Due : November 27th, 2016

UPDATES:

- Due date has been changed to November 27th, with a 2 day grace period (20% deduction).
- Added a new bullet point to the end of the AVLTree Implementation Notes section and altered another bullet point in the same section.
- Updated AVLTreeTest.java to remove all tests that rely on a specific convention for the insertion/deletion and swapping/rotating of nodes.
- Removed the insert(int data) method from CuckooHash.java.

Overview

This assignment consists of two parts: implement a Cuckoo Hash and implement an AVL Tree.

General Notes

- Do not change any method or class signatures. You should only edit inside of the functions. If your code changes any class or method names or signatures, you will receive an automatic 0. You should not implement any other functions or instance variables besides the ones that are provided, unless explicitly allowed.
- If you are using Eclipse, be sure to remove the line declaring your code a package when you submit your code. Failure to do so could result in you receiving a 0 if the autograder fails.
- Make sure your code compiles. Non-compiling code will automatically receive a 0. If you have a problem that is causing you to not be able to compile, it may be better to just comment out the incorrect code and return a dummy value (something like null or -1) so the rest can compile.
- Make sure that your code does not print out anything (there should be no System.out.println in your code). You will receive an automatic 0 if your code outputs something to STDOUT during the tests.
- To ensure that your code will be accepted by the autograder, you should submit your code on YSCEC, download it again, unzip it, recompile it and check the provided test suite. This way, you know that the file you are submitting is the correct one.

Cuckoo Hashing

Cuckoo (뻐꾸기) Hashing is a well known hashing scheme that has some desirable properties. Its name comes from the egg laying procedure of the Cuckoo bird. The bird sometimes lays its eggs in other nests. When the eggs hatch, they push out the eggs that were there. In a Cuckoo Hash, element containment and deletion are both worst case $O(1)$ operations, while insertion can be done in *amortized* $O(1)$ time.

Definition

A Cuckoo Hash is defined by an integer, $N > 1$, two hash functions, h_1 and h_2 both in the form (a, b, N) which map $h : \mathbb{Z} \to \{0, ..., N-1\}$, a constant, $0 < c \leq 1$, and another constant $t > 0$. The Cuckoo Hash contains two arrays of size N, A_1 and A_2, which are initially filled with some default value (in this homework we will use 0). When an element, x, is inserted, we insert it into position $h_1(x)$ in array A_1. If there is an element, y, already in this position, we bump y out, insert x, then insert y into position $h_2(y)$ of array A_2. If there is already an element, z, at *that* position, we bump z out, insert y, then insert z it at position $h_1(z)$ into array A_1. This repeats until one of two situations happens.

- An element is placed in a previously unoccupied spot.
- An insertion causes a chain of bumped elements of length at least t.

If we encounter a sufficiently long chain or we meet the threshold $\frac{e}{2N} \geq c$, where e is number of elements currently in the Cuckoo Hash, we activate a resizing operation. To resize, we set $N' = 2N$ and choose two new hash functions, h_1 and h_2 both in the form (a', b', N'), where a' and b' are not necessarily the same as the previous a and b. We then reinsert (with the new hash functions) every element into two new tables, A'_1, A'_2, both of size N'.

Implementation Notes

- For this assignment, we will only be inserting positive integers into the Cuckoo Hash.
- The chain length is defined by the number of bumped out elements during an insertion. So, if you insert an element and it immediately finds an open spot, that chain is length 0. If you insert an element and it bumps an element out

which then bumps another out before finding an open spot, that chain is length 2. We activate the resize as soon as we reach t bumps.

- If after inserting an element the threshold $\frac{e}{2N} \geq c$ would be met, resize the Cuckoo Hash and then insert the new element.
- We have provided an ArrayList that you are required to use. After each insertion into the Cuckoo Hash, append the element to the end of the array list. When you delete from the Cuckoo Hash, you must delete the element from the ArrayList. When you clear the Cuckoo Hash you must also clear the ArrayList. When you resize your Cuckoo Hash, you should reinsert elements in the order that they were originally inserted, so make sure the ArrayList preserves element insertion order. Usually you would not use an ArrayList with such a data structure, but we are requiring it so that we can deterministically check your Cuckoo Hash state in our tests.
- Your hash functions should satisfy $0 < a < N$, $0 \leq b < N$ where N is the current size of the hash tables since we will be calculating them modulo N. However, this is not an absolute requirement. Our only requirement is that a and b are always non-negative (so you can use the % operator). You will not have to check for the case that invalid parameters are passed into the *insert*() method.
- You can iterate over ArrayLists as follows:

```
for(int i = 0; i < list.size(); i++)
{
    list.get(i);
}
```

Directions

- Write your name and student ID number in the comment at the top of the CuckooHash.java file. Please write your name in English letters, not 한글.
- Implement all of the required functions in the CuckooHash class.
- You are not allowed to change the format of this class. You must only use the methods provided.
- Pay careful attention to the required return types.

AVL Tree

The second part of this assignment is implementing an AVL Tree, which was discussed in class.

We will be using a slightly different version of the AVL Tree than what you saw in class. We will be using the convention that the balance factor of a node is defined as $height(right) - height(left)$ where left and right are the node's left and right children. Additionally, in class you discussed trees that have a key and a value associated with each node. In this, the key and value are the same and are the same as the value stored in the *data* variable of a node. So, you should determine where to place nodes based on the data stored in the *data* variable.

Implementation Notes

- The height of a leaf is 0. The height of a null child is -1.
- The Balance Factor should always be one of $\{0, 1, -1\}$.
- ~~When deleting, if a rotation is necessary, you should choose the largest value in the left subtree to be the new root of the subtree. If there is no left subtree, then choose the smallest value in the right subtree to be the new root of the subtree. See the test suite *testThree* for an example.~~
- You should implement the operations as defined in the lectures. Be sure your AVL Tree maintains balance after insertions and deletions.
- We will now be testing mainly structural properties of your AVLTree implementation. This means you can implement your insertion/deletion rotation/swapping procedures using any convention that you want. We recommend you follow the procedures that were discussed in the lectures about BST and AVL Trees. We still require that after each insertion and deletion, your AVLTree must preserve the BST property (no element with larger data should be in the left subtree of a node, etc) and the AVL Balance Property (the balance factor of a node should always be one of $\{0, 1, -1\}$).

Directions

- Write your name and student ID number in the comment at the top of the AVLTree.java file. Please write your name in English letters, not 한글.

- Implement all of the required functions in the AVLTree class.
- You are allowed to add any helper functions and variables that you would like, as long as you do not delete or modify any of the existing ones. We will only be testing the ones that are provided, so make sure your code runs through the correct methods.
- Pay careful attention to the required return values.

Deliverables

IMPORTANT: THIS IS DIFFERENT THAN THE PREVIOUS HOMEWORKS! PLEASE READ CAREFULLY!

Submit **only** the following files in a directory named PHW3. This directory should be zipped inside a zip file called PHW3.zip. Be sure your code compiles and does not include the "package" line so that it will work with the autograder. This is your responsibility, not the TA's, so we will not check or correct for this error. It is a good idea to submit your code to YSCEC, download it again, unzip it, and verify the contents (that it compiles, passes the tests, etc). That way you know you are submitting the correct version of your code.

- CuckooHash.java
- AVLTree.java

To clarify, your file structure should look like PHW3.zip → PHW3 → CuckooHash.java, AVLTree.java. It should **not** look like PHW3.zip → CuckooHash.java, AVLTree.java. This means when you unzip the zip file, the first thing you should see is another directory named PHW3. This directory should contain CuckooHash.java and AVLTree.java.

Testing

As before, we have provided a small test suite for you to check your code. The directions for how to use it can be found on the YSCEC notice board. You are provided with the following files to help you test your code.

- CuckooHashTest.java
- CuckooHashTestRunner.java
- AVLTreeTest.java
- AVLTreeTestRunner.java

You will need to download the junit-4.10.jar file from the YSCEC Notice Board.
You should not include any of these files in your submission.
Note that the test suite we will use to grade your code will be much more rigorous than the one provided here (and not necessarily a superset of the provided tests). You should consider making your own test cases to check your code more thoroughly.