COMS 3137 Data Structures and Algorithms. Homework 3

Submit your electronic files via `http://courseworks.columbia.edu`. Do the written assignments electronically if possible, since it is easier for all parties involved. Use the provided makefile for the programming problems.

Directory structure to be followed for submission:

```
Root folder : <Your UNI>_homework3
  Problem1Src (containing code for the first programming problem)
  Problem2Src (containing code for the second programming problem)
Readme File (describing all your files and notes to the graders)
Theory solutions (optional)
Makefile (optional)
```

Zip up these files to make an archive as instructed in the homework.

1. If you are creating your classes within packages, please maintain the package directory structure during submission.

2. Include the makefile *only* if you need to change the makefile provided to you. However it is strongly advised to conform to the provided makefile.

3. The code should be commented as appropriate or the details should be explained in a readme file.

4. We recommend that you submit your theory/written part electronically.

5. If you are handing in your homeworks, do not take print outs of the code.

Multiple Submissions: You can submit multiple times, but we will only consider the latest submission based on the timestamp in courseworks. Please give at least 1-2 minutes between two submissions.

Late Submissions: Any unexcused late homework submissions will be penalized 10% each day it is late. The penalty begins at the beginning of class the day the assignment is due. The 10% penalties will continue for 3 full days at which point no more late submissions will be graded. If you are submitting late, do not use courseworks and mail your submission to *all* the TAs.

Upload your archive to the Class Files section of courseworks in the Homework 2 subdirectory. It is also recommended that you keep a pristine copy of your submission folder in case there is any submission error.

1 Written Problems

Make sure your solutions are clear. Diagrams and math are often insufficient to convey exactly what you mean, so supplement with some text. Either pseudocode or Java are acceptable when asked to provide algorithms. Nevertheless, clear, concise English is often preferable.

1. (6 points) **Weiss 4.6** A *full node* is a node with two children. Prove that the number of full nodes plus one is equal to the number of leaves in a nonempty binary tree.

2. (4 points) **Weiss 4.9**

   (a) Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.

   (b) Show the result of deleting the root

3. (4 points) **Weiss 4.19** Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.

4. (4 points) **Weiss 4.27** Show the result of accessing the keys 3, 9, 1, 5 in order of the splay tree in Figure 4.71.

5. (6 points) **Weiss 4.41** Write a routine to list out the nodes of a binary tree in *level-order*. List the root, then nodes at depth 1, followed bynodes at depth 2, and so on. You must do this in linear time. Prove your time bound.

6. (6 points) **Weiss 4.49** Suppose we want to add the operation `findKth` to our repertoire. The operation `findKth(k)` returns the kth smallest element item in the tree. Assume all items are distinct. Explain how to modify the binary search tree to support this operation in $O(\log N)$ [i.e., $O(d)$] average time, without sacrificing the time bounds of any other operation.

7. (6 points) **Weiss 4.47a** Two trees, T_1 and T_2 are *isomorphic* if T_1 can be transformed into T_2 by swapping left and right children of (some of the) nodes in T_1. For instance, the two trees in Figure 4.73 are isomorphic because they are the same if the children of A, B, and G, but not the other nodes, are swapped.

   (a) Give a polynomial time algorithm to decide if two trees are isomorphic.

2 Programming Problems

1. (12 points) **Drawing Binary Trees**

   Modifying the textbook's implementation of Binary Search Trees (the link to the source code is on the course homepage), add methods that draw the current tree in a graphical window.

   Use the file `DrawGraph.java`, which draws nodes, labels and edges, and displays them in a JFrame (you may edit this file if you wish to improve upon it).

   Replace the test main() routine for the extended tree class with one that inserts N random integers into the tree and display the tree at each step. Let the user input N via the command line, or set it to a reasonable value by default if there is no command line input.

   Finally, write a program `DrawBST.java` that allows users to insert and delete nodes from the tree, displaying the tree after each operation.

   - Make sure the graphical depiction of the tree properly displays the nodes in order: a scan from left to right, ignoring the vertical location of nodes, should read the nodes in the tree in order.
   - The class should draw any valid binary tree without crossing edges.
   - You'll also need to include the textbook's `UnderFlowException.java` file to compile BinarySearchTree.

2. (12 points) **Autocompletion via Tries**

   Write a program that prompts the user for the beginning of a word and outputs all the possible words that can complete what the user typed in alphabetical order. For example, one possible user interaction would be:

```
$ java AutoComplete dictionary.txt
Loading Dictonary. Standby...
Dictionary loaded!
Start typing a word and hit enter ('quit!' to end)
algori
Possible completions:
algorithm
algorithmic
```

   Perform the autocompletion lookup by storing a dictionary of words in a *trie*. Load the dictionary when your program starts from a text file of words. Use the included dictionary file which is a modified version of the located at `/usr/dict/words` on CUNIX. (I removed words with numbers and punctuation so you only have to handle letters; make your program case insensitive to keep the branching factor at 26.)

   Write your own trie class for Strings that only performs insertions, lookup and preorder toString() conversion (no deletion is necessary). At the cost of memory usage, simplify your code by storing the full word at each leaf. The lookup method should take the beginning of a word input string and return the sub-trie of words that start with the input string. Calling toString() on this sub-trie should return the desired output (i.e., put the newlines in the string conversion).