601.433/633 Introduction to Algorithms **Lecturer:** Michael Dinitz
Topic: Balanced Search Trees **Date:** 9/16/21

6.1 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. Today we will be focusing in particular on what are called dictionary data structures, that support insert and lookup operations (and often delete as well). Specifically,

Definition 6.1.1 *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.*

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 $\Omega(n)$ time. Our goal is going to be to construct data structures which let us perform all operations in $O(\log n)$ time.

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.

6.2 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: `H O P K I N S`.

On the positive side, this is very easy to implement (although deletes are a bit of a pain – think of how you would implement them). However, the worst-case performance is very bad. If the tree has depth d, then an insert or a lookup could take $\Omega(d)$ time. So if the tree is very unbalanced, operations could take $\Omega(n)$ time each! This is actually similar to what happens with quicksort – if the input is already sorted, then each recursive call only decreases the input size (quicksort) or depth of tree (BSTs) by one.

Main 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.

6.3 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 $t = 3$) the following three properties hold.

1. 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.

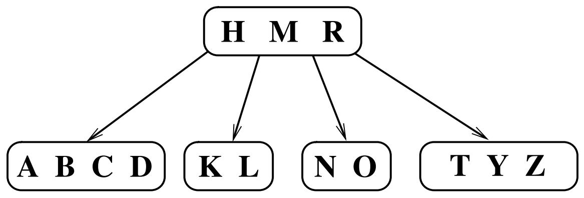
2. Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range $[t, 2t]$ except the root has degree in the range $[2, 2t]$. If v is a node with keys $[a_1, a_2, \ldots, a_k]$ and the children are $[v_1, v_2, \ldots, v_{k+1}]$, then the tree rooted at v_i contains only keys that are at least a_{i-1} and at most a_i (except the the edge cases: the tree rooted at v_1 has keys less than a_1, and the tree rooted at v_{k+1} has keys at least a_k).

   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}.

3. 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. The special case of $t = 2$ is called a *2-3-4 tree* since degrees are 2, 3, or 4.

As an 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).



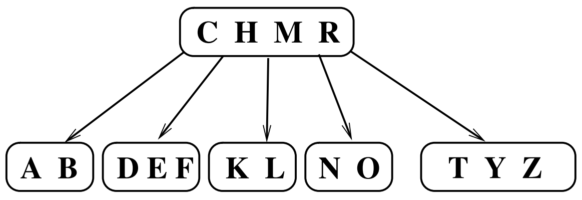
Now we have to define how to do inserts and lookups.

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 immediately (described below) before continuing. When you're at a leaf node, insert the new element into the array at that node.

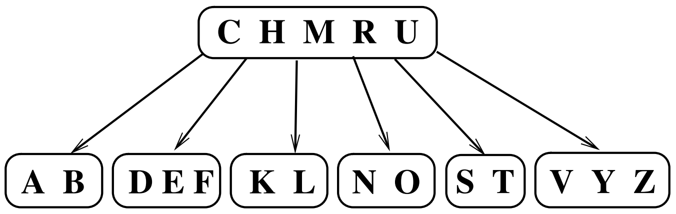
Split: To split a node, pull the median of its keys up to its parent and then split the remaining $2t-2$ keys into two nodes of $t-1$ keys each (one with the elements less than the median and one with the elements greater than the median). Then connect these nodes to their parent in the appropriate way (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:

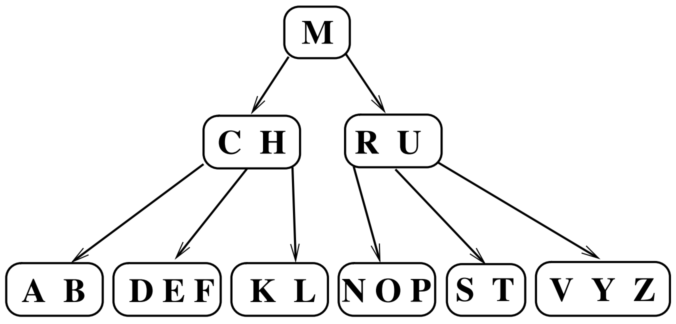


Sanity check: We know that performing a split maintains the requirement of at least $t-1$ keys per non-root node (because we split at the median) but is it possible for a split to make the parent over-full? No, since if the parent were already full then we would have split it on the way down.

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:



Let's prove (somewhat informally) the correctness of our insert procedure. In particular, let's prove that all nodes store the right number of keys (between $t-1$ and $2t-1$ for a non-root, and at most $2t-1$ for the root), that all elements in the subtree between two adjacent keys are indeed at least the smaller and at most the larger, and that all leaves are the same depth.

Theorem 6.3.1 *Our insert algorithm maintains all three properties of a B-tree.*

Proof: We will prove this by induction. Note that if we insert into an empty B-tree we get simply one node with one element in it. All three properties of a B-tree are indeed true.

Now suppose that we have a B-tree which satisfies all three properties, and insert an element x. We claim that the resulting tree still satisfied all three properties.

The third property is obvious since the tree grows "up": let h be the height of the tree before the insert, which by induction is the depth of all the leaves. If our insert does not split the root then the distance between the root and all leaves is unchanged (and so all of them are still h), while if we do split the root then all leaves are at distance $h+1$ from the new root.

For the first property, note that whenever we do a split operation, the parent gains one new key while each child has exactly $t-1$ keys in it. Since when we do an insert we split any node on the path which has $2t-1$ keys in it, whenever we do a split operation on a node, its parent must have strictly less than $2t-1$ keys (or else the parent would have already been split). Thus none of our split operations violate the degree bounds. And when we do finally insert the new key, the node we insert into must have strictly less than $2t-1$ keys before the insert (or else we would split it). Hence when we do an insert we never violate any of the degree upper bounds. Similarly, we never violate a degree lower bound since we never create a node with less than $t-1$ keys other than the root (which is allowed to have fewer keys).

So it remains only to prove that for every node, the tree rooted at the ith child of the node contains keys which are between the $(i-1)$ and i'th key in the node. Whenever we do a split operation we preserve this property, so it is preserved as we move down the tree on our insert, and in the end the new key that we insert is placed in a leaf node so that this property continues to hold. Hence this property (and this all three properties) continues to hold after inserting a new key. ∎

So, we have maintained our desired properties. What about running time? Suppose that the depth is d. To perform a lookup, we perform binary search in each node we pass through, so the total time for a lookup is $O(d \times \log t)$. 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:

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

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, for a total of $O(t \log n)$. So, if t is a constant, then we still get total time $O(\log n)$. Note that this motivates us to make t small, which is one reason that 2-3-4 trees get their own name.

More facts about B-trees:

- B-trees are used a lot in databases applications because they fit in nicely with the memory hierarchy 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 extra $O(t)$ cost for the insert. From a student four years ago: "I was personally asked in interviews why B-trees are well suited for extremely large data sets."
- 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", which is what we'll discuss next.

6.4 Red-Black Trees

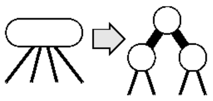
B-trees are great, but it's generally easier to use binary trees. Lookups are simpler, people find them easier to understand, and for general-purpose (non-database) applications, they tend to be a bit faster and a bit more flexible. There have been a whole bunch of balanced binary search trees developed – you may have seen AVL trees and/or Treaps when you took Data Structures

(601.226). One of the most popular and widely used are *red-black trees* – they're the default in the linux kernel, are used to optimize Java HashMaps, and are generally the balanced search tree of choice in practice. I don't have time to go over them in full detail, but you should read the book. What I'll try to explain here is how we can view them as essentially binary versions of 2-3-4 trees.

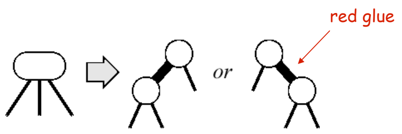
One quick note: unlike B-trees, for red-black trees (and binary trees in general) we'll use the convention that all missing pointers actually point to NULL. In other words, keys are only stored at internal nodes: all leafs are just NULL. This will make it a bit easier to analyze

So let's start from a 2-3-4 tree. How can we turn it into a binary tree and still have all of the nice properties of a 2-3-4 tree? Unfortunately, we can't – we're not going to be able to get *exact* height-balance on any binary tree. But recall that we don't need exact balance in order to have efficient lookups; we just need the depth to be $O(\log n)$. So let's be a bit flexible in the balance requirement.

Maybe we get lucky, and all the nodes of our 2-3-4 are actually degree-2 (they have one key stored at them). Then we don't need to do anything. But what if we have some nodes with 2 or 3 keys stored at them? Let's start with the case of 3 keys stored, since that's simpler: let's just transform the single degree-4 node into three degree-2 nodes!



So now we have two kinds of edges: some which correspond to the original edges in the 2-3-4 tree (we all these *black edges*), and some which are "internal" in the larger nodes in the 2-3-4 tree (we call these *red edges*). When we represent a node with 2 keys stored (degree 3), we actually have two choices:



So suppose that we start with a 2-3-4 tree, and turn it into a binary tree using the above correspondences. What important properties do we have that show this is a good binary tree?

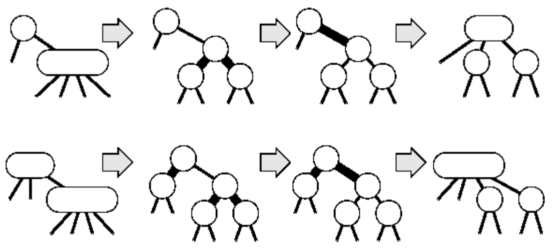
1. In every path from the root to any other node, every red edge is followed by a black edge (i.e., two red edges never appear in a row).

2. For each node, all paths to its descendent leaves have the same number of black edges. When this node is the root r, this value is called the *black-depth.*

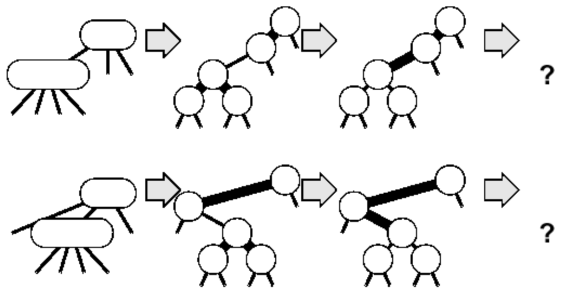
Note that these are both simple consequences of the properties of 2-3-4 trees we've talked about and the correspondences we chose. Together, they imply that the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf. This in turn implies that the depth of the tree is only $O(\log n)$, so lookups will be efficient (see Lemma 13.1 in the book).

So we just need to figure out a way of inserting which preserves this correspondence (or more importantly, preserves the two important properties). This turns out to be a little complicated, and is the main reason that red-black trees can seem a bit mysterious. Note that I'm going to do this slightly differently than how it's done in the book, in order to make the correspondence to 2-3-4 trees a bit clearer. I strongly suggest you also read the book on this. Informally, we're going to split full nodes on the way down (like in B-trees), while the book (and most real implementations) first insert and then "repair" by moving up the tree.

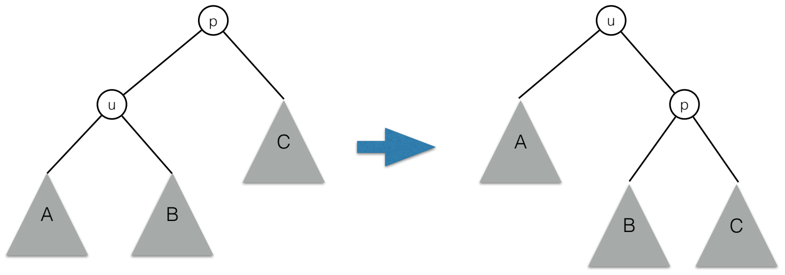
So suppose we're trying to insert. The basic idea is to do a normal binary tree insert, where the new link is red (since it's like inserting into a node at the bottom of the 2-3-4 tree). If this were a 2-3-4 tree, if we saw a full node on our way down when inserting, we would split it. What does that look like in red-black trees? Some cases are easy: we can just switch colors (bold is red).



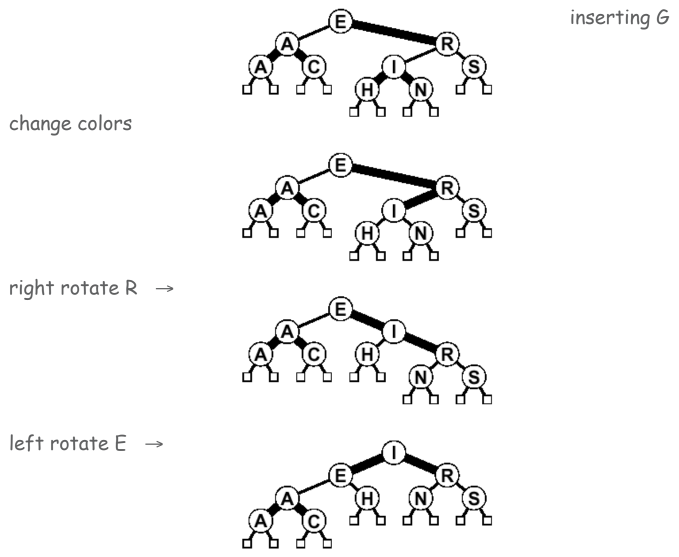
But other cases are harder – switching colors will violate one of the red-black requirements:



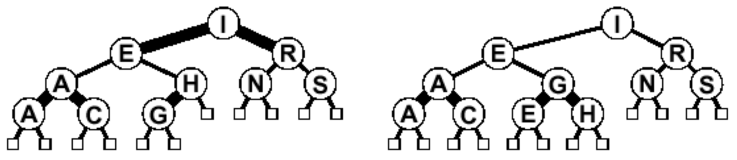
In order to handle these cases, we need to use the idea of *rotations*. Tree rotations are a way of changing binary search tree while still preserving the search tree property. They're useful all over the place, and are used in search tree data structures beyond just red-black trees. The basic idea is simple, as the following picture shows.



It turns out we can solve the annoying cases of red-black trees by using either one (the first case) or two (the second case) rotations. Let's see this through an example (proofs are in the book – read them!)



We also need to handle some other complications – even if we don't see any full nodes on our way down, inserting the new node might mess up the correspondence to 2-3-4 trees and the red-black properties. These can also be handled with appropriate tree rotations. For example, suppose we insert F (E in these notes since I've copied them):



There are a few other complications, but this should illustrate the basic idea. A red-black tree is a particular binary tree representation of a 2-3-4 tree, and by carefully designing our insert and delete algorithm to respect this representation, we can get a binary search tree with $O(\log n)$ for lookups, inserts, and deletes. As mentioned, the book does this slightly differently, which obscures the relationship to 2-3-4 trees (although it's still true) but makes red-black trees themselves quite a bit simpler. Please make sure you understand what's going on in the book.