

# DATA STRUCTURE

(DEC 2019)

**Q.1**

**a) Define data structure. Differentiate linear and Non-linear data structure with example. (5)**

→ Data Structure

A **data structure** is a specialized format for organizing, processing, retrieving and storing **data**.

Parameters	Linear	Non-Linear
Basic	The data items are arranged in an orderly manner where the elements are attached adjacently.	It arranges the data in a sorted order and there exists a relationship between the data elements.
Traversing of the data	The data elements can be accessed in one time (single run).	Traversing of data elements in one go is not possible.
Ease of implementation	Simpler	Complex
Levels involved	Single level	Multiple level
Memory utilization	Ineffective	Effective
Examples	Array, queue, stack, linked list, etc.	Tree and graph.

**b) Write C function to implement insertion sort. (5)**

→ Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

**Algorithm**

```
// Sort an arr[] of size n
```

```
insertionSort(arr, n)
```

```
Loop from i = 1 to n-1.
```

```
Pick element arr[i] and insert it into sorted sequence arr[0...i-1]
```

### -C function

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

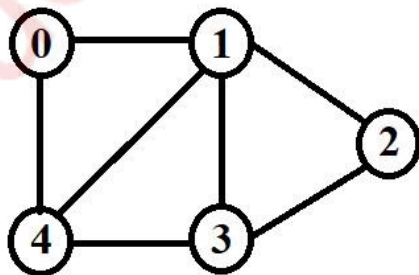
        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

c) What are the different ways to represent graphs in memory. (5)

→ Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form  $(u, v)$  called as edge. The pair is ordered because  $(u, v)$  is not same as  $(v, u)$  in case of a directed graph(digraph). The pair of the form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.

Following is an example of undirected graph with 5 vertices.



There are two most commonly used representations of a graph.

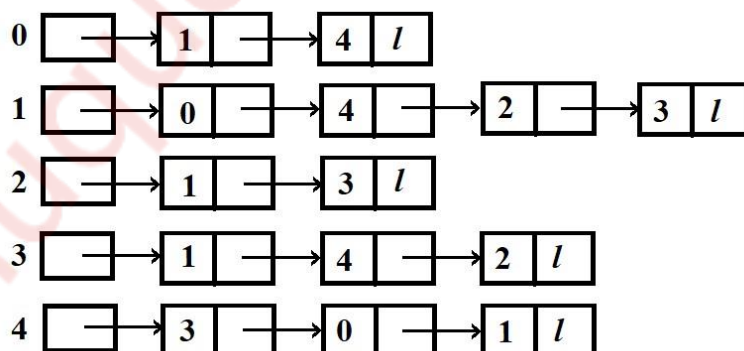
## 1. Adjacency Matrix

- Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph.
- Let the 2D array be  $adj[][]$ , a slot  $adj[i][j] = 1$  indicates that there is an edge from vertex  $i$  to vertex  $j$ .
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. - If  $adj[i][j] = w$ , then there is an edge from vertex  $i$  to vertex  $j$  with weight  $w$ .
- Following is adjacency matrix representation of the above graph.

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

## 2. Adjacency List

- An array of lists is used. Size of the array is equal to the number of vertices.
- Let the array be  $array[]$ . An entry  $array[i]$  represents the list of vertices adjacent to the  $i$ th vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be represented as lists of pairs.
- Following is adjacency list representation of the above graph.



d) What is expression tree? Derive an expression for  $(a+(b*c))/((d-c)*f)$ . (5)

→ Expression Tree: Compilers need to generate assembly code in which one operation is executed at a time and the result is retained for other operations.

- Therefore, all expression has to be broken down unambiguously into separate operations and put into their proper order.
- Hence, expression tree is useful which imposes an order on the execution of operations.
- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand
- Parentheses do not appear in expression trees, but their intent remains intact in tree representation. Construction of Expression Tree:

Now for constructing expression tree we use a stack. We loop through input expression and do following for every character.

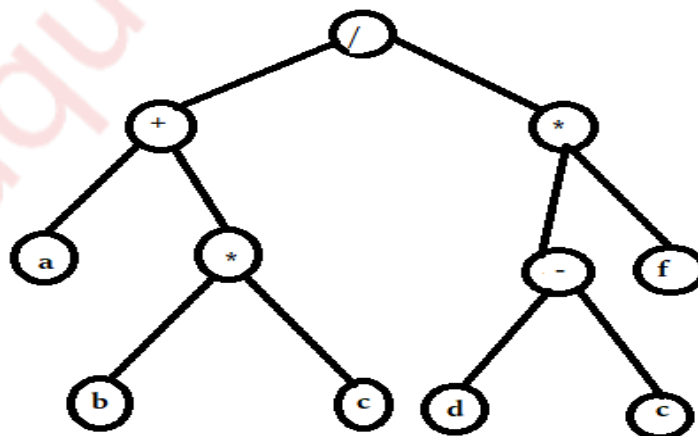
- 1) If character is operand push that into stack
- 2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Advantage:

1. Expression trees are using widely in LINQ to SQL, Entity Framework extensions where the runtime needs to interpret the expression in a different way (LINQ to SQL and EF: to create SQL, MVC: to determine the selected property or field).
2. Expression trees allow you to build code dynamically at runtime instead of statically typing it in the IDE and using a compiler.

- **Expression** **Tree**



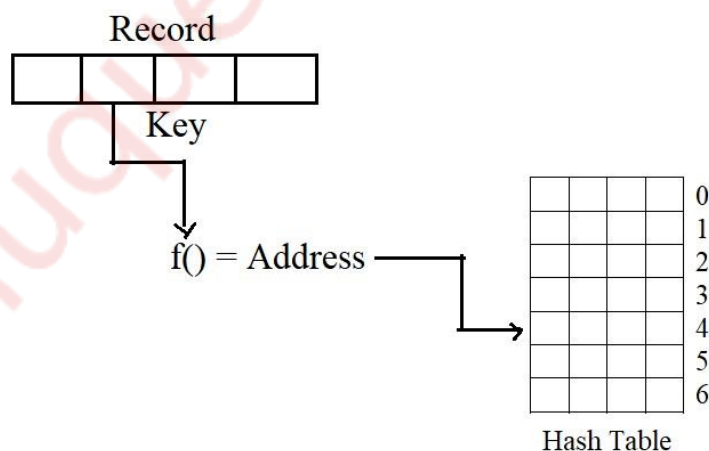
## Q.2

a) What is hashing? Hash the following data in table of size 10 using linear probing and quadratic probing. Also find the number of collisions.

63, 84, 94, 77, 53, 87, 23, 55, 10, 44 (10)

→ Hashing:

- Hashing is a technique by which updating or retrieving any entry can be achieved in constant time  $O(1)$ .
- In mathematics, a map is a relationship between two sets. A map  $M$  is a set of pairs, where each pair is in the form of (key, value). For a given key, its corresponding value can be found with the help of a function that maps keys to values. This function is known as the hash function.
- So, given a key  $k$  and a hash function  $h$ , we can compute the value/location of the value  $v$  by the formula  $v = h(k)$ .
- Usually the hash function is a division modulo operation, such as  $h(k) = k \text{ mod size}$ , where size is the size of the data structure that holds the values.
- Hashing is a way with the requirement of keeping data sorted.
- In best case time complexity is of constant order  $O(1)$  in worst case  $O(n)$
- Address or location of an element or record,  $x$ , is obtained by computing some arithmetic function  $f$ .  $f(\text{key})$  gives the address of  $x$  in the table.
- Table used for storing of records is known as hash table.
- Function  $f(\text{key})$  is known as hash function.



Mapping of records in hash table

→ Linear Probing

	Empty table	After 63	After 84	After 94	After 77	After 53	After 87	After 23	After 55	After 10	After 44
0										10	10
1									55	55	55
2											44
3		63	63	63	63	63	63	63	63	63	63
4			84	84	84	84	84	84	84	84	84
5				94	94	94	94	94	94	94	94
6						53	53	53	53	53	53
7					77	77	77	77	77	77	77
8							87	87	87	87	87
9								23	23	23	23

No. of collision= 6

→ Quadratic Probing

	Empty table	After 63	After 84	After 94	After 77	After 53	After 87	After 23	After 55	After 10	After 44	
0										10	10	
1									55	55	55	*
2											44	*
3		63	63	63	63	63	63	63	63	63	63	
4			84	84	84	84	84	84	84	84	84	
5				94	94	94	94	94	94	94	94	*
6						53	53	53	53	53	53	*
7					77	77	77	77	77	77	77	
8							87	87	87	87	87	*
9								23	23	23	23	*

**b) Write recursive function to perform preorder traversal of binary. (8)**

→ Recursive function

/\* Given a binary tree, print its nodes in preorder\*/

void printPreorder(struct Node\* node)

{

if (node == NULL)

return;

```

/* first print data of node */
cout << node->data << " ";

/* then recur on left subtree */
printPreorder(node->left);

/* now recur on right subtree */
printPreorder(node->right);
}

```

c) Given an array  $\text{int a[]}=\{23, 55, 63, 89, 45, 67, 85, 99\}$ . Calculate address of  $\text{a}[5]$  if base address is 5100. (2)

→

<b>Address</b>	5100	5104	5108	5112	5116	5120	5124	5128
<b>Elements</b>	23	55	63	89	45	67	85	99
<b>Array</b>	0	1	2	3	4	5	6	7

Address of  $A [ I ] = B + W * ( I - LB )$

Where, B = Base address 5100 (given)

W = Storage Size of one element stored in the array (in byte) = 4

I = Subscript of element whose address is to be found = 5 (given)

LB = Lower limit / Lower Bound of subscript, if not specified assume 0

Address of  $A [ 5 ] = 5100 + 4 * ( 5 - 0 )$

=  $5100 + 4 * 5$

=  $5100 + 20$

$A [ 5 ] = 5120$

One can verify it from table too  $A[5]$  has element 67 stored at address 5120.

### Q.3

a) Write a C program to convert infix expression to postfix expression. (10)

→ Program

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
stack[++top] = x;
}
char pop()
{
if(top == -1)
return -1;
else
return stack[top--];
}
int priority(char x)
{
if(x == '(')
return 0;
if(x == '+' || x == '-')
return 1;
if(x == '*' || x == '/')
return 2;
}
main()
{
char exp[20];
char *e, x;
printf("Enter the expression :: ");
scanf("%s",exp);
e = exp;
while(*e != '\0')
{
if(isalnum(*e))
printf("%c",*e);
else if(*e == '(')
push(*e);
else if(*e == ')')
{
```



```

while((x = pop()) != '(')
printf("%c", x);
}
else
{
while(priority(stack[top]) >= priority(*e))
printf("%c",pop());
push(*e);
}
e++;
}
while(top != -1)
{
printf("%c",pop());
}
}

```


**Output:**

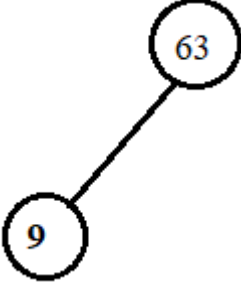
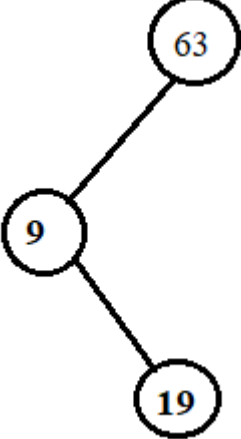
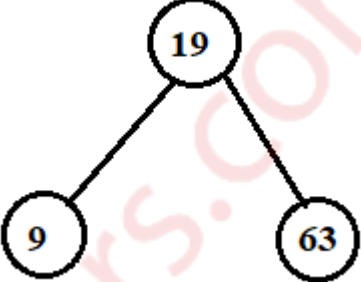
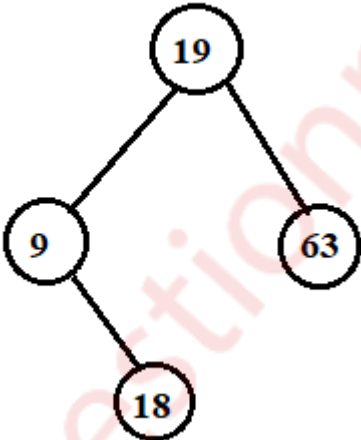
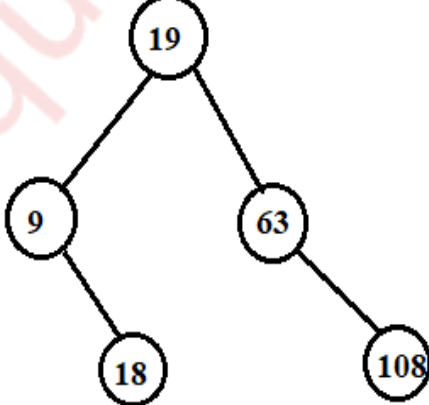
Enter the expression :: a+b\*c  
abc\*+

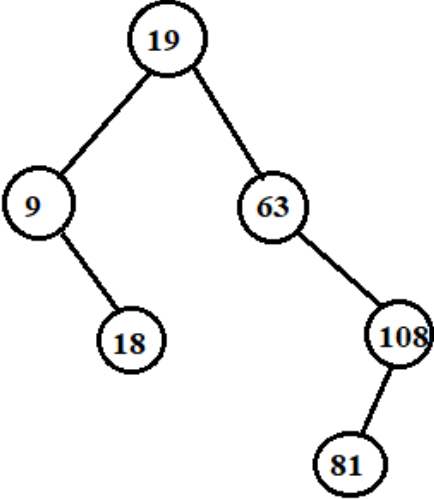
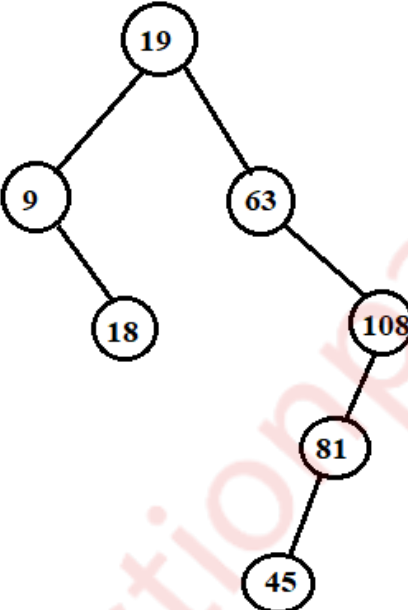
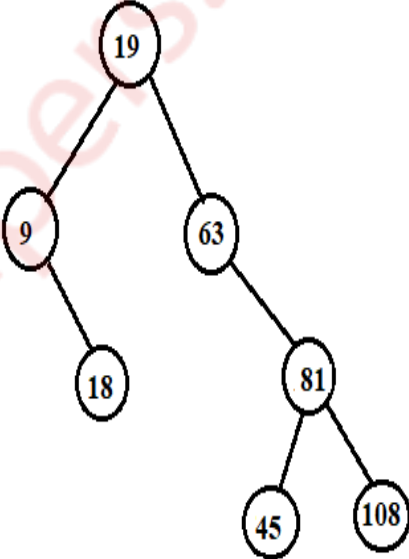
**b) Demonstrate step by step insertion of the following elements in an AVL tree.**

**63, 9, 19, 18, 108, 81, 45**

**(10)**

Sr no	Data to be insert	Tree after insertion	Tree after rotation
1	63		

2	9		
3	19		
4	18		
5	108		

6	81		
7	45		

**Q.4**

**a) Write C program to implement circular linked list that perform following functions.**

- insert a node at beginning
- insert a node at end
- Count the number of nodes
- Display the list

**(10)**

→ Program

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

```
#include<stdlib.h>
struct Node;
typedef struct Node * PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;
struct Node
{
int e;
Position next;
};
void Insert(int x, List l, Position p)
{
Position TmpCell;
TmpCell = (struct Node*) malloc(sizeof(struct Node));
if(TmpCell == NULL)
printf("Memory out of space\n");
else
{
TmpCell->e = x;
TmpCell->next = p->next;
p->next = TmpCell;
}
}
int isLast(Position p, List l)
{
return (p->next == l);
}
Position FindPrevious(int x, List l)
{
```

```
Position p = l;  
while(p->next != l && p->next->e != x)  
p = p->next;  
return p;  
}
```

```
Position Find(int x, List l)  
{  
Position p = l->next;  
while(p != l && p->e != x)  
p = p->next;  
return p;  
}
```

```
void Delete(int x, List l)  
{  
Position p, TmpCell;  
p = FindPrevious(x, l);  
if(!isLast(p, l))  
{  
TmpCell = p->next;  
p->next = TmpCell->next;  
free(TmpCell);  
}  
else  
printf("Element does not exist!!!\n");  
}
```

```
void Display(List l)  
{  
printf("The list element are :: ");  
Position p = l->next;
```

```

while(p != 1)
{
printf("%d -> ", p->e);
p = p->next;
}
}

void main()
{
int x, pos, ch, i;
List l, 11;
l = (struct Node *) malloc(sizeof(struct Node));
l->next = l;
List p = l;
printf("CIRCULAR LINKED LIST IMPLEMENTATION OF LIST ADT\n\n");
do
{
printf("\n\n1. INSERT\t 2. DELETE\t 3. FIND\t 4. PRINT\t 5. QUIT\n\nEnter
the choice :: ");
scanf("%d", &ch);
switch(ch)
{
case 1:
p = l;
printf("Enter the element to be inserted :: ");
scanf("%d",&x);
printf("Enter the position of the element :: ");
scanf("%d",&pos);
for(i = 1; i < pos; i++)
{

```

```
p = p->next;
}
Insert(x,l,p);
break;
case 2:
p = l;
printf("Enter the element to be deleted :: ");
scanf("%d",&x);
Delete(x,p);
break;
case 3:
p = l;
printf("Enter the element to be searched :: ");
scanf("%d",&x);
p = Find(x,p);
if(p == 1)
printf("Element does not exist!!!\n");
else
printf("Element exist!!!\n");
break;
case 4:
Display(l);
break;
}
}while(ch<5);
return 0;
}
```

**Output:**

```
CIRCULAR LINKED LIST
1. INSERT    2. DELETE    3. FIND    4. PRINT    5. QUIT
Enter the choice :: 1
Enter the element to be inserted :: 10
Enter the position of the element :: 1
1. INSERT    2. DELETE    3. FIND    4. PRINT    5. QUIT
Enter the choice :: 1
Enter the element to be inserted :: 20
Enter the position of the element :: 2
1. INSERT    2. DELETE    3. FIND    4. PRINT    5. QUIT
Enter the choice :: 1
Enter the element to be inserted :: 30
Enter the position of the element :: 3
1. INSERT    2. DELETE    3. FIND    4. PRINT    5. QUIT
Enter the choice :: 4
The list element are :: 10 -> 20 -> 30 ->
1. INSERT    2. DELETE    3. FIND    4. PRINT    5. QUIT
Enter the choice :: 5
```

**b) Given the frequency for the following symbol, compute the Huffman code for each symbol.**

Symbol	A	B	C	D	E	F
Frequency	9	12	5	45	16	13

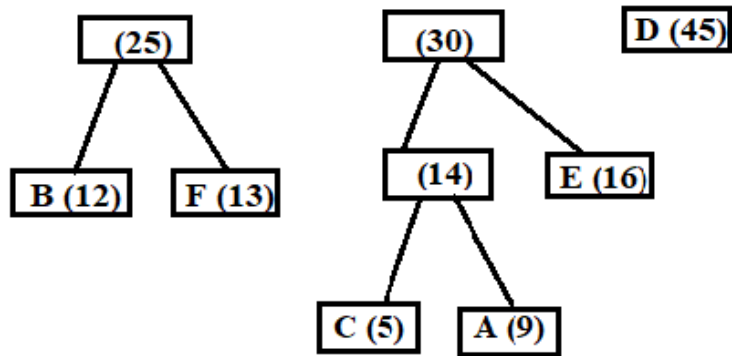
(10)

→ Huffman Code:

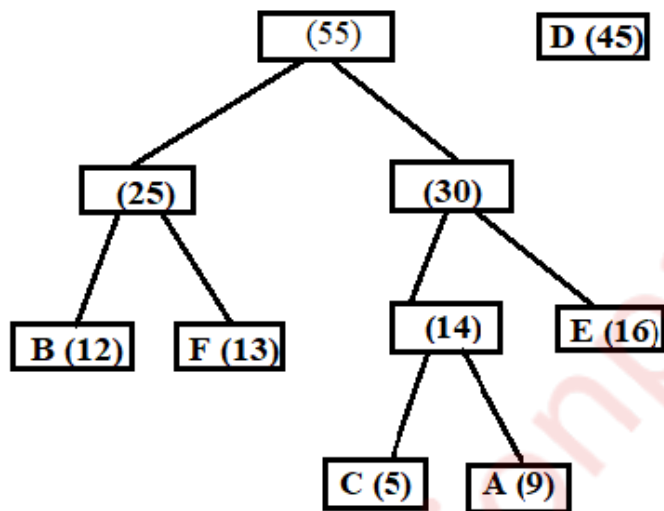
- Huffman code is an application of binary trees with minimum weighted external path length is to obtain an optimal set for messages  $M_1, M_2, \dots, M_n$
- Message is converted into a binary string
- Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.
- It use patterns of zeros and ones in communication system these are used at sending and receiving end.
- suppose there are  $n$  standard message  $M_1, M_2, \dots, M_n$ . Then the frequency of each message is considered, that is message with highest frequency is given priority for the encoding.



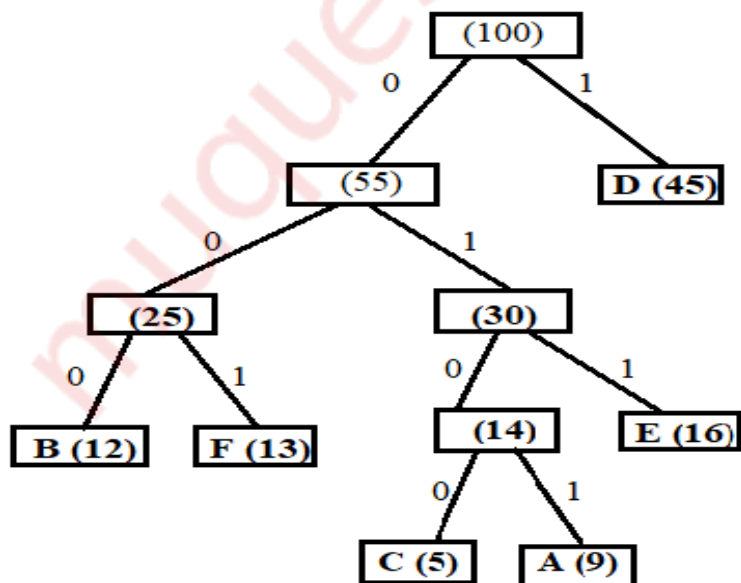




Merge two minimum frequency message



Merge two minimum frequency message



Huffman Code for each symbol

A= 0101

B= 000

C= 0100

D= 1

E= 011

F= 001

### Q.5

**a) Explain Double Ended Queue. Write a C program to implement Double Ended Queue. (10)**

Double Ended Queue

- It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end.

- However, no element can be added and deleted from the middle

- In a dequeue, two pointers are maintained, LEFT and RIGHT, which point to either end of the dequeue.

- They include,

#### **Input restricted dequeue**

– In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends. The following operations are possible in an input restricted dequeues.

i) Insertion of an element at the rear end and

ii) Deletion of an element from front end

iii) Deletion of an element from rear end

#### **Output restricted deque**

- In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

i) Deletion of an element at the front end

ii) Insertion of an element from rear end

iii) Insertion of an element from rear end

Operations on a dequeue

i. initialize(): Make the queue empty.

ii. empty(): Determine if queue is empty.

iii. full(): Determine if queue is full.

iv. enqueueF(): Insert an element at the front end of the queue.

v. enqueueR(): Insert an element at the rear end of the queue.

vi. dequeueF(): Delete the front end

vii. dequeueR(): Delete the rear end

viii. print(): print elements of the queue.

- There are various methods to implement a dequeue.

- Using a circular array

- Using a linked list

- Using a circular linked list

- Using a doubly linked list

- Using a doubly circular linked list

### **Program**

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

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

```
#define MAX 10
```

```
int deque[MAX];
```

```
int left=-1, right=-1;
```

```
void insert_right(void);
```

```
void insert_left(void);
```

```
void delete_right(void);
```

```
void delete_left(void);
```

```
void display(void);
```

```
int main()
```

```

{
int choice;
clrscr();
do
{
printf("\n1.Insert at right ");
printf("\n2.Insert at left ");
printf("\n3.Delete from right ");
printf("\n4.Delete from left ");
printf("\n5.Display ");
printf("\n6.Exit");
printf("\n\nEnter your choice ");
scanf("%d",&choice);
switch(choice)
{
case 1:
insert_right();
break;
case 2:
insert_left();
break;
case 3:
delete_right();
break;
case 4:
delete_left();
break;
case 5:
display();
break;
}
}while(choice!=6);
getch();
return 0;
}
void insert_right()
{
int val;
printf("\nEnter the value to be added ");
scanf("%d",&val);
if( (left==0 && right==MAX-1) || (left==right+1) )
{
printf("\nOVERFLOW");
}
}

```

```

}
if(left==-1)    //if queue is initially empty
{
left=0;
right=0;
}
else
{
if(right==MAX-1)
right=0;
else
right=right+1;
}
deque[right]=val;
}
void insert_left()
{
int val;
printf("\nEnter the value to be added ");
scanf("%d",&val);
if( (left==0 && right==MAX-1) || (left==right+1) )
{
printf("\nOVERFLOW");
}
if(left==-1)    //if queue is initially empty
{
left=0;
right=0;
}
else
{
if(left==0)
left=MAX-1;
else
left=left-1;
}
deque[left]=val;
}

```

//-----DELETE FROM RIGHT-----

```

void delete_right()
{

```

```

if(left==-1)
{
printf("\nUNDERFLOW");
return;
}
printf("\nThe deleted element is %d\n", deque[right]);
if(left==right) //Queue has only one element
{
left=-1;
right=-1;
}
else
{
if(right==0)
right=MAX-1;
else
right=right-1;
}
}

```

//-----DELETE FROM LEFT-----

```

void delete_left()
{
if(left==-1)
{
printf("\nUNDERFLOW");
return;
}
printf("\nThe deleted element is %d\n", deque[left]);
if(left==right) //Queue has only one element
{
left=-1;
right=-1;
}
else
{
if(left==MAX-1)
left=0;
else
left=left+1;
}
}

```

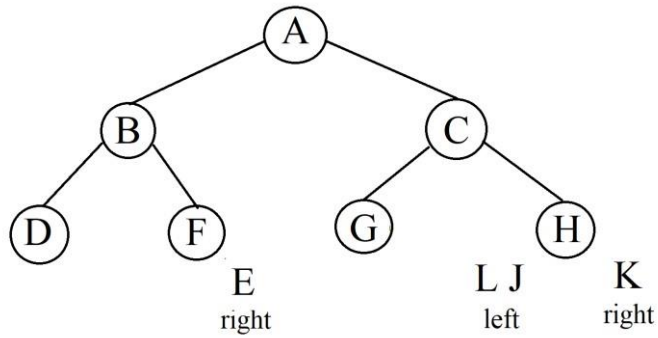
```
//-----DISPLAY-----
void display()
{
int front=left, rear=right;
if(front==-1)
{
printf("\nQueue is Empty\n");
return;
}
printf("\nThe elements in the queue are: ");
if(front<=rear)
{
while(front<=rear)
{
printf("%d\t",deque[front]);
front++;
}
}
else
{
while(front<=MAX-1)
{
printf("%d\t",deque[front]);
front++;
}
front=0;
while(front<=rear)
{
printf("%d\t",deque[front]);
front++;
}
}
printf("\n");
}
```

**Output:**

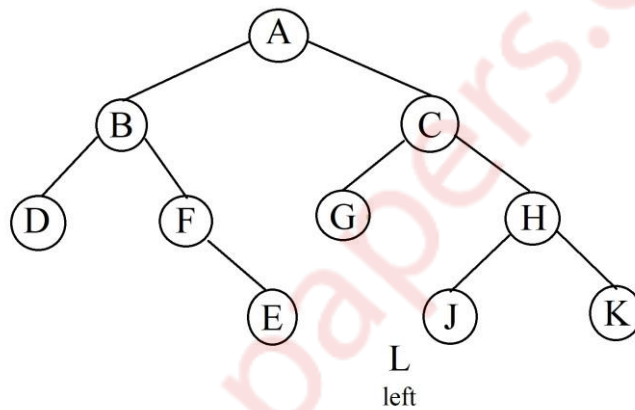
- 1.Insert at right
- 2.Insert at left
- 3.Delete from right
- 4.Delete from left



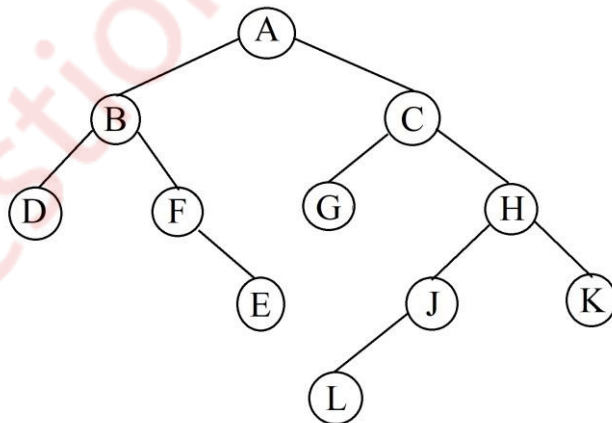




Step 4: As E is single it becomes child node of F. Traverse L J which is left element of node H in postorder as J comes last J becomes child node of H as K is single it becomes another child node.



Step 5: As L is single it becomes child node of J



Original Tree

**Q.6 Explain following with suitable example (any two)**

**(20)**

**I. B- tree and Splay tree**

➔ B- Tree

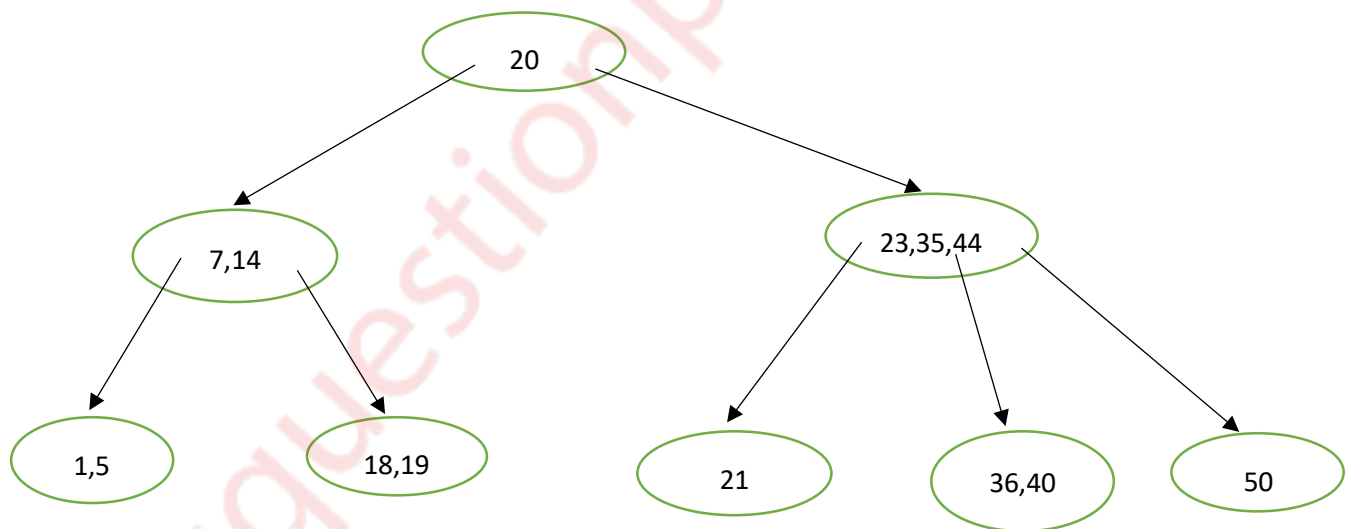
->A B-tree is a method of placing and locating files (called records or keys) in a database.

->The B-tree algorithm minimizes the number of times a medium must be accessed to locate a desired record, thereby speeding up the process.

->Properties:

1. All the leaf nodes must be at same level.
2. All nodes except root must have at least  $\lceil m/2 \rceil - 1$  keys and maximum of  $m-1$  keys.
3. All non leaf nodes except root (i.e. all internal nodes) must have at least  $m/2$  children.
4. If the root node is a non leaf node, then it must have at least 2 children.
5. A non leaf node with  $n-1$  keys must have  $n$  number of children.
6. All the key values within a node must be in Ascending Order.

->example:



### ➔ Splay Tree

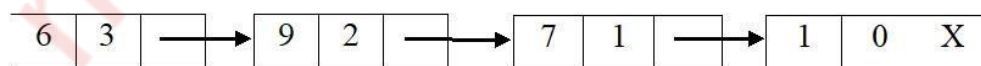
- A splay tree consists of a binary tree, with no additional fields.
- When a node in a splay tree is accessed, it is rotated or 'splayed' to the root, thereby changing the structure of the tree.
- Since the most frequently accessed node is always moved closer to the starting point of the search (or the root node), these nodes are therefore located faster.

- A simple idea behind it is that if an element is accessed, it is likely that it will be accessed again.
  - In a splay tree, operations such as insertion, search, and deletion are combined with one basic operation called splaying.
  - Splaying the tree for a particular node rearranges the tree to place that node at the root.
  - A technique to do this is to first perform a standard binary tree search for that node and then use rotations in a specific order to bring the node on top.
- Advantages and Disadvantages of Splay Trees**
- A splay tree gives good performance for search, insertion, and deletion operations.
  - This advantage centers on the fact that the splay tree is a self-balancing and a self-optimizing data structure.
  - Splay trees are considerably simpler to implement.
  - Splay trees minimize memory requirements as they do not store any bookkeeping data.
  - Unlike other types of self-balancing trees, splay trees provide good performance.

## II. Polynomial representation and addition using linked list.

### → Polynomial representation

- Let us see how a polynomial is represented in the memory using a linked list.
- Consider a polynomial  $6x^3 + 9x^2 + 7x + 1$ . Every individual term in a polynomial consists of two parts, a coefficient and a power.
- Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.
- Every term of a polynomial can be represented as a node of the linked list. Figure shows the linked representation of the terms of the above polynomial.



**Figure.** Linked representation of a polynomial

- Now that we know how polynomials are represented using nodes of a linked list.

- Example:

Input:

$$\text{1st number} = 5x^2 + 4x^1 + 2x^0$$

$$\text{2nd number} = 5x^1 + 5x^0$$

Output:

$$5x^2 + 9x^1 + 7x^0$$

Input:

$$\text{1st number} = 5x^3 + 4x^2 + 2x^0$$

$$\text{2nd number} = 5x^1 + 5x^0$$

Output:

$$5x^3 + 4x^2 + 5x^1 + 7x^0$$

### III. Topological Sorting

- Topological sort of a directed acyclic graph (DAG)  $G$  is defined as a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more number of topological sorts.
- A topological sort of a DAG  $G$  is an ordering of the vertices of  $G$  such that if  $G$  contains an edge  $(u, v)$ , then  $u$  appears before  $v$  in the ordering
- Note that topological sort is possible only on directed acyclic graphs that do not have any cycles.
- For a DAG that contains cycles, no linear ordering of its vertices is possible.
- In simple words, a topological ordering of a DAG  $G$  is an ordering of its vertices such that any directed path in  $G$  traverses the vertices in increasing order.
- Topological sorting is widely used in scheduling applications, jobs, or tasks. The jobs that have to be completed are represented by nodes, and there is an edge from node  $u$  to  $v$  if job  $u$  must be completed before job  $v$  can be started.
- A topological sort of such a graph gives an order in which the given jobs must be performed.
- The two main steps involved in the topological sort algorithm include:
  1. Selecting a node with zero in-degree

2. Deleting N from the graph along with its edges

- **Algorithm For topological Sorting**

Step 1: Find the in-degree  $INDEG(N)$  of every node in the graph

Step 2: Enqueue all the nodes with zero in-degree

Step 3: Repeat Steps 4 and 5 until the QUEUE is empty

Step 4: Remove the front node  $N$  of the QUEUE by setting

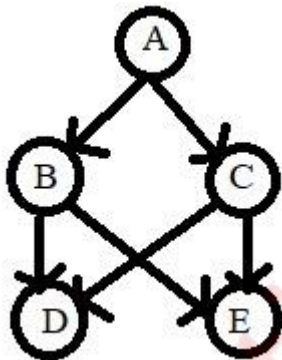
$FRONT = FRONT + 1$

Step 5: Repeat for each neighbour  $M$  of node  $N$ : a) Delete the edge from  $N$  to  $M$  by setting  $INDEG(M) = INDEG(M) - 1$  b) IF  $INDEG(M) = 0$ , then Enqueue  $M$ , that is, add  $M$  to the rear of the queue [END OF INNER

LOOP] [END OF LOOP]

Step 6: Exit •

Example:



Topological sort can be given as:

- A, B, C, D, E • A, B, C, E, D • A, C, B, D, E
- A, C, B, E, D