Recitation 11 — Binary Search Trees and Treaps

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

November 6, 2013

1 Announcements

- The second midterm will be next Thursday, November 14. It will likely be roughly similar in format to the first, and will cover lectures up to and including tomorrow (this midterm will *not* cover dynamic programming, which we will start on Tuesday.)
- No homework this week! Enjoy the break and use the time to prepare for the midterm.
- Next week's recitation will review for the midterm, and we will be having some review and problem-solving sessions next week. Watch the schedule and in lecture for final dates, times and locations.
- Today's recitation will review the join operation on binary search trees and treaps, give intuition for the analysis of treaps, and discuss the Laws of Thermodynamics.

2 Binary Search Trees

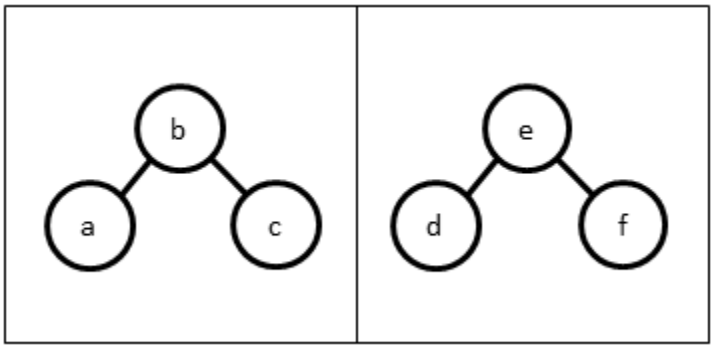
You're likely already familiar with binary search trees, trees that store data (keys and possibly values) at the nodes with the following property:

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.

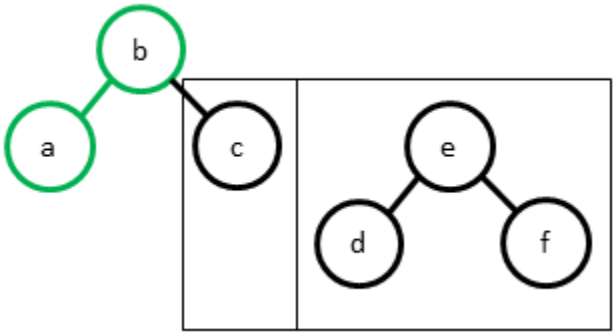
In lecture, we looked at two operations on binary search trees: `split` and `join`. `split` works similarly to standard tree search, but returns the trees to the left and right of the key. `join` is a bit trickier: it combines two valid BSTs (and, optionally, an additional key.) Note that all keys in the first tree argument must be lower than all keys in the second, and that the optional key, if present, must be greater than all keys in the first tree and less than all keys in the second. Here's the basic algorithm. Full pseudocode can be found in the lecture notes.

1. If a key is given, it can become a root node with the arguments as left and right trees as subtrees.
2. Otherwise, if the left subtree is a leaf, return the right subtree.
3. If the left subtree has a key k and subtrees L and R, return a tree with key k, left subtree L and a right subtree formed by recursively joining R with the right subtree argument.

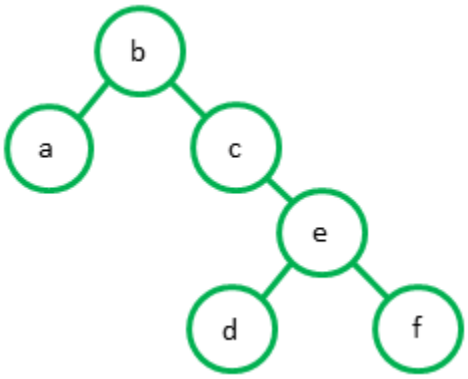
Let's work through an example of joining two trees with no extra key. The trees being joined are boxed. Nodes already established in the output are in green.



We keep the key and left subtree of the first argument, and recursively join on the right.



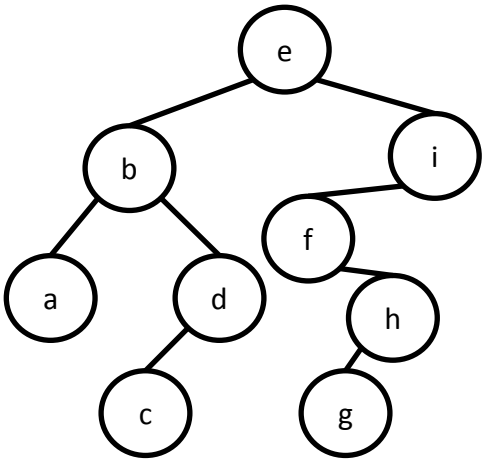
The right subtree of the first argument is a leaf (the key c implicitly contains two leaves as children), so the join simply places the second argument to the right of c.



These two functions allow us to build more familiar primitives like `insert` and `search`.

Q: Let's go over a series of inserts now to review this primitive. Assume that keys are letters, ordered alphabetically. Build a BST by inserting keys in this order, starting with an empty tree: $\{g, a, h, c, d, f, b, i, e\}$.

A:



Q: What if we insert in alphabetical order?

A: We get a long chain, on which we can't perform searches and inserts efficiently.

Clearly, it's important to find a way to keep our trees balanced. Many data structures exist whose insertion and/or search functions perform additional operations to rebalance the tree. However, as we did recently with Quicksort, we can do reasonably well just by using randomization!

3 Treaps

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]

We now combine BSTs and heaps to get treaps, binary trees with nodes containing keys *and* priorities (and possibly values as well.) Treaps satisfy both the BST property (on the keys) and the heap property (on the priorities.) While both the BST property and the heap property easily allow very unbalanced trees, surprisingly treaps will stay relatively balanced in expectation! Note that the "in expectation" is important here. Treaps can be extremely unbalanced, but, as we will see, this is quite rare in practice.

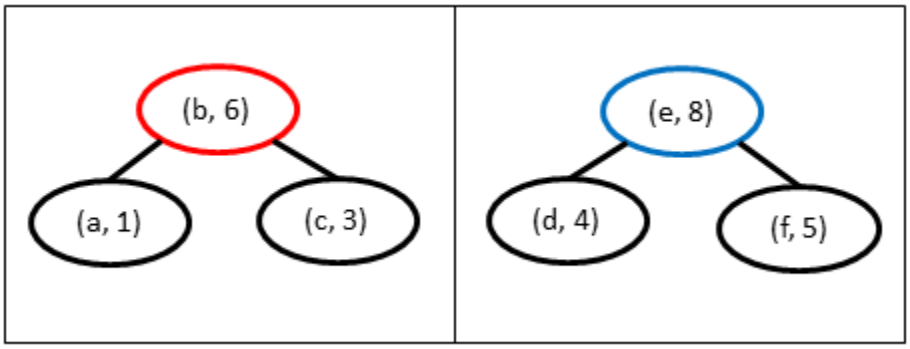
How do we have to change the `split` and `join` functions? Actually, the code for `split` doesn't need to change. It will already respect priorities. `join` is a bit more complicated.

3.1 Join

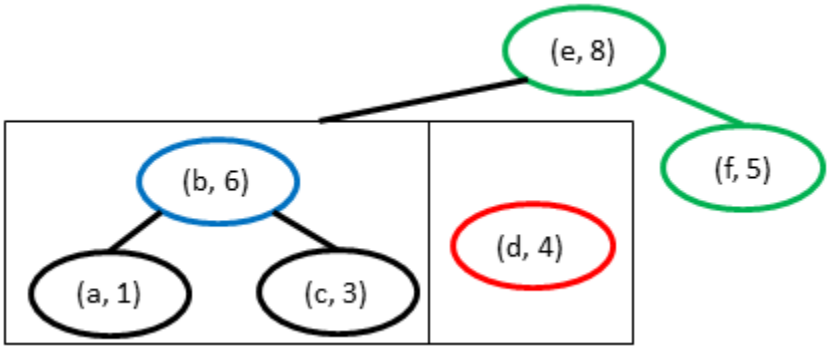
The `join` operation on treaps has the same requirements as `join` on BSTs, and works in a similar way. The change is that, when joining two non-leaves, we compare the priorities of the roots and always recursively join the argument with the lower priority together with the appropriate subtree (the one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

[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.

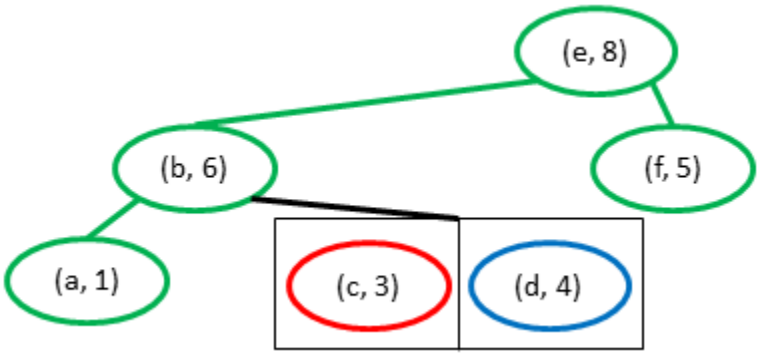
Let's go through an example of joining two treaps with nodes (k, p), where k is an alphabetic key and p is a numeric priority. The trees currently being joined are boxed. The root with the higher priority is highlighted in blue, the one with the lower priority is highlighted in red, and nodes already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



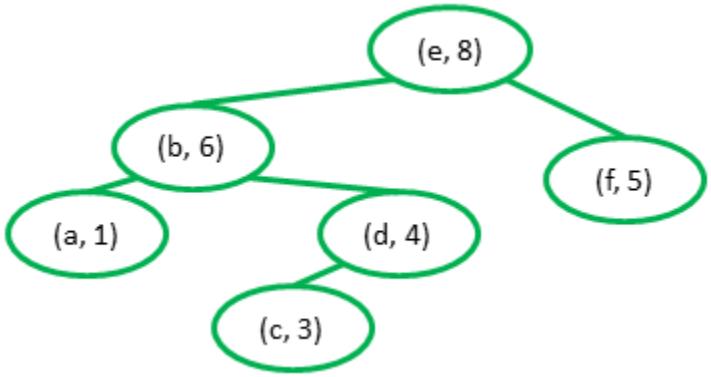
e has a higher priority than b, so e becomes the root and we recursively join the tree rooted at b with d. The result will become the left subtree of e.



Now b has a higher priority, so recursively join c with d to get the right subtree of b.



d has a higher priority, so that will be the root of the joined tree and c will be its left child. This completes the join.



4 Trees and Heaps and Sets, Oh My!

We will explore heaps further by using them to implement another familiar data structure: sets. Indeed, sets (and tables) in the 210 library are implemented using treaps.

Q: Why are treaps a good match for implementing sets?
A: They store keys, with $O(\lg n)$ expected lookup and insertion.

Elements of the set will be nodes in the treap. The element will be the key and there will be no value (in a table, there would be), but what should be the priority of each element? Note that the priority allows us to impose the heap property on top of a BST and keep the tree relatively balanced. So we want to make sure the priorities are well distributed throughout the keys.

Q: How would it be to assign elements priorities sequentially as they're inserted? (i.e. the first one has priority 0, the second one 1, and so on.)
A: It could be fine if we get lucky, but there are workloads where it would be terrible. For example, if keys are inserted in order, this will form a long chain.

We want to assign priorities to the nodes such that it will perform well in general, and no adversary can devise a worst-case workload. We'll use the same trick we did to do this in Quicksort: assign priorities randomly. However, unlike with Quicksort, we don't know in advance all the keys we might need to insert. As long as we have a hash function $h : K \to \mathbb{N}$ (where K is the type of keys), this is fine. Each key is hashed when it's inserted and the hash value is the priority. Assuming a good hash function (one that appears random), priorities will be evenly distributed.

Assume we have the following hash function from letters $a - i$ to numbers:

k	a	b	c	d	e	f	g	h	i
$h(k)$	1	6	3	4	8	5	0	2	7

Whenever we insert a key k, look up its hash value $h(k)$ and insert a node into the treap with key k and priority $h(k)$. The insert can be accomplished by performing a split on k to get left and right subtrees L and R, and then joining L and R with optional key k.

Figure 1 shows how the treap changes as the keys are inserted in alphabetical order, one at a time. In this case, `split` will always return the entire subtree as the left tree and an empty right tree (since all keys already in the tree will be less than the one to insert.) We then join the entire tree with the singleton of the new key.

Notice that after inserting key f, the treap looks the same as the one in the end result of the join example. Both have elements $a - f$ with the same priorities, but they were reached by performing very different sets of

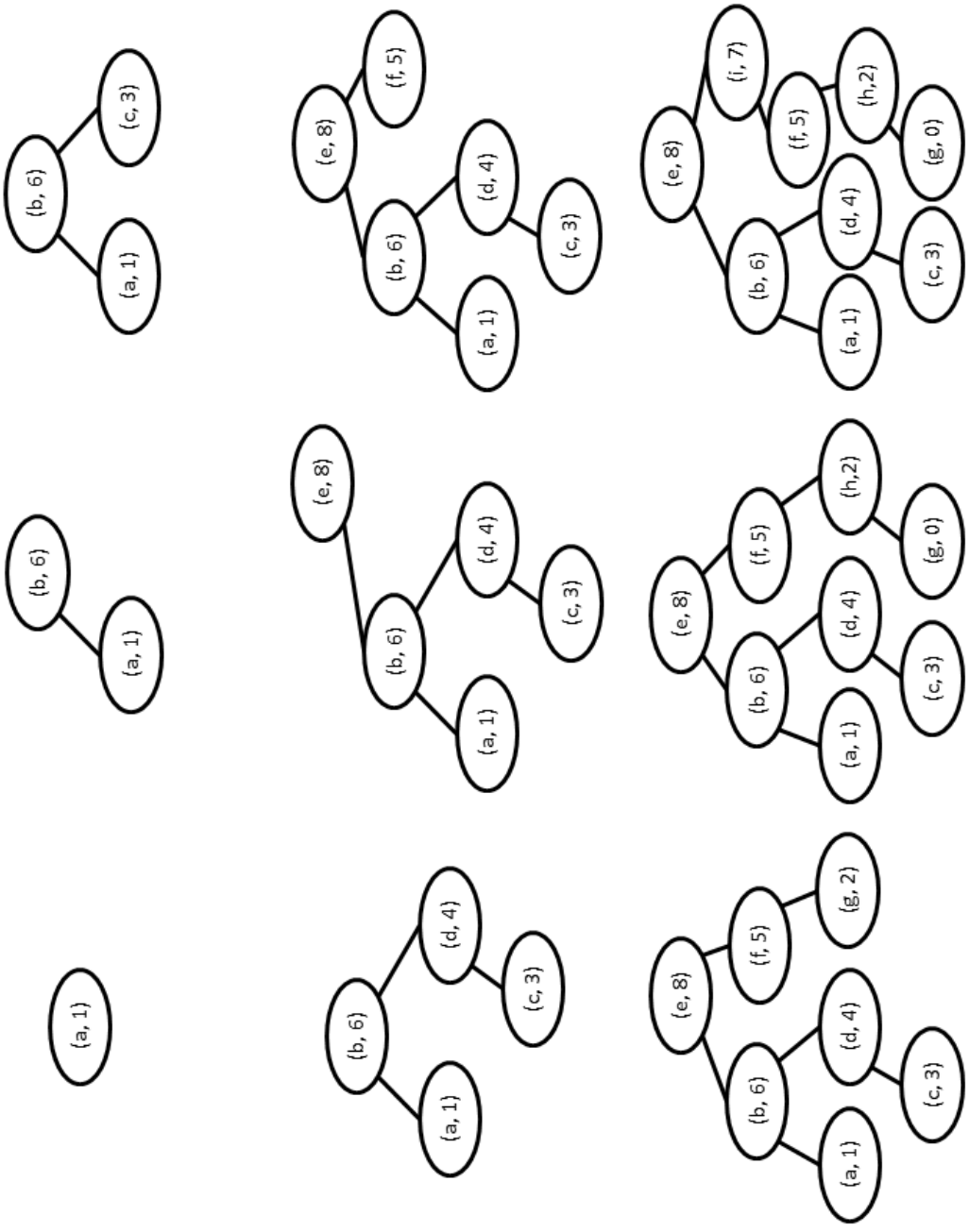


Figure 1: Inserting elements in alphabetical order

operations. With regular BSTs, performing different operations on identical elements will often give you very different trees. However, as proven in lecture, the treap on a given set of keys with priorities is unique, so we get the same tree no matter how we arrive at it.

5 I should be treapin' like a lg

In lecture, we saw a formal analysis showing that the expected depth of a treap on n elements with random priorities is $O(\lg n)$. This analysis is in the lecture notes. Today, we'll see two informal justifications of this analysis.

5.1 Justification 1: It works for Quicksort

Observe that the final treap after inserting all of the elements is identical (removing priorities) to the tree we got by inserting all of the elements in order from lowest to highest priorities (look at the order of inserts in the tree example, and the hash function h. Yeah, we planned that.) Just as choosing random pivots in Quicksort is equivalent to assigning each element a random priority and choosing as the pivot the element with the highest priority, inserting elements into a treap with hash values as priorities is equivalent, in terms of the resulting tree, to inserting elements into a BST in priority order. This gives us the same expected performance guarantees: the highest priority element (which becomes the root) is likely to split the elements approximately in half, resulting in an approximately balanced tree. Moreover, while we can get unlucky, since we are (essentially) randomizing the insert order, no adversary, however devious, can present a set of keys and insertion order that gives the treap worst-case performance, as one could easily do with ordinary BSTs.

5.2 Justification 2 - Entropy (not the store)

Treaps don't guarantee balance, they are just balanced in expectation. We can make arbitrarily unbalanced treaps, but we will show that these have low probability.

Q: What's the worst-case scenario for a treap, and what is the probability of this happening?
A: The treap forms a chain of length n. To construct a tree like this, at each level i, we need the one element at that level (the element with i^{th} highest priority) to be either the highest or lowest key not already in the tree. Thus, each priority except the lowest has two possible keys to which it can be assigned, and there are 2^{n-1} ways we can assign priorities in this way (note that the actual integer priorities don't matter, only the ordering does). This is exponential in n, which you've been trained to believe is huge. However, there are $n!$ possible orderings of priorities, so the probability of getting a chain is $\frac{2^{n-1}}{n!}$. Even for relatively small n, this is an *incredibly* small number (for $n = 10$ it is $\frac{2}{14{,}175}$.)

Figure 2 shows a few permutations of elements $a-d$ with priorities $0-3$. Only one of the permutations shown (and eight total) give a sub-optimal depth. Most permutations result in fairly balanced trees.

OK, but that's the worst case. What about the second-worst case, a depth of $n-1$? We get this by making one "mistake" in choosing either the lowest or highest keys as above. This doesn't change the probability by too much. By extending this reasoning, you can see that for large n, you have to try pretty hard to assign priorities that result in large chains: choosing random numbers will almost always result in nearly balanced trees.

For anyone with a background in physics, this is essentially saying that a balanced treap is a high-entropy state. If you're not familiar with the terminology, this means that there are many, many ways to construct a balanced treap, and few ways to construct a highly unbalanced one. Using the conventional informal terminology for entropy, balanced treaps are "disorderly" and unbalanced ones are "orderly." Recall the second law of thermodynamics: random physical processes will tend to increase total entropy. So, assigning random priorities

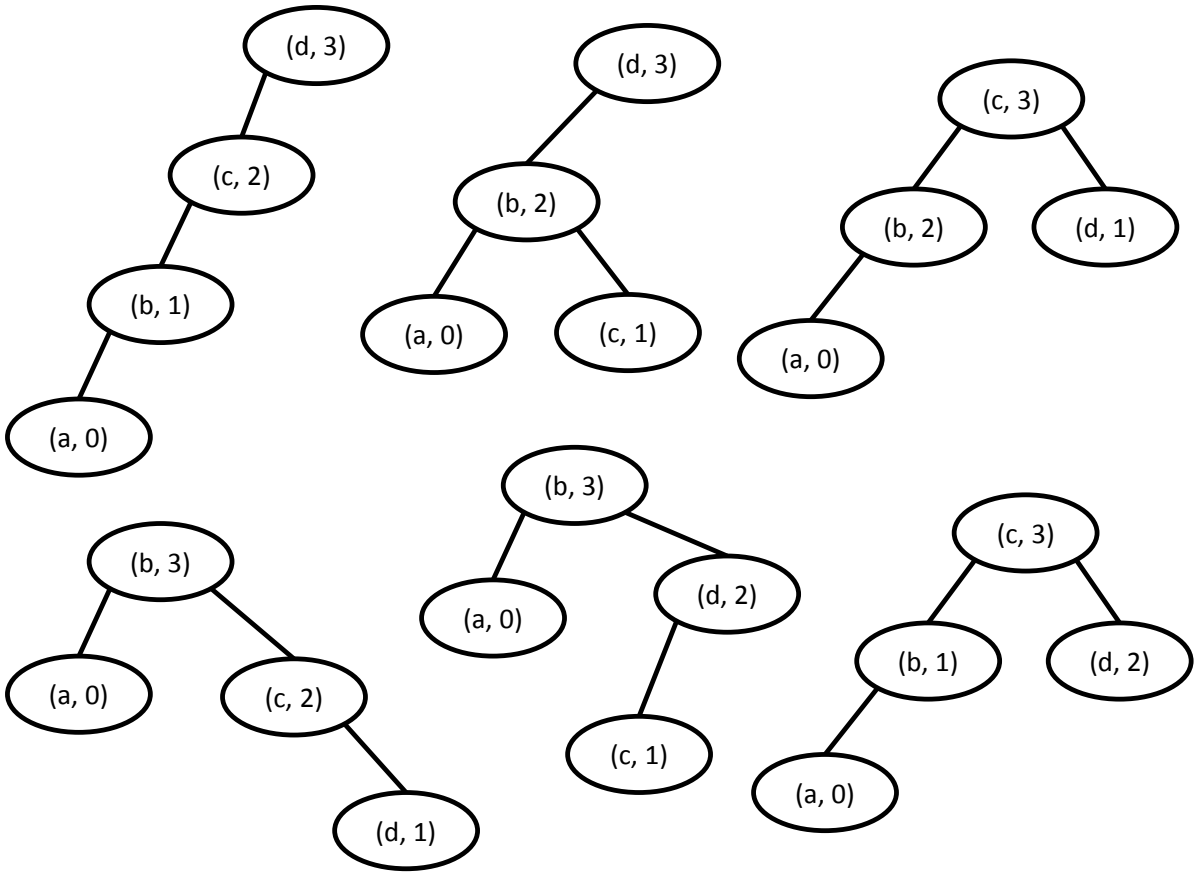


Figure 2: Six selected permutations of priorities on 4 elements

will, in the limit, keep your treaps balanced. It's not a clever optimization or trick, or a complex analysis. It's a fundamental law of the universe.