CSCI 2720: Data Structures

Fall Semester, 2016

Project 2

Instructor: Dr. Eman Saleh

Due date 10/26/2016 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, for example you may not use queues.

Project Requirements:

1. **[10 Points]** Add three member functions: **preOrderPrint(), inOrderPrint() and postOrderPrint()** to TreeType to print the tree on screen by preorder traversing, in-order traversing and post-order traversing respectively.

2. **[15 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. You should call the function **`PtrToSuccessor`** that finds a node with the smallest key value in a tree, unlinks (deletes) it from the tree, and returns a pointer to the unlinked node.

```
template<class ItemType>
NodeType<ItemType>* PtrToSuccessor(TreeNode<ItemType>*& tree)
// Sets data to the info member of the left-most node in tree
// which is the successor tree.

template<class ItemType>
void DeleteNode(TreeNode<ItemType>*& tree)
// Deletes the node pointed to by tree.
// Post: The user's data in the node pointed to by
//      tree is no longer in the tree.  If tree is a leaf
//      node or has only non-NULL child pointer
//      the node pointed to by tree is deleted; otherwise,
//      the user's data is replaced by its logical successor
//      and the successor’s node is deleted.
```

3. **[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`.

```
void Ancestors(ItemType value);            // prototype
//Precondition: Tree is initialized
// Postcondition: The ancestors of the node whose info member is value
//     have been printed.

void TreeType<ItemType>::Ancestors(ItemType value)
```

4. **[15 Points]** Add to `TreeType` a 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
```

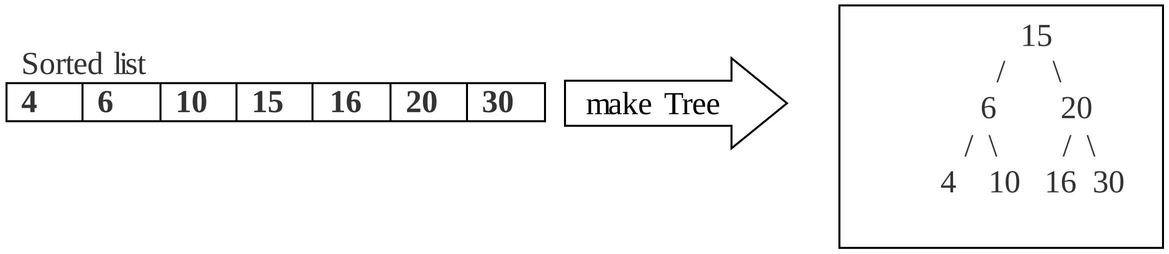
The function **`MirrorImage`** might call a recursive function **`Mirror`** that returns the mirror image of the original tree as follows: -

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

{
  TreeType image;

  Mirror(root, image.root);
  return image;
}
```

5. **[15 Points]** Write a <u>**client**</u> function **`MakeTree`** that creates a binary search tree from the elements in a sorted list of integers (input is sorted array). 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.



6. **[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 { ... }
```

7. **[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 paramaters, ouput and pre and post-conditions for all functions`

   In the README.txt file specify what makes your implementation more efficient compared to the implementation in the attached files.

What to submit:

1. The TreeType.h and TreeType.cpp files satisfying requirements 1 to 7 above. **[80 Points]**
   If your implementation used QueueType then submit QueueType.h and QueueTupe.cpp
   If your implementation used input or output files, then submit these files.

2. A README.txt file telling us how to compile your program and how to execute it. **[10 Points]**

   This also explains the usage of queues if used by the implementation.

3. A test driver TreeTest.cpp to test your program **[10 Points]**
   This program must display a menu for all operations (you can use and modify the attached file TreeDr.cpp)

4. Submit your ***Project2*** directory to **cs2720b** (if you are in CRN 30839) **or** to **cs2720c (**if you are in CRN 25654) on nike. You must use the submit command to submit your work, as shown below:
   $ submit Project2 cs2720b
   Or
   $ submit Project2 cs2720c

   Important: You should execute the submit command while being in the parent directory of Project2

5. Run your program on a variety of inputs ensuring that all error conditions are handled correctly.
6. **Reminder:** *No* part of your code may be copied from any other source unless noted on this assignment. All other code submitted must be original code written by you.

Evaluation of the projects will include:

1) evaluating test cases using a pass or fail metric

   Basic grading criteria:

   1. If the project did not compile 0%
   2. 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.
   3. If the project run with wrong output 0% - 60%
      Depending on the solution, 50-60% is given ONLY if the cause of the error was minor after checking all functional aspects.

See course syllabus for late submission evaluation

Preparation:

You must include the following comment at the top of the program file. Copy and agree to the entirety of the text below, and fill in the class name of your .cpp or .h file, your name, submission date, and the program's purpose.

```
/*

 [File name here].cpp

 Author: [Your name here]

Submission Date: [Submission date here]

 Purpose: A brief paragraph description of the program. What does it do?

Statement of Academic Honesty:

The following code represents my own work. I have neither received nor given inappropriate assistance. I have not copied or modified code from any source other than the course webpage or the course textbook. I recognize that any unauthorized assistance or plagiarism will be handled in accordance with the University of Georgia's Academic Honesty Policy and the policies of this course. I recognize that my work is based on an assignment created by the Department of Computer Science at the University of Georgia. Any publishing or posting of source code for this project is strictly prohibited unless you have written consent from the Department of Computer Science at the University of Georgia.

*/
```

Every file of every project you submit must have a comment such as this.

Otherwise, points will be deducted from your project.

For any questions or clarifications please ask me during lecture, or visit me or the TAs during office hours, or send me an email at eman.saleh@uga.edu from <u>your UGA email</u>