Instructor: Dr. Jie Shen
Release date: April 12, 2021
Due date: April 21, 2021

Student Name: Demetrius Johnson

*Special note: see my lab uploads of the .CPP and .EXE files for ease of access and testing any of the programs for any question.

Table of Contents

Question 1 Tree

In this assignment, perform the following four coding tasks:

(1) Create a struct TreeNode

(2) Use this TreeNode to construct a binary search tree based on the following input queue:

F, A, B, M, C, Q

(3) Use Inorder Traversal to print out the content of this tree

You need to provide screenshots of your code and running result.

Source code (USED C++ COMPILER on Microsoft Windows 10)

**note: I also uploaded these two .cpp files to canvas so you can view them easier.

MAIN FILE:

```
// CIS-200-LAB_11 -DemetriusJohnson.cpp : This file contains the 'main' function. Program execution
begins and ends there.
//

/*
//Author: Demetrius E Johnson
//Date: 15 April 2021
//Last Modification Date: 04-19-2021
//Purpose: Use a tree struct node to demonstrate the structure of a Binary Search Tree data type
*/

/*
Question 1:

Question 1  Tree
In this assignment, perform the following four coding tasks:
(1)         Create a struct TreeNode
(2)         Use this TreeNode to construct a binary search tree based on the following input queue:
               F,  A,  B,  M,  C, Q

(3)         Use Inorder Traversal to print out the content of this tree

You need to provide screenshots of your code and running result.

*/



#include <iostream>
#include "BinarySearchTree.cpp" //had to just use a cpp file since template classes require more than just
the declaration (because of how the compiler works at compile time)
```

```
//#include<assert.h>
using namespace std;


//FUNCTION DECLARATIONS



//FUNCTION DECLARATIONS




int main()
{
   cout << "--WELCOME: This program uses a tree struct node to demonstrate the implementation of a
Binary Search Tree data type--\n--BY Demetrius Johnson--\n\n\n";


   BST<char> charBST;

   cout << "Below is the In-Order print of a char Binary Search Tree:\n\n";
   charBST.insert('F');
   charBST.insert('A');
   charBST.insert('B');
   charBST.insert('M');
   charBST.insert('C');
   charBST.insert('Q');
   //charBST.remove('B'); //used for testing; function works but it is defective because then destructor will
no longer work

   charBST.print_InOrder();

   //did some additional testing below with a different data type to see if my program would hold up!

   BST<int> intBST;

   cout << "\n\n\n\nBelow is the In-Order print of an int Binary Search Tree:\n\n";
   intBST.insert(1);
   intBST.insert(4);
   intBST.insert(-4);
   intBST.insert(10);
   intBST.insert(33);
   intBST.insert(7);


   intBST.print_InOrder();


   cout << endl << endl << "\n\n\nThe program has finished execution....now exiting...thank you....\n\n";
   system("pause");

   return 0;
}
```

~END MAIN FILE

BST IMPLEMENTATION FILE:

```
/*
//Author: Demetrius E Johnson
//Date: 15 April 2021
//Last Modification Date: 04-19-2021
//Purpose: Use a tree struct node to demonstrate the structure of a Binary Search Tree data type
*/

//#include <vector>
#include<iostream>
//TREE NODE STRUCT:
template<class ItemType>
struct TreeNode
{

	ItemType info;  //hold the actual data information for the current node
	TreeNode<ItemType>* left; //left child pointer
	TreeNode<ItemType>* right; //right child pointer
	//bool markedForDeletion = false; //use this to assist with the destructor function
};

//BINARY SEARCH TREE CLASS:
template<class ItemType>
class BST
{

private:
	int length;
	TreeNode<ItemType>* tempPtr; //use this to point to different parts of the tree; this will serve to
help with writing insert and delete functions
	TreeNode<ItemType>* root; //use this to store the address of the first element (which is also the
root) of the tree structure
	//std::vector<TreeNode<ItemType>*> nodeVector; //use this to store all addresses to assist with
the destructor function

public:
	BST() {

		tempPtr = nullptr;
		root = nullptr;
		length = 0;

	} //default constructor

	bool binarySearch(ItemType searchItem) {


		if (length == 0) { return false; } //execute this if length is 0; no nodes to search

		tempPtr = root; //start search at the root node

		while (true) { //execute this loop as many times as needed until return false or true occurs
within the loop for if item is found or not

			if (searchItem == tempPtr->info) { return true; } //item found; return true
```

```
				else if (searchItem > tempPtr->info && tempPtr->right != nullptr) { //binary search using advantages of a BST structure: move to right child if greater

					tempPtr = tempPtr->right;
				}
				else if (searchItem < tempPtr->info && tempPtr->left != nullptr) {//binary search using advantages of a BST structure: move to left child if smaller

					tempPtr = tempPtr->left;
				}
				else { return false; } //item not found; there are no other nodes to search
			}
		}

		void insert(ItemType addItem) {

			bool isItemInTree;
			isItemInTree = binarySearch(addItem);
			if (isItemInTree == true) { return; } //item already in tree, no need to execute insertion alogrithm
			if (length == 0) { //execute this if length is 0, which mean root node is null

				root = new TreeNode<ItemType>; //allocate memory for the first node which is also the root to store the first element of the tree
				root->info = addItem; //store the item in the new node
				root->right = nullptr; //set right and left pointers to null since the new node has no children
				root->left = nullptr;
				length++;
				return;
			}

			tempPtr = root; //set temp pointer to root so that we start at first element during the below while loop

			while (true) { //this loop continues indefintely until the item is inserted, which should happen under all circumstances

				if (addItem > tempPtr->info && tempPtr->right == nullptr) { //use this when we have finally reached a null child node

					  //and thus now it is time to allocate memory and add the item

					tempPtr->right = new TreeNode<ItemType>; //allocate memory for new node to store the info
					tempPtr = tempPtr->right; //move to the new node
					tempPtr->info = addItem; //store the item in the new node
					tempPtr->right = nullptr; //set right and left pointers to null since the new node has no children
					tempPtr->left = nullptr;
					length++;
					return;
					//use this for when item needs to be added and is Larger
				}
				if (addItem < tempPtr->info && tempPtr->left == nullptr) { //use this when we have finally reached a null child node
```

```
                                        //and thus now it is time to allocate memory and add the item

                                        tempPtr->left = new TreeNode<ItemType>; //allocate memory for new
node to store the info
                                        tempPtr = tempPtr->left; //move to the new node
                                        tempPtr->info = addItem; //store the item in the new node
                                        tempPtr->right = nullptr; //set right and left pointers to null since the
new node has no children
                                        tempPtr->left = nullptr;
                                        length++;
                                        return;
                                        //use this for when item needs to be added and is Smaller
                              }
                              else if (addItem > tempPtr->info) { //use this to compare item to current node
and move to RIGHT node if it is larger

                                        tempPtr = tempPtr->right;
                              }
                              else if (addItem < tempPtr->info) { //use this to compare item to current node
and move to LEFT node if it is smaller

                                        tempPtr = tempPtr->left;
                              }

                    }

          } //insert function

          //I wanted to challenge myself so I wrote the remove function to see if I could figure it out;
          //it works but it negatively affects other functions such as the destructor; thus I don't test it/use it
          void remove(ItemType removeItem){

                    bool isItemInTree;
                    isItemInTree = binarySearch(removeItem); //this also sets tempPtr = to the node where
the item was found
                    if (length == 0 || isItemInTree == false) { return;} //execute this if length is 0 or if item is
not in the tree for removal

                    if (tempPtr->right == nullptr && tempPtr->left == nullptr) { //simple case; no children;
thus we can simply delete the node contaning the removal item

                              delete tempPtr;
                              length--;
                              return;
                    }

                    if (removeItem == root->info) { //execute this special case in the event that the ROOT is
to be DELETED from the tree

                              if (tempPtr->left != nullptr) { //if the root node has a left child, point to that node

                                        tempPtr = tempPtr->left; //point to left node

                                        if (tempPtr->right == nullptr) { //execute this if the left root branch's
right branch is empty; this means we can simply treat the left branch as the new root
```

```
                    tempPtr->right = root->right; //we can simply let the left root
branch take on the right branch if the left branch has an empty right branch
                    delete root; //delete old root
                    root = tempPtr; //set new root
                    length--;
                    return; //item removed; exit function
                }
                 //execute below to do a overwrite and delete removal algorithm
                while (true) { //go to right most node while in the left root branch,
                                        //which will be the largest node and
the node closest to the size of the current root

                    tempPtr = tempPtr->right; //move to the next right node
                    if(tempPtr->right == nullptr){ //use this to check: if the next
right node is null,

//then we have found the right-most (largest) value in the root's left branch

                        root->info = tempPtr->info; //overwrite the root node
info, which is to be removed, and replace its value with the largest value in the left root branch
                        delete tempPtr; //delete old node which has been
moved to the root
                        length--;
                        return;
                    }
                }
            }
            else { //else if left branch is empty, simply set root = right branch and delete the
old root

                root = root->right;
                delete tempPtr;
                length--;
                return;
            }
        }

        //now execute the below if the root is not the item to be removed:

        TreeNode<ItemType>* itemPtr = tempPtr; //use this helper ptr to store a the node where
the item is to be removed

        if (itemPtr->right == nullptr) { //execute if the right branch of the removal item node is
null

            itemPtr->info = tempPtr->left->info; //overwrite removal node with the next
largest value: the left branch value
            itemPtr->left = itemPtr->left->left; //set left ptr to skip over the item that was
just used to overwrite the removed item
            delete tempPtr->left; //delete the old left node
            length--;
            return;

        }
```

```
		while (true) { //go to right most node so we can use that larger value to overwrite the
node to be removed

			tempPtr = tempPtr->right; //move to the next right node
			if (tempPtr->right == nullptr) { //use this to check: if the next right node is null,
then we have found the right-most node

				itemPtr->info = tempPtr->info; //overwrite the removal item node info,
which is to be removed, and replace its value with the larger value in the right-most branch
				delete tempPtr; //delete old node which has been moved to the where
the removed item was overwrtten
				length--;
				return;
			}
		}

	} //delete function

	void print_InOrder(void) { print_InOrder(root); } //use this as the no-parameter call, which prints
a tree using the root as the default tree

	void print_InOrder(TreeNode<ItemType>* nodeToPrint) {

		//use recursion

		if (nodeToPrint != nullptr) { //base case: check to make sure current node is not null

			if (nodeToPrint->left != nullptr) {

				print_InOrder(nodeToPrint->left); //move to left most-node by always
moving to left node of the current node
			}

			std::cout << nodeToPrint->info << " "; //now that we are at the left-most portion
of a given tree or sub=tree, print current node;

//this statement must go here in order to correctly print as Inorder tree print

			if (nodeToPrint->right != nullptr) { //when we reach this if statement, it means
the left-most nodes were already taken care of and printed

//this is in accordance with in-order print alogrithm; now we can move to the right nodes and print if
necessary

				print_InOrder(nodeToPrint->right);

			}


		}
	} //print items using in-order traversal

	int num_Elements(void) { return length; } //print out the number of elements in the tree

	~BST() {
```

recursiveDestructor(root); //need to take advantage of recursion so I will tell the destructor to call a recursive function I have written

} //destructor: need to deallocate all tree nodes

void recursiveDestructor(TreeNode<ItemType>* nodeToDelete) {

if (nodeToDelete != nullptr) { //use this first if to ensure BST is not an empty tree

if (nodeToDelete->right != nullptr) { //use this to check right branches of any nodes

recursiveDestructor(nodeToDelete->right);

}
if (nodeToDelete->left != nullptr) { //use this to check left branches of any nodes

recursiveDestructor(nodeToDelete->left);

}

delete nodeToDelete; //now we have finally either reached a leaf node, or deleted all nodes below the current node, thus we are ready to delete current node

}

}
};

~END OF BST IMPLEMENTATION FILE

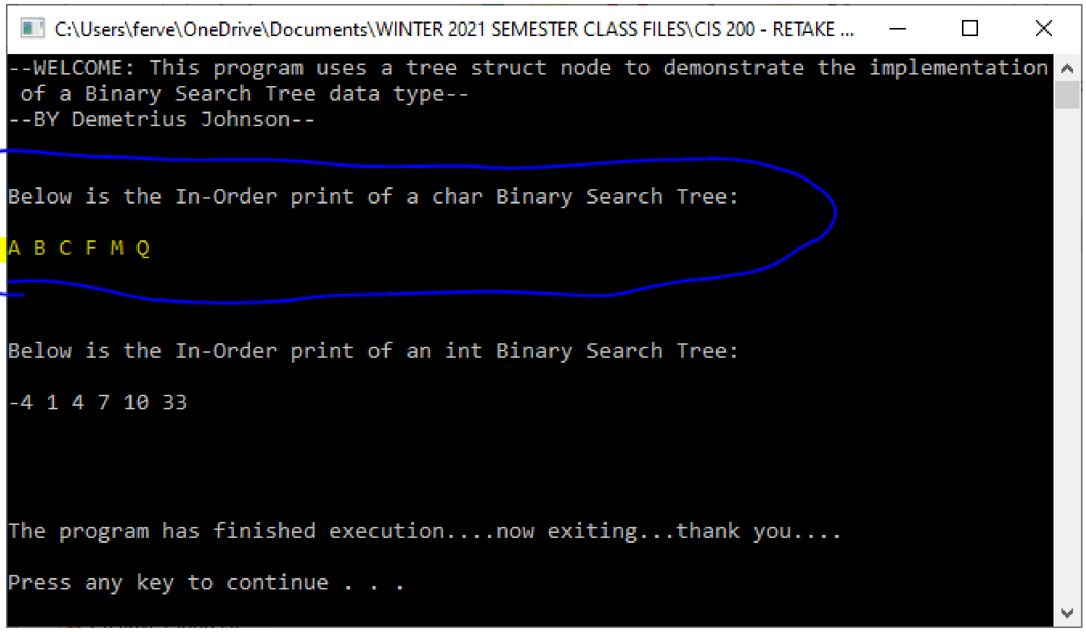
Test data and expected results

Test Table:

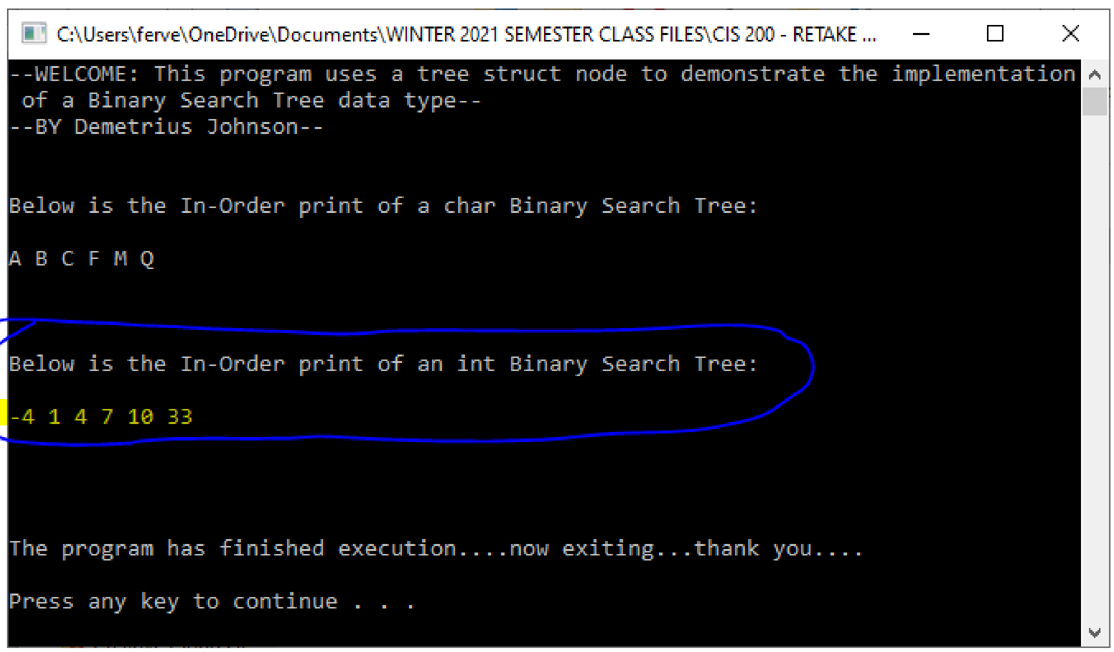
Test #	Valid / Invalid Data	Description of test	Input Value	Expected Output	Actual Output	Test Pass / Fail
1	valid	Test in-order print of a charBST	F A B M C Q	ABCFMQ	See screenshot	pass
2	valid	Test in-order print an intBST	1 -4 4 10 33 7	-4 1 4 7 10 33	See screenshot	pass

*Notes: I had some struggles with the destructor and in-order print functions but eventually figured out that I needed to use recursion. Just for practice, I also tried to right a remove function; it worked properly but it causes other functions not work properly, but it was good practice; I had difficulty with resetting the pointer from the actual tree to null after deleting it; I only set the temporary pointer to nullptr which has no effect on the other pointer that is actually in the root tree…Thus, it causes issues with my other functions. Overall, I can't believe I was able to finish this lab, it was very challenging, but it works!! And I wrote it as a template BST class!

TEST 1: CHAR BST



TEST 2: INT BST



Submission of Your Work:

The Word document should contain the following information

- Your name
- Machine type (Unix, Mac, Linux or PC machine ?)
- Compiler type
- Description of your code design and implementation
- Inclusion of your source
- A reasonable number of comment lines in your source code
- Screen shot of your test run (required)