CSCI 2720: Data Structures

Spring Semester, 2017

Project 3

Instructor: Dr. Eman Saleh

Due date

4/5/2017 at 11:30PM

The goal of this project is to give you an opportunity to practice implementing and using binary search trees. You will also practice writing recursive functions and continue practicing good software engineering approach through applying object-oriented concepts in your design and implementation.

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.
Tree is not full.

Preconditions:
item is not in tree.
item is in tree.

Postconditions:
Binary search property is maintained.

DeleteItem(ItemType item)

Function: Deletes the element whose key matches item's key.
Key member of item is initialized.

Preconditions:
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
Items in the tree have been printed in ascending key order.

Postconditions:
Tree is displayed on screen

ResetTree(OrderType order)

Function: Initializes current position for an iteration through the tree in OrderType order.

Postcondition: Current position is prior to root of tree.

ItemType GetNextItem(OrderType order, Boolean& finished)

Function: Gets the next element in tree.
Current position is defined.

Preconditions:
Element at current position is not last in tree.
Current position is one position beyond current position at entry to GetNextItem.

*Postconditions:*finished = (current position is last in tree).

A copy of element at current position is returned.

A specification and the implementation of TreeType and QueueType is attached with this assignment. Note that TreeType defines and uses three queues (See README.txt for more information about queues usage.)

You can modify the implementation or write your own code.

Project Requirements:

1. **[10 Points]** Redefine TreeType as a generic data type, i.e **Define a class template** instead of using **typedef** in statement 3 in the file **TreeType.h**

   ```
   template<class ItemType>
   class TreeType { ... }
   ```

2. **[10 Points]** Add the a non-recursive member functions: **LevelOrderPrint()** That prints the tree level by level. Your output should look as follows

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

   Figure 1

3. **[15 Points]** Add the three public member functions to TreeType: **PreOrderPrint(), InOrderPrint() and PostOrderPrint()**, to print the tree on screen in preorder, in-order 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.

4. **[15 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.

   ```
   template<class ItemType>
   TreeNode<ItemType>* PtrToSuccessor(TreeNode<ItemType>*& tree)
   ```

   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. You should call **PtrToSuccessor()** function that you defined in part 3.

5. **[10 Points]** Add to **TreeType the *iterative*** member function **Ancestors** 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<ItemType>::Ancestors(ItemType value)
   ```

6. [15 Points] Add to TreeType a public member function **MirrorImage** that creates and returns a mirror image of the tree.

Original Tree

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

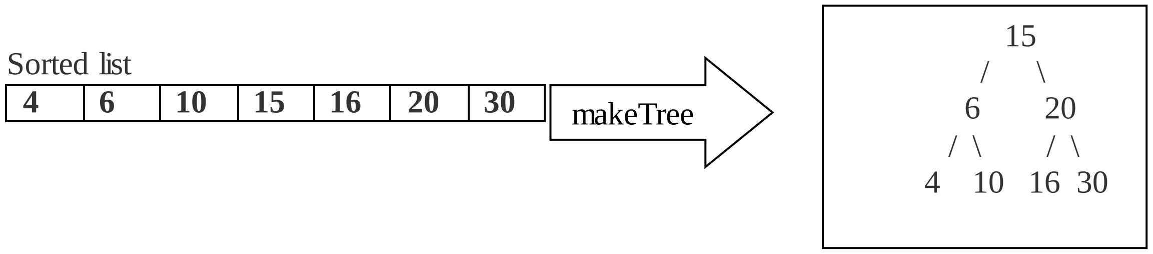
Mirror Image

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

The function **MirrorImage** calls a recursive function **Mirror** that returns the mirror image of the original tree as follows: -

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

7. [15 Points] Write a **client** function **MakeTree** that creates a binary search tree from the elements in a sorted list of integers. Input parameter to this function is a sorted array. (read the sorted array from the text file input.txt see part 8). 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.



8. [10 Points] Test your implementation using TreeDr.cpp. This program must display a menu for all operations (you can use and modify the attached file TreeDr.cpp). This program must read all input from a text file called **input.txt** and display output on screen.

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 paramaters, ouput and pre and post-conditions for all functions.

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. The attached QueueType.h and QueueType.cpp
5. A `README.txt` file telling us how to compile your program and how to execute it.
   As you will work with a partner, print your names and class CRN at the top of this file.

Submit your Project2 directory to **cs2720b** if you are in CRN 26490), **or** to **cs2720c** if you are in CRN 26486) on nike.

Do not modify the files QueueType.h and QueueType.cpp
Run your program on a variety of inputs ensuring that all error conditions are handled correctly.

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