replaced by the successor in the tree, that is, 22 which in turn is replaced by its successor, that is, 24. So the tree will now change as



7.10 Advantages and Disadvantages of Tree Data Structures

The advantages and disadvantages of tree data structures are as follows.

Ad v antages

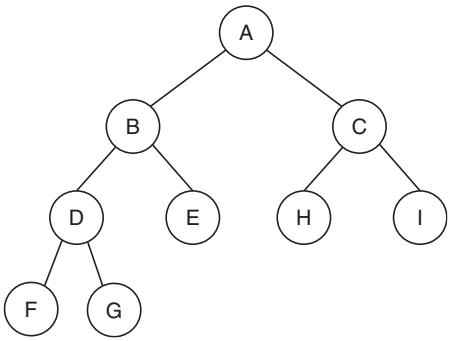
1. The nodes can be easily inserted and deleted.
2. The declaration of size is not required in advance and it can grow to any limit.
3. The in-order traversal of a tree always produces the sorted list so it is easy to keep the sorted data.
4. It can be represented in both ways, sequential and linked representations, so it has the advantages of both the data structures and linked lists and arrays:
5. It is used to represent the parent–child relationship.

Disad v antages

1. Every node of a tree requires more pointers so more memory space is required.
2. Playing with pointer is not an easy task, it requires extra care.
3. Tree can also be unbalanced which results in poor performance.

Solved Examples

Example 1 *Find the pre-order, in-order and post-order traversal of the following tree.*



Solution

Pre-order: A B D F G E C H I
In-order: F D G B E A H C I
Post-order: F G D E B H I C A

Example 2 *The different traversals of the tree is given here.*

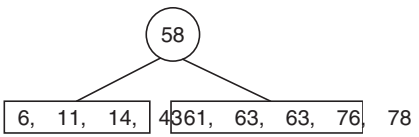
Pre-order:	58	14	6	11	43	76	63	61	63	78
In-order:	6	11	14	43	58	61	63	63	76	78

Construct the tree and find out the post-order traversal sequence.

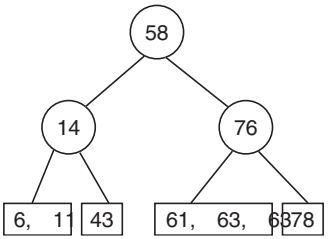
Solution

Step 1: The root of the tree is 58.

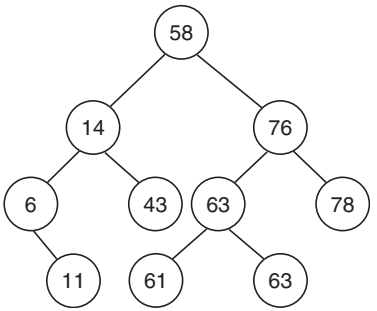
Step 2: The left subtree of 58 consists of 6, 11, 14 and 43, as we can see from the in-order traversal sequence. These elements are on the left of 58 in in-order traversal, similarly, 61, 63, 63, 76 and 78 will form the right subtree of 58.



Step 3: Now out of 6, 11, 14 and 43, the 14 appears first in pre-order traversal sequence. Therefore, 14 will be the root for left subtree. Similarly, out of 61, 63, 63, 76, 78, the 76 comes first in pre-order traversal sequence so it will be the root of the right subtree.



Step 4: Repeat Step 3 for the rest of elements in the left subtree and the right subtree.



The post-order traversal of the constructed tree is as follows.
Post-order: 11 6 43 14 61 63 63 78 76 58

Example 3 *Construct the B-tree of order 5 for the list of elements given as follows*

1, 7, 6, 2, 11, 4, 8, 13, 12, 5, 19, 9, 18, 24, 5, 12, 14, 20, 21

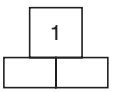
Solution

The B-tree of order 5 will have the following characteristics:

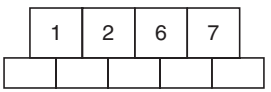
1. Each node can contain maximum 4 elements.
2. Each node can have maximum 5 children.
3. Each node except the root and leaf should contain at least 5/2 = 2 elements.

Now, let us construct the B-tree of order 5.

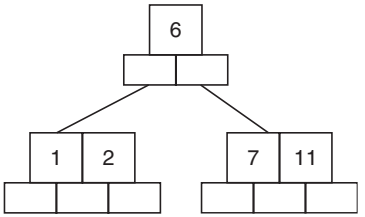
Step 1: Insert the first element (1) into the empty new node, root can have minimum two branches as shown below.



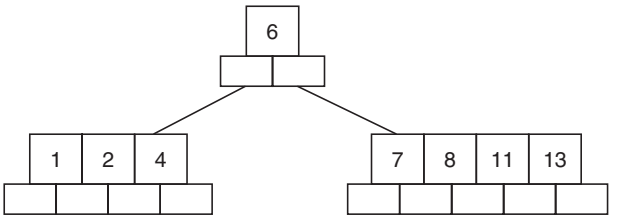
Step 2: Insert the next three elements (7, 6 and 2) into the root node. These elements can be inserted into the root node because each node can contain maximum 4 elements as shown below.



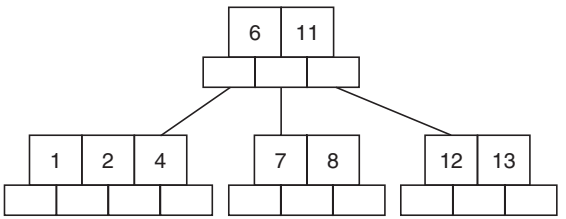
Step 3: Insert the next elements (11) into the root node. However, the root node is already full, so split the root node into two new nodes at the same level at the medial element (5/2 + 1 = 3; the element at position 3 is 6). And then insert the element into its proper position by using BST property. Compare the next element with root 6; since 11 > 6, so it will follow the right subtree.



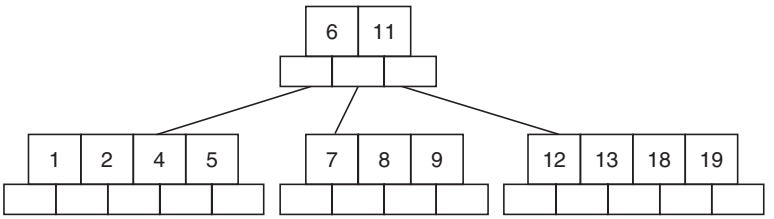
Step 4: Insert the next three elements (4, 8 and 13) into the tree. These elements will follow a different path.



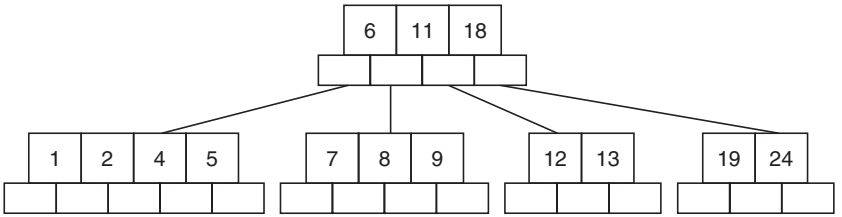
Step 5: Insert the next elements (12) into the tree; it follows the right subtree of the node 6. However, the node at the right subtree is already full, so split this node into two new nodes at the same level at the median element (5/2 + 1 = 3; the element at position 3 is 11) and then insert the element into its proper position by using BST property. Compare the next element with root 6, since 12 > 6 so it will follow the right subtree.



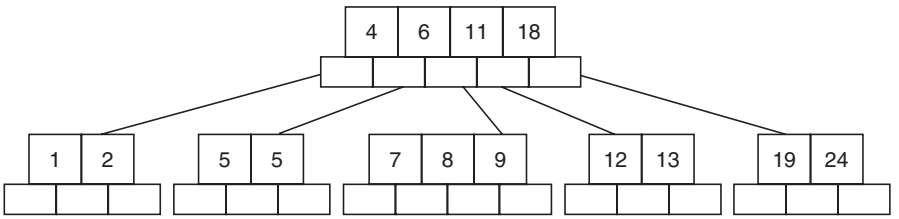
Step 6: Insert the next four elements (5, 19, 9 and 18) into the tree. These elements will follow the different path.



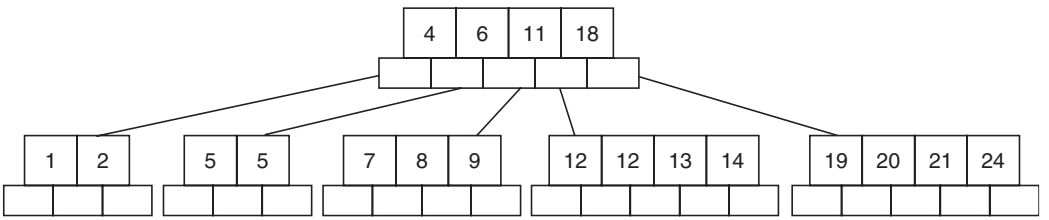
Step 7: Insert the next elements (24) into the tree; it follows the right subtree of the node 6. However, the node at the right subtree is already full, so split this node into two new nodes at the same level at the median element (5/2 + 1 = 3; the element at position 3 is 18) and then insert the element into its proper position by usin/g BST property.



Step 8: Insert the next elements (5) into the tree; it follows the left subtree of the node 6. However, the node at the left subtree is already full, so split this node into two new nodes at the same level at the median element (5/2 + 1 = 3; the element at position 3 is 4) and then insert the element into its proper position by using BST property.



Step 9: Insert the next four elements (12, 14, 20 and 21) into the tree. These elements will follow the different path.



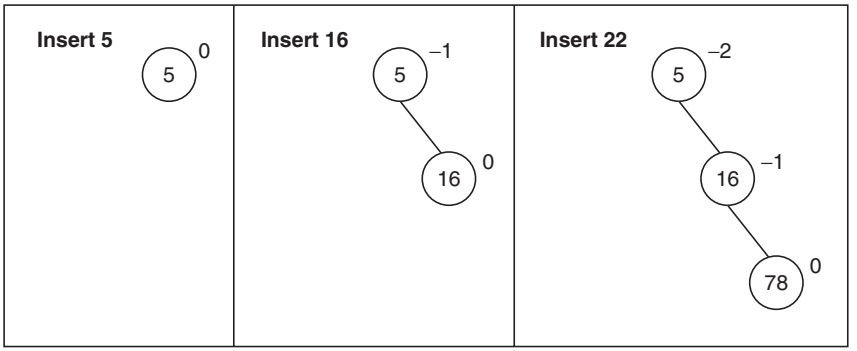
Example 5 *The following nodes are inserted in to the empty tree in following order*

5, 16, 22, 45, 2, 10, 18, 30, 50

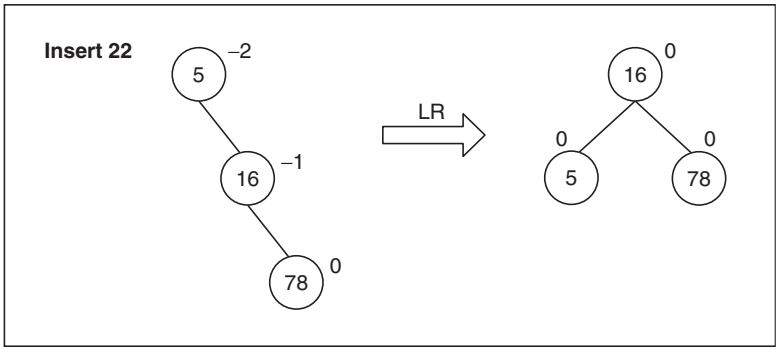
Construct the AVL tree.

Solution

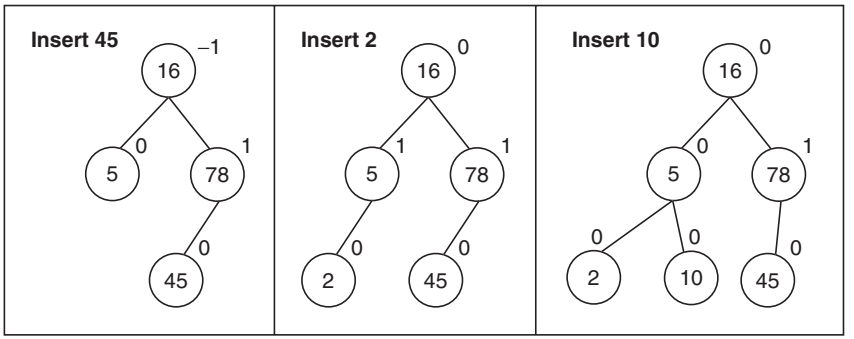
To construct the AVL tree, we have to deal with the balance factor and re-balancing of the tree wherever required.



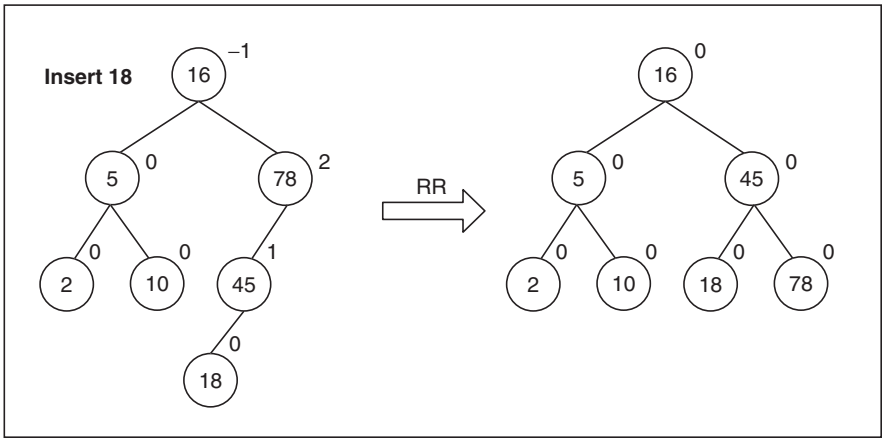
Now, the tree in imbalanced so rotation is required since the balance factor is – 2; and all the three nodes RN, IN and UB are in a straight line so left rotation is needed.



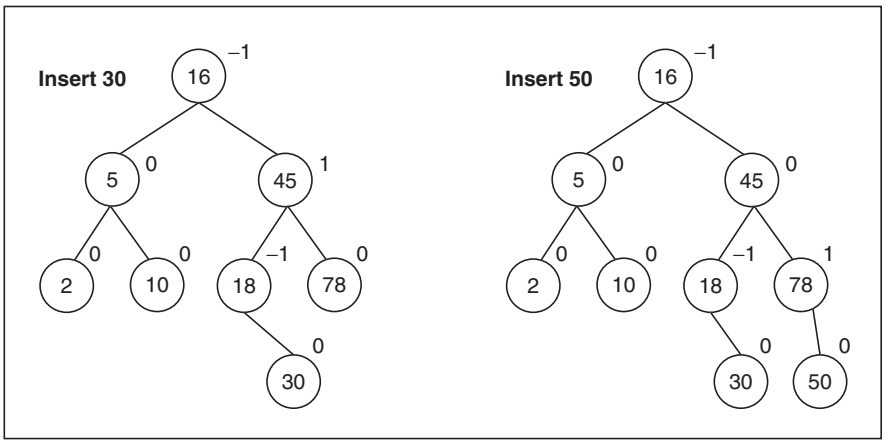
Now insertion of the next elements (45, 2, 10) in the list will not disturb the balancing of the tree so they are inserted as shown below.



Now, the insertion of 18 makes the tree imbalanced so rotation is required since the balance factor is 2; and all the three nodes RN, IN and UB are in a straight line so right rotation is needed.



Insertion of the next two elements 30 and 50 do not disturb the balance factor of the tree as shown below.



Summary

1. The number of nodes connected to any node is called the degree of that node, or we can say that the number of children that any node has is the degree of that node.
2. The length of the longest path from the root node to any terminal node is called the depth of the tree.
3. The maximum degree of any node is called the degree of a tree.
4. In the binary tree each node can have either 0 children, 1 child or 2 children. So the binary tree is a tree which is either empty or each node can have maximum two children.
5. In in-order traversal, the root node is visited in between the nodes in left and right sub-trees.
6. In pre-order traversal, the root node is visited prior to the left or right nodes in a subtree.

7. In post-order traversal, the root node is visited after the nodes in the left and right subtrees.
8. The pre-order traversal is also said to be depth first traversal.
9. The post-order traversal is said to be breadth first traversal.
10. A binary search tree is a binary tree that is either empty or in which every node contains a key; and the left subtree of this key node contains the smaller values and the right subtree contains equal or greater values than the key.
11. If all the terminal nodes of a tree are at level d, where d is the depth of the tree, then such trees are called strictly complete binary trees.
12. The maximum possible nodes at any level are 2^n where n is the level number.
13. If all the terminal nodes of a tree are at level d or $d - 1$ where d is the depth of the tree, then such trees are called almost complete binary tree.
14. All the nodes at the last level are filled from left to right direction in almost complete binary tree.
15. When a binary search tree is visited according to the in-order traversal rule, the node contents are printed in ascending order.
16. Basic operations on a binary search tree take time proportional to the height of the tree.
17. Binary search tree takes time proportional O(log n) in the best case – if tree is completely balanced.
18. AVL tree is named after Adelson, Velskii and Landis.
19. Balanced trees can be used to search, insert and delete arbitrary keys in O(log n) time.
20. The average-case and the worst-case complexities of the balanced tree are about O(log n) due to its balanced structure.

Key Terms

M-way search tree
Binary search tree
B-tree
Tree
Post-order traversal
In-order traversal
Pre-order traversal
Left subtree
Right subtree
Left child
Rights child
Complete binary tree
Almost complete binary tree
General tree
Expression tree

Multiple-Choice Questions

1. Which of the following has the in-degree 0?
   a. Root
   b. Leaf node
   c. Internal node
   d. All of the above
2. Which of the following has the out-degree 0?
   a. Root
   b. Leaf node
   c. Internal node
   d. None of the above
3. Which of the following sequence represents the depth first traversal of a binary tree?
   a. NLR
   b. NRL
   c. RLN
   d. LNR

4. Which of the following sequence represents the breadth first traversal?
   a. NLR
   b. LNR
   c. RLN
   d. LRN

5. The binary tree with n nodes can have maximum
   a. $n - 1$ branches.
   b. n branches.
   c. $n/2$ branches.
   d. $\log_2 n$ branches.

6. The nodes with zero descendents are called
   a. root nodes.
   b. internal nodes.
   c. leaf nodes.
   d. all of the above.

7. Which of the following statements is incorrect?
   a. The first node of the tree with in-degree zero is called the root of the tree.
   b. The number of edges that ends at particular node is called the in-degree of the node.
   c. The length of the longest path from the root node to any terminal node is called the depth of the tree.
   d. None of the above.

8. Which of the following statements is correct?
   a. The last node of the tree which does not have any descendents or the node with zero out-degree is called the leaf node or terminal node.
   b. The number of edges terminates at a particular node is called the out-degree of that node.
   c. The sum of in-degree and out-degree branches is the degree of the node.
   d. The degree of any node is called the degree of a tree.

9. A binary tree can have maximum
   a. zero children.
   b. one children.
   c. two children.
   d. zero, one or two children.

10. In the sequential representation of binary tree, which of the following is true?
    a. The left child of any node say k is placed at [2*k].
    b. The right child is placed at [2*k + 1].
    c. The size of the array depends on the depth d of the tree and it is 2^{d+1}.
    d. All of the above.

11. A tree is said to be a complete binary tree if
    a. all the terminal nodes of a tree are at level d where d is the depth of the tree.
    b. there will be only one node (Root) in the complete binary tree of depth zero.
    c. all the terminal nodes of a tree are at level d or $d - 1$ where d is the depth of the tree.
    d. all the nodes at the last level are filled from left to right direction. If the right child of any node is at level d, then its left child must also be at level d.

12. Which of the following sequence represents the post-order traversal?
    a. LRN
    b. LNR
    c. RLN
    d. LRN

13. Which of the following sequence represents the pre-order traversal?
    a. NLR
    b. NRL
    c. RLN
    d. LNR

14. The level of the root node of a binary tree is always
    a. 0.
    b. 1.
    c. 2.
    d. N.

15. Which of the following sequence represents the in-order traversal?
    a. NLR
    b. LNR
    c. RLN
    d. RNL

16. A binary search tree is a special kind of binary tree that satisfies which of the following conditions?
    a. The data elements of the left subtree are smaller than the root of the tree.
    b. The data elements of the right subtree are greater than or equal to the root of the tree.
    c. The left subtree and right subtree are also the binary search tree, that is, they must also follow the above two rules.
    d. All of the above.

17. Which of the following statement is not correct for a threaded binary tree?
    a. It has a special links that simplifies the traversal on binary trees.
    b. In which the null entries of the left and right subtrees are replaced by a special pointer called thread.
    c. A binary tree is threaded according to the particular traversal order.
    d. None of the above.

18. In one-way threading
    a. the null entry of the right field of the node is replaced by the successor of the node.
    b. the null entry of the right field of the node is replaced by the successor of the node, and null entry of the left field of the node is replaced by the predecessor of the node.
    c. the null entry of the left field of the node is replaced by the successor of the node.
    d. the null entry of the left field of the node is replaced by the predecessor of the node.

19. In two-way threading
    a. the null entry of the right field of the node is replaced by the successor of the node.
    b. the null entry of the right field of the node is replaced by the successor of the node, and null entry of the left field of the node is replaced by the predecessor of the node.
    c. the null entry of the right field of the node is replaced by the predecessor of the node, and null entry of the left field of the node is replaced by the successor of the node.
    d. The null entry of the left and right fields of the node is replaced by the successor of the node.

20. If the node having two children is deleted from the BST then we replace the node (to be deleted) by
    a. the pre-order successor of the node.
    b. the post-order successor of the node.
    c. the in-order successor of that node.
    d. It is simply deleted.

21. When the balance factor is – 2 then which of the rotation is applied?
    a. Left rotation
    b. Right rotation
    c. Left–left rotation
    d. Right–right rotation

22. When the balance factor is 2 then which of the rotation is applied?
    a. Left rotation
    b. Right rotation
    c. Left–left rotation
    d. Right–right rotation

23. If the balance factor is – 2 and its immediate successor (child) has the balance factor +1, then which of the following rotation is applied?
    a. Left rotation
    b. Right rotation
    c. Left–right rotation
    d. Right–left rotation

24. A tree node that has no children is called a
    a. leaf node.
    b. root node.

c. internal node.
d. loop node.

25. The in-order traversal of which of the following tree produces the elements is ascending order?
a. AVL
b. B-tree
c. BST
d. General tree

26. If the balance factor is 2 and its immediate successor (child) has the balance factor – 1, then which of the following rotation is applied?
a. Left rotation
b. Right rotation
c. Left–right rotation
d. Right–left rotation

27. If the RN, IN, and UB are in a single straight line and they are at the right subtree of the root (UB), then which of the rotation is performed?
a. Left rotation
b. Right rotation
c. Left–right rotation
d. Right–left rotation

28. Double rotation is performed in the AVL tree when
a. RN, IN and UB are in a single straight line and they are at the left subtree of the root (UB).
b. RN, IN and UB are in a single straight line and they are at the right subtree of the root (UB).
c. they are not in the straight line or they form the dog legs like structure or bend.
d. none of the above.

29. Which of the following statements is correct?
a. The balancing of the tree does not change the sequence of elements in the in-order traversal.
b. The unbalanced tree has the worst-case complexity O(n) for searching an element in the tree.
c. The balanced tree takes only O(log n) time complexity in the worst case.
d. All of the above.

30. If the complete binary tree contains 5 nodes at level 3, then it will have how many nodes at the next level 4?
a. 10
b. 9
c. 33
d. 8

31. Which of the following properties is satisfied by the B-tree of order M?
a. Each node can contain maximum M – 1 key elements.
b. Each node can have maximum M branches.
c. It satisfies the property of BST.
d. All of the above.

32. A B-tree is a
a. multiway search tree.
b. search tree in which the elements with smaller value than the key are positioned into the left subtree and the elements with greater or equal values are positioned in to the right subtree of the key.
c. all leaf nodes are on the same level so tree is balanced.
d. all of the above.

33. If the in-order traversal of some binary tree produced the sequence PMNXY Z and the post-order traversal of the same tree produced the sequence PNMY Z X. Which of the following is the correct pre-order traversal sequence?
a. XMPNZ Y
b. PNMY Z X
c. XNMY Z P
d. XY Z NPM

34. If the in-order traversal of some binary tree produced the sequence PMNXY Z and the post-order traversal of the same tree produced the sequence PNMY Z X. What will

be the total number of nodes in the left subtree?
a. 2
b. 3
c. 4
d. 5

35. If post-order traversal of some tree produced the sequence PNMY Z X, then which element will be the root node of the tree?
a. P
b. Y
c. X
d. Cannot say

36. Which of the following statement is correct for binary tree?
a. A binary tree of height h has at least h and at most $2^h - 1$ elements in it.
b. A binary tree with n elements has exactly $n - 1$ edges.
c. The number of elements at level k is at most $2^k - 1$.
d. All of the above.

37. A complete binary tree with height h can contain exactly
a. $2^h - 1$.
b. $2^h - 1$.
c. 2^h
d. $2^h - 1$.

38. Which of the following statement is correct?
a. BST holds the transitive property, that is, if a.key < b and b.key < c.key then a.key < c.key
b. Trees of height O(log n) are said to be balanced.
c. Balanced trees can be used to search, insert and delete arbitrary keys in O(log n) time, for example, B+ tree.
d. All of the above.

Review Questions

1. Construct the binary search tree for the following elements:

   12, 45, 5, 38, 19, 29, 12, 9, 9, 18, 78, 27, 18, 2, 20

2. Construct the B-tree of order 4 for the following list of elements:

   12, 45, 5, 38, 19, 29, 12, 9, 9, 18, 78, 27, 18, 2, 20, 45, 6, 47, 8

3. Construct the binary search tree for the following elements:

   D A T A S T R U C T U R E A N D A L G O R I T H M S

4. Construct the AVL for the following elements:

   D A T A S T R U C T U R E A N D A L G O R I T H M S

5. Construct the B-tree of order 2 for the following elements:

   D A T A S T R U C T U R E I N C P L U S P L U S

6. What are the different ways of tree traversal? Explain each one in detail.

7. What is the importance of threaded binary tree?

8. What is the threaded binary tree and what are the different types of threaded binary tree?

9. Draw expression for each of the following expressions, and show the sequence of traversal using different traversal techniques
   (a) $(a < b)$ && $b(< c) \mid\mid c < (a)$
   (b) $a - b/c*d\hat{}e + f$
   (c) $((a + b)*c)/d - e$

10. Explain the following with examples:
    (a) Binary tree;
    (b) complete binary tree;
    (c) almost complete binary tree.

11. How does the tree data structure differ from the other data structures?

12. What are the different techniques for tree traversal? Write the recursive algorithms for each traversal technique.

13. The following nodes are inserted into the empty tree in the following order

    5, 12, 62, 25, 29, 30, 18, 34, 42, 12, 19

    Construct the following:
    (a) Binary search tree;
    (b) AVL tree;
    (c) Heap tree;
    (d) B–tree.

14. A binary tree *T* has 9 nodes. The in-order and pre-order traversal of *T* yields the following sequence of nodes:

    In-order: 5 1 3 9 6 8 4 2 7
    Pre-order: 6 1 5 9 3 4 8 7 2

    Draw the tree.

15. Insert the following key into B-tree of order 5 in step-by-step manner

    1, 7, 6, 2, 11, 4, 8, 13, 10, 5, 19, 9, 18, 24, 3, 12, 14, 20, 21, 16

16. What is the difference between binary tree and binary search tree? Write down the algorithm to delete an element with two children from the BST

17. Write short notes on the threaded binary tree.

18. What is the importance of AVL tree? How are the rotations performed on the AVL tree?

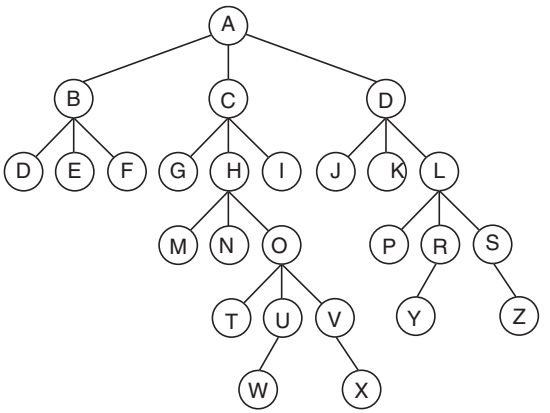
19. What are the properties of a B-tree? Construct the B-tree of order 5 with the following elements

    20, 90, 11, 33, 78, 95, 12, 67, 23, 15, 76, 24, 54, 38, 97, 29, 15, 27, 28, 47

20. What are the different representations of tree in the memory? Explain the sequential representation of a tree.

21. What is tree traversal? Write recursive algorithms for various types of traversal.

22. Differentiate between a complete binary tree and an almost complete binary tree. Construct the binary tree for the following general tree.



23. What do you understand by the height of a tree?

24. What do you understand by the degree of a tree?

25. How tree is represented using an array?

26. What are the differences between array representation and dynamic representation?

Programming Assignments

1. Write a program in C/C++ for in-order traversal of binary tree using non-recursive function.
2. Write a program in C/C++ for pre-order traversal of binary tree using recursive function.
3. Write a menu-driven program in C/C++ for different traversal techniques of the binary tree.
4. Write a program in C/C++ to insert an element in BST.
5. Write a program in C/C++ to insert an element in AVL tree.
6. Write a menu-driven program in C/C++ to delete an element from the BST considering all the cases.
7. Write a menu-driven program in C/C++ to delete an element from the B-tree considering all the cases.
8. Write a program in C/C++ to convert the general tree into binary tree.
9. Write a program in C/C++ to construct the expression tree of the given expression.
10. Write a menu-driven program in C/C++ to insert an element in the AVL tree and balancing the tree.
11. Write a program in C/C++ to count the internal nodes in the binary tree.
12. Write a program in C/C to print the width of the tree, that is, the maximum number of nodes on the same level.
13. Write a program in C to perform various operation on BST.
14. Write a program in C to construct the AVL tree.

Answers

Multiple-Choice Questions

1. (a)
2. (b)
3. (a)
4. (d)
5. (a)
6. (c)
7. (d)
8. (c)
9. (c)
10. (d)
11. (a)
12. (d)
13. (a)
14. (a)
15. (b)
16. (d)
17. (d)
18. (a)
19. (b)
20. (c)
21. (a)
22. (b)
23. (c)
24. (a)
25. (c)
26. (d)
27. (a)
28. (c)
29. (d)
30. (a)
31. (d)
32. (d)
33. (a)
34. (b)
35. (c)
36. (d)
37. (a)
38. (d)

8 Graph

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Concept of another non-linear data structure called graph.
- Graph terminology.
- Graph traversal.
- Shortest path.
- Minimum spanning tree.
- Shortest path algorithms.

We have already studied one non-linear data structure called "tree" in which one node has many successors. Now we will study another non-linear data structure called "graph" where each node has many successors and predecessors. We will discuss the representation of graph as a data structure in memory and the different ways of traversing a graph: Breadth first search (BFS) and depth first search (DFS). Graph is a very useful data structure used for solving different routing algorithms in computer networks, circuit problems in Electrical and Electronics Engineering. Graph theory finds application in many subjects, such as Physics, Chemistry, Mechanical Engineering, Electrical Engineering and other branches of Science and Engineering. We will also discuss the shortest path and minimum spanning tree (MST) problems.

8.1 Graph Terminologies

This section defines the graph and summarizes some of the main terminologies associated with the theory of graphs.

Defi nition

Graph is a collection of two finite sets (V and E) where V contains the finite number of elements called vertices and E contains the finite number of elements called the edges. An edge is defined as a pair of vertices such that they are connected by a line segment. The graph shown in Figure 1 can be defined as $G = \{V, E\}$, where $V = \{A, B, C, D, E, F\}$ and $E = \{(A, B), (A, C), (B, C), (B, D), (C, D), (C, E), (D, E), (D, F), (E, F)\}$. The pair (A, B) defines an edge between A and B and similarly other pairs also define the edge; therefore there are nine edges as we can see from the graph shown in Figure 1 also. The number of vertices of the graph constitutes the order of the graph. There are six vertices in the graph shown in Figure 1 so the order of the graph is six. Therefore a graph consists of two things:

1. A set V of vertices.
2. A set E of edges defined by an unordered and unique pair of vertices.

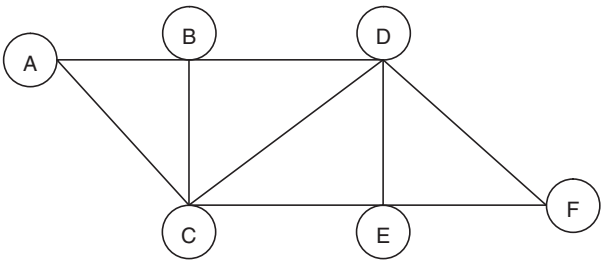


Figure 1 Example of graph.

Terminol ogies Associated with G raph

Now let us take a look on the various terminologies associated with the graph.

1. **Adjacent vertices:** Two vertices are said to be adjacent if they are connected by an edge. For example, *B* and *C* are adjacent to vertex *A* because separate edges connect *B* and *C* to *A*.
2. **Adjacent edges:** Two edges are said to be adjacent if they are incident on the common vertex. For example, edges *CD* and *CE* are adjacent because they are incident on the common vertex *C*.
3. **Path:** A path is a sequence of distinct vertices in which each vertex is adjacent to the next one. For example, there is a path *ABCDF* in Figure 1 from *A* to *F*. The path is said to be simple if all the nodes are distinct.
4. **Cycle:** A cycle is a path consisting of at least three vertices where last vertex is adjacent to the first vertex. For example, the cycle *CDFEC* in Figure 1.
5. **Loop:** It is a special cycle that starts and ends with the same vertex without visiting any other vertex. For example, if the edge starts and ends at vertex C without visiting any other vertex then this is called a loop.
6. **Degree of a vertex:** The degree of a vertex is the number of lines incident on it. The loop contributes two to the degree of the vertex. For example, the degree of *D* is four and degree of *C* will be six if there is a loop on C. In a graph, the number of vertices of odd degree is always even and the degree of a graph is twice the number of edges in a graph.
7. **Connected vertices:** Two vertices are said to be connected if there is a path between them. For example, *A* and *F* are connected in Figure 1 because there is a path between *A* and *F*.
8. **Out-degree:** The out-degree of a directed graph (defined in the next section) is the number of arcs leaving the vertex. The out-degree of vertex *u* is represented by OutDeg(*u*). In other words we can say that OutDeg(*u*) is the number of edges starts at *u*. *u* is called the source if it has positive OutDeg but zero in-degree.
9. **In-degree:** The in-degree of a directed graph is the number of arcs entering the vertex. The in-degree of vertex *u* is represented by InDeg(*u*). In other words, we can say that InDeg(*u*) is the number of edges terminates at *u*. *U* is called the sink if it has positive InDeg but zero out-degree.
10. **Multiple edges:** The distinct parallel edges that connect the same end points.

8.2 Types of Graph

This section discusses the various types of graphs:

1. **Simple graph:** A graph that does not contain any loop or multiple edges is called a simple graph. For example, the graph shown in Figure 2 is a simple graph.

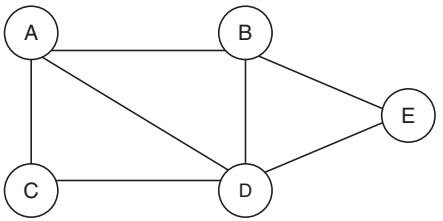


Figure 2 A simple graph.

2. **Multigraph:** A graph that contains more than one edge between two vertices is called a multigraph. For example, the graph shown in Figure 3 is a multigraph where there are two edges between vertex *A* and *D*.

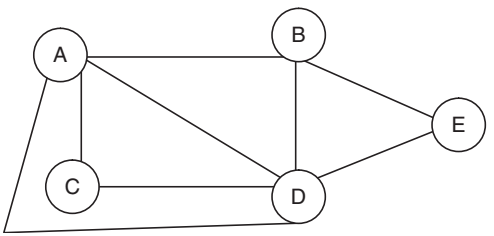


Figure 3 A multigraph.

3. **Directed graph:** A graph in which the set E of edges is a collection of the ordered pair of vertices is called a directed graph. In a directed graph, each edge is marked in a direction towards the second vertex (i.e., the successor vertex) from the first vertex of the ordered pair. One sample directed graph is shown in Figure 4. If "*e*" is an edge defined by an ordered pair (*u*, *v*) then *e* starts at *u* and ends at *v*. So in directed graph we can move only in one direction of the edge. For example, we can only move from *A* to *B* but we cannot move from *B* to *A* because the edge has a direction from *A* to *B* only. A directed graph is often called a diagraph. We can see that the out-degree of a vertex *A* is 1 because there is only one edge leaving the vertex *A*, and in-degree of *A* is also 1 because there is only one edge that ends at *A*. The in-degree of vertex *D* is two but the out-degree is 1. A directed graph *G* is said to be simple if *G* has no parallel edges.

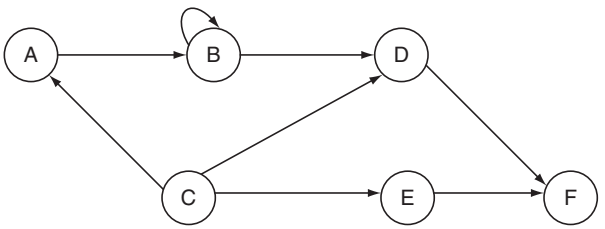


Figure 4 A directed graph.

4. **Undirected graph:** A graph in which the set *E* is a collection of unordered pair of vertices is called undirected graph. So in an undirected graph (as is shown in Figure 5) we can move in either of the direction between two vertices along the edge.

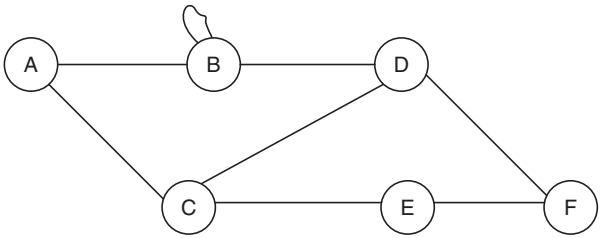


Figure 5 An undirected graph.

5. **Weighted graph:** A graph in which the weights (non-negative cost) are assigned to each edge is called a weighted graph. For example, the graph shown in Figure 6 is a weighted graph.

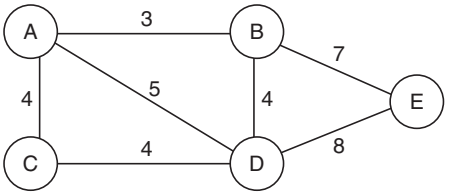


Figure 6 A weighted graph.

6. **Regular graph:** A graph in which each vertex has the same degree "k" is called k-regular graph. One such 2-regular graph is shown in Figure 7 where each vertex has the degree two.

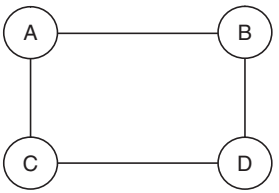


Figure 7 A regular graph.

7. **Complete graph:** A graph is said to be connected if there is a path between any two vertices; if every vertex is connected to every other vertex in the graph then the graph is said to be complete graph. The complete graph consists of $n(n-1)/2$ edges where n is the number of vertices. Figure 8 shows the complete graph.

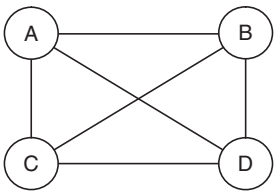


Figure 8 A complete graph.

8. **Strongly connected graph:** This is basically used with the directed graph where each vertex is connected to every other vertex with the direction. Figure 9 shows a strongly connected graph.

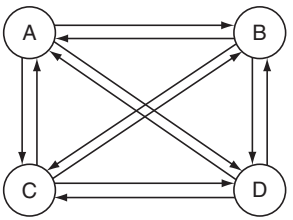


Figure 9 A strongly connected graph.

9. **Weakly connected graph:** A directed graph is weakly connected if at least two vertices are not connected (Figure 10).

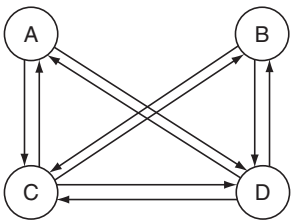


Figure 10 A weakly connected graph.

10. **Isomorphic Graph:** Two graphs in which there is a one-to-one correspondence between their vertices and between their edges is said to be isomorphic graph. Figures 11(a) and (b) show the isomorphic graph.

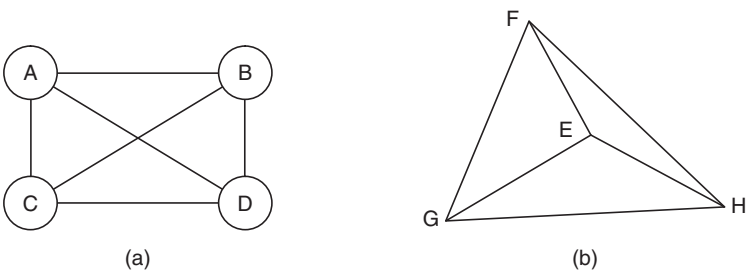


Figure 11 Isomorphic graph.

8.3 Representation of Graph

There are various possible representations of graphs depending on the application of the graph. We will discuss two standard and popular methods of maintaining the graph in the computer's memory.

1. Sequential representation using adjacency matrices.
2. Linked representation using adjacency list or linked list of neighbors.

The graph *G* is normally input into the computer by using its formal definition: A collection of nodes and a collection of edges.

Sequential Representation using Adjacency Matrix

The graph with N vertices can be represented by an $N \times N$ matrix. This matrix is called the adjacency matrix because it holds the value 1 (true) for adjacent vertices and 0 (false) for non-adjacent vertices. The space needed to represent a graph using its adjacency matrix is N^2 bits. The adjacency matrix M contains N rows and N columns with the following condition:

M_{ij} = 1 if there is an edge between vertex i and vertex j
= 0 otherwise (if there is no edge between i and j)

The rows and columns are named with the vertices.

Example 1

Let us take the example of both undirected graph and directed graph as shown in Figures 12(a) and (b).

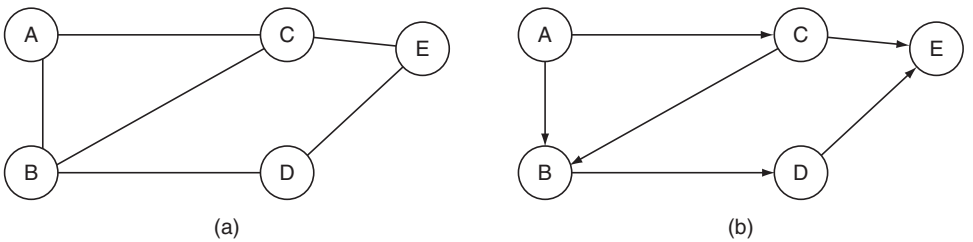


Figure 12 (a) Undirected graph; (b) directed graph.

The adjacency matrix corresponding to these graphs are shown in Figure 13.

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	0	1
D	0	1	0	0	1
E	0	0	1	1	0

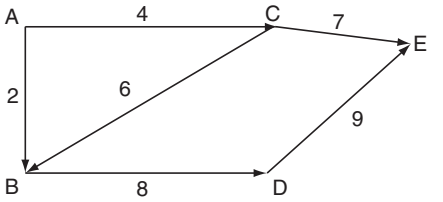
(a)

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	0
C	0	1	0	0	1
D	0	0	0	0	1
E	0	0	0	0	0

(b)

Figure 13 (a) Adjacency matrix for Figure 12(a); (b) adjacency matrix for Figure 12(b).

Here we can see that the adjacency matrix [Figure 13(a)] for undirected graph [Figure 12(a)] contains value 1 for both edges *AB* and *BA*. However, the adjacency matrix [Figure 13(b)] for directed graph [Figure 12(b)] contains 1 for edge *AB* because there is a directed edge from *A* to *B*. Since there is no directed edge from *B* to *A* so *BA* entry contains 0. A weighted graph can also be conveniently represented by adjacency matrices. The matrix for weighted matrix is formed by placing the cost or weight in the *i*th row and *j*th column of the matrix if there is an edge between vertex *i* and vertex *j* of the graph. The adjacency matrix corresponding to the graph shown in Figure 14(a) is given in Figure 14(b).



	A	B	C	D	E
A	0	2	4	0	0
B	0	0	0	8	0
C	0	6	0	0	7
D	0	0	0	0	9
E	0	0	0	0	0

(b)

Figure 14 (a) Weighted graph; (b) weight matrix for part (a).

The adjacency matrix is very useful in showing the number of paths between two vertices. The *ij*th entry of the square of adjacency matrix shows all the possible paths of length two between two vertices. The *ij*th entry of the adjacency matrix $(\text{Adj})^N$ gives the number of paths of length N between two vertices. However, the adjacency matrix representation of a graph is not suitable for insertion and/or deletion of vertices in the graph because it requires a lot of shifting operations in the adjacency matrix. The adjacency matrix is a bit matrix or the Boolean matrix because its entries are 1 and 0.

Linked List Representation

The adjacency matrix representation is also called as a static or sequential representation of a graph so it has the same disadvantages as with the sequential representation. In this representation, it is difficult to store additional information about vertices and edges in the graph. The storage space for adjacency matrix is required in advance and so it is very difficult to insert and/or delete the elements from the graph. The insertion and/or deletion of elements from the graph require many shifting operations, which is not an easy task because the dimension of the adjacency matrix is changed. Another disadvantage is that huge memory amount is wasted in storing all the information (like the storage for every edge even if edge does not exist) of the graph in adjacency matrix.

The obvious solution for these problems is the dynamic or the linked representation. In the linked representation, the graph is stored as a linked structure where each vertex is represented as a node that is divided into three parts: First contains the vertex name, second contains the link to the next vertex and third contains the link to another list that contains all the adjacent vertices. A linked representation is also called an adjacency list. This method represents vertex as the nodes and the linked list representation of the edges. Though it is generally used for the directed graph, but it may be used for undirected graph also by storing each edge twice in the list. Weighted graph can also be represented using this method by storing the corresponding weights along with the terminal vertex of the edge. Consider the graph shown in Figure 15. It has to be represented using the linked list representation.

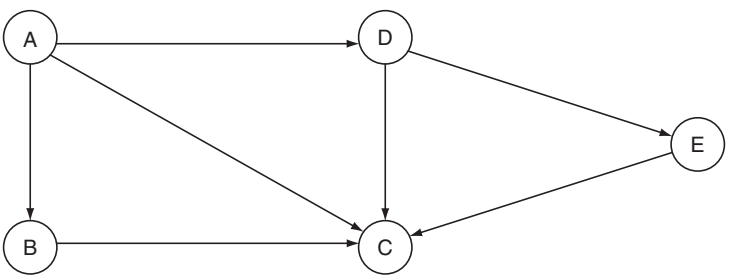


Figure 15 Directed graph for linked list representation.

Here we have five nodes so first list of nodes will contain five nodes each of the form

Name	Link to next node	Link to another list of adjacent node

This list is called the node list. Each node of the list of adjacent node will be of the form

Adjacent node	Link

This list is called the edge list. Now the linked representation of the given graph is shown in Figure 16.

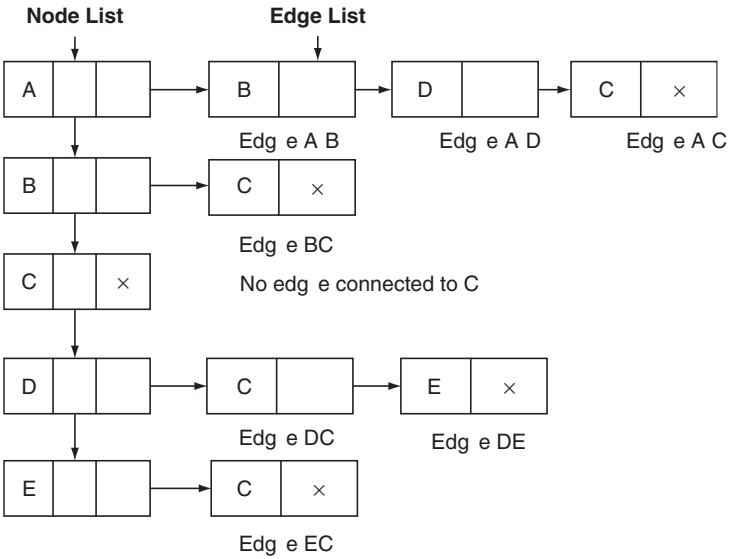


Figure 16 Linked list representation of a directed graph shown in Figure 15.

Therefore, we can say that the adjacency list contains two lists: Node list and edge list. The node list can be implemented using singly, doubly or the circular linked list. Sometimes depending on the application, the node may be organized as a sorted array or a binary search tree instead of a linked list. We can also keep the additional important information like weight, degree, etc. with each node. The nodes will have a start pointer variable for the beginning of the list. The adjacency list is very useful when the graph does not contain many edges. That is, the graph is a sparse graph. If the sparse graph is represented using the adjacency matrix, then it will contain many zero entries in the matrix where the adjacency list only maintains the existing edges. Similarly, when the graph contains many edges then the searching of a particular edge becomes easier in the adjacency list because we have to search the neighbors only. Whereas in the adjacency matrix, we have to search the whole matrix element by element until the desired element is not found.

8.4 Traversal of Graph

Some applications of the graph require systematically visiting all the nodes. In a graph, we can reach to the vertex from multiple paths existing in the graph, so we can say that graph traversal is not as simple as traversal in other data structures. There are two standard traversal methods generally used with graph:

1. Breadth first search (BFS).
2. Depth first search (DFS).

Bread th F irst Search

The general idea behind a BFS is that it starts traversing the graph from the starting node (any selected node) and then traverses all the nodes which are directly connected to the starting node in a particular order. In a graph traversal we have to keep track of the nodes that have been traversed, therefore queue is an obvious choice to keep track of such nodes.

First, the starting node is examined and all the adjacent nodes are placed into the queue. The node traversed is marked as visited. The next node is taken from the front of the queue and examined; this node is also marked as visited and all the neighbors of this node are placed into the rear of the queue. We already know that the elements are always inserted at the rear end of the queue and taken away from the front end of the queue. The order of traversal of nodes depends on the sequence in which the nodes were placed (inserted) into the queue. Therefore in BFS all the nodes at a particular level are processed first, only then can we move to the next level. The visited nodes are not traversed again and again if they come into the way and so no node is processed more than once.

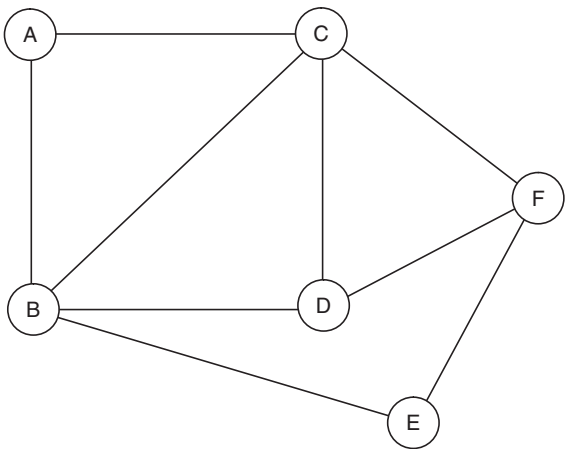
The BFS algorithm has many applications in artificial intelligence. The advantage of the BFS algorithms is that if a solution to a particular problem exists then BFS guarantees to find it because it explores all the nodes. However, it requires more memory and time for all the nodes to explore. The algorithm for BFS is summarized in Algorithm 1.

Algorithm 1 BFS

1. Start traversing the given graph with the given number of vertices and edges.
2. Insert the starting source vertex *S* to the queue and mark it as visited.
3. Repeat
   a. Remove the front node *K* of the queue and examine it.
   b. Insert all the adjacent nodes of *K* (which are not visited so far) to the queue and mark them as visited.
4. Until queue is empty.
5. Exit.

Problem 1

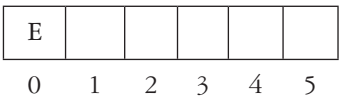
Traverse the following graph using the BFS starting from E.



Solution

Step 1: Start traversing the graph given.

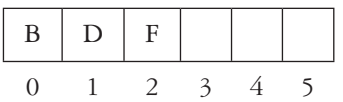
Step 2: Insert the starting source vertex *E* to the queue and mark it as *visited*.



$F = 0, R = 0$

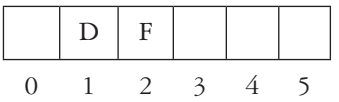
Step 3: Take the front node *E*. Examine it.

Step 4: Insert all the adjacent nodes of *E* to queue named as *Q* and mark them as *visited.*



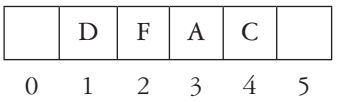
$F = 0, R = 2$

Step 5: Take the front node *B*. Examine it.



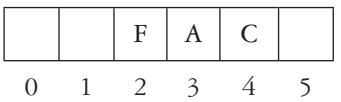
$F = 1, R = 2$
Examine *B*

Step 6: Insert all the adjacent nodes of *B* to *Q* and mark them as *visited.*



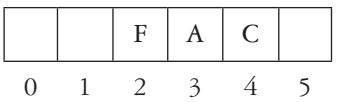
$F = 1, R = 4$, *D* is not included because it is already in *Q*

Step 7: Take the front node *D*. Examine it.



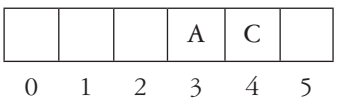
$F = 2, R = 4$
Examine *D*

Step 8: Insert all the adjacent nodes of *D* to *Q* and mark them as *visited*.



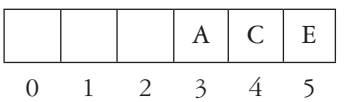
$F = 2, R = 4$, *B C F* are not included

Step 9: Take the front node *F*. Examine it.



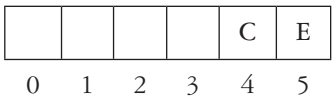
$F = 3, R = 4$
Examine *F*

Step 10: Insert all the adjacent node of *F* to *Q* and mark them as *visited.*



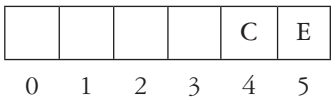
$F = 3, R = 5$, *C*, *D* are not included

Step 11: Take the front node *A*. Examine it.



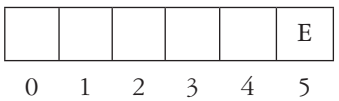
$F = 4, R = 5,$
Examine *A*

Step 12: Insert all the adjacent nodes of *A* to *Q* and mark them as *visited.*



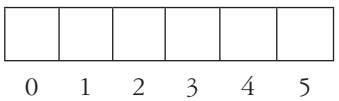
$F = 4, R = 5$, *B*, *C* are not included

Step 13: Take the front node *C*. Examine it.



$F = 5, R = 5$
Examine *C*

Step 14: Insert all the adjacent nodes of *C* to *Q*.



$F = -1, R = -1$, *A*, *B*, *D* and *F* are not included

Step 15: Stop now because *Q* is empty ($F = R = -1$).

The BFS order is EBDFACE. The order depends on the sequence of inserting the elements in the queue.

Program 1 A program in C to implement BFS.

```
#incl<sdtedi o>.h
#incl<rcoben i o>.h
#define  MAX  10
#define  True  1
#define  False  0
int  visited[MAX],i,j,k;
int  MAT[MAX][MAX],N,node;
main()
{
clrscr();
printf("  Enter  the  order  of  t=%de) :g r"a,p hM A(Xm)a;x
scanf%d(" ,&N);
printf("\n  Enter  the  edges  ");
for=0i;<N;++i)
{
visite=d0[;i]
for=0j;<N;++j)
```

```
{
printf("\nEdge %d from %d to %d ", i,j);
scanf("%d", &MAT[i][j]); /*Enter 1 for edge o
}
}
printf("Enter the node from where BFS is
scanf("%d", &node);
BFS(node);
getch();
}
BFS(int x)
{
int QUEUE[MAX], front=rear=1;
getch();
visited[x]=1;
printf("\nvisited %d", node);
QUEUE[rear]=x;
x=QUEUE[front];
if(x==N-1)
{
printf("\nAll nodes have been visited");
exit(1);
}
printf("%d\t", x);
for(j=0;j<N;j++)
if(MAT[x][j]==1&& visited[j]==0)
{
QUEUE[rear]=j;
BFS(j);
}
}
```

Depth First Search

The BFS takes more time and space because it traverses all the nodes. There is another method of graph traversal called depth first search (DFS) in which all descends of a vertex (node) are processed before we move to an adjacent vertex. The DFS algorithm is roughly analogous to tree traversal in pre-order.

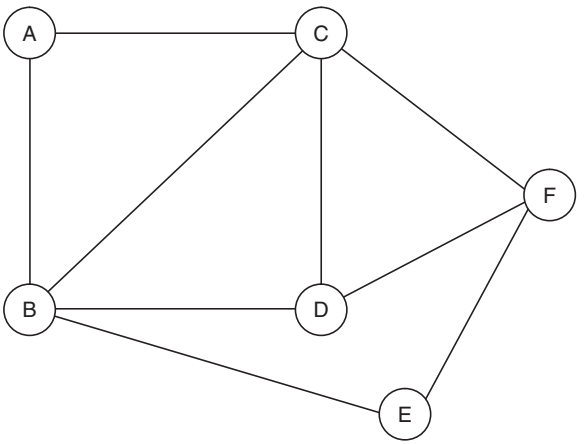
In this method, the traversal starts at the starting source vertex and traverses all the nodes along the path which begins at the source vertex. The implementation of the DFS is similar to the BFS except that *stack* is used in place of *queue*. After coming to the end of the path, we backtrack on the path until we can continue along another path and so on. The advantage of DFS is that it gives the solution of the problem in lesser time and it requires less space. The disadvantage of the DFS is that we can blindly follow the path (by traversing the vertices lying in that path) in which solution does not exist. For example, solution exists in the path ACFE for the graph shown in Problem 3. However, we follow the path ABE due to the sequence of pushing the elements in the stack. The algorithm for DFS is summarized in Algorithm 2.

Algorithm 2 DFS

1. Start traversing the given graph with the given number of vertices and edges.
2. Push the starting source vertex *S* to the stack and mark it as visited.
3. Repeat
   a. Remove the top node *K* of the stack and examine it.
   b. Push all adjacent nodes of *K* (which are not visited so far) to the stack and mark them as visited.
4. Until stack is empty
5. Exit.

Problem 2

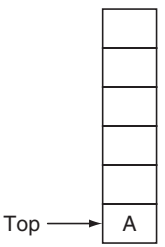
Traverse the following graph using the DFS starting from A.



Solution

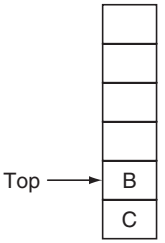
Step 1: Start traversing the given graph with number of vertices and edges.

Step 2: Push the starting source vertex *A* into the vertex and mark it as visited.



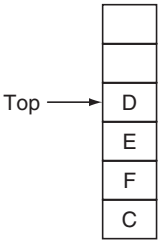
Step 3: Take away (pop) the top node *A*. Examine it.

Step 4: Push all the adjacent nodes of *A* to the stack in any order and mark them as visited.



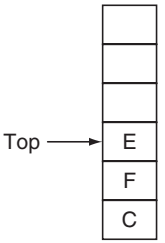
Step 5: Take away the top node *B*. Examine it.

Step 6: Insert all the adjacent nodes of *B* to the stack and mark them as visited.



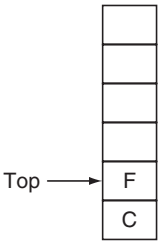
Step 7: Take the top node *D*. Examine it.

Step 8: Insert all the adjacent nodes of *D* to the stack. *B*, *E*, *F* are already in the stack so do not push them into the stack; they are already marked as visited.



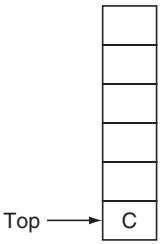
Step 9: Take the top node *E*. Examine it.

Step 10: Insert all the adjacent nodes of *E* to stack but *B* and *F* are already in the stack.



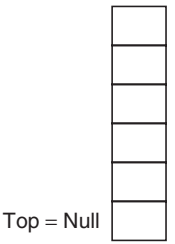
Step 11: Take the top node *F*. Examine it.

Step 12: Insert all the adjacent nodes of *F* to stack but *C*, *D* and *E* are already in the stack.



Step 13: Take the top node *C*. Examine it.

Step 14: Insert all the adjacent nodes of *C* to the stack. *A*, *B*, *D* and *F* are already in the stack.



Step 15: Stop now because stack is empty.

The DFS order is ABDEFC. The order depends on the sequence of inserting the elements in the stack.

Program 2 A program in C to implement DFS using adjacency matrix.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0
int visited[MAX],=1i;, j, k, flag
int MAT[MAX][MAX], N, node;
main()
{
clrscr();
printf("  Enter  the  order  of  t=%de) "g,r aMpAhX );(;m a x
scanf%d(" , &N);
printf("\n  Enter  the  edges  ");
for=0i; <N ; ++i)
{
visited=0[;i]
```

```
for(j=0;j<N;j++)
{
printf("\nEdge %d for node %d ?",i,j);
scanf("%d",&MAT[i][j]);
}
}
printf("Enter the node from where DFS is
scanf("%d",&node);
DFS(node);
getch();
}
DFS(int x)
{
getch();
visited[x]=1;
printf("\nvisit %d",node);
if(++x==N-1)
{
printf("\nAll nodes have been visited");
exit(1);
}
for(j=0;j<N;j++)
if(MAT[x][j]==1&&visited[j]==0)
DFS(j);
}
```

8.5 The Minimum Spanning Tree

Any tree consisting of all the vertices of a graph is called a spanning tree. We have already learned the two different traversal techniques (BFS and DFS) that traverse all the nodes of a graph. Therefore, there may be two ways of constructing a spanning tree. The spanning tree constructed by using DFS is called the depth first search spanning tree; and the tree constructed by using BFS is called the breadth first search spanning tree. The BFS and DFS spanning trees corresponding to the graph in Figure 17(a) are shown in Figures 17(b) and (c), respectively.

We would now select a set of edges that would minimize the total cost of the spanning tree. If the spanning tree contains the edges with minimum cost, then such trees are called minimum spanning trees (MSTs). An MST is a minimal subgraph G' of G such that $V(G') = V(G)$ and $E(G') \subseteq E(G)$. If there are N vertices in the graph then the MST contains $N - 1$ vertices. There are two different algorithms to find out the MST:

1. Kruskal algorithm.
2. Prim's algorithm.

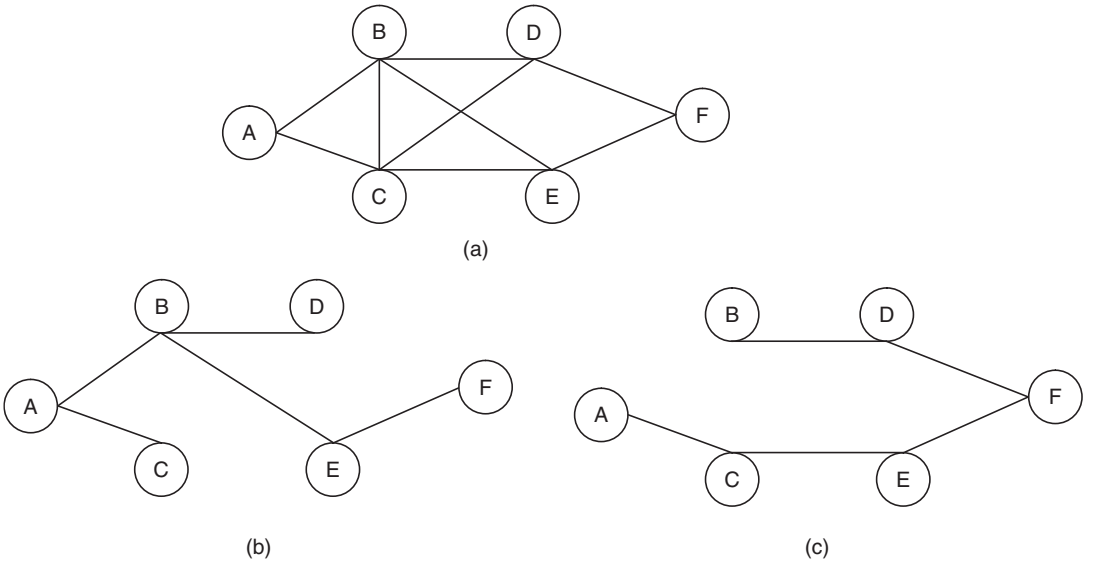


Figure 17 (a) Example graph; (b) BFS spanning tree; (c) DFS spanning tree for graph shown in part (a).

Kruskal Algorithm

This algorithm was developed by Joseph Kruskal. The Kruskal algorithm creates the MST T by adding the edges one at a time to T. A minimum cost spanning tree T is built edge by edge. We start with the edge of minimum cost. If there are several edges with the same minimum cost, then we select any one of them and add it to the spanning tree T, provided its addition does not form a cycle. We then add an edge with next lowest cost and so on. We repeat this process until we have selected $N - 1$ edges to form the complete MST. This algorithm selects the edges for addition in the MST in the decreasing order of their cost. We have to remember here that the edges can only be added if it does not form a cycle. In other words, we can arrange the edges in the ascending order of their cost, and then decide the edge to add into the spanning tree if it does not form a cycle.

Example 2

Let us take the graph shown in Figure 18 to find out the MST using Kruskal algorithm.

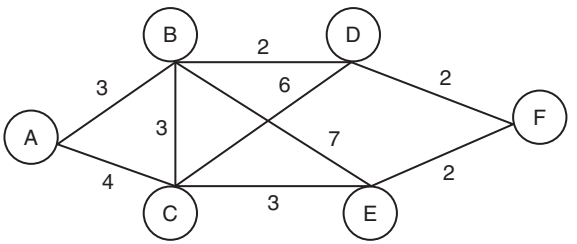
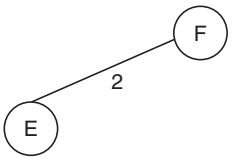
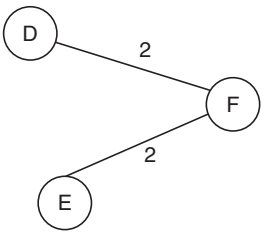


Figure 18 Example graph for Kruskal's algorithm.

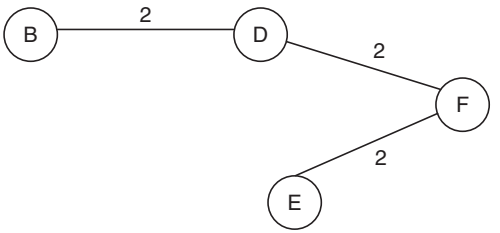
Step 1: The edges with minimum cost are BD, DF and EF. Select any one.



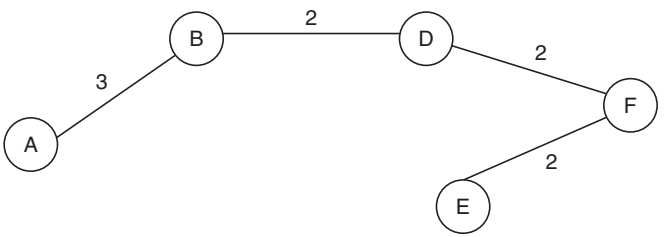
Step 2: The edges with the next minimum cost are BD and DF. Select any one.



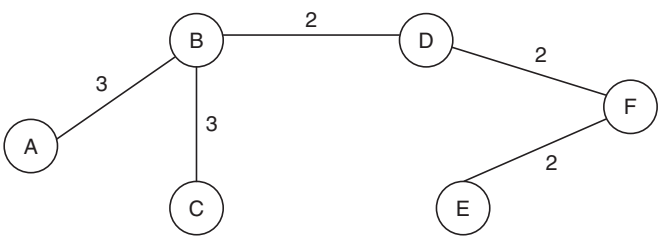
Step 3: The edge with the next minimum cost is BD. So add it if it does not form a cycle.



Step 4: The edges with the next minimum cost are AB, BC and CE. So add any one that does not form a cycle.



Step 5: The edges with the next minimum cost are BC and CE. Add any one that does not form a cycle.



Step 6: Stop, since 5 edges ($N - 1$ where $N = 6$) are included.

We can organize the whole process in a tabular form with three rows and number of columns equal to the number of edges. The first row contains the edges in the descending order of their cost; the second row contains the cost and the third row contains A if the corresponding edge is added. This can be shown for the graph in Figure 18.

Edges	EF	DF	BD	AB	BC	CE	AC	CD	BE
Cost	2	2	2	3	3	3	4	6	7
Add	A	A	A	A	A				

Prim's Algorithm

This algorithm is similar to the Kruskal's algorithm in the manner that it also creates MST T by adding the edges one at a time to T. Here we start with the single vertex of graph (generally the vertex from the lowest cost edge). Let it be A. Then we find the cost of the edges incident on A, add the edge with minimum cost. Let it be AB, so we have two vertices in the set $V = \{A, B\}$. Again find the cost of the edges incident on A and B. Then add the edge with the lowest cost and repeat the same process until $N - 1$ edges are added where N is the number of vertices in the graph. We have to remember here that the edges can only be added if it does not form a cycle.

Example 3

Let us take the graph shown in Figure 19 to find out the MST using Prim's algorithm.

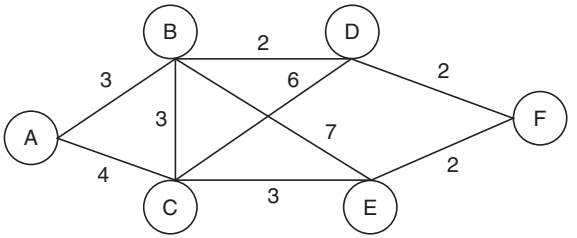
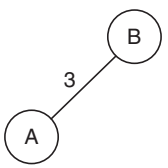


Figure 19 Example graph to show Prim's algorithm.

Step 1: Select the vertex A. Now $V = \{A\}$.

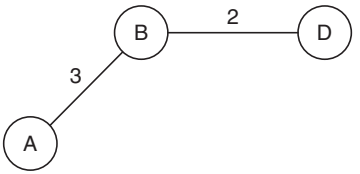
Step 2: The edges incident on A are AB and AC. So add the minimum of AB and AC.

$\text{Min}\{AB, AC\} = \text{Min}\{3, 4\} = 3 = AB$



Now $V = \{A, B\}$.

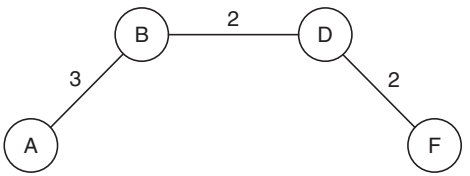
Step 3: Find the edges with minimum cost incident on either A or B. AB and AC are the edges incident on A and BC, BD and BE are incident on B. Since AB is already included so ignore it. Therefore, Min{AC, BD, BE, BC} = Min{4, 2, 7, 3} = 2 = BD. Add BD.



Now $V = \{A, B, D\}$.

Step 4: The edges with minimum cost are incident on A, B or D. AB and AC are the edges incident on A; BC, BD and BE are incident on B and BD, CD and DF are incident on D. Since AB and BD are already included so ignore them. Therefore,

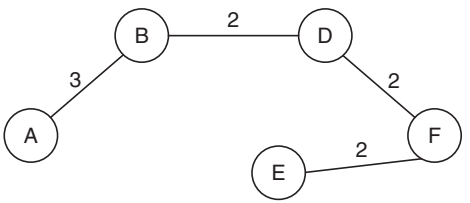
Min{AC, BE, BC, CD, DF} = Min{4, 7, 3, 6, 2} = 2 = DF. Add DF.



Now $V = \{A, B, D, F\}$.

Step 5: The edges with minimum cost incident on A, B, D or F. The edges incident on these vertices and which are not included in the tree are AC, BC, BE, DC, EF. Therefore,

Min{AC, BC, BE, DC, EF} = Min{4, 3, 7, 6, 2} = 2 = EF. So add this edge.

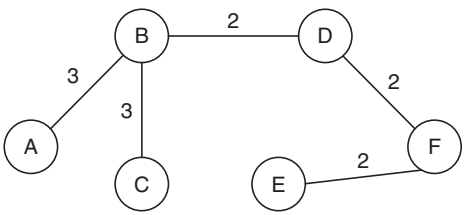


Now $V = \{A, B, D, F, E\}$.

Step 6: The edges with minimum cost incident on A, B, D, F or E. The edges incident on these vertices and which are not included in the tree are AC, BC, BE, DC, EC. Therefore,

Min{AC, BC, BE, DC, EC} = Min{4, 3, 7, 6, 3} = 3 = {BC, EC}.

Select any one (BC) from this set.



Now $V = \{A, B, C, D, F, E\}$.

Step 7: Stop, since 5 edges ($N - 1$ where $N = 6$) are included and V contains all the vertices.

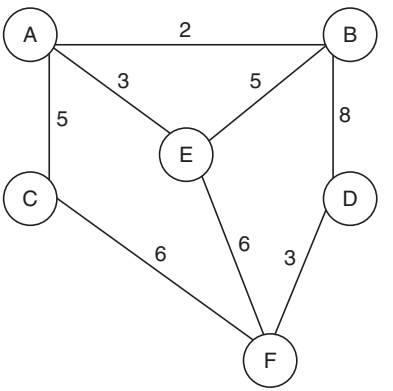
8.6 The Shortest Path

A path with minimum cost or distance between two vertices is said to be the shortest path. There are various algorithms to find out the shortest path. We will describe one of them called the E.W. Dijkstra's algorithm. This algorithm is used to find the shortest path between the source vertex to all other vertices in a graph. Hence, it is called the single-source shortest-path problem. The algorithm is given in Algorithm 4 as follows:

Algorithm 3 Dijkstra's algorithm

1. Set $P =$ {starting vertex S} and $T =$ V}.
2. Assign the index value 0 to the starting vertex S = 0 and ∞ to all other vertices.
3. Repeat.
   a. Take the vertex v_1 with minimum index value, delete it from T and mark it as visited.
   b. Find the new index value of adjacent vertex v_2 with respect to v_1 as follows: Index(v_2) = Min{Index(v_1), Index(v_1) + w(v_1, v_2)}
4. Until T is empty.

Program 3 Find the shortest path from the vertex A.



Step 1: The index value of the source vertex A is assigned to 0 and index value of other vertices to infinity $T = \{A, B, C, D, E, F\}$.

Vertex	*A*	*B*	*C*	*D*	*E*	*F*
Index value	0	∞	∞	∞	∞	∞
T	*A*	*B*	*C*	*D*	*E*	*F*

Step 2: Select the vertex A with minimum index value and delete A from T. So $T = \{B, C, D, E, F\}$. Find the new index value of all adjacent vertices of A. Therefore

NewIndex(B) = Min{Index(B), Index(A) + w(A, B)} = {∞, 0 + 2} = 2

NewIndex(C) = Min{Index(C), Index(A) + w(A, C)} = {∞, 0 + 5} = 5

NewIndex(E) = Min{Index(E), Index(A) + w(A, E)} = {∞, 0 + 3} = 3

Vertex	*A*	*B*	*C*	*D*	*E*	*F*
Index value	0	2	5	∞	3	∞
T		*B*	*C*	*D*	*E*	*F*

Step 3: Select the vertex B with minimum index value and delete B from T. So $T = \{C, D, E, F\}$. Find the new index value of all adjacent vertices of B, that is, A, E, D. However, A is not the member of T so

NewIndex(D) = Min{Index(D), Index(B) + w(B, C)} = {∞, 2 + 8} = 10

NewIndex(E) = Min{Index(E), Index(B) + w(B, E)} = {3, 2 + 5} = 3

Vertex	*A*	*B*	*C*	*D*	*E*	*F*
Index value	0	2	5	10	3	∞
T			*C*	*D*	*E*	*F*

Step 4: Select the vertex E with minimum index value and delete E from T. So $T = \{C, D, F\}$. Find the new index value of all adjacent vertices of E, that is, A, B, F. However, A and B are not the members of T so

NewIndex(F) = Min{Index(F), Index(E) + w(E, F)} = {∞, 3 + 6} = 9

Vertex	*A*	*B*	*C*	*D*	*E*	*F*
Index value	0	2	5	10	3	9
T			*C*	*D*		*F*

Step 5: Select the vertex C with minimum index value and delete C from T. So $T = \{D, F\}$. Find the new index value of all adjacent vertices of C, that is, A, F but A is not the member of T so

NewIndex(F) = Min{Index(F), Index(C) + w(C, F)} = {9, 5 + 6} = 9

It remains unchanged.

Vertex	*A*	*B*	*C*	*D*	*E*	*F*
Index value	0	2	5	10	3	9
T				*D*		*F*

Step 6: Select the vertex F with minimum index value and delete F from T. So $T = \{D\}$. Find the new index value of all adjacent vertices of F, that is, C, E, D but C and E are not members of T, so

NewIndex(D) = Min{Index(D), Index(F) + w(F, D)} = {10, 9 + 3} = 10

It remains unchanged.

Vertex	*A*	*B*	*C*	*D*	*E*	*F*
Index value	0	2	5	10	3	9
T				*D*		

Step 7: Select the vertex D with minimum index value and delete D from T. So $T = \{\}$. Since this is the last vertex so stop.

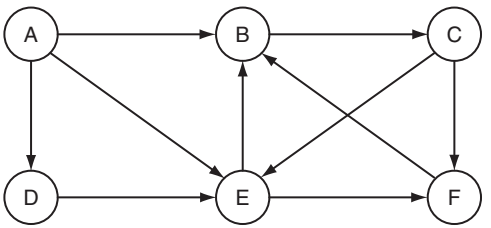
Vertex	*A*	*B*	*C*	*D*	*E*	*F*
Index value	0	2	5	10	3	9
T						

Therefore, now we can see that the shortest path between

A and $B = AB$ of cost 2
A and $C = AC$ of cost 5
A and $D = ABD$ of cost 10
A and $E = AE$ of cost 3
A and F is AEF of cost 9

Solved Examples

Example 1 *Find the adjacency matrix for the following graph.*



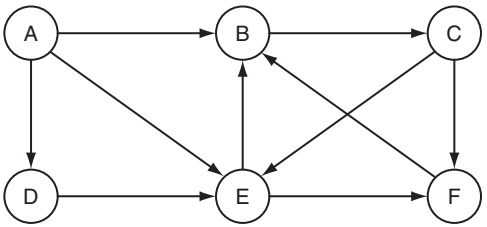
Solution

The adjacency matrix is as follows:

	A	*B*	*C*	*D*	*E*	*F*
A	0	1	0	1	1	0
B	0	0	1	0	0	0

	A	*B*	*C*	*D*	*E*	*F*
C	0	0	0	0	1	1
D	0	0	0	0	1	0
E	0	1	0	0	0	1
F	0	1	0	0	0	0

Example 2 *Find the in-degree and out-degree of all nodes for the following graph.*

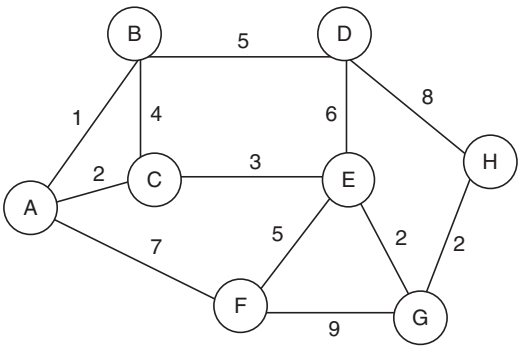


Solution

InDeg(A) = 0
InDeg(B) = 2
InDeg(C) = 1
InDeg(D) = 1
InDeg(E) = 3
InDeg(F) = 2

OutDeg(A) = 3
OutDeg(B) = 1
OutDeg(C) = 2
OutDeg(D) = 1
OutDeg(E) = 2
OutDeg(F) = 1

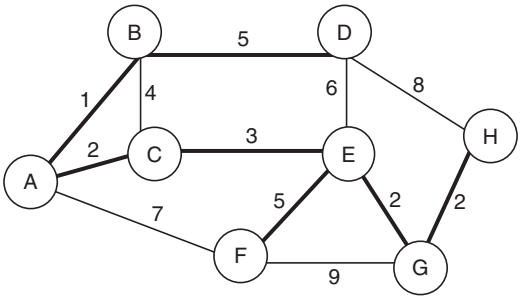
Example 3 *Find the MST for the following graph using Kruskal algorithm.*



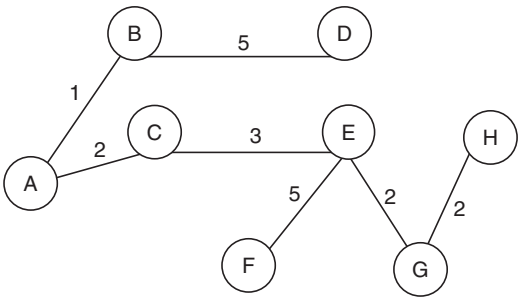
Solution

Edges in increasing order of their cost	*AB*	*AC*	*EG*	*GH*	*CE*	*BC*	*BD*	*EF*	*DE*	*AF*	*DH*	*FG*
Cost	1	2	2	2	3	4	5	5	6	7	8	9
Add	*A*	*A*	*A*	*A*	*A*	*D*	*A*	*A*	*D*	*D*	*D*	*D*

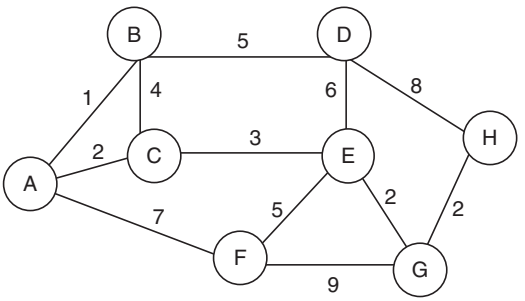
Here *A* stands for addition and *D* for deletion. The edges added are shown in bold in the following figure.



The MST is as follows:



Example 4 *Find the shortest path for the graph shown below from vertex G.*



Solution

Step 1: Since the source starting vertex is G so index value for $G = 0$ and for others $= \infty$.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	∞	∞	∞	∞	∞	∞	∞
T	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*

Step 2: Take the vertex G with minimum index value and find the new index value of the vertices adjacent to G. Delete G from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	∞	∞	∞	∞	∞	∞	∞
T		*A*	*B*	*C*	*D*	*E*	*F*	*H*

NewIndex $(H) = \text{Min}\{\text{CurrentIndex}(H), \text{Index}(G) + w(G, H)\} = \text{Min}\{\infty, 0 + 2\} = 2$

Similarly, the new index for $E = 2$ and for $F = 9$.

Step 3: Take the vertex E (either E or H) with minimum index value and find the new index value of the vertices adjacent to E. Delete E from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	∞	∞	∞	∞	2	9	2
T		*A*	*B*	*C*	*D*		*F*	*H*

NewIndex$(F) = \text{Min}\{9, 2 + 5\} = 7$

NewIndex$(D) = \text{Min}\{\infty, 2 + 6\} = 8$

NewIndex$(C) = \text{Min}\{\infty, 2 + 3\} = 5$

Step 4: Take the vertex H with minimum index value and find the new index value of the vertices adjacent to H. Delete H from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	∞	∞	5	8	2	7	2
T		*A*	*B*	*C*	*D*		*F*	

NewIndex$(D) = \text{Min}\{8, 2 + 8\} = 8$ and so unchanged

NewIndex$(G) =$ Not in T

Step 5: Take the vertex C with minimum index value and find the new index value of the vertices adjacent to C. Delete C from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	7	9	5	8	2	7	2
T		*A*	*B*		*D*		*F*	

NewIndex(A) = Min{∞, 5 + 2} = 7

NewIndex(B) = Min{∞, 5 + 4} = 9

NewIndex(E) = Not in T

Step 6: Take the vertex F with minimum index value and find the new index value of the vertices adjacent to F. Delete F from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	7	9	5	8	2	7	2
T		*A*	*B*		*D*			

NewIndex(A) = Min{7, 7 + 7} = 14 and so no change

NewIndex(G) = Not in T

NewIndex(E) = Not in T

Step 7: Take the vertex A with minimum index value and find the new index value of the vertices adjacent to A. Delete A from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	7	9	5	8	2	7	2
T			*B*		*D*			

NewIndex(B) = Min{9, 7 + 1} = 8

NewIndex(C) = Not in T

NewIndex(F) = Not in T

Step 8: Take the vertex B with minimum index value and find the new index value of the vertices adjacent to B. Delete B from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	7	8	5	8	2	7	2
T					*D*			

NewIndex(D) = Min{8, 8 + 5} = 8, therefore no change

NewIndex(C) = Not in T

NewIndex(A) = Not in T

Step 9: Take the vertex D with minimum index value. This is the only vertex in T, so stop now. Delete D from T.

Vertices	*G*	*A*	*B*	*C*	*D*	*E*	*F*	*H*
Index value	0	7	8	5	8	2	7	2
T								

Summary

1. A graph is defined as a collection of two sets V and E, where V is the set of vertices and E is the set of edges.
2. The degree of a node is equal to the number of edges incident on it.
3. A simple path is a path in which all vertices except possibly the first and the last are distinct.
4. The advantages of matrix representation of graphs are speed and simplicity.
5. Adjacency list is very useful if the graph is a sparse graph.
6. The adjacency lists use space proportional to $V + E$.
7. There are two ways of traversing a graph: BFS and DFS.
8. Spanning tree is a tree which consists of all the nodes of a graph.
9. A minimum spanning tree is a spanning tree with minimum cost.
10. DFS traversal method uses stack and BFS uses queue.

Key Terms

Graph
Traversal
Breadth first search
Depth first search
Spanning tree
Shortest path
Minimum spanning tree
Prim's algorithm
Kruskal algorithm
Dikjstra algorithm

Multiple-Choice Questions

1. The DFS traversal is analogous to
   a. pre-order traversal of a tree.
   b. in-order traversal of a tree.
   c. post-order traversal of a tree.
   d. none of the above.
2. The DFS uses which of the following data structure to hold the nodes?
   a. Stack
   b. Queue
   c. Tree
   d. None of the above
3. The BFS uses which of the following data structure to hold the nodes?
   a. Stack
   b. Queue
   c. Tree
   d. None of the above
4. Which of the following is true for undirected graph?
   a. In-degree is same as out-degree.
   b. In-degree and out-degree are different.
   c. Edge marked as (A, B) is the same as that marked as (B, A).
   d. Both (a) and (c) are true.
5. Depth first search
   a. scans all the vertices of the same level first before moving to the other vertex.
   b. scans all the vertices of the different level first before moving to the other vertex.
   c. scans all the vertices of the same path first before moving to the other vertex.
   d. scans all the vertices on the same path first before moving to the other path.

6. Breadth first search
   a. scans all the vertices of the same level first before moving to the other vertex.
   b. scans all the vertices of the different level first before moving to the other vertex.
   c. scans all the vertices of the same path first before moving to the other vertex.
   d. scans all the vertices on the same path first before moving to the other path.

7. The sum of the degree of all the vertices (say N) in a graph is
   a. even.
   b. odd.
   c. $N/2$.
   d. $N - 1$.

8. The vertex with zero degree is called
   a. isolated vertex.
   b. loop.
   c. pendant vertex.
   d. none of the above.

9. The vertex with degree one is called
   a. isolated vertex.
   b. loop.
   c. pendant vertex.
   d. none of the above.

10. Consider a graph G' with same number of vertices but no edge between vertices. If there is an edge between the same vertices in G then it is called
    a. isomorphic graph.
    b. homomorphism graph.
    c. complementary graph.
    d. complete graph.

11. A graph in which each vertex is connected to every other vertex is called
    a. complete graph.
    b. complementary graph.
    c. regular graph.
    d. isomorphic graph.

12. A graph in which multiple edges and loops are not allowed is called
    a. complete graph.
    b. multi graph.
    c. weighted graph.
    d. simple graph.

13. A graph in which all vertices have equal degree is called
    a. complete graph.
    b. multigraph.
    c. regular graph.
    d. complementary graph.

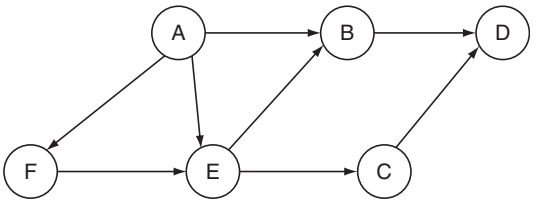
14. Which of the following statements is correct?
    a. The sum of the degree of a graph is twice the number of edges.
    b. The degree of a vertex in a simple graph of N vertices cannot exceed $N - 1$.
    c. The vertex with zero degree is called isolated vertex.
    d. All of the above.

15. Which of the following statements is correct?
    a. The total number of vertices in a graph is called the order of the graph.
    b. A graph having no edge is called the null graph.
    c. Traversal of a graph means visiting all the nodes.
    d. All of the above.

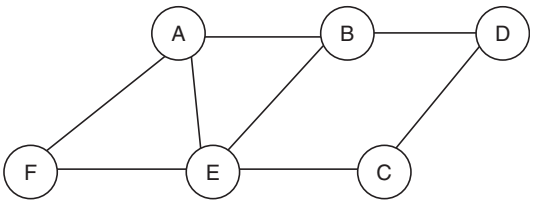
16. Adjacency matrix for a diagraph is
    a. unit matrix.
    b. symmetric.
    c. asymmetric.
    d. none of the above.

Review Questions

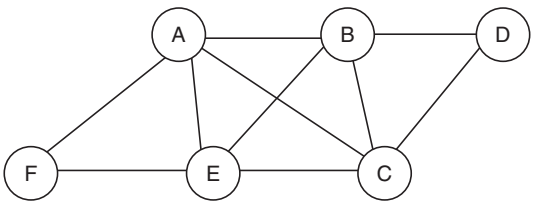
1. Find the degree of each node in the following graph.



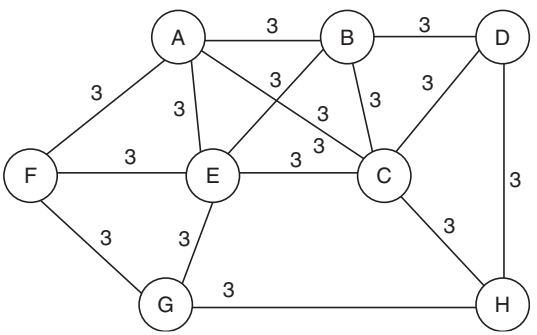
2. Find all the possible spanning trees for the following graph.



3. Find the minimum spanning tree for the following graph.



4. Use the Dijkstra's algorithm to find the shortest path in the following graph.



5. What is a graph? How it is represented?

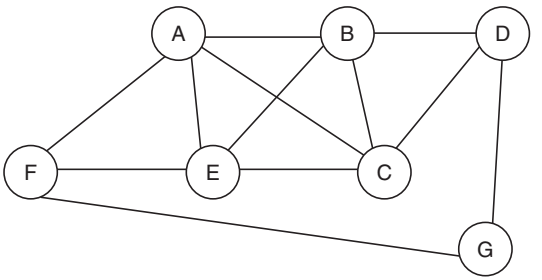
6. What are the different traversal techniques of a graph?

7. Write any minimum spanning tree algorithm.

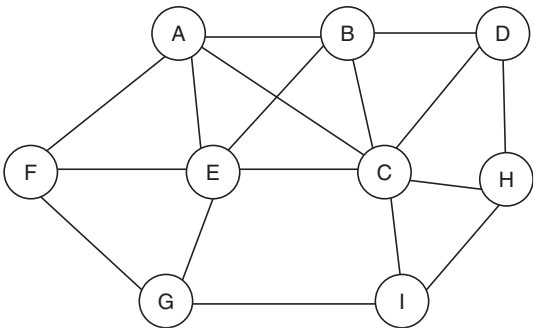
8. Describe the adjacency matrix and adjacency list representation of a graph.

9. Explain a spanning tree and a minimum spanning tree.

10. Perform the breadth first search traversal for the following graph:



11. Perform the depth first traversal for the following graph:



12. Explain the differences between the DFS and BFS.

13. What is the degree of a graph?

14. Define a graph and various terminologies associated with the graph.

15. What do you understand by the order of the graph?
16. Differentiate between strongly connected and weakly connected graphs..
17. What do you understand by the isomorphic graph?
18. Define the regular and complete graph.
19. What is adjacency matrix?
20. How is the graph represented using linked list?
21. What is difference between DFS and BFS?
22. Which data structure is used to implement BFS and DFS?
23. Define minimum spanning tree.
24. How many edges would be there in the minimum spanning tree if there are n vertices in the graph?

Programming Assignments

1. Write a program in C++ to find the minimum spanning tree using Prim's algorithms.
2. Write a program in C to find the shortest path between two nodes using Dikjstra's algorithms.
3. Write a program in C++ to traverse the graph using depth first search method.
4. Write a program in C to traverse the graph using breadth first search method.

Answers

Multiple-Choice Questions

1. (a)
2. (a)
3. (b)
4. (d)
5. (d)
6. (a)
7. (a)
8. (a)
9. (c)
10. (c)
11. (a)
12. (b)
13. (c)
14. (d)
15. (d)
16. (b)

9 Searching

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Problem of searching.
- Importance and use of searching.
- Different searching techniques.
- Process of sequential search and binary search.
- Hashing techniques and collision resolution.

Searching is a very fundamental activity associated in our daily lives, whether we search for a number in the telephone directory, an account number of an employee or a word in the dictionary. Searching is basically a process of finding an element within a list of elements stored either sequentially or randomly. If the element is found in the list then searching is said to be successful, otherwise it is said to be unsuccessful. Searching is broadly divided into two categories depending on whether the search is performed on a list of randomly organized elements or a list of sequentially organized elements. These categories are, respectively, called

1. linear search and
2. binary search.

There is another search technique called search by hashing, which will be discussed later in this chapter. Linear search is also called serial search and it is the simplest way of searching. The binary search is a more efficient searching method than linear search; however, it requires the sorted list to search an element. The efficiency of the searching techniques is measured by the run-time analysis.

9.1 Types of Searching

Searching is classified depending on whether searching is performed on a list of elements in either an organized order or a random order.

Linear Search

This is the simplest method of searching, in which we compare every element of the list with the key element we want to search. If any element of the list matches with the key element then search is said to be successful. A search will be unsuccessful if all the elements of the list are accessed and the key element does not match with any of the elements in the list. Therefore in linear search, we start searching the key element from the beginning of the list. If it matches with the desired key element, then search is successful otherwise we examine each element of the list sequentially until we find the desired key element or the list is finished. If list is finished without finding the match of the desired key element, then the search is said to be unsuccessful.

The linear search is also called sequential search, and it is performed on the list of unordered elements. The performance (efficiency) of the search algorithms is measured in the number of comparisons, which will depend on where the target is found. So the complexity of a search is measured in terms of number of comparisons, and it is expressed as O(*N*) where *N* is the number of comparisons.

Let us discuss the efficiency of a linear search, which is defined as the time taken to find an item matching a key element in the list, in three different cases.

1. **The best case:** For linear search, the best case occurs when the key element is found at the first position of the list. If the desired key element is found at the beginning, then the sequential or linear search will find it quickly because only one comparison is required. Therefore, the efficiency of a linear search in the best case is expressed as O(1).
2. **The average case:** The average case may be considered as the probability of finding the desired key element in the collection of *N* elements. The efficiency of the average case is obtained by averaging the different running times for all inputs. If we are searching for the key element that occurs at the first location, then there is only one comparison required. If we are searching for the key element that occurs at the second location, then there are two comparisons required and so on. The average of all these searches will be $(1 + 2 + 3 + 4 + \ldots + N)/2 = (N + 1)/2$. Therefore, the efficiency of a linear search for average case will be O(*N*).
3. **The worst case:** If the desired key element is found at the end of the list, then *N* comparisons are required. *N* comparisons will also be required in the case of an unsuccessful search, that is, the desired key element is not found. This is because in unsuccessful search, each element of a list of *N* elements is compared with the desired key element. This is called the worst case, where the largest number of array elements is accessed. Therefore, the linear search will take longer time to access more elements to find the element that matches the key element. The efficiency of a linear search in the worst case is expressed as O(*N*).

Algorithm 1 LSEARCH(List, N, KEY)

Here List is a collection of *N* elements and key is the desired element we want to search in the List.

```
1. Set i=0
2. Set flag=0
3. Repeat Steps 4 and 5 while i<N
4.   If (List[i]==KEY)
        {
            Set pos=i
            Set flag=1
            Break
        }
5. Set i=i+1
6.   If (flag==1)
       Print Key is found at Pos
       Else
       Print Key is not found
    End if
7.   Exit
```

Problem 1

Search the element 39 in the following list using linear search.

88	43	24	35	63	39	18	37	80	16
List[0]	List[1]	List[2]	List[3]	List[4]	List[5]	List[6]			

Solution

The key = 39 is given. The linear search will work in the following way:

1. Set i=0; Set fla=0.
2. Compare List [with] key, 88 ≠ 39 so increase i=i+1, that is, i=1.
3. Compare List [with] key, 43 ≠ 39 so increase i=i+1, that is, i=2.
4. Compare List [with] key, 24 ≠ 39 so increase i=i+1, that is, i=3.
5. Compare List [with] key, 35 ≠ 39 so increase i=i+1, that is, i=4.
6. Compare List [with] key, 63 ≠ 39 so increase i=i+1, that is, i=5.
7. Compare List [with] key, 39 = 39 so stop searching now and set pos=i, that is, pos=5 and fla=1.
8. Since fla=1 so print is found at pos=5.

The linear search may be modified by adding the desired key element at the end of the list so that searching of this list is always successful. During the search, if element is found only at the end of the list then search is said to be unsuccessful.

Algorithm 2 LSEARCH(List, N, key)

Here List is a collection of *N* elements and key is the desired element we want to search in the List.

1. Set i=0;
2. Set fla=0;
3. Set List[N]=key;
4. Repeat Step 5 while List[i] ≠
5. Set i=i+1;
6. If (i=N)
   print key is not found;
   Else
      print key is found at i;
   End if
7. Exit

Binary Search

The search can also be performed on the sorted list where the elements are arranged in either ascending or descending order. The searching of a key element in a sorted list will improve the efficiency. Sequential

searching on a sorted file is worthwhile because of its simplicity and efficiency. There is another technique to improve search efficiency of a sorted file; it is called binary search.

In binary search, we first compare the key with the item in the middle position of the array. If they are equal, the search ends successfully. Otherwise, if the key is less than the middle key, then we use the binary search in the lower half of the list and if the key element is larger than the middle element, then we perform the binary search in upper half of the list in a similar manner. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

From the example of searching a word in the dictionary, we can see that the words are arranged in an ascending order. So, we first go to the middle of the dictionary to search the given word; and if the word is smaller than the middle word, then we search the lower half of the dictionary and forget the second half. Otherwise, we search the upper half and forget the first half. In both cases, maximum we need to search the half of the dictionary only. Therefore, in the case of searching a sorted file using binary search, on an average, you will require only $n/2$ comparisons. This is because each step of the algorithm divides the block of items being searched in half, thus narrowing the range of possible locations by examining the element in the middle of this range.

The advantage of a binary search over a linear search is astounding for large numbers because we can divide a set of N items in half at the most $\log_2 N$ times. The prerequisite requirement of a binary search is that the list must be sorted first. So binary search is the efficient method of searching, but this performance comes at a price since sorting of the list is required before searching.

The running time of a binary search in the best case is O(1), since the key element is found in the middle of the list in the first iteration. In all other cases of successful search or in case of unsuccessful search, the efficiency is proportional to $O(\log_2 N)$.

The binary search can be implemented using the recursive and non-recursive methods. We will now present both the recursive and non-recursive methods for binary search.

Algorithm 3 `BSEARCHR(List, key, low, high)`

This searches the key item in the given list, starting with index low and ending with index high using binary search recursively.

```
1. Set mid=(low+high) / 2
2. If (low>high)
        Return (-1)
3. If (key==List[mid])
        Return (mid)
4. If (key<List[mid])
        Call BSEARCHR(List, key, low, mid-
        Else
        Call BSEARCHR(List, key, mid+1, high)
5. Stop
```

From the above algorithm, we observe that when the `key` is less than the `mid` element of the list and search is performed in the lower half, `high` value is changed to `mid-1`. And when the `key` is more than the `mid` element and search is performed in the upper half, `low` value is changed to `mid+1`. Now, we are presenting the non-recursive or iterative method of binary search.

Algorithm 4 `BSEARCHNR(low, high)`

1. Repeat Steps 3, 4 and 5 `while(low<=high)`
2. `mid=(low+high)/2`
3. `If(low>high)`
   `Return(-1)`
4. `If(key==List[mid])`
   `Return (mid)`
5. `If (key<List[mid])`
   `high=mid-1`
   `Else`
   `low=mid+1;`
6. `Return (-1);`

Problem 2

Search 20 in the following list using binary search.
12, 16, 17 , 19, 20, 22, 24, 29, 30, 32, 37

Solution

0	1	2	3	4	5	6	7	8	9	10
12	16	17	19	20	22	24	29	30	32	37

1. We find that `low=0, high=10`, so `mid=(0+10)/2`
2. Now, compare `key(20)` with `List[5]`, that is, 22. We can see that `key<List[5]` the value of `high` is changed to `mid-1`

0	1	2	3	4	5	6	7	8	9	10
12	16	17	19	20	22	24	29	30	32	37
					`key<List[5]`					
12	16	17	19	20	22					

3. Therefore, `low=0` and `high=5`, so `mid=(low+high)/2=5/2=2`
4. Now, compare `key(20)` with `List[2]`, that is, 17. We can see that `key>List[2]` the value of low is changed to `mid+1`.
5.

0	1	2	3	4	5
12	16	17	19	20	22
			`key>List[2]`		
			19	20	22

6. Therefore, `low=3 (mid+1)` and `high=5`, so `mid=(low+high)/2=8/2=4`.

Program 1 A program in C++ to implement linear and binary search on a sorted list of MAX elements.

```
#include<iostream.h>
#include<conio.h>
#define MAX 5
template<class T>
class SEARCHING
{
T List[MAX],ITEM;
public:
void readdata()
{
for(int i=0;i<MAX;i++)
cin>>List[i];
}
int BSEARCH()
{
int low=0;
int high=MAX-1;
int mid;
cout<<"Enter the item to search";
cin>>ITEM;
while(low<=high)
{
mid=(low+high)/2;
if(ITEM==List[mid])
return mid;
else if(ITEM<List[mid])
high=mid-1;
else
low=mid+1;
}
return (-1);
}
int LSEARCH()
{
cout<<"Enter the ITEM to search";
cin>>ITEM;
for (int i=0;i<MAX;i++)
{
if (ITEM==List[i])
{
return i;
}
}
return (-1);
}
```

```
void Display()
{
for(i=0;i<MAX;i++)
cout<<"List["<<i<<"]"<<List[i]<<endl;
}
};
void main()
{
int ch,value;
SEARCH obj;
clrscr();
cout<<"Enter the list of elements in ascending o
obj.readdata();
clrscr();
cout<<"List of elements is"<<endl;
obj.Display();
do
{
cout<<"1. Binary search"<<endl;
cout<<"2. Linear search"<<endl;
cout<<"Enter your choice(0 to quit)";
cin>>ch;
switch(ch)
{
case 1:
value=obj.BSEARCH();
if(value==-1)
cout<<"Item is not found";
else
cout<<"Item is found at location"<<value+1;
break;
case 2:
value=obj.LSEARCH();
if(value==-1)
cout<<"Item is not found";
else
cout<<"Item is found at location"<<value+1;
break;
}
}while(ch!=0);
getch();
}
```

Output

```
List of elements is
List[0] 2 4
List[1] 4 5
List[2] 6 7
List[3] 7 8
List[4] 9 0
1. Binary Search
2. Linear Search
Enter your choice(0 to quit)
1
Enter the item to search90
Item is found at 4location
1. Binary Search
2. Linear Search
Enter your choice(0 to quit)
0
```

9.2 Hashing

In a linear search we compare the key element with each element of the list, and in binary search the key element is compared with the middle element of the list. Further search is based on the value of the key element, whether it is smaller or larger than the middle element as already described in the previous section. In the binary search, the elements must be organized in a particular order. Therefore, the searching techniques described so far to find the required key data are based on (1) performing several tests by comparing keys with the elements of the given list and (2) the order in which the elements are inserted affects the number of keys to compare. Another searching technique that avoids number of comparisons and goes directly where the required data is present is called hashing. Hashing searches the element in a constant search time, no matter where the element is located in the list.

Let us look at an example. We have a list of students of a CSE branch and the information about each student in that branch is to be maintained. Since there are limited numbers of students in a branch, let us assume that there are 100 students only. Each of the 100 students has roll number in the range 0–99. If we store the students' records in the array, then it may be maintained in an array of 100 elements. The index of an array will represent the student of a particular roll number; that is, array index 1 will represent the student with roll number 1 and so on. Let us assume that we have to maintain the following information for each student.

Roll, Name, Age, Address and Mark for each student. So this will be an array of struct type defined as follows.

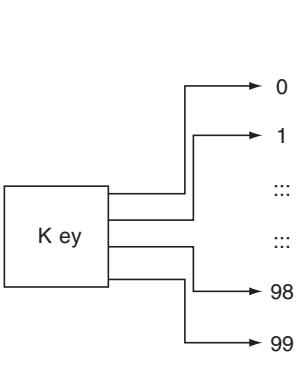
```
Student record[100];
```

where Student may be defined as

```
struct Student
{
   int RollNo;
   char Name[20];
```

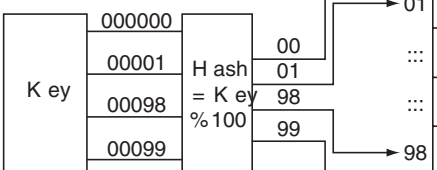
```
    int  Age;
    char  Address;
    int  Marks[5]
};
```

The record for student of roll number n can be directly accessed through the array index since we know it is in `student[n]`. Therefore, there is a one-to-one correspondence between the student's roll number and the array index as shown in the following figure.



	Roll No	Name	Age	Address	Marks
0	0	Sanjay	22	Bhopal	82
1	1	Vikas	34	Indore	63
:::	:::::	::::::::::	::::::	:::::::	::::
:::	::::::::::	::::::::::	:::::	:::::::	::::
98	98	Neelesh	26	Hatta	66
99	99	Prabhu	37	Sagar	87

This strategy works well for small number of students; however, if we have 10,000 students in the institute, we need an array of 10,000 elements. In that case also if we are dealing with the students of CSE branch, then we know that there are only 100 students whose record is to be maintained. However, we have an array of 10,000 elements and so there is a huge wastage of memory because only 100 locations are required to maintain the information of 100 students of CSE branch. One approach for this problem is to convert a roll number into an integer within a limited range, that is, 0–99. By doing so, we can use the array of 100 elements by using only the last two digits of the roll number, if first 100 roll numbers are assigned to CSE students. For example, a student with roll number 13245 will be stored in array index 45 (13245%100=45). The technique in which the address is transformed is called hashing, and the function that maps key space into address space is called a hash function. Hashing is a key to address transformation in which the keys map to addresses in a list. The array in which the address space is defined is called hash table. So the hash function generates the address that covers the entire set of indices in the hash table where the record may be located.



	Roll No	Name	Age	Address	Marks
00	00000	Sanjay	22	Bhopal	82
01	00001	Vikas	34	Patera	63
:::	:::::::::	:::::::	::::::	::::::::::	:::::::::
:::	::::::::::	:::::::	::::::	::::::::::	:::::::::
98	00098	Neelesh	26	Hatta	66
99	00099	Prabhu	37	Damoh	87

Hashing can provide almost direct access to records through hash function that maps keys from a large domain to array indices of a smaller range. Hash functions are used in hash tables. The easiest way to conceptualize a hash table is to consider it as an array. Hash tables use an array to hold, or reference, the stored records. Hash tables are most useful when a large number of records of data are to be stored. Hash tables are commonly used for symbol table by a language translator in compiler design, associative arrays, sets and caches. Hash table is a data structure that can provide constant time [O(1)] lookup on an average, and speed up information searching by a particular aspect of that information, regardless of the number of elements in the table. The hash table is used to process the hash value, generated by applying some function on the key, which determines where the record will be stored in the data structure. The hash function must be chosen so that its return value is always a valid index for the array.

When the hash function maps to the same location in the hash table for two different keys, then this is called a collision. Therefore, hash collision occurs when two distinct inputs into a hash function produce identical outputs. For example, the student with roll number 0012 and 1112 will result in the same hash key 12 (0012%100 and 1112%100). So both the roll numbers are hashed to the same array index 12. The hash function that never produces a collision is called a perfect hash function. However, it is very difficult to form a perfect hash function and it cannot always be found. The hash function that minimizes collisions by spreading the elements uniformly throughout the array is called a good hash function. A good hash function should have the following characteristics:

1. It may be formed by using some mathematical transformation.
2. It should be very easy and quick to compute.
3. It should produce a relatively random and unique distribution of values within the hash table so that it produces as few collisions as possible.
4. It must minimize collisions.

9.3 Hash Functions

In this section, we will study some of the popular hash functions.

Div ision Method

This is most commonly used hash function, and is defined as follows.

```
H a s h ( k=k e y %H a s h _ T a b l e _ S i z e
```

In this method, the modulo arithmetic (%) operator transforms a key into hash value and generates the addresses in the hash table. The hashing is done by taking the remainder of the division of the key by hash table size. If the array index starts from 1, then we can modify the formula as

```
H a s h ( k=k e y %H a s h _ T a b l e+1 S i z e )
```

So if key is 13456 and the table size is 100 then

```
H a s h   ( 1 3=4 15 3 6 4%1 60 0=5 6 or
H a s h   ( 1 3=4 15 3 6 4%1 60 0+1=5 7 (if hash table index starts from 1)
```

In this method the hash table size is usually chosen to be a large prime number because this will generate a more uniform distribution of addresses, and it will not yield the same remainder. The hash table size is larger so it is less likely that two keys hash to the same location. If a set of keys does not

contain integers, they must be converted into integers before applying any of the hashing functions. For example, to transform a string key, we can find the sum of the ASCII codes of the characters in the string and then use the division method as described here. The division method is very simple to use and calculate.

Mid-Square Method

In this method we calculate the square of the key, and then the hash key is obtained by selecting an appropriate number of digits from the middle of the square. Therefore, we can say that the hash key is obtained by deleting digits from both ends of the square of the key. The number of digits chosen for the hash key depends on the number of digits allowed in the index. And the same number of digits from the same positions of the square of the key must be used for all the keys. For example

Key	*Square of key*	*Hash key*
1724	2972176	72
2457	6036849	36
3241	10504081	50
4112	16908544	90
2134	4553956	53

Here we are taking the values at third and fourth positions counting from the left of the square of the key value as shown in bold face in the above table.

Folding Method

In the folding method, a key K is portioned into several parts k_1, k_2, k_3 … k_n where each part has the same number of digits (usually equal to the numbers in the required address) except the last part. The parts are then added together and the last carry is ignored. Therefore, hash key for the folding method is defined as

$$Hash(K) = k_1 + k_2 + k_3 + \ldots + k_n$$

For example, a 10-digit key can be divided into five parts of equal size; each part containing two digits if the hash table index is composed of two digits only. For example, the key 1234506789 is hashed into

Hash(12 34 50 67 89) = 12 + 34 + 50 + 67 + 89 = 252 (ignoring the last carry 2)

The 10-digit key can be divided four parts: three parts of equal size containing three digits and last part containing only one digit if array index is composed of maximum three digits. For example, the key 1234506789 is hashed into

Hash(123 450 678 9) = 123 + 450 + 678 + 9 = 1260 (ignoring the last carry 1)

For example, if a key is very large, we can first apply folding method to reduce the size of the key and then apply any other hashing function. The major drawback of these hashing functions is that they do not preserve the order of the key. This means that the hash values of the two keys are not necessarily in the same order as the keys themselves.

9.4 Collision and Collision Resolution Techniq ues

Ideally, two keys should not be transformed into the same hash address, but unfortunately no existing hash function guarantees this. In fact, we may also combine any of the two hashing functions one after another to get better results. However, the keys' domain are usually larger than the range of the hash table, and so, many keys are mapped to the same address in the hash table. This situation is called hash collision, which occurs when a hash function generates the hash table address that is already occupied by some other element. So when two or more keys hash to the same address in the array, there is a collision. For example, students with roll numbers 412 and 512 will hash to the same address 12; therefore, there are now two different records that belong to `s t u d e n t`. Hence, collision occurs at index number 12.

To resolve this, a collision resolution strategy is used. There are two types of collision resolution techniques which are used to resolve these collisions:

1. Open addressing and
2. chaining.

The collision resolution techniques search for an empty place somewhere else in the table for colliding records.

Col l ision R esol ution b y Open Ad d ressing

One way to resolve collisions is to search for an open or unoccupied place in the hash table where the colliding element can be placed. There are two ways to search for an open place.

1. **Linear probing:** This is the simplest way of resolving the collision. In this method, we compute the `H a s h ( k e y )`. If the `H a s h _ T a b l e [ H a s h ( k e y ) ]` already contains the element, then we go for the next location of the hash table by using `H a s h _ T a b l e [ H a s h ( k e y ) + 1 )`. If that location is also filled, then we go for the next location of the hash table using `H a s h _ T a b l e [ H a s h ( k e y ) + 2 )` and so on until an empty position is found and the record is placed at that position. Therefore, in linear probing we perform the sequential searching in the hash table to find the empty position. If the key hashes to the highest numbered index in the hash table and that space is already occupied, then we go to the starting index of the hash table. The array is considered to be circular so that when the last index is reached we search to the first index of the array. The linear probing is explained in the Figure 1.

   Linear probing is easy to use and implement; and elements remain near their home address and form the cluster. All the elements are placed in contiguous storage, so this speeds up the sequential searches when collisions occur. In this method when the hash table is large and empty, the record may be placed at an empty position within the array. However, when the array becomes half full, the inserted elements form a cluster. For example, in the Figure 1 we can see that 78, 68 and 48 form a cluster, so the next element with the hash value 8 will require more time to find a free cell. This is the major drawback of linear probing and this clustering problem is compounded, because keys that collide are loaded relatively close to the initial collision point.
2. **Quadratic probing:** The quadratic probing is a way to reduce the clustering problem of linear probing by selecting rehash function. This function allows the quick movement of the key within a considerable distance from the initial collision, by generating the random sequence of positions instead of a linear sequence. This method examines certain cells away from the original probe point. When the collision occurs, the quadratic probing applies the rehash function that is defined as follows.

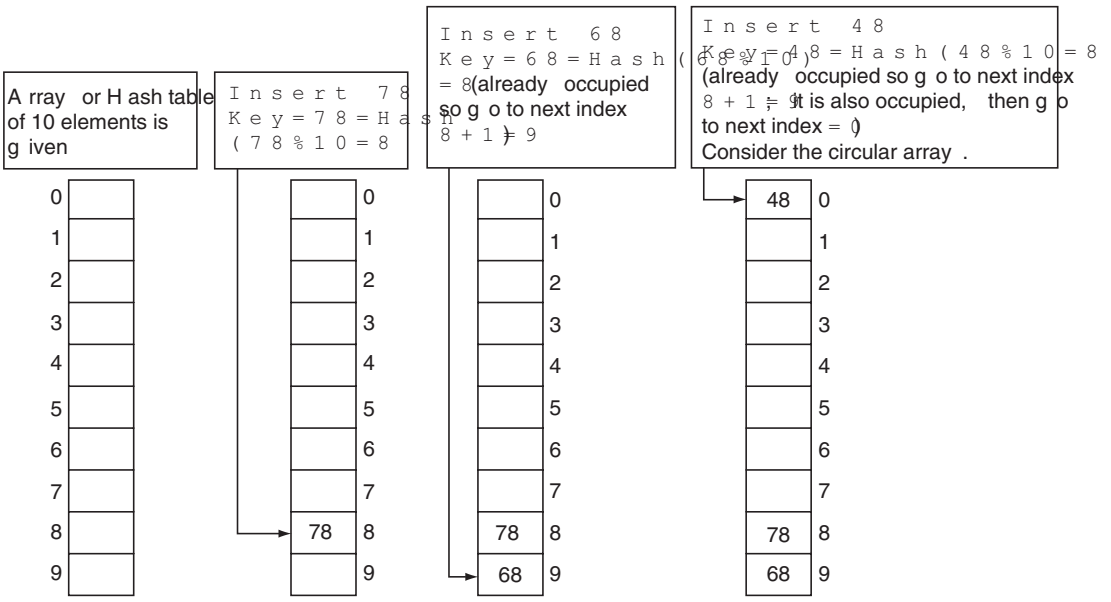


Figure 1 Collision resolution using linear probing.

R e H a s h (= k (e H y a s h (k i e) y % H a s h _ T a b l e _ s i = z 1 e , 2 f , 3 r n i . .

Therefore, quadratic probing probes the array at locations H a s h (k + 1 y) H a s h (k + 4 e , y) H a s h (k + 9 , y t c). Thus, for the first probe we add $m1^2$, for the second collision probe we add 2^2 and so forth until we either find an empty location or we exhaust the possible element.

R e H a s h (= k (e H y a s h (k i e) y % H a s h _ T a b l e _ s i z e
= [H a s h (k 1 e) y] H a s h _ T a b l e _ s i z e

If there is a collision, then it applies the rehash function again

= [H a s h (k 4 e) % H a s h _ T a b l e _ s i z e

If there is a collision, then it applies the rehash function again

= [H a s h (k 9 e) % H a s h _ T a b l e _ s i z e

and so on

One of the major disadvantages of quadratic probe is that it requires time to square the probe number and it may not visit all the indexes of the hash table, and therefore may not generate a new address for every element in the list. Quadratic probing does not suffer much clustering as compared to linear probing. The techniques of linear probing and quadratic probing are called rehashing techniques. The rehashing function can either be a new hash function or a reapplication of the original hash function (Figure 2).

Col l ision R esol ution b y Chaining

There is another technique called chaining for dealing with collisions. In chaining, the hash array is a collection of pointers where each pointer points to a separate list of elements maintained in a linked list.

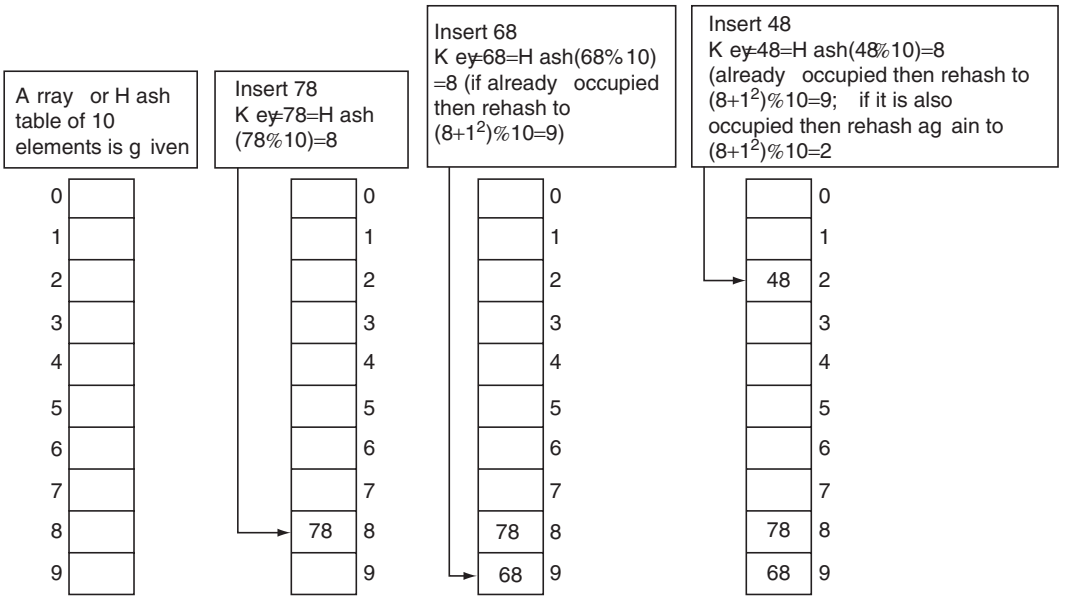


Figure 2 Collision resolution using quadratic probing.

In this method, the hash value does not represent the actual location, instead it represents the address of the linked list that contains all the elements of the same hash value. It is possible to keep only the header nodes in the hash array and the actual elements are kept in the linked list. Therefore, chaining is implemented by using an array where each element of an array is a linked list.

Each node in a particular linked list will have a key that hashes to the same value. To search for a given element is such a structure, we first apply the hash function to the key and then search the chain with the same hash key for the element. If the hash array contains null at the hashed address, then the linked list associated with that hash key is empty, and so the element is not present; although, we can insert the new element into the chain in its proper array component along with other entries that happened to hash to the same array index. The chaining does not form a cluster, so it is better than the linear or quadratic probing. In chaining, each element of the hash array can hold more than one entry as opposed to linear or quadratic probing, where only one element can be present at any location of the array. The disadvantage of chaining is that it requires extra space. This technique is also called as hashing with buckets that allow multiple element keys to hash to the same location. Each of the linked lists associated with different indexes of the array is called a bucket. When the bucket becomes full, we must again deal with handling collision. The collision resolution with chaining is shown in Figure 3.

Solved Examples

Example 1 *Search the element 39 in the following list using binary search.*

16	18	24	35	37	39	43	63	80	88
List[0]	List[1]	List[2]	List[3]	List[4]	List[5]	List[

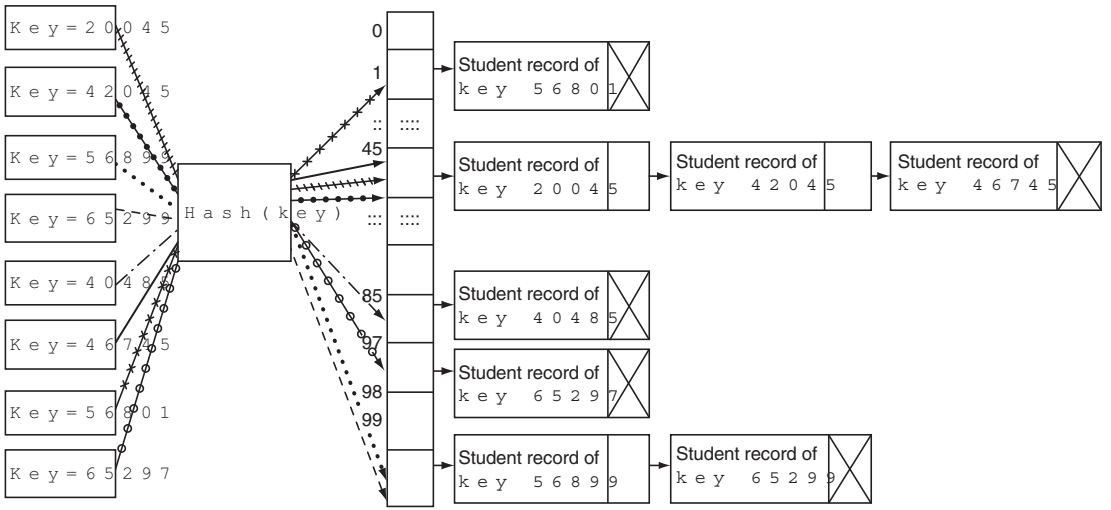


Figure 3 A collision resolution using chaining.

Solution

The key = 39 is given, lo = 0 and high = 9. The binary search will work in the following way:

1. First, the mid = (lo + high) / 2 = (0 + 9) / 2. Integer division results in 4. The mid element 4 holds the number 37, which is less than the key. We will look in the upper half of the given list, so lo is changed to mid + 1. Therefore, now lo = 5 and high = 9.

39	43	63	80	88	
List[5]	List[6]		List[7]	List[8]	List

2. Now, mid = (lo + high) / 2 = (5 + 9) / 2, integer division results in 7. The mid element 7 holds the number 63, which is more than the key. We will look in the lower half of the recent sublist, so high is changed to mid − 1. Therefore, now lo = 5 and high = 6.

39	43
List[5]	List[6]

3. Now, mid = (lo + high) / 2 = (5 + 6) / 2, integer division results in 5. The mid element 5 holds the number 39, which is equal to the key. Therefore, stop searching here. The key element is found at location 5.

Example 2 *Search the element 49 in the following list using linear search by placing the key element in the last location.*

88	43	24	35	63	39	18	37	80	16	49	
List[0]	List[1]	List[2]	List[3]	List[4]	List[5]	List[6]	List[7]				

Solution

The k e y = 4 9 is given, l o w = 0 and h i g h = 1 1. The binary search will work in the following way:

1. Set i = 0; Set f l a g = 0 and L i s t [1 0] is given
2. Compare L i s t [0] with k e y, 8 8 ! = 4 9 so increase i=i+1, that is, i=1.
3. Compare L i s t [1] with k e y, 4 3 ! = 4 9 so increase i=i+1, that is, i=2.
4. Compare L i s t [2] with k e y, 2 4 ! = 4 9 so increase i=i+1, that is, i=3.
5. Compare L i s t [3] with k e y, 3 5 ! = 4 9 so increase i=i+1, that is, i=4.
6. Compare L i s t [4] with k e y, 6 3 ! = 4 9 so increase i=i+1, that is, i=5.
7. Compare L i s t [5] with k e y, 3 9 ! = 4 9 so increase i=i+1, that is, i=6.
8. Compare L i s t [6] with k e y, 1 8 ! = 4 9 so increase i=i+1, that is, i=7.
9. Compare L i s t [7] with k e y, 3 7 ! = 4 9 so increase i=i+1, that is, i=8.
10. Compare L i s t [8] with k e y, 8 0 ! = 4 9 so increase i=i+1, that is, i=9.
11. Compare L i s t [9] with k e y, 1 6 ! = 4 9 so increase i=i+1, that is, i=1 0
12. Compare L i s t [1 0] with k e y, 4 9 = = 4 9 so Stop searching now since i=N so p r i n t k e y is not found.

Example 3 *Write a menu-driven program in C to search an element in the list of elements using linear search or binary search.*

Solution

```
# i n c l < s d t e d i o > . h
# i n c l < c o n i o > . h
# d e f i n e  M A X  5
i n t  L i s t [ M A X ] , I T E M , i ;
v o i d  r e a d d a t a ( )
{
f o r ( i = 0 ; i < M A X ; + + i )
s c a n f ( " % d " , & L i s t [ i ] ) ;
}
i n t  B S E A R C H R ( i n t  I T E M ,  i n t  l o w ,  i n t  h i g h )
{
i n t  m i d ;
m i d = ( l o w + h i g h ) / 2 ;
i f ( l o w > h i g h )
r e t u r n ( - 1 ) ;
i f ( L i s t [ m i d ] = = I T E M )
r e t u r n ( m i d ) ;
i f ( I T E M < L i s t [ m i d ] )
B S E A R C H R ( I T E M , l o w , m i d - 1 ) ;
e l s e
```

```
BSEARCHR(ITEM,mid+1,high);
}
int LSEARCH(int ITEM)
{
List[MAX]=ITEM;
for(i=0;i<=MAX;++i)
{
if(ITEM==List[i])
{
return i;
}
}
return(-1);
}
void Display()
{
for(i=0;i<MAX;++i)
printf("%d\t%d\n", i, List[i]);
}
void main()
{
int ch,value,i=0,ITEM,cnt
clrscr();
printf("Enter the list of elements in sorte
readdata();
clrscr();
printf("List of elements is\n");
Display();
do
{
printf("\nEnter the item to search\n");
scanf("%d",&ITEM);
printf("\n1. Binary Search\n");
printf("2. Linear Search\n");
printf("Enter your choice(0 to quit) ");
scanf("%d",&ch);
switch(ch)
{
case 1:
val=BSEARCHR(ITEM, 0,MAX);
```

```
if (val==-1)
printf("Item is not found\n");
else
printf("Item is found at location %d\n",val+1);
break;
case 2:
val=LSEARCH(ITEM);
if (val==MAX)
printf("Item is not found\n");
else
printf("Item is found at location %d\n",val+1);
break;
}
cnt++;
}while(getch()!=27);
getch();
}
```

Output

```
List of elements is
List[0]=10
List[1]=31
List[2]=52
List[3]=73
List[4]=94

Enter the item to search 5
1. Binary Search
2. Linear Search
Enter your choice(0 to quit) 1
Item is found at location 2

Enter the item to search 4
1. Binary Search
2. Linear Search
Enter your choice(0 to quit) 2
Item is not found
```

Summary

1. Sequential search is slow but robust with efficiency of O(N).
2. The efficiency of binary search algorithms is O($\log_2 N$), but it is appropriate only for arrays in a particular order.
3. Hash function generates an integer number when applied to the key value, and this number can be used as an array index.
4. Hashing is a searching technique that avoids number of comparisons and goes directly where the required data is present.

5. Hashing searches the element in a constant search time O(1), no matter where the element is located in the list.
6. Hash table can be searched in O(1) time using a hash function.
7. When two or more key values hash to the same location in an array, such a situation is called collision.
8. The goal of hashing is to produce a search that has an appropriate efficiency of O(1).
9. In linear probing, the sequential searching in the hash table is performed to find the empty position.
10. Linear probing is easy to use and implement, but suffers from the clustering problem.
11. The quadratic probing is a way to reduce the clustering problem of linear probing.
12. The rehash function of quadratic probing allows the quick movement of key within a considerable distance from the initial collision, by generating the random sequence of positions instead of linear sequence.
13. One of the major disadvantages of quadratic probing is that it requires time to square the probe number.
14. In quadratic probing, all indexes of the hash table may not be visited so it may not generate a new address for every element in the list.
15. Quadratic probing does not suffer much clustering as compared to linear probing.
16. The techniques of linear and quadratic probing are called rehashing techniques.
17. A good hash function minimizes collisions.
18. Sequential search is used when the records are not stored in any order.
19. Binary search is not good for a linked list since it requires jumping back and forth from one end to the middle.

Key Terms

Linear search
Sequential search
Binary search
Hashing
Linear probing
Quadratic probing
Division method
Mid-square
Folding method
Collision
Chaining

Multiple-Choice Questions

1. Linked lists are used in
   a. double hashing.
   b. quadratic probing.
   c. separate chaining.
   d. all of the above.
2. Primary cluster occurs in
   a. linear probing.
   b. quadratic probing.
   c. bucket hashing.
   d. all of the above.
3. Rehashing can be used in
   a. quadratic probing.
   b. linear probing.
   c. separate chaining.
   d. all of the above.
4. Which of the followings are the collision resolution techniques chaining?
   a. Rehashing
   b. Linear probing
   c. Quadratic probing
   d. All of the above
5. Secondary cluster occurs in
   a. linear probing.
   b. quadratic probing.
   c. bucket hashing.
   d. all of the above.

6. Which of the following statements is not correct?
   a. The techniques of linear probing and quadratic probing are called rehashing techniques.
   b. A good hash function minimizes collisions.
   c. Sequential search is used when the records are in stored in descending order.
   d. Binary search is not good for a linked list.

7. Which of the following statement is correct?
   a. Sequential search is slow.
   b. The efficiency of linear search is O(N).
   c. The efficiency of binary search algorithms is O ($\log_2 N$).
   d. All of the above.

8. Which of the following searching techniques is appropriate only for arrays in a particular order?
   a. Linear search
   b. Binary search
   c. Hashing
   d. Tree search

9. Which of the following statement is correct?
   a. Hash function generates an integer number.
   b. Hash key can be used as an array index.
   c. Hashing avoids number of comparisons and goes directly where the required data is present.
   d. Hashing searches the element in a constant search time O(1), no matter where the element is located in the list.
   e. All of the above.

10. The sequential searching in the hash table is performed to find the empty position in
    a. linear probing.
    b. quadratic probing.
    c. chaining.
    d. none of the above.

11. Which of the following is not a of hash function?
    a. Division method
    b. Folding method
    c. Chaining method
    d. Mid-square method

12. Which of the following generates a random sequence of positions instead of a linear sequence?
    a. Linear probing.
    b. Quadratic probing.
    c. Chaining.
    d. Mid-square method.

13. Which of the following statement is correct?
    a. Hash table can be searched in O(1) time using a hash function.
    b. When two or more key values hash to the same location in an array, such a situation is called collision.
    c. Quadratic probing does not suffer much clustering as compared to linear probing.
    d. All of the above.

14. Which of the following requires the time to square the probe number?
    a. Linear probing
    b. Quadratic probing
    c. Chaining
    d. Mid-square method

15. Which of the following suffers from the clustering problem?
    a. Linear probing
    b. Quadratic probing
    c. Both (a) and (b)
    d. None of the above

16. Which of the following searching is not good for a linked list?
    a. Linear search
    b. Binary search
    c. Hashing
    d. Tree search

17. An element with key 1234 is to be inserted in the hash table. If the quadratic probing

collision resolution technique is used, and first two locations are already occupied then the next empty location that will be tried is
a. 35
b. 36
c. 34
d. 38

18. Which of the following properties is not expected from a good hash technique?
a. It should be very easy and quick to compute.
b. It should produce a relatively random and unique distribution of values within the hash table.
c. It should be easy to program.
d. It should produce collision.

19. Which of the following properties are expected from a good hash technique?
a. It may be formed by using some mathematical transformation.
b. It should be very easy and quick to compute.
c. It should produce as few collisions as possible.
d. All of the above.

20. Which of the following uses linked list?
a. Binary search
b. Chaining
c. Linear probing
d. Quadratic probing

21. What is not true for linear collision processing?
a. It is quick and easy to implement.
b. It may maximizes collision.
c. It requires space for links.
d. It requires sequential search.

22. Which of the followings are the applications of hashing?
a. It can be used as a symbol table.
b. It can be used for online spelling checkers.
c. If no ordering information is required then hashing is a choice of data structures.
d. All of the above.

23. The hash function `Hash(key) = key % 100` (mod) will produce the following range of values.
a. 0–99
b. 0–100
c. 1–99
d. 1–100

24. The technique of linear probing for collision resolution can lead to
a. clustering.
b. overflow.
c. chaining.
d. all of the above.

25. Which of the following statements are correct?
a. Hashing performs insertions, deletions and searching in constant average time.
b. A separate chaining technique of hashing is a popular and space-efficient alternative to quadratic probing.
c. Compilers use hash tables called symbol table to keep track of declared variables in the source code.
d. All of the above.

26. Which of the following statements are correct?
a. A perfect hash function maps each key to a distinct integer within some manageable range.
b. Every element of the hash table is a pointer to a list in chaining.
c. Search for a desired element takes longer time in linear probing.
d. All of the above.

Review Questions

1. What is meant by collision in a hashing scheme? What are ways of resolving collisions?
2. What is separate chaining? Explain with an example.
3. What is hashing? Explain any one method for collision resolution.
4. What are the properties of a good hash function?
5. Explain why we require the table size to be a prime number in double hashing?
6. The following values are to be stored in a hash table 23, 47, 89, 78, 65, 45, 39, 14, 19. Use the division method of hashing with a table size of 10 and use the linear probing for resolving the collision.
7. Define the following terms:
   a. Hash function.
   b. Double hashing.
   c. Linear probing.
8. Explain the quadratic probing with an example.
9. What do you mean by collision? How it can be resolved?
10. What is searching? Explain various searching methods.
11. What is clustering in a hash table? Describe various methods of collision resolution.
12. Write down the binary search algorithms and obtain the worst-case and average-case complexities.
13. Write down the recursive and non-recursive algorithms for binary search.
14. What is the preliminary requirement to perform binary searching?
15. Write short notes on hashing.

Programming Assignments

1. Write a program in C/C++ to implement linear probing and quadratic probing.
2. Write a program in C/C++ to implement recursive binary search.
3. Write a menu-driven program in C/C++ to implement the hashing techniques.
4. Write a menu-driven program in C/C++ to select the search technique and then perform the searching on the list of given numbers.

Answers

Multiple-Choice Questions

1. (c)
2. (a)
3. (d)
4. (d)
5. (b)
6. (c)
7. (d)
8. (b)
9. (e)
10. (a)
11. (c)
12. (b)
13. (d)
14. (b)
15. (c)

16. (b)
17. (d)
18. (d)
19. (d)
20. (b)
21. (c)
22. (d)
23. (a)
24. (a)
25. (d)
26. (d)

10 Sorting Algorithms

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Importance and use of sorting.
- Different sorting techniques.
- Implementation of different sorting algorithms.
- Analysis and time complexities of different sorting algorithms.

We have studied many data structures to store the data and process them. Many business applications require data in a particular order in either ascending order or descending order. When data are arranged from the smallest to the largest values, they are in ascending order and when the data are arranged from the largest to the smallest values, they are in descending order. Retrieval and searching of data item becomes easier when the list of data item is sorted in some predefined order. There are various sorting methods which arrange the data in a particular order according to the predefined ordering criteria, either in ascending or descending. We should remember that no single sorting algorithm is best for all applications. The sorting algorithms can be broadly classified into two categories:

1. **Internal sorting:** If the elements which are to be sorted are in main memory then the sorting is called internal sorting. It is applied when the list of elements to sort is small so that the sorting can be carried out in main memory.
2. **External sorting:** If the elements to be sorted are in the secondary memory, for example, disks or tapes, then the sorting is called external sorting. It is applied to a large file of records so it cannot be carried out in the main memory. Therefore, it is performed using secondary storage. External sorting depends on the type of device (e.g., disks, tapes, etc.) and the number of such devices that can be used at a time. There are different ways of sorting like bubble sort, insertion sort, selection sort, quick sort, heap sort, merge sort, radix sort. We will discuss each of them in detail in the next section.

There are certain sorting that fall in both categories, for example, merge sort, selection sort, tree sort. The performance of the sorting algorithms depends on many factors such as:

1. Space required for the number of the elements to be sorted. This is called the space complexity of the sorting algorithm.
2. Run time for execution of an algorithm. This is generally based on the comparison made during the execution of the sorting algorithms. This is called the time complexity of a sorting algorithm. The sorting time or the time complexity is denoted by the Big Oh (O) notation. The complexity is said to be O(n), that is the order of n when it is proportional to n, where n is the number of elements in the list to be sorted. Therefore O(n) = Kn, where K is a positive constant.

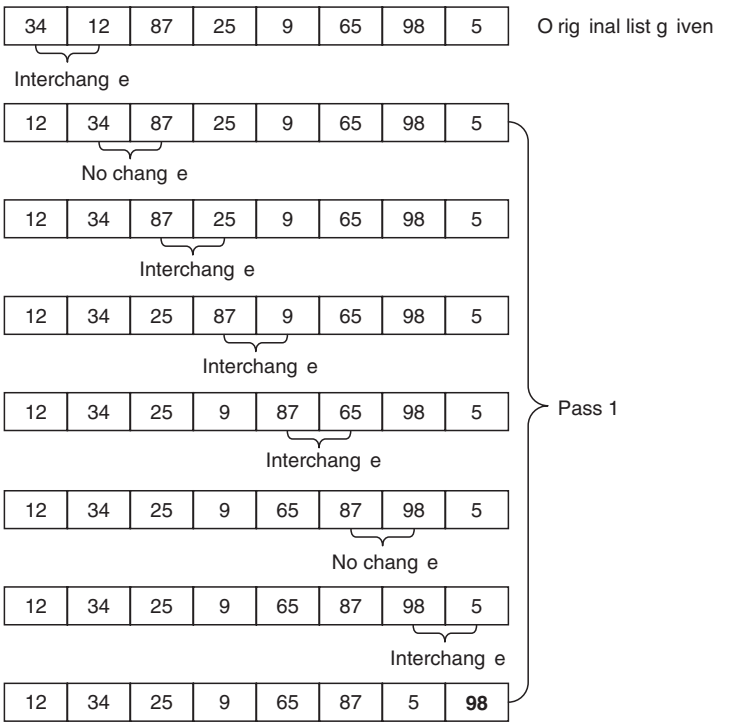
We can make a wise choice of a sorting algorithm on certain efficiency considerations. For calculating the efficiency of sorting methods, we have the following criteria:

1. **Best case:** If the elements are already in the required sorted order then this is called the best case for a sorting algorithm.
2. **Worst case:** If the elements are in reverse order than the required order, this is worst case of the sorting algorithm. In other words, if the list is already in ascending order and we want to sort the list in descending order, this is the worst case for a sorting algorithm.
3. **Average case:** In this case the elements are scattered in the list.

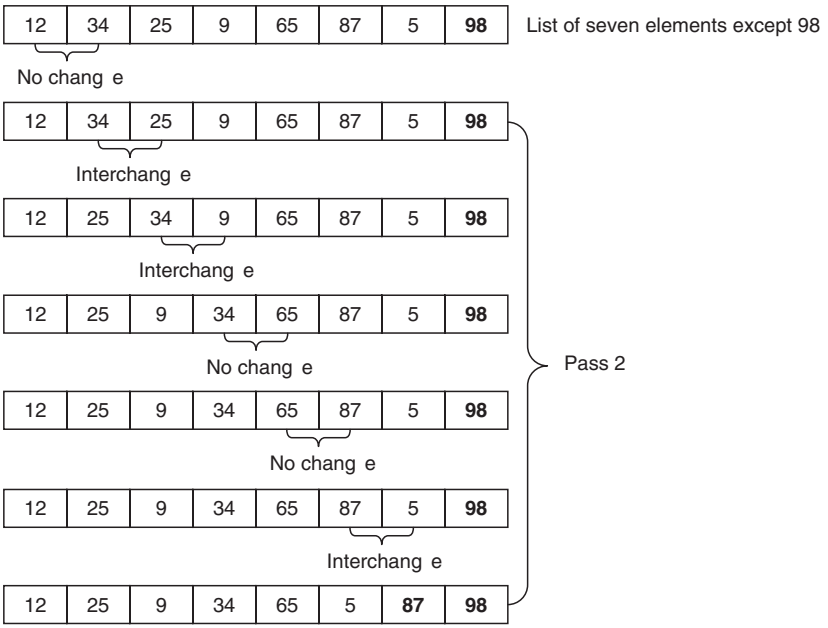
10.1 Bubble Sort

We start with a sorting algorithm that is very easy to understand and implement, the bubble sort method. In this method multiple swapping take place when we pass through the list of elements sequentially. In each pass we compare the element with its successor and interchange if they are not in a proper order. Therefore interchange is just like bubbling up the elements in the list and hence the name of the algorithm. The proper order depends on whether we want to arrange the elements in ascending order or descending order. Let us take the case of arranging the elements in ascending order. We interchange the element with its successor only if the element is greater than its successor. This process is carried on till the list is sorted.

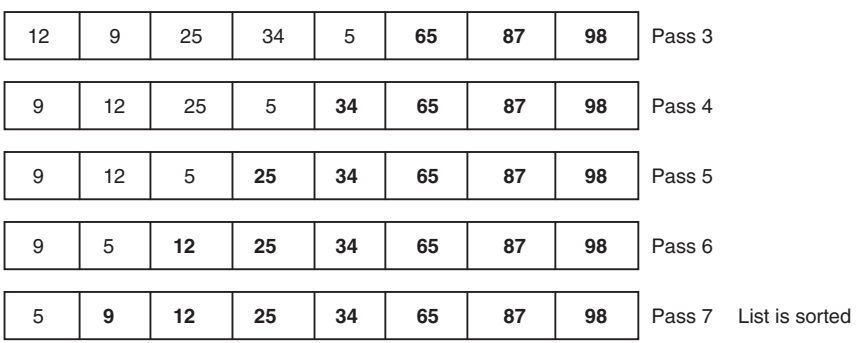
Let us take an example of list A of eight elements 34, 12, 87, 25 9, 65, 98, 5. We sort it in ascending order. In the first pass we compare A[i] and A[$i + 1$] for $i = 0, 1, 2, \ldots, 7$ and interchange the element A[i] and its successor A[$i + 1$] if A[i] > A[$i + 1$]. The process is shown below.



We can see that after the first pass the greatest element (98) of the list has found its proper place. The number of comparisons during the first pass that has taken place is seven (8 – 1) since there are eight elements in the list. In general A[$N - K$] will be in its proper position after K passes. Now the second pass is required for the list of remaining seven elements. Therefore we compare A[i] and A[$i + 1$] for $i = 0, 1, 2, \ldots, 6$ and interchange A[i] and A[$i + 1$] if A[i] > A[$i + 1$]. The process is shown below.



We can see that after the second pass the second largest element (87) of the list has found its proper place. The number of comparisons during the second pass that has taken place is six (7 – 1) since there are seven elements in the list or 8 – 2 = 6 according to $N - K$. We repeat the same process until we get the sorted list.



Algorithm 1 Bubble Sort (A, N)

This algorithm sorts the array A (list) of N elements using bubble sort.

1. Perform Step 2 for I=0 to N− 2
2. Perform Step 3 for J=0 to N−I− 2
3. If $jA_j > A_{j+1}$
   Set $Temp = A[j]$
   $j]=A_{j+1}$
   $j]=Temp$
   End if
   End and repeat Step 2
   End and repeat Step 1
4. Exit

Analysis of Bubble Sort

Here we can see that the number of passes required to sort the list of eight elements are seven, hence $N-1$ passes are required to sort the list of N elements using bubble sort. We have also seen that each pass requires $N-1$ comparisons where N is the number of elements in the list to be sorted. Therefore the first pass requires $8-1$ comparisons, the second pass requires $7-1$ comparisons, the third pass requires $6-1$ comparisons and so on. Hence the total number of comparisons required are $(N-1)\times(N-1) = N^2 - 2N + 1 = O(N^2)$. Bubble sort method of sorting is not considered to be an efficient one because it takes significant amount of time and memory in interchanging the elements and for storing temporary element, respectively. There are three cases to judge the efficiency.

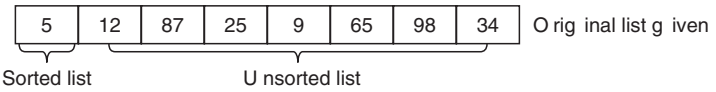
1. **Best case:** If the array is already sorted, no interchange will take place although we have to go through the $N-1$ passes. Therefore the complexity in the best case is $O(N-1) = O(N)$.
2. **Average case:** If the elements are scattered in the list, we need to interchange the elements during comparisons. We assume that on an average there are $(N-1)/2$ interchanges and we have to go through $N-1$ passes to sort the list. Therefore the average case complexity is $(N-1)\times(N-1)/2 = (N^2 - 2N + 1)/2 = O(N^2)$.
3. **Worse case:** When the list is in the reverse order we require $N-1$ interchanges during comparisons and we have to go through $N-1$ passes to sort the list. So the worst case complexity is $(N-1)\times(N-1) = N^2 - 2N + 1 = O(N^2)$.

10.2 Insertion Sort

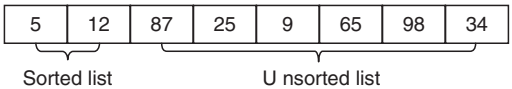
In this method, the element is inserted at its proper position in the sorted list. This method is used by the bridge players as they pick up the card and insert into the proper position amongst the cards already in their hand. In this method initially the whole list is divided into two parts, one part containing the first element and the second containing the other elements. The first part is called the sorted list and the second part is called the unsorted list. In other words, we have the list divided into two parts: sorted list and unsorted list. We start with the first element of unsorted list (the second element of the list) and insert it into its proper position in the sorted list. Now we have two elements in the sorted list and remaining elements in the unsorted list. Select the third element of the unsorted list and insert it at its proper position in the sorted list. In each pass of insertion the element from the unsorted array is inserted at the proper position into the sorted array. In this method the elements are inserted by shifting the larger elements one

slot down. This process is carried on till the list is sorted. If we want to sort the list in descending order then we have to look for the greatest element in the list and insert it into the first position. So the whole process is reversed. Let us take the example of list A of eight elements 5, 12, 87, 25, 9, 65, 98, 34, so sort it in ascending order. Here the first element (5) is in the first list, so it is itself trivially sorted. We take the second element (12) of the list and insert it either before the first element or after the first element so that first and second elements are in sorted order. Now we repeat the process with the third element so that first, second and third are in sorted order and so on. The complete process is given below.

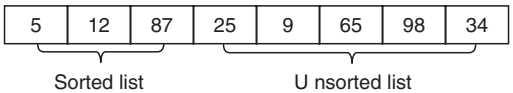
Step 1: Initially we take the first element of the list into sorted list so that now the given list is divided into two parts: Sorted and unsorted.



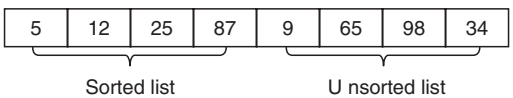
Step 2: Now take the first element (12) of the unsorted list and insert it at its proper position in the sorted list. Since 12 is greater than 5 so no insertion is required and it will remain in its place. Now we have two elements (5, 12) in the sorted list and remaining elements in the unsorted list as shown below.



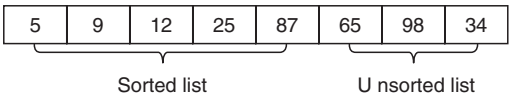
Step 3: Now take the next element 87 of the unsorted list and insert it into the proper position in the sorted list. Since it is greater than 12 so it will remain as it is. Now we have a sorted list of three elements and unsorted list of other elements as shown below.



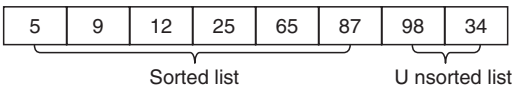
Step 4: Now take the next element (25) of the unsorted list and insert it into the proper position in the sorted list. Since it is less than 87, so 87 is shifted towards the right. Now compare 25 with next element (12) of the sorted list. Since 25 is greater than 12, it is inserted towards the right of 12. We have a sorted list of four elements and unsorted list of other elements as shown below.



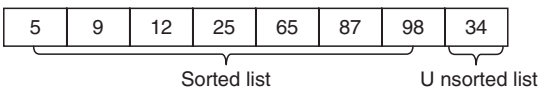
Step 5: Now take the next element (9) of the unsorted list and insert it at the proper position in the sorted list. Since it is less than 87, so 87 is shifted towards the right. Now compare 9 with next element (25) of the sorted list. Since it is more than 9, so 25 is also shifted towards the right. Finally compare 9 with the next element (12) of the sorted list. Again since 9 is less than 12, so 12 is shifted towards its right. Now compare 9 with the next element (5) of the sorted list. Since 9 is greater than 5, insert 9 towards the right of 5. We now have a sorted list of five elements and unsorted list of other elements as shown below.



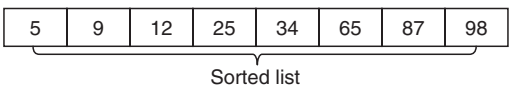
Step 6: Now repeat the same process for the next element 65 of unsorted list. Now we have a sorted list of six elements and an unsorted list of other elements as shown below.



Step 7: Now repeat the same process for the next element 98 of the unsorted list. We now have sorted list of seven elements and unsorted list of other elements as shown below.



Step 8: Repeat the same process for the last element 34 of the unsorted list. Now we have a sorted list of all the eight elements as shown below.



We can show the complete process as shown below.

Pass 1: A[1] is inserted after A[0] so that A[0] and A[1] are sorted.
Pass 2: A[2] is inserted before, in between or after A[0] and A[1] so that A[0], A[1] and A[2] are sorted.
Pass 3: A[3] is inserted before, in between or after A[0] and A[2] so that A[0], A[1], A[2] and A[3] are sorted.
:::
:::
:::
Pass *N* – 1: A[*N* – 1] is inserted before, in between or after A[0] and A[*N* – 2] so that A[0], A[1], A[2], A[3], …A[*N* – 1] are sorted.

Original list A is given

	5	12	87	25	9	65	98	34
Pass 1	**5**	**12**	87	25	9	65	98	34
Pass 2	**5**	**12**	**87**	25	9	65	98	34
Pass 3	**5**	**12**	**25**	**87**	9	65	98	34
Pass 4	**5**	**9**	**12**	**25**	**87**	65	98	34
Pass 5	**5**	**9**	**12**	**25**	**65**	**87**	98	34
Pass 6	**5**	**9**	**12**	**25**	**65**	**87**	**98**	34
Pass 7	**5**	**9**	**12**	**25**	**34**	**65**	**87**	**98**

The elements in the sorted list are shown in bold and the shifting is shown by the arrow line.

Algorithm 2 Insertion Sort (A, N)

This algorithm sorts the array A (list) of N elements using insertion sort. We are taking the first element at 0th location of the array A.

1. Set I=1 .
2. Repeat Steps 3, 4, 5 and 6 while I<=N
3. Set Temp=A[I]
4. Set J=I - 1
5. Repeat Steps a and b while Temp<A[J]
   a. Set A[J+1]=A[J]
   b. J=J - 1
   End and repeat Step 5
6. A[J+1]=Temp
7. Set I=I+1
   End and repeat Step 2
8. Exit

Analysis of Insertion Sort

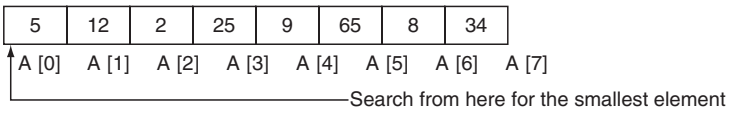
Here we can see that the number of passes required to sort the list of eight elements are seven, hence $N - 1$ passes are required to sort the list of N elements using insertion sort. Insertion sort is considered to be useful for a small collection of data. Insertion sort takes fewer comparisons than the bubble sort, so it will normally take less time than the bubble sort. Insertion sort is very efficient for a small list of elements and it is slow when numbers of elements in a list are very large.

1. **Best case:** If the array is already sorted, no interchange will take place although we have to go through $N - 1$ passes. Therefore the complexity in the best case is $O(N - 1) = O(N)$. Therefore when the input data contains sorted list of elements then the insertion sort works slightly faster and so it is highly efficient.
2. **Average case:** If the elements are scattered in the list, we need to interchange them during comparisons. We assume that on an average there are $(N - 1)/2$ interchanges and we have to go through $N - 1$ passes to sort the list. Therefore the average case complexity is $(N - 1) \times (N - 1)/2 = (N^2 - 2N + 1)/2 = O(N^2)$.
3. **Worse case:** When the list is in the reverse order then each element inserted will be compared with previous $N - 1$ elements in the sorted list and we have to go through the $N - 1$ passes to sort the list. So the worst case complexity is $(N - 1) \times (N - 1) = N^2 - 2N + 1 = O(N^2)$.

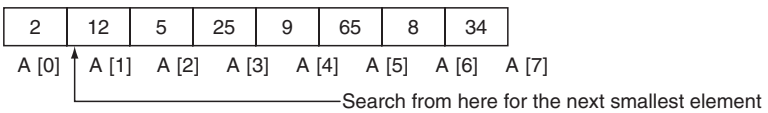
10.3 Selection Sort

In the selection sort we search for the smallest element in the list and interchange it with the element in the first (0th) position. Then we search for the second smallest element in the list and interchange it with the element in the second (1st) position. In other words, we can say that the smallest element in the list is placed at the first position. Then we look for the smallest element in the list of remaining elements (except 1st element). We place this element at the 2nd position of the array and so on.

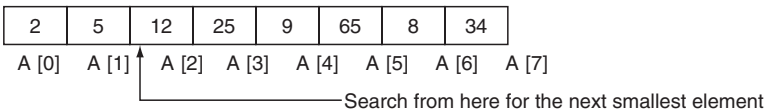
Therefore, in this algorithm element is selected then placed into the successive positions starting from the first (0th) position. This process is continued till the entire array is sorted. Let us take a list A of eight elements given as 5, 12, 2, 25, 9, 65, 8, 34.



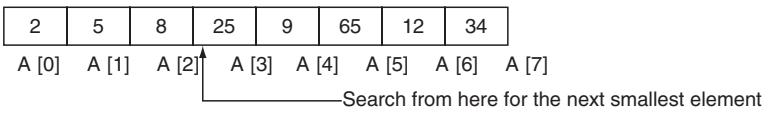
Pass 1: Find the position of the smallest element in the array. We can see that the smallest element is at position 2 in the list. Interchange the smallest element available at position 2 with the element at the first (0th) position. The resultant list is shown below.



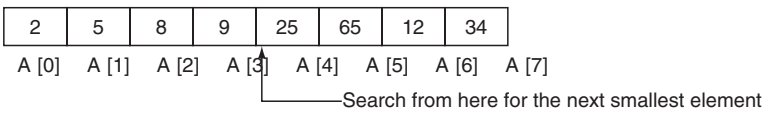
Pass 2: Now find the position of the smallest element in the sublists starting from A[1]. The smallest (second) element is found at position 2, so interchange the element at position 2 with the element at position 1. The resultant list is shown below.



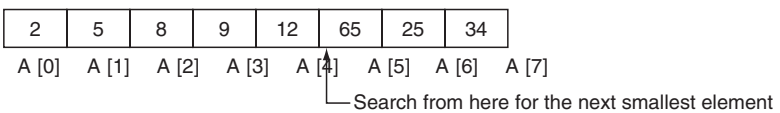
Pass 3: Now find the position of the smallest element in the sublists starting from A[2]. The smallest (third) element is found at position 6. Interchange the element at position 6 with the element at position 2. The resultant list is shown below.



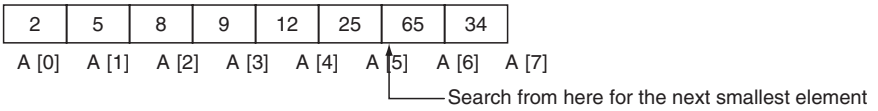
Pass 4: The next smallest (fourth) element is found at position 4. Interchange the element at position 4 with the element at position 3. The resultant list is shown below.



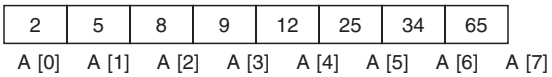
Pass 5: The next smallest (fifth) element is found at position 6. Interchange the element at position 6 with the element at position 4. The resultant list is shown below.



Pass 6: The next smallest (sixth) element is found at position 6. Interchange the element at position 6 with the element at position 5. The resultant list is shown below.



Pass 7: The next smallest (seventh) element is found at position 7. Interchange the element at position 7 with the element at position 6. The resultant list is shown below.



Now the list is sorted. The list contains eight elements so seven passes are required to sort the list.

Algorithm 3 Selection Sort (A, N)

This algorithm sorts the array A (list) of *N* elements using selection sort. We are assuming that the first element is at the 0th location of array A.

1. Set I=1
2. Repeat Steps 3to 8 while I<=N
3. Set Min=A [I]and Flag=0
4. Set J=I+1
5. Repeat Step 6 while J<=N
6. If (A<Min]n)
   a. Set Min=A[nJ]
   b. Pos=J
   c. Flag=1
   End if
7. Set J=J+1
   End and repeat Step 5
8. If Flag=1
   a. Set Temp=A[n]
   b. A [I]A][Pos]
   c. A [Pos=Temp
   End if
9. Set I=I+1
   End and repeat Step 2
10. Exit

Analysis of Selection Sort

Here we can see that seven passes are required to sort a list of eight elements, hence $N - 1$ passes are required to sort a list of N elements using selection sort. Selection sort is considered to be useful for small collection of data. Selection sort takes fewer comparisons than the bubble sort, so it will normally take less time than the bubble sort. Selection sort is more efficient than the bubble sort and the insertion sort because it does not require excessive swapping and it has the same complexity as the bubble sort or the insertion sort.

1. **Best case:** If the array is already sorted no interchange will take place although we have to go through $N - 1$ passes. Therefore the complexity in the best case is $O(N - 1) = O(N)$. So when the input data contains sorted list of elements then the selection sort works slightly faster than the bubble sort.

2. **Average case:** If the elements are scattered in the list we need to interchange the elements during comparisons. We assume that on an average there are $(N - 1)/2$ interchanges and we have to go through $N - 1$ passes to sort the list. The average case complexity is $(N - 1) \times (N - 1)/2 = (N^2 - 2N + 1)/2 = O(N^2)$.
3. **Worse case:** When the list is in the reverse order then there are $N - 1$ comparisons to find the first smallest element during pass 1. Similarly, there are $N - 2$ comparisons to find the second smallest element in the list and so on. We have to go through $N - 1$ passes to sort the list. So the worst case complexity is $(N - 1) + (N - 2) + (N - 3) + \ldots + 3 + 2 + 1 = N \times (N - 1)/2 = (N^2 - N)/2 = O(N^2)$.

Program 1 A menu-driven program to implement bubble sort, insertion sort and selection sort in C+ +using template.

```
#include<iostream.h>
#include<conio.h>
#define MAX 5
template<class T>
class Sorting
{
T List[MAX];
public:
void Readdata()
{
cout<<"***Enter elements of the list to sort
for(i=0;i<MAX;i++)
cin>>List[i];
}
void BSORT()
{
cout<<"Original list is";
Display();
for(i=0;i<MAX-1;i++)
for(i=0;j<MAX-i-1;j
if(List[j]>List[j+1])
{
T temp=List[j];
List[j]=List[j+1];
List[j+1]=temp;
}
}
Display()
{
for(i=0;i<MAX;i++)
cout<<"List["<<i<<"]"<<"List["<<i<<"];
}
```

```
void ISORT()
{
cout<<"Original list is :";
Display();
for(int i=1;i<MAX;i++)
{
T temp=list[i];
for(int j=i-1;j>=0 && List>temp[j];j--)
List[j+1]=List[j];
List[j+1]=temp;
}
}
void SSORT()
{
int flag=0,Pos;
cout<<"Original list is :";
Display();
for(int i=0;i<MAX-1;i++)
{
T MIN=list[0];
for(int k=i+1;k<MAX;k++)
{
if(List<list[k])
{
MIN=List[k];
Pos=k;
flag=1;
}
}
if(flag==1)
{
T temp=list[i];
List[i]=list[Pos];
List[Pos]=temp;
}
}
}
};
void main()
{
clrscr();
Sortstring obj;
int ch;
obj.Readdata();
do
```

```
{
clrscr();
cout<<"Enter your choice for sorting";
cout<<"1. Bubble sort";
cout<<"2. Insertion sort";
cout<<"3. Selection sort";
cout<<"4. Want to change the list";
cin>>ch;
switch(ch)
{
case 1:
obj.BSORT();
cout<<"The sorted list is";
obj.Display();
getch();
break;
case 2:
obj.ISORT();
cout<<"The sorted list is";
obj.Display();
getch();
break;
case 3:
obj.SSORT();
cout<<"The sorted list is";
obj.Display();
getch();
break;
case 4:
obj.Readdata();
break;
}
}while(ch!=0);
}
```

Output

```
Enter your choice for sorting
1. Bubble sort
2. Insertion sort
3. Selection sort
4. Want to change the list
1
Original list is
List[0] 4667
List[1] 12
```

```
List[2]  5 5 6
List[3]  8 9
List[4]  2 4
The  sorted  list  is
List[0]  1 2
List[1]  2 4
List[2]  8 9
List[3]  5 5 6
List[4]  4 6 6 7
```

10.4 Quick Sort

The quick sort algorithm is the most popular and widely used sorting algorithm developed by C.A.R. Hoare in 1962. This algorithm tends to be the quickest in the average case among all the sorting techniques. It works on the divide-and-conquer strategy. In divide-and-conquer policy, the problem is divided into smaller subproblems repeatedly until we reach the smallest size subproblem whose solution is easy to find. Then solutions of these small subproblems are combined to obtain the solution of the whole problem.

The quick sort algorithm selects the key element generally called pivot and then it performs the "partition" of the list into two sublists such that all the elements smaller than the pivot element are kept in one sublist which is placed before the pivot element, and all the elements greater than the pivot elements are kept in another sublist which is placed after the pivot element. So after choosing the key element, partitioning is performed; the elements are grouped into two sublists:

1. One list contains the elements that are lesser than the key value.
2. Another list contains the elements that are greater than the key value.

The quick sort algorithm is then performed individually on the sublists. The same procedure continues until the list has been sorted. The key value can be any element of the list; however, usually the middle element is selected as the key element. The quick sort algorithm is used on each sublist either through iteration or through recursion. Then the sublists are put together and the whole list will be in order. The algorithm starts from each end of the lists. It starts searching from the right end for the first element smaller than the key element and from the left end for the first element greater than the key element. It swaps these two values and continues the process until the left position value and right position value meet somewhere in the center. When this algorithm ends, the entire list is divided in two sublists as explained earlier. The pictorial representation of quick sort algorithm is shown in Figure 1.

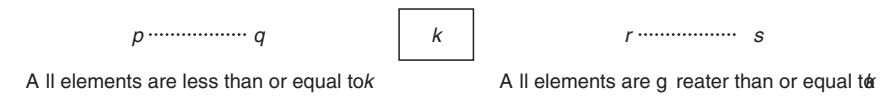
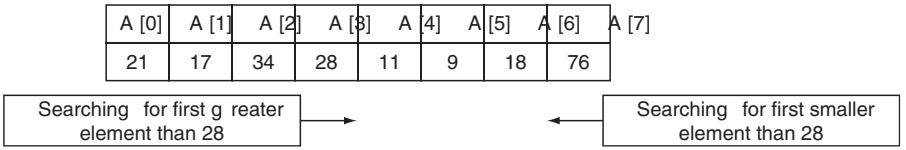
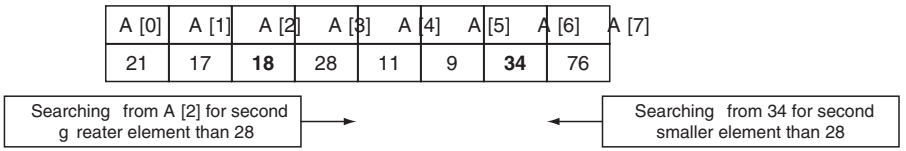


Figure 1 Pictorial representation of quick sort algorithm.

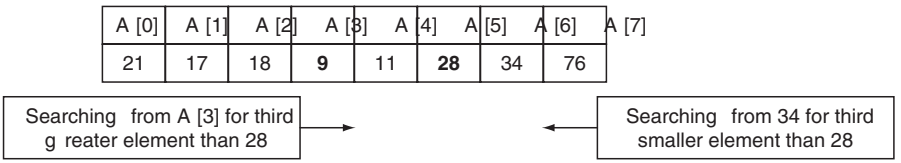
A quick sort algorithm is very good for average case but it does not perform well for the worst case. This algorithm can be better explained by taking an example. Let us take an array A of eight elements with the pivot element as 28.



1. The first larger or equal element found is 34 at A[2] when scanning from left to right and the first smaller element found is 18 at A[6] when scanning from right to left. So interchanging these two values results in



2. The second larger or equal element than 28 is found at A[3], that is 28 itself when scanning from left to right starting from A[2] and the smaller element than 28 is found at A[5], that is, 9 when scanning from right to left starting from A[6]. Interchanging them results in



3. The third larger or equal element than 28 is found at A[5], that is, 28 itself and the smaller element than 28 is not found before 28 (i.e., A[5]), so 28 has got its position at A[5]. Now we have two sublists, one to the left of 28 and another to the right of 28 as shown below.

First sublist						S econd sublist	
21	17	18	9	11	**28**	34	76

Here we can see that the first sublist contains elements smaller than 28 and the second sublist contains elements greater than 28.

4. Now the same procedure is applied to each sublist as shown below.

21 17 18 9 11 [28] 34 76

Let 18 be the pivot element for the first sublist.

11 17 18 9 21 [28] 34 76

11 17 9 18 21 [28] 34 76

Now **18** has got its position:

11 17 [9] 18 21 [28] 34 76

The first sublist is divided into two sublists. Now apply the same procedure to second sublist. The quick sort algorithm is as follows.

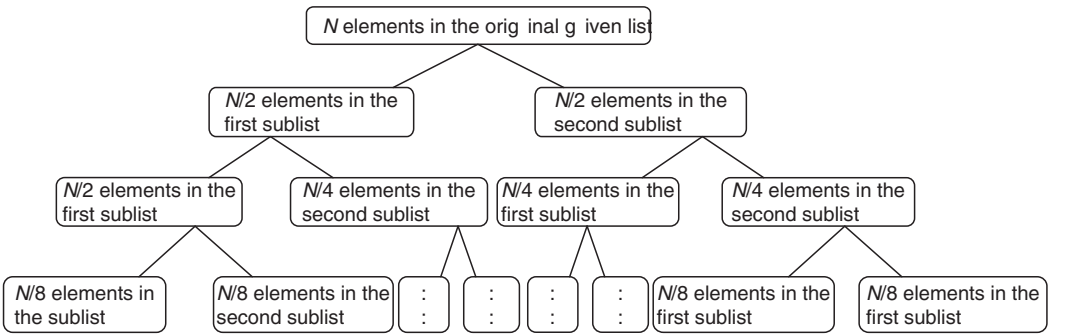
Algorithm 4 Quick Sort (A, Start, End)

Here array A is a collection of element to sort using quick sort. Start is the starting index and End is the last index of the array A.

1. Set Left = Start
   Set Right = End
   Key = A [Left + Right] / 2
2. Repeat Steps 3 to 7 while (Left < Right)
3. Repeat Step 4 while (A [Left] < Key)
4. Set Left = Left + 1 (increment the left pointer)
   End and repeat Step 3
5. Repeat Step 6 while (A [Right] > Key)
6. Right = Right – 1 (decrement the right pointer).
   End and repeat Step 5
7. If (Left <= Right)
   (Interchanges the elements at A [Left] and A [Right])
   a. Set Temp = A [Left]
   b. A [Left] = A [Right]
   c. A [Right] = Temp
   d. Left = Left + 1
   e. Right = Right – 1
   End Step 7
   End and repeat Step 2
8. If (Start < Right)
   Quicksort (A, Start, Right)
   End Step 8
9. If (Left < End)
   Quicksort (A, Left, End)
   End Step 9
10. Exit

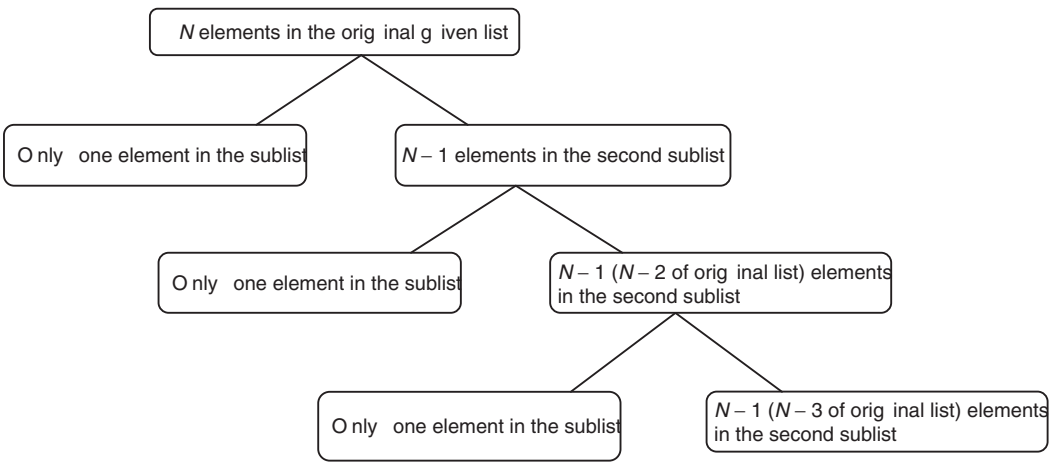
Efficiency of Quick Sort

1. **Best case:** If the array is already sorted, the original list will be divided into two sublists of almost equal size as shown below.



Therefore the complexity in the best case is O(N log N) = O(N). When the input data contains sorted list of elements then the selection sort works slightly faster than the bubble sort.

2. **Average case:** If the elements are scattered in the list we need to interchange the elements during comparisons. The average case complexity of quick sort is O(N log N).
3. **Worst case:** When the list is in the reverse order than the desired order, the quick sort algorithm will partition the list into two sublists such that one list will contain one element and the other list will contain $N - 1$ elements including the key element. The second list which contains $N - 1$ elements will again be split into two sublists with only one element in one list and $N - 1$ elements in the other (i.e., $N - 2$ of original list including the key element). The partitioning in each further step will follow the same trend as shown below.



We can see the unbalanced tree for the worst case of quick sort. The worst case complexity of a quick sort algorithm is O(N^2).

Program 2 A program in C++ to implement quick sort using template.

```
#include<iostream.h>
#include<conio.h>
#define MAX 5
template<class T>
class QSORTING
{
public:
QSORT(T List[],int start,int end)
{
int l=start;
int r=end;
T k=List[(left+right)/2];
do
```

```
{
while(List[key]<=List[left])
{
left++;
}
while(List[key]<List[right])
{
right--;
}
if(left<=right)
{
Temp=List[left];
List[left]=List[right];
List[right]=Temp;
left++;
right--;
}
}while(left<=right);
if(start<right)
QSORT(List,start,right);
if(left<end)
QSORT(List,left,end);
}
};
void main()
{
clrscr();
QSORT obj;
int List[MAX];
cout<<"***Enter the elements of a list***";
for(i=0;i<MAX;i++)
cin>>List[i];
obj.QSORT(List,0,4);
for(i=0;i<MAX;i++)
cout<<"List["<<i<<"]"<<List[i]<<endl;
getch();
}
```

Output

```
***Enter the elements of a list***
19 6 18 209 1
List[0] 1
List[1] 6
List[2] 18
List[3] 19
List[4] 209
```

10.5 Heap Sort

Before defining the heap sort, let us define a heap. A heap is a special kind of a tree with the following properties:

1. Each node of the tree has the value which is at least as large as the value of its children. This type of heap is called the max heap. The min heap can also be defined in the similar manner as the tree in which each node of the tree has a value that is at least as small as the value of its children.
2. In the case of max heap the root of the tree contains the maximum value and in the case of min heap the root of the tree contains the minimum value if all the values are distinct.
3. The elements are filled from left to right on the same level before moving to the next level, therefore the heap is usually defined as almost complete binary tree. Hence every leaf node is of depth D or $D - 1$.

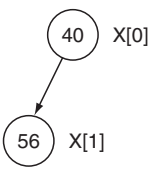
A max or min heap can be implemented using an array where if X[i] is the element stored at the root then its left child will be at X[$2 \times i + 1$] and right child will be at X[$2 \times i + 2$]. Here we must remember that the array index starts from 0 to Max – 1. The parent of a node present at index i can be found at $(i - 1)/2$.
The heap sort involves two steps:

1. **Construction of heap:** A heap can be constructed by inserting new elements one by one into an initially empty heap into the left – most open spot in the array. If the newly inserted element is greater than (in the case of max heap) its parent then swap their position recursively so that the newly inserted element rises to its appropriate place and preserves the heap order. We will see that the array index of the left child node is one more than twice of the array index of its immediate parent and the array index of a right child node is two more than twice the array index of its immediate parent node. Let us take the list of 40, 56, 28, 79, 20, 18, 67 and 56 to construct the max heap.

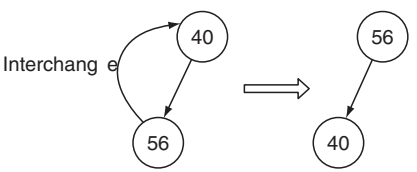
   Step 1: Take 40 as the root of the tree.

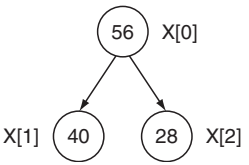
   Step 2: Now insert the next element of the list to the left of the root node because we have to insert the nodes from left to right at the same level of the binary tree.

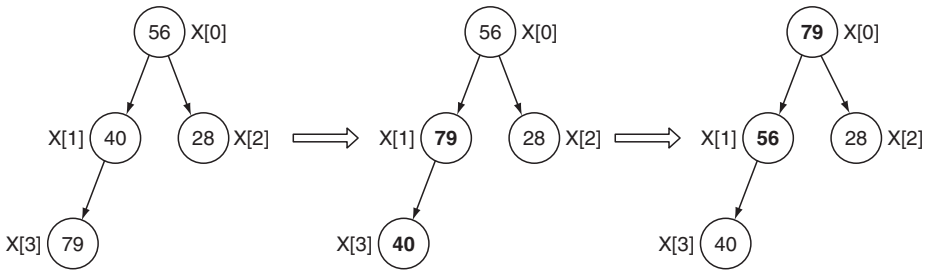
   This violates the max heap property so reheap the tree and it becomes the following:

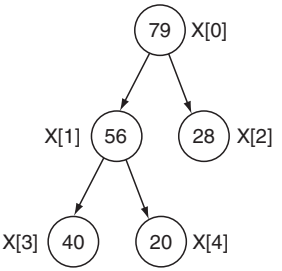
Step 3: Insert the next element 28 as the right child of the root (i.e., 56). It does not violate the heap property so interchange is not required.



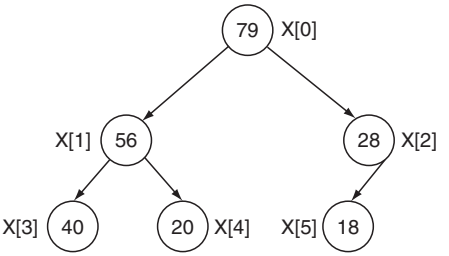
Step 4: Insert the next element 79 as the left child of 40. It violates the heap property, so interchanges are required which are shown in boldface.



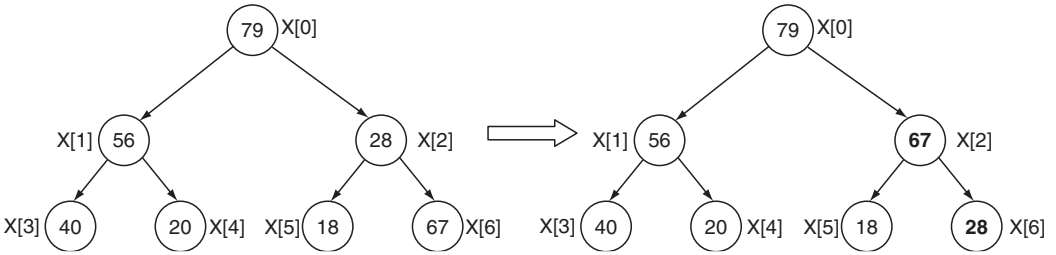
Step 5: Insert the next element 20 as the right child of the 56. It does not violate the heap property and so interchanges are not required.



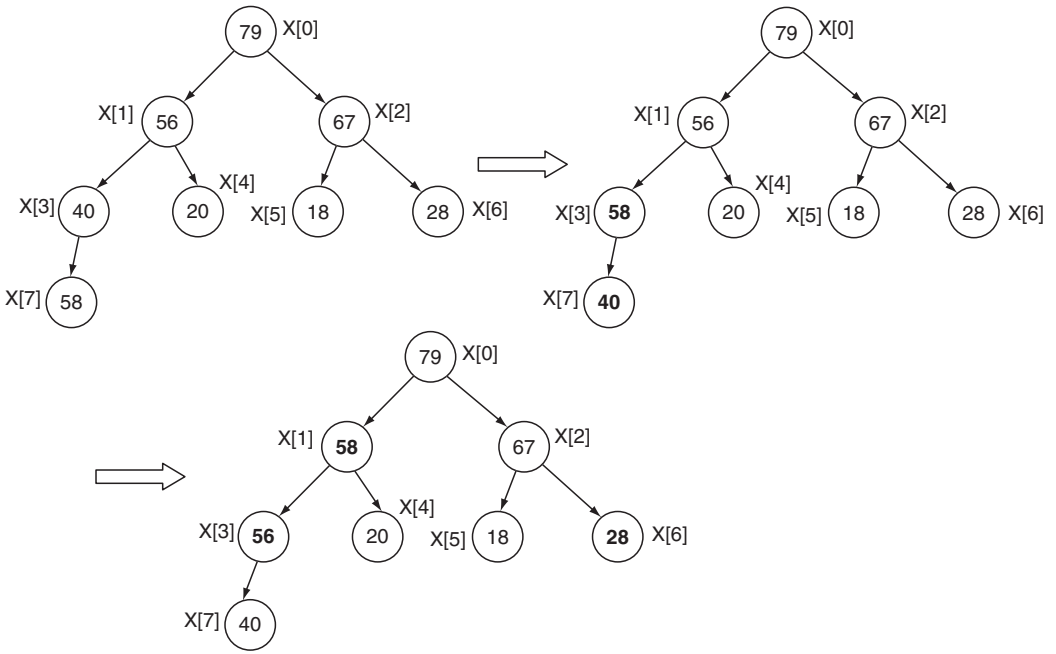
Step 6: Insert the next element 18 as the left child of the 28. It does not violate the heap property and so interchanges are not required.



Step 7: Insert the next element 67 as the right child of the 28. It violates the heap property and so interchanges are required which are shown in boldface.



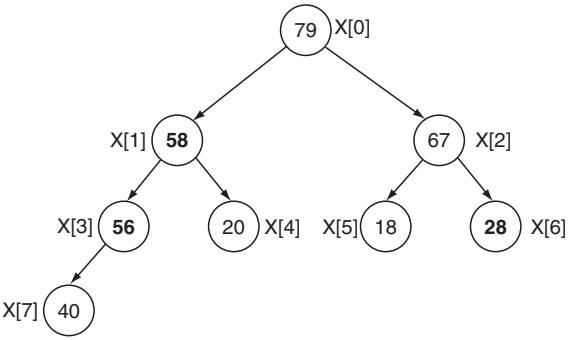
Step 8: Insert the next element 58 as the left child of the 40. It violates the heap property and so interchanges are required which are shown in boldface.



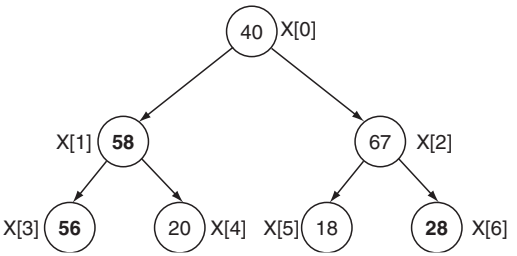
We can see here that the left child of X[0] is located at X[1] (i.e., X[$2 \times \boldsymbol{i} + 1$] where $\boldsymbol{i} = 0$) and the right child is located at X[2] (i.e., X[$2 \times i + 2$]). The same can be verified for $i = 1$, 2 and 3 as given below:

- The left child of X[**1**] is at X[3] (i.e., X[$2 \times \mathbf{1} + 1$]) and right child is at X[4] (i.e., X[$2 \times 1 + 2$]).
- The left child of X[**2**] is at X[5] (i.e., X[$2 \times \mathbf{2} + 1$]) and right child is at X[4] (i.e., X[$2 \times 1 + 2$]).
- The left child of X[**3**] is at X[7] (i.e., X[$2 \times \mathbf{3} + 1$]) and right child is not present.

2. **Sorting heap by repeatedly deleting the top of the heap:** We take another array of same size as of the heap tree to store the sorted elements. The root element is deleted and is placed at the 0th location of the new array. The root is replaced by the last leaf node of the heap and the heap is readjusted to maintain the properties of the heap. The new root element is compared with two of its children and interchanged with the largest of both of its children.

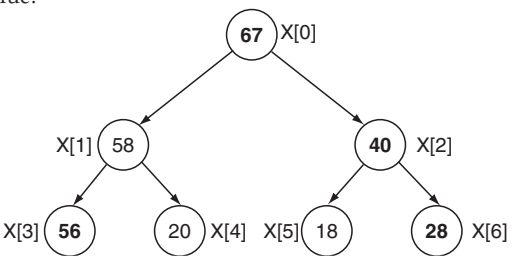


Step 1: Delete the value 79 at the root of the heap and place it into A[0]. Replace the root by the value 40 of the last leaf node and decrease the size of the array X by 1.

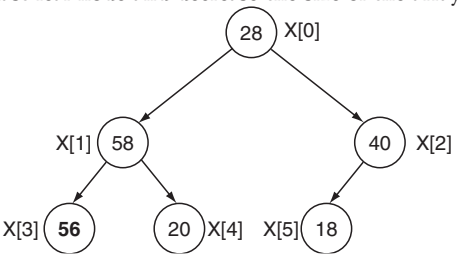


A	79							
Index	0	1	2	3	4	5	6	7

Step 2: Reheap the tree by comparing the root value with both its children and interchange it with the largest value.

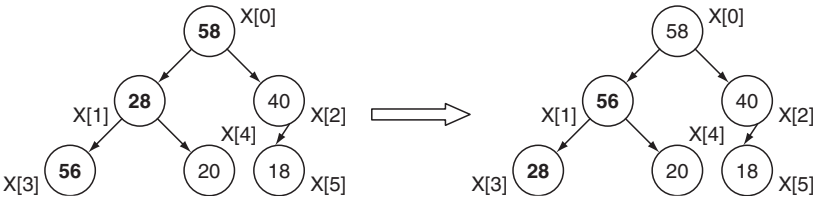


Step 3: Delete the value 67 at the root of the heap and place it into A[1]. Replace the root by the value 28 of the last leaf node and decrease the size of the array X by 1.

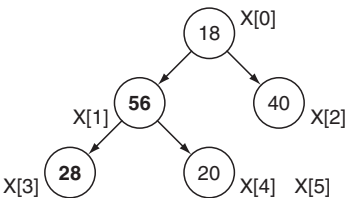


A	79	67						
Index	0	1	2	3	4	5	6	7

Step 4: Reheap the tree by comparing the root value with both its children and interchange it with the largest value (i.e., 58 again compare 28) with both its children and interchange with the largest value (i.e., 56).

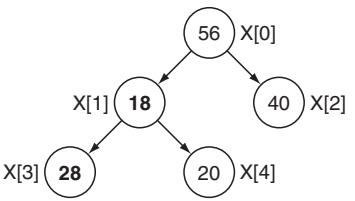


Step 5: Delete the value 58 at the root of the heap and place it into A[2]. Replace the root by the value 18 of the last leaf node and decrease the size of the array X by 1.

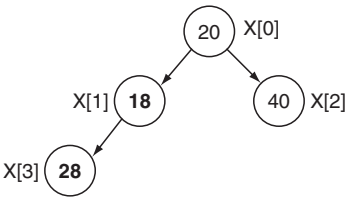


A	79	67	58					
Index	0	1	2	3	4	5	6	7

Step 6: Reheap the tree by comparing the root value with both its children and interchange it with the largest value (i.e., 56).

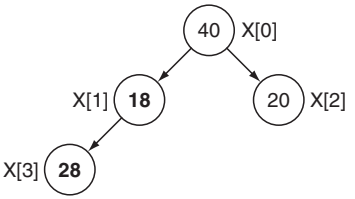


Step 7: Delete the value 56 at the root of the heap and place it at A[3]. Replace the root by the value 20 of the last leaf node and decrease the size of the array X by 1.

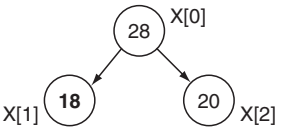


A	79	67	58	56				
Index	0	1	2	3	4	5	6	7

Step 8: Reheap the tree by comparing the root value 20 with both its children and interchange it with the largest value (i.e., 40).

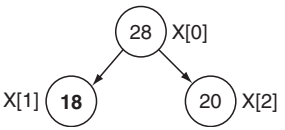


Step 9: Delete the value 40 at the root of the heap and place it at A[4]. Replace the root by the value 28 of the last leaf node and decrease the size of the array X by 1.

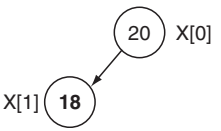


A	79	67	58	56	40			
Index	0	1	2	3	4	5	6	7

Step 10: Reheap the tree by comparing the root value 28 with both its children. The interchange is not required because the root contains larger value than its children.

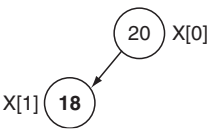


Step 11: Delete the value 28 at the root of the heap and place it at A[5]. Replace the root by the value 20 of the last leaf node and decrease the size of the array X by 1.



A	79	67	58	56	40	28		
Index	0	1	2	3	4	5	6	7

Step 12: Reheap the tree by comparing the root value 20 with its only remaining child. The interchange is not required because the root contains larger value than its child.



Step 13: Delete the value 20 at the root of the heap and place it at A[6]. Replace the root by the value 18 of the last leaf node and decrease the size of the array X by 1.

A	79	67	58	56	40	28	20	
Index	0	1	2	3	4	5	6	7

Step 14: Since only one node is there in the tree, place it in the next location A[7] of the array and delete the array X.

A	79	67	58	56	40	28	20	18
Index	0	1	2	3	4	5	6	7

The heap sort can be performed on the same array or without using the extra array. The root element is interchanged with the last element of the tree. Then reheap is performed and again the root element is interchanged with the second last element of the array and so on.

10.6 Merge Sort

This is a good example of external sorting where files are sorted using the external devices like disks or tapes. Merge sort algorithm is based on divide-and-conquer strategy same as the quick sort. The list of N elements is first divided into two sublists of $N/2$ elements. This phase is called the divide phase. Then each half is sorted independently; this phase is called the conquer phase. Then the two sorted halves are merged to a sorted sequence. The key operation of the merge sort is merging, therefore before going into the merge sort, let us first discuss the processing of merging.

Process of Merging

Merging means combining elements of two files to form a new file. In its simplest form merging is the process of combining elements of two arrays into a third array. The elements of the first array are copied into the third array, then the elements of second array are copied into the third array. The merging can be applied while sorting as follows. Let us assume that array A contains N elements and array B contains M elements. Then a third array of size $M + N$ is required. All the elements of array A that are less than a particular element B[j] of array B are copied into array C. The index pointers of array A and array C are incremented. However when the array element B[j] is smaller, it is copied into array C and scanning is continued in array B until some smaller element A[i] of array A is encountered. The process is repeated until either array A or array B is exhausted. In that case all the elements remaining either in A or in B are copied into the array C.

Example 1

Let us assume that array A[3] and B[4] are to be merged then new array C[9] is required.

A	8	2	16					
Index	0	1	2					
	↑							
B	12	3	7	14				
Index	0	1	2	3				
	↑							
C								
Index	0	1	2	3	4	5	6	
	↑							

Step 1: Compare A[0] with B[0]. Since A[0] < B[0], copy A[0] to C[0], move the pointer to next index value (i.e., 1) for A and C.

A	8	2	16				
Index	0	1	2				
B	12	3	7	14			
Index	0	1	2	3			
C	8						
Index	0	1	2	3	4	5	6

Step 2: Compare A[1] with B[0]. Since A[1] < B[0], copy A[1] to C[1], move the pointer to next index value (i.e., 2) for A and C.

A	8	2	16						
Index	0	1	2						
B	12	3	7	14					
Index	0	1	2	3					
C	8	2							
Index	0	1	2	3	4	5	6	7	8

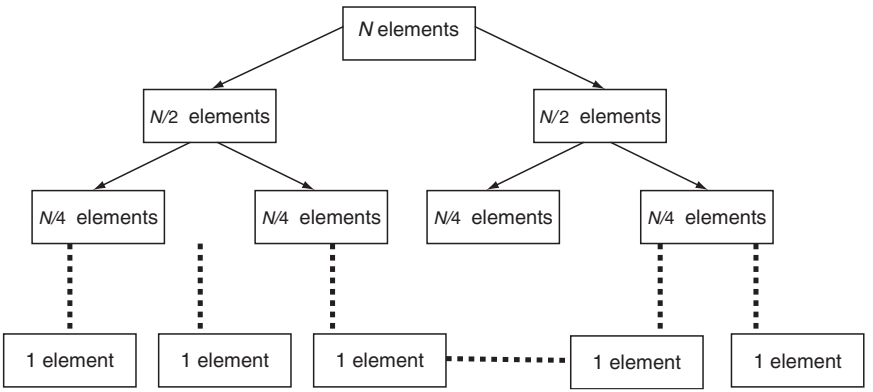
Step 3: Compare A[2] with B[0]. Since A[2] > B[0], copy B[0] to C[2], move the pointer to next index value for B and C as shown with an arrow.

A	8	2	16						
Index	0	1	2						
B	12	3	7	14					
Index	0	1	2	3					
C	8	2	12						
Index	0	1	2	3	4	5	6	7	8

Step 4: Continuing in the same way we will obtain the sorted list in array C.

Process of Merge Sort

The merge sort is a sorting algorithm that uses divide-and-conquer technique. The merge sort divides the given list into two sublists recursively until each sublist contains only one element. Then we apply the same merging process recursively as explained above to obtain the sorted list. If the single file or list contains N elements then it is divided into two sublists containing $N/2$ elements which are further divided into sublists containing $N/4$ elements and so on.



We now apply the same merge processing technique to adjacent sublist of one element taking two sublists at a time. We repeat this process until there is only one file remaining of size *n*. The algorithm is as follows.

Algorithm 5 MERGE(TEMP, LIST, SIZE1, SIZE2)

Here LIST is divided into sublists and these sublists are merged into the list TEMP

1. Set i=0, j=0, k=0
2. Repeat following while(i<SIZE2)
```
If (LIST[i]<=LIST[j] AND j<SIZE1) then
{
  Set TEMP[k]=LIST[i]
  Set i=i+1
  Set k=k+1
}
Else
{
  Set TEMP[k]=LIST[j]
  Set j=j+1
  Set k=k+1
}
```
3. Repeat Steps 4, 5 and 6 while(j<SIZE1)
4. Set TEMP[k]=LIST[j]
5. Set i=i+1
6. Set j=j+1
7. Repeat Steps 8, 9 and 10 while(k<SIZE2)
8. Set TEMP[k]=LIST[i]
9. Set i=i+1
10. Set k=k+1
11. Set i=0
12. Repeat Steps 13 and 14 while(i<SIZE1)

13. Set List[i]=temp[i]
14. Set i=i+1
15. Exit

Algorithm 6 MSORT(List, SIZE)

Here List is an array of elements of given SIZE.

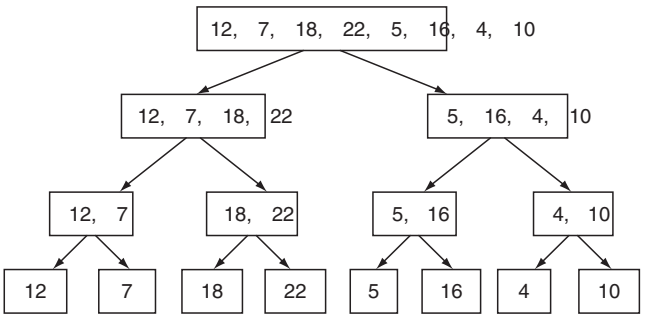
```
If (SIZE>1) then
  {
  int LEFT=SIZE/2;
  int MID=SIZE-LEFT;
  MSORT(List, LEFT);
  MSORT(List+LEFT, MID);
  merge(List, LEFT, MID);
  }
```

Problem 1

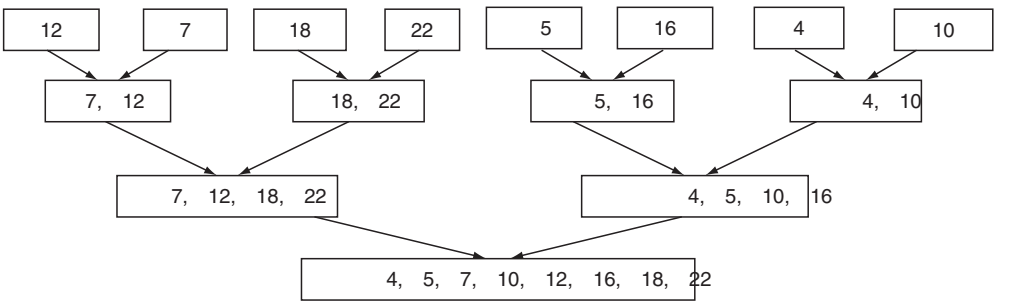
Sort the list 12, 7, 18, 22, 5, 16, 4, 10.

Solution

Step 1: Divide the list.



Step 2: Merge the adjacent lists.



Program 3 A program to implement merge sort in C++ using template.

```
#include<iostream.h>
#include<conio.h>
#define MAX 5
template<class T>
class MSORTING
{
public:
void merge(T List[], int SIZE1, int SIZE2)
{
T temp[MAX];
int i=0, j=0, k=0;
while(i<SIZE1 && j<SIZE2)
{
if(List[i]<List[j])
{
temp[k]=List[i];
i++;
j++;
}
else
{
temp[k]=List[j];
i++;
k++;
}
}
while(i<SIZE1)
{
temp[k]=List[i];
i++;
j++;
}
while(j<SIZE2)
{
temp[k]=List[j];
i++;
k++;
}
for(i=0;i<SIZE1+SIZE2;i++)
List[i]=temp[i];
}
MSORT(T List[],int SIZE)
{
if(SIZE>1)
{
```

```
int LEFT = SIZE / 2;
int MID = SIZE - LEFT;
MSORT(List, LEFT);
MSORT(List+LEFT, MID);
merge(List, LEFT, MID);
}
}
};
void main()
{
clrscr();
MSORT obj;
int List[MAX];
cout<<"*** Enter the elements of a list ***";
for(int i=0;i<MAX;i++)
cin>>List[i];
obj.MSORT(List,5);
cout<<"The sorted list is";
for(i=0;i<MAX;i++)
cout<<"List["<<i<<"] "<<List[i]<<endl;
getch();
}
```

Output

```
*** Enter the elements of a list ***
2 5 3
4 5 6
2 3
4 5
7 8
The sorted list is
List[0] 2 3
List[1] 4 5
List[2] 7 8
List[3] 2 5 3
List[4] 4 5 6
```

10.7 Radix Sort

Radix sort is a modified version of a bucket sort in which numbers of buckets are used. If the radix sort is used to sort the list of decimal numbers then 10 buckets are required and if we want to arrange the names in their alphabetical order then 26 buckets are required. In radix sort the list of elements to be sorted is broken down into several buckets and the elements are assigned to the bucket according to the position of the element in the digit. For example, the digits are assigned first according to the least significant element, then to the next least significant element and so on.

Problem 2

Sort the list of 27, 38, 29, 31, 45, 87, 26, 25 using the radix sort.

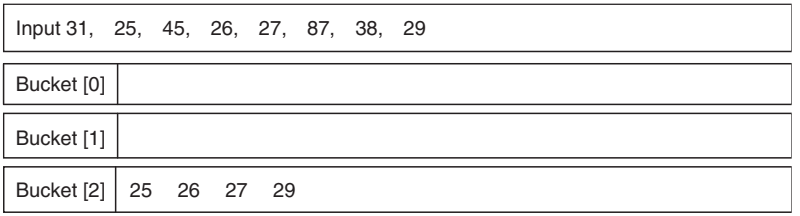
Solution

Pass 1: Assign the elements to each bucket according to the unit position (least significant element) of the digits.

Input 27, 38, 29, 31, 45, 87, 26, 25	
Bucket [0]	
Bucket [1]	31
Bucket [2]	
Bucket [3]	
Bucket [4]	
Bucket [5]	25 45
Bucket [6]	26
Bucket [7]	27 87
Bucket [8]	38
Bucket [9]	29

Pass 2: Read the elements from each bucket starting from bucket 0 to bucket 9. Therefore the input to next pass is 31, 25, 45, 26, 27, 87, 38, 29. Now assign the elements to each bucket according to the ten's position (i.e., second least significant digit) of the element.

Bucket [3]	31 38
Bucket [4]	45
Bucket [5]	
Bucket [6]	
Bucket [7]	
Bucket [8]	87
Bucket [9]	



Read the elements from each bucket starting from bucket 0 to bucket 9. The list now is 25, 26, 27, 29, 31, 38, 45, 87. The list is sorted. The number of passes required to sort the list depends on the maximum number of elements present in the digits. If the greatest number in the list is any three digit number or if maximum three digits are present in any number then three passes are required to sort the list. If maximum four digits are present in any number of the list then four passes are required to sort the list.

Anal y sis of R ad ix Sort

1. **Best case:** The best case complexity is $n \log n$.
2. **Average case**: The average case complexity is $O(n \log n)$.
3. **Worst case**: The worst case complexity is $O(n^2)$.

10.8 Sorting Summary

The approximate number of comparisons and the complexity of the various sorting algorithms presented to you are summarized in Tables 1.

Table 1 Sorting time

S. No.	*Algorithms*	*Worst case*	*Average case*
1	Bubble sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$
2	Quick sort	$n(n+3)/2 = O(n^2)$	$1.4n \log n = O(n \log n)$
3	Insertion sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/4 = O(n^2)$
4	Selection sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$
5	Merge sort	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$
6	Heap sort	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$

Solved Examples

Example 1 *Arrange the array A of elements 16, 12, 11, 13, 14, 15, 18, 17 , 10 in ascending order using quick sort algorithm.*

Solution

1. Left = Start = 0 and Right = End = 8 and Key = A[(Left + Right)/2] = A[(0 + 8)/2] = A[4] = 14

16	12	11	13	14	15	18	17	10
A [0]	A [1]	A [2]	A [3]	A [4]	A [5]	A [6]	A [7]	A [8]

2. Left <= Right (0 < 8).
3. A[Left] < Key is not true.
   Therefore Left will remain same, that is, Left = 0.
4. A[Right] > Key is not true.
   Therefore Right will remain same, that is, Right = 8.
5. Left < Right (i.e., 0 < 8) so interchange the values.
   a. Temp = A[Left] = A[0] = 16
   b. A[0] = A[Right] = A[8] = 10
   c. A[8] = Temp = 16
   d. Left = Left + 1, so Left = 1
   e. Right = Right – 1, so Right = 7
   The list of elements now looks as follows:

10	12	11	13	14	15	18	17	16
A [0]	A [1]	A [2]	A [3]	A [4]	A [5]	A [6]	A [7]	A [8]

   Now repeat the same process from Step 2 onwards.
6. Left <= Right (i.e., 1 < 7).
7. A[Left] < Key is now true.
   Therefore Left = Left + 1, so Left = 2.
8. A[Left] < Key is now true.
   Therefore Left = Left + 1, so Left = 3.
9. A[Left] < Key is now true.
   Therefore Left = Left + 1, so Left = 4.
10. A[Left] is not less than Key so stop changing left pointer.
11. A[Right] > Key is now true.
    Therefore Right = Right – 1 = 7 – 1, Right = 6.
12. A[Right] > Key is now true.
    Therefore Right = Right – 1 = 6 – 1, Right = 5.
13. A[Right] > Key is again now true.
    Therefore Right = Right – 1 = 5 – 1, Right = 4.
14. A[Right] is not greater than Key so stop changing right pointer so interchanges are not done and the key element (14) has got its position now.

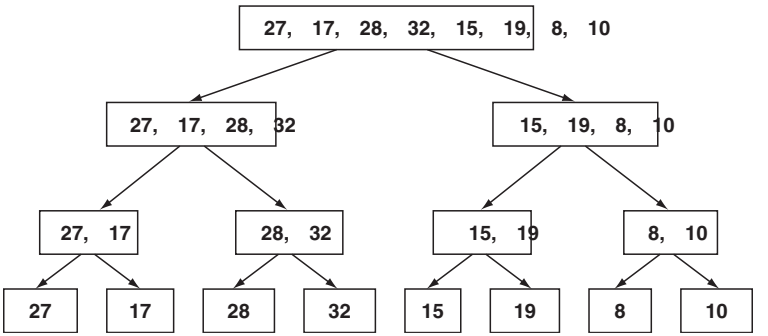
    Now the list of elements looks as follows:

10	12	11	13	14	15	18	17	16
A [0]	A [1]	A [2]	A [3]	A [4]	B[0]	B[1]	B[2]	B[3]

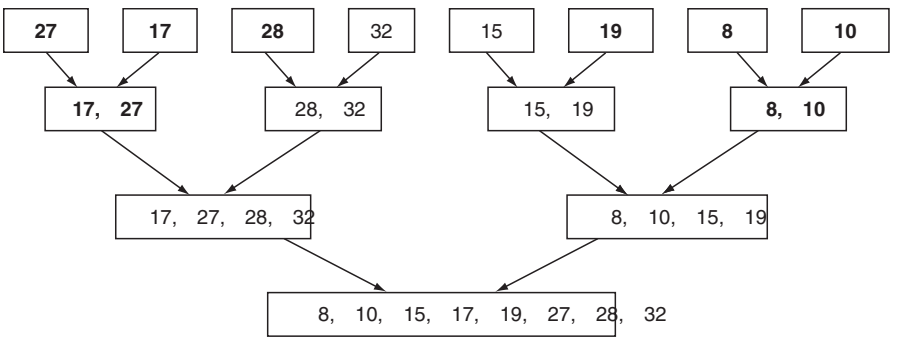
Example 2 *Sort the list 27 , 17 , 28, 32, 15, 19, 8, 10 using the merge sort method.*

Solution

Step 1: Divide the list.



Step 2: Merge the adjacent lists.



Example 3 *Write a menu-driven program in C for sorting a list of elements using quick sort, bubble sort, insertion sort or selection sort.*

Solution

```
#include<stdio.h>
#include<conio.h>
#define MAX 5
int List[MAX];
QSORT(int List[],int start,int end)
{
int temp, left, right, key;
left=start;
right=end;
key=List[(left+right)/2];
do
```

```
{
while ( List[left] <= key )
{
left++;
}
while ( List[right] > key )
{
right--;
}
if ( left <= right )
{
temp = List[left];
List[left] = List[right];
List[right] = temp;
left++;
right--;
}
} while ( left <= right );
if ( start < right )
QSORT ( List, start, right );
if ( left < end )
QSORT ( List, left, end );
}
void BSORT()
{
int i, j, temp;
printf ( "Original list is \n" );
Display();
for ( i = 0; i < MAX - 1; i++ )
for ( j = 0; j < MAX - i - 1; j++ )
if ( List[j] > List[j+1] )
{
temp = List[j];
List[j] = List[j+1];
List[j+1] = temp;
}
}
void ISORT()
{
int i, j, temp;
printf ( "Original list is \n" );
Display();
for ( i = 1; i < MAX; i++ )
```

```
{
temp=list[i];
for(j=i-1;j>=0 && List>temp;j--)
List[j+1]=List[j];
List[j+1]=temp;
}
}
void SSORT()
{
int k,temp,i,MIN,Pos,flag
printf("Original list is \n");
Display();
for(i=0;i<MAX-1;i++)
{
MIN=List[0];
for(k=i+1;k<MAX;k++)
{
if(List[k]<List[i])
{
MIN=List[k];
Pos=k;
flag=1;
}
}
if(flag==1)
{
temp=list[i];
List[i]=List[Pos];
List[Pos]=temp;
}
}
}
Display()
{
int i;
for(i=0;i<MAX;i++)
{
printf("List[%d]=%d",i,List[i]);
printf("\n");
}
}
void main()
{
int ch,i;
printf("*** Enter the elements of a list ***
```

```
for(i=0;i<MAX;i++)
scanf("%d",&List[i]);
clrscr();
do
{
clrscr();
printf("Enter your choice for sorting\n");
printf("1. Bubble sort\n");
printf("2. Insertion sort\n");
printf("3. Selection sort\n");
printf("4. Quick sort\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
BSORT();
printf(" The sorted list is \n");
Display();
getch();
break;
case 2:
ISORT();
printf(" The sorted list is \n");
Display();
getch();
break;
case 3:
SSORT();
printf(" The sorted list is \n");
Display();
getch();
break;
case 4:
printf("Original list is \n");
Display();
QSORT(List,0,4);
printf(" The sorted list is \n");
Display();
getch();
break;
}
}while(ch!=0);
getch();
}
```

Output

```
Enter your choice for sorting
1. Bubble sort
2. Insertion sort
3. Selection sort
4. Quick sort
Original list is
List = 20 3
List = 5 1 6 7
List = 22 4 5
List = 13 2
List = 8 4 1
  The sorted list is
List = 8 0 1
List = 11 2
List = 22 3
List = 23 4 5
List = 5 4 6 7
```

Summary

1. Sorting refers to the operation of arranging records either in ascending or in descending order.
2. Sorting can be divided into two categories: Internal sorting and external sorting.
3. During sorting if the elements are kept in main memory then it is called internal sorting and if the elements are kept in secondary devices like disks tapes then it is called external sorting.
4. Bubble sort, insertion sort, selection sort, quick sort and heap sort are the examples of internal sorting and merge sort is the most common example of external sorting.
5. The complexity of sorting algorithms depends upon coding, space and time required to run the algorithms.
6. Quick sort is the most widely used sorting algorithms based on divide-and-conquer method.
7. For a small list of elements selection, bubble or insertion sort may be used.
8. For a large list of elements quick, heap or merge sort may be used.
9. Quick sort should be avoided when the list is already sorted.

Key Terms

External sorting
Internal sorting
Complexity
Bubble sort
Selection sort
Insertion sort
Radix sort
Quick sort
Merge sort
Heap sort

Multiple-Choice Questions

1. In internal sorting methods all data reside in the
   a. main memory.
   b. disks.
   c. tapes.
   d. floppy.

2. Time for the best case in bubble, selection and insertion sort for sorting N elements is proportional to
   a. N
   b. N^2
   c. log(N)
   d. N log(N)

3. A heap is a tree structure where the largest/smallest elements are available at the
   a. root.
   b. leaf.
   c. right child.
   d. left child.

4. Name the sort for which time is not proportional to N^2.
   a. Selection sort
   b. Bubble sort
   c. Quick sort
   d. Insertion sort

5. Which of the following sort does not use divide-and-conquer methodology?
   a. Merge sort
   b. Quick sort
   c. Bubble sort
   d. Partition exchange

6. A max heap is a tree structure where the largest/smallest elements are available at the
   a. root.
   b. leaf.
   c. right child.
   d. left child.

7. After the first pass of insertion, on sorting the array 15 13 18 19 11 17 0 12 16 14 the array becomes
   a. 15 13 18 19 11 17 0 12 16 14
   b. 13 15 18 19 11 17 0 12 16 14
   c. 0 13 18 19 11 17 15 12 16 14
   d. 0 15 13 18 19 11 17 12 16 14

8. In a selection sort of N elements, how many times does the interchange take place in a complete execution of the algorithm?
   a. 0
   b. $N - 1$
   c. N log N
   d. N^2

9. In external sorting method all data reside in the
   a. main memory.
   b. secondary memory.
   c. either (a) or (b).
   d. none of the above.

10. Internal sorting is useful when the list of elements is
    a. small.
    b. long.
    c. file of records.
    d. any of the above.

11. What is the worst-case time complexity for merge sort to sort an array of N elements?
    a. O(N^2)
    b. O(log N)
    c. O(N log N)
    d. O(N)

12. What is the worst-case time complexity for quick sort to sort an array of N elements?
    a. O(N log N)
    b. O(N^2)
    c. O(log N)
    d. O(N)

13. What is the worst-case time complexity for heap sort to sort an array of N elements?
    a. O(N log N)
    b. O(N^2)
    c. O(log N)
    d. O(N)

14. If the worst-case time complexity to sort an array of N elements is $O(N^2)$, then the sorting algorithms applied may be
    a. bubble sort.
    b. insertion sort.
    c. selection sort.
    d. any of the above.

15. If we have the array 12 15 11 17 19 22 31 18 after first portioning of quick sort, then which of the following statement is correct?
    a. The pivot could be either 17 or 19.
    b. The pivot could be 17, but it is not 19.
    c. The pivot is not 17, but it could be 19.
    d. Neither 17 nor 19 is the pivot.

16. What is the worst-case time complexity for shell sort to sort an array of N elements?
    a. $O(N^2)$
    b. $O(\log N)$
    c. $O(N \log N)$
    d. $O(N)$

17. What is the best-case time complexity for merge sort to sort an array of N elements?
    a. $O(N^2)$
    b. $O(\log N)$
    c. $O(N \log N)$
    d. $O(N)$

18. External sorting is useful when the list of elements is
    a. small.
    b. long.
    c. both (a) and (b).
    d. either (a) or (b).

19. What is the average-case time complexity for shell sort to sort an array of N elements?
    a. $O(N^2)$
    b. $O(\log N)$
    c. $O(N \log N)$
    d. $O(N)$

20. Which of the following sorting algorithms searches for the smallest element in the list and interchanges it with the first element of the list?
    a. Quick sort
    b. Selection sort
    c. Tree sort
    d. Shell sort

21. What is the best-case time complexity for bubble sort to sort an array of N elements?
    a. $O(N^2)$
    b. $O(\log N)$
    c. $O(N \log N)$
    d. $O(N)$

22. The merge sort algorithm is based on
    a. divide and conquer.
    b. greedy techniques.
    c. backtracking.
    d. dynamic programming.

23. Sorting means arranging data elements in
    a. ascending order.
    b. descending order.
    c. either ascending or descending order.
    d. both ascending and descending order.

24. What is the best-case time complexity for insertion sort to sort an array of N elements?
    a. $O(N^2)$
    b. $O(\log N)$
    c. $O(N \log N)$
    d. $O(N)$

25. Which of the following sorting algorithms is slowest?
    a. Quick sort
    b. Bubble sort
    c. Tree sort
    d. Shell sort

26. Which of the following sorting algorithms compares adjacent elements and interchanges them whenever required?
    a. Quick sort
    b. Bubble sort
    c. Tree sort
    d. Shell sort

27. Which of the following sorting algorithms inserts the element into its proper position in the sorted subarray?

a. Quick sort
b. Insertion sort
c. Bubble sort
d. Shell sort

28. The quick sort algorithms is based on
a. divide and conquer.
b. greedy techniques.
c. backtracking.
d. dynamic programming.

29. Which of the following data structure is useful in implementing quick sort?
a. Stack
b. Deque
c. Linked list
d. Queue

Review Questions

1. What are the different types of sorting methods? Compare them in terms of the time complexity.
2. What is the difference between external sorting and internal sorting?
3. Describe the merge sort process with an example.
4. Explain quick sort method by taking a suitable example.
5. What is the complexity of quick sort in different cases?
6. Sort the following array using selection sort: 2, 84, 15, 75, 39, 27, 85, 71, 63, 97, 27, 14.
7. When is bubble sort better than quick sort? Sort the following integers using bubble sort: 32, 51, 27, 85, 66, 23, 13, 57.
8. Explain the selection sort method with example. What are the advantages of selection sort method over other methods?
9. Sort the following integers in descending order using insertion sort: 25, 57, 48, 37, 12, 92, 86, 33.
10. Given the array 40, 55, 20, 30, 50, 15, 25, show the content of the array after each sort listed below:
(a) Insertion sort after fourth pass.
(b) Bubble sort after third pass.
(c) Selection sort after fifth pass.
11. Explain the quick sort algorithm. Discuss its advantages over other sorting techniques.
12. Write any algorithm based on divide-and-conquer technique for sorting a list of N elements.
13. Apply the quick sort on the following list of 12 numbers and clearly show each step: 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66.
14. Write short note on address calculation sort.
15. Define the heap and its type. Describe the mechanism of heap sort with example. Is heap sort always better than the quick sort?
16. Apply quick sort on the following list of 10 numbers and clearly indicate the pivot element and the partitions at each step: 33, 22, 11, 55, 77, 99, 65, 30, 58, 46.
17. Discuss the importance of sorting in computer application.
18. Explain the process of insertion in a given heap in detail.
19. Sort the following numbers using heap sort: 11, 56, 23, 90, 12, 75, 27, 39, 21, 17, 89, 4, 18, 7, 2.
20. How many interchanges and passes are required to sort a file of size N using bubble sort, insertion sort and selection sort.

21. Compare merge sort and insertion sort algorithms.
22. Describe the behavior of quick sort when the list of elements is already sorted.
23. How many key comparisons and assignments are required in an insertion sort to sort list of N elements?

Programming Assignments

1. Write a C program for insertion sort using a doubly linked list.
2. Write a C function to sort a set of N inputs using bucket sort.
3. Write a menu-driven program in C++ to sort a list of N elements using insertion sort, bubble sort and selection sort.
4. Write a menu-driven program in C to sort a list of N elements using heap sort, merge sort and quick sort.

Answers

Multiple-Choice Questions

1. (a)
2. (a)
3. (a)
4. (c)
5. (c)
6. (a)
7. (a)
8. (b)
9. (b)
10. (a)
11. (c)
12. (c)
13. (b)
14. (d)
15. (a)
16. (c)
17. (c)
18. (b)
19. (d)
20. (b)
21. (d)
22. (a)
23. (c)
24. (a)
25. (b)
26. (b)
27. (b)
28. (a)
29. (a)

11 Files

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand the following:

- Definition of the file.
- Basic terminologies of files.
- Classification of files.
- Different types of file organization techniques.
- Implementation of files in C and C++.
- Various operations performed on a file.

In the preceding chapters, we have learnt the basic input/output (I/O) operations. The input to the system may be provided through the keyboard by using `scanf` function in C and `cin >>` in C++. The output may be taken onto the computer's screen by `printf` of C or `cout <<` of C++. These are the functions we have been using throughout the book. However, there are some limitations with these functions, such as

1. It is very difficult to handle large volume of data.
2. It is difficult to store the input data for future use.
3. It is difficult to store the output data for future use.
4. Some of the input data are in image form or in text form.
5. The data are stored into the main memory which has limited size.

To deal with such problems, there is a special kind of data structure called Files. A file is also called as an external data structure where other data structures we have discussed so far are called internal data structure. It handles the inputs and outputs to a program through disks. The file is a collection of records where each record may contain different fields. The records may be of same or variable sizes. If the file contains all the records of the same size then it is said to be made up of a fixed length record. If file consists of records of different sizes then it is said to be made up of variable length records. The variable size of the records may be due the variable size of the field. Some fields may be optional, some fields may consist of multiple values, etc. Before venturing into the details, let us discuss the basic terminologies.

11.1 Basic Terminologies

The various terminologies used are as follows:

1. **Field:** It is the basic unit of the record which represents meaningful information about the record. This is characterized by the data type and size. For example, name, age, sex, roll number of any student.
2. **Record:** It is a collection of logically related fields (information) about any real-world entity. For example, student is a collection of logically related information such as name, age, sex, roll number, etc. Hence, any student may be defined by his/her name, age, sex and roll number.

3. **File:** A file is a collection of records. For example, a file of students of CSE branch or a file of all students in the institute.
4. **File organization:** It is concerned with the arrangement of records on the disks.
5. **Key field:** It is a field that contains unique value for each record, so the record in the file is uniquely identified by specifying the value of a particular field of the record.
6. **Index:** It is a pointer used to determine the location of the record in a file that satisfies some condition.

Basic Operations to the File

1. **Creation:** First the file structure is created before feeding the data into the records. Creation of the file determines the name of the file; the position of the file where the I/O operation is performed; the file structure defined, whether it is a text file or binary file and the access method defined.
2. **Open:** This operation prepares the file for read and write operations.
3. **Retrieve or search**: This operation is performed to search for a record or a set of records having a particular value in a particular field or where the field value satisfies certain conditions.
4. **Insert:** This operation deals with the insertion of a new record or set of records at a specific location.
5. **Update:** The existing records of the file may be updated by changing the values of the fields or the new record may be inserted.
6. **Delete:** Any record or a set of records may be deleted from the file.

The efficiency of a file system depends on how efficiently these operations can be performed on the file.

11.2 Classification of Files

The files can be classified according to the functions, access method or on the basis of the way data are stored.

1. **Master file:** A master file represents the static view of some aspects of an organization at a particular point of time. The master file contains the records which are not frequently changed. So master files contain the data that are permanent in nature. For example, an organization may have the master file for each employee containing his personal details such as name, address, DOB, height, dependents, etc.; or a bank may have the master file containing the account details of each customer such as account number, customer name, address, balance, etc.
2. **Transaction file:** A transaction file contains the collection of data that are applied to a master file to reflect any changes in the master file. A transaction file may contain data to add new records or to modify a record from a master file. The transaction file is also called as "log" file.
3. **Program file:** A program file contains instructions in high-level or assembly or machine languages to process the data. This may be stored in the main memory of in some other files.
4. **Text files:** These files contain the data in the form of American standard code for information interchange (ASCII) codes irrespective of the data types. The character or numeric data are stored in the form of respective ASCII codes. These kind of files contain alphanumeric and graphic data input.
5. **Binary files:** These are similar to the text files, except that these allow writing the numeric data in less number of bytes as compared to text files. So binary file provides a more efficient way of storage, but we cannot read the data directly from the file as they are written. We can read the files only through the programs containing some special commands to read them.

11.3 File Organization

A file is a collection of logically organized records which are mapped onto disk blocks. A key element in file management is the way in which the records themselves are organized inside the file. File organization refers to the way the files are logically arranged on a file system and the physical arrangement of data in a file into records and pages on the disk. A file should be organized in such a way that the records are always available for processing with no delay. Therefore, single or multiple keys may be associated with the files. The key is the field or a combination of fields, which can uniquely define the records. Choosing a file organization is a design decision, so we must consider some factors that affect the organization of a file. Some of them are listed below.

1. Storage efficiency.
2. The nature of the operations that are to be performed on the file.
3. Easy record addition/updation/removal, without disruption.
4. The frequency and fast access of file.
5. The external storage medium on which the file is stored.
6. Ease of information retrieval.
7. Redundancy, reliability, security and integrity.
8. Response time in completion of a file operation.

The file organization differs in record sequencing and record retrieval methods. The most common file organization techniques based on their physical storage and the keys used to access records are as follows:

1. Sequential organization.
2. Direct organization.
3. Indexed-sequential organization.

Sequential File Organization

This is the most common technique of file organization. In this method, the records are arranged one after another in a sequence when the files are created. So in the sequential file organization, the records are placed sequentially on the storage medium. The records are kept in sequence to their key value. The key value is the value of some field which uniquely identifies the records. The records can be accessed in the order of their key values. The nth record of the file cannot be accessed directly until we traverse the preceding $(n - 1)$ records. The order of the records is fixed and they can only be read or written sequentially. Generally, sequential files are organized according to the sorted order of the key field; however, depending upon the application records, it can also be organized on other non-key fields according to the requirement. A sequential file is designed for efficient processing of records in sorted order on some search key. This organization is well suited for batch processing of the entire file, without adding or deleting items. The sequential file organization is shown in Figure 1.

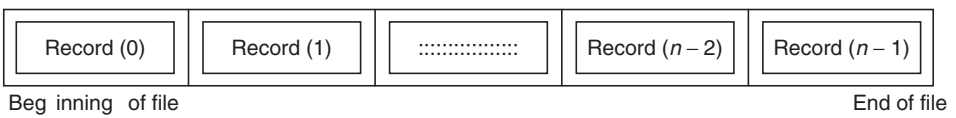


Figure 1 Sequential organization of file.

If a new element or field is added to the record then the entire file is reorganized. Deletion is also done in the similar fashion, but it is not done immediately. The records are always marked for deletion and it is written to the "log file" also called as transaction file; then the record is dropped from the primary data file. Sequential file can be implemented on both kinds of external storage devices, serial as well as direct access storage devices.

Advantages

1. Easy to process and implement.
2. Minimizes the number of block accesses.
3. A sequential file is accessed one record at a time, from first to last, in order. Each record can be of varying length.
4. There is no overhead to calculate any address calculation function to locate a particular record, only the key value is sufficient to locate it.
5. Sequential files can be implemented on magnetic tapes as well as on disks.
6. Well suited for batch processing application.
7. Record in a file can be of varying size.

Disadvantages

1. It is difficult to maintain physical sequential order as records are inserted and deleted.
2. Insertion of new field requires reorganization of files which is expensive, therefore, updates are not easily accommodated.
3. The records cannot be directly accessed so random access is not possible.
4. Data redundancy is very high because the same data are kept in various files which are sorted on different fields.
5. A record of a sequential file can only be accessed by reading all the previous records.
6. Especially not good for programs that make frequent searches in the file.

Direct File Organization

The disadvantage of the sequential organization is that the records can only be accessed in the order of the key value. The direct file organization technique allows us to access the records directly. Records are accessed by addresses that specify their disk location. This technique is also called as relative file organization or random file organization. The relation between the key value and the physical address is established by using a mapping function commonly known as hash function as given below.

H(key) → physical address

The technique to convert the record's key into the physical storage address is called hashing, and the mapping function is called hash function. This has already been covered in the Chapter 9. Many techniques have been developed to implement the concept of direct addressing, for example

1. division method;
2. mid-square method;
3. folding method.

These address-generating functions often map a large number of records to the same address. This situation is known as collision, and various collision resolution techniques to handle the collision have already been discussed in Chapter 9. Relative file organization provides an effective way to access an

individual record directly. The direct file organization requires a disk storage rather than tape. The direct file organization is shown in Figure 2.

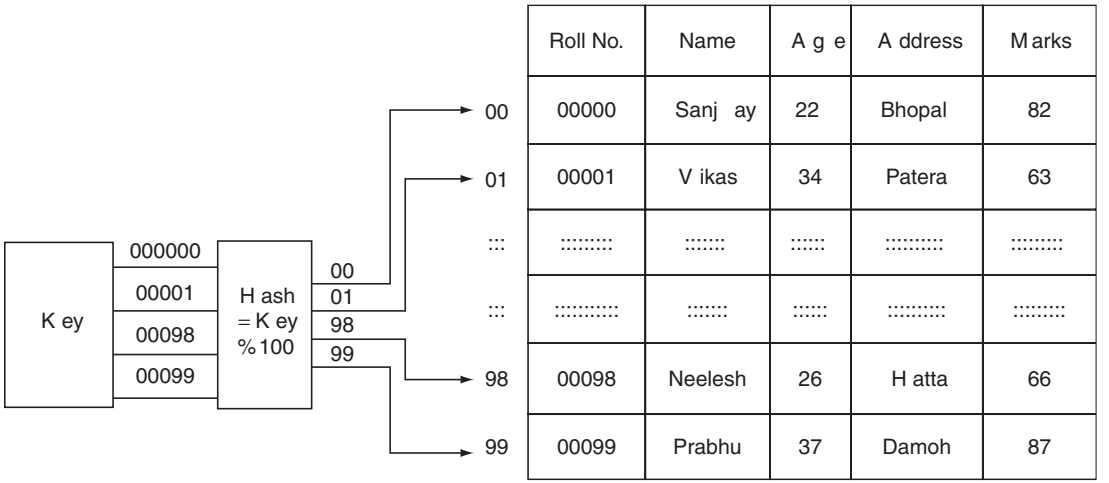


Figure 2 Direct file organization.

Advantages

1. A direct organization is an effective way to organize a file when there is need to access individual records directly.
2. Records can be directly or randomly accessed.
3. A relative file can also be accessed sequentially, but sequential files cannot be accessed directly.
4. Well suited for interactive application.
5. Updating the records is possible and can be easily implemented.
6. It has an excellent search retrieval performance.

Disadvantages

1. It requires the calculation of mapping function.
2. Records are scattered.
3. To get the advantages of this file organization, the files must be stored on some direct storage device like disk.
4. A key field is required for this organization, as well as fixed record length.
5. Additional space is required.

Indexed Sequential File

Indexed sequential files are the hybrid of sequential and direct access files which are used to effectively organize the data. Such data can be accessed directly through some key and it can also be accessed sequentially through the same key. Since indexed sequential file organization supports both direct and sequential file organization method, so it is better for the applications that require both batch and interactive processing. The indexed sequential method of file organization attempts to reduce the access

problems in the sequential files without losing the benefits of the sequential files organization. The indexes can be used to quickly locate any given record for random processing.

An indexed sequential file has two parts: Index part and sequential part. Index would be built as a tree (generally, binary search tree, BST; B-tree or B+ tree) of key values where each node contains a pointer to the sequential data file. Therefore, the indexed sequential search can be implemented using BST and the sequential file organization where the BST is used to structure the indexes. Multiple indexes provide a high degree of flexibility for accessing the data. The indexed sequential file organization allows the use of variable length records. The indexes speed up the search process. Single-level indexing structure is the simplest one where a file, whose records are in pairs, contains a key pointer. This pointer is the position in the data file of the record with the given key. A subset of the records, which are evenly spaced along the data file, is indexed, in order to mark intervals of data records. The indexed sequential file is shown in Figure 3.

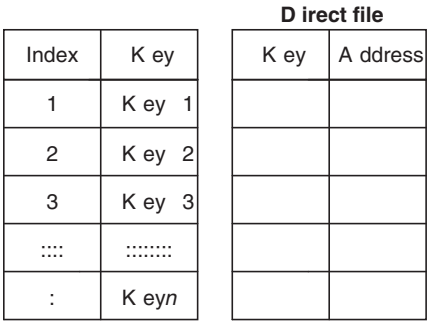


Figure 3 An indexed sequential file organization.

Advantages

1. This allows the sequential as well as random access of records.
2. Due to its capability of supporting both sequential and direct file organization, it is better for interactive as well as batch-oriented applications.

Disadvantages

1. It allows storage only on the disks.
2. It supports direct access so involves overhead of calculating the mapping function.
3. The records can only be of fixed sizes.

11.4 Basic Operations with Files in C++

To open a file in C++, an object of the appropriate stream is created and then it is associated with the filename. In other words, we can say that first the file stream is created and then it is linked to the filename. The object can be created from any of the classes `ifstream`, `ofstream` and `fstream` defined in the header file `fstream.h`. If we want to read data from the file then we have to use the object of `ifstream` class, if we want to write to the file then we have to use object of the `ofstream` class and if we want both reading and writing operations with the file then we have to use the object of `fstream` class. These are the three streams that are contained in the `fstream.h` header file. So we must include this file before creating the

object of any stream. In other words, we can say that if the file is declared in `ifstream` then that file is used for reading the data from the file. If the file is declared in the `ofstream` then the file is used only for writing the data to the file. Similarly, if the file is declared in `fstream` then the file can be used for both reading and writing. We can open the file in two ways: by using the constructor of the appropriate class and by using the `open` member function. Each method is described in detail in the following subsections.

Opening File by Using Constructor of the Appropriate Stream

We know that the constructor is used to initialize the object; similarly, when we create object of stream class it is automatically initialized with the filename passed to it. This method is useful when we use only one file in the stream. So when we want to open the file using the constructor, we have to pass the filename as an argument. For example, `ifstream` is used to create the input stream object and initialize the object with the filename.

Syntax

```
streamtype objectname(filename, openin
```

Example 1

```
ifstream inputobj("sample")
```

This statement creates an object `inputobj` of `ifstream` class, and the file `sample` is associated with the object. The operating system takes the name of the file as `sample`.

After linking the `inputobj` with `sample`, the `sample` file is opened and attached to the `ifstream`. Now, to read from this file, the `get from` operator (>>) is used with the stream object and to write to the file, the `put to` operator (<<) is used with the stream object. Here the opening mode is optional. When we do not supply the opening mode, it takes the default value `ios::in` for `ifstream` object and `ios::out` for `ofstream` object.

Example 2

To read from the file

```
ifstream inputobj("sample")
char str;
inputobj>>str;
```

Only reading is possible from the sample file because it is opened with the object of `ifstream` class.

Similarly, to write to the file

```
ofstream outputobj("sample")
char str;
output<<str;
```

Only writing is possible into the `sample` file because it is opened with the object of `ofstream` class.

Opening File Using open() Function

The open() function is used to open multiple files with the same stream object to process them sequentially. So the first file can be closed before opening the second file because a stream can be connected to only one file at a time. The general syntax for open() is as follows.

Syntax

```
streamtype objectname;
objectname.open(filename, opening mode);
```

Here the filename is any valid identifier and the opening mode is represented by a name with the ios using scope resolution operator. For ifstream, the default mode is ios::in; for ofstream, the default mode is ios::out and for fstream, there is no default mode. Therefore, one must specify the mode explicitly when using an object of fstream class. The bitwise OR operator is used to declare more than one mode. The possible values for the opening modes are discussed later in the section "File Access Modes".

If more than one file is required simultaneously, then the files may be opened in different streams; for example, one in ifstream and other in ofstream. However, if two files are required simultaneously in the same stream, then in that case we need to create separate streams for handling separate files. If the files are processed in a sequential manner, then we may open a single stream and associate it with each file in turn. We can say that we can use the same object with the different files.

Example 3

```
ifstream inputobj;
inputobj.open("sample");
ifstream inputobj1;
inputobj1.open("example");
```

When the file is opened for both reading and writing, the I/O stream keeps track of different file pointers for input and output operations.

Program 1 A program in C++ to read and write the natural number to the file and using the constructor to open the file.

```
#include<fstream.h
#include<conio.h>
  void main()
    {
    clrscr();
    ofstream objout("natural");
    int i,n;
    cin>>n;
```

```
    for=1(;id=n ;+i+)
    {
    obj<<\ut <<i ;
    }
    objout.close();
    ifstream inobj("natural");
    while(!inobj.eof())
    {
    in>>ij;
    co<<it<<"    ";
    }
    getch();
    }
```

Output

```
Enter the limit 5
1       2       3       4       5
```

Program 2 A program in C++ to create and read student's information from the file and display it and using the open ( ) function to open the file.

```
#incl<ufdsetrea>m.h
#incl<ccoeni>.h
int main()
{
ofstream oputobj;
oputobj.open("Sample"); //open file sample
char name[20];
int i,rollno;
int marks[5];
cou<<"Enter the name ";
ci>>name;                    //Read name from the k
oput<<nja<<endl;     //write the name to the fi
cou<<"Enter roll number ";
ci>>rollno;
oput<<rjll<<eondl; //write rollno to the file
cou<<"Enter the marks in five subjects ";
for=0i;<5 ;+i+)
{
cou<<"Enter marks";
ci>>marks[i];
oput<<mjarks[i];      // write marks to the fi
}
```

```
oputobj.close();
ifstream iputobj;
iputobj.open("sample");   //open sample
mode(ifstream)
iput>>hnjame;     //read name from sample file
iput>>hrjollno;  //read rollno from sample fi
for=(0i;<15;+i+)
{
iput>>hmjarks[i];
}
cou<<tendl;
cou<<t"The details of th<<ensdtlu;dent "
cou<<t"Name  <<n:am"<<endl;
cou<<t"Roll  n<<r:ol"l<<reondl;
for=(0i;<15;+i+)
{
cou<<t"Marks in su<<bij<<"qt[i]"<<m"arks<<[<<ei]ndl;
}
iputobj.close();
getch();
}
```

Output

```
Enter the name Ajay
Enter roll number 1001
Enter the marks in five subjects
Enter marks 77
Enter marks 77
Enter marks 77
Enter marks 77
Enter marks 77
The details of the student
Name    : Ajay
Roll no: 1001
Marks in subject [0] is  77
Marks in subject [1] is  77
Marks in subject [2] is  77
Marks in subject [3] is  77
Marks in subject [4] is  77
```

R ead ing and Writing

After a file has been opened and attached to an object of type `f s t r e a m` or `o f s t r e a m` or `f s t r e a m`, access to the file is achieved through the use of the extraction and insertion operators we use with `c i n` and `c o u t`. The << operator is used to send the data to the file and >> operator is used to retrieve data from the file. These operators are used with the object of the appropriate stream.

Example 4

```
ofstream outputobj;
outputobj.open("sample");
output<<"bCjS E<"\n";
output<<"bjNBA Accr<<"i\tne"d;"
outputobj.close();
ifstream inputobj;
inputobj.open("example");
char name[20];
input>>bjme;
inputobj.close();
```

File Access Modes

The file can be opened by using the stream constructor or by using the `open` function. The syntax for both the methods has already been discussed.

```
filestream object.open("filename",mode)
filestream object ("filename",mode);
```

We have not discussed the "mode" which specifies the purpose for which the file is opened. The file can be opened in various modes which describes how a file is to be used in the program: for reading only, for writing only or to be appended. File modes are specified at the time of opening files. The file modes can take one of such constants of class `ios` given in the following table.

When the file is created by `ifstream` and mode is not given, then the default mode is `ios::in`. Similarly, for the file created by using `ofstream` the default mode is `ios::out`. We can also use two modes by using the bitwise OR operator. When we open a file in read-only mode, the `input` pointer is automatically set at the beginning so that we can read the file from the start. Similarly when we open a file in write-only mode, output pointer is set at the beginning to allow us to write to the file from the start. In case we want to open an existing file to add more data, the file is opened in "append" mode. This moves the output pointer to the end of the file.

Name of the mode (member function)	*Meaning*
`ios::in`	This specifies that the data can be extracted from the file means it is open for reading.
`ios::out`	This specifies that the data can be inserted into the file means it is open for writing.
`ios::app`	This specifies that the data can be appended at the end of the file.
`ios::ate`	This specifies that upon opening of the file it seeks the end of a file instead of beginning and I/O operations can still be done anywhere in the file.
`ios::trunk`	Delete a file with the pre-existing one with same name and truncates the file to zero length.

(*Continued*)

Name of the mode (member function)	*Meaning*
`ios::nocreate`	It will cause `open` function to fail if the file does not already exist.
`ios::replace`	It will open a file if it exists.
`ios::binary`	Open a file for binary mode instead of default text mode. When a file is opened in text mode, various character translations may also take place.

Program 3 A program in C++ to read and write data into the file using `fstream` class.

```
#include<fstream.h
#include<conio.h
void main()
{
clrscr();
fstream fobj;
fobj.open("sample.dat",ios::in);
fobj<<"This is just a sample program "
fobj<<" to print department of CSE "
fobj<<" using fstream class"
fobj.close();
fobj.open("sample.dat",ios::in);
char temp[20],ch;
while(!fobj.eof())
{
fobj>>temp;
cout<<temp;
}
fobj.close();
fobj.open("sample.dat",ios::in);
cin.get(ch);
while(!fobj.eof())
{
fobj.getline(temp,20);
cout<<temp;
}
fobj.close();
getch();
}
```

Output

```
This is just a sample program to print de
fstream class
This is just a sample program to print de
fstream class
```

Closing a File

The connection with a file is automatically closed when the input and the output stream objects go out of scope and a file is disconnected with the stream it is associated with. So when the program terminates, the file is automatically closed. Closing such a connection does not eliminate the stream. The `close()` member function is also used to close the file it is called automatically by the destructor function. It may also be called explicitly. It does not take any argument and takes the following general form.

Syntax

```
streamobject.close();
```

Example 5

```
ifstream inputobj;
inputobj.open("sample");
inputobj.close();
inputobj.open("example");
inputobj.close();
```

Reading and Writing Characters

The file stream classes support a number of member functions for performing the input and output operations on files. The `get()` function is used to read single character from the file. `get()` is a function of `istream` class. The `get()` function have many forms.

```
istream inputobj;
char varch
```

1. `var=inputobj.get()`
   In this form the `get()` function returns the next character from the stream into the character variable `varch`. It returns EOF if end of file is encountered.
2. `inputobj.get(char &varch)`
   This form of `get()` function reads a single character from the associated stream and stores the value in `varch` variable of character type. In this form `get()` function returns the reference to the stream.
3. `inputobj.get(char *temp, int value, char e`
   This form of `get()` function reads character in the character array pointed by the `temp char` pointer until it reads value characters or the `endmarker` is encountered.
4. `outputobj.put(char varch)`

   This form of `put()` function is used to write characters to the file. It is a function from the `ostream` class and has the following form.

The `put()` function has the following form:

```
ostream outputobj;
outputobj.put(char varch)
```

The `put()` function writes the value of `varch` and returns a reference to the stream.

Example 6

```
ofstream outputobj;
outputobj.open("sample");
char ch;
while(!outputobj.eof())
{
cout<<"Get the character from the file";
ch=outputobj.get();
cout<<"Display the character";
cout.put(ch)
}
outputobj.close();
}
```

Program 4 This program is in C++ to copy the contents of a file to another file using `get` and `put` functions.

```
#include<fstream.h
#include<process.h
#include<conio.h>
void main()
{
clrscr();
ifstream file1;
ofstream file2;
char sfile[13],tfile[13];
cout<<"Input the source filename ";
cin>>sfile;
cout<<"Enter the target filename ";
cin>>tfile;
file2.open(sfile);// opens sfile for writ
file<<"This is the program ";
file<<"for copying the file ";
file2.close();
file1.open(sfile);//opens sfile for readi
file2.open(tfile);//opens tfile for writi
if(!file1)
{
cerr<<" sorry can not open file ";
exit(1);
}
cout<<"Copy the source file content into t
n";
```

```
char varch;
while(!file1.eof())
{
varch=file1.get();// we can use file1.get(var
file2.put(varch);
}
cout<<"The file is copied"<<endl;
file1.close();
file2.close();
cout<<"Display the copied contents"<<endl;
file1.open(tfile);
while(!file1.eof())
{
file1.get(varch);
cout<<varch;
}
file1.close();
file1.open(tfile);
while(!file1.eof())
{
file>>varch;
cout<<varch;
}
file1.close();
getch();
}
```

Output

```
Input the source filename : Source
Enter the target filename : Target
Copy the Source file content into the Targe
The file is copied
Display the copied contents
This is the program for copying the file
Display the copied contents
Thisistheprogramforcopyingthefile
```

Reading and Writing Strings

String can be read from and written into the file by using insertion and extraction operators. Apart from using the `get()` function, another function named `getline()` is also used to read the string. The `getline()` function is used to read character stream from the input stream and it has the same form as that of `get()` function. However, it removes the `endmarker` character when encountered. This is not done by the `get()` function, which means that with the `get()` function the

endmarker character remains in the string until the next input operation. The syntax for the getline function is

Syntax

```
istream inputobj;
inputobj.getline(char *temp, int=\mu'm),; ch
```

Example 7

```
ifstream inputobj;
char name[20];
inputobj.open("example");
while(inputobj)
{
inputobj.getline(name,20,'*');
cou<<name<<endl;
}
inputobj.close();
ofstream outputobj("sample");
output<<"bCjS E<"\n";
output<<"bNjB A Accre<<"i\tne"d;"
outputobj.close();
ifstream inputobj("example");
char name[20];
input>>bjme;
inputobj.close();
```

Binary Files

When a file is opened in binary mode it is called a binary file. In a binary file, the data are stored in binary format instead of ASCII characters format. The binary format data file normally takes less space as we can understand from the example; if we want to store 125 in the ASCII character format (text mode) it will take three bytes, one for each digit (character). However, if we store it in binary format then only two bytes (for 125 as an integer) are required to store it. Since the data conversion is not required when we use the binary mode, therefore data are saved faster than the text mode. Due to the change in the internal representation of data from one computer to another, the binary data file cannot be easily transferred from one computer to another. In order to open a binary file, it is required to use the following mode.

```
infile("sample_file",ios::binary);
read() and write()
```

The read and write functions are used to read and write a block of data to the secondary memory. The read and write functions take the following form

```
istream-object.read((char *) &temp, int si
ostream-object.write((char*) &temp, int si
```

These functions take two arguments, the first is the address of the variable and the second is the length of that variable in bytes. The address of the variable must be cast to type char because the

input and output values are both character pointers for `read()` and `write()` functions. The `read()` function reads a size of (`tem`) bytes from the associated stream and outs them in the buffer pointed to by `tem`. The `write()` function writes the size of (`tem`) bytes to the associated stream from the buffer pointed to by `tem`. The data written to a file using `write()` can only be read accurately using `read()`.

Random Access Files

In a sequential file, we can move from the start to the end of file and can access the data stored in the file one after the other. We cannot move to a particular record randomly. In the random access of file, the file pointer may be moved directly to some particular location. The `fstream` file is required to declare the random access file. This file may be opened in any of the following mode of access.

Mode of access	*Meaning*
`ios::in`	In order to read a file
`ios::out`	In order to write a file
`ios::ate`	In order to append
`ios::binary`	Binary format

Classes and File Operations

One of the disadvantages of the I/O system of C is that it cannot handle user-defined data types but C++ is an object oriented programming (OOP) language, and classes and objects are the key features of it. Therefore, it is quite natural that the languages allow us for writing to and reading objects from the disk files directly. The same functions `read()` and `write()` are used for reading and writing class objects. These functions handle the entire structure of an object as a single unit. So we can find the length of an object by using `sizeof` operator. We must note here that the `sizeof` operator will return the total size of the data members because member functions are not written to the disk. The `read()` function is used to get data for the stream of objects from a specified file and the `write()` function is used to write stream of objects into the specified fields.

Files of Arrays

It is well known that an array is a user-defined data type and it contains the similar type of data items. The arrays can also be written into the file and the array elements read from the file using the `write()` and `read()` functions, respectively. This is explained in the following program

Program 5 This program is used read and write array of objects in a file.

```
#include<fcntl.h>
#include<iostream.h>
#include<stdlib.h>
struct address
{
int hno;
```

```
char city[20];
long int pin;
};
class student
{
int rollno;
char name[20];
int marks[5];
address add;
float sum;
char grade;
public:
student()
{
sum=0.0;
}
void inputdata()
{
cout<<"Enter information for student ";
cout<<"\nRoll no. ";
cin>>rollno;
cout<<"\nName :";
gets(name);
cout<<"\Marks in five subjects ";
for(i=0;i<5;i++)
{
cin>>marks[i];
}
cout<<"\n** Address **";
cout<<"\nHouse no. ";
cin>>add.hno;
cout<<"\nCity ";
gets(add.city);
cout<<"\nPin ";
cin>>add.pin;
}
void display();
};
void student::display()
{
for(i=0;i<5;i++)
{
sum+=marks[i];
}
```

```
float avg = sum / 5;
if (avg >= 75)
grade = 'A';
else if (avg >= 60 && avg < 75)
grade = 'B';
else
grade = 'C';
cout << "\nThe information for student ";
cout << "\nRoll no. ";
cout << rollno;
cout << "\nName :";
cout << name;
cout << "\nMarks in five subjects ";
for (i = 0; i < 5; i++)
{
cout << marks[i] << " ";
}
cout << "\nGrade : ";
cout << grade;
cout << "\n** Address **";
cout << "\nHouse no. ";
cout << add.hno;
cout << "\nCity ";
cout << add.city;
cout << "\nPin ";
cout << add.pin;
}
void main()
{
clrscr();
student X[5];
char file[10], ch;
cout << "Enter filename ";
cin >> file;
fstream fobj;
fobj.open(file, ios::out);
int num;
cout << "\nHow many records";
cin >> num;
for (i = 0; i < num; ++i)
{
X[i].inputdata();
fobj.write((char *) &X[i], sizeof(X[i]));
}
```

```
fobj.close();
fobj.open(file,ios::in);
fobj.seekg(0);
for(i=0;i<num;++i)
{
cout<<"\nDisplay the students information";
fobj.read((char *)&X[i],sizeof(X[i]));
X[i].display();
}
getch();
}
```

Output

```
Display the students information
The information for student
Roll no. 1
Name :Rajiv
Marks in five subjects 67 88 99 77
Grade : H
** Address **
House no. 1
City Bhopal
Pin 462021
Display the students information
The information for student
Roll no. 2
Name :Rakesh
Marks in five subjects 55 45 66 44
Grade : S
** Address **
House no. 2
City Raipur
Pin 462042
```

Error Handling in Files

There are various functions of the `ios` class which handle errors in the file system, for example, whether the end of file is reached, a file for reading might not exist, we may attempt to reading past

Function	*Meaning*
`int bad()`	Returns true if any invalid file operation has been attempted or there is an unrecoverable error. The `bad` member function returns a non-zero value if it is true; otherwise returns a zero.

(*Continued*)

Function	*Meaning*
`int eof()`	`eof` is a member function of `ios` class. This is used to check whether a file pointer has reached to the end of a file character or not while reading. It returns a non-zero value if the end of file condition is encountered and a zero otherwise. Detection of end of file condition is necessary for preventing any further attempt to read data from the file.
`int fail()`	This member function of `ios` class is used to check whether a file has been opened for input or output successfully. It returns non-zero value when an input or output operation has failed otherwise returns zero.
`int good()`	This is used to check whether the previous file operation has been successful or not. If the value returned by `good()` function is non-zero then no error has occurred and we can proceed to perform I/O operations. So `good()` returns true if no error occurred otherwise returns false.

the end of file or there may not be sufficient space. These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby take the necessary corrective measures.

File Operations and Random Access

Each file has two associated pointers with it. They are known as the `get` pointer (also called as input pointer) and the `put` pointer (also called as output pointer). We can use these pointers to move through the files while reading or writing. When the I/O operations take place; an appropriate pointer is automatically set according to mode. When file is opened in read only mode then the `get` pointer is set to the start of the file. When the file is opened in the append mode, the file pointer is automatically set to the end of the file. Each time an I/O operation takes place; the appropriate pointer is automatically advanced. The input pointer is used for reading the contents of a given file location and the output file pointer is used for writing to a given file location. All the actions on the file pointers take place automatically by default but in C++ there are some manipulators by which we can control the movement of pointer and achieve the random access. We must note here that file pointer is set up within a file to read or write data they are not used to store the address as is done with normal pointers. The following functions of the `fstream` classes support random access files.

1. **`seekg()`**
   This is the function from input stream (`ifstream`) which is used to move `get` pointer to a specified location for random input operations. The general format for `seekg()` is

   ```
   seekg(offset);
   istream-object.seekg(int)
   istream-obj.seeekg(offset,seekar
   ```

   The parameter offset represents the number of bytes the `get` pointer moves to a position relative to `seekar`. For example, `istream-object.` moves the file pointer to the byte number 10 from the starting point. The `seekar` may take any value from the table given.

Seekarg	*Action*	*Example*	*Description*
`ios::beg`	Go to start of the file	`istream-object.seekg(0,ios::beg);`	Moves the file pointer to the beginning of the file
`ios::cur`	Stay at the current position	`istream-object.seekg(-2,ios::cur);`	Moves the file pointer back by 2 bytes
`ios::end`	Go to end of file	`istream-object.seekg(10,ios::end);`	Moves the file pointer to 10 bytes before the end of file

2. **`seekp()`**
   This is the function from output stream (`ofstream`) and is used to move `put` pointer to a specified location for random output operations. The general form of `seekp()` is

   ```
   seekp(offset) ostream-object.seekp(int)
                 ofstream-object.seekp(offset,
   ```

3. **`tellg()`**
   This is the function from input stream (`ifstream`) and is used to give the current position of the `get` pointer. The method `tellg()` returns the position (in terms of byte number) of `getpoi` in the input file. It can be used to check the current position of file pointer for input stream.
4. **`tellp()`**
   This is the function from output stream (`ofstream`) and is used to give the current position of the `put` pointer. The method `tellp()` returns the position (in terms of byte number) of `put` pointer in an output file. It can be used to check the current position of file pointer for the output stream.

Updating a File

To update a file, random access to the file is required because the file pointer is moved to the location corresponding to the element under consideration. The updation of a file may involve addition of new elements to the existing file, deletion of elements from the existing file, modification of the existing elements or display of the contents of the file. The movement of the file pointer can be accomplished by using `seekg()`, `seekp()`, `tellg()`, `tellp()` function. In updation of a file, a size of object is required if the file contains the elements of equal lengths and the location of the desired object(`m`) may be obtained by `m * sizeof(o)`. This gives us the starting address of the `m`th object. We can move the file pointer to this location by using `seekg()` or `seekp()` functions. To find the file size, we can use the `tellp()` or `tellg()` functions and then we can also find the number of objects in the file by using `file/object size`

Program 6 A program to read all the records in the file.

```
#include<fstream.h>
#include<conio.h>
#include<iomanip.h>
#include<stdio.h>
#include<stdlib.h>
```

```
struct address
{
int hno;
char city[10];
long int pin;
};
struct student
{
int rollno;
char name[10];
int age;
address add;
};
student X;
main()
{
clrscr();
fstream stdfile;
stdfile.open("studentf",ios::out);
char ans;
do
{
cout<<"Enter information for student ";
cout<<"\nRoll no. ";
cin>>X.rollno;
cout<<"\nName :";
gets(X.name);
cout<<"\nEnter age";
cin>>X.age;
cout<<"\n** Address **";
cout<<"\nHouse no. ";
cin>>X.add.hno;
cout<<"\nCity ";
gets(X.add.city);
cout<<"\nPin ";
cin>>X.add.pin;
stdfile.write((char *)&X,sizeof(X));
cout<<"\nDo you want to add more records ";
cin>>ans;
}while(ans=='y'||ans=='Y');
stdfile.open("studentf",ios::binary||ios::
cout<<"******** Students record *******\n";
cout<<"Roll"<<setw(10)<<"Name"<<setw(10)<<"Age"<<setw(10);
cout<<"House"<<setw(10)<<"City"<<setw(10)<<"Pin"<<endl;
stdfile.seekg(0,ios::beg);
```

```
stdfile.read((char *)&X,sizeof(X));
do
{
cout<<X.rollno<<setw(10)<<X.name<<setw(10)<<X.age<<setw(10);
cout<<X.add.hno<<setw(10)<<X.add.city<<setw(10)<<X.add.
pin<<endl;
stdfile.read((char *)&X,sizeof(X));
}while(stdfile.eof());
stdfile.close();
getch();
}
Enter information for student
Roll no. 1
Name :Ramesh
Enter age23
** Address **
House no. 123
City Bhopal
Pin 123456
Do you want to add more records n
******** Students record *******
Roll no.      Name          Age      House no.
    1         Ramesh         23          123
```

Command Line Arguments

We can execute the C program from the DOS prompt. In fact, we can pass some extra argument also at the same time. This mechanism is called command line argument. Command line arguments enable one to specify the arguments along with the executable filename when invoking it for execution. So from the command line, we can pass arguments to the `main` function and the format for the `main` function would be `main(argc, char *argv[])`

Here `argc` is the argument counter which represents the number of arguments in the command line and the second argument is an array of character type pointers that point to the command line argument. The size of this array is equal to the argument counter. For example, if we write

```
C:>example file1 file2
```

In this case the value of argument counter is 3 and the `argv[3]` is an array of three character pointers where each represents

```
argv[0] points to the example
argv[1] points to file1
argv[2] points to file2
```

Suppose the name for the following program (file) is example, and we pass following command from the command line

```
C:>example negfile posfile
```

Program 7 A program to use command line arguments.

```
#include<iomanip.h>
#include<conio.h>
int main(int argc,char *argv[])
{
clrscr();
int array[]={-12,01,22,1,-33,14,16,76,-79,-25,88};
if(argc!=3)
{
cout<<"Argument count"<<argc<<endl;
cerr<<"Error in number of arguments";
}
ofstream obj1,obj2;
obj1.open(argv[1]);
if(obj1.fail())
{
cerr<<"can not open file";
return(1);
}
obj2.open(argv[2]);
if(obj2.fail())
{
cerr<<"can not open file";
return(1);
}
int i=0;
while(i<10)
{
if(array[i]
obj1<<array[i]<<endl;
else
obj2<<array[i]<<endl;
i++;
}
obj1.close();
obj2.close();
ifstream obj;
i=1;
char ch;
while(argc)
{
obj.open(argv[i]);
cout<<"The data in the argv file"<<i<<endl;
do
```

```
{
obj>>ch;
cout<<ch;
}while(obj);
cout<<endl;
}
obj.close();
}
```

11.5 Basic Operations with Files in C

Open a File

To open a file in C, first we have to declare the file using `FILE` data type of C language. After declaring the file it can be used for any purpose. To open the file, the statements are as follows.

Syntax

```
FILE *fptr;
fptr=open("filename", "mode");
```

Here `fptr` is a pointer to `FILE` type, `filename` is the name of the file and `mode` may be any one from the following table.

Mode	*Description*
`w`	Open the file in `write` mode if it exists already then delete the contents.
`a`	Append the data to the end of the existing file.
`r`	Read data from the existing file.
`r+`	Read and write on the file.
`w+`	Read and write on the file.
`a+`	Grant the read permission for the file already opened in append (`a`) mode.

Example 8

```
FILE *fptr;
fptr=fopen(("sample.txt", "r"))
```

This statement creates a pointer of `FILE` type and the file named `sample.txt` is opened in `read` mode.

Closing a File

After performing various operation on the file it must be closed

Syntax

```
fclose(fptr);
where fptr is a pointer in which file w
```

Reading and Writing into the Files

To read from the file, the `get c` is used with the file pointer and to write into the file `put c` is used with the file pointer and the character to write.

Syntax

```
putc(ch, file pointer);
```

where `ch` is a character value to write and `file po` is a pointer in which file is opened.

Here we must remember that `file po` is automatically incremented or decremented when something is written using `put c` or read using `get c` from the file. The `put ch` and `get ch` are used to deal with the single character, but if we want to deal with multiple data then `fprintf()` and `fscanf()` are used.

Syntax

```
fprintf(file pointer, data to write sepa
```

Example 9

```
fscanf(fp, "%d %c %f", &age, &sex, &height)
fprintf(fp, "%d %c %f", age, sex, height)
```

`fputs` is used to write a string into the file. It takes two arguments: the string name and the file pointer. `fgets` is used to read the string from a file. It requires three arguments, the string name, the length of string to read and the file pointer.

Syntax

```
fputs(string, file pointer);
```

where `str` is a character array to write and `file po` is a pointer in which file is opened.

```
fgets(string, length, file pointer);
```

where `str` is a character array to write, `leng` is the number of characters to read and `file po` is a pointer in which file is opened.

Random Access of File in C

The different functions to access the file randomly are summarized as follows.

Function	*Description*	*Example*
`ftell(fptr)`	It returns the current position of the file pointer.	`N=ftell(fptr)`
`rewind(fptr)`	It sets the pointer *t* at the starting of the file from the current position.	`rewind(fptr)`
`fseek(fptr, offset, position)`	It moves the pointer by the given offset from the desired position in the file.	`fseek(fptr,n,1)` Moves the file pointer forward by *n* bytes from the current position.

Error Handling in C

There are two functions which usually deal with the error occurred during the execution of the file.

Function	*Description*	*Example*
`feof(fptr)`	This is used to check the end of file pointed by file pointer `fptr`	`feof(fptr)`
`ferror(fptr)`	This is used to tell the status of the file upto the current position.	`ferror(fp)` will return non-zero integer if there is an error occurred in the file upto the current position.

Summary

1. A file is a collection of meaningful logical records.
2. A record is a collection of meaningful and related information.
3. In sequential organization, records are arranged in sequence.
4. In direct file organization, the key value is mapped to the physical address.
5. Index sequential files are hybrid of sequential and direct file organizations.
6. Hash function generates an integer number when applied to the key value, and this number can be used as an array index.
7. Hashing is a searching technique that avoids number of comparisons and goes directly where the required data are present.
8. Hashing searches the element in a constant search time O(1), no matter where the element is located in the list.
9. Hash table can be searched in O(1) time using a hash function.
10. When two or more key values hash to the same location in an array, such a situation is called collision.

Key Terms

File
Direct
Sequential
Indexed sequential
Hash
Hashing
Mid-square
Division
Folding
Collision
Collision resolution

Multiple-Choice Questions

1. In a binary file
   a. no character translation takes place.
   b. only new line character is translated.
   c. only carriage return is translated.
   d. none of the above.

2. `fstream` class of C++ is derived from
   a. `iostream` class.
   b. `istream` class.
   c. `ostream` class.
   d. `iomanip` class.

3. A file in C++ can be opened using
   a. constructor of the appropriate class.
   b. `open` function.
   c. `append` function.
   d. both (a) and (b).

4. Which mode is used to seek the end of file upon opening a file in C++?
   a. `ios::in`
   b. `ios::out`
   c. `ios::ate`
   d. `ios::app`

5. Which of the following is not a file mode in C++?
   a. `ios::ate`
   b. `ios::octal`
   c. `ios::binary`
   d. `ios::nocreate`

6. Which of the following puts the pointer to end of the file in C++?
   a. `ios::ate`
   b. `ios::app`
   c. `ios::out`
   d. None of the above

7. Which of the following are used to read and write block of data in C++?
   a. `read` and `write()`
   b. `get` and `put()`
   c. `getline` and `cout`
   d. None of the above

8. Which of the following is not an I/O error handler in C++?
   a. `bad()`
   b. `eof()`
   c. `good()`
   d. None of the above

9. The extraction operator (>>) belongs to
   a. `ostream` class.
   b. istream class.
   c. conio.h header file.
   d. none of the above.

10. `cin` is
    a. an object of `istream` class.
    b. an object of `ostream` class.
    c. an operator not an object.
    d. none of the above.

11. Which class supports opening a file in write mode?
    a. `ofstream` class.
    b. `ifstream` class.
    c. `istream` class.
    d. `ostream` class.

12. `fobj.seekg(0)` will place the pointer to
    a. the beginning of `fobj`
    b. the end of `fobj`

c. It depends on the offset.
d. It depends on the arguments.

13. What is the meaning of `seek(fp, 2, SEEK_END)`?
a. Position the pointer to the second byte from the end.
b. Position the pointer to the second byte from the beginning.
c. It cannot move in the negative direction.
d. None of the above.

14. Which operator is used to output to an output file?
a. >>
b. <<
c. ? :
d. None of the above.

15. Which of the following is used to access the file randomly in C?
a. `ftell(fptr)`
b. `rewind(fptr)`
c. `fseek(fptr, offset, position)`
d. All of the above

16. Which of the following statement is correct to read multiple characters from a file in C?
a. `getc()`
b. `putc()`
c. `fscanf()`
d. `fprintf()`

17. Which of the following is used to write the string into the file?
a. `fputs()`
b. `fgets()`
c. `gets()`
d. `puts()`

18. Which of the following statement is not correct?
a. To read from the file, `getc` is used and it deals with the single character.
b. To write into the file, `putc` is used and it deals with the single character.
c. To read from the file, `fscanf` is used and it deals with multiple characters.
d. To write into the file, `fscanf` is used and it deals with multiple characters.

19. Which of the following is not correct for `ferror(fptr)`?
a. This is used to tell the status of the file upto the current position.
b. It will return non-zero integer if an error occurs in the file upto the current position.
c. It will return zero if an error occurs in the file upto the current position.
d. All of the above.

Review Questions

1. What are the different file organizations? Explain them.
2. What are the advantages and disadvantages of sequential file organization?
3. Explain the direct file organization. How it differs from the sequential file organization?
4. Describe the indexed sequential file organization
5. Explain `getc, putc, getchar()` `gets, puts, printf, fprintf, fscanf()`
6. How does an append mode differ from the write mode?
7. Explain the significance of `feof` and `ferror()`
8. What is the importance of file as a data structure? Explain it with an example.
9. What is the difference between sequential and indexed sequential files?
10. What are the factors involved in selection of the file organization technique?

11. What are the different modes in which file can be opened?
12. What are the different ways to open a file?
13. What is the purpose of `g e t` and `p u t` function?
14. What is the difference between the text mode and binary mode while performing the I/O operations?
15. What is the purpose of `r e a d` and `w r i t e` function?
16. What is a command line argument?
17. What is a file mode? Describe the various file mode options available.
18. Describe the purpose of `s e e k g, ( )` `s e e k p t( e) l l g` and `t e l l p ( )`

Programming Assignments

1. Write a program in C to copy the file into another file.
2. Write a program in C to enter characters in the file.
3. Write a program in C to read information from the file.
4. Write a program in C to read specific records from the file.
5. Write a program in C to read and write students' information into the file using data members in a `p r i v` access mode.
6. Write a program in C to read and write the employees' information, that is name, designation, salary and phone numbers using class and `r e a d` and `w r i t e` function with files.
7. Write a menu-driven program in C++ to append, display and modify students information (rollno, age, phnum and height) in a file.
8. Write a menu-driven program in C to append, modify, display all and display current students' information (rollno, name, branch, semester, add, email) in the file using class.
9. Write a menu-driven program in C/C++ to append, display current, display all, search and delete students' information from the file.

Answers

Mul tipl e-Choice Questions

1. (a)
2. (b)
3. (d)
4. (b)
5. (b)
6. (b)
7. (a)
8. (d)
9. (a)
10. (a)
11. (a)
12. (a)
13. (a)
14. (b)
15. (d)
16. (c)
17. (b)
18. (d)
19. (d)

Model Question Paper – 1

Time: Three hours **Maximum Marks: 100**

Note: Attempt any *five* questions. All questions carry equal marks. Make suitable assumptions if necessary.

1. **(a)** Write the properties of AVL tree and give *two* examples of AVL tree. (10)
   (b) A binary tree T has 9 nodes. The in-order and pre-order traversals of T yield the following sequence of nodes:
   In-order: E A C K F H D B G
   Pre-order: F A E K C D H G B
   Draw the tree. (10)

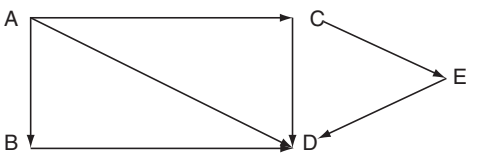
2. **(a)** Consider the following arithmetic expression P, written in postfix notation: (12)
   P: 12, 7, 3, −, 1, 2, 1, 5, +, ∗, +
   (i) Translate P by inspection and hand into its equivalent infix expression.
   (ii) Evaluate infix expression.
   (b) Differentiate between a tree and a graph. Is it possible to connect a graph into tree? If yes, how? (8)

3. **(a)** What do you mean by hashing and what is the hash function? Explain with examples. (10)
   (b) Sort the following integers using insertion sort: (10)
   25, 7, 48, 37, 12, 92, 86, 33

4. **(a)** It is possible to keep two stacks in a single array, if one grows from position ⊥ of the array and the other grows from the last position. Write a procedure PUSH (*x, s*) that pushes element *x* onto stacks where *s* is one or the other of these two stacks. Include all necessary error checks in your precedence. (10)
   (b) Enlist different operations which are normally performed on any linear array. Write the algorithm for any two such functions. (10)

5. **(a)** Explain memory allocation and garbage collection. (10)
   (b) Explain stack and write an algorithm for Push and Pop operation. (10)

6. **(a)** Write a C procedure which implements circular queue. (8)
   (b) Write depth first search algorithm. Also draw linked representation of the following graph: (12)



7. **(a)** Write a procedure which removes the first element of a list and adds it to the end of the list without changing any data field. Also write a procedure which deletes the *k*th element from a linked list. (10)
   (b) Write a program in C which does multiplication of two matrices. (10)

8. Write short notes on any *four* of the following: (5×4=20)
   (a) Abstract data type
   (b) Triangular matrix
   (c) Threaded binary tree
   (d) Priority queue
   (e) Types of data structure

Model Question Paper – 2

Time: Three hours **Maximum Marks: 120**

Note: Attempt any *five* questions. Assume suitable data wherever necessary. All questions carry equal marks.

1. **(a)** What is a data structure? How is data processed on a data structure? (10)
   (b) What is an array? Explain different types of array. (10)

   Or

   (a) What is triangular matrix? (10)
   (b) Explain the necessity of dynamic memory allocation?
   Discuss any *one* method used in dynamic memory management. (10)

2. **(a)** Write an algorithm to insert a node after a given node. (10)
   (b) What is header linked list? (10)

   Or

   (a) Explain the Josephus problem of circular linked list. (10)
   (b) Write an algorithm to insert a node in a doubly linked list. (10)

3. Write the algorithms of the following: (20)
   (a) In-order traversal
   (b) Post-order traversal

   Or

   How do you convert general trees into binary trees? Explain. (20)

4. Discuss the bubble sort and selection sort algorithms. What is the complexity of these algorithms? (20)

   Or

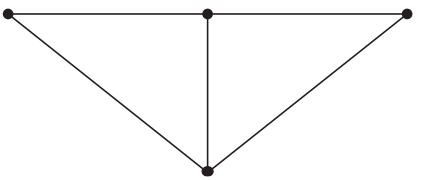
   Define the term heap. Describe the mechanism of heap sort with an example.

5. Write the various graph traversal schemes. (20)

   Or

   Define the term searchability. Also write the shortest path algorithm of the graph.

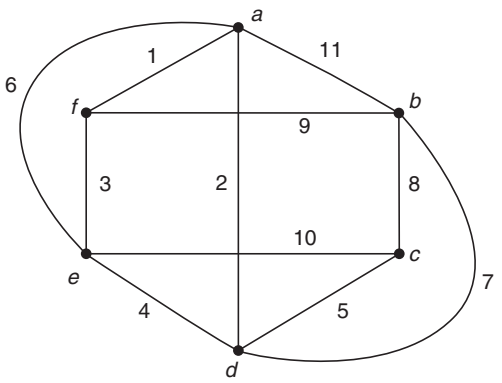
6. **(a)** Find all spanning trees for the following figure. (5)



   (b) Show that the maximum number of edges in simple graph with 'n' vertices is: (5)

$$\frac{n(n-1)}{2}$$

7. **(a)** Determine the minimum spanning tree for the following figure. (10)



Model Question Paper – 3

Time: Three hours **Maximum Marks: 100**

Note: Question paper is divided into five UNITS. Attempt *one* question from each unit. All Questions carry equal marks.

1. **(a)** Discuss the representation of three-dimensional array using sequential mapping. Derive an expression for addressing an arbitary element. (10)
   (b) How is dynamic memory management possible through pointers? Write a program to store n elements in an array using pointer. (10)

   Or

   (a) Write a program in C to match two given strings without using any string manipulation functions. (10)
   (b) Explain the following terms: (10)
   (i) Structured programming
   (ii) Triangular matrix

2. **(a)** Compare singly linked list and doubly linked list. Write an algorithm to insert an element into a doubly linked list. (10)
   (b) What is recursion? What do you mean by base criteria? Write a recursive procedure to ding x^y. (10)

   Or

   (a) What is a sparse matrix? How can a sparse matrix be represented in memory using linked list? Explain. (10)
   (b) What do you understand by multistack? Write insertion and deletion algorithms for n stacks stored in a one-dimensional array of size mn where m is integer. (10)

3. **(a)** Explain the following terms: (8)
   (i) Complete binary tree
   (ii) Full binary tree
   (iii) Strictly binary tree
   (b) What is a binary expression tree? Create a binary expression tree for the following expression and write its pre-and post-order traversals: (6)

   (A**B/C)*D + E + F/G

   (c) What are threaded binary trees? (6)

   Or

   (a) Explain the following terms with examples: (10)
   (i) height balance tree
   (ii) digital search tree
   (b) Write a detailed note on hashing and collision resolution techniques. (10)

4. **(a)** Sort the following sequence using bubble sort and insertion sort algorithm. Write the position of list after each step:

25, 11, 13, 42, 17, 90, 85, 7, 10, 20

(b) Write an algorithm for merge sort. Give example.

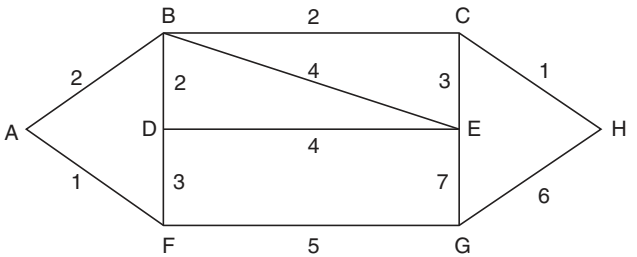
Or

(a) What is quick sort? Why is it called partition exchange sort? Write quick sort algorithm.

(b) What is complexity of an algorithm? How do you determine the complexity of an algorithm?

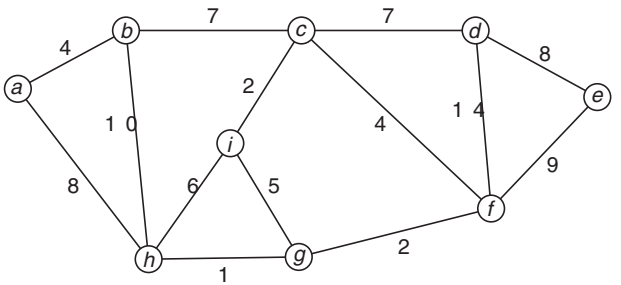
5. **(a)** Compare graph traversal techniques. (10)

(b) Using Prim's algorithm find the minimum spanning tree of the following graph: (10)



Or

(a) Using Dijkstra's algorithm find the shortest path between *a* and *e* in the following figure. (10)



(b) What are the various representations of graph? Explain. (10)

Index

P

Q

R

S

T

U

V

Highlights of the Book

The study of data structure is an essential subject of every graduate and undergraduate program related to Computer Science. Data structures are building blocks of a program as they are the structural representation of logical relationships between data elements. This book offers a solid, effective and easy-to-understand approach to the study of fundamental data structures. Each chapter has a uniform structure in presentation starting with definition, declaration, implementation, operations, types, summary, objective questions, review questions and finally concludes with solutions to objective questions.

This book covers the complete syllabus of most universities with B.Tech/B.E./MCA/PGDCA/BCA/M.Sc/B.Sc disciplines offering a course on "data structures using C and C++", "introduction to data structures and file organization" or "data structures".

Unlike most books that provide the implementation of data structures in C only, this book provides their implementation in both C as well as C++. It also illustrates several examples to amplify the concept of algorithm and implementation, exercise and sample programs and solved problems to prepare the students for the examination.

Precise Series

- Precise content as per syllabi.
- Attractive format.
- Learning objectives at the beginning of each chapter.
- Summary and key terms to quickly review the concepts.
- Focus on explanation of concepts.
- Sufficient examples for easy comprehension.
- Optimum balance of qualitative & quantitative problem set.

About the Book

- Covers the entire syllabi of data structure course for B.Tech/B.E./MCA/PGDCA/BCA/M.Sc/B.Sc disciplines.
- Concepts amply illustrated through examples and tested programs.
- Multiple-choice and review questions of varied difficulty levels and programming assignments (to test) at the end of chapters.
- Brings data structures and their implementation in C and C++ under one umbrella.
- Describes all important data structures and algorithms (language independent form) to process them.
- Objective questions, included at the end of each chapter, useful for competitive examinations like GATE.
- Model question papers included.
- Useful for C and C++ professionals and programmers also.
- Pedagogy:
  - 50+ algorithms.
  - 380+ figures.
  - 50+ programs with output.
  - 60+ solved and explanatory examples.
  - 30+ solved problems.
  - 60+ programming assignments.
  - 230+ objective questions.
  - 260+ review questions.

Visit us at
www.wileyindia.com
www.wileyprecisetextbook.com

READER LEVEL
Undergraduate

SHELVING CATEGORY
Engineering

Wiley India Pvt. Ltd.
4435-36/7, Ansari Road, Daryaganj,
New Delhi-110 002.
Tel: 91-11-43630000.
Fax: 91-11-23275895.
E-mail: csupport@wileyindia.com
Website: www.wileyindia.com

WILEY-INDIA
ISBN: 978-81-265-1997-2