**Assignment 1**

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

Code:

#include<iostream>

using namespace std;

class TreeNode {

public:

char data; //operator or operand

TreeNode\* lchild; //pointer to left child

TreeNode\* rchild; //pointer to right child

friend class Node;

friend class Stack;

TreeNode() { //constructor to set left and right pointers to NULL

data = '#';

lchild = NULL;

rchild = NULL;

}

};

class Node {

public:

TreeNode\* treenode; //node of the stack, it has a TreeNode and a pointer to the next node

Node\* next;

Node()

{

next = NULL;

}

friend class Stack;

};

class Stack {

public:

Node\* top; //top of stack

Stack()

{

top = NULL;

}

void push(TreeNode\* n); // to push a treenode(pointer) onto the stack

TreeNode\* pop(); // pop function returns the pointer to the treenode it contains

void postorder(TreeNode\* root);

void inorder(TreeNode\* root) ;

void preorder(TreeNode\* root) ;

};

void Stack::push(TreeNode\* n) //pushing in stack, i.e. adding at the beginning of the linked list

{

Node\* temp = this->top;

Node\* a = new Node;

a->treenode = n;

top = a;

top -> next = temp;

}

TreeNode\* Stack::pop() //popping from linked list, i.e. deleting from beginning of the linked list

{

Node\* temp = top;

top = top ->next;

temp -> next = NULL;

return (temp->treenode);

}

void Stack::postorder(TreeNode\* root)

{

if(root)

{

postorder(root->lchild); // first it calls using left child

postorder(root->rchild); // then it calls using right child

cout<<root->data<<" ";

}

}

void Stack::preorder(TreeNode\* root)

{

if(root)

{

cout<<root->data<<" "; //preorder, cout first then recursive call using lchild, then rchild

preorder(root->lchild);

preorder(root->rchild);

}

}

void Stack::inorder(TreeNode\* root)

{

if(root)

{

inorder(root->lchild); //recursive call using lchild, then cout and then recursive call passing rchild

cout<<root->data<<" ";

inorder(root->rchild);

}

}

int main()

{

char ip[20];

cout<<"ENTER POSTFIX EXPRESSION."<<endl;

cin>>ip; // Input the postfix expression into the string

int i = 0;

Stack st; // an object of the type Stack

while(ip[i] != '\0')

{ //Traverse the postfix expression till end

if( (ip[i] >= 65 && ip[i] <= 90) || (ip[i] >= 97 && ip[i] <= 122))

{ //if character is an alphabet, i.e., an operand

TreeNode\* n = new TreeNode; //create a treenode and put the character into treenode->data

n->data = ip[i];

st.push(n); //push the treenode into the stack

}

if( ip[i] == '+' || ip[i] == '-' || ip[i] == '\*' || ip[i] == '/')

{ //if character is an operator

TreeNode\* a = new TreeNode;

a->data = ip[i]; // create treenode and put its data as the operator

a->rchild = st.pop(); //first pop = rchild

a->lchild = st.pop(); //second pop = lchild

st.push(a); // push result onto stack

}

i++;

}

TreeNode\* root = (st.top) ->treenode; //top of stack = root of tree

cout<<"\n"<<"PRE-FIX EXPRESSION:";

st.preorder(root); //call the preorder function by passing to it the root of the tree

cout<<"\n"<<"IN-FIX EXPRESSION:";

st.inorder(root); // call the inorder function by passing to it the root of the tree

cout<<"\n"<<"POST-FIX EXPRESSION:";

st.postorder(root); // call the postorder function by passing to it the root of the tree

cout<<endl;

return 0;

}

/\*

OUTPUT:

ENTER THE POST-FIX EXPRESSION

>>(INPUT) AB+C\*

(OUTPUT)

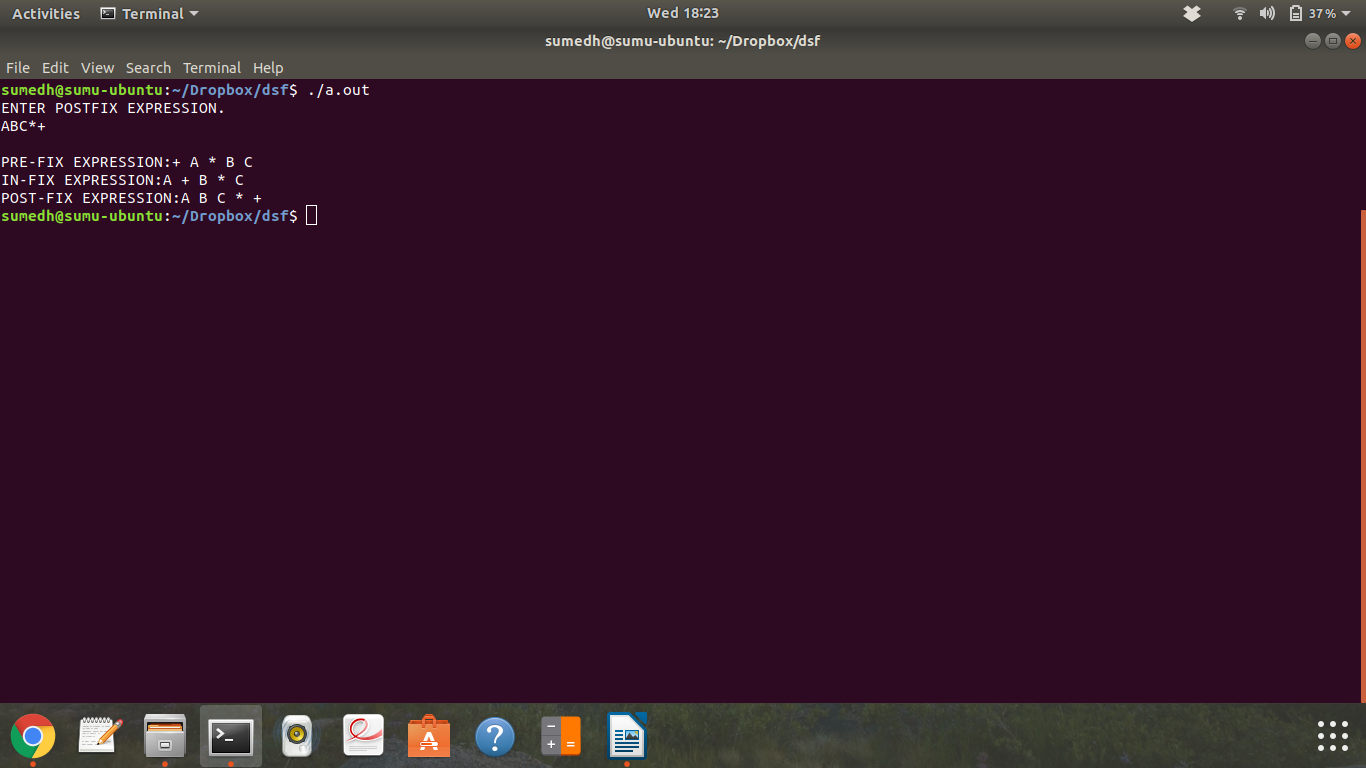
PRE-FIX EXPRESSION: \* + A B C

IN-FIX EXPRESSION: A + B \* C

POST-FIX EXPRESSION: A B + C \*

\*/

Output:



**Assignment 2**

Aim: Construct an expression tree from postfix expression and perform non-recursive Inorder and Preorder traversals.

Code:

#include<iostream>

using namespace std;

class TreeNode {

public:

char data; //operator or operand

TreeNode\* lchild; //pointer to left child

TreeNode\* rchild; //pointer to right child

friend class Node;

friend class Stack;

TreeNode() { //constructor to set left and right pointers to NULL

data = '#';

lchild = NULL;

rchild = NULL;

}

};

class Node {

public:

TreeNode\* treenode; //node of the stack, it has a TreeNode and a pointer to the next node

Node\* next;

Node()

{

next = NULL;

}

friend class Stack;

};

class Stack {

public:

Node\* top; //top of stack

Stack()

{

top = NULL;

}

void push(TreeNode\* n); // to push a treenode(pointer) onto the stack

TreeNode\* pop(); // pop function returns the pointer to the treenode it contains

void InOrder(TreeNode\* n);

bool isEmpty();

void PreOrder(TreeNode\* n);

};

bool Stack::isEmpty()

{

if(this->top == NULL)

return true;

else

return false;

}

void Stack::PreOrder(TreeNode\* current)

{

this->push(current); //push root onto stack

while(1) //enter the loop, breaking condition is inside the loop

{

while(current) // keep printing the node, pushing it onto stack and going to left till you reach NULL

{

cout<<current->data<<" ";

this->push(current);

current = current->lchild;

}

current = this->pop(); //after we reach NULL, put current = pop node from stack

current = current -> rchild; // and go to it's right

if(this->top == NULL ) // if stack is empty, our prefix expression is printed, so return

return;

}

}

void Stack::InOrder(TreeNode\* current)

{

while(1) //enter the loop, breaking condition is inside(when stack becomes empty)

{

while(current) //keep traversing till left child till it becomes null

{

this->push(current);

current = current->lchild;

}

if(this->top == NULL) //if stack is empty, our infix expression is printed, so return

return;

current = this->pop(); //if stack is not empty, then current = pop, print it's data, and go to it's right

cout<<current->data<<" ";

current = current->rchild;

}

}

void Stack::push(TreeNode\* n) //pushing in stack, i.e. adding at the beginning of the linked list

{

Node\* temp = this->top;

Node\* a = new Node;

a->treenode = n;

top = a;

top -> next = temp;

}

TreeNode\* Stack::pop() //popping from linked list, i.e. deleting from beginning of the linked list

{

Node\* temp = top;

top = top ->next;

temp -> next = NULL;

return (temp->treenode);

}

int main()

{

char ip[20];

cout<<"ENTER POSTFIX EXPRESSION."<<endl;

cin>>ip; // Input the postfix expression into the string

int i = 0;

Stack st; // an object of the type Stack for construction of tree

Stack traverse; // object of stack to traverse without recursion

while(ip[i] != '\0')

{ //Traverse the postfix expression till end

if( (ip[i] >= 65 && ip[i] <= 90) || (ip[i] >= 97 && ip[i] <= 122))

{ //if character is an alphabet, i.e., an operand

TreeNode\* n = new TreeNode; //create a treenode and put the character into treenode->data

n->data = ip[i];

st.push(n); //push the treenode into the stack

}

if( ip[i] == '+' || ip[i] == '-' || ip[i] == '\*' || ip[i] == '/')

{ //if character is an operator

TreeNode\* a = new TreeNode;

a->data = ip[i]; // create treenode and put its data as the operator

a->rchild = st.pop(); //first pop = rchild

a->lchild = st.pop(); //second pop = lchild

st.push(a); // push result onto stack

}

i++;

}

TreeNode\* root = (st.top) ->treenode;

Stack traverse\_pre; //object of stack to display preorder

cout<<"PREFIX EXPRESSION IS: ";

traverse\_pre.PreOrder(root);

cout<<endl;

cout<<"INFIX EXPRESSION IS: ";

traverse.InOrder(root);

cout<<endl;

return 0;

}

/\*

OUTPUT:

ENTER THE POST-FIX EXPRESSION

>>(INPUT) AB+C\*

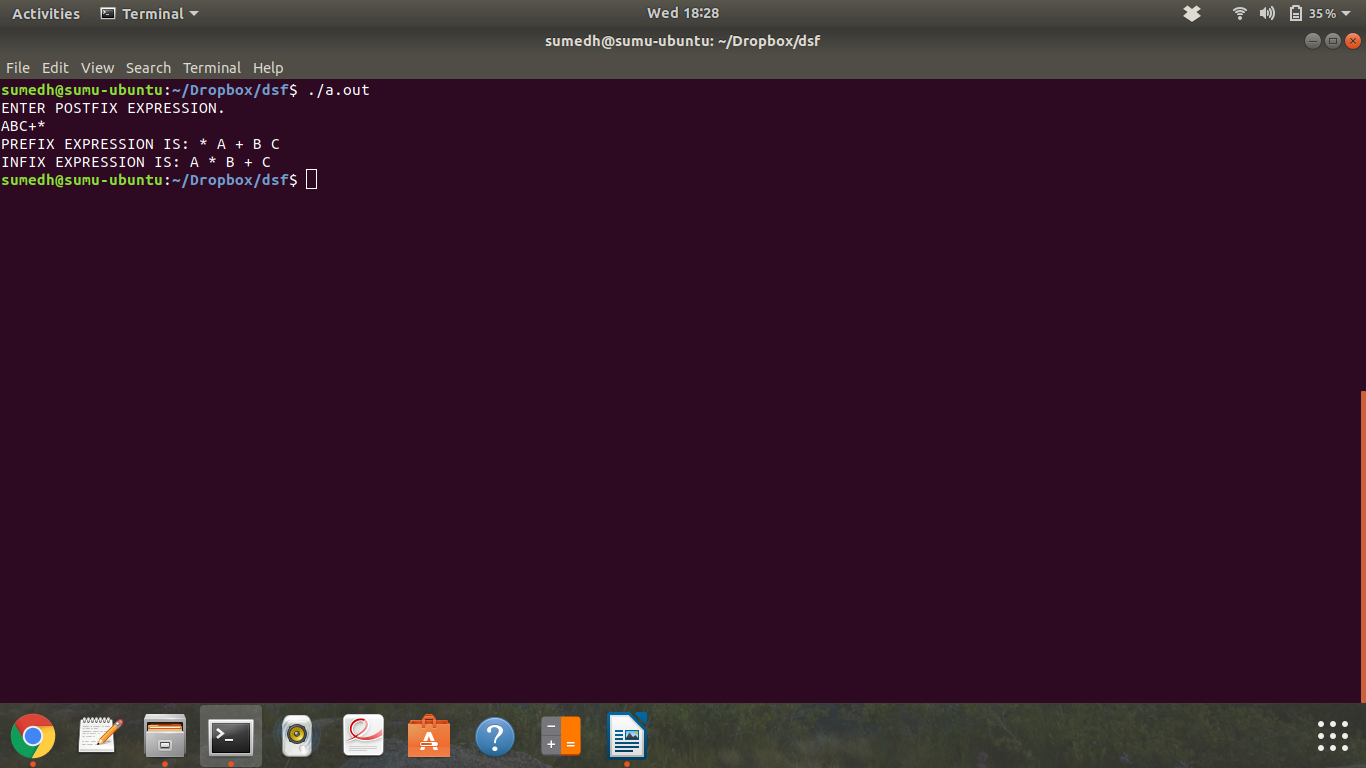
(OUTPUT)

PRE-FIX EXPRESSION IS: \* + A B C

IN-FIX EXPRESSION IS: A + B \* C

\*/

OUTPUT:



**Assignment 3**

Aim: Construct binary search tree by inserting the values in the order given. After constructing a binary search tree

i. Insert new node.

ii. Find number of nodes in longest path along with its height.

Iii. Minimum data value found in the tree.

Code:

#include<iostream>

using namespace std;

class Node

{

public:

int data;

Node\* left;

Node\* right;

Node()

{

left = NULL;

right = NULL;

}

friend class BSTree;

};

class BSTree

{

public:

Node\* root;

BSTree()

{

root = NULL;

}

void construct(int n);

void insert(int num);

void inorder(Node\* current);

int MaxHeight(Node\* curr);

int Min(Node\* curr);

};

int BSTree::Min(Node\* curr)

{

Node\* parent, \*child;

parent = this->root;

child = this -> root;

while(child)

{

parent = child;

child = child -> left;

}

return parent -> data;

}

int BSTree::MaxHeight(Node\* curr)

{

if(curr == NULL)

return -1;

else

{

int lheight = this -> MaxHeight(curr -> left) + 1;

int rheight = this -> MaxHeight(curr -> right) + 1;

if(lheight <= rheight)

return rheight;

else

return lheight;

}

}

void BSTree::inorder(Node\* current)

{

if(current)

{

inorder(current->left);

cout<<current->data<<" ";

inorder(current->right);

}

}

void BSTree::construct(int num)

{

int data;

cout<<"ENTER THE DATA."<<endl;

for(int i = 0; i<num; i++)

{

cin>>data;

this->insert(data);

}

}

void BSTree::insert(int num)

{

Node\* temp = new Node;

temp->data = num;

Node\* parent, \*child;

if(this->root == NULL)

root = temp;

else

{

child = root;

while(child != NULL)

{

if(child->data > num)

{

parent = child;

child = child->left;

}

else

{

parent = child;

child = child -> right;

}

}

if(parent->data > num)

parent -> left = temp;

else

parent -> right = temp;

}

}

int main()

{

int switch\_choice;

BSTree object;

char ch = 'y';

while(ch == 'y')

{

cout<<"\t1.CONSTRUCT BST\n\t2.INSERT NODE\n\t3.FIND NUMBER OF NODES IN LONGEST PATH, AND HEIGHT.\n\t4.MINIMUM DATA VALUE.\n\t5.DISPLAY. "<<endl;

cin>>switch\_choice;

switch(switch\_choice)

{

case 1:

{

int num;

cout<<"\nNUMBER OF VALUES TO BE INSERTED?"<<endl;

cin>>num;

object.construct(num);

break;

}

case 2:

{

cout<<"\nENTER DATA TO BE INSERTED."<<endl;

int num;

cin>>num;

object.insert(num);

break;

}

case 3:

{

int height = object.MaxHeight(object.root);

cout<<"\nHEIGHT IS: "<<height<<" AND NUMBER OF NODES IN LONGEST PATH IS "<<(height +1);

cout<<endl;

cout<<endl;

break;

}

case 4:

{

int min = object.Min(object.root);

cout<<"\nMINIMUM ELEMENT OF BST IS: "<<min;

cout<<"\n\n";

break;

}

case 5:

{

cout<<"\nINORDER OF BST IS: ";

object.inorder(object.root);

cout<<endl;

cout<<endl;

break;

}

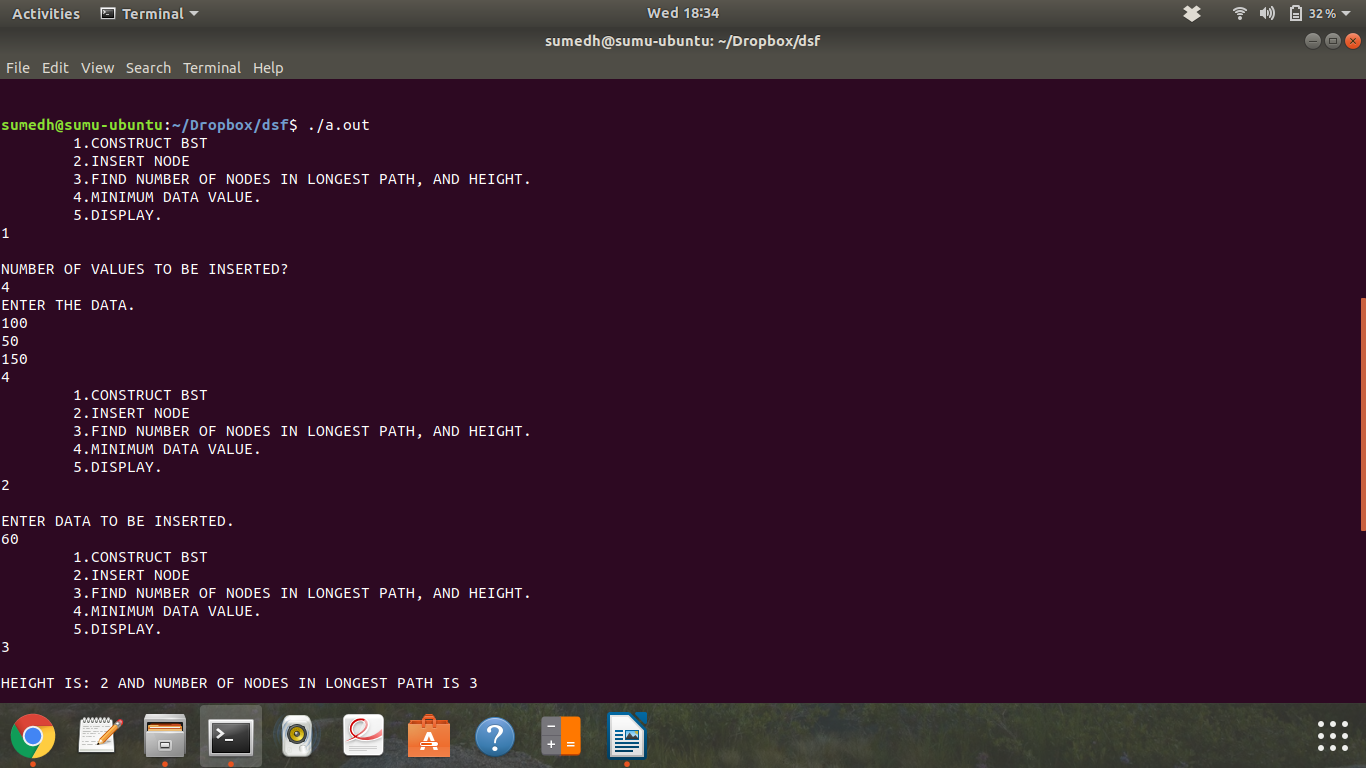
}

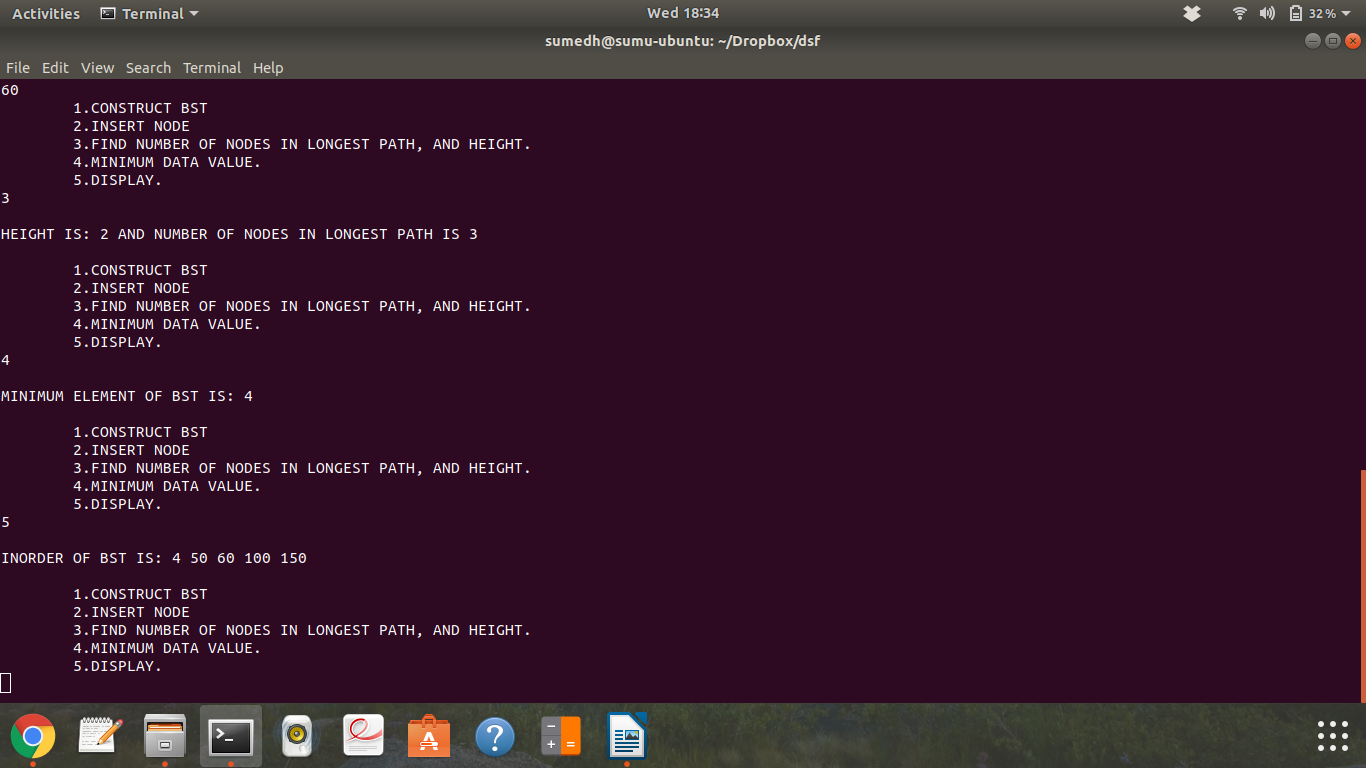
}

return 0;

}

Output:





**Assignment 4**

Aim: Modify BST such that roles of left and right pointers are swapped at every node.

Code:

#include<iostream>

using namespace std;

class Node

{

private:

int data;

Node\* left;

Node\* right;

public:

Node()

{

left = NULL;

right = NULL;

}

friend class BSTree;

};

class BSTree

{

private:

Node\* root;

public:

BSTree()

{

root = NULL;

}

void construct(int n);

void insert(int num);

void inorder(Node\* current);

void mirror(Node\* current);

Node\* give\_root();

};

Node\* BSTree::give\_root()

{

return root;

}

void BSTree::mirror(Node\* current)

{

if(current)

{

swap(current->left, current->right);

mirror(current->left);

mirror(current->right);

}

}

void BSTree::inorder(Node\* current)

{

if(current)

{

inorder(current->left);

cout<<current->data<<" ";

inorder(current->right);

}

}

void BSTree::construct(int num)

{

int data;

cout<<"ENTER THE DATA."<<endl;

for(int i = 0; i<num; i++)

{

cin>>data;

this->insert(data);

}

}

void BSTree::insert(int num)

{

Node\* temp = new Node;

temp->data = num;

Node\* parent, \*child;

if(this->root == NULL)

root = temp;

else

{

child = root;

while(child != NULL)

{

if(child->data > num)

{

parent = child;

child = child->left;

}

else

{

parent = child;

child = child -> right;

}

}

if(parent->data > num)

parent -> left = temp;

else

parent -> right = temp;

}

}

int main()

{

int switch\_choice;

BSTree object;

char ch = 'y';

while(ch == 'y')

{

cout<<"\t1.CONSTRUCT BST\n\t2.INSERT NODE\n\t3.MIRROR IMAGE\n\t4.DISPLAY\n\t5.EXIT "<<endl;

cin>>switch\_choice;

switch(switch\_choice)

{

case 1:

{

int num;

cout<<"\nNUMBER OF VALUES TO BE INSERTED?"<<endl;

cin>>num;

object.construct(num);

break;

}

case 2:

{

cout<<"\nENTER DATA TO BE INSERTED."<<endl;

int num;

cin>>num;

object.insert(num);

break;

}

case 3:

{

object.mirror(object.give\_root());

break;

}

case 4:

{

cout<<"\nINORDER TRAVERSAL OF TREE IS: ";

object.inorder(object.give\_root());

cout<<endl;

cout<<endl;

break;

}

case 5:

{

ch='n';

cout<<"EXITING..."<<endl;

break;

}

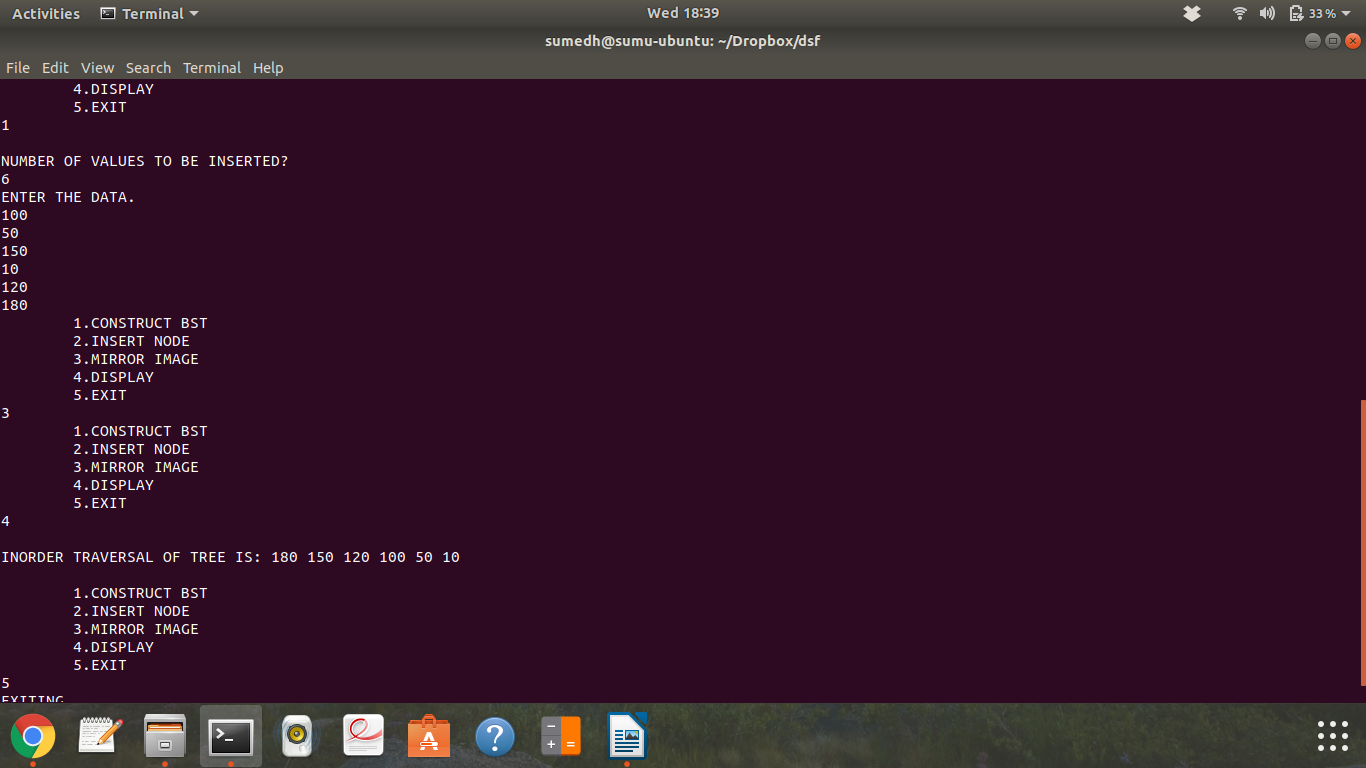
}

}

return 0;

}

OUTPUT:



**Assignment 5**

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

Code:

#include<iostream>

using namespace std;

class node // to push and pop for iterative dfs

{

private:

int num;

node\* next;

public:

node()

{

next = NULL;

}

friend class stack;

friend class depth;

};

class stack // to push and pop for iterative dfs

{

private:

node\* top;

public:

stack()

{

top = NULL;

}

void push(int);

int pop();

friend class depth;

};

void stack::push(int n)

{

node\* temp = new node;

temp->num = n;

if(top == NULL)

top = temp;

else

{

node\* tempy = top;

top = temp;

top->next = tempy;

}

}

int stack::pop()

{

node\* temp = top;

if(top->next == NULL)

top = NULL;

else

{

top = top -> next;

}

temp->next = NULL;

return temp->num;

}

class depth

{

private:

int visited[15]; //to store if vertex is visited or not

int adjacency[15][15]; // to store the adjacency matrix

int vertices; // number of vertices

int edges; // number of edges

public:

depth() // to initialize the visited array and adjacency matrix

{

for(int i = 0; i<15; i++)

visited[i] = 0;

for(int i = 0; i<15; i++)

for(int j = 0; j<15; j++)

adjacency[i][j] = 0;

}

void enter(); // to accept data and prepare adjacency matrix

void recursive\_traversal(int a);

void nonrecursive();

};

void depth::enter()

{

cout<<"\nENTER NUMBER OF VERTICES ";

cin>>vertices;

cout<<"\nENTER NUMBER OF EDGES ";

cin>>edges;

cout<<"\nENTER THE EDGES (E.G. FOR EDGE BETWEEN 0 AND 1 ENTER 0 1) "<<endl;

for(int i = 0; i < edges; i++)

{

int v1,v2;

cin>> v1>>v2;

adjacency[v1][v2] = 1;

adjacency[v2][v1] = 1; // marking the undirected edge in adjacency matrix

}

}

void depth::recursive\_traversal(int a)

{

cout<<a<<" "; // printing the vertex, and marking it as visited

visited[a] = 1;

for(int j= 0; j< vertices; j++)

if(adjacency[a][j] == 1 && visited[j] == 0) // calling function with vertex if it is not visited and current vertex has edge with it

recursive\_traversal(j);

}

void depth::nonrecursive()

{

for(int i = 0; i<vertices; i++) // initializing visited array, as we have traversed recursively before

visited[i] = 0;

stack object;

object.push(0); // push 0 onto stack

while(object.top != NULL) // while stack is not empty

{

int v = object.pop(); // pop vertex, print it, mark it as visited

if(visited[v] == 0)

cout<<" "<<v;

visited[v] = 1;

for(int i = 0; i<vertices; i++) // push all adjacent vertices that are unvisited

if(adjacency[v][i] !=0 && visited [i] == 0)

object.push(i);

}

cout<<endl;

}

int main()

{

depth obj;

obj.enter();

cout<<"DEPTH FIRST TRAVERSAL USING RECURSION: ";

obj.recursive\_traversal(0);

cout<<endl;

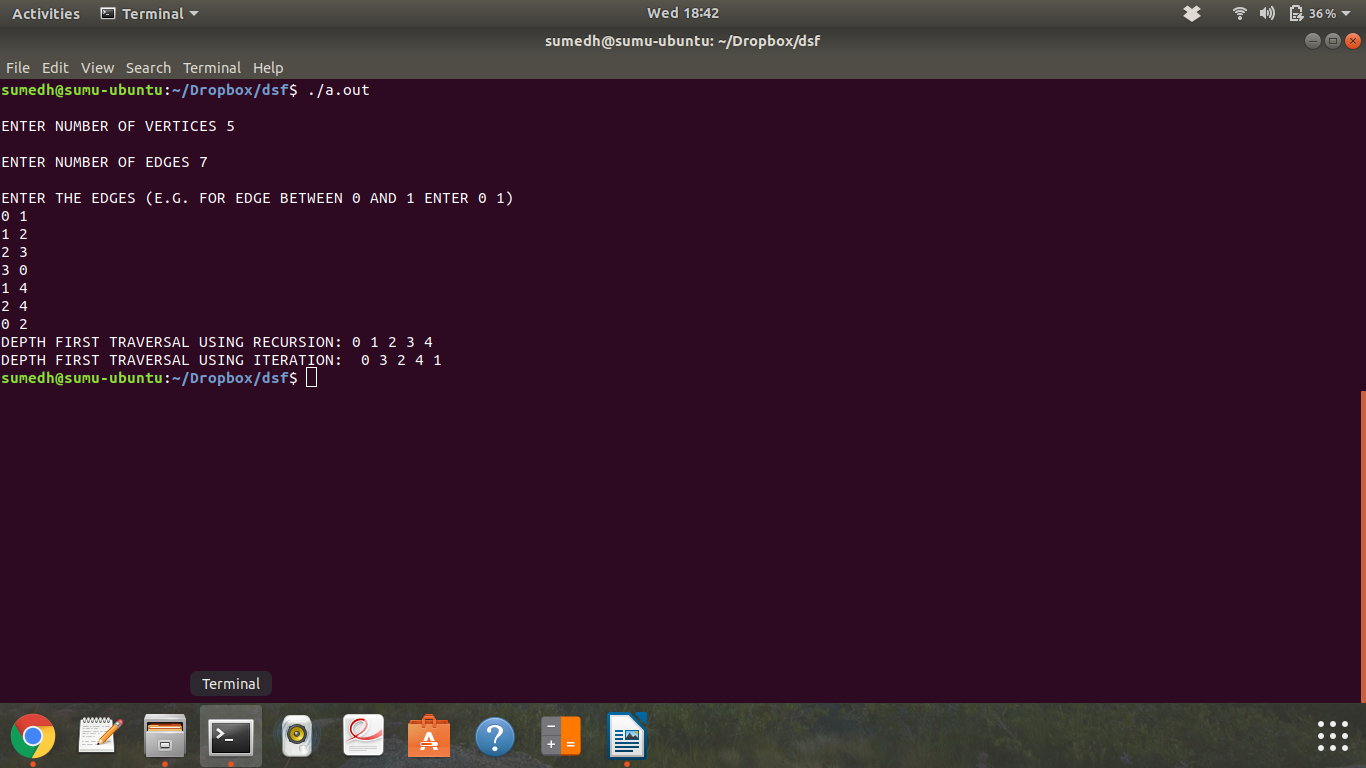
cout<<"DEPTH FIRST TRAVERSAL USING ITERATION: ";

obj.nonrecursive();

return 0;

}

OUTPUT:



**Assignment 6**

Aim: Represent a graph using adjacency list and traverse each node using Breadth first search.

Code:

#include<iostream>

using namespace std;

class GNode // to store the vertices in adjacency list

{

private:

int data;

GNode\* next;

public:

GNode()

{

next = NULL;

}

friend class breadth;

};

class q\_node //to enqueue and dequeue for breadth first search traversal

{

private:

int num;

q\_node\* next\_int;

public:

q\_node()

{

next\_int = NULL;

}

friend class breadth;

};

class breadth

{

private:

GNode\* heads[10]; //array of head pointers which will store head of each linked list

int vertices; //number of vertices

q\_node\* front; //front of queue

q\_node\* rear; //rear of queue

int visited[15]; //visited array to store whether vertex is visited or not

int edges;

public:

void enter(); //accepts graph and prepares it's adjacency list

GNode\* create(int); //creates a GNode and returns it's pointer

void append(int node, int edge);

void display();

breadth()

{

front = NULL;

rear = NULL;

for(int i = 0; i<15; i++)

visited[i] = 0;

}

void enq(int); //enqueue

int deq(); //dequeue

q\_node\* create\_qnode(int); //create a node of the q and return it's pointer

bool qempty(); //check if q is empty

void enq\_adjacent(int); //enq all adjacent vertices of passed vertex

void bfs(); //bfs traversal

};

void breadth::bfs() //traverse using bfs traversal

{

cout<<"\nBREADTH FIRST TRAVERSAL: ";

int v;

enq(0);

while(!qempty())

{

v = deq(); //dequeue

if(visited[v] == 0) // if vertex is not visited, then

{

cout<<" "<< v; // print it

visited[v] = 1; // mark it as visited

enq\_adjacent(v); // and enque all it's adjacent vertices

}

}

cout<<endl;

}

void breadth::enq\_adjacent(int n) //enqueue all the adjacent vertices of accepted vertex

{

GNode\* temp = heads[n];

temp = temp->next;

while(temp)

{

enq(temp->data);

temp = temp->next;

}

}

bool breadth::qempty() //return true if q empty else return false

{

if(front == NULL && rear == NULL)

return true;

else

return false;

}

void breadth::enq(int n) //code for enqueue

{

if(front == NULL && rear == NULL)

{

q\_node\* to\_insert=create\_qnode(n);

front = to\_insert;

rear = to\_insert;

}

else

{

rear->next\_int = create\_qnode(n);

rear = rear->next\_int;

}

}

q\_node\* breadth::create\_qnode(int n)

{

q\_node\* temp = new q\_node;

temp->num = n;

return temp;

}

int breadth::deq() //code for dequeue

{

int to\_pop;

if(front != rear)

{

q\_node\* temp = front;

front = front->next\_int;

to\_pop = temp->num;

temp->next\_int = NULL;

delete temp;

return to\_pop;

}

else if(front == rear)

{

to\_pop = front->num;

front = NULL;

rear = NULL;

return to\_pop;

}

}

void breadth::display() //displays all the edges present in the graph

{

GNode\* temp;

for(int i = 0; i<vertices; i++)

{

cout<<endl;

temp = heads[i]; // temp kept at head of the list, and as we traverse the list, the adjacent vertices are printed

temp = temp->next;

while(temp) // till we reach the end of the list. then we go to the next list using loop, till we print adjacent vertices of all heads

{

cout<<i<<"-"<<temp->data<<", ";

temp = temp->next;

}

}

}

void breadth::enter() // accepts graphs and prepares adjacency lists

{

int num,edge;

cout<<"ENTER THE NUMBER OF VERTICES: ";

cin>>vertices; // accept number of vertices

cout<<"\nENTER THE NUMBER OF EDGES: ";

cin>>edges;

for(int i = 0; i<vertices; i++)

{

heads[i] = create(i); // head nodes created.

}

cout<<"\nENTER THE EDGES: (E.G. ENTER 0 1, IF THERE IS AN EDGE BETWEEN 0 AND 1): "<<endl;

for(int i = 0; i<edges; i++)

{

int v1, v2;

cin>>v1>>v2;

append(v1,v2);

append(v2,v1);

}

}

void breadth::append(int node, int edge) //accept 2 ints (1st: head vertex 2nd: vertex to be appended)

{

GNode\* temp = heads[node];

while(temp->next != NULL)

temp = temp->next;

temp->next = create(edge);

}

GNode\* breadth::create(int n) //create GNode and return it's pointer

{

GNode\* temp = new GNode;

temp->data = n;

return temp;

}

int main()

{

breadth obj;

obj.enter(); //accept graph and prepare adjacency lists

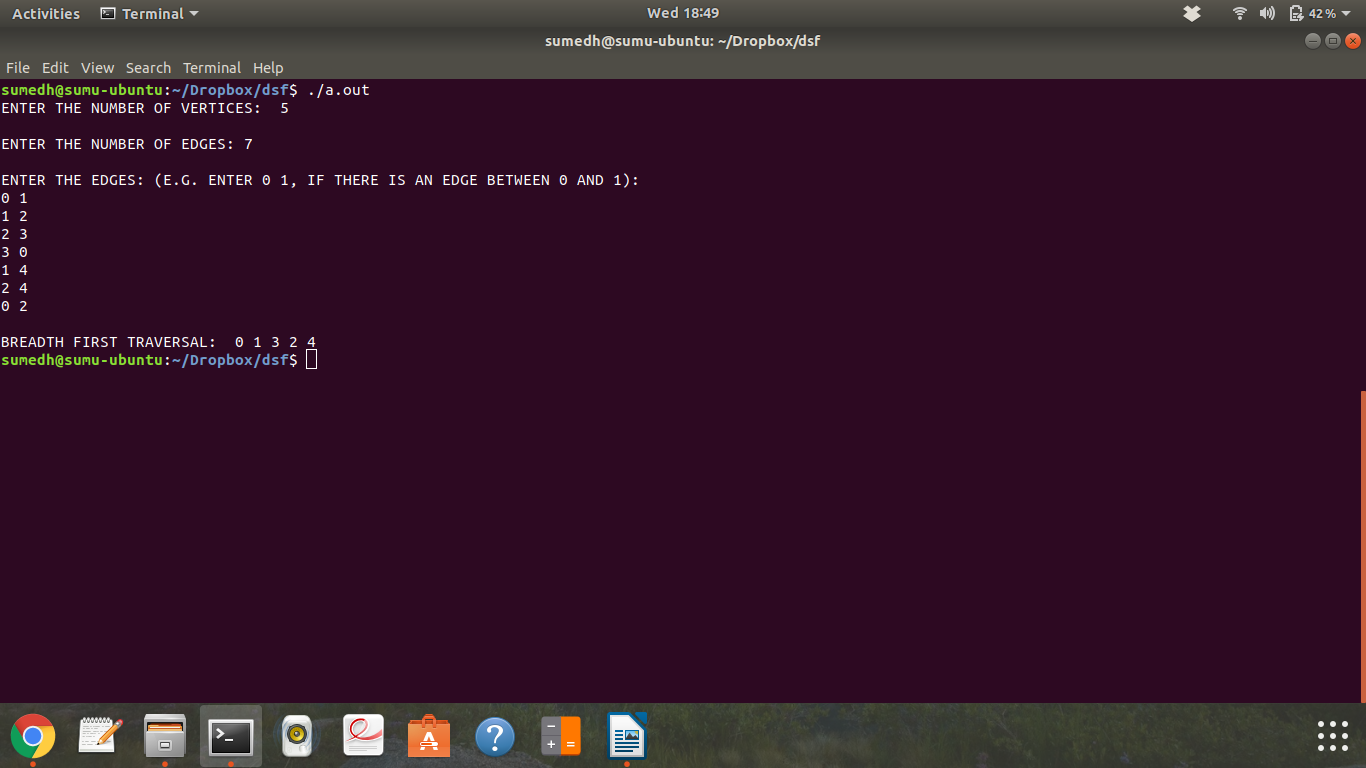
//obj.display();

obj.bfs(); //traverse using bfs

return 0;

}

OUTPUT:



**Assignment 7**

Aim: A customer wants to travel from source A to destination B, he books a cab from source A to reach destination B. Calculate shortest path by avoiding real time traffic to reach destination B.

Code:

#include<iostream>

using namespace std;

class stack\_node

{

public:

int data;

stack\_node\* next;

stack\_node()

{

next = NULL;

}

friend class stack;

friend class Dijkstras;

};

class stack

{

public:

stack\_node\* top;

stack()

{

top = NULL;

}

friend class Dijkstras;

void push(int s);

int pop();

int empty();

};

int stack::empty()

{

if(this->top == NULL)

return 1;

else

return 0;

}

void stack::push(int s)

{

stack\_node\* temp = top;

stack\_node\* n = new stack\_node;

n->data = s;

top = n;

n->next = temp;

}

int stack::pop()

{

stack\_node\* temp = top;

top = top->next;

temp->next = NULL;

return temp->data;

}

class Dijkstras

{

private:

int visited[20]; //visited array will store if node is visited or not

int dist[20]; // it will be initialized with a max value at beginning

int parent[200]; //to store the vertext because of which the vertext got updated

int nodes; //number of nodes

int edges; // number of edges

int adjacency[20][20];

public:

Dijkstras() // initializing all arrays and variables

{

for(int i = 0; i<20; i++)

for(int j= 0; j<20; j++)

adjacency[i][j] = 0; //initializing adjacency matrix to zero

}

void enter(); // will prepare adjacency matrix of weighted graph

void print(int src, int dest); // will print the output of classic djikstra algo (shortest path distance and the path)

void modify(int src, int dest);

};

void Dijkstras::modify(int src, int dest)

{

cout<<"\nENTER THE NUMBER OF MODIFIED EDGES. ";

int x;

cin>>x;

cout<<"\nENTER THE MODIFIED EDGE. ";

int a, b, c;

for(int i = 0; i<x; i++)

{

cin>>a>>b>>adjacency[a][b];

adjacency[b][a] = adjacency[a][b];//for undirected graph

}

for(int i=0; i<nodes; i++)

{

dist[i] = 1000; //initialize distance array

visited[i] = 0; //initialize visited array

}

dist[src] = 0;

parent[dest] = dest;

int min = 0; //to store unvisited vertex which has minimum distance

for(int i = 0; i< (nodes - 1); i++) //pick vertex with min dist, update dist of it's adjacent nodes, update parents etc, nodes-1 times

{

for(int j = 0; j<nodes; j++)

{

if(dist[j] < dist[min] && visited[j] == 0)

min = j;

}

visited[min] = 1;

for(int j = 0 ; j<nodes; j++)

{

if(adjacency[min][j] != 0 && dist[j] > dist[min] + adjacency[min][j]) //if edge exists and the dist of node can be relaxed, relax it and update the parent

{

dist[j] = dist[min] + adjacency[min][j];

parent[j] = min;

}

}

for(int k = 0; k<nodes; k++) //find an unvisited vertex and store min at that vertex

{

if(visited[k] == 0)

{

min = k;

break;

}

}

}

//push the path onto the stack such that top is source and bottom is the destination

stack s;

s.push(dest);

int m1 = dest;

while(parent[m1] != src)

{

s.push(parent[m1]);

m1 = parent[m1];

}

s.push(src);

//displaying the shortest path and it's length

int m;

char tempo;

m = s.pop();

while(s.empty()==0)

{

m = s.pop();

if(m != dest)

{

cout<<"\nTRAVELLED TO "<<m<<" ANY CHANGE IN TRAFFIC? Y/N. ";

cin>>tempo;

if(tempo == 'y' || tempo == 'Y')

{

modify(m, dest);

return;

}

}

else

{

cout<<"\nTRAVELLED TO "<<m<<", DESTINATION REACHED.";

}

}

}

void Dijkstras::print(int src, int dest)

{

for(int i=0; i<nodes; i++)

{

dist[i] = 1000; //initialize distance array

visited[i] = 0; //initialize visited array

}

dist[src] = 0;

parent[dest] = dest;

int min = 0; //to store unvisited vertex which has minimum distance

for(int i = 0; i< (nodes - 1); i++) //pick vertex with min dist, update dist of it's adjacent nodes, update parents etc, nodes-1 times

{

for(int j = 0; j<nodes; j++)

{

if(dist[j] < dist[min] && visited[j] == 0)

min = j;

}

visited[min] = 1;

for(int j = 0 ; j<nodes; j++)

{

if(adjacency[min][j] != 0 && dist[j] > dist[min] + adjacency[min][j]) //if edge exists and the dist of node can be relaxed, relax it and update the parent

{

dist[j] = dist[min] + adjacency[min][j];

parent[j] = min;

}

}

for(int k = 0; k<nodes; k++) //find an unvisited vertex and store min at that vertex

{

if(visited[k] == 0)

{

min = k;

break;

}

}

}

//push the path onto the stack such that top is source and bottom is the destination

stack s;

s.push(dest);

int m1 = dest;

while(parent[m1] != src)

{

s.push(parent[m1]);

m1 = parent[m1];

}

s.push(src);

//displaying the shortest path and it's length

int m;

char tempo;

m = s.pop();

while(s.empty()==0)

{

m = s.pop();

if(m != dest)

{

cout<<"\nTRAVELLED TO "<<m<<" ANY CHANGE IN TRAFFIC? Y/N. ";

cin>>tempo;

if(tempo == 'y' || tempo == 'Y')

{

modify(m, dest);

return;

}

}

else

{

cout<<"\nTRAVELLED TO "<<m<<", DESTINATION REACHED.";

}

}

}

void Dijkstras::enter()

{

int choice;

cout<<"\nPRESS 0 FOR UNDIRECTED GRAPH, 1 FOR DIRECTED GRAPH: ";

cin>>choice;

switch(choice)

{

case 0:

{

cout<<"\nENTER THE NUMBER OF VERTICES IN GRAPH. ";

cin>>nodes;

cout<<"\nENTER THE NUMBER OF EDGES. ";

cin>>edges;

cout<<"\nENTER THE VERTICES AND WEIGHT. (e.g. FOR EDGE BETWEEN 0 AND 1 OF WEIGHT 2, ENTER 0 1 2)"<<endl;

for(int i = 0; i<edges; i++)

{

int v1, v2, weight;

cin>>v1>>v2>>weight;

adjacency[v1][v2] = weight;

adjacency[v2][v1] = weight;

}

break;

}

case 1:

{

cout<<"\nENTER THE NUMBER OF VERTICES IN GRAPH: ";

cin>>nodes;

cout<<"\nENTER THE NUMBER OF EDGES: ";

cin>>edges;

cout<<"\nENTER THE EDGE AND THE WEIGHT (E.G FOR EDGE FROM 0 TO 1, OF WEIGHT 3, ENTER 0 1 3)"<<endl;

for(int i = 0; i<edges; i++)

{

int v1, v2, weight;

cin>>v1>>v2>>weight;

adjacency[v1][v2] = weight;

}

break;

}

}

}

int main()

{

Dijkstras obj; // create object of class

obj.enter(); //enter function will prepare adjacency matrix of the weighted graph

cout<<"\nENTER SOURCE AND DESTINATION VERTEX: ";

int n,m;

cin>>n>>m;

obj.print(n,m);

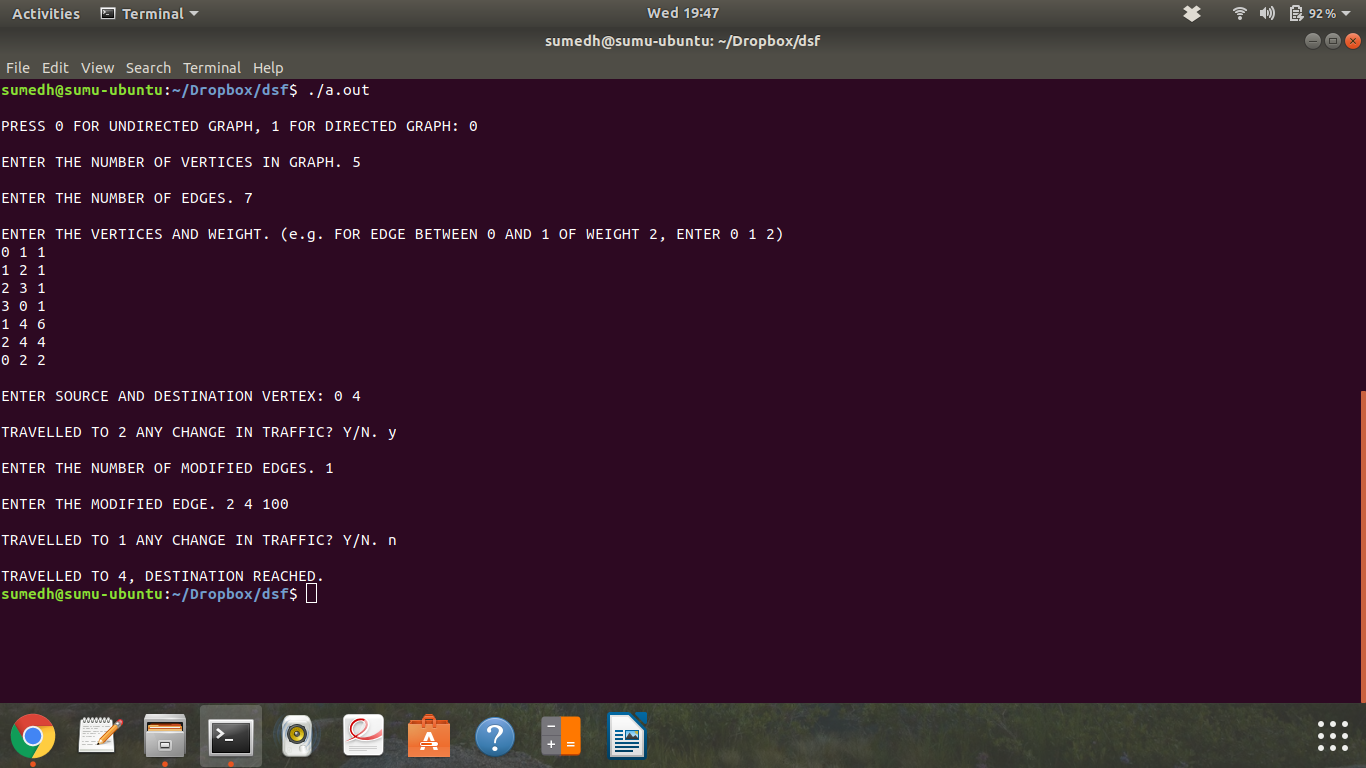
cout<<endl;

return 0;

}

OUTPUT:

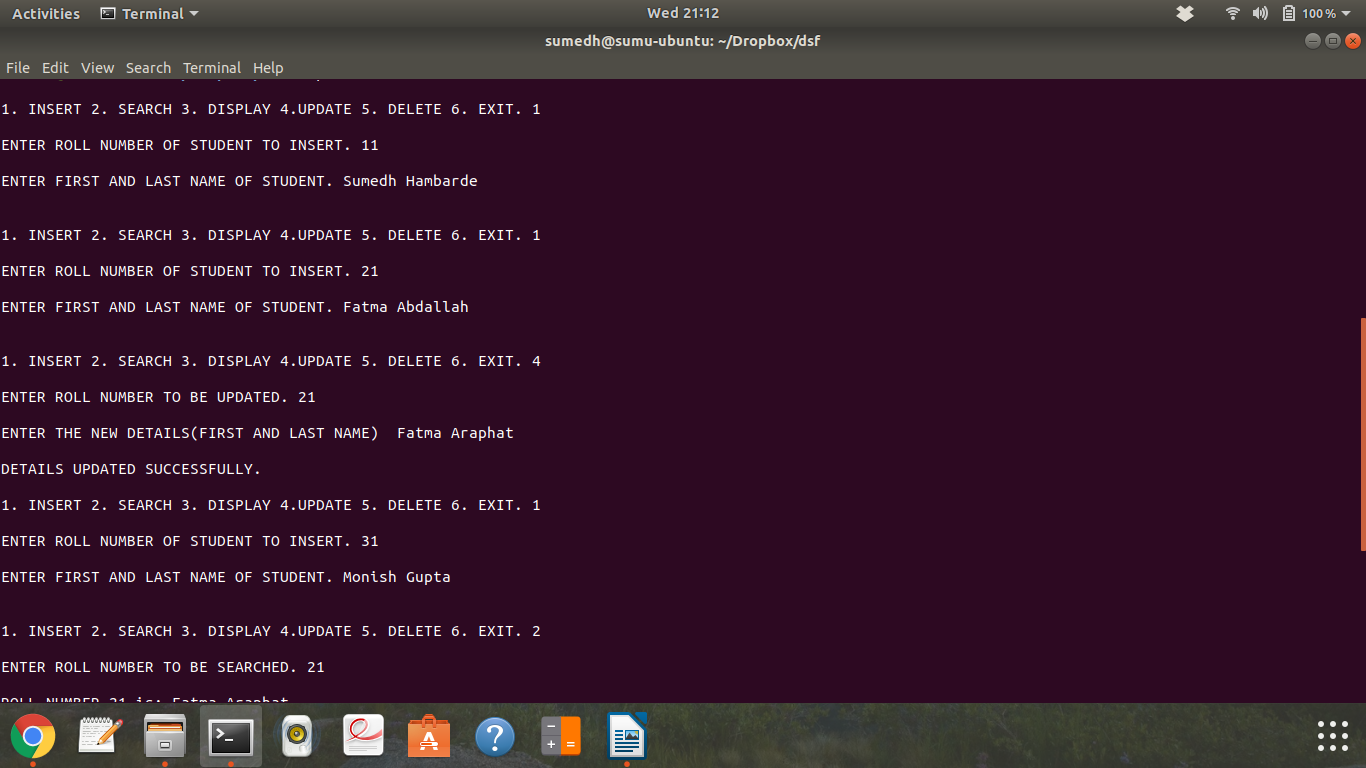
OUTPUT:

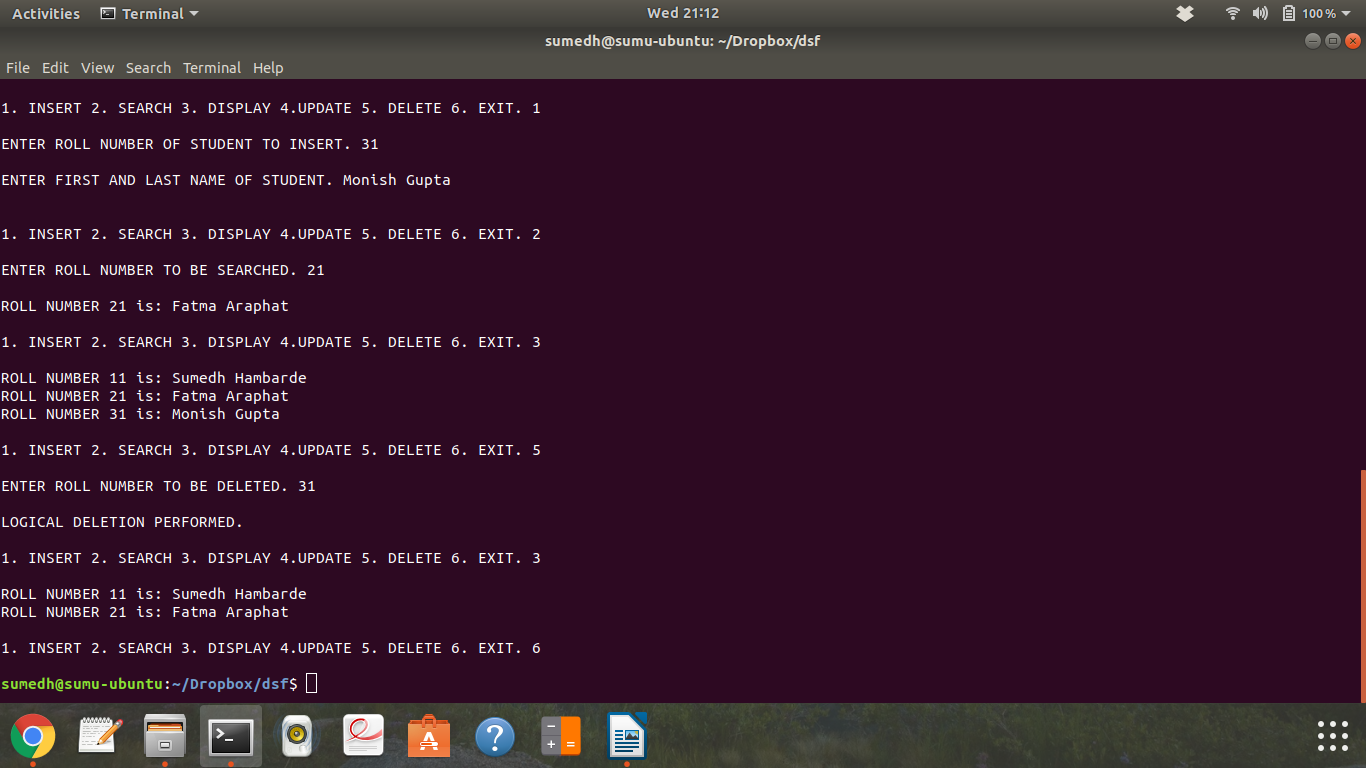


**Assignment 8**

Aim: A node consist of <Key, Value> pair, where nodes are compared and inserted on basis of key. Build a structure such that it should provide a facility of adding a new key, update meaning of a key, delete a key (linear probling without chaining)

Output:





Code:

#include<iostream>

using namespace std;

class student

{

private:

int roll\_no;

string fname;

string lname;

friend class hash\_table;

};

class hash\_table

{

private:

student arr[10];

public:

hash\_table()

{

for(int i = 0; i<10; i++)

arr[i].roll\_no = -1; //initialize all roll numbers to -1 initially, in constructor

}

void insert(int roll);

int search(int roll); //returns index of the record or if record not present returns -1

void display(); //display records of students

void update(int roll); //update record of roll number passed to it

void delete\_record(int roll); //delete record of roll number passed to it

void print\_record(int num); //print details stored at passed index

};

void hash\_table::update(int num)

{

int temp = search(num);

if(temp == - 1)

cout<<"\nNO SUCH ROLL NUMBER IN RECORDS."<<endl;

else

{

string first, second;

cout<<"\nENTER THE NEW DETAILS(FIRST AND LAST NAME) ";

cin>>first>>second;

arr[temp].fname = first;

arr[temp].lname = second;

cout<<"\nDETAILS UPDATED SUCCESSFULLY. "<<endl;

}

}

void hash\_table::delete\_record(int num)

{

int temp = search(num);

if(temp == - 1)

cout<<"\nNO SUCH ROLL NUMBER IN RECORDS."<<endl;

else

{

arr[temp].roll\_no = -2;

cout<<"\nLOGICAL DELETION PERFORMED. ";

}

cout<<endl;

}

void hash\_table::print\_record(int i)

{

cout<<"\nROLL NUMBER "<<arr[i].roll\_no<<" is: "<<arr[i].fname<<" "<<arr[i].lname;

cout<<endl;

}

void hash\_table::insert(int roll)

{

int i;

for( i = 0; i < 10; i++)

{

if( arr[(roll+i)%10].roll\_no < 0) // if position has roll number as -1 or -2, it is empty. insert

break;

}

if(i>= 10)

cout<<"\nHASH TABLE IS FULL.";

else

{

arr[(roll+i)%10].roll\_no = roll;

cout<<"\nENTER FIRST AND LAST NAME OF STUDENT.";

cin>>arr[(roll+i)%10].fname>>arr[(roll+i)%10].lname;

}

cout<<endl;

}

void hash\_table::display()

{

for(int i = 0; i<10; i++)

{

if(arr[i].roll\_no >= 0 ) //if record not blank

cout<<"\nROLL NUMBER "<<arr[i].roll\_no<<" is: "<<arr[i].fname<<" "<<arr[i].lname;

}

cout<<endl;

}

int hash\_table::search(int roll)

{

int i;

for(i = 0; i<10; i++)

{

if (arr[(roll+i)%10].roll\_no != -1) //if the position is not empty from beginning

{

if(arr[(roll+i)%10].roll\_no == roll) //if position contains the roll number to be searched,

{

return ((roll+i)%10); //return index

break;

}

}

else //if the roll number is -1, print no such entry and break

{

return -1;

break;

}

}

if(i>=10)

return -1;

}

int main()

{

hash\_table obj;

int n=0;

int m;

int num;

do

{

cout<<"\n1. INSERT 2. SEARCH 3. DISPLAY 4.UPDATE 5. DELETE 6. EXIT. ";

cin>>n;

switch(n)

{

case 1:

cout<<"\nENTER ROLL NUMBER OF STUDENT TO INSERT.";

cin>>m;

obj.insert(m);

break;

case 2:

cout<<"\nENTER ROLL NUMBER TO BE SEARCHED. ";

cin>>n;

num = obj.search(n);

if(num < 0)

cout<<"\nROLL NUMBER NOT FOUND. ";

else

obj.print\_record(num);

break;

case 3:

obj.display();

break;

case 5:

cout<<"\nENTER ROLL NUMBER TO BE DELETED. ";

cin>>num;

obj.delete\_record(num);

break;

case 4:

cout<<"\nENTER ROLL NUMBER TO BE UPDATED. ";

cin>>num;

obj.update(num);

break;

case 6:

n = 6;

break;

}

}while(n!=6);

cout<<endl;

return 0;

}

OUTPUT:

(BEFORE CODE ^)

**Assignment 9**

Aim: Implement all the functions of a dictionary (ADT) using hashing. Data: Set of (key, value) pairs, Keys are mapped to values. Keys must be comparable, Keys must be unique.Standard operations: Insert(key, value), Find(key), Delete(key) use linear probing with chaining

Code:

include<iostream>

#include<string>

using namespace std;

class hashing

{

typedef struct node

{

string word;

string meaning;

struct node\* next;

}node;

node \*New,\*t;

node\* arr[100];

public:

hashing()

{

for(int i = 0; i<100; i++)

{

arr[i] = new node;

arr[i]->next == NULL;

}

}

void input();

void deletefunc();

void display();

int hash(string word);

void search();

};

void hashing::input()

{

node\* temp;

char ans;

do

{

New=new node;

New->next=NULL;

cout<<"\nEnter word :";

cin>>New->word;

cout<<"\nEnter meaning :";

cin>>New->meaning;

int x=hash(New->word);

if(arr[x]==NULL)

arr[x]=New;

else

{

temp=arr[x];

while(temp->next!=NULL)

temp=temp->next;

temp->next=New;

}

cout<<"\nContinue?";

cin>>ans;

}while(ans=='y');

}

void hashing::deletefunc()

{

string delword;

int flag=0;

cout<<"\nEnter the word to be deleted :";

cin>>delword;

int x=hash(delword);

node \*temp=arr[x];

while(temp!=NULL && temp->word!=delword)

{

temp=temp->next;

}

if(temp==NULL)

cout<<"\nWord not present in dictionary.\n";

else

{

node \*p;

if(flag==0)

{

p=temp;

temp=temp->next;

arr[x]=temp;

delete(p);

display();

}

else

{

p=temp->next;

temp->next=(temp->next)->next;

delete(p);

display();

}

}

}

int hashing::hash(string word)

{

int i=0,sum=0,x=0;

while(i<word.length())

{

x=word[i];

sum+=x;

i++;

}

sum=sum%10;

return sum;

}

void hashing::search()

{

string person;

cout<<"\nEnter the word to find it's meaning :";

cin>>person;

int x=hash(person);

node \*temp=arr[x];

while(temp!=NULL && temp->word!=person)

temp=temp->next;

if(temp==NULL)

cout<<"\nWord not present in dictionary.\n";

else

cout<<"Required meaning is :"<<temp->meaning;

}

void hashing::display()

{

int j=0;

while(j<10)

{

node \*temp=arr[j];

if(temp==NULL)

{

j++;

continue;

}

else

{

while(temp!=NULL)

{

cout<<temp->word<<"\t";

cout<<temp->meaning<<endl;

temp=temp->next;

}

}

j++;

}

}

int main()

{

hashing s;

s.input();

s.display();

s.deletefunc();

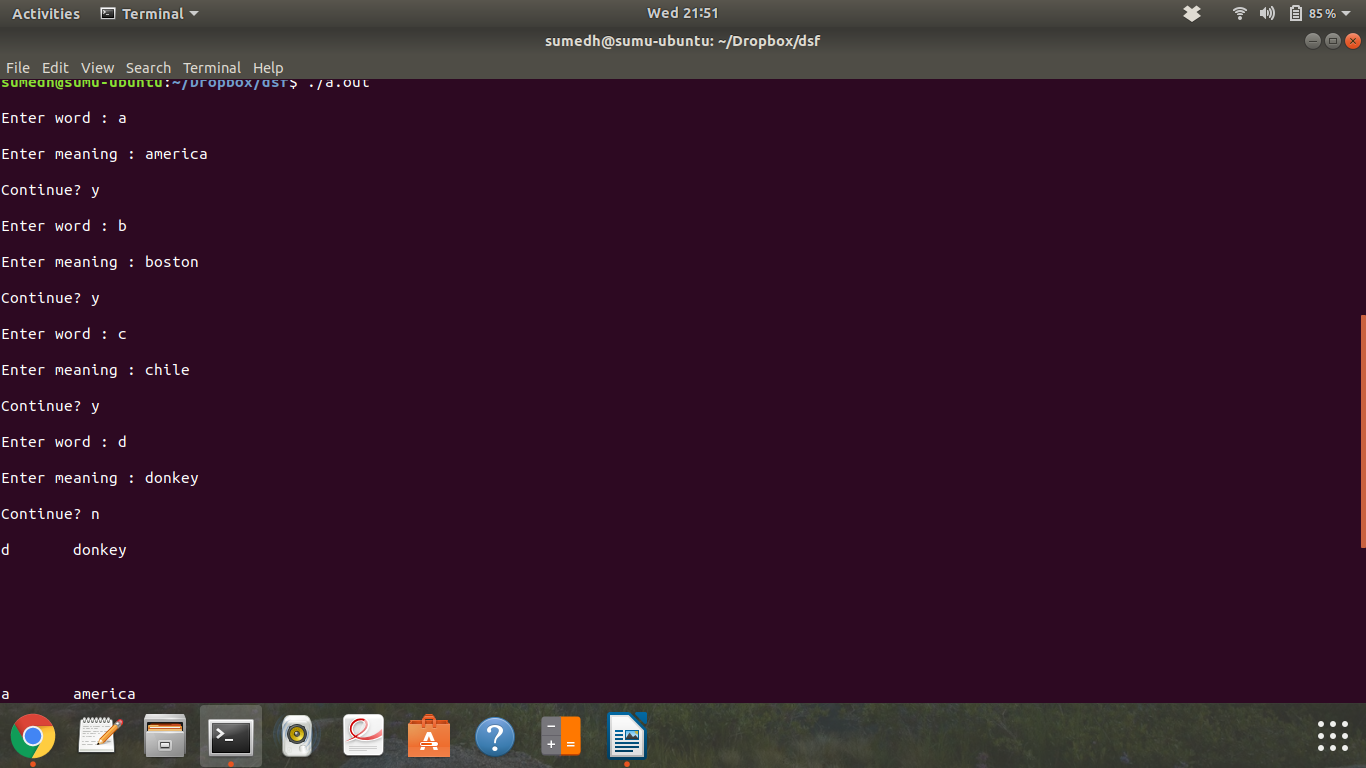
s.search();

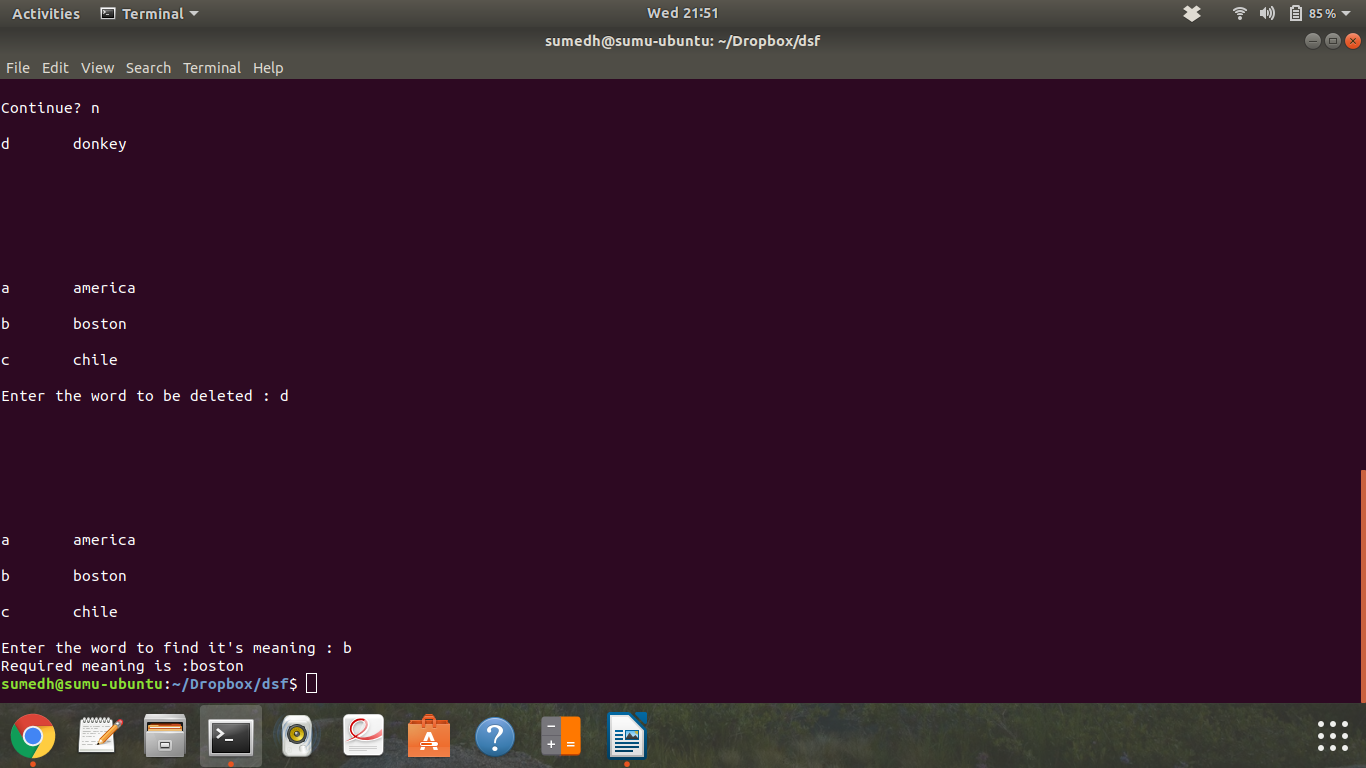
cout<<endl;

return 0;

}

OUTPUT:





**Assignment 10**

Aim:A dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/desecending order. Also find how many maximum comparisons may require for finding any keyword. Use height-balanced tree and find complexity for finding a keyword.

Code:

#include<iostream>

#include<string.h>

using namespace std;

typedef struct node

{

string word;

string meaning;

int fact;

node \*left;

node \*right;

}node;

class bst

{

public:

int diff(node\*);

void display(node\*);

node\* LL(node \*);

node\* RR(node\*);

node\* RL(node\*);

node\* LR(node\*);

int search(node\*,string);

node\* insert(node\*,string,string);

node\* balance(node\*);

int height(node\*);

void update(node\*,string);

void disprev(node\*);

};

void bst::update(node \*root,string key)

{

int temp;

temp = search(root,key);

if(temp==1)

{

//cout<<"\nCurrent meaning of "<<root->word<<" is "<<temp->meaning;

}

else

{

cout<<"\nDoesn't Exist\n";

}

}

int bst::search(node \*root,string key)

{

if(root==NULL)

{

return 0;

}

if(key.compare(root->word)==0)

{

cout<<"\n key is : "<<key ;

cout<<"\n root->word : "<<root->word<<" meaning : "<<root->meaning;

cout<<"\nEnter the new meaning\n";

cin>>root->meaning;

return 1;

}

else if(key.compare(root->word)<0)

{

search(root->left,key);

}

else if(key.compare(root->word)>0)

{

search(root->right,key);

}

//return NULL;

}

int bst::height(node \*temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = height (temp->left);

int r\_height = height (temp->right);

int max\_height = max (l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

int bst::diff(node \*temp)

{

int l = height(temp->left);

//cout<<"\nL : "<<l;

int r = height(temp->right);

//cout<<"\nR : "<<r<<endl;

int f = l-r;

return f;

}

node\* bst::LL(node \*parent)

{

node \*temp;

temp = parent->left;

parent->left = temp->right;

temp->right = parent;

return temp;

}

node\* bst::RR(node \*parent)

{

node \*temp;

temp = parent->right;

parent->right = temp->left;

temp->left = parent;

return temp;

}

node\* bst::LR(node \*parent)

{

node \*temp;

temp = parent->left;

parent->left = RR(temp);

return LL(parent);

}

node\* bst::RL(node \*parent)

{

node \*temp;

temp = parent->right;

parent->right = LL(temp);

return RR(parent);

}

node\* bst::balance(node \*temp)

{

int bal\_factor = diff (temp);

if (bal\_factor > 1)

{

if (diff (temp->left) > 0)

temp = LL(temp);

else

temp = LR(temp);

}

else if (bal\_factor < -1)

{

if (diff (temp->right) > 0)

temp = RL(temp);

else

temp = RR(temp);

}

return temp;

}

node\* bst::insert(node \*root, string value,string mean)

{

if (root == NULL)

{

root = new node;

root->word = value;

root->meaning = mean;

root->left = NULL;

root->right = NULL;

return root;

}

else if ((value.compare(root->word))<0)

{

root->left = insert(root->left, value,mean);

root = balance (root);

}

else if ((value.compare(root->word))>0)

{

root->right = insert(root->right, value,mean);

root = balance (root);

}

return root;

}

void bst::display(node\* root)

{

if(root)

{

display(root->left);

cout<<root->word<<" means "<<root->meaning<<endl;

display(root->right);

}

}

void bst::disprev(node \*root)

{

if(root)

{

disprev(root->right);

cout<<root->word<<" means "<<root->meaning<<endl;

disprev(root->left);

}

}

int main()

{

bst obj;

node \*rt;

int ch;

int choice;

char x;

string key,mean;

string keys;

rt = NULL;

do

{

cout<<"\nEnter Choice\n";

cout<<"\n1.Insert the data into dictionary. \n2.Update the dictionary ";

cout<<"\n3.Display the data in ascending order.\n4.Display the data in descending order";

cin>>choice;

switch(choice)

{

case 1:

do

{

cout<<"\nEnter the word\n";

cin>>key;

cout<<"\nEnter the meaning of the word\n";

cin>>mean;

rt = obj.insert(rt,key,mean);

//cout<<"\nROOT is : "<<rt->word;

cout<<"\nDo you want to insert more words? Y or N\n";

cin>>x;

}while(x=='y'||x=='Y');

break;

case 2:

cout<<"\nEnter the word whose meaning is supposed to be updated\n";

cin>>keys;

obj.update(rt,keys);

break;

case 3:

cout<<"\nDisplay :\n";

obj.display(rt);

break;

case 4:

cout<<"Reverse display:\n";

obj.disprev(rt);

break;

}

cout<<"\nDo you wish to continue? Y or N \n";

cin>>x;

}while(x=='y' || x=='Y');

return 0;

}

OUTPUT:

