CMSC 420: Lecture 6
2-3 Trees

2-3 Trees: In the previous lecture, we presented one way to establish balance in a binary search tree, namely through the AVL tree's height-balance condition. Today, we will explore an alternative approach which is achieved by allowing nodes to variable "widths." In particular, we will allow internal nodes to have either two or three children (see Fig. 1(a) and (b)). When a node has three children, it stores two keys. Given the two key values b and d, the three subtrees A, C, and E must satisfy the *generalized inorder property* that

$$A < b < C < d < E,$$

These are called *2-nodes* and *3-nodes*, respectively.

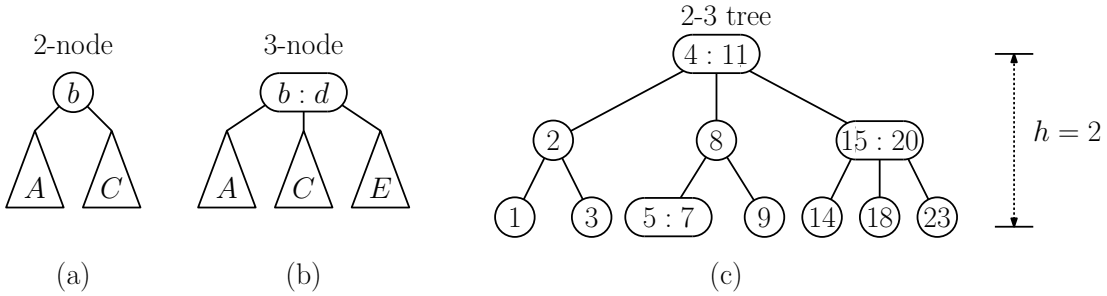

Fig. 1: (a) 2-node, (b) 3-node, and (c) a possible 2-3 tree of height 2.

A *2-3 tree* of height $h \geq -1$ is defined recursively as follows (see Fig. 1(c)). It is either:

- empty (i.e., `null`) when $h = -1$, or
- its root is a 2-node, and its two subtrees are each 2-3 trees of heights $h - 1$, or
- its root is a 3-node, and its three subtrees are each 2-3 trees of heights $h - 1$.

Since all the leaves are at the same level, and the sparsest possible 2-3 tree is a complete binary tree, which implies the following:

Theorem: A 2-3 tree with n nodes has height $O(\log n)$.

It is easy to see how to perform the operation `find(x)` in a 2-3 tree. We apply the usual recursive descent as for a standard binary tree, but whenever we come to a 3-node, we need to check the relationship between x and the two key values in this node in order to decide which of the three subtrees to visit. The important issues are insertion and deletion, which we discuss next.

For conceptual purposes, it will be convenient to temporarily allow for the existence of "invalid" *1-nodes* and *4-nodes*. As soon as one of these exceptional nodes comes into existence, we will take action to replace them with proper 2-nodes and 3-nodes.

Insertion into a 2-3 tree: The insertion procedure follows the general structure that we have established with prior binary search trees. We first search for the insertion key, and make a note of the last node we visited just before falling out of the tree. Because all leaf nodes are at the same level, we always fall out at the lowest level of the tree. We insert the new key

into this leaf node. If the node was a 2-node, it now becomes a 3-node, and we are fine. If it was a 3-node, it now becomes a 4-node, and we need to fix it.

While the initial insertion takes place at a leaf node, we will see that the restructuring process can propagate to internal nodes. So, let us consider how to remedy the problem of a 4-node in a general setting. A 4-node has three keys, say b, d, and f and four children, say A, C, E, and G (see Fig. 2(a)).

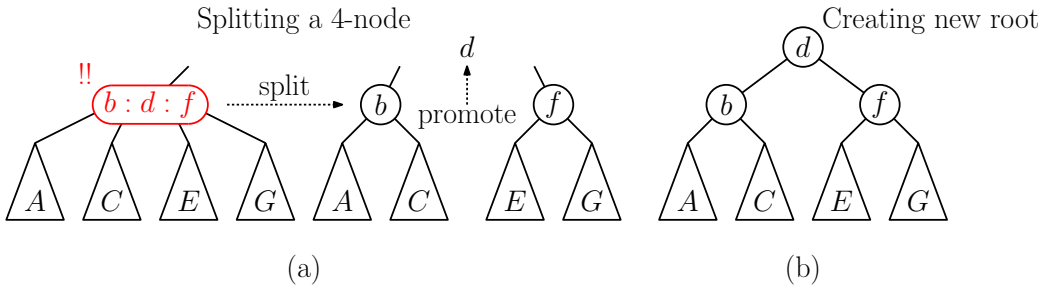

Fig. 2: 2-3 tree insertion: (a) splitting a 4-node into two 2-nodes and (b) creating a new root.

To resolve the problem we *split* this node into two 2-nodes: one for b with A and C as subtrees, and the other for f with E and G as subtrees. We then *promote* the middle key d by inserting it (recursively) into the parent node. What if there is no parent, because the current node is the root? We create a new root node storing d whose two children are b and f (see Fig. 2(b)). It is easy to check that this process preserves the 2-3 tree structural properties, and so we won't bother giving a proof.

In the figure below, we present an example of the result of inserting key 6 into a 2-3 tree. It requires two splits to restore the tree's structure.

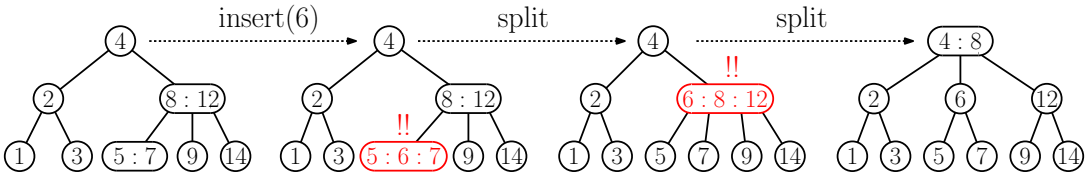

Fig. 3: 2-3 tree insertion involving two splits.

Deletion from a 2-3 tree: Consistent with our experience with binary search trees, deletion is more complicated than insertion. The general process follows the usual pattern. First, we find the key to be deleted. If it does not appear in a leaf node, then we identify a replacement key as the inorder successor. (The inorder predecessor would work equally well.) We copy the replacement key-value pair to replace the deleted key entry, and then we recursively delete the replacement key from its subtree. In this manner, we can always assume that we are deleting a key from a leaf node. So, let us focus on this.

As you might imagine, since insertion resulted in an overfull 4-node being split into two 2-nodes, the process of deletion will involve merging "underfull" nodes. This is indeed the case, but it will also be necessary to consider another restructuring primitive in which keys are taken or "adopted" from a sibling. This adoption process is also called *key rotation*, and I will use both terms interchangeably.

More formally, let us consider the deletion of an arbitrary key from an arbitrary node in the tree. If this is a 3-node, then the deletion results in a 2-node, which is fine. However, if this is a 2-node, the deletion results in an illegal 1-node, which has one subtree and zero keys. We remedy the situation in one of two ways.

Adoption (Key Rotation): Consider the left and right siblings of this node (if they exist). If either sibling is a 3-node, then it gives up an appropriate key (and subtree) to convert us back to 2-node. The tree is now properly structured.

Suppose, for the sake of illustration that the current node is an underfull 1-node, and it has a right sibling that is a 3-node (see Fig. 4(a)). Then, we adopt the leftmost key and subtree from this sibling, resulting in two 2-nodes. (A convenient mnemonic is the equation $1 + 3 = 2 + 2$.)

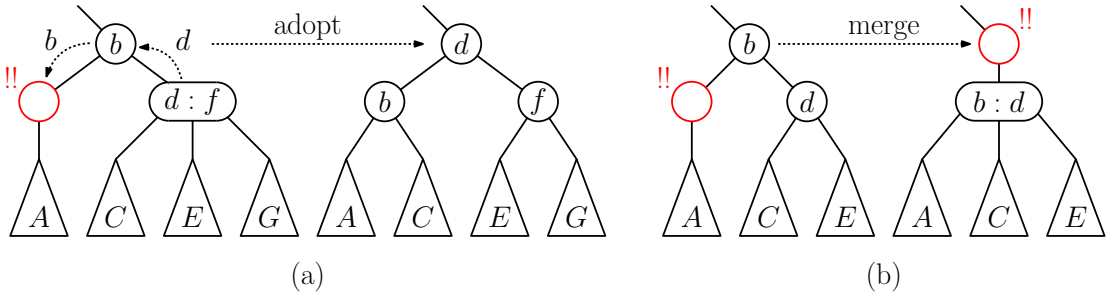

Fig. 4: 2-3 tree deletion: (a) adopting a key/subtree from a sibling and (b) merging two nodes.

Merging: On the other hand, if neither sibling can offer a key, then it follows that at least one sibling is a 2-node. Suppose for the sake of illustration that it is the right sibling (see Fig. 4(b)). In this case, we merge the 1-node with this 2-node to form a 3-node. (A convenient mnemonic is the equation $1 + 2 = 3$.)

But, we need a key to complete this 3-node. We take this key from our parent. If the parent was a 3-node, it now becomes a 2-node, and we are fine. If the parent was a 2-node, it has now become a 1-node (as in the figure), and the restructuring process continues on up the tree. Finally, if we ever arrive at a situation where the 1-node is the root of the tree, we remove this root and make its only child the new root.

An example of the deletion of a key is shown in Fig. 5. In this case, the initial deletion was from a 2-node, which left it as a 1-node. We merged it with its sibling to form a 3-node ($1 + 2 = 3$). This involved demoting the key 6 from the parent, which caused the parent to decrease from a 2-node to a 1-node. Since the parent has a 3-node sibling, we can adopt from it. (By the way, there is also a sibling which is a 2-node, containing the key 2. Could we have instead merged with this node? The answer is "yes", but it is not in our interest to do this. This is because merging results in more disruptions to ancestors of the tree, whereas a single adoption terminates the restructuring process.)

In Fig. 6 we illustrate two more examples of deletions from a 2-3 tree. In the upper example, we delete the key 3. Since its only sibling is a 2-node, we cannot perform an adoption, so we merge with the parent. (You might wonder, can't we adopt from our cousin, the $(6 : 7)$ node? The answer is "no." According to the 2-3 deletion algorithm, you can only adopt from your siblings. If we were to allow adoptions from arbitrarily distant cousins, the adoption process would take more than $O(1)$ time.) The parent node is now critical, and so we merge with its parent.

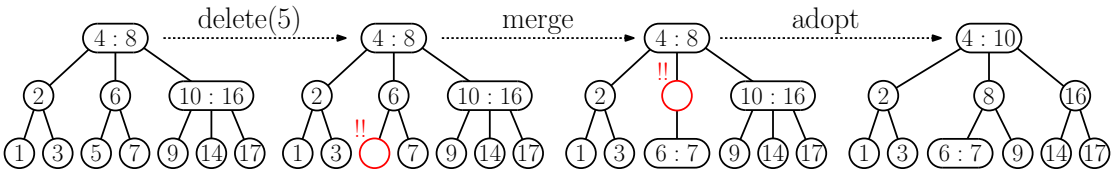

Fig. 5: 2-3 tree deletion involving a merge and an adoption.

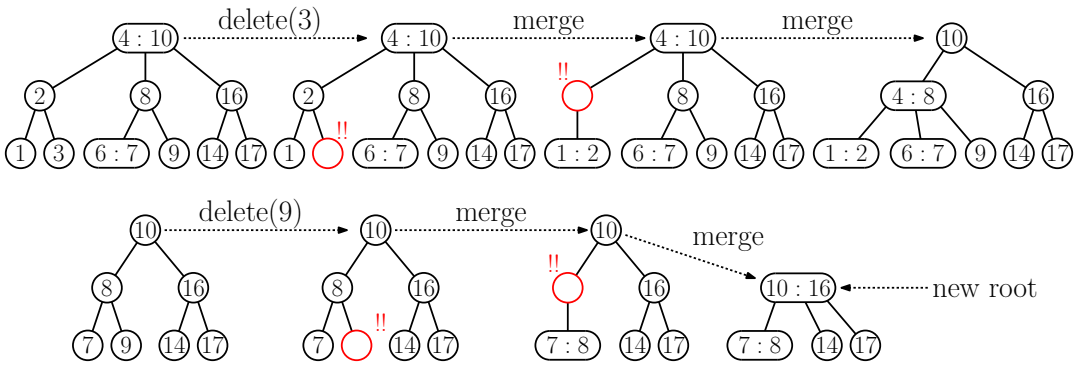

Fig. 6: Two more examples of 2-3 tree deletions.

In the second example in Fig. 6, we illustrate a case where the root node becomes critical. In this case, we remove the old root, and make its only child the new root.

Implementation? We will not discuss the 2-3 tree implementation in detail, since we will be presenting a number of related data structures, including red-black trees, AA trees, and B-trees in later lectures. The fact that there are two types of nodes might suggest using two different node classes (e.g., `TwoNode` and `ThreeNode`, which are both subclasses of a generic node class). However, because we often convert between one node type and the other (changing a 2-node into a 3-node and vice versa), it may be preferred to have just one node type that is capable of representing both, and adjusting the number of children/keys dynamically. An example is shown in the following code fragment.

2-3 Tree Node

```
class TwoThreeNode {
    int             nChildren;      // number of children (2 or 3)
    TwoThreeNode    child[3];       // children pointers
    Key             key[2];         // keys
    Value           value[2];       // values
}
```

In our examples, we temporarily created a 4-node, which we immediately split into two 2-nodes. This is really just a conceptual trick. In reality, a 4-node would never be generated. Instead, as soon as you see that you are about to create a 4-node, you would immediately invoke a split procedure to remedy the situation.

Another tricky implementation issue is that the current node will need to access its siblings on the left and right sides. To make this operation easier, it may be desirable to add a parent link to our node structure. Alternatively, the various recursive functions can be passed two node pointers as arguments, one to the current node and the other to its parent.