

Rather than covering Red-Black trees or AVL trees, I have covered 2-3 trees instead. I find them to be simpler (easier to remember), and they easily lead to B-trees (both for teaching and historically). Because they are not covered in the book, I am writing up the version we covered in class here. You may find that other write-ups of 2-3 trees differ in minor ways. I have not viewed the original (1970) presentation of 2-3 trees, but expect that all variations are still quite similar.

In a BINARY SEARCH TREE, each node is associated with a single key value. The node has two children, because it breaks a range into two subranges: those with smaller values, and those with larger values. It will be convenient for us to assume that all values in the tree are distinct, but for non-distinct values, if you want to allow for repetition of keys within the tree, I generally consider the values to go “to the right” when they have a choice, that is, for key  $x$ , values  $v < x$  go to the left subtree, while values  $v \geq x$  go to the right. For actual duplicate values, perhaps you need to be careful about searching, especially when the value duplicates enough to fill multiple sub-trees.

In a standard BINARY SEARCH TREE, a value may be stored in an internal node of the tree, or at a leaf node. You should all be familiar with this, as I will start from here. In a BINARY SEARCH TREE, these guide values are just single key values: if the key is  $x$ , and you are searching for  $v$ , you go to the left child of  $x$  if  $v < x$ , and the right child if  $v \geq x$ .

**2-3 Trees:** Binary trees seem natural because for key  $x$ , value  $v$  can be  $v < x$  or  $v \geq x$ . However, if an internal node has 2 key values,  $x$  and  $y$  such that  $x < y$ , it is natural for the node to have 3 children: the leftmost child has values  $v < x$ , the middle child has values  $x \leq v < y$ , and the rightmost child has values  $v \geq y$ . A 2-3 tree will allow each internal node to have 2 or 3 children, by allowing the internal nodes to each have 1 or 2 keys respectively. Note, no internal node ever has just one child, and a node with 1 (2) key values has exactly 2 (3) children.

Again, we can consider variants where full records can be stored either throughout the tree, or variants where they are stored just at the leaf nodes. In 2-3 trees, all leaves will be at the same level, and the number of children of various nodes are adjusted to ensure that.

**Search:** To search for a record with key value  $v$  is very similar to searching for the a value in a BINARY SEARCH TREE. The only difference is that instead of having a single comparison at a node to determine which of 2 children to recursively search, up to 2 comparisons may be needed to determine which of 3 children to recursively search, when a node contains two key values instead of just one.

**Insertion:** As in BINARY SEARCH TREES, to insert a new record, first search for that value to find where it belongs in the tree. Here, we assume that the leaf node has enough room for 2 records (keys), so if the leaf node where the value would reside has only one record, insertion is simple. Just insert this value into the same leaf node.

When the leaf already has 2 key values, it doesn't have room for the new value. Consider that the two values, plus the one to be inserted, are  $x, y, z$ , such that  $x \leq y \leq z$ . (We call the median value  $y$ , regardless of whether it is being inserted or already in the tree. Again, the new value should go “to the right” if there are repeat values, so if a new is being inserted which has a value equal to the larger of the two keys already in the tree, I would call the new key  $z$  above.)

The node has no room for 3 values, so we use  $y$  to separate  $x$  and  $z$ .  $y$  “gets promoted”, at which point it is recursively inserted into its parent node, which may also split if there are already two keys there. If the node has no parent, we create a new parent node, which will be the new root of the tree. (The root grows up here, leaving leaves at the same level. This is the only time that the height of the tree grows.) The root's two children will be  $x$  and  $z$ . (For non-leaf nodes, we will also need to set  $x$  and  $z$ 's children correctly, as seen in class.)

The height of the tree is logarithmic: if every internal node had 2 children, a tree with  $n$  nodes would have approximately  $\lg n$  levels. If every internal node had 3 children, it would have approximately  $\log_3 n$  levels.

**2-3-4 Trees:** You can guess what 2-3-4 trees are going to be: here, nodes are allowed 1-3 key values, and will have 2-4 children each. When we are adding a 4th value to a node, and thus need to split it, it makes sense to promote the median of the 3 values *not* being inserted, just for symmetry's sake. Also notice: during the search procedure, you need at most 2 comparisons to compare to 3 key values: first compare to the median value, and then compare to the smallest/largest value depending on the outcome of that first comparison.

**Top-Down Insertion:** A different implementation of 2-3-4 trees allows you to split nodes with 4 children before they are forced to split, and will not need any recursive calls to traverse back up the tree. Upon traversing your way down a tree, automatically split any vertices with 4 children (3 keys), sending the middle key up one level. The parent cannot already have 3 keys, because it would have been split in the previous step if it did, and thus when you

promote a value, there will always be room in its parent to take that value without needing to recurse farther up the tree.

Note, 2-3-4 trees are covered in the B-tree chapter, as they are just the special case of B-trees of minimum degree 2. The deletion algorithm in the book differs somewhat from the one I covered in the evening section of Wednesday, but I will cover this more again on Monday to make our procedure match the one of the book.

As for B-trees, there are even more variants, such as B+ trees and B\* trees. For these two variants, the biggest differences with B-trees (respectively) have to do with (a) whether records are stored within regular internal nodes, compared to just storing key values in internal nodes and keeping all records in the leaf nodes of the tree (key values in internal nodes then solely guide you to the correct leaf node), and (b) how large the minimum sized node is, compared to the maximum sized node. In any case, should you refer to B-trees via a source other than our textbook, be cautious, in that terminology may differ. Some books will describe B+ trees, but call them B-trees. Other variants in terminology have to do with the order of the tree. In our book,  $t$  describes the minimum degree of an internal (non-root) node. In other books, the order of the tree may refer to the maximum degree of the node, and odd values may be possible. (In our text, the maximum degree  $2t$  is always even.) Details on how nodes split may vary. While not claiming that our text is the one-and-only way, or even that it is strictly better than other presentations, we will normalize to the presentation in our text when discussing B-trees.

The differences I will consider with B-trees compared to those in the book: the book assumes linear search within each node, while I am assuming binary search. For actual external memory nodes, the runtime difference will not be large: the time needed to read the data from disk will still be significantly longer than the time to run linear search on it. For 2-3-4 trees, however, we might get a slight speed-up by comparing to the median node of 3 first.