Skip Lists

This is yet another data structure that allows for inserts, searches and deletes in an average of O(log n) time, where n is the total number of items in the skip list. Interestingly enough, this data structure is realized with a mesh of linked lists.

Unlike some of the other data structures we have looked at, a skip list uses randomization. Thus, if you give the exact same input to a skip list twice, you MAY actually get different behavior both times. (An example of an algorithm you have seen before that is randomized is Quick Sort when the partition element is picked randomly.)

<u>Skip List Definition</u>

A skip list consists of a series of lists $\{S_0, S_1, \ldots, S_h\}$. The list S_0 starts with the special value $-\infty$ and continues with all the numbers in numerical order and is terminated by the special value $+\infty$. (Note: These two special values, $-\infty$ and $+\infty$ are always stored in the first and last nodes respectively of every list.)

Each list S_i stores a subset of the values stored in S_{i-1}. You can visualize list S_i to appear directly above list S_{i-1}. Furthermore, each list will be connected to adjacent lists through vertical links between lists. In particular, if a value appears in both in S_i and S_{i-1}, then there will be a link between these two nodes.

Finally, in S_h will only contain $-\infty$ and $+\infty$.

Here is a visual representation of a skip list:

```
S4   -∞                  ->                                        ∞
      |                                                            |
S3   -∞         ->    3                                            ∞
      |               |                                            |
S2   -∞         ->    3     ->           12    ->           20->   ∞
      |               |                  |                   |     |
S1   -∞ ->  1   ->    3->   8 ->         12    ->           20->   ∞
      |     |         |     |            |                   |     |
S0   -∞ ->  1->2->    3->   8 ->9 ->     12->15  ->18   ->20->     ∞
```

So how do we decide what elements get repeated in higher lists?

Whenever we insert an element, we will first insert the element in the list S_0. (Although it may look like it, this will NOT take $\theta(n)$ time.) Once this is done, we will randomly pick a real number in between 0 and 1. If it is greater than .5, we will copy this value in S_1. This involved creating a new node, linking that new node to the node with the same value in S_0, and then linking the node in between the two appropriate values in S_1. Now, we will continue and create another random real number in between 0 and 1. If it is greater than .5 we'll repeat this process for S_2. We continue until we get a random number less than .5 and stop at that point. So, in the worst case, this algorithm could go forever, but technically speaking, the probability of that occurring is 0.

So what does this do?

It leaves about ½ as many nodes in S_i as in S_{i-1}. Thus, if the skip list is storing n values, there should roughly be n/2 values in S_1. Using the repeated halving principle, we see that the expected

"height" of the skip tree is logarithmic in n. In order to maintain a skip list, we will have to provide four operations for any node in the list in O(1) time. These are:

1) after(p)
2) before(p)
3) below(p)
4) above(p)

The first two provide a way to navigate between elements in a list, and the last two provide a way to navigate between lists. In order to implement a skip list with these features, a connected mesh of nodes with four separate references would suffice.

Searching

We need to efficiently use the setup of each list to help us find a value. To solve this problem, consider searching in a Multi-way Tree. We will go through the list of values at a node until we have gone too far. When we have, we know to traverse the link "right before" the node that stored the value that was too big:

```
              4,     10,    15,    25
        /           |       |      |       \
     1, 3        5, 6,8   11    20,24    28, 30, 33, 35
```

If we are searching for 23, we will look at 4, then 10, then 15. When we hit 25, we know if 23 is in the tree, it must be in the subtree to the right of the 15 and the left of the 25. Follow this link down. Now, after you look at the 20 and then the 24, you would search down the subtree to the right of 20 and left of 24, but this is null. Thus, the value isn't in the tree.

We will utilize this same idea in searching for a value in a skip list.

```
S4   -∞                ->                          ∞
      |                                            |
S3   -∞       ->   3                               ∞
      |            |                               |
S2   -∞       ->   3    ->        12    ->      20-> ∞
      |            |              |              |   |
S1   -∞ ->  1   ->   3->  8 ->      12    ->      20-> ∞
      |     |      |     |        |              |   |
S0   -∞ ->  1->2->   3->  8 ->9 ->   12->15  ->18  ->20-> ∞
```

Now, imagine searching for 18 in this skip list. First you start at the beginning of S_4 and traverse down to S_3 along the link of -∞. (This is because the next value in S_4, ∞, is too large.) Now, go to the 3 in S_3. Since the next value in S_3 is too big, follow the link down to 3 in S_2. Continue along S_2 to 12. Since 20 is too big, follow the link down to 12 in S_1. The same thing occurs here and we go down to the node storing 12 in S_0. At this point, since you are at the bottom of the list, continue forward until you find 18. If you pass over the value being searched in S_0, then the value is not in the list.

Insertion

This works very similar to searching. Imagine searching for the value and getting to S_0. Here you can find the correct location to insert the value. Once you insert this value in S_0, continue by creating a random number and if necessary inserting the same number in S_1, S_2, etc. When doing this, all the necessary references must be updated. It can be shown that the extra amount of work done by this part of the algorithm (storing repeated nodes in higher lists) after a single insertion is O(1) time on average.

To see this, note that the approximate number of physical nodes in the skip list will be $(n + n/2 + n/4 + \ldots + 1) + 2\log n \sim$

2(n+logn) using the sum of an infinite geometric series. (Note: the 2logn accounts for the -∞ and +∞ nodes in each of the approximately log n lists.)

<u>Removal</u>

Once again, implement the search procedure. In the book they make the search procedure go down all the way to S_0, but this isn't entirely necessary. (The only situation this would be necessary is if you only stored entire records in S_0 and just stored keys to those records in the list above. This idea has the potential of saving some space, but also is more complicated since you'd have a linked mesh structure with different types of nodes.)

Instead, you will find a node in the highest list it appears first if you implement the search algorithm. (In order to stop the search algorithm at this point, change the pseudocode in the book to read "while (below(p) != null && key(p) != k)"))

If you do this, you can simply delete each desired node, one by one until you've deleted the node storing the value to be deleted in S_0. For example, if you were to remove 12, you would do the following steps:

```
S4   -∞                     ->                              ∞
      |                                                     |
S3   -∞          ->   3                                     ∞
      |               |                                     |
S2   -∞          ->   3    ->   del this 1st         20-> ∞
      |               |                |               |    |
S1   -∞ ->  1    ->   3->  8 ->    then this         20-> ∞
      |     |         |    |           |               |    |
S0   -∞ ->  1->2->    3->  8 ->9   finally 15  ->18  ->20-> ∞
```

<u>Maintaining the Top-most Level</u>

You must always have a reference to the top-most left-most node. (This is the -∞ on list S_h.) One way to control this level is to fix the maximum height of the skip list based on the number of elements in the list. You could do something like max(10, 2log n), where n is the number of elements stored. This keeps the height logarithmic. (Thus, if you inserted an element, and got 2log n consecutive random numbers over .5, which would indicate to "grow" a list beyond S_{2logn}, you don't do it. In essence, when you get to the top row, you don't flip a coin to see if you will add a new row.)

Or, you could simply allow the height of the tree to expand as inserted elements "build" towers. (Though it does seem silly to have a single tower that's more than one element taller than all the rest.) Either way, the expected amount of time for the three operations is O(log n).

Bounding the height of a Skip List

The probability that an element gets to S_i upon insertion is $(1/2)^i$. Given that there are n items in the list, the maximum probability that at least one of those items is on level is is $n/2^i$. (To see this, consider the inclusion-exclusion principle for probability. We can simply add the probabilities of each of the events, but when we do this, we have "overcounted" by counting the probability that multiple items end up in S_i several times. Thus, adding the probabilities as we have done is an overestimate of the actual probability, since the events are NOT mutually exclusive.)

Thus, the height of the skip list does NOT exceed clogn with probability $n/n^c = 1/n^{c-1}$. The book plugs in c=3 to show that the probability of a skip list exceeding the height 3logn is at most $1/n^2$.

Analyzing Search Time in a Skip List

Basically, each "move" in the search algorithm takes several steps down and several steps forward. The number of steps down is bounded by the height of the list, which is expected to be O(log n) as shown above. Now, we must calculate the expected number of forward steps.

Note that each new element examined at level i of the list could not have existed in level i+1. This is because we drop levels in the skip list if we've "gone too far" in the level above. When we drop down, the new nodes we will traverse going forward will NOT be in the level above, because we would have stopped then.

```
S4   -∞                  ->                                      ∞
      |                                                          |
S3   -∞        ->   3                                            ∞
      |             |                                            |
S2   -∞        ->   3    ->          12    ->          20-> ∞
      |             |                 |                 |    |
S1   -∞ ->  1  ->   3->  8 ->        12    ->          20-> ∞
      |     |       |    |            |                 |    |
S0   -∞ ->  1->2->  3->  8 ->9 ->    12->15  ->18   ->20-> ∞
```

If we are searching for 8 above, we know that since 12 was too big in S_2, it's not one of the new elements we'll see in S_2. In particular, each of the underlined elements above denotes the "search ranges" for each list. We you drop down a list, you are confining yourself to the search region between adjacent elements in the list above. What is the expected number of elements on S_{i-1} in between adjacent elements from S_i? Since an element is twice as likely to be in S_{i-1} as S_i, the density of elements is twice as much. This would infer an average of one extra element in between two adjacent ones from the previous list. Certainly this value can be bounded by 2 new elements. Thus the number of elements searched in each level of the skip list is O(1). It follows that the expected search time in a skip list is O(log n), where the list stores n values.

The expected number of physical nodes used in a skip list is at most 2n. Can you prove this?