CMSC 420:Fall 2020 Dave Mount

Programming Assignment 2: Weight-Balanced Jackhammer Tree

Handed out: Tue, Nov 3. Due: Mon, Nov 16, 11pm.

Overview: In this assignment you will implement a weight-balanced variant of an extended binary search tree. As in standard ordered dictionaries, the entries stored in your data structure will consist of key-value pairs. Following the same structure of the first programming assignment, all the key-value pairs are stored in the external nodes (leaves) of the tree, and the internal nodes just contain keys as splitters. Given an internal node with key value x, the left subtree contains key-value pairs whose keys are strictly smaller than x and the right subtree contains key-value pairs whose keys are greater than or equal to x.

The new twist here is that each key-value entry will have an associated positive numeric *weight*, which for us will be represented by a Java `float`. So, we can think of each entry in the tree as consisting of three things (x, v, w), where x is its key, v is its associated value, and $w > 0$ is its associated weight. We think of the weight as an indication of the importance of a node, and entries of higher weight (relative to the total weight) should reside closer to the root.

Weight-Balanced Trees: One way to implement this is through the use of *weight-balanced trees.* This generalizes the notion of height balance with standard binary search trees. We assume that we are given a real parameter α, where $1/2 < \alpha < 1$. Given an extended binary search tree T and any internal node p of T, we define p's *weight*, denoted `weight`(p) to be the sum of the weights of the external nodes in the subtree rooted at p. (Note that the internal nodes do not contribute to the weight. They are just there to help us find external nodes.) Define p's *balance ratio*, denoted balance(p) to be

$$\text{balance}(p) \;=\; \frac{\max(\text{weight}(p.\text{left}), \text{weight}(p.\text{right}))}{\text{weight}(p)}.$$

We say that an extended binary search tree T is *α-balanced* if for all internal nodes p in T, balance$(p) \leq \alpha$. Note that as α gets closer to $1/2$, such a tree must be nearly perfectly balanced in the sense that its two subtrees have nearly half of the total weight, and as α approaches 1, the tree can be completely arbitrary, since either subtree can have an arbitrarily large fraction of the total weight.

Unfortunately, when nodes are associated with arbitrary weights, an α-balanced tree may not even exist. For example, if we have just two keys, with weights $w_1 = 0.1$ and $w_2 = 0.9$, the only possible tree will have a balance ratio of 0.9. For this reason, we introduce an additional rule. Given any node, define its *maximum weight*, denoted maxWt(p) to be the largest weight of any single entry in p's subtree. (If the node is an external node, this is just the weight of the associated entry.) Define the *max ratio* of an internal node p, denoted max(p) to be

$$\max(p) \;=\; \frac{\text{maxWt}(p))}{\text{weight}(p)}.$$

Given a numeric parameter $0 \leq \beta \leq 1$, we say that a node p is β-*exempt* if $\max(p) > \beta$. We say that a tree is (α, β)-*balanced* if every internal node p is either β-exempt or is α-balanced. If we chose α and β properly, it can be shown that, for any collection of weighted dictionary entries, there exists an (α, β)-balanced extended binary search containing these keys. In particular, it suffices to choose α and β such that $1/2 < \alpha < 1$ and $\beta < 2\alpha - 1$.[1] In our implementation, we will use the values $\alpha = 0.66667 \approx 2/3$ and $\beta = 1/4 = 0.25$. An example of such a tree is shown in Fig. 1. The tree is valid because all nodes that are not α-balanced are β-exempt.

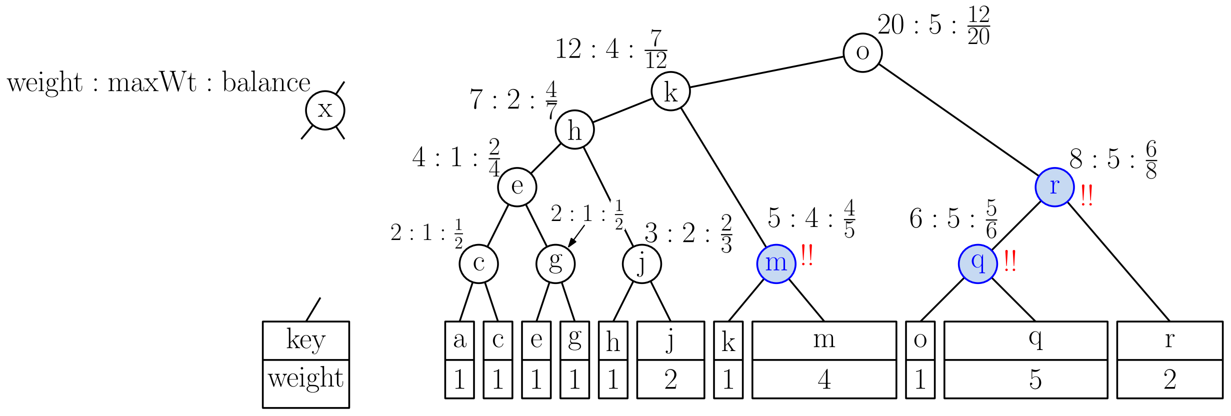

Figure 1: An (α, β)-balanced (extended) search tree, for $\alpha = 2/3$ and $\beta = 1/4$. We have omitted the values, showing just the keys. Each internal node p is labeled with weight(p) : max(p) : balance(p). Nodes have balance ratios larger than α are flagged with "!!" and β-exempt nodes are shaded in blue.

Operations: Below we outline the specifications of our weight-balanced trees, which we call *balanced jackhammer trees* or *BJ trees*, for short. The operations for extended binary search trees are very similar to the standard trees we have seen. Here is a formal definition of how the operations work.

Initialization: The initial tree consists of a `root` pointer whose value is `null`. If the tree consists of a single key-value pair, the root points directly to an external node containing this pair. Once the tree has two or more entries, the root will point to an internal node.

`Value find(Key x):` (Same as in Programming Assignment 1)

`void insert(Key x, Value v, float w):` Inserts the pair (x, v) of weight w. The insertion process starts out exactly as in Programming Assignment 1, but as we are returning from the recursive calls, we update the weights and max-weights associated with each node along the search path. Next, starting at the root, we retrace the search path,

[1]Why is this? Suppose that a node p is not α-balanced. Let us normalize the weights of p's subtree so they sum to 1. (Otherwise, just divide all the following computations through by p's total weight.) In order to be able to create a subtree whose root node is α-balanced, it must be possible to find a splitter that partitions p's weight somewhere within the interval $(1-\alpha, \alpha)$. If this were not possible, there must be an entry that spans this entire interval, meaning that its weight must be at least $\alpha - (1 - \alpha) = 2\alpha - 1$. If there were such an entry, however, its max ratio would be $2\alpha - 1 > \beta$, and so this node would be exempt from the weight-balance condition.

walking down from the root. At the first instance (if any) that we encounter a node p that violates the (α, β)-balance condition (that is, balance(p) $> \alpha$ and $\max(p) \leq \beta$), we apply a procedure (described in the function `buildTree` of Lecture 13) that rebuilds the tree in the most balanced manner possible, and replaces the subtree at p with this new tree.

For the sake of consistency, you should observe the following conventions:

- Use the float value α = `0.66667f` (rather than exactly 2/3) when checking the balance condition.
- Since there may be multiple nodes on the search path that violate the (α, β)-balance conditions, choose the one that is *closest* to the root. (Note that this is the opposite of what Scapegoat trees do.)
- As is done in `buildTree` given in Lecture 13, if there are two splitting indices i that achieve identical weight differences, favor the one that places more weight in the *right subtree*.

`void delete(Key x):` Again, the process starts the same as in Programming Assignment 1, and as we are returning from the recursive calls, we update the weights and max-weights associated with each node along the search path. When we return to the root, we apply the same rebalancing process as for insertion, by walking down the search path until first finding an instance (if any) of a node p that violates the (α, β)-balance condition. We apply the same rebuilding process to this node as in insertion.

For example, in Fig. 2, we consider the deletion of key "m" from the tree in Fig. 1. On left, we show result of the standard deletion algorithm (from Programming Assignment 1), and we update the node weights as we return up the search path. As we descend the search path, we find that node "k" is not α-balanced (its balance ratio is $7/8 > 2/3$), and it is not β-exempt (its max ratio is $2/8 \leq 1/4$), so we rebuild this subtree. The result is shown on the right side.

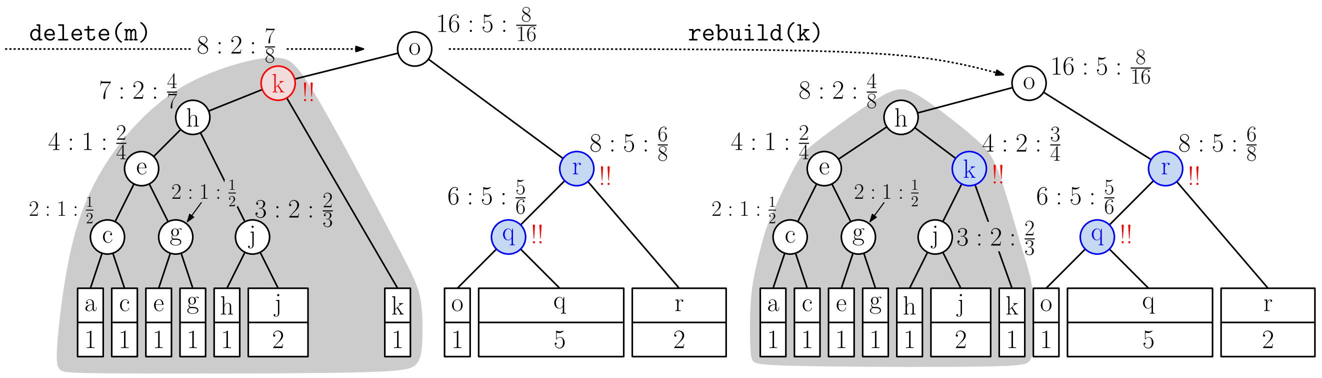

Figure 2: Deletion from a BJ tree.

`ArrayList<String> getPreorderList():` This is the same as in Programming Assignment 1, but we include the weight associated with each node.

- Internal node p storing a key x and having total weight `p.weight`:

```
"(" + x.toString() + ") wt: " + p.weight
```

- External node p storing the pair (x, v) with weight `p.weight`:

  ```
      "[" + x.toString() + " " + v.toString + "] wt: " + p.weight
  ```

  Our testing programs are based on Java `String` equality. Note the use of parentheses for internal nodes and square brackets for external nodes. Also, note the single space between the key and value.

  For example, given the tree of Fig. 1, the elements of the `ArrayList` would consist of the following strings:

  ```
      (...Finish this...)
  ```

Other operations: The remaining operations `getMin()`, `getMax()`, `findDown(x)`, `findUp(x)`, and `clear()` are the same as in Programming Assignment 1.

Running-time Requirements: If no rebuilding takes place, all the dictionary operations (`insert`, `delete`, `find`, `getMin`, `getMax`, `findUp`, and `findDown`) should run in time proportional to the height of the tree. (The find operation should run in time proportional to the length of the search path to the key being sought.) If rebuilding is necessary, the rebuilding time should be $O(k \log k)$ for a subtree with k external nodes and weights are uniform. (The algorithm presented in class achieves the bound.) The operation `getPreorderList` should run in time proportional to the number of nodes in the tree. We will determine this by inspection of your submitted code.

Program structure: We will provide a driver program that will input a set of commands. You need only implement the data structure and the functions listed above. In particular, you will implement a class called `BJTree`, which has the following public interface. (We have included the values of α and β as constants.)

```
package cmsc420_f20;

public class BJTree<Key extends Comparable<Key>, Value> {

    private final float ALPHA = 0.66667f; // maximum allowed balance ratio
    private final float BETA = 0.25f; // maximum weight exemption

    public BJSTree() { ... } // constructs an empty tree
    public Value find(Key x) { ... }
    public void insert(Key x, Value v, float w) throws Exception { ... }
    public void delete(Key x) throws Exception { ... }
    public ArrayList<String> getPreorderList() { ... }
    public Value getMin() { ... }
    public Value getMax() { ... }
    public Value findDown(Key x) { ... }
    public Value findUp(Key x) { ... }
    public void clear() { ... }

}
```

As in Programming Assignment 1, we assume that `Key` and `Value` support the `toString()` method, and the type `Key` implements `Comparable<Key>`.

Node structure: As in Programming Assignment 1, you should use inheritance in your node structure. The notable difference is that, since all nodes will store a weight value, this can be added to the parent class. For external nodes, this is the weight of the entry. For internal nodes, this is the total weight of the entire subtree rooted at this node (that is, `p.weight = p.left.weight + p.right.weight`). Also, for each internal node, the max-weight entry should be cached at this node, since computing it each time we visit the node would take too much time. So, we add an entry `float maxWeight` to each internal node. (If you prefer, you could place it instead in the parent class `Node`. While this is not ideal, it will make your coding simpler to assume that all keys have this field.)

```
public class BJTree<Key extends Comparable<Key>, Value> {

    private abstract class Node { // generic node (purely abstract)
        Key key;
        float weight; // total weight of all entries in this subtree
        abstract Value find(Key x); // no function body for these!
        abstract Node insert(Key x, Value v, float w) throws Exception;
        abstract Node delete(Key x) throws Exception;
        // ... other functions omitted
    }

    private class InternalNode extends Node { // internal node
        Node left;
        Node right;
        float maxWeight; // maximum weight of any single entry

        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
        Node delete(Key x) throws Exception { ... }
        // ... other functions omitted
    }

    private class ExternalNode extends Node { // external node
        Value value;
        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
        Node delete(Key x) throws Exception { ... }
        // ... other functions omitted
    }

    // ... public functions (see above)
}
```

Actual Test Data? We will use the same test data as before. The input format will change, but we will provide you with an updated version of the `CommandHandler` function, so you should not need to worry about this.

Skeleton Code: We will provide you with some skeleton code to start with. This consists of the following:

`BJTree.java`: A skeletal version of the main class for the extended binary search tree. **This is the only file you need modify.**

`Airport.java`: A class that stores information about airports

`Point2D.java`: A small utility class for storing (x, y) coordinates. Used for each airport's latitude and longitude.

`BJTreeTester.java`: Main program for testing your implementation. It inputs commands either from a file or standard input and sends output to another file or standard output.

`CommandHandler.java`: A class that processes commands that are read from the input file and produces the appropriate function calls to the member functions of your `BJTree` class. This also contains a function that converts the output of the `getPreorderList` operation into an indented inorder traversal of the tree, which is a bit easier to read.

You may create additional files as well. Other than `BJTree.java` avoid modifying or reusing any of the above files, since we overwrite them with our own when testing your program. Use the package "`cmsc420_f20`" for all your source files.

Testing/Grading: We will be using Gradescope's autograder and JUnit for testing and grading your submissions. All the tests and the expected results are visible. We will provide a link to the final test data on the class Projects page. Some grading will be done manually, and this will constitute 10% of the final score. We will be checking the following items:

- You should use Java's class inheritance to implement your internal and external nodes.
- All dictionary operations (including `findUp`, `findDown`, `getMin`, and `getMax`) should be implemented so they run in time proportional to the height of the tree (and *not* proportional to the number of nodes in the tree).
- We will not check in detail for adherence to coding standards, but we may deduct points if your code is unusually complex of messy.

Submission Instructions and Late Policy:

As with Programming Assignment 1, submissions will be made through Gradescope. I discovered there is one simplification you can make. Upload your modified `BJTree.java` file, and any other files (if any) that you created. *You do NOT need to upload the files that we provided in the skeleton code.*

Late Policy: The late policy is the same as that listed in the course syllabus:

Up to 6 hours late:	5% of total
Up to 24 hours late:	10% of the total
For each additional 24 hours late:	20% of the total