Recitation 7

Treaps and Combining BSTs

7.1 Announcements

- *FingerLab* is due **Friday afternoon**. It's worth 125 points.
- *RangeLab* will be released on **Friday**.

7.2 Deletion from a Treap

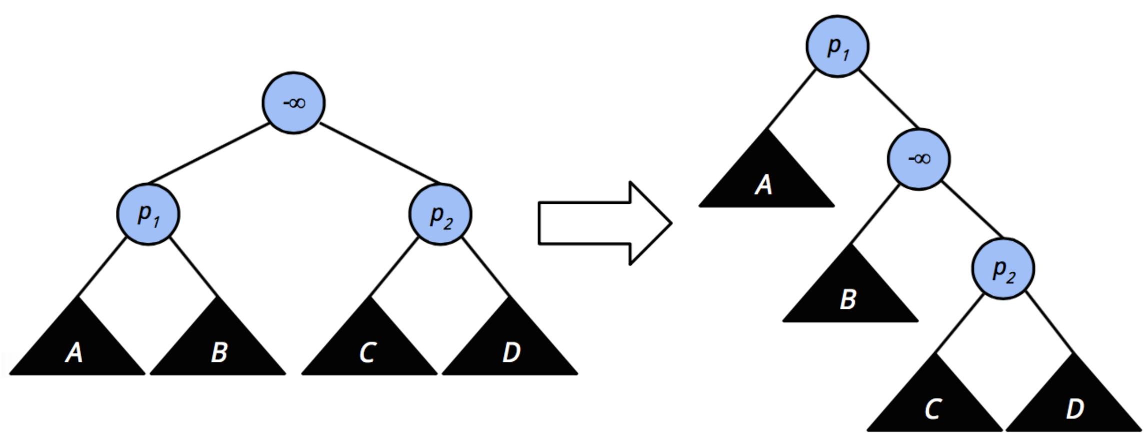
Recall that a treap is a BST with a priority function $p : U \to \mathbb{Z}$, where U is the universe of keys. You should think of p as a random number generator: for each key, it returns a random integer. A treap has two structural properties:

1. **BST invariant**: For every $\texttt{Node}(L, k, R)$, we have $\ell < k$ for every ℓ in L, and symmetrically $k < r$ for every r in R.
2. **Heap invariant**: For every $\texttt{Node}(L, k, R)$, we have that $p(k) > p(x)$ for every x in either L or R.

Consider the following strategy for deleting a key k from a treap:

1. Locate the node containing k,
2. Set the priority of k to be $-\infty$ (note that if k has children, then this breaks the heap invariant of the treap),
3. Restore the heap invariant by rotating k downwards until it has only leaves for children,
4. Delete k by replacing its node with a leaf.

A "rotation" in this case refers to the process of making one of k's children the root, depending on their relative priorities. For example, if k has two children with priorities p_1 and p_2 where $p_1 > p_2$, we rotate like so:



The case of $p_1 < p_2$ is symmetric. In turns out that this process is equivalent to calling `join` on the children of k. You should convince yourself of this.

We're interested in the following: in expectation, *how many rotations must we perform before we can delete k?*

Let's set up the specifics: we have a treap T formed from the sorted sequence of keys S, $|S| = n$. We're interested in deleting the key $S[d]$. Let T' be the same treap, except that the priority of $S[d]$ is now $-\infty$.

We need a couple indicator random variables:

$$X_j^i = \begin{cases} 1, & \text{if } S[i] \text{ is an ancestor of } S[j] \text{ in } T \\ 0, & \text{otherwise} \end{cases}$$

$$(X')_j^i = \begin{cases} 1, & \text{if } S[i] \text{ is an ancestor of } S[j] \text{ in } T' \\ 0, & \text{otherwise} \end{cases}$$

Task 7.1. *Write R_d, the number of rotations necessary to delete $S[d]$, in terms of the given random variables.*

The number of rotations is equal to the **number of nodes which aren't an ancestor of $S[d]$ in T, but are in T'**. Therefore we have

$$R_d = \sum_{i=0}^{n-1} (X')_d^i - \sum_{i=0}^{n-1} X_d^i$$

Task 7.2. *Give $\mathbf{E}\left[X_d^i\right]$ and $\mathbf{E}\left[(X')_d^i\right]$ in terms of i and d.*

We have both $X_d^i = 1$ and $(X')_d^i = 1$ if $S[i]$ has the largest priority among the $|d - i| + 1$ keys between $S[i]$ and $S[d]$. However, notice that in the latter case, we already know that the priority of $S[i]$ is larger than that of $S[d]$, unless $i = d$. So we only need that $S[i]$ is the largest among the $|d - i|$ significant keys in this range. Therefore:

$$\mathbf{E}\left[X_d^i\right] = \begin{cases} 1, & \text{if } i = d \\ \frac{1}{|d-i|+1}, & \text{otherwise} \end{cases}$$

$$\mathbf{E}\left[(X')_d^i\right] = \begin{cases} 1, & \text{if } i = d \\ \frac{1}{|d-i|}, & \text{otherwise} \end{cases}$$

Task 7.3. *Compute* $\mathbf{E}[R_d]$. *For simplicity, you may assume* $1 \leq d \leq n-2$.

$$\begin{aligned}
\mathbf{E}[R_d] &= \sum_{i=0}^{n-1} \mathbf{E}\left[(X')_d^i\right] - \sum_{i=0}^{n-1} \mathbf{E}\left[X_d^i\right] \\
&= \left(\sum_{i=0}^{d-1} \mathbf{E}\left[(X')_d^i\right] + 1 + \sum_{i=d+1}^{n-1} \mathbf{E}\left[(X')_d^i\right]\right) - \left(\sum_{i=0}^{d-1} \mathbf{E}\left[X_d^i\right] + 1 + \sum_{i=d+1}^{n-1} \mathbf{E}\left[X_d^i\right]\right) \\
&= \left(\sum_{i=0}^{d-1} \frac{1}{d-i} + \sum_{i=d+1}^{n-1} \frac{1}{i-d}\right) - \left(\sum_{i=0}^{d-1} \frac{1}{d-i+1} + \sum_{i=d+1}^{n-1} \frac{1}{i-d+1}\right) \\
&= \big(H_d + H_{n-d-1}\big) - \big(\,(H_{d+1} - 1) + (H_{n-d} - 1)\,\big) \\
&= 2 + \big(H_d - H_{d+1}\big) + \big(H_{n-d-1} - H_{n-d}\big) \\
&= 2 - \frac{1}{d+1} - \frac{1}{n-d} \\
&\leq 2
\end{aligned}$$

7.3 Generalized Combination

In lecture, we discussed `union`, and argued that it has $O\left(m \log\left(\frac{n}{m}+1\right)\right)$ work and $O(\log(n)\log(m))$ span. The latter bound can be improved to $O(\log n + \log m)$ using *futures*[1], but that is outside the scope of this course.

Let's begin by inspecting the code for `union`.

Algorithm 7.4. *BST union.*

```
fun union (T1,T2) =
  case (T1,T2) of
     (_,Leaf) => T1
  | (Leaf,_) => T2
  | (Node (L1,x,R1),_) =>
      let val (L2,_,R2) = split (T2,x)
          val (L,R) = (union (L1,L2) || union (R1,R2))
      in joinMid (L,x,R)
      end
```

What about the functions `intersection` and `difference`? These can be implemented in a similar fashion as `union`, and as such have the same cost bounds. In this recitation, we'll establish this more concretely.

Task 7.5. *Implement a helper function* `combine` *which has* $O\left(m \log\left(\frac{n}{m}+1\right)\right)$ *work and* $O(\log(n)\log(m))$ *span for BSTs of size* n *and* m, $n \geq m$. *Use* `combine` *to implement* `intersection` *and* `difference`. *Conclude that all three of the set functions have the same cost bounds.*

What do we have to change to generalize `union`? Notice that, for example, `intersection` returns `Leaf` in both base cases, while `difference` only returns `Leaf` in the second case. Next, consider that `intersection` only keeps the key x if it is also present in T_2, and `difference` specifically removes x if it is present in T_2. We can account for all of these differences by introducing new arguments which specify what to do in the base cases, and whether or not we should keep x in the recursive case (based on whether or not it is present in T_2).

[1] http://dl.acm.org/citation.cfm?id=258517

Algorithm 7.6. *Generalized BST combine.*

```
fun combine f1 f2 k =
  let
    fun combine' (T1,T2) =
      case (T1,T2) of
        (_,Leaf) => f1(T1)
      | (Leaf,_) => f2(T2)
      | (Node (L1,x,R1),_) =>
          let val (L2,y,R2) = split (T2,x)
              val (L,R) = (combine' (L1,L2) || combine' (R1,R2))
          in if k(y) then joinMid (L,x,R) else join (L,R)
          end
  in
    combine'
  end

val union =
  combine (fn T1 => T1) (fn T2 => T2) (fn y => true)

val intersection =
  combine (fn T1 => Leaf) (fn T2 => Leaf) (fn y => isSome y)

val difference =
  combine (fn T1 => T1) (fn T2 => Leaf) (fn y => not isSome y)
```

Task 7.7. *Consider a function* `symdiff` *where* $($`symdiff` $(A,B))$ *returns a BST containing all keys which are either in A or B, but not both. Implement* `symdiff` *in terms of* `combine`*.*

```
val symdiff = combine (fn T1 => T1) (fn T2 => T2) (fn y => not isSome y)
```

7.4 Additional Exercises

Exercise 7.8. *Describe an algorithm for inserting an element into a treap by "undoing" the deletion process described in Section 7.2.*

Exercise 7.9. *For treaps, suppose you are given implementations of* `find`*,* `insert`*, and* `delete`*. Implement* `split` *and* `joinMid` *in terms of these functions. You'll need to "hack" the keys and priorities; i.e., assume you can do funky things like insert a key with a* specific *priority.*

Exercise 7.10. *Given a set of key-priority pairs* $(k_i, p_i) : 0 \leq i < n$ *where all of the* k_i*'s are distinct and all of the* p_i*'s are distinct, prove that there is a unique corresponding treap* T*.*

7.4.1 Selected Solutions

Exercise 7.8.

- Implement $\texttt{split}(T, k)$ as follows. First, determine if k is present in T via `find`. Then, insert k with priority ∞ into T. The resulting treap will have the form $\texttt{Node}(L, k, R)$. We then return (L, m, R), where m was the result of the `find`.
- Implement $\texttt{joinMid}(L, k, R)$ as follows. Set $p(k) = \infty$, and then let $T = \texttt{delete}(\texttt{Node}(L, k, R), k)$. Finally, restore $p(k)$ to its correct value, and finish with $\texttt{insert}(T, k)$.

.