CS104: Data Structures and Object-Oriented Design (Fall 2013)
November 19, 2013: Deletions in 2-3 Trees and Red-Black Trees
Scribes: CS 104 Teaching Team

Lecture Summary

In this lecture, we reviewed in more detail how to delete a key from a 2-3 tree. Next, we briefly talked about 2-3-4 trees, saw the definition of Red-Black Trees, and how they can be seen as a way to implement 2-3-4 trees as binary trees. Finally, we got started on Hashtables. The latter part is missing from these notes, as we followed the separate handout on Hashtables quite closely — look there for the second half of this lecture and the next few lectures.

1 Deletions in 2-3 Trees

To delete a key (and its value) from a 2-3 tree, we of course first have to find it. Once we have found the key, deleting it from an internal node would be really complicated. Therefore, in 2-3 trees (and pretty much all search trees), we always have a first step of ensuring that we only delete keys from leaf nodes.

The way to do this is to search for the next-larger (or possibly next-smaller, but for concreteness, we'll stick with next-larger) key. This is accomplished by first going down the edge immediately to the right of the key to delete. (That is, the middle edge if the key was the left key, or the right edge if the key was the right key.) Then, we always go down the leftmost edge of all nodes, until we hit a leaf. At that point, the leftmost key in that leaf is the successor.

Once we have found the successor, we swap it with the key we wanted to delete. This is safe to do, as no other element could now be out of place — everything in the right subtrees is larger than the successor. Now, the key we want to delete is in a leaf, and we can start from there.

The deletion algorithm will start with deleting a key at a leaf, but then possibly propagate necessary "fixes" up the tree. By far the easiest case is if the leaf contained two keys. Then, we simply delete the key, and we are done.

Otherwise, we now have a leaf with no keys in it. We can't just delete the leaf, as this would violate the 2-3 tree property of all leaves being at the same level. So we need to find ways to fix the tree. The operations we perform will be the same at the leaf level as at the higher levels, so we'll phrase the problem case more generally.

At the point when we need to fix something, we have an internal node with 0 keys and one subtree hanging off. In the base case, this subtree is just `NULL`, but as we propagate up (recursively or iteratively), there will be other subtrees. Thus, the problem situation looks as shown in Figure 1. (The black node has no keys, and one subtree below it, which is empty for a leaf.)

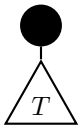

Figure 1: The problem case in deletion. The black node has no keys.

In order to deal with this problem case, we distinguish a few cases, based on how many neighbors the empty node has, and how many keys they in turn have.

1. If the empty node has a sibling with two keys, then the node can borrow a key from a sibling. A typical version of this case would look as shown in Figure 2:

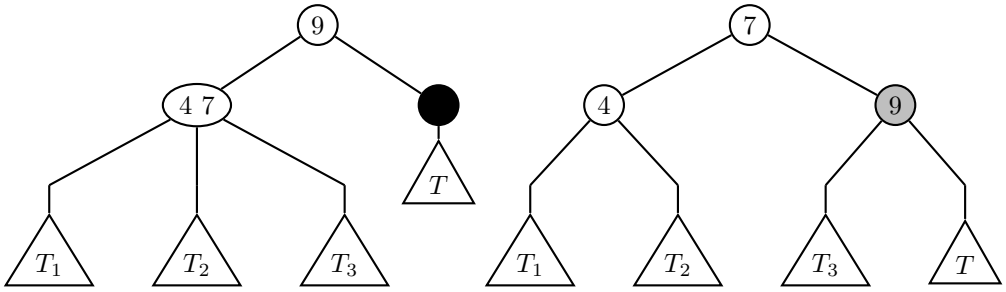

Figure 2: Borrowing from a sibling. The empty node borrows from its sibling (via its parent) and is shown in gray after.

It works similarly if the empty node has two siblings. When the immediately adjacent sibling has two keys, we can do the same thing (by pushing one of the sibling's keys to the parent, and the parent's key down to the empty node). When it is the sibling two hops away that has two keys, we push one key up, a key from the parent down to the middle child, and the key from the middle child up to the parent.

2. If no sibling of the empty node has two keys, then the next case is that the empty node has two siblings (i.e., the parent has two keys). In that case, the empty node can borrow a key from the parent, with help from the siblings. A typical case is shown in Figure 3.

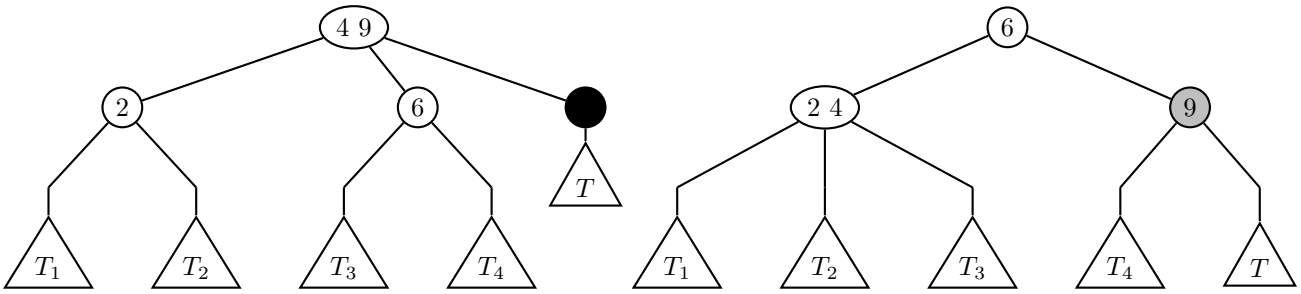

Figure 3: Borrowing from the parent. The empty node borrows from its parent and is shown in gray after.

This works pretty much the same way if instead of the right child, it is the middle or left child of the parent that was empty.

3. The final remaining case is when the empty node has only one sibling, and that sibling contains only two keys. In that case, we cannot fix the problem immediately. Instead, we merge the parent and the sibling into one node with two keys, and push up the empty node one level, where we will try to solve it instead. Thus, this is the one case where we need recursion (or an iterative approach), while all the others solve the problem right away. This case is shown in Figure 4.

4. There is one final case. If the empty node is the root now, then it can be safely deleted. Its one child (the root of T) becomes the new root of the tree, which is now one level smaller.

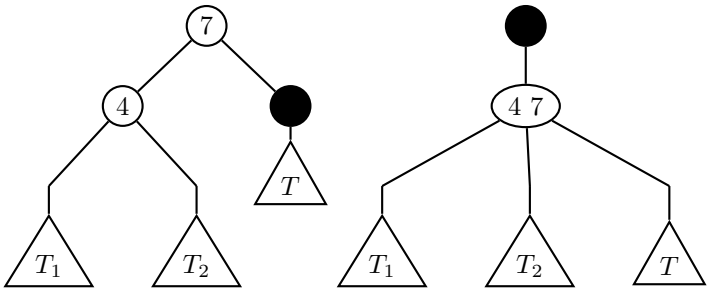

Figure 4: Merging two nodes and recursing. The sibling and parent are merged, and the empty node is pushed up one level, where it still needs to be dealt with.

This covers all the cases; each has a few subcases, depending on which of the children is empty, but they all follow the same pattern.

Notice that these operations carefully maintain all the properties: the leaves stay at the same level, each node has one or two children once we are done fixing problems, and the tree is still a 2-3 search tree.

When we look at the running time, we see that rearranging the tree is only moving a constant number of pointers and keys (and values) between nodes. Thus, all of the operations above take time $O(1)$. In cases 1, 2, 4, we are thus done in $O(1)$. In case 3, we may continue iteratively or recursively, but we perform a rearrangement operation at most once per level of the tree. Since we already worked out that a 2-3 Tree has at most $\log_2(n)$ levels, the running time is $O(\log n)$ for each of the operations. Thus, in summary, we get that using a 2-3 tree, all operations take time $\Theta(\log n)$. (It is not hard to construct trees on which you do need to propagate up all levels for an insertion or a removal.)

For many data structures, we say that which one to choose depends on the context. It is generally agreed that search trees (and Hashtables, which we will get to next) are simply better implementations of Dictionaries than those based on arrays, vectors, or lists. If you need an associate structure, don't use a linear structure — use a tree or Hashtable.

2 2-3-4 Trees

As the name suggests, 2-3-4 Trees are quite similar to 2-3 Trees; the difference is that each node can now have 2, 3, or 4 children. Correspondingly, the node has 1, 2, or 3 keys. The textbook discusses 2-3-4 Trees in more detail; but for the most part, the operations look almost exactly like for 2-3 Trees. There are just a few more cases to consider. 2-3-4 Trees do have some slight advantages, in that one can implement insertion a bit faster. (Basically, the idea is that as you search down the tree, when you see a node with 3 keys already, it looks like a potential for needing to split a node. So you just split it prophylactically right away, and when you get to the leaves, you can insert a key without having to propagate any insertions back up to the root.) The slight downside of 2-3-4 Trees is with implementing them: there are now more cases to consider for borrowing from siblings or parents, and the code would become a little longer.

In principle, you could implement B Trees with any combinations of lower and upper bounds on the number of keys per node. 2-3 Trees are just the simplest version that works, and probably the most efficient (along with 2-3-4 trees). But others are possible. The main reason to go to very different numbers such as trees with hundreds or more keys per node would be when they are stored on disk instead of in main memory. Then, you don't mind doing a linear search in memory through 100 keys in one node, but going to disk for the next node may take much longer. The textbook discusses data structures for external memory (hard disk) in more detail, but we won't go into it more in this class.

The actual main reason to introduce 2-3-4 Trees here is to set the stage for Red-Black Trees.

3 Red-Black Trees

2-3 trees and 2-3-4 trees are perfectly fine data structures. But some people prefer their trees to be binary, and there are some classic binary search tree data structures that any computer science student should have learned about. The two most common ones are AVL Trees and Red-Black Trees. In this class, we will see Red-Black Trees.

As we discussed when we got started on search trees, the important thing is to keep the height bounded by $O(\log n)$, so that search (and insertion and deletion) is fast. In order to do that, you want the search trees to be "balanced," in the sense that all the subtrees of a given node are "similar" in height or number of nodes. The goal of constructions such as Red-Black Trees or AVL Trees is to impose certain rules on the trees that can be maintained easily (in time $O(\log n)$ per insertion or deletion) and ensure that the height does not grow too fast. For AVL trees, these rules involve comparing the numbers of nodes in the two subtrees, and making sure they do not get too different. For Red-Black Trees, these are rules for coloring nodes, which will have a similar effect.

There are two ways to understand Red-Black Trees: one is to present the coloring rules, which seem pretty arbitrary. The other is to realize that Red-Black Trees are exactly a way to "squeeze" 2-3-4 Trees (or 2-3 Trees) into binary trees. We will start with the second way of thinking about them, as it is more intuitive. We will then see how the strange-looking rules for Red-Black Trees are simply the consequence of this encoding.

In order to turn a 2-3-4 Tree into a binary tree, we need to figure out what to do when a node has 3 or 4 children, since that's not allowed in binary trees. The solution is quite simple: we replace the one node with a little subtree of 2 or 3 nodes. This is illustrated in Figure 5.

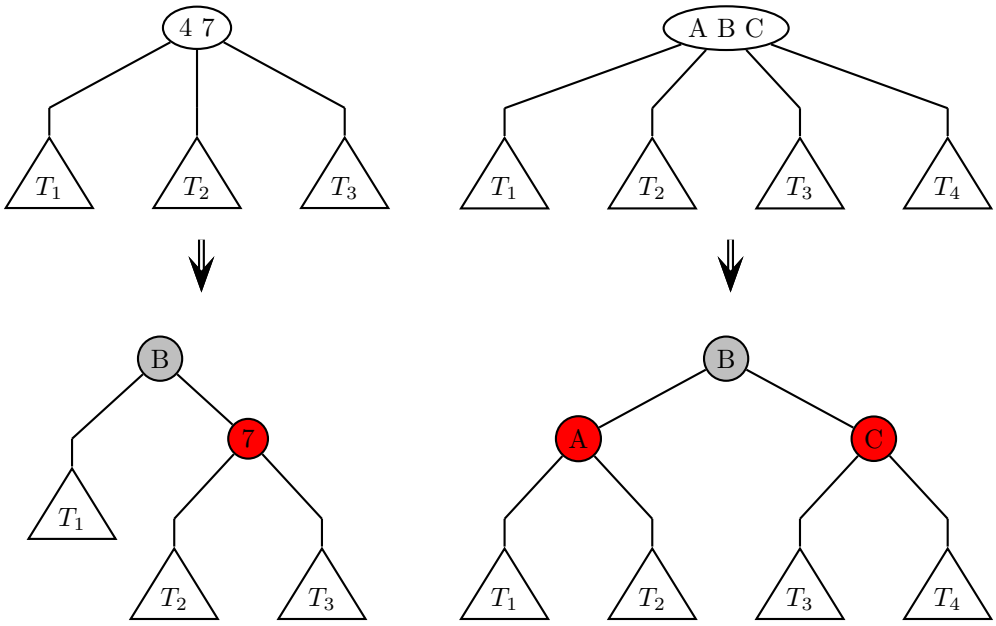

Figure 5: Turning 2-3-4 Trees into binary trees. The figure shows nodes with 3 and 4 children, and the corresponding little binary trees we build.

We can see that as a result of this operation, each level is split into at most two new levels, so the height of the tree at most doubles. In particular, it stays at $O(\log n)$. Also, notice that as a result, not all leaf nodes will now be at the same level, but they will be roughly at the same level (within a factor 2 of each other).

In the figure, we have also already added color coding corresponding to the way Red-Black Trees can be defined. When we split a node into 2 or 3 nodes as we did, we will color the top level *black* (in the figure, gray), and the lower level (1 or 2 nodes) *red*.

One way of defining Red-Black Trees is exactly as the trees you can get out of 2-3-4 Trees by performing the operations we just described. This is the motivation that your textbook also gives. The other definition is less intuitive, but used frequently, and you should be aware of it. It lists the properties that one can derive from the definition we just gave.

Definition 1 *A* Red-Black Tree *is a binary search tree with the following additional properties:*

- *Each node is either red or black*
- *A red node can only have black children. (However, a black node can have any colors among its children.)*
- *For any path from the root to a leaf, the number of black nodes on the path must be the same.*

An implication of these properties is that every Red-Black Tree has height at most $2\log_2(n)$.

To search for a key, you can just ignore the colors. They matter only when inserting or removing items, as that is when the tree is changing, and thus needs to be kept balanced.

To insert a key, as always, you search for it (unsuccessfully) in the tree. Then, it is inserted as a the only content of a red leaf. This may violate the rule that a red node can only have black children (but does not increase the number of black nodes on any root-leaf path). In order to fix this violation, we can then rotate pieces of the tree and recolor nodes up the tree. It is not all that intuitive to write down all these operations (you can look them up in a number of textbooks or other material), but perhaps the easiest way to think about it is to ask "What would a 2-3-4 Tree do?" That is, you reinterpret your Red-Black Tree as a 2-3-4 Tree, look at the correct operation, and then "translate" that operation into the Red-Black tree.

The same holds true for removal. As always, you search for the key, and swap the element with the successor in a leaf if necessary. Afterwards, there are a number of cases of rotations and recoloring, again corresponding to all cases of the 2-3-4 Tree.