



SURESH
GYAN VIHAR
UNIVERSITY
Accredited by NAAC with 'A+' Grade

Bachelor of Computer Application
(B.C.A.)

Data Structure using C

Semester-III

Author- B.J. Mohite

SURESH GYAN VIHAR UNIVERSITY
Centre for Distance and Online Education
Mahal, Jagatpura, Jaipur-302025

EDITORIAL BOARD (CDOE, SGVU)

Dr (Prof.) T.K. Jain
Director, CDOE, SGVU

Dr. Dev Brat Gupta
*Associate Professor (SILS) & Academic
Head, CDOE, SGVU*

Ms. Hemlalata Dharendra
Assistant Professor, CDOE, SGVU

Ms. Kapila Bishnoi
Assistant Professor, CDOE, SGVU

Dr. Manish Dwivedi
*Associate Professor & Dy, Director,
CDOE, SGVU*

Mr. Manvendra Narayan Mishra
*Assistant Professor (Deptt. of Mathematics)
SGVU*

Ms. Shreya Mathur
Assistant Professor, CDOE, SGVU

Mr. Ashphaq Ahmad
Assistant Professor, CDOE, SGVU

Published by:

S. B. Prakashan Pvt. Ltd.

WZ-6, Lajwanti Garden, New Delhi: 110046

Tel.: (011) 28520627 | Ph.: 9205476295

Email: info@sbprakashan.com | Web.: www.sbprakashan.com

© SGVU

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means (graphic, electronic or mechanical, including photocopying, recording, taping, or information retrieval system) or reproduced on any disc, tape, perforated media or other information storage device, etc., without the written permission of the publishers.

Every effort has been made to avoid errors or omissions in the publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice and it shall be taken care of in the next edition. It is notified that neither the publishers nor the author or seller will be responsible for any damage or loss of any kind, in any manner, therefrom.

For binding mistakes, misprints or for missing pages, etc., the publishers' liability is limited to replacement within one month of purchase by similar edition. All expenses in this connection are to be borne by the purchaser.

Designed & Graphic by : S. B. Prakashan Pvt. Ltd.

Printed at :

Syllabus

Data Structure Using C

Learning Objective

- To teach efficient storage mechanisms of data for an easy access.
- To design and implementation of various basic and advanced data structures.
- To introduce various techniques for representation of the data in the real world.
- To develop application using data structures.
- To improve the logical ability

Unit 1

Introduction to data structures: storage structure for arrays, sparse matrices, Stacks and Queues: representation and application. Linked lists: Single linked lists, linked list representation of stacks and Queues. Operations on polynomials, Double linked list, circular list.

Unit 2

Dynamic storage management-garbage collection and compaction, infix to post fix conversion, postfix expression evaluation. Trees: Tree terminology, Binary tree, Binary search tree, General tree, B+ tree, AVL Tree, Complete Binary Tree representation, Tree traversals, operation on Binary tree-expression Manipulation.

Unit 3

Graphs: Graph terminology, Representation of graphs, path matrix, BFS (breadth first search), DFS (depth first search), topological sorting, Warshall's algorithm (shortest path algorithm.)
Sorting and Searching techniques

Unit 4

Bubble sort, selection sort, Insertion sort, Quick sort, merge sort, Heap sort, Radix sort. Linear and binary search methods, Hashing techniques and hash functions.

References

- Gilberg and Forouzan: "Data Structure- A Pseudo code approach with C" by Thomson publication
- "Data structure in C" by Tanenbaum, PHI publication / Pearson publication.
- Pai: "Data Structures & Algorithms; Concepts, Techniques & Algorithms" Tata McGraw Hill. Reference Books:
- "Fundamentals of data structure in C" Horowitz, Sahani & Freed, Computer Science Press.
- "Fundamental of Data Structure" (Schaums Series) Tata-McGraw-Hill.

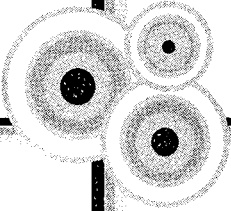
Contents

1.	Basic Concept and Introduction to Data Structure	44
1.	Pointers- An Introduction.....	1-1
1.1	Usage of Pointer 1-3	
1.2	Advantages of Pointer 1-4	
2.	Dynamic Memory Allocation.....	1-5
2.1	Dynamic Memory Allocation Functions 1-6	
3.	Algorithm Definition and Characteristics.....	1-13
4.	Algorithm Analysis.....	1-14
4.1	Time Complexity 1-14	
4.2	Space Complexity 1-15	
4.3	Asymptotic Notation 1-15	
5.	Introduction to Data Structure.....	1-17
5.1	Definition of Data Structure 1-18	
5.2	Types of Data Structures 1-19	
6.	Abstract Data Type (ADT).....	1-20
7.	Introduction to Array.....	1-21
7.1	Types of Array 1-22	
7.2	Use of Array as an Argument to the Function 1-28	
7.3	Array – As an Abstract Data Type 1-30	
7.4	Applications of Array 1-32	
8.	Polynomial.....	1-32
8.1	Polynomial Representation using Array 1-32	
8.2	Addition of Polynomials Using Array 1-34	
9.	Structure.....	1-35
9.1	Syntax to define Structure 1-35	
9.2	Accessing Members of a Structure 1-36	
9.3	Structures as Function Arguments 1-36	
9.4	Self-Referential Structures 1-37	
2.	Searching and Sorting Techniques	40
1.	Introduction.....	2-1
2.	Linear Search.....	2-1
3.	Binary Search.....	2-4
4.	Sorting.....	2-9
4.1	Sorting Techniques 2-10	
5.	Bubble Sort.....	2-11
6.	Insertion Sort.....	2-13
7.	Selection Sort.....	2-15
8.	Quick Sort.....	2-16
9.	Heap Sort.....	2-19
10.	Merge Sort.....	2-25
11.	Comparison of Sorting Methods.....	2-29
	Solved Examples.....	2-29
3.	Linked List	40
1.	Introduction.....	3-1
2.	Concept of Linked Organization.....	3-3
3.	Implementation of Linked List.....	3-4
3.1	Static Representation 3-4	
3.2	Dynamic Representation 3-6	
4.	Types of Linked Lists.....	3-7
5.	Operations on a Singly Linked List.....	3-8
5.1	Creation of a Singly Linked List 3-9	
5.2	Traversing a List 3-12	
5.3	Inserting an Element in the List 3-13	
5.4	Deleting an element from the list 3-16	
5.5	Searching an Element in the List 3-17	
5.6	Computation of Length of a Singly Linked List 3-22	
6.	Doubly Linked List.....	3-24

7.	Circular Linked List	3-29
4.	Stack and Queue	58
1.	Introduction	4-1
2.	Definition of a Stack	4-1
3.	Primitive Operations on Stack	4-2
4.	Implementation of a stack	4-3
4.1	Static Implementation of Stack	4-3
4.2	Declarations and Functions	4-4
4.3	Operations on the Stack	4-5
4.4	Dynamic Implementation of stack	4-9
5.	Applications of Stack	4-12
5.1	Recursion	4-12
5.2	Polish and Reverse Polish Notations	4-13
5.3	Interconversion between Infix, Postfix and Prefix Expressions	4-16
5.4	Matching Parentheses in an expression	4-20
6.	Queue	4-21
7.	Definition of a Queue	4-21
8.	Operations on a Queue	4-22
9.	Implementation of Queues	4-23
9.1	Static Implementation of Linear Queues	4-23
9.2	Static Implementation of Circular Queue	4-26
9.3	Dynamic Implementation of Circular Queue	4-32
10.	Types of Queues	4-34
10.1	Linear Queue	4-34
10.2	Circular Queue	4-35
11.	Priority Queue	4-35
12.	Doubly-Ended Queue (DEQUE)	4-38
13.	Applications of Queues	4-39
13.1	CPU Scheduling Algorithms	4-40
	Solved Examples	4-42
5.	Trees	44
1.	Introduction	5-1
2.	Tree Terminology	5-2
3.	Binary Trees	5-4
4.	Representation of Binary Trees	5-7
5.	Operations on Binary Tree	5-11
5.1	Creation of Binary Tree	5-11
5.2	Insertion	5-12
5.3	Deletion	5-13
6.	Traversing a Binary Tree	5-17
6.1	Pre Order Traversal	5-18
6.2	In Order Traversal	5-18
6.3	Post Order Traversal	5-19
6.4	Iterative Traversing	5-20
7.	Binary Search	5-24
7.1	Binary Trees and Binary Search Trees	5-24
8.	AVL Trees	5-27
6.	Graphs	22
1.	Graphs	6-1
1.1	Definitions and Terminology	6-1
1.2	Representation of Graph	6-6
1.3	Adjacency List Implementation	6-7
2.	Shortest Path Problem	6-9
3.	Spanning Tree	6-10
4.	Traversal of Graphs	6-15
4.1	Depth-First Search (DFS)	6-16
4.2	Breadth-First Search (BFS)	6-18
5.	Applications of Graphs	6-20

* * *

BASIC CONCEPT AND INTRODUCTION TO DATA STRUCTURE



1. Pointers- An Introduction

Programming can be defined as *set of instructions* that operates on a data to manipulate its value. This data can be divided in two types as constants and variables. The basic difference between variables and constant is their ability to change value at any point in the execution of program. Constants have fixed values while variables take different values in the execution. This can be done by providing memory space to variable in which contents can be changed.

Declaration of Variable

All variables must be declared before using them. Syntax to declare normal variable is,

```
<data type> <variable name>;
```

where, data type can be one of the basic data types of C i.e. char, int, float etc. and variable name is any valid C identifier name.

For example: Declaration of normal variable

```
int x=10;  
int y;
```

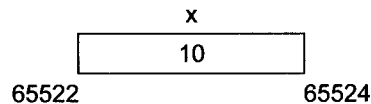
Once the variable is declared, it will be provided memory space statically, (*at compile time*) according to its data type, having address of its location and this memory location will be identified with the name of variable.

Hence, from the programmer's point of view the data can be manipulated by the name of variable while it can be manipulated by its address (memory location) in the actual execution of program.

Diagrammatically above code can be shown as

Example 1

```
int x =10;
```



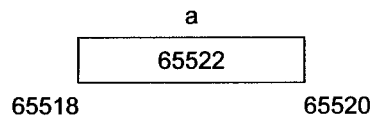
In this *example*, *x* is the normal variable having address *65522* its address of memory location and *10* is the actual value stored at this address.

As shown in the above *example* *x* is referred as normal variable as C has provided a very special type of variable *pointer*, most powerful as well as useful feature of it, which makes C different than other programming language.

Consider another *example*

Example 2

```
int a=&x;
```



In this *example*, *a* is the variable having address *65518* as its address of memory location and *65522* is the actual value stored at this address.

So, if we relate *example 1* and *example 2*, it can be easily observed that, variable *a* has stored the address of *x* as its value. This simple thing makes variable *a* to be treated as pointer variable rather than normal variable.

Hence pointer can be defined as follows:

Pointer is variable that stores memory address of another variable.

1.1 Usage of Pointer

Syntax to declare pointer variable

```
<data_type>*<variable_name>;
```

Where data type can be one of the basic data types of C i.e. char, int, float etc. and variable name is valid C identifier name.

Declaration of pointer variable

```
int *a;
```

As defined above, pointer can store the address of another normal variable as its value

Syntax to initialization of pointer variable:

- i. Initializing pointer variable with address of normal variable-

```
Pointer_Variable = &Normal_Variable;
```

For example: a = &x;

The unary or monadic operator and gives the 'address of a variable'.

- ii. Initializing pointer variable with array

```
Pointer_Variable = Array_Name; Or
```

```
Pointer_Variable = &Array_Name[0];
```

For example: int x[10];

```
a = x; or a = &x[0];
```

In both cases the starting address of array is assigned to pointer variable.

- iii. Initializing pointer variable with character string -

```
char *Pointer_Variable = <"String Constant">;
```

For example: char *p= "Vision";

As C programming has not provided separate data type for string, it can be declared either by using array or using character pointer as shown in the above example.

The value stored at normal variable now can be retrieved by two methods

1. Using normal variable itself
2. Using pointer variable.

To retrieve information from pointer variable

```
x = *a;
```

The indirection or dereference operator * gives the "contents of an object pointed to by a pointer".

2

Oct.15, Apr.15 – 2M

What is use of (&) address operator and dereferencing (*) operator?


```
/*Program to demonstrate pointer variable */
#include<stdio.h>
#include<conio.h>
void main(void)
{
    int x=10;          /* normal variable declaration */
    int * a;          /* pointer variable declaration */
    clrscr();         /* to clear screen */
    a=&x;              /* to assign address of x to a */
    printf("\n Address of x=%u",&x);    /* address of x */
    printf("\n Address of a=%u",&a);    /* address of a */
    printf("\n Value at x=%d",x);      /* value at x */
    printf("\n Value at a=%u\n",a);    /* value at a */
    printf("Value at x using pointer variable a=%d",*a); /* value at x */
    getch();
}
```

Output

Address of x = 65522

Address of a = 65520

Value at x = 10

Value at a = 65522

Value at x using pointer variable a = 10

1.2 Advantages of Pointer

- i. Pointers can be used to pass arguments with *pass by address* method.
- ii. Similarly pointers can be used to pass *array* and *string* as an argument to the function.
- iii. One of the most important features of pointer is that it can be used in *dynamic memory allocation* to allocate and release memory dynamically (*at run time*).
- iv. As pointer holds the address of another variable, data manipulation is done with address to get execution faster.
- v. Pointers are more efficient to handle data structures like *linked list*, *trees* and *graphs*.

- vi. Pointer of *void* data type can be used to store addresses of variables having different data types.

For example:

```
void *ptr;
char ch;
int a;
float b;
ptr=&ch; /* is valid */
ptr=&a; /* is valid */
ptr=&b; /* is valid */
```

- vii. The use of pointer to character string results in saving of data storage space in memory.
- viii. A pointer reduces length and complexity of a program.

2. Dynamic Memory Allocation

As stated earlier variables can be declared before using them. This method of declaring variable is called as static memory allocation. Hence the memory allocated statically (at compile time) must be used during the execution of program without increasing or decreasing it as per requirement.

Consider the following example,

Suppose you have invited some people on marriage party and you have ordered caterer to prepare food of 25 people. So food can be consumed when exact 25 people will arrive. But what is the case if only 20 people have come. The food of 5 people will be just wasted, and what if 30 people come, 5 of them should find some other resource as there is insufficient food.

Same problem can be considered in case of programming.

As we know, in C programming when we declare array, its size should be mentioned.

For example

```
int arr[25];
```

With this array the memory for exact 25 elements will be declared. But, in most of the cases the user does not know the number of elements to be entered. So, in above example if user entered 20 elements, it will be wastage of memory for 5 elements; generally known as *underflow*. In another case program will not allow user to enter 26th element as it will be out of capacity of an array. This situation is generally known as *overflow*. So as to make user free from both the situations, the memory should be allocated according to the requirement of user at run time (*dynamically*), instead

of declaring it at compile time (*statically*). This technique of Dynamic allocation is a pretty unique feature to C amongst high level languages. It enables user to create data types and structures of any size and length by allocating and de-allocating memory whenever required to suit the need of the program.

Allocating memory space at run time is called as Dynamic Memory Allocation.

2.1 Dynamic Memory Allocation Functions

5

Apr.15 Oct.14 – 4M

Explain different types of dynamic memory allocation functions.

Oct.2012 – 4M

Explain dynamic memory allocation functions with their syntax.

Apr.10, Oct.10 – 2M

Explain in brief the functions of dynamic memory allocation.

The C programming language has provided some *built-in functions* for the dynamic memory allocation. These functions are grouped in a header file *alloc.h*, and those are as follows:

- i. **sizeof()**: sizeof is a unary operator, which gives size of its argument in terms of bytes. Any data type (*int, float, char*), variables, array or even structures can be sent as an argument to this.

For example: sizeof(float);

This gives the bytes occupied by the float data type that is 4.

/ Program to understand the sizeof operator */*

```
#include<stdio.h>
main()
{
    struct
    {
        char Name[10];
        int RollNo;
    }S;
    int No[10];
    printf(" size of structure = %d",sizeof(S));
    printf("\n size of float = %d",sizeof(float));
    printf("\n size of array = %d",sizeof(No));
}
```

Output

size of structure = 12

size of float = 4

size of array = 20

- ii. **malloc()**: This function is used to allocate a memory for required number of bytes. The memory created by malloc at run-time should be assigned to the pointer to get starting address of it.

The prototype of the malloc function is as follows:

```
void * ptr = malloc(size);
```

Where *size* tells the specified number of bytes of memory that should be allocated and *ptr* is pointer, which points to the starting memory address that has been allocated.

For example

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

This allocates 20 bytes of memory space and its starting address will be assigned to the integer pointer ptr.

Program to demonstrate malloc function

/ Program to accept n number and display those numbers in reverse order */*

```
#include<stdio.h>  
#include<conio.h>  
#include<alloc.h>  
void main(void)  
{  
    int *num;  
    int n,i;  
    clrscr();  
    printf("\n Enter how many numbers you want to enter :");  
    scanf("%d",&n);  
    /* dynamic memory allocation function */  
    num =(int *) malloc(n * sizeof(int));  
    for(i=0; i<n; i++)  
    {  
        printf("Enter Number:");  
        scanf("%d", num);  
        num++; /* to move pointer to next location */  
    }  
    printf("You have accepted following number in reverse order:");  
    for(i=0; i<n; i++)  
    { num --; /* to move pointer to previous location */  
      printf("\n %d", *num);  
    }  
    getch();  
}
```

1

Oct.2011 – 2M
What is the difference
between Malloc and
Calloc?

Output

Enter how many number you want to enter: 5

Enter Number: 10

Enter Number: 20

Enter Number: 30

Enter Number: 40

Enter Number: 50

You have accepted following number in reverse order:

50

40

30

20

10

- iii. **calloc()**: This function is used to allocate a memory in multiple blocks for required number of bytes to same type of object. The memory created by calloc at run-time should be assigned to the pointer to get starting address of it.

The prototype of the calloc function is as follows:

```
void * ptr = calloc(rec, size);
```

Where rec tells total number of blocks where each block is having memory specified by size and ptr is pointer which points to the memory that has been allocated.

Generally calloc() function is used to allocate memory for array and structure.

For example

```
struct stud
{
    int roll_no;
    char name[20];
    float percent;
};
struct stud *ptr;
ptr = (struct stud*)calloc(20, sizeof(struct stud));
```

This allocates 20 blocks for structure student and each block will be having memory space required by the object of structure i.e. 26 bytes. Program to demonstrate calloc function.

```
/* Program to accept n numbers and display addition of numbers */
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main()
{
    int *num;
    int n,i,s=0;
    clrscr();
    printf("\n Enter how many numbers you want to enter :");
    scanf("%d",&n);
    /* dynamic memory allocation function */
    num=(int *) calloc(n, sizeof(int));
    for(i=0; i<n; i++)
    {
        printf("Enter Number:");
        scanf("%d", num);
        s=s+*num;
        num++; /* to move pointer to next location */
    }
    printf("\n Sum of given number = %d ", s);
    getch();
}
```

Output

Enter how many numbers you want to enter: 5

Enter Number: 10

Enter Number: 20

Enter Number: 30

Enter Number: 40

Enter Number: 50

Sum of given number = 150

- iv. **realloc()**: In dynamic memory allocation, sometimes such situations can arise where either we want to increase or decrease the memory space, allocated using `malloc()` or `calloc()` function. So in such situations, previous allocation should be changed and memory should be reallocated to fulfill the current need of user. The memory can be reallocated using `realloc()` function.

The prototype of the function is as follows:

```
void *ptr = realloc(ptr, new_rec_size);
```

Where `new_rec_size` tells the new required size of memory in terms of bytes and `ptr` is pointer which points to the memory that has been previously allocated.

In `realloc()` function, the main point to be noted is that, previous starting address of pointer can be replaced with the new address provided by the function as per availability. In other words the old memory block (*allocated by `malloc()` or `calloc()`*) will be replaced with new memory block (*allocated by `realloc()`*)

But what about data that is stored at the old memory blocks? Interestingly, the contents of old memory block will not be lost rather contents of old memory block will be preserved by copying them into new memory block.

Consider the following example

```
ptr = malloc(10);
```

The pointer `ptr` will be allocated with the starting address of allocated memory block of 10 bytes.

```
ptr = realloc(ptr, 20);
```

The pointer `ptr` can be allocated with new starting address of new memory block of 20 bytes other than previous address but at the same time contents of previous block will be kept as it is after copying them into new block.

Program to demonstrate `realloc` function.

/ Program to accept n numbers and display them in reverse order */*

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main(void)
{
    int * num;
    int n1,n2,i;
    clrscr();
    printf("\n Enter how many numbers you want to enter :");
    scanf ("%d",&n1);
    /* dynamic memory allocation using calloc function */
    num=(int *) calloc(n1,sizeof(int));
    for(i=0; i<n1; i++)
```

```
{
    printf(" Enter Number:");
    scanf("%d",num);
    num++; /* to move pointer to next location */
}
printf("How many extra number you want to enter:");
scanf("%d",&n2);
num=(int *) realloc(num,(n1+n2)*sizeof(int));
for(i=n1;i<n1+n2;i++)
{
    printf(" Enter Number:");
    scanf("%d",num);
    num++; /* to move pointer to next location */
}
printf("You have accepted following number in reverse
order :");
for(i=0;i<n1+n2;i++)
{
    num--; /* to move pointer to previous location */
    printf("\n %d",*num);
}
getch();
}
```

Output

Enter how many numbers you want to enter: 3

Enter Number: 10

Enter Number: 20

Enter Number: 30

How many extra numbers you want to enter: 2

Enter Number: 40

Enter Number: 50

You have accepted following number in reverse order:

50

40

30

20

10

- v. **free()**: To use memory very efficiently, it should be released, if not required. The **free()** function should be used to release the memory space allocated to pointer variable. The **free()** function normally releases the memory space created using **malloc()** and **calloc()** function.

For example: `free(ptr)`

Where ptr is the pointer of which memory has to be released, that was allocated using malloc() or calloc() function.

Program to demonstrate free function

```
/*Program to accept n numbers and display addition of number*/
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main(void)
{
    int * num;
    int n, i, s=0;
    clrscr();
    printf("\n Enter how many numbers you want to enter :");
    scanf("%d", &n);
    /* dynamic memory allocation function */
    num=(int *) calloc(n, sizeof(int));
    for(i=0; i<n; i++)
    {
        printf(" Enter Number:");
        scanf("%d", num);
        s=s + *num;
        num++; /* to move pointer to next location */
    }
    printf("\n Sum of given number = %d ", s);
    free(num);
    /* free the memory of variable num allocated by calloc
    function */
    getch();
}
```

Output

```
Enter how many numbers you want to enter: 5
Enter Number: 10
Enter Number: 20
Enter Number: 30
Enter Number: 40
Enter Number: 50
Sum of given number = 150
```

Note: The memory of 10 byte size allocated to pointer *num* will be released after the execution of free function.

3. Algorithm Definition and Characteristics

Algorithm: It is a method of solving a problem. It is a sequence of instructions that acts on some input data to produce some output in a finite number of steps.

An algorithm can be specified in many ways. *For example*, it can be written down in English or any other natural language. However, algorithms can be precisely specified using an appropriate mathematical format like a programming language. Obviously the behavior of an algorithm can be observed with the help of program which can be defined as an implementation of an algorithm.

If we run a program, implementing an algorithm, on a particular computer with a particular set of inputs, then behavior of the program is according to the single instance related with set of inputs and computer used.

Definitions

- i. *Algorithm is a set of rules or instructions that specify how to solve a particular problem or task.*
- ii. *Algorithm is a step-by-step procedure for accomplishing a task.*
- iii. *Algorithm is a list of instructions to solve a particular problem.*

Properties

An algorithm must have following properties:

- i. **Input:** Algorithm must receive some input data.
- ii. **Output:** Algorithm must produce at least one output as the result.
- iii. **Finiteness:** No matter what is the input, the algorithm must terminate after a finite no. of steps.
- iv. **Definiteness:** The steps to be performed in the algorithm must be clear and unambiguous.
- v. **Effectiveness:** One must be able to perform the steps in the algorithm without applying any intelligence.

1

Apr.2011 – 2M
Define Algorithm.

1

Oct.2012 – 2M
What is Algorithm? State its properties.

1

Oct.2015 – 4M
What is algorithm?
Explain its characteristics in detail.

4. Algorithm Analysis

In order to learn about the behavior of an algorithm, we can 'analyze' it by studying the specification of the algorithm and drawing conclusions about how the implementation of that algorithm can be made general for any computer for any set of input.

The analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them.

1**Apr.2015 – 4M**

What is an algorithm?
How to measure its performance?

Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense i.e. to estimate the complexity function for arbitrarily large input.

An algorithm can be analyzed by determining

- i. The running time of a program as a function of its inputs;
- ii. The total or maximum memory space needed for program data;
- iii. The total size of the program code;
- iv. Whether the program correctly computes the desired result;
- v. The complexity of the program--*example*, how easy is it to read, understand, modify;
- vi. The robustness of the program--*example*, how well does it deal with unexpected inputs?

But, primarily the analysis of algorithm is concerned with the running time and the memory space needed to execute the program. The time and space requirements, referred as *time and space complexity of an algorithm*, enable us to measure how efficient it is.

2**Oct.2014 – 2M**

What is time complexity?
How it is calculated?

Oct.2011 – 2M

Define time Complexity?

4.1 Time Complexity

Time complexity can be defined as running time of the program. There are many factors that affect the running time of a program which can be - the algorithm itself, the input data, and the computer system used to run the program.

The time complexity cannot be measured by calculating the time using the computer clock, because:

Program may also wait for I/O or other resources.

While running a program, a computer performs many other computations.

So, we must use some abstract notation to calculate time complexity. Generally the running time of an algorithm is proportional to the number of steps it takes to execute the algorithm.

In such theoretical analysis of algorithms it is common to estimate their complexity in *asymptotic sense*, i.e., running time can be defined as a function of the size of the input data which is noted as 'Big-O Notation'.

3

Apr.12, 10, Oct.10 – 2M

How to measure performance of an Algorithm?

4.2 Space Complexity

Space complexity can be defined as amount of computer memory required during the program execution. The space complexity also cannot be exactly measured, but can be calculated by considering data and its size.

So, space complexity can also be calculated exactly like time complexity i.e. using 'Big-O Notation' of the size of the items which require maximum storage in given data.

1

Oct.2015 – 2M

What is space complexity? How is it calculated?

4.3 Asymptotic Notation

To choose the best algorithm we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm.

Asymptotic notation is a shorthand way to represent the time complexity.

Using asymptotic notations we can give time complexity as 'fastest possible' 'shortest possible' or 'average time' various notations such as Ω , θ and O used are called asymptotic notations.

2

Oct.15, Apr.15 – 5M

Explain different types of Asymptotic notations in detail.

- i. **Big-O Notation:** Big-O notation is considered as a property of the algorithm. Big-O notation is used as functions, which specify the amount of resources consumed by an algorithm, when the input to the algorithm is of size N . This function is usually denoted by $O(N)$ where $O(N)$ is the amount of the resource, usually time or space of some specific operation consumed. Generally, the time required by an algorithm can be calculated as time required per basic step.

For *conditional*, we count number of basic steps on the branch that is executed i.e. number of statements either in if block or else block.

For a *loop*, we count number of basic steps in loop body i.e. number statements in looping statement block, times the number of iterations.

For a *method*, we count number of basic steps in method body.

Hence, efficiency of particular algorithm can be measured for a particular size of input (N), but when N increases and it becomes larger, then same algorithm behaves differently. So Big-O Notation defines order for growth.

While calculating the Complexity with the Big-O Notation

1. Simplify the basic steps and
2. Choose the highest term.

For example, consider the job offers from two companies. The first company offers a contract that will double the salary every year. The second company offers you a contract that gives a raise of Rs.1000/- per year.

Hence salary at the first company increases at the rate of 2^n

New salary = Salary $\times 2^n$ (where n is total service years)

The highest term in this equation is 2^n which can be denoted as $O(2^n)$ in Big-O notation.

While salary at the second company increases at a rate of 1000n.

New salary = Salary + 1000n (where n is total service years)

The highest term in this equation is 1000n which can be denoted as $O(n)$ in Big-O notation.

Hence from above *example* we can denote 'Big-O Notation' as a function of the size of the input data as follows.

$$f(n) = O(g(n))$$

Where $f(n)$ represents the computing time of some algorithm $O(g(n))$ which takes time not more than constant $g(n)$.

Commonly Seen Time Bounds		
$O(1)$	Constant	Excellent
$O(\log n)$	Logarithmic	Excellent
$O(n)$	Linear	Good
$O(n \log n)$		Good
$O(n^2)$	Quadratic	OK
$O(2^n)$	Exponential	Too slow

Big - O Examples
Let $f(n) = 3n^2 + 6n - 7$ Claim $f(n) = O(n^2)$
Let $f(n) = 4n \log n + 34n - 89$ Claim $f(n) = O(n \log n)$
Let $f(n) = 20 * 2^n + 40$ Claim $f(n) = O(2^n)$
Let $f(n) = 34$ Claim $f(n) = O(1)$

- ii. **Omega Notation (Ω):** $f(n) = \Omega(g(n))$ iff there exists a positive integer n_0 and a positive number M such that $|f(n)| \geq M |g(n)|$, for all $n \geq n_0$.

Example:

a. $1.3n + 2 = \Omega(n)$ as $3n + 2 \geq 4n$ for all $n \geq 1$.

- iii. **Theta Notation (θ):** $f(n) = \theta(g(n))$ iff there exist two positive constants c_1 and c_2 and a positive integer n_0 such that,

$$c_1 |g(n)| \leq f(n) \leq c_2 |g(n)| \text{ for all } n \geq n_0.$$

for all values of n , $n \geq n_0$. In other words, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Example:

a. $3n + 2 = \theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$. and

$$3n + 2 \leq 4n \text{ for all } n \geq 2.$$

$$\text{So } c_1 = 3, c_2 = 4 \text{ and } n_0 = 2.$$

- iv. **Little Oh Notation (o):** *Definition:* $f(n) = o(g(n))$ (f of n is little oh of g of n) iff $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

$$0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

Example:

For $f(n) = 18n + 9$, we have $f(n) = O(n^2)$ but $f(n) \neq \Omega(g(n))$. Hence $f(n) = o(n^2)$.

5. Introduction to Data Structure

Computer science can be defined as the study of data, its representation and transformation. Once data is stored in the form of bits, it has to be accessed and manipulated many times. To do so, there must be inbuilt mechanisms to access and store data.

Each programming language provides a set of built in data types, which allows data to be stored in a meaningful format. The language also provides a set of operations to manipulate this data.

However, these data types are not enough, since present day programming problems are complex and large. Thus, there is a need for a structured data type, which may be a combination or collection of basic data types with a set of properties and legal operations that may be performed on it. This is called the conceptual definition of the data type or the **Abstract Data Type (ADT)**.

The next stage is the **Implementation** stage where the ADT is implemented by hardware or software methods.

Depending upon the application, the data type has to be chosen and careful thought has to be given to how the data is to be stored so that there is efficient storage, convenient and faster retrieval and manipulation. This is called **time and space** consideration.

5.1 Definition of Data Structure

Before we proceed, there are several terms that we need to define. There is no standard definition for these terms and they are often used interchangeably.

Data Object

Data object refers to a set of elements (D) which may be finite or infinite.

If the set is infinite, we have to devise some mechanisms to represent that set in memory since available memory is limited.

Example:

A set of integer numbers is infinite

$$D = \{ 0, \pm 1, \pm 2, \dots \dots \dots \}$$

A set of alphabets is finite,

$$D = \{ 'A', 'B', \dots, 'Z' \}$$

Data Types

Data-type is a term used to describe the information type that can be processed by the computer and which is supported by the programming language.

It is also defined as a term which refers to the kinds of data that variables may 'hold' in a programming language.

Example

int, char, float, etc.

Some languages also allow users to combine built-in data types.

Example

record in Pascal, structures and unions in C.

Data Structure

A data structure consists of data objects, their properties and the set of legal operations, which may be applied to the elements of the data object.

Definition

A data structure is a set of domains D , a designated domain $d \in D$, a set of functions F and a set of axioms A . The triple (D, F, A) denotes the data structure.

This definition is also called the **Abstract Data Type**.

D - Denotes the data objects

F - Denotes the set of operations that can be carried out on the data objects.

A - Describes the properties and rules of the operations.

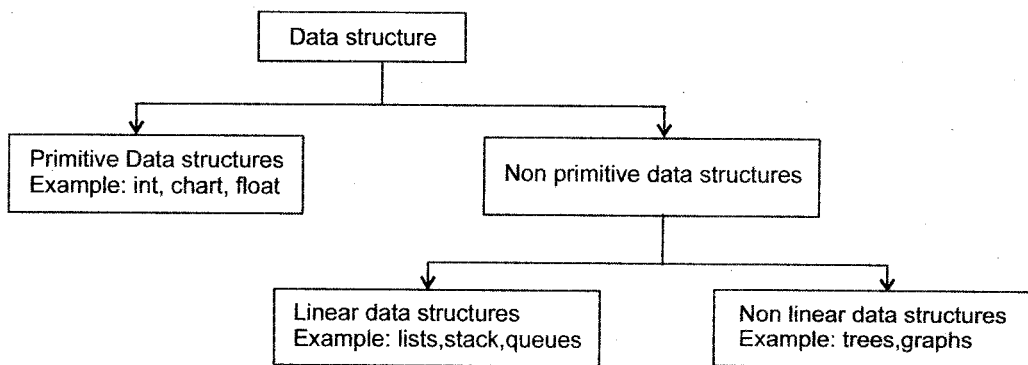
2

Apr.15, Oct.14 – 2M

What are the different types of data structures?

5.2 Types of Data Structures

The data structure can be divided into two basic types: Preliminary data structure and secondary data structures.



A set of primitive elements which do not involve any other elements as its subparts, is called as 'Primitive Data Structure'.

For example, int, char, float, double.

The non primitive data structures are the data structure which are basically derived from primitive data structures. They can be further categorized into linear data structure and non linear data structures.

1

Apr.2011 – 2M

What is primitive data structure?

Linear data structures are the data structures in which data is arranged in a list or in a straight sequence.

For example: arrays, lists

Non linear data structures are the data structures in which data may be arranged in a hierarchical manner.

6. Abstract Data Type (ADT)

1

Oct.2009 – 2M

Define ADT (Abstract Data Type).

In computer science, programming basically depends upon the data, its representation and manipulation i.e. the method by which the bits of data are accessed. Each programming language provides inbuilt mechanism to store data. The data can be distinguished in different data types like integer, floating or character type. Hence programming provides meaningful format for data and also set of operations to manipulate this data.

For example, when we declare a variable, say *x*, of type *int*, we know that *x* can represent an integer in the range of 16-bits and we can perform operations on *x* such as addition, subtraction, multiplication, and division. In case of data type *int* we don't need to know how integers are represented nor how the operations are implemented to be able to use them.

But, these basic data types are not useful in case of very large and complex programs, where we need combination of basic data types having set of properties and operations that can be performed on it, which can be called as *data structure* or *abstract data type*.

In general terms, an *abstract data type* can be considered as set of values and the operations performed on it with some rules or properties on the data.

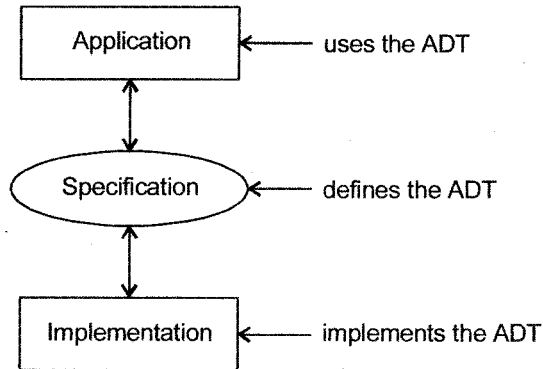
When we use *abstract data type*, our programs are divided into:

The Application

The part that uses the abstract datatype.

The Implementation

The part that implements the abstract datatype.



The *Abstract Data Type* is a type of a variable which specifies three sets:

1. A set of values
2. A set of properties; and,
3. A set of operations.

By organizing our program this way i.e. by using abstract data types (its values and operations) without referring as to how it will be implemented. Programs that use such data type only know the implementation of the set of values but make use of the operations defined abstractly without knowing their implementation.

7. Introduction to Array

Any programming language provides different terms to develop a program like identifiers, keywords, constants and data types. Similarly, C programming language has come up with some of its unique terms to facilitate programmers, *for example* C programming language has provided two categories of data types as primary data type and secondary data type where *data type is referred to as a type of data that can be held by variable or constant*.

As most of the programming language has provided primary data types, C programming language has provided following secondary data types.

Secondary Constants / Data Type

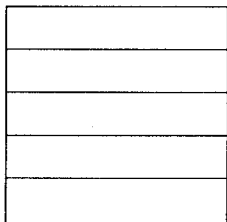
- i. **Array constant:** Used to store same type of data in continuous memory location under unique name.
- ii. **Pointers:** Pointer is a variable, which holds memory address of another variable.
- iii. **Structure:** Used to store/hold different types of data and allocates memory of all variables declared as structure data type.
- iv. **Union:** Used to store/hold different type of data and allocates memory of maximum size variable.

7.1 Types of Array

One-Dimensional Array

A one-dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner. A list of items having one variable name, one subscript is known as an One-dimensional array. In C single subscripted variable x_i can be expressed as $x[0]$, $x[1]$, $x[2]$,..... $x[n]$ e.g., to represent a set of five numbers 11, 22, 33, 44, 55 by an array variable number, declare the variable number as follows:

`int number[5];` and computer reserves five storage locations as shown below:



number[0]
number[1]
number[2]
number[3]
number[4]

The values to the array elements can be assigned as follows :

number[0]=11
number[1]=22
number[2]=33
number[3]=44
number[4]=55

Thus array number stores the values as shown below:

11	number [0]
22	number[1]
33	Number[2]
44	number[3]
55	number[4]

Storage Representation of One Dimensional Array

Declaration of array

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

Syntax: `type array-name[array_size];`

Here type specifies the type of element that will be contained in the array such as int, float or char and array-size specifies the maximum number of elements that can be stored inside the array.

Example,

```
float height[50];
```

declares height to be an array which contain a maximum of 50 real numbers.

```
int group[10];
```

declares group to be an array which contain a maximum of 10 integer numbers.

`char name[10];` declares name as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read a string "WELL DONE" into an array name.

Then each character of the string in an array name is stored in memory as follows.

'W'	'E'	'L'	'L'		'D'	'O'	'N'	'E'	'\0'
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

When the compiler sees a character string, it terminates it with an additional null character. Thus element[9] holds the null character '\0' at the end. So when declaring character array, we must always allow an extra element space for the null terminator.

Initialization of Array

You can initialize C++ array elements either one by one or using a single statement as follows

Syntax: `type array-name [array_size]={list of values};`

Values in the list are separated by commas. The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets []. If you omit the size of the array, an array just big enough to hold the initialization is created.

Example,

```
int number[3]={0,0,0};
```

```
double balance[ ]={50.45,65.85,70.42,45.60}; //The size can be omitted.
```

```
char name[]={'d', 'c', 'm'}; //Character array can be initialized in the similar manner.
```

You can assign the individual values in an array.

For example, `number [3] =80;`

Accessing Array Elements

After initializing an array, its elements are counted from left to right. Each element of the array, also called a member of the array, has a specific and constant position. The position of an item is also called its index. The first member of the array, the most left, has an index of 0. The second member of the array has an index of 1. Since each array has a number of items which can be specified as n , the last member of the array has an index of $n-1$. Based on this system of indexing, to locate a member of an array, use its index in the group. An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

For example,

```
int rollno=number[3];
```

The above statement will take 3rd element from the array and assign the value to rollno variable.

Once you can locate a member of the array, you can display its value using cout.

Here is an example program:

Create one dimensional array and access each element of array separately.

```
#include<iostream.h>
int main()
{
    double rollid[]={11.22,22.33,33.44,44.55,55.66};
    cout<<"Rollid 1:"<<rollid[0]<<endl;
    cout<<"Rollid 2:"<< rollid[1]<<endl;
    cout<<"Rollid 3:"<< rollid[2]<<endl;
    cout<<"Rollid 4:"<<rollid[3]<<endl;
    cout<<"Rollid 5:"<<rollid[4]<<endl;
    return 0;
}
```

Create one dimensional array and access each element of array by applying for loop.

```
#include<iostream.h>
int main()
{
    double rollid[] = {11.22, 22.33, 33.44,44.55,55.66};
    for(int i=0;i<=4;i++)
    {
        cout<<"Rollid["<<i<<"]="<<rollid[i]<<endl;
    }
    return 0;
}
```

Two-Dimensional Array

An array having two dimensions (sizes) is known as a Two-dimensional array. It is a collection of data elements of same data type arranged in rows and columns (that is, in two dimensions).

Declaration of Two dimensional arrays

A two dimensional array can be thought as a table which will have x number of rows and y number of columns.

Syntax: type array_name[row-size] [column_size];

Here type specifies the type of element that will be contained in the array such as int, float or char and row-size specifies the number of rows and column-size specifies the number of columns, array-name is the name of table or, matrix.

Oct.2014 – 2M

Give the formulae for address calculation for row and column major representation?

1

A 2-dimensional array 'a' which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Storage representation of two dimensional array

Thus, every element in array 'a' is identified by an element name of the form $a[i][j]$, where 'a' is the name of the array, and i and j are the subscripts that uniquely identify each element in 'a'.

Example: int a[3][3]; //A matrix of 3-rows and 3-columns

Initialization of Two-dimensional arrays

A two-dimensional array can be initialized along with declaration. For two-dimensional array initialization, elements of each row are enclosed within curly braces and separated by commas. All rows are enclosed within curly braces.

Syntax: int a[2][3]={0,0,0,1,1,1};

It initializes the elements of the first row to zero and second row to one.

int a[2][3]={{0,0,0},{1,1,1}};

Example: A matrix of 3×3 is initialize by

int a[3][3]={11,22,33,44,55,66,77,88,99};

Col-0	Col-1	Col-2	
↓	↓	↓	
A[0][0]	A[0][1]	A[0][2]	← Row-0
11	22	33	
A[1][0]	A[1][1]	A[1][2]	← Row-1
44	55	66	
A[2][0]	A[2][1]	A[2][2]	← Row-2
77	88	99	

Two-dimensional array are stored in memory

Accessing two-dimensional array element

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array.

For example,

```
int val=a[1][2]; //Statement will take 3rd element from the 1st row
of the array.
```

Multidimensional Arrays

As discussed in Single dimensional array, the pair of [] used to mention size of array, is considered as dimension of it. As such array is using only single pair of square braces it is called as single dimensional array. So, array having multiple pairs of square brackets are termed as Multidimensional array, or Multidimensional arrays can be described as 'arrays of arrays'.

For example, a two-dimensional array can be declared as-

```
int Mat[3][5];
```

This can be represented as a two-dimensional table as follows

		0	1	2	3	4
Mat	{	0				
		1				
		2				

Mat represents a two-dimensional array of 15 elements of type int, arranged in 3 rows and 5 columns as shown in above. Hence declaration syntax of two dimensional array is -

```
Data_type Array_name [Row] [Col];
```

Where, *Data_type* represent type of data that can be stored in array named '*Array_name*', which is any valid identifier name and Row and Col are the integer values, which represents total number of rows and columns into which the elements can be arranged logically.

/* Program to explain Matrix Multiplication using two dimensional array */

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i, j, mata[3][3],matb[3][3];
    void matmul(int mata[3][3], int matb[3][3]);
    printf("Enter elements of first 3 x 3 matrix \n" );
    for(i=0; i<3;i++)
```

```
{
for(j=0;j<3;j++)
{
scanf("%d", &matb[i][j]);
}
}
printf("Enter elements of second 3 × 3 matrix \n");
for(i=0; i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%d", &matb[i][j]);
}
}
matmul(mata,matb); /* array as an argument */
getch();
}
void matmul(int first[][[]], int second[][[]])
{
int matmul[3][3],i,j,k;
for(i=0; i<3;i++)
{
for(j=0;j<3;j++)
{
matmul[i][j]=0;
for(k=0;k<3;k++)
{
matmul[i][j]= matmul[i][j]+first[i][k]*second[k][j];
}
}
}
}
printf("Multiplication of entered 3 × 3 matrix \n") ;
for(i=0; i<3;i++)
{
for(j=0;j<3;j++)
{
printf("%d\t"matmul[i][j]);
}
}
printf("\n");
}
```

Output

Enter elements of first 3 × 3 matrix

```
2 3 4
4 2 1
5 3 2
```


Enter elements of second 3×3 matrix

```
3   2   5
6   2   1
6   3   5
```

Multiplication of entered 3×3 matrix

```
48  22  33
30  15  27
45  22  38
```

7.2 Use of Array as an Argument to the Function

In some cases we may need to pass an array to a function as an argument. In C programming it is not possible to pass a complete block of memory of an array, as a parameter to a function, with pass by value, but we can pass its address.

In order to pass array as a parameter, we have to declare the function by specifying the element type of the array, name of array and a pair of empty brackets [] in its parameters. *For example*, the following function:

```
void sum(int arr[])
```

Where, *sum* is a function name of void data type, which accepts array as a parameter of type int having name array name *arr*.

The syntax can be as follows:

```
return_type function_name(data_type name[])
```

Where, *return_type* is a valid type (like int, float...), *function_name* is a valid identifier and *data_type* is also a valid data type of array (like int, float...), *name* is any array name i.e. any valid identifier, square brackets[]; specifies the array as a parameter.

/ Program to accept 5 numbers using array and display it along with its sum.*/*

```
#include<stdio.h>
int sum(int arr[])
{
    int I, s=0;
    for (I=0;I<5;I++)
    {
        s+=arr[I];
    }
}
```

```

return s;
}
void main()
{
    int I, m, No[5];
    printf("Enter 5 Numbers \n");
    for(I=0;I<5;I++)
    {
        scanf("%d", &No[I]);
    }
    printf("Entered Numbers are \n");
    for(I=0;I<5;I++)
    {
        printf("%d\n", No[I] );
    }
    m=sum(No); /* also we can pass array as m=sum(&No[0]); */
    printf("Sum of entered numbers are ");
    printf("%d", m);
    getch();
}

```

No[0]	No[1]	No[2]	No[3]	No[4]
2	5	6	7	3

100

102

104

106

108

110

Output

Enter 5 numbers

2

5

6

7

3

Entered numbers are

2

5

6

7

3

Sum of entered numbers are 23

The parameter (int arr[]) accepts an array of any length and whose elements are of type int. The elements of an array can be accessed according to index using the for loop.

7.3 Array – As an Abstract Data Type

The *Abstract Data Type* is a type of a variable which specifies three sets:

- i. A set of values
- ii. A set of properties; and,
- iii. A set of operations.

Generally, abstract data type is a collection of elements having some properties on it, and operations to manipulate the data according to those properties.

The properties are related with the storage and retrieval of elements from the collection at specific position. Hence, as array being a collection of elements of same data type, it can also be considered as *Abstract Data Type*.

Whenever collection of elements is considered, the operations are related with storing and retrieving the elements into it and properties are related with the rule for the position at which either we can store the element or retrieve the element. Hence array, collection of elements having store and retrieval as its operations, is a strong competitor for being an *Abstract Data Type*. We can retrieve or store element at any position in an array without any specific rule, it is not having any set of properties to follow.

Hence, array as an Abstract Data Type (ADT) can be defined as follows:

Set of Values for Array

A set of values for array are array index and value at specified index. i.e. (index, value), where for each value of an array has its own index.

Set of Operations/ Functions for Array

- i. **Item Retrieve (Arr, i):** Where, Retrieve is a function name having return type item i.e. data type of array. if 'i' is the valid index of an element in an array 'Arr', which should be in the range from 0 to *Array_size - 1*, then this function returns the value of element associated with index 'i' in array 'Arr' else returns error as invalid index.
- ii. **Store (Arr, i, x):** Where, Store is a function name. if 'i' is the valid index of an element in an array 'Arr', which should be in the range from 0 to *Array_size - 1*, at which the value 'x' has to be store else returns error as invalid index.

```
/* Program to accept 10 numbers from user and sort them using Pointer */
#include<stdio.h>
#include<conio.h>
void main()
{
```

2

Oct.2012 – 2M

What is ADT for an Array?

Apr.2011 – 2M

Define Array. Give its ADT.

```
int i,j;
int no[10];
int *a;
for(i=0;i<10;i++)
{
printf("Enter Value of no[%d]\t",i);
scanf("%d",&no[i]);
}
printf("Values Before Sortintg \n");
for(i=0;i<10;i++)
{
printf("%d---",no[i]);
}
a=no;
for(i=0;i<10;i++)
{
for(j=i;j<10;j++)
{
if(*(no+i)>*(no+j))
{
swap(no+i,no+j);
}
}
}
printf("\n After Sorting \n");
for (i=0;i<10;i++)
{
printf("%d---",no[i]);
}
getch();
}
swap(int *b, int *c)
{
int temp;
temp=*b;
*b=*c;
*c=temp;
}
}
```

Output

```
Enter Value of no[0]    5
Enter Value of no[1]    3
Enter Value of no[2]    12
Enter Value of no[3]    33
Enter Value of no[4]    1
Enter Value of no[5]    13
Enter Value of no[6]    45
Enter Value of no[7]    36
```

Enter Value of no[8] 68

Enter Value of no[9] 34

Values Before Sorting

5---3---12---33---1---13---45---36---68---34---

After Sorting

1---3---5---12---13---33---34---36---45---68---

7.4 Applications of Array

Array can be used in numerous applications like

- i. Storing numeric data lists
- ii. Storing character strings
- iii. Manipulating matrix using two-dimensional type.
- iv. Implementing different data structures like stack, queue etc.
- v. Representing lists like polynomials.

8. Polynomial

A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term. However, for any polynomial operation, such as addition or multiplication of polynomials is easier to deal.

8.1 Polynomial Representation using Array

A polynomial is a expression, derived in terms of sum of elements which can be represented as CX^e , where C is a *coefficient*, X is a *variable* and e is a *exponent*.

For example: $X^4+10X^3+3X^2+7X + 1$

In above *example* the polynomial is represented as a sum of 5 elements where

- 1st element: X^4 having coefficient 1 and exponent 4 for variable X
- 2nd element: $10X^3$ having coefficient 10 and exponent 3 for variable X
- 3rd element: $3X^2$ having coefficient 3 and exponent 2 for variable X
- 4th element: $7X$ having coefficient 7 and exponent 1 for variable X
- 5th element: 1 having coefficient 1 and exponent 0 for variable X

Oct.2011 – 2M

What are the different ways to represent polynomial?

From programming point of view, representation of polynomial is important in case of carrying different operations on it like:

- i. Addition
- ii. Subtraction
- iii. Multiplication
- iv. Division

The polynomial can be stored and represented using array.

The array can have the size depending upon highest degree/exponent of polynomial. So to represent polynomial in above *example*, the array can be declared with size 5 i.e. highest exponent + 1.

Hence array can be declared as

```
int poly[5];
```

Hence, elements in an array will be having indices 0 to 4 which can be used to represent exponents of polynomial and value stored at these indices will represent the coefficients of respective term having exponent equal to the index.

For *example*: $X^4+10X^3+3X^2+7X+1$

It can be represented in an array

```
int poly={1,7,3,10,1}
```

Diagrammatically it is shown as below

	0	1	2	3	4
Poly	1	7	3	10	1
	$1 * X^0$	$7 * X^1$	$3 * X^2$	$10 * X^3$	$1 * X^4$

Hence, each index stores coefficient of term having exponent exactly equal to an index i.e. polynomial can be stored in reverse order according to their exponents in to an array.

8.2 Addition of Polynomials Using Array

When two polynomials are to be added, the resultant polynomial can be generated by adding the coefficients from both polynomials having same exponent.

For example

$$A = 5X^3 + 3X^2 + 2X + 1$$

$$B = 4X^2 + X + 9$$

$$\text{Hence } C = 5X^3 + 7X^2 + 3X + 10$$

To add two polynomials using array, the following steps can be followed.

- i. Store both given polynomials into an array in reverse order of their exponents.
- ii. Declare array for resultant polynomial.
- iii. Consider the values at each and every index from both arrays
- iv. Add those values and store the result at corresponding index, at which the values are considered, in resultant array.
- v. Repeat above step for all the indices.
- vi. Display resultant array.

For example

$$A = 5X^3 + 3X^2 + 2X + 1$$

A	0	1	2	3
	1	2	3	5
	$1 * X^0$	$2 * X^1$	$3 * X^2$	$5 * X^3$

$$B = 4X^2 + X + 9$$

B	0	1	2	3
	9	1	4	0
	$1 * X^0$	$1 * X^1$	$4 * X^2$	$0 * X^3$

$$C = A + B = 5X^3 + 7X^2 + 3X + 10$$

C	0	1	2	3
	1+9=10	2+1=3	4+3=7	5+0=5
	$10 * X^0$	$3 * X^1$	$7 * X^2$	$5 * X^3$

9. Structure

Structure is another secondary data type. Structure is user defined data type, which is used to store dissimilar / heterogeneous data type under unique name. Keyword 'struct' is used to declare structure data type. In structure all elements are public by default and referred as 'member' which must be enclosed within { } and the name given to structure is called as 'structure tag'.

9.1 Syntax to define Structure

```
struct structure_name
{
    structure element 1;
    structure element 2;
    -----
    -----
    structure element n;
} structure_variable_list;
```

Example

```
struct student
{
    char name[30];
    char course[5];
    int age;
    int year;
};
```

This defines *student* as a new user defined data type. The variables of type *student* can be declared as follows.

```
struct student s1;
```

Note: declaring variables of type student is similar to declaring them as int or float. Variable can be declared at the end of definition of structure as follows:

```
struct student
{
    char name[30];
    char course[5];
    int age;
    int year;
}s1;
```

The variable name is s1, it has members called name, course, age and year.

9.2 Accessing Members of a Structure

Each member of a structure can be used as a normal variable and can be accessed using structure variable name, dot operator and structure member name respectively.

This is as shown below:

```
s1.name
```

Here the dot is an operator which selects a member 'name' from a structure variable name 's1'. Similarly, we can also declare pointer variables of type student as:

```
struct student *s2;
```

To access a member of structure using a pointer variable of structure, it has its own operator arrow (→) which can be used as follows.

```
s2 → name; or (*s2).name;
```

9.3 Structures as Function Arguments

A structure can be passed as a function argument just like any other variable using both pass by value and pass by reference methods.

When we want to modify the value of members of the structure to be passed as argument, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we want to change.

In order to accept structure as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the structure with pointer to it.

For example, the following function:

```
void fun(struct student *s1)
```

where, *fun* is a function name which accepts pointer *s1* as a parameter of type structure student

The *syntax* can be as follows,

```
return_type function_name(struct struct_name *variable_name)
```

Where, *return_type* is any valid data type (like int, float...), *function_name* is a valid identifier, *struct* is a keyword to mention structure is passed as a parameter, *struct_name* is a valid identifier, ** variable_name* specifies the pointer to structure.

When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

/ Program to calculate SI using structure and Pointer.*/*

```
#include<stdio.h>
struct interest
{
    int p,n,r;
};
void main()
{
    struct interest si;
    float calc(struct interest );
    float simpintr;
    printf( "Enter the values of P,N &R \n" ) ;
    scanf( "%d%d%d", &si.p, &si.n, &si.r);
    simpintr=calc(si);
    printf("Simple interest is %.2f For amount %d For years %d By the rate
%d", simpintr, si.p, si.n, si.r);
    getch();
}
float calc(struct interest si1)
{
    return (si1.p*si1.n*si1.r)/100;
}
```

Output

Enter the values of P, N and R

1000 1 13

Simple interest is 130.00 for amount 1000. For year 1 By the rate 13

9.4 Self-Referential Structures

Self-Referential structure is a structure having one or more of its member as a pointer to structure itself. Simply, self-referential structure is a structure of which members can refer the same structure.

For example

```
struct list
{
    char data;
    struct list *link;
} l1;
```

In above *example* structure list contains one member struct list *link i.e. pointer variable of same structure. Hence structure list can be called as self referential structure.

Difference between Array and Structure

	Array	Structure
1.	An array is a collection of related data elements of same type.	Structure can have elements of different types.
2.	An array is a derived data type.	A structure is a programmer-defined data type.
3.	Any array behaves like a built-in data types. All we have to do is to declare an array variable and use it.	But in the case of structure, first we have to design and declare a data structure before the variable of that type are declared and used.
4.	All elements of array have the same type i.e. homogeneous type.	All elements of structure may be heterogeneous.
5.	Elements of array are referred to by its position.	Elements of a structure are its unique name.
6.	<i>For example:</i> To access 3 rd elements of array 'Number' & store value in variable 'A' we can write. A=Number[2];	<i>For example:</i> To access value of structure variable 'book' having structure element name as 'pages' we can write. no=book.pages;
7.	<i>Syntax</i> data_type array_name[size];	<i>Syntax</i> struct structure_name { structure element 1; structure element 2; ----- ----- }structure_variable_list; structure element n;
8.	<i>For example:</i> int no[10];	<i>For example</i> struct book { char name[25]; float price; } b1,b2;

Solved Examples

1

Oct.12,14 - 4M

1. Write a 'C' Program for evaluation of polynomial.

Solution

A 'C' program for evaluation of polynomial is as,

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 10
void main()
{
    int a[MAXSIZE];
    int i, N,power;
```

```

float x, polySum;
clrscr();
printf("Enter the order of the polynomial\n");
scanf("%d", &N);
printf("Enter the value of x\n");
scanf("%f", &x);
/*Read the coefficients into an array*/
printf("Enter %d coefficients\n",N+1);
for (i=0;i <= N;i++)
{
    scanf("%d",&a[i]);
}
polySum = a[0];
for (i=1;i<= N;i++)
{
    polySum = polySum * x + a[i];
}
power = N;
/*power--;*/
printf("Given polynomial is:\n");
for(i=0;i<= N;i++)
{
    if(power < 0)
    {
        break;
    }
    /* printing proper polynomial function*/
    if (a[i] > 0)
        printf(" + ");
    else if (a[i] < 0)
        printf(" - ");
    else
        printf (" ");
    printf("%dx^%d  ",abs(a[i]),power--);
}
printf("\nSum of the polynomial = %6.2f\n",polySum);
}

```

2. Write a program to accept roll no, name and marks for N students from the user and print the data. (Use structure array).

1

Oct.2011 – 4M

Solution

The following program is used to accept the data of N students and print all data on the screen.

```
#include<stdio.h>
```

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

```
struct stud /*structure definition of representing student information */
```

```

int rollno;
char name[20];
int sub1_marks, sub2_marks, sub3_marks, sub4_marks, sub5_marks;
float avg;
};
void main()                /* main function */
{
    struct stud s[20];     /* arrays of structure declaration */
    int n, i;
    clrscr();
    printf("\nEnter the number of students");
    scanf("%d", &n);       /* accept the number of students */
    for(i=0; i<n; i++)     /* accept all data from n number of students */
    { printf("\nEnter Roll No");
      scanf("%d", &s[i].rollno);
      printf("\nEnter Name");
      scanf("%s", &s[i].name);
      printf("\nEnter Marks of 5 Subjects");
      scanf("%d%d%d%d%d", &s[i].sub1_marks, &s[i].sub2_marks, &s[i].sub3_marks,
        &s[i].sub4_marks, &s[i].sub5_marks);
      /* calculate the percentage marks of each student */
      s[i].avg=(s[i].sub1_marks+s[i].sub2_marks+s[i].sub3_marks+s[i].sub4_ma
        rks+s[i].sub5_marks)/5;
    }
    /* print the roll no, name, marks of all 5 subjects and percentage of all
    students */
    for(i=0; i<n; i++)
    { printf("\n\t\t\t\t\t***MARHKSHEET***\n");
      printf("\t\t\t\t\tNAME :- %s", s[i].name);
      printf("\t\t\t\t\tROLL NO :- %d", s[i].rollno);
      printf("\t\t\t\t\tsub1_marks=%d\n\t\t\t\t\tsub2_marks=%d\n\t\t\t\t\tsub3_marks=%d\n\t\t\t\t\tsub4_marks=%d\n\t\t\t\t\tsub5_marks=%d", s[i].sub1_marks, s[i].sub2_marks,
        s[i].sub3_marks, s[i].sub4_marks, s[i].sub5_marks);
      printf("\n\t\t\t\t\tPERCENTAGE :- %f", s[i].avg);
    }
    getch();
} /* end of main */

```

2

Apr.10, Oct.10 - 4M

3. Write a 'C' program for evaluation of a given polynomial.
(e.g. $2x^3 + x + 3$)

Solution

```

#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct
{
    float coef;
    int exp;
}poly;
void readpoly(poly p[], int n);
void dispoly(poly p[], int n);
void main()
{
    poly a[MAX], b [MAX];
    int m, n;
    printf("\n Enter the number of m);
    scanf("%d", &m);
    readpoly(a, m);
    printf ("\n Enter the number of n);
    scanf("%d", &n);
    readpoly(b,n);
    printf("\n The Result");
    dispoly(a, n);
}
void readpoly( poly p[], int n)
{
    int i;
    printf ("\n Enter the terms in descending order");
    for(i=0;i<n;i++)
    {
        printf("\n Ceof and power %d", i+1);
        scanf("%f %d", &p[i].coef, p[i].exp);
    }
}
void dispoly(poly p[], int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%d.2fx^%d+", p[i]. coef, p[i].exp);
}

```

4. Write a 'C' program for addition of two Polynomials.**Solution**

```

#include<stdio.h>
#include<conio.h>
struct term    // structure to define each term in polynomial
{

```

```
int exp;
int coeff;
};
struct polynomial /* structure of polynomial represent number of terms.*/
{
    struct term a[10];
    int n;
};
void main()
{
    struct polynomial p,p1,p3;
    int i;
    clrscr();
    printf("\nEnter no of terms of first polynomial: ");
    scanf("%d",&p.n);
    for(i=0;i<p.n;i++)
    {
        printf("\nEnter the %dth term in order exponent and
                coefficient: ",i+1);
        scanf("%d%d",&p.a[i].exp,&p.a[i].coeff);
    }
    printf("\nFirst Polynomial is: ");
    for(i=0;i<p.n;i++)
    {
        printf("(%dx)^%d +",p.a[i].coeff,p.a[i].exp);
    }
    printf("\n\nEnter the no of terms of second polynomial: ");
    scanf("%d",&p1.n);
    for(i=0;i<p1.n;i++)
    {
        printf("\nEnter the %dth term in order exponent and coefficient: ",i+1);
        scanf("%d%d",&p1.a[i].exp,&p1.a[i].coeff);
    }
    printf("\nSecond Polynomial is: ");
    for(i=0;i<p1.n;i++)
    {
        printf("(%dx)^%d +",p1.a[i].coeff,p1.a[i].exp);
    }
    printf("\n\nAddition is: \n");
    if(p.n>p1.n)
    p3.n=p.n;
```

```

else if (p.n < p1.n)
p3.n = p1.n;
else
p3.n = p1.n;
for (i = 0; i < p3.n; i++)
{
    if (p.a[i].exp == p1.a[i].exp)
    {
        p3.a[i].exp = p.a[i].exp;
        p3.a[i].coeff = p.a[i].coeff + p1.a[i].coeff;
    }
    if (p3.n == p1.n && p1.a[i].exp > p.a[i].exp)
    {
        p3.a[i].exp = p1.a[i].exp;
        p3.a[i].coeff = p1.a[i].coeff;
    }
    if (p3.n == p.n && p.a[i].exp > p1.a[i].exp)
    {
        p3.a[i].exp = p.a[i].exp;
        p3.a[i].coeff = p.a[i].coeff;
    }
}
for (i = 0; i < p3.n; i++)
{
    printf("(%dx)^%d +", p3.a[i].coeff, p3.a[i].exp);
}
getch();
}

```



PU Questions

2 Marks

1. What is use of (&) address operator and dereferencing (*) operator? [Oct.15, Apr.15 – 2M]
2. What is space complexity? How is it calculated? [Oct.2015 – 2M]
3. What is difference between structure and polynomial? [Oct.15, Apr.15 – 2M]
4. What is self referential structure? [Oct.2015 – 2M]
5. What is pointer? What are the operations can be performed on the pointer? [Oct.2015 – 2M]
6. What are the different types of data structures? [Apr.15, Oct.14 – 2M]
7. What is time complexity? How it is calculated? [Oct.2014 – 2M]
8. Give the formulae for address calculation for row and column major representation? [Oct.2014 – 2M]
9. How to calculate count of Best, Worst and Average case? [Oct.2014 – 2M]

[Oct.2012 – 2M]

[Oct.2012 – 2M]

[Apr.12, 10, Oct.10 – 2M]

[Apr.12,10 Oct.10,09–2M]

[Oct.2011 – 2M]

[Oct.2011 – 2M]

[Oct.2011 – 2M]

[Oct.2011 – 2M]

[Apr.2011 – 2M]

[Apr.2011 – 2M]

[Oct.10, Apr.10 – 2M]

[Oct.2009 – 2M]

[Oct.2009 – 2M]

[Oct.2015 – 4M]

[Oct.15, Apr.15 – 4M]

[Apr.2015 – 4M]

[Apr.15,12 – 4M]

[Apr.15, Oct.14 – 4M]

[Oct.2014 – 4M]

[Oct.2012 – 4M]

[Oct.2012 – 4M]

[Oct.2011 – 4M]

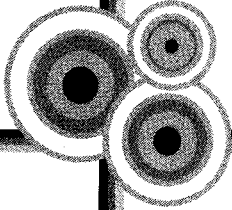
[Oct.10, Apr.10 – 4M]

10. What is Algorithm? State its properties.
11. What is ADT for an Array?
12. How to measure performance of an Algorithm?
13. What is Self-referential Structure?
14. Define time Complexity?
15. Define Array. Give its ADT.
16. What are the different ways to represent polynomial?
17. What is the difference between Malloc and Calloc?
18. Define Algorithm.
19. What is primitive data structure?
20. Explain in brief the functions of Dynamic Memory Allocation.
21. Define ADT (Abstract Data Type).
22. What is Array?

4 Marks

1. What is algorithm? Explain its characteristics in detail.
2. Explain different types of Asymptotic notations in detail.
3. What is an algorithm? How to measure its performance?
4. Write a "C" program for addition of two polynomials.
5. Explain different types of dynamic memory allocation functions.
6. Write a 'C' Program for evaluation of polynomial.
7. Explain Dynamic Memory Allocation Functions with their Syntax.
8. Write a 'C' Program for evaluation of polynomial.
9. Write a program to accept roll no, name and marks for N students from the user and print the data. (Use structure array).
10. Write a 'C' program for evaluation of a given polynomial. (e.g. $2x^3 + x + 3$).

SEARCHING AND SORTING TECHNIQUES



1. Introduction

Searching is a technique to locate a position of a particular item in a list. In real applications, the list is implemented as an array and the goal is to find a particular element. An element can be searched by matching the list with element.

There are two types of searching technique: *linear search* and *binary search*.

2. Linear Search

Linear Search is the simplest and most logical method used for searching. It can be applied for sequential storage structures like files, arrays or linked lists.

Linear Search is more efficient for the applications when data is unsorted.

2

Oct.2015- 4M
Explain linear search method with an example.

Apr.2012 - 2M
Explain Linear Search method.

It is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an unordered list.

When many values have to be searched in the same list, it often pays to pre-process the list in order to use a faster method.

As this method does not require additional memory so insertion and deletion can be easily done in method. This method will more efficient for the applications which are used on small to medium sized lists.

In this technique procedure used is to check the first item, then the second item, and so on until you find the required element or reach the end of the list. As the linear path is followed to check the element i.e. from first element to the last one in the list, this technique is called as linear (or sequential) search.

Main steps of Linear Search

Assume an array called list is having positive integers. The task is to search for the location of the number that is the value of a variable target. Hence to search for the location of target in list, linear search will work like:

1. location = 0;
2. while ((location is less than size of list)
if(element at location is not a target))
increment the value of location
else
location = position of target in original list
return location as the result
end while

```
/* Linear Search Program */  
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int list[5]={10,20,30,40,50},i,target;  
    clrscr();  
    printf("Enter element to search:-");  
    scanf("%d",&target);  
    for(i=0;i<5;i++)  
    {
```

```

if(list[i]==target)
{
    printf("Element found at position:%d",i+1);
    break;
}
}
if(i==5)
printf("Element not found");
getch();
}

```

A loop is rotated to search the target in the array. The target can be anywhere in the array or perhaps not in it at all. The program must be able to exit the search loop as soon the target is located, otherwise to search all the items in the array, if necessary.

The initial value of the control variable i is 0 and increments by 1 each time through the loop, since valid index values range from 0 to the number of items minus one. The loop repeats as long as i is less than the number of items. The expression `list[i] == target` tests whether the value of the *target* variable is the same as that of the current *list* element. This condition allows the loop to continue if the current element of the list is *not* the same as the value of *target*. At loop exit, either *target* has been found, (`list[i] == target`), or the entire array has been checked without finding it, and `i == size of array`. In the former case, *location* is displayed as the value of i , while in the latter *the message 'element not found'* will be displayed.

The following table shows an *example* of the operation of the linear search algorithm. The first row of the table is the array indices and second row is the data stored at the indexed location and remaining rows indicates the index values used for location (L), at each iteration of the algorithm.

If the target value is 52, its location is found on the 3rd iteration.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	68	34	52	32	27	47	46	41	49	52	78	23	71	74	77	55
1 st iteration	L															
2 nd iteration		L														
3 rd iteration			L													

Efficiency of Linear Search

The efficiency of algorithm depends upon the number of main steps require to finish. But, the exact number of steps depends on the input data. For the linear search algorithm, the number of steps

depends on whether the target is present in the list, if it is present, where is the location of it in the list, as well as on the length of the list.

1

Apr.2015– 2M

How to calculate count of Best, Worst and Average case?

For search algorithms, the main steps are the comparisons of list values with the target value. The number of comparisons required to locate target in the list, represents the *best case*, the *worst case*, and the *average case* as shown in the following table. For each case, the number of steps is expressed in terms of n , which represents the number of items in the list.

Case	Number of Comparisons (for $n = 100$)	Comparisons as a function of n
Best Case (fewest comparisons)	1 (target is first item)	$O(1)$
Worst Case (most comparisons)	100 (target is last item)	$O(n)$
Average Case (average number of comparisons)	50 (target is middle item)	$O(n/2)$

The best case analysis doesn't tell us much. If the first element checked happens to be the target, *any* algorithm will take only one comparison. The worst and average case analyses give a better indication of algorithm efficiency.

Notice that if the list grows in size, the number of comparisons required finding a target item in both worst and average cases grows *linearly*. In general, for a list of length n , the worst case is n comparisons. Hence the technique is called *linear search* because its complexity/efficiency can be expressed as a linear function.

3. Binary Search

This technique is more efficient as the list is sorted, *for example*, in ascending order. A sorted list can be used to narrow the search as explained below:

The technique of binary search is to check the middle (approximately) item in the list. If it is not the target and the target is smaller than the middle item, the target must be in the first half of the list. If the target is larger than the middle item, the target must be in the last half of the list. Thus, one unsuccessful comparison reduces the number of items to be checked by half.

Once, the required half is identified, the search continues by checking the middle item in the particular half of the list. If it's not the target, the search narrows to the half of the remaining part of

the list. This splitting process continues until the target is located or the remaining list consists of only one item. If that item is not the target, then it's not in the list.

The main steps of the binary search algorithm can be written as:

- i. Location = -1;
- ii. While ((more than one item in list) and (haven't yet found target))
 - a. Look at the middle item
 - b. If (middle item is target)
have found target
else
 - c. If (target < middle item)
list = first half of list
 - d. else (target > middle item)
list = last half of list
end while
- iii. If (have found target)
location = position of target in original list
- iv. Return location as the result

Iterative Implementation of Binary Search

```
/* Binary Search Program iterative */  
  
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int list[5]={10,20,30,40,50},i,target;  
    int first,middle,last;  
    first=0;last=4;  
    clrscr();  
    printf("Enter element to search:");  
    scanf("%d",&e);  
    while(first<=last)  
    {  
        middle=(first+last)/2;  
        if(target==list[middle])
```

```
{
    printf("Element found at position:%d",middle+1);
    break;
}
if(target<list[middle])
    last=middle-1;
if(target>list[middle])
    first=middle+1;
}
if(first>last)
    printf("Element not found");
getch();
}
```

Two integer variables, *first* and *last*, record the first and last index values for the part of the array remaining to be searched. The integer variable, *middle*, stores the middle position between *first* and *last*. Each time through the loop, the target is compared to the middle item. If the target is less than the middle item, the next iteration searches the lower half of the current part of the array by setting *last* to the position just before *middle*. Thus, the next part to search is from positions *first* to *middle - 1*. If target is greater than the middle item, the next iteration searches the upper half of the current part of the array by setting *first* to the position just after *middle*. Thus the next part to search is from *middle + 1* to *last*.

The search ends when the target item is found or the values of *first* and *last* cross over, so that $last < first$, indicating that there are no array items left to check.

Recursive Implementation of Binary Search

This technique can also be implemented using recursive function as follows:

```
/* Binary Search Program recursive */
#include<stdio.h>
#include<conio.h>
void main()
{
    int arr[5]={10,20,30,40,50},i,target;
    int first,middle,last;
    int binsrch(int [],int,int,int);
    first=0;last=4;
```

```
clrscr();
printf("Enter element to search:-");
scanf("%d",&e);
i=binsrch(arr,first,last,target)
if(i!=-1)
    printf("Element not found");
else
    printf("Element found at position:-%d",i+1);
getch();
}
int binsrch(int list[],int low,int high,int x)
{
    int mid;
    if(low<=high)
    {
        mid=(low+high)/2;
        if(x==list[mid])
            return mid;
        if(x<list[mid])
            binsrch(list,low,mid-1,x);
        if(x>list[mid])
            binsrch(list,mid+1,high,x);
    }
return -1;
}
```

In this technique, a recursive function *binsrch* is declared with four arguments as

int list[], an array in which the target is to be located,

int low as lower bound and

int high as upper bound of a part in which the target is to be located and

int x, a target element to be searched.

Initially, function *binsrch* will be called with main array i.e. *list[5]* where lower bound is 0 and upper bound is 4 (i.e. size - 1). The function will calculate the middle position of the array and will compare element with it.

If the target is less than the middle item, the function will be called again to search the lower half of the current part of the array by setting *high* to the position just before middle. Thus, the arguments

for low and high to the function will be low and middle -1 respectively where remaining arguments i.e. list and x will be same. If target is greater than the middle item, the function will be called again to search the lower upper half of the current part of the array by setting low to the position just after middle. Thus, the arguments for low and high to the function will be middle +1 and high respectively where remaining arguments i.e. list and x will be again same.

The following table shows an *example* of the operation of the binary search algorithm. The first row of the table is the array indices and second row is the data stored at the indexed location and remaining rows indicates the index values used for first (F), last (L) and middle (M), at each iteration of the algorithm. If the target value is 52, its location is found on the 3rd iteration.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	23	27	29	32	34	41	46	47	49	52	55	68	71	74	77	78
1 st iteration	F							M								L
2 nd iteration									F			M				L
3 rd iteration									F	M	L					

Efficiency of Binary Search

To evaluate efficiency of binary search, we have to count the number of comparisons in the best case and worst case. The average case, which is a bit more difficult, is omitted.

The best case occurs if the middle item happens to be the target. Then only one comparison is required to find it.

The worst case will occur if the target is not in the array and the process of dividing the list in half part continues until there is only one item left to check. Here is the pattern of the number of comparisons for an initial array having length as an even power of 2 (64).

Items Left to Search	Comparisons So Far
64	0
32	1
16	2
8	3
4	4
2	5
1	6

For a list size of 64, there are 6 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half. It can be represented as $6 = \log_2 64$

In general, if n is the size of the list to be searched and C is the number of comparisons to do so in the worst case, $C = \log_2 n$. Thus, the efficiency of binary search can be expressed as a logarithmic function, in which the number of comparisons required to find a target increases logarithmically with the size of the list.

The following table summarizes the analysis for binary search.

Model	Number of Comparisons (for $n = 100000$)	Comparisons as a function of n
Best Case (fewest comparisons)	1 (target is middle item)	1
Worst Case (most comparisons)	16 (target not in array)	$\log_2 n$

4. Sorting

Sorting is used to arrange the data in a meaningful order. The order can be implemented according to the type of elements. If elements are of alphabetic type the order will be ascending (from 'A' - 'Z') or descending ('Z' - 'A'). Similarly if elements are of numeric type, order will be ascending (from lowest value to highest value) or descending (highest value to lowest value).

So, accordingly sorting order can be ascending or descending, immaterial of type of elements.

For example, it is relatively easy to look up the phone number of a friend because the names in the phone directory are sorted in alphabetical order. This example clearly indicates that, sorting greatly improves the efficiency of searching. Because, if we want to find the names in any logical order, it will take a long time to look up that phone number.

For simplicity we have considered the integer elements to be sorted in ascending or descending order. Consider the following array.

Sorting can be defined as arrangement of elements in a specific order.

1

Apr.2011 – 2M
Define Sorting.

1

Oct.2015– 2M
What is difference
between sorting and
searching?

Unsorted array	Ascending Order	Descending Order															
<table border="1"><tr><td>12</td><td>37</td><td>8</td><td>45</td><td>21</td></tr></table>	12	37	8	45	21	<table border="1"><tr><td>8</td><td>12</td><td>21</td><td>37</td><td>45</td></tr></table>	8	12	21	37	45	<table border="1"><tr><td>45</td><td>37</td><td>21</td><td>12</td><td>8</td></tr></table>	45	37	21	12	8
12	37	8	45	21													
8	12	21	37	45													
45	37	21	12	8													
0 1 2 3 4	0 1 2 3 4	0 1 2 3 4															

Hence, from the above *example*, it is observed that original array will be rearranged to place elements at their appropriate position depending upon sorting order.

Hence, in sorting method elements will be reshuffled and placed at appropriate position according to their value as compared with others. Sorting can be performed in many ways. Several methods can be used to sort information using different algorithms. Some examples of these algorithms are the Selection Sort, the Insertion Sort, the Bubble Sort, the Quick Sort, and the Radix Sort. There is no sort algorithm that is clearly better than all others in all circumstances. While selecting the particular algorithm, the factors like complexity of the algorithm, the size of the data structure (an array) has to be considered.

We focus on the Bubble Sort, the Selection Sort, and the Quick Sort. The bubble and selection sort algorithms are very simple. Quick Sort is more complex, but it is highly efficient.

4.1 Sorting Techniques

3**Apr.12,10 – 2M**

What is sorting? State various techniques of Sorting.

Oct.2010 – 2M

State different types of Sorting Techniques

The array can be sorted by applying various techniques as mentioned below:

- i. Bubble sort
- ii. Insertion sort
- iii. Selection sort
- iv. Quick Sort

The techniques are different in the way they are rearranging the elements. Each and every technique will be compared with particular element from rest of the elements or all the remaining elements to decide the appropriate position of them. We will discuss the techniques in detail.

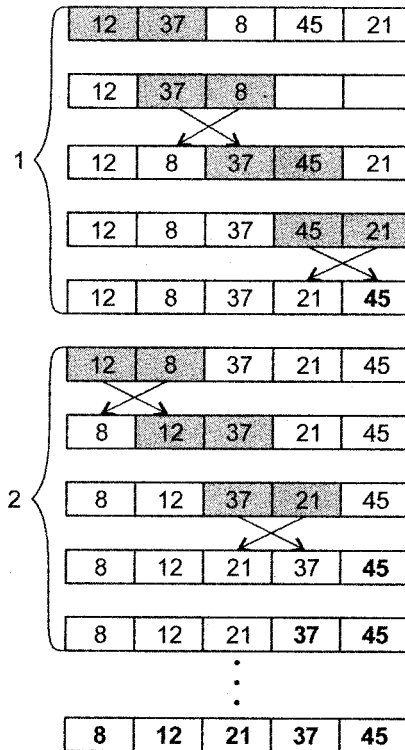
5. Bubble Sort

In this technique, the element will be compared with its immediate next element i.e. the elements are compared in adjacent pairs like (0, 1), (1, 2), (2, 3), (3, 4) etc. depending upon number of elements in an array. So, pair having two elements is to be considered as previous element and next element. While comparing these elements in pair, the elements will be interchanged if previous element is greater than next element.

When all the pairs will be compared for first time the largest element will be placed at the end of an array.

Consider the *example*,

Unsorted Array



As the array is having 5 elements there will $(5 - 1)$ i.e. 4 pairs should be compared. Hence, the pairs can be calculated by $\text{size} - 1$. Similarly, when all the pairs will be compared and elements are interchanged whenever required, we can find largest element at the end of an array. Hence comparing pairs for only single time is not sufficient. Then how many times we should compare it?

Here 45 is the largest element among all the 5 elements and will be placed at the last. But also 37 is second largest among remaining 4 elements. Hence, each and every element is having its own significance according to its value as compared with others and place these elements at their appropriate places we have to compare the pairs equal to the number of elements in an array.

Hence, to implement bubble sort technique for an array having n elements, we have to compare $(n - 1)$ pairs for n times.

Though in given example, the array is sorted in 2nd step itself, we have to consider next comparisons also as number of comparisons required will be totally depending upon the initial array in which some elements could be at their appropriate position initially or in few steps.

Algorithm

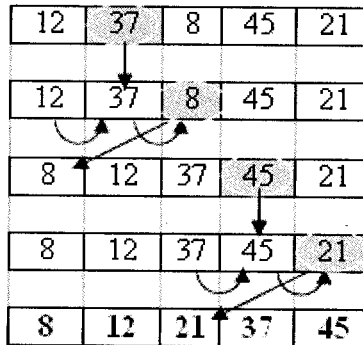
- i. Consider the first adjacent pair of elements.
- ii. If previous element in pair is greater than next element, exchange the values of them.
- iii. Consider next pair and repeat step no 2 for all the remaining pairs.
- iv. Repeat step no. 1 equals to the number of elements in an array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[5]={12,5,32,18,10},i,j,p,t;
    clrscr();
    for(i=0;i<5;i++)
    { for(j=0;j<4;j++)
        { if(a[j]>a[j+1])
            { t=a[j];
              a[j]=a[j+1];
              a[j+1]=t;
            }
        }
    }
    printf("Sorted array:-\n");
    for(i=0;i<5;i++)
    printf("%d\n",a[i]);
    getch();
}
```

6. Insertion Sort

In this technique, the element at a particular position will be compared with all the previous elements i.e. 1st element is compared with 0th, 2nd will be compared with 1st and 0th and so on depending upon number of elements in an array. The element which is to be compared with its previous element is considered as a Pivot element. While comparing these previous elements, if any of them is greater than the pivot element, that element will be moved to its next position and finally pivot element will be inserted at its appropriate position. Consider the *example*

Unsorted array



2
Apr.12, Oct.11 – 4M
Explain Insertion Sort with an example.

As we are going to compare previous element, we should start with the element at 1st position as pivot element and compare it with the element at 0th position. Similarly 2nd will be compared with 1st and 0th and so on depending upon number of elements in an array. While comparing previous elements, if any of the pivot element is smaller that element will be placed in its next position and at the same time position of pivot will be decreased. Though position of pivot element is decreased, it is not immediately placed at that position till all the previous elements will be compared. Once, all the previous elements will be compared, the pivot will be *inserted* at its proper position.

To apply Insertion sort technique to an array having n elements, the pivot elements will be at positions 1,2,3...n-1 and will be compared with the elements previous to them i.e. from i-1 to 0, where i is the position of pivot element.

1
Apr.2015– 2M
Compare the efficiency of Bubble Sort with Insertion Sort?

Again, the number of comparisons required will be totally dependent upon the initial array in which some elements could be at their appropriate position initially.

Algorithm

- i. Consider the second element as pivot element.
- ii. Compare it with previous elements.
- iii. If previous element is greater than pivot element place it at next position.
Decrement the position of pivot element.
- iv. Repeat step no. 3 for all the previous elements.
- v. Insert pivot element at its decremented position.
- vi. Consider next element as pivot element.
- vii. Repeat step no.2 up to the last element in an array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[5]={12,5,32,18,10},i,j,p,t;
    clrscr();
    for(i=1;i<5;i++)
    {
        p=a[i];
        t=i;
        for(j=i-1;j>=0;j--)
        {
            if(a[j]>p)
            {
                a[j+1]=a[j];
                t--;
            }
        }
        a[t]=p;
    }
    printf("Sorted array:-\n");
    for(i=0;i<5;i++)
    printf("%d\n",a[i]);
    getch();
}
```

7. Selection Sort

This technique selects the largest element and puts it in the position of higher index. Then it finds the next largest and places it to second last position and this technique will be followed until the array is sorted. To put an element at its appropriate place, it will be swapped with the element already present at that position. As a result, the array will be sorted from the end position of the array up to first position.

Consider the *example*;

12	37	8	45	21
0	1	2	3	4
12	37	8	21	45
0	1	2	3	4
12	21	8	37	45
0	1	2	3	4
12	8	21	37	45
0	1	2	3	4
8	12	21	37	45

In this method, the element at the highest index i.e. 4 is considered and it will be compared with all the remaining elements i.e. 0 to 3 and actual highest element will be found. The highest element and the element at highest index (4) will be swapped. Hence, in this method the highest element will be placed at highest index. Then element at previous highest index i.e. 3 is considered and again actual highest element from remaining elements will be placed at this position.

So the highest elements will be placed from highest index to lowest index.

Algorithm

- i. Consider the element at highest index as max element.
- ii. Compare it with all the remaining previous elements.
- iii. If any element is greater than max element.

Consider it as max element and store its position as 'pos'.

1

Apr.2012 – 4M
Explain Selection Sort technique with an example.

- iv. Swap the elements at the position of highest index and element at pos.
- v. Consider previous index of highest index as next highest index.
- vi. Repeat step no. 1 up to the first index.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[5]={12,5,32,18,10},i,j,p,t;
    clrscr();
    for(i=0;i<5;i++)
    {
        p=a[i];
        for(j=i+1;j<5;j++)
        {
            if(a[j]<p)
            {
                t=p;
                p=a[j];
                a[j]=t;
            }
        }
        a[i]=p;
    }
    printf("Sorted array:-\n");
    for(i=0;i<5;i++)
    printf("%d\n",a[i]);
    getch();
}
```

8. Quick Sort

3

Apr.2015- 2M

Explain Quick sort technique with an example.

Oct.12, 09 - 4M

Explain Quick sort with an example.

Quick sort is a very efficient sorting technique which sorts the array in two phases:

- i. Partition phase and
- ii. Sort phase.

In partition phase, we divide the array of items into two partitions and then recursively sort the two partitions in sort phase, *i.e.* we *divide* the problem into two smaller parts. The sort phase simply sorts

the two smaller p that are generated in the partition phase. This makes Quick Sort a good example of the divide and conquers strategy for solving problems.

As we will see, most of the work is done in the partition phase - it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase.

In this technique the *partition* phase must arrange all the items in the lower part and less in the upper part. To do this, we select a *pivot* element and arrange all the items which are less than the pivot in the lower part and all those greater than it in the upper part. In the final step, the pivot is dropped and then lower part as well as upper parts is divided individually by using same technique.

Quick sort: *Example*

lb									ub
6	1	4	3	5	2	9	7	8	
P	dn								up
6	1	4	3	5	2	9	7	8	
P					up	dn			
2	1	4	3	5	6	9	7	8	
				P					

Repeat Steps for Partition 1 and Partition 2

Partition 1				
2	1	4	3	5
P	dn			Up

Partition 2		
9	7	8
P	dn	up

Partition Phase

- In this *example*, to make partition, the element at lowest index is considered as pivot element. The position of element next to it is considered as down (dn) and the position of last element is considered as up(up). Firstly, the pivot element is compared with element at down(dn) position. If pivot element is greater than element at down(dn) will be incremented and same step will be repeated for the element at down(dn) position.
- If the element at down(dn) is greater than pivot, the pivot element will be compared with element at up position. Now condition is exactly reversed as if pivot element is less than element at up position, the position of up will be decremented and again step will be repeated for the element at up position.

Due to above steps, both down(dn) and up(up) will cross each other. Then pivot element and element at up will be swapped. But in some cases they will not cross each other as some element can be greater than pivot and some element at up is less than pivot. In this situation the element at down(dn) and up(up) should be swapped and whole process is repeated again. Hence, pivot element is placed at such position where left part elements are less than the pivot and elements at right will be greater than pivot.

3. Now by skipping pivot element, we will get two separate partitions.

Sort Phase

In the sort phase, both these partitions are sorted separately using above partition phase. Hence using both the phases array is divided into discrete no. of partitions, where each partition is having only single element which will be placed at its appropriate position and thus whole array will be sorted by recursively applying partition phase and sort phase.

1

Oct.2011 – 4M

Write an algorithm for Quick sort (use recursion).

The recursive algorithm consists of four steps

- i. If there are one or less elements in the array to be sorted, halve and exit.
- ii. Pick left-most element in the array as a 'pivot' element.
- iii. Split the array into two parts where left part is having elements larger than the pivot and the right with elements smaller than the pivot.
- iv. Recursively repeat the algorithm for both halves of the original array.

Example

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[5]={12,5,32,18,10},lb,ub,i;
    void quicksort(int [],int,int);
    clrscr();
    lb=0;ub=4;
    quicksort(a,lb,ub);
    printf("Sorted array:-\n");
    for(i=0;i<5;i++)
        printf("%d\n",a[i]);
    getch();
}
```

```
void quicksort(int a[],int lb,int ub)
{
    int p,dn,up,t;
    if (lb<ub)
    {
        dn=lb;up=ub;
        p=a[dn];
        while(dn<up)
        {
            while(p>=a[dn])
                dn++;
            while(p<a[up])
                up--;
            if (dn<up)
            {
                t=a[dn];
                a[dn]=a[up];
                a[up]=t;
            }
        }
        t=a[lb];
        a[lb]=a[up];
        a[up]=t;
        quicksort(a,lb,up-1);
        quicksort(a,up+1,ub)
    }
}
```

9. Heap Sort

1
Oct.2014 – 2M
Explain Heap sort
technique with an
example.

The Heap Sort method of sorting uses the data structure-binary tree. This procedure is based on a special type of binary tree called a heap.

Definition of heap

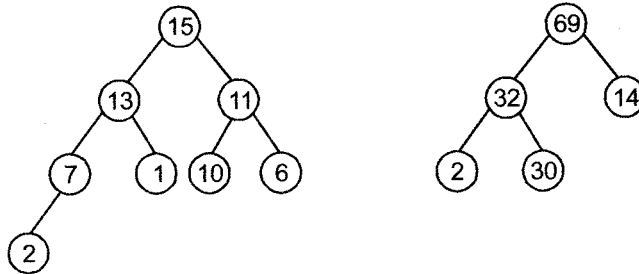
A heap of size 'n' is a binary tree of 'n' nodes such that,

- i. all the leaves of the tree are on adjacent levels

- ii. all the levels except the lowest level are full
- iii. the key in the parent is greater than or equal to the keys in the children.

This implies that the largest element is in the root node.

Example



Algorithm

Due to the heap property that all levels except the last are filled, we can organize the data in a sequential manner.

The root is at position 0 followed by its children.

Example

The first tree shown above can be represented as

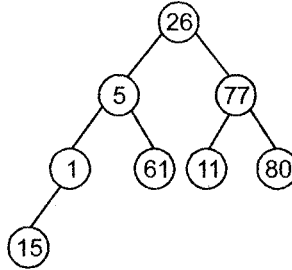
15	13	11	7	1	10	6	2
----	----	----	---	---	----	---	---

1. $top = 0, last = n-1$
2. Build a heap out of data $[top]$ to $data[last]$
3. Interchange $data [top]$ and $data[last]$
4. $last = last - 1$
5. Repeat from 2 as long as $last > 0$.

Let us first see how to create a heap. If the keys are

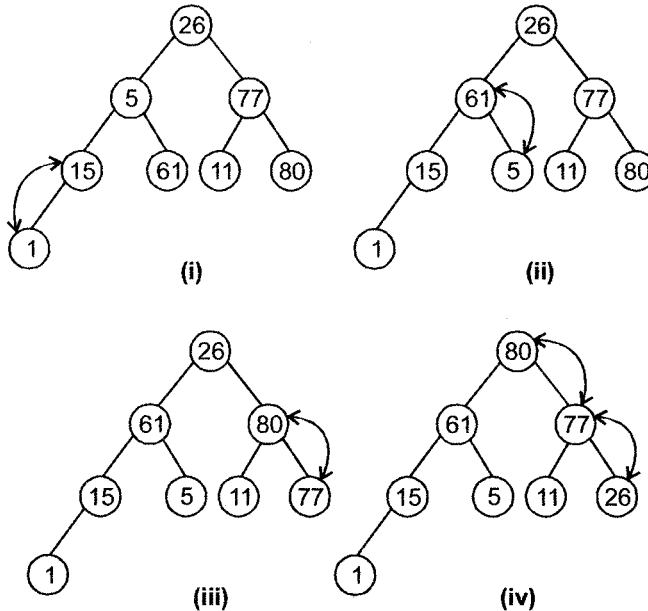
26 5 77 1 61 11 80 15

We first create a binary tree by successively adding elements to the left and right subtrees.



Binary Tree of unsorted elements

This tree is not a heap. Hence, we have to convert it into a heap by the following transformations.



Build Heap

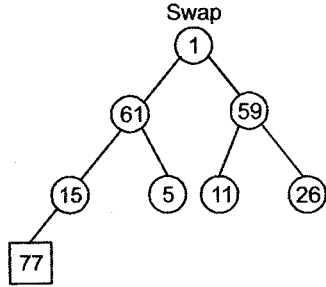
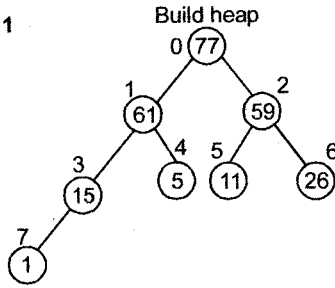
If the keys in the children is greater than the parent node, the key in the parent and the greater child are interchanged.

We shall now consider sample data to be sorted using the heap sort procedure.

Data: 26 5 7 1 61 11 59 15

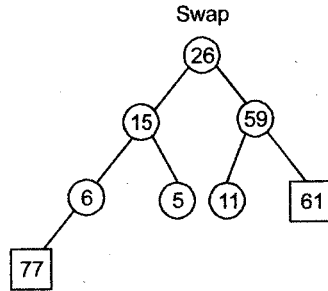
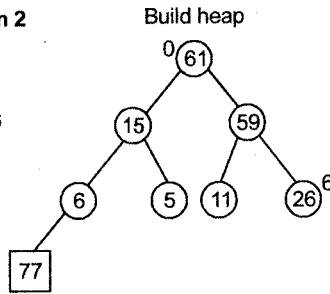
Iteration 1

Top = 0
Last = 7



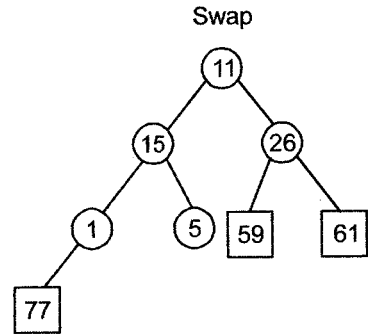
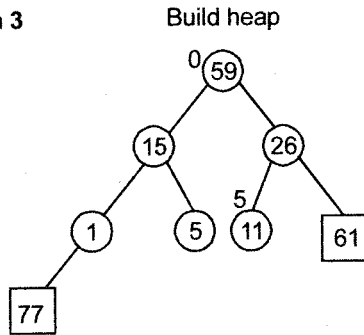
Iteration 2

Top = 0
Last = 6



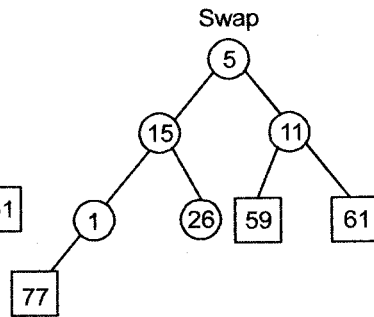
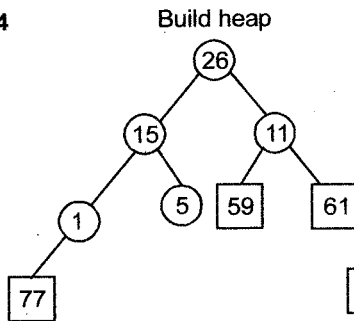
Iteration 3

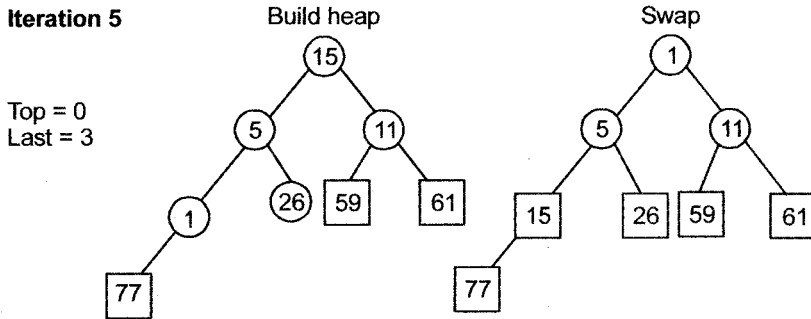
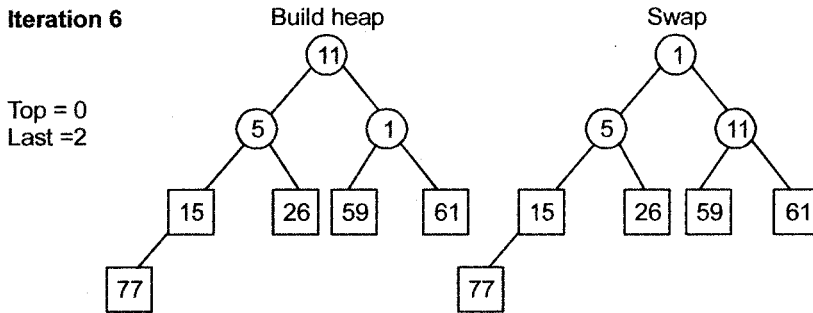
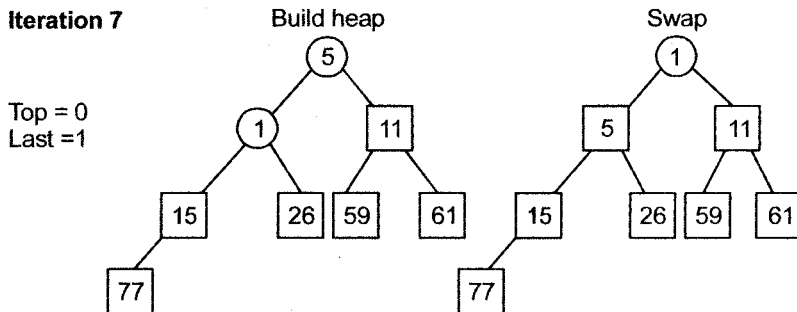
Top = 0
Last = 5



Iteration 4

Top = 0
Last = 4



Iteration 5**Iteration 6****Iteration 7****Heap Sort**

At this stage the keys in the tree are in the sorted order.

Algorithm for Heapsort

1. Start
2. Accept n elements in array data.

3. Convert data into a heap
for $i = n/2$ to 0
Heapify (i,n)
4. last = n - 1, top = 0.
5. Interchange data [top] and data [last]
6. last = last - 1
7. Heapify (top, last), i.e. recreate heap
8. If last > 0
goto 5
9. Stop

The procedures Heapify will create a heap with root i having children at position $2i + 1$ and $2i + 2$.

The steps will be as follows:

Algorithm for Heapify

1. Start
2. key = data [i]
3. $j = 2*i + 1$
4. If data [j] < data [j+1]
 $j = j + 1$
5. If key > data [j]
goto 8
6. Interchange data [i] and data [j]
7. Heapify (j,n)
8. Stop

Efficiency of heapsort

If the tree has k levels with 2^{i-1} nodes on level i , then the initial creation of the heap will take time = the sum of nodes on each level * the number of levels the node can move.

$$\text{i.e. } \sum_{i \leq i \leq k} 2^{i-1} (k-i) \approx O(n)$$

The next loop calls Heapify $n-1$ times.

Each call to Heapify takes $O(\log_2 n)$ time.

Hence the total computing time $O(n \log_2 n)$.

In the average case, quicksort is better but in the worst case, heapsort is much better than quicksort.

10. Merge Sort

Merging is the process of combining two or more sorted data lists into a third list such that it is also sorted.

For example, if the two sorted arrays are:

Array 1: 3 5 10 23 56

Array 2: 4 6 9 60

The merged array will be

Array 3: 3 4 5 6 9 10 23 56 60

Merge Sort is based on the above process of merging. In this process, a list is divided into two sub-lists which are sorted individually and then merged. To sort each sublist, it is further divided into two sub-lists and the process continues till sub-lists of size 1 are obtained. Since a list of size 1 is sorted, we can merge adjacent disjoint pairs of sub-lists. The merging process continues till only one list of size n is obtained.

Strategy

Merge sort follows the **Divide and Conquer** strategy

1. **Divide:** Divide an n element sequence into $n/2$ subsequences.
2. **Conquer:** Sort the two sequences recursively.
3. **Combine:** Merge the two sorted sequences into a single sequence.

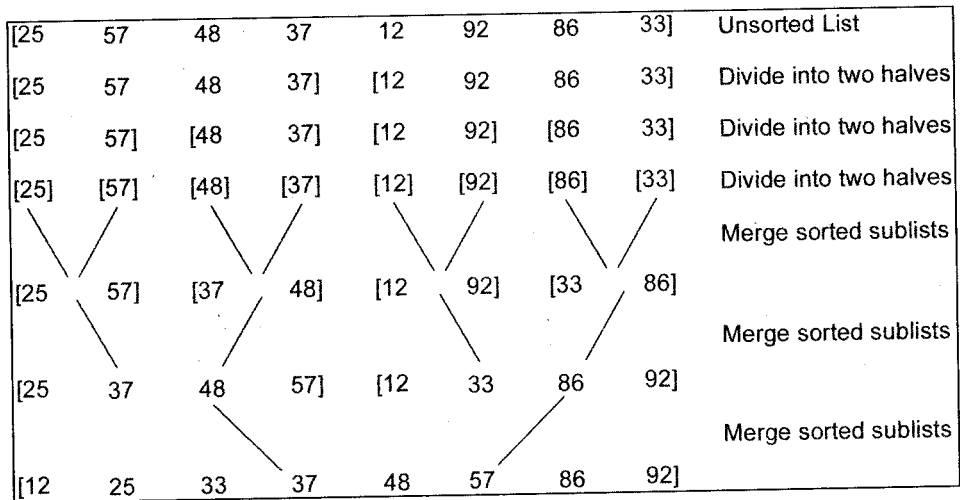
Algorithm for Merge Sort

1. Start
2. A is an array of 'n' elements.
3. low = 0 , high = n-1
4. if (low < high) i.e. if the array can be partitioned
 - mid = (low + high)/2 //mid is the middle position
 - MergeSort(A,low, mid) // sort the first half
 - MergeSort(A,mid+1, high) // sort the second half
 - Merge(A,low,mid,high); //Merge the sorted halves
5. Stop.

Example,

Let us consider an example of sorting 8 elements

a[8] = (25,57,48,37,12,92,86,33)



The following program gives the recursive procedure to sort n elements in the ascending order.



Merge Sort

```
void Merge (int a[], int low, int mid, int high)
{
  /*Merge a[low]..a[mid] and a[mid+1]..a[high] into a sorted list
  Store the sorted list in a[low]..a[high]
```

```
b is a temporary array for merging */
int i,j,k, b[20];
i = low; j = mid+1; k = 0;
while((i <= mid) && (j <= high))
{
    if(a[i] < a[j])
        b[k++] = a[i++];
    else
        b[k++] = a[j++];
}
while(i <= mid)
    b[k++] = a[i++];
while(j <= high)
    b[k++] = a[j++];
/* Copy merged elements from b to a */
for(j = low, k = 0; j<=high ; j++, k++) //
    a[j] = b[k];
}

void MergeSort(int a[], int low, int high)
{
    int mid;
    if(low < high) //more than one element
    {
        mid =(low + high) /2;        // Divide a into two sublists
        MergeSort(a, low, mid);      // Sort first sub list
        MergeSort(a, mid+1, high);   // Sort second sub list
        Merge(a, low, mid, high);    //Merge Sorted sub lists
    }
}

void main()

int a[20], i, n;
```

```

printf("How many numbers :");
scanf("%d", &n);
printf("\nEnter the unsorted numbers :");
for(i=0;i<n;i++)
scanf("%d", &a[i]);
MergeSort(a, 0, n-1);
printf("\nThe sorted list is ");
for(i=0;i<n;i++)
    printf("%d\t", a[i]);
}

```



Time Complexity of Merge Sort

The best case and worst case time complexity of Merge sort is $O(n \log_2 n)$.

In each step, the array is divided into two equal sublists. This takes $O(\log_2 n)$ time. In each step, a total of n elements are merged in the sublists. Thus, the total time taken is $O(n \log_2 n)$. This can be proved as follows:

$T(n)$ = Time taken to sort 2 sublists of size $n/2$ + time taken to merge. This can be written as:

$$T(n) = \begin{cases} a & \text{if } n=1, a \text{ is a constant} \\ 2T(n/2) + cn & \text{if } n > 1, c \text{ is a constant} \end{cases}$$

When n is a power of 2 i.e. $n = 2^k$, this equation can be solved as follows:

$$\begin{aligned}
 T(n) &= 2T(n/2) + cn \\
 &= 2(2T(n/4) + cn/2) + cn \\
 &= 4T(n/4) + 2cn \\
 &= 4(2T(n/8) + cn/4) + 2cn \\
 &= 8T(n/8) + 3cn \\
 &\quad \vdots \\
 &= 2^k T(n/2^k) + kcn \\
 &= an + cn \log_2 n \\
 &\approx O(n \log_2 n)
 \end{aligned}$$

Advantage of Merge Sort

1. It is a very efficient method since its best and worst case time complexity is $O(n \log_2 n)$.

Limitations of Merge Sort

1. Additional memory is required for the merging process. It is not in-place.
2. Stack space is needed due to recursion.

11. Comparison of Sorting Methods

We have studied various sorting methods. If we have to choose one method to sort n elements, we should compare the methods and then make the selection. The choice of a method should be done on the basis of various criteria like the number of data elements, time complexity and space complexity.

The following chart summarizes the sorting methods studied so far.

Method	Time Complexity	Space Complexity	Remarks
Bubble	Best : $O(n^2)$ Worst: $O(n^2)$	Additional space only for temporary variables is required. (In place)	Original Bubble Sort method has a time complexity of $O(n^2)$. Only the modified version has a best case complexity of $O(n)$. Stable method
Quick	Best: $O(n \log_2 n)$ Worst : $O(n^2)$	Space depends upon number of nested recursive calls or size of the stack. (In place)	Uses Divide and Conquer strategy. There is a large performance difference in the average and worst case. Non recursive implementation is complicated. Not stable.
Insertion	Best : $O(n)$ Worst : $O(n^2)$	Additional memory required only for temporary variables. (In place)	Better than Bubble sort. Performance largely depends upon the ordering of data. Better performance when data elements are almost sorted. Stable method
Merge	Best : $O(n \log_2 n)$ Worst : $O(n \log_2 n)$	Additional storage for auxiliary array is required. (not In place)	Uses divide and conquer, non recursive, implementation is complicated. Stable method.

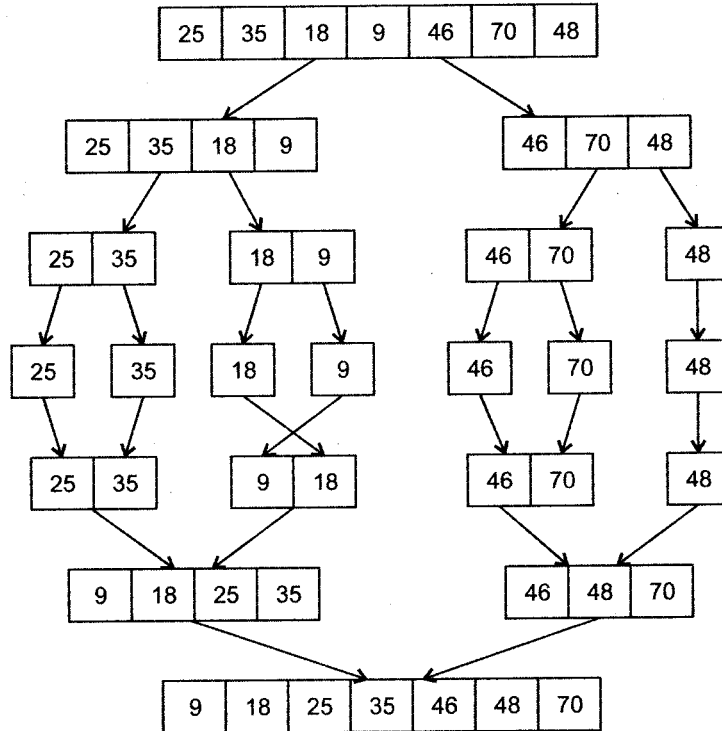
Solved Examples

1. Sort the following data using Merge Sort.
Show each step in detail: 25, 35, 18, 9, 46, 70, 48

Solution

Consider an array A with n indices ranging from A_0 to A_{n-1} . We apply merge sort to $A(A_0 \dots A_{c-1})$ and $A(A_c \dots A_{n-1})$ where c is the integer part of $n/2$. When the two halves are returned they will have been sorted. They can now be merged together to form a sorted array.

Steps for given data to apply the merge sort is,



1

Apr.2012 – 4M

2. Sort the following data using Bubble Sort show each step in detail: 108, 97, 71, 23, 12, 57, 93, 100

Solution

The basic idea behind Bubble sort is to pass the list sequentially several times. In each pass compare successive pairs of elements ($x[i]$ with $x[i+1]$) and interchange the two elements if they are not in required order. One element is placed in its correct position in each pass. In subsequent passes, we consider one element less than the previous pass. A total of $n-1$ passes are required to sort 'n' keys. In first pass, the largest element will sink to the bottom, second largest element in the second pass and so on.

Start comparing from first pair of elements from the given data as follows:

The given data is

108, 97, 71, 23, 12, 57, 93, 100

Pass 1:

108	97	97	97	97	97	97	97
97	108	71	71	71	71	71	71
71	71	108	23	23	23	23	23
23	23	23	108	12	12	12	12
12	12	12	12	108	57	57	57
57	57	57	57	57	108	93	93
93	93	93	93	93	93	108	100
100	100	100	100	100	100	100	108

Pass 2:

97	71	71	71	71	71	71
71	97	23	23	23	23	23
23	23	97	12	12	12	12
12	12	12	97	57	57	57
57	57	57	57	97	93	93
93	93	93	93	93	97	97
100	100	100	100	100	100	100
108	108	108	108	108	108	108

Pass 3:

71	23	23	23	23	23
23	71	12	12	12	12
12	12	71	57	57	57
57	57	57	71	71	71
93	93	93	93	93	93
97	97	97	97	97	97
100	100	100	100	100	100
108	108	108	108	108	108

Pass 4:

23	12	12	12
12	23	23	23
57	57	57	57
71	71	71	71
93	93	93	93
97	97	97	97
100	100	100	100
108	108	108	108

Pass 5:

12	12	12	12
23	23	23	23
57	57	57	57
71	71	71	71
93	93	93	93
97	97	97	97
100	100	100	100
108	108	108	108

Pass 6:

12	12	12
23	23	23
57	57	57
71	71	71
93	93	93
97	97	97
100	100	100
108	108	108

Pass 7:

12	12
23	23
57	57
71	71
93	93
97	97
100	100
108	108

1

Oct.2011 – 4M

3. Write a program to accept N numbers from the user and sort using merge sort.

Solution

This merge sort method follows divide and conquer algorithm.

```
#include<stdio.h>
#include<conio.h>
#define MAX_ARY 10
void merge_sort(int x[], int end, int start);
void main() /* main function */
{
    int ary[MAX_ARY];
    int j = 0;
    printf("\n\nEnter the elements to be sorted: \n");
    for(j=0;j<MAX_ARY;j++)
        scanf("%d", &ary[j]);
    printf("Array before Mergesort :");
```

```

for(j = 0; j < MAX_ARY; j++)
printf(" %d", ary[j]);
printf("\n");
merge_sort(ary, 0, MAX_ARY - 1);
printf("After Merge Sort :");
for(j = 0; j < MAX_ARY; j++)
printf(" %d", ary[j]);
printf("\n");
getch();
} /* end of main function */
/* Method to implement Merge Sort*/
void merge_sort(int x[], int end, int start)
{ int j = 0;
  const int size = start - end + 1;
  int mid = 0;
  int mrg1 = 0;
  int mrg2 = 0;
  int executing[MAX_ARY];
  if(end == start)
  return;
  mid = (end + start) / 2;
  /* recursive call to the merge sort function */
  merge_sort(x, end, mid);
  merge_sort(x, mid + 1, start);
  for(j = 0; j < size; j++)
  executing[j] = x[end + j];
  mrg1 = 0;
  mrg2 = mid - end + 1;
  for(j = 0; j < size; j++)
  { if(mrg2 <= start - end)
    if(mrg1 <= mid - end)
    if(executing[mrg1] > executing[mrg2])
    x[j + end] = executing[mrg2++];
    else
    x[j + end] = executing[mrg1++];
    else
    x[j + end] = executing[mrg2++];
    else
    x[j + end] = executing[mrg1++];
  }
}
}/* end of merge sort function*/

```

4. Sort the following numbers in an ascending order using Heap Sort Method: 23, 15, 29, 11, 01, 07

Solution

Step 1: Create a Heap

Heap contents	New element	Interchange element
Empty	23	
23	15	
25,15	29	23,29
29,15,23	11	
29,15,23,11	01	
29,15,23,11,01	07	
29,15,23,11,01,07		

Step 2: Sorting

Heap	Interchanged element	Delete element	Sorted array	Action
29,15,23,11,01,07	29,07			interchange 29 and 7 in order to delete 29 from heap
7, 15,23,11,1		29		Delete 29 from heap and add into sorted array
23,15,7,11,1	7,23		29	interchange 7 and 23 as they are not in order in the heap
1,15,7,11,23	23,1		29	interchange 23 and 1 in order to delete 23 from heap
1,15,7,11		23		Delete 23 from heap and add into sorted array
15,1,7,11	1,15		23,29	interchange 1 and 15 as they are not in order in the heap
15,11,7,1	1,11		23,29	interchange 1 and 11 as they are not in order in the heap
1,11,7,15	15,1		23,29	interchange 15 and 1 in order to delete 15 from heap
1,11,7		15		Delete 15 from heap and add into sorted array
11,1,7	1,11		15,23,29	interchange 1 and 11 as they are not in order in the heap
7,1,11	11,7		15,23,29	interchange 11 and 7 in order to delete 11 from heap
7,1		11		Delete 11 from heap and add into sorted array
1,7	7,1		11, 15,23,29	interchange 7 and 1 in order to delete 7 from heap
1		7	11, 15,23,29	Delete 7 from heap and add into sorted array
Empty		1	7, 11, 15,23,29	Delete 1 from heap and add into sorted array
Empty			1,7, 11, 15,23,29	

The sorted numbers are: 1, 7, 11, 15, 23, 29.

2

Apr.11, Oct.10 – 4M

5. Sort the following data, using insertion sort (show each step) in descending order: **-5, 8, 12, 64, 5, 88**

Solution

Sorting using insertion sort in descending order

Elements/values: **-5, 8, 12, 64, 5, 88**

Step 1: Compare 2nd element with 1st, as $8 > -5$, swap it.

$\therefore 8 -5 12 64 5 88$

Step 2: Compare 3rd element, i.e., 12 with 1st and 2nd elements as $12 > 8$, shift it as first element and move other elements by 1 position.

$\therefore 12 8 -5 64 5 88$

Step 3: Compare 4th element, i.e., 64, as $(64 > 12)$ shift it as first element and move others to the next.

\therefore 64 12 8 -5 5 88

Step 4: Compare 5th element, i.e., 5 with all previous elements. $(5 > -5)$ and shift it at appropriate position.

\therefore 64 12 8 5 -5

Step 5: Compare 6th element, i.e., 88, as $(88 > 64)$, add it at first position and move the other elements to the next.

\therefore 88 64 12 8 5 -5

\therefore Sorted elements in descending order is 88 64 12 8 5 -5

6. Sort the following data, using selection sort in descending order (show each step: 3, 66, -15, -99, 6, 27).

Solution

Element/values: 3 66 -15 -99 6 27

Step 1: Compare 1st element with all elements and swap if necessary.

a. 66 3 -15 -99 6 27

Here, 66 is compared with all next elements. And it is highest element so on need to swap.

66 3 -15 -99 6 27

Step 2: Compare IInd element with all other next elements and swap if necessary.

66 3 -15 -99 6 27

a. 66 6 -15 -99 3 27 $(6 > 3)$

b. 66 27 -15 -99 3 6 $(27 > 6)$

Step 3: Compare IIIrd element with all elements and swap if necessary.

66 27 -15 -99 3 6

a. 66 27 3 -99 -15 6 $(3 > -15)$

b. 66 27 6 -99 -15 3 $(6 > 3)$

Step 4: Compare IVth element with all elements and swap if necessary.

66 27 6 -99 -15 3

a. 66 27 6 -15 -99 3 $(-15 > -19)$

b. 66 27 6 3 -99 -15 $(3 > -15)$

Step 5: Compare Vth element with all elements and swap if necessary.

66 27 6 3 -99 -15

a. 66 27 6 3 -15 -99 $(-15 > -99)$

Thus, in 5 iterations, elements are sorted using selection sort in descending order.

66 27 3 -15 -99

1

Apr.2010 – 4M

7. Sort the following elements using merge sort. Show each step in detail: 5, 8, 89, 30, 42, 92, 64, 4, 21, 56.

Solution

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num[10] = {5,8,89,42,92,64,4,21,56};
int ans[10];
ans[0] = num[0];
int i,j,k,l,temp;
clrscr();
j=0,k=0;
printf("\n Original Numbers ");
for(i=0;i<10;i++)
{
printf("%d",num[i]);
}
for(i=1;i<=10;i++)
{
printf("\n Comparing Num %d", i+1);
if(num[i] <= ans[i])
{
ans[i] = num[i];
j++;
}
for(l=1;l<=i;l++)
{
printf("%d\t",ans[l]);
}
if(j==10||k==10)
break;
}
printf("\n Number After using Merge Sort
Method");
for(i=1;i<=10;i++)
{
printf("%d", ans[i]);
}
getch();
}
```

1

Apr.2010 – 4M

8. Sort the following elements by using quick sort: 48, 29, 8, 59, 72, 88

Solution

```
#include<stdio.h>
#include<conio.h>
int partition(inta[],int low, int high)
{
int i,j,temp,key;
key = a[low];
i = low+1;
j = high;
while(1)
```

```

    { while(i<high && key >= a[i])
      i++;
      while(key < a[j])
        j--;
      if(i<j)
      {
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
      }
      else
      {
        temp = a[low];
        a[low] = a[j];
        a[j] = temp;
      }
    }
  }
}
void quicksort(int a[], int low, int high)
{
  int j;
  if(low < high)
  {
    j = partition(a, low, high);
    quicksort(a, low, j-1);
    quicksort(a, j+1, high);
  }
}
void main()
{
  int i, n, a[6] = {48, 29, 8, 59, 72, 88};
  printf("\n Enter the value of N");
  scanf("%d", &n);
  quicksort(a, 0, n-1);
  printf("\n The Sorted Array is\n");
  for(i=0; i<n; i++)
    printf("\n%d", a[i]);
}

```

9. Sort the following elements by using selection sort method:
10, 22, 65, 223, 87, 343, 98, 244

Solution

```

#include<stdio.h>
#include<conio.h>
void selection_sort(omt a[],int n)
{ int i, j, pos, small temp;
  for(i=0;i<n;i++)
  {
    small = a[i];
    pos = i;
    for(j=i+1;j<n;j++)
    {
      if(a[j]<small)
      {
        small = a[j];
        pos = j;
      }
    }
    temp = a[i];
    a[i] = a[pos];
    a[pos] = temp;
  }
}

```

```

    } }
    temp = a[pos];
    a[pos] = a[i];
    a[i] = temp;
} }
void main()
{
    int i, n, a[8] = {10, 22, 65, 223, 87, 343, 98, 244};
    clrscr();
    printf("\n Enter the Number of Elements to Sort\n");
    scanf("%d", &n);
    selection_sort(a, n);
    printf("\n The Sort Elements are \n");
    for(i=0; i<n; i++)
        printf("%d", a[i]);
    getch();
}

```

1

Oct. 2014 – 4M

10. Sort following data by using insertion sort techniques:
56, 98, 23, 67, 3, 87, 45, 77, 99

Solution

Index-	0	1	2	3	4	5	6	7	8
	56	98 23 67			3	87 45 77 99			
	↑				↑				
	A list of sorted elements				A list of unsorted elements				

1st iteration: (place element a [1] at its correct place)

0	1	2	3	4	5	6	7	8
56	98	23 67 3			87 45 77 99			
sorted		unsorted						

2nd iteration: (place a[2] at its correct place)

0	1	2	3	4	5	6	7	8
23	56	98	67 3			87 45 77 99		
sorted			unsorted					

3rd iteration: (place a[3] at its correct place)

0	1	2	3	4	5	6	7	8			
23	56	67	98	3				87	45	77	99
sorted				unsorted							

4th iteration: (place a[4] at its correct place)

0	1	2	3	4	5	6	7	8
3	23	56	67	98	87 45 77 99			
sorted					unsorted			

5th iteration: (place a [5] at its correct place)

0	1	2	3	4	5	6	7	8
3	23	56	67	87	98	45	77	99
sorted						unsorted		

6th iteration: (place a[6] at its correct place)

0	1	2	3	4	5	6	7	8
3	23	45	56	67	87	98	77	99
sorted							unsorted	

7th iteration: (place a[7] at its correct place)

0	1	2	3	4	5	6	7	8
3	23	45	56	67	77	87	98	99
sorted								unsorted

8th iteration: (place a [8] at its correct place)

0	1	2	3	4	5	6	7	8
3	23	45	56	67	77	87	98	99
sorted								

11. Compare the efficiency of Bubble sort with Selection Sort?

Solution

Efficiency of bubble sort with selection sort

- i. **Simplicity:** Both algorithms are equally simple to write.
- ii. **Time complexity:** Both are not data sensitive. Both of them having a timing requirement of $O(n^2)$.
- iii. **Sort stability:** Both sorting algorithms are stable.
- iv. **Storage requirement:** No additional storage is required.

1
Oct.2014 – 2M



PU Questions

2 Marks

- | | |
|---|------------------------|
| 1. What is difference between sorting and searching? | <u>[Oct.2015 – 2M]</u> |
| 2. How to calculate count of Best, Worst and Average case? | <u>[Apr.2015 – 2M]</u> |
| 3. Compare the efficiency of Bubble Sort with Insertion Sort? | <u>[Apr.2015 – 2M]</u> |
| 4. Compare the efficiency of Bubble sort with Selection Sort? | <u>[Oct.2014 – 2M]</u> |
| 5. "When Linear Search Method will be more efficient". Comment. | <u>[Oct.2012 – 2M]</u> |
| 6. Explain Linear Search method. | <u>[Apr.2012 – 2M]</u> |

[Apr.2012 – 2M]

[Apr.12,10 – 2M]

[Apr.2011 – 2M]

[Oct.2015 – 4M]

[Oct.2015 – 4M]

[Apr.2015 – 4M]

[Apr.2015 – 4M]

[Oct.2014 – 4M]

[Oct.2014 – 4M]

[Oct.2012 – 4M]

[Oct.12,09 – 4M]

[Apr.2012 – 4M]

[Apr.2012 – 4M]

[Apr.12,Oct.11 – 4M]

[Oct.2011 – 4M]

[Oct.2011 – 4M]

[Oct.2011 – 4M]

[Apr.2011 – 4M]

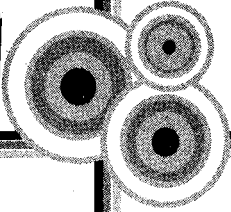
[Apr.11,Oct.10 – 4M]

7. What is a binary search tree?
8. What is Sorting? State various techniques of Sorting.
9. Define Sorting.

4 Marks

1. Sort the following data by using selection sort techniques:
87, 45, 12, 90, 67, 54, 34, 23, 88, 65.
2. Explain linear search method with an example.
3. Sort following data by using Insertion sort techniques:
12,5,122,9,7,54,4,23,88,60.
4. Explain Quick sort technique with an example.
5. Explain Heap sort technique with an example.
6. Sort following data by using insertion sort techniques:
56,98,23,67,3,87,45,77,99
7. Sort the following data using Merge Sort.
Show each step in detail: 25, 35, 18, 9, 46, 70, 48
8. Explain Quick sort with an example.
9. Sort the following data using Bubble Sort show each step in detail: 108, 97, 71, 23, 12, 57, 93, 100
10. Explain Selection Sort technique with an example.
11. Explain Insertion Sort with an example.
12. Sort the following numbers in an ascending order using Heap Sort Method: 23, 15, 29, 11, 01, 07
13. Write a program to accept N numbers from the user and sort using merge sort.
14. Write an algorithm for Quick sort (use recursion).
15. Sort the following elements using heap sort: 24, 6, 75, 12, 60, 15
16. Sort the following data, using insertion sort (show each step) in descending order: -5, 8, 12, 64, 5, 88

Chapter 3 LINKED LIST



1. Introduction

We use the concept of 'list' very frequently in our day-to-day lives. We make a list of tasks to be done in the day, a lady makes a list of shopping items, and students make a list of the topics to be studied and so on. However, once the list is made, it hardly remains the same. As the day progresses, there are new tasks to be added, completed tasks to be removed, tasks to be reordered and some tasks to be cancelled. Thus, there are constant modifications to the list i.e. the list is '*dynamic*' in nature.

The term 'list' refers to a linear collection of data items such that there is a first element, second and . . . a last element. Data processing frequently involves storing and processing data organized into lists.

The following shows a list of some beautiful colors.

Violet
Blue
Green
Orange
Red

If we wanted to store this list in memory, we could use the sequential representation i.e. store the colors in an array. Arrays use sequential mapping i.e. the data elements are stored in memory, fixed distances apart. This makes it easy to compute the location of any element in the array.

If the elements of the list are going to be fixed, then using an array will be a good method of storing the elements of the list. But if we wished to insert the colors 'Indigo' and 'Yellow' to complete the colors of the rainbow, it will mean that we would have to move some colors to make place for the missing colors. The same would apply if we had to remove certain colors from the list. Moreover, the use of arrays will impose an upper limit on the maximum number of colors in the list.

Thus, in general, the use of sequential representation for a list, which is dynamic in nature, proves to be inadequate due to the following reasons.

Limitations of Sequential Representation (Array)

1. An array is a static data structure i.e. the size of the array remains fixed.

Thus, even if the data structure actually uses less amount of storage to store elements or possibly uses no storage at all, the unutilized memory cannot be used for other purposes.

Moreover, if we require more space than allotted, it cannot be increased during run time.

2. Most real-time applications process variable size data. The amount of data to be manipulated and hence the storage requirements cannot be predicted in advance during design time.
3. Often, we need to insert, delete, move and reorder data. For this, a lot of elements will have to be moved, which will require a lot of processing time. If these operations are to be carried out very frequently, the processing time will be enormous.

Hence, to improve the efficiency, we have to find another method of representing the elements of the list so that the operations can be performed in a reasonable amount of time.

Apr.2015 – 4M

What are the drawbacks of sequential storage?

1

2. Concept of Linked Organization

One solution to the above problem is that instead of using sequential representation, a '**linked representation**' or '**linked organization**' should be used, i.e. unlike an array, where elements are stored sequentially in memory, items in a list may be located anywhere in memory. To access elements in the correct order, we store the address or location of the next element, with each element of the list.

Definition: Linked Organization

A Linked Organization is one in which the elements of the list are logically next to each other, but physically, they may not be adjacent.

Definition: Linked List

*A linked list is an ordered collection of data elements where the order is given by means of links i.e. each item is connected or '**linked**' to another item.*

Basically a linked list consists of '**nodes**'. Each node contains an item field and a link. The item field may contain a data item or a link.

Advantages of Linked List

The advantages of a Linked List over an Array are as:

- i. Linked List is an example of Dynamic Data Structure. They can grow and shrink during execution of the program.
- ii. Representation of linear data structure (polynomial, stack, and queue) can be easily represented using Linked List.
- iii. Linked list represents efficient memory utilization. Memory is not pre-allocated like array in linked list. Memory is deallocated when it is no longer needed.
- iv. Operations on linked list as Insertion and deletion are easier and efficient. It can be carried out in constant time.
- v. Linked lists do not need contiguous blocks of memory; extremely large data gets stored in an array might not be able to fit in memory.
- vi. Linked list storage does not need to be preallocated (again, due to arrays needing contiguous memory blocks).

2

Oct.2015 – 2M

What are the advantages of array over linked list?

Apr.2012 – 2M

What are the advantages of a Linked List over an Array?

3. Implementation of Linked List

In the previous section we have defined a linked list to be a linked collection of nodes, each containing some information. In this section, we will think about how to implement the linked list. The main issues to be considered are:

- a. How to store the elements of the linked list in memory?
- b. How to indicate their logical order?
- c. How to perform various operations on the list like insertion, deletion etc?

The way in which we will perform operations on the list will depend upon the method used to store the nodes of the linked list.

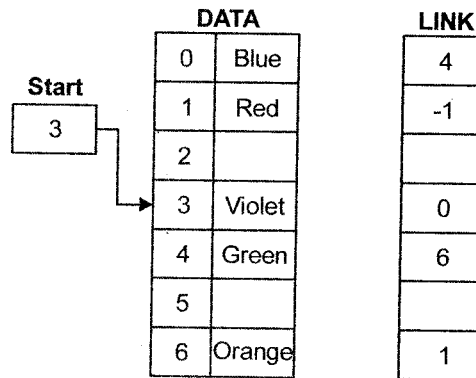
We can use two methods to implement the linked list:

1. Static Representation.
2. Dynamic Representation.

3.1 Static Representation

An array is used to store the elements of the list. The elements may not be stored in a sequential order. The elements can be stored at any position in the array. The logical order can be stored in another array called 'Link'. The values in this array tell the logical order of the elements in the array DATA. The corresponding 'link' of a data item tells where the next element is.

For example, let us consider our list of colors. The first color in the list is 'Violet' followed by 'Blue'. These colors can be stored anywhere in the array but the position in the Link field will tell where the next color in the sequence can be found. All we need to know is the starting position of the list. In this case, it is 3.



Start = 3 DATA [3] = Violet
 LINK[3] = 0 DATA [0] = Blue
 LINK[0] = 4 DATA [4] = Green
 LINK[4] = 6 DATA [6] = Orange
 LINK[6] = 1 DATA [1] = Red
 LINK[1] = -1 ⇒ List ended

Advantages

1. The implementation is simple.
2. For performing operations like insert, delete etc, all we have to do is update the links.

Disadvantage

1. Since we are using an array to implement the linked list, there will be the limitation of a static data structure; namely fixed memory size.

A linked list is a dynamic data structure i.e. the size of the list grows and shrinks depending upon the operations performed on it. This means that when we insert elements in the list, its size should increase and when elements are deleted, its size should decrease. This cannot be possible using an array which uses static memory allocation i.e. memory is allocated during compile time. Hence, we have to use '*dynamic memory allocation*' where memory can be allocated and de-allocated during run time.

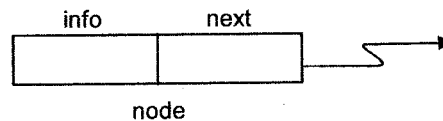
3.2 Dynamic Representation

The Static representation uses arrays, which is a static data structure and has its own limitations.

Another way of storing a list in memory is by dynamically allocating memory for each node one by one and linking them. Since we will dynamically allocate memory using functions like malloc and calloc, we will have to use pointers. Moreover, each node will be at random memory locations, so we have to store the address of the next node along-with the data in the node. All we have to do is remember the address of the first node in an external pointer.

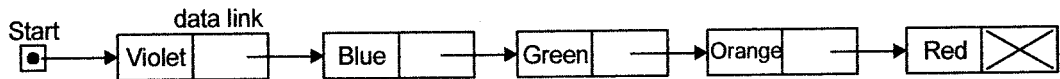
The node structure will thus contain two fields:

- a. *data or info* : which stores the information
- b. *link or next* : which stores the address of the next node.



Each node contains two parts, data and link. The data part may contain a single data item or a composite one like an entire record or it may be a link. The link or next part contains the address of the next node in the list. The last node contains a special value called 'NULL' which indicates end of the list.

The linked list for our colors can now be pictorially shown as below:



Advantages

1. Since memory is dynamically allocated during run-time, memory is efficiently utilized.
2. There is no limitation on the number of nodes in the list; except for the available memory.
3. Insertion, deletion and traversal can be easily done.
4. Memory can be freed when nodes have to be deleted.

Internal and External Pointers

In the example above, each node has a pointer called 'link' to the next node. This pointer is stored within the node. Hence it is called an **internal** pointer.

The pointer called 'start' stores the address of the first node of the list. This pointer is not contained within a node. Hence it is called an **external** pointer.

4. Types of Linked Lists

3

Oct.2015 – 2M
What is linked list structure

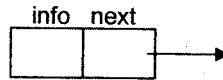
Apr.2012 – 2M
What is Linked List? State its types.

Apr.2011 – 4M
Explain types of Link Lists.

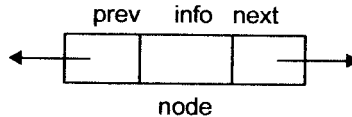
We can classify linked lists on the basis of:

1. Number of internal pointers in the node – singly or doubly linked list
2. Kind of collection – linear or circular linked list.

1. **Singly Linked List:** Each node in this list contains only one pointer which points to the next node of the list.

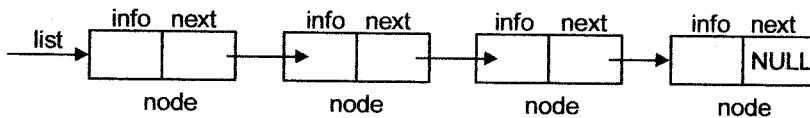


2. **Doubly Linked List:** Each node in this list contains two pointers; one pointing to the previous node and the other pointing to the next node. This list is used when traversing in both directions is required

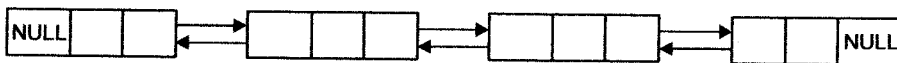


The Singly linked list and Doubly linked list can further be of two types depending on the kind of collection the list stores. The lists can be either organized in a linear manner or circular manner.

i. **Linear Linked List:** In this list, the elements are organized in a linear fashion and the list terminates at some point i.e. the last node contains a NULL pointer.

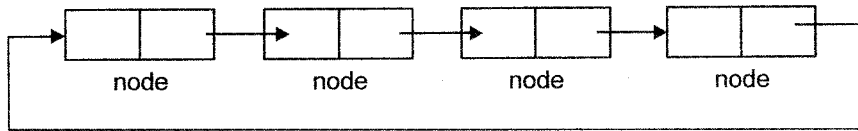


Linear Singly Linked List

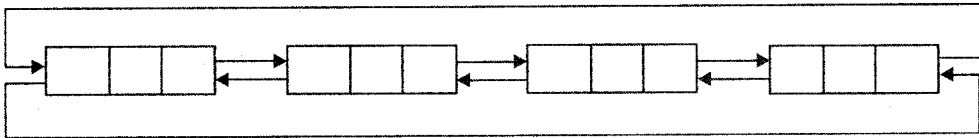


Linear Doubly Linked List

- ii. **Circular List** : In this list, the last node does not contain a NULL pointer at the end to signify the end of the list, but the last node points back to the first node i.e. it contains the address of the first node.

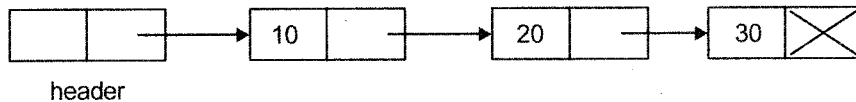


Circular Singly Linked List



Circular Doubly Linked List

Sometimes, an extra node is placed at the beginning of the list. Such a node is called 'Header Node'. This node does not store any data element but can be used to store some control information like number of elements etc.



5. Operations on a Singly Linked List

The various operations that can be performed on a singly linked list are given below.

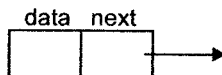
1. **Traversing the list:** Visiting each node of the list is called traversal.
2. **Computation of Length:** Count the total number of nodes in the list.
3. **Insertion:** A node can be inserted at the beginning, end or in between two nodes of the list.
4. **Deletion:** Deletion from a list may be done either position-wise or element-wise.
5. **Searching:** This process searches for a specific element in the list.

6. **Reversing or Inversion:** This process reverses the order of nodes in the list.
7. **Concatenation:** This process appends the nodes of the second list at the end of the first list i.e. it joins two lists.

We shall see how to perform these operations on a Singly linked list in the following sections. But first we have to study how to create the list. Only after that we will be able to perform the above operations.

5.1 Creation of a Singly Linked List

Each node of a linked list contains info / data part and a link/next part which is a pointer. The pointer stores the address of the next node.



We can implement a node using a **self referential structure** (structure which contains a pointer to itself).

This structure will have two fields which can be written as

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

We can also use the keyword `typedef` to create a user defined type called `NODE`. This can be done as follows.

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

Oct.2015 – 2M
What is use of "typedef" keyword?

This declaration just creates the structure template. No memory is allocated at this point. Memory will be allocated when variables are created.

To create nodes during run-time, we will have to use functions like **malloc** and **calloc** and nodes can be de-allocated using function **free**.

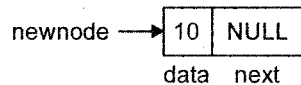
These operations require pointers. Hence we will have to use a pointer to `NODE`.

Example to create a single node:

```

NODE * newnode;
newnode = (NODE *)malloc(sizeof(NODE));
newnode → data = 10;
newnode → next = NULL;

```



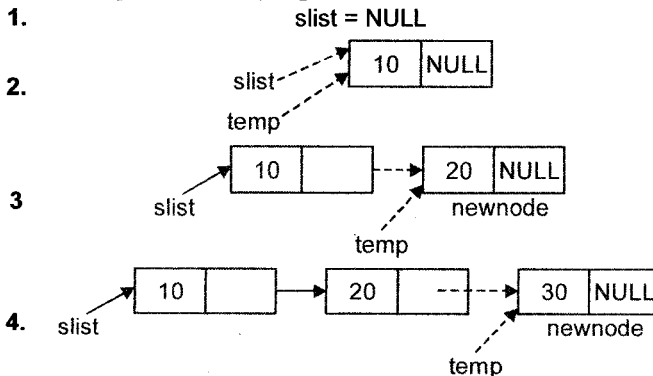
In order to make the code more readable, we can use the following statement.

```
# define NODEALLOC(NODE *)malloc(sizeof(NODE))
```

To create a linked list, we will create the nodes one by one and add them at the end of the list.

The list will be pointed to by an external pointer, which stores the address of the first node of the list. In the example below, we are creating three nodes with values 10, 20 and 30 and making the linked list.

The list is pointed to by a pointer called 'slist', which is initially NULL.



The algorithm to create a linked list containing 'n' nodes can be written as follows:

Algorithm

1. Start
2. Initialize the pointer to NULL i.e. `slist = NULL`
3. Accept number of nodes to be created in `n`.
4. Counter = 1
5. Create a new node using `malloc` and store its address in `newnode`

6. If slist is NULL then (i.e. the list is empty)
store the address of the new node in slist i.e. slist = temp= newnode
else
 attach newnode to temp i.e. temp->next=newnode
Move temp to the last node
7. Increment counter
8. If counter <= n
 goto 5
9. Stop

This function creates a list and returns a pointer to the first node of the list

Function: Create a List

```
#define NODEALLOC(NODE *)malloc(sizeof(NODE))
NODE * createlist(NODE * slist)
{
    int n, count;
    NODE * temp, newnode;
    slist = NULL; /* initialize pointer to NULL */
    printf("How many nodes;");
    scanf("%d", &n);
    for (count = 1; count <= n; count++)
    {
        newnode = NODEALLOC;
        newnode → next = NULL;
        printf("\n Enter the node data:");
        scanf("%d", &newnode → data);
        if(slist==NULL)
            slist = temp = newnode;
        else
        {
            temp → next = newnode;
            temp = newnode;
        }
    }
    return slist;
}
```

5.2 Traversing a List

1

Oct.2009 – 4MWhat do you mean by
Traversing a Linked List.

In order to display the elements of a list we will have to move from the first node to the last using the links till we reach NULL. This is called displaying the list. The address of the starting node has to be known.

Algorithm

1. Start
2. slist is the pointer to the first node of the list.
3. if slist == NULL
 Display "List is Empty"
 Go to step 8.
4. temp is a temporary pointer for traversal.
5. Make temp point to the first node i.e. temp=slist.
6. If temp ≠ NULL
 Display the data of temp i.e. display temp->data
 Move temp to the next node i.e. temp=temp->next.
7. Repeat from 6 as long as temp ≠ NULL.
8. Stop

Function: Display a List

```
void disp_list(NODE * slist)
{
    NODE * temp = slist;    /* Store address of first node in temp */
    if ( temp == NULL)
    {
        printf("List is empty");
        return;
    }
    while(temp != NULL)
    {
        printf("%d\n ", temp → data);
        temp = temp → next;    /*move temp to the next node */
    }
}
```

This function can also be written as a recursive function:

Recursive Function

```
void rec_display(NODE * slist)
{
    NODE * temp = slist;
    if(temp != NULL)
    {
        printf("%d", temp → data );
        rec_display(temp → next);
    }
}
```

5.3 Inserting an Element in the List

In many situations we may have to insert an element in the list. For this, we must know the position where it has to be inserted.

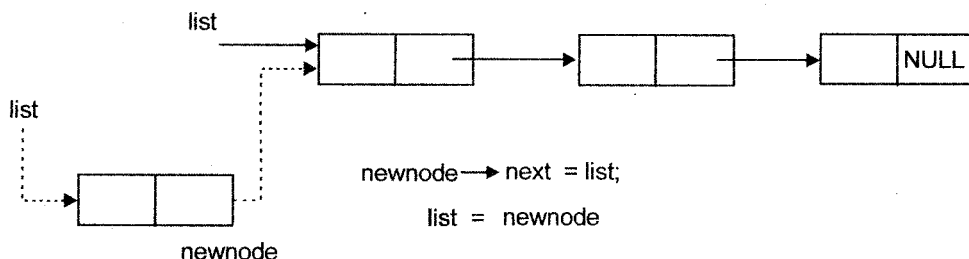
There are three possibilities that we can consider:

- i. **Insert at the beginning:** To insert a new node at the beginning of the list, the pointer pointing to the first node will have to be changed such that it now points to the new node.

Oct.2009 – 4M

Explain with suitable example how data is inserted into a linked list at beginning and at end.

The steps performed are illustrated in the diagram below. The function returns the new address of the first node.



Function: Insert at Beginning

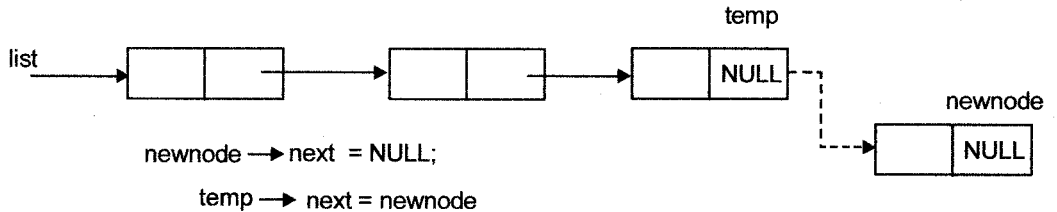
```
NODE * insertbeg (NODE * list, int num)
{
```

```

NODE * newnode;
newnode = NODEALLOC;
newnode → data = num;
newnode → next = list;
list = newnode;
return list;
}

```

- ii. **Insert at the end:** To insert a new node at the end of the list, we will have to move a temporary pointer to the last node and then attach the new node after the last node.



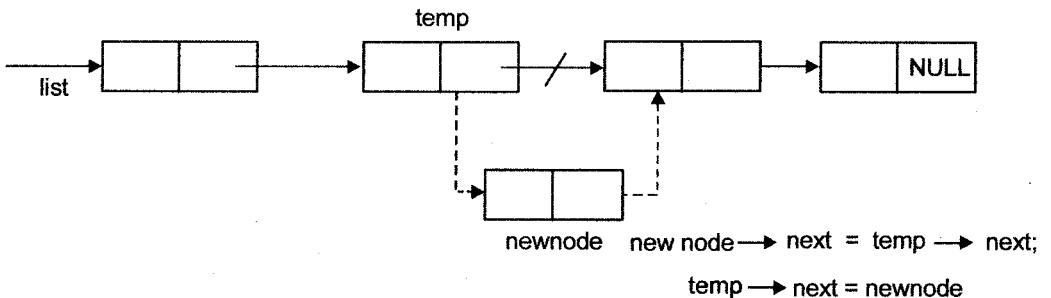
Function: Insert at End

```

NODE * insertend (NODE * list, int num)
{
    NODE * newnode, * temp;
    newnode = NODEALLOC;
    newnode → data = num;
    newnode → next = NULL;
    /* MOVE temp to the last node */
    for (temp = list; temp → next != NULL; temp = temp → next);
    /* Link newnode to temp */
    temp → next = newnode;
    return list;
}

```

- iii. **Insert in a intermediate position:** To insert a node in between, we have to know the position of insertion and move temp to the node before this position. The new node has to be linked between temp and the node after temp.



Function: Insert at intermediate position

```
NODE * insertbetn (NODE* list, int pos, int num)
{
    NODE * newnode, *temp;
    int i;
    newnode = NODEALLOC;
    newnode → next = NULL;
    newnode → data = num;
    /* Move temp to node at pos -1 */
    for (i=1; i < pos-1 && temp→next != NULL; i++)
        temp = temp → next;
    newnode → next = temp → next;
    temp → next = newnode;
    return list;
}
```

All the above three functions can be combined into a single function as below:

Function: Insertion

```
NODE * insert(NODE * list, int pos, int item)
{
    NODE * newnode, * temp;
    int i;
    newnode = NODEALLOC ;
    newnode → data = item ;

    if (pos == 1)          /*insert at beginning */
    {
        newnode → next = list;
        list = newnode ;
        return list;
    }

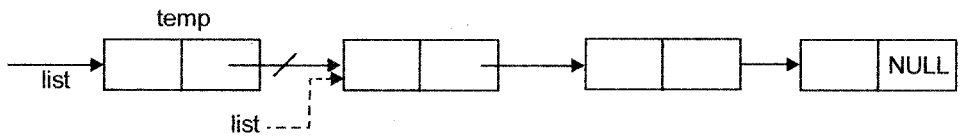
    /*Move temp to node at one position less */
    for(i = 1,temp = list;(i<pos -1)&&(temp→next != NULL);i++)
        temp = temp → next;
    newnode → next = temp → next;
    temp → next = newnode ;
    return list;
}
```


5.4 Deleting an element from the list

For deleting an element, we have to locate the node at the specified position and free the node after linking the nodes before and after the node to be deleted.

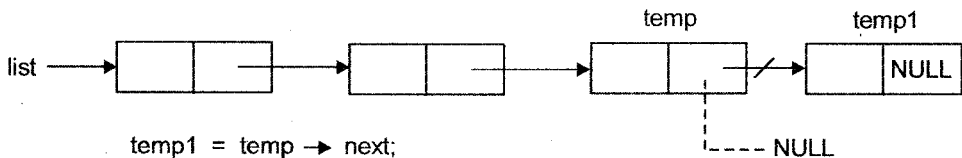
As in the case for insertion, deletion of an element can be done at three positions:

- i. **Deleting the first element:** After deleting the first node from the list, the next node will become the first node and hence, we will have to change the pointer to the list.



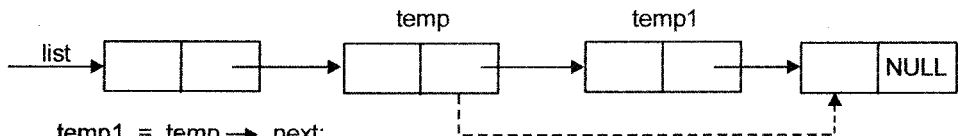
```
temp = list;
list = temp → next;
free(temp);
```

- ii. **Deleting the last element:** To delete the last node, we have to move a temporary pointer to the second last node and then free the last node. The second last node should now contain a NULL link.



```
temp1 = temp → next;
temp → next = NULL;
free(temp1);
```

- iii. **Deleting an intermediate element:** To delete a node in between, we have to move a temporary pointer to the node at position-1 so that the node at the specific position can be deleted by making appropriate links.



```
temp1 = temp → next;
temp → next = temp1 → next;
free(temp1);
```

Function: Delete at a specific position

```

NODE * delete (NODE * list, int pos)
{
    NODE * temp, * temp1 ;
    if (pos == 1) /* Delete the first node */
    {
        temp = list ;
        list = list → next ;
        free(temp);
        return list;
    }
    /* Move temp to the node at position - 1 */
    for (i=1 , temp = list ;(i < pos-1)&&(temp != NULL);i++)
        temp = temp → next;
    temp1 = temp → next ;
    /* temp1 is the node to be deleted */
    temp → next = temp1 → next;
    /* link temp to node after temp1 */
    free (temp1);
    return list;
}

```

1

Apr.2012 – 4M

Write the function to insert and delete a Node in the beginning of a Singly Linked List.

The above function deletes the element at a given position. We can also delete a specific number from the list. This can be done by ‘searching’ the number in the list and then deleting that node which contains the number.

5.5 Searching an Element in the List

To search an element in the list, we will have to traverse the entire list and compare the data in each node. If found, we will return the node address and NULL otherwise.

Algorithm

1. Start
2. list is the pointer to the first node of the list.
3. temp is a temporary pointer for traversal.
4. Accept num i.e. the number to be searched in the list.

5. Make temp point to the first node i.e. temp=slist.
6. pos = 1
7. If temp->data == num then
Display "Element found in the list at position pos"
Go to step 11
8. Move temp to the next node i.e. temp=temp->next.
9. pos = pos + 1
10. If temp ≠ NULL
Go to step 7
11. Return temp.
12. Stop

Function: Search a specific number in the list

```

NODE * search(NODE * list, int num)
{
    NODE * temp;
    for(temp = list; temp != NULL; temp = temp → next)
        if(temp → data == num)
            return temp;
    return temp;
}

```

We shall now write a menu-driven program to implement the above operations on the linked list.



Program : Menu driven Operations on Singly Linked List

```

#include <stdio.h>
/***** GLOBAL DECLARATIONS *****/
typedef struct node
{
    int data;
    struct node *next;
}NODE;

```

```
#define NODEALLOC (NODE *)malloc(sizeof(NODE))
```

1

Oct.2012 – 4M

Write a 'C' Program to search given element into the Singly Linked List.

```
/****** CREATE *****/
NODE * createlist(NODE * list)
{
    NODE * newnode,* temp;
    int i,n;
    printf("\nHow many elements :");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        newnode=NODEALLOC;
        newnode->next=NULL;
        printf("\nEnter the element");
        scanf("%d",&newnode->data);
        if(list==NULL)
            list=temp=newnode;
        else
        {
            temp->next=newnode;
            temp=newnode;
        }
    }
    return list;
}
/****** INSERT *****/
NODE* insert(NODE * list, int n, int pos)
{
    NODE * newnode, *temp;
    int i;
    newnode=NODEALLOC;
    newnode->next=NULL;
    newnode->data=n;
    if(list==NULL)
    { list=newnode;
      return list;
    }
    if(pos==1)          /*** insert at position 1 **/
    {
        newnode->next=list;
        list=newnode;
        return list;
    }
    /** move temp to the node at pos - 1 */
    for(i=1,temp=list;i<=pos-2 && temp->next!=NULL; i++)
        temp=temp->next;
    newnode->next=temp->next;
    temp->next=newnode;
}
```

```
        return list;
    }
}
/***** DISPLAY *****/
void display(NODE * list)
{
    NODE * temp;
    for(temp=list;temp!=NULL;temp=temp->next)
        printf("%d\t", temp->data);
}
/***** DELETE BY POSITION *****/
NODE * deletepos(NODE * list, int pos)
{
    NODE * temp,* temp1;
    int i;
    if(pos==1)        /* Delete the first node */
    {
        temp=list;
        list=list->next;
        free(temp);
        return list;
    }
    for(i=1,temp=list;i<=pos-2 && temp!=NULL; i++)
        temp=temp->next;
    if(temp==NULL)
    {
        printf("\nPosition out of range");
        return list;
    }
    temp1=temp->next;
    temp->next=temp1->next;
    free(temp1);
    return list;
}
/***** DELETE BY VALUE *****/
NODE * deleteval(NODE * list, int num)
{
    NODE * temp=list,* temp1;
    if(temp->data==num)    /* first node */
    {
        temp=list;
        list=list->next;
        free(temp);
        return list;
    }
    for(temp=list;temp->next!=NULL;temp=temp->next )
    {
        if(temp->next->data == num)
```

```
{
    temp1=temp->next;
    temp->next=temp1->next;
    free(temp1);
    return list;
}
}
printf("\nElement not found");
getch();
return list;
}
/***** SEARCH *****/
int search(NODE * list,int num)
{
NODE* temp;
    int i;
    for(i=1,temp=list;temp!=NULL;temp=temp->next,i++)
        if(temp->data==num)
            return i;
    return -1;
}
/***** MAIN FUNCTION *****/
void main()
{
    NODE * list=NULL;
    int choice, n, pos;
do
    {
        printf("\n1: CREATE");
        printf("\n2: INSERT");
        printf("\n3: DELETE BY NUMBER");
        printf("\n4: DELETE BY POSITION");
        printf("\n5: SEARCH");
        printf("\n6: DISPLAY");
        printf("\n7: EXIT");
        printf("\nEnter your choice :");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1 :
            list=createlist(list);
            break;
        case 2:
            printf("\nEnter the element and position :");
            scanf("%d%d", &n, &pos);
            list = insert(list, n, pos);
            display (list);
            break;
        }
```

```
case 3:
    printf("\nEnter the element :");
    scanf("%d",&n);
    list = deleteval(list, n);
    display (list);
    break;
case 4:
    printf("\nEnter the position :");
    scanf("%d", &pos);
    list = deletepos(list, pos);
    display(list);
    break;
case 5:
    printf("\nEnter the element to be searched :");
    scanf("%d",&n);
    pos = search(list, n);
    if(pos== -1)
        printf("\nElement not found");
    else
        printf("\nElement found at position %d", pos);
    break;
case 6:
    display(list);
    break;
}/* end switch */
getch();
} while(choice !=7);
getch();
}
```



5.6 Computation of Length of a Singly Linked List

The length of a linked list is the number of nodes in the list. We can count them by traversing the list from the first node to the last node of the list. The display method seen above can be slightly modified so that we can count the nodes.

Algorithm

1. Start
2. slist is the pointer to the first node of the list.
3. if slist == NULL

Display "Empty List"

Go to 10

4. temp is a temporary pointer for traversal.
5. Initialize count to 0.
6. Make temp point to the first node i.e. temp=slist.
7. If temp \neq NULL
Increment count.
Move temp to the next node i.e. temp=temp->next.
8. Repeat step 7 as long as temp \neq NULL.
9. Return count.
10. Stop.

Function: Length of a List

```
int length (NODE * slist)
{
    NODE * temp = slist ;      /* Store address of first node in temp */
    int count = 0;

    while (temp != NULL)
    {
        count++;
        temp = temp → next;    /*move temp to the next node */
    }
    return count;
}
```

This function can also be written as a recursive function:

Recursive Function 1: Length of List

```
int rec_length(NODE * slist)
{
    NODE * temp = slist;
    static int count;
    if(temp != NULL)
    {
        count++;
        rec_length(temp → next);
    }
    return count;
}
```


Another way of writing the recursive function is given below.

Recursive Function 2: Length of List

```
int rec_length(NODE * slist)
{
    NODE * temp = slist;
    if(temp == NULL)
        return 0;
    return 1 + rec_length( temp->next);
}
```

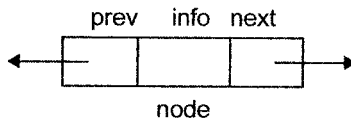
6. Doubly Linked List

1

Oct.2009 – 4M
Explain Doubly Linked List in detail. How it differs from Singly Linked List.

In a singly linked list, each node contains one pointer, which points to the next node. Thus, traversal is possible only in one direction.

A doubly linked list is a linked list in which each node contains two links – one pointing to the previous node and one pointing to the next node.



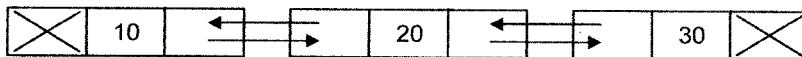
1

Oct.15, Apr.15 – 4M
What is doubly linked list? Explain its node structure.

The node structure will be

```
typedef struct node
{
    int data;
    struct node *prev, *next;
} NODE;
```

Example



Advantages

1. Traversal in both directions is possible.
2. Operations like insertion, deletion, searching can be efficiently done.

Disadvantage

Extra storage is needed for the pointers.

1. **Creating a doubly linked list:** This is the same as creating a singly linked list except that every node has to be linked using two pointers - prev and next.

Function

```

NODE * createlist(NODE *dlist)
{
    NODEPTR temp, newnode,
    int i,n
    printf ("How many nodes:");
    scanf ("%d",&n);
    for (i=1; i<=n;i++)
    {
        newnode = NODEALLOC;
        newnode → next = newnode → prev = NULL;
        printf("\n Enter the element:");
        scanf ("%d",&newnode → data);
        if (dlist == NULL)
            dlist = temp = newnode;
        else
        {
            temp → next = newnode;
            newnode → prev = temp;
            temp = newnodes;
        }
    }
    return dlist;
}

```

1

Oct.2012 – 4M

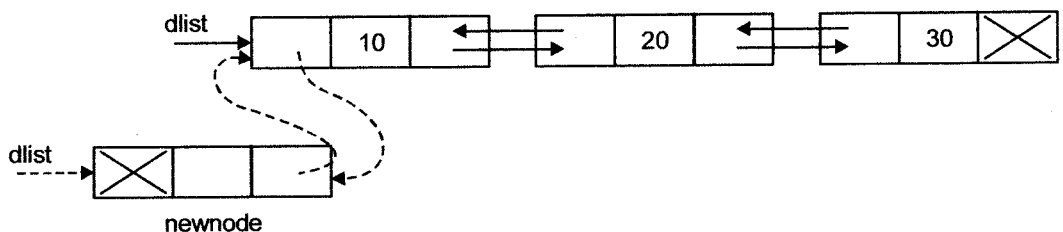
Write a function to Insert Node at the specified position in Doubly Linked List.

1

Oct.2014 – 4M

Write a function to create and display doubly linked list

2. **Inserting node at the beginning**

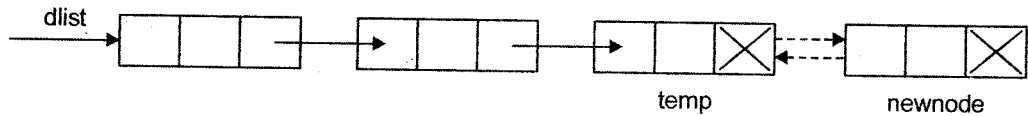


```

newnode → next = dlist;
dlist → prev = newnode;
dlist = newnode

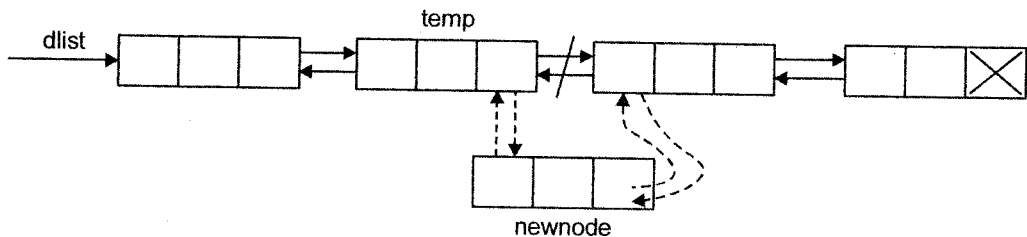
```

3. Inserting a node at the end



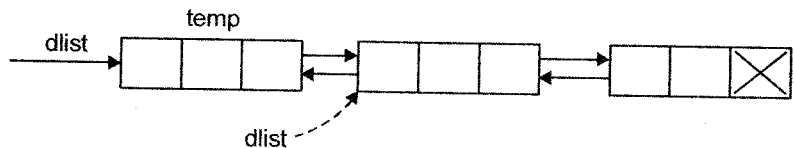
```
temp → next = newnode;
newnode → prev = temp;
```

4. Inserting a node in between



```
newnode → next = temp → next;
temp → next → prev = newnode;
temp → next = newnode;
newnode → prev = temp;
```

5. Deleting the first node



```
temp → prev → next = temp → next;
temp → next → prev = temp → prev;
free(temp);
```

2

Apr.11, Oct.10 – 4M
Write a function to delete
a node from Doubly
Linked List.

For simplicity of operations, it is preferable to keep an extra node called 'header' node at the beginning of the list. The following program illustrates the various operations on a doubly linked list, which contains an additional node at the beginning of the list.

**Program : Menu driven program for Doubly Linked List**

```
/** list is an empty node at the beginning **/  
#include <stdio.h>  
typedef struct node  
{  
    int data;  
    struct node *next,*prev;  
} NODE;  
#define NODEALLOC (NODE *)malloc(sizeof(NODE))  
  
/***** CREATE LIST *****/  
void create(NODE *list)  
{  
    int i,n;  
    NODE *newnode,*temp=list;  
    printf("\nHow many nodes :");  
    scanf("%d",&n);  
    for(i=1;i<=n;i++)  
    {  
        newnode=NODEALLOC;  
        scanf("%d",&newnode->data);  
        newnode->next=NULL;  
        temp->next=newnode;  
        newnode->prev=temp;  
        temp=newnode;  
    }  
}  
  
void insert(NODE *list, int n, int pos)  
{  
    NODE *temp=list,*newnode,*temp1;  
    int i;  
    newnode = NODEALLOC;  
    newnode->data=n;  
    newnode->next=NULL;  
    for(i=1;(i<pos) && (temp->next!=NULL);i++)  
        temp=temp->next;  
    temp1=temp->next;  
    newnode->next=temp1;  
    temp1->prev=newnode;  
    temp->next=newnode;  
    newnode->prev=temp;  
}  
  
void display(NODE *list)  
{  
    NODE *temp;  
    for(temp=list->next;temp!=NULL;temp=temp->next)
```

```
printf("%d\t",temp->data);
}
void delete(NODE *list, int pos)
{
    NODE *temp,*temp1,*temp2;
    int i;
    /** Move temp to the node to be deleted ****/
    for(i=1,temp=list;(i< pos) && (temp!=NULL); i++)
        temp=temp->next;
    if(temp==NULL)
    {
        printf("Position Out of range");
        return;
    }
    temp1=temp->next;
    temp2=temp1->next;
    temp->next=temp2;
    temp2->prev=temp;
    free(temp1);
}
void main()
{
    NODE *list;
    int n,pos,choice;
    list = NODEALLOC;
    list->next = list->prev =NULL;
    do
    {
        printf("\n1: Create");
        printf("\n2: Insert");
        printf("\n3: Delete");
        printf("\n4: Display");
        printf("\n5: Exit");
        printf("\n\nEnter your choice :");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                create(list);
                break;
            case 2:
                printf("\nEnter the element and position ");
                scanf("%d%d",&n,&pos);
                insert(list,n,pos);
                display(list);
                break;
            case 3 :
                printf("\nEnter the position ");
```

```

scanf("%d",&pos);
delete(list,pos);
display(list);
break;
case 4:
display(list);
break;
}
} while(choice !=5);
getch();
}

```



7. Circular Linked List

2

Apr.11, Oct.10 – 2M

Explain Singly Circular Linked List.

A circular list is different from a linear list because the last node points to the first node.

Using a circular list, we can traverse the list starting from any node to any other node. The insertion, deletion and display operations can be performed in a similar manner as for a singly or doubly linked list.

The only point to remember is that in a circular list, there is no NULL at the end of the list. The last node points to the first node.

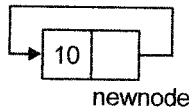
Hence the list termination condition will be different.

We shall see the functions to perform operations like create, traverse, insert and delete on a singly circular linked list.

Creation

The steps in creation of a circular list are

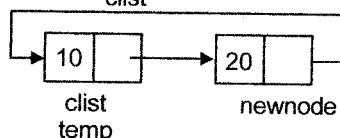
1. `clist = NULL`



2. Add 10

`clist = newnode`
`newnode → next = clist`

3. Add 20



`temp = clist`
`temp → next = newnode`
`newnode → next = clist`

2

Oct.2015- 4M

Write a function to create and display circular linked list.

Oct.2012- 4M

Write the function to insert node at the beginning of the Circular Singly Linked List.

Apr.2010- 4M

Write a C program to create node in circular linked list.

```

NODE* create(NODE* clist)
{
    NODE* newnode, * temp = clist;
    char ans;
    do
    {
        newnode = NODEALLOC;
        printf("\n Enter the value :");
        scanf("%d",&newnode → data);
        if(clist == NULL)
        {
            clist = temp = newnode;
            newnode → next = clist;
        }
        else
        {
            temp → next = newnode; /*Link newnode */
            newnode → next = clist ;
            temp = newnode ;
        }
        printf("\n Any more nodes ?");
        ans = getchar();
    } while(ans == `y`);
    return(clist);
}

```

Traversal

```

void display(NODE * clist)
{
    NODE * temp = clist;
    if(clist == NULL)
    {
        printf("List is empty");
        return; }
    do
    {
        printf ("%d \t", temp->data);
        temp=temp->next;
    } while (temp != clist);
}

```

Insertion

```
NODE * insert (NODE * clist , int n , int pos)
{
    NODE * temp = clist, *newnode;
    newnode = NODEALLOC;
    newnode->data = n;
    if (pos == 1)
    {
        /** Move temp to last node */
        for (temp = clist; temp ->next != clist; temp= temp->next);
        newnode->next = clist;
        temp->next = newnode;
        clist = newnode;
        return clist;
    }
    for(i=1; i<pos-1 && temp->next !=clist ; t++)
        temp = temp->next;
    newnode->next = temp->next;
    temp->next=newnode;
    return clist;
}
```

Deletion

```
NODE * delete (NODE * clist , int pos)
{
    NODE * temp = clist, *temp1;

    if ( pos == 1)
    {
        /** Move temp to last node */
        for ( temp = clist; temp ->next != clist; temp= temp->next);

        temp1=clist;
        clist = clist -> next;
        temp->next = clist;
        free ( temp1);
        return clist;
    }
    for(i=1; i<pos-1 && temp->next !=clist ; t++)
        temp = temp->next;
    if( i < pos -1 )
    {
        printf("Position out of range");
        return clist;
    }
    temp1 = temp->next;
    temp->next = temp1->next;
    free(temp1);
    return clist;
}
```


Solved Examples

1

Apr.2012 – 4M

1. What is a Doubly Linked List? Write a 'C' program to add new node at the end of a Doubly Linked List.

24, 6, 75, 12, 60, 15

Solution

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
# define NULL 0
typedef struct list
{
    int data;
    struct list *next;
    struct list *prev;
}node;
node *head = NULL;
node *prev = NULL;
node* create_node()
{
    node *new1;
    new1 = malloc(sizeof(node));
    printf("\nEnter data:");
    scanf("%d",&new1->data);
    new1->next = NULL;
    new1->prev = NULL;
    return(new1);
}
void add()
{
    int insdata;
    node *new1,*temp;
    printf("\nEnter data after which to insert:");
    scanf("%d",&insdata);
    temp = head;
    while(temp != NULL)
    {
        if(temp->data == insdata)
        {
            new1 = create_node();
            new1->next = temp->next;
            temp->next = new1;
            return;
        }
        else
            temp = temp->next;
    }
}
```

```

void display()
{
    node *temp;
    temp = head;
    while(temp != NULL)
    {
        printf("\n%d",temp->data);
        temp = temp->next;
    }
}

void main()
{
    int ch,ch1,x;
    clrscr();
    do
    {
        printf("\nMENU");
        printf("\n1.create a node\n2.Add a
node\n3.Display\n4.exit");
        printf("\nEnter choice");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: head = create_node();
                break;
            case 2: printf("\n Add a node");
                add();
                break;
            case 3: display();
                break;
            case 4: exit(0);
        }
        printf("\n\n");
    }while(ch != 4);
    getch();
}

```

2. Write a 'C' program to reverse a Singly Circular Linked List.

Solution

```

#include<stdio.h>
#include<malloc.h>
struct node
{ int info;
  struct node *link;
}*start;
void main()
{

```

```
int i,n,item;
start=NULL;
printf("How many nodes you want : ");
scanf("%d",&n);
for(i=0;i< n;i++)
{
    printf("Enter the item %d : ",i+1);
    scanf("%d",&item);
    create_list(item);
}
printf("Initially the linked list is :\n");
display();
reverse();
printf("Linked list after reversing is :\n");
display();
}/*End of main()*/
void create_list(int num)
{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=num;
    tmp->link=NULL;
    if(start==NULL)
        start=tmp;
    else
        { q=start;
          while(q->link!=NULL)
            q=q->link;
          q->link=tmp;
        }
}/*End of create_list() */
void display()
{
    struct node *q;
    if(start == NULL)
    {
        printf("List is empty\n");
        return;
    }
    q=start;
    while(q!=NULL)
    {
        printf("%d ", q->info);
        q=q->link;
    }
    printf("\n");
}/*End of display()*/
void reverse()
```

```

{
    struct node *p1,*p2,*p3;
    if(start->link==NULL) /*only one element*/
        return;
    p1=start;
    p2=p1->link;
    p3=p2->link;
    p1->link=NULL;
    p2->link=p1;
    while(p3!=NULL)
    {
        p1=p2;
        p2=p3;
        p3=p3->link;
        p2->link=p1;
    }
    start=p2;
} /*End of reverse() */

```

3. Write a program to add a node in a Doubly Linked List at the beginning and at the end.

Solution

```

#include<stdio.h>
#include<malloc.h>
struct node /* structure to represent a node of doubly linked list*/
{
    struct node *prev;
    int info;
    struct node *next;
} *start;
void main() /* main function */
{
    int choice, n ,m, i;
    start=NULL; /*initialize header node to NULL */
    while(1)
    {
        printf("1.Create List\n");
        printf("2.Add node at begining\n");
        printf("3.Add node at the end of list \n");
        printf("4.Display\n");
        printf("5.exit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: printf("How many nodes you want : ");
                    scanf("%d", &n);

```

1

Oct.2011 - 4M

```
for(i=0;i<n; i++)
{
    printf("Enter the element: ");
    scanf("%d", &m);
    create_list(m);
}
break;
case 2:printf("Enter the element: ");
        scanf("%d", &m);
        addatbeg(m);
        break;
case 3:printf("Enter the element: ");
        scanf("%d", &m);
        addatend(m);
        break;
case 4:display();
        break;
case 5: exit();
default:printf("Wrong choice\n");
}
/*End of switch*/
}
/*End of while*/
}
/*End of main()*/
create_list(int num) /*cerate function to create a single
node */
{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=num;
    tmp->next=NULL;
    if(start==NULL)
    {
        tmp->prev=NULL;
        start->prev=tmp;
        start=tmp;
    }
    else
    {
        q=start;
        while(q->next!=NULL)
            q=q->next;
        q->next=tmp;
        tmp->prev=q;
    }
}
/*End of create_list()*/
addatbeg(int num) /*function to add node at the beginning of
the list */
{
    struct node *tmp;
```

```

tmp=malloc(sizeof(struct node));
tmp->prev=NULL;
tmp->info=num;
tmp->next=start;
start->prev=tmp;
start=tmp;
}/*End of addatbeg()*/
addatend(int num)
{
    struct node *tmp,*q;
    int i;
    tmp=malloc(sizeof(struct node) );
    tmp->info=num;
    tmp->next=NULL;
    tmp->prev=NULL;
    q=start;
    while(q->next != NULL)
        q=q->next;
    q->next=tmp;
    tmp->prev=q;
}/*End of addatend() */
display()/*display function to print all nodes in the list */
{
    struct node *q;
    if(start==NULL)
    { printf("List is empty\n");
      return;
    }
    q=start;
    printf("List is :\n");
    while(q!=NULL)
    { printf("%d ", q->info);
      q=q->next;
    }
    printf("\n");
}/*End of display() */

```

4. Write a function to add node at given position in singly linked list.

Solution

```

node *insert(node *head,int x, int key)
{
    node *p,*q;
    p=(node*)malloc(sizeof(node));

```

```
p->data=x;
if (key== -1)
{ p->next=head;
  head=p;
}
else
{
  q=head;
  while(key!=q->data && q!= NULL)
  q=q->next;
  if(q!=NULL)
  { p->next=q->next;
    q->next=p;
  }
}
return(head);
}
```



PU Questions

2 Marks

1. What is use of “typedef” keyword? **[Oct.2015 – 2M]**
2. What are the advantages of array over linked list? **[Oct.2015 – 2M]**
3. What is linked list structure? **[Apr.2015 – 2M]**
4. What is Linked List? State its types. **[Apr.2012 – 2M]**
5. Explain Singly Circular Linked List. **[Apr.11, Oct.10 – 2M]**
6. What do you mean by Traversing a Linked List? **[Oct.2009 – 2M]**

4 Marks

1. Write a function to create and display circular linked list. **[Oct.2015 – 4M]**
2. What are the drawbacks of sequential storage? **[Oct.2015 – 4M]**
3. Write a recursive function for erasing linked list. **[Oct.2015 – 4M]**
4. Write a function to display circular linked list in reverse order. **[Oct.15, Apr.15 – 4M]**

- [Apr.2010 – 4M]** 25. Write a 'C' program to remove first node of singly linked list and insert it at the end of a list.
- [Apr.2010 – 4M]** 26. Write a 'C' program to create a node in doubly linked list.
- [Oct.2010 – 4M]** 27. Write a function to delete a node from Doubly Linked List.
- [Oct.2009 – 4M]** 28. Explain with suitable example how data is inserted into a linked list at beginning and at end.
- [Oct.2009 – 4M]** 29. Explain Doubly Linked List in detail. How it differs from Singly Linked List

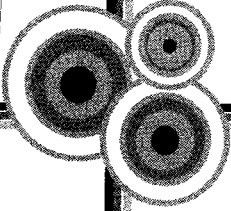
5. Write a function to merge given two singly linked lists. **[Oct.2015 – 4M]**
6. Write a function to remove last node of singly linked list and add it at the beginning. **[Apr.2015 – 4M]**
7. What is doubly circular linked list? Explain its node structure. **[Apr.2015 – 4M]**
8. Write a function to merge given two singly linked lists. **[Apr.2015 – 4M]**
9. Write a function to display circular linked list in reverse order. **[Apr.2015 – 4M]**
10. Write a function to remove given node from singly linked list and add it at the end of list. **[Apr.2015 – 4M]**
11. Write a function to add node at first position in singly linked list. **[Apr.2015 – 4M]**
12. Write a function to create and display circular doubly linked list. **[Oct.2014 – 4M]**
13. Write a function to add node at given position in singly linked list. **[Oct.2014 – 4M]**
14. Write a function to create and display doubly linked list. **[Oct.2014 – 4M]**
15. Write a 'C' Program to search given element into the Singly Linked List. **[Oct.2012 – 4M]**
16. Write the function to insert node at the beginning of the Circular Singly Linked List. **[Oct.2012 – 4M]**
17. Write a function to Insert Node at the specified position in Doubly Linked List. **[Oct.2012 – 4M]**
18. What are the advantages of a Linked List over an Array? **[Apr.2012 – 4M]**
19. Write the function to insert and delete a Node in the beginning of a Singly Linked List. **[Apr.2012 – 4M]**
20. Write a C program to create and display Singly Circular Linked List. **[Apr.2012 – 4M]**
21. What is a Doubly Linked List? Write a 'C' program to add new node at the end of a Doubly Linked List. 24, 6, 75, 12, 60, 15 **[Apr.2012 – 4M]**
22. Explain types of Link Lists. **[Apr.2011 – 4M]**
23. Write a function to delete a node from doubly linked list. **[Apr.2011 – 4M]**
24. Write a C program to create node in circular linked list. **[Apr.2010 – 4M]**

- [Apr.2010 – 4M]** 25. Write a 'C' program to remove first node of singly linked list and insert it at the end of a list.
- [Apr.2010 – 4M]** 26. Write a 'C' program to create a node in doubly linked list.
- [Oct.2010 – 4M]** 27. Write a function to delete a node from Doubly Linked List.
- [Oct.2009 – 4M]** 28. Explain with suitable example how data is inserted into a linked list at beginning and at end.
- [Oct.2009 – 4M]** 29. Explain Doubly Linked List in detail. How it differs from Singly Linked List


VISION

Chapter 4

STACK AND QUEUE



1. Introduction

Data structures play a very crucial role in computer science. One of the most important data structure is the stack. In this chapter we will study this data structure, its implementation and see why it plays such an important role in Computer Science.

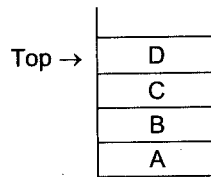
2. Definition of a Stack

A stack is an ordered collection of items into which items may be inserted and deleted from one end called the top of the stack. The stack operates in a **LIFO (Last In First Out)** manner i.e. the element which is put in Last is the First to come out.

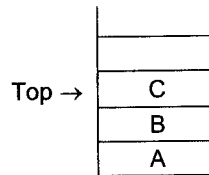
Example

A common example is a pile of plates kept one above the other. Only the topmost plate can be taken and any new plate has to be put at the top.

This is an example of a stack. The following diagram illustrates a stack after items A, B, C and D have been put into it.

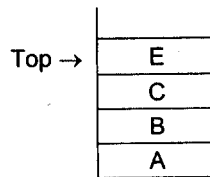


When we want to remove an item from stack, D will be the first to be removed and the top points to item C.



Stack after D is removed

A new element E will be added above C.



Adding E to the stack

Thus, the stack is dynamic in nature i.e. its size can increase or shrink during runtime.

3. Primitive Operations on Stack

There are primitive operations that can be performed on a stack

1. **Create:** Creates a new stack. This operation creates a new stack which is empty.
2. **Push:** Adds an element to the stack. The push operation inserts a new element at the top of the stack. Top now points to this new element.

3. **Pop:** Removing an element from the stack. The pop operation removes the topmost element from the stack thereby reducing the size of the stack by 1. The element below it now becomes the topmost element.
4. **Isempty:** Checks whether a stack is empty. This operation returns TRUE if the stack is empty and False otherwise. This is required for the pop operation because we cannot pop from an empty stack.

2

Oct.2010 – 2M

What is a Stack? Discuss Operations Performed on Stack.

Oct.2009 – 2M

Explain different operations performed on a Stack.

4. Implementation of a stack

A stack can be implemented in two ways:

1. Static
2. Dynamic

4.1 Static Implementation of Stack

Since the stack is an ordered list of items, we can implement it using a familiar and similar data structure i.e. arrays. The array elements have sequential representation which is required for a stack also. However, there are fundamental differences between a stack and an array. Some differences are as follows:

	Array	Stack
1.	Any element of an array can be accessed.	Only the topmost element can be accessed.
2.	An array essentially contains homogenous elements.	A stack may contain diverse elements.
3.	An array is a static data structure i.e. its size remains constant.	A stack is a dynamic data structure i.e. its size shrinks and grows as elements are pushed and popped.
4.	There is an upper limit on the size of the array, which is specified during declaration.	Logically, a stack can grow to any size.

Hence, if we are implementing a stack using an Array, there will be limitations imposed on the stack due to the restrictions on the array.

One important limitation will be on the stack size. Since an array can be of a fixed size only, the stack implemented using an array will also have an upper bound.

We will now also have to check if the upper bound has been reached before an element is Pushed into the stack. Thus, one more operation i.e. **Isfull** will have to be performed on the stack. This operations gives TRUE if the upper limit has been reached i.e. the stack is full.

We shall also define a new operation called 'Initstack' which will initialize the top to -1 to indicate an initially empty stack.

4.2 Declarations and Functions

Declaring a Stack

A stack to be implemented using an array will require

1. An array of a fixed size
2. An integer called top which stores the index or position of the topmost element.

We can use a structure for the above purpose.

```
#define MAX 100
#define EMPTY -1
#define FULL MAX-1
struct stack
{
    int top;
    int items[MAX];
};
```

This declaration only specifies the template. The actual stack can be declared as

```
struct stack s1;
Using typedef
```

The typedef can be used to create a new data type called STACK. This can be done as

```
typedef struct
{
    int top;
    int items[MAX];
} STACK;
```

The declaration

```
STACK s1, s2;
```

declares s1 and s2 as two stack variables.

Stack with objects of different data types.

Although an array allows elements only of the same data type, a stack can be designed to store elements of multiple data types. A union can be used for this purpose as shown.

```
#define INT 1
#define FLOAT 2
#define CHAR 3
struct element_type
{ int eletype;
  union
  {
    int intvalue;
    float floatvalue;
    char charvalue;
  } element;
};
struct stack
{ int top;
  struct element_type items[MAX];
};
```

Depending upon the value stored in eletype, either an integer, float or character value will be stored in the union.

2

Oct.2014 – 4M

What is stack? Explain its operations in details.

4.3 Operations on the Stack

1. **Initialize a stack:** When a stack variable is declared, the integer top has to be initialized to indicate an empty stack.

Since we are using an array, the first element will occupy position 0. Hence, to indicate an empty stack, top has to be initialized to -1. The function will be written as

```
void initstack(STACK *ps)
{ ps → top = -1;}
```

calling the function

```
initstack(&s1);
```

2. **Checking whether stack is empty:** An empty stack can be tested from the value contained in top. If top contains -1, it indicates an empty stack.

```
int isempty(STACK *ps)
{
```

```

    return(ps → top == EMPTY);
}

```

calling the function

```
if(isempty(&s1)
```

—
—

3. **Checking for a Full Stack:** If the value of top reaches the maximum array index i.e. MAX-1, no more elements can be pushed into the stack.

```
int isfull(STACK *ps)
{
    return(ps → top == FULL);
}

```

calling the function

```
if (isfull(&s1))
```

—
—

4. **The Push Operation:** An element can be pushed into the stack only if it is not FULL. In such a case, the top has to be incremented first and the element has to be put in this position.

```
void push(STACK *ps,int n)
{
    if(isfull(ps))
        { printf("\n stack overflow");
          return;
        }
    else
        { ++ps → top;
          ps → items [ps → top] = n;
        }
}

```

Calling the function

```
push(&s1,n);
```

The statements in the else part can also be written as

```
ps → items [++( ps → top) ] = n;
```

5. **The Pop Operation:** An element can be removed from the stack if it is not empty. The topmost element can be removed after which top has to be decremented.

```
int pop(STACK *ps)
{
    return(ps → item [ps → top--]);
}

```


calling the function

```
if(isempty(&s1))
    printf("\n Stack Empty");
else
    printf("%d", pop(&s1));
```

We shall now write a simple menu driven program to implement a stack using the functions and declarations as written above.

/ Illustrate implementation of stack using Array */*

```
main()
{
    STACK s1;
    int ch,n;
    do
    {
        printf("\n1: PUSH");
        printf("\n2: POP");
        printf("\n3: EXIT");
        printf("\n\n Enter your option :");
        scanf("%d",&ch);
        switch(ch){
            case 1 : printf("Enter the data to be pushed");
                    scanf("%d",&n);
                    push(&s1,n);
                    break;
            case 2 :
                    if(isempty (&s1))
                        printf("\n Stack is empty");
                    else
                        printf("\n The popped value is %d", pop (&s1));
                    break;
            case 3 : printf("\n Exiting ....");
        } /*end of switch */
    } while(ch!=3);
}/* end of main */
```

Output

```
1: PUSH
2: POP
3: EXIT
Enter your option: 2
Stack is empty
1: PUSH
2: POP
```

```
3: EXIT
Enter your option: 1
Enter the data to be pushed: 20
```

```
1: PUSH
2: POP
3: EXIT
Enter your option: 2
The popped value is 20
```

```
1: PUSH
2: POP
3: EXIT
Enter your option: 3
Exiting...
```

Another *example* of stack usage can be seen in the reversing of a string. Each character of the string is pushed into a stack and after all have been pushed, the characters are popped out one by one. This will give us the characters in the reversed order.

The stack has to store characters and hence the array has to be declared of type char.

/* Reversing a string using a stack */

```
main()
{
    STACK s;
    char str[20];
    int i = 0; initstack(&s);
    printf("\n Enter the string :");
    gets(str);
    while(str[i] != '\0');
    {push(&s, str[i]);
    i++;
    }
    i=0;
    while(!isempty(&s))
    { str[i] = pop(&s);
    i++;
    }
    str[i] = '\0';
    printf("\n The reversed string is :");
    puts(str);
}
```

Output

```
Enter the string: Computer
The reversed string is: retupmoC
```

4.4 Dynamic Implementation of stack

We have seen how a stack can be represented using sequential representation. However, in that method, there is a limitation on the size of the array. Moreover, if less elements are stored, memory will be wasted.

In the linked organization,

- i. The stack can grow to any size,
- ii. We need not have prior knowledge of the number of elements,
- iii. When an element is popped, the memory can be freed. Thus memory is not unnecessarily occupied.
- iv. Since random access to any element is not required in a stack, the linked representation is preferred over the sequential organization.

Apr.11, Oct.10 – 4M
Explain dynamic representation of Stack.

Oct.2009 – 4M
Explain static and dynamic representation of Stack.

To represent a stack dynamically, we will have to allocate memory whenever a new element has to be pushed into the stack and free memory when an element is popped.

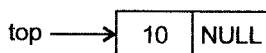
The stack will have to be maintained as 'linked list'. For this purpose, we will have to link the stack elements to each other. This can be done by defining a 'Node' structure as follows.

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

Since we have to allocate and deallocate memory dynamically, we will have to use pointers. Hence top will be defined as a pointer to the first node of the list.

Steps in creating a Dynamic Stack

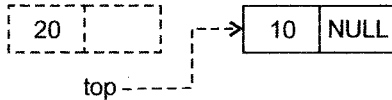
1. top = NULL
2. Push 10



3. Push 20



4. Pop



As seen from above, every new element pushed will be added to the beginning of the list, and will be pointed to by top.

The various functions will be as follows:

1. Initializing the stack

```
void initstack()
{
    top = NULL;
}
```

2. Push

```
void push(int n)
{
    struct node * newnode;
    newnode = (struct node *) malloc (sizeof (struct node));
    newnode → element = n;
    newnode → next = top;
    top = newnode;
}
```

3. Checking an empty condition

```
int stackempty()
{
    return(top == NULL);
}
```

4. Pop

```
int pop()
{ struct node*temp;
  int n;
  temp = top ;
  n = top → element;
  top = top → next;
```

```
    free(temp);
    return(n);
}
```

Since we are using dynamic allocation, the stack can grow to any size. Hence, the stackfull condition will not arise.

Note: For the above functions, top is assumed to be a global pointer.

Program to implement a stack dynamically.

```
struct node
{
    int element;
    struct node* next;
} * top ;      /* top is a global pointers*/
main()
{
    int ch,n;
    do
    { printf("\n1: PUSH");
      printf("\n2: POP");
      printf("\n3: EXIT");
      printf("\n\n Enter your option :");
      scanf("%d",&ch);
      switch(ch)
      { case 1:printf
          scanf
          push(n);
          break;
        case 2:if(isempty())
          printf("stack underflow");
          else
          printf("The popped value is %d", pop());
        }
    } while(ch != 3);
}
```

3

Oct.12, Apr.11 – 2M
What are the Applications of Stack?

Oct.2011 – 4M
What is stack? Discuss various applications of stack.

5. Applications of Stack

Stacks are widely used in operating systems, by compiler and by applications. Some of the applications are:

1. Subroutine calls, Recursion
2. Interrupt handling
3. Interconversion between Infix, Postfix and Prefix expressions.
4. Matching Parentheses in an expression

1

Oct.2011 – 2M
What is Recursion?

5.1 Recursion

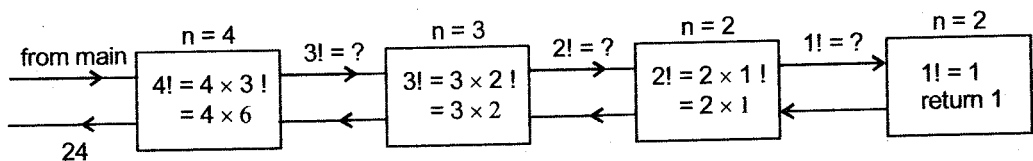
A function which calls itself is called a recursive function. The compiler uses a stack to store the values before the next function call is given.

After the last function call is executed, the compiler pops values from the stack to resume execution of the previous function call.

As an *example*, let us consider the recursive function to calculate the factorial of a number.

```
unsigned long int fact(int n)
{ if(n<=1)
  return 1;
  else
  return(n*fact(n-1));
}
```

Sequence of function calls given for $n = 4$ are



Each time the fact function is called, the value of 'n' is pushed into the stack, so that when control returns from the called function, the value of 'n' can be multiplied to the returned value.

After the last function call finishes execution and a value is returned, the value of n is popped from the stack and multiplied with the returned value. This process continues till the first function call is reached.

A similar strategy is used when control has to be transferred to a subroutine or to process the interrupt subroutine. In order that the execution of the calling routine is resumed, the values have to be pushed onto a stack and popped after the execution of the subroutine is over.

5.2 Polish and Reverse Polish Notations

An expression consists of operators and operands. The operands may be identifiers and constants and operators are symbols representing various operations. An expression can be written in three formats depending upon the placement of operators with respect to the operands.

Infix

In the infix notation the operand is placed between the operands. Example $A + B$. This operator is applied to the two operands surrounding it. Hence it may be necessary to explicitly specify the order of operations by using parentheses.

Example

If, we want to divide a number A by the sum of numbers B and C , the expression will be $A/B + C$. However, this will be wrong since A/B will be performed first. Hence the correct expression will be $A / (B + C)$.

This problem of ambiguity can be removed by using two non-parenthesized formats called prefix and postfix forms.

Prefix or Polish

In the prefix form, the operator precedes the two operands

Example $+AB$

Postfix or Polish

In this format, the operator follows the operands *Example* $AB+$

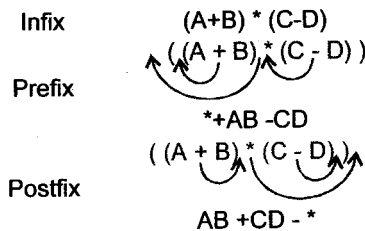
Converting from Infix to Prefix and Postfix

Infix	Prefix	Postfix
i. $A + (B * C)$	$(A+(B * C))$ $(A+ * BC)$ $+A * BC$	$(A+(B * C))$ $(A+BC *)$ $ABC * +$
ii. $(A+B) * (C-D)$	$(A+B) * (C-D)$ $(+AB * (C-D))$ $(A+B * -CD)$ $*+AB -CD$	$((A+B) * (C-D))$ $(AB+ * (C-D))$ $(AB+ * CD-)$ $AB + CD - *$

The above conversions can also be done by the following steps.

- Completely parenthesize the infix expression to specify the order of operations.
- Move each operator outside its corresponding left parenthesis (for Prefix) or right parenthesis (for Postfix)
- Remove all parentheses.

Example



Importance of Prefix and Postfix notations

Since they are free from parentheses, their evaluation is simplified. The compiler converts an infix expression to the postfix format before evaluating it.

Algorithm to Convert an Infix expression to Postfix

To convert an infix expression to postfix, we will need

- A string containing the infix expression, which is parenthesized.
- A stack (opstack) which will store the operators.
- A string to store the postfix expression.

Algorithm

- Opstk is the empty stack.
- Read a character from the infix string.

3. If it is an operand add it to the postfix string.
4. If it is an operator or opening bracket (add it to the opstk)
5. If the character is closing bracket,


```
ch = pop()
while(opstk is not empty and ch is not '(')
{
    add ch to postfix string
    ch = pop()
}
```
6. Repeat from 2 till the infix string does not end.
7. While stack is not empty, pop from stack and add to postfix string.
8. Stop.

Let us now evaluate this algorithm by taking examples.

- a. Infix string $\rightarrow (A+B)*C$

Next Input character	Postfix string	Opstk
((
A	A	(
+		(+)
B	AB	(+)
)	AB +	
*		*
C	AB +C	*
	AB +C*	

- b. Infix string $((A+B) * (C-D)) / E$

Next Input character	Postfix string	Opstk
((
(((
A	A	((
+		((+)
B	AB	((+)
)	AB+	(
*		(*
((* (
C	AB +C	(* (
-		(* (-
D	AB +CD	(* (-
)	AB +CD -	(*
)	AB + CD -*	
/	AB +CD -*	/
E	AB +CD -*E	/
	AB+CD -*E/	

Oct.14,12 – 4M

Write an algorithm to convert Infix Expression to Postfix Expression.

Apr.2011 – 4M

Explain algorithm to evaluate infix to postfix expression.

Oct.2010 – 4M

Explain algorithm to convert infix expression to its equivalent postfix expression.

Apr.2010 – 4M

Explain algorithm to convert infix expression to its equivalent postfix expression.

Evaluating a Postfix expression

To evaluate a postfix expression we need.

1. A postfix expression in a string like $ab + c^*$
2. A stack to store the operands (opndstk).

1

Oct.2009 – 4M

Explain algorithm to evaluate postfix expression.

The algorithm is

1. Opndstk is the empty stack.
2. Read a character from the postfix string.
3. if it is an operand
push it into the stack
else
pop two operands from the stack
apply the operator to these two operands
push the result into the stack
4. if postfix string has not ended go to 2
5. Pop from stack and display.

Example

We will analyze the above algorithm for the postfix string $AB + CD - *$ (corresponding to infix $(A+B)*(C-D)$). If the values are 5, 4, 6 and 2 respectively, the contents of the stack operands and result after each iteration will be.

Character	Opndstk	Operand 1	Operand 2	result
5	5			
4	5,4			
+	9	5	4	9
6	9,6			
2	9,6,2			
-	9,4	6	2	4
*	36	9	4	36

5.3 Interconversion between Infix, Postfix and Prefix Expressions

Converting an Infix expression to Prefix

Example

Infix $((A+B) * C)$

Prefix $* + ABC$

We need:

1. An infix string fully parenthesized
2. A stack to store operators (opstk)
3. A stack to store operands (opndstk)

Algorithm

1. Read a character from infix string
2. If it is '(' or operator
push in opstk
3. If it is an operand
push it in opndstk
4. If ')'


```
while((ch = pop(opstk)) != '(')
{
    op2 = pop(opndstk)
    op1 = pop(opndstk)
    Form an expression as ch op1 op2
    Push expression in opndstk
}
```
5. Repeat from 1 till infix string does not end.
6. Pop from opndstk and display.
7. stop.

Analysis

Infix string ((A+B) *C)

Char	optstr	Opndstk	Expression
((
(((
A	((A	
+	((+	A	
B		A,B	
)	(+A B	+A B
*	(*	+A B	
C	(*	+A B,C	
)		*+A B C	*+A B C

Conversion from Prefix to Infix

Example

Prefix * + A B C

Infix ((A+B) *C)

We need:

1. The prefix string
2. A stack to store operators and infix expression.

1

Oct.2011 – 4M

Write an algorithm for prefix to infix conversion of an expression.

Algorithm

1. symb is next input character
2. If symb is an operator
Push symb into stack
3. If symb is not operator

```

expr = symb
while(top of stack contains operand)
{
    opnd = pop();
    opr = pop();
    expr = parenthesized expression of opnd opr
}
push expr into stack

```

4. repeat from 1 till string does not end.
5. Pop from stack and display.

Analysis

Prefix string * + A B – C D

Symb	Stack	Expr
*	*	
+	*+	
A	*+ A	A
B	*,(A+B)	(A+B)
-	*, (A+B), -	
C	*,(A+B) , -, C	C
D	*, (A+B)	(C-D)
	((A+B) * (C-D))	((A+B) * (C-D))

Conversion from Postfix to Infix

Example

Postfix $AB + C *$

Infix $((A+B) * C)$

We need:

1. A postfix string
2. A stack to store operands and infix expressions.

Algorithm

1. Symb is next input character
2. If symb is an operand
push symb in stack
3. If symb is not operand
opnd 1 = pop ();
opnd 2 = pop ();
expr = (opnd 2 symb opnd 1)
Push expr in stack
4. Continue from 1 till string does not end
5. Pop from stack and display.

Analysis

Postfix string = $AB + CD - *$

Symb	Stack	Expr
A	A	
B	*+	
+	*+ A	A
C	*, (A+B)	(A+B)
D	*, (A+B), -	
-	*, (A+B), -, C	C
*	*, (A+B)	(C-D)
	((A+B) * (C-D))	((A+B) * (C-D))

Evaluating a Prefix Expression

We need:

1. A prefix string (pre)
2. A stack to store operators and operands(str)

Algorithms

1. stk is the empty stack.
2. Read a character from the prefix string (symb)
3. If symb is an operator
push it into stk
4. If symb is an operand
if top of stk contains an operand
opnd1 = pop ()
opr = pop ()
Apply opr to opnd1 and symb Push result into stk.
else
push symb in stk
5. Repeat from 2 until prefix does not end.
6. If stk is not empty
opnd2 = pop ()
opr = pop ()
opnd1 = pop ()
result = apply opr to opnd1 and opnd2.
Push result into stk
7. Pop from stk and display
8. Stop

5.4 Matching Parentheses in an expression

An expression containing parentheses will be considered invalid if the parentheses are not balanced. A Stack can be used to check if they are balanced or NOT.

Example

[(a+b) *C] Balanced
{ [(a+b) *C Imbalanced

Algorithm

1. Read a character *ch* from input expression.
2. If it is an opening bracket, push it into the stack.
3. If it is a closing bracket,
if(stack is not empty)
 { *ch1* = pop()
 If *ch1* is not the closing bracket of *ch*
 { display - parentheses not balanced
 goto 6
 } }
else
display parentheses not balanced, goto 6
4. repeat from 1 till input expression ends
5. If stack is not empty
display -parentheses not balanced
display - valid expression
6. Stop.

6. Queue

A **queue** is another important data structure which finds applications in numerous situations. In this chapter, we will study the concepts of a queue, the operations and applications of this data structure.

We are all familiar with the concept of a queue in day-to-day situations like queues at a bus stand, booking office etc. With reference to computers, queues are used for resource scheduling.

7. Definition of a Queue

A **queue** is an ordered collection of items from which items may be deleted from (or removed from) one end called the **front** and into which items may be inserted at the other end called **rear**.

The elements are added and deleted in a **FIFO (First In First Out)** manner.

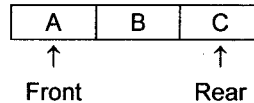
1

Apr.2011 – 2M

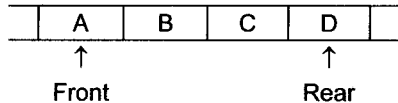
Explain Queue with example.

Example

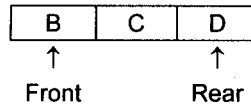
The following diagram shows a queue with three elements A,B and C. A is at the front and C is at the rear.



A new element, D will be added at the rear.



An element can be deleted only from the front. Thus, A will be the first to be removed from the queue.



8. Operations on a Queue

The operations on a queue are on similar lines to those on a stack.

2

Oct.2015 – 4M

What is queue? Explain its operations in detail.

Apr.2012 – 2M

What is Queue? State various operations performed on a Queue.

1. **Create:** Creates a new queue. This operation creates an empty queue.
2. **Add or Insert:** Adds an element to the queue. A new element can be added to the queue at the rear.
3. **Delete:** Removes an element from the queue. This operation removes the element, which is at the front of the queue. This operation can only be performed if the queue is not empty.

The result of an illegal attempt to remove an element from an empty queue is called **underflow**.

4. **Isempty:** Checks whether a queue is empty. This operation returns TRUE if the queue is empty and FALSE otherwise.

9. Implementation of Queues

Queues can be implemented in two ways:

1. Static using Arrays
2. Dynamic using linked representation

9.1 Static Implementation of Linear Queues

Logically, there is no limitation on the number of elements in a queue. However, since we are using an array to implement it, there will be an upper limit on its size. Thus, we will have to check for a **queuefull** condition before adding any element to the queue.

Declaration

A queue to be implemented using an array will require,

1. An array to store the elements.
2. An integer called front which stores the position of the first (or oldest) element of the queue.
3. An integer called rear which will store the position of the last (or newest) element of the queue.

The declarations will be

```
#define MAX 10
#define FULL MAX-1
#define EMPTY -1
struct queue
{ int front, rear;
  int items[MAX];
};
struct queue q1 ;      /* q1 is the actual queue */
```

We can also use typedef to create queue as a new data type name.

```
typedef struct
{ int front, rear;
  int items [MAX];
} Q;
Q q1, q2;              /* q1 and q2 are two queues */
```

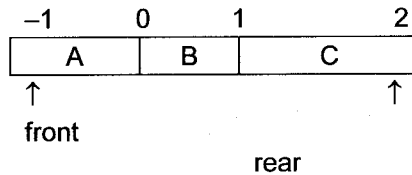
Operations on the Queue

- 1. Creating an empty queue:** Once we declare a variable of type Q, we have to initialize the front and rear values to indicate an empty queue. This can be done by initializing them to -1.

```
void initqueue(Q *pq)
{ pq->front = pq->rear = EMPTY;
}
```

The function will be called as
`initqueue(&q1);`

- 2. Checking for an empty queue:** A queue will be empty when both, front and rear are at the same position. For example, if the queue contains A, B, and C, the positions of front and rear will be



After removing the elements A, B and C, front and rear will coincide. Thus, the queue empty condition can be written as

```
int isqempty(Q *pq)
{
    return(pq->front == pq->rear);
}
```

- 3. Checking for a full queue:** As we insert elements into the queue, rear gets incremented. At some stage it will reach the array limit i.e., MAX-1 after which no more elements can be added. Thus, the function can be written as

```
int isfull(Q *pq)
{ return(pq->rear == FULL);
}
```

- 4. Adding an element to the queue:** A new element is added at the rear end of the queue. Since, we have initialized rear to -1 the first element should be put at position 0. Thus, rear should be first incremented and the element should be stored at that position.

```
void addq(Q *pq, int num)
{ if(!isfull(pq))
    pq->items[++(pq->rear)] = num;
  else
    printf("\n Sorry, queue is full \n");
}
```

Calling the function: `addq(&q1,n);`

5. **Deleting an element from the queue:** An element has to be deleted from the front. As is the case for rear, front has to be first incremented (since it has been initialized to -1) and then the element at that position has to be returned.

```
int delq(Q *pq)
{
    int n
    n = pq → items[+(pq → front)];
    return n;
}
```

Before this function is called, the queue empty condition has to be checked for. Calling the function.

```
if (isqempty (&q1) == 1)
    printf("Queue empty");
else
    n = delq(&q1);
```

A simple menu driven program to implement a queue using the declaration and functions discussed above is as follows.

/ Illustrates operations on a Linear Queue */*

/ Declarations and functions to be written here */*

```
main()
{
    Q q1;           /* Local declaration */
    int choice;
    initqueue (&q1);
    do
    { printf("\n1: ADD");
      printf("\n2: DELETE");
      printf("\n3: EXIT");
      printf("\n\n Enter your option :");
      scanf("%d", &choice);
      switch(choice)
      {
          case 1 : printf("\n Enter the data to be added");
                    scanf("%d", &n);
                    addq (&q1, n);
                    break;
          case 2 : if (isqempty (&q1))
                      printf("\n Queue is empty !");
                    else
```

```

        printf("\n %d is removed ", delq(&q1));
        break;
    case 3 : printf("\n Exiting ....");
    }
} while(choice !=3);

```

Output

```

1 : ADD
2 : DELETE
3 : EXIT
Enter your option: 1
Enter the data to be added 20
1 : ADD
2 : DELETE
3 : EXIT
Enter your option: 2
20 is removed.

1 : ADD
2 : DELETE
3 : EXIT
Enter your option: 2
Queue is empty!

1 : ADD
2 : DELETE
3 : EXIT
Enter your option: 3
Exiting.....

```

9.2 Static Implementation of Circular Queue

Just as we implement a stack using a linked list, we can implement a queue in a similar manner.

A queue can be considered as a list in which all insertions are made at one end called the rear and all deletions from the other end called front.

A queue can be easily represented using a linked list.

The front and rear will be two pointers pointing to the first and last node respectively.

```

struct node
{ struct node*next;
  int info;
} * front, *rear;

```

The various operations are illustrated below.

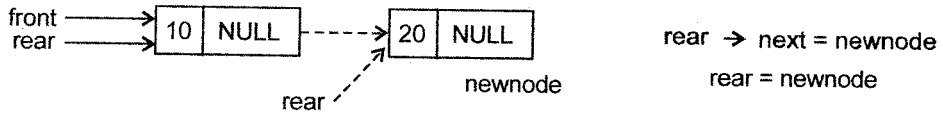
1. Initqueue

front = rear = NULL

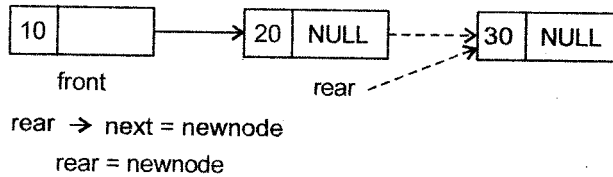
2. Add 10



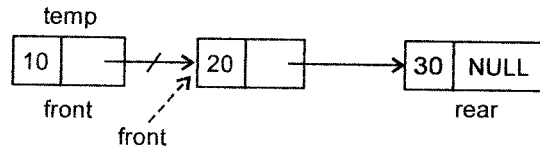
3. Add 20



4. Add 30



5. Delete

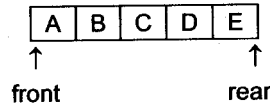


```
temp = front;
front = front  $\rightarrow$  next;
free(temp);
```

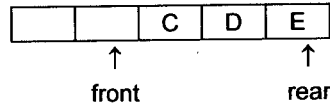
Disadvantage of a Linear Queue

We have seen the queue operations earlier. Let us consider the following scenario for a queue with maximum size 5.

Initially, the queue is empty i.e. front and rear are at -1 . After the elements A, B, C, D, and E are added, the queue will look like



If we remove two elements from the queue, we have,



We have two vacant positions at the beginning of the queue but we cannot add new elements there since rear has reached $\text{MAX}-1$ which is the queue full condition.

Thus, even if elements are removed from the queue, new elements cannot take their positions.

Notion of a Circular Queue

In order to reuse the vacant positions, we will have to bring rear to the 0^{th} position if it is empty.

i.e.,

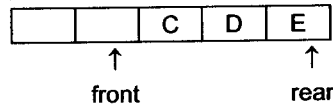
```
if(rear == MAX-1)
    rear = 0;
else
    rear = rear+1;
```

This can also be written as $\text{rear} = (\text{rear}+1) \% \text{MAX}$;

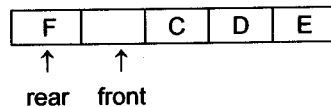
The queue is logically treated in a circular manner.

Example

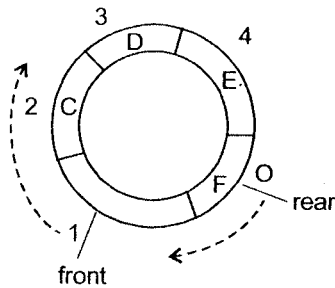
Assuming that the queue contains three elements as in the previous example i.e.



Now, we can insert an elements F at the beginning by bringing rear to the first position in the queue.



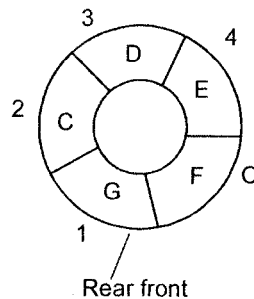
This can be represented circularly as shown.



Circular Queue

A peculiar situation arises when the queue is full.

In the above example, if another element, G is added to the queue, it will look like



Full Circular Queue

i.e. rear and front coincide. But rear and front coincide even when the queue is empty! Hence there is an ambiguity.

$\text{rear} == \text{front}$ cannot be used for both i.e. to check for an empty queue as well as the condition for a full queue. Hence some method of resolving this problem is needed.

We shall keep the $\text{rear} == \text{front}$ condition as a check for the empty queue since initially both are initialized to the same value. Thus, we need another method to check for a full queue.

There are three possible solutions

1. Use a counter to keep track of the number of elements in the queue. If this counter reaches MAX, the queue is full.

2. Allow only MAX-1 values to be put in the queue i.e. use one less element of queue. Thus the queue full condition will occur when

$$(\text{rear} + 1) \% \text{MAX} == \text{front}$$

3. The values of front and rear could be set to some values that are not valid array indices to indicate an empty condition.

i.e. if rear becomes equal to front after a remove operation it indicates queue empty and hence they could be reset to -1 and $(\text{rear} == -1) \ \&\& \ (\text{front} == -1)$ will be the conditions for queue empty.

If rear becomes equal to front after an add operation, it implies the queue is full. Thus $\text{rear} != -1 \ \&\& \ \text{front} != -1$ will be the condition for queue full.

Functions for Circular Queue

1. Initialize Q

```
void initqueue(Q *cq)
{
    cq → rear = cq → front = 0;
}
```

2. Is empty

```
int isemptycq(Q *cq)
{
    return(cq → rear == cq → front );
}
```

3. Is full

```
int isfullcq(Q*cq)
{
    return((rear +1)% MAX == front);
}
```

Here we have used the method (2) to solve the conflict between empty and full condition.

4. Add

```
void addcq(Q *cq, int num)
{
    cq → items[cq → rear] = num;
    cq → rear=(cq → rear +1) % MAX; /*Increment rear circularly */
}
```


5. Delete

```
int delcq(Q *cq)
{ n= cq → items [cq → front]};
cq → front= (cq → front+1)% MAX; /* Increment front circularly */
return n.
}
```

The following program illustrates a circular queue operations for an array of size MAX=3.

```
/* Illustrates circular queue operation */
/* Declarations and Functions here */
main()
{ Q cq1;
  int num,choice;
  initqueue(&cq1);
  do
  {
    printf("\n 1:ADD");
    printf("2 : DELETE");
    printf("3:EXIT");
    printf("\n Enter the option :");
    scanf("%d", &choice);
    switch(choice)
    { case 1 : printf("\n Enter the number :");
      scanf("%d",&num );
      if(isfullcq (&cq1))
        printf("\n Queue is full \n");
      else
        addcq(&cq1,num);
      break;
      case 2 : if(isemptycq(&cq1))
        printf("\n Empty Queue !");
      else
        printf("%d is deleted " delcq(&cq1));
      case 3 : printf("\n Exiting ....");
    }
  }while(choice != 3);
```

Output

```
1 : ADD2 : DELETE3: EXIT
Enter the option : 1
Enter the number : 10

: ADD2 : DELETE3: EXIT
Enter the option : 1
Enter the number : 20
```

```

1 : ADD2 : DELETE3 : EXIT
Enter the option : 1
Enter the number : 30

1 : ADD2 : DELETE3 : EXIT
Enter the option : 2
10 is deleted

1 : ADD2 : DELETE3 : EXIT
Enter the option : 1
Enter the number : 40

1 : ADD2 : DELETE3 : EXIT
Enter the option : 3
Exiting.....

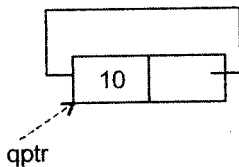
```

9.3 Dynamic Implementation of Circular Queue

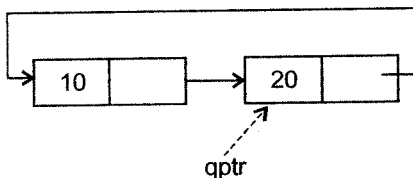
In a circular queue, implemented as a test, the last element will point to the first. Since it is a circular list, only one pointer to the list is needed to perform all operations on the list.

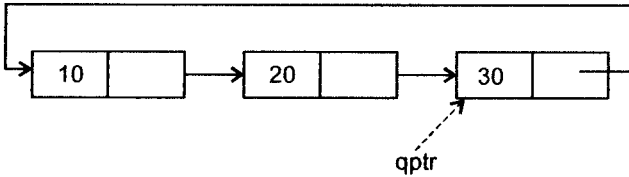
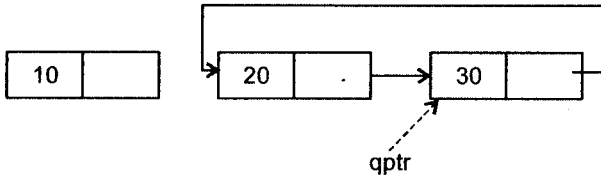
Operations on Circular Queue

1. `qptr = NULL;`
2. Add 10



3. Add 20



4. Add 30**5. Delete**

These operations can be performed as follows

1. Initialize Queues

```
void initq()
{
    qptr = NULL;
}
```

2. Add

```
void addcq(int n)
{
    struct node * newnode;
    newnode = (struct node *) malloc(sizeof (struct node));
    newnode -> element = n;
    if(qptr == NULL)
    {
        qptr = newnode;
        qptr -> next = qptr;
    }
    else
    {
        newnode -> next = qptr -> next;
        qptr -> next = newnode;
        qptr = newnode;
    }
}
```

3. Queue empty

```
int emptycq()
{
    return(qptr == NULL);
}
```

4. Delete

```
int deletecq()
{
    int n;
    struct node * temp;
    temp = qptr → next ; n = temp → element;
    if(temp == qptr) /* only one node*/
    {
        free(temp);
        qptr = NULL;
        return n ;
    }
    qptr → next = temp → next;
    free(temp);S
    return n;
}
```

10. Types of Queues

1

Oct.2011 – 2M

State different types of Queue.

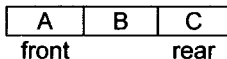
A queue can be of two types:

- i. Linear Queue
- ii. Circular Queue

10.1 Linear Queue

In this queue, the elements are arranged in a sequential manner such that front position is always less than or equal to the rear position. When an element is added, rear is incremented and when an element is removed, front is advanced. Thus, front always follows rear.

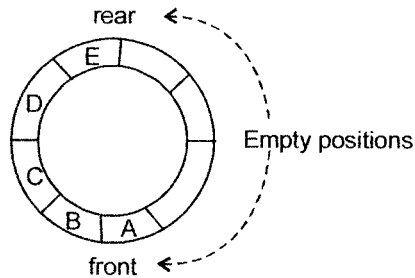
Example



10.2 Circular Queue

In this queue, the elements are arranged in a sequential manner but can logically be regarded as circularly arranged. The rear and front move in a clockwise direction.

Example



Circular Queue

3

Oct.2012 – 2M

What is Circular queue?
How it is represented.

Apr.2012 – 2M

Define: Circular Queue

Oct.2009 – 2M

What is Circular Queue?

11. Priority Queue

In a queue, the elements are inserted at the rear end and deleted from the front. Hence the FIFO ordering is maintained and all insertions and deletions are done in a strict sequence. Even though the element themselves may have some inherent order among themselves it is ignored.

However, in some cases, this strict order needs to be violated and the intrinsic ordering among elements will determine which element gets removed first.

Definition

A priority queue is a data structure in which the intrinsic ordering among the elements decides the result of its basic operations.

Example

Scheduling of jobs by the operating system.

4

Apr.2012 – 4M

What is Priority Queue?
Explain in short.

Apr.11, 10, Oct.10 – 2M

What is Priority Queue?

Here the operating system assigns priorities to each type of job. The jobs are placed in a queue and the job with the highest priority will be executed first.

Types of Priority Queues

1. **Ascending Priority Queue:** Elements can be inserted arbitrarily but only the smallest element can be removed.
2. **Descending Priority Queue:** This allows deletion only of the largest element.

Elements of the Priority Queue

The elements need not be numbers or characters but they may be complex structure that are ordered on several fields like Telephone directory listing comprising of last name, first name, address.

The basis of ordering need not be a part of the element. It could be an external value on the basis of which ordering is done *for example* - time.

Thus a queue can be viewed as an ascending priority queue whose elements are ordered by time of insertion. The element having the smallest time value comes out first.

List Implementation of Priority Queue

A Priority Queue can be implemented dynamically in two ways.

1. Maintaining an ordered linked list of elements
2. Maintaining unordered linked list of elements

Let us consider implementation of an ascending priority queues.

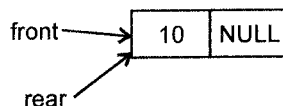
- i. **Ordered List Implementation:** In this method, the elements are added to the list in such a way that the list is in the sorted order.

Thus, when an element is to be added to the priority queue, it is inserted in its correct position in the queue.

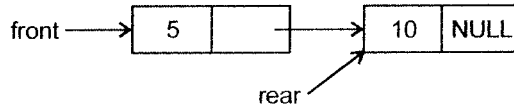
For deletion, the first element (which is the smallest element) is removed.

Example

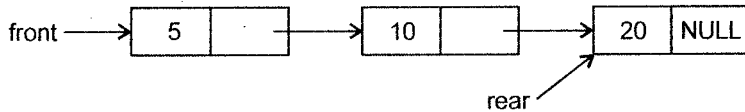
1. front = rear = NULL
2. Add 10



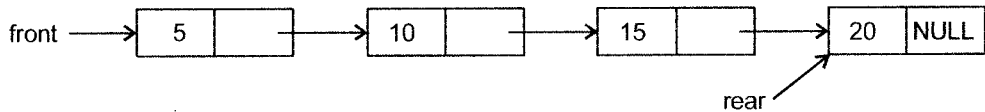
3. Add 5



4. Add 20



5. Add 15



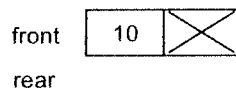
ii. **Priority Queue as unordered list:** In this method of implementation, elements are added to the rear of the queue. Thus, insertion is a simple process.

However for deletion, the entire list has to be examined in order to find the smallest element, which is then removed from the list.

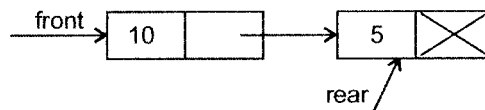
Example

1. front = rear = NULL

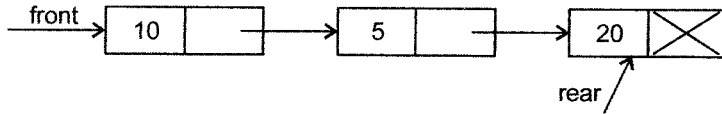
2. Add 10



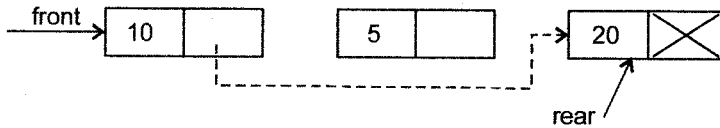
3. Add 5



4. Add 20



5. Delete



12. Doubly-Ended Queue (DEQUE)

A DEQUE is an ordered set of items into which items may be inserted and deleted from either end. If the two ends of the queue are called left and right, then the four operations on the deque will be

1. addleft or insertleft
2. addright or insertright
3. deleteleft
4. deleteright

A DEQUE can be of two types:

1. **Input-Restricted Deque (IRD):** This is a deque which allows insertions only at one end but allows deletions at both ends of the deque.

Valid operations

addleft, deleteleft, deleteright

2. **Output-Restricted Deque (ORD):** This is a deque which allows deletions at only one end of the deque but allows insertions at both ends.

Valid operations

addleft, addright, deleteleft

Example

Let us consider a deque, IRD and ORD implemented using an array. Left and right represent the two ends of the queue.

Operation	Deque	IRD	ORD																								
1. Addleft (A)	<table border="1"><tr><td>A</td><td></td><td></td><td></td></tr><tr><td colspan="2">Left</td><td colspan="2">Right</td></tr></table>	A				Left		Right		<table border="1"><tr><td>A</td><td></td><td></td><td></td></tr><tr><td colspan="2">Left</td><td colspan="2">Right</td></tr></table>	A				Left		Right		<table border="1"><tr><td>A</td><td></td><td></td><td></td></tr><tr><td colspan="2">Left</td><td colspan="2">Right</td></tr></table>	A				Left		Right	
A																											
Left		Right																									
A																											
Left		Right																									
A																											
Left		Right																									
2. Addright(B)	<table border="1"><tr><td>A</td><td>B</td><td></td><td></td></tr><tr><td>L</td><td>R</td><td></td><td></td></tr></table>	A	B			L	R			Invalid	<table border="1"><tr><td>A</td><td>B</td><td></td><td></td></tr><tr><td>L</td><td>R</td><td></td><td></td></tr></table>	A	B			L	R										
A	B																										
L	R																										
A	B																										
L	R																										
3. Addleft (C)	<table border="1"><tr><td>C</td><td>A</td><td>B</td><td></td></tr><tr><td>L</td><td></td><td>R</td><td></td></tr></table>	C	A	B		L		R		<table border="1"><tr><td>C</td><td>A</td><td></td><td></td></tr><tr><td>L</td><td>R</td><td></td><td></td></tr></table>	C	A			L	R			<table border="1"><tr><td>C</td><td>A</td><td>B</td><td></td></tr><tr><td>L</td><td></td><td>R</td><td></td></tr></table>	C	A	B		L		R	
C	A	B																									
L		R																									
C	A																										
L	R																										
C	A	B																									
L		R																									
4. Deleteright()	<table border="1"><tr><td>C</td><td>A</td><td></td><td></td></tr><tr><td>L</td><td>R</td><td></td><td></td></tr></table>	C	A			L	R			<table border="1"><tr><td>C</td><td></td><td></td><td></td></tr><tr><td>L</td><td></td><td>R</td><td></td></tr></table>	C				L		R		Invalid								
C	A																										
L	R																										
C																											
L		R																									
5. Deleteleft	<table border="1"><tr><td></td><td>A</td><td></td><td></td></tr><tr><td></td><td>L</td><td></td><td>R</td></tr></table>		A				L		R		<table border="1"><tr><td></td><td>A</td><td>B</td><td></td></tr><tr><td></td><td>L</td><td>R</td><td></td></tr></table>		A	B			L	R									
	A																										
	L		R																								
	A	B																									
	L	R																									

13. Applications of Queues

- Queues are used in computers for scheduling of resources to applications. These resources are CPU, printer, etc.
- Multiple print jobs given to a printer are organized in the FIFO manner in a print queue and given to the printer.
- In batch programming, multiple jobs are combined into a batch and the first program is executed first, the second is executed next and so on in a sequential manner.

4

Oct.14, Apr.15,10 – 2M

What are the Applications of Queue?

Oct.2009 – 4M

What is Queue? What are the various Applications of Queue?

13.1 CPU Scheduling Algorithms

In a multiprogramming environment, multiple processes are to be handled by the CPU. These processes can be

- Interactive programs
- Batch jobs
- System tasks

All these processes need the CPU. Hence, they should be scheduled properly to increase the CPU utilization and throughput. The scheduling is done by using queues.

The various methods used for scheduling the processes are

1. FCFS – First Come First Served Method
2. Round Robin Method

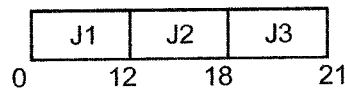
First Come First Served (FCFS)

- New jobs or processes are added to the queue at the end.
- The CPU executes the processes at the front of the queue.
- After the process is executed, it switches to the next process.
- Once the CPU is allotted to a process, it is released when the process is completed.

The average turn around time is an important criterion to determine the performance. Turn around time is the time interval between submission of a job till its completion.

Example

Job	Burst Time (Job time)
1	12
2	6
3	3



$$\therefore \text{Average turn-around time} = \frac{12 + 18 + 21}{3} = 17$$

Advantages

1. It is very simple method of scheduling.
2. A single queue can be used to implement FCFS.

Disadvantages

1. A smaller job or a high priority job may have to wait for a long time.
2. If the job at the front is an I/O job, it will utilize the CPU for a long time.

Round Robin Algorithm

This is a very popular method and is mainly used in time-sharing systems. Instead of waiting for the current job to be completed before taking the next job, each job in the queue is assigned a small unit of time called *time slice* or *time quantum*.

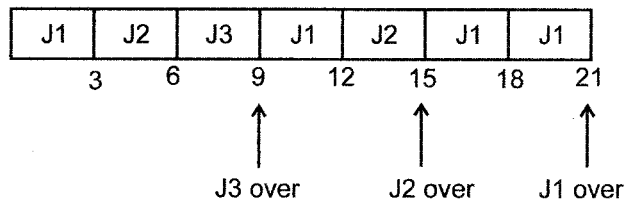
The CPU selects the first job from the queue, executes it for one time slice. After the time slice is over, it switches over to the next job in the queue. The interrupted job is added back to the end of the queue.

If the job completes execution before the time slice, it released the CPU.

Round Robin algorithm is implemented using circular queue.

Job	Burst Time
1	12
2	6
3	3

If a time slice = 3 units, the Round Robin Algorithm will work as shown.



$$\text{Average turn around time} = \frac{9 + 15 + 21}{3} = 15$$

Advantages

1. All jobs get a fair share of the CPU.
2. Priorities can be assigned to the jobs so that a higher priority job does not have to wait for a long time for its time slice.

Disadvantage: Implementing the Round Robin method is more complex.

Solved Examples

1

Oct.2012 – 4M

1. Convert the following Infix Expression into Prefix Expression:

i. $(A + B)/C \times D - E$

ii. $A - (B + C/D) \wedge E$

Solution

The steps to convert any infix expression into prefix are:

- i. Reverse the infix expression.
- ii. Make every '(' (opening bracket) as ')' (closing bracket) and every ')' as '('.
- iii. Convert the modified expression to postfix form.
- iv. Reverse the postfix expression.

i. $(A + B)/C * D - E$

a. Step 1: Reverse the infix expression

$$E - D * C / B + A ($$

b. Step 2: Make every '(' as ')' and every ')' as '('

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

c. Step 3: Convert the expression to postfix form:

Steps	Expression	Current Symbol	Stack	Output	Comments
1.	$E - D * C / (B + A)$	Empty	Empty	-	Initially
2.	$- D * C / (B + A)$	E	Empty	E	Print the operand/ Attach the operand at the end of postfix expression
3.	$D * C / (B + A)$	-	-	E	Push the current operator onto the top of the stack
4.	$* C / (B + A)$	D	-	ED	Attach the operand at the end of postfix expression
5.	$C / (B + A)$	*	- *	ED	Push the higher precedence operator onto the top of the stack
6.	$/ (B + A)$	C	- *	EDC	Attach the operand at the end of postfix expression
7.	$(B + A)$	/	- /	EDC*	Same priority operators so pop one and at end of postfix.
8.	$B + A)$	(- / (EDC*	As opening bracket so push it onto stack
9.	$+ A)$	B	- / (EDC*B	Attach the operand at the end of postfix expression
10.	$A)$	+	- / (+	EDC*B	Attach the operator onto stack
11.	$)$	A	- / (+	EDC*BA	Attach the operand at the end of postfix expression

12.	Empty)	- /	EDC*BA +	Pop all operators until (from stack and add at end of postfix
13.	Empty	Empty	empty	EDC*BA+/ -	Pop all operators until empty stack and add at end of postfix

Reverse the resultant expression, we get the prefix as “- / + AB * CDE”

ii. $A - (B + C/D) \wedge E$

a. Step 1: Reverse the infix expression

$$E \wedge D / C + B (- A$$

b. Step 2: Make every ‘(‘ as ‘)’ and every ‘)’ as ‘(‘

$$E \wedge (D / C + B) - A$$

c. Step 3: Convert the expression to postfix form:

Steps	Expression	Current Symbol	Stack	Output	Comments
1	$E \wedge (D / C + B) - A$	Empty	Empty	-	Initially
2	$\wedge (D / C + B) - A$	E	Empty	E	Print the operand/ Attach the operand at the end of postfix expression
3	$(D / C + B) - A$	\wedge	\wedge	E	Push the current operator onto the top of the stack
4	$D / C + B) - A$	(\wedge	E	As opening bracket so push it onto stack
5	$/ C + B) - A$	D	\wedge	ED	Attach the operand at the end of postfix expression
6	$C + B) - A$	/	/	ED \wedge	Higher precedence operator is popped and attach at the end of postfix expression
7	$+ B) - A$	C	/	ED \wedge C	Attach the operand at the end of postfix expression
8	$B) - A$	+	+	ED \wedge C/	Higher precedence operator is popped and attach at the end of postfix expression
9	$) - A$	B	+	ED \wedge C/B	Attach the operand at the end of postfix expression
10	$- A$)		ED \wedge C/B+	Pop all operators until (from stack and add at end of postfix
11	A	-	-	ED \wedge C/B+	Push the current operator onto the top of the stack
12	Empty	A	-	ED \wedge C/B+A	Attach the operand at the end of postfix expression
13	Empty	Empty	empty	ED \wedge C/B+A-	Pop all operators until empty stack and add at end of postfix

Reverse the resultant expression, we get the prefix as “- A+B/C \wedge DE”

1

Apr.2012 – 4M

2. Evaluate the following postfix expression using a Stack:
 $AB/CD * + A = 6, B = 2, C = 3, D = 4$

Solution

Step No.	Input	Stack	Operand 1	Operand 2	Value	Comment
1.	AB/CD*+	Empty	-	-	-	Initially
2.	B/CD*+	6	-	-	-	Push the operand onto the top of the stack
3.	/CD*+	2 6	-	-	-	Push the operand onto the top of the stack
4.	CD*+	3	6	2	6	Perform the division operation between 6 and 2
5.	D*+	3 3	-	-	-	Push the operand onto the top of the stack
6.	*+	4 3 3	-	-	-	Push the operand onto the top of the stack
7.	+	12 3	3	4	12	Perform the multiplication operation between 3 and 4
8.	empty	empty	3	12	15	Perform the addition operation between 3 and 12

1

Oct. 2011 – 4M

3. Convert the following infix expressions into postfix.
 i. $(A + B) * C$ ii. $(A+B) *(C+D)$

*Solution*The given expression is $(A+B)*C$

Steps	Expression (Input)	Current Symbol	Stack	Output (postfix Exp)	Comments
1	$(A+B)*C$	Empty	Empty	-	Initially
2	$A+B)*C$	((-	Push the opening bracket onto the top of the stack
3	$+B)*C$	A	(A	Print the operand/ attach the operand at the end of postfix expression
4	$B)*C$	+	(+	A	Push the current operator onto the top of the stack
5	$))*C$	B	(+	AB	Attach the current operand at the end of postfix expression
6	$*C$)	-	AB+	Pop the top operator from stack until '(' and add that operator at the end of postfix expression
7	C	*	*	AB+	Push the current operator onto the top of the stack
8	Empty	C	*	AB+C	Attach the current operand at the end of postfix expression
9	Empty	-	Empty	AB+C*	Pop all operators from stack and add at the end of postfix expression

The Postfix Expression is $AB+C*$

ii. $(A+B)*(C+D)$ The given expression is $(A+B)*(C+D)$

Steps	Expression (Input)	Current Symbol	Stack	Output (postfix Exp)	Comments
1	$(A+B)*(C+D)$	Empty	Empty	-	Initially
2	$A+B)*(C+D)$	((-	Push the opening bracket onto the top of the stack
3	$+B)*(C+D)$	A	(A	Print the operand/ attach the operand at the end of postfix expression
4	$B)*(C+D)$	+	(+	A	Push the current operator onto the top of the stack
5	$)*(C+D)$	B	(+	AB	Attach the current operand at the end of postfix expression
6	$*(C+D)$)	-	AB+	Pop the top operator from stack until '(' and add that operator at the end of postfix expression
7	$(C+D)$	*	*	AB+	Push the current operator onto the top of the stack
8	$C+D)$	(*($($	AB+	Push the opening bracket onto the top of the stack
9	$+D)$	C	*($($	AB+C	Attach the current operand at the end of postfix expression
10	$D)$	+	*($(+$	AB+C	Push the current operator onto the top of the stack
11	$)$	D	*($(+$	AB+CD	Attach the current operand at the end of postfix expression
12	Empty)	*	AB+CD+	Pop the top operator from stack until '(' and add that operator at the end of postfix expression
13	Empty	-	-	AB+CD+*	Pop all operators from stack and add that operator at the end of postfix expression

The postfix expression is $AB+CD+*$

4. Write a function to accept size of stack and insert element till it is full.

Apr.2011 – 4M

Solution

Function to accept size of stack and insert element till it is full

```
void push()
{
    int item, maxsize, top;
    printf("\n enter stack size:-");
    scanf("%d", &maxsize);
    if(top==maxsize-1)
```

```

{
    printf("\n stack is full");
    getch();
    exit(0);
}
else
{
    printf("\n enter the element to be inserted:-");
    scanf("%d", &item);
    top = top + 1;
    stack[top]=item;
}
}

```

2

Apr.11, Oct.10 - 4M

5. Convert the following infix expression into prefix:

- i. $A + B * C/D$ ii. $A + B + C + D$

Solution

- i. $A + B * C/D$
 $= A + B * /CD$
 $= A + * B /CD$
 $= + A * B /CD$
 $+ A * B /CD$
- ii. $A + B + C + D$
 $= + AB + C + D$
 $= ++ ABC + D$
 $= +++ ABCD$
 $+++ABCD$

2

Oct.10, Apr.10 - 4M

6. Write a 'C' program to reverse given string using stack.

Solution

```

#include<stdio.h>
#include<conio.h>
void main()
{ char str[80];
  int i,j,k,swap;
  clrscr();
  printf("\n Enter the String");
  gets(str);
  for(i=1;i<=strlen(str); i++)
  { for(j=0;j<1;j++)
    {

```



```

    if (str[i] >= str[j])
    { swap = str[j];
      str[j] = str[i];
    }
  }
}
printf("\n The String is in Reverse order");
puts(str);
getch();
}

```

7. Convert the following infix expression into postfix expression: $A * (B + C \wedge D) - E / F * (G + D)$

Apr.2010 - 4M

1

Solution

Convert Infix Expression into Postfix Expression

$A B + C \wedge D * / E - F * G D +$
 $A B \wedge D * / + E - F G D * +$
 $A \wedge B D * / + - E F G D * - +$
 $A \wedge B D * / + - E F G D * + -$

8. Convert following infix expressions into postfix.

i. $A + B * C$ ii. $A + B + C$

Oct.2009 - 4M

1

Solution

i. $A + B * C$

(Precedence of * is higher than of +)

Step 1: $A + (B * C)$ convert the multiplication

Step 2: $A + (BC *)$ convert the addition

Step 3: $A (BC *) +$ Remove parentheses

Step 4: $ABC * +$ Postfix

ii. $A + B + C$

(Precedence of + is higher than of *)

Step 1: $A + (B + C)$ convert into the Parentheses

Step 2: $A + (BC +)$ convert the Multiplication

Step 3: $A (BC +)$ Remove parentheses

Step 4: $ABC ++$ Postfix

9. Write a program to accept size of stack and add elements onto the stack one by one which are accepted from user until stack is full.

Oct.2009 - 4M

1

Solution

```

#include<stdio.h>
#include<conio.h>

```

```
void main()
{
int top, s[10];, item, choice;
clrscr();
top = -1;
for(;;)
{
printf("\n 1. Insert ");
printf("\n 2 Display");
printf("\n Enter your Choice ");
scanf("%d", & choice);
switch(choice)
{
case 1:
printf("\n Enter the Item to be Inserted ");
scanf("%d", &item);
push(item, &top, s);
break;
case 2:
display(top, s);
break;
}
}
getch();
}
void push(int item, int *top, int s[])
{
if (*top == STACK_SIZE -1)
{
printf("\n Stack Overflow\n");
return;
}
s[++(*top)] = item;
}
void display( int top, int s[])
{
int i;
if(top == -1)
{
printf("Stack is Empty\n");
return;
}
printf("\n The Contents of the Stack \n");
for (i=0;i<=top;i++)
{
printf("%d\n", s[i]);
}
}
}
```

1

Oct.2012 - 2M

10. Write a 'C' Program for Dynamic Implementation of Queue.**Solution**

A Program for Dynamic Implementation of Queue, i.e., implementation of Queue using linked list is as:

```
#include<stdio.h>
#include<malloc.h>
struct node // structure to represent Queue
node
{
    int info;
    struct node *link;
}*front=NULL,*rear=NULL;

main()
{
    int choice;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                del();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
            default:
                printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
    }/*End of main()*/
insert()
{
    struct node *tmp;
    int added_item;
```

```
tmp = (struct node *)malloc(sizeof(struct node));
printf("Input the element for adding in queue : ");
scanf("%d",&added_item);
tmp->info = added_item;
tmp->link=NULL;
if(front==NULL) /*If Queue is empty*/
    front=tmp;
else
    rear->link=tmp;
    rear=tmp;
}/*End of insert()*/

del()
{
    struct node *tmp;
    if(front == NULL)
        printf("Queue Underflow\n");
    else
    {
        tmp=front;
        printf("Deleted element is %d\n",tmp->info);
        front=front->link;
        free(tmp);
    }
}/*End of del()*/

display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
        printf("Queue is empty\n");
    else
    {
        printf("Queue elements :\n");
        while(ptr != NULL)
        {
            printf("%d ",ptr->info);
            ptr = ptr->link;
        }
        printf("\n");
    }/*End of else*/
}/*End of display()*/
```

11. Write a C program for implementation of dynamic queue.**Solution**

Oct.2014 - 4M

1

```
#include<stdio.h>
#include<malloc.h>
typedef struct node          // structure to represent Queue
node
{
    int info;
}
*front=NULL,*rear=NULL;
main()
{
    int choice;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                del();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
            default:
                printf("Wrong choice\n");
        }
        /*End of switch*/
    }
    /*End of while*/
}
/*End of main()*/
insert()
{
    struct node *tmp;
    int added_item;
    tmp = (struct node *)malloc(sizeof(struct node));
    printf("Input the element for adding in queue : ");
    scanf("%d",&added_item);
    tmp->info = added_item;
```

```

tmp->link=NULL;
if(front==NULL) /*If Queue is empty*/
    front=tmp;
else
    rear->link=tmp;
    rear=tmp;
}/*End of insert()*/
del()
{
    struct node *tmp;
    if(front == NULL)
        printf("Queue Underflow\n");
    else
    {
        tmp=front;
        printf("Deleted element is %d\n",tmp->info);
        front=front->link;
        free(tmp);
    }
}/*End of del()*/
display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
        printf("Queue is empty\n");
    else
    {
        printf("Queue elements :\n");
        while(ptr != NULL)
        {
            printf("%d ",ptr->info);
            ptr = ptr->link;
        }
        printf("\n");
    }/*End of else*/
}/*End of display()*/

```

12. Write a 'C' program for implementation of circular queue.

Solution

```

#include<stdio.h>
#include<process.h>
#define QUEUE_SIZE 5
/* Function to check queue overflow */
int qfull(int count)
{

```

```
    retrun(count == QUEUE_SIZE) ? 1:0;
}
/* Function to check queue underflow */
int qempty(int count)
{
    return(count == 0) ? 1: 0;
}
/* Function to insert an item at the read end*?
void insert_rear(int item, int q[], int *r, int *count)
{
    if(qfull(*count))
    {
        printf("\n Overflow of Queue");
        return;
    }
    *r = (*r + 1) % QUEUE_SIZE;
    q[*r] = item;
    *count += 1;
}

/* Function to delete an item from the front end of queue*/
void delete_front(int q[], int *f, int *count)
{
    if(qempty(*count))
    {
        printf("Underflow of Queue");
        return;
    }
    printf("\n The deleted element is %d\n", q[*f]);
    *f= (*f+1) % QUEUE_SIZE;
    *count -= 1;
}

/* Function to display the contents of the queue */
void display(int q[], int f, int count)
{
    int i,j;
    if(qempty(count))
    {
        printf("\n Q is Empty ");
        return;
    }
    printf("\n Contents of Queue ");
    i=f;
    for(j=1;j<=count;j++)
```

```
{
    printf("\n%d", q[i]);
    i = (i+1) % QUEUE_SIZE;
}
}
void main()
{
    int choice,item,f,r,count,q[20];
    f = 0;
    r = -1;
    count = 0;
    for(;;)
    {
        printf("\n 1 for Insert");
        printf("\n 2 for Delete");
        printf("\n 3 for Display");
        printf("\n Enter the Choice");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("\n Enter the Item to be Inserted ");
                scanf("%d", &item);
                insert_rear(item, q, &r, &count);
                break;
            case 2:
                printf("\n Enter the Item to be Inserted ");
                scanf("%d", &item);
                delete_front(q, &f, &count);
                break;
            case 3:
                display(q, f, count);
                break;
            default:
                exit(0);
        }
    }
}
```

1

Oct.2014 - 4M

13. Write a function which compares the contents of two stacks and display message accordingly.

Solution

```
#include<stdio.h>
#include<conio.h>
void checkstack(stack *S1,stack *S2)
{
    while(!isempty(S1)&&!isempty(S2)
    {
        if(item[S1→top] == item[S2→top])
        {
            pop(S1);
            pop(S2);
        }
        else
        {
            printf("stack does not equal");
            return 0;
        }
    }
    if(isempty(S1)&& isempty(S2))
    {
        printf("both stacks are equal");
    }
    else
    printf("stack does not equal");
}
int isempty(stack *S)
{
    if(S→top== -1)
    return 1;
    return 0;
}
char pop(stack *S)
{
    char ch;
    ch = S→item[S→top];
    S→top--;
}return ch;
```



PU Questions

[Oct.15,10,Apr.11,10 – 2M]

[Oct.14,Apr.15,10 – 2M]

[Oct.2012,Apr.11 – 2M]

[Oct.2012 – 2M]

[Apr.2012 – 2M]

[Apr.2012 – 2M]

[Oct.2011 – 2M]

[Oct.2011 – 2M]

[Oct.2010 – 2M]

[Oct.2009 – 2M]

[Apr.2009 – 2M]

[Oct.2015 – 4M]

[Oct.2015 – 4M]

[Apr.2015 – 4M]

[Apr.2015 – 4M]

[Oct.2014 – 4M]

[Oct.2014 – 4M]

[Oct.2014 – 4M]

[Oct.14,10,Apr.10 – 4M]

[Oct.14.12 – 4M]

2 Marks

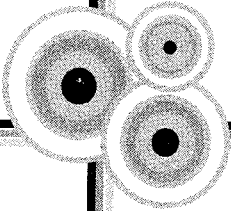
1. What is Priority queue?
2. What are the applications of queue?
3. What are the Application of Stack?
4. What is Circular queue? How it is represented.
5. What is Queue? State various operations performed on a Queue.
6. Define: Circular Queue
7. What is Recursion?
8. State different types of Queue.
9. What is a Stack? Discuss Operations Performed on Stack.
10. Explain different operations performed on a Stack.
11. What is Circular Queue?

4 Marks

1. Convert given infix expression to postfix expression:
 $(A - B) / C \wedge D * E + (F - G)$
2. What is queue? Explain its operations in detail.
3. Write a function which compares the contents of two queues and display message accordingly.
4. Write an algorithm to convert given infix expression to prefix expression.
5. What is stack? Explain its operations in details.
6. Write a C program for implementation of dynamic queue.
7. Write a function which compares the contents of two stacks and display message accordingly.
8. Write a "C" program for implementation of Circular queue.
9. Write an algorithm to convert Infix Expression to Postfix Expression.

10. Explain selection sort technique with an example. [Oct.2012 – 4M]
i. $A + B * C/D$ ii. $A + B + C + D$
11. Write a 'C' Program for Dynamic Implementation of Queue. [Oct.2012 – 4M]
12. Convert the following Infix Expression into Prefix Expression: i. $(A + B)/C \times D - E$ ii. $A - (B + C/D) \wedge E$ [Apr.2012 – 4M]
13. Write a 'C' program for Dynamic Implementation of stack. [Apr.2012 – 4M]
14. What is Priority Queue? Explain in short. [Apr.2012 – 4M]
15. Write a program to accept size of queue and add elements in the queue one by one from the user till the queue is filled. [Oct.2011 – 4M]
16. Evaluate the following postfix expression using a Stack: AB/CD
 $* +A = 6, B = 2, C = 3, D = 4$ [Oct.2011 – 4M]
17. Write an algorithm for prefix to infix conversion of an expression. [Oct.2011 – 4M]
18. What is stack? Discuss various applications of stack. [Oct.2011 – 4M]
19. Convert the following infix expressions into postfix. [Oct.2011 – 4M]
i. $(A + B) * C$ ii. $(A+B) *(C+D)$
20. Write a function for adding and deleting elements from stack. [Apr.2011 – 4M]
21. Explain algorithm to convert infix expression to its equivalent postfix expression. [Apr.11,Oct.10 – 4M]
22. Explain Dynamic Representation of Stack. [Apr.2011 – 4M]
23. Write a program to reverse a string using Stack. [Apr.2011 – 4M]
24. Explain Queue with example. [Apr.2011 – 4M]
25. Explain algorithm to convert infix expression to its equivalent postfix expression. [Oct.2010 – 4M]
26. Explain Dynamic Representation of Stack. [Oct.2010 – 4M]
27. Write a program to reverse a string using Stack. [Oct.2010 – 4M]
28. Convert the following infix to prefix expression: [Oct.2010 – 4M]
i. $A + B * C/D$ ii. $A + B + C + D$

- [Oct.2010 – 4M] 29. Convert the following Infix Expression into Postfix Expression. $A * (B + C \wedge D) - E / F * (G + D)$.
- [Apr.2010 – 4M] 30. Explain algorithm to convert infix expression to its equivalent postfix expression.
- [Apr.2010 – 4M] 31. Evaluate the following postfix expression: 4, 5, 4, 2, \wedge , +, *, 2, 2, \wedge , 9, 3, 1, *, -
- [Apr.2010 – 4M] 32. Convert the following infix expression into postfix expression:
 $A * (B + C \wedge D) - E / F * (G + D)$
- [Apr.2010 – 4M] 33. Write a 'C' program to reverse given string using stack.
- [Oct.2009 – 4M] 34. Explain algorithm to evaluate postfix expression.
- [Oct.2009 – 4M] 35. Convert following infix expressions into postfix.
a. $A + B * C$ b. $A + B + C$
- [Oct.2009 – 4M] 36. Explain Static and Dynamic Representation of Stack.
- [Oct.2009 – 4M] 37. Write a program to accept size of stack and add elements onto the stack one by one which are accepted from user until stack is full.
- [Oct.2009 – 4M] 38. What is Queue? What are the various Applications of Queue? How Queue is differ from Stack?



1. Introduction

Apr.15 Oct.14 – 2M
What is use of tree? How
it is differ from linked list?

2

A tree is a data structure which represents a hierarchical tree structure with a set of linked nodes. It is an acyclic connected graph where each node is connected to a set of zero or more children nodes, and also it can have at most one parent node.

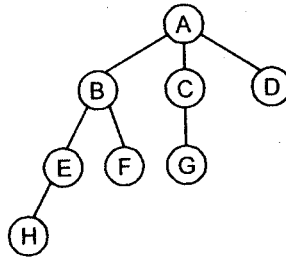
There are many different ways to represent trees, but in common representations either it represents the nodes as records allocated memory dynamically with pointers to their children, their parents (or both), or as items in an array, with relationships between them according to their positions in the array.

2. Tree Terminology

Definition of a Tree

A tree is a finite set of nodes with one specially designated node called the **root** and the remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1 and T_n where each of those sets is a tree. T_1 to T_n are called the sub trees of the root.

Example



Tree

In this example, A,B,C,D,E,F,G and H form a set of nodes with A as the root. The remaining nodes are partitioned into three sets (trees) with B,C and D as their respective roots.

Null Tree

A tree with no nodes is a Null Tree.

Node

A node of a tree is an item of information along with the branches to other nodes. The tree shown has 8 nodes.

Leaf Node

A leaf node is a terminal node of a tree. It does not have any nodes connected to it. All other nodes are called non-leaf nodes, or internal nodes.

H, F, G and D are leaf nodes.

Degree of a Node

The number of subtrees of a node is called its degree.

The degree of A is 3, B is 2 and D is 0. The degree of leaf nodes is zero.

Degree of the Tree

The degree of a tree is the maximum degree of the nodes in the tree.

The degree of the shown tree is 3.

1

Apr.2011 – 2M

Define: Degree of a Tree

Parent Node

A parent node is a node having other nodes connected to it. These nodes are called the children of that node.

The root is the parent of all its subtrees. A,B, and C are parent nodes.

Siblings

Children of the same parent are called siblings.

B,C and D are siblings. E and F are siblings.

Descendents

The descendents of a node are all those nodes which are reachable from that node.

Example

E, F and H are descendents of B.

2

Apr.15,Oct.14 – 2M

What is Ancestor of Node?

Ancestors

The ancestors of a node are all the nodes along the path from the root to that node.

Examples

B and A are ancestors of E.

Level of a node

This indicates the position of a node in the hierarchy. The level of any node = level of its parent + 1. The root is considered to be at level 0.

Examples

B, C and D are the level 1. H is at level 3.

1

Apr.2012 – 2M

Define the following terms: Height of a Tree

Height or Depth of a Tree

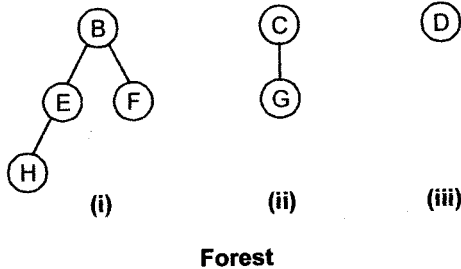
The maximum level of any node in the tree is the height or depth of the tree.

The given tree has a height = 3. This equals the length of the longest path from the root to any leaf.

Forest

A forest is a set of $n \geq 0$ disjoint trees i.e. if we remove the root, we get a forest of trees.

Example



These three trees form the forest if A is removed.

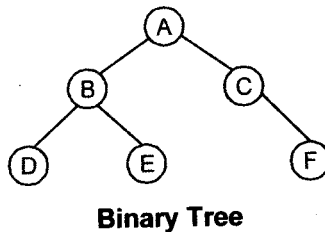
3. Binary Trees

1

Oct.2011 – 2M
Define: Binary Tree

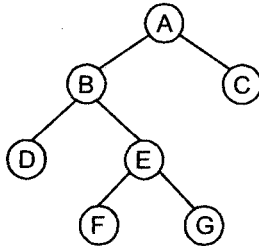
A binary tree is a finite set of nodes, which is empty or partitioned into three sets; one which is the root and the other two are binary trees called its left and right subtrees.

It is a tree where every node can have at the most two branches (children).



Strictly Binary Tree

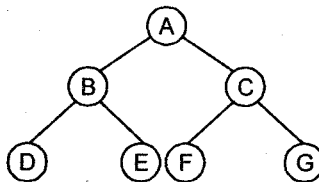
A strictly binary tree is a binary tree where all non-leaf nodes have two branches.



Strictly Binary Tree

Complete Binary Tree

A complete binary tree is a strictly binary tree with all its leaf nodes at the same level, d (d is the height or depth of the tree).



Complete Binary Tree

The maximum number of nodes on level i is 2^i .

The complete binary tree with a total of d levels (from 0 to $d-1$) contains

$$\text{number of nodes} = \sum_{i=0}^{d-1} 2^i = 2^d - 1$$

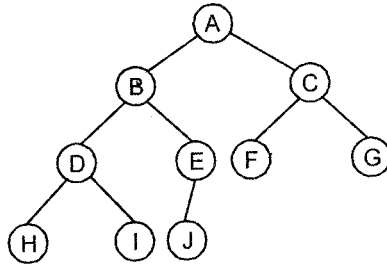
Almost complete binary tree

A binary tree with d levels is almost complete if levels 0 to $d-2$ are full and the last level i.e. level d is partially filled from left to right.

1

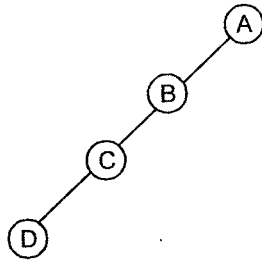
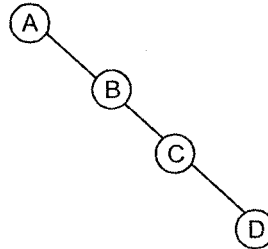
Oct.2009 – 2M

Define Almost Complete Binary Tree.

**Almost Complete Binary Tree**

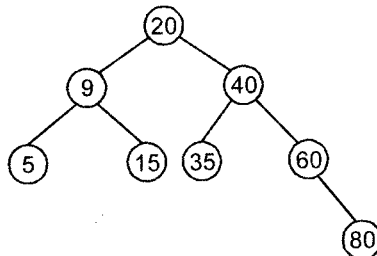
Skewed Binary Tree

The branches of this tree have either only left branches or only right branches. Accordingly the tree is called left skewed or right skewed tree.

**Left skewed binary tree****Right skewed binary tree**

Binary Search Tree

A binary search tree is a binary tree in which the nodes are arranged according to their values. The left node has a value less than its parent and the right node has a value greater than the parent node, i.e. all nodes in the left subtree have values less than the root and those in the right subtree have values greater than the root.

**Binary Search Tree**

4. Representation of Binary Trees

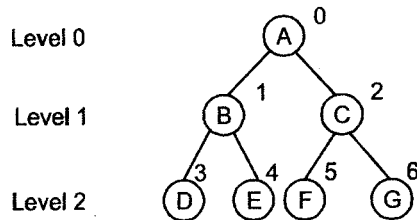
A binary tree can be represented using

1. Sequential Representation
2. Linked Representation

1. **Sequential Representation:** In this method, we will number each node of the tree starting from the root. Nodes on the same level will be numbered left to right. Since a binary tree with total 'd' levels will have maximum of $2^d - 1$ nodes, we can use an array of size $2^d - 1$ to represent the binary tree.

Thus, we can use static allocation method to represent a binary tree.

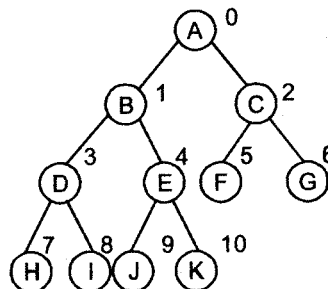
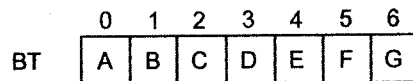
Example

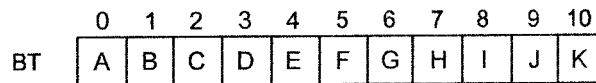


Complete Binary Tree

Here, the number of levels is 3. \therefore we need an array of size $2^3 - 1 = 7$ elements.

The tree representation will be:





Almost Complete Binary Tree

This representation allows random access to any node as shown

- i. The i^{th} node is at location i where
($0 \leq i < n$)
- ii. The parent of node i is at location $(i-1)/2$ – ($i=0$ is the root and has no parent)

Example

$$\begin{aligned} \text{parent of J (node 9)} &= (9-1)/2 \\ &= 8/2 = 4 \end{aligned}$$

i.e. E.

- iii. Left child of a node i is present at position $2i + 1$ if $2i + 1 < n$
If $2i + 1 \geq n$, the node i does not have a left child.

Example

$$\begin{aligned} \text{Left child of B (node 1)} &= 2 \times 1 + 1 \\ &= 3 \\ &= \text{D} \end{aligned}$$

$$\begin{aligned} \text{Left child of F (node 5)} &= 2 \times 5 + 1 \\ &= 10 + 1 \\ &= 11 \\ &= n \end{aligned}$$

Thus, F does not have a left child.

- iv. Right child of a node i is present at position $2i + 2$ if $2i + 2 < n$. If $2i + 2 \geq n$ then node i does not have a right child.

Example

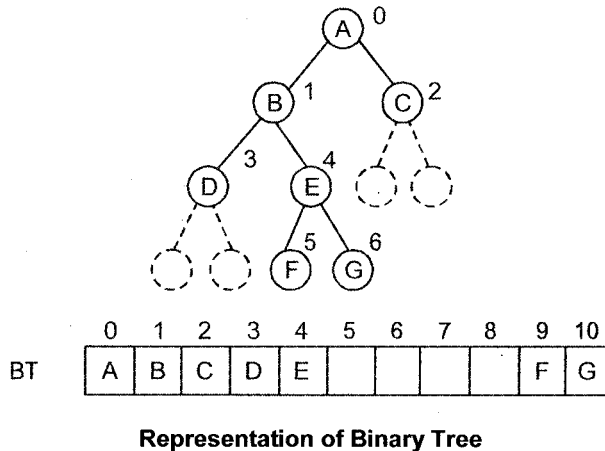
$$\begin{aligned} \text{Right child of B (node 1)} &= 2 \times 1 + 2 \\ &= 2 + 2 = 4 \end{aligned}$$

Since $4 < 7$, the node E is the right child.

$$\text{Right child of G (node 6)} = 2 \times 6 + 2 = 14 > n$$

Thus G does not have a right child.

Note: The above formulae will work only for almost complete binary trees. Any binary tree can be converted to almost complete binary tree by showing dummy nodes as shown.

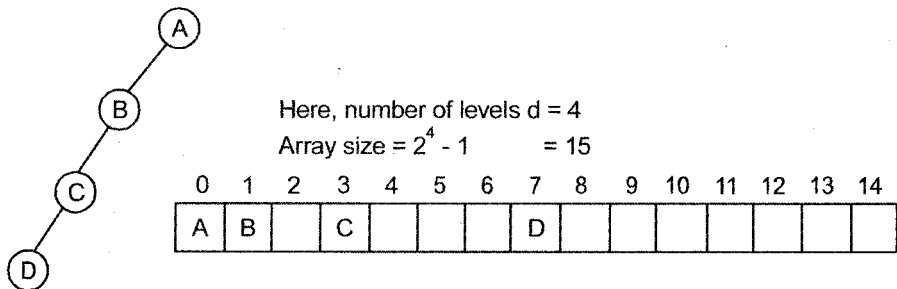


Disadvantages of Sequential representation

- i. Since we are using arrays, there is a limitation on its size i.e. we cannot store information of more nodes than the specified array size.
- ii. If the tree is not an almost complete binary tree, many array positions will be unutilized.

Example

Consider the skewed tree and its representation



Representation of skewed tree

Thus, only 4 positions are occupied and 11 are wasted.

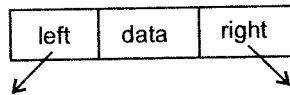
- iii. Nodes cannot be inserted or deleted. Hence, the linked representation is preferred.

2. **Linked Representation:** This is a more flexible representation and uses the dynamic memory allocation. Since each node represents information and contains at the most two children, we can define a node structure as follows.

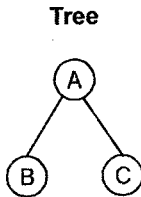
```
struct treenode
{
    struct treenode *left;
    int data;
    struct treenode *right;
};
typedef struct treenode *TREEPTR;
TREEPTR root;
```

We shall also be using the definition

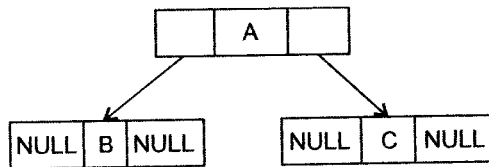
```
#define NODEALLOC (struct treenode *)malloc(sizeof(struct treenode))
```



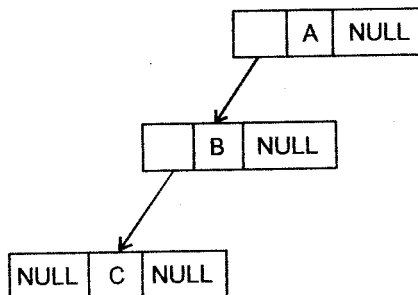
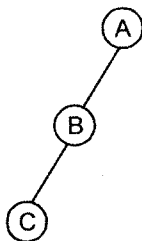
Example 1



Linked Representation



Example 2



Linked representation

5. Operations on Binary Tree

Following operations can be performed in the binary tree,

1. Create
2. Insert a value
3. Search for a given value
4. Delete a value
5. Traverse all the nodes (Preorder, Inorder and Postorder).

5.1 Creation of Binary Tree

A binary tree can be created by declaring a structure and creating pointer variables of it. The binary tree created must be empty initially and new nodes can be created by allocating memory dynamically to them. The node created using dynamic memory allocation will be having its own memory with size 6 i.e. 2 bytes for integer value stored at node, 2 bytes to store link of left child and remaining 2 bytes to store link of right child. Once the node is created its left link and right link should be kept NULL.

```
struct node
{
    int value;
    struct node *left;
    struct node *right;
};

struct node *create()
{
    struct node *newNode;
    newNode=(struct node *)malloc(sizeof(struct node));
    newNode->value= <value> /* User Input */
    newNode->left=NULL;
    newNode->right=NULL;
    return(newNode);
}
```

The create function is defined to create new nodes (newNode) with dynamic memory allocation and having its left and right links NULL and value to the node can be provided according to user choice.

5.2 Insertion

A binary tree is constructed by the repeated insertion of new nodes into a binary tree structure. Insertion must maintain the order of the tree such that values of the left node of a given node must be less than that node, and value of right node must be greater.

Inserting a node into a tree is actually two step operations.

First, the tree must be searched to find the position where the node is to be inserted. Second, on the completion of the search, the node is inserted at the specified position into the tree.

Assuming that duplicate entries are not allowed in the tree, two cases must be considered when constructing a binary tree.

1. Inserting into an-empty tree.

Before insertion of 4

After insertion of 4

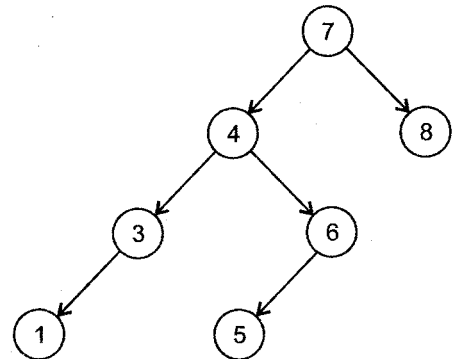
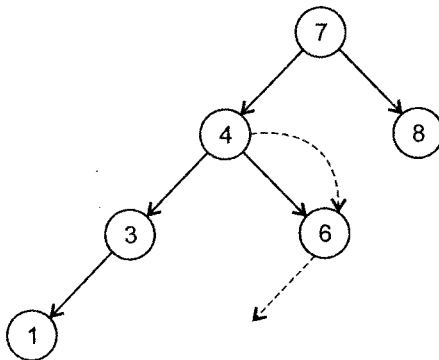


In this case, the node inserted into the tree is considered the root node.

2. Inserting into a non-empty tree.

Before insertion of 5

After insertion of 5



The tree must be searched to determine where the node is to be inserted.

Algorithm

- i. Compare the value of new node first to the root node of the tree.
- ii. If the value of the new node is less than the value of the root node.

if the left subtree is empty insert new node as the left child of the root node

else, the search continues down the left subtree.

iii. If the value of the new node is greater than the value of the root node.

if the right subtree is empty, insert new node as the right leaf of the root node

else, the search process continues down the right subtree.

iv. If the insertion node already exists in the tree, the terminate the procedure as it cannot insert duplicate node.

```
void insert(struct node *root, int child)
{
    struct node *tempNode;
    if(root==NULL)
    {
        root = newNode;
    }
    else
    {
        if(child<root->value)
        {
            if(root->left==NULL)
                root->left=newNode;
            else
                insert(root->left,child)
        }
        if(child>root->value)
        {
            if(root->right==NULL)
                root->right=newNode;
            else
                insert(root->right,child)
        }
        if(child==root->value)
            { printf("Duplicate Node..."); }
    }
}
```

5.3 Deletion

The algorithm to delete an arbitrary node from a binary tree is deceptively complex, as there are many special cases. The algorithm used for the delete function splits it into two separate operations,

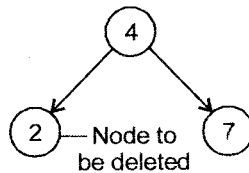
searching and deletion. Once the node which is to be deleted has been determined by the searching algorithm, it can be deleted from the tree. The algorithm must ensure that when the node is deleted from the tree, the ordering of the binary tree should be maintained.

Special Cases that have to be considered

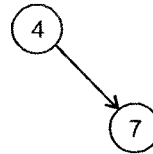
1. The node to be deleted has no children.

In this case the node may simply be deleted from the tree.

Before deletion of 2



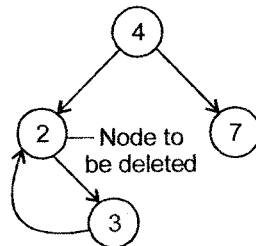
After deletion of 2



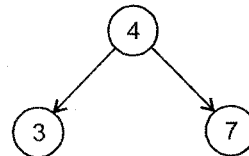
2. The node has one child.

The child node is appended to its grandparent. (The parent of the node to be deleted).

Before deletion of 2



After deletion of 2

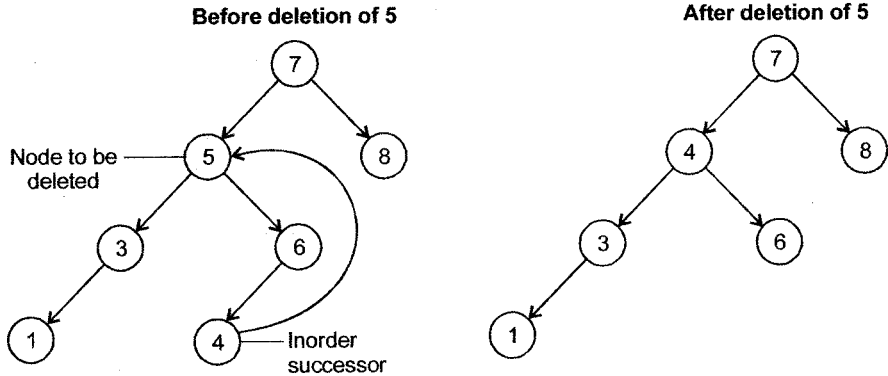


3. The node to be deleted has two children.

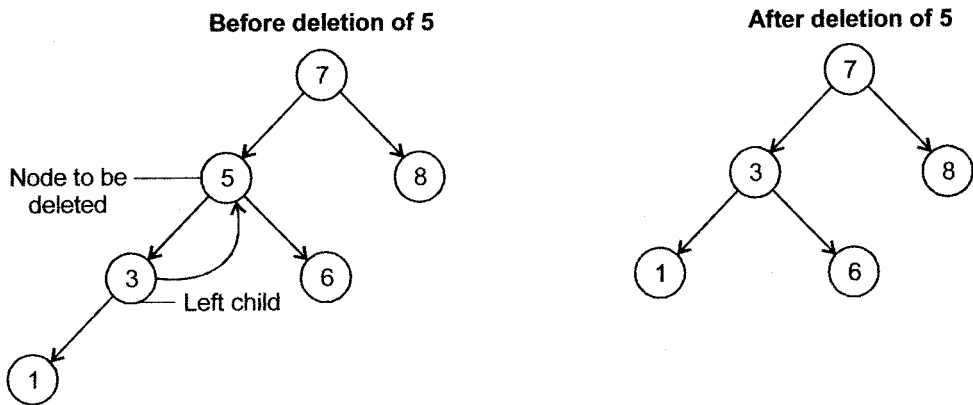
This case is much more complex than the previous two, because the order of the binary tree must be maintained.

In this algorithm it is important which node should be used in place of the node to be deleted.

- i. Use the inorder successor from right subtree of the node to be deleted.



- ii. Else if no right subtree exists replace the node to be deleted with its left child.



```

void delete(struct node *curr)
{
/* Node with single child*/
if((curr->left == NULL && curr->right != NULL) || (curr->left != NULL &&
curr->right == NULL))
{
if(curr->left == NULL && curr->right != NULL)
{ if(parent->left == curr)
{
parent->left = curr->right;
delete curr;
}
else
{
parent->right = curr->right;
delete curr;
}
}
}
}

```

```
}
else /* left child present, no right child */
{
    if(parent->left == curr)
    {
        parent->left = curr->left;
        delete curr;
        else
        {
            parent->right = curr->left;
            delete curr;
        }
    }
}
return;
}
/* A leaf node */
if( curr->left == NULL && curr->right == NULL)
{
    if(parent->left == curr)
        parent->left = NULL;
else
    parent->right = NULL;
    delete curr;
    return;
}
/* Node with 2 children */
/* replace node with smallest value in right subtree */
if (curr->left != NULL && curr->right != NULL)
{
    tree_node* chkr;
    chkr = curr->right;
    if((chkr->left == NULL) && (chkr->right == NULL))
    {
        curr = chkr;
        delete chkr;
        curr->right = NULL;
    }
else /* right child has children*/
    {
        /* if the node's right child has a left child Move all the way down
        left to locate smallest element*/
        if((curr->right) ->left != NULL)
        {
            struct node* lcurr;
            struct node* lcurrp;
            lcurrp = curr->right;
            lcurr = (curr->right) ->left;
            while(lcurr->left != NULL)
            {
                lcurrp = lcurr;
                lcurr = lcurr->left;
            }
        }
    }
}
```

```

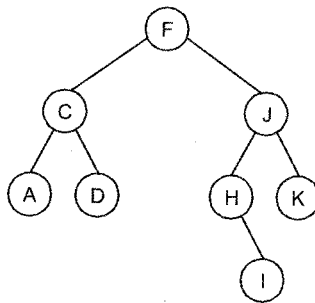
curr->key = lcurr->key;
delete lcurr;
lcurrp->left = NULL;
{
  else
  {
    struct* tmp;
    tmp = curr->right;
    curr->key = tmp->key;
    curr->right = tmp->right;
    delete tmp;
  }
}
return;
}
}

```

6. Traversing a Binary Tree

Traversing a binary tree means to do a print out of all the data elements in the tree in a specific order. According to this order there are three types of traversals.

All traversal algorithms are described with following diagram.



Binary Tree

These three types are as follows:

- i. Pre Order Traversal
- ii. In Order Traversal
- iii. Post Order Traversal

3

Oct.2012 – 2M

State different types of traversal technique of tree.

Oct.2011 – 4M

What are the different types of Tree Traversal methods? Explain any one with suitable example.

Oct.2010 – 4M

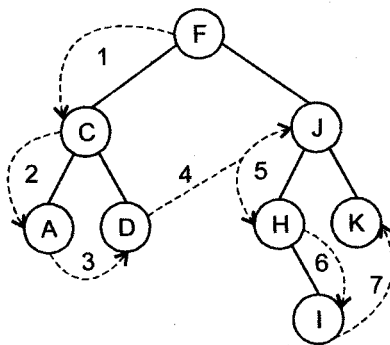
Discuss different Tree Traversal Methods.

6.1 Pre Order Traversal

A pre order traversal prints the contents of a sorted tree, in pre order. In other words, the contents of the root node are printed first, followed by left subtree and finally the right subtree. So in an In order traversal the result is in the following string: **F C A D J H I K**.

Pre order Traversal

- i. Do operation on root of the tree
- ii. Traverse left subtree
- iii. Traverse right subtree



```

void preorder(struct node *root)
{
    if(root!=NULL)
    {
        printf("%d", root->val);
        preorder(root->left);
        preorder(root->right);
    }
}
  
```

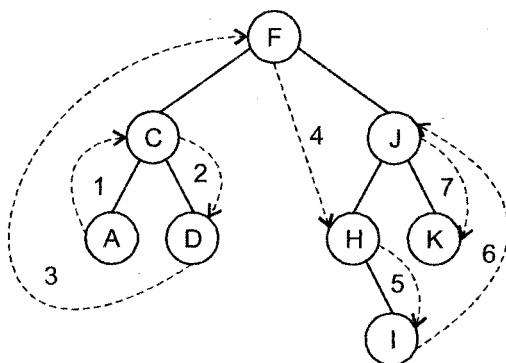
6.2 In Order Traversal

An in order traversal prints the contents of a sorted tree, in order. In other words, the lowest in value first, and then increasing in value as it traverses the tree. The order of a traversal would be 'a' to 'z' if the tree uses strings or characters, and would be increasing numerically from 0 if the tree contain

numerical values. So, as shown in figure, an in order traversal would result in the following string: **A C D F H I J K.**

In order Traversal

- i. Traverse left subtree
- ii. Do operation on root of the tree
- iii. Traverse right subtree



```

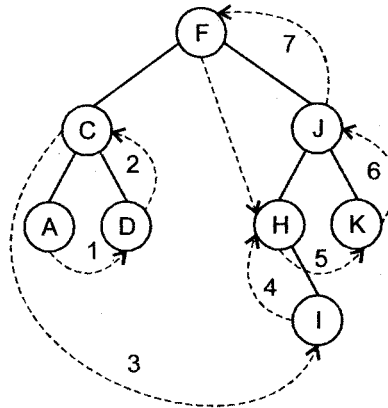
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder (root->left);
        printf("%d", root->val);
        inorder (root->right);
    }
}
  
```

6.3 Post Order Traversal

A post order traversal prints the contents of a sorted tree, in post order. In other words, the contents of the left subtree are printed first, followed by right subtree and finally the root node. So as shown in figure, an in order traversal would result in the following string: **A D C I H K J F.**

Post order Traversal

- i. Traverse left subtree
- ii. Traverse right subtree
- iii. Do operation on root of the tree



```
void postorder(struct node *root)
{
    if(root!=NULL)
    {
        postorder(root->left);
        postorder(root->right);
        printf("%d",root->val);
    }
}
```

6.4 Iterative Traversing

All the above recursive methods are implemented using recursive function. As we know recursive functions are executed by using stack. To execute above traversing methods stack space, proportional to the depth of the tree, is required. One of the methods to implement iterative traversing is to use stack, but to store intermediate nodes at different level instead of using it for recursion. Recursive traversal may be converted into an iterative as follows.

Iterative Preorder Traversal

Algorithm

- i. Push the root node into stack.
- ii. Consider the top item as current node from stack.
- iii. Print the value of current node.
- iv. If left child of current node is not NULL.
 - a. Push the left child of current node into stack.
else
If right child of current node is not NULL.
 - b. Push the right child of current node into stack.
else
 - c. Pop the topmost node from stack.
 - d. Repeat step no. 2 till stack gets empty.

```
void Preorder (rootNode)
{
    nodeStack.push (rootNode)
    while (!nodeStack.empty ())
    {
        currNode = nodeStack.peek () /*peek at top item*/
        printf ("%d", currNode->value)
        if ((currNode->left) != NULL)
            nodeStack.push (currNode->left) /*Put the next level of calls on stack */
        else
            if ((currNode->right) != NULL)
                nodeStack.push (currNode->right)
        else
            nodeStack.pop () /* Only do this if we didn't push anything on stack */
    }
}
```

Iterative Inorder Traversal

Algorithm

1. Push the root node into stack.
2. Consider the top item as current node from stack.

- iii. If left child of current node is not NULL.
 - a. Push the left child of current node into stack.
else
 If right child of current node is not NULL.
 - b. Push the right child of current node into stack.
else
 - c. Pop the topmost node from stack.
 - d. Print the value of current node.
 - e. Repeat step no. 2 till stack gets empty.

```
void Inorder (rootNode)
{
    nodeStack.push(rootNode)
    while (!nodeStack.empty ())
    {
        currNode = nodeStack.peek() /*peek at top item*/
        if ((currNode->left) != NULL)
            nodeStack.push(currNode->left) /*Put the next level of
            calls on stack */
        else
            if ((currNode->right) != NULL)
                nodeStack.push(currNode->right)
            else
                {
                    nodeStack.pop() /* Only do this if we didn't push
                    anything on stack */
                    printf ("%d", currNode->value);
                }
    }
}
```

Iterative Post Order Traversal

Algorithm

- i. Push the root node into stack.
- ii. Consider the top item as current node from stack.

- iii. If left child of current node is not NULL
 - a. Push the left child of current node into stack.
else
If right child of current node is not NULL.
 - b. Push the right child of current node into stack.
else
 - c. Print the value of current node.
 - d. Pop the topmost node from stack.
 - e. Repeat step no. 2 till stack gets empty.

```
void Postorder (rootNode)
{
    nodeStack.push(rootNode)
    while (!nodeStack.empty())
    {
        currNode = nodeStack.peek() /*peek at top item*/
        if ((currNode->left) != NULL)
            nodeStack.push(currNode->left) /*Put the next level of calls on stack */
        else
            if ((currNode->right) != NULL)
                nodeStack.push(currNode->right)
            else
            {
                printf ("%d", currNode->value)
                nodeStack.pop() /*Only do this if we didn't push anything stack */
            }
    }
}
```

7. Binary Search

The binary search is the standard method for searching the required element through a sorted array. It is much more efficient than a linear search, where we sequentially go through the array elements until the target is found. Binary search requires that the elements in an array should be in order.

The binary search repeatedly divides the array in two parts and each time restricting the search to the half part that contains the target element.

Here the binary search algorithm is implemented using array which is having static memory allocation. The binary search algorithm can also be implemented using data structures with dynamic storage and allows searching to be done efficiently. A linked list structure is not efficient when searching for a specific item as the node can only be accessed sequentially.

7.1 Binary Trees and Binary Search Trees

**1**

Apr.2012 – 2M

What is a Binary Search Tree?

Binary search tree is a special kind of tree, which is ideal for storing data for efficient searching. The binary search tree is a hierarchical structure in which data is accessed similar to a binary search algorithm.

A binary search tree is itself a special kind of binary tree. A binary tree is a tree which is either empty or consists of a node called the root, together with two children called the left subtree and the right subtree of the root. Each of these children is itself a binary tree.

A binary search tree satisfies the following additional conditions:

- i. Each element has a key value which is used to order the elements.
- ii. The keys of all the elements in the left subtree of the root are less than the key in the root.
- iii. The key in the root is less than all the keys in the right subtree.
- iv. The left and right subtrees of the root are again having a structure of search trees.
- v. The tree must be searched to determine where the node is present having required value to be searched.

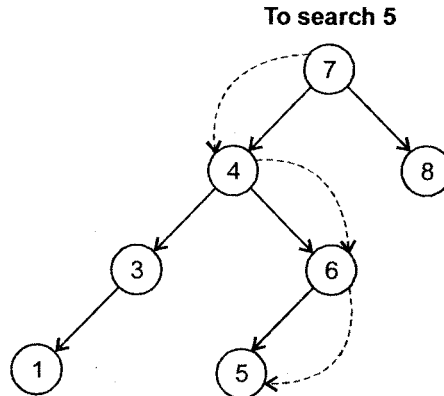
Algorithm

- i. Compare the given key value first to the root node of the tree.
- ii. If the value of the key is less than the value of the root node
if the left subtree is not empty, the search continues down the left subtree.
else display the message of not found and quit.
- iii. If the value of the key is greater than the value of the root node.
if the right subtree is not empty, search process continues down the right subtree.
else display the message of not found and quit.
- iv. If the key value not exists in the tree, then terminate the procedure.

1

Oct.2011 – 4M

Write an algorithm for Binary Search Method.



```

struct Node * BSearch( Node *root, int key)
{ while(root != NULL)
{ if(root->data == key)
return n;
parent=root;
if(root->data > key)
root = root->left;
else
root = root->right;
}
return NULL;
}

```

As explained in algorithm, the function BSearch is defined to search given key value in a Binary Search Tree. Firstly, the tree is checked whether it is empty or not. If tree is not empty, the key is compared with the value at root node. If it is less than value at root, the left child is selected and it is considered as root else if it is greater, the right child is considered as root and same process is applied. If key value is exactly equal to root or some other node, the node will be returned by the function else NULL value will be returned as element is not found.

```
/* Program to Implement Binary Search Tree*/
#include<stdio.h>
#include<conio.h>
struct node
{
    struct node*left;
    int value;
    struct node*right;
};
struct node *root,*parent,*newNode;
/* Define the functions for Create, Insert, BSearch, Delete Preorder,
Inorder and Postorder functions discussed above */
void main()
{
    int ch, a;
    do
    {
        clrscr();
        printf("1.Insert\n2.Search\n3.Delete\n4.Traverse\n5.Exit\n");
        printf("\nEnter your choice:-");
        scanf("%d",&ch);
        if(ch==1)
        {
            newNode=Create();
            Insert(root,newNode->value);
        }
        if(ch==2)
        {
            printf("Enter value to be searched:-");
            scanf("%d",&a);
            newNode=BSearch(root,a);
            if(newNode==NULL)
                printf("Element not found");
            else
```

```
        printf("Element Found...");
    }
    if(ch==3)
    {
        printf("Enter value to be searched:-");
        scanf("%d",&a);
        newNode=BSearch(root,a);
        if(newNode==NULL)
            printf("Element not found");
        else
            Delete(newNode);
    }
    if(ch==4)
    {
        printf("1.Preorder\n2.Inorder\n3.Postorder\n");
        printf("\nEnter your choice:-");
        scanf("%d",&a);
        if(a==1)
            Preorder(root);
        if(a==2)
            Inorder(root);
        if(a==3)
            Postorder(root);
    } while(ch<5);
}
```

8. AVL Trees

AVL trees are self-adjusting, height-balanced binary search trees and are named after the inventors: **Adelson-Velskii** and **Landis**. An AVL tree is a special type of binary tree that is always 'partially' balanced. The criteria that is used to determine the 'balanced-ness' is the difference between the heights of subtrees of a root in the tree.

The 'height' of tree is the 'number of levels' in the tree. Generally, the height of a tree is defined as follows:

- i. The height of a tree with no elements is 0
- ii. The height of a tree with 1 element is 1
- iii. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

An AVL tree is a binary tree in which the difference between the height of the right and left subtrees (or the root node) is never more than one. The **height** of a binary tree is the maximum path length from the root to a leaf. A single-node binary tree has height 0, and an empty binary tree has height -1. As another example, the following binary tree has height 3.

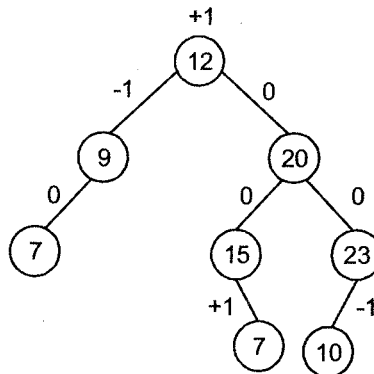
2

Oct.14,12 – 2M

What is Balance Factor?
How it is calculated?

As discussed above, an **AVL tree** is a binary search tree in which every node is **height balanced**, that is, the difference in the heights of its two subtrees is at most 1. The **balance factor** of a node can be calculated as the height of its right subtree minus the height of its left subtree. Hence in AVL tree each node has a balance factor of -1 , 0 , or $+1$. Note that a balance factor of -1 means that the left-subtree is heavy, a balance factor of $+1$ means that the right-subtree is heavy and a balance factor of 0 means that both left-subtree and right-subtree are having same height.

For example, in the following AVL tree, node.



That the root node with balance factor $+1$ has a right subtree of height 1 more than the height of the left subtree. (The balance factors are shown at the top of each node.)

A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees. Whenever we insert or delete an item, the AVL tree can be 'violated'. We must then restore it by performing a set of manipulations called 'rotations' on the tree. These rotations are having two types: single rotations and double rotations

In an AVL tree, the Balance Factor (BF) of a node means the difference between the heights of the left and right subtrees of the node must be -1 , 0 and 1 . If any node has a balance factor other than these values then the rotations are required to balance the tree.

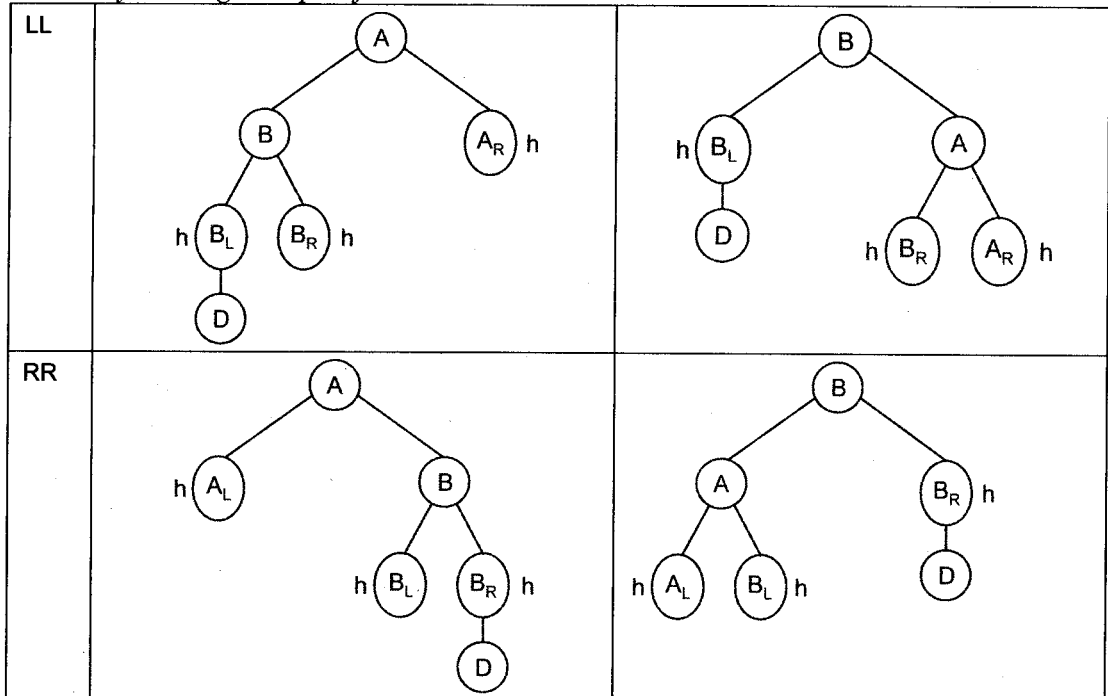
The four rotations performed to balance a tree are

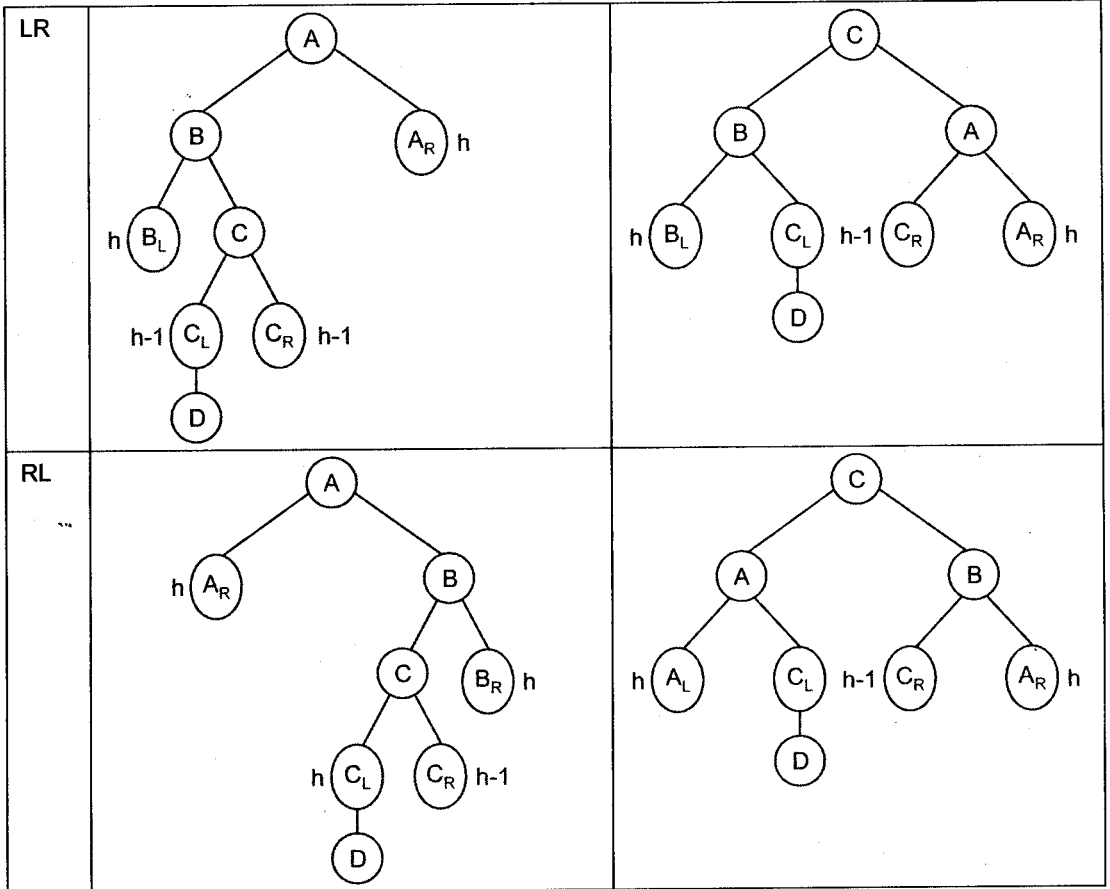
- i. LL (Left to left)
 - ii. LR (Left to right)
 - iii. RR (Right to right)
 - iv. RL (Right to left)
- i. **Left to Left Rotation (LL):** This operation is required when insertion in left subtree of left child of ancestor node (Unbalanced node whose $BF > 1$) is occurred.
 - ii. **Right Rotation (RR):** A right rotation is a mirror of the left rotation operation described above. This operation is required when insertion in right subtree of right child of ancestor node is occurred.
 - iii. **Left-Right Rotation (LR):** Sometimes a single left rotation is not sufficient to balance an unbalanced tree. This operation is required when insertion of a new node in right subtree of left child of ancestor node is occurred. In this method, RR rotation has to be applied on the node nearest to the pivot and newly inserted node. Then LL rotation has to apply on the pivot node.
 - iv. **Right-Left Rotation (RL):** This must be performed when attempting to balance a tree which has a left subtree that is right heavy. In this method, LL rotation has to be applied on the node nearest to the pivot and newly inserted node. Then RR rotation has to apply on the pivot node.

2
Oct.15, Apr.12 – 4M
Explain different types of AVL Rotations with an example.

1
Apr.2015 – 4M
What is height-balanced tree? Explain LL and RR rotations.

Consider the following examples for all rotations:





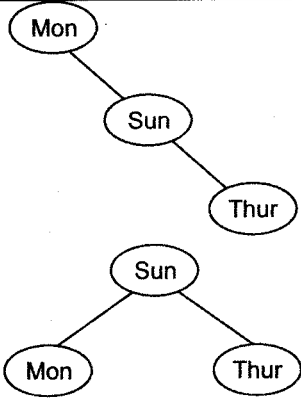
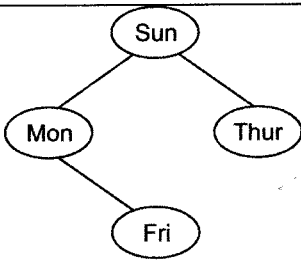
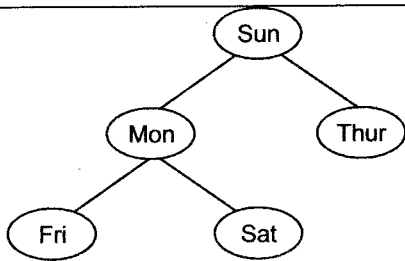
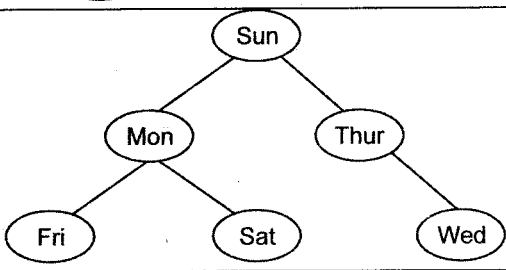
Solved Examples

1
 Oct.2012 – 4M

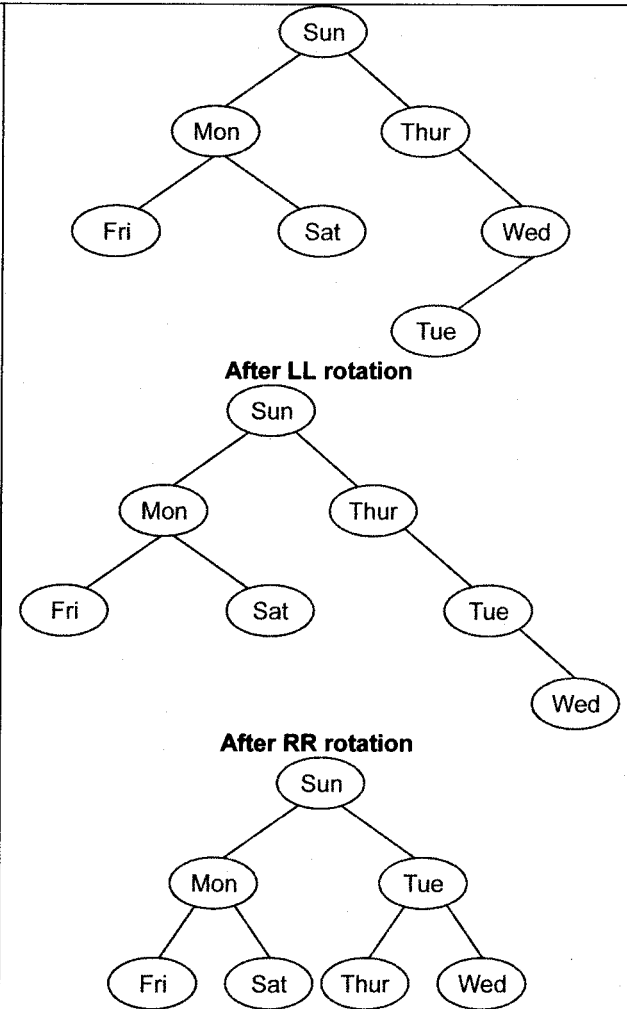
1. **Build on AVL Tree for the following data:**
 Mon, SUN, Thur, Fri, Sat, Wed, Tue.

Solution
 Function to find height of binary tree

	New Identifier	After Insertion	After Rebalancing
11.	Mon		No needed rebalancing
22.	Sun		No needed rebalancing

33.	Thur	 <pre> graph TD Sun1((Sun)) --- Mon1((Mon)) Sun1 --- Thur1((Thur)) Sun2((Sun)) --- Mon2((Mon)) Sun2 --- Thur2((Thur)) </pre>	<p>No needed rebalancing</p> <p>After applying RR rotation</p>
44.	Fri	 <pre> graph TD Sun1((Sun)) --- Mon1((Mon)) Sun1 --- Thur1((Thur)) Mon1 --- Fri1((Fri)) </pre>	<p>No needed rebalancing</p>
55.	Sat	 <pre> graph TD Sun1((Sun)) --- Mon1((Mon)) Sun1 --- Thur1((Thur)) Mon1 --- Fri1((Fri)) Mon1 --- Sat1((Sat)) </pre>	<p>No needed rebalancing</p>
66.	Wed	 <pre> graph TD Sun1((Sun)) --- Mon1((Mon)) Sun1 --- Thur1((Thur)) Mon1 --- Fri1((Fri)) Mon1 --- Sat1((Sat)) Thur1 --- Wed1((Wed)) </pre>	<p>No needed rebalancing</p>

77. Tue



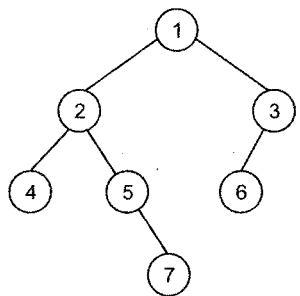
Apply RL rotation.
Apply LL on Wed.

1

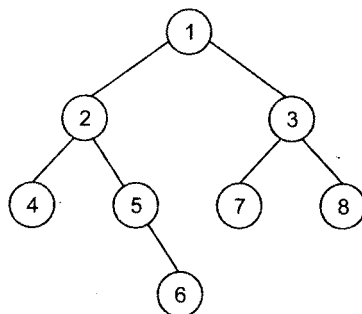
Oct.2012 – 4M

2. Give the Preorder, Inorder and Postorder Traversal of the following trees:

i.



ii.

**Solution**

Consider root is denoted by N, left subtree as L, right subtree as R.

i. Preorder Traversal (NLR)

- a. Visit the root.
- b. Traverse the left subtree of root in preorder.
- c. Traverse the right subtree of root in preorder.

ii. Inorder Traversal (LNR)

- a. Traverse the left subtree of root in inorder.
- b. Visit the root.
- c. Traverse the right subtree of root in inorder.

iii. Postorder Traversal (LRN)

- a. Traverse the left subtree of root in postorder.
- b. Traverse the right subtree of root in postorder.
- c. Visit the root.

According to the above rules,

Consider the first given tree

- i. Preorder Traversal: 1, 2, 4, 5, 7, 3, 6
- ii. Inorder Traversal: 4, 2, 5, 7, 1, 6, 3
- iii. Postorder Traversal: 4, 7, 5, 2, 6, 3, 1

Consider the second given tree,

- i. Preorder Traversal: 1, 2, 4, 5, 6, 3, 7, 8
- ii. Inorder Traversal: 4, 2, 5, 6, 1, 7, 3, 8
- iii. Postorder Traversal: 4, 6, 5, 2, 7, 8, 3, 1

3

Apr.12, 11, Oct.10 – 4M

3. Write the recursive functions of Pre-order and Post-order Traversal in a BST.

Solution

In preorder traversal method, sequence of traversing the nodes is visit the root value first, then visit left and lastly visit right child.

In case of postorder traversal method, sequence of traversing the nodes is visit the left child first, then visit right child and lastly visit the root value.

Preorder(NLR) and Postorder(LRN) traversal methods using recursive functions for BST are as follows,

```
void preorder(struct node *ptr) /* it follows the NLR(Root-
>Left->Right) */
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        printf("%d ",ptr->info);
        preorder(ptr->lchild);
        preorder(ptr->rchild);
    }
} /*End of preorder() */

void postorder(struct node *ptr) /* it follows the LRN (Left-
>Right->Root) */
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        inorder(ptr->lchild);
        inorder(ptr->rchild);
        printf("%d ",ptr->info);
    }
} /*End of postorder() */
```

1

Oct.2011 – 4M

4. Write a program to construct a Binary Search Tree and Traverse using Inorder and Preorder Traversal.

Solution

```
#include<stdio.h>
#include<malloc.h>
struct node /* structure of node in binary search tree */
```

```
{
    int info;
    struct node *lchild;
    struct node *rchild;
}*root;
main()          /*main function */
{
    int choice,num;
    root=NULL;
    while(1)
    {
        printf("\n");
        printf("1.Create\n");
        printf("2.Inorder Traversal\n");
        printf("3.Preorder Traversal\n");
        printf("4.Display\n");
        printf("5.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: printf("Enter the number to be inserted : ");
                    scanf("%d",&num);
                    insert(num);
                    break;
            case 2: inorder(root);
                    break;
            case 3: preorder(root);
                    break;
            case 4: display(root,1);
                    break;
            case 5:exit();
            default: printf("Wrong choice\n");
        }
        /*End of switch */
    }
    /*End of while */
}
/*End of main()*/
/* find function shows the proper position of new node to insert in
tree */
find(int item,struct node **par,struct node **loc)
{
    struct node *ptr,*ptrsave;
    if(root==NULL) /*tree empty*/
    {
        *loc=NULL;
        *par=NULL;
        return;
    }
    if(item==root->info) /*item is at root*/
```

```
{
    *loc=root;
    *par=NULL;
    return;
}
/*Initialize ptr and ptrsave*/
if(iteminfo)
    ptr=root->lchild;
else
    ptr=root->rchild;
ptrsave=root;
while(ptr!=NULL)
{
    if(item==ptr->info)
    {
        *loc=ptr;
        *par=ptrsave;
        return;
    }
    ptrsave=ptr;
    if(iteminfo)
        ptr=ptr->lchild;
    else
        ptr=ptr->rchild;
}/*End of while */
*loc=NULL; /*item not found*/
*par=ptrsave;
}/*End of find()*/
insert(int item) /* insert function inserts new node at its position in
tree */
{
    struct node *tmp,*parent,*location;
    find(item,&parent,&location);
    if(location!=NULL)
    {
        printf("Item already present");
        return;
    }
    tmp=(struct node *)malloc(sizeof(struct node));
    tmp->info=item;
    tmp->lchild=NULL;
    tmp->rchild=NULL;
    if(parent==NULL)
        root=tmp;
    else
        if(item< parent->info)
            parent->lchild=tmp;
        else
            parent->rchild=tmp;
```



```

}/*End of insert()*/
preorder(struct node *ptr) /* it follows the NLR (Root->Left->Right)*/
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        printf("%d ",ptr->info);
        preorder(ptr->lchild);
        preorder(ptr->rchild);
    }
}/*End of preorder()*/
inorder(struct node *ptr) /* it follows the LNR (Left->Root->Right)*/
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        inorder(ptr->lchild);
        printf("%d ",ptr->info);
        inorder(ptr->rchild);
    }
}/*End of inorder()*/
display(struct node *ptr,int level)
{
    int i;
    if (ptr!=NULL)
    {
        display(ptr->rchild, level+1);
        printf("\n");
        for (i = 0; i < level; i++)
            printf("    ");
        printf("%d", ptr->info);
        display(ptr->lchild, level+1);
    }/*End of if*/
}/*End of display()*/

```

5. **Define Height Balance Tree. Built an AVL tree for the following data:**

Jan, Feb, Mar, Apr, May, June, July

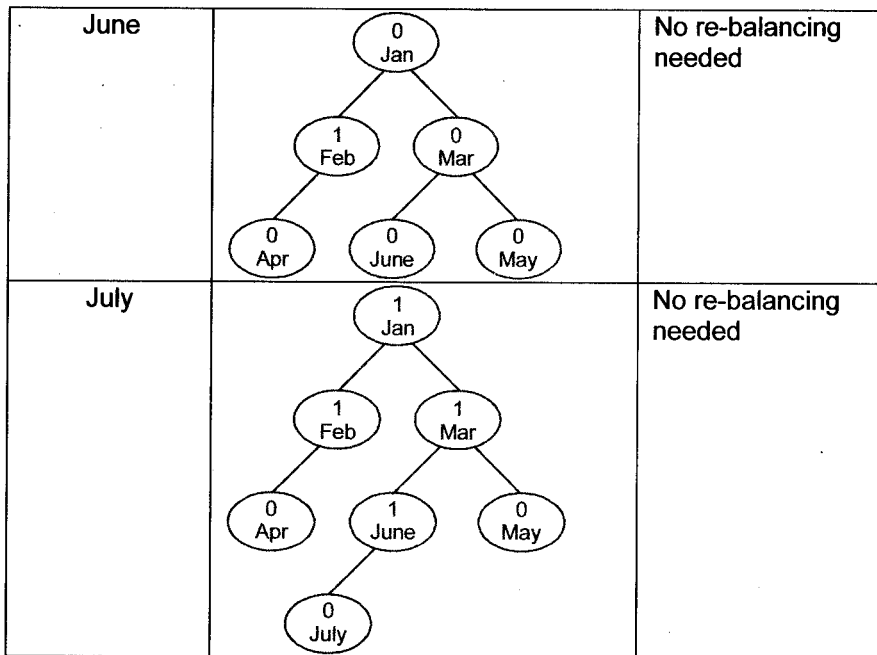
Solution

A **height-balanced tree** is a data structure tree which keeps its children similar in height to within some defined limit. For example, children of an AVL tree differ in height by at most 1.

A tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too. An empty tree is height-balanced.

Since the given tree is a binary search tree, so insert the data according to the alphabetical position in the tree.

New Identifier	After Insertion	After rebalancing
Jan		No re-balancing needed
Feb		No re-balancing needed
Mar		No re-balancing needed
Apr		No re-balancing needed
May		No re-balancing needed



6. Write a function to count number of nodes in a given tree.

Solution

Function to count number of nodes in a given tree.

```

int count(node *t)
{
    stack s;
    int count = 0;
    while(t!=NULL)
    {
        count = count + 1;
        s.push(t);
        t=t->left;
    }
    while(!s.empty())
    {
        t=s.pop();
        t=t->right;
        while(t!=NULL)
        { count = count + 1;
          s.push(t);
          t=t->left;
        }
    }
    return(count);
}

```

1

Oct.2014 - 4M

7. Construct Binary Search tree for following data:
 Jan, Oct, Dec, Nov, Feb, Mar, Apr, Sept, May, Jul, Jun,
 Aug

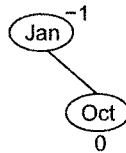
1
 Oct.2014 - 4M

Solution

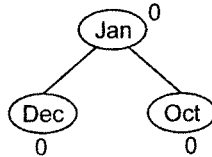
- i. Jan



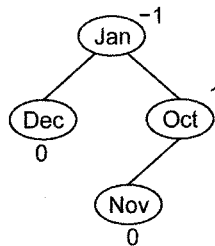
- ii. Oct



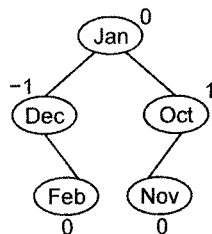
- iii. Dec



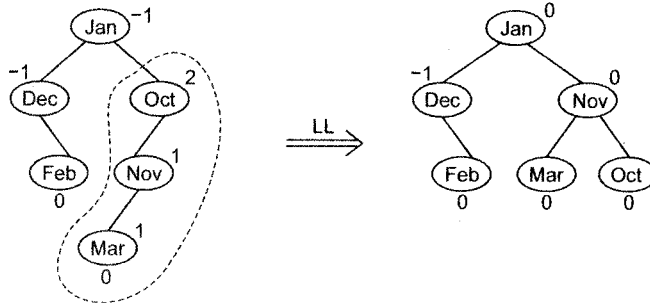
- iv. Nov



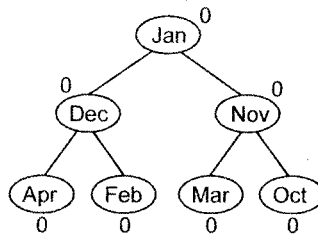
- v. Feb



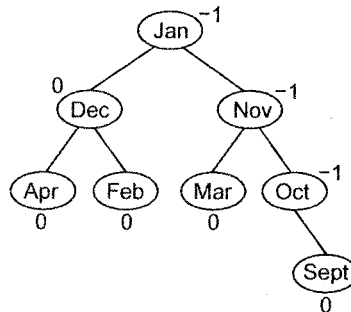
vi. Mar



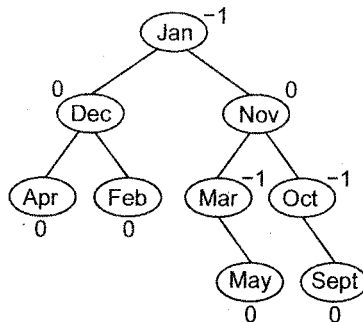
vii. Apr



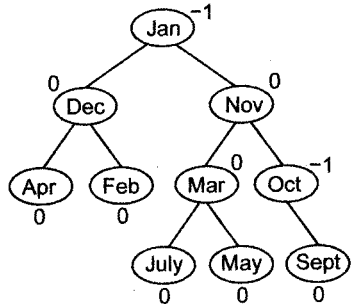
viii. Sept



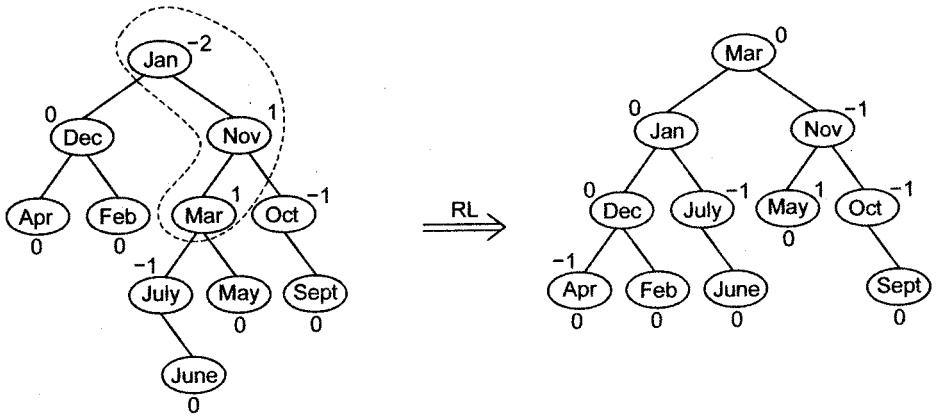
ix. May



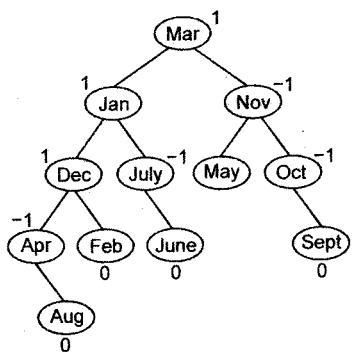
x. Jul



xi. Jun



xii. Aug





PU Questions

2 Marks

1. What is Ancestor of Node?
2. What is use of tree? How it is differ from linked list?
3. What is balance factor? How it is calculated?
4. State different types of Traversal Technique of Tree.
5. What is Balance Factor? How it is calculated?
6. What is a Binary Search Tree?
7. Define the following terms: Height of a Tree
8. Define Binary Tree.
9. Define "Degree of a Tree".
10. Define Almost Complete Binary Tree.

[Oct.14, Apr.15 – 2M]

[Oct.14, Apr.15 – 2M]

[Oct.14,12 – 2M]

[Oct.2012 – 2M]

[Oct.2012 – 2M]

[Apr.2012– 2M]

[Apr.2012 – 2M]

[Oct.2011 – 2M]

[Apr.2011 – 2M]

[Oct.2009 – 2M]

4 Marks

1. Write the recursive functions to traverse a tree by using inorder(), preorder() and postorder() traversing techniques.
2. Build AVL tree for the following data: FRI, MON, SAT, WED, SUN, TUE, THUR 1.
3. Write a function to count the number of leaf nodes in a tree.
4. Write a function to display mirror image of given tree.
5. What is height-balanced tree? Explain LL and RR rotations.
6. Explain different types of AVL rotations in details.
7. Construct Binary Search tree for following data: Jan, Oct, Dec, Nov, Feb, Mar, Apr, Sept, May, Jul, Jun, Aug
8. Write a function to count number of nodes in a given tree.
9. Build on AVL Tree for the following data: Mon, Sun, Thur, Fri, Sat, Wed, Tue.
10. Give the Preorder, Inorder and Postorder Traversal of the following trees:

[Oct.2015– 4M]

[Oct.2015– 4M]

[Apr.2015– 4M]

[Apr.2015– 4M]

[Apr.2015– 4M]

[Oct.14, Apr.11– 4M]

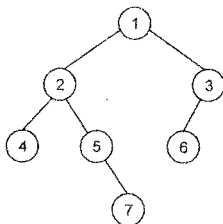
[Oct.2014– 4M]

[Oct.2014– 4M]

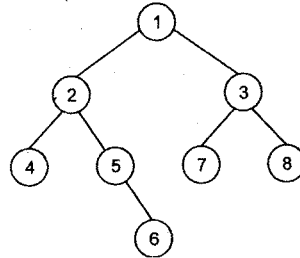
[Oct.2012– 4M]

[Oct.2012– 4M]

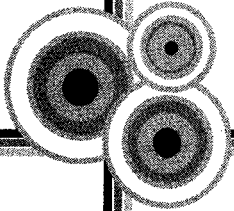
i.



ii.

[Apr.2012 – 4M][Apr.12,11,Oct.10 – 4M][Oct.2011 – 4M][Oct.2011 – 4M][Oct.2011 – 4M][Oct.2011 – 4M][Oct.2011 – 4M][Oct.2011 – 4M][Apr.2011 – 4M][Apr.2011 – 4M][Apr.2011 – 4M][Oct.2010 – 4M][Oct.10,09 – 4M][Apr.2010 – 4M][Oct.2009 – 4M]

11. Write the recursive functions of Pre-order and Post-order Traversal in a BST.
12. Construct Binary Tree for the following data: 12, 30,6,7,25,10,15,18,33
13. Write an algorithm for Binary Search Method.
14. What are the different types of Tree Traversal methods? Explain any one with suitable example.
15. Write a program to construct a Binary Search Tree and Traverse using Inorder and Preorder Traversal.
16. Define Height Balance Tree. Built an AVL tree for the following data: Jan, Feb, Mar, Apr, May, June, July
17. Write a program to count Leaf and non-leaf nodes of a tree.
18. Construct Binary Tree for the following data: 12, 30,6,7,25,10,15,18,33
19. Write a function to find height of Binary Tree.
20. Construct AVL Tree for the following;
Avinash, Janardan, Suresh, Prashant, Mahesh, Amar, Anup, Sachin, Sahdev, Vijay, Dhurandhar, Nitin.
21. Construct a binary tree for the following data: 12, 30, 6, 7, 25, 10, 15, 18, 33
22. Discuss different Tree Traversal Methods.
23. Construct Binary Search Tree for the following data and give in order, preorder and post order tree traversal: 20, 30, 10, 5, 16, 21, 29, 45, 0, 15, 6
24. Write a C function to search element in a Binary Search Tree.
25. Define Height Balanced Tree. Built an AVL Tree for the following data: Sun, Mon, Tue, Wed, Thur.



1. Graphs

In computer science, a graph is a kind of data structure, that consist of a set of *nodes* (also called *vertices*) and a set of *edges* that establish connections between the nodes. To denote mathematically, graph $G = (V, E)$ consists of vertices, V , which are connected by edges, the elements of E . Formally, V is a set (usually finite) and E is a set consisting of two subsets of V .

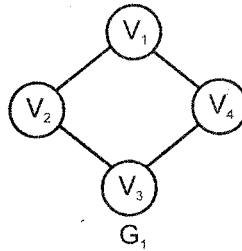
1.1 Definitions and Terminology

Graph

A graph G is a collection of two sets V and E . V is a finite non empty set of vertices (or nodes) and E is a finite non empty set of edges (or arcs) connecting a pair of vertices.

An edge is represented by two adjacent vertices G is represented as $G = (V, E)$

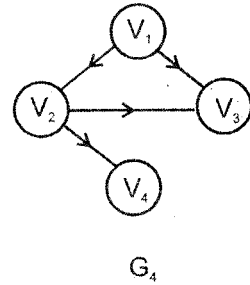
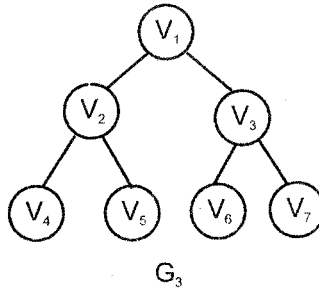
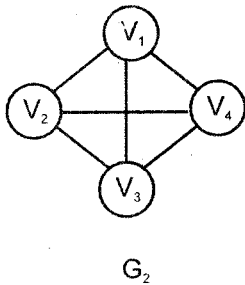
Example



$$G_1 = (V, E)$$

$$V = \{V_1, V_2, V_3, V_4\}$$

$$E = \{(V_1, V_2), (V_2, V_3), (V_3, V_4), (V_4, V_1)\}$$



Some examples of graphs: Graphs are two types, Undirected graph and Directed graph.

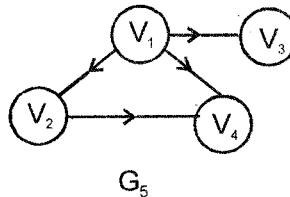
Types of Graphs

i. **Undirected Graph:** A graph is an undirected graph if the pairs of vertices that make up the edges are unordered pairs. i.e. an edge (V_i, V_j) is the same as (V_j, V_i) . The graph G_1 shown above is an undirected graph.

ii. **Directed Graph:** In a directed graph, each edge is represented by a pair of ordered vertices, i.e., an edge has a specific direction.

In such a case, edge $(V_i, V_j) \neq (V_j, V_i)$,

Example



Directed graph

1

Apr. 2015 – 2M

State the types of graphs.

2

Oct. 12, 14 – 2M

What is Graph? State its types.

1

Oct. 2015 – 4M

What are the different ways we can represent graph? Explain any one with an example.

$$G_5 = (V, E)$$

$$V = \{V_1, V_2, V_3, V_4\}$$

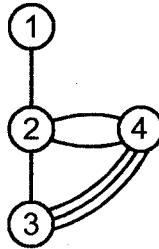
$$E = \{(V_1, V_2), (V_1, V_4), (V_2, V_4), (V_1, V_3)\}$$

For an edge (V_i, V_j) in a directed graph, vertex V_i is called the tail and V_j is the head of the edge. V_i is adjacent to V_j and V_j is adjacent from V_i .

- iii. **Complete Graph:** If an undirected graph has n vertices, the maximum number of edges it can have, $nC_2 = n(n-1)/2$. If an undirected graph G has ' n ' vertices and nC_2 edges, it is called a complete graph.

If the graph G is directed and has n vertices, G is complete if it has $n(n-1)$ edges.

- iv. **Multigraph:** A multigraph is a graph in which the set of edges may have multiple occurrences of the same edge. Note that it is not a graph.



Example of a multigraph that is not a graph

- **Degree of Vertex:** The degree of a vertex in an undirected graph is the number of edges incident to that vertex.

In the undirected graph G_1 , the degree of each vertex = 2.

- **Indegree of a Vertex:** If G is a directed graph, the indegree of a vertex is the number of edges for which it is head i.e. the number of edges coming to it.

Example

In graph G_5 , $\text{indegree}(V_4) = 2$

$\text{indegree}(V_1) = 0$

A node whose indegree is 0, is called a *source node*.

- **Outdegree of a Vertex:** If G is directed graph, the out degree of a vertex is the number of edges for which it is the tail i.e. the number of edges going out of it.

Example

$\text{outdegree}(V_1) = 3$

$\text{outdegree}(V_2) = 1$

Oct. 2014 - 2M

How to calculate
indegree and outdegree
of nodes on graph?

1

1

Oct. 2010 – 4M

Define the following terms:

- i. Cycle in a Graph
- ii. Adjacent Vertices
- iii. Indegree of Graph

A node whose outdegree is 0, is called a *sink node*.

- **Adjacent Vertices:** If (V_i, V_j) is an edge in G , then we say that V_i and V_j are adjacent and the edge (V_i, V_j) is *incident* on V_i and V_j .
- **Path:** A path from vertex V_p to V_q exists if there exists vertices $V_{i_1}, V_{i_2}, \dots, V_{i_n}$ such that there exist edges $(V_p, V_{i_1}), (V_{i_1}, V_{i_2}), \dots, (V_{i_n}, V_q)$

Length of a Path: The length of a path is the number of edges on it.

Linear Path: A linear path is a path whose first and last vertices are distinct.

- **Cycle:** A cycle is a path whose first and last vertices are the same.

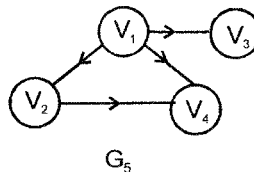
Example $V_1 V_2 V_3 V_4$ is a cycle in G_4 . A graph with no cycles is called an **acyclic** graph. A directed acyclic graph is called **dag**.

- v. **Connected Graph:** Two vertices V_i and V_j are said to be connected if there is a path in G from V_i to V_j .

Strongly Connected Graph: A directed graph G is said to be strongly connected if for every pair of distinct vertices V_i, V_j , there is a directed path from V_i to V_j and also from V_j to V_i .

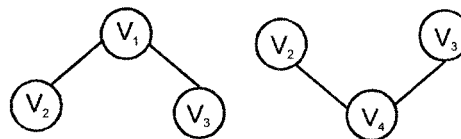
Weakly Connected Graph: A directed graph G is said to be weakly connected there exists atleast one set of distinct vertices V_i, V_j , such that there is a directed path from V_i to V_j but no path from V_j to V_i .

Example: The following is a weakly connected graph because there is a path from V_1 to V_4 but none from V_4 to V_1 .



- vi. **Subgraph:** A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Example: The subgraphs of G_1 are:



Subgraphs of G_1

- vii. **Forest:** A forest is defined as an acyclic graph in which every node has one or no predecessors.
- viii. **Spanning Tree:** When a graph G is connected, a traversal method visits all its vertices. In this case the edges of G are partitioned into two sets.

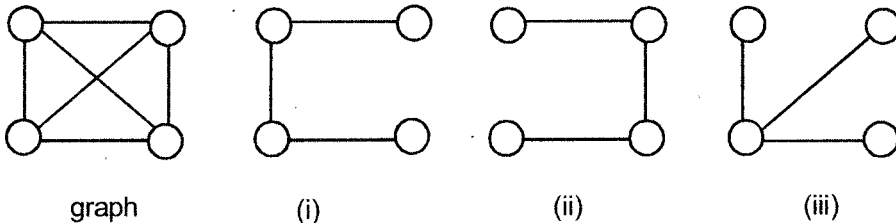
T for the edges traversed.

B (Back edges) which were not traversed.

The edges in T form a tree which connects all vertices of graph G . Such a tree is called a spanning tree.

A spanning tree consists of the minimum number of edges to connect all the vertices.

Example



Spanning trees

A graph and its spanning trees

- ix. **Minimal Cost Spanning Tree:** The spanning tree having the minimum sum of weights of edges is called minimum cost spanning tree.

These weights may represent the lengths, distances, cost, etc.

Such trees are widely used in practical applications such as network of road lines between cities, etc.

- x. **Spanning Forest:** A spanning forest of a graph $G = (V, E)$ is a collection of vertex disjoint trees $T_i = (V_i, E_i)$, $1 \leq i \leq k$ such that $V = \cup V_i$ for all $1 \leq i \leq k$ and $E_i \subseteq E(G)$, $1 \leq i \leq k$

1

Apr. 2011 – 4M

Define the following terms:

- i. Spanning Tree
- ii. Cycle in a Graph
- iii. Adjacent Vertices
- iv. In degree of Graph

1.2 Representation of Graph

The graph can be represented with several forms as *Adjacency Matrix* and *Adjacency list*.

Adjacency Matrix Implementation

Definition: Adjacency matrix is a representation of both *directed* and *undirected* graph using two dimensional array $n \times n$ elements where n is the number of vertices.

A $|V| \times |V|$ matrix of 0's and 1's.

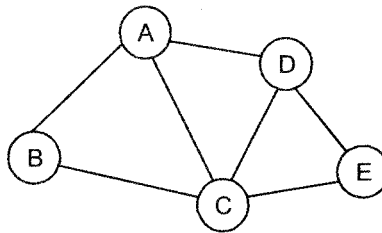
This indicates value stored at any location is either 0 or 1 where 1 represents a connection or an edge and 0 indicates no edge between vertices. The position is indicated by $[u, v]$ where $u \in V, v \in V$.

Adjacency Matrix

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

1

Apr. 2015 – 4M
What is graph? Explain its representation techniques in details.



Non-directed graph

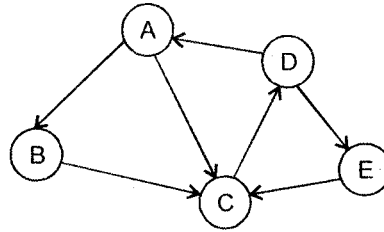
For a non-directed graph there will always be having symmetry along the top left to bottom right diagonal. The diagonal will always be filled with zero's.

In adjacency matrix it is easy to calculate degree of vertex, where degree is nothing but number of vertices connected to it. As in adjacency matrix 1 is placed at particular position i.e. u, v , if edge is present between the vertices u, v . Hence degree of vertex v can be easily calculated as sum of all the 1s present at the row that is represented by vertex u .

For example degree of vertex D is 3.

Adjacency Matrix

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



Directed graph

For a directed graph there will not symmetry along the top left to bottom right diagonal. Also the diagonal will always be filled with zero's.

In directed graph, the vertex is having two types of degrees i.e. *indegree* and *outdegree*.

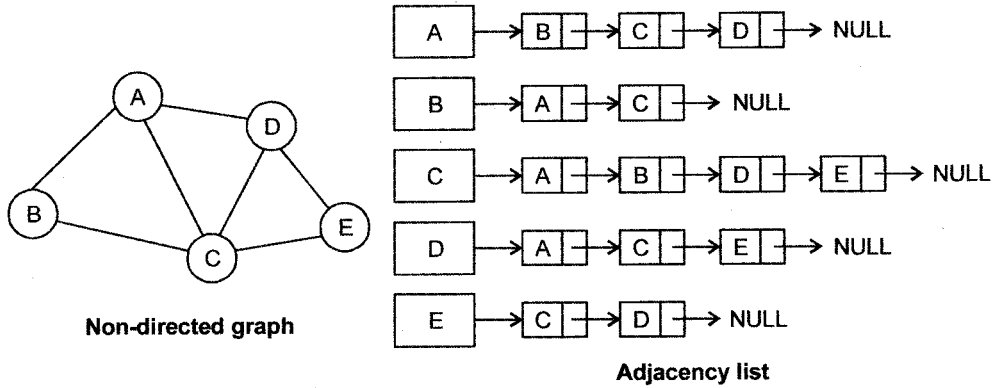
1. **Indegree:** Indegree can be defined as number of edges that coming in at vertex from other vertices. Hence indegree of vertex u is nothing but sum of 1s in the column that represents vertex u . For example: indegree of vertex D is 1.
2. **Outdegree:** Outdegree can be defined as number of edges that going out from vertex to other vertices. Hence, outdegree of vertex u is nothing but sum of 1s in the row that represents vertex u . For example: outdegree of vertex D is 2.

1.3 Adjacency List Implementation

Adjacency List Implementation (non-directed graph)

Definition: An adjacency list is also a representation of an undirected and directed graph with n vertices using an array of n lists of vertices. List i contains vertex j if there is an edge from vertex i to vertex j . An undirected graph may be represented by having vertex j in the list for vertex i and vertex i in the list for vertex j .

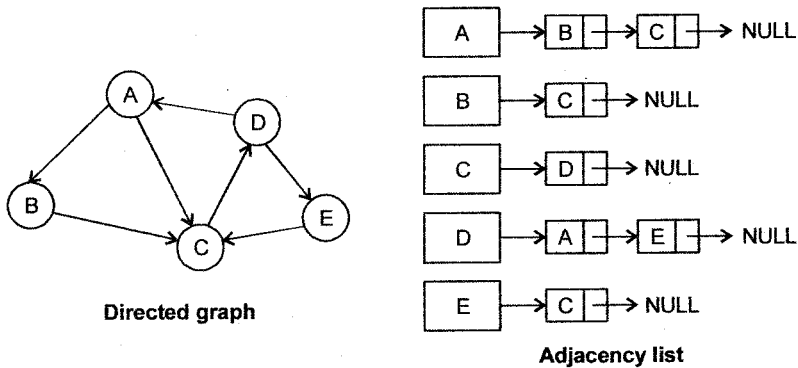
The Adjacency list implementation typically uses less space than the adjacency matrix implementation. It is exactly like hashing with chaining where we have a list (array) of vertices, each of which stores a linked list of all of its neighbors



In non directed graph using adjacency list it is easy to calculate degree of vertex which is nothing but number of nodes present in its linked list.

For example, degree of vertex C is 4.

Adjacency List Implementation (directed graph)



This method is very good for problems that involve traversing a graph. Storage = $|V|$ header cells + $2|E|$ linked list cells (since each edge in an undirected graph is counted by both vertices that it connects). In a directed graph the $2|E|$ is replaced by $|E|$. For a graph it is much better to use the adjacency list implementation with the storage $(|V| + |E|)$ versus storage of $|V|^2$ for the adjacency matrix implementation.

In directed graph using adjacency list indegree for vertex V can be calculated as the number of times, the vertex V presents in all the linked list.

For example, degree of vertex C is 3.

Similarly outdegree of vertex V can be calculated as the number of nodes present in the linked list of vertex V .

For example, outdegree of vertex C is 4.

Comparison with other Data Structures

Graph data structures are non-hierarchical and therefore suitable for data sets where the individual elements are interconnected in complex ways. *For example*, a computer network can be modeled with a graph. Hierarchical data sets can be represented by a binary or non binary tree. It is worth mentioning, however, that trees can be seen as a special form of graph.

2. Shortest Path Problem

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. *For example*, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, the following algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols.

Algorithm

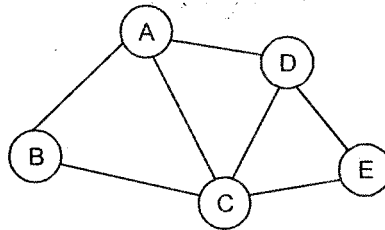
Let's call the node we are starting with an initial node. Let a distance of a node X be the distance from the initial node to it. This algorithm will assign some initial distance values and will try to improve them step-by-step.

- i. Assign to every node a distance value. Set it to zero for our initial node and to infinity for all other nodes.
- ii. Mark all nodes as unvisited. Set initial node as current.
- iii. For current node, consider all its unvisited neighbours and calculate their distance (from the initial node). *For example*, if current node (A) has distance of 6, and an edge connecting it with another node (B) is 2, the distance to B through A will be $6+2=8$. If this distance is less

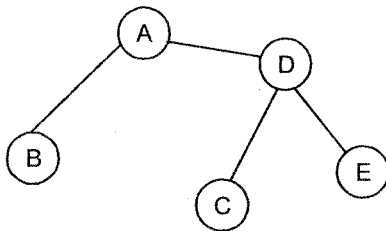
- than the previously recorded distance (infinity in the beginning, zero for the initial node), overwrite the distance.
- iv. When we are done considering all neighbours of the current node, mark it as visited. A visited node will not be checked ever again; its distance recorded now is final and minimal.
 - v. Set the unvisited node with the smallest distance (from the initial node) as the next 'current node' and continue from step 3.

3. Spanning Tree

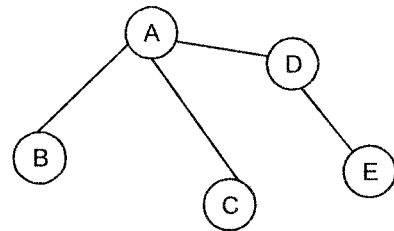
In the mathematical field of graph theory, a spanning tree T of a connected, undirected graph G is a tree composed of all the vertices and some of the edges of G . Informally, a spanning tree of G is a selection of edges of G that form a tree, spanning every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of G must belong to T .



Graph



Spanning tree - 1



Spanning tree - 2

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

In certain fields of graph theory it is often useful to find a minimum spanning tree of a weighted graph. Other optimization problems on spanning trees have also been studied, including the maximum spanning tree, the minimum tree that spans at least k vertices, the minimum spanning tree with at most k edges per vertex (MDST), the spanning tree with the largest number of leaves (closely related to the smallest connected dominating set), the spanning tree with the fewest leaves

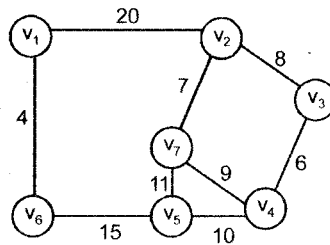
(closely related to the Hamiltonian path problem), the minimum diameter spanning tree, and the minimum dilation spanning tree.

We shall be studying two algorithms for finding the Minimum Cost Spanning tree.

- i. Prim's Algorithm:** This algorithm builds the minimum cost spanning tree edge by edge. The edge to be included in the tree T is chosen according to some optimization criteria. The criterion used here, is to select the edge (u, v) having the smallest cost such that (u, v) is not already in the tree and $T \cup \{(u,v)\}$ is also a tree.

While including (u, v) we must ensure that it does not form a cycle. The edge addition is repeated till T contains $n-1$ edges.

Let us apply this method to the following graph to obtain its minimum cost spanning tree.



Graph for Prim's Algorithm

Step	Set of vertices	Edges which have exactly one end belonging to the partial tree	Select	Spanning Tree
1.	V_1	$(V_1, V_2) (V_1, V_6)$	(V_1, V_6)	
2.	V_1, V_6	$(V_1, V_2) (V_5, V_6)$	(V_5, V_6)	

3.	V_1, V_5, V_6	$(V_1, V_2) (V_5, V_7) (V_4, V_5)$	(V_4, V_5)	
4.	V_1, V_4, V_5, V_6	$(V_1, V_2) (V_5, V_7) (V_4, V_7) (V_3, V_4)$	(V_3, V_4)	
5.	V_1, V_3, V_4, V_5, V_6	$(V_1, V_2) (V_2, V_3) (V_4, V_7) (V_5, V_7)$	(V_2, V_3)	
6.	$V_1, V_2, V_3, V_4, V_5, V_6$	$(V_1, V_2) (V_1, V_7) (V_4, V_7) (V_5, V_7)$	(V_2, V_7)	
7.	(V_4, V_7)			Forms a cycle
8.	(V_5, V_7)			Forms a cycle
9.	(V_1, V_2)			Forms a cycle

```

Algorithm PRIMS(E, Cost, n)
{
  /* E is the set of edges in G, Cost is the adjacency cost
  matrix, n are the number of vertices */
  T={0}; /* Start with vertex 0 and no edges */
  while(T contains less than n-1 edges)
  {
    select(u,v) from E such that cost [u,v] is minimum and u∈T and v∉T
    if(u,v) is found then
      Add v to T
    else
      break;
  }
  if(T contains fewer than n-1 edges print - No spanning tree)
}

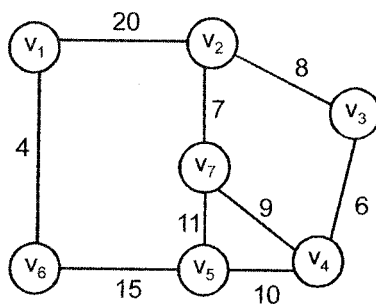
```

- ii. **Kruskal's Algorithm:** In the Prim's algorithm studied earlier, at any stage, the set of selected edges must form a tree.

In the Kruskal's algorithm, however, the set of edges may not be a tree at all stages. The set will generally be a forest and can be completed into a tree iff there are no cycles in the set. The edges will be considered one by one such that it has minimum cost among the remaining edges and does not form a cycle.

The method is simple. The spanning tree T is constructed edge by edge. We select the edges one-by-one. We select an unvisited edge having smallest cost and add it to the partially complete spanning tree. If the edge forms a cycle, it is not considered. When $n-1$ edges have been added to the spanning tree, the process stops.

Example,



Graph for Kruskal's Algorithm

Step	Consider	Spanning Tree
1	(V_1, V_6)	
2	(V_3, V_4)	
3	(V_2, V_7)	
4	(V_2, V_3)	
5	(V_4, V_7)	Forms a Cycle, Reject
6	(V_4, V_5)	
7	(V_5, V_7)	Forms a Cycle, Reject
8	(V_5, V_6)	

The Algorithm can be written as follows:

```
Algorithm Kruskal(E, cost, n)
{
  T = 0; /* start with a Tree having no edges */
  While(T contains less than n-1 edges and E is
        not empty)
  {
    choose edge(u,v) from E such that cost[u,v] is minimum
    delete(u,v) from E
    if(u,v) does not create a cycle in T
      add(u,v) to T
    else
      discard(u,v)
  }
  if(T contains fewer than n-1 edges)
    Print "no spanning tree"
}
```

Oct., Apr. 2010 – 4M

2
Explain Kruskal's Algorithm for minimum spanning tree.

Comparing Prim's and Kruskal's Algorithm

Both produce identical trees when edge weights are distinct. When G is connected, Kruskal's cannot produce a forest. When no weights are equal, then random edge selection cannot occur.

4. Traversal of Graphs

The traversal of graph means to visit the vertices in some systematic order. We have studied with various traversal methods for trees:

- i. **preorder:** Visit each node before its children.
- ii. **postorder:** Visit each node after its children.
- iii. **inorder (for binary trees only):** Visit left subtree, node, right subtree.

There are two other traversals: Breadth First Search (BFS) and Depth First Search (DFS). Both of these construct spanning trees with certain properties useful in other graph algorithms. These methods can be used for both undirected graphs, but they are both also very useful for directed graphs.

4.1 Depth-First Search (DFS)

Depth-First Search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Definition: Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

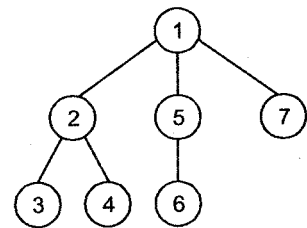
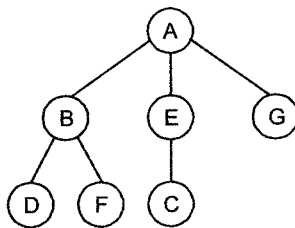
Space complexity of DFS is much lower than BFS (breadth-first search). It also lends itself much better to heuristic methods of choosing a likely-looking branch. Time complexity of both algorithms are proportional to the number of vertices plus the number of edges in the graphs they traverse ($O(|V| + |E|)$).

When searching large graphs that cannot be fully contained in memory, DFS suffers from non-termination when the length of a path in the search tree is infinite. The simple solution of "remember which nodes I have already seen" doesn't always work because there can be insufficient memory. This can be solved by maintaining an increasing limit on the depth of the tree, which is called iterative deepening depth-first search.

For the following graph:

2

Oct, Apr.10 – 4M
Explain Depth First Search with an example.



A depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G, which is sequentially shown in neighboring figure in sequence 1 to 7.

- i. **Output of a depth-first search:** The most natural result of a depth first search of a graph (if it is considered as a function rather than a procedure) is a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: *forward edges*, which point from a node of the tree to one of its descendants, *back edges*, which point from a node to one of its ancestors, and *cross edges*,

which do neither. Sometimes tree edges, edges which belong to the spanning tree itself, are classified separately from *forward edges*. It can be shown that if the graph is undirected then all of its edges are tree edges or back edges.

- ii. **Vertex Orderings:** It is also possible to use the depth-first search to linearly order the vertices (or nodes) of the original graph (or tree). There are three common ways of doing this:
- A **preordering** is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search. A preordering of an expression tree is the expression in Polish notation.
 - A **postordering** is a list of the vertices in the order that they were last visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.
 - A **reverse postordering** is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. When searching a tree, reverse postordering is the same as preordering, but in general they are different when searching a graph.
For example, when searching the directed graph.

Beginning at node A, one visits the nodes in sequence, to produce lists either A B D B A C A, or A C D C A B A (depending upon the algorithm chooses to visit B or C first). Note that repeat visits in the form of backtracking to a node, to check if it has still unvisited neighbors, are included here (even if it is found to have none). Thus the possible preorderings are A B D C and A C D B (order by node's leftmost occurrence in above list), while the possible reverse postorderings are A C B D and A B C D (order by node's rightmost occurrence in above list). Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flow.

Applications

Here are some algorithms where DFS is used:

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding strongly connected components.
- Solving puzzles with only one solution, such as mazes.

4.2 Breadth-First Search (BFS)

The breadth first search is like shortest path algorithm, but with every edge having the same length. However it is a lot simpler and doesn't need any data structures. In this method following things are used.

3

Oct. 15, Apr. 15 – 4M
Explain BFS with an example.

Apr. 2010 – 4M
Explain Breadth First Search with example.

1. A tree (the breadth first search tree),
2. A list of nodes to be added to the tree,
3. Markings (Boolean variables) on the vertices to tell whether they are in the tree or list.

In graph theory, Breadth-First Search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

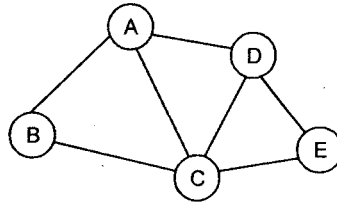
How it works

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combinations of sequence by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic.

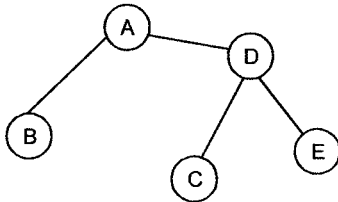
From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called 'open' and then once examined are placed in the container 'closed'.

Algorithm (informal)

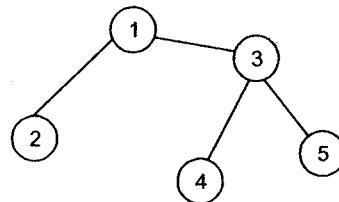
- i. Enqueue the root node.
- ii. Dequeue a node and examine it.
 - a. If the element sought is found in this node, quit the search and return a result.
 - b. Otherwise enqueue any successors (the direct child nodes) that have not yet been examined.
- iii. If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
- iv. Repeat from Step 2.



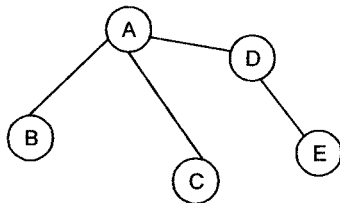
Graph



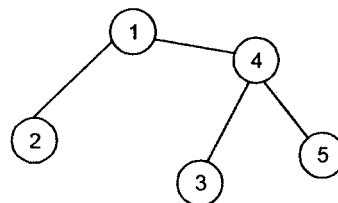
Spanning tree - 1



BFS for spanning tree - 1



Spanning tree - 2



BFS for spanning tree - 2

In the above *example*, two possible spanning trees are generated from given graph and according to the algorithm; the nodes are traversed starting from the root node to its child node.

Applications of BFS

Breadth-first search can be used to solve many problems in graph theory, *for example*:

- i. Finding all connected components in a graph.
- ii. Finding all nodes within one connected component.
- iii. Copying Collection, Cheney's algorithm.
- iv. Finding the shortest path between two nodes u and v (in an un-weighted graph).
- v. Finding the shortest path between two nodes u and v (in a weighted graph: see talk page).
- vi. Testing a graph for bipartiteness.
- vii. (Reverse) Cuthill–McKee mesh numbering.

Relation between BFS and DFS

BFS and DFS are very closely related to each other. (In fact in class I tried to describe a search in which I modified the 'add to end of list' line in the BFS pseudocode to 'add to start of list' but the resulting traversal algorithm was not the same as DFS.)

1

Apr. 2012 – 4M

Differentiate between DFS and BFS.

Both of these search algorithms now keep a list of edges to explore; the only difference between the two is that, while both algorithms adds items to the end of L, BFS removes them from the beginning, which results in maintaining the list as a queue, while DFS removes them from the end, maintaining the list as a stack.

5. Applications of Graphs

Some of the applications of graphs are:

- Graphs are used to represent Mazes (using stacks)
- Graphs are used to diagrammatically represent Networks (computer, cities)
- Graph is used to represent geographic databases of Maps.
- Graph is used to prepare Graphics: Geometrical Objects
- Graph is used to represent Neighborhood graphs and Voronoi diagrams.

Solved Examples

1

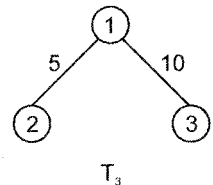
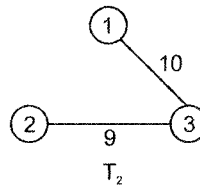
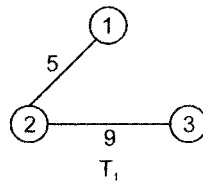
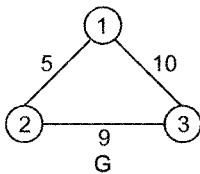
Oct. 2014 – 4M

1. Explain minimal spanning tree with an example.

Solution

The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a graph $G = (V, E)$ is called minimal cost spanning tree or simply minimal spanning tree of G if its cost is minimum.

Example,



G : A weighted graph

T_1 : A spanning tree of G with cost $5 + 9 = 14$

T_2 : A spanning tree of G with cost $10 + 9 = 19$

T_3 : A spanning tree of G with cost $5 + 10 = 15$

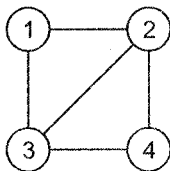
$\therefore T_1$ with cost 14 is the minimal cost spanning tree of the graph G .



PU Questions

2 Marks

1. What is graph? States its types.
2. State the types of graphs.
3. How to calculate indegree and outdegree of nodes on graph?
4. Give the Adjacency Matrix and Adjacency List for the following graph:



5. List the different methods for graph representation in memory.

4 Marks

1. Write a function in "C" to traverse a graph using Depth first search.
2. What are the different ways we can represent graph? Explain any one with an example.
3. Explain BFS with an example.
4. What is graph? Explain its representation techniques in details.
5. Write a function to read adjacency matrix and find the node with maximum indegrees.
6. Write a function in "C" to traverse a graph using Breadth First Search technique.
7. Explain minimal spanning tree with an example.

[Apr.2015 – 2M]

[Oct.2014 – 2M]

[Apr.2012 – 2M]

[Oct.11, 09 – 2M]

[Oct.2015 – 4M]

[Oct.2015 – 4M]

[Oct,15Apr.15 – 4M]

[Apr.215 – 4M]

[Oct.2014 – 4M]

[Oct.2014 – 4M]

[Oct.2014 – 4M]

[Oct.15,12 – 2M]

[Oct.2012 – 4M]

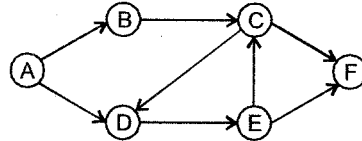
8. Write a function to calculate Indegree and Outdegree of each Node in the Graph.

[Apr.2012 – 4M]

9. Differentiate between DFS and BFS.

[Oct.2011 – 4M]

10. What is graph? Traverse the following graph using DFS (Start A).

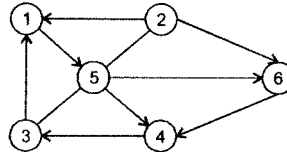
**[Apr.2011 – 4M]**

11. Define the following terms:

- i. Spanning Tree
- ii. Cycle in a Graph
- iii. Adjacent Vertices
- iv. In degree of Graph

[Apr.2011 – 4M]

12. Traverse following graph using BFS, Where starting vertex is 2.

**[Oct.2010 – 4M]**

13. Explain Depth First Search with an example.

[Oct., Apr. 10, – 4M]

14. Explain Kruskal's Algorithm for minimum spanning tree.

[Oct.2010 – 4M]

15. Define the following terms:

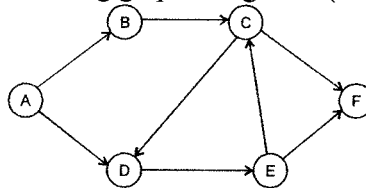
- i. Cycle in a Graph
- ii. Adjacent Vertices
- iii. Indegree of Graph

[Apr.2010 – 4M]

16. Explain Breadth First Search with example.

[Oct.2009 – 4M]

17. Traverse the following graph using DFS (Start A):

**[Oct.2009 – 4M]**

18. Define:

- i. Degree of Graph
- ii. Cycle in a Graph
- iii. Weighted Graph

Suggestive Readings:

1. Birkhanser-Boston, An Introduction to Data Structures and Algorithms, Springer-New York
2. Seymour Lipschutz, "Data Structure", Tata McGraw Hill.
3. Horowitz, Sahni & Anderson-Freed, "Fundamentals of Data Structure in C", Orient Longman.
4. Trembley, J.P. And Sorenson P.G., "An Introduction to Data Structures with Applications", McGraw- Hill International Student Edition, New York.
5. Yedidyan Langsam, Moshe J. Augenstein and Aaron M. Tenenbaum, "Data Structures using C", Prentice Hall of India Pvt. Ltd., New Delhi.
6. Mark Allen Weiss, "Data structures and Algorithm Analysis in C", Addison - Wesley (An Imprint of Pearson Education), Mexico City, Prentice -Hall of India Pvt. Ltd., New Delhi.
7. Rajni Jindal, Data structure using C, Umesh Publication
8. HorowitzE, Fundamental of data structure, Galgotia Publications