Homework #3: Empirical Study on the Average Height of Binary Search Trees

The height of the binary search tree:

Given a binary search tree (referred to as BST in the following), let's define the number of nodes on a longest path from the root to a node in the BST as the height of the BST. In the worst-case scenario, a BST of *n* nodes may essentially degenerate into a sorted linked list, ending in a BST of height *n*. However, on average the height of random BSTs of n nodes is $O(\log_2 n)$, i.e. no more than some constant times *log n* when *n* is big. The maximal amount of time for search, deletion, and insertion in a BST of n nodes is proportional to the height of the BST. Thus the average time for search, deletion, and insertion in a BST of n nodes is of the order of $O(\log_2 n)$. In this assignment, we would like to conduct experiments to see empirically how well the binary trees actually work on average.

Requirements:

Part I. Coding the height-related method and testing it extensively:

Based on the binary search tree class code and the C++ project you have for **Homework#2,** do the following implementation:

- **Method to recursively calculate the height of a given branch**: Add a new **private** member function, int *BranchHeight(TreeNode * ptrNode)*, into the binary search tree class you have for the previous programming assignment. This method should return the height of a given branch within the binary search tree rooted at the node pointed to by *ptrNode*. To implement this method, you should check whether ***ptrNode*** is NULL or not. If it is NULL, it is an empty branch and you return 0 as the height. Otherwise, you recursively call *BranchHeight(ptrNode->Left)* and *BranchHeight(ptrNode->Right)* to determine the heights of the two immediately branches, and return one plus the larger one of the heights of the two branches. Make sure you just call *BranchHeight(ptrNode-Left)* and *BranchHeight(ptrNode->Right)* once respectively and store the results in two local variables for later use to determine the tree height. If you call *BranchHeight(ptrNode-Left)* and *BranchHeight(ptrNode->Right)* more than once, your program will perform very slowly in the experiments in Part 2.

- **Method to calculate the height of a tree**: Add a new **public** member function, int *Height( )*, to the binary search tree class calculate the current height of the tree. For any concrete binary search tree object *myTree*, a member function call *myTree.Height( )* should return the current height of the tree. To implement this method, you can simply call *BranchHeight(Root)*, where *Root* is simply private data member of myTree pointing to the root of the underlying binary search tree.
- **Note on the Clear method**: This method delete all the tree nodes from the tree by calling *DeleteCompleteTree(Root)* recursively and set *Root* to *nullptr*. For any concrete binary search tree object *myTree*, a member function call *myTree.Clear( )* will free all the tree nodes used in the tree and make it an empty tree.
- **User menu and options for testing tree height and clearing the tree in the main function:** In the main function, declare a tree object *myTree* to store dates and then use a loop to repeated display a menu and prompt the user to choose one of the options: **(i)** an option ***T*** to enter a date and then insert it into the tree object *myTree*, **(ii)** an option to calculate and display the height of the tree object *myTree* by calling *myTree.Height( )*, and **(iii)** an option ***K*** to clear the tree to an empty tree. Your main function should serve the user according to the chosen option.
- **Extensive test to make sure the implementation is correct**: You should then extensively test the *Height* method to make sure it works. For example, after clearing the tree and then manually inserting the following series of dates *(1, 1, 2006), (1, 2, 2006), (1, 3, 2006), (1, 4, 2006), (1, 5, 2006), (1, 6, 2006), (1, 7 2006)* one by one into the binary search tree, the height should be 7. And after clearing the tree and then manually inserting the following series of dates *(1, 4, 2006), (1, 1, 2006), (1, 7, 2006), (1, 2, 2006), (1, 5, 2006), (1, 6, 2006), (1, 3 2006)* one by one into the binary search tree, the height should be 4.

Part II. Adding a new option *Z* into the menu of services in the main function: running experiments measuring the average height of random binary search trees:

First, declare an integer array *heightArray* of 1000 integers in the main function, i.e. "*int heightArray[1000];"* . We' ll use it to store the height information up to the 1000 random binary search trees. Enhance the menu provided by the main function by implementing the following additional option ***Z*** for running experiments measuring the average height of random binary search trees:

1. Ask the user the number of random dates to be inserted into each random binary search tree. Store it in an *int* variable ***n***. Set up a loop to through 1000 iterations. On iteration *i (i from 0 to 999)*, we'll randomly generate a binary search tree of about ***n*** nodes by **(i)** first calling *Clear ( )* first to empty the tree, say, *myTree*, **(ii)** set up an inner loop to go through ***n*** iterations and on each iteration randomly set and insert a random date into the *myTree*, **(iii)** call *myTree.Height( )* to calculate the height of the tree, and **(iv)** store the height information in the corresponding element of the *heightArray[i]*.
2. Calculate to report the average height of these 1000 trees according to the information now stored in *heightArray[i]* and store it in an integer variable *averageHeight*.
3. Calculate and report the estimated standard deviation from *averageHeight* according to the information now stored in *heightArray[i]* in the following way:
    a. Calculate $\sum$ (*heightArray*[i] – *averageHeight*)^2 over all *i* in the range of [0,1000-1]). In other words, calculate (*heightArray*[i] – *averageHeight*)^2 for each *i* in the range of 0 to 999 and add them up to find the sum of them.
    b. Store the sum found above into a ***double*** variable *sumOfSquaredErrors*.
    c. Divide *sumOfSquaredErrors* by 1000 to find the average value, and then take the square root of the average as the standard deviation. To do so, let's declare a *double* variable *deviation* to store the standard deviation and store the value *sqrt(sumOfSquaredErrors/1000)* into the variable *deviation.* Note that you need to call the *sqrt* function in

<cmath> to calculate the square root, and thus you should have "#include <cmath>" in the beginning of your program.

4. Calculate and report the percentage of the 1000 trees that have a height within the range of [*averageHeight – 2*deviation*, *averageHeight + 2*deviation*].
5. Calculate and report the percentage of the 1000 trees that have a height within the range of [*averageHeight – 3*deviation*, *averageHeight + 3*deviation*].

Part III. Run experiments and report the findings:

- **Run the option Z implemented in Part II above several times**: Let *n* vary from **8192 (i.e. 2^13), 16384 (i.e. 2^14), 32768 (i.e. 2^15), 65536 (i.e. 2^16), 131072 (i.e. 2^17), 262144 (i.e. 2^18), till 524288 (i.e. 2^19)** as far as possible. Note that this makes $log_2\ n$ vary accordingly from 14, 15, 16, 17, 18, till 19. Record in a WORD document the findings reported by your program. Note that if you implement *Height( )* correctly as described in Part I above, it should be able to finish at least the first few experiments efficiently. If it is extremely slow to finish the experiments, it is not quite right and you need to re-examine the descriptions in Part I and your implementation of *Height( )*.
- **Reflection**: Check and report whether the average height of the trees is always close to $c * log_2\ n$ for some fixed constant *c* (even when you use different values for *n*). In other words, find out whether the average height divided by $log_2\ n$ is close to some constant when *n* grows. If so, then your findings are consistent with the claim that on average the height of random BSTs of *n* nodes is $O(log_2\ n)$. In other words, is the average height bounded around or below some $c * log_2\ n$ when *n* grows bigger and bigger.

Submission of your work:

- Record all your experimental findings and reflection in **Part III** into a WORD document. Submit the WORD document under Canvas.
- Compress your entire Program folder into a zip file and upload it through Biola Canvas.
- Carefully fill out this self-evaluation report and upload it through Biola Canvas. Note that you will receive no point for missing the self-evaluation report or missing the integrity review in the report.