ASSIGNMENT 1

Aim:

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

Objective:

1. Understand the concept of expression tree and binary tree.

2. Understand the recursive traversal of an expression tree

Theory:

1. Definition of an expression tree with diagram.

Algebraic expressions such as a/b + (c-d) e The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (a, b, c, d, and e). The non-terminal nodes of an expression tree are the operators (+, -, , and ). Notice that the parentheses which appear in Equation do not appear in the tree. Nevertheless, the tree representation has captured the intent of the parentheses since the subtraction is lower in the tree than the multiplication.

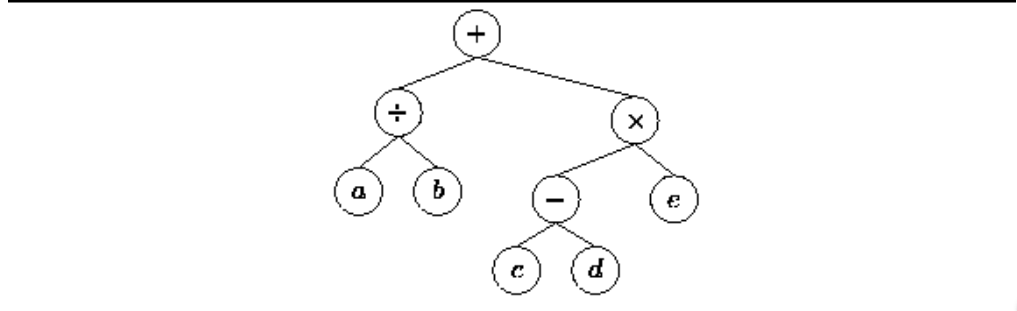


Figure: Tree representing the expression a/b+(c-d)e

1. Show the different type of traversals with example

To traverse a non-empty binary tree in preorder,

1. Visit the root.

2. Traverse the left subtree.

3. Traverse the right subtree.

To traverse a non-empty binary tree in inorder:

1. Traverse the left subtree.

2. Visit the root.

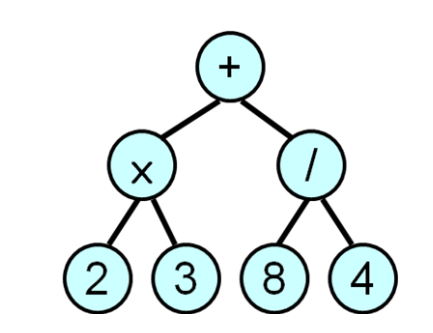
3. Traverse the right subtree.

To traverse a non-empty binary tree in postorder,

1. Traverse the left subtree.

2. Traverse the right subtree.

3. Visit the root.



Pre-order (prefix) +  2 3 / 8 4

In-order (infix) 2  3 + 8 / 4

Post-order (postfix) 2 3  8 4 / +

ALGORITHM:

Define a class for Binary Tree (Information, Left Pointer & Right Pointer)

* **Create Expression Tree:**

CreateTree() Root& Node pointer variable of type structure. Stack is an pointer array of type class. String is character array which contains postfix expression. Top is a variable of type node. t,tn,l,r of type treenode.

Step 1: Top = NULL,

Step 2: Do Steps 3,4,5 While String[I] != NULL

Step 3: Create Node of type of class

Step 4: node->data=String[I];

Step 5:

if(( (String[I]>=65 )&& (String[I]<=90) )||( (String[I]>=97) && (String[I]<=122 ) ) )

If top==NULL then, node->nxt=top; top=node;

Else top=node

Else

l=pop()

r=pop()

tn->lptr=l

tn->rptr=r

push(tn)

step 6: displaying traversals

for inorder traversal, displayinorder(root)

for preorder traversal, displaypreorder(root)

for postorder traversal, displaypostorder(root)

step 7: end

* **Inorder Traversal Recursive algorithm :**

Tree is pointer of type class.

displayinorder (root)

Step 1: If root!= NULL

Step 2: displayinorder ( root ->lptr)

Step 3: Print root -> expression

Step 4: displayinorder ( root -> rptr)

* **Postorder Traversal Recursive algorithm:**

Tree is pointer of type class.

displaypostorder (root)

Step 1: If root!= NULL

Step 2: displaypostorder ( root -> lptr)

Step 3: displaypostorder ( root -> rptr)

Step 4: Print root -> expression

* **Preorder Traversal Recursive algorithm:**

Tree is pointer of type class.

displaypreorder (root)

Step 1: If root!= NULL

Step 2: Print root ->expression

Step 3: displaypreorder ( root -> lptr)

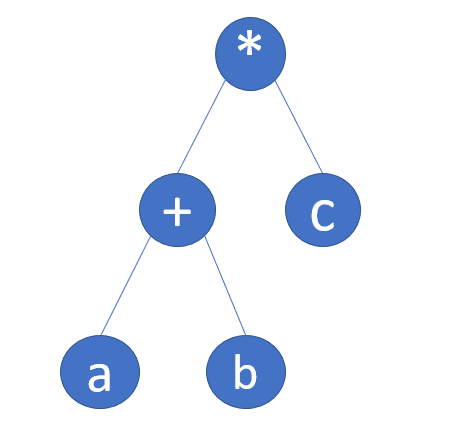
Step 4: displaypreorder ( root ->rptr )

**INPUT:**

Postfix Expression: a b + c \*

**OUTPUT:**

Display result of each operation with error checking.

**Expression tree:**

**OUTPUT:**

Inorder traversal :  a+b\*c

Preoreder traversal : \*a+bc

Postorder traversal :  a+bc\*

CPP SOURCE CODE:

#include<iostream>

using namespace std;

// treenode class containing data as well as create() function for creating a tree-node

class treenode{

public:

char expression;

treenode \*lptr;

treenode \*rptr;

treenode()

{

lptr=rptr=NULL;

}

treenode \*create(char c,treenode \*l,treenode \*r)

{

treenode \*newnode;

newnode=new treenode;

newnode->expression=c;

newnode->lptr=l;

newnode->rptr=r;

return newnode;

}

};

// node() function for having data which we want the node of a stack to contain

class node{

public:

treenode \*tptr;

node \*nxt;

};

// stack() function containing push(),pop() and display() functions

class stack{

public:

node \*top;

stack()

{

top=NULL;

}

void push(treenode \*a){

node \*b;

b=new node;

b->tptr=a;

b->nxt=NULL;

if(top)

{

b->nxt=top;

top=b; }

else{

top=b;

}

}

treenode \*pop()

{

node \*a;

a=top;

top=top->nxt;

treenode \*tn=a->tptr;

delete a;

return (tn);

}

void display()

{

cout<<"\n The inorder traversal of entered postfix expression : ";

displayinorder(top->tptr);

cout<<"\n The pre-order traversal of entered postfix expression : ";

displaypreorder(top->tptr);

cout<<"\n The post-order traversal of entered postfix expression : ";

displaypostorder(top->tptr);

cout<<"\n\n";

}

void displayinorder(treenode \*root)

{

if(root)

{

displayinorder(root->lptr);

cout<<root->expression;

displayinorder(root->rptr);

}

}

void displaypreorder(treenode \*root)

{

if(root)

{

cout<<root->expression;

displayinorder(root->lptr);

displayinorder(root->rptr);

}

}

void displaypostorder(treenode \*root)

{

if(root)

{

displayinorder(root->lptr);

displayinorder(root->rptr);

cout<<root->expression;

}

}

};

int main()

{

char e[50];

// we'll enter a postfix expression having maximum length of 50 characters

cout<<"\n enter the postfix expression : ";

cin>>e;

stack s;

treenode t,\*tn,\*l,\*r;

for(int i=0;e[i]!='\0';i++)

{

// checking if the entered character is an alphabet (operand)

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

{

tn=t.create(e[i],NULL,NULL);

s.push(tn);

}

else

{

// checking if the entered character is an operator

r=s.pop();

l=s.pop();

tn=t.create(e[i],l,r);

s.push(tn);

}

}

s.display();

return 0;

}

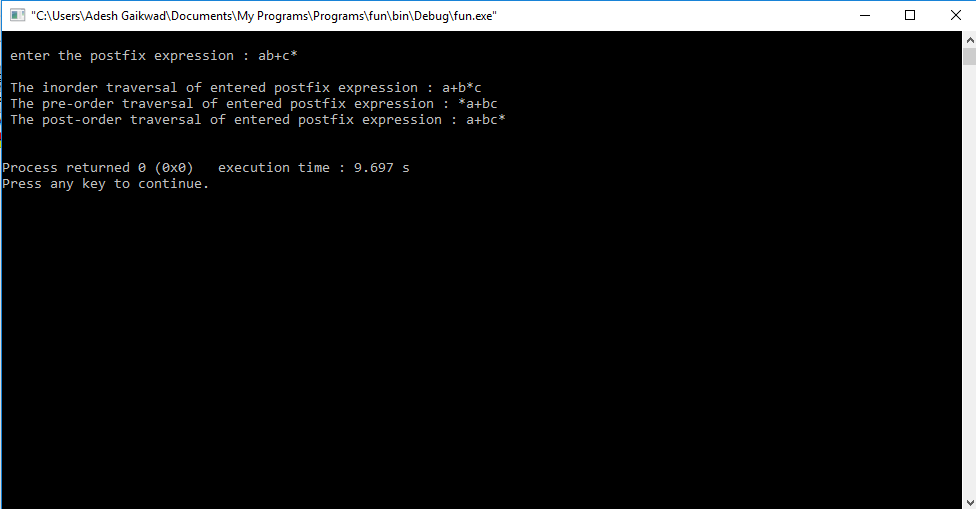
**OUTPUT:**

enter the postfix expression : ab+c\*

 The inorder traversal of entered postfix expression : a+b\*c

 The pre-order traversal of entered postfix expression : \*a+bc

 The post-order traversal of entered postfix expression : a+bc\*



**CONCLUSION:**

Through this assignment, we learnt and performed how to do recursive inorder, preorder and postorder traversal of an expression tree.

**DSF ASSIGNMENT 2**

**AIM :**

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

**THEORY:**

**Algorithm :**

**Non-recursive Inorder traversal**

1. 1) Create an empty stack S
2. 2) Initialize *Current* node as root
3. 3) Push the *Current* node to S; Set *Current* = *Current* ->left
4. 4) Until *Current* is NULL
5. 5) Push Current
6. 6) Set *Current* = *Current* ->left
7. 7) If current is NULL and stack is not empty then
8. a) Pop from the stack S
9. b) Print the popped item
10. c) Set current = popped\_item->right
11. d) Go to step 3
12. 8) If current is NULL and stack is empty then return

**Non-recursive Preorder**

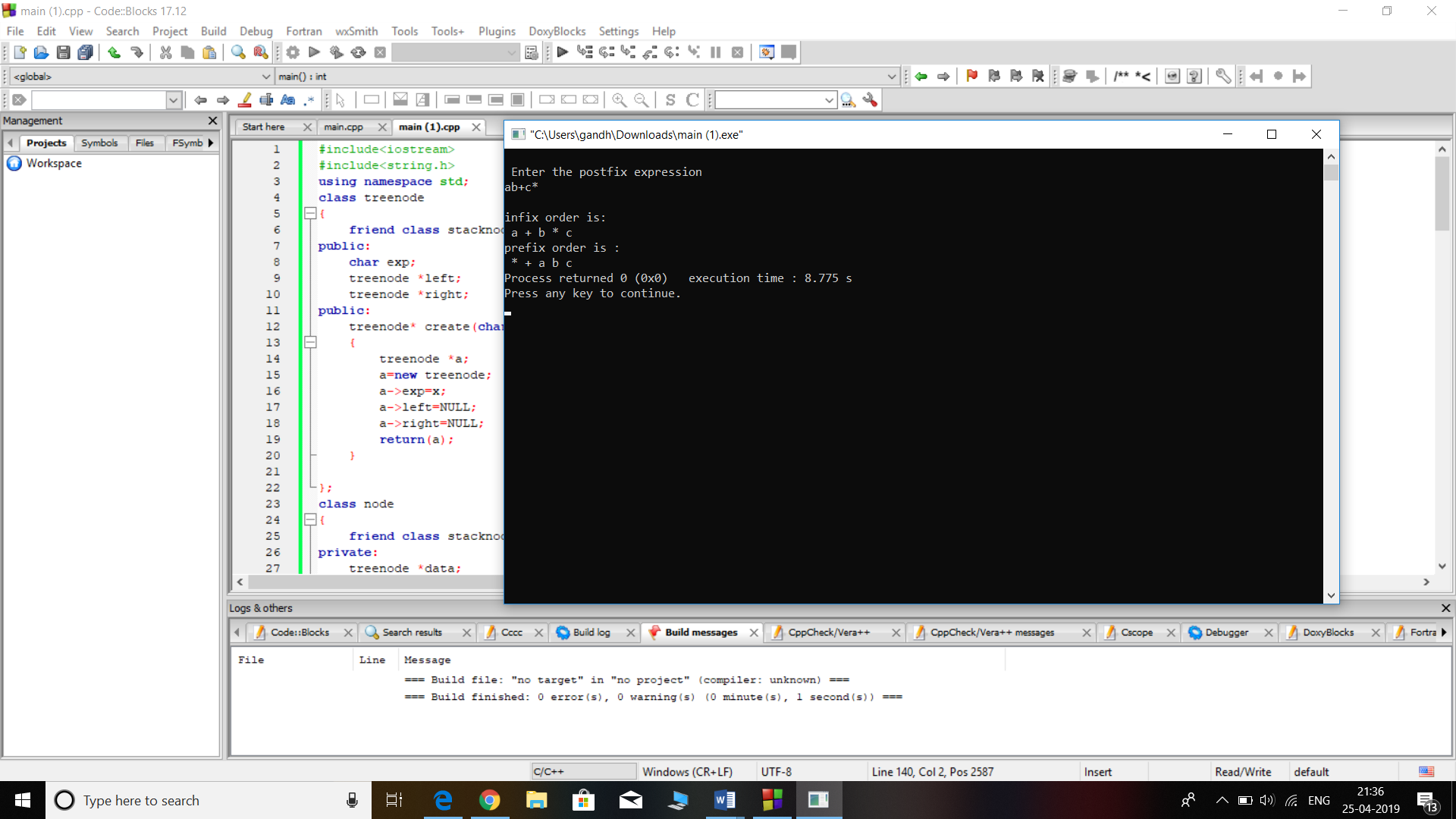
* Create an empty stack; make current=root
* Push current node to stack  
  Do following while is not empty

**a)** Pop an item from stack and print it.  
 **b)** Push right child of popped item to stack

**c)** Push left child of popped item to stack

*Note: Right child is pushed before left child to make sure that left subtree is processed first*

OUTPUT:



**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:

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

The left and right sub trees are also binary search tree

200px-Binary_search_tree.svg

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

/ \ / \

10 30 10 30

\

40

**Algorithm:**

To insert a new value *v* into a binary search tree *T*, we use the procedure TREE-INSERT. The procedure is passed a node *z* for which *key*[*z*] = *v*, *left*[*z*] = NIL, and *right*[*z*] = NIL. It modifies *T* and some of the fields of *z* in such a way that *z* is inserted into an appropriate position in the tree.

TREE-INSERT(*T*,*z*)

1 y http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif NIL

2 x http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif root[T]

3 while x http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/noteq.gif NIL

4 do y http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif x

5 if key[z] < key[x]

6 then x http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif left[x]

7 else x http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif right[x]

8 p[z] http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif y

9 if y = NIL

10 then root[T] http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif z

11 else if key[z] < key[y]

12 then left[y] http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif z

13 else right[y] http://staff.ustc.edu.cn/~csli/graduate/algorithms/images/arrlt12.gif z

**Program:**

**/\***

**Problem Statement : 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**

**\*/**

#include<iostream>

#include<math.h>

using namespace std;

struct tnode{

int data;

tnode \*lptr,\*rptr;

};

tnode \*root=NULL;

tnode \*insert(int val);

void create(int val);

void inorder(tnode \*);

void preorder(tnode \*);

void postorder(tnode\*);

int minimum(tnode \*);

int height(tnode \* );

int main()

{

int num,a,choise;

cout<<"Nodes in tree : ";

cin>>a;

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

{

cout<<"Enter the data : ";

cin>>num;

create(num);

}

do {

cout<<"-------------------------------------------------------------------------\n";

cout<<"\n\t\t\tHave Fun with Binary Search Tree!!\n";

cout<<"\n\t\t1.Inser a new node\n\t\t2.Find heiht of tree\n\t\t3.Minimum node present in Tree \n\t\t4.Inorder Traversal\n\t\t5.Exit\n\t\tYour choise : ";

cin>>choise;

switch(choise)

{

case 1:

cout<<"Enter the data : ";

cin>>num;

create(num);

cout<<"\tNode inserted Successflly in Tree!!\n";

break;

case 2:

cout<<"The Height of tree is : ";

cout<<height(root)<<endl;

break;

case 3:

cout<<"The minimum of tree is : ";

cout<<minimum(root)<<endl;

break;

case 4:

cout<<"Inorder Traversal of Tree : \n";

inorder(root);

break;

}

}

while(choise<5);

}

tnode \*insert(int val)

{

tnode \*newNode =new tnode;

newNode->data=val;

newNode->lptr =newNode->rptr =NULL;

return newNode;

}

void create(int val)

{

tnode \*parent,\*current=root;

if(root==NULL)

{

root = insert(val);

}

else

{

while(current!=NULL)

{

parent =current;

if(val<current->data)

current =current->lptr;

else

current = current->rptr;

}

if(val<parent->data)

parent->lptr=insert(val);

else

parent->rptr =insert(val);

}

}

void inorder(tnode \*Root)

{

tnode \*temp =Root;

if(temp==NULL)

return ;

inorder(temp->lptr);

cout<<temp->data<<endl;

inorder(temp->rptr);

}

void preorder(tnode \*Root)

{

tnode \*temp=Root;

if(temp==NULL)

return ;

cout<<temp->data<<endl;

preorder(temp->lptr);

preorder(temp->rptr);

}

void postorder(tnode \*Root)

{

tnode \*temp =Root;

if(temp==NULL)

return ;

preorder(temp->lptr);

preorder(temp->rptr);

cout<<temp->data<<endl;

}

int minimum(tnode \*Root)

{

tnode \*parent,\*temp = Root;

while(temp!=NULL)

{

parent = temp ;

temp =temp->lptr;

}

return parent->data;

}

int height(tnode \*Root)

{

tnode \*current =Root;

if(current == NULL)

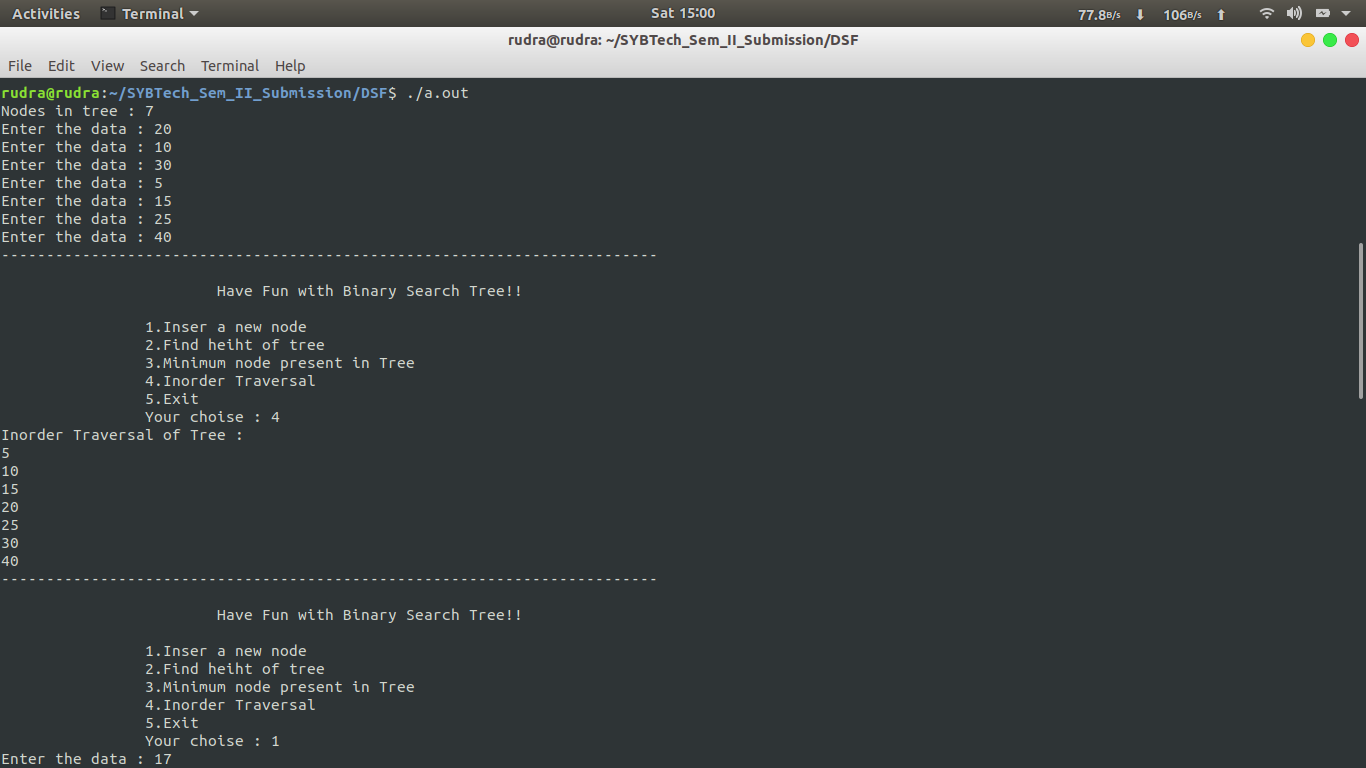
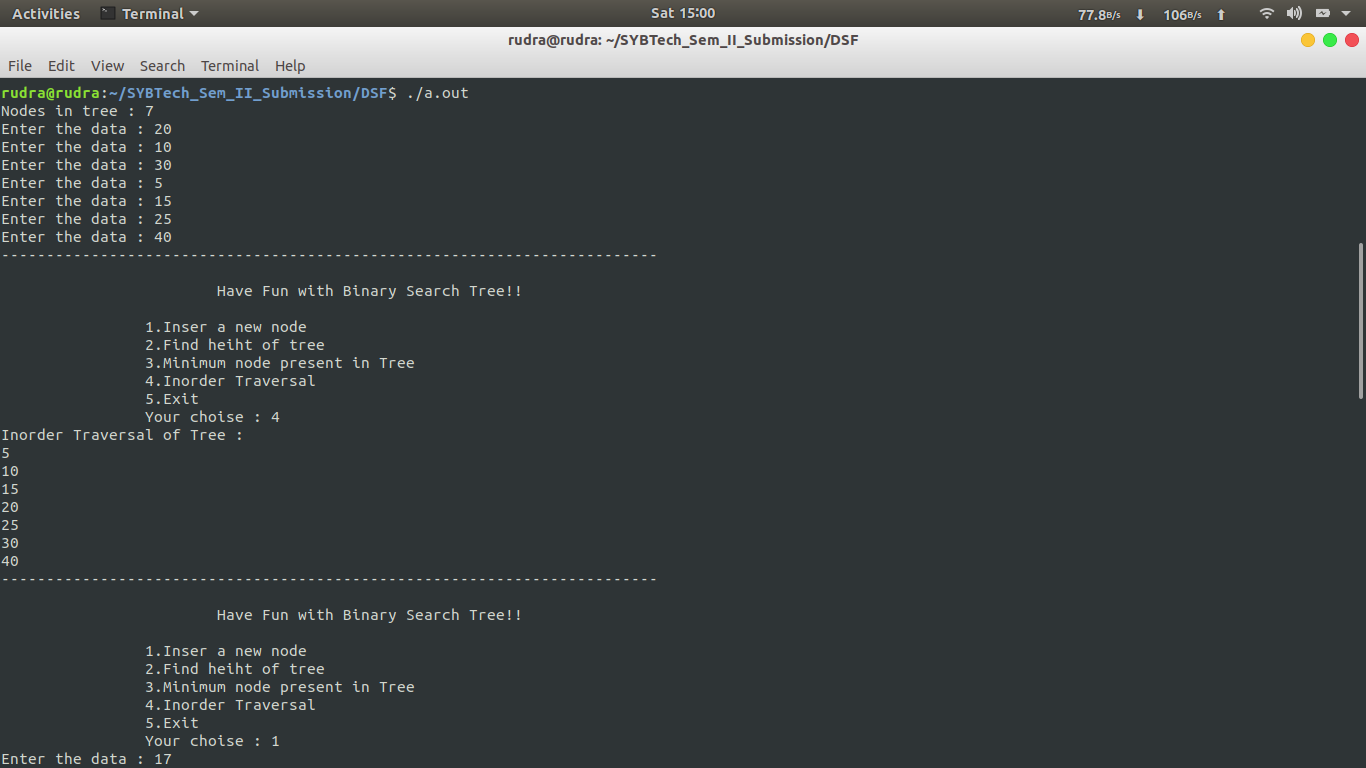
return 0;

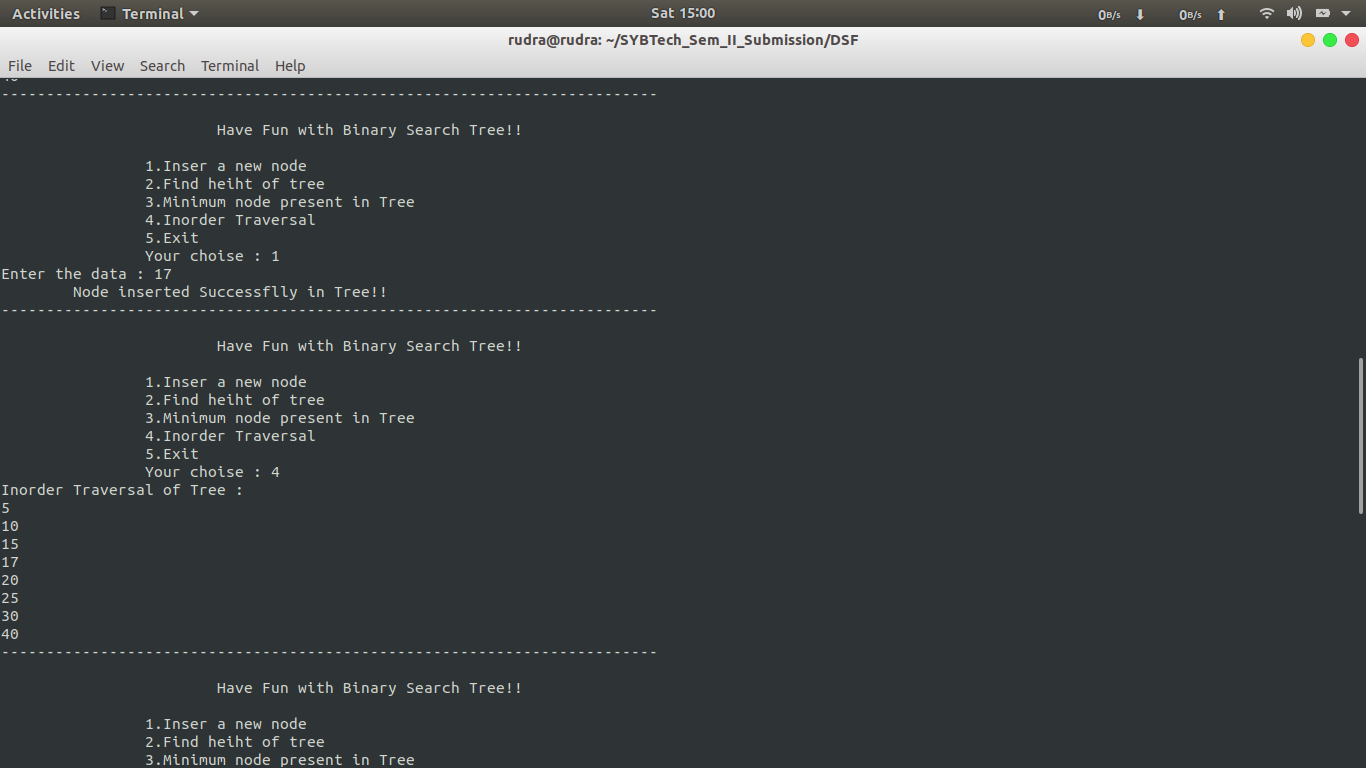
int ht = max(height(current->lptr),height(current->rptr));

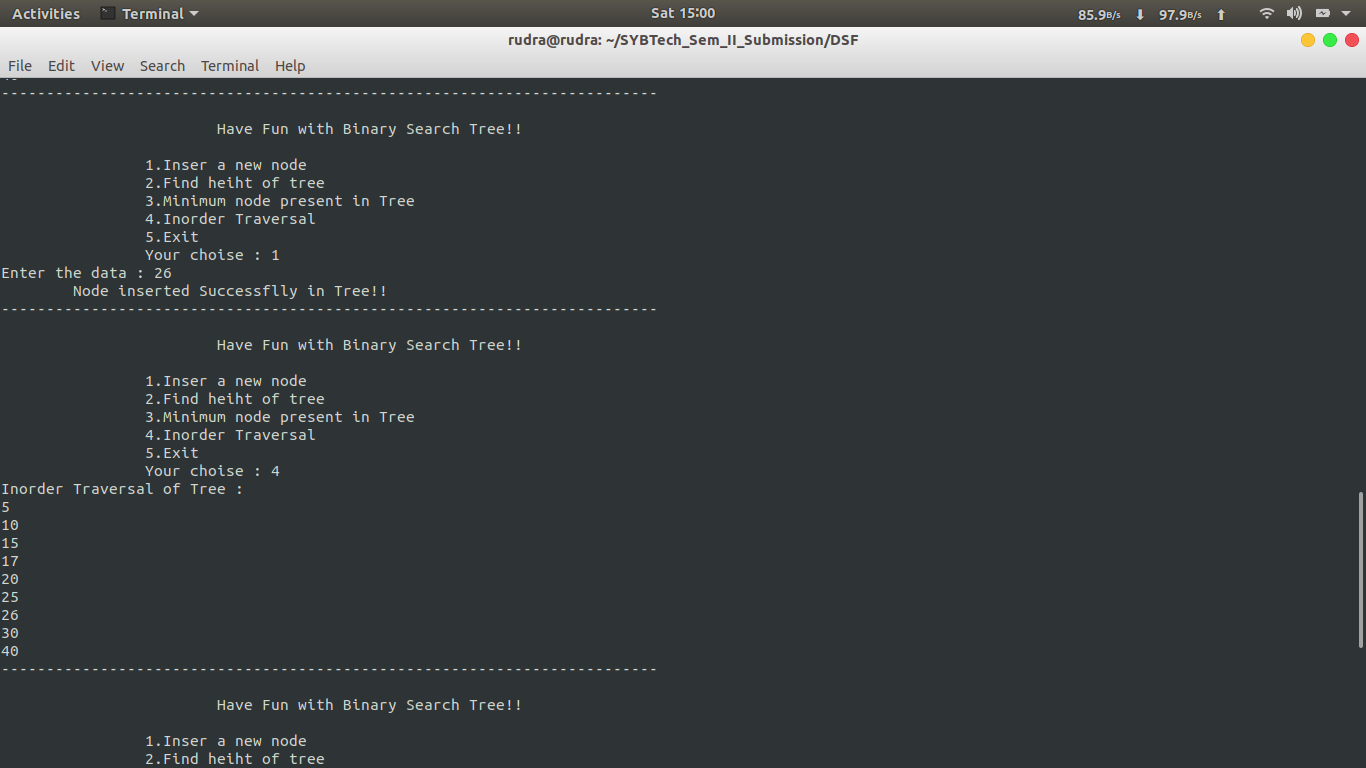
return ht+1;

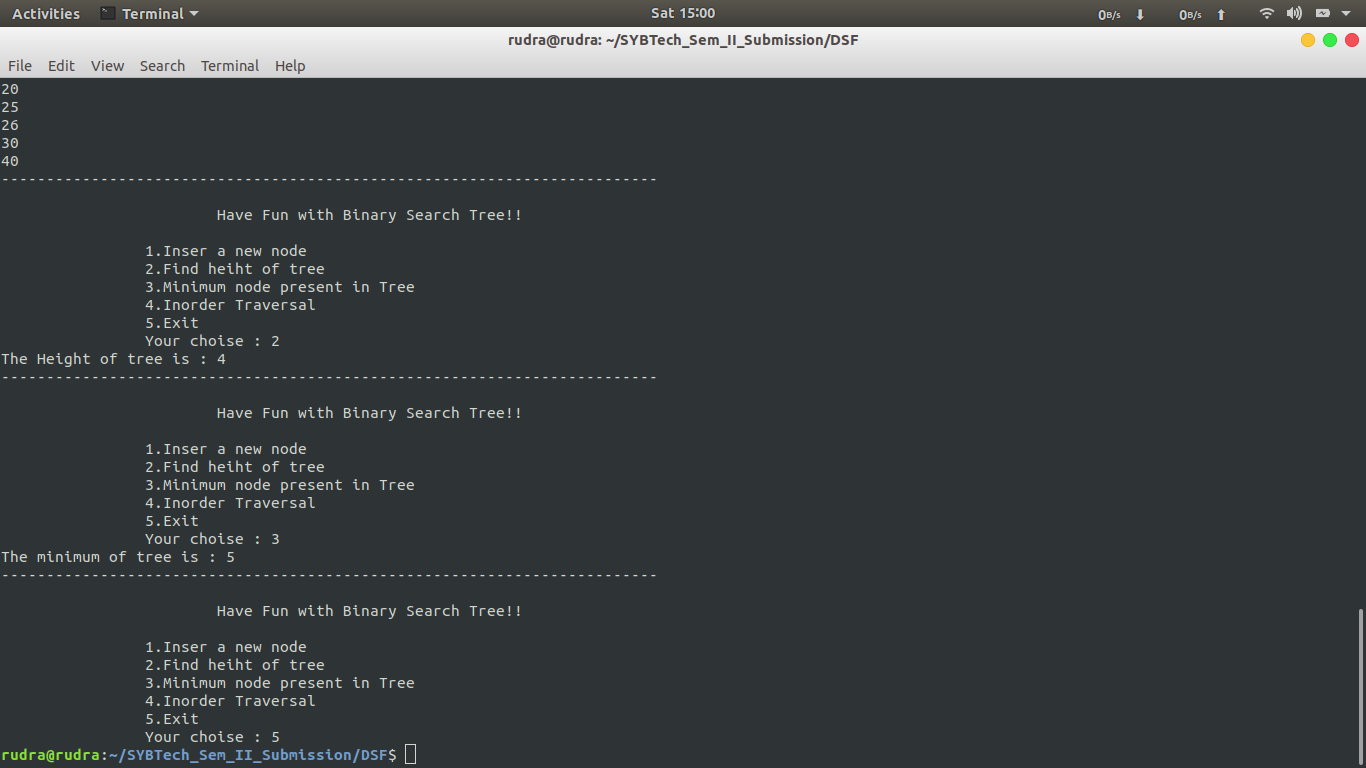
}

**Output:**









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

**Assignment 4**

**Aim:**

Modify the BST such that the roles of the left and right pointers are swapped at every node

**Objective :**

Swapping left and right node of the BST

**Theory:**

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.

Input Tree:

10

/ \

5 8

/ \

2 20

In the above tree, nodes 20 and 8 must be swapped to fix the tree.

Following is the output tree

10

/ \

5 20

/ \

2 8

The inordertraversal of a BST produces a sorted array. So a **simple method** is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiilary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is O(nLogn) and auxiliary space needed is O(n).

**1.** The swapped nodes are not adjacent in the inorder traversal of the BST.

For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.

The inorder traversal of the given tree is 3 25 7 8 10 15 20 5

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 ( current node ). Finally swap the two node’s values.

**Algorithm:**

(1) Call swap for left-subtree i.e., swap(left-subtree)

(2) Call swap for right-subtree i.e., swap(right-subtree)

(3) Swap left and right subtrees.

temp = left-subtree

left-subtree = right-subtree

right-subtree = temp

**Code :**

**Problem Statement : Modify the above BST such that the roles of the left and right pointers are swapped at every node**

#include<iostream>

#include<math.h>

using namespace std;

struct tnode{

int data;

tnode \*lptr,\*rptr;

};

tnode \*root=NULL;

tnode \*insert(int val);

void create(int val);

void inorder(tnode \*);

void preorder(tnode \*);

void postorder(tnode\*);

int minimum(tnode \*);

int height(tnode \* );

int swapTree(tnode \*);

void swap(tnode \*,tnode \*);

int main()

{

int num,a,choise;

cout<<"Nodes in tree : ";

cin>>a;

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

{

cout<<"Enter the data : ";

cin>>num;

create(num);

}

do

{

cout<<"\n-------------------------------------------------------------------------\n";

cout<<"\n\t\t\tPlay With Binary Search Tree\n";

cout<<"\n\t\t1.Inser a new node\n\t\t2.Swipe the tree\n\t\t3.Inorder traversal\n\t\t4.Exit\n\t\tYour choise : ";

cin>>choise;

switch(choise)

{

case 1:

cout<<"Enter the data : ";

cin>>num;

create(num);

break;

case 2:

swapTree(root);

cout<<"Tree Swiped ! ";

break;

case 3:

cout<<"Inorder Traversal of Tree : \n";

inorder(root);

break;

}

}while(choise<4);

}

tnode \*insert(int val)

{

tnode \*newNode =new tnode;

newNode->data=val;

newNode->lptr =newNode->rptr =NULL;

return newNode;

}

void create(int val)

{

tnode \*parent,\*current=root;

if(root==NULL)

{

root = insert(val);

}

else

{

while(current!=NULL)

{

parent =current;

if(val<current->data)

current =current->lptr;

else

current = current->rptr;

}

if(val<parent->data)

parent->lptr=insert(val);

else

parent->rptr =insert(val);

}

}

void inorder(tnode \*Root)

{

tnode \*temp =Root;

if(temp==NULL)

return ;

inorder(temp->lptr);

cout<<temp->data<<endl;

inorder(temp->rptr);

}

int swapTree(tnode \*Root)

{

tnode \*current = Root;

if(current==NULL)

return 0;

tnode \*temp =current->rptr;

current->rptr = current->lptr;

current->lptr = temp;

//swap(current->lptr,current->rptr);

swapTree(current->lptr);

swapTree(current->rptr);

}

void swap(tnode \*r,tnode \*l)

{

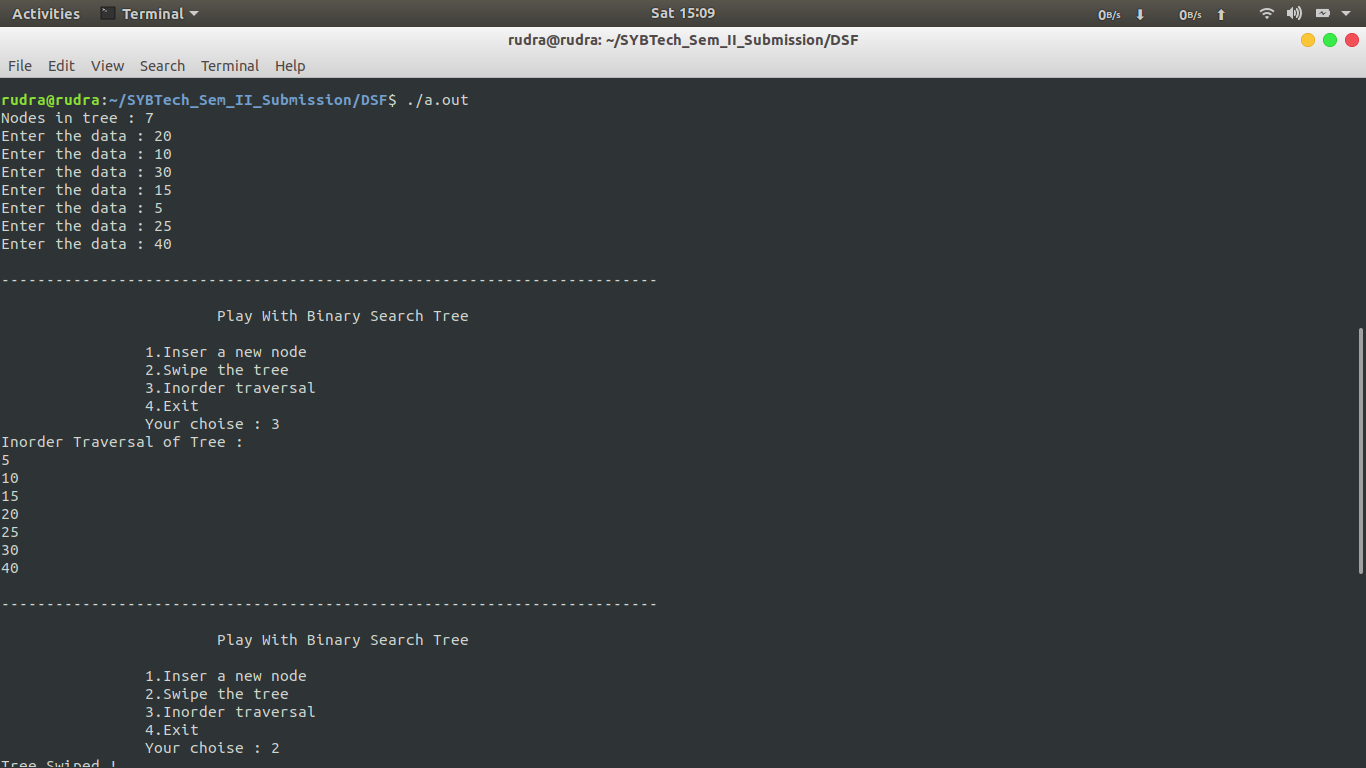
tnode \*temp = r;

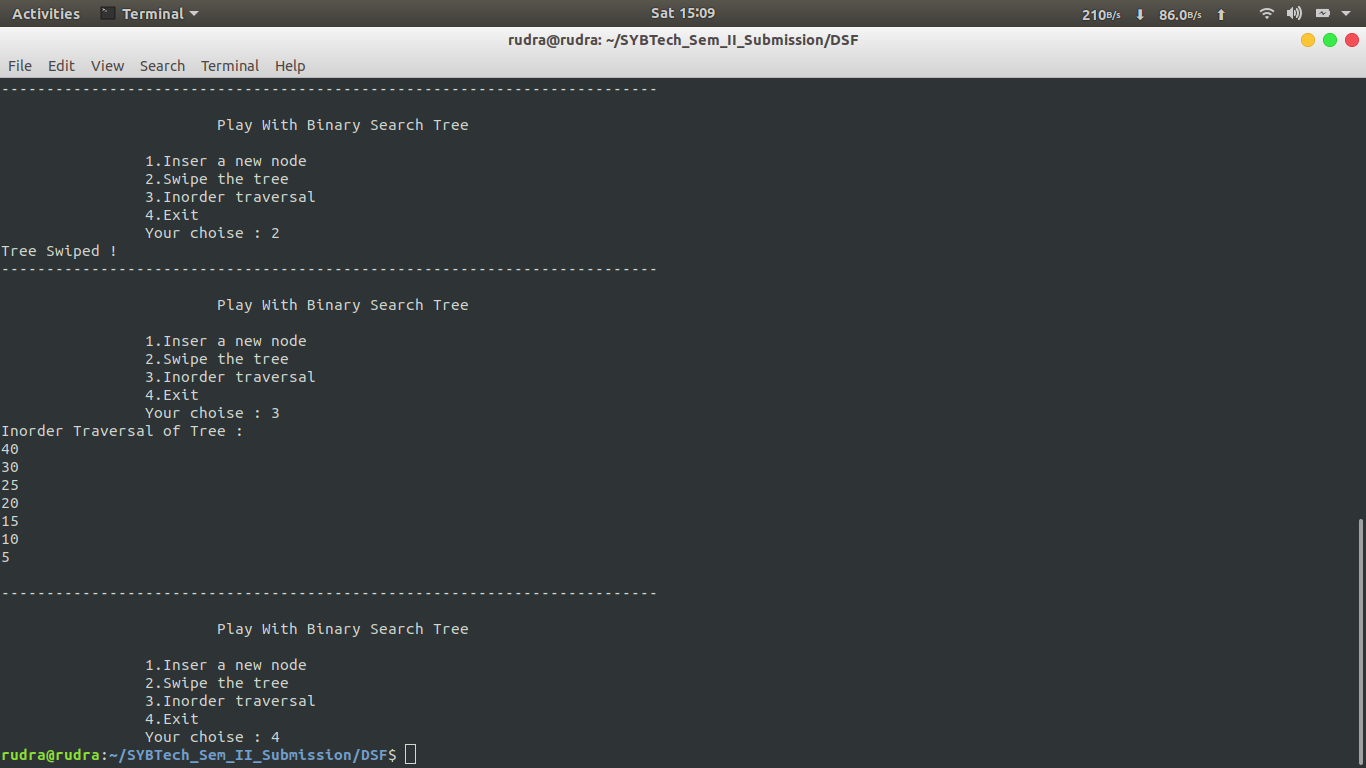
r = l;

l = temp;

}

**Output:**

****

****

**Conclusion:**

This assignment is used how to swap left and right node of the BST

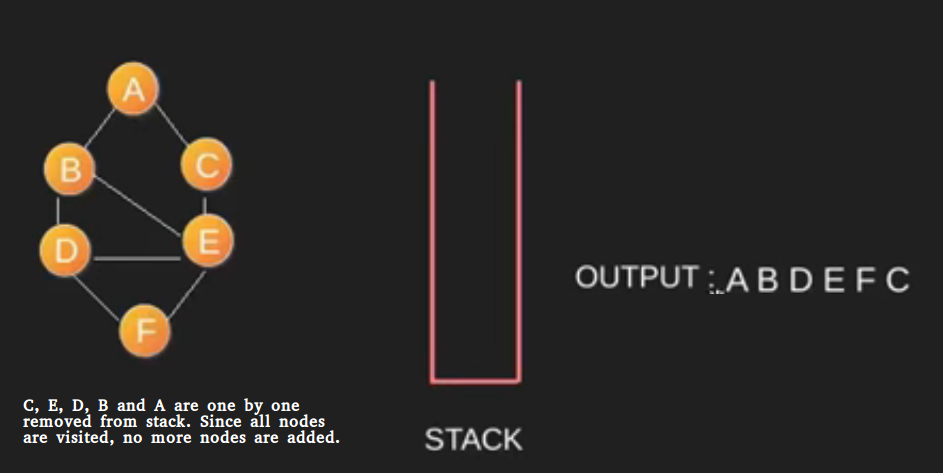
**Assignment 5**

**Aim :**

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

**Theory:**

**Depth-first search** (**DFS**) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It is similar to the depth first traversal of a tree, however, in order to ensure that roots are not repeated, we use a visited array. Depth first search can be performed recursively as well as iteratively. Stack is used as an assistive data structure in depth first search.



An example of depth first search.

**Algorithm**

DFS-iterative (G, s): //Where G is graph and s is source vertex

let S be stack

S.push( s ) //Inserting s in stack

mark s as visited.

while ( S is not empty):

//Pop a vertex from stack to visit next

v = S.top( )

S.pop( )

//Push all the neighbours of v in stack that are not visited

for all neighbours w of v in Graph G:

if w is not visited :

S.push( w )

mark w as visited

DFS-recursive(G, s):

mark s as visited

for all neighbours w of s in Graph G:

if w is not visited:

DFS-recursive(G, w)

**Program :**

**Problem Statement : Represvnt a given graph using adjacency matrix and traversv each node using Depth first svarch**

#include<iostream>

using namespace std;

void dfs\_rec(int v,int visit[],int u);

int adj[20][20];

int main()

{

int v,e,a,b;

cout<<"Enter the number of vertices :";

cin>>v;

// int adj[v][v];

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

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

{

adj[i][j] = 0;

}

cout<<"Enter the number of edges : ";

cin>>e;

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

{

cout<<"Enter the edge (Starting and ending point) : ";

cin>>a>>b;

adj[a][b]= adj[b][a] = 1;

}

int visited[v];

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

visited[i] =0;

cout<<"\t\tThe Depth First Search of your graph is : ";

dfs\_rec(0,visited,v);

cout<<"\n\n";

}

void dfs\_rec(int v,int visit[],int u)

{

cout<<v;

visit[v] = 1;

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

{

if(adj[v][j]==1&&visit[j]==0)

{

dfs\_rec(j,visit,u);

}

}

}

/\*

//IF YOU WANT TO PRINT THE ADJANCENCY MATRIX OF GRAPH

cout<<"\n";

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

{

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

{

cout<<adj[i][j]<<"\t";

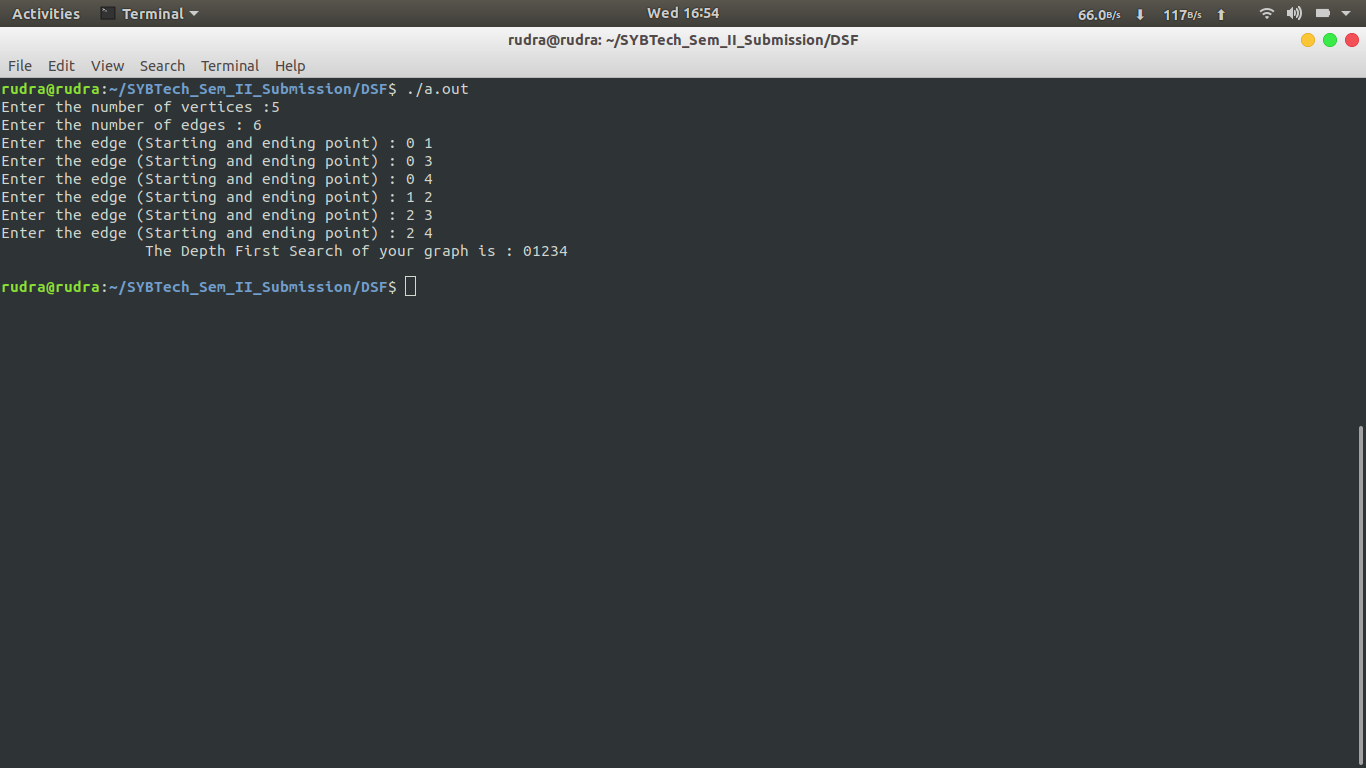
}

cout<<"\n";

}

\*/

**Output.**



**Conclusion :**

Through this assignment we learnt how to implement the Depth First Search

**Assignment 6**

**Aim**:

Represent a given graph using adjacency list and traverse each node using Breadth-firstsearch.

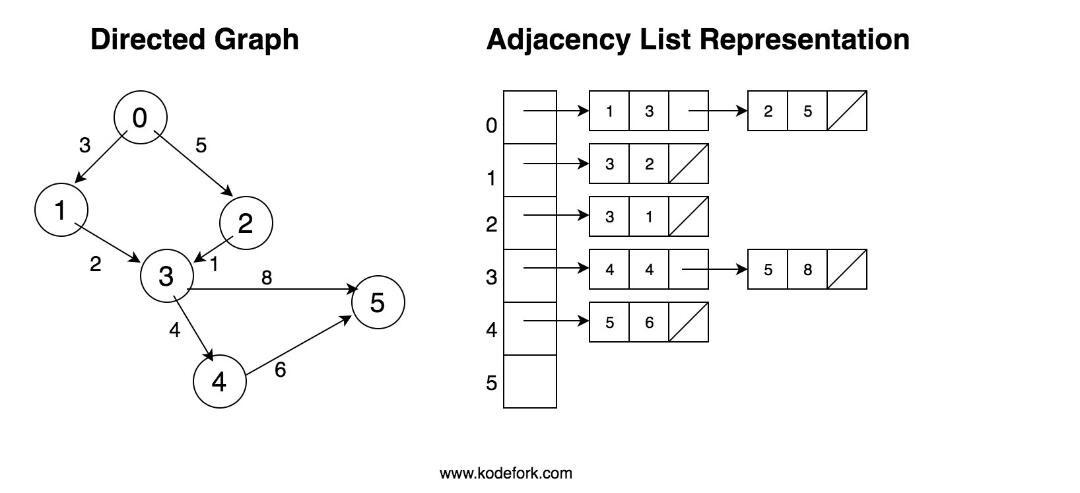
**Objective** :

To learn to implement adjacency-list representation of graph and BFS traversal of nodes.

**Theory**:

**1) Adjacency List:**

1. A graph can be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list. It creates a seperate linked list for each vertex V in the graph G = (V, E)
2. Adjacency list of a graph with n nodes can be represented by an array of pointers. Each pointer points to a linked list of the corresponding vertex.



**2) Breadth-First Traversal:**

* It is a popular approach used for visiting the vertices of a graph.
* This method starts from a given vertex suppose A. A is marked as visited. All vertices adjacent to A are visited next. The method continues until all vertices are visited.
* The algorithm for BFS has to maintain a list of vertices that are visited.

**Algorithm:**

**For Adjacency-list creation:**

int nodes be variable to store number of nodes

node\* heads[100] be array to store head pointer of each linked list.

* Enter number of nodes.
* for i = 0 to i = nodes, initialize heads[i] to pointer pointing to node with data ‘i’
* for i = 0 to i = nodes:

\* Enter number of adjacent nodes.

\* Enter the adjacent nodes and append it to end of linked list heads[i]

**Breadth-First Search Algorithm:**

* Enqueue the starting vertex.
* v = dequeue()
* if v is not visited, print it and mark it as visited (visited[v] = 1)
* Enqueue all adjacent vertices of v that are not visited.
* Repeat steps 2 to 4 till Queue is empty.

**Program:**

Problem Statement : Represent a given graph using adjacency list and traverse each node using Breadth first search

#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";

}

**Conclusion :**

We can use int visited array to maintain list of visited vertices.

BFS can be implemented with the help of a queue.

Adjacency list implementation of graph is memory efficient when graph has a large number of vertices, but small number of edges.

**Assignment-7**

**Problem Statement**:

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

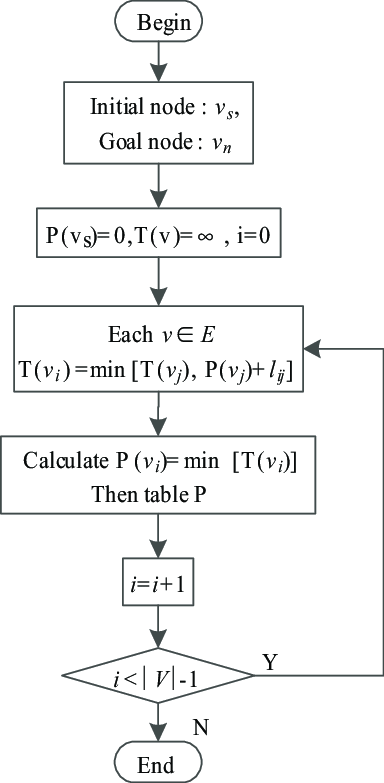
**Aim**: To solve the given problem using Dijkstra’s Algorithm.

**Objective**: To understand the problem statement and use suitable algorithm to solve the problem.To design and implement solution using Dijkstra’s algorithm.

**Theory**: Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

**Algorithm:**

1. Set the distance to the source to 0 and the distance to the remaining vertices to infinity.
2. Set the **current** vertex to the source.
3. Flag the **current** vertex as visited.
4. For all vertices adjacent to the **current** vertex, set the distance from the source to the **adjacent** vertex equal to the minimum of its present distance and the **sum** of the **weight of the edge** from the current vertex to the adjacent vertex and the distance from the source to the **current** vertex.
5. From the set of **unvisited vertices**, arbitrarily set one as the new **current** vertex, provided that there exists an edge to it such that it is the minimum of all edges from a vertex in the set of **visited vertices** to a vertex in the set of **unvisited vertices**. To reiterate: The new current vertex must be unvisited and have a minimum weight edges from a visited vertex to it. This can be done trivially by looping through all visited vertices and all adjacent unvisited vertices to those visited vertices, keeping the vertex with the minimum weight edge connecting it.
6. Repeat steps 3-5 until all vertices are flagged as visited.



**Code:**

**#**include<iostream>

#include<vector>

#include<stdlib.h>

using namespace std;

int min(int dis[],int n,int vis[])

{

int m,i,j;

m=32767;

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

{

if(dis[i]<=m && vis[i]==0)

{

m=dis[i];

j=i;

}

}

return j;

}

int main()

{

vector <int>v;

vector<int>::iterator it;

int n,i,j,u;

char ch;

cout<<"Enter the number of vertices\n";

cin>>n;

int arr[n][n],dest,source=0;

cout<<"Enter the destination\n";

cin>>dest;

do

{

cout<<"Source:"<<source<<"\n";

v.clear();

cout<<"Enter the adjecancy matrix w.r.t time\n";

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

{

for(j=0;j<n;j++)

{

cin>>arr[i][j];

}

}

cout<<"Entered matrix is:\n";

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

{

for(j=0;j<n;j++)

{

cout<<""<<arr[i][j]<<"\t";

}

cout<<"\n";

}

int count=1;

int vis[n],dis[n],parent[n];

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

{

vis[i]=0;

dis[i]=32767;

parent[i]=0;

}

dis[source]=0;

parent[source]=-1;

cout<<"Parent of source:"<<parent[source]<<"\n";

cout<<"\n";

while(count<=n)

{

u=min(dis,n,vis);

vis[u]=1;

for(j=0;j<n;j++)

{

if(arr[u][j]!=0 && dis[j]> dis[u]+arr[u][j])

{

dis[j]=dis[u]+arr[u][j];

parent[j]=u;

}

}

/\*for(i=0;i<n;i++)

{

cout<<""<<dis[i]<<"\t";

}

cout<<"\n";\*/

//cout<<"u:"<<u<<"\n";

if(u==dest)

{

while(parent[u]!=-1)

{

v.push\_back(u);

u=parent[u];

}

v.push\_back(source);

it=v.end();

it--;

while(it!=(v.begin()--))

{

cout<<""<<\*it<<" -\t";

it--;

}

cout<<""<<\*it<<"\n";

}

count++;

}

//cout<<"After using Dijkstra's algorithm:\n";

if(u==dest)

{

cout<<""<<u<<" : "<<dis[u]<<"\n";

}

it=v.end();

it--;

while(it!=(v.begin()-1))

{

cout<<"Has the traffic conditions changed?\n";

cin>>ch;

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

{

/\*if(source==0)

{

it=v.end();

it--;

}\*/

source=\*it;

//cout<<"Source:"<<source<<"\n";

break;

}

else

{

if(\*it==dest)

{

cout<<"You have reached your destintion\n";

exit(1);

}

it--;

cout<<""<<\*it<<"\n";

continue;

}

}

}while(1);

return 0;

}

/\*

0 4 0 0 0 0 0 8 0

4 0 8 0 0 0 0 11 0

0 8 0 7 0 4 0 0 2

0 0 7 0 9 14 0 0 0

0 0 0 9 0 10 0 0 0

0 0 4 14 10 0 2 0 0

0 0 0 0 0 2 0 1 6

8 11 0 0 0 0 1 0 7

0 0 2 0 0 0 6 7 0

Changed traffic conditions:

0 4 0 0 0 0 0 8 0

4 0 8 0 0 0 0 11 0

0 8 0 7 0 4 0 0 2

0 0 7 0 9 1 0 0 0

0 0 0 9 0 11 0 0 0

0 0 4 1 11 0 2 0 0

0 0 0 0 0 2 0 1 6

8 11 0 0 0 0 1 0 7

0 0 2 0 0 0 6 7 0

\*/

/\*

OUTPUT:

Enter the number of vertices

9

Enter the destination

4

Source:0

Enter the adjecancy matrix w.r.t time

0 4 0 0 0 0 0 8 0

4 0 8 0 0 0 0 11 0

0 8 0 7 0 4 0 0 2

0 0 7 0 9 14 0 0 0

0 0 0 9 0 10 0 0 0

0 0 4 14 10 0 2 0 0

0 0 0 0 0 2 0 1 6

8 11 0 0 0 0 1 0 7

0 0 2 0 0 0 6 7 0

Entered matrix is:

0 4 0 0 0 0 0 8 0

4 0 8 0 0 0 0 11 0

0 8 0 7 0 4 0 0 2

0 0 7 0 9 14 0 0 0

0 0 0 9 0 10 0 0 0

0 0 4 14 10 0 2 0 0

0 0 0 0 0 2 0 1 6

8 11 0 0 0 0 1 0 7

0 0 2 0 0 0 6 7 0

Parent of source:-1

0 - 7 - 6 - 5 - 4

Has the traffic conditions changed?

n

7

Has the traffic conditions changed?

n

6

Has the traffic conditions changed?

n

5

Has the traffic conditions changed?

Y

Source:5

Enter the adjecancy matrix w.r.t time

0 4 0 0 0 0 0 8 0

4 0 8 0 0 0 0 11 0

0 8 0 7 0 4 0 0 2

0 0 7 0 9 1 0 0 0

0 0 0 9 0 11 0 0 0

0 0 4 1 11 0 2 0 0

0 0 0 0 0 2 0 1 6

8 11 0 0 0 0 1 0 7

0 0 2 0 0 0 6 7 0

Entered matrix is:

0 4 0 0 0 0 0 8 0

4 0 8 0 0 0 0 11 0

0 8 0 7 0 4 0 0 2

0 0 7 0 9 1 0 0 0

0 0 0 9 0 11 0 0 0

0 0 4 1 11 0 2 0 0

0 0 0 0 0 2 0 1 6

8 11 0 0 0 0 1 0 7

0 0 2 0 0 0 6 7 0

Parent of source:-1

5 - 3 - 4

Has the traffic conditions changed?

n

3

Has the traffic conditions changed?

4

Has the traffic conditions changed?

n

You have reached your destintion

\*/

**Conclusion:** In this assignment we learnt the implementation of dijkatra’s algorithm.

**Assignment 8**

**Aim:**

A node consists of <Key,Value> pair where nodes are compared and inserted on the 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 probing without chaining)

**Objective:**

Understand the concept of Linear probing without chaining

**Theory:**

**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

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

..................................................

..................................................

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



**Disadvantage of Linear Probing :**

**Clustering:** The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

**Algorithm :**

1. Take the size of the hash table from the user and store it in size variable
2. Create a structure to store key and value and make array of objects
3. Initialize the hash keys to -1
4. For Insert

Start Loop from I=0 and insert at h[(key+i)%size] if this value is -1 or -2(if value deleted)

1. For Search

Take the key to be searched

Start a loop from I=0 and check h[(key+i)%2] is the key searched

If yes

break the loop and return location

Else

Continue

Return(-1) if not found

1. For Delete

Take the key to be deleted and Search the location and update its value to null and set its key to -2

**Code:**

#include<iostream>

#include<string.h>

using namespace std;

struct product

{

int serialNo;

char name[20];

};

void insertKey(product ht[],int size );

void searchValue(product ht[],int size);

void updateValue(product ht[],int size);

void deleteKey(product ht[],int size);

void display(product ht[],int size);

int hashKey(int,int);

int main()

{

int choise;

int size;

cout<<"Enter the size of hash table : ";

cin>>size;

product hashTable[size];

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

{

hashTable[i].serialNo = -1;

strcpy(hashTable[i].name,"");

}

do

{

cout<<"\n------------------------------------------------------------\n";

cout<<"\t\t\t Linear Probing Without Chaining \n\n";

cout<<"\t\t1.Insert a new Product \n";

cout<<"\t\t2.Search for Product details with serial number \n";

cout<<"\t\t3.Update the Product data \n";

cout<<"\t\t4.Delete the Product data \n";

cout<<"\t\t5.Display the hashing table data \n";

cout<<"\t\t6.Exit\n";

cout<<"\tYour choise : ";

cin>>choise;

switch(choise)

{

case 1:

insertKey(hashTable,size);

break;

case 2:

searchValue(hashTable,size);

break;

case 3:

updateValue(hashTable,size);

break;

case 4:

deleteKey(hashTable,size);

break;

case 5:

display(hashTable,size);

break;

}

}while(choise<6);

}

void insertKey(product ht[],int size)

{

int key;char name[20];

cout<<"Enter the Serial Number of product : ";

cin>>key;

cout<<"Enter the Name of Product : ";

cin>>name;

int hash = hashKey(key,size);

if(ht[hash].serialNo==(-1))

{

ht[hash].serialNo = key;

strcpy(ht[hash].name,name);

cout<<"\t\tProduct inserted Succesfully in table \n";

}

else

{

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

{

if(ht[(hash+i)%size].serialNo == (-1))

{

ht[(hash+i)%size].serialNo = key;

strcpy(ht[(hash+i)%size].name,name);

cout<<"\t\tProduct inserted Succesfully in table \n";

break;

}

}

}

}

void display(product ht[],int size)

{

cout<<"\n\t\t\tThe data present in hash table is as follows \n\n";

cout<<"\t\tSerial Number\tProduct Name\n";

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

{

cout<<"\t\t "<<ht[i].serialNo<<"\t\t\t"<<ht[i].name<<endl;

}

}

void searchValue(product ht[],int size)

{

int key,flag=0;

cout<<"Enter the Serial Number to be search : ";

cin>>key;

int hash = hashKey(key,size);

if(ht[hash].serialNo==key)

{

cout<<"Key found !\nThe details stored at key "

<<key<<" is as : Serial Number : "

<<ht[hash].serialNo<<"\t Product name : "<<ht[hash].name<<endl;

flag =1;

}

else{

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

{

if(ht[(hash+i)%size].serialNo == key)

{

cout<<"Key found !\nThe details stored at key "

<<key<<" is as : Serial Number : "

<<ht[(hash+i)%size].serialNo<<"\t Product name : "<<ht[(hash+i)%size].name<<endl;

flag =1;

break;

}

}

}

if(flag==0)

cout<<"\tOops !! Looks like the key is not in table\n";

}

void updateValue(product ht[],int size)

{

int key,flag =0;

char uName[20];

cout<<"Enter the Serial Number to be Update : ";

cin>>key;

int hash = hashKey(key,size);

if(ht[hash].serialNo ==key)

{

cout<<"Enter the updated name of product : ";

cin>>uName;

strcpy(ht[hash].name,uName);

cout<<"\tSuccess ! Data updated at Serial Number : "

<<ht[hash].serialNo<<endl;

flag =1;

}

else

{

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

{

if(ht[(hash+i)%size].serialNo == key)

{

cout<<"Enter the updated name of product : ";

cin>>uName;

strcpy(ht[(hash+i)%10].name,uName);

cout<<"\tSuccess ! Data updated at Serial Number : "

<<ht[(hash+i)%10].serialNo<<endl;

flag =1;

break;

}

}

}

if(flag==0)

cout<<"\tLooks like the key is not in table\n";

}

void deleteKey(product ht[],int size)

{

int key,flag =0;

char uName[20];

cout<<"Enter the Serial Number to be Delete : ";

cin>>key;

int hash = hashKey(key,size);

if(ht[hash].serialNo ==key)

{

ht[hash].serialNo = -1;

strcpy(ht[hash].name,"");

cout<<"\tSuccess ! Data deleted at Serial Number : "

<<key<<endl;

flag =1;

}

else

{

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

{

if(ht[(hash+i)%size].serialNo == key)

{

ht[(hash+i)%size].serialNo = -1;

strcpy(ht[(hash+i)%size].name,"");

cout<<"\tSuccess ! Data deleted at Serial Number : "

<<key<<endl;

flag =1;

break;

}

}

}

if(flag==0)

cout<<"\tLooks like the key is not in table\n";

}

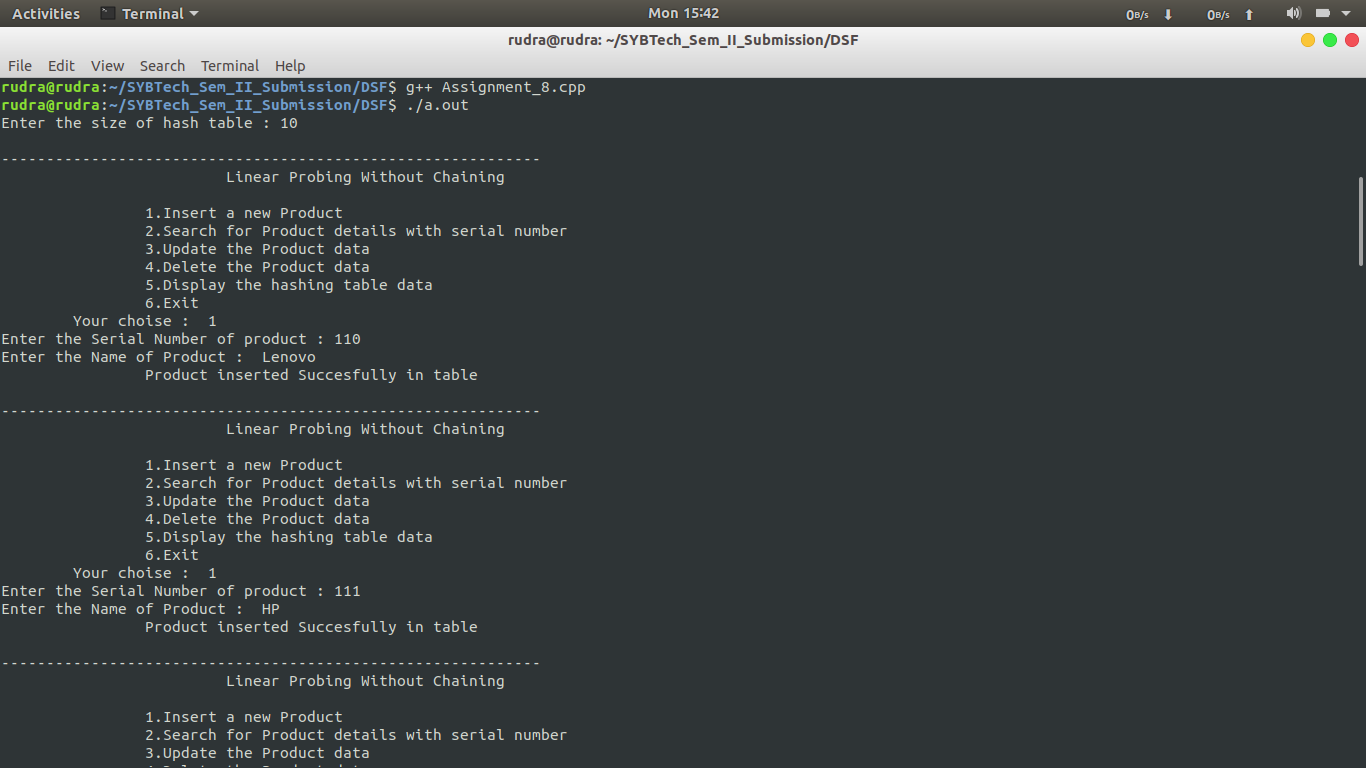
int hashKey(int key,int size)

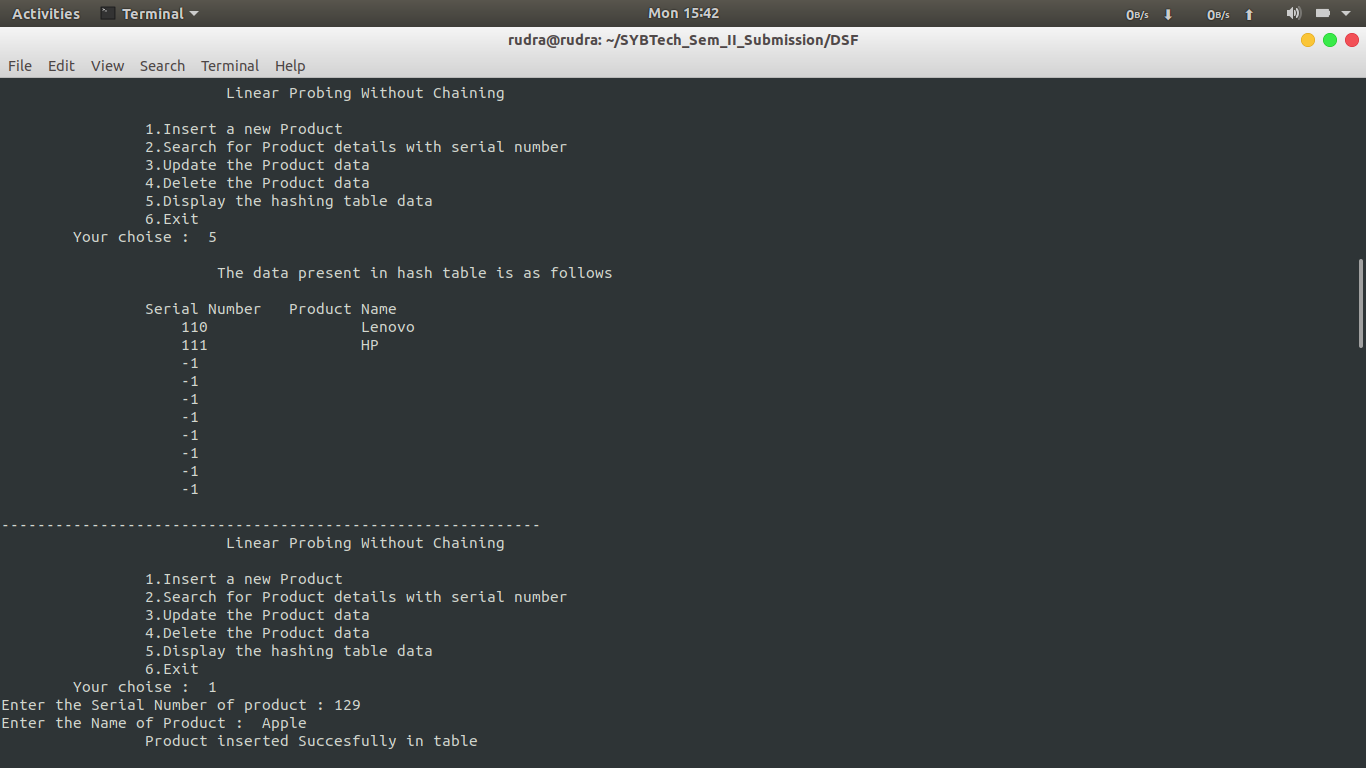
{

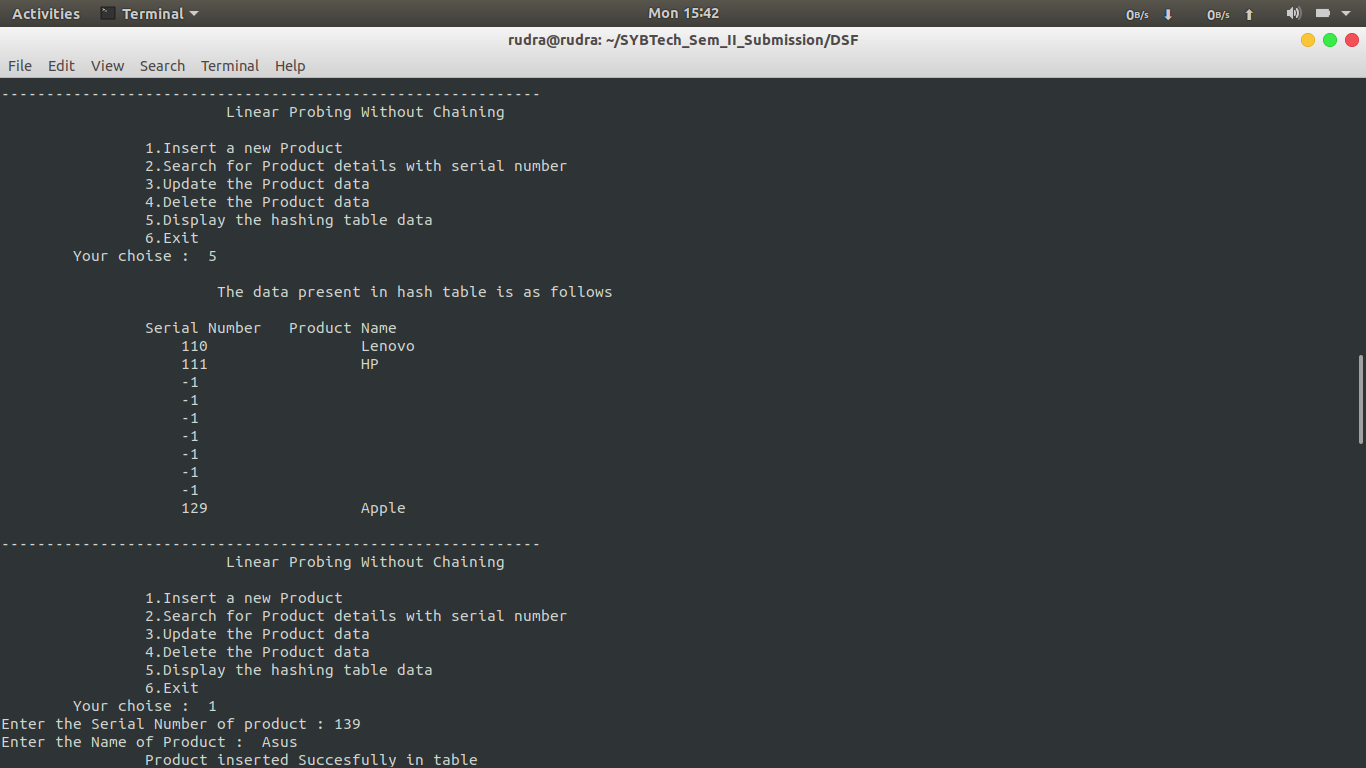
return(key%size);

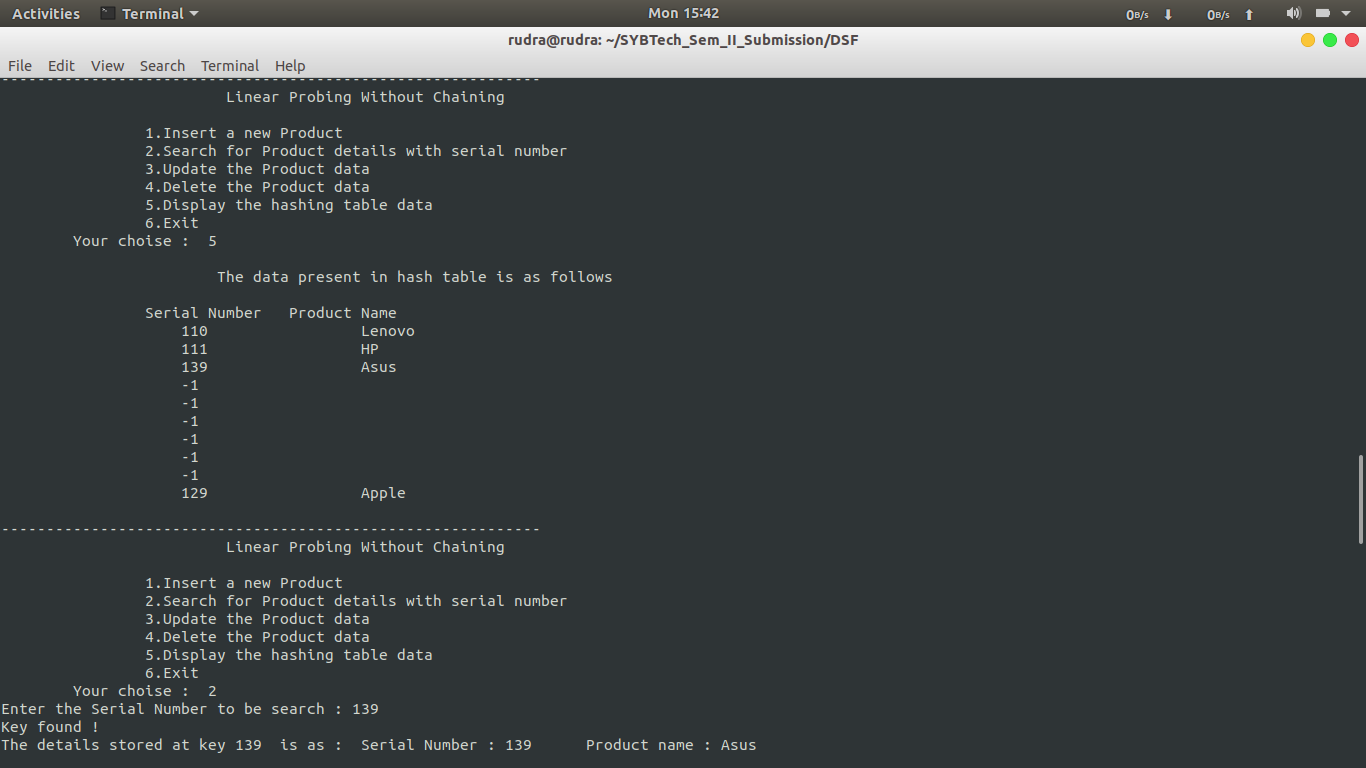
}

**Output:**

****

****

****

****

**Conclusion :**

Through this assignment we understand the concept of linear probing in hashing to avoid collisions .

**Assignment-9**

* **Problem Statement:**

Implement mall the functions of a dictionary (ADT) using hashing.

Data: Set of (key, value) pairs, keys are mapped to values, keys must be comparable and unique.

Standard operations:

* + Insert(key, value)
  + Find(key)
  + Delete(key)

# **Objective:**

Able to understand and use:

* + Basic concepts of hash tables and hash functions.
  + Basic collision resolution techniques in hash tables.
  + Implementation of various forms of hash tables i.e. separate chaining and linear probing.

# **Theory:**

## **Dictionary ADT:**

Dictionary (map, association list) is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key (and naturally an associated value) and lookup for a value by the key. Values are not required to be unique. Simple usage example is an explanatory dictionary. In the example, words are keys and explanations are values.

Methods that a dictionary can have:

* + Dictionary create()
  + boolean isEmpty(Dictionary d)
  + void put(Dictionary d, Key k, Value v)
  + void get(Dictionary d, Key k)
  + Key remove(Dictionary d,Key k)
  + void destroy(Dictionary d)

## **Hash tables:**

A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list *Θ*(*n*) time in the worst case in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is *O*(1).

## **Hash functions:**

Hash function is very important part of hash table design. Hash function is considered to be good, if it provides uniform distribution of hash values. Other hash function’s properties, required for quality hashing will be examined in detail later. The reason, why hash function is a subject to the principal concern, is that poor hash functions cause collisions and some other unwanted effects, which badly affect hash table overall performance.

## **Hash table and load factor:**

Basic underlying data structure used to store hash table is an array. The load factor is the ratio between the number of stored items and array’s size. Hash table can whether be of a constant size or being dynamically resized, when load factor exceeds some thresh- old. Resizing is done before the table becomes full to keep the number of collisions under certain amount and prevent performance degradation.

## **Collisions and Resolutions:**

What happens, if hash function returns the same hash value for different keys? It yields an effect, called collision. Collisions are practically unavoidable and should be considered when one implements hash table. Due to collisions, keys are also stored in the table, so one can distinguish between key-value pairs having the same hash. There are various ways of *collision resolution* . Basically, there are two different strategies:

* + **Closed addressing (open hashing):**

Each slot of the hash table contains a link to another data structure (i.e. linked list), which stores key-value pairs with the same hash. When collision occurs, this data structure is searched for key-value pair, which matches the key.

* + **Open Addressing (closed hashing):**

Each slot actually contains a key-value pair. When collision occurs, open addressing algorithm calculates another location (i.e. next one) to locate a free slot. Hash tables, based on open addressing strategy experience drastic performance decrease, when table is tightly filled (load factor is 0.7 or more).

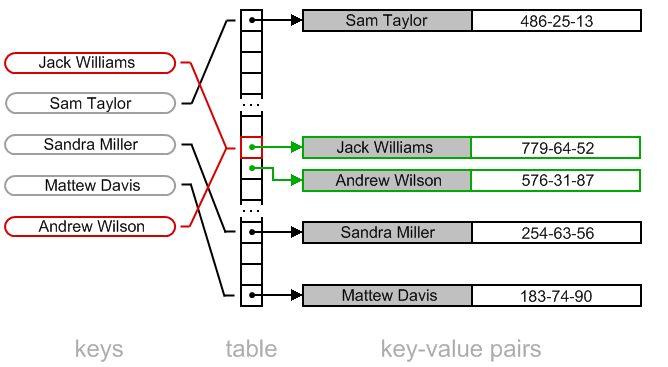
* + 

fig: Linear probing (Open addressing)

# **Algorithms:**

1. void input()
   1. Start
   2. while char c == ’y’:
      1. Create new node
      2. Take input the word and it’s meaning
      3. call hash(word) and hash at index recieved
      4. Continue? (y/n)
   3. Stop
2. int hash(string word)
   1. Start
   2. while i is less than length of word:
   3. Add the ASCII values of characters of word
   4. mod the sum by 10 to find the hash index
   5. return hash index
3. void deletefunc()
   1. Start
   2. Take input the word and calculate it’s index by hash(word)
   3. starting from the index found, find the word to be deleted
   4. if found, delete the node and return
   5. else, print “Element not found” and return (f) Stop
4. void search()
   1. Start
   2. Input the word
   3. find the index of the word by hash(word)
   4. Starting from the index, look for the given word
   5. if found, print it’s meaning (f) else, “print word not found”

(g) Stop

1. void display()
   1. Start
   2. while i is less than size of hash table:
      1. if node is empty, skip node
      2. else, print it’s word and meaning
   3. Stop

# **Code:**

# 

# #include<iostream>

# #include<string>

# using namespace std;

# class hashing

# {

# typedef struct node

# {

# string word;

# string meaning;

# struct node\* next;

# }node;

# node \*New,\*temp;

# int i;

# node \*arr[10];

# public:

# hashing()

# {

# i=0;

# New=temp=NULL;

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

# {

# arr[i]=NULL;

# }

# }

# void input()

# {

# 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 deletefunc()

# {

# string delword;

# int flag=0;

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

# cin>>delword;

# int x=hash(delword);

# 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 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 search()

# {

# string person;

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

# cin>>person;

# int x=hash(person);

# 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 display()

# {

# int j=0;

# while(j<10)

# {

# 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();

# return 0;

# }

# **6 Input And Output:**

Enter word :a

Enter meaning :apple

Continue?y

Enter word :b

Enter meaning :ball

Continue?y

Enter word :c

Enter meaning :cat

Continue?n

a apple

b ball

c cat

Enter the word to be deleted :a

b ball

c cat

Enter the word to find it's meaning :c

Required meaning is :cat

**7 Conclusion:**

By performing the above assignment, we learned the concepts of implementation of hash tables, the collisions and used a collision resolution method called seperate chaining.

**8 Applications:**

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, [distinct elements](https://www.geeksforgeeks.org/print-distinct-elements-given-integer-array/), counting frequencies of items, finding duplicates, etc.

There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

* Message Digest
* Password Verification
* Data Structures(Programming Languages)
* Compiler Operation
* Rabin-Karp Algortithm
* Linking File name and path together

**Assignment 10**

**Aim**:

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

**Objective**:

To learn to implement Height Balance Tree.

**Theory**:

AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node can’t be more than 1. Here are some key points about [AVL trees](https://www.geeksforgeeks.org/tag/avl-tree/):

1. If there are n nodes in AVL tree, minimum height of AVL tree is floor(log2n).
2. If there are n nodes in AVL tree, maximum height can’t exceed 1.44\*log2n.
3. If height of AVL tree is h, maximum number of nodes can be 2h+1 – 1.
4. Minimum number of nodes in a tree with height h can be represented as:  
   N(h) = N(h-1) + N(h-2) + 1 for n>2 where N(0) = 1 and N(1) = 2.
5. The complexity of searching, inserting and deletion in AVL tree is O(log n).

**Algorithm:**

Insert and rebalance:

Avlnode \*y=new avlnode

If(k<pp->key) pp->left=y

Else pp->right=y

//adjusting balancing factor from a to pp

int d;

Avlnode\*b,\*c;

If(k>a->key){b=p=a->right;d=-1}

Else{b=p=a->left;d=1}

While(p!=y)

{if(k>p->key){p->bf=-1;p=p->right}

Else

{p->bf=1;p=p->left}

Rotations:

* LL Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

a->left = b->right ;

b->right = a ;

a->bf = b->bf = 0;

* RR Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

// c will point to node which is going to replace unbalanced node

a->right = b -> left ; b -> left = a ;

a->bf = b -> bf = 0

* LR Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

// c will point to node which is going to replace unbalanced node

c = b -> right ; //LR rotation

b-> right = c ->left ;

a->left = c ->right ;

c -> left = b ;

c-> right = a;

switch ( c -> bf )

{ case 1 : a->bf = -1 ; b-> bf = 0 ; break ;

case -1 : b->bf = 1 ; a ->bf = 0 ; break ;

case 0 : a->bf = b->bf = 0 ; break;

}

c->bf = 0 ;

b = c ;

* RL Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

// c will point to node which is going to replace unbalanced node

c = b -> left ; //RL rotation

b ->left = c ->right ;

a ->right = c->left ;

c->right = b ;

c->left = a ;

switch (c->bf )

{

case 1: a->bf =0 ; b->bf = -1 ; break ;

case -1 : a->bf = +1 ; b->bf = 0 ; break ;

case 0 : a->bf =b->bf = 0 ;break ;

}

c->bf = 0

b= c ;

**Program:**

#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;

}

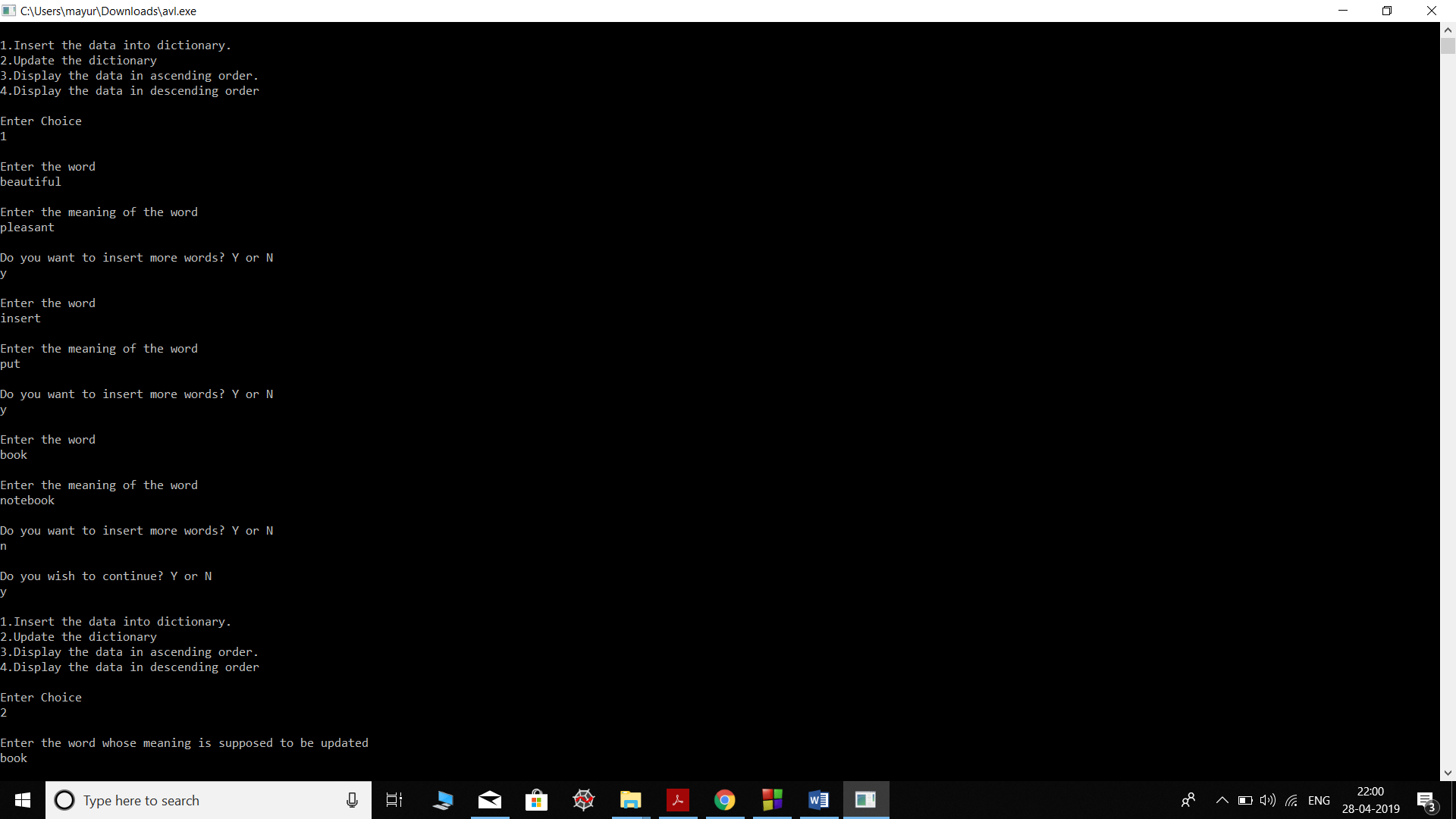
cout<<"\nDo you wish to continue? Y or N \n";

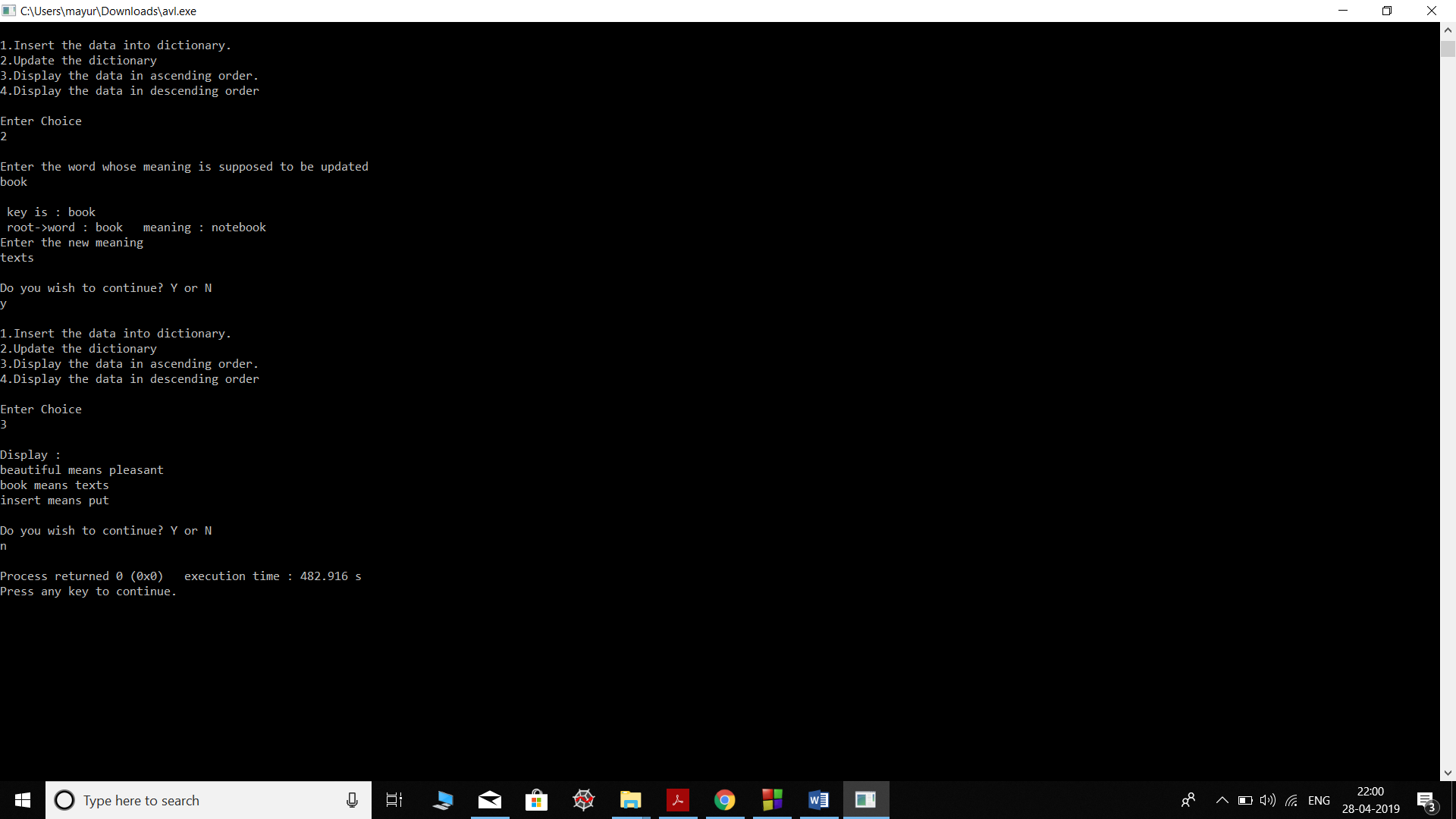
cin>>x;

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

return 0;

}

**Output:**



**Conclusion:**

Thus we have implemented a dictionaries and its functions using a height balanced trees.