CMSC 420:Fall 2020 Dave Mount

Programming Assignment 3: Persistent Weight-Balanced Jackhammer Tree

Handed out: Tue, Nov 24. Due: **Mon, Dec 14, 11pm**. (Note that there will be a written homework assignment that will overlap, so plan your time accordingly.) In addition to this handout, this material will be discussed in class on Tue, Nov 24. (Lecture slides and recording will be made available.)

Overview: A dynamic data structure is *persistent*, if it is possible to perform queries not only to the current version of the data structure but to earlier versions as well. In this assignment you will further extend your implementation of the weight-balanced jackhammer tree (BJ tree) to include persistence. To keep matters simple, we will restrict modifications of the data structure to the insert operation (thus, no delete and no clear). We call these *Persistent weight-Balanced Jackhammer trees* or *PBJ trees*, for short.

We assume that each insertion operation is tagged with a nonnegative integer *timestamp*, which indicates the time at which this operation is performed. We assume that these timestamps increase strictly monotonically. For example, suppose that we insert the following keys at times 01, 03, 05, and 07, respectively:

```
01:insert:IAD ...
03:insert:BWI ...
05:insert:SFO ...
07:insert:ATL ...
```

Each query operation (find, find-up, get-min, etc.) is also associated with a nonnegative timestamp t. The query is applied to the tree as it existed at time t. More formally, let t' denote the largest insertion time such that $t' \leq t$. The query is applied to the tree that results as a result of the insertion at time t'. If t is smaller than the timestamp of the earliest insertion, then it is applied to the initial (that is, empty) search tree. For example, given the above insertions, consider the following queries and the associated results:

```
00:find:IAD  ----> fails (prior to the first insertion and tree is empty)
01:find:IAD  ----> succeeds (IAD is inserted at time 01)
08:find:IAD  ----> succeeds (IAD is still present at all times greater than 01)
01:get-min   ----> returns IAD (IAD is the only, hence smallest at time 01)
06:get-min   ----> returns BWI (smallest among {IAD, BWI, SFO} at time 06)
00:get-min   ----> returns null (prior to first insertion and tree is empty)
```

There are many ways to implement persistence in search trees. Our approach will be to introduce a new node type in our tree, called a *temporal node* (in addition to *internal* and *external*). Each temporal node `q` stores a *timestamp* `q.time` and has two children, `q.pre` and `q.post` (see Fig. 1(a)). The subtree referenced by `q.pre` contains the contents of the tree strictly prior to time `q.time` and the subtree referenced by `q.post` contains the contents of the tree on or after this time. When processing a query at some given time `t`, if we arrive at a temporal node `q`, if `t < q.time`, we recurse on `q.pre` and otherwise we recurse on `q.post`.

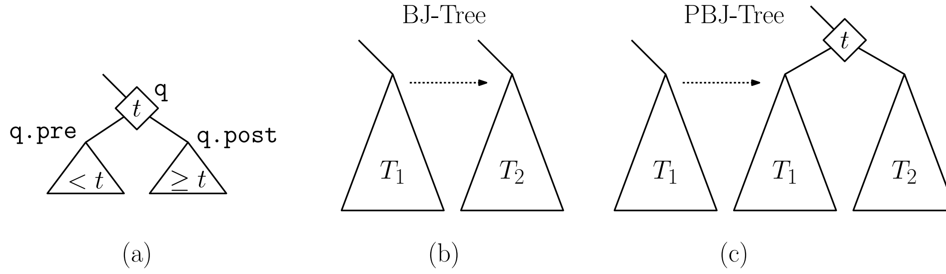

Figure 1: Temporal node and rebuilding subtrees.

For the sake of consistency with the other nodes of the jackhammer tree, it is useful to associate each temporal node with `weight` and `maxWt` values. Because weights are only of use when rebalancing the tree, which can only in the latest version, for a given temporal node `q`, we extract these values from the post subtree:

$$\texttt{q.weight} \leftarrow \texttt{q.post.weight} \quad \text{and} \quad \texttt{q.maxWt} \leftarrow \texttt{q.post.maxWt}$$

(Note that the weight values of `q.pre` have *no effect* on `q`'s weights.)

When and how are temporal nodes generated? It is easiest to see this in the context of when a subtree is rebuilt. Suppose that a subtree T_1 rooted at some node `p` is rebuilt at time t, due to the jackhammer balance conditions (as described in Part 2 of the assignment). After rebuilding, we now have a new subtree T_2. In the standard BJ tree, we would just unlink the old subtree and link in the new subtree (see Fig. 1(b)). In PBJ tree, we will create a new temporal node `q`, and we will set `q.pre` to point to the old subtree T_1 and set `q.post` to point to the new subtree T_2 (see Fig. 1(c)). Queries for times smaller than t are applied to T_1, and otherwise they are applied to T_2.

Technical Specifications: These are the minimum requirements. You may, of course, add additional members and functions.

Data members: The tree has three (private) data members, which are initialized as follows:

`Node root:` The tree's root. (Initial value `null`)

`int firstUpdateTime:` Time of the first insertion. (Initial value -1)

`int lastUpdateTime:` Time of the most recent insertion. (Initial value -1)

`void insert(Key x, Value v, float w, int t):` This inserts the key-value pair (x, v) with weight w in the tree at time t. If the root is `null`, then this is the first insertion into the tree. We create a new external node containing the pair (x, v) and point the root to this node. We also set `firstUpdateTime = lastUpdateTime = t`.

Otherwise, the tree is nonempty. We first check that `t > lastUpdateTime`. (Because insertion times must be monotonically increasing.) If not, we throw an exception ("Update time must exceed prior update time"). We begin by invoking the standard extended binary tree insertion process. Through the use of a recursive helper function, we descend the tree searching for x. (For internal nodes, we use the standard insertion form, `left = left.insert(...)` or `right = right.insert(...)`. For temporal nodes we always

recurse on the post side, `post = post.insert(...)`.) Eventually, we reach an external node. Let y denote the key in this node. If $x == y$, we signal a duplicate-key error. Otherwise, the standard insertion process replaces node y by a three-node subtree, consisting of an internal node containing $\max(x, y)$, and two external nodes containing x and y.

In the persistent setting, we create a new temporal node, labeled with t. Its pre child is the original external node y, and its post child is the three-node subtree described above (see Fig. 2).

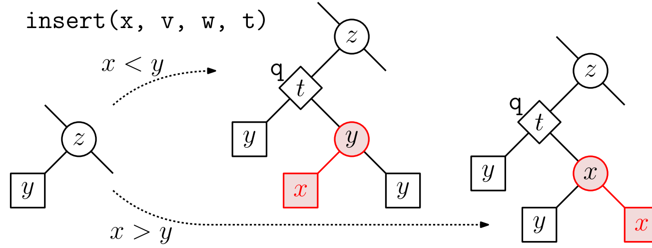

Figure 2: Temporal node and leaf-level insertion.

For example, suppose we insert the key `"D"` with weight 2 into the PBJ tree shown in Fig. 3 at time 7. We follow the search path (post of 3, left of `"I"`) until reaching the external node `"B"`. Since `"D" > "B"`, we create the three-node subtree with `"D"` as its root and make it the post child of a temporal node with timestamp $t = 7$. The original external node `"B"` is its pre child.

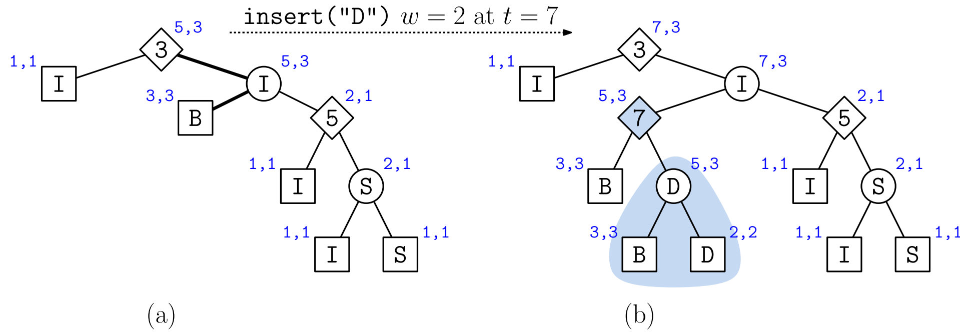

Figure 3: Inserting `"D"` with weight 2 at time 7. The blue numbers by each node are its `weight` and `maxWt` values.

Weight-Based Rebalancing: After performing the insertion, we apply the rebalancing process from Part 2. First, as we are returning from the recursive calls following the insertion, we update the `weight` and `maxWt` values at each node along the search path.

On returning to the root, we initiate a rebalancing process by walking down the tree along the search path. (This is analogous to insertion. We create a recursive helper function, `rebalance(Key x, int t)`. For internal nodes, we do `left = left.rebalance(...)` or `right = right.rebalance(...)`. For temporal nodes we always recurse on the post side, `post = post.rebalance(...)`.) We compute the balance ratio and the max ratio

(see Part 2) at each *internal node* (ignore temporal nodes). If/when we first encounter an internal node that violates the (α, β)-balance conditions (using the same α and β values from Part 2), we rebuild this subtree. To do this, we collect all the external nodes that exist in the tree *at the time of the insertion*. (In other words, you should ignore external nodes in the pre subtrees of any temporal nodes.)

We then apply the `buildTree` function from Part 2 on the resulting list to obtain an ideally balanced subtree. Finally, we create a temporal node whose timestamp is the insert time, and set its pre link to the original subtree and set its post link to the newly created subtree. The `weight` and `maxWt` values of this node are the same as those of the its post subtree. (There is one more step, described below under Temporal Rebalancing.)

This is illustrated in Fig. 4. Observe that only the external nodes that are active at time $t = 8$ are included in the list. Also, observe that the newly rebalanced tree (shaded in blue) is free of any temporal nodes.

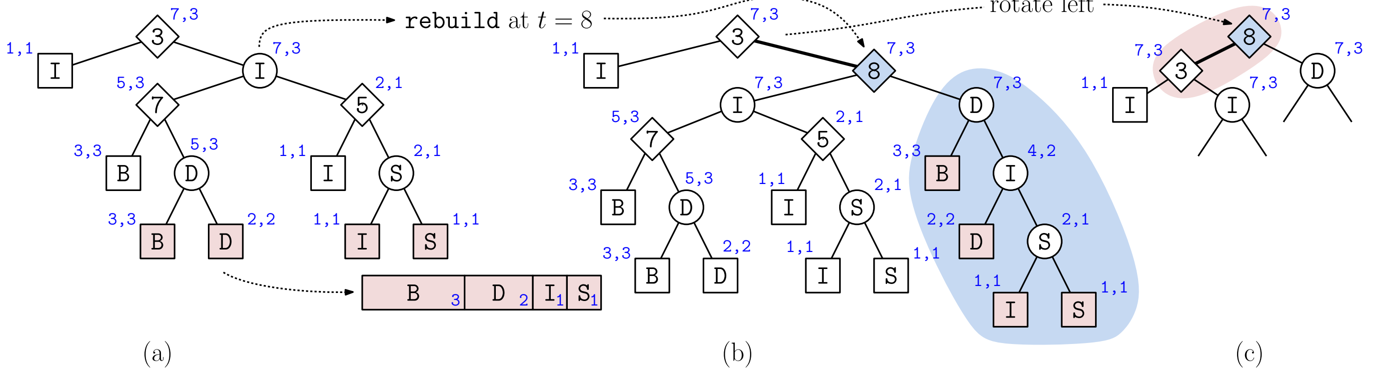

Figure 4: Rebuilding a subtree. (This is *not* a real instance because the subtree is β-exempt.)

Shallow or deep? When copying external nodes, it is always good to be mindful as to whether you are performing *shallow copies* (just copying a reference to a node) or *deep copies* (cloning a duplicate copy of the node). In our figures, we show the result of deep copying. (For example, in Fig. 3, each instance of the external node `"B"` is a distinct instantiation of an external node object.) We tried it both ways, and it appears that it does not matter. The reason is that, once created, the contents of an external node never change. Thus, there appears to be no harm in having many references to the same object in memory. In general, however, using shallow copying in data structure design is a risky practice.

Temporal Rebalancing: Weight-based rebalancing implies that the tree is balanced with respect to weights, but what about the temporal nodes. To keep the data structure simple, we are only going to perform only a very simple form of rebalancing for temporal nodes. To see what the issue is, consider Fig. 4(b), and observe that we have just created a new temporal node 8 as the right child of the temporal node 3. This is not good. If we were to perform a sequence of updates, it is possible to create a long post-chain sequence of temporal nodes with incresing time values. As a result, queries to the current version of the tree (which tend to be the most common) may take a long time.

To remedy this, we want to convert post chains of temporal nodes into pre chains. (While this is not necessarily more balanced, it means that we suffer the performance degradation for searches in the past, not for searches in the current structure.) We implement this as follows. Whenever we invoke `rebalance(x,t)` from a temporal node, recall that we always invoke the function recursively on its post side. (In the example, consider the invocation of `post.rebalance("D", 7)` from the temporal node 3.) When we return from this function, we check whether our post child is a temporal node. (In Java, this can be done using `TemporalNode.class.isInstance(post)`.) If so, we apply a left rotation at this node (see Fig. 4(c)). After doing so, we should also adjust the weights of the two rotated nodes.

`ArrayList<String> getPreorderList(int t)`: As in the previous parts, this returns a Java `ArrayList` of strings containing a preorder traversal of the tree contents at time t. Note that because the time is specified, this list will contain only internal and external nodes (not temporal nodes).

The procedure is almost the same as in Part 2, except we do *not* include the weights of the internal nodes. (Why not? The weight stored in each internal node reflects the weight of the node in the current version of the tree, and hence is not a persistent quantity.) More formally, on encountering a temporal node, we branch to the appropriate child depending on t. On encountering an internal node with key `key`, we generate the string:

```
"(" + key.toString() + ")"
```

and on encountering an external node storing the pair `key` and `value` with weight `weight`, we generate the string:

```
"[" + key.toString() + " " + value.toString() + "] wt: " + weight
```

For example, here is the result of the call `getPreorderList(7)`, for the tree of Fig. 4(a). The weights of the external nodes are shown, but not the internal nodes. Observe that no temporal nodes are given. When we arrive at a temporal node, we visit the appropriate subtree based on the value of t.

```
Preorder list at time = 7:
  (I)
  (D)
  [B Baltimore] wt: 3.0
  [D Washington] wt: 2.0
  (S)
  [I Washington] wt: 1.0
  [S San Francisco] wt: 1.0
```

`ArrayList<String> getFullPreorderList()`: In contrast to the previous command, this command produces a preorder listing of *all* the nodes of the tree, including the temporal nodes. For each temporal node containing a time value `time`, we generate the string:

```
"<" + time + ">"
```

For example, here is the result of the call `getFullPreorderList()`, for the tree of Fig. 4(a).

```
Full-Preorder list at time = 7:
  <3>
  [I Washington] wt: 1.0
```

```
(I)
<7>
[B Baltimore] wt: 3.0
(D)
[B Baltimore] wt: 3.0
[D Washington] wt: 2.0
<5>
[I Washington] wt: 1.0
(S)
[I Washington] wt: 1.0
[S San Francisco] wt: 1.0
```

The structure of this function is almost identical to `getPreorderList`, except that when we visit a temporal node, we add its string to the list and we recurse on both of its children, pre then post.

`Value find(Key x, int t)`: Searches for x in the tree at time t. If the tree is empty (that is, the root is `null`) or t is smaller than `firstUpdateTime`, this returns `null`. Otherwise, a search is performed for x at time t (that is, at each internal node we branch left-right depending on x and at each temporal node we branch pre-post depending on t). On arriving at an external node, we check whether the keys match, and if so we return the associated value. Otherwise, we return `null`.

Other operations: The other query operations `getMin(t)`, `getMax(t)`, `findDown(x,t)`, `findUp(x,t)` are straightforward generalizations of their Part 2 versions. They are given a time value t, and they are applied to the version of the tree at time t. If the tree is empty (that is, the root is `null`) or t is smaller than `firstUpdateTime`, they all return `null`. Otherwise, they apply the search procedure in the same manner as in Part 2, except that on arriving at a temporal node, they branch based on the value of t.

Program structure: As before, 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 `PBJTree`. 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 the first two parts, you should use inheritance in your node structure. There is a new `TemporalNode` node type. Because temporal nodes do not store keys, we have reorganized the node contents slightly. First, we store keys explicitly in the internal and external nodes. Second, we store both the weight and max-weight in the parent node class. (It just makes things a bit more convenient, since all nodes need to access these values.)

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

    private final float ALPHA = 0.66667f; // maximum allowed balance ratio
    private final float BETA = 0.25f; // max ratio filter

    private abstract class Node { // generic node type
        float weight; // the weight associated with this node
        float maxWt; // the max weight in this subtree
```

```
        abstract Value find(Key x, int t); // no function body for these!
        abstract Node insert(Key x, Value v, float w, int t) throws Exception;
        // ... other functions omitted
    }

    private class TemporalNode extends Node { // temporal node
        int time; // timestamp
        Node pre; // subtree prior to timestamp
        Node post; // subtree on or after timestamp

        Value find(Key x, int t) { ... }
        Node insert(Key x, Value v, float w, int t) throws Exception { ... }
        // ... other functions omitted
    }

    private class InternalNode extends Node {
        Key key; // the key
        Node left // subtree smaller than key;
        Node right; // subtree greater or equal to key

        Value find(Key x, int t) { ... }
        Node insert(Key x, Value v, float w, int t) throws Exception { ... }
        // ... other functions omitted
    }

    private class ExternalNode extends Node {
        Key key; // the key
        Value value; // the associated value object

        Value find(Key x, int t) { ... }
        Node insert(Key x, Value v, float w, int t) throws Exception { ... }
        // ... other functions omitted
    }

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

Is this efficient? How much space do we sacrifice for the sake of persistence? It may seem that by rebuilding subtrees, we are wasting lots of space. But this is not so. By the amortized analysis of scapegoat trees, we know that the overall time spent in rebuilding subtrees for m update operations is $O(m \log m)$. It follows that the total space needed is also $O(m \log m)$. Persistence demands that we use at least $O(m)$ space to store these updates. Thus, we pay at most an additional $O(\log n)$ factor to achieve persistence.

Another possible concern is that we do not do any rebalancing with respect to temporal nodes, and indeed, searches in the past may be slow. (We'll leave fixing this as an open problem!) But because of our temporal rebalancing, we know that there can be no two consecutive temporal nodes along the search path for queries in the current version of the tree. Because of weight rebalancing, we know that the internal-node height is $O(\log n)$, and hence the overall

height for the current version of the tree is $O(\log n)$. Thus, queries to the current state of the data structure are guaranteed to be efficient.

Skeleton code, test data, and the submission process will be essentially the same as in the previous two parts. We will provide a link to the final test data on the class Projects page.

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 `PBJTree.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