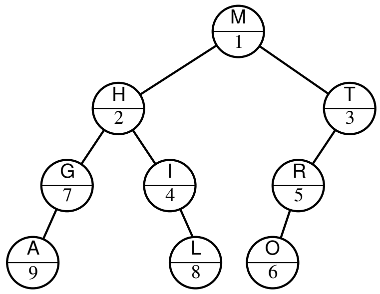
I thought the following four [rules] would be enough, provided that I made a firm and constant resolution not to fail even once in the observance of them. The first was never to accept anything as true if I had not evident knowledge of its being so. . . . The second, to divide each problem I examined into as many parts as was feasible, and as was requisite for its better solution. The third, to direct my thoughts in an orderly way. . . establishing an order in thought even when the objects had no natural priority one to another. And the last, to make throughout such complete enumerations and such general surveys that I might be sure of leaving nothing out.

— René Descartes, *Discours de la Méthode* (1637)

4 Randomized Treaps (September 17)

4.1 Treaps

In this lecture, we will consider binary trees where every internal node has both a *search key* and a *priority*. In our examples, we will use letters for the search keys and numbers for the priorities. A *treap* is a binary tree where the inorder sequence of search keys is sorted and each node's priority is smaller than the priorities of its children.[1] In other words, a treap is simultaneously a binary search tree for the search keys and a (min-)heap for the priorities.



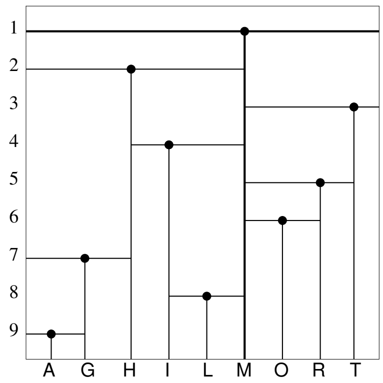
A treap. The top half of each node shows its search key and the bottom half shows its priority.

I'll assume from now on that all the keys and priorities are distinct. Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the search keys and priorities of its nodes. Since it's a heap, the node v with highest priority must be the root. Since it's also a binary search tree, any node u with $key(u) < key(v)$ must be in the left subtree, and any node w with $key(w) > key(v)$ must be in the right subtree. Finally, since the subtrees are treaps, by induction, their structures are completely determined. The base case is the trivial empty treap.

Another way to describe the structure is that a treap is exactly the binary tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the usual insertion algorithm. This is also easy to prove by induction.

A third way interprets the keys and priorities as the coordinates of a set of points in the plane. The root corresponds to a T whose joint lies on the topmost point. The T splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively. This interpretation has some interesting applications in computational geometry, which (unfortunately) we probably won't have time to talk about.

[1] Sometimes I hate English. Normally, 'higher priority' means 'more important', but 'first priority' is also more important than 'second priority'. Maybe 'posteriority' would be better; one student suggested 'unimportance'.



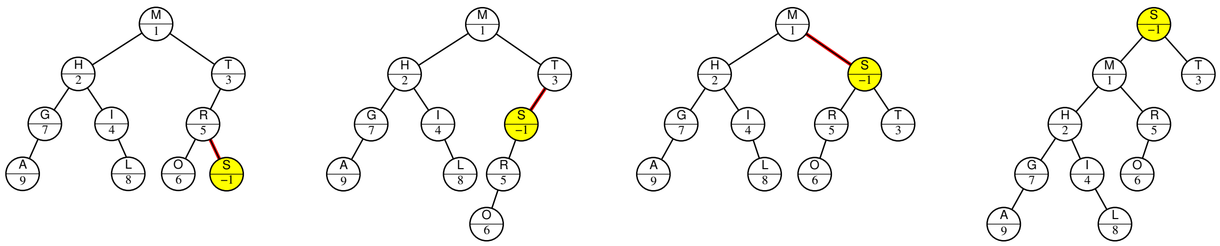
A geometric interpretation of the same treap.

Treaps were first discovered by Jean Vuillemin in 1980, but he called them *Cartesian trees*.[2] The word 'treap' was first used by Edward McCreight around 1980 to describe a slightly different data structure, but he later switched to the more prosaic name *priority search trees*.[3] Treaps were rediscovered and used to build randomized search trees by Cecilia Aragon and Raimund Seidel in 1989.[4] A different kind of randomized binary search tree, which uses random rebalancing instead of random priorities, was later discovered and analyzed by Conrado Martínez and Salvador Roura in 1996.[5]

4.2 Binary Search Tree Operations

The search algorithm is the usual one for binary search trees. The time for a successful search is proportional to the depth of the node. The time for an unsuccessful search is proportional to the depth of either its successor or its predecessor.

To insert a new node z, we start by using the standard binary search tree insertion algorithm to insert it at the bottom of the tree. At the point, the search keys still form a search tree, but the priorities may no longer form a heap. To fix the heap property, as long as z has smaller priority than its parent, perform a rotation at z. The running time is proportional to the depth of z before the rotations—we have to walk down the treap to insert z, and then walk back up the treap doing rotations. Another way to say this is that the time to insert z is roughly twice the time to perform an unsuccessful search for $key(z)$.



Left to right: After inserting $(S, 10)$, rotate it up to fix the heap property.
Right to left: Before deleting $(S, 10)$, rotate it down to make it a leaf.

[2] J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

[3] E. M. McCreight. Priority search trees. *SIAM J. Comput.* 14(2):257–276, 1985.

[4] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica* 16:464–497, 1996.

[5] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.

Deleting a node is *exactly* like inserting a node, but in reverse order. Suppose we want to delete node z. As long as z is not a leaf, perform a rotation at the child of z with smaller priority. This moves z down a level and its smaller-priority child up a level. The choice of which child to rotate preserves the heap property everywhere except at z. When z becomes a leaf, chop it off.

We sometimes also want to *split* a treap T into two treaps $T_<$ and $T_>$ along some pivot key π, so that all the nodes in $T_<$ have keys less than π and all the nodes in $T_>$ have keys bigger then π. A simple way to do this is to insert a new node z with $key(z) = \pi$ and $priority(z) = -\infty$. After the insertion, the new node is the root of the treap. If we delete the root, the left and right sub-treaps are exactly the trees we want. The time to split at π is roughly twice the time to (unsuccessfully) search for π.

Similarly, we may want to *merge* two treaps $T_<$ and $T_>$, where every node in $T_<$ has a smaller search key than any node in $T_>$, into one super-treap. Merging is just splitting in reverse—create a dummy root whose left sub-treap is $T_<$ and whose right sub-treap is $T_>$, rotate the dummy node down to a leaf, and then cut it off.

4.3 Analysis

The cost of each of these operations is proportional to the depth $d(v)$ of some node v in the treap.

- **Search:** A successful search for key k takes $O(d(v))$ time, where v is the node with $key(v) = k$. For an unsuccessful search, let v^- be the inorder *predecessor* of k (the node whose key is just barely smaller than k), and let v^+ be the inorder *successor* of k (the node whose key is just barely larger than k). Since the last node examined by the binary search is either v^- or v^+, the time for an unsuccessful search is either $O(d(v^+))$ or $O(d(v^-))$.

- **Insert/Delete:** Inserting a new node with key k takes either $O(d(v^+))$ time or $O(d(v^-))$ time, where v^+ and v^- are the predecessor and successor of the new node. Deletion is just insertion in reverse.

- **Split/Merge:** Splitting a treap at pivot value k takes either $O(d(v^+))$ time or $O(d(v^-))$ time, since it costs the same as inserting a new dummy root with search key k and priority $-\infty$. Merging is just splitting in reverse.

Since the depth of a node in a treap is $\Theta(n)$ in the worst case, each of these operations has a worst-case running time of $\Theta(n)$.

4.4 Random Priorities

A *randomized binary search tree* is a treap in which the priorities are *independently and uniformly distributed continuous random variables.* That means that whenever we insert a new search key into the treap, we generate a random real number between (say) 0 and 1 and use that number as the priority of the new node. The only reason we're using real numbers is so that the probability of two nodes having the same priority is zero, since equal priorities make the analysis messy. In practice, we could just choose random integers from a large range, like 0 to $2^{31} - 1$, or random floating point numbers. Also, since the priorities are independent, each node is equally likely to have the smallest priority.

The cost of all the operations we discussed—search, insert, delete, split, join—is proportional to the depth of some node in the tree. Here we'll see that the *expected* depth of *any* node is $O(\log n)$, which implies that the expected running time for any of those operations is also $O(\log n)$.

Let x_k denote the node with the kth smallest search key. To analyze the expected depth, we define an indicator variable

$$A_k^i = \big[x_i \text{ is a proper ancestor of } x_k\big].$$

(The superscript doesn't mean power in this case; it just a reminder of which node is supposed to be further up in the tree.) Since the depth $d(v)$ of v is just the number of proper ancestors of v, we have the following identity:

$$d(x_k) = \sum_{i=1}^{n} A_k^i.$$

Now we can express the *expected* depth of a node in terms of these indicator variables as follows.

$$\mathrm{E}[d(x_k)] = \sum_{i=1}^{n} \Pr[A_k^i = 1]$$

(Just as in our analysis of matching nuts and bolts in Lecture 3, we're using linearity of expectation and the fact that $\mathrm{E}[X] = \Pr[X = 1]$ for any indicator variable X.) So to compute the expected depth of a node, we just have to compute the probability that some node is a proper ancestor of some other node.

Fortunately, we can do this easily once we prove a simple structural lemma. Let $X(i, k)$ denote either the subset of treap nodes $\{x_i, x_{i+1}, \ldots, x_k\}$ or the subset $\{x_k, x_{k+1}, \ldots, x_i\}$, depending on whether $i < k$ or $i > k$. $X(i, k)$ and $X(k, i)$ always denote prceisly the same subset, and this subset contains $|k - i| + 1$ nodes. $X(1, n) = X(n, 1)$ contains all n nodes in the treap.

Lemma 1. *For all $i \neq k$, x_i is a proper ancestor of x_k if and only if x_i has the smallest priority among all nodes in $X(i, k)$.*

Proof: If x_i is the root, then it is an ancestor of x_k, and by definition, it has the smallest priority of *any* node in the treap, so it must have the smallest priority in $X(i, k)$.

On the other hand, if x_k is the root, then x_i is not an ancestor of x_k, and indeed x_i does not have the smallest priority in $X(i, k)$ — x_k does.

On the gripping hand[6], suppose some other node x_j is the root. If x_i and x_k are in different subtrees, then either $i < j < k$ or $i > j > k$, so $x_j \in X(i, k)$. In this case, x_i is not an ancestor of x_k, and indeed x_i does not have the smallest priority in $X(i, k)$ — x_j does.

Finally, if x_i and x_k are in the same subtree, the lemma follows inductively (or, if you prefer, recursively), since the subtree is a smaller treap. The empty treap is the trivial base case. □

Since each node in $X(i, k)$ is equally likely to have smallest priority, we immediately have the probability we wanted:

$$\Pr[A_k^i = 1] = \frac{[i \neq k]}{|k - i| + 1} = \begin{cases} \dfrac{1}{k - i + 1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \dfrac{1}{i - k + 1} & \text{if } i > k \end{cases}$$

[6]See Larry Niven and Jerry Pournelle, *The Gripping Hand*, Pocket Books, 1994.

To compute the expected depth of a node x_k, we just plug this probability into our formula and grind through the algebra.

$$\begin{aligned}
\mathrm{E}[d(x_k)] &= \sum_{i=1}^{n} \Pr[A_k^i = 1] \\
&= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\
&= \sum_{j=2}^{k} \frac{1}{j} + \sum_{i=2}^{n-k} \frac{1}{j} \\
&= H_k - 1 + H_{n-k} - 1 \\
&< \ln k + \ln(n-k) - 2 \\
&< 2\ln n - 2.
\end{aligned}$$

In conclusion, every search, insertion, deletion, split, and merge operation in an n-node randomized binary search tree takes $O(\log n)$ expected time.

Since a treap is exactly the binary tree that results when you insert the keys in order of increasing priority, a randomized treap is the result of inserting the keys in *random* order. So our analysis also automatically gives us the expected depth of any node in a binary tree built by random insertions (without using priorities).

4.5 Randomized Quicksort (Again?!)

We've already seen two completely different ways of describing randomized quicksort. The first is the familiar recursive one: choose a random pivot, partition, and recurse. The second is a less familiar iterative version: repeatedly choose a new random pivot, partition whatever subset contains it, and continue. But there's a third way to describe randomized quicksort, this time in terms of binary search trees.

```
RandomizedQuicksort:
  T ← an empty binary search tree
  insert the keys into T in random order
  output the inorder sequence of keys in T
```

Our treap analysis tells us is that this algorithm will run in $O(n \log n)$ expected time, since each key is inserted in $O(\log n)$ expected time.

Why is this quicksort? Just like last time, all we've done is rearrange the order of the comparisons. Intuitively, the binary tree is just the recursion tree created by the normal version of quicksort. In the recursive formulation, we compare the initial pivot against everything else and then recurse. In the binary tree formulation, the first "pivot" becomes the root of the tree without any comparisons, but then later as each other key is inserted into the tree, it is compared against the root. Either way, the first pivot chosen is compared with everything else. The partition splits the remaining items into a left subarray and a right subarray; in the binary tree version, these are exactly the items that go into the left subtree and the right subtree. Since both algorithms define the same two subproblems, by induction, both algorithms perform the same comparisons.

We even saw the probability $\frac{1}{|k-i|+1}$ before, when we were talking about sorting nuts and bolts with a variant of randomized quicksort. In the more familiar setting of sorting an array of numbers, the probability that randomized quicksort compares the ith largest and kth largest elements is exactly $\frac{2}{|k-i|+1}$. The binary tree version compares x_i and x_k if and only if x_i is an ancestor of x_k or vice versa, so the probabilities are exactly the same.