14.1 Skip Lists

Consider the following problem solved by the balanced binary search tree: the *dynamic predecessor problem* (*dynamic* for a data structure means that it supports updates, i.e. insertions and deletions; the opposite, where we do not support updates, is the *static* case). The predecessor problem is to maintain a set of keys $k_1, k_2, \ldots$ subject to the following operations:

- **insert(k):** insert a new item into the database with key k
- **delete(x):** delete item x from the database (we assume x is a pointer to the item)
- **pred(k):** return the item k' in the database with k' as large as possible such that $k' \leq k$ (so in particular if there is an item with key k in the database, we will find that item)

Throughout this section we let n denote the number of items in our database. We also assume that all items in the database have distinct keys for simplicity. If not, the semantics above are not well-defined (does **delete** mean delete all copies? or one copy? And should **pred** return all copies or one copy?). One could define some semantics for dealing with duplicate keys, but we will not deal with that here – it does not change the "big picture".

Deterministically it is possible to support the three operations above in $O(\log n)$ worst case time each, using for example red-black trees, AVL trees, 2-3-4 trees, or any other balanced binary search tree. Here we will see a randomized algorithm which is very simple and achieves $O(\log n)$ expected time for any operation: the *skip list*.

Before presenting the skip list, we first motivate it. One of the simplest data structures solving the dynamic predecessor problem is the doubly linked list. We will maintain the n items in the linked list in unsorted order. To insert a new item, we simply make it the new head of the list. To delete, given a pointer, we remove that node from the list and alter its previous and next nodes to point to each other. To answer a **pred** query, we start at the beginning of the list and walk to the right, keeping track of the largest k' we see which is at most k. Thus, **insert** and **delete** each take $\Theta(1)$ time, but **pred** takes $\Theta(n)$ time in the worst case (if the item we are looking for is at the end of the list).

For a moment let's forget about being dynamic, and suppose we know all n items in advance and simply want to organize them to support faster **pred** queries. One option is of course to keep the items in sorted order in an array

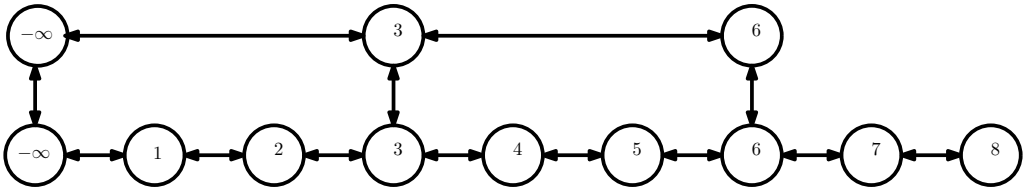

Figure 14.1: A two-level linked list. All items in a level are stored in sorted order.

then answer queries using binary search. Let's ignore that simple solution for now and try to just modify the linked list idea (since what we'll end up with we'll eventually be able to make dynamic). Consider the following idea, shown in Figure 14.1. Let's maintain one dummy element in our database with key $-\infty$ (it makes the implemntation easier – such dummy $\pm\infty$ elements to ease implementation are typically called "sentinels"). We will store all n items in the bottom linked list in sorted order. *In addition*, every $\sqrt{n}$th element in the list will be promoted one level up, so that we have also have an upper linked list on $\sqrt{n}$ elements. To perform a query, we start at ∞ in the upper linked list and move to the right until we've gone too far. At that point, we move down to the bottom level and keep going right until either (1) we've found the item, or (2) have gone too far (reaching a value bigger than it). The running time for answering **pred** queries is proportional to the number of links we traverse, which in the worst case is $2\sqrt{n} = \Theta(\sqrt{n})$.

The idea above can be generalized. Rather than just having two levels, we can have k levels. The top most level, level 1, has evenly spaced keys spaced with gaps of size $n^{1-1/k}$. In general level i has evenly spaced keys with gaps of size $n^{1-i/k}$. The number of linked that need to be traversed at the top level is then at most $n^{1/k}$, which is the size of the top level. For $i > 1$, the number of links traversed at level i is at most $n^{1-i/k}/n^{1-(i-1)/k} = n^{1/k}$. Thus we traverse at most $n^{1/k}$ items at each level. Since there are k levels, the total time to answer a query is $\Theta(kn^{1/k})$. By choosing $k = \log_2 n$, we have $kn^{/1k} = \log_2 n \cdot n^{1/\log_2 n} = 2\log_2 n$. To see this, just note $\log(n^{1/\log_2 n}) = (1/\log_2 n) \cdot \log_2 n = 1$. In fact when $k = \log_2 n$, the data structure essentially becomes a perfect binary tree (each gap between successive nodes x, y at level i has exactly 1 node z between them at level $i+1$; thus x, z can be seen as the children of x in a binary tree).

The trouble with the scheme above is the same as that with perfect binary trees. As we insert items in the data structure, the perfect balance that we so carefully constructed (every $n^{1/k}$ items is promoted upward) quickly becomes destroyed. In binary search trees this is typically fixed by carrying out some rebalancing operations after every update, such as with red-black trees. The solution we will describe now is the skip list, which is a randomized data structured invented by William Pugh in 1989.

The skip list works as follows. Just as in Figure 14.1, at the lowest level we keep all n items in a linked list in sorted order, with $-\infty$ in the front. Then, for each of the n items, we flip a fair coin to decide whether or not to

promote it one level upward to the next linked list. Then for all items which were promoted, we again for each of them flip a fair coin to decide which ones to promote again, etc. At some high level no items will be promoted any more, and that will determine our whole data structure. For ease of implementation, we assume that the $-\infty$ node is always promoted and appears at the beginning of the linked list at each level.

We now move on to the analysis of the skip list. For ease of description, we assume the items in the database have keys $1, 2, \ldots, n$.

Lemma 14.1 *The expected space consumption of a skip list is $O(n)$.*

Proof: Let X_i be the number of copies of item i in the data structure. Then note that the space consumption is proportional to $\sum_{i=1}^{n} X_i$. Then we have that the expected space is on the order of

$$\mathbb{E} \sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \mathbb{E} X_i$$

by linearity of expectation. Now note that X_i is distributed according to the following: we repeatedly flip a fair coin until seeing tails for the first time then let X_i be the number of flips. Then $\mathbb{E} X_i = 2$, and thus the expected space consumption is at most $2n$. ■

Now we analyze the expected query time.

Theorem 14.2 *For any item $i \in \{1, \ldots, n\}$, the expected time to query i is $O(\log n)$.*

Proof: As we query for i, we start at the $-\infty$ at the top most level and keep moving right (touching more elements on the current level) and downward (moving to a lower level). We always make right moves unless the item to the right is bigger than i, in which case we make a down move (or halt if we are at the bottom level). Thus the running time is equal to the number of right moves plus the number of down moves. The number of down moves is certainly at most H, the number of levels of the data structure. The number of right moves is equal to the number of distinct items we touch while executing the query. Let Y_j be a random variable which is 1 if we ever touch item j while processing the query on i, and $Y_j = 0$ otherwise. Thus the running time to query i is proportional to

$$H + \sum_{j=1}^{i-1} Y_j.$$

Thus by linearity of expectation, the expected query time is at most

$$\mathbb{E} H + \sum_{j=1}^{i-1} \mathbb{E} Y_j. \tag{14.1}$$

Let us first analyze $\mathbb{E}H$. Since H is a nonnegative integer random variable, we have

$$\begin{aligned}\mathbb{E}H &= \sum_{k=0}^{\infty} \mathbb{P}(H > k) \\ &= \sum_{k=0}^{\log_2 n - 1} \underbrace{\mathbb{P}(H > k)}_{\leq 1} + \sum_{k=\log_2 n}^{\infty} \mathbb{P}(H > k) \\ &= \log_2 n + \sum_{k=\log_2 n}^{\infty} \mathbb{P}(H > k) \end{aligned} \tag{14.2}$$

Now, let h_r denote the height of item r (where the bottom level is at height 0). Then

$$\begin{aligned}\mathbb{P}(H > k) &= \mathbb{P}(\exists r \in \{1, \ldots, n\} : h_r > k) \\ &\leq \sum_{r=1}^{n} \mathbb{P}(h_r > k) \\ &= \frac{n}{2^{k+1}}\end{aligned}$$

since for $h_r > k$ to occur we must have promoted r at least $k+1$ times. Turning back to Eq. (14.2),

$$\begin{aligned}\mathbb{E}H &\leq \log_2 n + \sum_{k=\log_2 n}^{\infty} \frac{n}{2^{k+1}} \\ &= \log_2 n + \sum_{t=1}^{\infty} \frac{1}{2^t} \text{ (letting } t = k + 1 - \log_2 n) \\ &= \log_2 n + 1\end{aligned}$$

Now we bound $\mathbb{E}Y_j$. Item j is touched while querying for i if and only if all items in the range $(j, i]$ have heights at most h_j. Thus

$$\begin{aligned}\mathbb{E}Y_j &= \mathbb{P}(Y_j = 1) \\ &= \sum_{t=0}^{\infty} \mathbb{P}((h_j = t) \wedge (\forall r \in (j, i]\ h_r \leq h_j)) \\ &= \sum_{t=0}^{\infty} \left[\mathbb{P}(h_j = t) \cdot \prod_{r=j+1}^{i} \mathbb{P}(h_r \leq h_j) \right] \\ &= \sum_{t=0}^{\infty} \frac{1}{2^{t+1}} \left(1 - \frac{1}{2^{t+1}}\right)^{i-j} \\ &\leq \sum_{t=0}^{\infty} \frac{1}{2^{t+1}} e^{-\frac{(i-j)}{2^{t+1}}} \\ &\leq \sum_{t=0}^{\lfloor \log_2 (i-j) \rfloor} \underbrace{\frac{1}{2^{t+1}} e^{-\frac{(i-j)}{2^{t+1}}}}_{z_t} + \underbrace{\sum_{t=\lfloor \log_2 (i-j) \rfloor + 1}^{\infty} \frac{1}{2^{t+1}}}_{O(\frac{1}{i-j})}\end{aligned} \tag{14.3}$$

We claim that $\sum_{t=0}^{\lfloor \log_2(i-j) \rfloor} z_t$ is also $O(1/(i-j))$. To see this, we claim that the summation is decaying at faster than a geometric rate as we decrease from $t = \lfloor \log_2(i-j) \rfloor$. To see this, fix some integer $q > 0$ and look at the ratio z_{t-q}/z_t. This ratio is at most $2^q e^{-2^q+1}$, which if you remember your little-oh's, is $o(1/2^q)$. Thus we can upper bound $\sum_{t=0}^{\lfloor \log_2(i-j) \rfloor} z_t$ by a geometric series dominated by its last term.

Returning to Eq. (14.1), and letting k denote $i-j$, the expected query time is at most for some constant $C > 0$

$$\begin{aligned}
\mathbb{E}H + \sum_{j=1}^{i-1} \mathbb{E}Y_j &\leq C(\log n + \sum_{j=1}^{i-1} \frac{1}{i-j}) \\
&= C(\log n + \sum_{k=1}^{i-1} \frac{1}{k}) \\
&\leq C(\log n + 1 + \int_1^{i-1} \frac{1}{k}) \\
&= C(\log n + 1 + \ln(i-1)) \\
&= O(\log n)
\end{aligned}$$

It is worth remembering that the sum $\sum_{k=1}^{b} \frac{1}{k}$ is typically called the *bth Harmonic number*, and what we have just seen above (as well as in the QuickSort analysis) is that it is $O(\log b)$. ■

14.2 Hashing

Consider the following data structural problem, usually called the *dictionary problem*. We have a set of items. Each item is a (key, value) pair. Keys are in the range $\{1, \ldots, U\}$, and values are arbitrary. A data structure supporting the following operations is called a *dynamic dictionary*.

- **insert(k,v):** insert a new item into the database with key k and value v. If an item with key k already exists in the database, update its value to v.
- **delete(x):** delete item x from the database (we assume x is a pointer to the item)
- **query(k):** return the the value associated with key k, or `null` if key k is not in the database

Note that the predecessor problem is strictly harder than the dictionary problem if we assume for the predecessor problem that inserting a key already in the database results in a no-op (since then a data structure for predecessor can also be used for dictionary operations).

One randomized solution to the dictionary problem is *hashing*. A hash family $\mathcal{H}$ is a set of functions $h : \{1, \ldots, U\} \to \{1, \ldots, m\}$. The basic idea of the solution is to maintain an array of size m where an item with key k is

stored in the $h(k)$th position of the array. Unfortunately this does not work as is due to *collisions*: two distinct keys k, k' in our database may have $h(k) = h(k')$, so how do we resolve this collision? In a *hash table with chaining*, we initialize some array A of size m, where $A[i]$ stores the pointer to the head of a doubly linked list. We pick some h uniformly at random from $\mathcal{H}$. Then $A[i]$ will be a doubly linked list containing (key,value) pairs of *all* items whose key k satisfies $h(k) = i$. That is, colliding items are stored in the same linked list. To insert a (k, v) pair, we simply insert this new item to the head of the list at $A[h(k)]$. To query k, we traverse the list at $A[h(k)]$ until we find an item with key k (or discover that none exists). Deletion splices the item out of the doubly linked list when it finds it.

Of course, the above scheme will not always be efficient. For example, consider $\mathcal{H}$ only containing a single hash function h satisfying $h(i) = 1$ for all i. Then our entire data structure is a single linked list. However, if $\mathcal{H}$ is *nice* in a certain way and m is sufficiently large, we will be able to prove good guarantees.

Definition 14.3 *We say a family $\mathcal{H}$ of hash function mapping $\{1,\ldots,U\}$ into $\{1,\ldots,m\}$ is* universal *if for all $1 \leq x < y \leq U$,*

$$\mathbb{P}_{h\in\mathcal{H}}(h(x) = h(y)) \leq \frac{1}{m},$$

where h is chosen uniformly at random from $\mathcal{H}$.

Example 14.4 The set $\mathcal{H}$ of *all* functions mapping $\{1,\ldots,U\}$ into $\{1,\ldots,m\}$ is universal and has $|\mathcal{H}| = m^U$. That is, an element $h \in \mathcal{H}$ can be described using $\log|\mathcal{H}| = U\log m$ bits of space.

Example 14.5 Another option is to pick some prime $p \geq U$ and define $h_a(x) = (ax \mod p) \mod m$. Then we let $\mathcal{H} = \{h_a : 0 < a < p\}$. Then $|\mathcal{H}| = p - 1$, and we can choose p for example to be at most polynomial in U, so a random $h \in \mathcal{H}$ can be represented using $\log|\mathcal{H}| = O(\log U)$ bits. The analysis of this scheme requires some abstract algebra beyond the scope of this course, but this family turns out to be "almost universal", in the sense that for any $x \neq y$, $\mathbb{P}_{h\in\mathcal{H}}(h(x) = h(y)) \leq C/m$ for some constant C that tends to 1 as p grows large (and for most applications, this weaker property suffices).

Theorem 14.6 *Consider a hash table with chaining on a database with n items using a universal hash family $\mathcal{H}$ with $m \geq n$. Then executing a query takes expected time $O(1+T)$, where T is the cost of evaluating a hash function $h \in \mathcal{H}$.*

Proof: Let the query be on some key k. A query performs one hash evaluation, taking time T, followed by traversing the list at $A[h(k)]$. For $i = 1,\ldots,n$ let X_i be a random variable which is 1 if the ith key k_i in the database has

$h(k_i) = h(k)$, and X_i is 0 otherwise. The the running time of a query is proportional to

$$T + \sum_{i=1}^{n} X_i.$$

Thus by linearity of expectation, the expected running time of a query is proportional to

$$T + \sum_{i=1}^{n} \mathop{\mathbb{E}}_{h} X_i = T + \sum_{i=1}^{n} \mathop{\mathbb{P}}_{h \in \mathcal{H}}(h(k_i) = h(k)) \leq T + \sum_{i=1}^{n} \frac{1}{m}$$

Noting that $m \geq n$, the above is $T + 1$. ■