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)

There are those who think that life has nothing left to chance
A host of holy horrors to direct our aimless dance

— Rush, "Freewill", *Permanent Waves* (1980), lyrics by Neal Peart

What is luck?
Luck is probability taken personally.
It is the excitement of bad math.

— Penn Jillette (2001), quoting Chip Denman (1998)

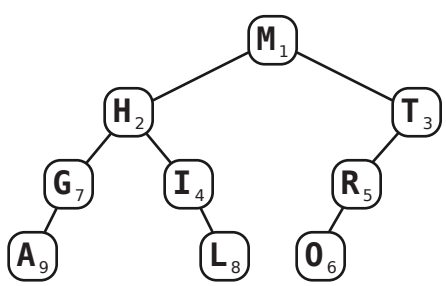
3 Randomized Binary Search Trees

In this lecture, we consider two randomized alternatives to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.

3.1 Treaps

3.1.1 Definitions

A *treap* is a binary tree in which every node has both a *search key* and a *priority*, 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. In our examples, we will use letters for the search keys and numbers for the priorities.



A treap. Letters are search keys; numbers are priorities.

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

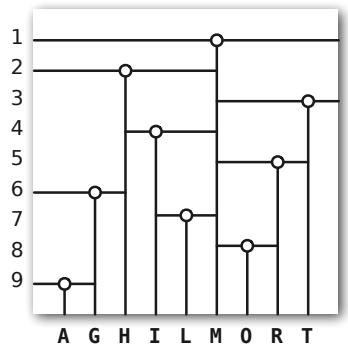
[1] Sometimes I hate English. Normally, 'higher priority' means 'more important', but 'first priority' is also more important than 'second priority'. Maybe 'posteriority' would be better; one student suggested 'unimportance'.

© Copyright 2020 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

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 search tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm. This characterization is also easy to prove by induction.

A third description 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 won't have time to talk about.



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]

3.1.2 Treap 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

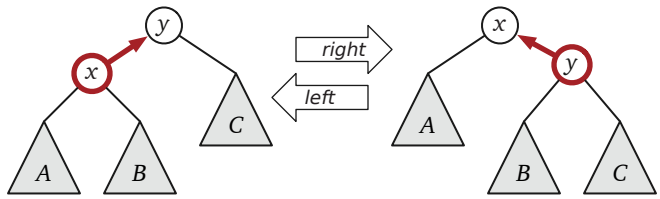
[2] J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

[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.

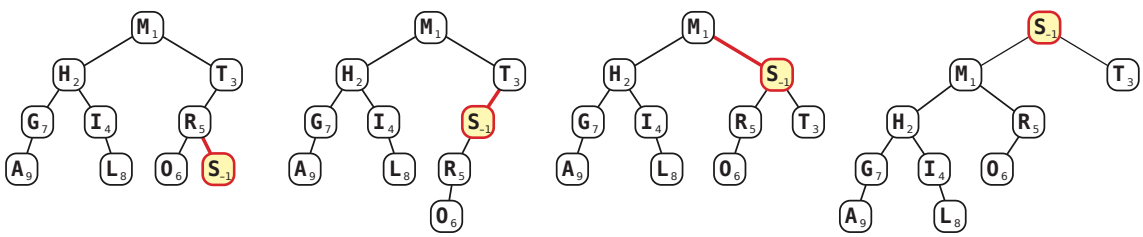
[5] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.

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, a local operation that decreases the depth of z by one and increases its parent's depth by one, while maintaining the search tree property. Rotations can be performed in constant time, since they only involve simple pointer manipulation.



A right rotation at x and a left rotation at y are inverses.

The overall time to insert z 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 with priority -1, rotate it up to fix the heap property.
Right to left: Before deleting S, rotate it down to make it a leaf.

To delete a node, we just run the insertion algorithm backward in time. 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 *join* 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.

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

- **Search:** A successful search for key k takes $O(depth(v))$ 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 $O(depth(v^+))$ or $O(depth(v^-))$.

- **Insert/Delete:** Inserting a new node with key k takes either $O(depth(v^+))$ time or $O(depth(v^-))$ time, where v^+ and v^- are the predecessor and successor of the new node. Deletion is insertion in reverse.
- **Split/Join:** Splitting a treap at pivot value k takes either $O(depth(v^+))$ time or $O(depth(v^-))$ time, because the split is identical to inserting a new dummy root with search key k and priority $-\infty$. Merging is splitting in reverse.

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

3.1.3 Random Priorities

A *randomized treap* is a treap in which the priorities are *independently and uniformly distributed continuous random variables*. Whenever we insert a new search key into the treap, we independently generate a real number (say) uniformly at random between 0 and 1 and use that number as the priority of the new node. The precise distribution is unimportant, as long as the same distribution is used for all nodes, and the probability that two nodes have equal priorities is zero. (Equal priorities make the analysis slightly messier; in practice, we can choose random integers from a large range, like 0 to $2^{31}-1$, and break ties arbitrarily; occasional ties have almost no practical effect on the performance of the data structure.) 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 simplify notation, let us write $\boldsymbol{i \uparrow k}$ (read "i above k") to mean that x_i is a proper ancestor of x_k. By definition, the depth of v is the number of proper ancestors of v, so we can write

$$depth(x_k) = \sum_{i=1}^{n} [i \uparrow k].$$

(Again, we're using Iverson bracket notation.) Now we can express the *expected* depth of a node in terms of these indicator variables as follows.

$$\mathrm{E}[depth(x_k)] = \sum_{i=1}^{n} E\big[[i \uparrow k]\big] = \sum_{i=1}^{n} \Pr[i \uparrow k]$$

(Just as in our analysis of matching nuts and bolts, we're using linearity of expectation and the fact that $\mathrm{E}[X] = \Pr[X = 1]$ for any zero-one variable X; in this case, $X = [i \uparrow k]$.) So to compute the expected depth of a node, we only need to compute the probabilities that one node is a proper ancestor of another.

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 $i > k$. The order of the arguments is unimportant; the subsets $X(i,k)$ and $X(k,i)$ are identical. The subset $X(1,n) = X(n,1)$ contains all n nodes in the treap.

Lemma 1. *For all $i \neq k$, we have $i \uparrow k$ if and only if x_i has the smallest priority among all nodes in $X(i,k)$.*

Proof: There are four cases to consider.

If x_i is the root, then $i \uparrow k$, 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 $k \uparrow i$, so $i \not\uparrow k$. Moreover, 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, we have both $i \not\uparrow k$ and $k \not\uparrow i$, and 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 from the inductive hypothesis (or, if you prefer, the Recursion Fairy), because 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:

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

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

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

In conclusion, every search, insertion, deletion, split, and join 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).

3.1.4 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 (generic, off-the-shelf, standard) binary search trees.

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

```
RandomizedQuicksort(A[1..n]):
  T ← an empty binary search tree
  for i ← 1 to n in random order
      insert A[i] into T
  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.

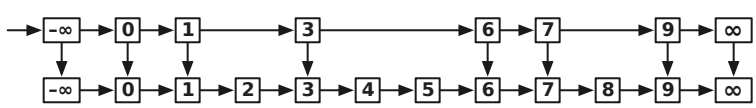
Why is this quicksort? As in our analysis of Rawlins nuts and bolts algorithm, all we've done is rearrange the *order* of the comparisons. The binary search tree ***is*** the recursion tree created by the normal version of quicksort. In the recursive formulation, we first compare the initial pivot against 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 other keys are inserted into the tree, each new key is compared against the root. Either way, the first pivot chosen is compared with every other key. 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. Because both algorithms define the same two subproblems, by induction, both algorithms perform the same comparisons within those subproblems.

We've even the probability $1/(|k-i|+1)$ before in the analysis of Rawlins' aglorithm. 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 $2/(|k-i|+1)$. The binary tree version of quicksort compares x_i and x_k if and only if either $i \uparrow k$ or $k \uparrow i$, so the probabilities are exactly the same.

3.2 Skip Lists

Skip lists, first discovered by Bill Pugh in the late 1980's,[7] have many of the usual desirable properties of balanced binary search trees, but their superficial structure is very different.

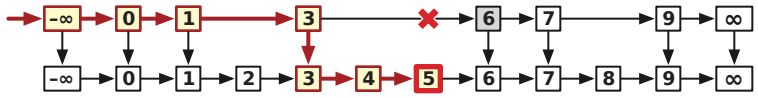
At a high level, a skip list is just a sorted linked list with some random shortcuts. To do a search in a normal singly-linked list of length n, we obviously need to look at n items in the worst case. To speed up this process, we can make a second-level list that contains roughly half the items from the original list. Specifically, for each item in the original list, we duplicate it with probability $1/2$. We then string together all the duplicates into a second sorted linked list, and add a pointer from each duplicate back to its original. Just to be safe, we also add sentinel nodes at the beginning and end of both lists.



A linked list with some randomly-chosen shortcuts.

Now we can find a value x in this augmented structure using a two-stage algorithm. First, we scan for x in the shortcut list, starting at the $-\infty$ sentinel node. If we find x, we're done. Otherwise, we reach some value bigger than x and we know that x is not in the shortcut list. Let w be the largest item less than x in the shortcut list. In the second phase, we scan for x in the original list, starting from w. Again, if we reach a value bigger than x, we know that x is not in the data structure.

[7] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33(6):668–676, 1990.

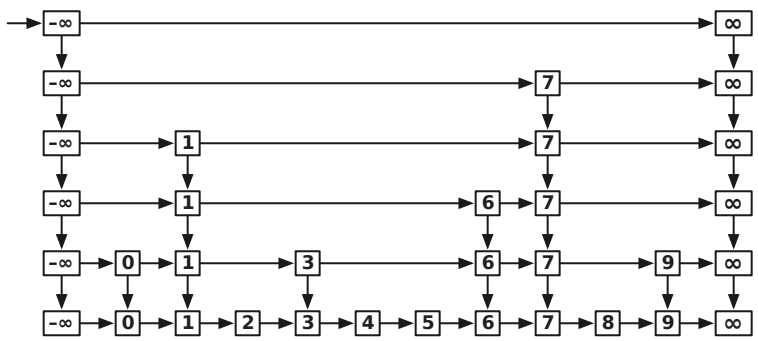


Searching for 5 in a list with shortcuts.

Since each node appears in the shortcut list with probability $1/2$, the expected number of nodes examined in the first phase is at most $n/2$. Only one of the nodes examined in the second phase has a duplicate. The probability that any node is followed by k nodes without duplicates is 2^{-k}, so the expected number of nodes examined in the second phase is at most $1 + \sum_{k \geq 0} 2^{-k} = 2$. Thus, by adding these random shortcuts, we've reduced the cost of a search from n to $n/2 + 2$, roughly a factor of two in savings.

3.2.1 Recursive Random Shortcuts

Now there's an obvious improvement—add shortcuts to the shortcuts, and repeat recursively. That's exactly how skip lists are constructed. For each node in the original list, we repeatedly flip a coin until we get tails. Each time we get heads, we make a new copy of the node. The duplicates are stacked up in levels, and the nodes on each level are strung together into sorted linked lists. Each node v stores a search key $key(v)$, a pointer $down(v)$ to its next lower copy, and a pointer $right(v)$ to the next node in its level.

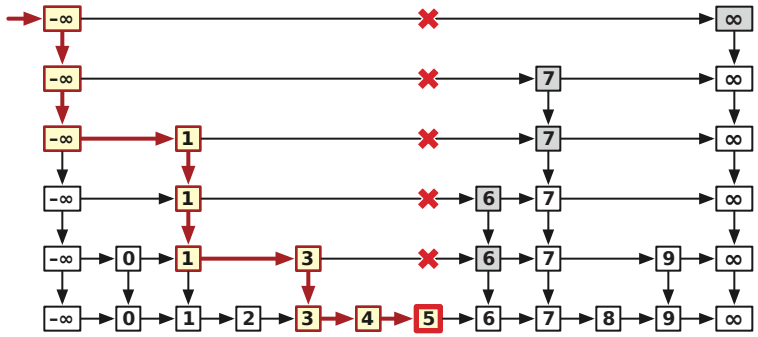


A skip list is a linked list with recursive random shortcuts.

The search algorithm for skip lists is very simple. Starting at the leftmost node L in the highest level, we scan through each level as far as we can without passing the target value x, and then proceed down to the next level. The search ends when we either reach a node with search key x or fail to find x on the lowest level.

```
SkipListFind(x, L):
  v ← L
  while (v ≠ Null and key(v) ≠ x)
      if key(right(v)) > x
          v ← down(v)
      else
          v ← right(v)
  return v
```

Intuitively, since each level of the skip lists has about half the number of nodes as the previous level, the total number of levels should be about $O(\log n)$. Similarly, each time we add another



Searching for 5 in a skip list.

level of random shortcuts to the skip list, we cut the search time roughly in half, except for a constant overhead, so $O(\log n)$ levels should give us an overall expected search time of $O(\log n)$. Let's formalize each of these two intuitive observations.

3.2.2 Number of Levels

The actual values of the search keys don't affect the skip list analysis, so let's assume the keys are the integers 1 through n. Let $L(x)$ be the number of levels of the skip list that contain some search key x, not counting the bottom level. Each new copy of x is created with probability $1/2$ from the previous level, essentially by flipping a coin. We can compute the expected value of $L(x)$ recursively—with probability $1/2$, we flip tails and $L(x) = 0$; and with probability $1/2$, we flip heads, increase $L(x)$ by one, and recurse:

$$E[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2}\big(1 + E[L(x)]\big)$$

Solving this equation gives us $\boldsymbol{E[L(x)] = 1}$.

In order to analyze the expected worst-case cost of a search, however, we need a bound on the *number of levels* $L = \max_x L(x)$. Unfortunately, we can't compute the average of a maximum the way we would compute the average of a sum. Instead, we derive a stronger result: ***The depth of a skip list storing n keys is $O(\log n)$ with high probability.*** "High probability" is a technical term that means the probability is at least $1 - 1/n^c$ for some constant $c \geq 1$; the hidden constant in the $O(\log n)$ bound could depend on c.

In order for a search key x to appear on level ℓ, it must have flipped ℓ heads in a row when it was inserted, so $\mathbf{Pr[L(x) \geq \ell] = 2^{-\ell}}$. The skip list has at least ℓ levels if and only if $L(x) \geq \ell$ for at least one of the n search keys.

$$\Pr[L \geq \ell] = \Pr\big[(L(1) \geq \ell) \vee (L(2) \geq \ell) \vee \cdots \vee (L(n) \geq \ell)\big]$$

Using the ***union bound*** — $\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$ for any random events A and B — we can simplify this as follows:

$$\Pr[L \geq \ell] \;\leq\; \sum_{x=1}^{n} \Pr[L(x) \geq \ell] \;=\; n \cdot \Pr[L(x) \geq \ell] \;=\; \frac{n}{2^\ell}.$$

When $\ell \leq \lg n$, this bound is trivial. However, for any constant $c > 1$, we have a strong upper bound

$$\Pr[L \geq c \lg n] \leq \frac{1}{n^{c-1}}.$$

We conclude that ***with high probability, a skip list has $O(\log n)$ levels.***

This high-probability bound indirectly implies a bound on the *expected* number of levels. Some simple algebra gives us the following alternate definition for expectation:

$$\mathrm{E}[L] = \sum_{\ell \geq 0} \ell \cdot \Pr[L = \ell] = \sum_{\ell \geq 1} \Pr[L \geq \ell]$$

Clearly, if $\ell < \ell'$, then $\Pr[L(x) \geq \ell] > \Pr[L(x) \geq \ell']$. So we can derive an upper bound on the expected number of levels as follows:

$$\begin{aligned}
\mathrm{E}[L(x)] = \sum_{\ell \geq 1} \Pr[L \geq \ell] &= \sum_{\ell=1}^{\lg n} \Pr[L \geq \ell] \;+ \sum_{\ell \geq \lg n + 1} \Pr[L \geq \ell] \\
&\leq \sum_{\ell=1}^{\lg n} 1 \;+ \sum_{\ell \geq \lg n + 1} \frac{n}{2^\ell} \\
&= \lg n + \sum_{i \geq 1} \frac{1}{2^i} && [i = \ell - \lg n] \\
&= \mathbf{\lg n + 2}
\end{aligned}$$

So in expectation, a skip list has *at most two* more levels than an ideal version where each level contains exactly half the nodes of the next level below. Notice that this is an *additive* penalty over a perfectly balanced structure, as opposed to treaps, where the expected depth is a constant *multiple* of the ideal $\lg n$.

3.2.3 Logarithmic Search Time

It's a little easier to analyze the cost of a search if we imagine running the algorithm backwards. SkipListFind takes the output from SkipListFind as input and traces back through the data structure to the upper left corner. Skip lists don't really have up and left pointers, but we'll pretend that they do so we don't have to write '$v \to \mathit{down}(v)$' or '$v \to \mathit{right}(v)$'.[8]

```
SkipListFind(v):
  while (level(v) ≠ L)
      if up(v) exists
          v ← up(v)
      else
          v ← left(v)
```

Now for *every* node v in the skip list, $up(v)$ exists with probability $1/2$. So for purposes of analysis, SkipListFind is equivalent to the following algorithm:

```
FlipWalk(v):
  while (v ≠ L)
      if CoinFlip = Heads
          v ← up(v)
      else
          v ← left(v)
```

[8] Leonardo da Vinci wrote all his notes using mirror-writing, but not because he wanted to keep his discoveries secret. He just had really bad arthritis in his right hand!

Obviously, the expected number of heads is exactly the same as the expected number of TAILS. Thus, the expected running time of this algorithm is twice the expected number of upward jumps. But we already know that the number of upward jumps is $O(\log n)$ with high probability. It follows the running time of FLIPWALK is $O(\log n)$ with high probability (and therefore in expectation).

Exercises

1. Prove that a treap is exactly the binary search tree that results from inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm.

2. Consider a treap T with n vertices. As in the notes above, identify nodes in T by the ranks of their search keys; thus, 'node 5' means the node with the 5th smallest search key. Let i, j, and k be integers such that $1 \leq i \leq j \leq k \leq n$.

   (a) Prove that the expected number of proper descendants of any node in a treap is exactly equal to the expected depth of that node.

   (b) The *left spine* of a binary tree is a path starting at the root and following only left-child pointers. What is the expected number of nodes in the left spine of T?

   (c) What is the expected number of leaves in T? *[Hint: What is the probability that node k is a leaf?]*

   (d) What is the expected number of nodes in T with two children?

   (e) What is the expected number of nodes in T with exactly one child?

   *(f) What is the expected number of nodes in T with exactly one *grandchild*?

   (g) Define the *priority rank* of a node in T to be one more than the number of nodes with smaller priority. For example, the root of T always has priority rank 1, and one of the children of the root has priority rank 2. What is the expected priority rank of node i?

   (h) What is the expected priority rank of the left child of the root (given that such a node exists)?

   *(i) What is the expected priority rank of the leftmost grandchild of the root (given that such a node exists)?

   *(j) What is the expected priority rank of a node with depth d?

   (k) What is the *exact* probability that node j is a common ancestor of node i and node k?

   (l) What is the *exact* expected length of the unique path in T from node i to node k?

   (m) What is the expected (key) rank of the leftmost leaf in T?

   (n) What is the expected (key) rank of the leftmost node in T with two children (given that such a node exists)?

   (o) What is the probability that T has no nodes with two children?

3. Let X be a set of n real numbers. For any two numbers $a \leq z$, let $[a, z]$ denote the closed interval $\{a \leq x \leq z \mid x \in \mathbb{R}\}$. An *interval emptiness query* over the set X asks, given two real numbers $a \leq z$, whether the subset $X \cap [a, z]$ is empty.

(a) Suppose X is stored (as the search keys) in a randomized treap. Describe an algorithm to answer an emptiness query for any interval $[a,z]$ in $O(1+\log(n/w))$ time, where $w = 1+|X\cap[a,z]|$. In particular, if $X\cap[a,z]=\varnothing$, your query algorithm should run in $O(\log n)$ time, and if $X\subseteq[a,z]$, your query algorithm should run in $O(1)$ time.

(b) Consider a *weighted* version of randomized treaps where each search key k comes with a positive integer *weight* $w(k)$, and the priority of k is defined as the minimum of $w(k)$ independent real random numbers.

Prove that the expected depth of the node storing any key k is at most $1+2\ln(\frac{W}{w(k)})$, where $W=\sum_i w(i)$ is the sum of the weights of all nodes. *[Hint: Use the identity* $x/y \le -\ln(1-\frac{x}{y}) = \ln y - \ln(y-x)$*, which holds for all* $0\le x<y$*.]*

(c) Suppose X is stored in a skip list. Describe an algorithm to answer an emptiness query for any interval $[a,z]$ in $O(1+\log(n/w))$ time, where $w = 1+|X\cap[a,z]|$. In particular, if $X\cap[a,z]=\varnothing$, your query algorithm should run in $O(\log n)$ time, and if $X\subseteq[a,z]$, your query algorithm should run in $O(1)$ time.

(d) Now consider a *weighted* version of skip lists, where each search key $k\in X$ comes with a positive integer *weight* $w(k)$. To determine the number of lists that contain a given key $k\in X$, we essentially insert $w(k)$ independent copies of k into the skip list, but then keep only most successful copy. More formally, each search key k at each level of the weighted skip list survives to the next level with probability $1-2^{-w(k)}$.

Prove that the expected time to find any key k in a weighted skip list is $O(1+\log(\frac{W}{w(k)}))$, where $W=\sum_i w(i)$ is the sum of the weights of all search keys.

4. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A ***heater*** is a priority search tree in which the *priorities* are given by the user, and the *search keys* are distributed uniformly and independently at random in the real interval $[0,1]$. Intuitively, a heater is a sort of anti-treap.[9]

The following problems consider an n-node heater T whose priorities are the integers from 1 to n. We identify nodes in T by their *priorities*; thus, 'node 5' means the node in T with *priority* 5. For example, the min-heap property implies that node 1 is the root of T. Finally, let i and j be integers with $1\le i<j\le n$.

(a) Prove that in a random permutation of the $(i+1)$-element set $\{1,2,\ldots,i,j\}$, elements i and j are adjacent with probability $2/(i+1)$.

(b) Prove that node i is an ancestor of node j with probability exactly $2/(i+1)$. *[Hint: Use part (a)!]*

(c) What is the exact probability that node i is a *descendant* of node j? *[Hint: Don't use part (a)!]*

(d) What is the *exact* expected depth of node j?

(e) Describe and analyze an algorithm to insert a new item into a heater. Express the expected running time of the algorithm in terms of the rank of the newly inserted item.

(f) Describe an algorithm to delete the minimum-priority item (the root) from an n-node heater. What is the expected running time of your algorithm?

[9] If you choose not to decide, you still have made a choice.

5. Prove the following basic facts about skip lists, where n is the number of keys.
   (a) The expected number of nodes is $O(n)$.
   (b) The number of nodes is $O(n)$ with high probability.
   (c) A new key can be inserted in $O(\log n)$ time with high probability.
   (d) A key can be deleted in $O(\log n)$ time with high probability.

6. Suppose we are given two skip lists, one storing a set A of m keys, and the other storing a set B of n keys. Describe and analyze an algorithm to merge these into a single skip list storing the set $A \cup B$ in $O(n+m)$ expected time. Do *not* assume that every key in A is smaller than every key in B; the two sets could be arbitrarily intermixed. *[Hint: Do the obvious thing.]*

7. Any skip list $\mathcal{L}$ can be transformed into a binary search tree $T(\mathcal{L})$ as follows. The root of $T(\mathcal{L})$ is the leftmost node on the highest non-empty level of $\mathcal{L}$; the left and right subtrees are constructed recursively from the nodes to the left and to the right of the root. Let's call the resulting tree $T(\mathcal{L})$ a *skip list tree*.
   (a) Show that any search in $T(\mathcal{L})$ is no more expensive than the corresponding search in $\mathcal{L}$. (Searching in $T(\mathcal{L})$ could be *considerably* cheaper—why?)
   (b) Describe an algorithm to insert a new search key into a skip list tree in $O(\log n)$ expected time. Inserting key x into the tree $T(\mathcal{L})$ should produce *exactly* the same tree as inserting x into the skip list $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree. *[Hint: You need to maintain some additional information in the tree nodes.]*
   (c) Describe an algorithm to delete a search key from a skip list tree in $O(\log n)$ expected time. Again, deleting key x from $T(\mathcal{L})$ should produce *exactly* the same tree as deleting x from $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree.

8. Consider the following "loose" variant of treaps. Instead of generating priorities uniformly at random, for each node, we flip an independent fair coin until it comes up heads, and define the priority of the node to be the number of flips. Thus, for every positive integer k, a node has priority k with probability 2^{-k}. In addition, we invert the heap property, by requiring that the priority of any node is *not* larger than the priority of its parent.

   This method creates many nodes with the same priority; in particular, about half the nodes will have priority 1. Thus, a single set of search keys and priorities may be consistent with many different loose treaps.

   Prove that the expected depth of any node in an n-node "loose treap" is $O(\log n)$. To be pedantic, the expectation is over the random choice of priorities, but for each choice of priorities, we consider the worst possible loose treap.

   [Hint: This is almost exactly the same as the previous question.]

*9. In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0,1]$. In practice, however, computers have access only to random *bits*. This problem asks you to analyze an implementation of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1..\infty]$ of random bits, which is interpreted as the rational number

$$priority(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or "reveal") new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LargerPriority(v, w):
  for i ← 1 to ∞
      if i > ℓ_v
          ℓ_v ← i;  π_v[i] ← RandomBit
      if i > ℓ_w
          ℓ_w ← i;  π_w[i] ← RandomBit
      if π_v[i] > π_w[i]
          return v
      else if π_v[i] < π_w[i]
          return w
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LargerPriority during these insertions.

(a) Prove that $E[L] = \Theta(n)$.

(b) Prove that $E[\ell_v] = \Theta(1)$ for any node v. *[Hint: This is equivalent to part (a). Why?]*

(c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. *[Hint: Why doesn't this contradict part (b)?]*

10. A ***meldable priority queue*** stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- MakeQueue: Return a new priority queue containing the empty set.
- FindMin(Q): Return the smallest element of Q (if any).
- DeleteMin(Q): Remove the smallest element in Q (if any).
- Insert(Q, x): Insert element x into Q, if it is not already there.
- DecreaseKey(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
- Delete(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
- Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. Meld can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1

  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2

  with probability 1/2
      left(Q1) ← MELD(left(Q1), Q2)
  else
      right(Q1) ← MELD(right(Q1), Q2)

  return Q1
```

(a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: What is the expected length of a random root-to-leaf path in an n-node binary tree, where each left/right choice is made with equal probability?]*

(b) Prove that MELD(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

(c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

*11. Our probabilistic analysis of treaps and skip lists assumes that the sequence of operations used to query and update the data structures are independent of the random choices used to build the data structure. However, if a malicious adversary has additional information about the data structure, he can force significantly worse performance.

(a) Let $\pi[0..n^2]$ be an arbitrary permutation of the numbers $0, 1, 2, \ldots, n^2$. For each index i, define two integers:

- a_i is the length of the longest increasing subsequence of π that ends with $\pi[i]$.
- b_i be the length of the longest decreasing subsequence of π that starts with $\pi[i]$.

Argue that the $n^2 + 1$ ordered pairs (a_i, b_i) are all distinct.

(b) Prove the ***Erdős-Szekeres Theorem***: Any permutation $\pi[0..n^2]$ of the numbers $0, 1, 2, \ldots, n^2$ contains either an increasing subsequence of length $n+1$ or a decreasing subsequence of length $n+1$.

(c) Suppose after constructing a skip list with n items, the adversary discovers the number of levels that contain each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum search time to be $\Omega(\sqrt{n})$.

(d) Suppose after constructing a skip list with n items, the adversary discovers the number of levels that contain each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum search time to be $\Omega(n)$ *with high probability*.

(e) Suppose after constructing a treap with n items, the adversary discovers the random priorities of each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum depth of the treap to be $\Omega(\sqrt{n})$.

© Copyright 2020 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.