CMSC 420: Lecture 9
Randomized Search Structures: Skip Lists

More Randomized Search Structures: In this lecture, we continue our discussion of randomized search structures. In the previous lecture, we introduced the treap, which uses randomization to simulate the structure of a binary tree where keys are inserted in random order. Recall that such a data structure uses a random number generator to guide its operations and structure, and the running time is expressed as the expected running time for an arbitrary sequence of operations, averaged over all possible outcomes of the random number generator.

Since the random number generator is not under the control of the data structure's user, even a malicious user cannot engineer a sequence of operations that will guarantee a higher than expected running time. The only thing that could go wrong is having bad luck with respect to the random number generator.

In this lecture we will discuss a data structure that is reputed to be among the fastest data structures for ordered dictionaries, the skip list. (It also has the feature that it was designed by Bill Pugh, a former professor at the University of Maryland!)

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 costly to locate items efficiently because we have to walk through the list one item at a time. If we could "skip" over multiple of items at a time, however, then we could perform searches efficiently. Intuitively, a skip list is a data structure that encodes a collection of sorted linked lists, where links skip over 2, then 4, then 8, and so on, elements with each link.

To make this more concrete, imagine a linked list, sorted by key value. There are two nodes at either end of the list, called `head` and `tail`. Take every other entry of this linked list (say the even numbered entries) and extend it up to a new linked list with 1/2 as many entries. Now take every other entry of this linked list and extend it up to another linked with 1/4 as many entries as the original list, and continue this until no elements remain. The `head` and `tail` nodes are always lifted (see Fig. 1). Clearly, we can repeat this process $\lceil \lg n \rceil$ times. (Recall that "lg" denotes log base 2.)

The result is something that we will call the *ideal skip list*. Unlike the standard linked list, where you may need to traverse $O(n)$ links to reach a given node, in this list *any* node can be reached with $O(\log n)$ links from the `head`.

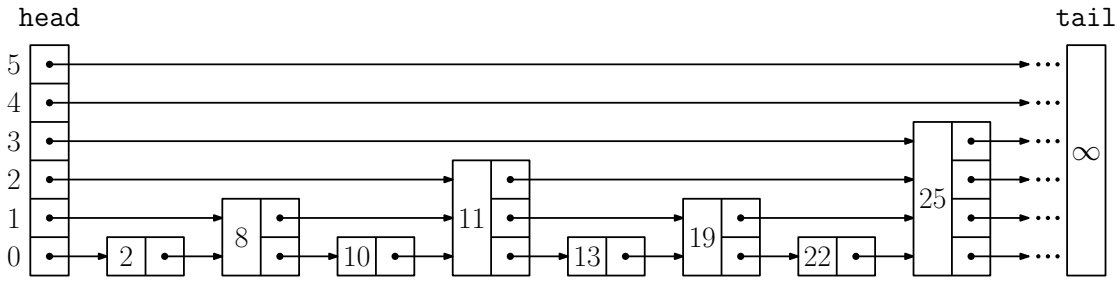

Fig. 1: The "ideal" skip list.

To search for a key x, we start at the highest level of `head`. We scan linearly along the list at the current level i, until we are about to jump to an node whose key value is strictly greater

than to x. Since `tail` is associated with ∞, we will always succeed in finding such a node. Let p point to the node just before this step. If p's data value is equal to x then we stop. Otherwise, if we are not yet at the lowest level, we descend to the next lower level $i-1$ and continue the search there. Finally, if we are at the lowest level and have not found x, we announce that the x is not in the list (see Fig. 2).

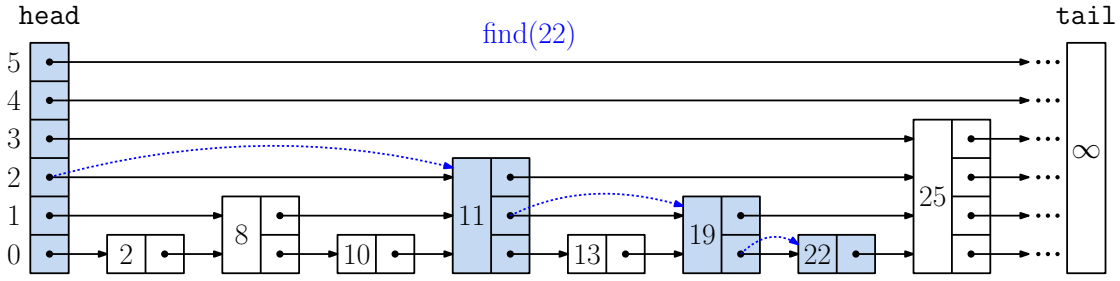

Fig. 2: Searching the ideal skip list.

How long would this search require in the worst case? Observe that we need never traverse more than one link at any given level in the path to the desired node. We will generally need to access two nodes at each level, however, because the need to determine the node whose key is greater than x's. As mentioned earlier, the number of levels is $\lceil \lg n \rceil$. Therefore, the total number of nodes accessed is at most $2\lceil \lg n \rceil = O(\log n)$.

Randomized Skip Lists: Unfortunately, like a perfectly balanced binary tree, the ideal skip list is too pure to be able to use for a dynamic data structure. As soon as a single node was added to the middle of the lists, all the heights following it would need to be modified. But we can relax this requirement to achieve an efficient data structure. In the ideal skip list, every other node from level i is extended up to level $i+1$. Instead, how about if we did this *randomly*?

Suppose that we have built the skip list up to some level i, and we want to extend this to level $i+1$. Imagine a node at level i tossing a coin. If the coin comes up heads (with probability $1/2$) this node promotes itself to level $i+1$, and otherwise it stops here. By the nature of randomization, the expected number of nodes at level $i+1$ will be half the number of nodes at level i. Thus, the expected number of nodes at level k will be $n/2^k$, which means that the expected number of nodes at level $\lceil \lg n \rceil$ is a constant. Fig. 3 shows what such a *randomized skip list*, or simply *skip list*, will look like.

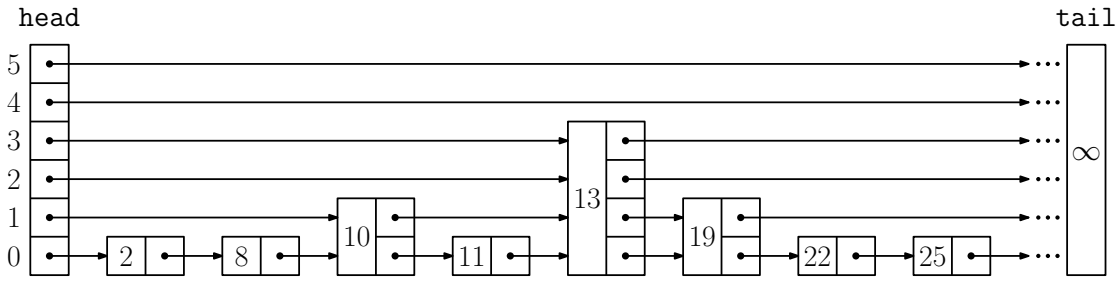

Fig. 3: A (randomized) skip list.

Implementation: As we saw with 2-3 and 2-3-4 trees, skip lists have variable sized nodes. In the skip list, there is no a priori limit on a node's size. This is not an issue in languages like Java,

where it is easy to dynamically allocate arrays of arbitrary length, however. For example, our skip list node might have the following structure

Node in a Skip List

```
class SkipNode {
    Key key
    Value value
    SkipNode[] next

    SkipNode(Key x, Value v, int size) {
        key = x; value = v; next = new SkipNode[size];
    }
}
```

Here is sample code to find a key in a Skip list. Notice it is very simple (and does not involve recursion!) Intuitively, we start at the topmost nonempty level of the skip list, and keep moving along a given level as long as the key in the next node is less than or equal to the key we are looking form. Whenever we get stuck, we drop down a level. When we drop out of the skip list (level is less than zero), we know that the current node's key is at most x and the next node's key is strictly greater than x. So, we check this node's key matches, and if so, we have found it. If so, we drop down a level. We stop when we arrive at level zero. If the key is in the skip list, this node will match. If not, we know that the key is not present.

Find in a Skip List

```
Value find(Key x) {
    int i = topmostLevel                        // start at topmost nonempty level
    SkipNode p = head                           // start at head node
    while (i >= 0) {                            // while levels remain
        if (p.next[i].key <= x) p = p.next[i]   // advance along same level
        else i--                                // drop down a level
    }
    return (p.key == x ? p.value : null)        // return value if found
}
```

Height and Space Analysis: Unlike binary search trees whose nodes are all of the same size, the nodes of a skip list have variable sizes. How many levels will the skip list have? The process we described earlier could create an arbitrarily large number of levels (if your coin keeps coming up heads). We can show that the maximum level in the skip list will be $O(\log n)$ with high probability.

Theorem: The expected number of levels in a skip list with n entries is $O(\log n)$.

Proof: We will show the stronger result that, with high probability, the number of levels in a skip list with n entries does not exceed $O(\log n)$. Let $\ell \geq 1$ denote some level number. Suppose that we have n entries and we observe that our skip list contains at least one node at level ℓ. In order for an arbitrary entry to make it to level ℓ, its coin toss came up heads at least ℓ times consecutively. The probability of this happening is clearly at most $1/2^\ell$. Since this holds independently for all of the n entries, the probability that

any of the entries survive to level ℓ is[1] at most $n/2^\ell$.

Setting $\ell = c \cdot \lg n$, it follows that the probability that the maximum height exceeds ℓ is at most

$$\frac{n}{2^\ell} \;=\; \frac{n}{2^{c \lg n}} \;=\; \frac{n}{(2^{\lg n})^c} \;=\; \frac{n}{n^c} \;=\; n^{1-c}.$$

So, if we take even a moderate value of c, say $c = 3$, the probability that the maximum height exceeds $3 \lg n$ is at most $1/n^2$. So, if n is large (say over 1000), the probability of exceeding $3 \lg n$ is less than one in a million!

This implies that, if you have an upper bound on the value of n, you are safe (with high probability) to allocate head and tail arrays of size $\lceil 3 \lg n \rceil$. Of course, even if a node were to attempt to exceed this level, you could just arbitrarily limit it. The analysis that follows would not change as a result.

Next, let's consider the storage space. Under the assumption that the maximum number of levels is $O(\log n)$, then in the worst case every node contributes to every level, and the skip list would have total storage of $O(n \log n)$. This is too high! The following lemma indicates that the expected space used is just linear in n.

Theorem: The expected storage for a skip list with n entries is $O(n)$.

Proof: For $i \geq 0$, let n_i denote the number of nodes that contribute to level i of the skip list. This is a *random variable*, meaning that its value varies with our random choices. The quantity that we are interested in is its expected value $E(n_i)$. The probability that an arbitrary entry contributes to level i, is just the probability of tossing i consecutive heads using a fair coin, which is clearly $1/2^i$. Thus, the expected number of entries that survive to this level is just the total population size (n) times the probability of contributing ($1/2^i$). Thus, $E(n_i) = n/2^i$.

By basic facts on the geometric series, we know that $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2$, and hence

$$\sum_{i=0}^{\infty} E(n_i) \;=\; \sum_{i=0}^{\infty} \frac{n}{2^i} \;=\; n \sum_{i=0}^{\infty} \frac{1}{2^i} \;=\; 2n.$$

Finally, by the linearity of expectation (the expected value of a sum of random variables is the sum of the expected values of the random variables), it follows that the total expected storage needed is the sum of $E(n_i)$ over all possible levels

$$E\left(\sum_{i=0}^{\infty} n_i\right) \;=\; \sum_{i=0}^{\infty} E(n_i) \;=\; 2n \;=\; O(n),$$

and this completes the proof.

Notice that the proof did not make use of the fact that we arbitrarily limit the maximum level. Clearly, adding such a upper limit can only reduce the storage requirements.

Search-Time Analysis: Earlier, we argued that the worst-case search time in an ideal skip list is $O(\log n)$. Now, we will show that the *expected case* search time in the randomized skip list will be $O(\log n)$. It is important to note that the analysis to follow will *not* depend on the

[1] This is called the *union bound*. It states that the probability that any among n events occurs is at most n times the maximum probability of each individual event.

choice of keys in the data structure nor the order in which they were inserted. Rather, it will depend solely on the randomized (coin-flipping) process used to build the data structure.

The analysis of skip lists is an example of a *probabilistic analysis*. As observed earlier, the expected number of levels in a skip list is $O(\log n)$. We will show that for any fixed node, the length of the search path leading here is $O(\log n)$ in expectation. Our analysis will be based on walking backwards along the search path. (This is sometimes called a *backwards analysis*.)

Observe that the forward search path drops down a level whenever the next link would have taken us "beyond" the node we are searching for. Thus, when we consider the reversed search path, it will always take a step up if it can (i.e., if the node it is visiting contributes to the next higher level), otherwise it will take a step to the left.

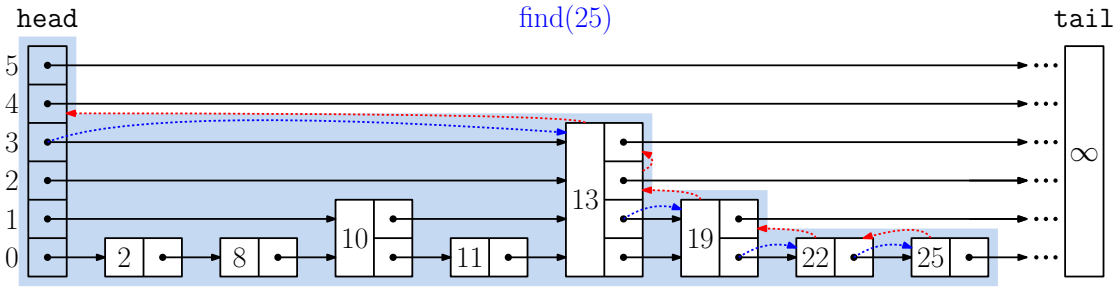

Fig. 4: The search path (blue) to $x = 25$ and the reverse search path (red).

Theorem: The expected number of nodes visited in a search in a skip list of n keys is $O(\log n)$.

Proof: We will prove this for the more general case, where the probability that a node is promoted to the next higher level is p, for some constant $0 < p < 1$. The analysis for our coin-flipping version of the skip follows by setting $p = 1/2$.

For $0 \leq i \leq O(\log n)$, let $E(i)$ denote the expected number of nodes visited in the skip list at the *top* i levels of the skip list. (For example, in Fig. 4, the skip list's top level is 5. In this case $E(2)$ would be the expected number of steps taken at levels 4 and 5, $E(3)$ would be the expected time spent in levels $3--5$, and $E(6)$ would be the expected number of steps at all the levels.)

As mentioned above, our analysis is based on walking backwards along the search path. Whenever we arrive at some node of level i, the probability that it contributes to the next higher level is exactly p, and if so, the search path came from above. On the other hand, with the remaining probability $1 - p$, this node stopped at this level, and the backwards search stays at the same level. Counting the current node we are visiting $(+1)$, we can express $E(i)$ by the recurrence:

$$E(i) \;=\; 1 + p \cdot E(i-1) + (1-p)E(i).$$

This is a weird recurrence, because we are defining $E(i)$ in terms of itself! But this is not circular, since each subsequent invocation occurs with a lower probability value, and so this converges. In particular, with a bit of algebra, we have:

$$E(i) \;=\; \frac{1}{p} + E(i-1).$$

By expansion, it is easy to verify that $E(i) = \frac{i}{p}$. Thus, if our skip list has ℓ levels, the expected search time is $E(\ell) = \ell/p$. By our assumption that p is a constant, we have $E(\ell) = O(\ell)$. By our earlier theorem on the maximum number of levels, we know that $\ell = O(\log n)$. Therefore, the expected search time for skip lists is $O(\log n)$.

Clearly, there is more math involved in establishing expected-case bounds for randomized data structures. (This is the bane of students of data structures!) The payoff is that these data structures are usually faster than traditional structures because they are so simple. There is no need for rotating nodes, merging or splitting nodes, skewing and splitting, as we have seen with tree-based data structures.

Insertion and Deletion: Insertion into a skip list is almost as easy as insertion into a standard linked list. Given a key x to insert, we first do a search on key x to find its immediate predecessors in the skip list at each level of the structure. Next, we create a new node x. To determine the height of this node, we toss a coin repeatedly until it comes up tails. (More practically, we generate a random number until its parity is odd.) Letting k denote the number of tosses needed, we create a node who height is the minimum of $k + 1$ and the maximum height of the skip list. We then link this node in to its $k + 1$ lowest predecessors in the list (see Fig. 5).

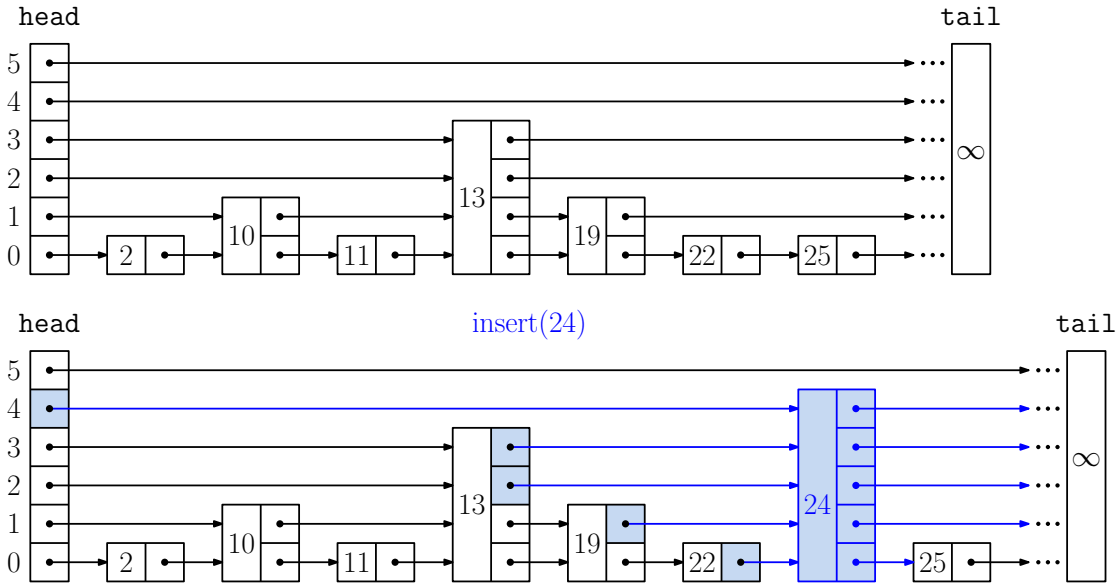

Fig. 5: Inserting a new key 24.

Deletion is quite similar. Again, we search for the node containing the key to delete, and we keep track of all its predecessors at various levels in the skip list. On finding it we unlink the node from each level, exactly as we would do in a standard linked-list deletion. Both operations take $O(\log n)$ time in expectation.

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 when performing searching.

Skip-list nodes have variable size, which is a bit unusual. This is not a problem in programming languages like Java that allow us to dynamically allocated arrays of variable size. Thus, each node of the skip list will generally contain the key-value pair associated with this entry, a variable-sized array of `next` pointers (so that `p.next[i]` points to the next node in the skip list from node `p` at level i). Finally, the structure has two special "sentinel nodes, " `head` and `tail`. We assume that `tail.key` is set to some incredibly large value so that searches always stop here.

Overall Performance: From a practical perspective, skip lists can do pretty much everything that standard binary trees structures can do. In expectation, they require $O(n)$ storage space, and all dictionary operations can be performed in time $O(\log n)$ in expectation. Given their simple linear structure, they are arguably easier to visualize and program. Experimental studies show that skip lists are among the fastest data structures for sorted dictionaries (with treaps). This is largely because the power of randomization keeps us from having to maintain more complex balance information, and thus simplifies the code and processing.