CMSC 420: Lecture 7
Randomized Search Structures: Treaps and Skip Lists

Randomized Data Structures: A common design techlque in the field of algorithm design involves the notion of using randomization. A *randomized algorithm* employs a pseudo-random number generator to inform some of its decisions. Randomization has proved to be a remarkably useful technique, and randomized algorithms are often the fastest and simplest algorithms for a given application.

This may seem perplexing at first. Shouldn't an intelligent, clever algorithm designer be able to make better decisions than a simple random number generator? The issue is that a deterministic decision-making process may be susceptible to systematic biases, which in turn can result in unbalanced data structures. Randomness creates a layer of "independence," which can alleviate these systematic biases.

In this lecture, we will consider two famous randomized data structures, which were invented at nearly the same time. The first is a randomized version of a binary tree, called a *treap*. This data structure's name is a portmanteau (combination) of "tree" and "heap." It was developed by Raimund Seidel and Cecilia Aragon in 1989. (Remarkably, this 1-dimensional data structure is closely related to two 2-dimensional data structures, the *Cartesian tree* by Jean Vuillemin and the *priority search tree* of Edward McCreight, both discovered in 1980.)

The other data structure is the *skip list*, which is a randomized version of a linked list where links can point to entries that are separated by a significant distance. This was invented by Bill Pugh (a professor at UMD!).

Because the treaps and skiplists are randomized data structure, running times depend on the random choices made by the algorithm. We will see that all the standard dictionary operations take $O(\log n)$ *expected time*. The expectation is taken over all possible random choices that the algorithm may make. You might protest, since this allows for rare instances where the performance is very bad. While this is always a possibility, a more refined analysis shows that (assuming n is fairly large) the probability of poor performance is so insanely small that it is not worth worrying about.

Treaps: The intuition behind the treap is easy to understand. Recall back when we discussed standard (unbalanced) binary search trees that if keys are inserted in *random order*, the expected height of the tree is $O(\log n)$. The problem is that your user may not be so accommodating to insert keys in this order. A treap is a binary search tree whose structure arises "as if" the keys had been inserted in random order.

Let's recall how standard binary tree insertion works. When a new key is inserted into such a tree, it is inserted at the leaf level. If we were to label each node with a "timestamp" indicating its insertion time, as we follow any path from the root to a leaf, the timestamp values must increase monotonically (see Fig. 1(b)). From your earlier courses you should know a data structure that has this very property—such a tree is generally called *heap*.

This suggests the following simple idea: When first inserted, each key is assigned a *random priority*, call it `p.priority`. As in a standard binary tree, keys are sorted according to an inorder traversal. But, the priorities are maintained according to heap order. Since the priorities are random, it follows that the tree's structure is consistent with a tree resulting from a sequence of random insertions. Thus, we have the following:

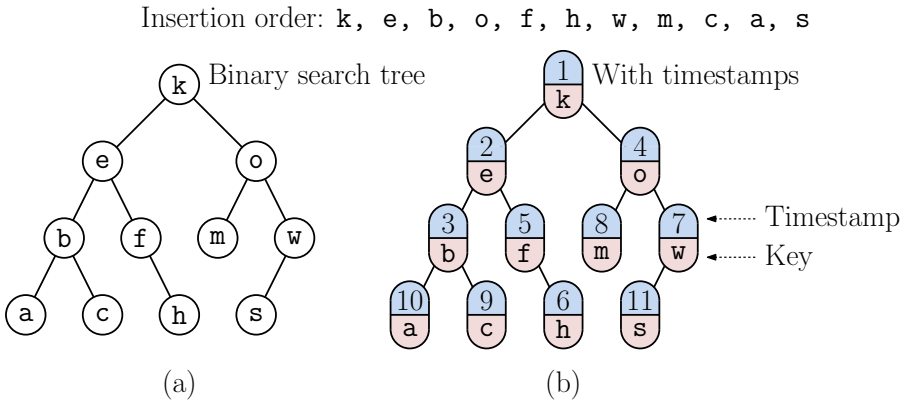

Fig. 1: (a) A binary search tree and (b) associating insertion timestamps with each node.

Theorem: A treap storing n nodes has height $O(\log n)$ in expectation (over all $n!$ possible orderings of the random priorities present in the tree).

Since priorities are random, you might wonder about possibility of two priorities being equal. This might happen, but if the domain of random numbers if much larger than n (say at least n^2) then these events will be sufficiently rare that they cannot significantly affect the tree's structure. The next question is whether we can maintain this structure efficiently. The answer is "yes", and it is remarkably easy.

Treap Insertion: Insertion into the treap is remarkably simple. First, we apply the standard binary-search-tree insertion procedure. When we "fall out" of the tree, we create a new node p, and set its priority, `p.priority`, to a random integer. We then walk retrace the path back up to the root (as we return from the recursive calls). Whenever we come to a node p whose child's priority is smaller than p's, we apply an appropriate rotation (depending on which child it is), thus reversing their parent-child relationship. We continue doing this until the newly inserted key node is lifted up to its proper position in heap order. The code is so simple, that we will leave as an exercise.

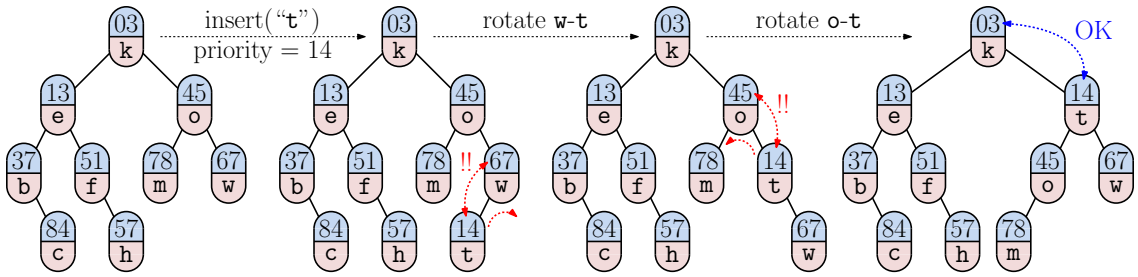

Fig. 2: Treap insertion.

Treap Deletion: Deletion is also quite easy, but as usual it is a bit more involved than insertion. If the deleted node is a leaf or has a single child, then we can remove it in the same manner that we did for binary trees, since the removal of the node preserves the heap order. However, if the node has two children, then normally we would have to find the replacement node, say its inorder successor and copy its contents to this node. The newly copied node will then be out of priority order, and rotations will be needed to restore it to its proper heap order.

There is, however, a cute trick for performing deletions. We first locate the node in the tree and then set its priority to ∞ (see Fig. 3). We then apply rotations to sift it down the tree to the leaf level, where we can easily unlink it from the tree.

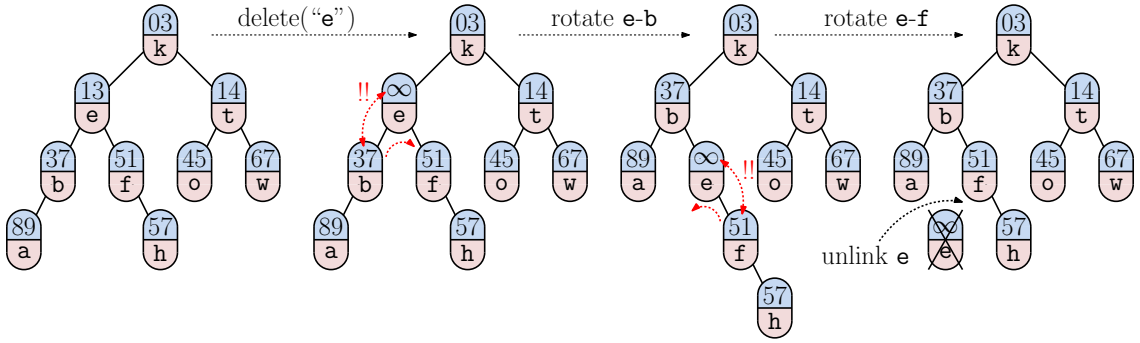

Fig. 3: Treap deletion.

The treap is particularly easy to implement because we never have to worry about adjusting the priority field. For this reason, treaps are among the fastest data tree-based dictionary structures.

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. 4). 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 an *"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`.

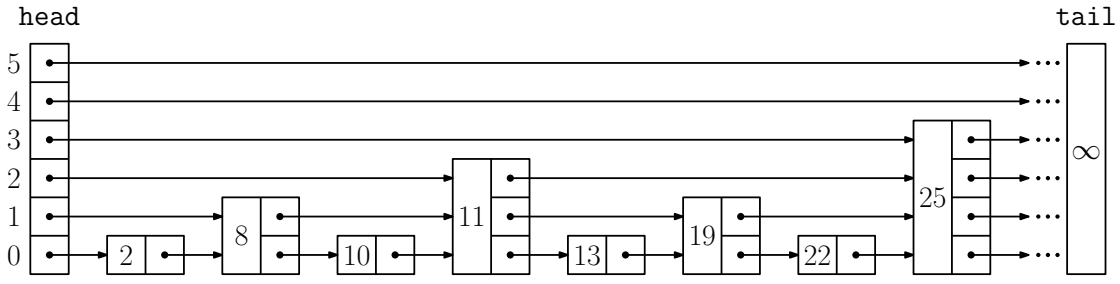

Fig. 4: The "ideal" skip list.

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. 5).

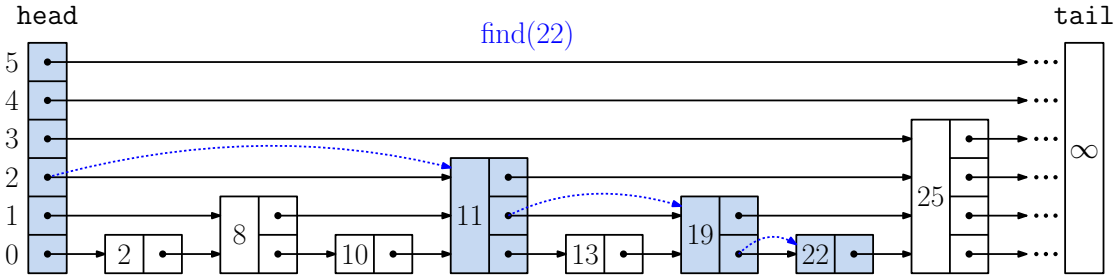

Fig. 5: Searching the ideal skip list.

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 $O(\log n)$.

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 of 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. 6 shows what such a *randomized skip list*, or simply *skip list*, will look like.

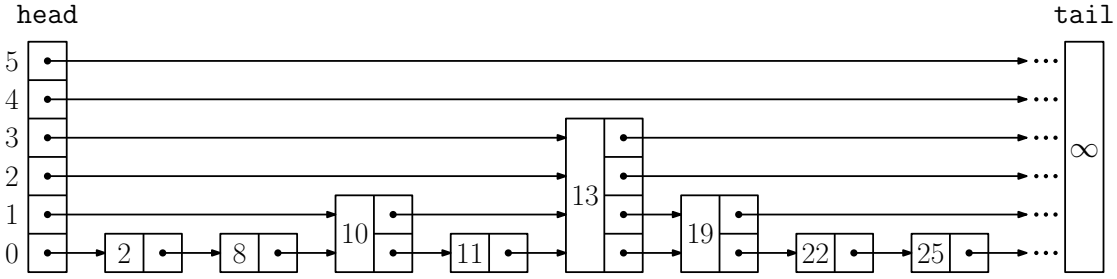

Fig. 6: A (randomized) skip list.

Space Analysis: Unlike binary search trees whose nodes are all of the same size, the nodes of a skip list have variable sizes. If we assume that the maximum number of levels of a skip is

$O(\log n)$, then in the worst case every node contributes to every level, and the skip list would have total storage of $O(n \log n)$. In the best case (from the perspective of storage), every node contributes only to the lowest level, and the total storage would be $O(n)$. Note that either of these cases is *extremely* unlikely.

To describe the expected case, observe that in expectation, exactly half of the nodes from one level are promoted to the next. Thus in expectation, all n nodes contribute to level 0, $n/2$ contribute to level 1, $n/4$ contribute to level 2, and generally $n/2^i$ contribute to level i. In summary in expectation, the total storage (for pointers) is:

$$\sum_{i=0}^{h-1} \frac{n}{2^i} \;=\; n \sum_{i=0}^{h-1} \frac{1}{2^i} \;=\; n\left(2 - \frac{1}{2^i}\right) \;\leq\; n\left(2 - \frac{1}{2^m}\right) \;\leq\; 2n.$$

(Here we have made use of the formula for the geometric series, $\sum_{i=0}^{m-1}(\frac{1}{2})^i = 2 - (\frac{1}{2})^m$.) Thus, the expected storage just for the pointers is $O(n)$. Storing the keys themselves takes just $O(n)$ storage. Finally, the head and tail nodes take $O(\log n)$ storage each, but $\log n$ is dominated by n asymptotically, so we can ignore their contribution.

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 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.

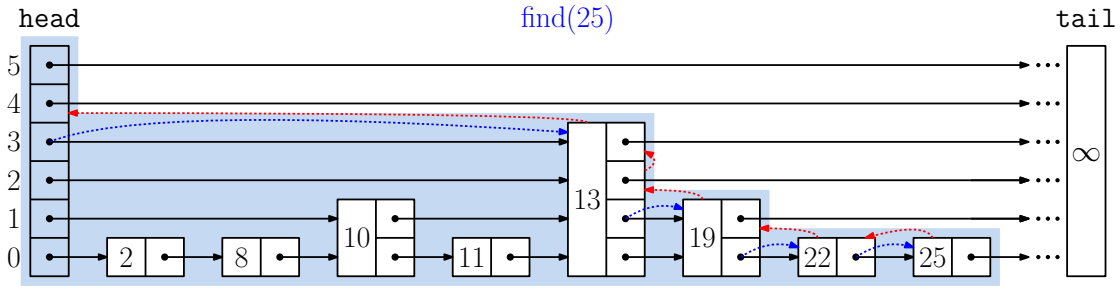

Fig. 7: The search path (blue) to $x = 25$ and the reverse search path (red).

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. 7, 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, and $E(6)$ would be the expected number of steps at all the levels.) Whenever we arrive at some node of level i, the probability that it contributes to the next higher level is exactly p. With the remaining probability $1-p$ we stay at the same level. Counting the current node we are visiting (+1), we can express $E(i)$ by the recurrence:

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

With a bit of algebra, we have:

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

By expansion, it is easy to verify that $E(i) = \frac{i}{p}$. Since $i \leq O(\log n)$, and by our assumption that p is a constant, it follows that the expected search time is $O(\log n)$.

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. 8).

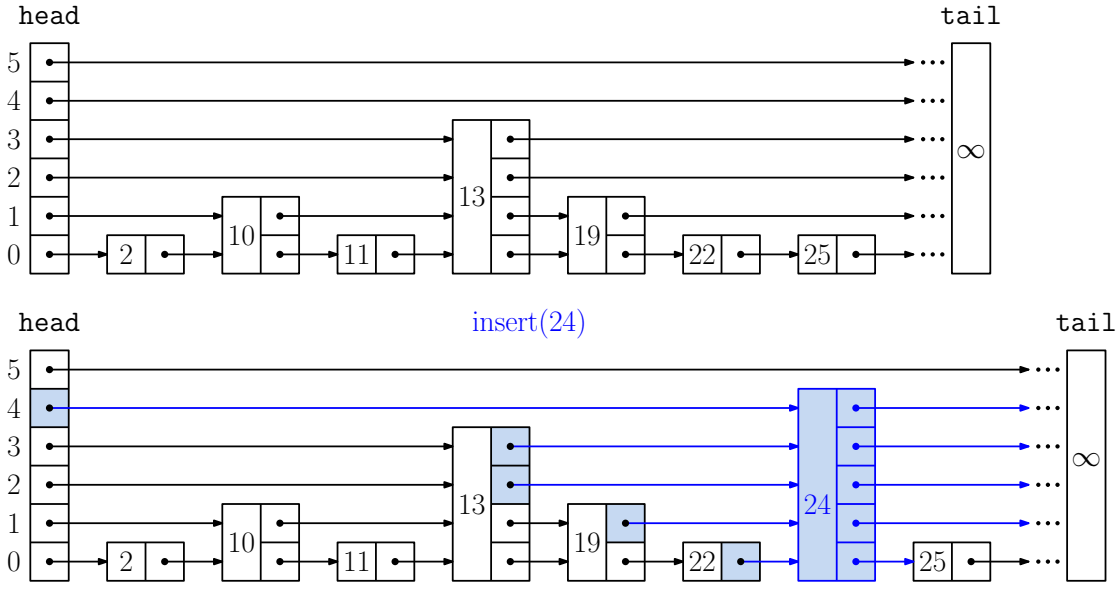

Fig. 8: Inserting a new key 24.

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.