CSC 230 – Project #5

Due: Friday, May 5th, before midnight

Project Details:

In this project, you will implement AVL tree, which is used to organize the SSN and Name information. We use AVL tree to fulfill the functionality that are implemented in project 2, 3, and 4.

So far, we have used vector (project 1), (dynamic) array (project 2), linked list (project 3), and hash table (project 4) to store data in our projects. After using these data structures, you should have a good understanding of the pros and cons of each data structures. In this project, we will implement AVL tree, the most complicated data structure we introduced in this course so far. Several things you should know about AVL tree:

- AVL tree is binary search tree (BST). The data in BST are ordered.
- AVL tree is somehow balanced, which means left subtree and right subtree have similar size.
- Searching in AVL tree is the same as the searching in any BST.
- One insertion may cause some node between the inserted node ant the root out of balance, you need to find the node and rebalance it. For one insertion, you need at most one rebalance.
- One deletion may cause some nodes between "the actual" deleted node and the root out of balance, you need to find the node and rebalance it. The difference between insertion and deletion is that deletion may need multiple rebalance.
- The time complexity of searching, insertion, and deletion are O(log n) for AVL tree for both average case and worst case. This performance is superior to the other data structures we introduced so far.
- Everything comes with a price, including AVL tree. The performance of AVL tree is good, but coding is a bit challenging.

When you work on this project, do NOT modify *AVLNode.h, AVLNode.cpp, AVLTree.h* files. You need to implement insert(), deleteNode(), and levelOrder() functions in *AVLTree.cpp* file. Function insert() tries to insert (ss, na) to the AVL, if value ss exists in current AVL, insert() function returns false; otherwise, (ss, na) is inserted to the AVL tree and the tree is rebalanced if it is necessary. After

insertion/balance checking/rebalance is done, insert() returns true.

Function deleteNode() tries to delete the node containing value ss. If there is no such node, it returns false. Otherwise, it deletes the node, check the balance of the tree, rebalance the tree if it is necessary. When you delete a node, consider three different scenarios:

- The node is a leaf
- The node has only ONE child subtree
- The node has two child subtrees

Function levelOrder() travers the nodes in the binary tree in level order. After the traversal is done, it prints out the total number of nodes of the tree. Please note that this function does NOT print out the contents of each node. It just prints out the total number of nodes. You can use this function result to help you do debugging.

Important:

- When you implement this project, do NOT use recursion to implement insert() or deleteNode() functions.
- Each node has three pointers: left, right, parent. The parent pointer allows the user to traverse back to the parent node.
- Understand the insertion/deletion examples in the slides first.
- Then understand the implementation of my single rotations. Draw the pointers on paper. There is no way to figure it out in your mind. Please use pen and paper to the pointers.
- When you implement insertion and deletion functions, use simple examples to test your implementation is correct. If it is, use larger set of data to test it. Implement insert() first, FULLY test it. Then implement deleteNode() later. The deletion part is significantly harder than insertion.
- **Be patient!**
- **Start early!**

Getting help

If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a tutor. Do not wait. A little in-person help can do the magic.

Your Implementation

This project includes three parts.

1. Implement the unfinished methods of AVLTree class. To reduce the complexity of coding, do NOT use template in this project.
2. Write a project5.cpp file, which is the driver to process the data.
3. There are multiple input files are included in the jar file. Use the data files to test your code.

To compile the source code, you can type the following command:

g++ AVLNode.cpp AVLTree.cpp project5.cpp –o project5

The executable file will be *project5*. If your hash table implementation is right, when we execute file *project5*, the result looks like the following contents:

```
jikaili$ ./project5 250000-idr
The Number of Valid Insertation :156536
The Number of Valid Deletion :28338
The Number of Valid Retrieval :18613
The height of the AVL tree :19
Time elaspsed :1.60915
tree size ... 128198

jikaili$ ./project5 500000-idr
The Number of Valid Insertation :312656
The Number of Valid Deletion :56010
The Number of Valid Retrieval :37360
The height of the AVL tree :20
Time elaspsed :3.41635
tree size ... 256646

jikaili$ ./project5 750000-idr
The Number of Valid Insertation :468893
The Number of Valid Deletion :84297
The Number of Valid Retrieval :56129
The height of the AVL tree :21
Time elaspsed :5.61061
tree size ... 384596
```

If you compare the above results with those of project 3 and project 4, the Time values are significantly smaller than those of project3, but larger than project 4. As we mentioned in classes, the time complexity of searching, insertion, and deletion of AVL tree is O(log n). When n value is not large enough, the time complexity of the AVL can

be larger than hash table. When n value is large enough, AVL has better time complexity of Hashing table.

Grading:

1. The **correctness** of insert, delete, and retrieval numbers is 50 points.
2. The **reasonable time** needed to finish the processing is 50 points.
3. A submission that cannot be compiled successfully, the maximum grade is 30.
4. A submission using array, vector, linked list, hash table, or any other data structure other than AVL tree class will get grade 0.

Suggestions!

Start working on the project **EARLY**! It takes a while to figure out how all the components work together. Do not wait until the last minute to start working.

Methodology on testing: Write and test one method at a time! Writing them all and then testing will waste your time. If you have not fully understood what is required, you will make the same mistakes many times. Good programmers write and test **incrementally**, gaining confidence gradually as each method is completed and tested.

Wrap up:

Put all your files, including all input files, your head files and cpp files into **project5.jar**. Submit this jar file to Canvas.