

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

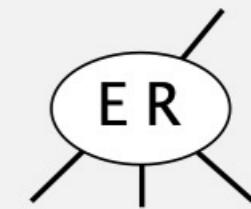
<https://algs4.cs.princeton.edu>

3.3 BALANCED SEARCH TREES

- ▶ *2-3 search trees*
- ▶ ***red-black BSTs***
- ▶ *B-trees*

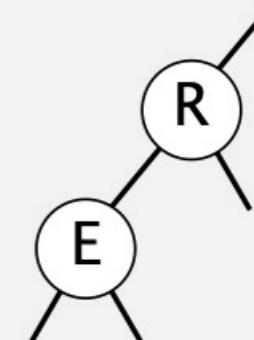
How to implement 2–3 trees with binary trees?

Challenge. How to represent a 3 node?



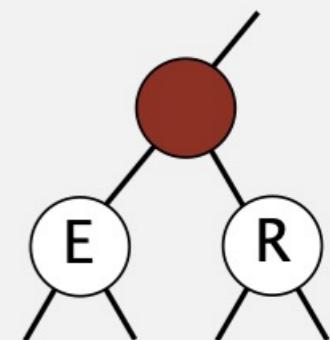
Approach 1. Regular BST.

- No way to tell a 3-node from two 2-nodes.
- Can't (uniquely) map from BST back to 2–3 tree.



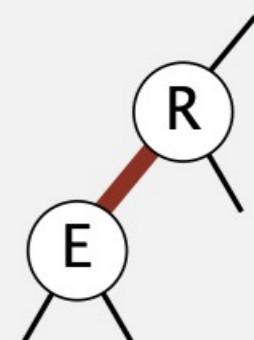
Approach 2. Regular BST with red “glue” nodes.

- Wastes space for extra node.
- Messy code.



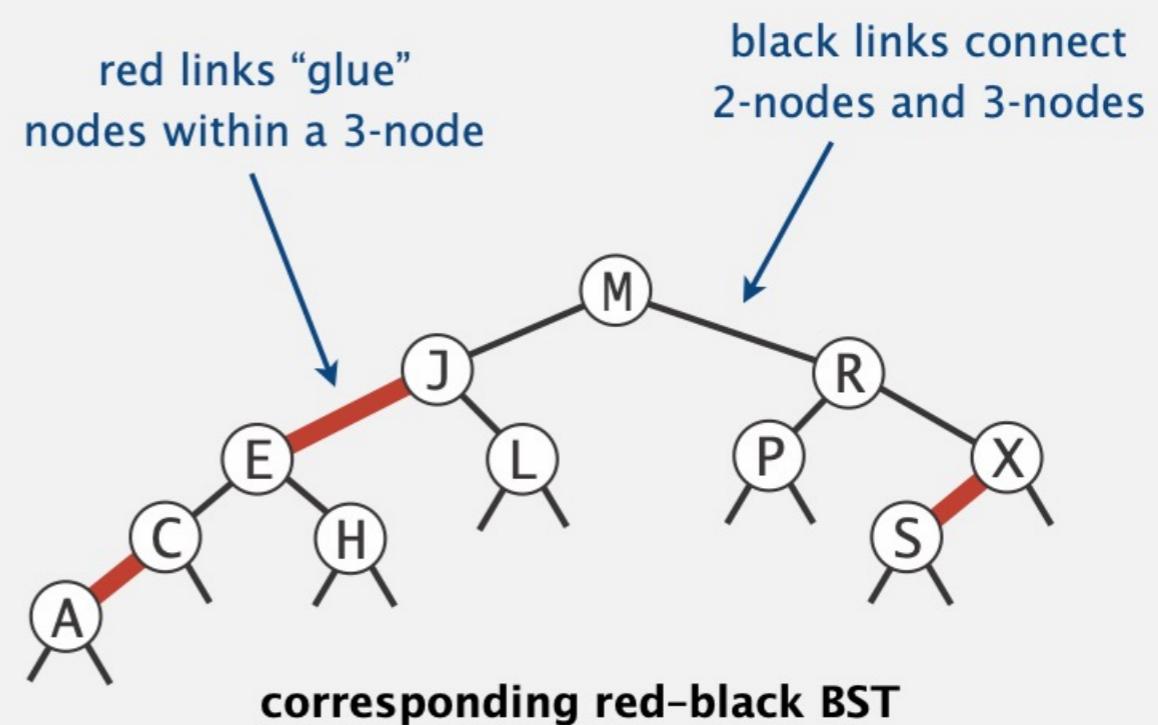
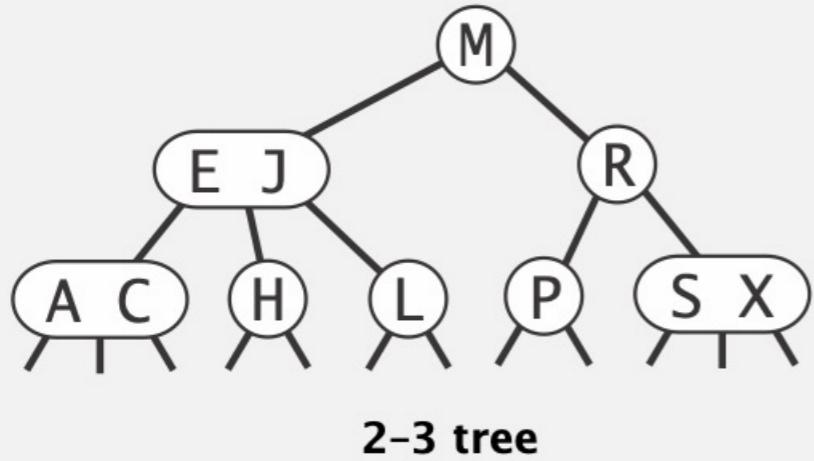
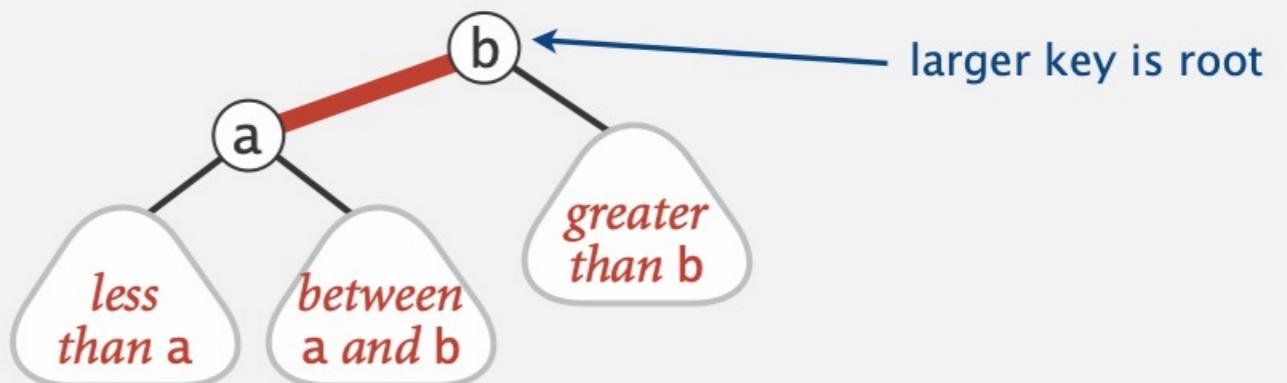
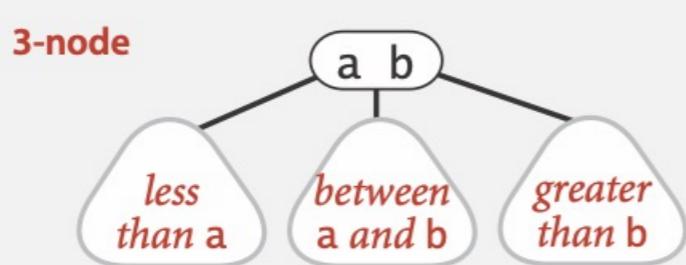
Approach 3. Regular BST with red “glue” links.

- Widely used in practice.
- Arbitrary restriction: red links lean left.



Left-leaning red-black BSTs (Guibas–Sedgewick 1979 and Sedgewick 2007)

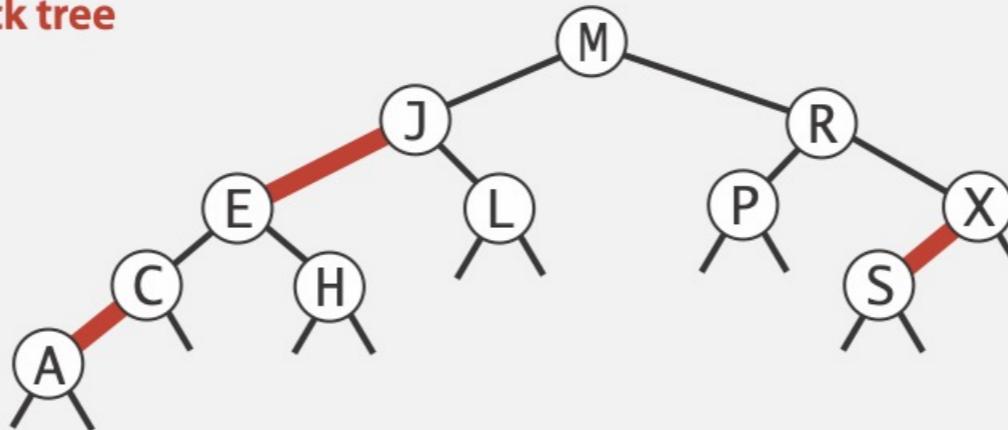
1. Represent 2–3 tree as a BST.
2. Use “internal” left-leaning links as “glue” for 3–nodes.



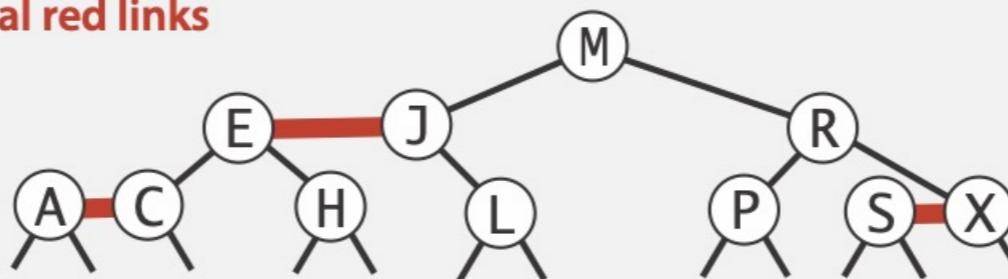
Left-leaning red-black BSTs: 1–1 correspondence with 2–3 trees

Key property. 1–1 correspondence between 2–3 trees and LLRB trees.

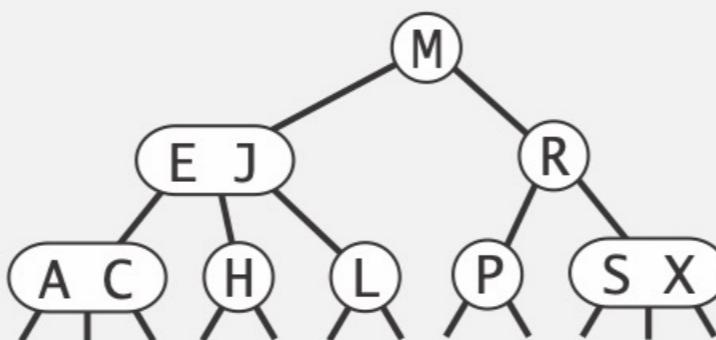
red-black tree



horizontal red links

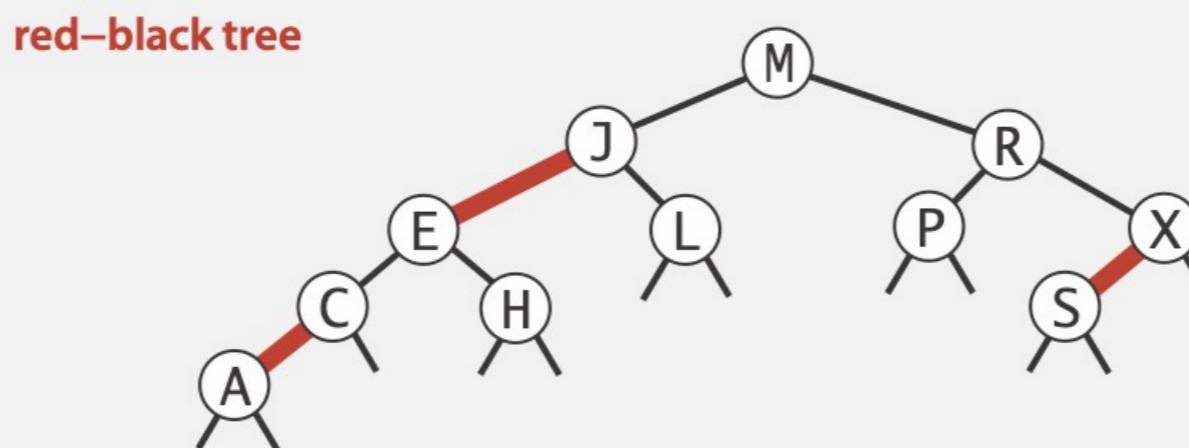


2-3 tree



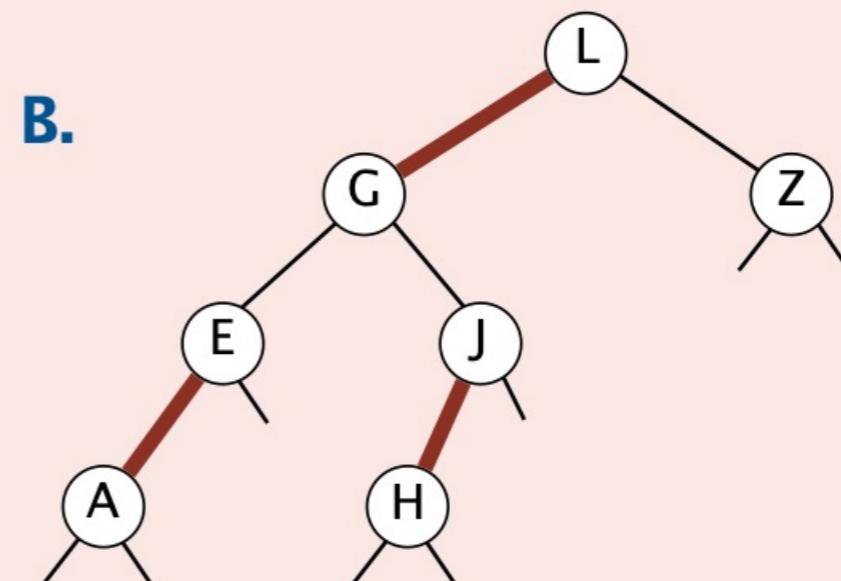
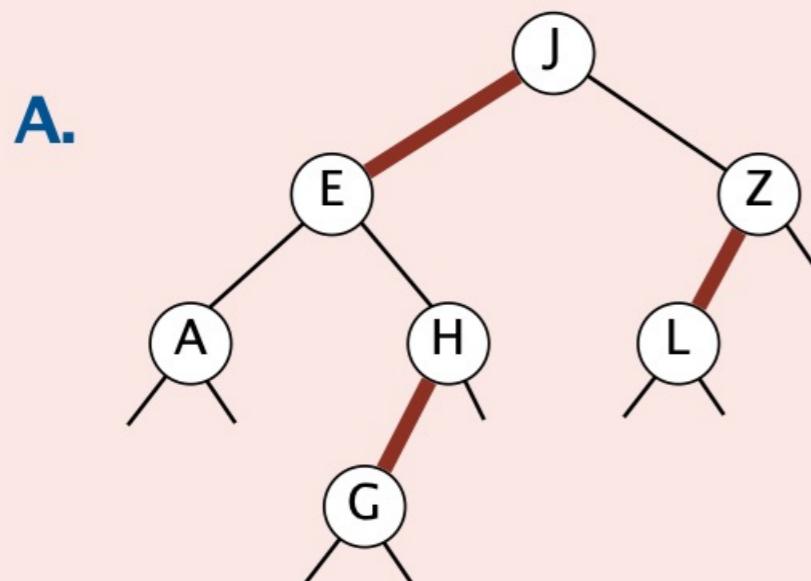
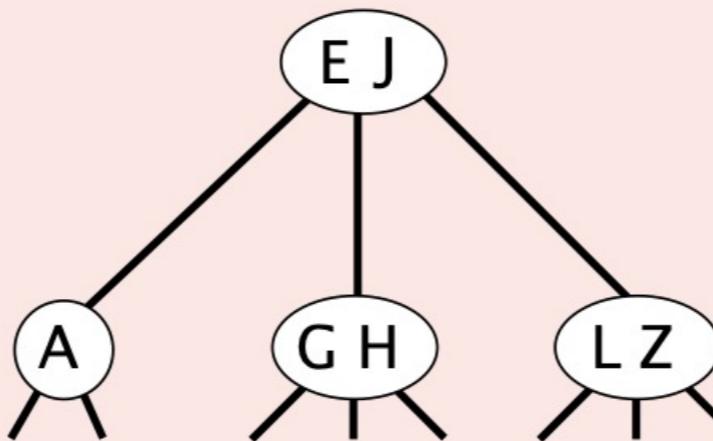
An equivalent definition of LLRB trees (without reference to 2–3 trees)

- A BST such that:
- No node has two red links connected to it.
 - Red links lean left.
 - Every path from root to null link has the same number of black links.
- ← color invariants
“perfect black balance”





Which LLRB tree corresponds to the following 2-3 tree?



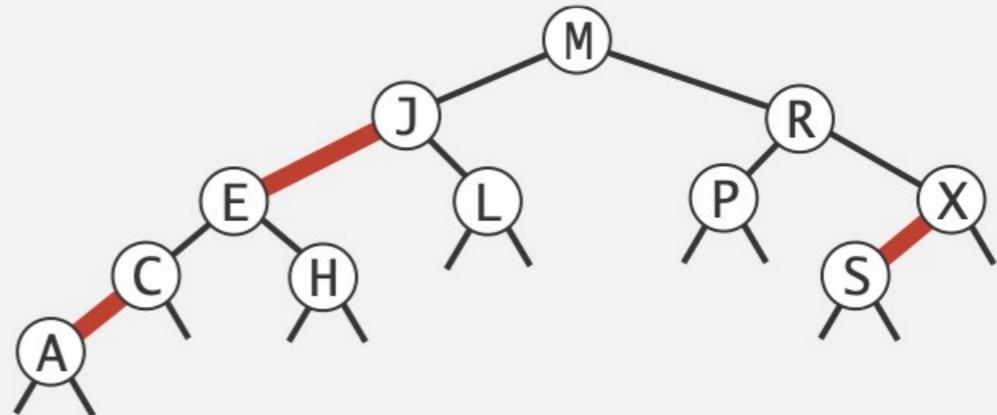
- C. Both A and B.
- D. Neither A nor B.

Search implementation for red-black BSTs

Observation. Search is the same as for BST (ignore color).

but runs faster
(because of better balance)

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```



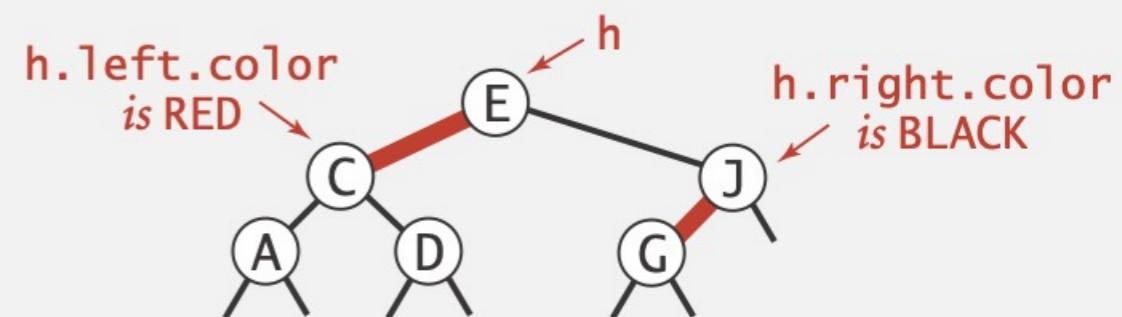
Remark. Many other ops (floor, iteration, rank, selection) are also identical.

Red-black BST representation

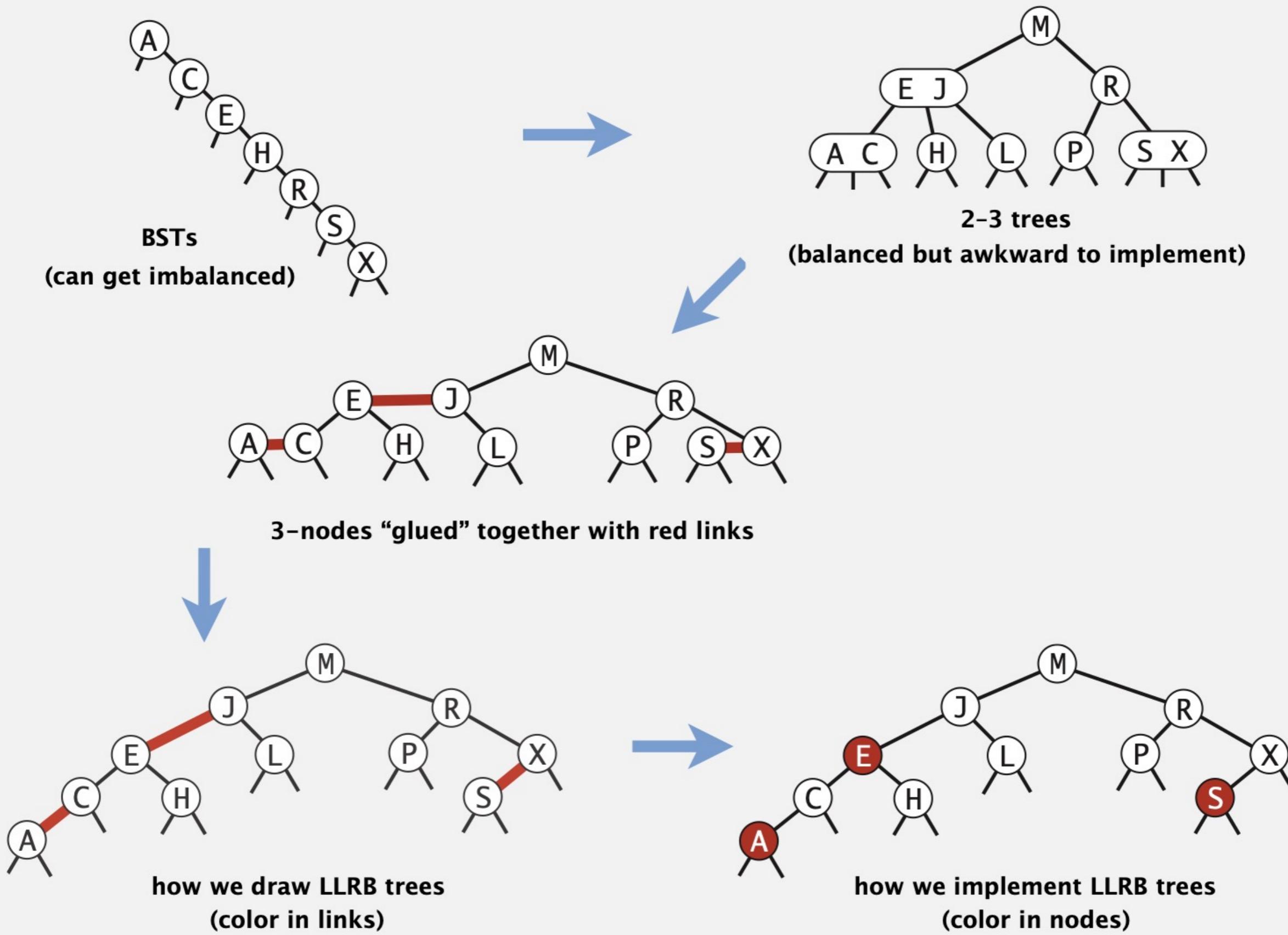
Each node is pointed to by precisely one link (from its parent) \Rightarrow
can encode color of links in nodes.

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

```
private class Node
{
    Key key;
    Value val;
    Node left, right;
    boolean color; // color of parent link
}
```



Review: the road to LLRB trees



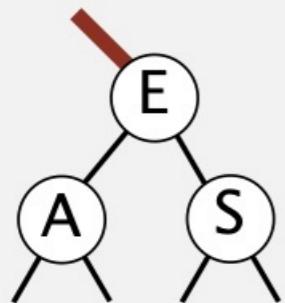
Insertion into a LLRB tree: overview

Basic strategy. Maintain 1–1 correspondence with 2–3 trees.

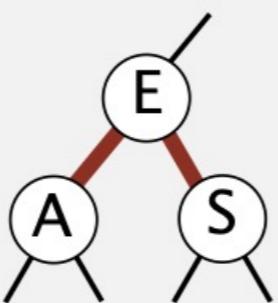
During internal operations, maintain:

- Symmetric order.
- Perfect black balance.
- [but not necessarily color invariants]

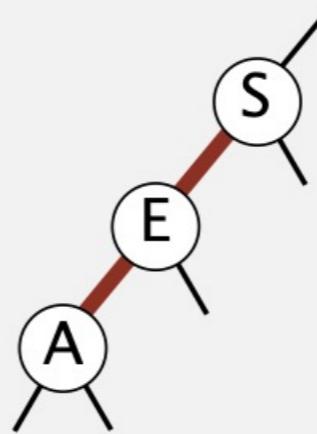
Example violations of color invariants:



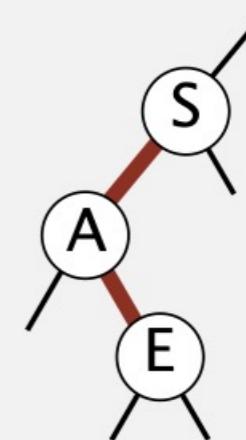
right-leaning
red link



two red children
(a temporary 4-node)



left-left red
(a temporary 4-node)



left-right red
(a temporary 4-node)

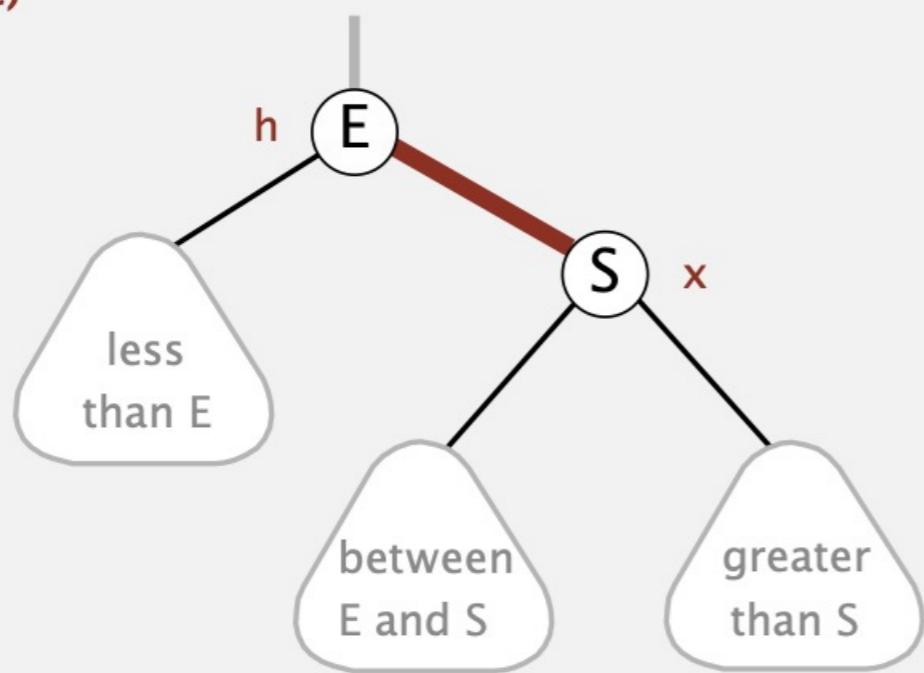
To restore color invariants: perform rotations and color flips.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left

(before)

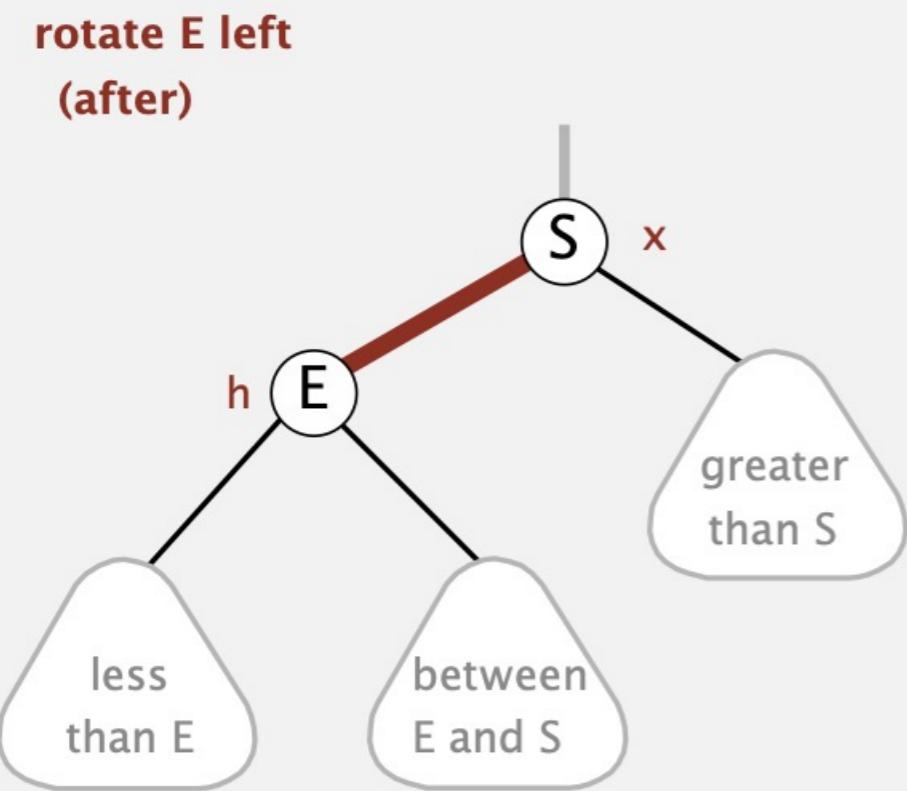


```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.



```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

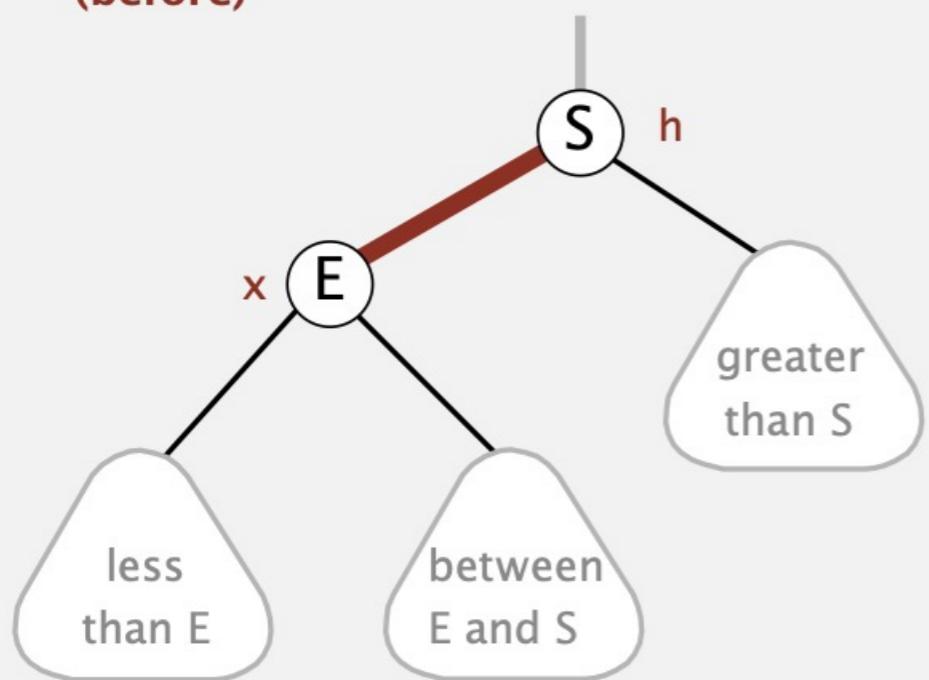
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right

(before)



```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

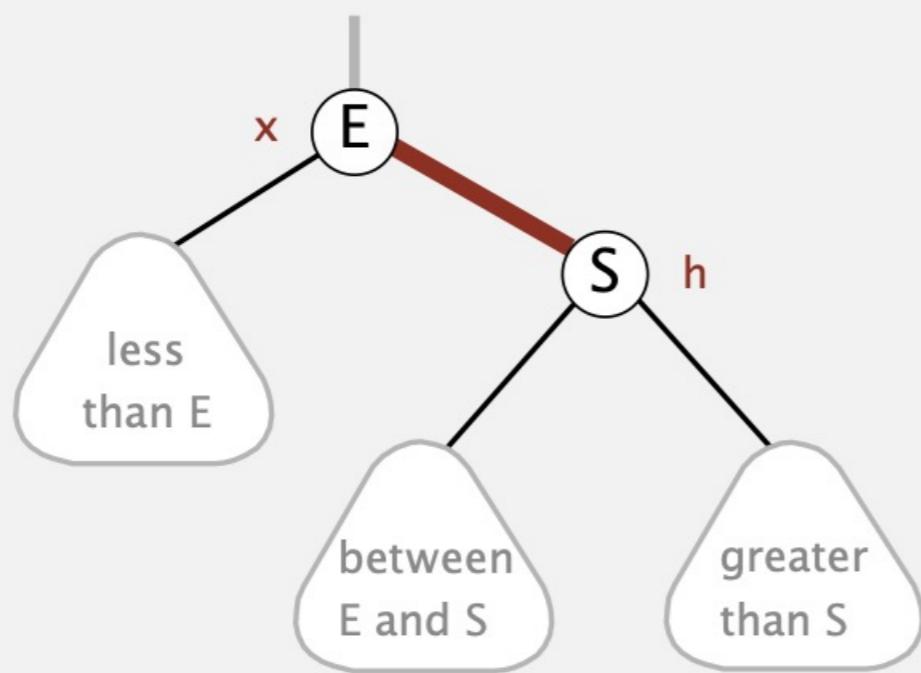
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right

(after)

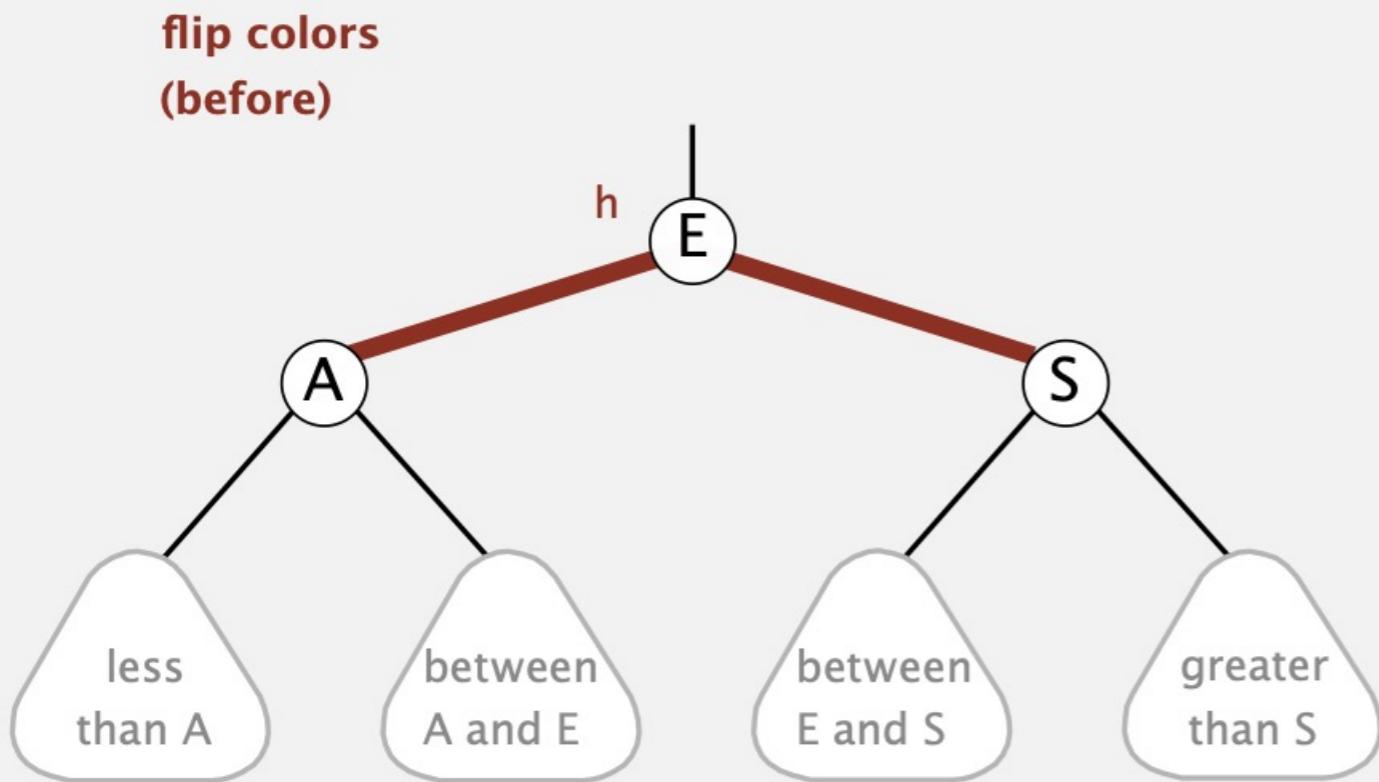


```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

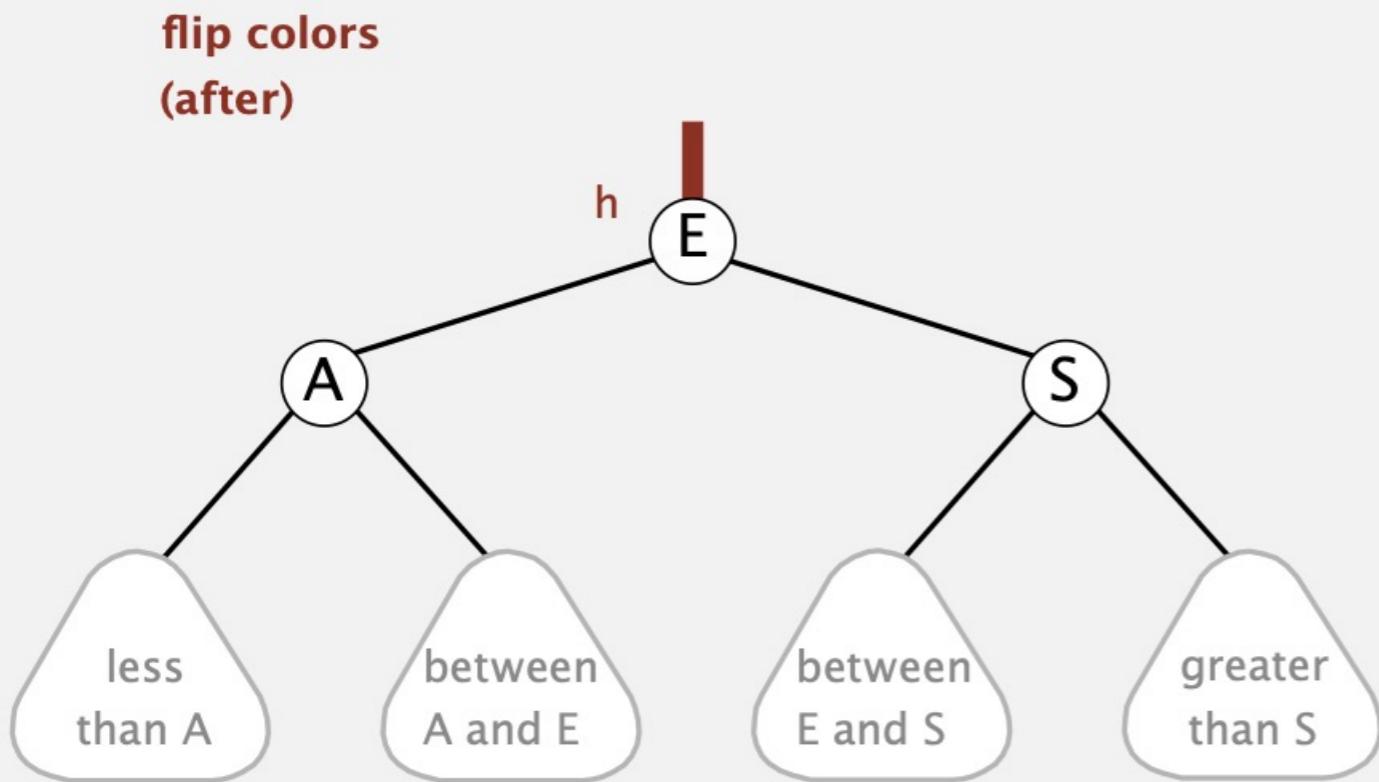


```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

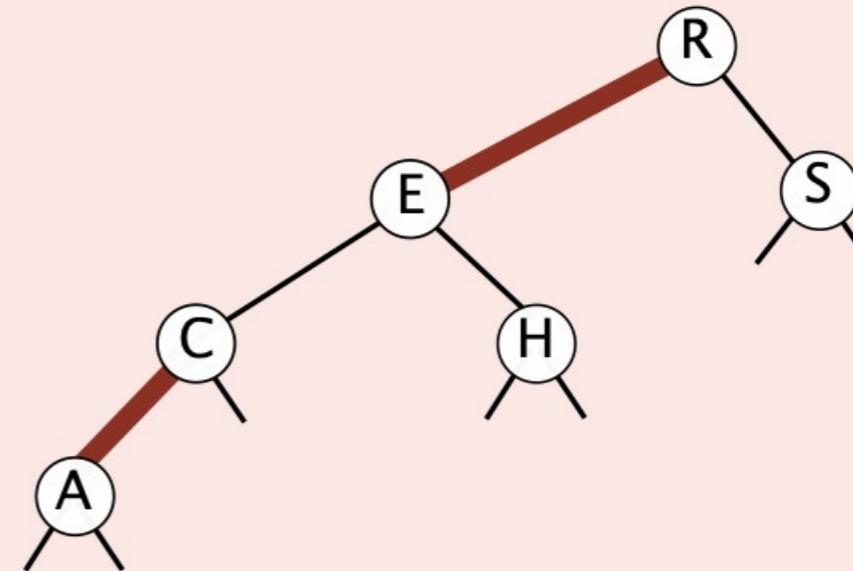
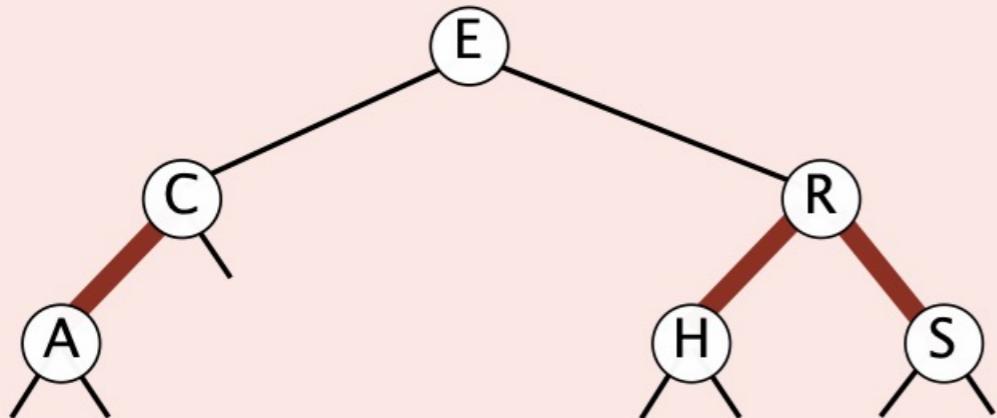


```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.



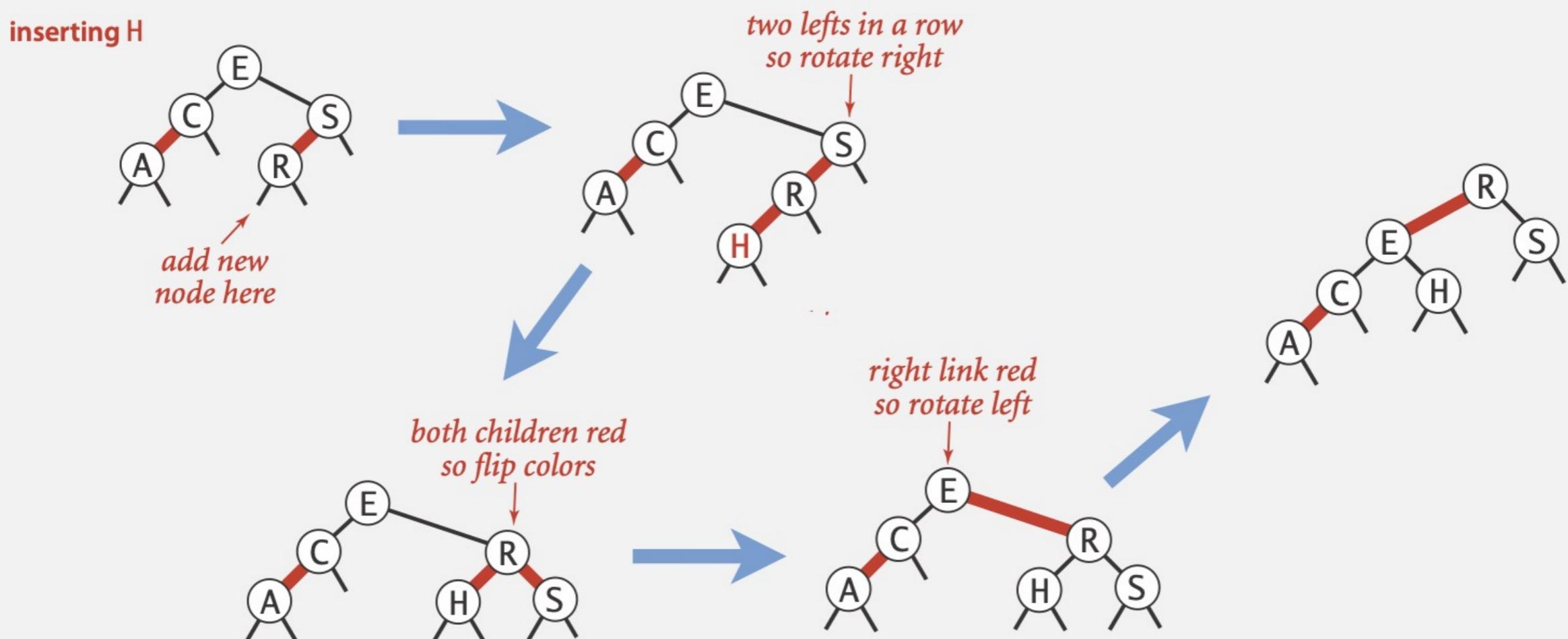
Which sequence of elementary operations transforms the red-black BST at left to the one at right?



- A. Color flip R; left rotate E.
- B. Color flip R; right rotate E.
- C. Color flip E; left rotate R.
- D. Color flip R; left rotate R.

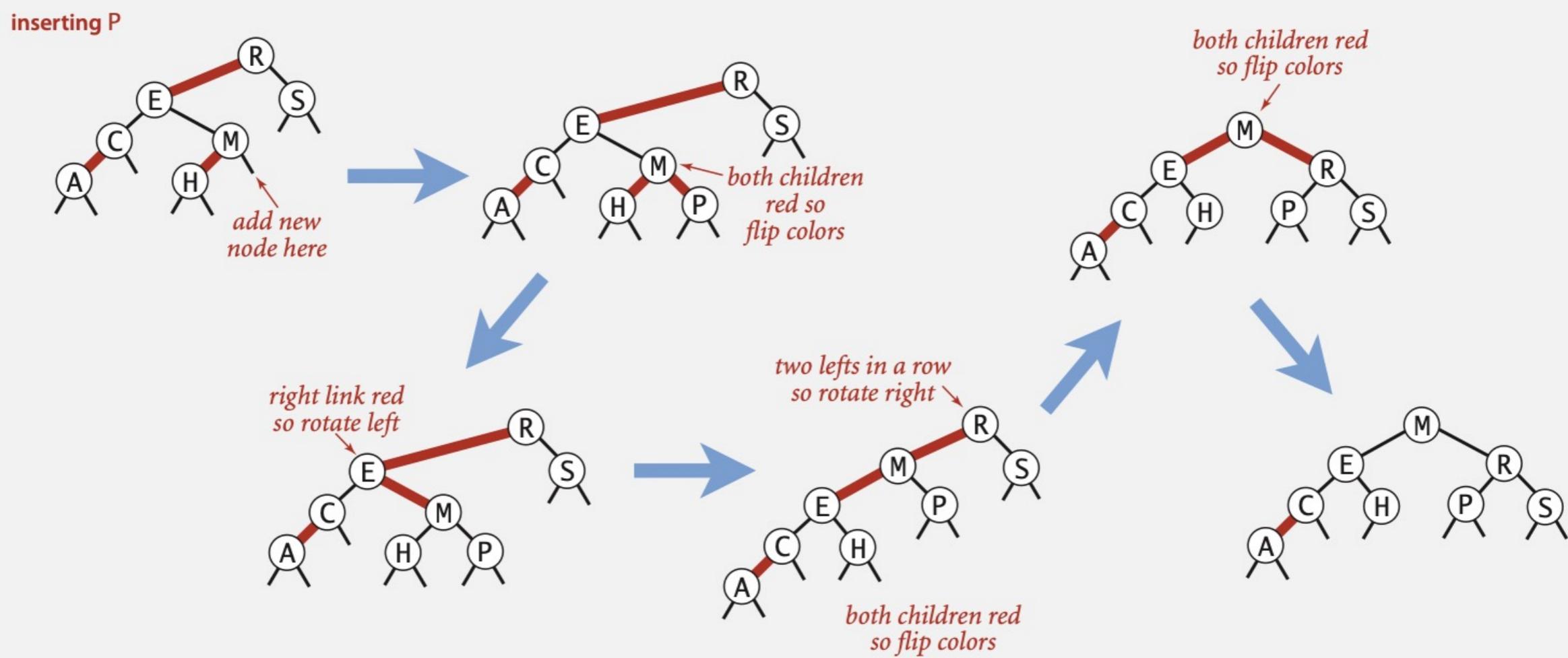
Insertion into a LLRB tree

- Do standard BST insert. ← to preserve symmetric order
- Color new link red. ← to preserve perfect black balance
- Repeat up the tree until color invariants restored:
 - two left red links in a row? ⇒ rotate right
 - left and right links both red? ⇒ color flip
 - right link only red? ⇒ rotate left



Insertion into a LLRB tree

- Do standard BST insert.
- Color new link red.
- Repeat up the tree until color invariants restored:
 - two left red links in a row? \Rightarrow rotate right
 - left and right links both red? \Rightarrow color flip
 - right link only red? \Rightarrow rotate left

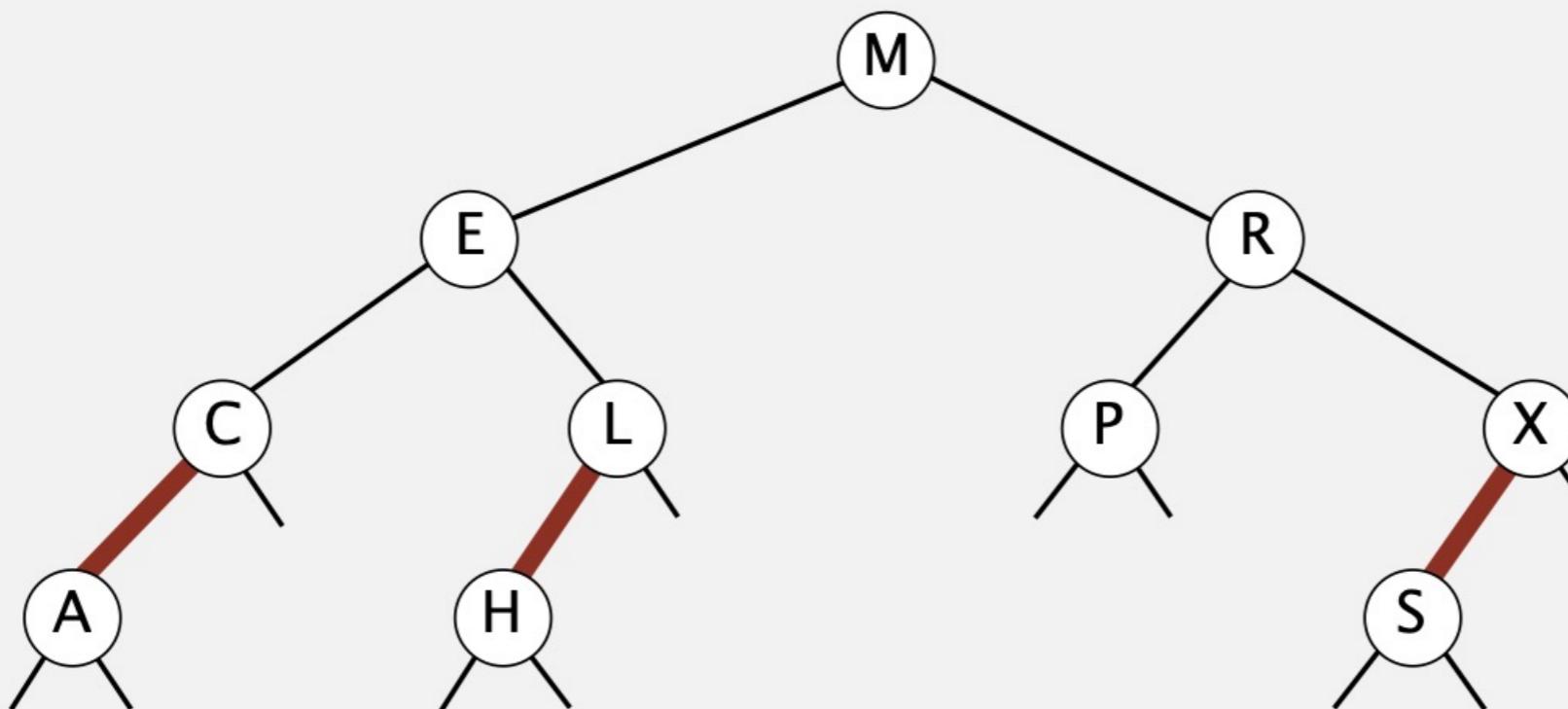


Construct a Red-Black Tree with the following keys:

S E A R C H X M P L

Red-black BST construction demo

insert S E A R C H X M P L



Insertion into a LLRB tree: Java implementation

- Do standard BST insert and color new link red.
- Repeat up the tree until color invariants restored:
 - right link only red? \Rightarrow rotate left
 - two left red links in a row? \Rightarrow rotate right
 - left and right links both red? \Rightarrow color flip

```
private Node put(Node h, Key key, Value val)
{
    if (h == null) return new Node(key, val, RED); ← insert at bottom  
(and color it red)
    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else if (cmp == 0) h.val  = val;

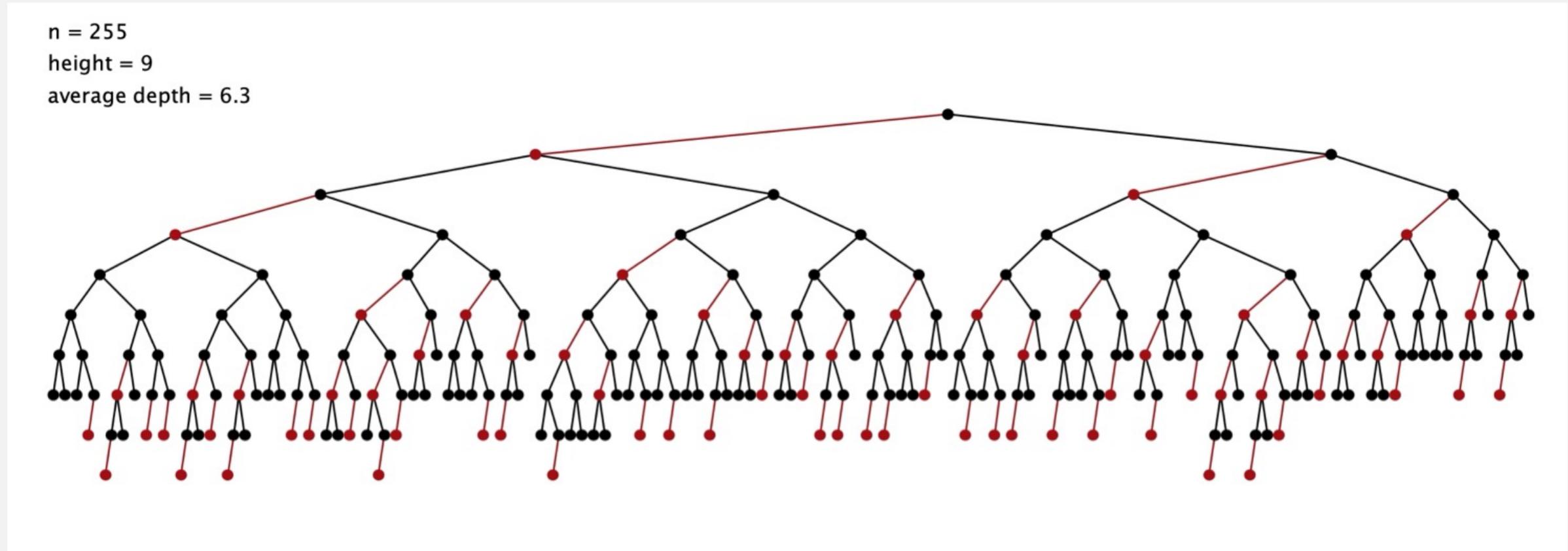
    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
    if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left)  && isRed(h.right))     flipColors(h);

    return h;
}
```

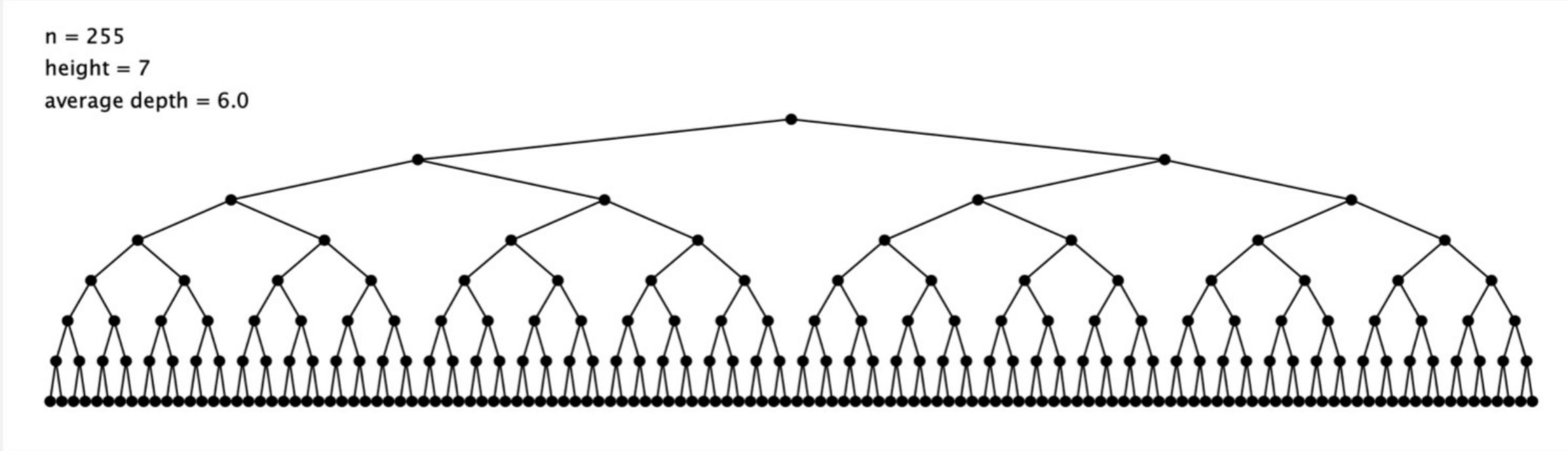
↑
only a few extra lines of code provides near-perfect balance

restore color
invariants ←

Insertion into a LLRB tree: visualization

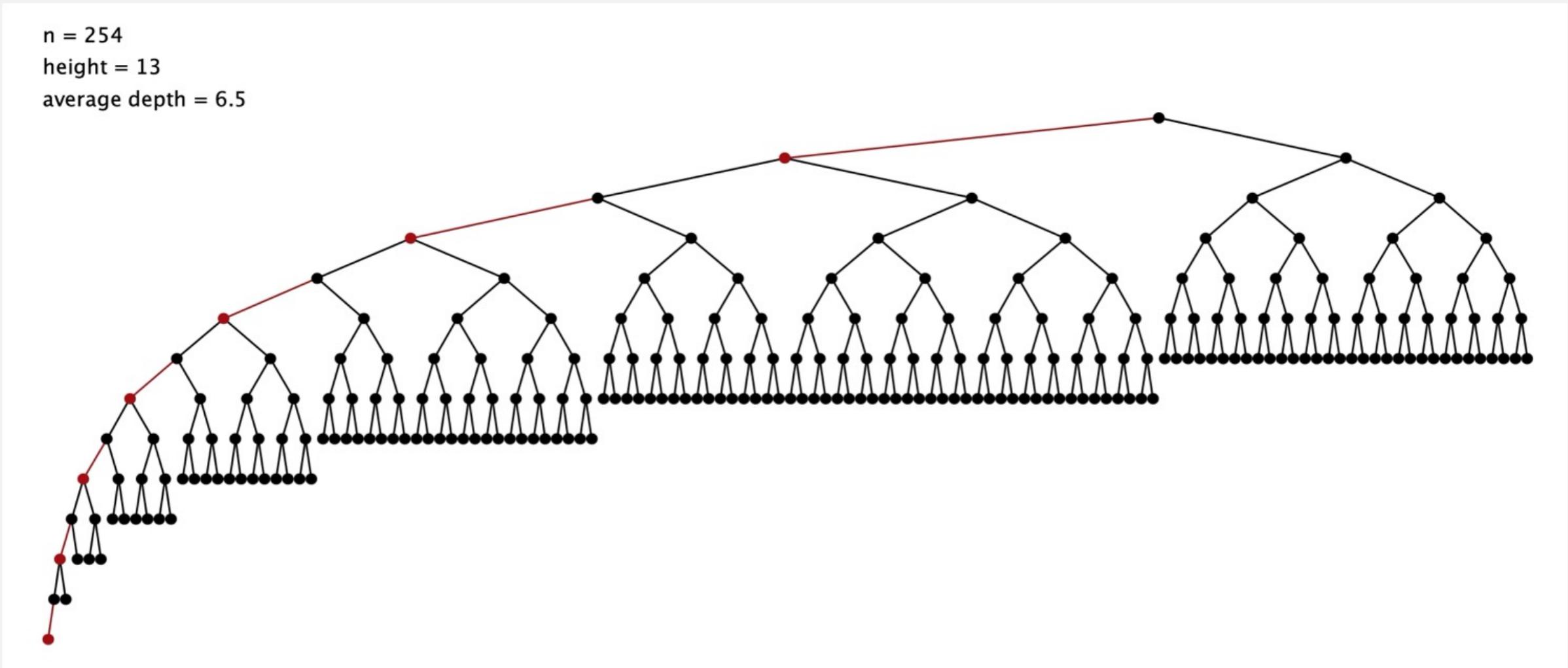


Insertion into a LLRB tree: visualization



255 insertions in ascending order

Insertion into a LLRB tree: visualization



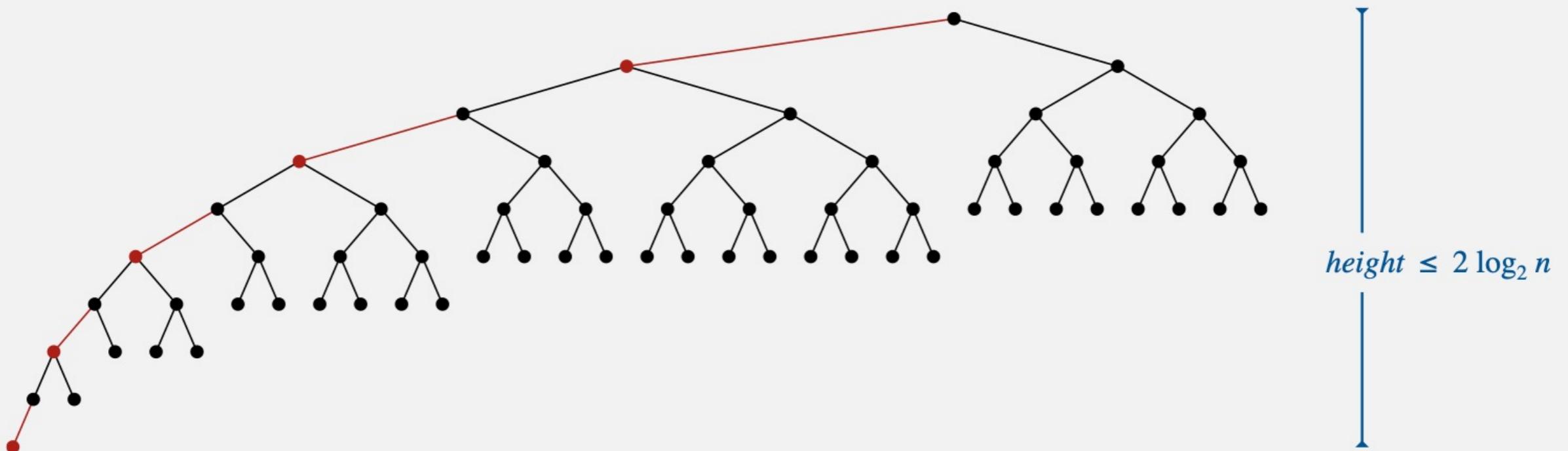
254 insertions in descending order

Balance in LLRB trees

Proposition. Height of LLRB tree is $\leq 2 \log_2 n$.

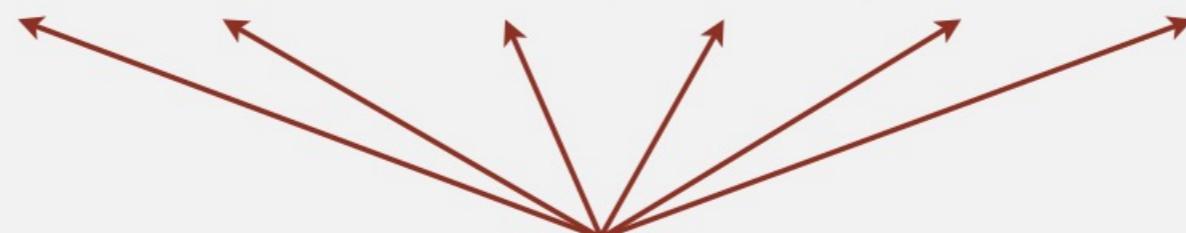
Pf.

- Black height = height of corresponding 2–3 tree $\leq \log_2 n$.
- Never two red links in-a-row.
 \Rightarrow height of LLRB tree $\leq (2 \times \text{black height}) + 1$
 $\leq 2 \log_2 n + 1$.
- [A slightly more refined arguments show height $\leq 2 \log_2 n$.]



ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search	insert	delete		
sequential search (unordered list)	n	n	n	n	n	n		equals()
binary search (ordered array)	$\log n$	n	n	$\log n$	n	n	✓	compareTo()
BST	n	n	n	$\log n$	$\log n$	\sqrt{n}	✓	compareTo()
2-3 tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✓	compareTo()
red-black BST	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✓	compareTo()



hidden constant c is small
(at most $2 \log_2 n$ compares)

Why named red-black BSTs?

Xerox PARC innovations. [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- Laser printing.
- Bitmapped display.
- WYSIWYG text editor.
- ...



Xerox Alto

A DICHROMATIC FRAMEWORK FOR BALANCED TREES

Leo J. Guibas
*Xerox Palo Alto Research Center,
Palo Alto, California, and
Carnegie-Mellon University*

and
Robert Sedgewick*
*Program in Computer Science
Brown University
Providence, R. I.*

ABSTRACT

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this

the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a number of variations on a common theme and exhibit full implementations which are notable for their brevity. One implementation is examined carefully, and some properties about its

Balanced trees in the wild

Red-black BSTs are widely used as system symbol tables.

- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: `map`, `multimap`, `multiset`.
- Linux kernel: CFQ I/O scheduler, `linux/rbtree.h`.

Other balanced BSTs. AVL trees, splay trees, randomized BSTs,

B-trees (and cousins) are widely used for file systems and databases.

- Windows: NTFS.
- Mac OS X: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS, BTRFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.



Mac

