**CSC 2201 – Computer Science II**

**Lab #9**

**Take-Home**

**60 Points**

1. Save a copy of this document with your name and the assignment number somewhere in the file name. For example, the file name *“Jane\_Doe\_CSC2200\_Lab1.docx”*
2. Copy-and-paste your answers (e.g., C++ source code) into the document.
3. Copy-and-paste the program output window.
4. Submit the following files separately (do not compress the files) to the Blackboard in one submission:

1) This document as a word document (i.e., with the extension ***.doc*** or ***.docx***).

2) All C++ source code solution file(s) (only the ***.cpp*** and ***.h*** files) to the Canvas item associated with this assignment/lab solution. ***\*\*If you modified it, submit it***

\*Submit entire Visual Studio solution, if possible, otherwise only the required files.

Questions:

Using the linked approach implement the BST ADT, implement only the following functions in the BSTree.cpp:

- *retrieve, remove, writeKeys (40)*

- *getCount, getHeight, writeLessThan (20)*

**Note**: Use recursive functions to traverse the tree. Read the implementation notes on using *helper* functions.

**Note**: *If you do not provide a screen shot of the tested functions I will take 10 pts*

*off.*

\* **Copying-and-pasting your *C++ program code* to a Word document**

1) From within the Visual Studio program, press **CTRL-A** and press **CTRL-C**.

2) From within the Word document, press **CTRL-V**.

#include "BSTree.h"

//Node constructor

template <typename DataType, class KeyType>

BSTree<DataType, KeyType>::BSTreeNode::BSTreeNode(const DataType& nodeDataItem, BSTreeNode\* leftPtr, BSTreeNode\* rightPtr)

{

dataItem = nodeDataItem;

left = leftPtr;

right = rightPtr;

}

//Default constructor

template < typename DataType, class KeyType >

BSTree<DataType, KeyType>::BSTree()

{

root = nullptr;

}

//Copy constructor

template < typename DataType, class KeyType >

BSTree<DataType, KeyType>::BSTree(const BSTree<DataType, KeyType>& other)

{

root = copyTree(other.root);

}

//Copy constructor helper

template < typename DataType, class KeyType >

typename BSTree<DataType, KeyType>::BSTreeNode\* BSTree<DataType, KeyType>::copyTree(BSTreeNode\* otherNode)

{

if (otherNode == nullptr) {

return nullptr; //Base case: if the node is NULL, return NULL

}

else {

//Recursive case: create a new node with the same data and copy its left and right subtrees

BSTreeNode\* newNode = new BSTreeNode(otherNode->dataItem, copyTree(otherNode->left), copyTree(otherNode->right));

return newNode;

}

}

//Assignment operator

template < typename DataType, class KeyType >

BSTree<DataType, KeyType>& BSTree<DataType, KeyType>:: operator= (const BSTree<DataType, KeyType>& other)

{

//Check for self-assignment

if (this != &other) {

//Clear the current tree to make room for the new data

clear();

//Perform a deep copy of the other tree using the copyTree helper function

root = copyTree(other.root);

}

//Return a reference to this object

return \*this;

}

//Destructor

template < typename DataType, class KeyType >

BSTree<DataType, KeyType>::~BSTree()

{

clear();

}

//Clear function

template < typename DataType, class KeyType >

void BSTree<DataType, KeyType>::clear()

{

clearHelper(root);

root = nullptr;

}

//Clear helper function

template < typename DataType, class KeyType >

void BSTree<DataType, KeyType>::clearHelper(BSTreeNode\* node)

{

if (node != nullptr) {

clearHelper(node->left);

clearHelper(node->right);

delete node;

}

}

//Insert function

template < typename DataType, class KeyType >

void BSTree<DataType, KeyType>::insert(const DataType& newDataItem)

{

root = insertOrUpdate(root, newDataItem);

}

//Insert function helper

template <typename DataType, class KeyType>

typename BSTree<DataType, KeyType>::BSTreeNode\* BSTree<DataType, KeyType>::insertOrUpdate(BSTreeNode\* node, const DataType& newDataItem)

{

if (node == nullptr)

{

//If the node is nullptr, create a new node with newDataItem

return new BSTreeNode(newDataItem, nullptr, nullptr);

}

else if (newDataItem.getKey() < node->dataItem.getKey())

{

//If the key of newDataItem is smaller, insert/update in the left subtree

node->left = insertOrUpdate(node->left, newDataItem);

}

else if (newDataItem.getKey() > node->dataItem.getKey())

{

//If the key of newDataItem is larger, insert/update in the right subtree

node->right = insertOrUpdate(node->right, newDataItem);

}

else

{

//If the key of newDataItem matches the current node, update the data item

node->dataItem = newDataItem;

cout << "Updating" << endl;

}

return node;

}

//Retrieve function

template < typename DataType, class KeyType >

bool BSTree<DataType, KeyType>::retrieve(const KeyType& searchKey, DataType& searchDataItem) const

{

return retrieveHelper(root, searchKey, searchDataItem);

}

//Retrieve helper function

template <typename DataType, class KeyType>

bool BSTree<DataType, KeyType>::retrieveHelper(BSTreeNode\* node, const KeyType& searchKey, DataType& searchDataItem) const

{

if (node == nullptr) {

return false; //Key not found

}

else if (searchKey < node->dataItem.getKey()) {

return retrieveHelper(node->left, searchKey, searchDataItem);

}

else if (searchKey > node->dataItem.getKey()) {

return retrieveHelper(node->right, searchKey, searchDataItem);

}

else {

searchDataItem = node->dataItem;

return true; //Key found

}

}

//Remove function

template < typename DataType, class KeyType >

bool BSTree<DataType, KeyType>::remove(const KeyType& deleteKey)

{

return removeHelper(root, deleteKey);

}

//Remove helper function

template <typename DataType, class KeyType>

bool BSTree<DataType, KeyType>::removeHelper(BSTreeNode\*& node, const KeyType& deleteKey)

{

if (node == nullptr) {

return false; //Key not found

}

if (deleteKey < node->dataItem.getKey()) {

return removeHelper(node->left, deleteKey);

}

else if (deleteKey > node->dataItem.getKey()) {

return removeHelper(node->right, deleteKey);

}

else {

//Node with the key found, perform deletion logic

if (node->left == nullptr) {

BSTreeNode\* temp = node;

node = node->right;

delete temp;

}

else if (node->right == nullptr) {

BSTreeNode\* temp = node;

node = node->left;

delete temp;

}

else {

//Node has two children, find successor and replace the node's data

BSTreeNode\* successor = findMin(node->right);

node->dataItem = successor->dataItem;

removeHelper(node->right, successor->dataItem.getKey());

}

return true; //Node with the key removed

}

}

//Find min function

template <typename DataType, class KeyType>

typename BSTree<DataType, KeyType>::BSTreeNode\* BSTree<DataType, KeyType>::findMin(BSTreeNode\* node) const

{

while (node->left != nullptr) {

node = node->left;

}

return node;

}

//Write keys function

template < typename DataType, class KeyType >

void BSTree<DataType, KeyType>::writeKeys() const

{

writeKeysHelper(root);

std::cout << std::endl;

}

//Write keys helper function

template <typename DataType, class KeyType>

void BSTree<DataType, KeyType>::writeKeysHelper(BSTreeNode\* node) const

{

if (node != nullptr) {

writeKeysHelper(node->left);

std::cout << node->dataItem.getKey() << " ";

writeKeysHelper(node->right);

}

}

//isEmpty function

template < typename DataType, class KeyType >

bool BSTree<DataType, KeyType>::isEmpty() const

{

if (root == NULL) {

return true;

}

else {

return false;

}

}

//getHeight function

template < typename DataType, class KeyType >

int BSTree<DataType, KeyType>::getHeight() const

{

return getHeightHelper(root);

}

//getHeight helper function

template <typename DataType, class KeyType>

int BSTree<DataType, KeyType>::getHeightHelper(BSTreeNode\* node) const

{

if (node == nullptr) {

return -1;

}

int leftHeight = getHeightHelper(node->left);

int rightHeight = getHeightHelper(node->right);

return 1 + std::max(leftHeight, rightHeight);

}

//getCount function

template < typename DataType, class KeyType >

int BSTree<DataType, KeyType>::getCount() const

{

return getCountHelper(root);

}

//getCount helper function

template <typename DataType, class KeyType>

int BSTree<DataType, KeyType>::getCountHelper(BSTreeNode\* node) const

{

if (node == nullptr) {

return 0;

}

return 1 + getCountHelper(node->left) + getCountHelper(node->right);

}

//writeLessThan function

template < typename DataType, class KeyType >

void BSTree<DataType, KeyType>::writeLessThan(const KeyType& searchKey) const

{

writeLessThanHelper(root, searchKey);

std::cout << std::endl;

}

//writeLessThan helper function

template <typename DataType, class KeyType>

void BSTree<DataType, KeyType>::writeLessThanHelper(BSTreeNode\* node, const KeyType& searchKey) const

{

if (node == nullptr) {

return;

}

if (node->dataItem.getKey() < searchKey) {

writeLessThanHelper(node->left, searchKey);

std::cout << node->dataItem.getKey() << " ";

writeLessThanHelper(node->right, searchKey);

}

else {

writeLessThanHelper(node->left, searchKey);

}

}

#include "show9.cpp"

\*\* **Copying-and-pasting a C++ “*output window”* to a Word document**

1) From the Visual Studio output window, press **ALT-PrintScreen**.

2) From within the Word document, press **CTRL-V**.

A black screen with a black background

Description automatically generated