BAOU
Education for All

DR.BABASAHEB AMBEDKAR OPEN UNIVERSITY

DCA

DIPLOMA IN COMPUTER APPLICATION

DCAR-201

Data Structure Using 'C'

DATA STRUCTURE USING C

DR. BABASAHEB AMBEDKAR OPEN UNIVERSITY
AHMEDABAD

Editorial Panel

Authors : Dr. Ketan D. Patel
Associate Professor,
Faculty of Computer Applications
Programme Coordinator (DCS–UG Programme)
Ganpat University

Dr. Kamesh Raval
Assistant Professor,
Somlalit Institute of Computer Application,
Ahmedabad.

Editor : Mr. Parimal Patel
I/C Director,
Khyati School of Computer Application,
Ahmedabad.

Language Editor : Dr. Jagdish Vinayakrao Anerao
Associate Professor,
Smt A. P. Patel Arts &
N. P. Patel Commerce College,
Ahmedabad.

ISBN 978-81-949223-9-1

Edition : 2020

Acknowledgment

Every attempt has been made to trace the copyright holders of material reproduced in this book. Should an infringement have occurred, we apologize for the same and will be pleased to make necessary correction/amendment in future edition of this book.

ROLE OF SELF–INSTRUCTIONAL MATERIAL IN DISTANCE LEARNING

The need to plan effective instruction is imperative for a successful distance teaching repertoire. This is due to the fact that the instructional designer, the tutor, the author (s) and the student are often separated by distance and may never meet in person. This is an increasingly common scenario in distance education instruction. As much as possible, teaching by distance should stimulate the student's intellectual involvement and contain all the necessary learning instructional activities that are capable of guiding the student through the course objectives. Therefore, the course / self–instructional material is completely equipped with everything that the syllabus prescribes.

To ensure effective instruction, a number of instructional design ideas are used and these help students to acquire knowledge, intellectual skills, motor skills and necessary attitudinal changes. In this respect, students' assessment and course evaluation are incorporated in the text.

The nature of instructional activities used in distance education self–instructional materials depends on the domain of learning that they reinforce in the text, that is, the cognitive, psychomotor and affective. These are further interpreted in the acquisition of knowledge, intellectual skills and motor skills. Students may be encouraged to gain, apply and communicate (orally or in writing) the knowledge acquired. Intellectual–skills objectives may be met by designing instructions that make use of students' prior knowledge and experiences in the discourse as the foundation on which newly acquired knowledge is built.

The provision of exercises in the form of assignments, projects and tutorial feedback is necessary. Instructional activities that teach motor skills need to be graphically demonstrated and the correct practices provided during tutorials. Instructional activities for inculcating change in attitude and behaviour should create interest and demonstrate need and benefits gained by adopting the required change. Information on the adoption and procedures for practice of new attitudes may then be introduced.

Teaching and learning at a distance eliminate interactive communication cues, such as pauses, intonation and gestures, associated with the face–to–face method of teaching. This is

particularly so with the exclusive use of print media. Instructional activities built into the instructional repertoire provide this missing interaction between the student and the teacher. Therefore, the use of instructional activities to affect better distance teaching is not optional, but mandatory.

Our team of successful writers and authors has tried to reduce this.

Divide and to bring this Self–Instructional Material as the best teaching and communication tool. Instructional activities are varied in order to assess the different facets of the domains of learning.

Distance education teaching repertoire involves extensive use of self–instructional materials, be they print or otherwise. These materials are designed to achieve certain pre–determined learning outcomes, namely goals and objectives that are contained in an instructional plan. Since the teaching process is affected over a distance, there is need to ensure that students actively participate in their learning by performing specific tasks that help them to understand the relevant concepts. Therefore, a set of exercises is built into the teaching repertoire in order to link what students and tutors do in the framework of the course outline. These could be in the form of students' assignments, a research project or a science practical exercise. Examples of instructional activities in distance education are too numerous to list. Instructional activities, when used in this context, help to motivate students, guide and measure students' performance (continuous assessment)

PREFACE

We have put in lots of hard work to make this book as user–friendly as possible, but we have not sacrificed quality. Experts were involved in preparing the materials. However, concepts are explained in easy language for you. We have included many tables and examples for easy understanding.

We sincerely hope this book will help you in every way you expect.

All the best for your studies from our team!

DATA STRUCTURE USING C

Contents

Unit 5 MORE ON Link–List

Introduction, Types of Link–List, Singly Link–List, Doubly Link–List, Circular Link–List, Doubly Link–List Implementation, Declaring of Node and First Pointer, Creating a Doubly Link–List, Inserting a Value to The Link–List, Displaying Doubly Link–List, Deleting a Node From Doubly Link–List

Unit 6 STACK AND THEIR APPLICATIONS

Introduction, Definitions, Array and Link Representation of Stack, Array Representation of Stack, Link–List Representation of Stack, Operations and Applications of Stack

Unit 7 QUEUES AND THEIR APPLICATIONS

Introduction, Definition, Basic Operations Performed on Queue, Array and Link–List Representation of Queue, Array Representation of Pueue, Link–List Representation of Queue, D–Queue, Circular Queue, Applications of Queue

---

BLOCK 3 : TREE AND GRAPHS

Unit 8 TREES

Introduction, Basic Terminology, Binary Tree, Binary Tree Representation using Array and Link–List, Array (Sequential) Representation, Link–List Representation, Binary Search Tree

Unit 9 OPERATIONS ON BINARY TREE

Introduction, Operations on Binary Search Tree, Binary Tree Traversals, Inorder Traversal, Preorder Traversal, Postorder Traversal, Recursive Algorithms for Inorder, Preorder and Postorder

Unit 10 GRAPHS

Introduction, Definition, Terminology, Types and Representation of a Graph, Graph Traversal, Breadth First Search (BFS), Depth First Search (DFS), Shortest Path Algorithm, Kruskal's Algorithm, Prim's Algorithm

---

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

Unit 11 SEARCHING TECHNIQUES

Introduction, Sequential or Linear and Binary Search, Algorithms for Sequential and Binary Search,

Implementation of Linear Search, Implementation of Binary Search

Unit 12 SORTING TECHNIQUES

Introduction, What is Sorting, Types of Sorting, Internal and External, Bubble, Insertion, Selection, Quick, Merge, Radix Sorting

Unit 13 FILE STRUCTURE

Introduction, File Structure – Concept of Fields, Files and Records, Sequential and Index File Organizations, Hashing Techniques

Unit 14 PROGRAMS OF SEARCHING AND SORTING

Dr. Babasaheb Ambedkar
Open University Ahmedabad

BCAR-201/
DCAR-201

Data Structure Using C

BLOCK 1 : DATA STRUCTURES AND ARRAYS

DATA STRUCTURES AND ARRAYS

Block Introduction :

Data can be defined as a piece of information. It can be of any type including character, integer, date, float etc. Data structure is a method of organizing data in such a way that various operations can be performed effectively. In short, data structure is nothing but a combination of data and operations which can be performed on data.

Block Objectives :

A data structure helps you to understand the relationship of data items with one another; it also gives us an idea of how data is organized in the memory and helps you to understand the concept of abstraction. Based on the need of the operation a user can select different types of data structure.

After the study of this block students are able to :

- Understand what is data structure and why data structure.
- Learn various types of data structures.
- Understand about single dimensional arrays and its representation into memory.
- Know the concept of multidimensional arrays and its representation into memory.
- Understand time and space complexity of an algorithm.

Block Structure :

Unit 1 : Introduction to Data Structures

Unit 2 : Arrays

Unit 3 : Representations of Arrays in Memory

Unit
01

INTRODUCTION TO DATA STRUCTURES

UNIT STRUCTURE

1.0 Learning Objectives :

After learning this unit, you will be able to :

- Understand the definition of Data, Information and Knowledge.
- Understand the concept and need of Data Structures.
- Understand the concept of Private and Composite Data Structures.
- Understand the Time and Space complexity of Algorithms.

1.1 Introduction :

Data Structures are one of the important fields in computer science that refers to different methods of storing and retrieving of data in the computer memory. In this Unit, we will be discussing about the concept of data, information and knowledge, the concept and need of data structures, difference between primitive and composite data structures and time and space complexity of algorithms.

1.2 Data, Information :

Data

1.2.1 Data :

Data is considered as one of the crucial component for any organization because many decisions are taken based on the data. Hence it is essential to know what data and information is. Data is in raw pieces or parts when it is processed as per the requirement of the user it becomes information. Data is the lowest level of abstraction from which meaningful information and knowledge are derived. So if data is wrong then the information or knowledge which is derived from the data would be wrong and ultimately the decisions which are taken based on this knowledge would be wrong. Hence data should be accurately collected and efficiently represented into computer memory for future use. For example, a customer after making his purchase, needs a bill to make the payment. The bill has a variety of data such as item, name, quantity, price, taxes if any and total amount. The data can be in the form of number, character, text etc. All these data are collected and processed as one compiled bill that becomes the information to a customer and the cashier shall now accept the bill amount.

1.2.2 Information :

As discussed in the above paragraph the term information is a collection of data from various corners that is processed in a meaningful form as per the requirement of the user. In simple terms information is a processed data in meaningful form. It has specific meaning. A collection of information in an organized manner which is stored and available to use as required is known as knowledge. The main difference in information and knowledge is in the level of abstraction being considered. Data is the very first level of abstraction followed by information being the next and knowledge being the final and highest level.

Let's understand the concept of knowledge with an example. Following table shows the result of student X in university exam.

BCA–201	BCA–202	BCA–203	BCA–204	BCA–205
75	80	55	91	60

Student X is passing with 72.20%. **His result is 10% more than previous semester.**

Here we have compared the result of Student X with his previous semester result and extract knowledge that his result is 10% more than his previous semester result.

❑ **Check Your Progress – 1 :**

1. What is data ? Explain with an example.
2. What is information ? Explain with an example.

.....

.....

.....

.....

.....

1.3 Data Structures :

We know that the processor can only process the data that is available in the main memory. If the data to be processed is not available in main memory, then it has to be transferred from the secondary memory to the main memory.

In order to represent and store data in the main memory we need an appropriate model to organize data in the main memory. This model is called a data structure. We can say that data structure is special way of organization and management of the data. Data structure provides the functionality to manage the data and information.

1.3.1 Definition :

As we discussed in the previous topic that data is important and hence should be stored in the memory effectively. Data Structure is a way of storing and retrieving the data into computer memory effectively. In simple words it is an organization of data in to computer memory. Data structures provide the way to manage data efficiently as per our requirement. Different Data structures are designed for data arrangement to meet a specific purpose. This arrangement helps to access data efficiently with minimum efforts. For example to represent hierarchical data (parent–child relationship) tree data structure is appropriate. To represent map data, graph data structure is good choice.

1.3.2 Primitive and Non–Primitive Data Type :

(1) **Primitive Data Type :** These are also called fundamental data types. These data types are not composed of other data types. Example : int float, char, double.

(2) **Non–Primitive Data Type :** These are composed of primitive data types. Sometimes it is called user–defined data type, such as, arrays, structures, enumeration, union etc.

1.3.3 Types of Data Structures :

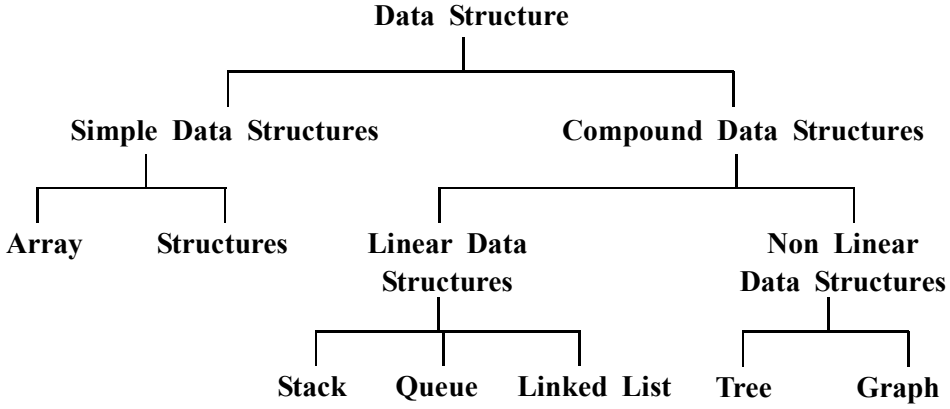
Data structures can be classified into the following two types :

(1) **Simple Data Structures :** Simple Data Structures are built from Basic or Primitive data types. They are built using int, char or real data types. Arrays are example of simple data structures. To form an array we use int, char or real types.

(2) **Compound Data Structures :** These data structure are formed after combining simple data structures in various ways and can be classified as :

(i) Linear Data Structures : All elements in these structures form a sequence and are hence, single level data structures. Data or items are stored in linear fashion. Insertion, deletion and traversal are done in sequential or linear manner. We can access the data in linear way. For example Stack, Queue and Linked List are linear data structures.

(ii) Non–Linear Data Structure : These are multilevel data structures. Data is not represented as well as processed in sequential manner. Elements are stored in non–linear fashion. For example, Tree and Graph data structures.



Types of Data Structures

1.3.4 Data Structure Operations :

The data present in our data structure is processed by means of certain operations. Generally, the particular data structure which is chosen depends largely on the frequency on which specific operations are performed on it. Given below are the major operations which are performed on the data :

(1) Insertion – Addition of a new record to the structure.

(2) Deleting – Deletion of a record from the structure.

(3) Searching – Finding the element with its location from the data structure.

(4) Traversing – Accessing each record exactly once, so that all the items in the structure are visited.

(5) Sorting – Arranging the elements in proper order.

❑ **Check Your Progress – 2 :**

1. What are linear and non–linear data structures ?

.....

.....

.....

.....

.....

1.4 Primitive and Composite Data Structure :

These are also called fundamental data types. These are not composed of other data types, that is, int float, char, double.

Primitive data is a data which is simple, single, and atomic and cannot be decomposed further.

e.g Integer, character, floating point. The examples include roll no. of students or seat no. of the candidate for the examination or bill no. of a customer, percentage of student etc.

Composite data type is derived from the primitive data type, slightly complex than primitive data. It can be similar or of a different type.

e.g Array, union, record. To represent changes in the temperature after every hour in the atmosphere or a change in the velocity speed of a car when it starts to when it is running on the road at various speeds such data types are used.

❑ **Check Your Progress – 3 :**

1. What is a data structure ? Give the types of data structure.

.....

.....

.....

.....

.....

2. What is the difference between a primitive and composite data structure ? Explain with an example.

.....

.....

.....

.....

.....

1.5 Time and Space Complexity of Algorithms :

An algorithm is a well–defined sequence of steps to solve a particular problem. There may be multiple algorithms to solve one problem. Choosing the best algorithm out of these is our topic of interest. The algorithm which required less storage and less execution time is considered as an efficient algorithm.

Efficiency or complexity of an algorithm is related with the number of steps (time complexity) or storage locations (space complexity). Each of the algorithms involves a particular data structure. And we cannot use the data structure of the most efficient algorithm, as it totally depends on the type of data and frequency of data operations. Efficiency of an algorithm is majored in terms of time and space requirements.

1.5.1 Time Complexity :

In simple terms time complexity is majored as time required executing the program. Running time of the program as a function of size of input is known as time complexity. It is the number of operations the algorithm performs to complete its task with respect to the input size. It is important to consider the time complexity while selecting the best algorithms because the algorithm should take less execution time to complete the task.

1.5.2 Space Complexity :

While we are considering the time complexity for choosing the efficient algorithm, it is essential to look at the memory requirements (space) of the algorithm.

During the program execution, the total amount of computer memory required as a function of input size is known as space complexity. These memory requirements can be of direct requirements such as memory required to store variable and arrays or indirect requirements such as memory required for backend process like managing stack for recursion process.

The analysis of algorithms is a major and critical task. There must be some criteria to measure the efficiency of the algorithms. Now, suppose A is an algorithm and n is the size of input data then, the time is measured by counting the number of key operations or comparisons and the space is measured by counting the total amount of memory required by the algorithm during the execution.

The complexity of an algorithm A is the function f(n) which gives the total running time and storage space requirement of the algorithm in the form of input data of size n.

Three cases which are usually investigated in the complexity theory are :

1. **Best Case :** The minimum value of f(n).
2. **Average case :** The expected value of f(n).
3. **Worst Case :** The maximum value of f(n) for any possible input.

Let us understands the concept of best, average and worst case with an example. Suppose you want to search an element form any array using sequential search then the function would be like :

```
Search(int a[5], int x)
{
   int i;
   for(i=0;i<5;i++)
   {
      if(a[i] == x) // it compares the value X with each array element
      return 1;
   }
   return 0;
}
```

Above function search will compare the search element x with each element of an array and if its finds the element then it returns 1 else return 0.

Now consider the time required to find an element from this array. Let's assume that each comparison takes 1 second of time.

Best Case : It is the case which has minimum execution time. Suppose the array elements are 10,20,30,40 and 50 and you want to search 10 then it takes only one comparison to find the element. Hence it takes only 1 second. It is Best Case for this searching algorithm.

Worst Case : It is a case where maximum time is required to perform the task. For example in the above array we want to search 100. Now the value 100 is not available in the array but to confirm that, it performs 5 comparisons. So it takes 5 seconds which is maximum and hence this case is considered as worst case for this searching algorithm.

Average Case : Average case is calculated by finding the average time required to search the element for all possible inputs. If we have n inputs then find the sum of running time for n inputs and divided this sum with n.

❑ **Check Your Progress – 4 :**

1. Define what are complexity and the types of complexity.

.....

.....

.....

.....

.....

2. What is the difference between space and time complexity ?

.....

.....

.....

.....

.....

3. Which of the following data structure is nonlinear data structure ?

(A) Array (B) Queue (C) Tree (D) Stack

4. Find the odd one out.

(A) int (B) char (C) float (D) array

5. Which of the following is the lowest level of abstraction ?

(A) Data (B) Information (C) Knowledge (D) Decision

6. Which of the following criteria's are considered while majoring the efficiency of an algorithm ?

(A) Memory (B) Time

(C) Both A and B (D) None of the above

7. Suppose A is an array having size N where all the elements are sorted in descending order. The time required to sort this array in ascending order with sorting algorithm would be considered as :

(A) Best Case (B) Average Case

(C) Worst Case (D) None of the above.

1.6 Let Us Sum Up :

Data is the lowest level of abstraction from which meaningful information and knowledge are derived. When data is processed and has some meaning it becomes information. Based on this information knowledge is derived and based on these knowledge, decisions are taken. There are mainly two type of data such as Primitive data type and Non–primitive data type.

Meanwhile we have understood about two types of data structures those are Simple data structures and Compound data structures : Simple data structure can be combined in various ways to form compound data structures further has classification having two types as Linear Data structures and Non–Linear Data Structure. If data is represented and operated in linear fashion then it is called linear data structure. Ex. Stack, Queue, Linked List. If data is not stored and operated in linear fashion then it is called nonlinear data structure. Ex. Tree and Graph.

Further we have studied and understood about algorithm which is nothing but list of steps for solving a particular problem. Algorithm has got complexity and hence efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity). Complexity of Algorithms can be majored in terms of Time Complexity and Space Complexity,

Finally we have also understood with an example about three types of analysis for an algorithm those are 1. Worst Case Running Time, 2. Average Case Running Time 3. Best Case Running Time.

1.7 Suggested Answer for Check Your Progress :

- **Check Your Progress 1 :**

  See Sec 1.2

- **Check Your Progress 2 :**

  See Sec 1.3

- **Check Your Progress 3 :**

  See Sec 1.4

- **Check Your Progress 4 :**

  1. See Sec 1.5 **2.** See Sec 1.5 **3.** C **4.** D

  5. A **6.** C **7.** C

1.8 Glossary :

1. **Data Structure –** Data Structure is a way of storing and retrieving the data into computer memory effectively. In simple words it is an organization of data in to computer memory.
2. **Primitive Data Structures –** These are also called fundamental data types. These data types are not composed of any other data types. Example : int, float, char, double.
3. **Compound Data Structures –** Simple data structures can be combined in various ways to form compound data structures.
4. **Algorithm –** It is a sequence of steps to solve a particular problem.
5. **Average Case Running Time –** It is an estimate of the running time of an algorithm for an average input.
6. **Time Complexity –** Running time of the program as a function of size of input is time complexity.
7. **Space Complexity –** During the program execution, the total amount of computer memory required as a function of input size is known as space complexity.

1.9 Assignment :

Gather more information about primitive and non–primitive data types and make a chart by representing them along with their sizes in terms of memory requirements and ranges.

1.10 Activities :

Identify the data required to print a patient bill at the hospital and classify data accordingly.

1.11 Case Study :

Find out which data structures are used in the following area and how they are used ?

1. Recursion
2. Computer File System
3. Computer Networking

1.12 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 Sahni.
4. Data Structures : An Advanced Approach Using 'C' by Esakov and Weises.
5. An Introduction to Data Structures with Applications by Tremblay & Sorenson.

Unit
02

ARRAY

UNIT STRUCTURE

2.0 Learning Objectives :

After learning this unit, you will be able to :

- Understand the concept and need of Arrays.
- Understand the compile and runtime initialization of Arrays.
- Understand the concept of Sparse Arrays.

2.1 Introduction :

Till now you have studied that if you want to perform a certain set of operations on a single number or a group of numbers, you need to store them in variables. But if the numbers to be stored in variables are more, suppose, 50 numbers are there which have to be stored in 50 separate different variables, now, managing 50 separate variables in a single program will make your

program lengthy and complicated. So, in order to reduce the number of variables from a program, the concept of arrays was introduced. This chapter will introduce the concept and characteristics of Arrays, the method of implementation of Arrays and Array representation.

2.2 Characteristics of an Array :

An array is nothing but a collection of elements of the same data type that share a common name. Array cannot have data of mixed data types. ie we cannot store name and marks of students in same array. If we declare array as int (Integer) type then all the elements of that array must be of integer types. Same array cannot have a mixture of different data types as its elements.

Array elements are stored in continuous memory locations. Same size of continuous memory blocks are allocated to array because all the elements of an array are of same data type. One block of memory is occupied by each element of an array. It is important to note down that the size of memory blocks allocated to an array will be depending on the data type of any array.

For example if we declare array of type int with size 5 then total 10 bytes of continuous memory block will be assigned to this array. Where in each 2 bytes of block have one element of an array to be stored. A variable having int data type occupies 2 bytes of memory in C language hence to store 5 int elements 10 bytes of memory is required. Similarly if an array is of float data type with size 5 then 20 bytes will be allocated because float variable requires 4 bytes of memory in C language.

The declaration of an array involves (include) three important information. The data type of an array, name of array and the size of array (maximum number of elements to be stored in array).

Examples –

int marks[100]; // Declaration of An array

marks[9] = 55; // Assigning value 55 to the 10th element of an array.

Here a marks is the name of an array. int is a data type of this array which indicates that this array can store only integer values and 100 indicates the size of an array means we can store 100 integer values in this array.

❖ **Array Characteristics :**

1. Array is a collection of elements of same data type.
2. All the elements of an array share the common name i.e. name of an array.
3. Array elements are stored in continuous memory locations.
4. An array index begins with 0 and ends with index(n–1) where n represents the size of array.

❑ **Check Your Progress – 1 :**

1. Define an array.

.....

.....

.....

.....

.....

2. What are the characteristics of an array ?

.....

.....

.....

.....

.....

2.3 Definition of an Array :

An array is a collection of homogeneous cells in the computer memory. You can also say that it is a group of similar types of data items which are referenced under a single name. Arrays allow you to store a collection of elements of the same data type. For example marks of 100 students, temperature of 10 cities, bills of 100 customers etc.

You can imagine an array in the computer's memory as a row of consecutive memory blocks, each of which can store a single data item, known as an element. Now suppose you want to store the roll nos. and percentage of 50 students, you need to declare 2 separate arrays for that, as the roll nos. will be of integer data type and percentage will be of float data type.

2.3.1 Declaration of Arrays :

Array must be declared first before they are used in the program. The syntax to declare a single dimensional array is :

Data–type array–name [size];

– The Data–type specifies the type of the elements that will be stored in this array.

– Array–name refers the name of the array which will be used to access the elements of this array.

– size indicates the maximum number of elements that can be stored in this array

Example :

int marks[10];

The above statement declares marks to be an array containing 10 integer elements. To access the elements of this array, an index value is used which is called array index. In C language, array index starts with 0 and the last index would be size–1. Here for array marks the valid array indexes are 0 to 9. marks[0] refers the marks of first student. marks[9] refers the marks of 10th student.

To assign the element to this array we can use assignment operator (=). Left side of the assignment operator should be array name with index value and at the right side of the assignment operator should be the array element.

For Example :

marks[0] = 60;

The value 60 is stored in the 0th index of the array marks. By changing the index value we can assign the elements to this array.

2.3.2 Initialization of Arrays :

There are different ways to initialize an array in C language. Array can be initialized at compile time and runtime. When we assign the value to the array at the time of declaration then it is called compile time initialization. When you assign the value to the array at runtime then that initialization is called runtime initialization.

There are different ways to initialize single dimensional array at compile time.

The general form of initialization of arrays is :

Data–type array_name[size]={list of values};

The values in the list are separated by commas, for example the statement

int marks[3]={70,80,90};

Will declare the array of size 3 and will assign 10, 20 and 30 values as the first, second and third element of this array. Index wise storage of these elements would be like :

Array Index	Marks (Array Elements)
↓	↓
marks[0]	70
marks[1]	80
marks[2]	90

Representation of Array Elements Index wise

The following statement initializes all the elements of an array with 0 values.

int a[3] ={0};

If the number of values in the list is less than the size of an array, then only that many elements are initialized. The remaining elements will be set to zero automatically. For example following statement only initializes first 3 elements of an array with values 10, 20 & 30. The values of last two elements will be initialized as 0.

int a[5] = {10,20,30};

Array size may be omitted at the time of the declaration of an array. In such cases the compiler allocates space to such array based on the number of initialized elements. For example, the statement

int a[ ]={10,20,30,40};

will declare the array of size 4. This approach only works when we initialize every element in the array at the time of declaration.

The issues with array declarations are :

– The size must be known in advance.

– Wastage of memory. If we declare array of type integer with size 5 then 10 bytes of memory will be allocated to this array. Now if we store only 3 elements to this array than 6 bytes of memory will be utilized and the remaining 4 bytes will remain unutilized. This is wastage of memory because those 4 bytes of memory will not be assigned to any program.

2.3.3 Accessing Elements of an Array :

Once the array is declared, the individual elements can be referred by their subscripts, that is, the number specified in the bracket with the array name. This subscript specifies the position of the element in the array. For example to access the first element of an array we should write :

```
int a[3] = {10,20,30};
printf("%d",a[0]); // a[0] means the first element.
```

The printf statement will print the first element of any array which is 10.

The valid indexes for the above array are 0, 1 and 2. a[0] represents first element, a[1] represents second and a[2] represents third element.

Given below is a small C program code which will ask the user to enter the values of array for the subsequent positions and keep on storing them at those specific positions. It also access all the elements one by one and print its.

```
void main()
{
int a[5], i;
for(i=0;i<5;i++)
{
   printf ("\nEnter %d element of an array",i+1) :
   scanf ("%d", &a[i]);
}
for(i=0;i<5;i++)
{
   printf ("\n %d ", a[i]);
}
}
```

The first for loop will ask user to enter 5 values of an array and assign these values to the array in the index 0 to 4. Initially the value of i is 0, which will let the value entered by the user to store at 0th position, making it the first element of the array.

In for loop, scanf () statement is used, with the "ampersand" (&) operator. The purpose of using it is to pass the address of the particular array element to the scanf ().

In the second for loop, it access and prints the array elements one by one.

/*Program to count the even and odd numbers from array*/

```
#include<stdio.h>
void main ()
{
int a[10],even=0,odd=0,i;
for(i=0; i<10;i++)
```

```
{
printf("Enter the element %d of the array\n",i+1);
scanf("%d", &a[i]);
}
for(i=0; i<10; i++)
{
if(a[i] % 2 == 0) // it will check that the number is even ?
   even=even+1;
 if (a[i] %2 !=0) // will check for odd number
   odd=odd+1;
}
printf ("\nTotal no. of even nos. are %d", even);
printf ("\nTotal no. of odd nos. are %d", odd);
}
```

The above program performs modulo operation on each element of any array with value 2 and check. If the reminder value is 0 means the number is even else the number is odd. Every time when it finds even or odd number it increase the value of even or odd variables which are here working as a counter variables. At last it prints the value of even and odd variables.

/* Program to find the sum of elements of an arrays*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[5]={1,2,3,4,5},i,sum=0;
   for(i=0;i<=4;i++)
   {
      sum=sum + a[i];
   }
   printf("Sum of the elements of an array = %d", sum);
}
```

2.3.4 Passing Array Elements to a Function :

We can easily pass the array elements to a user defined function by two ways. Call by value and Call by reference. In call by value method, we pass values of array to the function, whereas in call by reference method, the addresses of array elements are passed to the function instead of values. The call by value method is illustrated below :

```
void main()
{
int i=0,a[5];
for(i=0;i<5;i++)
```

```
{
printf("Enter elements\n");
scanf("%d",&a[i]);
}
for(i=0;i<5;i++)
{
display(a[i]);
}
    }
    void display(int v)
    {
    printf("%d",v);
    }
```

In the above program, we are passing an individual array element as value to display() function and printing it in the function display(). Here we are passing single value at a time at every function call and hence in function display() we have declare only single integer variable v. Every time variable v is assigned the value of an array and it is printed using function display. If we change the value of variable v then it will not affect the original array elements because here copy of the element is passed to the function.

The second approach is to pass the address of the element to the function. Let's understand the call by reference concept with following example.

```
void main()
{
int a[5],i=0;
for(i=0;i<5;i++)
{
printf("Enter elements\n");
scanf("%d",&a[i]);
}
for(i=0;i<5;i++)
{
display(&a[i]);
}
void display(int *p)
{
    printf("%d",*p);
}
```

In the above program we are passing the memory address of each of the array element to the function. As we know that in C language pointer variable is used to store the address of the variable. So in the display function we have declared variable p as a pointer variable. * is used to declare pointer

variable. Every time the function display () receives the address of the array element and prints its value using pointer variable p. If you change the value using pointer variable in the function display () then it will affect the original value of array and the array value will be updated with the new value.

❑ **Check Your Progress – 2 :**

1. How can you initialize single dimensional array ?

.....

.....

.....

2.4 Definition of Multidimensional Array :

An array having more than one dimensions are called multidimensional array. Two dimensional arrays are the arrays having two dimensions. If we want to store the marks of ten students for one subject then we can use single dimension array but if you want to store marks of 3 students for six subjects then in that case you need to declare 2D array. 2D array can be thought as a matrix having rows and columns. The elements of 2D array are arranged in rows and columns. To access the elements of 2D array we need two indexes : row index and column index.

2.4.1 Declaration of Two Dimensional Arrays :

Two dimensional arrays are declared as :

Datatype array_name[rowsize][columnsize];

int marks[3][3];

Above array is 2D having row and column size equals 3. We can store marks of 3 students of 3 subjects into this array.

The valid index values for this 2D array are :

marks[0][0], marks[0][1], marks[0][2],

marks[1][0], marks[1][1], marks[1][2],

marks[2][0], marks[2][1], marks[2][2].

Take a note that the row and column indexes are always starts with 0 in C language.

2.4.2 Initializing of Two Dimensional Arrays :

Two dimensional arrays can be initialised at compile time and runtime. If we assign the elements of 2D array at the time of declaration than it is called compile time initialization and if we assign the elements at run time then it is called the run time initialization.

❖ **Compile Time Initialization :**

int marks[3][3]={50,60,70,55,65,75,44,66,77};

The above array will looks like

	Col0	Col1	Col2
Row0	50	60	70
Row1	55	65	75
Row2	44	66	77

marks[0][0]=50 denotes the marks of first student for first subject.

marks[1][2]=75 denotes the marks of second student in third subject.

marks[2][1]=66 denotes the marks of third student in second subject.

❖ **Runtime Initialization :**

In runtime initialization of two dimensional arrays, the values are initialized during the execution of the program. The advantage of runtime initialization is that we can supply different values every time we run the program and ultimately we can test the logic with different sets of inputs.

The following code initializes 2D array runtime.

```
void main()
{
   int marks[3][3], i, j;

   for(i=0;i<3;i++)
   {
      for(j=0;j<3;j++)
         {
            printf("\n Enter the elements of an Array");
            scanf("%d",&a[i][j]);
         }
   }
}
```

Above program will ask to enter array elements to users when they run the program. Users can input different set of elements every time they run the program.

2.4.3 Accessing Elements of Two Dimensional Arrays :

Once the array is declared, the individual elements can be referred by their row and column subscripts, that is, the number specified in the brackets with the array name. This row and column subscripts specifies the position of the element in the array in terms of rows and columns.

Following code in C language is used to read the array elements run time and print it.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int marks[3][3];
   int i,j;
   clrscr();
   // loops to read the value of array from user at runtime.
   for(i=0;i<=2;i++)
```

```
    {
    for(j=0;j<=2;j++)
        {
        printf("enter marks");
        scanf("%d",&marks[i][j]);
        }
    }
// code to access each element of array and print it.
for(i=0;i<=2;i++)
    {
    for(j=0;j<=2;j++)
        {
        printf("%d ",marks[i][j]);
        }
    printf("\n");
    }
}
```

Consider one more example. We want to store marks of 3 students of 3 subjects and need to print total for each student. We can do it with 2D array easily.

```
/ C Program to calculate the total of students marks /
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[3][3],i,j,sum[3]={0};
    clrscr();

    for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
                {
                    printf("\n Enter the marks of student [%d] of subject
                                                        [%d] :->",i+1,j+1);
                    scanf("%d",&a[i][j]);
                }
        }
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
```

```
            {
                sum[i] = sum[i] + a[i][j];
            }
        }
        for(i=0;i<3;i++)
        printf("\n Total of student %d is = %d",i+1,sum[i]);
    }
```

Above program first read the marks of 3 students for 3 subjects. Here we need to calculate the total of each student and store the total marks. Students are 3 so need to have 3 variables. Instead of declaring 3 different variables we have declared a single dimensional array called sum with size 3 which is used to store the total of each student. It stores the total of first student in sum[0], total of second student in sum[1] and total of third student in sum[2]. Finally we have printed the values of sum array.

❑ **Check Your Progress – 3 :**

1. How to initialize 2D array compile time and runtime ?

.....

.....

.....

.....

.....

2.5 Sparse Arrays :

A sparse array is simply an array in which most of its elements are zero. An m x n matrix is called a sparse matrix if most of its elements are zero. The advantage of sparse array is that it requires lesser memory storage compare to normal array because only non–zero elements are going to be stored in the memory. We can also save computational time by logically designing a data structure with an intension to traverse only non–zero elements of the array.

2.5.1 Representation of Sparse Arrays :

If we represent sparse array using two dimensional arrays than it is not efficient storage because we are wasting memory storage by storing zero values. So the effective way should be of storing non zero values with triples {Row, Column, Value}. There are two different ways to represent sparse arrays efficiently. These are :

1. Array Representation
2. Linked List Representation

2.5.2 Array Representation :

2D array is used to represent sparse array (also called sparse matrix) in which we consider only non–zero elements of array along with their row and column indexes. In this type of representation the first row basically stores the information such as total number of rows, columns and total number of non–zero elements in sparse matrix.

Let's understand the array representation using an example. Following table is a 5 x 6 sparse matrix having 6 non–zero values and 24 zero values.

	C0	C1	C2	C3	C4	C5
R0	0	0	0	2	0	0
R1	0	0	1	0	0	5
R2	0	4	0	0	0	0
R3	3	0	0	0	0	0
R4	0	0	0	0	7	0

5 x 6 Sparse Matrix

2D Array Representation of above matrix is as below :

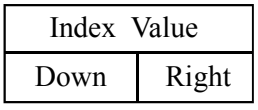
Rows	**Columns**	**Values**
5	6	6
0	3	2
1	2	1
1	5	5
2	1	4
3	0	3
4	4	7

2D Array Representation of Sparse Matrix

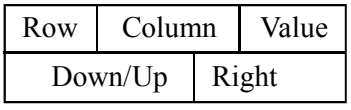
In the above 2D array representation, first row indicates that there are total 5 rows, 6 columns and 6 non–zero elements are there in sparse array. Second and onwards rows represents the row & column indexes of the non–zero elements along with the non–zero values. For example the values 0, 3, 2 indicate that the non–zero element 2 is located in 0th row and 3rd column. By representing only non–zero elements in the memory we are saving storage space and increasing computational time.

2.5.3 Linked List Representation :

In Linked List representation of Sparse Matrix, Linked List is used to represent the data. In linked list representation two kinds of node structures are used : Header Node and Element Node. Header nodes are used for rows and columns identification and element node represents the data. The node structures of header and element nodes are as below :



Header Node

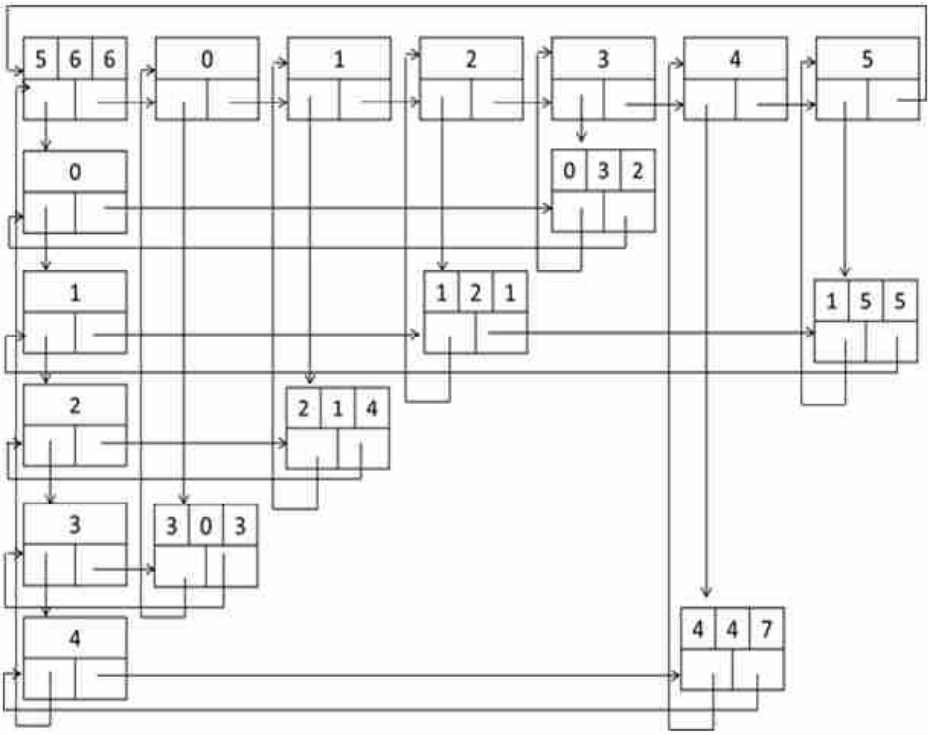


Element Node

In header node index value represents either row or column number. Down and Right are the pointer which are used to store the addresses of adjacent nodes in rows and columns.

In element node row represents the row number of non–zero element, Column represents column number of non–zero element and value is used to store non–zero value. Down/Up is the pointers which are used to store the

address of the next element node in the same column. Similarly Right is the pointer which is used to store the address on the next element node in the same row. Following figure shows the linked list representation of the sparse matrix we have taken as an example.



Link–List Representation of Sparse Matrix

In the above figure nodes with values only as 0, 1, 2, 3, 4, 5 are header nodes showing corresponding row or column number. All other nodes having five fields are element nodes which are used to represent non–zero elements of the matrix. The first node in the upper left corner having values as 5, 6, 6 is used to represent the abstract information of the sparse matrix. We can see that all the nodes are connected with each other in circular fashion so accessing of any non–zero element is very easy.

❑ **Check Your Progress – 4 :**

1. Define sparse array.

   .....

   .....

   .....

2. Explain linked list representation of sparse array with example

   .....

   .....

   .....

3. Which of the following is not true for array in C language ?

   (A) It must be declared before to use it.

   (B) It can be initialised at compile time and run time.

   (C) Size must be known in advance.

   (D) We can store elements of different data types.

4. Which of the following is correct way to initialize array in C language ?

   (A) int a[5]; (B) int a[5]={1,2,3};

   (C) int a[]={1,2,3}; (D) All of above.

5. The array where most of its elements are zero is called ?

   (A) Sparse Array (B) Multi–Dimensional Array

   (C) Static (D) Dynamic array

6. What will be the output of following code ?

```
int a[5]={1,2,3,4};
printf("%d", a[4]);
```

   (A) 4 (B) Error (C) 0 (D) Garbage Value

7. Which of the following is not the valid index for the following array ?

```
float temperature[3][4];
```

   (A) temperature[0][3] (B) temperature[0][4]

   (C) temperature[1][0] (D) temperature[2][2]

2.6 Let Us Sum Up :

In this unit we have understood the concept of array which is a collection of homogeneous cells in computers memory. You can also say that array is a collection of elements of same type which are referenced under a single common name. Arrays allow us to store a collection of elements of the same data type. You can imagine an array in the computer's memory as a row of consecutive memory blocks, each of which can store a single data item of same type, known as an element. suppose you want to store roll no. and percentages of 50 students, you need to declare 2 separate arrays for that, as the roll nos. will be of integer data type and percentages will be of float data type. Like variables, arrays are also must be declared before they are used in the program. The syntax for single dimensional array declaration is :

Data type array-name [size];

Further we have understood that there exists type of array like One or Single dimension array and two dimensional arrays or multidimensional arrays. In two-dimensional array - Unlike single dimensional array, has two indexes or subscripts. One subscript denotes the row number & the other denotes the column number.

The two dimensional arrays are declared as follows:

data_type array_name[row_size][column_size];

int marks[3][3]; // array to store the marks of 3 students of 3 subjects.

float temperature[10][12]; // array to store the temperature of 10 cities for 12 months.

We have also understood a kind of array called Sparse array. A sparse array is simply an array in which most of its elements are zero. An m x n matrix is called sparse matrix if most of its elements are zero. There are two ways to represent sparse arrays efficiently those are 1) Array Representation and 2) Linked list representation.

2.7 Suggested Answers for Check Your Progress :

- **Check Your Progress 1 :**

  See Sec 2.1 & 2.2

- **Check Your Progress 2 :**

  See Sec 2.3.

- **Check Your Progress 3 :**

  See Sec 2.4

- **Check Your Progress 4 :**

  1. See Sec 2.5 **2.** See Sec 2.5 **3.** D **4.** D

  5. A **6.** C **7.** B

2.8 Glossary :

1. **Array –** An array is a collection of homogeneous cells in the computer's memory.
2. **Sparse Array –** A sparse array is simply an array in which most of its entries are zero. A m x n matrix is called as sparse matrix if most of its elements are zero.
3. **Single Dimensional Array –** Single dimensional array or one dimensional array requires only one index to access an element.
4. **Multi–Dimensional Array –** Multi–dimensional array can be represented using a matrix which consists of r rows and c columns.

2.9 Assignment :

Suppose Arr is a linear single dimensional array with n numeric values. Write a program to find the average AVG of the values in Arr. The arithmetic mean or average of the values is defined by

AVG = (x1 + x2 + x3 + ... xn) / n

2.10 Activities :

Write a C program to understand the difference between call by value and call by reference. What will happen if we pass array to function as an argument and then change its values in the function ? Will it affect the original array ? Check practically.

2.11 Case Study :

Take one m × n matrix and Perform following operations on it :

1. Addition of Two Matrices
2. Multiplication of Two Matrices. Find out what are the prerequisites for performing multiplication of two matrices.
3. Transpose of the Matrix

2.12 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 Sahni.
4. Data Structures : An Advanced Approach Using 'C' by Esakov and Weises.
5. An Introduction to Data Structures with Applications by Tremblay and Sorenson.

Unit 03

REPRESENTATIONS OF ARRAYS IN MEMORY

UNIT STRUCTURE

3.0 Learning Objectives :

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

- How one dimensional arrays are represented in memory
- Row and Column Major order Representation of 2D Array
- How memory addresses are computed for array elements.

3.1 Introduction :

In previous units we have learned about single and multidimensional arrays. Now it is important to understand how array elements are stored into main memory and how these elements are accessed. In this chapter we will learn the storage representation of Single and Two dimensional arrays. We will also learn to calculate the memory address of an array element based on the given base address.

3.2 Representations of One Dimensional Array in Memory :

As we know that whenever we declare a variable or an array, it is stored in the main memory. If it is a simple int variable then in C language, two bytes of empty memory block is assigned to this variable to store the value of variable. Similarly if we want to store the values of arrays then multiple memory address are assigned because in array we are going to store multiple values. One Dimensional or Single Dimensional Arrays are stored in continuous block of same size memory addresses. Arrays elements are always stored in continue memory addresses.

When one dimensional array is declared in the program, compiler first calculate the memory requirement of that array based on the number of elements and then search in the memory for the availability of required memory block and based on that it assigns that free memory block to the array.

Let's understand the memory representations of One Dimensional Array with an Example.

Suppose we want to store the marks of 5 students. Now these marks are of integer types so we need to declare a one dimensional array with int type like :

```
int marks[5]={40,84,67,55,75};
```

We have declared an array called marks and assign marks of 5 students into it. As we know that these five elements are accessed using array index as marks[0],marks[1] and so on. These marks are represented as :

Array Index	Marks (Array Elements)
↓	↓
marks[0]	40
marks[1]	84
marks[2]	67
marks[3]	55
marks[4]	75

The marks of first student is stored in marks[0]. The marks of second student is stored in marks[1] and so on. When this array is declared, program calculates the total memory to store all the elements of this array. In C language, 2 bytes are required to store single integer value. Here in our example we are going to store 5 integer values (marks) so the total memory requirements for this array is 2*5 ie 10 bytes. So compiler will find the free block of 10 continuous bytes memory and assign that block to this array. The Storage representation will looks like :

Array Index	Marks	Memory Addresses	
↓	↓	↓	
marks[0]	40	2000	← Base Address
marks[1]	84	2002	
marks[2]	67	2004	
marks[3]	55	2006	
marks[4]	75	2008	

The array marks has assigned the block of memory starting from memory address 2000 to 2009. These memory addresses are nothing but the number which is assigned to each byte in a computer's memory that is used by CPU to track where data and instructions are stored in RAM. These numbers represents the single byte of memory storage. So these 10 bytes of memory block is assigned to array marks to store its 5 elements. These elements are stored as :

- 40 is stored at memory address 2000 & 2001 (2 Bytes due to integer values)

- 84 is stored at memory address 2002 & 2003
- 67 is stored at memory address 2004 & 2005
- 55 is stored at memory address 2006 & 2007
- 75 is stored at memory address 2008 & 2009

Address of the first element of an array is called as base address, using this base address we can calculate the subsequent address of an array. If you know the base address then you can easily calculate the memory address of any element of that array because we know that array elements are stored in continuous memory addresses.

3.3 Address Calculation for One Dimensional Array :

In previous topic we have learned that how single dimensional arrays are stored in memory. Now let's take some examples to calculate the memory addresses of array elements.

Address calculation in a single dimensional array :

Example, Suppose that int marks[10] is declared and its first element is stored at address 1000. Calculate the address of 4th element of an array, index of array starts from 0.

To calculate the address of any elements of an array, following details are required :

- Base address or the address of the first element of an array
- Array index of the elements for which we want to find memory address
- Width (Number of bytes required to store individual array element).
  - In this example Base address is 1000.
  - Array index of the 4th element is 3(Because array index starts with 0 means first elements is stored in marks[0] and so on).
  - Width is 2 bytes. As this is integer array so 2 bytes are required to store each array element.

The equation to calculate the memory address is :

Base address + width * index

So putting all this values we have

Address of 4th element = 1000 + 2 * 3 which is equal to 1000 + 6 equals **1006**.

So the 4th element is stored in 1006 memory location.

Example 2 : Suppose the array is int A[5]={10,20,30,40,50}; and base address is 2000 then at what address the element 30 is stored in memory ?

Here base address = 2000, width=2 bytes and the index of 30 is 2. Hence

Address of 30 is = 2000 + 2 * 2 which is equal to **2004**.

So the element 30 is stored at 2004 memory location.

Similarly if the array is storing character values then the width will be 1 because in C language characters are stored in single byte of memory.

If array is storing any floating values like percentage of students or temperatures of different cities then in that case width will be 4 because float values require 4 bytes of memory.

C program to display array elements with its memory address.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[5],i;
   clrscr();
   for(i=0;i<5;i++)
   {
   printf("\nEnter the %d element of Array",i+1);
   scanf("%d",&a[i]);
   }

   for(i=0;i<5;i++)
   {
   printf("\n Array Element %d is stored at %u location", a[i],&a[i]);
   }
}
```

Above program receives 5 integer elements from user, store them in array and then print each element with its memory address. %u format specifier in printf is used to display the memory address of the elements of an array.

The output is as :

Array Element 10 is stored at location 5000

Array Element 20 is stored at location 5002

Array Element 30 is stored at location 5004

Array Element 40 is stored at location 5006

Array Element 50 is stored at location 5008

❑ **Check Your Progress – 1 :**

1. What is the equation to calculate the memory address of the element of single dimensional array ?

.....

.....

.....

.....

2. If the given array is float percentage [10]; and base address is 3000 then at which position the 5th element is stored in memory ?

.....

.....

.....

.....

3.4 Representations of Two Dimensional Arrays in Memory :

Like one dimensional array, elements of two dimensional arrays are also stored in continuous memory locations. It can be represented in memory using two ways :

- Row–major Order
- Column–major Order

In computing, row–major order and column–major order representations describe the methods of storing multidimensional arrays in linear memory locations.

3.4.1 Row Major Order Representation :

Two Dimensional arrays are represented using rows and columns. Both rows and column indexes are used to access the element of two dimensional arrays. The elements of 2D arrays are stored into memory using Row or Column Major Order. In Row major Ordering, all the rows of 2D arrays are stored into memory continuously. First it stores the elements of first row into memory in continuous locations, after that the elements of second rows are stored and so on. This Row wise representation is called Row major Order Representation.

Let's understands the row major order representation using an example. Suppose we want to store the marks of 3 students for 3 subjects. Here we need to store two kinds of information, 3 students and 3 subjects, and hence to store these values we need to declare a two dimensional array like :

```
int marks[3][3]={50,60,70,55,65,75,44,66,77};
```

Here the row size 3 indicates the 3 students and column size 3 indicates 3 subjects. 50 represent the marks of first student in first subject. 75 represent the marks of second student in third subject and so on. In table structure it looks as :

	Col0	Col1	Col2
Row0	50	60	70
Row1	55	65	75
Row2	44	66	77

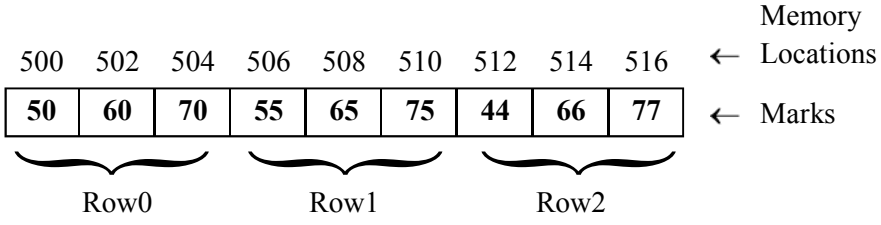
The elements are stored as :

marks[0][0] = 50 marks[0][1] = 60 marks[0][2] = 70

marks[1][0] = 55 marks[1][1] = 65 marks[1][2] = 75

marks[2][0] = 44 marks[2][1] = 66 marks[2][2] = 77

The Row major order representation of this array is as follows :



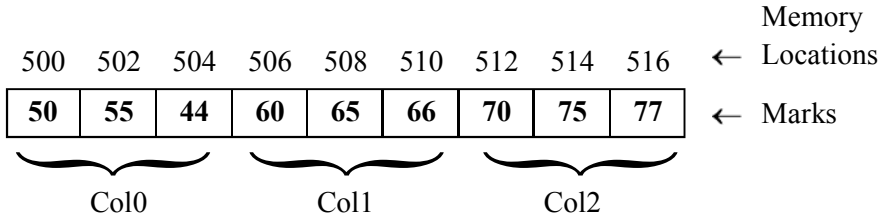
Row major order representations of 2D array

In the above representations we can see that elements of array are stored in continuous memory locations starting from 500 to 516. Here memory address 500 is considered as base address. Rows are stored in memory one after another in continuous memory locations.

3.4.2 Column Major Order Representation :

In column major order representations, the elements of two dimensional arrays are stored as column wise. First it stores the elements of first column into memory in continuous locations, after that the elements of second column are stored and so on. This column wise representation is called column major order Representation.

The Column major order representation of the above array is as follows :



Column major order representations of 2D array

In the above representations we can see that elements of array are stored in continuous memory locations starting from 500 to 516 according to column order. Here memory address 500 is considered as base address. Columns are stored in memory one after another in continuous memory locations.

3.5 Address Calculation for Two Dimensional Array :

In previous topic we have learned that how two dimensional arrays are stored in memory in row and column major order. Now let's take some examples to calculate the memory addresses of array elements in row and column order representations

Address calculation in Row major order :

Example, we have array int marks[3][3]={50,60,70,55,65,75,44,66,77};

The element 55 is stored at which location in row major order representations if base address is 500 ?

The equation to calculate the address of the element in row major order representation is,

Loc(array[i][j])=base(array)+w(n*i+j)

Where i and j represents the ith row and jth column and n is number of columns in array, w=width.

Here we need to calculates the address of the element 55 which is stored at marks[1][0] that is second row and first column hence i=1 and j=0.

Base Address = 500, n=3 (Number of columns) and w=2 (Integer elements require 2 bytes of memory).

Loc(marks[1][0]) = 500+2(3*1+0) = **506**

So the element 55 is stored at 506 memory location in row major order representation. We can confirm the same in figure available in the block 3.4.1

Address calculation in Column major order :

In previous block we learn to calculate the memory address of the element in row major order representation. Now let's take one more example to calculate the address of the element in column major order representation.

Example, we have array int marks[3][3]={50,60,70,55,65,75,44,66,77};

The element 55 is stored at which location in column major order representations if base address is 500 ?

The equation to calculate the address of the element in column major order representation is,

Equation : **Loc(array[i][j])=base(array)+w(m*j+i)**

Where i and j represents the ith row and jth column and m is number of rows in array, w=width

Here we need to calculates the address of the element 55 which is stored at marks[1][0] that is second row and first column hence i=1 and j=0.

Base Address = 500, m=3 (Number of rows) and w=2 (Integer elements require 2 bytes of memory).

Loc(marks[1][0]) = 500+2(3*0+1) = **502**

So the element 55 is stored at 502 memory location is column major order representation. We can confirm it from the figure available in the block 3.4.2

❑ **Check Your Progress – 2 :**

1. What are the equations to calculate the memory address of the element of two dimensional arrays in row and column major order representations ?

   .....

   .....

   .....

   .....

2. Suppose marks[2][2]={10,20,30,40}; is a 2D array then calculate the memory address of the element 30 in row and column major order representation if base address is 5000.

   .....

   .....

   .....

   .....

3. How many bytes of memory would be assigned to the following array in C language ?

   int marks[3][4];

   (A) 14 (B) 12 (C) 24 (D) 48

4. At which position the element 5 is stored in memory for following array where the bases address is 4000 ?

   int arra[5]={1,2,3,4,5};

   (A) 4000 (B) 4004 (C) 4005 (D) 4008

5. How will the following array be represented in memory in column major order representation ?

   int marks[2][3] = {40,50,60,70,80,90};

   (A) 40,50,60,70,80,90 (B) 40,70,50,80,60,90

   (C) 40,60,80,50,70,90 (D) None of the above.

6. What is the right equation to find memory address of an element in row major order representation for m × n matrix (m rows and n columns)

   (A) Loc(array[i][j])=base(array)+w(n*i+j)

   (B) Loc(array[i][j])=base(array)+w(m*i+j)

   (C) Loc(array[i][j])=base(array)+w(n*j+i)

   (D) Loc(array[i][j])=base(array)+w(m*i+j)

7. How many bytes of memory would be wastage for the following array if we assign only 7 elements to this array ?

   float percentage[10];

   (A) 28 (B) 6 (C) 12 (D) 16

3.6 Let Us Sum Up :

This unit has the logical connection with unit 2. In this unit we have learnt how single and two dimensional arrays are represented in the memory. We have seen that array elements are stored in continuous memory locations. If we know the address of the first element of a one dimensional array than we can calculate the address of any element of array using the equation

Base address + width * index

There are two methods to represent the two dimensional arrays in the memory : Row and Column major order representations. In row major order representations, array elements are stored in memory row wise. In column major order representations the array elements are stored as column wise in memory. The equations to calculate the memory address of row and column major order representations are as follows respectively :

Loc(array[i][j])=base(array)+w(n*i+j)

Loc(array[i][j])=base(array)+w(m*j+i)

With these equations we can easily calculate the memory address of 2D array elements.

3.7 Suggested Answer for Check Your Progress :

❑ **Check Your Progress 1 :**

1. Base address + width * index **2.** 3016

❑ **Check Your Progress 2 :**

1. Loc(array[i][j])=base(array)+w(n*i+j)

Loc(array[i][j])=base(array)+w(m*j+i)

2. 5004 in Row major order representations

5002 in Column major order representations

3. C **4.** D **5.** B **6.** A **7.** C

3.8 Glossary :

1. **Row major order –** When the elements of the 2 Dimensional arrays are stored row wise continuously in the memory then that representation is called Row major order representation.
2. **Column major order –** When the elements of the 2 Dimensional arrays are stored column wise continuously in the memory then that representation is called Column major order representation.

3.9 Assignment :

Store the marks of 10 students of 10 subjects in 2D array. Calculate the memory addresses of subject wise highest marks entries.

3.10 Activities :

- Prepare the list of programming languages that support the row and column major order representations.
- Write a C program to display the memory addresses of the 2D array elements.

3.11 Case Study :

Do performance analysis of row and column major order representations with an example. Find out which one is a better representation in terms of performance.

3.12 Further Readings :

1. Data Structures Using "C" by Tanenbaum.
2. An Introduction to Data Structures with Applications by Tremblay and Sorenson.

BLOCK SUMMARY :

In the First Unit collectively what we have studied can be said in this manner. Data is important for any organization. Data is in raw pieces or parts when it is processed and as per the requirement of the user it becomes information. Data is the lowest level of abstraction from which meaningful information and knowledge are derived. There are mainly two type of data such as Primitive data type and Non–primitive data type

Meanwhile we have understood about two types of data structures those are Simple data structures and Compound data structures : Simple data structure can be combined in various ways to form compound data structures further has classification having two types as Linear Data structures and Non–Linear Data Structure.

Further we have studied and understood about algorithm which is nothing but sequence of steps to solve a particular problem. Algorithms have complexity and hence efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity). We have also connected to type of Complexity of Algorithms. This is referred to Time Complexity and Space Complexity,

Finally we have also understood that there are three types of analysis for an algorithm 1.Worst Case Running Time, 2.Average Case Running Time 3. Best Case Running Time.

In Unit No. 2 we have understood about an array which is a collection of homogeneous cells in computers memory. You can also say that array is a collection of elements of same type which are referenced under a single common name. Arrays allow us to store a collection of elements of the same data type. You can imagine an array in the computer's memory as a row of consecutive memory blocks, each of which can store a single data item of same type, known as an element. Now suppose you want to store roll no. and marks of 50 students, you need to declare 2 separate arrays for that, as the roll nos. will be of integer data type and marks will be of float data type. Like variables, arrays are also must be declared before they are used in the program. The syntax for single dimensional array declaration is : Data type array–name [size]; Further we have understood that there exists type of array like One or Single dimension array and two dimensional arrays.

Two–dimensional array – Unlike single dimensional array, here the array has two indexes or subscripts. One subscript denotes the row number & the other denotes the column number. The two dimensional arrays are declared as follows :

data_type array_name [row_size][column_size];

int marks[3][3]; // array to store the marks of 3 students of 3 subjects.

float temperature[10][12]; // array to store the temperature of 10 cities for 12 months.

Sparse array – A sparse array is simply an array in which most of its elements are zero. An m x n matrix is called sparse matrix if most of its elements are zero. There are two ways to represent sparse arrays efficiently those are 1) Array Representation and 2) Linked list representation.

Row major order and Column major order Implementation of 2D

In computing, row–major order and column–major order representations are the methods of storing multidimensional arrays in linear memory blocks. In row major order representations, array elements are stored in memory row wise. In column major order representations the array elements are stored as column wise in memory. With the help of equations we can easily find the memory address of the elements of single or two dimensional arrays.

We have studied and understood All the Units of this Block with the help of various relevant examples.

BLOCK ASSIGNMENT :

❖ **Short Questions :**

1. What is a data structure ?
2. What are the types of data structure ?
3. Define complexity.
4. What are the types of complexity ?
5. Define arrays and types of arrays.
6. What is a sparse array ?
7. What is the equation to find the memory address of the element of one dimensional array ?

❖ **Long Questions :**

1. How arrays are initialized compile time and runtime ?
2. How to represent single dimensional and multidimensional array in the memory ?
3. How to calculate the memory address of single and two dimensional array elements ?
4. What do you mean by linked list representation of sparse matrix ? Explain with an example.
5. Explain time and space complexity with example.

❖ **Enrolment No. :** ☐

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

Unit No.	1	2	3
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 Ahmedabad

BCAR-201/
DCAR-201

Data Structure Using C

BLOCK 2 : STACK, QUEUES AND LINK–LIST

STACK, QUEUES AND LINK–LIST

Block Introduction :

We may need data structures which can be able to store the data in non–contiguous manner into the memory. As we know arrays are storing homogeneous collection of the data in contiguous manner in the memory. Similarly, Link–List stores multiple data in non–contiguous manner.

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 Objectives :

After learning this Block, you will be able to :

- Implement the Link–List
- Perform different operations on Link–List
- Understand the different types of Link–List
- 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 Link–List representation of queue
- Understand D–Queue
- Understand Circular queue
- Know application of queue Understand the concept of Link–List

Block Structure :

Unit 4 : Link–List

Unit 5 : More On Link–Lists

Unit 6 : Stacks And Their Applications

Unit 7 : Queues And Their Applications

Unit
04

LINK–LIST

UNIT STRUCTURE

4.0 Learning Objectives :

In this unit, we will discuss about the dynamic memory allocation and Link–List :

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

- The concept of the Link–List.
- Implementation of the Link–List.
- Perform different operations on Link–List.
- Understand the different types of Link–List.
- Understand the applications of Link–List.

4.1 Introduction :

As we know that the array implementation, can store multiple data elements of the same type under one name. Usually, we are declaring array using static method like int arr [10]. The problem with this approach is, you should know the size of an array well in advance. Dynamic approach, allows you to create the array dynamically at run time. So, even if, the size of the array is not known you can take the size from the user at runtime and create the array. To do this you need to learn, various functions of dynamic memory allocation, available in the C–Language. So, in this unit we will discuss all the functions of dynamic memory allocation and then we will see how can we implement Link–List implementation.

4.2 Dynamic Memory Allocation Functions :

Variables can be declared in two ways : [1] Static declaration and [2] Dynamic declaration. In the static declaration, you need to declare the variables in the declaration section of any function. C–Language does not allow you to declare the variable after any executable statement. When you run the program, compliers of the C–Language will read all declarative statements and make arrangement to occupy space for all of your variables into the main memory of the system. Once the memory space is allotted to all variables then program will start its execution. That means that, when program start its execution, before that it reserves the space for all variables declared in the function.

Dynamic memory allocation on the other hand, allows you to declare the variable whenever it is needed. It is not mandatory that you need to declare all variables first and then you need to write executable statements. Dynamic memory allocation, start the program first and when it is needed variables will be declared at runtime (not before start of execution of the program). The following functions need to be used, to declare the variables dynamically.

4.2.1 Malloc () Function :

The function malloc() allows you to create one variable (of any type) dynamically. To use the malloc() function you need to include header file "alloc.h" if you are using turbo C or Borland C. If you are using CodeBlocks the you need to include "stdlib.h" header file.

Syntax :

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

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

Int_pointer = (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 creates the variable at runtime; therefore, they don't have variable name. So, malloc function always returns the address of the memory space where it is created. In the above example, we are instructing to the malloc function to occupy 'sizeof(int)' means 2 bytes of space in the memory. The function malloc (), will find free space in the main memory and reserves 2 bytes of space. The address of this memory space will be return by malloc () function, which we are storing in the pointer variable 'p' (as we know, pointer are the variables which stores the address of another variables). Now, using the pointer variable 'p', we can store value 5 and print it on the console screen. If malloc () function fails to occupies memory space due to some reason, instead of memory address it returns NULL value to the pointer variable.

4.2.2 Calloc () Function :

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

Syntax :

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

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

Int_pointer = (int *) calloc (10, sizeof (int));

The following program guide you to declare an array of integer variables dynamically using calloc () function :

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

```
    printf("You have Entered :");
    for (i=0; i<n; i++)
    printf("\n%d", p[i]);
}
```

The following program allow user to create the array as per user's need. If user want to create the array of 5 elements, then user will enter 5 while the program will prompt for "Enter Number of Elements :". The function calloc() will now create the array of 5 elements of type int. Here, calloc () function allow user to create the array as per user's need, dynamically (at runtime). We take 5 integer values from the user and store it into the dynamically created array using for loop, and finally we are printing all 5 elements on the console screen using another for loop.

4.2.3 Free () Function :

The function free () is used to delete the variable created by malloc () function. Consider the following program, in which we are creating an integer variable dynamically using malloc () function. We store the value in it, and finally after printing the value of dynamic variable on the console, we delete it using free () function. Make sure, once the variable is deleted using free () function, cannot be used in the program.

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 in Dynamic variable is %d", *p);
    free(p); // Dynamic variable is disposed here
}
```

❑ **Check Your Progress – 1 :**

1. _____ dynamic memory allocation function is used to create one variable dynamically.

   [A] calloc () [B] malloc ()

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

2. _____ dynamic memory allocation function is used to declare an array dynamically.

   [A] calloc () [B] malloc ()

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

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

   [A] clear () [B] delete ()

   [C] free () [D] All of the above

4.3 Link–List :

We know that the array is a collection of same type of data. Similarly, Link–List are also collection of data. The difference between the Array and Link–List is, Array stores all data element sequentially (contiguous manner), where Link–List stores all data elements randomly (non–contiguous manner). For Example, if you declare array 'int x [10]', then program needs 20 bytes of continuous space in the memory. Suppose, if the memory has 20 Bytes of space but not continuously, then array will not be declared and system will throw 'Memory Overflow' error. But in the case of Link–List, you can store your data if memory has sufficient space to accommodate all data elements either in contiguous or non–contiguous manner.

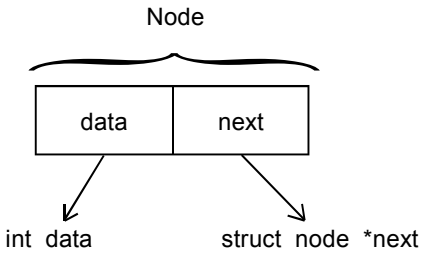
A Link–List is a collection of data elements called nodes, and these nodes are divided into two parts–the data part and the address part. The data part of the node contains the data to be stored in the list and the address part of the node contains the address of the next node.

You can analyse the same with the help of the given example. Suppose, a teacher takes her class students for a movie show and the total no. of students in the class are 30. Now, the seats allotted to the students in the movie hall are scattered, that is, are not contiguous, then it will be a difficult job for the teacher to collect the students from the movie hall after the end of the show. Now, what the teacher will do is she will keep a paper note with her in which the seat no. (address) of the first student is written and she will also distribute a paper note to each student in which the seat no. (address) of the next student is written. Now, at the end of the show, the teacher will go to the first student with her paper note on which his address is written and collect him, and will take the note from that student to find the seat no. (address) of the next student. Similarly, she'll go to each student and do the same, in order to collect all the students. Therefore, as the seats of each student is scattered and each student is having the seat no. (address) of the next node, so it will be an easier job to locate the other students.

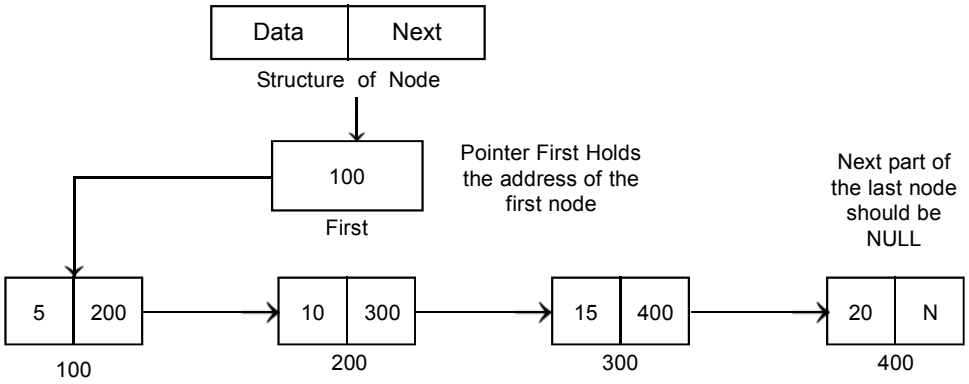
Similarly, in Link–List, all the nodes are not in contiguous order in the memory, they are scattered and each node keeps a track of the next node by storing its address with itself making it easy to traverse the entire link–list.

4.3.1 Node Structure :

Node is a data structure, which consists of two elements. First element is data in which the data given by the user is stored. Second element stores the address of the next node. Consider the following figure of node representation. In the figure node represents two part. Suppose if we consider, user wants to store some integer numbers then in the first part of the structure we declare 'int data' and in the second part we store the address of the next node, therefore the second variable in the node structure will 'struct node *next'.



4.3.2 Link–List Representations :



As shown in the figure given above, 'First' variable is a pointer variable which stores the address of the first node. 'First' pointer has value 100, which is the assumed address of the first node. First node, has data 5 and address 200, which is the assumed address of the second node. The Link–List has 4 nodes. The Last node has value 20 and because it is last node, NULL value is there in its next (address) part. In the given figure 100, 200, 300 and 400 are the assumed address of 4 nodes inserted into 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

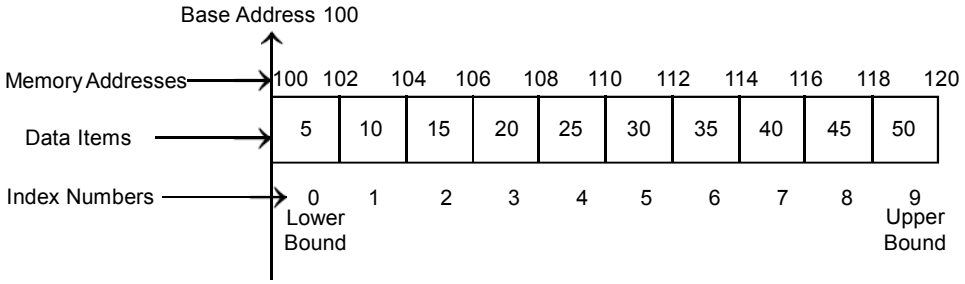
4.3.3 Defining Structure Node :

As we have discussed that a node is a basic building block of Link–List data structure. Structure node consists of two parts : [1] Data : where we will store the user's data and [2] Next : where we will store the address of the next node. Because of we are storing the address in the Next part, it must be declared as pointer. In short, we can define structure node as shown below :

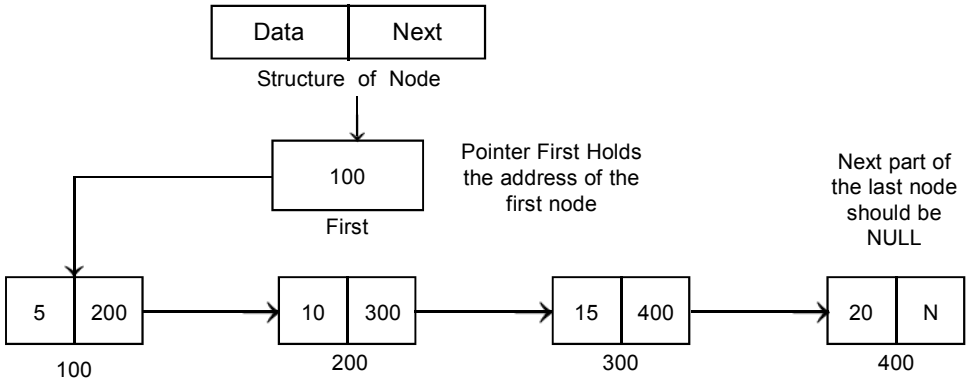
```
struct node
{
    int data;
    struct node *next;
};
```

Each of these expressions must be either an integer constant or a character constant. Each statement following the case labels may be either simple or compound. Compound statement does not require {} in switch statement. While executing switch statement, the expression is evaluated and control is transferred directly to the group of statements whose case labels value matches the value of the expression. If the case–label value does not match with the value of the expression, then none of the groups within the switch statement will be selected. In this case, the control is transferred directly to the statement that follows the switch statement.

4.3.4 Difference in Array and Link–List Data Structures :



In the figure give above, we have represented 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 contiguous manner in the memory. When you declare 'int x [10];', then system search for 10 (number of elements) * 2 (each int data elements needs 2 Bytes) = 20 Bytes of contiguous memory space into the main memory. Suppose, system gets contiguous 20 Bytes of space at some address 100, then address 100 will be a base address of that array. First element of an array, 5 will be stored between memory locations 100 to 102. Second data element 10, will be stored from 102 to 104 and so on. Array name 'x' will represent the starting address (base address) of an array.



Link–List also a collection of homogeneous data like array, but in the case of Link–List the data elements are stored in non– contiguous manner. Each data element is encapsulated into the data structure called node. Each node of the link list has data and address to the next node, which represent another data entity. As shown in the figure data elements 5 and 10 are stored in two different node and their addresses are non– contiguous (100 and 200). Each node has an address field which provides address to the next node (which contains next data element).

❑ Check Your Progress – 3 :

1. _____ Linear data structure stores data element in non– contiguous manner.

   [A] Array [B] Link–List

   [C] Tree [D] None of the above

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

   [A] Array [B] Link–List

   [C] Stack [D] All of the above

4.4 Link–List Implementation :

4.4.1 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 a pointer variable which can store the address of any node. Usually, first pointer holds the address of the first node of the link list. Initially, when the program is started, and there is no node is there in the link list (Link–List is empty) first pointer do not have any address, at that time first has NULL value.

When user insert the first value, and first node is created in the link list, pointer first will have the address of the first node.

4.4.2 Creating Link–List :

To create a link list, user will invoke create () function. In this function, we take how many nodes has to be created in the link list, from the user. Suppose, if user enters 4, then we take 4 values from the user and create 4 nodes in the link list. Each node contains one value given by the user. Each node also stores the address of the next node. The next part of the 4th node (last node) will be NULL and the address of the first node will be placed in the pointer 'first'.

```
void create()
{
   struct node *newnode;
   int i,n;
   printf("\nEnter Number of Elements to be entered in the Link List:");
   scanf("%d", &n);
   for(i=1; i<=n; i++)
   {
      if(first==NULL)
      {
         newnode = (struct node *) malloc (sizeof (struct node));
         first = newnode;
      }
      else
      {
```

```
            newnode–>next = (struct node *) malloc (sizeof (struct node));
            newnode = newnode–>next;
        }
        printf("Enter Value :");
        scanf("%d", &newnode–>data);
        newnode–>next=NULL;
    }
}
```

❑ **Check Your Progress – 4 :**

1. To create a new node _____ function is used.

   [A] create () [B] add () [C] new () [D] malloc ()

2. The default value (when the program is started), first pointer should have _____ value.

   [A] Address of first node [B] Address of last node

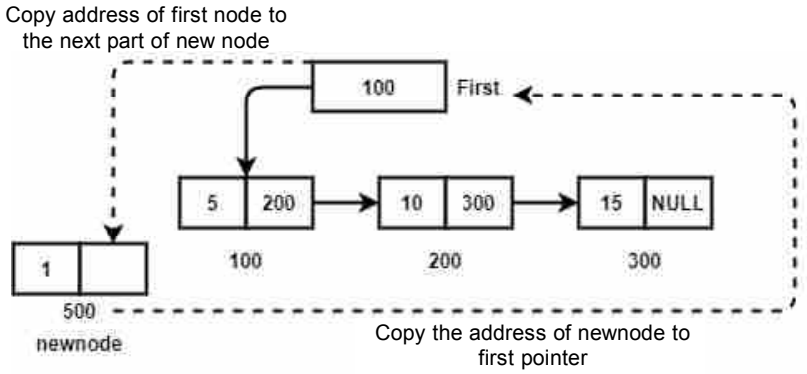
   [C] NULL [D] None of the Above

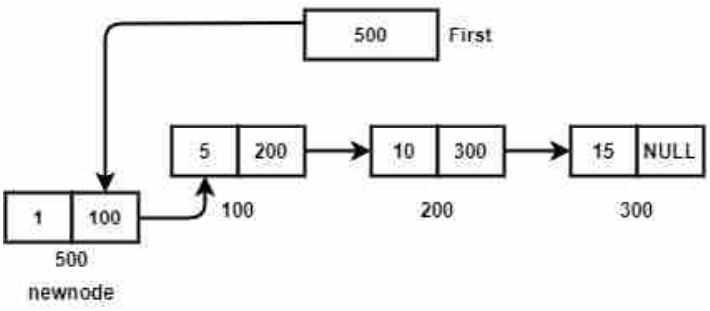
4.4.3 Inserting a Value to The Link–List :

Insertion can be done at both ends of the link list. To insert the value in the beginning of the link list, we will create a function called insert_at_beg() and to insert the value at the end of the link list, we will create a function called insert_at_end().

[1] Inserting value at the beginning of the Link List :

To insert the, at the beginning of the link list we will take a value from the user. We will create a new node and set the value taken from the user in the data part of the new node. We will copy the address stored in the first pointer to the next part of the new node. Finally, the address of the new node we will copy to the first pointer.





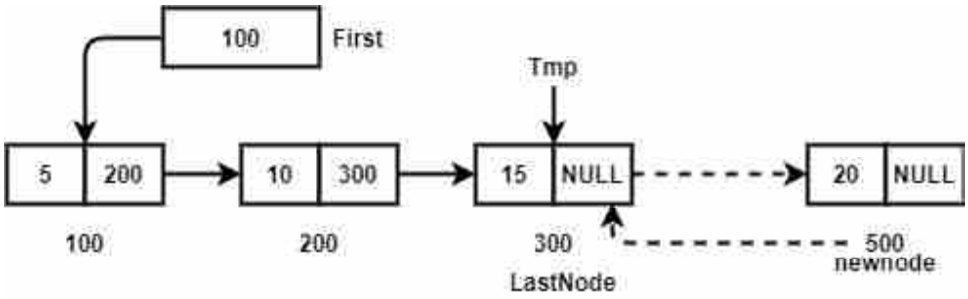
Suppose, in the Link–List, 3 nodes are inserted, by create () function. These values are 5, 10 and 15. So there are 3 nodes are there in the link list. Now, suppose user wants to insert a new node with value 1, at the beginning of the link list. To do this, first create a new node using malloc () function. Suppose the new node is created on memory address 500. Put 1 (value which user wants to insert) in the data part of newnode, copy the address 100 from the pointer first to the next part of newnode. so, the newnode will point to the node '5'. Copy the address of newnode (500) in the pointer first. Therefore, first will gives you the address 500 (first node) and the next part of the 500 will gives you the address 100, which will become second node.

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

   if (first == NULL)
   {
      printf ("\n linklist is empty");
      return;
   }
   new1–>next = first;
   first=new1;
}
```

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

To insert the value at the end of the link list, we will take a temporary pointer to find the last node of an existing link list. We will create a new node, and place the data given by the user in the data part of the new node. We will also set the next part of the new node to NULL. Then we will start to search last node. Last node is that node in which NULL value is there in the next part. When we get the last node, we will copy the address of the new node in the next part of the last node.



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

   tmp = first;
   while (tmp–>next != NULL)
   {
      tmp = tmp–>next;
   }
   tmp–>next = newnode;
}
```

4.4.4 Displaying Link–List :

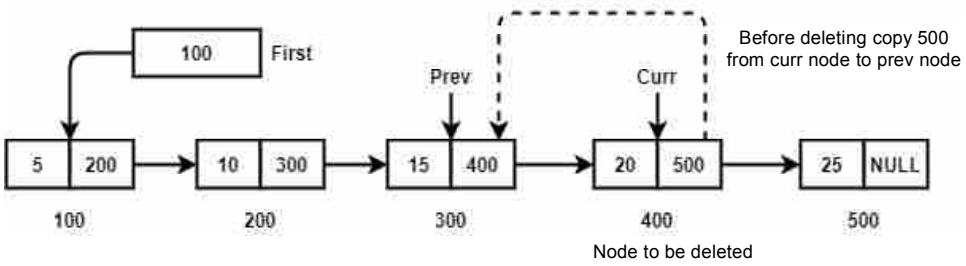
To display the link list, take the temporary pointer 'tmp' and place it on the first node, if the, first pointer doesn't have NULL value. If first pointer has NULL value then Link–List is empty and nothing to display, just return from the function. Else print the value from the data part and move to the second node, by taking new address from the next part of 'tmp' pointer. Continue this process till 'tmp' will not become NULL.

```
void display ()
{
   struct node *tmp;
   if (first == NULL)
   {
      printf ("Link–List is empty :");
      return;
   }
   tmp = first;
   while (tmp != NULL)
   {
      printf("\n%d —>",tmp–>data);
      tmp = tmp–>next;
   }
}
```

4.4.5 Deleting a value from the Link–List :

To delete the value from the Link–List, we need to check multiple conditions. First will check if, pointer first is NULL then link list is empty and nothing to delete from it, so simply return from the function. We need to check if user wants to delete the first node, then we need to copy the address

of the second node in the first pointer, and then we need to delete the first node. Otherwise, we need to search the element in the link list. If the value to be deleted is present in the link list, then we need to copy the address placed in the next part of that node, to the next part of its previous node. If the value is to be deleted is not exists then show message that the value does not exists in the Link–List.



To delete the node, we use a pointer 'curr' to find the node to be deleted. We place a 'curr' pointer on the first node, and start searching that node in which data part has a value which is given by the user to delete the node. When we find the value, we need to copy address of next node (500 in the figure, which is in the next part of the 'curr' node) to the next part of its previous node.

In a singly link list, each node has address to its next node. So, we can not access the previous node. To access the previous node, we need to declare additional pointer variable called 'prev'. The pointer variable 'prev' will follow the 'curr' pointer variable. That means, when the 'curr' pointer will reach to the address 400, at that time, 'prev' pointer will be on its previous node which is 300 in the figure. We can now easily copy the address 500 which is in the next part of the curr node, to the next part of the prev node by just writing an instruction :

pre–>next = curr –>next;

This is one of the drawbacks of singly link list. In singly link list, each node has address to its next node, but no node has address to its previous node. So, whenever we need to access the previous node, we need to use an additional pointer variable. To implement deletion in the singly link list, we make 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 described below :

```
void delnode (int val)
{
   struct node *curr, *prev;
   //If Link list is Empty then nothing to be deleted
   If (first == NULL)
   {
      printf("\n link–list is empty");
      return;
   }
   //Place curr pointer to the fist node
   curr = first;
```

```
    //If value to be deleted is found in the first node
    if (first–>data == val)
    {
       first = first–>next;
       free (curr);
       return;
    }
    //Search the value in the Link List
    While (curr–>data != val && curr != NULL)
    {
       //Previous pointer is following to the current pointer
       prev = curr;
       curr = curr–>next;
    }
    //If value to deleted is not found
    if (curr == NULL)
       {
          printf("\n value does not exist");
          return;
       }
    //Copying the address of next to the previous node's next part
    prev–>next = curr–>next;
    //Deleting node
    free (curr);
}
```

Finally, when we find the particular node in which the value of data part match with the value to be deleted (given by the user), we can use free () function to delete that node. To delete the desire node, we need to write :

free (curr);

Function free () will delete the node structure, specified on the address stored in the 'curr' pointer variable.

❑ Check Your Progress – 5 :

1. In _____ function of the singly link list we need to take an addition pointer variable to access previous node.

   [A] delnode () [B] display ()

   [C] create () [D] insert_at_end ()

2. _____ dynamic memory allocation function is used to delete the node from the link list.

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

4.5 Let Us Sum Up :

In this unit, we :

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

4.6 Suggested Answers for Check Your Progress :

❑ **Check Your Progress 1 :**

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

❑ **Check Your Progress 2 :**

1. [A] **2.** [D]

❑ **Check Your Progress 3 :**

1. [B] **2.** [D]

❑ **Check Your Progress 4 :**

1. [D] **2.** [C]

❑ **Check Your Progress 5 :**

1. [A] **2.** [C]

4.7 Glossary :

1. **Pointer :** Pointer is a special variable, which can store the address of another variable.

2. **Link–List :** Link–List is a linear data–structure, which stores (collection) homogeneous (same type of) data in non–contiguous memory locations.

4.8 Assignment :

1. Write a function called 'insbfr (int val, int key)', which will insert the new value to the link list before the key value.
2. Write a function called 'insaft (int val, int key)', which will insert the new value to the link list after the key value.

4.9 Activity :

1. Write a function to sort a link list.

4.10 Case Study :

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

4.11 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
05

MORE ON LINK–LIST

UNIT STRUCTURE

5.0 Learning Objectives :

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

- Different types of Link–List
- Understand doubly Link–List
- Implement the programs of Link–List

5.1 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 Link–List and their implementation.

5.2 Types of Link–List :

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

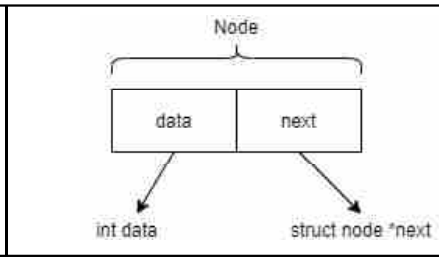
[2] Doubly Link–List and,

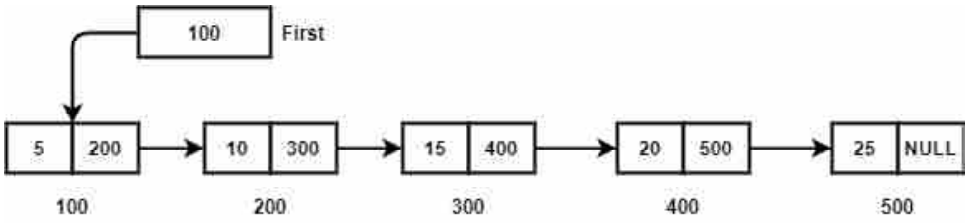
[3] Circular Link–List.

5.2.1 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;
};
```





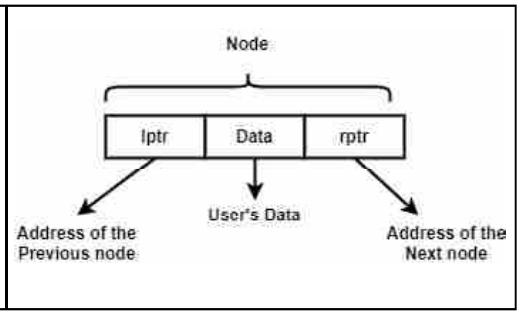
Singly Link–List

5.2.2 Doubly Link–List :

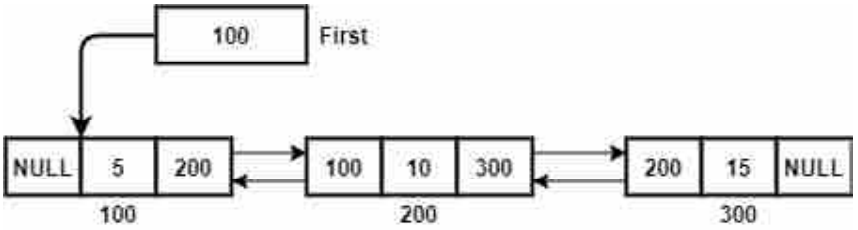
The problem with the singly Link–List is, it contains only one address part in each node of the Link–List in which we store the address of the next node. Due to this reason, we cannot have access to the previous node. Singly Link–List provides forward pathway, but it doesn't provide backward traversal 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;
};
```



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.



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

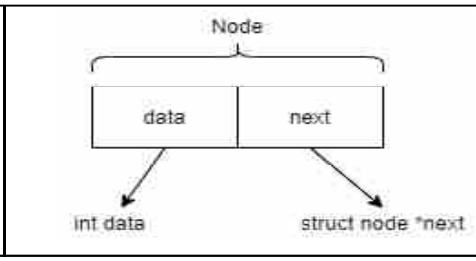
   [C] Circular [D] All of the above

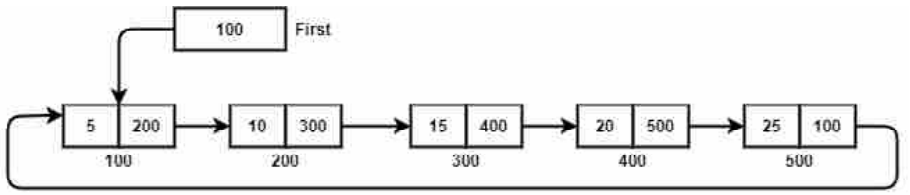
5.2.3 Circular Link–List :

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;
};
```





❑ **Check Your Progress – 2 :**

1. In doubly link–list, lptr of the first node and rptr of the last node will be _____.

   [A] first node [B] next node

   [C] previous node [D] NULL

2. In circular link–list, last node has _____ address in its next part.

   [A] first node [B] next node

   [C] previous node [D] NULL

5.3 Doubly Link–List Implementation :

5.3.1 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.

5.3.2 Creating a Doubly Link–List :

To create a doubly link–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 :

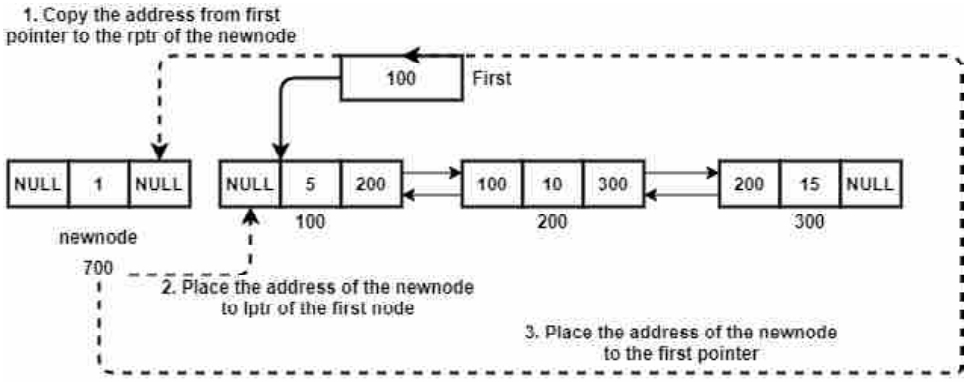
```
void create()
{
   struct node *tmp, *newnode;
   int val, i;
```

```
/*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;
   }
}
```

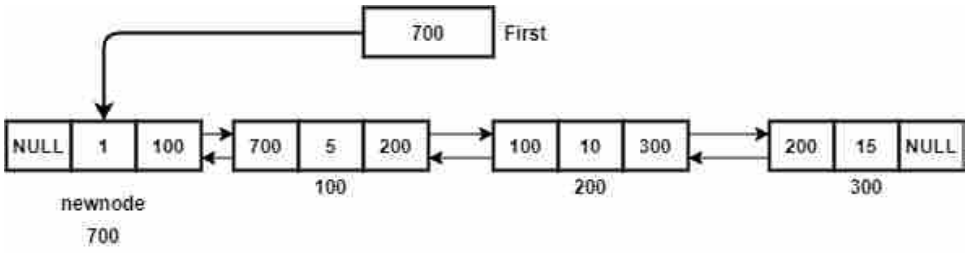
5.3.3 Inserting a Value to The Link–List :

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.

[1] Inserting value at the beginning of the Doubly Link–List :



As shown in the above figure, to insert the value at the beginning of a doubly Link–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.

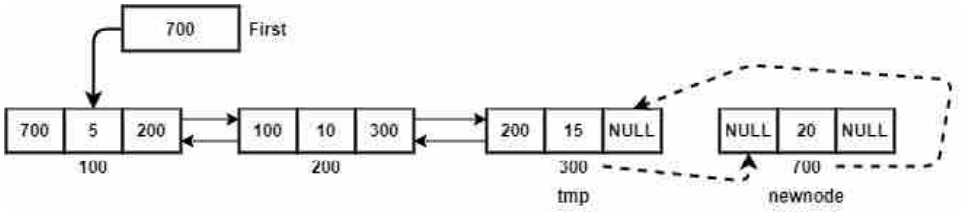


The code for the function inser_at_beg (), to insert the value to the beginning of the doubly linked is given below :

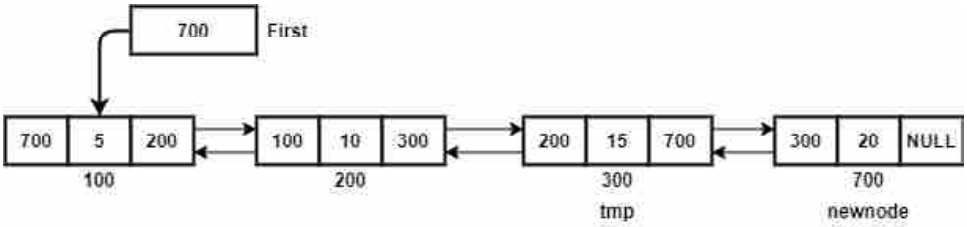
```
void inst_at_beg (int val)
{
struct node *newnode;
   if (first == NULL)
   {
      printf("Doubly Link–List is empty");
      return;
   }
newnode = (struct node *) malloc (sizeof (struct node));
newnode–>data = val;
newnode–>lptr = NULL;
newnode–>rptr = first;
first–>lptr = newnode;
first = newnode;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

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

To insert the value at the end of the doubly Link–List, first we will create a new node using malloc () function. We will store the user's data (20) into the data part of the newnode, and set NULL into the rptr of the newnode. We will then take a temporary variable tmp to search the last node of the doubly Link–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 :



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

5.3.4 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);
}
```

5.3.5 Deleting a Node From Doubly Link–List

To delete the node from a doubly link–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 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

5.4 Let Us Sum Up :

In this unit, we :

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

5.5 Suggested Answers For Check Your Progress :

- **Check Your Progress 1 :**

  1. [A] **2.** [A]

- **Check Your Progress 2 :**

  1. [D] **2.** [A]

- **Check Your Progress 3 :**

  1. [D] **2.** [A]

- **Check Your Progress 4 :**

  1. [C] **2.** [C]

5.6 Glossary :

1. **Doubly Link–List –** List is also a type of Link–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.

5.7 Assignment :

- List and Explain different types of Link–Lists.

5.8 Activity :

1. Write a function to reverse the doubly Link–List.
2. Write a function to sort the doubly Link–List.

5.9 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

5.10 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
06

STACK AND THEIR APPLICATIONS

UNIT STRUCTURE

6.0 Learning Objectives :

After learning this unit, you will be able to :

- Understand the Concept of Stacks.
- Understand the Method of Representation of Stacks.
- Understand the Applications of Stacks.
- Implement Stacks.

6.1 Introduction :

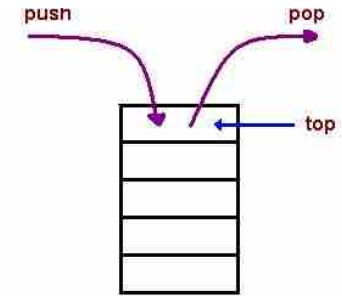
In this unit, we'll be discussing about one of the elementary data structures called Stacks. When the data elements are stored one above the other, then this type of arrangement is called a stack. A stack commonly stands for a block of memory cells, with the "bottom" on a fixed location, and the stack pointer holding the address of the current "top" cell in the stack.

We will also be discussing about the static (array) and dynamic (linked list) representations of stacks.

6.2 Definitions :

❖ **Stack :**

A stack is a linear data structure in which elements are inserted or deleted through one end which is called the top of the stack. Stack is a last in first out (LIFO) structure. This can be well understood by help of following diagram.



In stack, objects contained are inserted or removed based on the (LIFO) last–in–first–out method. Only two possible operations can be allowed in pushdown stacks namely, push in the item into the stack and secondly pop the item out of it. The stack data structure is rather limited or minimalistic; elements can be added and removed only at the top. In order to insert an item to the top push is used, whereas in the case of removing an item pop is used.

For example, think of a library where one can remove a book only from the top of the stack of books and can also add a new book only at the top.

Hence, we can say that stack is a recursive data structure.

❑ **Check Your Progress – 1 :**

1. A Stack is a _____ data structure.

   [A] FIFO [B] LIFO [C] FILO [D] LILO

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

   [A] front [B] rear

   [C] top [D] All of the above

6.3 Array and Link–List Representation of Stack :

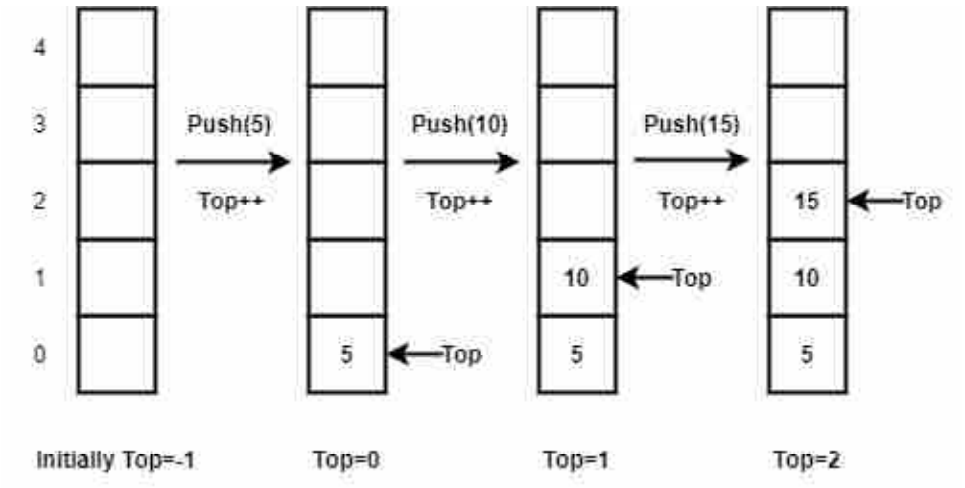
6.3.1 Array Representation of Stack :

Stack is a linear data structure, which is used as storage of data in LIFO ordering. Stack stores user's data and return it back to the user in Last In First Out manner. To store user's data, we can use either Array or Link–List. We have already discussed that the both Array and Link–List are linear data structure which stores user's data. Just the difference is, Array stores the data in contiguous memory location whereas the Link–List stores data elements in non–contiguous memory locations.

To implement the stack, we need to take an array and a top variable (integer). Initially the top variable has to be initialized with value –1.

[1] PUSH () Function :

Function push () is used, when user want to insert an element into the stack. Each time when user call a push function, first we need to check for overflow condition. Suppose if the capacity of the array is to store MAX elements, and if variable top is on MAX –1 then, we can say that the array is full and it doesn't have space to accommodate new element. This situation in the stack, where user want to insert an element but array is full is called OVERFLOW condition. If there is no overflow situation is there, then the variable top has to incremented by 1, and the new element has to be placed on the top position of the array. Each time when user, push an element top variable is incremented by 1 and element has to be placed on the top position of an array. It is shown in the following figure.



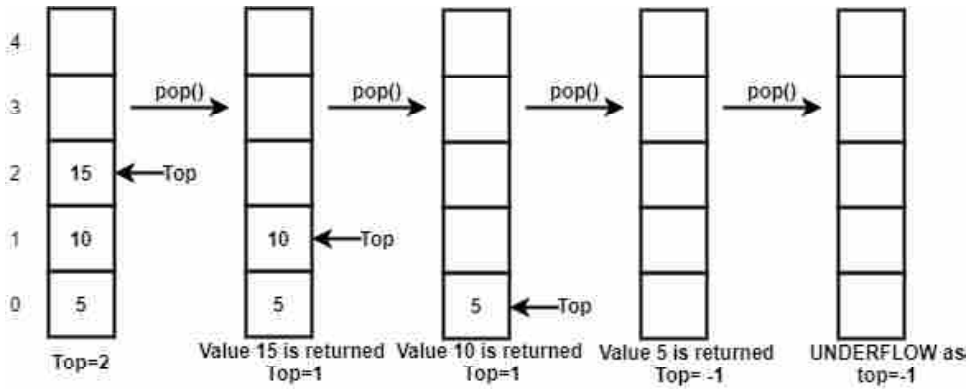
As shown in the figure, the initial value of top variable is –1. On each time when user call a push function, top variable will be incremented and on the top position of an array, the newer element has been placed. When the user calls a push function and top is on 4th position, we need to show overflow message.

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

```
#include <stdio.h>
#define MAX 5
int stack [MAX];
int top = –1;
void push (int val)
{
   if (top == MAX –1)
   {
      printf ("\n Stack Overflow :");
      return;
   }
   top++;
   stack (top) = val;
}
```

[2] POP () Function :

When the user invokes the pop () function, stack will return the value from the top position of an array (LIFO). If top is on –1 and user invoke pop () function, there is nothing in the stack to return. This situation is called UNDERFLOW condition. After returning the value, it is to be assumed that the value is deleted from the stack, therefore the value of top variable has to be decremented by 1.



As shown in the figure, consider a stack of size 5, and 3 elements are pushed in the stack called 5, 10 and 15. The variable top will be 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, 10 will be returned and the again top will be decremented to 0. On third pop () call, value 5 will be returned and top variable is decremented to –1. Now, there is no value is there in the stack. Suppose, still user calls the pop () function, we have to show underflow message as there is nothing 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)
   {
      printf ("\n Stack Underflow :");
      return 0;
   }
   tmp = stack[top];
   top—;
   return tmp;
}
```

[3] PEEP () Function :

The function is peep is similar to pop () function. It returns the value which is stored on the top of the stack. The only difference between pop () function and peep () function is, pop () function assumes the data to be deleted from the stack after 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, variable top will not be decremented by 1.

```
int peep ()
{
   int tmp;
   if (top == –1)
```

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

❑ **Check Your Progress – 2 :**

1. In the implementation of the stack by array, initial value of top will be _____.

   [A] 0 [B] 1 [C] –1 [D] NULL

2. Overflow condition for the stack has to check in _____ function.

   [A] push () [B] pop () [C] peep () [D] NULL

6.3.2 Link–List representation of Stack

Stack can also be implemented using Link–List. To implement the stack using link–list we need to define structure node as shown below :

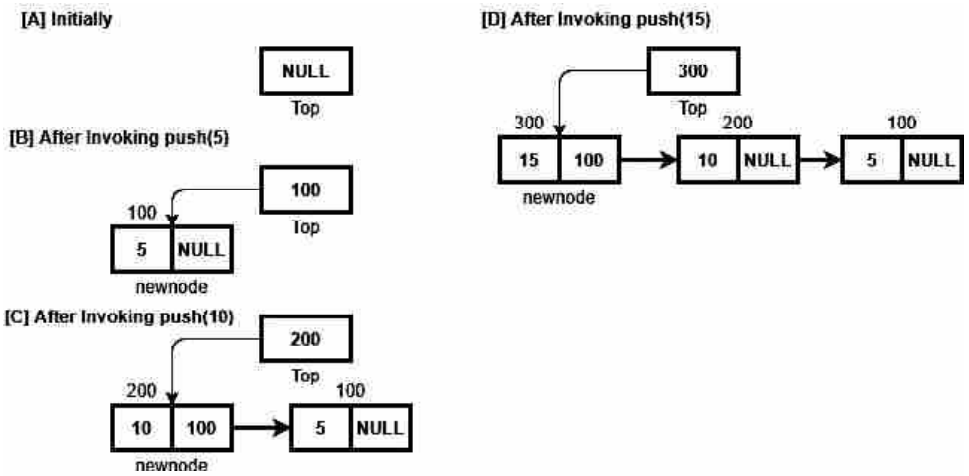
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
} *top =NULL;
```

Definition of structure node is similar to the Link–List program. Node is defined with an integer variable call data to store the user's data, and next variable of type struct node *, to store the address of next node. We are declaring a pointer variable top of type struct node *, which will always point to the recent (latest) value pushed (inserted) by the user. Pointer variable *top will be initialized with NULL value initially.

[1] PUSH () Function :

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

If the node is created, we need to place the address stored in the *top, to the next part of the newnode. The following figure demonstrate the stack implemented by Link–List.



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

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

[2] POP () Function :

As we know pop () function delete the value (node), which is on the top, and return that value. To do this, we take a pointer tmp pointer and place it on the top node. We will move the *top to its next node. We will copy the data element of the tmp pointer to some variable called val. We will delete the tmp node and finally return the value of the val variable. As we know, in the pop () function we need to check for UNDERFLOW condition. So, first we will check the *top. If *top is NULL, we will show the error message of underflow condition. The code for pop () function is given 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;
}
```

[3] PEEP () Function :

As we know the peep function, returns the value situated on the top of the stack, without deleting it from the stack. The following code need to be written to implement peep () function.

```
int peep ()
{
    if (top == NULL)
    {
        printf ("\n Stack Underflow");
        return 0;
    }
    return top –> data;
}
```

❑ **Check Your Progress – 3 :**

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

   [A] 0 [B] 1 [C] –1 [D] NULL

2. For Overflow condition in the stack implemented by Link–List, _____ condition is checked.

   [A] newnode == NULL [B] top == NULL

   [C] top == –1 [D] Node of the above

3. For Underflow condition in the stack implemented by Link–List, _____ condition is checked.

   [A] newnode == NULL [B] top == NULL

   [C] top == –1 [D] Node of the above

6.4 Operations and Applications of Stack :

❖ **Basic Operation of Stack :**

The bottom of a stack is a sealed end; this means there is no chance of insertion. Stack may have a capacity which is a limitation on the number of elements in a stack. The operations on stack are :

Push – Places an object on the top of the stack or in the beginning.

Pop – Removes or erases an object from the top of the stack.

Peep – Return the value of the tope of the stack without removing it from the stack.

Is Empty – This report whether the stack is empty or not.

Is Full – This report whether the stack exceeds limit or not.

1. Expression Evaluation,
2. Expression conversion like,
   - Infix to Postfix.
   - Infix to Prefix.
   - Postfix to Infix.
   - Prefix to Infix.
3. Parsing
4. Recursion and Function call
5. String Reversal

❑ **Check Your Progress – 4 :**

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

   [A] push [B] pop

   [C] peep [D] All of the above

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

   [A] Recursion [B] Infix to Postfix

   [C] Finding shortest path [D] String reversal

Expression Representation (conversion) – Evaluating arithmetic expressions

Infix	Prefix	Postfix
x +y	+ x y	x y+
x + y * z	+ x * y z	x y z * +
(x + y) * (z – n)	* + x y – z n	x y + z n – *

❖ **Procedure for Postfix Conversion :**

1. First Scan the Infix string from left to right and.
2. Then Initialize an empty stack.
3. In case the scanned character is an operand, add it to the Postfix string.
4. In case the scanned character is an operator and if the stack is empty push the character to stack.
5. If the scanned character is an Operator and the stack is not empty then compare the precedence of the character with the element on top of the stack properly.
6. In case the top Stack has higher precedence over the scanned character pop the stack else push the scanned character to stack. Repeat or redo this step until the stack is not empty and the top Stack always has precedence over the character.
7. Repeat or redo step 4 and 5 till all the characters are scanned.

8. In case all the characters are scanned, we have to add any character that the stack may have to the Postfix string.
9. Suppose if stack is not empty add top Stack to Postfix string and Pop the stack.
10. Repeat or redo this step as long as the stack is not empty (until and unless the stack is not empty).

❖ **Reversing Data :**

We can use stacks to reverse data (example : files, strings). It is also very useful and important for finding palindromes.

Consider the following pseudo code and see the outcome –

a. read (data)
b. loop (data not EOF and stack not full)
    1. push (data)
    2. read (data)
c. loop (while stack not Empty)
    1. pop (data)
    2. print (data)

Converting Decimal numbers to Binary –

Consider the following pseudo code and see the outcome –

1. Read (number)
2. Loop (number > 0)
    a. digit = number modulo 2
    b. print (digit)
c. number = number / 2

The problem with this code is that it will print the binary number backwards.

(For Example : 19 become 11001000 instead of 00010011).

To overcome this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

1. Evaluating arithmetic expressions (This is another example of arithmetic expression)

A prefix, infix and postfix expression can be written as :

Prefix : + a b

Infix : a + b

Postfix : a b +

In high level languages, an infix notation cannot be used to evaluate expressions. We must analyse the expression to determine the order in which we evaluate it. A common technique is to convert an infix notation into a postfix notation, then evaluating it.

Infix to Postfix Conversion –

Before discussing about the infix to postfix conversion first of all let's discuss about the priority of operators. Given below is the hierarchy of priority of operators :

(1) ^ (Exponentiation) – Highest Priority

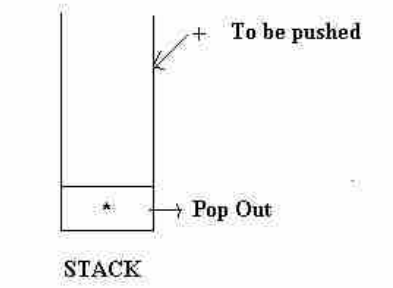
(2) *,/ (Multiplication/Division)

(3) +,– (Addition, Subtraction) – Lowest Priority

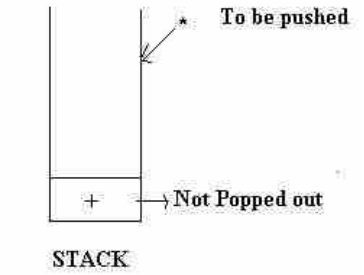
So, it can be concluded that exponentiation has higher priority multiplication/ division have equal priority and similarly addition/subtraction have equal priority.

An infix expression can be converted into postfix form by pushing the operands on the stack. The Rules for conversion of the same is given below :

If the stack contains a higher priority operator and we are trying to push a lower priority operator, then it will not be allowed until and unless the lowest priority operator is popped out, to understand this see following example,



2. Now if the stack contains a lower priority operator and we are trying to push a higher priority operator, then it will be allowed, to understand this see following example,



1. If the stack contains an equal priority operator and we are trying to push an operator of same priority, then it will be allowed, to understand this see the following example,

   Algorithm for Conversion of an Infix expression to Postfix. The following explanation will help you to understand better :

   Suppose A is an arithmetic expression written in infix notation. The given algorithm finds the equivalent postfix expression P, Push "("onto STACK, and add ")" to the end of A.

2. Scan A from left to right and repeat steps 3 to 6 for each element of A until the STACK is empty.

3. If an operand is encountered, add it to P.

4. If a left parenthesis is encountered, push it onto STACK.

5. If an operator is encountered, then :
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as, or higher precedence.
   (b) Add the operator to STACK. [End of if structure]
6. If a right parenthesis is encountered then :
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (b) Remove the left parenthesis. (Do not add the left parenthesis to P) [End of if structure]

   [End of Step 2 loop]
7. Exit

The example below shows us the outcome of the above algorithms, but with different expressions.

A * B - (C + D) + E

	Infix	Stack(bot->top)	Postfix
a)	A * B - (C - D) + E	empty	empty
b)	* B - (C + D) + E	empty	A
c)	B - (C + D) + E	*	A
d)	- (C + D) + E	*	A B
e)	- (C + D) + E	empty	A B *
f)	(C + D) + E	-	A B *
g)	C + D) + E	- (	A B *
h)	+ D) + E	- (	A B * C
i)	D) + E	- (+	A B * C
j)	) + E	- (+	A B * C D
k)	+	-	A B * C D
l)	+ E	empty	A B * C D + -
m)	E	+	A B * C D + -
n)		+	A B * C D + - E
o)		empty	A B * C D + - E +

❑ **Check Your Progress – 5 :**

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

   [A] 97–3*3%2+ [B] 2397–*3%+ [C] 23+97–*3% [D] 23973+*–%
2. _____ is also known as reverse polish notation.

   [A] Infix [B] Prefix

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

   [A] AB*CD+–E+ [B] ABCDE*–++

   [C] AB*CD+–E+ [D] CD+B–A*E+

6.5 Let Us Sum Up :

In this Unit there is lot of leaning about Stack. Whenever the data elements are arranged one above the other or one after another, then this type of structure is called stack. In a stack, data elements can be always inserted from top of the stack and can also always be deleted from the top position; hence, it follows, Last in First out (LIFO) order. The addition of an element

to a stack is called performing PUSH operation of function and deletion of an element from a stack is called performing POP operation or function.

Consider the time of insertion of an element, if the stack is full then this condition is always called Stack Overflow. And, similarly, consider that at the time of deletion if the stack is empty and we are trying to delete an element, then this condition is always known as Stack Underflow. Stacks are represented in two ways namely 1) Array Implementation and 2) Linked List Representation.

The Push and Pop operations for a Stack can be implemented as : PUSH Function : The Push function adds an element to the stack. At the time of addition of an element, the stack overflow condition is checked, that is, if the stack is full or not. If not, then the position of top pointer is shifted up by one place and the desired element is added at that position. Now, the element which was last inserted becomes the TOP element. There is POP Function : The Pop function deletes an element from the top of the stack. At the time of deletion of an element, the stack underflow condition is checked out, that is, if the stack is empty or not, if not, then, the element to be popped out is first stored into a variable, then the position of top pointer is decremented by one step, making the next element as TOP element.

There is Infix to Postfix Conversion. Now at the end of this unit we also discussed about the priority of operators. Given below is the hierarchy of priority of operators. Two types of priority of operator :

1. ^ (Exponentiation) – Highest Priority
2. *,/ (Multiplication/Division)
3. +,– (Addition, Subtraction) – Lowest Priority

It can be concluded that exponentiation has always higher priority multiplication/and division have always equal priority and similarly addition/ subtraction have equal priority.

6.6 Suggested Answers For Check Your Progress :

- **Check Your Progress 1 :**

  1. [B] **2.** [C]

- **Check Your Progress 2 :**

  1. [C] **2.** [A]

- **Check Your Progress 3 :**

  1. [D] **2.** [A] **3.** [B]

- **Check Your Progress 4 :**

  1. [D] **2.** [C]

- **Check Your Progress 5 :**

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

6.7 Glossary :

1. **Stack :** Stack is a linear data structure, which can be implemented by either array of Link–List. It is collection of data implemented in LIFO (Last In First Out) ordering.

2. **Inorder :** Inorder is a method of writing expression, in which all binary operators are written between two operands. In our day to day lives, we use Inorder expression.

6.8 Assignment :

- Write a program to convert any given Infix expression to Postfix.

6.9 Activity :

1. Write a program to reverse the string using stack.
2. Write a program to find Binary string of a given number using stack.

6.10 Case Study :

Search 8–Queen puzzle on the Internet and try to understand it. Also write the program to solve 8–Queen puzzle to find and print all possible combinations.

6.11 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
07

QUEUES AND THEIR APPLICATIONS

UNIT STRUCTURE

7.0 Learning Objectives :

After learning this unit, you will be able to :

- Understand Basic Operations Performed on Queue
- Understand Array and Link–List Representation of Queue
- Understand D–Queue
- Understand Circular Queue
- Know Application of Queue

7.1 Introduction :

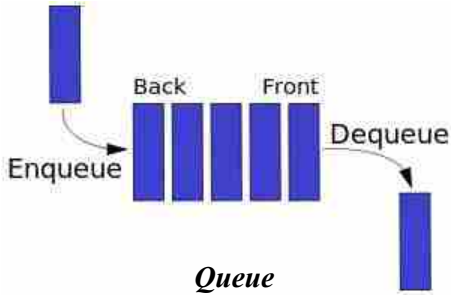
In the last unit we have studied about the stack data structure, in which the data elements were arranged one above the other. In this unit, we'll be discussing about another type of data structure called queues in which the data elements are arranged one after the other. For Example, a queue of people at a railway reservation counter or in a network where many computers share a few printers, then the print jobs may accumulate in a print queue. We'll also be discussing about the array and linked list representation of queues. Apart from this we are going to focus on various types of queues. This is of much importance in a data structure. As you know in the earlier chapter there is a limitation in case of stacks for the data structure hence in this chapter you

are going to focus on further features of the data structure operations or functions.

7.2 Definition :

❖ **Queue :**

Queue is a data structure in which elements are added through the rear end and deleted through the front end.



Queue

A queue is a container or open box of objects (of a linear collection) that are inserted one after the other and removed 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.

❖ **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 [B] LIFO [C] FILO [D] LILO

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

   [A] front [B] rear

   [C] top [D] All of the above

3. Value will be removed in the queue from _____ position.

   [A] front [B] rear

   [C] top [D] All of the above

7.3 Basic Operations Performed on Queue :

The two basic operations performed on queue are insert and delete.

Elements in queue are inserted through the rear end using the insert operations and deleted through the front end using delete operation.

Queue operations :

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

7.4 Array and Link–List Representation of Queue :

Queues can be implemented statically or dynamically i.e., as an array or as a linked list.

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

7.4.1 Array Representation of Queue :

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

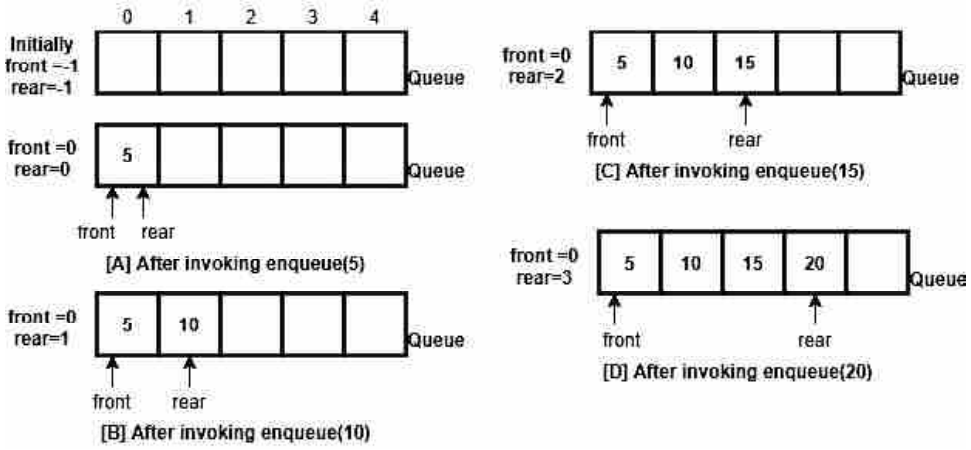
Recall stack data structure. In the stack data structure, we have taken a variable top, because in the stack, elements of the stack are inserted on the top, and removed from the top position. In the case of queue, elements are inserted at the rear position and removed from the front position. So, to implement queue we 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, we need to initialize front are rear variable with value –1. Variable front and rear are –1, that indicates that the queue is empty, and there is no element is there in the queue.

[1] Enqueue () Function :

As we know, user will invoke enqueue () function to insert an element into the queue. We have already discussed that the value in the queue is inserted at rear position. So, to implement enqueue () function, we will check the rear position if it is MAX –1, then queue is full (as array queue, do not have space to accommodate new value). Otherwise, we need to increment the value of rear variable by 1, and we need to place the value on the rear position of the queue array. When the first 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.

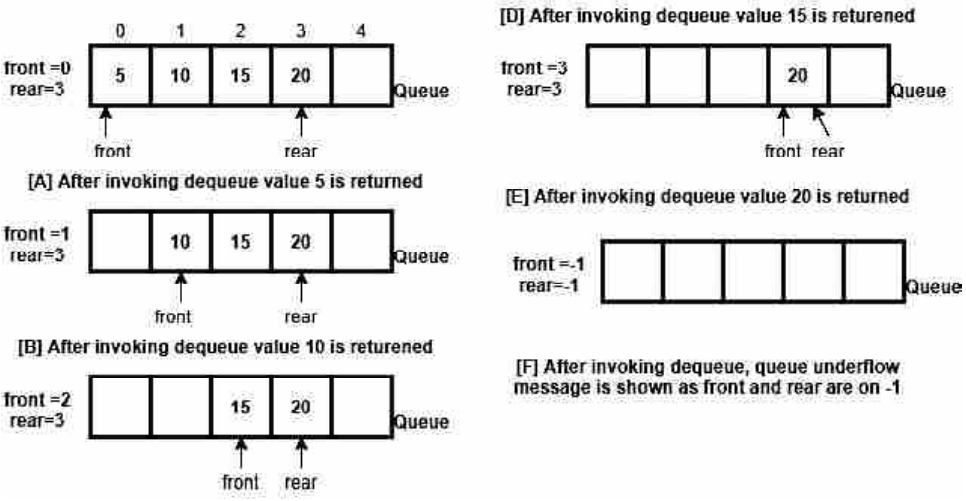


Function enqueue () can be implemented as follow :

```
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 dequeue () function, we need to check front variable, it is –1 then the queue is underflow (as there is no element in the queue array to delete and return). Otherwise, we copy the element situated on the front position of the array into the temporary variable 'tmp' and increment the front variable. Before doing this, we need to check one more condition. If the value of front and rear variables is same, that means user is deleting the last value from the queue. In such situation we need to set front and rear both variables to –1. The following figure will help you to understand the function 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++;
    return tmp;
}
```

❑ **Check Your Progress – 2 :**

1. Identify the TRUE statement from the given below :

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

   [B] In enqueue () function, front variable is decreased and in dequeue function it rear variable will be decrease.

   [C] In enqueue () function front will be increase and in dequeue () function rear will be increase

   [D] None of the above.

2. Identify the statement, we need to write in dequeue () function, while implementing queue using an array.

   [A] if (rear == MAX –1) [B] rear++;

   [C] queue[rear] = val; [D] front++;

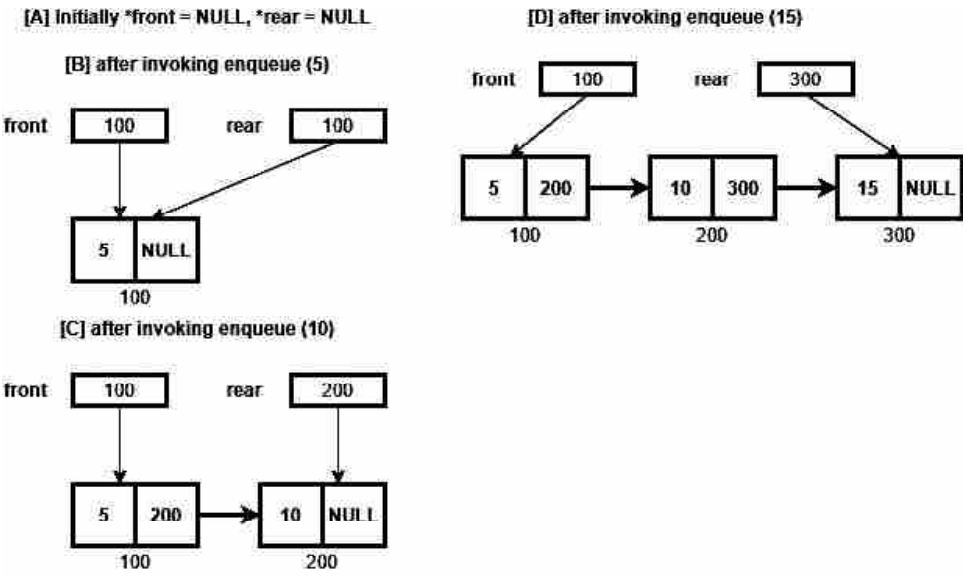
7.4.2 Link–List Representation of Queue :

Data structure Queue can be implemented using Link–List data structure. Like singly Link–List we will create a structure of node with two data elements : data and next. We will declare two pointer variables of type 'struct node' : front and rear. Both pointers, * front and * rear will ne initialized with NULL value.

```
Struct node
{
   Int data;
   Struct node *next;
} *front = NULL, *rear = NULL;
```

[1] Enqueue () Function :

To implement enqueue () function, we need to create a new node using malloc () function. If malloc () function returns NULL value, instead of address of new node, we will print the message that Queue is full. Otherwise, we will insert the newnode to 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 always point to the last node of the link list. Figure given below will help you to understand functioning of the enqueue () function of the queue implemented using Link–list.

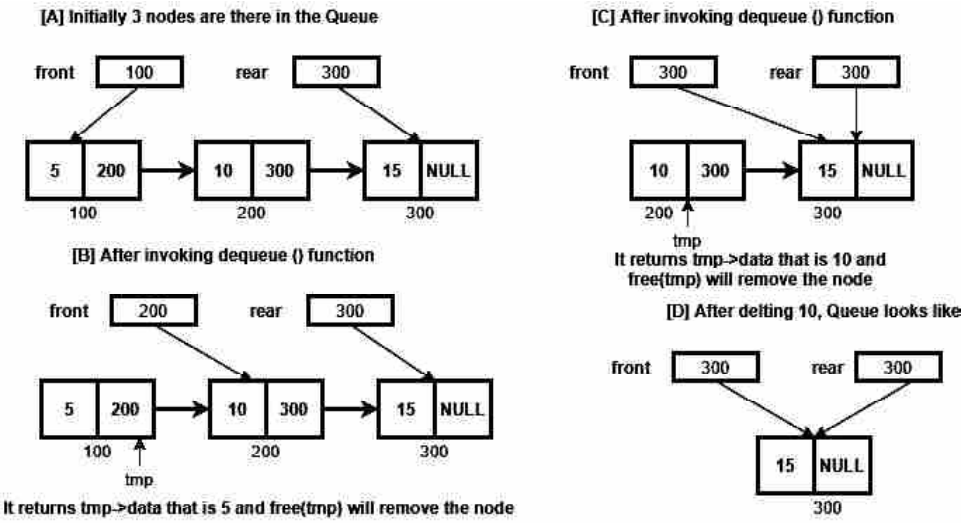


From the figure, you will come to know that, initially when we execute the program there no node is there in the Link–List and * front and * rear both will have NULL value. When user inserts value 5 to the queue, we create a new node and both pointers, *front and *rear points to this node. When user inserts another value 10, at that time a new node is created and will place value 10 into the data part of the new node. We will copy the address of the new node into the next part of the rear node (*rear points to memory location 100). Now because the new node with value 10 is inserted in the Link–List, rear pointer should point to newly inserted node. Therefore, we will place the address of new node into the rear pointer. The following code, we need to write to implement enqueue () function.

```
void enqueue (int val)
{
   struct node *newnode;
   newnode = (struct node *) malloc (sizeof (struct node));
   if (newnode == NULL)
   {
      printf("\nQueue is Full :");
      return;
   }
   newnode–>data = val;
   newnode–>next = NULL;
   if (front == NULL)
   {
      front = rear = newnode;
      return;
   }
   rear–>next = newnode;
   rear = newnode;
}
```

[2] Dequeue () Function :

As we know dequeue () function is used to remove element from the queue, and in the queue, element is removed from the front position. 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 will 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;
}
```

❑ Check Your Progress – 3 :

1. Queue implemented using Link–List, *front and *rear will be _____ initially.

   [A] –1 [B] 0 [C] 1 [D] NULL

2. In queue data structure, _____ condition needs to be check in dequeue ().

   [A] overflow [B] underflow

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

3. In _____ () function of the queue, we need to use malloc () function.

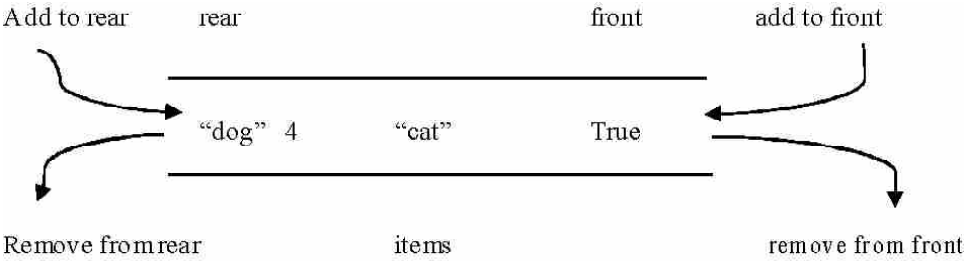
   [A] push [B] dequeue [C] peep [D] enqueue

7.5 D–Queue :

D–queue also called as double ended queue, is a data structure in which the element can be inserted or deleted either through the front end or rear end.

D–queue can be implemented and represented with a modified dynamic array or with a doubly linked list.

A Deque is a generalization way of both the FIFO Queue and the LIFO Queue (Stack). A Deque shows or represents a sequence of elements, with a front and a back. Elements can be added at the front of the sequence or the back of the sequence. The names of the Deque operations are self–explanatory :



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 [B] queue

   [C] circular queue [D] Deque

2. _____ is not a valid function for Deque.

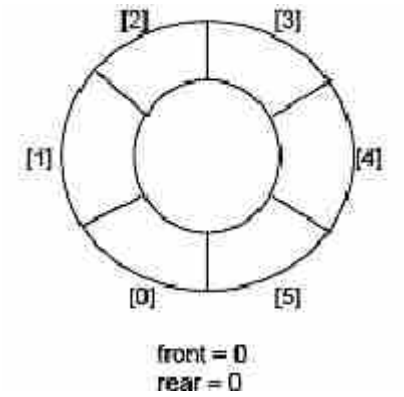
   [A] add_to_front [B] add_to_rear

   [C] enqueue [D] remove_from_front

7.6 Circular Queue :

In a circular queue, if the 'front' is equal to the 'rear', the circular queue will be considered as empty. If the circular queue is defined to contain n elements, then it can only support n–1 elements, as after insertion of the nth element, the rear will be equal to the front. So, we cannot say whether the queue is empty or full.

The given figure shows the representation of a circular queue, in which the value of the front and rear both are 0, if there is a single element.



[Circular Queue]

❖ **Algorithm for Insertion of an Element :**

1. If you see the rear is pointing to the last position, then you need go to step 2 or else step 3.
2. Then make the rear value as 0 (zero).
3. Increment the rear value by 1 (one).
4. In case the front points where the rear is pointing and also the queue holds a not NULL value for it, then in that case it is a queue overflow state, so quit, else go to step 5.
5. Enter or Insert the new value for the queue position pointed by the rear.

❖ **Algorithm for Deletion :**

(a) In case the queue is empty then display an Underflow, message and quit, else continue.

(b) Then delete the front element.

(c) In case the front is pointing to the last position of the queue then go to step 4 else step 5.

(d) Finally make the front point to the first position in the queue and quit.

(e) Increment the front position by one.

7.7 Applications of Queue :

Queue is an abstract data type which follows FIFO and because of this property queue is applicable in the following scenario. Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out (FIFO) order like Breadth First Search. This property of queue makes it useful in the following scenarios :

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.

❑ **Check Your Progress – 5 :**

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

   [A] CPU scheduling [B] Disk scheduling

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

2. _____ is a memory efficient version of queue.

   [A] stack [B] tree [C] graph [D] circular queue

7.8 Let Us Sum Up :

In this Unit we have focused on queue and types of queues. The queue data structure can be characterized by the fact that inducing or insertions are made at one end and at the same time deletions are made at another end. Further the first element of queue is called as a front element and the last element of queue is called as a rear element. In case of queues, we can insert an element from the rear end and we can delete an element from the front end.

The queue follows FIFO order, in which the element which is inserted first will be the first one to be 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 completely empty and we are trying to delete an element from it, then this condition is called as Underflow.

Further we have studied that queues can be implemented statically or dynamically i.e., as an array or as a linked list. There are two basic operations, add and delete, which can be performed on queues. As we have understood about the queue type one can conclude by noting following type of queues :

- **Circular Queue –** In circular queue, if the 'front' is equal to 'rear', the circular queue will be considered 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 elements can only be added to or removed from the front (head) or back (tail). 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.

7.9 Suggested Answers For Check Your Progress :

❑ **Check Your Progress 1 :**

1. [A] **2.** [B] **3.** [A]

❑ **Check Your Progress 2 :**

1. [D] **2.** [D]

❑ **Check Your Progress 3 :**

1. [D] **2.** [B] **3.** [D]

❑ **Check Your Progress 4 :**

1. [D] **2.** [C]

❑ **Check Your Progress 5 :**

1. [D] **2.** [D]

7.10 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).

7.11 Assignment :

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

7.12 Activity :

1. Watch the process of issuing tickets at the ticket window to understand a queue data structure.

7.13 Case Study :

- Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.

7.14 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.

BLOCK SUMMARY :

- Array is a linear data structure, used to store collection of homogeneous (same) type of data.
- Array stores data elements in contiguous memory location.
- Link–List is similar to Array. It is linear data structure which stores homogeneous collection of data.
- Difference between Array and Link–List is Array stores all data element in contiguous memory locations, whereas Lin–List stores all data elements in non– contiguous memory locations.
- To construct the Link–List we need to use dynamic memory allocation functions like malloc (), calloc (), and free ().
- Dynamic memory allocation functions allow user to create the variable (allocation of memory) at runtime (while the program is in running).
- Function malloc (), is used to create a single instance of the variable, function calloc (), is used to create an array (collection) of multiple instances at runtime.
- Function free (), is used to release the memory. It is used to delete the variable at run time, which is created by malloc () or calloc () function.
- There are three types of Link–Lists. [1] Singly Link–List, [2] Doubly Link–List and [3] Circular Link–List.
- Singly Link–List stores one data element and one address part. In the address part each node stores the address of the next node. Singly Link–List allow user to traverse in forward direction only. User cannot travers in reverse direction because the node structure does not store address of the previous node.
- The node of the Doubly Link–List has 3 parts, one data and two address parts. Two address parts (lptr and rptr) stores the address of the previous node and next node. Doubly Link–List allows user to travers in both (forward and reverse) directions.
- In the Circular Link–List we store the address of the first node in the address (next) part of the last node. This allows users to move to the first node after visiting last node.
- Stack is a linear data structure, which stores the data and returning to the user in LIFO (Last In First Out) manner.
- Stack data structure can be implemented using Array or Link–List.
- Stack data structure is useful in reversing the string, computing binary of integer number, to convert Infix expression to Postfix or Prefix, and also in the evaluation of Postfix expression.
- Calculator uses built–in stack circuitry, to evaluate Infix expression entered by the user.
- Stack is also a good replacement of recursion.

- Infix expressions are such expressions, in which we write all binary operators between two operands. In our daily life we use Infix expression. For example : 2 + 3 is infix expression.
- Prefix expressions also known as polish notations are such expressions, in which we write all binary operators first and the both operands. For example : + 2 3 is prefix expression.
- Postfix expressions also known as reverse polish notations are such expressions in which we write binary operators are written after the operands. For example : 2 3 + is a postfix expression. It is easy to develop a circuit, which can evaluate Postfix expression. Due to this reason calculator converts the Infix expression into the Postfix first using stack data structure and then evaluate Postfix expression in to the answer with the help of stack.
- Queue is a linear data structure, which stores the data element and return the value back to the user in FIFO (First In First Out) manner.
- Queue is implemented to schedule various processes to assign CPU to them.
- Queue can be implemented by using Array or Link–List.
- Circular queue is a kind of queue which utilise the memory more efficiently than queue data structure.

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.	4	5	6	7
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 Ahmedabad

BCAR-201/
DCAR-201

Data Structure Using C

BLOCK 3 : TREE AND GRAPHS

TREE AND GRAPHS

Block Introduction :

In Previous blocks we have studied linear data structures as Array, Stack, Queue and Linked List. In this block we will examine non–linear data structures such as Tree and Graph. Trees and Graph data structures have the ability to model real life problems and hence these data structures are important for modern programming.

Tree Data Structure is a non–linear data structure and it is represented in a hierarchical form. It is implemented by using a linked list which is best suitable. Hence, it becomes easier to do operations like insertion, searching and deletion as compared to linear data structure.

Graph is a collection of two sets i.e. set of vertices and set of edges that can also be implemented using non–linear data structure. These can be used in describing a wide range of relationships between objects. Such graphs allow real world situations to be represented as models, depicted by generalized structures.

Block Objectives :

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

- Basic terminologies of Tree
- Basic tree representation using array and linked list
- Different operations on trees
- Traversal of binary tree– Inorder, Preorder and Postorder
- Basic Terminologies, type and representation of Graph
- Graph traversal : BFS and DFS
- Shortest path algorithm : Kruskal's algorithm and Prim's Algorithm

Block Structure :

Unit
08

TREES

UNIT STRUCTURE

8.0 Learning Objectives :

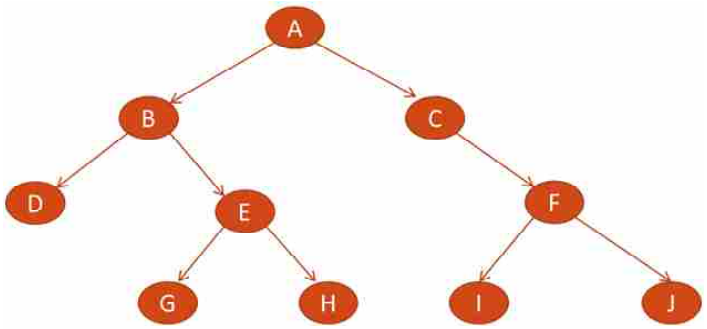
After learning this unit, you will be able to understand :

- Basic concepts and terminologies of Trees.
- Concepts of Binary Tree and Binary Search Trees.
- Representations of Trees using Arrays and Link–List.
- Applications of Trees.

8.1 Introduction :

In the last unit we have discussed about linked lists which is a method of dynamic memory allocation. In this unit we shall discuss about one of the most widely accepted data structures called trees. A tree is a widely used nonlinear data structure that represents a hierarchical structure (Parent – Child Relationship) with a set of connected nodes, such as the root node, the left child node and the right child node. It is also an acyclic connected graph in which each node has a zero or more children nodes and only one parent node. We shall also discuss the different representations of tree data structure.

Tree is a nonlinear data structure that contains nodes which are connected by directed edges. It contains one root node and its child nodes. Tree represents data in hierarchical form. Following Figure represents Tree Data structures.



Tree Data Structure

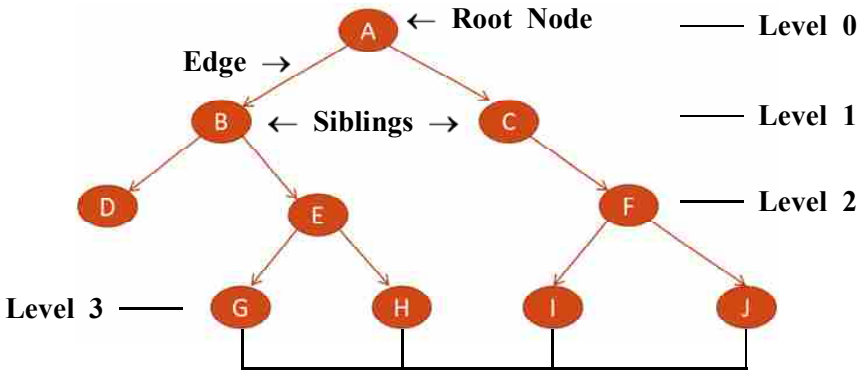
A, B, C, D, E, F, G, H, I and J are called the nodes and the lines that connects these nodes are called the edges. Node A is called the Root node of this tree. Let's understand the basic terminology of the tree.

8.2 Basic Terminology :

To understand the tree data structure, some of the basic terminologies should understand first. These terminologies are as below :

- **Root –** The very first node of the tree is called the root node. A special node which does not have any parent. In degree of root node is 0. In the above tree, node A is Root node because it has no parent node.
- **Parent and Child Node –** If there is an edge (X,Y) between nodes X and Y then X is known as Parents (Father) node of Y. The node Y is known as Child node of X. The child will have only once parent node. In Example B & C are child of A.
- **Degree of a node –** Total number of sub trees a node have. The Degree of node A is 2 because it has two sub trees.
- **Degree of a tree –** The maximum number of nodes a tree contains. The degree of the above tree is 10 because it contains 10 nodes.
- **Indegree –** number of branches terminated at given node. In degree of all the nodes except root node is 1 in above tree
- **Outdegree –** It is the number of branches emerging from a given node. In above tree Outdegree of nodes A, B, E, F is 2. Out degree of node C is 1. Outdegree of node D, G, H, I and J is 0 because they do not have any child.
- **Leaf (Leaves) Node –** The node having Outdegree 0 is known as Leaf node. In Simple words the node having no child node is called leaf node. In the given tree, D, G, H, I, J are leaf nodes. Leaf nodes are also known as External nodes in tree.
- **Internal Nodes –** All the nodes of the tree which have child are known as Internal Nodes. B, C, E, F are Internal Nodes of the given tree.
- **Siblings –** The nodes which are the children of the same parents. G and H are siblings because their parent is same.
- **Ancestors of a node –** all the nodes along the path from the root node to the node are called ancestors of that node. Ancestors of node J are F, C and A, because they fall in the path of root node to the node J.
- **Descendants of a node –** all the nodes that are in subtrees of a given node are called descendants of that node. Descendants of node C, are F, I and J.

- **Level of a node –** it represents the number of connections between the node and the root. Root node has level 0. B & C are at level 1 and so on. G, H, I and J are on level 3.
- **Height of a tree –** Height of the tree is number of edges from the root node to its most distant leaf node. In simple terms, the maximum level number of the node in the tree is called the height or depth of the tree. The Height of the given tree is 3 because the maximum level number is 3 or the number of edges between the root node to the deepest nodes G, H, I and J are 3. Hence height of this tree is 3.
- **Depth of node –** The depth of a node is the number of edges from the node to the root node. Means one need to traverse that number of edges if he wants to visit that node from the root node. The depth of E is 2. Depth of G is 3.
- Following figure has shown all the important terminologies



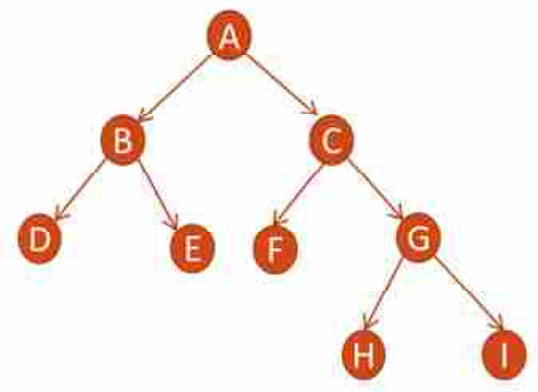
Leaf Nodes

❖ **Applications of Tree :**

- Trees are used to represent Computer File Systems.
- Syntax tree is used by compilers to check the syntax of the program we write.
- Trees are used in many popular databases to implement indexing.
- Trees are used to store data in routing tables in routers.

❑ **Check Your Progress – 1 :**

1. Calculate the height of the following tree.
2. Find Root node, leaf nodes, and siblings of the following tree.
3. Find the Ancestors of a node I.



8.3 Binary Tree :

In previous section we have discussed the terminologies of basic trees. Now it's time to learn the concept of the special type of tree called binary tree. A Binary Tree is a special type of tree in which each node has at most (maximum) two children. In simple words a binary tree is a tree in which each of its nodes has either 0 or 1 or 2 child. In General Tree it may possible that the tree has more than 2 children. Following figures shows the difference between general and binary tree. Fig. A is a Binary Tree where Fig. B is General Tree.

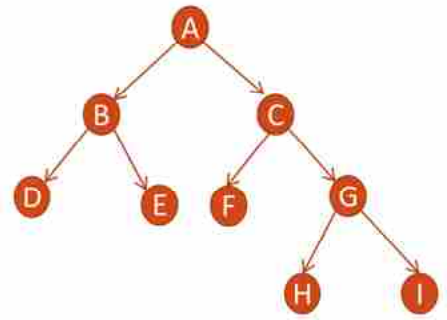


Fig. A – Binary Tree

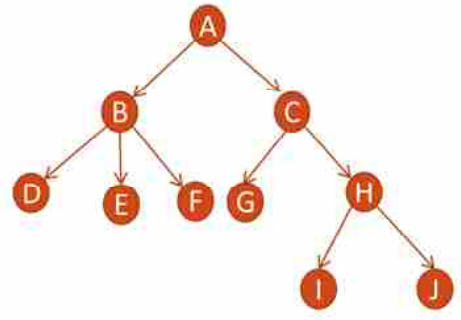
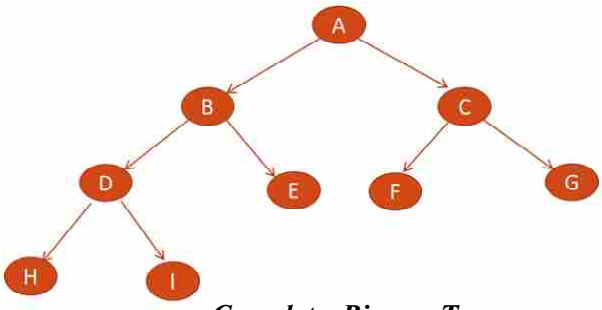


Fig. B – General Tree

Fig. A is Binary tree because each of its nodes has at most 2 children but Fig. B is not considered as Binary Tree because node B has more than 2 children. So it is known as basic or general tree.

❖ Complete Binary Tree :

A Complete Binary tree is a tree in which each level (except last level) must be completed (full), before we start a new level. Also note that all the nodes should be filled from the left.



Complete Binary Tree

Above tree is complete binary tree because except the last level each level has full nodes and at last level the nodes are filed from left.

❑ Check Your Progress – 2 :

1. What is the difference between general and binary tree ?
2. Define Complete Binary Tree.

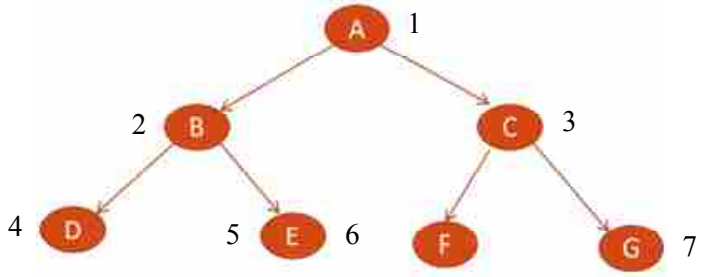
8.4 Binary Tree Representation using Array and Link–List :

We will learn two ways to represent binary tree :

1. Array Representation
2. Linked List Representation

8.4.1 Array Representation :

In array representation of the binary tree, an index array is used to represent the nodes of the tree. Let's represent the following tree using array.



To represent the binary tree using array, each nodes of the tree are assigned an index value starting from root node. Root node is stored at array index 1. All the nodes of next level are stored at subsequent index and so on. The array representation the above tree is as below :

A	B	C	D	E	F	G
1	**2**	**3**	**4**	**5**	**6**	**7**

Array Representation of the Tree

Root node A is stored at index 1. Node B and C are in the same levels and hence stored in index 2 and 3 subsequently. Next D, E, F and G are on next level and hence stored at 4, 5, 6 and 7th index.

In this type of implementation, the location of left child of a particular node can be identified using the following formula :

If the node is at Ith location then

Left child of that node would be at **2 * I** location.

Right child of that node would be at **(2 * I) + 1** location.

Suppose we want to find out the left and right child of node C. Node C is stored at 3rd location so I=3.

Left child of node C is stored at : 2 * 3 = 6th Location.

Right child of node C is stored at : (2 * 3) + 1 = 7th Location.

We can verify that left child of the node C is node F which is stored at 6th location in the array and right child of node C which is node G is stored at location 7 in the array representation.

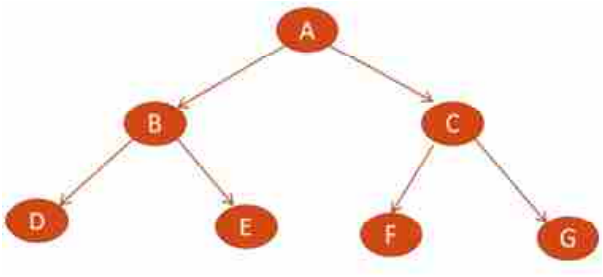
8.4.2 Link–List Representation :

A binary tree can be represented using link–list. Doubly link–list is used to represent binary tree, the node structure of the doubly linked list would be like :

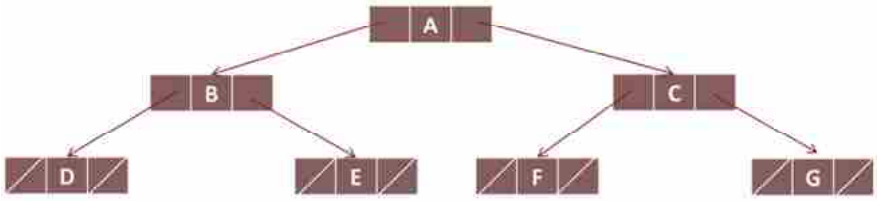
LPTR	Info	RPTR

Node Structure of Doubly Link–List

Where LPTR is the pointer used to store the memory address of the left sub tree (left child) of the node and RPTR is the pointer used to store the memory address of the right sub tree (right child) of the node. Info part is used to store the data.



The Linked Representation of the above Tree would be :

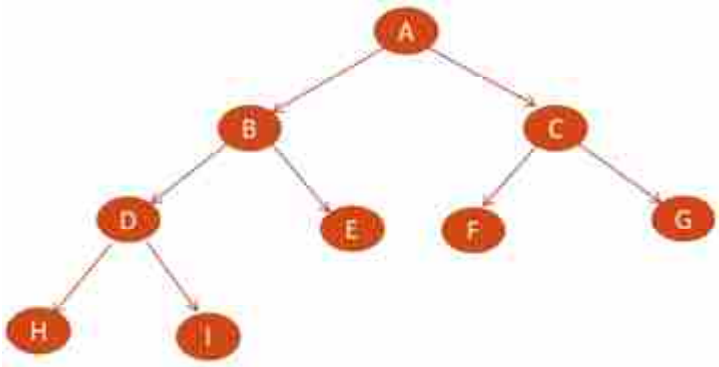


Linked Representation of Binary Tree

In the above figure we can see that A is the root node and in LPTR of A, the address of its left child B is stored, similarly in RPTR of A, the address of its right child is stored. By following LPTR and RPTR links, one can traverse the nodes of the binary tree. If the node has no left child then the LPTR would be NULL. Similarly if the node has no right child in that case RPTR would be NULL and if the node is leaf node then in that case both LPRT and RPTR would be NULL.

❑ **Check Your Progress – 3 :**

1. Represent the following tree using Array and Link–List.



2. The node in the tree having indegree 0 is called ?

   (A) Leaf Node (B) Isolated Node

   (C) External Node (D) Root Node

3. In which of this tree the value of left children are less than the value of its parent node and the value of right children are greater than the value of its parent ?

   (A) Complete Binary Tree (B) Binary Search Tree

   (C) Threaded Tree (D) Full Binary Tree

4. If the parent node in the binary tree is stored in ith location then at which position its left child is stored ?

   (A) 2 * i (B) 2 * i + 1 (C) 2 * i – 1 (D) 2 * i + 2

5. Which of the following type best represents tree using link–list ?

(A) Single Link–List (B) Single Circular Link–List

(C) Doubly Link–List (D) All of above

6. Which of the following are the applications of the tree data structure ?

(A) Representation of Computer File System

(B) Syntax Tree in compiler

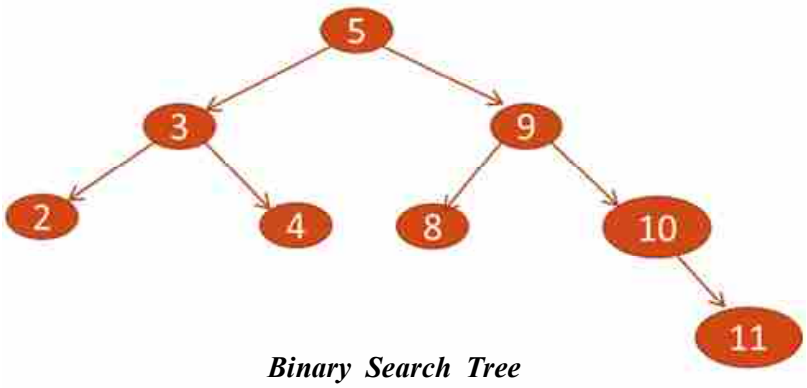
(C) Indexing in Database

(D) All of Above

8.5 Binary Search Tree :

A binary tree is called Binary Search Tree, if each node N of Tree must satisfies following property,

The value of node N is greater than every value in the left sub–tree of N node and it is less then every value in the right sub–tree of N node. In Simple words Binary Search Tree also called as BST or ordered tree is a binary tree in which each value of the left subtree of the node is less than the value of the node and each value of the right subtree of the node is greater than the value of the node. Example :



Binary Search Tree

We can see that for each node of the above tree, the values of its left child is smaller than the value of the node and the value of the right child is greater than the value of the node.

8.6 Lets Us Sum Up :

Tree is a nonlinear data structure that contains nodes which are connected by directed edges. It contains one root node and its child nodes. Tree represents data in hierarchical form. Trees are normally used to represent computer file system. It is also used in compilers and routers. Binary trees are the special types of general trees where each node has at most 2 children.

Binary trees can be represented using Array representation or Linked List representation. Doubly Linked list is used to represent binary tree where the node has three parts. LPTR, Info and RPTR to store the address of left child, data and the address of the right child respectively.

A Complete Binary tree is a tree in which each level (except last level) must be completed (full), before we start a new level. A binary tree is called Binary Search Tree or BST, if each node N of Tree must satisfies following property, the value of node N is greater than every value in the left sub–tree of N node and it is less than every value in the right sub–tree of N node.

8.7 Suggested Answer for Check Your Progress :

- **Check Your Progress 1 :**

  See sec 8.2

- **Check Your Progress 2 :**

  See sec 8.3

- **Check Your Progress 3 :**

  1. See sec 8.4 **2.** D **3.** B

  4. A **5.** C **6.** D

8.8 Glossary :

1. **Root –** The very first node of the tree is called the root node. A special node which does not have any parent.
2. **Parent and Child Node –** If there is an edge (X, Y) between nodes X and Y then X is known as Parents (Father) node of Y. The node Y is known as Child node of X. The child will have only once parent node.
3. **Degree of a node –** Total number of sub trees a node have.
4. **Degree of a tree –** The maximum number of nodes a tree contains.
5. **Indegree –** number of branches terminated at given node.
6. **Outdegree –** It is the number of branches emerging from a given node.
7. **Leaf (Leaves) Node –** The node having Outdegree 0 is known as Leaf node. In Simple words the node having no child node is called leaf node.
8. **Internal Nodes –** All the nodes of the tree which have child are known as Internal Nodes.
9. **Siblings –** The nodes which are the children of the same parents.
10. **Ancestors of a node –** all the nodes along the path from the root node to the node are called ancestors of that node.
11. **Descendants of a node –** all the nodes that are in subtrees of a given node are called descendants of that node.
12. **Level of a node –** it represents the number of connections between the node and the root. Root node has level 0.
13. **Height of a tree –** Height of the tree is number of edges from the root node to its most distant leaf node. In simple terms, the maximum level number of the node in the tree is called the height or depth of the tree.
14. **Depth of node –** The depth of a node is the number of edges from the node to the root node. Means one need to traverse that number of edges of he wants to visit that node from the root node.
15. **Binary Tree –** A Binary Tree is a special type of tree in which each node has at most two children.
16. **Complete Binary Tree –** A Complete Binary tree is a tree in which each level (except last level) must be completed (full), before we start a new level.
17. **Binary Search Tree –** A binary tree is called Binary Search Tree or BST, if each node N of Tree must satisfies following property, the value

of node N is greater than every value in the left sub–tree of N node and it is less than every value in the right sub–tree of N node.

8.9 Assignment :

1. How to calculate the height and depth of the tree ?
2. What are the applications of Tree ?
3. What characteristics make binary tree a Binary Search Tree ?

8.10 Activities :

1. Write a C program to display all the leaf nodes in tree.
2. Write a C program to calculate total number of nodes of the given binary tree.

8.11 Case Study :

An address book contains the name and other details of employee's. The Admin wants to search for a particular name and find out the information related to it. Data shall keep updating after each operation (Insertion, Deletion). Formulate the problem and write a C program with appropriate data structures and algorithms.

8.12 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 Sahni.
4. Data Structures : An Advanced Approach Using 'C' by Esakov and Weises.
5. Data Structures and 'C' Program by Christopher J. Van Wyk
6. An Introduction to Data Structures with Applications by Tremblay and Sorenson.

Unit 09

OPERATIONS ON BINARY TREE

UNIT STRUCTURE

9.0 Learning Objectives :

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

- Different operations on Binary Search Tree.
- Inorder, Preorder and Postorder Traversals of Binary Tree.
- Recursive Algorithms of Binary Tree Traversals

9.1 Introduction :

In previous unit we understood the concept of binary tree and binary search tree. Now it is essential to know which kind of operations we can perform on binary search tree. As we discussed that binary search tree is an ordered tree where the left sub–tree of the node has value less than it and the right sub–tree of the node has value greater than the value of node. We can perform insert, delete as well as search operations on the Binary Search Tree. In following section we will discuss these operations with example.

9.2 Operations on Binary Search Tree :

As we discuss we can perform following operations on the Binary Search Tree :

- Insert – Inserting a new node in to BST
- Delete – Deleting a node from BST
- Search – Searching a node from BST

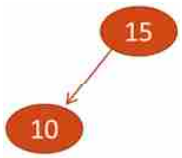
❖ **Insert Operation :**

To insert an element in the binary search tree, first finds its proper position in the tree. Start searching from the root node if the value of the new node is less than the value of the root node then the new node will be inserted in the left subtree and if the value of the new node is greater than the value of the root node then the new node will be inserted in the right subtree.

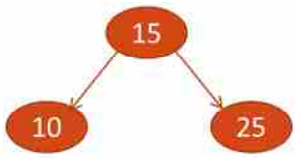
Let's understand the same by inserting multiple elements in the binary search tree.

– Insertion of element 15. As the tree is empty, element 15 is inserted and becomes the root node.

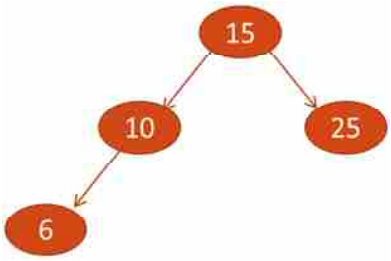
– Insertion of element 10. Compare the value 10 with the value of the root node which is 15. 10 is less than 15 so element 10 will be inserted at the left sub–tree. Left sub–tree is empty so 10 will be inserted as a left child of 15.



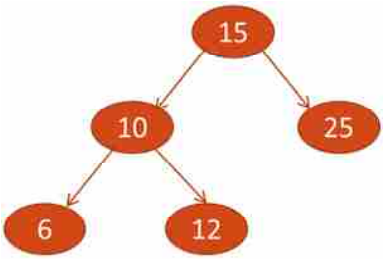
– Insertion of element 25. Again start form the root node and compare. Element 25 is greater than the root node element 15, so it will be inserted as right child.



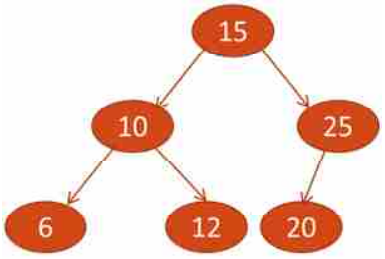
– Insertion of element 6. Compare element 6 with the element of the root node which is 15. Element 6 is less than 15 so it will be inserted at the left sub–tree. Here left sub–tree is not empty and has left child 10. So element 6 will be compared with the 10 and it is less than 10 so it will be inserted at the left child of 10.



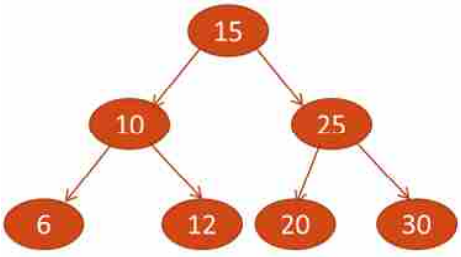
– Insertion of Element 12. Compare it with 15, it is less than 15 so will be inserted at the left sub–tree. Now at left sub–tree 10 is present so 12 is compared with 10 and is greater than 10 so it will be inserted at the right side of 10.



– Insertion of element 20. It is greater than 15 so will be inserted at right side. At right side 25 is present so 20 is compared with 25 and is less than it so 20 will be inserted at the left side of 25.



– Insertion of element 30. It is greater than 15 and hence should be inserted at the right side. At right position 25 is present so 30 is compared with 25 and found greater that it so will be inserted at the right of 25.



By this way the elements are added in the binary search tree. If you look at any node of this tree you can see that all the nodes in the left subtree have values less than the value of the node and all the nodes in the right subtree have values greater than the value of the node.

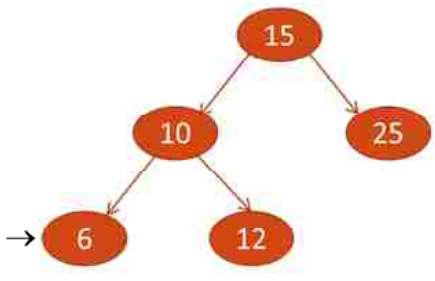
❖ **Search Operation :**

When you want to search any element in Binary Search Tree, Start Searching with root node. Compare the value of root node with search element, if both are same then element is found and search is complete. Else if the search value is less than the value of root node than search the element in the left subtree and if the value is greater than the value of the root node than search the element in the right subtree. Follow the same procedure for each node until you found the element.

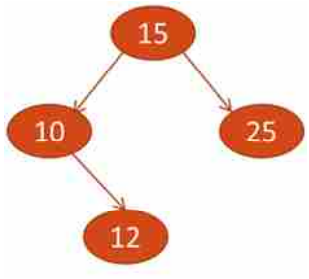
❖ **Delete Operation :**

There are three cases of Delete Operations in Binary Search Tree.

– **Case 1 : Deletion of leaf node.** We want to delete leaf node (a node with no child). As it has no child so it will be directly removed from the tree. Suppose we want to delete node 6 from the following tree than the tree will look like :

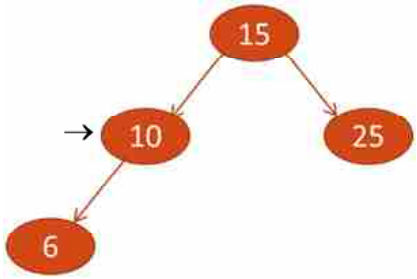


Before

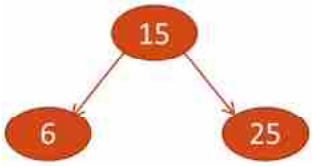


After

– **Case 2 : Deletion of node which has either left or right sub–tree.** If a node to be deleted has only one left sub–tree or right sub–tree then the sub tree of the deleted node is linked directly with the parent of the deleted node. Suppose we want to delete node 10 from the following tree. After deletion the tree will look like :



Before



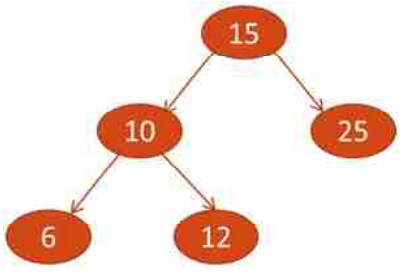
After

– **Case 3 : Deletion of node having both left and right sub tree :** If the node to be deleted has both left and right sub–tree then in that case follow these steps :

1. find inorder successor of the deleted node
2. append the right sub–tree of the inorder successor to its grand parent
3. replace the node to be deleted with inorder successor

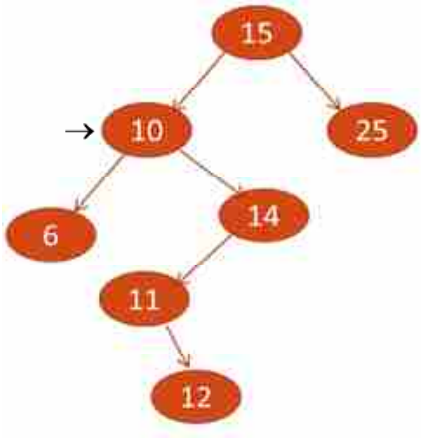
To find the inorder successor of the deleted node we need to traverse the tree in inorder. Inorder tree traversal is performed using following three steps :

- Traverse the left subtree in Inorder (Recursive Process)
- Visit the root node.
- Traverse the right subtree in Inorder (Recursive Process)



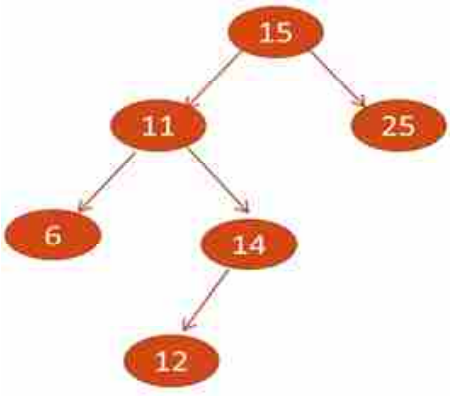
Inorder Traversal of the above tree is : 6, 10, 12, 15, 25. We will learn inorder traversal in detail in upcoming section.

In Case 3 we want to delete a node having both left and right sub–tree. Suppose in given tree we want to delete node 10.



Before

Here we need to delete node 10. So first we need to find the inorder successor of the node 10. Inorder traversal of this tree is 6, **10**, 11, 12, 14, 15, 25. So the inorder successor of the node 10 is node 11. So node 10 will be replaced by node 11 and the right subtree of node 11 which is node 12 will be appended with its grandparent 14. Hence after deletion of node 10 the Binary Search tree would be like :



After

❑ **Check Your Progress – 1 :**

1. Construct Binary Search Tree by adding the elements 50, 70, 40, 35, 45, 65, 80, 42.
2. How will the tree looks like after deletion of element 70 ?

9.3 Binary Tree Traversals :

The method of visiting each node of a tree is called a Tree Traversal. There are three types of tree traversal methods :

– Inorder Traversal

– Preorder Traversal

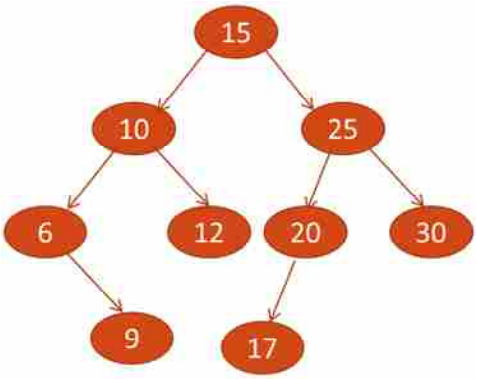
– Postorder Traversal

9.3.1 Inorder Traversal :

In case of Inorder tree traversal method, the left node/sub–tree of the tree is visited first, then the root node and finally the right node or right sub–tree is visited **(Left, Root, Right)**, which can be stated in steps as given below :

- Traverse the left sub tree in Inorder (Recursive Process)
- Visit the root node.
- Traverse the right sub tree in Inorder (Recursive Process)

Let's travers the given tree in inorder fashion. First it starts with root node. Traverse the left sub–tree until you found a node which has no left child. Visit that node and then traverse the right sub–tree of that node and repeat these 3 steps for each node of the tree recursively.



Inorder Traversal : 6, 9, 10, 12, 15, 17, 20, 25, 30

9.3.2 Preorder Traversal :

In the preorder tree traversal method, the root node is visited first, then the left sub–tree is visited and finally the right subtree is visited. **(Root, Left, Right)**. The steps are given below :

- Visit the root node.
- Traverse the left subtree in Preorder. (Recursive Process)
- Traverse the right subtree in Preorder (Recursive Process)

If we traverse the above tree in preorder then we got :

Preorder Traversal : 15, 10, 6, 9, 12, 25, 20, 17, 30

9.3.3 Postorder Traversal :

In the postorder traversal method, first the left subtree is visited, then the right subtree is visited and then the root node is visited **(Left, Right, Root)**. The steps for the postorder traversal are :

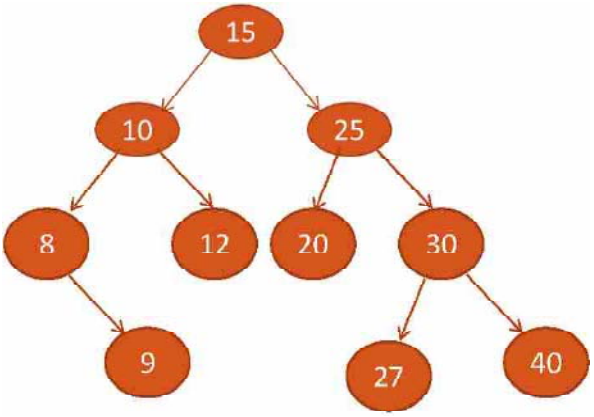
- Traverse the left subtree in Postorder (Recursive Process)
- Traverse the right subtree in Postorder (Recursive Process)
- Visit the root node.

The postorder traversal of the above tree would be :

Postorder Traversal : 9, 6, 12, 10, 17, 20, 30, 25, 15

❑ **Check Your Progress – 2 :**

1. Find inorder, preorder and postorder traversals of the following BST ?



9.4 Recursive Algorithms for Inorder, Preorder and Postorder :

In previous section we have seen the inorder, preorder and postorder traversals methods of the binary search tree. Now let's understand the recursive algorithms for binary tree traversals.

❖ **Recursive Algorithm for Inorder Traversal :**

INORDER(PT)

INORDER is recursive function which is used to traverse the tree in inorder. Here PT is the pointer that points to the root node of the tree. LPTR points the left child and RPTR points the right child.

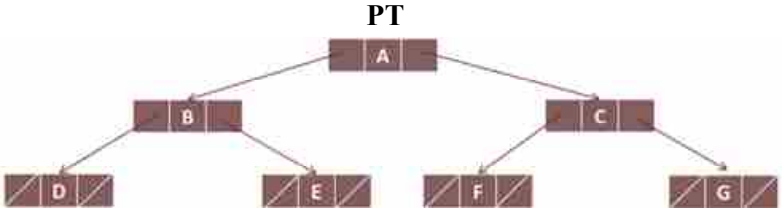
STEP 1 : [Check if tree is empty ?]
If PT=NULL then
Write "Tree is empty"
Return

STEP 2 : [Traverse the left sub tree in recursive fashion]
If LPTR(PT) != NULL then
CALL INORDER (LPTR(PT))

STEP 3 : [Visit the root node]
Write INFO(PT)

STEP 4 : [Traverse the right sub tree in recursive fashion]
If RPTR(PT) != NULL then
CALL INORDER (RPTR(PT))

STEP 5 : Return



Inorder : D, B, E, A, F, C, G

❖ **Recursive Algorithm for Preorder Traversal :**

PREORDER(PT)

PREORDER is recursive function which is used to traverse the tree in preorder. Here PT is the pointer that points to the root node of the tree.

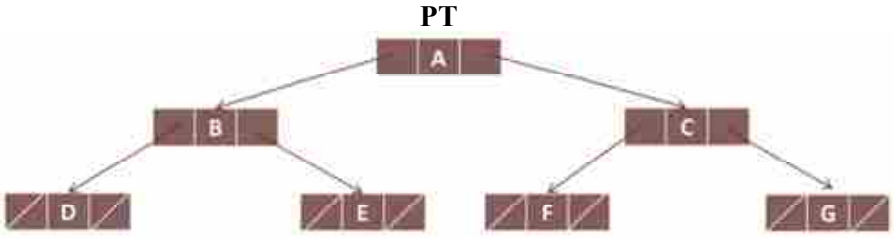
STEP 1 : [Check if tree is empty ?]
If PT=NULL then
Write "Tree is empty"
Return

STEP 2 : [Visit the root node]
Write INFO(PT)

STEP 3 : [Traverse the left sub tree in recursive fashion]
If LPTR(PT) != NULL then
CALL PREORDER (LPTR(PT))

STEP 4 : [Traverse the right sub tree in recursive fashion]
If RPTR(PT) != NULL then
CALL PREORDER (RPTR(PT))

STEP 5 : Return



Preorder : A, B, D, E, C, F, G

❖ **Recursive Algorithm for Postorder Traversal :**

POSTORDER(PT)

POSTORDER is recursive function which is used to traverse the tree in postorder. Here PT is the pointer that points to the root node of the tree.

STEP 1 : [Check if tree is empty ?]
If PT=NULL then
Write "Tree is empty"
Return

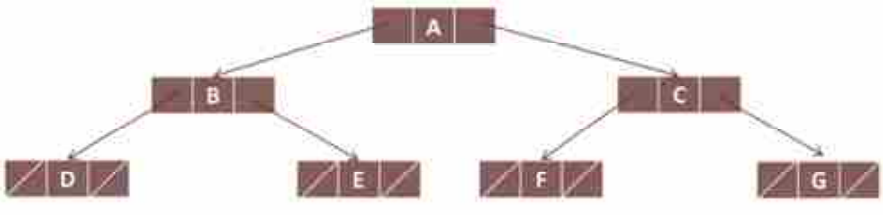
STEP 2 : [Traverse the left sub tree in recursive fashion]
If LPTR(PT) != NULL then
CALL POSTORDER (LPTR(PT))

STEP 3 : [Traverse the right sub tree in recursive fashion]
If RPTR(PT) != NULL then
CALL POSTORDER (RPTR(PT))

STEP 4 : [Visit the root node]
Write INFO(PT)

STEP 5 : Return

PT



Postorder : D, E, B, F, G, C, A

❑ **Check Your Progress – 3 :**

1. Write recursive algorithms for inorder, preorder and postorder traversals.
2. If you want to delete the node having both left and right subtree in Binary Search Tree then which of the following node will replace the deleted node ?

   (A) Inorder successor (B) Inorder predecessor

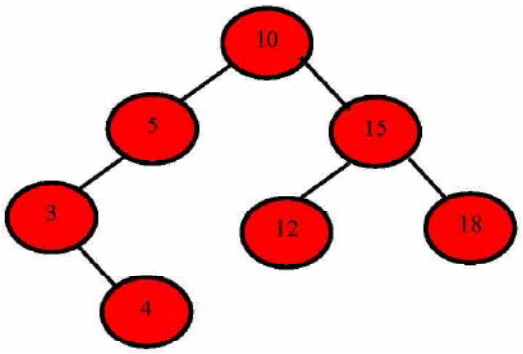
   (C) Preorder successor (D) Postorder successor
3. Which of the following is a right sequence to be followed in post order traversal of binary tree ?

   (A) Left, Root, Right (B) Left, Right, Root

   (C) Root, Left, Right (D) Left, Right, Left
4. Which of the following traversal technique in Binary Search Tree would print the numbers in ascending order ?

   (A) Postorder (B) Preorder

   (C) Inorder (D) None of the above
5. What will be the preorder traversal of this BST ?



   (A) 3, 4, 5, 10, 12, 15, 18 (B) 4, 3, 5, 12, 18, 15, 10

   (C) 10, 5, 3, 4, 12, 18, 15 (D) 10, 5, 3, 4, 15, 12, 18
6. What will be the postorder traversal of the above tree ?

   (A) 3, 4, 5, 10, 12, 15, 18 (B) 4, 3, 5, 12, 18, 15, 10

   (C) 10, 5, 3, 4, 12, 18, 15 (D) 10, 5, 3, 4, 15, 12, 18

9.5 Let Us Sum Up :

In this section we have understood the insert, delete and search operations of binary search tree. Insertion of new element requires finding out its proper position on the Binary Search Tree. This is done by start comparing the newly inserted element with the element of root node. If the new element is less than the element of root node than it will be inserted at the left subtree else it will be inserted at the right subtree.

Searching of element also requires comparing the value with the value of root node and then either on left subtree or in right subtree based on its value.

Deletion of element requires three criteria's to check. If the element to be inserted is a leaf node than it will directly deleted. If it has left or right subtree then the sub tree of the deleted node is linked directly with the parent of the deleted node. If the deleted node has both left and right subtree then the inorder successor of the deleted node will replace the deleted node and any right subtree of inorder successor will be appended to its grandparent.

Vising any node in the binary tree is known as tree traversal. There are there tree traversals techniques :

– Inorder – (Left, Root, Right)

– Preorder – (Root, Left, Right)

– Postorder – (Left, Right, Root)

We can implement these traversals using recursive algorithms. For that we need to represent tree as a doubly linked list.

9.6 Suggested Answer for Check Your Progress :

❑ **Check Your Progress 1 :**

Refer Section 9.2

❑ **Check Your Progress 2 :**

Inorder : 8, 9, 10, 12, 15, 20, 25, 27, 30, 40

Preorder : 15, 10, 8, 9, 12, 25, 20, 30, 27, 40

Postorder : 9, 8, 12, 10, 20, 27, 40, 30, 25, 15

❑ **Check Your Progress 3 :**

1. Refer Sec 9.4 **2.** A **3.** B

4. C **5.** D **6.** B

9.7 Glossary :

1. Inorder – This is one of the techniques to traverse a tree, which traverses to Left Subtree first then displays the elements and then traverses to Right Subtree.

2. Preorder – This is also one of the techniques to traverse a binary tree, where first the element is displayed then left subtree is traversed and then the right subtree is traversed.

3. Postorder – Another tree traversal technique where we traverse first to the left subtree and then traverse to right subtree and finally display the element.

9.8 Assignment :

1. How the elements are inserted in the binary search tree ?
2. How can you delete an element from the binary search tree where the deleted node has both left and right subtree ? Give example.
3. Explain different recursive tree traversing techniques.

9.9 Activities :

1. Write down C program to insert and delete element from Binary Search Tree
2. Implement recursive C language functions for inorder, preorder and postorder traversals based on the recursive algorithms.

9.10 Case Study :

List out the application areas of inorder, preorder and postorder traversals in computer science. Understand how different tree traversal techniques are used in such areas effectively.

9.11 Further Readings :

1. An Introduction to Data Structures with Applications by Tremblay and Sorenson.
2. Data Structures Using "C" by Tanenbaum.
3. Data Structures and Program Design in "C" by Robert L. Kruse.
4. Fundamentals of Data Structures by Horowitz and Sahni.
5. Data Structures : An Advanced Approach Using 'C' by Esakov and Weises.
6. Data Structures and 'C' Program by Christopher J. Van Wyk.

Unit
10

GRAPHS

UNIT STRUCTURE

10.0 Learning Objectives :

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

- The concept of Graphs.
- The different types of Graphs.
- The methods of representations of Graphs.
- The Graph traversal methods.
- The Shortest path algorithms.

10.1 Introduction :

In previous unit we studied binary tree data structures, which provide a useful way of storing data for efficient searching. In Binary tree each node can have at most two children however more general tree structures can be created in which each node can have more numbers of child nodes. In this unit we will be discussing about another type of non–linear data structure called graphs and its method of representation.

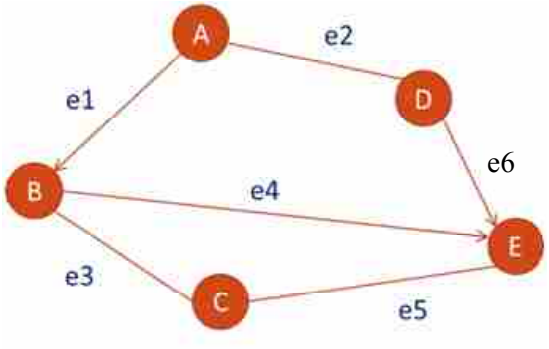
We will also be discussing about the different shortest path algorithms which are used to traverse a graph from a source node to a destination node.

10.2 Definition :

Graph is a nonlinear data structure that contains nodes which are connected by edges.

A graph is a collection of nodes called vertices, and the connections (links) between them called as edges.

A graph G, G = (V, E) comprises of a nonempty set V called the set of nodes or vertices, set E which is the set of edges and a mapping from the set of edges E to a set of pairs of element V.



Graph

Above figure represents the graph data structure. Where A, B, C, D, E are the nodes of the graph and e1, e2, e3, e4, e5, e6 are the edges of the graphs. We can see that these nodes are connected with edges.

Graphs are the important nonlinear data structures used to define flow of computation in computer science.

Graph theory is used to find shortest path in networks.

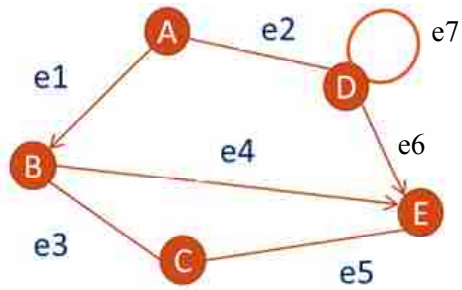
Graphs are used in Google maps to represent locations as vertices and roads connecting these locations as edges. Shortest path between two nodes is calculated based on this representation.

Social Media Platform like Facebook use graph theory to represent users and their friends.

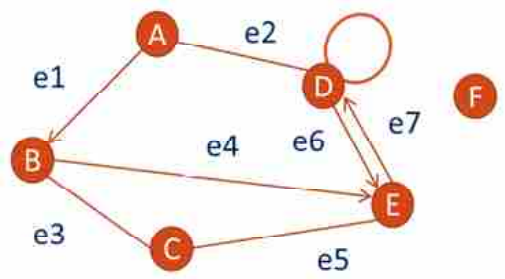
10.3 Terminology :

To understand the graph data structures, it is essential to learn some of the basic terminologies. These terminologies are :

- **Adjacent Nodes –** Two nodes of the graph which are directly connected by an edge are called adjacent nodes. In the above graph A & B, A & D etc. are called adjacent nodes.
- **Directed Edges –** An edge which has direction and which is directed from one node to another node in graph is called directed edge. e1, e4 and e6 are directed edges. Directions are given for source node to the destination node. (Origin and Termination)
- **Undirected Edges –** An edge in a graph which has no specific direction is known as undirected edge. e2, e3, e5 are undirected edges.
- **Incident –** An edge which joins two vertices of the graph is called incident to that nodes. e2 is incident to nodes A & D.
- **Loop –** An edge of the graph which connects a node itself is called loop. For example in following graph, edge e7 is called the loop.

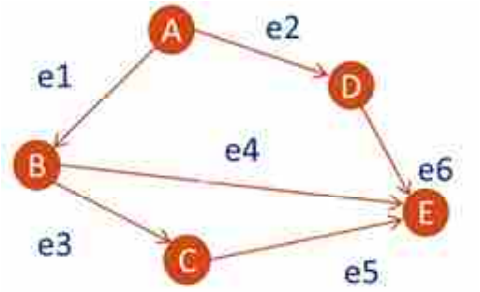


– **Parallel Edges –** In a graph, when pair of nodes are connected by more than one edge then such edges are called parallel edges. e6 & e7 are called parallel edges in the following graph.



– **Isolated Node –** A node which is not connected (adjacent) to any other node in the graph is called isolated node. Ex : in above graph node F is called isolated node because it is not connected to any node of the graph.

– **In degree & Out Degree –** In a directed graph (a graph in which all the edges are directed), number of edges incoming (terminating) to the node is called In Degree. Similarly, the number of edges outgoing (initiating) from the node is called Out Degree. In the following graph In degree of node B is 1 and out degree is 2. In degree of node E is 3 and Out degree 0

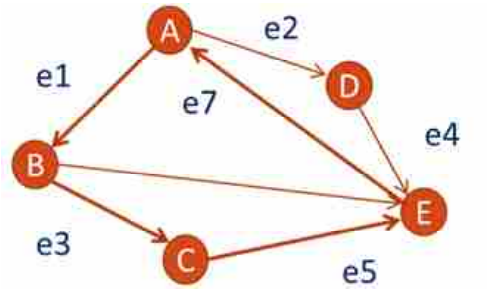


– **Total Degree of a node –** The sum of in degree and out degree for a node is called the total degree of that node. In above graph the total degree of node B is 3 (2 Out Degree + 1 In degree).

– **Path –** It is a sequence of edges that connects sequence of vertices. In the above graph Path from A to E is P1={A,B,C,E}

– **Simple Path –** A path in a graph, in which all the edges and vertices are distinct are called simple path. Ex. Path P1.

– **Cycle –** A Path in the graph that originates and ends to the same node or vertices is called a cycle. For example in below graph path P1={A,B,C,E,A} is cycle because it originates and terminates to the same vertex A.

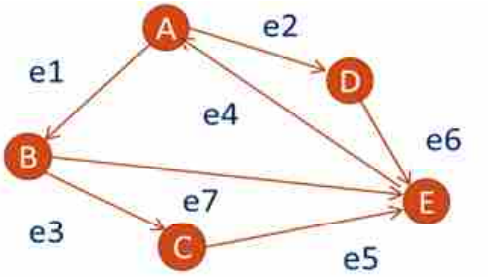


❑ **Check Your Progress – 1 :**

1. What is Graph ? What are the applications of the graph ?
2. Define the following terminologies : Directed Edge, Isolated node, Path, Cycle

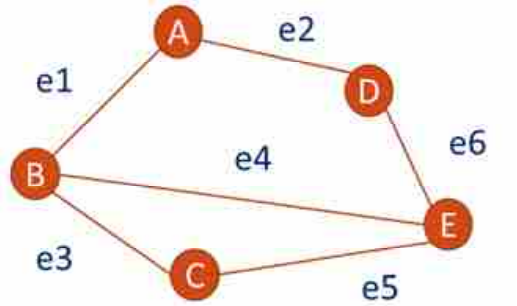
10.4 Types and Representation of a Graph :

– **Directed Graph :** A graph in which all the edges are directed is known as Directed Graph. Following graph is an example of directed graph because all the edges of this graph are directed.



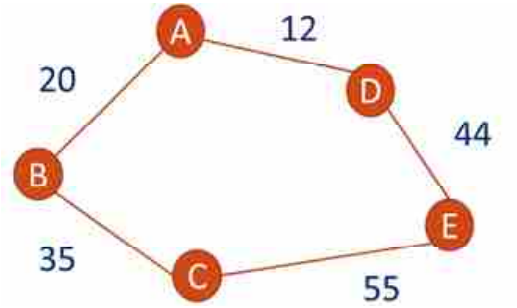
Directed Graph

– **Undirected Graph :** A graph in which all the edges are not directed is called Undirected Graph. Following graph is undirected graph.



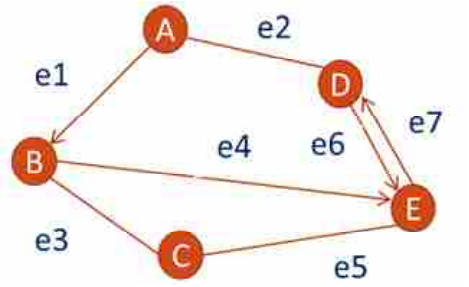
Undirected Graph

– **Weighted Graph :** A graph in which weights are assigned to all the edges of the graph is called weighted graph. Following graph is an example of weighted graph because weight is assigned to each edge of the graph. Such graph is used to display the distance between cities where nodes are cities, edges are roads and weights represents distance in KM.



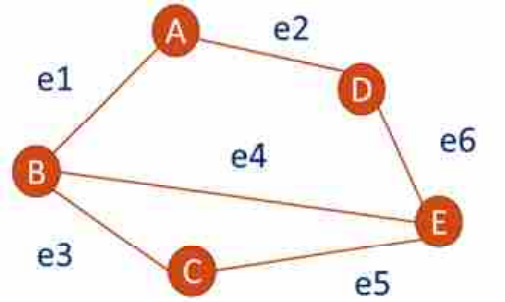
Weighted Graph

– **Multi Graph :** A graph having some parallel edges is called Multi Graph.



Multi Graph

– **Simple Graph :** A graph in which no more than one edge exists between a pair of nodes is called a Simple Graph. In other words a graph having no parallel edges is called Simple Graph.



Simple Graph

❖ **Representations of Graph :**

It is a technique to store graph into computer memory. There are two different methods for the representation of graphs :

1. Adjacency Matrix Representation
2. Adjacency List Representation

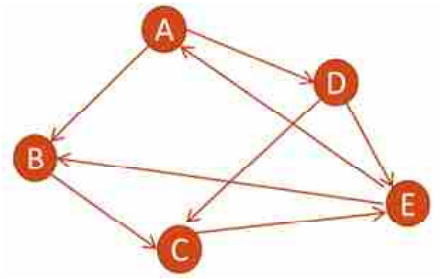
❖ **Adjacency Matrix Representation of Graph :**

In this type of representation, matrix is used to represent graph. Rows and Columns of the matrix represent the source and destination vertices and entries in graph indicate whether an edge exists between the vertices or not ? It's a two dimensional array having size V × V where V represents the vertices or nodes of the graph.

Suppose adjc[ ][ ] is a adjacency matrix then adjc[i][j]=1 indicates that there is a direct edge between vertices i and j. The value 0 indicates that there is no direct edge between vertices i and j means node i and j are not directly connected by an edge. It can be represented as

$$Adjc_{ij} = \begin{cases} 1 \text{ if there is a direct edge between nodes i and j} \\ 0 \text{ otherwise} \end{cases}$$

Let's represent the following directed graph using adjacency matrix.

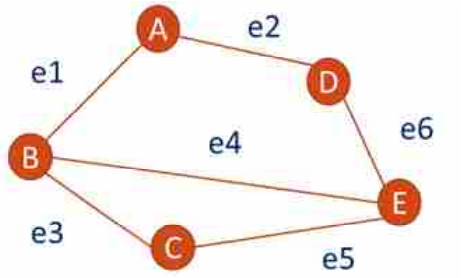


Directed Graph

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

Adjacency Matrix

Value 1 at particular row and column indicates that there is a direct edge between those two vertices. Value 1 at row A and column B indicates that there is an edge who origins from A and terminates at B. We can also represent undirected graph using adjacency matrix. Following example shows the adjacency matrix representation of undirected graph.



Undirected Graph

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

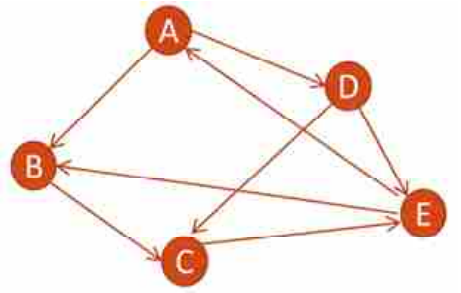
We can also represent weighted graph in the adjacency matrix. In the case of weighted graph, value 1 will be replaced by the actual weight of the connected edge.

The representation of the graph is simple and easy using adjacency matrix however it requires more storage space.

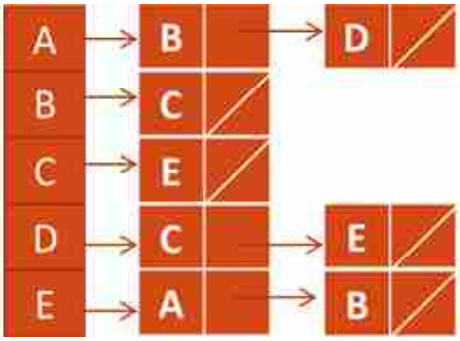
❖ **Adjacency List Representation of Graph :**

In Adjacency List representation, an array of linked list is used for representation. The size of the array equals the number of nodes or vertices of the graph. The ith array entry indicates the list of vertices which are adjacent to the ith node of the graph.

The Adjacency List representation of following directed graph is as under :



Directed Graph



Adjacency List

This type of representation requires less storage space. It is also easy to store additional information in the data structure. However there isn't a quick way to check whether a given edge is present in the graph ?

❑ **Check Your Progress – 2 :**

1. What are the different types of graphs ?
2. What are the different ways to represent graphs in memory ?

10.5 Graph Traversal :

The process to visit all the nodes of the graph is known as graph traversal. There are two popular approaches for Graph Traversals :

– Breadth First Search (BFS)

– Depth First Search (DFS)

10.5.1 Breadth First Search (BFS) :

Breadth First Search (BFS) is used to find the shortest distance between pair of nodes of the graph. Shortest distance means to traverse minimum edges to reach from node A to node B. It uses Queue data structure to store the visited nodes of the graph. BFS works as follows :

– Starting from one node, it is visited and marked as visited.

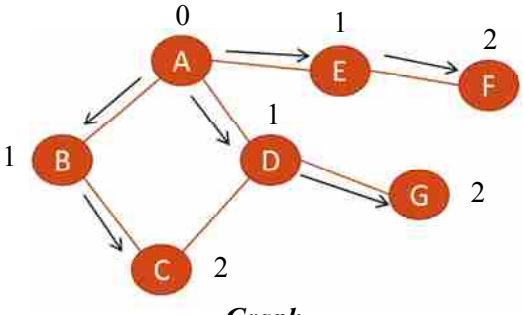
– All the nodes adjacent to the visited node are then visited and placed in the queue.

– In this fashion all the nodes of the queue are picked one by one and their adjacent nodes are visited. This process continues until the queue is empty.

Breadth first traversal/search visits all adjacent nodes of a visited node first. Then it picks one of these adjacent node and visit its adjacent nodes.

It traverses the nodes as per tree level. Web crawlers or bots use BFS concept. They first visit any webpage and then all the links of that web page are visited.

Example : The BFS traversal of the following graph would be as follows :



Graph

❖ **BFS Traversal : A, B, D, E, C, G, F :**

The traversal starts with node A. It is marked as visited and placed into the queue. Then all the adjacent nodes of A which are B, D, E are visited and placed into queue. Now no more adjacent node of A is left for visit so remove first element form the queue which is A and check if any adjacent node of it is left unvisited ? Here all are visited for node A so remove next node from the queue which is node B. All the adjacent nodes of B are then checked for visit, here node C which is adjacent to node B is not visited yet so it is marked as visited and placed into the queue. Similarly all the nodes of the queue are one by one deleted and their adjacent nodes are checked. If they are not visited then their status is marked as visited. The final sequence of visit for BFS would be A, B, D, E, C, G, F.

The labels 0, 1 and 2 on the nodes indicate the distance of that node from the starting node. Here in our case starting node is A so A is marked as label 0 because the distance between A to A is zero. B is marked as 1 means we need to travel one edge to traverse from node A to node B. Similarly node C is marked as 2 which indicate that you need to traverse minimum 2 edges to reach from A to C. The queue representation during the BFS is shown below.

Queue Operations	
Insert A	A
Insert B	A B
Insert D	A B D
Insert E	A B D E
Remove A	B D E
Remove B	D E
Insert C	D E C
Remove D	E C
Insert G	E C G
Remove E	C G
Insert F	C G F
Remove C	G F
Remove G	F
Remove F	Empty Queue

The left side of the image shows the insertion or deletion of the element form queue and the right part of the image shows the queue with its elements.

❖ **Algorithm for BFS :**

This algorithm performs a breadth first search on a graph G beginning with a starting node A.

In breadth first search, queue data structure is used

Suppose Graph G starts with node A then,

1. Insert starting node A into queue.
2. Repeat steps 3 and 4 until queue is empty.
3. Remove the first node N from queue, process it and change the status of it as visited.
4. Find out all the adjacent nodes (neighbours) of node N, if they are not visited yet then insert them into queue.
5. Step 3 loop ends.
6. Exit.

10.5.2 Depth First Search (DFS) :

Depth First Search (DFS) is another graph traversal technique. Unlike BFS, DFS uses stack data structure to store the visited nodes. DFS works as follows :

– Starting from node A, it is visited and marked as visited.

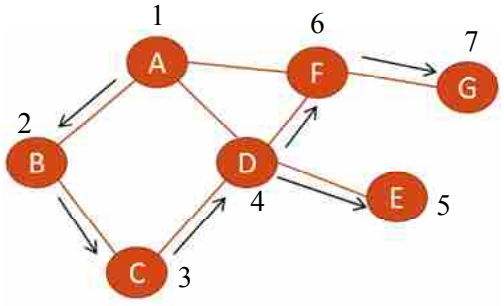
– Pick one adjacent node of A and mark it as visited. It becomes new starting node.

– Now visit the adjacent node of new starting node which leaves the exploration of original node.

– This visit continues until current path has a node which has out degree 0 or a node who's all the adjacent nodes are marked as visited.

– Search moves backward on the same path to last node that has adjacent nodes which are still not visited and marked. Process continues in such fashion until all the nodes are marked as visited.

The DFS traversal of the following graph is as under :



❖ **DFS Traversal : A, B, C, D, E, F, G :**

The traversal starts from node A. A is visited and pushed into stack. Now it will pick any of its adjacent node, here we take B so now B is marked as visited and pushed into stack. Now any adjacent node of B is picked, here we pick C so C is visited and pushed into the stack. Now D which is the adjacent node of C is picked and marked as visited and pushed into the stack. Now E which is adjacent to D is marked as visited and pushed into the stack. Now there are no more adjacent nodes to E so it will pop the last inserted element from the stack and start visiting its adjacent nodes which are left to

visit which is node F. So F is marked as visited and pushed into stack then it will visit G and it is pushed into the stack. The process continues until the stack is empty.

DFS is a recursive algorithm and it uses the concept of backtracking. Backtracking means that when you reach to the end of the current path where there isn't any node further in the path you move backwards to the same path to find the nodes to traverse.

The labels 1, 2, 3, 4, 5, 6, 7 indicate the sequence number of visited nodes. The stack representation during DFS is as below :

Stack Operations	
Push A	A
Push B	A B
Push C	A B C
Push D	A B C D
Push E	A B C D E
Pop E	A B C D
Push F	A B C D F
Push G	A B C D F G
Pop G	A B C D F
Pop F	A B C D
Pop D	A B C
Pop C	A B
Pop B	A
Pop A	Empty Stack

The left portion shows the push and pop operations and right part shows the status of stack with its elements at each stage.

❑ **Check Your Progress – 3 :**

1. What are BFS and DFS traversals ? Explain with examples.

.....

.....

10.6 Shortest Path Algorithms :

In Graph Theory, Shortest Path Problem is about to find the path between pair of vertices in the graph such that the total sum of the weights of the edges is minimum. Different Shortest Path Algorithms are available to find such path. We are going to discuss two shortest path algorithms which are :

– Kruskal's Algorithm

– Prim's Algorithm

Before we discuss these algorithms in detail, we should know the concept of Spanning Tree and Minimum Spanning Tree.

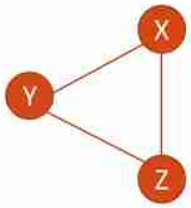
❖ **Spanning Tree :**

It is a subset of the graph in which all the vertices or nodes are covered with minimum number of edges.

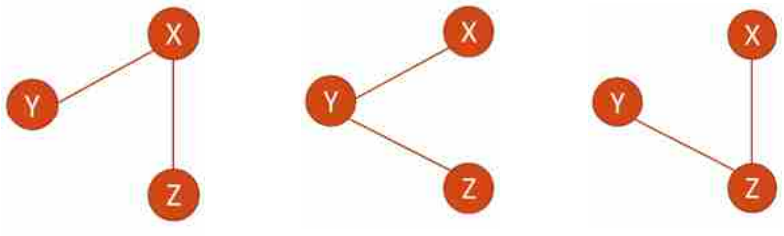
Therefor a spanning tree has :

- No cycle
- It cannot be disconnected. If you remove any edge from the spanning tree then the graph becomes disconnected.

A graph can have multiple spanning trees. Following figures shows the graph and its possible spanning trees.



Graph



Spanning Trees of the Graph

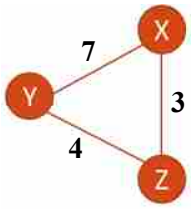
We can see in the above figure that this graph may have three different spanning trees. In all the spanning trees maximum 2 edges are required to connect all the nodes of the graph.

❖ **Minimum Spanning Tree :**

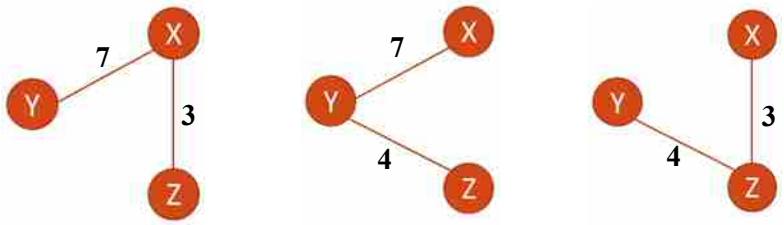
A minimum spanning tree (also known as MST) is a spanning tree whose sum of edge weight is minimum compare to all the spanning trees of the same connected and weighted graph. MST doesn't contain any cycle.

It is used to find the minimum path to connect all the vertices of the connected graph.

Example :



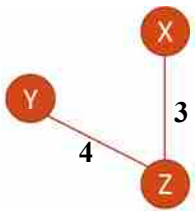
Graph



(a) **(b)** **(c)**

Spanning Trees of the Graph

For the above connected and weighted graph three spanning trees are possible which are shown in the figures (a), (b) and (c). If you count the sum of weights of the edges of all the spanning trees than it is 10, 11 and 7 respectively. So the minimum sum of edge weight is for spanning tree (c) which is 7, so it is called the Minimum Spanning Tree for this graph.



Minimum Spanning Tree

Minimum Spanning tree is used to find the shortest path in the maps.

10.6.1 Kruskal's Algorithm :

It is used to find the minimum spanning tree of a connected graph. The main goal is to find the subset of edges that creates the tree which includes all the vertices of the graph and has minimum sum of edge weights. In this approach a tree is connected to other tree (each node is considered as separate tree) if and only if it has least cost (edge weight) from all the available options.

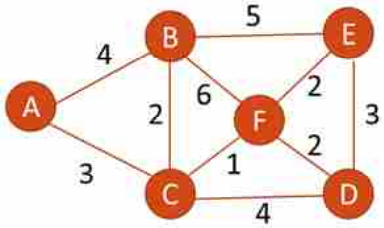
Algorithm starts with the edge which has lowest weight and keep adding edges in the minimum spanning tree having lowest weights until all the vertices are connected.

While adding the edge, care should be taken that no cycle is created. If adding an edge creates a cycle in the spanning tree then that edge is not considered. This connection should not violate Minimum Spanning Tree property (No Cycle).

This algorithm consists of the following steps :

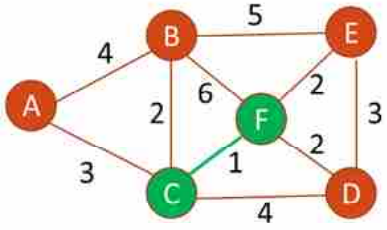
1. Arrange all the edges of the graph in ascending order of their weights.
2. Then select the edge e with the smallest (lowest) weight. If adding of that edge into MST creates cycle then discards it else adds it into the MST.
3. Repeat step 2 until n–1 edges are selected.

Example : Let's apply kruskal's algorithm to the following graph.

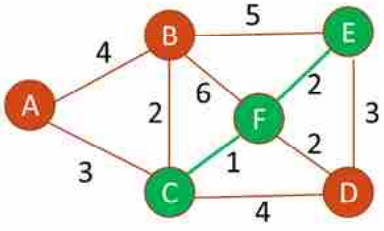


Weighted Graph

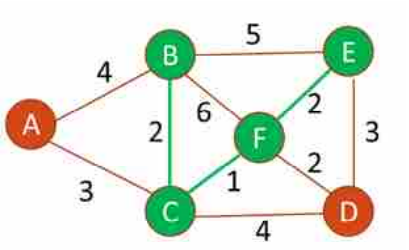
Step 1 :



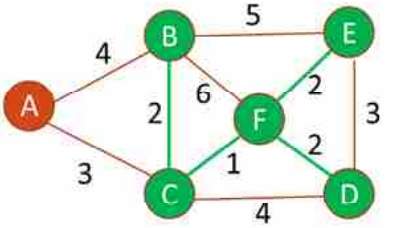
Step 2 :

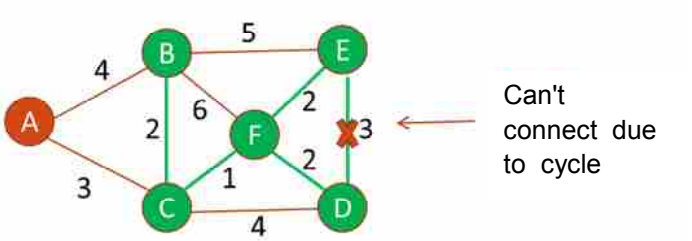


Step 3 :

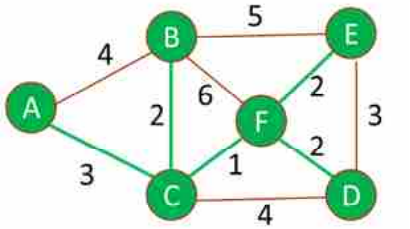


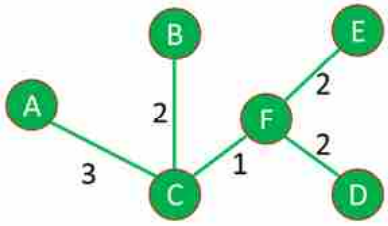
Step 4 :





Step 5 :





Minimum Spanning Tree

Step 1 : Algorithm first adds the edge between vertices C and F because it has lowest weight 1.

Step 2 : Then it adds the edge between F and E having weight 2.

Step 3 : Next it adds edge between B and C having weight 2.

Step 4 : Next it adds the edge between F and D which has also weight 2.

Now next lowest weight edges are between E and D and A & C. But if we add the edge between E and D then it creates cycle and there should not be any cycle in minimum spanning tree and hence that edge is not added.

Step 5 : Next it adds the edge between nodes A and C having weight 3. At this stage all the nodes are connected so we have the final minimum spanning tree as shown in the above figure. It connects all the vertices of the graph with minimum sum of edge weights.

10.6.2 Prim's Algorithm :

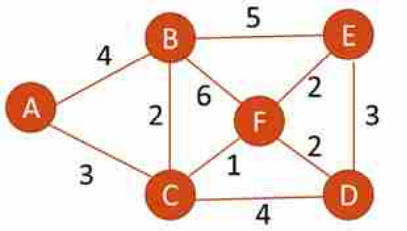
It is also used to find the minimum spanning tree of a connected graph. The main goal is to find the subset of edges that creates the tree which includes

all the vertices of the graph and has minimum sum of edge weights. As the goal is same as Kruskal's algorithm, the approach is different then it. It treats each node of the tree as a single tree and it adds new node to the spanning tree.

The algorithm has following steps :

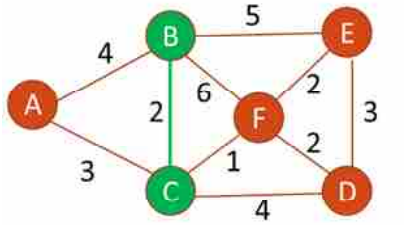
1. Choose any vertex randomly. We can start with any vertex because ultimately we have to connect all the vertices.
2. Find all the edges that connect the tree to new vertices and then find the edge having least weight among those edges and include it in the MST. Reject the edge if including that edge creates a cycle. Look for the next least weight edge.
3. Repeat step 2 until all the vertices are included and we obtain minimum spanning tree.

Example : Let's create minimum spanning tree of the following graph using Prim's Algorithm.

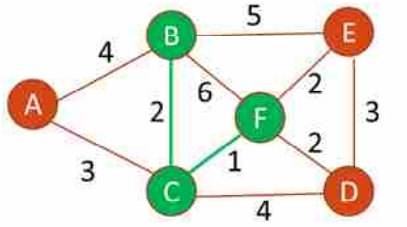


Weighted Graph

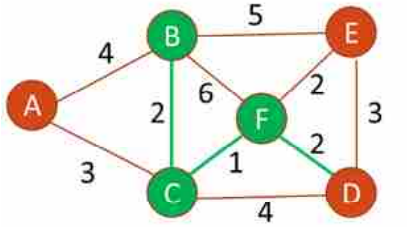
Step 1 :



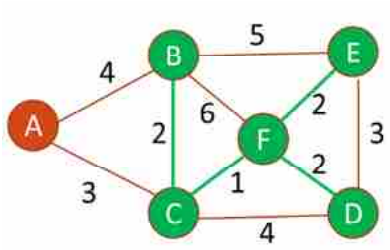
Step 2 :

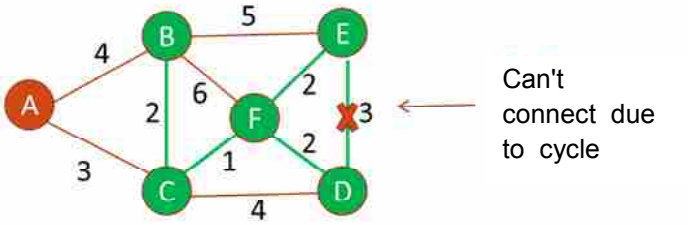


Step 3 :

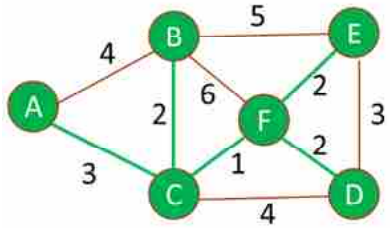


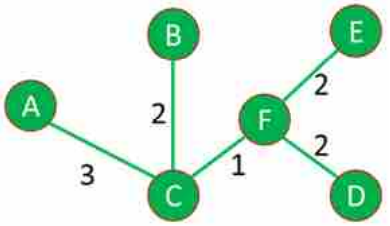
Step 4 :





Step 5 :





Minimum Spanning Tree

Step 1 : Algorithm starts with choosing one node randomly. In example B is Chosen. Now examine all the edges connected with B and find the edge which has minimum weight. Here we have edge between B and C with minimum weight 2 so it is added in the tree.

Step 2 : Now examine all the edges connected with B and C and find out the edge having least weight which is an edge between C and F having weight 1. Hence it is added in the tree.

Step 3 : Algorithm again examine all the edges connected with these added vertices and find the edge having least weight from these edges. It finds the edges between F and D and F and E with weight 2. So both the edges are added in the tree.

Step 4 : Now the edge between node E and D has lower weight 3. But adding it in the tree creates cycle so it is not added in the MST.

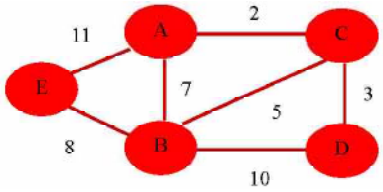
Step 5 : Next edge having minimum weight connected with the vertices of the new tree is edge between A and C having weight 3 so it is picked and added to the MST.

Now all the vertices are added in the tree so we have our final Minimum Spanning Tree as shown in the above figure.

❑ **Check Your Progress – 4 :**

1. What do you mean by Spanning Tree and Minimum Spanning Tree ?

2. Generate Minimum Spanning Tree of the following graph using Kruskal's and Prim's Algorithms.



3. A Graph in which all the edges are directed is called ?

(A) Simple Graph (B) Undirected Graph

(C) Directed Graph (D) Multi Graph

4. Which of the following data structure is used in Depth First Search ?

(A) Stack (B) Queue (C) Linked List (D) Array

5. What is the degree of isolated node in Graph ?

(A) 2 (B) 1 (C) 0 (D) 3

6. Which of the following is not the algorithm to find minimum spanning tree for the given graph ?

(A) Kruskal's Algorithm (B) Prim's Algorithm

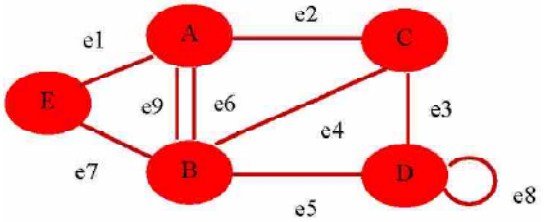
(C) Depth First Search (D) None of the above

7. Which of the following data structure is used in Breadth First Search ?

(A) Stack (B) Linked List (C) Array (D) Queue

10.7 Let Us Sum Up :

In this Unit we studied about nonlinear data structure called Graph. In a nut shell a graph is a collection of nodes called vertices, and the connections (links) between them called as edges.



There are several important terminologies with help of diagram can be quickly recapitulated A, B, C, D and E are called the vertices or nodes of the graph. e1, e2, e3, e4, e5, e6, e7, e8 and e9 are the edges of the graph. A & C are adjacent nodes. All the edges are undirected edges because no direction is given. This graph is called undirected graph because all the edges in the graph have no direction. If all the edges are directed than that graph is called directed graph. An edge of the graph which connects a node itself is called loop. Edge e8 is called the loop. The total degree of the node is the sum of edges originates and terminates to that node. Total degree of node B is 5. When pair of nodes is connected by more than one edge then such edges are called parallel edges. e6 and e9 are parallel edges. A graph which has some parallel edges is known as multi graph.

Path is a sequence of edges that connects sequence of vertices. Simple Path is a path in which all its vertices and edges are distinct.

There are different ways to represent graphs into memory such as Adjacency Matrix and Adjacency List. In adjacency matrix representation, two dimensional arrays are used to represent graph. Value 1 in the matrix indicates the presence of direct link between the pair of nodes. In Adjacency List, the array of Linked List is used to represent the graph. This approach is space efficient.

BFS and DFS are two techniques to traverse the nodes of the graphs. BFS uses queue data structure to store the details of visited node whereas DFS use stack data structure.

When you connect each vertices of the graph with minimum edges then that tree representation is called Spanning Tree. To find the shortest path in the graph, minimum spanning tree is generated. Minimum Spanning Tree (MST) is a spanning tree having minimum sum of edge weights.

Shortest path in the graph can be generated using Kruskal's as well as Prim's algorithms. Both the algorithm generates Minimum cost spanning tree form the graph.

10.8 Suggested Answer for Check Your Progress :

- **Check Your Progress 1 :**

  See Sec 10.2 and 10.3

- **Check Your Progress 2 :**

  See Sec 10.4

- **Check Your Progress 3 :**

  See Sec 10.5

- **Check Your Progress 4 :**

  1. See Sec 10.6 **2.** See Sec 10.6

  3. C **4.** A **5.** C **6.** C **7.** D

10.9 Glossary :

1. **Graph –** A graph G, G=(V,E) comprises of a nonempty set V called the set of nodes or vertices, set E which is the set of edges and a mapping from the set of edges E to a set of pairs of element V.

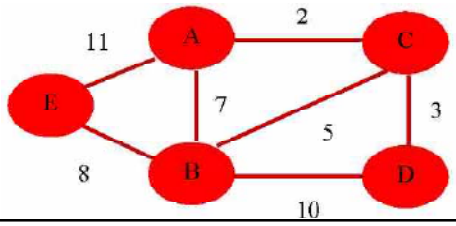
2. **Spanning Tree –** It is a subset of the graph in which all the vertices or nodes of the graph are covered (connected) with minimum number of edges.

3. **Minimum Spanning Tree –** A minimum spanning tree (also known as MST) is a spanning tree whose sum of edge weight is minimum compare to all the spanning trees of the same connected and weighted graph.

10.10 Assignment :

1. For the below graph, answer the following questions.
   - What is the type of this graph ?
   - Generate Spanning Trees for this graph

– Represent this graph using Adjacency Matrix



10.11 Activities :

1. Write a C function to traverse all the nodes of the graph.
2. Write a C program to perform BFS and DFS traversals on the graph.

10.12 Case Study :

Suppose a salesman wants to reach at five places of the city in the minimum time period. Which method shall help him in to reach at each place within a given time period in the shortest distance covered ? Draw the graph that represents these five places with distance between them and find the shortest path that connects all these five places with minimum distance.

10.13 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 Sahni.
4. Data Structures : An Advanced Approach Using 'C' by Esakov and Weises.
5. Data Structures and 'C' Programs by Christopher J. Van Wyk
6. An Introduction to Data Structures with Applications by Tremblay and Sorenson
7. https://www.javatpoint.com/breadth–first–search–algorithm
8. https://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm

BLOCK SUMMARY :

This block comprise with three units. Unit 8 gives detail understanding about tree data structure, its applications and Basic terminologies. Unit 9 is about different Binary Tree operations and traversals methods. Unit 10 gives detail understanding about another nonlinear data structure called Graph and its traversals.

In unit 8 we studied that Tree is a nonlinear data structure that contains nodes which are connected by directed edges. It contains one root node and its child nodes. Tree represents data in hierarchical form. Trees are normally used to represent computer file system. It is also used in compilers and routers. Binary trees are the special types of general trees where each node has at most 2 children.

Binary trees can be represented using Array representation or Linked List representation.

A Complete Binary tree is a tree in which each level (except last level) must be completed (full), before we start a new level. A binary tree is called Binary Search Tree or BST, if each node N of Tree must has following property, the value of node N is greater than every value in the left sub–tree of N node and it is less than every value in the right sub–tree of N node.

Unit 9 talks about the various operations and traversals methods of Binary Tree. Insertion of new element requires finding out its proper position on the Binary Search Tree. This is done by start comparing the newly inserted element with the element of root node. If the new element is less than the element of root node than it will be inserted at the left subtree else it will be inserted at the right subtree. Searching of element also requires comparing the value with the value of root node and then either on left subtree or in right subtree based on its value.

Deletion of element requires three criteria's to check. If the element to be inserted is a leaf node than it will directly deleted. If it has left or right subtree then the sub tree of the deleted node is linked directly with the parent of the deleted node. If the deleted node has both left and right subtree then the inorder successor of the deleted node will replace the deleted node and any right subtree of inorder successor will be appended to its grandparent.

Visiting any node in the binary tree is known as tree traversal. There are there tree traversals techniques :

- Inorder – (Left, Root, Right)
- Preorder – (Root, Left, Right)
- Postorder – (Left, Right, Root)

We can implement these traversals using recursive algorithms. For that we need to represent tree as a doubly linked list.

Finally in Unit 10 we have learnt the concept of graph. Graph is a collection of nodes called vertices, and the connections (links) between them called as edges. There are different types of graphs like Directed Graph – A graph in which all the edges are directed. Undirected Graph – a graph in which all the edges are not directed. Weighted graph – A graph in which weight is assigned to every edge of the graph.

There are different ways to represent graphs into memory such as Adjacency Matrix and Adjacency List. In adjacency matrix representation, two dimensional arrays are used to represent graph. Value 1 in the matrix indicates the presence of direct link between the pair of nodes. In Adjacency List, the array of Linked List is used to represent the graph. This approach is space efficient.

BFS and DFS are two techniques to traverse the nodes of the graphs. BFS uses queue data structure to store the details of visited node whereas DFS use stack data structure.

When you connect each vertices of the graph with minimum edges then that tree representation is called Spanning Tree. To find the shortest path in the graph, minimum spanning tree is generated. Minimum Spanning Tree (MST) is a spanning tree having minimum sum of edge weights.

Shortest path in the graph can be generated using Kruskal's as well as Prim's algorithms. Both the algorithm generates Minimum cost spanning tree form the graph.

BLOCK ASSIGNMENT :

❖ **Short Questions :**

1. In which situation a Graph or Tree is a more suitable data structure ?
2. How can you identify that the tree is complete binary tree or not ?
3. What is leaf node in tree ?
4. What is isolated node in graph ?
5. What is shortest path in graph ?

❖ **Long Questions :**

1. What are the real time applications of tree and graph ?
2. Write down C functions for searching an element from Binary Search Tree.
3. How can you draw a graph from adjacency matrix ? Explain with example.
4. What are the real time applications of Minimum Spanning Tree ?
5. Create one graph and perform BFS and DFS traversals on it.

❖ **Enrolment No. :** ☐

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

Unit No.	8	9	10
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 Ahmedabad

BCAR-201/
DCAR-201

Data Structure Using C

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

TECHNIQUES (SEARCHING AND SORTING) AND FILE STRUCTURE

Block Introduction :

This is a collection of algorithms for the sorting and searching of records from the data base or from a memory location.

The first unit introduces the basic searching techniques of records stored in the database stored in the memory location. The Second unit presents several sorting algorithms and various types of searching methods and techniques to retrieve the records from a database. Finally, the third unit deals with file structures that allow efficient insert, search, and delete operations.

Block Objectives :

After learning this block, you will be able to :

- Understand the concept of Searching
- Understand the types of Searching
- Implement algorithms of various Searching methods
- Understand the concept of Sorting
- Understand the difference between Internal and External Sorting
- Understand the different types of Sorting
- Implement the different Sorting methods
- Understand the concept of Sorting
- Understand the difference between Internal and External Sorting
- Understand the different types of Sorting
- Implement the different Sorting methods
- Understand file structure–concept of fields
- Understand files and Records
- Understand sequential and Index file organizations
- Understand hashing Techniques

Block Structure :

Unit
11

SEARCHING TECHNIQUES

UNIT STRUCTURE

11.0 Learning Objectives :

After learning this unit, you will be able to :

- Understand the concept of Searching
- Understand the types of Searching
- Implement algorithms of various Searching methods

11.1 Introduction :

Large amounts of data are often stored in computer systems and individuals use this data by retrieving records according to various search criteria. Therefore, storing data efficiently in order to facilitate quick searches is a vital element.

This section lays emphasis on the investigation of the performance of some searching algorithms and the data structures which they use.

11.2 Sequential or Linear and Binary Search :

The sequential search or linear search is an algorithm that is utilized to search through a list of data in order to find a particular value.

The search initiates with matching values of the array. Initially, the first element is compared with the element to be searched then with the second element and so on, continuing this process 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.

It is an easy and straightforward method for searching data in an array. The search shall initiate at the beginning of the data collection gradually moving forward until either the desired target has been found or the search space has exhausted. In order to employ this method, one must be aware of the size of

the area to search and the start of the data collection. Alternatively, a unique value could be used to signify the end of the search space. This method is commonly applied to such an array data structure that's upper and lower bounds are known.

For Example,

Consider the given array in which a total of 10 elements are stored and we want to search a particular element say, 15 in it. The Linear search can be illustrated on this set of data as given below :

12, 43, 25, 65, 11, 78, 42, 92, 6, 76

Now the same can be stored in the array as :

12	43	25	65	11	78	42	92	6	76
0	1	2	3	4	5	6	7	8	9

Now as 15 is to be searched in the above array, so we will start from the 0th position and compare the element present at that position with 15. As in this case, element at 0th position is 12, as 12 is not equal to 15, so the pointer will move to next position, that is, 1 and will compare the element 43 with 15, which is again not equals to 15. This process continues till the last position in order to search 15, now as 15 is not present in the list, so after traversing the entire list, a message of "Element not found" is displayed. If the element to be searched is present in the array then the element along with its position is displayed.

❑ Check Your Progress – 1 :

1. In _____ searching algorithm, we compare search element with all other elements.

   [A] Sequential [B] Binary [C] Bubble [D] Radix

2. _____ searching algorithm can be apply on sorted and un–sorted array.

   [A] Radix [B] Heap [C] Binary [D] Linear

Binary Search is one of the most efficient methods of searching a sequential table without the use of auxiliary indices or tables. Basically, the key value to be searched is compared with the key of the middle element of the table. If they are equal, the search ends successfully, otherwise the upper or lower parts of the table are searched correspondingly.

Basically, the elements of the table are in sorted (ascending/descending) order. So, to initiate the searching process first the array is divided into two equal halves and the key value to be searched is compared with the element present at the middle position. If both of them are equal, then the position of the search key is returned (turn back) and searching will be terminated, otherwise, the search key element will be compared with the middle element, where if the item is greater than the middle key then the item that is searched shall be placed in the upper half whereas, if it is less then it shall be placed on the lower half of the array. of the array only. 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 (or from the half) in which the target must reside. Hence the algorithm narrows the

search area by half at each step until it has either found the target data or the search fails.

This algorithm is easy to remember if you think about a child's guessing game properly. I am thinking of a number between 1 and 1000 and asked you to guess the number. Each time you guessed I would tell you "higher" or "lower". Of course, you would begin by guessing 600, then either 300 or 750 depending on my response. You may continue to refine your choice until you got the number correct according to your imagination.

For Example, for the above–mentioned set of data if the position of 15 is to be found by means of using binary search technique then there can be three possibilities :

If the data to be searched is present at the middle position then it can be explained with the given example,

223	555	666	772	777	885	889	992	996	999
0	1	2	3	4	5	6	7	8	9

Now, suppose the element to be searched is 77, then after applying the binary search technique (or method),

Lower bound is = 0

Upper bound is = 9

Mid value is = (0 + 9)/2 = 4.5 = 4

223	555	666	772	777	885	889	992	996	999
0	1	2	3	4	5	6	7	8	9

Now, compare with the element 77 which has to be searched with the element stored at the mid position, you see as in this case the element stored at mid position is 77 which is equals to the number to be searched from stored. So, the position of the same can be printed as 4.

Well, the second condition is illustrated as below. If the number to be searched is 23 then it will be searched in the sub array present to the left of the mid position from stored.

223	555	666	772	777	885	889	992	996	999
0	1	2	3	4	5	6	7	8	9

Now, by applying the binary search technique, the position of the required element can be obtained by following the given steps :

Find the mid value by dividing the values of lower and upper bounds, Mid = (lb + ub)/2

The mid value obtained is 4, now compare the value stored at that position with the number to be searched that is,23. Now, as 77 is not equals to 23 and is less than 77 so it will be searched to the left of the array, making the value of lb = 0 and ub = 3.

Now, the value of lb = 0 and ub = 3, find the mid value

Mid = (lb +)/2

(0 + 3)/2 = 1

Now as the value of mid is 1, so compare the element stored at that position with the element to be found. As 23 is not equals to 55 but is less

than 55 so it will be searched to the left of the 55, making the value of lb=0 and ub=0.

Now, again find the mid value using the values of lb and ub.

Mid = (lb + ub)/2

(0 + 0)/2 = 0

As the value of mid is 0, so compare the element stored at 0th position with 23. Now 23 is equals to the number which we are searching, that is, 23. So, the position 0 will be returned and we can say that Search is Successful.

The third situation can be if the number to be searched is greater than the mid value, that is, suppose we want to search 96 in the given array, then the steps of finding its position will be as given below :

223	555	666	772	777	885	889	992	996	999
0	1	2	3	4	5	6	7	8	9

Now, by applying the binary search technique, the position of the required element can be obtained by :

As the value of lb = 5 and ub = 9, the mid value can be obtained by

Mid = (lb + ub)/2 = (5 + 9)/2 = 7

Now, compare the element present at 7th position with the number to be searched. That is, 92 is not equals to 96. So, 96 will be compared to 92 which is greater than it. Now, 96 will be searched to the right of the array making the value of lb = 8 and ub = 9.

Obtain the mid value by

Mid = (lb + ub)/2 = (8 + 9)/2 = 17/2 = 8

Now, just compare the element present at the 8th position with the element to be searched that is, of course 96. Now as both the numbers are equal, hence, we can say that, Search is very successful.

❑ **Check Your Progress – 2 :**

1. In _____ searching algorithm, we compare search element with middle element of the array.

   [A] Sequential [B] Binary [C] Bubble [D] Radix

2. _____ searching algorithm can be applied on sorted array.

   [A] Radix [B] Heap [C] Binary [D] Sequential

❖ **Important Points about 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.

Termination Condition : The following two conditions are termination conditions.

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.

Addition of an element (add_data) : When an element is added, it is necessary to ensure that each item has been placed at its correct place in the array. If the process for this is simple then :

The array is searched until the correct place is found to insert the new item.

Shift all the existing items up by one position.

Then insert the new item into the empty position thus it is created.

❖ **Efficiency Analysis of Binary Search :**

We know earlier that in the case of Binary, at each step of the algorithm the block of items is divided and key item will be searched in both halves. We can divide a set of n items in half at most log2 n times.

Thus, we can say that the running time of Binary search is proportional to log n and also, we can say that it is O (log n) algorithm.

❑ **Check Your Progress – 3 :**

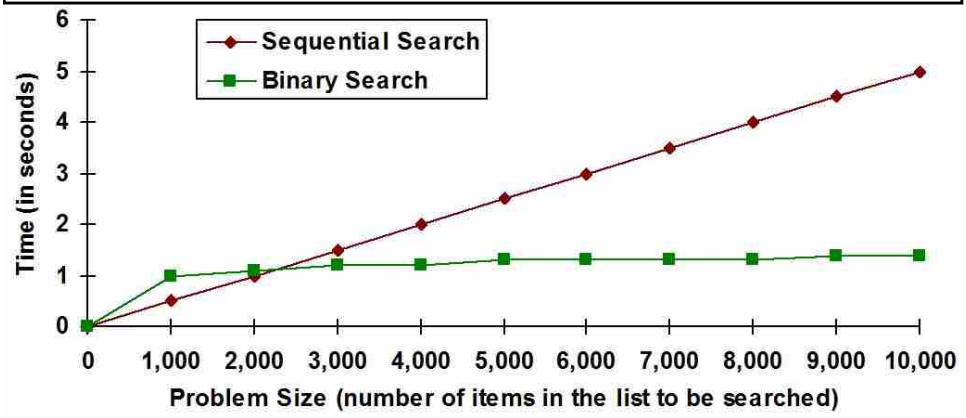
1. The best–case complexity of the linear search algorithm is _____.

   [A] O (n) [B] O (1) [C] O (log n) [D] O (n2)

2. The worst–case complexity of the linear search algorithm is _____.

   [A] O (n) [B] O (1) [C] O (log n) [D] O (n2)

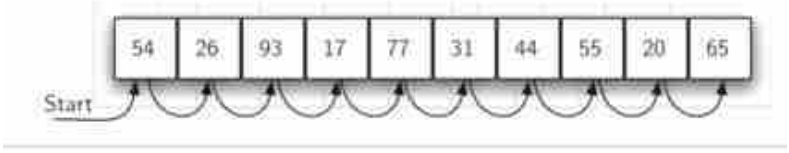
11.3 Algorithms for Sequential and Binary Search :



❖ **Algorithms for Sequential Search :**

When data items are stored in a collection such as a list or in the database files, we say that they have a linear or sequential relationship. Each and every data item 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.

Clearly shows how this search works. Starting at the first item in the list, we simply move from item to item (one by one), following the underlying sequential 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 discovered 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 compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub–array to the left of the middle element or, if the search key is greater, on the sub–array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication 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 ($\log_{2(N)}$ + 1) algorithm. Let see the best case, in which the first comparison returns a match, requires a single 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 binary search algorithm is _____.

   [A] O (n) [B] O (1) [C] 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)

11.4 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 :");
    }
}
```

11.5 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) */
#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 :");
        }
    }
```

11.6 Let Us Sum Up :

This Unit is also 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.

In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub–array to the left of the middle element or, if the search key is greater, on the sub–array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

11.7 Suggested Answer for Check Your Progress :

- **Check Your Progress 1 :**

  1. [A] **2.** [D]

- **Check Your Progress 2 :**

  1. [B] **2.** [C]

- **Check Your Progress 3 :**

  1. [B] **2.** [A]

- **Check Your Progress 4 :**

  1. [B] **2.** [C]

11.8 Glossary :

1. **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
2. **Sequential Search –** Sequential search that is suitable for searching a list of data for a particular value.

11.9 Assignment :

1. Write the applications of a binary search and a sequential search.
2. Write the advantages of a binary search over sequential search.

11.10 Activities :

1. Write a C–program to find the given element from linked list using a sequential search.
2. How do we find whether the given data is present in a linked list or not by using the Binary Search Method ?

11.11 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 ?

11.12 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.

Unit
12

SORTING TECHNIQUES

UNIT STRUCTURE

12.0 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

12.1 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.

12.2 What is sorting ?

Sorting is a method

Sorting 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.

Sorting algorithms are judged by their efficiency and are a vital aspect of managing data. In this case, efficiency refers to the algorithmic efficiency as the size of the input grows large and is generally based on the number of elements to sort. Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key record will be sorted.

❑ **Check Your Progress – 1 :**

1. _____ is a method of arranging data elements in a particular order.

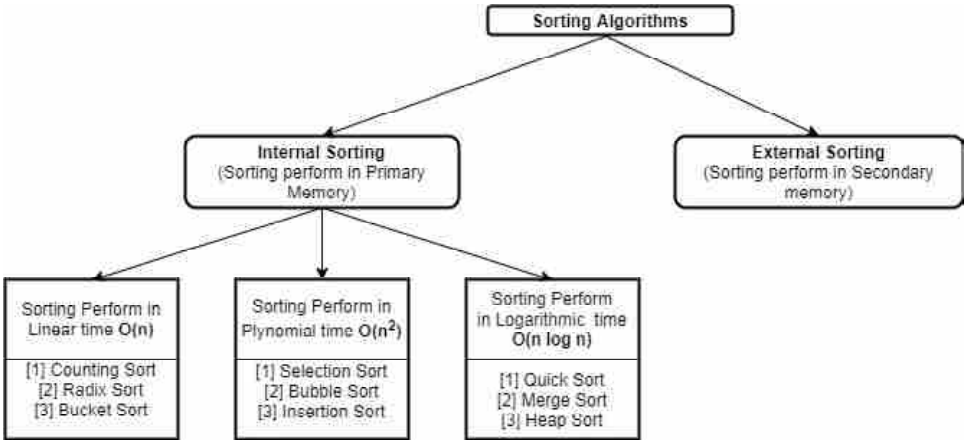
   [A] Searching [B] Sorting

   [C] File [D] None of the above.

2. Records are sorted based on _____.

   [A] File [B] Ascending [C] Descending [D] Key

12.3 Types of Sorting :



Types of Sorting

The sorting method is categorized into two types, internal sorting and external sorting, depending on the type of memory on which the data to be sorted is stored.

12.3.1 Internal and External :

❖ **Internal Sorting :**

Any sort algorithm which uses main memory exclusively during the sort is an example of Internal Sorting.

This assumes high–speed random access to all memory. It is independent of time to read/write a record and it also takes input only which can be fit into its memory i.e. it takes small input.

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 secondary devices. Time required to access is considered significant in external sorting. External sorting depends on device type and the number of such devices that can be used at a time. These sorting methods lay equal emphasis on the system aspect as well as on the algorithm aspect.

The most popular method for sorting on external storage device is Merge sort.

This method consists of two distinct phases. First, segments of the input file are sorted using a good internal sort method. These sorted segments are known as runs, which are written out onto external storage as they are generated. Second, the runs generated in phase one are merged together.

This will be clear by following example,

Let us consider a file F containing 6000 records to be sorted. The main memory is capable of sorting 1000 records at a time. The file F is stored on one disk and we have in addition another scratch disk, the block length of file is 500 records. File could be treated as 6 sets of 1000 records each.

Phase 1 – In this phase, each set is to be sorted by internal sorting method and stored on the scratch disk as a 'run'.

Phase 2 – The 6 runs will merge as follows :

Allocate 3 blocks of memory each capable of holding 500 records. Two buffers B1 and B2 will be treated as input buffers and third B3 as output buffer. Now, the method of merging is as follows :

1. 6 runs R1, R2….. R6 on the scratch disk
2. 3 buffers B1,B2,B3
   (a) Read 500 records from R1 into B1.
   (b) Read 500 records from R2 into B2
   (c) Merge B1 and B2 and write into B3.
   (d) When B3 is full, write it out to the disk as run R11. Similarly, merge R3 and R4 to get R12 and so on.

Thus from 6 runs of size 1000 each, we have now 3 runs each of size 2000. The above steps are repeated for step R11 and R12 to get a run of size 4000. This run is merged with R13 to get a single sorted run of size 6000.

❖ **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.

❑ **Check Your Progress – 2 :**

1. _____ sorting is done on secondary/ auxiliary memory.

   [A] Internal [B] External [C] Sequential [D] Linear

2. _____ sorting is done in the main memory.

   [A] Internal [B] External [C] Sequential [D] Linear

3. Identify the Internal sorting method(s) from the given options :

   [A] Bubble sort [B] Insertion sort

   [C] Selection sort [D] All of the above

4. From the given below options, which sorting algorithm is not type of Linear time.

   [A] Bubble sort [B] Counting sort

   [C] Radix sort [D] Bucket sort

5. From the given below options, identify sorting algorithm whose complexity is O (log n).

   [A] Bubble sort [B] Counting sort

   [C] Radix sort [D] Merge sort

12.4 Bubble :

Bubble

The first type of sorting which we are going to discuss is Bubble Sort which is very well known among beginners of programming. The Bubble sort

method is easy to understand and implement and of all the sorts it is considered as least efficient.

This works by continuously several time stepping through the list to be sorted, in which each pass consists(having) of comparing each element in the file with its descendant (successor), x[i] and x[i+1] and just interchanging the two elements if they are not in proper order (correct order). The pass through the list is repeated until no swaps are needed, this indicates that the list is sorted (registered). This method is called bubble sort because each number slowly "bubbles" up to its proper position in stored.

For Example, consider the given set of elements in an array which has to be sorted :

22, 56, 45, 39, 11, 99, 82, 32

The following comparisons are made on the first pass :

x[0] with x[1] (22 with 56) No Interchange

x[1] with x[2] (56 with 45) Interchange

x[2] with x[3] (56 with 39) Interchange

x[3] with x[4] (56 with 11) Interchange

x[4] with x[5] (56 with 99) No Interchange

x[5] with x[6] (99 with 82) Interchange

x[6] with x[7] (99 with 32) Interchange

Thus, after the first pass, the file is in the order

22, 45, 39, 11, 56, 82, 32, 99

Notice, that after the first pass, the largest element (i.e. 99) is in its proper position within the array. Now, after the second pass the elements will be in the given order :

22, 39, 11, 45, 56, 32, 82, 99

Notice that 82 is at second highest position. Since each iteration places a new element into its proper position, a file of n elements requires no more than n–1 iterations.

The complete set of iterations is as follows :

Iteration 0(Original file)	22	56	45	39	11	99	82	32
Iteration 1	22	45	39	11	56	82	32	99
Iteration 2	22	39	11	45	56	32	82	99
Iteration 3	22	11	39	45	32	56	82	99
Iteration 4	11	22	39	32	45	56	82	99
Iteration 5	11	22	32	39	45	56	82	99
Iteration 6	11	22	32	39	45	56	82	99
Iteration 7	11	22	32	39	45	56	82	99

We have seen that n–1 passes are sufficient to sort the file of size n. But, in the above example, where 8 elements were present, the file was sorted after five iterations, making the last two iterations unnecessary. So, to eliminate the unnecessary passes we should detect that the file is already sorted. Under this method, if the file can be sorted in fewer than n–1 passes, the final pass makes no interchanges.

❖ **Algorithm for Bubble Sort :**

```
Procedure bubblesort (A : list of sortable items)
do
swapped := false
for each i in 0 to length (A) – 2 inclusive do :
if A[i] > A [i+1] then
swap (A[i], A [i+ 1] )
swapped := true
end if
end for
while swapped
end procedure
```

❖ **Complexity of Bubble Sort :**

Bubble sort method has n–1 passes and n–1 comparisons on each pass. Then in the first pass we can see the largest element gets place at the last position. There are n–2 comparisons in the second step, which places the second largest element in the next to last position and so on. Thus,

$f(n) = (n - 1) + (n - 2) + \ldots\ldots\ldots\ldots + 2 + 1 = n(n - 1)/2$

$= n^2/2 + O(n) = O(n^2)$

Thus, the Bubble Sort has worst–case and average complexity both $O(n^2)$, where n is the number of items being sorted.

❑ **Check Your Progress – 3 :**

1. In the _____ sorting algorithm two nearby elements are compared :
   [A] Selection [B] Quick [C] Merge [D] Bubble
2. Complexity of the Bubble sort algorithm is _____.
   [A] O (1) [B] O (n) [C] O (n^2) [D] O (log n)
3. Identify the sorting algorithm in which nested loops are used.
   [A] Bubble [B] Quick
   [C] Merge [D] None of the above

12.5 Insertion :

Insertion Sort is a simple sorting algorithm, in which the sorted array is built one entry at a time. It is much less efficient on large lists. However, it provides several advantages :

Simple Implementation

Efficient for small sets of data

Adaptive, that is, efficient for data sets that are already substantially sorted.

Stable, that is, does not change the relative order of elements with equal keys.

In–place, that is, only requires a constant amount of additional memory space.

Suppose an array A with n elements is in memory then the insertion sort algorithm scans the array from first position to the last position, inserting each element into its proper position in the previously sorted subarray.

For Example,

Suppose an array A contains 8 elements as follows :

75, 32, 45, 12, 84, 23, 64, 57

Now, the given figure illustrates the insertion sort algorithm. The coloured element indicates the number with which the other elements will be compared and the arrow indicates the proper place of inserting the element to be compared.

PASS	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
K=1	75	32	45	12	84	23	64	57
K=2	75	32	45	12	84	23	64	57
K=3	32	75	45	12	84	23	64	57
K=4	32	45	75	12	84	23	64	57
K=5	12	32	45	75	84	23	64	57
K=6	12	32	45	75	84	23	64	57
K=7	12	23	32	45	75	84	64	57
K=8	12	23	32	45	64	75	84	57
SORTED ARRAY	12	23	32	45	64	75	84	57

❖ **Pseudo code for Insertion Sort :**

```
InsertionSort (array A)
begin
for K := 1 to length[A] do
begin
        value := A[K];
        j := K–1;
        while j>= 0 and A[j] > value do
            begin
            A[ j + 1] := A[j];
            A[j] := value;
            j := j–
            1; end;
        end;
    end;
```

❖ **Complexity Analysis :**

Worst Case : In this case, array is in reverse order and the inner loop must use the number k–1 of comparisons. Hence,

$f(n) = 1 + 2 + \ldots\ldots\ldots\ldots\ldots + (n - 1) = n(n - 1)/2 = O(n^2)$

Average Case : In this case there will be approximately $(k - 1)/2$ comparisons in the inner loop. Accordingly, for the average case,

$f(n) = 1/2 + 2/2 + \ldots\ldots\ldots\ldots (n - 1)/2 = n(n - 1)/4 = O(n^2)$

Insertion sort's overall complexity is O (n^2) on average, regardless of the method of insertion. Number of writes is O (n^2) on average, but number of comparisons may vary depending on the insertion algorithm. It is O (n^2) when shifting or swapping methods are used and O (n log n) for binary insertion sort.

❑ **Check Your Progress – 4 :**

1. How many nesting of loop is needed to implement insertion sort algorithm.

   [A] 2 [B] 1 [C] 4 [D] 3

2. Complexity for insertion sort is _____.

   [A] O (1) [B] O (n) [C] O (n2) [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 [B] 7 [C] 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 [B] 8, 32, 34, 51, 64, 21

   [C] 8, 34, 51, 64, 32, 21 [D] 8, 34, 64, 51, 32, 21

12.6 Selection :

Selection

The Selection Sort method completely consists of a selection phase in which the smallest (lowest) element of the array is selected (identified) and is positioned (placed) at the first place. In the second pass, next smallest element is selected and is placed at the second position and so on. The process continues till the end of the array. The initial queue is reduced by one element at each step. At the end of n – 1 selections, the entire array gets sorted.

For Example,

Suppose an array A contains 8 elements as follows :

75, 32, 45, 12, 84, 23, 64, 57

Now, applying the selection sort algorithm to A, we will receive the elements as shown in the given figure. The coloured elements indicate the elements which are to be exchanged.

PASS		X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]
I=0	J=1	**75**	**32**	45	12	84	23	64	57
	J=2	**32**	75	**45**	12	84	23	64	57
	J=3	**32**	75	45	**12**	84	23	64	57
	J=4	**12**	75	45	32	**84**	23	64	57
	J=5	**12**	75	45	32	84	**23**	64	57
	J=6	**12**	75	45	32	84	23	**64**	57
	J=7	**12**	75	45	32	84	23	64	**57**
I=1	J=2	*12*	**75**	**45**	32	84	23	64	57
	J=3	*12*	**45**	75	**32**	84	23	64	57
	J=4	*12*	**32**	75	45	**84**	23	64	57
	J=5	*12*	**32**	75	45	84	**23**	64	57
	J=6	*12*	**23**	75	45	84	32	**64**	57
	J=7	*12*	**23**	75	45	84	32	64	**57**
I=2	J=3	*12*	*23*	**75**	**45**	84	32	64	57
	J=4	*12*	*23*	**45**	75	**84**	32	64	57
	J=5	*12*	*23*	**45**	75	84	**32**	64	57
	J=6	*12*	*23*	**32**	75	84	45	**64**	57
	J=7	*12*	*23*	**32**	75	84	45	64	**57**
I=3	J=4	*12*	*23*	*32*	**75**	**84**	45	64	57
	J=5	*12*	*23*	*32*	**75**	84	**45**	64	57
	J=6	*12*	*23*	*32*	**45**	84	75	**64**	57
	J=7	*12*	*23*	*32*	**45**	84	75	64	**57**
I=4	J=5	*12*	*23*	*32*	*45*	**84**	**75**	64	57
	J=6	*12*	*23*	*32*	*45*	**75**	84	**64**	57
	J=7	*12*	*23*	*32*	*45*	**64**	84	75	**57**
I=5	J=6	*12*	*23*	*32*	*45*	*57*	**84**	**75**	64
	J=7	*12*	*23*	*32*	*45*	*57*	**75**	84	**64**
I=6	J=7	*12*	*23*	*32*	*45*	*57*	*64*	**84**	**75**
Sorted Array		*12*	*23*	*32*	*45*	*57*	*64*	*75*	**84**

❖ **Pseudocode of Selection Sort :**

If AR is the name of the array and small and pos are the variables which will hold the smallest element and its position in the array, then the pseudocode for the same can be written as :

```
for (i=0; i<MAX; i++)
{
    for (j=i+1; j<MAX; j++)
    {
        If (X[j] < X[i])
        {
            tmp=X[i];
            X[i] = X[j];
            X[j] = tmp;
        }
    }
}
```

❖ **Complexity of Selection Sort :**

The number f(n) of comparisons in the selection sort algorithm is independent of the original order of the elements. There are n–1 comparisons during pass1 to find the smallest element, there are n–2 comparisons during pass2 to find the second smallest number and so on. Accordingly,

$f(n) = (n - 1) + (n - 2) + (n - 3) + \ldots\ldots\ldots\ldots 3 + 2 + 1$

$= n(n - 1)/2 = O(n^2)$

❑ **Check Your Progress – 5 :**

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 [B] 4 and 5 [C] 2 and 4 [D] 2 and 5

2. What is the worst–case complexity of selection sort ?

   [A] O (1) [B] O (n) [C] 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 [B] 5 [C] 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 [B] 8, 32, 34, 51, 64, 21

   [C] 8, 21, 64, 51, 32, 34 [D] 8, 34, 64, 51, 32, 21

12.7 Quick :

The Quick Sort method is also called partition exchange sort. It is a very efficient sorting algorithm invented by C.A.R Hoare. It has two phases :

The Partition Phase

The Sort Phase

The majority of the work is done in the partition phase, that is, it works out where to divide the work and the Sort phase simply sorts the two smaller

problems that are generated in the partition phase. That is why we say that Quick Sort is a good example of Divide and Conquer strategy for solving problems. As, we divide the array of items to be sorted into two partitions and then call the quicksort procedure recursively to sort the two partitions, that is, we divide the problem into two smaller ones and conquer by solving the smaller ones.

The above can be achieved by choosing a pivot element and arranging all the items in the lower part which are less than the pivot and all those in the upper part greater than it. As, in most of the cases we do not know anything about the items to be sorted, so any choice of the pivot element will do, the first element can be a convenient one.

Let us illustrate this with the help of an example, if the initial array is given as :

24 55 46 32 12 95 87 36

If the first element 24 is chosen as pivot number and is placed in its proper position, the resulting array will be :

12 24 55 46 32 12 95 87 36

Now, at this point 24 is at its proper position, each element below 24 is less than or equal to 24 (12), and each element above that position (55, 46, 32, 12, 95, 87, 36) is greater than or equal to 24.

(12) 24 (55 46 32 12 95 87 36)

Now, as 24 is at its proper position, the elements present to its left and right are yet to be sorted. Repeating the above process of choosing a pivot number and dividing the array into smaller part, that is, let's choose the pivot number as 57 and applying the above steps to it, we'll yield :

12 24 (46 36 32) 55 (95 87)

Now, 55 is at its proper position, and again there are two subarrays, specified in brackets (46, 36, 32) and (95, 87) which need to be sorted again using the above concept, that is, by choosing a pivot number. Now, repeating the above process and further repetitions yield :

12 24 (36 32) 46 55 (95 87)

12 24 (32) 36 46 55 (95 87)

12 24 32 36 46 55 (95 87)

12 24 32 36 46 55 (87) 95

12 24 32 36 46 55 87 95

Now, the final array is sorted.

❖ **Algorithm for partition of quicksort :**

Algorithm partition (S, p) :

Input–sequence S, position p of pivot

Output–subsequence L, E, G of the elements of S less than, equal to,or greater than the pivot, respectively.

L, E, G ← empty sequences

x ← S.remove(p)

while ← S.isEmpty()

```
y ← S.remove(S.first())
if y < x
L.insertLast(y)
else if y = x
E.insertLast(y)
else { y > x }
G.insertLast(y)
return L, E, G
```

❖ **Complexity of Quicksort :**

Quick Sort is unstable because efficiency of quick sort depends upon the type of storage file/array. Quick sort has different efficiencies for unsorted array/file and sorted file/array. This can be well defined by the following proof :

1. In case of Unsorted Array :

Let array size n is some power of 2, say

$n = 2m$ and $m = \log_2 n$

Let position of pivot is exactly in the middle of the subarray. So, in first pass, there are n comparisons. In second pass, there are n/2 comparisons and in third pass n/8 comparisons and so on, up to m subarrays.

After having subfiles/subarray of m times, there are n files of size 1. Hence total no. of comparisons should be :

$n + 2 * n/2 + 4 * n/4 + 8 * n/8 + \ldots\ldots\ldots n * n/n$

$= n + n + n + n \ldots\ldots\ldots n$ (m times)

$= O(n * m)$ but $m = \log_2 n = O(n \log_2 n)$

That is, the efficiency of unsorted array is $O(n \log_2 n)$.

2. In case of Sorted array :

Let array is sorted. Suppose X[lb] is in its proper place, the original file/array is split into subfiles of size 0 and n – 1.

If the above process continues, a total of n – 1 subfiles are sorted.

size of first array = n

size of second array = n – 1

size of third array = n – 2 and so on.

Let K comparisons are required to rearrange a file of size K, the total no. of comparisons to sort the entire array is :

$n + (n - 1) + (n - 2) \ldots\ldots + (n - i) + \ldots. 3 + 2$

$= O(n^2)$

That is, the efficiency for sorted array is $O(n^2)$.

In worst case, efficiency is $O(n^2)$, but it is easy to avoid the worst case. On an average, its efficiency is O (n log n).

❑ **Check Your Progress – 6 :**

1. Which if the following algorithm is fastest.

[A] Merge sort [B] Quick sort [C] Insertion sort[D] Shell sort

2. What is the worst–case complexity of Quick sort ?

   [A] O (1) [B] O (n) [C] O (log n) [D] O (n^2)

3. What is the average running time of a quick sort algorithm ?

   [A] O (n^2) [B] O (n log n) [C] O (log n) [D] O (n)

12.8 Merge :

Suppose consider A is sorted list with n elements and consider B is a sorted list with m elements. Then the operation that combines the elements of A and B into a single sorted list C with s = m + n elements is called as merging. One good way to merge is to place the elements of B after the elements of A and then use some sorting algorithm on the entire list.

Suppose consider, an array with 14 elements is in memory and are to be sorted by merge sort algorithm, let us illustrate the same with the help of the given example,

Each pass of the merge sort algorithm starts at the beginning of the array A and merge pairs of sorted subarrays as follows :

65, 32, 45, 23, 57, 89, 60, 12, 80, 21, 50, 43, 76, 34

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs :

32, 65 23, 45 57, 89 12, 60 21, 80 43, 50 34, 76

Pass 2. Merge each pair of pairs to obtain the following list of sorted quadruplets :

23, 32, 45, 65 12, 57, 60, 89 21, 43, 50, 80 34, 76

Pass 3. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays :

12, 23, 32, 45, 57, 60, 65, 89 21, 34, 43, 50, 76, 80

Pass 4. Now, merge the two sorted subarrays to obtain the single sorted array

12, 23, 32, 34, 43, 45, 50, 57, 60, 65, 76, 80, 89

Now, finally the array A is sorted.

The merge sort algorithm has an important property, after pass k, the array will be partitioned into subarrays where each subarray except the possibly the last, will contain exactly L=2k elements.

Hence the algorithm requires at most log n passes to sort an n–element array A.

❖ **Algorithm for merge sort :**

```
functionmerge_sort(m)
var list left, right, result
if length(m) ? 1
return m
var middle = length(m) / 2 – 1
for each x in m up to middle
add x to left
```

```
for each x in m after middle
add x to right
left = merge_sort(left)
right = merge_sort(right)
ifleft.last_item>right.first_item
result = merge(left, right)
else
result = append(left, right)
return result
```

❖ **Complexity of Merge Sort Algorithm :**

Let f(n) denote the number of comparisons needed to sort an n–element array A using the merge sort algorithm. The algorithm requires at most log n passes. Moreover, each pass merges a total of n elements and on the complexity of merging each pass will require at most n comparisons. Accordingly, for both the worst case and average case,

$$f(n) <= n \log n$$

The main drawback of merge sort is that it requires an auxiliary array with n elements. So, from the above we can say that the merge sort has complexity of O (n log n) in worst case, O (n log n) in average case.

❑ **Check Your Progress – 7 :**

1. Merge sort uses which of the following technique to implement sorting ?

   [A] Backtracking [B] Greedy algorithm

   [C] Divide and Conquer [D] Dynamic programming

2. Which of the following method is used for sorting in merge sort ?

   [A] Merging [B] Partitioning [C] Selection [D] Exchanging

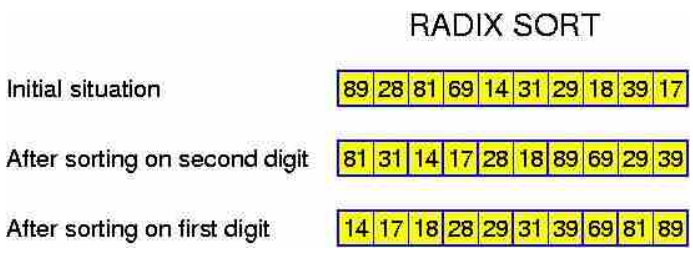
3. What is the average running time of a merge sort algorithm ?

   [A] O (n^2) [B] O (n log n) [C] O (log n^2) [D] O (n log n^2)

4. Which of the following is not a stable sorting algorithm ?

   [A] Quick sort [B] Cocktail sort [C] Bubble sort [D] Merge sort

12.9 Radix Sorting :



Radix Sorting

Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least–significant digit first, followed by the second–least significant digit and so on till the most significant digit.

The time complexity of the algorithm is as follows : Suppose that the n input numbers have maximum k digits. Then the Counting Sort procedure is called a total of k times. Counting Sort is a linear, or O(n) algorithm. So, the entire Radix Sort procedure takes O(kn) time. If the numbers are of finite size, the algorithm runs in O(n) asymptotic time.

Consider the list of integers :

36, 9, 0, 25, 1, 49, 64, 16, 81, 4

n is 10 and the numbers all lie in (0, 99). After the first phase, we will have :

Bin	0	1	2	3	4	5	6	7	8	9
Content	0	1 81	–	–	64 4	25	36 16	–	–	9 49

Note that in this phase, we placed each item in a bin indexed by the least significant decimal digit.

Repeating the process, will produce :

Bin	0	1	2	3	4	5	6	7	8	9
Content	0 1 4 9	16	25	36	49	–	64	–	81	–

In this second phase, we used the leading decimal digit to allocate items to bins, being careful to add each item to the end of the bin.

We can apply this process to numbers of any size expressed to any suitable base or radix.

12.10 Let Us Sum Up :

In this Unit we have confirmed our learning about 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 secondary devices. The most popular method for sorting on external storage device is Merge sort. Let us recapitulate the 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.

There is also learning about Insertion Sort which is a simple sorting algorithm, in which the sorted array is built one entry at a time. It is much less efficient on large lists. However, it provides several advantages.

12.11 Suggested Answers For Check Your Progress :

- **Check Your Progress 1 :**

  1. [A] **2.** [D]

- **Check Your Progress 2 :**

  1. [B] **2.** [A] **3.** [D] **4.** [A] **5.** [D]

- **Check Your Progress 3 :**

  1. [D] **2.** [C] **3.** [A]

- **Check Your Progress 4 :**

  1. [A] **2.** [C] **3.** [C] **4.** [D]

- **Check Your Progress 5 :**

  1. [A] **2.** [D] **3.** [B] **4.** [C]

- **Check Your Progress 6 :**

  1. [B] **2.** [D] **3.** [B]

- **Check Your Progress 7 :**

  1. [C] **2.** [A] **3.** [D] **4.** [A]

12.12 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 near by data element. 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 method 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).

12.13 Assignment :

1. Do the complexity analysis of Quick, Merge and Insertion Sort for worst, average and best cases in about 100–150 words.
2. Show the tracing of the following data using Selection sort, Bubble sort and Insertion sort :

   64, 34, 8, 51, 21, 32
3. Show the tracing of the following data using Quick sort algorithm.

   11, 2, 9, 13, 57, 25, 17, 1, 90, 3
4. Show the tracing of the following data using Merge sort algorithm.

   45, 9, 28, 39, 11, 20, 50, 41

12.14 Activities :

1. What are the types of Exchange sorts ? Explain algorithms with examples.
2. What is Pivot ?
3. What is stable sorting ?

12.15 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 ?

12.16 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
13

FILE STRUCTURE

UNIT STRUCTURE

13.0 Learning Objectives :

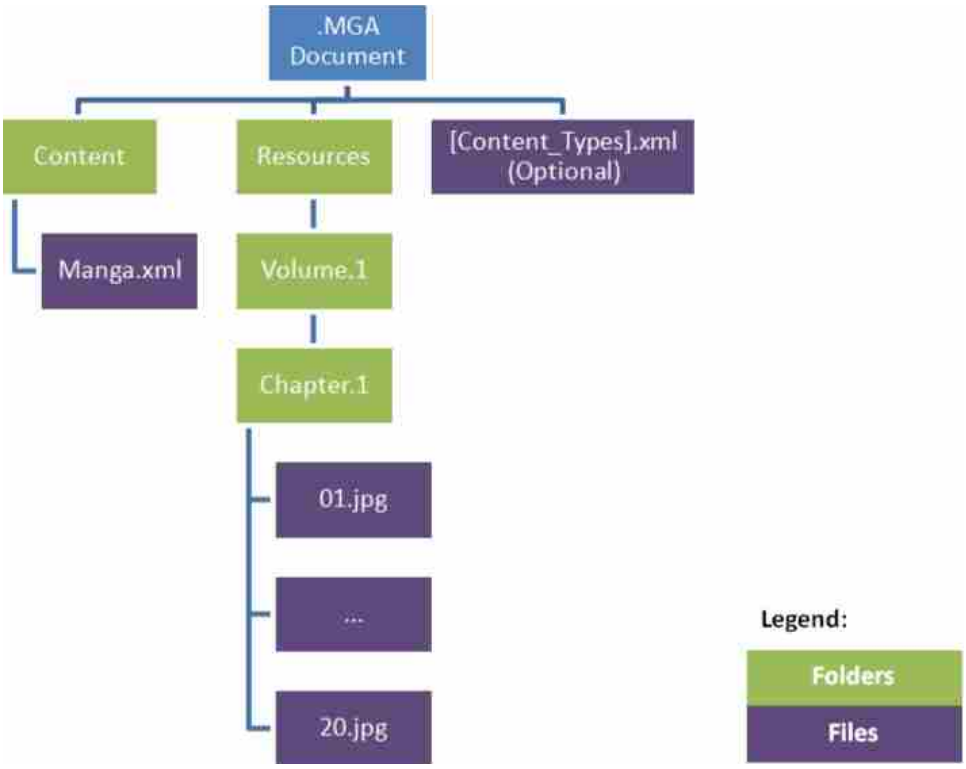
After learning this unit, you will be able to :

- Understand file structure – concept of fields
- Understand files and Records
- Understand sequential and Index file organizations
- Understand hashing Techniques

13.1 Introduction :

This chapter is mainly concerned with the way in which file structures are used. There is one important distinction 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. On the whole a file structure will specify the logical structure of the data that is the relationships that will exist between data items.

13.2 File Structure – Concept of Fields :



File Structure

❖ **File Operations :**

Operations on database files can be classified into two categories broadly.

Update Operations and Retrieval Operations

Update operations or function 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. Other than creation and deletion of a file, there are several operations, which can be done on files.

- **Open –** This operation takes care of file open and can be opened in one of two modes (forms/manners), read mode or write mode. In read mode, operating system does not allow anyone to alter data (change) it is solely for reading purpose. Files opened in read mode can be shared among several entities. Another mode is 'write' mode, in which, data modification is allowed. Files opened in write mode can be read also but cannot be shared.
- **Locate –** Each and every file has a file pointer (indicator), which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using the find (seek) operation it can be moved forward or backward.
- **Read –** By default (automatically), when files are opened in read mode (operation) the file pointer points to the beginning of file. There are options where the user can tell the (OS) operating system to where the file pointer to be located (to put on) at the time of file opening. The very next data to the file pointer is read.

- **Write –** In this operation user can select to open files in write mode, which enables them to edit the content of file. It can be deletion, insertion or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allowed doing so.
- **Close –** This operation is also most important operation from operating system point of view. When a request to close a file is generated, the operating system removes all the locks (if in shared mode) and (store or register) saves the content of data (if altered) to the secondary storage media and release all the buffers and file handlers associated with the file.

The organization of data content (records) inside the file plays a major role here. Seeking or locating the file pointer to the desired record inside file behaves 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 [B] Open

   [C] Close [D] None of the above

2. _____ file operation, writes all file blocks to the secondary memory and release all the buffers.

   [A] Create [B] Locate [C] Open [D] Close

3. To set the file position in forward or backward directions, _____ function is called.

   [A] seek [B] peek

   [C] read [D] All of the above

13.3 Files and Records :

Files and Records

A record is a sequence of values or a sequence of characters. There are three kinds of FORTRAN records, as follows :

- **Formatted –** A record containing (consists of) formatted data that requires translation from internal to external form. The formatted I/O statements have explicit format specifies (which can specify list–directed formatting)

or name list specifies (for name list formatting). Only formatted I/O statements can read formatted data.

- **Unformatted –** A record containing unformatted data (in a file) that is not translated from internal form. An unformatted record can also contain no data. The internal representation of unformatted data is processor–dependent. Only unformatted I/O statements can read unformatted data.
- **Endfile –** This indicator 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 (having various entities). There are two (forms) types of Fortran files, as follows :

- **External –** In an operation a file that exists in a medium (such as computer disks or terminals) external to the executable program.

Records in an external file must be either all formatted or all unformatted (as per the). There are three ways (methods of operation) to access records in external files : sequential, keyed access (VMS only), and direct access.

In sequential access operation, records are processed in the order in which they appear in the file. In direct access operation, records are selected by record number, so they can be processed in any order. In keyed access, records are processed by key–field value.

- **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. The organization of data, into the file is called _____.

   [A] field [B] record [C] file [D] table

2. A record containing _____ data (in a file) that is not translated from internal form.

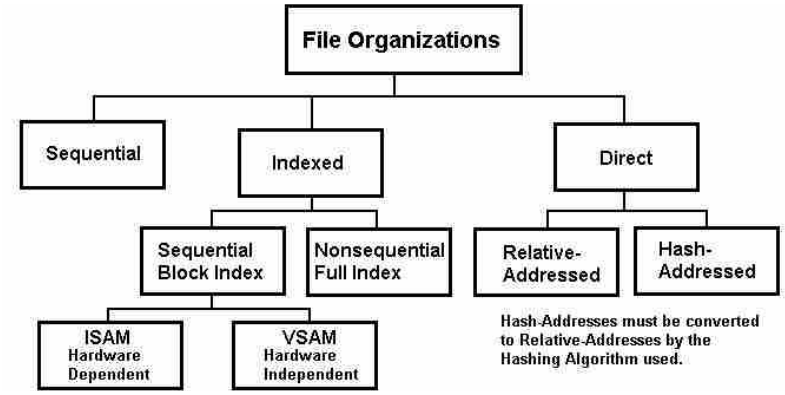
   [A] formatted [B] external [C] unformatted [D] internal

3. From the given below which is not a method of accessing records from external files :

   [A] Sequential [B] Keyed (VMS only)

   [C] Direct [D] Formatted

13.4 Sequential and Index File Organizations :



Sequential and Index File Organizations

It is a method of mapping file records to disk blocks defines file organization, i.e., how the file records are organized. The following are the types of file organization

- **Sequential File Organization –** mostly every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization mechanism, records are placed in the file in some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form some time 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 has no directory and no linking pointers. The records are generally organized in lexicographic order (one after the other) on the value of some key (assigning a key to it). In other words, a particular attribute is chosen whose value will determine the order of the records (in the manner of the arrangement). On some occasion when the attribute value is constant for a large number of records a second key is chosen to give 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 :

(1) Easy to implement;

(2) It provides fast access to the next record (In order) using lexicographic order.

- Its disadvantages :

(1) It is difficult to update(operation) – inserting (operation) a new record may require moving a large proportion of the file;

(2) Extremely slow in case of Random access.

- Few times a file is considered to be sequentially organized despite the fact that it is not ordered according to any key. Perhaps 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.
- **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.
- **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 [B] Index–sequential

   [C] Sequential [D] None of the above

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 [B] Direct

   [C] Index–sequential [D] All of the above

3. _____ file, has no directory and no linking pointers

   [A] Sequential [B] Direct

   [C] Index–sequential [D] All of the above

4. _____ file has unique keywords (no duplication), or one unique attribute–values.

   [A] Data [B] Index [C] Program [D] Driver

13.5 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 α = 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, (α = 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 · p0 – 1, so that always d bits are available.

Then, the generated address is reduced to the current range 0...p1 – 1 :

address : = KATd(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 + α/2.

❑ **Check Your Progress – 4 :**

1. _____ method is used to access the record directly into the file.

   [A] Hashing [B] Indexed

   [C] Sequential [D] None of the above

2. A _____ can be said as atypical disk block, can be chosen to be smaller or larger than a disk block

   [A] Field [B] Record [C] Track [D] Bucket

3. In basic allocation for hashing, if file has depth d=3 then how many pages will be there in the file ?

   [A] 3 [B] 6 [C] 8 [D] 9

4. In a hash function, Bi = h(Ki), The Ki indicates _____.

   [A] Index key [B] Search Key

   [C] Hash function [D] Record address

13.6 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, Endfile. 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.

13.7 Suggested Answer for Check Your Progress :

- **Check Your Progress 1 :**

  1. [B] **2.** [D] **3.** [A]

- **Check Your Progress 2 :**

  1. [B] **2.** [C] **3.** [D]

- **Check Your Progress 3 :**

  1. [C] **2.** [A] **3.** [A] **4.** [B]

- **Check Your Progress 4 :**

  1. [A] **2.** [D] **3.** [C] **4.** [B]

13.8 Glossary :

1. **Hashing –** Hashing a technique to calculate direct location of data record on the disk without using index structure.
2. **Linear hashing –** Linear hashing can be viewed on as a dynamic version of hashing with separate chaining
3. **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.
4. **Split pointer –** The "split pointer" points at the next page to be split.

13.9 Assignment :

Do the sequential indexing on 1000 no. of records present in database.

13.10 Activities :

1. What is mean by Indexed Sequential files ?
2. Write methods to implement hashing using static and dynamic hashing.

13.11 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.

1. Which organization of files is using full to perform searching on Class of students ?

13.12 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
14

PROGRAMS OF SEARCHING AND SORTING

UNIT STRUCTURE

14.0 Learning Objectives :

After learning this unit, you will be able to :

- Understand difference between Linear search and Binary search
- Understand How to implement Linear and Binary search algorithms practically
- Understand implementation of various sorting algorithms

14.1 Introduction :

In the first two units of this block, we have discussed the algorithms to implement various searching and sorting techniques. We have tried to explain each sorting and sorting algorithm in brief with some examples, whenever it is needed. In this unit, we are discussing how to implement all these algorithms practically using C-Language. We hope that, practical implementation will give you clear idea to understand each searching and sorting algorithm and also you can easily distinguish linear search with binary search and any sorting algorithm with other sorting algorithms.

14.2 Programs of Searching & Sorting :

14.2.1 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 :");
   }
}
```

14.2.2 Write a Program to Implement Binary Search

```
#include<stdio.h>
void main()
{
   int x[10];
   int i,beg,end, mid,val;
   // 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:");
        }
    }
```

14.2.3 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]);
}
```

14.2.4 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]);
  }
```

14.2.5 Write a Program to Implement Insertion 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 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]);
}
```

14.2.6 Write a Program to Implement Quick Sort

```
#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;
}
```

14.2.7 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];
    }
}
```

❑ **Check Your Progress – 1 :**

1. Which searching algorithm compares the search element with the middle element of an array ?

   [A] Linear [B] Binary [C] Bubble [D] Quick

2. In which sorting algorithm we compare Jth element with $(J+1)^{th}$ element ?

   [A] Bubble [B] Selection

   [C] Insertion [D] None of the above

3. Which sorting algorithm(s) is/are recursive ?

   [A] Quick [B] Merge

   [C] Both A and B [D] Insertion

4. Which sorting algorithm is based on divide and conquer principle ?

   [A] Linear [B] Bubble [C] Insertion [D] Merge

5. In which sorting algorithm, we choose an element to be a Pivot element?

   [A] Selection [B] Insertion [C] Quick [D] Heap

14.3 Let Us Sum Up :

In this Unit we have seen that the Linear search is perform by comparing each element of the array. For 'n' size of array, 'n' comparisons are needed in worst case. Therefore, Linear search is slow compare to Binary search method. Linear search can be applied on any type of array whether it is sorted or unsorted. Binary search algorithm on other hand can be applied on sorted array only. Here the element is compared with the middle element of the array, and based on the search value is smaller or bigger, the array is spitted. Each comparison of binary search will remove (n/2 + 1) elements (where the search element is not possible) and reduce the size of the array by n/2 – 1.

In the Bubble sort algorithm, we are comparing jth and $(j + 1)^{th}$ elements. If $(j + 1)^{th}$ element is smaller than jth element we exchange (swap) the values. In each iteration of variable 'i' we get the higher element at the end of the array.

In Selection sort algorithm, we are comparing ith and jth element. If jth element is smaller than ith element than we exchange (swap) the values. In this algorithm, we get the smallest element at ith place of the array after completion of each iteration of the loop of variable 'i'.

Insertion sort algorithm, is based on the inserting a smaller value (at proper position) by shifting greater elements towards right side.

Quick sort algorithm is a recursive process, in which we choose pivot element and shifting all the elements of the array so that smaller values (compare to pivot element) come at left side and bigger values of the array (compare to pivot element) come to the right side of the pivot element.

Merge sort algorithm is split the array recursively and then merge the array by sorting the small arrays

14.4 Suggested Answers For Check Your Progress :

❑ **Check Your Progress 1 :**

1. [B] **2.** [A] **3.** [C] **4.** [D] **5.** [C]

14.5 Glossary :

1. **stdio.h :** It is standard input output header file, which includes functions like printf(), scanf() etc.
2. **conio.h :** It is console input output header file, which includes functions like getch(), clrscr() etc.
3. **Array :** Array is a homogeneous (same type of data) collection of data, stored in conjunctive memory locations.

14.6 Assignment :

Implement each searching and sorting program using either Turbo C, Borland C or CodeBlocks.

14.7 Activities :

Make a table in which 4 columns are there. 1st column shows name of searching and sorting algorithm and rest of the 3 columns shows their Best, Average and Worst-case complexities. Find these complexities for each program we have discussed in this chapter.

14.8 Case Study :

Consider the following data for an array given in the table below and show the tracing (how data will be sorted) for each algorithm given in the 1st column of the table :

Selection Sort	57 23 75 19 10
Bubble Sort	57 23 75 19 10
Insertion Sort	57 23 75 19 10
Quick Sort	11 2 9 13 57 25 17 1 90 3
Merge Sort	11 2 9 13 57 25 17 1 90 3

14.9 Further Reading :

1. Data Structures Using "C" by Tanenbaum.
2. Data Structures Using "C" by Yashwant Kanetkar.
3. Data Structures and Program Design in "C" by Robert L. Kruse.
4. Fundamentals of Data Structures by Horowitz and Sahani.

BLOCK SUMMARY :

All the Units of this Block are extremely important learning as Unit No. 11 is 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. 12 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. 13 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 Questions :**

1. Write a note on internal and external sorting with help of example ?
2. Which sorting method is more efficient ?
3. What is indexed file structure ?
4. Discuss the efficiency of linear and binary search ?
5. What is bucket sort ?

❖ **Long Questions :**

1. Explain hashing technique ?
2. How to Calculate Hash Function ?
3. 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%20Compiler%20for%20Linux/main_for/lref_for/source_files/rfjrec.htm

❖ **Enrolment No. :** ☐

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

Unit No.	11	12	13	14
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

'Jyotirmay' Parisar,
Sarkhej-Gandhinagar Highway, Chharodi, Ahmedabad-382 481.
Website : www.baou.edu.in