Computer Science 228
Project 5
α-Balanced Trees
(170 Points)

Due: 11:59 pm, Friday, December 2

1 Overview

This project involves three important concepts: sets, balanced binary search trees, and maps.

- A ***set*** is a collection of distinct objects.
- A ***balanced tree*** represents a set of n elements with a natural ordering so that the running time per operation is $O(\log n)$. Depending on the type of balanced tree, this time bound may be worst-case, expected-case, or amortized.
- A ***map*** is an object that maps a finite set of ***keys*** to a collection of ***values***. Each key can map to at most one value, and a map cannot contain duplicate keys. Maps correspond to the mathematical concept of a ***function***. An example of a map is the function that maps the set of student ID numbers (integers) to student names (strings).

In this project, you will gain a deeper understanding of these concepts by writing the following two classes.

`ABTreeSet`: An implementation of sets based on ***α-balanced trees***. Any access or update operation on an n-node α-balanced tree takes $O(\log n)$ amortized time.

`ABTreeMap`: An implementation of maps that uses α-balanced trees.

We will provide you with templates for `ABTreeSet` and `ABTreeMap`. You may add new instance variables and methods to these two classes, but you cannot rename or remove any existing variables or methods, or change any of these variables and methods from `public` to `protected` (or `private`), or vice versa.

Note. Although the official due date is 11:59 pm, Friday, December 2, you may submit the assignment without penalty until 11:59 pm, Friday, December 9, 2016.

2 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 $n - 1$.

There are a number of ways to guarantee that the height of a tree is $O(\log_2 n)$; they all involve some sort of "rebalancing" after updates, thus these trees are sometimes called *self-balancing trees*. Self-balancing trees fall into roughly two categories.

Height-balanced trees. Here, rebalancing is done to ensure that the *heights* of the left and right subtrees of any node do not differ by much.

Weight-balanced trees. Here, rebalancing is done to ensure that the *the sizes* (numbers of elements) of the left and right subtrees of any node do not differ by much.

Examples of height-balanced trees are *AVL-trees*,where the heights of the left and right trees at any node differ by at most one, and *red-black trees*, the heights of the left and right subtrees at any node can differ by a factor of at most two. 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.

This assignment deals with a special kind of weight-balanced tree, which we describe next.

3 α-Balanced Trees

Let T be a binary search 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

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

and

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

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

Simple math shows that the height of an n-node α-balanced tree is at most $\log_{1/\alpha} n$ (the idea is to first observe that the size of the subtree rooted at a node at depth k is at most $\alpha^k n$, and that the size of a non-empty subtree is at least 1). 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). α-Balanced trees 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 CS 228, the intuition is that rebalancing is relatively rare, in the same way that array doubling is rare in the `FirstCollection` class that we saw several weeks ago.

Next, we explain the rebalancing method used by α-balanced trees, and how rebalancing is done after an update.

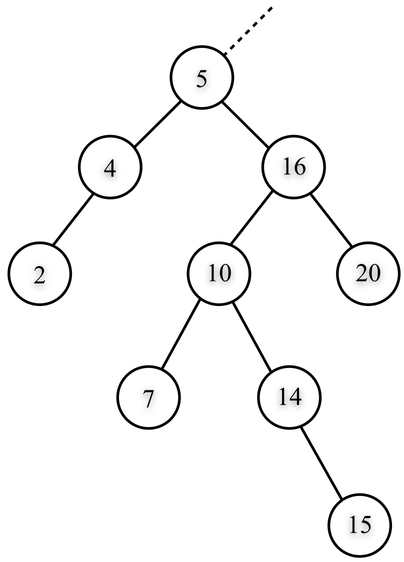

Figure 1: An α-balanced tree (which is a subtree of a larger tree) with $\alpha = 2/3$. For example, consider the node containing 5. The total number of nodes in the subtree is 9. The left subtree has 2 nodes, so we have $2/9 \leq 2/3$. The right subtree has 6 nodes, so we have $6/9 \leq 2/3$.

3.1 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.
- 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.

3.2 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

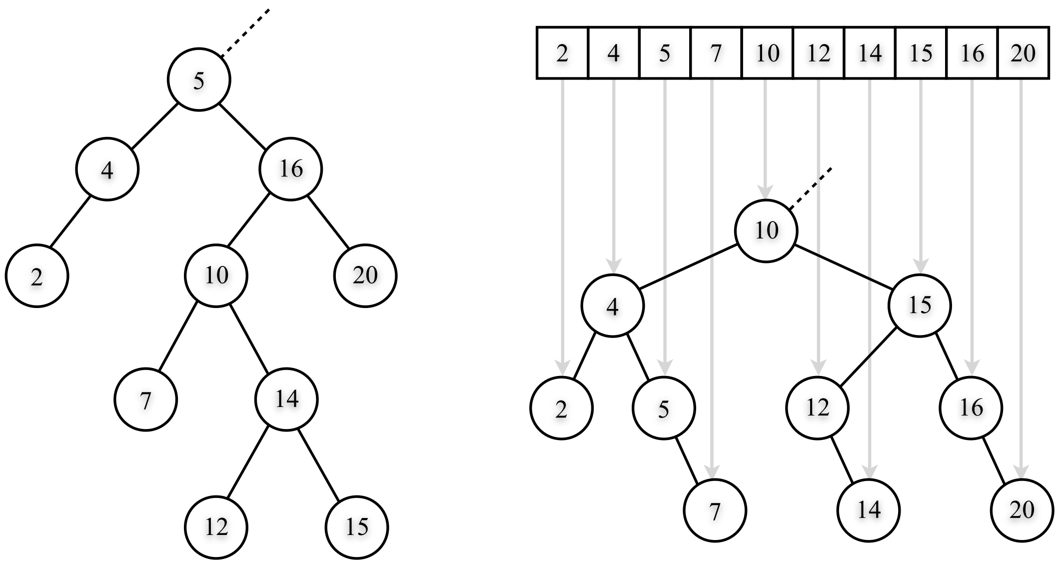

Figure 2: Rebalancing a subtree.

conditions —inequalities (1) and (2)— are violated, we maintain for each node a *count* of the number of elements in that node's subtree; note that this count includes the node itself. 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 node closest to the root.

4 Task 1: `ABTreeSet` (120 Points)

Your first task is to implement the class `ABTreeSet`, which extends Java's `AbstractSet` abstract class, using α-balanced trees. The `ABTreeSet` class implements a set of elements with a natural ordering. Duplicate elements are not allowed. We also disallow `null` elements; any attempt to add a `null` element should result in a `NullPointerException`. The `ABTreeSet` class has the following signature.

```
public class ABTreeSet<E extends Comparable<? super E>>
   extends AbstractSet<E>
```

The starting point for your implementation should be the sample code in `ABTreeSet.java` provided along with this assignment. `ABTreeSet` has methods in place to provide a public, read-

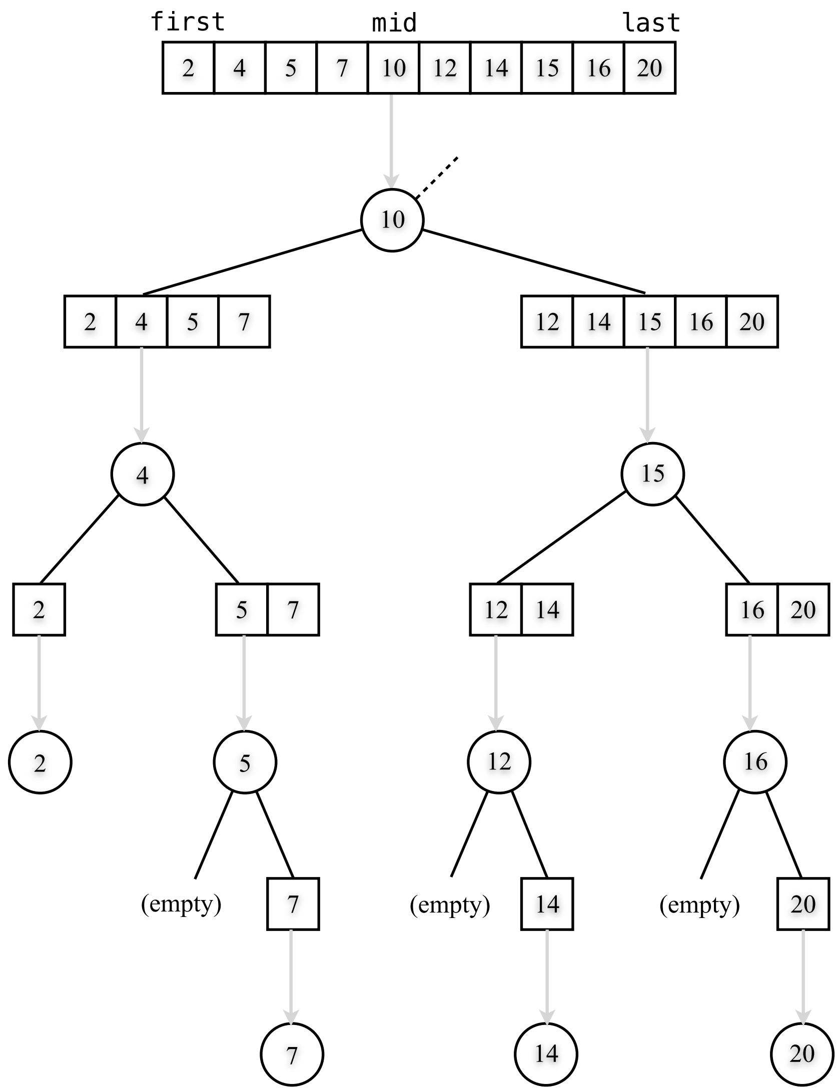

Figure 3: Recursive decomposition.

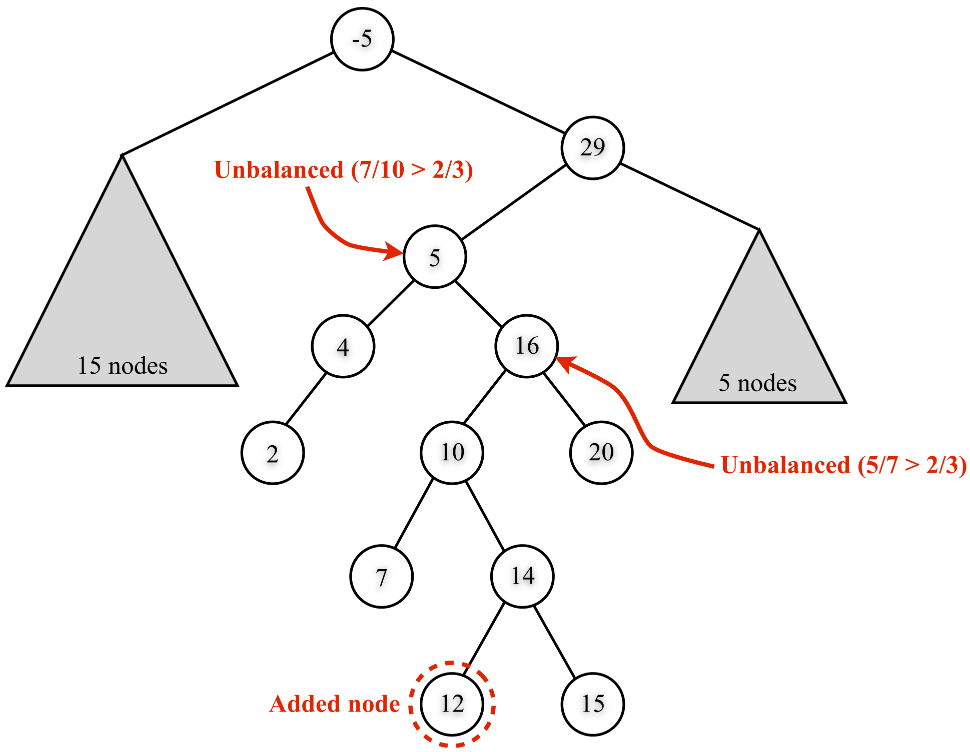

Figure 4: Adding key 12 to a balanced tree, using $\alpha = 2/3$. The node containing 5 is the highest unbalanced node.

only view of the tree structure, and a public `rebalance()` method, which should implement the rebalancing operation described in Section 3.1.

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 3.2). 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.

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

The class `ABTreeSet` 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 ABTreeSet()`

Default constructor. Builds a non-self-balancing tree.

`public ABTreeSet(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 ABTreeSet(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). Throws an `IllegalArgumentException` if `top/bottom` is not strictly greater than $1/2$ and strictly less than 1.

`ABTreeSet` must override `add()`, `contains()`, `remove()`, `size()`, and `iterator()`. You must also override `toString()`, to display the current configuration of the underlying α-balanced for the set. This should be done according to the following rules:

- Every node of the tree occupies a separate line with only its data on it.
- The data stored at a node at depth d is printed with indentation $4d$ (i.e., preceded by $4d$ blanks).
- Start at the root (at depth 0) and display the nodes in a preorder traversal. More specifically, suppose a node x is shown at line ℓ. Then, starting at line $\ell + 1$,
  - recursively print all nodes in the left subtree (if any) of x;
  - recursively print all nodes in the right subtree (if any) of x.
- If a node has a left child but no right child, print its right child as `null`.
- If a node has a right child but no left child, print its left child as `null`.
- If a node is a leaf, print it with no further recursion.

Figure 5 shows an α-balanced tree with 12 nodes, where $\alpha = 2/3$. Figure 6 shows the output that should be generated by calling the `toString()` and `System.out.println()`.

Summary. Your main tasks are as follows.

1. Implement a `rebalance()` operation for `ABTreeSet`.
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).

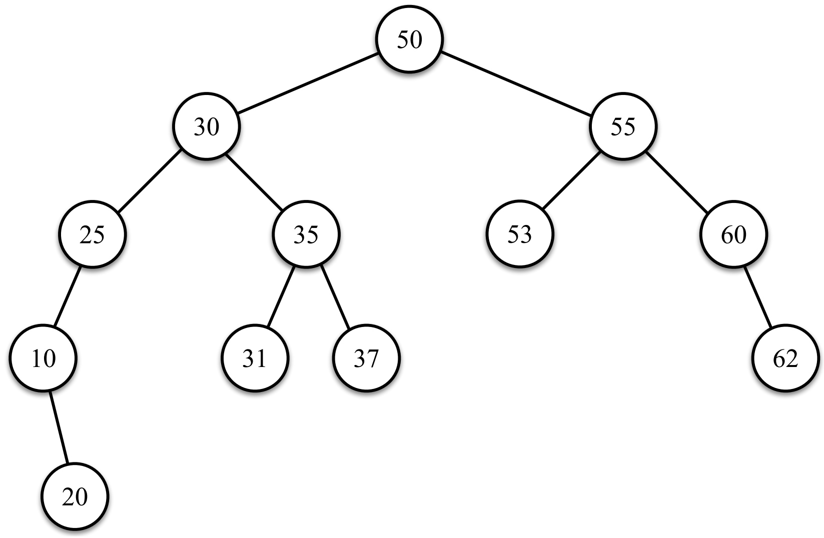

Figure 5: An α-balanced tree, where $\alpha = 2/3$.

```
50
    30
        25
            10
                null
                20
            null
        35
            31
            37
    55
        53
        60
            null
            62
```

Figure 6: The result of `toString()` for the tree of Figure 5.

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

Note the following.

- 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.

5 Task 2: `ABTreeMap` (50 Points)

Your second task is to implement the `ABTreeMap` class, which uses an α-balanced tree to implement a mapping between a set of keys with a natural ordering and a collection of values. Duplicate key values are not allowed. We also disallow `null` keys or values; any attempt to add a `null` key or value should result in a `NullPointerException`. The `ABTreeMap` class has the following signature.

```
public class ABTreeMap<K extends Comparable<? super K>, V>
```

`ABTreeMap` has three constructors.

`public ABTreeMap()`
Default constructor. Builds a map that uses a non-self-balancing tree.

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

`public ABTreeMap(boolean isSelfBalancing, int top, int bottom)`
If `isSelfBalancing` is true, builds a map that uses a self-balancing tree with $\alpha =$ `top/bottom`. If `isSelfBalancing` is false, builds a map that uses a non-self-balancing tree (`top` and `bottom` are ignored). Throws an `IllegalArgumentException` if `top/bottom` is not strictly greater than $1/2$ and strictly less than 1.

`ABTreeMap` has the following methods.

`public V put(K key, V value)`
Associates `value` with `key` in this map. Returns the previous value associated with `key`, or `null` if there was no mapping for `key`.

`public V get(K key)`
Returns the value to which `key` is mapped, or `null` if this map contains no mapping for `key`.

public V remove(K key)

Removes the mapping for `key` from this map if it is present. Returns the previous value associated with `key`, or `null` if there was no mapping for `key`.

public boolean containsKey(K key)

Returns `true` if this map contains a mapping for `key`; otherwise, it returns `false`.

public int size()

Returns the number of key-value mappings in this map.

public ABTreeSet<K> keySet()

Returns an `ABTreeSet` storing the keys (not the values) contained in this map. The tree structure of the `ABTreeSet` should be the same a the tree structure of this `ABTreeMap`.

Example. Suppose this map consists of the following $(\texttt{key}, \texttt{value})$ pairs: $(10, \texttt{Carol})$, $(21, \texttt{Bill})$, $(45, \texttt{Carol})$, $(81, \texttt{Alice})$, $(95, \texttt{Bill})$. Then, the `ABTreeSet` returned should consist of $10, 21, 45, 81, 91$.

public ArrayList<V> values()

Returns an `ArrayList` storing the values contained in this map. Note that there may be duplicate values. The `ArrayList` should contain the values in ascending order of their corresponding keys.

Example. Suppose this map consists of the following $(\texttt{key}, \texttt{value})$ pairs: $(10, \texttt{Carol})$, $(21, \texttt{Bill})$, $(45, \texttt{Carol})$, $(81, \texttt{Alice})$, $(95, \texttt{Bill})$. Then, the `ArrayList` returned should consist of the strings `Carol`, `Bill`, `Carol`, `Alice`, `Bill`, in that order.

Note. Both `keySet()` and `values()` should be implemented by iterating through the `ABTreeSet` that represents the map.

6 Submission

Write your classes in the `edu.iastate.cs228.hw5` package. Turn in the zip file, not your class files. Please follow the guidelines posted under Documents & Links on Blackboard Learn. Also, follow project clarifications on Blackboard. Include the Javadoc tag `@author` in each class source file. Your zip file should be named `Firstname_Lastname_HW5.zip`.