https://algs4.cs.princeton.edu

3.3 Balanced Search Trees

- *2–3 search trees*
- *red–black BSTs (representation)*
- *red–black BSTs (operations)*
- *context*

Symbol table review

implementation	worst case			ordered ops?	key interface	emoji
	search	insert	delete			
sequential search (unordered list)	n	n	n		`equals()`	🙁
binary search (sorted array)	$\log n$	n	n	✔	`compareTo()`	😕
BST	n	n	n	✔	`compareTo()`	😕
goal	$\log n$	$\log n$	$\log n$	✔	`compareTo()`	😎

Challenge. $O(\log n)$ time in worst case.

optimized for teaching and coding (introduced in COS 226)

This lecture. 2–3 trees and left-leaning red-black BSTs.

co-invented by Bob Sedgewick in the 1970s

https://algs4.cs.princeton.edu

3.3 Balanced Search Trees

- *2–3 search trees*
- *red–black BSTs (representation)*
- *red–black BSTs (operations)*
- *context*

2-3 tree

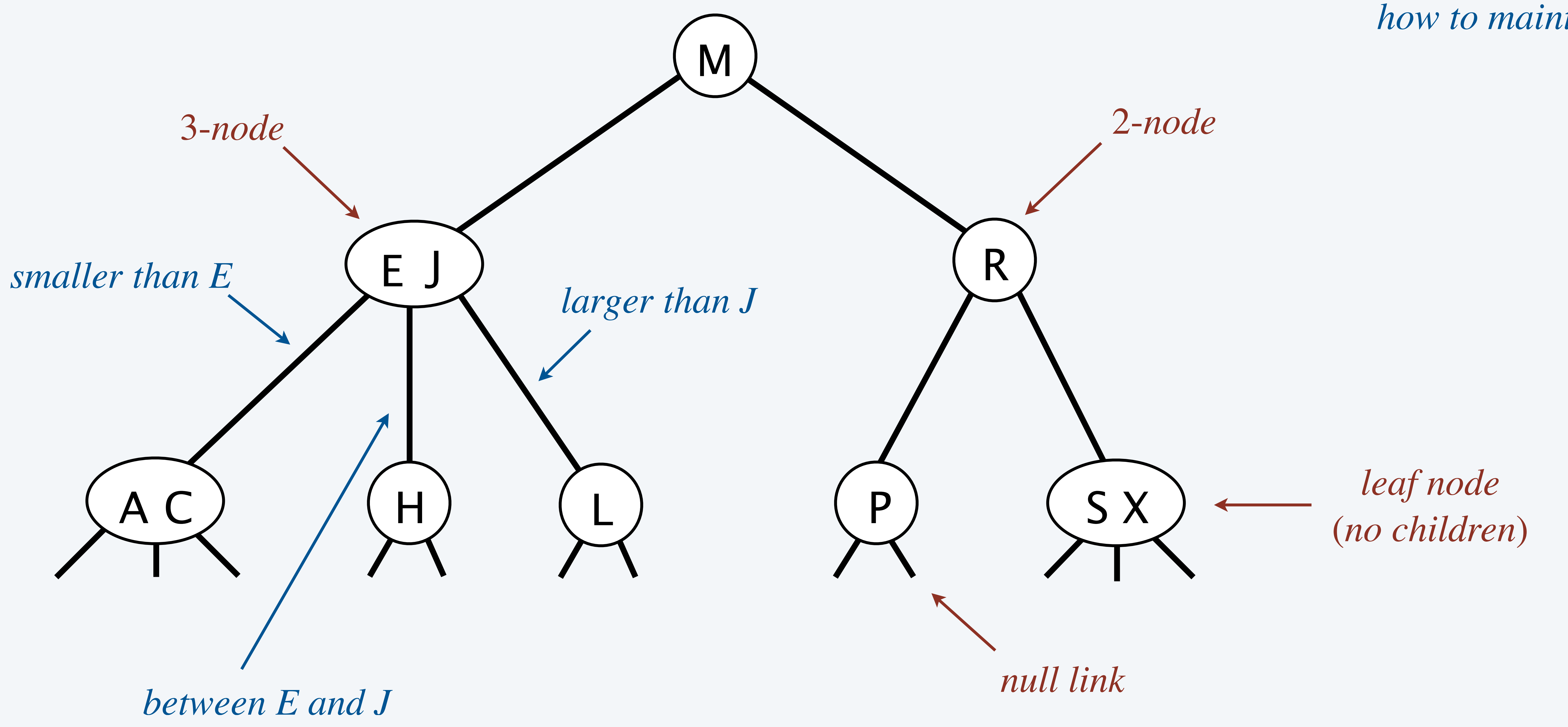
Each node contains either 1 or 2 keys.

- 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 the root to a leaf node has the same length.

how to maintain ?

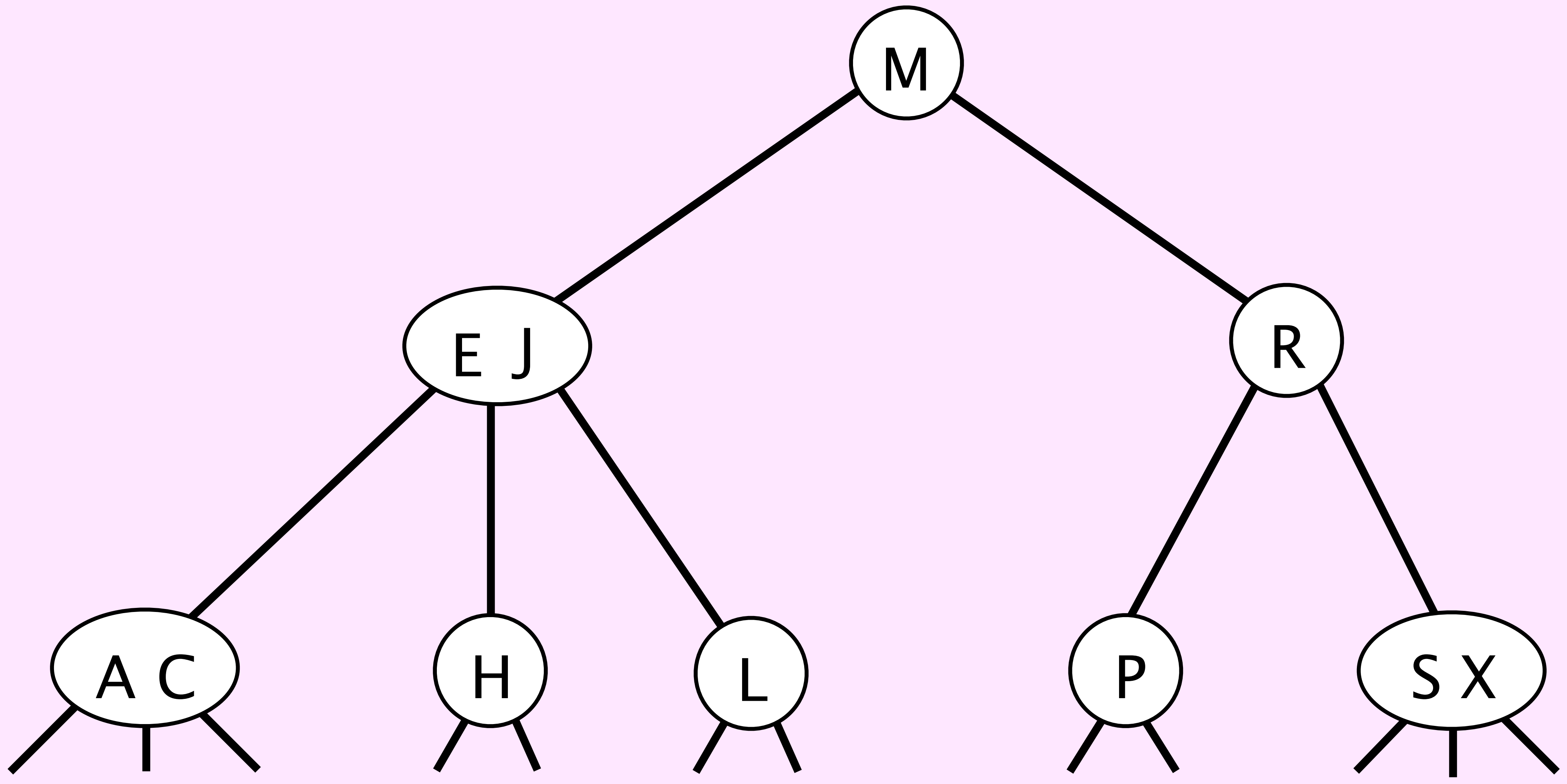


2–3 tree demo

Search.

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

search for H

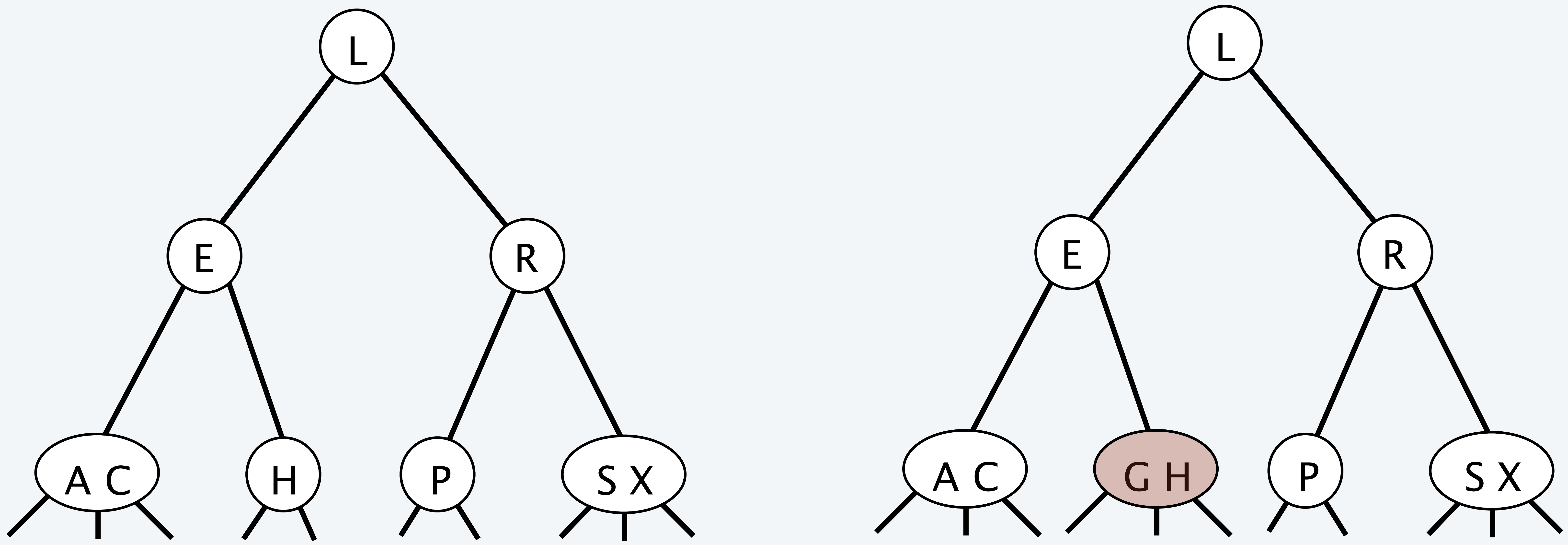


2-3 tree: insertion

Insertion into a 2-node at bottom.

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

insert G

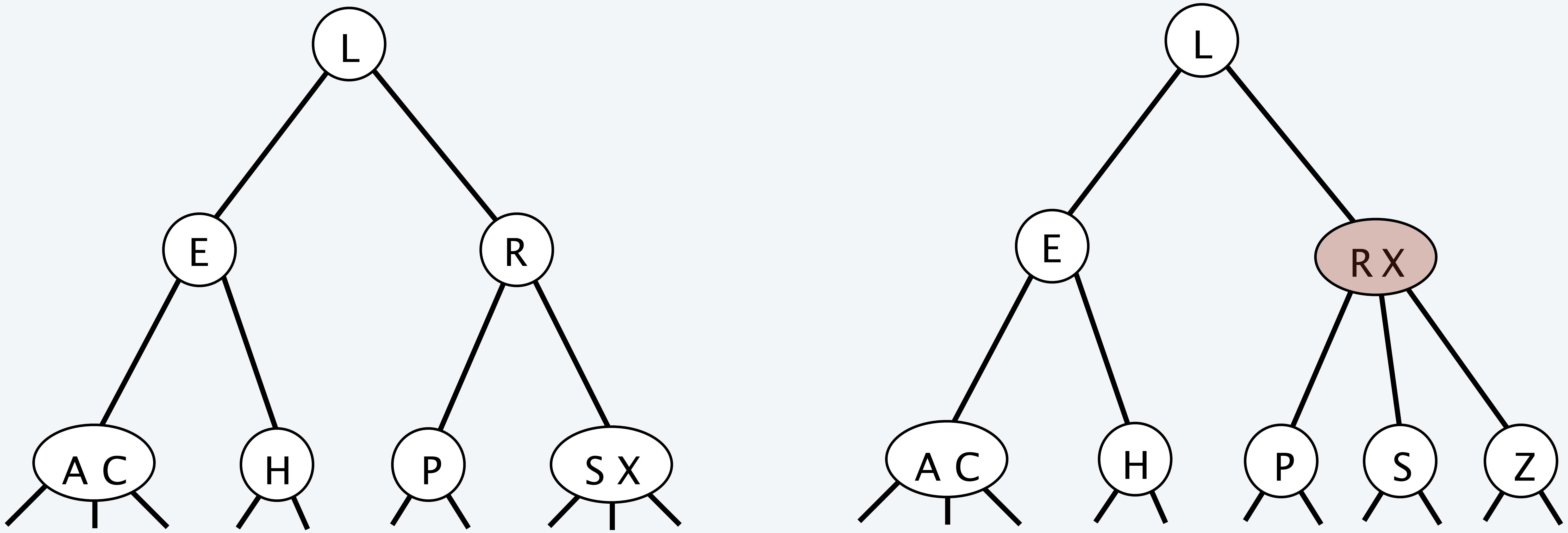


2-3 tree: insertion

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



Balanced search trees: poll 1

Suppose that you insert P into the following 2–3 tree.

What will be the root of the resulting 2–3 tree?

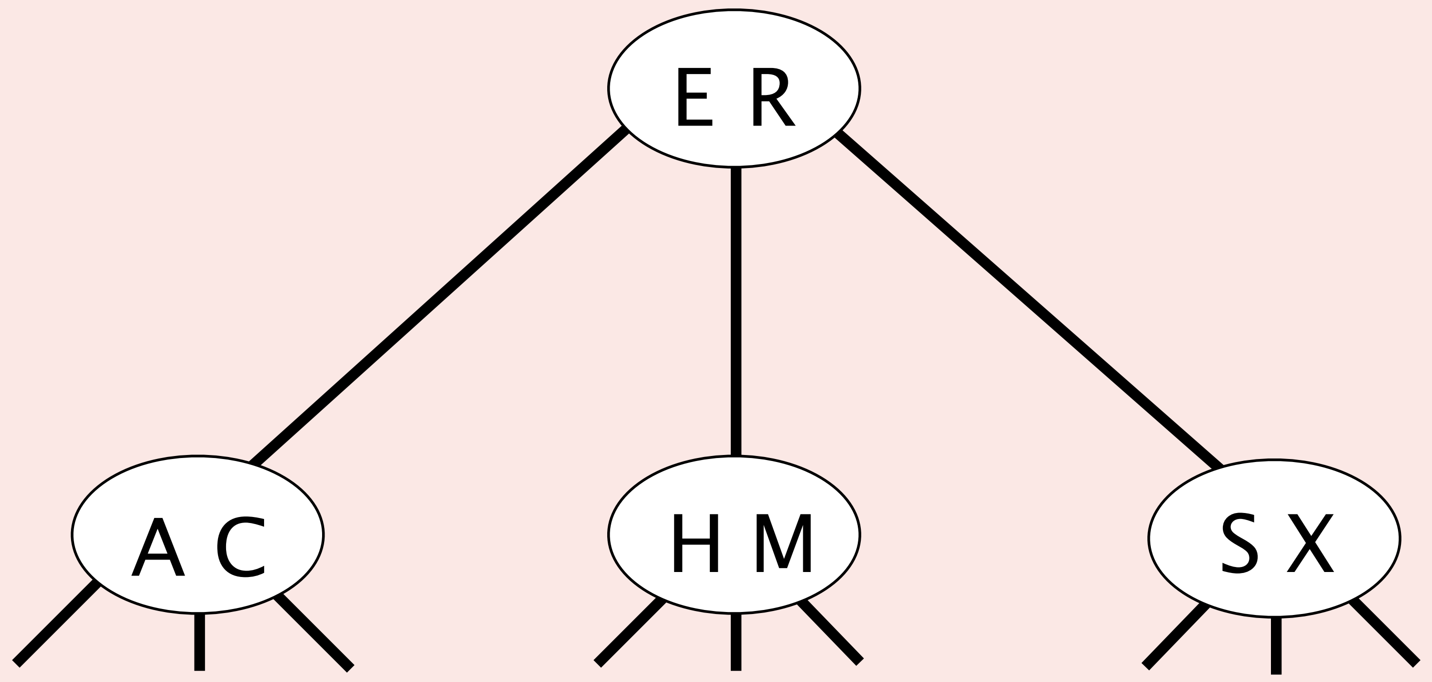
A. E

B. E R

C. M

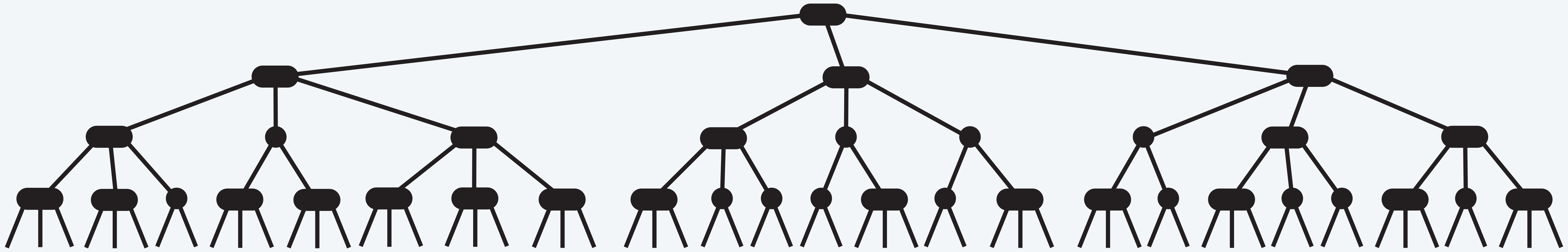
D. P

E. R



2–3 tree: performance

Perfect balance. Every path from the root to a leaf node has the same length.



Key property. The height of a 2–3 tree containing n keys is $\Theta(\log n)$.

- Min: $\sim \log_3 n \approx 0.631 \log_2 n$. [all 3-nodes]
- Max: $\sim \log_2 n$. [all 2-nodes]
- Between 18 and 30 for $n = 1$ billion keys.

Bottom line. Search and insert take $\Theta(\log n)$ time in the worst case.

ST implementations: summary

implementation	worst case			ordered ops?	key interface	emoji
	search	insert	delete			
sequential search (unordered list)	n	n	n		`equals()`	🙁
binary search (sorted array)	$\log n$	n	n	✔	`compareTo()`	😕
BST	n	n	n	✔	`compareTo()`	😕
2-3 trees	$\log n$	$\log n$	$\log n$	✔	`compareTo()`	😎

but hidden constant c is large
(depends upon implementation)

2-3 tree: implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Might need two compares to move one level 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 (see COS 326!), but there's a better way.

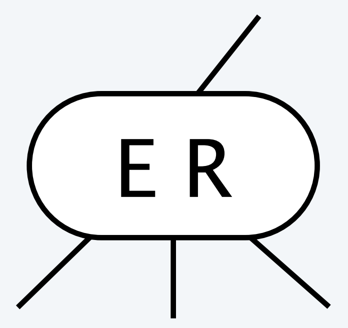
https://algs4.cs.princeton.edu

3.3 Balanced Search Trees

- 2–3 search trees
- *red–black BSTs (representation)*
- red–black BSTs (operations)
- context

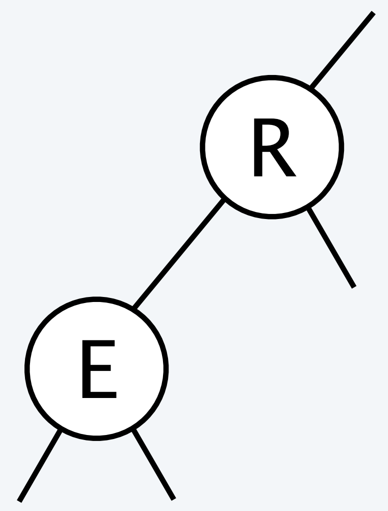
How to implement 2-3 trees as binary search trees?

Challenge. How to represent a 3 node?



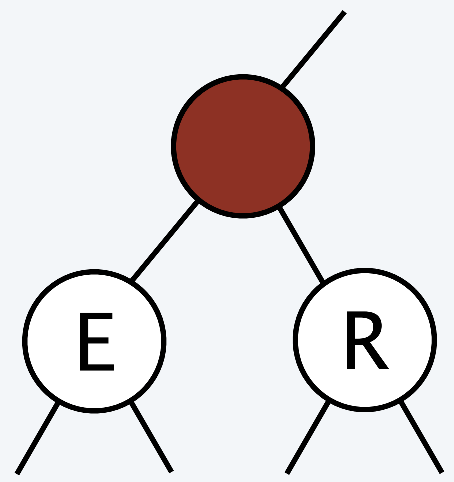
Approach 1. Two BST nodes.

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



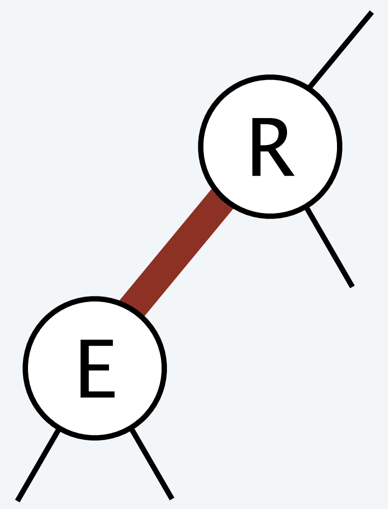
Approach 2. Two BST nodes, plus red "glue" node.

- Wastes space for extra node.
- Messy code.



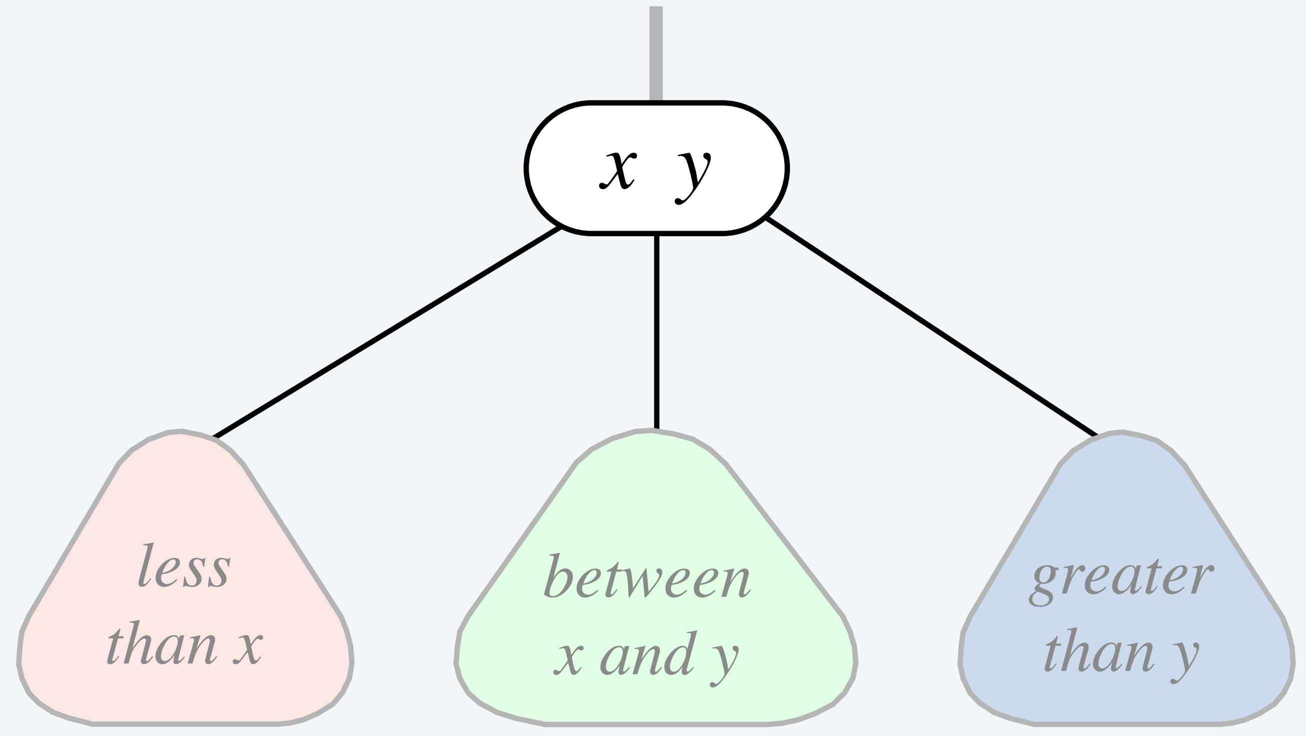
Approach 3. Two BST nodes, with red "glue" link.

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



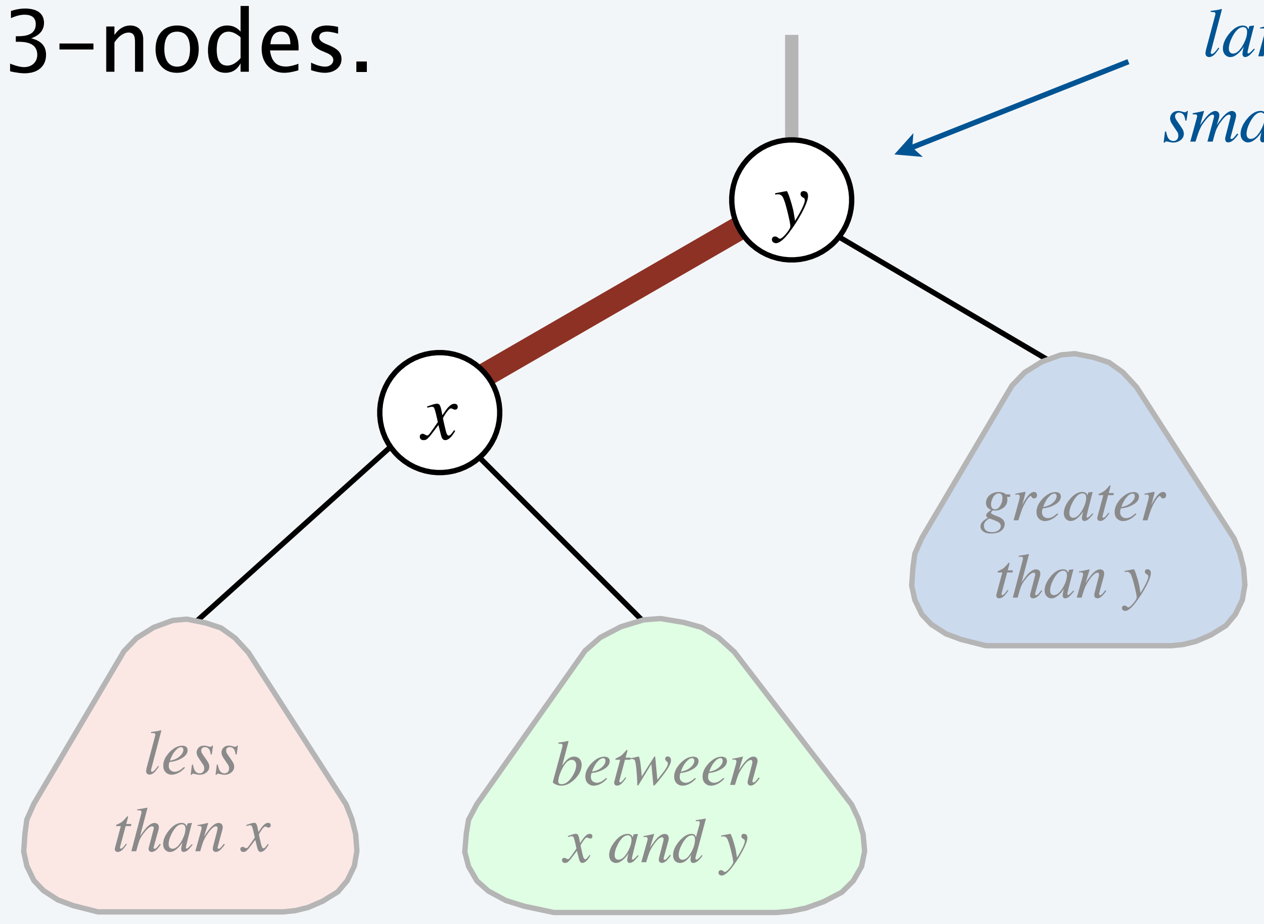
Left-leaning red–black BSTs

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

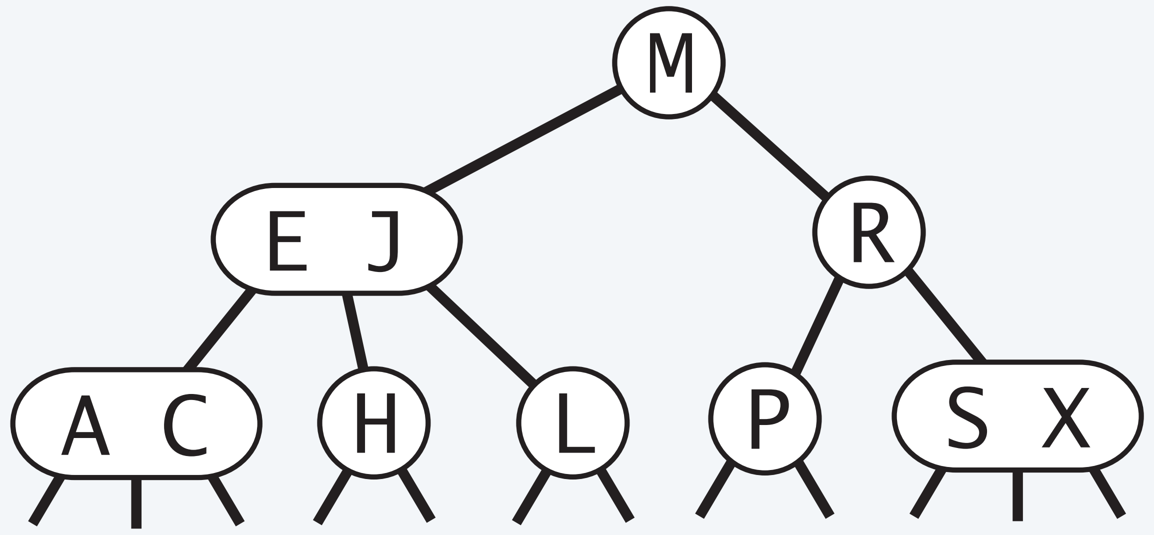


3-node in a 2–3 tree

larger key is parent; smaller key is left child



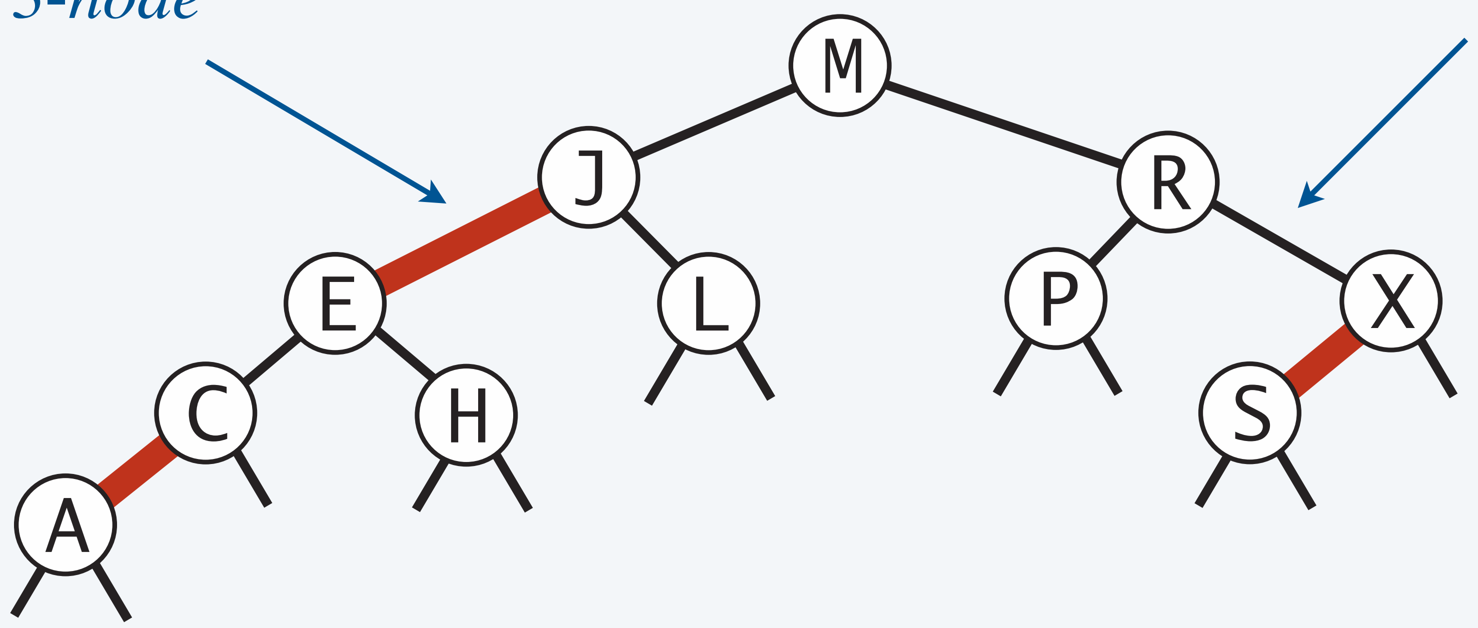
two nodes in the corresponding red–black BST



2–3 tree

red link “glues” the two BST nodes that correspond to a 3-node

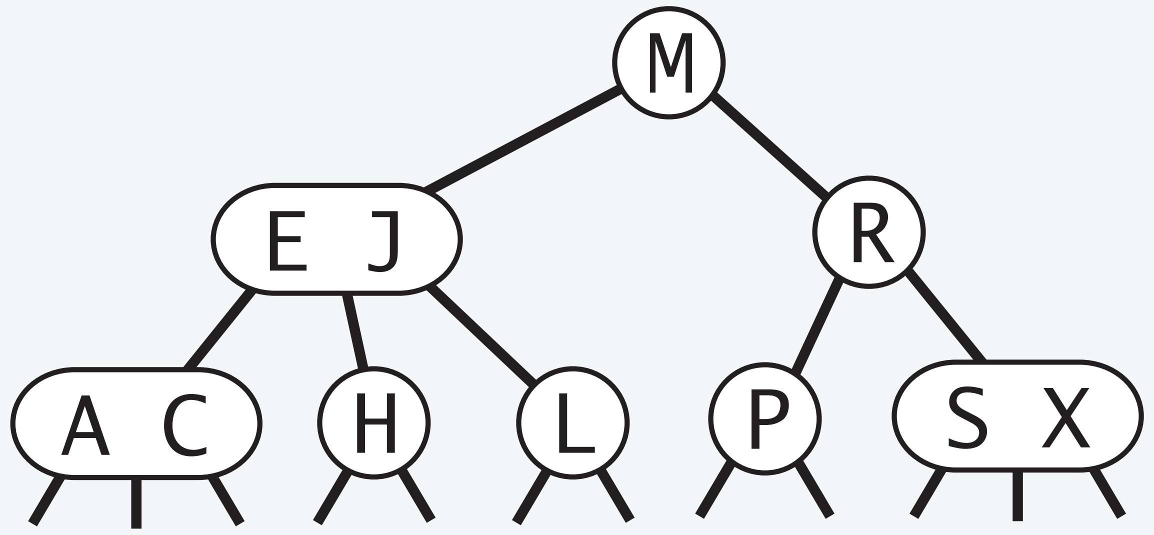
black links are the same as in 2–3 tree



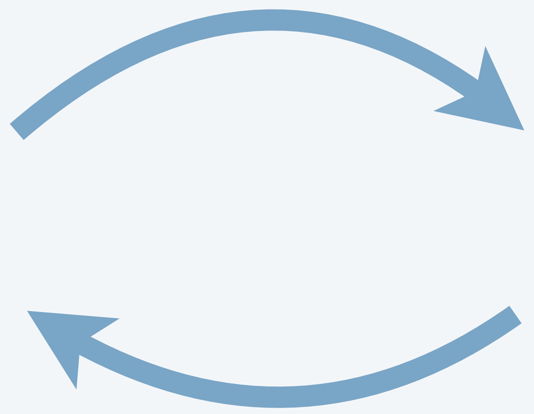
corresponding red–black BST

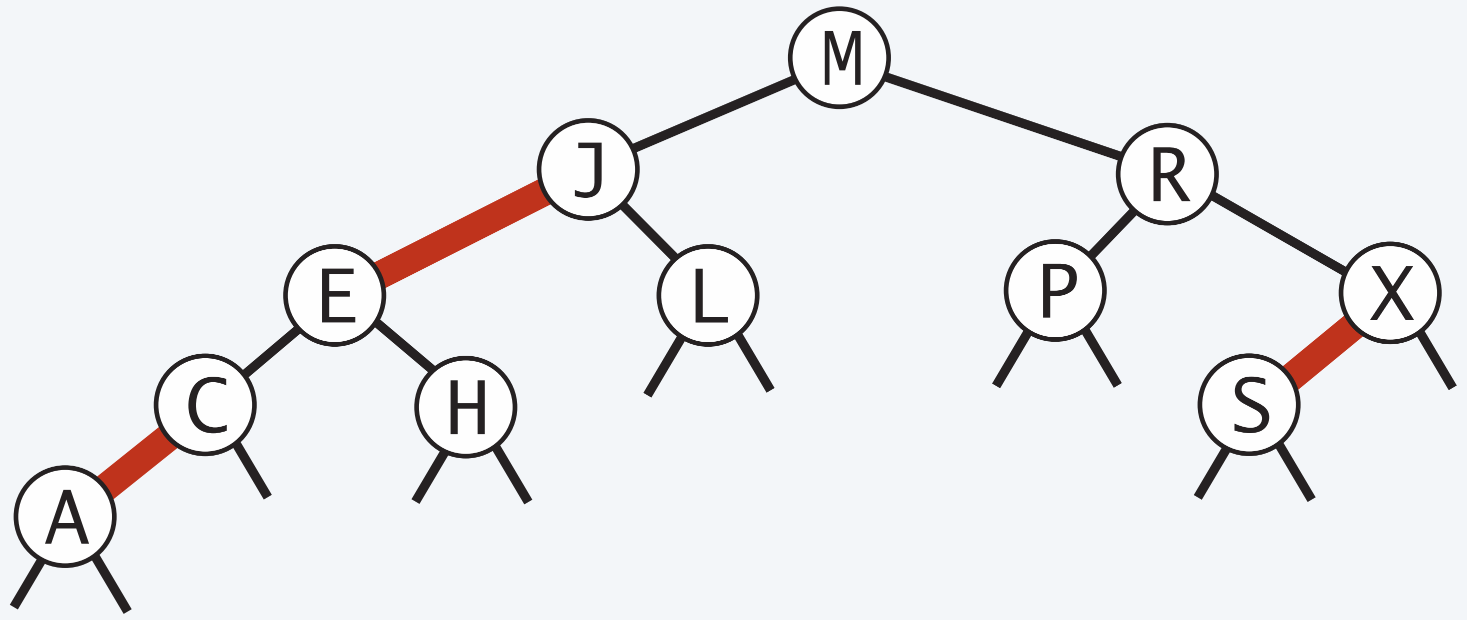
Left-leaning red–black BSTs

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



2–3 tree

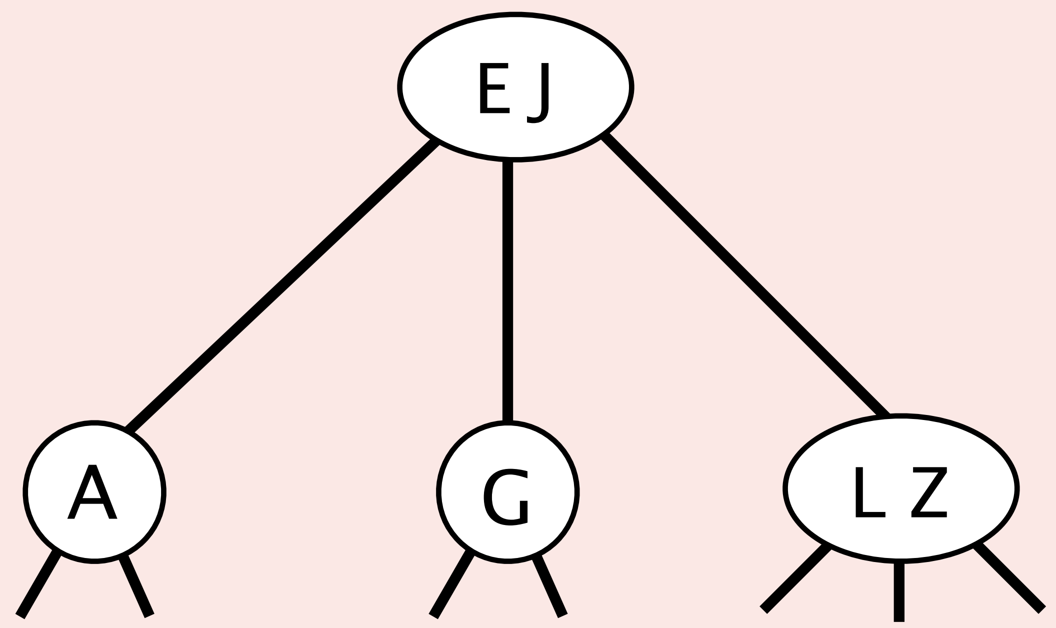




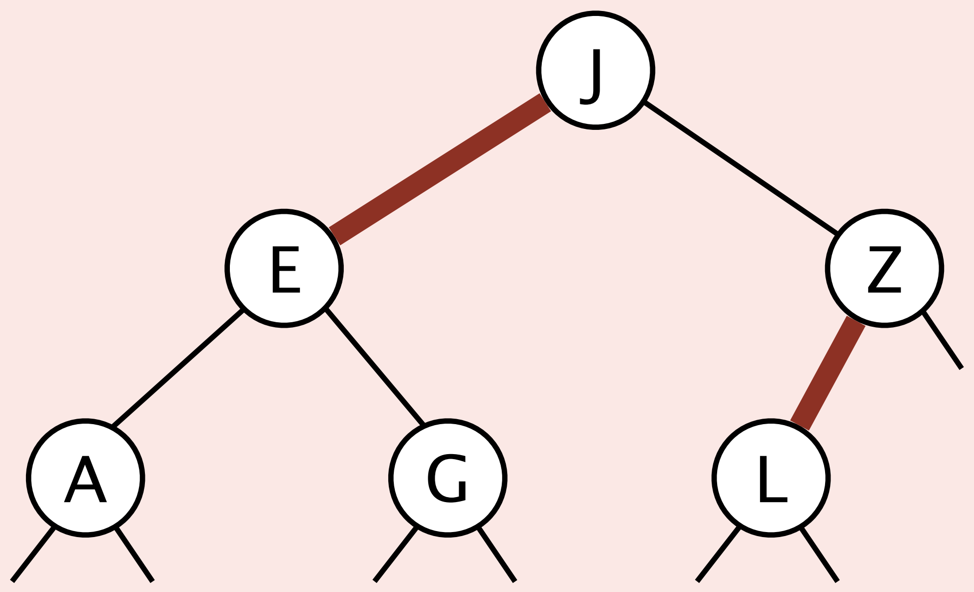
corresponding red–black BST

Balanced search trees: poll 3

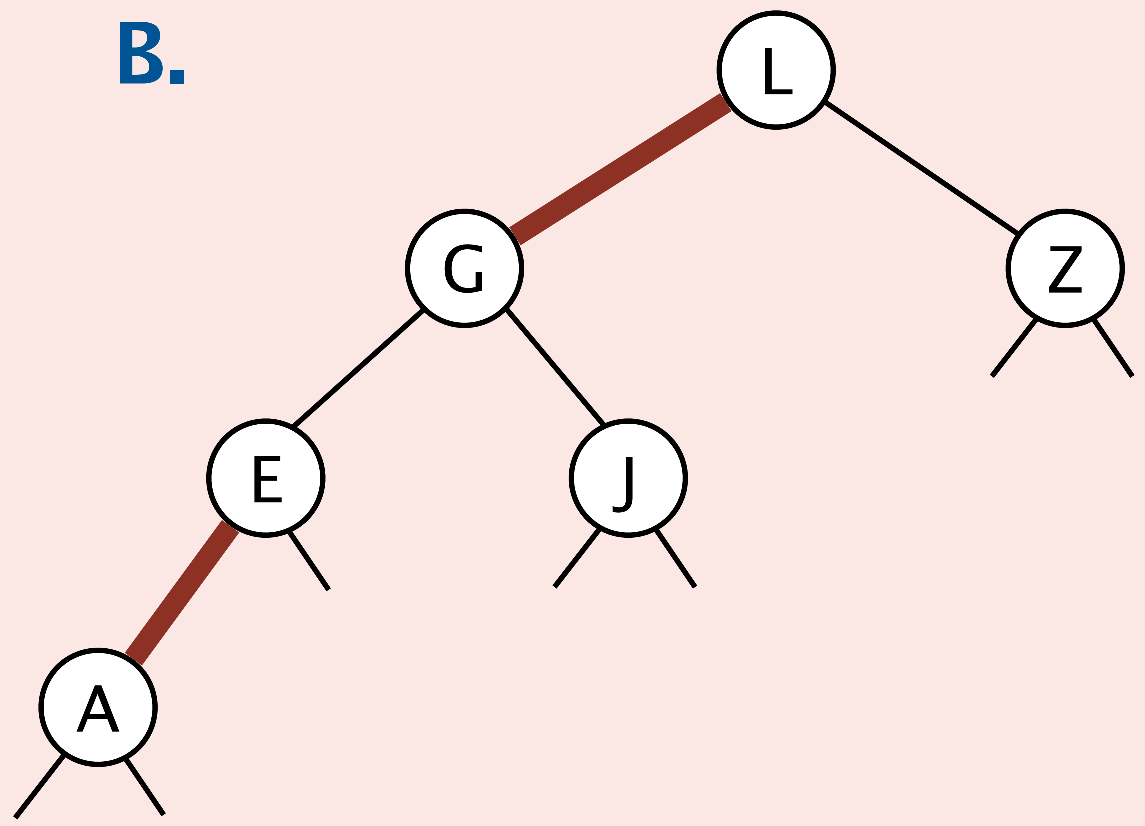
Which LLRB tree corresponds to the following 2–3 tree?



A.



B.



C. Both A and B.

D. Neither A nor B.

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

binary tree, symmetric order

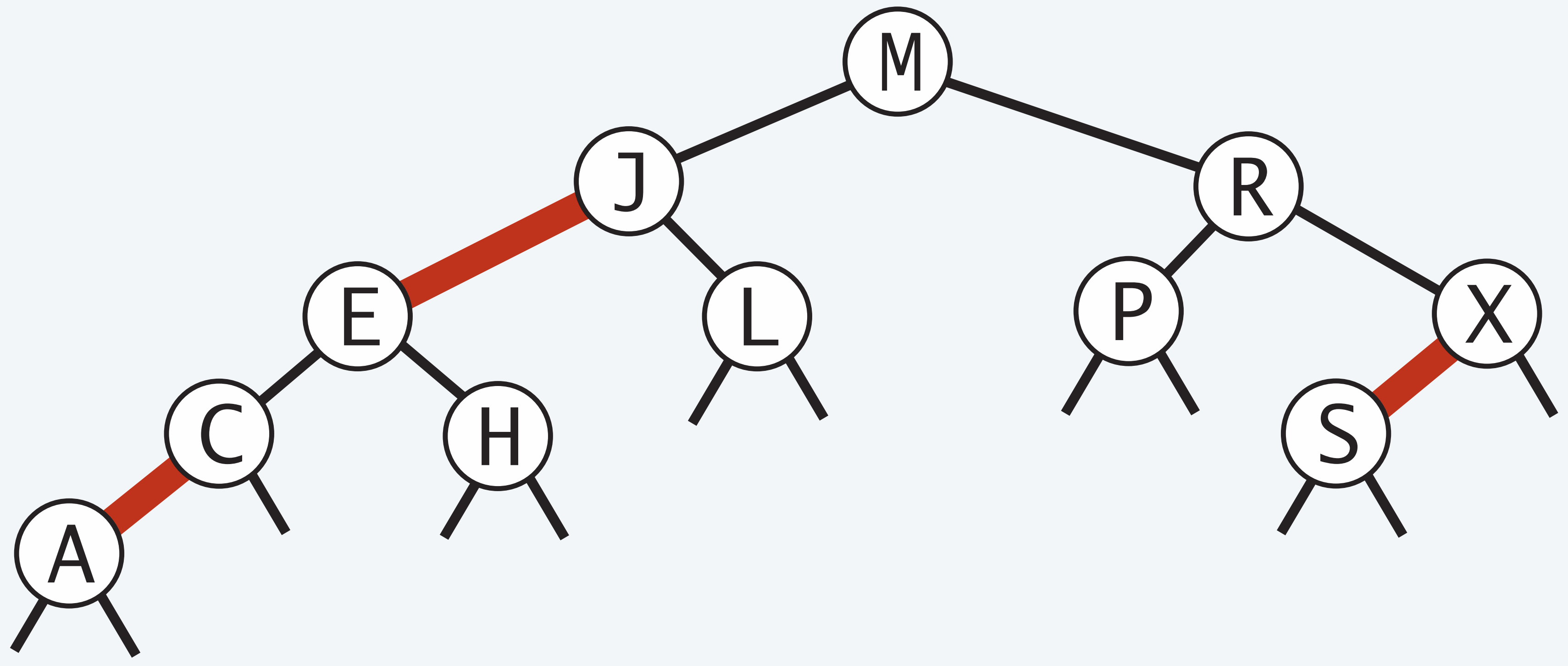
Def. A red–black BST is a BST such that:

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

color invariants

perfect black balance

black height



Balance in LLRB trees

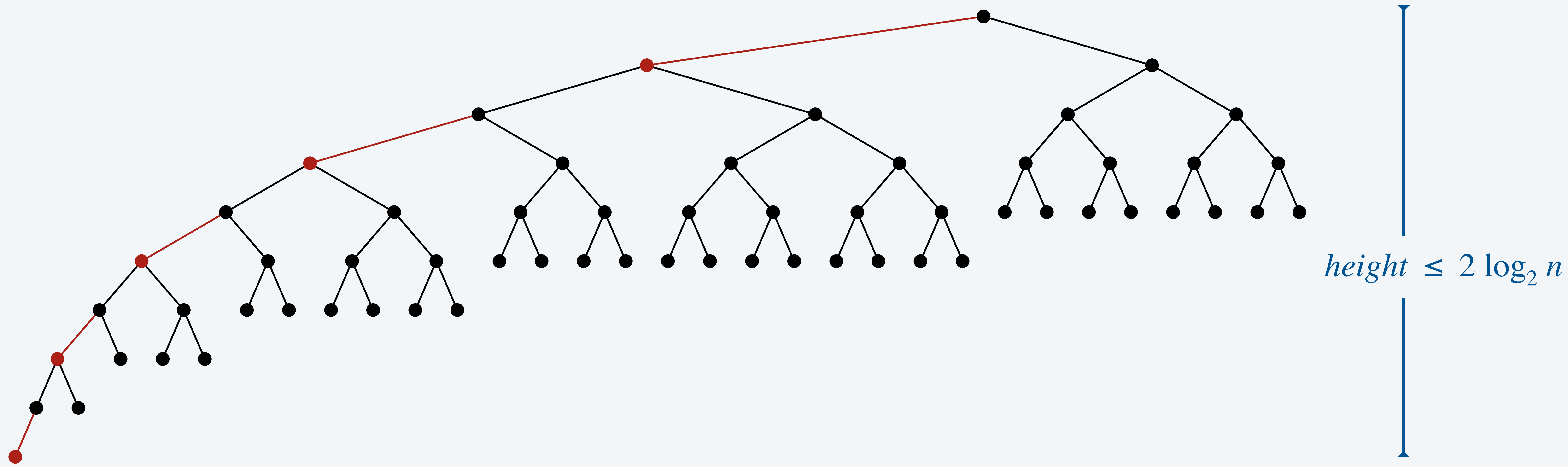
Proposition. Height of LLRB tree is $\leq 2 \log_2 n$.

Pf.

- Black height = height of corresponding 2–3 tree $\leq \log_2 n$.
- Never two red links in a row.

  $\Longrightarrow$ height of LLRB tree $\leq (2 \times \textit{black height}) + 1$

  $\leq 2 \log_2 n + 1.$
- [A more careful argument shows height $\leq 2 \log_2 n$.]



Red-black BST representation

Each node (except root) is pointed to by precisely one link (from its parent) $\Longrightarrow$ can encode color of links in child nodes.

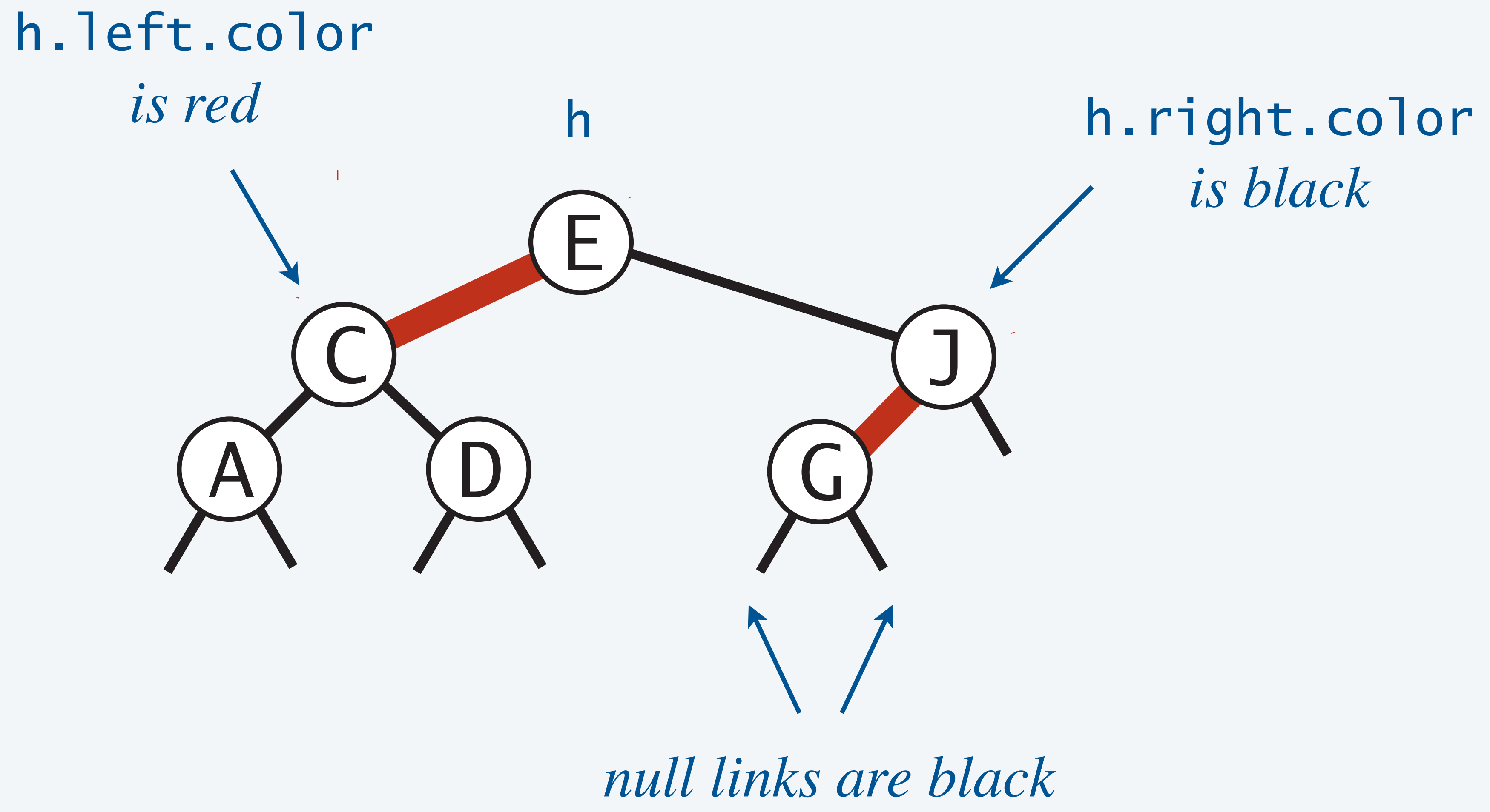
```
private static final boolean RED   = true;
private static final boolean BLACK = false;

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

color of parent link

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

by convention, null links are black



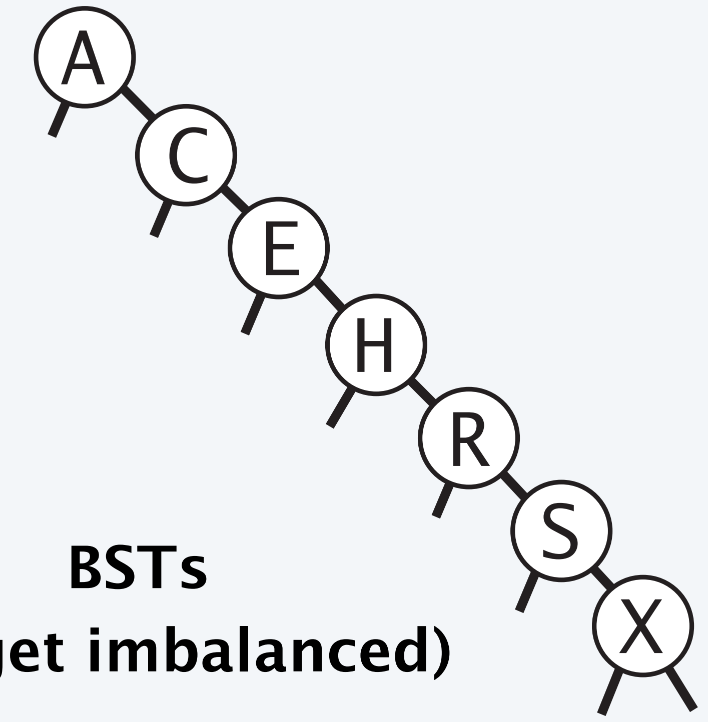
The red–black tree song (by Sean Sandys)

https://algs4.cs.princeton.edu

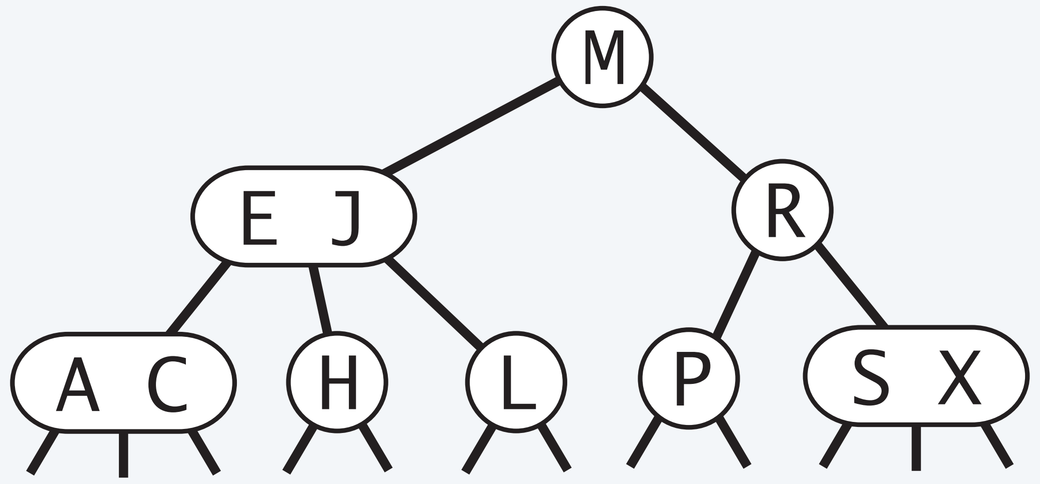
3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs (representation)
- red–black BSTs (operations)
- context

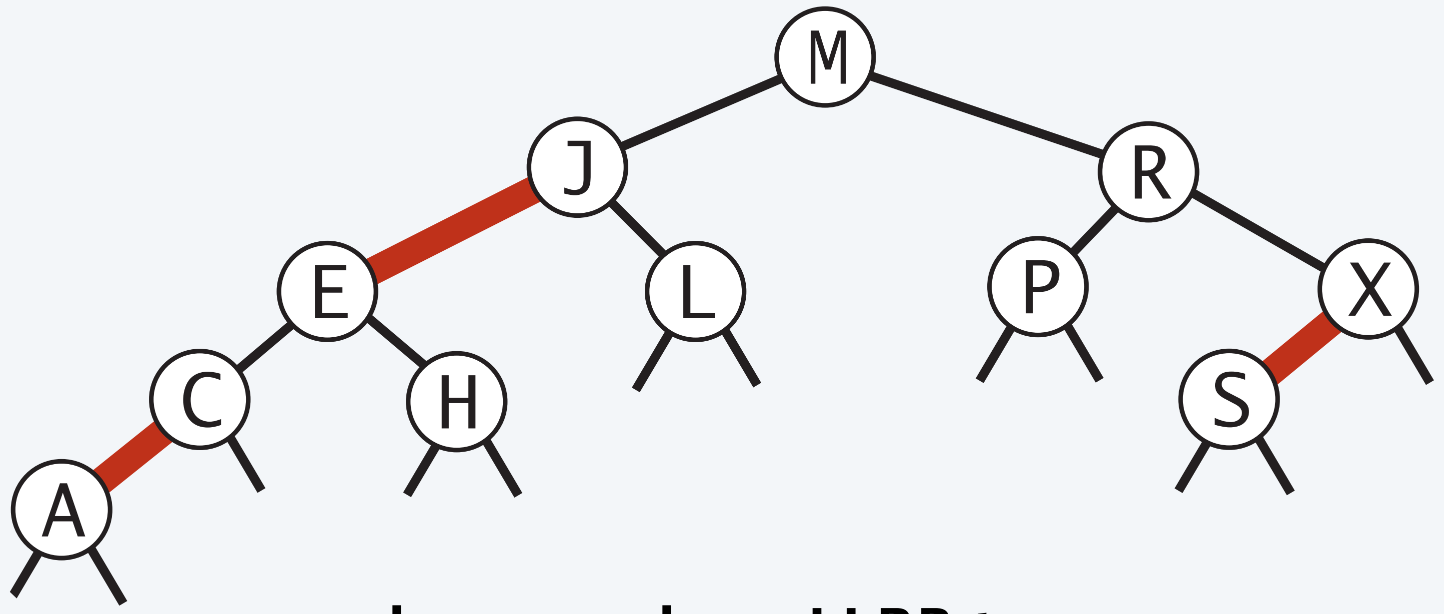
Review: the road to LLRB trees



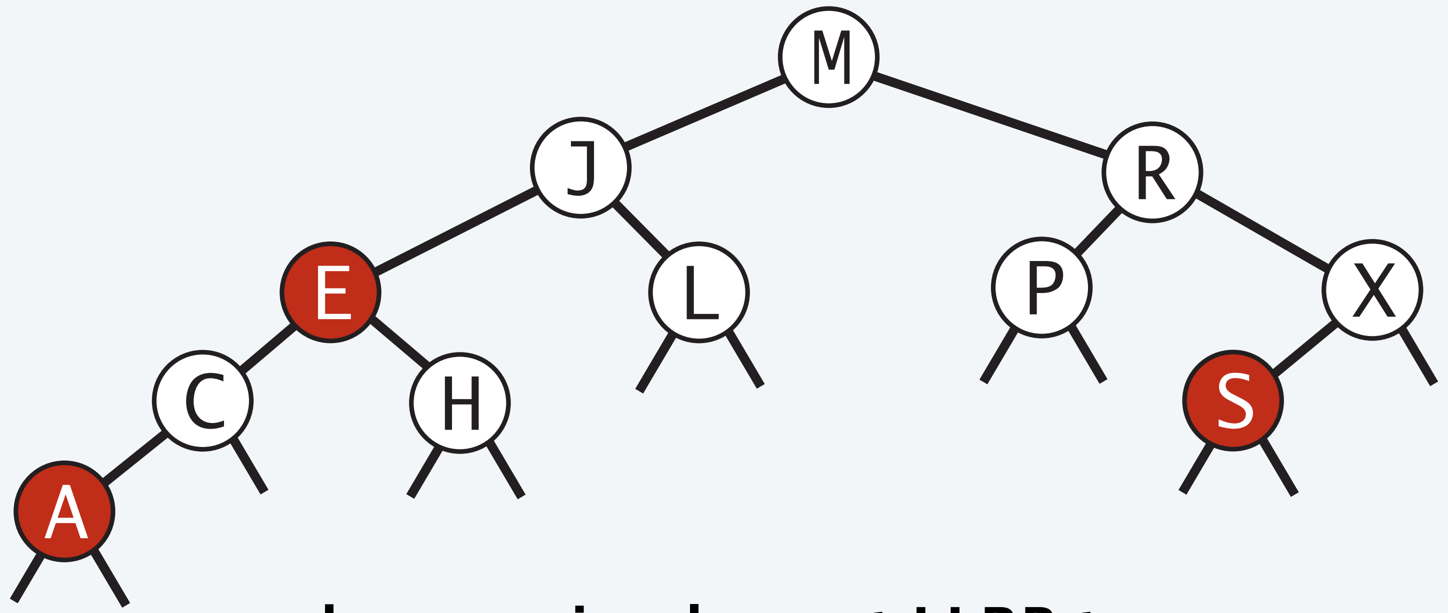
BSTs
(can get imbalanced)



2–3 trees
(balanced but awkward to implement)



how we draw LLRB trees
(color in links)



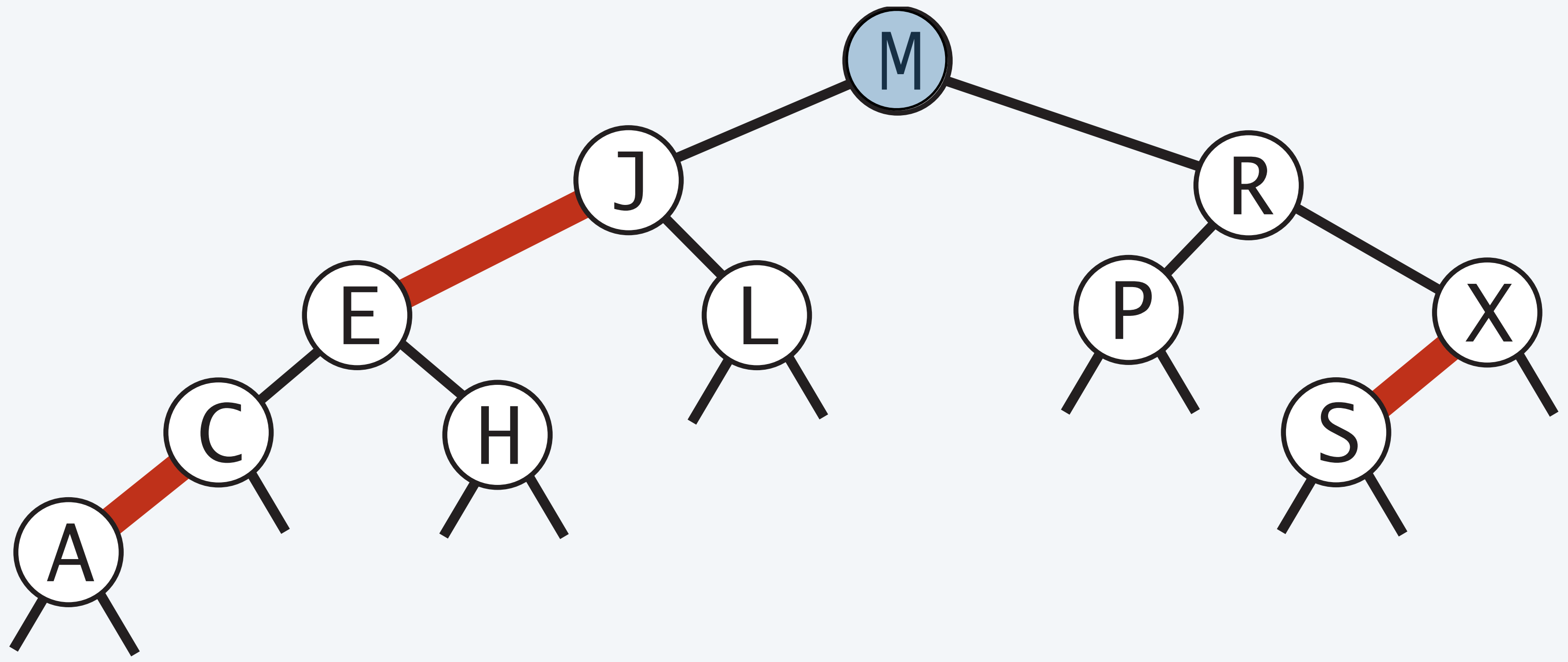
how we implement LLRB trees
(color in nodes)

Search in a red-black BST

Observation. Red-black BSTs are BSTs $\Longrightarrow$ search is the same as for BSTs (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 return x.val;
   }
   return null;
}
```



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

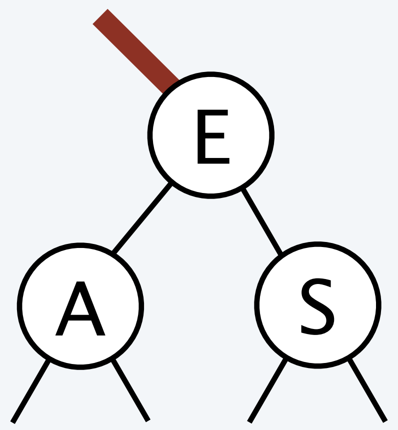
Insertion into a LLRB tree: overview

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

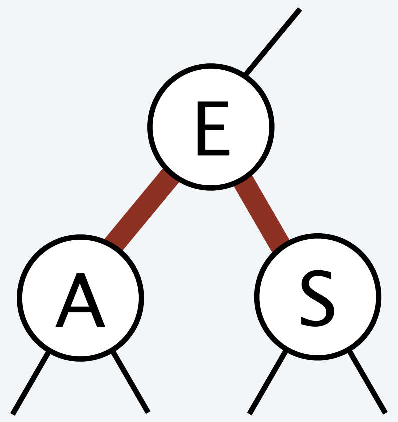
During internal operations, maintain:

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

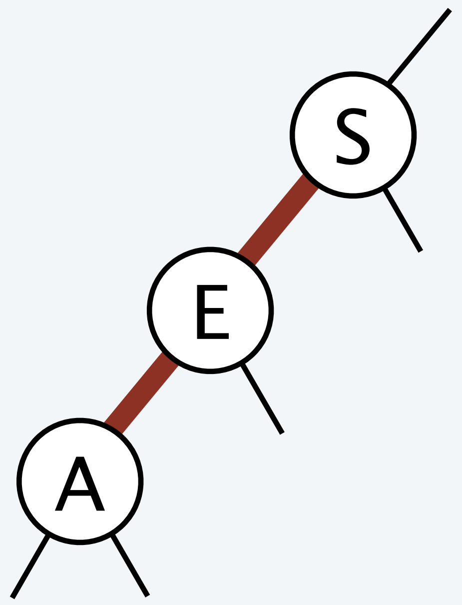
Example violations of color invariants:



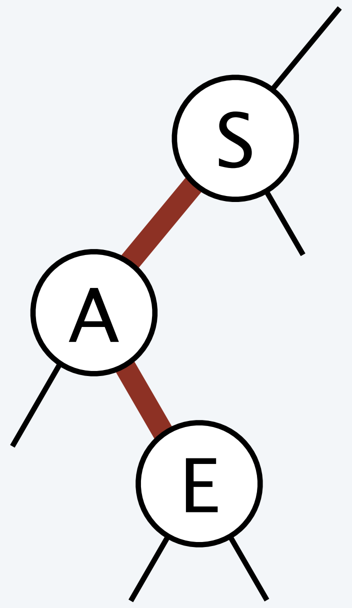
right-leaning red link



two red children (a temporary 4-node)



left-left red (a temporary 4-node)

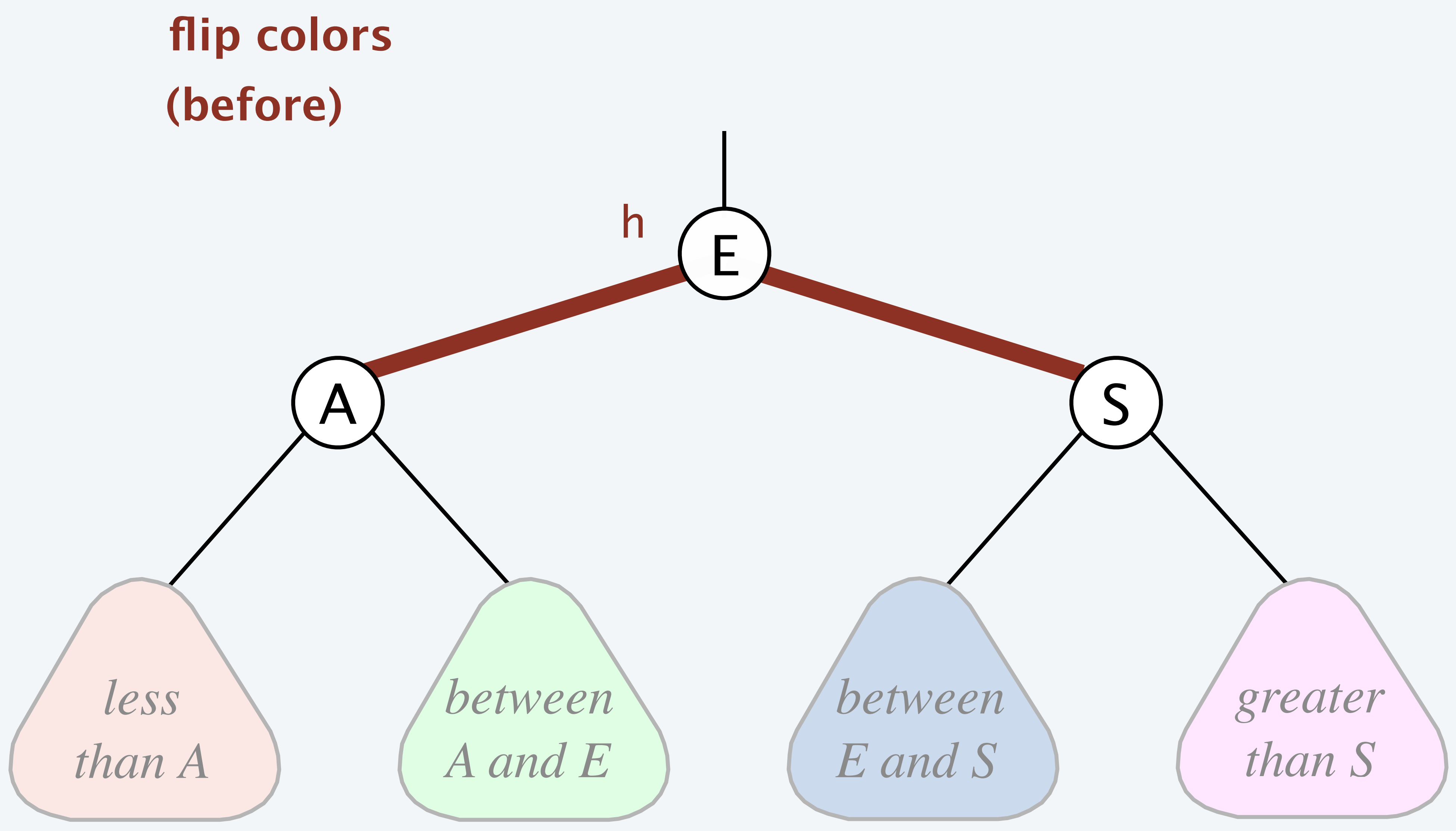


left-right red (a temporary 4-node)

To restore color invariants: perform color flips and rotations.

Elementary red-black BST operations

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

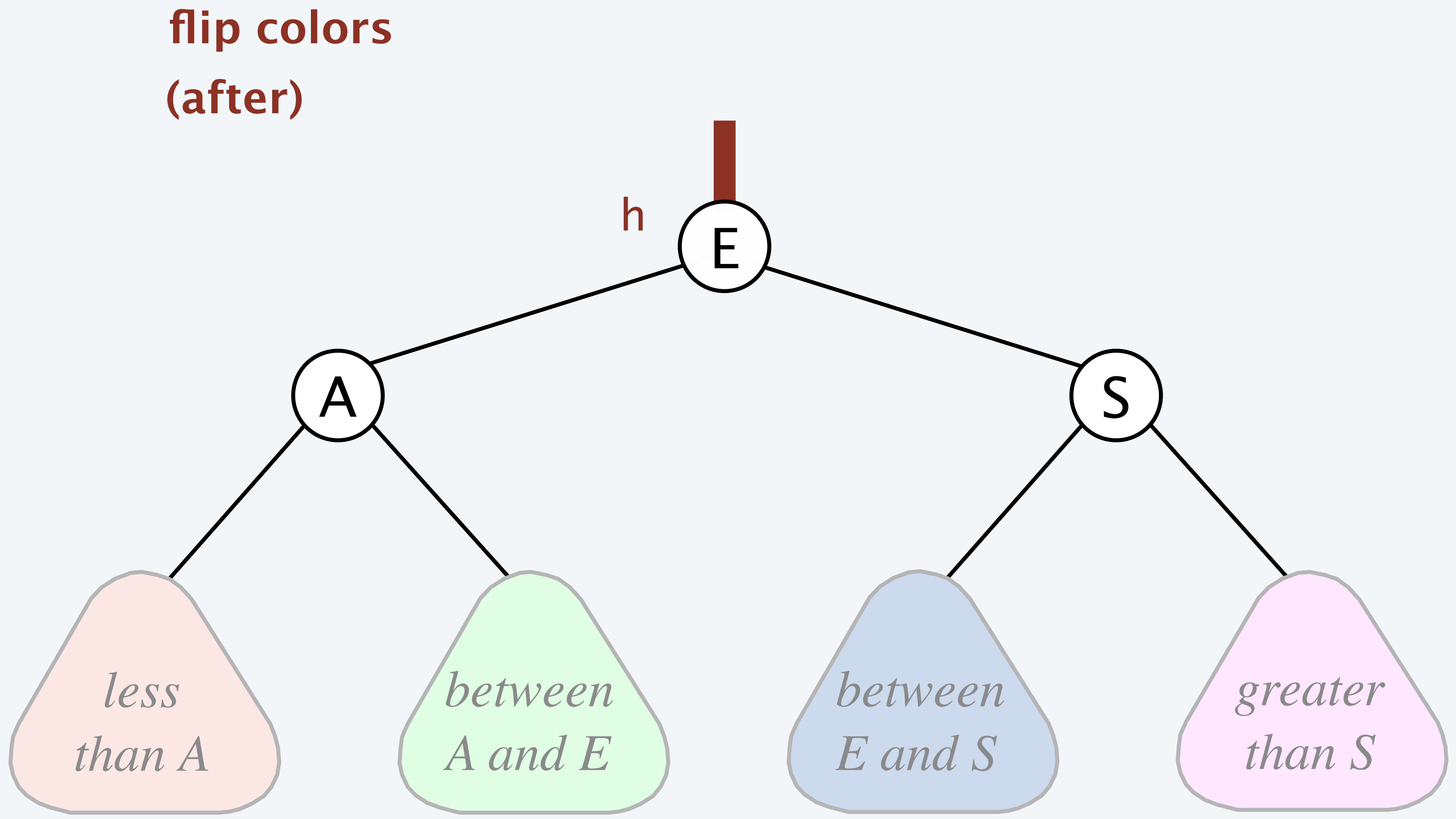



```
private void flipColors(Node h) {
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

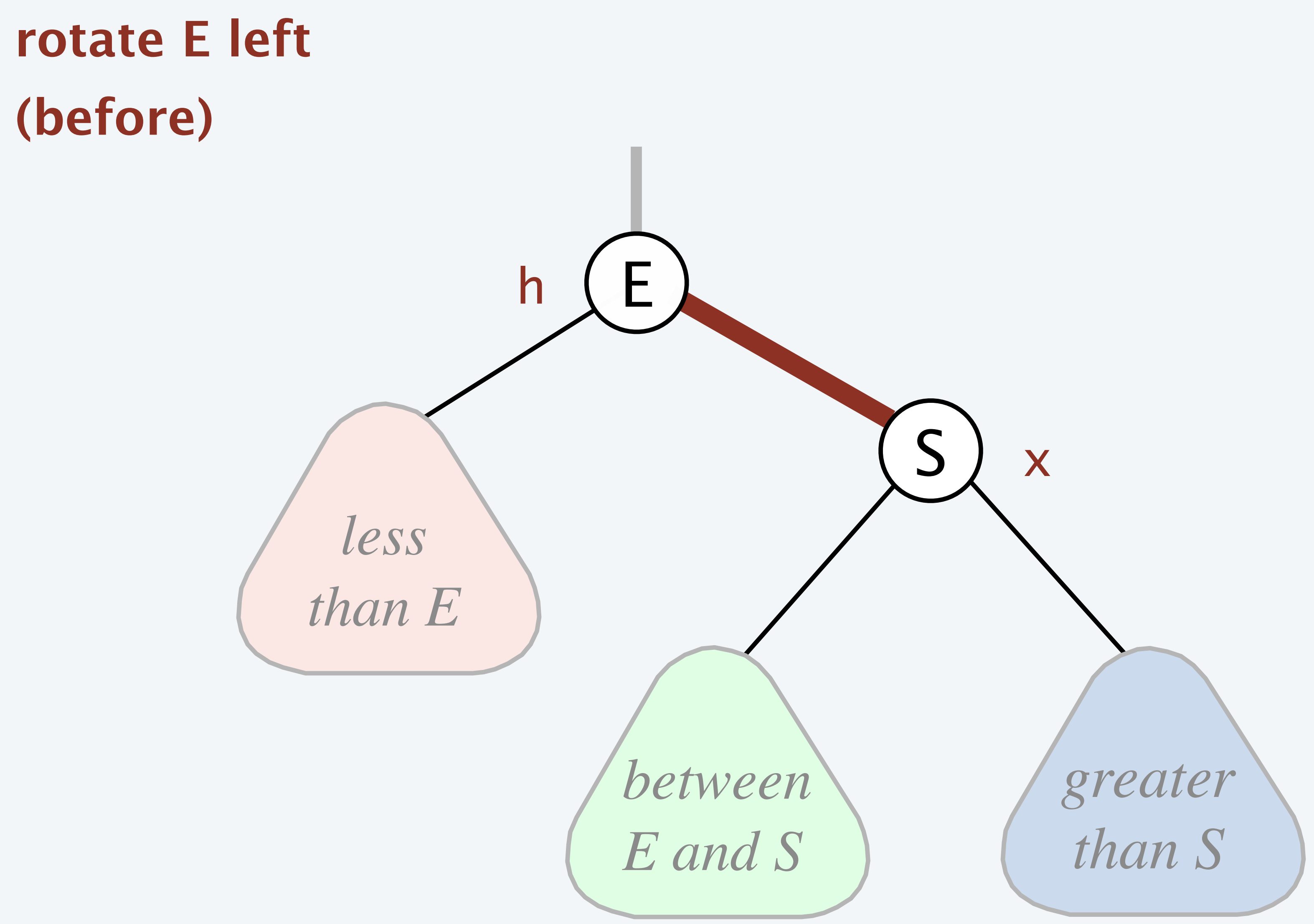



```
private void flipColors(Node h) {
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

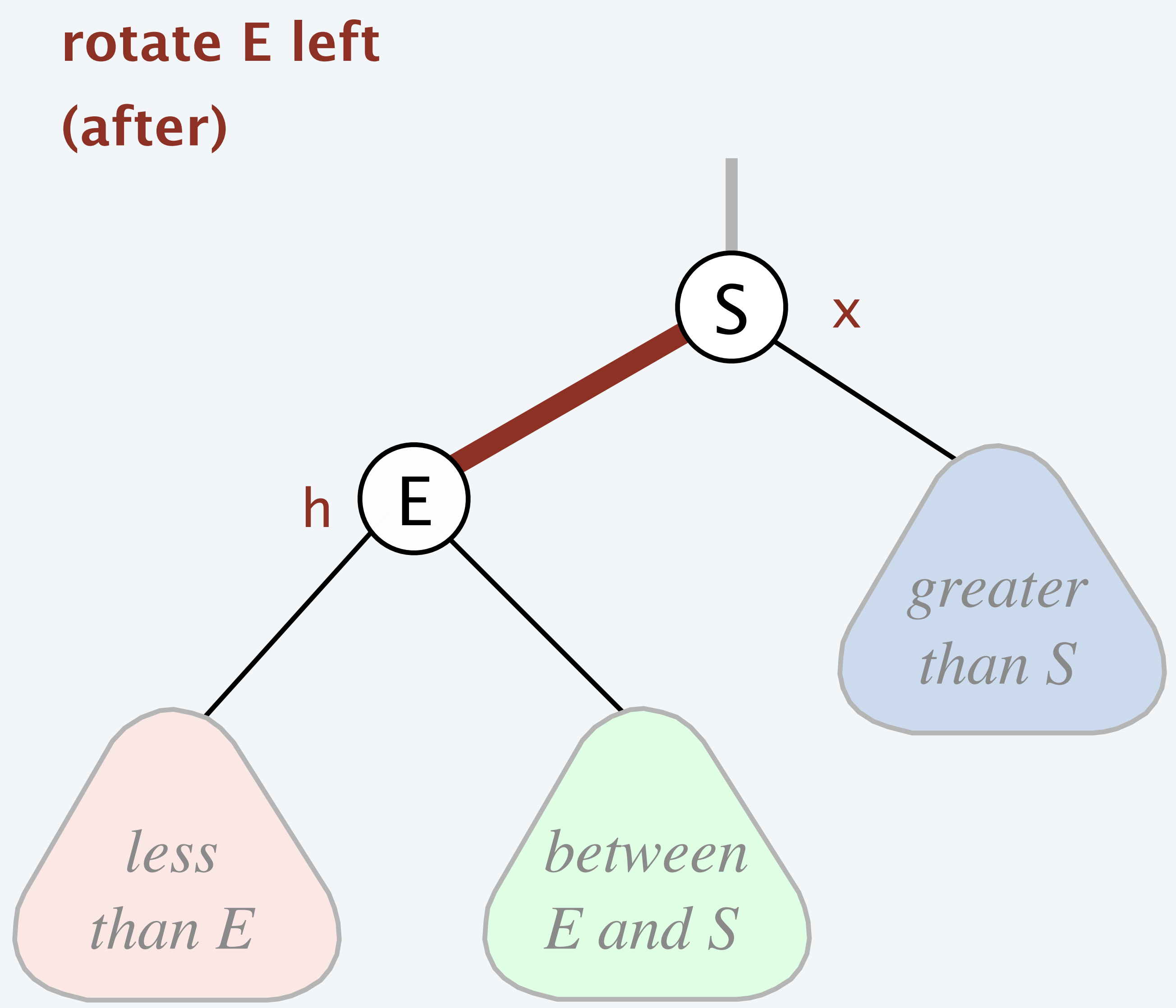



```
private Node rotateLeft(Node h) {
   assert !isRed(h.left);
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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.



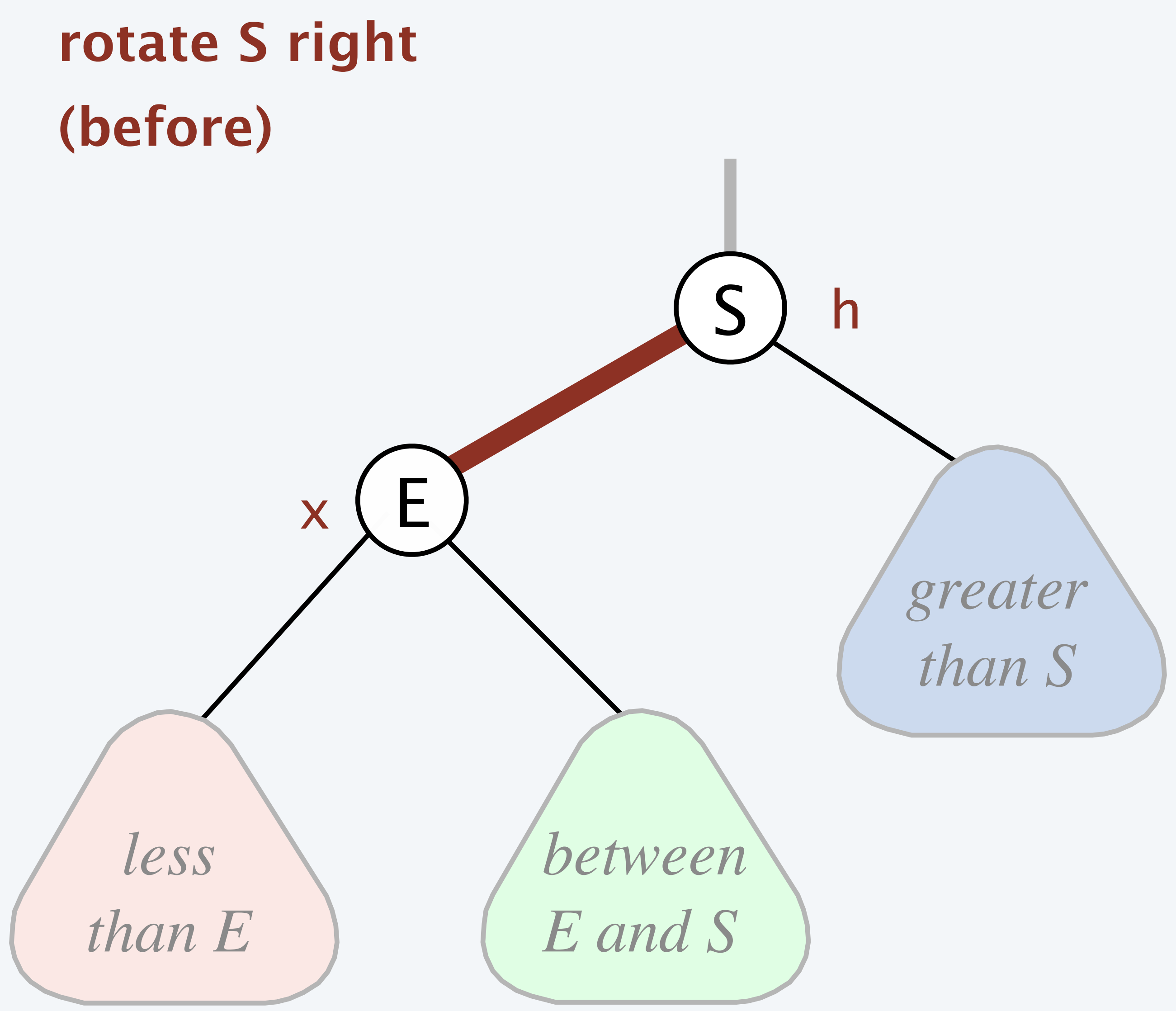

```
private Node rotateLeft(Node h) {
   assert !isRed(h.left);
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

returns root of resulting subtree
(typical call: `h = rotateLeft(h)` *)*

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.

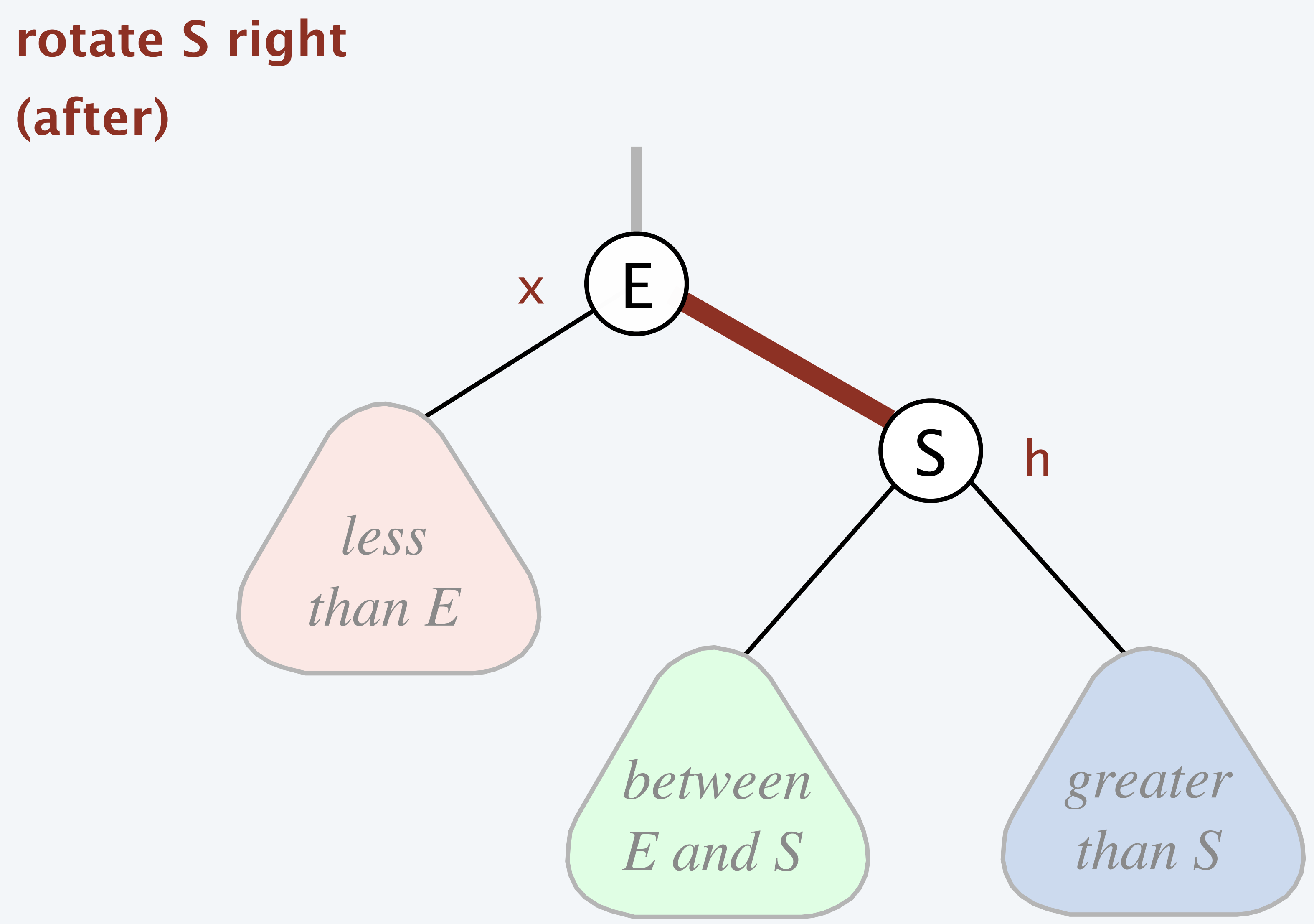



```
private Node rotateRight(Node h) {
   assert isRed(h.left);
   assert !isRed(h.right);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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.

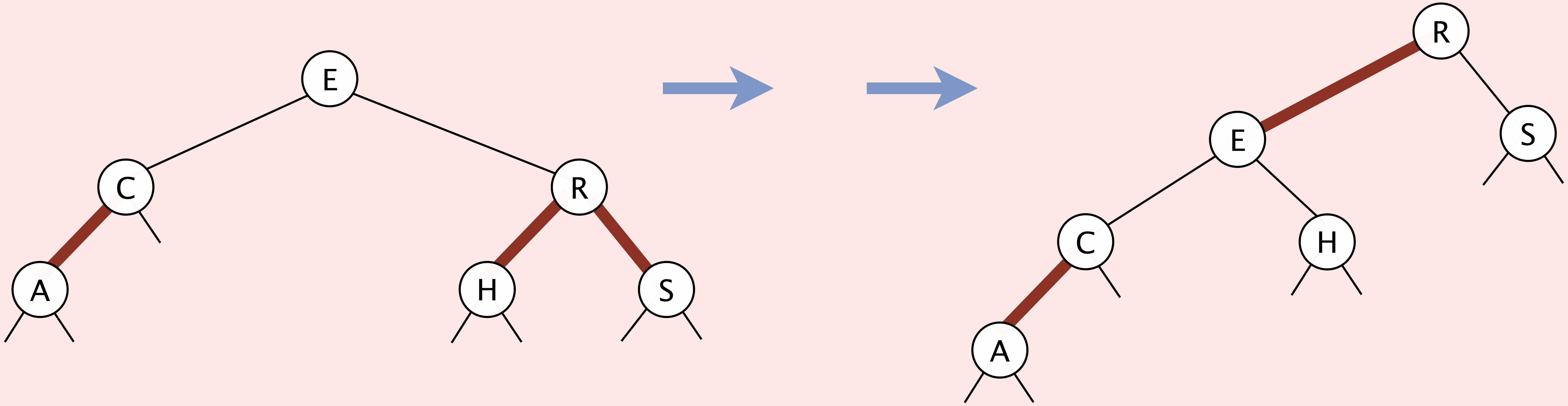



```
private Node rotateRight(Node h) {
   assert isRed(h.left);
   assert !isRed(h.right);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Balanced search trees: poll 4

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



A. Color flip E; left rotate R.

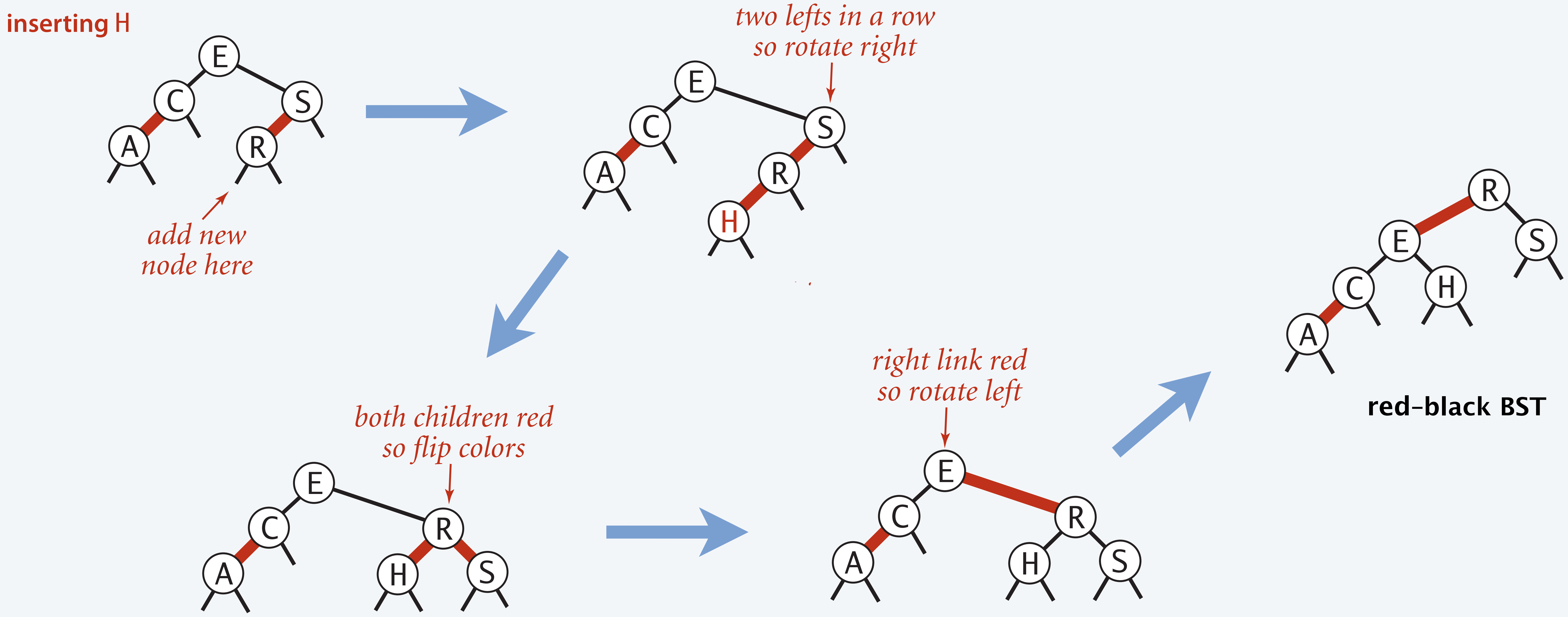
B. Color flip R; left rotate E.

C. Color flip R; left rotate R.

D. Color flip R; right rotate E.

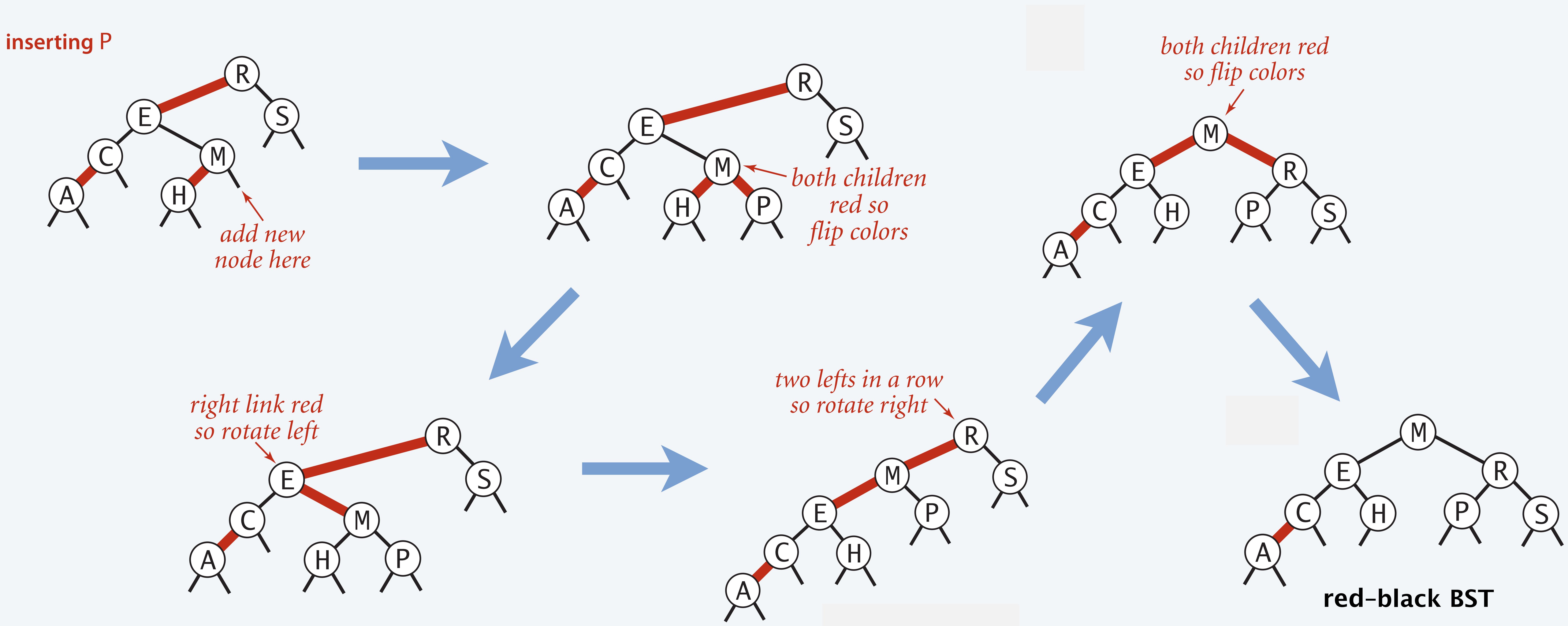
Insertion into a LLRB tree

- Do standard BST leaf insertion and color new link red. ← *to preserve symmetric order and 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
  - only right link red? ⇒ rotate left



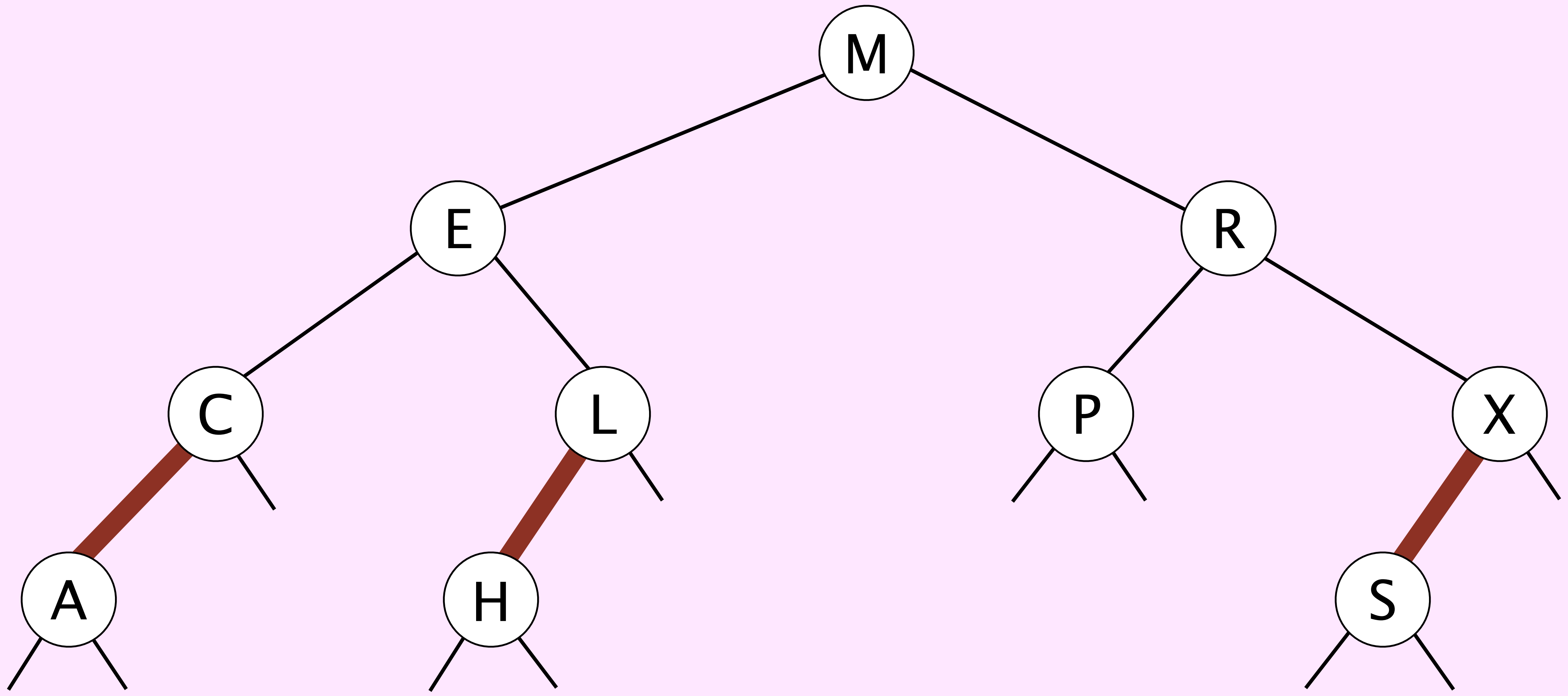
Insertion into a LLRB tree

- Do standard BST leaf insertion and 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
  - only right link red? $\Rightarrow$ rotate left



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

```java
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 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;
}
```

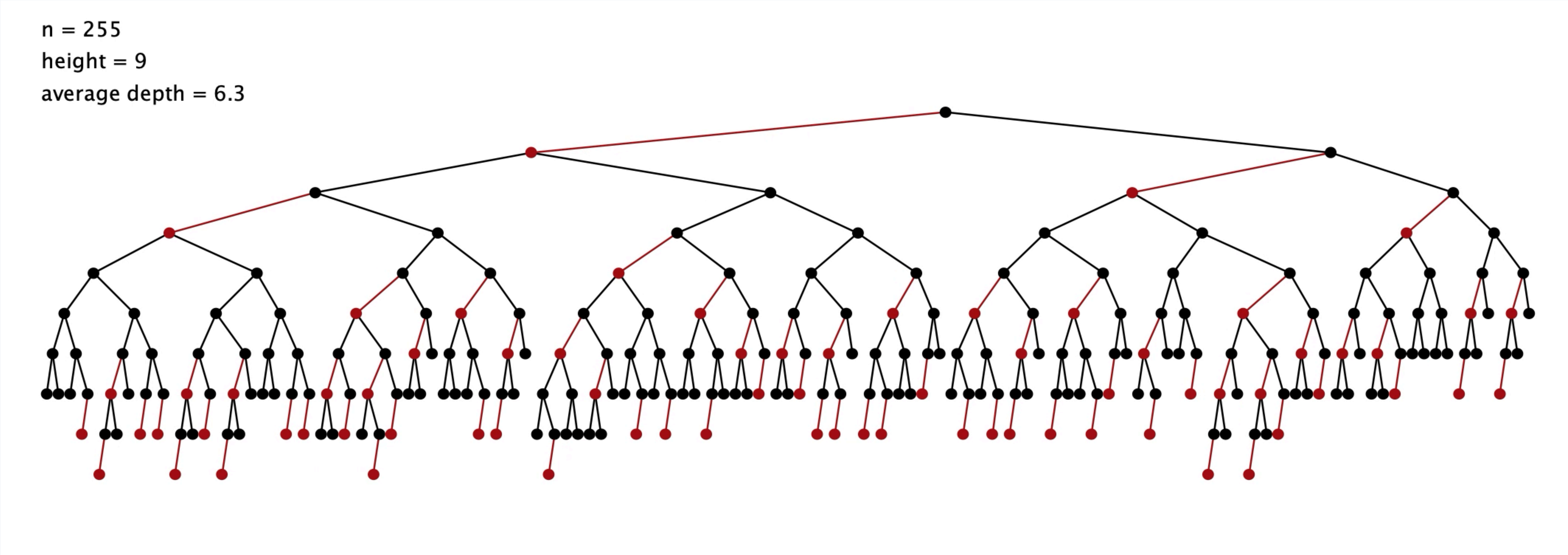
insert at bottom (and color it red)

each method that changes the tree shape returns the root of the resulting subtree

restore color invariants

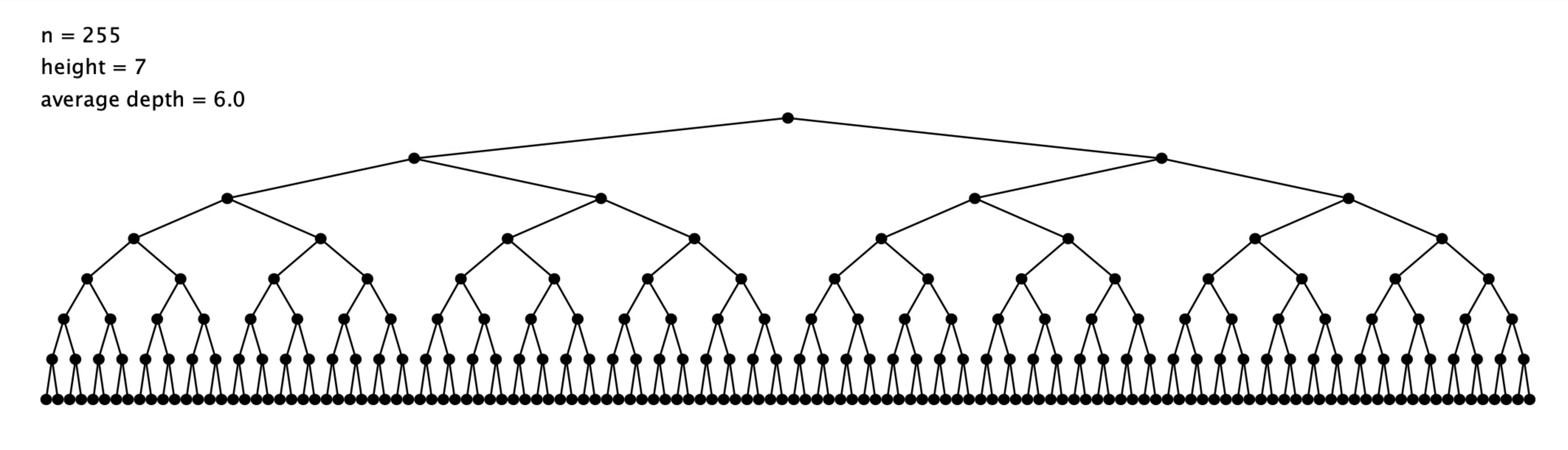
only a few extra lines of code guarantees $\Theta(\log n)$ height

Insertion into a LLRB tree: visualization



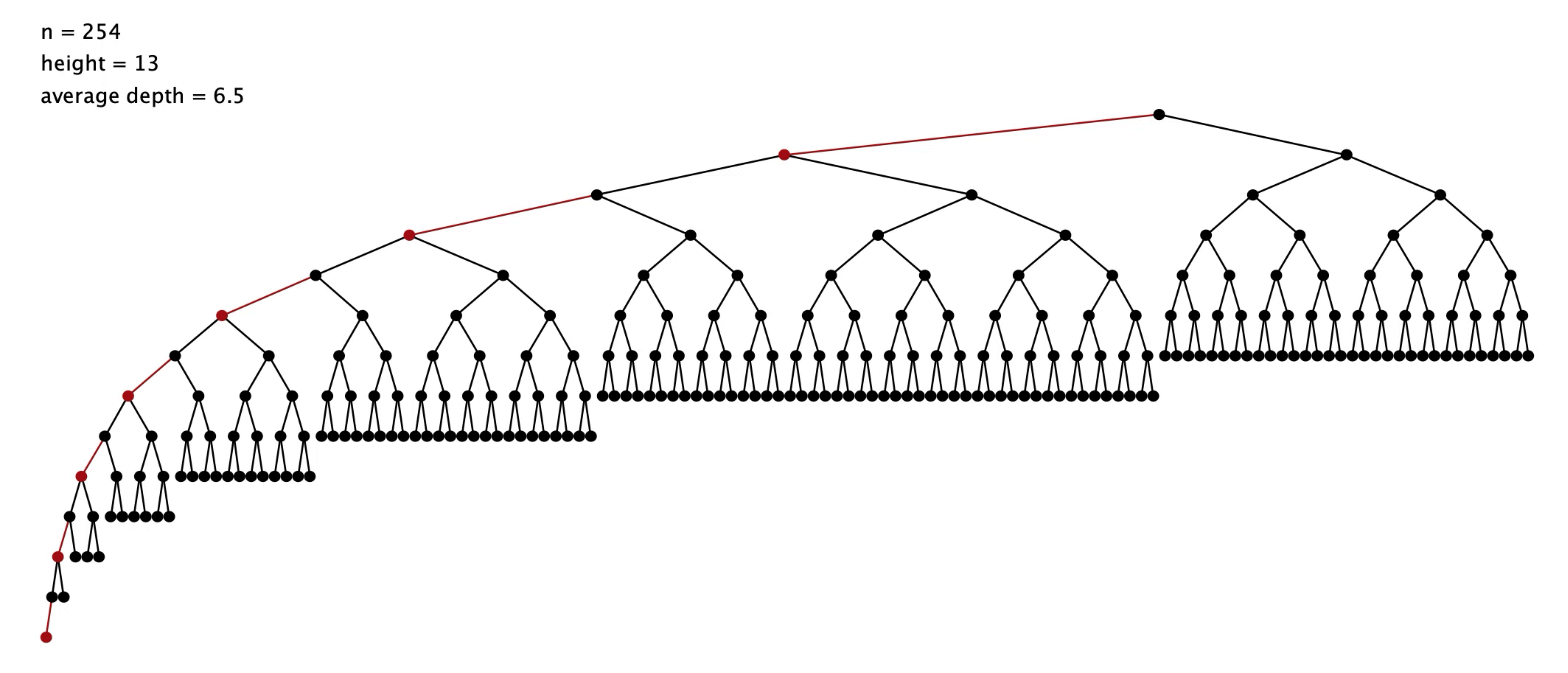
255 insertions in random order

Insertion into a LLRB tree: visualization



255 insertions in ascending order

Insertion into a LLRB tree: visualization



254 insertions in descending order

ST implementations: summary

implementation	worst case			ordered ops?	key interface	emoji
	search	insert	delete			
sequential search (unordered list)	n	n	n		`equals()`	☹️
binary search (sorted array)	$\log n$	n	n	✔	`compareTo()`	😕
BST	n	n	n	✔	`compareTo()`	😕
2-3 trees	$\log n$	$\log n$	$\log n$	✔	`compareTo()`	😎
red-black BSTs	$\log n$	$\log n$	$\log n$	✔	`compareTo()`	😍

hidden constant c is small
($\leq 2 \log_2 n$ *compares*)

https://algs4.cs.princeton.edu

3.3 Balanced Search Trees

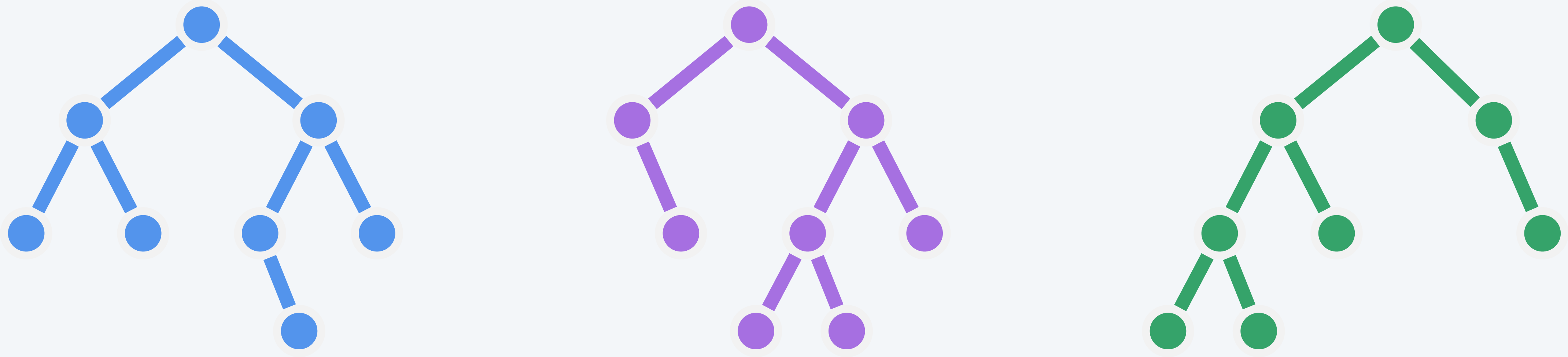
- 2–3 search trees
- red–black BSTs (representation)
- red–black BSTs (operations)
- *context*

Balanced search trees in the wild

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

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

Other balanced BSTs. AVL trees, splay trees, randomized BSTs, rank-balanced BSTs, ….



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

Industry story 1: red–black BSTs

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

Database implementation.

- Red–black BST.
- Exceeding height limit of 80 triggered error-recovery process.

should support up to 2^{40} keys

Database crashed.

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


[redacted]

> " *If implemented properly, the height of a red–black BST with n keys is at most* $2 \log_2 n$*.* " — expert witness

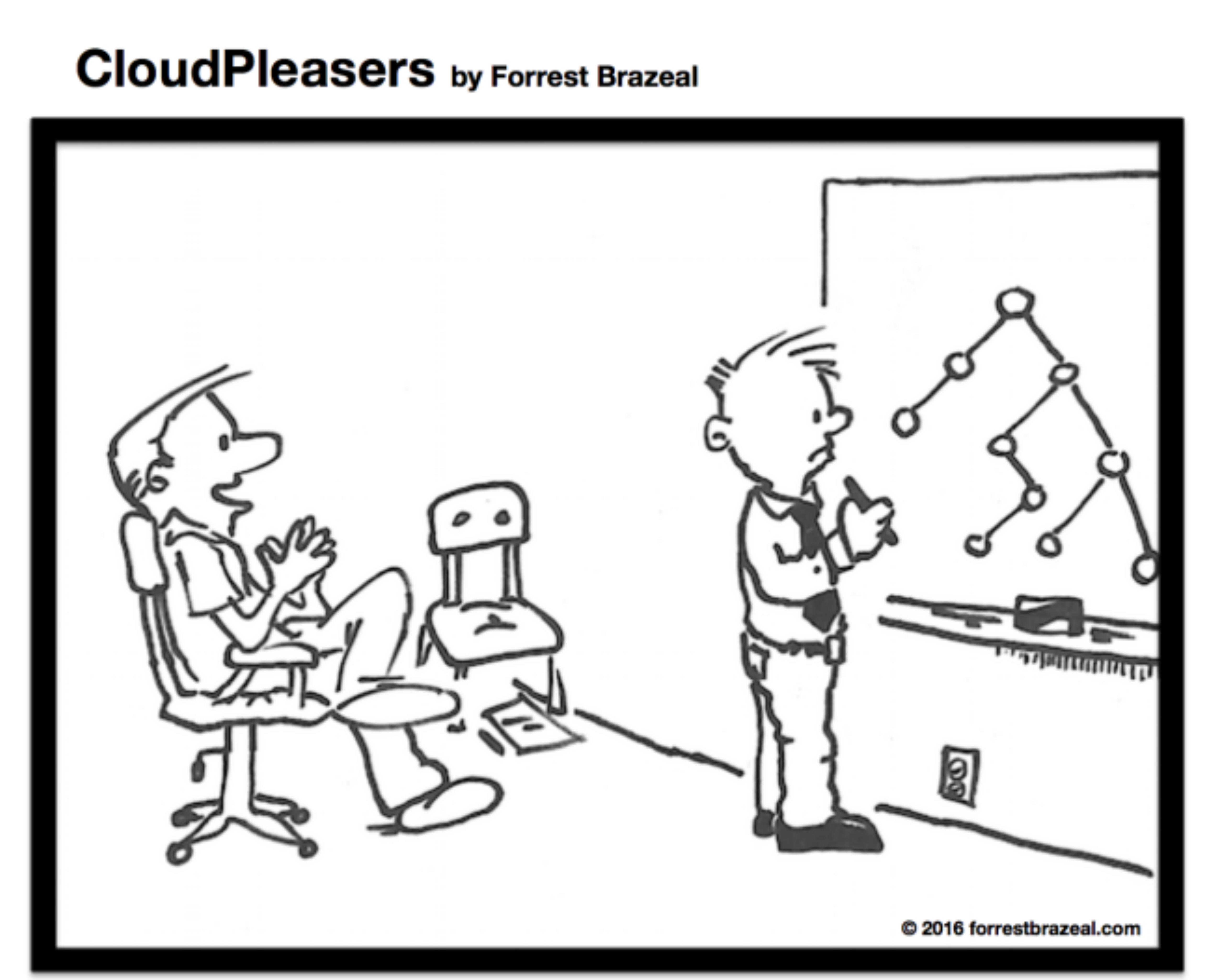
Industry story 2: red-black BSTs

https://twitter.com/cyberomin/status/835888786462625792

Credits

media	source	license
Technological Wizard	Adobe Stock	Education License
Red–Black Tree Song	Sean Sandys	by author
Red–Black Tree Song Video	U. Washington CSE Band	
Gavel	Adobe Stock	Education License
Redacted Document	Wikimedia	public domain
Celestine Omin	Twitter	
Balancing a BST Interview	Forrest Brazeal	

A final thought



"We want our interviewees to solve real-world problems. So while you balance this binary search tree, I'll be changing the requirements, imposing arbitrary deadlines and auditing you for regulatory compliance."