Chapter C2: Skip Lists

Skip lists support a dictionary (and more) for data that has an underlying ordering. For example, they can do the dictionary operations on a sorted set. They were introduced by Pugh.

C2.1 Levels

The idea is to use a sorted linked list. Of course, the problem with a linked list is that finding the data and/or finding the place to insert the new data is \( O(n) \). So the solution is to have pointers that traverse the list faster (e.g. every 2nd node), and then have pointers that traverse the list even faster (e.g. every 4th node), and so on.

So we will have levels of linked lists. These get sparser as one goes up the levels. Importantly, each node in a level has a pointer to the corresponding node in the level below. Thus, the find operation starts in the topmost level, and moves down a level each time it finds the bracketing pair.

In an ideal scenario, the first linked list has only the median. The second linked list has the three quartiles. And so on. In this ideal scenario there are about \( \log n \) levels. This means that find is \( O(\log n) \), as one essentially does binary search.

But what about insertion: one insertion would ruin the ideal properties described above. One could try to “rebalance” if the median drifted too far out. But a simpler approach is to only require the properties to hold approximately. And to use randomness.

C2.2 Implementation of Insertion

One uses randomness to determine the membership of the levels. Specifically:

\[
\text{when inserting, toss a coin and add a node to the level above with probability } 1/2 \text{ (and recurse)}
\]

So we only have expected \( O(\log n) \) running times. (We will call it a Las Vegas algorithm.)

A natural implementation is to have an ordinary linked list at the bottom level. Each level above is a linked list with a down-pointer to the corresponding node in the lower level. To keep these levels all anchored, have a linked list of pointers to the start of each level, essentially acting as a collection of dummy nodes.
A simple if inelegant procedure for insert is to first see if the value is in the list (if one is prohibiting repeats). Then toss the coin(s) to determine how far up the levels it will go. And then start the insertion process in the topmost level.

Note that the randomness provides a good average run time, and a very low chance of bad behavior. And this is regardless of the actual data.

**Exercises**

1. Explain how one can add an int to each node that allows one to offer the rank operation. This method should return the value of specified rank in expected $O(\log n)$ time.