COMP 2611 – Data Structures
2018/2019 Semester 3
Assignment #2

Date Due: 19th June, 2019 @ 11:55p.m.

Description

This assignment requires you to write several functions for a binary tree and a binary search tree. You also have to write a menu-based application to test these functions. The application reads the words from a passage and stores them in a binary search tree.

You are supplied with a Dev C++ project containing various `.h` and `.cpp` files. You do not have to modify the project. The code for this assignment must be written in the `BinaryTree.cpp`, `BinarySearchTree.cpp`, and `Main.cpp` files. There is no need to modify the code in the other files supplied.

Structure Definition of a Node in a Binary Tree (Defined in NodeTypes.h)

The functions that have to be written for this assignment assume that a node in the binary tree is defined as follows:

```
struct BTNode {
      string data;
      BTNode * left;
      BTNode * right;
      BTNode * parent;
};
```

It should be observed that each node stores the address of its parent node. Every node in a binary tree except the root node has a parent. The address of the parent of the root node is NULL.

It should also be observed that a node in a binary tree is defined in the same way as a node in a binary search tree. So, all the functions defined for a binary tree can be used unchanged with a binary search tree.

To create a new node for a binary tree, the following code can be used (instead of using `malloc`):

```
BTNode * newNode = new BTNode;
```

Binary Tree Functions (to be written in BinaryTree.cpp)

Return Type	Prototype of Function and Description
int	**height**(BTNode * root) Returns the height of the binary tree. The height of a binary tree is 1 + the highest level of a node in the tree. The code must be non-recursive.
void	**inOrderIterative(**BTNode * root) Outputs the in-order traversal of the binary tree. The code must be non-recursive.
bool	**isEmptyTree**(BTNode * root) Returns *true* if the binary tree is empty and *false*, otherwise.
bool	**isEqual(BTNode * root1, BTNode * root2)** Returns true if the two binary trees supplied as parameters are identical. Two binary trees are identical if they contain the same values in exactly the same positions in each tree.
void	**levelOrder(**BTNode * root) Outputs the level-order traversal of the binary tree. In a level-order traversal, the nodes are visited one level at a time, from left to right. The code must be non-recursive.
int	**moment**(BTNode * root) Returns the moment of the binary tree. The moment of a binary tree is the amount of nodes in the binary tree.
void	**preOrderIterative (**BTNode * root) Outputs the pre-order traversal of the binary tree. The code must be non-recursive.
void	**postOrderIterative (**BTNode * root) Outputs the post-order traversal of the binary tree. The code must be non-recursive. The Programming Guidelines section explains how to perform a post-order traversal using two stacks.
int	**weight**(BTNode * root) Returns the weight of the binary tree. The weight of a binary tree is the amount of leaves in the tree.
int	**width(**BTNode * root) Returns the width of the binary tree. The width of a binary tree is the maximum number of nodes at any level of the tree. The Programming Guidelines section explains how to find the nodes in a given level.

Binary Search Tree Functions (to be written in BinarySearchTree.cpp)

Return Type	Prototype of Function and Description

`BTNode *`	**`ceiling`**`(BTNode * root,` **`string`** `key)`
	
	`Returns the address of the node with the smallest key in the BST that is greater than or equal to` *`key`*`. Returns NULL if there is no such node.`

`BTNode *`	**`contains`**`(BTNode * root,` **`string`** `key)`
	
	`Returns the address of the node containing` *`key`*`. Returns NULL if` *`key`* `does not exist.`

`BTNode *`	**`deleteNode`**`(BTNode * root,` **`string`** `key)`
	
	`Removes the node containing the specified key (if` *`key`* `is present in the BST). Returns the address of the root of the tree without the deleted node.`

`BTNode *`	**`deleteMin`**`(BTNode * root)`
	
	`Removes the node containing the smallest key in the BST. Returns the address of the root of the tree without the deleted node.`

`BTNode *`	**`deleteMax`**`(BTNode * root)`
	
	`Removes the node containing the largest key in the BST. Returns the address of the root of the tree without the deleted node.`

`BTNode *`	**`floor`**`(BTNode * root,` **`string`** `key)`
	
	`Returns the address of the node with the biggest key in the BST that is less than or equal to` *`key`*`. Returns NULL if there is no such node.`

`BTNode *`	**`inOrderPredecessor (BTNode * node)`**
	
	`Returns the in-order predecessor of the node supplied as a parameter. Returns NULL if the node is NULL or if there is no in-order predecessor of the node.`

`BTNode *`	**`inOrderSuccessor (BTNode * node)`**
	
	`Returns the in-order successor of the node supplied as a parameter. Returns NULL if the node is NULL or there is no in-order successor of the node.`

`BTNode *`	**`insert(`**`BTNode * root, string key)`
	
	`Inserts the specified key into the BST (if` *`key`* `is not already present). Returns the address of the root of the tree containing the newly inserted key.`

`BTNode *`	**`max`**`(BTNode * root)`
	
	`Returns the address of the node containing the largest key in the BST.`

`BTNode *`	**`min`**`(BTNode * root)`
	
	`Returns the address of the node containing the smallest key in the BST.`

Application (To be written in Main.cpp)

You are required to write an application which reads the words from a passage and stores them in a binary search tree. The application provides a menu from which various operations are performed. The operations are performed by calling one or more of the binary tree or binary search tree functions written in the two previous sections. The following is the menu that must be displayed:

```
Binary Search Tree (BST)
---------------------------------------------

1.   Create BST from Passage
2.   Add Word to BST
3.   Delete Word from BST
4.   Delete Smallest Word from BST
5.   Delete Largest Word from BST
6.   Search for Word in BST
7.   Traverse BST
8.   What Comes Before Word in BST?
9.   What Comes After Word in BST?
10.  Compare BSTs
11.  Statistics
Q.   Quit

Please enter an option:
```

When an option is selected, the appropriate action must be taken, after which the menu is re-displayed. The following is a description of each option.

1. When this option is selected, the program should allow the user to specify the name of the file containing the passage.
   `Please enter the name of file containing the passage or M (Menu):`
   If the user enters "M" control should return to the main menu and no BST should be created; if a BST was previously created, this will continue to be the "active" or "current" BST. Otherwise, your program should read the appropriate file and create the BST.

2. When this option is selected, the program should request the user to specify a word.
   `Please enter the word to insert in the BST or M (Menu):`
   If the user chooses "M", control should return to the main menu. Otherwise, your program should insert the word into the BST. The BST should not contain duplicate words.

3. When this option is selected, the program should request the user to specify the word to be deleted.
   `Please enter the word to delete from the BST or M (Menu):`
   If the user chooses "M", control should return to the main menu. Otherwise, your program should delete the word from the BST, if it is present.

4. When this option is selected, the program deletes the smallest key from the BST.
   Control should then return to the main menu.

5. When this option is selected, the program deletes the largest key from the BST.
   Control should then return to the main menu.

6. When this option is selected, the program should prompt for a word and determine if it is present in the BST.

```
            Please enter a word to search for or M (Menu):
```

If the user chooses "M", control should return to the main menu. If the word is not present, display an appropriate message.

7. When this option is selected, the program should display a pre-order traversal, an in-order traversal, a post-order traversal, and a level-order traversal of the BST using the traversal functions written in the previous sections.

8. When this option is selected, the program should prompt for a word and display the word if it is present in the BST.

```
                  Please enter a word to search for:
```

If it is not present, your program should display the word that comes immediately ***before*** it in the BST, assuming that it was present. If there is no such word, display an appropriate message. Control should then return to the main menu.

9. When this option is selected, the program should prompt for a word and display the word if it is present in the BST.

```
                  Please enter a word to search for:
```

If it is not present, your program should display the word that comes immediately ***after*** it in the BST, assuming that it was present. If there is no such word, display an appropriate message. Control should then return to the main menu.

10. When this option is selected, the program should allow the user to specify the name of a file from which a new BST will be created; this BST will be compared with the current BST to determine if they are identical. The program should open the file and use the contents of the file to create a BST.

```
Please enter the name of the file to create the BST or M (Menu):
```

If the user chooses "M", control should return to the main menu. Otherwise, your program should read the data from the file, create the second BST, and check if it is identical to the current BST. Note that the current BST can only be created via Option 1.

If the BSTs are not equal, your program should display:
(1) a list of words that are common to both BSTs
(2) the number of words that are common to both BSTs
(3) if the "current" BST is heavier than the other BST

11. When this option is selected, the program should display the following information about the BST:

(1) The number of nodes in the tree.
(2) The height of the tree.
(3) The width of the tree.
(4) The weight of the tree.
(5) The smallest word and the biggest word (in alphabetical order) stored in the tree.

Files Required (Available at Course Web Site in myElearning)

File
`NodeTypes.h`
`Stack.h`
`Stack.cpp`
`Queue.h`
`Queue.cpp`
`BinaryTree.h`
`BinaryTree.cpp`
`BinarySearchTree.h`
`BinarySearchTree.cpp`
`Main.cpp`
`Assignment2.dev`

All the files are part of the *Assignment2* project (stored in `Assignment2.dev`).

Programming Guidelines

Storing the Address of a Parent Node

Parent pointers are very useful when finding the in-order successor or the in-order predecessor of a node. The course textbook explains how to store the address of a parent node. Each node in the binary search tree except the root node stores the address of its parent node. The address of the parent node of the root node is NULL. The address of the parent node should be stored when a node is about to be inserted in the binary search tree. However, care must be taken to adjust the addresses of parent nodes when a node is deleted.

Post-Order Traversal of Binary Tree Using Non-Recursive Algorithm

The post-order traversal of a binary tree visits the nodes in the opposite order of a pre-order traversal, except that the nodes in a particular level are visited from right to left. A stack can be used to reverse the order of the "modified" pre-order traversal. Another stack can be used to visit the nodes from right to left instead of from left to right.
The algorithm is similar to a level-order traversal except that a stack is used instead of a queue. Also, at the end of the level-order traversal, the nodes are in the opposite order to what they should be. The correct ordering of the nodes is obtained by using a second stack. Here is the algorithm:

```
store the root node in stack1
while stack1 is not empty  {
      pop stack1 and store the node in temp
      push temp onto stack2
      push the left of temp (if it exists) onto stack1
      push the right of temp (if it exists) onto stack1
}
pop elements from stack2 and visit each one as it is popped
```

Finding the Nodes in Each Level of a Binary Tree

The amount of nodes in each level of a binary tree can be found by doing a level-order traversal which traverses the tree level by level. Recall that a level-order traversal uses a queue to store the elements as they are being taken off the binary tree. The problem is how to count the amount of nodes stored at each level. This can be achieved by a little modification of the level-order algorithm. Once the amount of nodes at a particular level is found, this can be compared to the maximum value found at the previous nodes.