

DATA STRUCTURES

(CBCGS DEC 2017)

Q1. a) Explain ADT. List the Linear and Non-Linear data structures with example. (5)

Solution:

An ADT is a specified mathematical entity. It has a specific interface. A collection of signature of operations that can be involved on an instance.

It is a kind of abstraction which means the user does not know about the operations taking place at the backend.

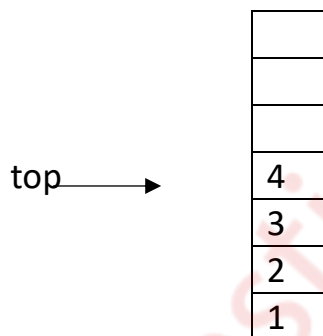
Linear data structures:

Elements are arranged in a linear fashion. Types of linear data structures are:

->Lists

0	1	2	3					n-1
---	---	---	---	--	--	--	--	-----

->stack



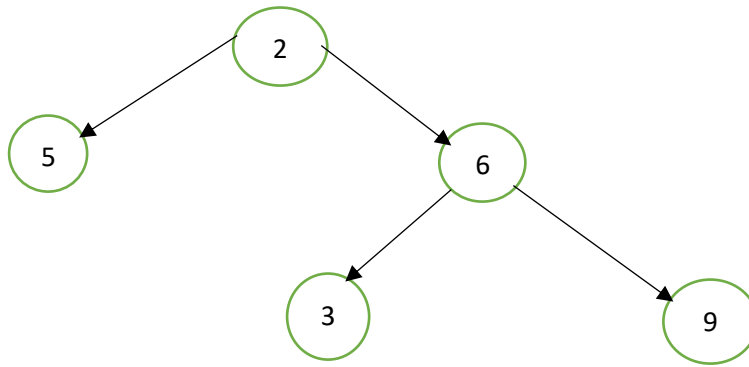
->queue

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---	-------

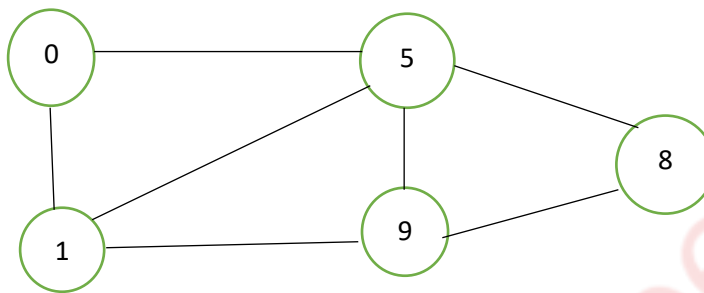
Non-linear data structures:

Every data element can have many successors and predecessors. Types of non-linear data structures:

->Tree



->Graph



->table

10	25	30	7	55
5	5	2	10	2
2	50	10	25	10

Q1. b) Explain B tree and B+ tree.

(5)

Solution:

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.

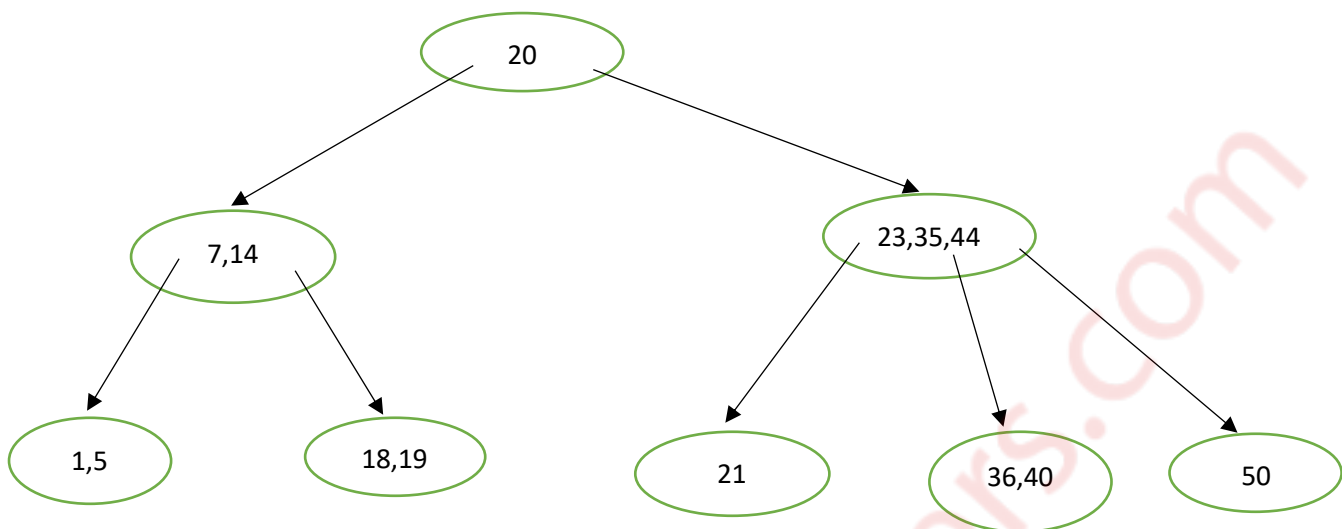
->Properties:

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:



II. B+ Tree

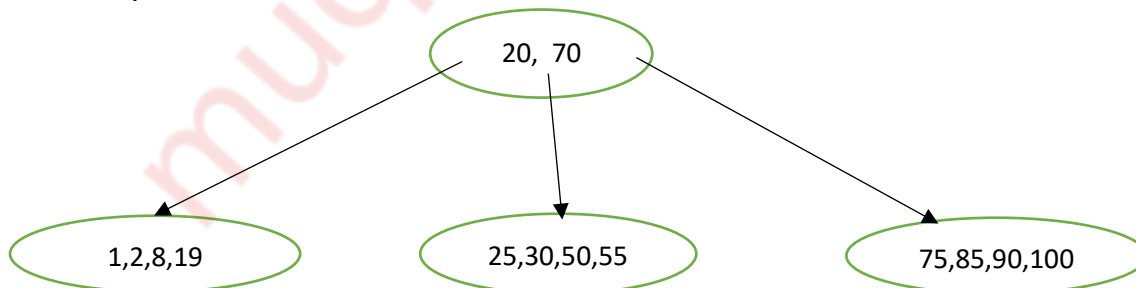
-> A B+ tree is a data structure often used in the implementation of database indexes.

-> Each node of the tree contains an ordered list of keys and pointers to lower level nodes in the tree. These pointers can be thought of as being between each of the keys.

-> Properties:

1. The root node points to at least two nodes.
2. All non-root nodes are at least half full.
3. For a tree of order m , all internal nodes have $m-1$ keys and m pointers.
4. A B+-Tree grows upwards.
5. A B+-Tree is balanced.
6. Sibling pointers allow sequential searching.

->example:



Q1. c) Write a program to implement Binary Search on sorted set of integers.

(10)

Solution:

Code:

```
#include<stdio.h>

int n;

int binary(int a[],int n,int x)
{
    int low=0,high=n-1,mid;
    while(low<=high)
    {
        mid=(low+high)/2;
        if(x==a[mid])
            return(mid);

        else if(x<a[mid])
            high=mid-1;
        else
            low=mid+1;
    }
    return(-1);
}

int main()
{
    int y,i,j;
    printf("enter the size of the array:");
    scanf("%d",&n);
    int a[n];
    printf("entered sorted list of array:");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
```

```

}

printf("enter the number to be searched:");

scanf("%d",&y);

j=binary(a,n,y);

i=j;

if(j!=-1)

printf("element not found");

else

{

printf("\n %d occurred at position=%d \n",y,i+1);

}

return 0;

}

```

Output:

```

enter the size of the array:5
entered sorted list of array:1 2 5 8 9
enter the number to be searched:5
5 occurred at position=3

```

Q2. a) Write a program to convert infix expression into postfix expression.

(10)

Solution:

Code:

```

#include<stdio.h>

#include<ctype.h>

#include<string.h>

#include<stdlib.h>

#define size 50

```

```
int top=-1;
char s[size];
void push(char ch)
{
    s[++top]=ch;
}
char pop()
{
    return(s[top--]);
}
int prec(char ch)
{
    switch(ch)
    {
        case '#':return 0;
        break;
        case '(':return 1;
        break;
        case '-': case '+':return 2;
        break;
        case '/': case '*': return 3;
        break;
    }
}
int main()
{
    char ch,elem,ix[50],px[50];
    int i=0,k=0;
    push('#');
```

```

printf("enter infix expression:");
gets(ix);
while((ch=ix[i++])!='\0')
{
    if(isalnum(ch))
        px[k++]=ch;
    else if (ch=='(')
        push(ch);
    else if(ch==')')
    {
        while(s[top]!='(')
            px[k++]=pop();
        elem=pop();
    }
    else
    {
        if(s[top]=='#')
            push(ch);
        else if( s[top]=='(')
            push(ch);
        else if( prec(ch)>prec(s[top]))
            push(ch);
        else
            px[k++]=pop();
    }
}

while(s[top]!='#')
    px[k++]=pop();
px[k]='\0';

```

```

printf("postfix expression :");

puts(px);

return 0;

}

```

Output:

enter infix expression:a+b*(c+d)

postfix expression :abcd+*+

Q2. b) Explain Huffman encoding with an example.

(10)

Solution:

-> Huffman encoding is a concept which is developed by David Huffman.

-> The original data can be perfectly reconstructed from compressed data.

-> It is mainly used in instant compression format, Jpack, png, mp3 and gzip.

-> Algorithm

Sort given symbol in increasing order of frequency then create a binary tree using following steps:-

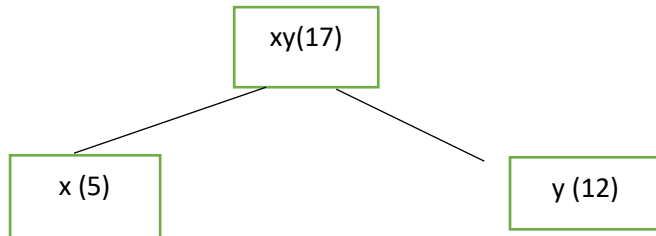
- 1.From the frequency table select 2 symbols s1 and s2 with minimum frequency f1 and f2. Create a node for s1 and s2.
- 2.Combine the nodes s1 and s2 and create a new symbol s12. Create a node for s12.
- 3.Remove s1 and s2 from frequency table and insert s1 keeping the frequency table sorted.
- 4.Repeat step 1 and 3 till only one symbol is left in frequency table.

-> Example:

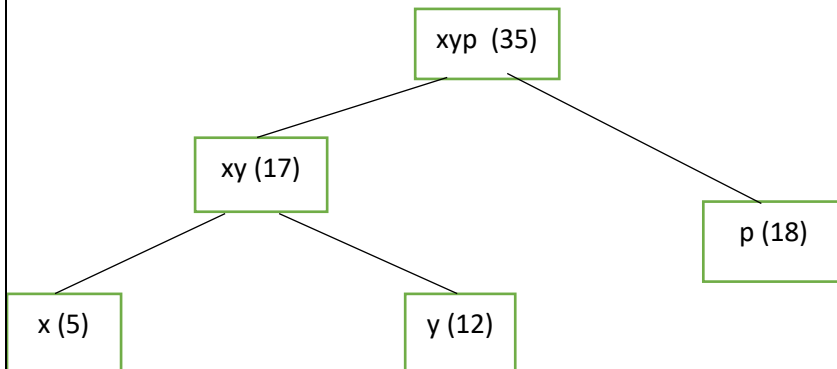
a	m	x	y	p
85	20	5	12	18

x	y	p	m	a
5	12	18	20	85

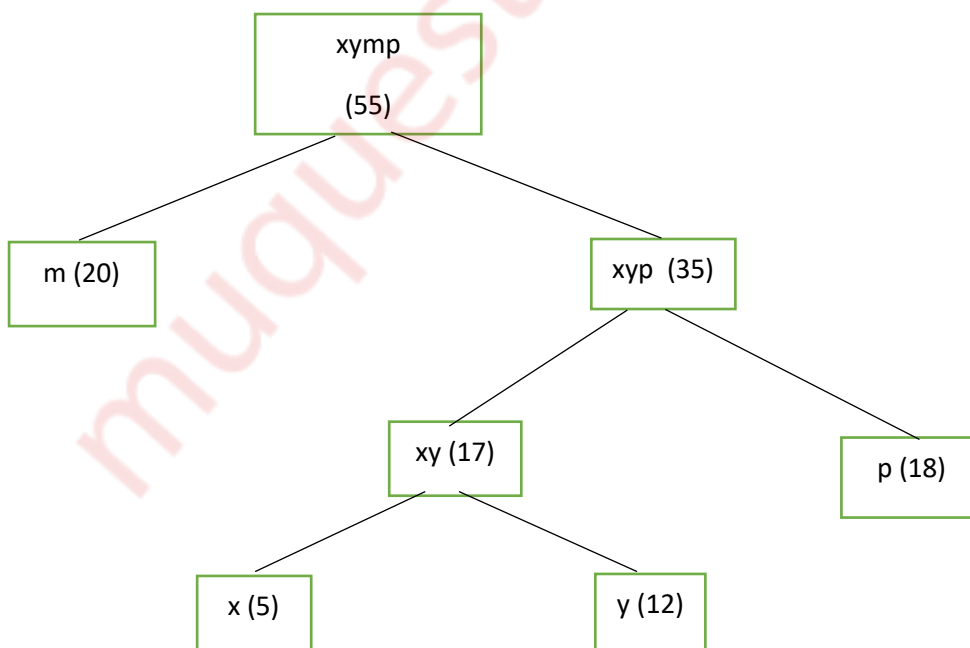
xy	p	m	a
17	18	20	85

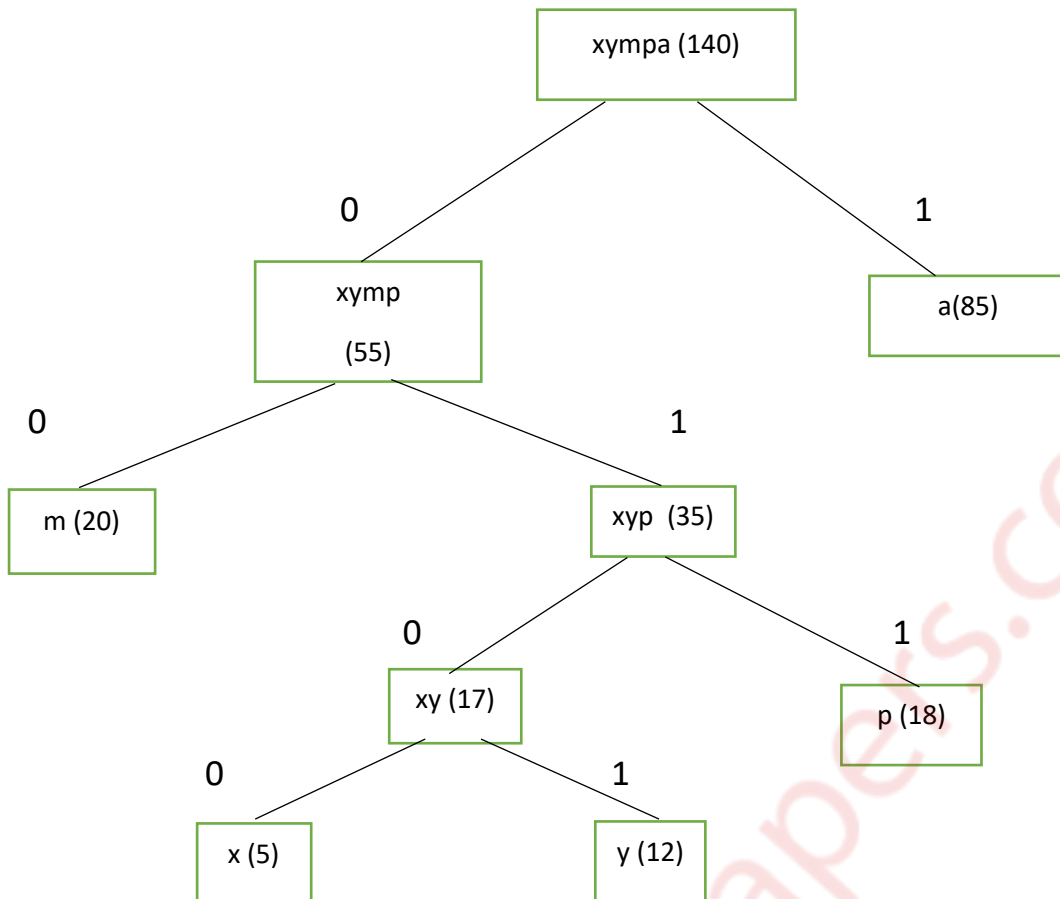


m	xyp	a
20	35	85



xypm	a
55	85





alphabets	frequency	bits	No. of bits
m	20	00	2
x	5	0100	4
y	12	0101	4
p	18	011	3
a	85	1	1

No. of bits required= $20 \times 2 + 5 \times 4 + 12 \times 4 + 18 \times 3 + 85 \times 1 = 247$ bits

Q3. a) Write a program to implement Doubly Linked List. Perform the following operations:

- (i) Insert a node in the beginning
- (ii) Insert a node in the end
- (iii) Delete a node from the end
- (iv) Display a list

(10)

Solution:

Code:

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

```
#include <stdlib.h>
```

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

```
struct node *head = NULL, *tail = NULL;
```

```
struct node * createNode(int data) {  
    struct node *newnode;  
    newnode = (struct node *)malloc(sizeof (struct node));  
    newnode->data = data;  
    newnode->next = NULL;  
    newnode->prev = NULL;  
    return (newnode);  
}
```

```
void createDummies() {  
    head = (struct node *)malloc(sizeof (struct node));  
    tail = (struct node *)malloc(sizeof (struct node));  
    head->data = tail->data = 0;  
    head->next = tail;  
    tail->prev = head;  
    head->prev = tail->next = NULL;  
}
```

```
void insertAtStart(int data)
```

```
{  
    struct node *newnode = createNode(data);
```

```
newnode->next = head->next;
newnode->prev = head;
head->next->prev = newnode;
head->next = newnode;
nodeCount++;
}
void insertAtEnd(int data)
{
    struct node *newnode = createNode(data);
    newnode->next = tail;
    newnode->prev = tail->prev;
    tail->prev->next = newnode;
    tail->prev = newnode;
    nodeCount++;
}

void deleteend()
{
    struct node *ptr=tail;
    tail->prev->next=tail->next;
    tail=tail->prev;

    free(ptr);
}

void display()
{
    struct node *ptr;
```

```

ptr=head->next;
while (ptr!=tail)
{
    printf("%d \t ",ptr->data);
    ptr=ptr->next;
}
}

int main()
{
    int data, ch, pos;
    createDummies();
    while (1)
    {
        printf("\n 1. Insert at start\n 2.Insert at end\n 3.delete at end\n 4. Exit\n");
        printf("Enter your choice:");
        scanf("%d", &ch);
        switch (ch) {
            case 1: printf("Enter your data to insert at start:");
                    scanf("%d", &data);
                    insertAtStart(data);
                    display();
                    break;
            case 2:
                    printf("Enter your data to insert:");
                    scanf("%d", &data);
                    insertAtEnd(data);
                    display();
                    break;
            case 3: deleteend();

```

```

        display();
        break;
    case 4:
        exit(0);
    default:
        printf("U have entered wrong option\n");
        break;
    }
}
return 0;
}

```

Output:

1. Insert at start

2.Insert at end

3.delete at end

4. Exit

Enter your choice:1

Enter your data to insert at start:2

2

1. Insert at start

2.Insert at end

3.delete at end

4. Exit

Enter your choice:1

Enter your data to insert at start:3

3 2

1. Insert at start

2.Insert at end

3.delete at end

4. Exit

Enter your choice:2

Enter your data to insert:5

3 2 5

1. Insert at start

2.Insert at end

3.delete at end

4. Exit

Enter your choice:3

3 2

1. Insert at start

2.Insert at end

3.delete at end

4. Exit

Enter your choice:4

Q3. b) Explain Topological sorting with example.

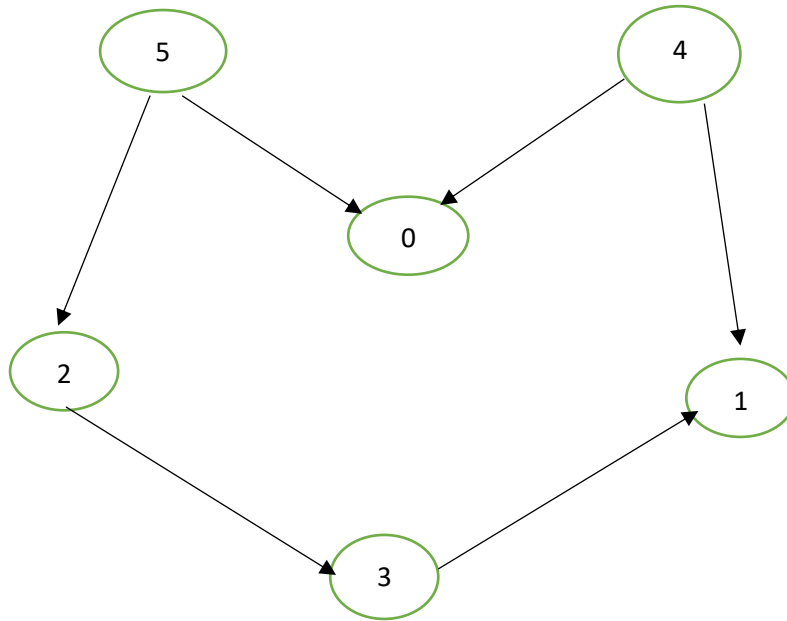
(10)

Solution:

->Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

->For example, a topological sorting of the following graph is "5 4 2 3 1 0".

->There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0



-> we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack.

-> Finally, print contents of stack. A vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Q4. a) Write a program to implement Quick sort. Show the steps to sort the given numbers: 25,13,7,34,56,23,13,96,14,2

(10)

Solution:

Code:

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

```
void quick(int a[40],int first,int last)
{
    int pivot,j,i,temp;
    if(first<last)
    {
        pivot=first;
        i=first;
        j=last;

        while(i<j)
        {
            while(a[i]<=a[pivot] && i<last)
                i++;
            while(a[j]>a[pivot] && j>first)
                j--;
            if(i<j)
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
        temp=a[pivot];
        a[pivot]=a[j];
        a[j]=temp;
        quick(a,first,i-1);
        quick(a,i,j);
    }
}
```



```

    while(a[j]>a[pivot])
        j--;

    if(i<j)
    {
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
    }
}
temp=a[pivot];
a[pivot]=a[j];
a[j]=temp;

quick(a,first,j-1);
quick(a,j+1,last);
}
}

void main()
{
    int a[20],n,i;
    printf("enter the no. of elements:");
    scanf("%d",&n);

    printf("enter the elements:");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }

    quick(a,0,n-1);
    printf("sorted elements are:");
    for(i=0;i<n;i++)
    {
        printf("%d \t",a[i]);
    }
}

```

Output:

enter the no. of elements:10

enter the elements:25 13 7 34 56 23 13 96 14 2

sorted elements are:2 7 13 13 14 23 25 34 56 96

steps:


```
void insert();
void delet();
void display();
int queue[MAX],rear=-1,front=-1,item;
int main()
{
    int ch;
    do
    {
        printf("\n\n1.Insert\n2 Delete\n3.Display\n4.Exit\n\n");
        printf("Enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert();
                break;

            case 2:
                delet();

                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice, Please try again\n");
```

```

        }

while(1);

    getch();
}

void insert()
{
    if(rear==MAX-1) // condition for Queue Full
        printf("Queue is Full\n");
    else
    {
        printf("\n\n Enter item\n");
        scanf("%d",&item);

        if(rear == -1 && front ==-1 ) // is Queue Empty
        {
            rear=0;
            front=0;
        }
        else // otherwise
            rear++;

        queue[rear]=item; // adding element to Queue
        printf("\n\n Item Inseted :%d",item);
    }
} // end insert

void delet()
{
    if(front==--1) // Queue is empty
        printf( "\nQueue is empty\n");
    else // it has element

```

```
{
    item = queue[front];
    if(front==rear) // the element is the last element
    {
        front = -1 ;
        rear=-1;
    }
    else // not the last element
        front++;
    printf("\n\n Item deleted: %d ",item);
}
}

void display()
{
    int i;
    if( front ==-1)
        printf("\nQueue is empty\n");
    else
    {
        printf("\n\n");

        for(i = front; i<=rear ; i++)
            printf("  %d",queue[i]);
    }
}
```

Output:

1.Insert

2 Delete

3.Display

4.Exit

Enter your choice

1

Enter item

12

Item Inserted :12

1.Insert

2 Delete

3.Display

4.Exit

Enter your choice

1

Enter item

55

Item Inserted :55

1.Insert

2 Delete

3.Display

4.Exit

Enter your choice

1

Enter item

88

Item Inserted :88

1.Insert

2 Delete

3.Display

4.Exit

Enter your choice

1

Enter item

66

Item Inserted :66

1.Insert

2 Delete

3.Display

4.Exit

Enter your choice

3

12 55 88 66

1.Insert

2 Delete

3.Display

4.Exit

Enter your choice

2

Item deleted: 12

1.Insert

2 Delete

3.Display

4.Exit

Enter your choice

3

55 88 66

1.Insert

2.Delete

3.Display

4.Exit

Enter your choice

4

Q5. a) Write a program to implement STACK using Linked list. What are the advantages linked list over array? (10)

Solution:

Code:

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

```
#include <stdlib.h>
```

```
struct Node
```

```
{
```

```
    int Data;
```

```
    struct Node *next;
```

```
}*top;
```

```
void popStack()
```

```
{
```

```
    struct Node *var=top;
```

```
    if (top==NULL)
```

```
    {
```

```
        printf("\nStack Empty");
```

```
    }
```

```
    else
```

```
    {
```

```
        top = top->next;
```



```
    free(var);
    printf("element removed");
}
}

void push(int value)
{
    struct Node *temp;

    temp=(struct Node *)malloc(sizeof(struct Node));

    temp->Data=value;

    if (top == NULL)
    {
        top=temp;
        top->next=NULL;
    }
    else
    {
        temp->next=top;
        top=temp;
    }
}

void display()
{
    struct Node *var=top;
    if(var!=NULL)
    {
        printf("\nElements are as:\n");
```

```
while(var!=NULL)
{
    printf("\t%d\n",var->Data);
    var=var->next;
}
printf("\n");
}
else
printf("\nStack is Empty");
}
int main()
{
    int c,val;
    top=NULL;
    printf(" \n1. Push to stack");
    printf(" \n2. Pop from Stack");
    printf(" \n3. Display data of Stack");
    printf(" \n4. Exit\n");
do
{
    printf(" \nChoose Option: ");
    scanf("%d",&c);
    switch(c)
    {
        case 1:
        {
            printf("\nEnter a value to be pushed into Stack: ");
            scanf("%d",&val);
```

```
push(val);
printf("%d added onto stack",val);

break;
}
case 2:
{
popStack();
break;
}
case 3:
{
display();
break;
}
case 4:
{
    struct Node *temp;
    while(top!=NULL)
    {
        temp = top->next;
        free(top);
        top=temp;
    }
    exit(0);
}
default:
{
    printf("\n Wrong choice.");
```

```
    }  
    }  
}while(c!=4);  
}
```

Output:

1. Push to stack
2. Pop from Stack
3. Display data of Stack
4. Exit

Choose Option: 1

Enter a value to be to pushed into Stack: 10

10 added onto stack

Choose Option: 1

Enter a value to be to pushed into Stack: 20

20 added onto stack

Choose Option: 1

Enter a value to be to pushed into Stack: 30

30 added onto stack

Choose Option: 1

Enter a value to be to pushed into Stack: 11

11 added onto stack

Choose Option: 3

Elements are as:

11

30

20

10

Choose Option: 2

element removed

Choose Option: 3

Elements are as:

30

20

10

Choose Option: 4

Advantages of linked list over array:

1. There is no need to specify the size for linked list as it can grow and shrink during execution.
2. Insertion and deletion of an element is easier, faster and efficient.
3. It involves dynamic allocation of memory.
4. Memory utilization is efficient.

Q5. b) Write a program to implement Binary Search Tree (BST), Show BST for the following inputs: 10,5,4,12,15,11,3

(10)

Solution:

Code:

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

```
#include<stdlib.h>
```

```
typedef struct node
```

```
{
```

```
    int data;
```

```
    struct node *left;
```

```
    struct node *right;
```

```
}node;
```

```
node *insert(node *t,int key)
```

```
{
```

```
    if(t==NULL)
```

```
{
```

```
t=(node *)(malloc(sizeof(node)));
t->data=key;
t->left=NULL;
t->right=NULL;
}
else
{
    if(key>t->data)
        t->right=insert(t->right,key);
    else
        t->left=insert(t->left,key);
}
return(t);
}
node *search(node *t,int key)
{
    if(t==NULL)
        return(NULL);
    else if(t->data==key)
        return(t);

    else if(key>t->data)
        return(search(t->right,key));
    else
        return(search(t->left,key));
}
node *max(node *t)
{
    if(t==NULL)
```

```
return(NULL);

while(t->right!=NULL)

t=t->right;

return(t);

}

node *min(node *t)

{

if(t==NULL)

return(NULL);

while(t->left!=NULL)

t=t->left;

return(t);

}

void preorder(node *temp)

{

if(temp!=NULL)

{

printf("\n \n data: %d",temp->data);

preorder(temp->left);

preorder(temp->right);

}

}

void postorder (node *temp)

{

if(temp!=NULL)

{

postorder(temp->left);
```

```

    postorder(temp->right);
    printf("\n \n data: %d",temp->data);
}
}

void inorder(node *temp)
{
    if(temp!=NULL)
    {
        postorder(temp->left);
        printf("\n \n data: %d",temp->data);
        postorder(temp->right);
    }
}

int main()
{
    struct node *root=NULL;
    struct node *temp=NULL;
    int c,key;
    do
    {
        printf("\n 1.insert node \n 2.search \n 3.maximum number \n 4.minimum number \n
        5.display in inorder format \n 6.display in postorfer format \n 7.display in preorder format
        \n 8.exit \n");

        printf("enter your choice:");
        scanf("%d",&c);
        switch(c)
        {
            case 1:printf("enter data:");
                    scanf("%d",&key);
                    if(root==NULL)

```



```
        root=insert(root,key);
    else
        temp=insert(root,key);
        break;
case 2:printf("enter key:");
        scanf("%d",&key);
        temp=search(root,key);
        if(temp!=NULL)
            printf("%d found",key);
        else
            printf("%d not found",key);
break;
case 3:temp=max(root);
        printf("max=%d",temp->data);
break;
case 4:temp=min(root);
        printf("min=%d",temp->data);
break;
case 5:inorder(root);
break;
case 6:postorder(root);
break;
case 7:preorder(root);
break;
}
}while(c!=8);
return 0;
}
```

Output:

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorfer format

7.display in preorder format

8.exit

enter your choice:1

enter data:10

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorfer format

7.display in preorder format

8.exit

enter your choice:1

enter data:55

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorfer format

7.display in preorder format

8.exit

enter your choice:1

enter data:7

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorder format

7.display in preorder format

8.exit

enter your choice:2

enter key:7

7 found

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorder format

7.display in preorder format

8.exit

enter your choice:3

max=55

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorfer format

7.display in preorder format

8.exit

enter your choice:4

min=7

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorfer format

7.display in preorder format

8.exit

enter your choice:5

data: 7

data: 10

data: 55

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorfer format

7.display in preorder format

8.exit

enter your choice:6

data: 7

data: 55

data: 10

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorder format

7.display in preorder format

8.exit

enter your choice:7

data: 10

data: 7

data: 55

1.insert node

2.search

3.maximum number

4.minimum number

5.display in inorder format

6.display in postorder format

7.display in preorder format

8.exit

enter your choice:8

Q6. Write Short notes on(any two) :

(a)AVL Tree

(10)

Solution:

->An AVL tree is a height balance tree. These trees are binary search trees in which the heights of two siblings are not permitted to differ by more than one.

->Searching time in a binary search tree is $O(h)$ where h is the height of the tree. For efficient searching it is necessary that the height should be kept minimum.

-> A full binary search tree with n nodes will have a height of $O(\log n)$. In practice, it is very difficult to control the height of a BST. It lies between $O(n)$ to $O(\log n)$. An AVL tree is a close approximation of full binary search tree.

->The balance factor of a node in a AVL tree could be -1, 0 or 1.

1. 0 indicates that the left and right subtrees are equal.

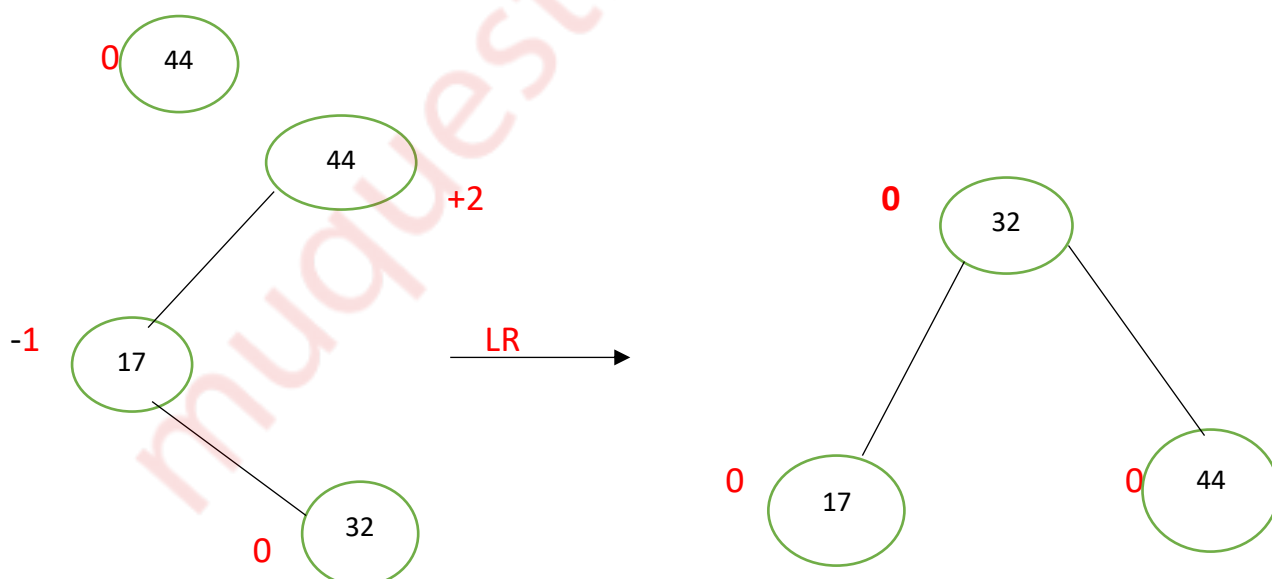
2. +1 means the height of the left subtree is one more than the height of the right subtree.

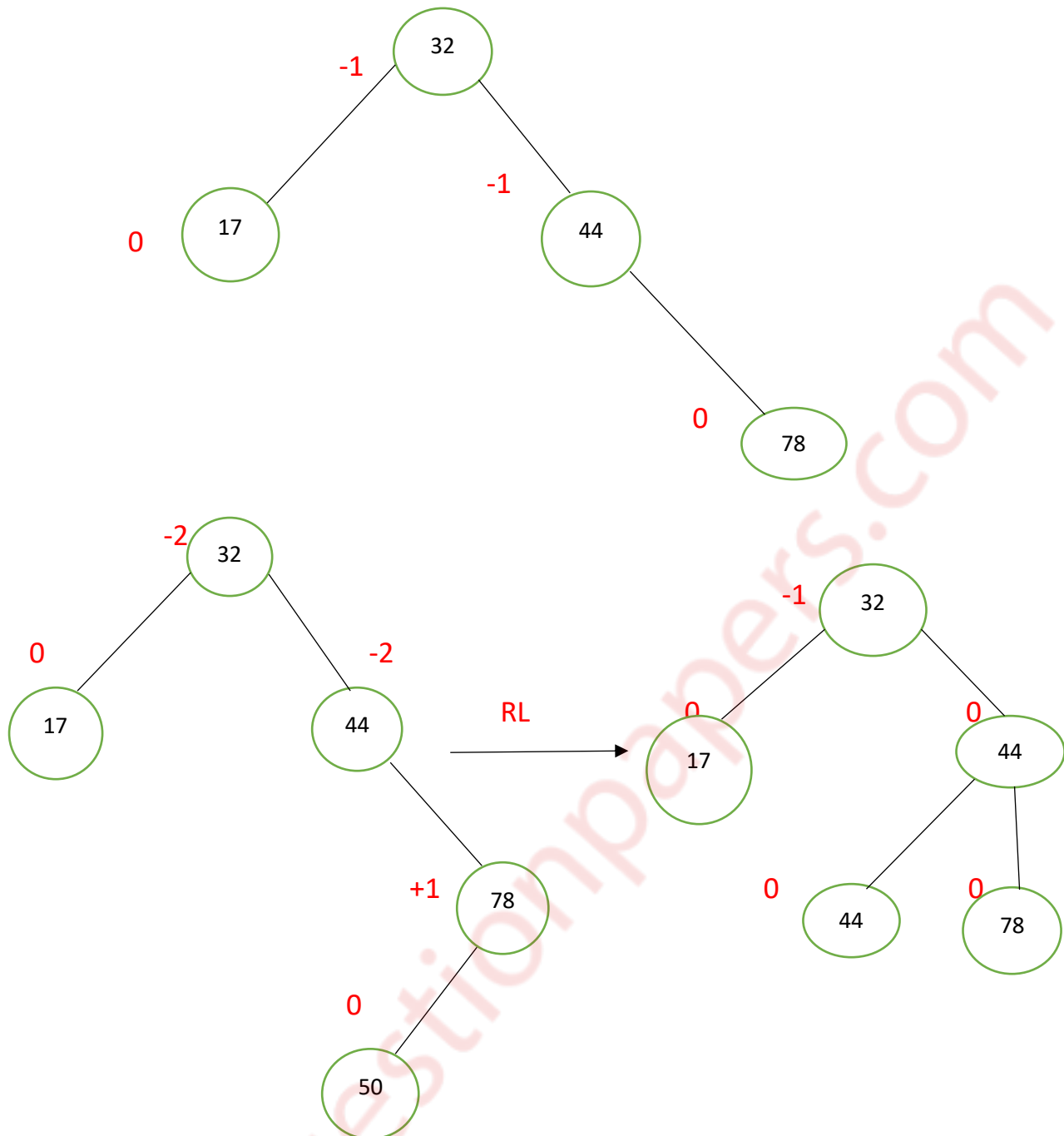
3. -1 indicates that the height of the right subtree is one more than the height of the left subtree.

->

Type of nodes	Rotation
Left left	Right
Right right	Left
Right left	i. right ii. left
Left right	i.left ii. right

->eg: 44,17,32,78,50,88,48,62,54





(a) Graph Traversal Techniques

(10)

Solution:

Traversal of a graph means visiting each node and visiting exactly once.

Two commonly used techniques are:

i) Depth-First Traversal

Rule 1-If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

RULE 2-If you can't follow Rule 1, then, if possible, pop a vertex off the stack.

RULE 3-If you can't follow Rule 1 or Rule 2, you're done

-> In this method, After visiting a vertex v , which is adjacent to w_1, w_2, w_3, \dots ; Next we visit one of v 's adjacent vertices, w_1 say. Next, we visit all vertices adjacent to w_1 before coming back to w_2 , etc.

-> Must keep track of vertices already visited to avoid cycles.

-> The method can be implemented using recursion or iteration.

-> The iterative preorder depth-first algorithm is:

push the starting vertex onto the stack

while(stack is not empty){

pop a vertex off the stack, call it v

if v is not already visited, visit it

push vertices adjacent to v , not visited, onto the stack

}

ii) Breadth-First Traversal

-> In this method, After visiting a vertex v , we must visit all its adjacent vertices w_1, w_2, w_3, \dots , before going down next level to visit vertices adjacent to w_1 etc.

-> The method can be implemented using a queue.

-> A boolean array is used to ensure that a vertex is enqueued only once.

enqueue the starting vertex

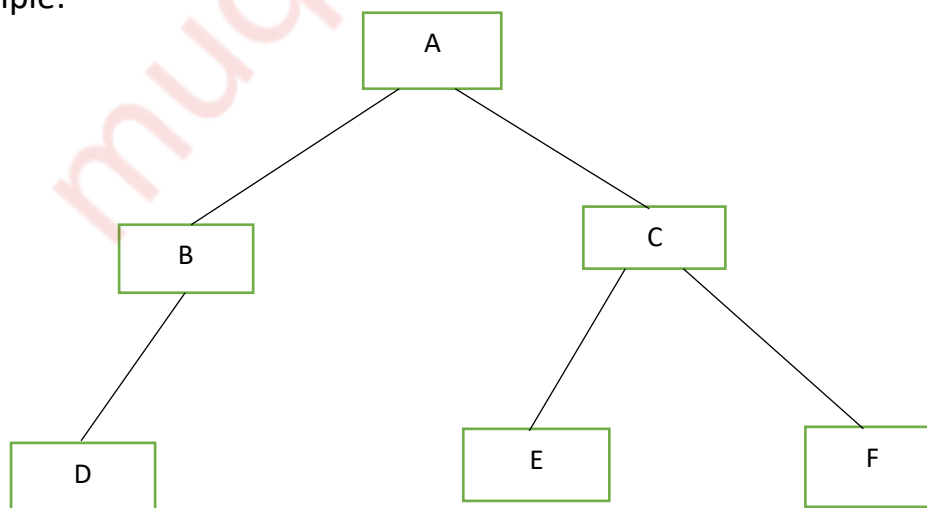
while(queue is not empty){

dequeue a vertex v from the queue;

visit v .

enqueue vertices adjacent to v that were never enqueued; }

Example:



DFS: A,B,D,C,E,F

BFS: A,B,C,D,E,F

(b) Expression Trees

(10)

Solution:

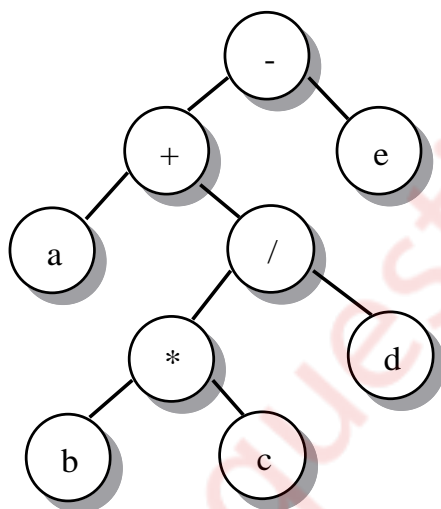
->Expression tree as name suggests is nothing but expressions arranged in a tree-like data structure. Each node in an expression tree is an expression.

->it is a tree with leaves as operands of the expression and nodes contain the operators. Similar to other data structures, data interaction is also possible in an expression tree.

->For example, an expression tree can be used to represent mathematical formula $x < y$ where x , $<$ and y will be represented as an expression and arranged in the tree like structure.

->Expression tree is an in-memory representation of a lambda expression. It holds the actual elements of the query, not the result of the query.

->Expression trees are mainly used for analyzing, evaluating and modifying expressions, especially complex expressions.



$a + b * c / d - e$

(c) Application of Linked list -Polynomial Addition

(10)

Solution:

Code:

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

```
#include<stdlib.h>
#include<ctype.h>
struct node
{
    int coef;
    int exp;
    struct node*next;
};

struct node *insert(struct node *s, int c, int e)
{
    struct node *temp;
    struct node *ptr;
    temp=(struct node *)malloc(sizeof(struct node));
    temp->coef=c;
    temp->exp=e;
    temp->next=NULL;
    if(s==NULL)
    {
        s=temp;
    }
    else
    {
        ptr=s;
        while(ptr->next!=NULL)
        {
            ptr=ptr->next;
        }
    }
}
```

```

        ptr->next=temp;

    }

    return s;
}

struct node *addpoly(struct node *t1,struct node *t2)
{
    struct node *t3=NULL;
    int sum;
    while(t1!=NULL && t2!=NULL)
    {
        if(t1->exp==t2->exp)
        {
            sum=t1->coef+t2->coef;
            t3=insert(t3,sum,t1->exp);
            t1=t1->next;
            t2=t2->next;
        }
        else if((t1->exp) > (t2->exp))
        {
            t3=insert(t3,t1->coef,t1->exp);
            t1=t1->next;
        }
        else if(t2->exp > t1->exp)
        {
            t3=insert(t3,t2->coef,t2->exp);
            t2=t2->next;
        }
    }
}

```

```
        if(t1==NULL)
        {
            while(t2!=NULL)
            {
                t3=insert(t3,t2->coef,t2->exp);
                t2=t2->next;
            }
        }
        if(t2==NULL)
        {
            while(t1!=NULL)
            {
                t3=insert(t3,t1->coef,t1->exp);
                t1=t1->next;
            }
        }
    }

    return t3;
}

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

```

int main( )
{
    int n,m,i,c,e;
    struct node *p1;
    struct node *p2;
    struct node *p3;
    p1=NULL;
    p2=NULL;
    p3=NULL;
    printf("Enter no. of terms in first polynomial \n");
    scanf("%d",&n);
    for(i=0;i<=(n-1);i++)

    {
        printf("enter coefficient and exponent\n");
        scanf("%d %d",&c,&e);
        p1=insert(p1,c,e);
    }
    printf("Enter no. of terms in second polynomial \n");
    scanf("%d",&m);
    for(i=0;i<=(m-1);i++)
    {
        printf("enter coefficient and exponent\n");
        scanf("%d%d",&c,&e);
        p2=insert(p2,c,e);
    }
    p3=addpoly(p1,p2);
    printf("first polynomail: \n");

```

```
display(p1);  
printf("second polynomail: \n");  
display(p2);  
printf("resultant polynomail: \n");  
display(p3);  
return 0;  
}
```

Output:

enter coefficient and exponent

4 1

enter coefficient and exponent

6 0

Enter no. of terms in second polynomial

3

enter coefficient and exponent

7 2

enter coefficient and exponent

3 1

enter coefficient and exponent

1 0

first polynomail:

1 3

2 2

4 1

6 0

second polynomail:

7 2

3 1

1 0

resultant polynomail:

1 3

9 2

7 1

7 0

muquestionpapers.com

DATA STRUCTURES

(MAY 2018)

Q.1

(a) Explain different types of data structure with example. (05)

→ Data structures are generally categorized into two classes:

1. Primitive Non-primitive Data Structure

Primitive: Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.

Non-primitive: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: linear and non-linear data structures

2. Linear and Non-linear Structures

Linear: If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

Example:

1. Linked Lists

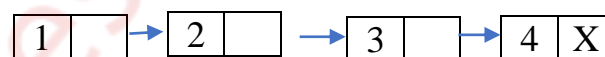


Fig.1 Simple Linked List

2. Stacks

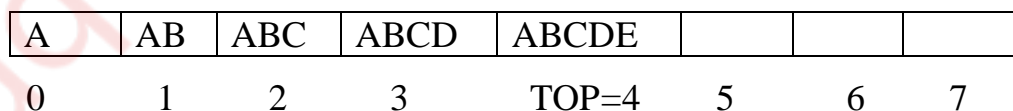
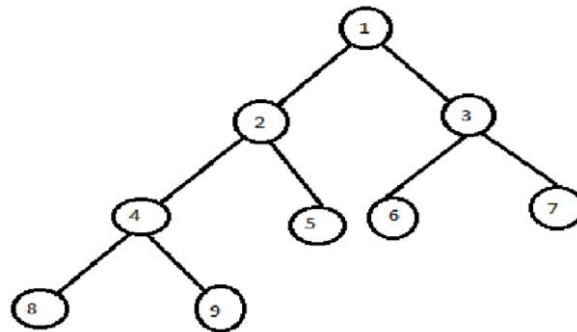


Fig.2 Array representation of a stack

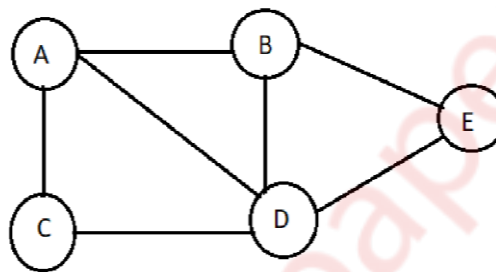
Non-Linear: if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

Example:

1. Trees



2. Graphs

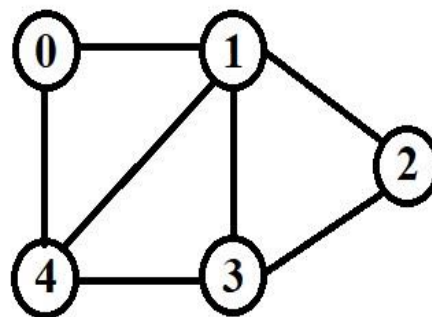


(b) what is graph? Explain methods to represent graph.

(05)

→ 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.

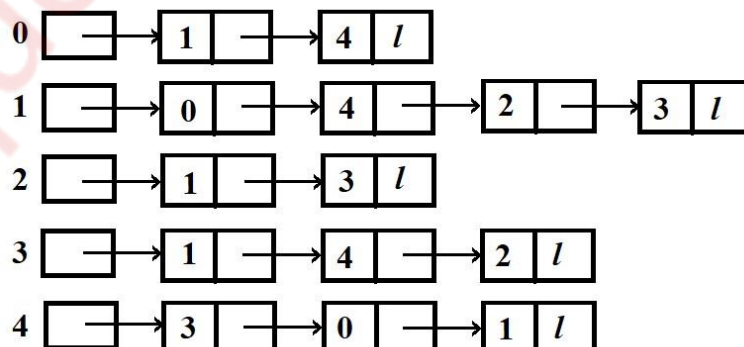
1. Adjacency Matrix

- 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.



(c) write a program in 'C' to implement Merge sort.

(10)

➔ PROGRAM

```
#include <stdio.h>
#include <conio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);

void main()
{
    int arr[size], i, n; printf("\n Enter the number of elements in the array : ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    merge_sort(arr, 0, n-1);
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++)
    printf(" %d\t", arr[i]);
    getch();
}

void merge(int arr[], int beg, int mid, int end)
{
    int i=beg, j=mid+1, index=beg, temp[size], k;
    while((i<=mid) && (j<=end))
    {
        if(arr[i] < arr[j])
        {
            temp[index] = arr[i];
            i++;
        }
        Else
        {
            temp[index] = arr[j];
            j++;
        }
        index++;
    }
    if(i>mid)
```

```

{
while(j<=end)
{
temp[index] = arr[j];
j++;
index++;
}
}
Else
{
while(i<=mid)
{
temp[index] = arr[i];
i++;
index++;
}
}
for(k=beg;k<index;k++)
arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
int mid;
if(beg<end)
{
mid = (beg+end)/2;
merge_sort(arr, beg, mid);
merge_sort(arr, mid+1, end);
merge(arr, beg, mid, end);
}
}

```

OUTPUT:

Enter the number of elements in the array :5

Enter the elements of the array: 12 45 32 67 88

The sorted array is: 12 32 45 67 88

Q.2

(a) Write a program in 'C' to implement QUEUE ADT using Linked-List. Perform the following Operation. (10)

- i) Insert node in the queue.
- ii) Delete node from the list.
- iii) display queue elements.

→ PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
typedef struct node
{
    int data;
    struct node *link;
}
NODE;
NODE *front=NULL,*rear=NULL,*s,*ptr,*disply;
int main()
{
    int no,item; char c;
    printf("\n\tPROGRAM QUEUE USING LINKEDLIST");
    do
    {
        printf("\t\t\tMENU");
        printf("\n\t\t1.INSERT\n\t\t2.DELETE\n\t\t3.DISPLAY\n\t\t4.EXIT\n\t\t");
        Enter your choice: ";
        scanf("%d",&no);
        if(no==1)
        {
            ptr=(NODE*)malloc(sizeof(NODE));
            printf("\t\tEnter the element: ");
            scanf("%d",&ptr->data);
            ptr->link=NULL;
            if(rear==NULL)
            {
                front=ptr; rear=ptr;
            }
            else
            {
                rear->link=ptr; rear=ptr;
            }
        }
    }
}
```

```

}
if(no==2)
{
if(front==NULL)
printf("\t\tStack is empty\n");
else
{
s=front;
printf("\t\tDeleted Element is %d\n",front->data);
front=front->link;
free(s);
if(front==NULL)
rear=NULL;
}
}
if(no==3)
{
if(front==NULL)
printf("\t\tQueue is empty\n");
else
{
printf("\t\tQueue elements are");
disply=front; while(disply!=NULL)
{
printf(" %d",disply->data);
disply=disply->link;
}
("\n");
}
}
if(no==4)
break;
printf("\t\tDo you want to continue(y/n) ");
scanf(" %c",&c);
}
while(c=='y'||c=='Y');
getch();
}

```

OUTPUT:

PROGRAM QUEUE USING LINKEDLIST

MENU

1.INSERT

2.DELETE

3.DISPLAY

4.EXIT

Enter your choice:1

Enter the element:23

Do you want to continue(y/n): y

Enter your choice:1

Enter the element:44

Do you want to continue(y/n): y

Enter your choice:2

Deleted Element is 23

Do you want to continue(y/n): y

Enter your choice:3

Queue elements are 44

Do you want to continue(y/n): n

(10)



Number of collision=2

2. Quadratic Probing

Q.3

(a) Write a program in 'C' to evaluate postfix expression using STACK ADT. (10)

→ Program

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=-1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
    float val;
    char exp[100];
    clrscr();
    printf("\n Enter any postfix expression : ");
    gets(exp);
    val = evaluatePostfixExp(exp);
    printf("\n Value of the postfix expression = %.2f", val);
    getch();
    return 0;
}

float evaluatePostfixExp(char exp[])
{
    int i=0;
    float op1, op2, value;
    while(exp[i] != '\0')
    {
        if(isdigit(exp[i]))
            push(st, (float)(exp[i]-'0'));
        else
        {
            op2 = pop(st);
            op1 = pop(st);
            switch(exp[i])
            {
                case '+':
                    value = op1 + op2;
                    break;
```

```

        case '-':
            value = op1 - op2;
            break;
        case '/':
            value = op1 / op2;
            break;
        case '*':
            value = op1 * op2;
            break;
        case '%':
            value = (int)op1 % (int)op2;
            break;
    }
    push(st, value);
}
i++;
}
return(pop(st));
}
void push(float st[], float val)
{
    if(top==MAX-1)
        printf("\n STACK OVERFLOW");
    else
    {
        top++; st[top]=val;
    }
}
float pop(float st[])
{
    float val=-1; if(top==-1)
        printf("\n STACK UNDERFLOW");
    else
    {
        val=st[top];
        top--;
    }
    return val;
}

```

OUTPUT:

Enter any postfix expression : $9 - ((3 * 4) + 8) / 4$

Value of the postfix expression $9\ 3\ 4\ *\ 8\ +\ 4\ /\ -$

(b) Explain different types of tree traversals techniques with example. Also write recursive function for each traversal technique. (10)

→ TRAVERSING A BINARY TREE

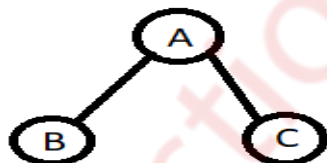
Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.

1. Pre-order Traversal

- To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node.
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

- Example



- The pre-order traversal of the tree is given as A, B, C. Root node first, the left sub-tree next, and then the right sub-tree.

- Recursive Function

Void preorder (node * T) /* address of the root node is passed in T */

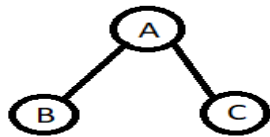
```
{  
    if (T != NULL)  
    {  
        printf("\n%d", T->data);  
        preorder( T->left);  
        preorder( T->right);  
    }  
}
```

2. In-order Traversal

- To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree.
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

- Example



- The pre-order traversal of the tree is given as B,A,C . Left sub-tree first, the root node next, and then the right sub-tree.

- Recursive Function

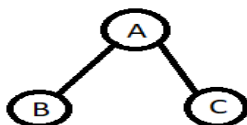
```
Void inorder (node * T) /* address of the root node is passed in T */  
{  
    if (T != NULL)  
    {  
        inorder( T -->left);  
        printf("\n%d, T --> data);  
        inorder( T -->right);  
    }  
}
```

3. Post-order Traversal

- To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree.
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

- Example



- The pre-order traversal of the tree is given as B,C,A . Left sub-tree first, the right sub-tree and then the root node .
- Recursive Function

Void postorder (node * T) /* address of the root node is passed in T */

```
{
    if (T != NULL)
    {
        postorder( T --> left);
        postorder( T --> right);
        printf("\n%d, T --> data);
    }
}
```

Q.4

(a) State advantages of Linked List over arrays. Explain different applications of Linked List. (10)

→ Linked list over arrays

- Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations.
- Another point of difference between an array and a linked list is that a linked list does not allow random access of data
- Nodes in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.
- Another advantage of a linked list over an array is that we can add any number of elements in the list
- linked lists provide an efficient way of storing related data and performing basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing the address of next nodes.

Application of Linked list

- Linked lists can be used to represent polynomials and the different operations that can be performed on them
- we will see how polynomials are represented in the memory using linked lists.

1. 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.

(b) Write a program in 'C' to implement Circular queue using arrays. (10)

→ Program

```
#include<stdio.h>
#include<conio.h>
#define size 5
int q[size],front=-1,rear=-1,i,element;
void insert(int ele);
int del();
void disp();

void main()
{
    int ch,ele;
    clrscr();
    printf("\t ***** Main Menu *****");
    printf("\n 1. insert \n 2.delete \n 3.display \n 4.Exit\n");
    do
    {
        printf("\n Enter your choice: \n \n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:printf("\n Enter Element to Insert \n ");
```

```
scanf("%d",&ele);
insert(ele);
disp();
break;
```

```
case 2:ele=del();
scanf("\n %d is the deleted element \n ",ele);
disp();
break;
```

```
case 3:disp();
break;
```

```
case 4:break;
default:printf(" \n Invalid Statement \n");
}
}
while(ch!=4); getch();
}
```

```
void insert(int ele)
{
if(front==-1 && rear==-1)
{
front=rear=0;  q[rear]=ele;
}
else if((rear+1)%size==front)
{
printf("\n Queue is Full \n");
}
else
{
rear=(rear+1)%size;
q[rear]=ele;
}
}
```

```
int del()
{
if(rear==-1 && front==-1)
{
printf("\n Queue is Empty \n");
}
}
```

```
else if(rear==front)
{
rear=-1;
front=-1;
printf("\n Queue is Empty \n");
}
else
{
element=q[front];  front=(front+1)%size;
}
return element;
}
```

```
void disp()
{
if(rear==-1 && front==-1)
{
printf("\n Queue is Empty \n");
}
else
{
for(i=front;i<(rear+1)%size;i++)
{
printf("\t %d",q[i]);
}
}
}
```

OUTPUT:

***** Main Menu *****

1. insert

2.delete

3.display

4.Exit

Enter your choice: 1

Enter Element to Insert: 23

Enter your choice: 1

Enter Element to Insert: 45

Enter your choice: 2
23 is the deleted element
Enter your choice: 3
45
Enter your choice: 4

Q.5

(a) Write a program to implement singly Linked List. Provide the following operations:

i) Insert a node at the specified location.

ii) Delete a node from end

iii) Display the list

(10)

→ PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void InsertAtPosition(int value,int position);
void RemoveAtEnd();
void display();
int CheckEmpty();

struct Node{
int data;
struct Node *next;
}* head=NULL;

int main()
{
int value,choice;
char c;
do{
printf("Enter\n1-Insert\n2-Remove\n3-Display\n");
printf("Enter your choice");
scanf("%d",&choice);
switch(choice)
{
```

```

case 1:
{
int x;
printf("Enter \n1-Insert at Position\n");
scanf("%d",&x);
printf("Enter Value to be Inserted\n");
scanf("%d",&value);
switch(x)
{
case 1:
{
int position;
printf("Enter position to insert a value(counted from 0)\n");
scanf("%d",&position);
InsertAtPosition(value,position);
break;
}
default :
{
printf("Enter Valid Choice\n");
break;
}
}
break;
}

case 2:
{
int x;
printf("Enter \n1-Delete At End\ n");
scanf("%d",&x);
switch(x)
{
case 1:
{
RemoveAtEnd();
break;
}
default :
{
printf("Enter Valid Choice\n");
break;
}
}
}

```

```
}  
break;  
}
```

```
case 3:
```

```
{  
display();  
break;  
}
```

```
default:
```

```
{  
printf("Enter Valid Choice\n");  
break;  
}  
}
```

```
printf("Enter 'Y' to continue else any letter\n");  
fflush(stdin);  
c=getche();  
printf("\n");  
} while(c=='Y' || c=='y');  
return(0);  
}
```

```
void InsertAtPosition(int value,int position){  
struct Node *newNumber,*temp;  
int count,flag;  
newNumber = (struct Node*)malloc(sizeof(struct Node));  
newNumber->data = value;  
temp=head;  
flag=CheckEmpty();  
if(flag==1)  
{  
int flag1=0;  
count=0;  
while(temp!=NULL)  
{  
if(count==position-1)  
{  
flag1=1;  
newNumber->next=temp->next;  
temp->next=newNumber;  
}  
}
```

```
else
{
temp=temp->next;
}
count++;
}
if(flag1==0)
{
printf("Entered Position Not available\n");
}
else
{
printf("Given number %d is inserted at position %d
successfully\n",value,position);
}
}
else
{
printf("List is Empty\n");
}
}
```

```
void RemoveAtEnd()
{
int flag=CheckEmpty();
if(flag==1)
{
if(head->next==NULL)
{
head=NULL;
}
else
{
struct Node *temp=head,*temp1;
while(temp->next!=NULL)
{
temp1=temp;
temp=temp->next;
}
temp1->next=NULL;
free(temp);
}
}
```

```

else
{
printf("List Empty.Try again!\n");
}
}
void display()
{
int flag=CheckEmpty();
if(flag==1)
{
struct Node *temp;
temp=head;
while(temp->next!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
printf("%d",temp->data);
printf("\n");
}
else
{
printf("No List Available\n");
}
}

int CheckEmpty()
{
if(head==NULL)
return 0;
else
return 1;
}

```

OUTPUT:

```

1.Insert
2.Remove
3.Display
Enter your choice:1
Enter
1-Insert at position
Enter value to be inserted
45

```

Enter position to insert a value(counted from 0)

2

Given number 45 is inserted at position 2 successfully

Enter 'Y' to continue else any letter

Y

Enter

1.Insert

2.Remove

3.Display

Enter your choice:3

45

Enter 'Y' to continue else any letter

Y

Enter

1.Insert

2.Remove

3.Display


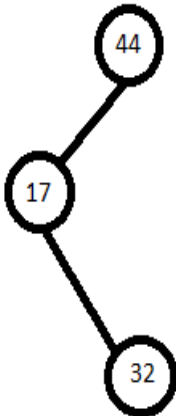
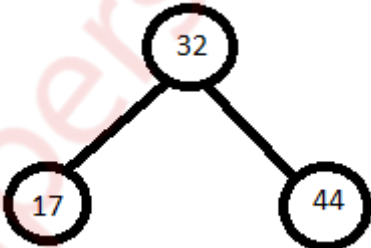
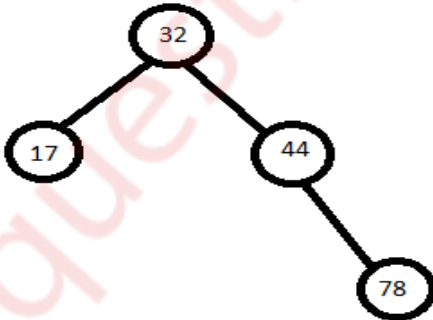
Enter your choice:2

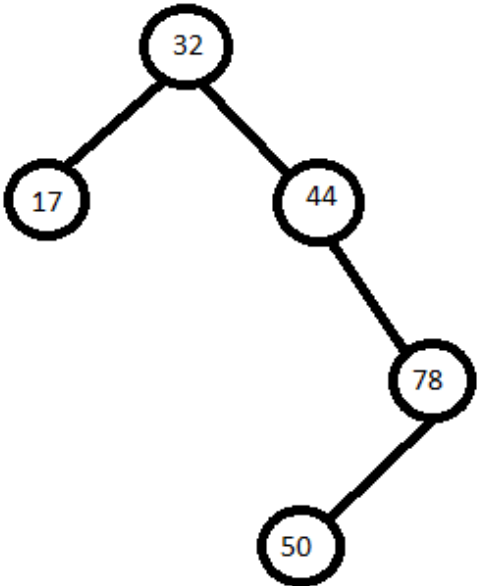
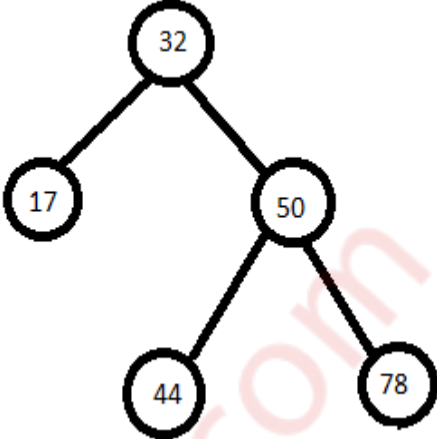
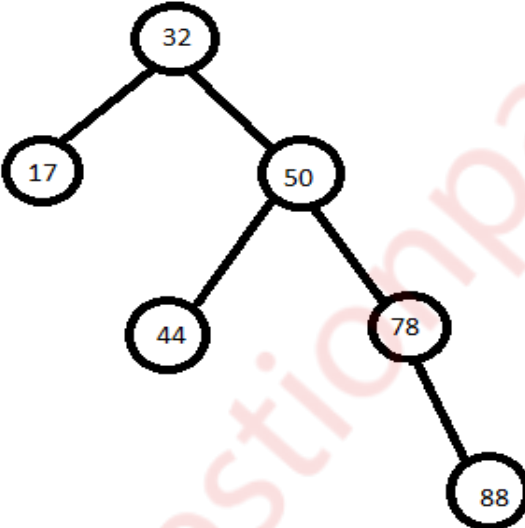
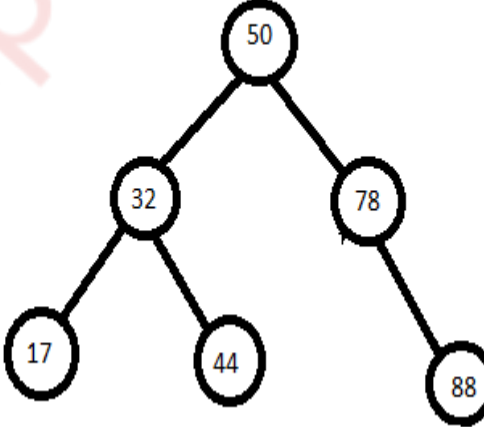
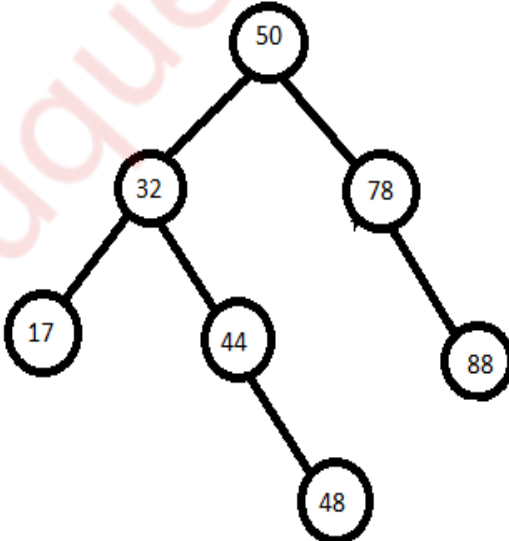
Enter

1-Delete at end

1

(b) Insert the following elements in AVL tree: 44, 17, 32, 78, 50, 88, 48, 62, 54. Explain different rotation that can be used. (10)
→

Sr. no	Data to be insert	Tree after insertion	Tree after Rotation
1	44		
2	17,32		
3	78		

4	50	 <pre> graph TD 32((32)) --- 17((17)) 32 --- 44((44)) 44 --- 78((78)) 78 --- 50((50)) </pre>	 <pre> graph TD 32((32)) --- 17((17)) 32 --- 50((50)) 50 --- 44((44)) 50 --- 78((78)) </pre>
5	88	 <pre> graph TD 32((32)) --- 17((17)) 32 --- 50((50)) 50 --- 44((44)) 50 --- 78((78)) 78 --- 88((88)) </pre>	 <pre> graph TD 50((50)) --- 32((32)) 50 --- 78((78)) 32 --- 17((17)) 32 --- 44((44)) 78 --- 88((88)) </pre>
6	48	 <pre> graph TD 50((50)) --- 32((32)) 50 --- 78((78)) 32 --- 17((17)) 32 --- 44((44)) 44 --- 48((48)) 78 --- 88((88)) </pre>	

7	62	<pre> graph TD 50((50)) --> 32((32)) 50 --> 78((78)) 32 --> 17((17)) 32 --> 44((44)) 44 --> 48((48)) 78 --> 62((62)) 78 --> 88((88)) </pre>	
8	54	<pre> graph TD 50((50)) --> 32((32)) 50 --> 78((78)) 32 --> 17((17)) 32 --> 44((44)) 44 --> 48((48)) 78 --> 62((62)) 78 --> 88((88)) 62 --> 54((54)) </pre>	

Q.6 Explain the following (any two)

(20)

(a) splay Tree and Trie

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.

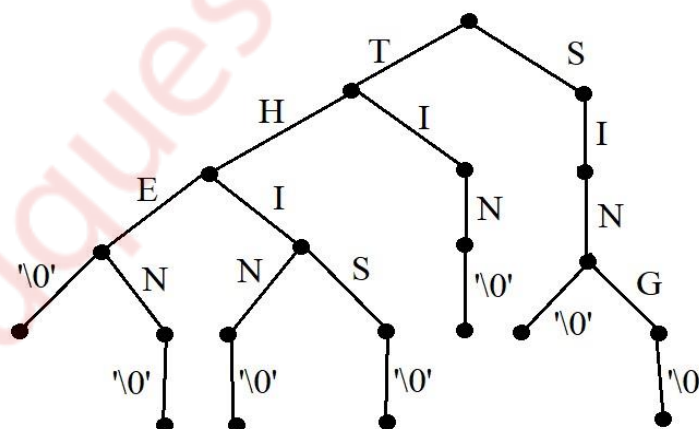
- Splaying the tree for a particular node rearranges the tree to place that node at the root.
- A technique to do this is to first perform a standard binary tree search for that node and then use rotations in a specific order to bring the node on top.

Advantages and Disadvantages of Splay Trees

- A splay tree gives good performance for search, insertion, and deletion operations.
- This advantage centres 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 book-keeping data.
- Unlike other types of self-balancing trees, splay trees provide good performance.

Trie

- A trie is a tree-like data structure whose nodes store the letters of an alphabet. by structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree.
- A trie is a tree of degree $P \geq 2$.
- Tries are useful for sorting words as a string of characters.
- In a trie, each path from the root to a leaf corresponds to one word.
- Root node is always null.
- To avoid confusion between words like THE and THEN, a special end marker symbol '\0' is added at the end of each word.
- Below fig shows the trie of the following words (THE, THEN, TIN, SIN, THIN, SING)



- Most nodes of a trie has at most 27 children one for each letter and for '\0'
- Most nodes will have fewer than 27 children.

- A leaf node reached by an edge labelled '\0' cannot have any children and it need not be there.

(b) Graph Traversal Techniques

➔ There are two standard methods of graph traversal.

1. Breadth-first search

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes.
- Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
- That is, we start examining the node A and then all the neighbours of A are examined.

Algorithm for Breadth-first search

- Step 1: SET STATUS=1 (ready state) for each node in G
- Step 2: Enqueue the starting node A and set its STATUS=2 (waiting state)
- Step 3: Repeat Steps 4 and 5 until QUEUE is empty
- Step 4: Dequeue a node N. Process it and set its STATUS=3 (processed state).
- Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2 (waiting state)
- [END OF LOOP]
- Step 6: EXIT

Applications of Breadth-First Search Algorithm

- Breadth-first search can be used to solve many problems such as:
- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

2. Depth-first Search

- The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- depth-first search begins at a starting node A which becomes the current node.
- Then, it examines each node N along a path P which begins at A.

- That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.

Algorithm for depth-first search

- Step 1: SET STATUS=1(ready state) for each node in G
- Step 2: Push the starting node A on the stack and set its STATUS=2(waiting state)
- Step 3: Repeat Steps 4 and 5 until STACK is empty
- Step 4: Pop the top node N. Process it and set its STATUS=3(processed state)
- Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2(waiting state) [END OF LOOP]
- Step 6: EXIT

(c) Huffman Encoding

☐ Huffman Code:

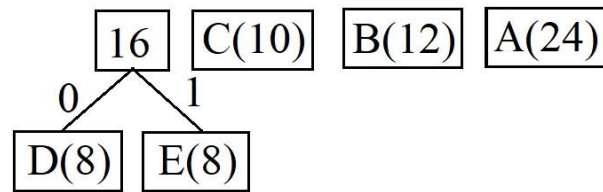
- Huffman code is an application of binary trees with minimum weighted external path length is to obtain an optimal set for messages M₁, M₂, ... M_n - Message is converted into a binary string.
 - Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.
 - It uses patterns of zeros and ones in communication system these are used at sending and receiving end.
 - Suppose there are n standard messages M₁, M₂, ..., M_n. Then the frequency of each message is considered, that is message with highest frequency is given priority for the encoding.
 - The tree is called encoding tree and is present at the sending end. - The decoding tree is present at the receiving end which decodes the string to get corresponding message.
 - The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree.
- Example

Symbol	A	B	C	D	E
Frequency	24	12	10	8	8

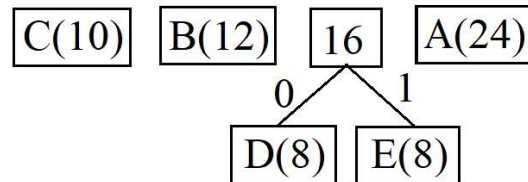
Arrange the message in ascending order according to their frequency

D(8) E(8) C(10) B(12) A(24)

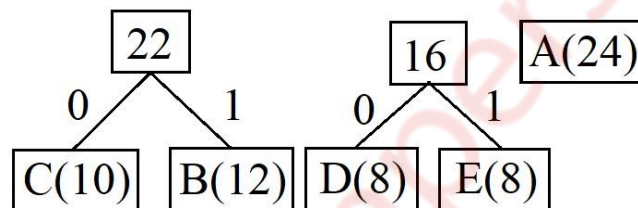
Merge two minimum frequency message



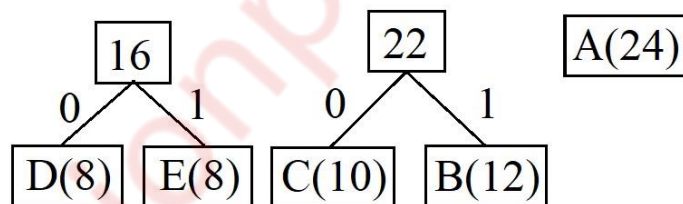
Rearrange in ascending order



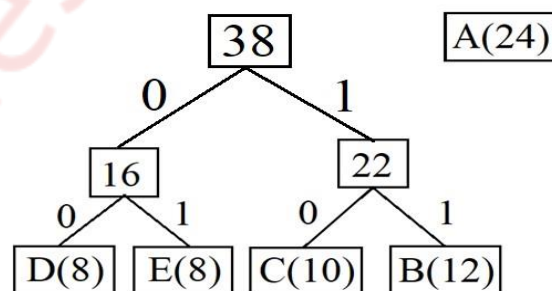
Merge two minimum frequency message



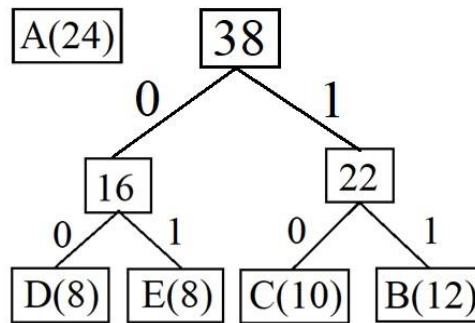
Rearrange in ascending order



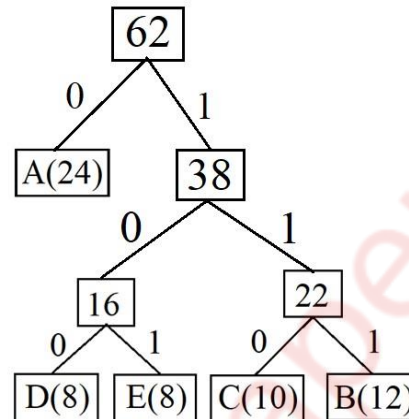
Merge two minimum frequency message



Again Rearrange in ascending order



Merge two minimum frequency message



Huffman code

A = 0

B = 111

C = 110

D = 100

E = 101

(d) 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 deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque.
- They include,

Input restricted deque – In this deque, 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 deque.

- 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 at element at the front end of the queue.
 - v. enqueueR(): Insert at 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

			29	37	45	54	63		
0	1	2	left = 3	4	5	6	right = 7	8	9
42	56						63	27	18
0	right = 1	2	3	4	5	6	left = 7	8	9

Fig. Double-ended queues

DATA STRUCTURES

(DEC 2018)

Q 1

a) What are various operations possible on data structures? (05)

→ The data appearing in our data structure is processed by means of certain operations. In fact, the particular data structure that once chooses for a given situation depends largely on the frequency with which specific operations are performed:

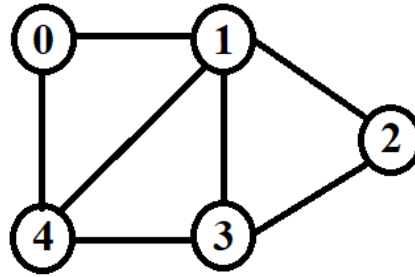
1. Insertion: Insertion means addition of a new data element in a data structure.
2. Deletion: Deletion means removal of a data element from a data structure if it is found.
3. Searching: Searching involves searching for the specified data element in a data structure.
4. Traversal: Traversal of a data structure means processing all the data elements present in data structure.
5. Sorting: Arranging data elements of a data structure in a specified order is called sorting.
6. Merging: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

b) What are different ways of representing a Graph data structure on a computer? (05)

→ 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.

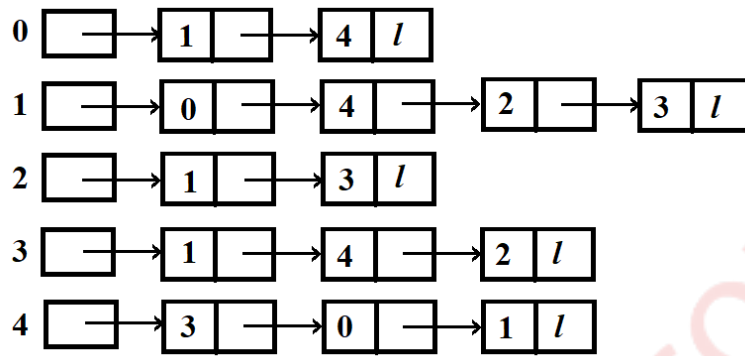
1. Adjacency Matrix

- 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 $adj[i][j]$, 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.

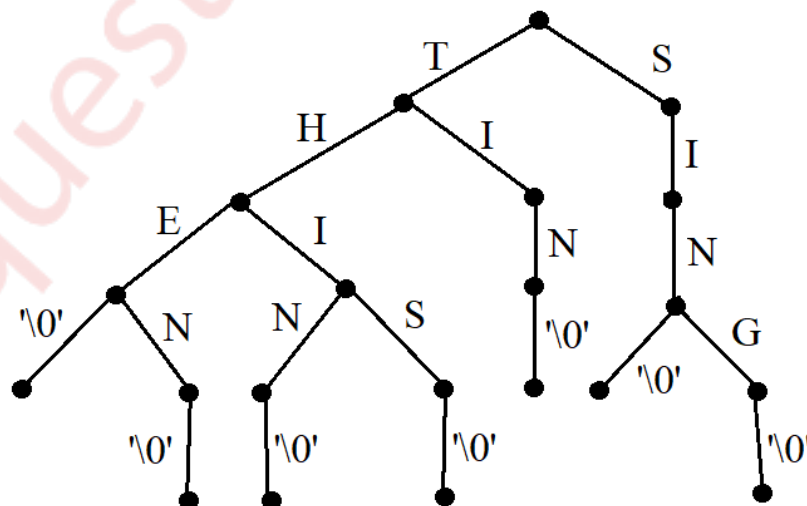


c) **Describe Tries with an example.**

(05)

➔ - A trie is a tree-like data structure whose nodes store the letters of an alphabet. By structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree.

- A trie is a tree of degree $P \geq 2$.
- Tries are useful for sorting words as a string of characters.
- In a trie, each path from the root to a leaf corresponds to one word.
- Root node is always null.
- To avoid confusion between words like THE and THEN, a special end marker symbol '\0' is added at the end of each word.
- Below fig shows the trie of the following words (THE, THEN, TIN, SIN, THIN, SING)



- Most nodes of a trie has at most 27 children one for each letter and for '\0'
- Most nodes will have fewer than 27 children.
- A leaf node reached by an edge labelled '\0' cannot have any children and it need not be there.

d) Write a function in C to implement binary search. (05)

➔ Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

C function to implement binary search:

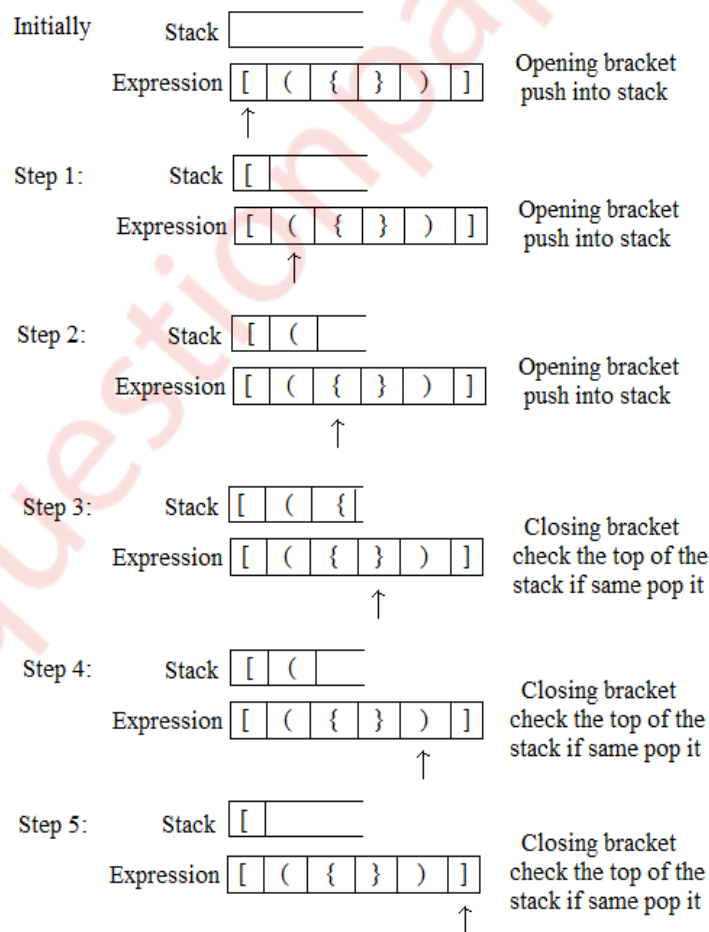
```
int binary_search(int sorted_list[], int low, int high, int element)
{
    int middle;
    while (low <= high)
    {
        middle = low + (high - low)/2;
        if (element > sorted_list[middle])
            low = middle + 1;
        else if (element < sorted_list[middle])
            high = middle - 1;
        else
            return middle;
    }
    return -1;
}
```

Q 2

a) Use stack data structure to check well-formedness of parentheses in an algebraic expression. Write C program for the same. (10)

→ An expression is said to be well formed with respect to parenthesis:

1. If every opening parenthesis has a closing parenthesis.
2. If we count number of parenthesis of left parenthesis and right parenthesis then at no time, count of right parenthesis should exceed the count of left parenthesis.
3. A stack is used to validate an expression using simple rules by scanning the expression from left to right.
 - If opening bracket is found then push it on the stack.
 - If closing bracket is found then check the top of the stack if it is same then pop
 - If stack is empty then string is valid or else invalid.



C program to check well-formedness of parentheses

```
#include<stdio.h>
#include<string.h>
#define MAX 20
#define true 1
#define false 0

int top = -1;
int stack[MAX];

/*Begin of push*/
char push(char item)
{
    if(top == (MAX-1))
        printf("Stack Overflow\n");
    else
    {
        top=top+1;
        stack[top] = item;
    }
}

/*Begin of pop*/
char pop()
{
    if(top == -1)
        printf("Stack Underflow\n");
    else
        return(stack[top--]);
}

main()
{
    char exp[MAX],temp;
    int i,valid=true;
    printf("Enter an algebraic expression : ");
```

```

gets(exp);

for(i=0;i<strlen(exp);i++)
{
    if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
        push( exp[i] );
    if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
        if(top == -1) /* stack empty */
            valid=false;
        else
        {
            temp=pop();
            if( exp[i]==')' && (temp=='{' || temp=='[') )
                valid=false;
            if( exp[i]=='}' && (temp=='(' || temp=='[') )
                valid=false;
            if( exp[i]==']' && (temp=='(' || temp=='{' ) )
                valid=false;
        }
    }
    if(top>=0) /*stack not empty*/
        valid=false;

    if( valid==true )
        printf("Valid expression\n");
    else
        printf("Invalid expression\n");
}

```

Output:

Enter an algebraic expression: {([])}

Valid expression

Enter an algebraic expression: (){ }

Valid expression

Enter an algebraic expression: (){[[[]]

Invalid expression

- b) Give the frequency for the following symbols, compute the Huffman code for each symbol. (10)

Symbol	A	B	C	D	E
Frequency	24	12	10	8	8

→ Huffman Code:

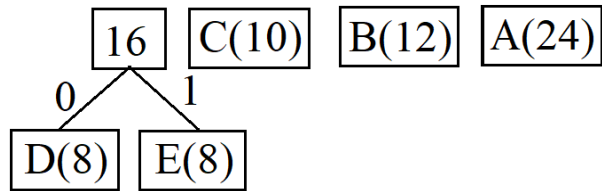
- Huffman code is an application of binary trees with minimum weighted external path length is to obtain an optimal set for messages M_1, M_2, \dots, M_n
- Message is converted into a binary string.
- Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication 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.
- The tree is called encoding tree and is present at the sending end.
- The decoding tree is present at the receiving end which decodes the string to get corresponding message.
- The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree.
- Example

Symbol	A	B	C	D	E
Frequency	24	12	10	8	8

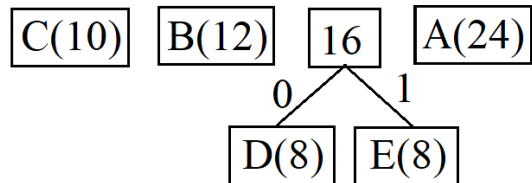
Arrange the message in ascending order according to their frequency

D(8) E(8) C(10) B(12) A(24)

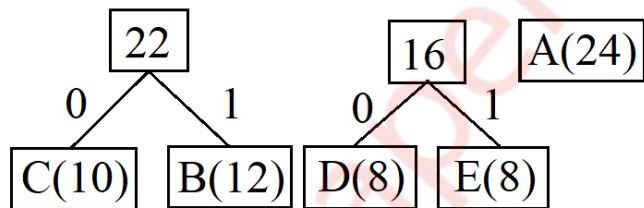
Merge two minimum frequency message



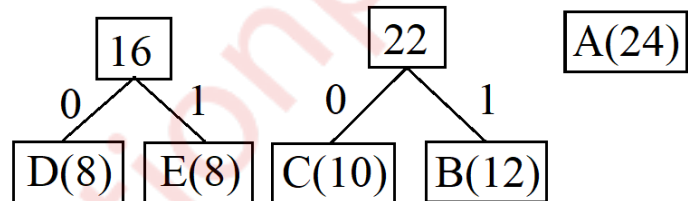
Rearrange in ascending order



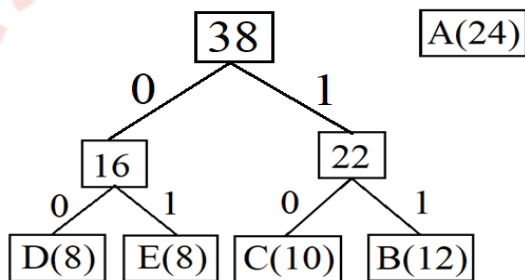
Merge two minimum frequency message



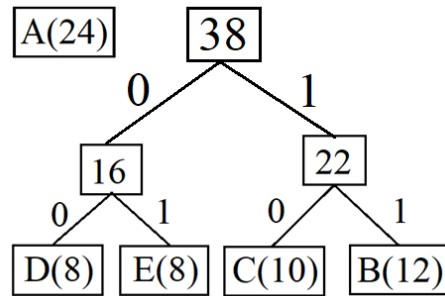
Rearrange in ascending order



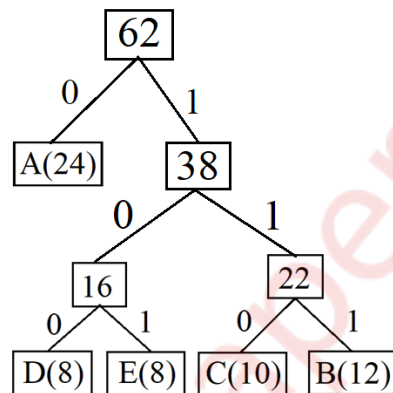
Merge two minimum frequency message



Again Rearrange in ascending order



Merge two minimum frequency message



Huffman code

A = 0

B = 111

C = 110

D = 100

E = 101

Q 3

a) Write a C program to implement priority queue using arrays. The program should perform the following operations (12)

i) Inserting in a priority queue

ii) Deletion from a queue

iii) Displaying contents of the queue

→ Program:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 30

typedef struct pqueue
{
    int data[MAX];
    int rear,front;
}pqueue;

void initialize(pqueue *p);
int empty(pqueue *p);
int full(pqueue *p);
void enqueue(pqueue *p, int x);
int dequeue(pqueue *p);
void display(pqueue *p);

void main()
{
    int x,op,n,i;
    pqueue q;
    initialize(&q);

    do
    {
        printf("\n1)Create \n2)Insert \n3)Delete \n4)Print \n5)EXIT");
        printf("\nEnter Choice: ");
        scanf("%d",&op);
```

```
switch (op) {
    case 1: printf("\nEnter Number of Elements");
            scanf("%d",&n );
            initialize(&q);
            printf("Enter the data");

            for(i=0; i<n; i++)
            {
                scanf("%d",&x);
                if(full(&q))
                {
                    printf("\nQueue is Full..");
                    exit(0);
                }
                enqueue(&q,x);
            }
            break;

    case 2: printf("\nEnter the element to be inserted");
            scanf("%d\n",&x);
            if(full(&q))
            {
                printf("\nQueue is Full");
                exit(0);
            }
            enqueue(&q,x);
            break;

    case 3: if(empty(&q))
            {
                printf("\nQueue is empty..");
                exit(0);
            }

            x=dequeue(&q);
            printf("\nDeleted Element=%d",x);
            break;
```

```
        case 4: display(&q);
                break;
        default: break;
    }
} while (op!=5);
}
```

```
void initialize(pqueue *p)
{
    p->rear=-1;
    p->front=-1;
}
```

```
int empty(pqueue *p)
{
    if(p->rear==-1)
        return(1);

    return(0);
}
```

```
int full(pqueue *p)
{
    if((p->rear+1)%MAX==p->front)
        return(1);

    return(0);
}
```

```
void enqueue(pqueue *p, int x)
{
    int i;
    if(full(p))
        printf("\nOverflow");
    else
    {
```

```

        if(empty(p))
        {
            p->rear=p->front=0;
            p->data[0]=x;
        }
        else
        {
            i=p->rear;

            while(x>p->data[i])
            {
                p->data[(i+1)%MAX]=p->data[i];
                i=(i-1+MAX)%MAX; //anticlockwise movement inside the
queue
                if((i+1)%MAX==p->front)
                    break;
            }

            //insert x
            i=(i+1)%MAX;
            p->data[i]=x;

            //re-adjust rear
            p->rear=(p->rear+1)%MAX;
        }
    }
}

int dequeue(pqueue *p)
{
    int x;

    if(empty(p))
    {
        printf("\nUnderflow..");
    }
    else

```

```

    {
        x=p->data[p->front];
        if(p->rear==p->front) //delete the last element
            initialize(p);
        else
            p->front=(p->front +1)%MAX;
    }

    return(x);
}

void display(pqueue *p)
{
    int i,x;

    if(empty(p))
    {
        printf("\nQueue is empty..");
    }
    else
    {
        i=p->front;
        while(i!=p->rear)
        {
            x=p->data[i];
            printf("\n%d",x);
            i=(i+1)%MAX;
        }

        //prints the last element
        x=p->data[i];
        printf("\n%d",x);
    }
}

```

Output:

1)Create

2)Insert

3)Delete

4)Display

5)EXIT

Enter Choice: 1

Enter Number of Elements4

Enter the data9

12

4

6

1)Create

2)Insert

3)Delete

4)Display

5)EXIT

Enter Choice: 4

12

9

6

4

1)Create

2)Insert

3)Delete

4)Display

5)EXIT

Enter Choice: 3

Deleted Element=12

1)Create

2)Insert

3)Delete

4)Display

5)EXIT

Enter Choice: 5

b) What are expression trees? What are its advantages? Derive the expression tree for the following algebraic expression (08)

$$(a + (b/c)) * ((d/e) - f)$$

→ 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.

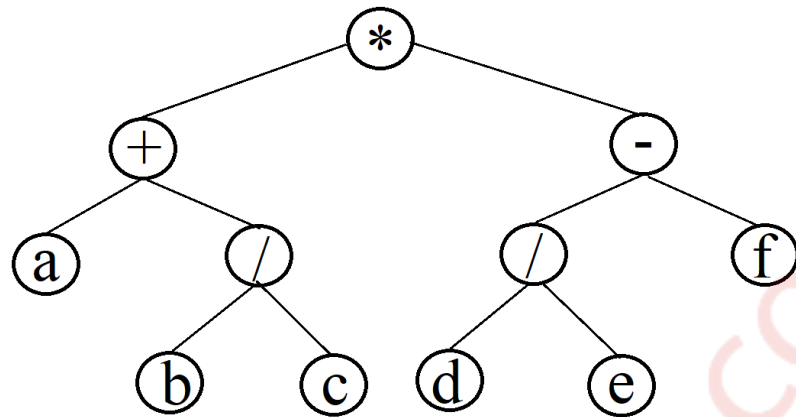
At the end only element of stack will be root of expression tree.

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: $(a + (b/c)) * ((d/e) - f)$



Q 4

- a) Write a C program to represent and add two polynomials using linked list. (12)

→ Program:

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

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

void get_input(node** head)
{
    node* temp,*ptr;
    ptr = *head;
    temp = (node*)malloc(sizeof(node));

    printf("\nEnter coef : ");
    scanf("%d",&(temp->coef));
```

```

printf("\nEnter exp : ");
scanf("%d",&(temp->exp));

if(NULL == *head)
{
    *head = temp;
    (*head)->next = NULL;
}
else
{
    while(NULL != ptr->next)
    {
        ptr = ptr->next;
    }
    ptr->next = temp;
    temp->next = NULL;
}
}

void display(node* head)
{
    while(head->next != NULL)
    {
        printf("(%d.x^%d)+",head->coef,head->exp);
        head = head->next;
    }
    printf("(%d.x^%d)",head->coef,head->exp);
    printf("\n");
}

void add(node* poly, node* poly1, node* poly2 ) //poly==result
{
    while(poly1->next && poly2->next)
    {
        if(poly1->exp > poly2->exp)
        {
            poly->coef = poly1->coef;

```

```

        poly->exp = poly1->exp;
        poly1 = poly1->next;
    }
    else if(poly1->exp < poly2->exp)
    {
        poly->coef = poly2->coef;
        poly->exp = poly2->exp;
        poly2 = poly2->next;
    }
    else
    {
        poly->coef = poly1->coef + poly2->coef;
        poly->exp = poly1->exp;
        poly1 = poly1->next;
        poly2 = poly2->next;
    }
    poly->next = (node*)malloc(sizeof(node));
    poly = poly->next;
    poly->next = NULL;
}

while(poly1->next || poly2->next)
{
    if(poly1->next)
    {
        poly->coef = poly1->coef;
        poly->exp = poly1->exp;
        poly1 = poly1->next;
    }
    if(poly2->next)
    {
        poly->coef = poly2->coef;
        poly->exp = poly2->exp;
        poly2 = poly2->next;
    }
    poly->next = (node*)malloc(sizeof(node));
    poly = poly->next;
}

```

```

        poly->next = NULL;
    }
}

int main()
{
    node* head1 = NULL;
    node* head2 = NULL;
    node* head = (node*)malloc(sizeof(node));
    int ch;

    do
    {
        get_input(&head1);
        printf("\nEnter more node in poly1? (1,0) :");
        scanf("%d",&ch);
    }while(ch);
    do
    {
        get_input(&head2);
        printf("\nEnter more node in poly2? (1,0) :");
        scanf("%d",&ch);
    }while(ch);

    add(head,head1,head2);
    display(head1);
    display(head2);
    display(head);

    return 0;
}

```

Output:

Enter coef: 6

Enter exp: 2

Enter more node in poly1? (1/0): 1

Enter coef: 4

Enter exp: 1
 Enter more node in poly1? (1/0): 1
 Enter coef: 2
 Enter exp: 0
 Enter more node in poly? (1/0): 0
 Enter coef: 4
 Enter exp: 2
 Enter more node in poly2? (1/0): 1
 Enter coef: 2
 Enter exp: 1
 Enter more node in poly2? (1/0): 1
 Enter coef: 3
 Enter exp: 0
 Enter more node in poly2? (1/0): 0
 $(6.x^2)+(4.x^1)+(2.x^0)$
 $(4.x^2)+(2.x^1)+(3.x^0)$
 $(10.x^2)+(6.x^1)+(5.x^0)$

b) How does the Quicksort technique work? Give C function for the same.

(08)

→ Quick Sort:

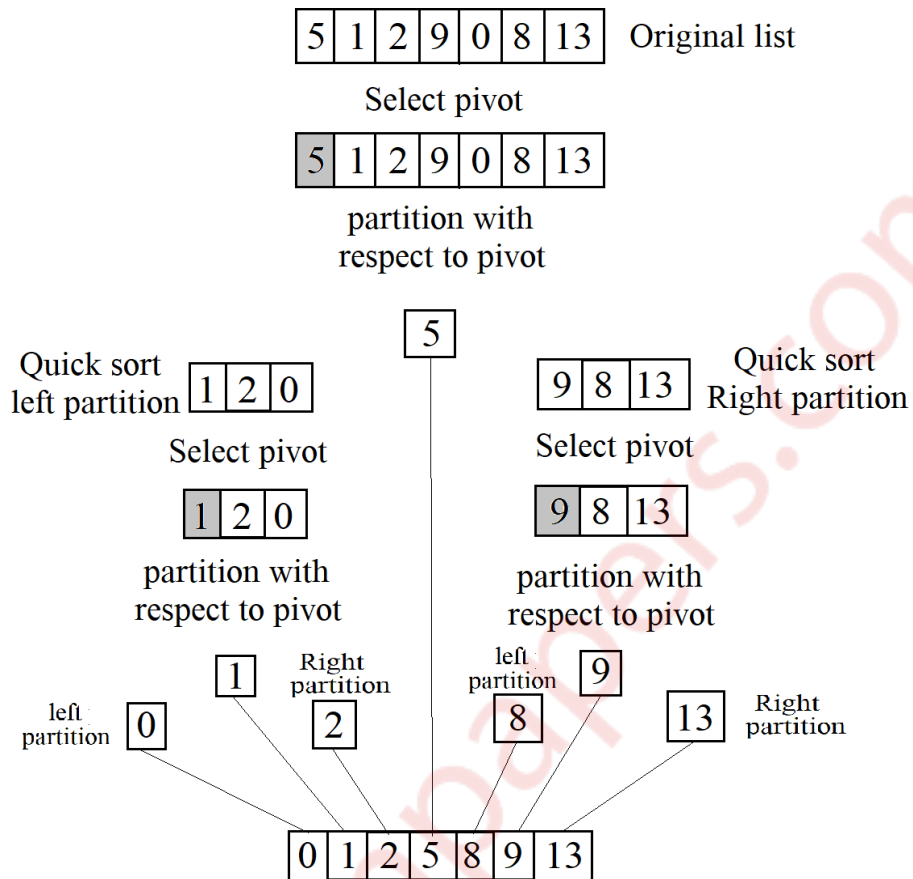
- Quick sort is the fastest internal sorting algorithm with the time and space complexity = $O(n \log n)$.
- The basic algorithm to sort an array $a[]$ of n elements can be describe recursively as follows:

1. If $n \leq 1$, then return
2. Pick any element V in array $a[]$. This element is called as pivot.

Rearrange elements of the array by moving all elements $x_i > V$ right of V and all elements $x_i \leq V$ left of V . if the place of the V after re-arrangement is j , all elements with value less than V , appear in $a[0], a[1] \dots a[j-1]$ and all those with value greater than V appear in $a[j+1] \dots a[n-1]$

3. Apply quick sort recursively to $a[0] \dots a[j-1]$ and to $a[j+1] \dots a[n-1]$
 Entire array will thus be sorted by as selecting an element V .

- a) Partitioning the array around V .
- b) Recursively, sorting the left partition.
- c) Recursively, sorting the right partition.



C function for partition

```
int partition(int a[], int l, int u)
{
    int v,i,j,temp;
    v=a[l];
    i=l;
    j=u+1;
    do;
    {
        do
            i++;
        while(a[i]<v && i<=u);
        do
            j--;
        while(v<a[j]);
        if(i<j)
```

```
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);
    a[l]=a[j];
    a[j]=v;
    return(j);
}
```

C function for Quick Sort

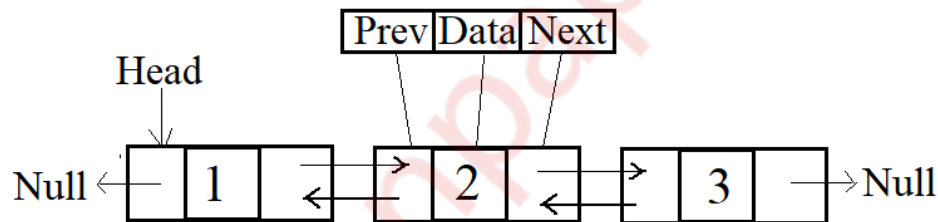
```
void quick_sort(int a[], int l, int u)
{
    int j;
    if(l<u)
    {
        j=partition(a, l, u);
        quick_sort(a, l, j-1);
        quick_sort(a, j+1, u);
    }
}
```

Q 5

a) What is a doubly linked list? Give C representation for the same. (05)

→ Doubly linked list:

1. Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.
2. Every nodes in the doubly linked list has three fields: LeftPointer, RightPointer and Data.
LeftPointer = Left pointer points towards the left node.
RightPointer = Right pointer points towards the right node.
Data = Node which stores the data.
3. The last node has a next link with value NULL, marking the end of the list, and the first node has a previous link with the value NULL. The start of the list is marked by the head pointer.



C Representation of Doubly Linked List

Structure of doubly link list will contain three fields LeftPointer (prev), RightPointer (next), and the data as shown below

```
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

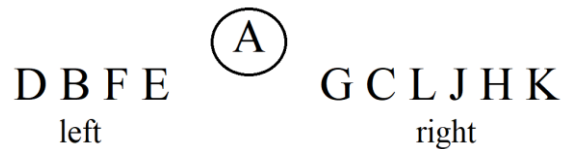
b) Given the postorder and inorder traversal of a binary tree, construct the original tree: (10)

Postorder: D E F B G L J K H C A

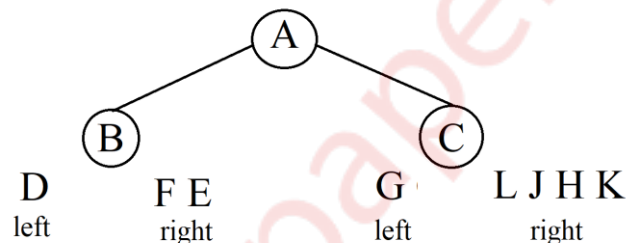
Inorder: D B F E A G C L J H K

→ Construction of Tree:

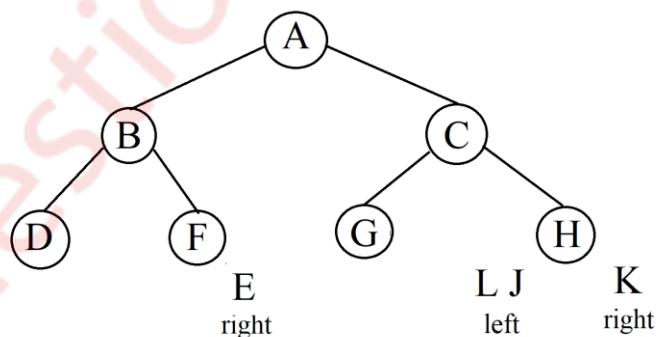
Step1: Select last element from the postorder as root node. So element A becomes root node. Divide the inorder into left and right with respect to root node A.



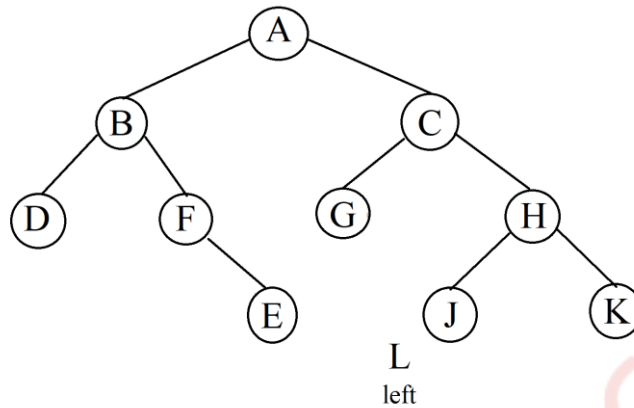
Step 2: Traverse element D B F E from postorder as B comes in last B becomes child node of A and similarly traverse G C L J H K in postorder C comes at last C becomes child node of A. Again with respect to B and C divide element into left and right as shown below



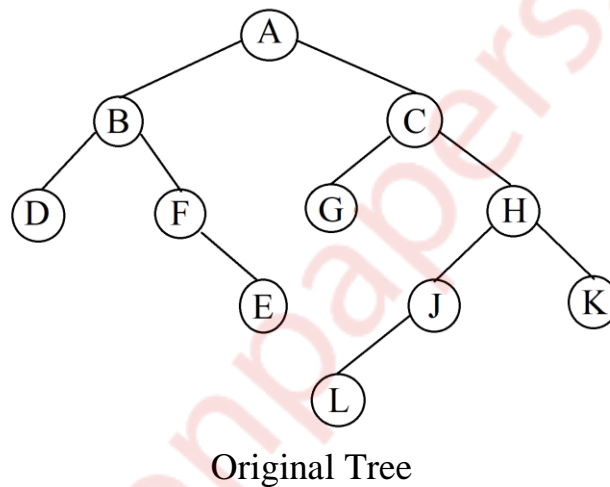
Step 3: As D is single it becomes child node of B and for left node of B Traverse F E in postorder F comes at last so F becomes child node of B. similarly, G and H become child node of C as shown below.



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

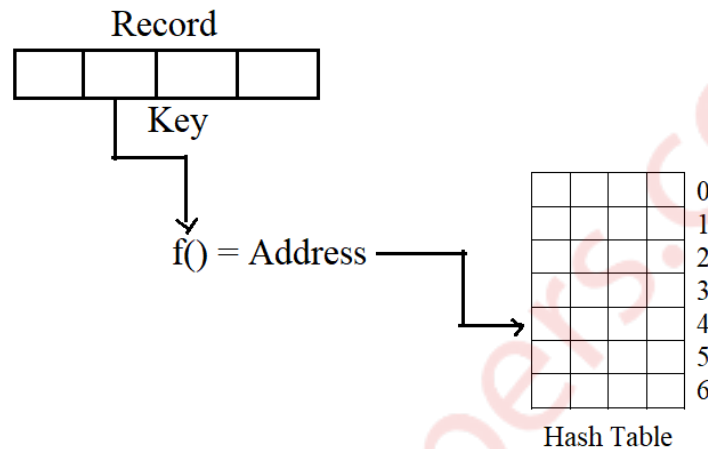


c) **What is hashing? What properties should a good hash function demonstrate?** (05)

→ 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 $f(\text{key})$ 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

Properties of good hash function:

1. A good hash function avoids collisions.
2. A good hash function tends to spread keys evenly in the array.
3. A good hash function is easy to compute.
4. The hash function should generate different hash values for the similar string.
5. The hash function is a perfect hash function when it uses all the input data.

Q 6

a) Given an array `int a[] = {69, 78, 63, 98, 67, 75, 66, 90, 81}`. Calculate address of `a[5]` if base address is 1600. (02)

→

Address	1600	1604	1608	1612	1616	1620	1624	1628	1632
Elements	69	78	63	98	67	75	66	90	81
Array	0	1	2	3	4	5	6	7	8

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

Where, B = Base address 1600 (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

$$\begin{aligned}\text{Address of A [5]} &= 1600 + 4 * (5 - 0) \\ &= 1600 + 4 * 5 \\ &= 1600 + 20 \\ \text{A [5]} &= 1620\end{aligned}$$

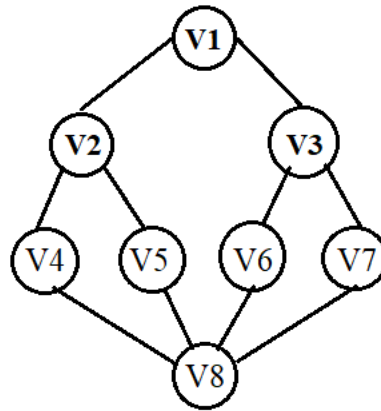
One can verify it from table too A[5] has element 75 stored at address 1620.

b) Give C function for Breadth First Search Traversal of a graph. Explain the code with an example. (10)

→ C function for Breadth First Search

```
void BFS(int V)
{
    q : a queue type variable;
    initialize q;
    visited[v] = 1; // mark v as visited
    add the vertex V to queue q;
    while(q is not empty)
    {
        v ← delete an element from the queue;
        for all vertices w adjacent from V
        {
            if(!visited[w])
            {
                visited[w] = 1;
                add the vertex w to queue q;
            }
        }
    }
}
```

Example:



Queue	Visited[]	Vertex visited	Action																
Null	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	0	0	0	0	0	0	0	0	-	-
1	2	3	4	5	6	7	8												
0	0	0	0	0	0	0	0												
V1	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	1	0	0	0	0	0	0	0	V1	Add (q, v1) visit (V1)
1	2	3	4	5	6	7	8												
1	0	0	0	0	0	0	0												
V2 V3	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	0	0	0	0	0	V1 V2 V3	Delete (q), add and visit adjacent vertices
1	2	3	4	5	6	7	8												
1	1	1	0	0	0	0	0												
V3 V4 V5	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	1	1	0	0	0	V1 V2 V3 V4 V5	Delete (q), add and visit adjacent vertices
1	2	3	4	5	6	7	8												
1	1	1	1	1	0	0	0												
V4 V5 V6 V7	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	1	1	1	1	0	V1 V2 V3 V4 V5 V6 V7	Delete (q), add and visit adjacent vertices
1	2	3	4	5	6	7	8												
1	1	1	1	1	1	1	0												
V5 V6 V7 V8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	1	1	1	1	1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q), add and visit adjacent vertices
1	2	3	4	5	6	7	8												
1	1	1	1	1	1	1	1												
V6 V7 V8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	1	1	1	1	1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q)
1	2	3	4	5	6	7	8												
1	1	1	1	1	1	1	1												
V7 V8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	1	1	1	1	1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q), add and visit adjacent vertices
1	2	3	4	5	6	7	8												
1	1	1	1	1	1	1	1												
V8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	1	1	1	1	1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q)
1	2	3	4	5	6	7	8												
1	1	1	1	1	1	1	1												
Null	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	6	7	8	1	1	1	1	1	1	1	1	V1 V2 V3 V4 V5 V6 V7 V8	Algorithm terminates as the queue is empty
1	2	3	4	5	6	7	8												
1	1	1	1	1	1	1	1												

c) Write a C program to implement a singly linked list. The program should be able to perform the following operations: (08)

i) Insert a node at the end of the list

ii) Deleting a particular element

iii) Display the linked list

→ Program:

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

struct node
{
    int data;
    struct node *next;
} *start=NULL,*q,*t;

int main()
{
    int ch;
    void insert_end();
    int delete_pos();
    void display();

    while(1)
    {
        printf("\n\n---- Singly Linked List(SLL) Menu ----");
        printf("\n1.Insert at end \n2.Delete specific node \n 3.Display \n4.Exit\n\n");
        printf("Enter your choice(1-4):");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1: insert_end();
                    break;
```

```

        case 2: delete_pos();
                break;

        case 3: display();
                break;

        case 4: exit(0);
        default:
                printf("Wrong Choice!!");
    }
}
return 0;
}

```

```

void insert_end()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    t->data=num;
    t->next=NULL;

    if(start==NULL) //If list is empty
    {
        start=t;
    }
    else
    {
        q=start;
        while(q->next!=NULL)
        q=q->next;
        q->next=t;
    }
}

```

```

int delete_pos()

```

```
{
    int pos,i;

    if(start==NULL)
    {
        printf("List is empty!!");
        return 0;
    }

    printf("Enter position to delete:");
    scanf("%d",&pos);

    q=start;
    for(i=1;i<pos-1;i++)
    {
        if(q->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
        }
        q=q->next;
    }

    t=q->next;
    q->next=t->next;
    printf("Deleted element is %d",t->data);
    free(t);

    return 0;
}
```

```
void display()
{
    if(start==NULL)
    {
        printf("List is empty!!");
    }
}
```



```
else
{
    q=start;
    printf("The linked list is:\n");
    while(q!=NULL)
    {
        printf("%d->",q->data);
        q=q->next;
    }
}
```

Output:

---- Singly Linked List(SLL) Menu ----

- 1.Insert at end
- 2.Delete specific node
- 3.Display
- 4.Exit

Enter your choice (1-4): 1

Enter data: 2

---- Singly Linked List(SLL) Menu ----

- 1.Insert at end
- 2.Delete specific node
- 3.Display
- 4.Exit

Enter your choice (1-4): 1

Enter data: 3

---- Singly Linked List(SLL) Menu ----

- 1.Insert at end
- 2.Delete specific node
- 3.Display
- 4.Exit

Enter your choice (1-4): 1

Enter data: 4

---- Singly Linked List(SLL) Menu ----

- 1.Insert at end
- 2.Delete specific node
- 3.Display
- 4.Exit

Enter your choice (1-4): 3

The linked list is:

2 -> 3->4->

---- Singly Linked List(SLL) Menu ----

- 1.Insert at end
- 2.Delete specific node
- 3.Display
- 4.Exit

Enter your choice (1-4): 2

Enter position to delete: 2

Deleted element is 3

---- Singly Linked List(SLL) Menu ----

- 1.Insert at end
- 2.Delete specific node
- 3.Display
- 4.Exit

Enter your choice (1-4): 3

The linked list is:

2 -> 4->

muquestionpapers.com

DATA STRUCTURES

(MAY 2019)

Q.1

(a) Explain Linear and Non-Linear data structures. (5)

→ Linear and Non-linear Structures

Linear: If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

Example:

1. Linked Lists

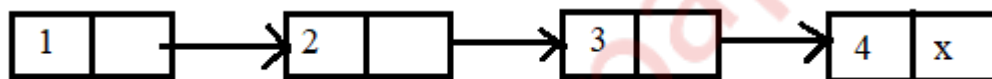


Fig.1

Simple Linked List

2. Stacks

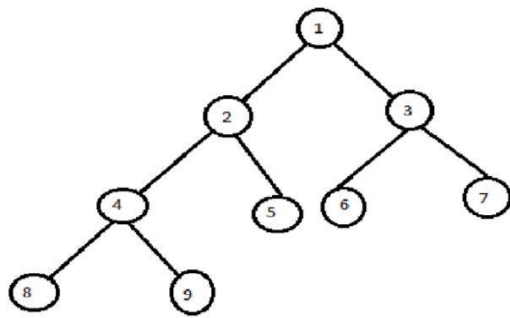
A	AB	ABC	ABCD	ABCDE			
0	1	2	3	TOP=4	5	6	7

Fig.2 Array representation of a stack

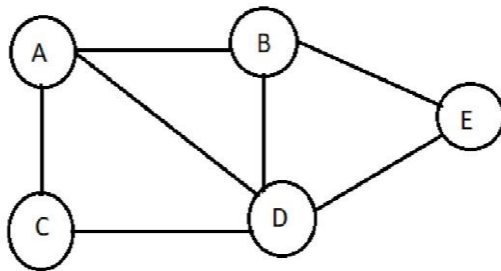
Non-Linear: if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

Example:

1. Trees



2. Graphs



(b) Explain Priority Queue with example. (5)

→ Priority Queue is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.
- A typical priority queue supports following operations.
- `insert(item, priority)`: Inserts an item with given priority.
- `getHighestPriority()`: Returns the highest priority item.
- `deleteHighestPriority()`: Removes the highest priority item.
- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space.
- `FRONT[K]` and `REAR[K]` contain the front and rear values of row `K`, where `K` is the priority number.

Example:

FRONT	REAR
3	3
1	3
4	5
4	1

1	2	3	4	5
1			A	
2	B	C	D	
3				E F
4	I		G	H

Priority Queue matrix

Of an element

FRONT	REAR
3	3
1	3
4	1
4	1

1	2	3	4	5
1			A	
2	B	C	D	
3	R			E F
4	I		G	H

Priority Queue Matrix after insertion

- To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element.
- . In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first.

(c) Write a Programme in 'c' to implement Quick sort. (10)

Program:

```
#include<stdio.h>
#include<conio.h>
void quicksort(int number[25],int first,int last)
{
int i,j,pivot,temp;
if(first<last)
{
pivot=first;
i=first;
j=last;
```

```
while(i<j)
{
while(number[i]<=number[pivot]&& i<last)
i++;
while(number[j]>number[pivot])
j--;
if(i<j)
{
temp=number[i];
number[i]=number[j];
number[j]=temp;

}
}
temp=number[pivot];
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);
}
}
int main()
{
int i,count,number[25];
printf("How many elements are u going to enter?");
scanf("%d",&count);
```

```
printf("enter %d element:",count);
for(i=0;i<count;i++)
scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("Order of sorted elements");
for(i=0;i<count;i++)
scanf("%d",&number[i]);
return 0;
}
```

OUTPUT:

```
How many elements are u going to enter?4
enter 4 element:34
56
22
31
Order of sorted elements
22
31
34
56
```

Q.2

(a) Write a programme to impliment Circular Lined list Provide the following operation: (10)

(i) Insert a node

(ii) Delete a node

(iv) Display the list

Program:

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



```
#include <stdlib.h>
#include <stdbool.h>

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

struct node *head = NULL;
struct node *current = NULL;

bool isEmpty()
{
    return head == NULL;
}

int length()
{
    int length = 0;
    if(head == NULL)
    {
        return 0;
    }
    current = head->next;
    while(current != head)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

```
void insertFirst(int key, int data)
{
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;
    if (isEmpty())
    {
        head = link;
        head->next = head;
    }
    else
    {
        link->next = head;
        head = link;
    }
}

struct node * deleteFirst()
{
    struct node *tempLink = head;
    if(head->next == head)
    {
        head = NULL;
        return tempLink;
    }
    head = head->next;
    return tempLink;
}

void printList()
{

```

```
struct node *ptr = head;
printf("\n[ "); if(head != NULL)
{
while(ptr->next != ptr)
{
printf("(%d,%d) ",ptr->key,ptr->data);
ptr = ptr->next;
}
}
printf(" ]");
}
main()
{
insertFirst(1,10);
insertFirst(2,20);
insertFirst(3,30);
insertFirst(4,1);
insertFirst(5,40);
insertFirst(6,56);
printf("Original List: ");
printList();
while(!isEmpty())
{
struct node *temp = deleteFirst();
printf("\nDeleted value:");
printf("(%d,%d) ",temp->key,temp->data);
}
printf("\nList after deleting all items: ");
printList();
```

}

OUTPUT:

Original List:

[(6,56) (5, 40) (4,1) (3,30) (2,20) (1,10)]

Deleted value: (6,56)

Deleted value: (5, 40)

Deleted value: (4,1)

Deleted value: (3,30)

Deleted value: (2,20)

Deleted value: (1,10)

nList after deleting all items:

[]

(b) Explain Threaded Binary tree in detail

(10)

→ Threaded Binary Tree:

- A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.
- In the linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both.
- For example, the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node.
- **These special pointers are called threads and binary trees containing threads are called threaded trees.**
- There are many ways of threading a binary tree and each type may vary according to the way the tree is traversed.
 - 1. One-way Threading
 - 2. Two-way Threading

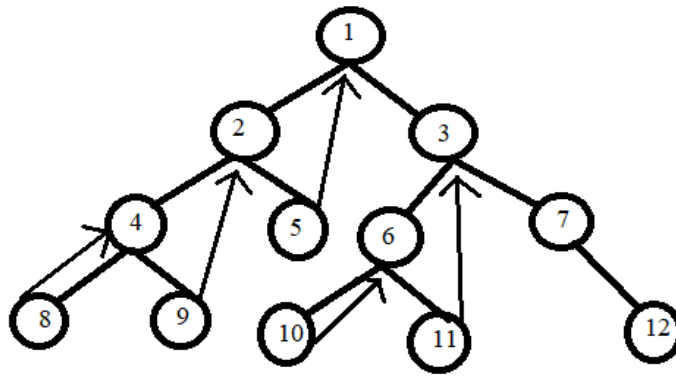


Fig1. Binary tree with one-way threading

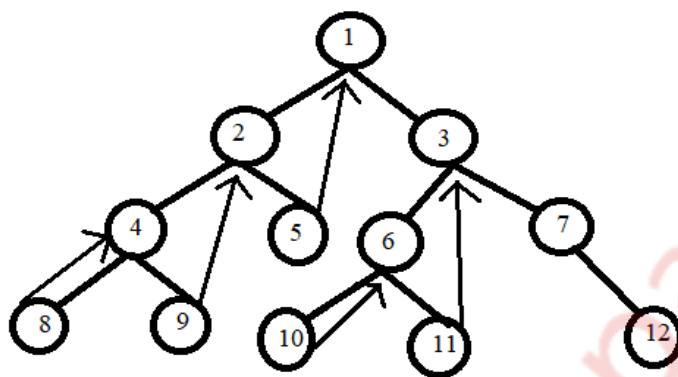


Fig2. Binary tree with two-way threading

- Apart from this, a threaded binary tree may correspond to one-way threading or a two-way threading.
- In one-way threading, a thread will appear either in the right field or the left field of the node.
- A one-way threaded tree is also called a single-threaded tree.
- one-way threaded tree is called a right-threaded binary tree.
- In a two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node.
- A two-way threaded binary tree is also called a fully threaded binary tree.
- Advantages of Threaded Binary Tree:
 1. It enables linear traversal of elements in the tree.
 2. Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
 3. It enables to find the parent of a given element without explicit use of parent pointers.

4. Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.
- we see the basic difference between a binary tree and a threaded binary tree is that in binary trees a node stores a NULL pointer if it has no child and so there is no way to traverse back.

Q.3

(a) Explain Huffman Encoding with suitable example (10)

Huffman Code:

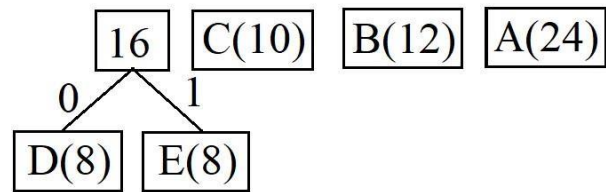
- Huffman code is an application of binary trees with minimum weighted external path length is to obtain an optimal set for messages M_1, M_2, \dots, M_n
- Message is converted into a binary string.
- Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication 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.
- The tree is called encoding tree and is present at the sending end. - The decoding tree is present at the receiving end which decodes the string to get corresponding message.
- The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree. Example

Symbol	A	B	C	D	E
Frequency	24	12	10	8	8

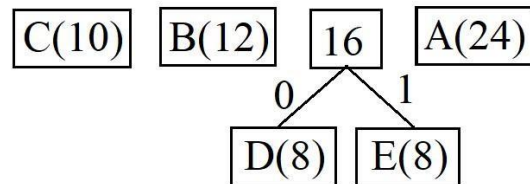
Arrange the message in ascending order according to their frequency

D(8) E(8) C(10) B(12) A(24)

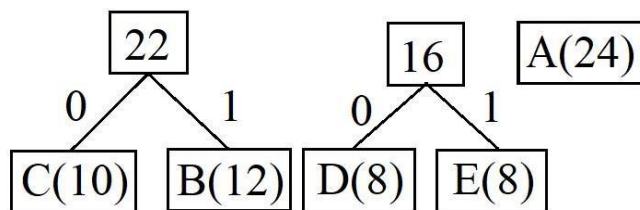
Merge two minimum frequency message



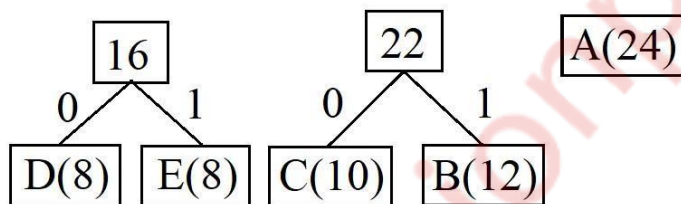
Rearrange in ascending order



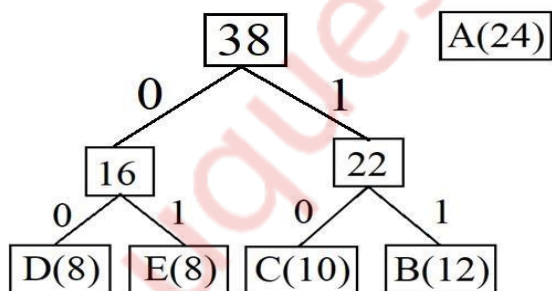
Merge two minimum frequency message



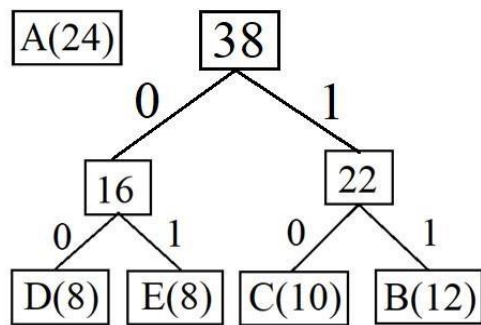
Rearrange in ascending order



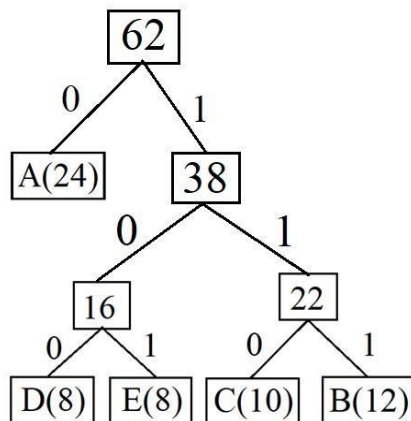
Merge two minimum frequency message



Again Rearrange in ascending order



Merge two minimum frequency message



Huffman code

A = 0

B = 111

C = 110

D = 100

E = 101

(b) Write a program in 'C' to check for balanced parenthesis in an expression using stack. (10)

→ Program

```

#include <stdio.h>
#include <string.h>
#define MAXSIZE 100
#define TRUE 1
#define FALSE 0
struct Stack {
  int top;

```



```
int array[MAXSIZE];
} st;
void initialize() {
st.top = -1;
}
int isFull() {
if(st.top >= MAXSIZE-1)
return TRUE;
else
return FALSE;
}
int isEmpty() {
if(st.top == -1)
return TRUE;
else
return FALSE;
}
void push(int num) {
if (isFull())
printf("Stack is Full...\n");
else {
st.array[st.top + 1] = num;
st.top++;
}
}
int pop() {
if (isEmpty())
printf("Stack is Empty...\n");
else {
st.top = st.top - 1;
return st.array[st.top+1];
}
}
int main() {
char inputString[100], c;
int i, length;
initialize();
printf("Enter a string of paranthesis\n");
gets(inputString);
length = strlen(inputString);
for(i = 0; i < length; i++){
if(inputString[i] == '{')
push(inputString[i]);
```

```

else if(inputString[i] == '}')
pop();
else {
printf("Error : Invalid Character !! \n");
return 0;
}
}

if(isEmpty())
printf("Valid Paranthesis Expression\n");
else
printf("InValid Paranthesis Expression\n");

return 0;
}

```

OUTPUT:

```

Enter a string of paranthesis
{{{}}}{{{}}}
Valid Paranthesis Expression

Enter a string of paranthesis
{{{}}}{}{}{}{}
InValid Paranthesis Expression

```

Q.4

(a) Write a program in 'C' to implement Queue using array. (10)

➔ Program

```

#include<stdio.h>

#include<conio.h>
#define size 5

int q[size],front=-1,rear=-1,i,element;

void insert(int ele);

int del();

void disp();

```

```

void main()

{
    int
    ch,ele;
    clrscr();

    printf("\t ***** Main Menu *****");

    printf("\n 1. insert \n2.delete \n3.display
    \n4.Exit\n"); do {

    printf("\n Enter your choice: \n \n");

    scanf("%d",&ch);
    switch(ch)

    {

    case 1:printf("\n Enter Element to Insert \n ");
    scanf("%d",&ele)
    ;   insert(ele);
    disp();   break;

    case 2:ele=del();
    scanf("\n %d is the deleted element \n ",ele);

    disp();
    break;

    case 3:disp();
    break;

    case 4:break;
    default:printf("\n Invalid Statement \n");

    }

    }

    while(ch!=4); getch();

    }

void insert(int ele)
{

if(front== -1 && rear== -1)

```

```

{
front=rear=0;  q[rear]=ele;
}
else if((rear+1)%size==front)
{
printf("\n Queue is Full \n");
}
else
{
rear=(rear+1)%size;
q[rear]=ele;
}
}

int del()
{
if(rear== -1 && front== -1)
{
printf("\n Queue is Empty \n");
}
else if(rear==front)
{ rear=-
1;
front=-1;
printf("\n Queue is Empty \n");
}
else
{
element=q[front];  front=(front+1)%size;
} return
element;

```

```

    }
    void disp()
    {
        if(rear== -1 && front== -1)
        {
            printf("\n Queue is Empty \n");
        }
        else
        {
            for(i=front;i<(rear+1)%size;i++)
            {
                printf("\t %d",q[i]);
            }
        }
    }
}

```

OUTPUT:

***** Main Menu *****

1. insert

2.delete

3.display

4.Exit

Enter your choice: 1

Enter Element to Insert: 23

Enter your choice: 1

Enter Element to Insert: 45

Enter your choice: 2

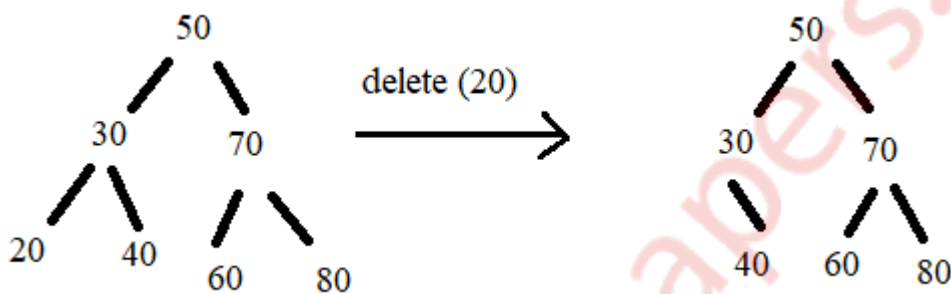
23 is the deleted element

Enter your choice: 3

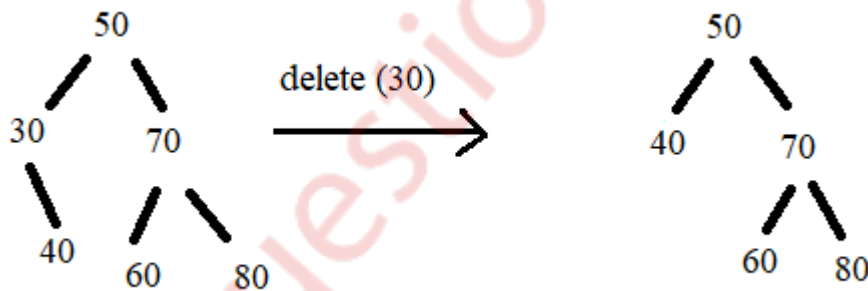
(b) Explain different cases for deletion of a node in binary search tree. Write function for each case (10)

→ When we delete a node, three possibilities arise.

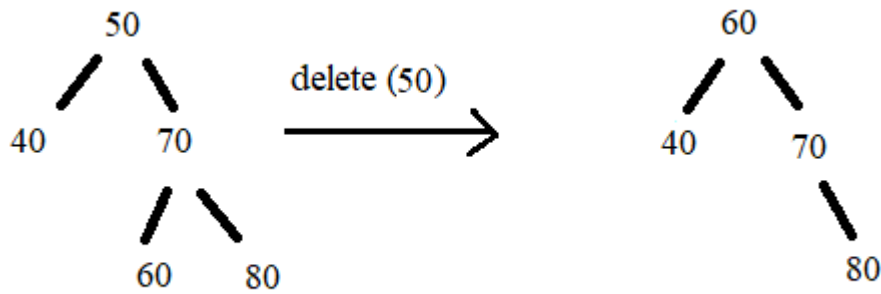
1) *Node to be deleted is leaf:* Simply remove from the tree.



2) *Node to be deleted has only one child:* Copy the child to the node and delete the child



3) *Node to be deleted has two children:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



- The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.
- C Function:

```

void deletion(Node*& root, int item)
{
    Node* parent = NULL;
    Node* cur = root;
    search(cur, item, parent);
    if (cur == NULL)
        return;
    if (cur->left == NULL && cur->right == NULL)
    {
        if (cur != root)
        {
            if (parent->left == cur)
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        else
            root = NULL;
        free(cur);
    }
    else if (cur->left && cur->right)
    {
        Node* succ = findMinimum(cur->right);
        int val = succ->data;
        deletion(root, succ->data);
        cur->data = val;
    }
}
  
```

```

else
{
Node* child = (cur->left)? Cur- >left: cur->right;
if (cur != root)
{
if (cur == parent->left)
parent->left = child;
else
parent->right = child;
}
else
root = child;
free(cur);
}
}

Node* findMinimum(Node* cur)
{
while(cur->left != NULL) {
cur = cur->left;
}
return cur;
}

```

Q.5

(a) Write a program in 'C' to implement Stack using Linked-List. Perform the following operation:

(i) Push

(ii) Pop

(iii) Peek

(iii) Display the stack contents

(10)

PROGRAM:

```

#include <stdio.h>
#include <stdlib.h>

struct node

```



```
{
int info;
struct node *ptr;
}*top,*top1,*temp;
int topelement();
void push(int data);
void pop();
void display();
void peek();
void create();
int count = 0;
void main()
{
int no, ch, e;
printf("\n 1 - Push");
printf("\n 2 - Pop");
printf("\n 3 - Peek");
printf("\n 4 - Exit");
printf("\n 5 - Dipslay");
create();
while (1)
{
printf("\n Enter choice : ");
scanf("%d", &ch);
switch (ch)
{
case 1:
printf("Enter data : ");
scanf("%d", &no);
```

```
push(no);
break;
case 2:
pop();
break;
case 3:
if (top == NULL)
printf("No elements in stack");
else
{
e = topelement();
printf("\n Top element : %d", e);
}
break;
case 4:
exit(0);
case 5:
display();
break;
default :
printf(" Wrong choice, Please enter correct choice ");
break;
}
}
}
void create()
{
top = NULL;
}
```

```
void push(int data)
{
if (top == NULL)
{
top =(struct node *)malloc(1*sizeof(struct node));
top->ptr = NULL;
top->info = data;
}
else
{
temp =(struct node *)malloc(1*sizeof(struct node));
temp->ptr = top;
temp->info = data;
top = temp;
}
count++;
}

void display()
{
top1 = top;
if (top1 == NULL)
{
printf("Stack is empty");
return;
}
while (top1 != NULL)
{
printf("%d ", top1->info);
top1 = top1->ptr;
```

```
}  
}  
void peek()  
{  
    top1 = top;  
    if (top1 == NULL)  
    {  
        printf("\n Error : Trying to pop from empty stack");  
        return;  
    }  
    else  
        top1 = top1->ptr;  
    printf("\n Popped value : %d", top->info);  
    free(top);  
    top = top1;  
    count--;  
}  
int topelement()  
{  
    return(top->info);  
}
```

OUTPUT:

```
1- Push
2- Pop
3- Peek
4- Display
5- Exit
Enter choice : 1
Enter data : 32

Enter choice : 1
Enter data : 34

Enter choice : 2
Popped value : 34

Enter choice : 3
Peek element : 32

Enter choice : 4
32

Enter choice : 5
```

**(b) Explain Depth First search (DFS) Traversal with an example.
Write the recursive function for DFS (10)**

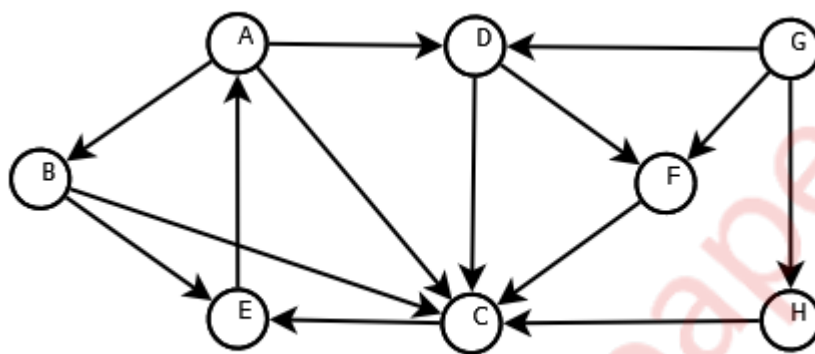
→Depth-first Search

- The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- depth-first search begins at a starting node A which becomes the current node.
- Then, it examines each node N along a path P which begins at A.
- That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.

Algorithm for depth-first search

- Step 1: SET STATUS=1(ready state) for each node in G
- Step 2: Push the starting node A on the stack and set its STATUS=2(waiting state)
- Step 3: Repeat Steps 4 and 5 until STACK is empty
- Step 4: Pop the top node N. Process it and set its STATUS=3(processed state)
- Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2(waiting state) [END OF LOOP]
- Step 6: EXIT

Example:



- Adjacency list for G:

- A: B, C, D
- B: C, E
- C: E
- D: C, F
- E: A
- F: C
- G: D, F, H
- H: C

Recursive Function:

```

void Graph::DFSUtil(int v, bool visited[])
{
    visited[v] = true;
    cout << v << " ";

```

```

// Recur for all the vertices adjacent
// to this vertex

```

```

list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
if (!visited[*i])
DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
// Mark all the vertices as not visited
bool *visited = new bool[V];
for (int i = 0; i < V; i++)
visited[i] = false;

// Call the recursive helper function
// to print DFS traversal
DFSUtil(v, visited);
}

```

Q.6. Write Short notes on (any two) (20)

(a) Application of Linked-List –Polynomial addition

➔ Application of Linked list

- Linked lists can be used to represent polynomials and the different operations that can be performed on them
- we will see how polynomials are represented in the memory using linked lists.

1. 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.

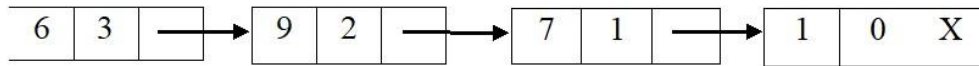


Figure. Linked representation of a polynomial

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

Input:

1st number = $5x^2 + 4x^1 + 2x^0$

2nd number = $5x^1 + 5x^0$

Output:

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

Input:

1st number = $5x^3 + 4x^2 + 2x^0$

2nd number = $5x^1 + 5x^0$

Output:

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

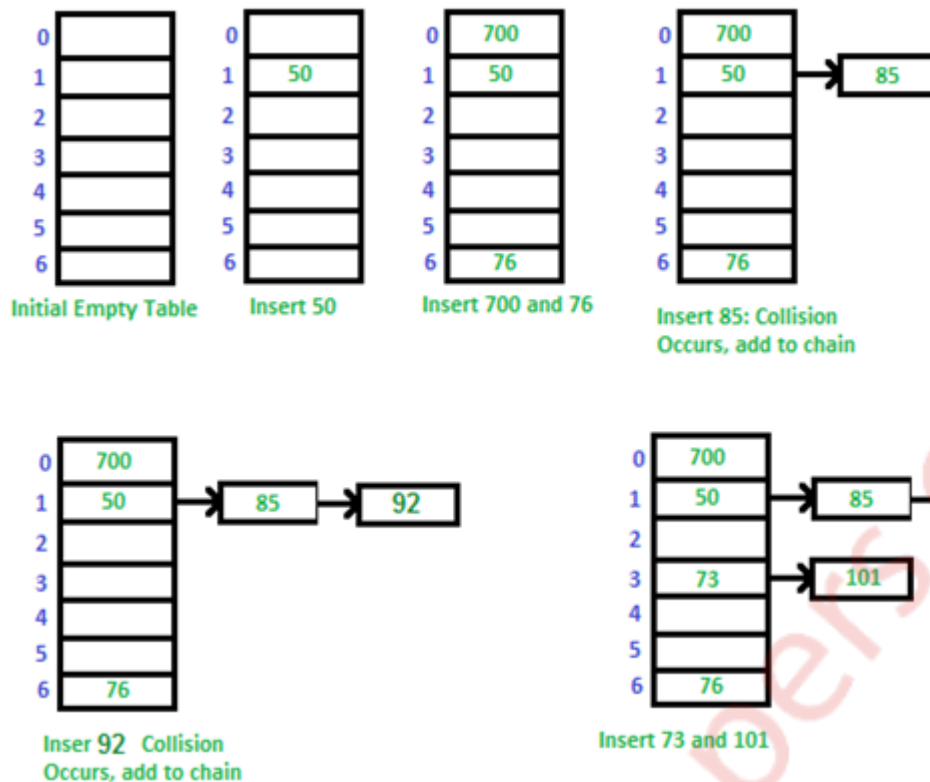
(b) Collision Handling technique

→ There are mainly two methods to handle collision:

- 1) Separate Chaining
- 2) Open Addressing

1. Separate Chaining

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
- To handle collisions, the hash table has a technique known as **separate chaining**. **Separate chaining** is defined as a method by which linked lists of values are built in association with each location within the hash table when a collision occurs.
- The concept of separate chaining involves a technique in which each index key is built with a linked list. This means that the table's cells have linked lists governed by the same hash function.
- Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



- **Advantages:**

- 1) Simple to implement.
- 2) Hash table never fills up, we can always add more elements to chain.
- 3) Less sensitive to the hash function or load factors.
- 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

2. Open Addressing

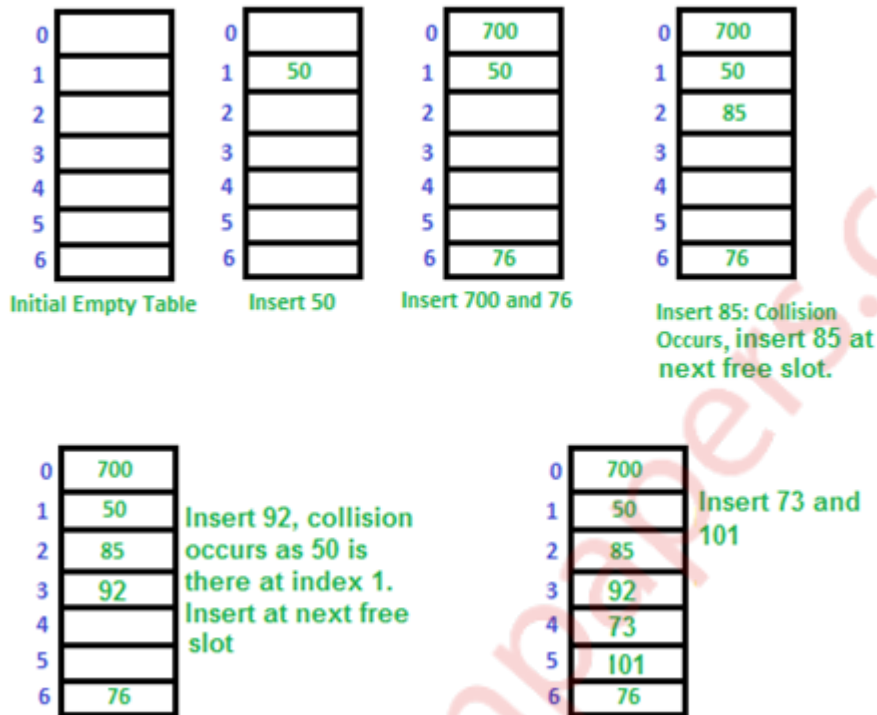
- Like separate chaining, open addressing is a method for handling collisions.
- In Open Addressing, all elements are stored in the hash table itself.
- So at any point, size of the table must be greater than or equal to the total number of keys
- In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell.
- This technique is called linear probing.

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let **hash(x)** be the slot index computed using hash function and **S** be the table size

a) Quadratic Probing: Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

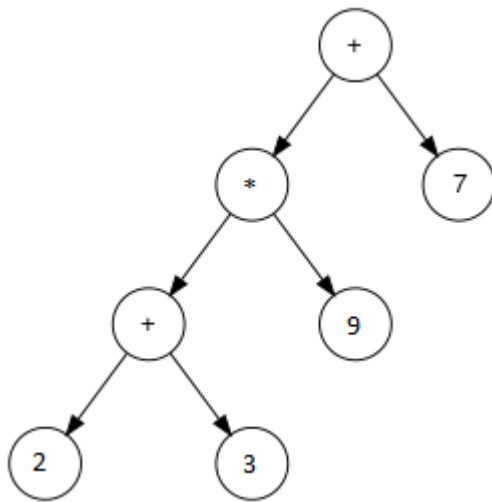
- Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



(c) Expression Tree

→ Expression Tree

- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.
- A binary expression tree is a specific kind of a binary tree used to represent expressions.
- The leaves of the binary expression tree are operands, such as constants or variable names, and the other nodes contain operators.
- Assume the set of possible operators are $\{ '+', '-', '*', '/' \}$. The set of possible operands are $['0' - '9']$. See the figure below, which is the binary expression tree for the expression (in in-fix notation): $((2+3)*9)+7$.



- **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

- **Algorithm For Expression tree**

Let t be the expression tree

If t is not null then

If t.value is operand then

Return t.value

A = solve(t.left)

B =solve(t.right)

// calculate applies operate 't.value'

// on A and B, and returns value

Return calculate (A, B, t.value)

(d) 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 $\text{INDEG}(N)$ of every node in the graph

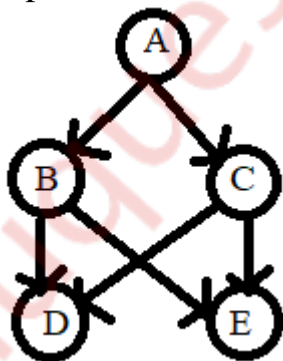
Step 2: Enqueue all the nodes with zero in-degree

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

Step 4: Remove the front node N of the QUEUE by setting $\text{FRONT} = \text{FRONT} + 1$

Step 5: Repeat for each neighbour M of node N : a) Delete the edge from N to M by setting $\text{INDEG}(M) = \text{INDEG}(M) - 1$ b) IF $\text{INDEG}(M) = 0$, 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

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;
```

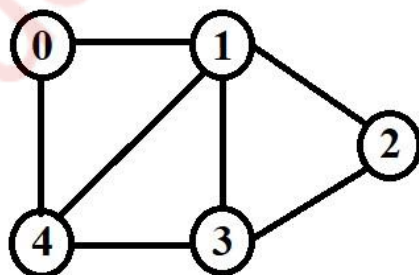
```
    }
}
```

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.

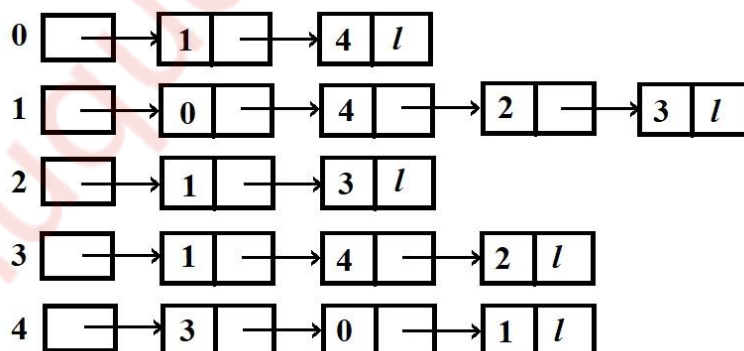
1. Adjacency Matrix

- 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.

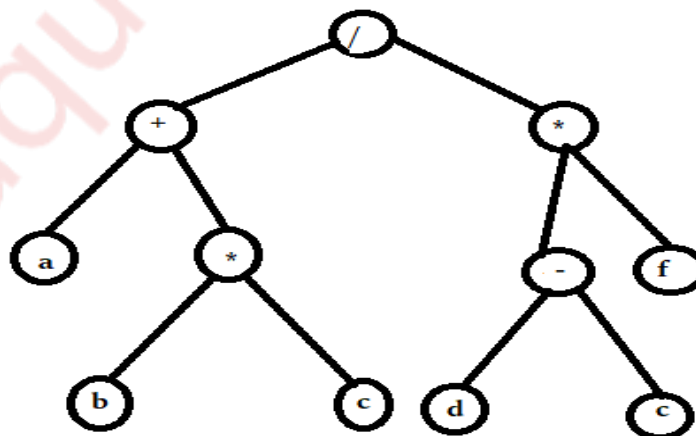
- 1) If character is operand push that into stack
- 2) If character is operator pop two values from stack make them its child and push current node again.

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

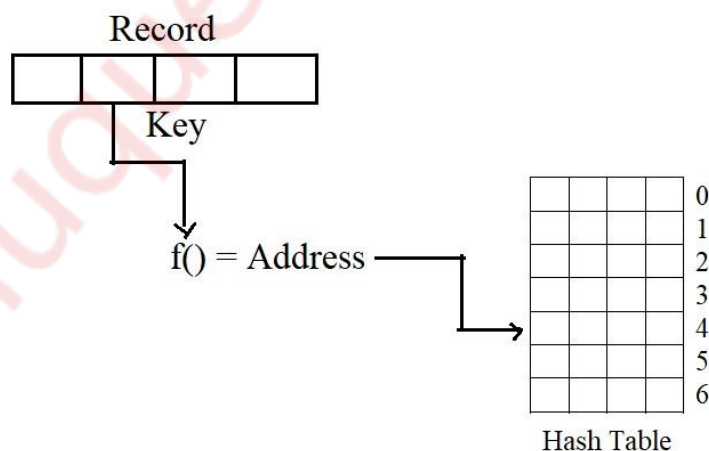
a) What is hashing? Hash the following data in table of size 10 using linear probing and quadratic probing. Also find the number of collisions.

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 $h(k) = k \bmod \text{size}$, 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 `int a[]={23, 55, 63, 89, 45, 67, 85, 99}`. Calculate address of `a[5]` if base address is 5100. (2)



Address	5100	5104	5108	5112	5116	5120	5124	5128
Elements	23	55	63	89	45	67	85	99
Array	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:

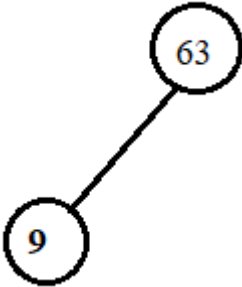
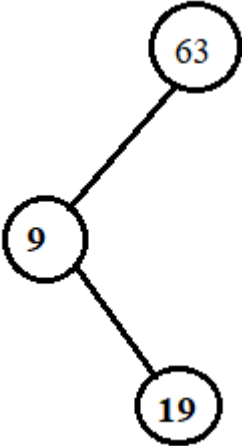
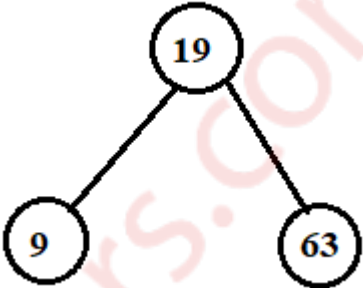
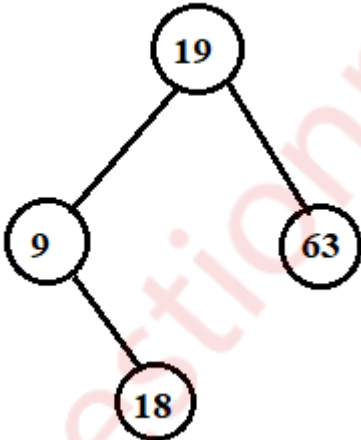
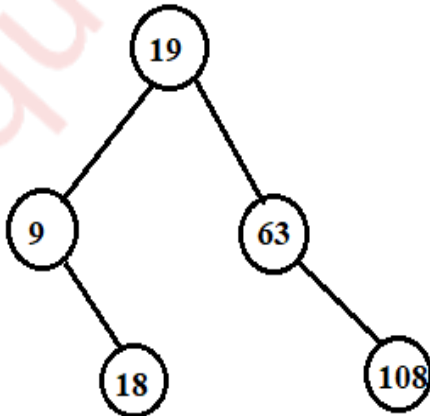
Enter the expression :: a+b*c
abc*+

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

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

(10)

Sr n o	Data to be inser t	Tree after insertion	Tree after rotation
1	63		

2	9		
3	19		
4	18		
5	108		

6	81	<pre> graph TD 19((19)) --> 9((9)) 19 --> 63((63)) 9 --> 18((18)) 63 --> 108((108)) 108 --> 81((81)) </pre>	
7	45	<pre> graph TD 19((19)) --> 9((9)) 19 --> 63((63)) 9 --> 18((18)) 63 --> 108((108)) 108 --> 81((81)) 81 --> 45((45)) </pre>	<pre> graph TD 19((19)) --> 9((9)) 19 --> 63((63)) 9 --> 18((18)) 63 --> 81((81)) 81 --> 45((45)) 81 --> 108((108)) </pre>

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

void main()
{
int x, pos, ch, i;
List l, l1;
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("\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 == l)
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.

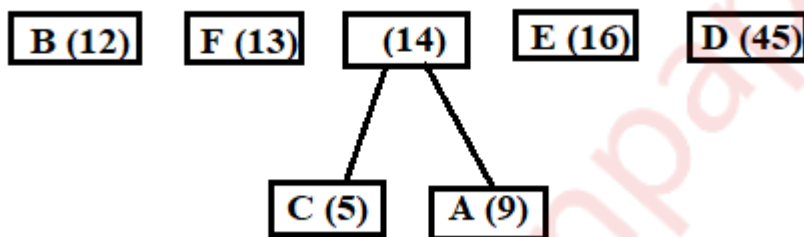
- The tree is called encoding tree and is present at the sending end.
- The decoding tree is present at the receiving end which decodes the string to get corresponding message.
- The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree.
- Example

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

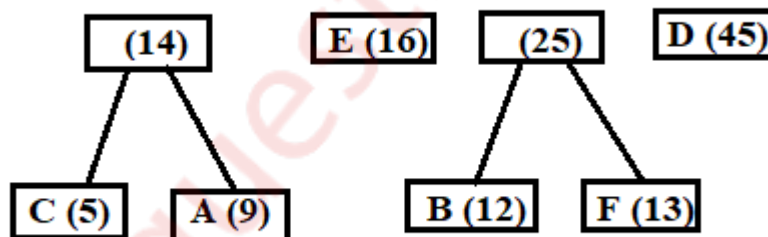
Arrange the message in ascending order according to their frequency

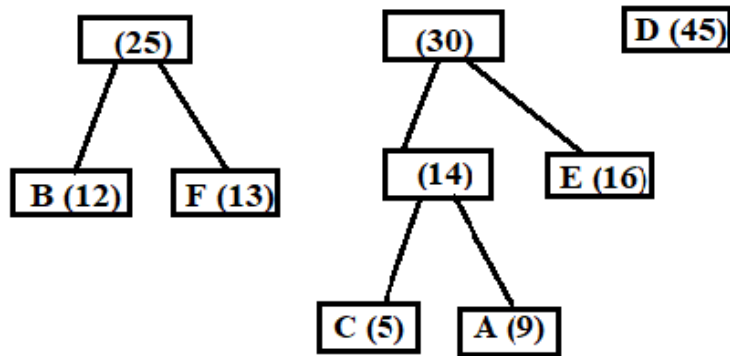


Merge two minimum frequency message

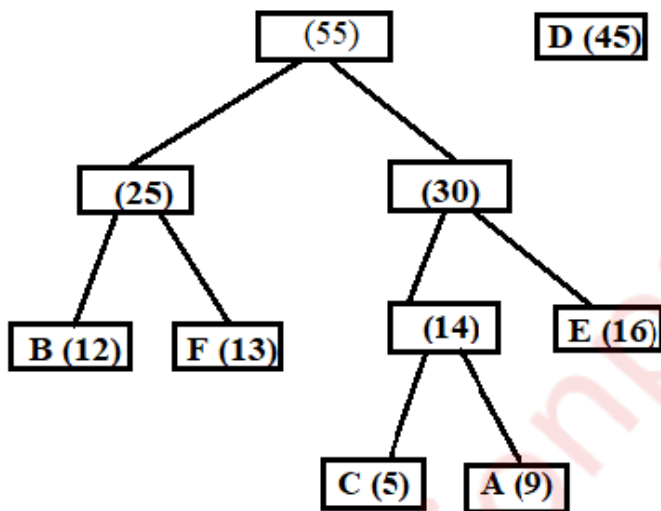


Merge two minimum frequency message

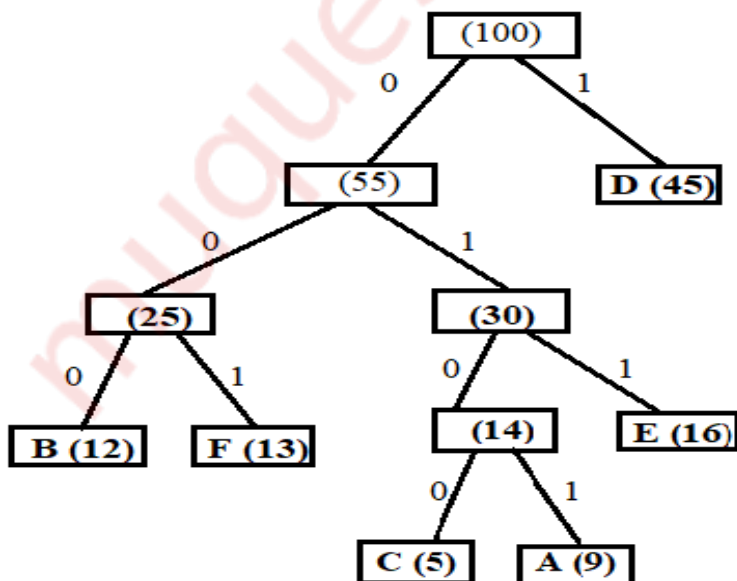




Merge two minimum frequency message



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 at element at the front end of the queue.

v. enqueueR(): Insert at 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

5.Display
6.Exit

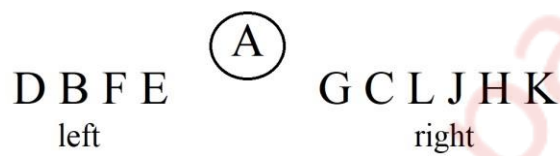
b) Given the postorder and inorder traversal of binary tree, construct the original tree: (10)

Postorder: D E F B G L J K H C A

Inorder: D B F E A G C L J H K

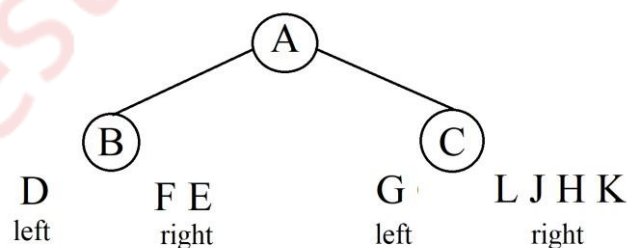
→ Construction of Tree:

Step1: Select last element from the postorder as root node. So element A becomes root node. Divide the inorder into left and right with respect to root node A.

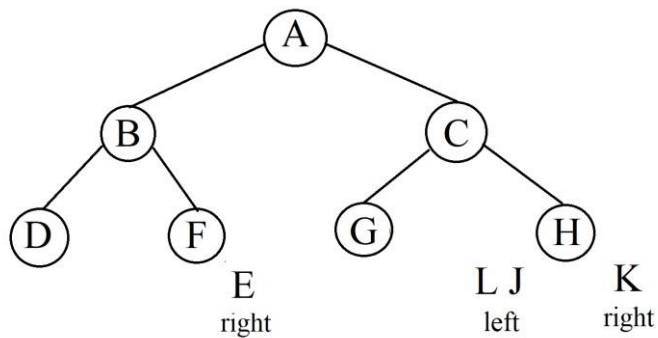


Step 2: Traverse element D B F E from postorder as B comes in last B becomes child node of A and similarly traverse G C L J H K in postorder

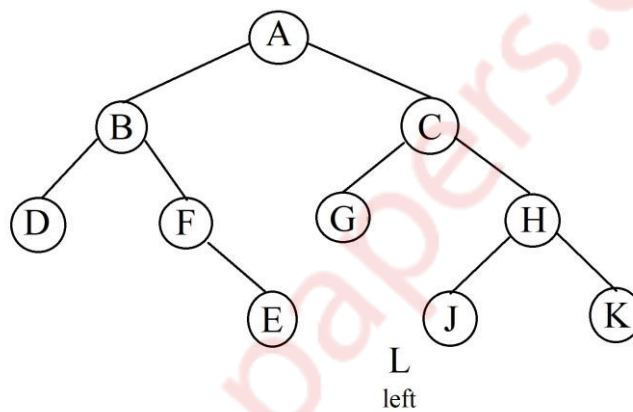
C comes at last C becomes child node of A. Again with respect to B and C divide element into left and right as shown below



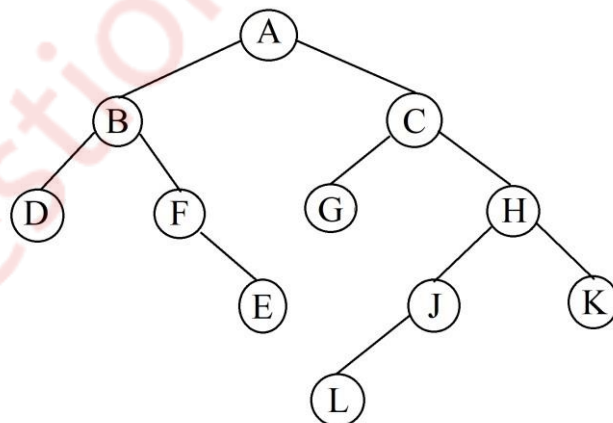
Step 3: As D is single it becomes child node of B and for left node of B Traverse F E in postorder F comes at last so F becomes child node of B. similarly, G and H become child node of C as shown below.



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

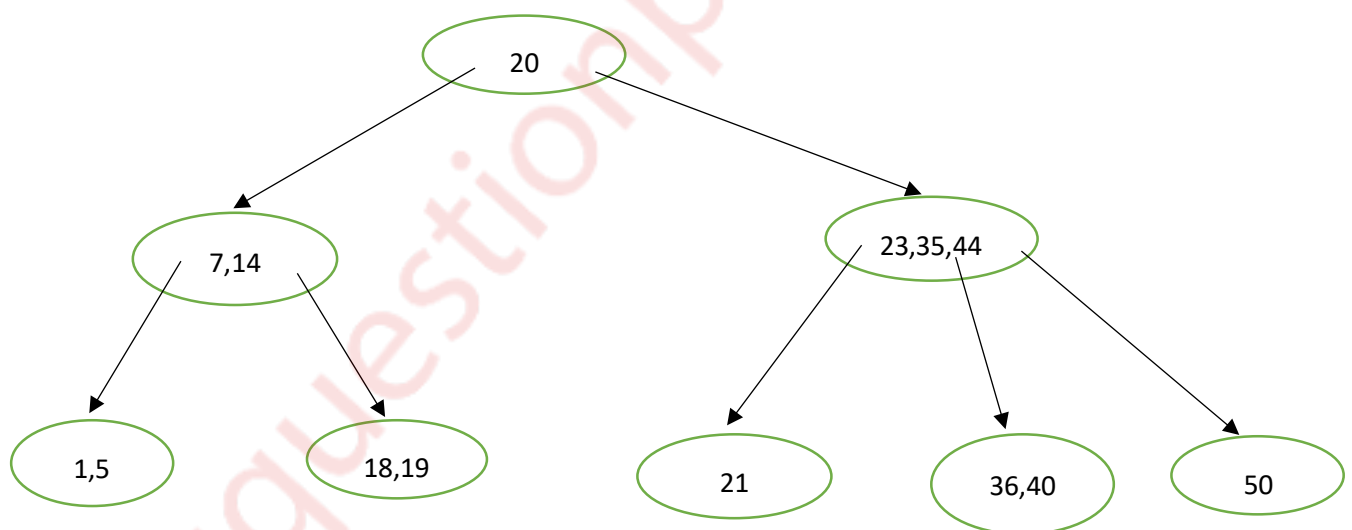
->A B-tree is a method of placing and locating files (called records or keys) in a database.

->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.

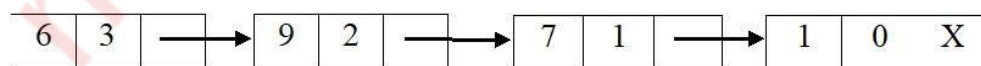


Figure. Linked representation of a polynomial

- 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 $\text{INDEG}(N)$ 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

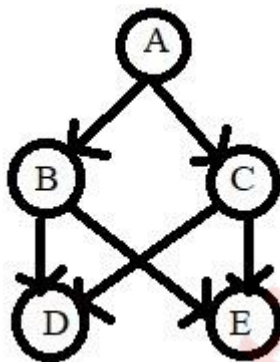
$\text{FRONT} = \text{FRONT} + 1$

Step 5: Repeat for each neighbour M of node N: a) Delete the edge from N to M by setting $\text{INDEG}(M) = \text{INDEG}(M) - 1$ b) IF $\text{INDEG}(M) = 0$, 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