Writeup for first project of
CMSC 420: "Data Structures"
Section 0102 , Summer 2017

Theme: Threaded AVL Trees

Handout date: 06-01
On-time deadline: 06-09, 11:59pm
Late deadline (30% penalty): 06-11, 11:59pm

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 Prerequisites

All prerequisites for handling this project are reasonable for an advanced CS student. We expect that you are well-familiar with Binary Search Trees and Java coding. Skills harnessed by a typical UMD freshman course such as CMSC 131 / 132 are sufficient. Also, the material presented in class on both Threaded and AVL trees will prove to be indispensable for you while implementing this project.

3 General Programming Guidelines

- The required public methods for implementation are available on the instructor's GitHub, under `src.edu.umd.cs.datastructures.projectskeletons.ThreadedAVL`

`Tree.Java`. Download this file, read the source code comments **carefully** and proceed with your implementation. [1] 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 like Eclipse or NetBeans 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.

- Grading of the assignment is handled by the performance of your code against the submit server tests. Passing a test gives you the points attached next to it on the submit server interface. As we mention later in the writeup, there is one test where just passing the test itself will not guarantee credit, since the method through which the test is passed is really the essence of the test. For that test, we will be also inspecting your source code to make sure you have utilized an element of your tree correctly.

4 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 - very simple - 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` [2]. 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 `Comparable` interface 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, `Integer`s are `Comparable`, since the set $\mathbb{Z}$ of integers is totally ordered. Same for `String`s or general `CharSequence`s. A large portion of this class is dedicated to learning about data structures that allow for efficient 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:

[1] For example, one crucial thing mentioned in the source code comments is the need for you to change the package declaration at the very top of the file!

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

- 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` or a `char` such that you minimize data storage redundancy.

- A (rooted) tree's height is defined as the length of the **longest** path that connects the root to a leaf node. By definition of path length (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. **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.

5 Suggested workflows

You are free to implement this project in any way you want. We will not grade you on readability / maintanability of your code, comments, or backwards compatibility. The following are just suggestions.

1. **Testing first, hardest stuff 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 presumably 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!

   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!

2. **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 file where you check for inorder traversal functionality. Remember: **nothing in your inorder traversal should be recursive!**

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

6 Hints / Tips

:

1. When inserting elements, your code should correctly maintain the tree's threads **and** re-balance the tree as needed. You do **not** have to actually use the existing threads in order to implement this method and, in fact, if you were to do so you would largely ignore the AVL structure of the tree, which guarantees that you will never traverse more than $\log_2 n + 1$ levels in the tree in order to insert a node.

   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.

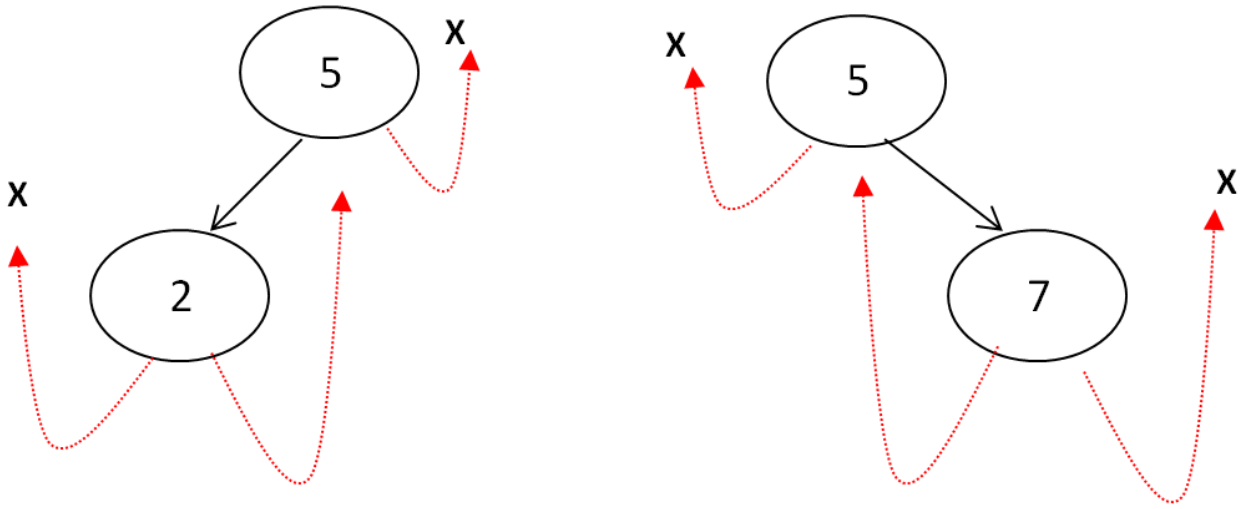

Figure 1: Insertion of a node either on the left or right of a root in a threaded tree. When we insert a node on the left of the root, our inorder predecessor is the root's inorder predecessor, and our successor will be (at most) the root. The converse logic applies when inserting a node on the right.

2. Deletion is, without a doubt, the most complex operation that you will have to implement for this data structure. You will need to figure out the algorithm for proper 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:

   - Deletion of a leaf node.
   - Deletion of a node who does not have a left child.
   - Deletion of a node who does not have a right child.

- Deletion of a purely inner node, with both a left and a right child.

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.

3. For `inorderTraversal`, you **absolutely need** to make use of the threads stored in some of the tree's nodes. It is **not ok** to implement this method recursively, as in a standard BST (or even a non-threaded AVL tree). We have access to your source code after submission, and **we will be making sure that you use the algorithm for finding the inorder successor of a threaded tree in order to implement `inorderTraversal`!** Submissions that pass the unit test but **do not use the tree's threads** will not receive credit for the test!

7 Submission / Grading

Projects in this class are different from your typical 131/2 projects in that we do not maintain a 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 (or any IDE for that matter) if you don't want to. To submit your project, make a .zip file off of your **ENTIRE PROJECT FOLDER** and upload it on the submit server. See figure 2 for the part of the online interface where you can upload your project.

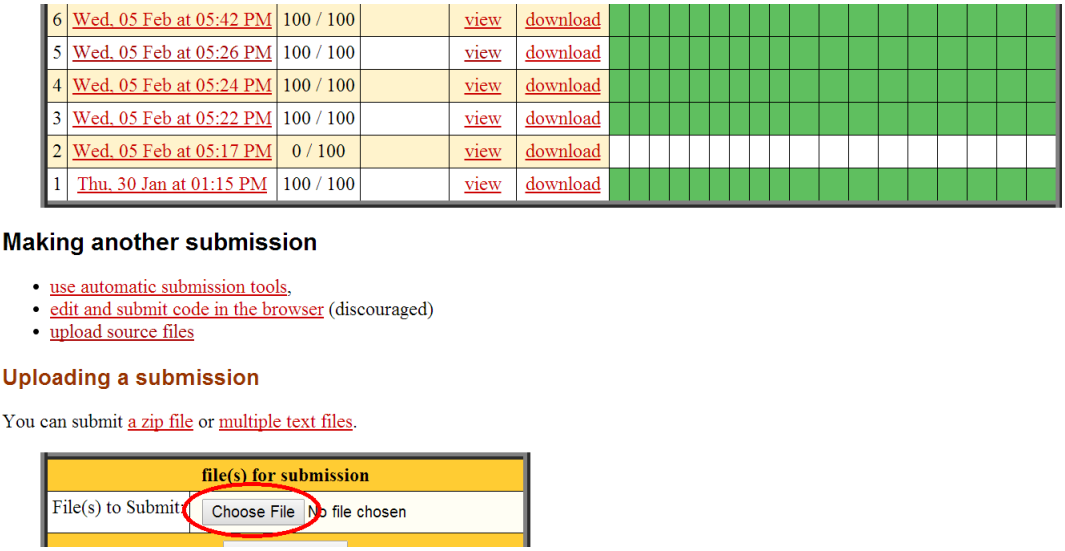

Figure 2: Uploading your project on the submit server.

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 gold bar that is not to be wasted! We will **not** share the source code of the unit tests with you until after the late 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 and implementing inorder traversal **as discussed** will give you 70% of the total grade.

Good luck!