CSCI 2720: Data Structures

Project 4

Due: Friday, April 9, 2021 @11:59:59 PM

The goal of this project is to give you an opportunity to (1) practice implementing and using binary search trees. (2) practice using other data structures as building blocks of the tree structure, (3) writing recursive functions, and (4) continue practicing good software engineering approach through applying object-oriented concepts in your design and implementation.

1. Introduction

The specification of the binary search tree (BST) ADT is on the last page of this document.

We suggest incremental development and incremental integration and testing.

1- **Read lecture slides or watch the lecture recordings**. Many functions are explained!
2- Run the attached TreeDr.cpp.
3- Implement one function from the required functions, test and debug until it works correctly.
4- Repeat step 3 until all required functions are completed.

Tips:

Try with only few commands first, then add more commands as you progress.
Always have the Quit command at the end of your input file.

2. Project Requirements:

1. **[20 Points]** Implement the member function **levelOrderPrint** that prints the tree level by level. Suppose you have a tree that looks like this:

```
          15
         /   \
        6      20
       / \     /
      4   10  16
```

The output of **levelOrderPrint** should be:

15 6 20 4 10 16

You may need to research on the algorithm of level order traversal since this was not discussed in class. You may use any auxiliary data structure and helper functions to implement **levelOrderPrint.**

2. **[10 Points]** Implement the two public member functions of TreeType: **preOrderPrint and postOrderPrint**, to print the tree elements on screen in preorder and post-order, respectively. You must implement these functions recursively. Items should be displayed on a single line separated by spaces.

   Note: The print() function implemented in TreeType.cpp is an inorder traversal function.

3. **[10 Points]** Implement the private helper function **getSuccessor** that finds the node with the smallest key value in a tree, and returns a pointer to that node. This function is to be used in the **deleteNode** function.

4. **[10 Points]** Modify the **deleteNode** function so that it uses the immediate successor (rather than the predecessor) of the value to be deleted. In the case of deleting a node with two children, deleteNode() should now call the **getSuccessor** function.

5. **[15 Points]** Implemented the **printAncestors** function that prints the ancestors of a given node whose info member contains **value**. You may need to do some research on how to implement this function since it was not explicitly discussed during class.

   If the command: **PrintAncestorsOf 10** was issued on the sample tree used in the description of levelOrderPrint, the program should output 6 15. **PrintAncestorsOf 7** will display an error message "7 is not in the tree". **PrintAncestorsOf 15** will print "15 has no ancestors".

6. **[15 Points]** Implement the function **mirror** that creates a mirror image of the original tree.

   Original Tree

   ```
          15
         /  \
        6    20
       / \   /
      4  10 16
   ```

   Mirror Image

   ```
         15
        /  \
       20   6
        \   / \
        16 10  4
   ```

   This function is called by the public member function **mirrorImage**.

7. **[15 Points]** In TreeDr.cpp, implement the **makeTree()** function that creates a binary search tree from the elements in a sorted array of integers. Input parameter to this function is a sorted array and its size. (read the sorted array from the text file input.txt)

   Note: You cannot traverse the list inserting the elements in order, as that would produce a tree of height N-1. You must create a tree of height $\lfloor \log_2 N \rfloor$ (the BST must be as balanced as possible).

   TreeType* makeTree(int [], int size)

   Sorted list

   [4 6 10 15 16 20 30]

   makeTree

   ```
          15
         /  \
        6    20
       / \   / \
      4  10 16 30
   ```

The input file has the command **MakeTree** followed by an integer *N* that represents the number of elements in the array and *N* integers that represent the array elements.

For the above example, to make a balanced tree, the input file will have this command line:

MakeTree 7 4 6 10 15 16 20 30

8. **[5 points]** Practice good programming style and apply efficient implementation with respect to space and time complexity. Make sure that each function you implement is well documented. Your documentation should specify the return type, input parameters, the pre and post-conditions of all functions.

We will test your implementation using the attached TreeDr.cpp. This program already includes all the required commands. This program must read all input from a text file called input.txt.
We will test your program with our input file. A sample input file is attached. Grading is based on passing our test cases. Run your program on a variety of inputs, ensuring that all error conditions are handled correctly.

3. Submission Instructions:

Name your project directory **project4**. Submit your project4 directory to **csci-2720c** on odin (use the following command: *submit project4 csci-2720c*).

If you are off campus, you must use UGA VPN (remote.uga.edu) before you can ssh into *odin.cs.uga.edu* using your MyID and password.

Your project4 directory should contain the following files:

1. TreeType.h and TreeType.cpp.
2. TreeDr.cpp.
3. The text file input.txt
4. README file to tell us how to compile and execute your program. **Include your name and UGA ID# on the top of this file.**
5. The makefile containing at least the *compile*, *run*, and *clean* directives (read the syllabus for details)

Basic grading criteria:

1. If the project passed all test cases defined in our commands files 100%
2. If the project did not compile on *odin* 0%
3. If the project compiled but did not run 0% - 30%
   30% is given if all required files were submitted and program code completely satisfies all functional requirements.
4. If the project runs with wrong output for some test cases, test case rubrics will apply

See course syllabus for late submission evaluation

Binary Search Tree Specification

Structure:	The placement of each element in the binary tree must satisfy the binary search property: The value of the key of an element is greater than the value of the key of every element in its left subtree, and less than the value of the key of every element in its right subtree.
Operations (provided by TreeADT):	
Assumption:	Before any call is made to a tree operation, the tree has been declared and a constructor has been invoked.
makeEmpty	
Function:	Set the tree to empty state.
Postcondition:	Tree exists and is empty.
bool isEmpty	
Function:	Determines whether tree is empty.
Postcondition:	Function value = (tree is empty).
int getLength	
Function:	Determines the number of elements in tree.
Postcondition:	Function value = number of elements in tree.
putItem(ItemType item)	
Function:	Adds item to tree.
Precondition:	Item is not in tree.
Postconditions:	Binary search property maintained.
deleteItem(ItemType item)	
Function:	Deletes the element whose key matches item's key.
Preconditions:	Key member of item is initialized. One and only one element in tree has a key matching item's key.
Postcondition:	No element in tree has a key matching item's key.
print()	
Function:	Prints the values in the tree in ascending key order
Precondition:	Tree has been initialized
Postconditions:	Items in the tree have been printed in ascending key order. Tree is displayed on screen