CSE 332: Data Structures and Parallelism

Exercise 4 - Spec

The objectives of this exercise are

- **Identify the relationships between binary search trees and AVL trees by implementing AVL trees as a subclass of binary search trees (and thereby inheriting methods whenever possible).**
- **Develop familiarity with the AVLTree data structure by implementing it. In particular: understanding what information needs to be maintained in order to preserve the structure property (height of subtrees differs by 0 or 1), using that information to correctly identify to perform rotation operations.**
- **Highlight the advantages of using an ordered dictionary data structure (like a binary search tree) by using one to implement a new data structure**

Overview

This exercise consists of the following parts:

1. Complete an AVL tree data structure by implementing the insert operation
2. Write operations `findNextKey` and `findPrevKey`, which leverage the ordering of binary search trees and are instrumental in the next part.
3. Complete a `RangeTree` data structure which is used to maintain a collection of non-overlapping ranges of real numbers.

Note that parts 1 and 2 can be attempted in any order. Part 3 requires part 2 to implement the correct behavior, but will need part 1 to satisfy the running time requirements.

Motivation: Event Scheduling

By the time we are finished with this assignment we will have completed three data structures: binary search tree, AVL tree, and range tree. This last data structure, range tree, will be used for storing ranges of numbers with the property that no ranges within the data structure may overlap. This data structure could be used for several applications, but for now let's just focus on one - event scheduling.

Suppose we were responsible for renting out an event space. This event space can host at most one event at a time, so as we fill out our schedule we will need to avoid booking two events that overlap. Our events will be represented by a start time and end time. Two ranges conflict if the start time of one falls between the start and end times of another (it may be worth pausing here to convince yourself that there are no other cases). The range tree data structure will allow us to efficiently check whether a potential event has a conflict with any of the already-booked events, and if it doesn't, to add that event to the data structure.

We will achieve efficiency in this data structure by using the ordering property of binary search trees. The order property of binary search trees (all items to the left of a key have smaller keys, and all items to the right have larger keys) will enable an easy way to check for conflicts among our events.

In case you're interested, here are some other applications of this data structure:

- Scheduling jobs on a cloud computer
- Collision detection in graphics/physics engines (this would likely require a 2d or 3d analog to this data structure, I encourage you to think through what that might look like after completing the assignment!)
- Union of ranges
    - Instead of maintaining a collection of disjoint ranges and refusing to insert conflicting ranges, we could merge together overlapping ranges in our data structure. This could be useful for something like calculating line-of-sight in graphics.

Implementation Guidelines

Your implementations in this assignment must follow these guidelines

- You may not add any import statements beyond those that are already present (except for where expressly permitted in this spec, such as when using an iterative approach for AVL insertion in Part 1). This course is about learning the mechanics of various data structures so that we know how to use them effectively, choose among them, and modify them as needed. Java has built-in implementations of many data structures that we will discuss, in general you should not use them.
- Do not have any package declarations in the code that you submit. The Gradescope autograder may not be able to run your code if it does.
- Remove all print statements from your code before submitting. These could interfere with the Gradescope autograder (mostly because printing consumes a lot of computing time).
- Your code will be evaluated by the gradescope autograder to assess correctness. It will be evaluated by a human to verify running time. Please write your code to be readable. This means adding comments to your methods and removing unused code (including “commented out” code).

Provided Code

Several java classes have been provided for you in this zip file. Here is a description of each.

- `OrderedDeletelessDictionary`
    - A dictionary interface. This differs from the dictionary ADT described in class in the following ways:
        - It does not have a delete operation (hence being called a "deleteless" dictionary). This means that once a key-value pair has been added to the dictionary there is no way of removing that key later. Insert should still update values of previously added keys.
        - It requires the operations `findNextKey` and `findPrevKey`, which take a key as input, and then return the smallest key in the dictionary that's larger than that input, or the largest key in the dictionary that's smaller than that input (respectively). These operations are the reason we call it an "Ordered" dictionary.
        - It requires the operations `getKeys` and `getValues` which return a list of the keys or values in the dictionary. These lists will *both* be sorted according to their keys, meaning index 0 of `getKeys` contains the smallest key and index 0 of `getValues` contains the value associated with the smallest key.
    - *Do not submit this file*
- `BinarySearchTree`
    - When you download the zip, this will be a nearly complete implementation of a binary search tree. We have implemented all of the methods necessary to satisfy the `OrderedDeletelessDictionary` interface except for `findNextKey` and `findPrevKey`, which you will implement as part of this exercise.
    - To help you with debugging, we have provided a method called `printSideways` which displays the structure of the tree. The root of the tree will be in the left-most column, its children in the column after, and the children's children in the column after that, etc.
    - *You will submit this file to Gradescope*
- `AVLTree`
    - After finishing this exercise, this class will represent an implementation of an AVL tree data structure (minus the delete operation). It inherits from BinarySearchTree because many methods can be shared. In particular, since AVL trees have the same order property as binary search trees, all read-only methods can be shared. Only methods which modify the tree will need to be overridden. For this exercise, the only such method is `insert`, which you will implement (along with any helper methods you choose).
    - *You will submit this file to Gradescope*
- `TreeNode`
    - This is a node class that is used by both AVLTree and BinarySearchTree. It has fields for the key, value, left child, right child, and height of the node. `BinarySearchTree` does not use the height field for anything, but it correctly

maintains it nonetheless (which may be helpful to reference when writing the `insert` method for `AVLTree`).
    - There is a method called `updateHeight` which will use the node's children's heights to correctly update its own. It assumes the heights of the children are correct, so it is up to you to meet that assumption before calling the method!
    - *Do not submit this file*
- `Range`
    - This is an object to represent an event for the RangeTree data structure. It contains a start field, an end field (both doubles), and a string for the range. The constructor throws an `IllegalArgumentException` if the start is not strictly less than the end.
    - *Do not submit this file*
- `RangeTree`
    - This class will be an implementation of the RangeTree data structure. You must complete the `hasConflict` and `insert` methods according to the descriptions in the comments.
    - *You will submit this file to Gradescope*

Part 1: AVLTree

To obtain good worst-case performance of our `OrderedDeletelessDictionary` we will implement an AVL tree data structure. The only method required by the interface which might modify the tree is the `insert` operation, so that is the only one that we will override within `AVLTree` (the rest will be inherited from `BinarySearchTree`). The sole task for this part is to write this insert method. The running time of this insert method should be O(log n).

Recall that when inserting into an AVL tree, the general procedure is:

- Navigate the tree as you would with find until you either find the key (in which case you update the value and return the old one), or else reach an empty spot in the tree (in which case you add your new key-value pair as a new leaf node at that spot and return null).
- If you added a node, update the heights of the ancestor nodes as necessary, checking to see if any node becomes unbalanced.
- If a node becomes unbalanced, perform the necessary rotations to correct the imbalance (make sure you update the nodes' heights after rotation!)

To implement this procedure, you'll likely want to attempt one of two approaches:

- A recursive method:
    - For this approach you would employ the "x=change(x)" pattern that you learned in CSE123 or CSE143, provided you took those courses at UW. The idea is that you would have a recursive helper method which takes in a parameter for the key, the value, and a TreeNode. This helper method will perform an insert into the subtree rooted at the given node, and then return the new root of that subtree after the insert is complete (since that might change if a rotation was necessary).
    - To do this you will likely have a base case for when the current root node matches the key we're looking for (meaning you'll update the value), and another when the current root is null (meaning that you'll be creating a new node).
    - In the recursive case you'll want to identify which subtree of the current root should contain the given key (left if the key is smaller than the root).
    - After the recursive insertion is complete, check if the node returned is balanced. If not, then identify and perform the appropriate rotations, then return the correct root (which the rotations may have changed).
- An iterative method
    - For this approach you will navigate through the tree as you would for a find. For each node you visit along the way, push that node onto a stack (so you can follow your path backwards to check balance later). **If you choose this approach you are welcome to import a stack data structure exclusively to be used in this way (e.g. java.util.Stack, java.util.ArrayList, or java.util.LinkedList).**

- If you find the key you're looking for you will update the value and return the old one.
- If you create a new leaf then one node at a time, pop a node off the stack, update that node's height, check for balance. If the node is unbalanced then perform the necessary rotations.

Part 2: findNextKey and findPrevKey

Because our dictionary is ordered (meaning we have information about the comparisons between keys based on where they appear in our data structure) we have extra opportunities to implement methods which leverage that order. For this part you will implement two new find-like operations for `BinarySearchTree` – `findNextKey` and `findPrevKey`.

These operations will be like find in that you will navigate the binary search tree using its order property. They will be unlike find, though, because we will be returning a key rather than a value.

For `findNextKey` we want to return the smallest key which is larger than the key given. In other words, we want to return the "next key up" from the given key from among those currently in the data structure. If the given key happens to be in the data structure then we will still return a key strictly larger than the given key (i.e. the successor of that key). If the given key is the greater than or equal to the largest key in the data structure then return null.

For `findPrevKey` we do the same, but with all comparisons reversed. That is, we want to return the *largest* key which is *smaller* than the key given. If the given key is the *less* than or equal to the *smallest* key in the data structure then we will return null.

If the tree is empty in `findNextKey` and `findPrevKey` return null.

The running time of both operations should be linear in the height of the binary search tree (and will therefore be logarithmic in the size of the tree for an AVL tree).

Part 3: RangeTree

Now the moment we've been waiting for, the `RangeTree`! In this data structure we make use of our `OrderedDeletelessDictionary` implementations. Implement the `hasConflict` and `insert` methods according to the descriptions in the method comments.