Skip Lists

This material is not covered in our textbook but you can read about skip-lists in Section 6.3 of Ordered Lists in Data Structures and Their Algorithms by LEWIS, DENENBERG.

In searching it is important that the data be stored in easily accessible order. We will see that even in this situation it is possible to take advantage of randomization. In this section we discuss skip lists, which use randomly constructed forward pointers to speed up a linked list to a fast dictionary implementation.

Skip List Data Structure. The idea of skip lists is best introduced by taking another look at binary search in a linear array. Interpret binary search as following pointers from left to right, as shown in Figure 34. If we overshoot we follow a shorter pointer instead.

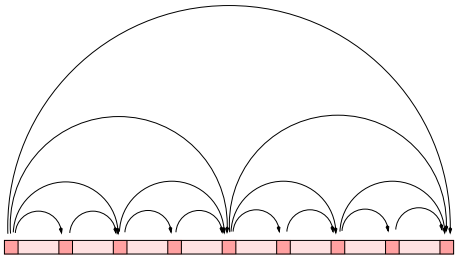

Figure 34: We search from left to right. The first and last elements in the array are dummy nodes.

A *skip list* is a linked list that provides extra pointers to support a search strategy similar to the one shown in Figure 34. The pointers are constructed using randomization as illustrated in Figure 35. Each node stores a small table of forward pointers. The number of forward pointers in the tables follows roughly the pattern of modified binary search but is less rigid. We use the following record structure in the pseudo-code algorithms that manipulate skip lists.

```
struct Node {item val; Node** fpt};
typedef Node* List.
```

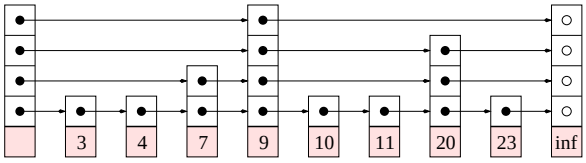

Figure 35: Each node of the skip list has a table of $H+1$ pointers, where H is the height of the node. The maximum height in the example is $m = 3$, and the first and last (dummy) nodes have this maximum height.

Here *ftp* is a table of pointers to nodes and the size of this table varied from node to node. An efficient implementation would embed all tables in a single linear array. Each node would store only the first and the last array indices of its table elements.

Distribution of Table Sizes. A node has *height* H if it stores a table of $H+1$ forward pointers. Ideally, we would like one node of height $k = \lfloor \log_2 n \rfloor$, two nodes of height $k-1$, four nodes of height $k-2$, etc. Instead of regulating the height we use randomness. Let COIN be a boolean function that randomly returns TRUE or FALSE, each with probability one half. We use function COIN to compute a random height drawn from a distribution that decays exponentially. To avoid tables that are excessively high, we introduce a maximum permitted height, which we denote as m.

```
int RANDOMHEIGHT
  H = 0;
  while COIN and H < m do H++ endwhile;
  return H.
```

We assume m is chosen roughly equal to the binary logarithm of the number of items. As claimed, the probability

decreases exponentially with the height:

$$\text{Prob}[H = i] \quad = \quad \begin{cases} 1/2^{i+1} & \text{for } 0 \le i < m \\ 1/2^m & \text{for } i = m. \end{cases}$$

It will be important that the heights of the nodes are chosen independent of each other. The expected height of a node is

$$\begin{aligned} \mathrm{E}[H] &= \sum_{i=0}^{m} i \cdot \text{Prob}[H = i] \\ &= \sum_{i=0}^{m-1} \frac{i}{2^{i+1}} + \frac{m}{2^m}. \end{aligned}$$

We can compute the sum by index transformation:

$$\begin{aligned} \sum_{i=0}^{m-1} \frac{i}{2^{i+1}} &= \sum_{i=1}^{m} \frac{i-1}{2^i} \\ &= 2\sum_{i=0}^{m} \frac{i}{2^{i+1}} - \sum_{i=1}^{m} \frac{1}{2^i} \\ &= 1 - \frac{m+1}{2^m}. \end{aligned}$$

This implies $\mathrm{E}[H] = 1 - \frac{1}{2^m} < 1$. Since the expectations of probability distributions are additive, the sum of heights taken over all n nodes is $\mathrm{E}[\sum_{i=1}^{n} H_i] = \sum_{i=1}^{n} \mathrm{E}[H_i] < n$. In other words, the expected number of forward pointers is $\mathrm{E}[\sum_{i=1}^{n}(H_i + 1)] < 2n$.

Search. Searching in a skip list is done similar to binary search, substituting forward pointers for index calculations:

```
Node *SEARCH(item a, Node *L)
  for i = m downto 0 do
    while L → fpt[i] → val < a do
      L = L → fpt[i]
    endwhile
  endfor;
  if L → fpt[0] → val = a then return L → fpt[0]
                          else return NULL
  endif.
```

Try the search function for $a = 11$ and for $a = 24$ on the skip list in Figure 35. By sheer bad luck, it is possible that all nodes (other than the two dummy nodes) have height 0. In this case, the search is the same as in a linked list, that is, it still works but is slow. In the expected case, it is much faster than that. A search starts at the left dummy node L and there are two types of steps:

1. follow a forward pointer on the same level;
2. drop to the next lower level of forward pointers.

When we follow a pointer we always come in at the highest level of the node. (Why?) We compute the expected number of steps by tracing a search path backwards. The probability of taking a vertical step upwards is $\frac{1}{2}$. The reason is that we take this step if and only if we did not yet reach the highest level at that node. Since the height is computed randomly, this is the same as the probability of getting TRUE from the coin-flipping function, which is $\frac{1}{2}$. The probability of taking a horizontal step backwards is then $1 - \frac{1}{2} = \frac{1}{2}$. Let now $C(j)$ be the expected number of steps in a backward path that rises j levels. Then

$$\begin{aligned} C(j) &= \frac{1}{2}(1 + C(j)) + \frac{1}{2}(1 + C(j-1)) \\ &= 2 + C(j-1) \\ &= 2j \end{aligned}$$

because $C(0) = 0$. In words, we get from level 0 to level $m = \lfloor \log_2 n \rfloor$ in expected $2m$ steps. If this node is not the header, L, we still need to follow backwards all the way to the left. The expected number of nodes with height m (not counting the two dummy nodes) is

$$n \cdot \frac{1}{2^m} \;\le\; n \cdot \frac{1}{2^{\log_2 n - 1}} \;\le\; n \cdot \frac{2}{n} \;=\; 2.$$

The expected length of a search path is therefore at most $2m + 2$. The analysis assumes that the number of items does not vary much. If we expect the skip list to grow significantly through a large number of insertions we may want to choose m significantly larger than $\lfloor \log_2 n \rfloor$. In this case we should start the search at the highest height of any real node instead of level m so that the above analysis still applies.

Insertion and Deletion. Instead of constructing a skip list in one shot from an input sequence, it is built incrementally by adding one item at a time. Since skip lists support searching, we can add the items in any order we like. Consider inserting the item 18 into the skip list of Figure 35. We first find its predecessor, which is the node storing 11. Before adding the new node we randomly pick its height, say $H = 2$. The new node is then added into the linked lists at all levels up to H, as illustrated in Figure 36.

When we insert a new item a, we search for its location and then add a into the lists at the various levels. We use an auxiliary array $F[0..m]$ that accumulates the predecessors

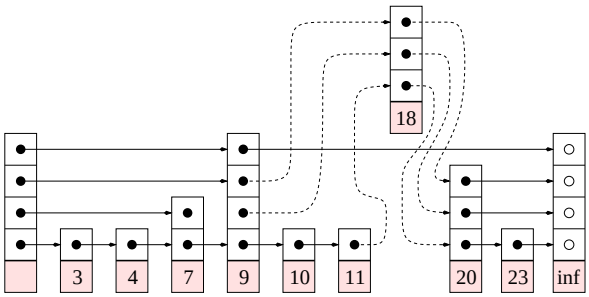

Figure 36: The dashed pointers are added during the insertion of 18.

of the new node at the various levels. At the end of the process, we insert the new node into the list at every level between 0 and the randomly chosen height H of the new node.

```
void INSERT(item a, Node *L)
  for i = m downto 0 do F[i] = L;
    while L → fpt[i] → val < a do
      L = L → fpt[i]; F[i] = L
    endwhile
  endfor;
  if L → fpt[0] → val ≠ a then
    H = RANDOMHEIGHT; p = Node(a, H);
    for i = 0 to H do
      p → fpt[i] = F[i] → fpt[i]; F[i] → fpt[i] = p
    endfor
  endif.
```

Function $\texttt{Node}(a, H)$ allocates memory for a new node with space for H forward pointers. Deleting an item a is similar to inserting it.

```
void DELETE(item a, Node *L)
  for i = m downto 0 do F[i] = L;
    while L → fpt[i] → val < a do
      L = L → fpt[i]; F[i] = L
    endwhile
  endfor;
  if L → fpt[0] → val = a then
    p = L → fpt[0]; i = 0;
    while i ≤ m and F[i] → fpt[i] = p do
      F[i] → fpt[i] = p → fpt[i]; i++
    endwhile
  endif.
```

Note that it is not necessary to know the height of p to remove it from the skip list.

Summary. Function SEARCH takes expected time $O(\log n)$, and so do functions INSERT and DELETE because they just search and then either add or remove a single node of height at most $m = O(\log n)$. The expected number of forward pointers for n nodes is less than $2n$, or about the same as for a doubly-linked list.

An interesting phenomenon about randomized data structures is their highly predictable performance. Although the worst-case number of steps for searching in a skip list is $\Theta(n)$, experiments show that the number of steps is almost all the time right around the expected value.