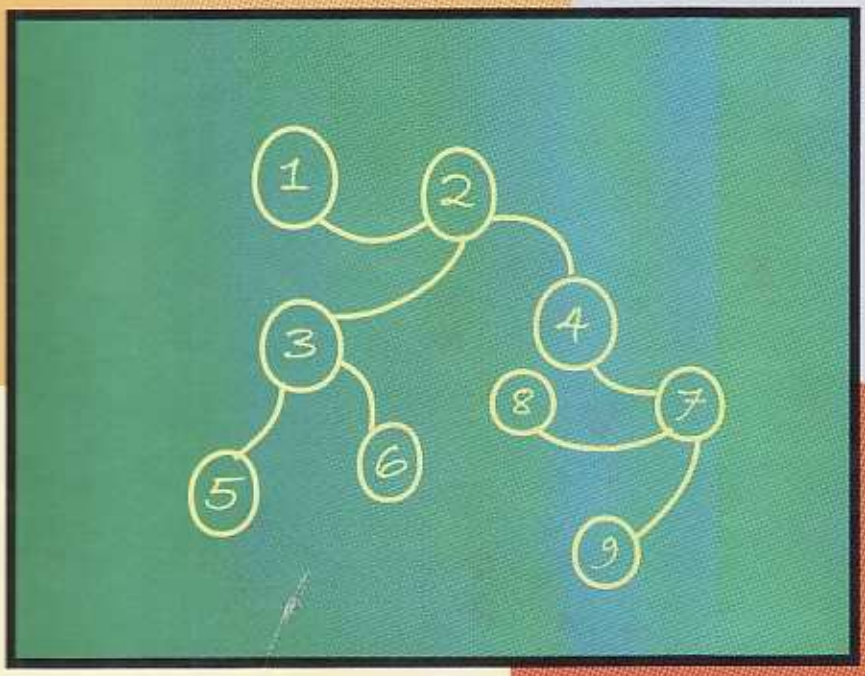
Data Structures Fundamentals

Md. Rafiqul Islam, Ph.D
M. A. Mottalib, Ph.D



Islamic University of Technology (IUT)
A Subsidiary Organ of the OIC

Second Edition

Data Structures Fundamentals

Md. Rafiqul Islam
Dept. of Computer
Science & Engineering
Khulna University
Khulna
Email: dmri1978@yahoo.com
Website: www.cseku.ac.bd

M. A. Mottalib
Dept. of Computer Science &
Information Technology (CIT)
Islamic University of Technology (IUT)
Gazipur
Email: mottalib@iut-dhaka.edu
Website: www.iutoic-dhaka.edu

Islamic University of Technology (IUT)
Board Bazar, Gazipur-1704, Bangladesh

Published by:
Research, Extension, Advisory Services and Publications (REASP)
Islamic University of Technology (IUT)
A subsidiary organ of Organisation of Islamic Cooperation (OIC)
Board Bazar, Gazipur 1704, Bangladesh
Telephone : +8802-9291254~59 Ext. 3215, Fax: +8802- 9291260
E-mail : reasp@iut-dhaka.edu, cit@iut-dhaka.edu
Website : www.iutoic-dhaka.edu

Year of Publication (1st Edition) : 2009
Year of Publication (2nd Edition) : 2011

ISBN: 978-984-33-0384-4

Price: Tk. 300
US$ 12

Contact for Ordering:
Librarian
Islamic University of Technology (IUT)
Board Bazar, Gazipur 1704, Bangladesh
Telephone : +8802-9291254~59 Ext. 3235, Fax: +8802- 9291260
E-mail : library@iut-dhaka.edu, cit@iut-dhaka.edu

Cover page & Graphics : A. K. M. Shahidur Rahman
Sr. Assistant Secretary
CIT Department, IUT

Printed by : **M/s Nishan Computer & Printing**
38, Bangla Bazar (2nd Floor)
Sadar Ghat, Dhaka

Dedicated to my father and father-in-law who died while I was doing my PhD.

---- Md. Rafiqul Islam

Dedicated to my beloved eldest son Zisan Mashroor Hrid, my father and father-in-law who have left this world for good.

---- Md. Abdul Mottalib

Message

I am pleased to note that the first edition of the text book **Data Structures Fundamentals** has already been exhausted. It proves that the book had good demand among the students. I appreciate that realizing the demand of the book the authors have revised it, updated it and steps have been taken for bringing out a second edition. I congratulate the authors for their meaningful initiative.

At this point, it is worth mentioning that IUT is very conscious about quality and it gives particular importance on the quality of education. Our efforts are always directed towards continuous updating of syllabi by including new topics to familiarize the students with the contemporary knowledge. I am confident that the authors given due consideration to this idea.

Although the book was reviewed before the publication of the first edition, there would always remain some scope for improving its quality both in terms of its contents and manner of expression. In my view, the best way to address such problems is to get comments from the users of the book. As such I hope that the authors have tried to get the comments of the users from all possible sources and all constructive suggestions have been incorporated in this edition. Still there might be some points left, which could be further developed. I would therefore, invite the future users to come up with their views and contribute for further improvement of the future editions.

Finally, I sincerely hope that this edition will be more acceptable to the users than the last edition; it will be more useful to them and meet most of their demands. Under the situation, I will also feel that the cooperation extended by IUT in publishing the book has been worthwhile.

Hossain 31.07.11.

Prof. Dr. M. Imtiaz Hossain
Vice-Chancellor, IUT

PREFACE

This book describes **data structures**, structural organization of data and operation on them. Now-a-days as computers become very faster, the need of processing large amount of data becomes more acute. And large amount of data can be easily organized and managed using data structures. The book is written using easy language and every data structure is described as briefly as possible, so that the students get the knowledge about data structure using short material. In each chapter we have identified a particular data structure with respect to its concept and graphical representation. The storing process (method) of data using structure in computer's memory has been described. Then we have discussed about the operations on the data structures. The main three operations are *addition or insertion, searching or finding location and deletion*. The operational process on data structures are shown using pictorial views and corresponding algorithms have been described and written in. The algorithms written here are very easy and these will be helpful to the students to build their foundation for algorithm course.

Other than the data structure, the book provides two chapters like searching and sorting, and hashing where we have discussed the methods of using data structures. In searching and sorting chapter we have provided the analysis of the algorithms, which gives knowledge about the estimation of the execution time of algorithms. That means, the students will get introductory idea on analysis of algorithms from here. At the end of each chapter the book is provided with questions.

In this second edition we have done necessary corrections of the first edition and major modifications have been done in chapter 4 (linked list), chapter 7 (tree) and chapter 10 (hashing). Each chapter except one is provided with the problem sets for the practical (lab) classes.

The book has additional three chapters given in Appendix. One is data structures in Java and another is data structures in C#. These two chapters give the idea of writing algorithms as well as programs using data structures in Java and C#. The third one is for practical issues, where codes of some programs using different types of data structures are given. The programs are written in C++ programming language.

This book is suitable for the undergraduate students, who need to study data structures as a foundation course. The book is required for the curricula of Bachelor of Computer Science and Engineering (CSE), Diploma, Higher Diploma and Postgraduate Diploma in Computer Science/Information Technology related courses. The book is very useful to the students who have the foundation on C/C++.

The authors are indebted to Mr. H. M. Mehedi Hasan and Pritish Kumar Ray, who help us a lot to write and prepare some part of the manuscript of this book. We wish to thank Mr. Md. Jahidul Islam, Kasif Nizam Khan and S. M. Mohidul Islam (who were the students of Computer Science and Engineering Discipline of Khulna University, Khulna) for their suggestions in enriching the writing and critical review of the manuscript. The authors also wish to thank Mr. C. M. Mofassil Wahid, Assistant Professor and Mr. Akramul Azim, Lecturer of Computer Science & IT (CIT) Dept., IUT for their help in preparing the Appendix A on Java and Appendix B on C# respectively. We are also thankful to unanimous reviewers of the manuscript of the book, for their suggestions in improving the material and organization of the book.

Dr. Md. Rafiqul Islam
Dr. M. A. Mottalib

TABLE OF CONTENTS

CHAPTER ONE
BACKGROUND

OBJECTIVES:

- Identify data structure
- Identify algorithm
- Identify program
- Describe the importance of data structure
- Identify complexity of algorithm

1.1 Data Structure

In "Data Structure" there are two words, one is data and another is structure. Data means raw facts or information that can be processed to get results or products.

Elementary items constitute a unit and that unit may be considered as a structure. As for example, some elementary items like hand, leg, eye, ear, nose, bone and some others constitute a human body. So, human body is a structure. Similarly some elementary items like pieces of wood, iron, raxine etc may constitute a chair, which is also a structure. A structure may be treated as a frame or proforma where we organize some elementary items in different ways. Like structures in our environment, data structure is also constituted with some elementary data items.

Data structure is a structure or unit where we organize elementary data items in different ways and there exists structural relationship among the items. That means, a data structure is a means of structural relationships of elementary data items for storing and retrieving data in computer's memory. Usually elementary data items are the *elements* of a data structure. However, **a data structure may be an *element* of another data structure.** In other words a data structure may contain another data structure.

Example of Data Structures: Array, Linked List, Stack, Queue, Tree, Graph, Hash Table etc.

Types of elementary data item: Character, Integer, Floating point numbers etc.

Expressions of elementary data in C/C++ programming language are shown below:

Elementary data item	Expression in C/C++
Character	char
Integer	int
Floating point number	float

1.2 Operations on Data Structure

We can also perform some operations on data structure such as insertion (addition), deletion (access), searching (locate), sorting, merging etc.

1.3 Algorithm

It is a set or sequence of instructions (steps) that can be followed to perform a task (problem). To write an algorithm we do not strictly follow grammar of any particular programming language. However its language may be near to a programming language. Each and every algorithm can be divided into *three sections*. First section is *input* section, where we show which data elements are to be given. The second section is very important one, which is *operational* or *processing* section. Here we have to do all necessary operations, such as computation, taking decision, calling other procedure (algorithm) etc. The third section is *output*, where we display the result found from the second section.

1.4 Program

A program is a set or sequence of instructions of any programming language that can be followed to perform a particular task. For a particular problem, at first we may write an algorithm then the algorithm may be converted into a program. In a program usually we use a large amount of data. Most of the cases these data are not elementary items, where exists structural relationship between elementary data items. So, the program uses data structure(s). Like an algorithm, a program can be divided into three sections such as input section, processing section and output section.

1.5 Importance of Data Structure

Computer science and computer engineering deal with two jargons which are *software* and *hardware*. Without software, hardware (electrical, mechanical, electronic parts of computer that we see and touch) is useless. So, study of software is very important in computer science, and software consists of programs, which use different types of data. In a program we not only use elementary data items but also use different types of organized data. In other words we use data structure in a program. As we know we write programs to solve problems. That means to solve problems we have to use data structures. The different data structures give us different types of facilities. If we need to store data in such a way that we have to retrieve data directly irrespective of their storage location, we can get this facility using one type of data structure such as *array* gives us such facility. In some cases, instead of direct access, we may need efficient use of memory and this can be performed using *linked list*. In our daily life we handle list of data such as list of students, list of customers, list of employees etc. However each of these entities (student, customer, employee etc.) may have different attributes. As for example, a student has roll number, name, marks attributes and these are different in types. But, how to organize them so that we can handle different types of data as a unit. We can get this facility from *record* or *structure*. Thus importance of data structures is many folds in storing, accessing, arranging data. By achieving the knowledge of data structure we can use different types of data in programs that are used to solve various problems required in our life. Without knowledge of data structures we are not be able to solve problems where we must use them. In programming language there are provisions to use different types of data structures, so that we can organize data in different ways and solve the problem properly. In fact, we can optimize the amount of memory by using proper data types. In other words, without knowledge of data structures we will not be able to write program properly, hence we will not be able to solve problem. Therefore, for the students and teachers of computer science and engineering the knowledge of data structure is very much essential.

1.6 Complexity of Algorithm

There are two types of complexities: One is **time complexity** and another is **space complexity.**

Time complexity: This complexity is related to execution time of the algorithm or a program. It depends on the number of element (item) comparisons and number of element movement (movement of data from one place to another). However, the complexity of the most of the algorithms described here related to the number of element comparisons. So, the complexity of the algorithm is computed with respect to the total number of element (item) comparisons needed for the algorithm.

Space complexity: This complexity is related to space (memory) needs in the main memory for the data used to implement the algorithm for solving any problem. If there n data items used in an algorithm, the space complexity of the algorithm will be proportional to n.

The complexity of an algorithm (either time complexity or space complexity) is represented using asymptotic notations. One of the asymptotic notations is O (big-oh) notation. In general we write $T(n) = O(g(n))$ if there are positive constants C and n. Such that $T(n) \leq cg(n)$ for all n, $n \geq n_0$. In words the value of T(n) always lies on or below cg(n) for $n \geq n_0$. In this book we have represented the complexity using O notation. Big-oh (O) notation is also called upper bound of the complexity. If we get the total number of element comparisons is $\frac{1}{2} n^2 - \frac{1}{2} n$, then we can write it as $O(n^2)$. Since $(\frac{1}{2} n^2 - \frac{1}{2} n) < n^2$. Similarly $10n^2 + 4n + 2 = O(n^2)$. Since $10n^2 + 4n + 2 \leq 11n^2$.

Table-1: Shows the advantages and disadvantages of the various data structures:

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known	Slow search, slow deletion, fixed size
Linked list	Quick insertion, quick deletion	Slow search
Stack	Provides last-in, first-out (LIFO) access	Slow access to other items
Queue	Provides first-in, first-out(FIFO) access	Slow access to other items

Data Structure	Advantages	Disadvantages
Binary tree	Quick search, insertion, deletion (if tree remains balanced)	Deletion algorithm is complex
Hash table	Very fast access if key known, Fast insertion	Slow deletion, access slow if key not known, inefficient memory usage.
Heap	Fast insertion, deletion	Slow access to other items
Graph	models real-world situations	Some algorithms are slow and complex

Summary:

Data structure is a structure or unit where we organize date items in different ways and there exists structural relationships of them (data items).

Algorithm is a set or sequence of instructions (steps) that can be followed to perform a particular task (problem).

Program is a set or sequence of instructions of a particular programming language that can be followed to perform a particular task.

Data structure is the most important building block of a program. It facilitates organized storage and easy retrieval of data.

There are two types of complexity-the time complexity and the space complexity. **Time complexity** is related with the number of element comparisons and element movement. **Space complexity** is used to determine the memory space usage and requirements for a particular problem.

Questions:

1. What do you mean by data structure?
2. What are the objectives of learning data structure?
3. What are elementary data types?
4. What do you mean by space complexity?
5. Define data structure. Give examples.
6. State two complementary goals of the study of data structure.
7. Data structure is a structure that may contain another data structure. Explain the statement with example.
8. Define elementary data type and data structure.
9. What do you mean by a data structure? Explain the basic operations that are normally performed on a particular data structure.
10. Differentiate between atomic data type and structured data type.
11. What is the difference between an algorithm and a program?

CHAPTER TWO

ARRAY

OBJECTIVES:

- Identify array
- Show data storing and accessing methods using array
- Write algorithms using array
- Identify two-dimensional array
- Show data storing and accessing processes using two-dimensional array
- Describe the process of representation of two-dimensional array in computer's memory
- Write algorithms using two-dimensional array

ARRAY

2.1 Definition

An array is a finite set of same type of data items. In other words, it is a collection of homogeneous data items (elements). The elements of an array are stored in successive memory locations. Any element of an array is referred by *array name* and *index number* (subscript). There may have many dimensional arrays. But usually two types of array are widely used; such as one dimensional (linear) array and two dimensional array.

2.2 One Dimensional Array

2.2.1 Definition

An array that can be represented by only one dimension such as row or column and that holds finite number of same type of data items is called one dimensional (linear) array.

	1	2	3	4	5	6	7	8	9	10
Array B →	0	10	12	13	19	20	18	23	29	39

Figure-2.1: Graphical representation of one dimensional array.

Here 1, 2, 3, … … …, 10 are index number, and 0, 10, 12, … … …, 39 are data items or elements of the array and B is the array name. Symbolically an element of the array is expressed as B_i or B[*i*], which denotes *i*th element of the array, B.

Thus B[4], B[9] denotes respectively the 4th element and the 9th element of the array, B.

The name of the array usually is a name constituted by one or more characters. Thus array name may be *A*, *S*, *Stock*, *Array1* etc. The element of an array may be number (integer or floating point number) or character.

Expression of one dimensional array in C/C++:

For integer array: int a[10];

For character array: char b[30];

For floating point array: float c[10];

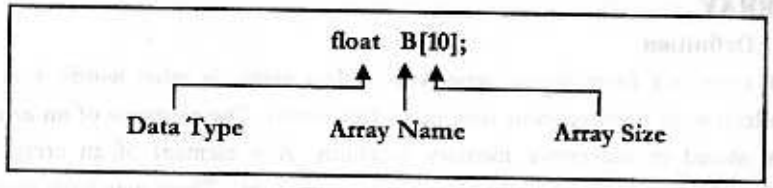


Figure-2.2: Declaration of Array in C/C++.

2.2.2 Store an element into an array

B[4] = 19; it means 19 will be stored in the cell number 4 of the array of *B* in the Figure-2.1. If there is any (previous) value that will be overwritten.

2.2.3 Read (retrieve) a value (element) from an array

x = *B*[6]; it means the value of *x* will be 20, since the cell number 6 of the array, *B* contains 20 (see Figure-2.1).

The above expressions are as like as C/C++.

2.2.4 Code in C/C++ for storing data in an array

```
int x[10];
for (i = 0; i < 10; ++i)
        scanf ("%d", &x[i]);
```

We can store integer type of data to the array, *x* using above segment of codes.

Code in C/C++ for reading data from an array and the data will be displayed on the monitor's screen:

```
for (i =0; i < 20; ++i)
        printf ("%d", x[i]);
```

Problem 2.1:

Given a list of elements, write an algorithm to store the list of elements (numbers) in an array and find out the largest element of the list.

Algorithm 2.1: Algorithm to search the largest element of a list

1. Input: x[1 . . . n];
2. for ($i = 1$; $i \leq n$; $i = i + 1$)
   store data to x[*i*];
3. *large* = x[1];
4. for ($i = 2$; $i \leq n$; $i = i + 1$)
   if (x[*i*] > *large*), *large* = x[*i*]; // that means if any element larger than the previous upgrade ***large***
5. Output: the largest number (print the value of ***large***)

Comments: *x* is the array to store data and *n* is the size of the list. The above code is as like as C/C++, but not exactly written in C/C++. We shall follow this style for other algorithms.

Problem 2.2:
Given a linear array with data, find out a particular (specific) element of x from the array. We do not know the index (cell) number where the element has been stored.

Algorithm 2.2: Algorithm to search a particular element from a list

1. Input: A set of data in array a, and variable x *i.e.*, the target element
   $a[1 \ldots n], x;$
2. found = 0
3. for ($i = 1$; $i \leq n$; $i = i + 1$)
   {
   if ($a[i] == x$)
   location = i, found = 1, break;
   }
4. Output: if (found = = 1), print "FOUND" message and location.
   else print "NOT FOUND" message.

Problem 2.3:
Given a list of integers stored in a linear array. Find out the summations of odd numbers and even numbers separately.

Solution: Given a list of integers, we have to find out the odd numbers and then we shall add those odd numbers. Similarly, we shall find out the even numbers in the list and adding those numbers we shall get the summation of even .

To store the results, we require two variables; *sum_even* and *sum_odd*. Initially, values of these variables will be zero (0) and every time we find an even number we shall add it to the *sum_even* and we find an odd number, we shall add it to the *sum_odd*. If a number is divisible by 2 it is even, otherwise odd.

We have to start from the first number of the list. If it is even, it will be added with the sum_even and if it is odd, it will be added with the sum_odd. Similarly, we shall access whole list one by one and add them with either sum_even (if a number is even) or sum_odd (if a number is odd).

Algorithm 2.3: Algorithm to find the summations of even and odd numbers

1. Input: An array and variables (to store the results of summations)
   $A[1...n]$, $sum\_odd = 0$, $sum\_even = 0$;
2. for ($i = 1$; $i \leq n$; $i = i + 1$)
   {
   if ($A[i]\%2 == 0$), $sum\_even = sum\_even + A[i]$;
   else $sum\_odd = sum\_odd + A[i]$;
   }
3. Output: Summation of odd numbers (print *sum_odd*) and summation of even numbers (print *sum_even*)

Comments: Here A is an array that holds a list of integers and n is the size of the list. *sum_odd*, *sum_even* are two variables to store the summation of odd numbers and the summation of even numbers respectively.

Problem 2.4:
Given a list of integers stored in a linear array, find out the summations of numbers in odd indices and even indices separately.

Solution: This problem is similar to the problem 2.3. Here the difference is that, we have to check whether the index is odd or even.

Algorithm 2.4: Algorithm to find the summations of even and odd indexed numbers

1. Input: An array and variables (to store the results of summations)
   $A[1...n]$, $sum\_odd = 0$, $sum\_even = 0$;
2. for ($i = 1$; $i \leq n$; $i = i + 1$)
   {
   if ($i\%2 == 0$), $sum\_even = sum\_even + A[i]$;

else *sum_odd* = *sum_odd* + *A*[*i*];
}

3. Output: Summation of numbers in odd indices (print *sum_odd*) and summation of numbers in even indices (print *sum_even*)

Comments: Here *A* is an array that holds a list of integers and *n* is the size of the list. *sum_odd*, *sum_even* are two variables to store the summation of numbers at odd positions and the summation of numbers at even positions respectively.

Problem 2.5:
Given a list of integers stored in a linear array and a data element, insert the element into the array at a given position.

Algorithm 2.5: Algorithm to insert an element into an array.
1. Input: An array *A*[1...*n*], the position of insertion *m* and the data *x*.
2. Increase the size of the array, *A*[1...*n* + 1]
3. for (*i* = *n*; *i* ≥ *m*; *i* = *i* - 1)
   A[*i* + 1] = *A*[*i*];
4. *A*[*m*] = *x*;
5. Output: The array, *A* with size *n* + 1.

Comments: Here it is assumed that all the cells of the array, *A* contains data. So the data from the positions *m* to *n* of *A* are copied (transferred) to the positions *m+1* to *n+1* of the array *A*. Now the position *m* of the array *A* is empty and the element to be inserted in placed at the position.

Problem 2.6:
Given a list of integers stored in a linear array, delete a data from a given position of the array.

Algorithm 2.6: Algorithm to delete an element from an array.
1. Input: An array *A*[1...*n*], and the position of deletion, *m*.
2. for (*i* = *m*; *i* < *n*; *i* = *i* + 1)
   A [*i*] = *A* [*i* + 1];
3. Output: The updated array, *A*.

Comments: Here the data of the positions *m* to *n-1* are overwritten by the data of the positions *m* + *1* to *n* of the array, *A*. Now the position *n* of the array is empty.

Problem 2.7:
Given two linear arrays of integers, merge the two arrays into a single array.

Algorithm 2.7: Algorithm to merge two arrays.
1. Input: Two arrays *A* [1...*m*] and *B*[1...*n*]
2. Take an array *M* [1... *m* + *n*]
3. for (*i* = 1; *i* <= *m*; *i* = *i* + 1)
   M[*i*] = *A*[*i*];
4. for (*i* = 1; *i* <= *n*; *i* = *i* + 1)
   M[*i*+*m*] = *B*[*i*];
5. Output: The merged array, *M*

Comments: Here the output array is *M*. For merging the data of the array, *A* are copied to the positions *1* to *m* of *M* and the data of the array, *B* are copied to the positions *m* + *1* to *m* + *n* of *M*.

2.3 Two Dimensional Array

2.3.1 Definition

Two dimensional array is an array that has two dimensions, such as *row* and *column*. Total number of elements in a two dimensional array can be calculated by multiplication of the number of rows and the number of columns. If there are *m* rows and *n* columns, then the total number of elements is *m* × *n*, and *m* × *n* is called the *size* of the array. Of course, the data elements of the array will be same type. In mathematics, the two dimensional array is called a *matrix* and in business it is called *table*. A two dimensional array can be expressed as follows:

A_{ij} or A[*i*, *j*] for 1 ≤*i* ≤m and 1 ≤*j* ≤*n* (where *m* and *n* are the number of rows and columns respectively, see Figure-2.3).

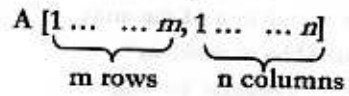
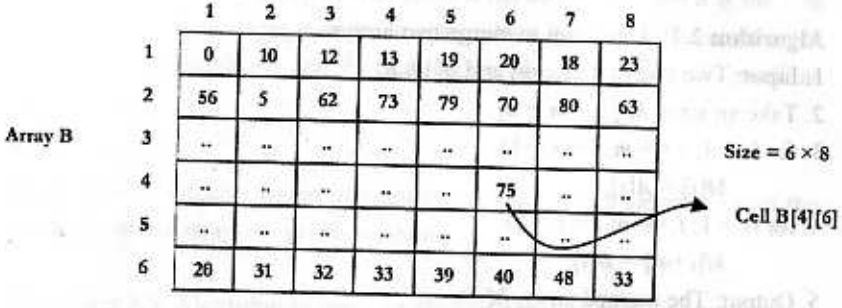


Figure-2.3: Symbolic representation of two dimensional array.



	1	2	3	4	5	6	7	8
1	0	10	12	13	19	20	18	23
2	56	5	62	73	79	70	80	63
3	..	..	..	..	..	..	..	..
4	..	..	..	..	..	75	..	..
5	..	..	..	..	..	..	..	..
6	20	31	32	33	39	40	48	33

Figure-2.4: Graphical representation of two dimensional array

Two dimensional array can be expressed in C/C^{++} as follows:

int A[*m*][*n*]; // Here *m* is the number of rows and *n* is the number of columns [see Figure-2.5]

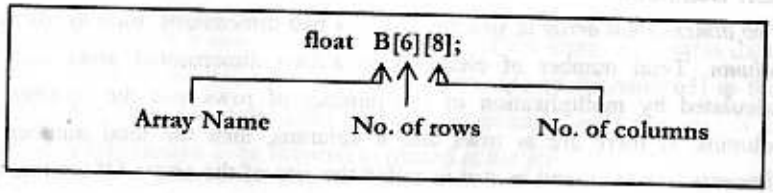


Figure-2.5: Declaration of two dimensional array in C/C++

Size of this array is 6 × 8. So, total number of elements or items is 6×8=48.

2.3.2 Store and retrieve values in and from array

Data elements can be stored in a two dimensional array using loop or directly as shown below:

```
int B[7][3];
for (int i = 0; i < 7; ++ i)
{
  for (int j = 0; j < 3; ++ j)
        scanf ("%d", &B[i][j]);
        }
```

//storing data taken from keyboard

Or,

```
int B[7][3] =  {
        { 1,  2,  3},
        { 9, 10, 11},
        ... .... ....,
        ... .... ....,
        ... .... ....,
        {22, 25, 40}
        };
```

//Direct insertion of elements

Figure-2.6: Program segment in C/C++ to store data in a two dimensional array.

B[3][5] = 75; which means 75 will be stored in the cross section of the 4th row and 6th column (see Figure-2.4).

On the other hand, *x* = B[5][2]; here the value of B[5][2] is assigned to x. So, the element in the position of 6th row and 3rd column will be assigned to the variable, *x*.

2.3.3 Two dimensional array representation in memory

The elements of a two dimensional array are stored in computer's memory ***row by row*** or ***column by column***. If the array is stored as *row by row*, it is called ***row-major order*** and if the array is stored as *column by column*, it is called ***column-major order***. Suppose that there is a two-dimensional array of size 5 × 6. That means, there are 5 rows and 6 columns in the array.

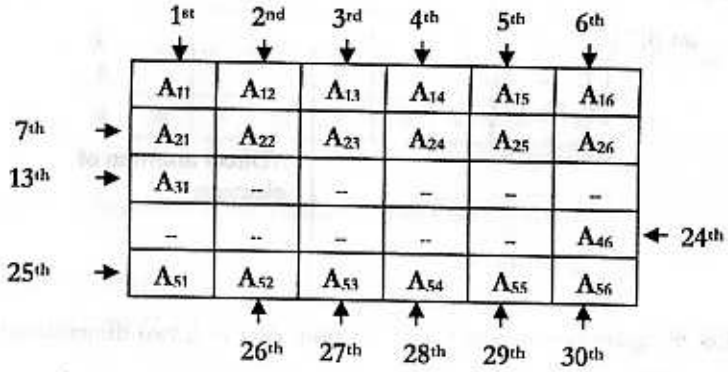
In *row-major order*, elements of a two dimensional array are ordered as –

$\underbrace{A_{11}, A_{12}, A_{13}, A_{14}, A_{15}, A_{16}}_{\text{row 1}}, \underbrace{A_{21}, A_{22}, A_{23}, A_{24}, A_{25}, A_{26}}_{\text{row 2}}, A_{31}, \ldots, \underbrace{A_{46}, A_{51}, A_{52}, \ldots\ldots, A_{56}}_{\text{row 5}}$

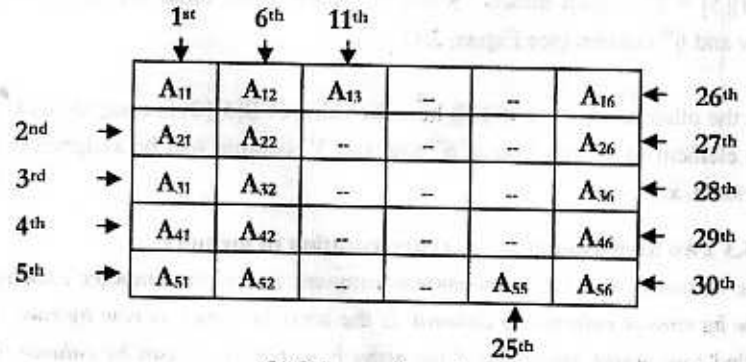
and in *column-major order*, elements are ordered as –

$\underbrace{A_{11}, A_{21}, A_{31}, A_{41}, A_{51}}_{\text{column 1}}, \underbrace{A_{12}, A_{22}, A_{32}, A_{42}, A_{52}}_{\text{column 2}}, A_{13}, \ldots\ldots\ldots, A_{55}, \underbrace{A_{16}, A_{26}, \ldots\ldots, A_{56}}_{\text{column 6}}$

Figure-2.7: shows these arrangements in details. Here, 1st, 2nd, 3rd etc. denote the position number (cell number) of the elements of the array.



(a) Row-major order



(b) Column-major order

Figure-2.7: Pictorial view of two dimensional array representation in memory

2.3.4 Location of an element of a two-dimensional array

***Row-major Order*:**

If Loc (A[*i*, *j*]) denotes the location in the memory of the element A[*i*][*j*] or A_{ij}, then in row-major order –

Loc (A[*i*, *j*]) = Base (A) + (*n* (*i* - 1) + (*j* - 1)) * w;

Here Base (A) is starting or base address of the array A, *n* is the number of columns and w is the width of each cell, *i.e,* number bytes per cell.

***Column-major Order*:**

In column-major order,

Loc (A[*i*, *j*]) = Base (A) + (*m* (*j* - 1) + (*i* - 1)) * w;

Here Base (A) is starting or base address of the array A, *m* is the number of rows and w is the cell width.

Example:

Base address, Base (A) = 100, Size of the array = 5 × 6. If the type of array is integer then find out Loc (A[4, 3]).

Solution:

If the array is stored in row-major order:

Loc (A[4, 3]) = Base (A) + (*n* (*i* - 1) + (*j* - 1))* 2 (2 bytes for each integer cell in C/C++)

= 100 + (6 × 3 + 2)* 2

= 100 + 40

= 140

If the array is stored in memory in column-major order:

Loc (A[4, 3]) = Base (A) + *m* (*j* - 1) + (*i* - 1)* 2

= 100 + (5 × 2 + 3)* 2

= 100 + 26

=126

Problem 2.8:

Given a two-dimensional array, find out the summation of the boundary elements of the array. Here no element should be added twice.

Problem 2.10:

Given a two dimensional array, find out the summation of diagonal elements of the array.

Algorithm 2.9: Algorithm to find out summation of diagonal elements

1. Input: a two dimensional array

   B[*1* . . . *n*, 1...*n*]

   sum = 0;

2. Find each diagonal element and add them with *sum*.

   {for (*i* = 1; *i* ≤*n*; *i* = *i* + 1)

   for (*j* = i; *j* ≤n; *j* = *j* + 1)

   if (*i* = *j* || *i* + *j* = *n* + 1), *sum* = *sum* + B[i, j]

   }

[Note: Diagonal elements are those elements whose indices are equal (*i.e.*, *i* = *j*) or their summation results n +1 (*i.e.*, *i* + *j* = *n* +1).]

3. Output: Print ***sum*** as the result of summation of diagonal elements.

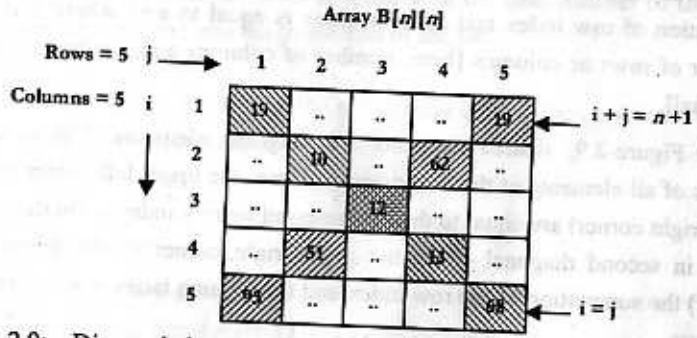


Figure-2.9: Diagonal elements of a two-dimensional array.

Problem 2.11:

There are 40 students in class. They have written 4 class tests of a course. Find out the average of the best 3 class tests for each student.

Solution: First, we compute the summation of marks of 4 class tests for each student. Then we find minimum marks among those 4 class tests. By subtracting the minimum marks from the summation and dividing by 3, we get average for each student.

Algorithm 2.10:

1. Input: a two-dimensional array (to store the marks of class tests of each student) and a one-dimensional array (to store the average of marks of each student)

   marks[1..40, 1..4], *avg_mrk*[1...40]

2. Add all class test marks for each student

   for (*i* = 1; *i* ≤40; *i* = *i* + 1) // first loop starts

   {

   sum = 0;

   min_mrk = *marks*[i,1];

   for (*j* = 1; *j* ≤4; *j* = *j* + 1) //second loop starts

   {

   sum = *sum* + *marks*[i,j];

3. Find out the minimum (worst) class test marks

   if (*min_mrk* > *marks*[*i*][*j*]), *min_mrk* = *marks*[i,j];

   } // end of the second loop

4. Compute the average marks of best three class tests.

   avg_mrk[*i*]= (*sum* − *min_mrk*)/3;

   } // end of the first loop

5. Output: Average of best three class test marks, *avg_mrk*[40];

Comments: Here ***marks*[40, 4]** is a two-dimensional array that holds four class test marks for each of the 40 students and ***avg_mrk*[40]** is an array to store the computed average marks of best three class tests. ***sum*** is a variable to store the summation of all (4) class test marks and ***min_mrk*** is a variable to store the lowest mark of four class tests for each student.

Solution: First, we have to identify the boundary elements. In a two dimensional array, elements of first column and the last column and the first row and last row are the boundary elements as shown in the Figure-2.8.

Here the index, i represents the row number and j represents the column-number. When i is 1, the row is the first row and when i is m (where m represents the number of rows in the array), the row is the last row. Similarly, when $j = 1$, the column is the first column and the index of the last column is $j = n$. So, a number in a two-dimensional array is a boundary element if $i = 1$, $i = m$, $j = 1$ or $j = n$.

We shall start from the first number of the array. If it is a boundary element, the number will be added to the sum (which is a variable to store the result and initially it is set zero). We shall check every number whether it is a boundary element or not, if the number is a boundary element it will be added to sum. Otherwise, we shall proceed with the next number of the list (array) and continue the process to the end of the list.

Algorithm 2.8: Algorithm to find the summation of boundary elements

1. Input: a two-dimensional array

   A[1...m, 1...n]

   sum = 0;

2. Find each boundary element

   for (i = 1; $i \leq m$; $i = i + 1$)

   for (j = 1; $j \leq n$; $j = j + 1$)

   if (i = 1 || j = 1 || $i = m$ || $j = n$), *sum* = *sum* + A[i, j],

[Boundary elements are those elements whose index $i = 1$ or $j = 1$, and those whose index $i = m$ or $j = n$) and add it with *sum* (previous result)].

3. Output: Print *sum* as the result of summation of boundary elements.

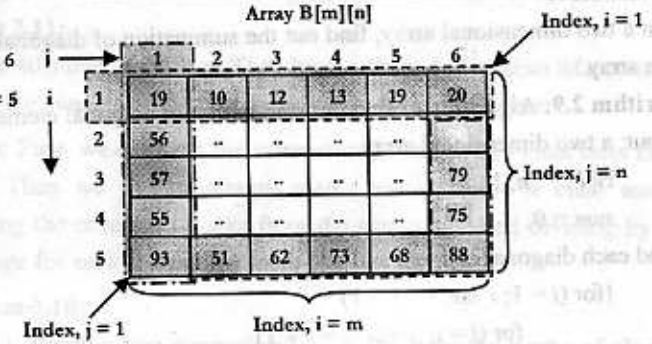


Figure-2.8: Boundary elements of a two-dimensional array in a pictorial view

Problem 2.9:

Given a two-dimensional array find out the summation of the diagonal elements of the array.

Solution: First, we have to identify the diagonal elements. A diagonal element is one, whose either row index and column index are equal or the summation of row index and column index is equal to $n+1$ where n is the number of rows or columns [here, number of columns and number of rows are equal].

In the Figure-2.9, shaded elements are diagonal elements. The column indices of all elements of the first diagonal (from the upper left corner to the lower right corner) are equal to their corresponding row indices. On the other hand, in second diagonal (from the upper right corner to the lower left corner) the summation of the row index and the column index is $n + 1$ for all elements.

We shall start from the first number of the array and check, whether it is a diagonal element or not. If it is a diagonal element, it will be added to the sum, which is a variable initially set zero to store the result of summation of all diagonal elements. We shall advance to the next element and one by one we shall go through the whole list.

If the value of n is an odd number, then the middle number of each of the two diagonals will be common to both diagonals. So, this number will be added twice. So, this number should be subtracted from final summation

Note: For each time, to calculate the summation of all class test marks and to identify the lowest class test ***mark*** of a particular student the variables **sum** and ***min_mrk*** are initialized within the loops.

Summary:

Array is a collection of homogeneous data items. The array, which is represented by only one dimension (row or column) is called **one dimensional** array, whereas an array represented by two dimensions such as row and column is known as **two dimensional** array.

Questions:

1. What is linear array? Explain with example.
2. Write an algorithm to find out a particular item from a given list of items stored in a linear array.
3. Write an algorithm to insert an item in a particular position of an array.
4. Given a list of elements stored in an array, write an algorithm to find out the largest element from the array.
5. You are given an array of sorted data in ascending order. Write an algorithm to rearrange the elements of the array in descending order without using any sorting algorithm and extra data structure.
6. What is a linear array? Write an algorithm to delete an item from a linear array.
7. Define two-dimensional array.
8. Describe how a two-dimensional array can be stored in computer memory.
9. How address of a particular element of a two dimensional array can be computed? Explain.
10. Write an algorithm to find out the summation of the diagonal elements of a two dimensional array.
11. Given an array of size 35 × 40. The base address of the array is 1000. Find the address of A[18][32]. Write down the formula first, then calculate.
12. Suppose there is an array A of size 40×30. The base address of the array is 100. Calculate the address of A[23][15].

13. Given a two dimensional array with m × n size. Write an algorithm to find out summation of the boundary elements, where no element will be added twice.
14. Describe when a two-dimensional array is required to use in programming or Algorithm.
15. Write an Algorithm to print out the following triangle.

```
1
2  3
4  5  6
7  8  9  0
1  2  3  4  5
```

16. What is string? Write a procedure to count the number of a given substring occurs in a given string.

Problems for Practical (Lab) Class

Array related problems

Problem 2-1: Write a program to enter 10 integers in a one dimensional array and display (print) the data on the screen.

Problem 2-2: Write a program to store 10 integers using an array and find out the summation and average of the numbers. Display the numbers, their sum and average (with points) separately.

Problem 2-3: Write a program to find out the largest and smallest of a given list of numbers in an array. Display the numbers, their largest and smallest separately.

Problem 2-4: Write a program to find out the summation of even numbers and odd numbers from a given list of numbers in an array. Display the numbers, the summation of even numbers and odd numbers separately.

Problem 2-5: Write a program to find out the summations of the numbers stored in even indices and odd indices **of an array**. Display the numbers, the two summations separately.

Problem 2-6: Write a program to merge two arrays (merge two arrays and make single one). Display the data of the three arrays.

Problem 2-7: Given an array with data in all positions. Write a program to insert an item in a particular position of an array in such a way that no data will be lost and data will be in previous order. Hints: insert an element, x in *i*th position where $i < n$ and n is the size of the array. Increase the size of the array, shift all the elements starting from position i, and insert the element in free position.

Problem 2-8: Given two arrays of same sizes with data (integers) arranged in ascending orders. Create third array that will contain the data of the first two arrays and the data will be arranged in ascending order. Condition: write the program without using any sorting procedure. Hints: compare and merge (write) in third array without using any sorting procedure.

Problem 2-9: write a program to find out the summation of boundary elements of a two-dimensional array.

Problem 2-10: write a program to find out the summation of diagonal elements of a two-dimensional array.

Problem 2-11: There are 10 students in a course. They have written 4 quizzes for the course. Write a program to find out the average of best three quizzes for each student of the course.

CHAPTER THREE
RECORD

OBJECTIVES:

- Identify record
- Show data storing and accessing processes using record
- Write algorithms using record
- Differentiate the array and the record

RECORD

3.1 Definition

It is a collection of non-homogenous (different types of) related data items. Each item of a record is called a field or attribute. Related data items of a person may constitute a record. The following data items may constitute a record.

Data Items	Types	Length
Person's ID	Numeric	6 digits
Person's Name	Character string	40 characters
Telephone No.	Numeric	9 digits
Due	Numeric	6 digits

Record in C/C++ is called *structure*. A structure can be defined as follows:

```
struct
        {
        int roll_no;
        char *name;
        int marks;
        }Student;
```

Here, "name" is a pointer variable that points a character type variable (data item). If the length of "name" field is 40, memory requirement for the above structure (record) = 2 + 40 + 2 = 44 bytes; as each integer occupies 2 bytes and each character is of 1 byte.

An array may be a member (field) of a structure (record). A structure (record) may be an element (data item) of an array.

Example:

1).

```
struct
    {
    int roll_no;
    char Name[40];
    int marks;
    }Stud;   // Here, Name[40] is an array which is a
                member of the structure, Stud.
```

2).

```
struct
    {
    int roll_no;
    char Name[40];
    int marks;
    }Student [30];
```

Here, *Student* is an array of structures where members are *roll_no*, *Name*[ ] and *marks*, and *Name*[ ] is an array.

We can assign value to the members of the structure (*Stud*) as follows:

```
Stud.roll_no = 5011;
Stud.name[ ] = "H.M. Mehedi Hasan";
Stud.marks  = 50;
```

If there is an array of structures, we can assign values to the members of the structure (*Student*) as follows:

```
Student [2].roll_no = 5011;
Student [2].name[ ] = "H.M. Mehedi Hasan";
Student [2].marks  = 50;
```


Memory requirement for the structure, *Student* (in example 2):

$(2 + 40 + 2) \times 30$ bytes

$= 44 \times 30$ bytes

$= 1320$ bytes

If we want to store data to a record (structure) for more than one person or student or customer, then we have to declare (define) *array of structures*. But disadvantage is that, for these types of array a huge amount of memory space may be misused or wasted.

Problem 3.1: Define a record (structure) with three fields for a student and store data for 30 students.

Solution:

```
struct
    {
    int roll_no;
    char name[40];
    int marks;
    } stud[30];
for (i = 0; i < 30; ++i)
    {
    store data in stud[i].roll_no;
    for (j = 0; j < 40; ++j)
        store data in stud[i].name[j];
    store data in stud[i].marks;
    } // For storing data we can use scanf ( ) function of C/C++
```

A data structure may be the member of another data structure also.

Code for problem 3.1 in C/C++ is as follows:

```
struct
{
int roll_no;
```

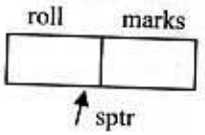
```
char name [40];
int marks;
 } stud[30];
for (i = 0; i < 30; ++i)
 {
scanf ("%d", &stud [i].roll_no;
fflash(stdin);
gets(stud[i].name);
scanf("%d", &stud[i].marks);
 }
```

To display data we can write code using printf();

Pointer to the structure:

We can store data using pointer to a structure (record). An example using C++ code is as follows:

```
struct stud
{
int roll;
int marks;
}
struct stud * sptr;
sptr = new(stud);
cin>> sptr →roll;
cout>>sptr → marks;
cout<<sptr → roll;
cout<< sptr → marks;
```



In the above code **sptr** is the pointer to the stricter named stud and **new** is a keyword for allocation memory (space) for the structure.

3.2 Difference between an array and a record

1. An array is a finite set of same type of data elements. In other words, it is a collection of homogeneous data items (elements). Whereas a record is a collection of non-homogenous (different types of) related data items.

2. The data items of an array are of same type. But in a record, data elements are usually of different types.

Summary:

Record is a collection of non-homogeneous related data items. It has different fields or attributes.

The basic difference between **array** and **record** is that, an array is a set of same types of data but a record is a set of different types of data items.

Questions:

1. Define record (structure) with example.
2. What is the difference between array and record?
3. Define array and record.
4. Assume an integer needs four byte, a real number needs eight byte and character needs one byte. Assume the following declaration:

```
struct Ku {
        char name[10];
        int roll;
        } ece[30];
```

If the starting address of ece is 100, what is the address of ece[15]?

Consider the following C program segment:

```
struct x{
        int sub [3];
        char name [10];
        long int roll;
}
struct y{
        struct x person;
        char addres[20];
        } m[20];
```

If the address of the m[0].person.sub[2] is 500, then what will be the address of m[10].person.sub[2]?

Problems for Practical (Lab) Class

Structure or record related problems

Problem 3-1: Given three attributes of student record such as roll, name and marks. Write a program using structure to store data for *five* students and display the data on the screen.

Problem 3-2: use the program for the **problem 3-1** and modify it to calculate grade for each student and display roll, name, marks and grade using one row for each student. Use grade calculation rules of your university.

Sample output:

Roll	Name	marks	Grade
1	Abdur Rahim	86	A-
2	Shameem Rahman	95	A+

Problem 3-3: Given three attributes of employee record such as ID, name and basic pay. Write a program using structure to store data for 5 persons (employees), calculate some benefits as follows:

i) House rent:
   a) House rent = 45 % of basic pay (for basic pay equals to 10000 or less)
   b) House rent = 40 % of basic pay (for basic pay more than 10000 or less than equals to 20000)
   c) House rent = 35 % of basic pay (for basic pay more than 20000)
ii) Transport allowance = 5 % of basic pay
iii) Medical allowance = 2000/- (fixed)
iv) PF deduction = 10 % of basic pay.
v) Gross pay = Basic pay + house rent + Transport allowance + Medical allowance
vi) Net pay = Gross pay – basic pay

Display the data as follows:

ID	Name	Basic pay	Gross pay	Deduction	Net pay
0908	M Karim	12000	190400	1200	18200
0910	A Mazid	23000	35350	2300	33050

Problem 3-4: Declare a structure with two attributes (fields) such as roll, name and marks. Declare a pointer variable of structure type (pointer to the structure) and store and display the data using the pointer and attributes. Hints: see pointer to the structure (page 28).

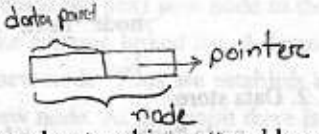
CHAPTER FOUR
LINKED LIST

OBJECTIVES:

- Identify Linked list and Doubly Linked List
- Describe the creation process of a linear linked list and Doubly linked list
- Write an algorithm to create a linked list and Doubly linked list
- Write an algorithm to locate a node of linked list and Doubly linked list
- Describe the insertion process of a node into a linked list and Doubly linked list
- Write an algorithm to insert a node into a linked list and Doubly linked list
- Describe the deletion process of node from a linked list and Doubly linked list
- Write an algorithm to delete a node from a linked list and Doubly linked list
- Write an algorithm to arrange data of linked list and Doubly linked list
- Differentiate the array and the linked list

LINKED LIST

4.1 Definition



It is a list or collection of data items that can be stored in scattered locations (positions) in memory. To store data in scattered locations in memory we have to make link between one data item to another. So, each data item or element must have two parts: one is data part and another is link (pointer) part. Each data item of a linked list is called a node. Data part contains (holds) actual data (information) and the link part points to the next node of the list. To locate the list an external pointer is used that points the first node of the list. The link part of the last node will not point any node. So, it will be *null*. This type of list is called linear (one way) linked list or simply linked list.

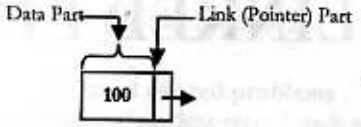


Figure-4.1 (a): A single node

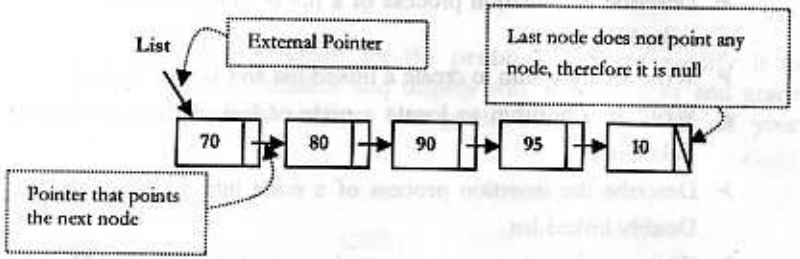


Figure- 4.1(b): Graphical representation of a linear linked list

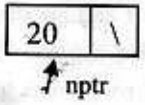
4.1.1 Node declaration and store data in a node (in C/C++)

1. Node Declaration:

```
struct node
    {
    int data;
    node *next;
    };
```

2. Data store:

```
node *nptr;          //declare nptr as node type
nptr→data = 20;      //store 20 in data part
nptr→next = NULL; //void the pointer part
```



4.1.2 Create a new node

1. Node declaration:

```
struct node
    {
    int data;
    node *next;
    };
```

2. Declare variables (pointer type) that point to the node:

```
node *nptr;
```

3. Allocate memory for new node:

```
nptr = new (node);
```

4. Insert node value:

```
nptr→data = 55;
nptr→next = NULL;
```

4.1.3 Create a linked list

To create a linked list at first we have to create an empty list. That means, the external pointer to the list will point no node or the external pointer will be *null*. After that, we have to create a new node. The data part of the new node will contain data (information) and the pointer part will contain *null*. This temporary pointer will help us to include the next new node to the list. Now we shall include this new node to the existing linked list. A temporary pointer will be used to point (trace) the new node. Thus we establish a link between the existing linked list and the new node. At this point there is only one node in the linked list. We shall create another new node and include the node to the linked list. We shall repeat the process to include any other node. Thus we can create a linked list.

We can create a linked list according to the following process given in points.

1. Create an empty linked list.
   (The external pointer will be *null*).
2. Create a new node with necessary data.

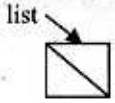
(The data part of the new node will contain data (information and the pointer part will contain *null*).

3. The external pointer will point the new node.
   (At this moment there is only one node in the list)
4. Create another new node and include the node to the linked list.
5. **Repeat and process** to include any other node.

Linked list creation process using pseudo code:

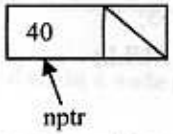
1. Create an empty list, the pointer list will point to *null:*

list=Null;

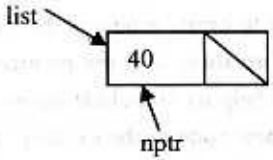


2) Create a new node with data:

npt = new (node);
nptr-> data = item;
nptr ->next = NULL;
(Here the value of item is 40).



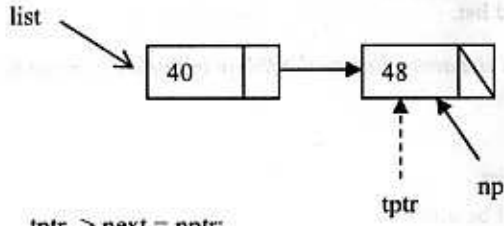
3) Include the first node to the list



list = nptr;
tprt = nptr;

Here **tptr** is temporary pointer used to make link between existing list and new node.

4) Create another node with data and include the node to the list:



tptr -> next = nptr;
tptr=nptr;

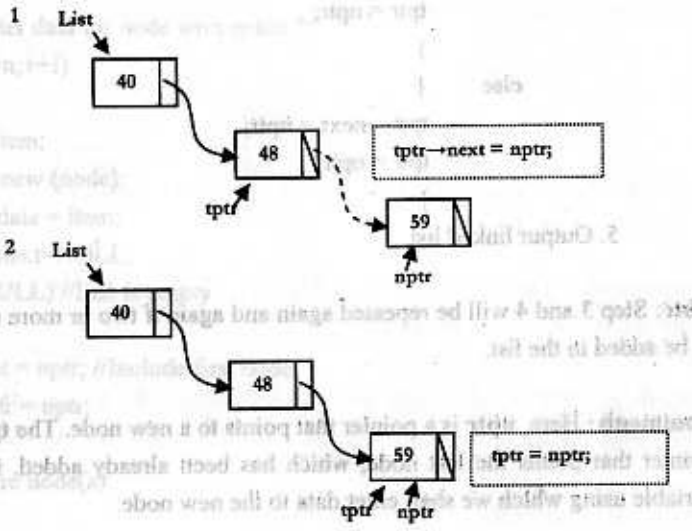


Figure-4.2: Addition of nodes to a linked list (pictorial view)

Algorithm 4.1: Algorithm (pseudo code) to create a linked list

1. Declare node and pointers (list, tptr, nptr):
   i. struct node
   {
   int data;
   node *next;
   };
   ii. node *list, *tptr, *nptr;
2. Create an empty list:
   list = NULL;
3. Create a new node:
   nptr = new (node);
   nptr→data = item;
   nptr→next = NULL;
4. Make link between the linked list and the new node:
   if (list = = NULL)
   {

```
        list = nptr;
                tptr = nptr;
                }
        else    {
                tptr→next = nptr;
                tptr = nptr;
                }
```

5. Output linked list

Note: Step 3 and 4 will be repeated again and again if two or more nodes are to be added in the list.

Comments: Here, **nptr** is a pointer that points to a new node. The **tptr** is the pointer that points the last node, which has been already added. **item** is a variable using which we shall enter data to the new node.

Program to create a linked list:

Although we have written linked list creation process in detail, however, there are some students they face problem to write a program to create a linked list. So, here we write a program to create a linked list (header files are not included). After creating a lined list we have to read the data of the linked list and display them on the monitor. Otherwise we cannot understand whether the list is created properly. At the end of the program we include code to display data of the list.

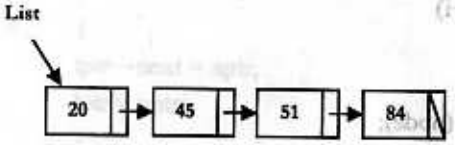
```
void main()
struct node
  {
int data;
node *next;
  };
int i,n,item;
node *nptr, *tptr, *list; // Necessary pointers
list = NULL;      // Create an empty list
cout<<"Enter number of nodes:";
```



```
cin>>n;
cout<<"Enter data for node with space:";
for (i=l;i<=n;++i)
 {
    cin>>item;
    nptr = new (node);
    nptr->data = item;
    nptr->next=NULL;
if (list==NULL) //List is empty
        {
        list = nptr; //Include first node
        tptr = nptr;
        }
//There is/are node(s)
else
{
    tptr->next = nptr;
    tptr = nptr;
}
}//end of for loop
//Display data
tptr = list;
for (i = 1; i <= n; ++i)
{
cout<<endl;
cout<<tptr->data;
tptr = tptr->next;
cout<< "";
}
cout<<endl;
cout<<endl;
}
```

4.1.4 Locate or search a node of a linked list

Here, we have to find out a node whose value is known.



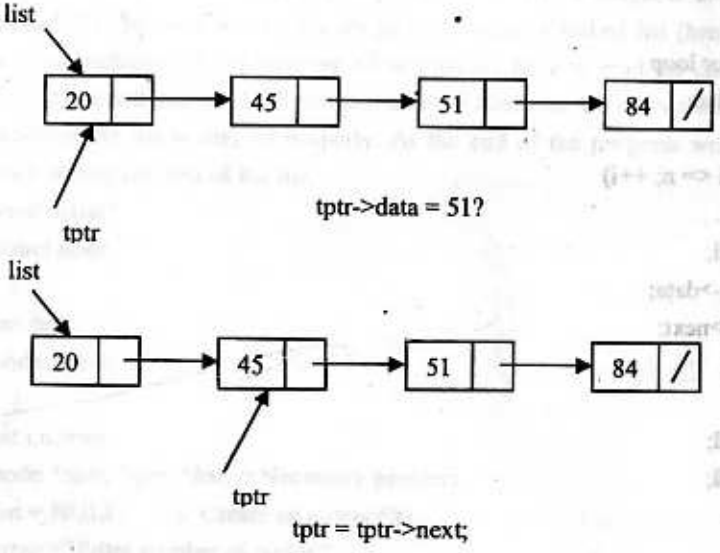
Problem 4.1: Find out the item 51 from above linked list. All items in the list were stored in ascending order.

Solution: To locate the node we have to traverse the list using a pointer. Here, we shall use a temporary pointer using which we shall find the node.

We can write in points:

a) We have to use a temporary pointer to traverse the list

b) At each node we compare and check whether we have found the node or not.



Algorithm 4.2: Algorithm (pseudo code) to search a node from a linked list

1. Declare **node** and **tptr**
2. Input the value to be located:
   item = 51;
3. Search the item:

```
tptr = list;
while (tptr→data != item or tptr! = NULL)
        {
        tptr = tptr→next;
        }
```

4. Output:

```
if (tptr→data = item)
        print "FOUND"
else print "NOT FOUND"
```

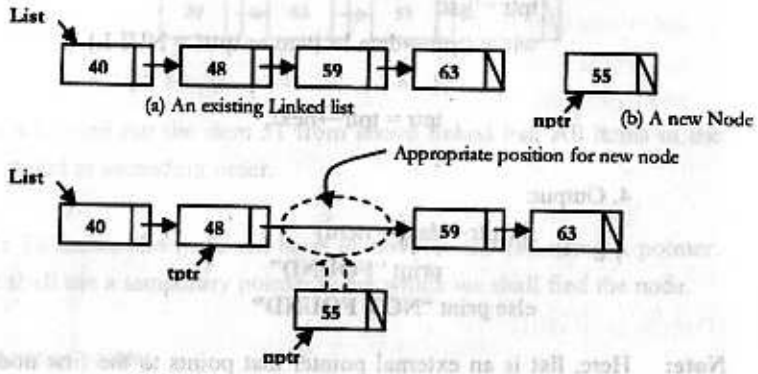
Note: Here, **list** is an external pointer that points to the first node of the list and **tptr** is a temporary pointer and **item** is a variable that contains the node value to be located.

4.1.5 Insert a node into a list

Here, we shall consider the insertion of a node after the first node or before the last node and the data of the list are arranged in ascending order.

To insert a (new) node into a linked list, we have to find out (locate) the position of the node after which the new node will be inserted. To find out the position we shall use a temporary pointer. As we know there is an external pointer to the linked list, which points the first node of the list. So, we shall assign the value of the external pointer to the temporary pointer. At the first stage temporary pointer will point the first node of the linked list. Now we shall compare the data of the second node of the list with the data of the new node. If the data of the second node of the list is greater than the data of the new node, then we have found the position for insertion. So, the new node will be inserted after the first node (before the second node). If the data of the second node is smaller than the data of the new node, then we

advance the temporary pointer. That is, at this stage the temporary pointer will point the second node of the list. Then we shall compare the data of the new node with the data of third node. If the data of the new node is smaller than the data of the third node, the new node will be inserted after the second node or before the third node. If not, we have to advance further.



To *insert a node in between two nodes we have to perform two major tasks:*

i) To locate (find out) the node after which the new node will be inserted.

ii) To perform insertion of making necessary link.

To locale the position for insertion we have to perform the following operations:

i) Use a temporary pointer (tptr) to the first node of the list (**tptr = list**).

ii) Compare the value of the next node with value of the new node

iii) Traverse the temporary pointer until we find a greater node value than the value of the new node. (**tptr = tptr -> next**).

To insert the node by making link the steps are:

i) Point the next node by pointer of the new mode (nptr -> next = tptr->next)

ii) Point the new mode by previous node of the new node.

tptr->next= nptr

Figure-4.3 gives a clear pictorial view of insertion of a new node in an existing linked list.

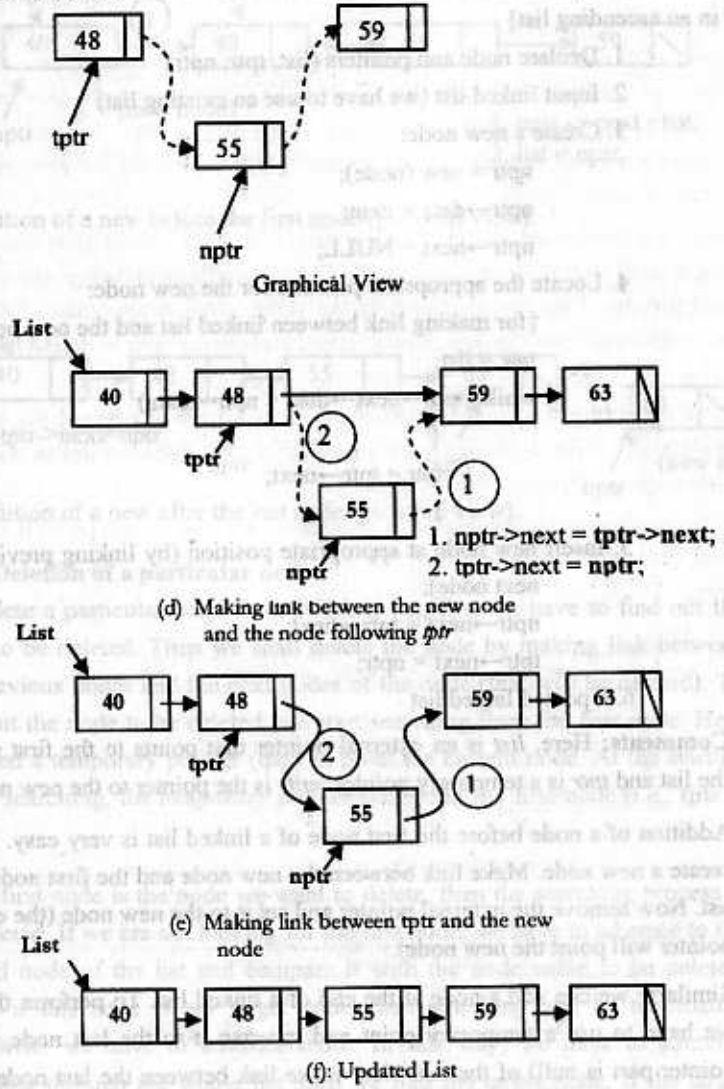


Figure-4.3: New node insertion in an existing list (pictorial view)

Algorithm 4.3: Algorithm (pseudo code) to insert a node into a linked list [Here, we shall consider insertion after the first node or before the last node in an ascending list]

1. Declare node and pointers (list, tptr, nptr)
2. Input linked list (we have to use an existing list)
3. Create a new node:

```
nptr = new (node);
nptr→data = item;
nptr→next = NULL;
```

4. Locate the appropriate position for the new node:

[for making link between linked list and the new node]

```
tptr = list;
while (tptr→next→data < nptr→data)
        {
        tptr = tptr→next;
        }
```

5. Insert new node at appropriate position (by linking previous and next node):

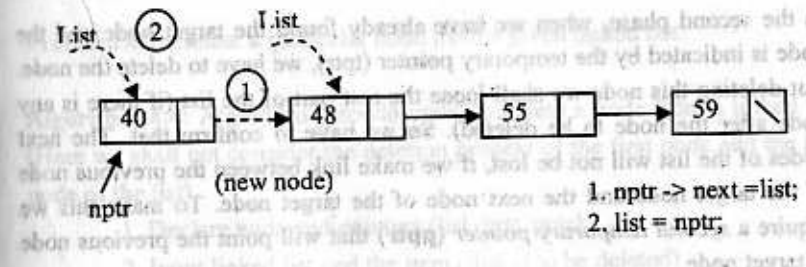
```
nptr→next = tptr→next;
tptr→next = nptr;
```

6. Updated linked list

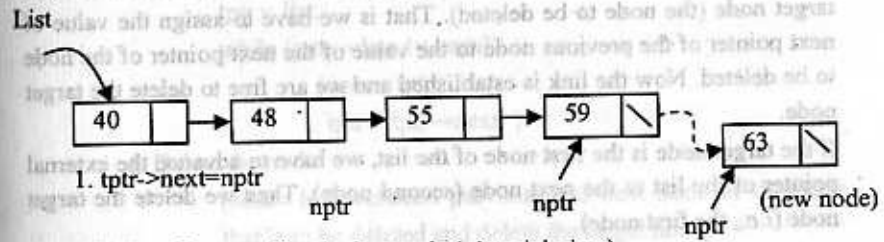
Comments: Here, *list* is an external pointer that points to the first node of the list and *tptr* is a temporary pointer. *nptr* is the pointer to the new node.

Addition of a node before the first node of a linked list is very easy. At first create a new node. Make link between the new node and the first node of the list. Now remove the external pointer and set it to the new node (the external pointer will point the new node).

Similarly we can add a node at the end of a linked list. To perform this task we have to use a temporary point and traverse it to the last node (whose pointer part is null) of the list. Now make link between the last node of the list and the new node.



i) Addition of a new before the first node (pictorial view).



ii) Addition of a new after the last node (pictorial view).

4.1.6 Deletion of a particular node

To delete a particular node from a linked list, first we have to find out the node to be deleted. Then we shall delete the node by making link between the previous nodes and the next nodes of the node (that will be deleted). To find out the node to be deleted, we start searching from the first node. Here we need a temporary pointer (tptr) to point the current node. At the starting of the searching, the *temporary pointer* will point the first node (*i.e.*, **tptr = list**).

If the first node is the node we want to delete, then the searching process is completed. If we are not looking for the first node, we have to advance to the second node of the list and compare it with the node value to be deleted. Again if this node is our target node, then searching will be terminated. Otherwise, we have to advance more. In this way, we have to continue searching through the whole list until we find the target node. If the target node is not present in the list, we shall simply close our mission.

In the second phase, when we have already found the target node and the node is indicated by the temporary pointer (tptr), we have to delete the node. But deleting this node we shall loose the rest part of the list (if there is any node after the node to be deleted). So we have to confirm that. The next nodes of the list will not be lost, if we make link between the previous node of the target node and the next node of the target node. To make this we require a *second temporary pointer* (**pptr**) that will point the previous node of target node.

Now, we shall make link between the previous node and the next node of the target node (the node to be deleted). That is we have to assign the value of next pointer of the previous node to the value of the next pointer of the node to be deleted. Now the link is established and we are free to delete the target node.

If the target node is the first node of the list, we have to advance the external pointer of the list to the next node (second node). Then we delete the target node (*i.e.*, the first node).

If the target node is the last node of the list, we have to assign NULL to the next pointer of the previous node of the target node. Then we delete the node.

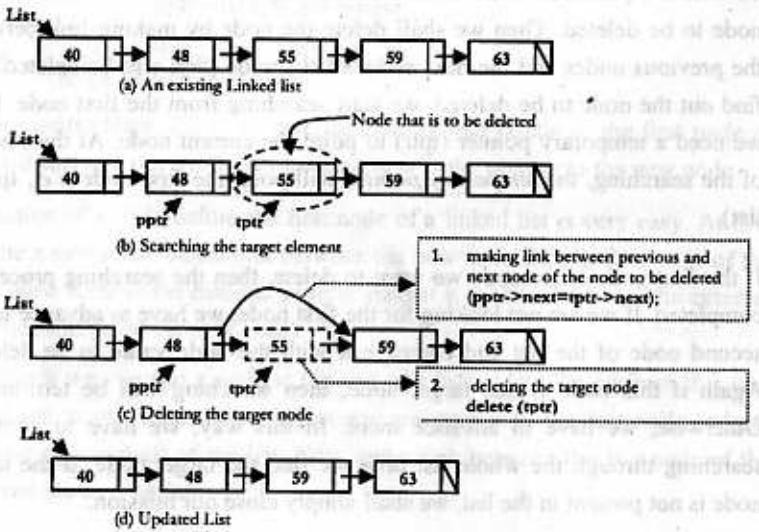


Figure-4.4: Deletion of an item from a linked list (pictorial view)

Problem 4.2: Delete a particular node from a given linked list.

Algorithm 4.4: Algorithm (pseudo code) to delete a node from a linked list (Here we shall not consider the deletion process of the first node and the last node of the list).

1. Declare node and pointers (list, tptr, pptr)
2. Input linked list and the item (that is to be deleted)
3. Search the item to be deleted in the list:

```
tptr = list;
while (tptr→data != item)
        {pptr = tptr;
         tptr = tptr→next; }
```

4. Delete the node:

[Make link between previous and next node of the node that is to be deleted and delete the target node]

```
pptr→next = tptr→next;
delete (tptr);
```

5. Output: updated linked lists

Comments: Here, **list** is a pointer to the linked list and **tptr** is a pointer that will point the node to be deleted. And **pptr** is a pointer that will point the previous node of the target node (the node is to be deleted) and **item** is the node value to be deleted.

Problem 4.3: Given a linked list, where the data of the nodes are not arranged in an order. Arrange the data of the linked list in ascending order.

Algorithm 4.5: Algorithm (pseudo code) to arrange data of linked list

1. Input linked list.
2. Take extra pointers pptr, fptr.
3. pptr = list;
4. while (pptr != NULL)

```
{
fptr = pptr→next;
```

```
5. while (fptr != NULL)
                {
                if (pptr→data > fptr→data)
                {
                  interchange (pptr→data, fptr→data);
                }
                fptr = fptr→next;
                } //end of while of step-5
        pptr = pptr→next;
        } //end of while of step-4
```

6. **Output:** The arranged (sorted) list.

Comments: *list* is the pointer to the first of the linked list. *pptr* and *fptr* are the pointers to first and second nodes, and three pointers will be used to arrange (sort) data.

4.2 Doubly linked list

4.2.1 Definition

A doubly or two way linked list is a list where each node has three parts. One is link or pointer to the previous (backward) node and one is data part to hold the data and another is link or pointer to the following (forward) node. There is an external pointer to the first node of the list. Doubly linked list is also called two-way linked list.

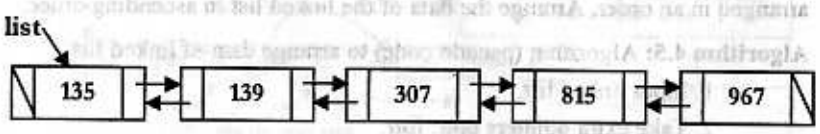


Figure-4.5: Graphical representation of a doubly linked list

4.2.2 Declare a node of a doubly linked list

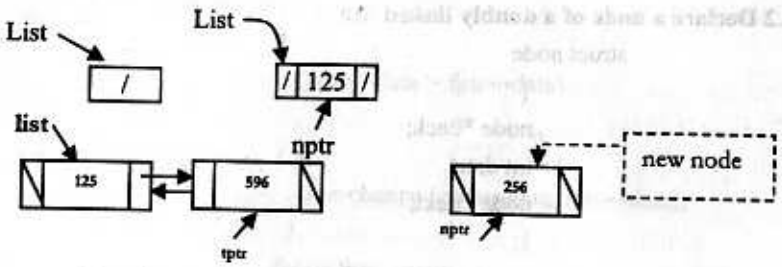
```
struct node
        {
        node *back;
        int data;
        node *next;
        };
```

4.2.3 Create a node

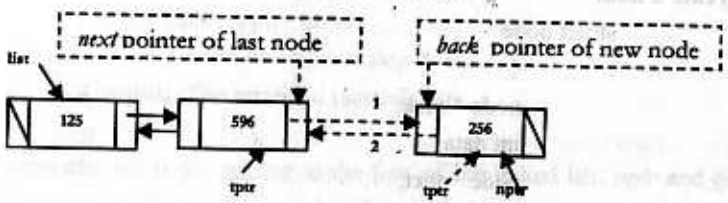
```
struct node
        {
        node *back;
        int data;
        node *next;
        };
node *nptr;
nptr = new (node);
nptr→back = NULL;
nptr→data = item;
nptr→next = NULL;
```

4.2.4 Create a doubly linked list

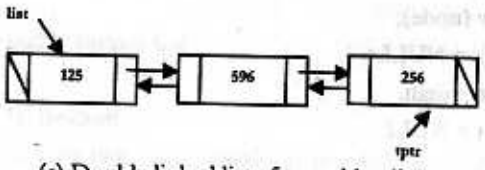
To create a doubly linked list, we have to create an empty linked list first. Then we shall create a new node with data and include the node to the list. After that, we shall create another new node with data and include this node to the list. To include a new node we have to make link between the last node of the list and the new node. To make link, the ***next*** pointer of the last node will point the new node and the ***back*** pointer of the new node will point the last node of the list. Thus we can create a doubly linked list. The pictorial view of this creation process is shown in Figure 4.6.



(a) A doubly linked list and new node



(b) Linking the last node of the list and the new node



(c) Doubly linked list after making link

Figure-4.6: Creation of doubly linked list (pictorial view)

Algorithm 4.6: Algorithm (pseudo code) to create a doubly linked list

1. Declare node and pointers:

```
a. struct node
        {
        node *back;
        int data;
        node *next;
        }
b. node *list, *tptr;
```


2. Create an empty list:

```
list = NULL;
```

3. Create a new node:

```
node *nptr;
nptr = new (node);
nptr→back = NULL;
nptr→data = item;
nptr→next = NULL;
```

4. Make link between the last node of the list and the new node:

```
if (list = NULL)
        {
        list = nptr;
        tptr = nptr;
        }
else    {
        tptr→next = nptr;
        nptr→back = tptr;
        tptr = nptr;
        }
```

5. Output a doubly linked list.

Note: To create several nodes we have to repeat the step 3 and the step 4.

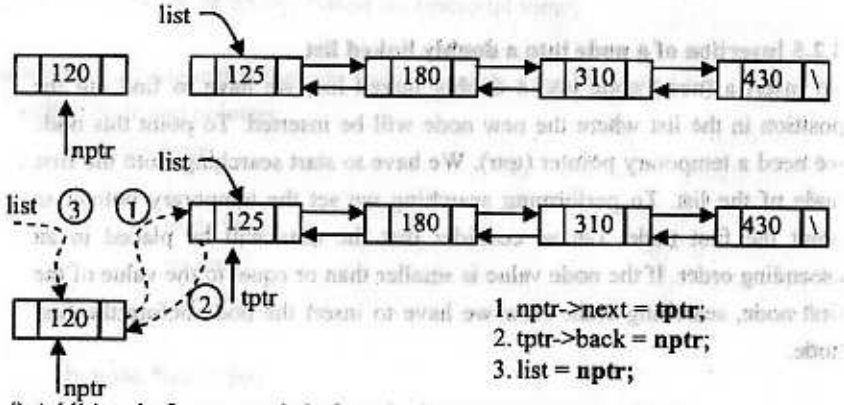
4.2.5 Insertion of a node into a doubly linked list

To insert a (new) node into a doubly linked list, we have to find out the position in the list where the new node will be inserted. To point this node we need a temporary pointer (tptr). We have to start searching from the first node of the list. To performing searching we set the temporary pointer to point the first node. Let us consider that the data will be placed in an ascending order. If the node value is smaller than or equal to the value of the first node, searching ends. Now we have to insert the node before the first node.

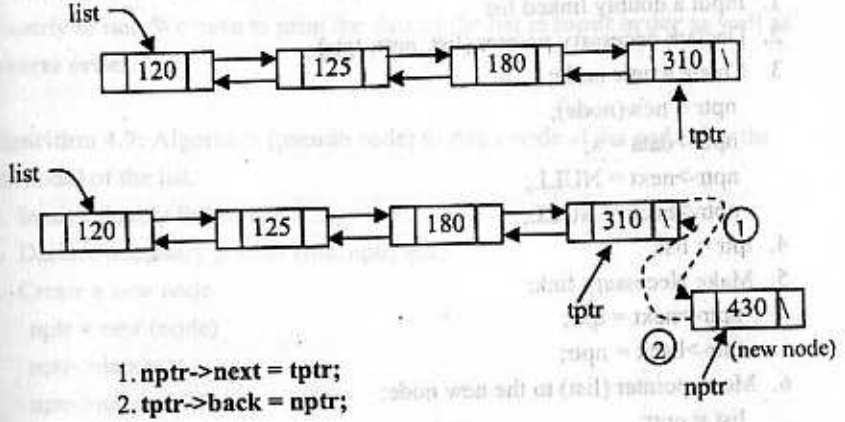
If the new node value is greater than the first node, we have to advance the temporary pointer (tptr) to the next node. Now, if the new node value is smaller than the current (second) node, the new node will be inserted before the current node. If new node value is still greater, advance the temporary pointer to the next node. This process will continue until either we find the node after which we shall insert the new node or we reach the last node of the list.

Now, when we have completed searching the position, we shall make link among the node after which the new node will be positioned (inserted), the node before which the node will be inserted and the new node itself. That means, as the list is a two way linked list, generally insertion takes place between two nodes; previous node and the next node (of the new node to be added). First, we have to make link between the node after which the new node will be inserted and the new node. Then we shall make link between the new node and the node before which the node will be inserted. After making links, insertion process terminates.

If the node is to be inserted before the first node, we have to make link between the new node and the first node. New node will be placed at the very first of the list and then link has to be made between the new node and the first node of the list. After that, we have to set list pointer (list) to point the new node. If the node is to be added at the end of the list, we have to make link between the last node and new node in such a manner that the new node will be added at the end of the last node (after the last node).



i) Additional of a new node before the first node (pictorial view)



ii) Addition of a new node after the last node (pictorial view).

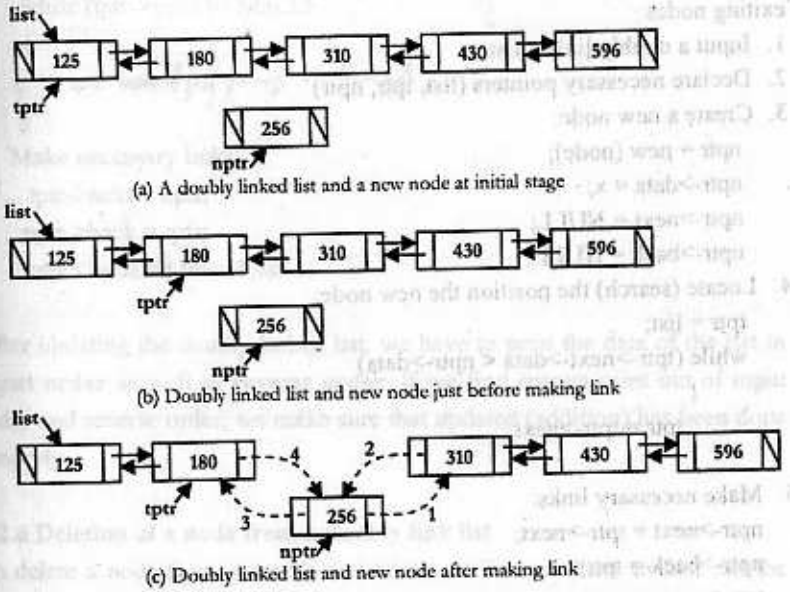


Figure-4.7: Insertion of a node into a doubly linked list (pictorial view)

Algorithm 4.7: Algorithm (pseudo code) to add a new node before the first node.

1. Input a doubly linked list
2. Declare necessary pointer (list, nptr, tptr)
3. Create a new node:
   nptr = new(node);
   nptr->data = x;
   nptr->next = NULL;
   nptr->back = NULL;
4. tptr = list;
5. Make Necessary link:
   nptr->next = tptr;
   tptr->back = nptr;
6. Move pointer (list) to the new node:
   list = nptr;
7. output updated linked list

Algorithm 4.8: Algorithm (pseudo code) to insert a node in between two exiting nodes

1. Input a doubly linked list.
2. Declare necessary pointers (list, tptr, nptr)
3. Create a new node:
   nptr = new (node);
   nptr->data = x;
   nptr->next = NULL;
   nptr->back = NULL;
4. Locate (search) the position the new node:
   tptr = list;
   while (tptr->next->data < nptr->data)
   {
   tptr = tptr->next
   }
5. Make necessary links:
   nptr->next = tptr->next;
   nptr->back = tptr;
   tptr->next = nptr;
7. Output updated linked list.

Comments: to check whether the doubly linked list has been updated properly or not. We have to print the data of the list in **input order** as well as **reverse order**.

Algorithm 4.9: Algorithm (pseudo code) to add a node at the end (after the last node) of the list.

1. Input a doubly linked list
2. Declare necessary pointer (list, nptr, tptr)
3. Create a new node:
   nptr = new (node)
   nptr->data = x;
   nptr->next = NULL;
   nptr->back = NULL;
4. Locate the last node of the list:
   tptr = list;
   while (tptr->next != NULL)
   {
   tptr=~~list;~~ tptr->next;
   }
5. Make necessary links:
   tptr->next = nptr;
   nptr->back = tptr;
6. Output updated linked list.

After updating the doubly linked list, we have to print the data of the list in **input order** as well as **reverse order**. If we find correct print out of input order and reverse order, we make sure that updated (addition) has been done properly.

4.2.6 Deletion of a node from a doubly link list

To delete a node from a doubly linked list, at first we have to find out the node to be deleted. For this, we shall use a temporary pointer to point the node. When the temporary pointer points the node to be deleted, before deleting the node we have to make link between the previous node and the

next node of the node (to be deleted). To do this, the ***next*** pointer of the previous node will point the next node and the ***back*** pointer of the next node will point the previous node of the node. After establishing the link we shall delete the target node. The pictorial view of the deletion process is shown in Figure 4.8.

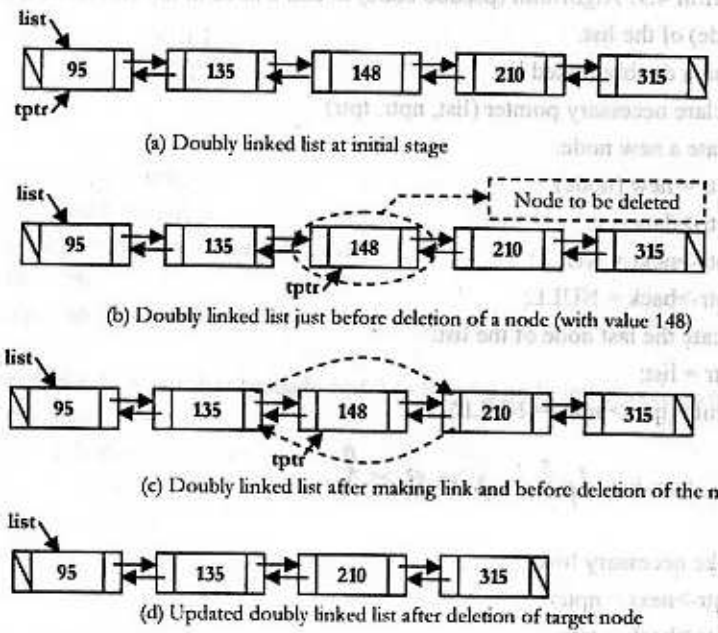


Figure-4.8: Deletion of a node from a doubly linked list (pictorial view)

Algorithm 4.10: Algorithm (pseudo code) to delete a node from a doubly linked list

1. Declare node and pointers (list, tptr);
2. Input a doubly link list and item (value of the node to be deleted);
3. Locate the node to be deleted:

```
tptr = list;

while (tptr→next != NULL)
{
if (tptr→data = = item) break;
tptr = tptr→next;
}
```


4. Make link among rest of the nodes (excluding the node to be deleted):
   a. If the node to be deleted is the first node of the list:

```
list = tptr→next;
list→back = NULL;
```

   [if there is only one node in the list then list = NULL]
   b. If the node to be deleted is not the last node:

```
tptr→back→next = tptr→next;
tptr→next→back = tptr→back;
```

   c. If the node to be deleted is the last node:

```
tptr→back→next = NULL;
```

5. Delete the target node:

```
delete (tptr);
```

6. Output: Updated doubly linked list.

4.3 XOR linked list

An XOR linked list is a linked list that uses the bitwise exclusive disjunction to decrease memory requirements. XOR linked lists were quite common in the early days of computers, when the available memory was quite sparse.

Principle

The principle used by XOR linked list is quite interesting. XOR linked lists are doubly-linked lists, but unlike traditional linked list, their nodes use only one pointer to obtain the predecessor and the successor. In order to do this, each node contains a pointer that stores the bitwise XOR of the previous and next field. Traversing the list from the beginning to the end is easy. For a given item A (n), XORing the address of A (n-1) and the address of A (n) will give you the address of A (n+1). The same procedure can be applied for the other direction. Starting the process of traversing the list is done using the address of two consecutive nodes, by XORing the addresses they store in order to obtain the one of the starting point.

An ordinary doubly-linked list stores addresses of the previous and next list items in each list node, requiring two address fields:

... A B C D E...
-> next -> next -> next ->
<- prev <- prev <- prev <-

An XOR linked list compresses the same information into one address field by storing the bitwise XOR of the address for previous and the address for next in one field:

... A B C D E...
<-> A⊕C <->B⊕D<->C⊕E <->

When you traverse the list from left to right: supposing you are at C, you can take the address of the previous item, B, and XOR it with the value in the link field (B⊕D). You will then have the address for D and you can continue traversing the list. The same pattern applies in the other direction.

To start traversing the list in either direction from some point, you need the address of two consecutive items, not just one. If the addresses of the two consecutive items are reversed, you will end up traversing the list in the opposite direction.

Features

* Given only one list item, one cannot immediately obtain the addresses of the other elements of the list.
* Two XOR operations suffice to do the traversal from one item to the next, the same instructions sufficing in both cases. Consider a list with items {...B C D...} and with R1 and R2 being registers containing, respectively, the address of the current (say C) list item and a work register containing the XOR of the current address with the previous address (say C⊕D). Cast as System/360 instructions:

```
X  R2,Link   R2 <- C⊕D ⊕ B⊕D (i.e. B⊕C, "Link" being the link field
                 in the current record, containing B⊕D)
XR R1,R2     R1 <- C ⊕ B⊕C   (i.e. B,  the next record)
```


* End of list is signified by imagining a list item at address zero placed adjacent to an end point, as in {0 A B C...}. The link field at A would be 0⊕B. An additional instruction is needed in the above sequence after the two XOR operations to detect a zero result in developing the address of the current item,
* A list end point can be made reflective by making the link pointer be zero. A zero pointer is a mirror. (The XOR of the left and right neighbor addresses, being the same, is zero.)

Why does it work?

The key is the first operation, and the properties of XOR:

* X⊕X=0
* X⊕0=X
* X⊕Y=Y⊕X
* (X⊕Y)⊕Z=X⊕(Y⊕Z)

The R2 register always contains the XOR of the address of current item C with the address of the predecessor item P: C⊕P. The Link fields in the records contain the XOR of the left and right successor addresses, say L⊕R. XOR of R2 (C⊕P) with the current link field (L⊕R) yields C⊕P⊕L⊕R.

* If the predecessor was L, the P(=L) and L cancel out leaving C⊕R.
* If the predecessor had been R, the P(=R) and R cancel, leaving C⊕L.

In each case, the result is the XOR of the current address with the next address. XOR of this with the current address in R1 leaves the next address. R2 is left with the requisite XOR pair of the (now) current address and the predecessor.

Use

Although XOR linked lists were heavily used a few decades ago, their usage is now discouraged unless it is absolutely necessary. It is generally used only for embedded devices and microcontrollers, because they do have a number of disadvantages:

- Most debuggers cannot follow the structure of such a list, making programs very hard to debug. The code required to use XOR lists is quite complex, too.
- Many high level languages do not support the XORing of pointers directly or at all.
- The pointers are not available if the list is not actually traversed
- Conservative garbage collection schemes cannot be used, since they need literal pointers to work. Implementing a special garbage collector is not practical on low-memory devices.
- Modern computer architectures have no use for such lists, since they do have enough memory. Unrolling is generally a better choice for programmers looking to decrease the overhead.

4.4 Circular linked list

A circular linked list is a list where each node has two parts; one is data part to hold the data and another is link or pointer part that points the next node and the last node's pointer points the first node of the list. Like other linked list there is an external pointer to the list to point the first node.

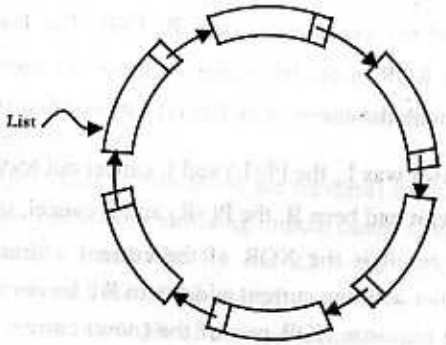


Figure-4.9: Pictorial view of a circular linked list (a circular diagram)

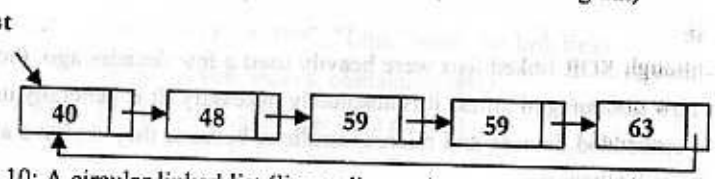


Figure-4.10: A circular linked list (linear diagram)

4.4.1 Create a circular linked list

Creation process of a circular linked list is similar to the creation process of a linear link list, which had been discussed in section 4.1.4. Here we have to do addition thing, that is make link between the last node and the first node which will create a circle (linked list as a circle).

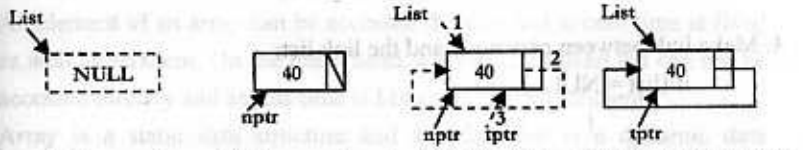
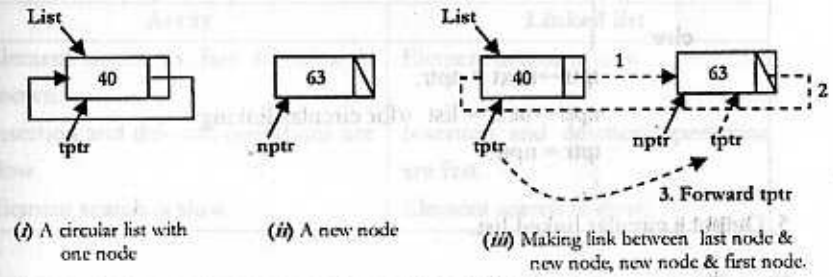


Figure-4.11: Linking process of the first node of a circular linked list



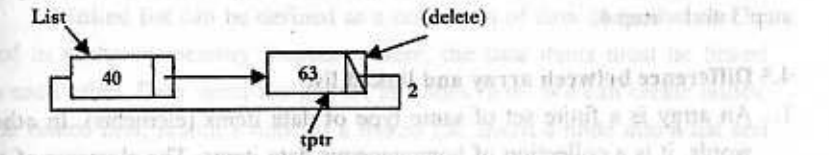


Figure-4.12: Creation of a circular linked list with more than one node

Algorithm 4.11: Algorithm to create a circular linked list

1. Declare node and pointers (list, tptr, nptr)
2. Create an empty inked list

   list = NULL;

3. Create a new node with data:

```
nptr = new (node);
nptr→data = item;
nptr→next = NULL;
```

4. Make link between new node and the link list:

```
if (list = NULL)
        {
        list = nptr;
        nptr→next = list; //for circular linking
        tptr = nptr;
        }
else    {
        tptr→next = nptr;
        nptr→next = list  //for circular linking
        tptr = nptr;
        }
```

5. Output a circular linked list.

Note: To create a circular linked list with several nodes we have to repeat step 3 and step 4.

4.5 Difference between array and linked list

1. An array is a finite set of same type of data items (elements). In other words, it is a collection of homogeneous data items. The elements of an array are stored in successive memory locations. Any element of an array is referred by array name and index number (subscript).

Whereas, linked list is a list or collection of data items stored in scattered memory locations. Each data item has two parts. One is data part and another is link (pointer) part. Each data item of a linked list is called node. Data part holds actual data (information) and the link part points to the next node of the list. To locate the list or the 1st node of the list, an external pointer is used. The link part of the last node will not point any node. That means it will be null.

2. Array implementation depends on size and it results in wastage of memory space. On the other hand, linked list does not depend on size.
3. Types of array are one dimensional, two dimensional etc. and types of linked list are linear, doubly, circular etc.
4. An element of an array can be accessed directly and access time is fixed as well as efficient. On the other hand, a node of a linked list can not be accessed directly and access time is linear and not so efficient.
5. Array is a static data structure and a linked list is a dynamic data structure.

4.6 Comparison of operations using array and linked list.

Array	Linked list
Element access is fast if index is known.	Element access is slow.
Insertion and deletion operations are slow.	Insertion and deletion operations are fast.
Element search is slow.	Element search is slow.

Summary:

Linked list can be defined as a collection of data items that can be sorted in scattered memory locations. Here, the data items must be linked with each other. Data items are known as nodes here. We can create nodes, create linked lists, search a node of a linked list, insert a node into a list and delete a node from the list.

A **linear linked list** is a list where there is only one way link between nodes. A doubly linked list is a list where there are links in both directions between the nodes. A circular linkd list is a list where the last node has a pointer which points to the first node. Each type of linked list requires an external pointer to point the first node of the list.

There are some differences between an **array** and a **linked list** such as array implementation depends on size but linked list implementation is independent of size. Array is a static data structure whereas linked list is a dynamic data structure etc.

Questions:

1. What is linear linked list ? Explain with example(s).
2. Describe the creation process of a linked list.
3. How a node can be inserted into a linked list? Explain with example.
4. Write an algorithm to create a linear lined list.
5. Write an algorithm to insert a node at the beginning of a linked list.
6. Write an algorithm to insert a node into a linked list. The node will not be first not or the last node.
7. Write an algorithm to delete a node from a linear linked list.
8. Write the algorithm to implement a sorted linked list into which elements can only be inserted into their proper positions.
9. Write a function, which will destroy a linked list.
10. "Insertion and deletion in linked list is easier than array", Explain.
11. What are the relative advantages and disadvantages of fixed length storage structure and linked-list storage structure ?
12. When is linked-list more convenient than array ?
13. Define doubly linked list with example.
14. Write an algorithm to delete an item from a two-way list.
15. Write down an algorithm to insert and delete an element into a doubly sorted linked list.
16. Write a function to swap two nodes for a doubly linked list.
17. Write an algorithm to find (locate) a node of a doubly linked list.
18. When will you use a doubly linked list ? Why ?
19. What is circular linked list? Give example.
20. Write an algorithm to delete a node from a circular linked list.
21. Write an algorithm to insert a node into a circular linked list.

Problems for Practical (Lab) Class

Linked list related problems

Problem 4-1: Write a program to create a linked list of five nodes where each node of the list will contain an integer. Display the data of the list on the screen.

Problem 4-2: Convert your program of the **problem 4-1** as follows:
Write a function for node creation, another for linkage and another function for display the data. Call them to show the result of execution.

Problem 4-3: Write a program to insert a node in between two existing nodes of a linked list. Display the data of the existing (old) linked list and the updated linked list.

Problem 4-4: Write a program for the followings:

i) Add a node before the first node of the list.
ii) Inset a node in between two existing nodes of the list.
iii) Add a node at the end of the list.

Display the data before addition and after addition or insertion of a node.
Conditions: Write a function for nodecreation, a function for addition and another function to display the data.

Problem 4-5: Write a program to delete a node from in-between two existing nodes of a linked list. Display the data before deletion and after deletion operation.

Problem 4-6: Write a program for the followings:

i) Delete the first node of the linked list.
ii) Delete the node from in-between two existing nodes.
iii) Delete the last node of the list.

Display the data before and after the deletion.

Conditions: write a function for creation, a function for deletion and another function to display the data.

Problem 4-7: Given a linked list with data arranged in random order. Write a program to modify the linked list so that it contains data in ascending order. Display the data before and after the modification.

Problem 4-8: Write a program to create a doubly linked list of five nodes and display the data in **input order as well as reverse order.**

Problem 4-9: Write a program using doubly linked list for the followings:

i) Add a node before the first node.
ii) Insert a node in-between two existing nodes.
iii) Add a node at the end of the list.

Display the data in **reverse order** before and after the addition of a node.

Condition: write a function for creation, a function for node addition and another function to display the data.

Problem 4-10: write a program to delete a node from in-between two existing nodes of a doubly linked list. Display the data in **input and reverse orders** before as well as after the deletion operation.

Problem 4-11: write a program in case of doubly linked list for the followings:

i) Delete the first node of the list.
ii) Delete a node from in-between two existing nodes.
iii) Delete the last node of the list.

Condition: write a function to create a list, a function for deletion operation and a function to display the data in **reverse order.**

Problem 4-12: Given a doubly linked list with integers arranged in ascending order. Write a program to display the data in descending order without using any sorting algorithm and/or stack.

CHAPTER FIVE
STACK

OBJECTIVES:

- Identify stack
- Describe push operation on array based stack
- Write an algorithm for push operation on array based stack
- Describe pop operation on array based stack
- Write an algorithm for pop operation on array based stack
- Describe the creation process of a linked stack
- Write algorithm to create a linked stack
- Describe push operation on linked stack
- Write an algorithm for push operation on linked stack
- Describe pop operation on linked stack
- Write an algorithm for pop operation on linked stack
- Application of stack

STACK

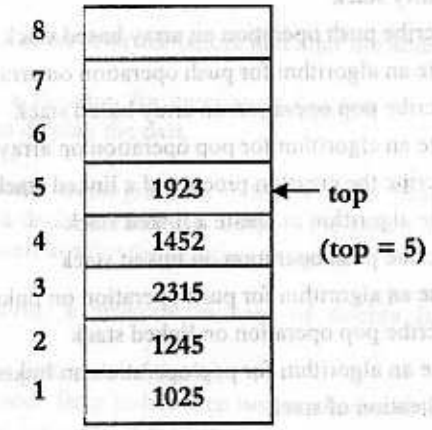
5.1 Definition

In our daily life we see stack (pile) of plates in cafeteria or restaurant, stack of books in book-shop. Even a packet of papers is also a stack of paper-sheet. A book is also a stack of written papers. When anybody takes a plate from a stack of plates, he takes it from the top. On the other hand, the person who cleans the plates, he puts it on the top of the stack.

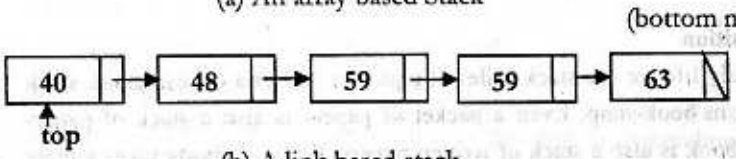
Stack is a linear list where any element is added at the *top* of the list and any element is deleted (accessed) from the *top* of the list. So, for stack an indicator or pointer must be used to indicate or point the *top element* of the stack. Add operation for a stack is called **'push'** operation and deletion operation is called **'pop'** operation. Stack is a LIFO (Last In First Out) structure. That means the element which was added last will be deleted or accessed first.

The elements of a stack are added from bottom to top, means *push* operation is performed from bottom to top. The elements are deleted from top to bottom, which means *pop* operation is performed from top to bottom.

Stack can be implemented in two ways; using array and using linked list.



(a) An array based Stack



(b) A link based stack

Figure-5.1: Graphical representation of Stack

5.2 Array based stack

The stack, which is implemented using array is called array based stack. To create an array based stack, at first we have to declare an array with required size.

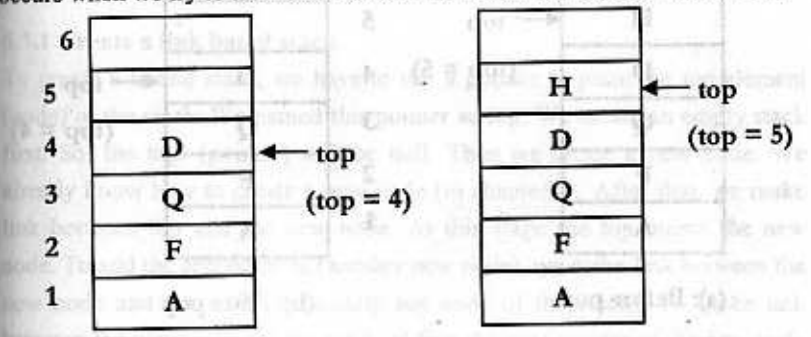
5.2.1 Push Operation

Push operation means to add an element to a stack.

Here, we shall use array based stack, so, an array will be treated as a stack. We need an indicator or index identifier to add element to the stack

and this indicator will mark the top of the stack. To add an element we have to check whether the array is already full or not. If the array is already full, then we can not add any element, otherwise we can.

Here, *top* is an indicator indicates the *top element* of the stack and *item* is an element to be added to the stack. *M* is the size of the stack (array). Overflow occurs when we try to insert an element into the stack, which is already full.



(a): Before push (b): After push

Figure-5.2: Pictorial view of push operation

Algorithm 5.1: Algorithm to add an element to a stack

1. Declare the stack and **top**:
   stack[1 . . . M], top;
2. Add an item in the stack:
   if (top < M)
   {
   top = top +1;
   stack [top] = item;
   }
   else print "Over Flow";
3. Output will be the updated stack.

5.2.2 Pop Operation

Pop operation means to delete (access) an element from a stack.

Here, *top* is an indicator indicates the *top element* of the stack, *M* is the size of the array and *x* is a variable where we access top element of the stack.

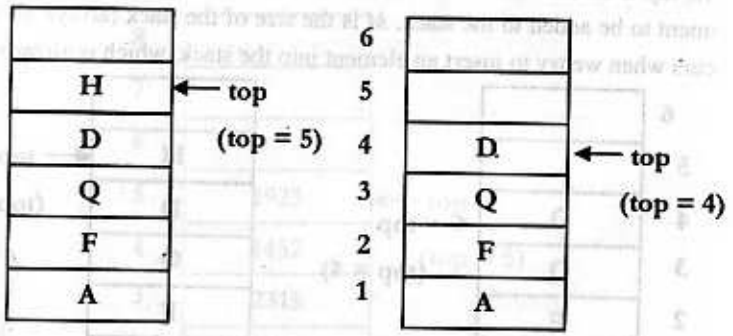


Figure-5.3: Pictorial view of pop operation

Algorithm 5.2: Algorithm to delete an element from a stack

1. Declare the stack and top:
   stack[1 . . . M], top;
2. Access the top element:
   if (top = = 0) print "stack is empty";
   else {
   x = stack[top];
   top = top – 1;
   }
3. Output updated list.

5.3 Link based stack

The stack that is created or implemented using a linked list is called a link based stack or linked stack.

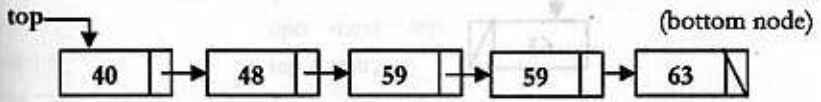
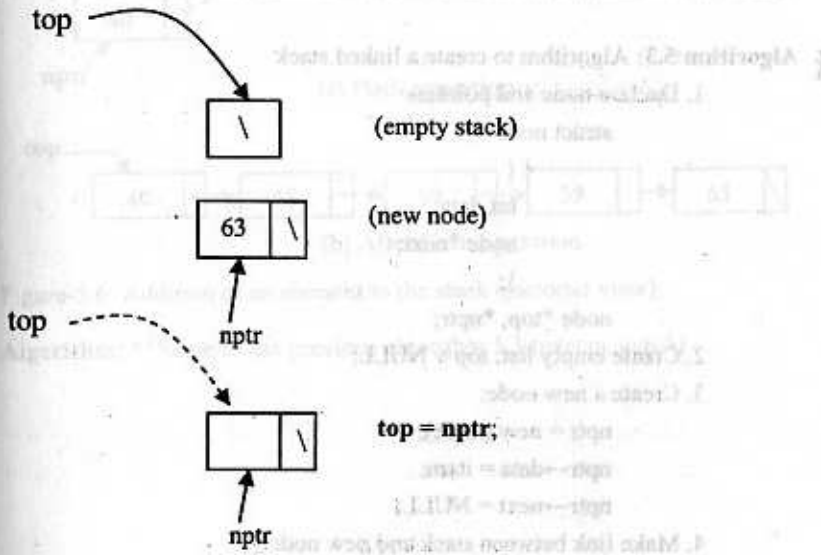


Figure-5.4: A link based stack

5.3.1 Create a link based stack

To create a linked stack, we have to use a pointer to point the top element (node) of the stack. We named this pointer as **top**. We create an empty stack first. So, the topr (pointer) will be null. Then we create a new node. We already know how to create a new node (in chapter-4). After that, we make link between top and the new node. At this stage the top points the new node. To add the second node (another new node), we make link between the new node and the stack (especially top node of the stack). To make link between the new node and the stack, at first the next pointer of the new node will point the top node of the stack, then top will point the new node. Thus we can create a linked stack.



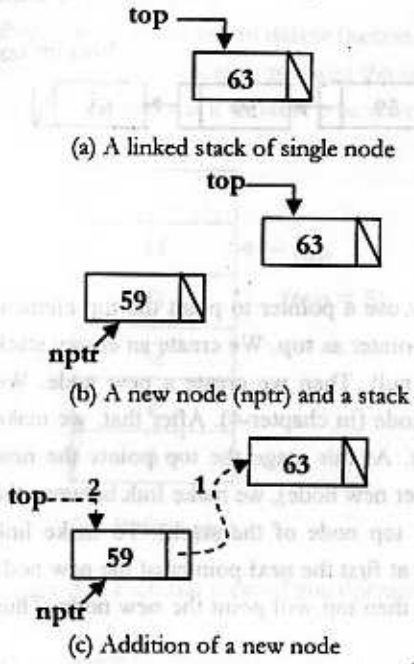


Figure-5.5: Pictorial view of linked stack (creation process).

Algorithm 5.3: Algorithm to create a linked stack

1. Declare node and pointers:

```
struct node
        {
        int data;
        node *next;
        };
node *top, *nptr;
```

2. Create empty list: top = NULL;
3. Create a new node:

```
nptr = new (node);
nptr→data = item;
nptr→next = NULL;
```

4. Make link between stack and new node:


```
if (top = = NULL), top = nptr;
else    {
        nptr→next = top;
        top = nptr;
        }
```

5. Repeat Step 3 and Step 4 to create stack of several nodes.
6. Output a linked stack

Comments: *top* is the pointer that points the top node of the stack and *nptr* is the pointer to the new node. *item* is an integer type variable.

5.3.2 Add an element to the stack (Push operation)

Push operation in a linked stack can be performed simply by adding a new node to the stack, which is already discussed when we create a linked stack.

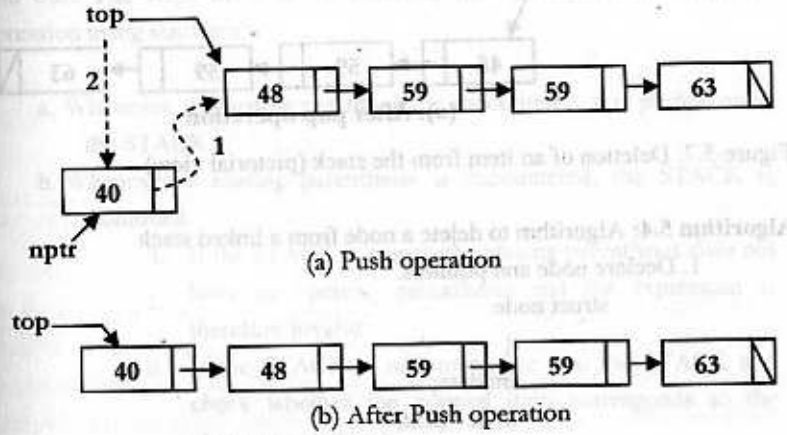
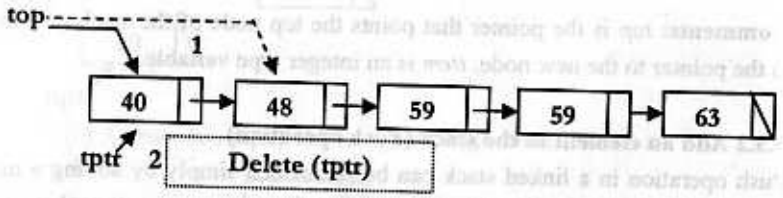


Figure-5.6: Addition of an element to the stack (pictorial view)

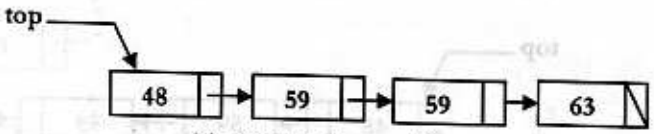
Algorithm: **Same as the previous algorithm 5.3 (except step 5).

5.3.3 Deletion of an item (Pop operation)

Pop operation in the linked stack is very simple. Pop operation means to delete top node from the linked stack. As we know the top (pointer), points the top node of the stack, so to delete top node, we use a temporary pointer that will point the top node and advance the top to the next node (using next pointer). Now the top is pointing the next node (next of the top node). We delete the top node using temporary pointer. Figure-5.7 shows deletion (*i.e.* pop) of an item from a link based stack.



(a): Pop operation



(a): After pop operation

Figure-5.7: Deletion of an item from the stack (pictorial view)

Algorithm 5.4: Algorithm to delete a node from a linked stack

1. Declare node and pointers:

```
struct node
        {
        int data;
        node *next;
        };
node *top, *tptr;
```

2. Input a linked stack
3. tptr = top;
4. Advance (move forward) the pointer, ***top*** to perform deletion:

```
if (top→next != NULL)
        {
        top = top→next;
        delete (tptr);
        }
else top = NULL;
```

5. Output updated linked list

Comments: *top* is a pointer that points the top node of the stack and *tptr* is a temporary pointer used to delete the node.

5.4 Applications of Stack

5.4.1 Checking the validity of an arithmetic expression

Using stack we can check the validity of an arithmetic expression. We know in a valid expression the parenthesis, brace or bracket must occur in pairs. That is, when there is an opening parenthesis; brace or bracket, there should be the corresponding closing parenthesis. Otherwise, the expression is not a valid one. The steps involved in checking the validity of an arithmetic expression using stack are:

a. Whenever an opening parenthesis is encountered, it is pushed on to the STACK.
b. Whenever a closing parenthesis is encountered, the STACK is examined.
   i. If the STACK is empty, the closing parenthesis does not have an opening parenthesis and the expression is therefore invalid.
   ii. If the STACK is nonempty, we pop the STACK and check whether the popped item corresponds to the closing parenthesis.
   iii. If a match occurs, we continue. Otherwise the expression is invalid.
c. When the end of the expression is reached, the STACK must be empty; otherwise one or more opening parenthesis does not have corresponding closing parenthesis and the expression is invalid.

Exp: [(A + B) – {C + D}] – [F + G]

Symbol Scanned	STACK
(1) [	[
(2) (	[, (
(3) A	[, (
(4) +	[, (
(5) B	[, (
(6))	[
(7) -	[
(8) {	[, {
(9) C	[, {
(10) +	[, {
(11) D	[, {
(12) }	[
(13)]	
(14) -	
(15) [	[
(16) F	[
(17) +	[
(18) G	[
(19)]	

In the above example we see the stack is empty at the end, so the expression is valid.

5.4.2 Converting an infix arithmetic expression to its postfix form

An arithmetic expression can be represented in various forms such as *prefix*, *infix* or *postfix*. The prefixes "pre-", "in-", and "post-" refer to the relative position of the operator with respect to its operands. If the operator is placed before its two operands then the expression is in *prefix* form, if the operator is placed in the middle it is known as *infix*, and if it is placed after the two operands it is known as postfix form. Let us consider a simple example with an operator "+" and two operands "A" and "B".

Prefix: + A B (operator before its operands)
Infix: A + B (operator in the middle of its operands)
Postfix: A B + (operator after its operands)

In an arithmetic expression we have to perform the operation on the operands with the highest precedence first. As for example, let us consider an expression 5 + 10 * 90 / 2. To evaluate this expression we have to perform the " * " or " / " first, then the " + " operation. Here " * " and " / " are said to have higher precedence than " + ". Moreover if parenthesis is present in the expression then we have to consider the operations within the parenthesis first. Precedence rules of operators are applied within the parenthesis. This precedence of operators and parenthesis are important to convert an arithmetic expression to various forms (*prefix*, *infix* or *postfix*). The precedence rules of operators can be stated as follows.

Highest: Exponential (^ or ↑)
Multiplication (* or ×) or Division (/ or ÷)
Lowest: Addition (+) or Subtraction (-)

Operators with the same precedence (*, / or +, -) are evaluated according to their order of occurrence in the expression. For the previous expression we shall compute 10 * 90 (=900) first, then 900/2 (=450), finally we get the result by computing 5 + 450 (=455).

We can convert an *infix* expression to *postfix* form using stack. We start to scan the expression from left to right. In an expression, there may be some operands, operators and parenthesis (opening or closing). Each time we get an operand it is added to the *postfix* expression. When we get an operator, we should check the top of the stack. If the operator at the top of the stack has the same or higher precedence than the current operator then we repeatedly pop from the stack and add it to the *postfix* expression, otherwise the current operator is pushed onto the stack. When an opening parenthesis is encountered it is pushed onto the stack and when the corresponding closing parenthesis is encountered we repeatedly pop from the stack and add the operators from the stack to the *postfix* expression. The corresponding opening parenthesis is deleted from the stack. The steps involved in the process discussed above can be written as follows.

Scan the expression from left to right. We shall get a symbol which may be an operand or a parenthesis (opening or closing) or an operator. The symbol is treated as follows.

Steps:

a) if the symbol is an operand, add it to the *postfix* expression.

b) if the symbol is an opening parenthesis, push it on to the stack.

c) if the symbol is an operator, then check the top of the stack
   i) if the precedence of the operator at the top of the stack is higher or the same as the current operator then repeatedly it is popped and added to the *postfix* expression.
   ii) otherwise, it is pushed onto the stack.

d) If the symbol is a closing parenthesis, then
   i) Repeatedly pop from the stack and add each operator to the *postfix* expression until the corresponding opening parenthesis is encountered.
   ii) Remove the opening parenthesis from the stack.

Let us consider the expression which we have converted by inspection.
Expression: 5 * (6 + 2) - (12/4)

At first we find "5" which is an operand so we add "5" to the *postfix* expression (step a). Next we get " * ", which is an operator, it is pushed onto the stack (step c(ii)). Then we get an opening parenthesis, we should push it on to the stack (step b). We proceed to the next symbol which is "6", an operand so it is treated as the previous operand "5". Then we get an operator " + ", we should add it to the *postfix* expression (step a). Next the operand "2" is added to the *postfix* expression. When we get the closing parenthesis after "2", we repeatedly pop from the stack and add the operators to the *postfix* expression (step d (i)). Then the corresponding opening parenthesis is removed from the stack (step d (ii)). The complete process is shown below.

Symbol Scanned	Stack	Postfix Expression
1. 5		5
2. *	*	5
3. (	*, (	5
4. 6	*, (	5, 6
5. +	*, (, +	5, 6
6. 2	*, (, +	5, 6, 2
7.)	*	5, 6, 2, +
8. -	-	5, 6, 2, +, *
9. (	-, (	5, 6, 2, +, *
10. 12	-, (	5, 6, 2, +, *, 12
11. /	-, (, /	5, 6, 2, +, *, 12
12. 4	-, (, /	5, 6, 2, +, *, 12, 4
13.)	-	5, 6, 2, +, *, 12, 4, /
14.		5, 6, 2, +, *, 12, 4, /, -

5.4.3 Evaluating a postfix expression

We can evaluate a postfix expression using stack. Let us consider the *postfix* expression we obtain in the previous section.

5, 6, 2, +, *, 12, 4, /, -

To evaluate the expression we scan the expression from left to right. The steps involved in evaluating a postfix expression are:

a. If an operand is encountered, push it on STACK
b. If an operator '*op*' is encountered
   i. Pop two elements of STACK, where A is the top element and B is the next top element.
   ii. Evaluate B *op* A.
   iii. Push the result on STACK.
c. The evaluated value is equal to the value at the top of STACK.

Symbol Scanned	STACK	Operation (B *op* A)
(1) 5	5	
(2) 6	5, 6	
(3) 2	5, 6, 2	
(4) +	5, 8	[6+2] (A=2, B=6)
(5) *	40	[5*8] (A=8, B=5)
(6) 12	40, 12	
(7) 4	40, 12, 4	
(8) /	40, 3	[12/4] (A=4, B=12)
(9) -	37	[40-3] (A=3, B=40)

Summary:

Stack is a linear list where the elements can be inserted or deleted from a specially designated position called top. We can "push" (insert) elements to a stack or "pop" (delete) elements from a stack.

Stack can be **implemented** using array or linked lists.

Stack can be **used** in various problems such as, checking the validity of an arithmetic expression, converting an infix expression to its postfix form or evaluating a postfix expression etc.

Questions:

1. Is stack a data structure? Why?
2. Write algorithms to perform push and pop operations for a stack when stack in an array based structure.
3. "All stacks are lists, but all lists are not stacks", explain this statement with examples.
4. Write algorithm(s) for push and pop operations when stack is a linked structure.
5. Write an algorithm to push an item to a stack, where stack is a linked structure.
6. Write algorithm(s) to add a node and to delete a node from a linked stack.
7. Write an algorithm to delete an element form a stack, when stack is a linked structure.

8. Convert the following infix expression into its equivalent postfix expression and evaluate the postfix expression. Use stack for the operations. 12/(7-3+2 * (1+5)
9. Show all the steps to evaluate the following postfix expression using postfix expression evaluation algorithm:

   ABC + * CBA - + *

   Assume A = 1, B = 2 and C = 3
10. Write the algorithm which transforms infix expression into postfix expression.
11. What do you mean by infix, prefix and postfix notations for an arithmetic expression? Explain.
12. Suppose, we have the following arithmetic infix expression Q:

    Q: A + (B * C - (D / E ↑ F) * G) * H

    Devise an algorithm to transform Q into its equivalent postfix expression P.

Problems for Practical (Lab) Class

Stack related problems

Problem 5-1: Create an array based stack with some integer data in ascending order. Print the data in descending order.

Problem 5-2: Solve the **problem 5-1** using linked stack.

Problem 5-3: Create an array based stack with character data and print it in reverse order. Your program must give correct output for all valid input.

Sample input: American International
Sample output: lanoitanretnI naciremA

Problem 5-4: Solve the **problem 5-3** using linked stack.

Problem 5-5: Given a mathematical expression, print the output as "valid" if the expression is valid otherwise print "invalid". In case of invalid, print out the reason(s). Hints: take input as [(a + b) - (c - d) + e] and validate expression using stack. Your program must give correct output for any input expression.

Problem 5-6: Evaluate arithmetic expression using stack. As for example take input 5 + (10 * 2 + 5)*8 / 2; convert the infix to postfix expression and evaluate it using stack. The output of the above expression will be 105. Your program must give correct output for any valid input.

CHAPTER SIX
QUEUE

OBJECTIVES:

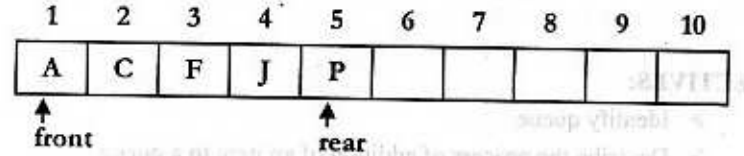
- Identify queue
- Describe the process of addition of an item to a queue
- Write an algorithm to add an item to an array based queue
- Describe the process of deletion of an item from an array based queue
- Write an algorithm to delete an item from an array based queue
- Drawbacks of array implementation of queue
- Describe the creation process of linked queue
- Describe the process of addition of an item to a linked queue
- Write an algorithm to add an item to a linked queue
- Describe the process of deletion of an item from a linked queue
- Write an algorithm to delete an item from a linked queue

QUEUE

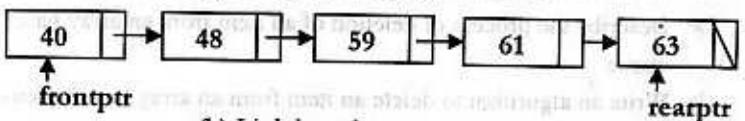
In our daily life, when we stand on line to get into the bus or to take money from bank counter, we make queue. The man who stands first will get into the bus first, and the man who stands last will get into the bus last. In other words, the element which is added first will get service first and the element which is added last will get service last.

Queue is a linear list where all additions are made at one end, called *rear*, and all deletions (accesses) are made from another end called *front* of the list. So, in a queue there must be two indicators or pointers. One is *rear* used to add elements and another is *front* used to delete (access) the elements from the queue.

Queue is a FIFO (First In First Out) structure. That means the element that is added first will be deleted (accessed) first. As stack, queue can be implemented using array and linked list.



(a) Array based queue



(b) Link based queue

Figure-6.1: Graphical representation of queue

6.1 Array based queue

The queue that will be created using an array is called array based queue. Here, we have to use two indicators (two index identifiers). One indicator will mark the *front* element and another will mark the *rear* element of the queue.

6.1.1 Addition of an element in an array based queue

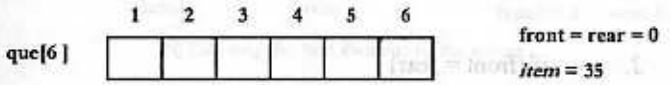
We know in a queue element is added at the *rear*. So, to add an item at first we increase rear index and then place the element in the array-based queue using the rear index.

Algorithm 6.1: Algorithm to add an element to queue

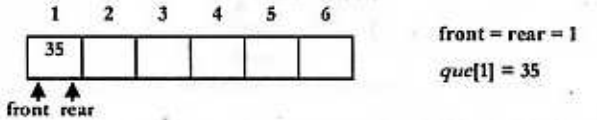
1. Declare array based queue and other variables:
   que[1……M], item, front, rear;
2. if (rear = 0)
   {
   front = rear =1;
   que[rear] = item;
   }

3. if (rear < M)
   {
   rear = rear +1;
   que[rear] = item;
   }
   else print "Over Flow message"
4. Output: Updated queue.

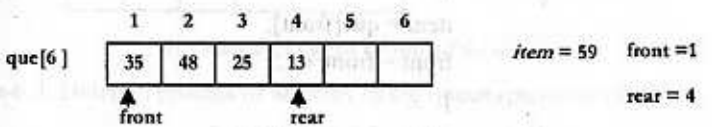
Comments: Here *que*[] is an array to make a queue and *M* is the size of the queue (array); *item* is a variable to add (insert) data in the queue. *front* and *rear* are two variables to indicate the first and last elements of the queue.



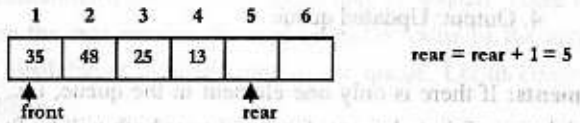
(a) An empty queue and item (35)



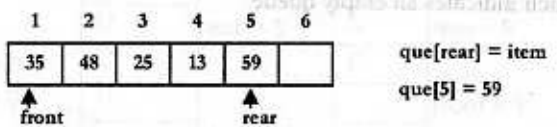
(b) Adding the first element of the queue



(c) An array based queue and item (59)



(d) Advancement of the rear



(e) Insert *item* in que[rear]

Figure-6.2: Addition of new items in an array based Queue (Pictorial view)

6.1.2 Deletion of an element from a queue

We know that, the element is deleted from the *front* of the queue. So, at first we access the element, and then we increase the front index.

Algorithm 6.2: Algorithm to delete an element from a queue

1. Declare array based queue and other variables:
   que[1……M], item, rear, front;

2.
```
if (front = rear)
        {
        item = que[front];
        front = rear = 0;
        }
```

3.
```
if (front != 0)
        {
        item = que[front];
        front = front + 1;
        }
else    print "Queue is empty"
```

4. Output: Updated queue

Comments: If there is only one element in the queue, *i.e.*, *front* = *rear* and after deletion of that element, *front* and *rear* both will be 0; *i.e.*, *front* = *rear* = 0, which indicates an empty queue.

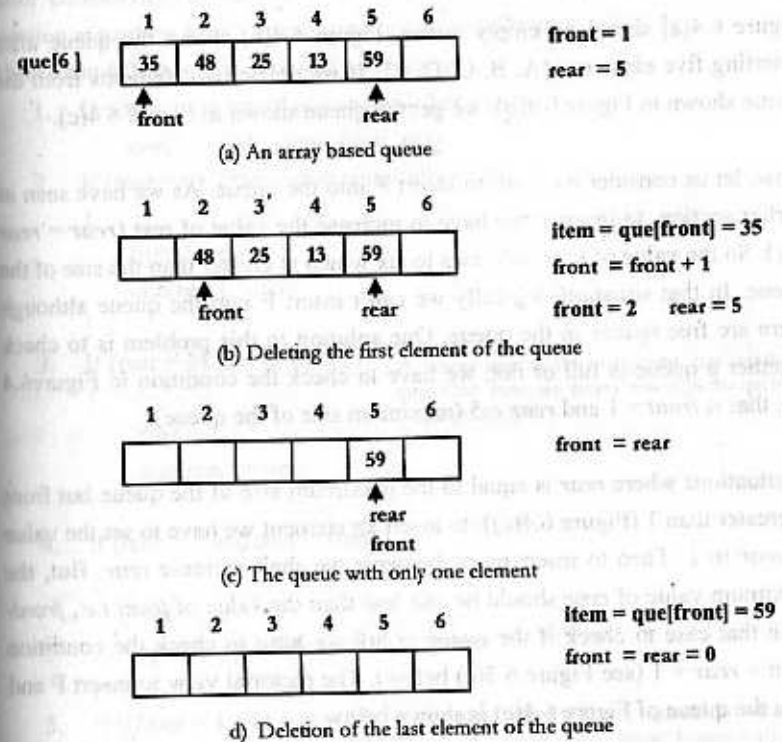


Figure-6.3: Deletion process of an array based queue (pictorial view)

6.1.3 Drawback of array implementation of queue

In array implementation of queue, some space may be wasted. There may be situations, where the rear reaches the highest index value of the array, but free spaces are available at the beginning of the queue. Let us consider the following situations.

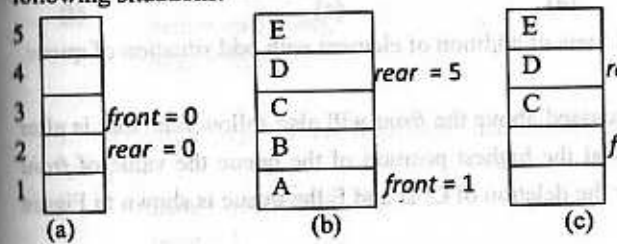


Figure-6.4: Pictorial view with odd situation of queue

Figure 6.4(a) shows an empty queue. Figure 6.4(b) shows the queue after inserting five elements {A, B, C, D, E}. If we delete two elements from the queue shown in Figure 6.4(b), we get the queue shown in Figure 6.4(c).

Now, let us consider we want to insert *F* into the queue. As we have seen in earlier section, to insert *F* we have to increase the value of rear (*rear* = *rear* + 1). So the value of *rear* increases to six which is greater than the size of the queue. In that situation, logically we can't insert F into the queue although there are free spaces in the queue. One solution to this problem is to check whether a queue is full or not, we have to check the condition in Figure6.4 (b), that is *front* = 1 and *rear* = 5 (maximum size of the queue).

In situations where *rear* is equal to the maximum size of the queue but front is greater than 1 (Figure 6.4(c)), to insert an element we have to set the value of *rear* to 1. Then to insert more elements we shall increase *rear*. But, the maximum value of *rear* should be one less than the value of *front i.e.*, *front*-1. In that case to check if the queue is full we have to check the condition *front* = *rear* + 1 (see Figure 6.5(e) below). The pictorial view to insert F and G in the queue of Figure 6.4(c) is shown below.

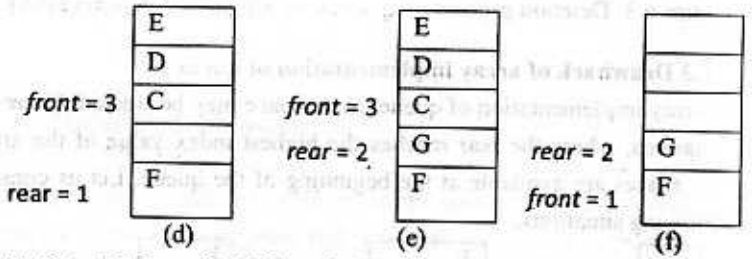


Figure-6.5: Pictorial view of addition of element with odd situation of queue

For the situation discussed above the *front* will also follow *rear* that is after deleting the element at the highest position of the queue the value of *front* should be one. After the deletion of C, D and E the queue is shown in Figure 6.5(f).

After considering the above situations, we can modify the insertion and deletion algorithm for an array based queue as follows.

Algorithm 6.3: Algorithm to add an element

1. Declare array based queue and other variables:
   que[1 . . . M], item, front, rear;
2. If (rear = 0) //the queue is initially empty
   {
   front = rear = 1;
   que[rear] = item;
   }
3. If (rear = M and front > 1) /* where rear is the maximum size of the queue but there are empty spaces in the queue (Figure 6.4(c))*/
   {
   rear = 1;
   que[rear] = item;
   }
4. If (rear !=1 and rear < front - 1)
   {
   rear = rear +1;
   que[rear] = item;
   }
5. If ((front = 1 and rear = M) or (front = rear +1)) /* queue is already filled? Figure 6.4(b) or Figure 6.5(e)*/
   print "Over Flow message"
6. Output: Updated queue.

Algorithm 6.4: Algorithm to delete an element

1. Declare array based queue and other variables:
   que[1 . . . M], item, front, rear;
2. If (front = rear)
   {
   item = que[front];
   front = rear = 0;
   }
3. If (front = M) /* where front is the maximum size of the queue*/
   {
   item = que[front];
   front = 1;
   }

```
4. If (front !=0 or front != 1)
   {
       item = que[front];
       front = front + 1;
   }
   else print "Queue is empty"
```

Output: Updated queue.

6.2 Link based queue

The queue that will be created using a linked list is called link based queue or linked queue. In a linked queue we use two pointers, one is *frontptr* points the front (first) node of the queue and another is *rearptr* points the rear (last) node of the queue.

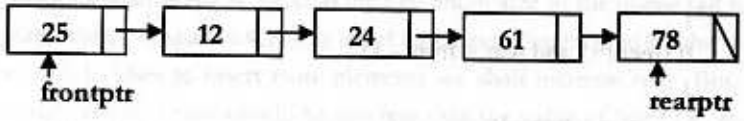


Figure-6.6: A linked queue

6.2.1 Create a link based queue

As we know in a linked queue, there are two pointers, *frontptr* and *rearptr*, so to create a linked queue we have to use these two pointers. At first we create an empty (linked) queue. So, the *frontptr* and *rearptr* both will be null. After that, we create a new node with an external pointer, *nptr*. Now, we shall add this new node to the queue. To add the new node, we assign the value of *nptr* to *frontptr* and *rearptr*. Thus, *frontptr* and *rearptr* both point the new node. Now we have a node in the queue. To add second node, we create another new node and add this node to the queue. To add the node we make link between the existing queue and the new node. Since the new node will be the rear element (node), so the next pointer of the new node will point the rear node (at present only one node) of the existing queue and the *rearptr* will point the new node. That means, at present the *frontptr* points the front (first) and the *rearptr* points the rear (second) node of the queue. Thus we can add another node and create the linked queue.

Algorithm 6.5: Algorithm to create a linked queue

1. Declare node and pointers:

```
struct node
        {
        int data;
        node *next;
        };
node *frontptr, *rearptr, *nptr;
```

2. Create an empty queue:

```
frontptr = NULL;
rearptr = NULL;
```

3. Create a new node:

```
nptr = new (node);
nptr→data = item;
nptr→next = NULL;
```

4. Make link new node with the **rearptr** and **frontptr**:

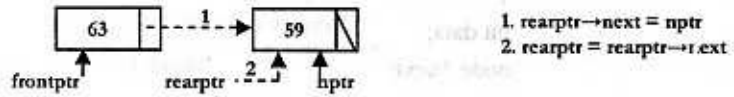
```
if (rearptr = = NULL)
        {
        rearptr = nptr;
        frontptr = nptr;
        }

else    {
        rearptr→next = nptr;
        rearptr = nptr;
        }
```

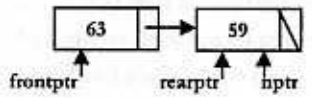
5. Repeat Step-3 and 4 to create queue with several nodes
6. Output: A link based queue

Comments: Here, *frontptr* is a pointer to the front node and *rearptr* is a pointer to the last node of the queue. *nptr* is a pointer that points to the new node and item is a variable used to enter data to the new node.

(a) A linked queue of single node and a new node (nptr)



(b) A new node (nptr) and a stack of single node



(c) Updated linked queue

Figure-6.7: Pictorial view of linked queue (creation process)

6.2.2 Add a new node to linked queue

Addition of new node to the linked list is similar to the creation process where we add a new node at each stage to create linked queue.

Algorithm 6.6: Algorithm to add a node to linked queue

1. Declare node and pointers:

```
struct node
        {
        int data;
        node *next;
        };
node *frontptr, *rearptr, *nptr;
```

2. Input a linked base queue
3. Create a new node:

```
nptr = new (node);
nptr→data = item;
nptr→next = NULL;
```

4. Make link among the necessary pointers:

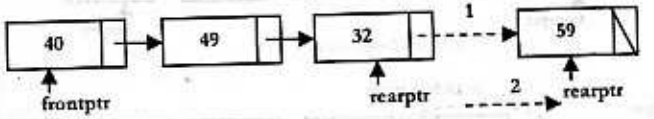
```
if (rearptr = = NULL)
```



```
        {
        rearptr = nptr;
        frontptr = nptr;
        }
else    {
        rearptr→next = nptr;
        rearptr = nptr;
        }
```

5. Output: Updated linked list

(a) A linked queue and a new node



(b) Making link between queue and new node.

Figure-6.8: Add a new node to a linked queue

6.2.3 Delete a node from a linked queue

As we know in a queue deletion operation must be performed from *front* of the queue, so to delete the front node of the linked queue, we have to use a temporary pointer that will point the front node. After that we advance the *frontptr* pointer to the next node. Now just delete the node using temporary pointer.

Algorithm 6.7: Algorithm to delete a node from a linked queue

1. Declare node and pointers:

```
struct node
        {
        int data;
        node *next;
        };
```

node *frontptr, *rearptr, *tptr;

2. Input a linked base queue

3. Use temporary pointer, tptr and advance (move forward) the forntptr:

tptr = frontptr;

frontptr = tptr→next;

4. delete (tptr);

5. Output: Updated linked queue.

Comments: Here, *frontptr* is the pointer to the front node and *rearptr* is the pointer to the last node of the queue. *tptr* is a temporary pointer that points to the first node to be deleted.

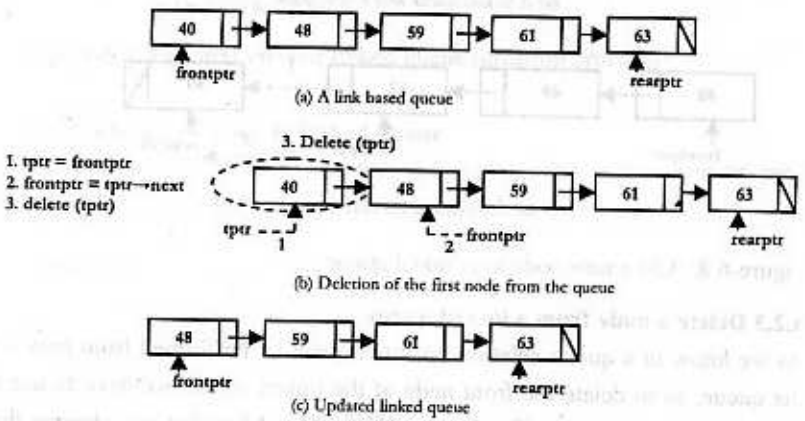


Figure-6.9: Deletion of node from a linked queue (pictorial view)

Summary:

Queue is a list where the elements can be inserted into a specially designated position called the *rear* of the queue and deleted from a specially designated position called the *front* of the queue. Queue can be implemented using array and linked list. Queue can be used in various problems where first in first out implementation is required, because queue is a first in first out (FIFO) structure.

Questions:

1. Define queue. Give example.
2. List some differences between stack and queue.
3. Show the disadvantages of array implementation of a queue. Also give the possible solutions to overcome these disadvantages.
4. Write a function that returns the number of elements is a queue that has been created previously.
5. Define linked queue with example.
6. Write algorithm to add a node to queue when queue is a linked list.
7. Write an algorithm to delete a node from a queue, when queue is a linked list.
8. Write algorithm(s) to insert an element and delete another element from a queue where queue is an array based structure.

CHAPTER SEVEN
TREE

OBJECTIVES:

- Identify tree
- Describe binary tree traversal methods
- Identify binary search tree (BST)
- Describe the addition process of a node to a BST
- Write an algorithm to add a node to a BST
- Describe the deletion process of a node from a BST
- Write an algorithm to delete a node from a BST
- Describe heap creation process
- Write an algorithm to create a heap
- Describe the process of heap sort
- Write an algorithm for heap sort

TREE

We see tree in nature. The tree has root, branches, sub-branches and leaves. From the concept of natural tree, the computer scientists get the idea of a data structure, which is graphically similar to natural tree. Natural tree is a bottom-up figure. However, the graphical representation of the data structure tree is a top-down figure.

A tree is a finite collection of nodes that reflects one to many relationship among the nodes. It is a hierarchical structure. An ordered tree has a specially designated node called root mode. The root node may have one or more child nodes. The connection line between two nodes is called edge.

The nodes of a level are connected to the nodes of the upper level.

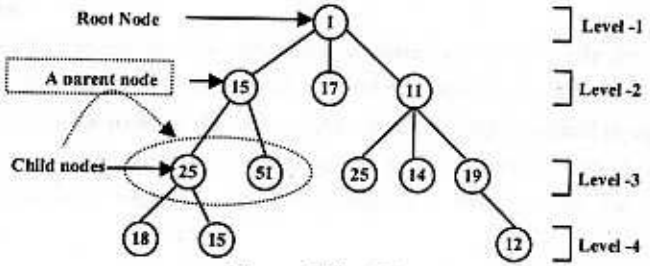


Figure-7.1: A Tree

The node that has no child node is called *leaf* node. If a node has child node is called *parent* node.

A tree can be implemented (stored in memory) as an *array* or a *linked list*.

7.1 Binary Tree

A binary tree is finite set of nodes, where there is a special node called root node and every node (including root node) has at best two children. The trees excluding root node are called left sub-tree and right sub-tree.

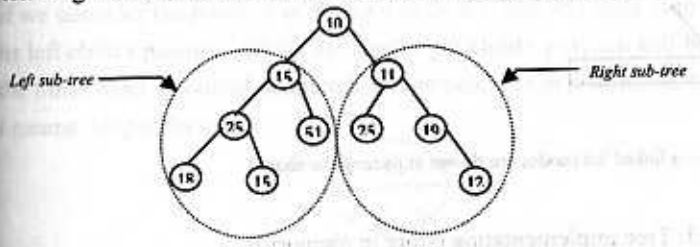
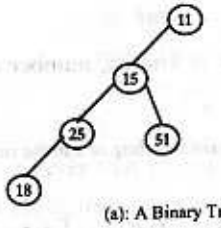
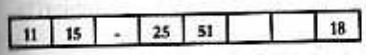


Figure-7.2: A binary tree

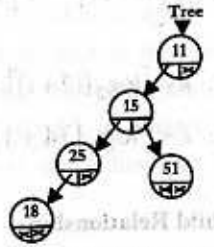
A binary tree can be implemented as an array or as a linked list. The tree in Figure-7.3(a), can be implemented as an array as shown in Figure-7.3(b). Whereas, Figure-7.3(c) shows tree implementation as linked list. In array implementation, memory space may be unused (wasted) as shown in Figure-7.3(b).



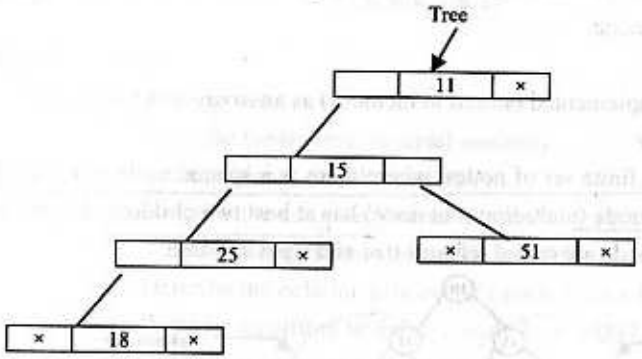
(a): A Binary Tree



(b): Tree as an array



(c): Tree as a linked list (nodes are shown in round shape)



(d) Tree as a linked list (nodes are shown in rectangular shape)

Figure-7.3: Tree implementation (store in memory)

If there are k levels in a binary tree, then the maximum number of nodes in the tree is as follows:

$$n = 2^k - 1.$$

For example, if $k = 3$, then the maximum number of nodes (n) is 7 and for $k = 4$, $n = 15$ and so on.

When the maximum number of nodes is known, number of levels can be calculated as –

$$k = \lceil \log_2 (n+1) \rceil \quad // \lceil x \rceil \text{ means ceiling of } x \text{ to the next integer}$$

For example,

when $n = 17$; $k = \lceil \log_2 (17+1) \rceil = \lceil \log_2 (18) \rceil = \lceil 4.1..... \rceil = 5$

when $n = 34$; $k = \lceil \log_2 (34+1) \rceil = \lceil \log_2 (35) \rceil = \lceil 5.2.... \rceil = 6$; etc.

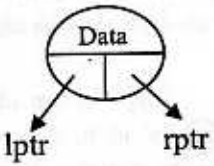
7.1.1 Parent-Child Relationship

If we consider the root's position (index) is 1. A node is in position (index) i of an array, then the position of its left child will be $2i$ and the position of its right child will be $2i + 1$. Thus

the position of its parent node will be $\left\lfloor \frac{i}{2} \right\rfloor$. $\lfloor x \rfloor$ means floor of x to the previous integer. The position of root node is 1 and its children's positions are 2 and 3. If a node value stored in position 4, then the positions of its children will be 8 and 9 and so on. When a node is stored in position 13, then the position of its parent is-

$$\left\lfloor \frac{13}{2} \right\rfloor = \lfloor 6.5 \rfloor = 6$$

If we consider the position of the root node is o and any node is in position i, its left child's position will be 2i+1 and right child's position will be 2i+2. On the other hand a node of position k has its parent is in position (k-1)/2, where / means integer division.



A binary tree can also be implemented (stored in computer's memory) as a linked list. The node of the binary tree can be defined (declared) as follows:

```
struct node
    {
    int data;
    node *lptr;
    node *rptr;
    };
```

where ***lptr*** is a pointer that points the left child and ***rptr*** is a pointer that points the right child. ***data*** is an integer variable to store value of the node. That means, a node has three parts. One is data part (value), the second is a pointer to the left side (node) and the third is a pointer to the right side (node). *Linked implementation* of binary tree other than complete tree is efficient.

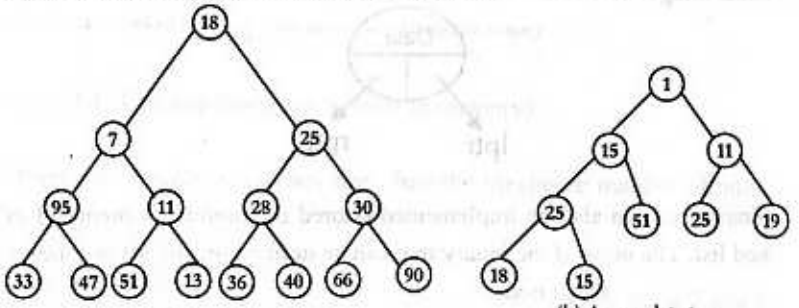
Full binary tree:

If a binary tree contains nodes in such a way that every node has at least two children except the nodes at the last one deepest level (the leaves are at the deepest level), then the tree is called a full binary tree.

Complete binary tree:

If a binary tree contains nodes in such a way that every level except the deepest has as many nodes as possible and the nodes of the deepest level are in as left as possible, the tree is called complete binary tree. All full binary trees are complete binary trees.

If a binary tree is a complete binary, then its *array implementation* is efficient; as there will be no wastage of memory.



(a) A full tree (b) A complete tree

Figure-7.4: Pictorial view of full tree and complete tree

7.1.2 Traversal technique of a binary tree

There are three main traversal techniques (methods) for a binary tree. Such as—

1. Pre-order Traversal Method
2. In-order Traversal Method
3. Post-order Traversal Method

7.1.2.1 Pre-order traversal method

In pre-order tree traversal method the points below are to be followed:

i. Visit the root,
ii. Traverse the left sub-tree (in pre-order),
iii. Traverse the right sub-tree (in pre-order).

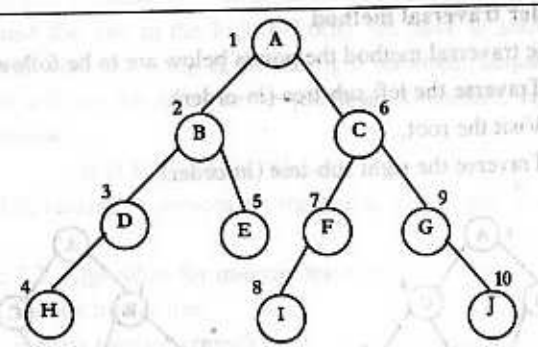


Figure-7.5: Pre-order traversal method

In pre-order traversal method, we have to visit the root node first. Then we traverse the left sub-tree following the above 3 points (i, ii, iii) recursively. After that we traverse the right sub-tree by following the above three points recursively.

As for example, to traverse the tree in Figure-7.5, at first we visit the root of the tree, then we visit the root node of the left sub-tree, where the node value is B. Then we will go to the left sub-tree of the tree with node value D. In such way we will complete the traversal process of the left sub-tree.

After that, we will visit the node value C, which is the root of right sub-tree and similarly other nodes will be visited by following the three points (i, ii, iii). Thus visiting sequence of the node values will be

A B D H E C F I G J

In Figure-7.5, visiting sequences are marked as 1, 2, 3 and so on.

Algorithm 7.1: Algorithm for pre-order traversal

1. Input a binary tree
2. preorder (node * curptr)

```
{
if (curptr!= NULL)
    {
    print curptr→data;
    preorder (curptr→lchild);
    preorder (curptr→ rchild);
    }
}
```

3. Output: the information of the nodes.

7.1.2.2 In-order traversal method

In in-order tree traversal method the points below are to be followed:

i. Traverse the left sub-tree (in-order),
ii. Visit the root,
iii. Traverse the right sub-tree (in-order).

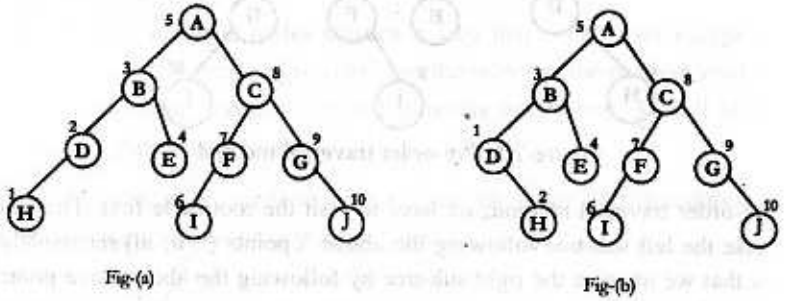


Figure-7.6: In-order traversal method

In in-order traversal method, we have to visit the left sub-tree first. So, we have to start from the last node at left of the left sub-tree. Then we visit the root and at last we will visit the right sub-tree by following the above mentioned three points (i, ii, iii). Each time we traverse a node, we must think its parent node as the root of that tree-section. The left children of that root will be treated as left sub-tree and the right children will be treated as the right sub-tree.

As for example, to traverse the tree in the Figure-7.6(a), we have to start the last node at left (**H**) of the left sub-tree. Then we visit its parent node, **D** (as the root of the tree-section). In the same way we visit the node, **B** as the root of **D** and at last we have to visit the right sub-tree of node **D**. That means now we shall visit node **E** and we shall continue to traverse the whole tree.

In in-order traversal method, the visiting sequence of the tree in the Figure-7.6(a) will be as follows

H D B E A I F C G J

If we traverse the tree in the Figure-7.6(b), we have to start from node **D**. Then we will visit the node **H** as the right sub-tree. Sequence of visiting other nodes will remain same as the previous. Therefore, visiting sequence will be as follows —

D H B E A I F C G J

In Figure-7.6, visiting sequences are marked as 1, 2, 3 and so on.

Algorithm 7.2: Algorithm for in-order traversal

```
1. Input a binary tree.
2. inorder (node * curptr)
{
if (curptr!= NULL)
        {
        inorder (curptr→lchild);
        print curptr→data;
        inorder (curptr→rchild);
        }
}
3. Output: the information of the nodes.
```

7.1.2.3 Post-order traversal method:

In post-order tree traversal method the points below are to be followed:

i. Traverse the left sub-tree (in post-order),
ii. Traverse the right sub-tree (in post-order),
iii. Visit the root.

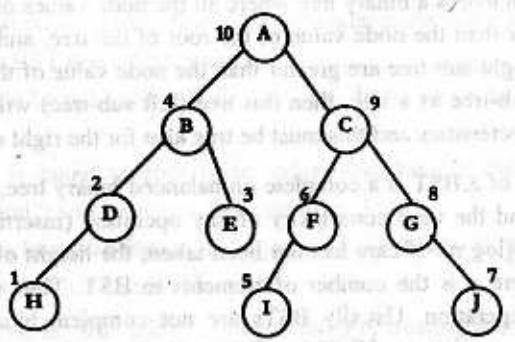


Figure-7.7: Post-order traversal method

In post-order traversal method, we shall start visiting the tree from the left sub-tree. Then we visit the right sub-tree and at last we visit the root of the tree by following above mentioned three points (i, ii, and iii).

For example, to traverse the tree in Figure-7.7, we shall start the left most node of the left sub-tree, **H** and we traverse the left sub-tree using the points i, ii, and iii. After that we visit the right sub-tree accordingly. At last we shall visit the root of the tree, **A**.

According to post order method, visiting sequence for the tree in Figure-7.7 will be as follows —

H D E B I F J G C A

In Figure-7.7, visiting sequences are marked as 1, 2, 3 and so on.

Algorithm 7.3: Algorithm for post-order traversal

1.Input a binary tree.
2. postorder (node * curptr)
{
if (curptr!= NULL)
{
postorder (curptr→lchild);
postorder (curptr→rchild);
print curptr→data;
}
}
3. Output: the information of the nodes.

7.2 Binary Search Tree (BST)

A binary search tree is a binary tree where all the node values of the left sub-tree are smaller than the node value of the root of the tree, and all the node values of the right sub-tree are greater than the node value of the root. If we treat the left sub-tree as a tree, then this tree (left sub-tree) will also follow the above characteristics and this must be true also for the right sub-tree.

If the structure of a BST is a complete on balanced binary tree, it gives best performance and the time complexity of any operation (insertion, deletion, searching) is O(log n). If care has not been taken, the height of a BST may become n, where n is the number of elements in BST. Thus it takes O(n) time for any operation. Usually BSTs are not complete binary trees, so linked list implementation of BST is efficient. Here, we assume that the BST is stored in memory as a linked list and for this type of implementation we shall write the algorithms for *operations on BST*.

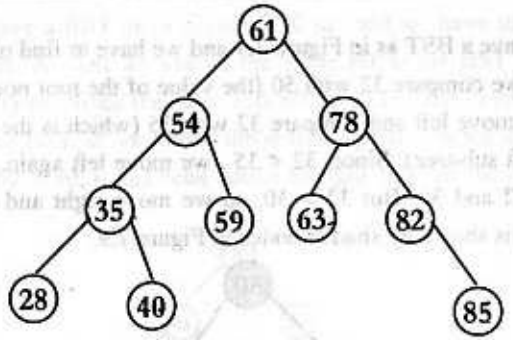


Figure-7.8: A binary search tree (BST)

7.2.1 Searching a particular node value of BST

Suppose that we are given a BST and value of a node, we have to determine whether the node exists in the BST or not. We know that in a BST each data in left sub-tree is smaller than the data in root and each data in right sub-tree is greater than the data in the root. So, to find out the target data, at first we compare the target value with the data in the root node, if they are equal, then the searching is successful and terminated. On the other hand, if the target value is smaller than the value of the root, then we search the target value in the left sub-tree in the same manner which is done for the tree. Otherwise, we search the target value in the right sub-tree in the same manner which is done for the tree.

Algorithm 7.4: Algorithm to find out a particular node value of BST

1. Input BST and a node value, *x*;
2. Repeat Step-3 to Step-5 until we find the value or we go beyond the tree.
3. If *x* is equal to root node value, searching is successful (print "Found") and terminate the algorithm.
4. If *x* is less than root node value, we have to search the left sub-tree (by treating it as a BST).
5. Else we have to search right sub-tree (by treating it as a BST).
6. Otherwise the node value is not present in BST (print "Not Found").
7. Output: Print message "FOUND" or "NOT FOUND"

Suppose, we have a BST as in Figure 7.9 and we have to find out the value x = 32. At first we compare 32 with 50 (the value of the root node). Since, 32 < 50, then we move left and compare 32 with 35 (which is the value of root node of the left sub-tree). Since, 32 < 35 , we move left again. At this point we compare 32 and 30. But 32 > 30, so we move right and find 32. The searching path is shown by shaded nodes in Figure 7.9.

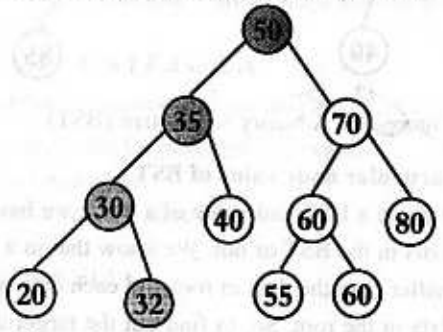


Figure-7.9: Searching of a node value (shaded nodes show the searching path

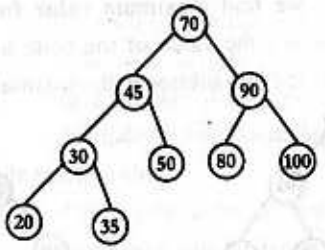
7.2.2 Add a node to a BST

To add a node to a BST, we have to find the proper position for the node. To do this, we use searching method which is stated above. If the value to be added is already present in the BST, the node should not be added. Otherwise, when we find last node in the searching path, we shall add the node on its (last node's) left or right.

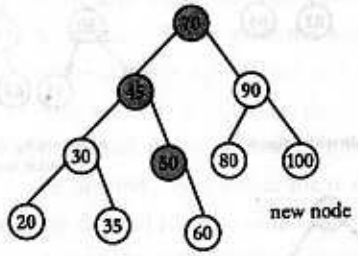
Algorithm 7.5: Algorithm to insert (add) a node to a BST

1. Input BST and a new node;
2. Repeat Step-3 to Step-5 until we find the value or we go beyond the tree.
3. If x is equal to root node value, searching is successful and terminate the algorithm.
4. If *x* is less than root node value, we have to search the left sub-tree (by treating it as a BST).
5. Else we have to search right sub-tree (by treating it as a BST).
6. Make link between the new node and the parent node of the new node.
7. Output: Updated BST.

Suppose, we have a BST as in Figure 7.10 (a) and we have to add 60 as a value of a node. At first we search the value 60 in the BST according to searching procedure. Since the node with 60 does not exist in the BST, so we can add it in the BST. If we notice the searching path shown in Figure 7.10 (b), the value 60 > 50, it has been added as right child of the node with the value 50.



a) A binary search tree



b) Addition of a new node (shaded nodes show the searching path).

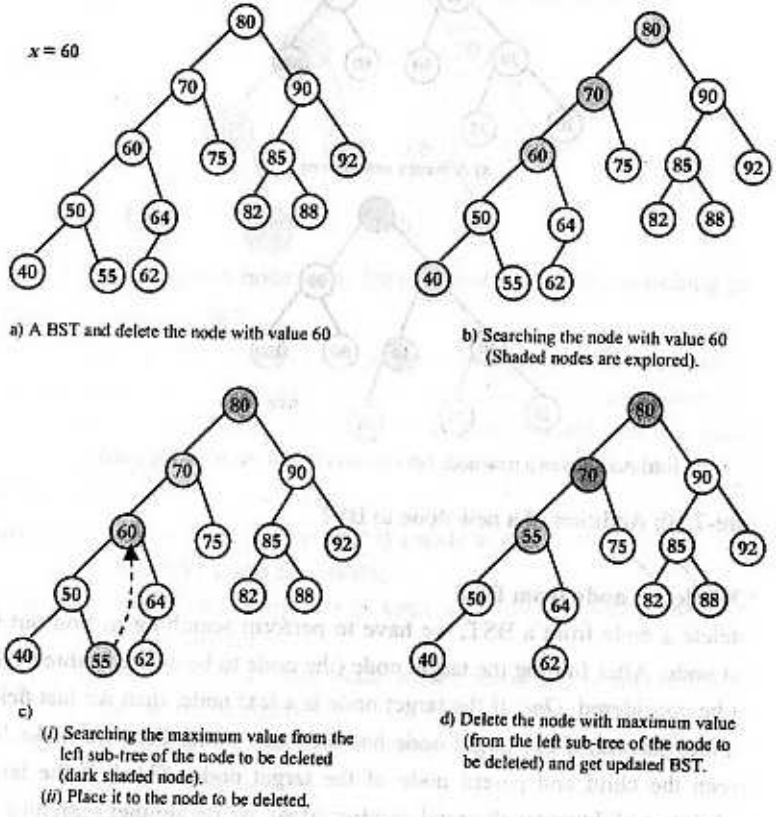
Figure-7.10: Addition of a new node to BST

7.2.3 Delete a node from BST

To delete a node from a BST, we have to perform searching to find out the target node. After finding the target node (the node to be deleted), three cases must be considered. *One*, if the target node is a leaf node, then we just delete the node. *Second*, if the target node has only one child, then we make link between the child and parent node of the target node. *Third*, if the target node has two children (with grand children also), we do another searching in

left sub-tree of the target node and find the node that has maximum value (considering the sub-tree), and mark it. Then the value of the target node is replaced by the maximum value and we delete the marked node (the node with maximum value).

In the Figure 7.11, we have shown the deletion of a node that has two children. At first we find the value of the node to be deleted (which is 60). Since it has two children, we find maximum value from the left-subtree (which is 55). Then we replace the value of the node to be deleted by 55. Lastly we delete the node of the left-subtree with maximum value.



a) A BST and delete the node with value 60

b) Searching the node with value 60 (Shaded nodes are explored).

c)
(i) Searching the maximum value from the left sub-tree of the node to be deleted (dark shaded node).
(ii) Place it to the node to be deleted.

d) Delete the node with maximum value (from the left sub-tree of the node to be deleted) and get updated BST.

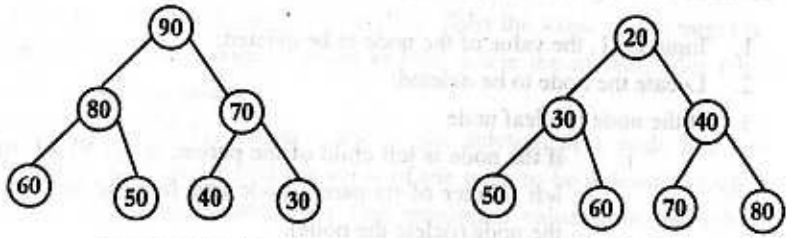
Figure-7.11: Deletion of a particular node from a BST

Algorithm 7.6: Algorithm to delete a node from a BST

1. Input BST, the value of the node to be deleted;
2. Locate the node to be deleted;
3. If the node is a leaf node:
   - i. if the node is left child of the parent, make NULL the left pointer of its parent node and free the space for the node (delete the node).
   - ii. if the node is right child of its parent, make NULL the right pointer of its parent node and free the space for the node (delete the node).
4. If the node has one child:
   - i. if the node to be deleted is a left child of its parent, then make link between the left pointer of its parent node and the child node (left or right) of the node to be deleted. Then delete the node.
   - ii. if the node to be deleted is a right child of its parent, then make link between the right pointer of its parent node and the child node (left or right) of the node to be deleted. Then delete the node.
5. If the node to be deleted has two children:
   - i. locate the node with minimum value from the right sub-tree of the node to be deleted or the node with maximum value from the left sub-tree of the node to be deleted.
   - ii. replace the node value to be deleted by the node value found in step-5(i)
   - iii. delete the node located in step-5(i)
6. Output: updated BST.

7.3 Heap

It is a complete binary tree each of whose node's value is greater (or smaller) than the node value of its children. If the node value is greater than the node value of its children, then the heap is called max heap. Otherwise, the heap is called min heap.



(a): A max heap (b): A min heap

Figure-7.12: Graphical representation of Heap

Since heap is a complete binary tree, so array implementation of heap is efficient.

7.3.1 Heap creation

Creation of a heap from a given list of elements (numbers):

We add the elements one by one and place them in respective positions. Here, we add the child at the left position first, then at the right position. Using the following steps given below, we can create a max heap:

1. Add an element as first element (root) of the heap.
2. Select a child and place it in proper position:
   i. Add the next element as left child. If the left child is already present, then add as the right child
   ii. Compare the child with its parent. If the parent value is smaller than the child's value, then place the parent's value in the child's position and the child's value in parent's position.
   iii. Repeat the step-2(ii) until we find a parent whose value is greater than the child's (which is added last) value or we go beyond the root.
3. Repeat the step-2(i) to 2(iii) for rest of the elements.
4. Get a heap as an output.

According to the above description the pictorial view of heap creation process is shown in Figure 7.13. At first we take 45 from the list and add it as root of the heap. Then we take 28 from the list and add it as left child of the root and compare it with the value of the root, since 28 < 45, so the tree is a heap. After that, we add 52 as the right child of the tree and compare it with 45 (which is at the root). Since 52 > 45, so we put 52 at the root and 45 in the place of right child (position C in Figure 7.13). In this way we can create a heap shown in position I of the Figure 7.13.

Given List: 45 28 52 25 60 70

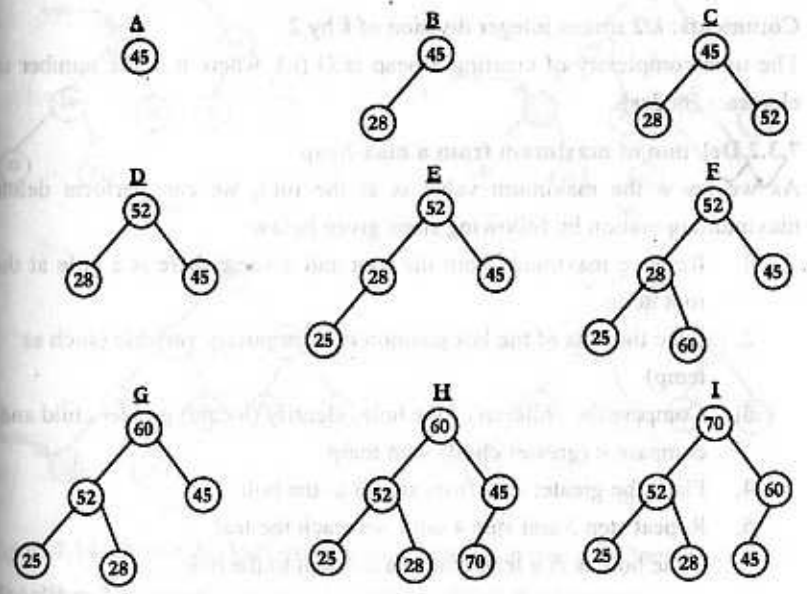


Figure-7.13: Heap creation process (pictorial view)

Algorithm 7.7: Algorithm to create a heap (pseudo code)

1. Input an array A[1...n] with data and a variable, temp
   (a list of data is in the array, A)
2. for (i =2; i ≤ n; ++i)
   {
   temp=A[i];
   k = i;

3. while (k > 0 and A[k/2] < temp)
   {
   A[k] = A[k/2];
   k = k/2;
   }
   A[k] = temp;
   }
4. Output array A [1...n] as a heap

Comments: *k*/2 means integer division of *k* by 2.
The time complexity of creating a heap is O (n), where n is the number of elements (nodes).

7.3.2 Deletion of maximum from a max-heap

As we know the maximum value is at the root, we can perform delete maximum operation by following steps given below:

1. Remove maximum from the root and assume there is a hole at the root node
2. Take the data of the last position in a temporary variable (such as temp)
3. Compare the children of the hole, identify (locate) greater child and compare it (greater child) with temp
4. Place the greater data from step-3 to the hole
5. Repeat step 3 and step 4 until we reach the leaf
6. If the hole is in a leaf, place data if tem to the hole.

Delete maximum operation is illustrated using Figure 7.14 as follows:

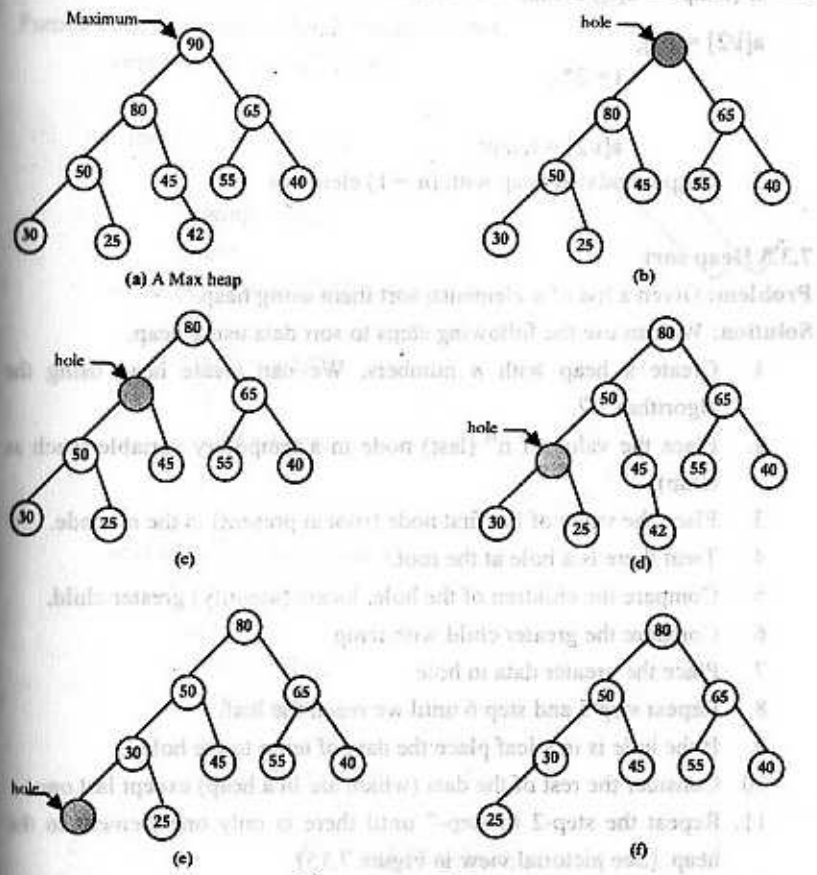


Figure-7.14: Delete the Maximum from a Max heap (pictorial view)

Algorithm 7.8: Algorithm (pseudo code) to delete the Maximum

1. Input a heap, *A*[1...*n*]. // The heap as an array
2. *i* = 2;
3. temp = a[k];
4. while (*i* <= k)
   {
   if ((*i* < k) and (a[i] < a[i+1]))
   i = i + 1;

```
if (temp >= a[i]) break;
a[i/2] = a[i];
        i = 2*i;
        }
        a[i/2] = temp;
```

5. Output updated heap with $(n-1)$ elements.

7.3.3 Heap sort

Problem: Given a list of n elements, sort them using heap.

Solution: We can use the following steps to sort data using heap.

1. Create a heap with n numbers. We can create heap using the algorithm 7.7.
2. Place the value of n^{th} (last) node in a temporary variable (such as temp).
3. Place the value of the first node (root at present) in the n^{th} node.
4. Treat there is a hole at the root.
5. Compare the children of the hole, locate (identify) greater child.
6. Compare the greater child with temp
7. Place the greater data in hole
8. Repeat step 5 and step 6 until we reach the leaf.
9. If the hole is in a leaf place the data of temp to the hole
10. Consider the rest of the data (which are in a heap) except last one.
11. Repeat the step-2 to step-7 until there is only one element in the heap. (See pictorial view in Figure 7.15)

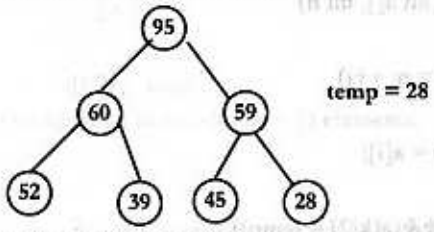
Algorithm 7.9: Heap sort algorithm.

1. Input an a[i...n] with random data;
2. heapify (a, n)
3. for (k = n; k >1, --k)
   rearange (a, k);
5. output a sorted list in a[n].

Pseudo code of the algorithm 7.9 is as follows:

```
void heapify (int a[], int n)
        {
 for (i =2; i< = n; ++i)
        {
        temp = a[i];
        k=i;
while((k > 0)&&(a[k/2] < temp))
        {
        a[k] = a[k/2];
        }
        a[k] = temp;
        }
         }
void rearange (int a[], ink k)
    {
     int temp, i;
        temp = a[k];
        a[k] = a[1];
        i=2;
        while ((i+1) < k))
        {
                if((i < k)&&(a[i] < a[i+1]))
                i=i+1;
                if(temp >= a[i]<a[i+1]))
                i=i+1;
                if(temp>=a[i]) break;
                a[i/2]=a[i];
                i=2*i;
                }
                a[i/2]=temp;
        }
```

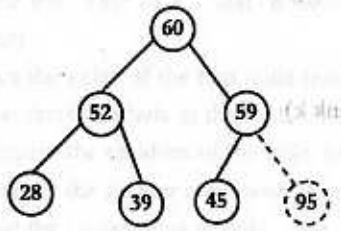
The time complexity of heapsort is O (n log n)



A Max heap with n elements

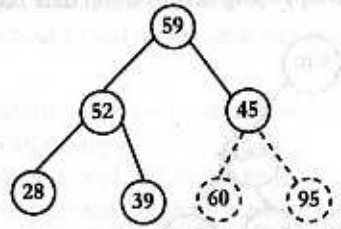
Array:	95	60	59	52	39	45	28

a) A heap and its corresponding array



Array:	60	52	59	28	39	45	95

b) First step of sorting process using heap



Array:	59	52	45	28	39	60	95

c) Second step of sorting process using heap

Array:	28	39	45	52	59	60	95

d) Sorted data

Figure-7.15: Pictorial view of heap sort process

7.3.4 Priority Queue

Priority queue is a structure or queue where elements are maintained based on their priority (priority number). In a priority queue each element is associated with a value and maintained according to its priority and often implemented as heap. We can put the element with the highest priority at the root of the heap. A priority queue allows at least two operations: Inserts and Delete minimum, or Delete maximum.

One application of priority queues is to schedule jobs on a multiuser system. The jobs are stored in or priority queue and to be performed according to their priority where a job is finished, the next job is selected based on their priority from the remaining jobs.

Let us consider the following example

Priority	Data	Abbreviation
1	Chairman	C
2	Director-1	D-1
3	Director-2	D-2
4	Chief Engineer	CE
5	Manager	M
6	Engineer	E
7	Deputy Manager	DM

We can create a priority queue (heap) using abbreviation data according to their priorities as follows:

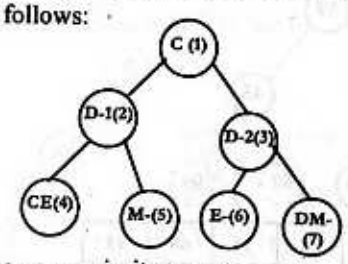


Figure-7.16: A heap as a priority queue.

The operations on priority queue are same for the heap.

Summary:

The concept of data structure tree is obtained from the concept of natural tree, except that natural tree is a bottom-up figure, whereas the data structure tree is a top-down figure. A **tree** is a collection of nodes that has a special designated node called *root node*. The root node may have one or more child nodes. When each node of a tree has at most two children, then the tree is called binary tree. If each node of a binary tree has two children (except the leaf nodes) then it is called *full binary tree*. If each level (except the deepest level) contains all possible nodes and the deepest level contains the nodes in as left as possible, then it is called *complete binary tree*.

There are three techniques of traversing a binary tree: *pre-order*, *in-order* and *post-order*. The prefix *pre*, *in* and *post* represent the order, the root node to be visited. In *pre-order* technique, first visit the root node then traverse the left subtree in pre-order and then traverse the right subtree also in pre-order. In *in-order* technique, first traverse the left subtree in in-order, then visit the root and finally traverse the right subtree in in-order. In *post-order* technique, first traverse the left subtree in post-order, then traverse the right subtree in post-order and finally visit the root node.

In a binary tree, if the values of all nodes in the left subtree (left side of the root) are smaller and the values of all nodes in the right subtree (right side of the root) are larger than the node value of the root, then it is called **binary search tree**. Binary search tree is used mainly for fast searching strategy.

If the value of each node in a complete binary tree is greater (or smaller) than the value of its children, then it is called a **heap**. For greater values of children it is called *max heap* otherwise it is called *minheap*. Heap is used mainly for sorting strategy.

The data structure tree is implemented using array and linked list.

An array implementation of a complete or full binary tree is efficient, otherwise linked implementation is efficient.

Questions:

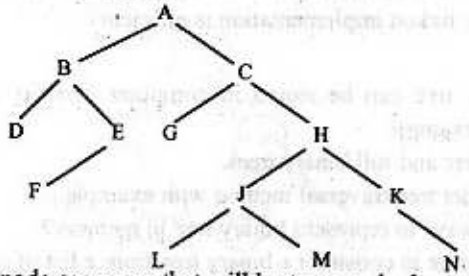
1. How a binary tree can be stored in computer memory using array ? Explain with example.
2. Define complete and full binary trees.
3. Describe inorder tree traversal method with example.
4. What are the ways to represent binary tree in memory?
5. Write a procedure to construct a binary tree from a list of input data.
6. Construct a BST with the following list of values:
   50, 15, 10, 13, 20, 22, 55, 60, 42, 57
7. Write a procedure that deletes a node from a BST having only one or no child.
8. Discuss linked representation of a binary tree.
9. What will be the depth of a complete binary tree which has 3000 nodes?
10. What will be the sequence obtained by in-order and post–order traversal.
11. Suppose a binary tree T is in memory. Write an algorithm to find the depth of T.
12. Suppose a binary tree T is in memory. Write an algorithm to delete all the terminal nodes in T.
13. Show the parent-child relationship and child-parent relationship with respect to their positions in binary tree.
14. Write an algorithm to create a heap.
15. Explain Prim's algorithm with example.
16. Write an algorithm to insert a node into a binary search tree.
17. What do you mean by minimum spanning tree?
18. What is the difference between BST and heap?
19. You are given i th node of a binary tree. Show that, the relationship of the parent of i th node and the relationship of i th node with its children. Give example.
20. Construct a heap using the following data (show each step separately)
    19, 40, 5, 17, 23, 51, 9, 29, 21, 3, 7, 24, 27.
21. Describe a heap creation process with example.
22. Write an algorithm to delete an item from a heap.
23. Suppose the following sequences list the nodes of a binary tree T in preorder and inorder respectively:

    Preorder: G, B, Q, A, C, K, F, P, D, E, R, H.

    Inorder: Q, B, K, C, F, A, G, P, E, D, H, R.

    Draw the diagram of the tree.

24. Consider the following tree



List the node sequence that will be traversed when we use (i) Inorder, (ii) Preodrer (iii) Post order traversal techniques.

25. Write an algorithm to create heap from an arbitrary list of elements.

26. Prove that the depth (height) D_n of a complete binary tree with n nodes is given by

$D_n = [\log_2 n+1]$, where $[\bullet]$ is the floor function.

27. Construct a down-heap H from the following list of numbers:

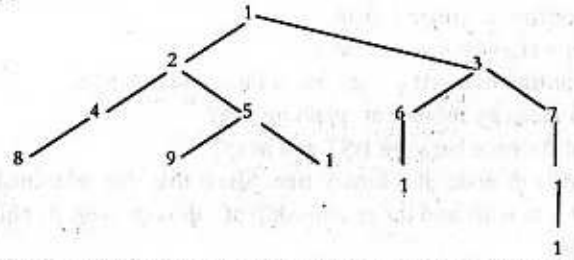
44, 30, 50, 22, 60, 2, 55, 77, 55

28. Write reheapdown function for implementing heap.

29. Draw the binary search tree whose elements are inserted in the following order:

50 72 96 107 26 12 11 9 2 10 25 51 16 17 95 51

30. Consider the following tree



List the node sequence that will be traversed when we use i) Inorder ii) Preorder iii) Postorder traversed techniques.

Problems for Practical (Lab) Class

Tree related problems

Problem 7-1: Take (store) 10 integers in an array. Consider the array as a complete binary tree. Now print:

i) all the data in a row.
ii) Print the data of any index and its children.

Print the data of any index and its parent.

Problem 7-2: Take (store) 10 integers in an array. Consider the array as a complete binary tree. Now print:

i) Print the data of leftmost path and rightmost path separately (from root to leaf).
ii) Print the data all paths (from root to the leaves).

Problem 7-3: Take (store) 10 integers in an array. Consider the array as a complete binary tree. Now print:

i) all the data using pre-order method.
ii) all the data using in-order method.
iii) all the data using post-order method.

Problem 7-4: Draw a BST with 14 integers in paper (note book). Store the data of the BST in an array. Now print:

i) all the data using in-order method.

Hints: the output will be in ascending order.

Problem 7-5: Create a linked binary search tree (BST) with 10 integers. Print the data of the BST using in-order traversal method. You cannot use any existing built in function that creates BST.

Problem 7-6: Add two additional nodes to the BST you have created for problem 7-5. Delete a node that has two children in the BST and print the data of the BST using in-order traversal method.

Problem 7-7: Create a heap (array based) with more than **seven integers.** Print data from **any index** and print its left child, right child and parent (if any). You cannot use any **existing built in function** that creates the heap.

Problem 7-8: Add two additional data to the heap you have created for the

problem 7-7. Delete the root and print the data with their indices.

Remember after addition and deletion you have to rearrange the data so that the data are in heap.
Sample output:

index	1	2	3	4	5
data	90	70	82	55	48

Problem 7-9: Given more than ten **arbitrary integers.** Sort them using heap sort algorithm. Print separately the data after heap creation phase and the sorted data. **Show** the required number of data comparisons for sorting. You cannot use any **existing built in function** that creates heap and/or sorts data.
Hints: Create a heap, swap the data and rearrange the data after swapping.

CHAPTER EIGHT
GRAPH

OBJECTIVES:

- Identify graph
- Describe how a graph can be stored in memory
- Describe graph traversal methods
- Identify minimum cost spanning tree
- Describe Prim's algorithm
- Write Prim's algorithm in algorithmic form
- Describe Kruskal's algorithm
- Write Kruskal's algorithm in algorithmic form
- Describe single source shortest paths problem
- Write an algorithm for single source shortest paths

8.1 Basics of Graph

A graph is a set of nodes (vertices) and edges. The node that holds data is called vertex and the line connecting two vertices is called edge. If G denotes a graph, $G = (V, E)$ where V denotes set of vertices and E denotes set of edges.

Undirected Graph: If each edge of a graph is undirected (without direction), the graph is called undirected graph. The undirected edge is also called unordered edge.

Directed Graph: If each edge of a graph is directed (with direction), the graph is called directed graph.

Weighted Graph: If the value (cost) of each edge is given, the graph is called weighted graph.

Path: A path is a sequence of vertices where each pair of successive vertices is connected by an edge.

Connected Graph: A graph is called connected, if there is a path between each pair of vertices.

Cycle: A cycle is a path where first and last vertices are the same.

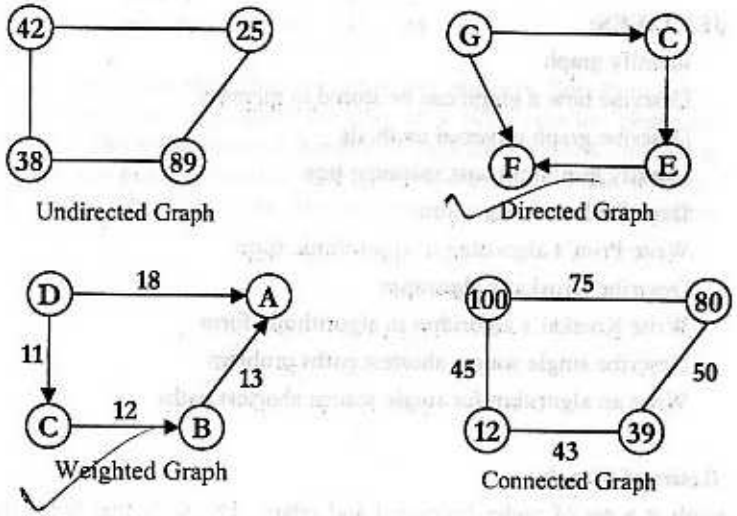
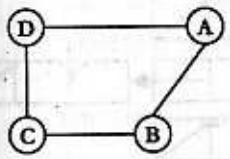


Figure-8.1: Different types of graphs

Graphs may be directed or undirected. In directed graphs, directions of edges are given.

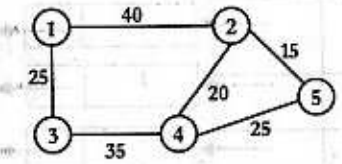
A graph is stored in the computer's memory using the matrix (two dimensional array). The matrix used to store graph in computer's memory is called adjacency matrix. Two vertices are adjacent to each other if there is an edge between them. If a vertex is adjacent to other, we put 1 in matrix on their cross point. If there is no edge, we put 0 in the matrix. That means A_{ij} = 1, if there is an edge between the vertices i and j. Otherwise, $A_{ij} = 0$, where A denotes adjacency matrix.

In case of weighted graphs, the value of edge is to be put in the adjacency matrix.



	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

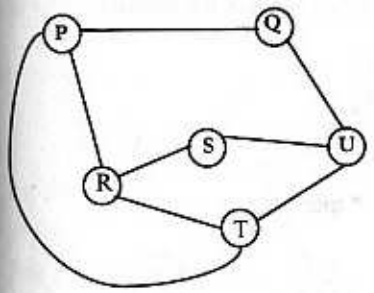
(a): Adjacency matrix of an undirected graph



	1	2	3	4	5
1	0	40	25	0	0
2	40	0	0	20	15
3	25	0	0	35	0
4	0	20	35	0	25
5	0	15	0	25	0

(b): Adjacency matrix of a weighted graph

Figure-8.2: Adjacency matrix of directed and undirected graph



Vertices	List of adjacent vertices		
P	Q	R	T
Q	P	U	
R	P	S	T
S	U	R	
T	P	R	U
U	Q	S	T

Figure-8.3: An undirected graph and its adjacency list

Graph can be stored (represented) as adjacency lists in computer's memory. Since adjacency lists are variable in length, so linked list implementation will be efficient for this type of lists. Linked representation of the Figure 8.3 is shown in Figure 8.4.

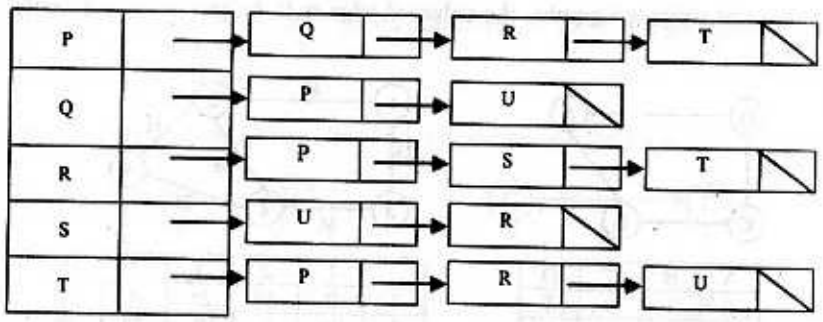


Figure-8.4: Linked representation of the adjacency lists

We can easily implement the above linked list using C/C++.

Algorithm 8.1: Algorithm to create adjacency list of a graph

1. Input a graph (information of graph) and take a variable, item
2. Declare vertex (node) and table of vertices

```
i) struct vertex
   {
    char data;
    node * next;
   }
ii) vertex table [1.....m], * nptr, * tptr;
```

3. Set i = 1
4. Create a table of vertices:

```
table [i].data = item;
table [i].next = NULL;
```

5. Create a vertex (node) with value:

```
nptr = new (vertex);
nptr → data = item;
nptr → next = NULL;
```


6. if (table[i].next = NULL), then, table[i].next = nptr;

```
else {
tptr = table[i]. next
while (tptr → next ! = NULL)
{
tptr = tptr → next;
}
tptr → next = nptr;
```

7. Set i = i + 1;
8. Repeat step- 4 to 7 for necessary times
9. Output a linked adjacency lists.

8.2 Graph Traversal (Search) Methods

There are two principal methods for graph traversal, which are as follows:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

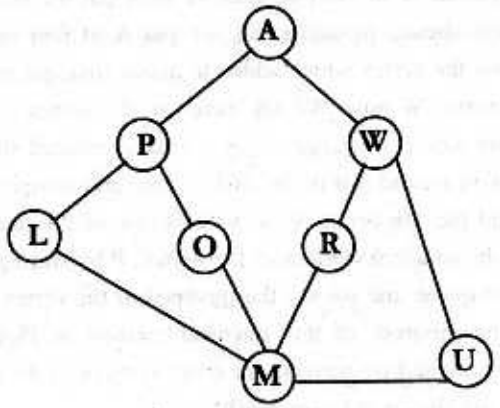


Figure-8.5: An undirected graph

8.2.1 Breadth First Search (BFS)

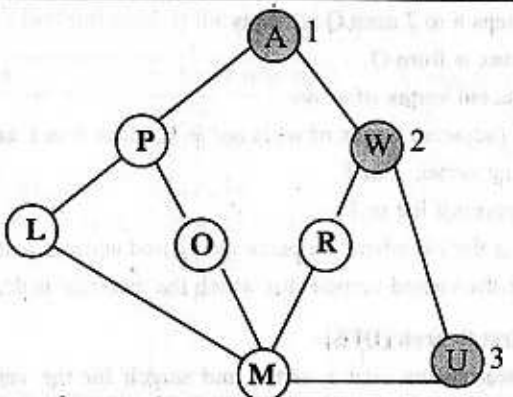
In breadth-first search, we visit a vertex first and search for the vertices that are adjacent to this vertex. In later steps we visit every adjacent vertex (adjacent to the first vertex) one by one and search for their adjacent vertices. We continue this process until we visit every node in the graph.

Here, one important issue is that we need to store all the information of the adjacent vertices (say, value of the vertices) that will be visited later after visiting the current vertex. Therefore, we have to store the values of the vertices to be visited lately in a **queue** so that we can track which vertex is to be visited next. To traverse the vertices properly, we make a table (Table-8.1). In first column of the table we have enlisted the vertex and its adjacent vertices.

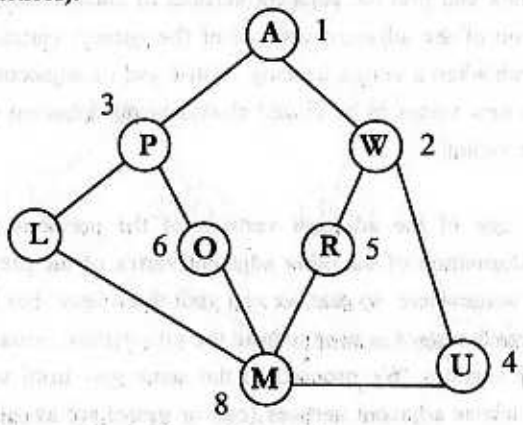
In the second column of the table we have shown the vertices stored in queue. According to the Figure 8.5, we have visited the vertex A first and we store two adjacent vertices W and P in queue. So, we have enlisted the vertices A, W, P in first row of first column, and shown the vertices W, P in first row of second column. In third column we have put the vertex, whose adjacent vertices are already in queue. So, we put A in first row of third column. As we know the vertex which added to queue first, get service first, so we traverse the vertex W now. We say traversal of a vertex is done if its adjacent vertices are stored in queue. Thus we have enlisted the adjacent vertices of vertex W in second row of the first column and stored the vertices (U, R) in queue and put the vertex W in second row of the third column. After that we visit the adjacent vertices of the vertex, P by storing them (the vertices U and R) in queue and we say the traversal of the vertex P is done. We have shown the progress of this traversal method in Figure 8.6(a). Following the above method we traverse the other vertices of the graph. The traversal sequences are shown in Figure 8.6(b).

Table-8.1: Breadth first traversal of the graph in Figure-8.5

Vertices visited	Vertices in queue	Vertex traversal done
A, W, P	W, P	A
A, W, P, U, R	P, U, R	W
A, W, P, U, R, O, L	U, R, O, L	P
A, W, P, U, R, O, L, M	R, O, L, M	U
	O, L, M	R
	L, M	O
	M	L
		M



(a) The progress of traversal sequences are shown by marking 1, 2, 3 (the shaded vertices).



(b) Traversal sequences are shown by marking 1, 2, 3, ..., 7.

Figure-8.6: Pictorial view of BFS

Algorithm 8.2: Algorithm for breadth first search

1. Input a graph, G
2. Create two empty lists L and T
3. Create an empty queue Q
4. For each vertex of G do
   i) Take a vertex *v* from G
   ii) Place the vertex *v* and its adjacent vertices in L
   iii) Add the adjacent vertices of the vertex *v* to Q
   iv) Place the vertex *v* in T
5. Repeat the steps 6 to 7 until Q is empty.
6. Access a vertex *w* from Q
7. For each adjacent vertex of *w*, do
   i) If it (adjacent vertex of *w*) is not in L, place it in L and add it to Q
   ii) Place vertex *w* in T
8. Output the traversal list in T

Comments: L is the list where we place the visited vertices and T is the list where we place the visited vertices for which the traversal is done.

8.2.2 Depth First Search (DFS)

In depth-first search, we visit a vertex and search for the vertices that are adjacent to the starting vertex. Then, we visit one of the adjacent vertices of the starting vertex and find the adjacent vertices of current vertex. Now, we have to visit one of the adjacent vertices of the current vertex. That is, in depth-first search when a vertex is being visited and its adjacent vertices are found, the very new vertex to be visited always be the adjacent vertex of the previous vertex visited.

While visiting one of the adjacent vertices of the previous vertex, it is obvious that information of the other adjacent vertex of the previous vertex must be stored somewhere, so that we can visit them later. For this purpose in depth-first search a *stack* is used to hold the information (usually *values* of vertices) of the vertices. We proceed in the same way until we reach the deepest vertex whose adjacent vertices (one or more) are already in the list (stack) or already visited. In this way, all vertices in the graph are visited.
To traverse the graph in Figure 8.5 we make a table.

In first column of the Table-8.2, we have enlisted the first vertex and its adjacent vertices. The adjacent vertices of vertex *A* are also stored in a *stack* (second column of the table). Since the adjacent vertex of *A* (vertices *P* and *W*) are stored in *stack*, the vertex traversal is done, which is marked in the third column of the table. So, we say traversal of a vertex is done, when its adjacent vertices are stored in the *stack*. Since we store the vertex, W in stack last, we accessed it first. That means, we traverse the vertex W by storing its adjacent vertices, (vertices R and U) to the stack. In this way we traverse other vertices of the graph.

Table-8.2: Depth first traversal of the graph in Figure-8.5

Vertices visited	Vertices in stack	Vertex traversal done
A, P, W	P,W	A
A, P, W, R, U	P, R, U	W
A, P, W, R, U, M	P, R, M	U
A, P, W, R, U, M, L, O, R	P, R, L, O	M
	P, R, L	O
	P, R	L
	P	R
		P

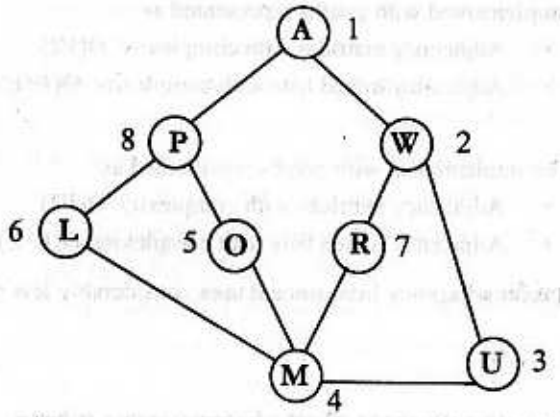


Figure-8.7: Pictorial view of depth first traversal method (traversal sequences are marked by 1, 2, 3, ..., 8)

Algorithm 8.3: Algorithm for depth first search

1. Input a graph, G
2. Create two empty lists L and T
3. Create an empty stack S
4. For each vertex of G do
   i) Take a vertex *v* from G
   ii) Place the vertex *v* and its adjacent vertices in L
   iii) Add the adjacent vertices of the vertex *v* to S
   iv) Place the vertex *v* in T
5. Repeat the steps 6 to 7 until S is empty.
6. Access a vertex *w* from S
7. For each adjacent vertex of *w* do
   i) If it (adjacent vertex of *w*) is not in L, place it in L and add it to S
   ii) Place vertex *w* in T
8. Output the traversal list in T

Comments: L is the list where we place the visited vertices and T is the list where we place the visited vertices for which the traversal is done.

8.2.3 Implementation of DFS & BFS using different data structure of graph.

DFS:

DFS can be implemented with graphs represented as:

- Adjacency matrices with complexity $\Theta(V2)$
- Adjacency linked lists with complexity $\Theta(V+E)$

BFS:

BFS has can be implemented with graphs represented as:

- Adjacency matrices with complexity $\Theta(V2)$
- Adjacency linked lists with complexity $\Theta(V+E)$

Usually, we prefer adjacency lists, since it uses considerably less memory when,

$|E| << |V^2|$.

There also exist some situations where adjacency matrix is better:

- Graph is "Dense" i.e., $|E| \approx |V^2|$.
- Often need to check whether an edge exists from *u* to *v*

8.3 Minimum cost spanning tree

Spanning subgraph: A subgraph of a graph, which contains all the vertices of the graph.

Spanning tree: A spanning tree is a subgraph (spanning subgraph) of a graph, which contains all the vertices of the graph and contains no cycle.

Minimum cost spanning tree: It is a spanning tree whose cost is minimum.

We can build a spanning tree from a undirected connected graph, In a spanning tree the total number of nodes (vertices) will be same as the total number of vertices in the graph. However, the total number of edges will be one less than the total number of nodes (vertices). In minimum cost spanning tree the total edge cost will be minimum. In other words summation of the cost of the edges of the tree must be minimum. There are two well known algorithms to build minimum cost spanning tree. One is Prim's algorithm and another is Kruskal's algorithm.

8.3.1 Prim's Algorithm

It is a method to build a minimum cost spanning tree edge by edge. Since in spanning tree total number of nodes is same as the total number of vertices in the graph, so at first we take the nodes (vertices) and at each stage, we select a vertex and an edge. When we choose a vertex and select an edge from the vertex that has minimum cost among all the edges from the (chosen) vertex. When we choose first vertex (the root of the tree), we consider the adjacent vertices of the vertex associated costs. We shall select the edge that has minimum cost. Then we choose another vertex and its adjacent vertices with cost. We shall select the edge with minimum cost again. To select an edge, we have to check two things. One is, when we select an edge, we should check whether any of vertices of this edge make another edge whose cost is less than the first edge. Second is, we should not select an edge more than once and the selected edge must not make a cycle. We consider the graph of Figure 8.8 and show the stages of Prim's algorithm 8.9.When we choose the cost vertices, we have to check the adjacent vertices from this vertex and select the edge with minimum cost.

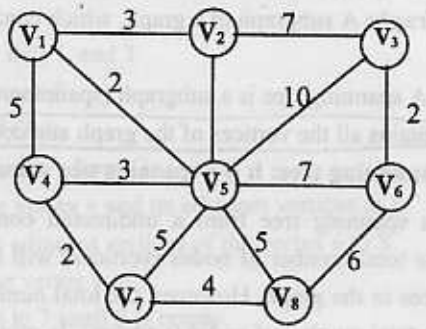
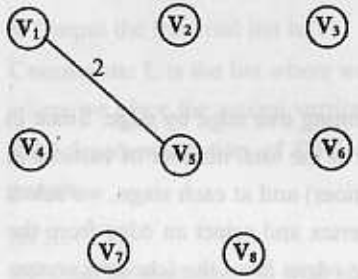
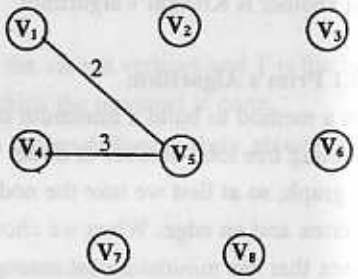


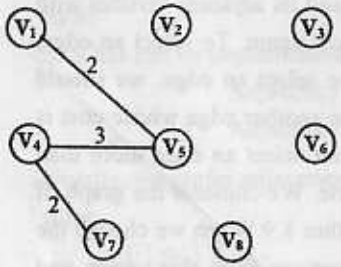
Figure-8.8: A weighted graph



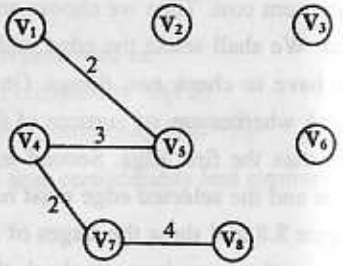
(a) Selection of first edge



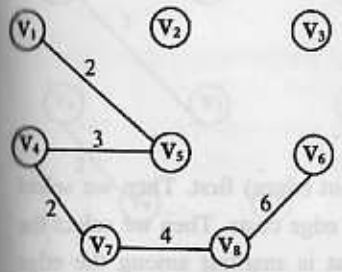
(b) Selection of second edge



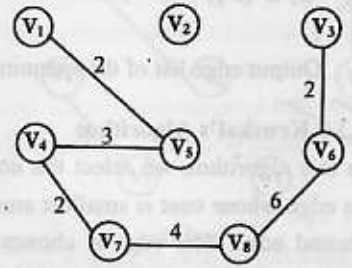
(c) Selection of third edge



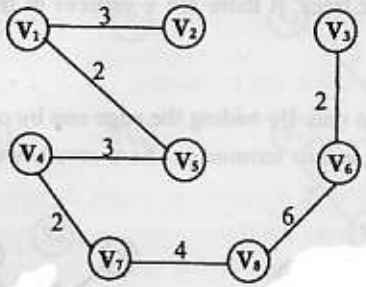
(d) Selection of fourth edge
Here we reject the edge (V_7 ,V_5), since it makes a cycle



(e) Selection of fifth edge We cannot select the edge (V_8 ,V_4), it has been already selected and we reject (V_8 ,V_5), since it makes a cycle



(f) Selection of sixth edge



(g) Selection of seventh edge

Figure-8.9: Pictorial view of stages Prim's algorithm

Algorithm 8.4: Prim's Algorithm

1. Input a weighted connected graph, G = (V, E). V is the set of vertices and E is the set of edges
2. Create an empty list E_T
3. Create a list V_T which will initially contain the starting vertex, *i.e* $V_T = \{ v_0 \}$
4. for each vertex i of G do

{

find a minimum-weight edge e' = (v' , u') among all edges (v, u) such that v is in V_T and u is in V - V_T.

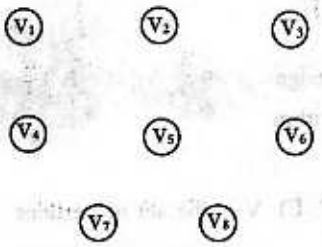
$V_T = V_T \cup \{u'\}$;
$E_T = E_T \cup \{e'\}$;
}

5. Output edge list of the spanning tree E_T.

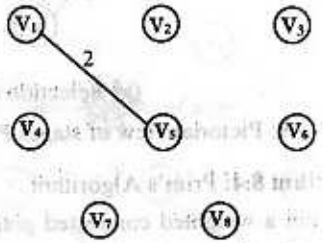
8.3.2 Kruskal's Algorithm

In this algorithm, we select the nodes (without edges) first. Then we select an edge whose cost is smallest among all the edge costs. Then we select the second edge. The edge is chosen whose cost is smallest among the edge costs except the first selected edge. We continue the process in this way. When we choose a vertex with edge it should not make a cycle. In Kruskal algorithm, initially it treats all the single nodes as trees. That means initially, there are v single node trees, if there are v vertices in the graph. When we add an edge,

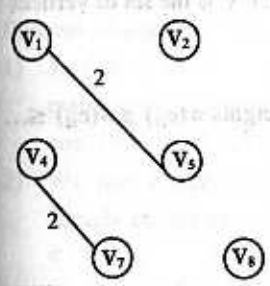
we merge two trees into one. By adding the edge one by one, finally we get a single tree and the algorithm terminates. As a result, we get the minimum spanning tree.



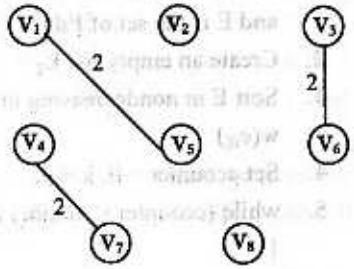
(a) Step 1 (Select all the nodes)



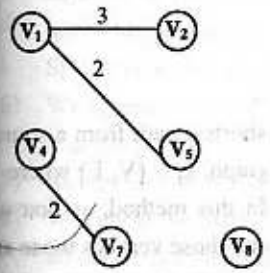
(b) Step 2 (Select an edge with minimum cost (minimum among all the edges))



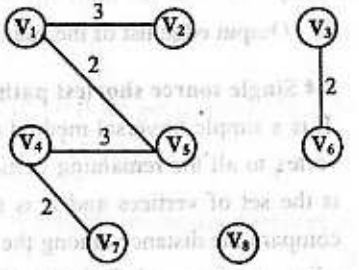
(c) Step 3



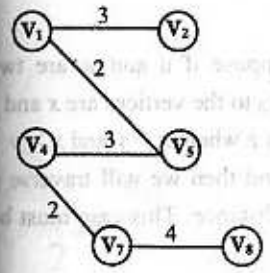
(d) Step 4



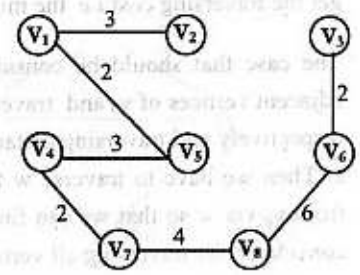
(e) Step 5



(f) Step 6



(g) Step 7



(h) Step 8

Figure-8.10: Pictorial view of the stages of the Kruskal's algorithm.

Algorithm 8.5: Kruskal's Algorithm

1. Input a weighted connected graph, G = (V, E). V is the set of vertices and E is the set of Edges
2. Create an empty list E_T
3. Sort E in nondecreasing order of the edge weights $w(e_{i1}) \leq w(e_{i2}) \leq \ldots \leq w(e_{iE})$
4. Set ecounter = 0; k = 0;
5. while (ecounter < number of vertices - 1)

{

k = k + 1;

if (E_T U { e_{ik} } is acyclic)

$E_T = E_T$ U { e_{ik} };

ecounter = ecounter +1 ;

}

6. Output edge list of the spanning tree E_T.

8.4 Single source shortest paths problem

It is a simple traversal method that finds out the shortest path from a source vertex to all the remaining vertices of a directed graph, G = (V, E) where V is the set of vertices and E is the set of edges. In this method, at first we compare the distance among the source vertex v_0 and those vertices those are adjacent of v_0 and find out that vertex whose distance is minimum from v_0.We repeat this process until we traverse the entire vertex in the graph and get the traversing cost *i.e.* the minimum distance.

The case that should be considered here is, suppose if u and w are two adjacent vertices of v_0 and traversing cost from v_0 to the vertices are x and y respectively and traversing distance from w to u is z where $x > y$ and $x > y + z$. Then we have to traverse w from v_0 at first and then we will traverse u from v_0 via w so that we can find the minimum distance. This case must be considered for traversing all vertices.

We can use the single source shortest path algorithm as described below to find out all minimum distances by traversing the graph.

1) At first, we create an adjacency matrix cost[1:n][1:n]. It will be a n* n matrix in which cost[i][j] = ∞ if there is no edge between v_i and v_j, and cost[i][j] = x if the distance from v_i to $v_j = x$.
2) We use a status array say S[1:n], to keep track which vertices are already considered, for all vertex in the graph G. Initially, S[i] = false, 0 < i <=n.
3) We assign S[v_0] = true where v_0 is the source vertex.
4) We take an array dist[1:n] for all vertex n in the graph G , and assign dist[i] = cost [v][i] , 0 < i <= n. and assign dist[v_0] = 0.
5) Next, we choose a vertex v_i = u, from those vertices not in S such as v_i is adjacent to the selected vertex and dist [u] from the selected vertex is minimum and assign S [u] = true. For each vj = w adjacent to u with S[w] = false we will repeat Step 6 and 7.
6) We compare dist[u] with dis [u] + cost[u][w] if we get positive result i.e if dist[w] > dist[u] + cost[u][w] then we go to Step 7.
7) Assign dist[w] = dist[u] + cost[u][w].
8) We repeat the Steps 5 to Step 7 untill we find the minimum distances of all the vertices from the source v_0.

Example:

Let the given graph be

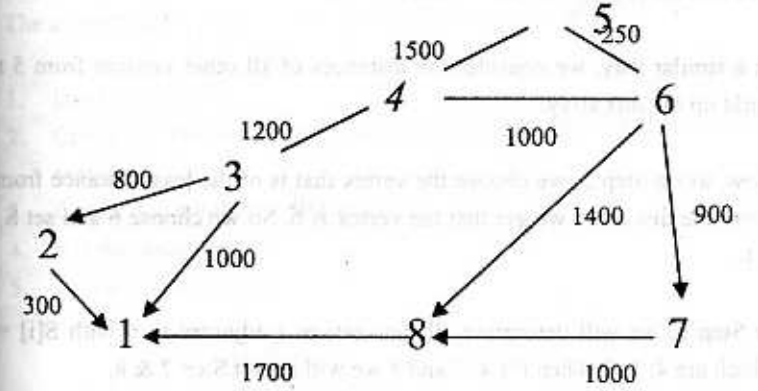


Figure-8.11: The given graph

By using step- 1 we will get the following adjacency matrix

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8								0

Figure-8.12: Adjacency matrix

Here 5 is the source vertex .So S [5] = 1 and dist [5] = 0 from Step 3 and Step 4

At first we check whether there is an edge from 5 to 1. From the adjacency matrix, cost [5] [1] = ∞, so dist [1] = ∞.

Next we check whether there is an edge from 5 to 2. From the adjacency matrix, cost [5] [2] = ∞, so dist [2] = ∞.

Then, we check whether there is an edge from 5 to 3. From the adjacency matrix, cost [5] [3] = ∞, so dist [3] = ∞.

In a similar way, we consider the distances of all other vertices from 5 and build up the dist array.

Now, using Step 5, we choose the vertex that is of the least distance from 5. From the dist array we get that the vertex is 6. So we choose 6 and set S [6] = 1.

In Step 6, we will determine all the vertices i adjacent to 6 with S[i] = 0, which are 4, 7, 8. Then for 4, 7 and 8 we will repeat Step 7 & 8.

For vertex 4 first we compare, in Step 7 & 8, whether dist [4] > dist [6] + cost [6] [4]. Here, dist[4]= 1500 and dist [6] + cost [6] [4] = 250 + 1000= 1250. Here, we have found an alternative path from 5 to 4 which has a shorter distance than the direct one and we update the dist as, dist [4] = dist [6] + cost [6][4]

We repeat the process for 7 and 8 for other vertices. Following the above process we get our desired output. In the following diagram the total process is displayed.

S	Vertex selected	Distance							
		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
----------	----------	#	#	#	1500	0	250	#	#
{5}	6	#	#	#	1250	0	250	1150	1650
{5,6}	7	#	#	#	1250	0	250	1150	1650
{5,6,7}	4	#	#	2450	1250	0	250	1150	1650
{5,6,7,4}	8	3350	#	2450	1250	0	250	1150	1650
{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
{5,6,7,4,8,3,2}		3350	3250	2450	1250	0	250	1150	1650

Figure-8.13: Action of single source shortest paths

Algorithm 8.6: Algorithm for single source shortest paths

The algorithm of single source shortest path is given below:

1. Input a Graph, G
2. Create the cost adjacency matrix COST[n][n] where n is the number of vertices
3. Create two empty list **dist** and S
4. v is the source vertex
5. for each vertex i in G

```
{
        S[i] = false;
dist[i] = COST[v][i];
}
```

6. S[v] = true; dist[v] = 0;
7. for (num =2 to n-1)do
   {
   Choose u from among those vertices not in S such that dist[u] is minimum;

```
S[u] = true;  //Put u in S
for (each w adjacent to u with S[w] = false) !
{
  if (dist[w] > dist[u] + cost[u][w])then
dist[w] = dist[u] + cost[u][w];
 }
  }
```

8. Output the shortest path dist of all vertices from the single source v.

Summary:

Graph is a set of nodes and edges. The node that contains data is called *vertex* and the line connecting two vertices is called *edge*. According to the direction presented in an edge, there are two types of graph: *Directed graph* and *Undirected graph*. When each edge of the graph is assigned a value, then it is called *weighted graph*. In a graph, if there exists a path between each pair of vertices, then it is called *connected graph*.

There are two principal methods of traversing a graph: **Breadth First search (BFS)** and **Depth First Search (DFS)**. *Breadth first search* is a simple strategy in which a node is visited first, then all the successors of the node are visited next, then their successors and so on. In general, all the nodes are visited at a given depth before any nodes at the next level are visited. *Depth first search* always visits the deepest node that means the search proceeds immediately to the deepest level of the graph, where the nodes have no successors. In this way, all vertices in the graph are visited.

From an undirected connected graph, we can build a spanning tree. *A* **spanning tree** is a subgraph that has no cycle. The spanning tree of total minimum cost is called *minimum cost spanning* tree. To build minimum cost spanning tree, the graph must be *weighted graph*. There are two well known algorithms to build minimum cost spanning tree: Prim's and Kruskal's algorithms. *Prim's algorithm* is a method to build a minimum cost spanning tree edge by edge. At first all the vertices are taken and at each next stages, we select a vertex and an edge from that vertex whose cost is minimum among all the edges from the (chosen) vertex. Then we choose another vertex and proceeds similarly. In *Kruskal's algorithm*, first an edge whose cost is smallest among all the edge costs is selected, then we select the second smallest cost assigned edge and next the third, fourth and so on. In both Prim's and Kruskal's algorithm in each stage we check so that it does not make a cycle.

The data structure graph can be implemented using aray and linked list. The difference between *tree* and *graph* is that tree contains no cycle whereas graph may contain cycle. That means all trees are graph but all graphs are not trees.

Questions:

1. Define graph with example.
2. How a graph can be stored in computer's memory ? Explain with example.
3. Define directed, weighted and unconnected graphs. Give examples.
4. All trees are graphs, but all graphs are not trees. Explain this statement example.
5. How many ways a graph can be traversed ? Discuss in details about breadth first traversal techniques.
6. Define adjacent list and adjacent matrix of a graph.

7. Define a graph. The daily flights of an airline company appear in the following figure. CITY lists the cities, and ORG[k] and DEST[k] denote the cities of origin and destination, respectively, of the fight NUMBER[k], (i) Draw the corresponding directed graph of the data, (ii) and give the adjacency structure of the graph.

1	Jessore
2	Chittagong
3	Sylhet
4	Rajshahi
5	Dhaka

	NUMBER	ORG	DEST
1	701	2	3
2	702	3	2
3	705	5	3
4	708	3	4
5	711	2	5
6	712	5	2
7	713	5	1
8	715	1	4
	717	5	4
	718	4	5

8. What is the difference between a tree and a graph ?
9. What is the difference between breadth first search and depth first search? Show with example.
10. Describe the stages of Prim's algorithm with example.
11. Describe the stages of Kruskal's algorithm with example.

Problems for Practical (Lab) Class

Graph related problems

Problem 8-1: Given a graph in Fig. 8-1, create an adjacency matrix for the graph. **Print** the adjacency matrix.

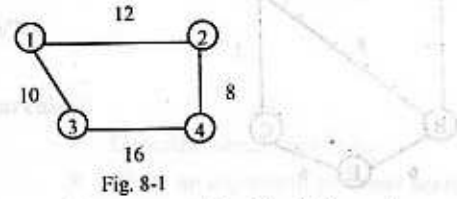


Fig. 8-1

Problem 8-2: Given a graph in Fig. 8-2, create an adjacency matrix for the graph. **Print:**

i) the adjacency matrix ii) the list of vertices and adjacency list with edge cost of each vertex.

Hints for ii)

Input: Enter vertex: **A** (from keyboard)
Output: B 10 D 12
Input: Enter vertex: **B**
Output: C 7 A 10
And so on.

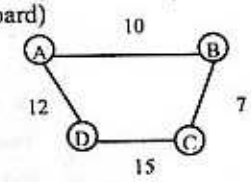


Fig. 8-2

Problem 8-3: Given a graph in Fig. 8-3 and create linked adjacency lists with edge cost of each edge and **print** the data of the linked list.

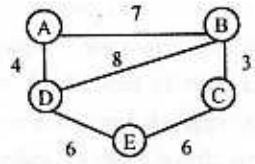


Fig. 8-3

Problem 8-4: Given a graph in Fig. 8-4 and create adjacency matrix of the graph and **detect the cycles** of the graph.

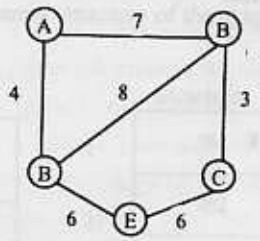


Fig. 8-4

CHAPTER NINE
SEARCHING AND SORTING

OBJECTIVES:

- **Searching:**
  - Describe linear searching
  - Write an algorithm of linear searching
  - Describe binary searching
  - Write an algorithm of binary searching
- **Sorting:**
  - Describe the process of selection sort
  - Write an algorithm of selection sort
  - Describe the process of insertion sort
  - Write an algorithm of insertion sort
  - Describe the process of merge sort
  - Write an algorithm of merge sort
  - Describe the process of quick sort
  - Write an algorithm of quick sort

9.1 Searching

Searching means to find out or locate any element from a given list of elements. In this chapter we have described methods (algorithms) of searching and sorting. At the end of each algorithm we have analyzed the algorithm, *i.e.*, we have estimated the approximate execution time and space required for the algorithm. In other words we have shown the complexity of each algorithm. To identify or locate an element or position of the element from a list of elements is called *searching*. Usually, there are two types of searching

i. Linear Searching and
ii. Binary Searching

9.1.1 Linear Searching

In this method we search the target element one by one until we find the element or we go beyond the list.

Suppose that we are given a list of elements and a target element. We have to determine whether the target element exists in the list or not and when it is found, the location of the element will be output. In linear searching, at first we take an element from the list and compare it with the target element. If the first element is the target element, then we have found the element and the searching is successful and completed. On the other hand, if the first element is not the target element, then we take second element from the list and compare it with the target element. If at this stage the second element is the target element, then the searching is successful and completed. If the second element is also not the target element then repeat the process as for the first element and the second element. Thus we shall repeat the process until we find the target element in the list or we go beyond the list.

Suppose that we are given a list of numbers as follows:

17 12 18 5 7 8 10

We have to find out whether a number, 7 is in the list or not.

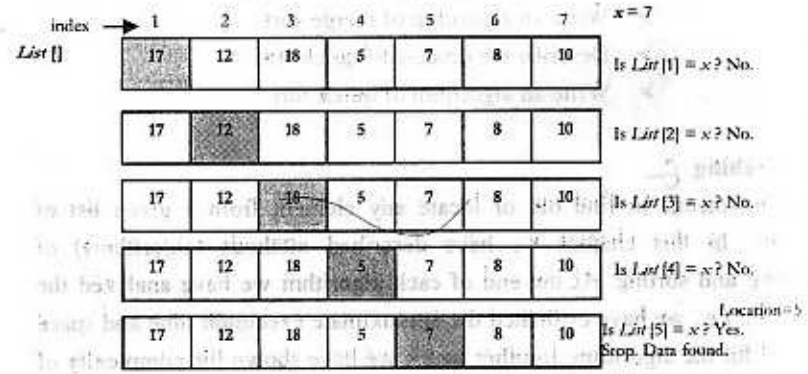


Figure-9.1: Pictorial view of linear searching (shaded items showed the searching sequence)

Algorithm 9.1: Algorithm for linear searching

1. Input a list and an item (element to be found)

```
A [1...n];
item = x;
location = 0
```

2. Search the list to find the target element

```
for (i = 1; i ≤ n; i = i + 1)
    {
    if (A[i] = item)
        {
        print "Found ";
        location = i
        Stop searching;
        }
    }
```

3. if ($i > n$) print "Not Found ";
4. Output : "Found" or "Not Found".

9.1.1.1 Complexity for linear searching

In average case, time complexity $= \dfrac{1+2+3+\ldots\ldots\ldots\ldots+n}{n}$

$$= \frac{n(n+1)}{2n}$$

$$= \frac{1}{2}n + \frac{1}{2}$$

$$= O(n)$$

Therefore, the time complexity = $O(n)$

9.1.2 Binary Searching

Precondition (prerequisite) for binary searching is that the elements must be arranged either in ascending or in descending order.

Problem: Given a list of data elements arranged in ascending or descending order, locate (find the position of) a particular (target) element from the list.

9.1.2.1 Process

Suppose that we are given a list of elements arranged in ascending or descending order (which is the prerequisite of binary searching) and a target element. In binary searching, at first we have to identify the middle element of the list and compare this middle element with the target element. If middle element is the target element, then the searching is successful and terminated. On the other hand, if the target element is smaller than the middle element (for ascending data), we have to perform searching on left half of the list in the same manner which was done for the whole list. We repeat the process until the target element will be found or the (searching) process will be completed. This process can be stated with points as follows.

1. In binary searching process, at first the middle element of the list is identified.

   Middle index = (1^{st} index + Last index) / 2

2. After that, the target element is compared with the middle element of the list.
   i. if the middle element is the target element, then the process is completed and terminated.
   ii. otherwise, if the target element is smaller than the middle element, we have to search the target element in the left half of the list in the same manner which is done for the whole list.
   iii. on the other hand, if the target element is greater than the middle element, the target element has to be searched in the right half of the list in the same manner which is done for the whole list.

We can explain the binary searching process using symbols as follows.
Let the list of the elements are —

$a_1, a_2, a_3, a_4, \ldots \ldots \ldots \ldots \ldots, a_m$; and x is the target element.

Then we shall follow the steps given below for binary searching:

1. Compute middle index, $mid = (first + last) / 2$
2. If $x = a_{mid}$, then the element has been found.

3. If $x < a_{mid}$, then we have to search the target element in $a_1, a_2, a_3, \ldots \ldots \ldots, a_{mid-1}$ and we will search the target element using Step-1 and Step-2, where we shall use $(mid - 1)$ instead of *last*.
4. If $x > a_{mid}$, then we have to search the target element in $a_{mid+1}, a_{mid+2}, a_{mid+3}, \ldots \ldots, a_m$ and we shall use the Step-1 and Step-2; where we will use $(mid + 1)$ instead of *first*.
5. Repeat the process until we find the target element x, or we go beyond the list.

Algorithm 9.2: Algorithm for Binary Searching (pseudocode)

1. Input $A[1 \ldots m]$, x; *//A* is an array with size m and x is the target element
2. *first* = 1, *last* = m;
3. while (*first* ≤*last*)

```
{
mid = (first + last) / 2;
(i) if (x = A[mid]), then print mid; //  target element = A[mid] or target
    break (stop searching);              //element is in index mid.
(ii) else if (x < A[mid]) then last = mid - 1;
(iii) else first = mid +1;
}
```

4. if(first > last) , print "not found";
5. Output: mid or "not found"

Example:

Suppose that we are given a list of numbers as follows:

17 19 28 30 45 55 58 61 63 67 72 76 80 89

and we have to find out 19 from the list.

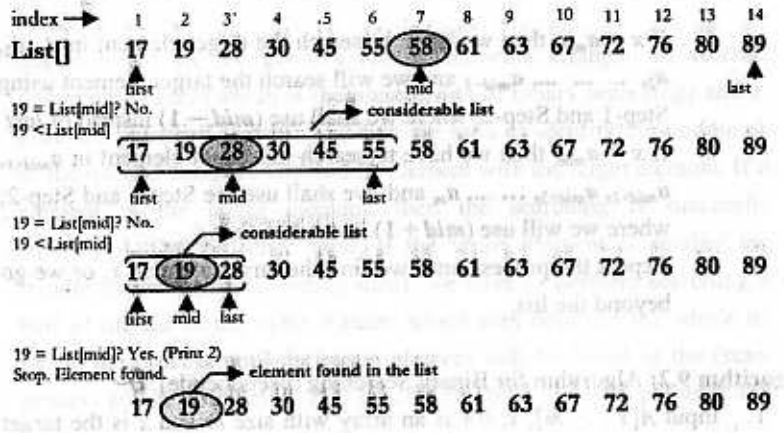


Figure-9.2: Pictorial view of binary searching

9.1.2.2 Number of comparison to search the target element

Suppose, there are n elements in the list.

Let $n = 2^k$; then $k = log_2 n$.

Therefore, Time complexity = $O(log_2 n)$

9.2 Sorting

To arrange a list of elements (data) in ascending or descending order is called sorting. There are two types of sorting –

i. Internal Sorting and
ii. External Sorting

9.2.1 Internal sorting

The method (algorithm) which sorts a list that is small enough to fit entirely in primary (internal) memory, is called internal sorting.

9.2.2 External sorting

The method (algorithm) which sorts a list (file) that can not fit entirely in primary memory, that means to sort the entire list the method uses external memory, is called external sorting.

9.2.3 Classes of internal sorting

Exchange Sort: Selection sort, Insertion sort, Bubble sort

Divide and Conquer Sort: Merge sort, Quick sort

Tree Sort: Heap sort, Tournament sort

Non - comparison based Sort: Radix sort, Bucket sort etc.

9.2.4 Selection sort

In this method, at first we select (find) the smallest data of the list. After selecting, we place the smallest data in first position and the data in first position is placed in the position where the smallest data was. After that we consider the list except the data in the first position. Again we select the (second) smallest data from the list and place it in the second position of the list and place the data in the second position, in the position where the second smallest data was. By repeating the process, we can sort the whole list. Using steps we can write the method as follows.

i. Given a list of data. Find out the smallest data from the list. Place the smallest in first position and the data of the first position in the position of the smallest data.
ii. Repeat the process for the list except data in first position, and so on.

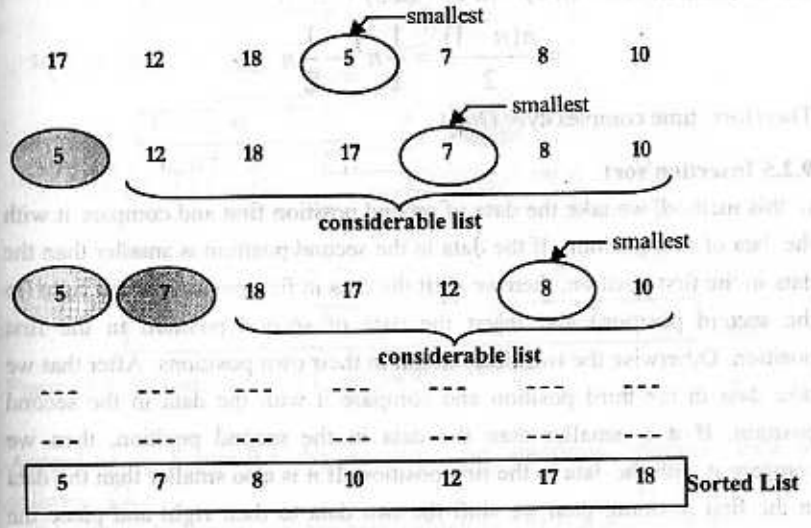


Figure-9.3: Pictorial view of selection sort

Algorithm 9.3: Algorithm for selection sort

1. Input Array $A[1.....n]$
2. (i). for ($i = 1$ to $n - 1$)

{

```
        small_index = i;
(ii). for (j = i + 1 to n)
                {
                if (A[j] < A[small_index]) then small_index = j;
                }    // end of second for
temp = A[i];
A[i] ← A[small_index];
A[small_index] ← temp;
}                    // end of first for
```

3. Output: sorted list.

Comments: A is an array of size n, where the data are stored and sorted later.

9.2.4.1 Complexity of selection sort

For first phase, number of comparison is n-1; second phase, number of comparison is n-2; third phase, number of comparison is n-3 and so on. Then,

No. of comparisons = (n-1) + (n-2) + (n-3) + ... + 1

$$= \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Therefore, time complexity = $O(n^2)$

9.2.5 Insertion sort

In this method, we take the data of second position first and compare it with the data of first position. If the data in the second position is smaller than the data in the first position, then we shift the data in first position to the right (to the second position) and insert the data of second position in the first position. Otherwise the two data remain in their own positions. After that we take data in the third position and compare it with the data in the second position. If it is smaller than the data in the second position, then we compare it with the data in the first position. If it is also smaller than the data in the first position, then we shift the two data to their right and place the data of the third position in the first position. In case if the data of the third position is smaller than the data of the second position and greater than the data of the first position, we shift the data of the second position to its right and the third data of the third position is inserted in the second position. By repeating the process for the data in the forth position, fifth position and so

on, we can sort the whole list. Using steps we can describe the process as follows.

i. Given a list of elements (data).

ii. We have to insert a data into its correct position by moving all data (before it) to the right (that are greater than the data which is being considered at this moment).

iii. By repeating Step-*ii* for all considerable data we can arrange the whole list in ascending order.

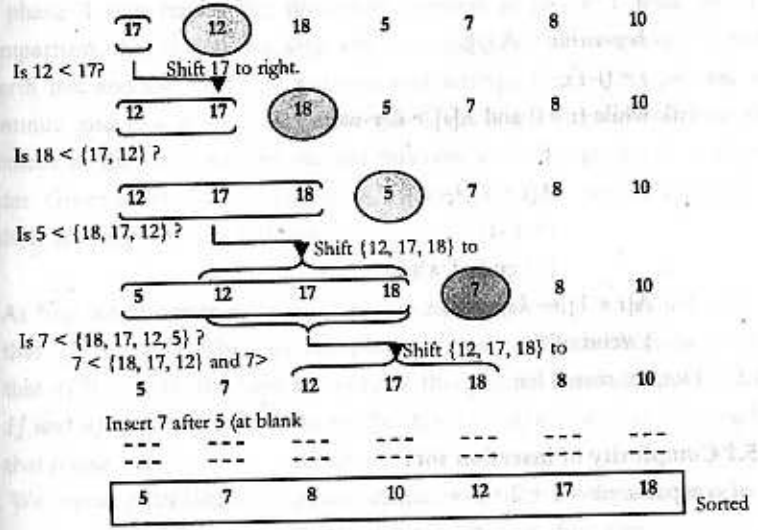


Figure-9.4: Pictorial view of Insertion Sort

According to Figure-9.4 the insertion sort algorithm rearrange data as follows:

i. We select item number two (which is 12) and compare it with the item number one (which is 17) and check 12 < 17; the answer is yes, then 17 will be shifted right and 12 will be put in the place of 17 (first position).

ii. Next we select the item number three (which is 18 now). Check 18 < 17; the answer is no, so the list will remain as it is.

iii. Next we select the item number four (which is 5) and compare it with the item number three (which is 18), check 5 < 18; the answer is yes. Again we check 5 < 17, the answer is yes. Again we check 5 < 12, the answer is yes. So, the items 12, 17, 18 will be shifted right and 5 will be place in first position, which is free after shifting.

iv. By repeating the process we get the sorted list.

Algorithm 9.4: Algorithm for insertion sorting

1. Input Array A[1.....*n*]
2. (i). for (*j* = 2 to *n*)
   {
   key-value = A [*j*];
   i = (*j*-1);
   (ii). while (*i* > 0 and A[*i*] > *key-value*)
   {
   A[*i* + 1] ← A[*i*];
   i = *i* -1;
   } // end of while
   A[*i* + 1] ← *key-value*;
   } //end of for
3. Output: sorted list.

9.2.5.1 Complexity of insertion sort

No. of comparisons = 1 + 2 + 3 + ... + (n-1)

$$= \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Therefore, time complexity = $O(n^2)$

9.2.6 Bubble sort

Bubble sort is a simple sorting algorithm. In this method, we at first compare the data element in the first position with the data element in the second position and arrange them in desired order. Then we compare the second data element with the third data element and arrange them in desired order. The same process continues until we compare the data element at the second last and last position. This is the first phase (phase-1) of this method. At the next phase, we repeat the procedure applied at phase 1 with one less comparison, that is now we stop after we compare the data element at third last and the second last position and arrange them in desired order. Similarly, at phase 3, we repeat the procedure applied at phase 1 with one less comparison, that is now we stop after we compare the data element at the fourth last and the third last position and arrange them. This process will continue and end at the last phase when we will compare only the data element at the first and the second position and arrange them in desired order. Given a list of data elements *A[1], A[2],..., A[n]*, we can describe the bubble sort algorithm as follows:

i. At first we compare *A[1]* and *A[2]* and arrange them in desired order so that *A[1] < A[2]*. Then we compare *A[2]* and *A[3]* and arrange them so that *A[2] < A[3]*. We have to continue this process until we compare *A[n-1]* and *A[n]* and arrange them so that *A[n-1] < A[n]*. (we can observe here that phase 1 requires *n-1* comparisons)

ii. We repeat phase 1 with one less comparison; that is now we stop after we compare and rearrange *A[n-2]* & *A[n-1]*. (so this step requires *n-2* comparisons)

iii. We repeat phase 1 with one less comparison than phase 2; that is now we stop after we compare and rearrange *A[n-3]* & *A[n-2]*. (so this step requires *n-3* comparisons)

iv. At step n-1 we compare only *A[1]* with *A[2]* and arrange them in desired order so that *A[1] < A[2]*.

After n-1 phases the list will be sorted in increasing order.

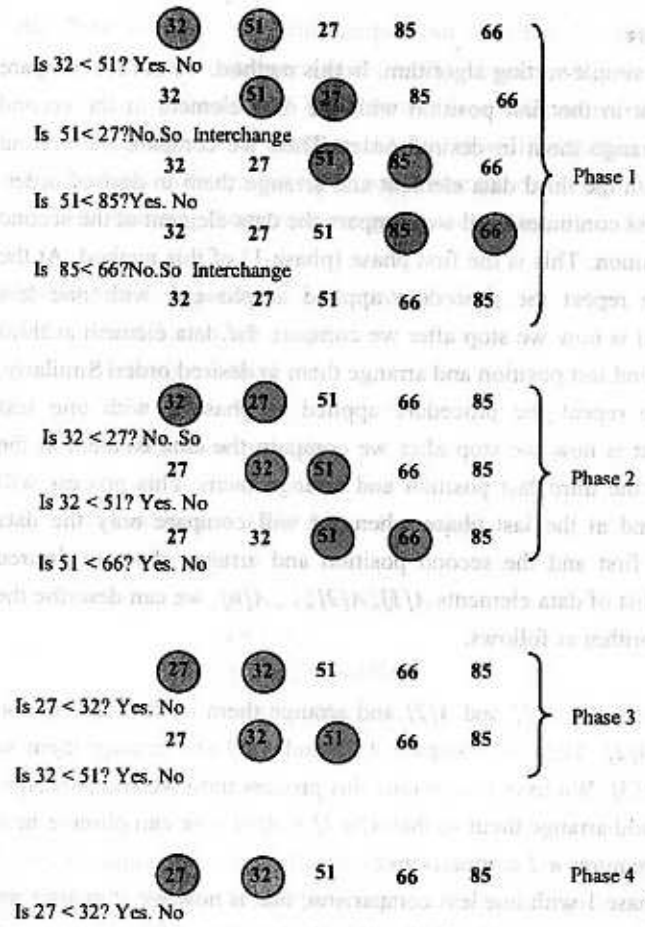


Figure-9.5: Pictorial view of bubble sort

According to the Figure 9.5 the bubble sort algorithm rearranges data as follows:

i. At phase 1, we select the first data 32 and the second data 51 and compare them whether 32<51, the answer is yes, so no interchange is needed. Then, we select the second data 51 and the third data 27 and compare them whether 51<27, the answer is no, so interchange is made. Similarly, we select the data 51 and 85 and compare them whether 51<85, the answer is yes, so no interchange is needed. Then we select the data 85 and 66 and compare them whether 85<66, the answer is no, so interchange is made.

ii. At phase 2, we select the first data 32 and the second data 27 and compare them whether 32<27, the answer is no, so interchange is made. Then, we select the second data 32 and the third data 51 and compare them whether 32<51, the answer is yes, so no interchange is made. Similarly, we select the data 51 and 66 and compare them whether 51<66, the answer is yes, so no interchange is needed.

iii. At phase 3, we select the first data 27 and the second data 32 and compare them whether 27<32, the answer is yes, so no interchange is made. Then, we select the second data 32 and the third data 51 and compare them whether 32<51, the answer is yes, so no interchange is made.

iv. At phase 4, we select the first data 27 and the second data 32 and compare them whether 27<32, the answer is yes, so no interchange is made.

Algorithm 9.5: Algorithm for bubble sort

1. Input: Data [1....n]
2. for k=1 to n do

```
{
    Set st =1;
    while (st ≤n-k)
    {
        if (Data[st] > Data[st+1])
            Interchange (Data[st] and Data[st+1]);
        Set st = st+1;
    }
}
```

3. Output: sorted list

9.2.6.1 Complexity of bubble sort

No. of comparisons = $(n-1) + (n-2) + \ldots + 2+1$

$$= \frac{n(n-1)}{2}$$

$$= \frac{n^2}{2} + O(n)$$

$$= O(n^2).$$

Therefore, time complexity = $O(n^2)$.

9.2.7 Merge Sort

Merge sort is a divide and conquer method. It works by dividing the list into parts, sorting the parts and merging them together.

Merge sort is a recursive procedure (function). If a function calls itself repeatedly, then the function is called recursive function. In merge sort, we divide the list into two parts. After getting two lists (parts), we divide each list into two parts again. We will repeat the division of the lists until we get a single element in each list.

After that the merging phase will start. That means we shall merge each two lists into one. By repeating the merging process, at last we will get a single sorted list.

A disadvantage of merge sort is that it requires extra spaces (array) for merging.

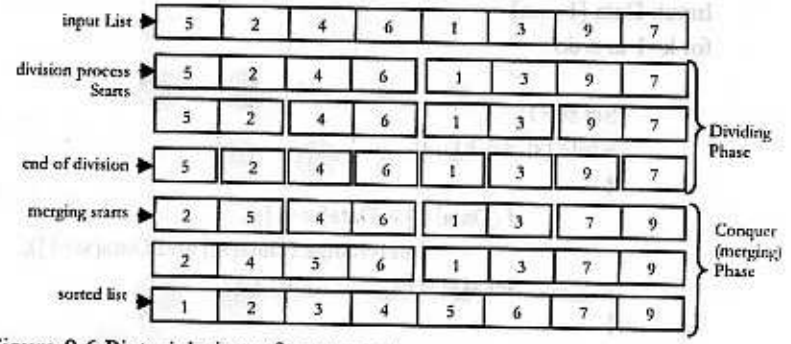


Figure-9.6:Pictorial view of merge sort

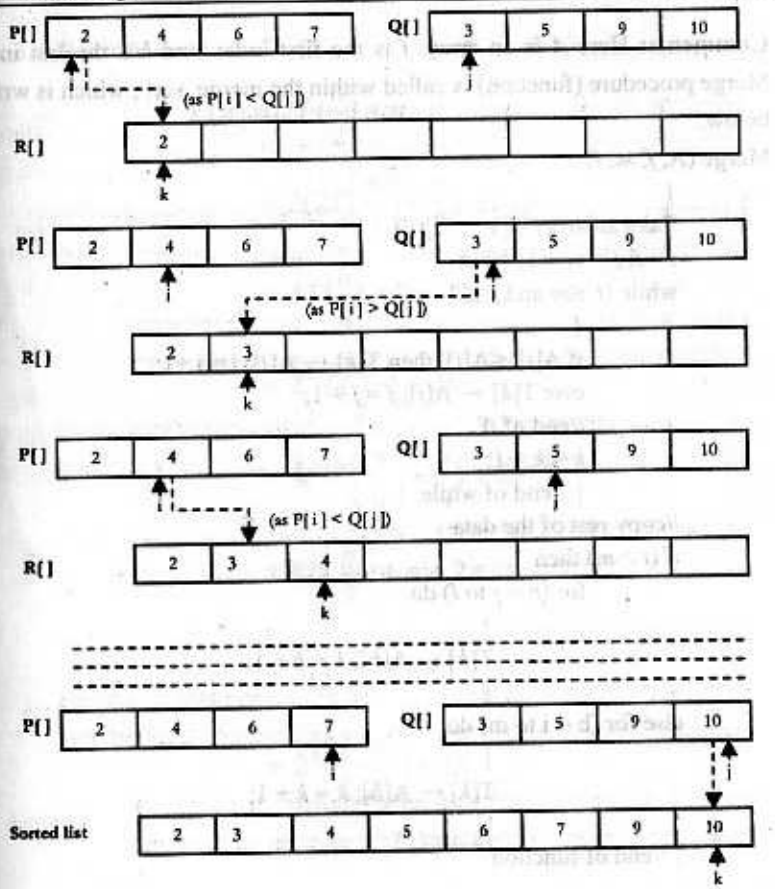


Figure-9.7: Pictorial view of merging process

Algorithm 9.6: Algorithm for merge sort

```
merge_sort (A, f, l)
{
if (f < l)
        {
        m = (f + l) / 2;
        merge_sort (A, f, m);
        merge_sort (A, m + 1, l);
        Merge (A, f, m, l);
        } // end of if
}
```

Comments: Here A is an array, f is the first index and l is the last index. Merge procedure (function) is called within the merge_sort , which is written below.

```
Merge (A, f, m, l)
    {
    Take an array T[1 . . . l];
    i = f; j = m + 1; k = f;
    while (i ≤m and j ≤l)
        {
        if A[i] ≤A[j], then T[k] ← A[i]; i = i +1;
        else T[k] ← A[j]; j = j + 1;
        //end of if
        k = k + 1;
        } //end of while
    //copy rest of the data
    if (i > m) then
        for (b = j to l) do
            {
            T[k] ← A[b]; k = k + 1;
            }
    else for (b = i to m) do
            {
            T[k] ← A[b]; k = k + 1;
            }
    } //end of function
```

9.2.7.1 Analysis of merge sort

If the time for the merging operation is proportional to n then the computing time for *merge sort* is described by the recurrence relation

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

When n is a power of 2, $n = 2^k$; we can solve this equation by successive substitutions.

For $n > 1$ we can write

$$T(n) = 2T(\frac{n}{2}) + n$$

$$= 2(2T\left(\frac{n}{4}\right) + \frac{n}{2}) + n$$

$$= 4T(\frac{n}{4}) + 2n$$

$$= 2^2 T(\frac{n}{2^2}) + 2n$$

$$= 4(2T\left(\frac{n}{8}\right) + \frac{n}{4}) + 2n$$

$$= 8T\left(\frac{n}{8}\right) + n + 2n$$

$$= 8T\left(\frac{n}{8}\right) + 3n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$= .... = 2^k T(1) + kn$$

$$[2^k T\left(\frac{n}{2^k}\right) = T(1)] and \quad n = 2^k]$$

$$= an + n\log_2 n$$

$$[T(1) = a, \quad 2^k = n, \quad k = \log_2 n]$$

$$T(n) = O(n\log_2 n).$$

9.2.8 Quick Sort

Quick sort is a divide and conquer method. In this method one element is to be chosen as partitioning element. We divide the whole list of data into two parts with respect to the partitioning element. The data which are smaller than or equal to the partitioning element, remain in the first sub-list or first part and the data which are greater than the partitioning element remain in the second sub-list (second part). If we find any data (in the first part) which is greater than the partitioning value that will be transferred to the second part. In the second part, if we find any data which is smaller than the partitioning element, that will be transferred to the first part. Transferring of data have been done by exchanging the positions of the data found in first part and second part. By repeating the process, we can sort the whole list of data. See pictorial view in Figure 9.8.

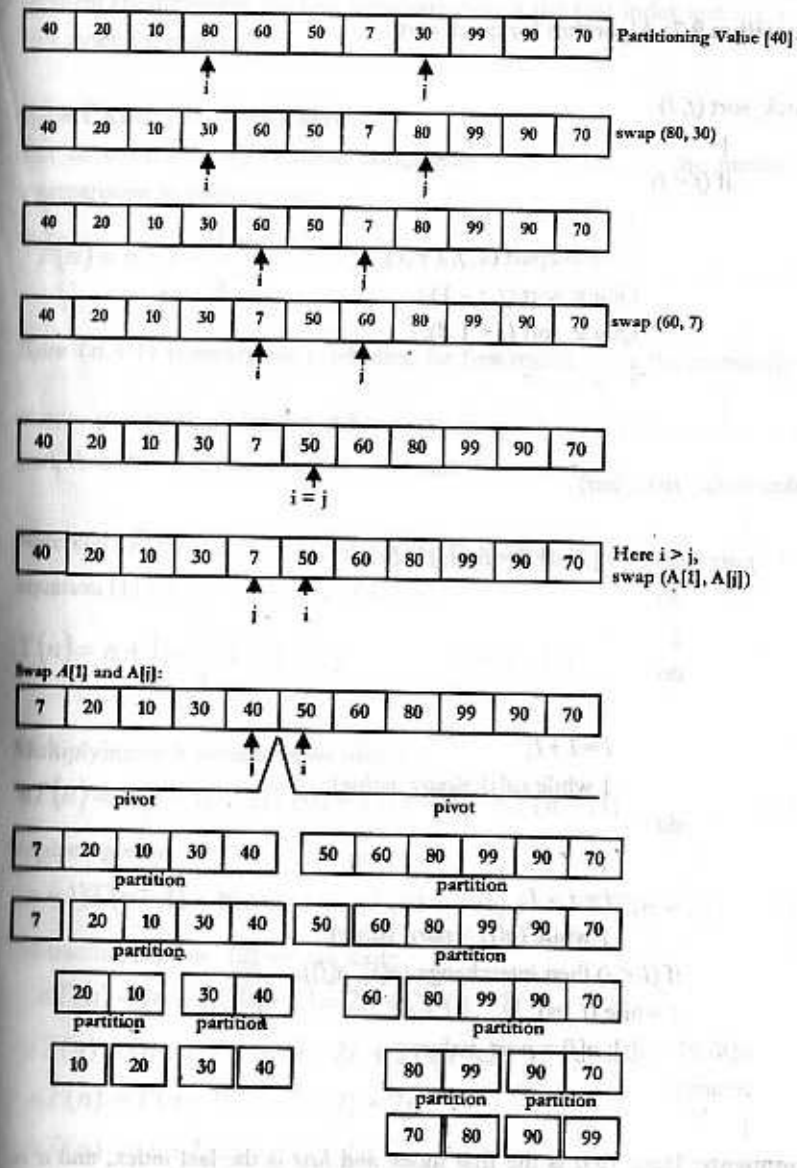


Figure-9.8: Pictorial view of quicksort partitioning

Algorithm 9.7: Algorithm for quick sort

```
Quick_sort (f, l)
        {
        if (f < l)
                {
                j = makepart (a, f, l + 1) ;
                Quick_sort (f, j – 1) ;
                Quick_sort (j + 1, l);
                }
        }

makepart (a, first, last)
      {
     part_value = a[first]; i = first; j = last;
            do
            {
            do
                    {
                    i = i + 1;
                    } while (a[i] ≤part_value);
            do
                    {
                    j = j - 1;
                    } while (a[i] > part_value);
            if (i < j) then interchange (a[i], a[j]);
            } while (i ≥j);
     a[first] = a[j]; a[j] = part_value;
     return j;
     }
```

Comments: Here, *first* is the first index and *last* is the last index; and *a* is the array which holds the list of data. And *part_value* is the partitioning element (first element for first iteration), *first* is the first index and *last* is the last index for each part.

9.2.8.1 Analysis of quick sort

Let us consider $T(n)$ be time complexity with respect to the number of comparisons in average case.

$$T(n) = n+1+\frac{1}{n}\sum_{i=1}^{n}\{T(i-1)+T(n-i)\} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Here $(n+1)$ comparisons is required for first round, $\frac{1}{n}$ is the probability to choose partitioning element. After partitioning, if $i-1$ elements are in one part, then $n-i$ elements are in other part.

Note that, $T(0) = T(1) = 0$. By putting the value of $i = 1, 2, 3, \ldots\ldots, n$, the equation (1) can be rewritten as follows:

$$T(n) = n+1+\frac{2}{n}\{T(0)+T(1)+\ldots+T(n-1)\}$$

Multiplying both sides by n we obtain

$$nT(n) = n(n+1)+2\{T(0)+T(1)+\ldots+T(n-1)\} \ldots\ldots\ldots\ldots (2)$$

Replacing n by $n-1$ in (2) we get

$$(n-1)T(n-1) = n(n-1)+2\{T(0)+T(1)+\ldots+T(n-2)\} \ldots (3)$$

Subtracting (3) from (2) we can write

$$nT(n)-(n-1)T(n-1) = 2n+2T(n-1)$$

$$\Rightarrow nT(n) = (n-1)T(n-1)+2n+2T(n-1)$$

$$\Rightarrow nT(n) = T(n-1)\{n-1+2\}+2n$$

$$\Rightarrow nT(n) = (n+1)T(n-1)+2n$$

$$\Rightarrow \frac{T(n)}{n+1} = \frac{T(n-1)}{n}+\frac{2}{n+1}$$ [dividing by $n(n+1)$]

Repeatedly using this to substitute for $T(n-2)$, $T(n-3)$, . . . we get

$$\frac{T(n)}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(n-4)}{n-3} + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\vdots$$

$$= \frac{T\{n-(n-1)\}}{\{n-(n-2)\}} + \frac{2}{\{n-(n-3)\}} + \frac{2}{\{n-(n-4)\}} + \ldots$$

$$+ \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(1)}{2} + \frac{2}{3} + \frac{2}{4} + \ldots + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right)$$

$$= \frac{T(1)}{2} + 2\sum_{x=3}^{n+1} \frac{1}{x}$$

$$= 2\sum_{x=3}^{n+1} \frac{1}{x}$$

Here $\sum_{x=3}^{n+1} \frac{1}{x} \le \int_{2}^{n+1} \frac{1}{x} dx = \log_e(n+1) - \log_e 2$

Therefore, $\frac{T(n)}{n+1} \le 2[\log_e(n+1) - \log_e 2]$

$$T(n) \le 2(n+1)[\log_e(n+1) - \log_e 2]$$

$$T(n) \le 2n \log_e n + \ldots$$

$$T(n) = O(n \log_e n)$$

That means, the quicksort algorithm takes $O(n \log_e n)$ time in average case. On the other hand the worst case time is $O(n^2)$.

Remember that merge sort takes $O(n \log_2 n)$ time in average and worst case, whereas quick sort takes $O(n \log_e n)$ time in average case and $O(n^2)$ time in worst case.

9.3 External Sorting

External sorting is required when the number of records (data) to be stored is larger than the computer can hold in its internal (main) memory. Now-a-days, to sort extremely large data is becoming more and more important for large corporations, banks and government institutions. External sorting is quite different from internal sorting, even though the problem in both cases is to sort a given list of data into increasing or decreasing order. The most common external sorting algorithm used is still the *merge sort*.

In external sorting method, at first the sorted runs (sorted sub-files) are produced, and then the sorted runs are merged to produce single run which gives us a sorted file (list of data). The runs can be produced using any internal sorting algorithm like quick sort.

Let us consider that we have to sort three thousand records R_1, R_2, ……, R_{3000} and each record is 20 words long. It is assumed that only one thousand of the records will fit in the internal memory of our computer at a time. Now this data can be sorted in the following ways. Suppose the runs are written in different files on a disk. According to our example there will be three runs and let there are three files as follows

file-1: $R_1, R_2, R_3, \ldots\ldots, R_{1000}$

file-2: $R_{1001}, R_{1002}, \ldots\ldots, R_{2000}$

file-3: $R_{2001}, R_{2002}, \ldots\ldots, R_{3000}$

Now, we shall produce a run by merging the file-1 and file-2 and these will write in file-4. Again we merge the file-3 and file-4 and produce a single run which may be written to file-1.

file-4: $R_1, R_2, \ldots\ldots, R_{2000}$
file-3: $R_{2001}, \ldots\ldots, R_{3000}$
file-1: $R_1, R_2, \ldots\ldots, R_{3000}$

During merging phase we shall read some records suppose 500 records from file-1 and 500 records from the file-2 and merge them. Merging file will be written to file-4. Again we shall read the last 500 records from the file-1 and 500 records from file-2 and merge them. The merging records will be written to file-4. Similarly we merge the file-4 and the file-3.

Summary:

To identify or locate an element or position of an element from a list of elements is called **searching**. Usually there are two types of searching: *Linear* and *Binary*. Linear searching does not require the elements to be in sorted order, whereas the binary searching must requires it (sorted order).

To arrange a list of elements in either ascending or descending order is called **sorting**. There are two types of sorting according to the requirement of memory: *Internal* and *External*. *Internal sorting* sorts the list that is small enough to fit entirely in primary (internal) memory whereas *external sorting* sorts the list that is not small enough to fit entirely in primary memory, that means it uses external memory to sort the entire list. There are various types of sorting methods: Selection, Insertion, Bubble, Merge, Quick, Heap, Tournament, Radix, Bucket Sort etc.

Questions:

1. Compare linear search and binary search.
2. Explain the binary search method with example.
3. Show that the complexity of binary search algorithm is $[\log_2 n]$, where n is the number of elements.
4. Prove that the average case complexity of linear search algorithm is

$$f(n) = \frac{n+1}{2}$$

5. Write an algorithm to find out a number from a list of given numbers.
6. Write binary search algorithm and explain it with example.
7. What is sorting?
8. Explain insertion sorting method with example.
9. Show that the complexity of merge sort is $O(n\log_2 n)$.
10. Write an algorithm for quick sort.
11. Sort the following numbers in ascending order using selection sort algorithm. Show each step.
    28, 53, 32, 84, 46, 92, 14, 63
12. Find out the complexity of selection sort algorithm.
13. Write down the algorithm of merge sort and show its complexity.
14. Write an algorithm for insertion sort and show its complexity is $O(n^2)$.
15. Describe the process of quick sort with example. Show its complexity.
16. Write an algorithm for selection sort.
17. Why is it necessary to have the auxiliary array in algorithm merge ?
18. Write the algorithm for bubble sort. Show its complexity.
19. Show with example how data are partitioned in quick sort. Show the complexity of quick sort in average case.
20. Define internal and external sorting.
21. Show the steps on selection sort to sort the following data in descending order.
    (77, 33, 44, 12, 88, 22, 66 & 95)
22. What are the basic differences between quick sort and merge sort ?
23. Show the steps in bubble sort algorithm for the following data:
    32, 51, 27, 85, 66, 23, 13, 57
24. What is external sorting? Explain with example.

Problems for Practical (Lab) Class

Sorting and searching related problems

Problem 9-1: write a program to sort some random data using **selection sort** algorithm. Hints: generate random data and take them in an array and sort them. Condition: you **cannot use** any existing built in function for sorting algorithm.

Problem 9-2: write a program to sort some random data using **insertion sort** algorithm. Hints: generate random data and take them in an array and sort them. Condition: you **cannot use** any existing built in function for sorting algorithm.

Problem 9-3: write a program to sort some random data using **heap sort** algorithm. You cannot use any existing built in function that creates heap and/or sorts data.

Problem 9-4: Write a program to sort some random data using **merge sort** algorithm. You cannot use any existing built in function to merge or sort data.

Problem 9-5: write a program to sort some random data using **quick sort** algorithm. You cannot use any existing built in function to sort data.

Problem 9-6: Can you show (print) the element (data) comparisons and time comparisons of all sorting algorithms for same sets of data?

Problem 9-7: write a program to search (find out) any integer out of 10 integers stored in an array. Use linear searching.

Hints: store 10 integers in an **array** and use linear search to search any desired number. The output will be the **number** and its **position** (index). See algorithm for linear searching.

Problem 9-8: write a program to **search** (find out) any integer out of 16 integers arranged in ascending order and stored in an array. Use binary search here.

Hints: store 16 integers in an **array** in ascending order and search any desired number using **binary search**. The output will be the **number** and its **position** (index). See algorithm for binary search.

Problem 9-9: write a program for the following:

1) Generate 100 random integers and sorts them using **insertion sort**.
2) Use **binary search** to locate any desired number.

Hints: output will be the **sorted list** and the **position** (index) of the number to be searched.

Problem 9-10: Write a program to sort same sets of data using *selection sort*, *insertion sort*, *heap sort*, *merge sort* and *quick sort*. Print the number of data comparisons and execution time of each sorting algorithm separately.

Hints: write a program that will generate **10000** random numbers and sorts them using the above mentioned sorting algorithms. The output will be sorted data, number of **data comparisons** and **execution time** of each algorithm. Repeat the process for **100000** numbers and more.

CHAPTER TEN
HASHING

OBJECTIVES:

- Identify hashing, hash table and hash function
- Identify hash collision
- Describe linear probing method
- Write algorithms for linear probing method
- Describe quadratic and random probing methods
- Describe double hashing method
- Write algorithms for double hashing method
- Describe rehashing method
- Write algorithms for rehashing method
- Describe chaining method
- Write algorithms for chaining method

10.1 Hashing

Hashing means special type of searching for storing and retrieving data. In hashing method, an array or table is used to store data that is called *hash table*. A special function called hash function is used to store and retrieve data. For example a simple hash function is as follows:

hash function = key_value mod m;

where key_value may be *Roll No.* of a student, *Personal ID* etc. *m* is the size of the hash table, say *m* = 100.

In hashing method, the function (hash function) which is used to store the records (data) in the hash table, the same function must be used to retrieve the data (records) from the hash table.

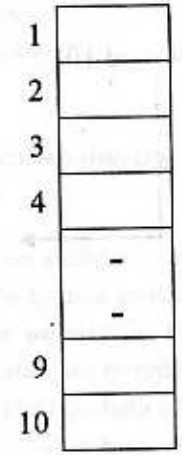


Figure-10.1: Hash table

10.2 Hash function

A function that is used to transform a value (key_value) into a direct address of the hash table is called a hash function. There are several methods to find out the direct address using hash function such as division method, mid-square method, folding method etc.

10.2.1 Division method

Suppose *m* is the size of the hash table. In this method **modulus** operation is used as follows:

h (key_value) = key_value mod m

Example: Let *m* = 10, key values are 13, 4, 12, 9, 24 etc. Then direct addresses are 3, 4, 2, 9, 5 etc.

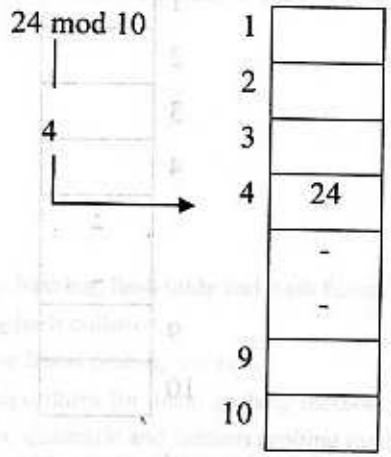


Figure-10.2: Storing key-value in hash table

10.2.2 Mid-square method

Here we have to take the binary representation of the key_value. Then we square these binary numbers and pick out the middle k bits. Now these k bits will be used as hash address. The size of the hash table will be (at least) 2^k 1.

Example: Binary form of *key_value* = 1011001, by squaring we get 1111011110001, let $k = 3$, then middle 3 bits = 111 = 7. So, the hash address is 7.

10.2.3 Folding methods

Take a key value and divide it into equal parts. Each part will be an integer. The hash address is calculated by adding all the integers.

Example: Suppose a *key_value* is 324112. If we divide it into 2-digit integers (three parts) and add then, we get

$$32 + 41 + 12 = 85.$$

If all the key values are 6-digits long, then partitioning into 2-digit groups yields hash address from 0 to 297 ($6 + 3 = 2 \Rightarrow 99 \times 3 = 297$). Using 2-digits highest number we can represent is 99.

10.3 Hash collision

Hash collision is the situation when two data (records) have to be stored in the same position of the hash table.

As for example, if a record has been stored in index number 15 of the hash table, then usually it is not possible to store another record in index number 15. However, from hash function we may get the same result, *i.e.*15 (as before). It means now we have to store the record in index number 15 of the hash table. But the index number 15 of the hash table is already occupied by another record. This situation is called hash collision. There are some schemes or methods for resolution or resolving of hash collision.

10.3.1 Linear probing method

Linear probing is a hash collision resolution scheme. If the target cell of the hash table is already occupied, then we have to store the necessary data in the hash table using following methods.

Suppose, y_0 is the result of the hash function, which means, we have to store data in the cell of the hash table whose index is y_0. If y_0 is occupied, then we try to store the data in the cell whose index is

$$y_1 = y_0 + 1$$

If y_1 is also occupied, then we store the data in the cell whose index is

$$y_2 = y_1 + 1$$

We shall follow the above method until we find an empty cell in the hash table. If we store data in the hash table using above method we have to use same hash function and same method at the time of retrieval of the data.

$$y_0 = \text{key\_value mod m}$$

Let *key-value* = 205 and $m = 100$.

Now $y_0 = 205$ *mod* $100 = 5$, if cell number 5 is occupied then $y_1 = 5 + 1 = 6$, that means we shall try to store data in cell number 6.

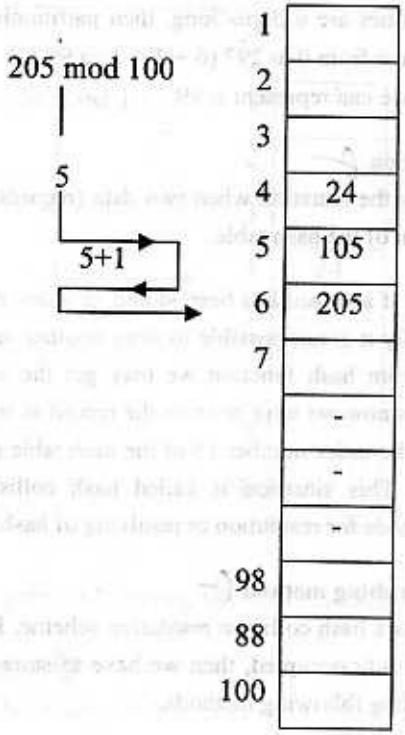


Figure-10.3: Hash collision and resolution with linear probing

Algorithm 10.1: Algorithm to create a hash table with data to resolve hash collision using linear probing method

1. Input list of elements (*key_values*);
2. Create an empty hash table, T[1……*m*]
3. for (*i* = 1 to *m*)
   {
   i) *index* = *key_value* mod *m*;
   if (T[*index*] = 0), then T[*index*] = *key_value*;
   ii) else {
   do {
   index = *index* +1;
   if (index >= m) index= 0;
   } while (T[*index*] != 0);
4. T[*index*] = *key_value*;
   }
5. Output a hash table with data (elements).

Comments: T[*m*] is an array of size ***m***. Here we assume that the number of elements in the list is less than or equal to *m*. We have taken a hash function, ***key-value mod m*** to find out the index of hash table. Here we resolve the hash collision using linear probing method.

Algorithm 10.2: Algorithm to retrieve a data element from a hash table

1. Input the hash table, T[1……*m*], *key-value* to be retrieved;
2. *h* = *key-value* mod *m*;
3. if (T[*h*] = *key-value*), then *item* = T[*h*];
4. else {
   do {
   h = *h* + 1;
   } while (T[*h*] != *key-value*)
5. *item* = T[*h*];
   }
6. Output: the target value in *item*

Comments: Here we assume that we have stored data using hash function, ***key-value* mod *m*** and resolved the collision using linear probing method. We have taken a variable, *item* where we retrieve the target key-value.

10.3.2 Quadratic probing method

In this method, if we find the target cell of the table is occupied, then we use following function as probing function to store the item.

$y_{i+1} = (y_i + k^2) \bmod m$; where $k = 1, 2, 3$ etc. and so on.

And, $y_i = key\text{-}value \bmod m$.

That means, at first we try $y_0 + 1$, then $y_0 + 4$, $y_0 + 9$ and so on until an empty cell of the hash table is found.

If $y_0 = 1$, then $y_1 = 2$, $y_3 = 5$, $y_4 = 10$ etc.

10.3.3 Random probing method

Here the following function may be used as probing function:

$y_{i+1} = (y_i + r) \bmod m$, where $i = 0, 1, 2, 3$ and so on and r is an integer value that is relatively prime to m. Two integers are relatively prime to each other if their greatest common divisor is 1, *i.e.*,

gcd $(r, m) = 1$; //gcd for Greater Common Divisor.

If $y_0 = 3$, and $r = 2$, then $y_1 = 5, y_2 = 7, y_3 = 9, y_4 = 11$ etc.

10.3.4 Double hashing method

In this method we find a hash address using a hash function. If the address (position) is not empty we use second hash function and find a hash value. Using the first hash value (address) and the second hash value we find the new hash address (position) in the hash table. We can define hash functions and find out the new hash address as follows:

h_o= key-value mod m;

if h_o is occupied, we find the address as below:

$h_i = ((h_o + r)*k) \bmod m$;

where, r = key-value mod p,

k=1,2,3... ; here m and p should be prime numbers and p < m.

Example: Suppose we have key-value 211

i) For the first key value:
   y_o=911 mod 7=1

ii) For the second key-value:
   232 mod 7 = 1 is not empty.
   r = 232 and 5 = 2
   $h_1 = ((h_o + r)*k) \bmod m$
   $= ((1+2)*1) \bmod 7$
   $= 3 \bmod 7 = 3$

iii) 624 mod 7=1 is not empty
   624 mod 5 = 4
   $h_1 = ((1+4)*1) \bmod 7$
   = 5, which is empty.

Algorithm 10.3: Algorithm to create a hash table with data to resolve hash collision using double hashing method.

```
1. Create a table with zeros in all cells, T[1.. m];
2. for (i = 1 to m)
   {
   k = 1;
3. ind = key % m;
4. if (table [ind] == 0) table[ind] = key;
5. else
    {
    r = (key % p) + 1;
   do
    {
   ind = ((ind+r)*k) mod m;
   k = k + 1;
    } while (table[ind]!=0);
   } //end of else
6. table [ind] = key;
   } //end of for
7. output: a hash table with key values.
```

Algorithm 10.4: Algorithm to retrieve an item from a hash table, where collision has been resolved using double hashing

```
1. Input the hash table with data, , T[1.....m]
   (the hash table that was created with data using
   algorithm 10.3)
2. Input the key-value to be retrieved
3. h = key-value % m;
4. k = 1;
5. if (T[h] == key-value), item = T[h];
6. else {
        r = (key-value % p) + 1;
7.      do
         {
         h = ((h + r)*k) mod m
         k = k + 1
         } while (T[h] != key-value);
         } // end of else
8. item =T[h];
9. Output: the valuable item (the target key-value).
```

10.3.5 Rehashing method

In this method two hash functions are used. Suppose, one function is f and another function is h. Then we use following process to resolve the collision. If f (*key-value*) = y_0 is occupied, then we find $h(y_0) = y_1$, $h(y_1) = y_2$, $h(y_2) = y_3$ and so on. Here we repeatedly use the second hash function.

Suppose, f (key_value) = key_value mod m and h = (f(key_value + r) mod m, where r is a small prime number and gcd (m, r) =1, Let m = 10, r = 3 and key values are 621, 901 and 811.

i. For the first key value, $y_0 = 611$ and 10 = 1 is free.

ii. For the second value,

$y_0 = 901 \bmod 10 = 1$ is occupied.

$y_1 = (1+3) \bmod 10 = 4$ is free.

iii) For the third key value,

$y_0 = 811 \bmod 10 = 1$ is occupied.

$y_1 = (1 + 3) \bmod 10 = 4$ is occupied.

$y_2 = (4 + 3) \bmod 10 = 7$ is free.

Algorithm 10.5: An algorithm to create a hash table with data to resolve hash collision using rehashing method.

1. Input key values and r
2. Create a table with zero in all cells
3. for (1 to m)

```
{
ind = key mod m;
if (table[ind] == 0), table [ind] = key;
```

4. else

```
  {
do
  {
  ind = (md + r) mod m;
  } while (table[ind]!=0);
  } // end of else
 table[ind] = key;
} // end of for
```

5. Output: table with key values

Algorithm 10.6: Retrieve data element from a hash table, where collision has been resolved using rehashing method.

1. Input the hash table, $T[1.....m]$, *key_value* to be retrieved (the table that was created using the algorithm 10.5)
2. for (i = 1 to m)

```
{
h = key_value mod m;
```

3. if ($T[h]$ = key-value), then *item* = $T[h]$;
4. else

```
{
do {
    h = (h + r) mod m;
    } while (T[h] != key_value);
}
```

5. *item* = $T[h]$;

```
}
```

6. Output: the variable item.

10.3.6 Chaining method

It is a hash collision resolution scheme. Here hash table is an array of pointers. Each cell contains a pointer that points to the first node of a linked list. We shall store the data not in the hash table but as some linked lists, which are linked (connected) with the hash table. This has been illustrated in Figure-10.4 and Figure-10.5.

Suppose we have a hash function, *key_value mod* 100. Let *key_value* = 102. Then h_0 = 102 mod 100 = 02; that means we shall store 102 in a node that will be linked by the hash table whose index number is 02.

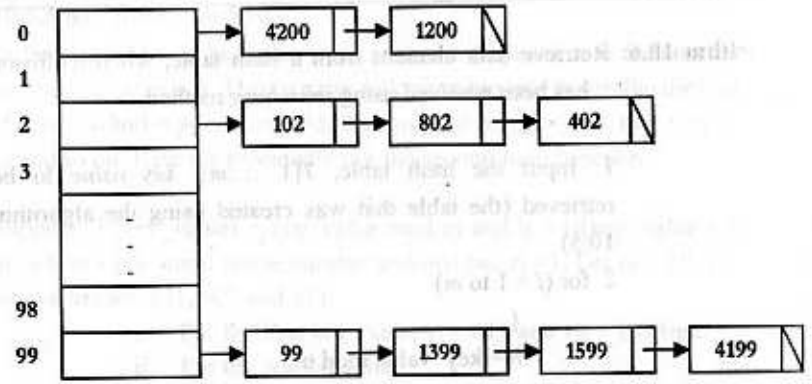


Figure-10.4: Implementation of Chaining Method

If another *key_value* is 802, then 802 mod 100 = 2, now we will create another node for 802 and link this node with the first node (where data value is 102).

Here, we have to use same hash function for storing and retrieving data.

Algorithm 10.7: Algorithm to create a hash table using chaining method

1. Declare node and table as array of pointers:
   (i). struct node
   {
   int *key_value*;
   node *next;
   }
   (ii). node *table[*m*], *nptr, *tptr;
2. Create an empty hash table:
   for (*i* = 1 to *m*)
   table[*i*] = NULL;
3. Input key_value and create a new node (with key_value)
4. *index* = *key_value mod m*;
   if (table[*index*] = NULL), table[*index*] = nptr;
   else {
   tptr = table[*index*];
   while (tptr→next != NULL)
   {
   tptr = tptr→next;
   }
   tptr→next = nptr;
   }
5. Repeat Step-2 to 4 each time you want to input another key_value.
6. Output: a chain of linked list

Comment: ***table[m]*** is a array of pointers of ***m*** size, ***nptr*** is the pointer to the new node and ***tptr*** is a temporary pointer that is used to make link between existing last node and new node.

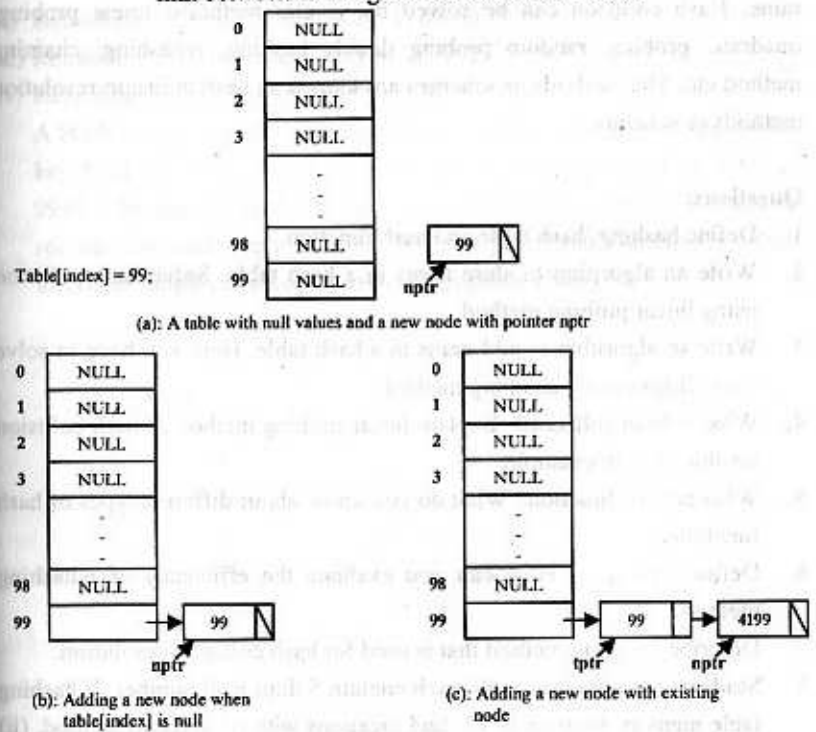


Figure-10.5: Create a hash table using chaining method to avoid hash collision.

Summary:

Hashing is the method of storing and retrieving data by searching in/from a table called *hash table* using a special function called *hash function*. There are several methods to find out the position (address) using hash function: division, mid-square, folding method etc. In division method, key-value (the value that is used as a means of finding the position) is modulo divided by the size of the hash table. In mid-square method, the key-value is represented in binary form, then we square these values, and finally we pick out the middle k bits. These k bits are used as hash address. In folding methods, the key-value is divided into equal parts. The hash address is calculated by adding all of these parts. Hashing has a limitation of causing ***hash collision***, which is the situation when hash address for two data (records) is found the same. Hash collision can be solved by several methods: linear probing, quadratic probing, random probing double hashing, rehashing, chaining method etc. The methods or schemes are known as hash collision resolution methods or schemes.

Questions:

1. Define hashing, hash table and hash function.
2. Write an algorithm to store items in a hash table. Solve hash collision using linear probing method.
3. Write an algorithm to add items in a hash table. Here you have to solve the collision using chaining method.
4. What is hash collision? Explain linear probing method of hash collision resolution with example.
5. What is hash function? What do you know about different types of hash functions?
6. Define rehashing. How can you evaluate the efficiency of rehashing method?
7. Describe chaining method that is used for hash collision resolution.
8. Students in your class is 60, each contain 5 digit Roll number. If hashing table memory location is 97, find locations with (i) division method, (ii) Mid-square method & (iii) Folding method for the following Roll numbers. (99705, 99735 & 99759)

9. Suppose a Hash table contain 11 memory locations and a file contain 8 Records, where

Record	A	B	C	D	E	F	G	H
Hash address H(k)	4	8	2	11	4	11	5	1

   Using linear probing method show how the records will appear in memory.

10. A Hash table contains 1000 slots, indexed from 0 to 999. The elements in the table have keys that range from 1 to 100000. The original hash function is Key MOD 1000. Which, if any, of the following collision resolution schemes would work correctly and why?

(i) Rehashing, with function = (Key+1) MOD 1000
(ii) Rehashing, with function = (Key+2) MOD 1000
(iii) Rehashing, with function = Key MOD 998
(iv) Rehashing, with function = (Key+3) MOD 1000

A Hash table is used to store employee records of EmployeeType whose key field is called IdNum. The key may range in value from 1000 to 9999. The original hash function is key MOD 100. There is a chain of records for each hash address. Write a procedure (function), which inserts an employee record into the appropriate chain.

Problems for Practical (Lab) Class

Hashing related problems

Problem 10-1: write a program to **store data** (key values) in a hash table using a hash function

where **collision** has been resolved using **linear probing** method. Display the data and respective indices of the table.

Hints: generate 10 three digit integers. Find out the hash address (index) using hash function such as h = key value mod m, where m = 10 and store them in the hash table. Display the data and indices of the table.

Problem 10-2: write a program to **retrieve** a particular data (key value) from the hash table using a hash function where **collision** has been resolved using **linear probing** method. Display the data and respective index of the table.

Hints: use the hash table that was created by the program as a solution of the **problem 10-1.**

Problem 10-3: write a program to **store data** (key values) in a hash table using a hash function where **collision** has been resolved using **double hashing** method. Display the data and respective index of the table.

Problem 10-4: write a program to **store data** (key values) in a hash table using a hash function where **collision** has been resolved using **double hashing** method. Display the data and respective index of the table.

Problem 10-5: write a program to **store** the data (records) of the 10 students in a hash table using a hash function where **collision** has been resolved using **linear probing** method. Display the data of all the students and respective indices of the table. Each student record contains id of six digits, name and marks.

Problem 10-6: write a program to retrieve the data (records) of **a particular student** from the hash table using a hash function where collision has been resolved using linear probing method. Display the data of the students and respective index of the table.

Hints: use the hash table that was created by the program as a solution of the **problem 10-5.**

Problem 10-7: write a program to **store data** (key values) in a hash table using a hash function where **collision** has been resolved using **chaining** method. Display the data and respective indices of the table.

Problem 10-8: write a program to **retrieve** a particular data (key value) from the hash table using a hash function where **collision** has been resolved using **chaining** method. Display the data, node number and respective index of the table.

Hints: use the hash table that was created by the program as a solution of the **problem 10-7.**

APPENDIX-A

DATA STRUCTURES IN JAVA

OBJECTIVES:

A data structure is the organization of data in a computer's memory or in a disk file. The correct choice of data structure allows major improvements in program efficiency. A way to look at data structures is to focus on their strengths and weaknesses. Some data structure operations on the following important data structures are implemented in JAVA:

- Array
- Stack
- Queue
- Linked list
- Recursion
- Tree

11.1 Array

11.1.1 Creating an array

There are two kinds of data in Java: primitive types (such as int and double), and objects. Arrays are treated as objects in Java. Accordingly new operator is used to create an array.

Example:

```
int[] intArray; // defines a reference to an array
intArray = new int[100]; // creates the array, and
// sets intArray to refer to it
```

These two lines can be combined in a single-statement approach:

```
int[] intArray = new int[100];
```

Because an array is an object, its name is a reference to an array; it's not the array itself. The array is stored at an address elsewhere in memory, and intArray holds only this address.

Arrays have a length field, which you can use to find the size, in bytes, of an array:

```
int arrayLength = intArray.length; // find array length
```

Size of an array cannot be changed after it's been created.

11.1.2 Accessing array elements

Array elements are accessed using square brackets.

Example:

```
temp = intArray[3]; // get contents of fourth element of array
intArray[7] = 66; // insert 66 into the eighth cell
```

In Java, the first element is numbered 0, if an index is less than 0 or greater than the size of the array less than 1, the "Array Index Out of Bounds" runtime error will occur.

Initialization

An array of integers is automatically initialized to 0 when it's created. If an array of objects is created like the following:

```
autoData[] carArray = new autoData[4000];
```

then, until they're given explicit values, the array elements contain the special null object.

The following syntax will initialize an array of a primitive type to something besides 0:

```
int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };
```

Example:

Problem 11.1: Given an array of 10 elements.

a) Display the contents of the array
b) Find, whether the array contains the value 66
c) Delete the value 55 from the array and display the remaining elements.

Let's look at some example programs that show how an array can be used.

Algorithm 11.1: Array

```
import java.io.*; // for I/O
class ArrayApp
{
public static void main(String[] args) throws IOException
{
        int[] intArray = {77, 99, 44, 55, 22, 88, 11, 0, 66, 33}; //array
        reference with initialization
        int nElems = 10; // number of items in array
        int j; // loop counter
        int searchKey; // key of item to search for

        // display items of the array
        for(j=0; j<nElems; j++)
                System.out.print(intArray[j] + " ");
        System.out.println("");

        // find item with key 66
        searchKey = 66;
        for(j=0; j<nElems; j++) // for each element,
                if(intArray [j] == searchKey) // found item?
                        break; // yes, exit before end
        if(j == nElems) // at the end?
                System.out.println("Can't find " + searchKey); // yes
        else
                System.out.println("Found " + searchKey); // no
        // delete item with key 55
        searchKey = 55; for(j=0; j<nElems; j++) // look for it
                if(intArray [j] == searchKey)
                        break;
        for(int k=j; k<nElems; k++) // move higher ones down
                intArray [k] = intArray [k+1];
        nElems--; // decrement size
```



```
        // display items after deletion
        for(j=0; j<nElems; j++)
                System.out.print(intArray [j] + " ");
        System.out.println("");

} // end main()

} // end class ArrayApp
```

Comments: In this program, an array called intArray is created with 10 data items in it, an item with value 66 is searched, items with value 55 is deleted, and then the remaining nine items are displayed. The output of the program looks like this:

Output:

```
77 99 44 55 22 88 11 0 66 33
Found 66
77 99 44 22 88 11 0 66 33
```

11.2 Stacks

A stack allows access to only one data item: the last item inserted. If this item is removed, then the next-to-last item can be accessed and so on. This is a useful capability in many programming situations.

11.2.1 Java code for a stack

Stack implements last in first out approach.

Problem 11.2: Create a stack with the capacity 10 and

a) Push 4 given items in the stack

b) Display all the items by popping them from the stack.

Algorithm 11.2: The StackImpl.java Program

```
package stack;
```

```
class StackUnderflowException extends Throwable
{
}

class StackOverflowException extends Throwable
{
}

public class StackImpl {

  private int stackData[];
  private int stackTop;
  private int capacity;

  //constructor
  public StackImpl (int capacity)
  {
    stackData = new int[capacity];
    this.capacity = capacity;
    this.stackTop = -1;
  }
  public int pop ()throws StackUnderflowException
  {
    if (stackTop >= 0)
      return stackData[stackTop--];
    else
    {
      throw new StackUnderflowException();
    }

  }

  public void push(int item) throws StackOverflowException
  {
    if (stackTop < capacity-1)
```



```
      stackData[++stackTop]= item;
    else
      throw new StackOverflowException();
  }
  public boolean isEmpty()
  {
        if (stackTop < 0) return true;
    else return false;
  }
  public boolean isFull()
  {
        if (stackTop >= capacity-1) return true;
    else return false;
  }
}//end class StackImpl
class StackApp
{
    public static void main(String[] args)
    {
        StackImpl aStack = new StackImpl(10); // make new stack
        with the capacity 10
        aStack.push(20); // push items onto stack
        aStack.push(40);
        aStack.push(60);
        aStack.push(80);
        while( !aStack.isEmpty() ) // until it's empty,
        { // delete item from stack
            int value = aStack.pop();
            System.out.print(value); // display it
            System.out.print(" ");
        } // end while
        System.out.println("");

    } // end main()
} // end class StackApp
```

The main() method in the StackApp class creates a stack that can hold 10 items, pushes 4 items onto the stack, and then displays all the items by popping them off the stack until it's empty.

Here's the Output:

80 60 40 20

It is worth to notice that how the order of the data is reversed. Because the last item pushed is the first one popped; the 80 appears first in the output.

11.3 Queue

The word *queue* is British for *line* (the kind you wait in). In Britain, to "queue up" means to get in line. In computer science a queue is a data structure that is similar to a stack, except that in a queue the first item inserted is the first to be removed (FIFO), while in a stack, as we've seen, the last item inserted is the first to be removed (LIFO).

11.3.1 A circular queue

When you insert a new item in the queue in the Workshop applet, the Front arrow moves upward, toward higher numbers in the array. When you remove an item, Rear also moves upward. Try these operations with the Workshop applet to convince yourself it's true. You may find the arrangement counter-intuitive, because the people in a line at the movies all move forward, toward the front, when a person leaves the line. We could move all the items in a queue whenever we deleted one, but that wouldn't be very efficient. Instead we keep all the items in the same place and move the front and rear of the queue.

The trouble with this arrangement is that pretty soon the rear of the queue is at the end of the array (the highest index). Even if there are empty cells at the beginning of the array, because you've removed them with Rem, you still can't insert a new item because Rear can't go any further. Or can it?

11.3.2 Wrapping around

To avoid the problem of not being able to insert more items into the queue even when it's not full, the Front and Rear arrows *wrap around* to the beginning of the array. The result is a *circular queue* (sometimes called a *ring buffer*).

You can see how wraparound works with the Workshop applet. Insert enough items to bring the Rear arrow to the top of the array (index 9). Remove some items from the front of the array. Now, insert another item. You'll see the Rear arrow wrap around from index 9 to index 0; the new item will be inserted there.

Insert a few more items. The Rear arrow moves upward as you'd expect. Notice that once Rear has wrapped around, it's now below Front, the reverse of the original arrangement. You can call this a *broken sequence*: the items in the queue are in two different sequences in the array. Delete enough items so that the Front arrow also wraps around. Now you're back to the original arrangement, with Front below Rear. The items are in a single *contiguous sequence.*

11.3.3 Java code for a queue

The queue.java program features a Queue class with enqueue (), dequeue (), isFull(), isEmpty() and size() methods.

The main() program creates a queue of five cells, inserts four items, removes three items, and inserts four more. The sixth insertion invokes the wraparound feature. All the items are then removed and displayed. The output looks like this:

40 50 60 70 80

Problem 11.3: Create a queue of five items and

a) insert four given into the queue
b) remove three items from the queue
c) again insert four more items into the queue
d) display all the items by removing them from the queue.

Algorithm 11.3: Queue

```
package queue;
class QueueUnderflowException extends Throwable
{
}
class QueueOverflowException extends Throwable
{
}
class QueueImpl
{
private int capacity ;
private int[] queueData;
private int front;
private int rear;
private int nItems;
//--------------------------------------------------------------
public QueueImpl(int capacity) // constructor
{
        this.capacity = capacity;
        queueData = new int[capacity];
        front = 0;
        rear = -1;
        nItems = 0;
}
//--------------------------------------------------------------
public void enqueue (int item) // put item at rear of queue
{
        if (isFull()) throw new QueueOverflowException();
        else
        {
                if(rear == capacity-1) // deal with wraparound
                rear = -1;
                queueData[++rear] = item; // increment rear and insert
                nItems++; // one more item
        }
```



```
}
public int dequeue () // take item from front of queue
{
        if (isEmpty()) throw new QueueUnderflowException ();
        else
        {
                int temp = queueData [front++]; // get value and incr front
                if(front == capacity) // deal with wraparound
                front = 0;
                nItems--; // one less item
                return temp;
        }
}
public boolean isEmpty() // true if queue is empty
{
        return (nItems==0);
}
public boolean isFull() // true if queue is full
{
        return (nItems==capacity);
}
//--------------------------------------------------------------
public int size() // number of items in queue
{
        return nItems;
}
//--------------------------------------------------------------
} // end class Queue
class QueueApp
{
public static void main(String[] args)
{
        QueueImpl aQueue = new QueueImpl (10); // queue holds 10 items
        aQueue.enqueue(10); // insert 4 items
        aQueue. enqueue (20);
```

```
        aQueue. enqueue (30);
        aQueue. enqueue (40);
        aQueue.dequeue (); // remove 3 items
        aQueue. dequeue (); // (10, 20, 30)
        aQueue. dequeue ();
        aQueue. enqueue (50); // insert 4 more items
        aQueue. enqueue (60); // (wraps around)
        aQueue. enqueue (70);
        aQueue. enqueue (80);
        while( !aQueue.isEmpty() ) // remove and display all items
        {
                int n = aQueue. dequeue (); // (40, 50, 60, 70, 80)
                System.out.print(n);
                System.out.print(" ");
        }
        System.out.println("");
} // end main()
} // end class QueueApp
```

QueueImpl class includes some fields, namely, capacity, front, rear and number of items currently in the queue.

The enqueue() method assumes that the queue is not full. If the queue is full it raises a QueueOverflowException and quits. Otherwise an item can be inserted in the queue. Normally, insertion involves incrementing rear and inserting at the cell rear now points to. However, if rear is at the top of the array, at capacity -1, then it must wrap around to the bottom of the array before the insertion takes place. This is done by setting rear to –1, so when the increment occurs rear will become 0, the bottom of the array. Finally nItems is incremented.

The dequeue() method assumes that the queue is not empty. If the queue is empty it will raise a QueueUnderflowException and quits. Removal always starts by obtaining the value at front and then incrementing front. However, if this puts front beyond the end of the array, it must then be wrapped around to

0. The return value is stored temporarily while this possibility is checked. Finally, nItems is decremented.

The isEmpty(), isFull() and size() methods all rely on the nItems field, respectively checking if it's 0, if it's capacity, or returning its value.

11.4 A simple linked list

Our first example program, linkList.java, demonstrates a simple linked list. The only operations allowed in this version of a list are

- Inserting an item at the beginning of the list
- Deleting the item at the beginning of the list
- Iterating through the list to display its contents

11.4.1 The Link class

Here is the complete class definition:

```
class Link
{
        public int value ; // data item
        public Link next; // next link in list

        public Link(int value ) // constructor
        {
                this.value = value ; // initialize data
                this. next = null; // ('next' is set to null)
        }
        public void displayLink() // display ourself
        {
                System.out.print("{" + value + "} ");
        }
} // end class Link
```

Here in addition to the data, there's a constructor and a method, displayLink(), that displays the link's data in the format {22}.

The constructor initializes the data. The next field is explicitly initialized to null for clarity, although its automatically set to null when it's created. The null value means it doesn't refer to anything, which is the situation until the link is connected to other links.

11.4.2 The LinkList class

The LinkList class contains only one data item: a reference to the first link on the list. This reference is called first. It's the only permanent information the list maintains about the location of any of the links. It finds the other links by following the chain of references from first, using each link's next field.

Algorithm 11.4: Linked List

```
class LinkList
{
        private Link first; // ref to first link on list
        public void LinkList() // constructor
        {
                first = null; // no items on list yet
        }
        public boolean isEmpty() // true if list is empty
        {
                return (first==null);
        }
        // insert at start of list
        public void insertFirst(int id)
        { // make new link
                Link newLink = new Link(id);
                newLink.next = first; // newLink --> old first
                first = newLink; // first --> newLink
        }
        public Link deleteFirst() // delete first item
        { // (assumes list not empty)
                Link temp = first; // save reference to link
                first = first.next; // delete it: first-->old next
```



```
                return temp; // return deleted link
        }
        public void displayList()
        {
                System.out.print("List (first-->last): ");
                Link current = first; // start at beginning of list
                while(current != null) // until end of list,
                {
                        current.displayLink(); // print data
                        current = current.next; // move to next link
                }
                System.out.println("");
        }
}//end LinkList
```

The constructor for LinkList sets first to null. However, the explicit constructor makes it clear that this is how first begins.When first has the value null, we know there are no items on the list. If there were any items, first would contain a reference to the first one. The isEmpty() method uses this fact to determine whether the list is empty.

The insertFirst() method of LinkList inserts a new link at the beginning of the list. This is the easiest place to insert a link, because first already points to the first link. To insert the new link, we need only set the next field in the newly created link to point to the old first link, and then change first so it points to the newly created link. In insertFirst() we begin by creating the new link using the data passed as
arguments. Then we change the link references as we just noted.

The deleteFirst() method is the reverse of insertFirst(). It disconnects the first link by rerouting first to point to the second link. This second link is found by looking at the next field in the first link.

The second statement is all you need to remove the first link from the list. We choose to also return the link, for the convenience of the user of the linked list, so we save it in temp before deleting it, and return the value of temp.

The deleteFirst() method assumes the list is not empty. Before calling it, program verify this with the isEmpty() method.

To display the list, you start at first and follow the chain of references from link to link. A variable current points to (or technically *refers* to) each link in turn. It starts off pointing to first, which holds a reference to the first link. The statement changes current to point to the next link, because that's what's in the next field in each link.

At each link, the displayList() method calls the displayLink() method to display the data in the link.

Problem 11.4: Insert four given items in the linked list and display them. Then removed all the elements from the linked list.

```
class LinkListApp
{
public static void main(String[] args)
{
        LinkList theList = new LinkList(); // make new list
        theList.insertFirst(22); // insert four items
        theList.insertFirst(44);
        theList.insertFirst(66);
        theList.insertFirst(88);
        theList.displayList(); // display list
        while( !theList.isEmpty() ) // until it's empty,
        {
                Link aLink = theList.deleteFirst(); // delete link
                System.out.print("Deleted "); // display it
                aLink.displayLink();
                System.out.println("");
        }
        theList.displayList(); // display list
} // end main()
} // end class LinkListApp
```


In main() we create a new list, insert four new links into it with insertFirst(), and display it. Then, in the while loop, we remove the items one by one with deleteFirst() until the list is empty. The empty list is then displayed. Here's the output from linkList.java:

```
List (first-->last): {88} {66} {44} {22}
Deleted {88}
Deleted {66}
Deleted {44}
Deleted {22}
List (first-->last):
```

Linked-List efficiency

Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two references, which takes O(1) time.

Finding, deleting, or insertion next to a specific item requires searching through, on the average, half the items in the list. This requires O(N) comparisons. An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or deleted.

11.5 Recursion: Finding factorials

Factorials are similar in concept to triangular numbers, except that multiplication is used instead of addition. The triangular number corresponding to n is found by adding n to the triangular number of n–1, while the factorial of n is found by multiplying n by the factorial of n–1. That is, the fifth triangular number is 5+4+3+2+1, while the factorial of 5 is 5*4*3*2*1, which equals 120. The factorial of 0 is defined to be 1. Factorial numbers grow large very rapidly, A recursive method can be used to calculate factorials. It looks like this:

Algorithm 11.5: Factorials

```
long int factorial(int n)
{
        if(n==0)
                return 1;
        else
                return (n * factorial(n-1) );
}
```

The base condition occurs when n is 0.

```
Enter a number: 6
Factorial =720
```

Various other numerological entities lend themselves to calculation using recursion in a similar way, such as finding the greatest common divisor of two numbers (which is used to reduce a fraction to lowest terms), raising a number to a power, and so on. Again, while these calculations are interesting for demonstrating recursion, they probably wouldn't be used in practice because a loop-based approach is more efficient.

11.6 Binary tree

Tree combines the advantages of two other structures: an ordered array and a linked list. You a tree can be searched, as an ordered array, and also inserted and deleted items quickly, as with a linked list.

Slow insertion in an ordered array

In an ordered array, where all items are arranged in an order, it's quick to search such an array for a particular value, using a binary search. Applying this process repeatedly finds the object in O(logN) time. It's also quick to iterate through an ordered array, visiting each object in sorted order. On the other hand, if an item be inserted into an ordered array, firstly a position to be found where the item will go, and then move all the objects with greater keys up one space in the array to make room for it. These multiple moves are time consuming, requiring, on the average, moving half the items (N/2 moves). Deletion involves the same multimove operation, and is thus equally slow. If there is a lot of insertions and deletions, an ordered array is a bad choice.

Slow searching in a linked list

On the other hand, insertions and deletions are quick to perform on a linked list. They are accomplished simply by changing a few links. These operations require O(1) time Unfortunately, however, *finding* a specified element in a linked list is not easy. It should be started at the beginning of the list and to visit each element until the sought item is found.

It will take on an average of N/2 items, comparing each one's key with the desired value. This is slow, requiring O(N) time.

Binary trees to the rescue

Trees provide quick insertion and deletion of a linked list, and also the quick searching of an ordered array. It shows both these characteristics, and are also one of the most interesting data structures.

11.6.1 The Node class

First, we need a class of node objects. These objects contain the data representing the objects being stored, references to each of the node's two children and a reference to the parent of each node. . Here's how that looks:

```
class BST {
   //members
   private int value          = 0;
   private BST left           = null;
   private BST right          = null;
   private BST parent         = null;
   public int getValue ()     {return this.value;}
   public BST getLeft()       {return this.left;}
   public BST getRight()      {return this.right;}
   public BST getParent()     {return this.parent;}
} //end BST class
```

The BST class has a number of methods: some for finding, inserting, and deleting nodes, several for different kinds of traverses, and one to get the sorted data. BST also has two overloaded constructors to properly initialize the BST class. The following is a skeleton version:

```
package bst;
class BST {
```

```
//constructors
public BST (int aValue)  {  }
public BST (int[] data)  {  }
public void insertNode(int aValue)
{
}
public void traverseInOrder()
{
}
public void traversePreOrder()
{
}
public void traversePostOrder()
{
}
public Vector getSortedData()
{
}
public BST search(int aValue)
{
}
public void deleteNode(int item)
{
}
}//end BST class
```

11.6.2 The TreeApp class

Finally, a class is required to perform operations on the tree. Here's how you might write a class with a main() routine to create a tree, insert three nodes into it, and then search for one of them and traverse the tree in different ways. Here is the listing of the class BSTApp:

Problem 11.5: Given 10 data.

a) Create a binary search tree
b) Show the output for preorder, in order and postorder traversing of the BST

c) Display the sorted list using Vector
d) Delete the node containing value=50 and then display the output for inorder traversing
e) Find, whether the BST contains the elements 90 and 120 or not.

```
package bst;
import java.util.Vector;
class BstApp
{
  public static void main (String[] args)
  {
    // test data
    int testData [] = {50, 25, 75, 22, 40,60, 90, 15, 23, 80};
    BST bst = new BST(testData);
    System.out.print("Preorder Traversing:\t");
    bst.traversePreOrder();
    System.out.println();
    System.out.println();

    System.out.print("Inorder Traversing:\t");
    bst.traverseInOrder();
    System.out.println();
    System.out.println();


    System.out.print("Postorder Traversing:\t");
    bst.traversePostOrder();
    System.out.println();
    System.out.println();


    Vector sData = bst.getSortedData();
```

```
    System.out.print("Sorted List:\t\t" );
    for (int i = 0; i< sData.size(); i++)
    {
      Integer j = (Integer ) sData.elementAt(i);

      System.out.print(j.intValue() + "\t");
    }
    System.out.println();
    System.out.println();

    //try to delete a node
    int item = 50 ;
    bst.deleteNode(item);
    System.out.print("Inorder Traversing after deleting ["+ item + "]:\t");
    bst.traverseInOrder();
    System.out.println();
    System.out.println();

    // /*
     int aValue = 90;
     if (bst.search(aValue) != null)
       System.out.println(aValue + " is found in bst");
     else
       System.out.println(aValue + "is not found in bst");
                aValue = 120;
     if (bst.search(aValue) != null)
       System.out.println(aValue + " is found in bst");

     else
       System.out.println(aValue +" is not found in bst");
    // */
  } //end main

}//end class
```


Next we'll look at individual tree operations: finding a node, inserting a node, traversing the tree, and deleting a node.

11.6.3 Searching for a node

Finding a node with a specific key is the simplest of the major tree operations, so let's start with that. The nodes in a binary search tree correspond to objects containing information. They could be *person objects*, with an employee number as the key and also perhaps name, address, telephone number, salary, and other fields. Or they could represent car parts, with a part number as the key value and fields for quantity on hand, price, and so on. But for simplicity's sake we include only an integer number as the data of tree node.

Algorithm 11.6: Searching in BST

```
public BST search(int aValue)
  {
      BST node = this;

      while (node != null)
      {
            if (aValue == node.getValue()) return node;
            else if (aValue > node.getValue())
                  node = node.right;
            else
              node = node.left;

      }
      return null;

  }//end search
```

Here we start from the current node (by this keyword). A match is sought to see whether it is matched with the searching value (aValue). If a match is found we return the matching node. If a match is not found, we follow either left or right child depending on the value on the node. If all the items in the link are exhausted we return a null indicating that no match could be found.

11.6.4 Inserting a node

To insert a node we must first find the place to insert it. This is much the same process as trying to find a node that turns out not to exist, as described in the section on Find. We follow the path from the root to the appropriate node, which will be the parent of the new node. Once this parent is found, the new node is connected as its left or right child, depending on whether the new node's key is less than or greater than that of the parent.

Algorithm 11.7: Inserting a node

```
public void insertNode(int aValue)
{
  BST node = this;  //insert data from the current node
  if (node == null) return ;

  BST parent = null;

  //find the suitable position within bst
  while (node != null )
  {
    if (aValue > node.value) {
      parent = node;
      node = node.right;
    } else {
      parent = node;
      node = node.left;
    }
  }

  //create a new node

  node = new BST(aValue);

  //add this newly created node to the existing tree
  if (parent == null) return ;
```



```
  node.parent = parent;

  if (aValue > parent.getValue())
    parent.right = node;
  else
    parent.left = node;


}//end insertNode
```

11.6.5 Traversing the tree

Traversing a tree means visiting each node in a specified order. This process is not as commonly used as finding, inserting and deleting nodes. One reason for this is that traversal is not particularly fast. But traversing a tree is useful in some circumstances and the algorithm is interesting. There are three simple ways to traverse a tree. They're called *preorder*, *inorder* and *postorder*. The order most commonly used for binary search trees is inorder, so let's lookat that first, and then return briefly to the other two.

Inorder traversal

An inorder traversal of a binary search tree will cause all the nodes to be visited *in ascending order*, based on their key values. If you want to create a sorted list of the data in a binary tree, this is one way to do it.

The simplest way to carry out a traversal is the use of recursion. A recursive method to traverse the entire tree is called with a node as an argument. Initially, this node is the root. The method needs to do only three things:

1. Call itself to traverse the node's left subtree
2. Visit the node
3. Call itself to traverse the node's right subtree

Visiting a node means doing something to it: displaying it, writing it to a file or whatever.

Algorithm 11.8: Inorder Traversal of BST

```
public void traverseInOrder()
{
        if (this.left != null ) this.left.traverseInOrder();
    System.out.print(this.value +"\t");
    if (this.right != null) this.right.traverseInOrder();

}//end traverseInOrder
```

Preorder and Postorder traversals

There are two more ways besides inorder; they are called preorder and postorder.

A binary tree (not a binary search tree) can be used to represent an algebraic expression that involves the binary arithmetic operators +, -, / and *. The root node holds an operator and each of its subtrees represents either a variable name (like A, B or C) or another expression.

Here's the sequence for a preorder() method:

1. Visit the node.
2. Call itself to traverse the node's left subtree.
3. Call itself to traverse the node's right subtree.

The postorder traversal method contains the three steps arranged in yet another way:

1. Call itself to traverse the node's left subtree.
2. Call itself to traverse the node's right subtree.
3. Visit the node.

Algorithm 11.9: Preorder Traversal of BST

```
public void traversePreOrder()
 {

    System.out.print(this.value +"\t");
```



```
    if (this.left != null ) this.left.traversePreOrder();
    if (this.right != null) this.right.traversePreOrder();

 }//end traverseInOrder
```

Algorithm 11.10: Postorder Traversal of BST

```
 public void traversePostOrder()
 {

    if (this.left != null ) this.left.traversePostOrder();
    if (this.right != null) this.right.traversePostOrder();
    System.out.print(this.value +"\t");

 }//end traverseInOrder
```

11.6.6 Deleting a node

Deleting a node is the most complicated common operation required for binary search trees. However, deletion is important in many tree applications, and studying the details builds character.

You start by finding the node you want to delete, using the same approach we saw in find() and insert(). Once you've found the node, there are three cases to consider.

1. The node to be deleted is a leaf (has no children).
2. The node to be deleted has one child.
3. The node to be deleted has two children.

Algorithm 11.11: Deleting Node from BST

```
public void deleteNode(int item)
{
```

```
BST node = search (item);

BST parent = node.parent;


//item not in bst
if (node == null)
{
        System.out.println (item + " Not found in the BST");
        return ;

}

if (node.left == null && node.right == null && parent == null)
{
        //only the root is present
   System.out.println ("BST contains only a single node. Root can't
be deleted.");
        return ;
    }
    //node has no child, can safely be removed from bst
    if (node.left == null && node.right == null)
    {

        {
        //determine whether it was left or right child of parent
        if (item > parent.value)
            parent.right = null;
        else
            parent.left = null;


        }


            return ;
```



```
    }

    //node has only one child
    if (node.left == null)
    {
            //node has only right child
            //determine whether it was left or right child of parent
                if (item > parent.value)
                parent.right = node.right;
            else
                parent.left = node.right;

        return ;

    }
    else if (node.right == null)
    {
                //node has only left child
            //determine whether it was left or right child of parent
                if (item > parent.value)
            parent.right = node.left;
        else
            parent.left = node.left;
        return ;
    }
    //node has both two children
    //update the value of deleting node with that of the inorder
successor
    //be sure to delete the inorder successor!

    if (node.left != null && node.right != null)
    {
```

```
        BST successor = node.right;

        while (successor.left != null)
        {
                successor = successor.left;
        }
        //now insert value of successor to node.
        node.value = successor.value;
        successor.deleteNode(successor.value);

        return ;

    }

}//end deleteNode
```

Case 1: The node to be deleted has no children

To delete a leaf node, you simply change the appropriate child field in the node's parent to point to null instead of to the node. The node will still exist, but it will no longer be part of the tree.

Case 2: The node to be deleted has one child

This case isn't so bad either. The node has only two connections: to its parent and to its only child. You want to "snip" the node out of this sequence by connecting its parent directly to its child. This involves changing the appropriate reference in the parent to point to the deleted node's child.

Case 3: The node to be deleted has two children

If the deleted node has two children, you can't just replace it with one of these children, at least if the child has its own children. To delete a node with two children, *the node is replaced with its inorder successor.*

The efficiency of Binary tree

Most operations with trees involve descending the tree from level to level to find a particular node. In a full tree, about half the nodes are on the bottom level. (Actually there's one more node on the bottom row than in the rest of the tree.) Thus about half of all searches or insertions or deletions require finding a node on the lowest level. (An additional quarter of these operations require finding the node on the next-to-lowest level, and so on.) During a search we need to visit one node on each level so we can get a good idea how long it takes to carry out these operations by knowing how many levels there are.

Assuming a full tree

In that case, the number of comparisons for a binary search was approximately equal to the base-2 logarithm of the number of cells in the array. Here, if we call the number of nodes in the first column N, and the number of levels in the second column L, then we can say that N is 1 less than 2 raised to the power L, or

$N = 2^L - 1$

Adding 1 to both sides of the equation, we have

$N+1 = 2L$

This is equivalent to

$L = \log2(N+1)$

Summary:

In this chapter we have shown different types of operations on array, stack, queue, linked list, tree etc. The operations are depicted in algorithms or programs using Java.

In Java an array is an object, its name is a reference to an array; it's not the array itself. The array is stored at an address elsewhere in memory. Arrays have a length field, which can be used to find the array size. Size of an array cannot be changed after it's been created. We have discussed various algorithms, such as linear search, binary search, bubble sort etc, that make use of simple arrays. We have also demonstrated some aspects of data structure operations which involve array of objects.

Three important data structures which involve arrays for storage, namely, stack, queue, and priority queue, are examined. We have seen how these structures differ from simple arrays.

A stack allows access to only one data item: the last item inserted. If this item is removed, then the next-to-last item can be accessed and so on. This is a useful capability in many programming situations. In computer science a queue is a data structure that maintains that the first item inserted is the first to be removed (FIFO), while in a stack the last item inserted is the first to be removed (LIFO). The priority queue (heap) is a partially ordered data structure that can readily give the top priority item.

Unlike arrays linked list is an exquisite piece of data structures in which each node of the structure contains a pointer to the next node of the list. Insertion and deletion from a linked list is easy while traversing or sorting a linked list is costly because we have to move from node to node to find the desired one.

Divide and conquer method is analyzed by introducing quick sort and merge sort algorithms. We have also discussed the concept of recursion through various examples.

Trees provide quick insertion and deletion of a linked list, and also the quick searching of an ordered array. It shows both these characteristics, and is also one of the most interesting data structures. Here we provide in depth study of Binary Search Tree (BST): inserting nodes in BST, deleting nodes from BST and traversing BST in various ways. Interestingly preorder traversing will produce a sorted list of nodes.

Questions:

1. Briefly describe the notions of
   - i) The Complexity of algorithms and
   - ii) The Space-Time tradeoff of algorithms.
2. Give flowcharts for
   - i) Double alternative
   - ii) Repeat-For and
   - iii) Repeat-While Structures.
3. Find 26 (mod 7), -2345 (mod 6)
4. Suppose T = 'THE STUDENT IS ILL'. Use INSERT to change T so that it reads:
   - i) THE STUDENT IS VERY ILL
   - ii) THE STUDENT IS ILL TODAY
   - iii) THE STUDENT IS VERY ILL TODAY
5. Consider the linear arrays AAA (5:50), BBB (-5:10) and CCC (1:18). Also suppose Base (AAA) = 300 and w = 4 words per memory cell for AAA.
   - i) Find the number of elements in each array.
   - ii) Find the address of AAA [25], BBB [7] and CCC [15]
6. Sort the following in descending order of complexity:

   $$n\log_2 n,\ 2^n,\ n^2,\ n,\ \log_2 n$$

7. Suppose a company keeps a linear array YEAR (1920: 1970) such that YEAR [K] contains the number of employees born in year K. Write a module to find the number NNN of years in which no employee was born.
8. Consider the alphabetized linear array NAME with the following data:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	C	D	E	G	H	I	K	L	M	R	S	T	W

   - i) Using the linear search algorithm, how many comparisons are used to locate H, M and F?
   - ii) Suppose the Binary Search algorithm is applied to find the location of G. Find the ends BEG and END and middle MID for the test segment in each step.

9. Suppose we want to store the lower triangular sparse matrix A in a linear array B such that $B[1] = a_{11}, B[2] = a_{21}, B[3] = a_{22}, B[4] = a_{31}$ ... and so on.

   Build up a formula that gives the integer L in terms of J and K where $B[L] = a_{JK}$

10. Write pseudo code for **push, pop, enque** and **deque** operations.

11. Consider the following arithmetic expression P written in postfix notation:

    P: 5, 6, 2, +, *, 12, 4, /, -

    i) Convert the above expression in equivalent infix expression.

    ii) Show step by step the contents of the stack as P is scanned element by element for evaluation.

12. Consider the following arithmetic infix expression Q:

    Q: A + (B * C – (D / E ^ F) * G) * H

    Follow step-by-step procedure to convert Q to equivalent postfix expression.

13. Suppose S is the following list of 14 alphabetic characters:

    D, A, T, A, S, T, R, U, C, T, U, R, E, S

    Use quicksort algorithm to find the final position of the first character D. Show each step.

14. Translate by inspection the following infix expression to postfix expression:

    i) (A + B ^ D) / (E – F) + G

    ii) A * (B + D) / E – F * (G + H / K)

15. Let a and b denote positive integers. Suppose a recursive function Q is defined as follows:

$$Q(a,b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b,b)+1 & \text{if } b \le a \end{cases}$$

    Find the value of Q (2, 3), Q (14, 3) and Q (5861, 7)

16. Write a recursive definition of the factorial of a number. Write a function to calculate recursively the factorial of a given number.

17. Consider the following queue of characters, where QUEUE is a circular array with six memory cells:

    FRONT = 2, REAR = 4 QUEUE: __, A, C, D, __, __

    Describe the QUEUE as the following operations take place:

    i) F is added, ii) Two letters are deleted, iii) K, L, M are added, iv) Two letters are deleted, v) R is added.

18. Define binary trees. Express E = (a – b) / ((c * d) + e) using a binary tree.

19. Suppose the following list of letters are inserted in order into an empty binary search tree:

    J, R, D, G, T, E, M, H, P, A, F, Q

    i) Find the final tree T.

    ii) Describe the tree after the node M and D is deleted.

20. Consider the complete tree T with N = 10 nodes:

1	2	3	4	5	6	7	8	9	10
30	50	22	33	40	60	11	60	22	55

    i) By inserting each element once at a time build a Max Heap.

    ii) Delete the top element from the final tree and reconstruct the heap.

APPENDIX-B

DATA STRUCTURE IN C SHARP (C#)

OBJECTIVES:

The Data Structures described in this chapter are as follows which are implemented in C #.

- Arrays
- Array List
- Pointers
- Linked List
- Stacks
- Queues
- Hashing
- Sorting
- Sorted List
- Searching
- Set
- Trees

12.1 Arrays

1. One Dimensional Array
2. Two Dimensional Array
3. Multi Dimensional Array
4. Jagged Array
5. Bit Array

12.1.1 One dimensional array

One dimensional Array in C sharp is declared as follows:

```
int[] sample = new int[number];
```


It means that a one dimensional array named "sample" (user given) is created of length (number+1). Here type of data items is integer.

And after creation the array can be used in the following way:

```
sample[num] = 8;
```

("num" is within the range of "number").

Example: A program that will store some data to its associated index and display those at the output.

Sample Program=

```
Demonstrate a one-dimensional array.
using System;
public class ArrayOD{
public static void Main() {
  int[] sample = new int[10];
  int i;
  for(i = 0; i < 16; i = I + 3)
   sample[i] = i;
  for(i = 0; i < 6; i = i + 1)
   Console.WriteLine("sampleArray[" + i + "]: " +
            sample[i]);
 }
}
```

Sample output=

```
sampleArray[0]=0
sampleArray[1]=3
sampleArray[2]=6
sampleArray[3]=9
sampleArray[4]=12
sampleArray[5]=15
```

12.1.2 Two dimensional array

Two dimensional Array in C sharp is declared as follows:

```
int[,] table = new int[number1, number2];
```

It means that a two dimensional array named "table" (user given) is created where no. of rows is number1 and no. of columns is number2. Here type of data item is integer.

And after creation the array can be used in the following way:

table[num1,num2] = 8;

("num1" and "num2" are within the range of "number1" and "number2")

Example: A program that will store some values (as calculated) to its associated two dimensional index and displays at the output.

Sample Program=

```
using System;
public class ArrayOD {
 public static void Main() {
  int t, i;
  int[,] matrix = new int[3, 4];  // Declaration of 2D Array
      Console.Writeline ("Sample out put=");
        for( t = 0; t < 3; ++t) {
         for(i = 0; i < 4; ++i) {
          matrix[t,i] = (t*4)+i+1;
          Console.Write(matrix[t,i] + " ");
         }
         Console.WriteLine();
        }
      }
     }
```

Sample output=

```
1 2 3 4
5 6 7 8
9 10 11 12
```


12.1.3 Multi dimensional array

Multi dimensional Array in C sharp is declared by the following Structure.

```
int[,] matrix = { {num1, num2}, {num3, num4}, {num5, num6}, {num7, nu
m8}, {num9, num10} };
```

It means that Multi dimensional array named "mdimen" (user given) is created whose elements are integers and 5 parts are in the array and in each part there resides two elements.

And after creation the array can be used in the following way

```
Console.WriteLine(mdimen [2,0]);
```

This will print value "num5". Because 2 means it resides at 3rd index of the "mdimen" array. And the following 0 means the value of the first element of two elements.

Example: A program that will store value from 1 to 10 in a multidimensional array and displays at the output.

Sample Program=

```
using System;
public class MultidimensionalArrays
{
  public static void Main()
  {
    int[,] mdimen = { {5, 7},{9, 11},{10, 12},{14, 16},{21, 29} };
   for (int i = 0; i < mdimen.GetLength(0); i++)
     {
      for (int j = 0; j < mdimen.GetLength(1); j++)
       {
     Console.WriteLine("mdimen[{0}, {1}] = {2}", i, j, mdimen[i, j]);
       }
     }
  }
}
```

Sample output=

```
Mdimen[0,0]=5
Mdimen[0,1]=7
Mdimen[1,0]=9
Mdimen[1,1]=11
Mdimen[2,0]=10
Mdimen[2,1]=12
Mdimen[3,0]=14
Mdimen[3,1]=16
Mdimen[4,0]=21
Mdimen[4,2]=29
```

12.1.4 Jagged Array

What is Jagged array?

Jagged Array in C sharp is declared by the following Structure.

```
int[][] JArray = new int [number][];
jagged [0] = new int [range];
jagged [1] = new int [range];
jagged [2] = new int [range];
.....
.....
jagged [number-1] = new int [range];
```

It means that Jagged array named "JArray" (user given) is created where each jagged array is further holds a one dimensional array.

And after creation the array can be used in the following way

```
jagged [0][number1] = 2;
Console.WriteLine (jagged [0],[number1]);
```

("number1" is within the range of "number")

This will print value 2 as the array of the specified point contains 2.

Example: A program that will store value from 0 to 4 in a jagged array which consists three arrays and displays the values the output of three separate arrays.

Sample Program=

```
using System;
public class Jagged1 {
 public static void Main() {
  int[][] JArray = new int[3][]; // Demonstrate jagged arrays
  JArray[0] = new int[4];
  JArray[1] = new int[4];
  JArray[2] = new int[4];
  int i;
Console.Writeline("Sample output");
Console.Writeline();
  // store values in first array
  for(i=0; i < 4; i++)
   JArray [0][i] = i+2;
  // store values in second array
  for(i=0; i < 4; i++)
   JArray [1][i] = i+4;
  // store values in third array
  for(i=0; i < 4; i++)
   JArray [2][i] = i+4;
  // display values in first array
  for(i=0; i < 4; i++)
   Console.Write(JArray [0][i] + " ");
  Console.WriteLine();
  // display values in second array
  for(i=0; i < 4; i++)
   Console.Write(JArray[1][i] + " ");
  Console.WriteLine();
  // display values in third array
  for(i=0; i < 4; i++)
   Console.Write(JArray[2][i] + " ");
  Console.WriteLine();
 }
}
```

Sample output=

```
0 3 4 5
0 4 5 6
0 5 6 7
```

12.1.5 Bit Array

What is bit array?

For accessing the individual bits in the bit array, BitArray class implements an indexer. bool[] performs the same thing but An instance of the BitArray class consumes substantially less memory than a corresponding bool[].

The class of BitArray is described below.

```
using System;
class BitArray {
int[] bits;
int length;
public BitArray(int length) {
if (length < 0) throw new ArgumentException();
bits = new int[((length - 1) >> 5) + 1];
this.length = length;
}
public int Length {
get { return length; }
}
public bool this[int index] {
get {
if (index < 0 || index >= length) {
throw new IndexOutOfRangeException();
}
return (bits[index >> 5] & 1 << index) != 0;
}
set {
if (index < 0 || index >= length) {
throw new IndexOutOfRangeException();
}
```



```
if (value) {
bits[index >> 5] |= 1 << index;
}
else {
bits[index >> 5] &= ~(1 << index);
   }
  }
 }
}
```

Any Program can use the BitArray by creating an instance of BitArray.

```
BitArray ba = new BitArray(8);
```

Example: A program that uses the BitArray Class by creating instance of BitArray.

Sample Program=

```
using System;
using System.Collections;
public class BADemo {
 public static void showbits(string rem,
                BitArray bits) {
  Console.WriteLine(rem);
  for(int i=0; i < bits.Count; i++)
   Console.Write("{0, -6} ", bits[i]);
  Console.WriteLine("\n");
}
public static void Main() {

 BitArray ba = new BitArray(8);   // Demonstrate BitArray
 byte[] b = { 67 };
 BitArray ba2 = new BitArray(b);
 showbits("Original contents of BitArray:", ba);
 ba = ba.Not();
 showbits("Contents of BitArray after Not:", ba);
  }
}
```

Sample output=

Original contents of BitArray:
False False False False False False False False
Contents of BitArray after Not:
True True True True True True True True

12.1.6 ArrayList

What is ArrayList?

ArrayList is implemented in a class of C Sharp. Any program can use this ArrayList by creating an instance of this class for holding the elements.

```
ArrayList al = new ArrayList();
```

The elements can be added in the array in the following way,

```
al.Add(number);
```

To get the Array,

```
int[] ia = (int[]) al.ToArray(typeof(int));
```

Now all the elements are in one dimensional integer array. Anyone can access or manipulate the array as like one dimensional Array.

Sample Program=

```
using System;
using System.Collections;
public class ArrayListToArray {
 public static void Main() {
  ArrayList al = new ArrayList();
    Console.WriteLine("Initial number of elements: " +
            al.Count);
  Console.WriteLine();
  // Add elements to the array list.
  al.Add(5);
  al.Add(6);
  al.Add(7);
  al.Add(8);
```



```
  Console.Write("Contents: ");
  foreach(int i in a])
  Console.Write(i + " ");
  Console.WriteLine();
  Console.WriteLine("After adding Number of elements: " +
            al.Count);
  Console.WriteLine();

  // Get the array.
  int[] ia = (int[]) al.ToArray(typeof(int));
  int sum = 0;
  // sum the array
  for(int i=0; i<ia.Length; i++)
   sum += ia[i];
  Console.WriteLine("Sum is: " + sum);
 }
}
```

Sample output=

Initial number of elements: 0
Contents: 5 6 7 8
After adding number of elements: 4
Sum is: 26

12.2 Pointers

A pointer is a data type whose value refers directly to ("points to") another value stored elsewhere in the computer memory using its address. Thus the pointer has an address and contains (as value) an address. Obtaining the value that a pointer refers to is called dereferencing. The dereference operator is *. Pointers in C sharp is declared in the following way,

```
int* p;
```

The value can be assigned in this pointer in the following way,

```
int i = "number";
p=&number;
```

```
p=&i
```

Dereferencing of pointers is done in the following way,

```
int r = *q;
```

Sample Program=

```
using System;
class Pointers
{
public static unsafe void Main()
{
int i = 15;
int* p = &i; // declare pointer and assignment to address of i
int j = 15;
int* q = &j;
bool b2 = (p == q);
Console.WriteLine("b2 = " + b2);
bool b1 = (i == j);
Console.WriteLine("b1 = " + b1);
// dereferencing pointers
int r = *q;
Console.WriteLine("r = " + r);
}
}
```

Sample output=

b_2 = false
b_1 = true
r = 15

12.3 Linked list

LinkedList is implemented in a class of C Sharp. Any program can use this LinkedList by creating an instance of this class for holding the elements.

```
LinkedList list = new LinkedList();
```

The elements can be added in the List in the following way,

```
list.Insert(number/String);
```

To get the elements,

```
List.GetData();
```

To display the elements,

```
List.Display();
```


The linked list class is described below,

```
using System;
class Node {
 internal Object data;
 internal Node next;
 public Node(Object o, Node n){
  data = o;
  next = n;
 }
}
public class LinkedList {
 private Node head;
 private Node previous;
 private Node current;
 public LinkedList() {
  head = null;
  previous = null;
  current = null;
 }
 public bool IsEmpty() {
  return head == null;
 }
 public void Insert(Object o) {
  Node n = new Node(o,current);
  if (previous == null)
   head = n;
  else
   previous.next = n;
  current = n;
 }
 public void Remove() {
  if (head != null){
   if (previous == null)
    head = head.next;
   else
    previous.next = current.next;
   current = current.next;
  }
 }
 public Object GetData(){
```

```
 if (current != null)
   return current.data;
 return null;
}
public bool AtEnd() {
 return current == null;
}
public void Advance(){
 if (!AtEnd()){
   previous = current;
   current = current .next;
 }
}
public void Reset() {
 previous = null;
 current = head;
}
public void Display() {
 Reset();
 if (head != null)
   do {
     Console.WriteLine("   {0}", GetData());
     Advance();
   }while (!AtEnd());
}
```

In this linked list class a node is created using

Node n = **new** Node(o, current);

Where o is an object and current is Node type variable. An internal node can be created by the following syntax,

internal Node next,

This internal node is be used in the following way,

current = current.next;

Example: The program that uses the LinkedList Class and stores the elements and displays accordingly.

Sample Program=

```
Using System;
Using System.Collection;
 public static void Main() {
   LinkedList list = new LinkedList();
   Console.WriteLine("Is Empty {0}",list.IsEmpty());
   list.Insert("AB");
   list.Insert("BC");
   list.Insert("CA");
   Console.WriteLine("The original list is:");
   list.Display();
   list.Reset();
   list.Advance();
   Console.WriteLine("The current element is {0}",list.GetData());
   list.Remove();
   list.Display();
 }
}
```

Sample output=

The Original list is:
CA
BC
AB
The Current element is BC

12.4 Stacks

Stack is implemented in a class of C Sharp. Any program can use this Stack by creating an instance of this class for holding the elements.

Stack stack1 = **new** Stack();

The elements can be added in the Stack in the following way,

```
stack1.Push(number/String);
```

To get the elements,

```
stack1.Pop();
```

To display the top elements,

```
stack1.Top();
```

The Stack class is described below,

```
using System;
public class Stack {
 private int[] data;
 private int size;
 private int top = -1;

 public Stack() {
  size = 10;
  data = new int[size];
 }
 public Stack(int size) {
  this.size = size;
  data = new int[size];
 }
 public bool IsEmpty() {
  return top == -1;
 }
 public bool IsFull() {
  return top == size - 1;
 }
 public void Push(int i){
  if (IsFull())
    throw new ApplicationException("Stack full");
  else
    data[++top] = i;
 }
 public int Pop(){
  if (IsEmpty())
    throw new StackEmptyException("Stack empty");
  else
```



```
    return data[top--];
 }
 public int Top(){
  if (IsEmpty())
    throw new StackEmptyException("Stack empty");
  else
    return data[top];
 }
```

The stack is implemented using Array. The elements are inserted and accessed in the Last in First Out way.

Example: The program that uses the Stack Class and stores the elements and displays accordingly.

Sample Program=

```
Using System;
Using System.Collection;

 public static void Main() {
  try {
    Stack stack1 = new Stack();
    stack1.Push(45);
    stack1.Push(55);
    Console.WriteLine("The top is now {0}", stack1.Top());
    stack1.Push(66);
    Console.WriteLine("Popping stack returns {0}", stack1.Pop());
    Console.WriteLine("Stack 1 has size {0}", stack1.size);
    Console.WriteLine("Stack 1 empty? {0}", stack1.IsEmpty());
    stack1.Pop();
    Console.WriteLine("Throws exception before we get here");
  }catch(Exception e) {
     Console.WriteLine(e);
  }
 }
}
```

```
class StackEmptyException : ApplicationException {
  public StackEmptyException(String message) : base(message) {
  }
}
```

Sample output=

The Top is Now 55
Popping Stack returns 66
Stack 1 has size 10
Stack 1 empty? False
Throws exception before we get here

12.5 Queue

Queue is implemented in a class of C Sharp. Any program can use this Queue by creating an instance of this class for holding the elements.

```
Queue queue1 = new Queue();
```

The elements can be added in the Stack in the following way,

```
queue1.add(number/String);
```

To get the elements,

```
queue1.Remove();
```

To display the front elements,

```
queue1.Head();
```

The Queue class is described below,

```
using System;
public class Queue {
 private int[] data;
 private int size;
 private int front = -1;
 private int back = 0;
 private int count = 0;

 public Queue() {
  size = 10;
  data = new int[size];
 }
```



```
 public Queue(int size) {
  this.size = size;
  data = new int[size];
 }
 public bool IsEmpty() {
  return  count == 0;
 }
 public bool IsFull() {
  return count == size;
 }
 public void Add(int i){
  if (IsFull())
    throw new ApplicationException("Queue full");
  else {
   count++;
   data[back++ % size] = i;
  }
 }
 public int Remove(){
  if (IsEmpty())
    throw new ApplicationException("Queue empty");
  else {
    count--;
    return data[++front % size];
  }
 }
 public int Head(){
  if (IsEmpty()){
    throw new ApplicationException("Queue empty");
  }
  Else
    return data[(front+1) % size];
 }
```

The Queue is implemented using Array. The elements are inserted and accessed in the First in First Out way.

Example: The program that uses the Queue Class and stores the elements and displays accordingly.

Sample Program=

```
Using System;
Using System.Collection;
Public class QueueDemo{
 public static void Main() {
  try {
   Queue q1 = new Queue();
   q1.Add(44);
   q1.Add(55);
   Console.WriteLine("The front is now {0}", q1.Head());
   q1.Add(6);
   Console.WriteLine("Removing from q1 returns {0}", q1.Remove());
   Console.WriteLine("Queue 1 has size {0}", q1.size);
   Console.WriteLine("Queue 1 empty? {0}", q1.IsEmpty());
   q1.Remove();
   Console.WriteLine("Throws exception before we get here");
  }catch(Exception e) {
    Console.WriteLine(e);
  }
 }
}
```

Sample output=

The front is Now 44
Removing from q1 returns 44
Queue 1 has size 10
Queue 1 empty? False
Throws exception before we get here

12.6 Hashing

Hashing is done using Hashtable which is implemented in a class of C Sharp. Any program can use this Hashtable by creating an instance of this class for holding the elements.

```
Hashtable ht = new Hashtable();
```

The elements can be added in the Stack in the following way,

```
ht.add(number/String);
```

Can also be added by using indexer,

```
ht["key"] = "value";
```

Here key indicates based on keys values are hashed or will be found out.
To get the keys,

```
ht.keys();
```

To display the elements,

```
Console.writeline(ht[key]);
```

Example: The program that uses the Hashtable Class for Hashing and stores the elements and displays accordingly.

Sample Program=

```
using System;
using System.Collections;

public class HashtableDemo {
 public static void Main() {
  // Create a hash table.
  Hashtable ht = new Hashtable(); // Demonstrate Hashtable.

  // Add elements to the table
  ht.Add("h", "Dwelling");
  ht.Add("c", "Means of transport");
  ht.Add("b", "Collection of printed words");
  ht.Add("a", "Edible fruit");

  // Can also add by using the indexer.
  ht["t"] = "farm implement";
```

```
    // Get a collection of the keys.
    ICollection c = ht.Keys;
    // Use the keys to obtain the values.
    foreach(string str in c)
      Console.WriteLine(str + ": " + ht[str]);
  }
}
```

Sample output=

b: Collection of printed words
t: farm implement
a: Edible Fruit
h: Dwelling
c: Means of transport

12.7 Sorting

There is a method named *sort* in C Sharp by which sorting can be done in an ascending order. The Syntax is,

Array.Sort (Arr);

For descending order,

Array.Reverse (Arr);

Where "Arr" is the name of the Array to be sorted.

Example: The program that takes randomly some unsorted elements and sorts them in the ascending order.

Sample Program=

```
public class Sort
  {
    static public void Main ()
    {
      DateTime now = DateTime.Now;
      Random rand = new Random ((int) now.Millisecond);
```



```
      int [] Arr = new int [12];
      for (int x = 0; x < Arr.Length; ++x)
      {
        Arr [x] = rand.Next () % 101;
      }
      Console.WriteLine ("The unsorted array elements:");
      foreach (int x in Arr)
      {
        Console.Write (x + " ");
      }
      Array.Sort (Arr);
  Console.WriteLine ("\r\n\r\nThe array sorted in ascending order:");
      foreach (int x in Arr)
      {
        Console.Write (x + " ");
      }
      Array.Reverse (Arr);
 Console.WriteLine ("\r\n\r\nThe array sorted in descending order:");
      foreach (int x in Arr)
      {
        Console.Write (x + " ");
      }
    }
  }
```

Sample output=

The unsorted Array elements:
75 83 16 30 52 43 50 79 96 84 69 82
The array Sorted in ascending order:
16 30 43 50 52 69 75 79 82 83 84 96

12.7.1 Bubble sort

Example: Use of Bubble sort for sorting unsorted elements in the ascending order.

Sample Program=

```
using System;
 public class BubbleSort {
  public static void Main() {
   int[] nums = { 99, -11, 100123, 18, -978,
            5623, 463, -10, 287, 49 };
   int a, b, t;
   int size;
   size = 10; // number of elements to sort
   // display original array
   Console.Write("Original array is:");
   for(int i=0; i < size; i++)
    Console.Write(" " + nums[i]);
   Console.WriteLine();

   // This is the bubble sort.
   for(a=1; a < size; a++)
    for(b=size-1; b >= a; b--) {
     if(nums[b-1] > nums[b]) { // if out of order
      // exchange elements
      t = nums[b-1];
      nums[b-1] = nums[b];
      nums[b] = t;
     }
    }

   // display sorted array
   Console.Write("Sorted array is:");
   for(int i=0; i < size; i++)
    Console.Write(" " + nums[i]);
   Console.WriteLine();
  }
}
```


Sample output=

```
Original Array is:
99 -11 100123 18 -978 5623 463 -10 287 49
Sorted Array is:
-978 -11 -10 18 49 99 287 463 5623 100123
```

12.7.2 Quick sort

Example: Use of Quick sort mechanisms for sorting unsorted elements in the ascending order.

Sample Program=

```
using System;
 class Quicksort {

 // Set up a call to the actual Quicksort method.
  public static void qsort(char[] items) {
   qs(items, 0, items.Length-1);
  }

  // A recursive version of Quicksort for characters.
  static void qs(char[] items, int left, int right)
  {
   int i, j;
   char x, y;
   i = left; j = right;
   x = items[(left+right)/2];
   do {
    while((items[i] < x) && (i < right)) i++;
    while((x < items[j]) && (j > left)) j--;
    if(i <= j) {
     y = items[i];
     items[i] = items[j];
     items[j] = y;
     i++; j--;
    }
   } while(i <= j);
   if(left < j) qs(items, left, j);
   if(i < right) qs(items, i, right);
  }
}
```

The class that uses the quick sort class by creating an instance of that class is described below.

```
public class QSDemo {
 public static void Main() {
  char[] a = { 'e', 'y', 'a', 'r', 'p', 'j', 'i' };
  int i;

  Console.Write("Original array: ");
  for(i=0; i < a.Length; i++)
   Console.Write(a[i]);

  Console.WriteLine();
  // now, sort the array
  Quicksort.qsort(a);

  Console.Write("Sorted array: ");
  for(i=0; i < a.Length; i++)
   Console.Write(a[i]);
 }
}
```

Sample output=

Original Array is: eyarpji
Sorted Array is: aeijpry

12.7.3 Merge sort

Example: Use of Merge sort mechanisms for sorting unsorted elements in the ascending order.

Sample Program=

```
using System;
public class MergeSort {
 public static void Sort (int[] data, int left, int right) {
  if (left < right) {
```



```
   int middle = (left + right)/2;
   Sort(data, left, middle);
   Sort(data, middle + 1, right);
   Merge(data,left, middle, middle+1, right);
  }
 }
public static void Merge(int[] data, int left, int middle, int middle1, int right) {
  int oldPosition = left;
  int size = right - left + 1;
  int[] temp = new int[size];
  int i = 0;
  while (left <= middle && middle1 <= right) {
   if (data[left] <= data[middle1])
    temp[i++] = data[left++];
   else
    temp[i++] = data[middle1++];
  }
  if (left > middle)
   for (int j = middle1; j <= right; j++)
    temp[i++] = data[middle1++];
  else
   for (int j = left; j <= middle; j++)
    temp[i++] = data[left++];
  Array.Copy(temp, 0, data, oldPosition, size);
 }
```

The class that uses the quick sort class by creating an instance of that class is described below.

```
 public static void Main (String[] args) {
  int[] data = new int[]{2,3,1,6,3,98,4,6,4,3,45};
  for (int i = 0; i < data.Length; i++) {
   Console.WriteLine(data[i]);
  }
  Sort(data, 0, data.Length-1);
  for (int i = 0; i < data.Length; i++) {
   Console.WriteLine(data[i]);
  }
 }
}
```

Sample output=

Original Array is: 2 3 1 6 3 98 4 6 4 3 45
Sorted Array is: 1 2 3 3 3 4 4 6 6 45 98

12.7.4 Insertion sort

Example: Use of Insertion sort mechanisms for sorting unsorted elements in the ascending order.

Sample Program=

```
public class InsertionSort {
 public static void InsertNext(int i, int[] item) {
  int current = item[i];
  int j = 0;
  while (current > item[j]) j++;
  for (int k = i; k > j; k--)
   item[k] = item[k-1];
  item[j] = current;
 }
 public static void Sort(int[] item) {
  for (int i = 1; i < item.Length; i++) {
   InsertNext(i, item);
  }
 }
```

The class that uses the Insertion sort class by creating an instance of that class is described below.

```
 public static void Main() {
  int[] item = new int[]{8,1,2,6,3,6,3,6,4,1,2,0};
  for(int i=0; i<item.Length;i++){
    Console.WriteLine("Original Array is:""item[i]);
  Sort(item);
  for(int i=0; i<item.Length;i++){
    Console.WriteLine("Sorted Array is:"item[i]);
  }
 }
}
```


Sample output=

Original Array is: 8 1 2 6 3 6 3 6 4 1 2 0
Sorted Array is: 0 1 1 2 2 3 3 4 6 6 6 8

12.7.5 Sorted list

SortedList is implemented in a class of C Sharp. Any program can use this SortedList by creating an instance of this class for holding the elements.

```
SortedList al = new SortedList();
```

The elements can be added in the array in the following way,

```
al.Add(key,number/string);
```

To get the value by Key,

```
string my = (string) al[key];
```

To get the value by Index,

```
string another = (string) al.GetByIndex(index_number);
```

Now all the elements are in one dimensional integer array. Anyone can access or manipulate the array as like one dimensional Array.

Example: Use of Sorted List for sorting unsorted elements.

Sample Program=

```
using System;
using System.Collections;
public class Example11_8
{
 public static void Main()
 {
  // create a SortedList object
  SortedList mySortedList = new SortedList();

  // add elements containing US state abbreviations and state
  // names to mySortedList using the Add() method

  mySortedList.Add("N", "New York");
  mySortedList.Add("F", "Florida");
  mySortedList.Add("A", "Alabama");
```

```
    mySortedList.Add("W", "Wyoming");
    mySortedList.Add("C", "California");

    // get the state name value for "CA"
    string myState = (string) mySortedList["C"];
    Console.WriteLine("myState = " + myState);

  // get the state name value at index 3 using the GetByIndex() metho d
    string anotherState = (string) mySortedList.GetByIndex(3);
    Console.WriteLine("anotherState = " + anotherState);

    // display the keys for mySortedList using the Keys property
    foreach (string myKey in mySortedList.Keys)
    {
     Console.WriteLine("myKey = " + myKey);
    }
    // display the values for mySortedList using the Values property
    foreach(string myValue in mySortedList.Values)
    {
     Console.WriteLine("myValue = " + myValue);
    }
  }
}
```

Sample output=

```
myState=California
anotherState=New York
myKey=A
myKey=C
myKey=F
myKey=N
myKey=W
myValue=Alabama
myValue=California
myValue=Florida
myValue=New York
myValue=Wyoming
```


12.8 Searching

12.8.1 Binary searching

Example: Use of Binary Search for finding out the index by using some key.

Sample Program=

```
using System;
public class BinarySearch {
 public static int Search (int[] data, int key, int left, int right) {
  if (left <= right) {
   int middle = (left + right)/2;
   if (key == data[middle])
    return middle;
   else if (key < data[middle])
    return Search(data,key,left,middle-1);
   else
    return Search(data,key,middle+1,right);
  }
  return -1;
 }
```

The class that uses the Binary search class by creating an instance of that class is described below.

```
 public static void Main(String[] args) {
  int key;      // the search key
  int index;    // the index returned
  int[] data = new int[10];
 for(int i = 0; i < data.Length; i++)
    data[i] = i;
  key = 5;
  index = Search(data, key, 0, data.Length-1);
  if (index == -1)
   Console.WriteLine("Key {0} not found", key);
  Else
   Console.WriteLine ("Key {0} found at index {1}", key, index);
 }
}
```

Sample output=

```
Key 5 found at index 5
```

12.9 Set

We are familiar with mathematical Set from higher mathematics. The set data structure in C sharp is almost like that where the set contains some elements and elements can be added and subtracted from the set where addition here indicates the union operation of set. The algorithm of set data Structure is given below,

```
using System;
using MyTypes.Set;
namespace MyTypes.Set {
 class Set {
  char[] members; // this array holds the set
  int len; // number of members

  // Construct a null set.
  public Set() {
   len = 0;
  }
  // Construct an empty set of a given size.
  public Set(int size) {
   members = new char[size]; // allocate memory for set
   len = 0; // no members when constructed
  }

  // Construct a set from another set.
  public Set(Set s) {
   members = new char[s.len]; // allocate memory for set
   for(int i=0; i < s.len; i++) members[i] = s[i];
   len = s.len; // number of members
  }

  // Implement read-only Length property.
  public int Length {

   get{
     return len;
   }
  }
```



```
  // Implement read-only indexer.
  public char this[int idx]{
   get {
    if(idx >= 0 & idx < len) return members[idx];
    else return (char)0;
   }
  }
  /* See if an element is in the set.
    Return the index of the element
    or -1 if not found. */
  int find(char ch) {
   int i;
   for(i=0; i < len; i++)
    if(members[i] == ch) return i;
   return -1;
  }

  // Add a unique element to a set.
  public static Set operator +(Set ob, char ch) {
Set newset = new Set(ob.len + 1); // make a new set one element larger

   // copy elements
   for(int i=0; i < ob.len; i++)
    newset.members[i] = ob.members[i];

   // set len
   newset.len = ob.len;

   // see if element already exists
   if(ob.find(ch) == -1) { // if not found, then add
    // add new element to new set
    newset.members[newset.len] = ch;


    newset.len++;
   }
   return newset; // return updated set
  }
```

```
  // Remove an element from the set.
  public static Set operator -(Set ob, char ch) {
    Set newset = new Set();
    int i = ob.find(ch); // i will be -1 if element not found

    // copy and compress the remaining elements
    for(int j=0; j < ob.len; j++)
      if(j != i) newset = newset + ob.members[j];
    return newset;
  }

  // Set union.
  public static Set operator +(Set ob1, Set ob2) {
    Set newset = new Set(ob1); // copy the first set

    // add unique elements from second set
    for(int i=0; i < ob2.len; i++)
      newset = newset + ob2[i];
    return newset; // return updated set
  }

  // Set difference.
  public static Set operator -(Set ob1, Set ob2) {
    Set newset = new Set(ob1); // copy the first set

    // subtract elements from second set
    for(int i=0; i < ob2.len; i++)
      newset = newset - ob2[i];
    return newset; // return updated set
  }
}
}
```


Example: Use of Set data Structure (Addition, Subtraction).

Sample Program=

The class that uses the Binary search class by creating an instance of that class is described below.

```
// Demonstrate the Set class.
public class SetDemo10 {
  public static void Main() {
    // construct 10-element empty Set
    Set s1 = new Set();
    Set s2 = new Set();
    Set s3 = new Set();
    s1 = s1 + 'A';
    s1 = s1 + 'B';
    s1 = s1 + 'C';

    Console.Write("s1 after adding A B C: ");
    for(int i=0; i<s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    s1 = s1 - 'B';
    Console.Write("s1 after s1 = s1 - 'B': ");
    for(int i=0; i<s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    s1 = s1 - 'A';
    Console.Write("s1 after s1 = s1 - 'A': ");
    for(int i=0; i<s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    s1 = s1 - 'C';
    Console.Write("s1 after a1 = s1 - 'C': ");
```

```
  for(int i=0; i<s1.Length; i++)
    Console.Write(s1[i] + " ");
  Console.WriteLine("\n");

  s1 = s1 + 'A';
  s1 = s1 + 'B';
  s1 = s1 + 'C';
  Console.Write("s1 after adding A B C: ");
  for(int i=0; i<s1.Length; i++)
    Console.Write(s1[i] + " ");
  Console.WriteLine();

  Console.Write("s1 is now: ");
  for(int i=0; i<s1.Length; i++)
    Console.Write(s1[i] + " ");
  Console.WriteLine();
 }
}
```

Sample output=

S1 after adding A B C: A B C
S1 after s1=s1 – `B`: A C
S1 after s1=s1 – `A`: C
S1 after s1=s1 – `C`:

S1 after adding A B C: A B C
S1 is now: A B C

12.10 Trees

The Program of Binary search tree is described below:

```
using System;
public sealed class BinarySearchTree<T> : ICollection<T> where T :
IComparable<T>
    {
        internal sealed class Node
```



```
        {
            public T Value;
            public Node Parent;
            public Node Left;
            public Node Right;
            public Node(T Value)
            {
                this.Value = Value;
            }
            public Node(T Value, Node Parent)
            {
                this.Value  = Value;
                this.Parent = Parent;
            }
        }
    public struct AscendingOrderEnumerator : IEnumerator<T>
        {
            private T _Current;
            public T Current
            {
                get
                {
                    return this._Current;
                }
            }
            public bool MoveNext()
            {
                if (this._Next == null)
                {
                    return false;
                }
```

```
                    this._Current = this._Next.Value;

                    if (this._Next.Right == null)
                    {
while((this._Next.Parent != null) &&
        (this._Next == this._Next.Parent.Right))
                        {
                        this._Next = this._Next.Parent;
                        }
                        this._Next = this._Next.Parent;
                    }
                    else
                    {
for(this._Next = this._Next.Right; this._Next.Left != null;
  this._Next= this._Next.Left);
                    }

                    return true;
                }
                internal AscendingOrderEnumerator(Node Node)
                {
                    if (Node != null)
                    {
                        while (Node.Left != null)
                        {
                            Node = Node.Left;
                        }
                    }

                    this._Next          = Node;
                    this._Current       = default(T);
                }
            }
    private Node _Root;
            private int _Count;
```



```
        public int Count
        {
            get
            {
                return this._Count;
            }
        }
        public bool Add(T Item)
        {
            if (Item == null)
            {
                throw new ArgumentNullException();
            }
            if (this._Root == null)
            {
                this._Root = new Node(Item);
            }
            else
            {
                for(Node p = this._Root; ; )
                {
            int Comparer = Item.CompareTo(p.Value);
                    if (Comparer < 0)
                    {
                        if (p.Left != null)
                        {
                            p = p.Left;
                        }
                        else
                        {
                    p.Left = new Node(Item, p);
                            break;
                        }
                    }
```

```
                else if (Comparer > 0)
                {
                        if (p.Right != null)
                        {
                                p = p.Right;
                        }
                        else
                        {
                p.Right = new Node(Item, p);
                                break;
                        }
                }
                else
                {
                        return false;
                }
            }
        }
        this._Count++;
        return true;
    }
    public void Clear()
    {
        this._Root      = null;
        this._Count     = 0;
    }
    public bool Contains(T Item)
    {
        if (Item == null)
        {
            throw new ArgumentNullException();
        }
```



```
        for (Node p = this._Root; p != null; )
        {
        int Comparer = Item.CompareTo(p.Value);
            if (Comparer < 0)
            {
                p = p.Left;
            }
            else if (Comparer > 0)
            {
                p = p.Right;
            }
            else
            {
                return true;
            }
        }
        return false;
    }
    public void CopyTo(T[] Array, int Index)
    {
        if (Array == null)
        {
            throw new ArgumentNullException();
        }
        if ((Index < 0) || (Index >= Array.Length))
        {
            throw new
ArgumentOutOfRangeException();
        }
        if ((Array.Length - Index) < this._Count)
        {
            throw new ArgumentException();
        }
```

```
if (this._Root != null)
{
        Node p = this._Root;

        while (p.Left != null)
        {
                p = p.Left;
        }

        for(;;)
        {
                Array[Index] = p.Value;

                if (p.Right == null)
                {
                        for(;;)
                        {
                        if (p.Parent == null)
                                {

                                        return;
                                }

                        if (p != p.Parent.Right)
                                {

                                        break;
                                }

                                p = p.Parent;
                        }

                        p = p.Parent;
                }
```



```
                                else
                                {
                        for (p = p.Right; p.Left != null; p = p.Left);
                                }

                                Index++;
                        }
                }
        }
        public AscendingOrderEnumerator GetEnumerator()
        {
                return new
AscendingOrderEnumerator(this._Root);
        }
        public bool Remove(T Item)
        {
                if (Item == null)
                {
                        throw new ArgumentNullException();
                }
                for (Node p = this._Root; p != null; )
                {
                int Comparer = Item.CompareTo(p.Value);
                        if (Comparer < 0)
                        {
                                p = p.Left;
                        }
                        else if (Comparer > 0)
                        {
                                p = p.Right;
                        }
                        else
                        {
                if (p.Right == null)
```

```
// Case 1: p has no right child
                {
                    if (p.Left != null)
                    {
                p.Left.Parent = p.Parent;
                    }

                    if (p.Parent == null)
                    {
                    this._Root = p.Left;
                    }
                    else
                    {
                    if (p == p.Parent.Left)
                        {

                     p.Parent.Left = p.Left;
                        }
                        else
                        {
                p.Parent.Right = p.Left;
                        }
                    }
                }
            else if (p.Right.Left == null)

// Case 2: p's right child has no left child
                {
                    if (p.Left != null)
                    {
                p.Left.Parent = p.Right;
                    p.Right.Left = p.Left;
                    }
                p.Right.Parent = p.Parent;
```



```
                    if (p.Parent == null)
                    {
                    this._Root = p.Right;
                    }
                    else
                    {
                    if (p == p.Parent.Left)
                        {
                p.Parent.Left = p.Right;
                        }
                        else
                        {
                p.Parent.Right = p.Right;
                        }
                    }
                }
            else

// Case 3: p's right child has a left child

{
                    Node s = p.Right.Left;
                    while (s.Left != null)
                    {
                        s = s.Left;
                    }

                    if (p.Left != null)
                    {
                    p.Left.Parent = s;
                    s.Left = p.Left;
                    }

                    s.Parent.Left = s.Right;
```

```
                    if (s.Right != null)
                    {
                s.Right.Parent = s.Parent;
                    }
                    p.Right.Parent = s;
                    s.Right = p.Right;

                    s.Parent = p.Parent;

                    if (p.Parent == null)
                    {
                    this._Root = s;
                    }
                    else
                    {
                    if (p == p.Parent.Left)
                            {

                                p.Parent.Left = s;
                                }
                                else
                                {

                            p.Parent.Right = s;
                                }
                    }
                }
                this._Count--;
                return true;
            }
        }
        return false;
    }

    public BinarySearchTree()
```



```
    {
            this._Root      = null;
            this._Count     = 0;
    }
```

12.11 Graph

The Graph class has a number of methods for adding nodes and directed or undirected and weighted or unweighted edges between nodes. The AddNode() method adds a node to the graph, while AddDirectedEdge() and AddUndirectedEdge() allow a weighted or unweighted edge to be associated between two nodes.

In addition to its methods for adding edges, the Graph class has a Contains() method that returns a Boolean indicating if a particular value exists in the graph or not. There is also a Remove() method that deletes a GraphNode and all edges to and from it. The relevant code for the Graph class is shown below,

```
public class Graph<T> : IEnumerable<T> {
private NodeList<T> nodeSet;
public Graph() : this(null) {}
public Graph(NodeList<T> nodeSet)
{ if (nodeSet == null) this.nodeSet = new NodeList<T>(); else this.nodeSet
= nodeSet; } public void AddNode(GraphNode<T> node)
{
 nodeSet.Add(node); } public void AddNode(T value) {

// adds a node to the graph
nodeSet.Add(new GraphNode<T>(value)); } public void
AddDirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
{ from.Neighbors.Add(to); from.Costs.Add(cost); }
public void AddUndirectedEdge(GraphNode<T> from, GraphNode<T> to,
int cost)
```

```
{ from.Neighbors.Add(to); from.Costs.Add(cost); to.Neighbors.Add(from);
to.Costs.Add(cost); }
public bool Contains(T value)
{
return nodeSet.FindByValue(value) != null; }
public bool Remove(T value)
{
// first remove the node from the nodeset
GraphNode<T> nodeToRemove = (GraphNode<T>)
nodeSet.FindByValue(value);
if (nodeToRemove == null)
// node wasn't found return false, otherwise, the node was found
nodeSet.Remove(nodeToRemove); // enumerate through each node in the
nodeSet, removing edges to this node
foreach (GraphNode<T> gnode in nodeSet)
{ int index = gnode.Neighbors.IndexOf(nodeToRemove);
if (index != -1)
{
// remove the reference to the node and associated cost
gnode.Neighbors.RemoveAt(index);
gnode.Costs.RemoveAt(index);
}
} return true;
}
public NodeList<T> Nodes { get { return nodeSet; } }
public int Count { get { return nodeSet.Count; } }
}
```


Summary:

In this chapter we have shown different types of operations on array, linked list, stack, queue, tree etc. These operations are depicted in algorithms or programs using C Sharp.

- One dimensional array in C sharp is declared as int[] array_name=new int [length_of_array]. Two dimensional array is declared as int [,] array_name=new int [row_number, column_number]. Multi dimensional array is declared as int [,] array_name ={{numbe_1,number_2}, {number_3,number_4},.....,{number_n-1, number_n}}. Jagged array consist several arrays. It is declared as int [] [] array_name=new int [number_of_arrays] []. And lastly for accessing individual bits, Bit array is used. The syntax for creating a Bit array is BitArray array_name=new BitArray(number_of_bits).

- An ArrayList is a special kind of data structure. It stores information in an array that can be dynamically resized. This data structure in C Sharp contains methods that assist the programmer in accessing and storing data within the ArrayList.

- Pointers in C sharp is declared as int* p; In C sharp, Linked List is implemented in a class. In Programs Linked List can be used by creating an instance of this class.

- A program can use stack by creating an instance of stack class and in similar way the program can use queue by creating an instance of queue class.

- In C Sharp, Hashing needs Hash table. Hash table is implemented in a class.

- Sorting can be done by using a method named "sort" in C sharp. Bubble sort, quick sort, Insertion sort etc. class can be implemented. In Any program, bubble sort mechanism can be used by creating an instance of Bubble sort class. In the same way, in any program, quick sort mechanism can be used by creating an instance of quick sort class and can use Insertion sort by creating an instance of Insertion sort class. SortedList stores information in an array in the appropriate order. This data structure in C Sharp contains methods that assist the programmer in accessing and storing data within the SortedList.

- Binary search class is defined in C sharp. So, any program may use binary search class for searching.

- The set data structure in C sharp contains some elements. Elements can be added and subtracted from the set. Addition indicates the union operation of set. Binary search tree is a special kind of tree like data structure.

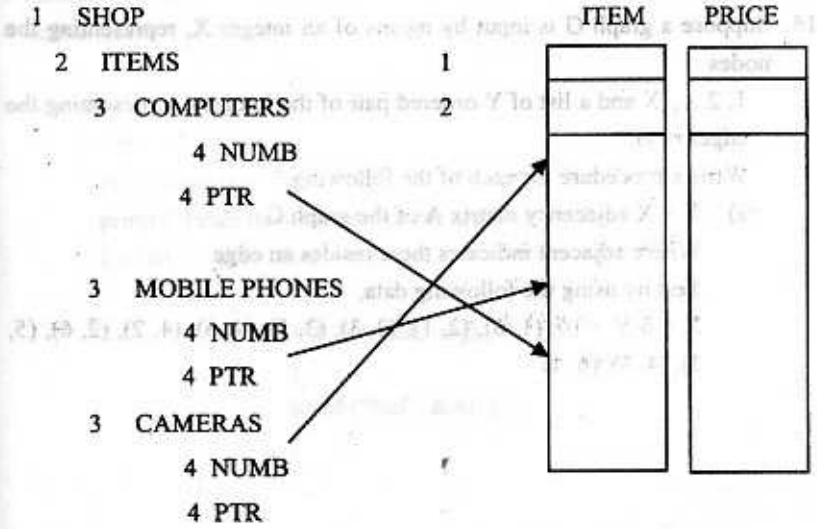
- There exist several methods of Graph class in C sharp. AddNode() method adds a node to the graph, while AddDirectedEdge() and AddUndirectedEdge() allow a weighted or unweighted edge to be associated between two nodes. It contains some other methods. The methods help to represent the nodes in a graph.

Questions:

1. What are the different kinds of data structures in C sharp ? Give short description to each.
2. Suppose, SAMPLE is a linear array with n numbers. Write a procedure which finds the average of the values. The average of the values $x_1, x_2, x_3, x_4, \ldots\ldots, x_n$ is defined by,

$$\text{Avg} = (x_1, x_2, x_3, x_4, \ldots\ldots, x_n)/n$$

3. Each batsman in a cricket team of 11 players plays 5 games in which scores range between 0 and 50. Suppose the scores are stored in a 11 × 5 array named SCORE. Write a module which
   a) Finds the average score for each game
   b) Finds the average of the player's four highest scores for each player.
   c) Finds the number of players who will be eliminated for the next match, i.e., whose average score is less than 10.
4. What is the difference between array data structure in C and C Sharp ?
5. What is the difference between Pointers in C and C Sharp ?
6. A shop keeps track of the serial number and price of its items in arrays ITEM and PRICE, respectively. In addition, it uses the data structure stated in figure, which combines a record structure with pointer variables. Computers, Mobile phones, Cameras are listed together in ITEM. The variables NUMB and PTR under USED indicates, respectively, the number and location of different items.
   a) How does anyone index the location of the list of Computers in ITEM ?
   Write a procedure to print the serial number of all Mobile phones.



7. What is linked list? How linked list class is implemented in C sharp?
8. Given an integer I, write a procedure which deletes the I_{th} element from a linked list.
9. Write a program which adds a user given item at the sorted list.
10. Consider the following stack where stack is allocated N=3 cells,

    STACK: A, B, _

    Describe the stack as the following operations take place,

    a) PUSH (STACK, C) b) POP(STACK, ITEM)

    c) POP(STACK, ITEM)
11. Consider a priority queue which contains 6 elements. Write a program which deletes a user specified element in the queue.
12. How sorting and searching are performed in C Sharp ? Write a program that can sort elements and apply binary search to find out a user specified element and remove that from the list.
13. What is hash table ? How Hashing uses hash table for its processing ?
14. Suppose the following 7 numbers are inserted into an empty binary search tree T,

    50, 22, 33, 44, 35, 60, 77

    Draw the tree T.
15. Suppose a graph G is input by means of an integer X, representing the nodes

    1, 2,…,X and a list of Y ordered pair of the integers, representing the edges of G.

    Write a procedure for each of the following,

    a) X × X adjacency matrix A of the graph G

    Where adjacent indicates there resides an edge

    Test by using the following data,

    X = 6 Y = 10; (1, 6), (2, 1), (2, 3), (3, 5), (4, 5), (4, 2), (2, 6), (5, 3), (4, 3), (6, 4)

APPENDIX-C
PRACTICAL ISSUES

OBJECTIVES:

In this chapter we have shown codes of some programs related to the algorithms written in different chapters. The programs are written in C++ programming language. However no program written in C# and JAVA has been shown in this chapter, as these are already been stated in Appendix A (JAVA) and Appendix B (C#).

13.1 ARRAY

Find out the summation of diagonal elements of a two dimensional array

The following program is based on the Algorithm 2.6 (see Chapter 2).

Program 1:

```
# include <stdio.h>
# include <conio.h>
# define size 3

void main ( )
        clrscr ( );
        int A[size][size], sum = 0;
        printf ("Enter the elements of the array:\t");
        for (int i = 0; i < size; ++i)
                {
                for ( int j = 0; j < size; ++j)
                        {
                        scanf ("%d", &A[i][j]);
                        }
                }

        for (i = 0; i < size; ++i)
```

```
        {
        for (int j = 0; i <size; ++j)
                {
        if (i = = j || (i + j) = = (size – n)) sum = sum + A[i][j];
                }
        }
    printf ("\n\nSummation of diagonal elements: \t%d", sum);
    getch ( );
    }
```

Program Output:

```
Enter the elements of the array:    12 45 56 88 12 45 36 25 84
Summation of diagonal elements:  200
```

Problem: There are 40 students in a class and 4 class tests for each student. Find out the average of the best 3 class tests for each student.

This program is written based on Algorithm 2.7 (see Chapter 2).

Program 2:

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

void main ( )
        {
        clrscr ( );
        int marks[40][4];
        float avg_mrk[4];
        int sum, min_mrk;

        printf ("Enter class test martks:\t");
        for (int i = 0; i < 40; ++i)
```



```
                {
                printf ("Serial No %d: \t", i + 1);
                for (int j = 0; j < 4; j ++)
                        {
                        scanf ("%d", &marks[i][j]);
                        }
                sum = 0;
                min_mrk = marks [i][0];
                for (j = 0; j < 4; ++j)
                        {
                        sum = sum + marks[i][j];
                        if min_mrk > marks [i][j]) min_mrk = marks[i][j];
                        }
                avg_mrk[i] = float (sum – min_mrk) / 3;
                }
        printf ("Average marks:\n");
        for (i = 0; i < 4; ++i)
                {
                printf ("Serial No %d:\t", i + 1);
                printf ("%.2f \n", avg_mrk[i]);
                }
        getch ( );
        }
```

Program Output:

```
Enter class test marks:
Serial No 1:      25 12 28 27
Serial No 2:      22 25 19 25
Serial No 3:      24 27 26 0
.....
Average Marks:
Serial No 1:      26.7
Serial No 2:      24.0
Serial No 3:      25.7
```

13.2 LINKED LIST

13.2.1 Creation of linked list

This program is to create a linear linked list. Here we shall write a program in C++ following the algorithm 4.1 of chapter four. After the declaration of the node, here we declare the class and use constructor function to create an empty linked list. In this program we have three functions. First is ***newnode ( )***, which creates a new node. The second function is for making link and which is named as ***link ( )***. The last one is ***showdata ( )***, which displays the data of the linked list. In ***main ( )***, we declare the necessary variables and call the functions.

Program 3: Code of program to create a linked list

```
struct node
        {
        int data;
        node *next;
        };
class linklist
        {
        node *list, *nptr, *tptr;
        public:
        linklist ( )                          // Constructor
                {
                list = NULL;
                }
        void newnode (int item);
        void link ( );
        void showdata ( );
        };
void  linklist::newnode (int item)
        {
        nptr = new (node);
        nptr→data = item;
        nptr→next = NULL;
        }
```



```
void linklist :: link ( )
        {
        if (list == NULL)
                {
                list = nptr;
                tptr = nptr;
                }
        else    {
                tptr→next = nptr;
                tptr = nptr;
                }
        }

void linklist :: showdata ( )
        {
        node *curptr;
        curptr = list;
        while (curptr != NULL)
                {
                cout << " " << curptr→data;
                curptr = curptr→next;
                }
        }
int main ( )
        {
        clrscr ( );

        int n, d;
        linklist mylist;

        cout << "\nHow many nodes you have?\t";
        cin >> n;
        cout << "\nEnter data for nodes (separated by space):\t";

        for (int i = 0; i < n; ++i)
                {
                cin >> d;
```

```
            mylist.newnode (d);
            mylist.link ( );
            }

        cout << "\nData in the list:\t";
        mylist.showdata ( );

        getch ( );
        return 0;
        }
```

Program output:

```
How many nodes you have?      5

Enter data for nodes:  23 56 65 12 6

Data in the list:      23 56 65 12 6
```

13.2.2 Search a node from a linked list

Here we shall write a program in C++ following the algorithm 4.2 of chapter four. To perform searching we have to create the linked list first. So, we shall use here all the functions that are used in program 1. To perform searching here, we shall use a function named *search ( )*.

Program 4: Code of program for searching a particular node from a linked list

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

struct node
        {
        int data;
        node *next;
        };
```



```
class linklist
        {
        node *list, *nptr, *tptr;
        public:
        linklist ( )
                {
                list = NULL;
                }
        void newnode (int item);
        void link ( );
        void showdata ( );
        void search (int item);
        };

void  linklist::newnode (int item)
        {
        nptr = new (node);
        nptr→data = item;
        nptr→next = NULL;
        }

void linklist :: link ( )
        {
        if (list == NULL)
                {
                list = nptr;
                tptr = nptr;
                }
        else    {

                tptr→next = nptr;
                tptr = nptr;
                }
        }
```

```
void linklist::search (int item)
        {
        tptr = list;
        while (tptr != NULL)
                {
                if (tptr→data = = item)
                        {
                        cout << "Data found!";
                        break;
                        }
                tptr = tptr→next;
                if (tptr = = NULL) cout << "Data is not in the list";
                }
        }
void linklist :: showdata ( )
        {
        node *curptr;
        curptr = list;
        while (curptr != NULL)
                {
                cout <<" "<< curptr→data;
                curptr = curptr→next;
                }
        }
int main ( )
        {
        clrscr ( );
        int n, d;

        linklist lis;
        cout << "\nHow many nodes you have?\t";
        cin >> n;
```



```
        cout << "\nEnter data for nodes:\t";
        for (int i = 0; i < n; ++i)
                {
                cin >> d;
                lis.newnode (d);
                lis.link ( );
                }
        cout << "\nData in the list:\t";
        lis.showdata ( );

        int item;
        cout << "\n\nEnter the data to be found\t";
        cin >> item;
        cout << "\nSearch Result:";
        lis.search (item);

        getch ( );
        return 0;
        }
```

Program output:

```
How many nodes you have?      5

Enter data for nodes:  23 56 65 12 6

Data in the list:      23 56 65 12 6

Enter the item to be found:   6

Search Result:    Data found!
```

13.2.3 Delete a particular node from a linked list

This program is according to the algorithm 4.4 of the chapter four. Here deletion is possible from any part of the list *i.e.*, the first node, the last node and a node between first node and last node.

Program 5: Code of program for deleting a node from a linked list

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

struct node
        {
        int data;
        node *next;
        };
class linklist
        {
        node *list, *nptr, *tptr;
        public:
        linklist ( )
                {
                list = NULL;
                }
        void newnode (int item);
        void link ( );
        void showdata ( );
        void deletion (int item);
        };
void  linklist::newnode (int item)
        {
        nptr = new (node);
        nptr→data = item;
        nptr→next = NULL;
        }
```



```
void linklist :: link ( )
        {
        if (list == NULL)
                {
                list = nptr;
                tptr = nptr;
                }
        else    {
                tptr→next = nptr;
                tptr = nptr;
                }
        }

void linklist :: showdata ( )
        {
        node *curptr;
        curptr = list;
        while (curptr != NULL)
                {
                cout <<" "<< curptr→data;
                curptr = curptr→next;
                }
        }

void linklist::deletion (int item)
        {
        node *pptr;
        tptr = list;
        if (list→data != item)
                {
                while (tptr→data != item)
                        {
                        if (tptr = = NULL)
                                {
```

```
                    cout << "\nItem is not found in the list!";
                    break;
                    }
                pptr = tptr;
                tptr = tptr→next;
                }
            pptr→next = tptr→next;
            delete (tptr);
            }
        else    {
            list = list→next;
            delete (tptr);
            }
    }
int main ( )
 {
 clrscr ( );
 int n, d;
 linklist lis;
 cout << "\nHow many nodes you have?\t";
 cin >> n;
 cout << "\nEnter data for nodes:\t";
 for (int i = 0; i < n; ++i)
     {
     cin >> d;
     lis.newnode (d);
     lis.link ( );
     }
 cout << "\nData in the list:\t";
 lis.showdata ( );
 char ans;
 cout << "\n\nDo you want to delete a node?\t";
 cin >> ans;
```



```
 if (ans= = 'y' || ans = = 'Y')
     {
     int x;
     cout << "\n\nEnter the node value to be deleted:\t";
     cin >> x;
     lis.deletion (x);
     cout << "\n\nData in the list:\t";
     lis.showdata ( );
     }
 getch ( );
 return 0;
 }
```

Program output:

```
How many nodes you have?      5
Enter data for nodes:  23 56 65 12 6
Data in the list:      23 56 65 12 6
Enter the node value to be deleted:    65
Updated list: 23 56 12 6
```

13.2.4 Arrange data of a linked list

To sort the data of the nodes in a linked list the following function can be effectively used. Here, data is to be sorted in ascending order. This program is based on the principle of algorithm 4.5 of chapter four.

Program 6: Code of program for arranging data of a linked list

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

```
struct node
        {
        int data;
        node *next;
        };

class linklist
        {
        node *list, *nptr, *tptr;
        public:
        linklist ( )
                {
                list = NULL;
                }
        void newnode (int item);
        void link ( );
        void showdata ( );
        void sort (int item);
        };

void  linklist::newnode (int item)
        {
        nptr = new (node);
        nptr→data = item;
        nptr→next = NULL;
        }

void linklist :: link ( )
        {
        if (list == NULL)
                {
                list = nptr;
                tptr = nptr;
                }
```



```
        else    {
                tptr→next = nptr;
                tptr = nptr;
                }
        }
void linklist :: showdata ( )
        {
        node *curptr;
        curptr = list;
        while (curptr != NULL)
                {
                cout <<" "<< curptr→data;
                curptr = curptr→next;
                }
        }
void linklist::sort ( )
        {
        node *pptr, *fptr;
        pptr = list;
        while (pptr != NULL)
                {
                fptr = pptr→next;
                while (fptr != NULL)
                        {
                        if (pptr →data > fptr→data)
                                {
                                //interchange (pptr→data, fptr→data)
                                int temp;
                                temp = pptr→data;
                                pptr→data = fptr→data;
                                fptr→data = temp;
                                }
                        fptr = fptr→next;
                        }
```

```
            pptr = pptr→next;
            }
    }
int main ( )
    {
    clrscr ( );
    int n, d;
    linklist lis;
    cout << "\nHow many nodes you have?\t";
    cin >> n;
    cout << "\nEnter data for nodes:\t";
    for (int i = 0; i < n; ++i)
            {
            cin >> d;
            lis.newnode (d);
            lis.link ( );
            }
    cout << "\nData in the list: [before sort]\t";
    lis.showdata ( );
    lis.sort ( );
    cout << "\n\nData in the list [after sort]:\t";
    lis.showdata ( );
    getch ( );
    return 0;
    }
```

Program output:

```
How many nodes you have?      15
Enter data for nodes:  12 65 2 1 89 45 65 15 19 29 32 54 18 9 3
Data in the list:      12 65 2 1 89 45 65 15 19 29 32 54 18 9 3
Do you want to sort the list? Y
Sorted List:  1 2 3 9 12 15 18 19 29 32 45 54 65 65 89
```


13.3 DOUBLE LINKED LIST

13.3.1 Creation of a double linked list

This program is according to the algorithm 4.6 of chapter 4.

Program 7: Program Codes for creating a double linked list

```
struct node
    {
    node *back;
    int data;
    node *next;
    };

class linklist
    {
    node *list, *nptr, *tptr;
    public:
    linklist ( )
            {
            list = NULL;
            }

    void newnode (int item);
    void link ( );
    void showdata ( );
    };

void  linklist::newnode (int item)
    {
    nptr = new (node);
    nptr→back = NULL;
    nptr→data = item;
    nptr→next = NULL;
    }
```

```
void linklist :: link ( )
        {
        if (list = = NULL)
                {
                list = nptr;
                tptr = nptr;
                }
        else    {
                tptr→next = nptr;
                nptr→back = tptr;
                tptr = nptr;
                }
        }

void linklist :: showdata ( )
        {
        node *curptr;
        curptr = list;
        while (curptr != NULL)
                {
                cout << " " << curptr→data;
                curptr = curptr→next;
                }
        }
int main ( )
        {
        clrscr ( );

        int n, d;
        linklist mylist;

        cout << "\nHow many nodes you have?\t";
        cin >> n;
        cout << "\nEnter data for nodes (separated by space):\t";
```



```
        for (int i = 0; i < n; ++i)
                {
                cin >> d;
                mylist.newnode (d);
                mylist.link ( );
                }

        cout << "\nData in the list:\t";
        mylist.showdata ( );

        getch ( );
        return 0;
        }
```

Program output:

```
How many nodes you have?      5

Enter data for nodes:  236 652 125 653 265

Data in the list:      236 652 125 653 265
```

13.4 STACK

13.4.1 Creation of a linked based stack

This algorithm is according to the algorithm 5.3 of the chapter five.

Program 8: Code of program for creating a linked based stack

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

struct node
        {
        int data;
        node *next;
        };
```

```
class stack
        {
        node *top, *newptr;
        public:
        stack ( )
                {
                top = NULL;
                }
        void newnode (int item);
        void push ( );
        void show ( );
        };
void stack::newnode (int item)
        {
        newptr = new node;
        newptr→data = item;
        newptr→next = NULL;
        }
void stack::push ( )
        {
        if (top = = NULL) top = newptr;
        else    {
                newptr→next = top;
                top = newptr;
                }
        }
void queue::show ( )
        {
        node *curptr;
        curptr = fptr;
        while (curptr != NULL)
                {
                cout << "  " << curptr→data;
```



```
                curptr = curptr→next;
                }
        }

int main ( )
        {
        clrscr ( );
        int d, n;
        stack st;
        cout << "\n Enter how many nodes will be added: ";
        cin >> n;
        cout << "\n Enter data for the nodes with space: ";
        for (int i = 0; i < n; ++i)
                {
                cin >> d;
                st.newnode (d);
                st.push ( );
                }
        cout << "\n Data in the stack:";
        st.show ( );
        getch ( );
        return 0;
        }
```

Program output:

```
Enter how many nodes will be added:        5

Enter data for the nodes with space:   23 56 65 12 6

Data in the stack:      23 56 65 12 6
```

13.5 QUEUE

13.5.1 Creation of a linked based queue

This program is coding according to the principle of algorithm 6.3 in chapter six.

Program 9: Code of program for creating a linked based queue

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

struct node
        {
        int data;
        node* next;
        };

class queue
        {
        node *fptr, *rptr, *newptr;
        public:
        queue ( )
                {
                fptr = NULL;
                rptr = NULL;
                }
        void newnode (int item);
        void add ( );
        void delet ( );
        void show ( );
        };
void queue::newnode (int item)
        {
        newptr = new node;
        newptr→data = item;
        newptr→next = NULL;
        }
```



```
void queue::add ( )
        {
        if (rptr = = NULL)
                {
                rptr = newptr;
                fptr = newptr;
                }
        else    {
                rptr→next = newptr;
                rptr = newptr;
                }
        };

void queue::show ( )
        {
        node *curptr;
        curptr = fptr;
        while (curptr != NULL)
                {
                cout << "  " << curptr→data;
                curptr = curptr→next;
                }
        }

int main ( )
        {
        clrscr ( );
        int d, n;
        queue qt;
        cout << "\n Enter how many nodes will be added: ";
        cin >> n;
        cout << "\n Enter data for the nodes with space: ";
        for (int i = 0; i < n; ++i)
```

```
        {
        cin >> d;
        qt.newnode (d);
        qt.add ( );
        }
    cout << "\n Data in the queue:";
    qt.show ( );
    getch ( );
    return 0;
    }
```

Program output:

```
Enter how many nodes will be added:        5

Enter data for the nodes with space:   23 56 65 12 6

Data in the queue:      23 56 65 12 6
```

13.6 TREE

13.6.1 Creation of a tree (Binary Search Tree)

This is an example of linked tree as BST in which some nodes will be added to the tree and display the data using inorder method.

Program 10: Code of program for creating a linked based tree

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

struct node
{
 node* lchild;
 int data;
 node* rchild;
};
```



```
class tree
{
 node *parentptr,*temptr,*newptr;
 public:
 node * treeptr;

 tree()  //constructor;
  {      treeptr=NULL;
  }
  void newnode(int item);
  void insert();
  void inorder(node* curptr);
};

void tree::newnode(int item)
{              //Separate function for creation
  newptr=new node;    //of new node
  newptr->lchild=NULL;
  newptr->data=item;
  newptr->rchild=NULL;
}

void tree::insert() // Function for establishing linkage
{              //between the existing linkedlist (if any)
 if(treeptr==NULL)    //and new node
 {
          treeptr=newptr;
          }
 else
 {
          temptr=treeptr;
          while(temptr!=NULL)
          {     parentptr=temptr;
            if(temptr->data >= newptr->data)
```

```
                temptr=temptr->lchild;
            else
                temptr=temptr->rchild;
        }

        if(parentptr->data >= newptr->data)
                parentptr->lchild=newptr;
        else
                parentptr->rchild=newptr;

    }

}

void tree::inorder(node* curptr)
{
  if(curptr!=NULL)
  {   inorder(curptr->lchild);
      cout<<" "<<curptr->data;
      inorder(curptr->rchild);
  }

}

main()
{
  clrscr();
  int d,n;
  tree tr;

  cout<<"\n Enter how many nodes will be added: ";
  cin>>n;
  cout<<"\n Enter data for the nodes with space: ";
  for(int i=0;i<n;++i)
```



```
  {
      cin>>d;
      tr.newnode(d);
      tr.insert();
  }
  cout<<"\n The data in tree:";
  tr.inorder(tr.treeptr);
  getch();
  return 0;
}
```

13.7 GRAPH

This program is for the depth first traversal of a graph. In this program we have a class ***graph ()***, which creates the graph and three functions. First is ***read_graph ()***, which reads the vertices and their adjacent nodes of a graph. The second and the third functions are ***dfs*** (int) and ***dfs*** (). The first of these two, traverses the graph for a single node and the second one traverses the entire graph. The last function ***ftraverse*** () displays the data for the depth first traversal. In ***main ()***, we declare the necessary variables and call the functions.

Program 11: Code for Depth First Search

```
#include<iostream.h>
int visit[100];
class graph
{
private:
int n;
graph*next;
public:
graph* read_graph(graph*);
void dfs(int); //dfs for a single node
void dfs(); //dfs of the entire graph
void ftraverse(graph*);
}*g[100];
```

```
graph* graph::read_graph(graph*head)
{
int x;
graph*last;
head=last=NULL;
cout<<"Enter adjecent node ,-1 to stop:\n";
cin>>x;
while(x!=-1)
{
graph*NEW;
NEW=new graph;
NEW->n=x;
NEW->next=NULL;
if(head==NULL)
head=NEW;
else
last->next=NEW;
last=NEW;

cout<<"Enter adjecent node ,-1 to stop:\n";
cin>>x;
}
return head;
}

void graph::ftraverse(graph*h)
{
while(h!=NULL)
{
cout<<h->n<<"->";
h=h->next;
}
cout<<"NULL"<<endl;
}
```



```
void graph::dfs(int x)
{
cout<<"node "<<x<<" is visited\n";
visit[x]=1;

graph *p;
p=g[x];
while(p!=NULL)
{
int x1=p->n;
if(visit[x1]==0)
{
cout<<"from node "<<x;
dfs(x1);
}
p=p->next;
}
}

void graph::dfs()
{
int i;
cout<<"Enter the no of nodes ::";
cin>>n;
for(i=1;i<=n;i++)
g[i]=NULL;

for(i=1;i<=n;i++)
{
cout<<"Enter the adjacent nodes to node no. "<<i<<endl;
cout<<"**************************************\n";
g[i]=read_graph(g[i]);
}
```

```
//display the graph
cout<<"\n\nThe entered graph is ::\n";
for(i=1;i<=n;i++)
{
cout<<" < "<<i<<" > ::";
ftraverse(g[i]);
}

for(i=1;i<=n;i++)
visit[i]=0; //mark all nodes as unvisited

cout<<"\nEnter the start vertex ::";
int start;
cin>>start;
cout<<"\nThe dfs for the above graph is ::\n";
dfs(start);
}
int main()
{
graph obj;
obj.dfs();
return 0;
}
```


Sample Input:

```
Enter the no of nodes :: 5
Enter the adjacent nodes to node no. 1
Enter adjecent node ,-1 to stop:
2
Enter adjecent node ,-1 to stop:
5 -1
Enter the adjacent nodes to node no. 2
Enter adjecent node ,-1 to stop:
1
Enter adjecent node ,-1 to stop:
3
Enter adjecent node ,-1 to stop:
4
Enter adjecent node ,-1 to stop:
5 -1
Enter the adjacent nodes to node no. 3
Enter adjecent node ,-1 to stop:
2
Enter adjecent node ,-1 to stop:
4 -1
Enter the adjacent nodes to node no. 4
Enter adjecent node ,-1 to stop:
2
Enter adjecent node ,-1 to stop:
3 -1
Enter the adjacent nodes to node no. 5
Enter adjecent node ,-1 to stop:
1
Enter adjecent node ,-1 to stop:
2 -1
The entered graph is ::
< 1 > :: 2 -> 5 -> NULL
< 2 > :: 1 -> 3-> 5 -> 4-> NULL
< 3 > :: 2 -> 4 -> NULL
< 4 > :: 2 -> 3 -> NULL
< 5 > :: 1 -> 2 -> NULL

Enter the start vertex: 1
```

Sample Output

The dfs for the above graph is ::
node 1 is visited
from node1 node 2 is visited
from node2 node 3 is visited
from node3 node 4 is visited
from node2 node 5 is visited

13.8 SORTING

Generate random numbers (integers) and sort them using Selection Sort Method

This program is written based on the Algorithm 9.3 (see Chapter 9). Here, randomize () and random () functions are used to produce random numbers upto 999. Neverthless, malloc () is used for allocation memory space in the primry memory for Array A. For dynamic memory allocation pointer is used in this program.

Program 12:

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

void main ( )
        {
        clrscr ( );
        int n, *A, temp;

        printf ("Insert how many integers you need:\t");
        scanf ("%d", &n);

        A = (int *) malloc (n * sizeof (int));
        randomize ( );
        for (int i = 0; i < n; ++i)
```



```
        {
        A[i] = random (1000);
        }

printf ("\n\nRandom numbers in the array:\n");
for (i = 0; i < n; ++i)
        {
        printf ("%d\t", A[i]);
        }

for (i = 0; i < n; ++i)
        {
        int small_index = i;
        for (int j = 0; j < n; ++j)
                {
                if (A[j] < A[small_index]) small_index = j;
                }
        temp = A[i];
        A[i] = A[small_index];
        A[small_index] = temp;
        }

printf ("\nSorte list:\n");
for (i = 0; i < n; ++i)
        {
        printf ("%d\t", A[i]);
        }
getch();
}
```

Program Output:

```
Insert how many integers you need:        10

Random numbers in the array:
325   925   104   88   142   32   887   113   235   653

Sorted list:
32    88    104   113  142   235  325   653   887   925
```

13.9 SEARCHING

Program 13: Code for searching using Binary Search algorithm

```
#include<stdio.h>
#include<conio.h>
int x,no_of_element,element[100];

int binary_search()
{
 int mid,first,last;
 first=0;
 last=no_of_elements-1;
 while(first<=last)
 {
        mid=(first+last)/2;
        if(x<element[mid])
                last=mid-1;
        else if(x>element[mid])
                first=mid+1;
        else return mid;
 }
return -1;
}
```



```
void main()
{
 int i,found;
 printf("\nEnter the number of  elements::");
 scanf("%d",& no_of_element);
 printf("\nEnter the elements(sorted list)::");
 for(i=0;i<no_of_elements;i++)
        scanf("%d",& element[i]);
 printf("\nEnter the element to be searched in the list::");
 scanf("%d",&x);
 found=binary_search();
 if(found==-1)
        printf("\n%d does not exists in the list",x);
 else
        printf("\n%d is element number %d in the list",x,found+1);
 getch();
}
```

Sample Input: Enter the number of elements:: 10
Enter the elements(sorted list):: -15 -10 5 12 29 110 114 115 250 290
Enter the element to be searched in the list:: 12
Sample Output: 12 is element number 4 in the list

13.10 HASHING

Program 14: Chaining Method

```
#include<iostream.h>
#include<conio.h>
#define NULL 0

struct node
{
int keyval;
node *next;
};
```

```
class linklist
{
node *table[10],*nptr,*tptr;
public:
    linklist()
    {
    for(int i=0;i<10;i++)
    table[i]=NULL;
    }
void newnode(int item);
void link();
void show();
};

void linklist::newnode(int item)
{
        nptr=new node();
        nptr->keyval=item;
        nptr->next=NULL;
}
void linklist::link()
{
        int ind;
        ind=nptr->keyval%10;
                if(table[ind]==NULL)
                table[ind]=nptr;
                else
                {
                        tptr=table[ind];
                                while(tptr->next!=NULL)
                                {
                                tptr=tptr->next;
                                }
                        tptr->next=nptr;
                }
}
void linklist::show()
{
node *cptr;
```



```
        for(int i=0;i<10;i++)
        {
                cptr=table[i];
                        while(cptr!=NULL)
                        {
                        cout<<" "<<cptr->keyval;
                        cptr=cptr->next;
                        }
                cout<<"\n\n";
        }
}
void main()
{
clrscr();
int n,d;
linklist list;
  cout<<"\nHow many data:\t";
cin>>n;
cout<<"\nEnter data for nodes:\t";
        for(int i=0;i<n;i++)
        {
        cin>>d;
        list.newnode(d);
        list.link();
        }
cout<<"\nData in the list:\t";
list.show();
getch();
}
```

Sample Input: How many data: 5
Enter data for nodes: 11 12 21 22 23

Sample Output: Data in the list:
11 21
12 22
23

BIBLIOGRAPHY

1. Fundamentals of Data structure in C++, E. Horowitz, S. Sahni, D. Mehta, Galgotia Publication Pvt. Ltd, New Delhi.
2. The Art of Computer Programming, Volume 1, Fundamental Algorithms, D.E. Knuth, Addison-Wesley, Publishing Company, 2001.
3. The Art of Computer Programming, Volume 3, Sorting and Searching, D.E. Knuth, Addison-Wesley, Publishing Company, 2001.
4. Fundamentals of Computer Algorithms, E. Horowitz, S, Sahni, S. Rajasekharan, Galgotia Publishing Pvt. Ltd, New Delhi.
5. Data Structures, Edward M. Reingold, Wilfred J. Hansen, CBS Publication & Distributions, 1983.
6. Theory and Problems of Data Structures, Seymour Lipschutz, Schaum's Outline Series, McGraw-Hill Book Company, 1986.
7. Introduction of Algorithms, Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, Prentice-Hall of India Private Limited, 1995.
8. Data Structures and Algorithms in Java, Robert Lafore, Techmedia, 2003.
9. Data Structures and Program Design in C, L. Kruse, Bruce P. Leung, Clon's L Tondo, Prentice-Hall of India Private Limited, 1999.
10. Data Structures with Java, John R. Hubbard, Anita Huray, Prentice-Hall of India Private Limited, 2005.
11. Data Structures using C and C++, Yedidyah Langsam, Moshe J. Augenstein, Aaron M. Tenenbaum, Prentice-Hall of India Private Limited, 2006.
12. Data Structure and Program Design, Robert L, Kruse, Prentice-Hall of India Private Limited, 2005.
13. Data Structures and Algorithms in Java, Michel T. Goodgrich, Roberto Tamassia, John Wiley and Sons, Inc. 1998.
14. Data Structures and Algorithms using C#, Michael McMillan, Cambridge University Press, 2007.

INDEX