# CMSC 420: Lecture 7
# Red-black and AA trees

**"A rose by any other name . . . ":** In the previous lecture, we presented the 2-3 tree, which allows nodes of variable widths. In this lecture, we will explore a closely related binary-tree data structure, called a red-black tree, and we will discuss a particular implementaiton, which is called an AA trees. (Before reading this lecture, please review the material on 2-3 trees.)

**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 inspires the question, "Can we encode a 2-3 tree as an equivalent 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. 1(a) below. The 2-3 tree shown in the inset would be encoded in the manner shown in Fig. 1(b).

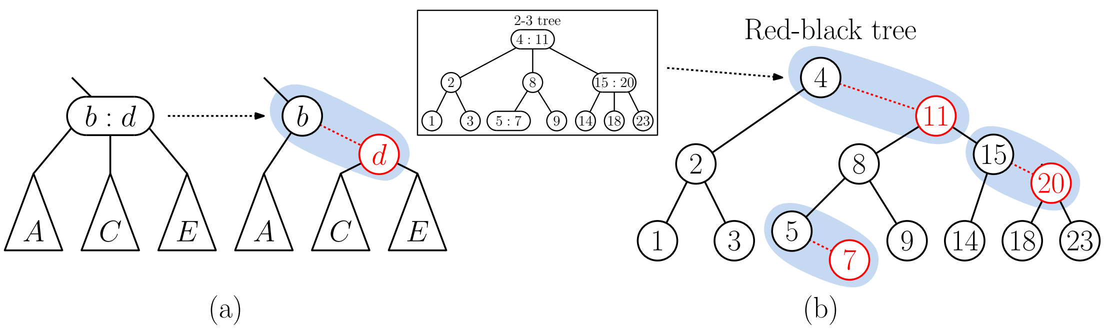


Fig. 1: Encoding a 2-3 tree as an equivalent binary tree.

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:

(1) Each node is either red or black.

(2) The root is black.

(3) All `null` pointers are treated *as if* they point to black nodes (a conceptual convenience).

(4) If a node is red, then both its children are black.

(5) 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*. It is easy to see that the above encoding of any 2-3 tree to this binary form satisfies all of these properties. (On the other hand, if you just saw this list of properties without having seen the 2-3 tree, it would seem to be very arcane!) 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 $O(\log n)$.

Note that not every red-black tree is the binary encoding of a 2-3 tree. 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. 2(a)). More seriously, the red-black condition allows for the sort of structure in Fig. 2(b), which clearly does not correspond to a node of a 2-3 tree.

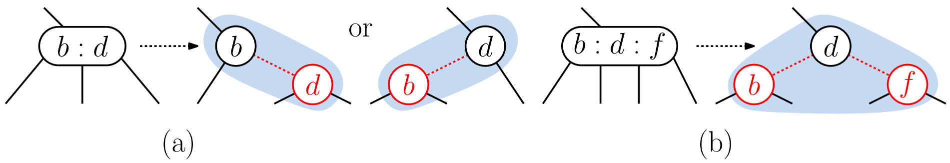


Fig. 2: Color combinations allowed by the red-black tree rules.

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 variant of the red-black tree below, called an AA tree, which is easier to code.

**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[$\alpha$] tree).

Anderson's idea was to allow the conversion described above between 2-3 trees and red-black trees but add a rule that forbids the alternate configurations shown in Fig. 2. The additional rule is:

(6) Each red node can arise only as the right child of a black node.

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. Note that, while AA-trees are simpler to code, experiments show that are a bit slower than red-black trees in practice.

The implementation of the AA tree has the following two noteworthy features, which further simplify the coding:

**We do not use `null` pointers:** Instead, we create a special *sentinel node*, called `nil` (see Fig. 3(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`.

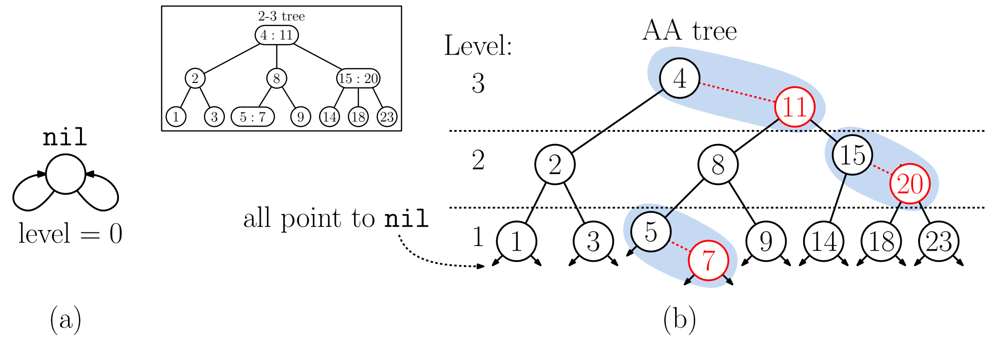


Fig. 3: AA trees: (a) the `nil` sentinel node, (b) the AA tree for a 2-3 tree.

**We do not store node colors:** Instead, each node `p` stores a *level number*, denoted `p.level` (see Fig. 3(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. For example, we can determine whether a black node `p`'s right child is red, it suffices to test `p.right.level == p.level`.

**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:

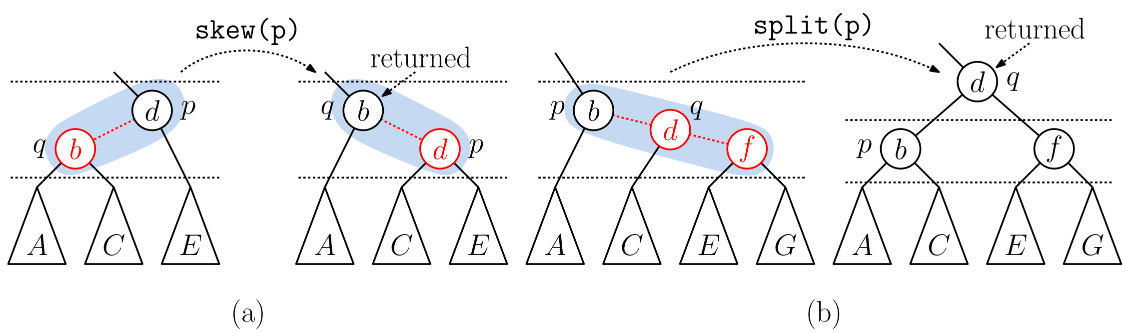


Fig. 4: AA restructuring opertions (a) skew and (b) split. (Afterwards `q` may be red or black.)

`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. 4(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 (that is, `p.level == p.right.level == p.right.right.level`), 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. 4(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 == nil) return 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 == nil) return 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. 5(a)).

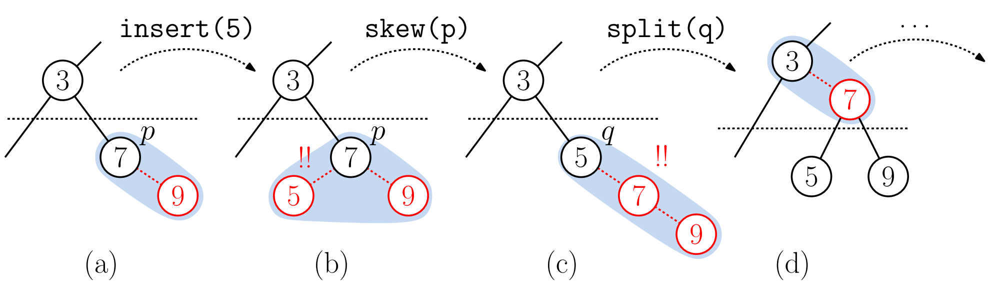


Fig. 5: AA insertion: (a) Initial tree, (b) after insertion, (c) after skewing, (d) after splitting.

The first problem might arise is that this newly inserted red node is a left child, which is not allowed (see Fig. 5(b)). Letting `p` denote the node's parent, this is easily remedied by performing `skew(p)` (see Fig. 5(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. 5(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. Observe that (as with the AVL tree) the function is almost the same as the standard (unbalanced) binary tree insertion except for the final rebalancing step, which is performed by the call "`return split(skew(p))`". (This simplicity 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. 6. (See the lecture on 2-3 trees for the analogous process.)

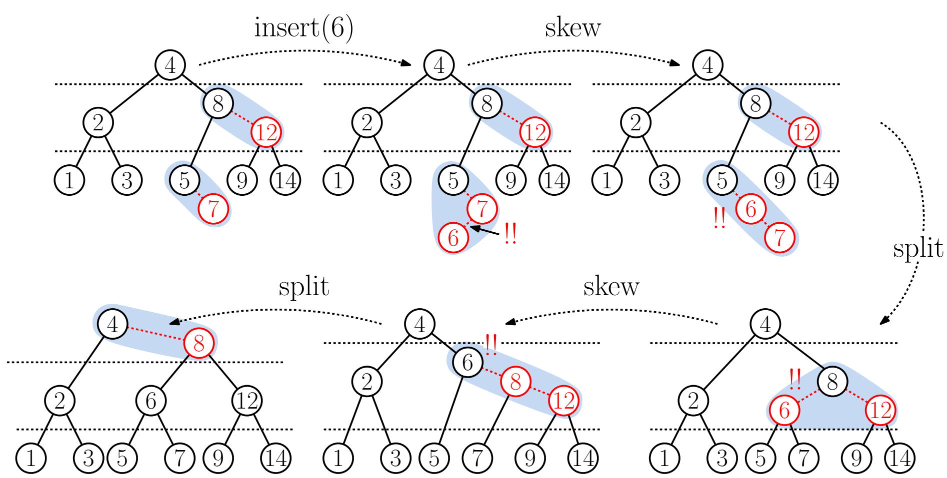


Fig. 6: Example of AA-tree insertion. (Remember, a node is red if it is at the same level as its parent.)

**AA-tree deletion:** As usual deletion is more complex than insertion. If this is not a leaf node, we find a suitable replacement node. (This will either be the inorder predecessor or inorder successor, depending on the tree's structure.) 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

```
void updateLevel(AANode p) {                        // update p's level
    int idealLevel = 1 + min(p.left.level, p.right.level);
    if (p.level > idealLevel) {                     // is p's level too high?
        p.level = idealLevel;                       // decrease its level
        if (p.right.level > idealLevel)             // is p's right child red?
            p.right.level = idealLevel;             // ...pull it down as well
    }
}
```

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 on `p`, one on `p.right`, and one on `p.right.right` (see Fig. 7). 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. 7). Whew! These splits may not be needed, but remember that the split function only modifies the tree if needed. The restructuring function, called `fixAfterDelete`, is presented in the following code fragment. As an exercise, you might draw the equivalent 2-3 tree both before and after the deletion. You will discover that, although the intermediate results differ, the final tree is the encoding of the 2-3 tree after the deletion.

AA-Tree Deletion Utility

```
AANode fixAfterDelete(AANode 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 `fixAfterDelete` to restructure the tree. We will omit the (messy) details showing that after this restructuring, the tree is in valid AA-form. (We refer you to Anderson's original paper.)

**Analysis:** All of these algorithms take $O(1)$ time per level of the tree, which implies that the running time of all the dictionary operations is $O(h)$ where $h$ is the height of the tree. As we

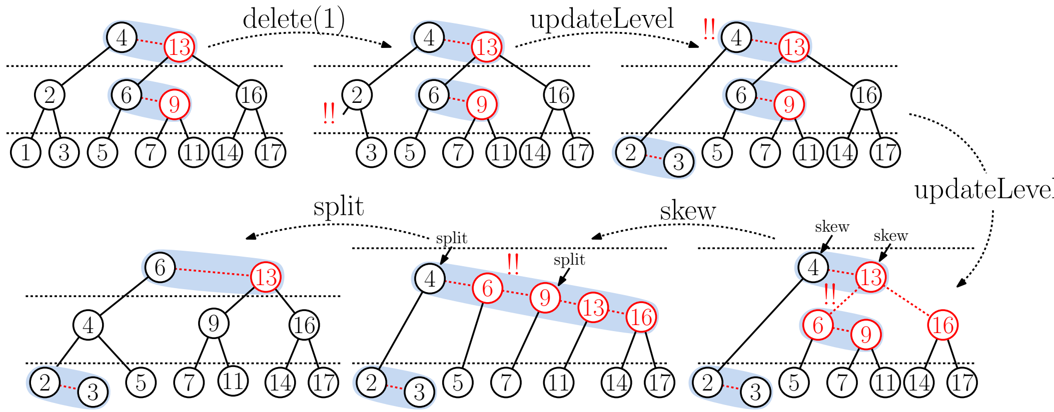


Fig. 7: Example of AA-tree deletion. (Remember, a node is red if it is at the same level as its parent.)

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 fixAfterDelete(p);              // fix structure after deletion
    }
}
```

saw abow, the tree's height is $O(\log n)$ height, which implies that all the dictionary operations run in $O(\log n)$ time.