CSCI205 CS3 Data Structures and Algorithms
Lab Assignment
Binary Search (BST) and Adelson-Velskii, Landis (AVL) Tree Exercises

TASK ONE: Using a "professional" diagramming tool like https://app.diagrams.net/ draw the resulting BST structures for the following lists of integers. For each node include its **height** and **balanceFactor** as given by the following equations:

a. H = max(height(leftSubTree), height(rightSubTree))
b. Bf = height(leftSubTree) – height(rightSubTree)
c. Using the AVL balance factor set of {-1, 0, 1} label each tree as "balanced", "skewed left" or "skewed right"

Assume insertion begins at **list[0]**

1. Show the resulting BST for [13, 20, 18, 3, 13, 2, 9, 27, 17, 28]
2. Show the resulting BST for [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3. Show the resulting BST for [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
4. Show the resulting BST for [68, 88, 61, 89, 94, 50, 4, 76, 66, 82]
5. Include each tree on a single document. Label them clearly, showing the original input list followed by the tree. Export the diagram as a PDF or image type.

TASK TWO: For the following Binary Search Tree problems, use the BST implementation in this weeks repo. You may modify the class as you see fit, just be sure to include comments on what your modifications are along with a justification.

To keep things easy to manage with this lab, all exercises can be completed using integers.

1. Plot some experiments with Binary Search Tree efficiency

    a. **Average Case:** Create BSTs with some randomly generated integers of various sizes (N). Track the insertion cost. Plot
    b. **Worst Case:** Create BSTs with some inversely sorted integers of various sizes (N) Track the insertion cost. Also track the cost of flattening the tree to a list. Plot
    c. **Include both cases on one graph.** Include a readMe file that contains your conclusions and Big O analysis. Specifically mention the degeneration of the BST when out of balance.

2. Write the BST method **vector<T> flatten( )** that converts "this" BST to a vector of sorted type T. This provides yet another way to arrange items in a collection.

3. Write the function **int closest(target)** that finds the value closest to "target" in "this" BST.

4. Write the function **bool is_valid(BST)** that returns true if BST is a valid Binary Search Tree, false otherwise. Review the important BST properties. Understand that this does not have anything to do with balance.

5. Write the function **BST generate(vector<T>)** that creates a height balanced BST from the unordered items in array. Balance the tree as best you can. Experiment with randomly generated collections of integers. This exercise is not referring to an AVL tree. What I want you to do is:

    a. Given an unordered collection of integers, create the most balanced tree you can. Perhaps you'll need to sort the integers first and perform some sort of recursive divide and conquer to best determine the roots of the left and right subtrees.
    b. Include comments to describe your approach to this problem. Supply sufficient detail and include a detailed Big O analysis.

6. Write the BST method **int find_kth_smallest(k)** that finds and returns the k^{th} smallest number in "this" BST.

7. Write the BST method **int height(Node)** that returns the height of Node as defined by
    a. H = max(height(leftSubTree), height(rightSubTree))

8. Write the BST method **int balance_factor(Node)** that returns the "balance factor" of Node as defined by
    a. Bf = height(leftSubTree) – height(rightSubTree)

AVL Tree Characteristics

- If there are ***n*** nodes in an AVL tree, minimum height of AVL tree is ***floor($log_2 n$)*** and the maximum height can't exceed ***1.44*$log_2 n$***

- If height of AVL tree is ***h***, maximum number of nodes can be ***$2^{h+1} - 1$*** (perfect binary tree)

- Minimum number of nodes in a tree with height ***h*** can be represented as:

  N(h) = N(h-1) + N(h-2) + 1 for n>2 where N(0) = 1 and N(1) = 2

- The complexity of searching, inserting and deletion in AVL tree is ***O(log n)***

TASK THREE:

1. Implement the AVL Tree presented in Runestone. For the adventurous type write your own. You will be asked to perform some AVL rotations on the final exam so be sure to fully understand the methodology. Simply copy and pasting will not give that deep

understanding. There are a couple of approaches for AVL tree implementation

a. The Tree Node can store a **parent reference**. This will allow backtracking of the insertion path to perform tree balancing rotations. Utilizing this method will allow you to forgo the call stack consumption of a recursive approach but will require the added memory of the parent reference

b. You can use recursion. Employing a recursive approach to AVL tree insertion and deletion allows you to forgo the added memory reference to the parent, because the traversal path will be stored in the call stack. As you return from the insert or delete operation you can backtrack through the tree performing AVL rotations.

2. The Runestone implementation does not include **delete.** Define the delete behavior and re-balance the tree according to AVL rules.

3. Define the function **bool is_balanced( )** that will return true if "this" tree is AVL balanced, false if not. Use the balance factor set {-1, 0, 1} in your tests.

4. AVL tree's time complexity of searching, insertion and deletion is ***O(log n)***. A non-balancing BST may be skewed left or right, resulting in worst case BST searching, insertion and deletion complexity of ***O(n)***. Prove this by inserting a sorted list of integers into both a non-balance BST and an AVL tree. Run this experiment a number of times on increasing values of N. Graph the results and provide some comments on your conclusions in your readMe. Give us thoughtful details.

SUBMISSION: All code must be error and leak free. Include a readMe file with comments on analysis and any other important information the grader should know. Graphs should be included in an image format and be clearly labeled. BST diagrams should be included in some portable format and be clearly labeled. It is in your best interest to make this as easy as possible on the grader.