Assignment 1

Problem Statement –

Construct an expression tree from postfix expression and perform recursive Inorder, Preorder, Postorder traversals

Theory and Applications –

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

Inorder Traversal:

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversals reversed can be used.

Preorder Traversal:

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree

Postorder Traversal:

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Uses of Postorder
Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

Expected Input Type and Range –

- Type - Alphabets and Symbols
- Range – a to z and {+ , - , * , / , ^ }

Expected Output Type and Range –

- Type - String of Alphabets and symbols mentioned above
- Range - a to z and {+ , - , * , / , ^ }

Code –

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;
class expNode
{
public:
    char data;
    expNode *left,*right;
    expNode* accept(char s)
    {
        expNode* newNode=new expNode;
        newNode->data=s;
        newNode->left=newNode->right=nullptr;
        return newNode;
    }
    friend class stack;
};

class stackNode
{
    stackNode *next;
    expNode *val;
public:
    friend class stackptr;
};

class stackptr
{
    stackNode *top;
public:
    void push(expNode *node)
    {
        if(top==NULL)
        {
            stackNode *s=new stackNode;
            s->val=node;
            s->next=nullptr;
            top=s;
```

```
        }
        else
        {
            stackNode *s=new stackNode;
            s->val=node;
            s->next=top;
            top=s;
        }
    }
    expNode *pop()
    {
        stackNode *temp=top;;
        expNode *tn=temp->val;
        top=top->next;
        temp->next=nullptr;
        delete temp;
        return tn;
    }

};

bool isop(char c)
{
    if(c == '+' || c == '-' || c == '/' || c == '*' || c == '^')
        return true;
    else
        return false;
}


void inorder(expNode *t)
{
    if(t)
    {
        inorder(t->left);
        cout<<t->data<<" ";
        inorder(t->right);
    }
}

void preorder(expNode *t)
{
    if(t)
    {
        cout<<t->data<<" ";
        preorder(t->left);
        preorder(t->right);
    }
```

```
}

void postorder(expNode *t)
{
    if(t)
    {
        postorder(t->left);
        postorder(t->right);
        cout<<t->data<<" ";
    }
}


int main()
{

    char pf[100];
    stackptr s1;
    expNode *tr,*tl,*n1;
    cout<<"Enter postfix code "<<endl;
    cin>>pf;
    for(int i=0;i<strlen(pf);i++)
    {
        n1=n1->accept(pf[i]);

        if(!isop(pf[i]))
        {
            s1.push(n1);
        }

        else
        {
            tr=s1.pop();
            tl=s1.pop();
            n1->right=tr;
            n1->left=tl;
            s1.push(n1);
        }

    }

    n1=s1.pop();

    cout<<"Preorder is  - \t";
    preorder(n1);
    cout<<"\nInorder is  - \t";
    inorder(n1);
    cout<<"\nPostorder is  - \t";
```

```
    postorder(n1);
}
```

DSF_1 (Build, Run) × DSF_1 (Run) ×

```
Enter postfix code
abc-_de-fg-h+/*
Preorder is  -  * _ / - d e + - f g h
Inorder is  -   _ * d - e / f - g + h
Postorder is  -         _ d e - f g - h + / *
RUN SUCCESSFUL (total time: 25s)
```

Assignment 2

Problem statement : Construct an expression tree from postfix expression and perform non-recursive Inorder and Preorder traversals

Theory:

Expression Tree :

An expression tree is a representation of expressions arranged in a tree-like data structure. In other words, it is a tree with leaves as operands of the expression and nodes contain the operators.

Postfix expression :

Postfix notation is a notation for writing arithmetic expressions in which the operands appear before their operators.

Prefix expression :

Prefix notation is a notation for writing arithmetic expressions in which the operators appear before their operands.

Infix expression :

***Infix notation* is a notation for writing arithmetic expressions in which the operands appear along with operands.**

Expected i/p type :

Postfix expression

Expected o/p type :

Required infix,prefix and postfix expressions

Program :

```
#include<cstring>
#include<iostream>
using namespace std;

class treeNode
{
        char exp;
        treeNode *left,*right;
        public:
        treeNode()
        {
                exp='0';
                left=NULL;
                right=NULL;
        }
        treeNode *assign(char x,treeNode *l,treeNode *r)
        {
                treeNode *newnode;
                newnode=new treeNode;
                newnode->exp=x;
                newnode->left=l;
                newnode->right=r;
                return(newnode);
        }
```

```
        friend class stack;
};

class node
{
        friend class stack;
        treeNode *data;
        node *next;
};

class stack
{
        node *top;
        public:
        stack()
        {
                top=NULL;
        }

        void push(treeNode *x)
        {
                node *nn,*temp;
                nn=new node;
                nn->data=x;
                if(top!=NULL)
           nn->next=top;
           top=nn;
        }

        treeNode *pop()
        {
        node *temp;
           treeNode *topData;
        temp=top;
           topData=temp->data;
           top=top->next;
           delete(temp);
           return(topData);
         }

        int empty()
        {
                if(top==NULL)
                return 1;
                else
                return 0;
        }

        void maketree(char x)
        {
                treeNode *nn;
                if((x=='+')||(x=='-')||(x=='*')||(x=='/'))
                {
                        nn=nn->assign(x,pop(),pop());
```

```
                push(nn);
        }
        else
        {
                nn=nn->assign(x,NULL,NULL);
                push(nn);
        }
}

void inorder(treeNode *root)
{
        while(root)
        {
                push(root);
                root=root->left;
        }
        while(!empty())
        {
                root=pop();
                cout<<root->exp;
                if(root->right)
                {
                        root=root->right;
                        cout<<root->exp;
                }
        }
}

void preorder(treeNode *root)
{
        while(root)
        {
                push(root);
                cout<<root->exp;
                root=root->left;
        }
        while(!empty())
        {
                root=pop();
                if(root->right)
                {
                        root=root->right;
                        cout<<root->exp;
                }
        }
}

void postorder(treeNode *root)
{
        while(root)
        {
                push(root);
                root=root->left;
        }
        while(!empty())
```

```
		{
			root=pop();
			if(root->right)
			{
				push(root);
				root=root->right;
				cout<<root->exp;
				root=pop();
			}
			cout<<root->exp;
		}
	}

	void traverse()
	{
		treeNode *root=pop();
		int ch;
		char c;
		do
		{
		cout<<"Enter\n1.Inorder\n2.Preorder\n3.Postorder"<<endl;
		cin>>ch;
		switch(ch)
		{
			case 1:
			{
			inorder(root);
			cout<<endl;
			}
			break;
			case 2:
			{
			preorder(root);
			cout<<endl;
			}
			break;
			case 3:
			{
			postorder(root);
			cout<<endl;
			}
			break;
			default:
			cout<<"Invalid input"<<endl;
		}
		cout<<"Do you want to continue?(y/n)"<<endl;
		cin>>c;
		}
		while(c=='y'||c=='Y');
	}


};

int main()
```

```
{
        char pos[10];
        int len,i;
        stack st;
        cout<<"Enter the postfix expression"<<endl;
        cin>>pos;
        len=strlen(pos);
        for(i=0;i<len;i++)
        st.maketree(pos[i]);
        st.traverse();
        return 0;
}
```

Output :

```
Enter the postfix expression
ab*c-
Enter
1.Inorder
2.Preorder
3.Postorder
1
a*b-c
Do you want to continue?(y/n)
y
Enter
1.Inorder
2.Preorder
3.Postorder
2
-*abc
Do you want to continue?(y/n)
y
Enter
1.Inorder
2.Preorder
3.Postorder
3
ab*c-
Do you want to continue?(y/n)
n
```

Assignment No: 3

Aim:

Construct binary search tree by inserting the values in the order given. After constructing a binary search tree
1.Insert new node

Objective:

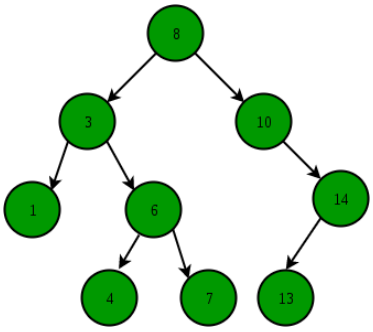
Understand the problem statement, determine and implement the various constructions on BST.
Understand the use of Stack for BST insertion.

Theory:

Binary search tree is a binary tree in which every node satisfies the following conditions:

- All values in the left sub tree of a node are less than the value of the node.
- All values in the right sub tree of a node are greater than the value of the node.
- Every node has a key and no two elements have same keys

The left and right sub trees are also binary search tree



Insertion:

- The way to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.
- Another way is examine the root and recursively insert the new node to the **left sub tree** if the new value is less than or equal to the root, or the **right sub tree** if the new value is greater than the root.

Example:

```
      100                          100
     /   \        Insert 40       /   \
   20    500     --------->      20    500
```

Applications:

- Binary Search Tree - Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages and libraries.
- Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality of service in routers. Also used in heap-sort.
- Huffman Coding Tree - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

Expected Input Type and Range:

Type - Alphabets and Symbols

Range – a to z and {+ , - , * , / , ^ }

Expected Output Type and Range:

Type - String of Alphabets and symbols mentioned above

Range - a to z and {+ , - , * , / , ^ }

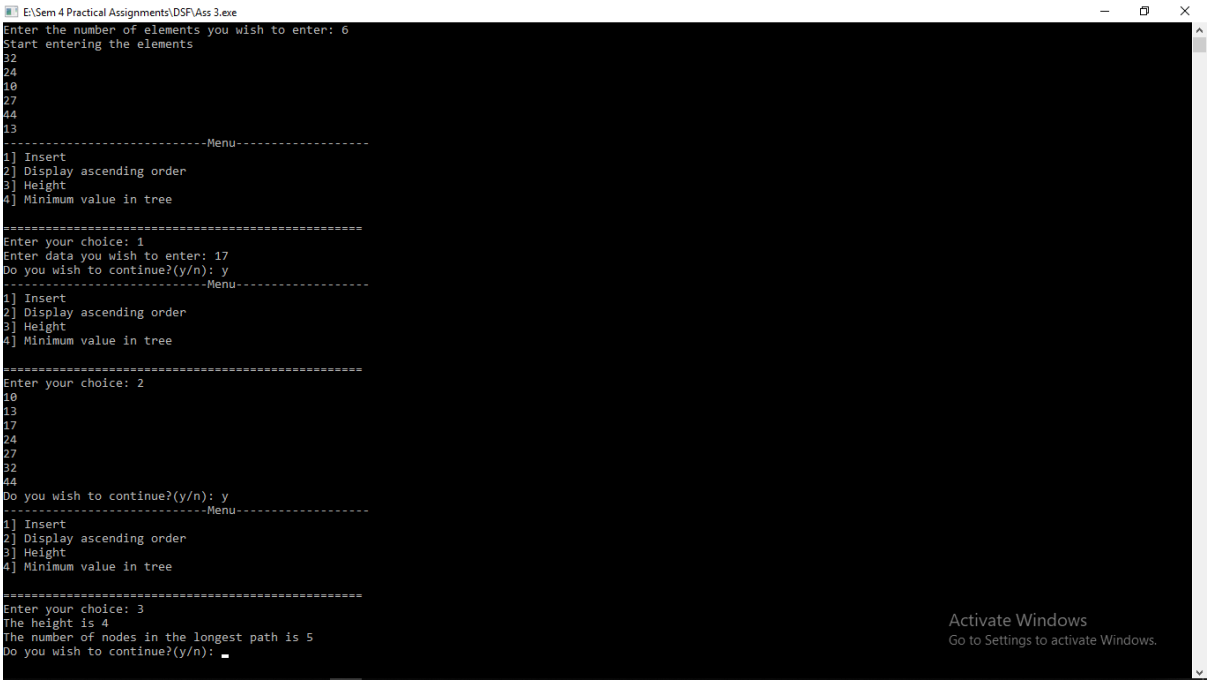
Program:

```
#include <iostream>
using namespace std;
class tnode
{
   int data;
   tnode *lt;
   tnode *rt;
   public:
   tnode *create(int s)
   {
      tnode *nn=new tnode;
      nn->data=s;
      nn->lt=NULL;
      nn->rt=NULL;
      return nn;
   }
   tnode *insert_r(tnode *root,int item)
   {
```

```
        tnode *nn;
        if(root == NULL)
            return root->create(item);
        if(item < root->data)
            root->lt=insert_r(root->lt, item);
        else
            root->rt=insert_r(root->rt, item);
        return root;
    }
    void inorder (tnode *root )
    {
        if(root!=NULL)
        {
            inorder (root -> lt);
            cout<<root->data<<"\n";
            inorder (root -> rt);
        }
    }
    int height(tnode *root)
    {
        if(root==NULL)
            return 0;
        int l_ht=height(root->lt);
        int r_ht=height(root->rt);
        if(l_ht>r_ht)
            return (l_ht+1);
        else
            return (r_ht+1);
    }
    int min(tnode *root)
    {
        tnode *temp=root;
        while(temp->lt!=NULL)
        {
            temp=temp->lt;
        }
        return temp->data;
    }
};
tnode *root;
int main()
{
    char r;
    int flag=0;
    do
    {
        root=NULL;
        char op;
        do
        {
            if(flag==0)
            {
                int n;
                cout<<"Enter the number of elements you wish to enter: ";
                cin>>n;
```

```
            int a[n];
            cout<<"Start entering the elements\n";
            for(int i =0;i<n;i++)
            {
               cin>>a[i];
               root=root->insert_r(root,a[i]);
            }
            flag=1;
         }
         int c;
         cout<<"----------------------------Menu-------------------\n";
         cout<<"1] Insert\n2] Display ascending order\n3] Height\n4] Minimum value in tree\n";
         cout<<"\n================================================\n";
         cout<<"Enter your choice: ";
         cin>>c;
         switch(c)
         {
            case 1: {
                     int item;
                     cout<<"Enter data you wish to enter: ";
                     cin>>item;
                     root=root->insert_r(root, item);
                  }
                  break;
            case 2: {
                     root->inorder(root);
                  }
                  break;
            case 3: {
                     int h=root->height(root);
                     cout<<"The height is "<<h-1<<endl;
                     cout<<"The number of nodes in the longest path is "<<h<<endl;
                  }
                  break;
            case 4: {
                     int min=root->min(root);
                     cout<<"The minimum value in the tree is "<<min<<endl;
                  }
                  break;
            default:cout<<"Invalid\n";
      }
      cout<<"Do you wish to continue?(y/n): ";
      cin>>op;
      }while(op=='y' || op=='Y');
      cout<<"Test pass?(y/n): ";
      cin>>r;
   }while(r=='n' || r=='N');
   cout<<"********************\n";
   cout<<"*   Thank You!   *\n";
   cout<<"********************\n";
   return 0;
}
```

Output:



Conclusion:

In this assignment, we have learnt the construction of binary search tree and implementation of insertion.

ASSIGNMENT 4

Problem statement:

Modify the above BST(assignment 3) such that the roles of the left and right pointers are swapped at every node.

Theory:

Binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: data structures that store "items" in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key. It has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Applications:

- Binary Search Tree - Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers. Also used in heap-sort.
- Huffman Coding Tree - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

Expected Input Type and Range:
Type - Alphabets and Symbols
Range – a to z and {+ , - , * , / , ^ }

Expected Output Type and Range:
Type - String of Alphabets and symbols mentioned above
Range - a to z and {+ , - , * , / , ^ }

Code:

```
#include<iostream>
#include<string.h>
using namespace std;
class node
{
public:
```

```
   int key;
   node *left;
   node *right;
};

node* create(int x)
{
       node *a;
       a=new node;
       a->key=x;
       a->left=NULL;
       a->right=NULL;
       return(a);
}

void inorder(node *root)
{
     if(root != NULL)
  {
     inorder(root -> left);
     cout<<"\t"<<root->key;
     inorder(root->right);
  }
}

node *insert(node *node, int key)
{
  if(node == NULL)
      return create(key);
  if(key < node->key)
   {
      node -> left = insert(node->left , key);
   }
  else if(key > node->key)
   {
       node -> right = insert(node->right , key);
   }
      return node;
}

node *small(node *root)
```

```
{
    node *temp;
    temp=root;
  while( temp->right != NULL)
   {
     temp=temp->right;
   }
   cout<<"\n Largest element of tree is "<<temp->key<<"\n";
}

int Height(node *root)
{
     if(root==NULL)
    {
       return 0;
    }
     int lheight=Height(root->left);
    int rheight=Height(root->right);
    if(lheight>rheight)
       return (1+lheight);
    else
       return (1+rheight);
}

node *swap(node *n)
{
        if(n==NULL)
        {
                return 0;
        }
        else
        {
                node *temp;
                swap(n->left);
                swap(n->right);
                temp=n->left;
                n->left=n->right;
                n->right=temp;
        return n;
        }
}
```

```
int main()
{
  int n,ch, op,h,s;
  node *root=NULL;
 do
 {
    cout<<"\nEnter choice\n1.Insert\t2.Smallest element\t3.Height\t4.Display tree\t5.Swap tree\t6.Exit ";
    cin>>op;

   switch(op)
   {
     case 1:
            cout<<"\nEnter the number of nodes you like to have ";
            cin>>n;
            cout<<"\n Enter no. : ";
            cin>>ch;
            root=insert(root,ch);

            for(int i=0; i<n-1;i++)
             {
                 cout<<"\n Enter no. : ";
                 cin>>ch;
                 insert(root,ch);
             }
              break;
     case 2:
             small(root);
             break;
     case 3:
             h=Height(root);
             cout<<"\n Height of tree is "<<h<<"\n";
              break;
     case 4:
             cout<<"\nTree is : \n";
             inorder(root);
             break;
     case 5:
             swap(root);
             inorder(root);
```

```
                break;
            case 6:
              cout<<"\n Exit ";
              break;
     default:
             cout<<"\n Invalid case";
  }
  }while(op!=6);
cout<<"\n\n";
  return 0;
}
```

Assignment No:- 5

Aim: Represent a given graph using adjacency matrix and traverse each node using Depth first search.

Theory:

Depth First Search:

Depth First Traversal or search (DFS) for a graph is similar to Depth Traversal of tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking**.**

Adjacency Matrix:

The adjacency matrix, sometimes also called the connection matrix, of a simple labeled graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position according to whether and are adjacent or not. For a simple graph with no self-loops, the adjacency matrix must have 0s on the diagonal.

The Adjacency matrix may be used as a data structure for the representation of the graphs in computer programing for manipulating graphs

Program:-

```
#include<iostream>

#define MAX 100

using namespace std;

class stack1

{

 private : int top;

 int item[MAX];

 public  : stack1()

 {

    top=-1;

 }


//PUSH function starts here
```

```
void push(int data)
{
  if(isfull())
  cout<<"\nStack is full!!!";
  else
  {
    top++;
    item[top] = data;
  }
}

//POP function starts here
 int pop()
 {
   int data;
   if(isempty())
   cout<<"\nStack is empty!!!";
   else
  {
    data=item[top];
    top--;
  }
  return data;
 }

 // checking if the stack is overflow or not
 int isfull()
 {
   if(top==MAX-1)
   return 1;
```

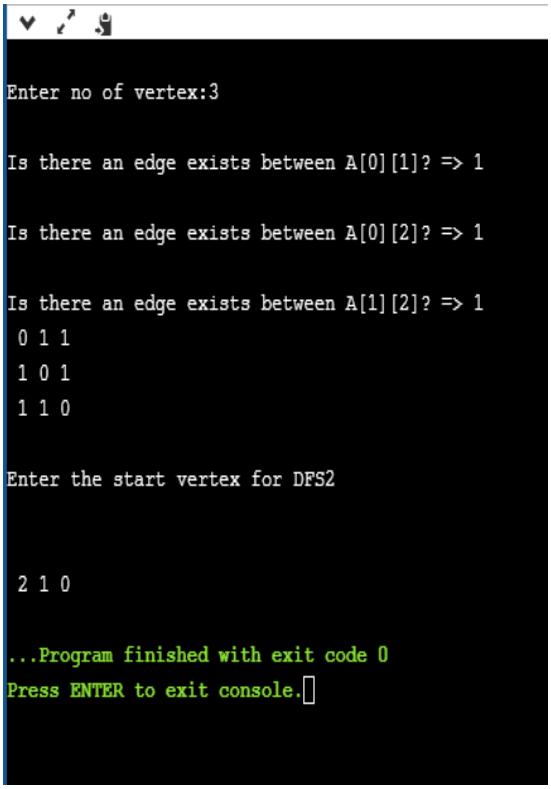
```
    else
    return 0;
  }

  // checking if the stack is underflow or not
  int isempty()
  {
    if(top==-1)
    return 1;
    else
    return 0;
  }
};

class Graph
{
  private : int A[MAX][MAX],n;
  public  : Graph()
  {
    cout<<"\nEnter no of vertex:";
    cin>>n;
  }

  //Initialising operation
  void initialize()
  {
   for(int i=0;i<n;i++)
   {
    for(int j=0;j<n;j++)
    {
```

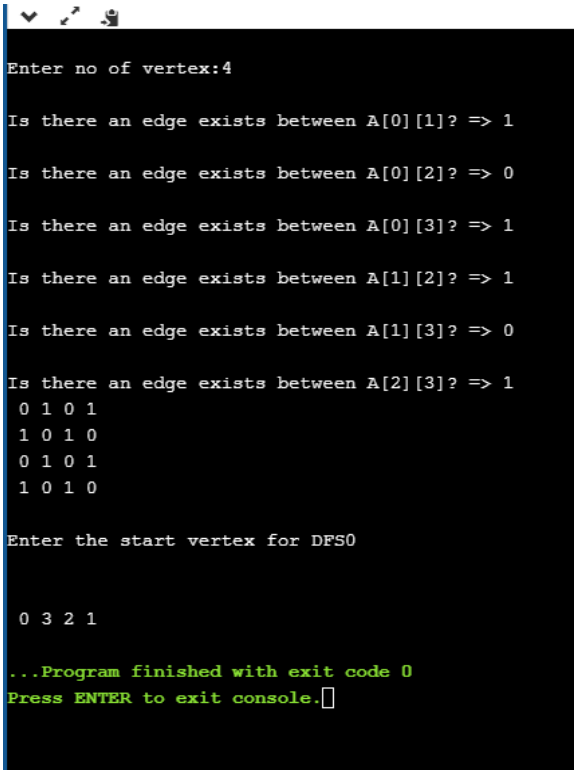
```
    A[i][j]=0;
   }
  }
 }

 //Displaying operation
 void displayadj()
 {
   for(int i=0;i<n;i++)
  {
    for(int j=0;j<n;j++)
    {
      cout<<" "<<A[i][j];
    }
   cout<<"\n";
  }
 }

 void createadj()
{
  int data;
  for(int i=0;i<n;i++)
  {
    for(int j=i+1;j<n;j++)
    {
      cout<<"\nIs there an edge exists between A["<<i<<"]["<<j<<"]? => ";
      cin>>data;
      if(data==1)
      A[i][j]=A[j][i]=1;
    }
```

```
  }
}

 void createadjdirec()
{
  int data;
 for(int i=0;i<n;i++)
 {
   for(int j=0;j<n;j++)
  {
    if(i!=j)
   {
     cout<<"\nIs there an edge exists between A["<<i<<"]["<<j<<"]? => ";
     cin>>data;
     if(data==1)
     A[i][j]=A[j][i]=1;
   }
  }
 }
}

 void DFS(int startvertex)
{
  stack1 s;
  int i, j, visited[MAX];
  for(i=0;i<n;i++)
  visited[i]=0;
  cout<<"\n\n";
  s.push(startvertex);
  while(!s.isempty())
```

```
   {
     startvertex=s.pop();
     if(visited[startvertex]==0)
     cout<<" "<<startvertex;
     visited[startvertex] = 1;
     for(i=0;i<n;i++)
     {
       if( A[startvertex][i] == 1 && visited[i]==0)
       {
        s.push(i);
       }
     }
   }
 }
};

int main()
{
  int startvertex;
  Graph g;
  g.initialize();
  g.createadj();
  g.displayadj();
  cout<<"\nEnter the start vertex for DFS";
  cin>>startvertex;
  g.DFS(startvertex);
  return 0;
}
```

Output:-


Enter no of vertex:3
Is there an edge exists between A[0][1]? => 1
Is there an edge exists between A[0][2]? => 1
Is there an edge exists between A[1][2]? => 1
0 1 1
1 0 1
1 1 0
Enter the start vertex for DFS2
2 1 0
...Program finished with exit code 0
Press ENTER to exit console.


Enter no of vertex:4
Is there an edge exists between A[0][1]? => 1
Is there an edge exists between A[0][2]? => 0
Is there an edge exists between A[0][3]? => 1
Is there an edge exists between A[1][2]? => 1
Is there an edge exists between A[1][3]? => 0
Is there an edge exists between A[2][3]? => 1
0 1 0 1
1 0 1 0
0 1 0 1
1 0 1 0
Enter the start vertex for DFS0
0 3 2 1
...Program finished with exit code 0
Press ENTER to exit console.

ASSIGNMENT - 6

Problem statement:

Represent a graph using adjacency list and traverse each node using Breadth First Search.

Theory:

Breadth First Search:

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors. Thus closer nodes get visited first

Breadth-first search (BFS) is an important graph search algorithm that is used to solve many problems including finding the shortest path in a graph and solving puzzle games (such as Rubik's Cubes). Many problems in computer science can be thought of in terms of graphs.

Adjacency List:

In graph theory, an adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbours of a vertex in the graph. This is one of several commonly used representations of graph for use in computer programs.

The main operation performed by the adjacency list data structure is to report a list of the neighbors of a given vertex

Expected i/p type :

If edge exists between 2 vertices

Expected o/p type :

Breadth first search

Code:

```
#include<iostream>
using namespace std;

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

```
 node *heads[10];

void adjacencyListRepre(int);
node *createNode(int value);
void display(int );
void bfs(int vertices);
void enqueue(int value);
int dequeue();

//QUEUE
int Front = -1;
int Rear = -1;
int queue[20];



int main()
{
	int choise;

		int vertices;
		cout<<"Enter the number of vertices :";
		cin>>vertices;

	do{
		cout<<"---------------------------------------------------------------------\n";
		cout<<"\t\t\tBreadth First Search \n\n";
		cout<<"\t\t1.Create a Graph\n";
		cout<<"\t\t2.Display the Graph\n";
		cout<<"\t\t3.Breadth First Search\n";
		cout<<"\t\t4.Exit\n";
```

```
            cout<<"\tYour choise : ";
            cin>>choise;
            switch(choise)
            {
                  case 1:
                        {

                              adjacencyListRepre(vertices);
                        }
                  break;
                  case 2:
                              display(vertices);
                  break;
                  case 3:
                        bfs(vertices);
                  break;

            }


      }while(choise<4);

}


void adjacencyListRepre(int vertices)
{
      int edges,ev;
```

```
//node *heads[vertices];

for(int i=0;i<vertices;i++)
        heads[i] =NULL;

for (int i = 0; i < vertices; i++)
{
        cout<<"Enter the number of edges connected to vertex "<<i<<" : ";
        cin>>edges;

        for (int j = 0; j < edges; j++)
        {
                cout<<"Enter the "<<j+1 <<" th edge (Enter only ending vertex): ";
                cin>>ev;

                if(heads[i]==NULL)
                        heads[i] = createNode(ev);
                else
                {
                        node *temp = heads[i];

                        while(temp->next!=NULL)
                        temp = temp->next;

                        temp->next = createNode(ev);


                }
```

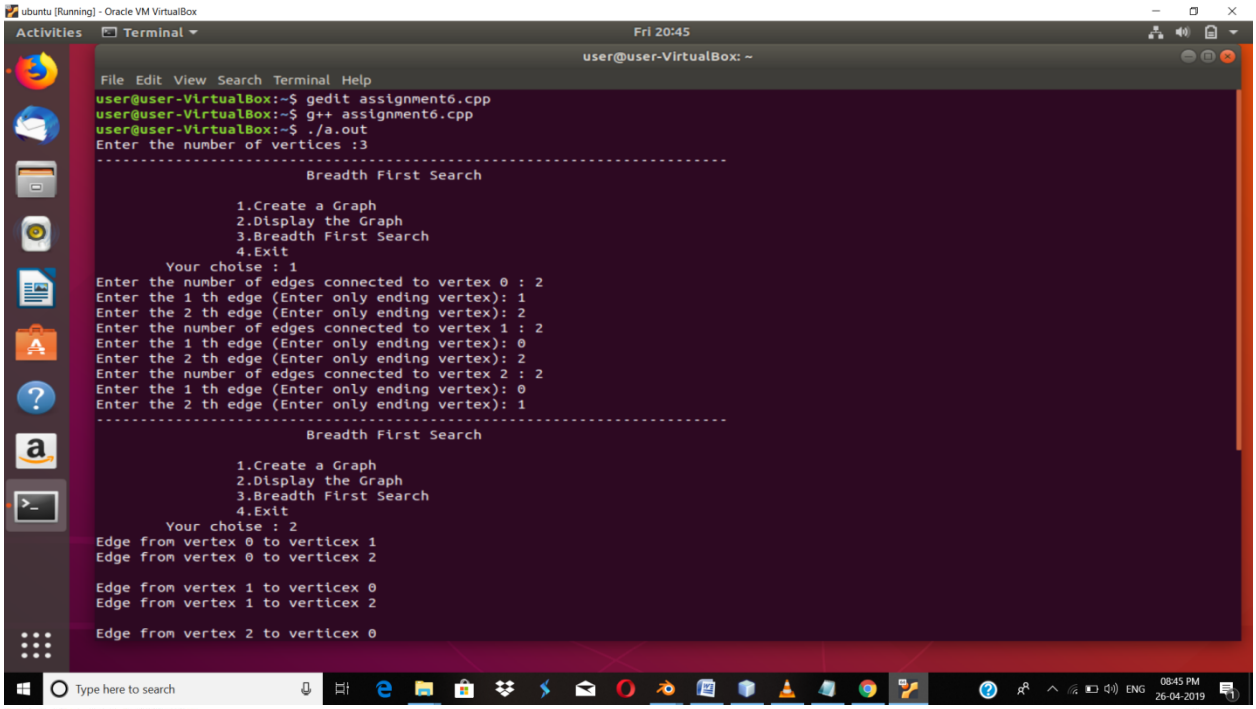
```
		}
	}



}
void display(int vertices)
{

for (int i = 0; i < vertices; i++)
	{
		node *temp = heads[i];

		while(temp!=NULL)
			{
				cout<<"Edge from vertex "<<i<<" to verticex "<<temp->data<<"\n";
				temp = temp->next;
			}
			cout<<"\n";
	}
}

node *createNode(int value)
{
	node *temp =new node;
	temp->data = value;
	temp->next = NULL;
	return temp;
}
```

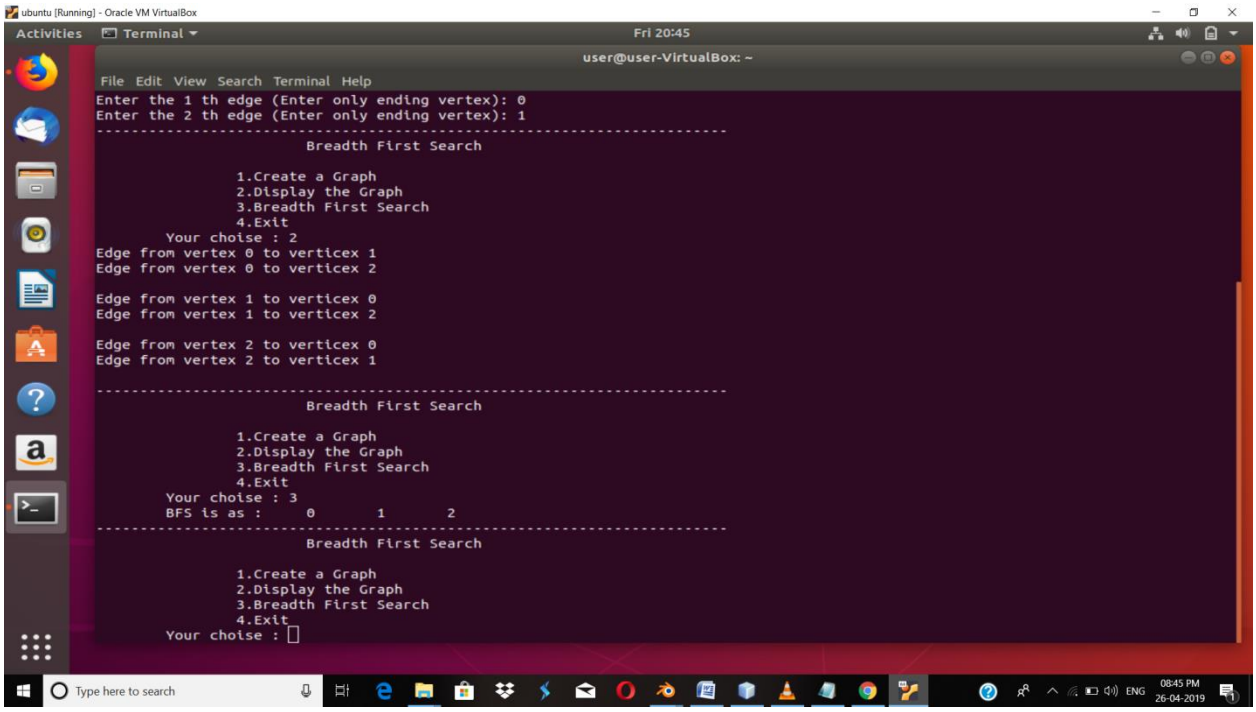
```
void enqueue(int value)
{
	if(Front ==-1&&Rear == -1)
	{
		Front = 0;
		Rear = 0;
	}
	else
	{
		Rear = Rear +1;
	}
	queue[Rear] =value;
}

int dequeue()
{
	int x = queue[Front];
	if(Front==Rear)
	{
		Front = -1;
		Rear = -1;
	}
	else
	{
		Front = Front+1;
	}
	return x;
}
```

```
void bfs(int vertices)
{
	int visited[vertices];
	for (int i = 0; i < vertices; i++)
	{
		visited[i] = 0;
	}

	enqueue(0);
	visited[0] = 1;
	cout<<"\tBFS is as : ";


	while(Front!=(-1))
	{
		int x= dequeue();
		cout<<"\t"<<x;
		node *temp = heads[x];

		while(temp!=NULL)
		{
			if(!visited[temp->data])
			{
				enqueue(temp->data);
				visited[temp->data] = 1;
			}
			temp = temp->next;
		}
	}
cout<<"\n";
```

}

Output:


ubuntu [Running] - Oracle VM VirtualBox
Activities
Terminal
Fri 20:45
user@user-VirtualBox: ~
File Edit View Search Terminal Help
user@user-VirtualBox:~$ gedit assignment6.cpp
user@user-VirtualBox:~$ g++ assignment6.cpp
user@user-VirtualBox:~$ ./a.out
Enter the number of vertices :3
Breadth First Search
1.Create a Graph
2.Display the Graph
3.Breadth First Search
4.Exit
Your choise : 1
Enter the number of edges connected to vertex 0 : 2
Enter the 1 th edge (Enter only ending vertex): 1
Enter the 2 th edge (Enter only ending vertex): 2
Enter the number of edges connected to vertex 1 : 2
Enter the 1 th edge (Enter only ending vertex): 0
Enter the 2 th edge (Enter only ending vertex): 2
Enter the number of edges connected to vertex 2 : 2
Enter the 1 th edge (Enter only ending vertex): 0
Enter the 2 th edge (Enter only ending vertex): 1
Breadth First Search
1.Create a Graph
2.Display the Graph
3.Breadth First Search
4.Exit
Your choise : 2
Edge from vertex 0 to verticex 1
Edge from vertex 0 to verticex 2
Edge from vertex 1 to verticex 0
Edge from vertex 1 to verticex 2
Edge from vertex 2 to verticex 0
Type here to search
ENG
08:45 PM
26-04-2019


ubuntu [Running] - Oracle VM VirtualBox
Activities
Terminal
Fri 20:45
user@user-VirtualBox: ~
File Edit View Search Terminal Help
Enter the 1 th edge (Enter only ending vertex): 0
Enter the 2 th edge (Enter only ending vertex): 1
Breadth First Search
1.Create a Graph
2.Display the Graph
3.Breadth First Search
4.Exit
Your choise : 2
Edge from vertex 0 to verticex 1
Edge from vertex 0 to verticex 2
Edge from vertex 1 to verticex 0
Edge from vertex 1 to verticex 2
Edge from vertex 2 to verticex 0
Edge from vertex 2 to verticex 1
Breadth First Search
1.Create a Graph
2.Display the Graph
3.Breadth First Search
4.Exit
Your choise : 3
BFS is as : 0 1 2
Breadth First Search
1.Create a Graph
2.Display the Graph
3.Breadth First Search
4.Exit
Your choise :
Type here to search
ENG
08:45 PM
26-04-2019

ASSIGNMENT 7

Problem Statement:- A customer wants to travel from A and B. He books a cab from A and B . Calculate the shortest path by avoiding real time traffic from destination A to destination B.

Algorithm:-

```
using namespace std;

// Function to calculate distance
float distance(int x1, int y1, int x2, int y2)
{
    // Calculating distance
    return sqrt(pow(x2 - x1, 2) +
                pow(y2 - y1, 2) * 1.0);
}

// Drivers Code
int main()
{
    cout << distance(3, 4, 4, 3);
    return 0;
}
```

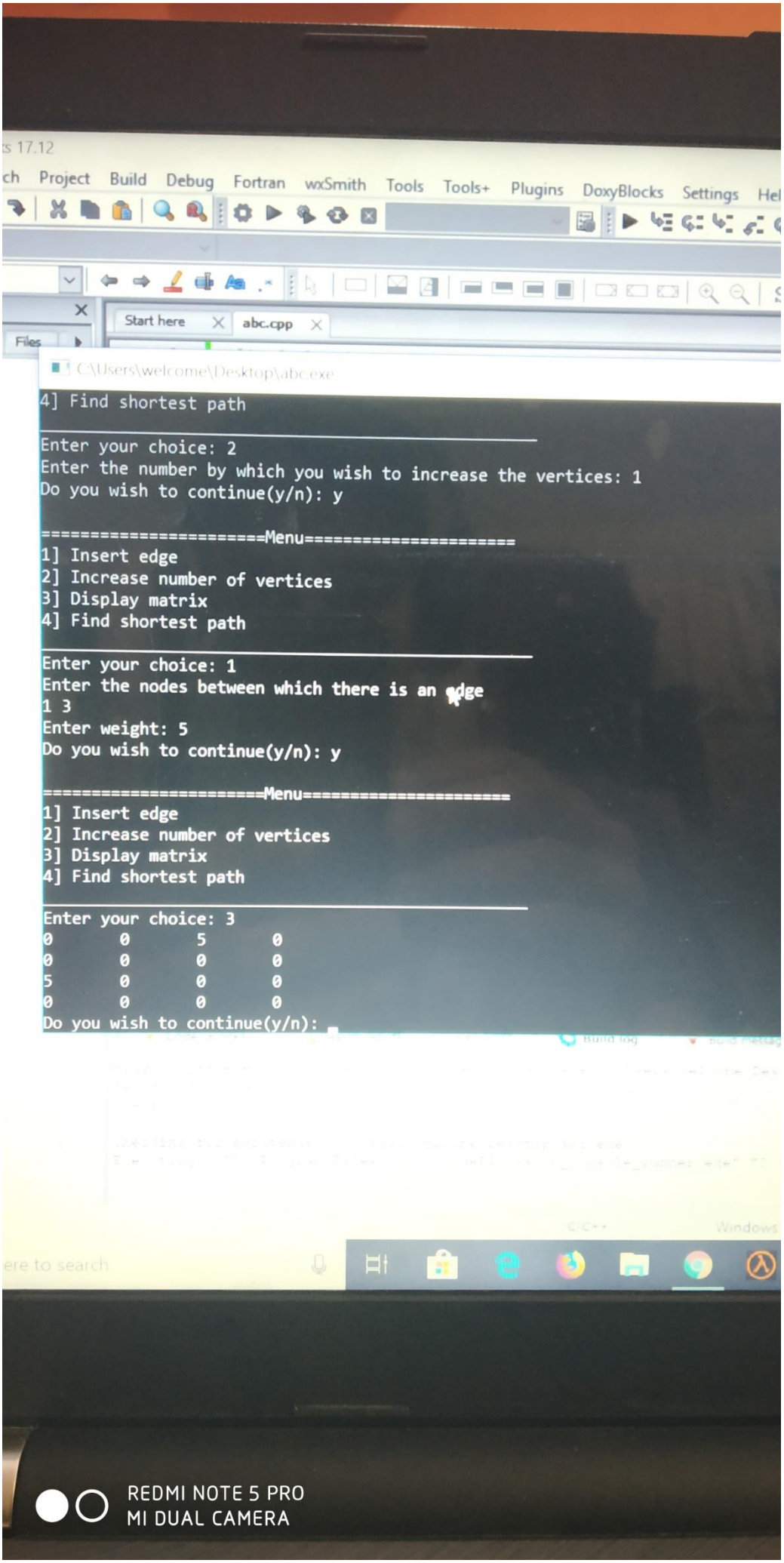
CODE:-

```
#include <iostream>
using namespace std;
class graph
{
    int a[100][100];
    int v;
public:
    void insert_edge(int n1,int n2,int wt)
    {
        if(n1-1>=v||n2-1>=v)
            cout<<"Vertex request out of range\n";
        else
        {
            a[n1-1][n2-1]=wt;
            a[n2-1][n1-1]=wt;
        }
    }
    void display()
    {
        for(int i=0;i<v;i++)
        {
            for(int j=0;j<v;j++)
            {
                cout<<a[i][j]<<"\t";
            }
            cout<<endl;
        }
    }
    void update_v(int n)
    {
        v=n;
    }
    void dj(int src,int dst)
```

```
    {
        int sp[v],dist[v],visited[v],c=0;
        for(int i=0;i<v;i++)
        {
            visited[i]=0;
            dist[i]=9999;
        }
        dist[src-1]=0;
        for(int i=0;i<v;i++)
        {
            int min=9999,min_ind;
            for(int j=0;j<v;j++)
            {
                if(!visited[j] && dist[j]<min)
                {

                    min=dist[j];
                    min_ind=j;
                }
            }
            int U=min_ind;
            visited[U]=1;
            sp[c]=U;
            c++;
            for(int V=0;V<v;V++)
            {
                if(!visited[V] && a[U][V] && dist[U]+a[U][V]<dist[V] &&
dist[U]!=9999)
                  {

                  dist[V]=dist[U]+a[U][V];
                  }
            }
            if(U==(dst-1))
            {
                cout<<"The minimum distance is "<<dist[U]<<" following the path ";
                break;
            }
        }
        for(int i=0;i<c;i++)
        {
            cout<<sp[i]+1;
        }
        cout<<endl;
    }
};
int main()
{
    char r;
    do
    {
        graph g;
        char op;
        int v;
        cout<<"Enter number of vertices: ";
        cin>>v;
        g.update_v(v);
        do
        {
            int c;
            cout<<"\n=======================Menu=====================\n";
            cout<<"1] Insert edge\n2] Increase number of vertices\n3] Display
matrix\n4] Find shortest path\n";
            cout<<"_____________________________________________________\n";
```

```
            cout<<"Enter your choice: ";
            cin>>c;
            switch(c)
            {
                case 1: {
                            int n1,n2,wt;
                            cout<<"Enter the nodes between which there is an
edge\n";
                            cin>>n1>>n2;
                            cout<<"Enter weight: ";
                            cin>>wt;
                            g.insert_edge(n1,n2,wt);
                        }
                        break;
                case 2: {
                            int n;
                            cout<<"Enter the number by which you wish to increase
the vertices: ";
                            cin>>n;
                            v+=n;
                            g.update_v(v);
                        }
                        break;
                case 3: {
                            g.display();
                        }
                        break;
                case 4: {
                            int src,dst;
                            cout<<"Source: ";
                            cin>>src;
                            cout<<"Destination: ";
                            cin>>dst;
                            g.dj(src,dst);
                        }
                        break;
                default:cout<<"Error 404.....page not found\n";
            }
            cout<<"Do you wish to continue(y/n): ";
            cin>>op;
        }while(op=='y' || op=='Y');
        cout<<"Test pass(y/n): ";
        cin>>r;
    }while(r=='n' || r=='N');
    cout<<"******************\n";
    cout<<"*   Thank You!   *\n";
    cout<<"******************\n";
    return 0;
}
```

OUTPUT:-


Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings
Start here
abc.cpp
C:\Users\welcome\Desktop\abc.exe
4] Find shortest path
Enter your choice: 2
Enter the number by which you wish to increase the vertices: 1
Do you wish to continue(y/n): y
=====Menu=====
1] Insert edge
2] Increase number of vertices
3] Display matrix
4] Find shortest path
Enter your choice: 1
Enter the nodes between which there is an edge
1 3
Enter weight: 5
Do you wish to continue(y/n): y
=====Menu=====
1] Insert edge
2] Increase number of vertices
3] Display matrix
4] Find shortest path
Enter your choice: 3
0 0 5 0
0 0 0 0
5 0 0 0
0 0 0 0
Do you wish to continue(y/n):
REDMI NOTE 5 PRO
MI DUAL CAMERA

ASSIGNMENT 10

PROBLEM STATEMENT: A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword .

THEORY: A **binary** search **tree** is called **AVL tree** or **height-balanced tree**, if for each node v the **height** of the right subtree h(Tr) of v and the **height** of the left subtree h(Tl) of v differ by at most. **AVL trees** cannot degenerate into linear lists. **AVL trees** with n nodes have a **height** in O(log n).

APPLICATIONS:

1. Self-**balancing** binary search **trees** can be used in a natural way to construct and maintain ordered lists, such as priority queues.
2. AVL trees are beneficial in the cases where you are designing some database where insertions and deletions are not that frequent but you have to frequently look-up for the items present in there.
3. AVL trees are applied in the following situations:

a) There are few insertion and deletion operations
b) Short search time is needed
c) Input data is sorted or nearly sorted

CODE:

```
#include <iostream>

#include<string>

using namespace std;

class dictionary;

class node

{
```

```
 string word,meaning;
 node *left,*right;
public:
 friend class dictionary;
 node()
 {
  left=NULL;
  right=NULL;

 }
 node(string word, string meaning)
 {
  this->word=word;
  this->meaning=meaning;
  left=NULL;
  right=NULL;
 }
};

class dictionary
{
 node *root;
public:
 dictionary()
{
  root=NULL;}
 void create();
 void inorder_rec(node *rnode);
 void postorder_rec(node *rnode);
 void inorder()
```

```
{
 inorder_rec(root);
}
void postorder();

bool insert(string word,string meaning);
int search(string key);

};
int dictionary::search(string key)
{
 node *tmp=root;
 int count;
 if(tmp==NULL)
 {
  return -1;
 } if(root->word==key)
  return 1;
 while(tmp!=NULL)
 {

  if((tmp->word)>key)
  {
   tmp=tmp->left;
   count++;
  }
  else if((tmp->word)<key)
  {
   tmp=tmp->right;
   count++;
```

```
 }
 else if(tmp->word==key)
 {
  return ++count;
 }
}
 return -1;
}
void dictionary::postorder()
{
 postorder_rec(root);
}
void dictionary::postorder_rec(node *rnode)
{
if(rnode)
 {
  postorder_rec(rnode->right);
  cout<<" "<<rnode->word<<" : "<<rnode->meaning<<endl;
  postorder_rec(rnode->left);
 }
}
void dictionary::create()
{
 int n;
 string wordI,meaningI;
 cout<<"\nHow many Word to insert?:\n";
 cin>>n;
 for(int i=0;i<n;i++)
 {
  cout<<"\nEnter Word: ";
```

```
 cin>>wordI;
 cout<<"\nEnter Meaning: ";
 cin>>meaningI;
 insert(wordI,meaningI);
 }
}
void dictionary::inorder_rec(node *rnode)
{ if(rnode)
 {
 inorder_rec(rnode->left);
 cout<<" "<<rnode->word<<" : "<<rnode->meaning<<endl;
 inorder_rec(rnode->right);
 }
}
bool dictionary::insert(string word, string meaning)
{
 node *p=new node(word, meaning);
 if(root==NULL)
 {
 root=p;
 return true;
 }
 node *cur=root;
 node *par=root;
 while(cur!=NULL) //traversal
 {
 if(word>cur->word)
 {par=cur;  cur=cur->right;
 }
 else if(word<cur->word)
```

```
 {
  par=cur;
  cur=cur->left;
 }
 else
 {
  cout<<"\nWord is already in the dictionary.";
  return false;
 }
}
if(word>par->word)
{
 par->right=p;
 return true;
}
else
{
 par->left=p;
 return true;
}
}
int main()
{
 string word;
 dictionary months;
 months.create();
 cout<<"Ascending order\n";
 months.inorder();

 cout<<"\nDescending order:\n";
```

```
months.postorder();


cout<<"\nEnter word to search: ";
cin>>word;
int comparisons=months.search(word);
if(comparisons==-1)
{
 cout<<"\nNot found word";
}
else
{
 cout<<"\n "<<word<<" found in "<<comparisons<<" comparisons";
}
return 0;
}
```

OUTPUT: