CS302: Data Structures using C++

Fall 2020

Final Exam Preparation - Additional Exercises: **Binary Search Tree and Inorder, Preorder, Postorder Traversal**

Kostas Alexis

Problem Description: This comes from the associated assignment for Binary Search Trees as released in the Fall 2019, 2020 semesters. It asks to randomly generate 100 unique values in the range of [1-200] and insert them into a binary search tree (BST). Print height and inorder, preorder, and postorder output of the BST tree. To implement the tree an interfacing code in the file BinarySearchTree.h was provided. We deliver source code and test file that shows the result of printing height, inorder, preorder and postorder traversal. Furthermore, comment on the computational complexity of BST operations.

Solution: The solution is provided in the files below:

File: BinaryNode.h

```
#ifndef BINARY_NODE_
#define BINARY_NODE_

#include <memory>

template<class ItemType>
class BinaryNode
{
    private:
        ItemType item;
        BinaryNode<ItemType> *leftChildPtr;
        BinaryNode<ItemType> *rightChildPtr;
    public:
        //Default ctor
        BinaryNode();
        //Param ctor
        BinaryNode(const ItemType & anItem);

        //Set item to item private member
        void setItem(const ItemType & anItem);

        //Returns item
        ItemType getItem() const;

        //Is node a leaf?
```

```
        bool isLeaf() const;

        //Returns child ptrs
        BinaryNode<ItemType> * getLeftChildPtr() const;
        BinaryNode<ItemType> * getRightChildPtr() const;

        //Sets child ptr
        void setLeftChildPtr(BinaryNode<ItemType> * leftPtr);
        void setRightChildPtr(BinaryNode<ItemType> * rightPtr);
};

#include "BinaryNode.cpp"
#endif
```

File: BinaryNode.cpp

```
#include "BinaryNode.h"

template<class ItemType>
//Default ctor - add stuff maybe?
BinaryNode<ItemType>::BinaryNode()
{}

//Parametrized ctor, sets all private members to default values
template<class ItemType>
BinaryNode<ItemType>::BinaryNode(const ItemType & anItem) : item(anItem), leftChi
ldPtr(nullptr) , rightChildPtr(nullptr)
{}

//Return item, which is of ItemType (from template)
template<class ItemType>
ItemType BinaryNode<ItemType>::getItem() const
{
    return item;
}

//Sets item to anItem
template<class ItemType>
void BinaryNode<ItemType>::setItem(const ItemType & anItem)
{
    item = anItem;
}

//Getters and Setters
template<class ItemType>
BinaryNode<ItemType> * BinaryNode<ItemType>::getLeftChildPtr() const
{
    return leftChildPtr;
```

```
}
template<class ItemType>
BinaryNode<ItemType> * BinaryNode<ItemType>::getRightChildPtr() const
{
    return rightChildPtr;
}
template<class ItemType>
void BinaryNode<ItemType>::setLeftChildPtr(BinaryNode<ItemType> *leftPtr)
{
    leftChildPtr = leftPtr;
}
template<class ItemType>
void BinaryNode<ItemType>::setRightChildPtr(BinaryNode<ItemType> * rightPtr)
{
    rightChildPtr = rightPtr;
}
```

File: BinarySearchTree.h

```
#ifndef BINARY_SEARCH_TREE_
#define BINARY_SEARCH_TREE_

#include "BinaryNode.h"
#include <memory>
#include <algorithm>
#include <iostream>

template<class ItemType>
class BinarySearchTree
{
    private:
        BinaryNode<ItemType> *rootPtr;
    protected: //Could just make public?
        BinaryNode<ItemType> * placeNode(BinaryNode<ItemType> *subTreePtr, Binary
Node<ItemType> * newNode);
        BinaryNode<ItemType> * removeValue(BinaryNode<ItemType> * subTreePtr, con
st ItemType & target, bool & isSuccessful);
    public:
        BinarySearchTree();
```

```
        BinarySearchTree(const ItemType & rootItem);
        BinarySearchTree(const BinarySearchTree<ItemType> & tree);

        bool isEmpty() const;
        int getHeight() const;
        int getHeightHelper(BinaryNode<ItemType> * subTreePtr) const;
        int getNumberOfNodes() const;

        bool add(const ItemType & newEntry);
        bool remove(const ItemType & target);
        void clear();

        bool contain(const ItemType & target) const;

        //Helper functions
        void preorder();
        void inorder();
        void postorder();

        //Traverse functions
        void preorderTraverse(BinaryNode<ItemType> *treePtr) const;
        void inorderTraverse(BinaryNode<ItemType> *treePtr) const;
        void postorderTraverse(BinaryNode<ItemType> *treePtr) const;

        BinarySearchTree<ItemType> & operator=(const BinarySearchTree<ItemType> &
 rightHandSide);
};

#include "BinarySearchTree.cpp"
#endif
```

File: BinarySearchTree.cpp

```
#include "BinarySearchTree.h"
#include <algorithm>

template<class ItemType>
BinarySearchTree<ItemType>::BinarySearchTree() : rootPtr(nullptr)
{}
template<class ItemType>
bool BinarySearchTree<ItemType>::add(const ItemType & newData)
{
    BinaryNode<ItemType> * newNodePtr = new BinaryNode<ItemType>(newData);
    rootPtr = placeNode(rootPtr, newNodePtr);
    return true;
}
template<class ItemType>
```

```
BinaryNode<ItemType> * BinarySearchTree<ItemType>::placeNode(BinaryNode<ItemType>
 *subTreePtr, BinaryNode<ItemType> * newNodePtr)
{
    BinaryNode<ItemType> * tempPtr;
    if (subTreePtr == nullptr)
        return newNodePtr;
    else if (subTreePtr->getItem() > newNodePtr->getItem())
    {
        tempPtr = placeNode(subTreePtr->getLeftChildPtr(), newNodePtr);
        subTreePtr->setLeftChildPtr(tempPtr);
    }
    else
    {
        tempPtr = placeNode(subTreePtr->getRightChildPtr(), newNodePtr);
        subTreePtr->setRightChildPtr(tempPtr);
    }
    return subTreePtr;
}
template<class ItemType>
BinaryNode<ItemType> * BinarySearchTree<ItemType>::removeValue(BinaryNode<ItemTyp
e> * subTreePtr, const ItemType & target, bool & isSuccessful)
{
    BinaryNode<ItemType> * tempPtr;
    if(subTreePtr == nullptr)
    {
        isSuccessful = false;
    }
    else if (subTreePtr->getItem() == target)
    {
        subTreePtr = removeNode(subTreePtr);
        isSuccessful = true;
    }
    else if (subTreePtr->getItem() > target)
    {
        tempPtr = removeValue(subTreePtr-
>getLeftChildPtr(), target, isSuccessful);
        subTreePtr->setLeftChildPtr(tempPtr);
    }
    else
    {
        tempPtr = removeValue(subTreePtr-
>getRightChildPtr(), target, isSuccessful);
        subTreePtr->setRightChildPtr(tempPtr);
    }
    return subTreePtr;
}
template<class ItemType>
void BinarySearchTree<ItemType>::preorder()
```

```
{
    std::cout <<"_____________________________"<< std::endl;
    std::cout << "TRAVERSING IN PREORDER" << std::endl;
    std::cout <<"_____________________________\n"<< std::endl;
    preorderTraverse(rootPtr);
    std::cout << std::endl;
}
template<class ItemType>
void BinarySearchTree<ItemType>::inorder()
{
    std::cout <<"\n_____________________________"<< std::endl;
    std::cout << "TRAVERSING IN INORDER" << std::endl;
    std::cout <<"_____________________________\n"<< std::endl;
    inorderTraverse(rootPtr);
    std::cout << std::endl;
}
template<class ItemType>
void BinarySearchTree<ItemType>::postorder()
{
    std::cout <<"\n_____________________________"<< std::endl;
    std::cout << "TRAVERSING IN POSTORDER" << std::endl;
    std::cout <<"_____________________________\n"<< std::endl;
    postorderTraverse(rootPtr);
    std::cout << std::endl;
}
//Traverse functions
template<class ItemType>
void BinarySearchTree<ItemType>::preorderTraverse(BinaryNode<ItemType> *treePtr)
const
{
    if (treePtr == nullptr)
    {
        return;
    }
    std::cout << treePtr->getItem() << " ";
    preorderTraverse(treePtr->getLeftChildPtr());
    preorderTraverse(treePtr->getRightChildPtr());
}
template<class ItemType>
void BinarySearchTree<ItemType>::inorderTraverse(BinaryNode<ItemType> *treePtr) c
onst
{
    if(treePtr == nullptr)
    {
        return;
    }
    inorderTraverse(treePtr->getLeftChildPtr());
    std::cout << treePtr->getItem() << " ";
```

```
    inorderTraverse(treePtr->getRightChildPtr());
}
template<class ItemType>
void BinarySearchTree<ItemType>::postorderTraverse(BinaryNode<ItemType> *treePtr)
 const
{
    if(treePtr == nullptr)
    {
        return;
    }
    postorderTraverse(treePtr->getLeftChildPtr());
    postorderTraverse(treePtr->getRightChildPtr());
    std::cout << treePtr->getItem() << " ";
}
template<class ItemType>
int BinarySearchTree<ItemType>::getHeight() const
{
    return getHeightHelper(rootPtr);
}
template<class ItemType>
int BinarySearchTree<ItemType>::getHeightHelper(BinaryNode<ItemType> * subTreePtr
) const
{
    if(subTreePtr == nullptr)
    {
        return 0;
    }
    else
    {
        return 1 + std::max(getHeightHelper(subTreePtr-
>getLeftChildPtr()), getHeightHelper(subTreePtr->getRightChildPtr()));
    }
}
```

File: main.cpp

```
// Project 4 || CS 302
// Originally considered in Fall 2019

//For the nodes of the search tree
#include "BinaryNode.h"
//For the operations on the search tree
#include "BinarySearchTree.h"
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <time.h>
```

```
int genRand();
int NUM_SIZE = 200;

int main()
{
    std::ifstream inFile;
    std::ofstream outFile;

    BinarySearchTree<int> bst;

    int randomNum;
    srand(time(NULL));

    inFile.open("hundred_nums.txt");
    int i = 0;
    while(inFile && i < NUM_SIZE)
    {
        inFile >> randomNum;
        i++;
        bst.add(randomNum);
    }

    std::cout << "Height is: " << std::endl;
    std::cout << bst.getHeight() << std::endl;
    std::cout <<"\n\n"<< std::endl;
    bst.preorder();
    bst.inorder();
    bst.postorder();

    return 0;
}

int genRand()
{
    return rand() % NUM_SIZE +1;
}
```

Then considering a file with 100 numbers as follows:

```
54
22
51
63
75
85
23
64
45
23
24
32
51
63
67
85
23
```

14
25
13
11
62
34
36
19
10
51
63
67
85
75
85
23
64
45
23
24
32
51
63
67
85
23
14
25
13
11
62
34
39
12
56
89
78
91
97
67
43
75
79
32
34
12
14
15
17
18
19
10
14
63
67
85
23
14
25
13
11
62
34
39
12
56
89
78
91
97
67
43
75
67
85
23
14
25
13
11

We then derive the following results for the inorder, preorder, postorder traversals:

```
Height is:
13


_____________________________________
TRAVERSING IN PREORDER

_____________________________________
54 22 14 13 11 10 10 11 12 11 11 11 12 12 13 13 13 19 14 14 15 14 14 14
17 18 19 51 23 45 23 24 23 23 23 23 23 23 32 25 24 25 25 25 34 32 32 36
34 34 34 39 43 39 43 45 51 51 51 63 62 56 56 62 62 75 64 63 63 63 63 67
64 67 67 67 67 67 67 85 75 78 75 75 79 78 85 85 85 85 89 85 85 91 89 97
91 97


_____________________________________
TRAVERSING IN INORDER

_____________________________________
10 10 11 11 11 11 11 12 12 12 13 13 13 13 14 14 14 14 14 14 15 17 18 19
19 22 23 23 23 23 23 23 23 23 24 24 25 25 25 25 32 32 32 34 34 34 34 36
39 39 43 43 45 45 51 51 51 51 54 56 56 62 62 62 63 63 63 63 63 64 64 67
67 67 67 67 67 67 75 75 75 75 78 78 79 85 85 85 85 85 85 85 89 89 91 91
97 97


_____________________________________
TRAVERSING IN POSTORDER

_____________________________________
10 10 11 11 11 12 12 12 11 11 13 13 13 13 14 14 14 18 17 15 14 14 19 19
14 23 23 23 23 23 23 24 25 25 25 25 32 32 34 34 34 39 43 43 39 36 34 32
24 23 45 45 23 51 51 51 51 22 56 56 62 62 62 63 63 63 63 64 67 67 67 67
67 67 67 64 75 75 78 79 78 75 85 85 89 91 97 97 91 89 85 85 85 85 85 75
63 54
```

Comment on Computational Complexity: The computational cost of all traversals is O(n), where n is the number of nodes, as we have to visit all the nodes of tree. The following holds for the average and worst-case computational complexity of BST operations:

Operation	Average	Worst-Case
Space	O(n)	O(n)
Search	O(logn)	O(n)
Insertion	O(logn)	O(n)
Deletion	O(logn)	O(n)

Follow-up questions:

- Derive the computational complexity for all the above operations.
  Present a case where the worst-case of the search, insertion and deletion operation would occur.