

Searching V

Left-Leaning Red-Black Trees

BST vs. 2-3 Trees

perfectly balanced →

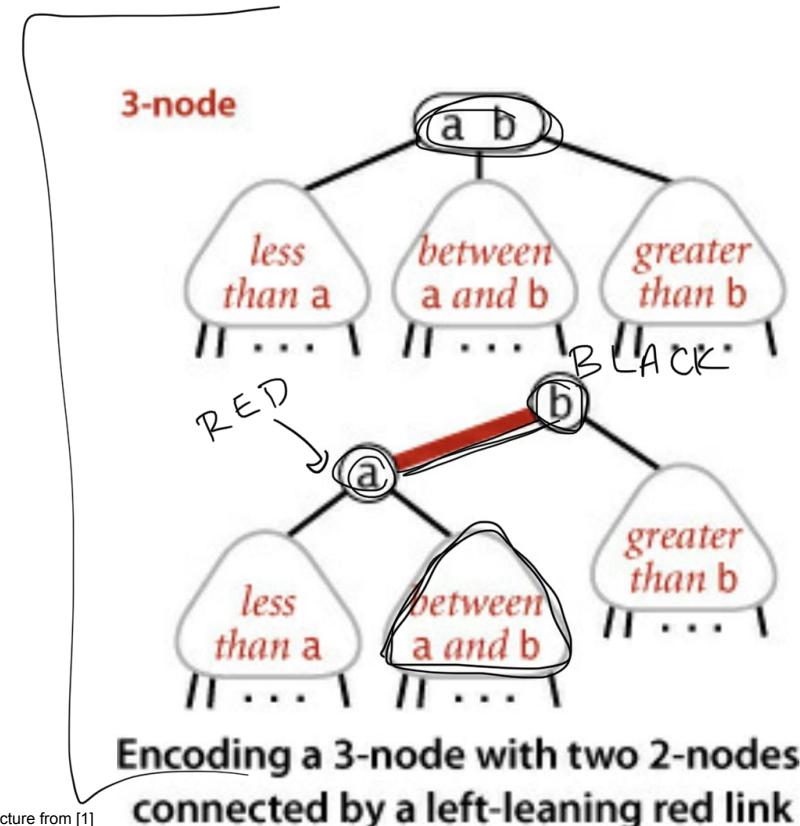
$O(\log N)$

for main operations

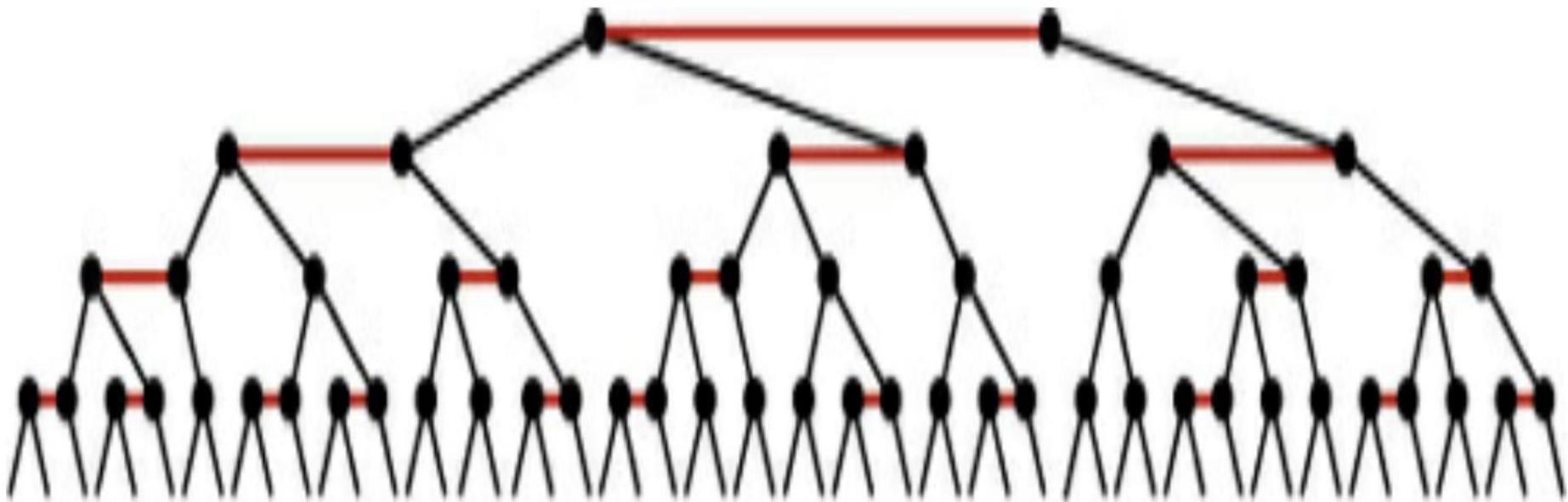
↓
difficult to implement

(Left-leaning) Red-Black Trees

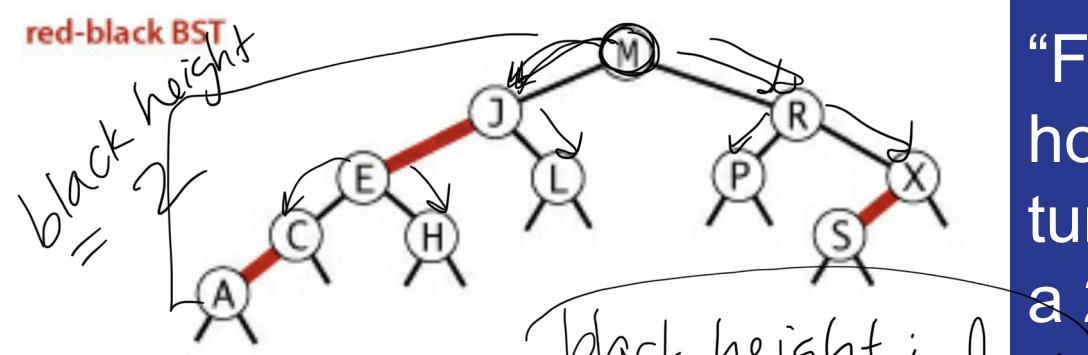
- A (left-leaning) red-black tree combines the simplicity of a BST with the balance of a 2-3 Tree.
- There is a 1 - to - 1 correspondence between a 2-3 tree and a left-leaning red-black tree.
- We start with a regular BST and add “color” to the nodes: RED or BLACK.



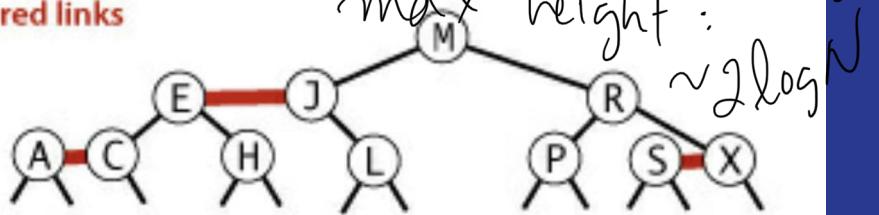
“Flattening” the red links into horizontal links essentially turns the binary R-B tree into a 2-3 tree.



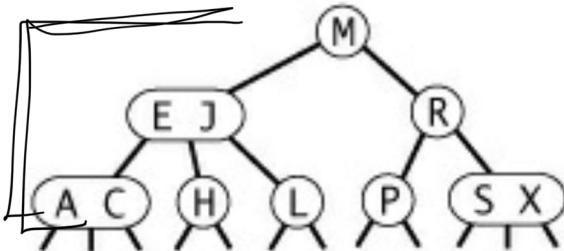
A red-black tree with horizontal red links is a 2-3 tree



horizontal red links



2-3 tree



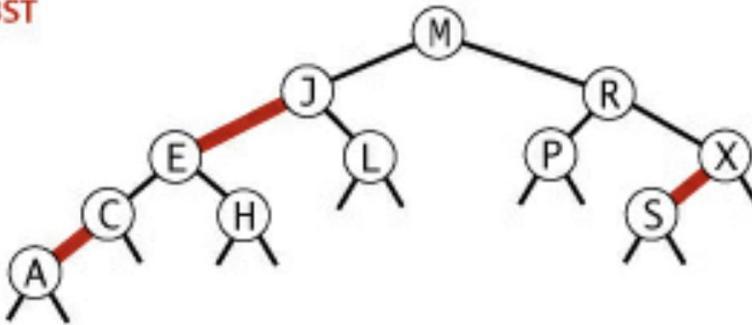
“Flattening” the red links into horizontal links essentially turns the binary R-B tree into a 2-3 tree.

Red-Black Tree: A BST with these properties...

- Links are **red** or **black**
 - Typically, we call the **nodes** red or black depending on the color of the link that connects it to its parent because there is no “link” object. It’s just a pointer from one node to another.
 - We can encode this by adding a boolean variable to each node (**true if red, false if black**).
- Red nodes cannot have red children.
- Red nodes cannot have red siblings.
- The root is always black.
- Tree has perfect black balance (i.e. all paths from the root to a leaf have the same number of black links—also known as the **black height** of the tree)

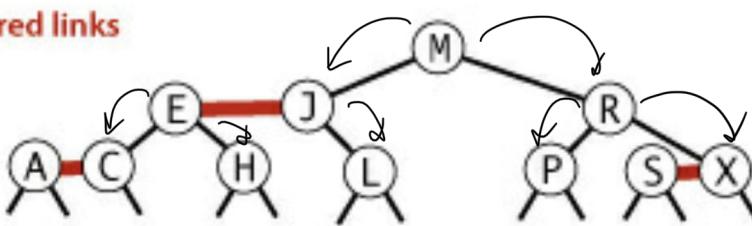
In a *left-leaning* red-black tree, all red links lean to the left. This is an implementation decision that can reduce the number of “cases” and therefore shorten the code.

red-black BST

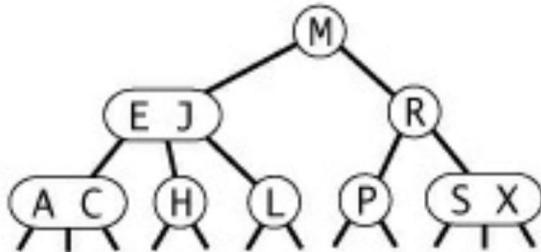


Understanding Black Balance

horizontal red links



2-3 tree

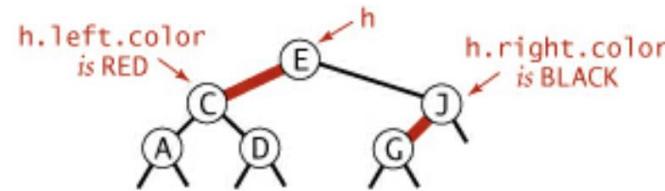


1-1 correspondence between red-black BSTs and 2-3 trees

Picture from [1]

Color Representation

Each node has a new boolean variable for color (according to the link connecting it to its parent)—**true if red, false if black.**



```
private static final boolean RED = true;
private static final boolean BLACK = false;

private class Node
{
    Key key;           // key
    Value val;         // associated data
    Node left, right; // subtrees
    int N;             // # nodes in this subtree
    boolean color;    // color of link from
                      // parent to this node

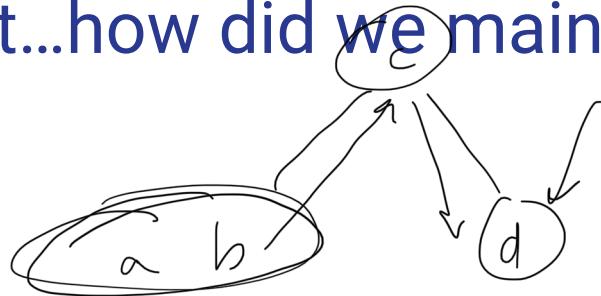
    Node(Key key, Value val, int N, boolean color)
    {
        this.key = key;
        this.val = val;
        this.N = N;
        this.color = color;
    }
}

private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

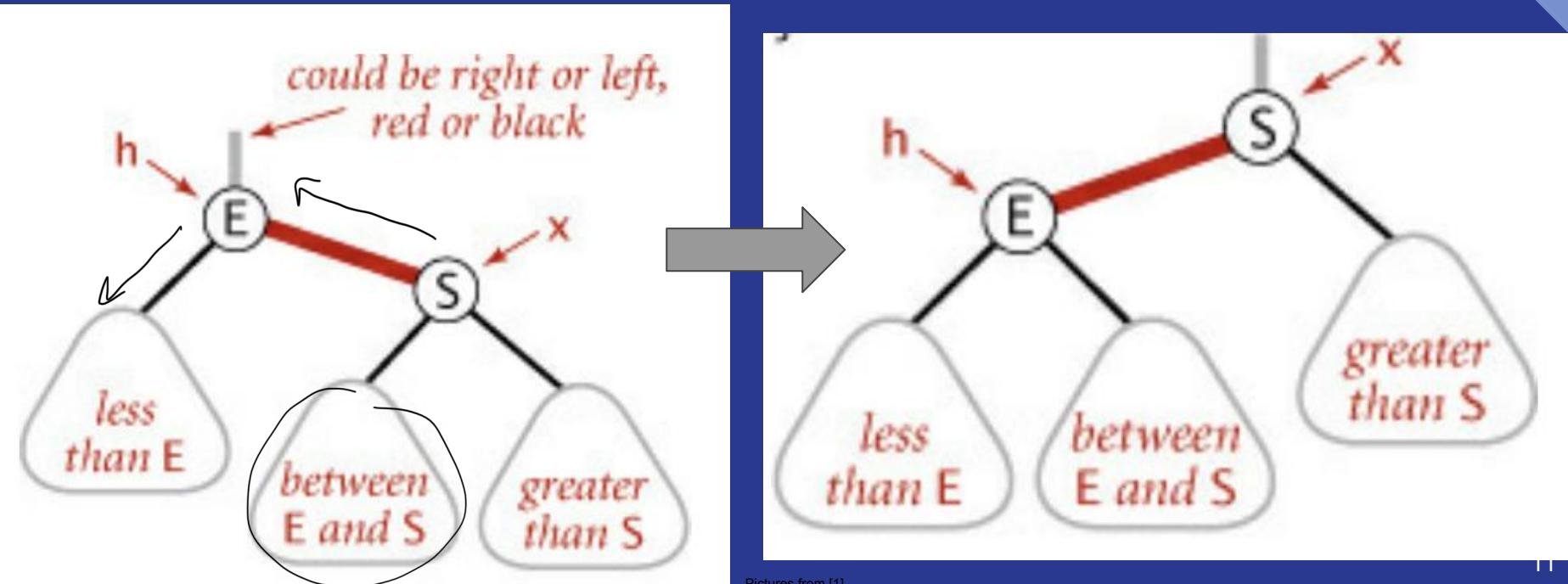
Transformations

This is how we maintain the properties that ensure that the black height remains balanced.

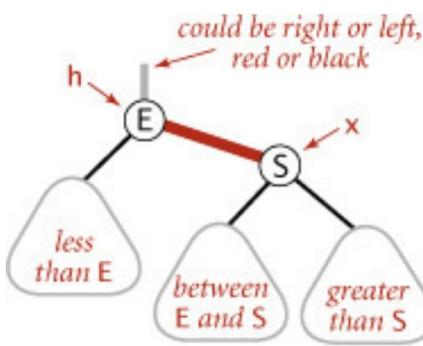
But first...how did we maintain balance in a 2-3 Tree?



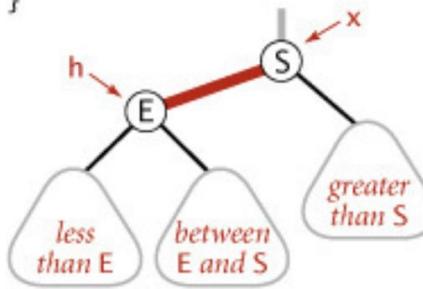
Left Rotation



Left Rotation Code

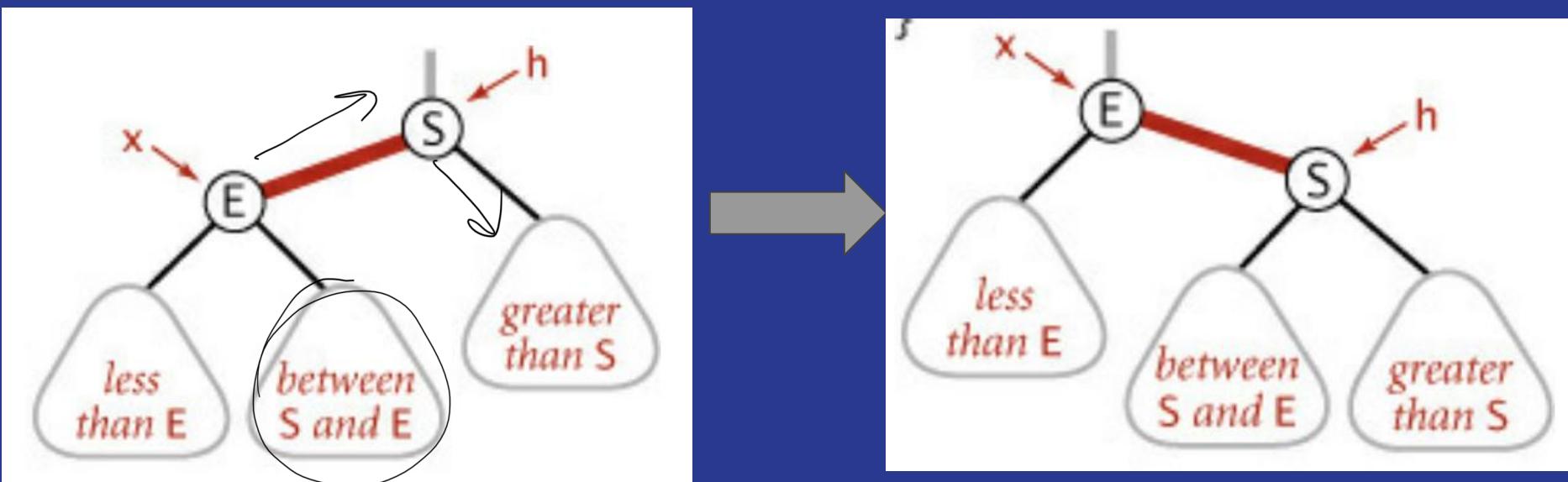


```
Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
          + size(h.right);
    return x;
}
```

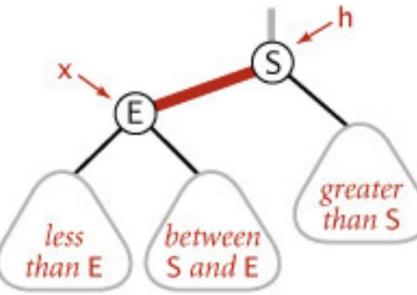


Left rotate (right link of h)

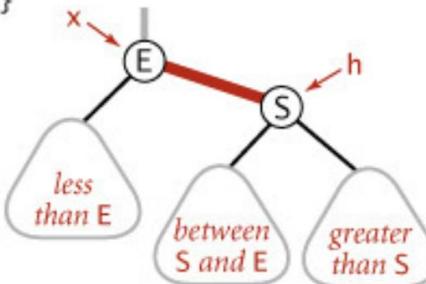
Right Rotation



Right Rotation Code

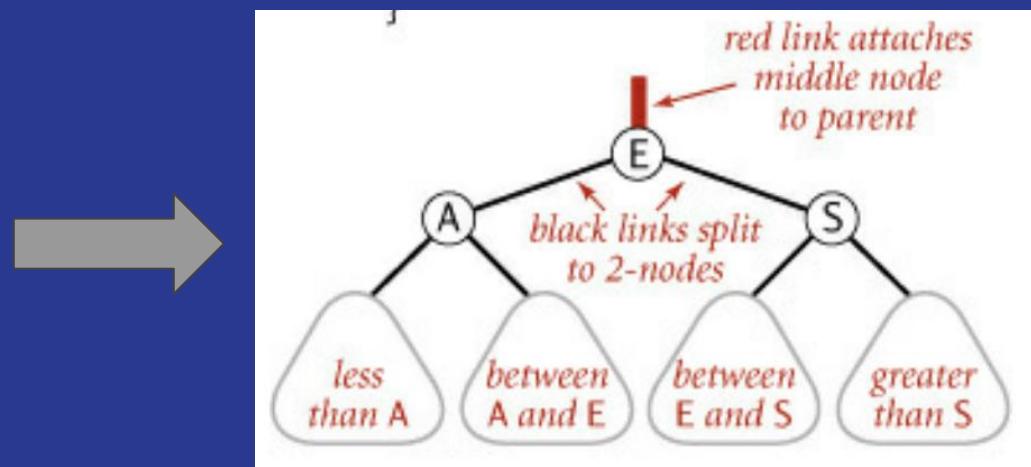
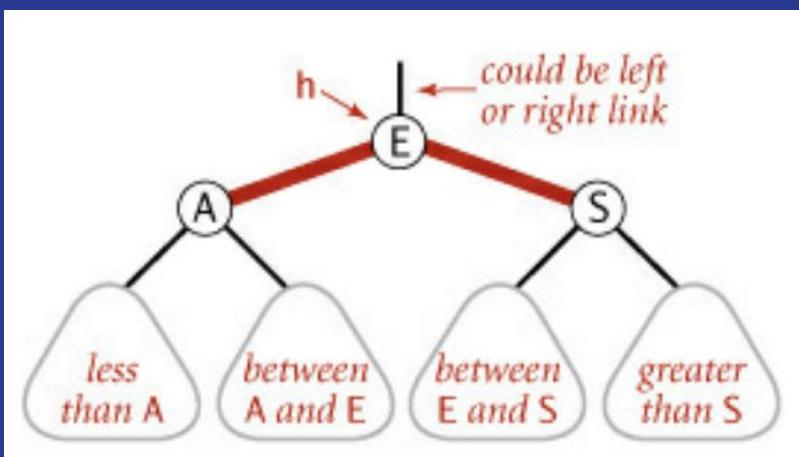


```
Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
          + size(h.right);
    return x;
}
```

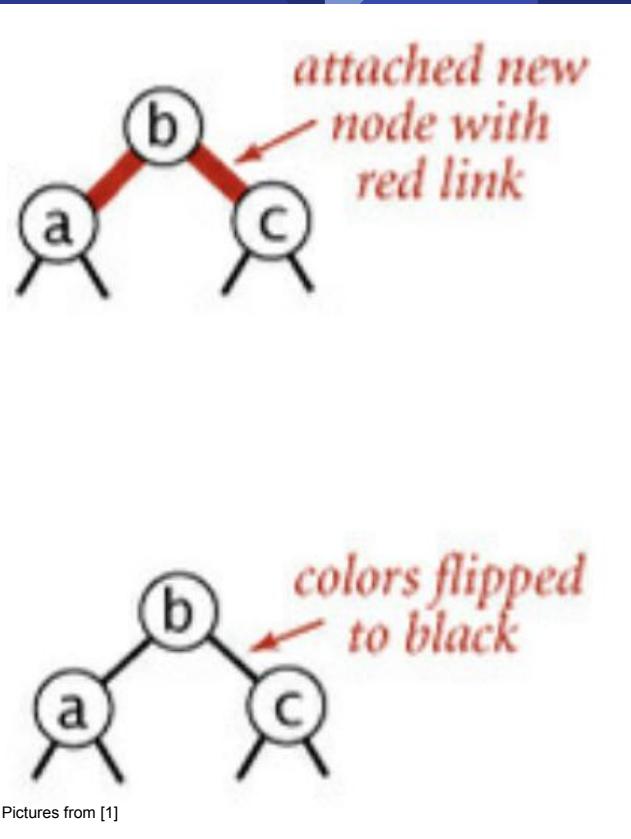


Right rotate (left link of h)

Color Flip

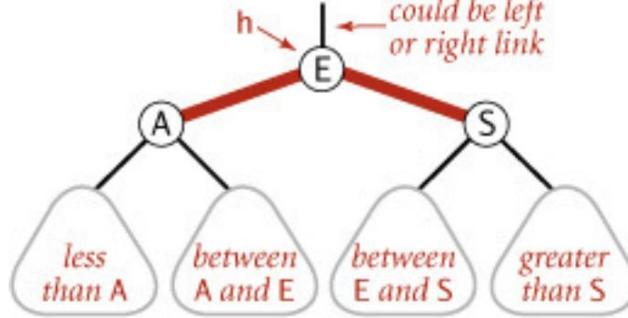


Color Flip at the root

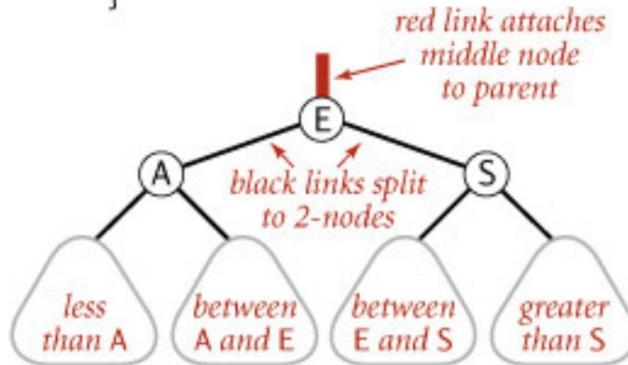


Color Flip Code

Pictures from [1]

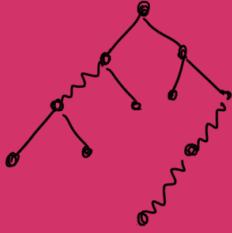


```
void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```



Flipping colors to split a 4-node

Operations

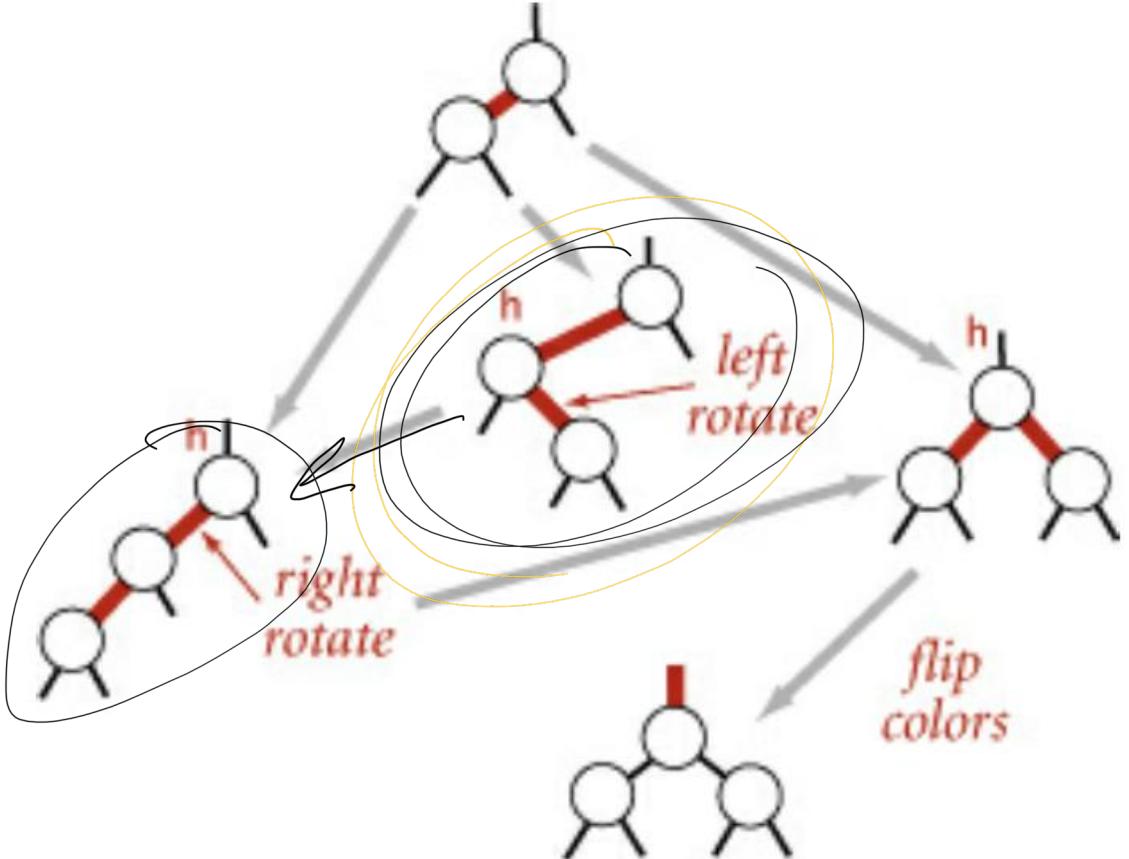


put

- New nodes are always inserted at the bottom of the tree.
- New nodes are always red.
- Transformations are used to re-balance the black height of the tree by ensuring the properties discussed previously.
- Transformations may pass red links up the tree just like splitting 4-nodes.



Fixing the tree...



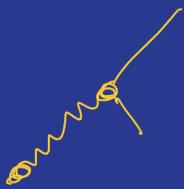
Passing a red link up a red-black BST

3 main operations to maintain balance:

- rotate left
- rotate right
- color flip

Insertion uses those three operations as you “pass the red link up the tree”.

Case 1: $X.left$ is red, $X.right$ is null or black.



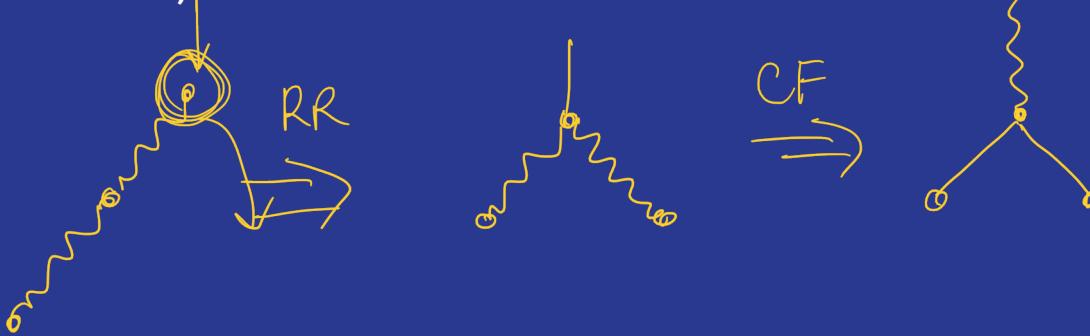
Case 2: $X.right$ is red, $X.left$ is null or black.



Case 3: $X.left$ is red, $X.right$ is red. (Note that X itself will always be black in this case.)



Case 4: $X.left$ is red, $X.left.left$ is red.



$X.left.right$

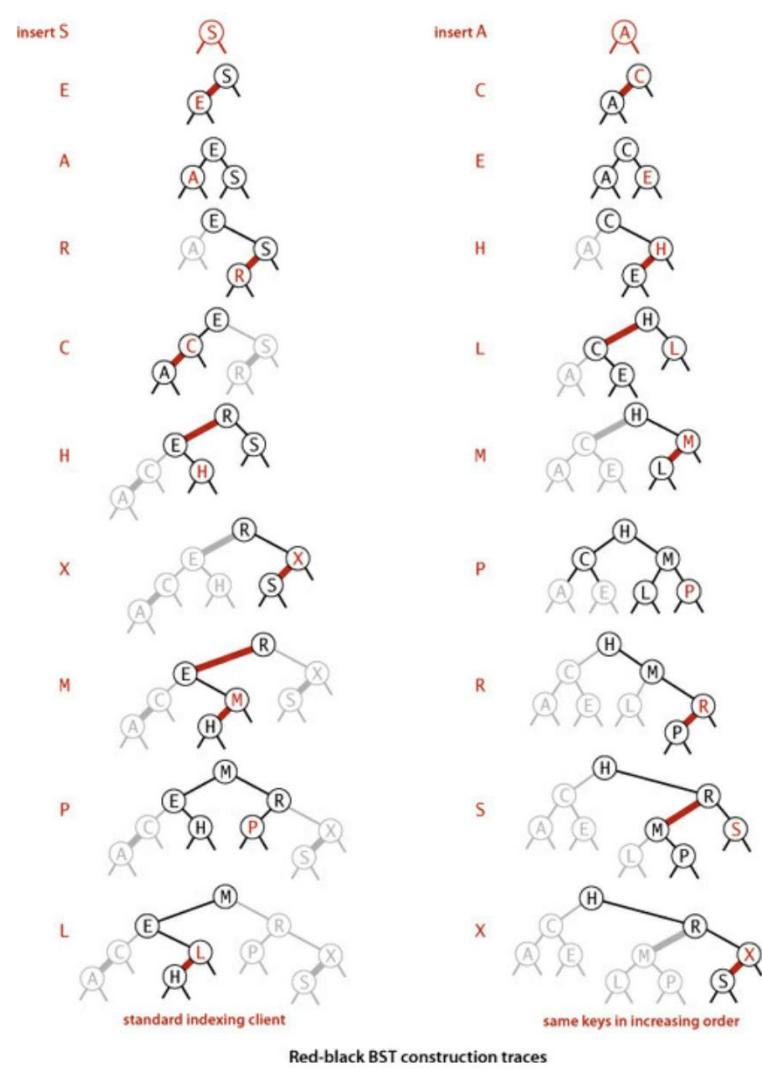
Case 5: $X.left$ is red, $X.left.$ is red.



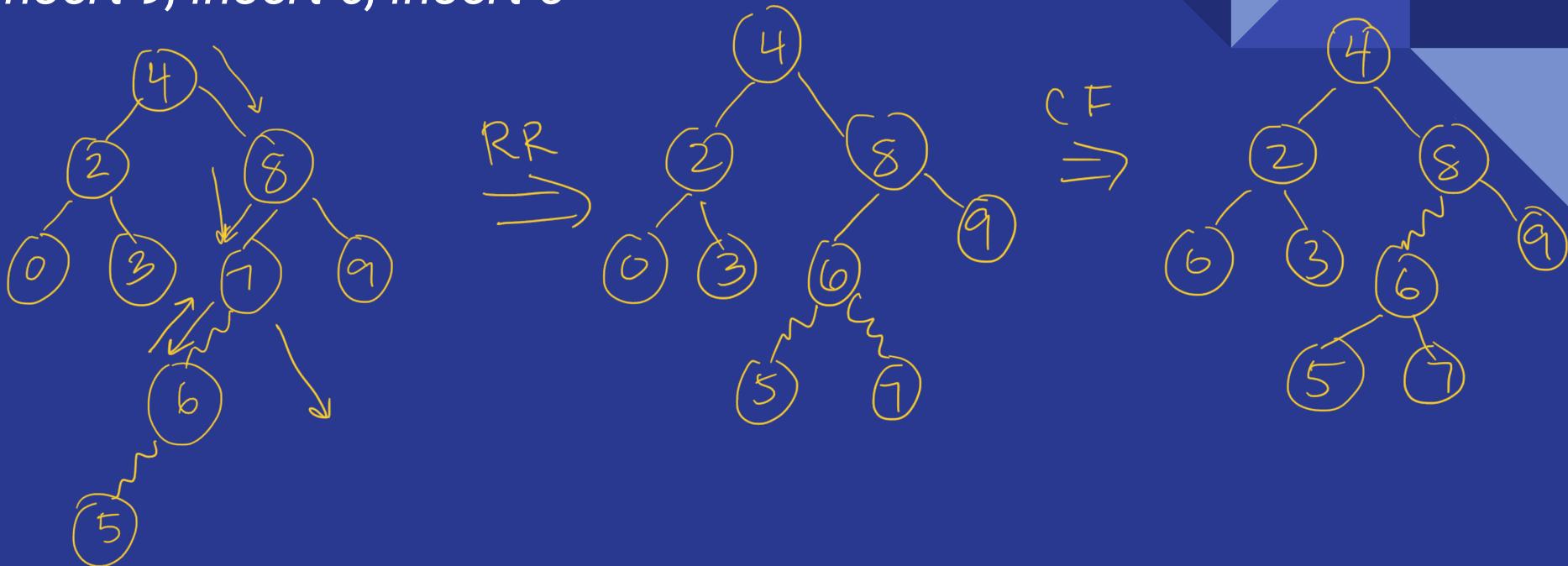
This case is only possible with *delete*.

Case 5: $X.right$ is red, $X.right.left$ is red.





Example: insert 0, insert 4, insert 7, insert 2, insert 3, insert 8, insert 9, insert 6, insert 5





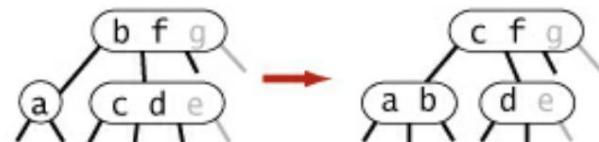
delete

- It's easier to delete a Red node than a black node.
- Key Idea: Use the inverses of the transformations discussed previously to carry a red link down the path--just in case.
- How is this related to deleting from a 2-3 tree?

at the root



on the way down



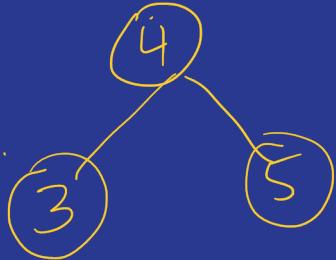
at the bottom



Transformations for delete the minimum

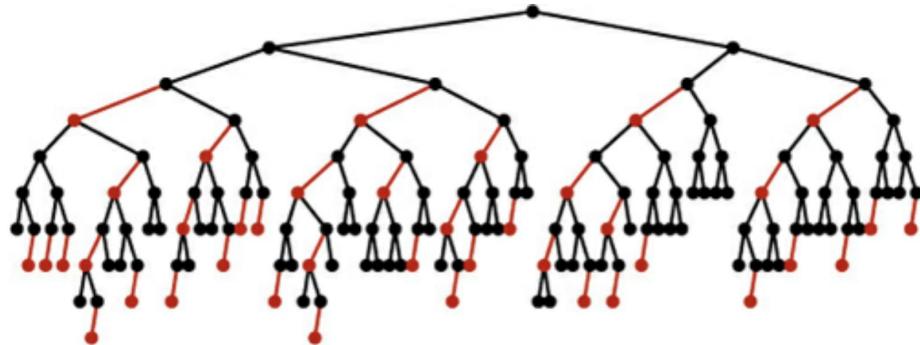
Picture from [1]

Example: delete 1, delete 6, delete 2, delete 7, delete 4, delete 3,
delete 5



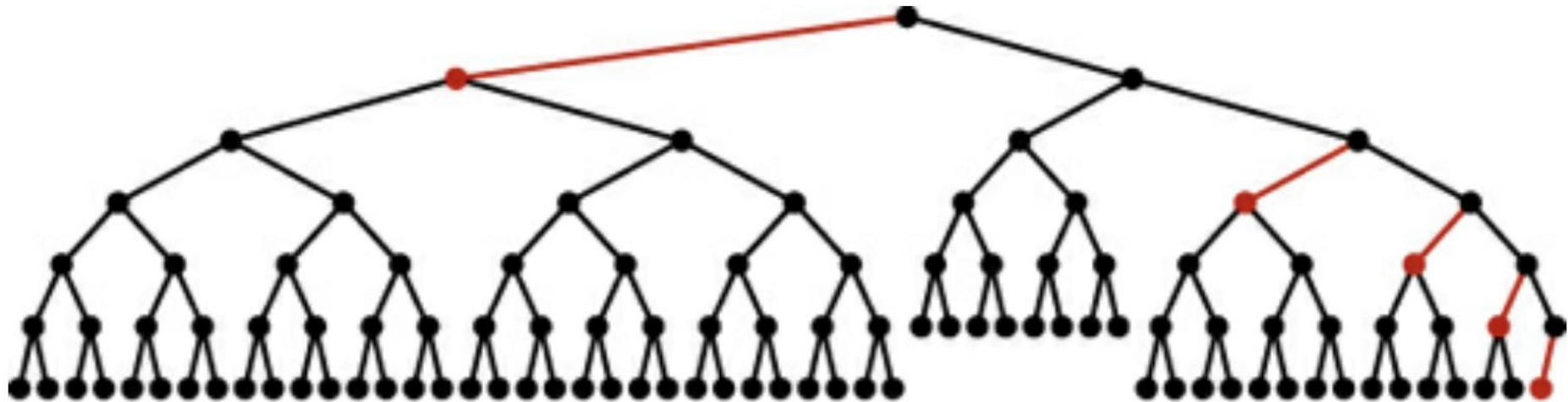
Properties of Red Black Trees

- Near perfect balance
- 1-1 correspondence with 2-3 Trees
- Height is no more than $\frac{\sim 2 \log N}{}$.
- Average length of root to a node is $\sim 1.00 \lg N$



Typical red-black BST built from random keys (null links omitted)

Picture from [1]



Red-black BST built from ascending keys (null links omitted)

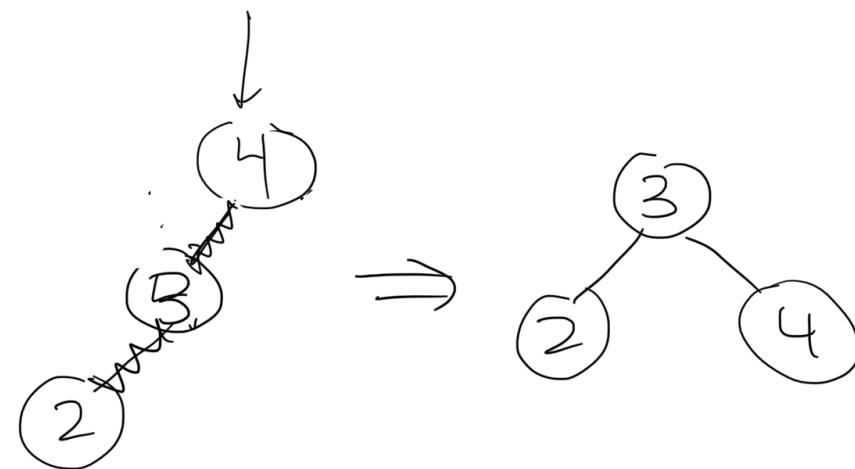
Proposition I. In a red-black BST, the following operations take logarithmic time in the worst case: search, insertion, finding the minimum, finding the maximum, floor, ceiling, rank, select, delete the minimum, delete the maximum, delete, and range count.

Picture from [1]

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
<i>sequential search (unordered linked list)</i>	N	N	N/2	N	no
<i>binary search (ordered array)</i>	$\lg N$	<u>N</u>	$\lg N$	<u>N/2</u>	yes
<i>binary tree search (BST)</i>	<u>N</u>	<u>N</u>	<u>$1.39 \lg N$</u>	<u>$1.39 \lg N$</u>	yes
<i>2-3 tree search (red-black BST)</i>	<u>$2 \lg N$</u>	<u>$2 \lg N$</u>	<u>$1.00 \lg N$</u>	<u>$1.00 \lg N$</u>	yes

Cost summary for symbol-table implementations (updated)

put 7, put 8, put 3, put 4, put 2,
delete 7



References

- [1] *Algorithms, Fourth Edition*; Robert Sedgewick and Kevin Wayne (and associated slides)
- [2] Slides from <https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>