

Balanced Search Trees

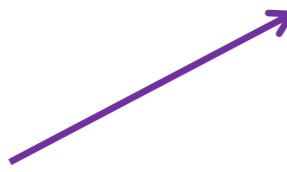
Hash Tables

Dr. 何明晰, He Mingxin, Max

program06 @ yeah.net

Email Subject: (L1-|L2-|L3-) + *last 4 digits of ID* + *Name: TOPIC*

Your Lab Class



Sakai: CS203B Fall 2022

数据结构与算法分析B

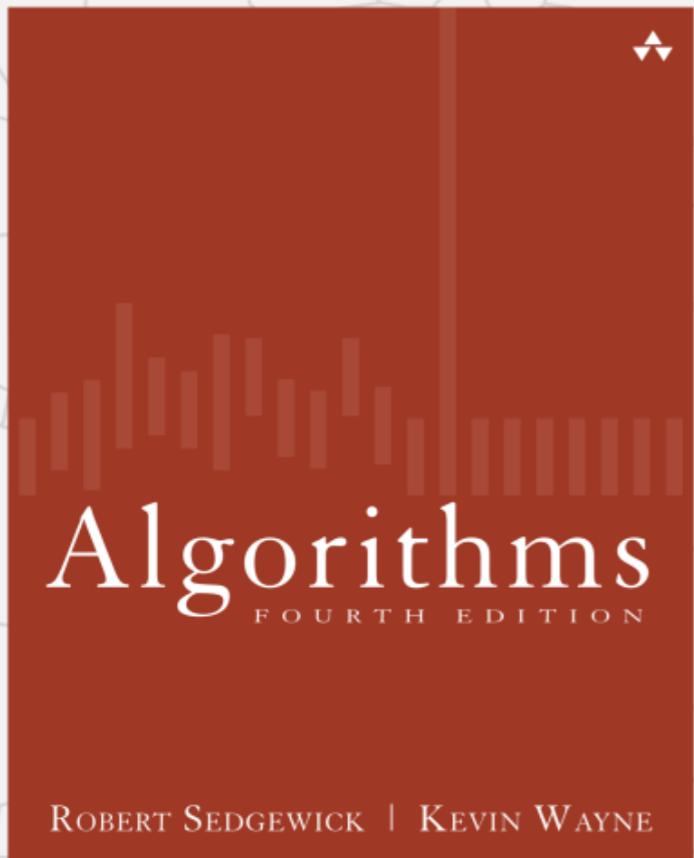
Data Structures and Algorithm Analysis

Lecture 11

- Binary Search Trees (3.2 of Text A)
- Balanced Search Trees (3.3 of Text A)
- Hash Tables (3.4 of Text A)

To be discussed in Lecture 12:

- Hash Tables (3.4 of Text A)
- Undirected Graphs (4.1 of Text A)



ROBERT SEDGEWICK | KEVIN WAYNE

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

3.3 BALANCED SEARCH TREES

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

Symbol table review

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✓	<code>compareTo()</code>

Challenge. Guarantee performance.

This lecture. 2-3 trees, left-leaning red-black BSTs, B-trees.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.3 BALANCED SEARCH TREES

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

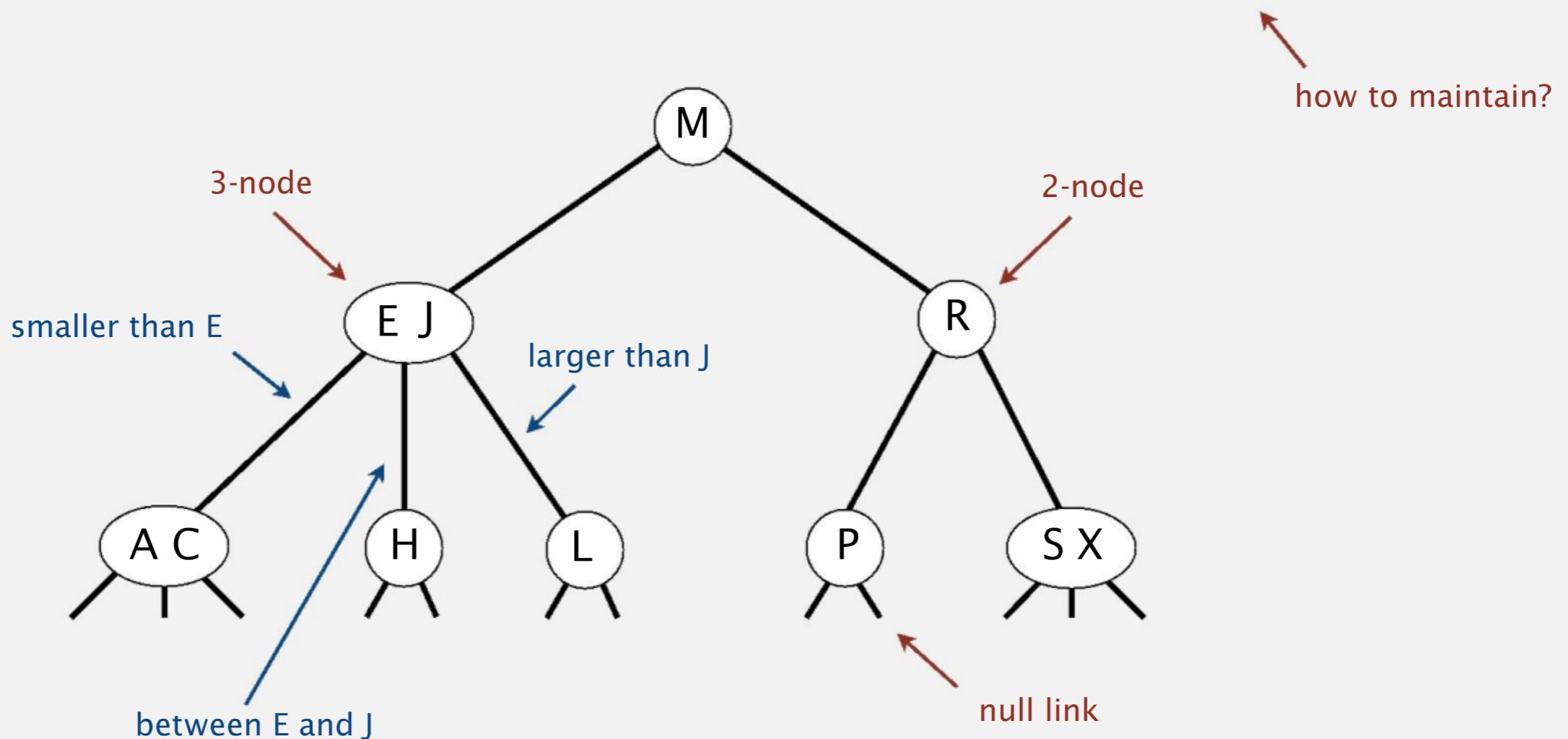
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.



2-3 tree demo

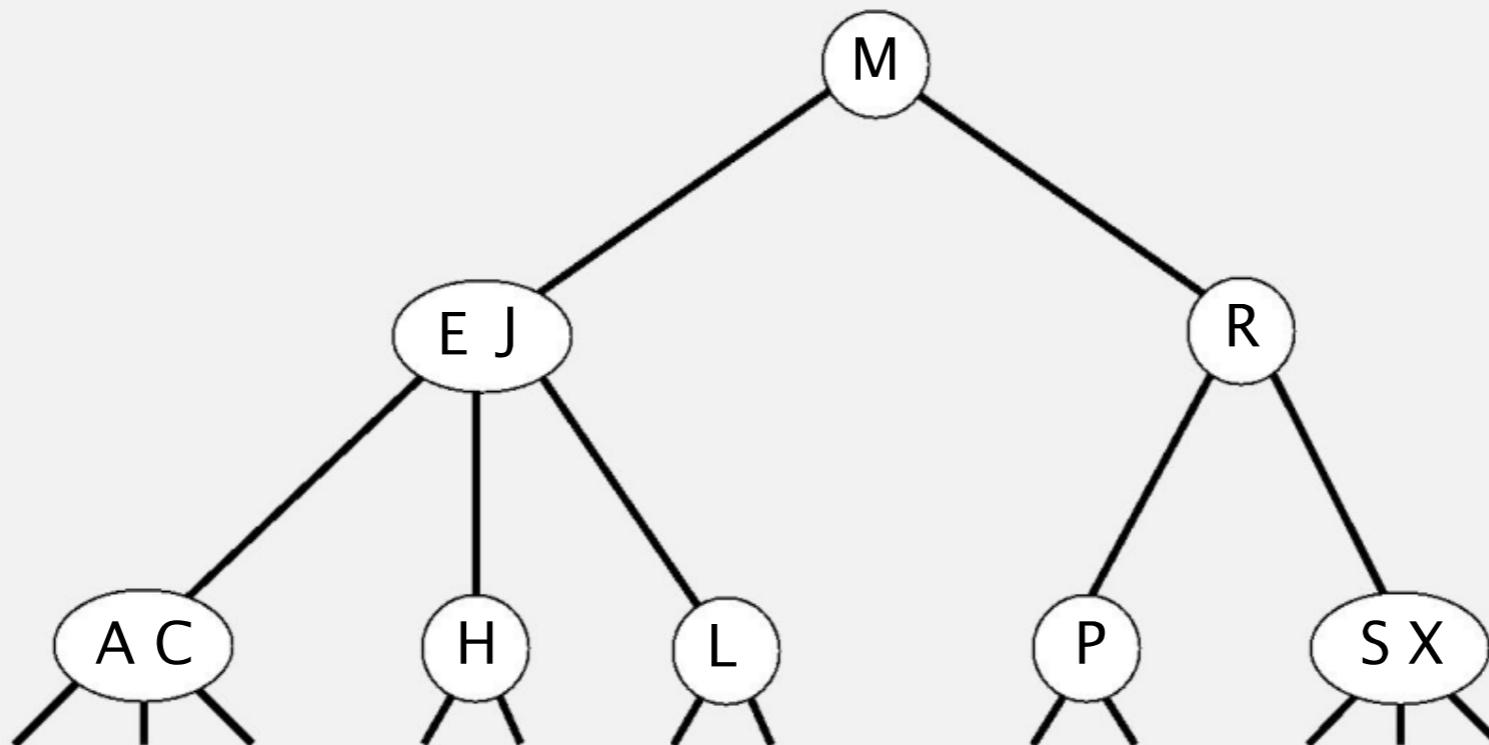
Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).



search for H

<https://algs4.cs.princeton.edu/lectures/demo/33Demo23Tree.mov>

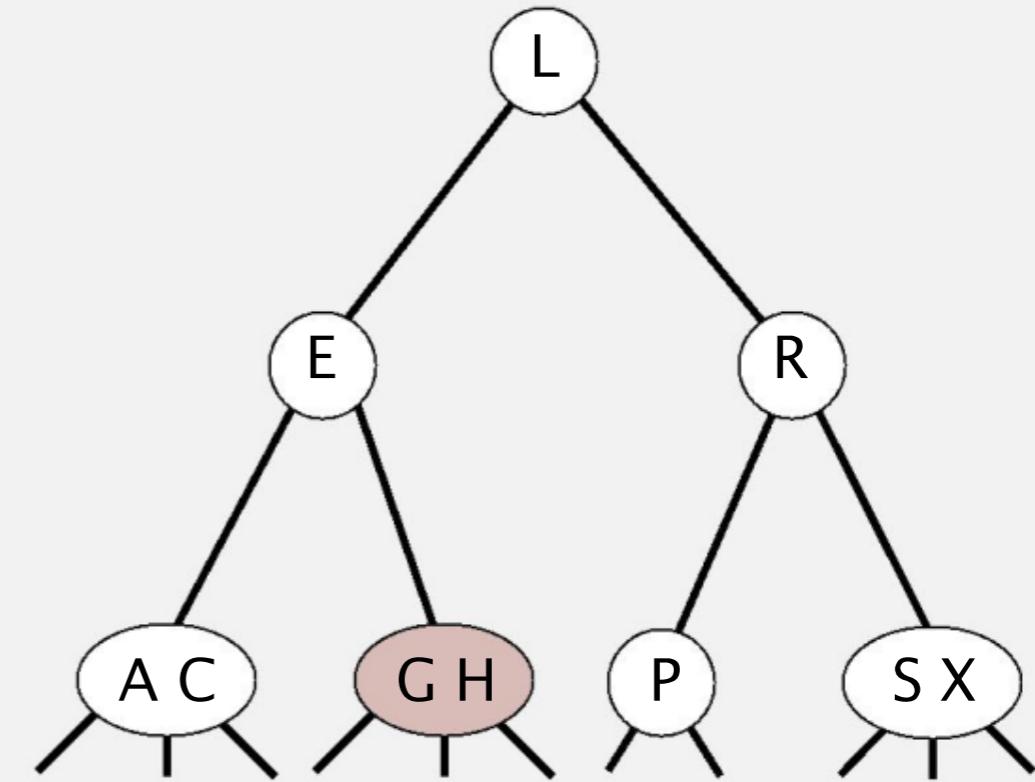
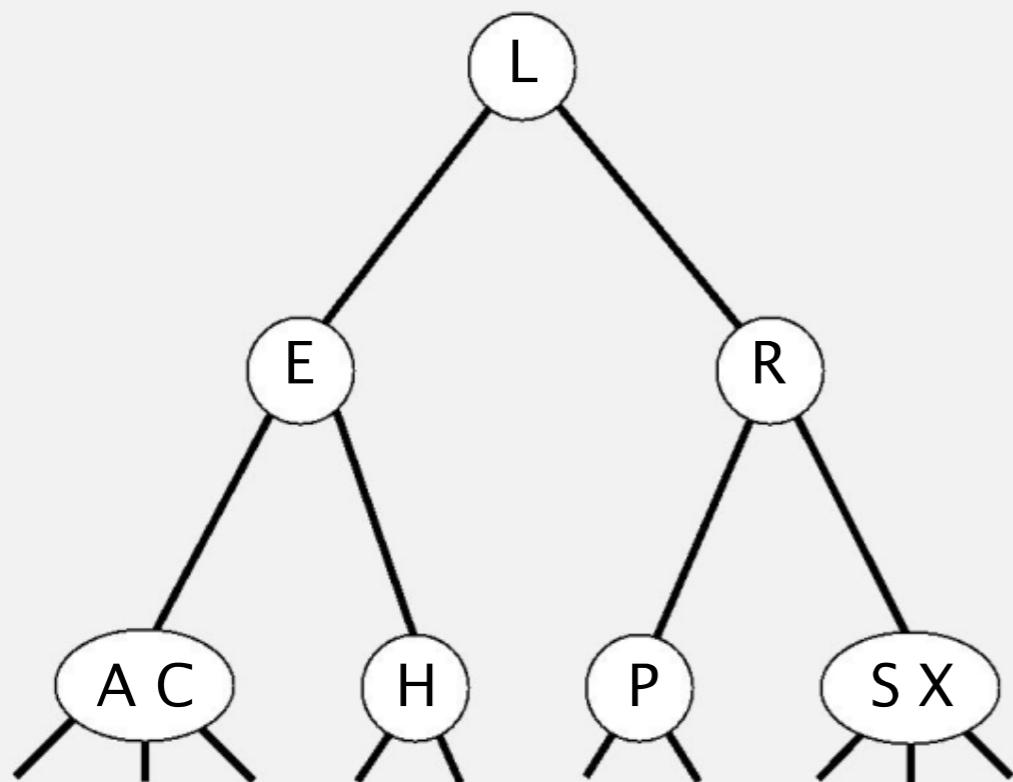


Insertion into a 2-3 tree

Insertion into a 2-node at bottom.

- Add new key to 2-node to create a 3-node.

insert G

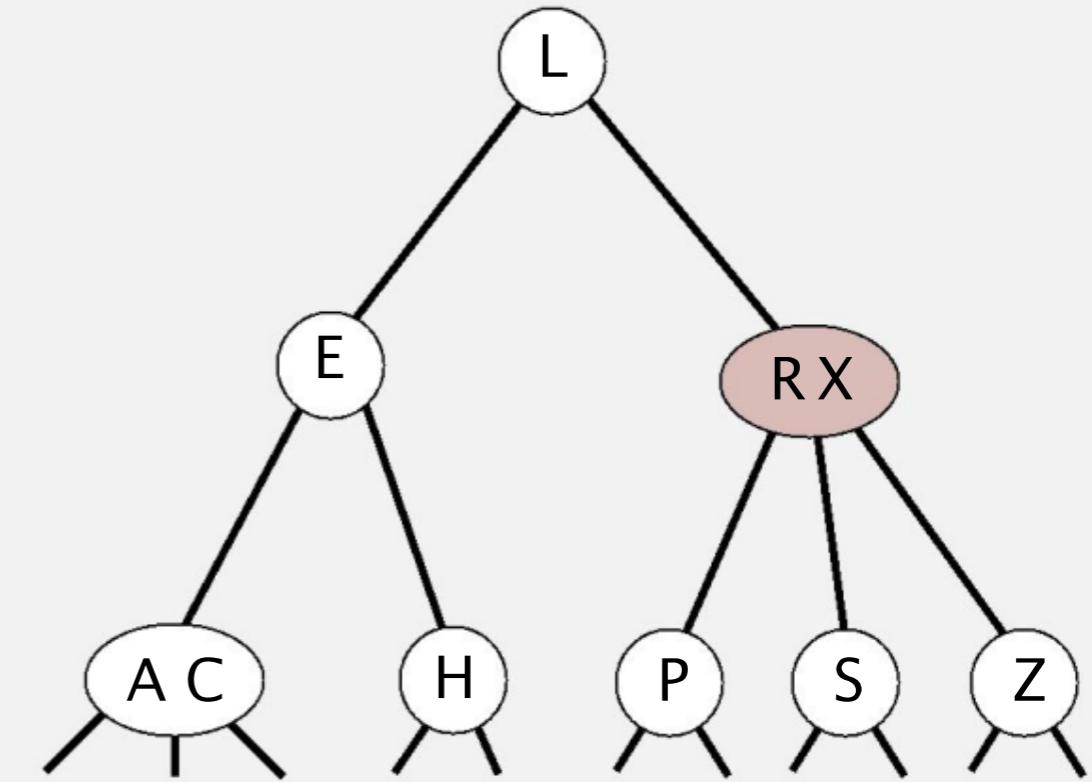
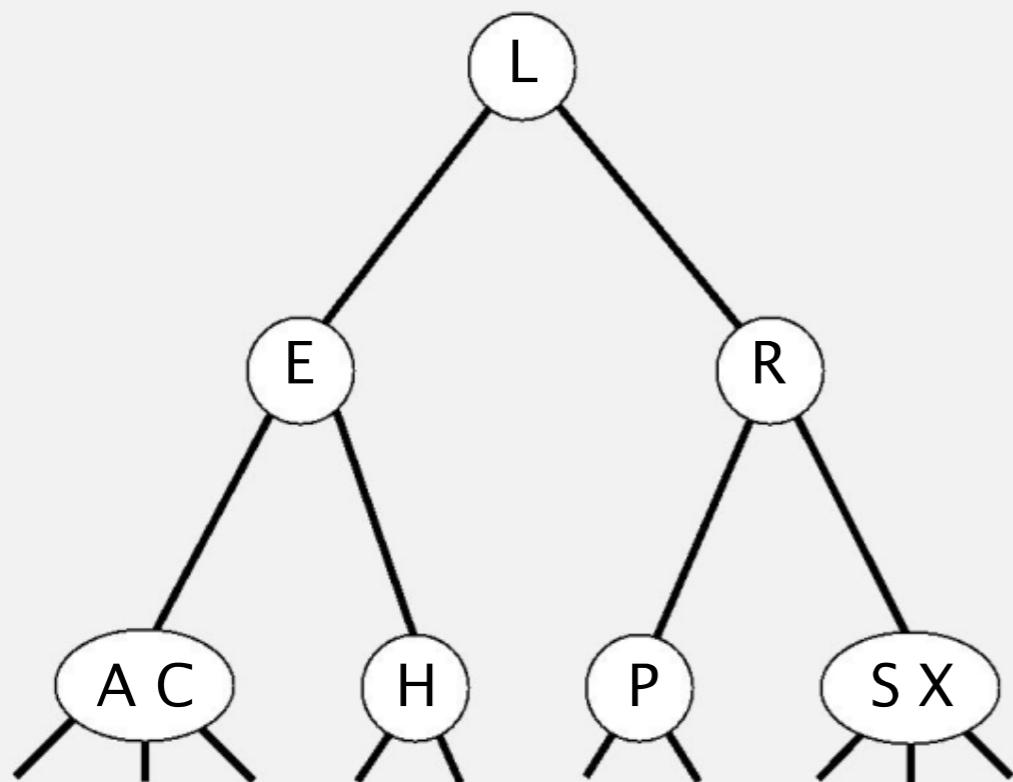


Insertion into a 2-3 tree

Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert Z

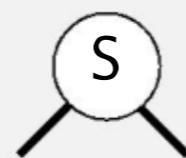


2-3 tree construction demo

insert S

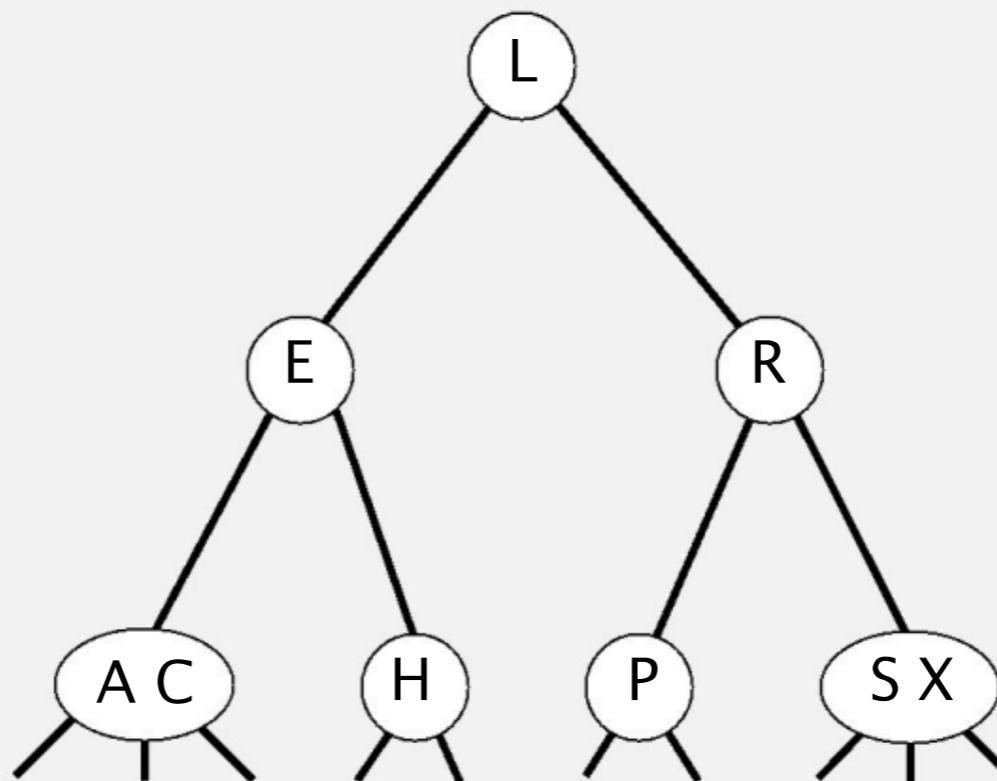


[https://algs4.cs.princeton.edu/lectures/
demo/33Demo23Tree.mov](https://algs4.cs.princeton.edu/lectures/demo/33Demo23Tree.mov)



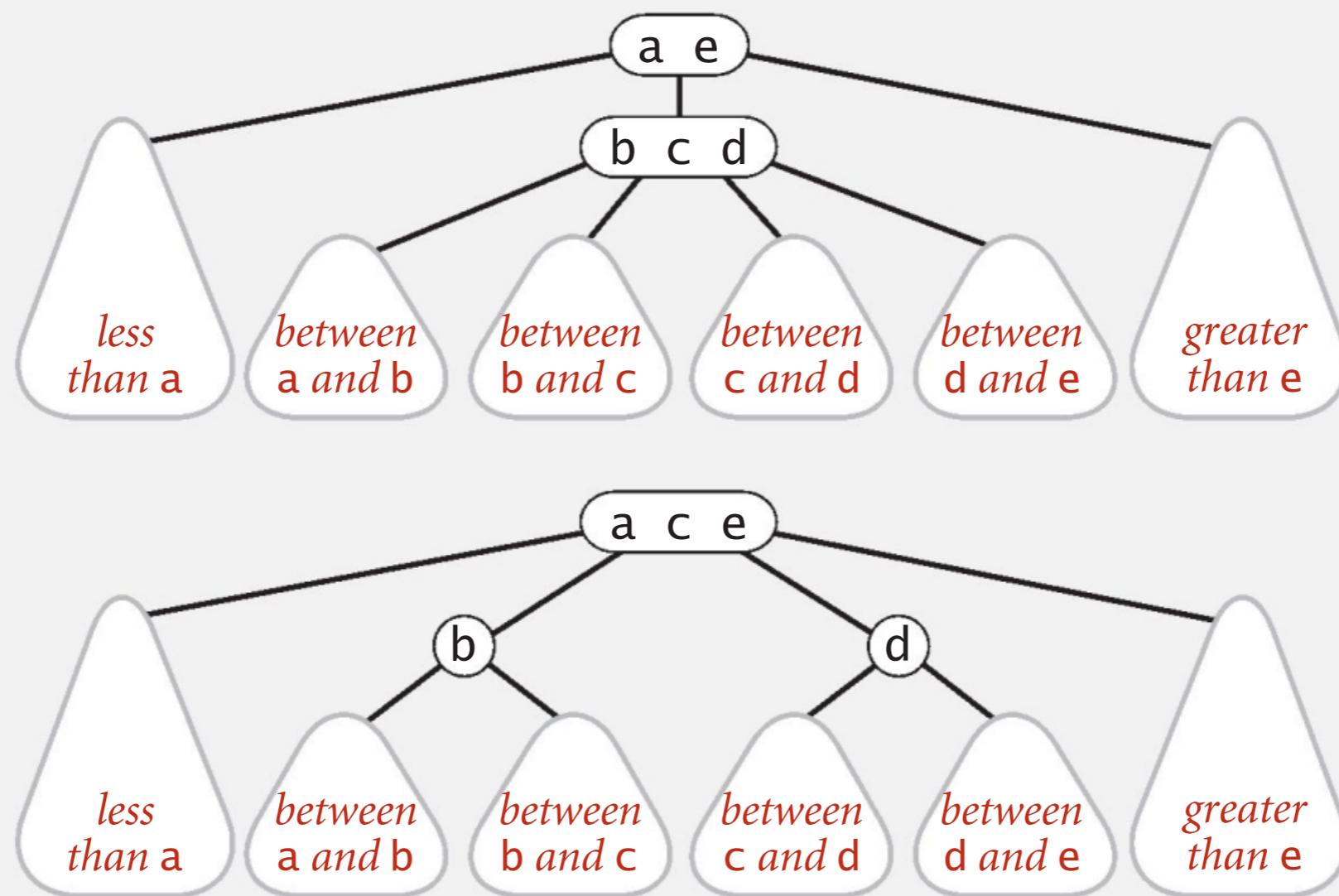
2-3 tree construction demo

2-3 tree



Local transformations in a 2-3 tree

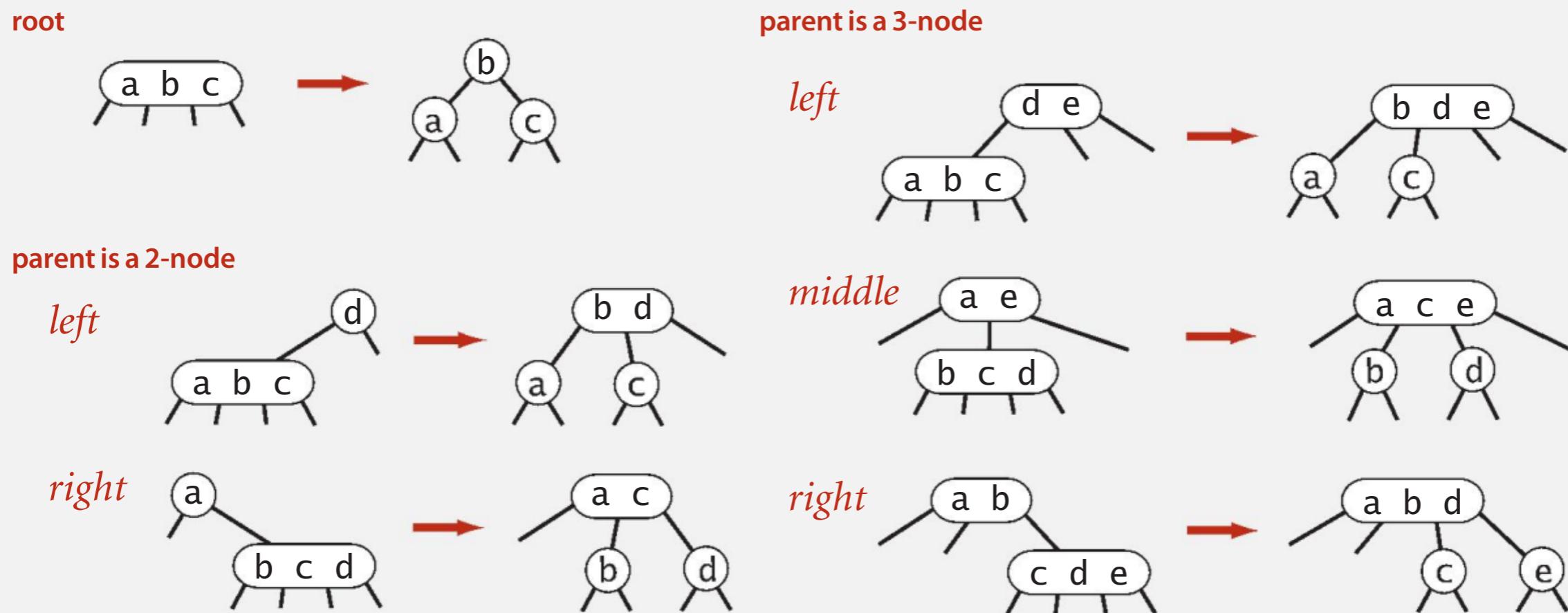
Splitting a 4-node is a **local** transformation: constant number of operations.



Global properties in a 2-3 tree

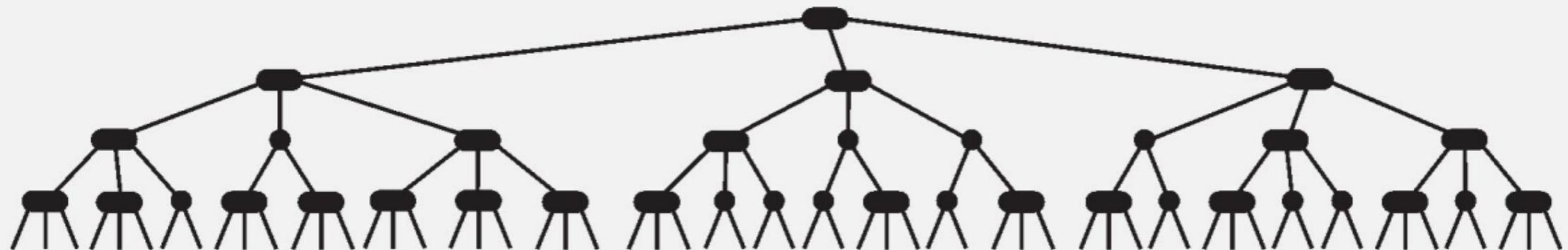
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

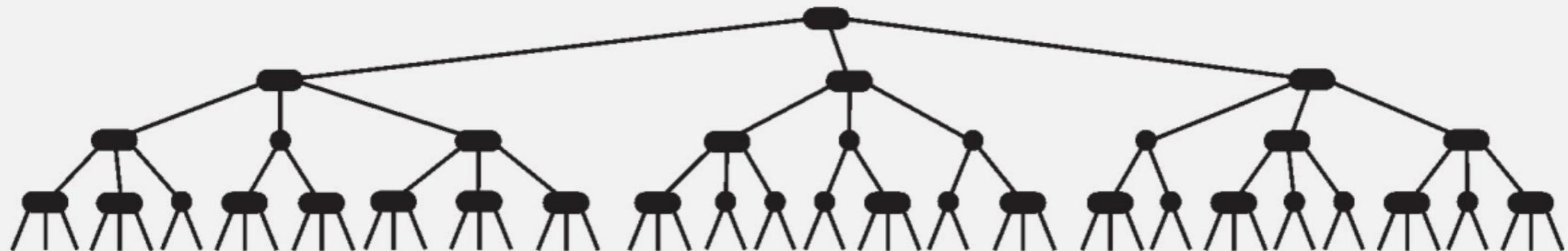


Tree height.

- Worst case:
- Best case:

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



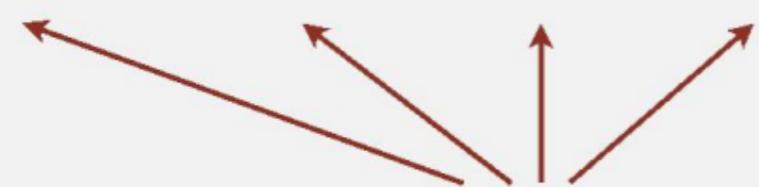
Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Bottom line. Guaranteed logarithmic performance for search and insert.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✓	<code>compareTo()</code>



 constant c depend upon implementation

2-3 tree: implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

fantasy code

```
public void put(Key key, Value val)
{
    Node x = root;
    while (x.getTheCorrectChild(key) != null)
    {
        x = x.getTheCorrectChildKey();
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
}
```

Bottom line. Could do it, but there's a better way.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

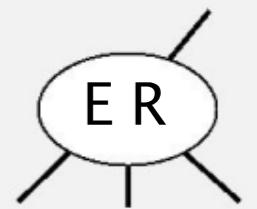
<http://algs4.cs.princeton.edu>

3.3 BALANCED SEARCH TREES

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

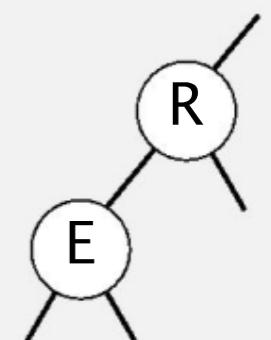
How to implement 2-3 trees with binary trees?

Challenge. How to represent a 3 node?



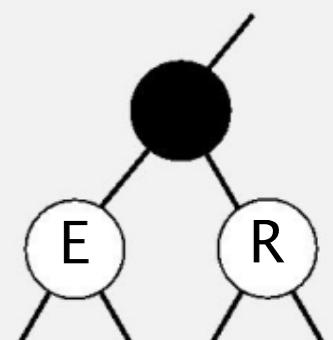
Approach 1: regular BST.

- No way to tell a 3-node from a 2-node.
- Cannot map from BST back to 2-3 tree.



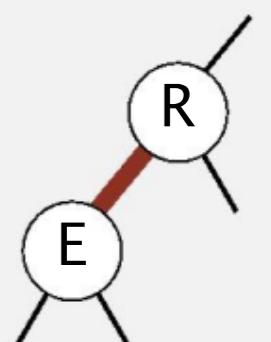
Approach 2: regular BST with "glue" nodes.

- Wastes space, wasted link.
- Code probably messy.



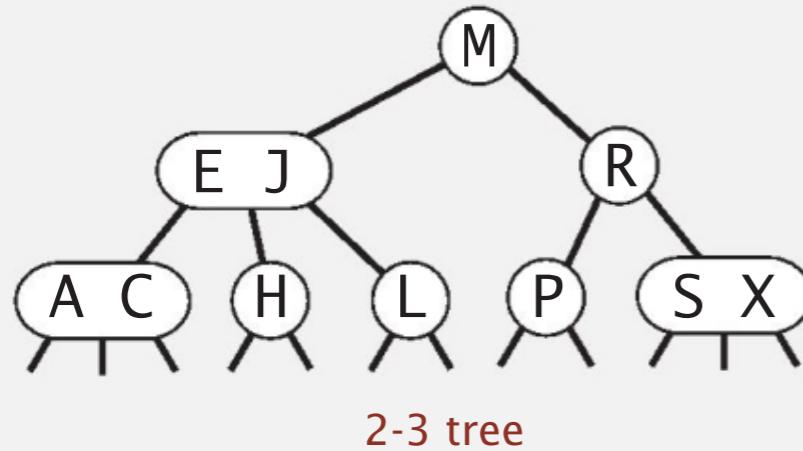
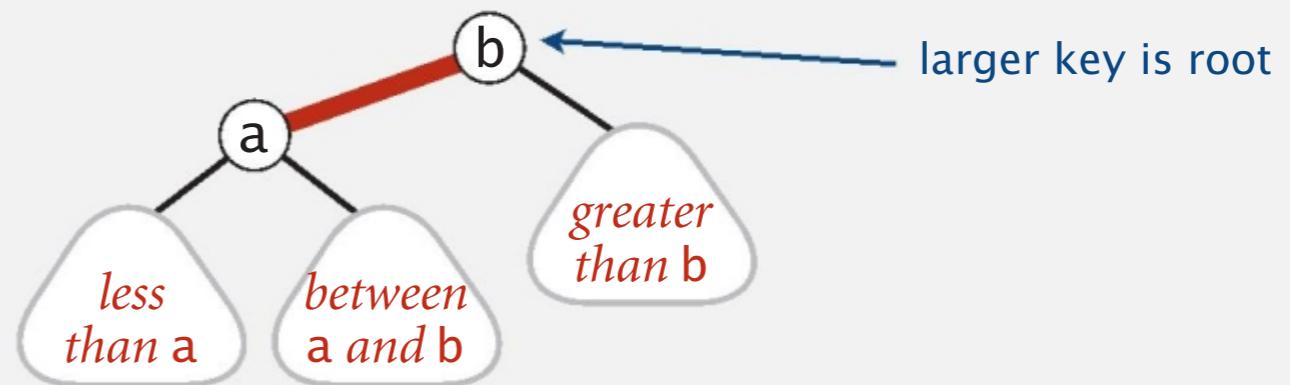
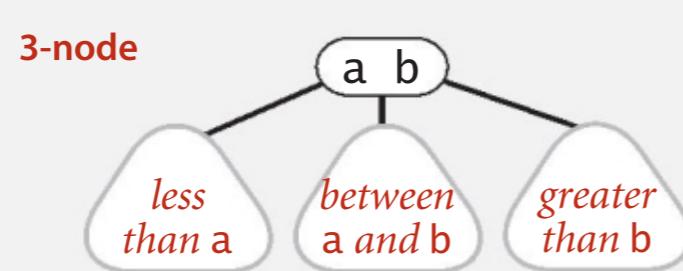
Approach 3: regular BST with red "glue" links.

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



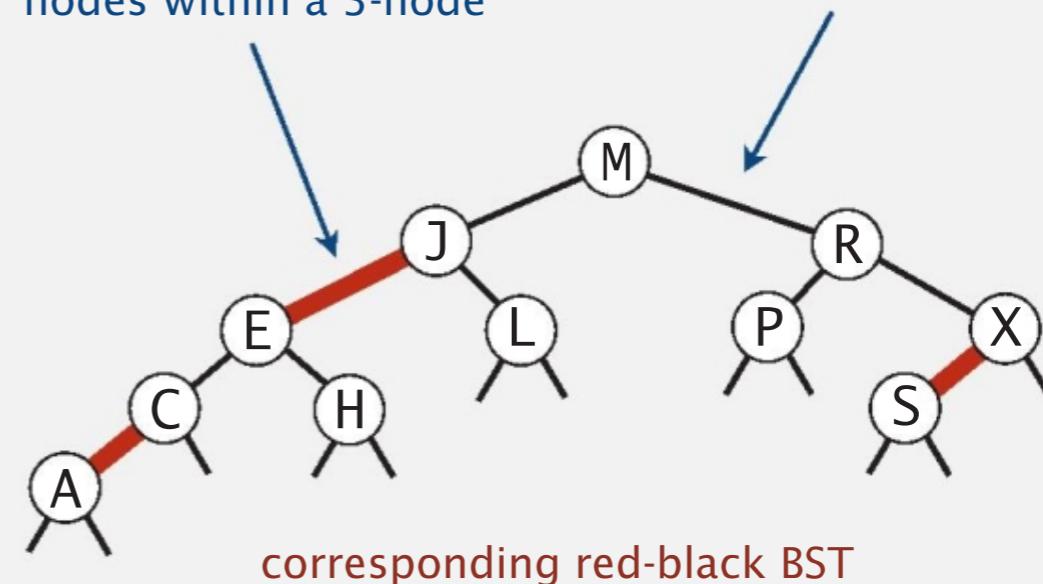
Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

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



red links "glue"
nodes within a 3-node

black links connect
2-nodes and 3-nodes

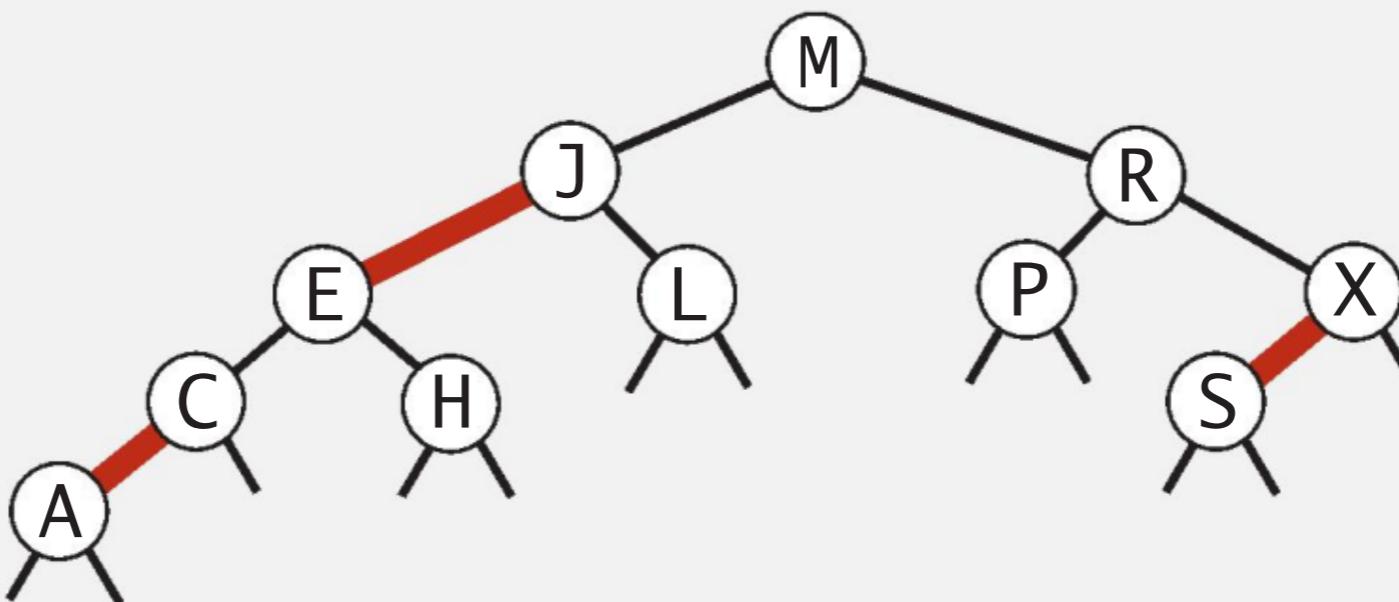


An equivalent definition

A BST such that:

- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

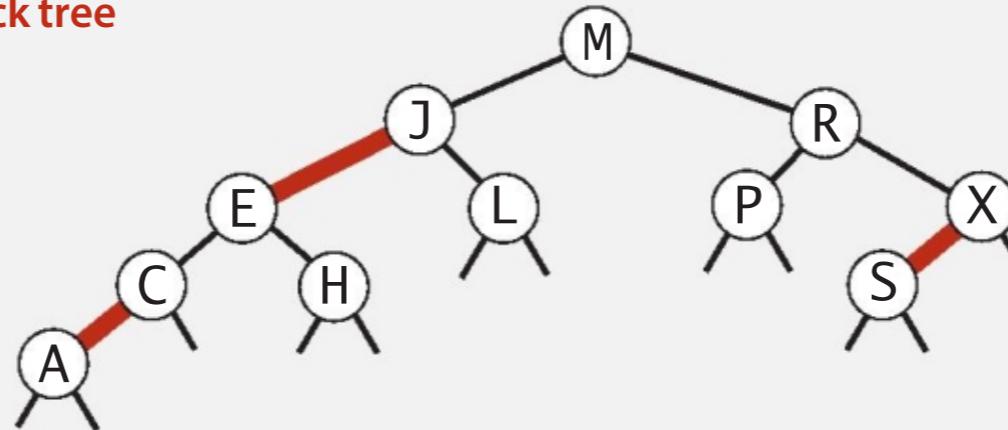
"perfect black balance"



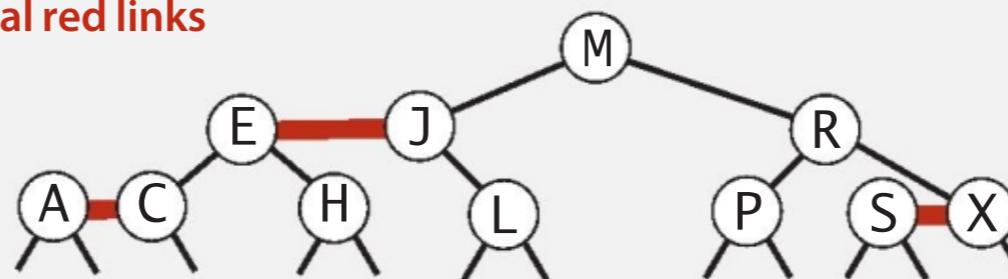
Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

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

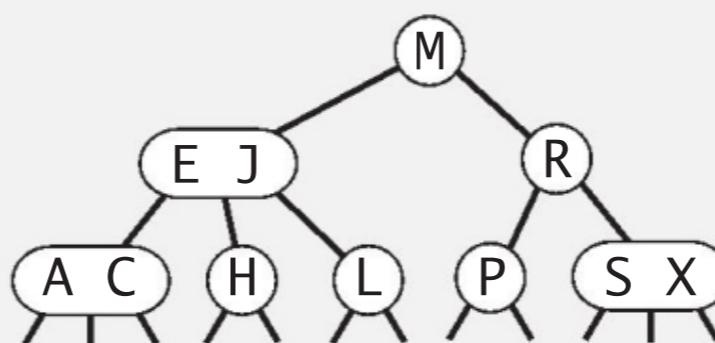
red–black tree



horizontal red links



2-3 tree

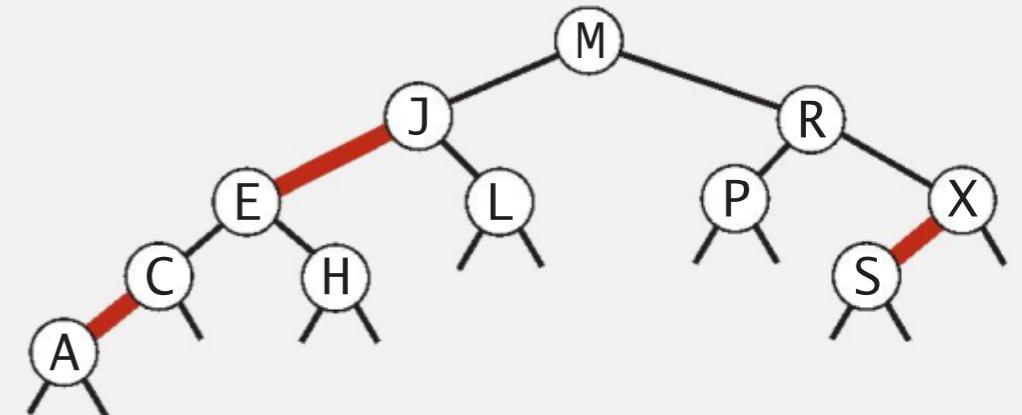


Search implementation for red-black BSTs

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

but runs faster
because of better balance

```
public Val 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. Most other ops (e.g., floor, iteration, selection) are also identical.

Red-black BST representation

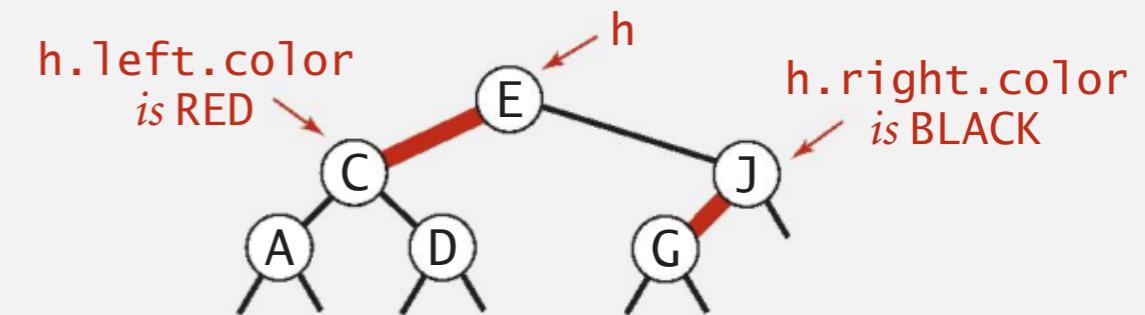
Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

```
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
}
```

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

null links are black

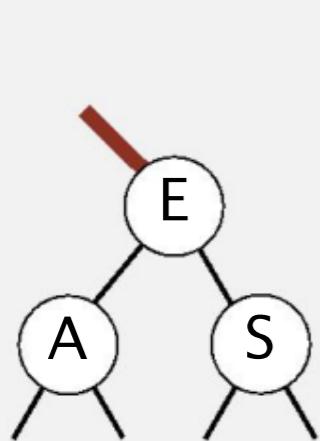


Insertion in a LLRB tree: overview

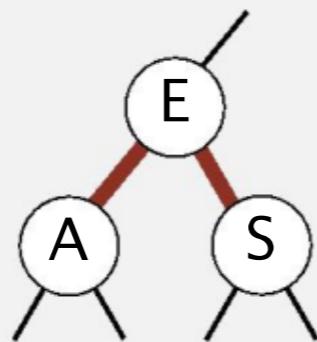
Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

During internal operations, maintain:

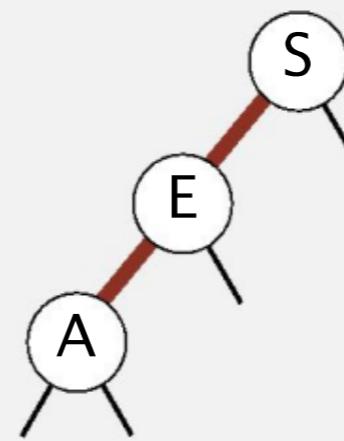
- Symmetric order.
- Perfect black balance.
[but not necessarily color invariants]



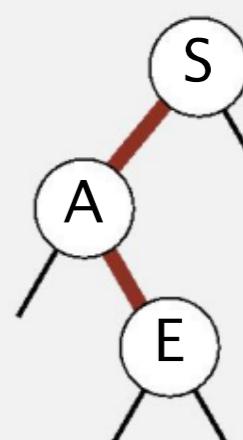
right-leaning
red link



two red children
(a temporary 4-node)



left-left red
(a temporary 4-node)



left-right red
(a temporary 4-node)

How? Apply elementary red-black BST operations: rotation and color flip.

Insertion in a LLRB tree: overview

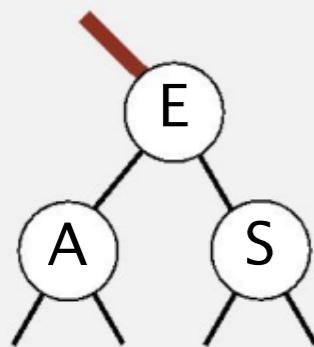
Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

During internal operations, maintain:

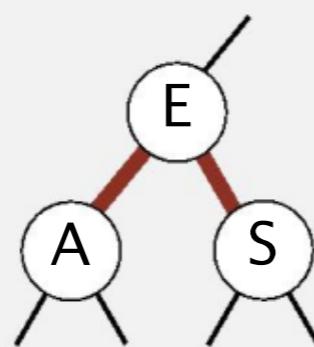
- Symmetric order.
- Perfect black balance.

[but not necessarily color invariants]

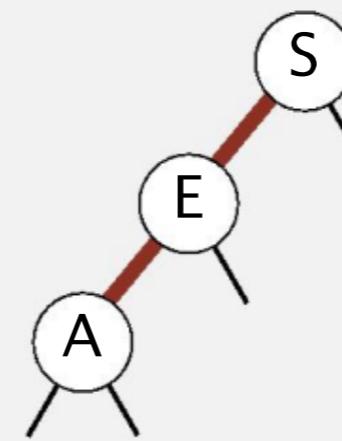
Imagine: all the RED Links
are in horizontal direction!



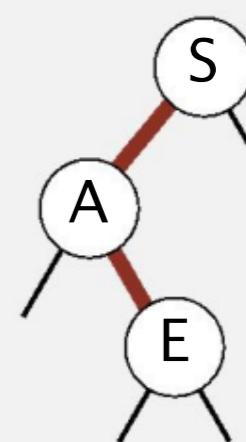
right-leaning
red link



two red children
(a temporary 4-node)



left-left red
(a temporary 4-node)



left-right red
(a temporary 4-node)

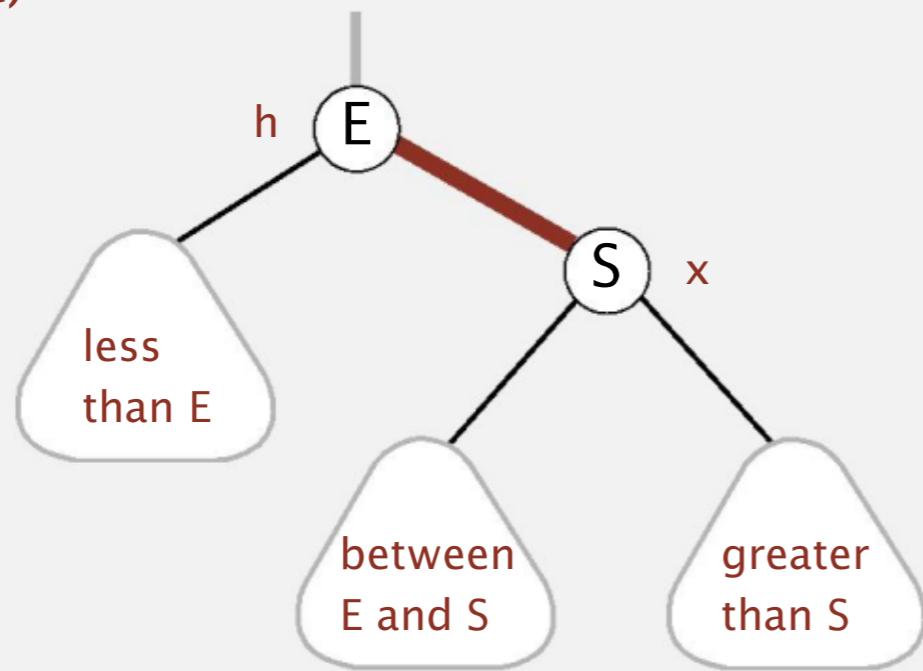
How? Apply elementary red-black BST operations: rotation and color flip.

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;  
}
```

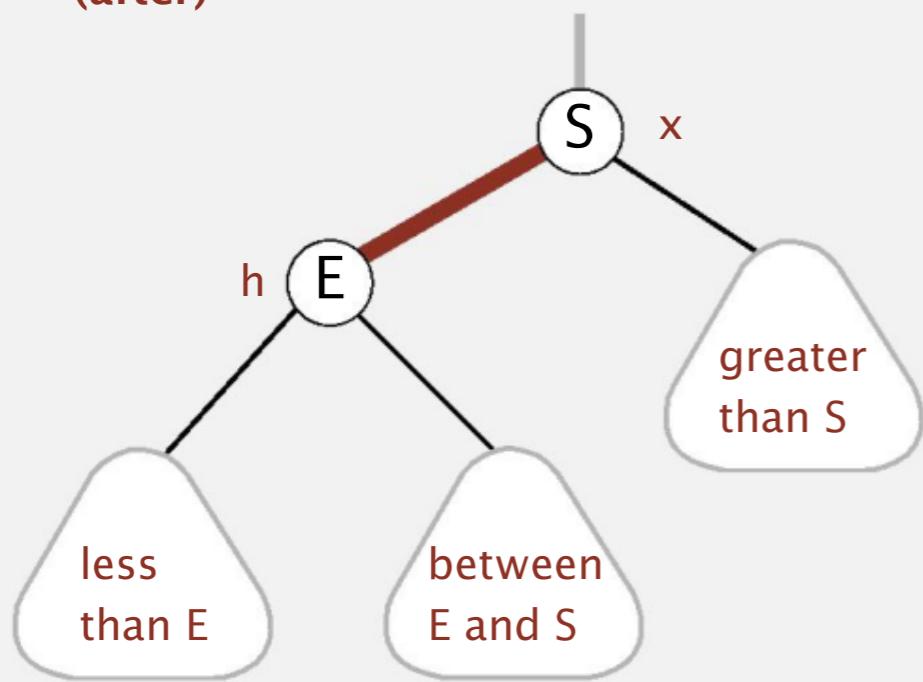
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

rotate E left

(after)



```
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;  
}
```

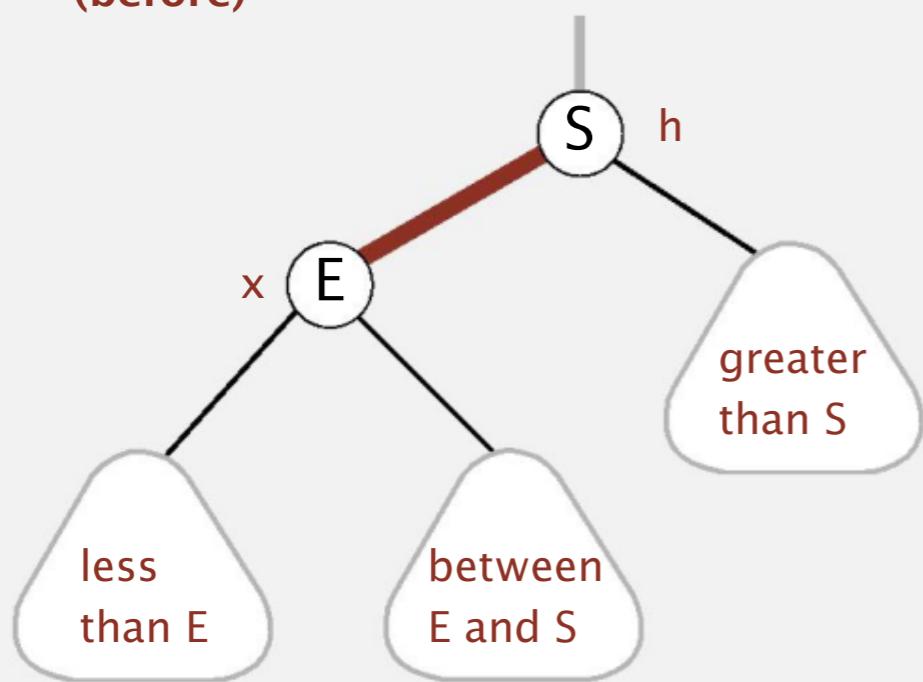
Invariants. Maintains symmetric order and perfect black balance.

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;  
}
```

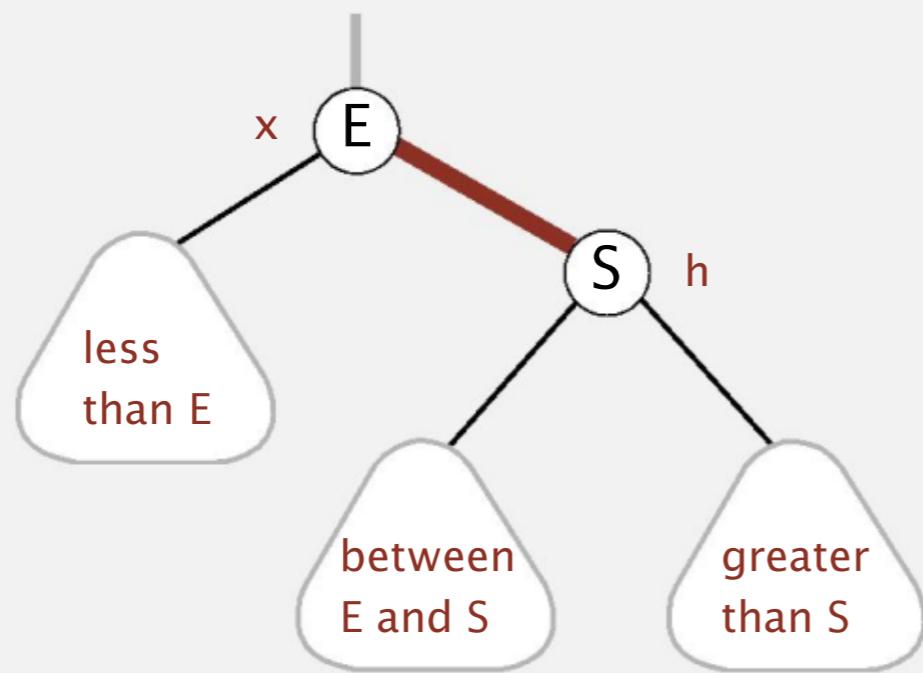
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

rotate S right

(after)

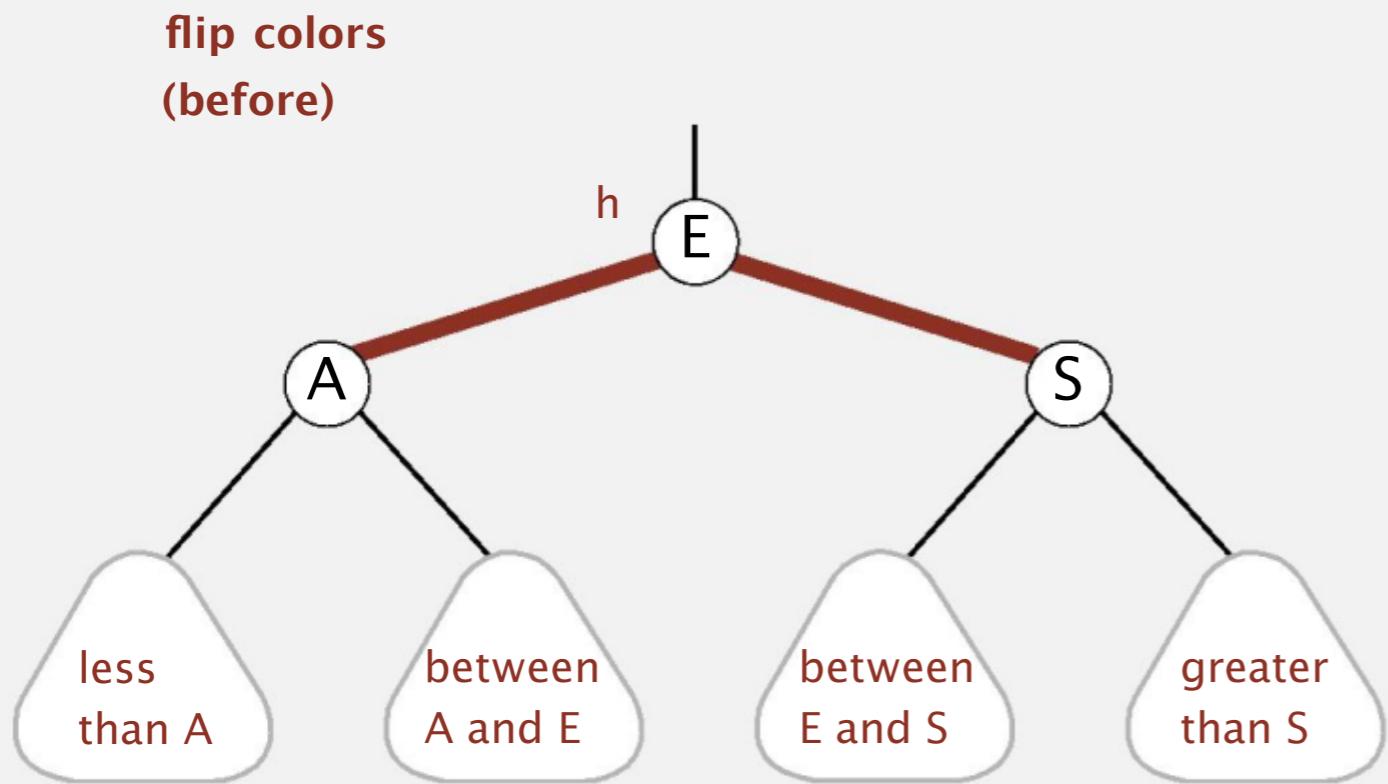


```
private Node rotateRight(Node h) {  
    assert isRed(h.left);  
    Node x = h.left;  
    h.left = x.right;  
    x.right = h;  
    x.color = h.color;  
    h.color = RED;  
    return x;  
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

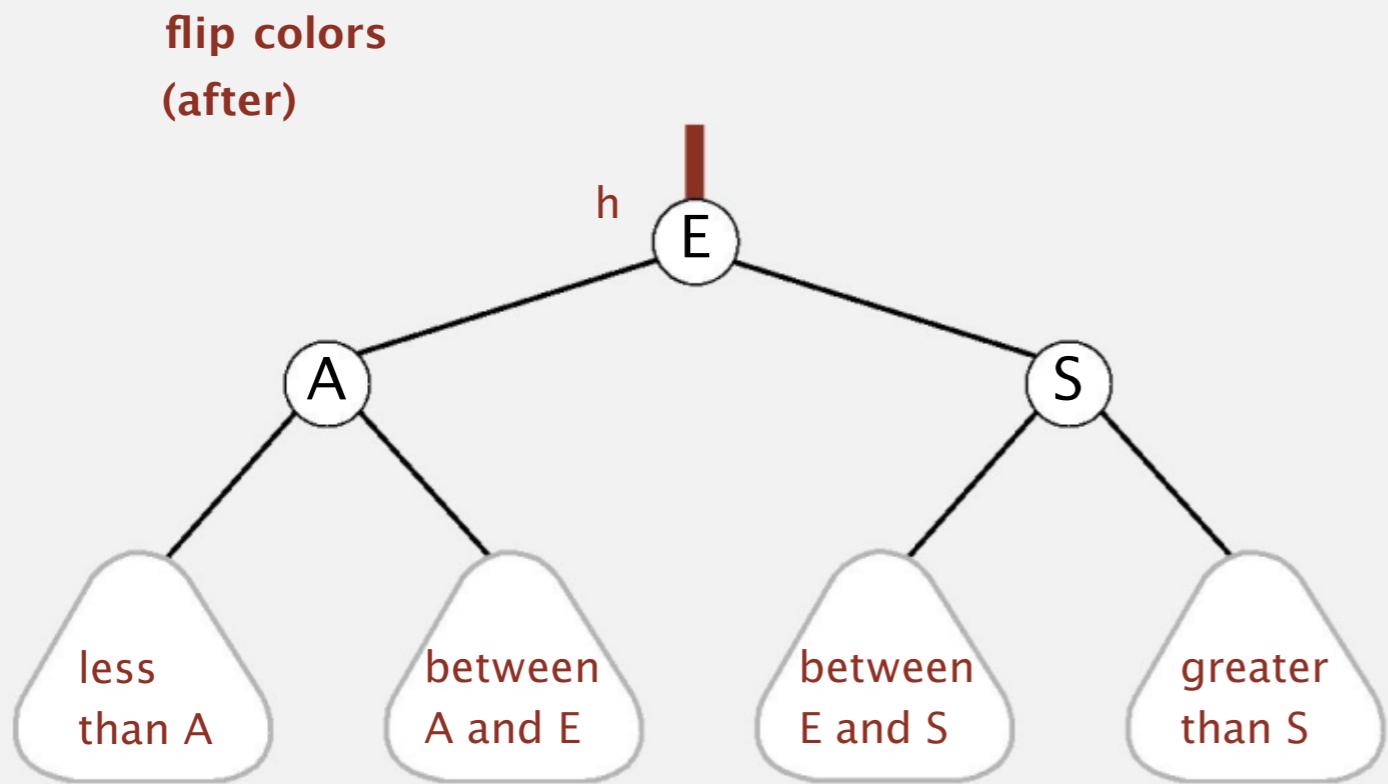


```
private void flipColors(Node h) {  
    assert !isRed( h );  
    assert isRed( h.left );  
    assert isRed( h.right );  
    h.color = RED;  
    h.left.color = BLACK;  
    h.right.color = BLACK;  
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

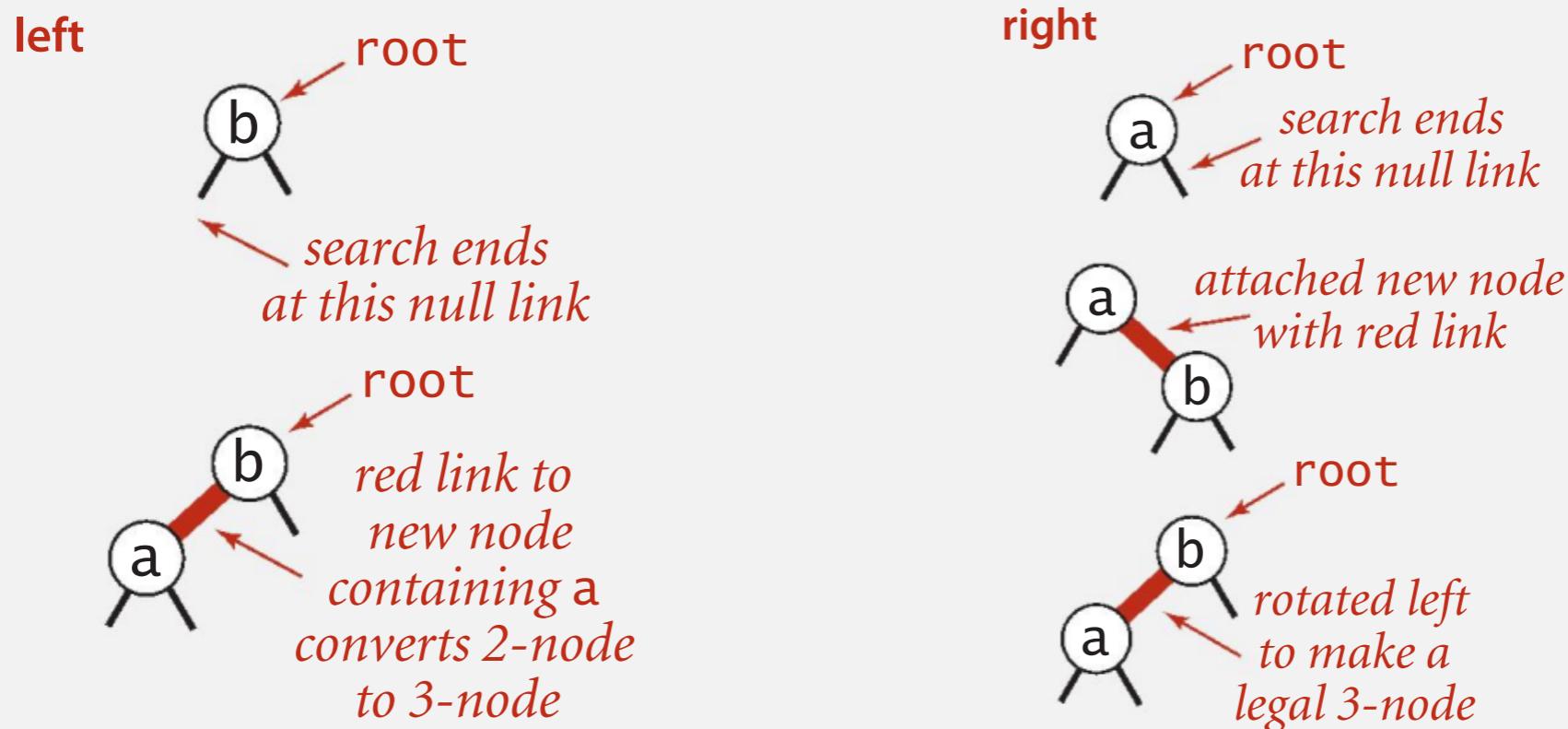


```
private void flipColors(Node h) {  
    assert !isRed( h );  
    assert isRed( h.left );  
    assert isRed( h.right );  
    h.color = RED;  
    h.left.color = BLACK;  
    h.right.color = BLACK;  
}
```

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree

Warmup 1. Insert into a tree with exactly 1 node.



Insertion in a LLRB tree

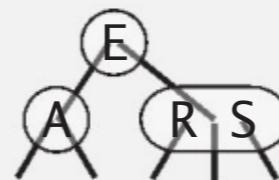
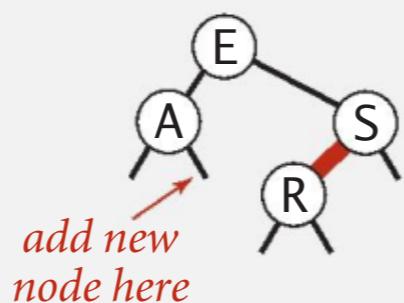
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red. ←
- If new red link is a right link, rotate left. ←

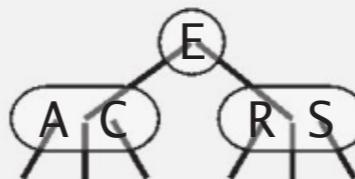
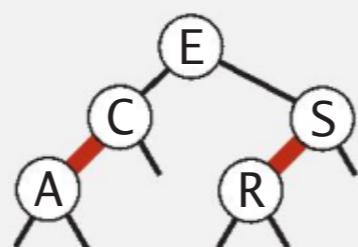
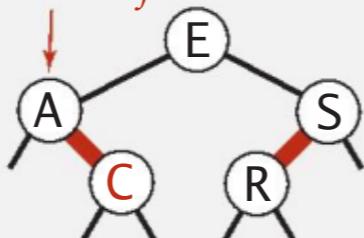
to maintain symmetric order
and perfect black balance

to fix color invariants

insert C

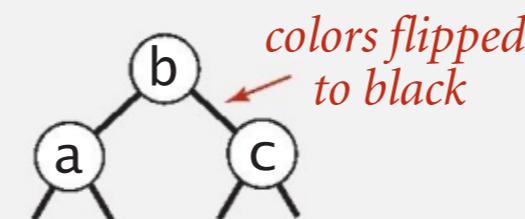
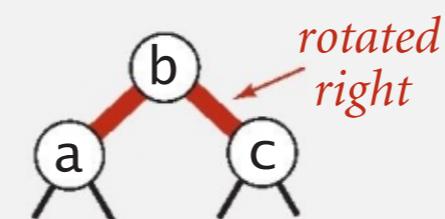
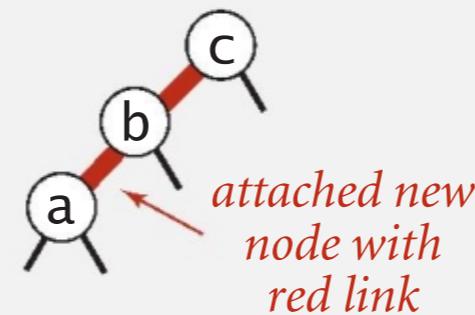
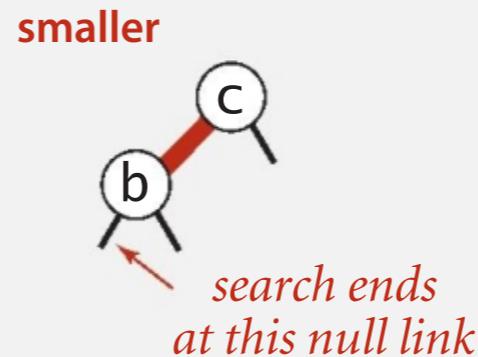
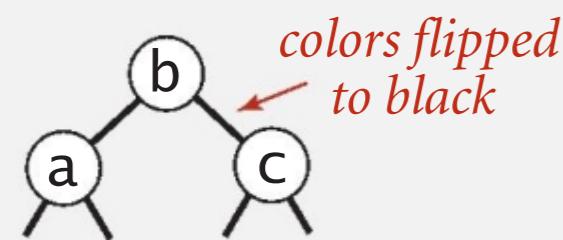
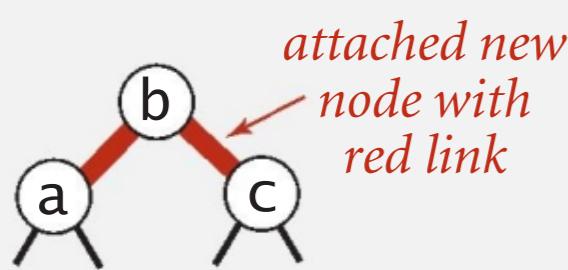
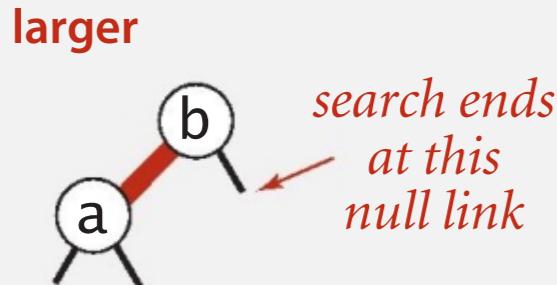


right link red
so rotate left

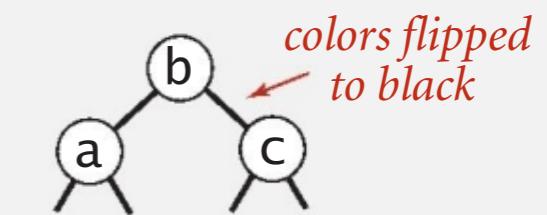
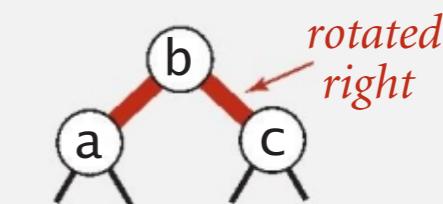
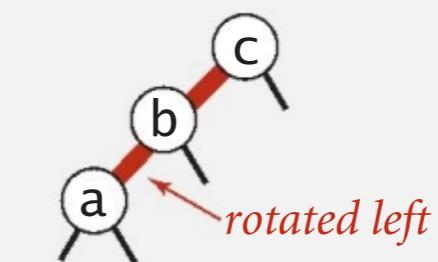
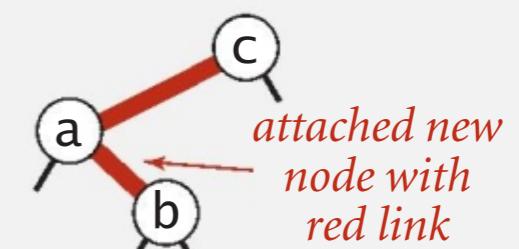
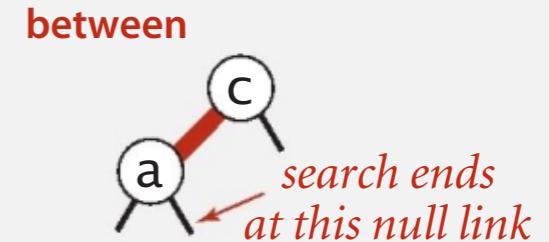


Insertion in a LLRB tree

Warmup 2. Insert into a tree with exactly 2 nodes.



b



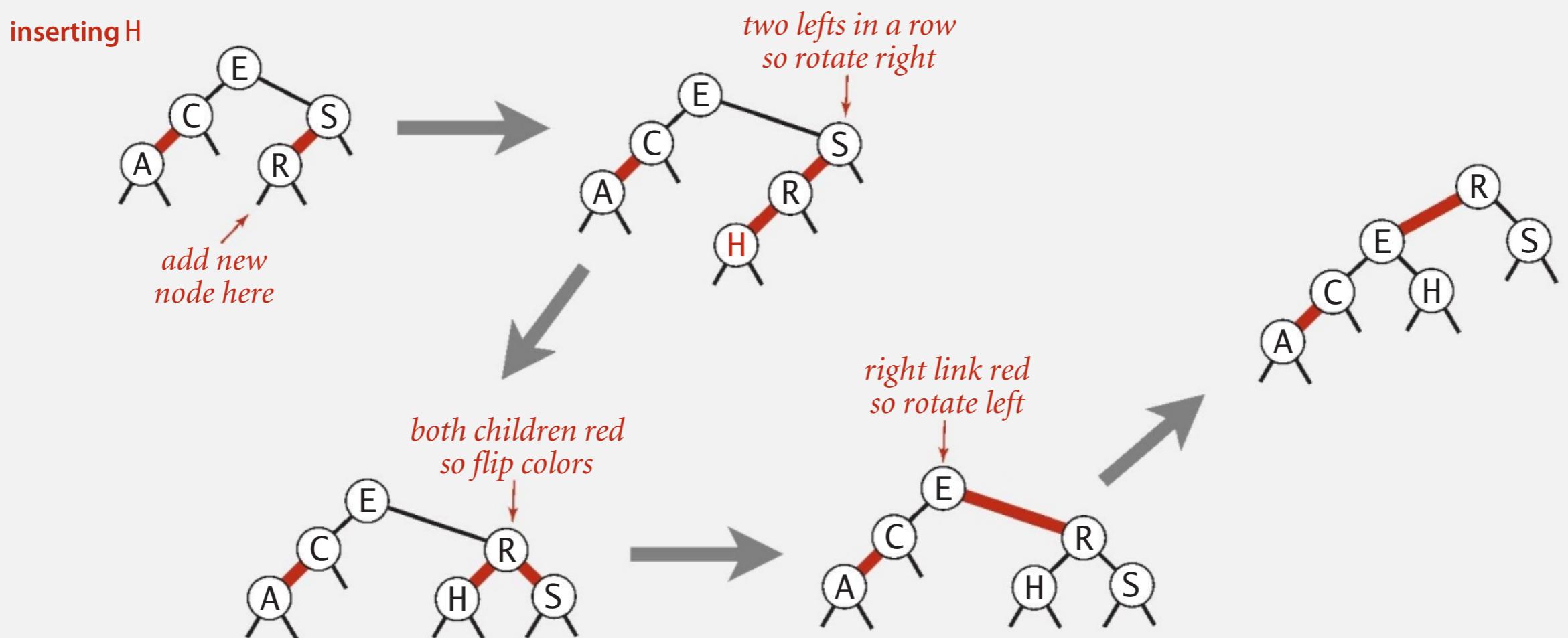
Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
 - Rotate to balance the 4-node (if needed).
 - Flip colors to pass red link up one level.
 - Rotate to make lean left (if needed).

to maintain symmetric order
and perfect black balance

• to fix color invariants



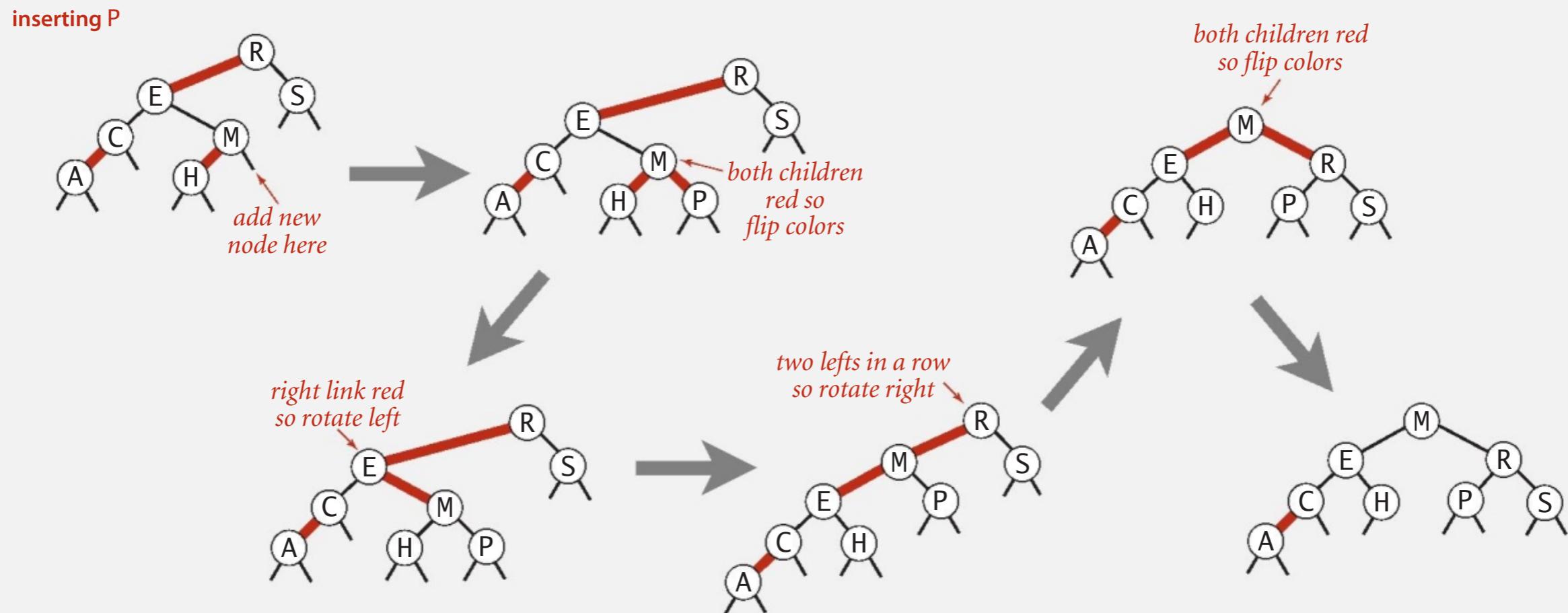
Insertion in a LLRB tree: passing red links up the tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat case 1 or case 2 up the tree (if needed).

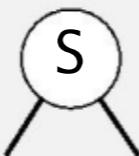
to maintain symmetric order
and perfect black balance

to fix color invariants



Red-black BST construction demo

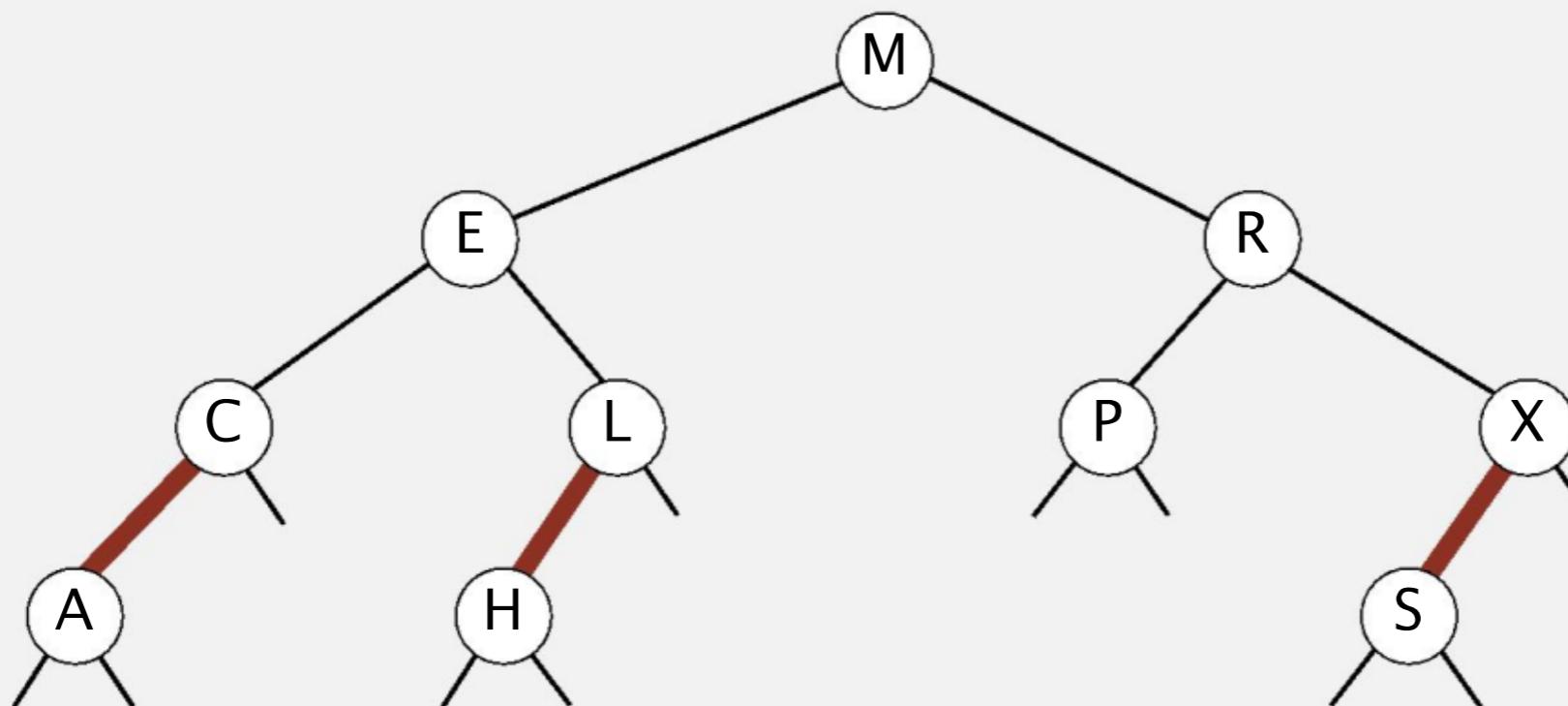
insert S



<https://algs4.cs.princeton.edu/lectures/demo/33DemoRedBlackBST.mov>

Red-black BST construction demo

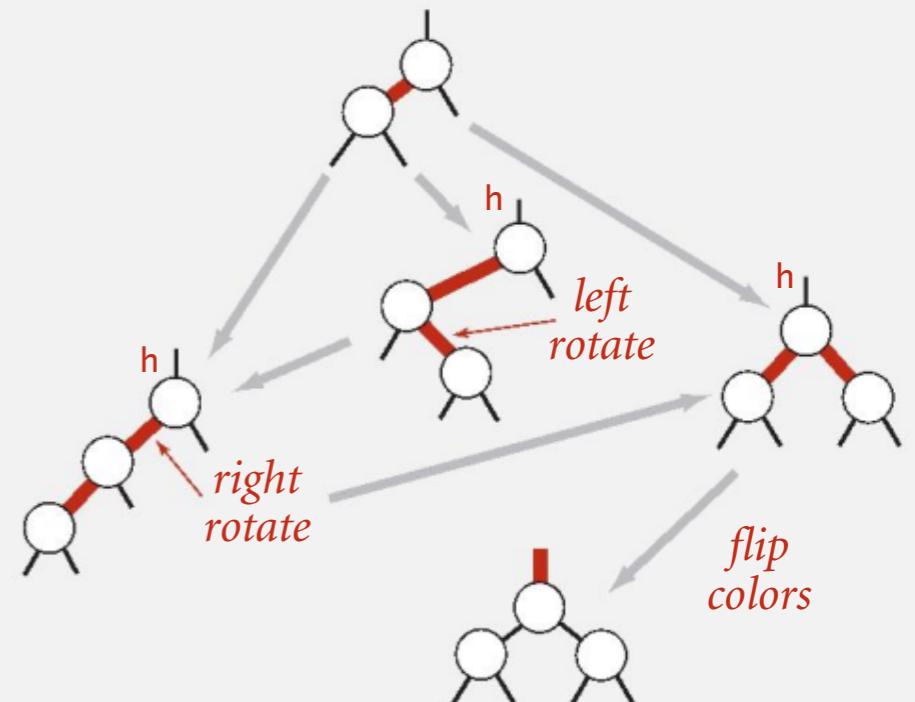
red-black BST



Insertion in a LLRB tree: Java implementation

Same code for all cases.

- Right child red, left child black: **rotate left**.
- Left child, left-left grandchild red: **rotate right**.
- Both children red: **flip colors**.



```
private Node put (Node h, Key key, Value val) {  
    if (h == null) return new Node( key, val, 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;  
}
```

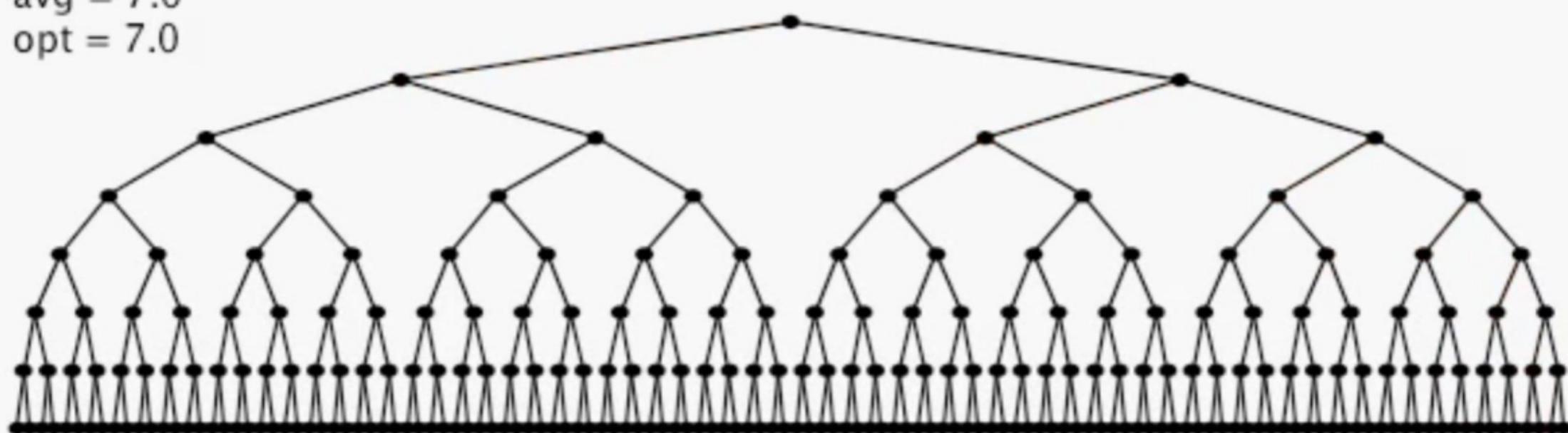
insert at bottom
(and color it red)

lean left
balance 4-node
split 4-node

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

Insertion in a LLRB tree: visualization

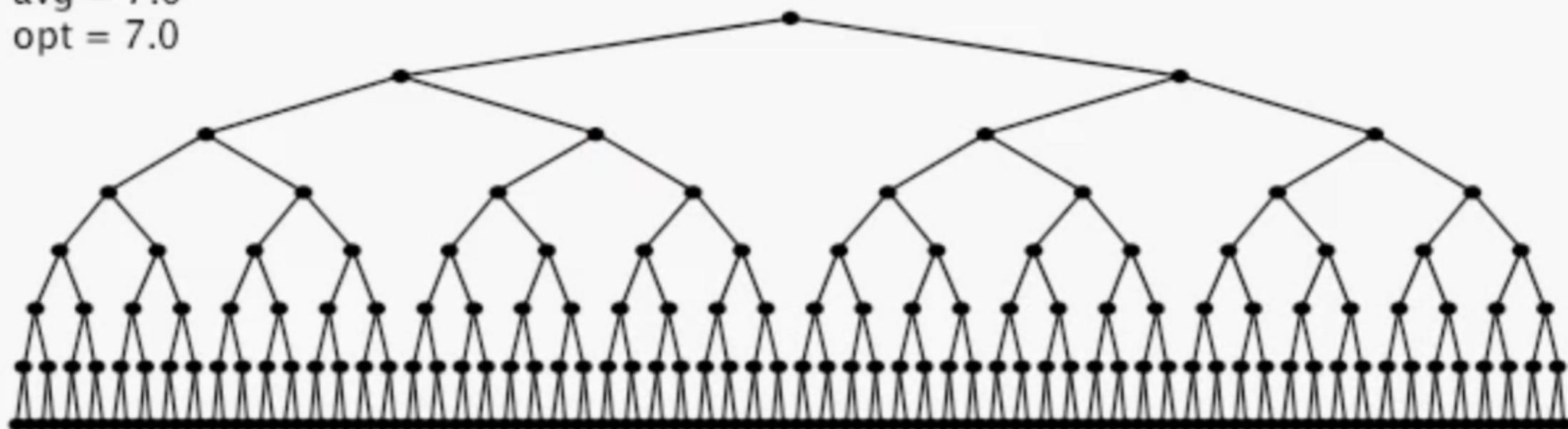
N = 255
max = 8
avg = 7.0
opt = 7.0



255 insertions in ascending order

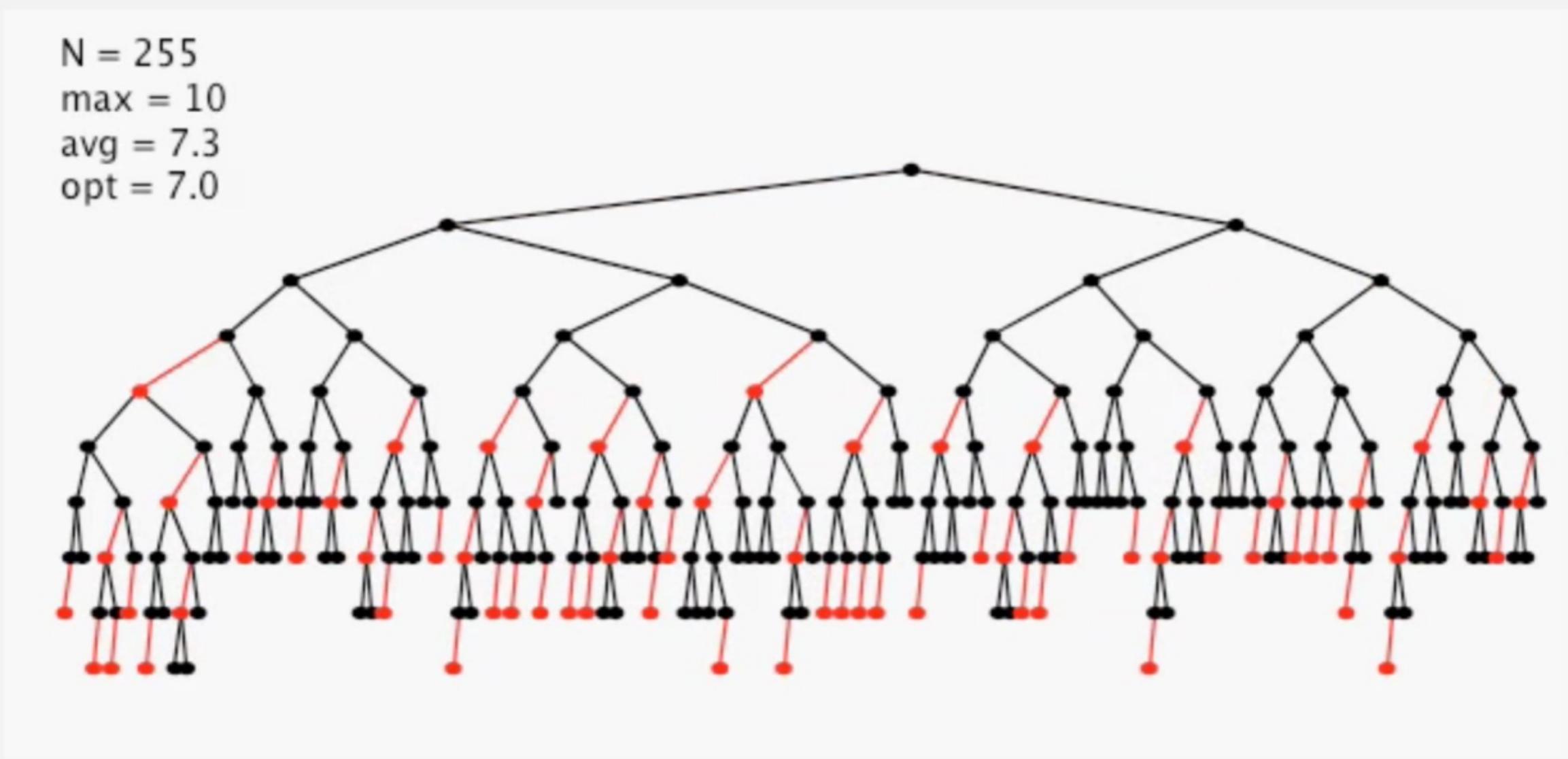
Insertion in a LLRB tree: visualization

N = 255
max = 8
avg = 7.0
opt = 7.0



255 insertions in descending order

Insertion in a LLRB tree: visualization

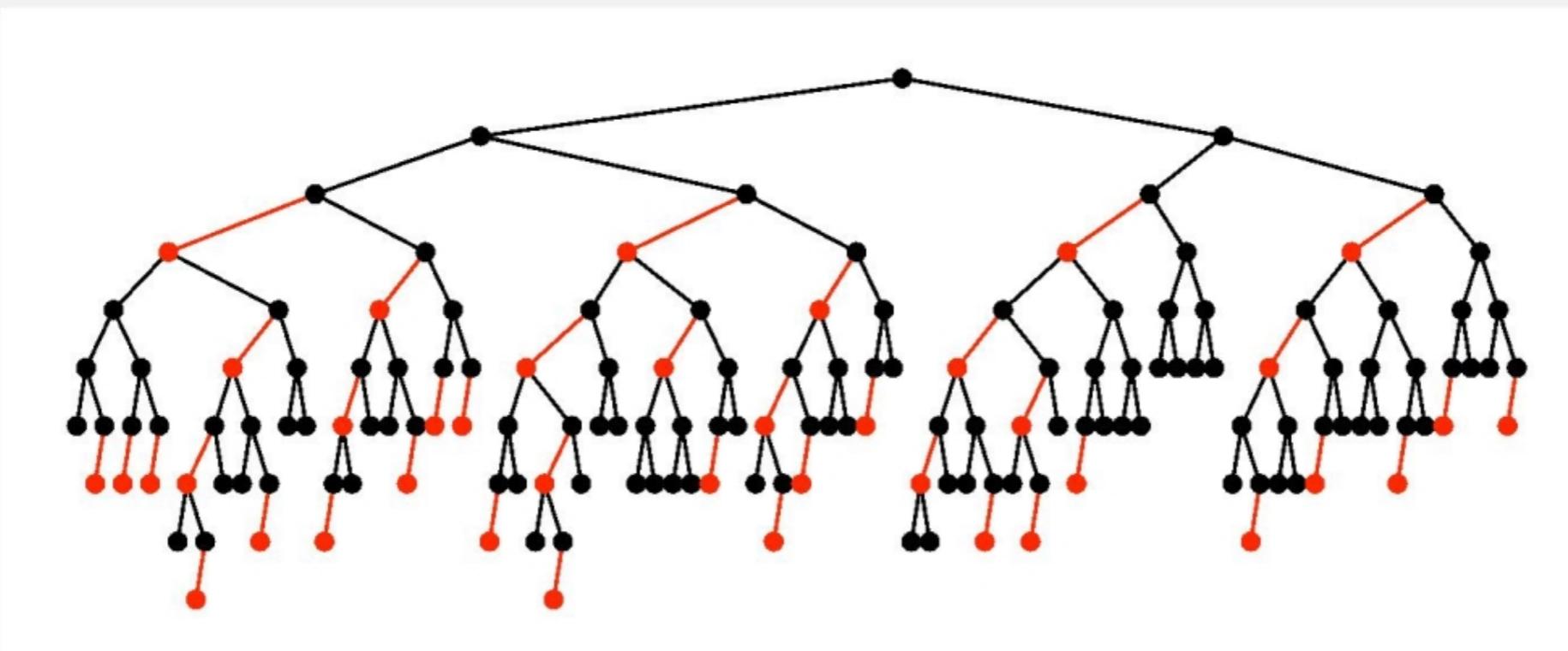


Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \lg N$ in the worst case.

Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.



Property. Height of tree is $\sim 1.0 \lg N$ in typical applications.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N^*$	$1.0 \lg N^*$	$1.0 \lg N^*$	✓	<code>compareTo()</code>

* exact value of coefficient unknown but extremely close to 1

War story: why red-black?

Xerox PARC innovations. [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- InterPress.
- 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

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

and

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

War story: red-black BSTs

Telephone company contracted with database provider to build real-time database to store customer information.

Database implementation.

- Red-black BST search and insert; Hibbard deletion.
- Exceeding height limit of 80 triggered error-recovery process.

allows for up to 2^{40} keys

Extended telephone service outage.

- Main cause = height bounded exceeded!
- Telephone company sues database provider.
- Legal testimony:

Hibbard deletion
was the problem



“If implemented properly, the height of a red-black BST with N keys is at most $2 \lg N$. ” — expert witness



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.3 BALANCED SEARCH TREES

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

File system model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).



Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

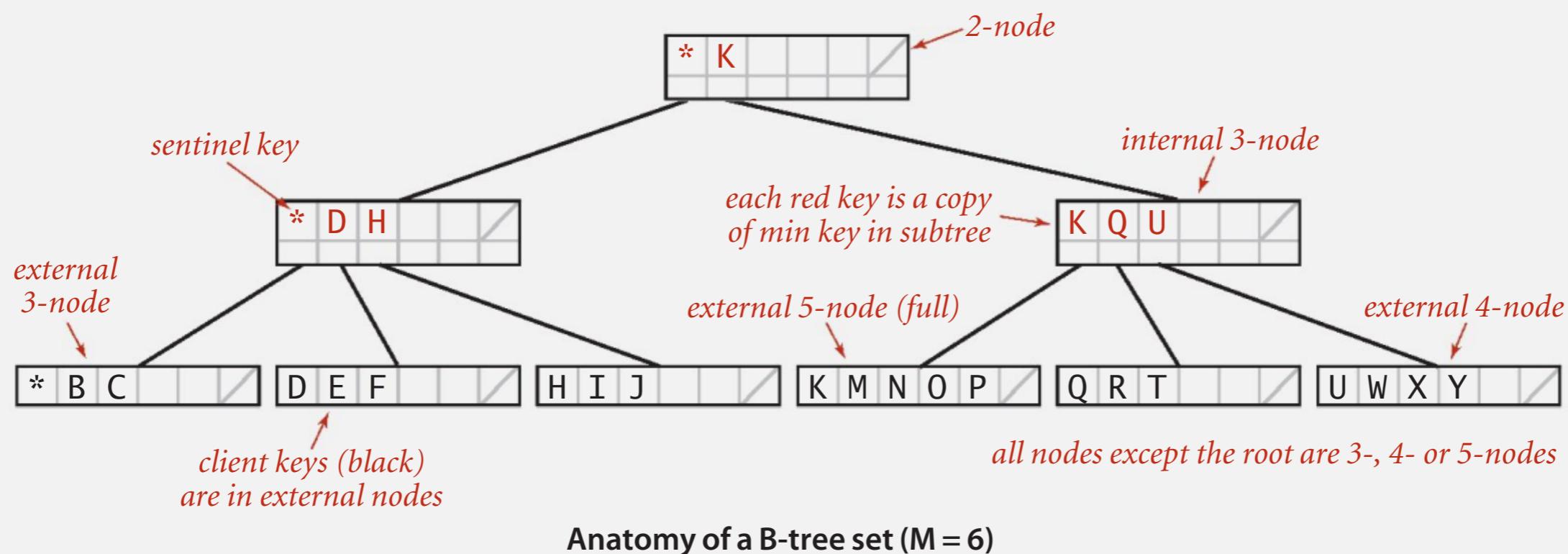
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

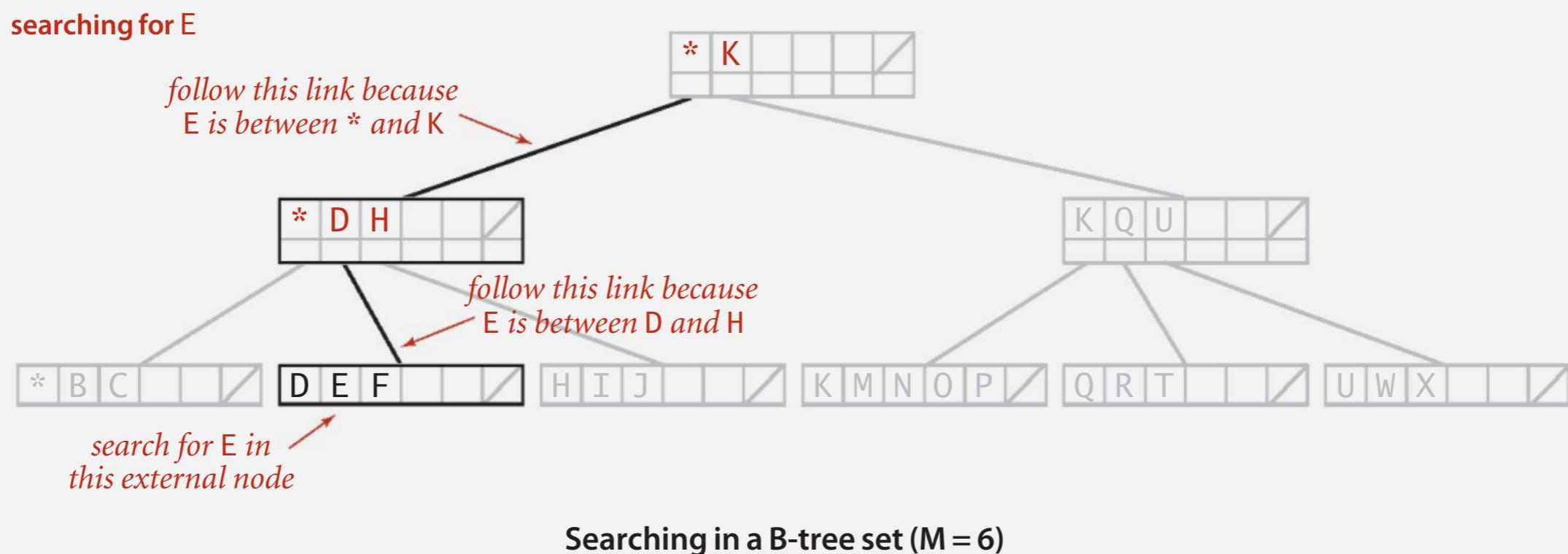
- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

choose M as large as possible so that M links fit in a page, e.g., $M = 1024$



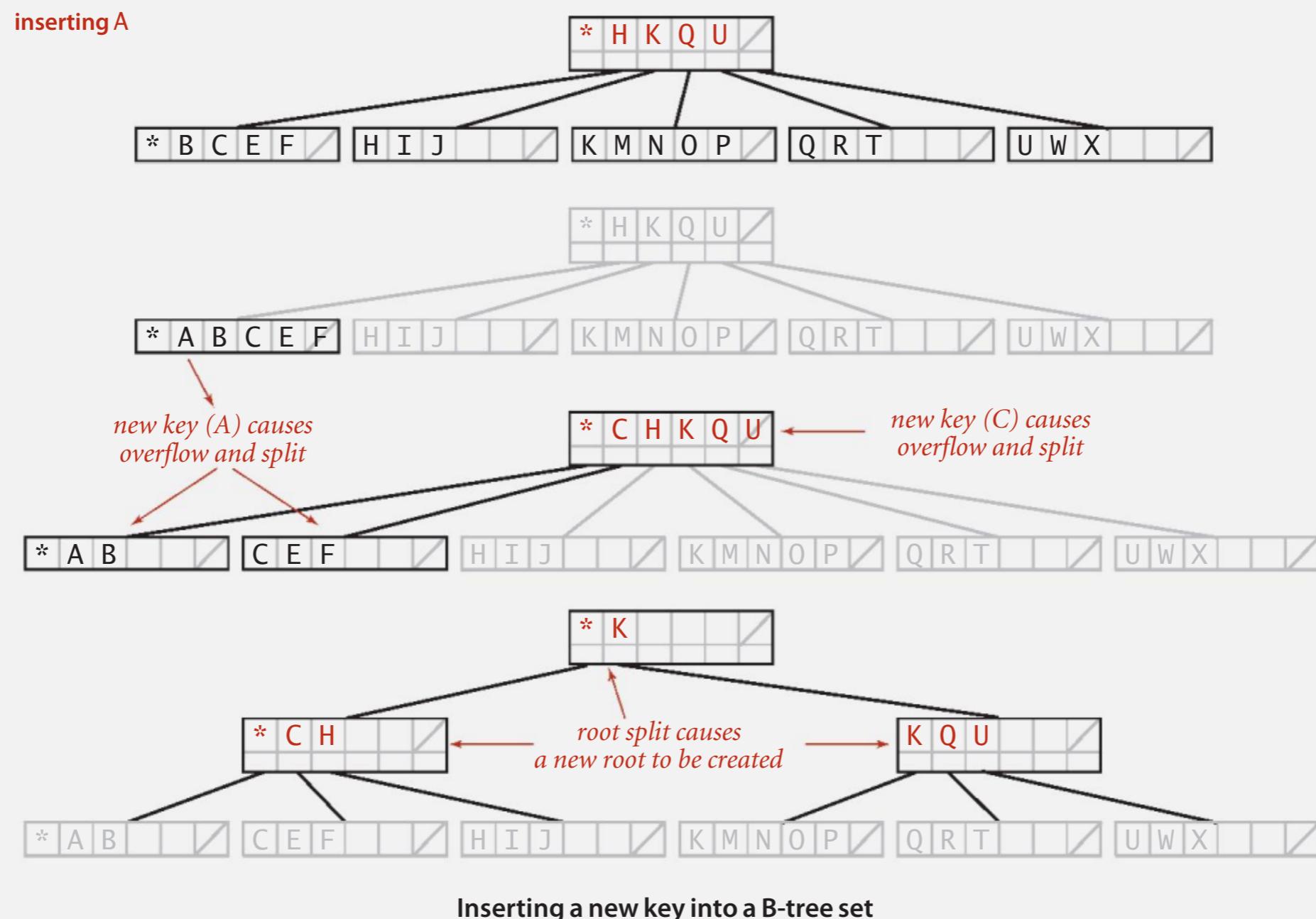
Searching in a B-tree

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Balance in B-tree

Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

Pf. All internal nodes (besides root) have between $M/2$ and $M - 1$ links.

In practice. Number of probes is at most 4. \leftarrow $M = 1024; N = 62 \text{ billion}$
 $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

Building a large B tree



Balanced trees in the wild

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

- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: `map`, `multimap`, `multiset`.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.
- Emacs: conservative stack scanning.

B-tree variants. B+ tree, B*tree, B# tree, ...

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

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

Summary

- Binary Search Trees (3.2 of Text A)
- Balanced Search Trees (3.3 of Text A)
- Hash Tables (3.4 of Text A)

To be discussed in Lecture 12:

- Hash Tables (3.4 of Text A)
- Undirected Graphs (4.1 of Text A)