CSCI 2720: Data Structures

Project 4

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.
There are 6 files attaches to this project document, including the README.txt.

Notes:

1- TreeDr.cpp is the driver program. **We will use it to test your program**. Complete the code of the makeTree function defined in this file.
2- The Queue implementation is attached to this project, but you may need to convert to a template in order to implement the required functions.
3- You need to add exceptions and their handlers. No restrictions, use the best way to implement exception handling.
4- You can use your own queue implementation, instead of the attached one. For example, you can use your circular array queue that you have implemented in assignment 1. I've tested the attached QueueType and it is working perfectly!

2. Suggested Solution Steps:

We suggest incremental development and incremental integration and testing.

1- **Read lecture notes or watch the online lectures**. Many functions are explained!
2- Run the attached TreeDr.cpp.
3- Implement one function from the required functions, test and fix, until works correctly.
4- Repeat step 3 until completing all required functions.
5- If worked correctly, change the implementation to a template and test again.

Tips:

Try with few commands, add commands as you progress.
Apply equivalence partitioning, marginal tests, test with valid an invalid test cases, e.g. Test deleting from an empty tree, deleting a non existing item, valid command, invalid command,..
Always have the Quit command at the end of your input file.

3. Project Requirements:

1. **[20 Points]** Add a non-recursive member functions: **LevelOrderPrint()** That prints the tree level by level. Your output should look as follows: -

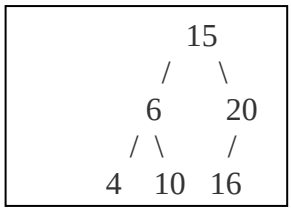


Figure 1

This function will be called when you execute the command **LevelOrderPrint**. You may use a data structure and any number of helper functions.

You may print your tree horizontally as well.

2. **[10 Points]** Add the two public member functions to TreeType: **preOrderPrint() and postOrderPrint()**, to print the tree elements on screen in preorder and post-order, respectively. Use recursive helper functions and do not define any data structure. Items should be displayed on a single line separated by spaces.

   Note: The Print function implemented in TreeType.cpp is an In-Order traversal function.

3. **[10 Points]** Add a private function **ptrToSuccessor** that finds the node with the smallest key value in a tree, and returns a pointer to that node.
   TreeNode* ptrToSuccessor(TreeNode *& tree)

4. **[10 points]** Add the public member function int **getSuccessor(int item)**
   This function returns the logical successor of item if item was in the list. If item is not in the list, the function throws an exception. This function must call **ptrToSuccessor** defined above.

   The driver should handle the command **GetSuccessor item.** For example, If the input file has the command: **GetSuccessor 6**, issued on the tree in Figure 1, the program must display the output 10. While **GetSuccessor 7,** will display an error message, like: "Item is not in the tree".
   getSuccessor (4) will return NULL.

5. **[5 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 call **ptrToSuccessor()** function directly or indirectly.

6. **[10 Points]** Add to **TreeType the *iterative*** public member function **printAncestors** that **prints** the ancestors of a given node whose info member contains **value**. Do not print value.

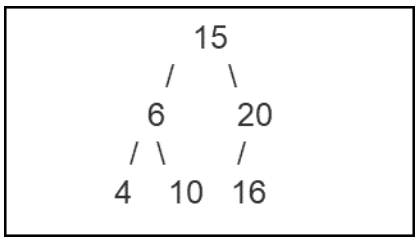
   //Precondition: Tree is initialized
   // Postcondition: The ancestors of the node whose info member is value have been printed.

   void TreeType::printAncestors(int value)

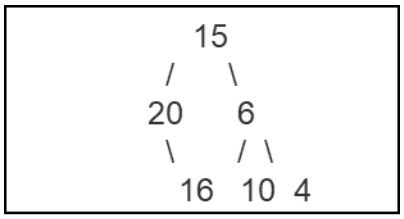
If the command: **PrintAncestorsOf 10** was issued on the tree in Figure 1, the program must display the output 6 15. **PrintAncestorsOf 7,** will display an error message, like: "Item is not in the tree. **PrintAncestors 15,** will print "15 has no ancestors".

7. **[15 Points]** Add to TreeType a public member function **mirrorImage** that creates and returns a mirror image of the tree.

Original Tree



Mirror Image



The function **mirrorImage** calls a private recursive function **mirror** that returns the mirror image of the original tree. See the prototypes in the attached tree.h and tree.cpp files.

```
TreeType TreeType::MirrorImage();
// Calls recursive function Mirror.
// Post: A tree that is the mirror image the tree is returned.
```

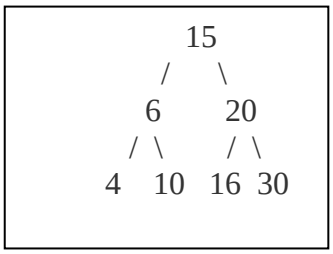
8. **[15 Points]** Write the **client (driver)** function **makeTree** that creates a binary search tree from the elements in a sorted array of integers. Input parameter to this function is a sorted array and its size. (read the sorted array from the text file input.txt)

Note: You cannot traverse the list inserting the elements in order, as that would produce a tree that has *N* levels. You must create a tree with at most $\log_2 N + 1$ levels.

TreeType& makeTree(int [], int size)

Sorted list

[4 6 10 15 16 20 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 6 4 10 15 16 20 30

9. **[5 points]** Practice good programming style and apply efficient implementation with respect to space and time efficiency. Make sure that each function is well documented (use comments only). Your

documentation should specify the type and function of the input parameters, output, and the pre and post-conditions for 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. Our input file will test for the following 15 commands: -

PutItem, DeleteItem, GetItem, GetLength, IsEmpty, IsFull, Print, PreOrderPrint, PostOrderPrint, LevelOrderPrint, MakeEmpty, GetSuccessorOf, PrintAncestorsOf, MirrorImage, and MakeTree

Note that commands capitalize each word and correspond to method names.
MakeTree will be the last command to be tested.
If you do not have an implementation of any command or the command is invalid, the program should display the message "Undefined Command".

What to submit:

1. TreeType.h and TreeType.cpp files satisfying requirements 1 to 7 above**.**
2. TreeDr.cpp program.
3. The text file input.txt
4. QueType.h and QueType.cpp
5. A README.txt file telling us how to compile your program and how to execute it.
6. If you are printing the output to a text file, attach the output File output.txt

Submit your Project4 directory to **cs2720a** on nike.

No part of your code may be copied from any other source unless those attached to this assignment or noted on this document. All other code submitted must be original code written by you.

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 any element in its left subtree, and less than the value of the key of any 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 applied.

MakeEmpty

Function: Initializes tree to empty state.
Postcondition: Tree exists and is empty.

Boolean IsEmpty

Function: Determines whether tree is empty.
Postcondition: Function value = (tree is empty).

Boolean IsFull

Function: Determines whether tree is full.
Postcondition: Function value = (tree is full).

int GetLength

Function: Determines the number of elements in tree.
Postcondition: Function value = number of elements in tree.

int GetItem(ItemType item, Boolean& found)

Function: Retrieves item whose key matches item's key (if present).
Precondition: Key member of item is initialized.
Postconditions: If there is an element someItem whose key matches item's key, then found = true and a copy of someItem is returned; otherwise, found = false and item is returned. Tree is unchanged.

PutItem(ItemType item)

Function: Adds item to tree.

Preconditions:

Tree is not full.

Postconditions: item is in tree.

Binary search property is 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