Lecture 11: Skip Lists

Read: Section 10.4.2 in Weiss, and Samet's notes Section 5.1.

Recap: So far we have seen three different method for storing dictionaries. Unbalanced binary trees are simple, and worked well on average, but an adversary could force very bad running time. AVL trees guarantee good performance, but are somewhat harder to implement. Splay trees provided an interesting alternative to AVL trees, because they are simple and self-organizing.

Today we are going to continue our investigation of different data structures for storing dictionaries. The data structure we will consider today is called a *skip list*. This is among the most practical of the data structures we have seen, since it is quite simple and (based on my own experience) seems to be the fastest of all these methods. A skip list is an interesting generalization of a linked list. As such, it has much of the simplicity of linked lists, but provides optimal $O(\log n)$ performance. Another interesting feature of skip lists is that they are a *randomized* data structure. In other words, we use a random number generator in creating these trees. We will show that skip lists are efficient in the expected case. However, unlike unbalanced binary search trees, the expectation has nothing to do with the distribution of the keys. It depends only on the random number generator. Hence an adversary cannot pick a sequence of operations for our tree that will always be bad. And in fact, the probability that a skip list might perform badly is very small.

Perfect Skip Lists: Skip lists began with the idea, "how can we make sorted linked lists better?" It is easy to do operations like insertion and deletion into linked lists, but it is hard to locate items efficiently because we have to walk through the list one item at a time. If we could "skip" over lots of items at a time, then we could solve this problem. One way to think of skip lists is as a hierarchy of sorted linked lists, stacked one on top of the other.

To make this more concrete, imagine a linked list, sorted by key value. Let us assume that we have a special *sentinel node* at the end, called *nil*, which behaves as if it has a key of ∞. Take every other entry of this linked list (say the even numbered entries) and lift them up to a new linked list with $1/2$ as many entries. Now take every other entry of this linked list and lift it up to another linked with $1/4$ as many entries as the original list. The head and nil nodes are always lifted. We could repeat this process $\lceil \lg n \rceil$ times, until there are is only one element in the topmost list. To search in such a list you would use the pointers at high levels to "skip" over lots of elements, and then descend to lower levels only as needed. An example of such a "perfect" skip list is shown below.

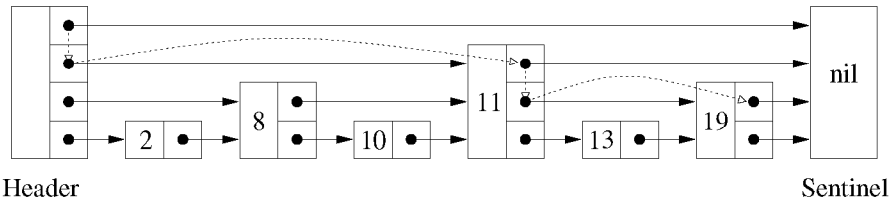

Figure 32: Perfect skip list.

To search for a key x we would start at the highest level. We scan linearly along the list at the current level i searching for first item that is greater than to x (recalling that the nil key value is ∞). Let p point to the node just before this step. If p's data value is equal to x then we stop. Otherwise, we descend to the next lower level $i-1$ and repeat the search. At level

0 we have all the keys stored, so if we do not find it at this level we quit. For example, the figure shows the search for $x = 19$ in dotted lines.

The search time in the worst case may have to go through all $\lceil \lg n \rceil$ levels (if the key is not in the list). We claim this search visits at most two noded per level of this perfect skip list. This is true because you know that at the previous (higher) level you lie between two consecutive nodes p and q, where p's data value is less than x and q's data value is greater than x. Between any two consecutive nodes at the one level there is exactly one new node at the next lower level. Thus as we descend a level, the search will visit the current node and at most one additional node. Thus the there are at most two nodes visited per level and $O(\log n)$ levels, for a total of $O(\log n)$ time.

Randomized Skip Lists: The problem with the data structure mentioned above is that it is exactly balanced (somewhat like a perfectly balanced binary tree). The insertion of any node would result in a complete restructuring of the list if we insisted on this much structure. Skip lists (like all good balanced data structures) allow a certain amount of imbalance to be present. In fact, skip lists achieve this extra "slop factor" through randomization.

Let's take a look at the probabilistic structure of a skip list at any point in time. (By the way, this is not exactly how the structure is actually built, but it serves to give the intuition behind its structure.) In a skip list we do not demand that exactly every other node at level i be promoted to level $i + 1$, instead think of each node at level i tossing a coin. If the coin comes up heads (i.e. with probability $1/2$) this node promotes itself to the next higher level linked list, and otherwise it stays where it is. Randomization being what it is, it follows that the expected number of nodes at level 1 is $n/2$, the expected number at level 2 is $n/4$, and so on. Furthermore, since nodes appear randomly at each of the levels, we would expect the nodes at a given level to be well distributed throughout (not all bunching up at one end). Thus a randomized skip list behaves much like an idealized skip list in the expected case. The search procedure is exactly the same as it was in the idealized case. See the figure below.

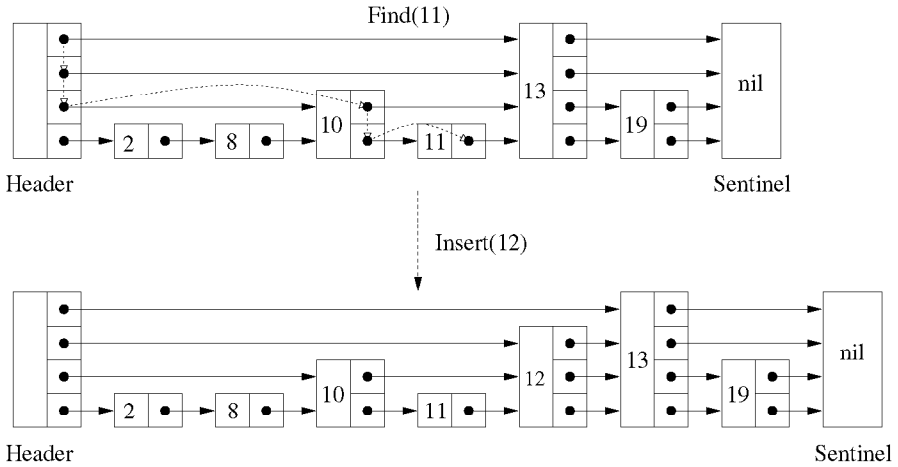

Figure 33: Randomized skip list.

The interesting thing about skip lists is that it is possible to insert and delete nodes into a list, so that this probabilistic structure will hold at any time. For insertion of key x we first do a search on key x to find its immediate predecessors in the skip list (at each level of the structure). If x is not in the list, we create a new node x and insert it at the lowest level of

the skip list. We then toss a coin (or equivalently generate a random integer). If the result is tails (if the random number is even) we stop. Otherwise we insert x at the next higher level of the structure. We repeat this process until the coin comes up tails, or we hit the maximum level in the structure. Since this is just repeated link list insertion the code is very simple. To do deletion, we simply delete the node from every level it appears in.

Note that at any point in time, the linked list will have the desired probabilistic structure we mentioned earlier. The reason is that (1) because the adversary cannot see our random number generator, he has no way to selectively delete nodes at a particular level, and (2) each node tosses it's coins independently of the other nodes, so the levels of nodes in the skip list are independent of one another. (This seems to be one important difference between skip lists and trees, since it is hard to do anything independently in a tree, without affecting your children.)

Analysis: The analysis of skip lists is an example of a *probabilistic analysis*. We want to argue that in the expected case, the search time is $O(\log n)$. Clearly this is the time that dominates in insertion and deletion. First observe that the expected number of levels in a skip list is $O(\log n)$. The reason is that at level 0 we have n keys, at level 1 we expect that $n/2$ keys survive, at level 2 we expect $n/4$ keys survive, and so forth. By the same argument we used in the ideal case, after $O(\log n)$ levels, there will be no keys left.

The argument to prove the expected bound on the search time is rather interesting. What we do is look at the reversal of the search path. (This is a common technique is probabilistic algorithms, and is sometimes called *backwards analysis*.) Observe that the forward search path drops down a level whenever the next link would take us "beyond" the node we are searching for. When we reverse the search path, observe that it will always take a step up if it can (i.e. if the node it is visiting appears at the next higher level), otherwise it will take a step to the left.

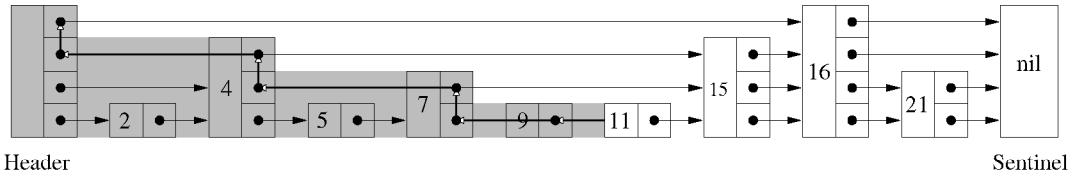

Figure 34: Reversal of search path to $x = 11$.

Now, when we arrive at level i of any node in the skip list, we argue that the probability that there is a level above us is just $1/2$. The reason is that when we inserted the node, this is the probability that it promoted itself to the next higher level. Therefore with probability $1/2$ we step to the next higher level. With the remaining probability $1 - (1/2) = 1/2$ we stay at the same level. The expected number of steps needed to walk through j levels of the skip list is given by the following recurrence.

$$C(j) = 1 + \frac{1}{2}C(j-1) + \frac{1}{2}C(j).$$

The 1 counts the current step. With probability $1/2$ we step to the next higher level and so have one fewer level to pass through, and with probability $1/2$ we stay at the same level. This can be rewritten as

$$C(j) = 2 + C(j-1).$$

By expansion it is easy to verify that $C(j) = 2j$. Since j is at most the number of levels in the tree, we have that the expected search time is at most $O(\log n)$.

Implementation Notes: One of the appeals of skip lists is their ease of implementation. Most of procedures that operate on skip lists make use of the same simple code that is used for linked-list operations. One additional element is that you need to keep track of the level that you are currently on. The way that this is done most efficiently, is to have nodes of variable size, where the size of a node is determined randomly as it is created. We take advantage of the fact that Java (and C++) allow us to dynamically allocated arrays of variable size.

The skip list object consists of a header node (`header`) and the constructor creates a *sentinel* node, whose key value is set to some special *infinite* value (which presumably depends on the key type). We assume that the constructor is given the maximum allowable number of levels. (A somewhat smarter implementation would adaptively determine the proper number of levels.)

Skip List Classes

```
class SkipListNode {
    Element data;                   // key data
    SkipListNode forward[];         // array of forward pointers

                                    // Constructor given data and level
    SkipListNode(Element d, int level) {
        data = d
        forward = new SkipListNode[level+1];
    }
}

class SkipList {
    int maxLevel;                   // maximum level
    SkipListNode header;            // header node

    SkipList(int maxLev) {          // constructor given max level number
maxLevel = maxLev;          // allocate header node
        header = new SkipListNode(null, maxLevel);
                                    // append the "nil" node to the header
        SkipListNode sentinel = new SkipListNode(INFINITY, maxLevel);
        for (int i = 0; i <= maxLevel; i++)
            header.forward[i] = sentinel;
    }
}
```

The code for finding an element in the skip list is given below. Observe that the search is very much the same as a standard linked list search, except for the loop that moves us down one level at a time.

Find an object in the skip list

```
Element find(Element key) {
    SkipListNode current = header;      // start at header
                                        // start search at max level
    for (int i = maxLevel; i >= 0; i--) {
        SkipListNode next = current.forward[i];
        while (next.data < key) {       // search forward on level i
            current = next;
            next = current.forward[i];
        }
    }
```

```
    current = current.forward[0];       // this must be it

    if (current.data == key) return current.data;
    else return null;
}
```

It is worth noting that you do not need to store the of a node as part of the forward pointer field. The skip list routines maintain their own knowledge of the level as they move around the data structure. Also notice that (if correctly implemented) you will never attempt to index beyond the limit of a node.

We will leave the other elements of the skip list implementation as an exercise. One of the important elements needed for skip list insertion is the generation of the random level number for a node. Here is the code for generating a random level.

Compute a Random Level

```
int generateRandomLevel() {
    int newLevel = 0;
    while (newLevel < maxLevel && Math.random() < 0.5) newLevel++;
    return newLevel;
}
```