CS 345 (Spring 18): Computer Organization

Project #4

AVL and 2-3-4 Trees

due at 5pm, Fri 30 Mar 2018

1 Introduction

In this project, you'll be implementing a two more trees: an AVL tree, and a 2-3-4 tree. We'll be using a setup that is very similar to Project 3: I've provided example implementations (and a `.dot` file tester), and you will write your own versions of the trees.

For this project, I will be adding a requirement that you implement your AVL tree using the `x=change(x)` style; I've discussed this in class, but I'll discuss it again below. For the 2-3-4 tree, the `x=change(x)` style doesn't really work well; you're welcome to try it, but I didn't end up using it in my own implementation.

I've replaced the `Proj03_BST` interface, which we used previously, with `Proj04_Dictionary`. These are actually quite similar, but I renamed it because I wanted to emphasize that we are now moving away from specifically just **BSTs** and now thinking more generally about **dictionaries** - and there are many possible implementations of a dictionary. The updated interface no longer has rotate functions, and I've added a pre-order traversal to the in-order and post-order that it already had.

In this project, I will only test that your `.dot` file generator works for your AVL tree - but I certainly hope that you will write one for your 2-3-4 tree as well! (Remember, you should be using it for debug.)

1.1 Required Filenames to Turn in

You must turn in the following files:

```
Proj04_AVL_student.java
Proj04_234_student.java
Proj04_GenDotFile.java
```

2 Style Requirements

2.1 No Global Variables

You've seen this before.

2.2 AVL Tree: Must use `x=change(x)`

You **must** use the `x=change(x)` style in the AVL tree for both `insert` and `delete`.

I **urge** you to use a recursive style for `search`, and to use `x=change(x)` for your rotate and rebalance helper functions. However, I won't write a formal requirement to that effect.

I'll detail how to use the `x=change(x)` style below.

3 Tree Design Details

3.1 AVL

Your AVL tree must:

- Support (key,value) pairs, just like Project 3. Duplicates are still illegal.
- In delete case 3, you must always pull up the predecessor, just like in Project 3.
- You must keep the `height` field in each node up-to-date.
- When you are looking to perform a rotation to rebalance the tree and you compare two grandchildren, if they have equal height, then always perform the **single** rotation. (That is, assume that this is a L-L or R-R issue.)
- `insert,delete,search` must all run in $O(\lg n)$ time. **Never perform any global, brute-force search or scan of all of the nodes.** (Except for a traversal, of course.)
- The various traversals must run in $O(n)$ time.

3.2 2-3-4

Your 2-3-4 tree must:

- Support the same (key,value) pairs. It also must not allow duplicates.
- Use the **top-down** splitting strategy. That is, as you recurse (during an `insert`), if you find that a node already has 3 keys in it, split it **proactively** - whether or not you will need to split it later.

  Do **not** split any nodes during a `search` or traversal.
- **It is not required to support delete.** The grading script will simply **skip over** any testcase which includes the word `delete` in it; it will run that testcase on your AVL tree, but not on your 2-3-4 tree.
- `x=change(x)` is not required. You may attempt it if you want, but frankly it didn't seem like a good design choice to me. I didn't use it. (My implementation was recursive, but not in that specific style.)
- The various operations have the same performance requirements as AVL.

3.2.1 Traversals of a 2-3-4 Tree

What is the proper order to print the (key,value) pairs when traversing a 2-3-4 tree? This is easy for the in-order traversal: recurse into the first child, then print the first local key, then recurse into the second child, etc. (Make sure that you pay attention to how many keys, and how many child nodes, exist.)

For the pre-order and post-order traversals, you must print **all** of the keys, for this node, at the proper time: in a pre-order you print this before you recurse, and for a post-order, you will print them after.

4 Java Implementation Overview

4.1 `Proj04_GenDotFile`

WRITE THIS CLASS FIRST.

The first part of this class will be a small change from your Project 3 solution. Update your `gen()` function to include the new BSTNode class name; if you want you may update how it draws the tree to include heights. (I found this useful, for debugging the height calculations. But the TAs won't check for it; it's up to you.)

In addition, you will write a second function, also named `gen()`, but it will take a `Proj04_234Node` object, instead of a `Proj04_BSTNode`. I **strongly urge** you to draw a `.dot` file for your 2-3-4 tree as well - but I won't check it automatically, and the TAs won't check it. It is **allowable but a terrible idea** to simply make this a "stub" function that does nothing.

(Unfortunately, you can't simply **skip** this second function, since the example class for the 2-3-4 tree calls it.)

4.2 `Proj04_AVL_student`

This class is very similar to the BST class that you implemented for Project 3; use that code as a baseline[1]. The key changes are:

- The interface has been renamed to `Proj04_Dictionary`, and a few methods have been added and removed.
- Your new tree must (of course!) keep track of the heights of the nodes (the new `BSTNode` class includes that field), and perform AVL rebalancing as required.
- `rotateLeft()`,`rotateRight()` are no longer part of the interface - although you will probably use a version of them as helper functions. **The `x=change(x)` style is strongly encouraged for these helper functions - but not required.**

[1]If you didn't finish Project 3, then you should do so now. I have chosen not to release solutions for that project.

- I urge you (but won't require you) to write helper functions for common operations, such as rebalancing a subtree.
- You must also implement pre-order traversals.

4.3 `Proj04_234_student`

This class implements a 2-3-4 tree. It must implement the same interface as the AVL tree. Of course, I've said that you don't have to implement deletion; however, the interface requires it. Thus you should define the function as normal - but throw an exception if the function is called[2]. I won't test this case, so it won't matter exactly what exception you throw.

You must use the `Proj04_234Node` object to store the nodes of your tree. (I wanted to give you freedom to write your own node class, but I could not because the example code needs to use the same node class as you do.)

5 Test Driver Updates

Generally, the two test driver classes should be familiar from Project 3. The only big change is that the `Proj04_TestDriver` class now requires one more parameter (it must be the first): you must give it either "AVL" or "234" (without the quotes), to tell it which of the two trees to test.

You will also notice that the grading script will automatically refuse to test the 2-3-4 tree with any testcase which includes the word "delete". We do this because the 2-3-4 tree is not supposed to support that operation.

6 Testcase Format

The testcase format is unchanged from Project 3.

7 Matching Error Messages

As with Project 3, you must match the exception error messages which can arise from an `insert` operation (throw an exception if the key is a duplicate) or a `delete` operation. Test the example classes (using the test driver) to find out the proper formats for these messages.

8 `x=change(x)`

The `x=change(x)` style is a way to build recursive operations over trees. The basic idea is that we have functions which perform modifications on the tree - but instead of being **methods** on objects, these are global functions, which

[2] This sounds very silly, but it's actually more common than you might think.

take **a tree as the first parameter.** Each function then returns a **new tree reference**. (Often, but definitely not always, the one returned is the same as the old - but the internal contents have changed.)

The parameter represents the tree (or subtree) **as it used to be,** before the modification occurred. We use global (in Java, we call these "`static`") functions, instead of methods, because **it is perfectly valid for the reference to be null** - this simply means that the old tree was empty! In all cases, however, the parameter is a reference to a node (or null) - and the node is the **root of the subtree.**

The return value from a function like this is the root of the subtree, as it stands **after modification.** So the job of the function is to examine the parameter, look for some simple base cases, and then (if the base cases don't apply), recurse.

The basic layout of an `x=change(x)` function looks like this:

```
NodeType myRecursiveFunction(NodeType oldRoot, ...other params...)
{
    // base cases
    if (oldRoot == null)
        ...what should this function do with an empty tree?...

    if (oldRoot.val == key)  // or some other special case...
        ...what should this function a node that "matches"?...

    // recursive cases
    if (oldRoot.val > key)
        oldRoot.left  = myRecursiveFunction(oldRoot.left,  ...params...);
    else
        oldRoot.right = myRecursiveFunction(oldRoot.right, ...params...);

    return oldRoot;
}
```

You should notice the following features of the function:

- There are almost always some additional parameters (usually including a key) which tell us **what** we are doing, or looking for.
- There is a base case for an empty tree (even if that might be to throw an exception).
- There is a base case for **finding what you're looking for.** (In the case of `insert`, this means throwing an exception.)
- When all of the base cases are handled, we either recurse right or left. We choose by comparing the key inside the node to something from our parameters.

- We pass a reference to the **child** (which might be null) as the first parameter to the recursive call - and we then save the return value back, as the **very same variable.** (This is why we call it `x=change(x)`.)

8.1 How it Works

To see how it works, consider the following tree:

```
   K
  / \
 C   Z
```

First, let's thinkg about how we store the tree itself. We store it as a reference to the `K` node. (In this Project, it's stored as an instance variable inside of your `AVL` class.)

Now, let's assume that we want to insert `D` into the tree. The first call to your function, then, will take `K` as the parameter - because `K` (the node) is the root of the entire tree.

Your code checks the base cases; the parameter is not null, but it also is not the node it was looking for. So it compares the node's key, to the value it is looking for; it finds that it needs to go left. So it runs the line of code:

```
node.left = insert(node.left, key);
```

(Of course, `key` here is D, the key we're looking for, `node` points at node `K`, and `node.left` points at node `C`, its left child.

Thus, in the second recursive call, we are passing node `C` as the parameter; this time, we recurse right.

In the third recursive call, the paramter is null, so we stop. The proper way to handle an empty tree (when we're inserting) is to **create a new node.** So we create a node, fill it with the key `D`, and then **returns the new node.** This is the function's way of saying, "The old version of the tree was empty; the new version is the subtree rooted at `D`."

Now, this returns back to the 2nd recursive call (the one at `C`). Remember, we are in the middle of the line:

```
node.right = insert(node.right, key);
```

When `insert()` returns, we now save the return value as the **new value of `node.right`.** Of course, this is a change, but it's exactly as we want: a reference which used to be null now points at a newly-created node.

But when we return again, something strange happens. In this case, what we return is **the exact same reference** as was given to us as a parameter. So when node `K` does

```
node.left = insert(node.left, key);
```

, the value **doesn't change.** It is set back to the **exact same value as before.**

However, we know that the tree **inside** it has changed. That is, while we use the same object (the node `C`) to represent "the entire subtree which is left of `K`," the exact **contents** of that tree have changed.

So, should we not update the pointers in these cases? That's an option. (And if you care about cache performance in a multi-core system, it has some real merit.) However, we choose to update it **always** for two reasons:

- It makes the code lots easier to write.
- It allows the recursive call to **change the root.**

The second feature will be critical in your AVL tree. When you insert into a non-empty AVL tree, very often the root node won't need to be updated. But occasionally it will - when a rotation is required at the root.

So use `x=change(x)` for your AVL Tree. The style doesn't work for all tree types, but when it works, it's very nice!

9 Base Code

Download all of the files from the project directory

`http://lecturer-russ.appspot.com/classes/cs345/spring18/projects/proj04/`

If you want to access any of the files from Lectura, you can also find a mirror of the class website (on any department computer) at:

`/home/russelll/cs345_website/`

10 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

10.1 Testcases

You can find a set of testcases for this project at
`http://lecturer-russ.appspot.com/classes/cs345/spring18/projects/proj04/`

See the descriptions above for the precise testcase format, and also for information about how to run the two test driver classes.

10.2 Other Testcases

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcaes of your own, in order to better test your code.** You are also encouraged to share your testcases on Piazza!

10.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj03`. Place this script, all of the testcase files, and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)