CMSC 420: Lecture 6
2-3, Red-black, and AA trees

"A rose by any other name": In today's lecture, we consider three closely related search trees. All three have the property that they support find, insert, and delete in time $O(\log n)$ for a tree with n nodes. Although the definitions appear at first glance to be different, they are essentially equivalent or very slight variants of each other. These are *2-3 trees*, *red-black trees*, and *AA trees*. Together, they show that the same idea can be viewed from many different perspectives.

2-3 Trees: An "ideal" binary search tree has n nodes and height roughly $\lg n$. (More precisely, the ideal would be $\lfloor \lg n \rfloor$, where we recall our convention that "lg" means logarithm base 2.) However, it is not possible to efficiently maintain a tree subject to such rigid requirements. AVL trees relax the height restriction by allowing the two subtrees associated with each node to be of *similar heights*.

Another way to relax the requirements is to say that a node may 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 requirement that for all $a \in A$, $c \in C$, and $e \in E$, we have

$$a < b < c < d < e,$$

(The concept of an *inorder traversal* of such a tree can be generalized, but it involves visiting each 3-node twice, once to visit the first key and again to visit the second key.) These are called *2-nodes* and *3-nodes*, respectively.

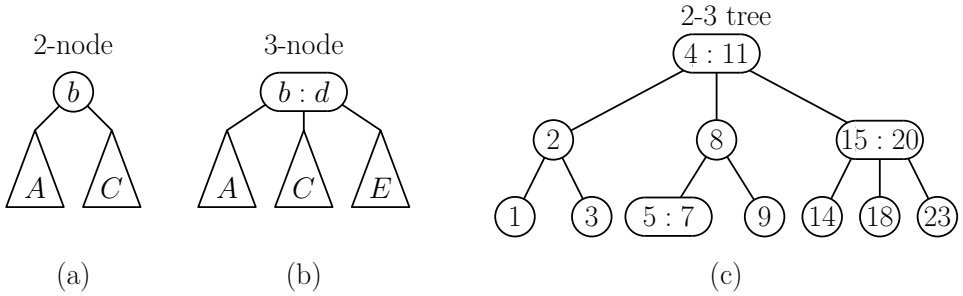

Fig. 1: (a) 2-node, (b) 3-node, and (c) a 2-3 tree.

A *2-3 tree* is defined recursively as follows. It is either:

- empty (i.e., `null`), or
- its root is a 2-node, and its two subtrees are each 2-3 trees of equal height, or
- its root is a 3-node, and its three subtrees are each 2-3 trees of equal height.

As we did with AVL trees, we define the height of an empty tree to be -1. (For an example, Fig. 1(c) shows a 2-3 tree of height 2.)

Since all the leaves are at the same level, and the sparsest possible 2-3 tree is a complete binary tree. We have the following direct consequence.

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

Since our foray into 2-3 trees will be brief, we will not bother to present an implementation. (Later this semester, we will discuss B-trees in detail, and a 2-3 tree is a special case of a B-tree.)

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.

In the descriptions that follow, 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 need to 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 AVL 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. 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 e with E and F as subtrees. We then *promote* the middle key d by inserting it (recursively) into the parent node (see Fig. 2(a)). If the parent is a 2-node, this can be accommodated without problem. On the other hand, if the parent is a 3-node, this creates a 4-node, and the splitting process continues recursively until either (a) we arrive at a node that does not overflow, or (b) we reach the root. When the root node overflows, the promoted key becomes the new root node, which must be a 2-node (see Fig. 2(b)). It is easy to see that this process preserves the 2-3 tree structural properties.

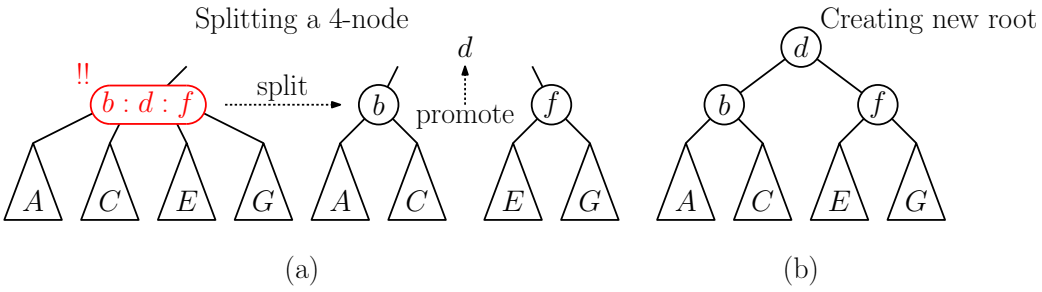

Fig. 2: 2-3 tree insertion: (a) splitting a 4-node into two 2-nodes and (b) creating a new root.

In the figure below, we present an example of the result of inserting key 6 into a 2-3 tree, which required two splits to resolve.

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

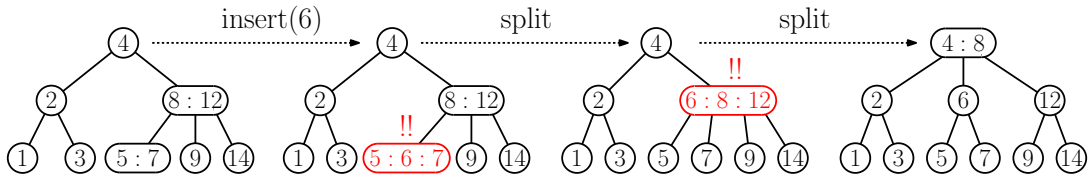

Fig. 3: 2-3 tree insertion involving two splits.

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.

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: 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 $1 + 3 = 2 + 2$.)

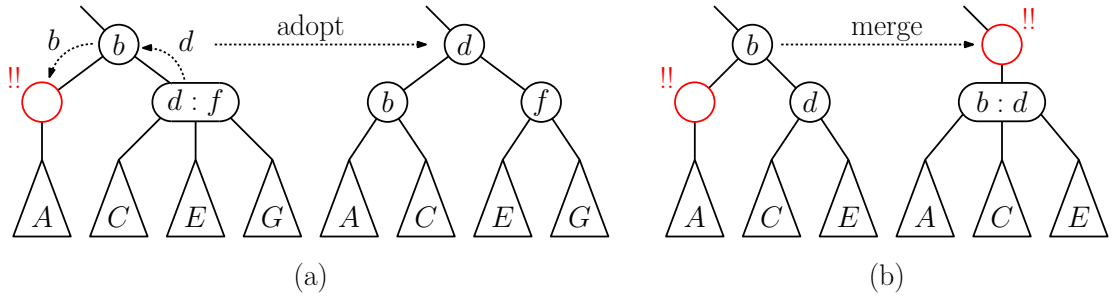

Fig. 4: 2-3 tree deletion: (a) adopting a key/subtree from a sibling and (b) merging two nodes.

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.)

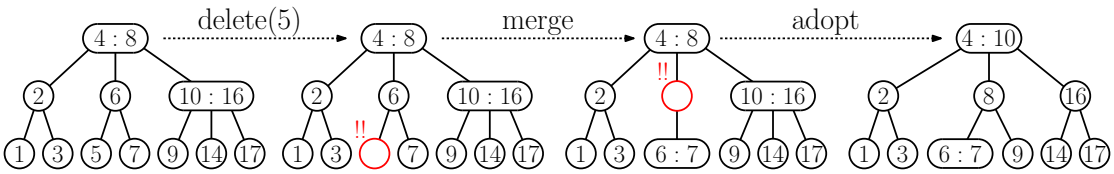

Fig. 5: 2-3 tree deletion involving a merge and an adoption.

Red-Black trees: While a 2-3 tree provides an interesting alternative to AVL trees, the fact that it is *not* a binary tree is a bit annoying. As we saw earlier in the semester, there are ways of representing arbitrary trees as binary trees. This suggests the idea of encoding a 2-3 tree as a binary tree. Unfortunately, the first-child, next-sibling approach presented earlier in the semester will not work. (Can you see why not? At issue is whether the inorder properties of the tree hold under this representation.)

Here is a simple approach that works, however. First, there is no need to modify 2-nodes, since they are already binary-tree nodes. To represent a 3-node as a binary-tree node, we create a two-node combination, as shown in Fig. 6(a) below. The 2-3 tree shown in Fig. 1(c) above would be represented in the manner shown in Fig. 6(b).

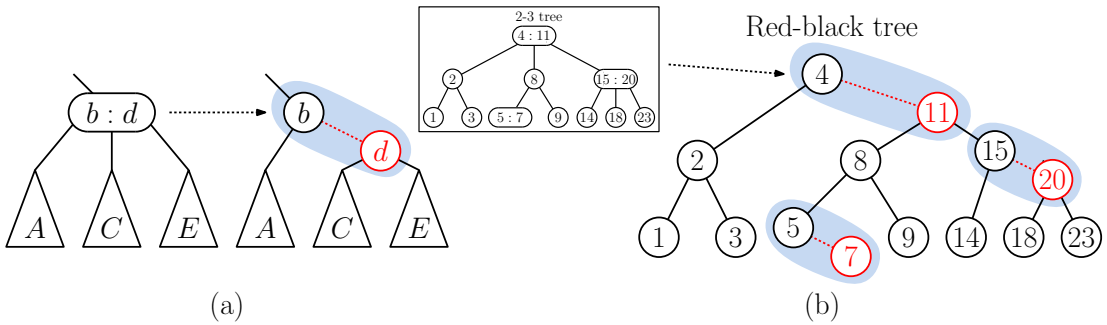

Fig. 6: Representing the 2-3 tree of Fig. 1 as an equivalent binary tree.

If we label each of "second nodes" of the 3-nodes as *red* and label all the other nodes as *black*, we obtain a binary tree with both red and black nodes. It is easy to see that the resulting binary tree satisfies the following properties:

- Each node is either red or black.
- The root is black.
- All `null` pointers are labeled as black. (This is just a convenient convention.)
- If a node is red, then both its children are black.

- Every path from a given node to any of its `null` descendants contains the same number of black nodes.

A binary search tree that satisfies these conditions is called a *red-black tree*. Because 2-3 trees have $O(\log n)$ height, the following is an immediate consequence:

Lemma: A red-black tree with n nodes has height $O(\log n)$.

It is easy to see that any the transformation that we described above for 2-3 trees always results in a valid red-black tree. Thus, we have:

Lemma: Every 2-3 tree corresponds to a red-black tree.

However, the converse is not true. There are two issues. First, the red-black conditions do not distinguish between left and right children, so a 3-node could be encoded in two different ways in a red-black tree (see Fig. 7(a)). More seriously, the red-black condition allows for the sort of structure in Fig. 7(b), which clearly does not correspond to a node of a 2-3 tree.

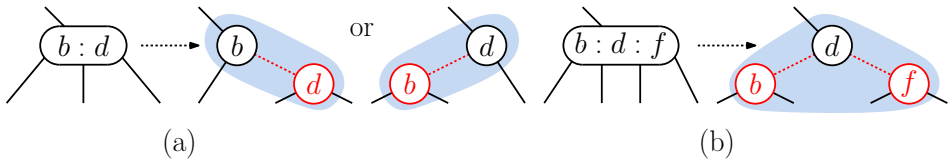

Fig. 7: Color combinations allowed by the red-black tree rules.

It is interesting to observe that this three-node combination can be seen as a way of modeling a node with four children. Indeed, there is a generalization of the 2-3 tree, called a *2-3-4 tree*, which allows 2-, 3-, and 4-nodes. Red-black trees as defined above correspond 1–1 with 2-3-4 trees. Red-black trees are the basis of `TreeMap` class in the `java.util` package. The principle drawback of red-black trees is that they are rather complicated to implement. For this reason, we will introduce a simpler variant of the red-black tree below, called an AA tree.

AA trees (Red-Black trees simplified): In an effort to simplify the complicated cases that arise with the red-black tree, in 1993 Arne Anderson developed a restriction of the red-black tree. He called his data structure a BB tree (for "Binary B-tree"), but over time the name has evolved into AA trees, named for the inventor (and to avoid confusion with another popular but unrelated data structure called a BB[α] tree).

Anderson's idea was to allow the conversion described above between 2-3 trees and red-black trees, and forbid the other red-black combinations. In particular, each red node can arise only as the *right child* of a black node. (The other rules of red-black trees are the same.) The edge between a red node and its black parent is called a *red edge*, and is shown as a dashed red edge in our figures. While AA-trees are simpler to code, they are a bit slower than red-black trees in practice.

The implementation of the AA tree has the following two noteworthy features:

We do not use `null` pointers: Instead, we create a special *sentinel node*, called `nil` (see Fig. 8(a)), and every `null` pointer is replaced with a pointer to `nil`. (Although the tree may have many `null` pointers, there is only one `nil` node allocated, with potentially many incoming pointers.) This node is considered to be black.

Why do this? Observe that `nil.left == nil` and `nil.right == nil`. This simplifies the code because we can always de-reference a pointer, without having to check first whether it is `null`.

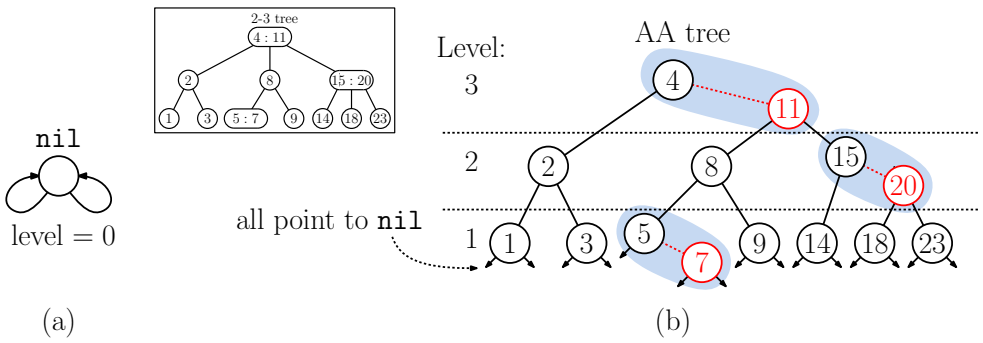

Fig. 8: AA trees: (a) the `nil` sentinel node, (b) the AA tree for the 2-3 tree of Fig. 1.

We do not store node colors: Instead, each node p stores a *level number*, denoted `p.level` (see Fig. 8(b)). Intuitively, the level number encodes the level of the associated node in the 2-3 tree. Formally, `nil` node is at level 0, and if `q` is a black child of some node `p`, then `p.level` = `q.level` + 1, and if `q` is a red child of `p`, then they have the same level numbers.

We do not need to store node colors, because we can determine whether a node is red by testing that its level number is the same as its parent.

It is surprising that our representation does not actually assign color to the nodes! It is not needed because color information is implicitly encoded in the level numbers. For example, if `p.right.level == p.level`, then we can infer that `p.right` is a red node (assuming it is not `nil`).

AA tree operations: Since an AA tree is essentially a binary search tree, the `find` operation is exactly the same as for any binary search tree. Insertions and deletions are performed in essentially the same way as for AVL trees: first the key is inserted or deleted at the leaf level, and then we retrace the search path back to the root and restructure the tree as we go. As with AVL trees, restructuring essentially involves rotations. For AA trees the two rotation operations go under the special names `skew` and `split`. They are defined as follows:

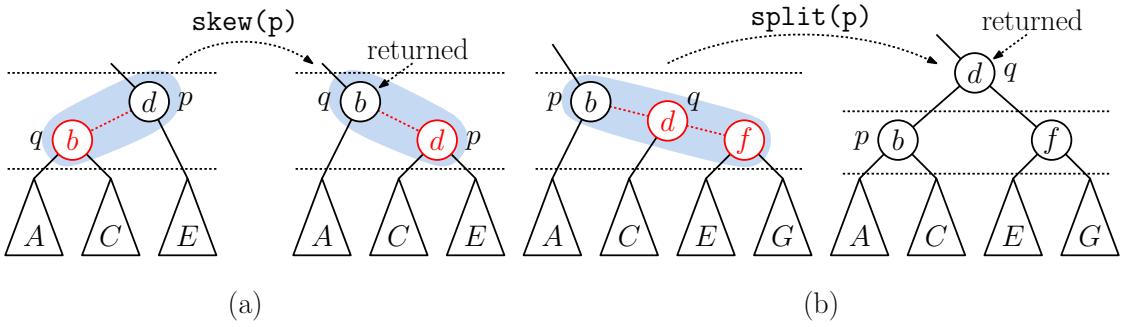

Fig. 9: AA restructuring opertions (a) skew and (b) split. (Afterwards q may be red or black.)

`skew(p)`: If p is black and has a red left child, rotate so that the red child is now on the right (see Fig. 9(a)). The level of these two nodes are unchanged. Return a pointer to upper node of the resulting subtree.

`split(p)`: If p is black and has a chain of two consecutive red nodes to its right , split this triple by performing a left rotation at p and promoting p's right child, call it q, to the next higher level (see Fig. 9(b)).

In the figure, we have shown p as a black node, but in the context of restructuring p may be either red or black. As a result, the node q that is returned from the operations may either be red or black. The implementation of these two operations is shown in the code block below.

AA tree skew and split utilities

```
AANode skew(AANode p) {
    if (p.left.level == p.level) {          // red node to our left?
        AANode q = p.left;                  // do a right rotation at p
        p.left = q.right;
        q.right = p;
        return q;                           // return pointer to new upper node
    }
    else return p;                          // else, no change needed
}

AANode split(AANode p) {
    if (p.right.right.level == p.level) {   // right-right red chain?
        AANode q = p.right;                 // do a left rotation at p
        p.right = q.left;
        q.left = p;
        q.level += 1;                       // promote q to next higher level
        return q;                           // return pointer to new upper node
    }
    else return p;                          // else, no change needed
}
```

AA-tree insertion: As mentioned above, we insert a node just as for a standard binary-search tree and then work back up the tree restructuring as we go. What sort of restructuring is needed? Recall first that (following the policies of 2-3 trees) all leaves should be at the same level of the tree. To achieve this, when the new node is inserted, we assign it the same level number as its parent. This is equivalent to saying that the newly inserted node is red (see Fig. 10(a)).

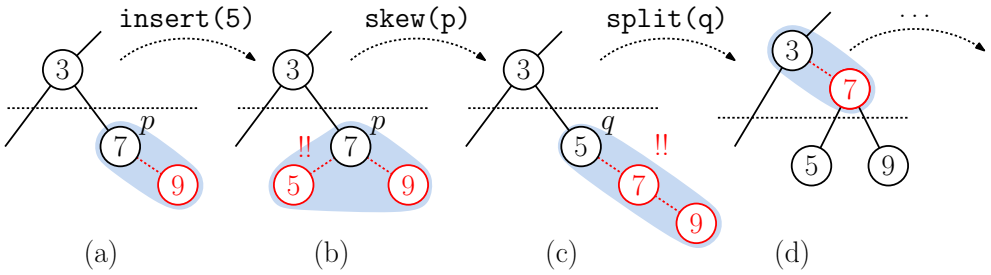

Fig. 10: AA insertion: (a) Initial tree, (b) after insertion, (c) after skewing, (d) after splitting.

The first problem might arise is that this newly inserted red node is a left child, which is not allowed (see Fig. 10(b)). Letting p denote the node's parent, this is easily remedied by performing `skew(p)` (see Fig. 10(c)). Let q be the pointer to the resulting subtree.

Next, it might be that p already had a right child that was red, and the skew could have resulted in a right-right chain starting from q. (This is equivalent to having a 4-node in a 2-3 tree.) We remedy this by invoking the split operation on q (see Fig. 10(d)). Note that the split operation moves the middle node of the chain up to the next level of the tree. The problems that we just experienced may occur with this promoted node, and so the skewing/splitting process generally propagates up the tree to the root.

The insertion function is provided in the following code block. In spite of this lengthy above explanation of how restructuring is performed, the entire restructuring process is handled very elegantly by the statement "`return split(skew(p))`". (This is the principle appeal of AA-trees over traditional red-black trees.)

AA Tree Insertion

```
AANode insert(Key x, Value v, AANode p) {
    if (p == nil)                           // fell out of the tree?
        p = new AANode(x, v, 1, nil, nil);  // ... create a new leaf node here
    else if (x < p.key)                     // x is smaller?
        p.left = insert(x, v, p.left);      // ...insert left
    else if (x > p.key)                     // x is larger?
        p.right = insert(x, v, p.right);    // ...insert right
    else
        throw DuplicateKeyException;        // duplicate key!
    return split(skew(p));                  // restructure and return result
}
```

An example of insertion is shown in Fig. 11 (mimicking the 2-3 tree of Fig. 3).

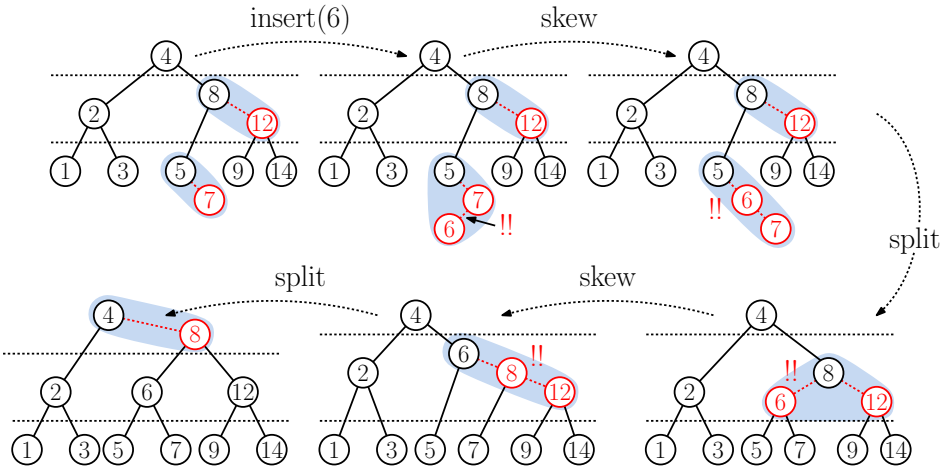

Fig. 11: Example of AA-tree insertion.

AA-tree deletion: As usual deletion is more complex than insertion. If this is not a leaf node, we find a suitable replacement node (it's inorder successor). We copy the contents of the replacement node to the deleted node and then we proceed to delete the replacement. After

deleting the replacement node (which must be a leaf), we retrace the search path towards the root and restructure as we go.

Before discussing deletion, let's first consider a useful utility function. In the process of deletion, a node can lose one of its children. As a result, we may need to decrease this node's level in the tree. To assist in this process we define two functions. The first, called `updateLevel(p)`, updates the level of a node `p` based on the levels of its children. Every node has at least one black child, and therefore, the ideal level of any node is one more than the minimum level of its two children. If we discover that `p`'s current level is higher than this ideal value, we set it to its proper value. If `p`'s right child is a red node (that is, `p.right.level == p.level` prior to the adjustment), then the level of `p.right` needs to be decreased as well.

AA-Tree update level utility

```
AANode updateLevel(AANode p) {
    int idealLevel = 1 + min(p.left.level,  p.right.level);
    if (p.level > idealLevel) {
        p.level = idealLevel;
        if (p.right.level > idealLevel)
            p.right.level = idealLevel;
    }
    return p;
}
```

When the restructuring process arrives at a node `p`, we first fix its level using `updateLevel(p)`. Next we need to skew to make sure that any red children are to its right. Deletion is complicated in that we may generally need to perform up to three skew operations to achieve this: one to `p`, one to `p.right`, and one to `p.right.right` (see Fig. 12). After this, `p` may generally be at the top of a right-leaning chain consisting of `p` followed by four red nodes. To remedy this, we perform two splits, one at `p`, and the other to its right-right grandchild, which becomes its right child after the first split (see Fig. 12). Whew! These splits may not be needed, but remember that the split function only modifies the tree if needed. The restructuring function, called `fixupAfterDelete`, is presented in the following code fragment.

AA-Tree Deletion Utility

```
AANode fixupAfterDelete(AANode p) {
    p = updateLevel(p);                         // update p's level
    p = skew(p);                                // skew p
    p.right = skew(p.right);                    // ...and p's right child
    p.right.right = skew(p.right.right);        // ...and p's right-right grandchild
    p = split(p);                               // split p
    p.right = split(p.right);                   // ...and p's (new) right child
    return p;
}
```

Finally, we can present the full deletion code. It looks almost the same as the deletion code for the standard binary search tree, but after deleting the leaf node, we invoke `fixupAfterDelete` to restructure the tree. All the operations (find, insert, and delete) take time proportional to the height of the tree, that is, $O(\log n)$.

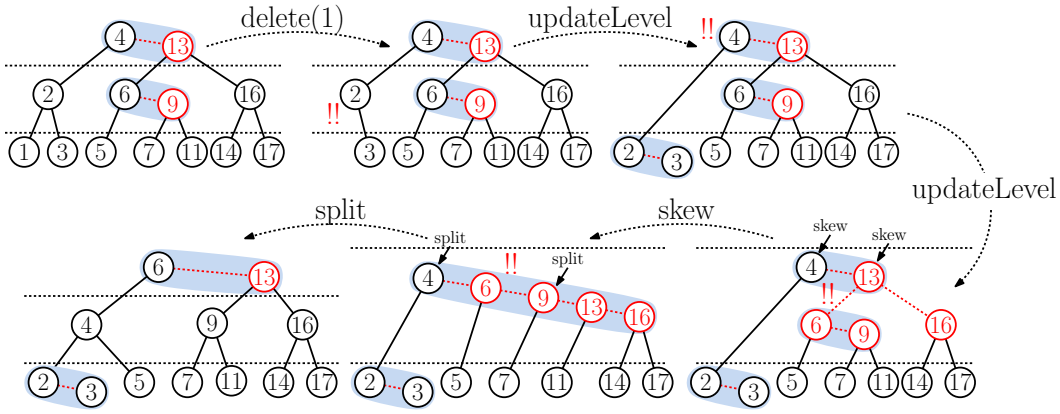

Fig. 12: Example of AA-tree deletion.

AA Tree Deletion

```
AANode delete(Key x, AANode p) {
    if (p == nil)                                  // fell out of tree?
        throw KeyNotFoundException;                // ...error - no such key
    else {
        if (x < p.key)                             // look in left subtree
            p.left = delete(x, p.left);
        else if (x > p.key)                        // look in right subtree
            p.right = delete(x, p.right);
        else {                                     // found it!
            if (p.left == nil && p.right == nil) // leaf node?
                return nil;                        // just unlink the node
            else if (p.left == nil) {              // no left child?
                AANode r = inorderSuccessor(p); // get replacement from right
                p.copyContentsFrom(r);             // copy replacement contents here
                p.right = delete(r.key, p.right); // delete replacement
            }
            else {                                 // no right child?
                AANode r = inorderPredecessor(p); // get replacement from left
                p.copyContentsFrom(r);             // copy replacement contents here
                p.left = delete(r.key, p.left); // delete replacement
            }
        }
        return fixupAfterDelete(p);                // fix structure after deletion
    }
}
```