Writeup for second project of
CMSC 420: "Data Structures"
Section 0201, Fall 2018

Theme: Threaded AVL Trees

On-time deadline: Monday, 10-08, 11:59pm (midnight)
Late deadline (30% penalty): Wednesday, 10-10, 11:59pm (midnight)

1 Overview

In this programming project, you will combine two powerful ideas that we have talked about in class in a common framework: **Threaded Binary Search Trees** and **AVL Trees**. The goal is this: we want to combine the efficient lookup guarantees of an AVL tree as well as the amortized constant time of finding the inorder successor of any particular node, which is given to us by threaded trees. All programming will be done in Java, and you will be graded automatically by the CS department's submit server. This write-up contains mostly suggested guidelines and hints as well as instructions about how to submit your project on the submit server.

2 General Programming Guidelines

- The required public methods for implementation are available on our GitHub. Pull the project files, read the source code comments **carefully** and proceed with your implementation. The description of what the public methods should be doing is available in the relevant JavaDoc.

- It is **supremely important** that you check your code for consistency by creating your own JUnit tests. The unit tests that we test your code against tend to be **very comprehensive** in terms of code coverage, so the more corner cases you make sure your code passes, the better. Most IDEs make it very easy to define your own JUnit tests, but if you need help or work outside IDEs, talk to us during office hours or post your questions on Piazza. **Big, big recommendation: author your unit tests before your implementation. Do not even open `ThreadedAVLTree.java` until you have authored your tests**. Another suggestion: have a classmate write your unit tests for you while you write their own. In addition to guaranteeing that you are not in any way influenced by how **you** are thinking about the implementation of the structure, this division will make you have some very elementary practice with the real-life division between an SDE and an SDET. The SDET only knows what the class' *interface* (i.e, set of `public` methods) is, and makes calls **exactly** to the part of the class that is supposed to work well irrespective of inner implementation.

- Grading of the assignment is handled by the performance of your code against the submit server tests as well as **satisfying the criterion of attempting to build a threaded AVL Tree** (as opposed to a simple, unthreaded AVL tree). **An implementation that does not attempt to thread a tree can get at most 50% (fifty per-cent) credit.** Passing a test gives you the points attached next to it on the submit server interface, with the notable special case of the inorder traversal test, for which no credit will be granted unless you have implemented inorder traversal **stacklessly**, using the tree's threads. Your implementation of inorder traversal should contain no stack of any kind, including the one created by the system for recursive calls (so, **no recursive calls should happen**). To further clarify, using recursion is **completely fine** for methods such as `insert()`, `delete()`, etc. It is only for generation of the **inorder traversal** that we require that you do **not** use a stack of any kind (yours or the system's).

3 Provided Code

You will need to fill in the implementation of class `ThreadedAVLTree`. All the details of what the interface (the `public` methods) of this class should be doing are available for you in the JavaDoc.

Note that the class is a generic, which means that it is designed to contain objects of other classes. We constrain it to hold Comparable objects. Comparable is a Java interface that provides access to a method called `compareTo`[1]. All typical classes that you have used so far in your academic careers to represent numerical or string data are `Comparable` classes. In essence, any class of objects that implements the inteface `Comparable` is one that induces a *total ordering* between its objects (or, alternatively, the entire set of objects is isomorphic to the natural numbers). For example, the type `Integer` is defined as `Comparable` to other `Integer`s, since the set $\mathbb{Z}$ of integers is *totally ordered*. Same for `String`s and other `CharSequence`s. A large portion of this class is dedicated to learning about data structures that allow for effcient operations on such totally ordered data. Later on, we will discuss data structures suitable for querying data for which there does not exist a total ordering, such as points in two or more dimensions.

Some notes on the implementation:

- As mentioned in lecture and on the slides, Threaded Trees need one bit of information per pointer to allow the implementation code to discern between ordinary pointers and threads. In a high-level programming language like Java, it is **simply impossible** to store one bit of information in a variable. Hence, for the purposes of this assignment, you may use **any bit scheme that makes sense to you**. For example, you can use a primitive `int` which will have values 0 or 1, or, even better, a `byte`, to minimize data storage redundancy.

- A tree's height is defined as the length of the **longest** path that connects the root to a leaf node. Since the path length is defined as the number of edges connecting the origin with the terminal node, we conclude that **a stub tree (a tree consisting of a single node) has a height of 0 (zero)**. We follow these definitions in our unit tests, where we test your trees against their expected heights. By convention, **the height of a `null` tree is -1.**

- For this project, we assume that there are no duplicate keys in your data structure. This means that, in our unit tests, whenever we delete a key from your tree, we expect it to no longer be found in the tree. You may deal with this invariant in any way you please, e.g. throw an exception if a duplicate is inserted, or delete all instances of a key when we ask for a deletion.

[1] More information on this interface is available on Oracle's website.

4 Suggested workflows

You are free to implement this project **in any way you want, as long as the constraint mentioned above is satisfied** and, of course, you aren't implementing the project by using a library you downloaded off the Internet. We will **not** grade you on readability / maintanability of your code, comments, or backwards compatibility. The following are just suggestions.

- **Test first, implement hardest methods second** (*recommended*): Read the documentation and this PDF **thoroughly**, and **spend two days authoring unit tests based only on the contract given to you by the documentation.** Alternatively, as mentioned above, do this for a friend and have a friend do it for you. It would even be better if the friend was not in the class, since they will be completely untethered from possible implementation specifics! Seriously, spend two days doing this. Our model implementation is just under 500 lines of code, whereas our unit tests are just under 900 lines of code. In other projects, the difference is even more dramatic.

- After you're done with this, take several pieces of paper and try to figure out how insertions and deletions should maintain both the threaded tree and the AVL tree invariants. **Examine all corner cases**. Write pseudocode and verify with your pencil and paper whether all cases are covered. You can rest assured that our unit tests cover virtually everything that you can imagine!

- **AVL first**: Build an AVL tree first and write a unit test file **just for the AVL component** (hint: you should probably make *several assertions based on height*). Then, write another unit test where you check for inorder traversal functionality. Remember: **if you want more than 50% credit, your inorder traversal should be implemented stacklessly, using the tree' s threads**!

- **Threaded first**: Similar to the previous one, only you start with the Threaded BST functionality testing.

5 Hints / Tips

- When **inserting** elements, your code should correctly maintain the tree's threads **and** re-balance the tree as needed. Remember that AVL trees behave like classic BSTs **on the way down**, when the code has to look for the appropriate position to insert the key. However, now that we have threads in our tree, we must make sure that when we follow a pointer, we are still "going down" on the tree, and we aren't following a thread "upwards" into the tree! Additionally, when we allocate space for a new node (which always happens at the leaf level in such trees), we must make sure that we update existing threads and we introduce new ones appropriately! Finally, since the AVL component of the tree has to check for re-balancings of subtrees, you should start thinking about *how you should be updating your threads after a single or double rotation!*

  You might be originally mystified about how to properly update the tree's threads after an insertion. To point you towards the right direction, refer to the simple example in Figure 1, with two insertions about the root which do **not** impose any rotations.

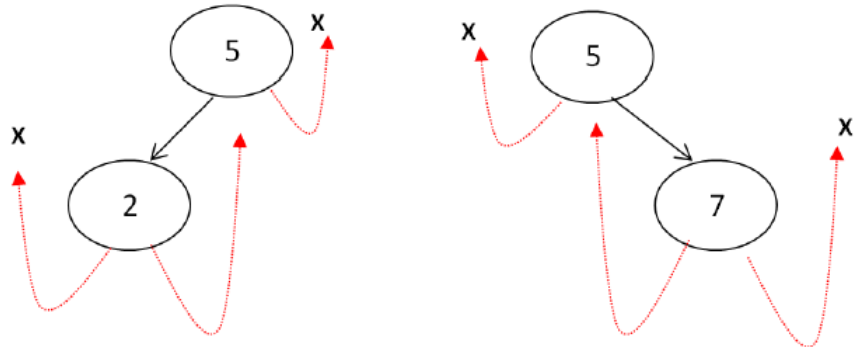

Figure 1: Insertion of a node either on the left or right of a root in a threaded tree. When we insert a node on the left of the root, our inorder predecessor is the root's inorder predecessor, and our successor will be (at most) the root. The converse logic applies when inserting a node on the right.

- Deletion is, without a doubt, the most complex operation that you will have to implement for this data structure. You will need to first figure out the algorithm for deletion of a node from an AVL tree. Furthermore, you must also maintain the threads properly. Recall that there are 4 different cases for deletion of a particular node n:

  (i) n is a **leaf** node.

  (ii) n does **not** have a left child, but **does** have a right child.

  (iii) n **does** have a left child, but does **not** have a right child.

  (iv) n is an inner node, with **both** left **and** right children.

- In addition to considering all of the above, it might be helpful for you to clarify what it means for a node to be a leaf of a **threaded** tree. How do we check for some nodes on an unthreaded tree vs on a threaded tree? What kinds of updates do we need to do in a threaded tree, for all nodes that might be affected by a new insertion?

- Note that the operations of insertion, deletion and search don't really *make use* of the threads; the only method that makes *use* of the threads is `inorderTraversal()`. Of course, if the other methods mess up the threads somehow, `inorderTraversal()` will break!

6 Submission / Grading

Projects in this class are different from your typical 131/2 projects in that **we do not maintain an Eclipse - accessed CVS repository** for you or us. This means that you can no longer use the Eclipse Course Management Plugin to submit your project on the submit server. This turns out to be a good thing, since it frees you up from the need to use Eclipse if you don't want to.

To submit your project, run the script `src/Archiver.java` as a **Java application** from your IDE (tested with Eclipse and IntelliJ). This will create a .zip file of your entire project directory at the same directory level of your entire project directory, without including the hidden `git` directory `.git` that can sometimes be very large and cause problems with uploads on `submit.cs`. For example, if your project directory is under `/home/users/me/mycode/project1/`, this script will create the .zip archive `/home/users/me/mycode/project1.zip`, which will contain `src`, `doc`, and any other directories that you may have, but will **not** contain the directory `.git`. After you have done this, upload the archive on the submit server as seen on figure 2.

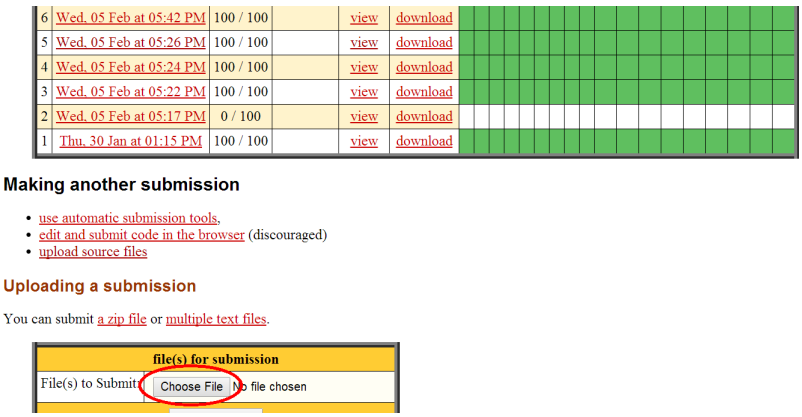

Figure 2: Uploading your project on the submit server.

All tests are release tests, and you can submit **up to 5 times** every 24 hours. **We urge you to unit-test your code thoroughly before submitting**: treat every token like a bar of gold that is not to be wasted! We will **not** share the source code of the unit tests with you, not even after the deadline for the project!

We maintain your **highest-scoring submission** for grading purposes. Finally, for the late deadline, we take 30% off your maximum possible score. This means that, if you submit late, passing all the unit tests will give you 70% of the total grade.

Finally, we should remind you that for the past few years the Software Similarity Detection System MoSS has been incorporated into the CS department's submit server. For n student submissions, it is **ridiculously easy** (literally a single click) for us to run MoSS against all $\binom{n}{2}$ pairs of submissions. MoSS is tuned towards higher than 50% Recall, which means that plagiarized submissions **will** be caught. We would much rather be spending time teaching you data structures and assisting you with your queries than going back and forth with the Honor Council; **help us help you**!