Lecture 8

Balanced search trees

8.1 Overview

In this lecture we discuss search trees as a method for storing data in a way that supports fast insert, lookup, and delete operations. (Data structures handling these operations are often called a *dictionary* data structures.) The key issue with search trees is that you want them to be *balanced* so that lookups can be performed quickly, and yet you don't want to require them to be perfect because that would be too expensive to maintain when a new element is inserted or deleted. In this lecture, we discuss *B-trees* and *treaps*, which are two methods that handle this tradeoff so that all desired operations can be performed in time $O(\log n)$.

Topics covered in this lecture:

- Simple binary search trees: like an on-the-fly version of quicksort.
- B-trees: a form of balanced search tree that uses flexibility in its node degrees to efficiently keep the tree balanced.
- Treaps: like an on-the-fly version of *randomized* quicksort, that uses randomization to keep the tree balanced with high probability.
- Tree-rotations: an important concept when talking about binary search trees, that is used inside many binary search tree data structures (including treaps).

8.2 Introduction

For the next few lectures we will be looking at several important data-structures. A data-structure is a method for storing data so that operations you care about can be performed quickly. Data structures are typically used as part of some larger algorithm or system, and good data structures are often crucial when especially fast performance is needed.

We will be focusing in particular on what are called *dictionary* data structures, that support `insert` and `lookup` operations (and usually `delete` as well). Specificially,

Definition 8.1 *A* **dictionary data structure** *is a data structure supporting the following operations:*

- **insert(key, object)**: *insert the (key, object) pair. For instance, this could be a word and its definition, a name and phone number, etc. The key is what will be used to access the object.*
- **lookup(key)**: *return the associated object.*
- **delete(key)**: *remove the key and its object from the data structure. We may or may not care about this operation.*

For example, perhaps we are the phone company and we have a database of people and their phone numbers (plus addresses, billing information and so on). As new people come in, we'd like to be able to insert them into our database. And, given a name, we'd like to be able to quickly find their associated information.

One option is we could use a sorted array. Then, a lookup takes $O(\log n)$ time using binary search. However, an insert may take $\Omega(n)$ time in the worst case because we have to shift everything to the right in order to make room for the new key. Another option might be an unsorted list. In that case, inserting can be done in $O(1)$ time, but a lookup may take $\Omega(n)$. In the last lecture we saw a data structure that consisted of an unsorted *set* of sorted arrays, where insert took $O(\log n)$ amortized time and lookup took time $O(\log^2 n)$. Today we will look at search tree methods that allow us to perform both operation in time $O(\log n)$.

A binary search tree is a binary tree in which each node stores a (key, object) pair such that all descendants to the left have smaller keys and all descendants to the right have larger keys (let's not worry about the case of multiple equal keys). To do a lookup operation you simply walk down from the root, going left or right depending on whether the query is smaller or larger than the key in the current node, until you get to the correct key or walk off the tree. We will also talk about non-binary search trees that potentially have more than one key in each node, and nodes may have more than two children.

For the rest of this discussion, we will ignore the "object" part of things. We will just worry about the keys since that is all that matters as far as understanding the data structures is concerned.

8.3 Simple binary search trees

The simplest way to maintain a binary search tree is to implement the insert operations as follows.

insert(x): If the tree is empty then put x in the root. Otherwise, compare it to the root: if x is smaller then recursively insert on the left; otherwise recursively insert on the right.

Equivalently: walk down the tree as if doing a lookup, and then insert x into a new leaf at the end.

Example: build a tree by inserting the sequence: `C A R N E G I E'` (where `E'` > `E`).

Plusses and minuses: On the positive side, this is very easy to implement (though deletes are slightly painful — think about how you might handle them). On the negative side, this has very bad worst-case behavior. In fact, it behaves exactly like quicksort using the leftmost element as the pivot, and the search tree is the same as the quicksort recursion tree. In particular, if elements are in sorted order, this will produce a very unbalanced tree where all operations take time $\Omega(n)$.

Today we will examine two ways to fix this problem, *B-trees* and *treaps*. B-trees are a particularly nice method used in database applications, and treaps are a lot like *randomized* quicksort, but trickier since the keys are coming in one at a time.

An important idea: the problem with the basic binary search tree was that we were not maintaining balance. On the other hand, if we try to maintain a perfectly balanced tree, we will spend too much time rearranging things. So, we want to be balanced but also give ourselves some slack. It's a bit like how in the median-finding algorithm, we gave ourselves slack by allowing the pivot to be "near" the middle. For B-trees, we will make the tree perfectly balanced, but give ourselves slack by allowing some nodes to have more children than others.

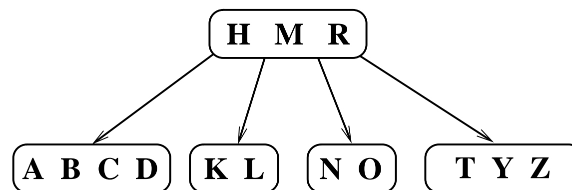
8.4 B-trees and 2-3-4 trees

A **B-tree** is a search tree where for some pre-specified $t \geq 2$ (think of $t = 2$ or $t = 3$):

- Each node has between $t-1$ and $2t-1$ keys in it (except the root has between 1 and $2t-1$ keys). Keys in a node are stored in a sorted array.
- Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range $[t, 2t]$ except the root has degree in the range $[2, 2t]$. The semantics are that the ith child has items between the $(i-1)$st and ith keys. E.g., if the keys are $[a_1, a_2, \ldots, a_{10}]$ then there is one child for keys less than a_1, one child for keys between a_1 and a_2, and so on, until the rightmost child has keys greater than a_{10}.
- All leaves are at the same depth.

The idea is that by using flexibility in the sizes and degrees of nodes, we will be able to keep trees perfectly balanced (in the sense of all leaves being at the same level) while still being able to do inserts cheaply. Note that the case of $t = 2$ is called a **2-3-4 tree** since degrees are 2, 3, or 4.

Example: here is a tree for $t = 3$ (so, non-leaves have between 3 and 6 children—though the root can have fewer—and the maximum size of any node is 5).



Now, the rules for lookup and insert turn out to be pretty easy:

Lookup: Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse on that child.

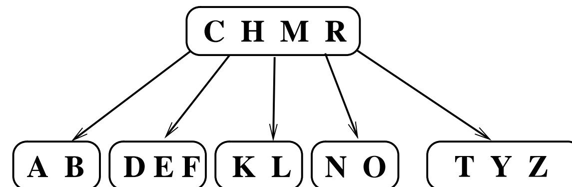
Insert: To insert, walk down the tree as if you are doing a lookup, but if you ever encounter a *full* node (a node with the maximum $2t-1$ keys in it), perform a **split** operation on it (described below) before continuing.

Finally, insert the new key into the leaf reached.

Split: To split a node, pull the median of its keys up to its parent and then split the remaining $2t-2$ keys into two nodes of $t-1$ keys each (one with the elements less than the median and

one with the elements greater than the median). Then connect these nodes to their parent in the appropriate way (one as the child to the left of the median and one as the child to its right). If the node being split is the root, then create a fresh new root node to put the median in.

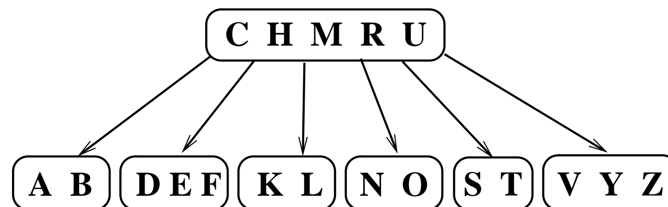
Let's consider the example above. If we insert an "**E**" then that will go into the leftmost leaf, making it full. If we now insert an "**F**", then in the process of walking down the tree we will split the full node, bringing the "**C**" up to the root. So, after inserting the "**F**" we will now have:



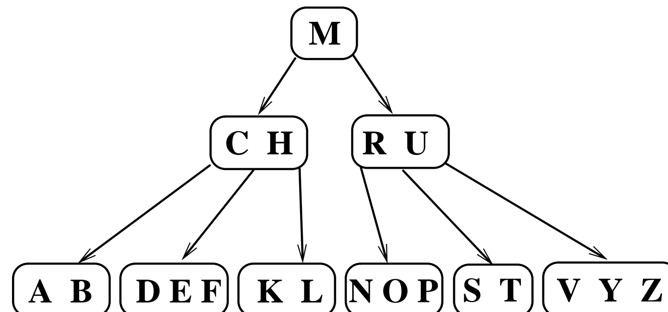
Question: We know that performing a split maintains the requirement of at least $t-1$ keys per non-root node (because we split at the median) but is it possible for a split to make the parent *over-full*?

Answer: No, since if the parent was full we would have already split it on the way down.

Let's now continue the above example, inserting "**S**", "**U**", "**V**":



Now, suppose we insert "**P**". Doing this will bring "**M**" up to a new root, and then we finally insert "**P**" in the appropriate leaf node:



Question: is the tree always height-balanced (all leaves at the same depth)?

Answer: yes, since we only grow the tree *up*.

So, we have maintained our desired properties. What about running time? To perform a lookup, we perform binary search in each node we pass through, so the total time for a lookup is $O(\text{depth} \times \log t)$. What is the depth of the tree? Since at each level we have a branching factor of at least t (except possibly at the root), the depth is $O(\log_t n)$. Combining these together, we see that the "t" cancels out in the expression for lookup time:

$$\text{Time for lookup } = O(\log_t n \times \log t) = O(\log n).$$

Inserts are similar to lookups except for two issues. First, we may need to split nodes on the way down, and secondly we need to insert the element into the leaf. So, we have:

$$\text{Time for insert } = \text{ lookup-time } + \text{ splitting-time } + \text{ time-to-insert-into-leaf.}$$

The time to insert into a leaf is $O(t)$. The splitting time is $O(t)$ per split, which could happen at each step on the way down. So, if t is a *constant*, then we still get total time $O(\log n)$.

What if we don't want to think of t as a constant, though. The interesting thing is that even if t is large, amortized analysis comes to our rescue. In particular, if we create a tree from n inserts, we can have made at most $O(n/t)$ splits *total* in the process. **Why?** Because each split creates a new node, and there are $O(n/t)$ nodes total. So the *total* time spent on splits over all n inserts is $O(n)$, which means that we are only spending $O(1)$ time on average on splits per insert. So, the amortized time per insert is:

$$O(\log n) + O(1) + O(t) = O(\log n + t).$$

More facts about B-trees:

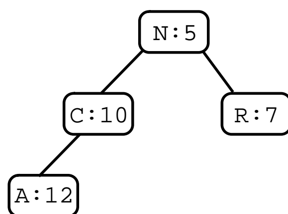
- B-trees are used a lot in databases applications because they fit in nicely with the memory heirarchy when you use a large value of t. For instance, if you have 1 billion items and use $t = 1,000$, then you can probably keep the top two levels in fast memory and only make one disk access at the bottom level. The savings in disk accesses more than makes up for the additive $O(t)$ cost for the insert.

- If you use $t = 2$, you have what is known as a 2-3-4 tree. What is special about 2-3-4 trees is that they can be implemented efficiently as binary trees using an idea called "red-black-trees". We will not discuss these in detail, but they use the same notion of tree rotation as treaps (which are discussed below).

8.5 Treaps

Going back to binary search trees, we saw how a standard BST is like quicksort using the leftmost element as a pivot, with all of its worst-case problems. A natural question is: can we come up with a method that is instead like *randomized* quicksort? The problem is that we don't have all the elements at the start, so it's not so obvious (we can't just say "let the root be some element we are *going* to see in the future"). However, it turns out we *can* come up with an analog to randomized quicksort, and the data structure based on this is called a "treap".

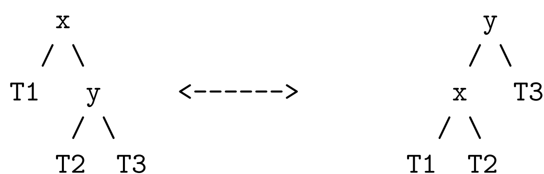
The idea for a treap is that when an element x is inserted, we also give it a random *priority* value. Think of the priorities as giving the order in which they are supposed to be chosen as pivots. (Also, think of priorities as real numbers so we don't get any ties). In particular, the property we will require is that if v is a child of u, then priority$(v) >$ priority(u). For example:



So, the keys are *search-tree ordered* and the priorities are *heap ordered*, which is why this is called a *treap*!

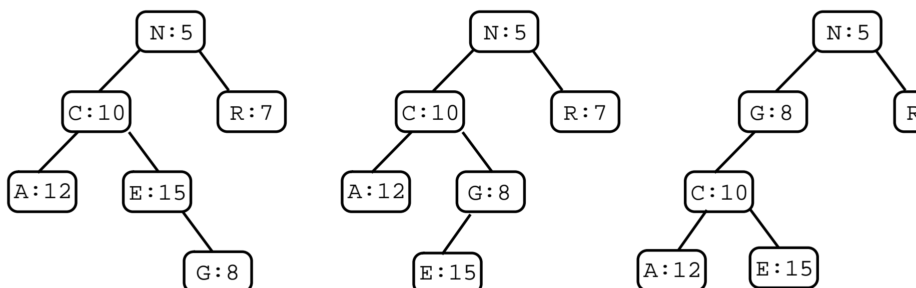
Question: Why must such a thing even exist? Given a set of (key, priority) pairs, how do we know it is even *possible* to design a tree so that the keys are in search-tree order and the priorities are in heap order? **Answer:** just sort by priority and run the standard BST algorithm. Moreover, notice that if we choose priorities at random, the tree is exactly the same as the recursion tree of a randomized quicksort that chooses the pivots in the same random order as the priorities.

The big question now is: how can we perform inserts to maintain the treap property? It turns out it is not too difficult. To insert a new element into a treap, just do the usual binary search tree insert (walking down and inserting at a leaf) and then *rotate* the new item up the tree so long as its parent has a larger priority value. A *tree rotation* is the following operation (which can be done in either direction) that maintains search tree order:



Here, T_1, T_2, T_3 represent subtrees. This rotation is legal (maintains search tree order) because both trees correspond to the statement $T_1 < x < T_2 < y < T_3$. In the above picture, in the left-to-right direction, we will call this "rotating y above x" (or "rotating x above y" in the right-to-left direction).

Let's do an example of inserting into a treap. Suppose we are inserting the letters `C A R N E G I E'`, and so far we have the tree with 4 letters above. If we now insert `E` with priority 15 (so no rotations) and then `G` with priority 8, we would do:



We now need to prove this maintains the treap property. First, the search-tree property on keys is maintained since that is not affected by rotations. We can analyze the heap property as follows. Initially, all descendant relations are satisfied (if y is descendant of x then priority(y) > priority(x)) *except* for case that y is the new node. Now, suppose the new node y does violate the heap property. Then it must do so with its parent x, and we will do a rotation. Without loss of generality, assume

it is left-to-right in the generic picture above. Notice now that the only new descendant relation we add is that x and T_1 become descendants of y. But since priority(x) > priority(y), and priority(T_1) > priority(x) by assumption, these are all satisfied. So, we maintain our invariant. Finally when new node y has priority greater than its parent, all descendant relations are satisfied and we are done.

So, insert can be done in time proportional to the search time, since at worst the number of rotations equals the number of steps on the way down. (One can actually furthermore show that the *expected* number of rotations per insert is $O(1)$.)

Depth analysis. Inserts and searches both take time proportional to the depth of the tree, so all that remains is to analyze depth. When we analyzed randomized quicksort, we showed that in expectation, the total number of comparisons is $O(n \log n)$. This means that in expectation, the *sum* of all node depths is $O(n \log n)$, or equivalently, in expectation, the *average* node depth is $O(\log n)$. However, we can actually show something a lot stronger: in fact, with high probability, the *maximum* node depth is $O(\log n)$. (This also implies that the quicksort $O(n \log n)$ bound holds with high probability.) Here is a sketch of one way to prove it:

Proof: let's go back to our "dart-throwing" argument for quicksort. Let's line up the elements in sorted order, and pick one element X that we care about. We can think about the depth of this node as follows: we throw a dart at random into this sorted list, representing whatever element is at the root of the tree, and whereever it lands, it cuts the list and the part that X is not on disappears. We do this again, representing the next node on the path to X, and keep going until only X is left. Now, if you think about it, whether the dart lands to the left or right of X, it has a 50% chance of deleting at least half of that side of the list. This can happen at most $\lg n$ times to the left of X, and at most $\lg n$ times to the right.

So, we can think of it like this: each dart/pivot is like a coin toss with a 50% chance of heads. Each heads cuts off at least half of that side of the list. We have to stop sometime before getting $2 \lg n$ heads. There's a nice bound called "Hoeffding's inequality" that says that if you flip a coin t times, the chance of getting less than $t/4$ heads is at most $e^{-t/8}$. So, if we flip $8 \lg n$ times, the chance of getting at most $2 \lg n$ heads is at most $e^{-\lg n} = e^{-(\ln n)/(\ln 2)} = 1/n^{1.44}$. Even if you multiply this by n to account for the fact that we want this to be true for *every* node X, you still get that it's unlikely *any* element has depth more than $8 \lg n$.[1]

[1] Hoeffding bounds also say that the chance you get fewer than $t/8$ heads in t flips is at most $e^{-9t/32}$. So in $16 \lg n$ flips, the chance of failure is at most $n^{-6.49}$. This means the chance that *any* X has depth greater than $16 \lg n$ is at most $1/n^{5.49}$.