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)

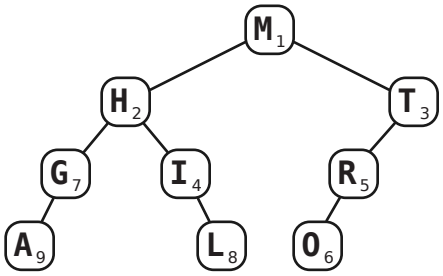
3 Randomized Binary Search Trees

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.

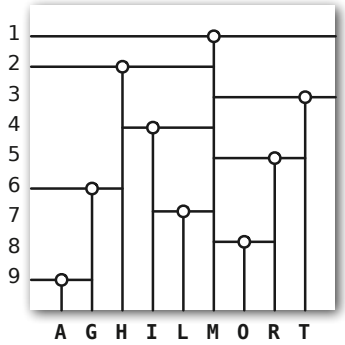
I'll assume from now on that all the keys and priorities are distinct. Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the

[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'.

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

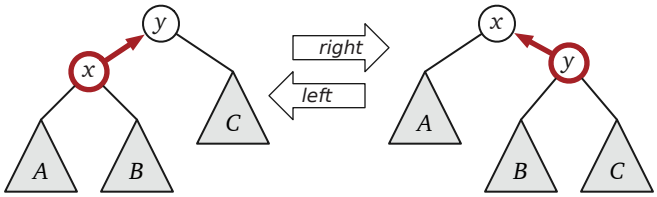
[2] J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

[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.

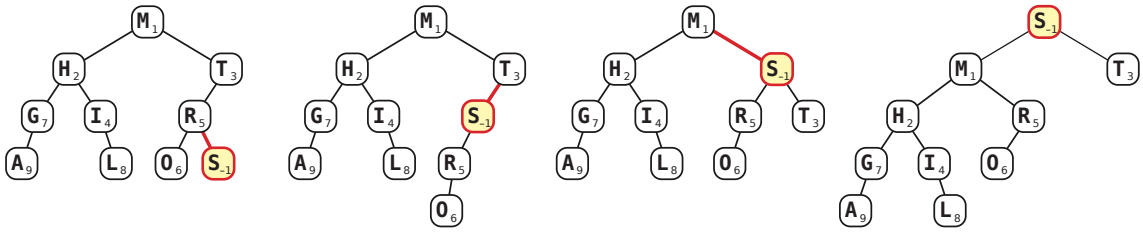
[5] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.

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 just insertion in reverse.
- **Split/Join:** Splitting a treap at pivot value k takes either $O(depth(v^+))$ time or $O(depth(v^-))$ 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.

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 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 slightly messier. (In practice, we could just 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. Since the depth of v is just the number of proper ancestors of v, we have the following identity:

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

(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 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 $i > k$. 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 just 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 binary search trees.

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

```
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.

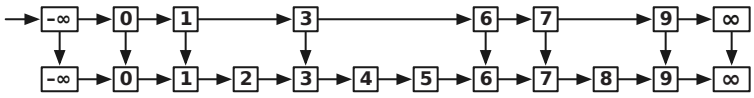
Why is this quicksort? Just like last time, all we've done is rearrange the order of the comparisons. Intuitively, the binary tree is just the recursion tree created by the normal version of quicksort. In the recursive formulation, we compare the initial pivot against everything else and then recurse. 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 $1/(|k-i|+1)$ 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 $2/(|k-i|+1)$. The binary tree version of quicksort compares x_i and x_k if and only if $i \uparrow k$ 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.

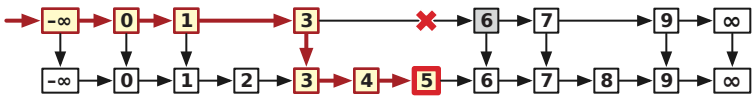


A linked list with some randomly-chosen shortcuts.

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.

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

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

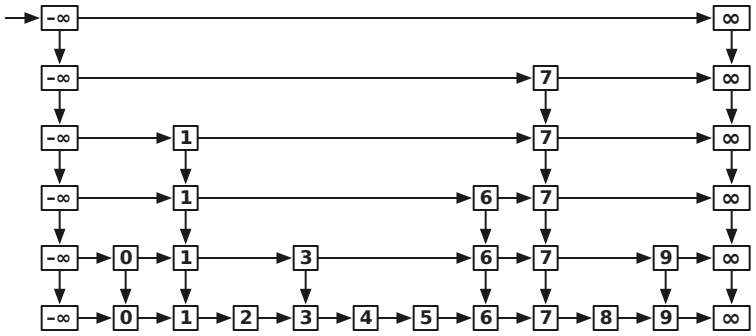


Searching for 5 in a list with shortcuts.

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.

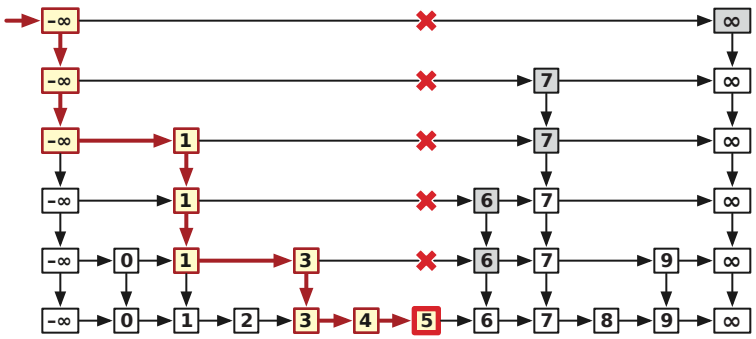


A skip list is a linked list with recursive random shortcuts.

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 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.



Searching for 5 in a skip list.

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 $\boldsymbol{\Pr[L(x) \geq \ell] = 2^{-\ell}}$. 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:

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

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 '$v \rightarrow$ down' or '$v \rightarrow$ right'.[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 $1 \le i \le j \le k \le n$.

   (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. 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*

Casus ubique valet; semper tibi pendeat hamus:
Quo minime credas gurgite, piscis erit.
[Luck affects everything. Let your hook always be cast.
Where you least expect it, there will be a fish.]

— Publius Ovidius Naso [Ovid], *Ars Amatoria*, Book III (2 AD)

There is no sense being precise
when you don't even know what you're talking about.

— Attributed to John von Neumann

6 Filtering and Streaming

The randomized algorithms and data structures we have seen so far *always* produce the correct answer but have a small probability of being slow. In this lecture, we will consider randomized algorithms that are always fast, but return the wrong answer with some small probability.[1] More generally, we are interested in tradeoffs between the (likely) efficiency of the algorithm and the (likely) quality of its output.

Specifically, we introduce an ***error rate*** δ and analyze the running time required to guarantee the output is correct with probability $1-\delta$. For "high probability" correctness, we need $\delta < 1/n^c$ for some constant c. In practice, it may be sufficient (or even necessary) to set δ to a small constant; for example, setting $\delta = 1/1000$ means the algorithm produces correct results at least 99.9% of the time.

6.1 Bloom Filters

Bloom filters are a natural variant of hashing proposed by Burton Bloom in 1970 as a mechanism for supporting membership queries in sets. In strict accordance with Stigler's Law of Autonomy, Bloom filters are identical to ***Zatocoding***, a coding system for library cards developed by Calvin Mooers in 1947. (Mooers was the person who coined the phrase "information retrieval".) A probabilistic analysis of Zatocoding appears in the personal notes of cybernetics pioneer W. Ross Ashby from 1960.

A Bloom filter (or Zatocode) for a set X of n items from some universe $\mathcal{U}$ allows one to test whether a given item $x \in \mathcal{U}$ is an element of X. Of course we can already do this with hash tables in $O(1)$ expected time, using $O(n)$ space. Bloom (and Mooers) observed that by allowing false positives—occasionally reporting $x \in X$ when in fact $x \notin X$—we can still answer queries in $O(1)$ expected time using considerably less space. False positives make Bloom filters unsuitable as an exact membership data structure, but because of their speed and low false positive rate, they are commonly used as filters or sanity checks for more complex data structures.

A Bloom filter consists of an array $B[0..m-1]$ of bits, together with k hash functions $h_1, h_2, \ldots, h_k : \mathcal{U} \to \{0, 1, \ldots, m-1\}$. For purposes of theoretical analysis, we assume the hash functions h_i are mutually independent, ideal random functions. This assumption is of course unsupportable in practice, but may be necessary to guarantee theoretical performance. Unlike many other types of hashing, nobody knows whether the same theoretical guarantees can be achieved using practical hash functions with more limited independence.[2] Fortunately, the actual

[1] Some textbooks (and Wikipedia) use the terms "Las Vegas" and "Monte Carlo" algorithms to respectively describe these two types of randomized algorithms. Most algorithms researchers don't.

[2] PhD thesis, anyone?

real-world behavior of Bloom filters appears to be consistent with this unrealistic theoretical analysis.

A Bloom filter for a set $X = \{x_1, x_2, \ldots, x_n\}$ is initialized by setting the bit $B[h_j(x_i)]$ to 1 for all indices i and j. Because of collisions, some bits may be set more than once, but that's fine.

```
MakeBloomFilter(X):
  for h ← 0 to m − 1
      B[h] ← 0
  for i ← 1 to n
      for j ← 1 to k
          B[h_j(x_i)] ← 1
  return B
```

Given a new item y, the Bloom filter determines whether $y \in X$ by checking each bit $B[h_j(y)]$. If any of those bits is 0, the Bloom filter correctly reports that $y \notin X$ However, if all bits are 1, the Bloom filter reports that $y \in X$, although this is not necessarily correct.

```
BloomMembership(B, y):
  for j ← 1 to k
      if B[h_j(y)] = 0
          return False
  return Maybe
```

One nice feature of Bloom filters is that the various hash functions h_i can be evaluated in parallel on a multicore machine.

6.2 False Positive Rate

Let's estimate the probability of a false positive, as a function of the various parameters n, m, and k. For all indices h, i, and j, we have $\Pr[h_j(x_i) = h] = 1/m$, so ideal randomness gives us

$$Pr[B[h] = 0] \;=\; \left(1 - \frac{1}{m}\right)^{kn} \;\approx\; e^{-kn/m}$$

for every index h. Using this exact probability is rather unwieldy; to keep things sane, we will use the close approximation $\boldsymbol{p := e^{-kn/m}}$ instead.

The expected number of 0-bits in the Bloom filter is approximately mp; moreover, Chernoff bounds imply that the number of 0-bits is close to mp with very high probability. Thus, the probabililty of a false positive is very close[3] to

$$(1-p)^k = (1 - e^{-kn/m})^k.$$

If all other parameters are held constant, then the false positive rate increases with n (the number of items) and decreases with m (the number of bits). The dependence on k (the number of hash functions) is a bit more complicated, but we can derive the best value of k for given n and m as follows. Consider the logarithm of the false-positive rate:

$$\ln\left((1-p)^k\right) = k\ln(1-p) = -\frac{m}{n}\ln p \ln(1-p).$$

By symmetry, this expression is minimized when $\boldsymbol{p = 1/2}$. We conclude that, the optimal number of hash functions is $\boldsymbol{k = \ln 2 \cdot (m/n)}$, which would give us the false positive rate

[3]This analysis, originally due to Bloom, assumes that certain events are independent even though they are not; as a result, the estimate given here is slightly below the true false positive rate.

$(1/2)^{\ln 2(m/n)} \approx (0.61850)^{m/n}$. Of course, in practice, k must be an integer, so we cannot achieve precisely this rate, but we can get reasonably close (at least if $m \gg n$).

Finally, the previous analysis implies that we can achieve any desired false positive rate $\delta > 0$ using a Bloom filter of size

$$m = \left\lceil \frac{\lg(1/\delta)}{\ln 2} n \right\rceil = \Theta(n \log(1/\delta))$$

that uses with $k = \lceil \lg(1/\delta) \rceil$ hash functions. For example, we can achieve a 1% false-positive rate using a Bloom filter of size $10n$ bits with 7 hash functions; in practice, this is *considerably* fewer bits than we would need to store all the elements of S explicitly. With a $32n$-bit table (equivalent to one integer per item) and 22 hash functions, we get a false positive rate of just over $2 \cdot 10^{-7}$.

★★★ Deletions via counting filters? Key-value pairs via Bloomier filters? Other extensions?

6.3 Streaming Algorithms

A ***data stream*** is an extremely long sequence S of items from some universe $\mathcal{U}$ that can be read only once, in order. Good examples of data streams include the sequence of packets that pass through a network router, the sequence of searches at google.com, the sequence of all bids on the New York Stock Exchange, and the sequence of humans passing through the Shinjuku Railway Station in Tokyo. Standard algorithms are not appropriate for data streams; there is simply too much data to store, and it arrives too quickly for any complex computations.

A ***streaming algorithm*** processes each item in a data stream stream as it arrives, maintaining some summary information in a local data structure. The basic structure of every streaming algorithm is the following:

```
DoSomething(S):
  ⟨⟨initialize⟩⟩
  while S is not done
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  return ⟨⟨something⟩⟩
```

Ideally, neither the running time per item nor the space used by the data structure depends on the overall length of the stream; viewed as algorithms in the traditional sense, all streaming algorithms run in constant time! Somewhat surprisingly, even within this very restricted model, it is possible to compute interesting properties of the stream using randomization, provided we are willing to tolerate some errors in the output.

6.4 The Count-Min Sketch

As an example, consider the following problem: At any point during the stream, we want to estimate the number of times that an arbitrary item $x \in \mathcal{U}$ has appeared in the stream so far. This problem can be solved with a variant of Bloom filters called the ***count-min sketch***, first published by Graham Cormode and S. Muthu Muthukrishnan in 2005.

The count-min sketch consists of a $w \times d$ array of counters (all initially zero) and d hash functions $h_1, h_2, \ldots, h_d : \mathcal{U} \to [m]$, drawn independently and uniformly at random from a 2-uniform family of hash functions. Each time an item x arrives in the stream, we call CMIncrement(x). Whenever we want to estimate the number of occurrences of an item x so far, we call CMEstimate(x).

```
CMIncrement(x):
  for i ← 1 to d
      j ← h_i(x)
      Count[i, j] ← Count[i, j] + 1
```

```
CMEstimate(x):
  est ← ∞
  for i ← 1 to d
      j ← h_i(x)
      est ← min{est, Count[i, j]}
  return est
```

If we set $w := \lceil e/\varepsilon \rceil$ and $d := \lceil \ln(1/\delta) \rceil$, then the data structure uses $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ space and processes updates and queries in $O(\log \frac{1}{\delta})$ time.

Let f_x be the true frequency (number of occurrences) of x, and let $\hat{f}_x$ be the value returned by CMEstimate. It is easy to see that $f_x \le \hat{f}_x$; we can never return a value smaller than the actual number of occurrences of x. We claim that $\mathbf{Pr}[\hat{f}_x > f_x + \varepsilon N] < \delta$, where N is the total length of the stream. In other words, our estimate is never too small, and with high probability, it isn't a significant overestimate either. (Notice that the error here is additive; the estimates or truly infrequent items may be much larger than their true frequencies.)

For any items $x \neq y$ and any index j, we define an indicator variable $\boldsymbol{X_{i,x,y} = [h_i(x) = h_i(y)]}$; because the hash functions h_i are universal, we have

$$\mathrm{E}[X_{i,x,y}] = \Pr[h_i(x) = h_i(y)] = \frac{1}{w}.$$

Let $\boldsymbol{X_{i,x} := \sum_{y \neq x} X_{i,x,y} \cdot f_y}$ denote the total number of collisions with x in row i of the table. Then we immediately have

$$Count[i, h_i(x)] \;=\; f_x + X_{i,x} \;\ge\; f_x.$$

On the other hand, linearity of expectation implies

$$\mathrm{E}[X_{i,x}] \;=\; \sum_{y \neq x} \mathrm{E}[X_{i,x,y}] \cdot f_y \;=\; \frac{1}{w} \sum_{y \neq x} f_y \;\le\; \frac{N}{w}.$$

Now Markov's inequality implies

$$\begin{aligned}
\Pr[\hat{f}_x > f_x + \varepsilon N] &= \Pr[X_{i,x} > \varepsilon N \text{ for all i}] && \text{[definition]} \\
&= \Pr[X_{1,x} > \varepsilon N]^d && \text{[independence of } h_i\text{'s]} \\
&\le \left(\frac{\mathrm{E}[X_{1,x}]}{\varepsilon N} \right)^d && \text{[Markov's inequality]} \\
&\le \left(\frac{N/w}{\varepsilon N} \right)^d = \left(\frac{1}{w\varepsilon} \right)^d && \text{[derived earlier]}
\end{aligned}$$

Now setting $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$ gives us $\Pr[\hat{a}_x > a_x + \varepsilon N] \le (1/e)^{\ln(1/\delta)} = \delta$, as claimed.

6.5 Estimating Distinct Items

★★★

Write this, possibly as a simpler replacement for the count-min sketch. AMS estimator: $2^{z+1/2}$ where $z = \max\{zeros(x) \mid x \in S\}$? Or stick with the Flajolet-Martin/Bar-Yossef-et-al estimator in the exercises? Median amplification.

Estimating larger moments?