Array

Data Structure

Linked List

Stack

Queue

Data Structure

Data Structure

Dr. Babasaheb Ambedkar Open University

Data Structure

Expert Committee

Prof. (Dr.) Nilesh K. Modi Professor and Director, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	(Chairman)
Prof. (Dr.) Ajay Parikh Professor and Head, Department of Computer Science Gujarat Vidyapith, Ahmedabad	(Member)
Prof. (Dr.) Satyen Parikh Dean, School of Computer Science and Application Ganpat University, Kherva, Mahesana	(Member)
M. T. Savaliya Associate Professor and Head Computer Engineering Department Vishwakarma Engineering College, Ahmedabad	(Member)
Mr. Nilesh Bokhani Assistant Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	(Member)
Dr. Himanshu Patel Assistant Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	(Member Secretary)

Course Writer

Mr. Kameshkumar Raval   Assistant Professor, Som Lalit Institute of Computer Application, Ahmedabad

Content Editor

Mr. Nilesh N. Bokhani   Assistant Professar, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad

Content Reviewer

Prof. (Dr.) Nilesh K. Modi   Professor and Director, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad

	BSCIT-203

Dr. Babasaheb Ambedkar Open University

Data Structure

BLOCK-1:INTRODUCTION TO DATA-STRUCTURES AND ARRAYS

BLOCK-2: STACK, QUEUES AND LINKED LISTS

BLOCK-3: TREES AND GRAPHS

BLOCK-4: TECHNIQUES (SEARCHING AND SORTING) AND FILE STRUCTURE

Dr. Babasaheb Ambedkar Open University

BSc(IT)-203

Data Structure

BLOCK1:   INTRODUCTION TO DATA-STRUCTURES AND ARRAYS

BLOCK 1: INTRODUCTION TO DATA-STRUCTURES AND ARRAYS

Block Introduction

Data Structures is an important subject of Computer Science and Engineering which teaches sets of important algorithms and its implementations.

In this block, we will start our discussion from the basic data-types. Which is also known as primitive data types or system defined data types. Some other types of data types such as Derived data types, User-Defined data types, and Abstract data types are briefly discussed. After defining data structures, we have discussed different types of data structures like linear and non-linear data structures are explained in brief.

In Unit: 2 we have focused on very first and simplest data structure that is array. We have tried to define the array and tried to explain basic terms related to arrays. We have also explained some basic operations like Traversal, Insertion and Deletion on array.

At the last in Unit:3, we have extended our discussion and explained more array operations such as Merging, Searching and Sorting of arrays. Along with that we have discussed 2-Dimessional arrays, its matrix representations and Sparse matrix in more details.

Block Objective

The objective of the block is to aware students, about different types of data structures. Student will know, what is data structures, how it can be programmatically implemented into the program? Students will be able to learn about Linear and Non-Linear data structures.

Main objective of this block is to aware students, about Array data structure. Student will learn Arrays, basic terminologies associated with an array and operations can be performed on the arrays.

Finally, the block will clear the concept of Arrays, 2-Dimessional array and its matrix representation. Practical implementation of Transpose of Matrix, Addition of matrices, Multiplication of matrices and Sparse Matrix.

Block Structure

BLOCK1:   BASICS OF C   3

UNIT1   INTRODUCTION OF DATA STRUCTURES AND KEY TERMS

Objectives, Definitions, Datatypes and Variables, Introduction to Data Structures, Let Us Sum Up

UNIT2   ARRAYS

Objectives, Introduction, Array terminologies, Operations on Arrays, Traversal, Insertion, Deletion, Searching and Sorting, Let Us Sum Up

UNIT3   MORE ABOUT ARRAYS

Objectives, Introduction, Other array operations, 2-Dimessional Arrays, Operations on 2-Dimessional arrays, Sparse matrices, Let us Sum Up

Unit 1: Introduction of Data Structures and Key Terms

1

Unit Structure

1.0   LEARNING OBJECTIVES:

In this unit, we will discuss about the basics of data structures, what are data structures and why it is essential to learn.

After learning this unit, you should be able to:

- Comprehend data structures and their types
- Understand the basic terms like data, information and system
- Understand variables and datatypes
- Learn different types of data types and difference between them

1.1   DEFINATIONS:

Before we start with the discussion of: What Data-Structure is? we will focus on few important terms which are related to Data-Structure. In this section we will try to define and understand some key terms which will help you to understand Data-Structure and its importance.

1.1.1 Computer or System

A Computer or System is an electronic machine which is able perform arithmetic and logical operations at very fast speed. It able to store the data and retrieve the information to the user depending upon users' request.

The word 'Computer' is actually derived from the word 'Compute', which means doing calculations. But in our modern computer, most work done by the people using computer which doesn't have any kind of computation. For example, Modern computer system can play music, play video, enables you to chat with your friend, or allows you to send a mail or browse information from the Internet. In all these activities discussed above, there no, or very less amount of computation is involved. Then the question is: How can we define our computer system? In a simple way, we can define a modern computer as a data processor which takes data as an input, process it and finally produce information as an output.

1.1.2 Data

The term, Data is used for unstructured facts and unstructured raw material, which provides necessary input(s) to the computer system. Here, the word unstructured represents, a value or set of values which are not in structured or prescribed format or not processed.

Data can be found and available in different formats. For example, integer numbers like 11, 28, 1976 etc., numbers with decimal points like 1.5, 22.11, 2500.1 etc., alphabets like 'A', 'a', 'E' etc., group of characters which also known as strings like "B.Sc. IT", "Data-Structures", "BAOU", you name etc.

Data is a value or set of values which is not processed, come from the natural facts. For example, "Geeta is present today in the class", "Mohan gets 90 marks in the Maths" etc. By mean of today's attendance, no one can predict that Geeta is very regular student and attending the class almost every day. Similarly, from the marks of only one subject, we cannot predict that the Mohan is passed in all subjects.

1.1.3 Information

The processed data is called 'Information', which we want to obtain from. We need information, so that we can take some decision on the bases of it. For example, if we record attendance of Geeta every day (which called data) for particular semester and after processing it if we find her attendance is 87.25% is an information. We can say that Geeta is a regular student for that particular semester. If we collect marks of the Mohan for all the subjects of particular semester and generate the grade sheet after processing the marks is called information. So, information is nothing but collection of processed data in desired format, which produces some meaningful output.

From the above examples it is clear that data is unstructured facts and input raw material for the system, whereas information is the processed data which represents some meaningful or summarized data format, produced as an output by the computer system. Following figure [Fig. 1.1] will further clarify the key differences between 'data' and 'information'.

Diagram shown above is describing how system act as a data processor, which takes data from the input devices, processed it and produces information as an output. System uses storage unit, which consists of primary memory to store intermediate results and data at the time of processing and secondary storage to store information.
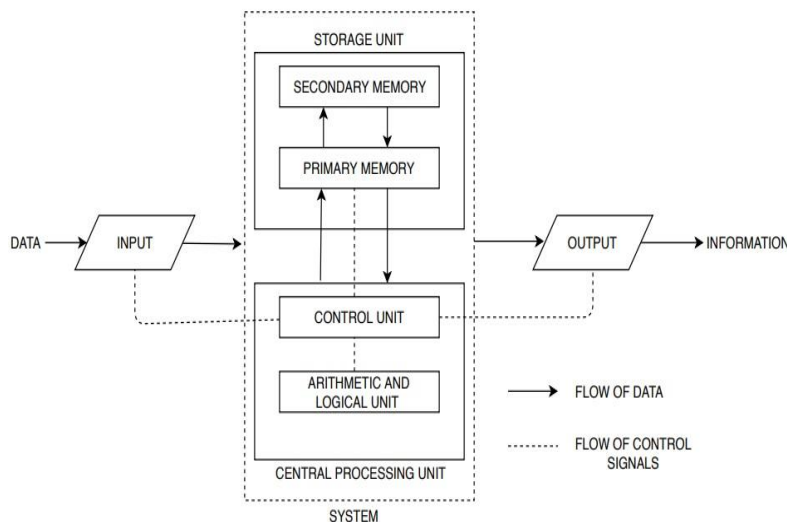
Fig. 1.1 Data – System – Information diagram

Check Your Progress-1

1. _____ means unstructured facts and raw material, which provides input to the system.

[A] Information   [C] Data

[B] System   [D] None of these

2.A system can perform _____.

[A] Arithmetic operations   [C] Logical operations

[B] Processing of data   [D] All of the above

3. Information is nothing but _____.

[A] Processed data   [C] Unstructured raw material

[B] Input for the system   [D] Facts in unstructured format

1.2   DATA TYPES AND VARAIBLES:

In most programming languages, to store the data we need to acquire some memory space. For example, in the C-programming language, if we want to store an Integer value 11, then first we need to declare a variable like:

#include<stdio.h>

void main ( )

{

  int x;

7

x=11;

printf("Value of variable x is:%d", x);

}

In the above program, we have declared variable 'x' (as we have written the statement called '*int x;*'), which reserves some space in the computer's memory. Now, in this memory space we can store any Integer value like '11' by writing a statement called '*x = 11;*'. Here, because we can change the value of 'x' anytime, it is called as integer variable. Variables are used by most programming languages, to store data and information into the memory of computer system. The last statement, will fetch the value of variable 'x' from the memory and print that value on the console screen with the help of '*printf()*' function call statement.

Now, looking to the above program, obviously one question will come into your mind that what is 'int' and why it requires in the declaration of variable 'x'? The answer is – it is a datatype, which represent, how much memory is given to the variable 'x'. In the ANSI C, data types 'int' means 2 bytes of space. Therefore, when you declare any variable of type 'int', then it reserves 2 bytes of memory space in which you can store -32768 as a smallest number and 32767 as a largest value. As per ANSI C standard, if you want to store a number larger than 32767, then you need to declare variable of type 'long int'. A variable of type long int can stores 4 bytes of memory space, and hence it can accommodate long integer value. Similarly, to store, numbers with decimal point, you need to declare a variable of type 'float' which need again 4 bytes of memory space. To store a character (any symbol available on Keyboard), you need to declare a variable of type 'char' (character) which occupies 1 byte of space in the memory.

Now, let us discuss how many different categories are there for different types of Data-types in details:

1.2.1 Primitive or Built-in data type

Every programming language should have its own set of data types. These data types are known as Primitive data types, Built-in data types or System defined data types. For example.

C-Language has int, float, char, double etc. data types

Language 'Basic' has Single, Integer, Char, String etc. data types

Language 'Pascal' has Boolean, Integer, Real, Character etc. data types

1.2.2 Derived data types

Derived data types are those data types which are defined by the user which extend the capability of basic (primitive) data types. As we have discussed, 'int' is primitive data type. Variable to data type 'int' can store single integer value at a time, but if we declare any pointer variable of type 'int *' then it will store the address of any one integer variable instead of integer value. Because of, pointer extends capability to store addresses, it is derived data type. Similarly, if we declare an array like 'int x [5]', then variable 'x' can store 5 integer values instead of single value. Array is also extending capability of basic (primitive) data types and permit users to store more than one values under single unit, and it is also an example of derived data types. Therefore, the datatypes like pointers, arrays etc. which are made from the primitive data types to extends their capabilities are called derived data types.

Therefore, those datatypes which are derived from basic data types such as pointer, array, structure, class etc. are called used defined datatypes.

1.2.3 User defined data types

C, C++, Java and many other languages allow programmer to create new and customized data types like student, employee, customer etc., by using basic data types which are called user defined data type. Enumeration, Union, Structure, Class, etc. are user defined data types. Because of programmer or user can defined data types as per their application's need, they are called user defined data types or simply UDFs.

1.2.4 Abstract data types

Abstract data types are basically a mathematical model of set of data elements that shares similar type of Behavior and independent of implementation. In the process of problem-solving for some complex problem, it is necessary to reduce programming complexity. To reduce complexity and understand the nature of the problem, programmer uses abstraction process. In the abstraction process, rather than considering whole problem, implementation of individual data elements to be declared and methods to be implemented is focused. For example, in the application in which stack is used, programmer is thinking stack as abstract (which is either encapsulated or hidden). Regardless of what data is stored in the stack, programmer is focusing on the function, how to add an element (push function) and how to pull out an element (function pop). Abstract data types are also known as its abbreviation called ADT.

Therefore, to reduce the programming complexity of a complex program, abstract data type is used where stack, queue become abstract and how insertion and deletion is done so that LIFO (Last-In First-Out in stack) and FIFO (First-In-First-Out in queue) can be

employed is focused. Abstract data types are hypothetical concepts and used in design and analysis, data structure, developing of system software to reduce complexity.

1.2.5 Data Structures

Data structure is nothing but the implementation of abstract data types. As we have discussed in the section 1.2.4 that, abstract data types are logical, whereas data structures have concrete implementation. For example, implementation of List or Collection can be done using Array or Link List. Because of array, link list, stack, queue etc. are implementation, they are considered as data structures.

Check Your Progress-2

1. Implementation of abstract data types are called _____.

[A] User defined data types   [C] Primitive data types

[B] Built-in data type   [D] Data structures

2. _____ are mathematical model and the don't have concrete implementation.

[A] User defined data types   [C] Primitive data types

[B] Abstract data types   [D] Data structures

3. The data types available Built into the system like (int, float, char etc.) are called _____ data types.

[A] User defined   [C] Primitive

[B] Abstract   [D] Data structures

1.3   INTRODUCTION TO DATA STRUCTURES:

As we have discussed in the previous section that the data structures are implementation, now in this section, we discuss different types of data structures.

There two types of data structure:

[1] Static data structures

[2] Dynamic data structures.

Static data structures are those, which occupies fixed amount of memory size, which can be determined before executing the program. When we declare '*int arr[5];*' then array arr, is a static data structure, as it is occupying fixed size of memory and not be able to store more than 5 elements. But Link-List is an example of dynamic data structure, in which memory allocation is made at runtime, when user is requesting to do so.

Data structures can also be classified in two main categories:

[1] Linear data structure

[2] Non-Linear data structures

In Linear data structure each element except first and last elements, fixed predecessor and successor. Array, Link-List, Stack, Queue are linear data structures. In the array or Link-List, apart from first and last elements every element has fixed one previous and one next element is there. In the linear data structures like array or Link-List all elements are arranged in a fixed sequential position. In case of Non-Linear data structures, elements can be arranged in any craving order (not following any predefined sequence) and there no restriction in the arrangement of element is followed. Graphs, Trees are two examples of non-linear data structures. Where any element of tree and graph can have n number of predecessor or successor. Linear and non-linear data structures are shown in Fig:1.2

In the figure [fig. 1.2] [A] and [B] represents linear data structures array and Link-List. In array and Link-List data elements follows strict sequential order. In the Link-List, we cannot visit 20 before visiting 5, 10 and 15. Where, [C] and [D] represent non-linear data structures tree and graph. In the tree data structure, after visiting node 'A' user can visit any node from 'B' or 'C'. Similarly, in the graph after visiting node 'C' user can either visit 'D' or 'E'. There is no strict sequence or order is there and also no order is followed in its implementation. Such data structures are called non-linear data structures.
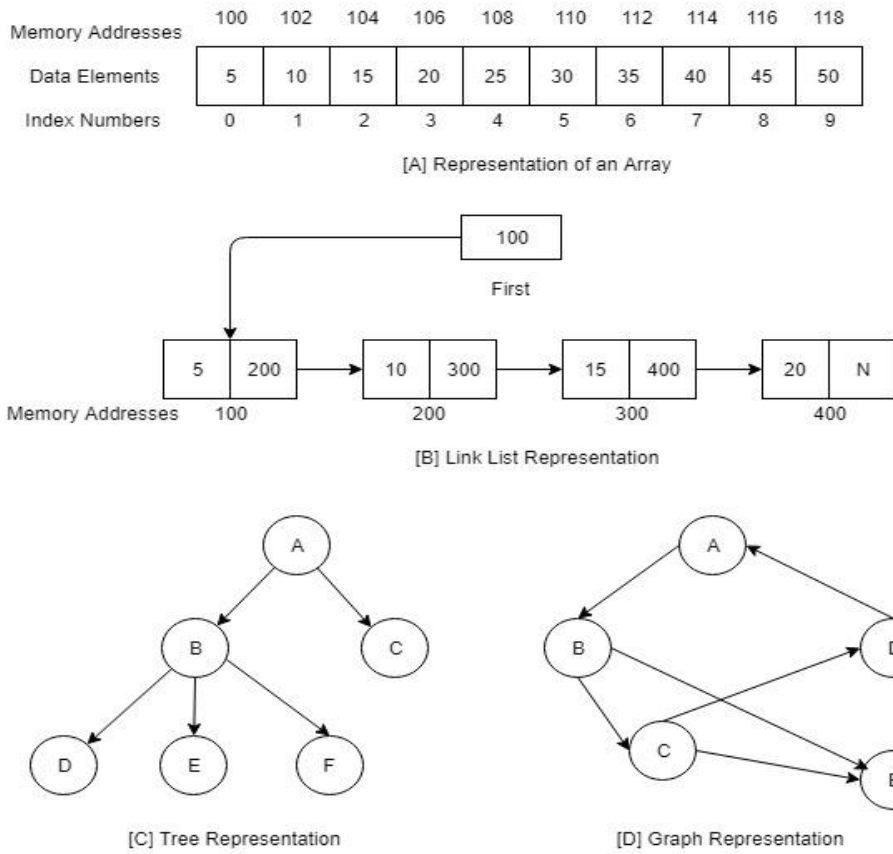
Fig. 1.2 Linear and Non-Linear Data Structures

Data structure is essential for computer science, can be defined as a triplet of Domain, Functions and Axioms which can be expressed as follows:

[1] **Domain**: Domain means the set of possible values. For example, 0, ±1, ±2, ±3, …. etc. For example in the design of queue data structure if you take an array of type integer then, data must in the range of -32,768 to 32, 767. Domain represents a set of all possible values which data structure might store as a data.

[2] **Functions**: Function is a set of permissible operations that can be accomplished on the data structure. For example, in the stack insertion is done by function push () and deletion of the data element is done by using function pop (). User cannot add or delete the element in the stack in any other way.

[3] **Axioms**: Axioms are set of rules which are associated with each function. For example, in the stack, push () is a function which always insert a value on the top position if the stack in not full, similarly if stack is not empty then data element can be deleted from the top of the stack when function pop () is invoked by the user.

Data structure is a triplet of Domain, Functions, and Axioms.

1.3.1 List of data structures

As we know now what is data structure? Here is the list of data structures we will discuss in this course in more details.

[1] Arrays

[2] Link Lists

[3] Stacks

[4] Queues

[5] Trees

[6] Graphs

[7] Files

Check Your Progress-3

1. _____ represents a set of all possible values which data structure might store as a data.

[A] Axioms   [C] Domain

[B] Functions   [D] None of the above

2. _____ is/are non-linear data structure(s).

[A] Graph   [C] Tree

[B] Both A and B   [D] Array

3. _____ are set of rules which are associated with each function.

[A] Axioms   [C] Domain

[B] Functions   [D] None of the above

1.4   LET US SUM UP

In this unit, we:

- Discussed basic terms like data, information and system
- Elaborated data types and variables
- Explained Linear and Non-Linear data structures
- Differentiated static and dynamic data structures.

1.5   SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

1. [C] Data
2. [D] All of the above
3. [A] Processed data

Check Your Progress-2

1. [D] Data Structures
2. [B] Abstract Data types
3. [C] Primitive

Check Your Progress-3

1. [C] Domain
2. [B] Both A and B
3. [A] Axioms

1.6   GLOSSARY

1. **Data** is used for unstructured facts and unstructured raw material, which provides necessary input(s) to the computer system.
2. **Information** is processed data which described meaningful representation of data.
3. **Abstract data** types are basically a mathematical model of set of data elements that shares similar type of Behavior and independent of implementation.
4. **Axioms** are set of rules which are associated with each function.

1.7   Assignment

1. List and explain different types of data structures.
2. What are abstract datatypes? How it differs from data structure?

1.8   Activity

1. Categorize the following terms into Primitive data types, Derived data types, User-defined data types, Abstract data types and Data structures.

- Int
- Structure

- Class
- Array
- Stack

  to reverse all the values (nodes) of the link list using recursion.

1.9 Further Reading

- Data Structure through C by Yashvant Kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.

Unit 2: Arrays   **2**

Unit Structure

2.0   LEARNING OBJECTIVES:

In this unit, we will discuss about Array data structure. As you know Arrays are simple linear data structure. We will learn Array data structure in this unit.

After learning this unit, you should be able to:

- Understand importance and need or arrays
- Learn different terminologies related to array
- Learn different methods to initialized single dimensional array
- Understand different array operations

2.1   INTRODUCTION:

Array is a finite and ordered collection of same type of (homogeneous) data. Array is a collection or set of data elements of similar data type. As array is collection or set of data, you can store more than one data item in the array under common name as a single unit, but make sure all items stored in the array should have same type.

2.1.1 Need of an Array

Consider the following program, where we have declared one variable named 'num' and we initialize it with value 28. Afterwards we assign a new value to it which is 11. Now suppose if we print the value of 'num' variable then try to predict that which value for variable 'num' should be printed on the screen?

#include<stdio.h>
void main ( )
{
    int num = 28;
    num=11;
    printf("Value of variable num is:%d", num);
}

If you have executed the program given above, then you may notice that it prints the value for 'num' variable is 11. But the question is: what happened to value 28? The answer is it is simply overwritten. It is deleted and newer value 11 gets stored when computer executes a statement 'num=11;'. That means that, one variable can hold only one value.

But what happen if we want to store more than one element like 10, 20 or 100 of elements. The simple idea will come into your mind is, declare 100 variables to store 100 values. Yes, you can do this, but it not good idea because there are some problems are there:

1. To store 100 values, you need to think 100 unique names for each of your variables.
2. Suppose, if you want to take all these values from the user, then you need to write 100 printf() and scanf() statement (for each variable).
3. Managing large number of variables in the programming makes you program to more complex, not readable and not easy to understand.

OK, then; what is the solution? The solution is array. Consider the following program, where we store multiple values, in the same variable (array) with overwriting each other. To accept the multiple values, instead of writing multiple printf() and scanf() statements, we will loop, which will be repeated for N times, to take N values from the user and stored it into the array of N size.

```
int nums [10] , i ;
for (i=0; i< 10; i++)
{
        printf ("Enter Any Number:");
        scanf("%d", & nums[i]);
}
```

In the above program loop of variable 'i' will be repeated for 10 times, which takes 10 values from the user and stored all the values in the array of size 10.

2.1.2 Defining an Array

If in the C-Language, if you write '*int num [10];*' in the declaration section of the C-program then you have a single variable (array) having name 'num' which is able to store 10 data items of same type (in this case data items are of type int) for you.

2.1.3 Initializing an Array

Array can be initialized in three different ways:

[1] **Initializing array at the time of its declaration**

```
int x [5] = {1,11,21,31,41};
char y [5] = {'a' , 'e' , 'i' , 'o' , 'u' };
```

in the above example, array 'x' is an integer array which is initialized by 5 data elements i.e. 1, 11, 21, 31 and 41. First data element '1' will be stored on the first i.e. 0th position (as array always start from index 0) of the array x. That means x [0] is '1'. Similarly, 11 is stored on 1st position of array x, 21 is stored on 2nd position and so on. Finally, data

element 41 will be stored on 4th position. In C-Language array index starts from 0, therefore last element we need to put at position Size – 1. In the example, the size of array 'x' is 5, so last data element '41' will be stored on passion 4.

In another declaration, we have declared an array 'y' of type character, which is initialize by 5 character type data items i.e. 'a', 'e', 'i', 'o' and 'u'. First data item 'a' will be stored on 0th position of array 'y' and last character data item 'u' will be stored on 4th position.

In this example, we have hard coded data items in the array. That means we have initialized both arrays with some fixed values (values are not taken by the user). This type of initialization of the array is called static initialization.

[2]   Declaring array and initializing it with some static values:

int x [5];
char y [5];
x [0] =1;
x [1] = 11;
x [2] = 21;
x [3] = 31;
x [4] = 41;

and so on. Similarly, array 'y' can be initialized as,
y [0] = 'a';
y [1] = 'e';
y [2] = 'i';
y [3] = 'o';
y [4] = 'u';
and so on.

This is again static declaration, where we have initialized each data item of an array with some fixed value assignment.

[3]   Declaring and initializing array by user input

int num [10] , i ;
for (i=0; i< 10; i++)
{
        printf (" Enter Number:");
        scanf("%d", & num[i]);
}
In this example, we have declared an array 'num' with the size 10. We are initializing an array by accepting values from the use. To accept 10 values, and stored it in the array, we

are running a loop for variable 'I', which will run from i=0 to i=9 (10 times), which accept 10 integers values from the user and stored in in the array from num[0] to num[9].

In this method we accept the data elements from the user to initialize our 'num' array, it is called dynamic initialization of an array.

<div style="border:1px solid">

Check Your Progress-1

1. _____ is a finite order of homogeneous collection of data.

[A] Pointer   [C] Structure

[B] Array   [D] Enumeration

2. Array index in C-Language always starts with _____.

[A] 1   [C] 0

[B] 10   [D] Size -1

3. In static initialization of an array, array items should be encased in _____.

[A] Between [and]   [C] Between (and)

[B] Between { and }   [D] Between /* and */

</div>

2.2   ARRAY TERMINOLOGIES:

1. **Type**: Type specifies type of data items that can be stored in the array. As we have discussed that array is finite set of ordered data items of same type of data, and hence we need to specify what type of data is to be stored in the array at the time of its declaration. For example, array can be of type char, int, float, double etc.

2. **Size**: Size of an array denotes maximum number of data items that can be stored in an array. For example, if we declare 'char name [10]' then array 'name' can store maximum 10 data items. So, we can say size of array 'name' is 10.

3. **Index**: Each element kept in the array has sole reference number is called index number. It is denoted by the square brackets like [ ]. As we know in the above example x [0] = 1, means data element 1 is stored in at the index number 0 of the array 'x'. In most programming languages like C-Language, array index starts with 0.

4. **Range**: Range is a set of all valid index numbers. We know that, in most cases array index starts from 0, it is also known as Lower bound of an array. If we declare array 'int x [4];' then last element we can place on 4th position. This is called upper bound of an array, where we store last data item. Set of all possible integers from lower bound to upper bound is called range of an array. So, the range of array x is 0 to 4.

5. **Base**: Memory address from where array starts in the memory is called base address of that array. Usually, it the memory location of the first data item of an array. Remember, array stores all of its data item in sequential (continuous) memory locations. In the programming language like C, array name itself represent the base address of an array.

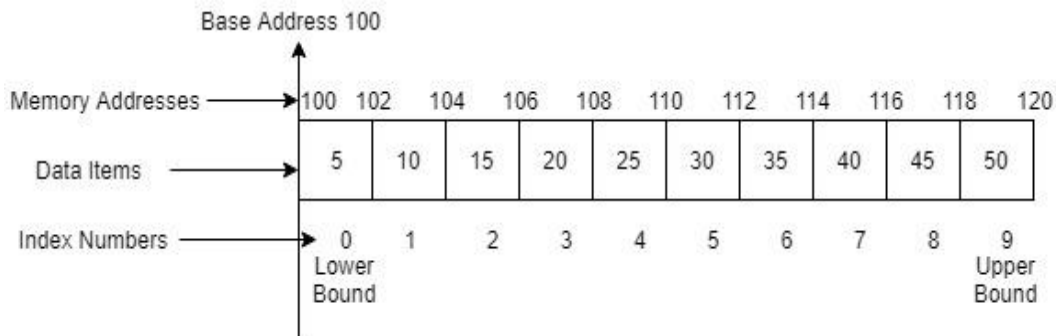Consider the figure given below, which will clear all the terms we have discussed above in an easier way.

Fig.2.1 Terminologies of an Array

In the figure [Fig.2.1] memory address representation of an integer one dimensional array is shown. Array with 10 integer values, is stored in the consecutive (sequential) memory locations starting from 100 to 120. 100 is the initial address of an array, which is called base address. Because the data type of an array is 'int' and we know that each integer occupies 2 bytes of space in the memory first data element of an array i.e., '5' will be stored between memory addresses 100 to 102. Second value 10 will be stored between memory addresses 102 to 104 and so on. Index number of first data item 5, is 0 is also called a lower-bound of an array and in the same way, last data item is stored on 9th position, which is called upper-bound of an array.

Let is we have an array having Lower-bound LB and upper-bound UB. Then the index number of ith element should be always: Index $(Xi) = LB + i - 1$. For example, in C-Language index number of 6th element will be $0 + 6 - 1 = 5$.

Similarly, size of an array is: Size $= UB - LB - 1$. In C-Language if UB of and array is 9 the size $= 9 - 0 + 1 = 10$.

The figure [Fig. 2.1] will explain all the terms discussed above in pictorial representation. Array can be of One – dimensional (Linear), Two – dimensional (Matrix) or Multi – dimensional. One – dimensional and Two – dimensional arrays are discussed in this unit in greater detail below.

2.3   OPERATIONS ON ARRAYS:

Different types of operations (actions) can be performed on the array such as, Traversing, Insertion, Deletion, Searching, Sorting, Merging etc. Let's discuss one by one in detail with practical implementation of each:

2.3.1 Traversal

Traversal is the process of visiting or accessing each element of an array. For example, consider the program given below, where after declaration of the array, we take values from the user and store all values in the array. After storing all the values, we are running a loop which will print all the values stored in the array on the console screen.

```
#include<stdio.h>
void main ()
{
  int arr [10];
  int i;
  printf("\nEnter 10 data items for Array: ");
  for (i=0; i<10; i++)
  {
    printf("Enter Value:");
    scanf("%d", &arr[i]);
  }
```

```
    printf("\nArray contains: ");
    for (i=0; i<10; i++)
    {
        printf("%d\t", arr[i]);
    }
}
```

The second for loop in the program given above, which access one by one data element from the array and print those data elements on the console screen is called traversal of an array.

2.3.2 Insertion

Insertion in the array means, a newer data item is inserted at some specific index number. But what happen if we have some data item? In this case all the data items are shifted towards right direction to create a space on that particular index position and then the newer data item is placed.

Consider an array having some values are already inserted and other cells of an array are empty (having value 0). Now suppose user want to insert a new data element on 4th position. In this case we have to move 9th element of an array to 10th position, 8th element has to move on 9th position and in that way all the elements of the array up to 4th position, has to be moved at right side. Finally, the newer data item has to be placed on the 4th position. The following figure [Fig.2.2] shows how the data items are moved to make a space for the new data item and how the new element is inserted in the array. Data item stored on the upper bound (the last data item) of an array (i.e., 0 in our case) will be discarded.

Array Before Insertion

| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 0 | 0 | 0 |

Insertion of Element 18 at position 4

| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 0 | 0 | 0 |

Discard last element 0

18

Array After Insertion

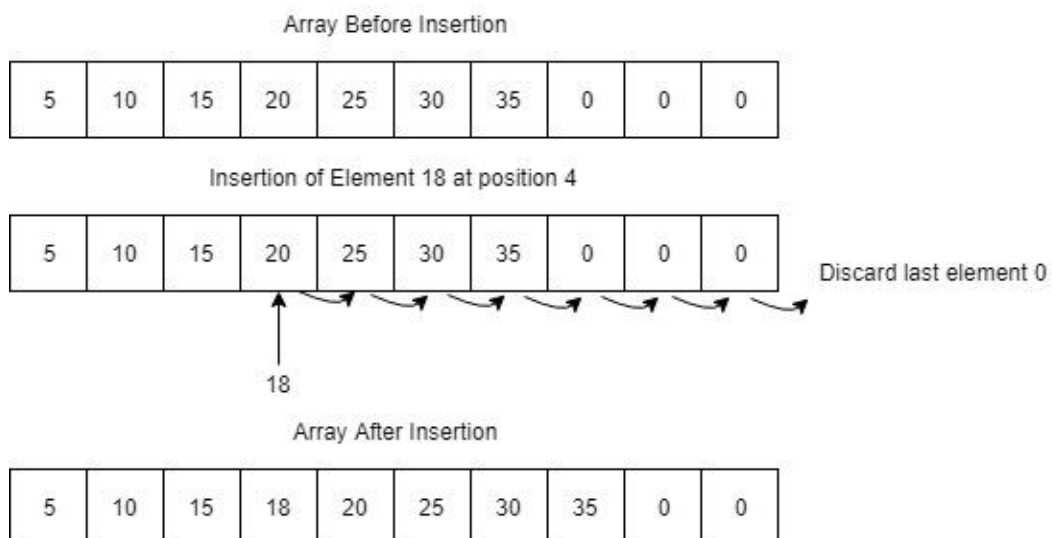| 5 | 10 | 15 | 18 | 20 | 25 | 30 | 35 | 0 | 0 |

Fig: 2.2 Insertion

Program to insert data item in the array is given below. In the program some elements are already inserted, whereas some data items are empty (initialized with 0). Program needs two inputs from the user. It accepts position where new data item to be inserted and the value of the data item to be inserted. Program needs to move all data items from that position to end. Last element of an array has to be discarded.

```
#include<stdio.h>
#define SIZE 10
void main()
{
  int arr[SIZE]={1,11,21,31,41,51,61, 0,0,0}, i, pos, val;
  printf("\nValues in the Array:");
  for(i=0; i<SIZE && arr[i]!=0; i++)
    printf("\n%d", arr[i]);
  printf("\nEnter Position:");
  scanf("%d", &pos);
  printf("Enter Value:");
  scanf("%d", &val);
  for(i=SIZE-1; i>=pos; i--)
    arr[i]=arr[i-1];
  arr[i]=val;
  printf("\nData items in the Array after Insertion:");
  for(i=0;i<SIZE && arr[i]!=0;i++)
    printf("\n%d", arr[i]);
}
```

Output:
Values in the Array:
1
11
21
31
41
51
61
Enter Position where you want to Insert the value:4
Enter Value:28
Data items in the Array after Insertion:
1
11
21
28
31
41

51

61

2.3.3 Deletion

In the insertion, we have moved all data items in the right direction from the index number where we are inserting new data item to end. In the case of deletion, we are deleting a specified data item from the array, which creates empty space in the array. To fill up this empty space you need to move all data items from right to left and at the end you need to add one '0'.

Deleting a data item from an array requires opposite process than insertion. In the previous program we have moved all elements by one position towards right. In the deletion, we have to move all elements of an array towards left. At the last position we have to add 0 (Empty) element. Removal process in the array is shown as below:
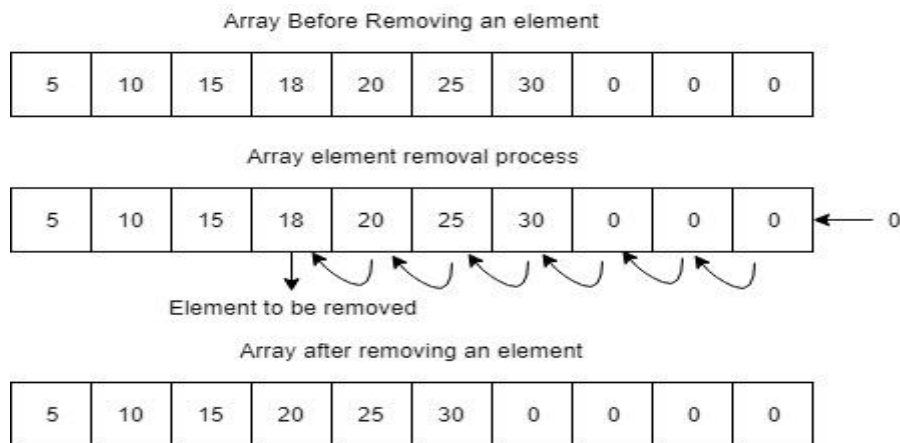
Fig: 2.3 Deletion

Program to delete a data item is given below, where user will give position and program will remove a data item from that particular position. It moves all other elements to the left direction, and append 0 at the end. Finally, it prints all data items of an array after deletion.

```c
#include<stdio.h>
#define SIZE 10
void main()
{
  int arr[SIZE]={1,11,21,28,31,41,51,0,0,0}, i, pos;
  printf("\nValues in the Array:");
  for(i=0;i<SIZE && arr[i]!=0;i++)
    printf("\n%d", arr[i]);
  printf("\nEnter Position to Remove data item:");
  scanf("%d", &pos);
```

```
  for(i=pos-1; i<SIZE-2; i++)
    arr[i]=arr[i+1];
 arr[SIZE-1]=0;
 printf("\nValues after deletion in the Array are:");
 for(i=0; i<SIZE && arr[i]!=0;i++)
    printf("\n%d", arr[i]);
}
```

Output:

Values in the Array:

1

11

21

28

31

41

51

Values after deletion in the Array are:

1

11

21

31

41

51

Check Your Progress-3

1. In _____ array operation we need to move all data elements towards right side by ignoring last element.

[A] Deletion   [C] Insertion

[B] Traversal   [D] Sorting

2. In _____ array operation we need to visit each data-element of it.

[A] Deletion   [C] Insertion

[B] Traversal   [D] Sorting

3. In _____ array operation we need to move all data elements towards left side by adding one zero as a last element.

[A] Deletion   [C] Searching

[B] Insertion   [D] Merging

2.4   LET US SUM UP

In this unit, we:

- Discussed the importance of an array in programming languages
- Elaborated array, it's initialization and data-elements in arrays
- Explained different terminologies related to an array
- Discussed different operations like Traversal, Insertion and Deletion.

2.5   SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

1. [B] Array
2. [C] 0
3. [B] Between { and }

Check Your Progress-2

1. [C] Index
2. [A] Range
3. [B] Size

Check Your Progress-3

1. [C] Insertion
2. [B] Traversal
3. [A] Deletion

1.6   GLOSSARY

1. **Array** is a set of ordered collection of data-elements with same type (homogeneous).
2. **Size** of an array denotes maximum number of data items that can be stored in an array.
3. **Base**: Memory address from where array starts in the memory is called base address of that array.
4. **Index** is a reference number, used to access particular data-element from an array. It is always start with 0 and index number should be mentioned in the square bracket like x[3] (which will gives you 4th data-element of an array).

2.7   Assignment

1. List and explain different array terminologies.

2. What is an array? How can we initialize array?

2.8   Activity

Write a program to do following operations in array:
1. Traversal
2. Insertion
3. Deletion

2.9   Further Reading

- Data Structure through C by Yashvant Kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.

Unit 3: More about Arrays 3

Unit Structure

3.0 LEARNING OBJECTIVES:

In this unit, we will discuss few more things about Array data structure. As you know Arrays are simple linear data structure. We will learn 2-Dimisinal array and matrix representation in this unit.

After learning this unit, you should be able to:

- Understand merging, searching and sorting operations on arrays
- Learn 2-Dimession array, it's declaration, initialization and representation
- Learn different operations on 2-Dimesional arrays
- Understand Sparse matrices and its array representation

3.1 INTRODUCTION:

In the previous unit, we have focused on Traversal, Insertion and Deletion operations. In this unit we will focus on some other operations like Merging, Searching and Sorting operations. After discussing it we will discuss different types of arrays and its programs. Finally, we will close our discussion with Sparce Matrices.

3.2 OTHER ARRAY OPERATIONS:

3.2.1 Merging of Arrays

Merging is the process of combining data-elements into one. For example, if there are two different arrays are given, and if you have third array with larger size, in which you copy all elements of first array, and then all data-elements of second array, then the third array is called merger of both arrays. If we print third array then we will get all data-elements of first and as well as second array.

Consider the following program in which, we have taken two sorted arrays, and we need to merge both the arrays into third array in such a way that our third array should be sorted.

```c
#include<stdio.h>
void main()
{
  int x[5]={2, 28, 39, 45, 67};
  int y[5]= {5, 11, 50, 70, 95};
  int z[10], i,j,k;
  printf("\nContent of Array:1\n");
  for(i=0;i<5;i++)
    printf("%d\t", x[i]);
  printf("\nContent of Array:2\n");
```

```c
    for(i=0;i<5;i++)
      printf("%d\t", y[i]);
    for(i=0,j=0,k=0; i<5 && j<5; k++)
    {
      if(x[i]<y[j])
      {
        z[k]=x[i];
        i++;
      }
      else
      {
        z[k]=y[j];
        j++;
      }
    }
    if(i==5)
    {
      while(j<5)
      {
        z[k]=y[j];
        k++;
        j++;
      }
    }
    if(j==5)
    {
      while(i<5)
      {
        z[k]=x[i];
        i++;
        k++;
      }
    }
    printf("\nContent of Merged Array:\n");
    for(i=0;i<10;i++)
      printf("%d\t", z[i]);
}
```

Output:

Content of Array:1

2	28	39	45	67

Content of Array:2

5	11	50	70	95

Content of Merged Array:

2	5	11	28	39	45	50	67	70	95

3.2.2 Searching in an Array

Searching is the process of finding the position of particular data-element in the array. As we have already discussed that the data-elements are stored at specific and unique index number. If we retrieve the index number from the particular data-element which is given then we can say that we have searched that particular data-element from an array.

To do the searching process, there two methods are there:

[1] Linear Search:

In this method we are comparing the search element with all the data-elements of an array. When we found an element, it's index number we are printing on the screen. In the case of linear search algorithm, because of we are comparing search element with all other data-element of an array, if there are N data-elements are there in the array then N comparisons are possible. That is the reason that Linear Search algorithm is slow. Linear Search algorithm can be applied on both sorted or unsorted array.

[2] Binary Search:

In this method, we should have a sorted array. Binary search algorithm can not be applied on unsorted array. Now, because our array is sorted, we are comparing search element, with directly with that data-element which is situated at middle position of an array. After comparison if data-element of an array match with search element, then we print the index number of the data-element situated at middle position. If search element is smaller than middle element then we decrease the size of array from 0 to mid-1. If search element is greater than the middle element then we decrease the size of array from mid+1 to end.

Because, one comparison brings half of the elements out of comparison, it takes a smaller number of comparisons, which means this is the fastest algorithm.

Programming implementation of both Linear search and Binary search are greatly explained in the Block-4 of this subject.

In the above program loop of variable 'i' will be repeated for 10 times, which takes 10 values from the user and stored all the values in the array of size 10.

3.2.3 Sorting

Sorting is a method of arranging data-element into some order. Array can be sorted by arranging data-elements from smaller to higher which is known as ascending order sorting as

well as by arranging data-elements from higher to smaller which is known as descending order sorting. Array can be sort by different methods, which are called sorting algorithms. Here is the list of different sorting algorithms:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort etc.

We will discuss all the sorting algorithm in great details, with its practical implementation in the Block-4 of this course.

Check Your Progress-1

1. _____ is an array operation, in which we arrange all data-element in order.

[A] Searching   [C] Sorting

[B] Merging   [D] Deletion

2. The process of combining two arrays into one is called _____ array operation.

[A] Searching   [C] Sorting

[B] Merging   [D] Inserting

3. In _____ array operation, we find index number of given data.

[A] Searching   [C] Sorting

[B] Traversal   [D] Deletion

3.3   2-DIMENSSIONAL ARRAYS:

Up to hear, we have discussed many examples of arrays. But in all the example we have considered only one type of array that is one-dimensional array. In a one-dimensional array we have multiple rows but only one column is there. When we talk about two-dimensional array, it has multiple rows and multiple columns. Usually, two-dimensional array is used to represent a matrix. The intersection of rows and columns are cells. In each cell, one value can be stored. Following figure will describe both types of array representations.
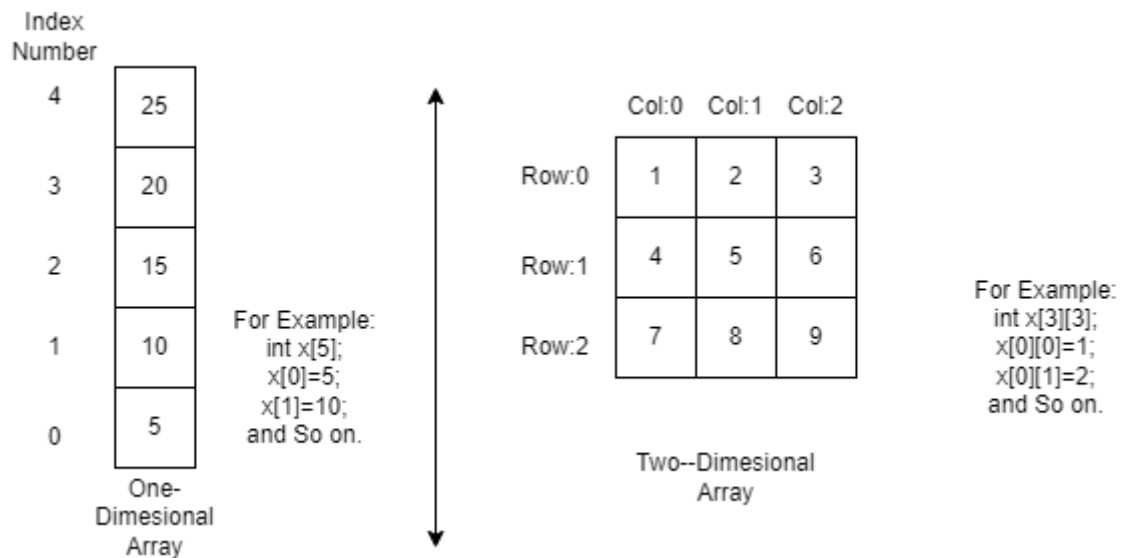
Fig: 3.1 One-Dimension and Two-Dimension Arrays

3.3.1 Declaration of 2-Dimesion Array:

In 1-Dimesional array we are declaring array like: 'int x[5];'. But because of in two-dimensional array, we should have multiple rows and multiple columns, it should be declared in the following way:

int x[3][3]; // This will declare an array of 3 rows and 3 columns which contains 9 elements

int y[4][5]; // This will declare an array of 4 rows and 5 columns which contains 20 elements

3.3.2 Array Initialization:

Array can also be initialized in 3 different ways:

[1] Static value assignment at the time of declaration:

For Example,

int x[3][3]={{1,2,3}, {4,5,6}, {7,8,9}};

In this example we have declared array x, which has 3 rows and 3 columns, which can accommodate 9 elements from 1 to 9.

[2] Static value individual data-element initialization:

For Example,

```
int x[3][3];
int x[0][0]=1;
```

```
int x[0][1]=2;
int x[0][2]=3;
int x[1][0]=4;
int x[1][1]=5;
```

and so on,

[3] Dynamic value assignment to 2-dimesional array:

For Example,

```
int x[3][3], i, j;
for(i=0; i<3; i++)
{
        For(j=0; j<3; j++)
        {
                printf("Enter Data:");
                scanf("%d", &x[i][j]);
        }
}
```

Make sure here, first we have declared an array 'x[3][3]' of type int. Then we have taken two variables 'i' and 'j' to run a loop for rows and column. We are running a loop for variable 'i' for three times as there are 3 rows are there in the matrix. For each iteration of 'I' we are running a loop of variable 'j' for 3-times as there are 3 columns are there in each row. The inner loop (of variable 'j') will run for 9 time (3time for each iteration of variable 'i'), will take 9 values from the user (from console) and stored it in the array 'x'.

3.3.3   Matrix representation of 2-Dimesional array:

To print the 2-dimesional array in the form of Matrix, we will run nested for loops for variable 'i' and 'j'. The following source code will print all 9 values of 2-dimensional array 'x' in the form of Matrix.

```
#include<stdio.h>
void main()
{
  int x[3][3]={{1,2,3}, {4,5,6}, {7,8,9}};
  int i,j;
  for(i=0;i<3;i++)
  {
    for(j=0;j<3;j++)
    {
      printf("%d\t",x[i][j] );
    }
    printf("\n");
```

```
    }
}
```

3.4   OPERATIONS ON 2-DIMESIONAL ARRAY:

Different types of operations (actions) can be performed on the 2-dimensional array such as, Displaying matrix, Transpose of matrix, Addition of two matrices and Multiplication of two matrices etc. Let's discuss one by one operation in detail with practical implementation of each:

3.4.1 Displaying Matrix

As we have discussed that, to store or represent the matrices we need to use 2-Dimesional array which should have 'm' rows and 'n' columns. To display the matrices, we need to use nested loops. Outer loop will keep track of rows, and make sure to change the row or adding new row we need to print "\n" character. Outer loop will be repeated for 'm' times, which will print 'm' rows on the console screen. Because each rows have 'n' values or columns, inner loop will print 'n' column value for each row. That means inner loop will be executed for 'n' times for each row, and we have 'm' rows that is m*n times. The following program will demonstrate how to print a matrix of 4*3. Here the value of m=4 and also n=3;

```
#include<stdio.h>

void main()
{
```

```
    int x[4][3]={{1,2,3}, {4,5,6}, {7,8,9}, {1,1,1}};
    int row, col;
    printf("Matrix X is:\n");
    for(row=0;row<4;row++)
    {
       for(col=0;col<3;col++)
       {
          printf("%d\t",x[row][col] );
       }
       printf("\n");
    }
}
```

Output:

Matrix X is:

1   2   3

4   5   6

7   8   9

1   1   1

3.4.2 Transpose of Matrix:

If a new matrix is created from given matrix, where rows of the original matrix is transformed in the column of new matrix as well as columns of the original matrix is transformed in to rows of new matrix, then the new matric is called transform of original matrix. The following program will demonstrate how to transform a matrix using 2-Dimensional arrays.

```
#include<stdio.h>
void main ()
{
   int mat [3][3], t_mat [3][3];
   int row, col;
   printf ("Enter values for Original Matrix:\n");
   for (row=0; row<3; row++)
   {
      for (col=0; col<3; col++)
      {
         printf("Enter Value:");
         scanf("%d",&mat[row][col]);
      }
   }
   printf("\nOriginal Matrix is:\n");
```

```
  for (row=0; row<3; row++)
  {
    for (col=0; col<3; col++)
    {
      printf("%d\t",mat[row][col] );
    }
    printf("\n");
  }
  /* Creating Transpose Matrix */
  for (row=0; row<3; row++)
  {
    for (col=0; col<3; col++)
    {
      t_mat[row][col]= mat[col][row];
    }
    printf ("\n");
  }
   printf ("\nTranspose Matrix is:\n");
  for (row=0; row<3; row++)
  {
    for (col=0; col<3; col++)
    {
      printf ("%d\t", t_mat[row][col]);
    }
    printf ("\n");
  }
}
```

Output:

Enter values for Original Matrix:

Enter Value:1

Enter Value:2

Enter Value:3

Enter Value:4

Enter Value:5

Enter Value:6

Enter Value:7

Enter Value:8

Enter Value:9

Original Matrix is:

1   2   3

4   5   6

7   8   9

Transpose Matrix is:

1   4   7

2   5   8

3   6   9

3.4.3 Addition of two 3*3 Matrices:

In the addition of two 3*3 matrices two matrices X and Y are given. We need to generate third matrix Z in such way that z[0][0] = x[0][0] + y[0][0], z[0][1] = x[0][1] + y[0][1] and so on. The following program demonstrate the addition of two 3*3 matrices.

```
#include<stdio.h>
void main()
{
  int x[3][3], y[3][3], z[3][3];
  int i, j;
  printf("Enter values for Matrix X:\n");
  for(i=0; i<3; i++)
  {
    for(j=0;j<3; j++)
    {
      printf("Enter Value:");
      scanf("%d",&x[i][j]);
    }
  }
   printf("Enter values for Matrix Y:\n");
  for(i=0; i<3; i++)
  {
    for(j=0;j<3; j++)
    {
      printf("Enter Value:");
      scanf("%d",&y[i][j]);
    }
  }
  /* Addition of Matrices */
  for(i=0; i<3; i++)
```

```
    {
      for(j=0;j<3; j++)
      {
        z[i][j]=x[i][j]+y[i][j];
      }
    }
    printf("\nMatrix X:\n");
    for(i=0; i<3; i++)
    {
      for(j=0;j<3; j++)
      {
        printf("%d\t", x[i][j]);
      }
      printf("\n");
    }
    printf("\nMatrix Y:\n");
    for(i=0; i<3; i++)
    {
      for(j=0;j<3; j++)
      {
        printf("%d\t", y[i][j]);
      }
      printf("\n");
    }
    printf("\nMatrix Z:\n");
    for(i=0; i<3; i++)
    {
      for(j=0;j<3; j++)
      {
        printf("%d\t", z[i][j]);
      }
      printf("\n");
    }
}
```

Output:

Enter values for Matrix X:
Enter Value:1
Enter Value:2
Enter Value:3
Enter Value:4
Enter Value:5
Enter Value:6
Enter Value:7
Enter Value:8

Enter Value:9
Enter values for Matrix Y:
Enter Value:11
Enter Value:12
Enter Value:13
Enter Value:14
Enter Value:15
Enter Value:16
Enter Value:17
Enter Value:18
Enter Value:19

Matrix X:

1	2	3
4	5	6
7	8	9

Matrix Y:

11	12	13
14	15	16
17	18	19

Matrix Z:

12	14	16
18	20	22
24	26	28

3.4.4 Multiplication of two 3*3 Matrices:

Multiplication of matrices are little bit complicate compare to addition of matrices. To multiply the matrices, we need to multiply each row of the first matrix to each column of another matrix and then we need to add them. Consider the formulation given below to get more idea of matrix multiplication of matrices.

Z[0][0] = X[0][0] * Y[0][0] + X[0][1] * Y[1][0] + X[0][2] * Y[2][0]

Z[0][1] = X[0][0] * Y[0][1] + X[0][1] * Y[1][1] + X[0][2] * Y[2][1]

Z[0][2] = X[0][0] * Y[0][2] + X[0][1] * Y[1][2] + X[0][2] * Y[2][2]

$$Z[1][0] = X[1][0] * Y[0][0] + X[1][1] * Y[1][0] + X[1][2] * Y[2][0]$$

$$Z[1][1] = X[1][0] * Y[0][1] + X[1][1] * Y[1][1] + X[1][2] * Y[2][1]$$

And so on, the following program will do the addition of two matrices of size 3*3.

```c
#include<stdio.h>
void main()
{
  int x[3][3], y[3][3], z[3][3];
  int i, j, k;
  printf("Enter values for Matrix X:\n");
  for(i=0; i<3; i++)
  {
    for(j=0;j<3; j++)
    {
      printf("Enter Value:");
      scanf("%d",&x[i][j]);
    }
  }
   printf("Enter values for Matrix Y:\n");
  for(i=0; i<3; i++)
  {
    for(j=0;j<3; j++)
    {
      printf("Enter Value:");
      scanf("%d",&y[i][j]);
      z[i][j]=0;
    }
  }
  /* Multiplication of Matrices */
  for(i=0; i<3; i++)
  {
    for(j=0;j<3; j++)
    {
      for(k=0; k<3; k++)
      {
        z[i][j]=z[i][j]+x[i][k]*y[k][j];
      }
    }
  }
  printf("\nMatrix X:\n");
  for(i=0; i<3; i++)
  {
```

```c
      for(j=0;j<3; j++)
      {
         printf("%d\t", x[i][j]);
      }
      printf("\n");
   }
   printf("\nMatrix Y:\n");
   for(i=0; i<3; i++)
   {
      for(j=0;j<3; j++)
      {
         printf("%d\t", y[i][j]);
      }
      printf("\n");
   }
   printf("\nMatrix Z:\n");
   for(i=0; i<3; i++)
   {
      for(j=0;j<3; j++)
      {
         printf("%d\t", z[i][j]);
      }
      printf("\n");
   }
}
```

Output:
Enter values for Matrix X:
Enter Value:1
Enter Value:2
Enter Value:3
Enter Value:4
Enter Value:5
Enter Value:6
Enter Value:7
Enter Value:8
Enter Value:9
Enter values for Matrix Y:
Enter Value:11
Enter Value:12
Enter Value:13
Enter Value:14
Enter Value:15
Enter Value:16

Enter Value:17

Enter Value:18

Enter Value:19

Matrix X:

1	2	3
4	5	6
7	8	9

Matrix Y:

11	12	13
14	15	16
17	18	19

Matrix Z:

90	96	102
216	231	246
342	366	390

Check Your Progress-3

1. In _____ operation of matrix, we interchange the rows and columns of the matrix.

[A] Transpose   [C] Addition

[B] Multiplication   [D] None of the above

2. How many for loops (nested) is used, to print a Matrix.

[A] 4   [C] 3

[B] 2   [D] 7

3. How many for loops (nested) is used, to multiply 2 Matrices.

[A] 4   [C] 3

[B] 2   [D] 7

3.5   SPARCE MATRICES:

Sparse matrix is a matrix that has many elements with a value 0. In order to effectively utilize the memory, special algorithms and data structures developed which can store the sparse matrices and also permits operations like addition and multiplication of two sparse matrices.

In many applications we need to preserve matrices for either data processing or in the form of information (output) having most elements are 0. It is not good idea to waste costly memory space for storing those data-elements which don't deserve a place (having value 0). If many data-elements from a matrix have a value o then the matrix is called **sparse matrix**.

There is no clear definition is there, whether a matrix is a sparse matric or not. It is just a concept. But if it is to be feel that the matrix is a sparse matrix then it should be converted so that we can represent the entire matrix information in another way, such that it reduces memory space.

Consider the following 7*7 matrix. By looking to the matrix, you will come to know that the most elements of the matrix are 0 and hence it is sparce matrix.

	0	1	2	3	4	5	6
0	0	0	0	25	0	0	0
1	0	20	0	0	0	0	35
2	0	0	0	0	45	0	0
3	0	15	0	10	0	0	0
4	5	0	10	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	40	0	0	0	0

To store the matrix given above, if declare an array 'int x [7][7]', then it will occupy 98 Bytes of space as there are 7*7=49 data-elements are there and each data-elements are of type int which occupies 2 bytes of space in the memory as per ANSI C standard.

Now suppose instead of storing the array straightaway in the computer memory by using conventional method, if we store the following details then?

int sparce_mat [10][3] = {   7, 7, 9,

                    0, 3, 25,

                    1, 1, 20,

                    1, 6, 35,

                    2, 4, 45,

                    3, 1, 15,

                    3, 3, 10,

                    4, 0, 5,

                    4, 2, 10,

                    6, 2, 40 };

In this matrix representation, first triplet is 7, 7, 9 which indicates we have a matrix or 7 rows,7 columns and it has 9 non-zero element. Second triplet 0, 3, 25 indicate on row 0 and column 3 non zero data-element 25 is there. Third triplet 1, 1, 20 represent non zero data element 20 is at row 3 and column 3 and so on.

In this representation, we are not storing those data-element which has value 0. This representation gives information only about non-zero data-element. All those elements which are not there in this representation will be assume to 0. For example, in this representation we do not have any triplet for row 4 and column 3 that means it is 0.

In this representation we have taken an array of size 10*3=30. Therefore, it will occupy 30 * 2 = 60 Bytes of memory without loosing any kind of data (Remember the original sparce matrix required 98 Bytes of memory).

Check Your Progress-4

1. A 2-Dimensional array, with most elements are 0 in it, is called _____ matrix.

[A] Transpose   [C] Sparse

[B] Symmetric   [D] Asymmetric

2. In the array representation of sparce matrix first row represents _____.

[A] Row, Col, Data

[B] Number of Rows, Number of Columns and Number of Non-zero data elements

[C] Index number of row, Index number of column, non-zero data element value

[D] None of the above

3.6   LET US SUM UP

In this unit, we:

- Discussed Merging, Searching and Sorting operations on Array
- Elaborated 2-Dimesional arrays
- Explained operations on 2-Dimesional arrays
- Discussed Sparse matrices.

3.7   SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

1. [C] Sorting
2. [B] Merging
3. [A] Searching

Check Your Progress-2

1. [B] 2-Dimensional
2. [C] 20
3. [B] 2-Dimensional

Check Your Progress-3

1. [A] Transpose
2. [B] 2
3. [C] 3

Check Your Progress-4

1. [C] Sparse
2. [C] Number of Rows, Number of Columns and Number of Non-zero data elements

3.8 GLOSSARY

1. **Matrix** is a 2-dimessional array in which first index represent row and second index represent column
2. **Sparse Matrix** is a matrix representation of data, where most data-elements are 0.
3. **Linear search is** a method of searching where we match search element with all data-elements of an array.
4. **Sorting** is a is a process of arranging data-element into some order. Order can be ascending (lower to higher) or descending (higher to lower).

3.9 Assignment

1. What is Sparce Matrix? Explain in brief.
2. How can we represent matrix in a programming language? Explain it with an example.
3. List and explain different operations can be performed on an array.

3.10 Activity

Write a program to do following operations in array:
1. Represent sparse matrix
2. Adding 2 matrices of size 3*3
3. Multiplication of 2 matrices of size 3*3

3.11 Further Reading

- Data Structure through C by Yashvant Kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.
- Data Structures Using C, By Reema Thareja, From Oxford Publications.

Block Summary

- The Data Structures has set of algorithms, which are used in many important applications.

- Data Structures can be divided into two parts: [1] Linear data structures and [2] Non-Linear data structures. Array, Stack, Queue are linear data-structures, where Tree and Graph are examples of non-linear data-structures.

- Data types can be primitive, derived, abstract or user-defined.

- Primitive data types are those which built-into the system such as int, float, char, double etc.

- Derived data types are those which are derived from the basic (Primitive) data types like pointer, structure, union, class etc.

- User-Define data types are those which allows programmer to define his/her own data type like structures, union or class.

- Abstract data types are mathematical model, which are abstract into some logic. For example, to implement a program to convert Infix expression into Postfix, we are using stack as an abstract data type, where its methods push and pop are abstract methods.

- Data structures are implementation of Abstract Data-Types.

- Array is an ordered collection of homogeneous (same type of) data. It is a Linear data-structure.

- Array can be one-dimensional and two-dimensional.

- Traversal, Insertion, Deletion, Searching, Sorting, Merging are the operations which can be performed on one-dimensional array.

- Matrix can be represented in the form of 2-Dimesional arrays.

- Representation of Matrices, Addition and Multiplication of Matrices are operations of 2-Dimesional arrays.

- A matrix (2-dimesional array) having most data-elements are 0 is called a Sparse Matrix.

Dr. Babasaheb Ambedkar Open University

BSCIT-203

Data Structure

BLOCK2:   STACK, QUEUES AND LINKED LISTS

BLOCK 2: STACK, QUEUES AND LINKED LISTS

Block Introduction

We may need data structures which can be able to store the data in non-contiguous manner into the memory. As you know array finite set of homogeneous collection of the data-element stored in contiguous manner in the computer memory. Similarly, Linked Lists stores multiple data but in non-contiguous manner in the computer memory.

A stack is a linear data structure in which elements can be inserted or deleted through the same end called as the top of the stack. e.g., a stack of coins where elements can be inserted through the top.

A queue is a linear data structure in which elements can be inserted from rear end and deleted through the front end.

Block Objective

After learning this Block, you will be able to:
- Implement the Linked List
- Perform different operations on Linked Lists
- Understand the different types of Linked Lists
- Understand the Concept of Stacks
- Understand the Method of Representation of Stacks
- Understand the Applications of Stacks
- Implement Stacks
- Understand Basic operations performed on queue
- Understand Array and Linked List representation of queue
- Understand D-Queue
- Understand Circular queue
- Know application of queue Understand the concept of Linked Lists

Block Structure

Unit 1: Linked Lists

1

Unit Structure

1.0   LEARNING OBJECTIVES

In this unit, we will discuss different functions of dynamic memory allocation and Linked Lists:

After learning this unit, you will be able to understand:

- What is Link-List?
- How to implement Link-List?
- How can we implement different operations on Link-List?
- Different types of Link-Lists.
- Applications of Link-List.

1.1   INTRODUCTION

As you know that implementation of an array, can store multiple data-elements of the similar type under one name. Generally, we are declaring array using static method like 'int x [10]'. The problem within this approach is, you should know the exact size of an array in prior to its implementation. However, dynamic approach enables you to create an array dynamically at run time. Therefore, even if, the size of the array is unknown to you, still you can take the size from the user while program is running and create the array. To do this you have to learn, different functions of dynamic memory allocation, provided by C-Language. Therefore, in this unit we are going to discuss different functions of dynamic memory allocation and then we will discuss how can we implement Link-List programmatically.

1.2   DYNAMIC MEMORY ALLOCATION FUNCTIONS

Variables are of two types: [1] Static variables and [2] Dynamic variables. In the case of static variables, you have to declare the variables in the declaration segment of any function. C-Language is not allowing you to declare the variables after writing any executable statement. When you execute the program, compliers of the C-Language will read all declarative instructions and make preparation to occupy memory space for all of your variables into the main memory of the system. Once the memory space is allocated to all variables then program will start its execution. That means, when program starts its execution, before that it reserves the memory space for all variables declared in that particular function.

Dynamic memory allocation on the other hand, enables you to declare the variable whenever it is actually needed. It is not compulsory that you need to declare all variables in prior and then you can write executable statements. In dynamic memory allocation, program will start its execution, and it declare the variables when it is actually required, or when user is instructing to do so (at runtime).

1.2.1 malloc () function:

The malloc() function enables you to declare one variable (of required type) dynamically. In order to use the malloc() function, you need to include header file "alloc.h" if you are using Turbo-C or Borland-C software. If you are using Code-Blocks the you need to include header file "stdlib.h".

Syntax:

PTR_Variable = (Data Type *) malloc (size);

For example, if you need to declare a variable of type 'int' then you need to write:

Int_ptr = (int *) malloc (sizeof (int));

The following program guide you to declare an integer variable dynamically using malloc () function:

```
#include<stdio.h>
#include<stdlib.h> // Turbo-C and Borland-C users include alloc.h
void main ()
{
    int *p;
    p= (int *) malloc (sizeof (int) );
    *p = 5;
     printf ("The value stored in Dynamic variable is %d", *p);
}
```

Dynamic memory allocation declares the variable at runtime; therefore, they do not have variable name. So, function malloc() always returns the address of the memory location where it is actually created. In the above example, we are type casting the malloc() function to occupy 'sizeof(int)', means 2 bytes of space in the main memory. The function malloc(), will find unusable space in the main memory and occupies 2 bytes of space. The address of this memory space will be return back by malloc () function, which we are preserving in the pointer variable 'p' (as we know, pointer are those variables which stores the addresses of another variables). Now, by using the pointer variable 'p', we can store data value 5 and print it on the console screen. If malloc() function fails to occupy memory space due to some

reason (if free space is not available because of main memory is full or any other reason), instead of memory address malloc() function returns NULL value to the pointer variable.

1.2.2 calloc () function:

As function malloc() is used to create or declare any type of single variable, calloc () function enables you to create or declare an array of any type (dynamically), where multiple values you can store.

Syntax:

PTR_Variable = (Data Type *) calloc (number_of_elements, size_of_element);

For example, if you need to declare an array of type 'float' with size 10, then you need to write:

Int_ptr = (float *) calloc (10, sizeof (float));

The following program will teach you to create an array of integer values dynamically using calloc () function:

```
#include<stdio.h>
#include<stdlib.h> // Turbo-C and Borland-C users include alloc.h
void main ()
{
  int i, num, *p;
  printf ("Enter Number of Elements:");
  scanf ("%d", &num);
  p = (int *) calloc (num, sizeof (int));
  for (i=0; i<num; i++)
  {
    printf ("Enter Number:");
    scanf ("%d", &p[i]);
  }
  printf ("You have Entered:");
  for (i=0; i<num; i++)
    printf ("\n%d", p[i]);
}
```

The program shown above will allow user to create an array as per user's requirement. If user want to create an array of 5 elements, then user need to enter 5 while the program will ask for "Enter Number of Elements:". The function calloc() will now declare the array of 5 elements of type int. Here, calloc () function enables user to create the array as per user's requirement, at runtime (dynamically). We accept 5 integer numbers from the user and store them into the dynamically created array using for loop, and lastly we are printing all 5 data-elements on the console screen using other for loop.

1.2.3 free () function:

The free () function is used to destroy or delete a variable created by malloc() or calloc() function. Consider the program give below, in which we are declaring an integer variable at run-time using malloc () function. We store a value in it, and lastly after printing the value of dynamically declared variable on the console, we are deleting it using free() function. Make sure, once the variable is destroyed or deleted using free() function, cannot be used in the program, as it don't exists in the computer memory.

For Example:

```
#include<stdio.h>
#include<stdlib.h> // Turbo C and Borland C users include alloc.h

void main ()

{

    int *p;
    p = (int *) malloc (sizeof (int));
    *p = 5;
    printf ("The value stored is %d", *p);
    free(p); // Dynamic variable is destroyed here

}
```

Check Your Progress-1

1.Which dynamic memory allocation function need to use, to create one variable at run-time?

[A] calloc ()   [C] malloc ()

[B] realloc ()	[D] None of the above

2. To create an array, _____ function from the given below options should be used.

[A] calloc ()	[C] malloc ()
[B] realloc ()	[D] None of the above

3.To delete a variable, created by function malloc(), _____ function is used.

[A] clear ()	[C] delete ()
[B] free ()	[D] All of the above

1.3 LINK-LISTS

We know that the array is a collection of homogeneous data. Similarly, to arrays, Link-Lists are also collection of homogenous data. The difference between the Array and Link-Lists is, Array stores all data-elements continuously in the memory, where Link-List stores all data-elements randomly (non-contiguous manner). For Example, if you declare an array 'int x [10];', then program requests 20 bytes of continuous space in the memory. Suppose, if the computer memory has 20 Bytes of space, but not continuously, then array will not be created and system will give you error of 'Memory Overflow'. But in the case of Link-List, we can store our data if memory has sufficient space to accommodate all data-elements either in contiguous or non-contiguous manner.

A Link-List is also set of finite data-elements of similar type, stored in the data structure called nodes. These nodes are divided into two parts: [1] Data part and [2] Address part. The data part of the node preserves the data to be stored in the Link-List and the address part of the node preserves the address of the next node.

You can understand the same with the help of the given example. Suppose, a teacher takes her class students for an educational movie and the total no. of students in the class are 20. Now, the theatre has given random seats to students (not in continuous order), in this case it will be a problematic job for the teacher to collect the students from the movie theatre after the show ends. Now, what the teacher will do? Instead of memorizing seat numbers of all students (as it is difficult), she will memorize the seat number of only one student. That student also memorizing a seat number of another student (may be his friend). His friend also in-term memorizing a seat number of some another student.

Here you can assume that the seat numbers are nothing but the memory address. Teacher is memorizing (storing) the address on first student. First student will give a seat number (address) of another student. And in that way, a student who don't have any seat number further is a last student. In this example, Teacher can gather all students properly, by memorizing (storing) the address of only one student.

In the same way, in Link-Lists, all the nodes are not in sequential order in the memory, they are randomly deployed and each node preserves the address for the next node which makes process easier to retrieve all Link-List data in an easier way.

1.3.1 Node Structure:

Data structure none is a fundamental part of Link-List, which consists of two sections. First section is data which responsible to store the data given by the user. Second section of the node, preserves the address of the next node. Refer the following figure [Fig:1.1] of node representation. In the figure node represents two sections. Suppose, if we consider, user is going to store some integer numbers then for the first section of the structure node, we declare 'int data' and in the second section, we need to preserve the address of the next node, therefore the second variable in the structure node will 'struct node *next'.
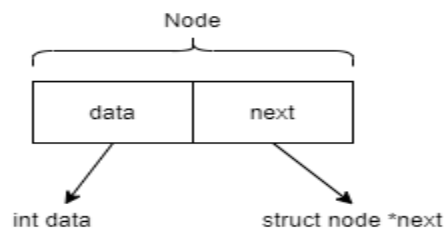
[Fig: 1.1 Representation of Structure Node in the Link-List]

1.3.2 Linked List Representations
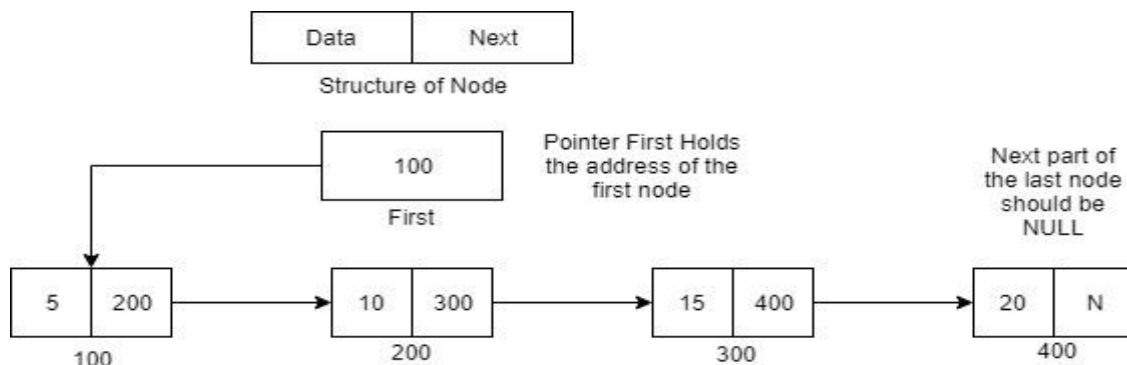
[Fig: 1.2 Representation of the Link-List]

As shown in the figure [Fig:1.2] given above, pointer variable 'First' is a global variable which is declared to preserve the address of the first node. Pointer 'First' has a value 100, which is the hypothetical address of the first node. First node, has 5 in the data section and 200 in the address section next, which is again hypothetical address of the second node. As shown in the figure, Link-List has 4 nodes. The last node has value 20 and because of it is last node, NULL value is stored in its next (address) section. In the given figure 100, 200, 300 and 400 are the hypothetical addresses of 4 nodes added in the Link-List.

Check Your Progress-2

1.In a singly Link-List each node has _____ parts.

[A] 2   [B] 3

[C] 4   [D] 5

2. In a singly Link-List each node has an address of _____ node.

[A] First   [B] Last

[C] Previous   [D] Next

1.3.3 Defining structure node:

As we know that a node is a rudimentary building block of data structure Link-List. Structure node consists of two sections: [1] Data: where we will preserve the user's data and [2] Next: where we will put the address of the next node. Because of we are storing the address in the Next part, it must be declared as pointer variable of type struct node. In short, we can define structure node as shown below:
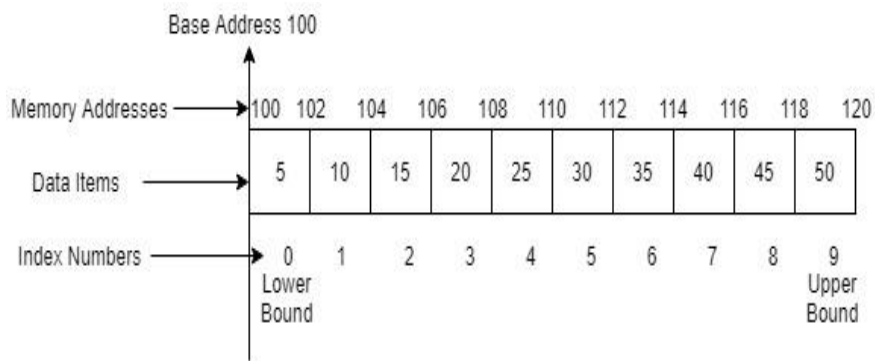
struct node

{

    int data;
   *struct node *next;*

};

1.3.4 Difference in Array and Link-List data structures:

[Fig: 1.3 Array representation in the Memory]

In the figure [Fig:1.3] give above, we have demonstrated an array data structure. An array is homogeneous (same type of) collection of data, where all the data-elements of an array are stored in the adjoining manner in the memory. When we declare 'int x [10];', then system search for 10 (total number of elements) * 2 (each int data elements desires 2 Bytes) = 20 Bytes, of contiguous memory space into the main memory. Suppose, system finds contiguous 20 Bytes of adjoining memory space at some address 100, then address 100 will becomes a base address for that array. First element of an array which is 5, will be stored between memory locations 100 to 102. Second data-element 10, will be stored from memory addresses 102 to 104 and so on. Name of the array 'x' will indicate the starting address (base address) of an array.
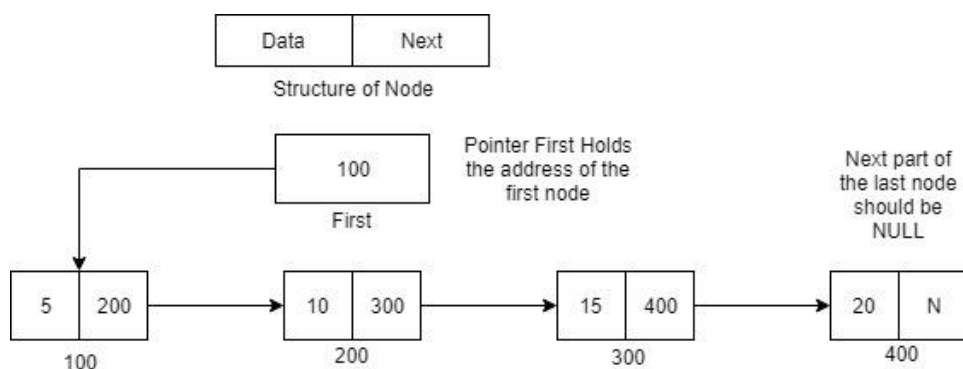
[Fig: 1.4 Link-List representation in the Memory]

Link-List also a collection of similar type of data like array, but in the case of Link-List the data-elements are stored in non-contiguous manner. Each data-element is compressed into the data structure called node. Each node of the Link-List has data and address to the next node, which represent another node. As shown in the figure [Fig:1.4] data elements 5 and 10 are stored in two different nodes and their addresses are non- contiguous (100 and 200). Each

node has an address field which stores address to the next node (which contains next data element).

Check Your Progress-3

1._____ is a Linear data structure stores data-element in non-contiguous manner.

[A] Array   [C] Link-List

[B] Tree   [D] None of the above

2. From the given below _____ is/are Linear data structure(s).

[A] Array   [C] Link-List

[B] Stack   [D] All of the above

1.4 LINKED LIST IMPLEMENTATION

1.4.1 Declaration of node and first pointer:

As we know, Link-List is a linear data structure, can be stored in non-contiguous memory locations. Link-List is a collection of nodes, which encapsulate data element and address of the next node.

struct node

{

    int data;

    struct node *next;

} * first= NULL;

Pointer first is variable can store address of any node. Pointer first holds the address of the first node of the Link-List. Initially, when the program is started, and when there is no node is there in the Link-List (Link-List is empty) first pointer should not have any address, that is the reason, we have initialized it with NULL value. When user add the first value, and first node is created to encapsulate that value in the Link-List, *first will have the address of the first node.

1.4.2 Creating Link-List:

To create a Link-List, user will invoke a function create(). In this function, we will take how many nodes has to be inserted in the Link-List, from the user. Suppose, if user enters 6, then function will take 6 values from the user and add 6 nodes in the Link-List. Each node of the Link-List contains one value entered by the user. Each node also has the address of the next node. The next part of the 6th node (last node) will be NULL and the address of the first node will be placed in the *first.

```
void create ()
{
  struct node *new_node;
  int i, n;
  printf ("\nEnter Number of Elements you want to insert in the Link-List:");
  scanf ("%d", &n);
  for (i=1; i<=n; i++)
  {
    if (first==NULL)
    {
      new_node = (struct node *) malloc (sizeof (struct node));
      first = new_node;
    }
    else
    {
      new_node->next = (struct node *) malloc (sizeof (struct node));
      new_node = new_node->next;
    }
    printf ("Enter Data-Value:");
    scanf ("%d", &new_node->data);
    new_node->next=NULL;
  }
}
```

1.4.3 Inserting a value to the Link-List:

Insertion can be done at both sides of the Link-List. To insert the value in the beginning of the Link-List, we will design a function called insert_at_beg() and to insert the value at the end of the Link-List, we will design another function called insert_at_end().

[1] Insertion at the beginning of the Link-List:

To insert the new value, at the beginning of the Link-List, we will accept a value from the user. We will now create a new node and set the value accepted from the user in the data part of the newly created node. We will copy the address of the first node from the *first. Finally, the address of the newly created node we will be replaced in the *first pointer.
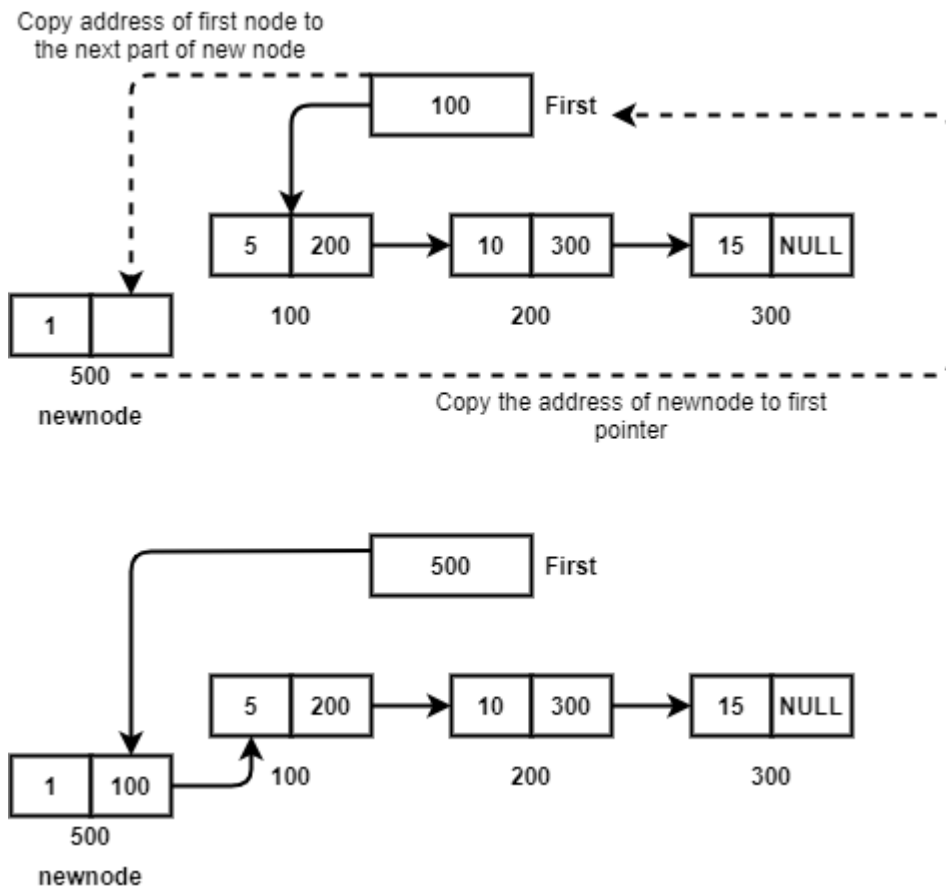
[Fig: 1.5 Process of inserting a new value at the beginning of the Link-List]

Suppose, in the Link-List, three nodes are inserted, by create() function. These values are 5, 10 and 15. So there are three nodes are there in the Link-List. Now, suppose user needs to insert a new node with value 1, at the beginning of the Link-List. To do this, first we will create a new node using function malloc(). Suppose, the new node is created on memory address 500. Store 1 (value given by user) in the data part of new node, copy the address 100 from the *first to the next part of new node. so, the new node will point to the node '5'. Copy the address of new node (that is 500) in the *first. Therefore, first will provides you the address 500 (first node) and the next part of the first node will gives you the address 100, which will become second node.

```
void insert_at_beg (int val)
{
    struct node *new_node;
    new_node = (struct node *) malloc (sizeof (struct node));
    new_node->data=val;
    new_node->next=NULL;
```

```
    if (first == NULL)
    {
      printf ("\nLink-List is empty");
      return;
    }
    new_node->next = first;
    first=new_node;
}
```

[2] Inserting value at the end of the Link List:

To insert the new node at the end of the Link-List, we will take a provisional pointer to search the last node in an existing Link-List. We need to create a new node, and store the data entered by the user in the data section of the new node. We also need to set the next section of the newly created node to NULL. Then we need to start searching of a last node. To identify last node, we will search that node, in which next section of the node should have NULL value. When we reach at the last node, we will copy the address of the new node in the next section of the last node.
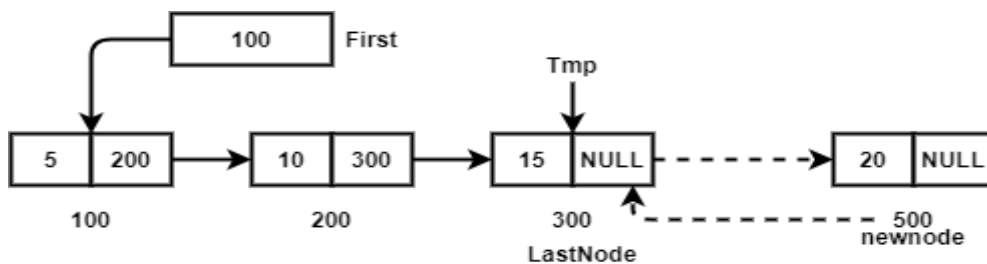
[Fig:1.6 Inserting a new node at the end of the Link-List]

```
void insert_at_end (int val)
{
    struct node *new_node,*tmp;
    new_node = (struct node *) malloc (sizeof (struct node));
    new_node->data = val;
    new_node->next = NULL;

    tmp = first; //Setting tmp pointer on the First Node
    while (tmp->next != NULL) // Searching for Last Node
    {
      tmp = tmp->next; //Moving pointer to the next node till we reach at Last Node
    }
//Now, Putting the address of newly created node to next section of last node
    tmp->next = new_node;
}
```

1.4.4 Displaying Link-List:

To display the Link-List, take the provisional pointer 'tmp' and put it on the first node. Make sure, first pointer should not have NULL value. If *first has NULL value, which means Link-List is empty and nothing has to be displayed, just return from the function. Else print the value from the data section and move, *tmp to the second node, by copying new address from the next section of 'tmp' *tmp. Continue this process till *tmp will not be NULL.

```
void display ()
{
    struct node *tmp;
    if (first == NULL)
    {
        printf ("Link-List is empty:");
        return;
    }
    tmp = first;  //Setting *tmp on the first node of Link-List
    while (tmp != NULL) // Loop will be repeated till last node is not printed
    {
        printf("\n%d -->",tmp->data); //Printing data of the node
        tmp = tmp->next; //Moving *tmp towards next node
    }
}
```

1.4.5 Deleting a value from the Link-List:

For deleting a value from the Link-List, we need to evaluate multiple conditions. First need to check if, pointer first is NULL or not. If it is then Link-List is empty and deletion is not possible, so simply come out (return) from the function.

We also need to check whether user wants to delete the first node or not. If yes, then we need to copy the address of the second node in the *first, and then we need to delete the first node. Otherwise, we need to find the element in the Link-List. If the value to be deleted is exists in the Link-List, then we need to copy the address placed in the next section of that node, to the next section of its previous node. If the value is to be deleted is not exists, then appropriate message is shown that the value does not exists in the Link-List.
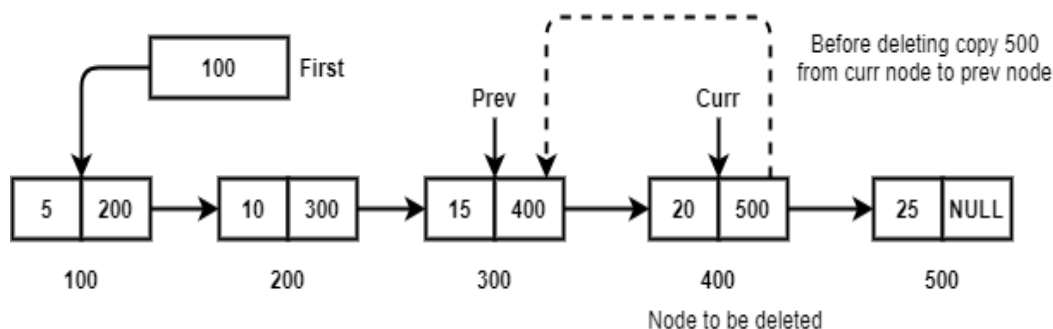
[Fig:1.7 Inserting a new node at the end of the Link-List]

66

For deleting a node, we use a pointer 'tmp' to search the node to be deleted. We place a 'tmp' pointer on the first node of Link-List, and start finding that node in which data section has a value which user want to delete from the Link-List. When we locate the value, we have to copy address of next node (500 in the figure [Fig:1.7], which is in the next section of the 'tmp' node) to the next section of its previous node.

In a singly Link-List, each node has address to its successor node. So, we should not have access to the previous node. To access the previous node, we need to take additional pointer variable called 'previous'. The *previous will follow the *tmp variable. That means, when the *tmp will reach to the address 400, at that time, *previous should be on its previous node that is 300 in the figure [Fig:1.7]. We should now copy the address 500 which is in the next section of the tmp node, to the next section of the previous node by just writing an instruction:

previous->next = tmp ->next;

This is one of the limitation of singly Link-List. In singly Link-List, each node has address to its successor node, but no node has address to its predecessor node. So, whenever we have to access the previous node, we need to use an extra pointer variable. To implement deletion in the singly Link-List, we have designed a function called 'delnode()'. User will invoke 'delnode()' function and pass the value to be deleted to this function as an argument. The code to implement delnode () function is explained below:

```
void delnode (int val)
{
  struct node *tmp, *previous;
  //If Link list is Empty then nothing to be deleted
  if (first == NULL)
  {
    printf ("\n Link-List is empty");
    return;
  }
  //Place tmp pointer to the fist node
  tmp = first;
  //If value to be deleted is found in the first node
  if (first->data == val)
  {
    first = first->next;
    free (tmp);
    return;
  }
```

```
//Search the value in the Link-List
   while (tmp != NULL && tmp->data != val)
     {
     //Previous pointer is following to the current pointer
     previous = tmp;
     tmp = tmp->next;
   }

//If value to deleted is not found
   if (tmp == NULL)
     {
      printf ("\n Value to be Deleted does not exist");
      return;
     }
   //Copying the address of next to the previous node's next part
   previous ->next = tmp->next;
   //Deleting node
   free (tmp);
}
```

At the end, when we search the particular node in which the value of data section matched with the value to be deleted (given by the user, and passed as an argument to the delnode() function), we will use function free() to delete that node (which is pointing by *tmp). To delete the desire node, we need to give following instruction:

free (tmp);

Function free() will delete the node structure, specified on the address stored in the *tmp variable.

After implementing this function, note down your observation about following:

1. Try to delete any value which is in the middle place of the Link-List,
2. Delete a value which is the first node,
3. Try to delete that value which is no there in the Link-List and also,
4. Try to delete that node which is a last node of the Link-List.

Check Your Progress-5

1. In which function of the singly Link-List, we need to take an extra pointer variable to access previous node.

[A] delnode ()   [C] display ()

[B] create ()	[D] insert_at_end ()

2. _____ dynamic memory allocation function is used to delete the node from the Link-List.

[A] delete ()	[C] clear ()
[B] erase ()	[D] free ()

1.5 LET US SUM UP

In this unit, we:

- Discussed important functions of dynamic memory allocation
- Elaborated on the basic structure of a Link-List.
- Talked about How to implement a Link-List?
- Explained How to program functions like insert at end, insert at beginning, delete node, and display functions.

1.6 SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

4. [C] malloc ()
5. [A] calloc ()
6. [B] free ()

Check Your Progress-2

4. [A] 2
5. [D] Next

Check Your Progress-3

4. [C] Link-List
5. [D] All of the above

Check Your Progress-4

1. [D] malloc ()
2. [B] NULL

1. [A] delnode ()
2. [B] free ()

1.7 GLOSSARY

5. **Pointer:** Pointer is a special variable, which is used to store the address of another variable.
6. **Link-List:** Link-List is a linear data-structure, which stores (collection) homogeneous (same type of) data in non-contiguous memory locations.

1.8 Assignment

3. Implement a function called 'insbfr (int val, int key)', which will insert a new node to the Link-List before the key value.
4. Implement a function called 'insaft (int val, int key)', which will insert a new node to the Link-List after the key value.

1.9 Activity

1. Write a function to sort a Link-List.

1.10 Case Study

1. Write a function to reverse all the nodes of the Link-List using recursion.

1.11 Further Reading

- Data Structure through C by Yashvant kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.

Unit 2: More on Linked Listsists ◼ 2

Unit Structure

2.0   LEARNING OBJECTIVES

After working through this unit, you should be able to:

- Different types of Linked-Lists
- Understand doubly Link-List
- Implementation of the program of Link-List

2.1   INTRODUCTION

In the previous unit, we have seen how can we implement Link-List, what are the advantages are there and what are the similarities and differences are there between Array and Link-List. In this Unit we will discuss different types of Linked-Lists and their implementation.

2.2   TYPES OF LINKED-LISTS

The Link-List we have discussed in the previous unit, is a singly link list. There are three types of link lists are there:
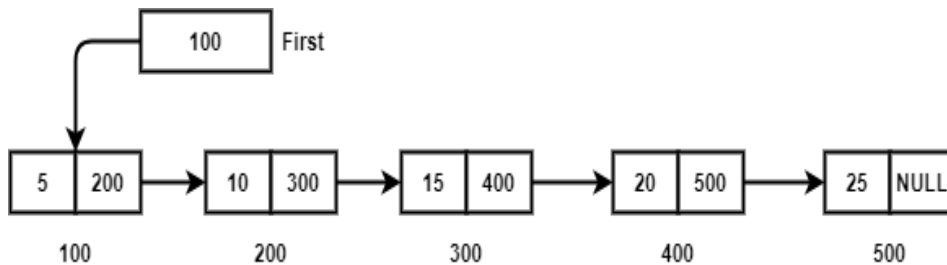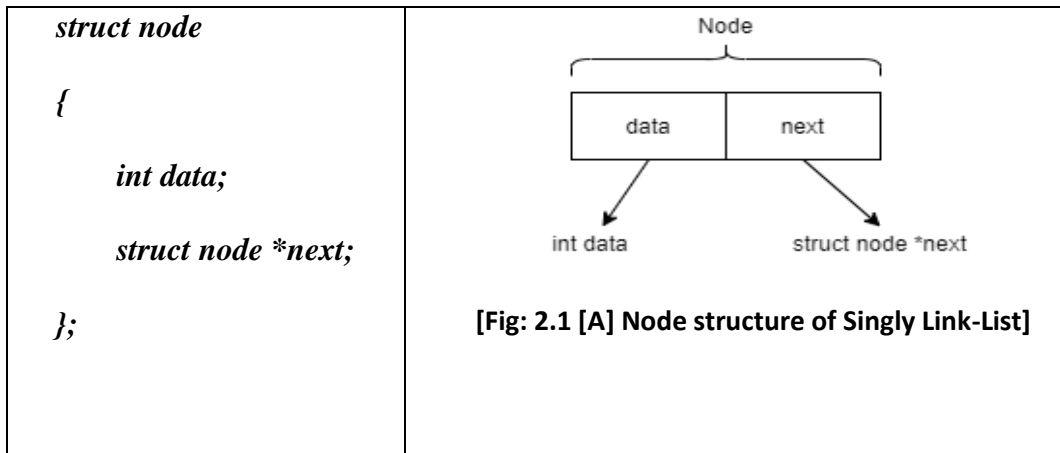
[1] Singly Link-List

[2] Doubly Link-List and,

[3] Circular Link-List.

2.2.1 Singly Link-List:

In the singly Link-List, each node of the Link-List has only one address, and that is the address of a next node. Because, in the structure node, there is single address is encapsulated, it is called a singly link list. In the singly link list, global pointer variable first, holds the address of the first node of the link list. The next (address) part of the link list, gives an address to the second node and so on. The next part of the Last node has NULL value, which indicates end of the link list. The structure of the singly link list is described below:
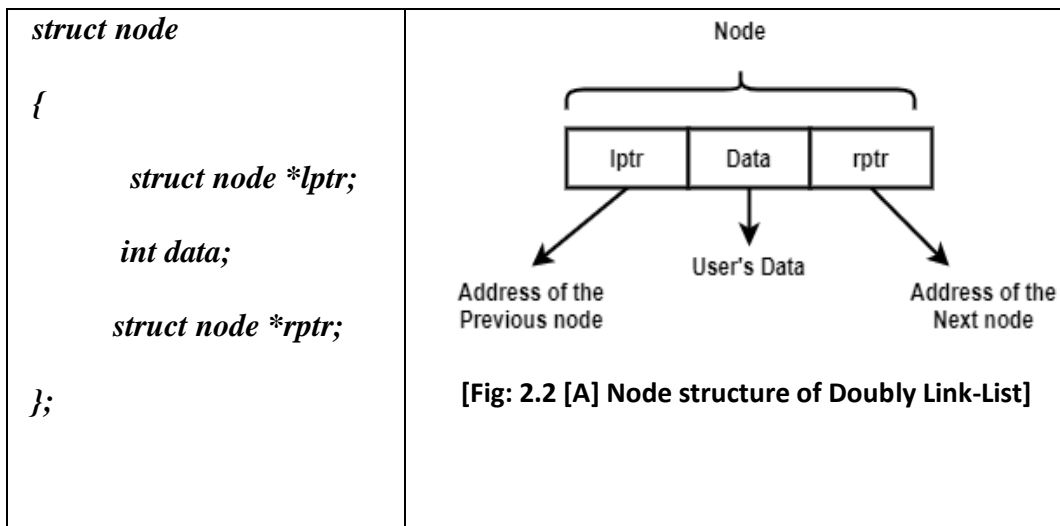
|	
struct node *{* 　　*int data;* 　　*struct node *next;* 　*};*	 **[Fig: 2.1 [A] Node structure of Singly Link-List]**

[Fig: 2.1 [B] Singly Link-List]

2.2.2 Doubly Link-List:

The problem with the singly link list is, it contains only one address part in each node of the link list in which we store the address of the next node. Due to this reason, we cannot have access to the previous node. Singly link list provides forward pathway, but it doesn't provide backward traversal compatibility. Recall the delnode () function, we have implemented in the previous unit. To access the node to be deleted we have taken an additional pointer variable 'prev', which was following the 'curr' pointer.

To overcome this problem, we have developed doubly link list. In the case of doubly link list, we modify the structure of node. In the node of doubly link list we will place one data and two address parts (total three elements). Each node hold user data, address on the next node and in addition address of the previous node too. That means in doubly link list, from any node you can visit next node as well as its previous node. It provides forward and backward compatibility. The structure of the doubly link list is shown below:

struct node
 	
{
 	
    struct node *lptr;
 	
    int data;
 	
    struct node *rptr;
 	
};
	
	[Fig: 2.2 [A] Node structure of Doubly Link-List]

As shown in the figure given above, node has three parts data, lptr and rptr. Data part of the node will store user's data, lptr and rptr are pointer variables which store address of the left node and right node. Means, each node stores two addresses, address of the previous node as well as address of the next node. User can visit the next node using address stored in rptr and also visit previous node using the address stored in lptr.
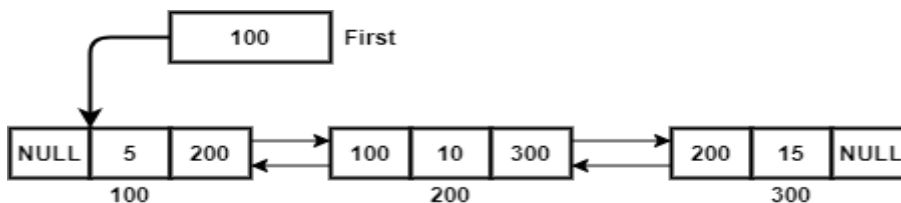
[Fig: 2.2 [B] Doubly Link-List]

Make sure in the lptr of first node, we will store NULL because there no node is there on left side of it, and in the same way we will also store NULL in the rptr of the last node as there is no node is three on the right side of the last node.

Check Your Progress-1

1. In _____ link-list each node has address for its next node only.

[A] Singly   [B] Doubly

[C] Both [A] and [B]   [D] None of the above

2. In _____ link-list each node has address for its next node and previous node too.

[A] Singly   [B] Doubly

[C] Circular   [D] All of the above

2.2.3 Circular Link-List:

Circular link list is similar to the singly link list. In the singly link list we store NULL value in the next part of the last node, so that we can come to know that this node is the last node and there no further node is there. Singly link list suffers from the problem that, it provides way to forward direction only. We can not visit the previous node. To overcome this problem, in the circular link list, we store the address of the first node, in the next part of the last node. This will create a cycle and user can visit first node after visiting last node.

Circular link list and Singly link list share the same data structure. The difference between circular and singly link list is, in a singly link list last node's next part will remain NULL whereas in the circular link list, last node's next part will store the address of the first node. The data structure and circular link list is shown in the figure given below:

struct node	
{	
int data;	
struct node *next;	
};	**[Fig: 2.3 [A] Node structure of Circular Link-List]**
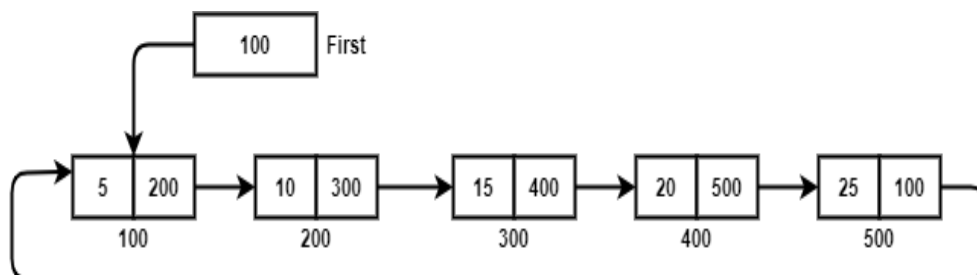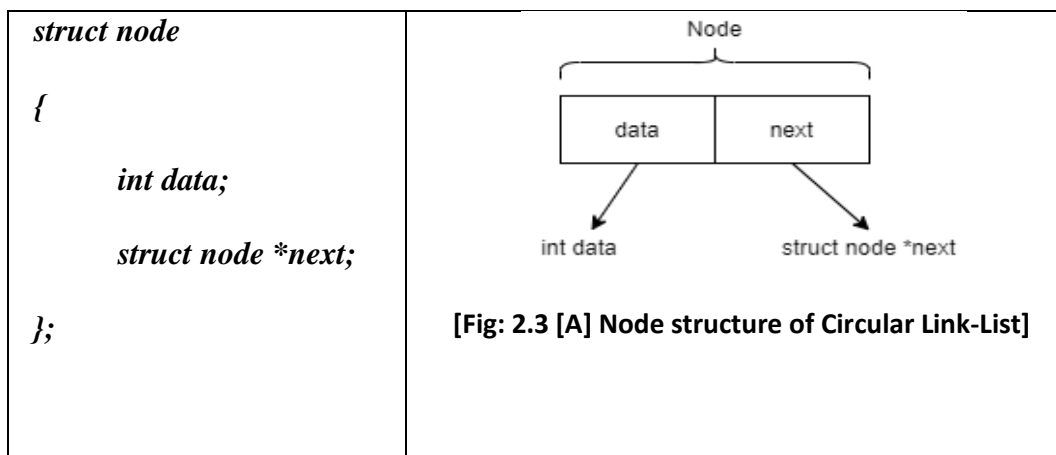
[Fig: 2.3 [B] Circular Link-List]

2.3   DOUBLY LINKED LIST IMPLEMENTATION

2.3.1 Declaration of node and first pointer:

We know that the doubly link list is a linear data structure, in which each node has two addresses. Pointer 'lptr' preserves the address of the previous node and pointer 'next' preserves the address of the next node. The data structure for the node of doubly link list is as given below:

struct node

{

*   int data;*

*   struct node *lptr, *rptr;*

*} * first= NULL;*

Pointer variable first is a global variable, which holds NULL value initially (when the program is started and there is no node is there in the doubly link list). Once the nodes are created in the doubly link list, first pointer will store the address of the first node.

2.3.2 Creating a doubly linked list:

To create a doubly linked list, we prompt to the user that 'How many values, user want to insert in the doubly link list'. Based on user input we will create that many nodes. We will take integer numbers from the user as use data and store it into each node of the doubly link list.

We set the address of the first node of the doubly link list into the first pointer variable. Each node, stores the address of the previous node into the lptr and address of the next node into the rptr. The lptr of the first node will be set to NULL and the rptr of the last node will also be set to NULL. The source code of the create () function, for the doubly link list is given below:

```
void create()
{
    struct node *tmp, *newnode;
    int val, i;
    /*Taking how many values user want to insert at the time of Creation */
    printf ("Enter number of Elements:");
    scanf ("%d", &val);
    for (i=0; i<val; i++)
    {
    /* Creating a new node in the memory */
        newnode = (struct node *) malloc (sizeof (struct node));
        printf ("Enter Value");
        scanf ("%d", &newnode->data);
        newnode->lptr = newnode->rptr = NULL;
    /* If first pointer is NULL then this the first value inserted by the user, and this is
    the first node      */
        if (first == NULL)
        {
            first = newnode;
            tmp = newnode;
        }
        else
        {
            newnode->lptr = tmp;
            tmp->rptr = newnode;
            tmp = newnode;
        }
    }
}
```
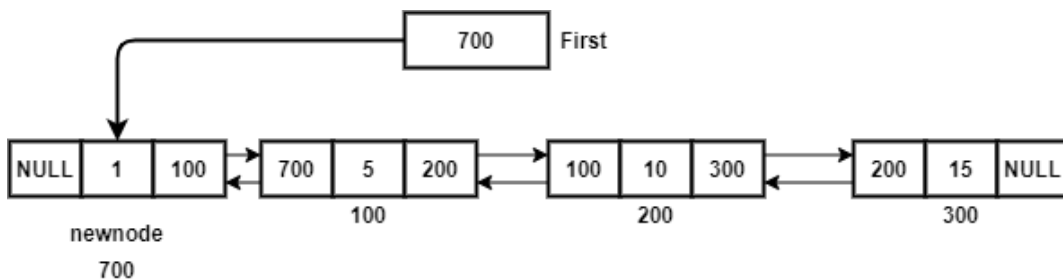
2.3.3 Inserting a value to the Link-List:

Like singly link list, insertion can be done at both ends. User can insert the value either to the beginning of the doubly link list, or end of the doubly link list.
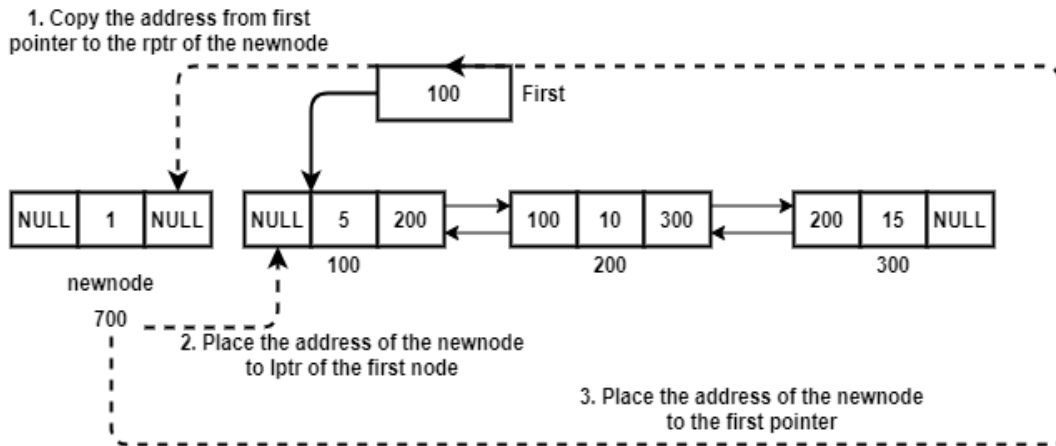
[Fig: 2.4 Inserting a value in Circular Link-List at Beginning of the List]

As shown in the above figure, to insert the value at the beginning of a doubly linked list, we will create a newnode using malloc () function. We will place the value inputted by the user into the data part, and NULL value in the lptr of the newnode. Copy the address of the first node, from the first pointer variable to the rptr of the newnode. Then copy the address of newnode to the lptr of the first node. Finally set the newnode as a first node, by copying the address of the newnode to the first pointer variable. At the end, you will get the following link list.

[Fig: 2.5 Inserting a value in Circular Link-List at Beginning of the List]

The code for the function inser_at_beg (), to insert the value to the beginning of the doubly linked is given below:

```
void inst_at_beg (int val)
{
struct node *newnode;
        if (first == NULL)
        {
                printf("Doubly Link List is empty");
                return;
        }
        newnode = (struct node *) malloc (sizeof (struct node));
```

78

```
        newnode->data = val;
        newnode->lptr = NULL;
        newnode->rptr = first;
        first->lptr = newnode;
        first = newnode;
}
```

[2] Inserting value at the end of the Doubly Link List:

To insert the value at the end of the doubly linked list, first we will create a new

We will then take a temporary variable tmp to search the last node of the doubly linked list. Last node of the link list is that node in which NULL is there in the rptr. After finding last node we will copy the address of the newnode to the rptr of the tmp node, and address of the tmp node to the lptr of the newnode as shown in the following figure.
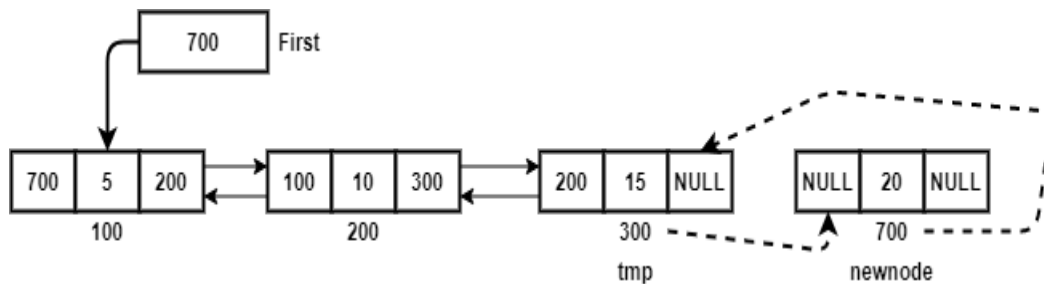
After the insertion, links list looks like as in the following figure:
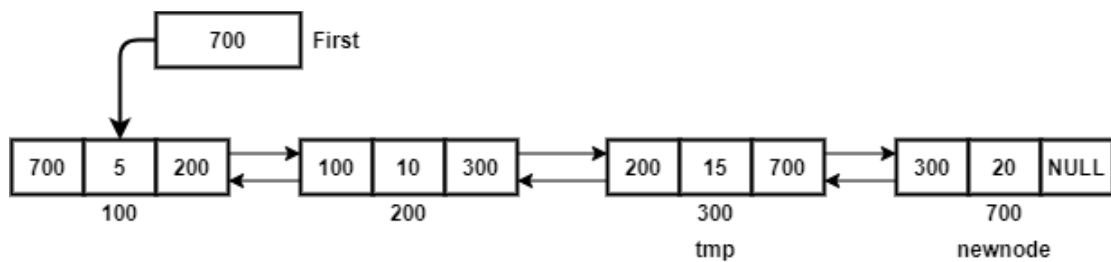
[Fig: 2.6 Inserting a value in Circular Link-List at the end of the List]

The following code we need to write to implement the function 'insert_at_end'.

```
void ins_at_end(int val)
{
    struct node *newnode, *tmp;
    newnode  =(struct node*) malloc (sizeof (struct node));
    newnode->data = val;
    newnode->rptr = NULL;
    if (first == NULL)
    {
        printf ("Doubly Link list is Empty");
        return;
```

```
    }
    tmp = first;
    while (tmp->rptr != NULL)
    {
       tmp = tmp->rptr;
    }
    if (tmp->rptr == NULL)
    {
    tmp->rptr = newnode;
    newnode->lptr = tmp;
    }
}
```

Check Your Progress-3

1. Identify the given data structure is of _____ link-list.

struct node

{

        int data; struct node *lptr, *rptr;

};

[A] Singly   [B] Doubly

[C] Circular   [D] All of the above

2. In doubly link-list, when the same node has NULL value in its lptr and rptr?

[A] Link-List is Empty   [B] Link-List has more than 2 nodes

[C] Link-List has only one node   [D] It is not possible

2.3.4 Displaying Doubly Link List:

Implementation of the display () function is similar to the display () function of singly link list. Just the difference is instead of next, we need to use rptr. The following code needs to be written to implement display () function to display all the values inserted by the user.

```
void display()
{
   struct node *curr;
/* If Link List is not created by user or there is no value in the Link List */
   If (first == NULL)
   {
      printf ("Doubly Link-List is empty");
      return;
   }
/* Setting current pointer to first node */
   curr = first;
/*Printing value and moving current pointer to the next node, till we reach to Last
node. */
   while (curr->rptr != NULL)
   {
      printf ("%d-->",curr->data);
      curr = curr->rptr;
   }
/* Printing the value of the Last node. */
   printf ("%d",curr->data);
}
```

2.3.5 Deleting a node from Doubly Linked List

To delete the node from a doubly linked list, we need to search for a node, in which data part has same value which user want to delete. To do this we need to take only one pointer 'curr'. Recall the delnode () function, of singly link-list, where we have taken two pointers 'curr' and 'prev'. The pointer 'prev' was following 'curr' pointer, as in singly link-list node do not store the address of previous node. In the case of doubly link-list each node has addresses of previous node and next node. So, there is no reason to take additional pointer called 'prev'.

To search the value, we will palace 'curr' pointer on the first node, by copying the address of the first node from pointer first to curr. We will run a loop till the data part of the curr does not match to the value to be deleted or pointer 'curr' does not becomes NULL. If 'curr' contain NULL means, the value to be deleted is not present in the link-list. If the value to be deleted is exists then we adjust links and then by using free () function, we will delete the node.

To adjust the links, following code we need to write:

curr->lptr->rptr = curr->rptr;

curr->rptr->lptr=curr->lptr;

In the first statement we copy the address, stored in the rptr of the node to be deleted to the rptr of its previous node. In the second statement, we copy the address stored in the lptr of the curr node to the lprt of the next node.

Some conditions we need to check during deletion of the node. The conditions are: [1] If the link-list is Empty, then nothing to be deleted. [2] If user want to delete the first node, then second node will become first node and the address of the second node, has to be placed in the first pointer. [3] If the value to be deleted is not exists in the link-list. If this is the case then nothing to be deleted. [4] If user is deleting a last node.

The code given below, take care of all the conditions described above, and delete the node which user want to delete:

```
void delnode (int val)
{
   struct node *curr;
/*If first pointer has NULL value, no value is there in the Link List, so deletion is not
possible. */
   if (first == NULL)
   {
     printf ("Doubly Link List is empty:");
     return;
   }
}
```

```
/* Place the address of first node in the curr pointer */
   curr = first;
/* If user want to delete first node, then second node will become first node after
deletion */
   if (first->data == val)
   {
     first = first->rptr;
     free(curr);
     first->lptr = NULL;
     return;
   }
/* Searching the value to be deleted in the Doubly Link List */
   while ( curr != NULL && curr->data != val )
   {
     curr = curr->rptr;
   }
/* If User wants to delete the last node of the Doubly Link-List */
   if (curr->rptr == NULL)
   {
     curr->lptr->rptr=NULL;
```

```
    free(curr);
    return;
  }
/* If current pointer is NULL, that means the value we are searching is not present in
the list */
  if (curr == NULL)
  {
    printf("value doesnt exsist");
    return;
  }
/* If value found then we will set (1) Next node's address in to the rptr part of
Previous node and (2) Previous node's address in the lptr of Next node.Finally,
Deleting current node. */
curr->lptr->rptr=curr->rptr;
curr->rptr->lptr=curr->lptr;
free(curr);
}
```

Check Your Progress-4

1. In _____ function of doubly link list, we use only rptr and doesn't use lptr.

[A] create ()   [B] insert_at_end ()

[C] display ()   [D] delnode ()

2. In _____ link-list, we do not need extra pointer variable to keep track of previous
node in the delnode () function.

[A] Singly   [B] Doubly

[C] Circular   [D] All of the above

2.4   LET US SUM UP

In this unit, we:

- Have seen the different types of link lists.
- Have elaborated how to implement doubly linked list.

2.5 SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

1. [A] Singly
2. [A] Doubly

Check Your Progress-2

1. [D] NULL
2. [A] First node

Check Your Progress-3

1. [D] Doubly
2. [A] Link-List has only one node

Check Your Progress-4

1. [C] display ()
2. [C] Doubly

2.6 GLOSSARY

1. **Doubly Link List**- List is also a type of linked list, which is a collection of several nodes, in which each node is divided into three parts.
2. **Circular Link List:** It is a type of link list, in which last node stores the address of the first node instead on NULL value.

2.7 ASSIGNMENT

- List and explain different types of link lists.

2.8 ACTIVITY

1. Write a function to reverse the doubly link list.
2. Write a function to sort the doubly link list.

2.9 CASE STUDY

Write a program to add two polynomials. To store both polynomials, take two link lists, and to store result take one more link list. You can use either singly link lists or doubly link lists. The node will have 3 parts (in the case of singly link list) which are coefficient, exponent and address of next node. For Example,

1.4 x^5 : 1.5 x^4 : 1.7 x^2 : 1.8 x^1 : 1.9 x^0

1.5 x^6 : 2.5 x^5 : -3.5 x^4 : 4.5 x^3 : 6.5 x^1

1.5 x^6 : 3.9 x^5 : -2.0 x^4 : 4.5 x^3 : 1.7 x^2 : 8.3 x^1 : 1.9 x^0

2.10 FURTHER READING

- Data Structure through C by Yashvant kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.

Unit 3: Stacks and Their Applications

3

Unit Structure

3.0   Learning Objectives

After learning this unit, you will be able to:

- Understand, what is Stack?
- Understand methods of stack.
- Understand the Applications of Stacks.
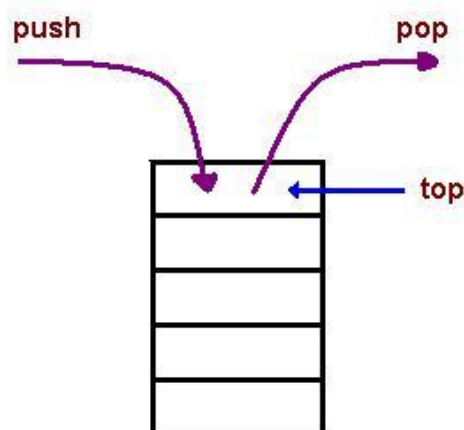- Implement Stacks.

3.1   Introduction

In this unit, we will focus on an important data-structure called Stack. In computer science there are many applications are there which need LIFO (Last-In First -Out) implementation. Stack is nothing but a LIFO implementation, where the value inserted Last will be remove first from the stack. Stack can be implemented by using array or Link-List data structure.

3.2   Definitions

Stack -

Data-Structure stack is a linear data-structure which strictly follow insertion of an element as well as deletion of an element on the top position (of stack). Stack is a enforcement of last in first out (LIFO) structure. This can be well understood by help of following diagram.

In stack, data-elements contained are inserted or deleted based on the (LIFO) Lat-In First-Our basis. Mainly two possible operations have to be implemented, while implementing a stack. When user want to add any element into the stack, user will invoke a function called push() which takes the data-element as an argument, which is given by the user and place it on the top position. Similarly, when user want to remove any element then function pop() is invoked, which returns the element from the top position (recently inserted data-element) and it will be deleted from the stack.

For example, think of multiple books which are placed on the top of one another (stack of book). You can place a new book on the top of the stack, and similarly you can take a from the top of the book stack.

Check Your Progress-1

1. A Stack is a _____ data structure.

[A] FIFO   [C] LIFO

[B] FILO   [D] LILO

2. Value is inserted or removed in the stack on _____ position.

[A] front   [C] rear

[B] top   [D] All of the above

3.3   Stack Implementation

As discussed earlier, in the implementation of the stack, we need to implement LIFO ordering. We need to develop function push() which insert the value on the top of the stack, and similarly function pop() which will remove data-element from the top of the stack. Implementation of the stack can be done by using either taking Array or Link-List data-structure. In this section, we will learn how can we implement a stack by using both of these data structures.

3.3.1 Implementation of Stack using Arrays

Stack can be implemented using array data-structure, where we will declare an array and one integer variable top. Variable top will be initialized with value -1, which means the stack is empty and there is no data-element is there in the stack (hear array).

We will enforce user that user can insert the value through push() function and remove the value using pop() function.

[1] PUSH () Function:

Function push () is used, when user want to add a data-element into the stack. Each time when user invoke a push function, first we need to evaluate for overflow condition. Suppose if the size of an array is to store MAX elements, and if top variable is on MAX -1 then, we can show that the array is full and it doesn't have space to put up new data-element. The situation of the stack, where user want to insert a data-element but array stack is full is called OVERFLOW condition. If there is no hot for overflow condition, then we will increment the value of top variable by 1, and the new data-element has to be placed on top position of an array stack. Each time when user, push a data-element, top variable is incremented by 1 and element has to be placed on the top. Execution of the stack function is demonstrated in the following figure.
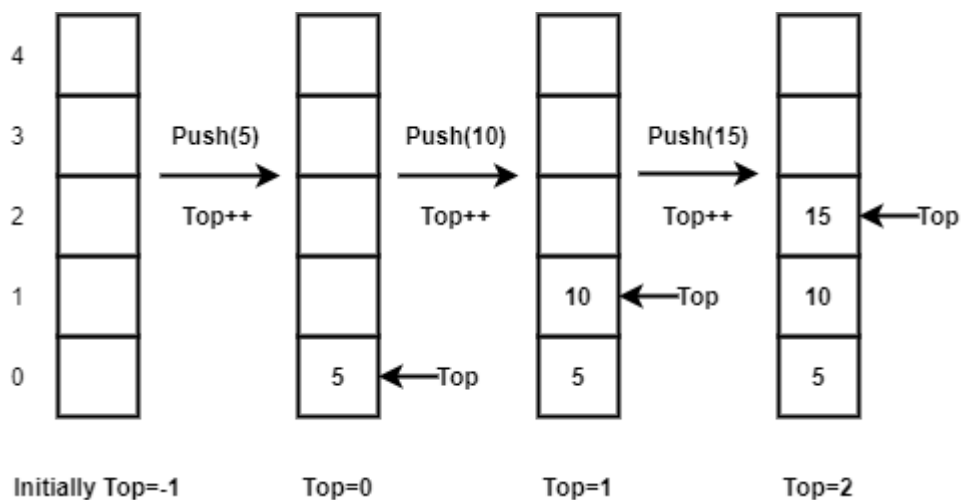
[Fig:3.1 Execution if PUSH function in the stack implemented using Array]

As shown in the figure [Fig:3.1], variable top is initialized with -1. On each time when user invoke a push function, top variable is incremented by 1. After incrementing top, the new data-element is placed on the top index number of an array. If user invoke a push() and function and suppose top is on 4th position (index), we need to show overflow message.

To implement push () function, you need to write the following code:

```
#include <stdio.h>
#define MAX 5 //Setting up MAX constant to 5
int stack [MAX]; //Declaring array stack of MAX size
int top = -1; //Declaring variable top and initializing it with value -1

void push (int val)
{
        if (top == MAX -1) //Evaluating Overflow Condition
```

```
        {
                printf ("\n Stack Overflow:");
                return; //Coming out of the Push Function
        }
        top++;  //Incrementing top variable by 1.
        stack [top] = val; //Placing the data-element on the top of the stack
}
```

[2] POP () Function:

When the user executes the pop () function, stack will return the value from the top position of an array (LIFO). If variable top is on -1 and user execute pop() function, then there is nothing is there in the stack to return. This scenario is called UNDERFLOW condition. After returning the value, it is to be assumed that the value is removed from the stack, hence the value of variable top has to be decremented by 1.
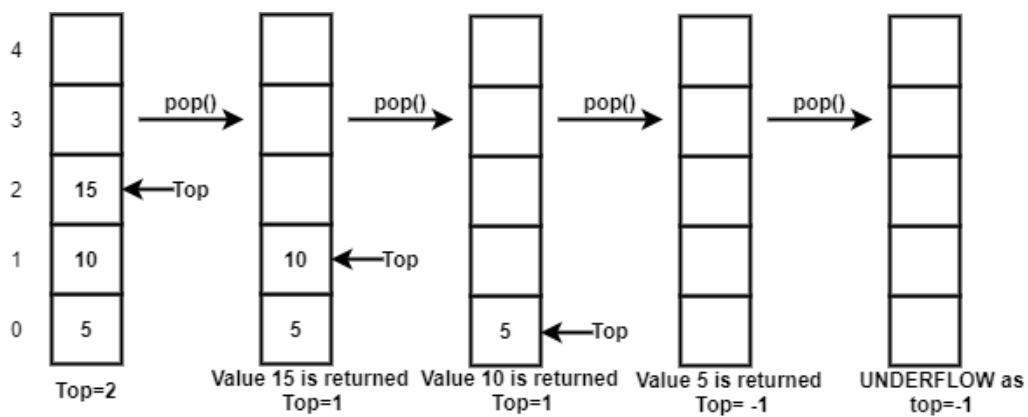
[Fig:3.2 Execution if POP function in the stack implemented using Array]

As shown in the figure [Fig:3.2], consider a stack of size 5, and there are 3 elements are already pushed in the stack called 5, 10 and 15. The variable top is on 2. On first pop() call, the value 15 has to be returned to the user, and variable top will be decremented by 1, so it will become 1. On second pop() call, value 10 will be returned and again, top will be decremented by 1 so that it will become 0. On third pop() call, last value 5 will be returned and top variable becomes -1. Now, there is no data-element is there in the stack. Suppose, still user executes the pop() function, we need to show underflow message as there is no data-element in the stack and top is on -1.

To implement the pop () function following code needs to be write:

```
int pop ()
{
        int tmp;
```

```
        if (top == -1) // Checking for Underflow condition
        {
                printf ("\n Stack Underflow:");
                return 0; //Retuning 0 when stack underflows
        }
        tmp = stack[top]; //Copying data-element in the tmp variable
        top--; //Decreasing top variable by 1
        return tmp; //Returning the value back to the caller function
}
```

[3] PEEP () Function:

Peep() function is similar to the function pop(). It returns the value which is stored on the top position of a stack, but the main difference between function pop() and function peep() is, pop() function delete the data-element from a stack and returning the value, whereas in the peep() function the value will remain as it is in the stack. Therefore, the code of the peep() is similar to pop() function but in peep() function, top varaible will not be decremented by 1.

```
int peep ()
{
        int tmp;
        if (top == -1)
        {
                printf ("\n Stack Underflow:");
                return 0;
        }
        tmp = stack[top];
        return tmp;
}
```

3.3.2 Stack implementation using Link-List

Instead of Array, Stack can also be implemented using data-structure Link-List. To implement the stack using Link-List we need to define node as shown below:
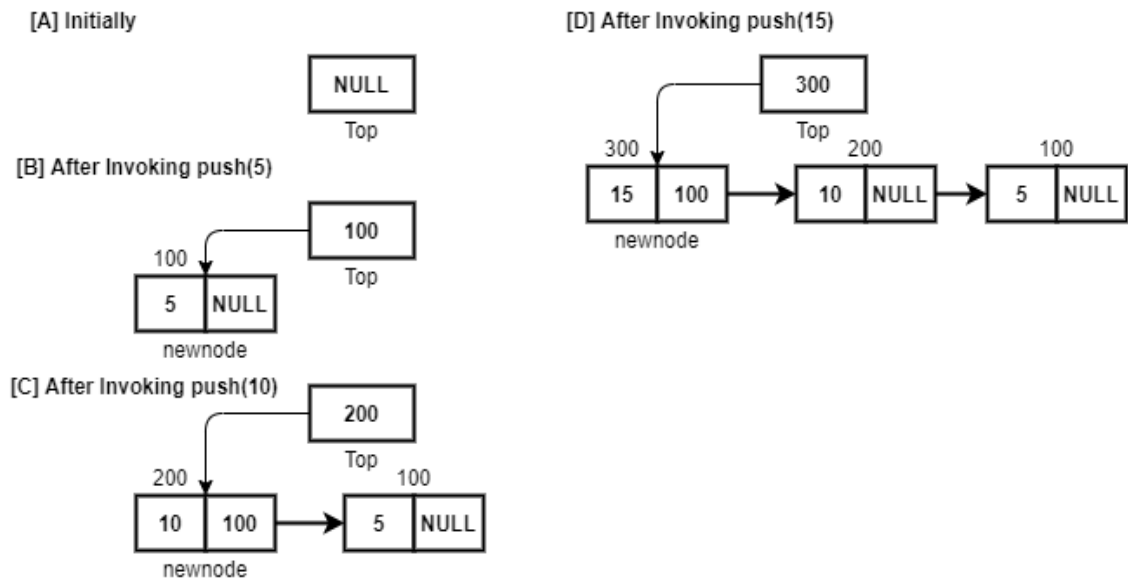
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
 int data;
 struct node *next;
} *top =NULL;
```

Definition of structure node is same as singly Link-List program. Node have two data-elements, one is data which stores users' data and another pointer variable *next to store the address of next node. We also declare a pointer variable *top of type struct node *, which will always point to the recently (latest) pushed (inserted) node. Pointer variable *top will be initialized with NULL value.

[1] PUSH () Function:

As we know, push() function indicates insertion of new node into the stack. We have to create a node using malloc() function. If malloc() function returns NULL value instead of any address of newly formed node, which means that the memory is full and system doesn't have free space to create new node. In that situation, we need to print OVERFLOW message.

If the node is created, we need to place the address stored in the *top, to the next part of the new node as well as the address of new node has to be placed in the *top. The following figure demonstrate the stack implemented by Link-List.

[A] Initially

```
NULL
```
Top

[B] After Invoking push(5)

```
100
```
Top

```
100
5    NULL
```
newnode

[C] After Invoking push(10)

```
200
```
Top
```
100
200
10   100  →  5    NULL
```
newnode

[D] After Invoking push(15)

```
300
```
Top
```
300          200          100
15  100  →  10  NULL  →  5  NULL
```
newnode

[Fig:3.3 Execution if PUSH function in the stack implemented using Link-List]

Implementation of push () function of the stack implemented using link list is given below:

```
void push (int value)
{
        struct node *new_node;
        new_node = (struct node *) malloc (sizeof (struct node));
        if (new_node == NULL)
        {
                printf ("\n Stack Overflow");
                return;
        }
        new_node->data = value;
        new_node->next = top;
        top = new_node;
}
```

[2] POP () Function:

As we know pop() function remove the data-element (node), which is on the top (pointed by *top), and return that value. To do this, we declare a pointer tmp pointer and place it on the top node. We will move the *top to top->next. We need to copy the data-element of the tmp pointer to some integer variable called val. We will delete the tmp node using free() function and finally return the value of the val variable. As we know, in the pop() function, we need to check for UNDERFLOW condition. Therefore, first we will check *top. If *top is NULL, we will show the error message for underflow condition. The code for pop() function is explained below:

```
int pop ()
{
        struct node *tmp;
 int val;
 if (top == NULL)
 {
         printf ("\n Stack Underflow");
          return 0;
 }
 tmp = top;
 top = top -> next;
 val = tmp ->data;
 free (tmp);
 return val;
```

} Check Your Progress-3

1. In the Link-List based implementation of stack, initial value of top will be _____.

[A] 0   [C] 1

[B] -1   [D] NULL

2. To detect Overflow condition in Link-List based stack, _____ condition is checked.

[A] new_node == NULL   [C] top == NULL

[B] top == -1   [D] Node of the above

3. To evaluate Underflow condition in Link-List based stack, _____ condition is checked.

[A] new_node == NULL   [C] top == NULL

[B] top == -1   [D] Node of the above

3.4   Applications of stack

In Computer Science, there are many applications are there, where stacks are important. Some applications are explained below:

1. Stack can be used to reverse the string.
2. Stack can be used in the conversion of Infix expression into Prefix or Postfix.
3. Stack can also be used to evaluate postfix expressions.
4. Stack can also be used in the application which needs backtracking.

Even though, there are many other applications are there of the stack, we will discuss the above-mentioned application in details in the following section of this unit.

[1] String Reversal using Stack:

If we take a string from the user and we push all characters of that string in the stack till we are not getting '\0' (null), and then we pop() one-one character from the stack and print all the characters till stack does not becomes empty, we will get the string to printed in the reverse order.

The following program demonstrate, how to reverse the given string using stack:

```
#include<stdio.h>
#define SIZE 10
char stack[SIZE];
int top=-1;
void push(char ch)
{
  if(top==SIZE-1)
    return;
  top++;
  stack[top]=ch;
}
char pop()
{
  char tmp;
  if(top==-1)
    return '\0';
  tmp=stack[top];
  top--;
  return tmp;
}
void main()
{
  char name[10];
```

```
    int i;
    printf("Enter Any String:");
    scanf("%s", name);
    for(i=0;name[i]!='\0';i++)


        push(name[i]);
    printf("Reverse String is:");
    while(top>-1)
    {
        printf("%c",pop());
    }
}
```

Output:

Enter Any String: BAOU

Reverse String is: UOAB

Check Your Progress-4

1. From the given below, _____ is/are basic operation of stack.

[A] push   [C] pop

[B] peep   [D] All of the above

2. From the given below _____ is not a stack application.

[A] Recursion   [C] Infix to Postfix

[B] Finding shortest path   [D] String reversal

[2] Converting an Expression from Infix to Postfix:

Usually, in our day-to-day life we write an infix expression. Consider the following expression: 2 + 3 * 5 – 7 % 3. In this expression we write an operate in-between of operands for example 2 + 3, in which '+' is an operator and '2' and '3' are operands. This type of expression (Infix expression) is easy to understand. But it is very difficult to implement it into the circuitry of calculator so that it can evaluate Infix expression easily.

To design the circuit, which can evaluate (finding answer of the expression) Infix expression, we need to convert the expression into the either postfix of prefix. Let us try to understand what is post fix expression and how can we convert the Infix expression into Postfix?

Postfix Expression:

Postfix expressions are those expressions, in which the operators are written after operands: For example, if we write Infix expression: 2 + 3, in the form of 2 3 + then operator '+' comes after operands '2' and '3' and hence 2 3 + is a postfix expression. You will feel it is difficult to understand expression to be written in 2 3 +. Yes, it is difficult for humans but for circuits of calculator, this expression is much easier to evaluate. But how can we exactly convert this? Let us learn this by an example. But before learning it, you should know about priorities of an operator first.

Priority	Operator	Meaning and name of operator
4	^ or $	Exponential operator. For example, $2^5 = 32$
3	*and /	Multiplication and Division
2	%	Modulo operator
1	+ and -	Addition and Subtraction

Make sure, the operator in the expression having highest priority is evaluate first. When two operators in the expression having same priority then associativity rule is applied, in which the operators have to be evaluate from Left to Right.

Suppose, we want to evaluate the following Infix expression into Postfix:

Steps	Expression	Explanation
1	2 + 3 * 5 – 7 % 3	Infix Expression
2	2 + (3 5 *) – 7 % 3	Multiplication is converted into postfix as * having highest priority
3	2 + (3 5 *) – (7 3 %)	Modulo is converted now into postfix
4	(2 (3 5 *) +) – (7 3 %)	As per associativity rule + operator is converted in the postfix.
5	((2 (3 5 *) +) (7 3 %) -)	Finally – operator is converted to postfix.

If we remove all parenthesis then the final postfix expression can be written as:

Postfix Expression: 2 3 5 * + 7 3 % -

I think from the above conversion from Infix to Prefix, you get the idea of how expression is converted mathematically. But the question is how machines like calculator can covert Infix expression into Postfix? Yes, machines are following program (set of instruction). Now, we will discuss how can we convert the Infix expression to postfix expression programmatically using stack.

When, any Infix expression is given to the machine (Calculator or Computer), that expression will be converted from Infix to Postfix, by a program (using stack), and then another program will evaluate (find answer) from the Postfix expression again by using stack. Let us see, how we can write a program which convert the Infix expression into Postfix?

In this program, we will take an array called infix, in which whatever Infix expression entered by the user is stored. We will take another, array called postfix in which we will store converted Postfix expression. We will also use stack, as well as we will make a function called priority which takes character (operator) as an argument and returns integer, so that we can compare the priorities between two operators. Infix expression will be converted into Postfix using following rules:

1. If character taken from the infix string is space, then that character will be skipped.
2. If character taken from the infix string is a digit or alphabet then, it will be added to the postfix string.

3. If character taken from infix string is a opening parenthesis, then it will be added to the stack by calling function push().

4. If character taken from infix string is an operator then, we need to pop() an another from the stack, an then we need to perform following comparisons:

   a. If stack is empty then the operator taken from the infix string is added to the stack by calling push() function.

   b. If the operator taken from the infix string is having higher priority than an operator popped out from a stack, then it is to be added in the stack by calling push() function.

   c. If the operator taken from the infix string is having less or equal priority than the operator popped out from a stack then, operator popped out from the stack is added into the postfix string, and again step 4 is repeated till an operator taken from an infix string is not added to the stack.

5. If character taken from infix string is closing parenthesis, then all the operators from the stack is popped out till its opening parenthesis is popped out an all those operators are added to the postfix string. Opening and Closing parenthesis are ignored and they are not be inserted in the postfix string.

6. Step1 to 5 are repeated for each character in the infix string till we are not getting '\0' from infix string.

Consider the following example, which will clear the steps we have discussed above:

Infix Expression: 4 ^ 2 * 3 – 3 + 8 / 4 / (1 + 1)		
Character taken from Infix string	**Content of the Stack**	**Postfix Expression**
4	Empty	4
^	^	4
2	^	4 2
*	*	4 2 ^
3	*	4 2 ^ 3
-	-	4 2 ^ 3 *
3	-	4 2 ^ 3 * 3
+	+	4 2 ^ 3 * 3 -
8	+	4 2 ^ 3 * 3 - 8
/	+ /	4 2 ^ 3 * 3 - 8
4	+ /	4 2 ^ 3 * 3 – 8 4
/	+ /	4 2 ^ 3 * 3 – 8 4 /
(	+ / (	4 2 ^ 3 * 3 – 8 4 /
1	+ / (	4 2 ^ 3 * 3 – 8 4 / 1
+	+ / (+	4 2 ^ 3 * 3 – 8 4 / 1
1	+ / (+	4 2 ^ 3 * 3 – 8 4 / 1 1
)	+ /	4 2 ^ 3 * 3 – 8 4 / 1 1 +
\0	Empty	4 2 ^ 3 * 3 – 8 4 / 1 1 + / +

The program to convert Infix expression to Postfix is given below:

```
#include<stdio.h>
#include<ctype.h>
#define MAX 100
```

```c
char st[MAX];
int top=-1;
void display()
{
    int i;
    for (i=0; i<=top; i++)
    {
        printf("%c", st[i]);
    }
}
void push(char val)
{
    if(top==MAX-1)
    {
        printf("\nStack Overflow:");
        return;
    }
    top++;
        st[top]= val;
}
char pop()
{
    char tmp;
    if (top == -1)
    {
        printf("\nStack Underflow:");
        return '0';
    }
    tmp = st[top];
    top--;
    return tmp;
}
int getPriority(char op)
{
    if (op=='^' || op=='$')
        return 3;
    else if (op=='/' || op =='*' )
        return 2;
    else if (op=='%')
        return 1;
    else
        return 0;
}
void infixtopostfix (char *source, char *target)
{
```

```
int i=0, j=0;
char tmp;
while(source[i] != '\0')
{
  if(source[i] == '(' )
  {
    push(source[i]);
    i++;
  }
  else if (source[i] ==')' )
  {
    while ((top==-1) || st[top] != '(' )
    {
      target[j] = pop();
      j++;
    }
    if (top == -1)
    {
      printf ("\nInvalid Expression:");
      return;
    }
    tmp = pop(); //Removing left parenthesis
    i++;
  }
  else if (isdigit(source[i]) || isalpha(source[i]))
  {
    target[j] = source[i];
    j++;
    i++;
  }
  else if(source[i] == '+' || source[i] == '-' || source[i] == '*' || source[i] == '%' ||
source[i] == '/'  || source[i] == '^' || source[i] == '$')
  {
    while((top!=-1)   &&   (st[top]   !=   '(')   &&   (getPriority(st[top])   >=
getPriority(source[i])))
    {
      target[j]=pop();
      j++;
    }
    push(source[i]);
    i++;
  }
  else
  {
    printf ("\nInvalid Expression:");
```

```
      return;
    }
  }
  while ( (top != -1) && (st[top] != '(' ) )
  {
    target[j] = pop();
    j++;
  }
  target[j] = '\0';
}
void main()
{
  char infix[MAX], postfix[MAX];
  printf("\nEnter Any Infix Expression:");
  gets(infix);
  infixtopostfix(infix, postfix);
  printf("\nCorresponding postfix expression is:");
  puts(postfix);
}
```

Output:

Enter Any Infix Expression:4^2*3-3+8/4/(1+1)

Corresponding postfix expression is:42^3*3-84/11+/+

Prefix Expression:

Prefix expression are those, in which we write the operator first and then operands. For Example: Infix expression $2 + 3$ is converted into prefix as $+ 2\ 3$. Consider the following example in which, we want to convert a given Infix expression: $2 + 3 * 5 - 7\ \%\ 3$ in to the prefix then:

Steps	Expression	Explanation
1	2 + 3 * 5 – 7 % 3	Infix Expression
2	2 + (* 3 5) – 7 % 3	Multiplication is converted into prefix as * having highest priority
3	2 + (* 3 5) – (% 7 3)	Modulo is converted now into prefix
4	(+ 2 (* 3 5)) – (% 7 3)	As per associativity rule + operator is converted in the prefix.
5	– (+ 2 (* 3 5) (% 7 3)	Finally – operator is converted to postfix.

If we remove all parenthesis then the final prefix expression can be written as:

Postfix Expression: – + 2 * 3 5 % 7 3

Yes, it is true that designing of circuits which can evaluate Infix expression is difficult, but if we convert the expression into the Prefix or Postfix then we can easily design the circuit to evaluate the given Prefix or Postfix expression. Because from both the options (Prefix and Postfix), Postfix is chosen as a universal standard, In the implementation of all different types of circuits of calculators the given infix expression is converted into postfix.

So that we will not go into the details of how Infix expression is converted into Prefix, programmatically.

[3] Evaluating a Postfix expression using Stack:

As we have discussed that the circuits of the calculator take an Infix expression and convert it into the Postfix expression. Finally, it processes that Postfix expression into the final result. But how? Let us see with an example.

Let us assume that the Postfix expression: **2 3 5 * + 7 3 % -** is given then the expression is evaluated as follows:

1. If character taken from the operator is alphabet or number then it is to be inserted in the stack.
2. If it is an operator then two operands are popped out from the stack, operation is performed and the resultant value is pushed back to stack.
3. When we get '\0' from the Postfix expression, then there will be only one value will be there in the stack and it will be final answer.
   Consider the following figure, which express the evaluation of the above equation:
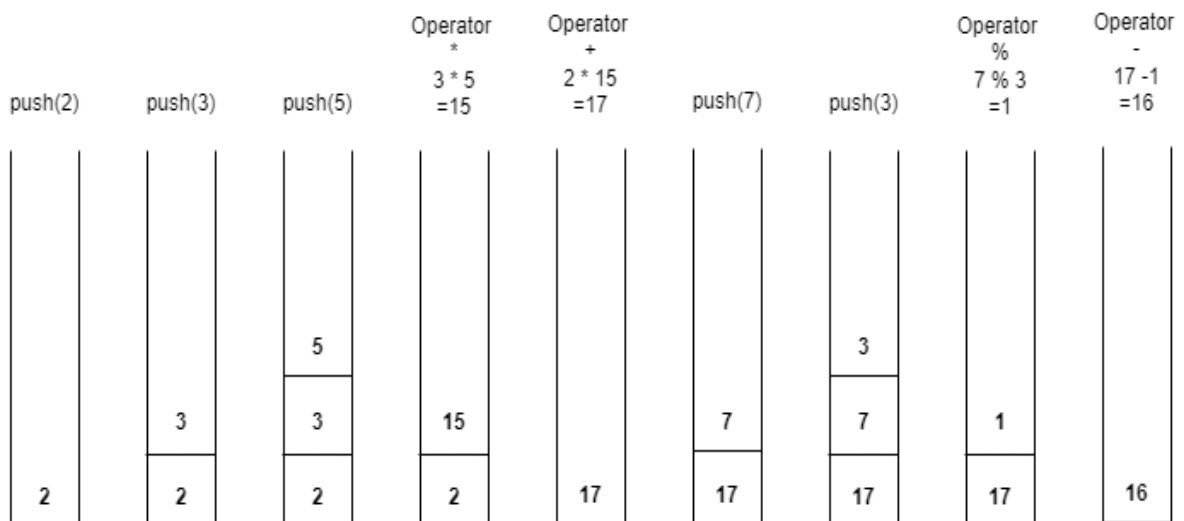
So, the Final answer is: 16

The program for evaluating the postfix expressing into the answer is given below:

```c
#include<stdio.h>
#include<ctype.h>
#define MAX 100
float st [MAX];
int top = -1;
void push (float val)
{
   if (top == MAX-1)
   {
     printf ("\nStack Overflow:");
     return;
   }
   top++;
     st[top] = val;
}
float pop()
{
   float tmp;
   if (top == -1)
   {
     printf ("\nStack Underflow:");
     return '0';
   }
   tmp = st[top];
   top--;
   return tmp;
}
float evalpostfixexpr (char *exp)
{
   int i=0;
   float op1, op2, value;
   while (exp[i] != '\0 ')
   {
     if ( isdigit (exp[i]) )
     {
       push ((float)(exp[i]-'0'));
     }
     else
     {
       op2 = pop();
       op1 = pop();
```

```
            switch ( exp[i] )
            {
            case '+':
                value = op1 + op2;
                break;
             case '-':
                value = op1 - op2;
                break;
             case '*':
                value = op1 * op2;
                break;
             case '/':
                value = op1 / op2;
                break;
             case '%':
                value = (int) op1 % (int) op2;
                break;
             }
             push(value);
          }
        i++;
      }
      return( pop() );
   }
   void main()
   {
      float val;
      char exp[MAX];
      printf("\nEnter Any Postfix Expression:");
      gets(exp);
      printf("\nValue of the Postfix Expression is:");
      val = evalpostfixexpr(exp);
      printf("%f", val);
   }
```

Output:

Enter Any Postfix Expression:235*+73%-

Value of the Postfix Expression is:16.000000

Check Your Progress-5

1. Postfix of the Infix expression: $2 + 3 * (9 - 7) \% 3$ is _____

[A] 97-3*3%2+   [C] 2397-*3%+

[B] 23+97-*3%   [D] 23973+*-%

2. _____ is also known as reverse polish notation.

[A] Infix   [C] Prefix

[B] Postfix   [D] Node of the above

3. Postfix of the Infix expression: A * B – (C + D) + E is _____.

[A] AB*CD+-E+   [C] ABCDE*-++

[B] AB*CD+-E+   [D] CD+B-A*E+

3.5   Let Us Sum Up

In this Unit we have done detailed discussion on the Stack data-structure. Whenever the data elements are arranged one above the other or in LIFO ordering, then stack data-structure is used. In a stack, data elements can be always inserted at top of the stack and can also always be deleted from the top position; hence, it follows, Last in First out (LIFO) order. The insertion of an element to a stack is called performing PUSH operation and removal of an element from a stack is called performing POP operation.

Consider, at the time of insertion of a data-element, if the stack is full then this condition is called Stack Overflow. And, similarly, consider that at the time of removal if the stack is empty and we are trying to delete a data-element, then this condition is known as Stack Underflow. Stacks are represented in two ways namely 1) Array implementation and 2) Link-List representation.

Stack can have many applications. Stack can be used to reverse the given string, to convert the Infix expression into either Prefix or Postfix, it can also use in evaluating of Postfix expression. Stack can also be used in the solving of Backtracking kind of complex problems like solving 8 Queen puzzle. Stack data-structure is also a good replacement of recursion.

3.6   SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

1. [C] LIFO
2. [B] top

Check Your Progress-2

1. [B] -1
2. [A] push

Check Your Progress-3

1. [D] NULL
2. [A] newnode == NULL
3. [C] top ==NULL

Check Your Progress-4

1. [D] All of the above
2. [B] Finding shortest path

Check Your Progress-5

1. [C] 2397-*3%+
2. [B] Postfix
3. [A] AB*CD+-E+

3.7   GLOSSARY

1. **Stack**: Stack is a linear data structure, which can be implemented by either array or Link-List. It is collection of data implemented in LIFO (Last-In First-Out) ordering.
2. **Infix:** Infix is a method of writing expression, in which all binary operators are written between two operands. In our day to day lives, we use Infix expression.

3.8   Assignment

- Implement a program which accepts Infix string from the user and convert it into the postfix as well as it evaluates the postfix expression.

3.9   Activity

1. Implement a program which reverse the given string using stack.
2. Write a program which accepts integer number from the user and find its equivalent binary number using stack.

3.10   Case Study

Make a note on Eight Queen Puzzle. Search about Backtracking and Eight Queen puzzle on the Internet and develop an algorithm to solve Eight Queen Puzzle.

3.11   Further Reading

- Data Structure through C by Yashvant Kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.

Unit 4: Queues and Their Applications

<div style="text-align:right">**4**</div>

Unit Structure

4.0   Learning Objectives

After learning this unit, you will be able to understand:

- Basic operations can be performed on queue
- Array and Link-List representation of queue
- D-Queue
- Circular queue and its implementation
- Applications of queue

4.1   Introduction

In the last unit we have studied data-structure queue, in which the data elements were arranged on the top of the other. In this unit, we are going to discuss about another type of data structure called queue in that the data-elements are arranged one after other. For Example, you might have seen the queue at bus stop or a railway station, where people might wait for their turn. One thing to be noticed there, the Queue server the people in FIFO (First-In First-Out) ordering. We will also be discussing about the array and Link-List representation of queue. Apart from this we are going to focus on various types of queues. This is very important data structure, used in many different applications. Queue plays an important role in the CPU scheduling. Today, we are using multitasking operating systems, where we can run more than one task or application at the same time because of queue data structure.

4.2   Definition

Queue

Queue is a linear data structure in which elements are inserted through the rear end and removed from the front end.
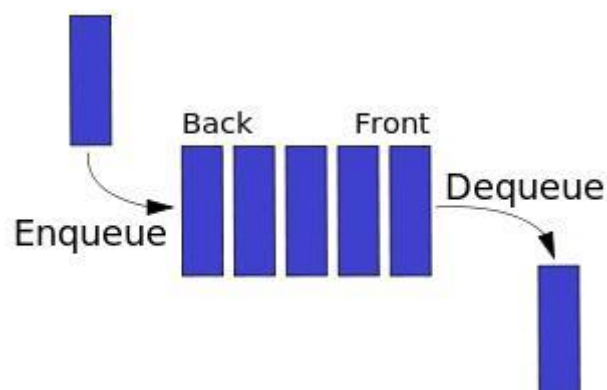

Fig: 4.1 Queue operations

A queue is a container of data-elements, that are inserted one after the other and deleted according to the (FIFO) first-in first-out principle. For example, a queue of customers waiting to pay their bills at the cashier's counters in a mall. Data structure queue is a linear data structure and hence it can be implemented by using either an array or a Link-List data structure. In the queue data structure, data-element is to be inserted from the rear (one end) while data-element is to be removed from front (another end).

Stack

Stack is data structure in which elements are added and deleted through the same end that is top of the stack. (As we have already studied in the previous topic in details).

Check Your Progress-1

1. A Queue is a _____ data structure.

[A] FIFO   [C] LIFO

[B] FILO   [D] LILO

2. Value is added in the queue from _____ position.

[A] front   [C] rear

[B] top   [D] All of the above

3. Value will be deleted from the queue from _____ position.

[A] front   [C] rear

[B] top   [D] All of the above

4.3   Basic operations performed on queue

The two main operations can be performed on queue are insert and delete.

Data-elements in queue are inserted through the rear-end using the enqueue operation and deleted from the front-end using dequeue operation.

Operations on queue are:

1.   **Enqueue -** insert item at the back of queue

2.   **Dequeue -** return (and virtually remove) the front item from queue

3.   **Init -** initialize queue, reset all variables

4.4 Array and Link-List representation of queue

Queue can be implemented statically or dynamically i.e., as an array or as a Link-List.

1. Array implementation of Queue
2. Link-List implementation of Queue

4.4.1 Array representation of Queue

As discussed, we have discussed earlier, implementation of the Queue can be done by either taking of an Array or a Link-List. In the case of array, you need to take an array to store the data-elements of queue. Along with that, you need to take two more variables front and rear to keep track of front and rear ends (positions) of array.

Recall data structure stack. In the stack data structure, we have taken a variable top, because in stack data structure, data-elements are inserted on the top, and removed from the top position (same end). In the case of queue, data-elements are inserted at the rear end and removed from the front end. So, to implement queue you need to declare following global variables.

```
# include <stdio.h>
# define MAX 10
int queue [MAX];
int front = -1;
int queue = -1;
```

Similar to the top variable of stack program, you need to initialize front and rear variables with value -1. Variables front and rear are -1, indicates that the queue is empty, and there is no data-element is there in the queue.

[1] Enqueue () Function:

As we know, user will invoke function enqueue() to insert a data-element into the queue. We have already discussed that the data-element in the queue is added at rear end. So, to implement enqueue() function, we need to check the rear end, if it is MAX -1, then queue is full (as array cannot accommodate any element on MAX position). Otherwise, we have to increment value of rear variable by 1, and we need to place the newer value on the rear index number of the queue array. When the first data-element is inserted, we also need to set front variable to 0. The following figure will help you to understand the functioning of enqueue () function.
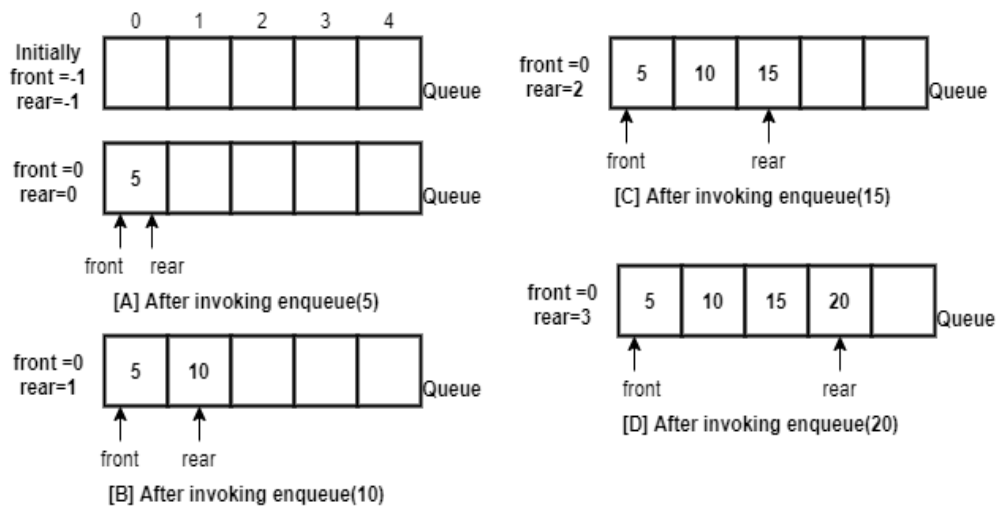
Fig: 4.2 Queue operation – enqueue

Function enqueue () can be implemented as follow:

```
void enqueue (int val)

{

  if (rear == MAX -1)

  {

        printf ("Queue Overflow:");

        return;

  }

  rear++;

  queue[rear] = val;

  if (front == -1)

        front++;

}
```

[2] Dequeue() Function:

In the function dequeue(), you need to check variable front, if it is -1 then the queue is underflow (as there is no data-element there to delete in the queue array). Otherwise, you need to copy the data-element situated on the front position of the queue array into any temporary variable called 'tmp' and increment the front variable. Before doing this, you need to check one more condition. If the value of front and rear variables are similar, that means user is going to delete the last data-element from the queue. In such situation you need to set front and rear both variables to -1. The following figure [Fig:4.3] will help you to understand the function dequeue().
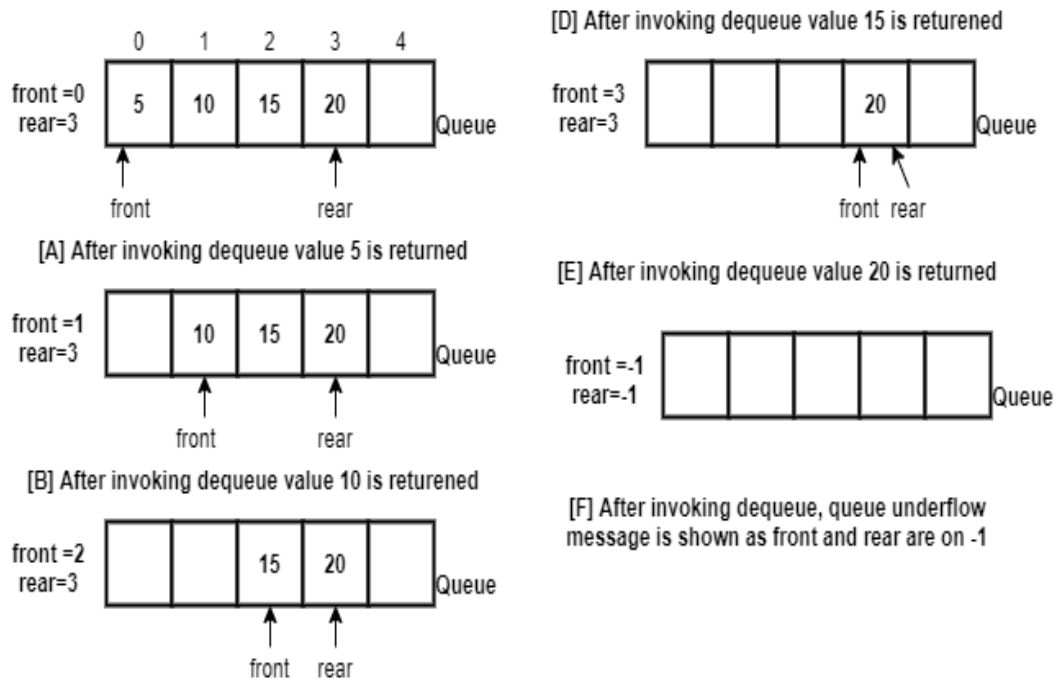
114

Fig: 4.3 Queue operation – dequeue

To implement the dequeue() function, following code need to be written:

```
int dequeue ()
{
  int tmp;
  if (front == -1)
  {
        printf ("Queue Underflow:");
        return 0;
  }
  tmp = queue[front];
  if (front == rear)
        front = rear = -1;
  else
        front++;
  if (front == -1)
        front++;
        return tmp;
}
```

Check Your Progress-2

1. Identify the correct statement from the given below:

[A] In enqueue() function, front variable is increased and in dequeue() function it will be decreased.

115

4.4.2 Link-List representation of Queue

Data structure Queue can also be implemented using Link-List. Like singly Link-List you can create a structure of node with two elements: data and next. Here you need to declare two pointer variables of type 'struct node': *front and *rear. Both pointers, *front and *rear will be initialized with NULL value.

Struct node

{

  int data;

  *struct node *next;*

*} *front = NULL, *rear = NULL;*

[1] Enqueue () Function:

To implement enqueue() function, you need to create a new node using malloc() function. If function malloc() returns NULL value, instead of address of newly created node, you need to print the message that Queue is full. Otherwise, you can insert the new node at the end (rear) of the queue. Make sure, in the implementation of the queue using Link-List, *front will always point to first node of the Link-List and *rear will always point to the last node of the Link-List. Figure [Fig:4.4] given below will help you to understand functioning of the enqueue() function of the queue implemented using Link-List.
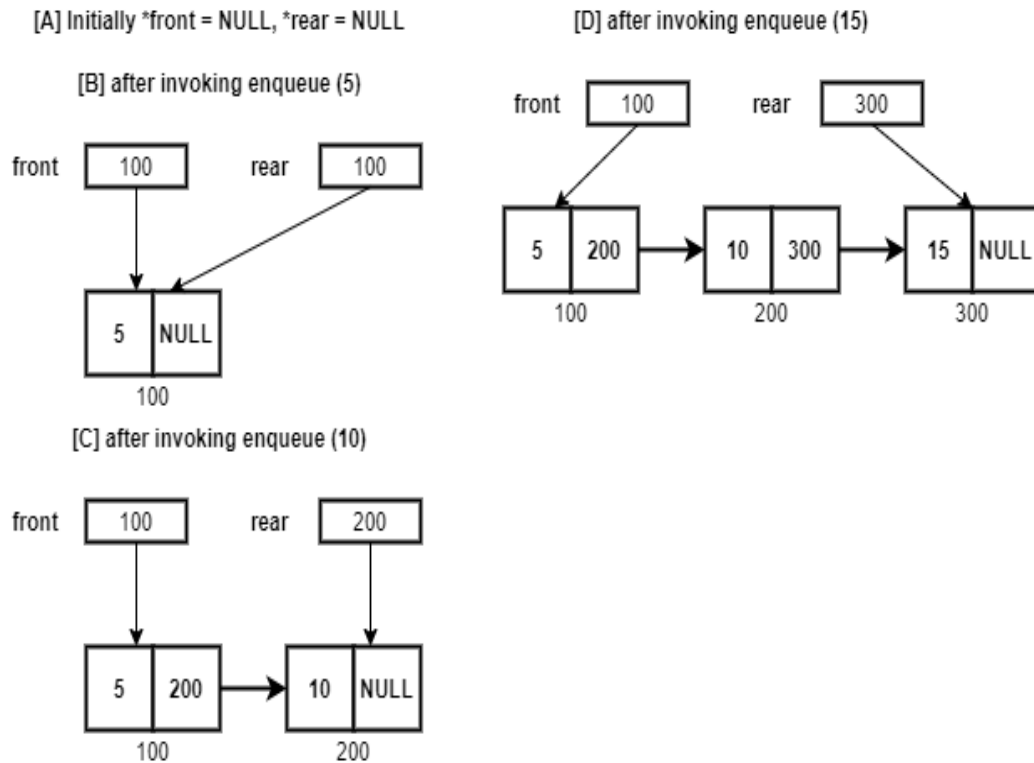
Fig: 4.4 Queue operation – enqueue using Link-List

From the figure [Fig:4.4], you will come to know that, initially when program starts its execution, there no node is there in the Link-List, therefore *front and *rear both will have NULL value. When user inserts value 5 in the queue, we create a new node and both pointers, *front and *rear points to this new node. When user inserts another value 10, at that time a new node is created and we will put value 10 in the data section of the new node. We will place the address of the new node into the next part of the rear node (*rear pointer which pointe to address 100). Now because the new node with value 10 is inserted in the Link-List, *rear should point to newly inserted node. Therefore, we will place the address of new node in the *rear. The following code, you need to write to implement enqueue() function.

```
void enqueue (int val)
{
  struct node *new_node;
  new_node = (struct node *) malloc (sizeof (struct node));
  if (new_node == NULL)
  {
    printf ("\nQueue is Full:");
    return;
  }
```

```
    new_node->data = val;
    new_node->next = NULL;
    if (front == NULL)
    {
       front = rear = new_node;
       return;
    }
    rear->next = new_node;
    rear = new_node;
}
```

[2] Dequeue () Function:

As we know function dequeue() is used to delete element from the queue, and in queue, data-element is deleted from the front end. To implement dequeue() function, we will place a temporary pointer on the first node (by copying address from *front). We will move *front to its next node and then we can delete that node. The following figure will give you better idea to understand the functioning of dequeue () function.

The following code, we need to write to implement dequeue () function:

```
int dequeue ()
{
    int data;
    struct node * tmp;
    if (front == NULL)
    {
       printf ("\nQueue Underflow:");
       return 0;
    }
    tmp = front;
    front = front->next;
    data = tmp->data;
    free(curr);
    if (front == NULL)
       rear = NULL;
    return data;
}
```

4.5   D-Queue

D-Queue is known as double ended queue. It is a data structure in which the element can be inserted or deleted from both ends (either through the front end or rear end). D-Queue can be implemented with a modified dynamic array or with a doubly Link-List.

A D-Queue is a generalization way of both the FIFO Queue and the LIFO stack. A D-Queue represents a sequence of elements, with front and back variables. Data-elements can be inserted at the front of the queue or the back of the queue. The names of the D-Queue operations are self-explanatory:
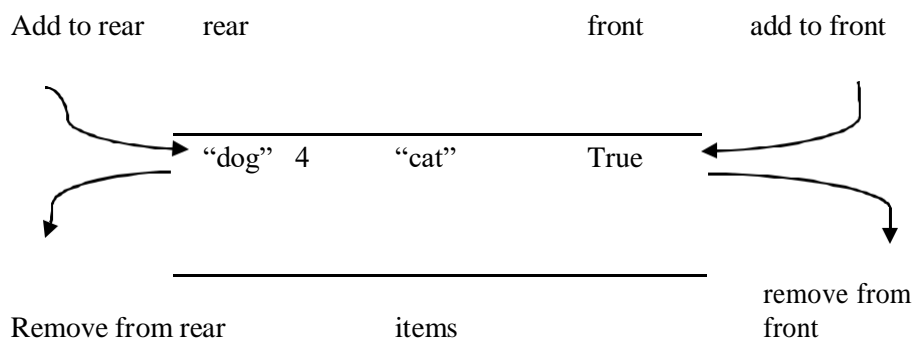

Fig: 4.4 Queue operation – enqueue using Link-List

However, unlike stack and queue, the deque (pronounced "deck") has very few restrictions. Also, be careful that you do not confuse the spelling of "deque" with the queue removal operation "dequeue."

Check Your Progress-4

1. _____ data structure can be used as FIFO and LIFO ordering.

[A] stack   [C] queue

[B] circular queue   [D] Deque

2. _____ is not a valid function for Deque.

[A] add_to_front   [C] add_to_rear

[B] enqueue   [D] remove_from_front
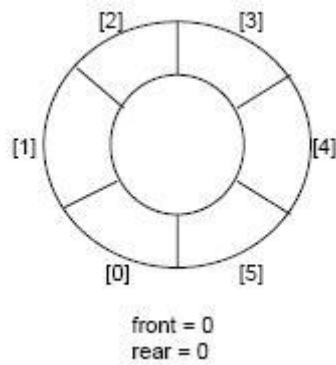

4.6   Circular queue

Circular queue is an improved algorithm over queue. Circular queue is similar to queue, which enforce FIFO ordering. Consider the following operations on Queue and answer the following question:

[1] enqueue(5)   [2] enqueue(10) [3] enqueue(15) [4] enqueue(20) [5]enqueue(25)
[5] dequeue()   [6] dequeue()   [7] dequeue ()

0	1	2	3	4
			20	25

Front Rear
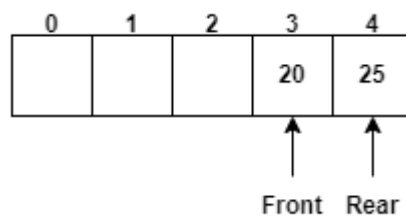
You can see here, there are only 2 elements are there in the queue and those are 20 and 25. Data-elements 5, 10 and 15 are deleted from the queue. Now think what happen if we call enqueue(30). Obviously, it gives an error message that the queue is full (as rear == MAX -1).

We can see here, that the queue has capacity to store 5 data-elements. But in this scenario in just two elements, we get the message that the queue is full. Therefore, we can say that data-structure simple queue is not memory efficient. To overcome this problem, data-structure circular queue is proposed. In this the queue is to be assumed as follows:

[Fig:4.5 Circular Queue]

Circular queue, is memory efficient compare to simple queue, as it guarantees you can store maximum 5 elements of the size of an array is to accommodate 5 elements. Circular queue can be implemented (enqueue and dequeue functions) as follows:

```
 #include<stdio.h>
#define MAX 5
int queue[MAX];
int front = -1;
int rear = -1;
/* Function enqueue to insert the data-element into the Queue */
void enqueue (int val)
{
  if ((front==0 && rear == MAX -1) || (rear+1==front))
  {
        printf ("Queue Overflow:");
        return;
  }
  if(rear==MAX-1)
     rear=0;
   else
     rear++;
  queue[rear] = val;
  if (front == -1)
        front++;
}
```

```
/* Function dequeue to delete or remove the data-element from the queue */
int dequeue ()
{
  int tmp;
  if (front ==  -1)
```

```c
    {
            printf ("Queue Underflow:");
            return 0;
    }
    tmp = queue[front];
    if (front == rear)
            front = rear = -1;
    else if (front==MAX-1)
        front =0;
     else
            front++;

    return tmp;
}
/* Main Function of the Program Circular Queue */
void main()
{
    int choice, val;
    do
    {
        printf ("\n/--------MENU--------/");
        printf ("\n 1. EnQueue:");
        printf ("\n 2. DeQueue:");
        printf ("\n 3. Exit:");
        printf ("\n/--------------------/");
        printf ("\n Enter Your Choice:");
        scanf ("%d", &choice);
        if (choice == 1)
        {
            printf ("Enter Value:");
            scanf ("%d", &val);
            enqueue (val);
        }
        else if (choice == 2)
        {
            val = dequeue();
            printf ("\nValue Removed from Queue is:%d", val);
        }
        else if(choice==3)
        {
            printf("\nGood Bye:");
            break;
        }
        else
```

```
    {
      printf ("\nInvalid Choice:");
    }
  } while (choice != 3);
}
```

4.7   Applications of queue

Queue is an abstract data-type which follows FIFO and because of this property queue is applicable in the following situation. Queue is a linear data structure can be used to serve the resources on the basic of first come first serve basis. In the operating system, resources like processor, memory, printers etc. are given to different processes (programs in execution) using queue. The following are some applications where data structure queue is used.

1.   When resources are required to be shared between multiple users. Examples in CPU scheduling, Disk Scheduling.
2.   When data is transferred asynchronously between two processes. Examples in pipes, IO Buffers, file IO, etc.
3.   Simulations like, think of a situation at airport when multiple aircrafts are in ready queue and waiting for (their turn in a queue for) a run-way to take off. Similarly other queue is there in the air, where several aircrafts are waiting (in flying condition), and they also need runway to land. Here, a queue in the air has to be more prioritized because plan waiting in the air, consume Hugh amount of fuel.

Check Your Progress-5

1. _____ is/are application(s) of queue data structure.

[A] CPU scheduling   [C] Disk scheduling

[B] Breadth First Search   [D] All of the above

2. _____ is a memory efficient version of queue.

[A] stack   [C] tree

[B] graph   [D] circular queue

4.8   Let Us Sum Up

In this Unit we have focused on queue and their types. A queue data structure can be considered by the fact that inducing or insertions of data-element is made at one end and deletion of a data-element is made from another end. Further the first data-element of queue is called as a front element and the last data-element of queue is called as a rear element. In case of queue, we have to insert an element from the rear end and we need to delete an element from the front end.

The queue follows FIFO ordering, in which the data-element which is inserted first will be the first one to be removed and taken out. In case the queue is full and we are trying to insert an element to it, then this condition is called Overflow and if the queue is does not have any data-element and we are trying to delete an element from it, then such condition is called as Underflow.

Further we have studied both types of implementations of queue i.e., Implementation of queue by array and by Link-List. There are two basic operations, insertion and deletion, which can be performed on queues. As we have understood about the queue type one can summarize by noting following type of queues:

- **Circular Queue -** In circular queue, if the 'front' is equal to 'rear', the circular queue will be measured as empty. If the circular queue is defined to contain N elements, then it can only support N-1 elements, as after insertion of Nth element, the rear will be equal to front. So, we cannot say whether the queue is empty or full.
- **Double Ended Queue -** Always a double ended queue is an abstract data structure which implements a queue for which data-elements can only be inserted to or removed from the front or rear. These dequeues can be of following given types: a) See that an input-restricted deque, where deletion can be made from both ends, but input can be made at one end.2) See that an output-restricted deque, where input can be made at both ends, but output can be made from one-end only.
- **Priority Queue -** A priority queue can be considered as a modified or changed queue, but when any one would get the next element off the queue, the highest priority one is retrieved or accessed first. Even a queue is a special case of a priority queue where an element's priority is the time, that element was inserted. The highest priority element is at the top.

At the end of this unit, it is also important to learn about few applications of queue: those are

I. When resources are required to share between multiple users. Examples in CPU scheduling, Disk Scheduling.

II. When data is transferred asynchronously between two processes. Examples in pipes, IO Buffers, file IO, etc.

4.9 SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

1. [A] FIFO
2. [C] rear
3. [A] front

Check Your Progress-2

1. [D] None of the above
2. [D] rear++

Check Your Progress-3

1. [D] NULL
2. [C] underflow
3. [D] enqueue

Check Your Progress-4

1. [D] Dequeue
2. [B] Enqueue

Check Your Progress-5

1. [D] All of the above
2. [D] Circular queue

4.10 GLOSSARY

1. **Queue** - is an abstract data type which follows FIFO.
2. **Deque** - A Deque is a generalization way of both the FIFO Queue and LIFO Queue (Stack).

4.11   Assignment

- Write following programs: [1] Implementation of queue using array, [2] Implementation of queue using Link-List and [3] Circular queue.

Block Summary

- Link-List is similar to an array. It is an ordered collection of homogeneous data. Unlike array the data-elements in Link-List are stored in the form of nodes, in non-contiguous memory locations.

- There are three types are there of the Link-List [1] Singly Link-List [2] Doubly Link-List and [3] Circular Link-List.

- In Singly Link-List each node has data section in which data given by the user is stored, and next section in which the address of the next node is stored. The address of the first node is stored in the *first. Similarly in the next section of the last node value 'NULL' is stored.

- In Doubly Link-List each node has three sections. Data section stores user's data, Lptr section stores the address of the node situated at left side of the node, and Rptr stores the of the node situated at right side. Lptr section of the first node, and Rptr section of the last node must be 'NULL'. In Doubly Link-List we can travers in the forward as well as in the reverse direction, as each node has two addresses stored in *lptr and *rptr.

- In Circular Link-List (which similar to the singly Link-List), in the next section of the last node we store the address of the first node, so user can come to the first node after visiting the last node.

- Stack is a linear data structure, which stores and retrieves the data-elements in LIFO (Last-In First-Out) ordering.

- Stack can be implemented by either using an array or a Link-List.

- Two important methods of the stack are: [1] Push: which is used to insert a data-element in the stack and, [2] Pop: which is used to remove an element from the stack as well as it returns the value deleted form the stack.

- Stack can be used in the applications like:[1] String reversal [2] In the conversion of Infix expression into either Prefix or Postfix [3] Evaluating Postfix expression and [4] Solving backtracking (8-Queeen puzzle) type of complex problems.

- Stack is also good replacement over recursion.

- Queue is also a liner data structure, in which insertion is done at rear end and deletion is done from front end.

- Queue can also be implemented using either array or Link-List data structures.

- Queue data structure enforce FIFO (First-In First-Out) ordering. Function enqueue() is used to insert a data-element into queue, and function dequeue() is used to remove and return a data-element from the queue.

- D-Queue is data structure is also known as double ended queue, which allows data-elements to be inserted or removed from both ends (front and rear). It can be used as stack (for LIFO implementation), as well as a Queue (for FIFO implementations).
- Queue can be of four types: [1] Simple Queue [2] D-Queue [3] Circular Queue [4] Priority Queue.

Block Assignment

Short Questions:

1) Explain node structure for singly Link-List.

2) Explain node structure for doubly Link-List.

3) Define Stack data structure.

4) What are the different data structures can be used to implement stack?

5) List different applications of the Stack data structure.

6) Define: Queue.

Long Questions:

1) List and explain different types of Link-Lists in detail.

2) What is Stack? Write a program to implement stack using Array.

3) What is Stack? Write a program to implement stack using Link-List.

4) What is Queue? Write a program to implement queue using Array.

5) What is Queue? Write a program to implement queue using Link-List.

6) What is Circular Queue? Write a program to implement circular queue.

Enrolment No.

1. How many hours did you need for studying the units?

Unit No.	1	2	3	4
No. of Hrs				

2. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific example if any
Presentation Quality	☐	☐	☐	☐	_____
Language and Style	☐	☐	☐	☐	_____
Illustration used (Diagram, tables etc)	☐	☐	☐	☐	_____
Conceptual Clarity	☐	☐	☐	☐	_____
Check your progress Quest	☐	☐	☐	☐	_____
Feed back to CYP Question	☐	☐	☐	☐	_____

3. Any other
Comments

……………
……………
……………
……………
……………
……………
……………
…

……………
……………
……………
……………
……………
……………
……………
…

Dr. Babasaheb
Ambedkar
Open University

BSCIT-203

Fundamentals of Programming
Using C Language

BLOCK 3:   TREES AND GRAPHS

BLOCK 3: TREES AND GRAPHS

Block Introduction

In this Block, we will discuss about two important data structures, which are categorized as non-linear data structure. These data structures are Trees and Graphs. In the first unit we will try to define tree data structure. We will learn What is Tree data structure? And will understand about different terminologies associated with trees. We will discuss a special type of tree, that is BST – Binary Search Tree, and discuss its importance, insertion and traversal process.

In the Unit:2, we will learn implementation of BST, not only that the problems in BST algorithm and to solve those problems, why AVL-Tree (Height balance tree) is required? Will understand. We will discuss different tree rotation to make AVL-Tree to be height balanced. Finally, we will end this unit with discussion of M-way trees and its one category, that is B-Tree.

In the last unit:3, we will discuss about Graph, its definition, terminologies, types etc. We will learn representation methods and traversal methods of a graph. Finally, we end our discussion with Dijkstra's shortest path algorithm and to build minimum cost spanning tree from a graph we will discuss Kruskal's and Prim's algorithms.

Block Objective

After learning this Block, you will be able to:

- Understand tree data structure and terminologies related to tree.
- Know special type of tree that is Binary Search Tree
- Understand Insertion and Traversal in BST
- Learn Implementation of BST
- Understand concepts of AVL-Tree and B-Tree.
- Understand Graph and Applications of graph
- Learn different terminologies related to graph and types of graphs
- Perform representation and traversal in graph
- Understand solution of shortest path problem using Dijkstra's algorithm
- Know Minimum cost Spanning Trees (MST) and Kruskal's and Prim's algorithms.

Block Structure

BLOCK 3: TREES AND GRAPHS

UNIT1   TREES

Objectives, Introduction, Trees – as Abstract Datatype, Terminologies, Binary Search Trees (BST), Traversal in Binary Search Tree, Let Us Sum Up

UNIT 2   ADVANCED TREES

Objectives, Introduction, Implementation of Binary Search Tree (BST), AVL – Tree, B – Tree, Let Us Sum Up

UNIT 3   GRAPHS

Objectives, Introduction, Definitions, Types of graphs, Terminologies, Graph representation methods Adjacency Matrix and Adjacency List, Traversal Methods DFS and BFS, Dijkstra's shortest path algorithm, Minimum cost Spanning tree – Kruskal's and Prim's algorithms, Let Us Sum Up

Unit 1: Trees   1

Unit Structure

1.0   LEARNING OBJECTIVES

In this unit, we will discuss data-structure Tree and its types such as Binary Search Tree (BST), AVL-Tree, B-Tree etc.

After learning this unit, you will be able to understand:

- Defining a tree as ADT (Abstract Data Type)
- Properties of a Tree and Binary tree
- Implementation of the Tree and Binary tree, and
- Applications of Tree data-structure.

1.1 INTRODUCTION

FinalBlock-2.doc

Do you know, how does the files we are creating is stored by the operating system, we are using? Yes, the answer is: in hierarchical structure. But why do we need a hierarchical file system? Well, you have answers to all these questions in this unit through a hierarchical data-structure know as – Trees. Although in the most general form a three can be defined as – **acyclic graph**, in this unit we will consider, only those Tree data-structure in which a root node is present, and nodes are arranged in hierarchical manner with parent-child relationship.

Tree is a data-structure which enables you to subordinate a parent-child relationship between different pieces of data-elements and thus enable us to arrange our data, record and files in a hierarchical manner. Consider a Tree, which represents your family structure. Let us say that we start with grandparent; then come to your parent, sons and daughter of grandparent and finally, you and your sisters and brothers. In this unit, we will go through the elementary tree structures initially, and then we will discuss more popular three structures like binary-trees, binary search trees, AVL tree and B-Trees.

1.2   TREES – AS ABSTRACT DATA-TYPE

Definition: A set of data-elements and related operations that are precisely stated independent of any particular implementation.

Since the data-elements and operations are defined with mathematical accuracy, rather than an implementation in a computer language, we may reason about effects of the operations, relationship to other abstract data types, whether a programming language implements the particular data type, etc.

1.3   TERMINOLOGIES

In this section, we will discuss some basic terminologies related to data structure tree:

1.3.1 Tree

A Tree is non-linear data structure. Tree can be defined as non-cyclic (Acyclic), connected graph. Here, the term graph can be defined as set of, set of vertices and set of edges. A tree data structure should have following properties:

➢ It is connected data structure as; all the nodes are reachable from any other node.

➢ It is non-cyclic data structure, therefore there must be exists only one and only one path is there from one node any other node. It does not have any cycle (close path or circuit).

➢ Tree can be constructed, with the help of nodes and edges which represents relationship between nodes.

➢ Data structure tree is a sub-set of Graph data structure. That means, every tree is a Graph but every Graph may or may not be a Tree.

Now, we will discuss some basic terms related to Tree data structure.

[1] Root Node:

A node from which data structure Tree starts is called a Root Node. Root node is node which does not have parent. Most general trees, which we are studying has one root node. Such type of tree structure, which has only one root node is called Rooted Tree.

[2] Edge:

Edge represents an association, between two nodes. In a tree data structures nodes are associated with an edge which represent parent and child relationship.

[3] Leaf node:

A node which does not have any child node is called a leaf node. Leaf nodes are also called as external nodes of the tree data structure.

[4] Siblings:

Two or more nodes which are derived from the same parent node is called siblings.

[5] Internal nodes:

All those nodes of the tree having at least one child node are called internal nodes of the tree. All non-leaf nodes are called internal nodes of the tree.

[6] Level of a node:

The level of a particular node is a number of edges on the path from that particular node to root node. The level of a root is always to be considered as 0.

[7] Depth of a node:

The depth of a particular node is the number of edges present in path from that particular node of a tree to the root node.

[8] Height of a node:

The height of a particular node is the number of edges presented in the longest path connecting that particular node to its leaf node.
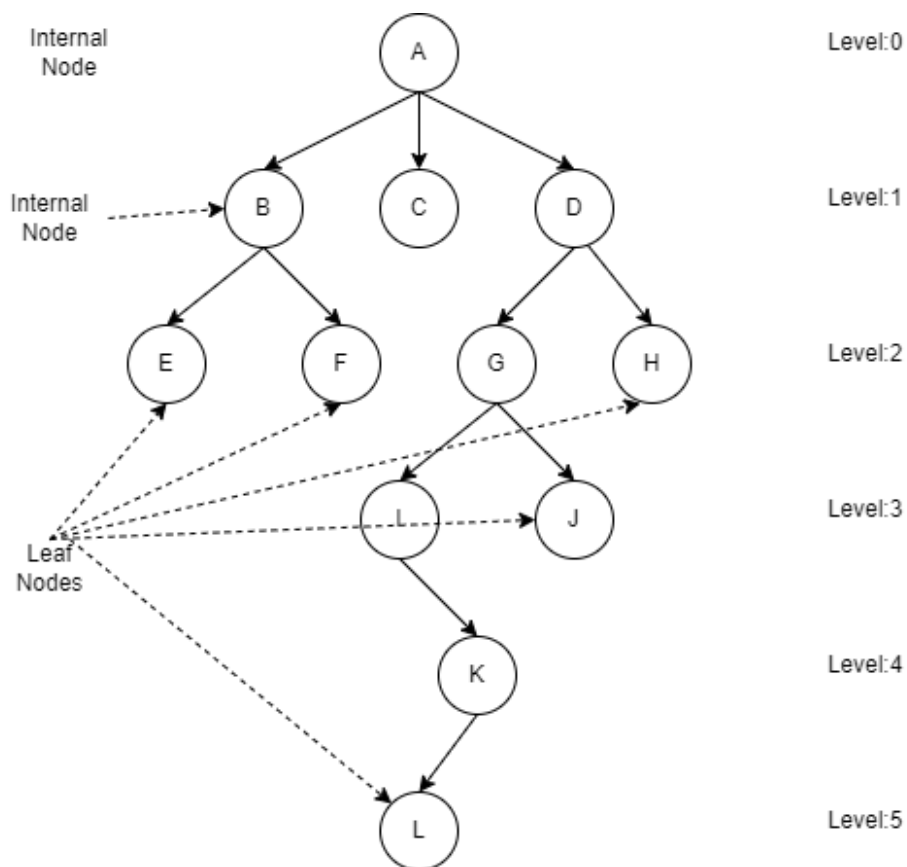
[Fig: 1.1 Tree Terminologies]

In the above figure [Fig:1.1] Node 'A' does not have any parent, so it is called the root node of the three. Node 'C', 'E', 'F', 'H' and 'L' does not have any child so they are called leaf nodes of the tree. Nodes 'B', 'C' and 'D' are derived from the same parent 'A'; therefore, they are called siblings. The Level of Root node 'A' is 0, Nodes 'B', 'C' and 'D' are at level 1 and so on. Apart from the leaf nodes all other nodes are called internal nodes. For example, nodes 'B' and 'D' are internal node. The hight of a tree (also referred as height of root node 'A') is 5. The depth of node 'G' is number of edges from root node 'A' to node 'G', which is 2. Similarly, the height of node 'G' is number of edges from node 'L' (leaf node at longest under node 'G') to node 'G' is 3. Solid lines with arrows represent the association between two nodes, which are called edges.

We hope the figure [Fig: 1.1] given above, has clear all of your terms we have discussed earlier in this unit.

Check Your Progress-1

1.Two or more nodes derived from the same parent node are called _____.

[A] leaf nodes   [C] siblings

[B] root node   [D] external nodes

2. A node which does not has any child is called _____.

[A] leaf node   [C] sibling

[B] root node   [D] external node

1.3.2 Binary Tree

Binary tree is a tree data structure in which all node must have at most two children. That means in binary tree no node should have more than two children. A tree represented in the [Fig:1.1] is not a binary tree, because node 'A' has three children 'B', 'C' and 'D'. But a tree represented in the following figure [Fig:1.2] is an example of Binary Tree.
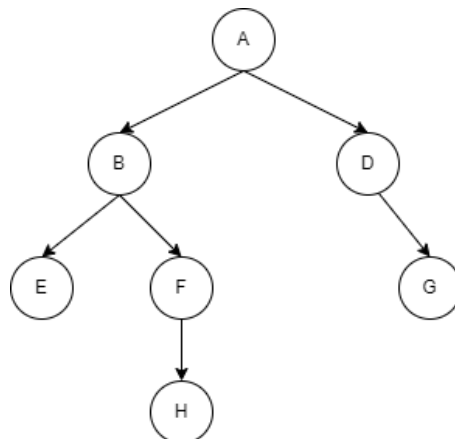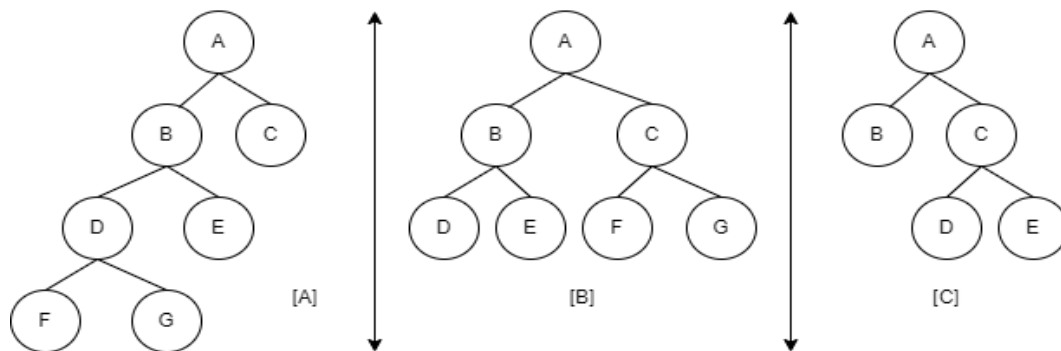

[Fig: 1.2 Binary Tree]

As we can see in the figure give above, there are no node is there in the tree which can have more than two children, and hence we can say that the tree structure represented in the above figure is a Binary Tree data structure.

1.3.3 Full Binary Tree

It is a binary tree (every node has at most children), in which every internal (parent) node has either no children (leaf node) or two children. It is also recognized as a proper binary tree. In the following figure [Fig:1.3] we have demonstrated a full binary tree.

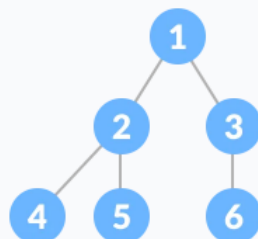**[Fig: 1.3 [A], [B], [C] Examples of Full Binary Tree]**

<u>**Properties:**</u>

➢ In Full binary tree, Maximum number of nodes should be $2^{h+1} - 1$. Where h is the height of the binary tree.

➢ A Full binary tree, should have minimum $(2*h - 1)$ nodes, where h is the height of the binary tree.

➢ The height of a full binary tree should be $\log_2(n+1) - 1$, where n is the total number of nodes in the binary tree
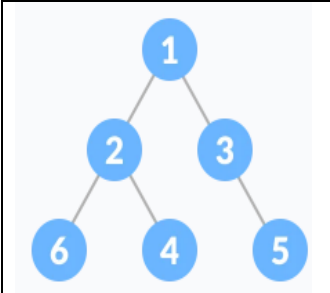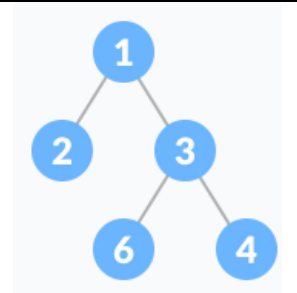
1.3.4 Complete Binary Tree
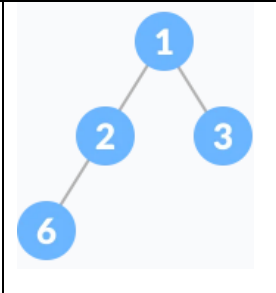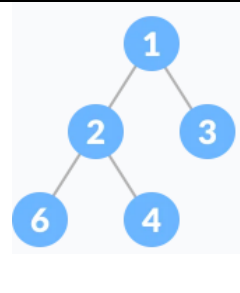
A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

A complete binary tree is just like a full binary tree, but with two major differences

1. All the leaf nodes must lean towards the left.
2. The last leaf node should not have a right sibling.

Check Your Progress-2

1. For the given binary tree, identify it is a Full Binary Tree or Complete Binary Tree:

[A]	[B]	[C]	[D]

1.3.5 Expression Tree

Binary trees can also be widely used to store and represent arithmetic expressions. A binary tree which represents an arithmetic expression is called expression tree. For Example, an arithmetic expression: (a – b) + (c * d) can be represented in the expression tree as follows:
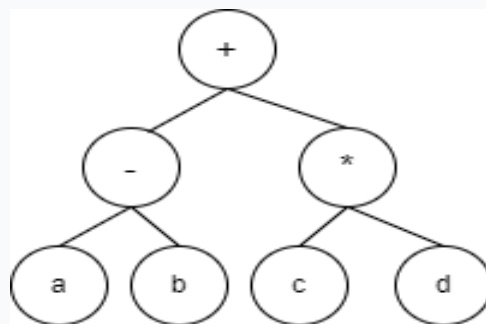
[Fig: 1.5 Expression Tree for the arithmetic expression: (a-b) * (c*d)]

Similarly, an arithmetic expression a + b / c * d – e can be represented in the form of expression tree as follows:
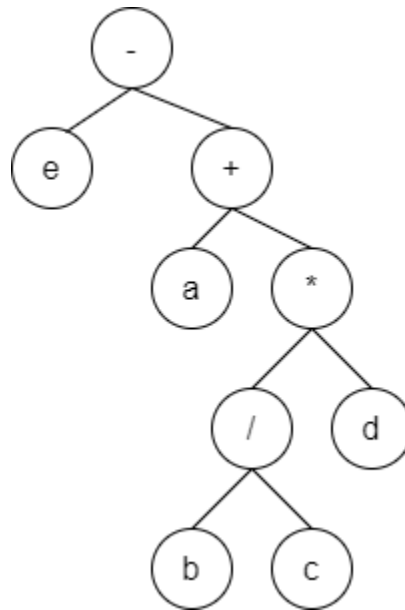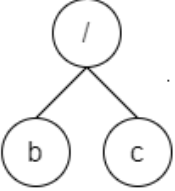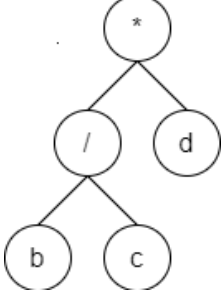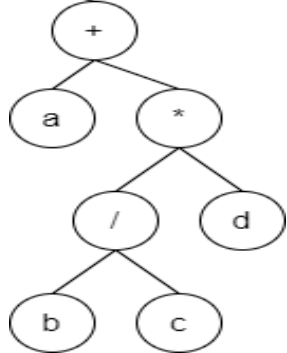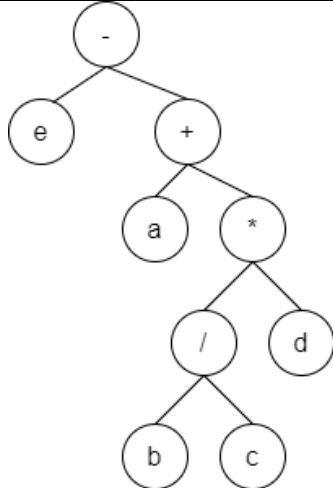
[Fig: 1.6 Expression Tree for the arithmetic expression: a + b / c * d - e]

Explanation:

To prepare the expression tree from the given arithmetic expression, you need to evaluate precedence (priority) of the operators used in the expression. In the given expression we know that multiplication (*) and division (/) operators have highest precedence that addition (+) and subtraction (-).

Now, in the multiplication (*) and division (/) operators both have same precedence, therefore associativity rule (Left to Right) is applied. In the expression, from left to right we are getting division (/) first. Therefore, we will represent b/c first.

Step:1	Represent b / c as the division has highest priority as per associativity rule.	
Step:2	After division the next highest priority of multiplication (*) operator. Therefore, we will represent (b/c) * d as given the right-side figure.	

Step:3	After considering multiplication (*) and division (/) operators, we have addition (+) and subtraction (-) operators remains. Both has same precedence. So as per associativity rule, because of addition (+) is at the left-hand side, we will give higher preference to it than subtraction. So, we will now consider: a + ((b/c) * d)	
Step:4	At the last, we have only one operator remains and that is subtraction (-). Which can be represented as follows: (a + ((b/c) * d)) - e	

1.4 BINARY SEARCH TREES – BST

Binary Search Tree (BST), is a practical implementation of the binary tree, which is usually intended to improve the searching process of data-element much faster. Specifically, in databases which stores huge amount of data in the secondary memory (Memory which store the data permanently – non-volatile). The speed of secondary memory is very slow compare to the speed of primary memory. Searching a particular record from the huge amount of data preserved in the database and also in secondary memory, which is slow in speed is very time-consuming operation.

To speed up the searching process in such scenarios, BST – Binary Search Tree is used. Now let us discuss how to implement the BST. What is the algorithm is there to implement BST?
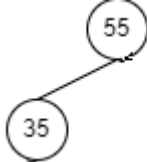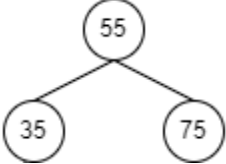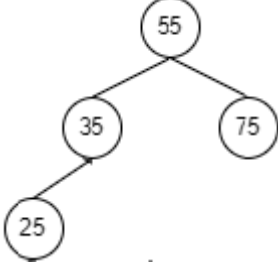
1.4.1 BST (Binary Search Tree) Implementation:
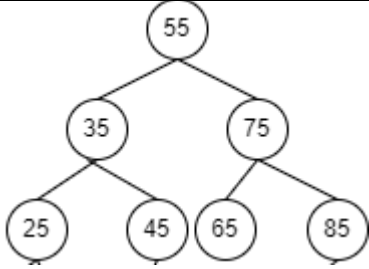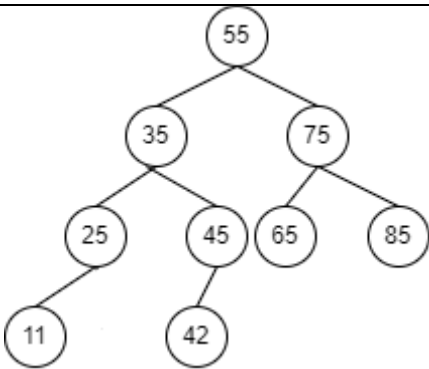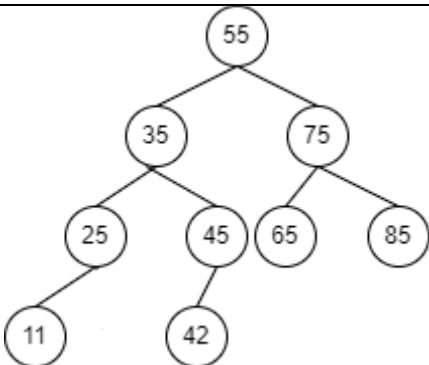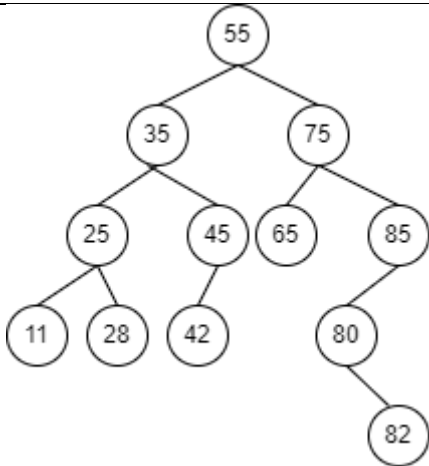
To implement Binary Search Tree, we need to follow the following steps:

1. When the first value is inserted by the user (when Binary Search Tree is empty), It will become the root node.
2. If the value inserted by the user and if the BST is non-empty, then this value will be compared with the value of root node, if the value to be inserted by the user is smaller that the value of root node, it will be inserted in the left-sub tree.
3. If the value inserted by the user and if the BST is non-empty, then this value will be compared with the value of root node, if the value to be inserted by the user is greater than the value of root node, it will be inserted in the right-sub tree.

Let us, understand how, the following values to be inserted in the Binary Search Tree:

55, 35, 75, 25, 45, 85, 65, 42, 11, 80, 85, 82, 28

Step	Description	BST
1	The first value inserted by the user is:55 Therefore, as per algorithm of BST it will become root node.	
2	The next value is 35, BST is non-empty therefore 35 is compared with the root value 55. Because 35 is lesser than 55, it should be inserted at the left side of 55.	
3	Similarly, third value 75 is greater than the value 55 (value of root node). Therefore, it should be inserted at the right side of it.	
4	The next value 25 is smaller than root value 55. That means it should be inserted at left-side of 55. But 55 already has its left value 35. Now, 25 will be compared with 35. Because 25 is smaller than 35, it is inserted at the left side of 35.	

5	The next value is 45, which is lesser than 55 and greater than 35. That means 45 should be inserted in the right side of 35. Similarly, next value 85 is greater than 55, and also greater than 75. Therefore, it should be inserted at the right side of 75. Next value in the sequence is 65, which greater than 55, but smaller than 75. Therefore, 65 is inserted at left side of 75.	
6	The next value in the sequence is 42, which is less than 55, greater than 35 and less than 45. Therefore value 42 is inserted at the left side of 45. The next value in the sequence is 11, which is less than 55, 35 and 25. So it is inserted at left side of 25.	
7	Next value in the sequence is 80, which is greater than 55 and 75, but smaller than 85 and hence it is inserted at the left side of 85. *Next value is 85. Which is already exists in the tree and hence it is discarded.*	
8	Next value 82, which is greater than 55, greater than 75, but smaller than 85 and hence it is inserted at left side of 85. Last value 28, is smaller than 55, smaller than 35, but greater than 25. Therefore, it is inserted at right side of 25. After inserting all the values, a Binary Tree looks like as given in the figure at right.	

By understanding the example discussed above, you can easily understand how, to develop a binary search tree from the given data.

1.4.2 Searching in BST:

Now, let us discuss why we need to store, data-elements in the binary search tree (as it is complicated) instead of arrays (simple linear data structure). To understand this let us assume that all elements of the above examples are stored in the array:

55, 35, 75, 25, 45, 85, 65, 42, 11, 80, 82, 28

Suppose, if we want to search an element 28, in the array then first we need to start comparing it with 55 (first element). Now, the first element of the array is not 28, we need to compare it second element which 35 (not 28), So we will again compare 28 with third element, and in that way wee need to compare 28 sequentially with all the data-elements of an array. We know that 28 is last data-element of the array. After 12 comparisons we will be able to locate our value (28) in the array.

Now, think the data-elements are stored in the form of BST as shown below:
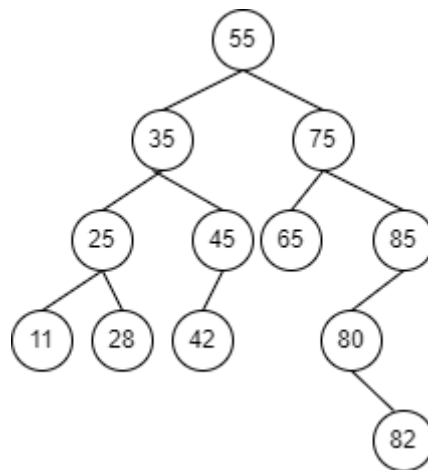
[Fig: 1.7 Data-elements in Binary Search Tree]

First, we will compare 28 with the root node that is 55. Now because of 28 is smaller than 55, you need to search 28 in the left subtree of 55. Which means in one comparison 55, 75, 65, 85, 80, and 82 values will be out from the comparison.
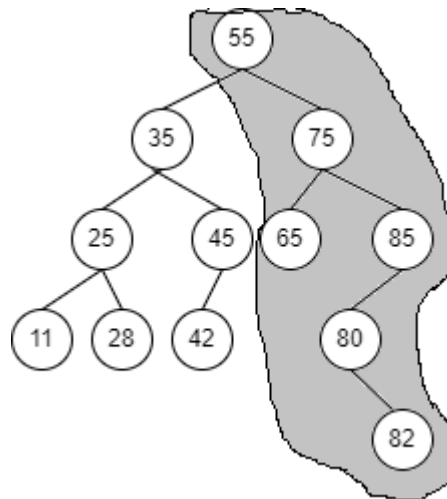
[Fig: 1.8 Data-elements in the grey region will be out of comparison]

Now, we will compare 28 with 35, in the second comparison. Because 28 is smaller than 35, values of right sub-tree of node 35 (45 and 42) will be out from the comparisons. That means in the second comparisons 35, 45 and 42 values are eliminated from the searching process.

Now, in the third comparisons, we will compare 28 with 25. Because of 28 is greater than 25, it has to be in right sub-tree of 25. In this comparation values 25 and 11 are eliminated from the comparisons.

In the fourth comparisons, we will be able to locate 28. So, from this example we can say that if the data-elements or records are stored in binary search tree then we can easily find them with few numbers of comparisons, and hence it enhances the searching speed. If a binary tree is properly balanced then 13 to 14 comparisons are sufficient to find a value from few lacks of data-elements, and 17 to 18 comparisons are sufficient to find a data-element from corers of values.

1.5   TRAVERSAL IN BINARY SEARCH TREE

We have discussed about traversal in Link-List. Because of they are linear data structure, it has only one method display() which will visit all the nodes and the value in each node is displaying on the console screen. BST is non-linear data structure therefore

there are three methods are there to visit all the nodes. Traversal methods for Binary Search Tree are:

1.5.1 Pre-order Traversal

In the pre-order traversal, you need to visit Root node first then you need to visit Left sub-tree and finally you need to visit Right sub-tree. That means the visit sequence in pre-order traversal: **Root – Left – Right**.

Consider the tree given below in the figure, and write its pre-order visit sequence.

To do pre-order traversal you need to start from the root node of the tree. Here root node is 55. As per pre-order visit sequence you need to visit root node, therefore you need to visit:**55**.

Now after visiting a root node, you need to visit Left sub-tree (Tree of 35) again you have to visit root node of it that is **35**.

After visiting 35 as root node, you need to visit it's left sub tree (a tree whose root is 25). In this tree you need to visit root node **25,** then left node **11**, and then right node **28**.

After visiting all the nodes of sub-tree of 25, go back to sub-tree of 35. You have visited 35 (root), you have

Also visited 25(left) so now you have to visit right sub-tree (tree of 45). In this tree you need to visit **45** (root node) and **42** (left node). Sub-tree of 45 does not have right node.

Similarly, again you go back think of entire tree (Tree of 55). In this three you have already visited 55 (root) and 35 (left). Now you have to visit right sub-tree (Tree of 75). In this tree you have to visit **75** (root) and then **65** (left). After visiting root and left come to visit right sub-tree (Tree of 85). In this you need to visit **85** (root) and then you need to come to the left sub-tree (Tree of 80). In this tree, you need to visit **80** (root), there is no left node is there and then right node that is **82.**

So, final Pre-order visit sequence is: 55, 35, 25, 11, 28, 45, 42, 75, 65, 85, 80, 82

1.5.2 In-order Traversal

In the in-order traversal, you need to visit Left sub-tree first then you need to visit Root node and finally you need to visit Right sub-tree. That means the visit sequence in in-order traversal: **Left – Root – Right**. The in-order visit sequence for the binary search tree given

above is: 11, 25, 28, 35, 42, 45, 55, 65, 75, 80, 82, 85. As you can see that In-order traversal of a binary search tree provides all the data-element to be arranged in ascending order.

1.5.3 Post-order Traversal

In the post-order traversal, you need to visit Left sub-tree first then you need to visit Right sub-tree and finally you need to visit Root node. That means the visit sequence in post-order traversal: **Left – Right – Root**. The post-order visit sequence for the binary search tree given above is: 11, 28, 25, 42, 45, 35, 65, 82, 80, 85, 75, 55. As you can see that In-order traversal of a binary search tree provides all the data-element to be arranged in ascending order.

Check Your Progress-3

1.From the given below options, identify correct traversal method(s) for Binary Search Tree.

[A] in-order   [C] post-order

[B] pre-order   [D] All of the above

2. In _____ traversal method of Binary Search Tree, we get the data-elements in ascending order.

[A] in-order   [C] post-order

[B] pre-order   [D] None of the above

3.Identify the correct visit sequence for post order traversal.

[A] Left-Right-Root   [C] Left-Root-Right

[B] Right-Left-Root   [D] Root-Left-Right

1.6 LET US SUM UP

In this unit, we have:

- Discussed about tree data-structure and its definition
- Elaborated different types of tree structure.
- Talked about Binary Search Tree (BST)
- Explained construction of BST and its traversal methods.

1.7 SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

7. [C] Siblings
8. [A] Leaf node

Check Your Progress-2

6. [A] Not Full binary tree nor Complete binary tree
7. [B] Full binary tree, but not Complete binary tree
8. [C] Not Full binary tree, but Complete binary tree
9. [D] Full binary tree as well as Complete binary tree

Check Your Progress-3

6. [D] All of the above
7. [A] In-Order
8. [A] Left – Right – Root

1.8 GLOSSARY

7. **Tree:** A tree is a non-cyclic data structure which represent the data in hierarchical manner. Non-Cyclic graph (set of, set vertices (nodes) and set of edges).
8. **Root node:** A node from which a tree structure starts is called root node and that tree is called rooted tree. Root node does not have any parent.
9. **Leaf node:** A node which does not have any child is called a leaf node,
10. **Siblings:** Two or more nodes having same parent (derived from similar parent) are called siblings

1.9 Assignment

5. Draw Binary Tree if the following traversal details are given.
   In-order: 11, 25, 28, 35, 42, 45, 55, 65, 75, 80, 82, 85

   Pre-order: 55, 35, 25, 11, 28, 45, 42, 75, 65, 85, 80, 82

1.10 Activity

2. Write a function to insert a value in the BST.
3. Write functions to perform in-order, pre-order and post-order traversal of BST.

1.11 Case Study

2. Try to find out area of applications where BST is used on the Internet.

1.12 Further Reading

- Data Structure through C by Yashvant kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.

Unit 2: Advanced Trees 2

Unit Structure

2.0   LEARNING OBJECTIVES

After working through this unit, you should be able to:

- Implement Binary Search Tree
- Understand AVL-Tree
- Know about B-Trees

2.1   INTRODUCTION

In the previous unit, we have seen tree data structure. We have also discussed about Binary Search Tree (BST) and its traversal methods. In this Unit, we will focus on how can we practically implement insertion and traversal in BST. Along with that what is the problem is BST, so that we have developed AVL tree and finally we will discuss about B-Trees.

2.2   IMPLEMENTATION OF BINARY SEARCH TREE

Generally, BST can be implemented by implementing insert() function, and to traverse BST three traversal functions pre-order(), in-order() and post-order(). Let us discuss insert() function first:

2.2.1 Insertion in Binary Search Tree:

To insert the value into the BST, we need to use the same data-structure node which we have used in the doubly Link-List. We need to define a structure node, in which *int data;* (To store user's data), *struct node *lptr*; (which will store the address of the root node of its left-sub tree) and *struct node * rptr*; (which will store the address of the root node of its right-sub tree) as follows:

struct node
 	
{
 	
       *struct node *lptr;*
 	
       int data;
 	
       *struct node *rptr;*
 	
};
	Node
	
	[lptr | Data | rptr]
	
	Address of the Root node of Left sub-tree User's Data Address of the Root node of Right sub-tree
	
	[Fig: 2.1 Node structure of BST]
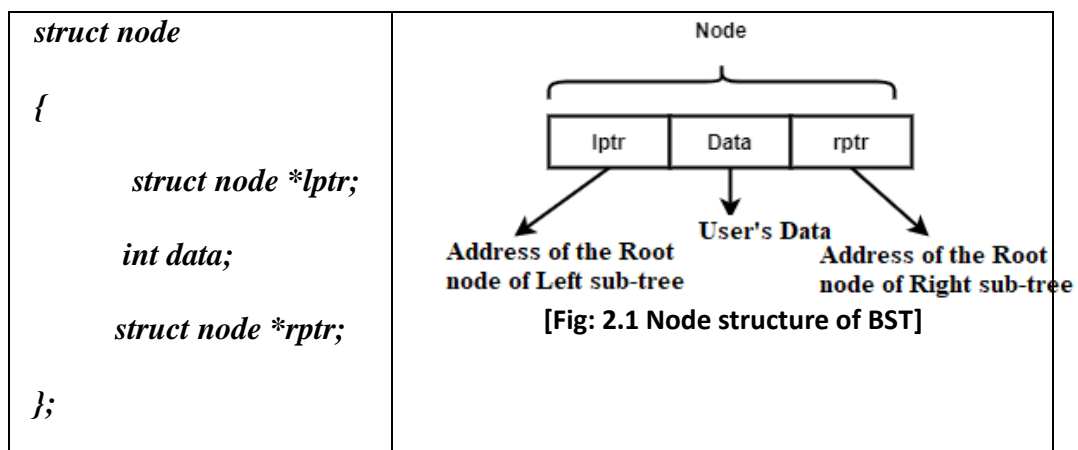
Similar, to the doubly Link-List, we will create a pointer which stores the address of the first node. In the tree, first node is a root node. Therefore, we will declare a pointer variable ***struct node *root;*** which will be initialized with NULL value.

To implement insert() function, following code we need to write:

```c
#include<stdio.h>
/***********Declaring Structure Node **********/
struct node
{
  struct node *lptr;
  int data;
  struct node *rptr;
} *root = NULL;
/********** Inserting value to BST **********/
void insert (int val)
{
  struct node *p, *c, *n;
  int d;
  n = (struct node *) malloc (sizeof (struct node));
  n->data = val;
  n->lptr = n->rptr = NULL;
  if (root == NULL)
  {
    root = n;
    return;
  }
  c = root;
  while (c != NULL)
  {
    if (val < c->data)
    {
      p=c;
      c=c->lptr;
      d=0;
    }
    else if (val > c->data)
    {
      p=c;
      c=c->rptr;
      d=1;
    }
    else
    {
      printf ("\nValue already exists:");
```

```
        return;
    }
  }
  if (d == 0)
  {
    p->lptr = n;
  }
  else
  {
    p->rptr = n;
  }
}
```

In the insert function, we have declared three variables of type 'struct node *'. Point variable *n is used to create a new node. It will always point to the new node, whereas, *c and *p is used to find out appropriate position where new node has to be fitter. Variable int d, represents direction. When the value to inserted in the tree is less than the data value pointed by *c then direction d=0 (left side). But if the value to inserted in the tree is greater than the data value pointed by *c the direction d=1 (right side). If the root node is having NULL, means this is the first value entered by the user and we will copy the address of new node (*n) into *root.

If three is not empty (*root != NULL), we will copy the address of root node into *c and start the loop till *c does not becomes NULL. Pointer variable *p, points to the parent node of *c.

When pointer *c will become NULL, at that time, if d=0 then the address of new node has to be placed in the lptr section of the parent node, otherwise the address of the new node has to be placed in the rptr section of the parent node (*p).

2.2.3 Traversal in BST

As we have discussed there are three traversal algorithms are there. All those traversal methods, their visit sequence and code to implement that traversal methods are discussed below:

[1] Pre-Order Traversal:

In the Pre-Order traversal, we need to visit Root node first, then left sub-tree and then right -sub tree. The implementation of Pre-Order traversal is given below:

```
void preorder(struct node *r)
{
  if(r==NULL)
    return;
  printf("%d--", r->data);
  preorder(r->lptr);
```

```
    preorder(r->rptr);
}
```

[2] In-Order Traversal:

In the In-Order traversal, we need to visit left sub-tree, then Root node and then right -sub tree. The implementation of In-Order traversal is given below:

```
void inorder(struct node *r)
{
   if(r==NULL)
      return;
   inorder(r->lptr);
   printf("%d--", r->data);
   inorder(r->rptr);
}
```

[3] Post-Order Traversal:

In the Post-Order traversal, we need to visit left sub-tree, then right sub-tree and finally Root node. The implementation of Post-Order traversal is given below:

```
void postorder(struct node *r)
{
   if(r==NULL)
      return;
   postorder(r->lptr);
   postorder(r->rptr);
   printf("%d--", r->data);
}
```

The source code of implementation of Binary Search Tree, with Insert, Preorder, Postorder and Inorder functionalities are given below:

```
#include<stdio.h>
/***********Declaring Structure Node **********/
struct node
{
   struct node *lptr;
   int data;
   struct node *rptr;
} *root = NULL;
/********** Inserting value to BST *********/
void insert (int val)
{
   struct node *p, *c, *n;
   int d;
   n = (struct node *) malloc (sizeof (struct node) );
```

```c
      n->data = val;
      n->lptr = n->rptr = NULL;
      if (root == NULL)
      {
         root = n;
         return;
      }
      c = root;
      while (c != NULL)
      {
         if (val < c->data)
         {
            p = c;
            c = c->lptr;
            d = 0;
         }
         else if (val > c->data)
         {
            p = c;
            c = c->rptr;
            d = 1;
         }
         else
         {
            printf ("\nValue already exists:");
            return;
         }
      }
      if (d == 0)
      {
         p->lptr = n;
      }
      else
      {
         p->rptr = n;
      }
}
/*********** Pre-order Traversal *********/
void preorder (struct node *r)
{
   if (r == NULL)
      return;
   printf ("%d--", r->data);
   preorder (r->lptr);
   preorder (r->rptr);
```

155

```c
}
/*********** In-order Traversal *********/
void inorder (struct node *r)
{
   if (r == NULL)
      return;
   inorder (r->lptr);
   printf ("%d--", r->data);
   inorder (r->rptr);
}
/*********** Post-order Traversal *********/
void postorder (struct node *r)
{
   if (r == NULL)
      return;
   postorder (r->lptr);
   postorder (r->rptr);
   printf ("%d--", r->data);
}
/********** Function Main ****************/
void main()
{
   int ch, val;
   do
   {
      printf ("\n/----------MENU----------/");
      printf ("\n/ 1. Insert:");
      printf ("\n/ 2. Pre-Order:");
      printf ("\n/ 3. In-Order");
      printf ("\n/ 4. Post-Order:");
      printf ("\n/ 5. Exit:");
      printf ("\n/-----------------------/");
      printf ("\n Enter Your Choice:");
      scanf ("%d", &ch);
      if (ch == 1)
      {
         printf ("Enter Value:");
         scanf ("%d", &val);
         insert (val);
      }
      else if (ch == 2)
      {
         printf ("\n");
         preorder (root);
      }
```

156

```
    else if (ch == 3)
    {
      printf ("\n");
      inorder (root);
    }
    else if (ch == 4)
    {
      printf ("\n");
      postorder (root);
    }
    else if (ch == 5)
    {
      printf ("Good Bye:");
      break;
    }
    else
    {
      printf ("\nInvalid Choice:");
    }
  } while (ch != 5);
}
```

2.2.3 Representation of BST in Memory:
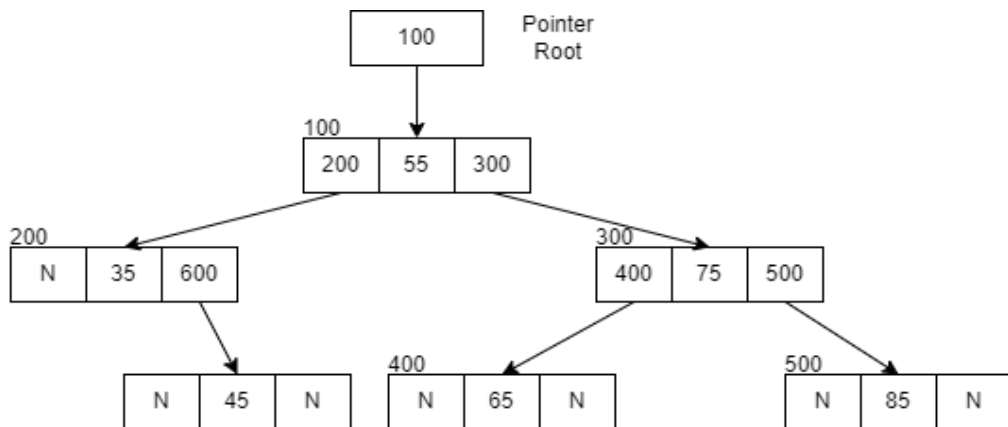
[Fig: 2.2 Memory representation of BST]

In the above figure [Fig:2.2], we have represented BST in the memory. You can see, root pointer is having address 100, where our root node is there. In the lptr of the root node address 200 is there which pointes to the node in which 35 is there in the data part. Similarly, in the rptr section of the root node, we have 300, which points to a node

in which 75 is there. All of those nodes in which lptr and rptr sections are NULL (represented as N in the figure) are called leaf node.

2.3   AVL-TREE

The name AVL-Tree is given from its inventors **A**delson,**V**elsky and **L**andis. AVA-tree and hight balanced binary search tree. After reading the previous statements you get the idea that – AVL tree is a binary tree, which maintain height of its left sub-tree and right sub-tree to be balanced. But the question is: Why we need a tree data structure, which balance its height? Let us try to understand it:

2.3.1 Need of AVL-Tree:

BST is invented for the purpose, that it enhances searching capabilities by storing smaller values in the left side of the tree and larger values at right side of the tree. At the time of searching, if the value to be search is smaller then we need to look into left sub-tree and all the values of right sub-tree including root will be eliminated from the comparison process. Similarly of the value to be searched is larger then, we need to look into right sub-tree, and all values in the left-sub-tree including root node will be eliminated from the comparison.

In each comparison number of candidates, will be halved, which enhances the searching process. But now think of the following scenarios, where user insert the following values:

5, 10, 15, 20, 25. By following BST algorithm if you insert all these values in the sequence given, then you will get the tree as follows:

  [Fig: 2.3 Inserting values in ascending order in BST]	Here the first value 5 will become root. 10 is greater than 5, so that it is inserted at right side of 5. 15 is again greater than 5 and 10, therefore it is inserted at right side of 15 and so on. By looking to the figure, we can say that out BST is formed in linear shape (instead of binary shape). The tree demonstrated in the figure at left is called Right-Skewed Tree.

Similarly, if you enter the values in the descending order then it form a Left-Skewed tree. The problem with skewed tree is, it does not eliminate any value in the searching process. To search 25 you need to make 5 comparisons (same as array).

By understanding the nature of the problem described above, you can easily determine that, problem occurs because of, BST has loosened its shape and turned into linear structure. To maintain the shape of BST, we need to develop an algorithm, which maintain the shape of BST. And AVL-tree, a height balanced tree comes into the picture. Let us see, how the insertion process will be done in the AVL-Tree.

2.3.2 Inserting value to AVL-Tree:
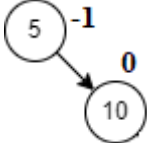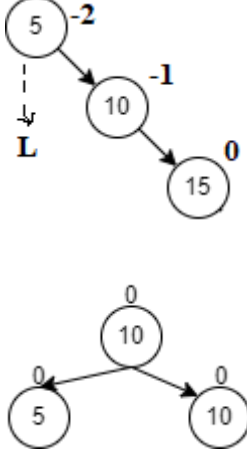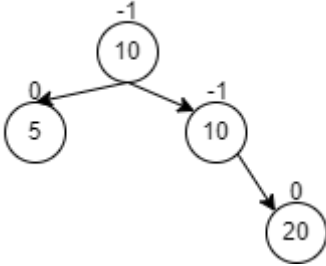
Insertion into an AVL-Tree has to be done in the same way as done in the BST. First value inserted by the user will become root node. Smaller values have to be inserted in the left-sub tree, and greater values are inserted in the right sub-tree.

In the AVL-Tree, every node of the tree has additional field called balance factor. Balance factor is nothing but the integer variable. Every time after inserting a new value balance factor has to be computed using following formula:

Balance factor of the node = height pf its left sub-tree – Height of its right sub-tree

If the balance factor of any node is -1, 0 or +1 then that node is called balanced node and if all the nodes of a tree are balanced then that tree is called height balance tree or AVL-Tree.

If any node has balance factor <=-2 or >=+2 then that node is not balanced and to balance it, either one or two rotations are required. We will discuss rotations of AVL-Tree in more details. But before discussing those rotation rules let us see how balancing is done in the BST we have designed in figure [Fig:2.3].

Inserting value	Tree Structure	Explanation
5		After inserting first value 5, tree looks as shown in the figure. Because there is no left-sub tree and no right sub-tree, therefore the balance factor is 0.
10		After inserting 10 at right-side (10 > 5) again the balance factor is computed for both nodes. Balance factor of 10 is 0 (0-0), and balance factor of 5 is -1 because (Height of left tree is 0 – Height if right tree 1). Still tree is balanced.
15		After inserting 15, the balance factors for 15 is 0, 20 is -1 and 5 is -2. Balance factor out of -1, 0 and 1 is not allowed as the tree is not balanced. Here, tree need a rotation. Here balance factor of 5 is -2, therefore node 5 is rotated in Left (anti-clock) direction as shown in the figure.
20		After inserting 20 the balance factor (height of left sub-tree – height of right sub-tree) of each node is displayed on top of every node. No node violates the rules to be balanced as all nodes are having

		balance factor in valid range that is -1, 0 or 1.
25		After inserting 25, balance factor to be count from 25 (leaf node) to 10 (root node). Balance factor for 25 is 0, Bal Factor of 20 is -1, and bal. fact of 10 is -2. Which is invalid. To balance this tree, you need to give Left rotation to node 10. After giving left rotation to node 10 tree will becomes balanced as shown in the figure below.
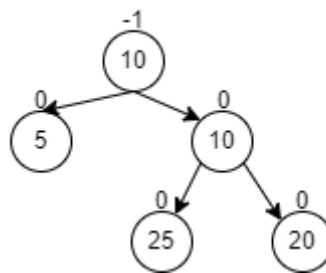
[Fig: 2.4 AVL-Tree]

Check Your Progress-2

1. From the given below option, which value for the balance factor is invalid for an AVL-Tree node?

[A] -1   [C] 0

[B] 1   [D] 2

2. From the given below, _____ is hight balanced tree.

[A] Tree   [C] Binary Search Tree

[B] AVL Tree   [D] None of the above

2.3.3 Rotations in AVL-Tree:

As we have discussed that the after inserting a value into the AVL-Tree if any node violates its balance factor (balance factor of the is either <=-2 or >=2), then we need to give rotation to the tree such that, the tree becomes balanced tree. But how many rotations we need to give? Which type of rotation we need to give Left rotation or Right rotation? Let us discuss it.

To give proper rotation(s) to imbalanced tree, first try to find out the nature of the problem. To understand the nature of the problem, apply following algorithm:

Step:1 Insert the new (given) value to the AVL-Tree by using insertion algorithm of BST.

Step:2 Count the balance factor of each node from newly inserted node (which must be leaf node having balance factor 0) toward root node.

Step:3 If you notice any node which violate the balance factor and go to 2-steps from that node to newly inserted node. Check the two edges from the node which violate balance factor to newly inserted node. If they are Left-Left then it possesses LL-Problem, if it is in Left-Right then it possesses LR-Problem. If it in Right-Left direction then it is RL-Problem and if it is in Right-Right direction, then it possesses RR-Problem.

Let see all one by one with an example:

Case:1 LL-Problem:

Consider the following AVL tree having two nodes are inserted and those nodes are 20, 10, which is shown in the figure [Fig:2.5(a)]. Let us insert 5 to this tree. After inserting 5 to AVL tree, balance factor of each node is represented in the Figure [Fig:2.5(b)].
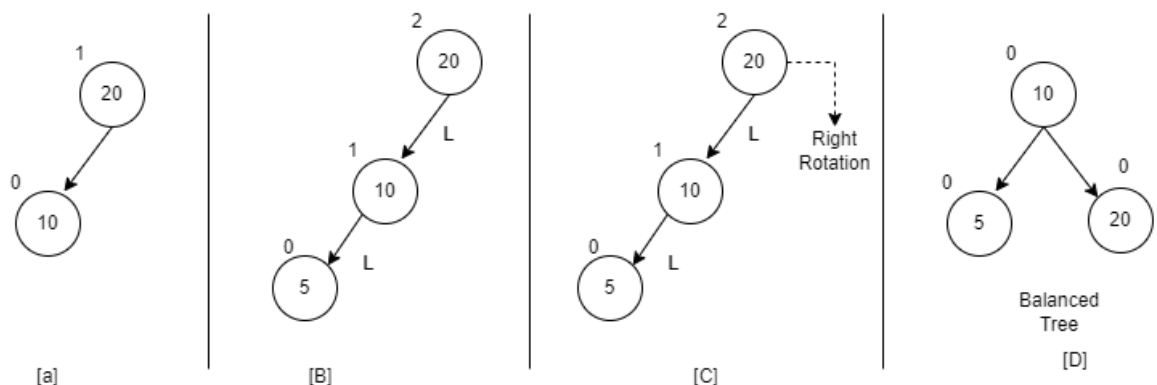
[Fig: 2.5 LL problem and its solution in AVL-Tree]

Data value 5 is inserted in the left direction of 10. Now we need to count the balance factor for each node from 5 to 20. The balance factor for 5 is 0, balance factor for 10 is 1 (one node on Left side and no node at right side), balance factor of 20 is 2 (hight of left sub-tree is 2 – height of right sub-tree is 0). Here, 20 has violated its balance factor. Now we will go 2-steps from 20 (who has violated balance) towards 5. We know that 5 is in the Left-Left side. This problem is called LL Problem.

The solution to LL problem is single Right rotation to 20 (who has violated balance factor). After giving Right rotation the AVL-Tree becomes balanced as shown in the figure [Fig:2.5(d)].

Case:2 RR-Problem:

Consider the following AVL tree having two nodes are inserted and those nodes are 10, 20, which is shown in the figure [Fig:2.6(a)]. Let us insert 35 to this tree. After inserting 35 to AVL tree, balance factor of each node is represented in the Figure [Fig:2.6(b)].

[Fig: 2.6 RR problem and its solution in AVL-Tree]

Data value 35 is inserted in the right direction of 20. Now we need to count the balance factor for each node from 35 to 10. The balance factor for 35 is 0, balance factor for 20 is -1 (one node on Right side and no node at Left side), balance factor of 10 is -2 (hight of left sub-tree is 0 – height of right sub-tree is 2). Here, 10 has violated its balance factor. Now we will go 2-steps from 10 (who has violated balance) towards 35. We know that 35 is in the right-Right side. This problem is called RR Problem.

The solution to RR problem is single Left rotation to 10 (who has violated balance factor). After giving Left rotation the AVL-Tree becomes balanced as shown in the figure [Fig:2.6(d)].

Case:3 LR-Problem:

Consider the following AVL tree having two nodes are inserted and those nodes are 20, 10, which is shown in the figure [Fig:2.7(a)]. Let us insert 15 to this tree.

After inserting 15 to AVL tree, balance factor of each node is represented in the Figure [Fig:2.7(b)].

[Fig: 2.7 LR problem and its solution in AVL-Tree]

In figure [Fig: 2.7 (a)] two values 20 and 10 are inserted. Node 20 is a root node and 10 is inserted at left side of 20 (10<20). Now in [Fig.2.7(b)] new value 15 is inserted. Because 15<20 (left sub-tree) and 15>10, 15 is inserted right side of 10. Now after insertion of 15, Balance factors of 15, 10 and 20 becomes 0, -1 and 2 subsequently. Here, node 10 violates the balance factor. Therefore, we need to go two steps from 20 to 15. It is Left-Right direction. This problem is called LR problem. The solution for LR problem is LR (Left and Right) rotations. Here we need to give two rotations to the tree to balance it. Which is explained below:

[1] Node 10 has Right problem; therefore, we need to give Left rotation to node 10. After giving Left rotation a tree looks same as shown in the [Fig:2.7(c)].

[2] Now, node 20 has Left problem, that means we need to give a Right rotation to node 20. After giving right rotation your tree looks like same as demonstrated in [Fig:2.7(d)].

Here, we have given Left rotation to 10, and Right rotation to 20 (Solution: LR).

Case:4 RL-Problem:

Consider the following AVL tree having two nodes are inserted and those nodes are 10, 20, which is shown in the figure [Fig:2.8(a)]. Let us insert 15 to this tree. After inserting 15 to AVL tree, balance factor of each node is represented in the Figure [Fig:2.7(b)].
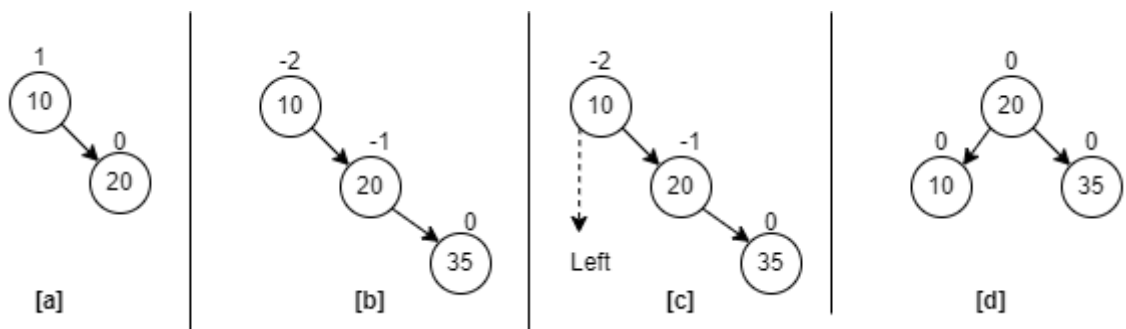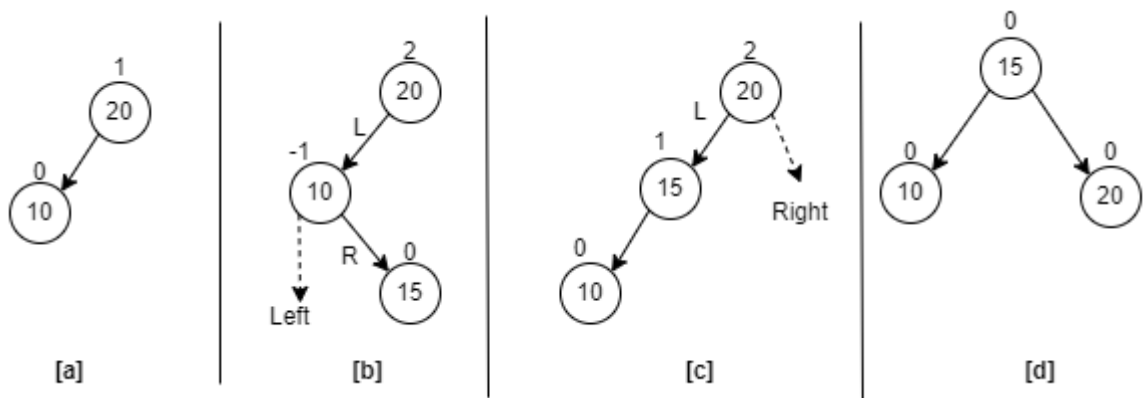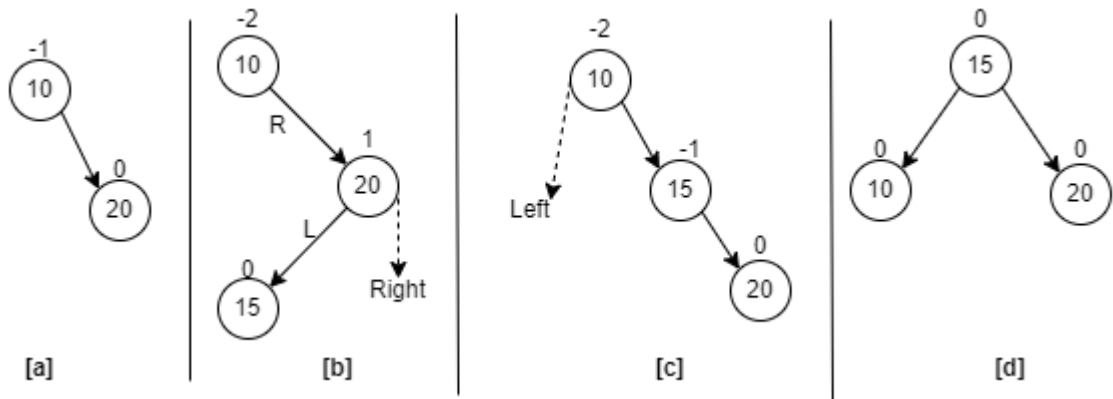
[Fig: 2.8 RL problem and its solution in AVL-Tree]

In figure [Fig: 2.8 (a)] two values 10 and 20 are inserted. Node 10 is a root node and 20 is inserted at right side of 10 (20>10). Now in [Fig.2.8(b)] new value 15 is inserted. Because 15>10 (right sub-tree) and 15<20, 15 is inserted left side of 20. Now after insertion of 15, Balance factors of 15, 20 and 10 becomes 0, 1 and -2 subsequently. Here, node 10 violates the balance factor. Therefore, we need to go two steps from 10 to 15. It is Right-Left direction. This problem is called RL problem. The solution for RL problem is RL (Right and Left) rotations. Here we need to give two rotations to the tree to balance it. Which is explained below:

[1] Node 20 has Left problem; therefore, we need to give Right rotation to node 20. After giving Right rotation a tree looks same as shown in the [Fig:2.8(c)].

[2] Now, node 10 has Right problem, that means we need to give a Left rotation to node 10. After giving left rotation your tree looks like same as demonstrated in [Fig:2.8(d)].

Here, we have given Right rotation to 20, and Left rotation to 10 (Solution: RL).

Check Your Progress-3

1. How many rotations are required in the AVL-Tree if we insert, 15, 20, and 10.

[A] 3   [C] 0

[B] 1   [D] 2

2. How many rotations are required in the AVL-Tree if we insert, 10, 20, and 15.

[A] 3   [C] 0

[B] 1   [D] 2

2.4 B-TREE

By understanding the topics, we have discussed in this unit, we have learnt that data structure tree is most suitable for searching of data-element from huge amount of data. Up to here, we have discussed how to search the data-element from the main memory, and for that we have discussed BST and AVL trees. Now, we will discuss a tree which can be implement on the databases. Usually, databases are created in the secondary memory. Compare to primary memory, secondary memories are slow in nature. Algorithms which can be used to search data-elements from the primary memory is called Internal Searching, whereas searching of data-elements from the secondary memory is called External Searching. B-Tree, B^+-Tree are example of External searching.

2.4.1 M-way (Multiway) Search Tree:

The idea of generalizing a tree to M-way tree is derived from the Binary Search Tree. In M-way tree for some integer value m called the order of the tree, each node can have at most m children. If $k \leq m$ is the number of children, then node contains exactly k-1 keys, which partitions all the keys in the subtrees into k subsets. If few of these subsets are empty, then the corresponding children in the tree are empty.

Our main goal is to find a multiway search tree which will reduces the file access, and for that purpose we want to keep the height of the tree as smaller as we can. Another important thing is, similar to AVL-tree, the height of the tree should be balanced. Therefore, we need to keep all leaf node at same level. Third important thing is all internal nodes should have some minimal number of children.

2.4.2 B-Tree and its Properties:

A B-Tree of order M is an M-way tree in which:

[1] All leaf nodes must be at same level.
[2] Except root node, all internal nodes should have at most m non-empty children, and at least $\lceil m/2 \rceil$ (celling value of m/2) non-empty children.
[3] The number of keys in each internal node is one less than the number of its non-empty children, and these keys divide the keys in the children in the fashion of a search tree.
[4] Root node has at most m children, but may have as few as 2 if it is not a leaf, or none if the tree consists of the root alone.

2.4.3 Insertion in B-Tree:

To understand the concept of B-Tree, let us discuss how data-elements are to be inserted in the B-Tree with an example.

Suppose we want to draw a B-Tree having values: 5, 10, 12, 13, 14, 1, 2, 3, 4 of order 3 then:

Our B-Tree is having order m=3, that means, each node can store m-1=3-1=2 data values in ascending order. Therefore, our structure node should have 2 data-elements and 3 pointers which will point to at most 3 sub-trees.

First value is 5 and second value is 10 which will be stored in the B-Tree as follows:

	5		10	
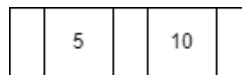
In the node, data-elements are 5 and 10, and all 3-address part having NULL values, as there is no sub-tree is there. Make sure, unless and until the node does overflow, we cannot split it and we can not start new level of the tree.

When we will insert 12, then node should have 3 values in order which are 5, 10, and 12. But in our case order m=3, which means each node can store maximum 2 data-elements. So, the node will be split from the median value. In 5, 10, 12 median is 10. So, a new root will have value 10. Node with data 5, will goes to at its extreme left side and 12 will be stored in another node which has to be at right side of 12.
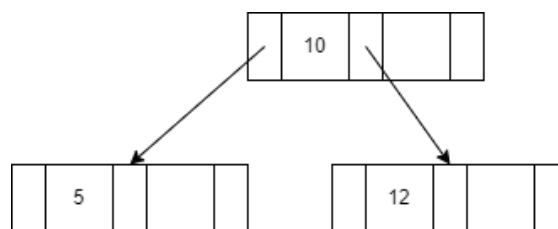
When we will insert 14 then it is greater than 10, so it will be added in the node in the right sub-tree of 10, and in the node where 12 is already there as shown in the following figure. Make sure, you can insert new values only in Leaf nodes.
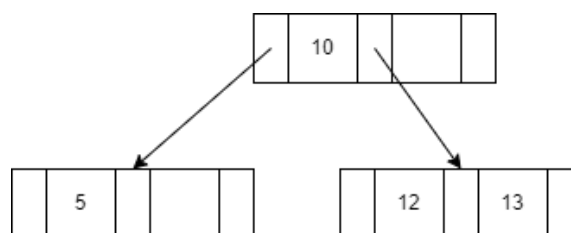
Now if we add 14, then 14 is greater than 12 and 13 therefore it should be added with 12 and 13, which makes a node should have 3 values 12, 13, and 14. But

because now our node is overflow (node can store only 2 values maximum) the node will be spited from the median value. In value 12, 13 and 14 median value is 13, so value 13 has to go in upward direction (in root node) which is demonstrates as below.

As shown in the figure node with value 5 should be extreme left-sub tree, because the value 12 comes between 10, and 13, it is in the middle sub-tree and value 15 is greater than 13, node with value 15 is stored as extreme right sub-tree.
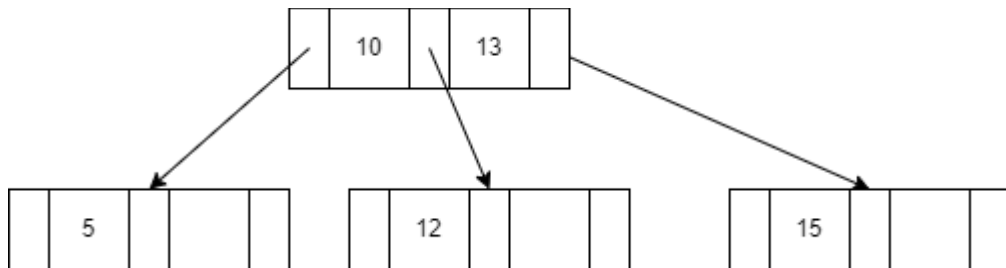
Now, when we will insert 1 then, it should be less than 5 and hence it is added into the node which already has 5 as data-element. Remember values in the node should be inserted in the ascending order as shown in the following figure.
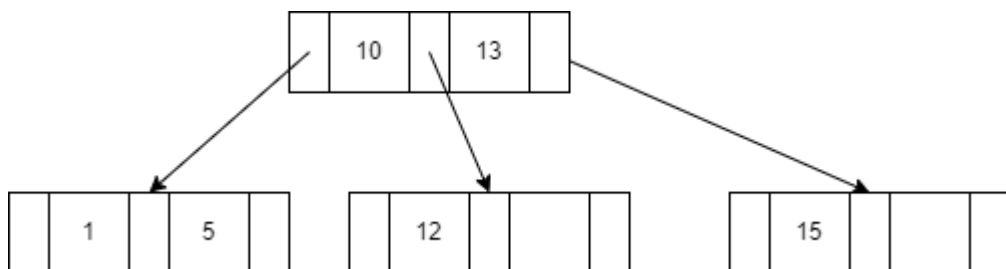
The next value is 2, we know that it has to be inserted in the node between 1 and 5. That makes values 1, 2 and 5 in the node. So, node will be split from median value 2. Value 2 need to go upward in the root node. Root node already has 10 and 13 and new value 2 inserted to it, which makes 3 values in root are 2, 10 and 13. Which will further split the root node and median value from 2, 10 and 13 that is 10 will becomes a new root. After insertion of data-element 2 out B-Tree looks like:
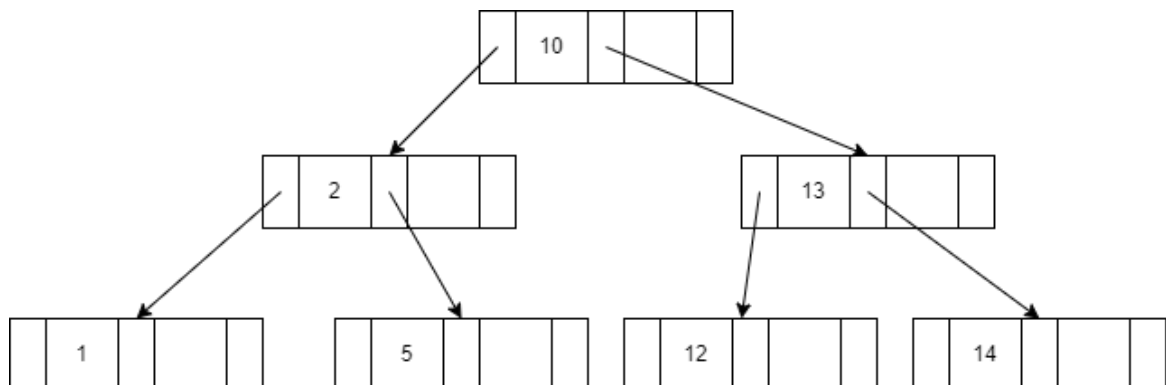
The next value is 3 which less than 10, and greater than 2. Therefore, it should be inserted in the leaf node having already 5 is there.
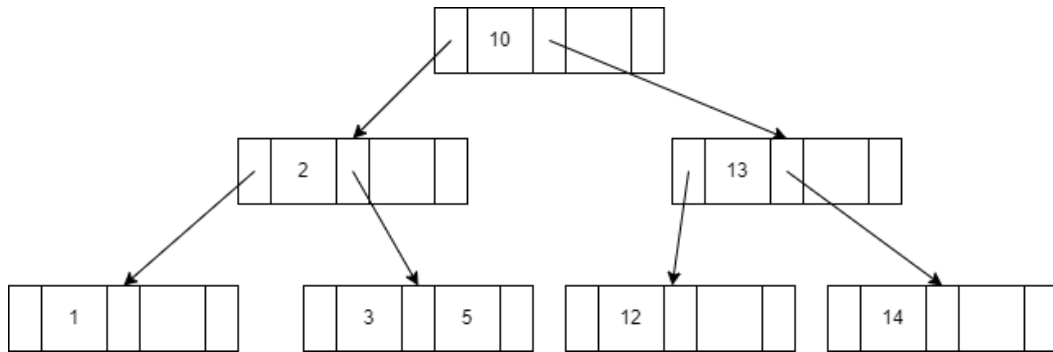
Now, when we add 4, it should be added in the leaf node which already have value 3 and 5. After adding value 4, node should have 3 values 3,4 and 5 which will overflow the node and node should be split from the median value which is 4. Here, data-element 4 will go upward and should be inserted in its parent node where 2 is there. After inserting all the values our B-Tree looks like:
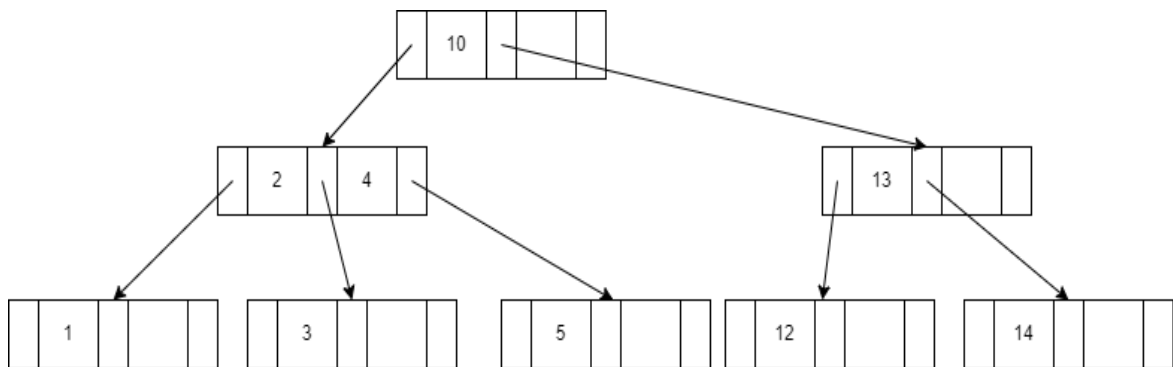
Check Your Progress-4

1. B-Tree is used for _____.

[A] Internal searching   [C] Internal Sorting

[B] External Searching   [D] External Sorting

2. From the given below _____ tree is an example of perfectly balanced tree.

[A] Binary Tree   [C] Binary Search Tree

[B] B-Tree   [D] AVL Tree

2.5 LET US SUM UP

In this unit, we:

- Have discussed implementation of BST.
- Have elaborated Binary Search Tree (BST) with Insertion and Traversal.

- Have discussed AVL-Tree and rotations in AVL-Tree.
- Have learn B-Trees.

2.6 SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

3. [B] 3
4. [A] Left

Check Your Progress-2

3. [D] 2
4. [B] AVL-Tree

Check Your Progress-3

3. [C] 0
4. [D] 2

Check Your Progress-4

3. [B] External Searching
4. [B] B-Tree

2.7 GLOSSARY

3. **Tree**- Tree is a non-linear data-structure. It is non-cyclic graph.
4. **BST:** BST stand for Binary Search Tree. It an implementation of Binary Tree where smaller values are inserted at left side and larger values are inserted at right side.
5. **AVL-Tree**: it is a roughly height balanced binary search tree. In every node if AVL-Tree should have balance factor. If balance factor is -1, 0 or 1, then it is to be considered as valid balance factor. If any node violates this balance factor, then rotation is required to maintain height of the binary tree.
6. **B-Tree**: It a M-way search tree, where M is the order of the tree. In B-Tree of M-way there are M sub-trees are there and M-1 data-elements can be stored in each node. It is perfectly balanced tree as all its leaf nodes must be at same level.

2.8   ASSIGNMENT

1. Explain Tree data-structure in details.
2. Explain different terminologies associated with tree data-structure.
3. Write a short note on AVL-Tree.

2.9   ACTIVITY

3. Explain how the values: 11, 6, 8, 19, 4, 10, 5, 17, 43, 49 and 31 should be inserted in the BST. Also perform its pre-order, in-order and post-order traversal.
4. Explain how values: 9, 15, 20, 8, 7, 13, and 10 are inserted in AVL-Tree.

2.10   CASE STUDY

*   Design a B-Tree of order 4 with following values:

5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

2.11   FURTHER READING

*   Data Structure through C by Yashvant kanetkar.
*   Data Structures Using "C" by Tanenbaum.
*   Data Structures and Program Design in "C" by Robert L. Kruse.

Unit 3: Graphs **3**

Unit Structure

3.0   Learning Objectives

After learning this unit, you will be able to:

- Understand, what is Stack?
- Understand methods of stack.
- Understand the Applications of Stacks.
- Implement Stacks.

3.1   Introduction

In this unit, we are going to discuss an important data structure called Graph. In fact, graph is a cyclic structure with no parent-child relationship. Graphs have many applications in the field of computer science and other fields of science. In general, graphs characterize a relatively less preventive relationship between the data items. We shall discuss about both undirected graphs and directed graphs. The unit also provides information of different algorithms which are based on graphs.

3.2   Definitions

Graph -

A graph **G** can be defined as a finite set of set **V** of vertices and a set **E** of edges (pair of connected vertices). The representation used is as follows:

Graph G = {V, E}

Consider the following figure to understand the definition of Graph.
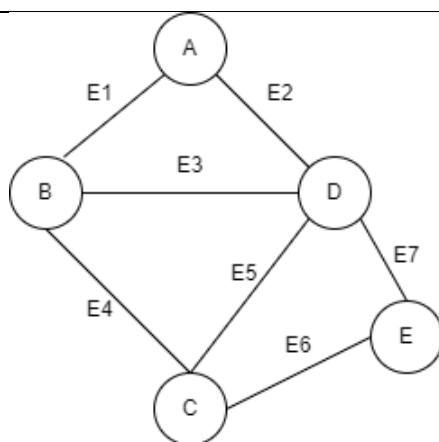
[Fig: 3.1 Graph]

As discussed in the definition, Graph G can be defined as:

G = {V, E}

Where, V = {A, B, C, D, E}

And, E= {E1, E2, E3, E4, E5, E6, E7}

Here, E1 = AB, E2 = AD, E3 = BD, E4 = BC and so on.

3.3 TYPES OF GRAPHS:

3.3.1 Undirected Graph:

Undirected Graph is a graph in which edges of a graph do represent direction. For Example, a graph is given in the figure [Fig:3.1] is a kind of undirected graph. Edge E1, connects two vertices (nodes) A and B. Here, edge E1, do not represent direction, that means it is to be assumed that the there is a way from A to B and similarly there is also a way from B to A as well.

3.3.2 Directed Graph:

Directed Graph is a graph in which edges of a graph represent direction. Consider the following figure [Fig:3.2], in which edge E1, which connects two vertices A and B. Edge E1 is represented as an arrow from A to B. That means there is a way from vertex A to vertex B, but there is no path is there from B to A.

[Fig: 3.2 Directed Graph]

As shown in the figure, if there is a path from B to C as well from C to B, then in the directed graph it has to be represented by using two edges E4 and E5 as shown in the figure. Edge E4 represent path from vertex C to vertex B, and edge E5 represent path from vertex B to vertex C.
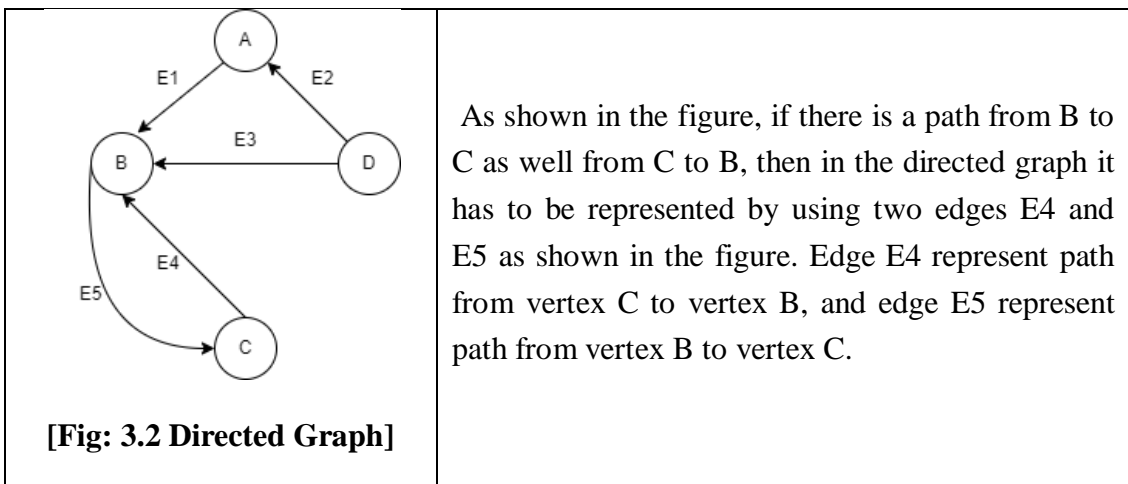
3.3.3 Connected Graph:

If any two vertices of the graph are connected to each other then that graph is called connected graph.

[Fig: 3.3 Connected Graph]

As shown in the figure, all the vertices of the graph are connected by some path. There are no two vertices are there between them path is not exists. Refer the following table in which we have demonstrated the paths between one vertex to any other vertex.

174
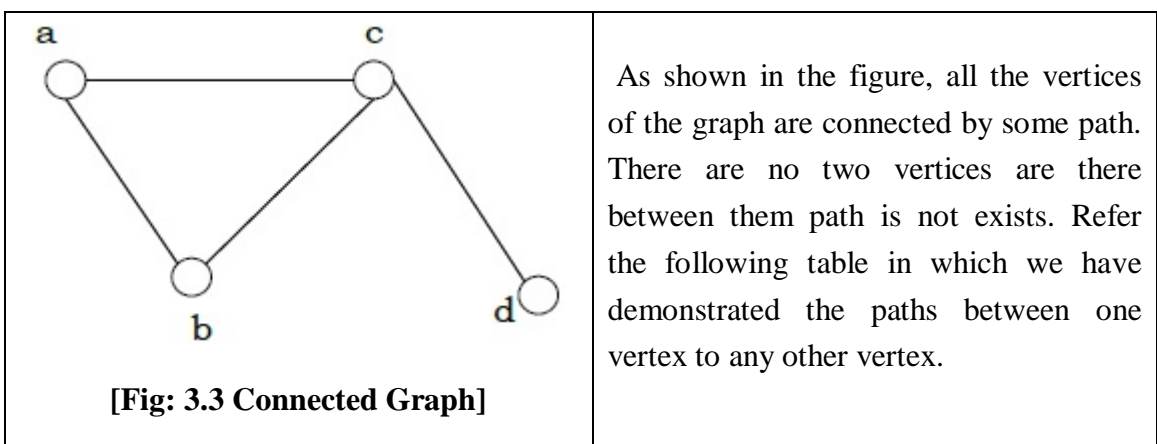
Source Vertex	Destination Vertex	Path
a	b	a-b
a	c	a-b-c, a-c
a	d	a-b-c-d, a-c-d
b	c	b-a-c, b-c
c	d	c-d

3.3.4 Disconnected Graph:

If any two vertices of the graph are not connected by any path, then that graph is called disconnected graph. A disconnected, undirected and acyclic graph is called forest.

[Fig: 3.4 Disconnected Graph]

As shown in the figure, all the vertices of the graph are not connected. Vertex a and b are connected with each other by direct edge between them but there is no direct or indirect path is possible between vertex a and c, or vertex a and d. Hence, the graph demonstrated in the figure is called Disconnected graph.
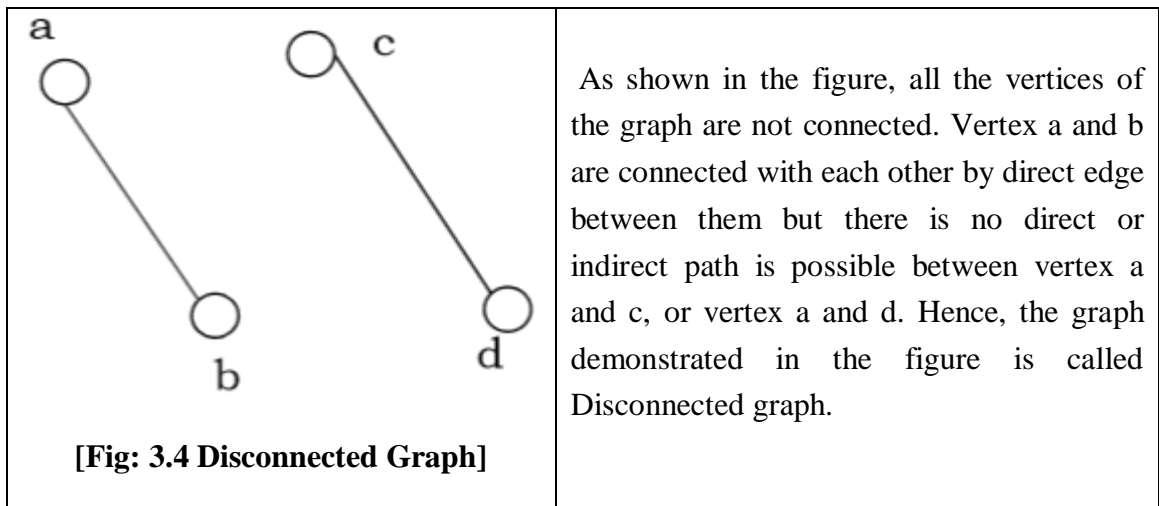
3.3.5 Weighted Graph:

If all edges of a graph, represent some numerical weight on it, then that graph is called weighted graph.
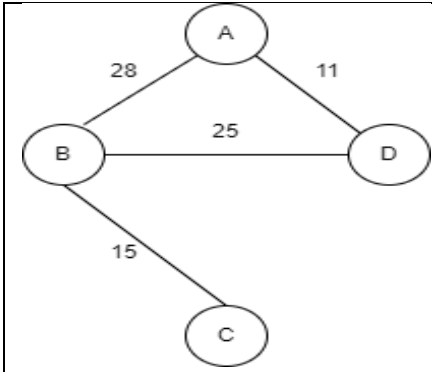
[Fig: 3.5 Weighted Graph]

As shown in the figure, all the edges of a graph have some numerical weight is assigned on them. This weight can be anything. It may be distance between two vertices, it may be time to travel from one vertex to any other vertex or cost (fare) to travel from one vertex to any other vertex. This type of graph where all edges represent some numerical weight on them is called weighted graph.

3.3.6 Strongly Connected Graph:

In a directed graph, if path exists from anyone vertex to all other vertices, then, that graph is called strongly connected graph.
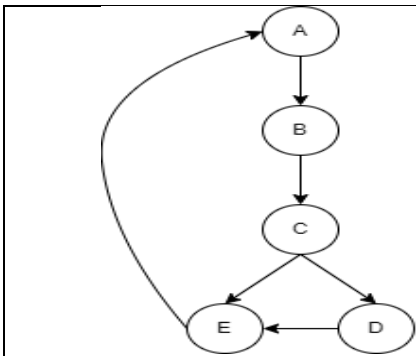
[Fig: 3.6 Strongly Connected Graph]

As shown in the figure, all the vertices of the graph are connected with each other and hence it is a connected graph, not only that there exists at least one path from one vertex to any other vertex and hence it is called strongly connected graph. For example, the path from D to C is (D-E-A-B-C)

3.3.7 Weakly Connected Graph:

In a directed graph, if vertices are connected with each other and if at least one vertex is not having path to any other vertex, then this type of graph is called weakly connected graph.

As shown in the figure, all the vertices of the graph are connected with each other and hence it is a connected graph. Now there is no path exists from vertex D to vertex B. Therefore, this graph is an example of weakly connected graph. In this graph you can notice that vertex D has two incoming edges, but there is no outgoing edge is there.

[Fig: 3.7 Weakly Connected Graph]
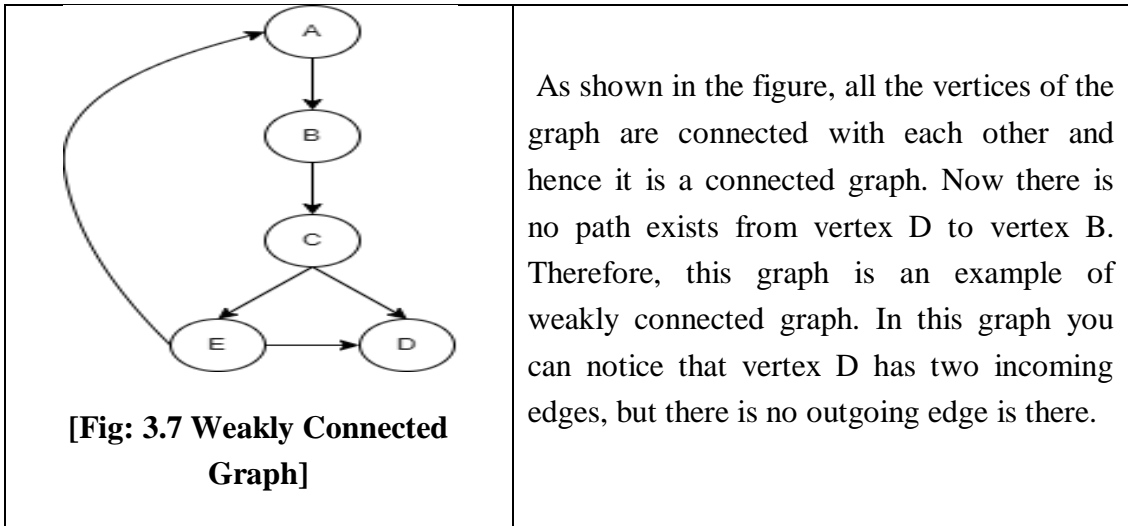
3.4   TERMINILOGIES:

1. **Indegree:** Indegree of any vertex v, which can be represented as $d^-(v)$ is the number of incoming edges into the vertex v.

2. **Outdegree:** Outdegree if any vertex v, which can be represented as $d^+(v)$ is the number of outgoing edges from the vertex v.

3. **Adjacent Vertices:** Two vertices are said to be adjacent to each other, if there is a direct edge is there between them. For example, in the figure [Fig:3.1] vertices A and B are adjacent vertices. But because of vertex A and C do not have direct edge between them there are not adjacent. Vertex A and C are connected vertices via vertex B.

4. **Path:** A path can be defined as sequence of unique vertices in which each vertex must be adjacent to the next. A path p of length n can be represented as _

$$P=<V_0, V_1, V_2, …, V_n>$$

5. **Cycle:** A graph contains cycles if there is a path of non-zero length through the graph, $p = <V_0, V_1, …, Vn>$ such that $v_0 = v_n$. For Example, in the figure [Fig:3.7] A-B-C-E-A is cycle.

6. **Loop:** If there exist an edge, from any vertex V to V is called Loop. For example, in the figure [Fig: 3.8] there is an edge from vertex B to B is called loop.
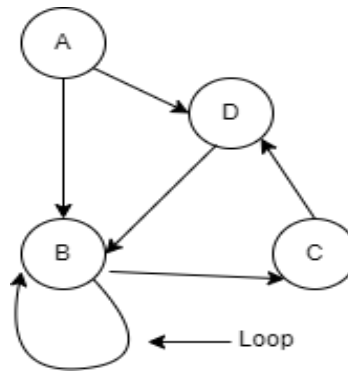
[Fig: 3.8 Loop in a Graph]

7. **Isolated node/vertex:** A node/vertex do not have any incoming or outgoing edge is called isolated node. An isolated node does not have any relation with any other node of a graph or it is disconnected node.

Check Your Progress-1

1. Two nodes having direct edge between them are called _____.

[A] Isolated nodes   [C] Adjacent nodes

[B] Root nodes   [D] None of the above

2. A graph on which, every edge has some numerical value is assigned is called _____ graph.

[A] Directed   [C] Connected

[B] Disconnected   [D] Weighted

3. A sequence of adjacent vertices from one node to another node is called _____.

[A] Path   [C] Cycle

[B] Loop   [D] Outdegree

3. An edge on the same vertex is called _____.

[A] Path   [C] Cycle

[B] Loop   [D] Outdegree

3.5   GRAPH REPRESENTATION METHODS:

As we know there are many different applications are there of the graph theory. But to process the graph by the computer system using different algorithms, we need to represent the information available with us about the graph, into the computer system. To feed information about graph into the system, we need to use representation method.

To represent the graph into the system, one can use any one method from the given two methods:

3.5.1 Adjacency Matrix Representation:

In adjacency matrix representation method of the graph, we are representing the information about graph in matrix (2-dimensional array). In this matrix number of rows and number of columns has to be same as number of vertices in the graph. As we are representing details about adjacent (two vertices connected with each other with direct edge), if the row-column pair of vertices are adjacent, then we will mention 1 in the cell of their intersection otherwise we will mention 0.

Consider the following graph shown in the figure [Fig: 3.9]. There is total 5 vertices are there. To represent this information to system we need to take a 2-dimensional array of 5 rows and 5 columns. Now, for each node we need to find their adjacent nodes as shown below:

Adj [A] = {B, D}

Adj [B] = {A, C, D}

Adj [C] = {B, D, E} and so on.

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	1	1
D	1	1	1	0	1
E	0	0	1	1	0
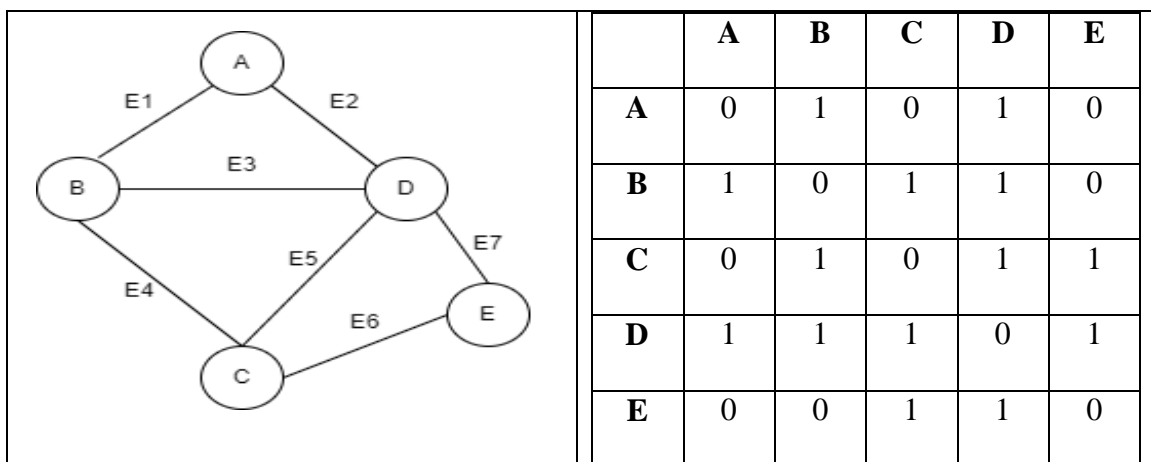
[Fig: 3.9 A Graph and its Adjacency Matrix]

You can see from the graph those vertices which are adjacent to each other we will write 1 in the cell, and if the vertices are not adjacent then we will write 0 in that cell. By this mean we provide information that which vertices are directly connected with each other. If the graph is weighted graph, then we instead of 1 we need to put weight of an edge instead of 1.

The adjacency metric of the above graph is a symmetric matric. The reason behind this is, we have made an adjacency matric of undirected graph. In undirected graph is there is a way from A to B means, there has to be a way from B to A, which makes our metric symmetric. Therefore, for all undirected graph you will get adjacency matrix to be a symmetric matrix, but in the case of directed graph it may or may not be.

3.5.2 Adjacency List Representation:

In adjacency list representation method, we make a Link-List of adjacent vertices for each vertex of the graph. For example, in the above figure B and C are the adjacent of vertex A. Therefore, we will prepare a Link-List of two nodes to store B and C. Finally, we will take an array of the pointers, which can store the address of the first node of every Link-List. Adjacency List representation of the above graph is shown below in the figure [Fig: 3.10].
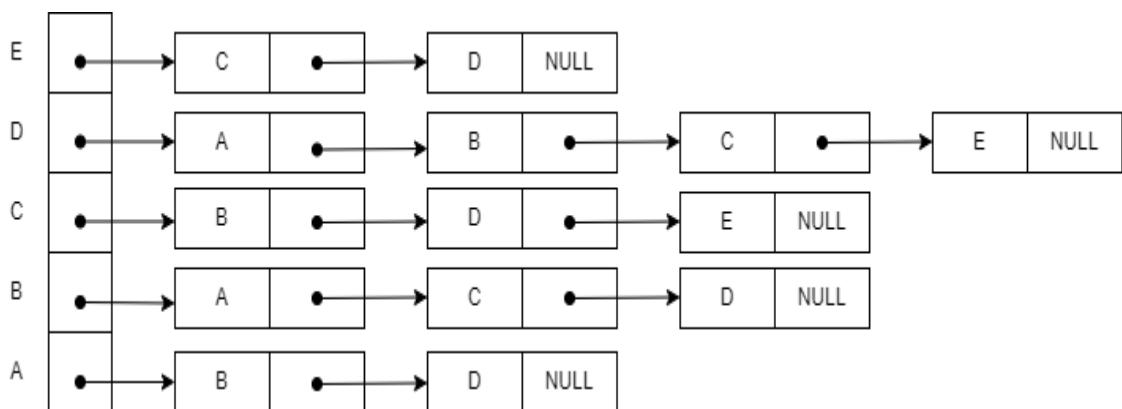
[Fig: 3.10 Adjacency List representation of the Graph]

Check Your Progress-2

1. In _____ representation method of graph we make Link-List for each vertex of a graph, to represent their adjacent nodes.

[A] Adjacency Matrix   [C] Adjacency List

[B] Depth First Search   [D] Breadth First Search

2. Adjacency matrix of a _____ graph is always symmetric matrix.

[A] Directed graph   [C] Undirected graph

[B] Weighted graph   [D] Diagraph

3.6   TRAVERSAL METHODS OF GRAPH:

As we know traversal is the process, where we need to visit all the nodes once. Tree and Graph are non-linear data-structures. As we have seen in the previous unit, there are three traversal methods are there for tree data-structure. Similarly, there are two traversal methods are in the graph: [1] DFS: Depth First Search and [2] BFS: Breadth First Search. Let us see both the methods one by one.

3.6.1 DFS: Dept First Search

In Depth First Search, algorithm you need to use data-structure stack. You can start your traversal by selecting any random vertex and follow the algorithm explained below:

```
V = Select_Any_Random_Vertex()

push(V)

while (!EmptyStack())

{

        V = pop()

        if (!visited(V))

        {

                visit (V)

                push_adjacent (V)

        }

}
```

As shown in the algorithm, you can start with any random vertex of the graph. You need to push that vertex in the stack, and repeat the process till stack becomes empty. You

need to pop() the vertex, and check whether that vertex is visited or not. We have an integer array named 'visited' with size same as number of vertices in the graph. The visited array is pre-filled with 0 for all the vertices, which indicates that there is no vertex is visited yet.

Now, if the vertex is not visited, then you need to visit it, means you need to put 1 in the particular element of an array which represent that vertex. After visiting the vertex, you need to push all of its adjacent vertices in the stack. This process is repeated till all the nodes are not visited and stack becomes empty.

In the figure [Fig:3.11], a graph, an array visited and a stack is demonstrated. Initially all the elements in the array are 0, which represent no vertex is visited yet, as well as our stack is also empty (top=-1). Now let us start with vertex V1, and push V1 in the stack. Check if stack is empty. Here stack contains V1, so it is not empty. The pop() as vertex from the stack. Obviously, you will get V1. Check whether V1 is visited or not. V1 is not visited as in the array we have 0 at place of V1. Then visit V1 (make 1 in the visited array and push adjacent of V1 (V9 and V2) in the stack.
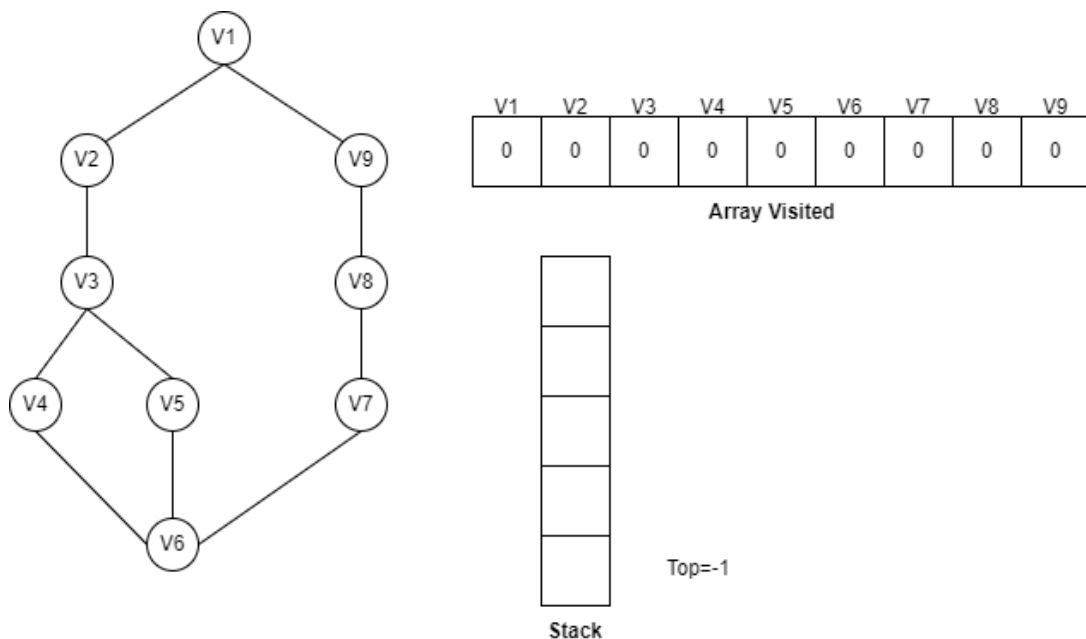
[Fig: 3.10 Depth First Search in initial stage]

Now, repeat the process because stack is not empty it contains V9 and V2. Pop an element from the stack, we will get V2, check it is visited or not. No, it is not visited, then visit it, by changing 0 to 1 at place of V2 in the array visited. Visit V2 and push its adjacent like V3 and V1 in the stack. Now, stack has V9, V3 and V1.

Pop an element from the stack. This time it is V1. Because of V1 is already visited you don't have to do anything. Now stack has only two vertices V9 and V3.

Again, pop an element from the stack. This time V3 will come out. Visit V3 as it is not visited and push their adjacent V5, V2 and V4 in the stack. Stack will now have V9, V5, V2 and V4. Continue this process till stack does not becomes empty.

Here, we have started with V1. Vertices V2 and V9 are adjacent of it. We take one from them V2. We visit V2 and going in the depth of adjacent of V2 (V3). Then we visit V3 and going into depth of it. That is the reason, this traversal method is known as Depth First Search. The nodes will be visited in the following manner. Final visit sequence in DFS is as follows:

V1 – V2 – V3 – V4 – V6 – V7 – V8 – V5 – V9.

[Fig: 3.12 DFS after visiting 5 nodes]

3.6.2 BFS: Breadth First Search

Breadth First Search (BFS) algorithm is similar to DFS, just the difference is instead of using stack, in BFS we are using Queue data-structure. The algorithm for BFS is given below:

```
V = Select_Any_Random_Vertex()
enqueue(V)
while (!EmptyStack())
{
        V = delqueue()
        if (!visited(V))
        {
                visit (V)
                enqueue_adjacent (V)
        }
}
```

In BFS, algorithm the vertices are visited level wise. After visiting node V1, we will visit V2 and V9 both, and then we will visit adjacent of V2 and adjacent of V9, which is depicted in the following figure [Fig: 3.13].
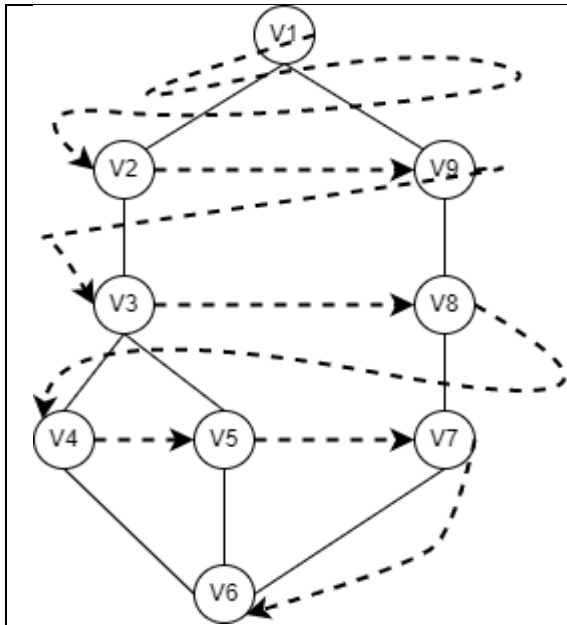
In BFS, if we start with V1, then we will visit V1. After Visiting V1 there are two adjacent are there V2 and V9. Make sure here you don't need to keep pending visit of V9. You have to visit both V2 and V9. Now visit all the adjacent of V2 that is V3. Visit all the adjacent of V9 that is V8. After visiting V8 you need to visit V4 and V5 (adjacent of V3). Then you will visit V7 (adjacent of V8) and finally you will visit V6. So, the visit sequence in BFS:

V1 – V2 – V9 – V3 – V8 – V4 – V5 – V7 – V6.

[Fig: 3.13 BFS Traversal]

Check Your Progress-3

1. _____ data-structure is used to implement Depth First Search traversal in Graph.

[A] Stack   [C] Queue

[B] Circular queue   [D] None of the above

2. _____ traversal method visits all the vertices of a graph level wise.

[A] DFS   [C] BFS

[B] Adjacency Matrix   [D] Adjacency List

3. _____ data-structure is used to implement Breadth First Search traversal in Graph.

[A] Stack   [C] Queue
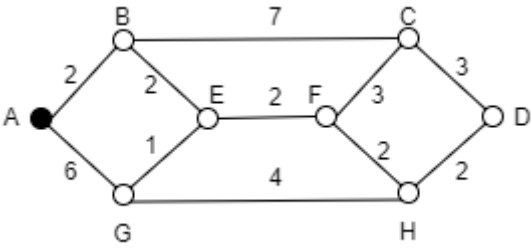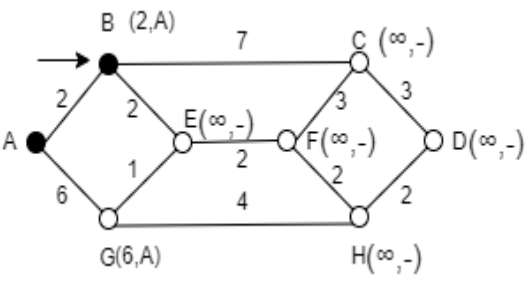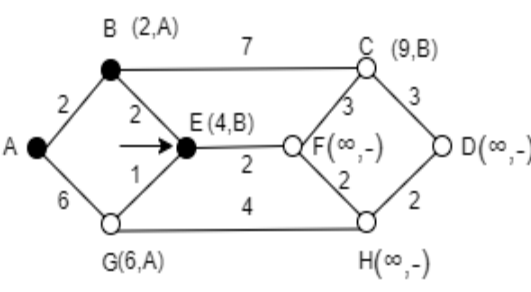
[B] Circular queue   [D] None of the above

3.7   DIJKSTRA'S ALGORITHM

Dijkstra has provided a solution, to solve shortest path problem. It is used in the Computer Network, to find the shortest path from source machine to destination machine. Shortest Path algorithm helps router to compute the shortest path

from source to destination, so that router can take the decision, on which outgoing line an input packet should be redirected. In fact, Dijkstra's algorithm is an example of static routing, and many advanced and dynamic algorithms are also available and used as routing algorithms.

Let us understand how shortest path can be determined using Dijkstra's algorithm using an example.

 [Fig: 3.14(A)]	Consider the following figure, where we need to find a shortest path from vertex A to vertex D. Initially all nodes except A, will set their distance to Infinity. Vertex B and G are adjacent to A. They know the distance from A; hence they will update their routing tables as shown in figure [Fig: 3.14(B)].
 [Fig: 3.14(B)]	Because, the distance from B to A is smaller than distance from A to G, vertex B considered in the shortest path. Now, Because, vertices C and E are adjacent to B they will update their route from infinity to (9,B) and (4,B) as shown in the figure [Fig:3.14 (C)].
 [Fig:3.14 (C)].	Because the distance of vertex E from B is smaller than C to B, vertex E should be chosen as in the shortest path. (4, B) in the routing table of E denotes that the vertex A is 4 distances far from E via vertex B. Once E is chosen its adjacent G and F update their routing tables as (5, E) and (6, E) as shown in the Figure [Fig: 3.14(D)]
Here, vertex G has two entries in its routing table. G knows that there is a direct edge from G to A which is 6 steps far (6, A), where E claims that A is 4 steps far from it via B (4, B). G know that E is 1 step far from it. Vertex G now have two entries (6, A)	

and (5, E). Obviously, G will consider the shortest path from G to A via E that is (5, E).
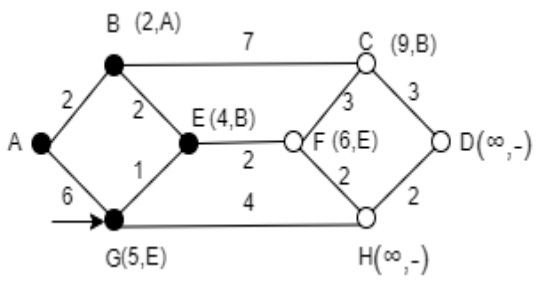
 [Fig: 3.14(D)]	From, vertex C (9,B), F(6,E) and G(5,E), vertex G has the smallest distance to A. Therefore, G is chosen for the shortest path. Once, node G is chosen then its adjacent H will update its routing table from infinity to (9, G) as G is (5,E) and distance from G to H is 4, which is shown in the figure [Fig:1.14 (E)).
 [Fig: 3.14(E)]	Vertex F is chosen in the shortest path as distance of F (6,E) to A is less than the distance from C (9,B) to A and H (9,G). Vertex H will update its routing table from (9,G) to (8,F) as A is just 8 steps away from A via F.
 [Fig: 3.14(F)]	Once the routing table of H is updated then D will choose its routing option from (12,C) and (10,H). Obviously, D will chose shortest path, that is (10,H). Now the shortest path from A to D is: A – B – E – F – H – D.
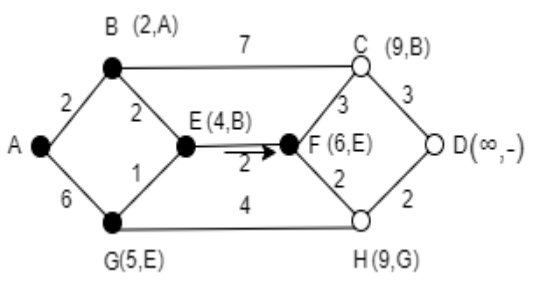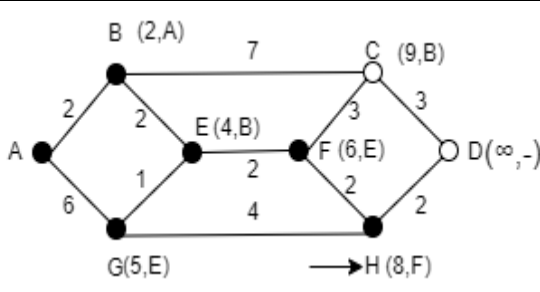
To implement Dijkstra's shortest path, use the following algorithm:

* Initialise d and pi*

for each vertex v in V(g)

g.d[v] := infinity

g.pi[v] := nil

*g.d[s] := 0; * Set S to empty * S := { 0 } Q := V(g) * While (V-S) is not null* while not Empty(Q)*

1. *Sort the vertices in V-S according to the current best estimate of their distance from the source u := Extract-Min (Q);*
2. *Add vertex u, the closest vertex in V-S, to S, Add_Node (S, u);*
4. *Relax all the vertices still in V-S connected to u relax (Node u, Node v, double w[][])*

if d[v] > d[u] + w[u][v] then

d[v] := d[u] + w[u][v]

pi[v] := u

3.8   MINIMUM COST SPANNING TREE:

A spanning tree is a subgraph derived from some graph that includes all the vertices and selective edges which form a tree (non-cyclic) structure. One graph may have many different spanning trees. In the following figure, Fig:3.15 we have demonstrated a Graph and its possible spanning trees.

| A Graph | Possible Spanning Trees |
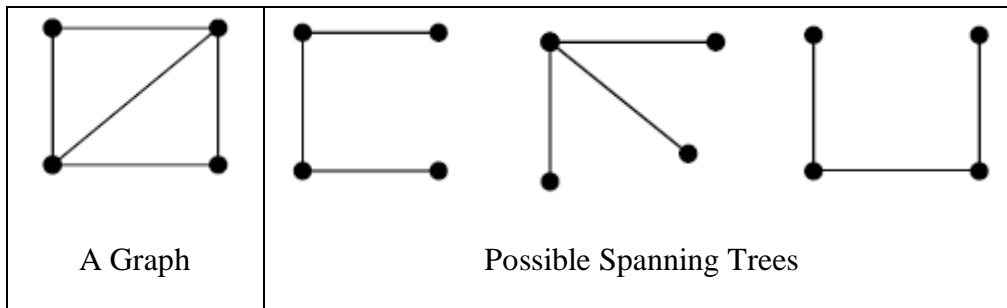
[Fig: 3.14 Graph and some possible spanning trees]

As we have discussed that the main difference between a Graph and Tree data-structure is Graph is a cyclic data-structure, where Tree is a non-cyclic data-structure. There are many applications are there in which, we need to convert graph into tree. For example, if you search your destination from any other (your) current location in the Google Map then, it is possible that there are multiple path or ways are there from source to destination. These all paths are conversions into spanning trees. From multiple paths from source to destination an optimal path is highlighted, which is nothing but a minimum cost spanning tree.

Therefore, our intention is not to find, only number of spanning trees, but to find out spanning tree having minimum costs. There are two algorithms are there to find the minimum cost spanning tree from a given graph. Let us discuss both algorithms one by one.
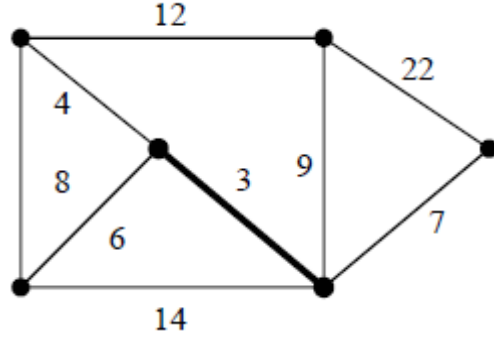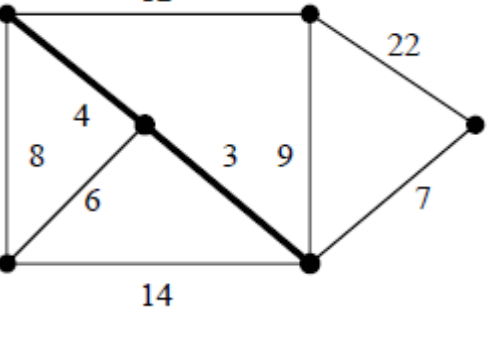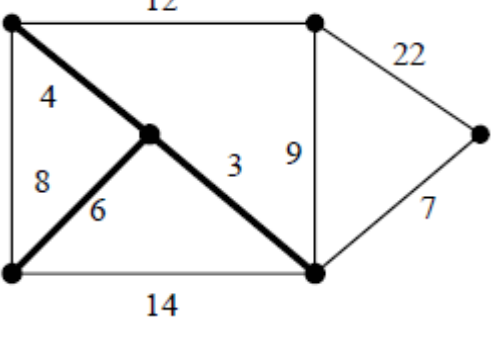
3.8.1   Kruskal's Algorithm:
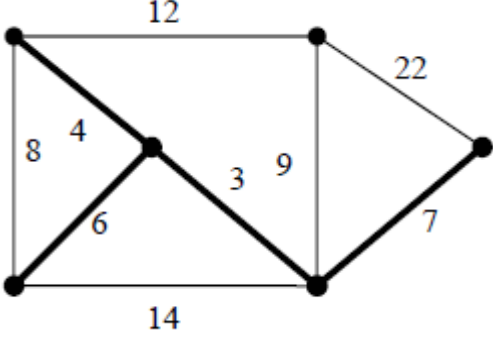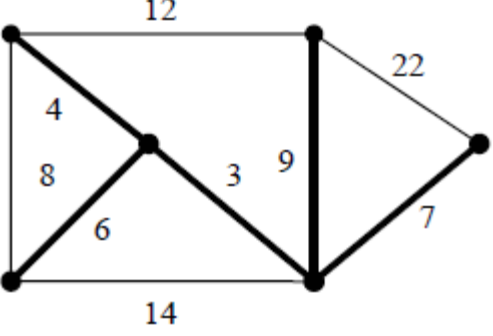
Kruskal's algorithm considers the idea of forest of trees. Initially, the forest contains of **n** individual node trees (and there are no edges). At each step, we add one (the smallest cost) edge so that it connects two trees together. If after adding any edge cycle is formed then, it would simply mean that it connects two nodes that were already connected. So, we will reject it. The steps for the Kruskal's Algorithm are explained below:

1. *The forest is built from the graph G, with each node as a separate tree in the forest.*

2. *The edges are added in a priority queue, where smaller cost of edge has higher priority.*

3. *Perform the following steps until you have added n-1 edges to the graph,*
    3.1   Extract the edge having smallest cost value from the queue and add it graph.
    3.2   If it makes a cyclic structure into the graph, then connection is already established between those vertices and hence remove that edge.
    3.3   Else make that edge as permanent.

Let us discuss and understand how the Kruskal algorithm is used to create Minimum Spanning Tree (MST) from the given graph. Consider the following graph.

[Fig: 3.15 [A] Graph]

As show in the figure [Fig: 3.15 [A]], a Graph is given. We have to derive Minimum Spanning Tree from the given graph using Kruskal's algorithm. In the first step we will consider only vertices and there is no edge is there in the graph. All the edges of the graph are inserted in the priority queue in such a way that smaller the cost of an edge, having higher priority.

[Fig: 3.15 [B] Adding first edge]

Now, we will retrieve an edge from the priority queue, here smallest cost edge, whose weight is 3 has highest priority, it will be removed from the queue. We will place this edge in the graph, and will check whether the newly added edge forms a cycle or not? In this case our graph has only one edge which does not form any cycle, and hence it will be added to graph permanently.

[Fig: 3.15 [C] Adding second edge]

Similarly, remove another edge having weight 4 from the priority queue and add it to the graph. Check for the cycle formation. From the figure [Fig:3.15 [C], we can see that there are only two edges are there in the graph which do not foam any cycle in the graph. Therefore, the edge having weight 4 will become permanent in the graph.

[Fig: 3.15 [D] Adding third edge]

Yet all tree nodes are not connected, so we will continue the process, and pick up one more edge having weight 6 from the priority queue. We add this edge to the graph by connecting its vertices and will check for the cycle formation. After adding the edge, because of cycle is not formed, we will make this edge (having weight 6) to be permanent.

Now, we will pick up one more edge from the priority queue having edge 7, we will add it to the graph. Because addition of this edge does

[Fig: 3.15 [E] Adding fourth edge]	not introduce cycle in the graph and hence this edge will be added as permanent edge. Now the, edge coming from the priority queue is having weight 8. But if add this edge to the graph it introduces cycle in the graph hence the edge whose weight is 8 is ignored and will not be considered in the MST.
[Fig: 3.15 [F] Adding last edge]	After ignoring edge with weight 8, now will pull out another edge from priority queue, which is nothing but an edge with weight 9. After adding this edge, no cycle is formed and hence it will be added as a permanent. After adding this edge (edge with weight 9), all the vertices of the graph will be connected with each other, without any cycle, which is nothing but MST (Minimum cost Spanning Tree).

3.8.2   Prim's Algorithm:

Prim's algorithm uses the idea of sets. Instead of considering the graph by sorting its edges on the basis of its weight, this algorithm processes the edges in the graph randomly by building up disjoint sets.

Prim's algorithm uses disjoint sets A and A'. Prim's algorithm works by iterating through the vertices and then finding the shortest edge from the set A to that of set A' (i.e., set of edges excluding A), followed by the addition of the vertex to the new graph. When all the vertices are processed, we have an MST.

To perform Prim's algorithm, we need to follow the following steps:

Let G is the graph with N vertices for which MST is to be generated.

Let T be the MST.

Let T be an any randomly selected single vertex x.

while (T has fewer than N vertices)

{
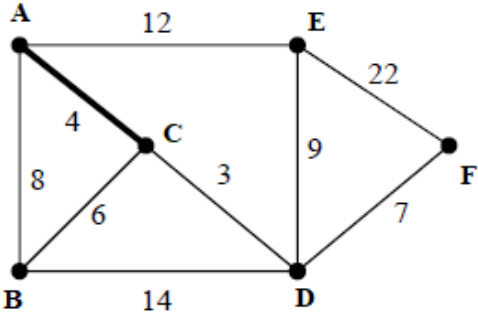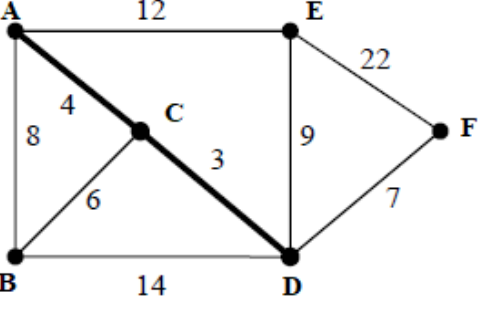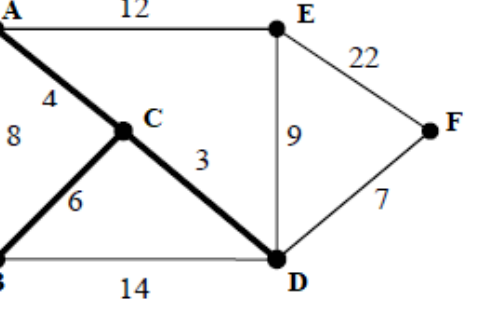
     find the smallest edge joining T to G-T

     add that edge to T

}

Let us understand it, by taking an example:

	Consider the graph given 3.15[A]. We want to convert the graph into MST using Prim's algorithm. Let us start with any random vertex A. B, C and E are adjacent of A having AC=4, AE=12 and AB=8. We will consider the smallest edge and will connect AC. Our two sets are: A={ AC(4) } and, A'= { AB(8), AE(12) }
	Now, we have added new vertex C, in the MST, so all the edges from C should be added to A' and will sort the A'. Therefore, A'={ **CD(3)**, **CB(6)**, AB(8), AE(12) } We will pick the smallest edge from A' and ad it to A. Therefore, A={ AC(4), CD(3) }
	New, vertex connected in MST is D. So, put all the edges from D, into A'. A'={ CB(6), **DF(7)**, AB(8), **DE(9)**, AE(12), **DB(14)** } Pick the smallest edge from A' that is CB(6) and connect vertex B. Set A is: A={ AC(4), CD(3), CB(6) }

	New vertex added is B. Therefore, we will add all edges from B, but here edges from B, AB and BD is already in the A'. So, A' will be: A'={ DF(7), **AB(8)**, DE(9), AE(12), **DB(14)** } We will pick up a new smallest edge DF(7) and connect F. Our set A has: A={ AC(4), CD(3), CB(6), DF(7) }
	New vertex added is F. Therefore, edge EF(22) should be added to A'. Our set A' will now: A'={AB(8), DE(9), AE(12), DB(14), FE(22) } We, will take smallest edge AB(8) from A' but if add it to MST it will create a cycle. So, ignore AB(8) and pick another smaller edge that is DE(9) and will add it to MST. Our set A will now: A={ AC(4), CD(3), CB(6), DF(7), DE(9) }

Because, we have connected all the vertices of a graph, we have build our MST by connecting edges AC(4), CD(3), CB(6), DF(7), DE(9). Now suppose, we pick up edges from set A', it will foam a cycle and we know that the tree is non-cyclic structure. So, cycle is not allowed in the MST.

Check Your Progress-4

1. _____ algorithm is used to find shortest path.

[A] BFS   [C] Depth First Search

[B] Kruskal's   [D] Dijkstra's

2. Prim's algorithm is for _____.

[A] Building MST	[C] Representing graph
[B] Finding shortest path	[D] Traversal of graph

3.9   Let Us Sum Up

In this Unit we have made detailed discussion on graph theory. We have defined a graph as a set of, set of vertices and set of edges. Then we have discussed different types of graphs and some terminologies related to graph.

In this unit, we have discussed that graph can be represented into the memory of computer using [1] Adjacency Matrix or [2] Adjacency List, where term adjacent means two vertices connected by direct edge between them. To traverse all the vertices of the graph two traversal methods: [1] DFS: Depth First Search and [2] BFS: Breadth First Search is used. In the DFS algorithm, data-structure stack is used, and in the BFS algorithm data-structure queue is used.

We have also discussed that, how the graph is processed by Dijkstra's algorithm to find shortest path and finally we have discussed how a graph can be converted into Minimum cost Spanning Tree (MST) using Prim's and Kruskal algorithms.

3.10   SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

3. [C] Adjacent nodes
4. [D] weighted
5. [A] path
6. [B] Loop

Check Your Progress-2

3. [C] Adjacency List

4. [C] Undirected graph

Check Your Progress-3

4. [A] stack
5. [C] BFS
6. [C] Queue

Check Your Progress-4

3. [D] Dijkstra's
4. [A] Building MST

3.11   GLOSSARY

3. **Graph**: A Graph G is a set of, set of vertices and set of edges. It is non-linear and cyclic data structure.
4. **Adjacent vertices:** Vertices of a graph having direct edge between them is called adjacent vertices or adjacent nodes.
5. **Directed Graph:** A Graph in which, edges of the represent direction, is called directed graph.
6. **Weighted Graph:** A Graph in which, edges of the represent weight (come numerical value), is called directed graph. Weight can be cost, distance or time.
7. **Isolated Node:** A node or vertex of a graph do not have any incoming or outgoing edge is called Isolated Node.

3.12   Assignment

1. Define Graph and explain related terms.
2. Write a short note on: Dijkstra's algorithm
3. Explain Representation methods of a graph.
4. Explain Traversal methods of a graph.
5. List and explain methods, used to construct MST from a given Graph.

3.13   Activity

- Implement a program to find shortest path problem using Dijkstra's Algorithm.

3.14   Case Study

Find the following problems on the Internet and write your comments on it:
1. Travelling Salesman problem
2. Shortest Path Problem
3. 7-Bridge Problem

3.15   Further Reading

- Data Structure through C by Yashvant Kanetkar.
- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.

Block Summary

- Tree is a non-cyclic, non-liner data structure. It is a subset of graph. A tree having a root node is called rooted tree.
- A node from which tree structure starts is called root node. Root node is node which does not have parent.
- A node which does not have any child is called leaf node. Leaf nodes are also called external nodes of a tree.
- A Tree in which each node of the tree has at most two children is called Binary Tree. A Binary tree in which smaller values are inserted to left sub-tree and larger values are inserted into the right sub-tree is called binary search tree.
- Tree can be traversed using [1] In order (Left-Root-Right), [2] Pre order (Root-Left-Right) or [3] Post order (Left-Right-Root).
- AVL tree is a height balanced binary search tree, in which each node has balance factor. If Balance factor of each node in a binary tree is either -1, 0 or 1 then that is called balanced binary search tree or AVL-Tree.

- To maintain the balance of AVL tree, after inserting new node to the tree, balance factor is recomputed from that node to root node. If any node is found, which should not have balance factor from -1, 0 and 1, then rotation(s) are applied in the tree to make the tree balance.
- B-Trees are M-way tree, in which each node can have at most m-1 values in it. B-Trees are used for the Indexing purpose in the database like applications, which usually are stored on secondary storage.
- Graph is a cyclic structure. It is non-linear data structure. Graph G can be defined as set of, set of vertices and set of edges: G = { V, E }.
- Two nodes which are directly connected to each other with direct edge are called adjacent nodes or adjacent vertices.
- Graph can be represented into the memory of computer using [1] Adjacency Matrix or [2] Adjacency List methods.
- Graph can be traversed using DFS (Depth First Search) and BFS (Breadth First Search). In the implementation of DFS algorithm data-structure stack is used, whereas in the algorithm of BFS data structure queue is used.
- Dijkstra has given a solution to find shortest path from any one vertex to other vertex in graph using static method which is called Dijkstra's shortest path algorithm.
- In many applications a graph (cyclic structure) has to be transformed into tree (non-cyclic structure) by removing cycles, by only considering those edges which has minimum weight. This structure is called Minimum cost Spanning Tree or simply MST. To build MST from a graph Kruskal's or Prims' algorithms are used.

Block Assignment

Short Questions:

7) Explain node structure for Binary Search Tree.

8) What is BST? Explain it with an example.

9) Define AVL-Tree.

10) What is Graph? Draw a graph and explain related terms

11) List different types of graphs you know.

Long Questions:

7) List and explain representation of method of a graph.

8) List and explain traversal methods of graph.

9) Explain Dijkstra's shortest path problem using an example.

10) What is MST? Discuss different methods to find MST from a given graph.

11) Draw an AVL-Tree for the following data:
     [1] 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

     [2] 9, 15, 20, 8, 7, 13, 10

12) Draw a B-Tree for the following data for order m=4:
     [1] 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

     [2] 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Enrolment No.

1. How many hours did you need for studying the units?

Unit No.	1	2	3	4
No. of Hrs				

2. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific example if any
Presentation Quality	☐	☐	☐	☐	_____ _____
Language and Style	☐	☐	☐	☐	_____ _____
Illustration used (Diagram, tables etc)	☐	☐	☐	☐	_____ _____
Conceptual Clarity	☐	☐	☐	☐	_____ _____
Check your progress Quest	☐	☐	☐	☐	_____ _____
Feed back to CYP Question	☐	☐	☐	☐	_____ _____

....

4. Any other
Comments
....

....
....
....
....
....
...

....
....
....
....
....
...

....

Data Structure

BLOCK 4: TECHNIQUES (SEARCHING AND SORTING) AND
FILE STRUCTURE

BLOCK 4: TECHNIQUES (SEARCHING & SORTING) & FILE\ STRUCTURE

Block Introduction

This block of data structures introduces several algorithms for the searching and sorting of data-elements or records from the database or from a memory location.

The first unit of this bloc introduces the basic searching techniques of data-elements stored in the array, which are stored in the main memory. The second unit focuses several sorting techniques, which can be used to sort data-elements stored in an array. The third unit highlights the file structures, in which different types of file organizations, which can be used for different purposes are largely discussed.

Block Objective

After learning this block, you will be able to:

- Understand, what is searching?
- Understand different types of searching.
- Implementation of searching algorithms
- Understand, what is sorting?
- Understand different types of sorting
- Implementation of different sorting algorithms
- Understand the concept of file-structure
- Understand records in file
- Understand sequential and index-sequential file organizations
- Understand hashing Techniques

BLOCK STRUCTURE

BLOCK 4: **TECHNIQUES (SEARCHING AND SORTING) AND FILE STRUCTURE**

UNIT 1 **SEARCHING TECHNIQUES**

Learning Objectives, Introduction, Sequential or Linear and Binary search, Algorithms for Sequential and Binary search, Implementation of Linear and Binary search, Let us sum up.

UNIT 2 **SORTING TECHNIQUES**

Learning Objectives, Introduction, What is Sorting, Types of Sorting – Internal and External, Bubble, Insertion, Selection, Quick, Marge, Let us sum up.

UNIT 3 **FILE STRUCTURE**

Learning Objectives, Introduction, File structure- concept of fields, Fields and Records, Sequential and Index file organizations, Hashing Techniques, Let us sum up.

UNIT 4 **PROGRAMS OF SEARCHING AND SORTING**

Unit 1: Searching Techniques

<div style="text-align: right">**1**</div>

Unit Structure

1.0   Learning Objectives

After learning this unit, you will be able to:

- Understand the process of searching

- Understand different types of searching

- Implementation of algorithms for various searching methods

1.1   Introduction

Computer system are often used to store large amount of data, in the computer's memory, which has to be searched on the basis of several criterion periodically. Therefore, storing data competently in order to enable quick searches is a vital component.

This section lays stress on the examination of the performance of some searching algorithms and the data-structures which they use.

1.2   LINEAR AND BINARY SEARCH ALGORITHMS

1.2.1 Linear Search:

The linear search which also known as Sequential search is an algorithm that is used to search through an array or list of data-items in order to find a specific value.

The searching process starts with matching data-elements of the array. Initially, the first data-element is compared with the element to be searched, if the value to searched is not matched with first data-element then we will compare it with second data-element and so on. We will continue this process till the end of an array. That means, it operates by checking each and every data-elements of an array, one at a time in sequential order until a match is found.

The process described above is an easy and straightforward method for searching data into an array. The search shall start at the beginning of the data collection, and gradually moving forward until, the desired target has been found or the all elements of an array has been compared. In order to use this method, one must be knowing the size of the area to search and the starting position of the data collection. Otherwise, a unique value could be used to specify the end of the search space. This method is usually applied to an array data-structure where lower bound (mostly 0) and upper bound is known.

For Example,

Consider the following example, where array is given, in which a total of 10 elements are there and we want to search a particular element say, 11 in it. The Linear search can be demonstrated on this set of data as given below:

49, 73, 35, 55, 11, 28, 24, 37, 19, 76

Now the same are stored in the array as shown below:

0	1	2	3	4	5	6	7	8	9
49	73	35	55	11	28	24	37	19	76

Now as 11 is to be searched from the given array above, we will start from the 0th position and will match the element stored at that position (which is 49) with 11. As in this case, element at 0th position is 49, as 49 is not equal to 11, so the variable 'i' will be incremented and move to next position, that is,1 and will compare the element 73 with 11, which is again not equal. This process continues till the last position in order to search 11, now suppose, 11 is not present in the array, so after traversing the entire list, a message of "Element not do not exist in the List" is displayed. If the search element to be searched is available in the array, then the element and its position is displayed on the console screen.

Tracing-1:

The tracing steps for above array with search element is explained below. Here, the search element is supposed 11 and name of an array is supposed X.

Step	Comparison	Conclusion
Step:1	I=0, X[I] ≠ se as 49 ≠ 11	Value not found and hence I++
Step:2	I=1, X[I] ≠ se as 73 ≠ 11	Value not found and hence I++
Step:3	I=2, X[I] ≠ se as 35 ≠ 11	Value not found and hence I++
Step:4	I=3, X[I] ≠ se as 55 ≠ 11	Value not found and hence I++
Step:5	I=4, X[I] = se as 11 = 11	Value found and hence loop is broken. We will print value found at I+1 that is 5th position.

Tracing-2:

The tracing steps for the same array with search element 101 is explained below. Here, the search element we are changing to 101 and name of an array is supposed X.

Step	Comparison	Conclusion
Step:1	I=0, X[I] ≠ se as 49 ≠ 101	Value not found and hence I++
Step:2	I=1, X[I] ≠ se as 73 ≠ 101	Value not found and hence I++
Step:3	I=2, X[I] ≠ se as 35 ≠ 101	Value not found and hence I++

Step:4	I=3, X[I] ≠ se as 55 ≠ 101	Value not found and hence I++
Step:5	I=4, X[I] ≠ se as 11 ≠ 101	Value not found and hence I++
Step:6	I=5, X[I] ≠ se as 28 ≠ 101	Value not found and hence I++
Step:7	I=6, X[I] ≠ se as 24 ≠ 101	Value not found and hence I++
Step:8	I=7, X[I] ≠ se as 37 ≠ 101	Value not found and hence I++
Step:9	I=8, X[I] ≠ se as 19 ≠ 101	Value not found and hence I++
Step:10	I=9, X[I] ≠ se as 76 ≠ 101	Value not found and hence I++

In the step:10, because the value stored at upper bound (9), which 76 do not match with search element 101. So, loop will not be broken and variable I will be incremented to 10. Now loop will be broken as the condition we have placed in the loop is I < 10. When we come out of the loop, we will check the value of I. Because the value of I is 10, means the element we are searching into the array do not exist.

Linear search algorithm, will match search element will all other elements of an array sequentially, and hence it is known as sequential search or linear search. In the best-case scenario where the element is located on the first position, we need to do only one comparison. Therefore the best-case complexity for linear search is O(1). In the worst-case, where the element is located at last position and if we assume that there are n number of elements are there in the array, we need to perform n comparisons, and hence the worst-case complexity on linear search algorithm is O(n). This is high complexity, so we can conclude that linear search algorithm is slower.

1.2.1 Binary Search:

Binary Search is most efficient method for searching a data-element from the data-elements stored sequentially in the array without using of auxiliary indices or tables. Basically, the key value to be searched is compared with the value of the middle element of an array. If they are equal, the search completed successfully, otherwise the upper or lower parts of an array are searched accordingly.

Basically, the elements of an array are in sorted either in ascending or in descending order. So, to start the searching process first the array is divided into two equal halves and the key value to be searched is matched with the data-element present at the middle position. If both of them are same, then the position of the key value is returned and

searching process will be terminated, otherwise, the search key will be compared with the middle data-element again, if the search key is greater than the middle data-element then the key element should be searched in the upper half area of an array whereas, if search key is less then it should be searched in the lower half of an array. This process is repeated until the key value is not found.

Now after knowing that the target must be in one half of the array or the other, the binary search will examine the median value of the half, in which the target must reside. Hence this algorithm narrows the search area by half at in each comparison, until it has either found the key value or the search fails (key value does not exists).

Now consider an example demonstrated below, in which we have stored some data-elements in the array as shown below:

0	1	2	3	4	5	6	7	8	9
2	7	11	18	25	28	30	37	45	76

The values denoted in the first row (0, 1, 2, etc) represents the index numbers and values in the second row (2, 7, 11 etc) are the data-elements stored in the array which is name as X. Suppose, we want to search key value 30 from the array. Here you can see that the values stored in array are in the ascending order. The tracing steps for searching key value 30 is shown below:

Step:1

Initially we will start with a variable beg=0 and end=9 (As we are searching our key value in whole array). We will compute mid now as follows:

mid = (beg + end) / 2 $\Rightarrow$ (0 + 9) /2 = 4.5 $\approx$ 4 (as mid is an integer variable)

Now, X[4] = 25 $\neq$ 30 (value not found at 4th position)

Because key value 30 > 25, beg = mid + 1 $\Rightarrow$ beg = 4 + 1 = 5.

(Key value 30 should be from 5th to 9th position)

Step:2

In the second step variables beg=5 and end=9 (we are searching now in upper half portion in the array). We will compute new mid as follows:

mid = (beg + end) / 2 ⇒ (5 + 9) /2 = 14/2 = 7

Now, X[7] = 37 ≠ 30 (value not found at 7th position)

Because key value 30 < 37, end = mid -1 ⇒ end = 7 - 1 = 6

(Key value 30 should be from 5th to 6th position)

Step:3

In the third step variables beg=5 and end=6

We will compute new mid as follows:

mid = (beg + end) / 2 ⇒ (5 + 6) /2 = 11/2 = 5.5 ≈ 5 (as mid is an integer variable)

Now, X[5] = 28 ≠ 30 (value not found at 5th position)

Because key value 30 > 28, beg = mid +1 ⇒ end = 5 + 1 = 6

(Key value 30 should be from 6th to 6th position)

Step:3

In the fourth step variables beg=6 and end=6

mid = (beg + end) / 2 ⇒ (6 + 6) /2 = 12/2 = 6

Now, X[6] = 30 = 30 (Value Found at Index number 6, at position 7th)

Because value is found, the loop is broken.

Now, for how much time we need to repeat this loop? See, initially we have started the loop by setting our variable beg=0 and end = 9. On every iteration of loop either beg will be increased of end will be decreased, but in every iteration beg < end. When end > beg (means beg and end crosses each other) then, the key element we are searching is not present in the array.

Check Your Progress-2

1. In which searching algorithm, we compare search value with middle element of an array?

[A] Sequential   [C] Binary

[B] Bubble   [D] Radix

---

2. Which searching algorithm can be applied only on sorted array?

[A] Radix

[C] Linear

[B] Binary

[D] Sequential

Important Points about Binary Search

Binary search is a recursive process: Initially we start our searching process by setting beg=0 and end=upper bound of an array. We calculate mid = (beg + end) /2. Depending upon key value is in lower half or upper half, either end or beg will change its value. For newer value of either beg or end, new value of mid should be computed and process will be repeated again.

Termination Condition: The iterations will be terminated when following condition occur.

If beg > end then the segment to be searched absolutely has no elements in it and if there is a match with the data-element present at the middle position of the current segment from stored, then the searching process will be terminated.

Data-elements in the array: As we have discussed that the data-element in an must be in sorted order. Sorted array is the pre-condition for using Binary Search algorithm.

Efficiency Analysis of Binary Search

We have discussed earlier that in the case of Binary Search, in each iteration of the algorithm the block of items is divided in two segments and key item will be searched in any one half. We are dividing a set of n values in half at most $\log_2 n$ times.

Thus, we can say that the running time of Binary search is directly proportional to log n and also, So, here we can say that the time complexity of binary search algorithm is O(log n).

Check Your Progress-3

1. The best-case complexity of the binary search algorithm is _____.

[A] O (n)

[C] O (1)

[B] O (log n)   [D] O (n^2)

2. The worst-case complexity of the binary search algorithm is _____.

[A] O (n)   [B] O (1)

[C] O (log n)   [D] O (n^2)

1.3 Algorithms for Sequential and Binary Search
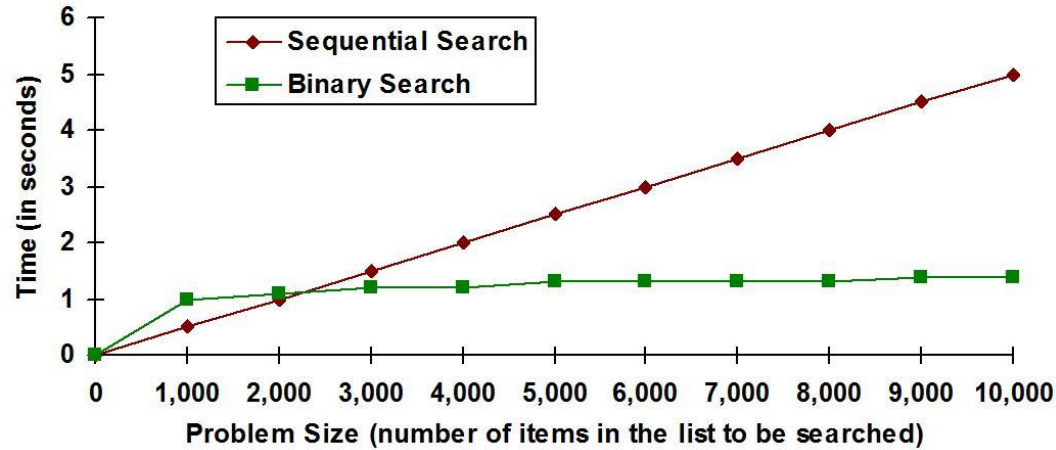
Algorithms for Sequential Search:

When data items are stored in a collection such as an array or in the database files, we can say that they have a sequential or linear relationship. Each and every data-element is stored in a position relative to the others (connected). In Python lists, these relative positions are having the index values of the individual items. Since these index values are ordered accordingly it is possible for us to visit them in sequence respectively. This process is obviously giving rise to our first searching technique, the sequential search**.**

Obviously shows how this search works. Starting at the first item in the array, we simply move from item to item (one by one), following the underlying linear proper order until we either find what we are looking for or run out of items. In case if we run out of items, we have revealed that the item we were searching for was not present.
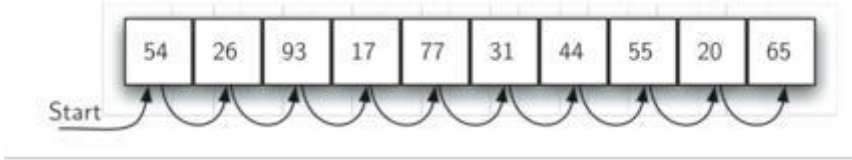
The Sequential Search of a List of Integers

Algorithm for Binary search:

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order.

In each step, the algorithm is comparing the search key value with the data-element located at middle position of the array. If the key value matched with data-element at middle, then a key value found at middle index position, and middle index number is returned. Otherwise, if the search key is less than the middle data-element's value, then the algorithm repeats its action on the sub-array starts from beg position to mid-1 position, if the search key is greater, on the sub-array to the right (part of the array starts from mid+1 to end). If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" error message is returned.

Efficiency Analysis of Binary Search

Here, we will be generally most concerned with the worst-case time, as calculations (or estimations) based on them can lead to guaranteed performance predictions.

Now we can suppose that there are n items in our collection, whether it can be stored as an array or stored as a linked list, then it is obvious in worst case, when we find no item in the collection with the desired key, then n comparisons of the keys with keys of the items in the collection will have to be made (or identified).

See that in case of Binary search, the operation to be performed on the key values is the comparison (key values), since the search requires n comparisons in the worst case; then we say this is an $O(\log_2 N)$ algorithm. Let see the best case, in which the first comparison returns a match, requires only one comparison and is $O(1)$.

The average time depended on the probability that the key will be found in the collection, sometime this we would not expect to know in the majority of cases (most of the cases). Hence in most of the cases, estimation of the average time is of little utility.

Check Your Progress-4
1. The best-case complexity of the linear search algorithm is _____.

[A] O (n)	[C] O (1)
[B] O (log n)	[D] O (n^2)

2. The worst-case complexity of the linear search algorithm is _____.

[A] O (n)	[C] O (1)
[B] O (log n)	[D] O (n^2)

1.4   Implementation of Linear Search

To implement the Linear Search algorithm, we need to write following code in C-Language:

/* Program to implement Linear Search Algorithm, which can be applied on any type of Array, whether it is sorted or un-sorted */

```
#include <stdio.h>
void main ()
{
   int x[10];
   int i, ser_ele;
   /* Accepting values for the Array from the User */
   for (i=0; i<10; i++)
   {
     printf("Enter Value:");
     scanf ("%d", &x[i]);
   }
   /* Accepting Search Element from the user */
   printf ("Enter Element to Search:");
   scanf ("%d", &ser_ele);
   for (i=0; i<10; i++)
   {
     if (x[i] == ser_ele)
     {
       printf ("\nValue found on %d position", i+1);
       break;
     }
   }
if (i ==10)
 {
  printf ("\n Value Does not Exists:");
 }
 }
```

1.5   Implementation of Binary Search

To implement the Binary Search algorithm, we need to write following code in C-Language:

/* Implementation of Binary Search (Can be used only on Sorted Array) */

```c
#include <stdio.h>
void main()
{
  int x[10];
  int i, beg, end, mid, ser_ele;
  /* Accepting values for the Array from the User */
  printf ("Enter Array elements in Sorted order:\n");
  for (i=0; i<10; i++)
  {
    printf ("Enter Value:");
    scanf ("%d", &x[i]);
  }
  printf ("\n Enter Search Element:");
  scanf ("%d",&ser_ele);
  /* Searching Array elements in the array using binary search method*/
  beg=0;
  end=9;
  for (mid=(beg+end)/2; beg<=end; mid = (beg+end)/2)
  {
    if (x[mid] == ser_ele)
    {
      printf ("\n Value found on %d position:", mid+1);
      break;
    }
    else if (x[mid] < val)
      beg = mid+1;
    else
      end = mid-1;
  }
  /* If Search Element does not Exists */
  if (beg > end)
  {
    printf ("\n Value does not Exists:");
  }
}
```

1.6   Let Us Sum Up

This Unit provides the knowledge about two important searching methods. Searching is always important as human stores lots of data for future use. When needed, based on the criteria, the data has to be searched from the storages like data warehouse or database, so that data can be processed and fruitful information can be generated from the past data.

In this unit we have discussed two algorisms to perform search operations. Linear search which is also called sequential search match search key with all the values in the sequential manner. More comparisons will be done in this algorithm, so this algorithm is slower. The plus point of linear search is, it can be used whether the array given is sorted or unsorted.

Another algorithm we have discussed in this unit is Binary Search algorithm. In this algorithm array has to be in sorted order. Rather than comparing search key with each data-element of an array we are comparing it with middle element of an array. Based on this comparison we can take the discission that whether value found, or value should at left side (if search key is smaller), or it should be search in right (if search key is greater)

1.7 Suggested Answers for Check Your Progress

Check Your Progress-1

9. [A] Linear
10. [D] Sequential

Check Your Progress-2

10. [C] Binary
11. [B] Binary

Check Your Progress-3

9. [C] O (1)
10. [C] O (log n)

3. [C] O (1)
4. [A] O (n)

1.8   Glossary

1. **Binary Search** - A binary search which is used only with sorted array, is a fastest search method which divides searching space half on each comparison.

2. **Sequential search -** Sequential search which can be used with sorted or unsorted array, in which search key is compared with all data-members of an array sequentially.

1.9   Assignment

1. Write the applications of a binary search and a sequential search.
2. Write the advantages of a binary search over sequential search.

1.10 Activities

1. Write a C-program, which find the search key given by the user from a Link-List.

2. Express the method to find out search key from a Link-List using Binary Search algorithm?

1.11 Case Study

To search the name of a customer from a large number of databases. if the information of the customer is stored in sorted order, then which method is useful for searching a name with minimum amount of time?

1.12 Further Reading

- Data Structures Using "C" by Tanenbaum.
- Data Structures and Program Design in "C" by Robert L. Kruse.
- Fundamentals of Data Structures by Horowitz and Sahani.

Unit 2: Sorting Techniques ■2

Unit Structure

2.0 Learning Objectives

After learning this unit, you will be able to:

- Understand the concept of Sorting

- Understand the difference between Internal and External Sorting

- Understand the different types of Sorting

- Implement the different Sorting methods

2.1 Introduction

The concept of an ordered set of elements has considerable impact on our daily lives. Consider, for example, if someone is searching for a book in a library. As the books are arranged in a specific order, each book is assigned a specific position relative to the others and can be retrieved in a reasonable amount of time. Similarly, if we want to search a number in a telephone directory, then it will be quite easier as the names in the directory are listed in alphabetical order. So, we can see that sorting plays an important role while searching for a specific data. In this unit, we will be discussing the different sorting methods along with their implementations.

2.2 What is sorting?

Sorting is the process of arranging data items, by following some order. Order can be anything like you can arrange the data-items by following lower to higher order, which is known as ascending order, or data-items can be arranged by following higher to lower, which is known as descending order.

Sorting is an important process, where we are arranging the data-elements, into certain orders, so that when needed, we can search data-element faster. You can understand, in the dictionary thousands of spellings are there. Now thing all those spellings are not arranged into some specific sequence, it will be very difficult for someone to find the spelling. But we know that all the spellings in the dictionary is arranged in ascending order, which makes fining of particular spelling from the dictionary quite simpler and faster.

As we have seen in the previous unit, if the data-elements are stored in the array, are not sorted then we need to compare each element (Linear or Sequential search), which is time consuming process. But suppose the data-elements of an array is arranged or sorted irrespective of order, we can apply binary search algorithm which can find the data faster.

There are many different methods or algorithms are there which can perform sorting operation efficiently. Sorting algorithms can be evaluated by their performance. Different algorithms can complete their sorting operation on the same set of data-elements in different time period. That algorithm is better, complete its sorting process withing shorter time period. Therefore, to evaluate performance of different sorting algorithm, we are considering its time complexity. An algorithm having lesser time complexity is a better algorithm. In this unit, we have discussed different sorting algorithm and their time complexities, we will give you the idea about different types sorting techniques not only that by their time complexity, you can judge, which algorithm is better for your prospective. Usually, the records are stored in the database, based on some field which can used frequently, to search record. For example, in banking application customer's details can be fetched by customer id or bank account details can be fetched by account number.

A field or attribute of a database table, on which the searching of a record is more frequent is called key field or key attribute. Usually, DBMS stores all the record in sorted order by following the key, so when the record is searched on the basis of key value, it can make available record faster.

Check Your Progress-1

1. _____ is a method of arranging data elements in a particular order.

[A] Searching   [B] Sorting

[C] File   [D] None of the above.

2. Records are sorted based on _____.

[A] File   [B] Ascending

[C] Descending   [D] Key

2.3 Types of Sorting
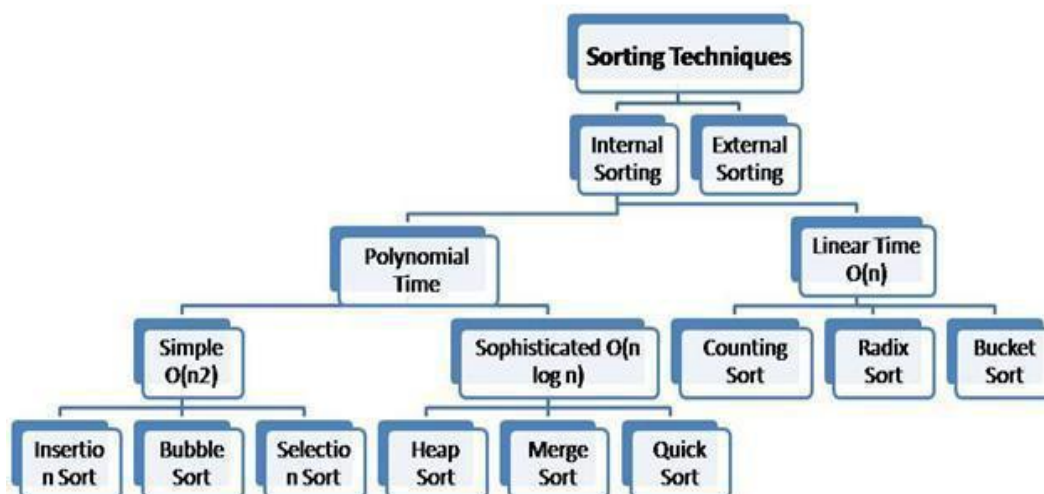
Types of Sorting

There are many different logic and algorithms are there to sort the data. Manly all sorting algorithms can be classified into two main categories. [1] Internal Sorting and [2] External Sorting. Generally, Internal sorting is that which is used to stored the data into main memory. For example, when we sort an array, all data-elements should be there in the main memory. Whereas, External sorting is used when the data-elements to be sorted are present on the secondary memory (storage or auxiliary devices). For example, if the data is stored on disk or any auxiliary memory, in the form of files or databases then we need to use External sorting methods.

2.3.1 Internal and External Sorting algorithms:

Internal Sorting -

Any sorting algorithm which uses main memory completely during the sorting process is an example of Internal Sorting.

Usually, main memory is faster memory, and that why internal sorting algorithm performs faster sorting operations compare to External sorting algorithms. Depending up on complexities Internal sorting algorithms can be classified into two main categories. [1] Sorting algorithms which complete their sorting process in Linear time with complexity O(n) like Counting sort, Radix sort and Bucket sort algorithms and, [2] Those sorting algorithms which requires polynomial time to complete their sorting process.

Algorithms which need polynomial time to complete its sorting process can be further classified into two categories. Some algorithms take n^2 time and hence their time complexities are $O(n^2)$ for n number of elements. These algorithms are placed in Simple sorting algorithms. Bubble sort, Selection sort and Insertion sort are example of it. Some other algorithms which are known as Sophisticated algorithms required $O(n \log n)$ time to complete their sorting process. This are little bit complex algorithms compare to simple sorting algorithms. Quick sort, merge sort and Heap sort are examples of it.

External Sorting -

External sorting is an important sorting process especially in huge business. It is thus vital that it is performed competently. External sorting is used to sort records of files which are very large to fit into the main memory of the computer. This sorting is useful when larger collection of data which is stored on secondary devices. Measuring of time required to access the memory is important in external sorting. External sorting depends on type of device and the number of devices used at the same time.

Merge sort is the most popular method of sorting especially when sorting is performed on secondary memory (External sort). In this method, the huge of amount of data stored in a file, is divided into two segments. Then each segment is again further divided into sub-segment, and the process continues recursively. Now, a small segment is stored in the memory and perform sort operation on it. Another small segment is stored in the memory and it should also be sort. Finally, when all segments are sorted, it starts to merge two small segments into one in such a way that it the merged segment should also be available in the sorted form. When all segments of the files are merged than you will get completely sorted file.

This thing can be easily understood with following example. Think you are examiner and you need to collect all the answer scripts from the student at the end of the examination and then you need to sort it according to their names. Thin what you will do?

Obviously, we make three bunches from the one bunch of unsorted answer scripts. In the first bunch we will put all the answer scripts in which student name starts with A to M. In second bunch we will put all answer scripts in which student name start with N and in third bunch we will place all answer scripts in which student name starts from O to Z. Now you have three bunches, you are picking first bunch of A-M and further split it into A-F, G and H-M, similarly third bunch is further divided into three bunches of O-T, U and V-Z. Continuation of this process will gives you 26 bunches (of all English alphabets). Now, you sort one-one bunch of each alphabet internally and collect all bunches in sorted manner (Take first bunch of letter 'A', then take another bunch of alphabet 'B' and place it

below the bunch of 'A'). This will give you a single sorted bunch of all of your answer scripts.

Difference between Internal and External Sorting

Internal Sorting	External Sorting
1) If the sorting occurs on records which is in main memory then it is called internal sorting.	1) External sorting is that sorting in which records are stored in auxiliary storage devices.
2) It is applied on small collection of data which reside in main memory.	2) It is applied to larger collection of data which resides on secondary memory.
3) Time is not required to read or write.	3) Time required to read or write is considered significant.
4) It is simple sorting method.	4) External sorting is more complex than internal sorting.

2.4 Bubble Sort:

Bubble sort is the easiest method for internal sorting. In this algorithm, we compare the value of a data-element to the value of its neighbour data-element. Let discuss how the data-elements stored in an array will be stored using Bubble sort algorithm with some tracing steps:

Assume that we have the following data in the array of size 5.

<div align="center">

56   22   76   18   11

</div>

In the Bubble sort algorithm, we need to compare neighbouring data-elements like Jth and (J+1)th element. If the value (J+1)th element is smaller than the value of Jth element then we will swap both the values of both elements. In this algorithm, in each iteration of variable I smaller data-elements are coming ahead (same as air in the water, coming up in the form of bubbles) and the largest element we are getting at the end of the array.

Following example given below will shows the tracing on bubble sort algorithm. To do the tracing consider an array of 5 elements as shown below. In the array, you need to consider the index number of value 56 is 0, index number of value 22 is 1, index number of value 74 is 2 and so on. Index number of the data value 11 is 4.

Iteration 1: Value of I =0		
J=0	**56 22** 76 18 11	Here (J+1)th element 22 is compared with Jth data-element that is 56. Because of 22 is smaller than 56, will exchange the values of (J+1)th and Jth element.
J=1	22 **56 76** 19 10	If J=1 then J+1=2; 1st and 2nd elements which are 56 and 76 are compared. 1st element 56 is smaller so no swap is performed.
J=2	22 56 **76 18** 11	If J=2 then J+1=3; 3rd element 18 is smaller than 2nd element 76 so algorithm will swap both values.
J=3	22 56 18 **76 11**	If J=3 then J+1=4; 3rd and 4th value, 76 and 11 will be compared, because 11 is smaller than 76, algorithm will swap the values and we can get the largest number 76 at the end of the list.
	22 56 18 11 **76**	

Iteration 2: Value of I =1		
J=0	**22 56** 18 11 **76**	Here (J+1)th element 56 is compared with Jth element that is 22. Because of 22 is smaller than 56, will not swap the values.
J=1	22 **56 18** 11 **76**	If J=1 then J+1=2; 1st and 2nd elements 56 and 18 are compared. 1st element 56 is greater than 2nd element 18. So, will swap the values.
J=2	22 18 **56 11**	If J=2 then J+1=3; 3rd element 11 is smaller than 2nd

	76	element 56 so, algorithm will swap both values.
	22 18 11 **56** **76**	Here we get the second largest element 56 at the second last position of an array.

Iteration 3: Value of I =2		
J=0	**22 18** 11 **56** **76**	Here (J+1)th element 18 is compared with Jth element that is 22. Because of 18 is smaller than 22, it will swap the values.
J=1	18 **22 11 56** **76**	If J=1 then J+1=2; 1st and 2nd elements 22 and 11 are compared. 1st element 22 is greater than 2nd element 11. So, will swap the values.
	18 11 **22 56** **76**	Here we get the third largest element 22, placed on third last position of an array.

Iteration 3: Value of I =2		
J=0	**18 11 22 56** **76**	Here (J+1)th element 11 is compared with Jth element that is 18. Because of 11 is smaller than 18, will swap the values.
	11 18 22 56 **76**	At the end we get our sorted array as shown in second column.

Algorithm for Bubble sort

Procedure bubble_sort (X: list of sortable elements)

For (i=0; i< n-1; i++) // where n is total number of sortable elements
{
*   For (j=0; j < (n-1) -i; j++)*
*   {*
*   If(X[j+1] < X[j])*
*   {*
*   // Swapping of data-elements X[j] and X[j+1]*
*   Tmp=x[j]*
*   X[j] = X[j+1]*
*   X[j+1] = Tmp*
*   }*

}

}

Time complexity in Best, Average and Worst case as well as worst case space complexity of bubble sort algorithm is described below:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity Worst Case
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$

Check Your Progress-3

1. In the which sorting algorithm two nearby elements are compared?

[A] Selection   [C] Quick

[B] Merge   [D] Bubble

2. Worst case complexity of the bubble sort algorithm is _____.

[A] O (1)   [C] O (n)

[B] O (n^2)   [D] O (log n)

3. From the given below options, which sorting algorithm use nested loops?

[A] Bubble   [C] Quick

[B] Merge   [D] None of the above

2.5 Selection Sort:

Selection sort is another method of Internal sorting. In this method we are selecting or targeting a specific position in the array. Value at the selected position will be compared with all other data-elements of an array, and if any smaller value is found then the values (value of selected position in array and value stored at other position) will be swapped. Consider the following example, in which two variables 'i' and 'j' are taken. Variable 'i' is used to select element (target) in the array, while variable 'j' will point to other elements of an array. We will compare value stored at Ith place with value stored at Jth place. If value stored at Jth place is smaller than these

values will be swapped. In each iteration of the loop of variable 'i' you will get the smallest element in the beginning of the array. Let us see the tracing of a selection sort algorithm by following example.

Assume that we have the following data in the array of size 5.

56   22   76   18   11

Iteration 1: Value of I =0 and J=I+1=0+1=1		
J=1	**56 22** 76 18 11	Here I[th] element 56 is compared with J[th] element that is 22. Because of 22 is smaller than 56, algorithm will exchange the values of Ith and J[th] element.
J=2	**22** 56 **76** 18 11	Here I[th] element 22 is compared with J[th] element that is 76. Because of Jth element 76 is greater than I[th] element, there is no swap
J=3	**22** 56 76 **18** 11	Here I[th] element 22 is greater than J[th] element, that is 18. Because of Jth element is smaller than I[th] element then we will swap I[th] and J[th] elements.
J=4	**18** 56 76 22 **11** **11** 56 76 22 18	Here again J[th] element 11 is smaller than I[th] element, so after swapping we will get...

After completion of the first iteration of variable 'i', we will get the smallest element at the beginning of an array that is:11. Now the same process will be repeated for remaining 4 elements of an array.

Iteration 2: Value of I =1 and J=I+1=1+1=2		
J=2	**11 56 76** 22 18	56 is smaller than 76, so no swap
J=3	**11 56** 76 **22** 18	22 is smaller than 56, so will swap the values
J=4	**11 22** 76 56	18 is smaller than 22, so will swap the values

	18	
	11 18 76 56 22	So, at the end of the second iteration will get second smaller value 18 at 1st position.

Iteration 3: Value of I =2 and J=I+1=2+1=3		
J=3	**11 18 76 56** 22	56 is smaller than 76, so will swap the values
J=4	**11 18 56** 76 **22**	22 is smaller than 56, so will swap the values
	11 18 22 76 56	So, at the end of the third iteration will get third smaller value 22 at 2nd position.

Iteration 4: Value of I =3 and J=I+1=3+1=4		
J=4	**11 18 22 76 56**	56 is smaller than 76, so will swap the values
	11 18 22 56 76	So, at the end of the fourth iteration will get fourth smaller value 56 at 3rd position.

Algorithm for Selection sort

Procedure selection_sort (X: list of sortable elements)

For (i=0; i< n-1; i++) // where n is total number of sortable elements

{

    For (j=i+1; j < n; j++)

    {

        If(X[j] < X[i])

        {

*// Swapping data element of i*th *and j*th *place*

Tmp=x[i]

X[i] = X[j]

X[j] = Tmp

　　　}

　　}

}

Time complexity in Best, Average and Worst case as well as worst case space complexity of selection sort algorithm is described below:

Algorithm				Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$	$O(1)$

Check Your Progress-4

1. The given array is arr = {3, 4, 5, 2, 1}. The number of iterations in bubble sort and selection sort respectively are,

[A] 5 and 4   [C] 4 and 5

[B] 2 and 4   [D] 2 and 5

2. What is the worst-case complexity of selection sort?

[A] O (1)   [C] O (n)

[B] O (log n)   [D] O (n^2)

3. How many iterations will take inner loop for each iteration of outer loop in selection sort?

14, 12,16, 6, 3, 10

[A] 6   [C] 5

[B] 7   [D] 1

4. For the following question, how will the array elements look like after second pass in Selection sort algorithm?

34, 8, 64, 51, 32, 21

[A] 8, 21, 32, 34, 51, 64   [C] 8, 32, 34, 51, 64, 21

[B] 8, 21, 64, 51, 32, 34   [D] 8, 34, 64, 51, 32, 21

2.6 Insertion Sort:

In the insertion sort algorithm, we are inserting a particular value between smaller value than it and greater values than it. This requires shifting of all grater values to the right. The following trace of an Insertion sort algorithm will guide you to understand the logic of an Insertion sort algorithm.

Let us consider the same values in the array of 5 elements, which we have take in the trace of Bubble and Selection sort algorithm, with some minor changes (so that we can cover all different type of cases), which is given below:

56   22   76   11   28

Step:1 In the first step we start with value of variable I=1 and J=I. Now, because of I=1, data-element 22 is our target value. We will copy 22 into tmp variable and we need to compare it will all elements till we reach to index number 0. Here, we will compare Jth element with (J-1)th element if it smaller we need to copy (J-1)th element to Jth place. In this case 22 is smaller than 56, so 22 is inserted before 56 and you will get the following:

22   56   76   11   28

Iteration 1: Value of I =1 and J=1		
J=1	**56 22** 76 11 28	X[I]=Tmp=22, X[J] = 22, X[J-1] =56 56 > 22 hence 22 is inserted before 56
	22 56 76 11	Finally, we made arrangement of data-elements as shown in the

230

	28	left column.

Step:2 In this step I=2 (our target value is 76). We will start a loop of J from 2 to 0. We will compare Jth element 76 with J-1th element 56. Value 76 is not smaller than 56, so we will decrease the value of J by 1.

Iteration 2: Value of I =2 and J=2		
J=2	22 **56 76** 11 28	X[I]=Tmp=76 X[J] = 76, X[J-1] =56 76 > 56 hence J--
J=1	**22** 56 **76** 11 28	X[I]=Tmp=76 X[J] = 56, X[J-1] =22 76 > 22 hence J— After J—value of J becomes 0 and hence there is no change in the array.

Step:3 In this step I=3, therefore our target is 11, which is compare with 76, 56 and 22, because 11 is the smallest among all it is inserted before 22. So, elements 22, 56 and 76 should be shifted one place at right side.

Iteration 3: Value of I =3 and J=3		
J=3	22 56 **76 11** 28	X[I]=Tmp=11 X[J] = 11, X[J-1] =76 76 > 11 hence J--
J=2	22 **56** 76 **11** 28	X[I]=Tmp=11 X[J] = 76, X[J-1] =56 56 > 11 hence J—
J=1	**22** 56 76 **11** 28	X[I]=Tmp=11 X[J] = 56, X[J-1] =22 After J—value of J becomes 0. Here, 11 is smallest element than 22, 56 and 76 therefore all elements are shifted to right and 11 is inserted before 22.
	11 22 56 76 28	Finally, we made arrangement of data-elements as shown in the left column.

Step:4 In this step I=4, therefore our target is 28, which is compare with 76, 56 and 22, and 11. Because 28 is smaller than 76 and 56 as well as greater than 22, therefore 28 should be placed between 22 and 56.

Iteration 3: Value of I =3 and J=3		
J=4	11 22 56 76 **28**	X[I]=Tmp=28 X[J] = 28, X[J-1] =76 76 > 28 hence J--
J=3	11 22 **56** 76 **28**	X[I]=Tmp=28 X[J] = 76, X[J-1] =56 56 > 28 hence J—
J=2	11 **22** 56 76 **28**	X[I]=Tmp=28 X[J] = 56, X[J-1] =22 Here 22 < 28, therefore 28 is inserted after 22.
	11 22 28 56 76	Finally, we made arrangement of data-elements as shown in the left column.

Algorithm for Insertion sort

Procedure insertion_sort (X: list of sortable elements)

For (i=0; i< n-1; i++) // where n is total number of sortable elements
{
   Tmp = X[i];
   For(J=I; J>0 && X[J-1]>Tmp; J--)
         X[J] = X[J-1];

   X[J] = Tmp;
}

Time complexity in Best, Average and Worst case as well as worst case space complexity of insertion sort algorithm is described below:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
				Worst Case
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$

Check Your Progress-4

1. How many nesting of loop is needed to implement insertion sort algorithm.

[A] 2   [C] 1

[B] 4   [D] 3

2. Worst-case complexity for insertion sort is _____.

[A] O (1)   [C] O (n)

[B] O (n^2)   [D] O (log n)

3. What will be the number of passes to sort the elements using insertion sort?

14, 12,16, 6, 3, 10

[A] 6   [C] 7

[B] 5   [D] 1

4. For the following question, how will the array elements look like after second pass in Insertion sort algorithm?

34, 8, 64, 51, 32, 21

[A] 8, 21, 32, 34, 51, 64   [C] 8, 32, 34, 51, 64, 21
[B] 8, 34, 51, 64, 32, 21   [D] 8, 34, 64, 51, 32, 21

2.7 Quick Sort:

In the algorithm of quick sort, we consider one element as a pivot element, and we are trying to place it to its proper place in the array. Which means, after placing pivot element to its proper place all small elements than pivot element should be at the left-side, and larger elements should be at right side of pivot element in an array. If we do the same process in the

set of smaller values (than pivot element), and set of larger elements (than pivot element) then all elements should be placed on their proper position and we will get sorted array. Tracing of first Iteration of Quick sort algorithm is given below:

Consider an array having 10 elements as shown in the figure given below:

12	3	10	14	58	26	18	2	91	4

Now, first element of an array is assumed as pivot element. Here, 12 will be a pivot element. We take two variables 'p' and 'q' initialize as 1 (lower bound +1) and 9 (upper bound) as shown below, last row of the table represents index number of each data-element in the array:

12	3	10	14	58	26	18	2	91	4
Pivot	p								q
0	1	2	3	4	5	6	7	8	9

Now, start a loop of variable 'p' compare p^{th} element of an array with pivot element. If the pth element is smaller, than pivot element then increments variable p by 1. When value of pth element becomes greater than pivot element the break the loop.

Here 3 and 10 are smaller than pivot element 12, and 14 is greater than pivot element 12, so loop of variable 'p' is exited here and variable 'p' has value 3.

12	**3**	**10**	**14**	58	26	18	2	91	4
Pivot			p						q
0	1	2	3	4	5	6	7	8	9

In the same start another loop of variable 'q' from 9 and compare q^{th} element of the array with pivot element 12. If the qth element of the array, is greater than pivot element then decrements variable q by 1. When the value of qth element becomes smaller than pivot element then breaks the loop of variable q.

Here q^{th} element is 4 which is smaller than pivot element 12. So, the loop of variable q will quit on its first iteration.

12	**3**	**10**	**14**	58	26	18	2	91	**4**
Pivot			p						q
0	1	2	3	4	5	6	7	8	9

Now, when both loops are broken then, swap the pth and q^{th} variable of an array.

12	**3**	**10**	**4**	58	26	18	2	91	**14**
Pivot			p						q
0	1	2	3	4	5	6	7	8	9

Repeat the same process again run the loop of 'p' variable, until will not get the greater value than pivot. When p becomes 4 then loop is terminated as 58 is greater than pivot element 12. Start the loop for variable q until we will not get smaller element then pivot. In this case 14, 91 are greater than pivot, and next element 2 is smaller than pivot. So, loop of variable q will stop on index number 7 and value 1.

12	**3**	**10**	**4**	**58**	26	18	**2**	**91**	**14**
Pivot				p			q		
0	1	2	3	4	5	6	7	8	9

Swap again, p^{th} and q^{th} element.

12	**3**	**10**	**4**	**2**	26	18	**58**	**91**	**14**
Pivot				p			q		
0	1	2	3	4	5	6	7	8	9

Again, start the loop of p until we get greater value than 12. So, loop of p variable will stop at index 5, as it has value 26. Start the loop of q until we get the smaller element than pivot. So, q loop will be stopped when q=4, because 58, 18 and 26 are greater than 12 and 2 is smaller than 12.

12	**3**	**10**	**4**	**2**	**26**	**18**	**58**	**91**	**14**
Pivot				q	p				

When, variable p and q cross each other (p > q), then swap q^{th} element with pivot.

2	3	10	4	12	26	18	58	91	14
Pivot				q	p				

0	1	2	3	4	5	6	7	8	9

Here pivot element 12 is placed on its proper place as all the left-side elements are smaller than 12, and right-side elements are greater than 12. Here, pivot is placed on position 4, so repeat the same process recursively on the part of an array from 0 to 3, as well as 5 to 9. If we continue this process, then all data-elements will be placed to their proper place in an array and we get the sorted array.

Algorithm for Quick sort

Procedure quick_sort (X: list of sortable elements)

```
{
      int i ;
      if ( upper > lower )
      {
             i = split ( a, lower, upper ) ;
             quicksort ( a, lower, i - 1 ) ;
             quicksort ( a, i + 1, upper ) ;
      }
}
int split ( int a[ ], int lower, int upper )
{
      int pivot, p, q, t;
      p = lower + 1;
      q = upper;
      pivot = a[lower];
      while ( q >= p )
      {

             while ( a[p] < pivot )
                   p++;
             while ( a[q] > pivot )
                   q--;
             if ( q > p )
```

```
            {
                    t = a[p];
                    a[p] = a[q];
                    a[q] = t;
            }
    }
    t = a[lower];
    a[lower] = a[q];
    a[q] = t;
    return q;
}
```

Time complexity in Best, Average and Worst case as well as worst case space complexity of quick sort algorithm is described below:

Algorithm				Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Quick Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n^2)$	$O(\log n)$

Check Your Progress-6

1. Which if the following algorithm is fastest.

[A] Merge sort   [C] Quick sort

[B] Insertion sort   [D] Shell sort

2. What is the worst-case complexity of Quick sort?

[A] O (1)   [C] O (n)

[B] O (log n)   [D] O (n^2)

3. What is the average running time of a quick sort algorithm?

[A] O (n^2)   [C] O (n log n)

[B] O (log n)   [D] O (n)

2.8 Merge Sort:

Merge sort algorithm uses the following three steps to sort an array:

1. **Divide:** In this process, we are recursively splitting the array in two parts. We find middle, that is simply mean of lower bound and upper bound of an array, and split the array from the middle. So, array of n elements is split into two arrays of size n/2. We will continue the process dividing arrays recursively unless we are not getting individual element.
2. **Conquer:** In this process we perform sorting, two sub arrays recursively using merge sort.
3. **Combine:** In this process we merge two sorted arrays of size n/2, to generate sorted array of size n.

    To understand these processes discussed above, consider the following example.

46	10	29	40	11	21	51	42

Consider an array given above. We want to split the array using merge sort algorithm. So, first we use split method which will divide the array of n elements into two sub-arrays of size n/2. Once again, these two sub-arrays are further divided in two-two sub-arrays recursively till we not get individual element.

46	10	29	40	11	21	51	42

46	10	29	40		11	21	51	42

46	10		29	40		11	21		51	42

46		10		29		40		11		21		51		42

Once, we divide an array recursively till, each element of an array is separated, we will start next process of merging and sorting the arrays as shown below:

| 46 | 10 | 29 | 40 | 11 | 21 | 51 | 42 |

| 10 46 | 29 40 | 11 21 | 42 51 |

| 10 29 40 46 | 11 21 42 51 |

| 10 | 11 | 21 | 29 | 40 | 42 | 46 | 51 |
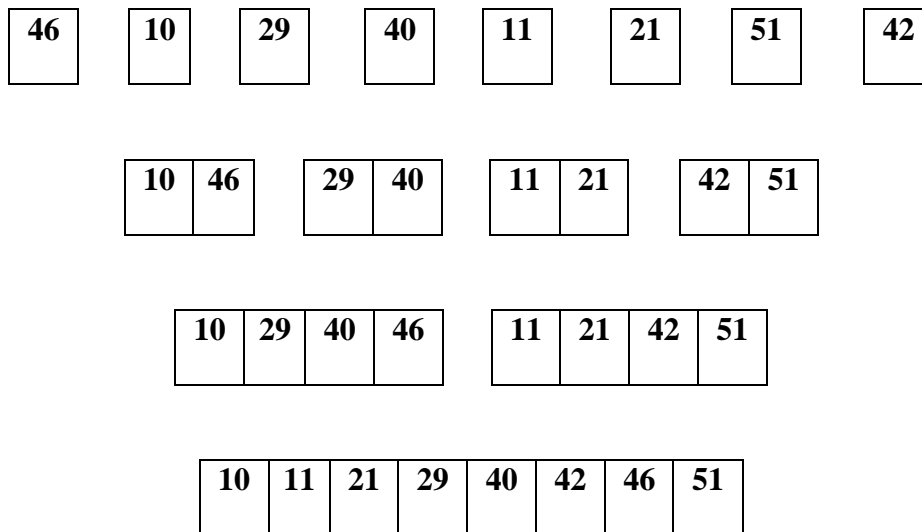
To understand the process of merging in more details, consider the following example:

In the merging process we have an array of n elements, divided into two sub-arrays of size n/2. Both the sub-arrays are sorted array. To generate sorted merged array from the given two sorted sub-arrays of size n/2, we take an extra array TEMP. We will take variable i=0 and j=mid+1, to refer first elements of both the sorted sub-arrays.

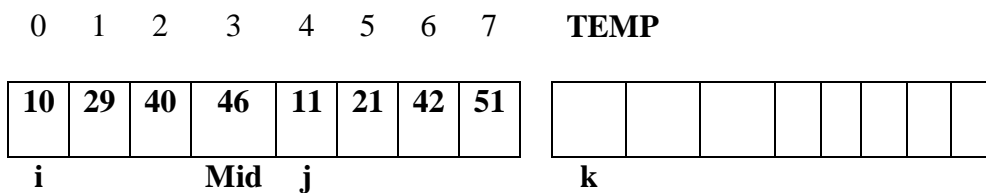|	0	1	2	3	4	5	6	7	**TEMP**
	10	29	40	46	11	21	42	51									
	i			Mid	j				k

Consider an array contains two sorted sub-arrays. From Index number 0 to Mid which is 3, we have one sorted sub-array which has elements 10, 29, 40 and 46. In the same way from Mid+1 which is 4 to upper bound of an array that is 7, another sorted sub-array which has 11, 21, 42 and 51. We start with i=0 and j=Mid+1=4 which indicates first elements of both sorted sub-arrays. We will compare i^{th} element with j^{th} element. Because i^{th} element is 10, which is smaller than j^{th} element 11, we will copy smaller element 10 to the TEMP array on k^{th} position (variable k is initialized with 0, which is used to reference TEMP array). Now because of we have copied i^{th} element, will increment variables i and k as shown below:

TEMP

| 10 | 29 | 40 | 46 | 11 | 21 | 42 | 51 | 10 | | | | | | | |

i   **Mid**   **j**   **k**
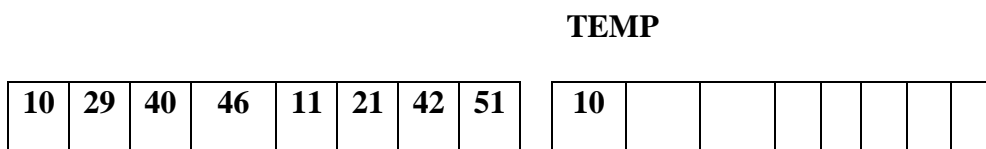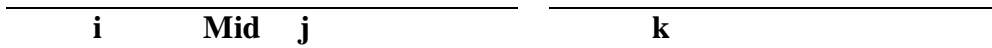
Now, again we will compare i^{th} element 29 and j^{th} element 11. Because j^{th} element 11 is smaller than i^{th} element 29, we will copy j^{th} element 11 of an array to TEMP array at k^{th} position. Because we have copied j^{th} data-element variable j and k will be increased by 1.

TEMP

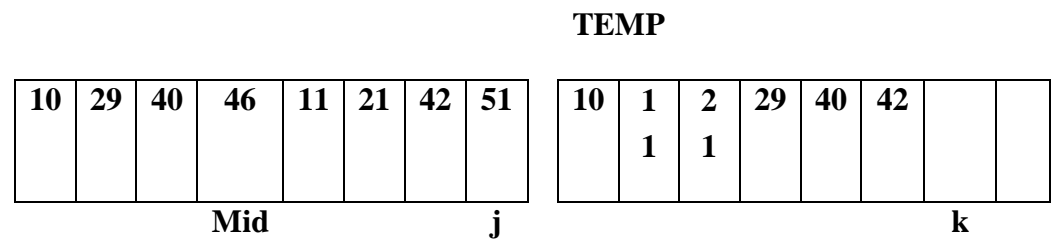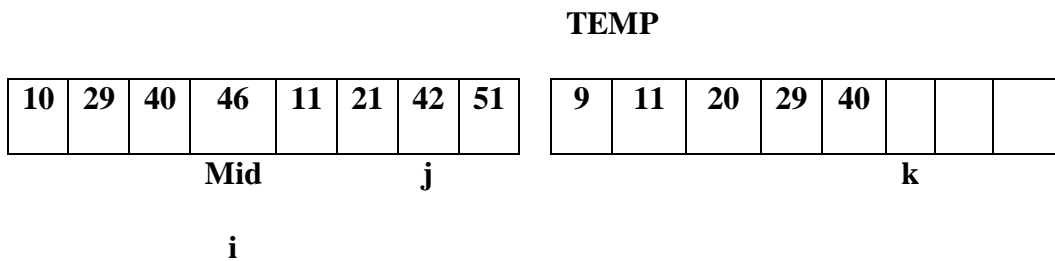| 10	29	40	46	11	21	42	51		10	11						

i   **Mid**   **j**   **k**
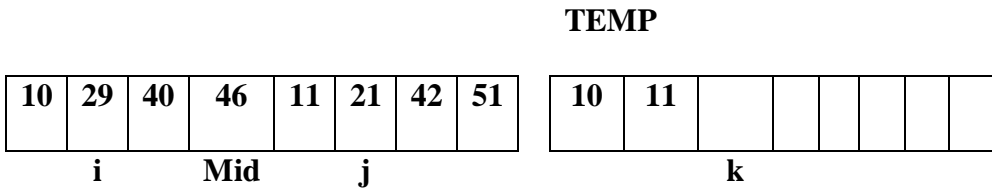
The similar process is repeated again and again as shown below:

TEMP

10	29	40	46	11	21	42	51		10	11	21					

i   **Mid**   **j**   **k**

TEMP

10	29	40	46	11	21	42	51		10	11	21	29				

       i   **Mid**   **j**   **k**

TEMP

10	29	40	46	11	21	42	51		9	11	20	29	40			

          Mid   **j**   **k**

         i

TEMP

10	29	40	46	11	21	42	51		10	11	21	29	40	42		

         Mid   **j**   **k**

i

TEMP

10	29	40	46	11	21	42	51	10	11	21	29	40	42	46	
		Mid				j									k

i
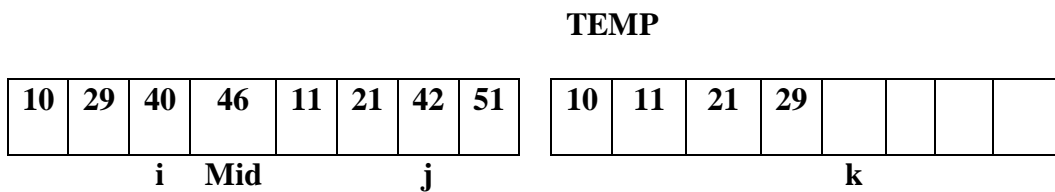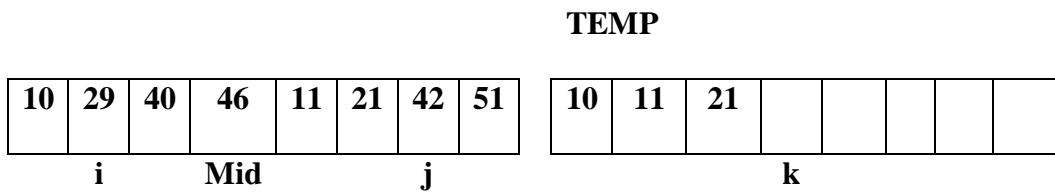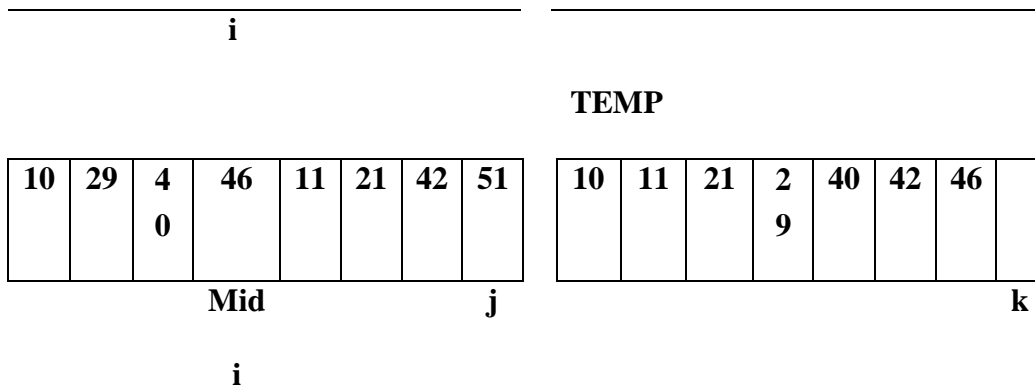
When all the elements of any one sub-array copied to the TEMP array, rest of the elements of other sub-array has to be copied to the TEMP array. In this example variable i reached at Mid and that element is also copied in the TEMP array, so left array has been copied entirely. Now we have to copy all the pending elements of the sub-array resided at right side. Right sub-array has only one element pending, which is 51, so copy 51 to the TEMP array at kth position.

TEMP

9	28	39	45	11	20	41	50	9	11	20	28	39	41	45	50
		Mid			j										k

i
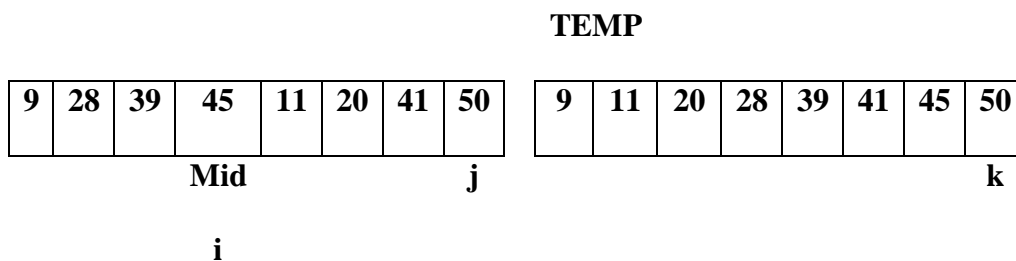
After completion of this process, TEMP array is merged and sorted. We will copy all elements of TEMP array to the original array.

Algorithm for Merge sort

Procedure merge_sort (X: list of sortable elements)

```
{
   int mid;
   if (beg < end)
   {
     mid = (beg + end) / 2;
     merge_sort (x, beg, mid);
     merge_sort (x, mid+1, end);
     mrg (x, beg, mid, end);
   }
}
```

```
void mrg (int *x, int beg, int mid, int end)
{
    int i=beg, j=mid+1, k=beg, temp[10];
    while((i<=mid) && (j<=end))
    {
        if (x[i] < x[j])
        {
            temp[k] = x[i];
            i++;
        }
        else
        {
            temp[k] = x[j];
            j++;
        }
        k++;
    }
    if (i > mid)
    {
        while (j <= end)
        {
            temp[k] = x[j];
            k++;
            j++;
        }
    }
}
```

Time complexity in Best, Average and Worst case as well as worst case space complexity of merge sort algorithm is described below:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity Worst Case
Merge Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n \log n)$	$O(n)$

2.9 Let Us Sum Up

In this particular Unit we have made detailed discussion on sorting algorithms. Sorting, which is a method of arranging data-elements into either ascending or descending order. We have discussed different types of algorithms like Bubble sort, Selection sort, Insertion sort, Quick sort and Merge sort. We have also discussed that the sorting algorithms can be classified into tow categories depending upon memory they used for the sorting process.

Internal Sorting: Any sort algorithm which uses main memory solely during the sorting process, is an example of Internal Sorting. For Example: Bubble Sort, Insertion Sort etc.

External Sorting: External sorting is also an important activity especially used in large business, where the business transactions are stored in the files (in the form of records) on the secondary (slow) memory. Therefore, it is crucial that it is performed efficiently.

External sorting is applied to sort records of files which are too large to fit in the main memory of the computer. This sorting is applied to larger collection of data which rests on secondary devices. The most popular method for sorting on external storage device is Merge sort.

2.10 SUGGESTED ANSWERS FOR CHECK YOUR PROGRESS

Check Your Progress-1

1. [A] Searching
2. [D] Key

Check Your Progress-2

1. [C] External
2. [A] Internal
3. [D] All of the above
4. [A] Bubble sort
5. [D] Merge sort

Check Your Progress-3

1. [D] Bubble
2. [B] $O(n^2)$
3. [A] Bubble

Check Your Progress-4

1. [A] 2
2. [B] $O(n^2)$
3. [B] 5
4. [D] 8, 34, 64, 51, 32, 21

Check Your Progress-5

1. [A] 5 and 4
2. [D] O (n^2)
3. [B] 5
4. [C] 8, 21, 64, 51, 32, 34

Check Your Progress-6

1. [C] Quick sort
2. [D] O (n^2)
3. [C] O (n log n)

Check Your Progress-7

1. [C] Quick sort
2. [D] O (n^2)
3. [C] O (n log n)

2.11 Glossary

1.   **Pivot Element -** Element in array which is compared with all other elements.

2.   **Sorting -** Sorting is a method of arranging data elements in a particular order. The arrangement of the data is either lower element to higher (Ascending) or higher element to lower (Descending).

3.   **Internal Sorting - If** the sorting occurs on records which is in main memory, then it is called internal sorting.

4.   **External Sorting -** External sorting is that sorting in which records are stored in auxiliary storage devices.

5.   **Radix sort -** Radix sort is one of the linear sorting algorithms for integers.

6.   **Bubble sort** – Bubble sort is a sorting algorithm, which sorts the data by comparing two nearby data-elements. The worst-case complexity of the bubble sort algorithm is $O(n^2)$.

7.   **Selection sort** – Selection sort is a sorting algorithm, which sorts the data by comparing all other element with selected data element. The worst-case complexity of selection sort algorithm is $O(n^2)$.

8.   **Searching** – Searching is the process of finding the position of particular element in the array. To search a data element into the Array there are two methods are there: [1] Linear search and [2] Binary search.

9.   **Linear search** – Linear search is the simplest algorithm in which we compare the search element with all other elements. The time complexity of Linear search is higher ($O(n)$). Therefore, it is slower algorithm. It can be applied on any kind of array, whether it is sorted or unsorted.

10.   **Binary search** – Binary search is a searching method in which, we are comparing search element directly to the element situated at the middle of the array. If the value is smaller than the element at the middle then it might be from 0 to mid-1. If it is greater than it might be between mid+1 to upper bound of the array. It can be applied on the sorted array only. It is faster than Linear search as the complexity of binary search is $O(\log n)$.

2.12   Assignment

1. Prepare a table having all searching and sorting algorithms and their different types of complexities are depicted in the form of table.
2. Explain the tracing steps of the following data, using Selection sort, Bubble sort and Insertion sort:
   65, 35, 9, 51, 22, 37

3. Explain the tracing steps of the following data using Quick sort algorithm.
   11, 3, 19, 14, 58, 28, 17, 2, 90, 3

4. Explain the tracing steps of the following data using Merge sort algorithm.
   45, 7, 28, 38, 11, 21, 49, 40

2.13 Activities

1.   What are the types of sorting algorithms? Explain it with examples.

2.   What is Pivot-Element?

2.14 Case Study

To apply sorting on data this is present in industry related to stock details of product. Which Algorithm is suitable to sort product details on basis of demands from customer?

2.15 Further Reading

1.   Data Structures Using "C" by Tanenbaum.

2.   Data Structures and Program Design in "C" by Robert L. Kruse.

3.   Fundamentals of Data Structures by Horowitz and Sahani.

4.   Data Structures: An Advanced Appraoch Using 'C' by Esakov and Weises.

5.   Data Structures and 'C' Programming by Cristopher J. Vanwyk.

Unit 3: File Structure ▎3

Unit Structure

3.0   Learning Objectives

After learning this unit, you will be able to:

- Understand file structure- concept of fields
- Understand files and Records
- Understand sequential and Index file organizations
- Understand hashing Techniques

3.1   Introduction

This unit is mainly focused on the way in which file structures are used. There is one important change that must be made at the outset when discussing file structures, and that is the difference between the logical and physical organisation of the data. File structure will specify the logical structure of the data that is the relationships that will exist between data items.

3.2   File structure- concept of fields
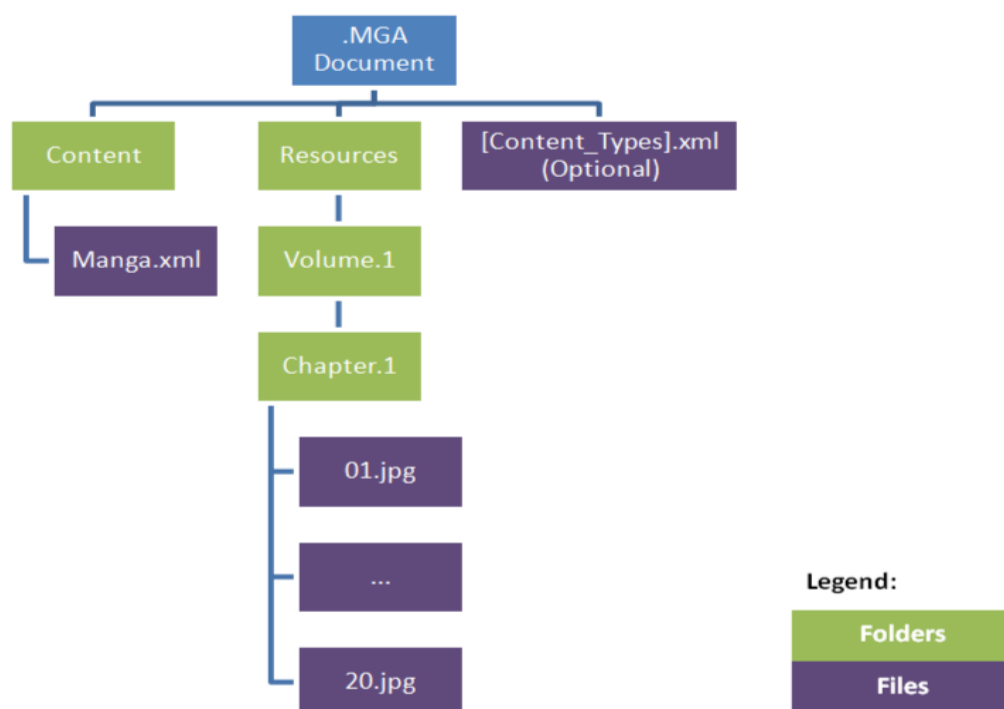
3.2 File structure

File Operations:

Operations on database files can be categorized into two broad categories:

Update Operations and Retrieval Operations

Update operations refers to the changes in the data values by insertion, deletion or update. Retrieval operations on the other hand is just use to retrieve data and do not alter (or change) the data. It is simply used to fetch (select) them after filtering the data on the basis of some criteria. In both the types of operations, selection operation plays significant role. Apart from creating and deleting a file, there are some other operations, which can be performed on files which are listed and explained as below:

3   **Open** - This process takes care of opening the file and further more files can be opened in two modes (manners), write-mode or read mode. In the read-mode, operating system does not allow any kind of alteration (changes) in data, it is exclusively for reading purpose. Files, which are opened in read mode can be shared among numerous entities. Another mode is write-mode, in which, modification of data is allowed. Files, which are opened in write-mode can be read, but cannot be shared.

4   **Locate** – File should be accessed by file pointer (indicator), which represents the current position, where the data is to be read or written. This pointer can be adjusted depending upon requirement. Using the seek operation it can be moved any location within the file.

5   **Read** – Default mode, when any file is opened is read-mode. The file pointer is set to the beginning of file. There are certain options where the user can request the operating system to where the file pointer should be placed at the time of opening of a file. We can read very next character from the file pointer is placed.

6   **Write** - In this mode user can select to open files in write-mode, which allows them to edit the content of file. Write-mode allows deletion, insertion or modification in data. The file pointer can be pre-set at the time of opening or can be dynamically changed if the operating system allows to do so.

7   **Close** - This operation is also important operation by the point of view of operating system. When a request to close a file is initiated, the operating system eliminates all the locks (if it is in shared mode) and saves the content of data (if changed) to the secondary storage and free all the buffers and file-handlers related with the file.

The organization of data-content which is also known as records, inside the file plays a key role here. Seeking the file pointer on some desired record inside file performs differently if the file has records arranged sequentially or clustered, and so on.

Check Your Progress-1

1. Read are Write are the modes of _____ file operation.

[A] Locate   [C] Open

[B] Close   [D] None of the above

2. _____ file operation, writes all file blocks to the secondary memory and release all the buffers.

[A] Create   [C] Locate

[B] Open   [D] Close

3. To set the file position in forward or backward directions, _____ function is called.

[A] seek   [C] peek

[B] read   [D] All of the above

3.3   Files and Records

3.2 Files and Records

A record is an arrangement of values or a series of characters. As per FORTRAN, there are three kinds of records are there, which are as follows:

➢ **Formatted** - A record consists of formatted data, that requires translation from internal to external system. The formatted I/O statements have explicitly specifies format or name list specifies, only formatted I/O statements can read formatted data.

➢ **Unformatted** - A record containing unformatted data in a file, that is not translated from internal to external system. An unformatted record can also contain no data. The internal representation of unformatted data is dependent on the processor. Only unformatted I/O instructions can read unformatted data.

➢ **Endfile** - This indicator is used for last record of a file. An endfile record can be explicitly written to a sequential file by an ENDFILE statement

A file is a sequence of records can represent different entities. There are two types of Fortran files, are there, which are as follows:

➢ **External** – While operating a file, if file exists or available on some external devices like in the secondary memory like disk or any auxiliary memory, then it will be treated as external file to the process.

Records in an external file must be either all formatted or all unformatted (as per the requirement). There are three methods to access the records stored in external files: [1] sequential, [2] Keyed access (VMS only), and [3] Direct access.

In sequential access, records are processed in the order in which they appear in the file. In direct access, records are selected by record number, so they can be processed in any order. In keyed access, records are processed by the value of some key field.

➢ **Internal** - Memory or internal storage behaves like a file. This type of file provides a way to transfer and convert data in memory from one format to another (one form to the other form). The contents of these files are stored as scalar character variables.

Check Your Progress-2

1. Data is organized, into the file is known as _____.

[A] field   [C] record

| [B] file | [D] table |

2. A record containing _____ data, that is not translated from internal form.

| [A] formatted | [C] external |
| [B] unformatted | [D] internal |

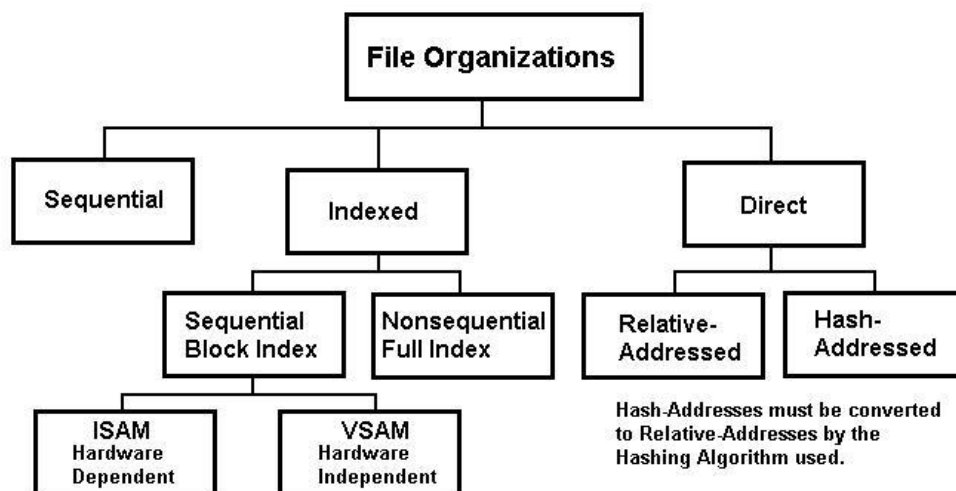3. From the given below which is not a correct method of accessing records from external files:

| [A] Sequential | [C] Keyed (VMS only) |
| [B] Direct | [D] Formatted |

3.4   Sequential and Index file organizations

3.3 Sequential and Index file organizations

File organization is a method of mapping file records, with blocks of a disk that is, how the file records are organized. The file organization can be classified as follows:

- **Sequential File Organization** - mostly in every file record, there should be a data field (attribute) which uniquely identify that record. In sequential file organization mechanism, records are placed in the file in some sequential order based on that unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form. Sometime it should be kept as per the requirement of the record to be processed.

- **Sequential Files -** A sequential file is the most primitive (modern) of all file structures. It should not have directory and no linking pointers. The records are usually organized in lexicographic (one after the other) order on the value of some key attribute. In other words, a specific attribute is chosen, whose value will regulate the order of the records. On some occasion when the attribute value is constant for a large number of records, a secondary key is chosen to provide an order when the first key fails to discriminate.

  For the implementation of this file structure requires the use of a sorting routine operation.

- The main advantages are:

  ☐ Easy to implement;
  ☐ It provides fast access to the next record (In order) using lexicographic order.

- Its disadvantages:

  4. It is difficult to update(operation) - inserting (operation) a new record may require moving a large proportion of the file;
  5. Extremely slow in case of Random access.

- Few times a file is considered to be sequentially organized contempt the fact that it is not ordered conferring to any key. Possibly the date of acquisition is considered to be the key value (on the basis of date); the newest entries are added to the end of the file and hence pose no difficulty to updating.

o **Indexed   files -** The file is made of "data cells", it is not necessary that it should be of the same size, and contain "indexes", lists of "pointers" to these cells arranged by some order.

o **Index-sequential files -** When we talk about an index-sequential file then it is an inverted file in which for every keyword Ki, we have ni = hi = 1 and a11 <a21 . . . <am1. In this case we can only arise if each record has just one unique keyword (no duplication), or one unique attribute-value. In practice these set of records may be order sequentially by a key. Each and every key value appears (in list) in the directory with the associated address of its record. The record number will be an obvious interpretation of a key of this kind would be the record number. In this example none of the attributes would do the job except the record number.

Check Your Progress-3

1. A _____ file is the most primitive (modern) of all file structures.

[A] Direct   [C] Index-sequential

---

2. In _____ file, the records are generally organized in lexicographic order (one after the other) on the value of some key (assigning a key to it).

[A] Sequential   [C] Direct

[B] Index-sequential   [D] All of the above

3. _____file, has no directory and no linking pointers

[A] Sequential   [C] Direct

[B] Index-sequential   [D] All of the above

4. _____ file has unique keywords (no duplication), or one unique attribute-values.

[A] Data   [C] Index

[B] Program   [D] Driver

3.5   Hashing Techniques

We must access an index structure to locate data, or must and should use binary search, and that results in more I/O operations (function). Space taken by index structure.

Hashing allows us to avoid (to leave) accessing an index structure.

We obtain (get) the address of the disk block containing a desired record directly by computing a function on the search-key value of the record.

Now Bucket is a unit of storage (memory) that can store one or more records.

A bucket can be said as atypical disk block, can be chosen to be smaller or larger than a disk block.

❖   A hash function h is a function from K to B.

h – hash function Ki - search key Bi = h(Ki)

If h(K7) = h(K5)

bucket h(K5) contains records with

search-key values K5 and records with search-key values K7

❖ The worst possible hash function maps all search-key values to the same bucket

❖ An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records

❖ We want to choose a hash function that assigns search-key values to buckets

Linear hashing:

Linear hashing can be viewed on as a dynamic version of hashing with separate chaining: the overflows are chained together in a separate overflow area, but the size of the file grows (or shrinks) incrementally accordingly, page by page, keeping the file density approximately constant. In the following illustrations, records are represented by letters x, y, z, ..., the blocking factor y = 3 (i.e., three records per page), the file density is $\alpha = n/m = 2/3$, where n is the number of records in the file, m is the file capacity (in records).

1. Initial situation:

basic allocation (always a power of 2) is p0 = 8 pages (file depth d = 3), current allocation p1 equal to basic allocation, n = 16 records in the file, ($\alpha = 16/24 = 2/3$).

g

| 0	1	2 '	3	4	5	6	7
xyz	d	h i j	l m	n o q	s		ef

p0=p1

Split pointer

The hash function values (pseudo-keys) of records a, b, c has suffix (i.e., low order bits) 000, for record d - suffix 001, for records g, h, i, j - suffix 010, etc. The suffix determines the address of page where the record is stored. The length of the suffix is equal to the file depth.

Page No 2 has already overflown.

The "split pointer" points at the next page to be split.

The file grows from p1 to p1+1 pages every $\alpha \cdot y = (2/3) \cdot 3 = 2$ insertions. Linear hashing (2)

2. After record r with pseudo key suffix 0100 has been inserted:

g   r

0	1	2	3	4	5	6	7	8
xyz	D	h i j	l m	n o q	s		ef	z

p0   p1
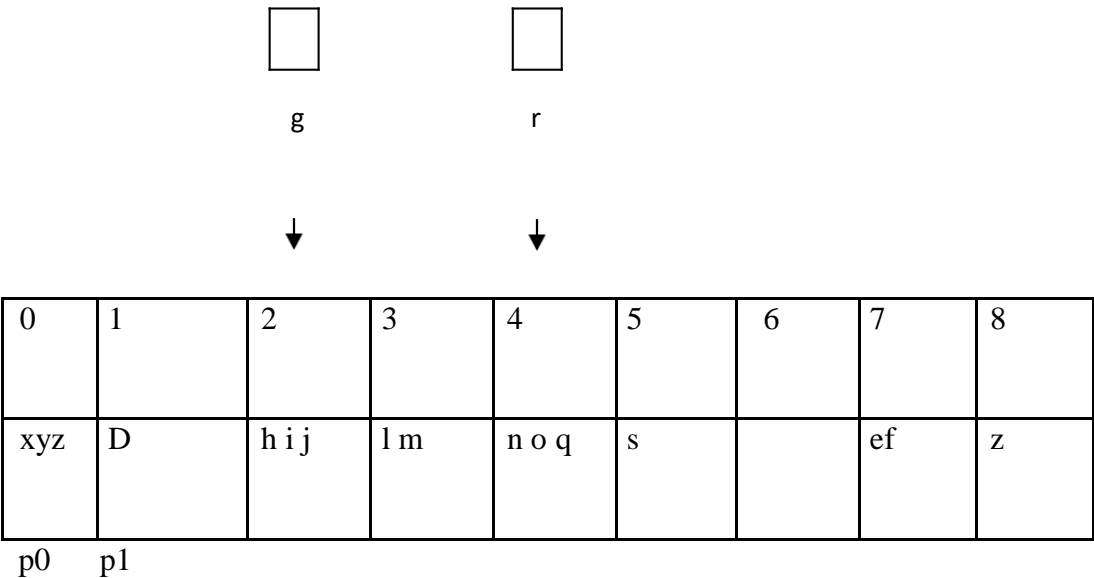
As the file density would exceed 2/3, so:

- the new page is allocated to the file,
- the page pointed at by the split pointer (the page with address p1-p0) is split,
- Current allocation p1 increases by one: p1 = 9,
- The file depth increases by one,
- The records from the split page (x, y, z) are rehashed with the pseudo key extended by one bit to the left: records x, y that have suffix 0000, remain on page 0; record z, that has suffix 1000, is moved to the newly allocated page 8.
3. After next 2 insertions: record k (suffix 0011) and w (suffix 0110), the next page (address 1) is split.
   The record d has the pseudo-key suffix 0001, so it remained on page 1. In this case the newly allocated page 9 is empty at the moment. Linear hashing

   (3)

4. After next 2 insertions: record f (suffix 1001) and t (suffix 0101), the next page (address 2) together with its overflow page is split.

The process continues until p1 becomes 2·p0. Then, the whole process starts anew with the value of p0 doubled and the file depth increased by one:

0	1	2	3	4	5		6	7	8	9	10		11	12	13	14	15
xy	d	g h	k l m	n q	s		w	x	z	f	i j		...	...	...	...	...

↑ p0=p1

Note that the split pointer points at page 0 again. Linear hashing (4)

For the linear hashing scheme to work properly, the hashing function KAT must dynamically modify according to the current file depth d. For a given key k, KAT produces an address in the range $0...2 \cdot p0 - 1$, so that always d bits are available.

Then, the generated address is reduced to the current range $0...p1 - 1$:

address: = KAT d(k)

IF address > p1-1 THEN address: = address - p0

The records from the split page (possibly - with overflow pages) have either the original address ("0" on the extra bit) and they stay in their home page, or have the new address p1 ("1" on the extra bit), referring to the new page.

When the file has grown to twice its size (p1 = 2·p0), KAT is modified to produce another bit with the low order bits remaining the same, the values of p0 and p1 are doubled and the process continues starting with p1 = p0.

If there are deletions (erase of record), the whole procedure can work the other way round, with merging appropriate pages instead of splitting.

Performance of linear hashing may be estimated in a similar way as for separate chaining. The cost of accessing a record is 1 access (address calculated by KAT) plus expected cost of traversing the overflow chain. $S+ = 1 + \alpha/2$.

3.6 Let Us Sum Up

In this Unit, the high level of understanding relating to File Organization which has to be implemented as per the organization / company requirement so that data retrieval speed gets minimum so by using efficient method we can store it on disk.

There is understanding related to file and its operations, Operations on database files can be classified into two categories broadly:

Update Operations and Retrieval Operations change the data values by insertion, deletion or update. Retrieval operations (or function) on the other hand do not alter (or change) the data but retrieve them after optional conditional filtering. In both types of operations, selection operation plays significant role. To carry out these operations there are

having some important basic operations need to be performed like Open, Locate, Read, Write, Close.

Now further we have learnt about a record. A record is a sequence of values or a sequence of characters. There are three kinds of FORTRAN records, such as Formatted, Unformatted, End file. After this we have continue understanding about a file which is a sequence of records. There are two types of Fortran files as external and Internal

There is also learning related to Hashing which allows us to avoid accessing an index structure. We obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record Bucket is a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block. In mean time we understood about Linear hashing.

Linear hashing can be viewed on as a dynamic version of hashing with separate chaining: the overflows are chained together in a separate overflow area, but the size of the file grows (or shrinks) incrementally, page by page, keeping the file density approximately constant.

3.7   Suggested Answer for Check Your Progress

Check Your Progress-1

1. [C] Open
2. [D] Close
3. [A] Seek

Check Your Progress-2

1. [C] Record
2. [B] Unformatted
3. [D] Formatted

Check Your Progress-3

1. [B] Sequential
2. [A] Sequential
3. [A] Sequential

4. [C] Index

1. [A] Hashing
2. [D] Bucket
3. [B] 8
4. [C] Search Key

3.8 Glossary

a) **Hashing -** Hashing a technique to calculate direct location of data record on the disk without using index structure.

b) **Linear hashing -** Linear hashing can be viewed on as a dynamic version of hashing with separate chaining

c) **Separate chaining -** the overflows are chained together in a separate overflow area, but the size of the file grows (or shrinks) incrementally, page by page, keeping the file density approximately constant.

d) **Split pointer -** The "split pointer" points at the next page to be split.

3.9 Assignment

Do the sequential indexing on 1000 no. of records present in database.

3.10 Activities

1. What is mean by Indexed Sequential files?

2. Write methods to implement hashing using static and dynamic hashing.

3.11   Case Study

Consider data stored in college database which is used by committee for checking the progress of college for last 20 years then how could we find the records of all last year student details which are gets passed through this college.

4.   Which organization of files is using full to perform searching on Class of students?

3.12 Further Reading

1.   Data Structures Using "C" by Tanenbaum.

2.   Data Structures and Program Design in "C" by Robert L. Kruse.

3.   Fundamentals of Data Structures by Horowitz and Sahani.

4.   Data Structures: An Advanced Appraoch Using 'C' by Esakov and Weises.

5.   Data Structures and 'C' Programming by Cristopher J. Vanwyk.

Unit 4: Programs of Searching and Sorting

<div style="text-align:right">**4**</div>

Program: 1 Write a program to implement linear search

```
#include<stdio.h>
void main()
{
   int x[10];
   int i,se;
   // Accepting values for the Array from the User
   for(i=0;i<10;i++)
   {
      printf("Enter Value:");
      scanf("%d",&x[i]);
   }
   // Taking Search Element from the user
   printf("Enter Search Element:");
   scanf("%d",&se);
   for(i=0;i<10;i++)
   {
      if(x[i]==se)
      {
         printf("\nValue found on %d position", i+1);
         break;
      }
   }
   if(i==10)
   {
      printf("\nValue Does not Exists:");
   }
}
```

Program: 2 Write a program to implement binary search

```
#include<stdio.h>
void main()
{
   int x[10];
   int i,beg,end, mid,val;
```

```c
// Accepting values for the Array from the User
printf("Enter Array elements in Sorted order:\n");
for(i=0;i<10;i++)
{
  printf("Enter Value:");
  scanf("%d",&x[i]);
}

printf ("\nEnter Search Element:");
scanf ("%d", &val);
//Searching Array elements in the array
beg=0;
end=9;
for(mid=(beg+end)/2; beg<=end; mid=(beg+end)/2)
{
  if(x[mid]==val)
  {
    printf ("\nValue found on %d position:", mid+1);
    break;
  }
  else if (x[mid]<val)
  {
    beg=mid+1;
  }
  else
  {
    end=mid-1;
  }
}
//If value does not Exists
if (beg > end)
{
  printf("\nValue does not Exists:");
}
}
```

Program: 3 Write a program to implement selection sort

```
#include<stdio.h>
void main()
{
   int x[5];
   int i, j, tmp;
   // Accepting values for the Array from the User
   for (i=0; i<5; i++)
   {
      printf ("Enter Value:");
      scanf ("%d", &x[i]);
   }
   //Printing unsorted array
   printf ("\n Array Before Sorting:\n");
   for (i=0; i<5; i++)
         printf ("\t%d", x[i]);
//Sorting an array

   for (i=0; i<4; i++)

   {

      for (j=i+1; j<5; j++)

      {

         If (x[i]>x[j])

         {

            //If Ith element is Greater than Jth element then swap the value

            tmp = x[i];

            x[i] = x[j];

            x[j] = tmp;

         }

      }

   }

   //Printing Sorted Array

   printf ("\n Array After Sorting:\n");

   for (i=0 ;i<5; i++)
```

```
        printf ("\t%d", x[i]);

}
```

Program: 4 Write a program to implement bubble sort

```
#include<stdio.h>
void main ()
{
   int x[5];
   int i, j, tmp;
   // Accepting values for the Array from the User
   for (i=0; i<5; i++)
   {
      printf ("Enter Value:");
      scanf ("%d",&x[i]);
   }
   //Printing unsorted array
   printf ("\n Array Before Sorting:\n");
   for (i=0 ;i<5; i++)
      printf ("\t%d", x[i]);
 //Sorting an array using Bubble sort algorithm
   for (i=0; i<5; i++)
   {
      for (j=0; j<5-i; j++)
      {
         if (x[j] > x[j+1])
         {
            //If Jth element is Greater than J+1th element then swap the value
            tmp = x[j];
            x[j] = x[j+1];
            x[j+1] = tmp;
         }
      }
   }
   //Printing the Sorted Array
   printf ("\n Array After Sorting:\n");
   for (i=0; i<5; i++)
      printf ("\t%d", x[i]);
}
```

Program: 5 Write a program to implement insertion sort

```c
#include<stdio.h>
void main ()
{
   int x[5];
   int i, j, tmp;
   // Accepting values for the Array from the User
   for (i=0; i<5; i++)
   {
     printf ("Enter Value:");
     scanf ("%d", &x[i]);
   }
   //Printing an unsorted array
   printf ("\n Array Before Sorting:\n");
   for (i=0; i<5; i++)
     printf ("\t%d", x[i]);
   /*Sorting an array using Insertion sort (Remember the loop of I starts from 1)
*/
   for (i=1; i<5; i++)
   {
     //Copying x[i] element to variable tmp
     tmp = x[i];
     /* If J is greater than 0 and x[J-1]th element is greater than tmp then bring
J-1th element to Jth position.Remember the loop of J starts from I and J will be
decremented on every iteration. */
     for (j=i; j>0 && x[j-1]>tmp; j--)
        x[j] = x[j-1];
     //place tmp to proper position that is Jth position
     x[j] = tmp;
   }
//Printing Sorted Array
  printf ("\n Array After Sorting:\n");
  for (i=0; i<5; i++)
     printf ("\t%d", x[i]);
}
```

Program: 6 Write a program to implement quick sort

```c
#include <stdio.h>
//Prototype declaration of split and quicksort functions
int split (int*, int, int);
void quicksort (int *, int, int);
//Main function
```

```
void main ( )
{
        int arr[10] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 } ;
        int i ;

        printf ("\nArray before sorting:\n");
        for ( i = 0 ; i <= 9 ; i++ )
                printf ("%d\t", arr[i]);
/*Calling quicksort function and passing base address of the array, start and end
index of the array */
        quicksort (arr, 0, 9);
        printf ("\nArray after sorting:\n");
        for (i = 0; i <= 9; i++)
                printf ("%d\t", arr[i]);
}
/* Function Quick Sort */
void quicksort (int *a, int lower, int upper)
{
        int i;
        if (upper > lower)
        {
                i = split (a, lower, upper);
                quicksort (a, lower, i - 1);
                quicksort (a, i + 1, upper);
        }
}
/*Function Split */
int split (int a [ ], int lower, int upper )
{
        int pivot, p, q, t;
        p = lower + 1;
        q = upper;
        pivot = a[lower];
/* Variable I which is initialized as 0th element is a pivot element. Start loop of P
variable from 1 and loop of Q from 9 */

        while (q >= p )
        {
        //if Pth element is smaller than pivot element then continues the loop
                while (a[p] < pivot)
                        p++;
        //If Qth element is greater than pivot element then continues the loop
                while (a[q] > pivot)
                        q--;
        //When program comes out of both loop swap Pth and Qth elements
                if (q > p)
```

```
                    {
                            t = a[p];
                            a[p] = a[q];
                            a[q] = t;
                    }
            }
        //Finally swap Qth element with pivot element and return the value of Q
        t = a[lower];
        a[lower] = a[q];
        a[q] = t;

        return q;
}
```

Program: 7 Write a program to implement merge sort

```
#include<stdio.h>
// Prototype declaration for function mrg and merge_sort
void mrg (int *, int, int, int);
void merge_sort (int *, int, int);
//Main function
void main ()
{
    int x[10], i, j, k, n;
    printf ("Enter Number of Elements you want to Enter:");
    scanf ("%d", &n);
    for (i=0; i<n; i++)
    {
        printf ("Enter Element for X[%d]:", i);
        scanf ("%d", &x[i]);
    }
    printf ("\nArray Before Sorting:\n");
    for (i=0; i<n; i++)
        printf ("\t%d", x[i]);
    //Sorting an array
    merge_sort (x, 0, n-1);
    printf ("\nArray After Sorting:\n");
    for (i=0; i<n; i++)
        printf ("\t%d", x[i]);
}
void merge_sort (int *x, int beg, int end)
{
    int mid;
    if (beg < end)
```

```
    {
      mid = (beg + end) / 2;
      //After calculating mid split the array recursively
      merge_sort (x, beg, mid);
      merge_sort (x, mid+1, end);
      //After completing split start merge process
      mrg (x, beg, mid, end);
    }
}
void mrg (int *x, int beg, int mid, int end)
{
    int i=beg, j=mid+1, k=beg, temp[10];
    /* Array1 is an array from beg to mid, Array2 is an Array from mid+1 to end
    Compare Ith element of the Array1 to Jth element of Array2 and copy the
    smaller element to the temp array at Kth position. */

    while((i<=mid) && (j<=end))
    {
      if (x[i] < x[j])
      {
        temp[k] = x[i];
        i++;
      }
      else
      {
        temp[k] = x[j];
        j++;
      }
      k++;
    }
    if (i > mid)
    {
      //Copying all remaining elements of Array2 to temp array
      while (j <= end)
      {
        temp[k] = x[j];
        k++;
        j++;
      }
    }
else
  {
      //Copying all remaining elements of Array1 to temp array
      while (i <= mid)
      {
        temp[k] = x[i];
```

```
        i++;
        k++;
    }
  }
  //Copy temp array to original array X
  for (i=beg; i<k; i++)
  {
    x[i] = temp[i];
  }
}
```

Block Summary

All the Units of this Block are extremely important learning as Unit No.1is important due to the huge use of knowledge management, Data ware housing and data mining hence importance of this unit becomes obvious. There is learning related to linear search, the searching begins by matching the values of the array. Initially, the first element is compared with the element to be searched then with the second element and so on. The process continues till the end of the array. That is, it operates by checking every element of a list one at a time in sequence until a match is found. This method of searching for data in an array is straightforward and easy to understand. To find a given item, begin your search at the start of the data collection and continue to look until you have either found the target or exhausted the search space. Further there is learning related to binary search.

Binary Search is Recursive (bin search): This determines that whether the search key lies in the lower or upper half of the array from the stored, by calling itself on the appropriate half. There is also Termination Condition for the search

If low>high then the partition to be searched definitely has no elements in it and If there is a match with the element present at the middle position of the current partition from stored, then the searching can be terminated. There is also learning related to addition of an element (add_data). We have also understood about efficiency Analysis of Binary Search. A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order.

Unit No.2 Gives lot of learning related to the types of sorting which is a method of arranging data elements in a particular order. And, a sorting algorithm is an algorithm that puts elements of a list in a certain order. There are various types of sorting namely Internal Sorting and External Sorting

Internal Sorting: Any sort algorithm which uses main memory exclusively during the sort is an example of Internal Sorting. For Example: Bubble Sort, Insertion Sort

External Sorting: External sorting is an important activity especially in large business. It is thus crucial that it is performed efficiently. External sorting is applied to sort records of files which are too large to fit in the main memory of the computer. This sorting is applied to larger collection of data which rests on.

The Unit No.3 gives great detail of learning about File Organization which has to be implemented as per the organization / company requirement so that data retrieval speed gets minimum so by using efficient method we can store it on disk.

There is understanding related to file and its operations, Operations on database files can be classified into two categories broadly:

Update Operations and Retrieval Operations change the data values by insertion, deletion or update. Retrieval operations (or function) on the other hand do not alter (or change) the data but retrieve them after optional conditional filtering. In both types of operations, selection operation plays significant role. To carry out these operations there are having some important basic operations need to be performed like Open, Locate, Read, Write, Close

Now further we have learnt about a record. A record is a sequence of values or a sequence of characters. There are three kinds of FORTRAN records, such as Formatted, Unformatted, and End file. After this we have continue understanding about a file which is a sequence of records. There are two types of Fortran files as external and Internal

There is also learning related to Hashing which allows us to avoid accessing an index structure. We obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record Bucket is a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block. In mean time we understood about linear hashing.

At the end of this block there is also Linear hashing which can be viewed on as a dynamic version of hashing with separate chaining: the overflows are chained together in a separate overflow area, but the size of the file grows (or shrinks) incrementally, page by page, keeping the file density approximately constant.

Block Assignment

Short Answer Questions

☐   Write a note on internal and external sorting with help of example?

☐   Which sorting method is more efficient?

☐   What is indexed file structure?

☐   Discuss the efficiency of linear and binary search?

☐   What is bucket sort?

Long Answer Questions

J. Explain hashing technique?

K. How to Calculate Hash Function?

L. Write algorithm for Quick Sort?

Bibliography

http://www.studiesinn.com/learn/Programming-Languages/Data-Structure-Theory/Linear-Search.html

http://java2novice.com/java-search-algorithms/binary-search-recursion/

http://www.tutorialspoint.com/dbms/dbms_file_structure.htm

http://scc.qibebt.cas.cn/docs/compiler/intel/11.1/Intel%20Fortran%20Com
piler%2 0for%20Linux/main_for/lref_for/source_files/rfjrec.htm

Enrolment No.

JJ.   How many hours did you need for studying the units?

Unit No	1	2	3	4

No of Hrs

3.   Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific example if any
Presentation Quality	☐	☐	☐	☐	_____
Language and Style	☐	☐	☐	☐	_____
Illustration used (Diagram, tables etc)	☐	☐	☐	☐	_____
Conceptual Clarity	☐	☐	☐	☐	_____
Check your progress Quest	☐	☐	☐	☐	_____
Feed back to CYP Question	☐	☐	☐	☐	_____

1.   Any Other Comments

        ……………………………………………………………………………………

        ……………………………………………………………………………………

        ……………………………………………………………………………………

        ……………………………………………………………………………………

        ……………………………………………………………………………………

        ……………………………………………………………………………………

        ……………………………………………………………………………………

        ……………………………………………………………………………………

        …………………………………………………………………………………

યુનિવર્સિટી ગીત

સ્વાધ્યાયઃ પરમં તપઃ
સ્વાધ્યાયઃ પરમં તપઃ
સ્વાધ્યાયઃ પરમં તપઃ

શિક્ષણ, સંસ્કૃતિ, સદ્ભાવ, દિવ્યબોધનું ધામ
ડૉ. બાબાસાહેબ આંબેડકર ઓપન યુનિવર્સિટી નામ;
સૌને સૌની પાંખ મળે, ને સૌને સૌનું આભ,
દશે દિશામાં સ્મિત વહે હો દશે દિશે શુભ-લાભ.

અભણ રહી અજ્ઞાનના શાને, અંધકારને પીવો ?
કહે બુદ્ધ આંબેડકર કહે, તું થા તારો દીવો;
શારદીય અજવાળા પહોંચ્યાં ગુર્જર ગામે ગામ
ધ્રુવ તારકની જેમ ઝળહળે એકલવ્યની શાન.

સરસ્વતીના મયૂર તમારે ફળિયે આવી ગહેકે
અંધકારને હડસેલીને ઉજાસના ફૂલ મહેંકે;
બંધન નહીં કો સ્થાન સમયના જવું ન ઘરથી દૂર
ઘર આવી મા હરે શારદા દૈન્ય તિમિરના પૂર.

સંસ્કારોની સુગંધ મહેંકે, મન મંદિરને ધામે
સુખની ટપાલ પહોંચે સૌને પોતાને સરનામે;
સમાજ કેરે દરિયે હાંકી શિક્ષણ કેરું વહાણ,
આવો કરીયે આપણ સૌ
ભવ્ય રાષ્ટ્ર નિર્માણ...
દિવ્ય રાષ્ટ્ર નિર્માણ...
ભવ્ય રાષ્ટ્ર નિર્માણ

●