BINARY SEARCH TREE

Experiment No.:4

18/08/2012

# AIM:

# Implementation of a Binary Search Tree

# ALGORITHM:

All the algorithms here use a binary search tree with nodes with structure as following:

|  |  |  |
| --- | --- | --- |
| Left | Data | Right |

The members of the structure node are:

Left: Holds the link to the left node

Data: Holds the data for the current node.

Right: Holds the link to the right node.

The tree contains a root node

The function ins() inserts a node in a BST.

Arguments passed:

tree: A BST .

data: The data to be inserted in tree.

Return value: none.

ins(tree, data)

{

temp=:root;

while(temp≠NULL)

{

if(data < temp⇾data)

temp=temp⇾left

else if(data > temp⇾data)

temp=temp ⇾right

}

temp=createNode(data)

}

The function del() deletes a node in a BST.

Arguments passed:

tree: A BST.

data: The data to be deleted from the tree.

Return value: none.

Del(tree, data)

{

temp=root

while(temp≠NULL)

{

if(data < temp⇾data)

temp=temp⇾left

else if(data > temp⇾data)

temp=temp⇾right

else {

oldnode=temp

lc=temp⇾left

temp=temp⇾right

while(temp≠NULL)

temp=temp⇾left

temp=lc

freeNode(oldnode)

break

}

}

}

The following function are used to traverse a BST in in-order, pre-order and post-order.

Inorder(tree)

{

If(tree≠NULL)

{

Inorder(tree⇾left)

Print tree⇾data

Inorder(tree⇾right)

}

}

Preorder(tree)

{

If(tree≠NULL)

{

Print tree⇾data

Preorder (tree⇾left)

Preorder (tree⇾right)

}

}

Postorder (tree)

{

If(tree≠NULL)

{

Postorder (tree⇾right)

Postorder(tree⇾left)

Print tree⇾data

}

}

# SOURCE CODE:

# #include <iostream>

# #include <cstdlib>

# #include <cstdio>

# #include "../myexception.h"

# using namespace std;

# using namespace exception;

# template <class DT> class Queue {

# DT \*buf;

# int rear;

# int length;

# public:

# Queue(const Queue<DT>&);

# Queue(int);

# ~Queue();

# bool isFull();

# bool isEmpty();

# void enqueue(DT);

# DT dequeue();

# int getLength();

# int getNumberOfElements();

# void display();

# void clear();

# };

# template <class DT> Queue<DT>::Queue(const Queue<DT>& q) {

# Queue::length=q.length;

# Queue::front=q.front;

# Queue::rear=q.rear;

# Queue::buf=new DT[q.length];

# for(int i=0;i<q.length;i++)

# Queue::buf[i]=q.buf[i];

# }

# template <class DT> Queue<DT>::Queue(int length=0) {

# if(length>=0)

# {

# Queue::length=length;

# Queue::buf=new DT[length];

# }

# else

# {

# Queue::length=0;

# Queue::buf=new DT[0];

# }

# Queue::rear=-1;

# }

# template <class DT> Queue<DT>::~Queue() {

# delete Queue::buf;

# }

# template <class DT> bool Queue<DT>::isFull() {

# return Queue::rear==Queue::length-1;

# }

# template <class DT> bool Queue<DT>::isEmpty() {

# return Queue::rear==-1;

# }

# template <class DT> void Queue<DT>::enqueue(DT e) {

# if(!Queue::isFull()){

# Queue::rear++;

# Queue::buf[Queue::rear]=e;

# }

# else

# throw QueueFullException();

# }

# template <class DT> DT Queue<DT>::dequeue() {

# if(!Queue::isEmpty())

# {

# DT e=Queue::buf[0];

# for(int i=0; i<Queue::rear; i++)

# Queue::buf[i]=Queue::buf[i+1];

# Queue::rear--;

# return e;

# }

# else

# throw QueueEmptyException();

# }

# template <class DT> int Queue<DT>::getLength() {

# return Queue::length;

# }

# template <class DT> int Queue<DT>::getNumberOfElements() {

# return Queue::rear+1;

# }

# template <class DT> void Queue<DT>::clear() {

# Queue::rear=-1;

# }

# template <class DT> class Node {

# public:

# DT data;

# Node<DT> \*left, \*right;

# Node(DT, Node<DT>\*, Node<DT>\*);

# };

# template <class DT> Node<DT>::Node(DT data, Node<DT> \*left=NULL, Node<DT> \*right=NULL) {

# Node::data=data;

# Node::left=left;

# Node::right=right;

# }

# template <class DT> void traverseInorder(Node<DT> \*\*t) {

# if(\*t!=NULL) {

# traverseInorder(&(\*t)->left);

# cout<<(\*t)->data<<" ";

# traverseInorder(&(\*t)->right);

# }

# }

# template <class DT> void traversePostorder(Node<DT> \*\*t) {

# if(\*t!=NULL) {

# traversePostorder(&(\*t)->left);

# traversePostorder(&(\*t)->right);

# cout<<(\*t)->data<<" ";

# }

# }

# template <class DT> void traversePreorder(Node<DT> \*\*t) {

# if(\*t!=NULL) {

# cout<<(\*t)->data<<" ";

# traversePreorder(&(\*t)->left);

# traversePreorder(&(\*t)->right);

# }

# }

# template <class DT> void clear(Node<DT> \*\*t) {

# if(\*t!=NULL) {

# clear(&(\*t)->left);

# clear(&(\*t)->right);

# delete \*t;

# }

# }

# template <class DT> void copyTree(Node<DT> \*\*t2, Node<DT> \*\*t1) {

# if(\*t1!=NULL) {

# \*t2=new Node<DT>((\*t1)->data);

# copyTree(&(\*t2)->left, &(\*t2)->left);

# copyTree(&(\*t2)->right, &(\*t2)->right);

# }

# }

# template <class DT> class BST {

# int numberOfNodes;

# Node<DT> \*\*root;

# public:

# BST();

# BST(const BST&);

# ~BST();

# void insert(DT);

# void remove(DT);

# void traverseInorder();

# void traversePreorder();

# void traversePostorder();

# void display();

# bool isEmpty();

# int height();

# };

# template <class DT> bool BST<DT>::isEmpty() {

# return numberOfNodes==0;

# }

# template <class DT> BST<DT>::BST() {

# root=new Node<DT>\*;

# \*root=NULL;

# numberOfNodes=0;

# }

# template <class DT> BST<DT>::BST(const BST &b) {

# BST::root=new Node<DT>\*;

# copyTree(BST::root, b.root);

# numberOfNodes=b.numberOfNodes;

# }

# template <class DT> BST<DT>::~BST() {

# clear(BST::root);

# delete root;

# }

# template <class DT> void BST<DT>::insert(DT data) {

# Node<DT> \*\*temp=BST::root;

# while(\*temp!=NULL) {

# if(data < (\*temp)->data)

# temp=&(\*temp)->left;

# else if(data > (\*temp)->data)

# temp=&(\*temp)->right;

# else

# throw NodeAlreadyExistsException();

# }

# \*temp=new Node<DT>(data);

# numberOfNodes++;

# }

# template <class DT> void BST<DT>::remove(DT data) {

# Node<DT> \*\*temp=BST::root;

# while(\*temp!=NULL) {

# if(data < (\*temp)->data)

# temp=&(\*temp)->left;

# else if(data > (\*temp)->data)

# temp=&(\*temp)->right;

# else {

# Node<DT> \*oldnode=\*temp;

# Node<DT> \*left=(\*temp)->left;

# \*temp=(\*temp)->right;

# while(\*temp!=NULL)

# temp=&(\*temp)->left;

# \*temp=left;

# delete oldnode;

# numberOfNodes--;

# return;

# }

# }

# if(\*temp==NULL)

# throw NodeNotFoundException();

# }

# template <class DT> void BST<DT>::traverseInorder() {

# ::traverseInorder(BST::root);

# cout<<endl;

# }

# template <class DT> void BST<DT>::traversePostorder() {

# ::traversePostorder(BST::root);

# cout<<endl;

# }

# template <class DT> void BST<DT>::traversePreorder() {

# ::traversePreorder(BST::root);

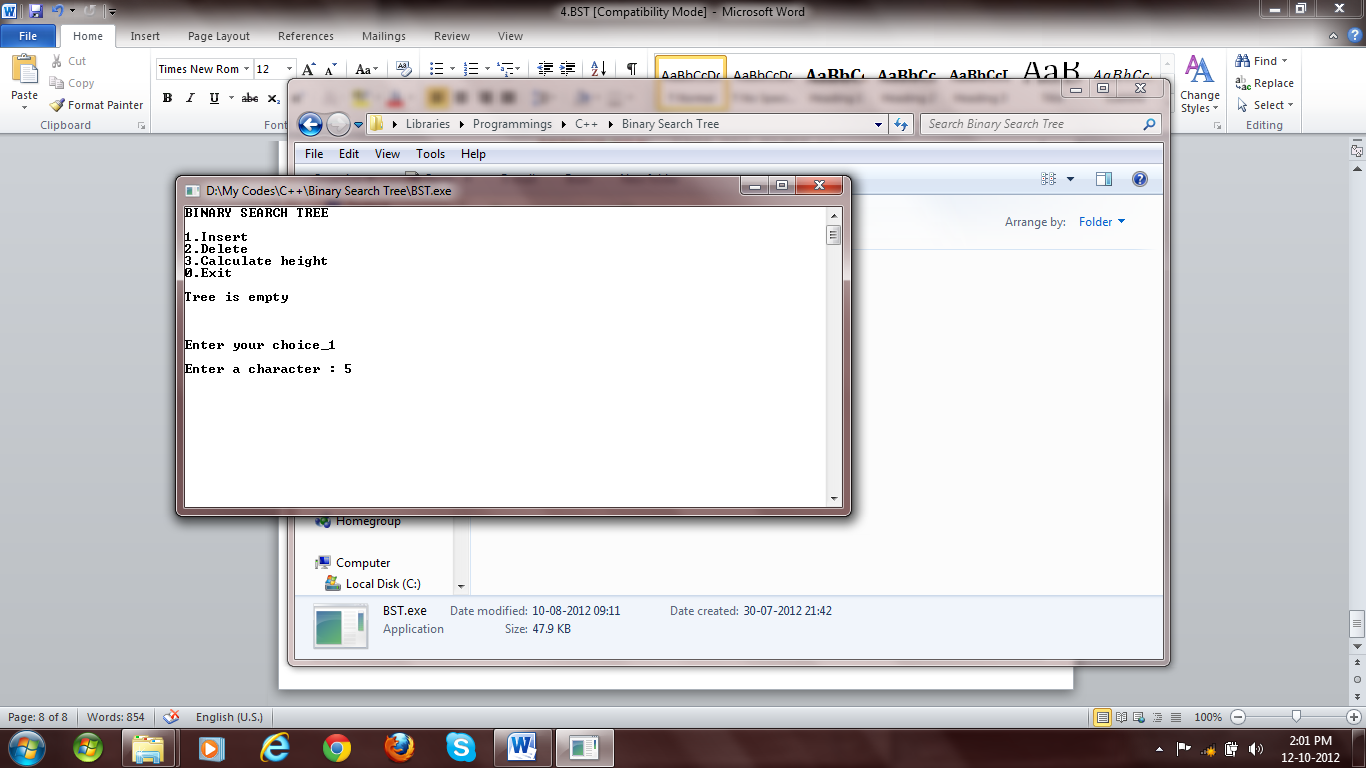


Fig 1: Empty Tree at start

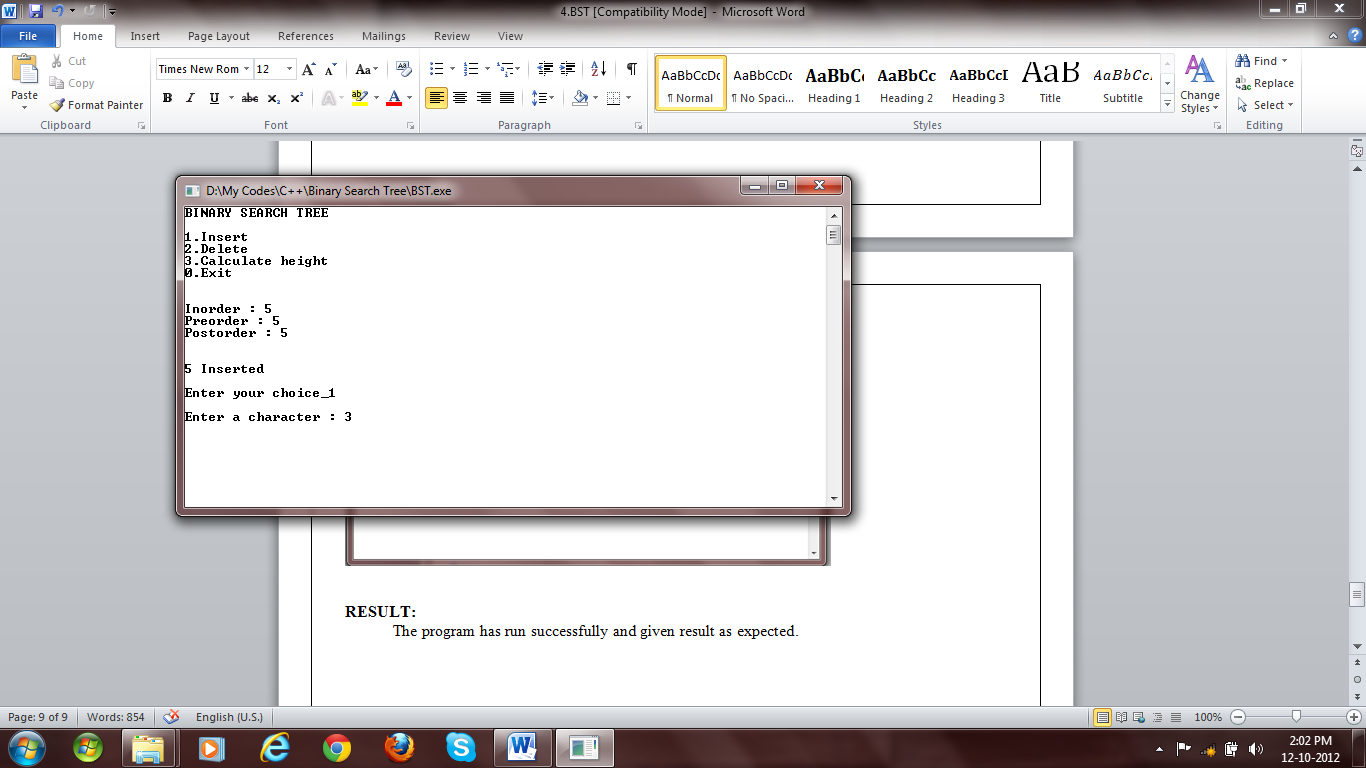


Fig 2: Insertion

# cout<<endl;

# }

# template <class DT> int BST<DT>::height() {

# Queue< Node<DT>\* > q(2\*numberOfNodes);

# q.enqueue(\*root);

# q.enqueue(NULL);

# int h=0;

# while(true) {

# Node<DT>\* temp=q.dequeue();

# if(temp==NULL) {

# if(q.isEmpty())

# break;

# else {

# h++;

# q.enqueue(NULL);

# }

# } else {

# if(temp->left!=NULL)

# q.enqueue(temp->left);

# if(temp->right!=NULL)

# q.enqueue(temp->right);

# }

# }

# return h+1;

# }

# main() {

# BST<char> tree;

# int choice=1;

# char e;

# char msg[50]="";

# while(choice) {

# system("cls");

# cout<<"BINARY SEARCH TREE"<<endl

# <<endl

# <<"1.Insert"<<endl

# <<"2.Delete"<<endl

# <<"3.Calculate height"<<endl

# <<"0.Exit"<<endl

# <<endl;

# if(tree.isEmpty())

# cout<<"Tree is empty"<<endl;

# else {

# cout<<endl

# <<"Inorder : ";

# tree.traverseInorder();

# cout<<"Preorder : ";

# tree.traversePreorder();

# cout<<"Postorder : ";

# tree.traversePostorder();

# cout<<endl;

# }

# cout<<endl

# <<msg<<endl

# <<endl

# <<"Enter your choice\_";

# cin>>choice;

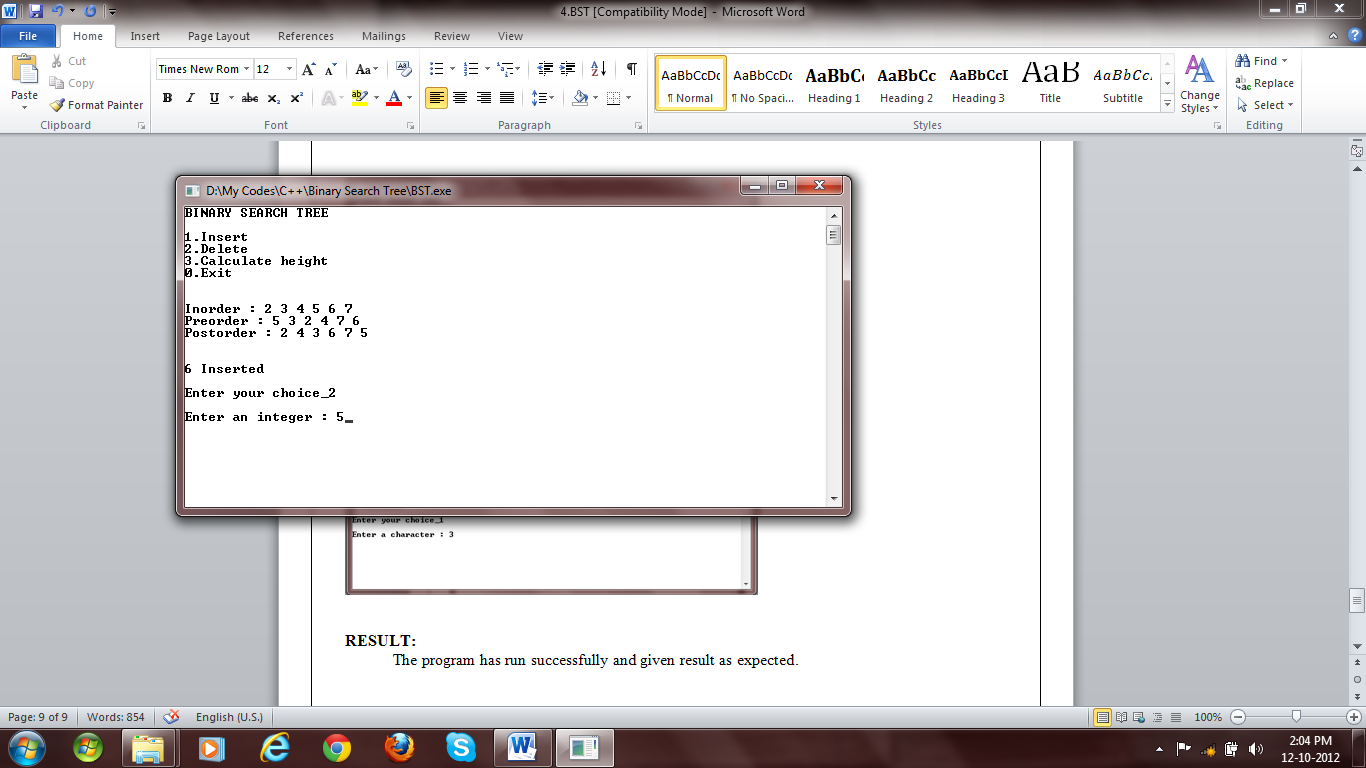
# cout<<endl;

# switch(choice) {

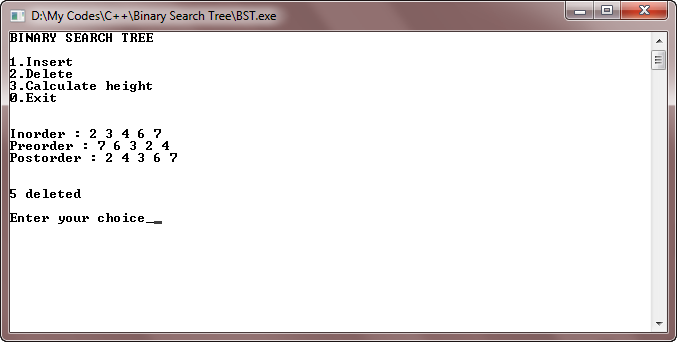
# case 0:

# return 0;

# case 1:



**Fig 3: Insertion**



**Fig 4: Deletion**

# cout<<"Enter a character : ";

# cin>>e;

# try{

# tree.insert(e);

# sprintf(msg, "%c Inserted", e);

# } catch (NodeAlreadyExistsException ex) {

# sprintf(msg, "Node already exists");

# }

# break;

# case 2:

# cout<<"Enter an integer : ";

# cin>>e;

# try {

# tree.remove(e);

# sprintf(msg, "%c deleted", e);

# }catch(NodeNotFoundException ex) {

# sprintf(msg, "Node not found");

# }

# break;

# case 3:

# try {

# sprintf(msg, "Height = %d", tree.height());

# } catch(...) {

# sprintf(msg, "Height = 0");

# }

# break;

# default:

# sprintf(msg, "Wrong choice");

# }

# }

# }

# RESULT:

The program has run successfully and given result as expected.