Lecture 13 Random Search Trees

In this lecture we will describe a very simple probabilistic data structure that allows inserts, deletes, and membership tests (among other operations) in expected logarithmic time.

These results were first obtained by Pugh in 1988 (see [88]), who called his probabilistic data structure *skip lists*. We will follow the presentation of Aragon and Seidel [7], whose data structure is somewhat different and more closely related to the self-adjusting trees presented in the last lecture, and whose probabilistic analysis is particularly elegant.

13.1 Treaps

Consider a binary tree, not necessarily balanced, with nodes drawn from a totally ordered set, ordered in inorder; that is, if i is in the left subtree of k and j is in the right subtree of k, then $i < k < j$. Recall that the **rotate** operation discussed in the previous lecture preserves this order.

Now suppose that each element k has a unique *priority* $p(k)$ drawn from some other totally ordered set, and that the elements are ordered in heap order according to priority; that is, an element of maximum priority in any subtree is found at the root of that subtree.

A tree in which the data values k are ordered in inorder and the priorities $p(k)$ are ordered in heap order is called a *treap* (for *tree-heap*, one supposes).

It may not be obvious at first that treaps always exist for every priority assignment. They do! Moreover, if the priorities are distinct, then the treap is unique.

Lemma 13.1 *Let X and Y be totally ordered sets, and let p be a function assigning a distinct priority in Y to each element of X. Then there exists a unique treap with nodes X and priorities p.*

Proof. Let k be the unique element of X of maximum priority; this must be the root. Partition the remaining elements into two sets

$$\{i \in X \mid i < k\}, \qquad \{i \in X \mid i > k\}$$

Inductively build the unique treaps out of these two sets and make them the left and right subtrees of k, respectively. □

13.2 Random Treaps

A *random treap* is a treap in which the priorities have been assigned randomly. This is best done in practice by calling a random number generator each time a new element m is presented for insertion into the treap to assign a random priority to m. Under some highly idealized but reasonable assumptions about the random number generator[3], two elements receive the same priority with probability zero, and if all elements in the treap are sorted by priority, then every permutation is equally likely.

When a new element m is presented for insertion or to test membership, we start at the root and work our way down some path in the treap, comparing m to elements along the path to see which way to go to find m's appropriate inorder position. If we see m on the path on the way down, we can answer the membership query affirmatively. If we make it all the way down without seeing m, we can answer the membership query negatively.

If m is to be inserted, we attach m as a new leaf in its appropriate inorder position. At that point we call the random number generator to assign a random priority $p(m)$, which by Lemma 13.1 specifies a unique position in the treap. We then rotate m upward as long as its priority is greater than that of its parent, or until m becomes the root. At that point the tree is in heap order with respect to the priorities and in inorder with respect to the data values.

To delete m, we first find m by searching down from the root as described above, then rotate m down until it is a leaf, taking care to choose the direction

[3] A call to the random number generator gives a uniformly distributed random real number in the interval $[0, 1)$, and successive calls are statistically independent; *i.e.* if $x_1, \ldots, x_n$ are the results of n successive calls, then

$$\Pr(\bigwedge_{1 \le i \le n} x_i \in A_i) = \prod_{1 \le i \le n} \Pr(x_i \in A_i) .$$

of rotation so as to maintain heap order. For example, if the children of m are j and k and $p(j) > p(k)$, then we rotate m down in the direction of j, since the rotate operation will make j an ancestor of k. When m becomes a leaf, we prune it off.

The beauty of this approach is that the position of any element in the treap is determined once and for all at the time it is inserted, and it stays put at that level until it is deleted; there is not a lot of restructuring going on as with splay trees. Moreover, as we will show below, the expected number of rotations for an insertion or deletion is at most two.

13.3 Analysis

We now show that, averaged over all random priority assignments, the expected time for any insert, membership test, or delete is $O(\log n)$.

We will do the analysis for deletes only; it is not hard to see that the time bound for membership tests and inserts is proportionally no worse than for deletes. Suppose that at the moment, the treap contains n data items (without loss of generality, say $\{1,2,\ldots,n\}$), and we wish to delete m. The priorities have been chosen randomly, so that if the set $\{1,2,\ldots,n\}$ is sorted in decreasing order by priority to obtain a permutation σ of $\{1,2,\ldots,n\}$, every σ is equally likely.

In order to locate m in the treap, we follow the path from the root down to m. The amount of time to do this is proportional to the length of the path. Let us calculate the expected length of this path, averaged over all possible random permutations σ.

Let

$$\begin{aligned} m_\leq &= \{1,2,\ldots,m\} \\ m_\geq &= \{m,m+1,\ldots,n\} \end{aligned}$$

Let A be the set of ancestors of m, including m itself. The definitions of $m_\leq$ and $m_\geq$ do not depend on σ, but the definition of A does. Let X be the random variable

$$\begin{aligned} X &= \text{length of the path from the root down to } m \\ &= |m_\leq \cap A| + |m_\geq \cap A| - 2\ . \end{aligned}$$

The 2 is subtracted because m is counted in both $m_\leq$ and $m_\geq$.

We are interested in $\mathcal{E}X$, the expected value of X; by linearity of expectation, we have

$$\mathcal{E}X = \mathcal{E}|m_\leq \cap A| + \mathcal{E}|m_\geq \cap A| - 2\ .$$

By symmetry, it will suffice to calculate $\mathcal{E}|m_\leq \cap A|$.

Note that if the elements of $m_\leq$ are sorted in descending order by priority, then

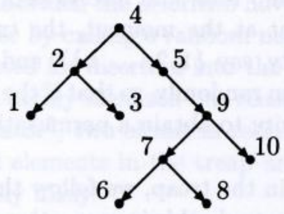
- every permutation of $m_\leq$ is equally likely;
- an element of $m_\leq$ is in A if and only if it is larger than all previous elements of $m_\leq$ in sorted order.

In other words, permute $m_\leq$ randomly, then scan the resulting list from left to right, checking off those elements k that are larger than anything to the left of k; the quantity $\mathcal{E}|m_\leq \cap A|$ is the expected number of checks.

Example 13.2 Let $n = 10$ and $m = 8$. Suppose that when priorities are assigned randomly to $\{1, 2, \ldots, 10\}$ and these elements are sorted in decreasing order by priority, we get the permutation

$$\sigma \;=\; (4, 5, 9, 2, 1, 7, 3, 10, 8, 6)$$

This results in the following treap:



Then $m_\leq = \{1, 2, 3, 4, 5, 6, 7, 8\}$. If we restrict the random permutation σ to this set, we obtain the permutation $(4, 5, 2, 1, 7, 3, 8, 6)$. Scanning from left to right and checking only those elements k that are greater than all elements to the left of k, we get the sequence $(4, 5, 7, 8)$. This is exactly the sequence of elements in $m_\leq$ appearing on the path from the root down to m in the treap.

A symmetric argument using $m_\geq$ gives the sequence $(9, 8)$, which is the sequence of elements in $m_\geq$ appearing on the path from the root down to m. The length of the path is then the sum of the two lengths of these sequences less 2. □

We are thus left with the problem of determining the expected value of the random variable H_m, the number of checks obtained when scanning a random permutation of $\{1, 2, \ldots, m\}$ from left to right and checking every element that is greater than anything to its left.

We claim that this number is exactly

$$\mathcal{E}H_m \;=\; \sum_{k=1}^{m} \frac{1}{k} . \tag{23}$$

We will obtain this by solving a simple recurrence, using the linearity of expectation.

Suppose we permute $\{1,\ldots,m\}$ randomly to get the random permutation σ. Deleting 1 from σ, we get a random permutation σ' of $\{2,3,\ldots,m\}$. Note that an element other than 1 is checked when scanning σ if and only if it is checked when scanning σ'; thus the presence or absence of 1 does not affect whether 2 is checked (however, the presence or absence of 2 might very well affect whether 1 is checked). Thus the expected number of checks on elements other than 1 is the same in σ as in σ', or $\mathcal{E}H_{m-1}$. The element 1 is checked if and only if it occurs first in σ, and this occurs with probability $\frac{1}{m}$. Thus the expected number of checks on the element 1, averaged over all permutations, is $\frac{1}{m}$. By linearity of expectation,

$$\mathcal{E}H_m = \mathcal{E}H_{m-1} + \frac{1}{m} .$$

The unique solution to this recurrence with $\mathcal{E}H_1 = 1$ is (23).

The quantity (23) is $O(\log m)$. This can be verified by approximating the sum above and below with definite integrals involving the functions $\frac{1}{x}$ and $\frac{1}{x+1}$, and recalling from calculus that

$$\int_1^m \frac{dx}{x} = \ln m = \ln 2 \cdot \log_2 m .$$

13.4 Expected time for deletion

A similar analysis allows us to calculate the expected number of rotations necessary to delete m from its position in the treap. The number of rotations needed is the sum of the length of the rightmost path in the left subtree of m and the length of the leftmost path in the right subtree of m. To see this, try rotating m down; if you rotate to the left (right), the length of the rightmost (leftmost) path in the left (right) subtree decreases by one and the length of the leftmost (rightmost) path in the right (left) subtree stays the same.

Let us calculate the expected value of G_m, the length of the rightmost path of the left subtree of m. By symmetry, the expected length of the leftmost path of the right subtree of m is $\mathcal{E}G_{n-m+1}$, and by the linearity of expectation, the expected number of rotations to remove m is $\mathcal{E}G_m + \mathcal{E}G_{n-m+1}$. We will show below that this number is less than 2!

An analysis similar to the analysis for $\mathcal{E}H_m$ above reveals that $\mathcal{E}G_m$ is the expected number of checks obtained when scanning a random permutation of the set $\{1,2,\ldots,m\}$ from left to right, where we check an element k provided that

- k occurs strictly to the right of m;
- k is greater than all elements of $\{1,2,\ldots,m-1\}$ occurring to the left of k and either to the left or to the right of m.

This is the same as the expected number of checks obtained when scanning a random permutation of the set $\{1, 2, \ldots, m-1\}$ from left to right, where we check element k if it is greater than all elements to its left, then place m randomly in the list and erase those checks occurring to the left of m.

Example 13.3 For $m = 3$, we have the following six situations, all occurring with equal probability:

$$\begin{array}{ccccccc} 3 & \overset{\checkmark}{1} & \overset{\checkmark}{2} & \qquad & 3 & \overset{\checkmark}{2} & 1 \\ 1 & 3 & \overset{\checkmark}{2} & & 2 & 3 & 1 \\ 1 & 2 & 3 & & 2 & 1 & 3 \end{array}$$

The expected number of checks is $\frac{1}{6} \cdot 2 + \frac{1}{3} \cdot 1 = \frac{2}{3}$. □

It is easy to see that the expected value of G_m is at most that of H_{m-1}, which we would get if the checks to the left of m were not erased; thus $\mathcal{E}G_m \leq \mathcal{E}H_{m-1} = O(\log m)$, and this suffices for our complexity bound.

In fact, it turns out that $\mathcal{E}G_m < 1$. As above, the expected number of checks on elements other than 1 is $\mathcal{E}G_{m-1}$, and the probability that 1 is checked is $\frac{1}{m(m-1)}$, since 1 is checked if and only if m occurs leftmost, followed immediately by 1. Again, by linearity of expectation, $\mathcal{E}G_m$ is the expected number of checks on elements other than 1 plus the expected number of checks on 1:

$$\mathcal{E}G_m = \mathcal{E}G_{m-1} + \frac{1}{m(m-1)}$$

and $\mathcal{E}G_1 = 0$. The solution to this recurrence is

$$\mathcal{E}G_m = \frac{m-1}{m}.$$