Name: Saransh Kalra

R number: R11528524

Lab section: 502

Lab 7

**BST.hpp**

#ifndef BST\_hpp

#define BST\_hpp

#include <stdio.h>

*//Node class for making nodes in the binary search tree*

**class** Node

{

*//variable for storing teh information in each node*

**int** info;

**public**:

*//constructor*

Node(**int** info);

*//accessor for the information stored in node*

**int** getInfo();

*//mutator for the information stored in node*

**void** setInfo(**int**);

*//left and right pointers to point to the left and right nodes in the BST*

Node \*left, \*right;

};

*//Class for the binary search tree*

**class** BST

{

*//root pointer to point to the root of the BST*

Node \*root;

**public**:

*//constructor*

BST();

*//destructor*

~BST();

*//destructor helper to make the destructor recursive*

**void** destr\_helper(Node\*);

*//insert function for inserting a new node into teh BST*

**void** insert(**int**);

*//search function for searching the value passed as argument in the BST*

**void** search(**int**);

*//recursive search helper function to make search easier*

Node\* search\_helper(Node\*, **int**);

*//recursive inorder traversal function to make the print easier*

**void** inorderTraversal(Node\*);

*//recursive preorder traversal function to make the print easier*

**void** preorderTraversal(Node\*);

*//recursive postorder traversal function to make the print easier*

**void** postorderTraversal(Node\*);

*//print function to print the BST according to inorder, preorder and postorder traversal*

**void** print();

*//function for counting the leaves in the BST*

**void** leavesCount();

*//recursive leaves count function helper to make leave counting function easier*

**int** leavesCountHelper(Node\*);

*//parent finder recursive helper for remove function*

Node\* parent\_helper(Node\*, **int**);

*//remove function to remove the argument passed from the BST*

**void** remove(**int** num);

};

#endif */\* BST\_hpp \*/*

**BST.cpp**

#include "BST.hpp"

#include <iostream>

**using** **namespace** std;

*//Node class constructor definition*

Node::Node(**int** info){

*//assigning the argument value to the info of the node*

**this**->info = info;

*//intialising its left to 0*

**this**->left = 0;

*//initialising its right to zero*

**this**->right = 0;

}

*//accessor for the information stored in node*

**int** Node::getInfo(){

*//return the information stored in the node*

**return** **this**->info;

}

*//mutator for the information stored in node*

**void** Node::setInfo(**int** info){

*//sets the argument value to the info of the node*

**this**->info = info;

}

*//constructor for the BST class*

BST::BST() {

*//initialises the root to 0*

**this**->root = 0;

}

*//destructor for the BST class*

BST::~BST(){

*//caling the destructor helper with root as the argument*

**this**->destr\_helper(**this**->root);

}

*//recursive destructor helper function for making destructor easier*

**void** BST::destr\_helper(Node\* node){

*//if node exists*

**if**(node){

*//do postorder traversal for the whole BST and delete all the nodes*

**this**->destr\_helper(node->left);

**this**->destr\_helper(node->right);

**delete** node;

}

}

*//insert function for inserting a new node to the BST*

**void** BST::insert(**int** info) {

*//if root doesn't exist*

**if**(!**this**->root) {

*//make a new node with the argument as the constructor argument*

Node \*p = **new** Node(info);

*//assign that node to the root*

**this**->root = p;

}

*//else add according to its info if its bigger smaller or equal to*

**else** {

*//variable flag to exit out of the loop*

**int** flag = 0;

*//make a new node with the argument as the constructor argument*

Node \*add = **new** Node(info);

*//make a traversal node initialized to root*

Node \*p1 = **this**->root;

*//while flag is 0, be in the loop i.e. we can exit the loop whenever we want to making flag 1*

**while**(flag==0) {

*//if the value to be added is greater that the traversal node it will be stored in the right side*

**if**(info > p1->getInfo()) {

*//if a right exists then update the traversal node to be its right*

**if**(p1->right) {

p1 = p1->right;

}

*//if a right doesnt exist then add the new node at the right and make flag 1 to exit out of the loop*

**else** {

p1->right = add;

flag = 1;

}

}

*//if the value to be added is less than the current traversal node then it would be stored in the left side*

**else** **if** (info < p1->getInfo()) {

*//if a left exists then update the traversal node to be its left*

**if**(p1->left) {

p1 = p1->left;

}

*//if a left doesnt exist then add the new node at the left and make flag 1 to exit out of the loop*

**else** {

p1->left = add;

flag = 1;

}

}

*//else it means its equal to the node info already in ther tree which means we should give an error that duplicates are not allowed and exit the loop by making flag 1*

**else** {

cout<<"Duplicate not allowed! this value already exists in the BST!"<<endl;

flag = 1;

}

}

}

}

*//search function for searching the value passed as argument in the BST*

**void** BST::search(**int** info) {

*//if there is no root give error that the BST is empty*

**if**(!**this**->root) {

cout<<"Nothing to search for! The Binary Search Tree is empty"<<endl;

}

*//else search for the node*

**else** {

*//if the search hepler function returns true then display that this value exists in the BST*

**if**(**this**->search\_helper(**this**->root, info)) {

cout<<"Yes, "<<info<<" exists in the Binary Search Tree."<<endl;

}

*//else display that it doesn't exist in the BST*

**else** {

cout<<"The value "<<info<<" was not found in the Binary Search Tree."<<endl;

}

}

}

*//recursive search helper function to make search easier*

Node\* BST::search\_helper(Node \*node, **int** info) {

*//if a node doesn't exist return the node i.e. NULL*

**if**(!node) {

*//return false*

**return** node;

}

*//if the value is found at this node return the node*

**else** **if** (node->getInfo() == info) {

*//return true*

**return** node;

}

*//if its less than the node passed as argument then pass the left of the node as an argument and call the search helper again because if it is in BST will be found to the left*

**else** **if** (info < node->getInfo()) {

**return** **this**->search\_helper(node->left, info);

}

*//if its more than the node passed as argument then pass the right of the node as an argument and call the search helper again because if it is in BST will be found to the right*

**else** {

**return** **this**->search\_helper(node->right, info);

}

}

*//recursive inorder traversal function to make the print easier*

**void** BST::inorderTraversal(Node \*temp) {

*//if temp exists follow LVR that means go to left then print that node and then go to right*

**if**(temp) {

inorderTraversal(temp->left);

cout<<temp->getInfo()<<" ";

inorderTraversal(temp->right);

}

}

*//recursive preorder traversal function to make the print easier*

**void** BST::preorderTraversal(Node \*temp) {

*//if temp exists follow VLR that means print that node then go to left and then go to right*

**if**(temp) {

cout<<temp->getInfo()<<" ";

preorderTraversal(temp->left);

preorderTraversal(temp->right);

}

}

*//recursive postorder traversal function to make the print easier*

**void** BST::postorderTraversal(Node \*temp) {

*//if temp exists follow LRV that means go to left then go to right and then print that node*

**if**(temp) {

postorderTraversal(temp->left);

postorderTraversal(temp->right);

cout<<temp->getInfo()<<" ";

}

}

*//print function to print the BST according to inorder, preorder and postorder traversal*

**void** BST::print() {

*//if the root doesn't exist, give an error that the BST is empty*

**if**(!**this**->root) {

cout<<"Nothing to display! The Binary Search Tree is empty"<<endl;

}

*// else just call the inorder, preorder and postorder traversal helper with root as the argument and print the BST with that in that particular order*

**else** {

cout<<" Inorder Traversal: "<<endl;

cout<<"{ ";

inorderTraversal(**this**->root);

cout<<"}"<<endl;

cout<<" Preorder Traversal: "<<endl;

cout<<"{ ";

preorderTraversal(**this**->root);

cout<<"}"<<endl;

cout<<" Postorder Traversal: "<<endl;

cout<<"{ ";

postorderTraversal(**this**->root);

cout<<"}"<<endl;

}

}

*//function for counting the leaves in the BST*

**void** BST::leavesCount() {

*//if the root doesn't exist, give an error that the BST is empty*

**if**(!**this**->root) {

cout<<"Nothing to count leaves for! The Binary Search Tree is empty"<<endl;

}

*//else give the nodes in the BST*

**else** {

*//pass the root as an argument to the leaves count helper function ehich returns the numer of leaves in the BST and print it*

Node \*p = **this**->root;

cout<<"No of leaves in the BST: "<<**this**->leavesCountHelper(p)<<endl;

}

}

*//recursive leaves count function helper to make leave counting function easier*

**int** BST::leavesCountHelper(Node \*temp) {

*//if the node exists*

**if**(temp) {

*//store the leaves count in the left side of the current node in a*

**int** a = **this**->leavesCountHelper(temp->left);

*//store the leaves count in the right side of the current node in b*

**int** b = **this**->leavesCountHelper(temp->right);

*//initialise c to zero which is used for storing the total leaves till this node*

**int** c = 0;

*//if there is no left or right children then its a leaf so add one to the leaves it has to the left and right and then return the total*

**if**(!(temp->left) && !(temp->right)) {

c = 1 + a + b ;

}

*//else its a not a leaf so add zero to the leaves it has to the left and right and then return the total*

**else** {

c = 0 + a + b ;

}

*//returning the total*

**return** c;

}

*//if the node is null return 0*

**else** {

**return** 0;

}

}

*//parent finder recursive helper for remove function*

Node\* BST::parent\_helper(Node \*node, **int** info) {

*//if the node doesn't exist*

**if** (node == **NULL**) {

*//return NULL*

**return** **NULL**;

}

*//else if the root is the node to be deleted then return null as it has no parent*

**else** **if** (**this**->root->getInfo() == info) {

**return** **NULL**;

}

*//else if it has two children*

**else** **if** (node->right && node->left) {

*//return the node as parent of any of its children have the value to be deleted*

**if** ((node->left->getInfo() == info) || (node->right->getInfo() == info)) {

*//return true*

**return** node;

}

*//else if the current node has value bigger than whats required go to the left of it*

**else** **if** (node->getInfo() > info) {

**return** **this**->parent\_helper(node->left, info);

}

*//else go the right of it*

**else** {

**return** **this**->parent\_helper(node->right, info);

}

}

*//else if the node has just a right child just check the right child*

**else** **if** (node->right && !node->left) {

*//if the right child has the node to be deleted return the node as parent*

**if** (node->right->getInfo() == info) {

*//return true*

**return** node;

}

*//else if the current node has value bigger than whats required go to the left of it*

**else** **if** (node->getInfo() > info) {

**return** **this**->parent\_helper(node->left, info);

}

*//else go the right of it*

**else** {

**return** **this**->parent\_helper(node->right, info);

}

}

*//else if the node has just a left child just check the right child*

**else** **if** (!node->right && node->left) {

*//if the left child has the node to be deleted return the node as parent*

**if** (node->left->getInfo() == info) {

*//return true*

**return** node;

}

*//else if the current node has value bigger than whats required go to the left of it*

**else** **if** (node->getInfo() > info) {

**return** **this**->parent\_helper(node->left, info);

}

*//else go the right of it*

**else** {

**return** **this**->parent\_helper(node->right, info);

}

}

*//else*

**else** {

*//if the current node has value bigger than whats required go to the left of it*

**if** (node->getInfo() > info) {

**return** **this**->parent\_helper(node->left, info);

}

*//else go the right of it*

**else** {

**return** **this**->parent\_helper(node->right, info);

}

}

}

*//remove function to remove the argument passed from the BST*

**void** BST::remove(**int** num) {

*//variable to store the node to be deleted, searched in the BST using search helper*

Node \*remNode = **this**->search\_helper(**this**->root, num);

*//if the node to be deleted doesn't exist then display error that the value deos not exist in the BST*

**if** (!remNode) {

cout<<"node you want to delete does not exist! "<<endl;

}

*//else*

**else** {

*//serach for the parent of the node to be deleted using parent helper and store in this variable*

Node \*remNode\_parent = parent\_helper(**this**->root, num);

*//if the node to be deleted has no child*

**if**(!(remNode->right || remNode->left)) {

*//if the node to be deleted has a parent*

**if**(remNode\_parent) {

*//if the node to be deleted is the left child then set the left child of the parent to be null*

**if**(remNode->getInfo() < remNode\_parent->getInfo()) {

remNode\_parent->left = 0;

}

*//else it is the right child then set the right child of the parent to be null*

**else** {

remNode\_parent->right = 0;

}

}

*//else it means it doesnt have a parent, so its the root, set it to be null*

**else** {

**this**->root = 0;

}

*//delete the node*

**delete** remNode;

}

*//else if the node to be deleted has two children*

**else** **if**(remNode->right && remNode->left) {

*//make a temporary pointer which will store the value that will take the place of the deleted node initialised to the right of the node to be deleted*

Node \*temp = remNode->right;

*//while there is a left go to left means the leftmost node*

**while**(temp->left) {

temp = temp->left;

}

*//make the remnode parent store the parent of the node that will take place of the deleted node using parent helper*

remNode\_parent = parent\_helper(**this**->root, temp->getInfo());

*//if the value that will replace is a right child*

**if**(temp->getInfo() > remNode\_parent->getInfo()) {

*//then set its parent's right to be its right*

remNode\_parent->right = temp->right;

}

*//else set its parent's left to be its right*

**else** {

remNode\_parent->left = temp->right;

}

*//replace the value in place of node to be deleted*

remNode->setInfo(temp->getInfo());

*//finally delete the node*

**delete** temp;

}

*//else means it has only one child*

**else** {

*//if it doesnt have a parent meaning its the root*

**if**(!remNode\_parent) {

*//if it has a left child*

**if**(remNode->left) {

*//set the left child to be the new root*

**this**->root = remNode->left;

}

*//else set the right child to be the new root*

**else** {

**this**->root = remNode->right;

}

}

*//else its not a root then*

**else** {

*//if it has a left child only*

**if**(remNode->left) {

*//if the value to be deleted was the left child*

**if**(remNode->getInfo() < remNode\_parent->getInfo()) {

*//set the left child to be the deleted node's left*

remNode\_parent->left = remNode->left;

}

*//else set the right child to be the deleted node's left*

**else** {

remNode\_parent->right = remNode->left;

}

}

*//else it has a right child only*

**else** {

*//if the value to be deleted was the left child*

**if**(remNode->getInfo() < remNode\_parent->getInfo()) {

*//set the left child to be the deleted node's right*

remNode\_parent->left = remNode->right;

}

*//else set the right child to be the deleted node's right*

**else** {

remNode\_parent->right = remNode->right;

}

}

*//finally delete the node*

**delete** remNode;

}

}

}

}

**Main.cpp**

#include <iostream>

#include "BST.hpp"

**using** **namespace** std;

**int** main() {

*//instance variable for the BST class*

BST example;

*//initialising the count variable to 0 for counting numbers entered*

**int** count = 0;

*//initialising the ch storing yes/no for the menu to y*

**char** ch = 'y';

*//initialising num to zero for storing the number entered by the user*

**int** num = 0;

*//call the search function to search for number 5 when the BST is empty*

example.search(5);

*//call the print function to print the BST according to inorder traversal algorithm if BST is empty*

example.print();

*//call the leaves count function to count the number of leaves in the BST if BST is empty*

example.leavesCount();

*//ask the user for atleast 5 numbers*

cout<<"Enter minimum 5 numbers to make a binary search tree"<<endl;

*//while ch is y means the user wants to continue the loop runs*

**while**(ch == 'y') {

*//ask for a number from the user*

cout<<"Enter a number to insert into the BST: ";

cin>>num;

*//increase the count of the numbers entered by 1*

count++;

*//insert the number entered by the user to the BST*

example.insert(**int**(num));

*//ask the user if he wants to continue entering more numbers once he has entered 5 numbers atleast*

**if**(count>=5) {

cout<<"Do you want to continue? ";

cin>>ch;

}

}

*//call the search function to search for number 5*

example.search(5);

*//call the search function to search for number 7*

example.search(7);

*//call the print function to print the BST according to inorder traversal algorithm*

example.print();

*//call the leaves count function to count the number of leaves in the BST*

example.leavesCount();

*//variable to store the number the user enters to be deleted from the BST initialised to 0*

**int** num1 = 0;

*//asking teh user for the number he wants to delete*

cout<<"Enter the number you want to delete: ";

cin>>num1;

*//remove that value by calling the remove function*

example.remove(num1);

*//calling the print function to show the BST after the delete*

example.print();

**return** 0;

}

**OUTPUT**

**Test case 1**

**Nothing to search for! The Binary Search Tree is empty**

**Nothing to display! The Binary Search Tree is empty**

**Nothing to count leaves for! The Binary Search Tree is empty**

**Enter minimum 5 numbers to make a binary search tree**

**Enter a number to insert into the BST:** 6

**Enter a number to insert into the BST:** 2

**Enter a number to insert into the BST:** 11

**Enter a number to insert into the BST:** 1

**Enter a number to insert into the BST:** 4

**Do you want to continue?** y

**Enter a number to insert into the BST:** 15

**Do you want to continue?** y

**Enter a number to insert into the BST:** 17

**Do you want to continue?** n

**The value 5 was not found in the Binary Search Tree.**

**The value 7 was not found in the Binary Search Tree.**

**Inorder Traversal:**

**{ 1 2 4 6 11 15 17 }**

**Preorder Traversal:**

**{ 6 2 1 4 11 15 17 }**

**Postorder Traversal:**

**{ 1 4 2 17 15 11 6 }**

**No of leaves in the BST: 3**

**Enter the number you want to delete:** 6

**Inorder Traversal:**

**{ 1 2 4 11 15 17 }**

**Preorder Traversal:**

**{ 11 2 1 4 15 17 }**

**Postorder Traversal:**

**{ 1 4 2 17 15 11 }**

**Program ended with exit code: 0**

**Test case 2**

**Nothing to search for! The Binary Search Tree is empty**

**Nothing to display! The Binary Search Tree is empty**

**Nothing to count leaves for! The Binary Search Tree is empty**

**Enter minimum 5 numbers to make a binary search tree**

**Enter a number to insert into the BST:** 2

**Enter a number to insert into the BST:** 1

**Enter a number to insert into the BST:** 6

**Enter a number to insert into the BST:** 3

**Enter a number to insert into the BST:** 7

**Do you want to continue?** y

**Enter a number to insert into the BST:** 4

**Do you want to continue?** y

**Enter a number to insert into the BST:** 0

**Do you want to continue?** n

**The value 5 was not found in the Binary Search Tree.**

**Yes, 7 exists in the Binary Search Tree.**

**Inorder Traversal:**

**{ 0 1 2 3 4 6 7 }**

**Preorder Traversal:**

**{ 2 1 0 6 3 4 7 }**

**Postorder Traversal:**

**{ 0 1 4 3 7 6 2 }**

**No of leaves in the BST: 3**

**Enter the number you want to delete:** 2

**Inorder Traversal:**

**{ 0 1 3 4 6 7 }**

**Preorder Traversal:**

**{ 3 1 0 6 4 7 }**

**Postorder Traversal:**

**{ 0 1 4 7 6 3 }**

**Program ended with exit code: 0**

**Test case 3**

**Nothing to search for! The Binary Search Tree is empty**

**Nothing to display! The Binary Search Tree is empty**

**Nothing to count leaves for! The Binary Search Tree is empty**

**Enter minimum 5 numbers to make a binary search tree**

**Enter a number to insert into the BST:** 5

**Enter a number to insert into the BST:** 2

**Enter a number to insert into the BST:** 12

**Enter a number to insert into the BST:** 1

**Enter a number to insert into the BST:** 3

**Do you want to continue?** y

**Enter a number to insert into the BST:** 9

**Do you want to continue?** y

**Enter a number to insert into the BST:** 21

**Do you want to continue?** y

**Enter a number to insert into the BST:** 19

**Do you want to continue?** y

**Enter a number to insert into the BST:** 25

**Do you want to continue?** n

**Yes, 5 exists in the Binary Search Tree.**

**The value 7 was not found in the Binary Search Tree.**

**Inorder Traversal:**

**{ 1 2 3 5 9 12 19 21 25 }**

**Preorder Traversal:**

**{ 5 2 1 3 12 9 21 19 25 }**

**Postorder Traversal:**

**{ 1 3 2 9 19 25 21 12 5 }**

**No of leaves in the BST: 5**

**Enter the number you want to delete:** 12

**Inorder Traversal:**

**{ 1 2 3 5 9 19 21 25 }**

**Preorder Traversal:**

**{ 5 2 1 3 19 9 21 25 }**

**Postorder Traversal:**

**{ 1 3 2 9 25 21 19 5 }**

**Program ended with exit code: 0**