Lecture 3: Randomized searching: skip lists and treaps

Let $S$ be a set of elements, where each element $x$ has a key $\text{key}[x]$ associated to it. A dictionary on a set $S$ is an abstract data structure that supports the following operations:

- $\text{Search}(S, k)$: return an element $x \in S$ such that $\text{key}[x] = k$ if such an element exists, and $\text{nil}$ otherwise.
- $\text{Insert}(S, x)$: insert element $x$ with associated key $\text{key}[x]$ into the structure (thereby setting $S \leftarrow S \cup \{x\}$), where it is assumed that $x$ is not yet present.
- $\text{Delete}(S, x)$: delete element $x$ from the structure (thereby setting $S \leftarrow S \setminus \{x\}$). Here the parameter of the procedure is (a pointer to) the element $x$, so we do not have to search for it.

If the set $S$ is static—the operations $\text{Insert}$ and $\text{Delete}$ need not be supported—then a sorted array is a good structure: it is simple and a $\text{Search}$ operation takes only $O(\log n)$ time, where $n = |S|$. Insertions into and deletions from a sorted array are expensive, however, so when $S$ is dynamic then an array is not a good choice to implement a dictionary. Instead we can use a red-black tree or any other balanced search tree, so that not only $\text{Search}$ but also $\text{Insert}$ and $\text{Delete}$ can be done in $O(\log n)$ time in the worst case—see Chapters 12 and 13 from [CLRS] for more information on search trees and red-black trees. Even though red-black trees are not very difficult to implement, the details of $\text{Insert}$ and $\text{Delete}$ are somewhat tedious. Here we shall look at two simpler, randomized solutions. For simplicity of presentation we will assume that all the keys are distinct.

### 3.1 Treaps

A binary search tree on a $S$ is a binary tree whose internal nodes store the elements from $S$ such that the search-tree property holds:

Let $\nu$ be a node in the tree storing an element $x$. Then for any element $y$ stored in the left subtree of $\nu$ we have $\text{key}[y] < \text{key}[x]$, and for any element $z$ in the right subtree of $\nu$ we have $\text{key}[x] < \text{key}[z]$.

A search operation on a binary search tree takes time linear in the depth of the tree. Hence, we want a tree with small depth. In a so-called treap this is achieved as follows.

We also assign a priority $\text{prio}[x]$ to each element $x \in S$, where we assume that all priorities are distinct. A treap on $S$ is a binary tree storing the element from $S$ in its internal nodes such that it has the search-tree property with respect to its keys and the heap property with respect to its priorities:

Let $\nu$ be a node in the heap storing an element $x$. Then for any element $y$ stored in the left or right subtree of $\nu$ we have $\text{prio}[y] < \text{prio}[x]$.

Is it always possible to satisfy both the search-tree property and the heap property? The answer is yes. However, for any set $S$ with given keys and priorities there is only one way to do this.

**Lemma 3.1** There is a unique treap for any set $S$. 
The idea of a treap is to assign each element in $S$ a random priority. To avoid some technical difficulties we shall assume that we can generate a real number uniformly at random in the interval $[0,1]$ in $O(1)$ time. If we now choose $\text{prio}[x]$ uniformly at random from $[0,1]$ for each $x \in S$ independently, then all priorities are distinct with probability one.\footnote{Of course we cannot really generate random real numbers. Instead we can generate enough random bits from the binary representation of each priority to ensure all priorities are distinct. To focus on the main ideas behind treaps and their analysis, we will ignore this issue and just assume we can generate random real numbers.}

The hope is of course that by assigning random priorities, the treap is expected to be rather well balanced. The next lemma shows that this is indeed the case. Define the depth of an element $x$ to be the number of nodes on the path from the root of the treap to $x$.\footnote{In fact, we should say “to the node of the tree where $x$ is stored.” To simplify the presentation we will permit ourselves this slight abuse of terminology and we will not distinguish between the elements and the nodes where they are stored.}

We want to count the expected number of elements that are on the path to—the node storing $x$.

\textbf{Lemma 3.2}

\begin{equation}
E[\text{depth}(x_k)] = H_k + H_{n-k+1} - 1,
\end{equation}

where $H_j := \sum_{i=1}^{j} \frac{1}{i} \approx \ln j$.

\textbf{Proof.} We want to count the expected number of elements $i$ that are an ancestor of—that is, that are on the path to—the node storing $x_k$. To this end we introduce indicator random variables

\begin{equation}
X_i := \begin{cases} 
1 & \text{if } i \text{ is an ancestor of } x_k \\
0 & \text{otherwise}
\end{cases}
\end{equation}

Thus we want to bound $E[\sum_{i=1}^{n} X_i]$. By linearity of expectation, this boils down to bounding the values $E[X_i]$. By definition of $X_i$ we have $E[X_i] = \Pr[i \text{ is an ancestor of } x_k]$. What is this probability? Suppose first that $i \leq k$. We claim that $i$ is an ancestor of $x_k$ if and only if $\text{prio}(x_i) > \text{prio}(x_j)$ for all $i < j \leq k$. In other words, if and only if $i$ has the highest priority among the elements in $\{x_i, x_{i+1}, \ldots, x_k\}$. Before we prove this claim, we show that it implies the lemma. Indeed, since the priorities are chosen independently and uniformly at random from $[0,1]$, each of the elements in $\{x_i, x_{i+1}, \ldots, x_k\}$ has the same probability of getting the highest priority. This probability is simply $1/(k-i+1)$. Similarly, for an element $x_i$ with $i \geq k$, we have $\Pr[i \text{ is an ancestor of } x_k] = 1/(i-k+1)$. We get

\begin{equation}
E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{k} \Pr[i \text{ is an ancestor of } x_k] + \sum_{i=k}^{n} \Pr[i \text{ is an ancestor of } x_k] - 1 = \sum_{i=1}^{k} \frac{1}{k+i} + \sum_{i=k}^{n} \frac{1}{i-k+1} - 1 = \sum_{i=1}^{k} \frac{1}{i} + \sum_{i=1}^{n-k+1} \frac{1}{i} - 1 = H_k + H_{n-k+1} - 1
\end{equation}

It remains to prove our claim that $i$ is an ancestor of $x_k$ (for $i < k$) if and only if $\text{prio}(x_i) > \text{prio}(x_j)$ for all $i < j \leq k$.\hfill \Box
To prove the if-part of the statement, assume that \( \text{prio}(x_i) > \text{prio}(x_j) \) for all \( i < j \leq k \). By the heap property, \( x_i \) cannot be in the subtree rooted at \( x_k \). Moreover, there cannot be an ancestor \( x_j \) of \( x_k \) with \( x_i \) and \( x_k \) in different subtrees. Such an element would have higher priority than \( x_i \) and would satisfy \( x_i < x_j < x_k \), contradicting our assumption that \( \text{prio}(x_i) > \text{prio}(x_j) \) for all \( i < j \leq k \). Hence, \( x_i \) must be an ancestor of \( x_k \) if \( \text{prio}(x_i) > \text{prio}(x_j) \) for all \( i < j \leq k \).

To prove the only-if-part, take an ancestor \( x_i \) of \( x_k \) and suppose for a contradiction that there is some \( x_j \) with \( i < j \leq k \) that has higher priority than \( x_i \). By the heap property, \( x_j \) cannot be stored in the subtree rooted at \( x_i \). But \( x_j \) cannot be stored at an ancestor of \( x_i \); either: we have \( x_i < x_j \leq x_k \), so \( x_i \) would be stored in the left subtree of \( x_j \) and \( x_k \) would not be stored in that subtree, contradicting that \( x_i \) is an ancestor of \( x_k \). The only remaining possibility is that there is an ancestor \( x_{\ell} \) of \( x_i \) that has \( x_i \) in one of its subtrees and \( x_j \) in the other. But together with \( x_i < x_j \leq x_k \) this again contradicts that \( x_i \) is an ancestor of \( x_k \), since \( x_k \) would be in the same subtree of \( x_i \) as \( x_j \). We conclude that if \( x_i \) is an ancestor of \( x_k \) then \( \text{prio}(x_i) > \text{prio}(x_j) \) for all \( i < j \leq k \).

We have proved that the expected depth of any element \( x \in S \) is \( O(\log n) \), which means that the expected search time for any element is \( O(\log n) \). Can we conclude that the expected depth of the whole tree \( T \) is \( O(\log n) \)? Not immediately. The problem is that in general for a collection \( Y_1, \ldots, Y_n \) of random variables \( E[\max_i Y_i] \neq \max_i E[Y_i] \): the expected maximum of a collection of random variables is not necessarily the same as the maximum of the expectations of those random variables. In particular

\[
E[\max_{x \in S} \text{depth}(x)] \neq \max_{x \in S} E[\text{depth}(x)].
\]

Stated differently, the fact that the depth of every individual \( x \in S \) is expected to be \( O(\log n) \) does not imply that the depths of all of them are expected to be \( O(\log n) \) simultaneously. For this we need a stronger statement about the individual depth: instead of just saying that the expected depth of any \( x \in S \) is \( O(\log n) \), we must argue that the depth of every \( x \) is \( O(\log n) \) with high probability, that is, that the probability that the depth is \( O(\log n) \) is of the form \( 1 - \frac{1}{n^c} \) from some constant \( c > 0 \).

**Lemma 3.3**

\[
\Pr[ \text{depth}(x_k) \leq 7(H_k + H_{n-k+1} - 1) ] \geq 1 - (1/n^6).
\]

**Proof.** Recall from the proof of Lemma 3.2 that the depth of an element \( x_k \) can be written as \( \sum_{i=1}^{n} X_i \), where \( X_i \) is the indicator random variable for the event “\( x_i \) is an ancestor of \( x_k \)”. We now observe that the events “\( x_i \) is an ancestor of \( x_k \)” and “\( x_j \) is an ancestor of \( x_k \)” are independent for \( i \neq j \). Indeed, \( x_i \) is an ancestor of \( x_k \) (for some \( i < k \)) if and only if \( x_i \) has the highest priority among \( \{x_i, \ldots, x_k\} \), and the probability of this happening is not influenced by the fact that \( x_j \) has the highest priority in some subset \( \{x_j, \ldots, x_k\} \). Hence, we are in the situation of \( n \) independent Poisson trials \( X_1, \ldots, X_n \). This means we can use the following result: for Poisson trials, if we let \( X = \sum_{i=1}^{n} X_i \) and \( \Pr[X_i = 1] = p_i \), then we have for any \( \delta > 0 \) that

\[
\Pr[X > (1 + \delta)\mu] \leq \left( \frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu,
\]
where $\mu = \mathbb{E}[X]$ and $e = 2.718\ldots$ (the base of the natural logarithm). Applying this result with $\delta = e^2 - 1 > 6$ and noting that $H_k + H_{n-k+1} - 1 \geq H_n$ for all $k$, we get

$$\Pr[\text{depth}(x_k) > (1 + \delta)(H_k + H_{n-k+1} - 1)] \leq \left(\frac{e^\delta}{(1+3)^{1+\pi}}\right)^H_n \leq \left(\frac{e^\delta}{(e^2)^{1+\pi}}\right)^H_n \leq \left(\frac{\delta}{e}\right)^{\delta H_n} \leq \left(\frac{1}{n}\right)^6 \quad \text{(since } H_n \geq \ln n \text{ and } \delta > 6)$$

Okay, we have established that every $x_i \in S$ has depth $O(\log n)$ with high probability, namely probability at least $1 - (1/n)^6$. What is the probability that all elements have depth $O(\log n)$ simultaneously? To bound this probability we define $A_k$ as the event “$\text{depth}(x_k) \geq 7(H_k + H_{n-k+1} - 1)$”. Now we use that the probability that at least one of a given set of events takes place is upper bounded by the sum of the probabilities of the individual events. Hence

$$\Pr[A_1 \cup \cdots \cup A_n] \leq \Pr[A_1] + \ldots + \Pr[A_n] \leq n \cdot (1/n^6) = 1/n^5.$$ 

We conclude that the depth of the whole tree is $O(\log n)$ with high probability. (Note that since the depth is never more than $n$, this also implies that the expected depth is $O(\log n)$.)

We have seen that the search time in a treap is $O(\log n)$ with high probability. Now what about insertions and deletions into a treap?

First consider the insertion of a new element $x$ with key $key[x]$ into the treap $T$. We do this as follows. We assign $x$ a priority by taking a real number in the range $[0, 1]$ uniformly at random. Then we first insert $x$ into $T$ without paying attention to its priority: we just search in $T$ with the value $key[x]$ to replace the leaf where the search ends with a node storing $x$. This is exactly as one would insert into an unbalanced search tree—see Chapter 12 from [CLRS]. After doing this, the new tree $T$ will have the search-tree property. Of course it may happen that the priority of $x$ is higher than the priority of its parent, thus violating the heap property. When this happens we perform a rotation around the parent of $x$—see Chapter 13 of [CLRS] for a description of rotations—thus bringing $x$ one level higher up in the tree. If the priority of $x$ is still higher than the priority of its parent, we move $x$ up one more level by doing another rotation. This continues until the priority of $x$ is lower than that of its parent (or $x$ becomes the root), at which moment the heap property is restored. Since at each rotation $x$ moves up one level in $T$, we do at most $\text{depth}(T)$ rotations, and the insertion runs in time $O(\log n)$ with high probability.

Deletions work in the reverse way: first we move $x$ down in the tree by rotations (pretending it has priority $-\infty$) until both its children are leaves, and then we can easily delete $x$.

### 3.2 Skip lists

Let $S$ be a set of $n$ elements. Let’s slightly generalize the search operation, so that when there is no element $x$ with $key[x] = k$ it returns the element $x \in S$ with the largest key smaller than $k$. We call this element the predecessor of $k$, and we call the query a predecessor query. Thus $\text{Predecessor}(S, k)$ must return the element with the largest key smaller than or equal to $k$. 

\[\]
A skip list for a set $S$ consists of a number of sorted linked lists $L_0, L_1, \ldots, L_h$. Each list $L_i$ stores a subset $S_i \subset S$, such that $S_0 = S$, and $S_i \subset S_{i-1}$ for all $0 < i \leq h$, and $S_h = \emptyset$. Each sorted list also stores two dummy elements, one with key $-\infty$ at the beginning of the list and one with key $+\infty$ at the end of the list. For a set $S_i$ (or list $L_i$) we call $i$ the level of that set (or list), and we call $h$ the height of the skip list. We also have pointers between consecutive lists. More precisely, for every element $x \in S_i$ (with $i > 0$) we have a pointer from its occurrence in $L_i$ to its occurrence in $L_{i-1}$—see Fig. 1 for an example.

![Figure 1: A skip list on a set of seven elements. The search path taken by a predecessor query with query value $k$ (for some $k$ with $\text{key}[x_4] \leq k < \text{key}[x_5]$) is indicated.](image)

Next we describe the algorithm for performing a predecessor query in a skip list $S$. We denote the pointer from an element $x \in L_i$ to the next element in $L_i$ by $\text{next}[x]$, and the pointer from $x$ in $L_i$ to the copy of $x$ in $L_{i-1}$ by $\text{down}[x]$.

**Algorithm** $\text{Predecessor}(S, k)$

1. Let $y$ be the dummy element $-\infty$ of $L_h$.
2. for $i \leftarrow h$ downto 1
3. \hspace{1em} do $y \leftarrow \text{down}[y]$  
4. \hspace{1em} while $\text{key[next}[y]] \leq k$
5. \hspace{1em} do $y \leftarrow \text{next}[y]$
6. return $y$

Recall that $S_0$ is the whole set $S$. The main question is how the sets $S_i$, $i \geq 1$, should be chosen. Let $x_1, x_2, \ldots, x_n$ denote the elements from $S$ in sorted order. One idea would be to take $S_1 = \{x_2, x_4, x_6, \ldots\}$, to take $S_2 = \{x_4, x_8, \ldots\}$, and so on. In other words, we obtain $S_i$ by deleting the first, third, fifth, etc. element from $S_{i-1}$. This way the depth of the skip list would be $O(\log n)$ and the while-loop of lines 4–5 of $\text{Predecessor}$ would do at most one step step forward in the list $L_i$. However, this would make updating the skip list very costly. Hence, skip lists employ the following simple but powerful idea: for each element $x \in S_{i-1}$ we flip a fair coin; if the outcome is HEADS we put $x$ into $S_i$, if the outcome is TAILS then we do not put $x$ into $S_i$. In other words, each element in $S_{i-1}$ is selected independently with probability $1/2$. Since each element is selected with probability $1/2$, we expect that $|S_i| = |S_{i-1}|/2$. In other words, the number of elements is expected to halve at every level.
The following lemma proves that the height is $O(\log n)$ with high probability, a result that we will need to prove a bound on the query time.

**Lemma 3.4** The height of a skip list on a set of $n$ elements is more than $1 + t \log n$ with probability less than $1/n^{t-1}$.

**Proof.** Consider an element $x \in S$. We define the height of $x$ to be the largest $i$ such that $x \in S_i$. Thus the height of the skip list is equal to $1 + \max_{x \in S} \text{height}(x)$. Now observe that for an element $x$ to have height at least $s$, it must have had $s$ successive coin flips turn up Heads. Hence,

$$\Pr[ \text{height}(x) \geq s ] = (1/2)^s.$$ 

Setting $s = t \log n$ we get

$$\Pr[ \max_{x \in S} \text{height}(x) \geq t \log n ] = 1/n^{t-1}.$$ 

We are now ready to prove a bound on the query time in a skip list.

**Lemma 3.5** The expected time to answer a predecessor query in a skip list is $O(\log n)$.

**Proof.** The query time is equal to

$$O(\sum_{i=0}^{h} (1 + \# \text{next-pointers followed in } L_i)).$$

Let $X_i$ be a random variable denoting the number of next-pointers followed in $L_i$. We want to bound

$$\mathbb{E}[\sum_{i=0}^{h} (1 + X_i)].$$

Recall that $S_{i+1}$ is obtained from $S_i$ by sampling every element independently with probability $1/2$. How many next-pointers do we have to follow in $L_{i+1}$? Well, we start at an element $x_j$ in $S_i$ that is also present in $S_{i+1}$ and then walk to the right. We cannot reach an element $x_{\ell}$ of $S_i$ that is also present in $S_{i+1}$, otherwise we would already have reached $x_{\ell}$ in $L_{i+1}$. Hence, the number of next-pointers followed in $L_i$ is at most the number of elements in $S_i$ between $x_j$ and $x_{\ell}$, the smallest element greater than $x_j$ in $S_{i+1}$. Since the elements $x_{j+1}$, $x_{j+2}$, etc. are all present in $S_{i+1}$ with probability $1/2$, we expect that in between two successive elements in $S_{i+1}$ there is only one element in $S_i$. Hence, $\mathbb{E}[X_i] \leq 1$. Using linearity of expectation we obtain a bound on the number of next-pointers followed in the first $3 \log n$ levels:

$$\mathbb{E}[\sum_{i=0}^{3 \log n} (1 + X_i)] = O(\log n) + \sum_{i=0}^{3 \log n} \mathbb{E}[X_i] = O(\log n).$$
For $i > 3 \log n$ we of course also have $E[X_i] \leq 1$. However, the total number of levels could be much more than logarithmic, so we have to be a bit careful. Fortunately Lemma 3.4 tells us that the probability that the total number of levels is more than $1 + 3 \log n$ is quite small. Combining this with the fact that obviously $X_i \leq n$ and working out the details, we get:

$$E[\sum_{i \geq 3 \log n} (1 + X_i)] \leq \sum_{i \geq 3 \log n} \Pr[\text{ height of skip list } \geq i] \cdot (1 + n)$$
$$\leq \sum_{t \geq 3} \Pr[\text{ height of skip list } \geq t \log n] \cdot (1 + n) \log n$$
$$\leq \sum_{t \geq 3} (1/n^{t-1}) \cdot (1 + n) \log n$$
$$= O(1).$$

It remains to describe how to insert and delete elements. We only sketch these operations, leaving the details as an exercise.

Suppose we want to insert an element $x$. The first idea that comes to mind is to perform a predecessor query with key $[x]$ to find the position of $x$ in the bottom list $L_0$. Then we throw a coin to decide if $x$ also has to be inserted into $L_1$. If so, we insert $x$ into $L_1$ and throw a coin to see if we also need to insert $x$ into $L_2$, and so on. The problem with this approach is that a skip list does not have up-pointers, so we cannot go from $L_0$ to $L_1$ for instance. Hence, we proceed differently, as follows. First, we determine $\text{height}(x)$, the number of lists into which we need to insert $x$. To this end we flip a coin until we get TAILS; the number of coin flips then gives us $\text{height}(x)$. Then we search the skip list with key $[x]$, inserting $x$ into the relevant lists. (Note that it can happen that $\text{height}(x)$ is larger than the current height of the entire skip list. When this happens the skip list will get a new level above the already existing levels.)

To delete an element $x$ from a skip list we walk down the skip list, making sure that at every level we reach the largest element smaller than $key[x]$. In the lists $L_i$ containing $x$, this will give us the element just before $x$, so we can easily delete $x$ from those lists.