Lecture X01: Red-Black Trees

Reading: Section 5.2 in Samet's notes.

Red-Black Trees: Red-Black trees are balanced binary search trees. Like the other structures we have studied (AVL trees, B-trees) this structure provides $O(\log n)$ insertion, deletion, and search times. What makes this data structure interesting is that it bears a similarity to both AVL trees (because of the style of manipulation through rotations) and B-trees (through a type of isomorphism). Our presentation follows Samet's. Note that this is different from usual presentations, since Samet colors edges, and most other presentations color vertices.

A *red-black tree* is a binary search tree in which each edge of the tree is associated with a color, either red or black. Each node t of the data structure contains an additional one-bit field, `t.color`, which indicates the color of the incoming edge from the parent. We will assume that the color of nonexistent edge coming into the root node is set to black. Define the *black-depth* of a node to be the number of black edges traversed from the root to that node. A red-black tree has the following properties, which assure balance.

(1) Consider the external nodes in the augmented binary tree (where each null pointer in the original tree is replaced by a black edge to a special node). The black-depths of all these external nodes are equal. (Note: The augmented tree is introduced for the purposes of definition only, we do not actually construct these external nodes in our representation of red-black trees.)

(2) No path from the root to a leaf has two consecutive red edges.

(3) (Optional) The two edges leaving a node to its children cannot both be red.

Property (3) can be omitted without altering the basic balance properties of red-black trees (but the rebalancing procedures will have to be modified to handle this case). We will assume that property (3) is enforced. An example is shown in the figure below. Red edge are indicated with dashed lines and black edges with solid lines. The small nodes are the external nodes (not really part of the tree). Note that all these nodes are of black-depth 3 (counting the edge coming into them).

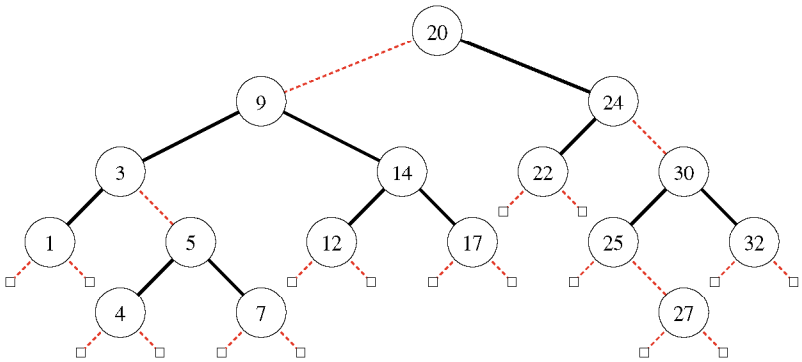

Figure 83: Red-black tree.

Red-black trees have obvious connections with 2-3 trees (B-trees of order $m = 3$), and 2-3-4 trees (B-trees of order $m = 4$). When property (3) is enforced, the red edges are entirely

isolated from one another. By merging two nodes connected by a red-edge together into a single node, we get a 3-node of a 2-3 tree tree. This is shown in part (a) of the figure below. (Note that there is a symmetric case in which the red edge connects a right child. Thus, a given 2-3 tree can be converted into many different possible red-black trees.)

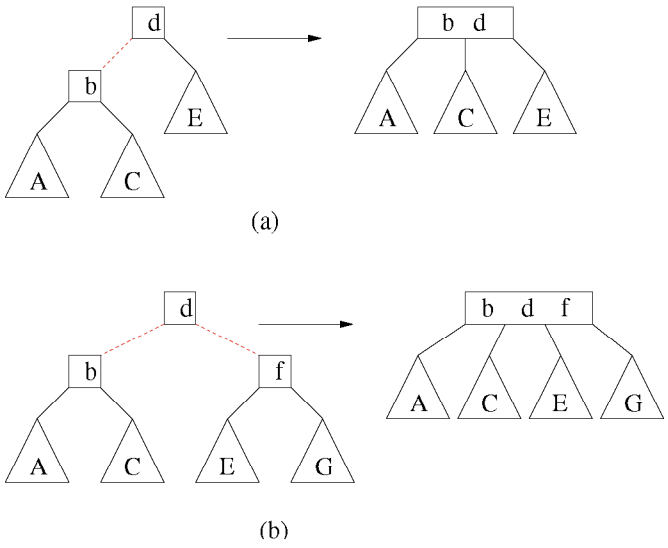

Figure 84: Red-black and 2-3 or 2-3-4 trees.

In the case where property (3) is not enforced, then it is possible to have a two red sibling edges. If we join all three nodes connected by these edges we get a 4-node (see part (b) of the same figure). Thus by enforcing property (3) we get something isomorphic to a 2-3 tree, and by not enforcing it we get something isomorphic to 2-3-4 trees. For example, the red-black tree given earlier could be expressed as the following (2-3)-tree.

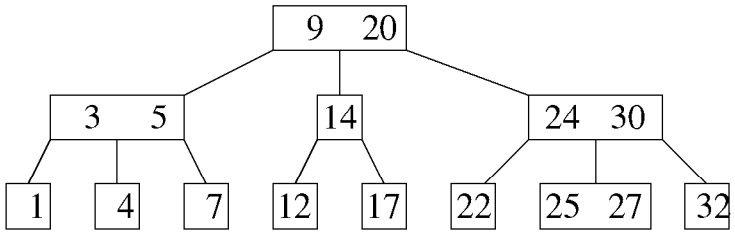

Figure 85: Equivalent (2-3) tree.

Thus red-black trees are certainly similar to B-trees. One reason for studying red-black trees independently is that when storing B-trees in main memory (as opposed to the disk) red-black trees have the nice property that every node is fully utilized (since B-tree nodes can be as much as half empty). Another reason is to illustrate the fact that there are many ways to implement a single "concept", and these other implementations may be cleaner in certain circumstances. Unfortunately, red-black trees cannot replace B-trees when it comes to storing data on disks.

It follows that the height of a red-black tree of n nodes is $O(\log n)$, since these trees are essentially the same as 2-3 or 2-3-4 trees, which also have logarithmic height.

Red-Black Insertion: It should be obvious from the discussion above that the methods used to restore balance in red-black tree should be an easy extension of rebalancing in B-trees. We simply see what a B-tree would do in a particular circumstance, and figure out what the equivalent rebalancing in the red-black tree is needed. Here is where the similarity with AVL trees comes in, because it turns out that the fundamental steps needed to rebalance red-black trees are just rotations. Let's start by considering node insertion.

To insert an key K into a red-black tree, we begin with the standard binary tree insertion. Search for the key until we either find the key, or until we fall out of the tree at some node x. If the key is found, generate an error message. Otherwise create a new node y with the new key value K and attach it as a child of x. Assign the color of the newly created edge (y, x) to be red. Observe at this point that the black-depth to every extended leaf node is unchanged (since we have used a red edge for the insertion), but we may have have violated the red-edge constraints. Our goal will be to reestablish the red-edge constraints. We backtrack along the search path from the point of insertion to the root of the tree, performing rebalancing operations.

Our description of the algorithm will be nonrecursive, because (as was the case with splay trees) some of the operations require hopping up two levels of the tree at a time. The algorithm begins with a call to `insert()`, which is the standard (unbalanced) binary tree insertion. We assume that this routine returns a pointer `y` to the newly created node. In the absence of recursion, walking back to the root requires that we save the search path (e.g. in a stack, or using parent pointers). Here we assume parent pointers (but it could be done either way). We also assume we have two routines `leftRotate()` and `rightRotate()`, which perform a single left and single right rotation, respectively (and update parent pointers).

Whenever we arrive at the top of the while-loop we make the following assumptions:

- The black-depths of all external leaf nodes in the tree are equal.
- The edge to y from its parent is red (that is, `y.color == red`).
- All edges of the tree satisfy the red-conditions (2) and (3) above, except possibly for the edge to y from its parent.

```
RBinsert(Object K, RBNode root) {
    y = insert(K, root)                // standard insert; let y be new node
    y.color = red                      // y's incoming edge is red
    while (y != root) {                // repeat until getting back to root
        x = y.parent                   // x is y's parent
        if (x.color == black) {        // edge into x is black?
            z = otherChild(x,y)        // z is x's other child
            if (z == null || z.color == black)
                return                 // no red violation - done
            else {                     // Case 1
                y.color = black        // reverse colors
                z.color = black
                x.color = red
                y = x                  // move up one level higher
            }
        }
        else {                         // edge into x is red
            w = x.parent               // w is x's parent
            if (x == w.left) {         // x is w's left child
                if (y == x.right) {    // Case 2(a): zig-zag path
                    leftRotate(x)      // make it a zig-zig path
                    swap(x,y)          // swap pointers x and y
```

```
                }
                rightRotate(w)          // Case 2(b): zig-zig path
            }
            else {                      // x is w's right child
                ...this case is symmetric...
            }
            y.color = black             // reverse colors
            w.color = black
            x.color = red
            y = x;                      // move up two levels
        }
    }
    root.color = black;                 // edge above root is black
}
```

Here are the basic cases considered by the algorithm. Recall that y is the current node, whose incoming edge is red, and may violate the red-constraints. We let x be y's parent, and z be the other child of x.

Case 1: (Edge above x is black:) There are two subcases. If z is null or the edge above z is black, then we are done, since edge (x, y) can have no red neighboring edges, and hence cannot violate the red-constraints. Return.

Otherwise if (x, z) exists and is red, then we have violated property (3). Change the color of edges above y and z to be black, change the color of the edge above x to be red. (Note: This corresponds to a node split in a 2-3 tree.)

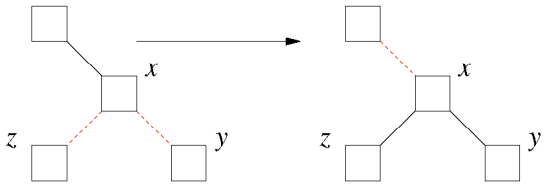

Figure 86: Case 1.

Case 2: (Edge above x is red:) Since the root always has an incoming black edge we know x is not the root, so let w be x's parent. In this case we have two consecutive red edges on a path. If the path is a zig-zag path (Case 2(a)), we apply a single rotation at x and swap x and y to make it a zig-zig path, producing Case 2(b). We then apply a single rotation at w to make the two red edges siblings of each other, and bringing x to the top. Finally we change the lower edge colors to black, and change the color of the edge above x to be red. (Note: This also corresponds to a node split in a 2-3 tree.)

We will not discuss red-black deletion, but suffice it to say that it is only a little more complicated, but operates in essentially the same way.

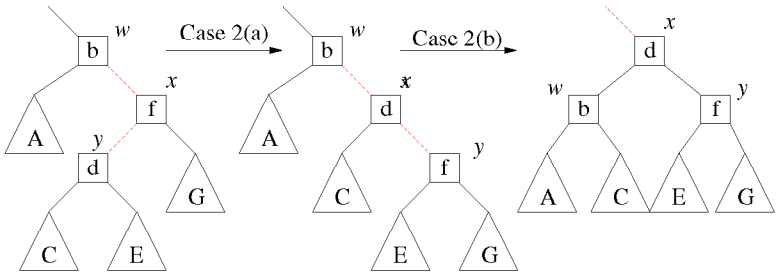

Figure 87: Case 2.

Lecture X02: BB-trees

Reading: This is not covered in our readings.

BB-trees: We introduced B-trees earlier, and mentioned that they are good for storing data on disks. However, B-trees can be stored in main memory as well, and are an alternative to AVL trees, since both guarantee $O(\log n)$ wost case time for dictionary operations (not amortized, not randomized, not average case). Unfortunately, implementing B-trees is quite a messy programming task in general. BB-trees are a special binary tree implementation of 2-3 trees, and one of their appealing aspects is that they are very easy to code (arguably even easier than AVL trees).

Recall that in a 2-3 tree, each node has either 2 or 3 children, and if a node has j children, then it has $j - 1$ key values. We can represent a single node in a 2-3 tree using either 1 or 2 nodes in a binary tree, as illustrated below.

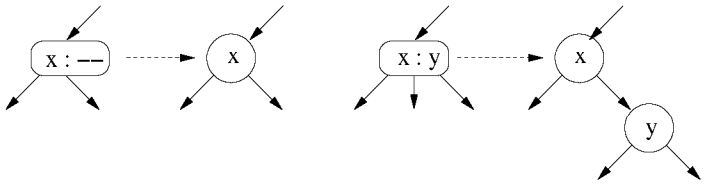

Figure 88: Binary representation of 2-3 tree nodes.

When two nodes of a BB-tree are used to represent a single node of a 2-3 tree, this pair of nodes is called *pseudo-node*. When we represent a 2-3 tree using this binary representation, we must be careful to keep straight which pairs of vertices are pseudo-nodes. To do this, we create an additional field in each node that contains the *level* of the node in the 2-3 tree. The leaves of the 2-3 tree are at level 1, and the root is at the highest level. Two adjacent nodes in a BB-tree (parent and right child) that are of equal level form a single pseudo-node.

The term "BB-tree" stands for "binary B-tree". Note that in other textbooks there is a data structure called a "bounded balance" tree, which also goes by the name BB-tree. Be sure you are aware of the difference. BB-trees are closely related to red-black trees, which are a binary representation of 2-3-4 trees. However, because of their additional structure the code for BB-trees is quite a bit simpler than the corresponding code for red-black trees.

As is done with skip-lists, it simplifies coding to create a special *sentinel* node called `nil`.