Computer Science 228
Project 5
Amortized Weight-Balanced Trees

Important Notes

- This project is worth 250 points.
- The due date is 11:59 pm, Friday, April 26, 2013. As usual, we recommend that you read through this specification completely before you begin coding (and get your questions answered early) and that you start the assignment as soon as possible.
- Some aspects of this specification are subject to change in response to issues detected by students or the course staff. ***Check Blackboard regularly for updates and clarifications.***
- ***This assignment is to be done on your own.*** If you need help, see one of the instructors or the TAs. Please make sure you understand the "Academic Dishonesty Policy" section of the syllabus.

First posted: April 12

Goals

In this assignment, you will

- become more familiar with binary search trees and
- get additional practice with manipulating linked data structures.

1 Introduction

The time complexities of the basic operations on a binary search tree —`contains()`, `add()`, and `remove()`— are proportional to the height of the tree. In the ideal case, the height of an n-element tree is at most $\log_2 n$. If no precautions are taken, however, the height can be $n - 1$.

One way to guarantee that the height of a tree is $O(\log_2 n)$ is to keep it *weight-balanced*. Informally, this means that for each node x in the tree, we ensure that the *the sizes* (numbers of elements) of x's left and right subtrees do not differ by much. Formally, let T be a binary search

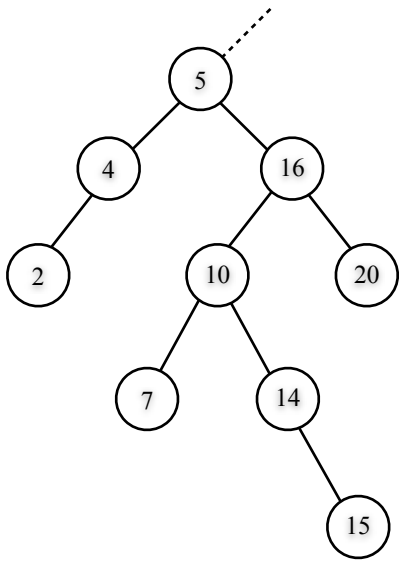

Figure 1: An α-balanced tree (which is a subtree of a larger tree) with $\alpha = 2/3$. For example, consider the node containing 5. The total number of nodes in the subtree is 9. The left subtree has 2 nodes, so we have $2/9 \leq 2/3$. The right subtree has 6 nodes, so we have $6/9 \leq 2/3$.

tree and let α be a constant, such that $\frac{1}{2} < \alpha < 1$. Let x be any node in T, and `size` be the number of elements in the subtree rooted at x. We say x is ***α-balanced*** if

$$(\text{number of elements in } x\text{'s left subtree}) \leq \alpha \cdot \texttt{size}, \tag{1}$$

and

$$(\text{number of elements in } x\text{'s right subtree}) \leq \alpha \cdot \texttt{size}. \tag{2}$$

We say the tree T is ***α-balanced*** if every node is α-balanced. An α-balanced tree is shown in Figure 1.

Note that the height of an n-node α-balanced tree is at most $\log_{1/\alpha} n$. Since α is a constant, this means that `contains`, `add`, and `remove` take logarithmic time. However, adding or removing elements can lead to trees that no longer satisfy the balance conditions (1) and (2). In this assignment, you will implement a certain kind of weight-balanced binary search tree that maintains α-balance through a rebalancing strategy described in the next section.

Note. An alternative approach to achieving logarithmic height is to use *height-balanced trees*. In such trees, the *heights* of the left and right subtrees of any node do not differ too much from each other. For example, in an *AVL-tree*, the heights of the left and right trees at any node differ by at most one. *Red-black trees* are another popular height-balanced tree, with a rather more complex balance condition. Red-black trees are used in Java's implementation of the `TreeSet` and `TreeMap` classes. AVL-trees and red-black trees are described in Wikipedia, where you can find links to additional information. We will not discuss height-balanced trees further here.

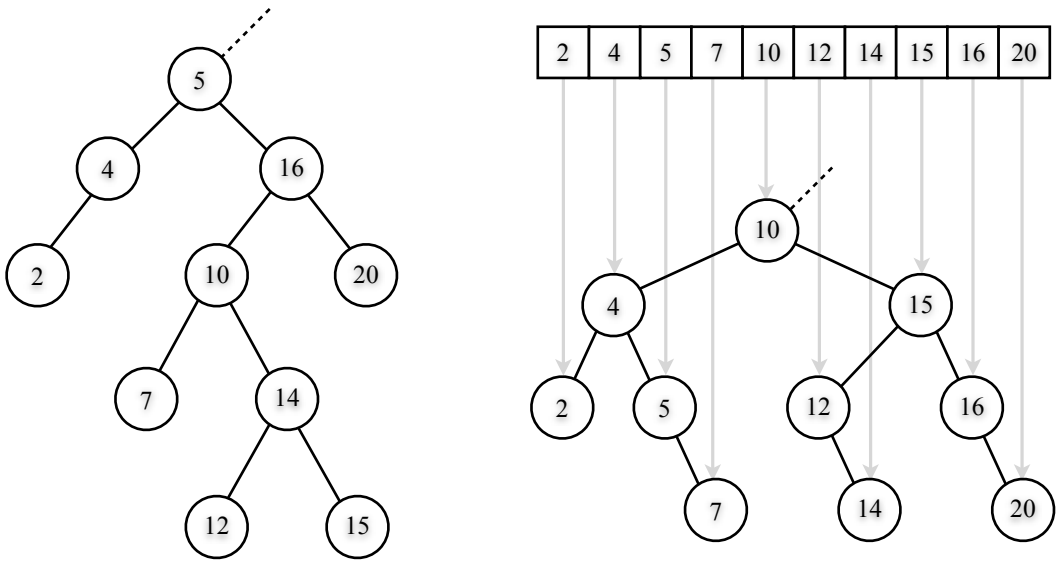

Figure 2: Rebalancing a subtree.

2 Amortized α-Balanced Trees

The weight-balanced trees that you must implement maintain balance by periodically restructuring entire subtrees, rebuilding them so that they become $\frac{1}{2}$-balanced. The work required for rebalancing is $O(n)$ in the worst case, but it can be shown that the *amortized* time for an `add` or `remove` is $O(\log n)$. Although a formal proof of this is beyond the scope of this course, the intuition is that rebalancing is relatively rare.

Next, we explain the rebalancing method on which amortized α-balanced trees rely. We then explain how rebalancing is done after an update.

2.1 The Rebalancing Operation

Suppose x is some node in a BST, and that the subtree rooted at x has k nodes. The ***rebalancing*** operation rearranges the structure of a subtree rooted at x so that it has the same keys, but its height is at most $\log_2 k$. Rebalancing can be done using an inorder traversal of the subtree rooted at x. As we traverse the tree, we put the nodes, in order, into an array or `ArrayList`. The midpoint of the array will be the root of the new subtree, where as usual the midpoint is $(\texttt{first} + \texttt{last})/2$. All the elements to the left of the midpoint will go into its left child, and all the elements to the right of the midpoint go into the right child. An example is shown in Figure 2. Perhaps the most natural way to construct the tree is to use recursion, as shown in Figure 3.

Notes

- Rebalancing a subtree is a purely structural operation that rearranges the links among existing nodes. You should not create any new nodes and you should not have to perform any key comparisons when rebalancing.

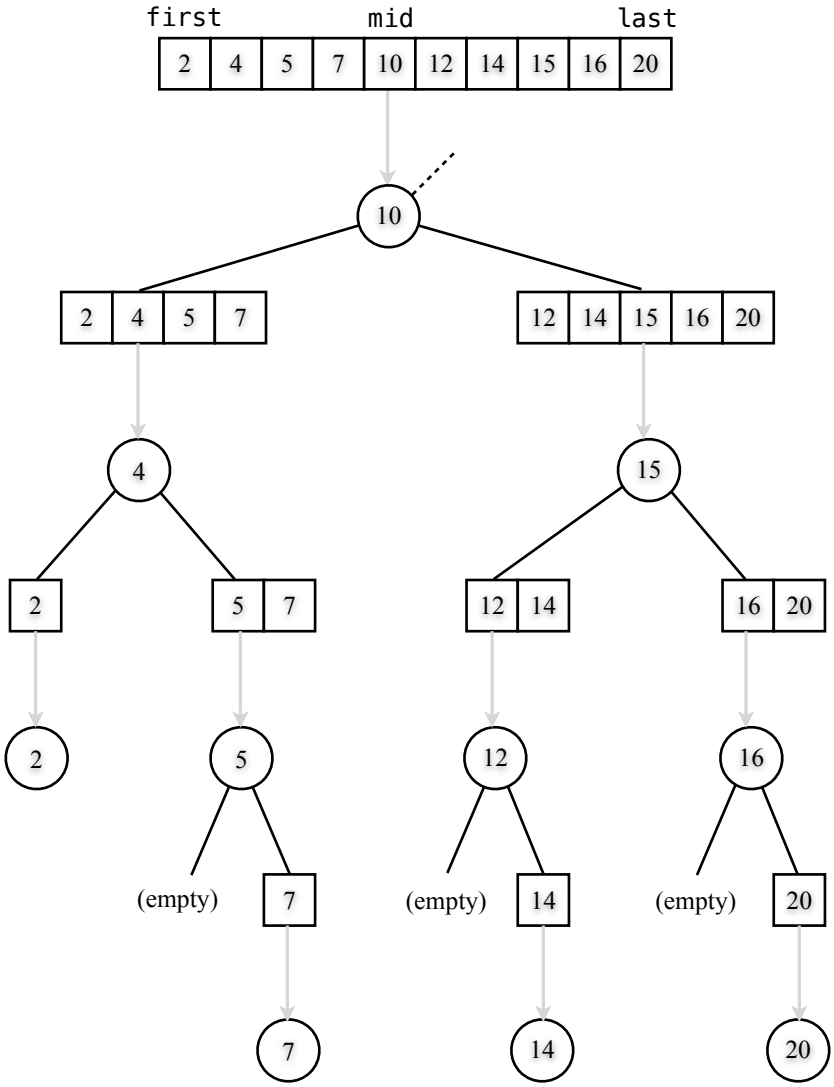

Figure 3: Recursive decomposition.

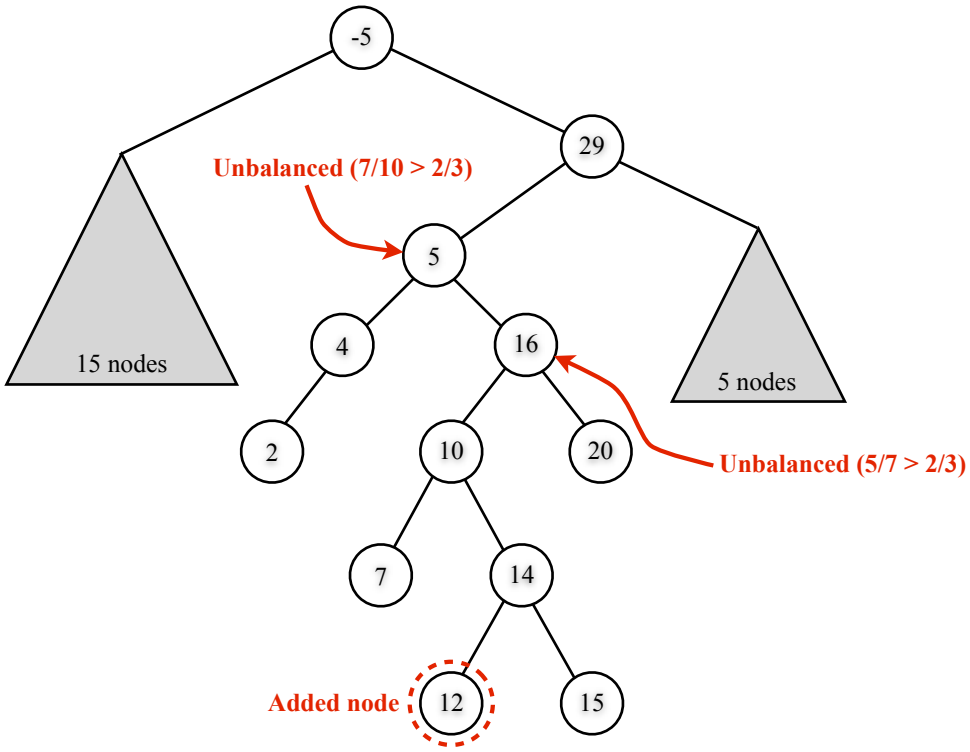

Figure 4: Adding key 12 to a balanced tree, using $\alpha = 2/3$. The node containing 5 is the highest unbalanced node.

- Rebalancing a subtree of size k should take $O(k)$ time.
- The operation may be performed on a subtree, so do not forget to update its parent if necessary.

2.2 Restoring Balance after Updates

After we update a tree, we must check whether it remains α-balanced. If so, nothing more needs to be done. Otherwise, we must rebalance the tree. To be able to detect quickly whether the balance conditions —inequalities (1) and (2)— are violated, we maintain for each node a *count* of the number of elements in that node's subtree. Whenever a node is added or removed, we need to iterate up the tree along the path to the root, starting with the node's parent, updating the node counts. We also need to check whether any nodes along the path have become unbalanced, and identify the highest unbalanced node (if any) along that path. The rebalance operation should be performed on the node closest to the root.

Figure 4 illustrates a tree with 31 elements prior to the addition of key 12. Using a value of $\alpha = 2/3$, the tree is initially balanced. After 12 is added, two of the nodes along the path to the root become unbalanced: the nodes containing 5 and 16, respectively. We rebalance at the node containing 5, since it is the one closest to the root.

3 Tasks

Your job is to implement the class `BalancedBSTSet`, which extends Java's `AbstractSet` abstract class, using amortized α-balanced trees. The starting point for your implementation should be the sample code in `BalancedBSTSet.java` provided along with this assignment. The class `BalancedBSTSet` is almost identical to the binary search tree code developed in class and posted on Blackboard, with minor changes to assist in testing. Specifically, there are methods in place to provide a public, read-only view of the tree structure, and there is a public `rebalance()` method, which should implement the rebalancing operation described in Section 2.1.

To avoid any problems with floating point arithmetic that could arise from using Inequalities (1) and (2), we represent α using two integer instance variables `top` and `bottom` that give its numerator and denominator; i.e., $\alpha = \texttt{top}/\texttt{bottom}$. Then, Inequalities (1) and (2) are expressed as

$$(\text{number of elements in } x\text{'s left subtree}) \cdot \texttt{bottom} \leq \texttt{size} \cdot \texttt{top}, \tag{3}$$

and

$$(\text{number of elements in } x\text{'s right subtree}) \cdot \texttt{bottom} \leq \texttt{size} \cdot \texttt{top}. \tag{4}$$

The default value should be $\texttt{top} = 2$ and $\texttt{bottom} = 3$ (i.e., $\alpha = 2/3$).

The public interface `BSTNode<E>`, provided with this assignment, defines the following read-only accessors for a node in a binary search tree (see the javadoc for details):

```
BSTNode<E>left();
BSTNode<E>right();
BSTNode<E>parent();
int count();
E data();
```

The `left()`, `right()`, `parent()`, and `data()` methods are self-explanatory. The `count()` method should return the total number of elements in the subtree rooted at that node. This method is needed to determine which, if any, nodes have become unbalanced as a result of an update, and is used to find the root of the subtree at which the rebalance operation must be applied (see Section 2.2). The method can be implemented by maintaining the size of the entire subtree, or by separately maintaining the sizes of the left and right subtrees. In any case, we require that the `count()` method run in constant time.

`BalancedBSTSet` has an inner class `Node` that implements the `BSTNode` interface. You can make any modifications you wish to the inner class `Node`, provided that it continues to conform to the `BSTNode` interface.

The class `BalancedBSTSet` has two additional public methods:

`BSTNode<E> root()`
: Return the root of the tree.

`void rebalance(BSTNode<E> bstNode)`
: Perform a rebalance operation on the subtree rooted at the given node.

There are three constructors.

`public BalancedBSTSet()`
Default constructor. Builds a non-self-balancing tree.

`public BalancedBSTSet(boolean isSelfBalancing)`
If `isSelfBalancing` is true, builds a self-balancing tree with the default value $\alpha = 2/3$. If `isSelfBalancing` is false, builds a non-self-balancing tree.

`public BalancedBSTSet(boolean isSelfBalancing, int top, int bottom)`
If `isSelfBalancing` is true, builds a self-balancing tree with $\alpha =$ `top`/`bottom`. If `isSelfBalancing` is false, builds a non-self-balancing tree (`top` and `bottom` are ignored).

To summarize, your main tasks are as follows.

1. Implement a `rebalance()` operation for `BalancedBSTSet`.
2. Modify the `Node` class and the `add()`, `remove()`, and `Iterator.remove()` methods to maintain counts at each node. The `count()` method must be $O(1)$.
3. Modify the `add()`, `remove()`, and `Iterator.remove()` methods so that, if the tree is constructed with the `isSelfBalancing` flag true, the tree is self-balancing. That is, if an operation causes any node to become unbalanced, a rebalance is automatically performed on the highest unbalanced node (which will always be somewhere along the path to the root).

Observe that items (1) and (2) can be done independently.

Notes

- The tree should maintain correct node counts whether or not it is self-balancing.
- Any subtree can be explicitly rebalanced using the `rebalance()` method, whether or not the tree is self-balancing.

4 Submission

Turn in a zip file named `Firstname_Lastname_HW5.zip` containing all your source code for the `NonRecursiveMergeSort` class. The package name for the project must be `edu.iastate.cs228.hw5`. Include the Javadoc tag `@author` in each class source file. ***Do not turn in*** `.class` ***files.***

You are strongly encouraged to write unit tests as you develop your solution. However, you do not have to turn in any test code, so you may share your tests on Blackboard.

All submissions must be done through Blackboard Learn. Please follow the guidelines posted under "Documents & Links".

5 Grading

A significant portion of your grade will be based on running our unit tests on your classes. We will also inspect your code to verify that your methods are implemented as specified. Documentation and style will count for roughly 15% of the total points.

Late Penalties

Assignments may be submitted up to 24 hours after the the deadline with a 25% penalty (not counting weekends or holidays). No credit will be given for assignments submitted after that time.