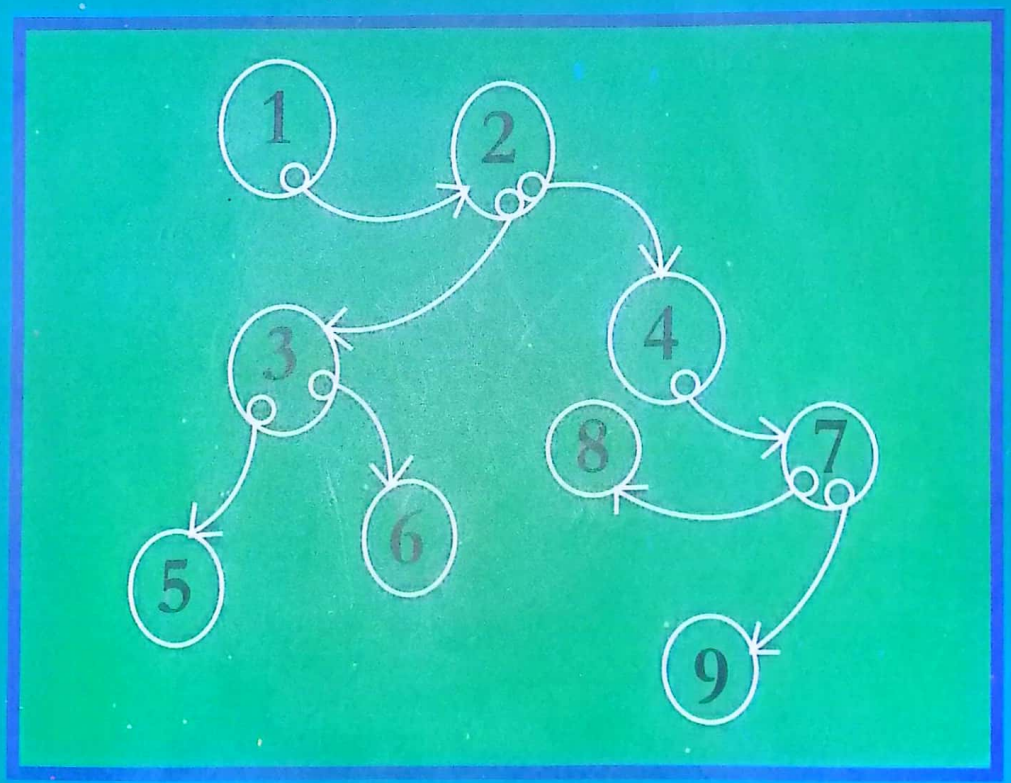
Third Edition

Data Structure Fundamentals

Prof Md Rafiqul Islam, Ph.D
Prof Md Abdul Mottalib, Ph.D



ULAB
UNIVERSITY OF LIBERAL ARTS
BANGLADESH

Published by:
University of Liberal Arts Bangladesh (ULAB), House 56,
Road 4/A (Satmasjid Road) Dhanmondi, Dhaka - 1209
Website: www.ulab.edu.bd

Year of Publication (1st Edition): 2009
Year of Publication (2nd Edition): 2011
Year of Publication (3rd Edition): 2020

ISBN: 978-984-33-0384-4

Price: Tk.350
US$12

Contacts for ordering:
dmri1978@gmail.com
mottalib@gmail.com

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

Dedicated to my father, father-in-law who died while I was doing my PhD and my mother who died in 2016.

------ Prof. Md. Rafiqul Islam, Ph.D.

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

------- Prof. Md. Abdul Mottalib, Ph.D.

PREFACE

This book describes data structures, structural organization of data and operations 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 the 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 the operations on the data structures. The main three operations are *addition or insertion, searching or finding location and deletion*. The operational processes 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 and problems for practical classes.

In this third edition we have done necessary corrections of the second edition and major modifications have been done in chapter 2(array), chapter 4 (linked list), chapter 7 (tree) and chapter 8 (graph).

The book has additional two 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#. 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. AkramulAzim, 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.

Prof. Md. Rafiqul Islam, Ph.D.
Prof. Md. Adul Mottalib, Ph.D.

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 mean raw facts or information those can be processed to get one or more results or products.

A structure is such a unit where elementary items are organized in different ways to perform one or more operations. As for example, some elementary items like hand, leg, eye, ear, nose, bone and some other organs constitute a human body. So, the 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, the data structure is also constituted with some elementary data items.

The *data structure* is a structure or compound unit where we organize elementary data items in different ways to perform operations on them and there exists structural relationship among the items. It indicates a data structure is a means of structural relationships of elementary data items for storing into and retrieving from 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.

Examples 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.

Insertion (addition) means to place (add) an element into a particular data structure. As a result, the number of elements in the data structure will be increased. Deletion (access) indicates removal of an element from a data structure. In some cases, we can delete or remove an element from the data structure. In such cases, the number of elements will be decreased. However, in some cases actual removal (deletion) does not occur, we can only access (read) the element from the data structure and use it in any operation. If the element is not required anymore we can use the space of the element (which was accessed) to store another element. By searching, we understand to locate or find out a specific element of the data structure. Sorting is an arrangement of the elements in any order. The order may be ascending (smaller to greater) or descending (greater to smaller). Merging indicates to collect the elements of two or more data structure into another data structure. To perform any operation we have to design algorithm or/and program.

1.3 Algorithm

It is a set or sequence of instructions arranged in steps that can be followed to solve a problem or perform a task. Each and every algorithm can be

divided into *three sections*. The first section is *input* section, where we show which data items are to be used for processing. Here we should think about the use of proper data structure to store data items. The second section is very important one, which is *operational* or *processing* section. Here we have to perform all necessary operations, such as computation, taking a decision, calling other procedure (algorithm) etc. The third section is *output*, where we display the result(s) found from the second section. When we design an algorithm we have to remember presentation, clarity, and efficiency of it. An algorithm should look good, easy to understand and code using any programming language should give correct result(s) for all valid inputs as quickly as possible. It should make efficient use of computing resources. The efficiency of an algorithm can be measured by analyzing its complexity. To write an algorithm we do not strictly follow the grammar of any particular programming language. However, its language may be near to a programming language.

1.4 Program

A program is a set or sequence of instructions arranged in some statements of any programming language that can be followed to solve a problem or 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 discrete elementary items, rather there exists a structural relationship among elementary data items. So, the program uses the 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 a computer that we see and touch) is useless. So, the 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 the data structure(s) in a program. As we know we write programs to solve problems. It means to solve the problem(s) 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 a *linked list*. In our daily life, we handle a list of data such as a list of students, a list of customers, a 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. 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 the importance of data structures is many folds in storing, accessing, arranging data.

Since all other data structures are implemented using the basic data structures like array, record (structure) and linked list so very good background on these basic data structures is very much essential. There is no single data structure that can be used to solve all the problems. So by achieving the knowledge of data structure, we can use different types of data in programs those are used to solve various problems related to our life. Without knowledge of data structures, we are not able to solve problems where we must use them. In a programming language, there are provisions to use different types of data structures so that data can be organized in different ways and solve the problem properly. In fact, optimization of usage of memory can be done by using proper data structures. In other words, without knowledge of data structures we will not be able to write program properly, hence we will not able to solve the problem.

Sometimes practitioners think that the computing resources like processors and its associated parts are powerful now and there are enormous spaces in memory so the need for data structure is limited. However, let us think

about a large amount of data created by a big organization or group of companies and efficient use of computing resources and Big Data. Even scientists are thinking and doing research on new data structure which can process a large amount of data efficiently.

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

The efficiency of an algorithm or a program depends on its complexity. The complexity of an algorithm can be measured by its running (execution) time and space in memory required for the data used in it. There are two types of complexities: One is **time complexity** and another is **space complexity**.

Time complexity: This complexity is related to the execution time of an algorithm or a program. It depends on the number of operations taken by the program or algorithm. The operations are arithmetic and/or relational (greater, less, equal). However, the comparison based algorithm described the number of element comparisons. So, the complexity of an 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 are 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 a number 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 $10n^2 + 4n + 2 \leq 11n^2$. Some other examples are as follows:

$$5n^2 + 3n + 4 = O(n^2)$$
$$6.2^n + n^2 + 3 = O(2^n)$$
$$3n + 5 = O(n)$$
$$5n\log_2 n + 4 = O(n\log_2 n)$$
$$2n^3 + n^2 + 5n = O(n^3)$$

Complexity in notation	Complexity in word
logn	Logarithmic
n	Linear
nlogn	
n^2	Quadratic
n^3	Cubic
2^n	Exponential

$\log n < n < n\log n < n^2 < n^3 < 2^n$
logn means $log_2 n$ and it denotes the lowest complexity and 2^n denotes the highest complexity.

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

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index is 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 except for top item
Queue	Provides first-in, first-out(FIFO) access	Slow access to other items except for front or rear item.

Data Structure	Advantages	Disadvantages
Binary tree	Quick search, insertion, deletion (if tree remains balanced)	The deletion process is complex for non-leaf nodes.
Hashtable	Very fast access if key is known, Fast insertion	Slow deletion, access slow if key is not known, inefficient memory usage.
Heap	Fast insertion, deletion	Slow access to other items other than root.
Graph	models real-world situations	Some algorithms are slow and complex

Summary:

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

Algorithm is a set or sequence of instructions arranged in steps that can be followed to solve a problem or perform a particular task.

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 to 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 the data structure.
7. Data structure is a structure that may contain another data structure. Explain the statement with example(s).
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
- Identify dynamic array
- Write algorithms using dynamic array
- Give some problems related to array

ARRAY

2.1 Definition

An array is a finite set of the 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. The elements of an array are stored using its name and size (number of elements in the array). Any element of an array is referred by *array name* and the *index number* (subscript). There may have many dimensional arrays. But usually, two types of arrays are widely used; such as one dimensional (linear) array and two dimensional array.

2.2 One Dimensional Array

2.2.1 Definition

A one dimensional (linear) array can be represented by only one dimension such as row or column and holds a finite number of the same type of data items associated with a name. Graphical view of a one dimensional array is as follows:

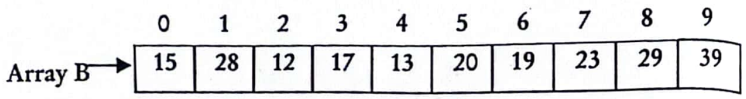


Figure-2.1: Graphical representation of one dimensional array.

Here 0, 1, 2, 3, ..., 9 are index numbers, and 15, 28, 12, ... , 39 are data items or elements of the array and B is a name of the array. Symbolically an element of the array is expressed as B_i or B[*i*], which denotes element in the i^{th} index of the array, B. We will use 0 as starting index.

Thus B[4] and B[9] denote the elements in the 4^{th} index and in the 9^{th} index 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 elements of an array may be numbers (integers or floating point numbers) or characters.

A one dimensional array at a glance as below:
1) A list of same type of data items.
2) It has a name.
3) It has a size (number of data items).
4) Its elements are associated with sequential numbers called indices such as 0, 1, 2, 3, . . .and so on.
5) Any element or item is represented using a name of the array and its index number.

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 B[10];

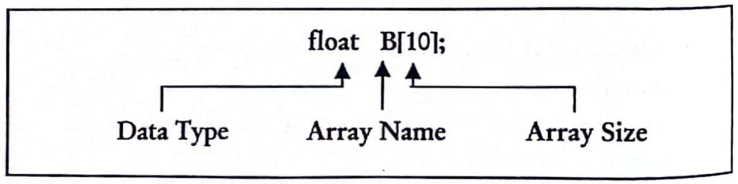


Figure-2.2: Declaration of Array in C/C++.

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

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

The equivalent code in C++ is as follows:

```
int x[10];
for (i = 0; i < 10; i++)
cin >> x[i];
```

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

The following code in C is for reading data from an array and the data will be displayed on the monitor's screen:

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

The equivalent code in C++ is as follows:

```
for (i = 0; i< 10; i++)
cout << x[i];
```

In the above examples of code, type of the array, x is an integer. If we want to access (read) any element from the array to a variable p, the type of the variable, p should be an integer. The example of code is as below.

```
int p;
p = x[0];
```

Suppose there is an array c of 10 characters and we want to assign a value to its index 4 (counting from 0), the code will be as follows.

```
char c [10];
c[4]= 'v';
```

If we want to assign a value from a variable the code will be as below.

```
char c[10];
```

```
char ch;
ch = '(';
c[4] = ch;
```

It can be seen that types of the array and the variable used to assigned values to the elements of the array are same.

2.2.3 Store an element into an array

B[3] = 19; here a value, 19 is assigned to the 4th element (index 3) of the array, B. That is, 19 is stored to the index 3 of the array, B. If there was any previous value, it would be overwritten.

We can assign a variable to any element of the array also. In this case, the type of the array and the type of the variable should be **same**. Such as, *B[2] = x*; here the variable *x* is assigned to the 3rd item of the array, B. If there is any value of *x* that would be copied to the item number 3 of the array. The previous value of item number 3 of the array will be overwritten and there will be no change in the value of *x*.

0	1	2	3	4
7	25	28	54	32

Array, B before execution of the statement (B[3]=19).

0	1	2	3	4
7	25	28	**19**	32

Array, B after execution of the statement (B[3]=19).

Figure-2.3: Data storing using assignment

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

C = B[3]; this statement indicates that the item number 4 (index 3) of the array B is assigned to a variable, *c*. The value from the index 3 of the array will be copied to the variable c. Any previous value of c will be overwritten. There will be no change in the value of item number 4 of the array, B. Let us consider the data of the array in Fig-2.3. After execution of the statement C = B[3]; the value of the variable C will be 19.

From the above description, we can say that the assignment statement works *from direction right to left*. The right side contains the value transferring item and the left side is the value receiving item. When it needs to store more than one value to several items of an array, we have to use a

loop of the programming language. Such as we can use **for** or **while** loop. Similarly, if we want to access several items from an array we have to use a loop. Since in C/C++ *index of an array start from 0* (zero) instead of 1, so for 10 items in an array, the indices will be 0, 1, 2, 3, . . ., 9.

0	1	2	3	.....	8	9
d	F	g	h	.....	m	p

Figure-2.4: Array with character

The type of the array indicates the type of its elements. Such as if the type of the array B is character, the elements B[0], B[1], ..., B[9] will also be characters.

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*[0 . . . n-1];
2. for (*i* = 0; *i* < n; *i* ++) //n is the size of the array
   cin >> x [i];
3. *large* = *x*[0];
4. for (*i* = 1; *i* < n; *i* + +)
   if (*x*[*i*] > *large*), *large* = *x*[*i*]; //if any element larger than the previous data in *large*
5. Output: *large* (the largest number);

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.

Now we will see the simulation results of Algorithm 2.1. Suppose that there is an array, x with the following data items.

15	12	9	17	12

We see the simulation results from step 3 since steps 3 and 4 are the steps of processing section.

1) *large* =15
2) Within *for* loop:
   a) (12 > 15) **no** (no change in the value of *large* = i.e. *large* =15)
   b) (9 > 15) **no** (as above)
   c) (17 > 15) **yes** *large* =17 (change in the value of *large* =15)

d) (12 > 17) **no** (no change in the value of *large*, which was 17).

From the above simulations, we have got *large* =17 which is the largest item of the array, x.

Problem 2.2:

Given a linear array with data, find out a particular (specific) element, *x* from the array. We do not know the index (cell) number where the element was stored.

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

1. Input: A set of data in an array a and variable ***x*** *i.e.*, the target element
   $a[0 \ldots n\text{-}1], x$; //n is the size of the array
2. flag = 0;
3. for ($i = 0; i < n; i++$)
   {
   if ($a[i] == x$)
   location = i, flag = 1, break;
   }
4. Output: if (flag = = 1), print "Found" and *i* (location).
   else print "Not Found".

Simulation results of the algorithm 2.2 are as follows:

	0	1	2	3	4	5
A	7	15	12	9	24	32

We will see from step 2 and let $x = 24$

1) Initially *flag* = 0

2) Within for loop:
a) (7 = x) no (no change in the value of flag *i.e.*, *flag* = 0)
b) (15 = x) no (flag = 0)
c) (12 = x) no (flag = 0)
d) (9 = x) no (flag = 0)
e) (24 = x) yes, *flag* = 1 and break (exit the *for* loop)
3) Since *flag* = 1, output will be Found and 4 (*index* = 4)

Problem 2.3:

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

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

To store the results, we require two variables such as *sum_even* and *sum_odd*. Initially, values of these variables will be zero (0) and when we find an even number we add it to the *sum_even* and we find an odd number, we add it to the *sum_odd*. If a number is divisible by 2 it is even, otherwise odd. We have to start with the first number of the list. If it is even, it is added to the sum_even and if it is odd, it is added to the sum_odd. Similarly, we

Similarly, we shall access the whole list one by one and add them to 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)

   $A[0...n-1]$, $sum\_odd = 0$, $sum\_even = 0$; //n is the size of the array

2. for ($i = 0$; $i < n$; $i++$)

   {

   if ($A[i]\%2 == 0$), $sum\_even = sum\_even + A[i]$;

   else $sum\_odd = sum\_odd + A[i]$;

   }

3. Output: *sum_even* (the summation of the even numbers) and *sum_odd* (summation of the odd numbers)

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

Problem 2.4:

Given a list of integers stored in a linear array, find out the summations of numbers in odd 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 (not the number) is odd or even.

Algorithm 2.4: Algorithm to find the summations of the numbers in even and odd indices.

1. Input: An array and variables (to store the results of summations)

   $A[0...n-1]$, $sum\_odd = 0$, $sum\_even = 0$; //n is the size of the array

2. for ($i = 0$; $i < n$; $i++$)

   {

   if ($i\%2 == 0$), $sum\_even = sum\_even + A[i]$;

   else $sum\_odd = sum\_odd + A[i]$;

   }

3. Output: *sum-even* (Summation of the numbers in even indices), *sum_odd* (summation of the numbers in odd indices)

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

Problem 2.5:

Given *n-1* integers in a linear array, stored in 0 to *n-2* positions (indices). **The (*n*-1)th index is empty.** Insert a data (integer) in *k*th position (index), the relative order of the data before the position and after the position will remain as it is, where $0 < k < n-2$. **Condition:** You cannot delete (overwrite any item). Any item and arrangement of data before and after *k*th index should be same as previous.

0	1	2	3	4	5	6	7	8
12	15	25	28	35	45	52	60	Empty position

Figure-2.5: Array with data before insertion

Let $k=5$ and $x=40$ should be inserted at 5th index of the array. The output should be as follows:

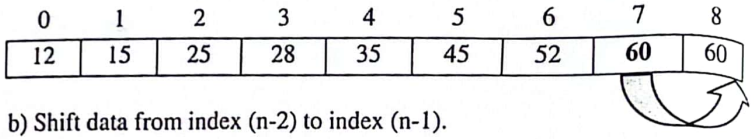
0	1	2	3	4	5	6	7	8
12	15	25	28	35	**40**	45	52	60

Figure-2.6: Array with data after insertion of 40 at 5th index.

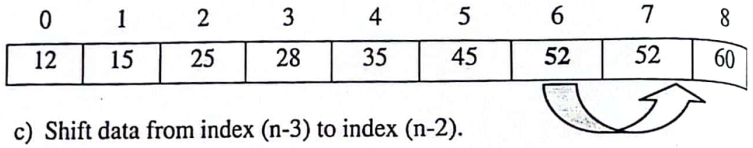
We see now the solution process of this problem in pictorial view.

12	15	25	28	35	45	52	60	

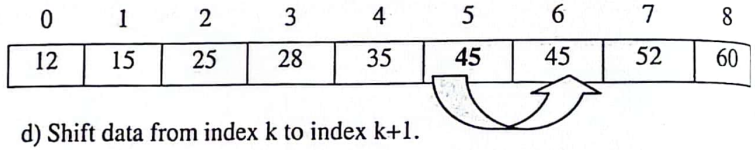
a) Array with data before insertion



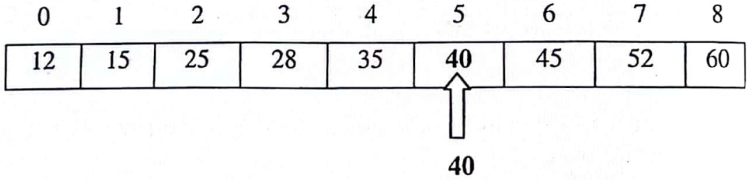
b) Shift data from index (n-2) to index (n-1).



c) Shift data from index (n-3) to index (n-2).



d) Shift data from index k to index k+1.



e) Insert data to the desired position

Figure-2.7: A pictorial view of data insertion to a particular position.

Algorithm 2.5: Algorithm to insert an item into a particular position (index) of an array.

1. Input: An array A[0.....n-1] with data in indices 0 to n-2, and the data item, x should be inserted into the index, k.

2. for (i = n-2 ;i >= k; i--)
   A[i+1] = A[i]; //A[i+1] ← A[i]
3. A[k] = x; //A[k] ← x

4. Output: Array A with data in all indices.

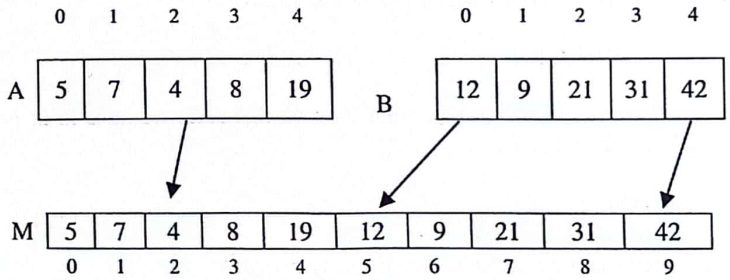
Problem 2.6:
Given a list of integers stored in a linear array, delete a data from a given position of the array.

Algorithm 2.6: Algorithm to delete an element from an array.

1. Input: An array *A*[0...*n-1*], //n is the size of the array
2. for (*i* = *k*; *i*< *n*; *i* + +)
   A [*i*] = *A* [*i* + 1]; //A[i] ← A[i+1]
3. Output: The updated array, *A*.

Comments: Here the data of the positions (index) *k* to *n-2* are overwritten by the data of the positions *k* + *1* to *n-1* of the array, *A*. Now the position (index) *n-1* 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* [0...*m-1*] and *B*[0...*n-1*] //m is the size of A and //n is the size of B
2. Take an array *M* [0... *m* + *n-1*] //m + n is the size of M

3. for ($i = 0$; $i < m$; i ++)
   $M[i] = A[i]$; //M[i] ← A[i]
4. for ($i = 0$; $i < n$; i ++)
   $M[i+m] = B[i]$; //M[i + m] ← B[i]

5. Output: The merged array, *M*.

Comments: Here the output array is *M*. The data of the array, *A* are copied to the positions *0* to *m-1* of *M* and the data of the array, *B* are copied to the positions *m* to *m + n-1* of *M*. Here position means index.

Now we see how we can perform some simple operation using array of characters. We can assign values to an array of characters as follows:

```
char test[5]={'a','b','c','d','e'};
We can see output using for loop
for(i = 0;i < 5; ++i)
cout<<test[i];
```

The above statement is equivalent to input some charters to the array. If we want to see the output we can use the code as below.

```
cout<<test;
```

A character array can be assigned values as follows.

```
char name [10]="A Rahim";
```

We can do the same thing in another way. In this case, we have to include "string" as a header file.

```
char name[10];
strcpy(name, "A Rahim");
We can see separate charter above the array using for loop.
for(i=0; i<10; ++i)
cout<<name[i];
```

Another way to enter data into an array of characters and display them on the screen is as below. Here **gets** is a function that takes character data

including space. To use **gets** function we have to include string.h as a header file.

```
char name[30];
fflush(stdin);
gets(name);
cout<<name;
```

Problem 2.8: Find a particular character from an array of characters

Algorithm 2.8: Searching a character

```
1. Input an array of character
char test[30] =" Mid term examination";
2. Search for the character:
 for (k=0; k<30; k++)
{
        if(test[k] == 'x')
        {
                cout<< "Found";
                break;
        }
}
```

We can write the above algorithm in another way.

```
1. Input an array of character
char test[30] =" Mid term examination";
2. input the character you search for:
char ch;
cin>>ch
2. Search for the character:
 for (k=0; k<30; ++k)
{
if(test[k] == ch)
{ cout<<"Found in index"<<k;
break;}
}
3. if (k>29) cout<<"Not Found";
```

Problem 2.9: There two very large numbers say each number is 40 digits long. How can we add these two numbers?

Description of Solution: We can solve this problem using arrays. Take digits of numbers in two arrays and we have to put results in the third array. Each digit of the numbers is an element of the array. Two elements of two array are added and we check whether there is any carry. If there is a carry we will add it to the sum of the next two elements. This method is as like as a paper-pencil method of addition.

Algorithm 2.9: Algorithm to add two very large numbers.

1. Input the digits of the first numbers in array, a[0 …n-1];
2. Input the digits of the second number in array, b[0…n-1].
3. Take an empty array c[0…..n]. //size of the array c will be n+1.
4. k = 0;
5. for (i = n-1; i >= 0; i--)
{
c[i+1] = (a[i] + b[i] + k) % 10;
k = (a[i] + b[i] + k) / 10;
}
6. c[0] = k;
7. Output the array, c[0….n] (Display the data from c[0] to c[n]);

Problem 2.10: There two very large numbers say each number is 40 digits long. How can we multiply these two numbers?

Algorithm 2.10: Algorithm to multiply two very large numbers.

1. Input the digits of the first numbers in array, a[0 …n-1];
2. Input the digits of the second number in array, b[0…m-1].
3. Take an empty array c[0…m+n-1].
4. Initialize all the items of c to zeros.
5. for (j = m-1; j >= 0; j--)
{
i) k = 0;
ii). for (i = n-1; i >= 0; i--)
{
t = a[i]*b[j] + c[i+j+1] + k;
c[i+j+1] = t % 10;
k = t / 10;

}
iii). c[j] = k;
6. Output the array, c (Display the data from c[0] to c[m+n-1]).

2.3 Pointer and dynamic array

To understand dynamic array, knowledge of pointer is necessary. So, in this section at first, we describe pointer shortly and then dynamic array.

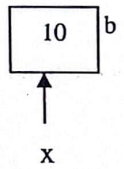
Pointer: A pointer is like a variable, but it does not hold actual data. It holds the *address* of the data. It is a variable that holds the *address* of another variable. In other words, a pointer points any other variable. We can declare pointer variable as follows:

```
int *x;
char *ch;
```

To declare pointer variable we use an *asterisk* (*). Here *x* is a pointer variable that points *integer* type data. We can say alternatively *x* is a pointer variable that holds the address of an integer type data. Similarly, ch is a pointer variable that points *character* type data. Let us see examples.

Example 2.1: Use of pointer

```
1. int *x;
2. int b;
3. b = 10;
4. x = &b;
5. cout << x;
6. cout << *x;
```



In the above example, *x* is a pointer and b is a variable for integer type data. Since *x* can contain the address, so we assign the address of b to *x* (line 4 in example 1). Line 5 displays address of b and line 6 shows 10 (the data that was assigned to b in line 3).

Example 2.2: Use of pointer using **new**

```
1. int *p;
2. p = new int;
3. *p = 12;
4. cout << p;
5. cout << *p;
```

Line 4 shows the address and line 5 displays 12 (the data that was assigned to p). At line 2 we have allocated space in memory for an integer. Here *new* is the keyword in c++ for allocating *space* in memory. Without allocating space we cannot assign any value directly using a pointer.

Example 2.3: Incorrect use of pointer

```
1. int *x;
2. int p =5;
3. *x = p;
4. cout <<*x;
```

The above code will not give any error during compilation. But it will not execute (run) and shows a run time error when we will try to execute it. Since we do not assign an address to the pointer *x*, an error occurs here. Either we have to assign the address of a variable to the pointer or we have to allocate space using *new* keyword and assign data.

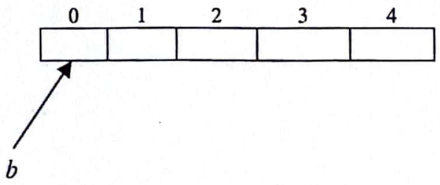
2.4 Dynamic array (array using pointer)

A dynamic array can be defined using a pointer and keyword *new*. To declare an array we have to define the size of the array. Without defining the size we cannot declare an array. A dynamic array can be defined as follows:

Example 2.4: space allocation for dynamic array.

```
1. int *b;
2. b= new int[5];
```

Here b is a pointer to an integer type data. Using pointer we can allocate space in memory for a number of data items. At line 2 of the above code, we have allocated space for five integers. The graphical view for allocating space for five integers is as follows:



To store data to a dynamic array the code will be as follows:

Example 2.5: Store data in a dynamic array

```
int *b, i;
b = new int[5];
for (i = 0; i < 5; i++)
cin >> b[i];
```

2.4.1 Increasing size of the array using dynamic array

After defining the size of an array we cannot increase or decrease without redefining it. However, we can increase the size of the array using a dynamic array. Let us consider that using the code of **related to problem 2-5** and after entering data we will get the array with data as follows.

0	1	2	3	4
15	21	25	19	45

Figure-2.8: Before increasing size

Now we want to increase the size of the array by one. In the above figure, there are data in the indices 0 to 4. We can assign data to index 5 as below.

```
a[5] = 99;
```

0	1	2	3	4	5
15	21	25	19	45	99

Figure-2.9: After increasing size

Now the size of the array is six and it happens dynamically due to the assignment of data in index 5. We can display the data of the array using the following code.

```
for (i=0; i<6; i++)
cout << b[i];
```

Now let us see a problem similar to the **problem 2-5.**

Problem 2-11: Given an array with data in all the positions (indices). Now insert a data at a particular index without deleting any data of the array.

Condition: If we have to insert data in the kth index, the arrangement of data before and after *k*th index must be same as it was. The Figure-2.10 and Figure-2.11 depict the scenario of input array and output array respectively.

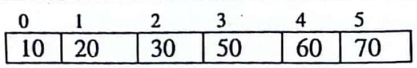


Figure-2.10: Before insertion of data

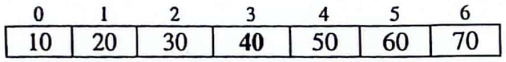


Figure-2.11: After insertion of data, 40 in index 3.

Description of Solution: We shift the data one by one as follows and solve the problem. Let us see the pictorial view of solving process of the problem.

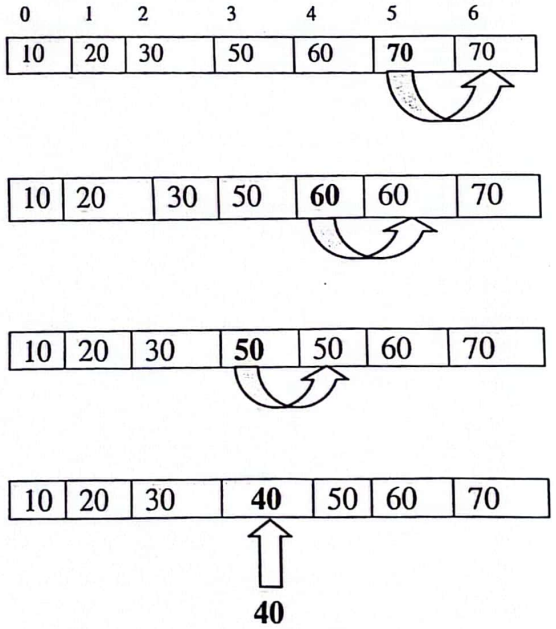


Figure-2.12: Pictorial view to insert a data in a particular position using a dynamic array.

We can solve the above problem using a dynamic array of the algorithm written below.

Algorithm 2-11: Algorithm for inserting data in a particular position

1. Input:
   a) a dynamic array, d with data in all indices

   b) index to which the data will be inserted, *k*.
   c) data (variable) to be inserted, *x* in *k*th position.

2. Shift the data from *(n-1)*th index to nth index, *(n-2)*th index to *(n-1)*th index and do this until you reach at *k*th index:

```
for (i = n-1; i>=k; i--)
  d[i+i] = d[i];
```

3. d[k] = *x*;
4. output: updated array, d;

Let us see how we can decrease the size of an array using a dynamic array. Just before this, we saw that we can increase the size of the array dynamically or using a dynamic array. However, we cannot decrease the size of the array using only one array. In the case of reduction of the size of the array, we need an assisting array. Let us consider that we have an array of 10 elements. Now we have to decrease the size of the array to 7. Finally, there will be 7 elements in the array. We can do it as follows:

```
int *a, *b;
a= new int [10];
for (i=0; i<10; i++)
cin>> a[i];
for (i=0; i<7; i++)
b[i]=a[i];
delete [ ] a;
a = b;
```

Now the data of array can be easily displayed and there are 7 elements of the array, but not 10.

2.5 Set operations using array

In this section, let us see some operations on set using an array. The operations are union, intersection and difference.

2.5.1 Union of two sets

Let A and B are two sets. We can represent the union of sets A and B as follows:

A U B = {x: x ∈ A or x ∈ B} where x denotes an element of any of these two sets.

If A = {1, 3, 7, 8} and B = {2, 7, 8}
So, A U B = {1, 3, 7, 8, 2}

Another example,

X = {1, 3, 7, 9} and Y = {3, 4, 6}

So, X U Y = {1, 3, 7, 9, 4, 6}

We can represent a set using an array. When we make a union of two sets, the resulting set will be another set, but the size of this set is not fixed or cannot be predefined. So, here we can use a dynamic array for set union operation. We can do a union of two sets using the following algorithm.

Algorithm 2-12: Algorithm for union of two sets

1. Input: an array, A[m] with its elements (data) and an array, B[n] with its elements (data)
2. Take an array, C[m+n] with size of A and B.
3. for(i = 0; i < m; i++) // we consider that m >= n.
   C[i]=A[i]; //copy elements of A to C
   p=m;
4. i) for (i=0; i<n; i++) //with the size of smaller array
   {
   flag=0;
   for(j=0;j<m; j++) //with the size of larger array
   {
   if(C[j] = = B[i])

   {
   flag=1; break; //for equal items flag =1;
   }
   } //end of j loop
   if(flag==0)
   {
   C[p]=B[i]; //copy item which was not in C before.
   p++ //next index in array, C.
   }
   } //end of i loop
5. Output array C with data.

The students can try to implement the above algorithm using a dynamic array.

Intersection of two sets: Intersection of sets A and B can be represented as A ∩ B. We can write
A ∩ B = {x : x ∈ A and x ∈ B}.
Let A = {2, 4, 6, 8, 10} and B = {8, 10, 12}
A ∩ B = {8, 10}

We can perform intersection operation of two sets using the following algorithm.

Algorithm 2-13: Algorithm for intersection of two sets

1. Input an array A with data and array B with data
2. Take a dynamic array, C and a variable p = -1;
3. for (i=0; i<n; i++) //with the size of smaller array
   {
   for(j=0;j<m; j++) //with the size of larger array
   {
   if(C[j] = = B[i])
   {
   ++p;
   C[p] = B[i]; break;
   }

4. Output: array C with data.

Set Difference: Difference of two sets can be represented as follows:

A – B = {x : x ∈ A, x not ∈ B} and
B-A = {x : x ∈ B, x not ∈ A} where A and B two sets.
Let A = {1, 2, 3, 4} and B = {6, 2, 4, 8}
A- B = {1, 3} and B –A = {6, 8}
Algorithm for sets difference operation is left as an assignment for the students.

2.6 Two Dimensional Array

2.6.1 Definition

Two dimensional array is an array that has two dimensions, such as *row* and *column*. The total number of elements in a two dimensional array can be calculated by multiplying 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* x *n*, and *m* × *n* is 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 a *table*. It short it is called 2-D array.
The two-dimensional array at a glance:
1) A list of the same type of data times.
2) It has a name.
3) It has a size represented by the number of rows and the number of columns.
4) Rows are numbering as 1, 2, 3,…., m or 0, 1, 2,…., m-1
5) Columns are numbering as 1, 2, 3,…, n or 0, 1, 2,…., n-1
6) Each item is associated with a specific row number and a specific column number.

We will use row numbers as 0, 1, 2,…., m-1 and column numbers as 0, 1, 2,…., n-1.
A two dimensional array can be expressed as follows:

A_{ij} or A[*i*, *j*] for $0 \le i \le$ m-1 and $0 \le j \le n-1$ (where *m* and *n* are the numbers of rows and columns respectively, see Figure-2.13).

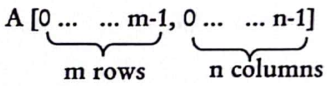
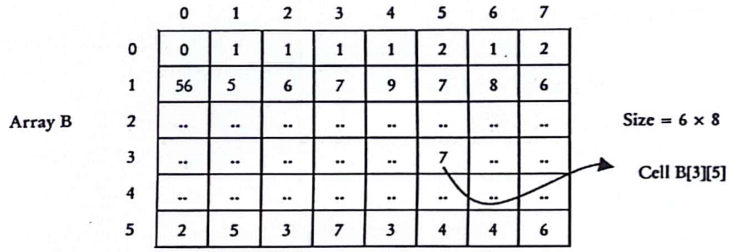


Figure-2.13: Symbolic representation of two dimensional array.



	0	1	2	3	4	5	6	7
0	0	1	1	1	1	2	1	2
1	56	5	6	7	9	7	8	6
2	..	..	..	..	..	..	..	..
3	..	..	..	..	..	7	..	..
4	..	..	..	..	..	..	..	..
5	2	5	3	7	3	4	4	6

Figure-2.14: Graphical representation of a two dimensional array

An element represented as B[1][2] indicates that B is the name of the array, the row number is 2 and column number is 3. If we notice **Figure-2.14,** the item B[1][2] is 6. Thus any item can be found in cross-point of the row number and column number.

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-2.15]

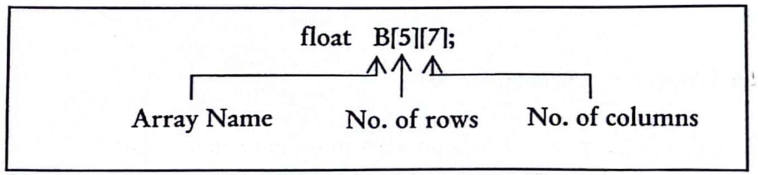


Figure-2.15: Declaration of a two dimensional array in C/C++

The size of this array is 6 × 8. So, the total number of elements or items is 6x8=48.

2.6.2 Store into and retrieve values from 2-D array

As we know in C/C++ indexing start at 0(zero), so we have to store data considering index number zero. 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++)
      cin >> B[i][j];
      }
```

//storing data taken from keyboard

Another way to store data:

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

//Direct insertion of elements

2.6.3 Assigning a value to an item

B[3][4] = 25; Here the item represents in cross-section of index 3 (row 4) and index 4 (column 5) is assigned a value 25.

2.6.4 Reading (Retrieving) an item

y = B[4][2]; The item represented by row number 5 (index 4) and column number 3 (index 2) is assigned to a variable y. If the value of B[4][2] is 85, this value is assigned to the variable y, consequently, the value of y will be 85.

2.6.5 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 a *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 belwo.

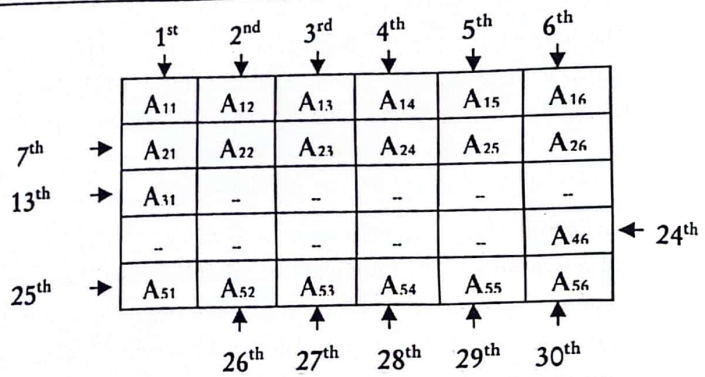
$\underbrace{A_{11}, A_{12}, A_{13}, A_{14}, A_{15}, A_{16}}_{\text{row 1}}\ \underbrace{A_{21}, A_{22}, A_{23}, A_{24}, A_{25}, A_{26}}_{\text{row 2}}\ , A_{31}, \ldots ., A_{46}\ldots., \underbrace{A_{51}, A_{52}, \ldots\ldots, A_{56}}_{\text{row 5}}$

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

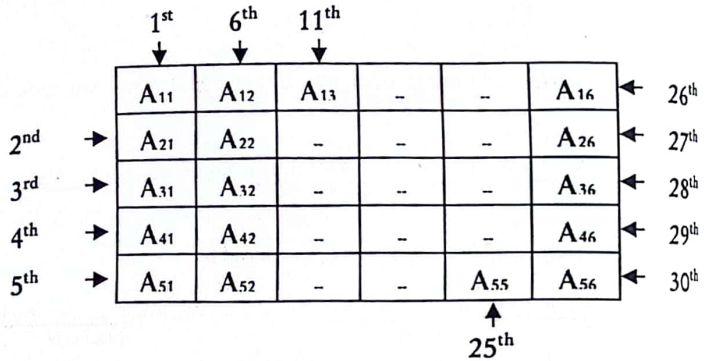
$\underbrace{A_{11}, A_{21}, A_{31}, A_{41}, A_{51}}_{\text{column 1}}\ \underbrace{A_{12}, A_{22}, A_{32}, A_{42}, A_{52}}_{\text{column 2}}\ . A_{13}, \ldots\ldots\ldots, A_{55}, \underbrace{A_{16}, A_{26}, \ldots\ldots\ldots, A_{56}}_{\text{column 6}}$

Figure-2.16: shows these arrangements in details.

In row-major order all the rows make a long row (one dimensional array) in the memory. Similarly, in column-major order all the columns make a row (one dimensional array) in the memory. In figure 2.17 1st, 2nd, 3rd etc denote cell numbers of the elements in the array.



(a) Row-major order



(b) Column-major order

Figure-2.17: Pictorial view of a two dimensional array representation in memory

2.6.6 Location of an element of a two-dimensional array

***Row-major Order*:**

Let Loc (A[*i*, *j*]) be a 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))* 4 (4 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)* 4
= 100 + (5 × 2 + 3)* 2
= 100 + 26
=126

Problem 2.12:

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

Description of Solution: First, we have to identify the boundary elements. In a two dimensional array, elements of the first and the last columns and the first and last rows are the boundary elements as shown in the Figure-2.18.

Here the index, *i* represents the row number and *j* represents the column-number. When *i* is 0, it indicates the first row and when *i* is m-1 (where m

represents the number of rows in the array), if is the last row. Similarly, when $j = 0$, it indicates the first column and the index of the last column is $j = n - 1$. So, we get the number of boundary elements if $i = 0$, $i = m-1$, $j \approx 0$ or, $j = n-1$.

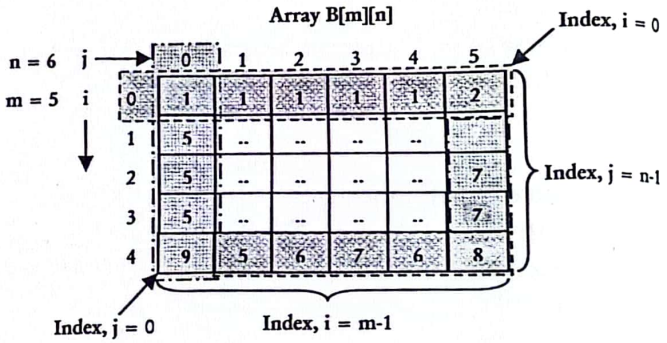


Figure-2.18: Boundary elements of a two-dimensional array

Algorithm 2.14: Algorithm to find the summation of boundary elements

1. Input: a two-dimensional array

   A[0...*m-1*, 0...*n-1*]

   sum = 0;

2. Find the boundary element

   for ($i = 0$; $i < m$; i++1)

   for ($j = 0$; $j < n$; j ++1)

   if (i ==0 || j == 0 || i ==*m-1* || j = -1), *sum* = *sum* + A[i][j];

[Boundary elements are those elements whose index $i = 0$ or $j = 0$, and whose index $i = m-1$ or $j = n-1$) and add it with *sum* (previous result)]

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

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

Description of 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, the number of columns and number of rows are same].

In the Figure-2.19, 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.

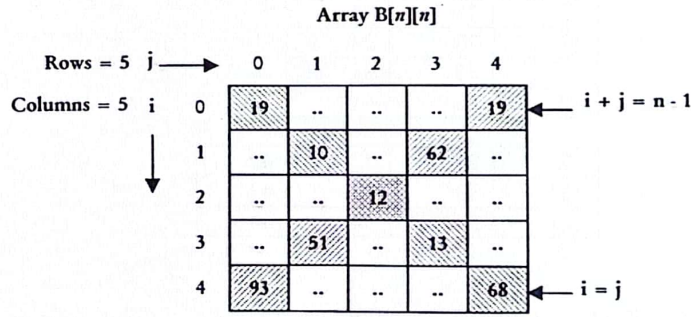


Figure-2.19: Diagonal elements of a two-dimensional array.

Algorithm 2.15: Algorithm to find out summation of diagonal elements

1. Input: a two dimensional array

   B[0 . . . *n-1*, 0...*n-1*]

   sum = 0;

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

```
for (i = 0; i< n; i ++)
        for (j = 0; j <n; j ++)
        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 is n -1 (*i.e.*, $i + j == n-1$).]

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

Problem 2-13: Find out the transpose of a matrix.

Description of Solution: There is a matrix (two-dimensional array), A. We can find out transpose of the matrix by converting the elements of each row as the elements of the respective column of the matrix. If the size of the matrix is m × n, the size of the transpose matrix will be n × m. Let us see the following example.

A

12	13	14
22	23	24
32	33	34
42	43	44

B

12	22	32	42
13	23	33	43
14	24	34	44

Here A is a matrix and its transpose matrix is B. A have the size 4 x 3 and the size of the matrix B is 3 x 4.

We can find transpose of a matrix using the following algorithm.

Algorithm 2-16: Algorithm to find out the transpose of a matrix.

1. Input a matrix A[m][n] with data
2. Take a matrix B [n][m].
3. for (i=0; i<m; i++)

```
for(j=0; j<n; j++)
{
```



```
B[j][i] = A[i]lj]; //row to column and column to row
}
}
```

4. Output matrix B with data

Problem 2-14: Multiply two matrices and find a product matrix.

Algorithm 2-17: Algorithm for multiplication of two matrices.

Description of Solution: For multiplying two matrices there is a condition, the number of columns of the first matrix must be equals to the number of rows of the second matrix. If there are two matrices A with size m × n and B with n × p, the size of the product matrix will be m × p.

Algorithm 2-18: Algorithm to find the product of two matrices

1. Input two matrices A [m, n] and B[n, p] with data
2. Take another matrix C [m, p].
3. i) for (i = 0; i < m; i++)

   ii) for (j = 0; j < p; j++)

   C[i, j] =0;

   iii) for (k = 0; k < n; k++)

   C[i, j] = C[i, j] + A[i, k]*B[k, j];

4. Output: matrix C of size m × p with data

Problem 2.15: There are 40 students in a class. They have written 4 class tests of a course. Find out the average of the best 3 class tests for each student.

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

Algorithm 2.19 : Find average of best three class tests

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[40][4]], *avg_mrk*[40] ;

2. Add all class test marks for each student

```
for (i = 0; i < 40; i ++)     // first loop starts
        {
        sum = 0;
        min_mrk = marks[i,1];
        for (j = 0; j <4; j ++)        //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 marks of four class tests for each of the 40 students and ***avg_mrk*[40]** is an array to store the computed average marks of best three class tests. The ***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.

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.

Like the linear array of pointers we can use 2-D array of pointers also. In this case, we need some rows of pointers and each of these pointers has to point some columns of pointers. In **Fig. 2-20,** we have shown four rows of pointers and each pointer points five columns of pointers. Using this array we can store 20 (4x5) elements. Using pointers we can use variable

numbers rows and columns in the array and this type of 2-D array is called dynamic 2-D array.

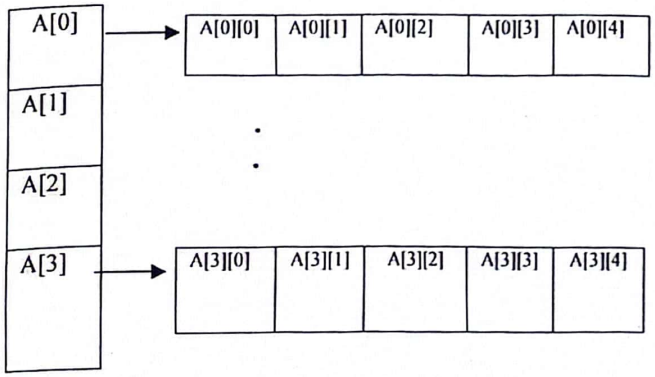


Figure-2.20: Pictorial view of a 2-D dynamic array

Let us see how to declare a 2D pointer array. For the above figure, we can write pseudo-code for declaration as follows.

```
int *A[4];
for (i=0;i<4;i++)
A[i] = new int [5];
```

The first line indicates an array of four pointers and the second and third lines are for five columns for each pointer. The above array, A is not fully dynamic, since here the number of rows should be a constant, but number columns may be variable. So, we can write similar code as

```
int *A[4];
int c, cin >> c
for(i=0;i<4;i++)
A[i]=new int [c];
```

We can use any value of c to get a variable number of columns.

To see how it will work we can use the following code.

```
int *A[4];
```

```
int i,j;
for (i=0;i<4;i++)
A[i] = new int [5];
for (i=0; i < 4; i++)
{
  for (j=0; j < 5; j++)
  {
    A[i][j] = i+j;
cout<<A[i][j]<<" ";
  }
cout<<endl;
  }
```

Now we see we can write code for fully dynamic 2D array.

```
int r, c;
cin>>r;
cin>>c;
int  **b=new int*[r];   //r is the number of rows
for (i=0;i<r;i++)
b[i] = new int [c];    //each row has c columns.
```

Let us see the problem (summation of boundary elements). Using dynamic array we can test this problem for different sizes of arrays. Followings are codes for the problem.

```
        int r,c;
        cout<<"Enter number of rows:";
        cin>>r;
        cout<<"Enter number of columns:";
        cin>>c;
int **b=new int*[r]; //r rows

int i, sum=0;
for (i=0;i<r;i++)
b[i] = new int [c]; //c columns
for (i=0; i < r; i++)
{
  for (int j=0; j < c; j++)
  {
    b[i][j] = i+j;
    cout <<b[i][j] << " ";
                if(i==0||j==0||i==r-1||j==c-1)
```



```
                sum=sum+b[i][j];
  }
  cout << endl;
}

cout<<"sum="<<sum;
```

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. In any array same type of elements are used.

Questions:

1. What is a 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 can address of a particular element of a two dimensional array 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 a 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 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 2-2: Write a program to find out the largest and smallest of a given list of numbers in an array. Display the numbers, the largest and smallest separately.

Problem 2-3: 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 2-4: 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 2-5: Write a program to merge two arrays (merge two arrays and make single one). Display the data of the three arrays.

Problem 2-6: 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 ith 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 2-7: Given two arrays of the same size with data (integers) arranged in ascending order. Create the 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 the third array without using any sorting procedure.

Problem 2-8: write a program to find out the summation of boundary elements of a two-dimensional array.

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

Problem 2-10: 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.

Problem 2-11: Take a 2-D array of 3*4 and find out the transpose matrix as output.

Problem 2-12: Take a 2-D array of 4*4 (square) and find out its transpose matrix. There should be some changes in the program, which is for 2-11.

Problem 2-13: Implement **algorithm 2-18** for A[4, 3] and B[3, 4]. Find out the number of multiplications needed to implement the algorithm.

Problem 2-14: There are three matrices as follows:

A [5, 4], B[4, 3] and C[3, 6]. Now write a function that will multiply two matrices and using this function find out the multiplication of three matrices. You can find the multiplication in two ways: i) (A.B).C ii) (B.C)A

Now find out the number of multiplications of both ways and show that which way gives less number of multiplications.

Problem 2-15: Find out the first largest and second largest from a list of numbers. Take 10 integers to execute your program.

Problem 2-16: There are 15 numbers in an array. Among them, there are more than one duplicate numbers. Find out the duplicate numbers from the array.

Problem 2-17: There is a 2-D array of size m × n. We will say a point as a min-max point if there is a number a[i, j] which is smallest in the ith row and largest in the jth column. Write a program to find out the min-max point as <i, j> and the data in the point. Take the following data set and test your program.

8 13 10
9 11 12
5 4 10

Problem 2-18: Take an expression such as (125 + 20 -75) in an array. Copy (put) all the numbers in an array and the +, - in the *third* array. Can you find the results of the expression?

Problem 2-19: Take two numbers of twenty digits each and multiply them. Display the numbers and result **(See algorithm 2-18).**

Problem 2-20: Take two numbers of twenty digits each and divide them. Display the quotient and remainder. After getting results multiply divisor and quotient using **algorithm 2-18**. Add the result of multiplication and remainder using **algorithm 2-18**. Now verify whether your final result is equal to dividend or not.

CHAPTER THREE
RECORD

OBJECTIVES:

- Identify record
- Show data storing and accessing processes using record
- Identify array of records/structures
- Write algorithms using record
- Return values using structure in a function
- Identify 2-D array of records
- Identify dynamic 2-D array of records
- Differentiate the array and the record

RECORD or STRUCTURE

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. As for example, 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 person
    {
    int id;
    char name[40];
```

```
    int phone;
    float due; };
```

Here is another example of a structure using student attributes.

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

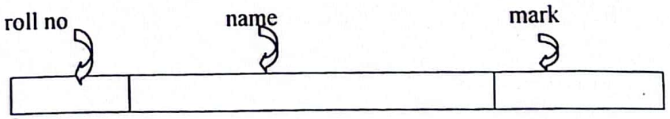


Figure 3.1: Pictorial view of a student record or structure

Here, "name" is a pointer variable that points a character type variable (data item). Student is a structure variable. If the length of "name" field is 40, memory requirement for the above structure (record) = 4 + 40 + 4 = 48 bytes; here each integer takes 4 bytes space and each character is 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 3-1: Example of a structure using c/c++.

```
struct
    {
    int roll_no;
    char name[40];
```



```
    int marks;
    }stud;    // Here, name[40] is an array which is a
              member of the structure, stud.
```

We can assign values to the members of the structure (***stud***) and store data to the members of the structure as follows.

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

Another way to store data to the member of the structure is shown below.

```
cin>>stud.roll_no;
gets(stud.name);
cin>>stud.marks;
```

3.2 Array of structures

In a simple array usually, the data items are of the elementary data type. In an array of structures, the data items are structures. However, all the items should be of the same structure, where members may be of different types. If we want to store data to a record (structure) for more than one entity (person or student or customer), then we have to take (define) an *array of structures*.

Example 3-2: Example of an array of structures using c/c++.

```
struct
    {
    int roll_no;
    char name[40];
    int marks;
    }stud [30];
```

Here, **stud** is an array of structures where members are *roll_no*, *name*[ ] and *marks*, and n*ame* is an array.

stud[0]	stud[1]	stud[2]	stud[3]	stud[4]

Figure 3.2: An array of five structures (each structure have items shown in Figure 3.1)

Here the elements of the array must be the same structure. If there is an array of structures, we can assign values to the members of the structure (stud) as follows:

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

Memory requirement for the array of structures, *stud* (in example 3-2):

{(4 + 40 + 4) × 30} bytes

= 48 × 30 bytes

= 1440 bytes

The disadvantage of the use of an array of structures is that for this type of the array an amount of memory space may be misused or wasted.

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

Solution:

We can show the solution using code in C/C++ as below.

```
struct
 {
int roll_no;
char name [40];
int marks;
 } stud[30];

for (i = 0; i < 30; i++)
 {
cin>>stud [i].roll_no;
fflush(stdin);
gets(stud[i].name);
cin>>stud[i].marks;
 }
```


To display data we can write code using **printf()** or **cout**.

Problem 3.2: There is a 4×4 matrix as follows.

	0	1	2	3
0	0	10	0	8
1	10	0	6	0
2	0	6	0	12
3	8	0	12	0

Create a list of non-zero values along with their respective row and column numbers using only one additional array.

We can solve the problem using the following pseudo-code:

1. Take matrix as input

```
int m[4][4] = { {0, 10, 0, 8},
                {10, 0, 6, 0},
                {0, 6, 0, 12},
                {8, 0, 12, 0} }
```

2. Declare an array of structures for row, column and value.

```
struct
 {
 int u;
int v;
int val;
 } ra[10];
```

3. Store row number, column number and non-zero value to an array of structure.

```
k= 0;
for (i = 0; i < 4; i++)
for (j = 0; j < 4; j++)
 {
```

```
if (m[i][j] != 0)
{
ra[k].u = i+1;
ra[k].v = j+1;
ra[k].val = m[i][j];
++k;
}
```

4.
```
for (i = 0; i < k; i++)
{
cout <<" < "<< ra[i].u;
cout <<"  "<< ra[i].v<<">";
cout <<"  "<< ra[i].val;
cout<<endl;
}
```

Problem 3.3: There are 10 items. Each item has weight and value. Find out the value of unit weight for each item.

1. Take an array of structures as follows.

```
struct
 {
float wt;          // weight of an item
float val;         // value of an item
float uval;        // value of unit weight
 } list[10];
```

2. Enter the weights and values of the items.
3. Compute the unit value of each item.

```
for (i = 0; i < 10; i++)
{
list[i].uval = list[i]val / list[i]wt; //find unit value.
}
```

4. Output all items of the array, list.

3.3 Return values using structure

Usually, a function can return only one value. Using a structure we can return more than one value. At first, let us see a simple function that returns an integer value.

```
int sum (int a, int b)
{
s = a + b;
return s;
}
void main ( )
{
int x, y;
cin >> x;
cin >> y;
cout << "see the summation:";
cout << sum(x,y);
}
```

Now we see a problem, where we have to return more than one value.

Problem 3.4: There are 10 students in a section of a course. Find out the student who obtained the highest marks along with his/her *id*, *name* using a function (student record contains *id*, *name*, and *marks*).

Algorithm 3.4: Pseudo-code for returning a structure (more than one value)

1. Declare an array of structure of student record:

```
struct stud
{
int id;
char name[40];
int marks;
}sa[10];
```

2. Find the student, who obtained the highest marks using a function:

```
stud maxf (stud a[ ])
{
stud max;
max.marks=a[0].marks
for (i = 1; i < 10; i++)
{
if(a[i].marks>max.marks) max = a[i];
}
return max;
}
```

4. Call the function under main ( );

```
void main ( )
{
```

a) Enter data for the students using an array, sa.
b) Find the student with highest marks using function maxf( ):

```
stud b;          //take a variable
b = maxf(sa);
cout << b.id;
cout << " " <<b.name;
cout << " " <<b.marks;
```

Comments: Here type of the function should be **stud**, since the function returns the value that has type **stud**.

Problem 3-5: There are 40 students in a class. Each student record contains *id*, *name*, and *marks*. Now arrange the data of the students according to descending order of the marks of the students (the student, who has got highest marks, will be on top of the data list).

Hints: to solve this problem we have to sort student records/structures using any sorting algorithm.

Algorithm 3.5: algorithm to sort data using structure

1. Declare an array of structures:

```
 struct stud
    {
      int id;
     char name[35];
```



```
      int marks;
     }sa[40];
```

2. Enter data for the students.
3. Sort the data of the students in descending order of their marks:

```
stud temp;
for (i=0;i<40; i++)
  {
large_ind = i;
for(j=i+1; j<40; j++)
  {
if(sa[j].marks > sa[lage_ind].marks) large_ind = j;
  }
temp = sa[i];
sa[i] = sa[lage_ind];
sa[large_ind] = temp;
  }
```

4. Output the list of the students' data in an array, sa[ ].

3.4 Two dimensional (2-D) array of structures

Suppose that there are **three** categories of members in a club. Such as associate members, members, and fellows. Each category has **four** members and we have to store id, name and mobile number of each member. Here we have to organize a record of each member in such a way that the members are associated with their respective categories. To store these type of records we can use a **2-D array of structures.** This array is as like as a 2-D array, but its elements will be structures. In Figure 3.3, c-1, c-2 and c-3 denote categories, here ***am-i*** is the *i*th member of an associate member, ***m-i*** is the *i*th member of member category and ***f-i*** means an ith member of the fellow category:

c-1	*am-1*	*am-2*	*am-3*	*am-4*
c-2	*m-1*	*m-3*	*m-3*	*m-4*
c-3	*f-1*	*f-2*	*f-3*	*f-4*

Figure 3.3: A 2-D array (each element is a structure)

Each member has three attributes such as *id*, *name* and *mobile number* which indicates each element of the above 2-D array contains three items. The pictorial view of each element is as follows:

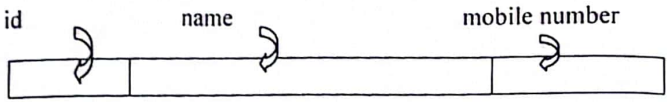


Figure 3.4: Pictorial view of each structure

We can store and data and display them in the following way.

```
struct st
        {
        int id;
        char name[30];
         int mobile;
        };
        st tb[3][4];
        int i,j;
        for(i=0;i<3;i++)
                for(j=0;j<4;j++)
                {
                        cin>>tb[i][j].id;
                        fflush(stdin);
                        gets(tb[i][j].name);
                        cin>>tb[i][j].mobile.
                }

  for(i=0;i<3;i++)
        {
                for(j=0;j<4;j++)
                {
                        cout<<" "<<tb[i][j].id;
                        cout<<" "<<tb[i][j].name;
                        cout<< " "<<tb[i][j].mobile;
                }
     cout<<endl;
```



```
        }
        cout<<endl;
        return 0;
}
```

3.5 Dynamic 2-D array of structures

Let us consider the example of a club described in the previous section. What will we do if each category has variable numbers of members? Such as there are five associate members, three members, and two fellows. In this case, to accommodate all the items of three categories we have to take a 2-D array of 3×5 size. However, if we take a 2-D array of structures with size 3×5 there will be wastage of space in memory. If each integer takes four bytes and each character takes one byte of memory space, each record/structure takes 4+30+4=38 bytes of memory space. So, the wastage of memory will be 2x38 + 3x38 = 190 bytes. This is only for small differences in the number of members. Such as if there are 20 associate members, 14 members, and five fellows, the wastage of memory will be more. However, we can use memory space efficiently by using a **dynamic 2-D array of structures**. We can sketch a figure for the above example as below.

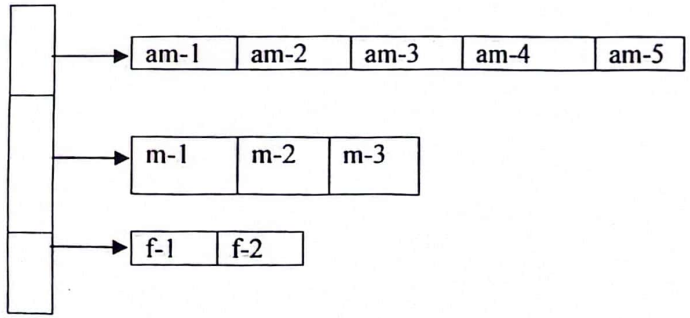


Figure 3.5: Dynamic 2-D array of structures.

We can store data using the following pseudo-code.

```
1. struct st
        {
```

```
    int id;
    char name[30];
    int mobile;
};
```

2. For three categories array of the pointer is as below.

```
st *tb[3];
```

3. Allocation of space in memory for each category (by considering the number of members).

```
tb[0]=new st[5];
tb[1]=new st[3];
tb[2]=new st[2];
```

4. Enter data for the three categories.

```
for(i = 0;i<3;i++)
{
    for(j = 0;j < 5; j++)  //for five members of first category
    {
    cin >> tb[i][j].id;
    fflush(stdin);
    gets(tb[i][j].name);
    cin>>tb[i][j].mobile
    }

    for (j=0;j<3;j++)  //for three members of second category
    {
    cin>> tb[0][j];
    fflush(stdin);
    gets(tb[0][j].name);
    cin>>tb[0].mobile;
    }

    for (j=0;j<2;j++)  //for two members of third category
    {
    cin>> tb[0][j];
```



```
    fflush(stdin);
    gets(tb[0][j].name);
    cin>>tb[0].mobile;
    }
} //end of for using i
```

Similarly we can display data using cout.

3.6 Difference between an array and a record

1. An array is a finite set of the same type of data items. 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 array is declared as using the data type used in it, its name and the size of the array, such as int A[10]. On the other hand, a structure is declared using keyword **struct**, names of the members along with their types and the name of the structure, such as

```
struct stud{
    int id;
    char name[30];
    int marks;
        };
```

3. Each element of an array is represented by a name of the array and index number of the element in the array. However, each element of a record (structure) is represented by a name of the structure along with the name of the element. Such as B[2] is an element of the array B and stud.id is an element of the structure stud.

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 type of data but a record is a set of different types of data items.

Questions:

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

Next, we will give some problems related to structure those should be solved by the student in lab classes

Structure or record related problems

Problem 3-1: Given three attributes of a 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 a grade for each student and display roll, name, marks, and grade in 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 an employee record such as ID, name and basic pay. Write a program using structure to store data for five 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

Sample output:

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: Given a matrix as follows:

	1	2	3	4	5
1	0	8	0	14	0
2	8	0	15	0	9
3	0	15	0	11	0
4	14	0	11	0	6
5	0	9	0	6	0

Print all the non-zero values of the above matrix along with row and column number as show below.

<1, 2> = 8, <1, 4> = 14, <2, 1> = 8, <2, 3> = 15, <2, 5> = 9 and so on.

If we denote the above matrix as m, we can see m[1, 2] = m[2, 1], m[1, 4] =m[4, 1]. Can you print only one value for this type of duplicate values such as you will print <1, 2> = 8 and <1, 4> = 14, **but not** <2, 1> =8 and <4, 1> = 14. Print all such values of the matrix.

Problem 3-5: Given a matrix as follows:

	1	2	3	4	5
1	0	8	99	14	99
2	8	0	15	99	7
3	99	15	0	11	99
4	14	99	11	0	6
5	99	7	99	6	0

Find out the minimum values from each row, where **zero** will not be the minimum value. That means, there will be five minimum values for five rows.

Problem 3-5: Implement algorithm 3-3 (see book).
Problem 3-6: Implement algorithm 3-4 for five students (see book).
Problem 3-7: Implement algorithm 3-5 for five students (see book).
Problem 3-8: Write a program to store data and display data for the problem stated under the sub-section dynamic 2-D array of structures (see Figure 3.5).

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 the linked list and Doubly linked list
- Describe the insertion process of a node into the 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 computer's memory. To store data in scattered locations in memory we have to make the link between one data item and another. So, each data item or element must have two parts. One is data part and another is a 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. The pointer part of a node is called an internal pointer.

The pointer which is not the part of a node is called an external pointer. To locate the list an external pointer is used to point the first node of the list. The link part of a node usually points the node that is next to it. The link part of the last node will not point any node. So, it will be *null*. This type of list is called a linear (one way) linked list or simple linked list.

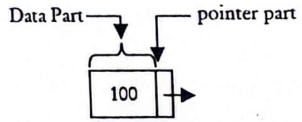


Figure-4.1 (a): A single node

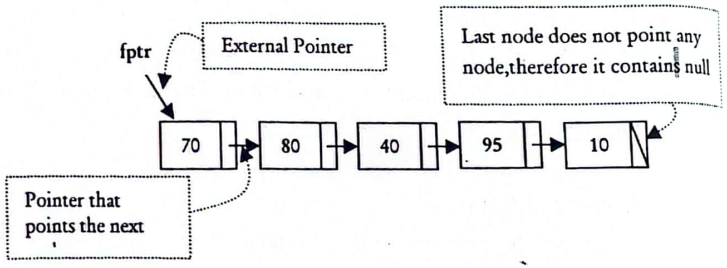


Figure- 4.1(b): Graphical representation of a linear linked list

To create a node we have to follow the steps:

1) Declare a structure of a node.

2) Allocate space in memory for the node.

3) Store data to the node.

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

1. Node Declaration:

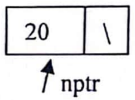
```
struct node
    {
    int data;
    node *next;
    };
```


2. Allocation space in memory:

```
node *nptr;      //we need a pointer
  nptr= new (node);
```

3. Data storing:

```
    nptr→data = 20;      //store 20 in data part
   nptr→next = NULL; //void the pointer part
```



4.1.2 Create a new node and enter data using a variable

1. Node declaration:

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

3. Allocate memory for new node:

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

4. Insert data to the node:

```
    int item;
    cin>>item;
    nptr→data = item;
    nptr→next = NULL;
```

4.1.3 Create a linear linked list

To create a linear linked list at first we have to create an empty list. T identify or locate the linked list an external pointer is used that points th first node of the list. For an empty linked list, this external pointer will no point any node, it means this pointer will be *null*. Next, we have to create new node. As it has been shown in the previous sub-sections to create a ne node an external pointer is required. The data part of the new node wi contain data (information) and the pointer part will be *null*. After creating node (first node) we assign (include) this node to the list or the extern pointer will point this node. For making the link between two nodes w have to use another external pointer and this pointer always (in the case creation of a list) will be with the last node. After that, we create anothe new (second) node. With the help of the external pointer to the last no and the external pointer to the new node, we make link between the la node and the new node. So, a link is established between the existing linke list and the new node. By repeating the process stated above any other no can be included in the list. Thus we can create a linked list.

We can create a linear linked list according to the process given 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 no will contain data and the pointer part will be *null*.

3. The external pointer points the new node. At this moment there is o one node in the list.

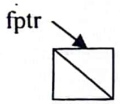
4. Create another new node and include the node to the linked list making link between the first node and the new node.

5. **Repeat the process** to include any other new node.

4.1.4 Linked list creation process using pseudo code and graphical view

1. Create an empty list, the pointer to the first node of the list will point nothing (null):

```
fptr=Null;
```

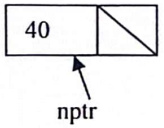


a) An empty list.

2) Create a new node with data:

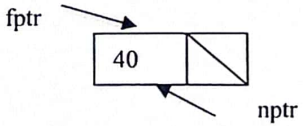
```
npt = new (node);
nptr-> data = item;
nptr ->next = NULL;
```

(Here the value of item is 40).



b) A new node with data

3) Include the first node in the list



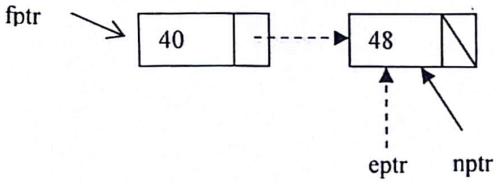

```
fptr = nptr;
eprt = nptr;
```

c) Inclusion of the new node to the list.

Here **eptr** is a pointer used to make the link between the end (last) node and the new node.

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

```
eptr -> next = nptr;
  eptr=nptr;
```



d) Creation and inclusion of the second node.

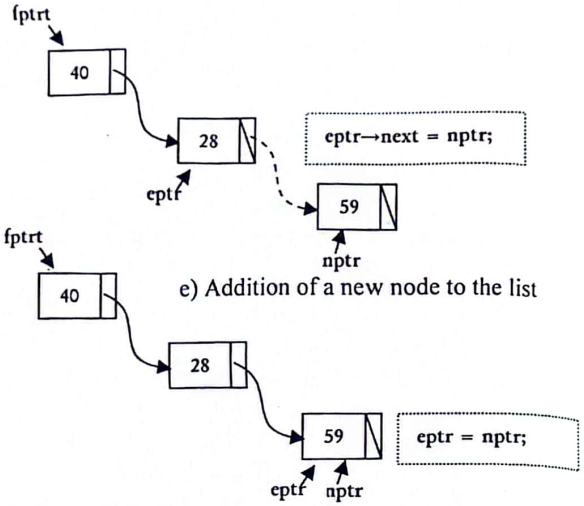


Figure-4.2: Creation process of a linked list (pictorial view)

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

1. Declare a node and pointers (fptr, eptr, nptr):
   i. struct node
   {
   int data;
   node *next;
   };
   ii. node *fptr, *eptr, *nptr;
2. Create an empty list:
   fptr = NULL;
3. Create a new node:
   nptr = new (node);
   nptr→data = item; //item is a variable
   nptr→next = NULL;

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

```
if (fptr = = NULL)     // when the list is empty.
        {
        fptr = nptr;
        eptr = nptr; }


else                          // when the list has node (s)
        {
        eptr→next = nptr;
        eptr = nptr;
        }
```

5. Output linked list

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

Comments: Here, **nptr** is a pointer to a new node. The **eptr** is the pointer that points the last (end) node of the list and **item** is a variable used to enter data into the new node.

Program to create a linked list:

Although we have written creation process of a linked list in detail, however, there are some students they face problem in writing a program to create a linked list. So, a program is coded to create a linked list (header files are not included). After creating a linked list the data should be read from the linked list and display them on the monitor. Otherwise, we will not understand whether the list is created properly. At the end of the program, the code to display data of the list has been added.

```
void main()
struct node
 {
int data;
node *next;
```

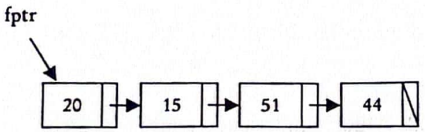
```
 };
int i, item;
node *nptr, *eptr, *fist; // Necessary pointers
fist = NULL;       // Create an empty list
cout<<"Enter number of nodes:";
for (i=1;i<=6;i++)
 {
     cin>>item;
     nptr = new (node);
     nptr->data = item;


     nptr->next=NULL;
if (fptr==NULL) //List is empty
          {
          fptr = nptr; //Include first node to the list
          eptr = nptr; //eptr is at the last node
          }
//Inclusion (addition) of more node
else
{
     eptr->next = nptr; //make link
     eptr = nptr;
}
}//end of for loop
//Display data
eptr = fptr;
for (i = 1; i <= 6; i++)
{
```



```
cout<<endl;
cout<<eptr->data;
tptr = eptr->next;
cout<< "";
}
cout<<endl;
cout<<endl;
}
```

4.1.5 Locate or search a node of a linked list

To locate or find out a node of a linked list value or data of the node must be known.



Problem 4.1: Find out the item 51 from the above linked list.

Description of Solution: To locate the node we have to traverse the list using a pointer. Here, a temporary pointer will be used to find the node.

Operations to be done:

1) Take a temporary pointer (tptr) to locate the node.

2) Assign the pointer to the first node of the list to this temporary pointer (tptr = fptr).

3) Compare the data of the node pointed by the temporary pointer with the data of the node to be located.

4) Move the pointer to the next node until the node is found.

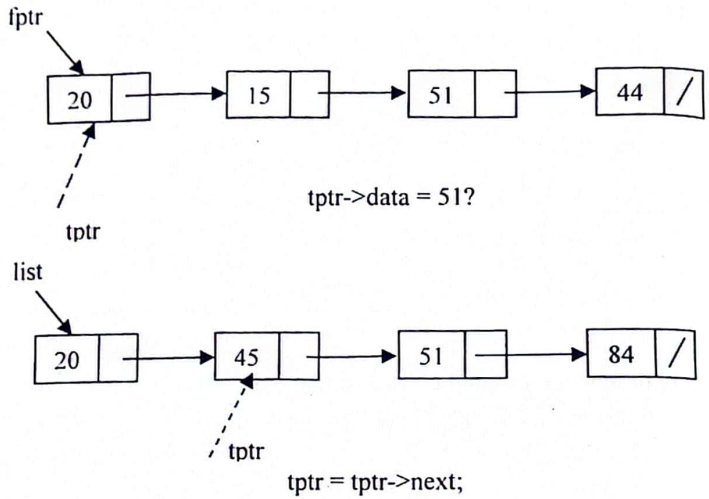


Figure-4.3: Pictorial view of searching (locating)

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

1. Declare **node**

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

2. Take a temporary pointer, **tptr**
3. Take the value to be located in a variable:

```
item = 51;
```

4. Search the item:

```
tptr = fptr;
while (tptr→data != item or tptr->next! = NULL)

        {
        tptr = tptr→next;
        }
```

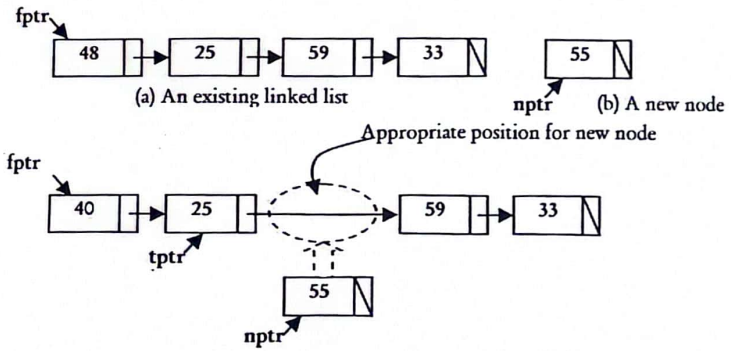

5. Output:

```
if (tptr→data = = item)  print "FOUND"

else print "NOT FOUND"
```

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

4.1.6 Insert a node into a linked list

To insert a (new) node into a linked list, we have to find out (locate) the position of the node first. For this, the data of the two nodes must be known in-between which the new node should be inserted. To locate the position we can use the searching (locating) process described in the previous sub-section. In this case, we can locate the node after which the new node will be inserted. After locating the node, links should be established to perform the task of insertion.



(c) Finding out appropriate position for the new node

Figure-4.4 Finding position for the new node

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

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

2) To perform insertion by making necessary links.

Suppose that we want to insert a node after the node with data 25

To locate 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 = fptrt**).

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

iii) Traverse the temporary pointer until we find node value after which the new node will be inserted. (**tptr = tptr -> next**).

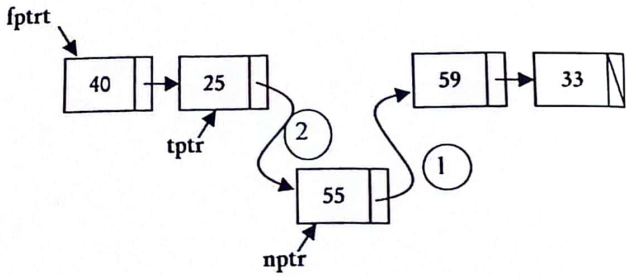
To insert the node by making the link the steps are:

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

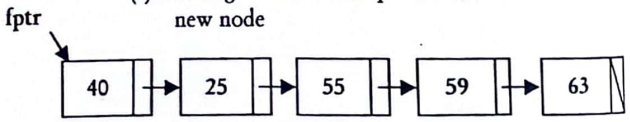
ii) Point the new node by the previous node of the new node.

tptr->next= nptr

1. nptr->next = **tptr->next;**
2. tptr->next = **nptr;**



(e) Making link between tptr and the new node



(f): Updated list

Figure-4.5: New node insertion in an existing list (pictorial view)

Algorithm 4.3: Algorithm (pseudo code) to insert a node into a linked list

1. Declare a node and pointers

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

```
tptr = fptr; //fptr is the pointer to the first node of the list.

//Suppose that we will insert the node after the node with data 25
   while (tptr→data !=25)
            {
            tptr = tptr→next;
            }
```

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

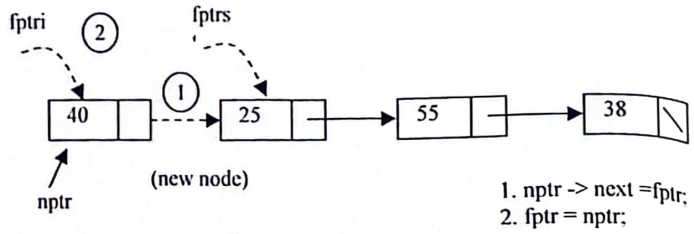
```
nptr→next = tptr→next;
tptr→next = nptr;
```

6. Output: An updated linked list

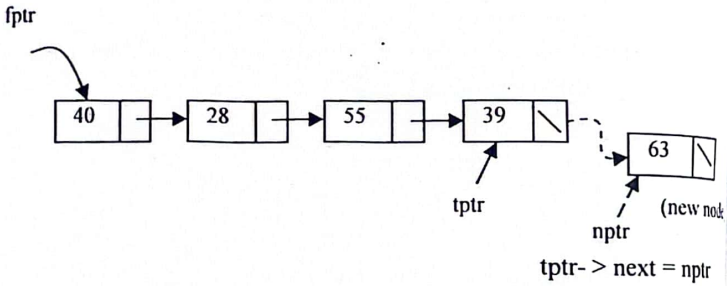
Comments: Here, *fptr* is an external pointer that points to the first node of the list, *tptr* is a temporary pointer and *nptr* is the pointer to the new node.

The addition of a node before the first node of a linked list is very easy. At first, create a new node. Make the link between the new node and the first node of the list. Now set the pointer to the first node to 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 the link between the last node of the list and the new node.



a) The addition of a new before the first node.



b) The addition of a new after the last node.

Figure-4.6: Pictorial view of node addition

4.1.7 Deletion of a particular node

To delete a particular node from a linked list, we have to locate the node to be deleted. Then the necessary links must be updated before performing the deletion operation. We can locate or find the node using the searching method described above. However, to update the links a pointer is needed that will point the node before the node to be deleted. Otherwise, we cannot update the necessary links. For this two temporary pointers will be used

One will locate the target node and other will point the node just before the target node (node to be deleted). Now, we make the link between the previous node and the next node of the target node. That is we have to take the next pointer of the previous node to the next pointer of the target. Now the link has been established and we can delete the target node.

Deletion operation at a glance :

1) Locate the target node. For this, we use two temporary pointers. One pointer will be at the previous node of the target node and another pointer to the target node.

2) Update necessary links by making the link between the previous node and the next node of the target node.

3) Delete the target node.

Let us see the pictorial view for a deletion operation. Suppose that the target node (node to be deleted) has the data 55.

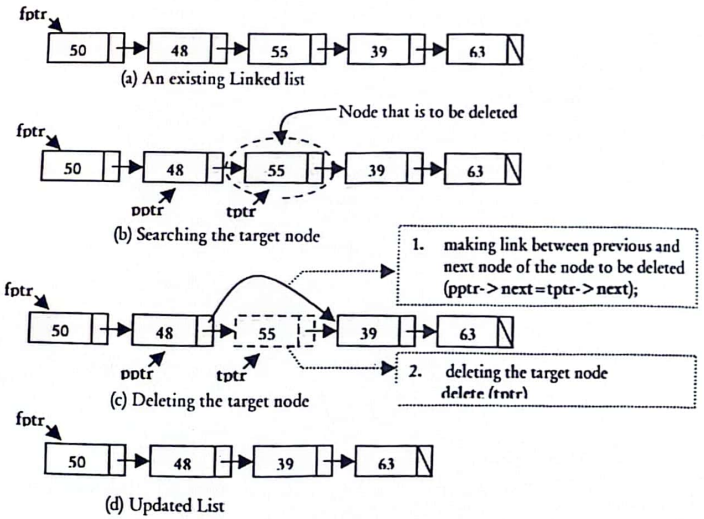


Figure-4.7: Deletion of a node from a linked list (pictorial view)

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 can 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 (last) node. Then we delete the node.

Problem 4.2: Write an algorithm to 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:

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

2. Input linked list and the item (that is to be deleted)
3. Search the item to be deleted from the list:

```
tptr = fptr;

while (tptr→data != item)      // pptr and tptr are two
        {pptr = tptr;          // auxiliary pointers
         tptr = tptr→next; }
```

4. Update links:

```
(Make link between previous and next nodes of the target
node)
pptr→next = tptr→next;
```


5. Delete the node:

```
delete (tptr);
```

6. Output: Updated linked lists

Comments: In the above algorithm, **fptr** is a pointer to the first node, **pptr** is a pointer that points the previous node of the target node (the node is to be deleted) and **tptr** is a pointer that points the target node. Here **item** is a variable where we hold the data of the target node.

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. Show node structure:

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

3. Take assisting pointers: node *pptr, *tptr;.
4. pptr = fptr;
5. while (pptr != NULL)

```
{
tptr = pptr→next;
```

6. while (tptr != NULL)

```
{
if (pptr→data > tptr→data)

{
  interchange (pptr→data, tptr→data);
 }
tptr = tptr→next;
} //end of while of step-6
```

```
        pptr = pptr→next;
        } //end of while of step-5
```

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

Comments: *fptr* is the pointer to the first node of the linked list. *pptr* and *tptr* are the pointers to first and second nodes, and these pointers have been used to arrange (sort) data.

Problem 4-4: There is a linked list of *n* nodes. Split the list into two linked lists so that first list contains *n/2* nodes and second list contains n/2 nodes. If *n* is odd the first list will contain *(n/2+1)* nodes.

Algorithm 4-I: Algorithm to split a linked list into lists.

1. Show node structure:

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

```
2. Input a linked list of n nodes
3. i=1; k = n/2;
4. if (n%2 = = 1)
   k = k+1;
5. tptr = fptr;  list1 = fptr;
6. while (i < k)
{
tptr=tptr->next;
++i;
}
7. list2 = tptr->next;
   tptr->next = NULL;
```

8. Output two linked lists (one with a pointer, *list1* at the first node and another with a pointer, *list2* at the first node of the second list).

4.2 Representation of polynomial using linked list

Let us see a polynomial $f(x) = 10x^8 + 3x^6 - 4x^3 + 8$. We can represent this polynomial as shown in Figure 4. 8 following linked list.

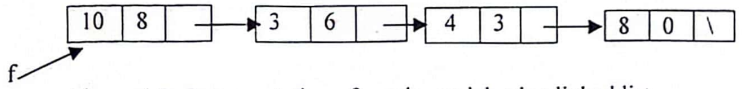


Figure 4.8: Representation of a polynomial using linked list

The node declaration of the polynomial as follows.

```
struct node
 {
int coef;
int exp;
node *next;
 }
```

Now we see how we can add two polynomials. Suppose, there are two polynomials such as, a $f_1(x) = 10x^9 + 3x^6 - 6x^3 + 10$ and $f_2(x) = 12x^8 + 7x^6 + 4x^3 - 7$.

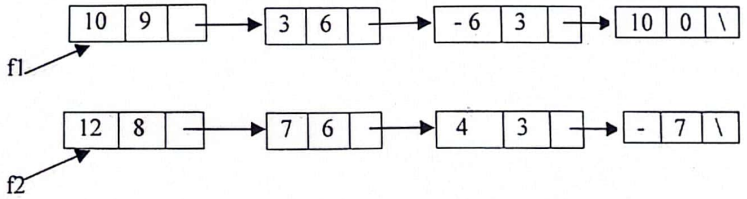


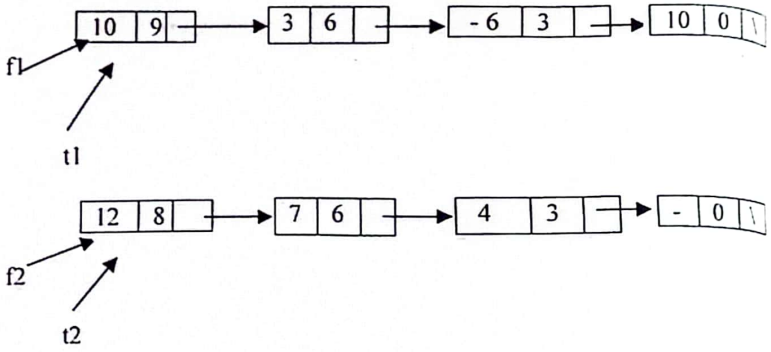
Figure 4.9: Representation of two polynomials using linked list

According to rules of the addition of two polynomials, we have to add the coefficient if the exponent or power is same. As for example, for the above two polynomials the results will be as follows :

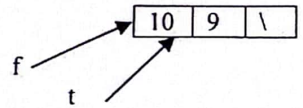
$f(x) = 10x^9+12x^8 + (3+7)x^6 + (-6+4)x^3 + (10-7)$

$f(x) = 10x^9 + 12x^8 + 10x^6 -2x^3 +3$

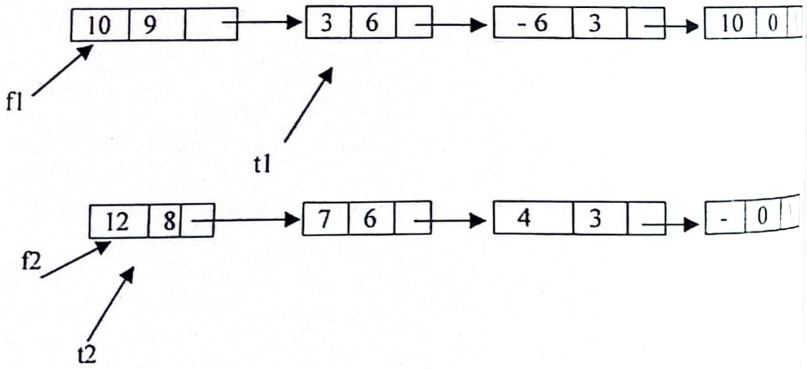
Now we find a polynomial as f(x) = f1(x) + f2(x). The process of addition can be shown in the following pictorial view. Take two temporary pointers t1 and t2 as t1=f1 and t2=f2. The pointer f is the first pointer for f(x).



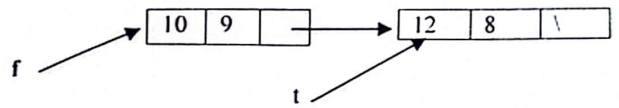
if (t1->exp > t2->exp) copy the data from node with t1 to f and t1 moves forward.



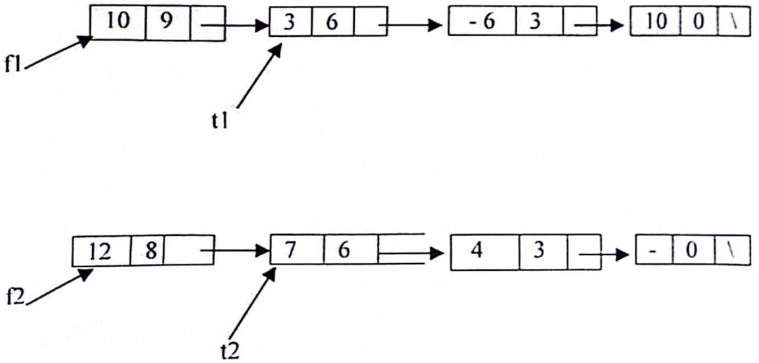
a) Creation of the first node of polynomial f(x).



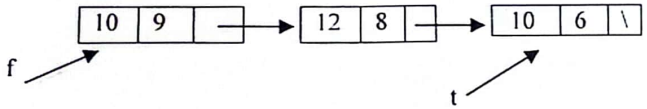
if (t1->exp < t2->exp) copy the data from node with t2 to t.



b) First two nodes for f(x).



if (t1->exp = = t2->exp) t->coef = t1->coef + t2->coef and t->exp = t1->exp .



c) Creation of the third node of f(x)

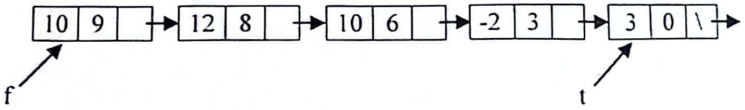


Figure 4.10: Pictorial view of the addition of two polynomials

Let us see the pseudo-code to add two polynomials. There are some pointers used in the code such as f1 and t1 used as pointers for the first polynomial; f2 and t2 are the pointers used for the second polynomial; f and t are used for the polynomial, which contains the sum (**See Figure 4.10.**) Node and pointers declaration are as follows.

```
strut node
 {
 int coef;
  int exp;
  node *next;
 };
node *f1, *f2, *f, *t1, *t2, *t.
```

Algorithm 4.6: Algorithm for Addition of two polynomials

1. Input a linked list for polynomial f1(x)
2. Input another linked list for polynomial,f2(x)
3. Create a thirst list, which is the sum of f1(x) and f2(x) as below.

```
while(t1!=NULL&&t2!=NULL)
{
        nptr=new(node);          //for new node
        nptr->next=NULL;
        if(t1->exp= =t2->exp)    //exp of two polynomials are equal
        {
  nptr->coef=t1->coef+t2->coef;
  nptr->exp=t1->exp;
         t1=t1->next;
         t2=t2->next;
        }
       else if (t1->exp > t2->exp)   //exp of first polynomila is greater,
       {
                nptr->coef=t1->coef;
                nptr->exp=t1->exp;
                t1=t1->next;
                        }
      else                           //exp of second polynomail is greater
       {
 nptr->coef=t2->coef;
                nptr->exp=t2->exp;
                t2=t2->next;
```



```
        }

    if(f==NULL)              //for the first node of the final polynomial
        {
                f=nptr;
                t=f;
                }
        else {               //for other nodes
                t->next=nptr;
                t=nptr;
                }
} //end of while loop;


while(t1!=NULL)   //When there are some nodes in the first polynomial,
yet which are not considered
 {
          nptr=new(node);
          nptr->coef=t1->coef;
          nptr->exp=t1->exp;
          nptr->next=NULL;
          t1=t1->next;
          t->next=nptr;
          t=nptr;
}
while(t2!=NULL)  //When there are some nodes in second polynomail,
yet which are not considered
{
          nptr=new(node);
          nptr->coef=t2->coef;
          nptr->exp=t2->exp;
          nptr->next=NULL;
          t2=t2->next;
          t->next=nptr;
          t=nptr;
}
```

4.3 Doubly linked list

Using linear linked list we can move forward only. If it is necessary move backward it is not possible in a linear linked list. When we have move forward and backward doubly or two-way linked list is necessary.

4.3.1 Definition

A doubly or two-way linked list is a list where each node has three pa One is link or pointer to the previous (backward) node, one is data part hold the data and another is link or pointer to the following (forward) no Like a linear linked list, there is an external pointer to the first node of list. The doubly linked list is also known as a two-way linked list.

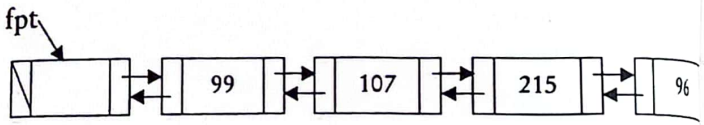
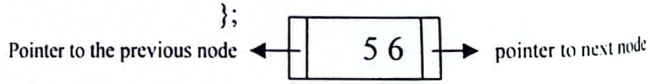


Figure-4.11: Graphical representation of a doubly linked list

4.3.2 Declare a node of a doubly linked list

Node declaration is similar to node declaration in a linear linked list, onl pointer is extra.

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



4.3.3 Create a node with data

i) Declare a node:

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

ii) Allocate space for the node:

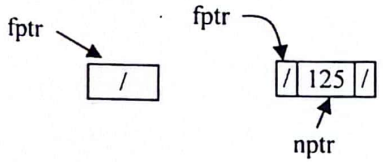
```
node *nptr;
nptr = new (node);
```

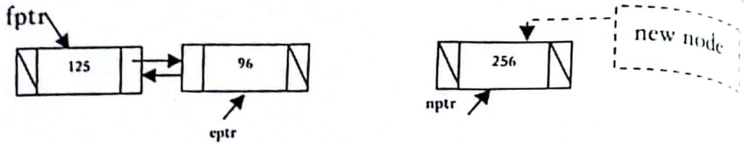
iii) Store data to the node:

```
nptr→back = NULL;
nptr→data = item;   //item is a variable
nptr→next = NULL;
```

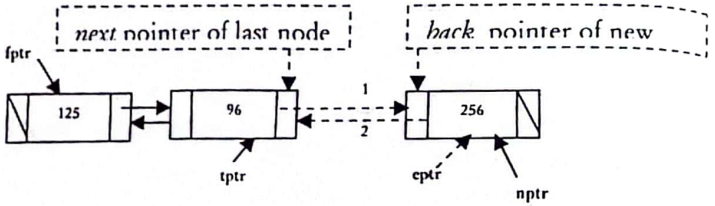
4.3.4 Create a doubly linked list

To create a doubly linked list, we have to create an empty linked list first. Then a new node is created with data and the node is included in the list. After that, we shall create another new node with data and make the link between the last node of the list and the new node. To make a link, the ***next*** pointer of the last node points the new node and the ***back*** pointer of the new node points the last node of the list. By creating the necessary number of nodes and making links we can create a doubly linked list. The pictorial view of this creation process is shown in Figure 4.12.

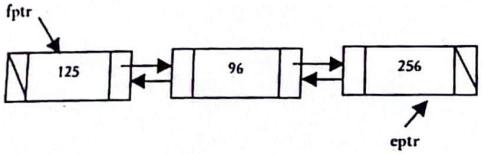




(a) A doubly linked list and new node



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



(c) Doubly linked list after making link

Figure-4.12: Pictorial view of creation process of a doubly linked list

Algorithm 4.7: 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 *fptr, *eptr;
```


2. Create an empty list:

```
fptr = 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 (fptr = = NULL)
        {
        fptr = nptr; //fptr points the first node
        eptr = nptr; // eptr is at the last node
        }
else    {
 eptr→next = nptr;  //make link to the new nodes
        nptr→back = eptr; //make link to the previous node
        eptr = nptr;
        }
```

5. Output: a doubly linked list.

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

4.3.5 Insertion of a node into a doubly linked list

To insert a new node into a doubly linked list, we have to locate the position in the list where the new node should 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 perform searching we set the temporary pointer to point the first node. Here we must know the values of two nodes in-between which the new node should be inserted. By comparing the data of the node after which the new node has to be inserted and the data of the new node the position can be located.

Now, we shall make link among the node after which the new node will be 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 inserted). First, we have to make the link between the node after which the new node will be inserted and the new node. Then we shall make the link between the new node and the node before which the node will be inserted. After making links, insertion process terminates.

Let us see how we can add a new node before the first node of a linked list. For this, we do not have to locate the position (it is clear). So, to solve the problem we do the following operations :

1) Create a new node with given (necessary) data.

2) Update the necessary links :
   i) Take next link of the new node to the first node of the list.
   ii) Take back link of the first node to the new node.
   iii) Take the pointer at the first node (fptr) to the new node.

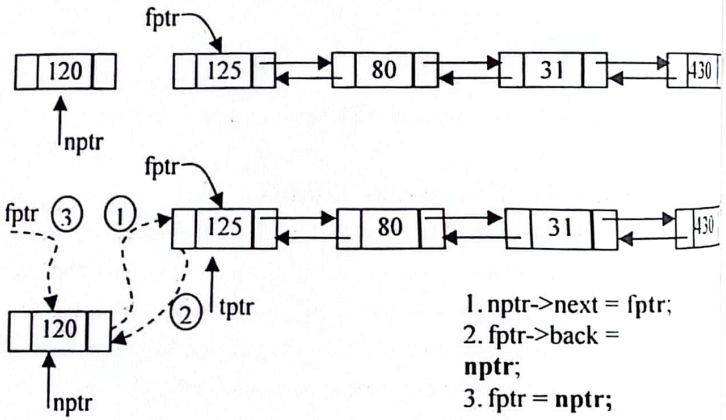


Figure-4.13: Addition of a new node before the first node (pictorial view)

Algorithm 4.8: Algorithm (pseudo code) to add a new node before the first node.

1. Input a doubly linked list
2. Show node structure:

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

3. Declare necessary pointer : node *nptr, *tptr;
4. Create a new node:

```
nptr = new(node);
nptr->data = item; //item is variable for to hold data
nptr->next = NULL;
nptr->back = NULL;
```

5. tptr = fptr;
6. Make Necessary links:

```
nptr->next = fptr;
fptr->back = nptr;
```

7. Move pointer (fptr) to the new node:

```
fptr = nptr;
```

8. Output: updated linked list

Now we see the addition of a new node after the last node of the list.
For this we have to do the following operations:
1) Create a new node with necessary data.
2) Locate the last node of the list.
3) Update the necessary links as follows:
i) Take the next pointer of the last node to the new node.
ii) Take the back pointer of the new node to the last node of the list.

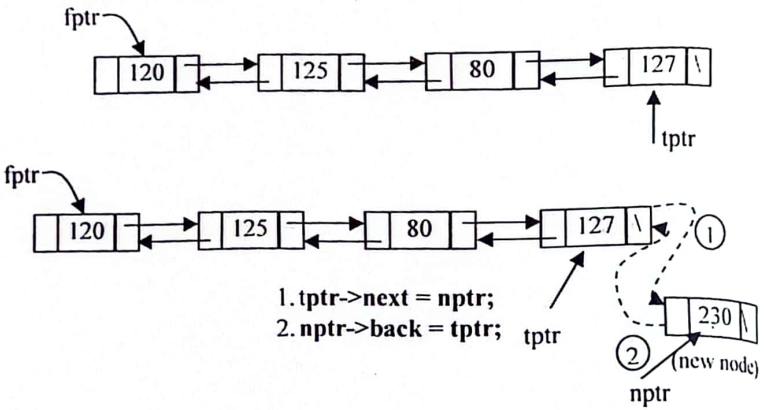


Figure-4.14: The addition of a new node after the last node (pictorial view).

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. Show node structure:

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

3. Declare necessary pointer : node *nptr, *tptr;
4. Create a new node:

```
nptr = new (node)
nptr->data = x;  //x is a variable
nptr->next = NULL;
nptr->back = NULL;
```

5. Locate the last node of the list:

```
tptr = fptr;  //fptr is the pointer to the first node of the list.
while (tptr->next != NULL)
{
tptr=tptr->next;
}
```

6. Make necessary links:

```
tptr->next = nptr;
nptr->back = tptr;
```

7. Output updated linked list.

After updating the doubly linked list, we have to print (display) the data of the list in **input order** as well as **reverse order**. If we find a correct print ensures out of input and reverse orders, it ensures that the addition operation has been done properly.

Now let us see the process of insertion of a node in-between two nodes of a doubly linked. To solve this problem we have to perform the following operations:

1) Create a new node with given (necessary) data.
2) Locate the node after which the new node will be inserted.
3) Update the necessary links as follows:

i) Take the next pointer of the new node to the node before which the new node will be inserted.

ii) Take the back pointer of the new node to the back node of the new node.

iii) Take the back pointer of the next node (the node before which the new node will be inserted) of the new node to the new node.

iv) Take the next pointer of the previous node of the new node to the new node.

Insertion process of a node into a doubly linked list is shown in Figure 4.15. In the figure dashed arrow lines along with numbers show the linking operations.

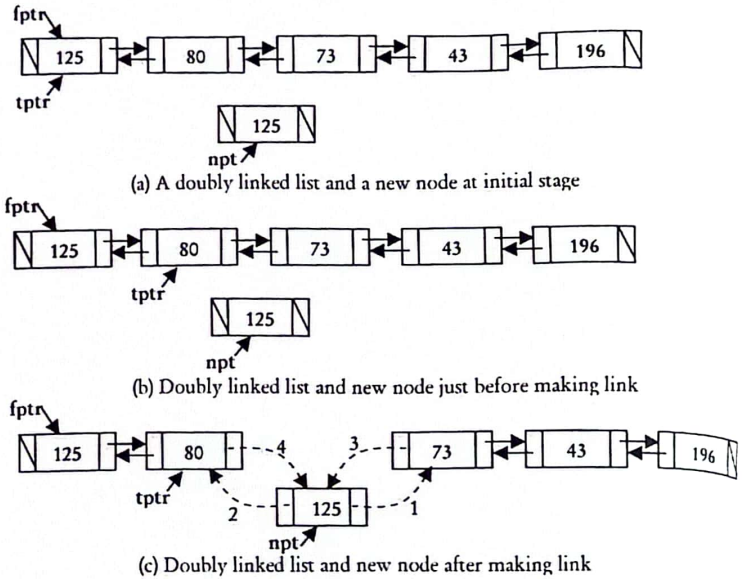


Figure-4.15: Insertion of a node into a doubly linked list (pictorial view)

Algorithm 4.10: Algorithm (pseudo code) to insert a node in between two exiting nodes of the list.

1. Input a doubly linked list.
2. Show node structure:

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

3. Declare necessary pointer: node *nptr, *tptr;
4. Declare necessary pointers (tptr, nptr)
5. Create a new node:

```
nptr = new (node);
nptr->data = x;
nptr->next = NULL;
nptr->back = NULL;
```


6. Locate (search) the position the new node:

```
tptr = fptr;
while (tptr->data != item)
    {
      tptr = tptr->next
    }
```

5. Make necessary links:

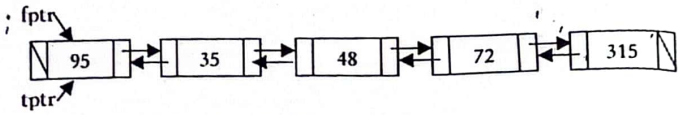
```
nptr->next = tptr->next;
nptr->back = tptr;
tptr->next->back=nptr;
tptr->next = nptr;
```

7. Output updated linked list.

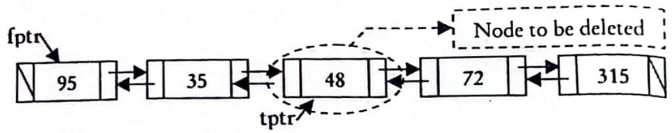
Comments: If we want of verify whether the doubly linked list has been updated properly or not we have to print (display) the data from the list in **input order** as well as **reverse order**.

4.3.6 Deletion of a node from a doubly linked list

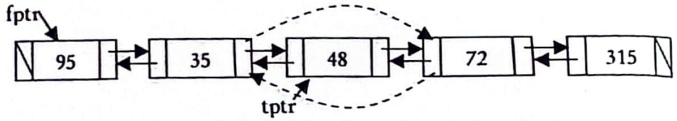
To delete a node from a doubly linked list, at first we have to locate the node to be deleted. For this, a temporary pointer is used to point the node. When the temporary pointer points the target node, before deleting the node we have to update links between the previous node and the next node of the 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 links we shall delete the target node. The pictorial view of the deletion process is shown in Figure 4.16. In the pictorial view, the deletion process of the node with data 48 has been shown.



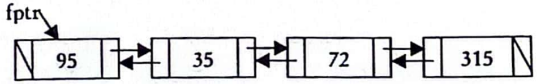
(a) Doubly linked list at initial stage



(b) Doubly linked list just before deletion of a node (with value



(c) Doubly linked list after making link and before deletion of the



(d) Updated doubly linked list after deletion of target node

Figure-4.16: Deletion of a node from a doubly linked list (pictorial view)

Algorithm 4.11: Algorithm (pseudo code) to delete a node from a doubly linked list

1. Show node structure:

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

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

```
tptr = fptr;
item = 48;

while (tptr→data != item)
{
```



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

5. Delete the target node:

```
delete (tptr);
```

6. Output: Updated doubly linked list.

The deletion processes of the first node and the last node of a doubly link list are easy and students can try themselves.

4.4 XOR linked list

A 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 points.

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

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

A 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 for 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 of items {...B C D...} and with R1 and R2 be the 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 when 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.

4.5 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. A circular linked list with a linear diagram shown in Figure 4.17

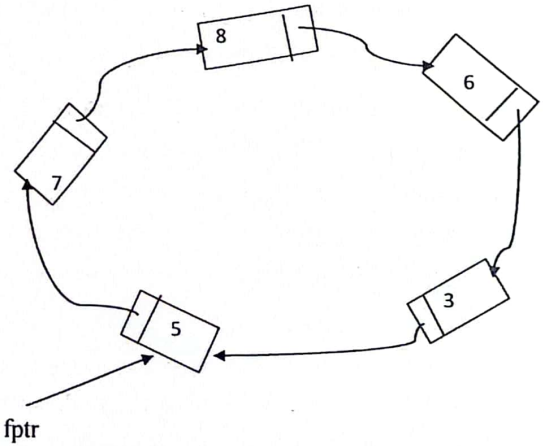


Figure-4.17: Pictorial view of a circular linked list (circular diagram)

A circular linked list with a linear diagram shown in Figure 4.18

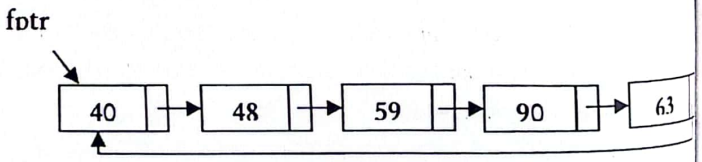


Figure-4.18: A circular linked list (linear diagram)

4.5.1 Create a circular linked list

The creation process of a circular linked list is similar to the creation process of a linear link list, which had been discussed in section 4.1.3. Here we have to do one addition thing, that is a link between the last node and the first node which will create a circular linked list.

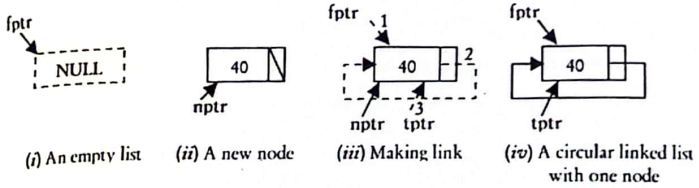


Figure-4.19: Linking process of the first node of a circular linked list

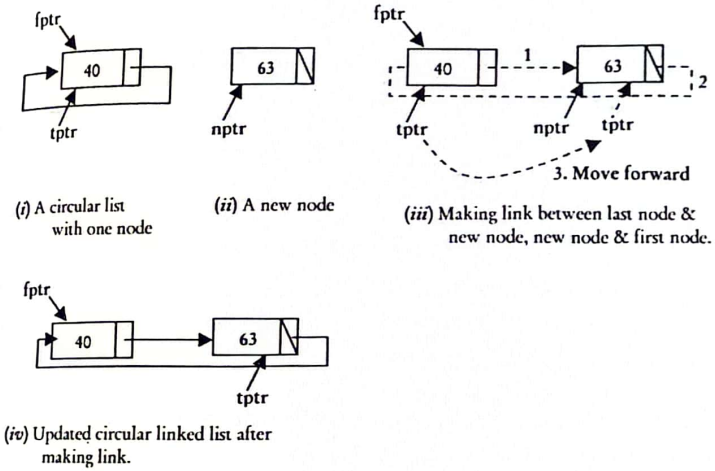


Figure-4.20: Creation of a circular linked list with more than one node

Algorithm 4.12: Algorithm to create a circular linked list

1. Declare node and necessary pointers:

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

```
};
node *fptr, *tptr, *nptr
```

2. Create an empty inked list

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

5. Output a circular linked list.

Note: To create a circular linked list with several nodes we have to repe the steps 3 and 4.

4.6 Difference between array and linked list

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

   Whereas, a linked list is a list or collection of data items (nodes) stor in scattered memory locations. Each data item has two parts. One is d

part and another is a 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.

2. Array implementation depends on the size and it results in wastage of memory space. On the other hand, the linked list does not depend on size.
3. Types of the 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 cannot be accessed directly and access time is linear and not so efficient.
5. The array is a static data structure and a linked list is a dynamic data structure.

4.7 Comparison of operations using an array and linked list.

Array	Linked list
Element access is fast if the index is known.	Element access is slow.
Insertion and deletion operations are slow.	Insertion and deletion operations are fast.
Element search is fast, if index is known, otherwise 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 to each other. Data items are known as nodes. Operations related to a linked list are: creation of a linked, addition or insertion of a node, deletion of a node.

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 linked 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. The array is a static data structure whereas the linked list is a dynamic data structure etc.

Questions:

1. What is a linear linked list? Explain with example(s).
2. Describe the creation process of a linked list.
3. How can a node be inserted into a linked list? Explain with example.
4. Write an algorithm to create a linear linked 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 the linked list are 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 an array?
13. Define doubly linked list with an 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 in 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 and another function for display the data. Call them to show the result of the 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 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) 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 node creation, a function for addition and another function to display the data.

Problem 4-5: Write a program to delete a node which is 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 a linked list.
ii) Delete the node which is in-between two existing nodes.
iii) Delete the last node of the list.
iv) 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: Create two linked lists, one with five nodes and another with four nodes. Merge them to make a single linked list.

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 the 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 which is 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 the doubly linked list for the followings:

i) Delete the first node of the list.
ii) Delete a node which is 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.

Problem 4-13: Remove (delete) the nodes with duplicate values from an existing list. Hints: Create a list that contains nodes with data 15, 20, 25, 25, 30, 35, 40, 40, 45, 50; you have to delete one node with data 25 and another node with data 40.

Problem 4-14: Create a linked list of random data such that there are several duplicate data. Now delete the nodes with duplicate data without sorting the data of the list.

Problem 4-15: Create linear linked list. Print the data of the linked list in reverse order using following the conditions:
1) You are free to use any other data structure.
2. You cannot use any other extra data structure (you will use only linked list).

Problem 4-16: Create a linear (one way) linked list and print the data. Now do the followings:
1) Change the data of the linked list such that present list contains the data which are reverse order of the initial set of data.
2) Can you solve the above problem without using any other extra data structure?

Problem 4-17: Create two linked lists for two polynomials, f1(x) and f1(x) as given below. Create another linked list, which will be the summation of the two polynomials as f(x) = f1(x) + f2(x).

$f1(x) = 10x^9 + 3x^6 - 6x^4 + 10x^2 + 4x + 2$

$f2(x) = 12x^8 + 8x^6 + 4x^3 - 7x^2$

Problem 4-18: Create a linked list, which will be a product (multiplication) of two polynomials used in the above problem.

Problem 4-19: There are two linked lists with sorted data, one with six nodes and another with four nodes. Create a third list that will contain 10 nodes of sorted data; however, you cannot sort the data of the third list. Hints: consider a linked with data 3, 7, 10, 15, 19, 35 and another linked list with data 2, 6, 12, 17. Now create a list, which will contain the data 2, 3, 6, 7, 10, 12, 15, 17, 19, 35. You have to create the third linked list of sorted data **without using any sorting algorithm**.

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
- Applications of stack and write related algorithms

STACK

5.1 Definition

In our daily life, we see stack (pile) of plates in cafeteria or restaurant, a stack of books in book-shop. Even a packet of papers is also a stack of paper-sheets. 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 the 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. The stack is a LIFO (Last In First Out) structure. It means the element which was added last will be deleted or accessed first.

The elements of a stack are added from bottom to top, which indicates *push* operation is performed from bottom to top. The elements are deleted from top to bottom, so we can say the *pop* operation is performed from top to bottom.

A stack can be implemented in any of two ways; using an array or using a linked list.

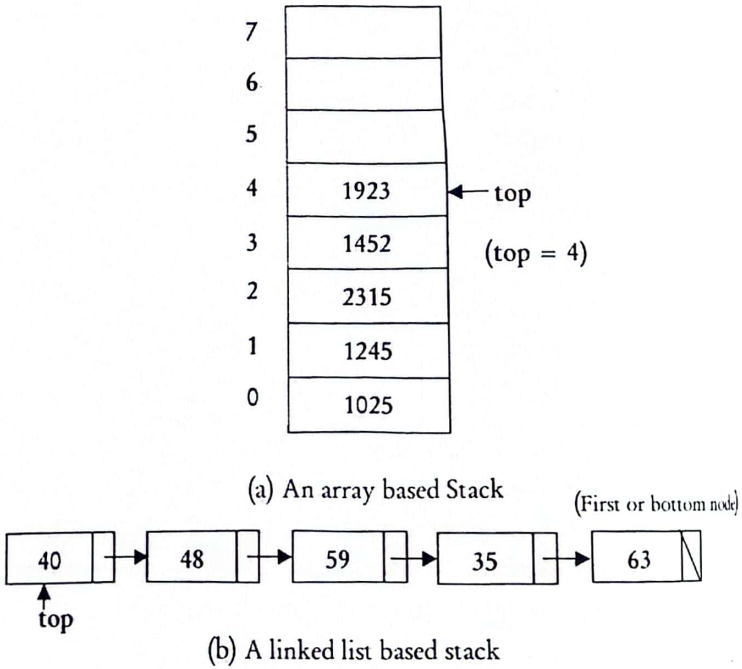


Figure-5.1: Graphical representation of a stack

5.2 Array based stack

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

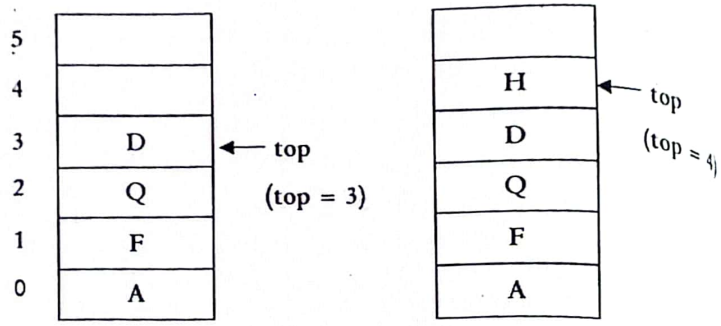
As for example if we want to create a stack of 10 integers we can define it as follows:

```
int stack[10];
```

The above declaration indicates array based stack, as like as an array.

5.2.1 Push operation

Push operation means to add an element at the top of the stack. Here, we shall use the array based stack, so, an array will be treated as a stack. We need an indicator or index identifier to add an element onto the stack and this indicator will mark the top element 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 cannot add any element to it, otherwise, we can. To indicate the top element an indicator (a variable), the *top* is used and *item* is a variable used to add an element to the stack. If *M* is the size of the stack (array) an overflow occurs when there are M elements in the stack and we try to add any other element to it. Let us see the following figure. In the figure, there is a stack of characters. The size of the stack is six and there are four elements in the stack. Since we use array according to C/C++, the value of the *top* indicator is zero for the first element and in Figure 5.2 (a) its value is 3 or *top = 3*. After pushing an element the value of the *top* is 4 or *top = 4* (Figure 5.2 (b)). When an element is added to the stack, at first the value of the *top* indicator must be increased and after that, we can push (add) an element.



(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. Input an array based stack;
2. Add an item in the stack:

```
if (top < M-1)
        {
        top= top +1;          //increase the value of top.
        stack [top] = item;  //put the element at the top

}
        else print "Over Flow";
```

3. Output: An updated stack.

The above algorithm is for the addition (push) of only one item. Now let us see how we can create an array based stack and add several elements. Here we will create a stack of four characters using C++ code.

```
char stack[10];
int top = -1;
char item;
while (top < 4)
```

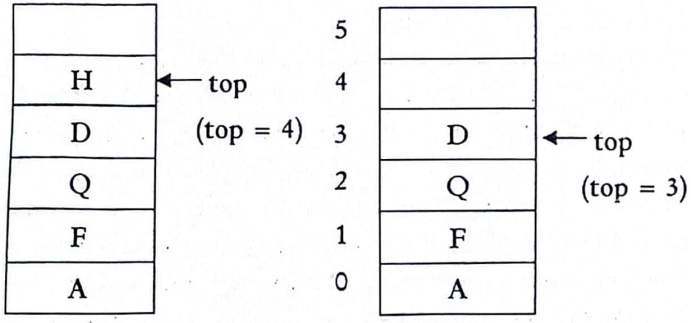


```
{
    cout<<"Enter a character: ";
    cin>>item;
    top=top+1;
    stack [top] = item;
}
```

After creating a stack of several elements we can push any element using **Algorithm 5.1**.

5.2.2 Pop Operation

The pop operation means to delete (access) an element from a stack. Here, the *top* is an indicator that holds the index of the *top element* of the stack. *M* is the size of the array and *x* is a variable where we access a top element from the stack. In the case of the pop operation, we must access the top element first and then decrease the value of top (top = top - 1).



(a): Before pop (b): After pop

Figure-5.3: Pictorial view of pop operation

Algorithm 5.2: Algorithm to delete an element from a stack

1. Input an array based stack.
2. Access the top element:

```
if (top < 0) print "stack is empty";
```

```
else
{
        x = stack[top];
        top = top - 1;
}
```

3. Output updated stack.

For accessing several elements we have to call the above algorithm within a loop.

5.3 Link based stack

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

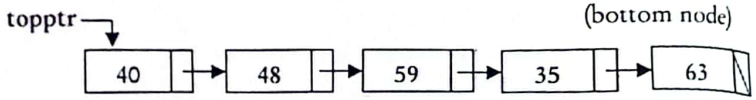


Figure-5.4: A linked list based stack

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. The name of this pointer is **topptr**. We create an empty stack first. So, the *topptr* (pointer) will be null. Then a new node has to be created. From the chapter 4, we already know how to create a new node. After that, we make the link between the top and the new node. At this stage, the *topptr* points the new node. To add the second node (another new node), we make the link between the new node and top node of the stack. To make the 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 *topptr* will point the new node. By repeating the process of new node creation and making link it with the top node of the stack we can create a link based stack. The algorithm (pseudo-code) to create a linked list based stack is given in Algorithm 5.3.

Here we can observe that we add (push) a new node before the top node of the stack.

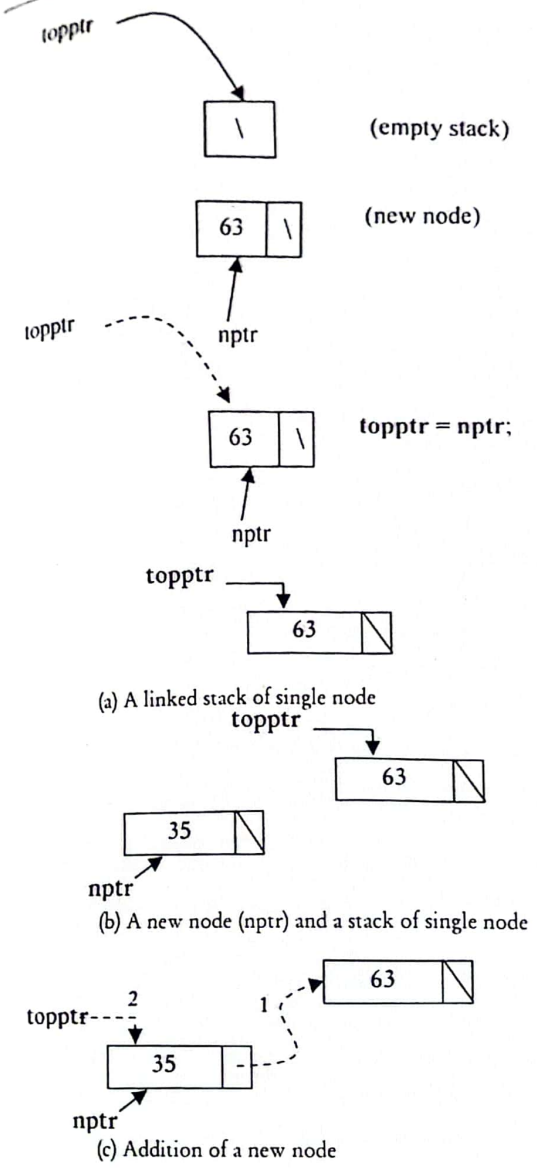


Figure-5.5: Pictorial view of a linked stack (creation process).

Algorithm 5.3: Algorithm to create a linked stack

1. Declare node and pointers:

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

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

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

4. Make link between stack and new node:

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

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

Comments: *topptr* 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 was discussed when we create a linked stack. The algorithm (pseudo-code) to push a node to a linked stack is shown in Algorithm 5.4. The pictorial view is depicted in Figure 5.6.

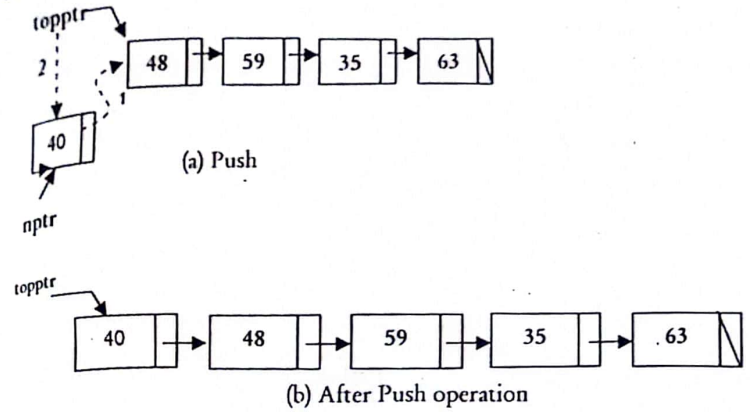


Figure-5.6: Addition of an element to the stack (pictorial view)

Algorithm 5.4: Algorithm to add a node.

1. Declare the node:

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

2. Take necessary pointers: node *nptr;
3. Create a new node:

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

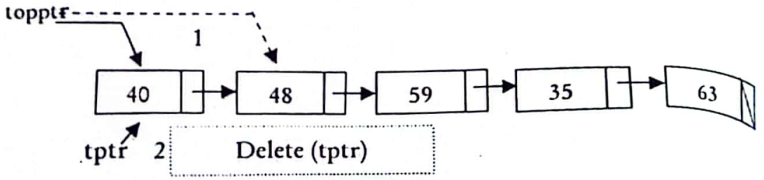
4. Input a linked stack.
5. Make necessary links:

```
nptr→next= topptr;
topptr= nptr;
```

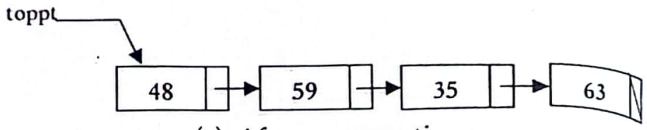
5.3.3 Deletion of an item (Pop operation)

Pop operation in the linked stack is very simple. As we know pop operation means to access the data of the top node from the linked stack and delete the node. Here the *topptr* points the top node of the stack, so to delete a top

node, we use a temporary pointer that will point the top node, access the data from the top node and advance the *topptr* to the next node (using next pointer). Now the *topptr* is pointing the next node. Figure-5.7 shows deletion (*i.e.* pop) of an element from a linked list based stack. The algorithm to delete a node from a stack is stated in Algorithm 5.5.



(a): Pop operation



(a): After pop operation

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

Algorithm 5.5: Algorithm to delete a node from a linked stack

1. Declare node:

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

2. Take necessary pointers:

```
node *topptr, *tptr;
```

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

```
if (topptr→next != NULL)
        {
```



```
topptr = topptr→next;
delete (tptr);
}
else topptr = NULL;
```

6. Output updated linked list

Comments: *topptr* 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 must be the corresponding closing parenthesis, brace or bracket. Otherwise, the expression is not a valid one. The algorithm to check the validity of an arithmetic expression using stack is stated below.

Algorithm 5.6: Algorithm to check the validity of an expression.

1. Input the expression to be validated.
2. Whenever an opening parenthesis is encountered, it is **pushed** onto the stack.
3. Whenever a closing parenthesis is encountered, the **stack** is examined as follows.
    i. If the **stack** is empty, the closing parenthesis does not have an opening parenthesis the expression is therefore **invalid.**
    ii. If the **stack** is not empty, we **pop the top element** from the stack and check whether the popped element **matches** (corresponds) to the closing parenthesis.

iii. If a match occurs, we continue. Otherwise, the expression is **invalid.**

4. 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}]

Table 5.1: Validation process of an expression

Sl #	Symbol scanned	Stack	Operation
1.	[	[	Push
2.	(	[, (	Push
3.	A	[, (	None
4.	+	[, (	None
5.	B	[, (	None
6.	)	[	pop and match
7.	-	[	None
8.	{	[, {	Push
9.	C	[, {	None
10.	+	[, {	None
11.	D	[, {	None
12.	}	[	pop and match
13.	]		pop and match

In the above example, we see the stack is empty at the end, so the expression is **valid**. During the process, if we find no matching of the

opening and closing parenthesis, we can say the expression is **invalid**. When we find an extra opening or closing parenthesis, then the expression is also **invalid.**

Now we describe how the above example can be implemented. We can store the characters of the expression suing a character array as follows:

```
char exp[13];
fflush (stdin);
gets(exp);
```

Next step is to read each character from the array *exp*, and check whether it is '(' or '{'or '['. If we find any of these characters it is pushed into a stack, which is also a character array. It can be done as follows:

```
len = strlen(exp);
k = 0;
for( i = 0; i < len; i++)
 {
ch = exp[i];
if (ch  ='('  || ch = ='{'  || ch= =']')
{ st [k] = exp[i]; ++k; }  //push the item on to the stack.
 } //end of for loop.
```

Now we have to check another thing. If we read ')' or '}' or ']' then we pop top character from the stack and verify whether it matches with the character read from the array, exp. It can be done as follows:

```
if (ch = =')' ||ch = ='}' || ch = =']')
{ b = st[k]; --k;}  //pop the item from the stack.
 if (b = =ch) match =1; //matching occurs.
```

The above checking should be done within the loop. Finally, if we get match=1 and the stack is empty the expression is **valid** otherwise it is **invalid.**

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-", or "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".

1. **Prefix:** + A B (operator before its operands)
2. **Infix:** A + B (operator in the middle of its operands)
3. **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 * 70 / 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 presents in an expression then we have to consider the operations within the parenthesis first. Precedence rules of operators have to be applied within the parenthesis also. 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.

1. **Highest:** i) Exponential (^ or ↑)
   ii) Multiplication (* or ×) or Division (/ or ÷)
2. **Lowest:** Addition (+) or Subtraction (-)

Operators with the same **precedence** (*, / or +, -) are evaluated according to their order of occurrences in the expression. For the previous expression we shall compute 10 * 70 (= 700) first, then 700/2 (= 350), finally we get the result by computing 5 + 350 (= 355).

We can convert an *infix* expression to *postfix* form using the **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. An algorithm to convert an *infix* expression to *postfix* form is given below.

Algorithm 5.7: Algorithm to convert an infix expression to postfix form.

1. Input an expression.
2. Scan (read) the expression from left to right (during scanning we shall get a symbol which may be an operand or a parenthesis (opening or closing) or an operator).
3. If the symbol is an operand, **add** it to the *postfix* expression.
4. If the symbol is an opening parenthesis, **push** it onto the stack.
5. 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 it is **popped** and **added** to the *postfix* expression.
   ii) Otherwise, it is **pushed** onto the stack.
6. If the symbol is a closing parenthesis, then
   i) Pop items 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 an expression to see the working process of the above algorithm.

Expression: 5 * (6 + 2) - (8/4)

At first, we find "5" which is an operand so we add "5" to the *postfix* expression (step 3). Next, we get "*", which is an operator; it is pushed onto the stack (step 5(ii)). Then we get an opening parenthesis, we should push it onto the stack (step 4). We proceed to the next symbol which is "6", an operand so it is added to the *postfix*. Then we get an operator " + ", we add it to the *postfix* expression (step 2). 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 6 (i)). Then the corresponding opening parenthesis is removed from the stack (step 6 (ii)). The complete process is shown in the following table (Table 5.2) .

Table 5.2: infix to postfix conversion

Sl #	Symbol Scanned	Stack	Postfix Expression	Stack operation
1.	5		5	none
2.	*	*	5	push
3.	(	*, (	5	push
4.	6	*, (	5, 6	none
5.	+	*, (, +	5, 6	push
6.	2	*, (, +	5, 6, 2	none
7.	)	*	5, 6, 2, +	pop
8.	-	-	5, 6, 2, +, *	pop and push
9.	(	-, (	5, 6, 2, +, *	push
10.	8	-, (	5, 6, 2, +, *, 8	none
11.	/	-, (, /	5, 6, 2, +, *, 8	push
12	4	-, (, /	5, 6, 2, +, *, 8, 4	none
13.	)	-	5, 6, 2, +, *, 8, 4, /	pop
14.			5, 6, 2, +, *, 8, 4, /, -	pop

Let us take a part of the above expression such as 5*(6+2) and the operations should be done and shown in the following figure.

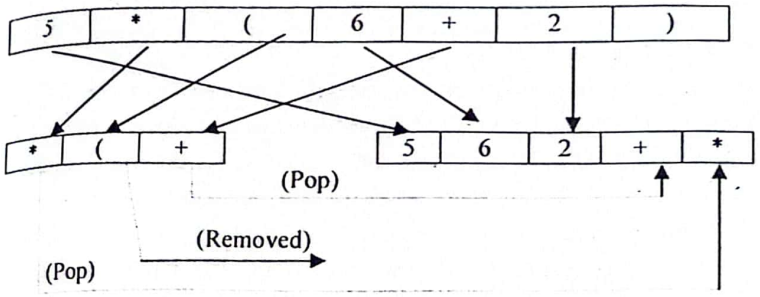


Figure-5.8: Part of expressing checking process

5.4.3 Evaluating a postfix expression

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

5, 6, 2, +, *, 8, 4, /, -

To evaluate the expression we can use the following algorithm.

Algorithm 5.8: Algorithm to evaluate a postfix expression.

1. If an operand is encountered, push it onto the **stack.**
2. If an operator '*op*' is encountered (*op* indicates any arithmetic operation).
   - i. Pop two elements from **stack** where A is the top element and B is the second element.
   - ii. Evaluate B *op* A.
   - iii. Push the result onto the **stack.**
3. The evaluated value is equal to the value at the top of the **stack.**

The process of evaluation can be viewed also as follows.

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) 8	40, 8	

(7) 4	40, 8, 4	
(8) /	40, 2	[8/4] (A=4, B=8)
(9) -	38	[40-2] (A=2, B=40)

5.4.4 Implementation issue

Now let us see how the process of converting an infix expression to postfix form can be implemented. For this, we have to take **three arrays** such as one is for the *infix* (input) expression, another is for the *stack* and the last one is for the *postfix* expression. For the implementation we have to follow the following points:

1. At first, we take *infix* expression using gets ( ). If *exp* is an array to hold the *infix* expression, we will store all the symbols (operand, operator, parentheses) in this array.
2. Next, we scan (read) a symbol from *exp*, and following the **Algorithm 5.7**, the operands are put to the *postfix* array and operator or opening parenthesis are pushed onto the *stack*.
3. The **pop** operation will be performed when we find (read) any closing parenthesis.
4. After popping, we will put the operator into the postfix array by following the rules of precedence of the operators.
5. To differentiate digit and operator we can use isdigit ( ) function. The function isdigit( ) gives (returns) 1 as output for digit and 0 for non-digit symbol. Let us consider the postfix expression is stored in array *i*, for any index *i* we can check it as follows:

   If (isdigit(P[i])> 0) push P[i] to stack.

As we described the expression containing numbers and other characters are stored as characters. If any number contains more than one digit, it should be identified after converting it in postfix form. Now let us describe how we can store the numbers in postfix form if the input expression containing a number, which has more than one digit. Let us take an expression as below.

7*(125/5-20)-(6+12/3)

To identify each number with one or more digits we will store a special character such as # at the end of each number in postfix expression. The process can be viewed as shown in the table.

Table 5.3: Infix to postfix conversion where numbers can be more than one digit

Sl#	Symbol scanned	Stack	Postfix	Stack operation
1	7		7#	
2	*	*	7#	Push
3	(	*, (	7#	Push
4	1	*, (	7#, 1	None
5	2	*, (	7#, 12	None
6	5	*, (	7#, 125#	None
7	/	*, (, /	7#, 125#	Push
8	5	*, (, /	7#, 125#, 5#	None
9	-	*, (, -	7#, 125#, 5#, /	Pop and push
10	2	*, (, -	7#, 125#, 5#, /, 2	None
11	0	*, (, -	7#, 125#, 5#, /, 20#	None
12	)	*	7#, 125#, 5#, /, 20#, -	Pop
13			7#, 125#, 5#, /, 20#, -, *	Pop

The postfix expression is as follows:
7#, 125#, 5#, /, 20#, -, *

Now we evaluate the above postfix expression. The process of evaluation is shown in the following table (Table 5.4).

Table 5.4: Evaluation of postfix where numbers can be more than one digit

Sl#	Scanned number/operation	Stack	Operation	Stack operation
1	7	7		Push
2	125	7, 125		Push
3	5	7, 125, 5		Push
4	/	7, 25	125/5=25	Push after operation
5	20	7, 25, 20		Push
6	-	7, 5	25-20=5	Push after operation
7	*	35	7*5=35	Push after operation

5.4.4.1 Pseudo-code to separate digits from infix and put a special character after digits in postfix form

```
len = strlen(exp); j = 0;
 for(i = 0; i < len; i++)
  {
if(isdigit(exp[i])!=0)
 {
 while(isdigit(exp[i])!=0) //when we find a digit
    {
    post[j++] = exp[i];   //take it into the postfix
     i++;
    }
post[j++] = '#';   //if there is no digit in the number put # into the postfix
  }   //end of if
}   //end of for
```

5.4.4.2 Pseudo-code to separate digits from postfix expression

Now we will see how we can make a number from digits, which are stored as characters in postfix expression.

```
 j = 0;
len = strlen(post);//post is an array that holds postfix
for(i = 0; i < len; i++)
{
        if(isdigit(post[i])! = 0)
        {
           while(isdigit(post[i])!=0) //when we find any digit
                   {
                        d =post[i] - 48; //48 is the ASCII value of zero
                        temp = temp*10;
                        temp = temp + d;
                        ++i;
                   }//end of while
           stack[j]=temp; //put the number onto the stack.
            j++;
        }
}//end of for
```


Summary:

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

The stack can be **implemented** using an array or linked lists.

The 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. It is used in recursive function and graph algorithm.

Questions:

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

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

Problem 5-5: Given a mathematical expression, print the output as "valid" if the expression is valid otherwise print "invalid". In the 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 the stack. As for example take input 5 + (9 * 2 + 2)*8 / 2; convert the infix to postfix expression and evaluate it using stack. Your program must give correct output for any valid input. Implement using arrays.

Problem 5-7: Implement the problem 5-6 using linked lists.

Problem 5-8: Evaluate arithmetic expression using the stack. As for example take input 25 + (12 * 2 + 16)/8 + 52; convert the infix to postfix expression and evaluate it using stack. Your program must give correct output for any valid input.

CHAPTER SIX
QUEUE

OBJECTIVES:

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

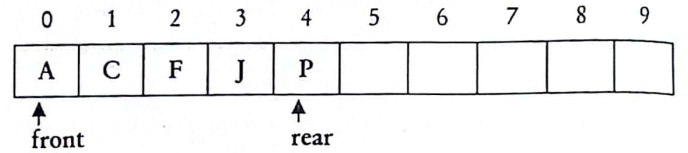
QUEUE

6.1 Definition

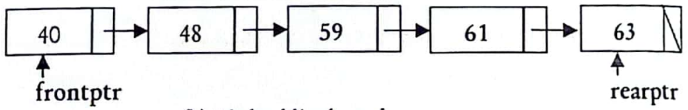
In our daily life, when we stand in a line to get into the bus or to take money from the bank counter, we make a queue. The person who stands first will get into the bus at first, and the person who stands last will get into the bus at last. However, here we will not deal with persons but we will deal with the data items. So in the case of data structure, the element which is added first will be accessed at first and the element which is added last will be accessed 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 the *front* of the queue. So, in a queue, there must be two indicators or pointers. One is *rear* used to add one or more elements, and another is *front* used to delete (access) one or more elements from the queue.

Queue is a FIFO (First In First Out) structure, which tells us the element that is added first will be deleted (accessed) first. Like a stack, the queue can be implemented using an array or a linked list.



(a) Array based queue



(b) Linked list based queue

Figure-6.1: Graphical representation of queue

6.2 Array based queue

The queue created using an array is called array based queue. Here, we have to use two indicators (two index identifiers). One indicator marks the *front* element and another marks the *rear* element of the queue. At the initial stage to create an array based queue we take rear = front = -1, since index of an array starts at zero (0) and we will store first element in zero index.

6.1.1 Addition of an element in an array based queue

As it is mentioned earlier in a queue element is added at the *rear*. So, to add an element at first we increase the value of *rear* (it is an index value) and then place the element in the array-based queue using the new value of the *rear*.

Algorithm 6.1: Algorithm to add an element to queue

1. Take an array based queue and other variables:
   que[0.....M-1], item, front, rear;

2.
```
if (rear = = -1)
        {
        front = rear = 0;
        que[rear] = item;
        }
```

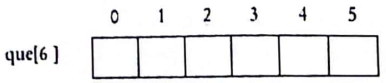
3.
```
if (rear < M-1)
        {
        rear = rear +1;
        que[rear] = item;
        }
else    print "Over Flow message"
```

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.

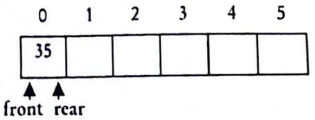
An overflow may occur if we try to add an element after the last index of the array.

The addition operation in an array based queue is depicted in Figure 6.2. We have shown the different stages of the addition operation in the figure.



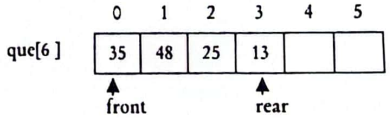
front = rear = -1
item = 35

(a) An empty queue and item (35)



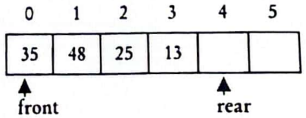
front = rear = 0
que[0] = 35

(b) Adding the first element of the queue



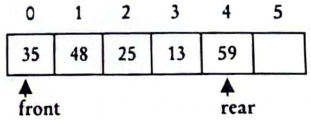
item = 59 front = 0
rear = 3

(c) An array based queue and item



rear = rear + 1 = 4

(d) Advancement of the rear



que[rear] = item
que[4] = 59

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

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

6.1.2 Access (Deletion) of an element from a queue

We know that the element is deleted from the *front* of a queue. So, at fir we access the element, and then we increase the front index. The deleti operation is shown in Algorithm 6.2 and Figure 6.3 is a pictorial view of it deletion operation.

Algorithm 6.2: Algorithm to delete an element from a queue

1. Take an array based queue and other variables:
   que[0……M-1], item, rear, front;

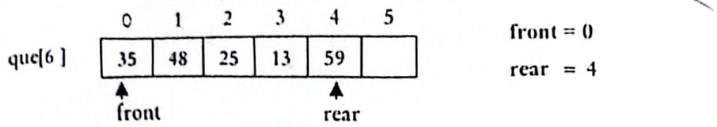
2.
```
if (front = = rear)
        {
        item = que[front];
        front = rear = -1
        }
```

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

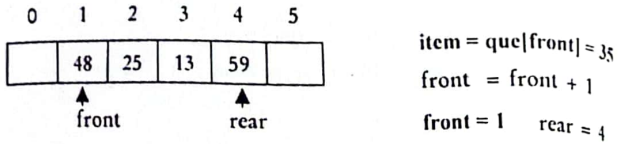
4. Output: Updated queue

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 -1; *i.e.*, *front* = *rear* = -1, which indicates an empty queue.

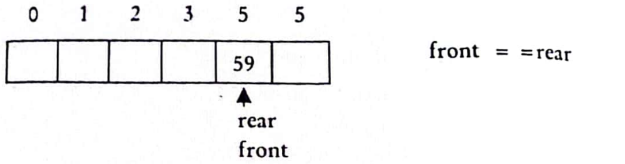
In the case of deletion operation for an array based queue actual deletion of an element will not be performed. We just access the element from the queue and use it in a necessary operation. That is why here deletion means access of an element from the queue.



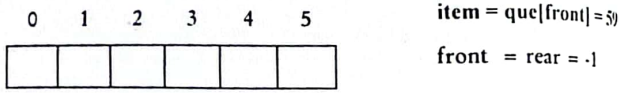
(a) An array based queue



(b) Deleting the first element of the queue



(c) The queue with only one element

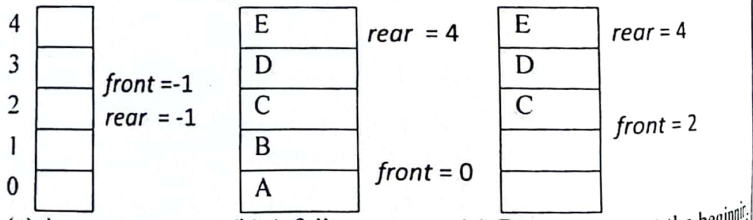


d) Deletion of the last element of the queue

Figure-6.3: Deletion process of an array based queue (a 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 array. Let us consider the following situations.



(a) An empty queue (b) A full queue (c) Empty space at the beginning

Figure- 6.4: Pictorial view of odd situation in a queue.

Figure 6.4(a) shows an empty queue. Figure 6.4(b) shows a full 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, suppose that 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* will be six which is greater than the size of the queue. In this situation, logically we can not 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* = 0 and *rear* = 4 (maximum index value of the rear).

In such situations where the *rear* is equal to the maximum index value of the queue but the front is greater than 0 (Figure 6.4(c)), to insert an element we have to set the value of *rear* to 0. 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.

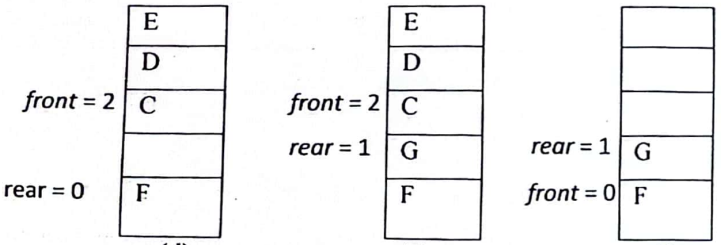


Figure-6.5: Pictorial view of addition of element with odd situation of queue

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, the queue can be considered as a circular queue and we can modify the insertion and deletion algorithm for an array based queue as described in below.

6.1.4 Circular queue

The circular way is an efficient representation of queue using an array. If we consider a queue as a circular queue, we can easily overcome the odd situation shown in Figure 6.5. The pictorial view of a circular queue is shown in Figure 6.6. Here we use position or indexing as in C/C++. Indexing starts at 0 and ends at M-1 (if the size of the queue is M). Initially we set front = -1 and rear = -1. For the addition of first element to the queue we set front = 0 and rear = 0;

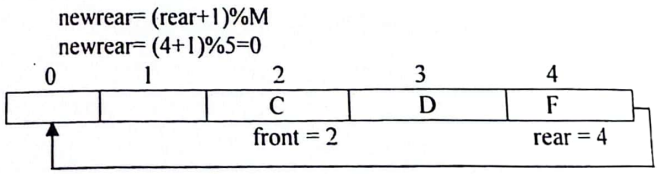


Figure-6.6: Consideration of a circular queue

When we find the condition, where the new value of rear equals front (newrear = = front) in the case of addition, no addition operation can be performed (the queue if full). Otherwise, we can add an element to the queue. We will find new rear position as newrear = (rear + 1) % M.

In the case of deletion, if there is a condition, where front equals rear (front = = rear), there is only one element in the queue. We access the element and

reset the values of rear and front to -1. Otherwise, we access the element and increase the front as front = (front + 1) %M. The algorithms for addition and deletion are given below.

Algorithm 6.3: Algorithm to add an element to a circular queue

1. Input a circular queue as que [0....M-1].
2. If (rear = = -1 and front==-1)
   {
   front = 0; rear = 0;
   que [rear] = item;
   }
3. else
   {
   newrear = (rear + 1) % M;
   if (newrear = =front)
   {
   print "Queue is Full";
   Terminate the algorithm;
   }
   else {
   rear= newrear;
   que[rear] =item;
   } //end of step 3.
4. Output an updated queue.

Algorithm 6.4: Algorithm to delete (access) an element from a circular queue.

1. Input a queue as que [1...M]
2. if (front==-1&&rear==-1) //No element in the queue

```
  {
  print "Queue is empty";
  }
3. else if (front= = rear) //Only one element in the queue
   {
  item = que [front];
  front =-1; rear =-1;
   }
4. else
   {
   item =que[front];

front = (front +1) % M;
   }
```

5. Output an updated queue.

6.2 Link based queue

The queue created using a linked list is called linked list 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. A linked queue is shown in Figure-6.7.

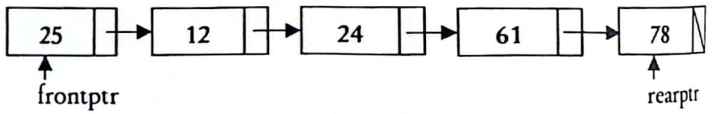


Figure-6.7: A linked queue

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 and we have a node in the queue. To add the second node, we create another new node and add this node to the queue. For this, we make the 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 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. The algorithm (pseudo-code) to create a linked list based queue is given in Algorithm 6.5. The pictorial view is shown in Figure 6.8.

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**:

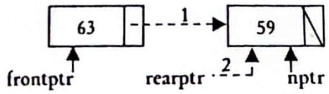
```
if (rearptr = = NULL)
        {
        rearptr = nptr;
        frontptr = nptr;
        }
```

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

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

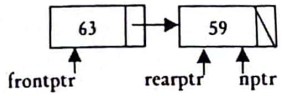
Comments: Here, *frontptr* is a pointer to the front node and *rearptr* is a pointer to the last node of the queue. *nptr* is a pointer that points to the new node, and the item is a variable used to enter data into the new node.

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



1. rearptr→next = nptr
2. rearptr = nptr ;

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



(c) Updated linked queue

Figure-6.8: 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 a linked queue. The algorithm (pseudo-code) is shown below.

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;
```

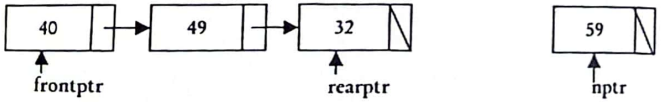
2. Input a linked base queue
3. Create a new node:

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

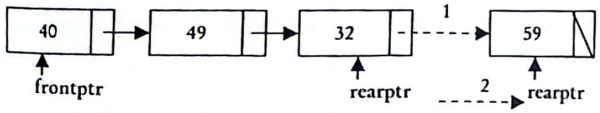
4. Make link among the necessary pointers:

```
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-6.9: Add a new node to a linked queue

6.2.3 Delete a node from a linked queue

We know that in a queue deletion operation must be performed from the *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 a temporary pointer. The algorithm (pseudo-code) is given below. The Figure 6.10 is a pictorial view of a deletion operation.

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

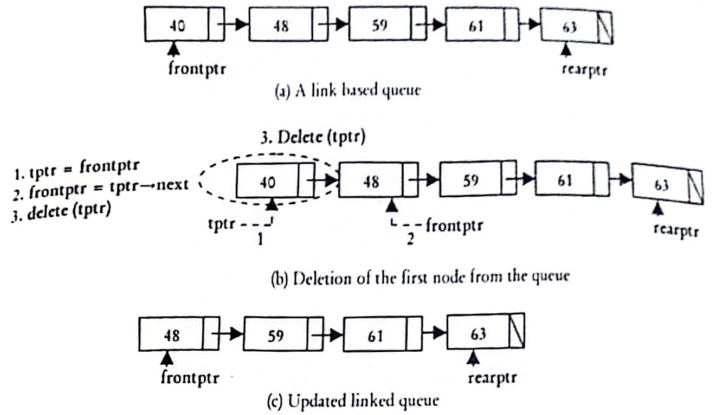


Figure-6.10: Deletion of a node from a linked queue (pictorial view)

Summary:

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

Questions:

1. Define queue. Give an 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 a linked queue with an example.

6. Write an algorithm to add a node to queue when the queue is a linked list.
7. Write an algorithm to delete a node from a queue, when the queue is a linked list.
8. Write an algorithm to insert an element and delete another element from a queue where the queue is an array based structure.

Problems for practical (Lab) classes

1. \Write a program to create an array based queue and add five elements to it. Next, add three elements and access one element. Hints : take the size of the array 10.

2/ Write a program to create a linked queue with six nodes. Add two nodes and delete a node from the queue. Print the data of the queue.

3. You are given a stream of character find the first repeating character and; put-1 in its position each time a character is added to the stream.

Sample input : b b c d

Sample output: b -1 c d

4. You are given a queue of integers and an integer k, reverse the order of the first k items of the queue, the relative order of the other elements remains as it was.

Sample input : [15, 25, 30, 45, 50, 70, 75, 80, 95, 100], k = 4

Sample output : [45, 30, 25, 15, 50, 70, 75, 80, 95, 100]

CHAPTER SEVEN
TREE

OBJECTIVES:

- Identify tree
- Describe tree representation methods
- 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 a tree in nature. A tree has root, branches, sub-branches, and leaves. From the concept of a natural tree, the computer scientists get the idea of a data structure, which is graphically similar to a natural tree. The 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 relationships among the nodes. It is a hierarchical structure. An ordered tree has a specially designated node called root node. The root node may have one or more child nodes. The connection line between two nodes is called an edge.

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

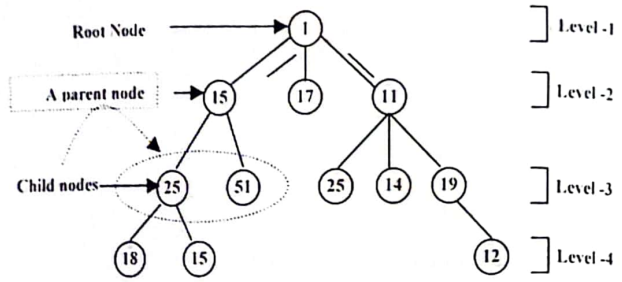


Figure-7.1: A Tree

The node that has no child is called a *leaf* node. If a node has a child, the 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 a 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 two trees excluding root node are called left sub-tree and right sub-tree.

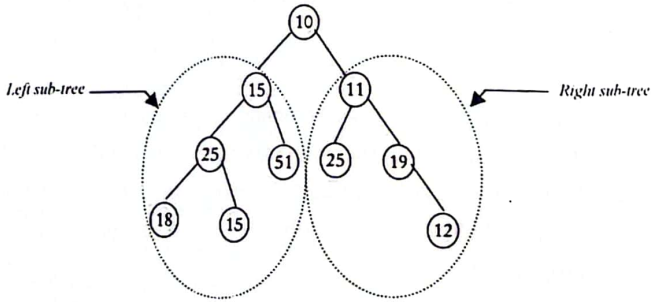


Figure - 7.2: A binary tree

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

Figure-7.3(b). When a binary tree is represented using an array, usually the index of the root node will be 0 (the first index of an array in C/C++). Next, we traverse left to right at one level to another such as at level two the left node's index is 1 and the right node's index is 2 and we start the third level. In an array, the nodes are stored according to the respective indices. If a node has no any of two children such as a left or right child, the place of that child node will be empty. Nodes of a binary tree with their respective indices are shown in Figure 7.3(a).

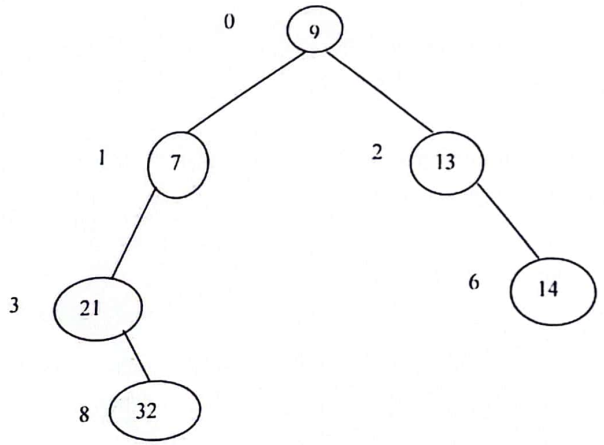


Figure 7.3(a): A binary tree with respective index number of an array.

Now let us describe the tree in Figure 7.3(a) for array representation. As we mentioned the root's index will be considered as 0. Next, we go to the left child of the root and its index is 1 and for the right child, the index is 2. Now the next level is considered. The left child of the node at index 1 (the node with data 7) exists, so its index is 3 but the right does not exist so the index number 4 of the array will be **empty**. Similarly, we have to consider up to the last node of the last level. The pictorial view of an array of the tree is as follows.

0	1	2	3	4	5	6	7	8
9	7	13	21			14		32

Figure 7.3(b): An array of tree in Figure 7.3(a)

When a binary tree is in complete form, its array implementation is efficient.

If there are k levels in a binary tree, then the maximum number of nodes can be in a binary 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 in a complete binary tree is known, number of levels can be calculated as

$$k = \lceil \log_2 (n + 1) \rceil$$

For example,

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

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

// $\lceil x \rceil$ means ceiling of x to the next integer

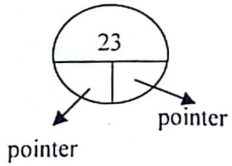
7.1.1 Parent-Child Relationship

If we consider the position (index) of the root node is 0 and any node is in position i, its left child's position will be $2i + 1$ and right child's position will be $2i + 2$. As for example, there is a node at position or index 3, its left child's index will be 7 and the right child's index will be 8. On the other hand, a node of position k has its parent is in position $(k - 1)/2$ where / means integer division. Such as, there is a node at 12th index its parent will be at the index 5.

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 $i / 2$. The position of the 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 12, then the position of its parent is 6.

A binary tree can also be implemented (stored in computer's memory) as a linked list. Each node should have three parts, one is for holding data, another is a pointer to the left child and the third one is a pointer to the right child. The node of the binary tree can be defined (declared) as follows:

```
struct node
    {
    int data;
    node *lchild;
    node *rchild;
    };
```



where ***lchild*** is a pointer that points the left child and ***rchild*** is a pointer that points the right child. In the node structure, ***data*** is an integer variable to store the value of the node. It means a node has three parts. One is data part (value), the second is a pointer to the left node and the third is a pointer to the right node. For a new node, the pointers to the left child and the right child both will be null. When we make link if a node does not have left child its left pointer (lchild) will be null. Linked list implementation of a binary tree other than the complete tree is efficient. Let us see how we can create a new node for a binary tree. At first, we declare the structure for the node.

```
struct node
  {
  int data;
  node *lchild;
```

```
node *rchild;
  };
node *nptr;
nptr = new(node);
nptr->data = 14;
nptr->lptr= NULL;
nptr->rptr= NULL;
```

7.1.2 Creation of a binary tree using linked list

Now we see how a binary tree can be constructed using a linked list. The tree has been created using the pre-order method. The *createtree* function uses the pre-order method to create a link based tree. After creation, the data have been accessed or printed using in-order traversal method. If we print the data with pre-order traversal method we will get the output as the input order.

```
struct node
  {
int data;
node *lchild;
 node *rchild;
  };

node *nptr;
node *createtree(node *curptr, int item)
 {
if(curptr==NULL)
{
nptr=new(node);
nptr->data=item;
nptr->lchild=NULL;
nptr->rchild=NULL;
```



```
curptr=nptr;
 }
else if (curptr->lchild==NULL) curptr->lchild=createtree(curptr->lchild,
item);
else curptr->rchild=createtree(curptr->rchild, item);
 return curptr;
}
void inorder(node* curptr) {

  if (curptr!= NULL)

   {

 inorder(curptr->lchild);
cout <<" "<< curptr->data;
 inorder(curptr->rchild);
 }
}
int main()
  {
 node *root;
 root=NULL;
 int i,item;
 for (i=0;i<5;i++)
 {
cin>>item;
 root = createtree(root, item);
 }
cout<<endl;
inorder(root);
cout<<endl;
return 0;
}
```

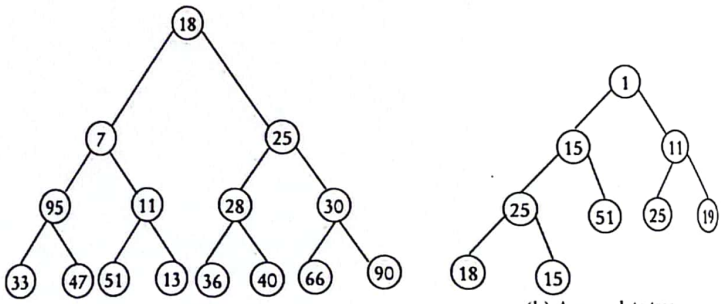
Full binary tree:

If a binary tree contains nodes in such a way that every node except the node at the last or deepest level has at least two children (the last level is full) then the tree is called a *full binary tree*. Here, all the levels are full with nodes.

Complete binary tree:

If a binary tree contains nodes in such a way that every level except the deepest one contains as many nodes as possible and the nodes of the deepest level remain 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 full binary tree is also a complete binary tree.



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

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

7.1.3 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

7.1.3.1 Pre-order traversal method

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

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

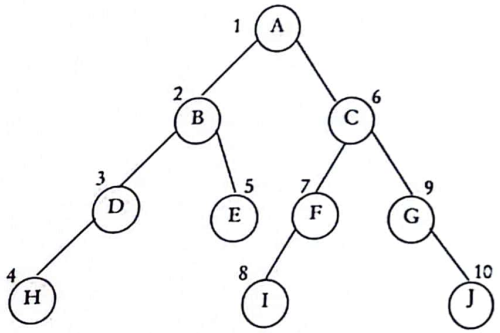


Figure-7.5: Pre-order traversal method

In pre-order traversal method, we have to visit the root node first. Then we traverse the left sub-tree following the above three 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.

Next, 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 a 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. The algorithm (pseudo-code) of pre-order traversal method is given in Algorithm 7.1 when a tree is constructed using an array.

Algorithm 7.1 Algorithm (pseudo-code) for pre-order tree traversal using array

```
preorder (int i, int tree [ ])
   {
  if (i<n)
 {
cout<< tree[i];
 preorder (2*i+1, tree);
preorder(2*i+2, tree);
  } //end of if
  } //end of preorder
main()
 {
input tree[n]  // an array of size n
preorder (0, tree);
}
```

When the tree is constructed using a linked list, we can traverse it using the algorithm (pseudo-code) given in Algorithm 7.2.

Algorithm 7.2: Algorithm for pre-order traversal using linked list

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.

In algorithms 7.1 and 7.2 we have written pseudo-codes for preorder traversal method. Both algorithms use a recursive function to print or access the data from the tree. Now we see how recursive function accesses the data. Any recursive function uses a stack to save address and it uses this address when it is necessary to return to the address. Let us see the tree in

Figure 7.5. The function access the data A and save its address to a stack (See Figure 7.6), if it's left pointer is not null, it goes to the left child. It accesses the data and saves the address of B to the stack. Thus it accesses data of the left children and saves their addresses to the stack until it finds null in left. How the addresses will be saved shown in Figure 7.6 (a). Now the function uses the stack and popes the address of the data from the top of the stack. The last accessed data was H (See figure), so the top address would be the address of H. Now the function checks the right pointer of the node with data H. Since it is null, it accesses the next address from the stack. It is the address of the node with data D. Since it is also null, the address of the node with data B is popped. The right pointer of the node is not null, so the data of the right child will be accessed and address of the right child (E) saves at the top of the stack. Now the function checks the left pointer of E since it is null, it checks the right pointer of the node. It is null also and next address from the stack will be popped. This address is the address of data A. The function checks the right pointer of A, this pointer's data is C. It is accessed (printed) and its address will be saved to stack. The process will be going on until the stack is empty. In Figure 7.6 AD (A) means the address of the node with data A.

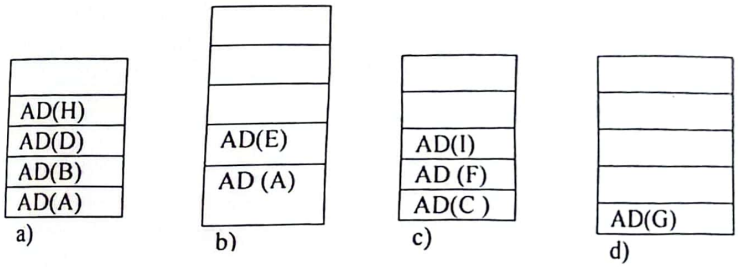


Figure 7.6: Use of stack in pre-order traversal method.

Let us see an iterative version of an algorithm which uses the stack and accesses data in the pre-order method. The algorithm (pseudo-code) uses the stack but does not use a recursive function. We have used array based tree for the Figure 7.5.

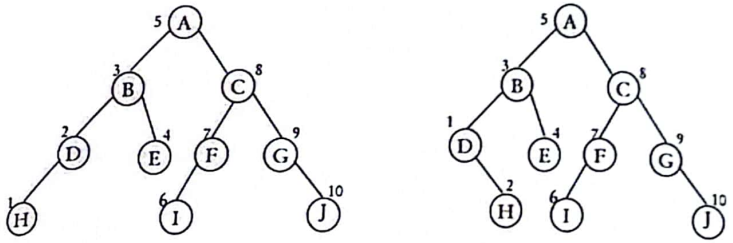
Algorithm 7.3: Algorithm (pseudo-code) to traverse a tree in preorder method using stack

```
preorder(int i, char tree[])
 {
        int s[10]; //s is a stack
        int top;top=-1;
        while(i<15)
         {
                 ++top;
                s[top]=i;     //push or save address to stack
                cout<<" "<<tree[i];
                i=2*i+1;      //address of left child
                }
        while(top>=0)
                {
                i=s[top];
                top--;
          i=2*i+2;       //address of right child
                while(i<15)
          {
                 ++top;
                s[top]=i;
                cout<<" "<<tree[i];
                        i=2*i+1;  //address of left child of sub-tree
                }
                i=2*i+2;       //address of the right child of sub-tree
         while(i<15)
         {
                ++top;
                s[top]=i;
                cout<<" "<<tree[i];
                i=2*i+1;
        }

        } //end of while with top;

} //end of function


main()
{
char tree[15]={'a', 'b','c','d','e','f','g','h',' ',' ',' ',' ','i',' ',' ','j'};
preorder (0, tree);
 }
```


7.1.3.2 In-order traversal method

In in-order tree traversal method we have to follow the points below:

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



(a) (b)

Figure-7.7: In-order traversal method

In in-order traversal method, we have to traverse the left sub-tree first. So, we have to start from the last node at the left of the left sub-tree. Then we traverse the root and at last, we will traverse 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 sub-tree. The left children of the 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 Figure-7.7(a), we have to start the last node at the left of the left sub-tree. The data of the left most node is **H**. Then we visit its parent node, **D** (as the root of this section of the tree). 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 the 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 Figure-7.7(a) is as follows:

H D B E A I F C G J

If we traverse the tree in Figure-7.7(b), we have to start from the node **D**. Then we will visit the node **H** as the right sub-tree. The sequence of visiting other nodes will remain same as the previous. Therefore, visiting sequence is as follows:

D H B E A I F C G J

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

Algorithm 7.4: 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.

7.1.3.3 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.

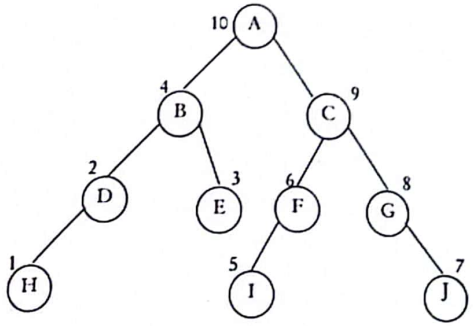


Figure-7.8: Post-order traversal method

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

For example, to traverse the tree in Figure-7.8, we 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 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-7.8, visiting sequences are marked as 1, 2, 3 and so on.

Algorithm 7.5: 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 at the root of the tree, and all the node values of the right sub-tree are greater than the node value at the root. If we treat the left sub-tree as a tree, then this tree (left sub-tree) also have the above properties and this must be true also for the right sub-tree.

If the structure of a BST is a complete or a balanced binary tree, it gives the 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 most of the cases 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*.

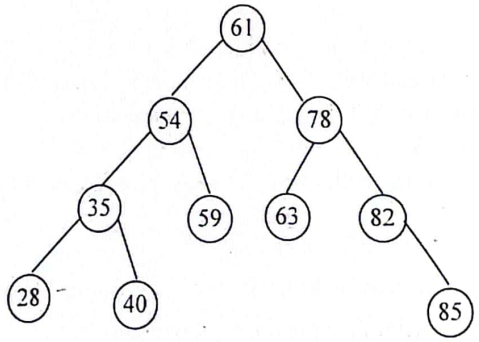


Figure-7.9: A binary search tree (BST)

7.2.1 Create Binary Search Tree

To create a BST we can use same node declaration as we declared before. Using the following recursive function we can create a BST.

```
node *creatBST(node *root, int item)
  {
```



```
if (root = = NULL)
      {

nptr = new(node);
      nptr->data= item;
      nptr->lchild = NULL;
      nptr->rchild = NULL;
      root = nptr;
      }
else
      {
        if (item < root->data)
                    root->lchild=creatBST(root->lchild, item);
            else if (item> root->data)
                    root->rchild=creatBST(root->rchild, item);
            else cout<<"Duplicate not allowed";
            }

return root;

} //end of function
```

We can call this function in main function as follows:

```
node *root;

root = NULL;
```

Within a loop use the code as, cin>>item; root=creatBST(root, item);

To print or access the data of BST *in-order* traversal method can be used. The results of *in-order* traversal method will be sorted data in *ascending* order.

7.2.2 Searching a particular node value of a BST

Suppose that there is a BST and value of a node, we have to determine whether the value exists in the BST or not. It is known 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 7.6: Algorithm to find out a particular node value of BST

1. Input BST and a node value, *x*;
2. Repeat Step-3 to 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 the root node value, we have to search the left sub-tree (by treating it as a BST).
5. Else we have to search the 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 that 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.

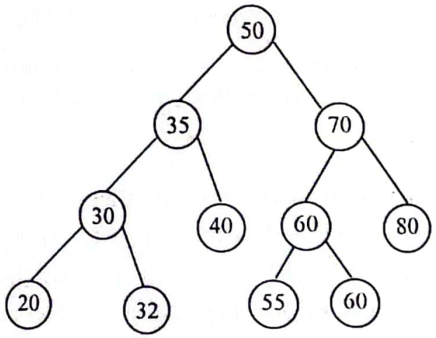


Figure-7.10: Searching for a node value (shaded nodes show the searching path)

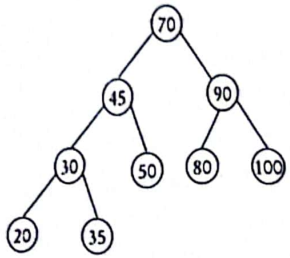
7.2.3 Add a node to a BST

To add a node to a BST, we have to find the proper position for the node. To do this, we use a 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 the last node in the searching path, we shall add the node on its (last node's) left or right.

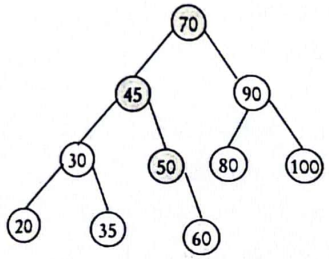
Algorithm 7.7: Algorithm to insert (add) a node to a BST

1. Input BST and a new node;
2. Repeat Steps-3 to 5 until we find the value or we go beyond the tree.
3. If *x* is equal to the root node value, searching is successful and terminate the algorithm.
4. If *x* is less than the 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 a link between the new node and the parent node of the new node. If the value of the new node is less than its parent's node, link it as a left child otherwise, link it as a right child.
7. Output: Updated BST.

Let us consider that there is a BST as in Figure 7.11 (a) and we have to add a node with value 60. At first, we search the value 60 in the BST according to the 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 7.11 (b), the value 60 > 50, it has been added as a right child of the node with the value 50.



a) A binary search tree



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

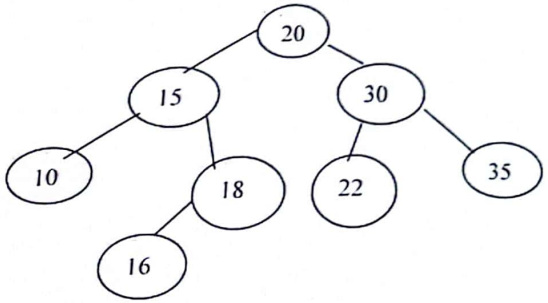
Figure-7.11: Addition of a new node to BST

7.24 Delete a node from BST

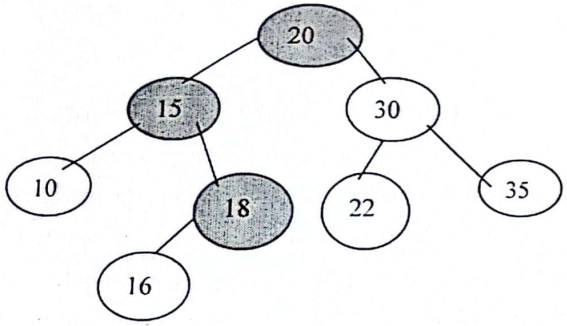
To delete a node from a BST, we have to perform searching to locate the target node. After finding the target node (the node to be deleted), one of the three cases must be considered. *Case 1*: if the target node is a leaf node. *Case 2*: if the target node has only one child. *Case 3*: if the target node has two children (grand children also). In deletion process we face any of the three cases and perform operation accordingly.

Let us see *case 1* and *case 2* first. In *case 1*, we locate the node using the searching algorithm, make *null* the respective pointer of the parent node such as if the target node is a left child make the left pointer (link) of the parent node *null* and delete the node. In *case 2*, we locate the node and make the link between the child and parent node of the target node and delete the target node.

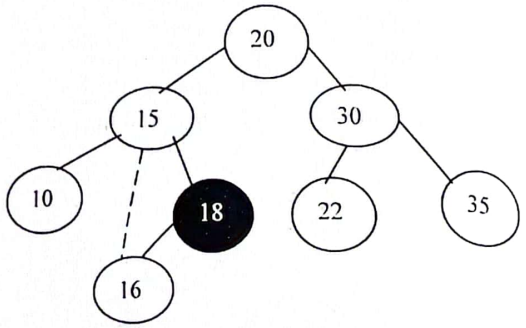
Let us see the pictorial view how we can delete a node from a BST if the node has only one child. The view is shown in Figure 7.12. Now if we want to delete the node with value 18. First, we perform searching. The searching path is shown by shaded nodes. After finding the target node we make the link between the parent of the target node and the child of the node. Next deletion can be performed.



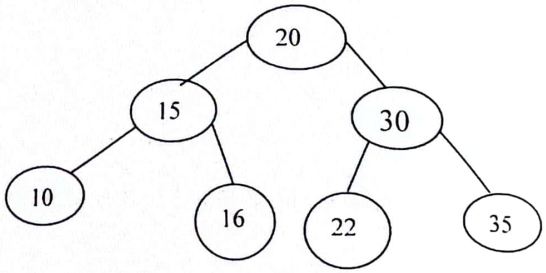
a) Delete node with value 18.



b) Searching has been done



c) Updating link



d) BST after deletion of the node

Figure 7.12: Deletion of a node with one child.

Now we describe how to make the link between two nodes in the above example. During searching, we have to use two pointers, there should be a pointer to the parent node and another pointer to the target node. Let the pointer to the root node is *root*. We use a pointer, **curptr** for searching the node and **pptr** points the parent of the node to be deleted. In our example **curptr** will point the node with value 18 and pptr will point the node with value 15. After finding the node with the searching procedure we make a link and delete the node using the following code.

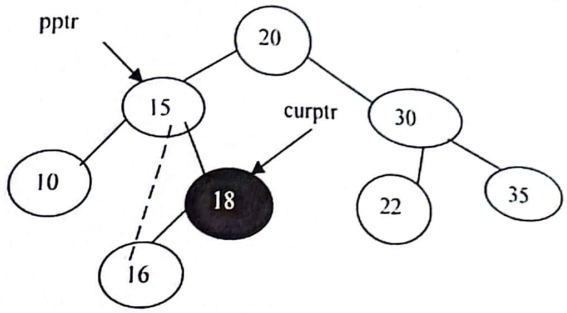
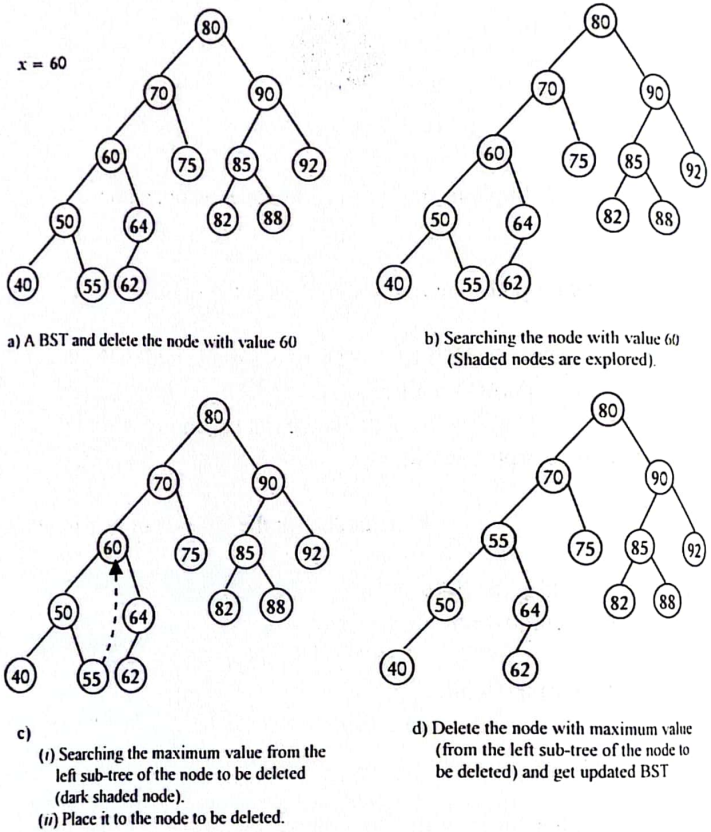


Figure 7.13: Implementation issue for deletion operation

```
if (curptr->data > pptr->data)  //The node is at the right side of its parent
   {
   if (curptr->rchild != NULL)    //There is a right child of the node
   pptr->rchild = curptr->rchild;
   else                           //There is a left child
   pptr-rchild = curptr->lchild;
  }
else                          //The node is at the left side of its parent
 {
  if(curptr->rchild != NULL)
 pptr->lchild = curptr->rchild;
 else
 pptr-lchild = curptr->lchild;
  }
delete (curptr);
```

In this *case 3*: after locating the target node we do another searching in left sub-tree of the target node and find the node with the 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). When we go to delete the node with the maximum value we may find *case 2* here, in such situation we have to perform *case 2* with *case 3*. In the Figure 7.14, 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-7.14: Deletion of a particular node from a BST

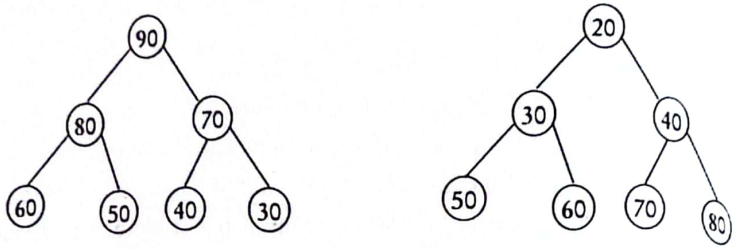
Algorithm 7.8: Algorithm to delete a node from a BST

1. Input BST, the value of the node to be deleted.

2. Locate the node to be deleted using a searching algorithm (Algorithm 7.6).
3. If the node is a leaf node:
   i. if the node is a left child of the parent, make null the left pointer of its parent node and delete the node.
   ii. if the node is a right child of its parent, make NULL the right pointer of its parent node and 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 the link between the left pointer of its parent and the child (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 the link between the right pointer of its parent and the child (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. if the node with maximum (minimum) value has a child, do any of steps 4(i) and 4 (ii).
   iv. delete the node located in step-5(i)
6. Output: Updated BST.

7.3 Heap

It is a *complete* binary tree where each node 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*. On the other hand, if the node value is smaller than its children, the heap is called *min heap*.



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

Figure-7.15: Graphical representation of Heap

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

7.3.1 Heap creation

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

To create a heap, a list of data is used and the data are stored in an array. We rearrange the data of the array and create a heap. To create a heap with the data of an array, we consider the data of any position and its children. We will create a *max heap* here. As it is known for a max heap the data of the parent will be greater than the data of its child. Let us recall that child-parent relationship when the index of the data at the root is 1. In this case, if the child's index (position) is k, the parent's position will be k/2. Here we consider any data and its parent. If the data of the parent's position is smaller than the data of the child's position, the data of the parent's position will be moved down (to the child's position). Thus we consider any data and compare with its parent, move down if necessary. By considering the data of all positions of the array we can create a heap. To create a heap we have to follow the following steps, which we may consider as an algorithm for creating a heap.

1. Take a list of data in an array.
2. Consider the data of the first position of the array and it is a heap (when we consider only first data it is a heap).
3. Consider the data of the current (from second to last) position and save it in a temporary variable, *temp*.
4. Compare the data of the current position with its parent.
5. If the parent is smaller than temp, move it down (move it to the child's position).
6. Consider the parent's position as (new) current position.

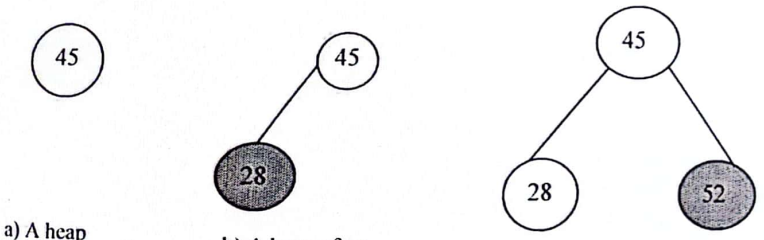
7. Repeat the steps 4 to 6 until we reach the root.
8. If the parent is greater than the *temp*, put the *temp* in the current position.
9. The output will be a heap.

Using the above points we can create a heap. Let us consider an array of data as follows.

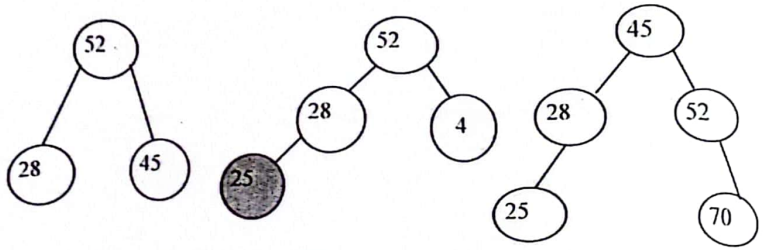
A[ ] = { 45, 28, 52, 25, 60, 70};

At first, we consider 45 only. It is a heap (only one data). Next, we consider 28 (the data of the second's position). We save it in temp, so temp = 28. Now we compare temp and 45, since the parent is greater, so array (tree) of 45 and 28 is also a heap. Next we consider 52 and temp = 52. If we compare temp and 45 (data in parent's position), parent (45) is smaller than temp (52), so the data 45 is moved down (move 45 to the third position). Parent's position is the current position now. Since we reach the root, we put data in temp (52) at the current position (root). We can consider all the data of the array, A and create a heap as illustrated in the following figure (Figure 7.16). The algorithm using pseudo-code has been given in Algorithm 7.9.

Given List: 45 28 52 25 60 70



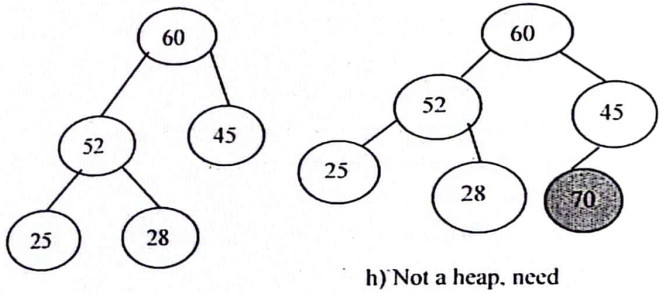
a) A heap b) A heap after addition c) Not a heap, need rearrangement



d) A heap after rearrangement

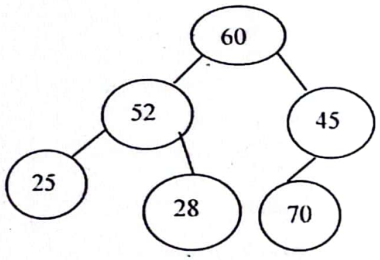
e) A heap after addition

f) Not a heap, need rearrangement



g) A heap after rearrangement

h) Not a heap, need rearrangement



f) A heap after rearrangement

Figure 7-16: Pictorial view of a heap creation process

Algorithm 7.9: Algorithm [pseudo code] to create a heap

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]; //take data from the second position to last position
   //one by one
   k = i;
3. while (k > 0 and A[k/2] < temp)
   {
   A[k] = A[k/2]; //movement of data from parent's to child's
   //position
   k = k/2; //consider parent's (upper) position now.
   }
   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).

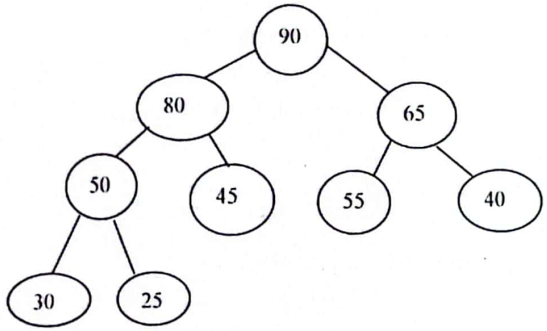
7.3.2 Deletion of maximum from a max-heap

In the case of deletion of maximum, we will not delete the node with the maximum value but we delete the value only. As we know that in a max heap the maximum value remains at the root, so the value will be removed from the root. After removal of the value from the root, the root node is empty and the tree will not be a heap after deletion. So we have to rearrange the data and reconstruct the tree to get a heap. To delete the maximum from the root and reconstruct the tree as a heap we have to follow the following steps given below.

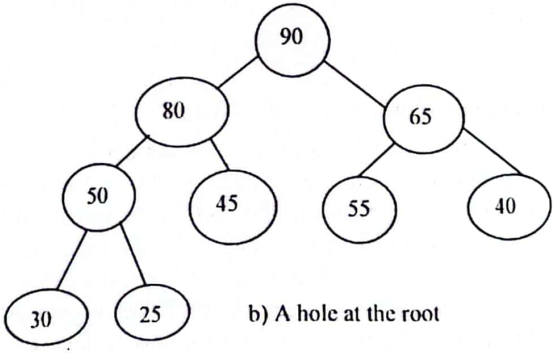
1. Keep (save) the data of the nth (last) position in a temporary variable, *temp.*

2. Remove (delete) maximum from the root and assume there is a hole at the root.
3. Compare the children of the hole and identify the greater child.
4. Compare the greater child with *temp*. If *temp* is larger than the greater child, terminate (break) the loop and put the *temp* in the hole.
5. Otherwise, put the greater child in the hole.
6. Repeat the steps 3 to 5 until the hole is at the last level.
7. If the hole at the last level put the temp in the hole and delete the last node.

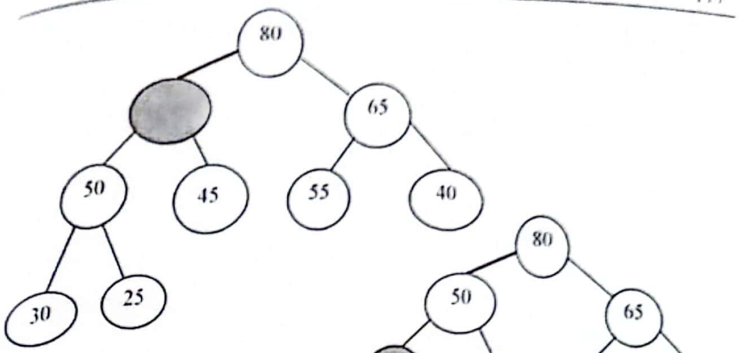
The operation on delete maximum is illustrated using Figure 7-17 (a to f) as follows:



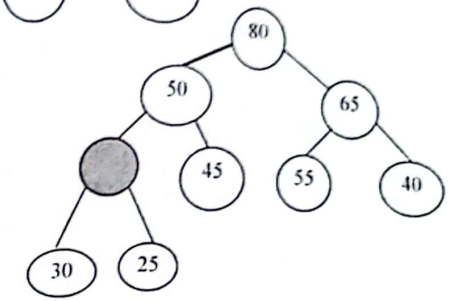
a) A max heap for root [maximum] deletion



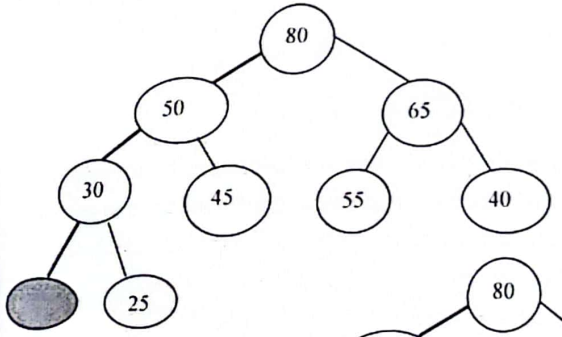
b) A hole at the root



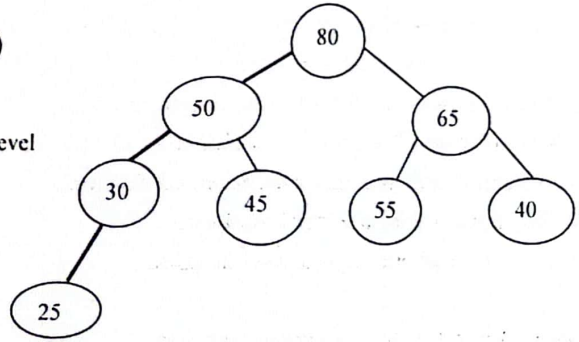
c) A hole at the second level



d) A hole at the third level



e) A hole at the last level



f) Data at the last position (temp) in the hole and delete last node

Figure 7-17: Delete maximum value from a max heap (Pictorial view)

Algorithm 7.10: Algorithm (pseudo code) to delete the maximum value

```
1. Input a heap, A[1...n]. // The heap as an array
2. i = 2;
3. temp = a[n]; //the data of the last position in temp
4. while (i <= n)
        {
        if ((i < n) and (a[i] < a[i+1])) // if it not last level find the
                                         //greater child
        i = i + 1;
   if (temp >= a[i]) break;  //the data in temp greater or equals the
                             //greater child

   a[i/2] = a[i];          //If the data in temp is not greater or equals,
                           //put the greater child at parents position
   i = 2*i;                // consider the child now
        }
        a[i/2] = temp;

5. Output updated heap with (n − 1) elements.
```

7.3.3 Heap sort

A list of data stored in an array can be sorted in ascending or descending order using a heap. To sort the data in ascending order we have to use *max heap* and to *sort* the data in descending order *min heap* should be used. To sort the data using a heap we have to use the following steps.

1. Create a heap using a list of data.
2. Delete the root and rearrange the data to reconstruct a heap.
3. Repeat the step 2 n-1 times to delete n-1 roots (except the last root, when heap contains only one data after repeated deletion).
4. The output will be a sorted list of data.

Let us see the above steps as pseudo-code.

Algorithm 7.11: Algorithm for heap sort

1. Input (Take) an array with a list of random data.


```
2. createheap (a, n).      //create a heap using creatheap function where a is
                           //an array and n is the size of the array
3. for (j = n; j > 1; j--)
     rearrange (a, j);     //rearrange data and construct heap n-1 times
   if (a[1] > a[2]) {temp = a[1]; a[1] = a[2]; a[2] = temp;}
4. Output a sorted list in array, a.
```

In the above small algorithm, we have called createheap function to create heap and rearrange function for deletion the root and rearrange the remaining data in a heap. For implementation, we have considered max heap here, which indicates the data will be sorted in ascending order. If anyone wants to execute the algorithm 7.11 using C/C++, pseudo-code of the algorithms createheap and rearrange are as follows. In the code we have considered that the root's index is 1. Let us recall if the root's index is 1, the left child's index for any parent in index i will be 2i and the right child's index will be 2i + 1. On the other hand if the child's index is k, the parent's index will be k / 2.

```
void createheap (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] indicates the data at child's position and a[k/2]
                    // indicates the data at parent's position.
  k = k/2;
}
a[k] = temp;
} /end of for
}//end of the function.

void rearrange (int a[], int j)
{
 int temp, i;
 i = 2; s = j-1;
 temp = a[j];   //last data in temp.
```

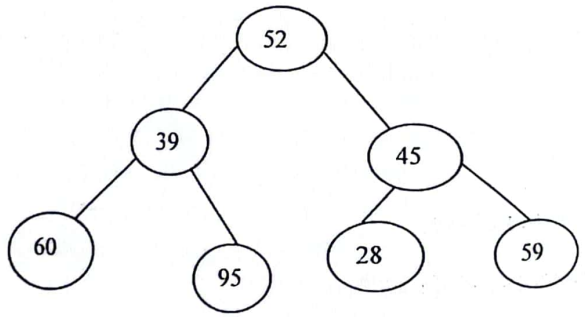
```
a[i] =a[1];          //put first data is at the last place of the array.

while (i < s)
    {
    if (a[i] < a[i+1]  //find the greater child.
    i = i +1;
    if (temp >= a[i] break;
   a[i/2] = a[i];      //put the data of index i at the parent's position.
    i = 2*i;
    }                  //end of while
  a[i/2] = temp;
  }                    //end of rearrange function.
```

To create a heap the complexity is O (n). The complexity of rearrange function is O (log n) and to delete n-1 roots the complexity will be (n-1)log n. According to the rules of asymptotic notation, we can write the complexity of heap sort is O(n log n). The sorting process has been illustrated in figure 7-18. In the figure tree and its corresponding has been shown.

Step 1: Input an array with random data.

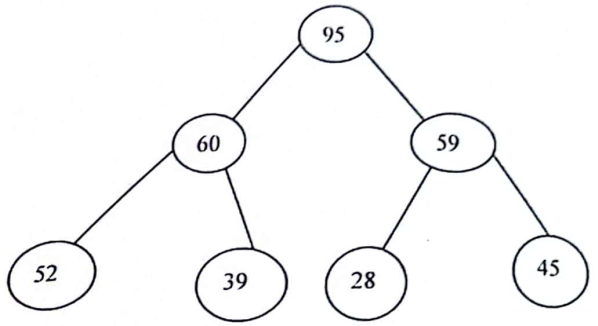
52	39	45	60	95	28	59

a) An array of random data



b) A binary tree for the above array

Step 2: Create a max-heap with the random data

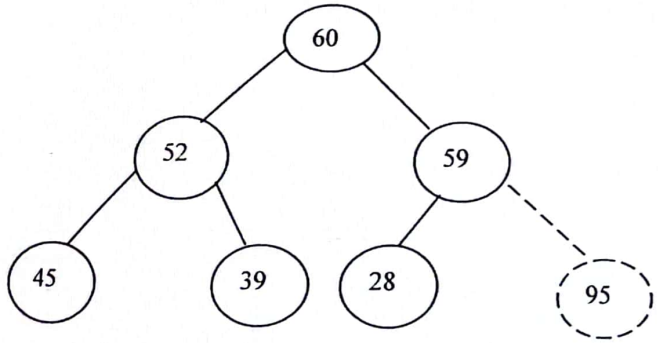


a) Max-heap of the above tree

95	60	59	52	39	28	45

b) Corresponding array of the heap

Step 3: Delete the max value in root, put it in the last position and rearrange data (except last data) a heap.

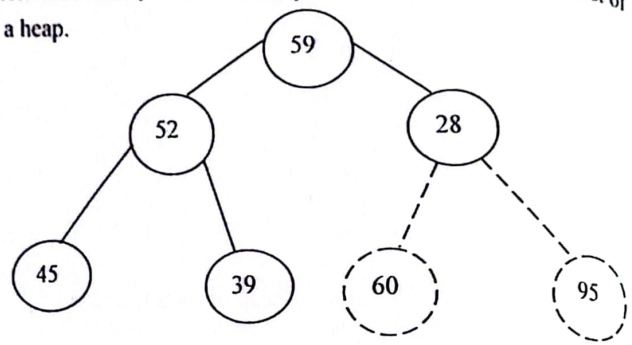


a) Max-heap of (n-1) data except for data in nth position.

60	52	59	45	39	28	95

b) Corresponding array of the tree

Step 4: Delete root value put it at the nth position and rearrange the rest of the data in a heap.

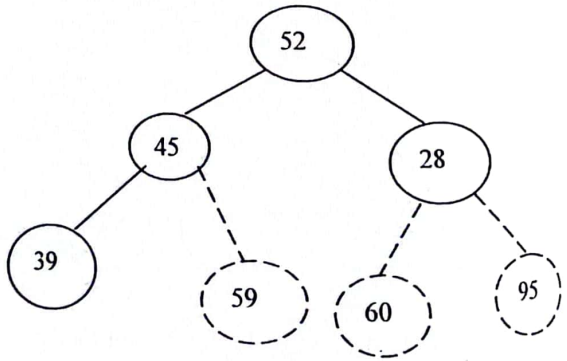


a) A heap of (n-2) data (except last two data of (n-1)th and nth positions).

59	52	28	45	39	**60**	**95**

b) Corresponding array of the tree

Step 5: Delete root value put it at (n-3)th position.



a) A heap of (n-3) data.

52	45	28	39	59	60	95

Sorted data

b) Corresponding array of the tree

28	39	45	52	59	60	95

Sorted data in the array (after repeated deletion)

Figure 7-18: Pictorial view proc of the heap sort process.

7.3.4 Priority Queue

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

One application of priority queue is to schedule jobs on a multiuser system. The jobs are stored in a priority queue and to be performed according to their priorities when a job is finished, the next job is selected based on the priorities from the remaining jobs. Let us consider the following example.

Table 7.1: Data with their priorities

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 abbreviated data according to their priorities as follows:

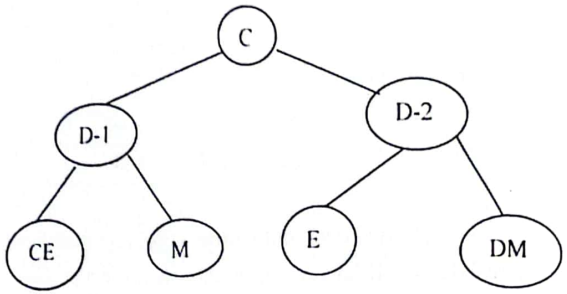


Figure 7-19: A priority queue in a tree

We can build a priority queue using a linked list also. If a linked list is constructed using the data of Figure 7.18, the figure will be as below.

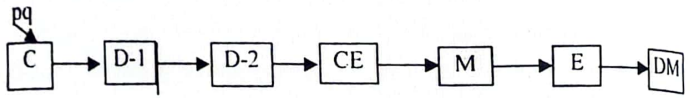


Figure 7-20: A priority queue in a linked list

From any list of data with their priority numbers, we can sort them according to priority numbers and construct the linked list or a heap, which will be a priority queue. In the above priority queue, the position (index) indicates the priority number of the respective member. The operations on priority queue are same for the heap.

7.3.5 AVL tree

AVL tree is a kind of Binary Search Tree (BST). BST is a complete or balanced tree in rare case. For a BST to maintain the shape of a complete tree for BST is very difficult. That is why any operation on BST (searching, insertion, deletion) cannot always be done in *log n* time. If any BST is a complete binary tree, the operation can be done in *log n* time. To perform any operation such as searching, insertion, deletion efficiently in 1962 Adelson-Velskii and Landis (AVL) introduced a binary search tree known as AVL tree, which is a balanced binary search tree. AVL tree is a binary search tree where the heights of the left and right sub-trees can differ by at

best 1 and the left and sub-trees have the same properties. That is, the left and right sub-trees are also AVL trees. In this regard, if the height of the left sub-tree is h, the height of the right sub-tree will be h-1 or h+1.

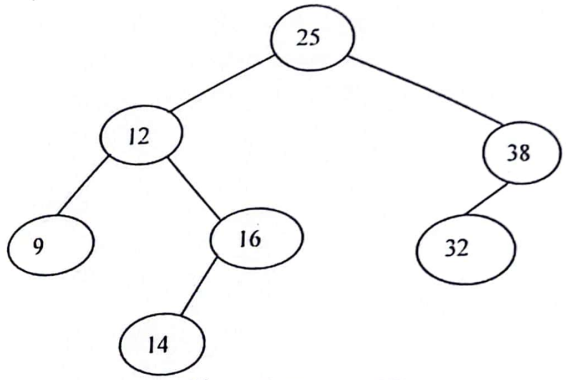


Figure 7-21: An AVL tree

In Figure 7-21 left sub-tree has the height 3 and the right sub-tree has the height 2, where the root has the height 0.

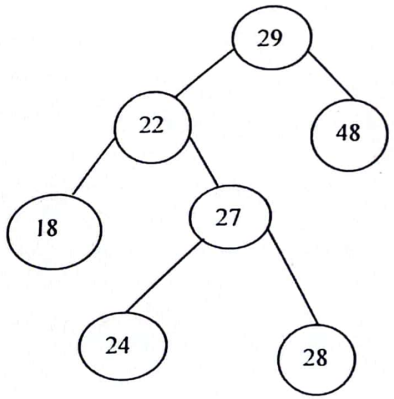
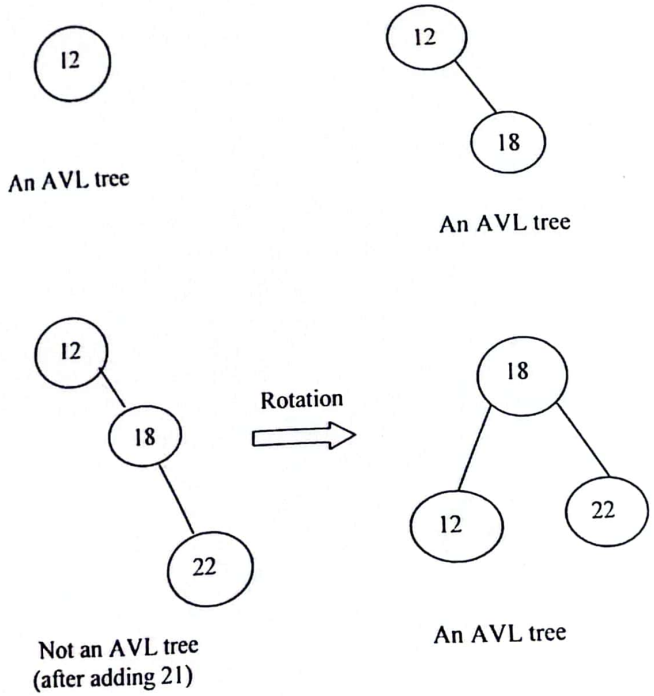


Figure 7-22: It is not an AVL tree

The tree in **Figure 7-22** is not an AVL tree since the height of the left sub-tree is 3 and the height of the right sub-tree is 1, although the tree is a BST.

Now we describe how we can create an AVL tree. Suppose we have a list of data as. 12 18 22 15 27 35. We have to create an AVL tree with the list of data. Since AVL is a BST usually it is built using a linked list. At first, we create a node with data and insert it into the tree. Similarly, we will create a node with other data and insert it into the proper position to get a BST. After inserting each node according to insertion procedure of BST we have oriented the node to get an AVL tree. Let us see the creation process in a pictorial view.



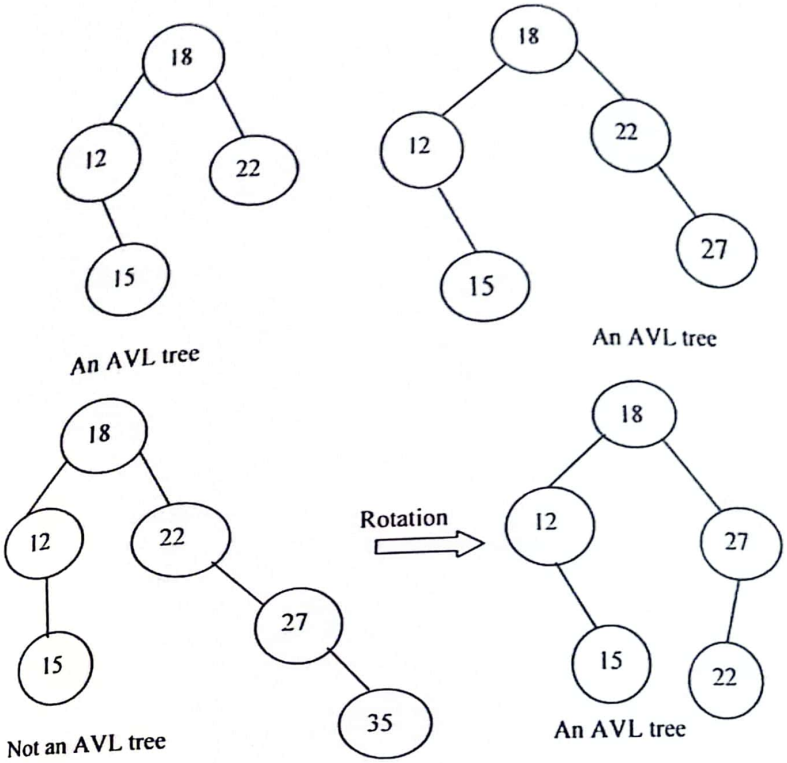
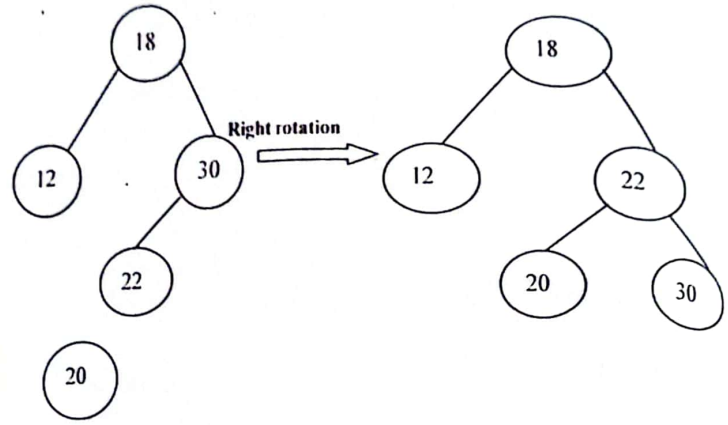


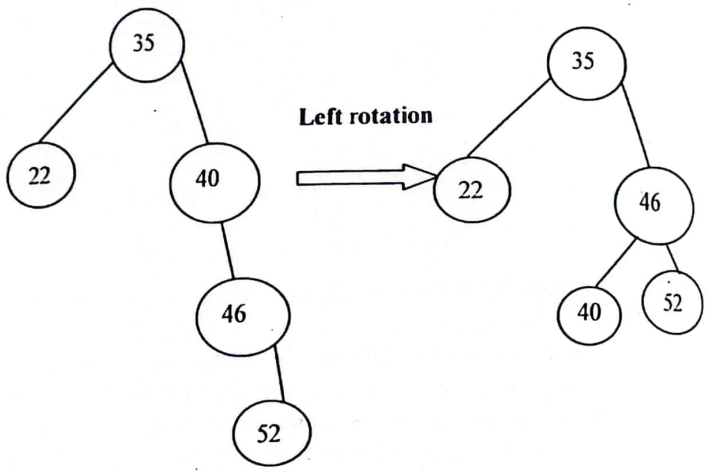
Figure 7-23: Pictorial view of AVL tree creation process

We have shown an orientation of the tree by a single rotation. However, in some cases, a single rotation is not enough to orient the tree to get an AVL tree and double rotation is necessary. Now we see examples of single and double rotations. Let us see how the rotation is performed. We say rotation to the right or *right rotation* if there is no right node to the root of the tree or sub-tree. Here the height of the right sub-tree differs from right sub-tree by 2. On the other hand, if there is no left node to the root of the tree or sub-tree we say rotation to the left or *left rotation*. The examples of rotation left or right is shown in the **Figure 7-24.** We have to perform rotation when heights of any two sub-trees differ by 2. We consider that the height of the root node is 0. If we see the sub-tree rooted at the node with value 30, the

left sub-tree of the node has the height 2, but the right sub-tree has the height 0. The difference is 2, so we need rotation here. This rotation will be considered as right rotation (rotation to the right).



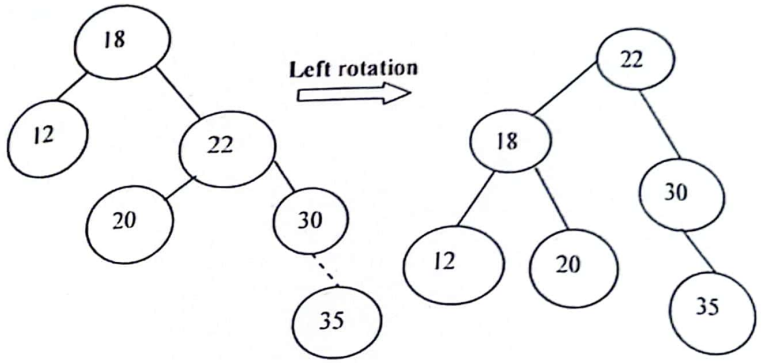
a) Not an AVL tree b) AVL tree after rotation



c) Not an AVL tree d) An AVL tree after rotation

Figure 7-24: Single rotation for an AVL tree

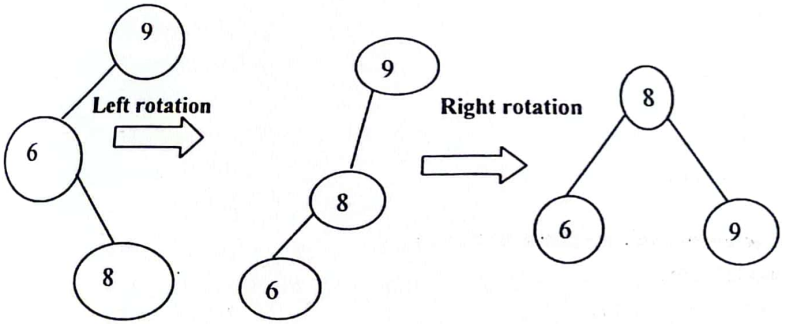
Suppose we want to add a node with data 25 to Figure 7-25 (a). After addition, the tree is shown without rotation in (a) and with rotation in (b) of Figure 7-25.



a) BST, but not AVL tree b) AVL tree after rotation

Figure 7-25: Single rotation to left for an AVL tree

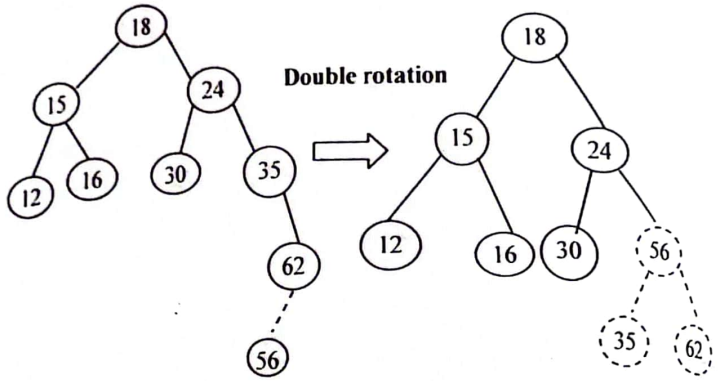
Now we see the situation of double rotation. In **Figure 7-26 (a)** the tree is not an AVL tree. To perform balancing the tree needs double rotation. At first, it need left rotation, then right rotation. The results of rotation are shown in (b) and (c) of **Figure 7-26.** We have needed the double rotation to orient the tree for an AVL tree.



a) Not an AVL tree b) After left rotation c) After right rotation (AVL tree after double rotation)

Figure 7-26: Pictorial view of double rotation

The tree in **Figure 7-27** is not an AVL tree after addition of the node with data 56 at the left side of the node with data 62. However, we can get an AVL tree by performing the double rotation. The nodes used in the rotation are shown in dashed shapes.



a) Not an AVL tree b) An AVL tree

Figure 7-27. Pictorial view of Double rotation

Node declaration for an AVL tree is as shown below.

```
struct node
 {
 int data;
 node *lchild;
 node *rchild;
 int hight;
  };
```

For a new node we can use code as,

```
node *nptr;
nptr->data = item;
nptr->lchild = NULL:
ntr->rchild = NULL;
nptr->hight = 0;
```


In algorithm 7.14 we have used a variable *item* to enter data for a new node. R is the pointer to root node of the tree. The algorithm returns R at the end. This algorithm can be used as a recursive function to add a node for an AVL tree. We can call it when we need to add a node. Suppose name of the function with parameters for the algorithm is add_node(node *R, int item). So, in step 3(i) we can call it as R->lchild = add_node (R, item). Similarly, we can call it in other places. For this type of code we can see the code to create a BST. This algorithm uses some other function such as a function of right rotation, left rotation. Left-right double rotation and Right-left Double rotation. Here we see the pseudo-code for left rotation and right rotation. For double rotation, we have to call both functions within a function.

Algorithm 7.14: Algorithm to add node to an AVL tree

```
1. Take a variable to enter data, item;
2. If (R = = NULL)
   {
    i) create a new node for AVL tree;
    ii) R=nptr;
    }
3.  if (item < R->data)
     {
    i) add a node as  a left child of R;
    iii) Find the difference between the heights of left child  and right child
of R, d
    ii) if (d = = 2)
         {
         if (item < R->lchild->data;
         Do right rotation with R;
         else
          Do Left-Right Double rotation with R;
    } // end of if in step 3.
4. if (item >R->data)
    i) add the node as the right child of  R.
    ii) Find the difference of the height of right child and left child of R, d
    iii) if (d = = 2)
         {
         if (item > R->rchild->data)
           Do Left roration with R;
         else
```

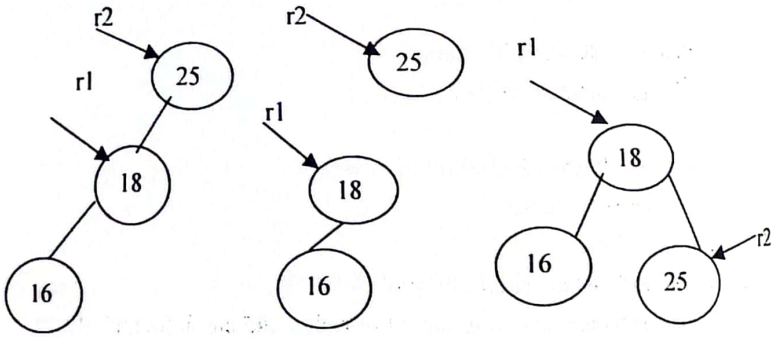
Do Right-Left Double rotation of with R.

5. Find the maximum of heights of the left child and the right child of R. max.
6. Update the height as R->hight= max + 1;
7. Return R.

The procedure for right rotation and left rotation are given below.

```
node * right_rotat(node *r2)
{
        cout<<endl<<"Rotat to the right."<<endl;
        node *r1;
        r1 = r2->lchild;
        r2->lchild = r1->rchild;
        r1->rchild = r2;
         h2 = max(Height(r2->lchild), Height(r2->rchild));
        r2->hight = h2+1;
        int h1= max(Height(r1->lchild), r2->hight);
        r1->hight = h1+1;
        return r1;
}
```

Let us see working process of the *right rotation* procedure.



a) r1 is at left child of r2 b) r2->lchild is null, since r1->rchild is null c) r1->rchild = r2

```
node * left_rotat(node *r2)
 {
```



```
        cout<<endl<<"Rotat to the left."<<endl;
        node *r1;
        r1 = r2->rchild;
        r2->rchild = r1->lchild;
        r1->lchild = r2;
        int h2 = max(Height(r2->rchild), Height(r2->lchild));
        r2->hight = h2+1;
        int h1 = max(Height(r1->rchild), r2->hight);
        r1->hight = h1+1;
        return r1;
 }
```

The function of finding the height of any sub-tree is given below.

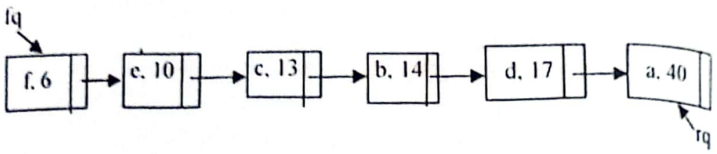
```
int Height(node *r)
{
        int h;
        if(r = =NULL)    h = -1;
        else h = r-> hight;
        return h;
```

7.4 Huffman Tree and Encoding

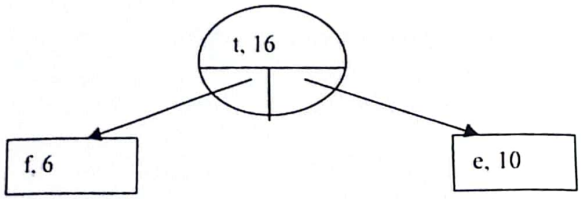
Huffman tree is needed to create Huffman code and this code is used for information coding as well as in data compression. Using variable length coding we can save 20% to 90% of total number of bits of a text file. For variable length coding, we have to create a tree that is called Huffman tree. To construct a tree let us consider a text with characters a, b, c, d, e and f only. The frequencies of these six characters are 40, 14, 13, 17, 10 and 6 thousands respectively. Here a total of frequencies of six characters is 100. To create a tree we can follow the points given below :

1) Sort the characters in ascending order of their frequencies.
2) Construct a linked list (priority queue)

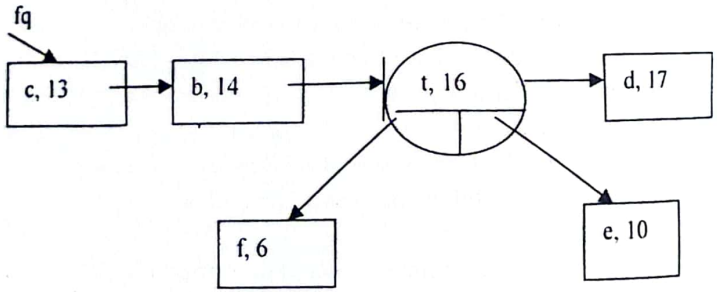
The linked list for the above data is as follows where frequency is shown with the character.



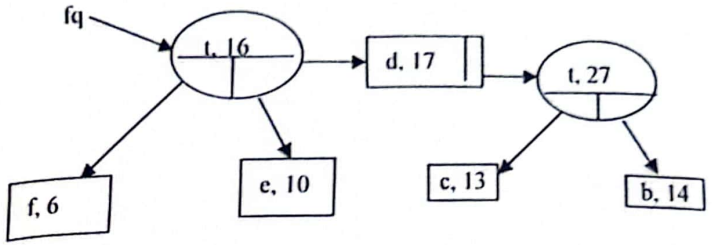
3) Create a new node which contains the value (data) equals to the sum of the first two nodes.
4) Make the first node of the list as a left child of the new node and second node of the list as a right child of the new node as shown in the following figure.

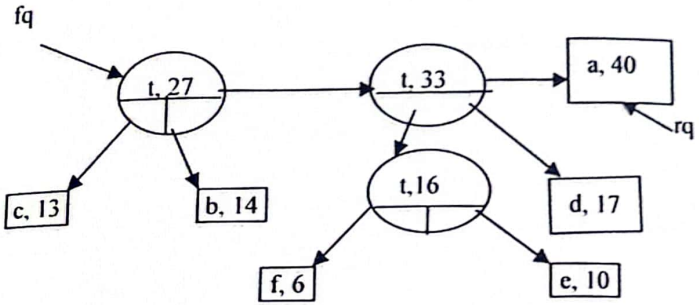


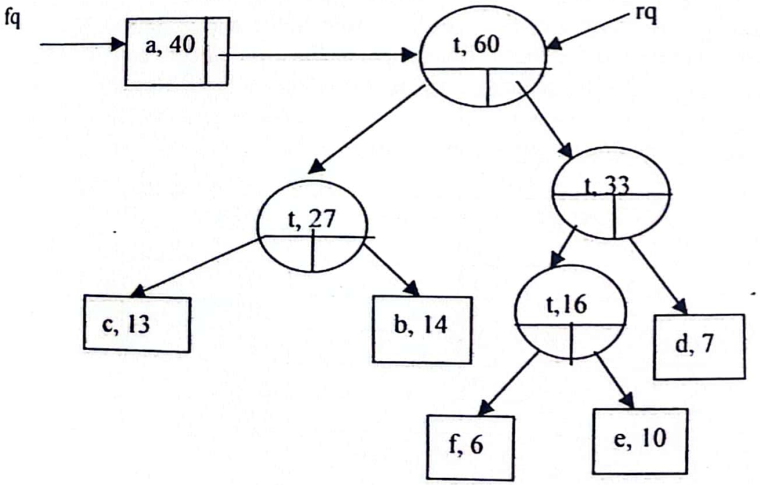
5) Insert the new node at the proper position of the priority queue (linked list).

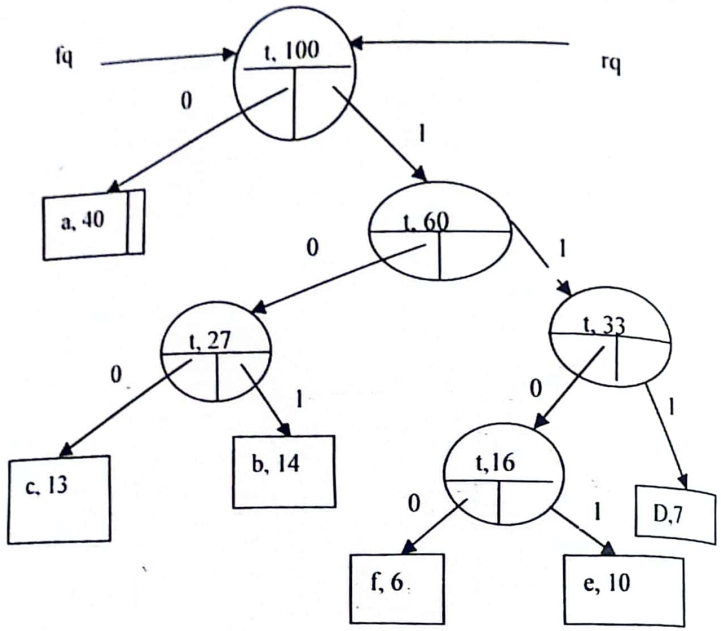


6) Repeat the steps 3 to 5 until we get the value (data) of the new node is 100, which is the total of frequencies of the characters. We can see the development of the linked list (priority queue) as follows:









7. After constructing tree we assign 0 to left edge and 1 to the right edge starting from the root of the tree (it showed in the above figure). We assign a value (0 or 1) to the edges from parent to the child node. From the tree, we can get the code for each character as shown in Table 7.1.

Table 7.1: variable length Huffman code

Character	frequency	Code
a	40	0
b	14	101
c	13	100
d	17	111
e	10	1101
f	6	1100

Implementation issue:

Now we describe how we can implement the above process of tree construction. Each node of the tree needs character, frequency, and three pointers, the node declaration is as follows:

1. Node declaration

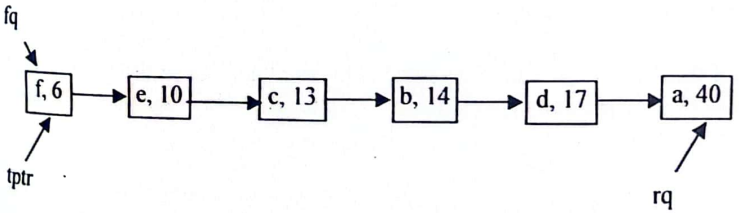
```
Struct node
{
char ch;
int freq;
node *next;
node *left;
node *next;
};
```

2. For the leaf nodes (the nodes of the initial lists) left and right pointers will be null. A leaf node can be created using following code:

```
node *nptr;
nptr = new(node);
nptr->ch = c1;
nptr->freq = fr1;
nptr->next = null;
nptr->next = null;
nptr->next= null;
```

Now we can create a linked list similar to the linked list created in chapter 3 (see algorithm 3.1).

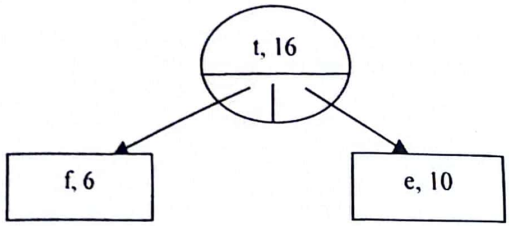
3. Now we see how a parent node can be created. As we described the parent will contain data(value) equals to the sum of the first two nodes and these two nodes will be the children of the parent node. Let us see the following figure (linked list, which was created before).



We can create a parent node as follows:

```
tptr = fq;
v1 = tptr->freq;
v2 = tptr->next->freq;
nptr = new(node)
nptr->ch = 't'; //t is put as a character value of the parent node
nptr->freq = v1+v2; //frequency of the parent node
nptr->next = null;
nptr->left = tptr;
nptr->right = tptr->next;
```

The figure related to the above code is shown below.



4. After this fq will point the node with character c and frequency value 13. We can do this as follows:

```
fq=fq->next->next;
```

5. Now we have to insert the new node (parent node) at the proper place. To insert a parent node we have to check the following conditions:

i) If the freq(data) of the new node is less than that of the node pointed by fq at present, we add the new node before the node pointed by fq. Condition: if(nptr->freq < fq->freq)

ii) if the data (freq) of the new node is greater than that of the node pointed by rq (last node), add the new node after the last node of the list. Condition: if (nptr->freq > rq->freq)

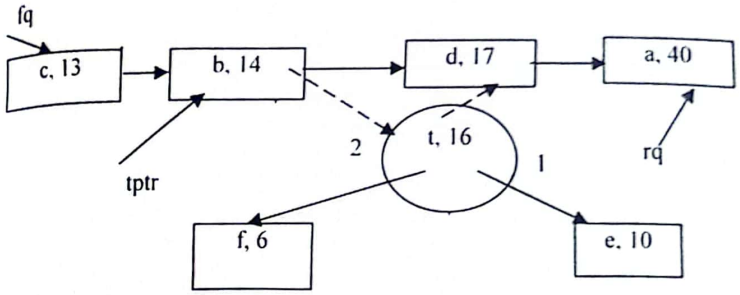
iii) Otherwise, the new node should be inserted at any place between the first node and the last node. To do this, locate the node after which the new

node should be inserted. The pointer tptr can locate (point) the node if we use the following code:

```
tptr = fq;
while (tptr->next->freq < nptr->freq)
    tptr = tptr->next;
```

The following code will help us to insert the node:

```
1. nptr->next = tptr->next;
2. tptr->next = nptr;
```



6. We continue this process until we get the root node and the value of the root node will equal to the total value of the all the frequencies.

After creating the tree we can check whether construction is correct or not. For this, we can use **pre-order** traversal method and **print** the data of the tree.

Now we describe how the code of each character can be assigned and printed. For this, we will use the Huffman tree, a stack (an array) to hold the code for the character. We use a function *codeprint()* to assign and print the code for the character.

```
codeprint (node *root, int stack[], int top)
{
//Assign 0 to left edge recursively
        if(root->left != NULL)
```

```
        {
                stack[top] = 0;
                codeprint (root->left, stack, top+1);
        }
        //Assign 1 to right edge recursively
        if(root->right!=NULL)
        {
                stack[top]=1;
                codeprint (root->right, stack, top+1);
        }
        // If the node is a leave node, it contains one of the input
characters and
        // we can print the character and its code
        if ((root->left==NULL) && (root->right==NULL))
        {
                cout<<root->ch<<":";
                for(i = 0; i < top; i++)
                {
                        cout<<stack[i];
                        cout<<endl;
                }
        } //end of if
} //end of function
```

Let us consider that the tree is constructed the tree using the function huffmantree(). We can call codeprint() as follows in main().

```
root = huffmantree();
int stack[6];
int top = 0;
codeprint (root, stack, top);
```

The type of the function *huffmantree()* should be a pointer as the type of the variable root is a pointer.

Summary:

The concept of data structure tree is obtained from the concept of a natural tree, except that a 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 specially 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 the *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 the *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 the *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**. The 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 an array and linked list.

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

Questions:

1. How can a binary tree be stored in computer memory using an array? Explain with example.
2. Define complete and full binary trees.
3. Describe inorder tree traversal method with an example.
4. What are the ways to represent a 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

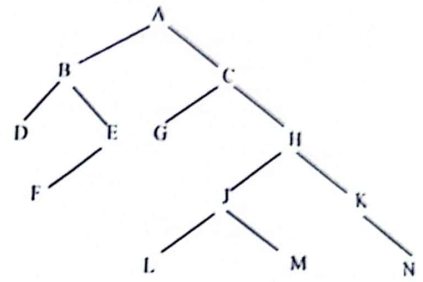
7. Write a procedure that deletes a node from a BST having only one or no child.
8. Discuss linked representation of a binary tree.
9. What will be the depth of a complete binary tree which has 3000 nodes?
10. What will be the sequence obtained by in-order and post-order traversal.
11. Suppose a binary 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 a binary tree.
14. Write an algorithm to create a heap.
15. Explain Prim's algorithm with an 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 the *i* th node of a binary tree. Show that, the relationship of the parent of the *i* th node and the relationship of the *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 an 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 [•] is the floor function.

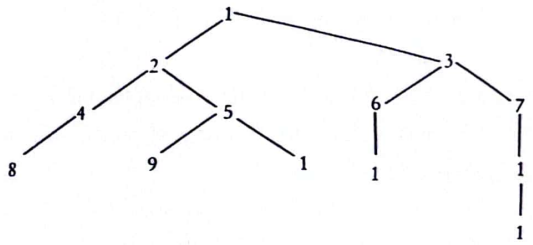
27. Construct a heap from the following list of numbers:

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

28. Write reheapdown function for implementing heap.
29. Draw a 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 the 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: Change your program which is written for problem 7-3 and execute it for tree shown in Figure 7.5.

Problem 7-5: Write a program for in-order traversal method using the stack (see algorithm 7.4, which is written for in-order method). Execute your program for a tree in Figure 7.5.

Problem 7-6: Write a program for post-order traversal method using the stack (see algorithm 7.5, which is written for post-order method). Execute your program for a tree in Figure 7.5.

Problem 7-7: Write a program to create a linked binary tree and print the data using pre-order traversal method without using a recursive function.

Problem 7-8: Draw a BST with 14 integers in the paper (note book). Store the data of the BST in an array. Now print:

i) All the data using in-order method.

Hint : the output will be in ascending order.

Problem 7-9: 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 7-10: 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 7-11: 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 7-12: 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 7-13: 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.

Problem 7-14: Create an AVL tree of 10 integers (for help see algorithm 7.14), print the height of each sub-tree and data of the tree in pre-order and in-order methods. If you face problem to write procedures for double rotation, you can use the following data set, which does not need double rotation. Data set: 12, 18, 22, 15, 27, 35. The height of left sub-tree will be 1 and the height of right sub-tree will be 2. If your program is okay for a single rotation, next you can try to add the procedure for double rotation and execute your program using random data. As for example the small data set 30, 25, 28 needs double rotation.

Problem 7-15: Create a Huffman tree using character a, b, c, d, e, f with frequencies 30, 13, 20, 10, 22, 5. Print the characters and their frequencies in pre-order and in-order methods.

Problem 7-16: Modify your program written for problem 7-15 and print code for each character according to Huffman coding method.

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 represents a graph, $G = (V, E)$ where V denotes set of vertices and E denotes set of edges. An edge contains two vertices. In Figure 8-1 V_1 is a vertex and AB is an edge.

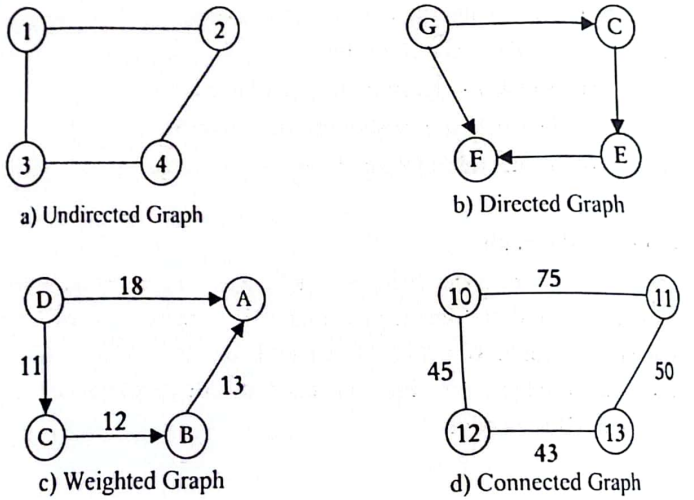
Figure 8-1: Vertex and Edge

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. Figure 8-2(a) shows an undirected graph.

Directed Graph: If each edge of a graph is directed (with direction), the graph is called directed graph. A directed graph is given in Figure 8-2(b).

Weighted Graph: In a graph, if the value (cost) of each edge is given, the graph is called weighted graph. A pictorial view of a weighted graph is shown in Figure 8-2(c).

Connected Graph: A graph is called connected when there is a path between each pair of vertices and in a connected graph every vertex is reachable. A graph is in Figure 8-2(d) shows a connected graph.



a) Undirected Graph

b) Directed Graph

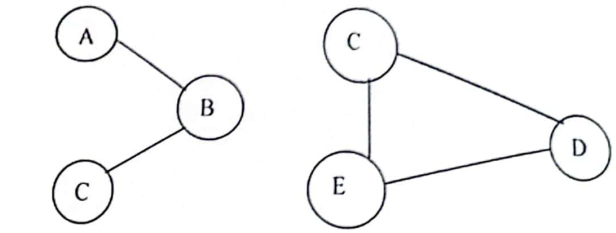
c) Weighted Graph

d) Connected Graph

Figure-8.2: Different types of graphs

Path: A path is a sequence of edges between two particular vertices where each pair of successive vertices is connected by an edge. Figure 8-3(a) depicts a path between vertices A and C.

Cycle: A cycle is a path where first and last vertices are the same. A cycle is shown in Figure 8-3(b).



a) A path

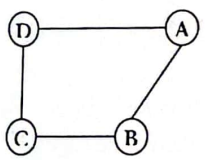
b) A cycle

Figure 8-3: A path and a cycle

8.2 Representation of a graph in computer memory

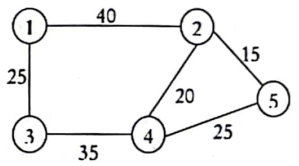
A graph is stored in the computer's memory using a matrix (two-dimensional array) or a linked list. The matrix used to store the graph in computer's memory is called *adjacency matrix*. Two vertices are *adjacent* to each other if there is an edge between them. If a vertex is adjacent to other, we put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

In case of weighted graphs, the values of edges are put in the adjacency matrix.



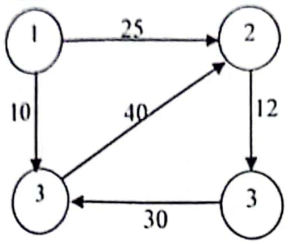
	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

(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



	1	2	3	4
1	0	25	0	10
2	0	0	12	0
3	0	0	0	30
4	0	40	0	0

(c) Adjacent matrix of a weighted and directed graph

Figure-8.4: Adjacency matrices of different graphs

For *implementation issue* we can take the above adjacency matrix of Figure 8.4 (c) as follows:

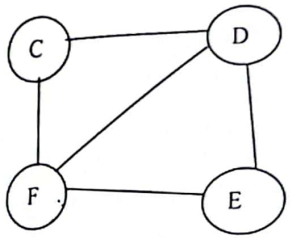
```
int m[5[5] = {
              {0, 25, 0, 10},
              {0, 0, 12, 0},
             {0, 0, 0, 30},
            {0, 40, 0, 0},
            };
```

In the above adjacency matrix we have put 0 if there is no edge or there is opposite edge direction. In same case we can put infinity or large number also. Such as, in the above matrix we can put 0 if there is no edge and 999 if there is opposite edge direction.

	1	2	3	4
1	999	25	0	10
2	999	999	12	999
3	0	999	999	30
4	999	40	999	999

Figure 8-5: An adjacency matrix of a directed graph.

Let us consider the graph of Figure 8-6 (a) for implementation issue and we want to print adjacent to each edge. For this, we have to consider one thing, how we can consider each vertex such as C, D, E and F as indices of the adjacency matrix. Actually, we cannot do it directly, since indices of a matrix are 0, 1, 2, and so on. However, we can do it indirectly using a character array where each index will match with each charter. For this, we can use *mapping array* shown in Figure 8-6(c). The pseudo-code for input the data of the graph is shown below.



a) An undirected graph

	C	D	E	F
C	0	1	0	1
D	1	0	1	1
E	0	1	0	1
F	1	1	1	0

b) Adjacency matrix

0	1	2	3
C	D	E	F

c) Mapping array

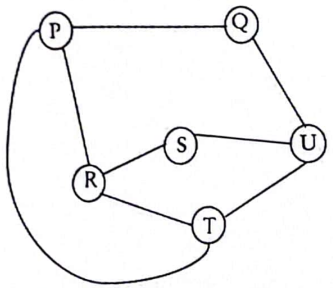
Figure 8-6: A graph and a mapping array

The pseudo-code is given below.

```
int m[4][4]= {
              {0, 1, 0, 1},
              { 1, 0, 1, 1},
             {   0, 1,0,1},
             {1, 1, 1, 0},
              };
char map[4] ={ 'C', 'D', 'E', 'F'};

for (i  = 0; i < 4;i++)
```

```
for (j = 0; j < 4;j++)
 {
if(m[i][j] != 0)
cout<<" "<<map[j];
}
```

Graph can be stored (represented) as adjacency lists in computer's memory.



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-8.7: An undirected graph and its adjacency list

Since adjacency lists are variable in length, so linked list implementation will be efficient for this type of lists. Linked representation of Figure 8.7 is shown in Figure 8.8.

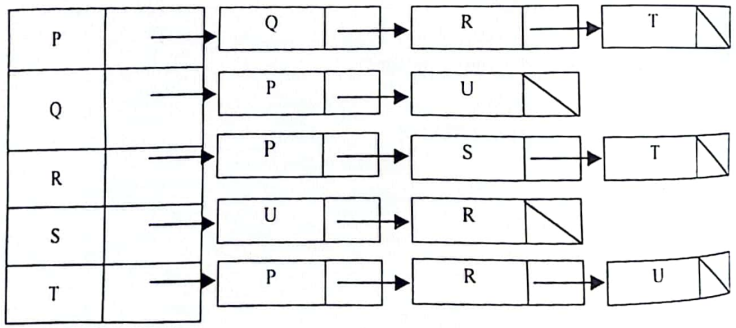


Figure-8.8: Linked representation of the adjacency lists

We can easily implement the above linked list using C/C++. The algorithm (pseudo-code) to create adjacency list is given below.

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;
           vertex  * next;
          }
        ii) vertex table [1.....m], * nptr, * tptr;
for (i=0; i<m; i++)
 {
```

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

 } //end of for loop
```

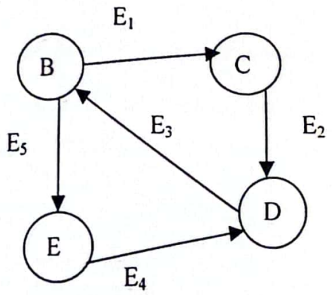
9. Output a linked adjacency lists.

Comments: m is the number of vertices, the table is an array of vertex type.

Incidence Matrix:

Incidence matrix is a matrix represented by a total number of vertices by the total number of edges. An adjacency matrix is a square matrix, but an incidence matrix may be a square or a rectangular matrix. In an incidence matrix cells are filled with either 0 or 1 or -1. Here usually vertices represent for rows and edges represent for columns. If there is an outgoing edge from a vertex we take 1 for the cell representing row to a column. If there is an incoming edge to a vertex we take -1 for the cell. For others we take zeros, that means if an edge is neither outgoing from nor incoming to a vertex, we take zero for this cell.

For example, consider the following directed graph representation using incidence a matrix.



a) A directed graph

	E_1	E_2	E_3	E_4	E_5
B	1	1	-1	0	0
C	-1	1	0	0	0
D	0	-1	1	-1	0
E	0	0	0	1	-1

b) An incidence matrix

Figure 8-9: A directed graph with its incidence matrix.

8.3 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)

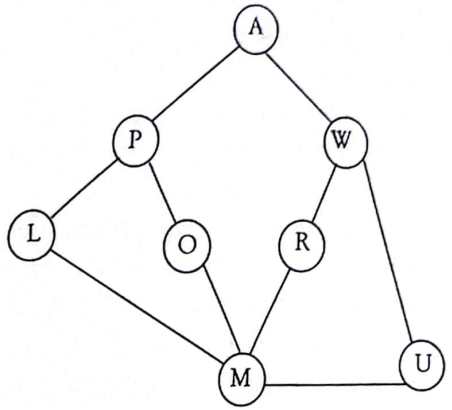


Figure-8.10: An undirected graph

8.3.1 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, the value of the vertices) that will be visited after visiting the current vertex. Therefore, we have to store the values of the vertices to be visited later 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 the first column of the table, we have enlisted in *an array* the vertex and its adjacent vertices. In the second column of the table, we have shown the vertices stored in a *queue*. In the third column we have enlisted the vertex for which traversal is done.

According to the Figure 8.10, we have taken (visited) the vertex A first and we store two adjacent vertices W and P in the *queue*. So, we have enlisted the vertices A, W, P in the array in the first column, and the vertices W, P in

the queue in the second column. In the third column, we have put the vertex, whose adjacent vertices are already in the queue. So, we put A in the first row of third column (done list). Now the element from the queue will be accessed from the front of the queue and proceed accordingly. The bold character in the queue is the front element.

As we know the vertex which is added to queue first, gets service first, so we traverse the vertex W now. We say traversal of a vertex is done if its adjacent vertices are stored in the queue. Thus we have enlisted the adjacent vertices of vertex W in the array of the first column and stored the vertices (O, L) in the queue and put the vertex W in the done list. The next front element is P, we visit the adjacent vertices of P and store them (the vertices U and R) in the queue and the traversal of the vertex P is done. The progress of this traversal method has been shown in Figure 8.11. Following the above method, we traverse the other vertices of the graph. The traversal of the graph is done when the queue is empty.

Table-8.1: Breadth first traversal of the graph in Figure-8.10

a) Visited list and queue

Visited list	Queue
A W P	**W** P
A W P U R	**P** U R
A W P U R O L	**U** R O L
A W P U R O L M	**R** O L M
	O L M
	L M
	M

b) Traversal done

Done list
A
A **W**
A W **P**
A W P **U**
A W P U **R**
A W P U R **O**
A W P U R O **L**
A W P U R O L **M**

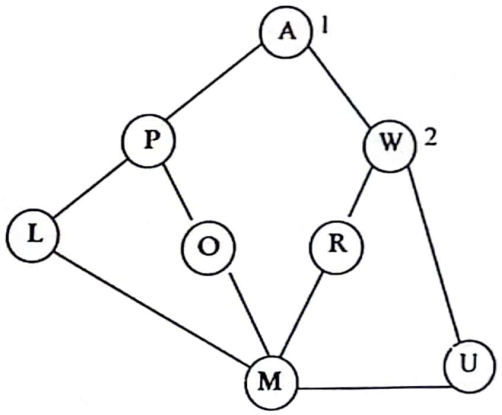


Figure 8-11: The progress of traversal sequences is shown by marking 1, 2 (the shaded vertices).

A spanning tree of graph is a sub-graph that has no cycle. The BFS produces a spanning tree, which is shown with traversal sequence in Figure 8.12.

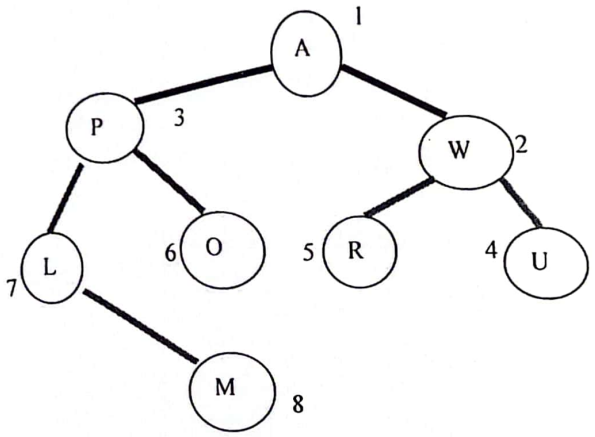


Figure 8-12: A spanning tree produced by BFS

The algorithm for BFS method is given in Algorithm 8.2.

Algorithm 8.2: Algorithm for breadth first search

1. Input a graph, G (adjacency matrix of the graph).
2. Take two arrays L and T
3. Create an empty queue (array) Q
4. Take vertex v from G
5. Put the vertex v and its adjacent vertices in L
6. Add the adjacent vertices of the vertex v in Q.
7. Put the vertex v in the done list, T.
8. Repeat the steps 9 and 10 until Q is empty.
9. Access a vertex w from the Q
10. For each adjacent vertex of w do

i) If it (the adjacent vertex of *w*) is not in L, put it in L and add it to Q

ii) Place vertex *w* in T

11. Output the traversal done list in T, which is a spanning tree.

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

8.3.2 Depth First Search (DFS)

In this method, 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. In Depth First Search (DFS) a *stack* is used to hold the adjacent vertices of the currently visited vertex. Next, the top element from the stack is accessed and search for the adjacent vertices of the top element. Similar to BFS we have to use three arrays here. One is for the visited vertices next one is for a *stack*, which is used to take the next vertex to be visited. The third one is traversal done list. For this purpose, a table (Table 8.2) is maintained. In the first column of the table the visited vertices are enlisted, the *stack* with its elements is in the second column and the third column holds the array for the traversal done list.

Let us use graph of Figure 8.13 (the same graph was in Figure 8.10). We have used the same graph so that we can see the difference in visiting sequence in both methods. The vertex A has been taken as stating vertex. The vertex A and its adjacent vertices are enlisted in the first array (first column of the table). The adjacent vertices of vertex *A* are also stored in a *stack*. Since the adjacent vertices of A (vertices W and P) are stored in the *stack*, the traversal of the vertex, A is done and it is stored in the traversal done list (third column of the table). P is the last (top) element in the stack, we access it first. The adjacent vertices (O and L) of P are stored in the visited list and also in the stack. Now we can say traversal of the vertex P is done. Consequently, we store the vertex P in the done list. Now the last (top) element of the stack is L. The process will be continued for this vertex. We continue the process until the stack is empty. Thus we traverse all the vertices of the graph. The vertices stored in the array at current stage have been shown in boldface.

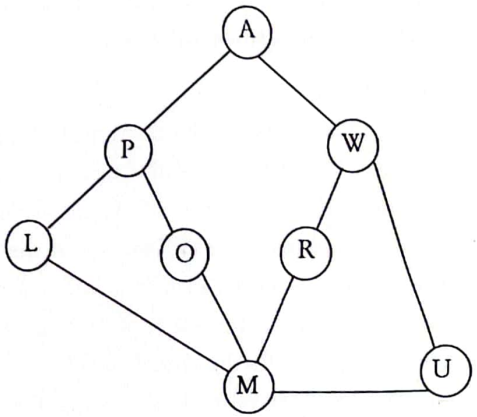


Figure-8.13 An undirected graph

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

a) Visited list and Stack

Visited list	Stack
A W P	W P
A W P L O	W O L
A W P L O M	W O M
A W P L O M U R	W O U R
	W O U
	W O
	W

b) Traversal done

A
A P
A P L
A P L M
A P L M R
A P L M R U
A P L M R U O
A P L M R U O W

A spanning tree has been produced by the DFS, which is shown in Figure 8.14 with the sequence numbers.

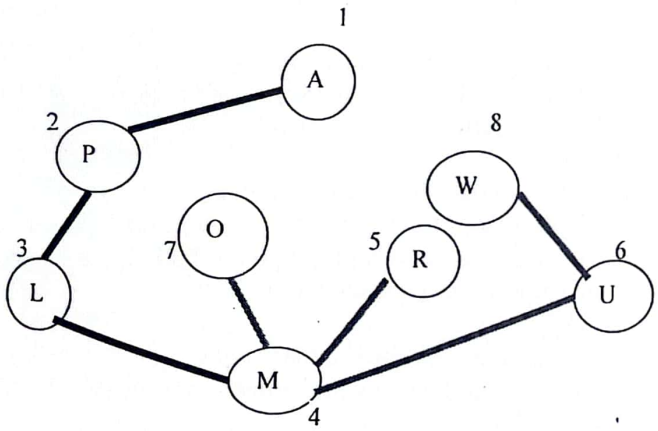


Figure 8-14: A spanning tree produced by DFS

Algorithm 8.3 is an algorithm for DFS method.

Algorithm 8.3: Algorithm for depth first search

1. Input a graph, G (Adjacency matrix of the graph).
2. Create two empty lists (arrays) L and T.
3. Create an empty stack S.
4. Take a vertex v from G.
5. Place the vertex *v* and its adjacent vertices in L.
6. Add the adjacent vertices of the vertex *v* to S (stack).
7. Put the vertex *v* in T
8. Repeat the steps 9 to 10 until S is empty.
9. Access a vertex *w* from the top of 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.
11. Output the traversal list in T.

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

Now we see how we can implement Algorithm 8-3 using C/C++. We write pseudo-code that will help us to implement the algorithm. We write pseudo-code with respect to the graph in Fig. 8.9.

See the pseudo-code of algorithm 8-3 below.

1. Take a matrix m[8][8] and initialize the matrix as m[i, j] =1, if there is an edge between the vertex i and j. Otherwise
   m[i, j]=0;
3. Take an array to store the vertices as follows:
   Char map[8]={'A', 'P', 'W', 'L', 'O', 'R', 'U', 'M'};
3. Take three character arrays VL[8], S[8] and T[8]. One is for visited vertex list, one is for stack and other is for final
   traversal list.
4. Take three indicator variables for three arrays as, k=0, top=0, and h= -1;
5. Initialize vertex list and stack S with first vertex, VL[0] = map[0]; S[top] = map[0];
   (The first vertex is in the visited list and in the stack).
6. a) Now we process until the stack is not empty.


```
   while (top >= 0)
    {
b)  for ( i = 0; i < 8; i++)  //We find which vertex of the graph is in the
stack and save its index
    if (S[top]==map[i] { u = i, break;} //save the vertex index

c) ++h; T[h]= S[top];--top; //pop the first vertex from the stack and save it
to final traversal list
d) Find the adjacent vertices of the vertex number, u. For this, we have to
see all the columns of the matrix
  for (j = 0;j < 8; j++)
    if (m[u][j] != 0)  //if there is an edge from u to j
      {
   flag = 0;
 for (i = 0;i < k; ++i) //for all vertices in the VL [ ]
  {
 If (map[j] = =VL[i]) flag =1 //if the adjacent vertex is already in the VL [ ]
 }
 If (flag = = 0)    //if the vertex is not in the VL [ ]
 {
 ++top; ++k;
 S[top] = map[j]; VL[k] = map[j]; //save the vertex in stack and visited list
 (VL[ ]).
 }
 } //end of if m[u][j]= =0 an d for in point  d
 } //end of while
7. Output is the T, the final traversal list is the T [ ].
```

Here pseudo-code has been given as the outlines for implementation. Students can implement the algorithm using their own coding.

8.3.3 Efficiency of Implementation of DFS and BFS using different data structure of the 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 $\Theta(V^2)$
- 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:

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

8.4 Minimum cost spanning tree

Spanning tree: It is a sub-graph of a graph, which contains all the vertices of the graph and has no cycle. If there is a graph G with V_G is the number of vertices and E_G is the number of edges in G, the spanning tree T will contain the number vertices, $V_T = V_G$ and the number of edges, $E_T < E_G$. There may be more than one spanning tree from a graph. If the graph is a weighted graph, their costs (cost of the spanning trees) will be not same.

Minimum cost spanning tree: It is a Spanning Tree (ST) whose total cost (sum of costs of all the edges of the tree) is minimum. For graph in Figure 8.15, there are three spanning trees. However, the minimum cost spanning tree is in Figure 8.16 (b).

There are two well-known algorithms to build minimum cost spanning tree. One is Prim's algorithm and another is Kruskal's algorithm. We describe these two algorithms below.

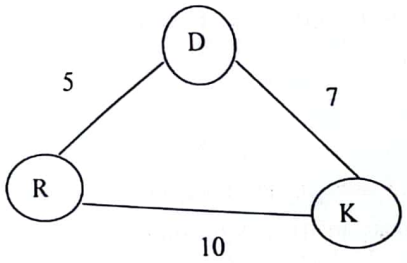


Figure 8-15: An undirected and weighed graph.

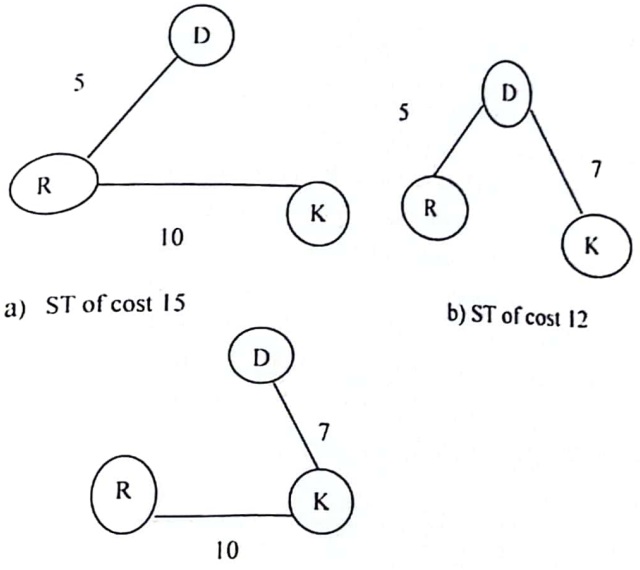


Figure 8-16: Spanning tress of graph in Figure 8-15.

8.4.1 Prim's algorithm

This method is used to build a minimum cost spanning tree. To create a minimum cost spanning tree according to Prim's algorithm the process is as follows. Since in a spanning tree the number of nodes is same as the number of vertices in the graph, so at first, we take all the nodes (vertices) to make a forest. Now n-1 vertices will be considered to select the edges if there are n vertices in a graph. Next, we choose a vertex and all related edges with costs and store in a *priority queue*. Here the edge of minimum value (cost) has highest or higher priority. From the queue, we select the edge with minimum cost and delete it from the queue. At the second step, we consider the vertex, which is at the other end of the selected edge. Now we take all the related edges and store them into the priority queue. We select the edge that has the highest priority (the edge with minimum cost) and delete it from

the queue. At any stage, the edge with minim cost will be at the beginning or front of the queue.

At the next step, we choose the vertex at the second end of the selected edge and store the related edges of the vertex and store them into the queue. We consider the edge with minimum cost and check whether the edge creates a cycle. If it does not create a cycle we select (accept) it otherwise we reject it. If any edge related to the vertex has been rejected, we will consider next edge until we can select an edge. Thus we consider a vertex, its related edges and store them to the queue. We will select the edge with the minimum cost if it does not create a cycle. The process will continue until we consider n-1 vertices. We consider the graph in Figure 8.17 and show the stages of Prim's algorithm.

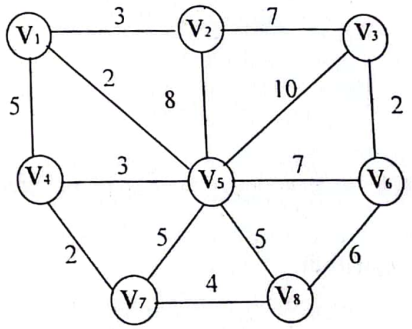


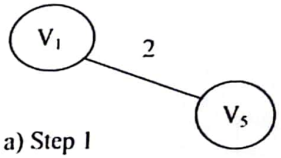
Figure-8.17: A weighted graph

(a) A priority queue

Edge	Cost
V_1V_5	2
V_1V_2	3
V_1V_4	5

1) a) Consider the vertex V_1, its related edges and all the related edges are stored in a priority queue in (a). The edge with minimum cost has been shown at the beginning of the queue.

b) We select the edge V_1V_5 and delete it from the queue.



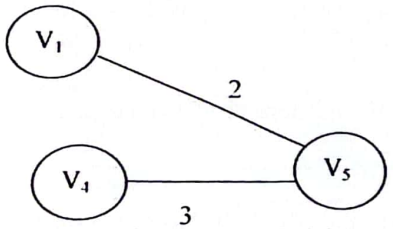
a) Step 1

2) a) Now we consider the vertex V_5 and store its related edges into the queue. The edge with minimum cost is at the beginning of the queue.

b) The edge V_5V_4 has been selected and will be deleted from the queue.

(b) Priority Queue

Edge	Cost
V_5V_4	3
V_1V_2	3
V_1V_4	5
V_5V_2	8
V_5V_3	10
V_5V_6	7
V_5V_7	5
V_5V_8	5



b) Step 2

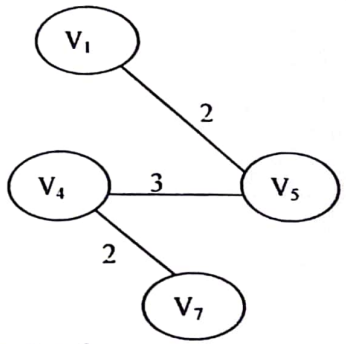
3) a) Next we consider the vertex V_4 and store its related edges into the

queue.

b) From the priority queue, the edge V_4V_7 has been considered since it does not create a cycle the edge has been selected and will be deleted from the queue [priority queue in (c)].

(c) Priority Queue

Edge	Cost
V_4V_7	2
V_1V_2	3
V_1V_4	5
V_5V_2	8
V_5V_3	10
V_5V_6	7
V_5V_7	5
V_5V_8	5



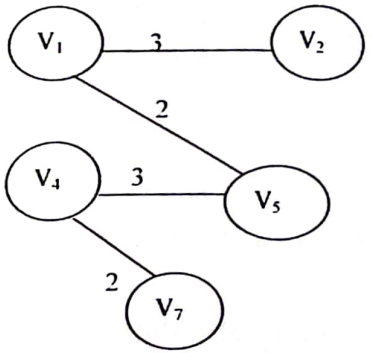
c) Step 3

4) a) Consider the vertex V_7 and store its related edges into the queue.

b) From the priority queue, the edge V_1V_2 has been considered since it does not create a cycle the edge has been selected and will be deleted from the queue priority queue in (d)].

(d) Priority Queue

Edge	cost
V_1V_2	3
V_1V_4	5
V_5V_2	8
V_5V_3	10
V_5V_6	7
V_5V_7	5
V_5V_8	5
V_7V_8	4



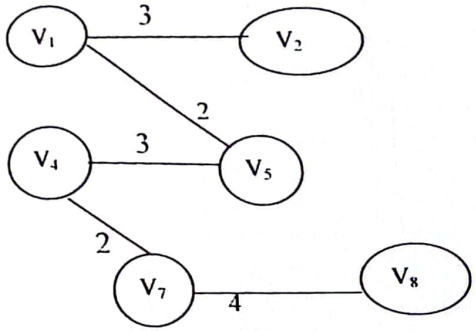
d) Step 4

5) a) Now we consider the vertex V_2 and store its related edges into the queue.

b) From the priority queue, the edge V_7V_8 has been considered since it has a minimum value and does not create a cycle the edge has been selected and will be deleted from the queue.

(e) Priority Queue

Edge	Cost
V_7V_8	4
V_2V_3	7
V_1V_4	5
V_5V_2	8
V_5V_3	10
V_5V_6	7
V_5V_7	5
V_5V_8	5



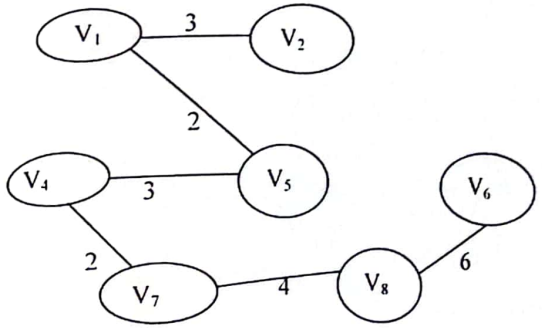
e) Step 5

6) a) Now we consider the vertex V_8 and store its related edges into the queue.

b) At this stage we cannot select any of the edges V_1V_4, V_5V_7, V_5V_8 any of these edges create a cycle, so from the priority queue the edge V_8V_6 has been selected since it does not create a cycle and will be deleted from the queue.

(f) Priority Queue

Edge	Cost
V_1V_4	5
V_5V_7	5
V_5V_8	5
V_8V_6	6
V_5V_6	7
V_2V_3	7
V_5V_2	8
V_5V_3	10



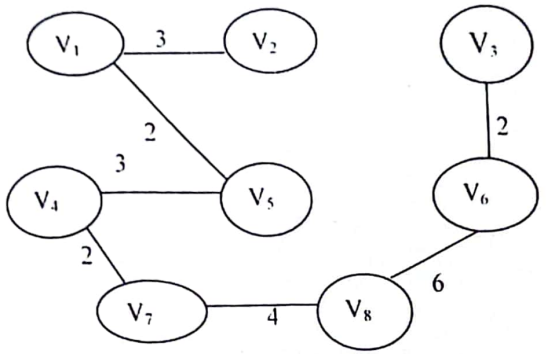
f) Step 6

7) a) Next we consider the vertex V_6 and store its related edges into the queue.

b) From the priority queue, the edge V_6V_3 has been selected since it does not create a cycle and will be deleted from the queue.

(g) Priority Queue

Edge	Cost
V_6V_3	2
V_5V_6	7
V_2V_3	7
V_5V_2	8
V_5V_3	10
V_5V_6	7



g) Step 7

Figure 8-18: Spanning tree according to Prim's algorithm

Since we considered n-1 vertices and all the vertices are connected by edges so the process ends here. The total cost of the spanning tree is 22, which is a minimum and hence the spanning tree is a minimum cost spanning tree.

Algorithm 8.4: Prim's Algorithm

1. Input a weighted connected graph, G = (V, E). V is a set of vertices and E is a set of edges.
2. V = {V_1, V_2, . . .V_n};
3. Create an empty edge list ET.
4. for (i =1; i <n; i++)

{

Select a vertex v from V and do

{

a) Find all the edges related to v and store them into a priority queue.

b) Find a minimum-weight edge among the edges consider it as a candidate edge.

c) If the edge does not create a cycle, select it and store it into ET and delete the edge from the queue.

}

} // End of for loop;

5. Output edge list of the spanning tree ET.

Implementation issue for Prim's algorithm:

Here we consider the implementation issue for the graph in (Figure 8. 17).

In Prim's algorithm, if any of two vertices is not visited we can select the vertex otherwise we can not select means it creates cycle

1. Define a list of structure, where the structure contains edge (vertex u and vertex v) and edge value as follows:

```
struct vt
{
        int u;
        int v;
        int val;
}tree[10];
```

2. Take a 2-D array m[8][8]; the data of the **first row** will be as follow:

{99, 3, 99, 5, 2, 99, 99, 99},

Here we take 99 if there is no edge between vertices. There will be eigh such rows.

3. Take a list (array) visited[ ], which indicates whether any vertex is considered or not.

Initially, visited [8]={0};

4. Take first vertex as visited or considered: visited[0]= 1;

5. n = 8, k = 0;

(n is the number of vertices, k if the number of edges selected yet).

6. while (k < n-1) // We have to take n-1 edges.

```
{
 min =99;
for(i=0;i<n;i++) //find minimum value from each row
for(j=0;j<n;j++)
  if(m[i][j]<min)
        if(visited[i]!=0)
         {
         min=m[i][j];
         u = i;  //save the edge (two vertices) with minimum value
         v = j;
         }
```

```
if(visited[u]==0 ||visited[v]==0)  //if any of the two vertices is not visited
        {
        tree[k].u=u;  //Select the edge as tree edge
        tree[k].v=v;
        tree[k].val=min;
        visited[v]=1; //consider vertex v as visited, so put 1 to visited list
        ++k;  //increase the number of selected edge
        }
m[u][v]=m[v][u]=99; //mark the selected edge or two vertices by putting
                    //99 as edge value
        } //end of while
```

8. The edge list in the list of structures, edges of the minimum spanning tree as follows:

```
for(i=0;i<k;++i)
    {
        cout<<tree[i].u+1<<" ,"<<tree[i].v+1<< " = "<<tree[i].val;
        cout<<endl;
   }// end of for loop
```

We can take any undirected and weighted graph and implement the above code with necessary modifications and test the code. The above code is just for assisting purpose. Students can write code in their own way and implement the algorithm.

8.4.2 Kruskal's algorithm

In this method, we create a forest of nodes first. Next, we take all the edges from the graph with their respective costs and sort them to create a sorted edge list. From this list, we consider edge that has minimum cost and check whether the edge creates a cycle. If the edge does not create a cycle we select it for the spanning tree. Otherwise, the edge will be rejected. In this way, we will select n-1 edges where there are n vertices in the graph. Finally, we get a spanning tree of minimum cost. Let us consider the graph of Figure 8-19 and Table 8-3. The progress according to the Kruskal's algorithm is shown below.

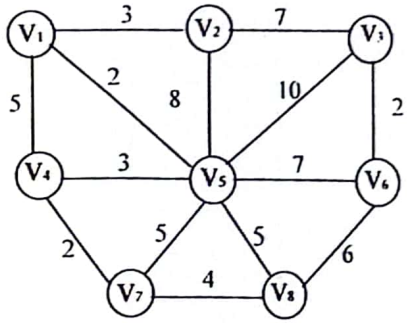


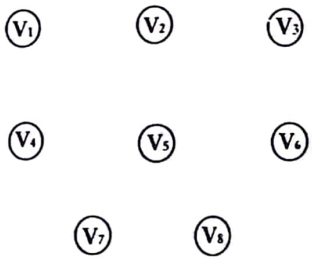
Figure-8.19: A weighted graph

Table 8-3: List of Edges with costs Edge Cost

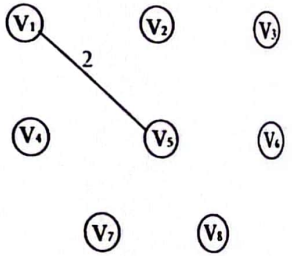
V_1V_2	3
V_1V_4	5
V_1V_5	2
V_2V_3	7
V_2V_5	8
V_3V_6	2
V_3V_5	10
V_4V_5	3
V_4V_7	2
V_5V_6	7
V_5V_7	5
V_5V_8	5
V_6V_8	6
V_7V_8	4

Table 8-4: Sorted list of edges with cost

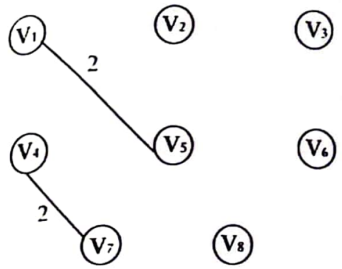
V_1V_5	2
V_3V_6	2
V_4V_7	2
V_1V_2	3
V_4V_5	3
V_7V_8	4
V_1V_4	5
V_5V_7	5
V_5V_8	5
V_6V_8	6
V_2V_3	7
V_5V_6	7
V_2V_5	8
V_3V_5	10



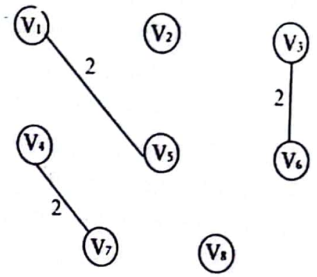
(a) Step 1 (Select all the nodes)



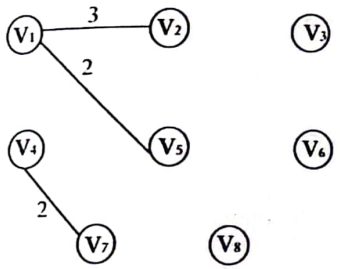
(b) Step 2 (Select an edge with minimum cost (minimum among all the edges))



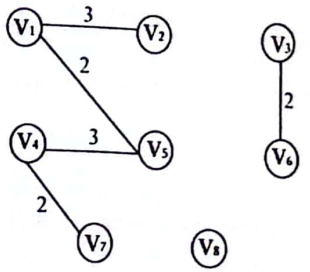
(c) Step 3



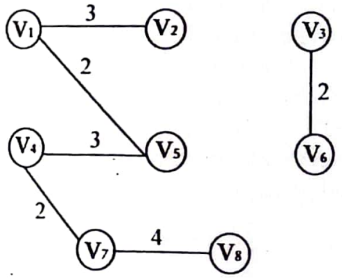
(d) Step 4



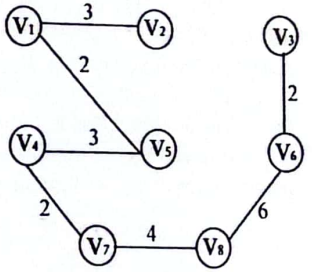
(e) Step 5



(f) Step 6



(g) Step 7



(h) Step 8, we could not select any edge with cost 5 since it creates cycle.

Figure-8.20: Pictorial view of the stages of the Kruskal's algorithm.

Algorithm 8.5: Kruskal's Algorithm

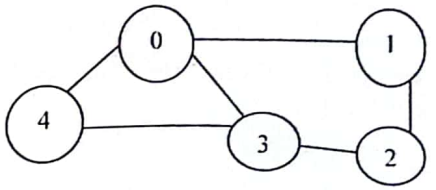
1. Input a weighted connected graph, G = (V, E). V is the set of vertices and E is the set of Edges
2. Create a sorted list ES and an empty list ET.
3. Take an edge E_k from the sorted list ES
4. Set ecounter = 0; k = 0;
5. while (ecounter < n - 1)

```
{
k = k + 1;
if (ET U { Ek } is acyclic)
ET = ET U { Ek };
ecounter = ecounter +1 ;
}
```

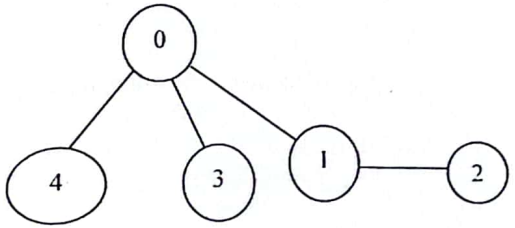
6. Output edge list of the spanning tree ET.

8.4.3 Determine whether an edge of a graph creates a cycle

We initialize the parent of each vertex as -1. For fiver vertices parents' values are {-1, -1, -1, -1, -1} (stage-1). Let us see a graph of Fig. 8.21 (a). We see the concept and the implementation issue. From vertex 0, there are three edges {0, 1}, {0, 3} and {0, 4}. At first, we consider the edge {0, 1} and select it as an edge of a tree shown in Figure 8.21 (b). Now we find the parents of the two vertices. If the parents are -1 and -1, we make one vertex as a parent of other. We make the parent of vertex 1 is 0 (see the figure) and the parent of vertex 0 is initially considered as -1. Now the next edge {0, 3} is considered. We select this edge as an edge of the tree considering the parent of vertex 3 is 0. Similarly, we can select {0, 4}.



a) A graph



b) A parent-child relationship of tree

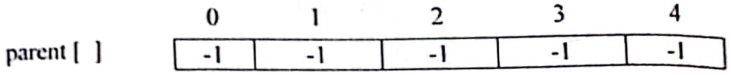
Figure 8.21: A graph and tree for cycle test.

Now let us consider the edges from the vertex 1 (adjacent vertices of vertex 1). These are {1, 0} and {1, 2}. We already considered the edge {1, 0}, since the edge {1, 0} and {0, 1} are same. So, we have to consider the edge {1, 2} only. Here see the parent's value of vertex 1, it is 0. On the other hand, the parent's value of vertex 2 is -1. Since we have found different parent values (0 and -1), so we add this edge making the vertex 1 as a parent of vertex 2 (as shown in Figure). At this stage the values of parents are {-1, 0, 1, 0, 0 (Stage-5)}. Next, we can see the edge {2, 3}, that means the vertices 2 and 3. we find parents of the vertices. If the parent is greater or equals to 0, we search the parent values of the vertices 2 and 3 until we reach the root or 0. The parent of vertex 2 is **1** and the parent of 1 is **0**. On the other hand, the parent of the vertex 3 is **0**. We found same parent (final parent) or same root. So, we cannot select this edge {2, 3} as tree edge. In another word *the edge {2, 3} creates a cycle*. Similarly, we can see that the

edge {3, 4} also creates a cycle, since the parent of the vertex 3 is 0 and the parent of the vertex 4 is also 0.

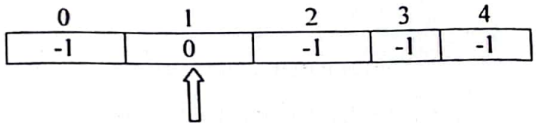
Initially, the array contains -1 in all cells. The initial stage of the parent array for the Figure 8.21 as follows.

Stage-1:



After considering the edge {0, 1} we get the following output. Assign 0 to parent [1].

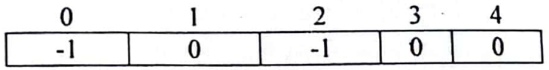
Stage-2:



Put 0 to parent [1].

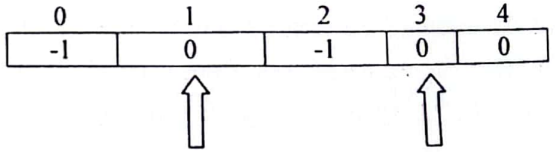
After considering the edges {0, 3} and {0, 4} the array will be as follows. Assign 0 to parent [3] and to parent [4].

Stage-3:



Next, we consider the edge {1, 2}

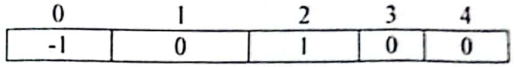
Stage-4:



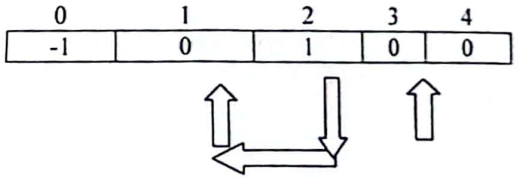
Parent [1] is 0 and parent [2] is -1, not same parent value. So we can select edge {1, 2}.

Next, when we select the edge {1, 2} the array is as follows.

Stage: 5



Let us see the implementation issue. For this two functions will be used; one is **fnd ( )** and another is **uni ( )**. The **fnd ( )** will find or return the value of parent of the vertex, u until we get the root or zero. Using the **uni ( )**, we select the vertices or the edge if the parents of two vertices u and v are not same. For this, we take a parent array (array for tracking parent) which contains the value of parent with respect to the vertex equals to the index of the array. Let us see the consideration of the edge {2, 3}, the parent of 2 is 1 and the parent of 1 is 0. Again the parent of 3 is 0. We got same parent (final parent) value for the two vertices 2 and 3. So, we cannot select this edge or **this edge creates a cycle.** If both the parent values are -1 and -1, we will consider this case as the same parent.



For vertex 2:
i) parent [2] is 1
ii) parent [1] is 0

For vertex 3 (index):
i) parent [3] is 0

For vertius 2 and vertius 3 we got same final parent value. so this edge creates a cycle. We find the value of parent using **fnd ( )** and we select the edge using **uni ( ).** The code of the two functions is as follows.

```
int fnd (int i)
     {
    while (parent[i] > = 0)
     {
     i = parent[i];
       }
    return i;                    / / return the final parent at the root.
     }
int uni (int i, int j)
{
if (i != j)
{
parent [j] = i;
return 1;                    / / if not the same parent return 1
}
return 0;
}
```

Let us see pseudo-code to find the edge that creates a cycle in a graph. We will consider the graph of Figure 8.21.

Code to find the edge that creates cycle:

```
int parent[5] = {-1, -1, -1, -1, -1}
int m[5][5] = { {0, 1, 0, 1, 1},
                {1, 0, 1, 0, 0},
                {0, 1, 0, 1, 0},
                {1, 0, 1, 0,1},
                {1, 0, 0, 1, 0} }
ver1[10];
ver2[10];
k=0;
for (i = 0; i <= 5; ++i)
{
for (j = 0; j <= 5; ++j)
  if (m[i][j] != 0)  // if there is an edge or non-zero value in matrix
    {
     if (m[i][j] == m[j][i] && i < j)        / / take (1, 2) not (2, 1)
      {
     ver1[k]=j;    //save all j in an array
```



```
     ver2[k]=i;   //save all i in another array
     k++;
      }//end of if
  } //end of if
} //end of for
for(i=0;i<k;i++)
// First vertex is in ver1[ ]and second vertex is in ver2[ ]

{
u = fnd(ver1[i]);
v = fnd(ver2[i]);
if (uni(u, v))  //when union is okay
{
cout << tree edge<<ver1[i]<<ver2[i];
}
else {
cout << ver1[i] << ver2[i] << "creates cycle";
}                    //end of if
}                   // end of if
}                  // end of for
```

Kruskal algorithm using find and union procedures is given below.

Algorithm 8.6: Kruskal's algorithm with respect to Find and Union

1. Input a weighted graph, G (take cost matrix of the graph).
2. Create an empty list E and store all non-zero weights (values) of the matrix to E.
3. Sort E in non-increasing order of edge values (weights)
4. For (i=0; i<k; i++) //k is the number of non-zero values

```
  {
 Take an edge (u, v ) from the sorted list.
 fnd (u); //use fnd function described earlier
  fnd (v);
if (uni (u, v)) select the edge (u, v); //if the edge does not creates cycle,
                                        //select it
} //end of for
```

5. Output edge list of a MST (minimum spanning tree).

Implementation issue of Kruskal's algorithm:

Let us see the graph of Figure 8.19 .

1. We can take the data of the graph in a matrix as follows.

```
int m[8][8]= {
                {0,3,0,5,2,0,0,0},
                    {3,0,7,0,8,0,0,0},
                    {0,7,0,0,10,2,0,0},
                    {5,0,0,0,3,0,2,0},
                    {2,8,10,3,0,7,5,5},
                    {0,0,2,0,7,0,0,6},
                        {0,0,0,2,5,0,0,4},
                    {0,0,0,0,5,6,4,0}
            };
```

2. Now create a list of structures as follows:

```
    struct ES
            {
                int u;    //vertex u
                int v;    //vertex v
                int val;  //edge value
            }EL[40];
```

3. Save all the non-zero values of the matrix, m to the list, EL. We will store row value of matrix to u, column value to v and edge value to val of the list EL.

```
    k=0;
        for(i=0;i<8;++i)
          {
              for(j=0;j<8;++j)
               if(m[i][j]!=0)
               {
if( m[i][j]==m[j][i]&& i<j) //take edge (1, 2) not (2, 1) for equal values
        {
        EL[k].u=i;   //store vertex u, v and edge value in the structure (ES)
        EL[k].v=j;
        EL[k].val=m[i][j];
        ++k;
         }
          }
          }
```

4. Sort the EL list in no-increasing (descending order) order.

Let us see how we can sort a list, which is a list of structures. Here we are using insertion sorting algorithm. (Detail about insertion sort is given in chapter 10). Here we have considered that there are k non-zero values.


```
    ES temp;
            for(j=1;j<=k;++j)
    {
        temp=EL[j];
        i=j-1;
       while(i>=0 && EL[i].val>temp.val)
        {
            EL[i+1]=EL[i];
            --i;
        }
       EL[i+1]=temp;
   } //end of sorting
```

5. Next, we have taken to select edges using fnd () and uni().

```
   cout<<"Selected edges for tree are:";
            cout<<endl;
            for(i=0;i<k;++i)
            {
                u=fnd(vs[i].u);
                v=fnd(vs[i].v);
                if(uni(u,v))
                {

    cout<<"("<<EL[i].u<<","<<EL[i].v<<")"<<"="<<EL[i].val;
                cout<<endl;
                }
            }
```

We can take any undirected and weighted graph and implement the above code with necessary modifications and test the code.

8.5.1 Single source shortest paths problem

It is a simple traversal method that finds 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 which are adjacent to v_0 and find out that vertex whose distance is minimum from v_0. We repeat this process until we traverse all vertices in the graph and get the traversing cost *i.e.* the minimum distance.

The case that should be considered here is, if u and w are two adjacent vertices of v_0 and traversing cost from v_0 to the vertices are x and y

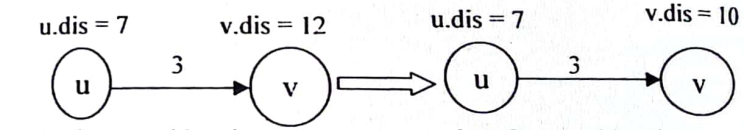
respectively and traversing the distance from w to u is z where $x > y$ and $x - y + z$. Then we have to traverse w from v_0 at first and then we will traverse u from v_0 via w so that we can find the minimum distance. This case must be considered for traversing all vertices.

Let us see the updating process of shortest distance to a vertex. The process is as follows.

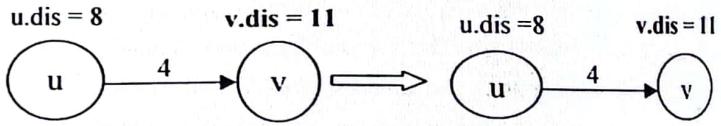
8.5.2 Consideration of an edge and updating shortest path distance

Let *v.dis* estimates the shortest distance from source vertex, s to vertex v (directly from s or through one or more vertices). We can update the shortest distance by considering an edge value. We will verify whether we can update the shortest distance from s to v found so far by going through vertex u. It means we will consider the value of edge (u, v) here.

Now we see the example given below.



a) Before consideration, b) After consideration



c) Before consideration d) After consideration

Figure 8.22. Consideration of an edge value in shortest path distance

In Figure 8.22(a) a, u.dis = 7, v.dis = 12 and the edge value = 3. Here, 12 > 7 + 3 or 12 > 10, so we update v.dis = 10 (new value) in Figure 8.22 (b).

In Figure 8.22(c), u.dis = 8, v.dis = 11 and the edge value = 4. We get, 11 < 8 +4 or 11 < 12, so we keep v.dis = 11 (previous value) in Figure 8.22(d).

Procedure of consideration of an edge:

```
if (v.dis > u.dis + m [u, v]
  {
  v.dis = u.dis + m[u, v]; //update the v.dis
v. pred = u; // u is the predecessor vertex of the vertex, v.
}
```

Algorithm 8.7: 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.

8.5.3 Implementation issue of shortest path algorithm

The pseudo-code of algorithm 8.7 is given below.

1. Take a structure:

```
struct ds
{
        int u;     //vertex u
        int v;     //vertex v
        int dis;   //distance
        int pred;  //predecessor
};
```

2.Take two array of structures. One is for distance of each vertex from source. Another is for priority queue used to find minimum value and respective index:

```
ds da[5],qa[5];
```

3. Take matrix for the graph:

```
  m=99;
   int mx[5][5]={
            {0,10,5,m,m},
             {m,0,2,1,m},
```



```
             {m,3,0,9,2},
             {m,m,m,0,4},
             {7,m,m,6,0},
      };

   int i,j,k,v;
```

4. Store data from first row (source row) to array for distance (da)

```
   da[0].dis=0;
   da[0].u=0;  //Here we consider source vertex is zero
   da[0].v=0;
   da[0].pred=0;
```

5. Initialize all the verices: distance as 99, source is zero, predecessor is also zero so far.

```
  for(j=1;j<ver;++j)
    {

      da[j].dis=m;da[j].u=0;da[j].v=j;da[j].pred=0;

    }
```

6. Take initial data from da[ ] to qa [] (qa used to find out minimum)

```
for(j=0;j<ver;++j)
        qa[j]=da[j];

int n;int u;
 n=ver-1;

while(n>0)
{
```

i) Take first from queue as vertex u:

```
u=qa[0].u;
```

ii) Consideration of edges and update distance. Save the value in da [ ]

```
for(v=1;v<ver;++v)
 {
        if(mx[u][v]!=0)
```

```
        }  if(da[v].dis>da[u].dis + mx[u][v])
  {da[v].dis=da[u].dis+mx[u][v];da[v].pred=u;
  qa[v]=da[v];}  //save the value in qa[ ]

  }

} //end of for
```

7. Now we find minimum value from qa[ ]:

```
int ver = 5;  //number of vertices is 5
int min, ind;
min=qa[1].dis;
for(i=1;i<ver;i++)
{
        if(qa[i].dis<=min) {min=qa[i].dis;ind=i;}
}
--n;

qa[ind].dis=m; //mark it as done by putting 99
qa[0].u=qa[ind].v; //vertex that have minimum distance will be the first
                //vertex of queu:
cout<<endl;
}//end of while n>0;
cout<<endl;
//print vetices u(source), v(destination) and distance
cout<<endl;
for(k=0;k<ver;k++)
{
        cout<<"  "<< da[k].u; //vertex will be as 0, 1, 2, 3 and so on
        cout<<"  "<<da[k].v;
        cout<<"  "<<da[k].dis;
        cout<<endl;
}

cout<<endl;
```

8. Using da[ ].pred and a stack we can print the path as follow. Stack helps us to print the results in reverse order.

```
        cout<<endl;
        int stack[7];
```



```
        int top;
        for(i=0;i<ver;++i)   //print the path for each vertex
        {
                cout<<da[i].u;          //source
                cout<<" "<<da[i].v<<": "; //destination
          k=i;
          top=-1;
        while(da[k].pred!=0)
          {
          ++top;
  stack[top]=da[k].pred;
          k=da[k].pred;
          }
          while(top>=0)
          {
                  cout<<" "<<stack[top];
                  --top;
  }
                cout<<" "<<da[i].v;  //print whole path
        cout<<endl;
        }
```

For simulation we will consider the graph of shown in Figure 8.23

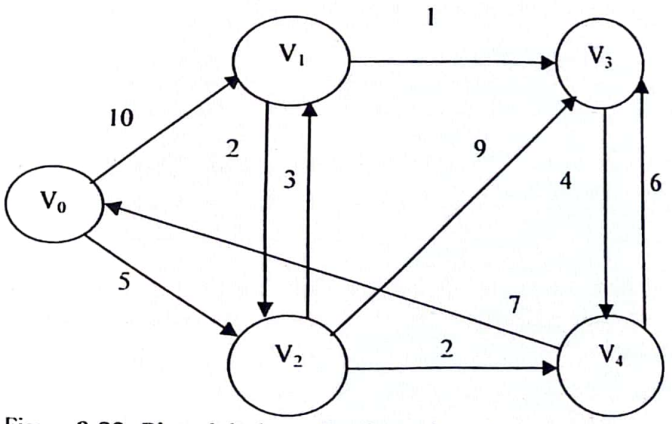


Figure 8-23: Pictorial view of a directed graph.

Let us see how the above pseudo-code of algorithm 8.6 will work. For this, we can see the matrix of the graph again. The matrix of the graph is as follows:

	0	1	2	3	4
0	0	10	5	m	m
1	m	0	2	1	m
2	m	3	0	9	2
3	m	m	m	0	4
4	7	m	m	6	0

In the matrix m = 99;

We initialize da[ ].dis as follows:

0	m	m	m	m

1) Consider row number 0 (1st row) of the matrix mx and update da[ ].dis, Here vertices 1 and 2 minimize the previous values, so we update da[ ].dis as follows.

0	1	2	3	4
0	10	5	m	m

2) Take minimum value from above da [].dis, to find out minimum we have used an array, qa in the code. Minimum is 5 and value is index 2 of the array. Now go to row number 2 of the matrix mx and update the values of da[ ].dis. We have put a value, which is minimum previous or new. Such as, in da[1].dis we have put 8, since 5 + 3 <10 or 8 < 10.

0	1	2	3	4
0	8	5	14	7

3) The next minimum is 7 and index is 4. Now consider the row 4 of the matrix, mx and update da[ ].dis

0	8	5	13	7

4) Next minimum is 8 and it is in index 1. Consider the row 1 of the matrix and update the values. Below is the shortest distance between each vertex from source vertex 0.

0	8	5	9	7

Now we will explain how we get the shortest paths from the source. We can get this from the array da[ ].pred. The predecessor of each destination is saved in this array.

1) Initially, the da[ ].pred is as follow. The predecessor of all vertices is source or 0.

0	0	0	0	0

2) The predecessors of vertices 1, 2, 3, 4 are 2, 0, 2, 2. (index of the array indicates vertex and the content of the array indicates predecessor (it is also vertex).

0	1	2	3	4
0	2	0	2	2

3) The predecessors of vertices 1, 2, 3, 4 are 2, 0, 4, 2.

0	1	2	3	4
0	2	0	4	2

4) The predecessors of vertices 1, 2, 3, 4 are 2, 0, 1, 2.

0	1	2	3	4
0	2	0	1	2

We can find the shortest path from above final results in the array. Let us see the path one by one. From the array, we find the predecessor until we get 0 and have to print the path in reverse order.

1) Source is 0 its predecessor is 0 (as taken)

2) The predecessor of vertex 1 is **2**, the predecessor of 2 is **0**. So the path is 0, 2, 1.

3) The predecessor of vertex 2 is **0**. So the path is 0, 2.

4) The predecessor of vertex 3 is **1**, the predecessor of 1 is **2**, the predecessor of 2 if **0**. So, the path is 0, 2, 1, 5) Predecessor of vertex 4 is 2, the predecessor of 2 is 0. So, the path is 0, 2, 4.

We have found out the minimum values using a priority queue, which is done in step 7 of the pseudo-code. We can perform this using heap of a structure also. For a larger graph, we can get better performance if we use the heap.

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 the total minimum cost is called *minimum cost spanning* tree. To build minimum cost spanning tree, the graph must be a *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 to 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 a cycle. That means all trees are graph but all graphs are not trees.

Questions:

1. Define graph with an example.
2. How can a graph 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 can a graph be traversed? Discuss in details about breadth first traversal techniques.
6. Define adjacent list and an adjacent matrix of a graph.
7. Define a graph. The daily flights of an airline company appear in the figure Q-8.1. 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.
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 an example.
10. Describe the stages of Prim's algorithm with an example.
11. Describe the stages of Kruskal's algorithm with an example.

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

Figure Q-8.1: Data for a graph

Problems for Practical (Lab) Class

Graph related problems

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

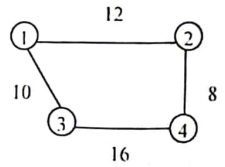


Fig. 8-1p

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

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

ii) Print the adjacent list of vertices with edge costs of each vertex as shown below.

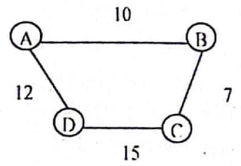


Fig. 8-2p

Adjacent of **A:** B 10, D 12;
Adjacent of **B:** A 10, C 7;
Adjacent of **C:** B 7, D 15;
Adjacent of **D:** C 15, A 12.

Problem 8-3: Given a graph in Fig. 8-3p and create linked adjacency lists with edge cost of each edge and **print** the data of the linked list (see algorithm 8-1).

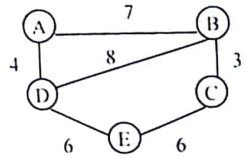


Fig. 8-3p

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

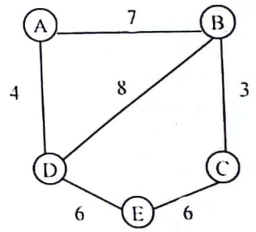


Fig. 8-4p

Problem 8-5: Write a program to access (print) all the vertices in BFS method (use the graph in Figure 8-10).

Problem 8-6: Write a program to access (print) all the vertices in DFS method (use the graph of Figure 8-11).

Problem 8-7: Write a program to create a MST using Kruskal algorithm (use the graph in Figure 8-19.

Problem 8-8: Write a program to find out the shortest paths for the graph in Figure 8-23.

CHAPTER NINE
SEARCHING AND SORTING

OBJECTIVES:

- ☐ **Searching:**
  - ➢ Describe linear searching
  - ➢ Write an algorithm for 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) for 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, the complexity of each algorithm has been shown. To identify or locate an element or position of the element from a list of elements is called *searching*. Here we have considered 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 there is 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 the 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 there is 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.

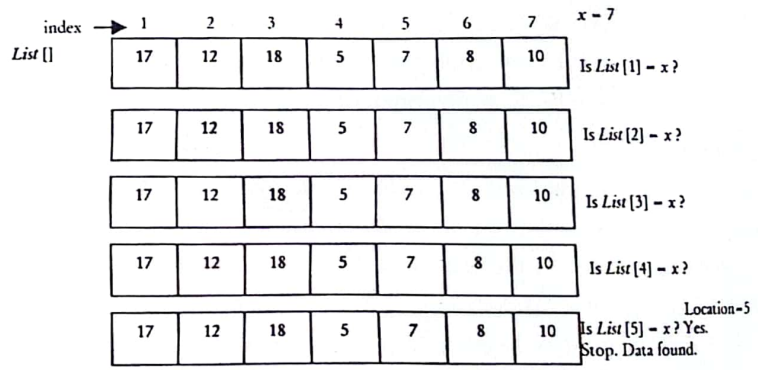


Figure-9.1: Pictorial view of linear searching (shaded items showed the searching sequence)

Algorithm 9.1: Algorithm for linear searching

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

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

2. Search the list to find the target element

```
for (i = 0; i < n; i ++)
        {
        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

Number of comparisons if the target element is in the first position = 1.

Number of comparisons if the target element is in the second position = 2.

.

.

.

Number of comparisons if the target element is in the nth position = n.

So, total comparisons = 1 + 2+ 3+ . . .+ n.

In the average case, time complexity

$$= \frac{1+2+3+\ldots\ldots\ldots\ldots+n}{n}$$

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

$$= \frac{1}{2}n + \frac{1}{2} < n \text{ for } n \geq 1$$

$$= O(n)$$

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 there is 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 the 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 searches on the left half of the list in the same manner which was done for the whole list. If the target element is greater than the middle element we do searches on the right half of the 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 the 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 symbolic notations as follows:

Let the list of the elements are as follow:

$a_1, a_2, a_3, a_4, \ldots \ldots \ldots \ldots \ldots, a_m$; 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 $A[0 \ldots m-1]$, x; //A is an array with size m and x is the target element
2. $first = 0$, $last = m-1$;
3. while ($first \leq last$)

```
{
mid = (first + last) / 2;
(i) if (x = =A[mid]), then print mid;  //target element =
  break (stop searching);                //A[mid] or target
                                         //element is in 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.

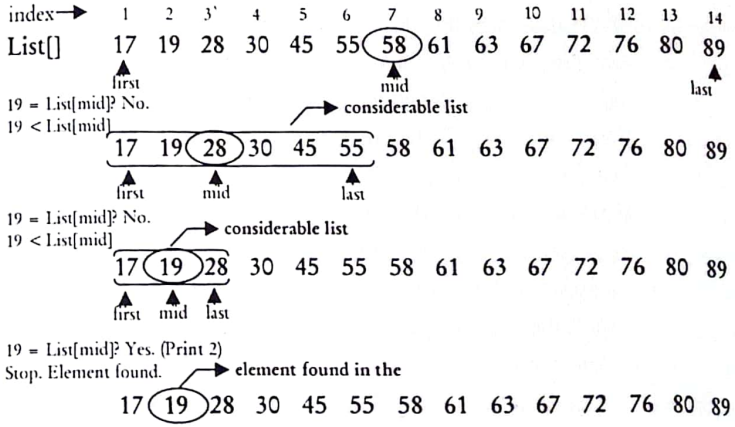


Figure-9.2: Pictorial view of binary searching

9.1.2.2 Complexity of binary search

Suppose that there are n elements in the list.

Let $n \leq 2^k$; then $k \leq log_2 n$.

To find any element maximum number of comparisons will be k.

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 such as:

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 the first position and the data in the first position is placed in the position where the smallest data was. After that, we consider the list except for 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 the first position and the data of the first position in the position of the smallest data.
ii. Repeat the process for the list except for data in the first position, and so on.

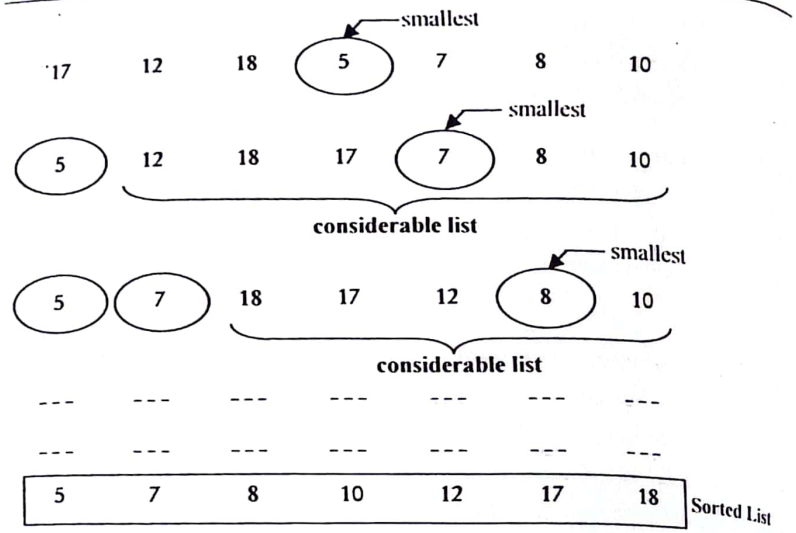


Figure-9.3: Pictorial view of selection sort

Algorithm 9.3: Algorithm for selection sort

1. Input Array $A[0\ldots\ldots n-1]$
2. (i). for (i = 0; i<n; i++)

```
    {
    small_index = i;
(ii). for (j = i + 1; j < n; j++)
        {
        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 the first phase, number of comparisons is n-1; the second phase, number of comparisons is n-2; the third phase, number of comparisons is n-3 and so on. Then,

Maximum number of comparisons to find the first smallest number = n-1.

Maximum number of comparisons to find the second smallest number= n-2.

Maximum number of comparisons to find the last smallest number = 1.

Total comparisons = (n-1) + (n-2) + (n-3) + ... + 1

$$= \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n < n^2 \text{ for } n \geq 1.$$

Therefore, time complexity = $O(n^2)$

9.2.5 Insertion sort

In this method, we take the data of the second position first and compare it with the data of the first position. If the data in the second position is smaller than the data in the first position, then we shift the data in the first position to the right (to the second position) and insert the data of the 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 data of the third position is inserted in the second position. By repeating the process for the data in the fourth 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 (those 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.

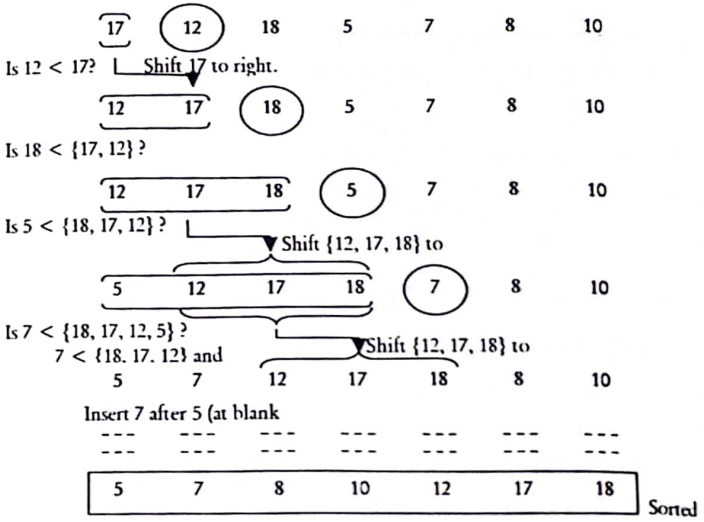


Figure-9.4: Pictorial view of Insertion Sort

According to Figure-9.4, the insertion sort algorithm rearranges 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 placed in the 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[0.....*n-1*]
2. (i). for (*j* = 1; j < n; j++)
   {
   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

Maximum number of comparisons for the data in the second position = 1.

Maximum number of comparisons for the data in the third position = 2.

.

.

.

Maximum number of comparisons for the data in the last position = n-1.

Total comparisons = 1 + 2 + 3 + ... + (n-1)

$$= \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \leq n^2 \text{ for } n \geq 1$$

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 the desired order. Then we compare the second data element with the third data element and arrange them in the 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 the 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 the 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 a 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 a desired order so that *A[1] < A[2]*.

After n-1 phases, the list will be sorted in increasing order.

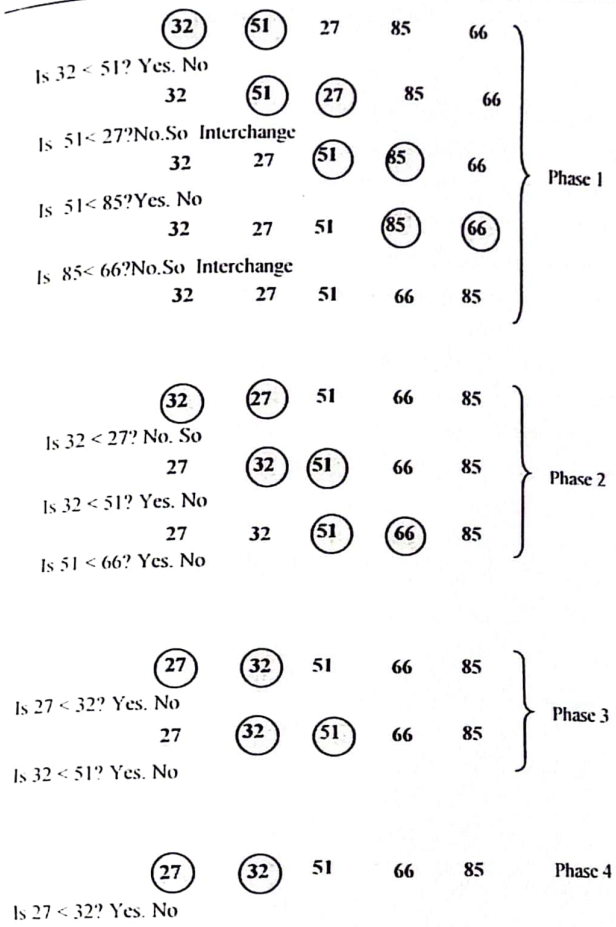


Figure-9.5: Pictorial view of bubble sort

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: D [0....n-1]
2. for (k=0; k<n; k++)
{
        s =1;
        while (s ≤ n-k)
        {
            if (D[s] > D[s+1])
                Interchange (D[s] and D[s+1]);
            s = s+1;
        }
}
3. Output: sorted list
```


9.2.6.1 Complexity of bubble sort

No. of comparisons = $(n-1) + (n-2) + \ldots + 2 + 1$

$$= \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n < n^2 \text{ for } n \geq 1$$

$$= O(n^2).$$

Therefore, time complexity = $O(n^2)$.

9.2.7 Merge Sort

Merge sort is a divide and conquer method. It has mainly two phases. One is *dividing phase* and another is *merging phase*. In dividing phase at first it divides the whole list into two sub-lists (parts). Next, it divides the first part into two parts. The leftmost part of the new two parts will be divided into two parts again. The division of the leftmost part will be completed when each part contains only one or a single item. Then the method starts merging of two single items by comparing them. In case of ascending order, the smaller value is put into the first position and other into the next position. As a result, we get a sorted part of two items. Next, the method divides the next part into two single items and merges them by comparing. Now there are two merged parts. These two parts will be merged again by comparing items of each part. The two parts give a longer sorted part. Thus the method divides each part and merges them. Lastly, we get two sorted parts, which are the results of merging. These two parts will be merged again by comparing the items. Finally, we get a sorted list in the array. The pictorial view of the method is given below. When merging is performed we have shown the process using arrows.

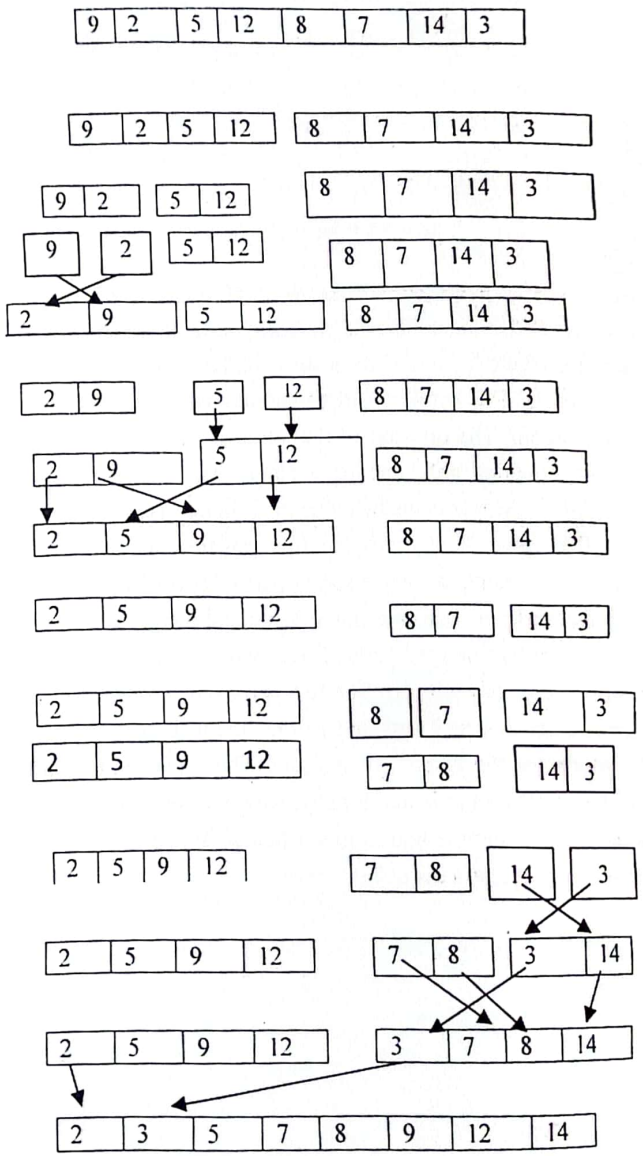


Figure 9.6: Pictorial view of the merge sort

Let us see the process at a glance.

1. Divide the list of data into two sub-lists (parts).
2. Divide left part again (recursively) and the right part will remain as it is.
3. Divide left sub-part again. Repeat the process until we get single element in a single part.
4. If there is single data item in a part, merge two parts of single items.
5. Divide right sub-part and remain the merged left part as it is.
6. If there is a single item in a part, merge right sub-part.
7. Merge left sub-part and right sub-part.
8. Similarly, division and merging will be performed with the data of the right part.

Now we explain how two sorted parts can be merged. Let A is an array that holds data and we need another array T to perform merging. We set indicators (variable for index) at the beginning of the two parts such two indicators are i and k. Another indicator h is used for the array T. The data of the indices i and k will be compared and the smaller data will be put at the first position of the array T and the value of the corresponding indicator will be increased. In any case, we also increase the value of the indicator, h used for array T. The process of comparison and copying the data to the array, T will be continued until we put all the data in the array T. The pseudo-code of the main part of the merging process is shown below.

```
h = 0;
if (A[i] < A[k]) { T[h] = A[i]; i = i + 1;}
else { T[h] = A[k]; k= k + 1;}
h = h + 1;
```

The pictorial view of the merging process is depicted in the figure 9.7.

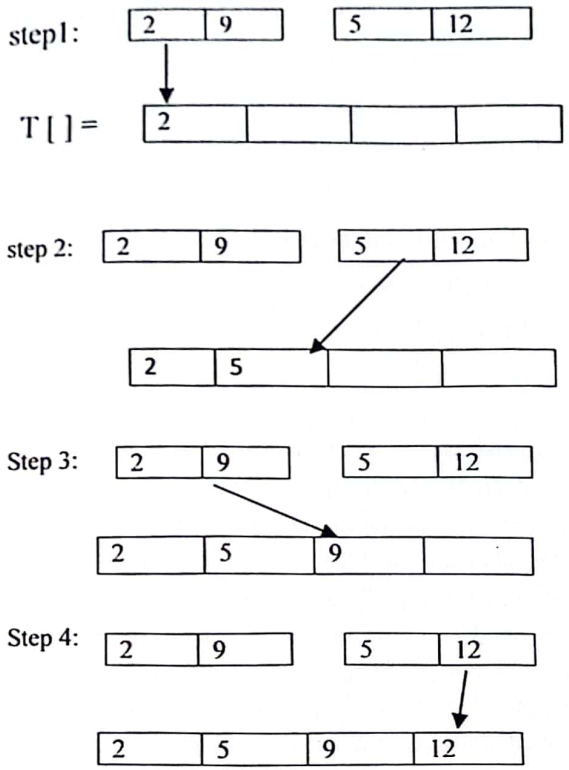


Figure 9.7: Pictorial view of the merging process.

The algorithm (pseudo-code) for merge sort is given in Algorithm 9.6. The algorithm is recursive in nature. It calls itself twice, once with left sub-list and next with right sub-list. After completing division it calls .merging procedure to merge them.

Algorithm 9.6: Algorithm for merge sort.

```
Merge-sort(A[], f, l)
{
if (f < l)
        {
        m = (f + l)/2;
    merge-sort(a, f, m);   //recursive call
    merge-sort(a, m+1, l); //recursive call
    merge(a, f, m, l);     //call merging procedure
      } //end of if
}

merge( A[], f, m, l)
{
        i =f; k=m+1;
        h=f;
        while(i<=m && k<=l)
        {
        if(A[i]<=A[k])
    {T[h]=A[i];++i;} //copy from left half
      else {
    T[h] = A[k]; ++k;}  //copy from right half
       ++h;    //increase the index value of array T
      }
      if(i > m)   // If there is no element in the left half, copy all the elements
      {           //from right half
      for(b = k; b<=l; ++b)
      {T[h]=A [b]; ++h;}
      }
      else        // If there is no element in the right half, copy all the elements
      {           //from left half
      for(c = i; c <= m; ++c)
      {T[h] = A[c]; ++h;}
      }

      for(i = f; i<=l; i++)
      {
      A[i]=T[i];
            }
}
```

To implement the above algorithm we have to call merge-sort function after declaring and assigning data to the array as *merge-sort* (A, 0, 9) for an array with 10 elements. If we implement the Algorithm 9.6 using the data set given in Figure 9.6, we can create a tree shown in Figure 9.8. For each call, we can create a node with the values of indices (parameters) and construct the tree for all calls. The data inside a node indicate index values and number beside the nodes are sequences of nodes that can be produced after each call. Sequence beside the nodes such as 1, 2, 3 ..., 15 correspond to calls of merge sort function. Each call uses index values as parameters. Such as, for the first call we use indices (second and third parameters) 0 and 7. For second recursive call, it uses 0 and 3 and so on.

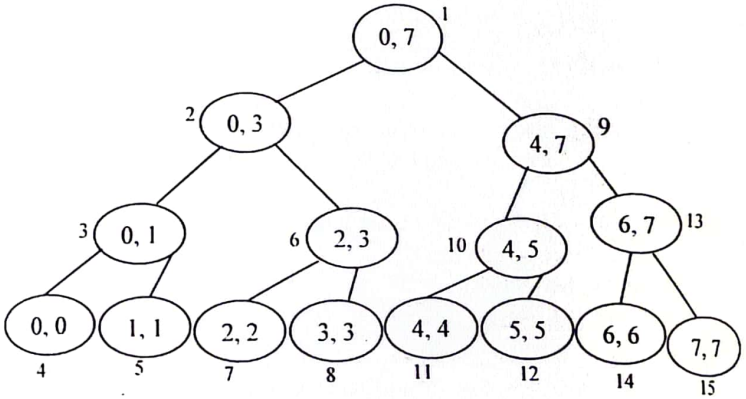


Figure 9.8: Tree for division process of merge sort procedure

In this tree, each node represents each of the sub-lists processed in merge sort. Now we write how the function goes to all of the nodes. For this, the right link of each node will be stored in a stack (recursive function uses stack). After completing work with the left sub tree it backs to the right sub-tree. Let us see how data of the stack are pushed and popped. We denote R(i) as a right link (pointer to the right child) of node i such as R(1) denote

right link of node 1. The process will go the left until it finds null. At first, the R(1) is stored on the stack. It goes the left and saves the right link of the present node into the stack. So, now R(2) is pushed at the top of the stack. When it finds null, it popped data from the stack. So, the control will be transferred to the right of each of sub-lists (parts) using the stack.

In merging process the control goes to the leaf nodes to the root. Here in Figure 9.9, we have depicted how the data will be merged after division. Indices of the data are shown inside the node and the sequence of processing are given as serial numbers 1, 2, 3 etc, which correspond to the merging of sub-lists. Such as nodes 1 and 2ares merged to produce node 3, next nodes 4 and 5 are merged to produce node 6 and so on. Finally nodes 7 and 14 are merged to produce the root node.

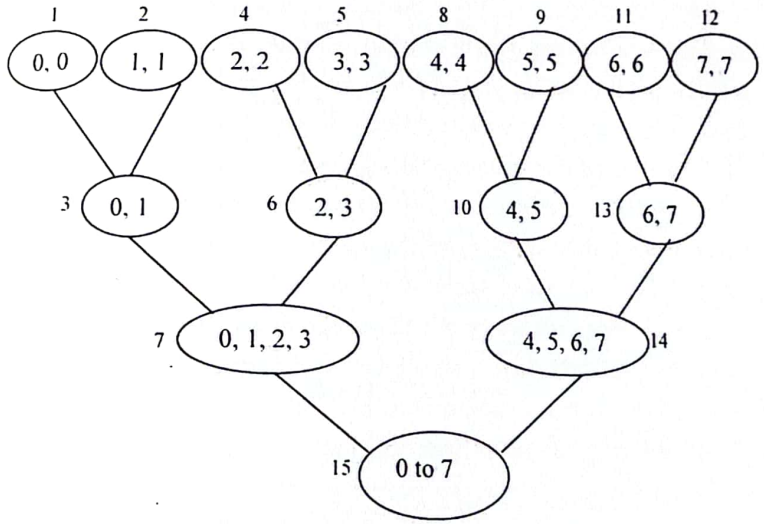


Figure 9.9: Tree for process sequence of merging procedure

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

$$T(n) = \begin{cases} 1 & n = 1 \\ 2\,T(n/2) + n & n > 1 \end{cases}$$

When n is a power of 2, $n = 2^k$; we can solve this equation by successive substitutions.

For $n > 1$ we can write

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &= 2(2T\left(\frac{n}{4}\right) + \frac{n}{2}) + n \\ &= 4T(\frac{n}{4}) + 2n \\ &= 2^2 T(\frac{n}{2^2}) + 2n \\ &= 4(2T\left(\frac{n}{8}\right) + \frac{n}{4}) + 2n \\ &= 8T\left(\frac{n}{8}\right) + n + 2n \\ &= 8T\left(\frac{n}{8}\right) + 3n \\ &= 2^3 T\left(\frac{n}{2^3}\right) + 3n \\ &= \ldots = 2^k T(1) + kn \end{aligned}$$

$$[2^k T\left(\frac{n}{2^k}\right) = T(1) \text{ and } \quad n = 2^k]$$

$$= an + n\log_2 n$$

$$[T(1) = a, \quad 2^k = n, \quad k = \log_2 n]$$

$$T(n) = O(n \log_2 n).$$

9.2.8 Quick Sort

Quick sort is a divide and conquer method. In this method, one element is to be selected as partitioning element (value). Now we divide the whole list of data into two parts with respect to the partitioning element. The data which are smaller than or equal the partitioning element will remain in the first sub-list and which are greater than the partitioning element will remain in the second sub-list (part). Any greater data of the left sub-list will be transferred to right (second part) and any smaller data of the right sub-list will be transferred to the left (first part) of the list. To transfer data we exchange them using their positions. We have to exchange greater for smaller data. To search data (smaller and greater) two indicators are used. Using one indicator such as i we find greater data and using another indicator, k we find smaller data. The first indicator starts searching from left (beginning) side of the list, on the other hand, second indicator searches from right (end) side of the list. In other words, the first indicator moves to the right and the second indicator moves to the left. If any *greater* data is found, the indicator i points (indicates) it, similarly any *smaller* data will be pointed by k. Now we exchange the data in ith position with the data in kth position. As for example if any data which is greater than the partitioning element is found in 4th position and smaller data is found in 9th position respectively, the data of the 4th and 9th positions will be exchanged. Next, we try to find another two data (greater and smaller), if they are found we exchange them. The partitioning point can be found when the two indicators (i and k) cross each other. Such as, at any moment i may be at the 6th position and k may be in 5th position. It means the indicators cross each other and the 5th position is the partitioning point. The first part ends at 5th position and the second part starts at the 6th position. A similar situation can be seen in Figure 9.10. We apply the same method to both parts (sub-lists). By repeating the method for each of the sub-lists we can get a sorted list.

Now let us explain the process using Figure 9.10. We have selected 40 as a partitioning element (value). Initially i = 1 and k = 12 (end position +1).

Now we increase *i* by 1 and continue until we get a data which is greater than 40 as well as we decrease *k* by 1 and continue until we get a data which is smaller than 40. We have got a greater data (80) by *i* and a smaller data (30) by *k*. Now we exchange 80 with 40. Next, by moving on to the next data, we get 60 and 17. So, data (60) has been exchanged with 17. The two indicators move until i > k (i crosses k). When k points 17 and i points 50 (i= 6 and k=5, see stage 1(e) in Figure 9.10), we exchange 17 with 40 (partitioning element) and the partitioning process completed for the first stage (round). Figure 9.10 shows how we get a sorted list by repeating the partitioning process. Partitioning will stop for any sub-list if it contains only one data. Sorting has been completed in 8 stages. In stage 1(a) to 1(f) partitioning process has been shown in detail. In stages, 2 to 9 only partitions are shown, but data swapping have not been shown.

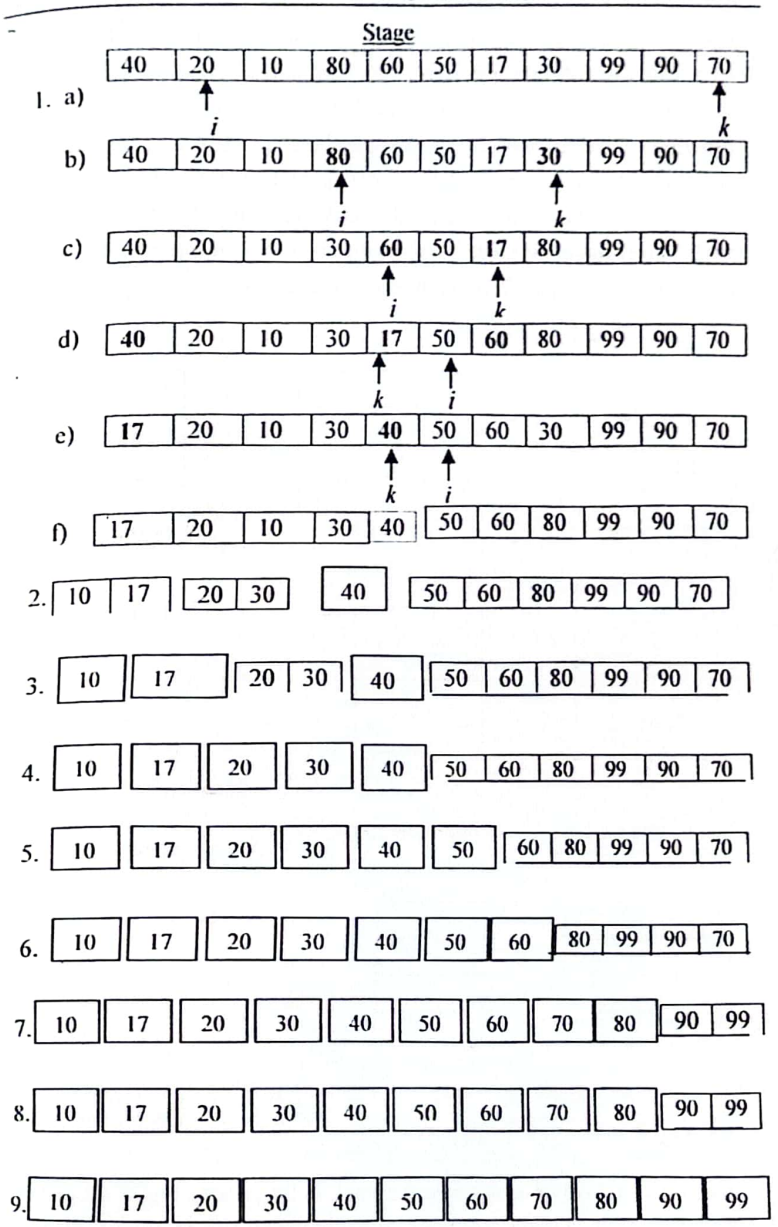
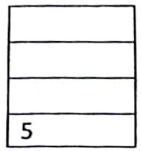
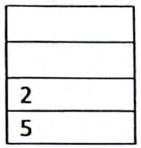


Figure 9.10: Pictorial view of quick sort

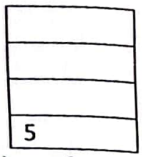
Quick sort is a divide and conquer method. The algorithm can be devised using recursive function. Recursive function stores necessary addresses in the stack to process next sub-problem. Let us see how the algorithm will keep track of each sub-list. In our example there are 11 data in the array and indices of the array are 0 to 10 (for implementation in C/C++). The index values are pushed and popped from the stack are depicted in Figure 9.11 (a) to (g). In this figure (a) corresponds to the stage 1(f) of Figure 9.10 and (b) corresponds to the stage 2, (c) to the stage 3, and so on.



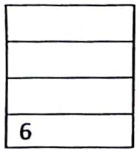
a) push index 5



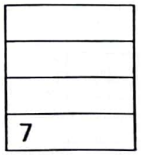
b) push 2



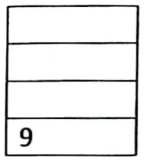
c) pop 2



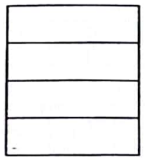
d) pop 5 and push 6



e) pop 6 and push 7



f) pop 7 and push 9



g) pop 9

Figure 9.11: Indices of sub-lists in the stack in the processing of quick sort.

The pseudo-code of quick sort is given in algorithm 9.7. Here A is array, *f* is the first index, *l* is the last index and pv is the partitioning element (value).

Algorithm 9.7: Algorithm for quick sort

```
q-sort(f, l)
{
        int j;
        if (f <l)
        {
        j = makepart(A, f, l+1);
        q-sort(f, j-1);
        q-sort(j+1, l);
        }
}

makepart(A[], first, last)
{
        pv = A[first];
        i = first; k = last;
        do{
            do{
                i++;
              }while(A[i] <= pv);
            do{
                k--;
              }while(A[k] > pv);

    if (i < k) exchange (A[i], A[k] )

      } while(i <= k);

   A[first] = A[k]; A[k] = pv;
       return k;
      }
```

To implement the algorithm in C/C++, we have to declare the array, A globally. For a list of 10 data, we have to call the q-sort function as *q-sort* (0, 9). The makepart function returns the value of k, which is the partitioning point for each sub-list.

9.2.8.1 Analysis of quick sort

Let us consider *T(n)* be time complexity with respect to the number of comparisons in the 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 are required for the 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 another part.

Note that, $T(0) = T(1) = 0$. By putting the value of *i = 1, 2, 3,……, 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

$$nT(n) = n(n+1)+2\{T(0)+T(1)+\ldots+T(n-1)\} \quad \ldots\ldots\ldots\ldots (2)$$

Replacing *n* by $n-1$ in (2) we get

$$(n-1)T(n-1) = n(n-1)+2\{T(0)+T(1)+\ldots+T(n-2)\} \quad \ldots (3)$$

Subtracting (3) from (2) we can write

$$nT(n)-(n-1)T(n-1) = 2n+2T(n-1)$$

$$\Rightarrow nT(n) = (n-1)T(n-1)+2n+2T(n-1)$$

$$\Rightarrow nT(n) = T(n-1)\{n-1+2\}+2n$$

$$\Rightarrow nT(n) = (n+1)T(n-1)+2n$$

$$\Rightarrow \frac{T(n)}{n+1} = \frac{T(n-1)}{n}+\frac{2}{n+1} \quad \text{[dividing by } n\,(n+1)\text{]}$$

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 quick sort algorithm takes $O(n \log_e n)$ time in the 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. Nowadays, 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 require 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 an 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 an example.
7. What is sorting?
8. Explain insertion sorting method with an 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 an 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 an example how data are partitioned in quick sort. Show the complexity of quick sort in the 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 a 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 the 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, then repeat the process for **100000** numbers and more.

CHAPTER TEN
HASHING

OBJECTIVES:

- Identify hashing, hash table, and hash function
- Identify hash collision
- Describe the linear probing method
- Write algorithms for the 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

Hash means to chop or make anything small. In hashing, we perform several tasks such as, we make key value small, which is used as an address (index) to store data items in a table and retrieve them from the table whenever it is required to read (access) them. We do these tasks for efficient searching. The purpose of hashing is to store and retrieve data from a table efficiently. If we can find the address (index) in an efficient way we can perform the task of data storing and retrieving efficiently. So, hashing means a special type of searching by making key value small with the help a function used for storing data in a table and retrieving them from the table. We get small value from the key value using a function (such as an arithmetic function), which is called the *hash function*. The array (table) used to store data items is called *hash table*. The small value, which we get from the key value with the help of hash function, is called *hash address* or hash value or hash index. Examples of key values are the id of a student, id of an employee of an organization, passport number of a citizen, national id number of a dweller, mobile number of a person etc. The data items may be

key values or records related to key values. We can take hash function using symbolic notation as follows:

$y = h(x)$

Where y is the hash value or hash address and h is the hash function and x is a variable such as key value. An example of a hash function is as below.

y = key value mod m

Where m is the size of the table and the size of the table will be estimated as the possible number of data items to be stored in the table (considering the future issue also). Let us consider three digits key values.

If key value = 104 and m = 10.

y = 104 mod 10 = 4, which indicates we have to store the key value 104 to index number 4 of the hash table.

Similarly 108 mod 10 = 8 so, 108 will be stored in the index 8 of the table.

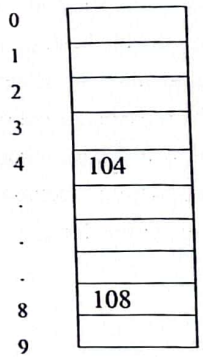


Figure 10.1: A hash table with data

In Figure 10.1 a hash table is shown and the key value (data item) 104 is stored to index number 4 of the table and data item 108 is stored to the index number 8 of the table.

10.2 Hash function

Here division or modular method is used to find the address in the hash table. A function that is used to transform a value (key value) into a direct address (index) of the hash table is called a *hash function*. There are several methods to find out the direct address using a hash function such as division

method, mid-square method, folding method etc. These methods are described below.

10.2.1 Division method

Suppose m is the size of the hash table. In this method **modular** operation is used as follows:

$$h\ (key\_value) = key\_value \bmod m$$

Example: Let m = 10, key values are 103, 108, 109, 204 etc. Then direct addresses are 3, 8, 9, 4 etc.

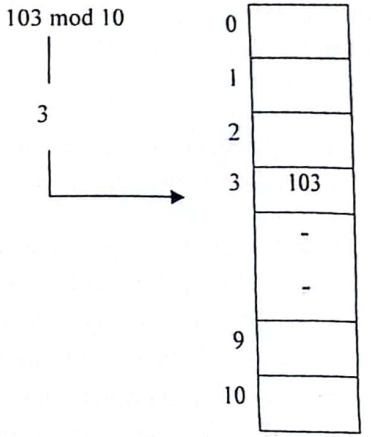


Figure-10.2: Storing key-value in the hash table

10.2.2 Mid-square method

Here we have to take the binary representation of the key value. Then we square this binary number and take the middle k bits. Now, these k bits will be used as a 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 = 5, then middle five bits = 01111 = 15. So, the hash address is 15.

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 them, 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 ($6 \div 3 = 2 \Rightarrow 99 \times 3 = 297$). Using 2-digits we can represent highest number 99.

10.3 Hash collision

The hash collision is a situation, which appears when two data items (records) have to be stored in the same address (index) in a hash table.

As for example, if data item (record) has been already stored in an index number such as index 15 of the hash table, then usually we will not store another data item to the same address (index number 15). However, from the hash function, we may get the same result, *i.e.*15 (as before). It means now we have to store the data item to the index number 15 in the hash table. But the space related to the index number 15 of the hash table is already occupied by another data item. This situation is called hash collision. There are some schemes or methods for resolution or resolving of a 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 way.

Suppose, y_0 is the result of the hash function, it indicates, we have to store the data item in the cell of the hash table whose index is y_0. If y_0 is already occupied, then we try to store the data in the cell whose index is as follows.

$$y_1 = y_0 + 1.$$

If y_1 is also occupied, then we will try to store the data in the cell whose index is $y_2 = y_1 + 1$. In general $y_{i+1} = y_i + 1$, where i = 0, 1, 2, 3, . . ., m-2.

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 the above method we have to use the same hash function and the same method at the time of retrieval of the data. Let us see an example, where the hash function is h(x) = key value mod m, m = 100 and key value = 205. Now, $y_0 = 205$ *mod* $100 = 5$, if cell number 5 is occupied then we find $y_1 = 5 + 1 = 6$, that means we shall try to store data in cell number 6. The pictorial view of this situation (hash collision) is shown in Figure 10.3.

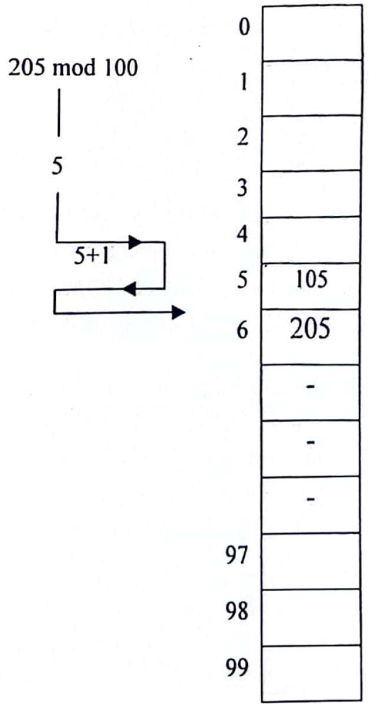


Figure-10.3: Hash collision and resolution with linear probing

Algorithm 10.1: Algorithm to create a hash table with data and resolve hash collision using the linear probing method

1. Input list of elements (*key_values*);

2. Create an empty hash table, T [0.....*m-1*]
3. for (*i* = 0; i < m; i++)
{
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 items (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 the hash table. Here we resolve the hash collision using the linear probing method.

Algorithm 10.2: Algorithm to retrieve a data element from a hash table

1. Input the hash table, T [0.....*m-1*], *key-value* to be retrieved;
2. *h* = *key-value* mod *m*;
3. if (T[*h*] = *key-value*) *item* = T[*h*];
4. else {
do {
h = *h* + 1;
} while (T[*h*] != *key-value*)
5. *item* = T[*h*];
}
6. Output: the target value in the *item*

Comments: Here we assume that we have stored data using a hash function, ***key-value* mod *m*** and resolved the collision using the 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 the target cell of the table is already occupied, then we use the following function as probing function to store the item.

$y_i = (y_0 + k^2)$ mod m; where k = 1, 2, 3 and so on.

And, y_i = *key-value mod 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 $y_1 = 2, y_2 = 5, y_3 = 10$ etc.

10.3.3 Random probing method

Here the following function can be used for probing function:

$y_{i+1} = (y_i + r)$ mod 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 (gcd) 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.

Algorithms for quadratic and random probing methods are similar to the algorithm of the linear probing method.

10.3.4 Double hashing method

In this method, at first, we find a hash address using a hash function. If the address is occupied by any other data item, the second hash function is used to get another hash value. Using the first and the second hash values we find another hash address in the hash table. The process is as follows.

h_0 = key value mod m;

If h_0 is occupied, we use the second hash function as,

r = key value mod p;

Where, p < m and the next address is calculated as below.

$h_{i+1} = (h_i + r) * k$ (1)

Where k = 1, 2, 3, .., m-1. and i = 0, 1, 2, . . ., m-2.

We use equation (1) until we get an empty address in the table.

Example: Suppose we have key values 211, 232 and 624. Where m = 7 and p = 5.

i) For the first key value:

$h_0 = 211 \bmod 7 = 1$ (empty).

ii) For the second key value:

$h_0 = 232 \bmod 7 = 1$ is not empty.

$r = 232 \bmod 5 = 2$

$h_1 = (h_0 + r)*k = (1+2)*1 = 3$ (empty).

iii) For the third key value:

$h_0 = 624 \bmod 7 = 1$ is not empty.

$r = 624 \bmod 5 = 4$

$h_1 = (h_0 + r)*k = (1+4)*1 = 5$ (empty).

So, we store the key values 211, 232 and 624 in the indices 1, 3 and 4 respectively. The pictorial view is depicted in Figure 10. 4.

0	
1	211
2	
3	232
4	
5	624
6	

Figure 10-4: Hash Table

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 [0.. m-1];
2. for (i = 0; i < m; i++)
   {
   k = 1;
3. ind = key % m;
4. if (table [ind] == 0) table[ind] = key;
5. else
   {
   r = (key % p);
   do
   {
   ind = ((ind + r)*k);
   k = k + 1;
   if (ind = = m) ind = 0;
   } 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 a collision has been resolved using double hashing

```
1. Input the hash table with data, , T [0.....m-1], m, p.
   (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);
7.      do
        {
        h = ((h + r)*k)
        k = k + 1
        if (h = = m) h =0;
        } while (T[h] != key-value);
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 the following process to resolve the collision.

If $y_0 = f$ (*key-value*) is occupied, then we find $y_1 = h(y_0)$, $y_2 = h(y_1)$ and so on. Here we repeatedly use the second hash function.

Suppose, y_0 = key_value mod m and h = (y_0 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 = 621 mod 10 = 1 is free.

ii. For the second value,

y_0 = 901 mod 10 = 1 is occupied.

y_1 = (1+3) mod 10 = 4 is free.

iii) For the third key value,

y_0 = 811 mod 10 = 1 is occupied.

y_1 = (1 + 3) mod 10 = 4 is occupied.

y_2 = (4 + 3) mod 10 = 7 is free.

Algorithm 10.5: An algorithm to create a hash table with data and resolve hash collision using rehashing method.

1. Input key values, m and r.
2. Create a table T [0 . . . m-1].
3. for (i = 0; i < m; i++)
   {
   ind = key mod m;
   if (table[ind] == 0), table [ind] = key;
4. else
   {
   do
   {
   ind = (ind + 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 a collision has been resolved using rehashing method.

1. Input the hash table, T [0.....m-1], *key_value* to be retrieved (the table that was created using the algorithm 10.5)
2. input a key value
   {
   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 the 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-10.5 and Figure-10.6. In the hash table, we have considered the key value of three digits.

Suppose that we have a hash function, *key_value mod* 100. Let *key_value* = 102. Then h_0 = 102 mod 100 = 02, so we store 102 in a node that will be linked by the pointer of index 02 of the hash table.

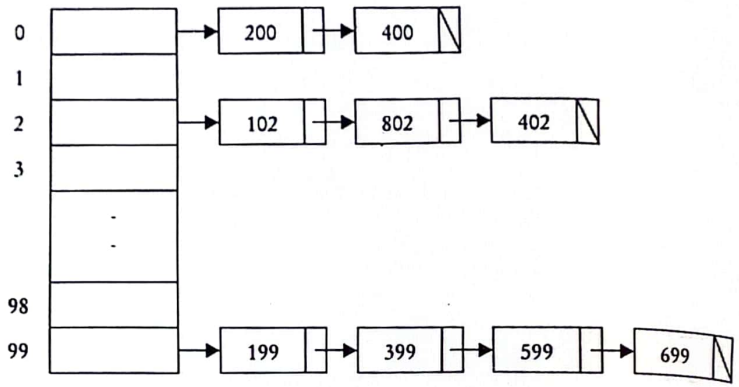


Figure-10.5: Hash table in Chaining Method

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 node of value 102. Here, we have to use the same hash function for storing and retrieving data.

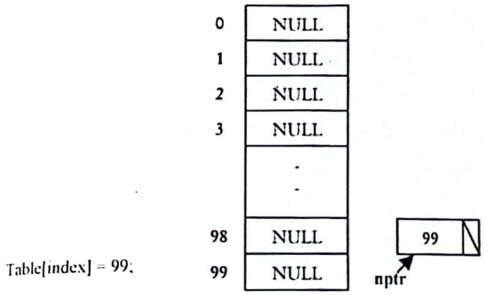
Algorithm 10.7: Algorithm to create a hash table using chaining method

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* = 0; *i* <*m*; *i*++)
   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 {

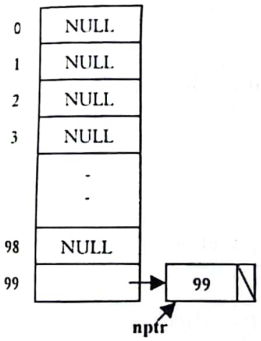

```
tptr = table[index];
while (tptr→next != NULL)
  {
  tptr = tptr→next;
  }
tptr→next = nptr;
}
```

5. Repeat Steps 3 to 4 to include another node.
6. Output: a chain of linked list

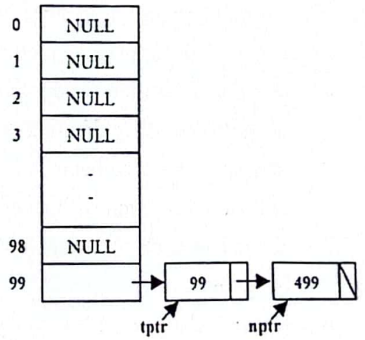
Comment: ***table[m]*** is an array of pointers of *m* size, ***nptr*** is the pointer to the new node and ***tptr*** is a temporary pointer that is used to make the link between existing last node and the new node.



(a): A table with null values and a new node with pointer nptr



(b): Adding a new node when table[index] is null



(c): Adding a new node with existing node

Figure-10.5: Create a hash table using chaining method

Summary:

Hashing is the method of storing and retrieving data by searching in/from a table called *hash table* using a special function called the *hash function*. There are several methods to find out the position (address) using the hash function: division, mid-square, folding method etc. In division method, key-value divided by the size of the hash table and remainder is used as an address (index) in the table. In the 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 a 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 a ***hash collision***, which is the situation when the 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 the linear probing method.
3. Write an algorithm to add items to a hash table. Here you have to solve the collision using chaining method.
4. What is a hash collision? Explain the linear probing method of hash collision resolution with an example.
5. What is a 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 are 60, each contains 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 contains 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 Employee Type 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 a **collision** has been resolved using the **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 a 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 a **collision** has been resolved using the **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 to the **problem 10-1**.

Problem 10-3: write a program to **store data** (key values) in a hash table using a hash function where a **collision** has been resolved using the **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 a **collision** has been resolved using the **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 a **collision** has been resolved using the **linear probing** method. Display the data of all the students and respective indices of the table. Each student record contains the 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 a collision has been resolved using the 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 to the **problem 10-5**.

Problem 10-7: write a program to **store data** (key values) in a hash table using a hash function where a **collision** has been resolved using the **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 a **collision** has been resolved using the **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 to 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
- 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 array Length = 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 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 and operations 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 Array List

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,

```
a1.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 al)
  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,

```
a1.Add(key,number/string);
```

To get the value by Key,

```
string my = (string) a1[key];
```

To get the value by Index,

```
string another = (string) a1.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() method
 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

                                {

                                p.Right.Left;

                                        Node s =

                                        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.

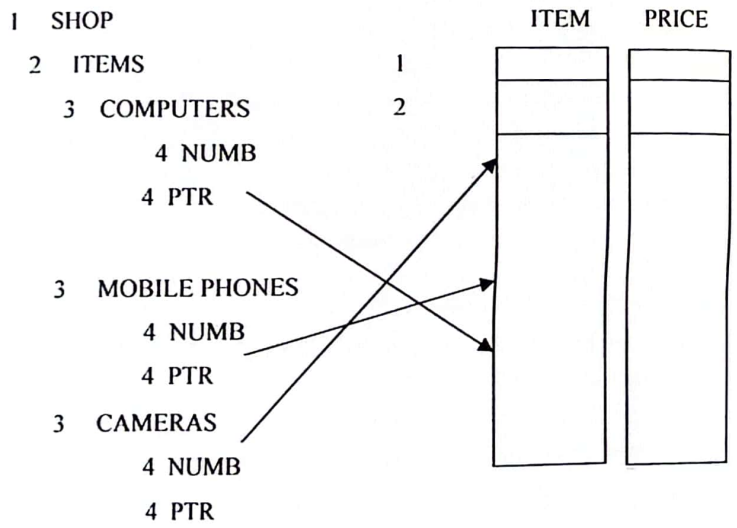
- Sorting can be done by using a method named "sort" in C sharp. Bubble sort, quick sort, Insertion sort etc. class can be implemented. In Any program, bubble sort mechanism can be used by creating an instance of Bubble sort class. In the same way, in any program, quick sort mechanism can be used by 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 $x_1, x_2, x_3, x_4, \ldots, x_n$ is defined by,

$$Avg = (x_1, x_2, x_3, x_4, \ldots, x_n)/n$$

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)

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

Prof Dr Md Rafiqul Islam obtained Ph.D. in Computer Science from Universiti Teknologi Malaysia (UTM) in 1999 and a combined Master (MS) and Bachelor Degree in Engineering (Computers) from Azerbaijan Technical University in 1987. He was a visiting fellow (a post doctoral researcher) in Japan Advanced Institute of Science and Technology (JAIST) in 2001. He worked as head of the Discipline of Computer Science and Engineering of Khulna University and as the Dean of the School of Science, Engineering and Technology of the same university. From October 2009 to September 2014 he worked as a Professor in the Department of Computer Science (CS) of American International University-Bangladesh (AIUB). Currently, he is working as a professor in Computer Science and Engineering (CSE) Discipline of Khulna University, Khulna. He has 28 years of teaching and research experiences. He has also about 4 years of industrial experience. He has 116 research papers, which have been published in international and national journals as will as in international conference proceedings. His international journal papers have been published by IEEE, Elsevier, Springer, Willey, Academy publisher of UK and others. His publication in refereed international conference proceedings include papers published by IEEE, Springer and other publishers. He has reviewed papers submitted to international journals of IEEE, Wiley, Springer, Elsevier science, Taylor & Francis, and papers submitted to journals of the different universities of Bangladesh and papers submitted to several International Conferences. He worked as the supervisor of several master's students of Computer Science Department of AIUB. He supervised many undergraduate and MS students of CSE discipline of Khulna University and CS Department of AIUB. He examined a Ph.D. thesis Computer Science and Engineering thesis of Bangladesh Engineering and Technology (BUET), Several Master's these of CSE department of BUET, Several Master's theses of CSE Department of Dhaka University. At present he is supervising two Ph.D. students and several MS and undergraduate students. He conducted an international conference as an organizing co-chair and worked as session chair of several international conferences. His research areas include design and analysis of algorithms in the area of Big Data, cloud computing, external sorting, Information security, data compression, bioinformatics, information retrieval, and grid computing. He has written a book titled "Data Structures Fundamentals", 1st and 2nd Editions that have been published by IUT, Gazipur, Bangladesh.

Prof Dr Md Abdul Mottalib received his Ph.D. in Computer Science and Engineering from Indian Institute of Technology (IIT), Kharagpur, India in 1993 and M.S. in Computer Science from Asian Institute of Technology, Thailand in 1984. He secured first class second position and first class first position in his B.Sc.(Hons.) and M.Sc. in Applied Physics & Electronics from Dhaka University in 1976 (held in 1978) and 1977 (held in 1980) respectively. Currently he has been working as a Professor and Head of the Department of Computer Science and Engineering (CSE), University of Liberal Arts Bangladesh (ULAB), Dhaka since October 2019. He served BracU as Professor and Chairman of CSE department during Aug 2017 to Sept 2019. He served as Head / Professor of CSE department of Islamic University of Technology (IUT), Board Bazar, Gazipur, Bangladesh from September 1998 to May 2017. Before joining IUT, he has worked as a Professor / Chairman of the Department of Computer Science in the Dhaka University where he served since 1993. Earlier he worked as a faculty member in various positions in Applied Physics & Electronics Department, Dhaka University since 1982 and 1 year in Bangladesh Atomic Energy Commission. He has been working as an expert member in many National & International important Committees. He served as examiner of many PhD, MSc and BSc thesis. In 1984, he developed ever first computer based Bangla software "Bangla". Dr. Mottalib achieved a Gold Medal as best paper award for his research paper on "Computer Based Bengali Voice Synthesis" that was presented in the International Conference on Computer and Information Technology in January 2001.

Dr. Mottalib has published more than 117 research papers in various International & National Journals and Conferences proceedings and visited around 25 countries to attend conference, seminar, workshops and training programs. He supervised more than 113 thesis works. He also worked as organizing chair, session chair, keynote speaker, paper reviewer etc. in various national & international seminar, workshop, conference etc. He was a member of National syllabus committee for Computer Science of SSC and HSC, NCTB, Govt. of Bangladesh, 1995.

He is an author of the books on Data Structure Fundamentals (1st & 2nd edition), HSC Computer Studies I & II, compiled 2 text books on MSc IT teaching manuals (Theory and Practical) for National University, edited a text book on Database Management system of Open University and reviewed many books for publication. He was a presenter and course teacher in television.

ULAB
UNIVERSITY OF LIBERAL ARTS BANGLADESH