Homework 3: Trees

Due: July 18 2023 11:59pm PST (on Github)

Instructions

Over the last couple weeks of lecture, we discussed various different types of trees that are common in Computer Science applications. In this assignment, you'll get practice working with these data structures. The problems in this assignment cover the topics of Binary Search Trees, AVL Trees and Tries.

For each of the sections of the homework, we provide you with some test cases that you can use to verify the correctness of your code. Also, there are hidden test cases that we will not be releasing. However, by using the autograder we set up on Gradescope (see more details in the **Testing and Autograder** section), you'll have an idea of whether or not your code passes the hidden test cases.

Setting Up

By now, you should have received an invitation to join the CS245-Summer23 organization on Github. Within this organization, there is a public repo titled Assignments. This is where all assignments will be posted from now on. Pull the starter code from the Assignments repo. Create a copy of the Homework 3 directory and move the copy of the directory to your working space. **You do not need to add any jar files for this assignment from the 'jars' folder from Canvas.**

Binary Search Trees (5 points)

We saw that Binary Search Trees on average are really efficient at inserting and looking up keys due to the inherent ordering they maintain. We can also use BSTs to search for ranges of data.

In **BST.java,** fill in the function **rangeQuery** that takes in a lower and upper bound and returns a list of values that are in the BST and fall between the lower and upper bound. For example, if the BST contains the values 1-10 and the lower and upper bounds are -3 and 5 respectively, the list your function outputs should contain the values 1-5 inclusive. On average, your algorithm should not be visiting every node in the BST. Having an average runtime of O(N) where N is the number of nodes will yield you **only 50%** on this question. The ordering of elements in the final output list does not matter **(5 points).**

BSTTest.java Test Cases

- testNoOverlap(1 point)

- testOverlapLeft(1 point)
- testOverlapRight(1 point)
- testOverlapBoth(1 point)
- Hidden Test Case (1 point)

AVL Trees (10 points)

In order to ensure that BSTs have O(log(N)) runtimes for the insert and contains operations, we have to make sure our trees are balanced. We introduced a concept known as rotations to help ensure this property. Trees that use these rotations are known as AVL Trees.

In this next exercise, you are going to implement the left and right rotations we discussed in lecture. We are going to do so by utilizing a concept known as subtree augmentation. Recall that through subtree augmentation, we can compute subtree properties. Subtree properties of a node can be computed quickly if we know the property values of its children. For AVL trees, one of the subtree properties we care about is height.

The general algorithm for inserting into an AVL tree is outlined below:

1. Insert the node into the right spot so that the AVL tree is still a BST
2. Update the subtree properties of the appropriate nodes
3. Find the lowest node that is no longer balanced after the insertion
4. Perform the appropriate series of rotations
5. Update the subtree properties of the appropriate nodes after the rotations have been performed.

In **AVL.java**, we filled in the skeleton of the insert(T value) function. Your job is to complete the helper methods the insert function uses so that it works correctly and maintains the balanced property of the tree. In particular, you'll be implementing:

- **findLowestUnBalancedNode(BSTNode insertedNode)**: find the lowest node in the AVL tree that is unbalanced after putting **insertedNode** into the tree. If all the nodes are balanced, this function should return null.

  A node is considered balanced if the **skew** of the node is in {-1, 0, 1}. We have already provided a method that computes the skew of a node for you.

- **rotateLeft(BSTNode node)**: Perform a left rotation around the node.
- **rotateRight(BSTNode node)**: Perform a right rotation around the node.

If you don't remember how rotations work, review the lectures on AVL trees.

Note: Your rotateLeft and rotateRight functions will need to call the updateAugmentation method that we implemented for you. The updateAugmentation computes the height subtree

property of a node and all its ancestors. Think about why we need to update only the properties of an inserted node's ancestors and not every single node in the tree.

To test the correctness of your AVL implementation, **you'll need to upload your code to the Gradescope autograder**. See the Testing section for more information. The AVL portion of the assignment is worth **10 point**s.

We implemented the **toString()** method in BST.java which is inherited by AVL.java to help you see what the tree looks like. **Do not modify the toString() method in BST.java in any way.** You can do some sanity checks to make sure your AVL implementation is working by printing out your tree after inserting elements in a random order.

Tries(7 points + 3 points EC)

Tries are really useful when the keys we want to handle are strings. For example, suppose we wanted to get all the keys that started with a specific prefix. After traversing to the appropriate node in the Trie (i.e. the node that represents the last letter in the prefix), we would then start the process of determining all the keys that can be reached from that point.

1. To develop some intuition of how this would work, in Trie.java, implement the **collectKeys()** method. This method should return a list of all the keys that are present in the trie. The ordering of the elements in the output list does not matter(**2 points**)

   Test Cases in TrieTest.java
   - testCollect(): 2 points

2. Now, fill in **keysWithPrefix(String prefix)** so that the function returns a list of all keys that start with the specified prefix. The ordering of the elements in your output list does not matter. If your **keysWithPrefix** function calls collectKeys() and then just filters all the keys that start with the prefix, you will receive no credit for this question (**5 points**).

   Test Cases in TrieTest.java
   - **testPrefixCollectNoMatches(): 1 point**
   - **testPrefixCollectEmptyPrefix(): 1 point**
   - **testPrefixCollectAllMatches(): 1 point**
   - **Hidden Test Case: 2 points**

Extra Credit (3 points)

In the skeleton code provided for the Trie class, we maintain the children of a TrieNode using an array. As discussed in lecture, this representation is memory inefficient. Modify the Trie class so

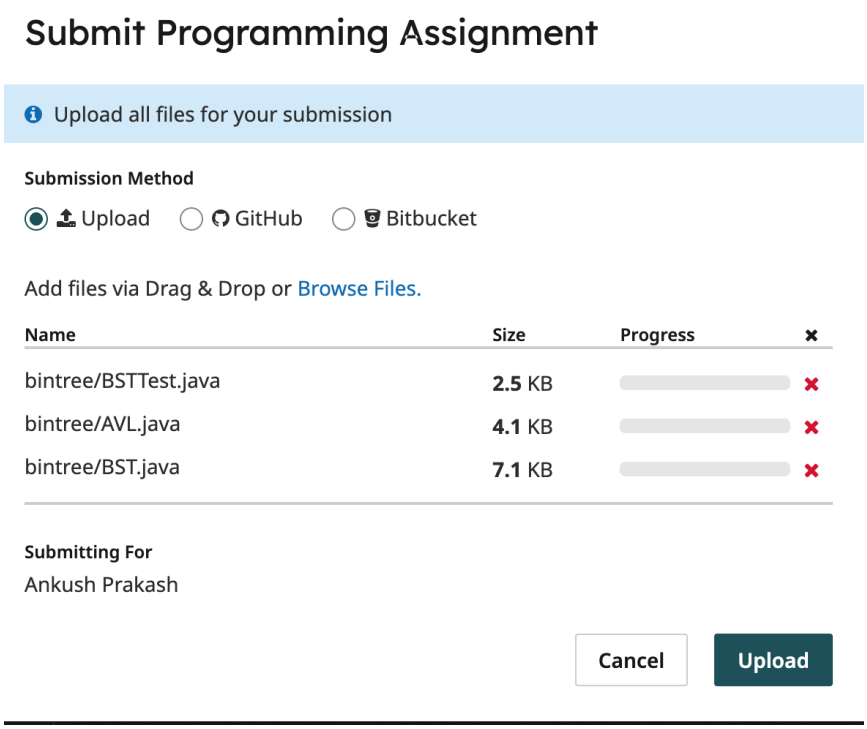
that the children are represented either as a HashMap or a Binary Search Tree. Be sure to adapt your implementations of collectKeys() and keysWithPrefix so that it can support the new way the child nodes are represented.

Testing and Autograder

Everyone should have received an invitation to join the CS245-Summer23 course on Gradescope. Create an account with your student email. Once you log in, you'll see the two assignments on Gradescope that you will be making submissions to: **Homework 3: Binary Trees** and **Homework3: Tries. Remove or comment out all print statements before submitting to the autograder.**

Submitting To Homework3: Binary Trees

If you want to test your code for the Binary Trees and AVL portion, create a zipped version of the **bintree** folder and upload the zipped file to the **Homework3: Binary Trees** assignment on Gradescope. After you upload the zip file, Gradescope will ask you to specify which files from the zip file you want to upload (see example below). Be sure to upload **both** BST.java and AVL.java, even if you have not completed both parts of the assignment. After a few seconds (assuming no infinite loops or recursion depth problems), Gradescope will show you which test cases your code passes and the corresponding point value.



Submitting To Homework3: Tries

Create a zipped version of the **trie** folder and upload it to the **Homework 3: Tries** assignment on **Gradescope**.

If you change any of the package names or signatures of the functions you have been asked to implement, the autograder will fail. If you don't upload a zipped version of the folders mentioned above, the autograder will fail. It is your responsibility to make sure the autograder can process your submission.

Grade Breakdown

BST.java: 5 points
AVL.java: 10 points
Trie.java: 7 points
Extra Credit: 3 points

Total: 22 points
Max Score: 25 points

Submissions

Your final code must be checked into your **private Github repo** that follows the naming convention of **cs245-{firstname}-{lastname}**. See the Github instructions on Canvas for more information on how to setup your Github repo.