Lecture IX
RANDOM SEARCH TREES

Recall the concept of binary search trees from Lecture III. This lecture focuses on a class of random binary search trees called **random treaps**. Basically, treaps are search trees whose shape is determined by the assignment of **priorities** to each key. If the priorities are chosen randomly, the result is a random treap with expected height of $\Theta(\log n)$. Why is expected $\Theta(\log n)$ height interesting when worst case $\Theta(\log n)$ is achievable? One reason is that randomized algorithms are usually simpler to implement. Roughly speaking, they do not have to "work too hard" to achieve balance, but can rely on randomness as an ally.

The analysis of random binary search trees has a long history. The analysis of binary search trees under random insertions only was known for a long time. It had been an open problem to analyze the expected behavior of binary search trees under insertions *and deletions*. A moment reflection will indicate the difficulty of formulating such a model. As an indication of the state of affairs, a paper *A trivial algorithm whose analysis isn't* by Jonassen and Knuth [2] analyzed the random insertion and deletion of a binary tree containing no more than 3 items at any time. The currently accepted probabilistic model for random search trees is due to Aragon and Seidel [1] who introduced the treap data-structure.[1] Prior to the treap solution, Pugh [3] introduced a simple but important data structure called **skip list** whose analysis is similar to treaps. The real breakthrough of the treap solution lies in its introduction of a new model for randomized search trees, thus changing[2] the ground rules for analyzing random trees.

Notations. We write $[i..j]$ for the set $\{i, i+1, \ldots, j-1, j\}$ where $i \leq j$ are integers. Often, the set of keys is $[1..n]$. If u is an item, we write $u.\texttt{Key}$ and $u.\texttt{Priority}$ for the key and priorities associated with u. Since we usually confuse an item with the node it is stored in, we may also write $u.\texttt{Parent}$ or $u.\texttt{leftChild}$. By convention, that $u.\texttt{Parent} = u$ if u is the root and $u.\texttt{leftChild} = u$ if u has no left child.

§1. Skip Lists

It is instructive to first look at the skip list data structure. Pugh [3] gave experimental evidence that its performance is comparable to non-recursive AVL trees algorithms, and superior to splay trees (the experiments use 2^{16} integer keys).

Suppose we want to maintain an ordered list L_0 of keys $x_1 < x_2 < \cdots < x_n$. We shall construct a **hierarchy** of sublists

$$L_0 \supseteq L_1 \supseteq \cdots \supseteq L_{m-1} \supseteq L_m$$

where L_m is only empty list in this hierarchy. Each L_{i+1} is a *random sample* of L_i in the following sense: each key of L_i is put into L_{i+1} with probability $1/2$. The hierarchy stops the first time the list becomes empty. In storing these lists, we shall add an artificial key $-\infty$ at the head of each list. Let L_i' be the list L_i augmented with $-\infty$. It is sufficient to store the lists L_i' as a linked list with $-\infty$ as the head (a singly-linked list is sufficient). In addition, corresponding keys in two consecutive levels shares a two-way link (called the up- and down-links, respectively. This is illustrated in figure 1.

Clearly, the expected length of L_i is $\texttt{E}[|L_i|] = n2^{-i}$. Let us compute the expected height $\texttt{E}[m]$. The probability of any key of L_0 appearing in level $i \geq 0$ is $1/2^i$. Now $m \geq i+1$ iff some key x_j appears level i.

[1] The name "treap" comes from the fact that these are binary search trees with the heap property. It was originally coined by McCreight for a different data structure.

[2] Another historical example in the same category is Alexander the Great's solution to the Gordan Knot problem.

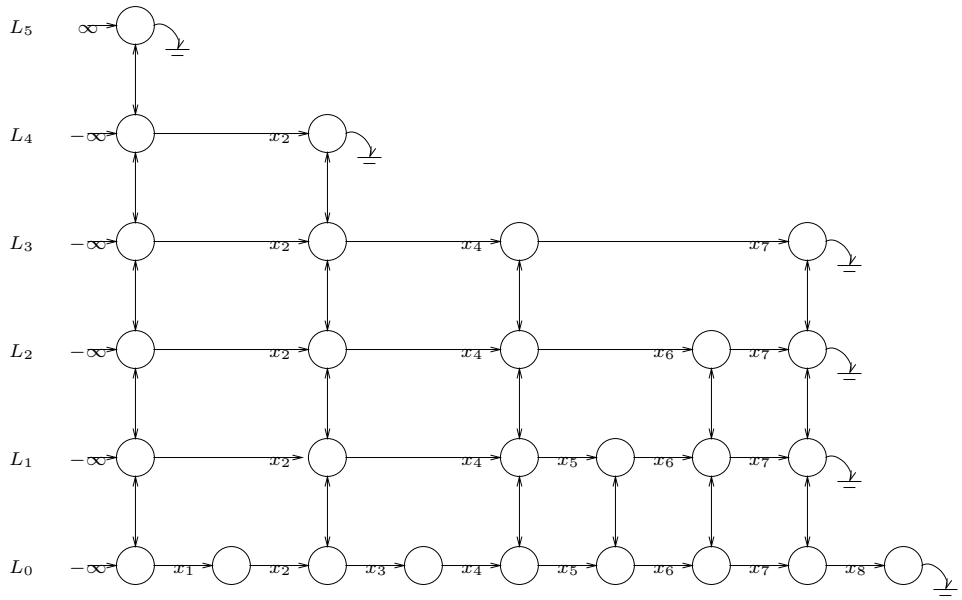

Figure 1: A skip list on 8 keys.

Since there are n choices for j, we have

$$\Pr\{m \geq i+1\} \leq \frac{n}{2^i}.$$

Hence

$$\begin{array}{rcll} \mathtt{E}[m] & = & \sum_{i\geq 1} i \Pr\{m = i\} & \\ & = & \sum_{i\geq 1} \Pr\{m \geq i\} & \text{(standard trick)} \\ & = & \sum_{i\leq \lceil \lg n\rceil} \Pr\{m \geq i\} + \sum_{j>\lceil \lg n\rceil} \Pr\{m \geq j\} & \text{(another one!)} \\ & \leq & \lceil \lg n\rceil + \sum_{j>\lceil \lg n\rceil} \frac{n}{2^{j-1}} & \\ & \leq & \lceil \lg n\rceil + \sum_{j>\lceil \lg n\rceil} \frac{1}{2^{j-1-\lg n}} & \\ & \leq & \lceil \lg n\rceil + 2. & \end{array}$$

Have we really achieved anything special? You rightly think: why not just **deterministically** omit every other item in L_i to form L_{i+1}? then $m \leq \lceil \lg n\rceil$ with less fuss! The reason why the probabilistic solution is superior is because the analysis will hold up even in the presence of insertion or deletion of arbitrary items. The deterministic solution may look very bad for particular sequence of deletions, for instance (how?). The critical idea is that the probabilistic behavior of each item x is *independent* of the other items in the list: it is determined by its own sequence of coin tosses to decide whether to promote x to the next level.

Analysis of Lookup. Our algorithm to lookup a key k in a skip list returns the largest key k_0 in L_0 such that $k_0 \leq k$. If k is less than the smallest key x_1 in L_0, we return the special value $k_0 = -\infty$.

You may also be interested in finding the smallest key k_1 in L_0 such that $k_1 \geq k$. But k_1 is either k_0 or the successor of k_0 in L_0. If k_0 has no successor, then we imagine "$+\infty$" as its successor.

In general, define k_i $(i = 0, \ldots, m)$ to be the largest key in level L_i' that is less than or equal to k. So $k_m = -\infty$. In general, given k_{i+1} in level $i+1$, we can find k_i by following a down-link and then "scanning forward" in search of k_i. The search ends after the scan in list L_0: we end up with the element k_0. Let us call the sequence of nodes visited from k_m to k_0 the **scan-path**. Let s be the random variable denoting the

length of the scan-path. If s_i is the number of nodes of the scan-path that that lies in L'_i, then we have

$$s = \sum_{i=0}^{m} s_i.$$

The expected cost of looking up key k is $\Theta(\mathtt{E}[s])$.

Lemma 1 $\mathtt{E}[s] < 2\lceil \lg n \rceil + 4$.

Before presenting the proof, it is instructive to give a wrong argument: since $\mathtt{E}[m] \leq \ln n + 3$, and $\mathtt{E}[s_i] < 2$ (why?), we have $\mathtt{E}[s] \leq 2(\ln n + 2)$. This would be correct if each s_i is i.i.d. and also independent of m (see §VII.5). Unfortunately, s_i is not independent of m (e.g., $s_i = 0$ iff $m < i$).

Proof of lemma 1. Following Pugh, we introduce the random variable

$$s(\ell) = \sum_{i=0}^{\ell-1} s_i$$

where $\ell \geq 0$ is any constant (the threshhold level). By definition, $s(0) = 0$. We note that

$$s \leq s(\ell) + |L_\ell| + (m - \ell + 1).$$

To see this, note that the edges in the scan-path at levels ℓ and above involve at most $|L_\ell|$ horizontal edges, and at most $m - \ell$ vertical (down-link) edges.

Choose $\ell = \lceil \lg n \rceil$. Then $\mathtt{E}[|L_\ell|] = n/2^\ell \leq 1$. Also $\mathtt{E}[m - \ell + 1] \leq 3$ since $\mathtt{E}[m] \leq \lceil \lg n \rceil + 2$. The next lemma will imply $\mathtt{E}[s(\ell)] \leq 2\lceil \lg n \rceil$. Combining these bounds yields lemma 1.

A Random Walk. To bound $s(\ell)$, we introduce a random walk on the lattice $\mathbb{N} \times \mathbb{N}$.

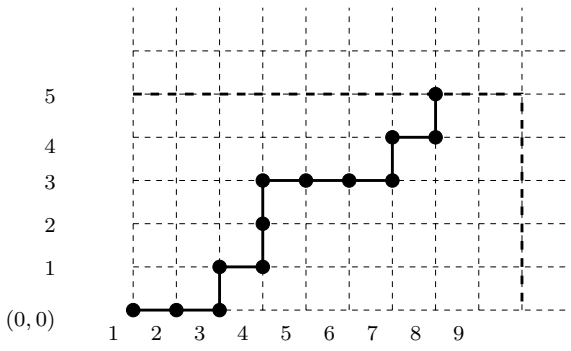

Figure 2: Random walk with $X_{9,5} = 12$.

Let u_t denote the position at time $t \in \mathbb{N}$. We begin with $u_0 = (0, 0)$, and we terminate at time t if $u_t = (x, y)$ with either $x = n$ or $y = \ell$. At each discrete time step, we can move from the current position (x, y) to $(x, y+1)$ with probability p $(0 < p < 1)$ or to $(x+1, y)$ with probability $q = 1 - p$. So p denotes the probability of "promotion" to the next level. Let $X_{n,\ell}$ denoting the number of steps in this random walk at termination. For $p = 1/2$, we obtain the following connection with $s(\ell)$:

$$\mathtt{E}[X_{n,\ell}] \geq \mathtt{E}[s(\ell)]. \tag{1}$$

To see this, imagine walking along the scan-path of a lookup, but in the reverse direction. We start from key k_0. At each step, we get to move up to the next level with probability p or to scan backwards one step with probability $q = 1 - p$. There is a difference, however, between a horizontal step in our random walk and a backwards scan of the scan-path. In the random walk, each step is unit length, but the backwards scan may take a step of larger lengths with some non-zero probability. But this only means that $X_{n,\ell} \succeq s(\ell)$ (denoting stochastic domination, §VII.3). Hence (1) is an inequality instead of an equality. For example, in the skip list of figure 1, if $k_0 = x_8$ then $s(\ell) = s(4) = 7$ while $X_{n,\ell} = X_{8,4} = 10$.

Lemma 2 *For all $\ell \geq 0$, $\mathtt{E}[X_{n,\ell}] \leq \ell/p$.*

Proof. Note that this bound removes the dependence on n. Clearly $\mathtt{E}[X_{n,0}] = 0$, so assume $\ell \geq 1$. Consider what happens after the first step: with probability p, the random walk moves to the next level, and the number of steps in rest of the random walk is distributed as $X_{n,\ell-1}$; similarly, with probability $1 - p$, the random walk remain in the same level and the number of steps in rest of the random walk is distributed as $X_{n-1,\ell}$. This gives the recurrence

$$\mathtt{E}[X_{n,\ell}] = 1 + p \cdot \mathtt{E}[X_{n,\ell-1}] + q \cdot \mathtt{E}[X_{n-1,\ell}].$$

Since $\mathtt{E}[X_{n-1,\ell}] \leq \mathtt{E}[X_{n,\ell}]$, we have

$$\mathtt{E}[X_{n,\ell}] \leq 1 + p \cdot \mathtt{E}[X_{n,\ell-1}] + q \cdot \mathtt{E}[X_{n,\ell}].$$

This simplifies to

$$\mathtt{E}[X_{n,\ell}] \leq \frac{1}{p} + \mathtt{E}[X_{n,\ell-1}].$$

This implies $\mathtt{E}[X_{n,\ell}] \leq \frac{\ell}{p}$ since $\mathtt{E}[X_{n,0}] = 0$. **Q.E.D.**

If $\ell = \lceil \lg n \rceil$ and $p = 1/2$ then

$$\mathtt{E}[s(\ell)] \leq \mathtt{E}[X_{n,\ell}] \leq 2 \lceil \lg n \rceil ,$$

as noted in the proof of lemma 1.

We leave as an exercise to the reader to specify, and to analyze, algorithms for insertion and deletion in skip lists.

Exercises

Exercise 1.1:
(i) The expected space for a skip list on n keys is $\mathcal{O}(n)$.
(ii) Describe an algorithm for insertion and analyze its expected cost.
(iii) Describe an algorithm for deletion and analyze its expected cost. □

Exercise 1.2: The above (abstract) description of skip lists allows the possibility that $L_{i+1} = L_i$. So, the height m can be arbitrarily large.
(i) Pugh suggests some á priori maximum bound on the height (say, $m \leq k \lg n$) for some small constant k. Discuss the modifications needed to the algorithms and analysis in this case.
(ii) Consider another simple method to bound m: if $L_{i+1} = L_i$ should happen, we simply omit the last key in L_i from L_{i+1}. This implies $m \leq n$. Re-work the above algorithms and complexity analysis for this approach. □

Exercise 1.3: Determine the exact formula for $\mathtt{E}[X_{n,\ell}]$. □

Exercise 1.4: We have described skip lists in which each key is promoted to the next level with probability $p = 1/2$. Give reasons for choosing some other $p \neq 1/2$. □

§2. Treaps

In our treatment of treaps, we assume that each item is associated with a *priority* as well as the usual *key*. A *treap* T on a set of items is a binary search tree with respect to the keys of items, and a max-heap with respect to the priorities of items. Recall that a tree is a max-heap with respect to the priorities if the priority at any node is greater than the priorities in its descendents.

Suppose the set of keys and set of priorities are both $[1..n]$. Let the priority of key k be denoted $\sigma(k)$. We assume σ is a permutation of $[1..n]$. We explicitly describe σ by listing the keys, in order of decreasing priority:

$$\sigma = (\sigma^{-1}(n), \sigma^{-1}(n-1), \ldots, \sigma^{-1}(1)). \tag{2}$$

We sometimes call σ a *list* in reference to this representation. In general, if the range of σ is not in $[1..n]$, we may define analogues of (2): a list of all the keys is called a σ-**list** if the priorities of the keys in the list are non-increasing. So σ-lists are unique if σ assigns unique keys.

Example. Let $\sigma = (5, 2, 1, 7, 3, 6, 4)$. So the key 5 has highest priority of 7 and key 4 has priority 1. The corresponding treap is given by figure 3 where the key values are written in the nodes.

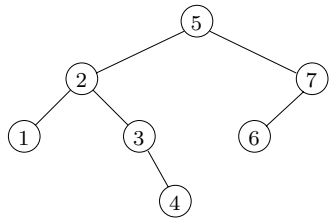

Figure 3: Treap $T_7(\sigma)$ where $\sigma = (5, 2, 1, 7, 3, 6, 4)$.

■

There is no particular relation between the key and the priority of an item. Informally, a *random treap* is a treap whose items are assigned priorities in some random manner.

Fact 1 *For any set S of items, there is a treap whose node set is S. If the keys and priorities are unique for each item, the treap is unique.*

Proof. The proof is constructive: given S, pick the item u_0 with highest priority to be the root of a treap. The remaining items are put into two subsets that make up the left and right subtrees, respectively. The construction proceeds inductively. If keys and priorities are unique then u_0 and the partition into two subsets are unique. **Q.E.D.**

Left paths and left spines. The *left path* at a node u_0 is the path $(u_0, \ldots, u_m)$ where each u_{i+1} is the left child of u_i and the last node u_m has no left child. The node u_m is called the *tip* of the left path. The *left spine* at a node u_0 is the path $(u_0, u_1, \ldots, u_m)$ where $(u_1, \ldots, u_m)$ is the right path of the left child of u_0. If u_0 has no left child, the left spine is the trivial path (u_0) of length 0. The *right path* and *right spine* is similarly defined. The *spine height* of u is the sum of the lengths of the left and right spines of u. As usual, by identifying the root of a tree with the tree, we may speak of the left path, right spine, etc, of a tree. Note that the smallest item in a binary search tree is at the tip of the left path.

EXAMPLE: In figure 3, the left and right paths at node 2 are $(2, 1)$ and $(2, 3, 4)$, respectively. The left spine of the tree $(5, 2, 3, 4)$. The spine height of the tree is $3 + 2 = 5$.

§3. Almost-treap and Treapification

We define a binary search tree T to be an *almost-treap at* u ($u \in T$) if T is not a treap, but we can adjust the priority of u so that T becomes a treap. Roughly speaking, this means the heap property is satisfied everywhere except at u. Thus if we start with a treap T, we can change the priority of any node u to get an almost-treap at u. We call an almost-treap an *under-treap* if the priority of u must be adjusted upwards to get a treap, and an *over-treap* otherwise.

Defects. Suppose T is an over-treap at u. Then there is a unique maximal prefix π of the path from u to the root such that the priority of each node in π is less than the priority of u. The number of nodes in π is called the *defect* in T. Note that u is not counted in this prefix.

Next suppose T is an under-treap at u. Then there are unique maximal prefixes of the left and right spines at u such that the priority of each node in these prefixes is greater than the priority of u. The total number of nodes in these two prefixes is called the *defect* in T.

A pair (u, v) of nodes is a *violation* if either u is an ancestor of v but u has lower priority than v, or u is a descendent of v but with higher priority. It is not hard to check that that number of violations in an over-treap is equal to the defect. But this is not true in the case of an under-treap at u: this is because we only count violations along the two spines of u. We make some simple observations.

Fact 2 *Let T be an almost-treap at u with $k \geq 1$ defects. Let v be a child of u with priority at least that of its sibling (if any).*

(a) If T is an over-treap then rotating at u produces[3] *an over-treap with $k - 1$ defects.*

(b) If T is an over-treap, v is defined and and v.`Priority` $<$ *u.*`Priority` *then rotate(v) produces an over-treap with $k + 1$ defects.*

(c) If T be an under-treap and v is defined then a rotation at v results in an under-treap at u with $k - 1$ defects.

We now give a simple algorithm to convert an almost-treap into a treap. The argument to the algorithm is the node u if the tree is an almost-treap at u. It is easy to determine from u whether the tree is a treap or an under-treap or an over-treap.

[3] If $k = 1$, the result is actually a treap. But we shall accept the terminology "almost-treap with 0 defects" as a designation for a treap.

```
Algorithm Treapify(u, T):
Input: T is an almost-treap at u.
Output: a treap T.
    if u.Priority > u.Parent.Priority
        then Violation=OverTreap elseViolation=UnderTreap.
    switch(Violation)
        case OverTreap:
            do rotate(u) until
                u.Priority ≤ u.Parent.Priority.
        case UnderTreap:
            let v be the child of u with the largest priority.
            while u.Priority < v.Priority do
                rotate(v).
                v ← the child of u with largest priority.
```

We let the reader verify the following observation:

Fact 3 *Let T be an almost-treap at u.*
a) (Correctness) Treapify(u) converts T into a treap.
b) If T is an under-treap, then number of rotations is at most the spine height of u.
c) If T is an over-treap, the number of rotations is at most the depth of u.

Exercises

Exercise 3.1: Let $n = 5$ and $\sigma = (3, 1, 5, 2, 4)$ *i.e.*, $\sigma(3) = 5$ and $\sigma(4) = 1$. Draw the treap. Next, change the priority of key 4 to 10 (from 1) and treapify. Next, change the priority of key 3 to 0 (from 5) and treapify. □

Exercise 3.2: Start with a treap T. Compare treapifying at u after we increase the priority of u by two different amounts. Is it true that the treapifying work is of the smaller increase is no more than the work for the larger increase? What if we decrease the priority of u instead? □

§4. Operations on Treaps

We show the remarkable fact that all the usual operations on binary search trees can be unified within a simple framework based on treapification.

In particular, we show how implement the following binary search tree operations:

(a) Lookup(Key,Tree)→Item,
(b) Insert(Item,Tree),
(c) Delete(Node,Tree),
(d) Successor(Item,Tree)→Item,
(e) Min(Tree)→Item,
(f) DeleteMin(Tree)→Item,
(g) Split(Tree1,Key)→Tree2,
(h) Join(Tree1,Tree2).

The meaning of these operations are defined as in the case of binary search trees. So the only twist is to simultaneously maintain the properties of a max-heap with respect to the priorities. First note that the usual binary tree operations of Lookup(Key,Tree), Successor(Item,Tree), Min(Tree) and Max(Tree) can be extended to treaps without change since these do not depend modify the tree structure. Next consider insertion and deletion.

(i) Insert(u, T): we first insert in T the item u ignoring its priority. The result is an almost-treap at u. We now treapify T at u.

(ii) Delete(v, T): assuming no item in T has priority $-\infty$, we first change the priority of the item at node v into $-\infty$. Then we treapify the resulting under-treap at v. As a result, v is now a leaf which we can delete directly.

We can implement DeleteMin(Tree) using Min(Tree) and Delete(item,Tree). It is interesting to observe that this deletion algorithm for treaps translates into a new deletion algorithm for ordinary binary search trees: to delete node u, just treapify at u by pretending that the tree is an under-treap at u until u has at most one child. Since there are no priorities in ordinary binary search trees, we can rotate at any child v of u while treapifying.

Split and join of treaps. Let T be a treap on a set S of items. We can implement these two operations easily using insert and deletes.

(i) *To split T at k:* We insert a new item $u = (k, \infty)$ (*i.e.*, item u has key k and infinite priority). This yields a new treap with root u, and we can return the left subtree and right subtree to be the desired T_L, T_R.

(ii) *To join T_L, T_R:* first perform a Min(T_R) to obtain the minimum key k^* in T_R. Form the almost-treap whose root is the artificial node $(k^*, -\infty)$ and whose left and right subtrees are T_L and T_R. Finally, treapify at $(k^*, -\infty)$ and delete it.

The beauty of treap algorithms is that they are reduced to two simple subroutines: binary insertion and treapification.

Exercises

Exercise 4.1: For any binary search tree T and node $u \in T$, we can assign priorities to items such that T is a treap and a deletion at u takes a number of rotation equal to the the spine height of u. □

Exercise 4.2: Modify the definition of the split operation so that all the keys in S_R is strictly greater than the key k. Show how to implement this version. □

Exercise 4.3: Define "almost-treap at U" where U is a set of nodes. Try to generalize all the preceding results. □

Exercise 4.4: Alternative deletion algorithm: above, we delete a node u by making it a leaf. Describe an alternative algorithm where we step the rotations as soon as u has only one child. We then delete u by replacing u by its in-order successor or predecessor in the tree. Discuss the pros and cons of this approach. □

§5. Searching in a Random Treap

We analyze the expected cost of searching for a key in a random treap. Let us set up the random model. Assume the set of keys in the treap is $[1..n]$ and S_n the set of permutations on $[1..n]$. The event space is $(S_n, 2^{S_n})$ with a uniform probability function $\Pr_n$. Each permutation $\sigma \in S_n$ represents a choice of priority for the keys $[1..n]$ where $\sigma(k)$ is the priority for key k. Recall our "list" convention (2) (§1) for writing σ.

The *random treap* on the keys $[1..n]$ is T_n defined as follows: for $\sigma \in S_n$, $T_n(\sigma)$ is the treap on keys $[1..n]$ where the priorities of the keys are specified by σ as described above. There are three ways in which such a sample space arise.
• We randomly generate σ directly in a uniform manner. In section §6 we describe how to do this.
• We have a fixed but otherwise arbitrary continuous distribution function $F : \mathbb{R} \to [0, 1]$ and the priority of key i is a r.v. $X_i \in [0, 1]$ with distribution F. The set of r.v.'s $X_1, \ldots, X_n$ are i.i.d.. In the continuous case, the probability that $X_i = X_j$ is negligible. In practice, we can approximate the range of F by a suitably large finite set of values to achieve the same effect.
• The r.v.'s $X_1, \ldots, X_n$ could be given a uniform probability distribution if successive bits of each X_i is obtained by tossing a fair coin. Moreover, we will generate as many bits of X_i as are needed by our algorithm when comparing priorities. It can be shown that the expected number of bits needed is a small constant.

We frequently relate a r.v. of $\Pr_n$ to another r.v. of $\Pr_k$ for $k \neq n$. The following simple lemma illustrates a typical setting.

Lemma 3 *Let X, Y be r.v.'s of $\Pr_n, \Pr_k$ (respectively) where $1 \leq k < n$. Suppose there is a map $\mu : S_n \to S_k$ such that*
(i) for each $\sigma' \in S_k$, the set $\mu^{-1}(\sigma')$ of inverse images has size $n!/k!$, and
(ii) for each $\sigma \in S_n$, $X(\sigma) = Y(\mu(\sigma))$.
Then $\mathtt{E}[X] = \mathtt{E}[Y]$.

Proof.

$$\mathtt{E}[X] = \frac{\sum_{\sigma \in S_n} X(\sigma)}{n!} = \frac{\sum_{\sigma' \in S_k} (n!/k!) Y(\sigma')}{n!} = \frac{\sum_{\sigma' \in S_k} Y(\sigma')}{k!} = \mathtt{E}[Y].$$

Q.E.D.

The map μ that "erases" from the string σ all keys greater than k has the properties stated in the lemma. E.g., if $n = 5$, $k = 3$ then $\sigma(3, 1, 4, 2, 5) = (3, 1, 2)$ and $\sigma^{-1}(3, 1, 2) = 4 \cdot 5 = 20$.

Recalling our example in figure 3: notice that if we insert the keys [1..7] into an initially empty binary search tree in order of decreasing priority, (i.e., first insert key 5, then key 2, then key 1, etc), the result is the desired treap. This illustrates the following lemma:

Lemma 4 *Suppose we repeatedly insert into a binary search tree the following sequence of keys $\sigma^{-1}(n), \sigma^{-1}(n-1), \ldots, \sigma^{-1}(1)$, in this order. If the initial binary tree is empty, the final binary tree is in fact the treap $T_n(\sigma)$.*

Proof. The proposed sequence of insertions results in a binary search tree, by definition. We only have to check that the heap property. This is seen by induction: the first insertion clearly results in a heap. If the ith insertion resulted in a heap, then the $(i+1)$st insertion is an almost heap. Since this newly inserted node

is a leaf, and its rank is less than all the other nodes already in the tree, it cannot cause any violation of the heap property. So the almost heap is really a heap. Since treaps are unique for unique keys and ranks, this tree must be equal to T_σ. **Q.E.D.**

The above lemma suggests that a random treap can be regarded as a binary search tree that arises from random insertions.

Random ancestor sets A_k. We are going to fix $k \in [1..n]$ as the key we are searching for in the random treap T_n. To analyze the expected cost of this search, we define the random set $A_k : S_n \to 2^{[1..n]}$ where

$$A_k(\sigma) := \{j : j \text{ is an ancestor of } k \text{ in } T_n(\sigma)\}.$$

By definition, $k \in A_k(\sigma)$. Clearly, the cost of LookUp(T_n, k) is equal to $\Theta(|A_k|)$. Hence our goal is to determine $\mathtt{E}[|A_k|]$. To do this, we split the random variable $|A_k|$ into two parts:

$$|A_k| = |A_k \cap [1..k]| + |A_k \cap [k..n]| - 1.$$

By the linearity of expectation, we can determine the expectations of the two parts separately.

Running k-maximas of σ. We give a descriptive name to elements of the set

$$A_k(\sigma) \cap [1..k].$$

A key j is called a *running k-maxima of σ* if $j \in [1..k]$ and for all $i \in [1..k]$,

$$\sigma(i) > \sigma(j) \Rightarrow i < j.$$

Of course, $\sigma(i) > \sigma(j)$ just means that i appears before j in the list

$$(\sigma^{-1}(n), \ldots, \sigma^{-1}(1)).$$

In other words, as we run down this list, by the time we come to j, it must be bigger than any other elements from the set $[1..k]$ that we have seen so far. Hence we call j a "running maxima".

EXAMPLE. Taking $\sigma = (4, 3, 1, 6, 2, 7, 5)$ as in figure 3, the running 7-maximas of σ are 4, 6 and 7. Note that no running k-maximas appears after k in the list σ.

Lemma 5 *$j \in A_k(\sigma) \cap [1..k]$ iff j is a running k-maxima of σ.*

Proof. ($\Rightarrow$) Suppose $j \in A_k(\sigma) \cap [1..k]$. We want to show that j is a running k-maxima. That is, for all $i \in [1..k]$ and $\sigma(i) > \sigma(j)$, we must show that $i < j$. Look at the path π_j from the root to j that is traced while inserting j. It will initially retrace the corresponding path π_i for i (which was inserted earlier). Let i' be the last node that is common to π_i and π_j (so i' is the least common ancestor of i and j). We allow the possibility $i = i'$ (but surely $j \neq i'$ since $j = i'$ would make j an ancestor of i, violating the max-heap property). There are two cases: j is either (a) in the right subtree of i' or (b) in the left subtree of i'. We claim that it cannot be (b). To see this, consider the path π_k traced by inserting k. Since j is an ancestor of k, π_j must be a prefix of π_k. If j is a left descendent of i', then so is k. Since i is either i' or lies in the left subtree of i', this means $i > k$, contradiction. Hence (a) holds and we have $j > i' \geq i$.

($\Leftarrow$) Conversely, suppose j is a running k-maxima in σ. It suffices to show that $j \in A_k(\sigma)$. This is equivalent to showing

$$A_j(\sigma) \subseteq A_k(\sigma).$$

View $T_n(\sigma)$ as the result of inserting a sequence of keys by decreasing priority. At the moment of inserting k, key j has already been inserted. We just have to make sure that the search algorithm for k follows the path π_j from the root to j. This is seen inductively: clearly k visits the root. Inductively, suppose k is visiting a key i on the path π_j. If $i > k$ then surely both j and k next visit the left child of i. If $i < k$ then $j > i$ (since j is a running maxima) and hence again both j and k next visit the right child of i. This proves that k eventually visits j. **Q.E.D.**

Running k-minimas of σ. Define a key j to be a *running k-minima of σ* if $j \in [k..n]$ and for all $i \in [k..n]$,

$$\sigma(i) > \sigma(j) \Rightarrow j < i.$$

With $\sigma = (4, 3, 1, 6, 2, 7, 5)$ as before, the running 2-minimas of σ are 4, 3 and 2. As for running k-maximas, no running k-minimas appear after k in the list σ. We similarly have:

Lemma 6 *$A_k(\sigma) \cap [k..n]$ is the set of running k-minimas of σ.*

Define the random variable U_n where $U_n(\sigma)$ is the number of running n-maximas in $\sigma \in S_n$.

We will show that U_k has the same expected value as $|A_k \cap [1..k]|$. Note that U_k is a r.v. of $\Pr_k$ while $|A_k \cap [1..k]|$ is a r.v. of $\Pr_n$. Similarly, define the r.v. V_n where $V_n(\sigma)$ is the number of running 1-minimas in $\sigma \in S_n$.

Lemma 7
(a) $\mathtt{E}[|A_k \cap [1..k]|] = \mathtt{E}[U_k]$.
(b) $\mathtt{E}[|A_k \cap [k..n]|] = \mathtt{E}[V_{n-k+1}]$.

Proof. Note that the r.v. $|A_k \cap [1..k]|$ is related to the r.v. U_k exactly as described in lemma 3. This is because in each $\sigma \in S_n$, the keys in $[k+1..n]$ are clearly irrelevant to the number we are counting. Hence (a) is a direct application of lemma 3. A similar remark applies to (b) because the keys in $[1..k-1]$ are irrelevant in the running k-minimas. **Q.E.D.**

We give a recurrence for the expected number of running k-maximas.

Lemma 8
(a) $\mathtt{E}[U_1] = 1$ *and* $\mathtt{E}[U_k] = \mathtt{E}[U_{k-1}] + \frac{1}{k}$ *for* $k \geq 2$. *Hence* $\mathtt{E}[U_k] = H_k$
(b) $\mathtt{E}[V_1] = 1$ *and* $\mathtt{E}[V_k] = \mathtt{E}[V_{k-1}] + \frac{1}{k}$ *for* $k \geq 2$. *Hence* $\mathtt{E}[V_k] = H_k$

Proof. We consider (a) only, since (b) is similar. Consider the map taking $\sigma \in S_k$ to $\sigma' \in S_{k-1}$ where we delete 1 from the sequence $(\sigma^{-1}(k), \ldots, \sigma^{-1}(1))$ and replace each remaining key i by $i-1$, and take this sequence as the permutation σ'. For instance, $\sigma = (4, 3, 1, 5, 2)$ becomes $\sigma' = (3, 2, 4, 1)$. Note that $U_5(\sigma) = 2 = U_4(\sigma')$. On the other hand, if $\sigma = (1, 4, 3, 5, 2)$ then $\sigma' = (3, 2, 4, 1)$. and $U_5(\sigma) = 3$ and $U_4(\sigma') = 2$. In general, we have

$$U_k(\sigma) = U_{k-1}(\sigma') + \delta(\sigma)$$

where $\delta(\sigma) = 1$ or 0 depending on whether or not $\sigma(1) = k$. For each $\sigma' \in S_{k-1}$, that there are exactly k permutations $\sigma \in S_k$ that maps to σ'; moreover, of these k permutations, exactly one has $\delta(\sigma) = 1$. Thus

$$\begin{aligned}
\mathtt{E}[U_k] &= \frac{\sum_{\sigma \in S_k} U_k(\sigma)}{k!} \\
&= \frac{\sum_{\sigma \in S_k} (U_{k-1}(\sigma') + \delta(\sigma))}{k!} \\
&= \frac{\sum_{\sigma' \in S_{k-1}} (1 + kU_{k-1}(\sigma'))}{k!} \\
&= \frac{1}{k} + \frac{\sum_{\sigma' \in S_{k-1}} U_{k-1}(\sigma')}{(k-1)!} \\
&= \frac{1}{k} + \mathtt{E}[U_{k-1}].
\end{aligned}$$

Clearly, the solution to $\mathtt{E}[U_k]$ is the kth harmonic number $H_k = \sum_{i=1}^{k} 1/i$. **Q.E.D.**

The depth of k in the treap $T_n(\sigma)$ is

$$|A_k(\sigma)| - 1 = |A_k(\sigma) \cap [1..k]| + |A_k(\sigma) \cap [k..n]| - 2.$$

Since the cost of LookUp(k) is proportional to 1 plus the depth of k, we obtain:

Corollary 9 *The expected cost of LookUp(k) in a random treap of n elements is proportional to*

$$\mathtt{E}[U_k] + \mathtt{E}[V_{n-k+1}] - 1 = H_k + H_{n-k+1} - 1 \leq 1 + 2\ln n.$$

Exercises

Exercise 5.1: (i) Prove lemma 6. (ii) Minimize $H_k + H_{n-k+1}$ for k ranging over $[1..n]$. □

Exercise 5.2:
(i) Show that the variance of U_n is $\mathcal{O}(\log n)$. In fact, $\mathtt{Var}(U_n) < H_n$.
(ii) Use Chebyshev's inequality to show that the probability that U_n is more than twice its expected value is $\mathcal{O}(\frac{1}{\log n})$. □

§6. Insertion and Deletion

A treap insertion has two phases:
Insertion Phase. This puts the item in a leaf position, resulting in an almost-treap.
Rotation Phase. This performs a sequence of rotations to bring the item to its final position.
The insertion phase has the same cost as searching for the successor or predecessor of the item to be inserted. By the previous section, this work is expected to be $\Theta(\log n)$. What about the rotation phase?

Lemma 10 *For any set S of items and $u \in S$, there is an almost-treap T at u in which u is a leaf and the set of nodes is S. Moreover, if the keys and priorities in S are unique, then T is unique.*

Proof. To see the existence of T, we first construct the treap T' on $S - \{u\}$. Then we insert u into T' to get an almost-treap at u. For uniqueness, we note that when the keys and priorities are unique and u is a leaf of an almost-treap T, then the deletion of u from T gives a unique treap T'. On the other hand, T is uniquely determined from u and T'. **Q.E.D.**

This lemma shows that insertion and deletion are inverses in a very strong sense. Recall that for deletion, we assume that we are given a node u in the treap to be deleted. Then there are two parts again:

Rotation Phase. This performs a sequence of rotations to bring the node to a leaf position.

Deletion Step. This simply deletes the leaf.

We now clarify the precise sense in which rotation and deletion are inverses of each other.

Corollary 11 *Let T be a treap containing a node u and T' be the treap after deleting u from T. Let T_+ be the almost-treap just before the rotation phase while inserting u into T'. Let T_- be the almost-treap just after the rotation phase while deleting u from T.*

(i) T_+ and T_- are identical.

(ii) The intermediate almost-treaps obtained in the deletion rotations are the same as the intermediate almost-treaps obtained in the insertion rotations (but in the opposite order).

In particular, the costs of the rotation phase in deletion and in insertion are equal. Hence it suffices to analyze the cost of rotations in when deleting a key k. This cost is $\mathcal{O}(L_k + R_k)$ where L_k and R_k are the lengths of the left and right spines of k. These lengths are at most the depth of the tree, and we already proved that the expected depth is $\mathcal{O}(\log n)$. Hence we conclude: *the expected time to insert into or delete from a random treap is $\mathcal{O}(\log n)$.*

In this section, we give a sharper bound: the expected number of rotations during an insertion or deletion is at most 2.

Random left-spine sets B_k. Let us fix the key $k \in [1..n]$ to be deleted. We want to determine the expected spine height of k. We compute the left and right spine heights separately. For any $\sigma \in S_n$ and any key $k \in [1..n]$, define the set

$$B_k(\sigma) := \{j : j \text{ is in the left spine of } k\}.$$

Triggered running k-maxima. Again, we can give a descriptive name for elements in the random set B_k. Define j to be a *triggered running k-maxima of σ* if $j \in [1..k-1]$ and $\sigma(k) > \sigma(j)$ and for all $i \in [1..k-1]$,

$$\sigma(i) > \sigma(j) \Rightarrow i < j.$$

In other words, as we run down the list

$$(\sigma^{-1}(n), \ldots, k, \ldots, j, \ldots, \sigma^{-1}(1)),$$

by the time we come to j, we must have seen k (this is the trigger) and j must be bigger than any other elements in $[1..k-1]$ that we have seen so far. Note that the other elements in $[1..k-1]$ may occur before k in the list (*i.e.*, they need not be triggered).

With $\sigma = (4, 3, 1, 6, 2, 7, 5)$ as in figure 3, the triggered running 3-maximas of σ are 1 and 2. The triggered running 6-maximas of σ is comprised of just 7; note that 2 is not a 6-maxima because 2 is less than some keys in [1..5] which occurred earlier than 2.

Lemma 12 *(Characterization of left-spine) Then $j \in B_k(\sigma)$ iff j is a triggered running k-maxima of σ.*

Proof. ($\Rightarrow$) Suppose $j \in B_k(\sigma)$. We want to show that
(i) $j < k$,
(ii) $\sigma(k) > \sigma(j)$, and
(iii) for all $i \in [1..k-1]$, $\sigma(i) > \sigma(j)$ implies $i < j$.
But (i) holds because of the binary search tree property, and (ii) holds by the heap property. What about (iii)? Let $i \in [1..k-1]$ and $\sigma(i) > \sigma(j)$. We want to prove that $i < j$. *First assume i lies in the path from root to j.* There are two cases.

CASE (1): k is a descendent of i. Then we see that $k > i$ and j is a descendent of k implies $j > i$.

CASE (2): i is a descendent of k. But in a left-spine, the items have increasing keys as they lie further from the root. In particular, $i < j$, as desired.

Now consider the case where i does not lie in the path from the root to j. Let i' be the least common ancestor of i and j. Clearly either

$$i < i' < j \qquad \text{or} \qquad j < i' < i$$

must hold. So it suffices to prove that $i' < j$ (and so the first case hold). Note that $i' \neq k$. If i' is a descendent of k then essentially CASE (2) above applies, and we are done. So assume k is a descendent of i'. Since j is also a descendent of k, it follows that i and k lie in different (left or right) subtrees of i'. By definition $i < k$. Hence i lies in the left subtree of i' and k lies in the right subtree of i'. Hence j lies in the right subtree of i', i.e., $j > i'$, as desired. This proves (iii).

($\Leftarrow$) Suppose j is a triggered running k-maxima. We want to show that $j \in B_k(\sigma)$. We view $T_n(\sigma)$ as the result of inserting items according to priority (§3): assume that we are about to insert key j. Note that k is already in the tree. We must show that j ends up in the "tip" of the left-spine of k. For each key i along the path from the root to the tip of the left-spine of k, we consider two cases.

CASE (3): $i > k$. Then i is an ancestor of k and (by (i)) we have $i > k > j$. This means j will move into the subtree of i which contains k.

CASE (4): $i < k$. Then by (iii), $i < j$. Again, this means that if i is an ancestor of k, j will move into the subtree of i that contains k, and otherwise i is in the left-spine of k and j will move into the right subtree of i (and thus remain in the left-spine of k). **Q.E.D.**

It is evident that the keys in $[k+1..n]$ is irrelevant to the number of triggered running k-maximas. Hence we may as well assume that $\sigma \in S_k$ (instead of S_n). To capture this, let us define the r.v. T_k that counts the number of triggered running k-maximas in case $n = k$.

Lemma 13 *Consider the r.v. $|B_k|$ of $\Pr_n$. It is related to the r.v. T_k of $\Pr_k$ via $\mathtt{E}[|B_k|] = \mathtt{E}[T_k]$.*

Proof. For each $\sigma \in S_n$, let us define its *image* σ' *in* S_k obtained by taking the sequence $(\sigma^{-1}(n), \ldots, \sigma^{-1}(1))$, deleting all keys of $[k+1..n]$ from this sequence, and considering the result as a permutation $\sigma' \in S_k$. Note that $T_k(\sigma') = |B_k(\sigma)|$. It is not hard to see that there are exactly $n!/k!$ choices of $\sigma \in S_k$ whose images equal a given $\sigma' \in S_k$. Hence lemma 3 implies $\mathtt{E}[|B_k|] = \mathtt{E}[T_k]$. **Q.E.D.**

Lemma 14 *We have $\mathtt{E}[T_1] = 0$. For $k \geq 2$,*

$$\mathtt{E}[T_k] = \mathtt{E}[T_{k-1}] + \frac{1}{k(k-1)}$$

and hence $\mathtt{E}[T_k] = \frac{k-1}{k}$.

Proof. It is immediate that $\mathtt{E}[T_1] = 0$. So assume $k \geq 2$. Again consider the map from $\sigma \in S_k$ to a corresponding $\sigma' \in S_{k-1}$ where σ' is the sequence obtained from $(\sigma^{-1}(k), \ldots, \sigma^{-1}(1))$ by deleting the key 1

and subtracting 1 from each of the remaining keys. Note that

• each $\sigma' \in S_{k-1}$ is the image of exactly k distinct $\sigma \in S_k$, and

• for each $j \in [2..k-1]$, j is a triggered running k-maxima in σ iff $j-1$ is a triggered running $(k-1)$-maxima in σ'.

• key 1 is a triggered running k-maxima in σ iff $\sigma(k) = k$ and $\sigma(1) = k-1$.

This proves

$$T_k(\sigma) = T_{k-1}(\sigma') + \delta(\sigma)$$

where $\delta(\sigma) = 1$ or 0 depending on whether or not $\sigma = (k, 1, \sigma^{-1}(k-2), \ldots, \sigma^{-1}(1))$. Now, there are exactly $(k-2)!$ choices of $\sigma \in S_k$ such that $\delta(\sigma) = 1$. Hence

$$\begin{aligned}
\mathtt{E}[T_k] &= \frac{\sum_{\sigma \in S_k} T_k(\sigma)}{k!} \\
&= \frac{\sum_{\sigma \in S_k} (T_{k-1}(\sigma') + \delta(\sigma))}{k!} \\
&= \frac{\sum_{\sigma \in S_k} T_{k-1}(\sigma')}{k!} + \frac{(k-2)!}{k!} \\
&= \frac{\sum_{\sigma' \in S_{k-1}} k T_{k-1}(\sigma')}{k!} + \frac{1}{k(k-2)} \\
&= \mathtt{E}[T_{k-1}] + \frac{1}{k(k-2)}.
\end{aligned}$$

It is immediate that

$$\mathtt{E}[T_k] = \sum_{i=2}^{k} \frac{1}{i(i-1)},$$

using the fact that $\mathtt{E}[T_1] = 0$. This is a telescoping sum in disguise because

$$\frac{1}{i(i-1)} = \frac{1}{i-1} - \frac{1}{i}.$$

The solution follows at once. **Q.E.D.**

Right spine. Using symmetry, the expected length for the right spine of k is easily obtained. We define j to be a *triggered running k-minimas in σ* if (i) $j > k$, (ii) $\sigma(j) < \sigma(k)$, (iii) for all $i \in [k+1..n]$, $\sigma(i) > \sigma(j)$ implies $j < i$. We leave as an exercise to show:

Lemma 15 *j is in the right spine of k iff j is a triggered running k-minima of σ.*

The keys in $[1..k-1]$ are irrelevant for this counting. Hence we may define the random variable

$$R_k(\sigma)$$

which counts the number of triggered running 1-minimas in $\sigma \in S_k$. We have:

Lemma 16
(a) The expected length of the right spine is given by $\mathtt{E}[R_{n-k+1}]$.
(b) For $k \geq 2$, $\mathtt{E}[R_k] = \mathtt{E}[R_{k-1}] + \frac{1}{k(k-1)}$. The solution is $\mathtt{E}[R_k] = (k-1)/k$.

Corollary 17 *The expected length of the right spine of k is*

$$\mathtt{E}[R_{n-k+1}] = \frac{n-k}{n-k+1}.$$

Hence the expected lengths of the left and right spines of a key k is less than 1 each. This leads to the surprising result:

Corollary 18 *The expected number of rotations in deleting or inserting a key from the random treap T_n is less than* 2.

This is true because each key is likely to be a leaf or near one. So this result is perhaps not surprising since more than half of the keys are leaves.

Exercises

Exercise 6.1: Show lemma 15. □

§7. Cost of a Sequence of Requests

We have shown that for single requests, it takes expected $\mathcal{O}(\log n)$ time to search or insert, and $\mathcal{O}(1)$ time to delete. Does this bound apply to a sequence of such operations? The answer is not obvious because it is unclear whether the operations may upset our randomness model. How do we ensure that inserts or deletes to a random treap yields another random treap?

We formalize the problem as follows: fix any sequence

$$p_1, \ldots, p_m$$

of treap requests on the set $[1..n]$ of keys. In other words, each p_i ($k \in [1..n]$ has one of the forms

$$\text{LookUp}(k),\ \text{Insert}(k),\ \text{Delete}(k).$$

We emphasize that that the above sequence of requests is arbitrary, not "random" in any probabilistic sense. Also note that we assume that deletion operation is based on a key value, not on a "node" as we normally postulate (§4). This variation should not be a problem in most applications.

Our probability space is again

$$(S_n, 2^{S_n}, \Pr_n).$$

For each $i = 1, \ldots, m$ and $\sigma \in S_n$, let $T_i(\sigma)$ be the treap obtained after the sequence of operations $p_1, \ldots, p_i$, where σ specifies the priorities assigned to keys and $T_0(\sigma)$ is the initial empty treap. Define X_i to be the r.v. such that $X_i(\sigma)$ is the cost of performing the request p_i on the treap $T_{i-1}(\sigma)$, resulting in $T_i(\sigma)$. The expected cost of operation p_i is simply $\mathtt{E}[X_i]$.

Lemma 19

$$\mathtt{E}[X_i] = \mathcal{O}(\log i).$$

Proof. Fix $i = 1, \ldots, m$. Let $K_i \subseteq [1..n]$ denote the set of keys that are in $T_i(\sigma)$. A key observation is the dependence of $T_{i-1}(\sigma)$ on $p_1, \ldots, p_{i-1}$ amounts only to the fact that these $i-1$ operations determine K_i. Let $|K_i| = n_i$. Then there is a natural map taking $\sigma \in S_n$ to $\sigma' \in S_{n_i}$ (*i.e.*, the map deletes from the sequence $(\sigma^{-1}(n), \ldots, \sigma^{-1}(1))$ all elements not in K_i, and finally replacing each remaining key $k \in K_i$ by a corresponding key $\pi_i(k) \in [1..n_i]$ which preserves key-order). Each σ' is the image of $n!/(n_i)!$ distinct $\sigma \in S_n$. Moreover, for any key $k \in K_i$, its relevant statistics (the depth of k and length of left/right spines of k) in $T_i(\sigma)$ is the same as that of its replacement key $\pi_i(k)$ in the treap on $[1..k]$. As shown above, the expected values of these statistics is the same as the expected values of these statistics on the uniform probability space on S_{n_i}. But we have proven that the expected depth of any $k \in [1..n_i]$ is $\mathcal{O}(\log n_i)$, and the expected length of the spines of k is at most 1. This proves that when the update operation is based on a key $k \in K_i$, $\mathtt{E}[X_i] = \mathcal{O}(\log n_i) = \mathcal{O}(\log i)$, since $n_i \leq i$.

What if the operation p_i is an unsuccessful search, i.e., we do a LookUp(k) where $k \notin K_i$? In this case, the length of the path that the algorithm traverses is bounded by the depth of either the successor or predecessor of k in K_i. Again, the expected lengths in $\mathcal{O}(\log i)$. This concludes our proof. **Q.E.D.**

Application to Sorting. We give a randomized sorting algorithm as follows: on input a set of n keys, we assign them a random priority (by putting them in an array in decreasing order of priority). Then we do a sequence of n inserts in order of this priority. Finally we do a sequence of n DeleteMins. This gives an expected $\mathcal{O}(n \log n)$ algorithm for sorting.

NOTES: Galli and Simon (1996) suggested an alternative approach to treaps which avoid the use of random priorities. Instead, they choose the root of the binary search tree randomly. Mulmuley introduced several abstract probabilistic "game models" that corresponds to our analysis of running (triggered) maximas or minimas.

Exercises

Exercise 7.1: What is the worst case and best case key for searching and insertion in our model? □

References

[1] C. R. Aragon and R. G. Seidel. Randomized search trees. *IEEE Foundations of Computer Science*, 30:540–545, 1989.

[2] A. T. Jonassen and D. E. Knuth. A trivial algorithm whose analysis isn't. *J. of Computer and System Sciences*, 16:301–322, 1978.

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