Recitation 6 – *Treaps and Augmented Trees*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2015)

February 17th, 2015

1 Announcements

- *Exam I* will be on Friday, February 20th - Prepare early!
- How did *RandomLab* go?
- No lab assignment this week! :D

2 Logistics

Q: Where will the exam take place?

A: Rashid, as usual.

Q: When will the exam take place?

A: Friday, 12 – 1:20pm, as usual.

Q: Am I allowed a cheatsheet?

A: YES. 1 sheet (letter... no exotic paper sizes please!) double-sided.

Q: Do I really have to write in blue or black ink?

A: Absolutely. We won't regrade any questions written in pencil.

Q: Is it going to be hard?

A: You bet it is. It's a learning experience! Don't worry, study hard, and you'll be perfectly fine. Most importantly, HAVE FUN!

3 Treaps

You're likely already familiar with binary search trees, trees that store data (keys and possibly values) at the nodes with the following property:

BST Property: If a node has key k, its left subtree contains only keys less than k, and its right subtree contains only keys greater than k.

A heap is a tree with nodes that contain a priority that satisfies the heap property:

Heap Property: The priority of a node is always less than the priority of its parent.[1]

Remember how we construct treaps: A treap is a rooted BST where each node has both a key and a priority. Treaps satisfy both the BST property (on the keys) and the heap property (on the priorities.) Think about it this way: the key decides where the node ends up from left to right, and the priority decides where it goes in the vertical dimension. The nice thing about this is that two treaps with the same keys and priority will always have the same structure, regardless of the order we add the nodes in. Additionally, if we're smart about how we assign priorities to keys, we can make our trees well balanced.

To see this in action, let's try making a basic BST for this sequence of key/value pairs (add nodes in order from left to right).

```
<(1,A),(2,B),(3,C),(4,D),(5,E),(6,F),(7,G),(8,H),(9,I)>
```

Now let's construct a treap, where the tuples are (key, priority, value)

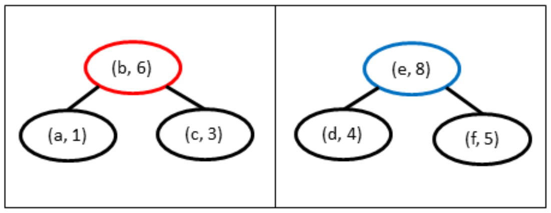
```
<(1,5,A),(2,4,B),(3,7,C),(4,3,D),(5,9,E),(6,2,F),(7,6,G),(8,8,H),(9,1,I)>
```

[1]Technically, this property describes *max*-heaps. We could also use min-heaps which satisfy the corresponding property with "greater" instead of "less." In the notes that follow, we consider only max-heaps.

3.1 Join

The `join` operation on treaps has the same requirements as `join` on BSTs, and works in a similar way. The change is that, when joining two non-leaves, we compare the priorities of the roots and always recursively join the argument with the lower priority together with the appropriate subtree (the one that maintains the BST property) of the node with the higher priority. This ensures that `join` maintains the heap property.

Let's go through an example of joining two treaps with nodes (k, p), where k is an alphabetic key and p is a numeric priority. The trees currently being joined are boxed. The root with the higher priority is highlighted in blue, the one with the lower priority is highlighted in red, and nodes already in the output tree are in green.



4 Treap Analysis

In the lecture notes, you'll see an analysis of the expected depth of a node in a treap. Here, we'll analyze the expected number of children that a node has. The analysis is almost identical, but it'll be a useful exercise in probability and reasoning about trees.

Let's start by turning the question around. Let's find out the probability that node j is an ancestor of node i. We'll designate an indicator random variable for this, A_i^j. Given this, what's an expression for the expected size of the subtree rooted at node j?

So what's the probability of $A_i^j = 1$? (This expression should feel very much like something you did for quicksort.)

5 Augmented Trees

Augmented tree is a balanced binary tree (such as a BST or treap) with a designated (associative) operation $f : \alpha \times \alpha \rightarrow \alpha$ and an operation `reduceVal`: $T \rightarrow \alpha$ which is equivalent to `reduce` f I_f T where I_f is an identity of f.

How is `reduceVal` different from `reduce`? I'm glad you asked.

Unlike `reduce`, `reduceVal` takes $O(1)$ work and span.

What?? How is this possible? (This one isn't rhetorical)

Let's keep growing our trees! Augmenting a BST can get very interesting when we associate with each node a value that reflects some property of the BST. In particular, let's extend the BST ADT with key-ranking! We will augment each node with the sizes of subtrees rooted at that node.

Function `rank(T, k)` returns the rank of the key k in the tree, i.e., the number of keys in T that are less than or equal to k.

Function `select(T,i)` returns the key with the rank i in T.

6 Bonus - Let's Grow Some Money Trees!

You're working as a consultant for the famous QADSAN market, which wants to support the following query: given a time range, return the maximum increase in stock value during that range (i.e. the largest difference between two trades in which the larger amount comes after the smaller amount).

What keys, elements and function f would you use in an augmented tree to support such queries? (**Hint:** the reduced value, on which f operates, may include more information than the answer to the query.)

Calling `reduceVal` and taking the difference between the max and min will only give the maximum increase across an entire (sub)tree. How can we make this query for a specific range (t_1, t_2) in $O(\lg n)$ work?