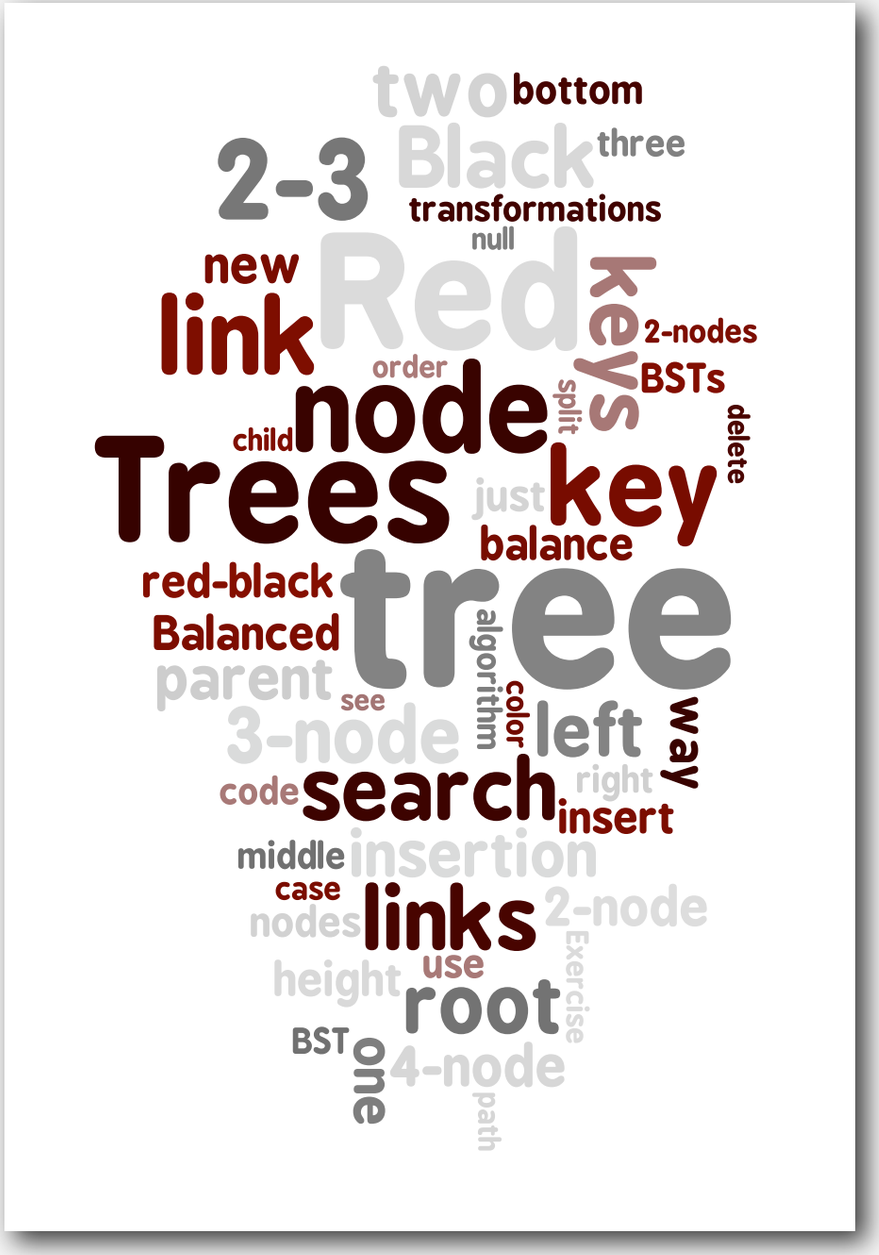
3.3 Balanced Search Trees



- 2-3 search trees
- red-black BST

Symbol table review

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
goal	log N	log N	log N	log N	log N	log N	yes	`compareTo()`

Challenge. Guarantee performance.

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

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

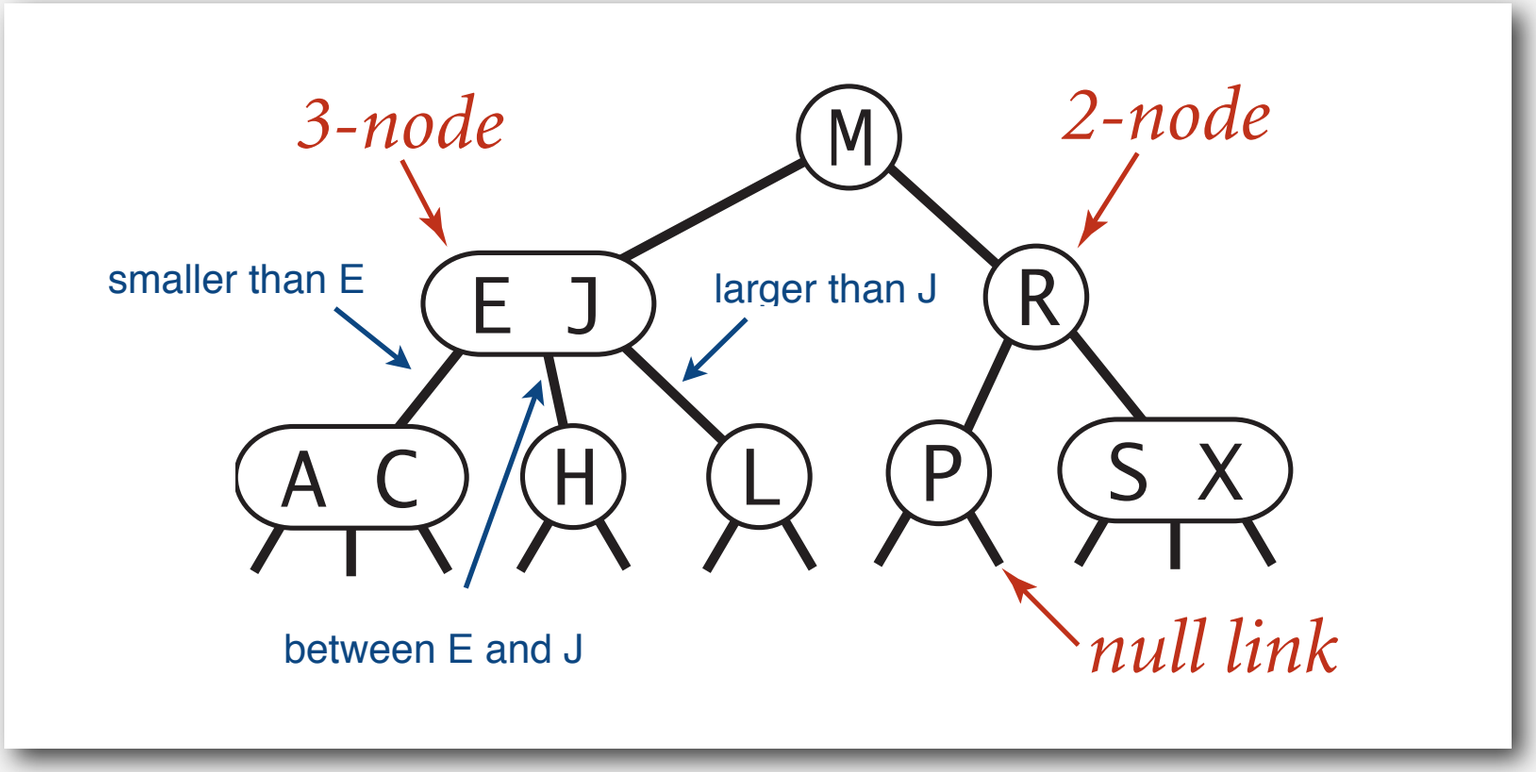
2-3 tree

Allow 1 or 2 keys per node.

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

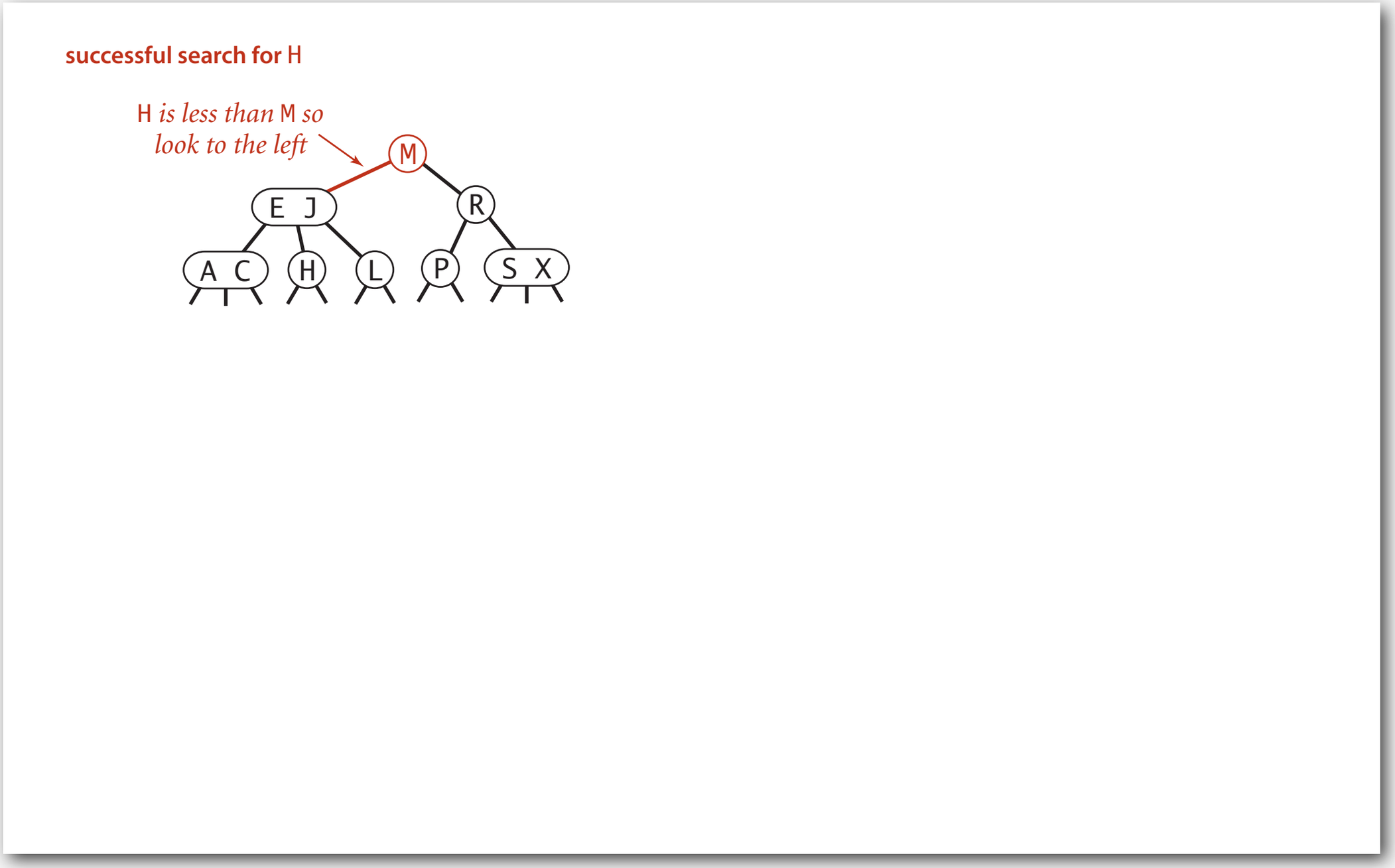
Symmetric order. Inorder traversal yields keys in ascending order.

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



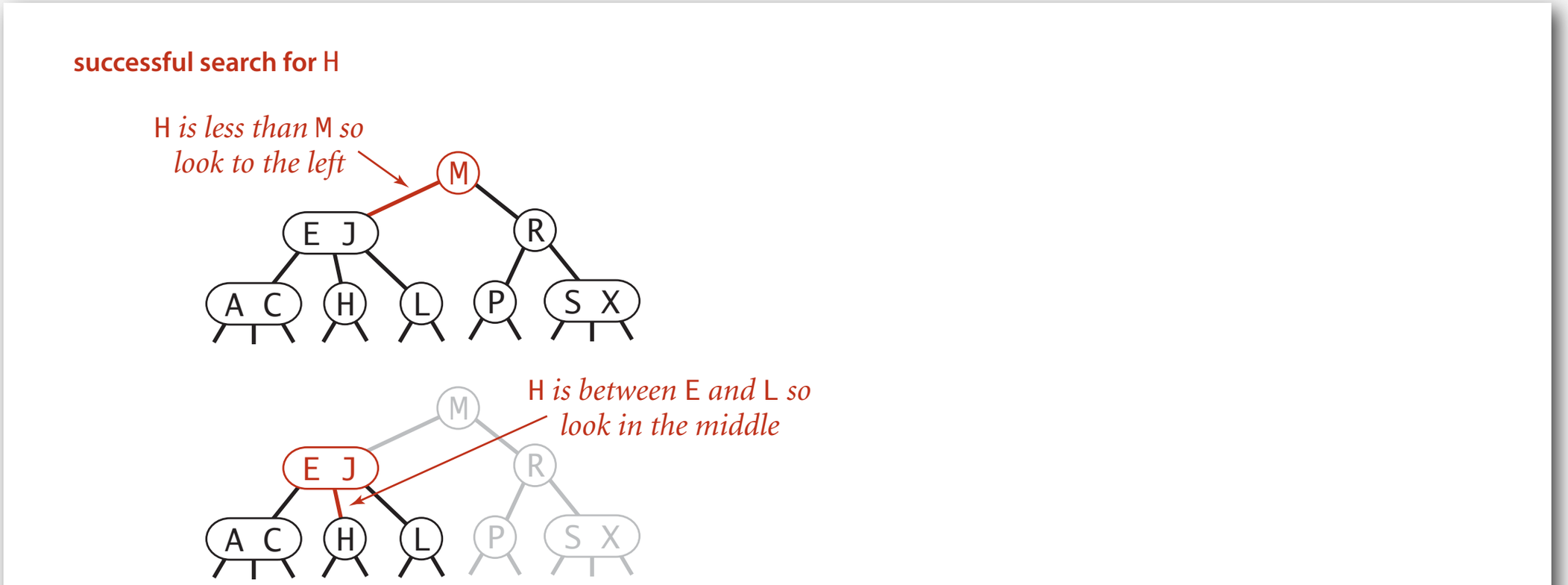
Search in a 2-3 tree

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



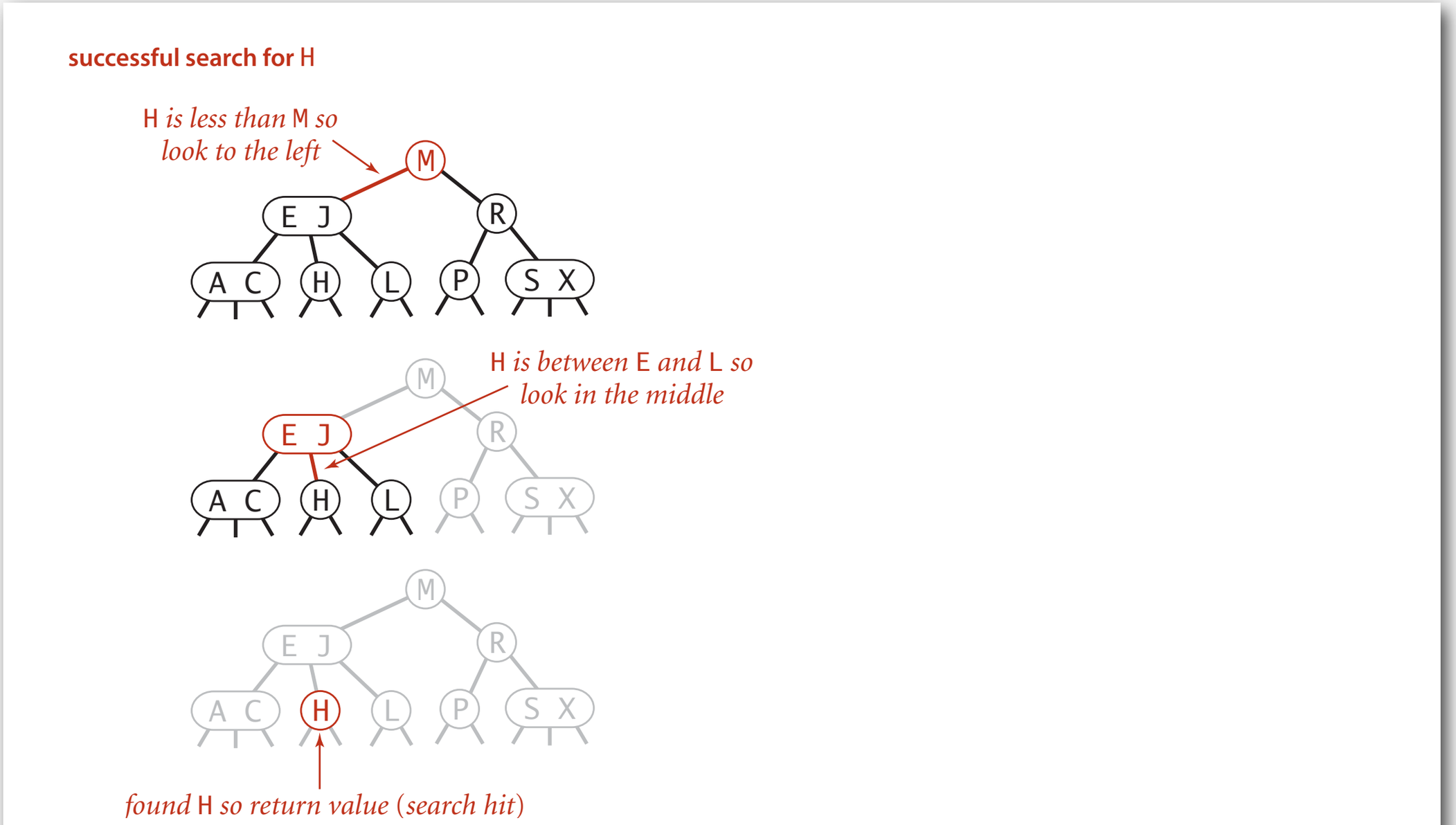
Search in a 2-3 tree

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



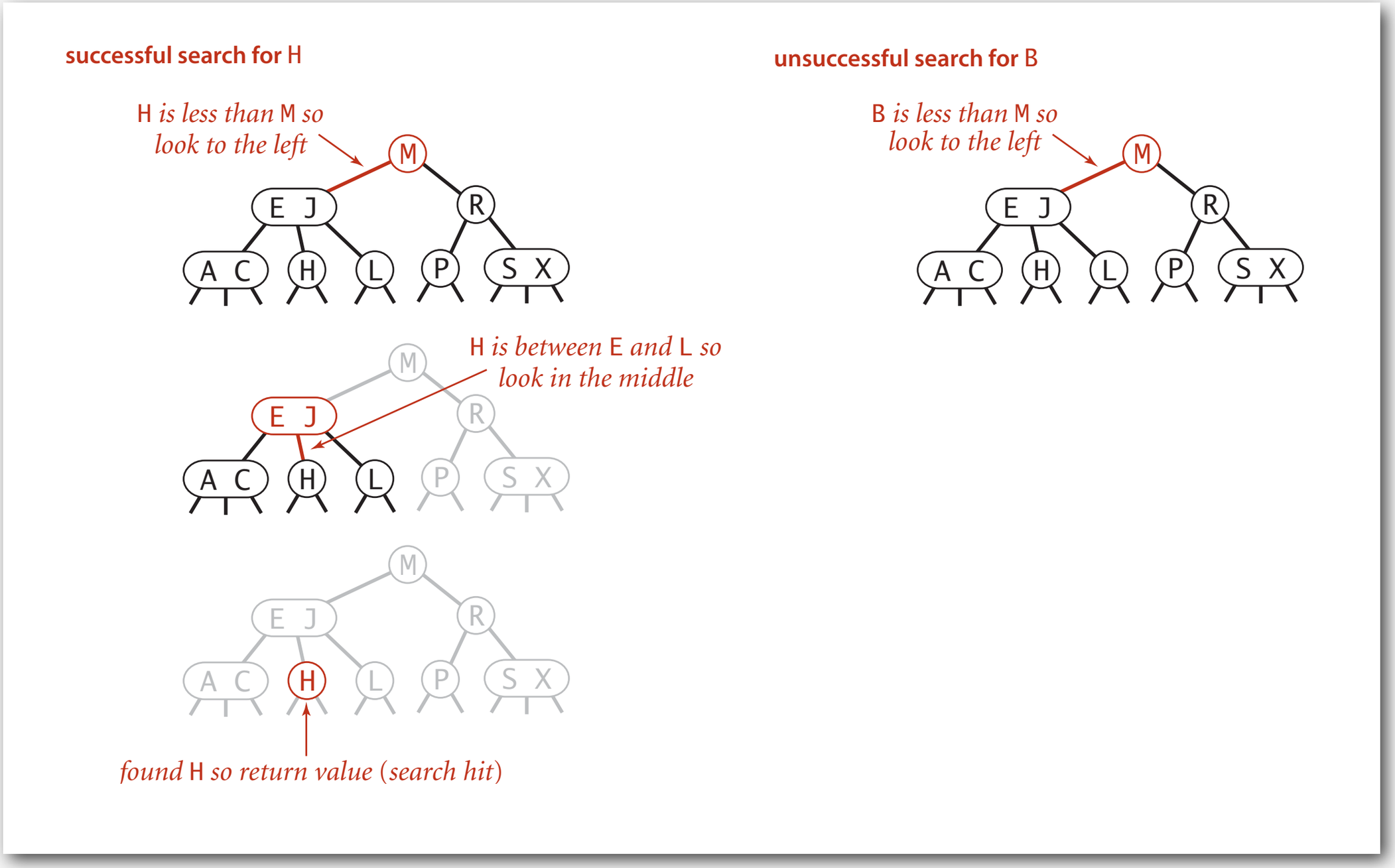
Search in a 2-3 tree

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



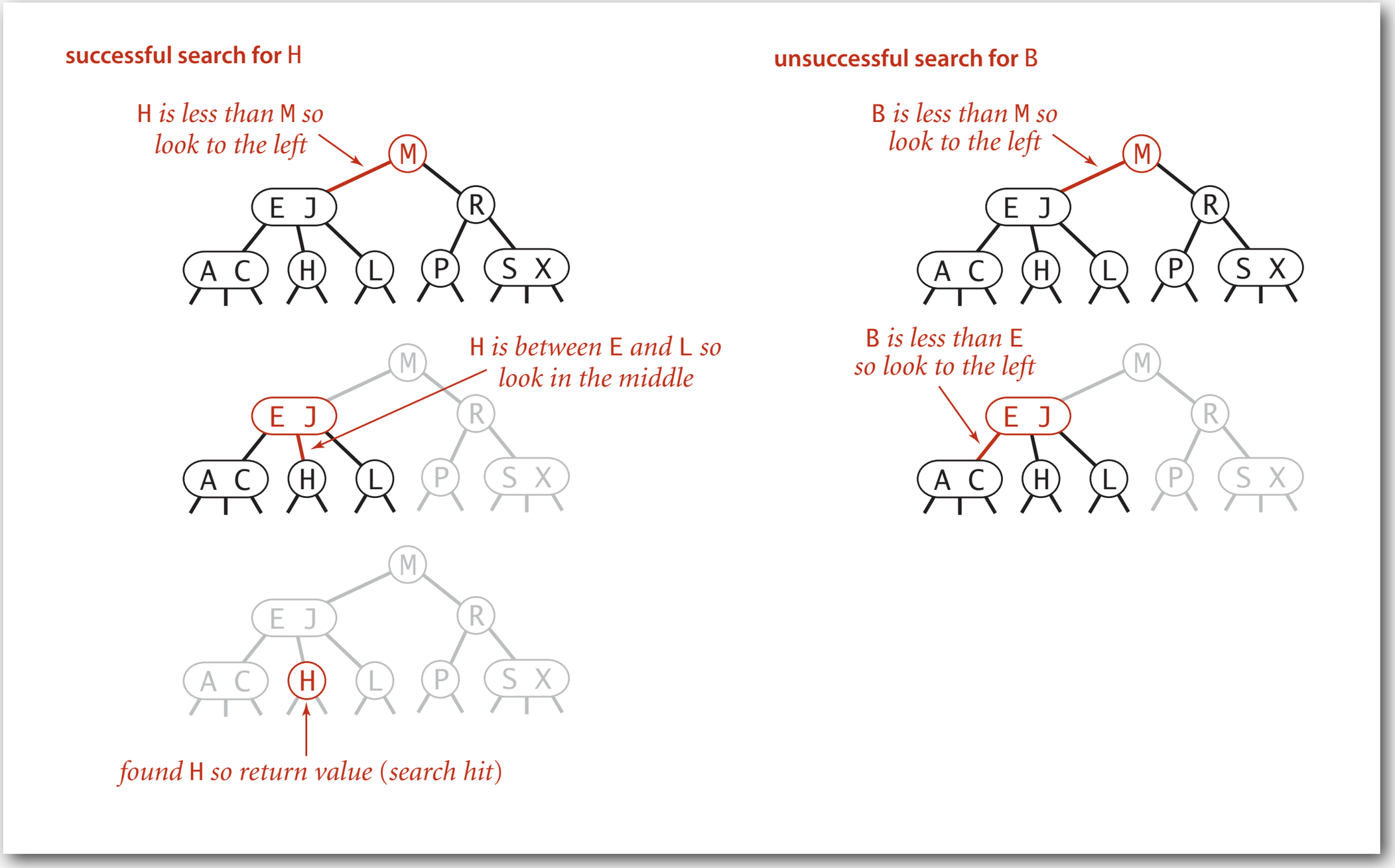
Search in a 2-3 tree

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



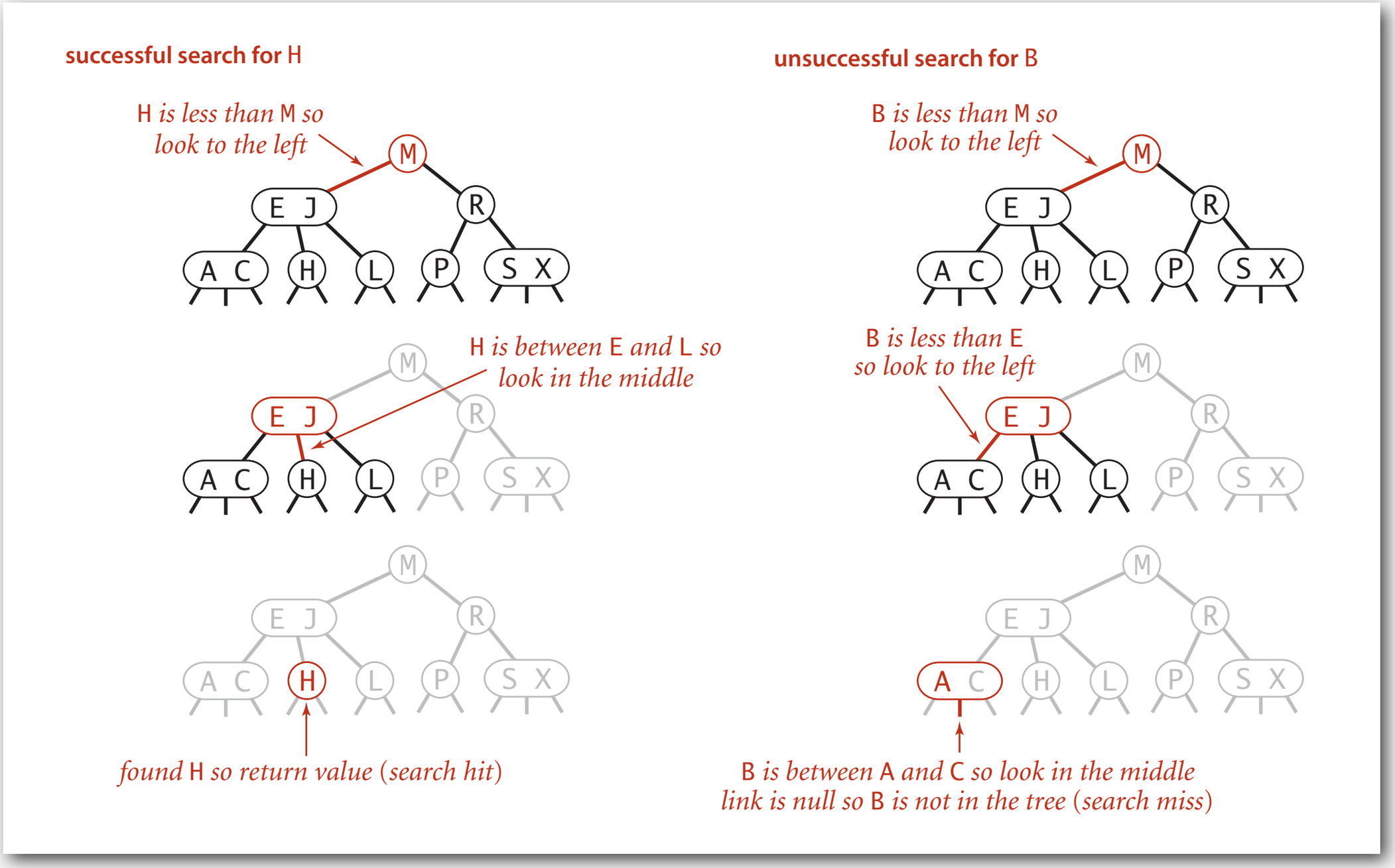
Search in a 2-3 tree

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



Search in a 2-3 tree

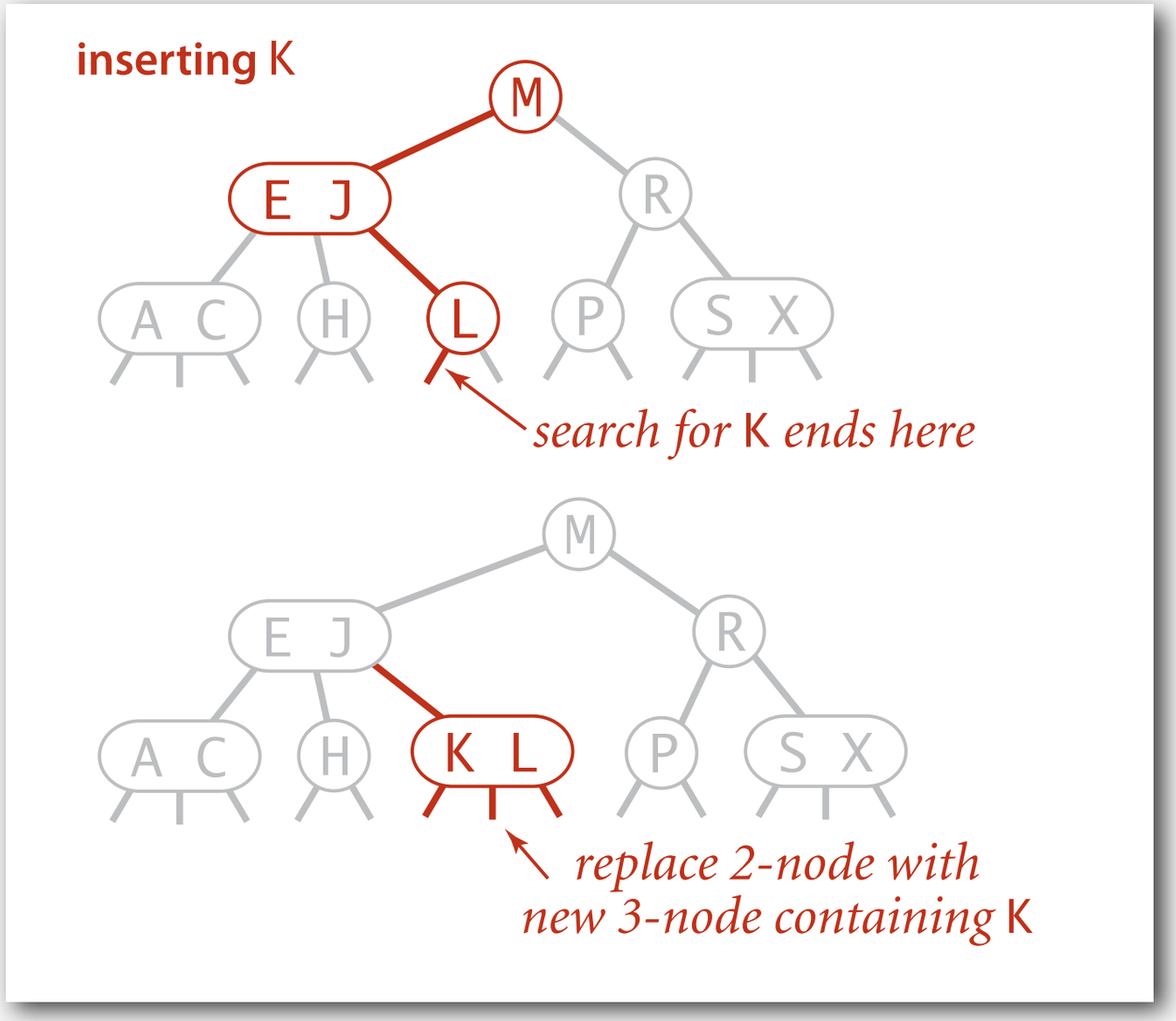
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).



Insertion in a 2-3 tree

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

- Search for key, as usual.
- Replace 2-node with 3-node.

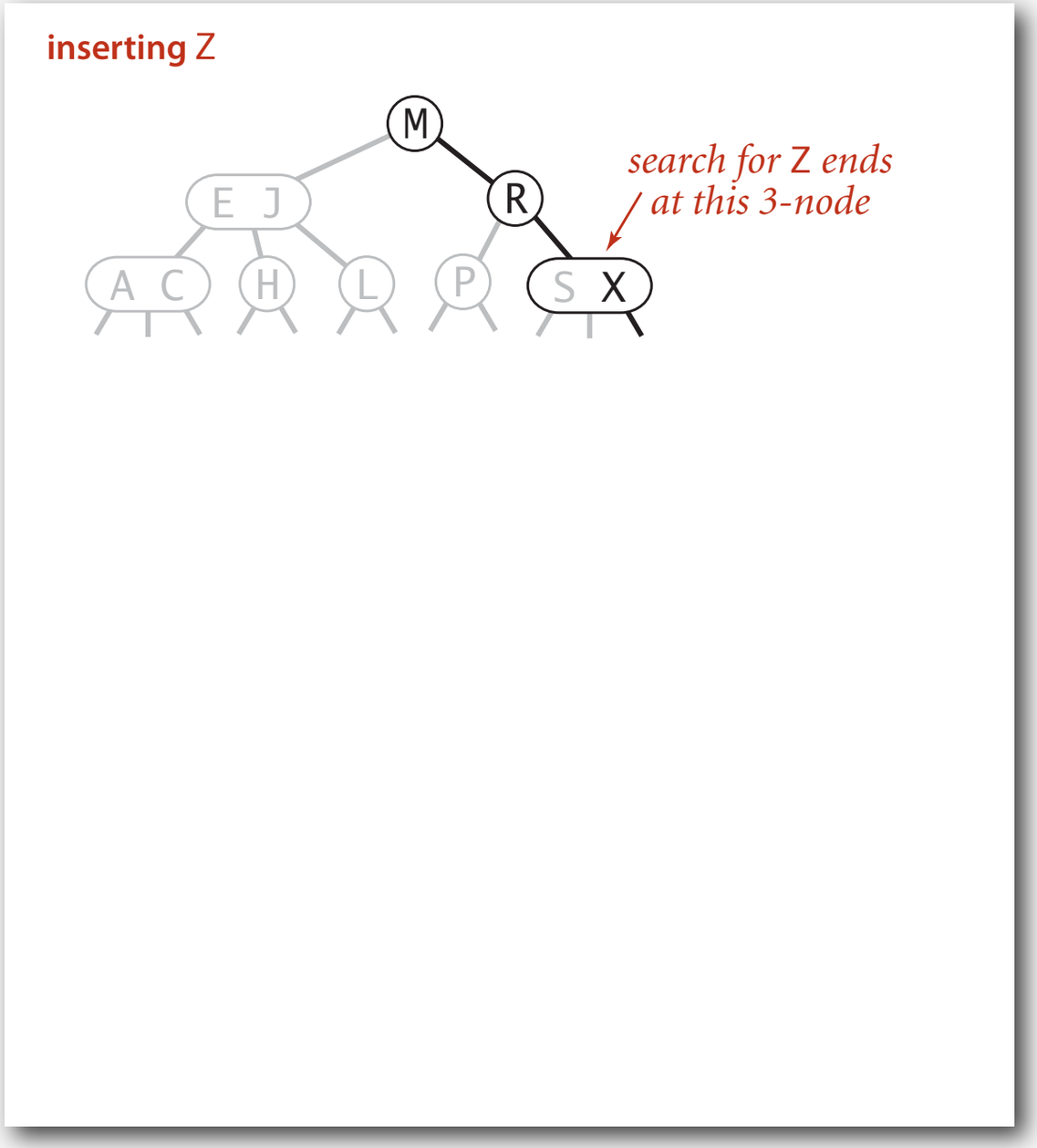


Insertion in a 2-3 tree

Case 2. Insert 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.

why middle key?

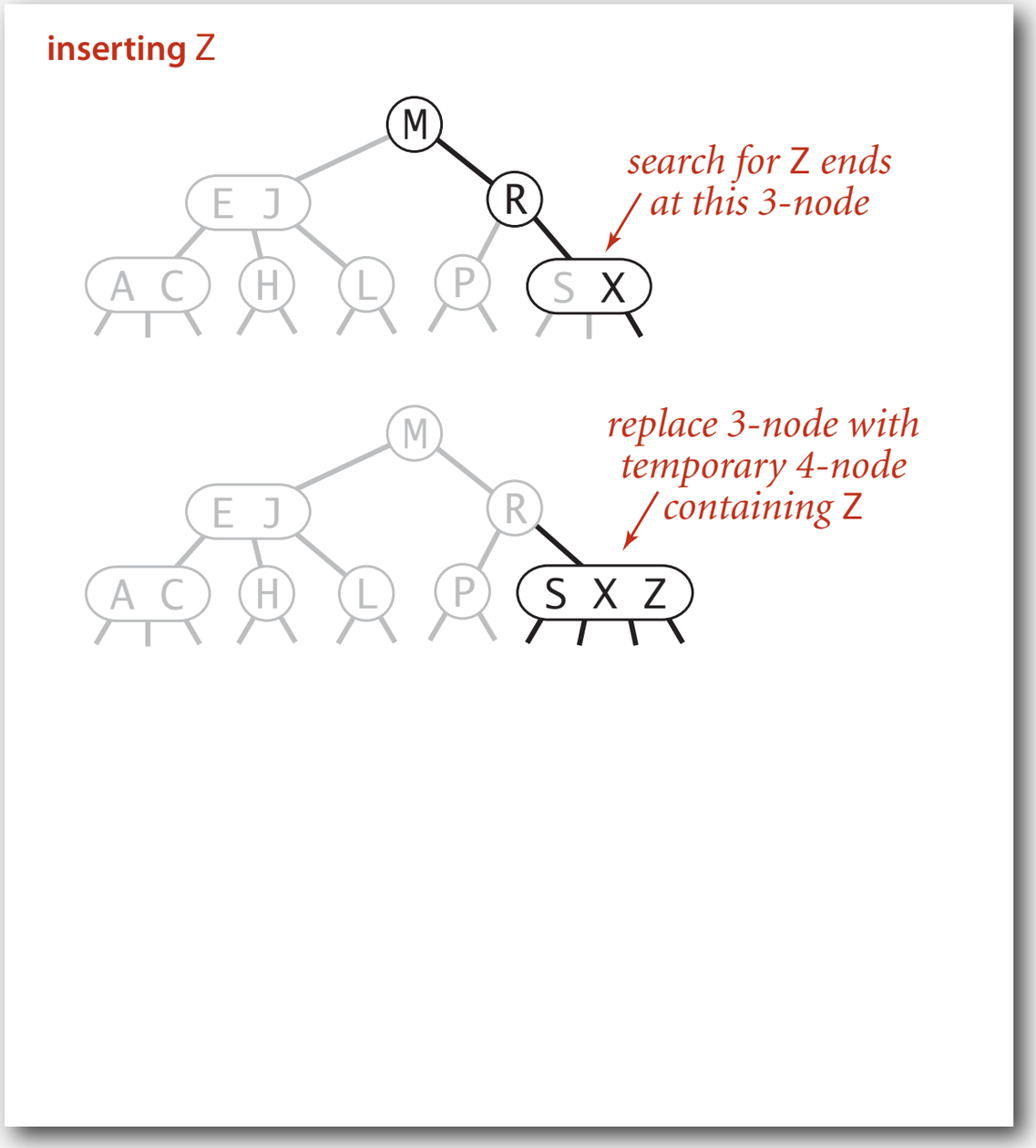


Insertion in a 2-3 tree

Case 2. Insert 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.

why middle key?

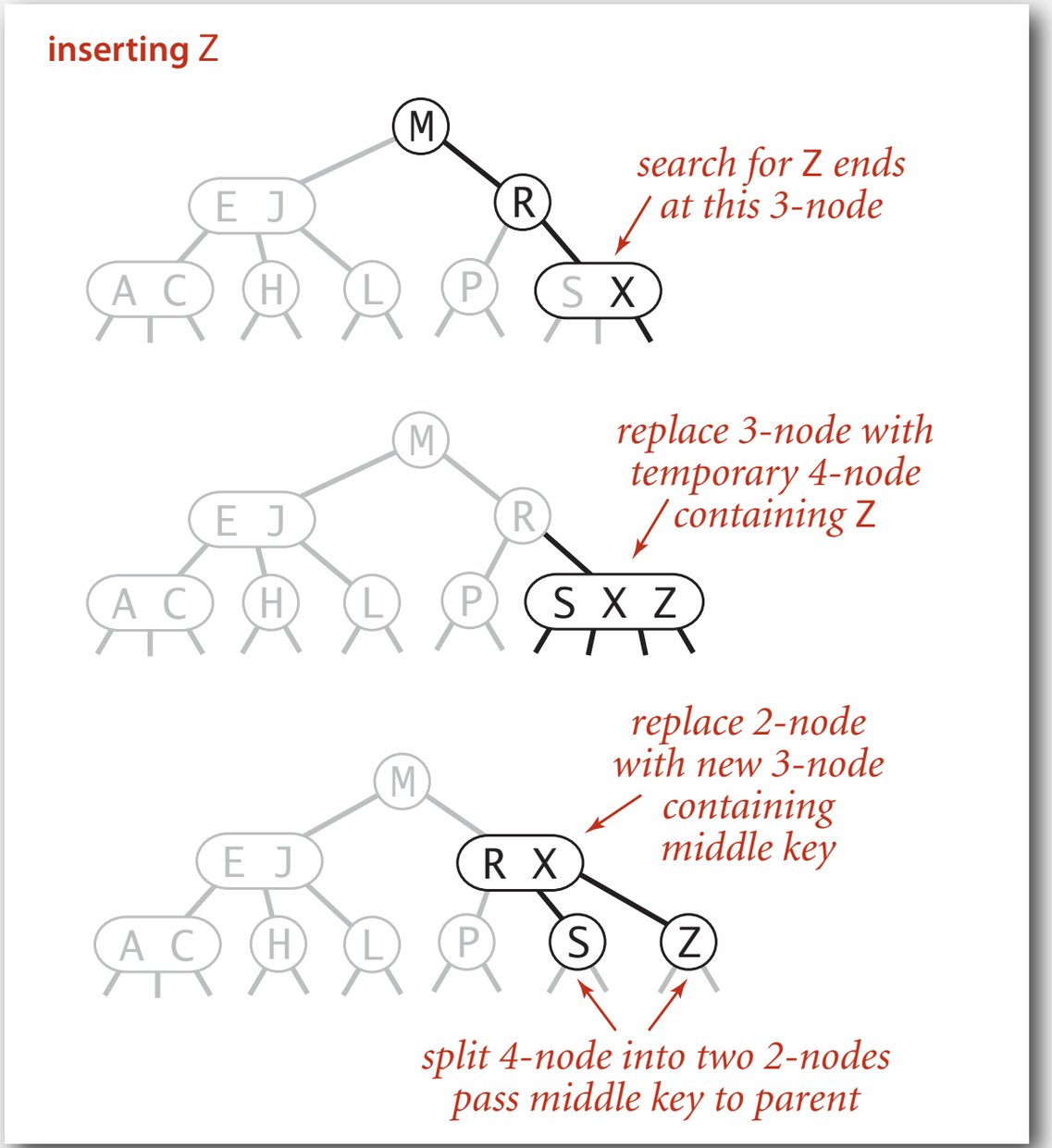


Insertion in a 2-3 tree

Case 2. Insert 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.

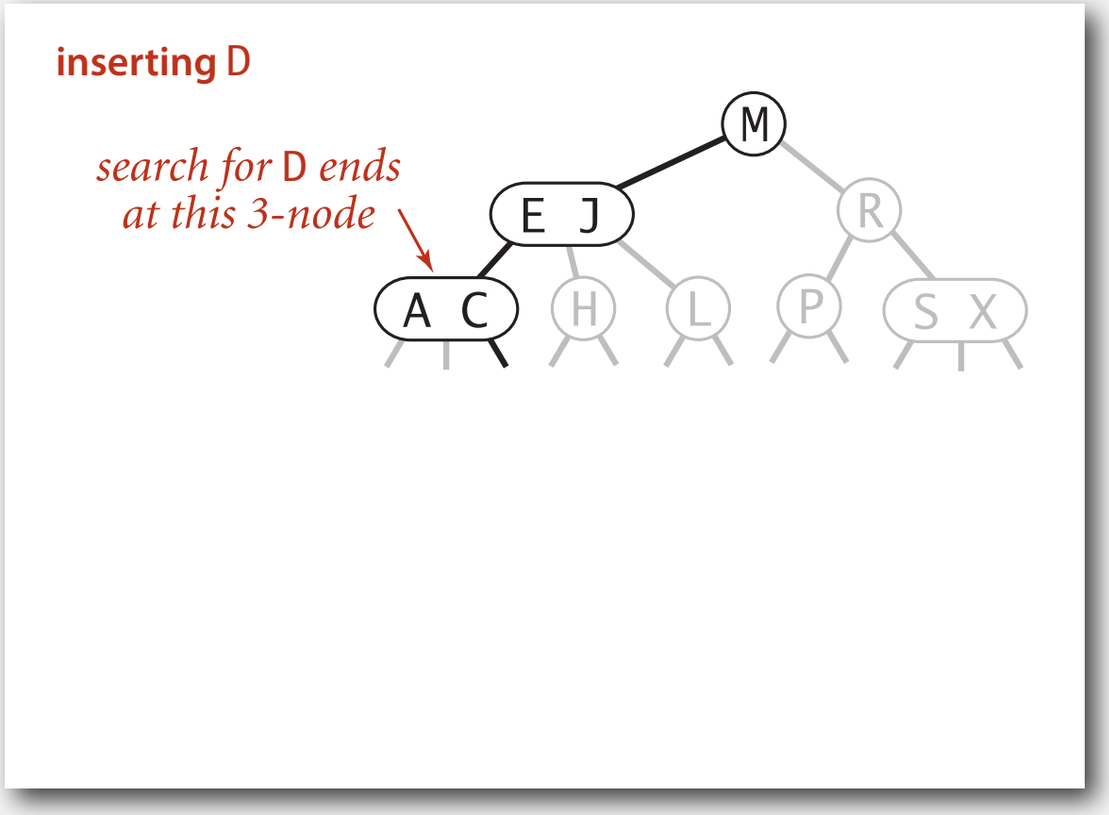
why middle key?



Insertion in a 2-3 tree

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

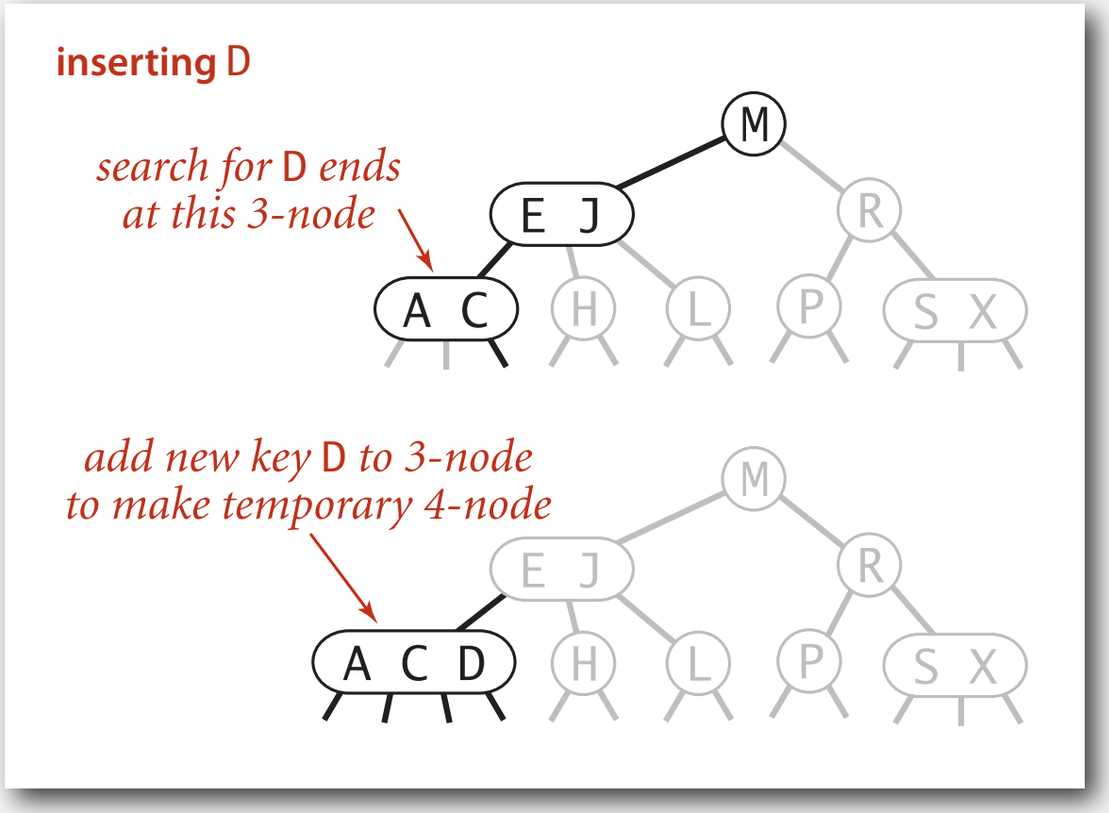
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.



Insertion in a 2-3 tree

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

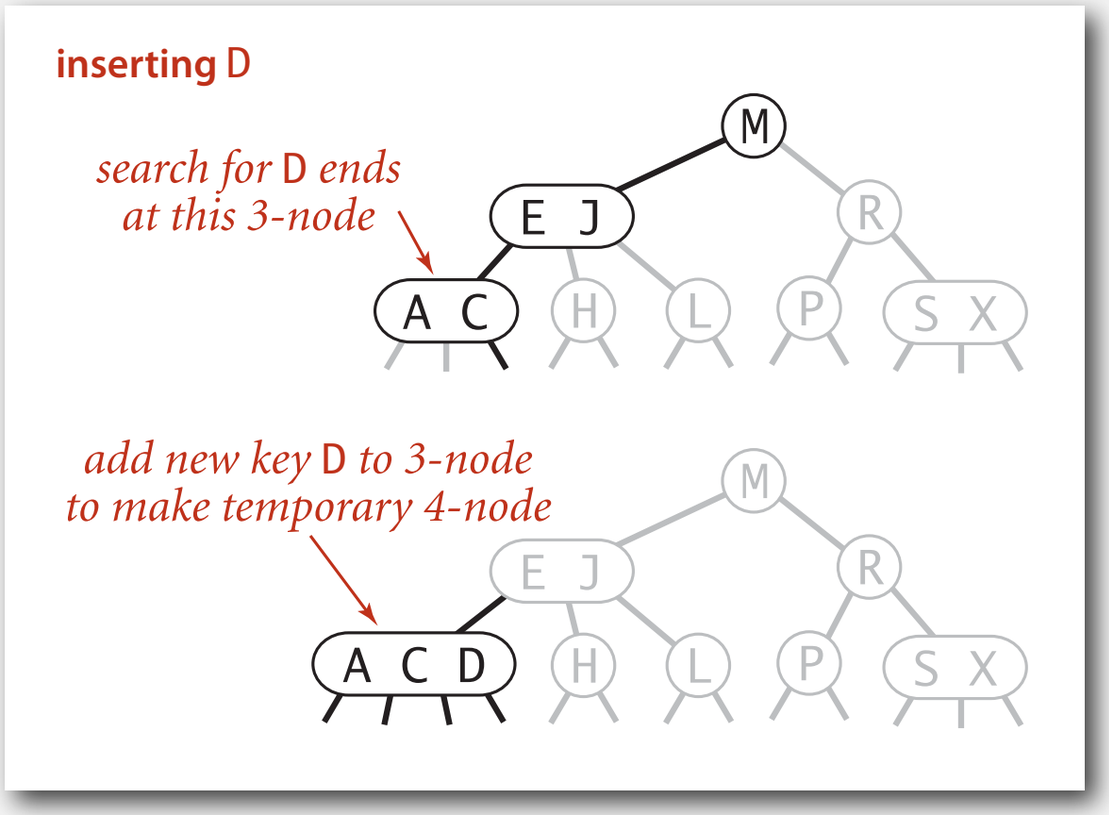
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.

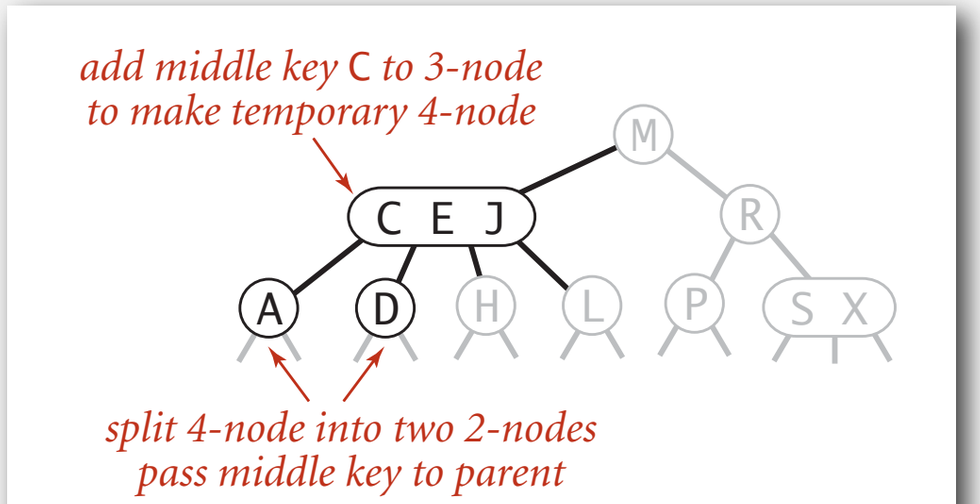


Insertion in a 2-3 tree

Case 2. Insert 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.

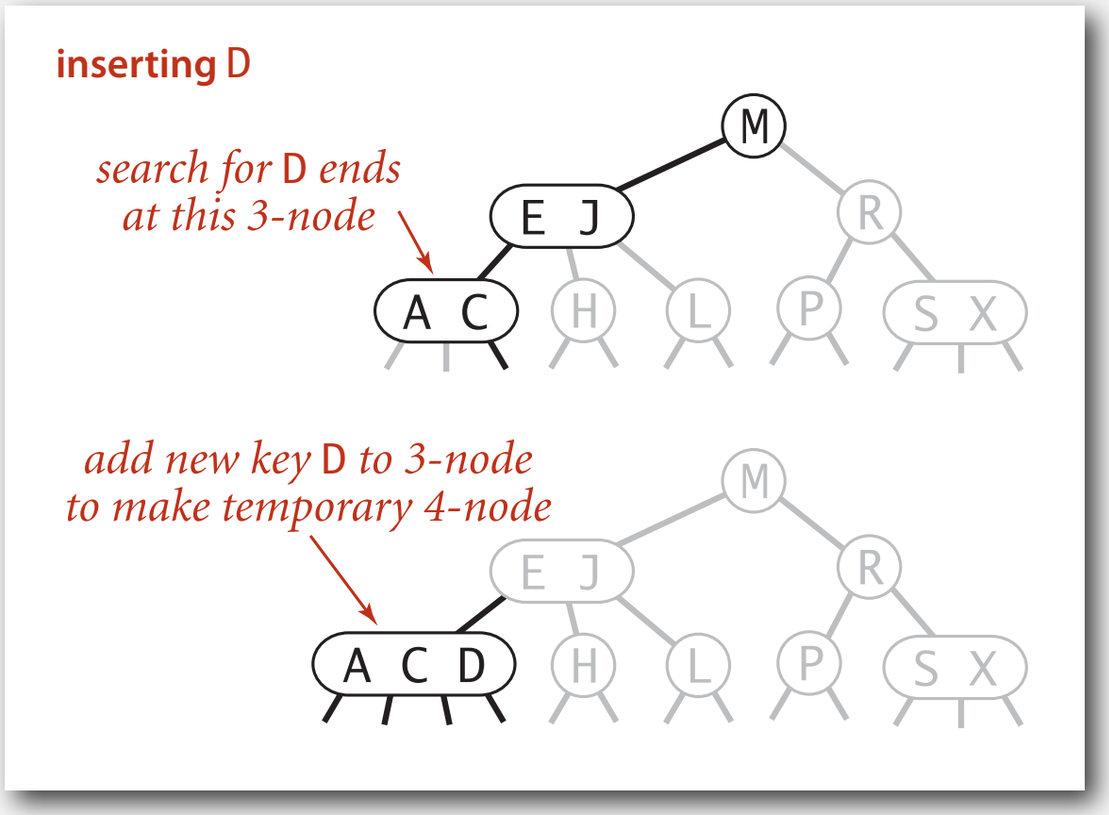


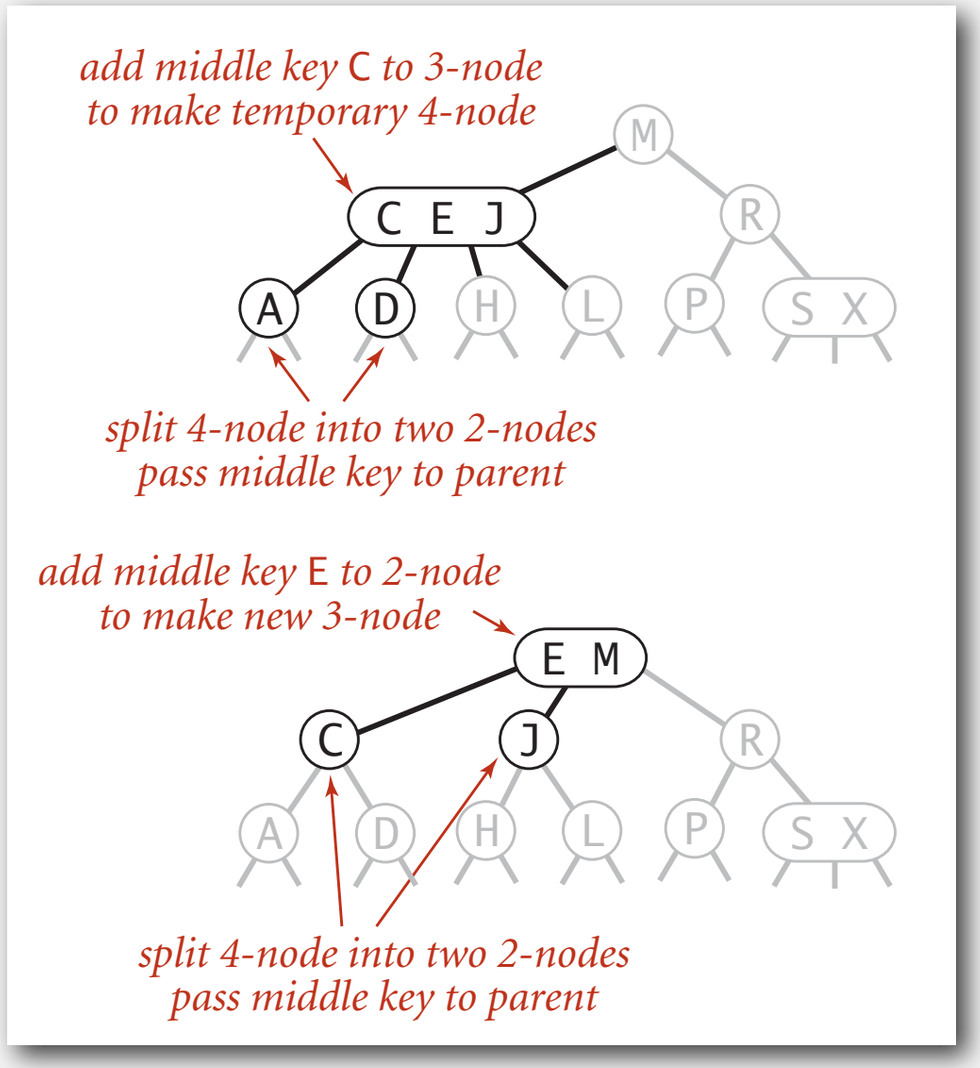


Insertion in a 2-3 tree

Case 2. Insert 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.

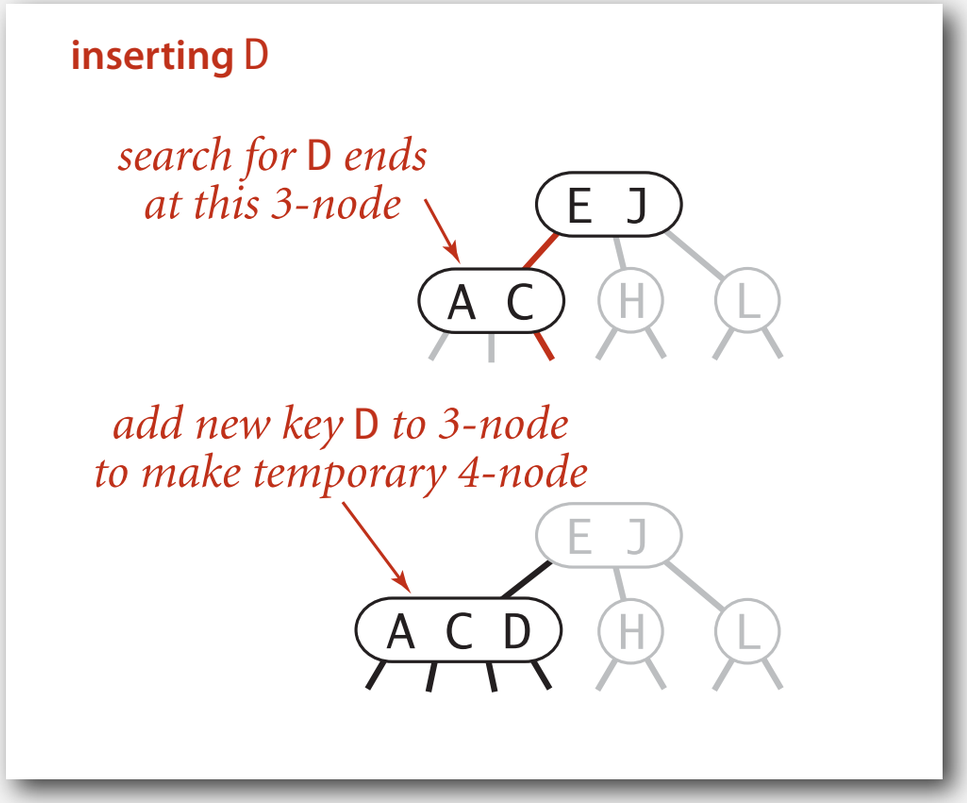


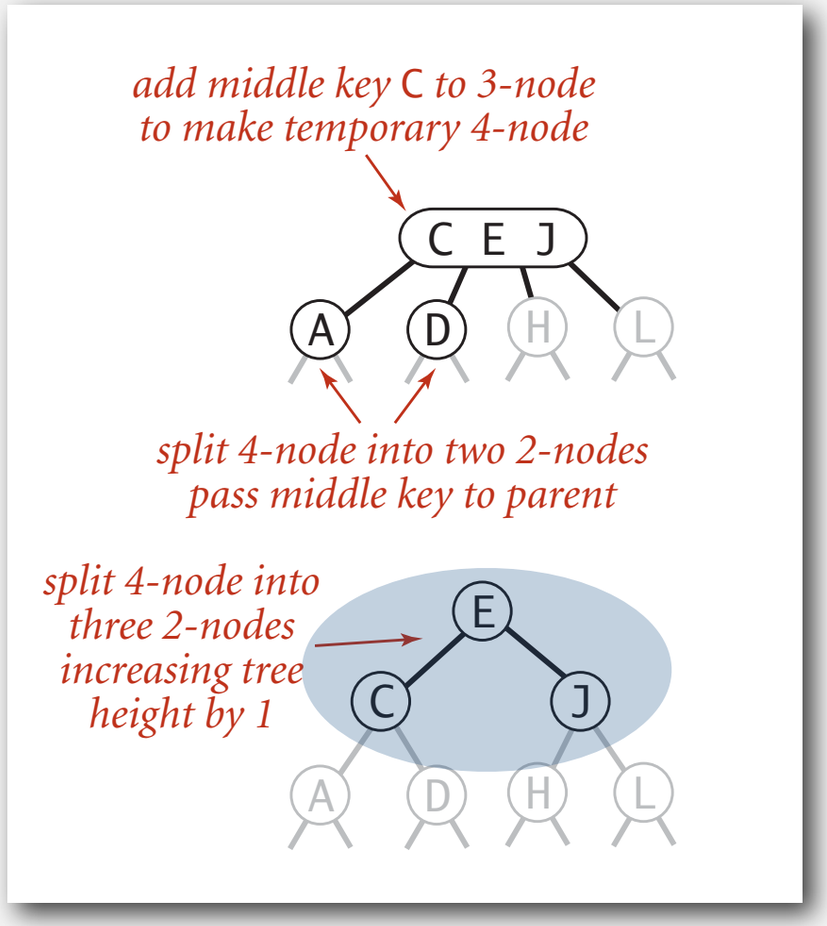


Insertion in a 2-3 tree

Case 2. Insert 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.





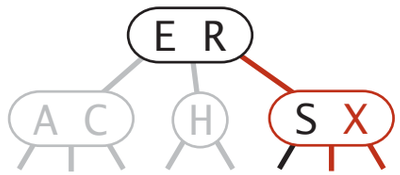
Remark. Splitting the root increases height by 1.

2-3 tree construction trace

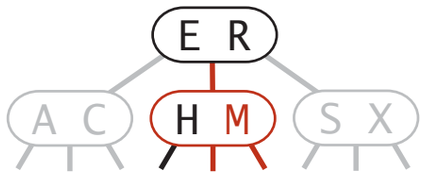
Standard indexing client.

insert S

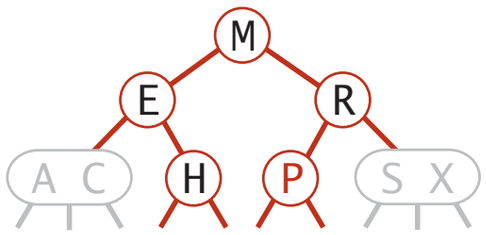
X



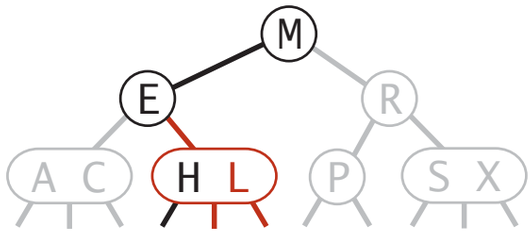
M



P



L



2-3 tree construction trace

Standard indexing client.

2-3 tree construction trace

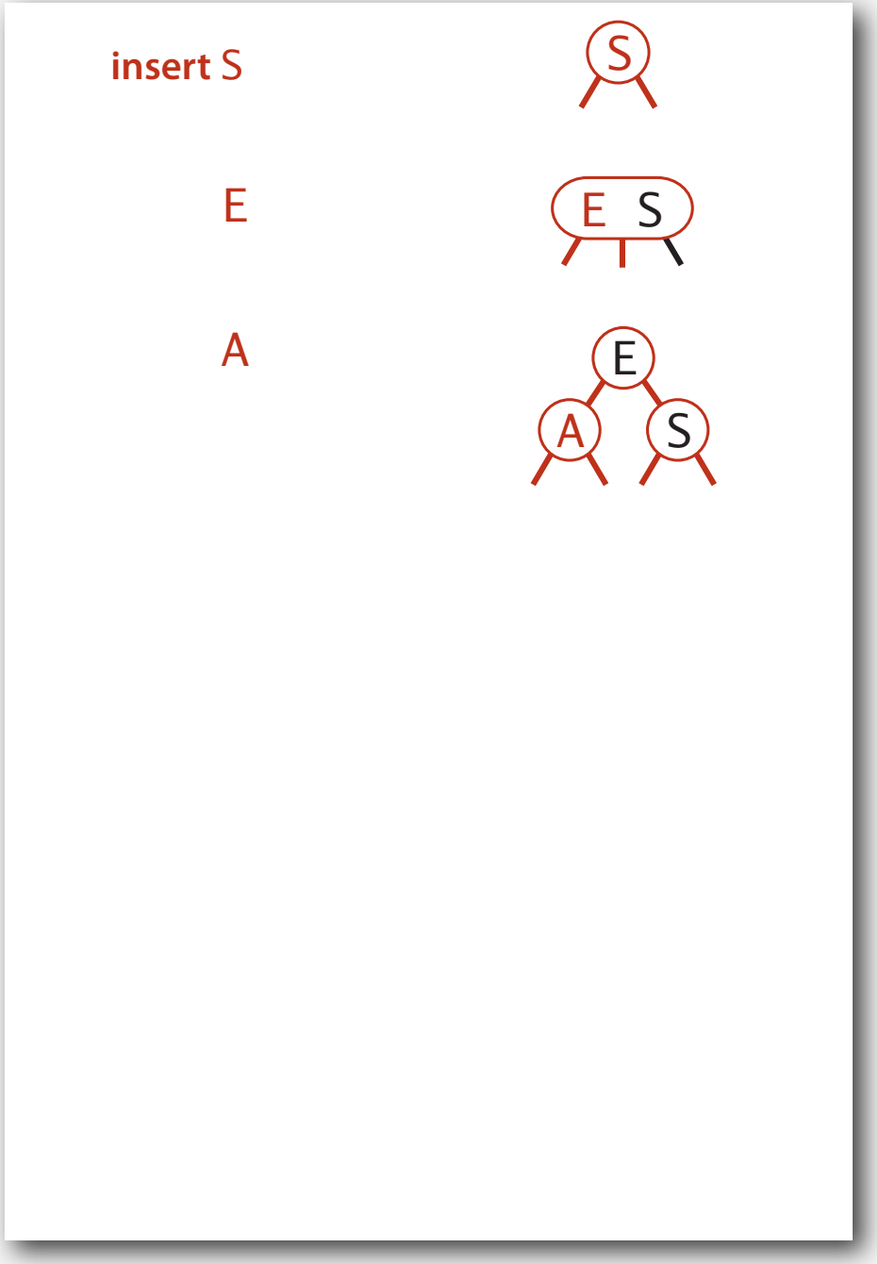
Standard indexing client.

insert S

E

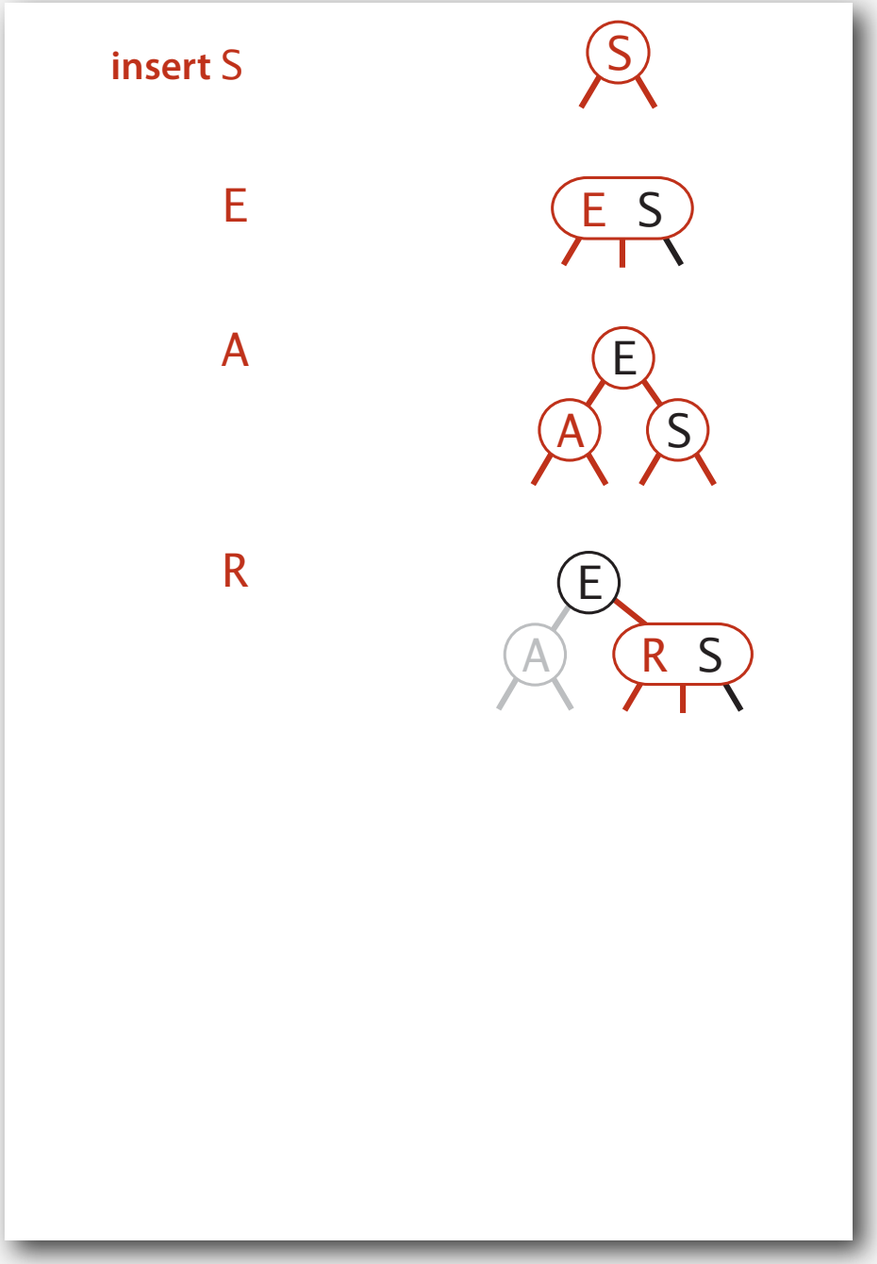
2-3 tree construction trace

Standard indexing client.



2-3 tree construction trace

Standard indexing client.



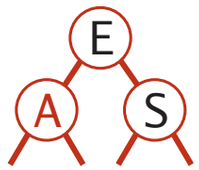
2-3 tree construction trace

Standard indexing client.

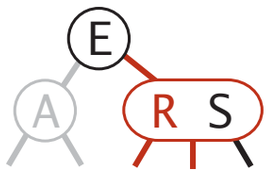
insert S

E

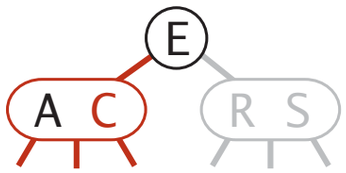
A



R

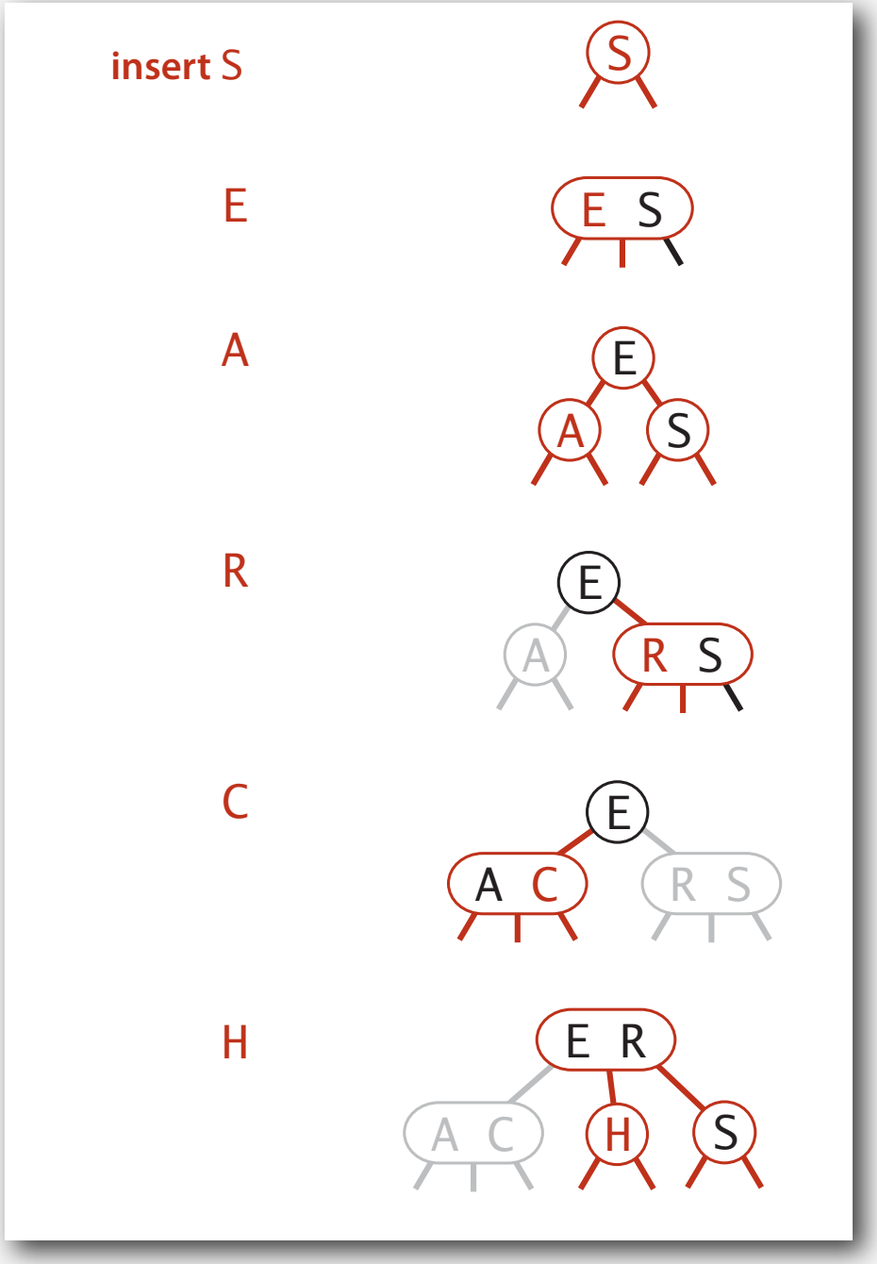


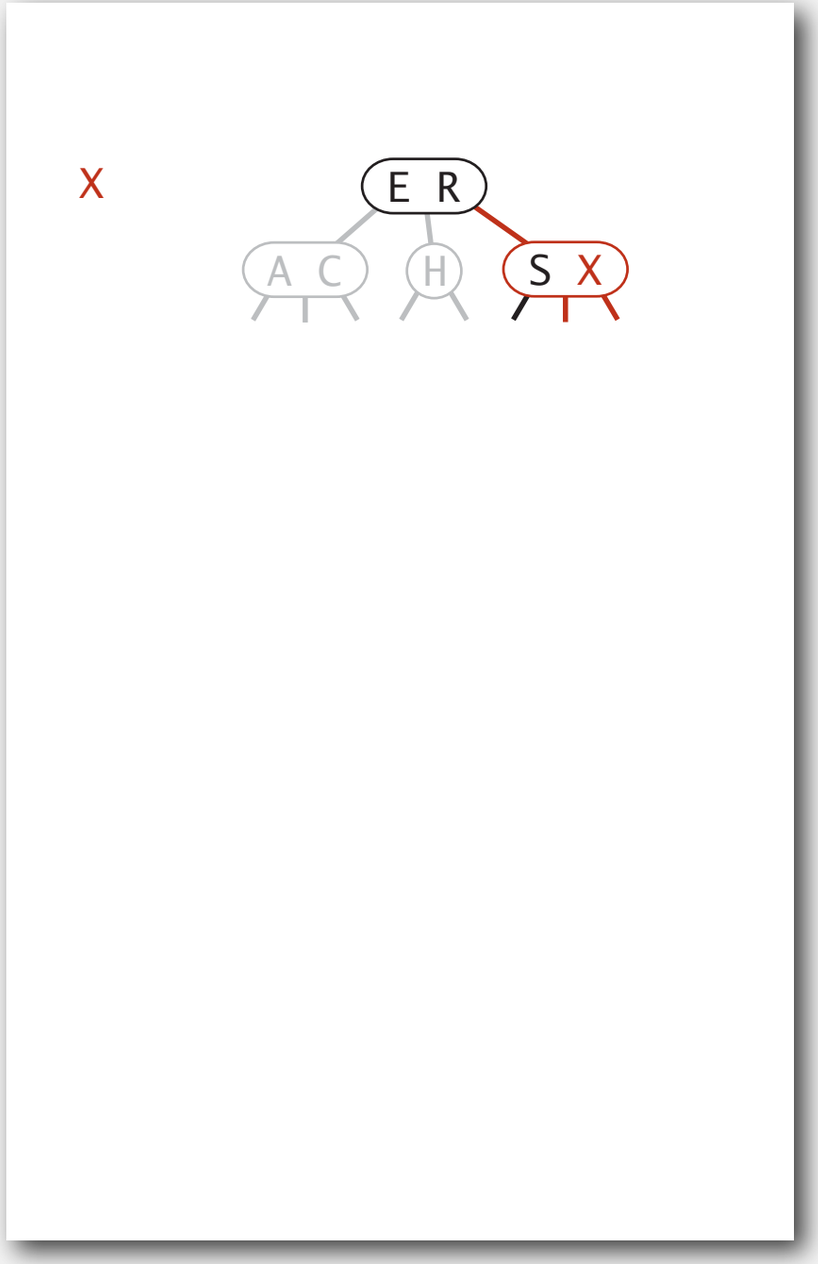
C



2-3 tree construction trace

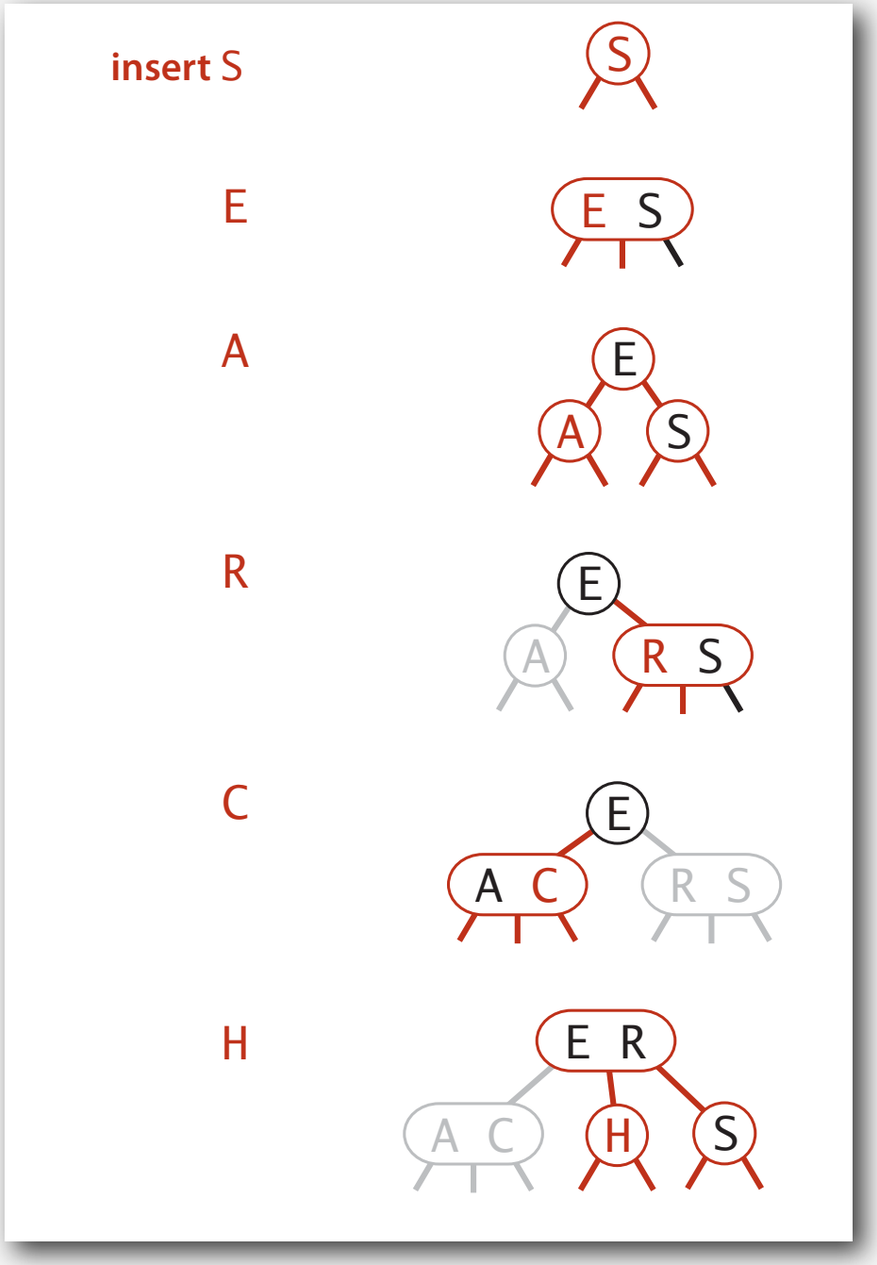
Standard indexing client.

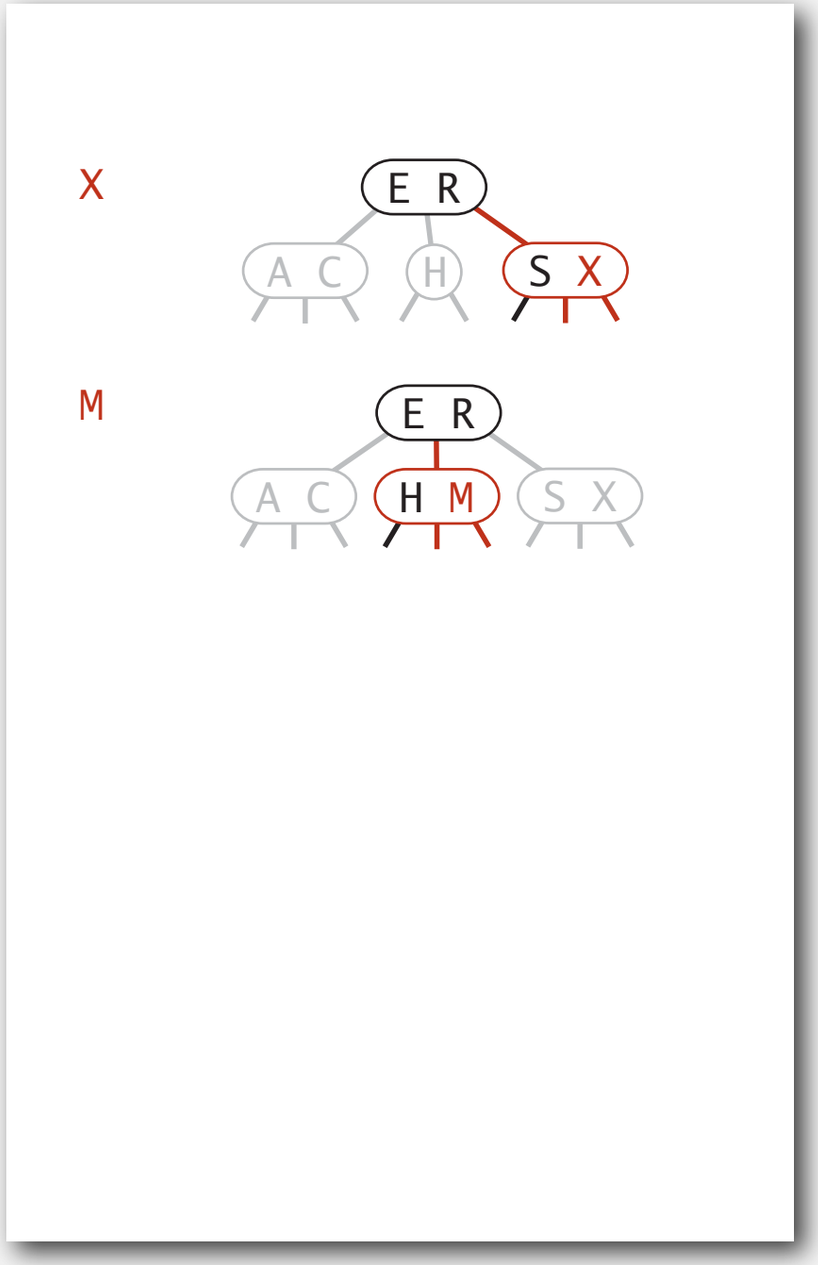




2-3 tree construction trace

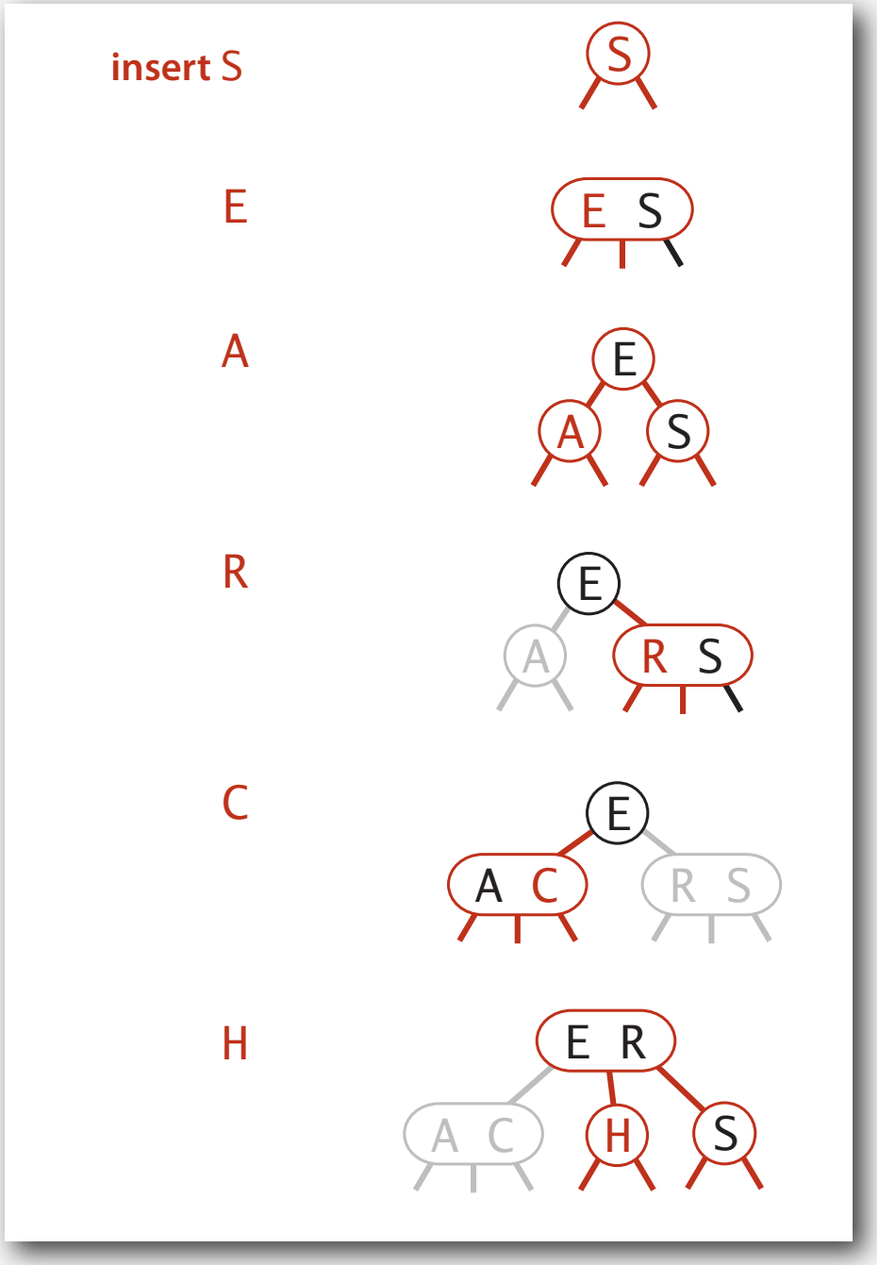
Standard indexing client.

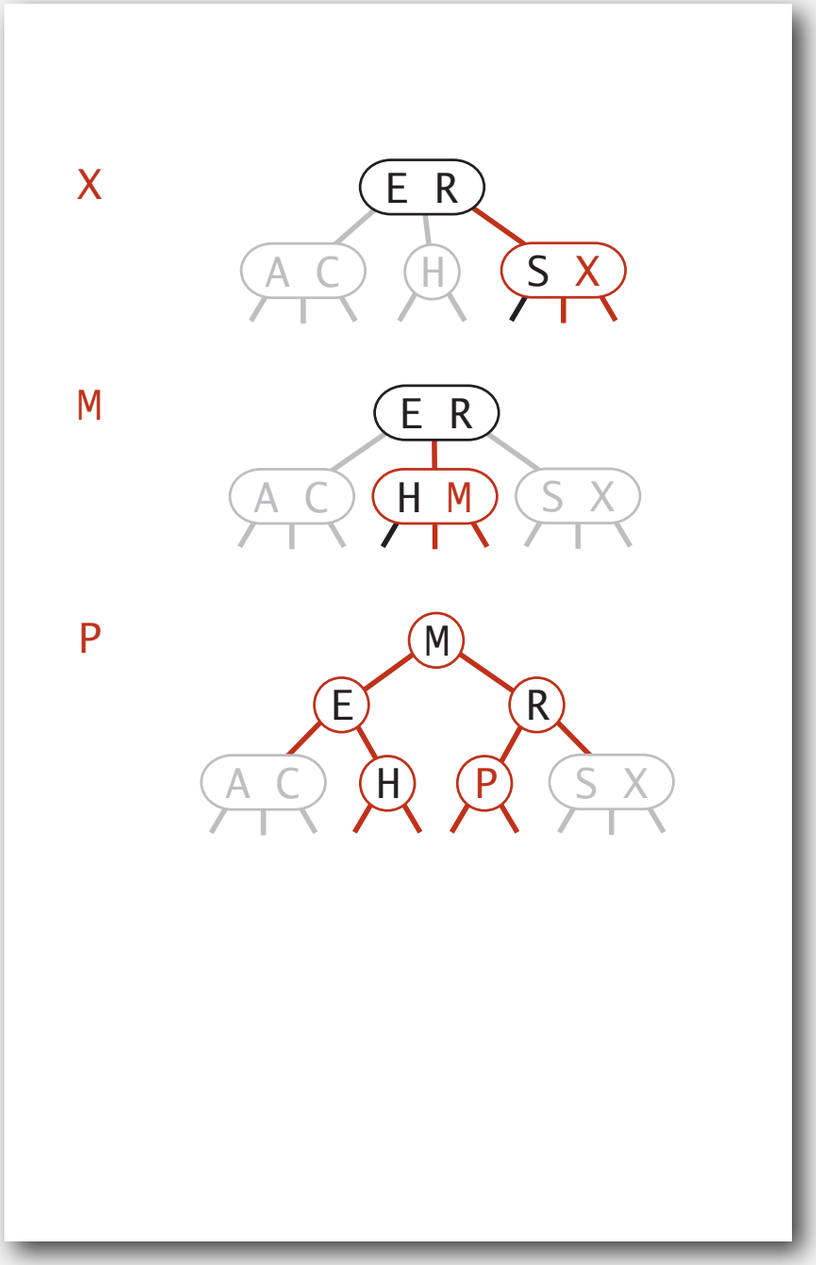




2-3 tree construction trace

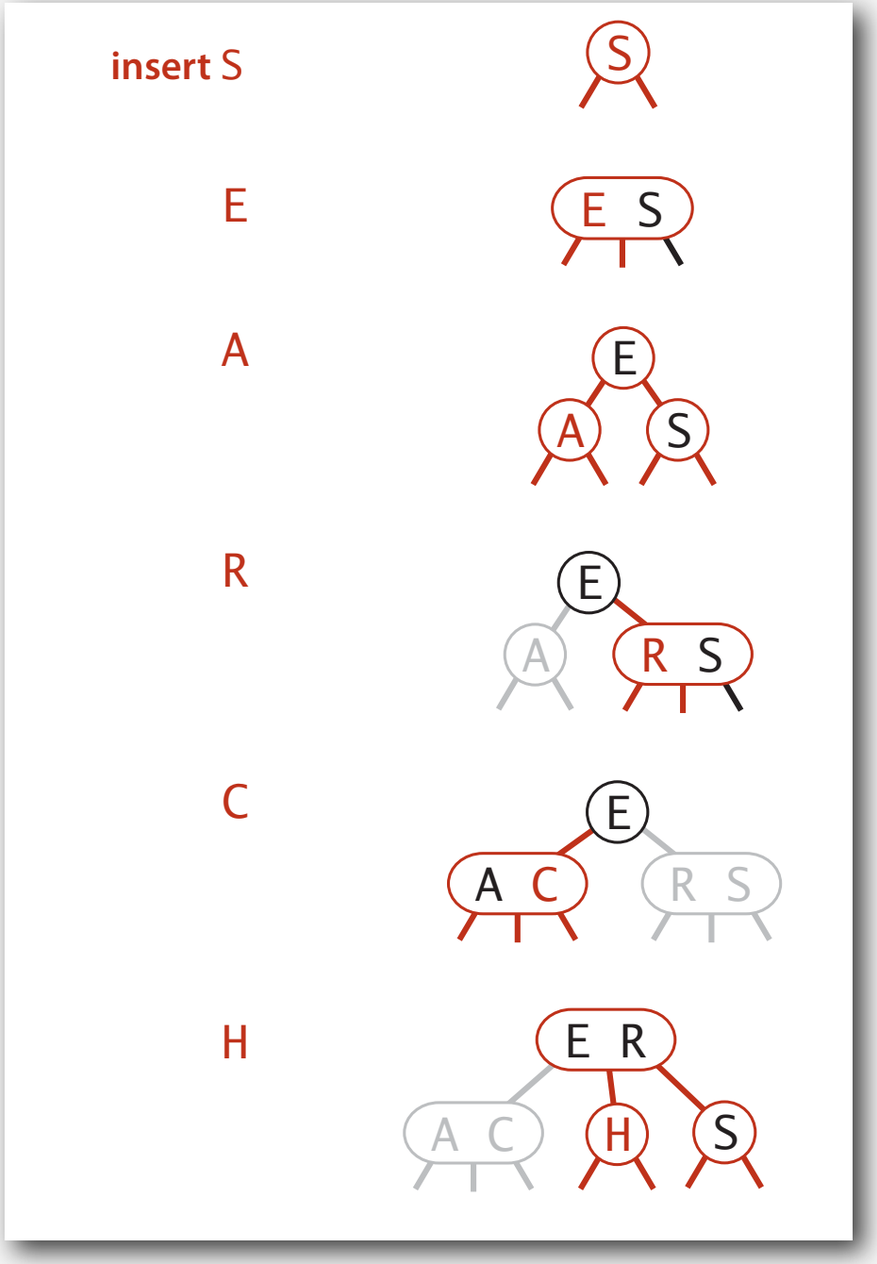
Standard indexing client.

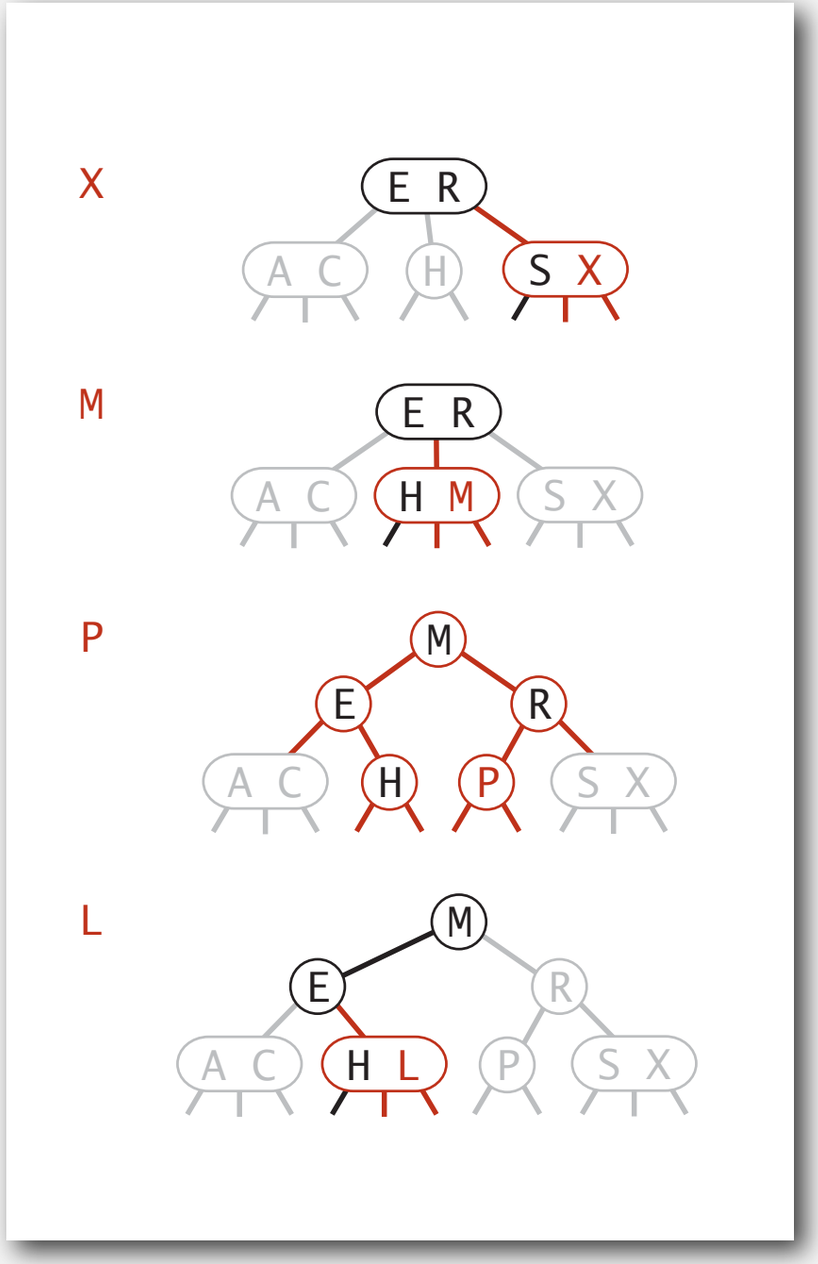




2-3 tree construction trace

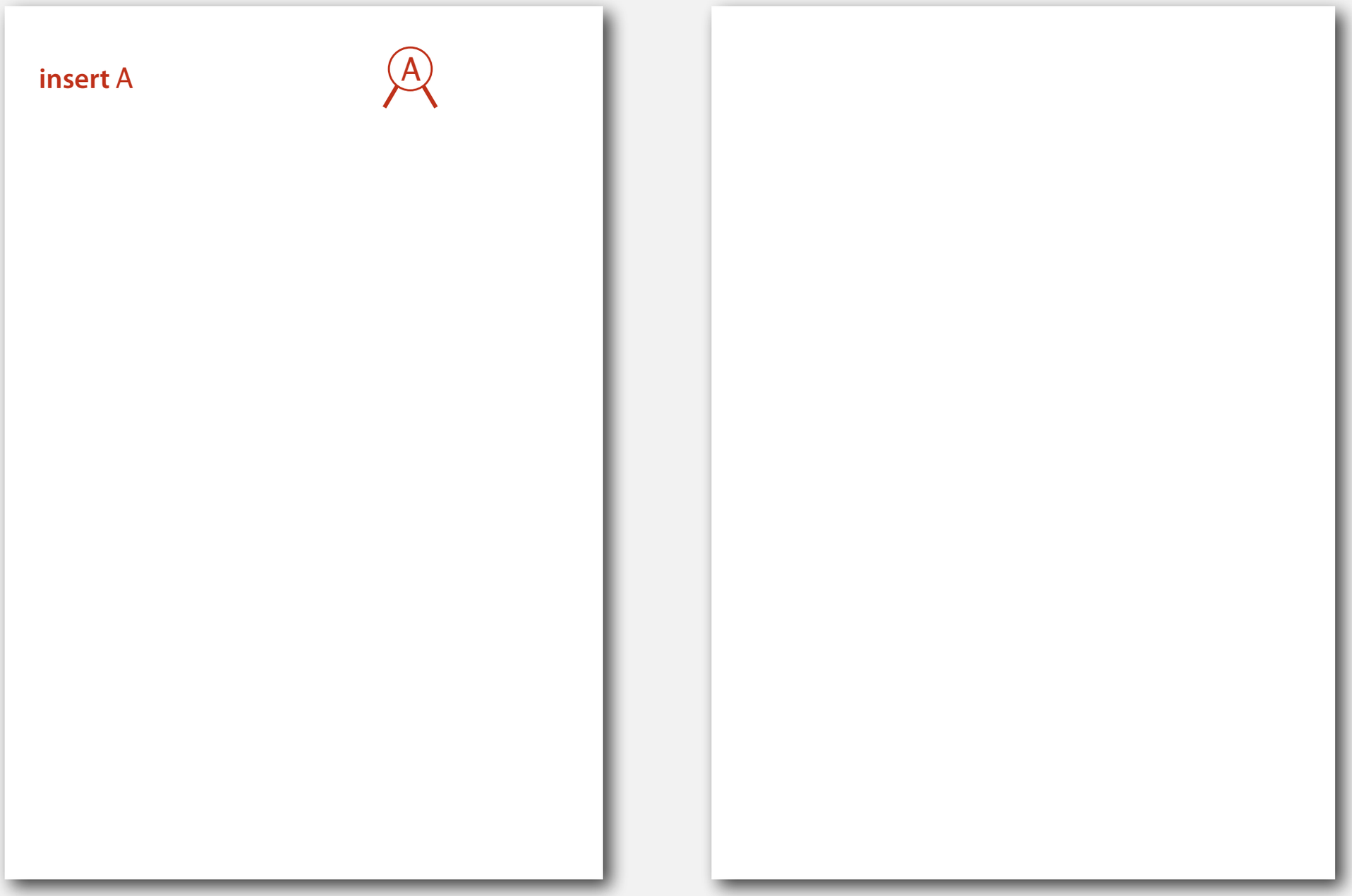
Standard indexing client.





2-3 tree construction trace

The same keys inserted in ascending order.



2-3 tree construction trace

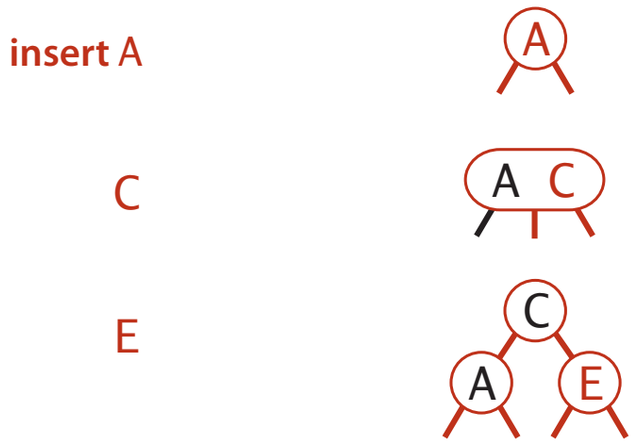
The same keys inserted in ascending order.

insert A

C

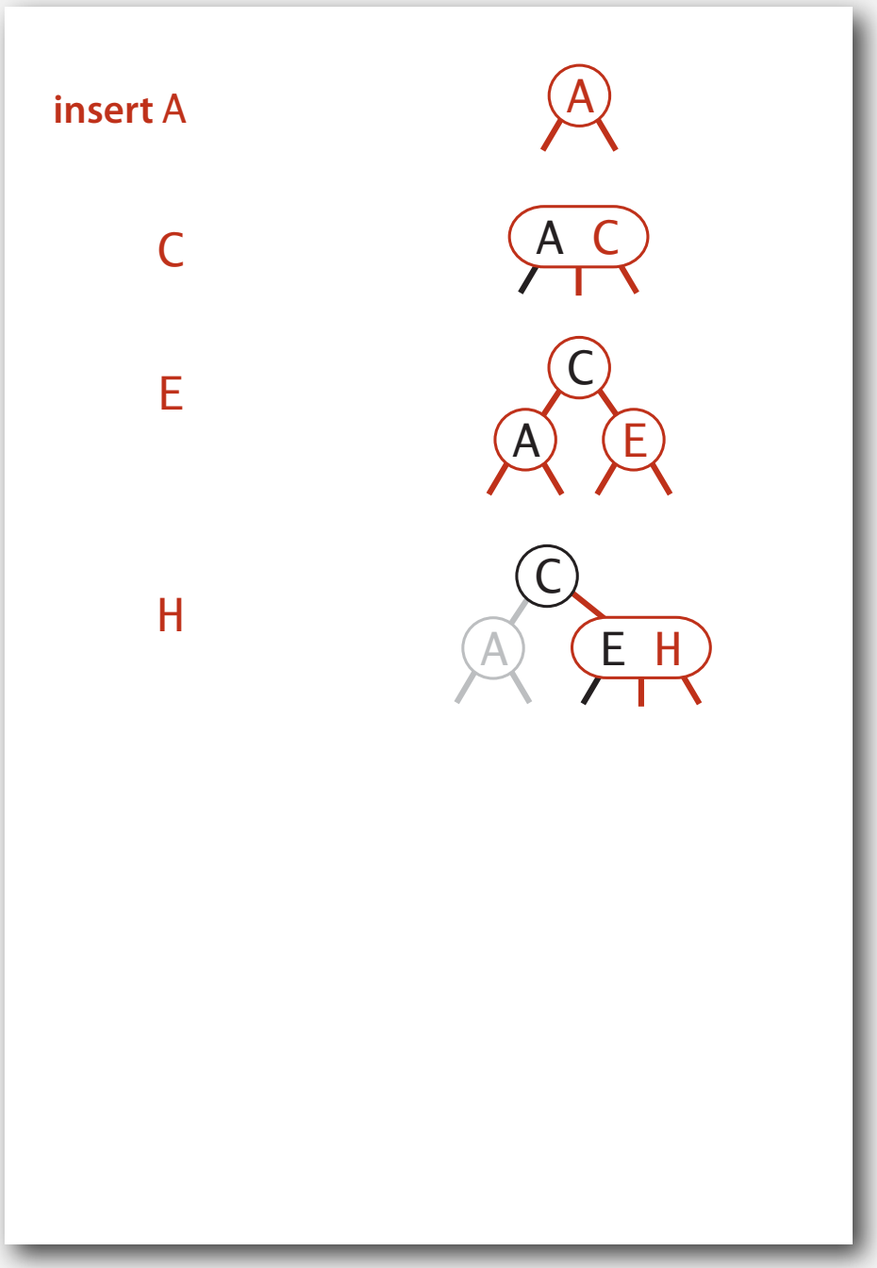
2-3 tree construction trace

The same keys inserted in ascending order.



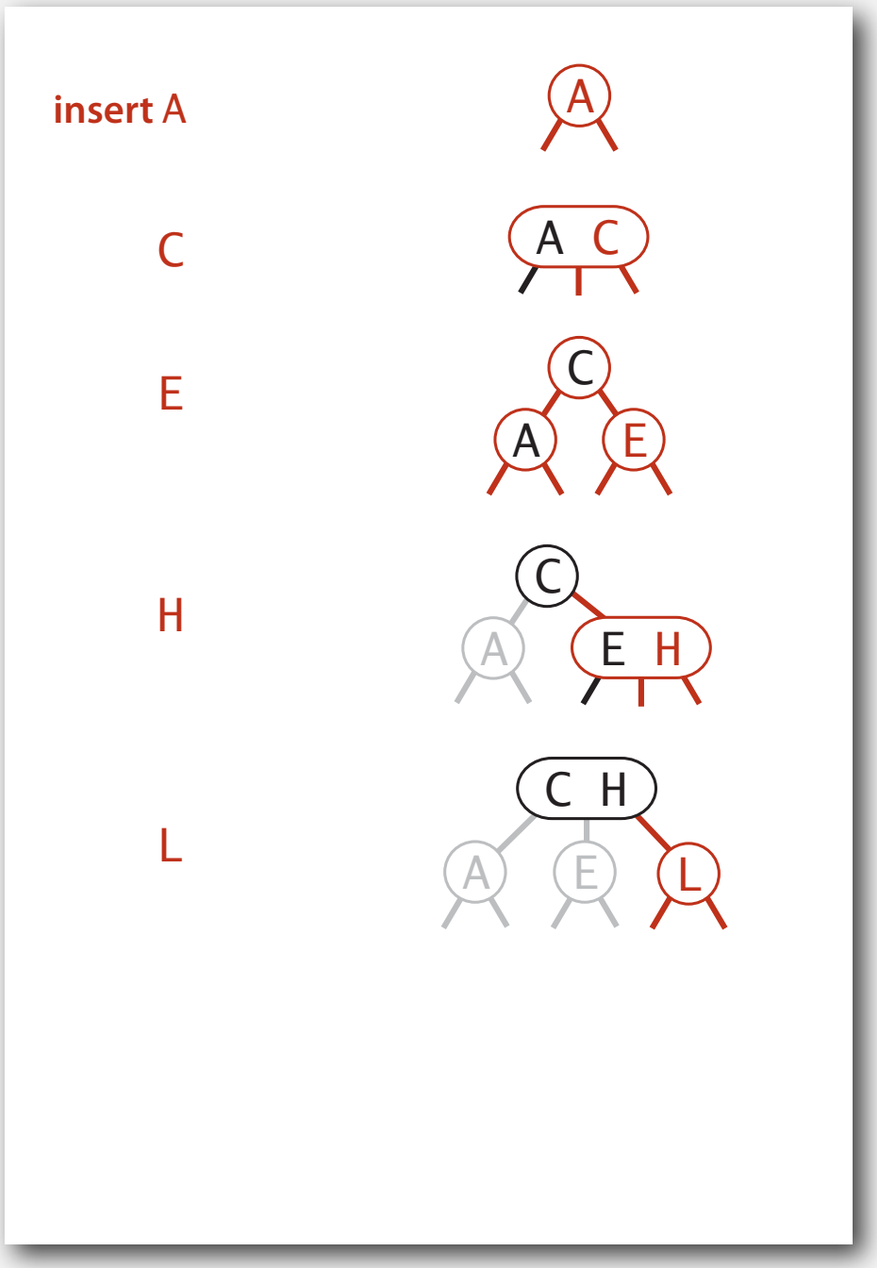
2-3 tree construction trace

The same keys inserted in ascending order.



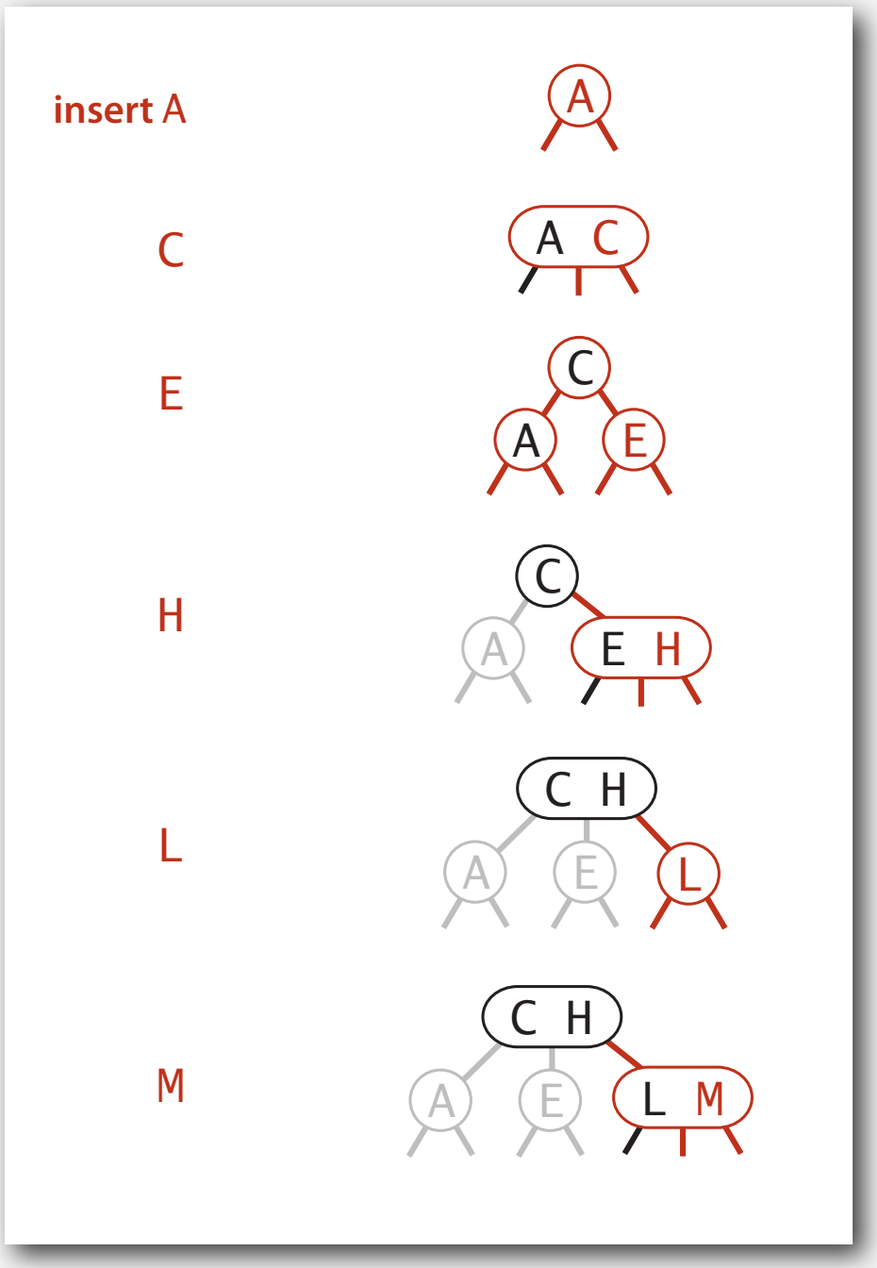
2-3 tree construction trace

The same keys inserted in ascending order.



2-3 tree construction trace

The same keys inserted in ascending order.



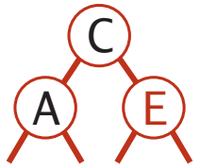
2-3 tree construction trace

The same keys inserted in ascending order.

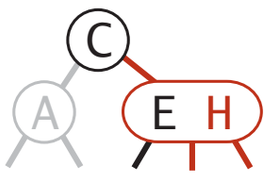
insert A

C

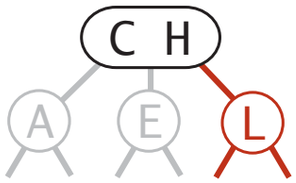
E



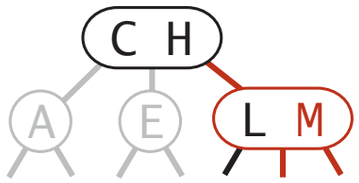
H



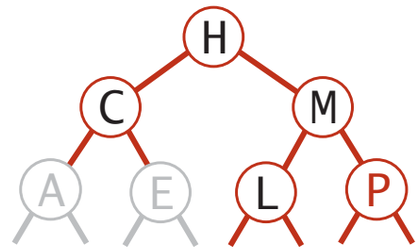
L



M



P



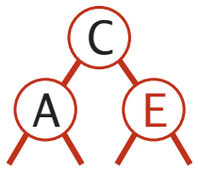
2-3 tree construction trace

The same keys inserted in ascending order.

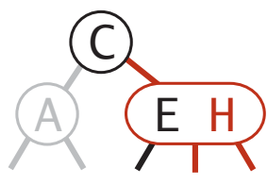
insert A

C

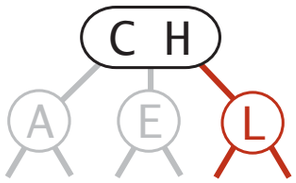
E



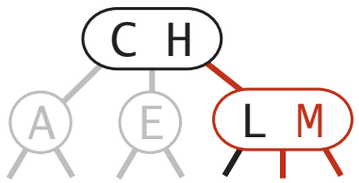
H



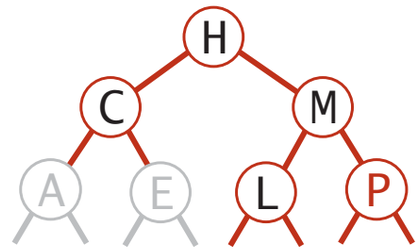
L



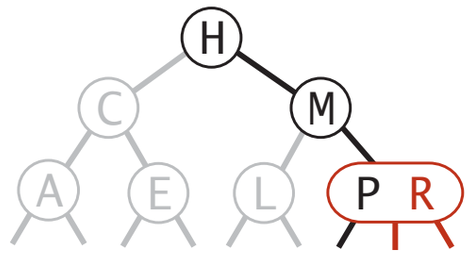
M



P



R



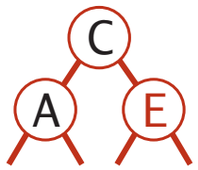
2-3 tree construction trace

The same keys inserted in ascending order.

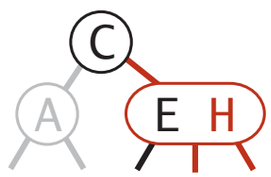
insert A

C

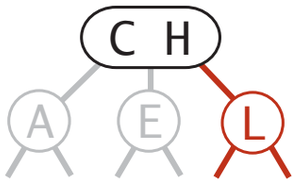
E



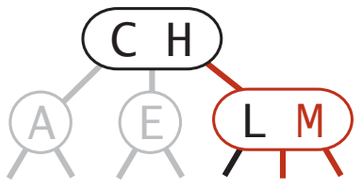
H



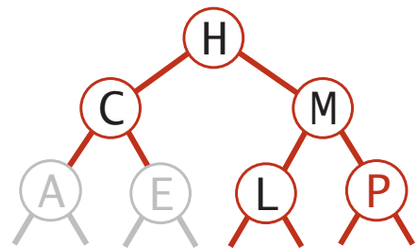
L



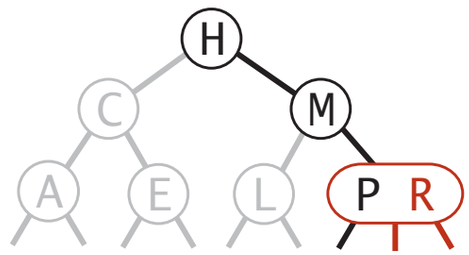
M



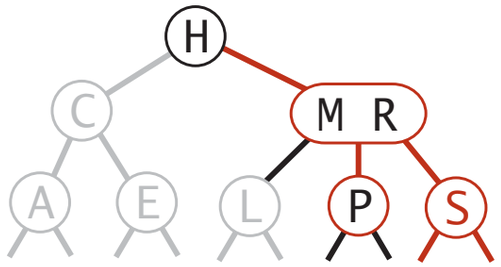
P



R

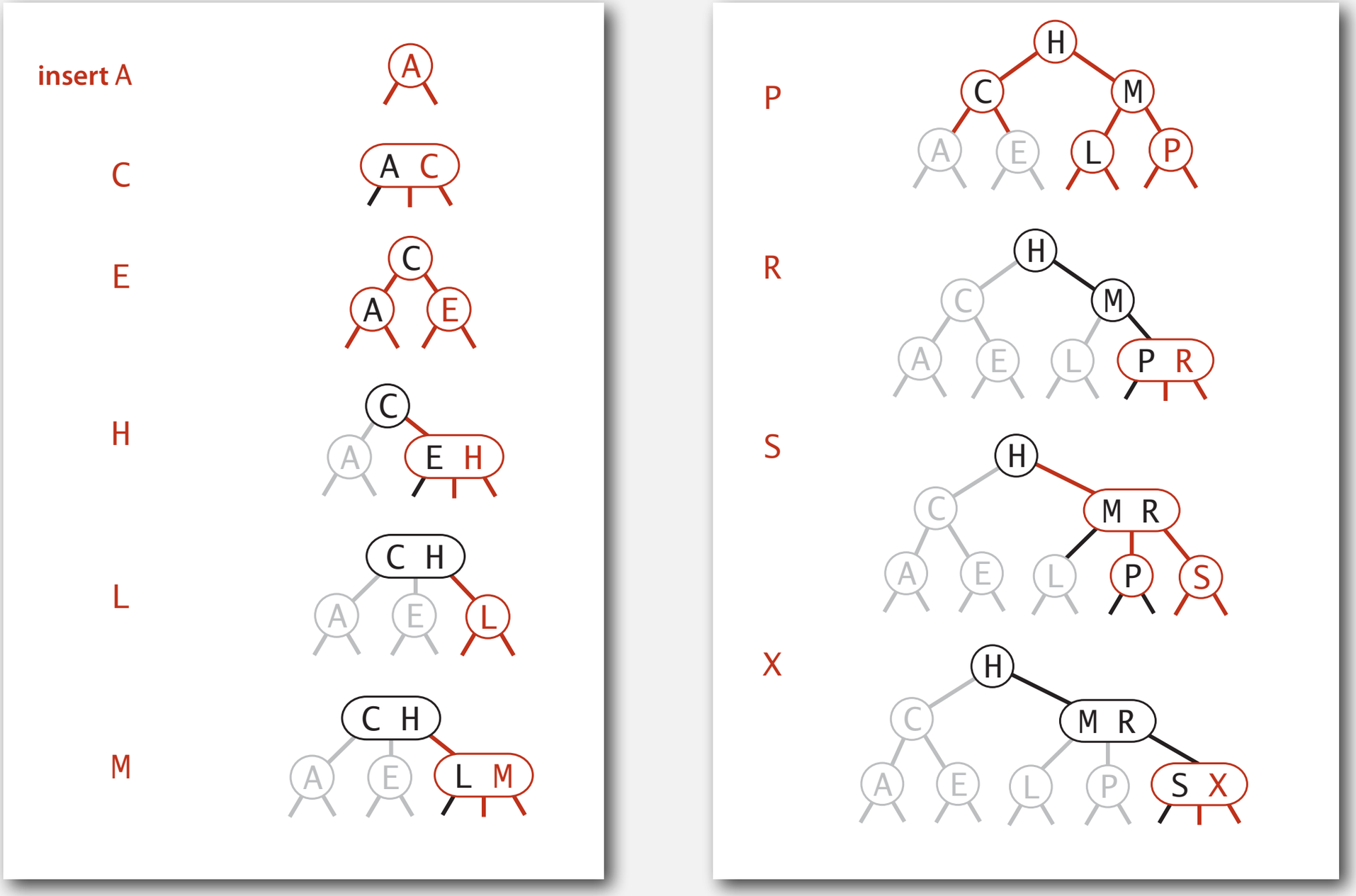


S



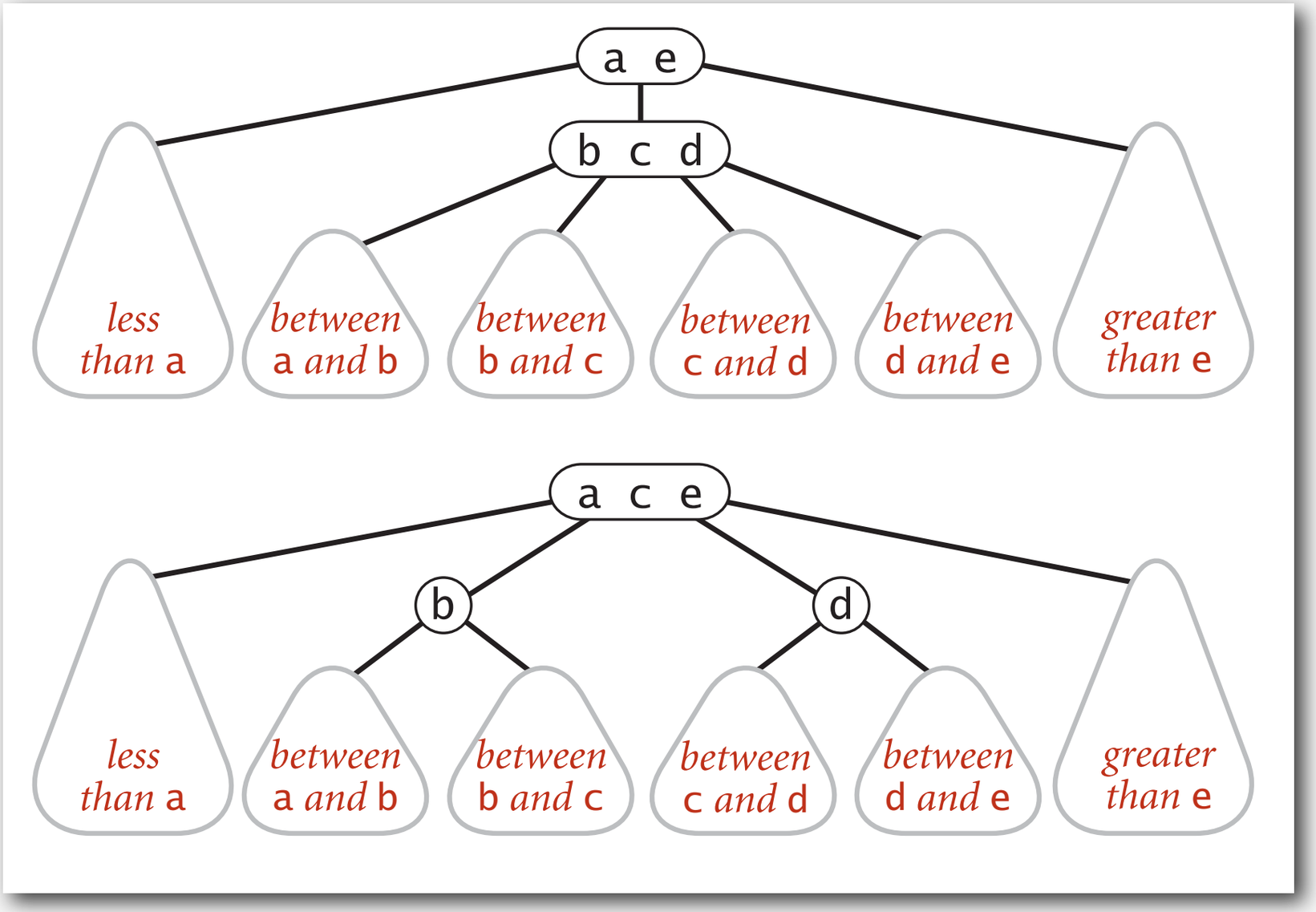
2-3 tree construction trace

The same keys inserted in ascending order.



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

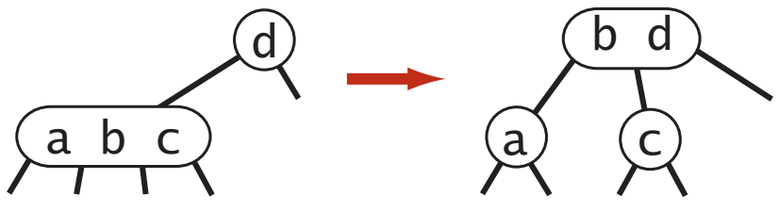
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.

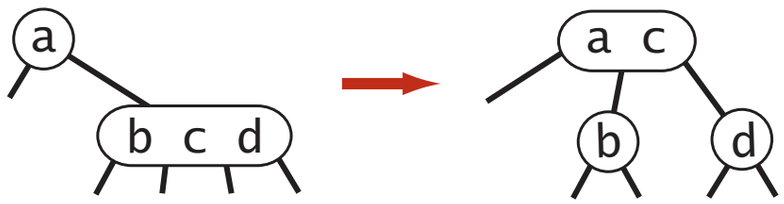
root

parent is a 2-node

left

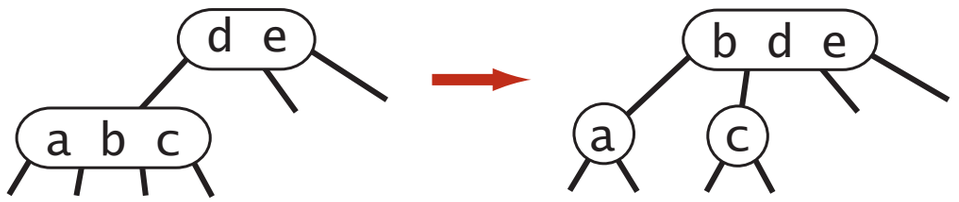


right

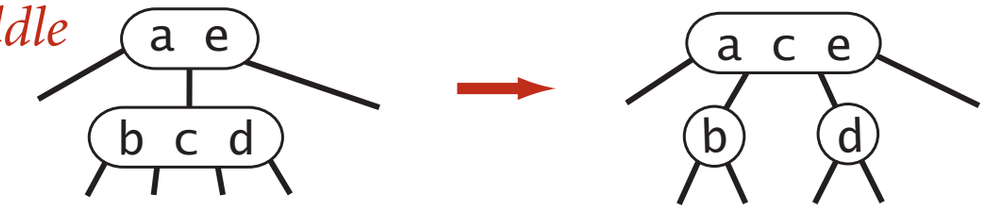


parent is a 3-node

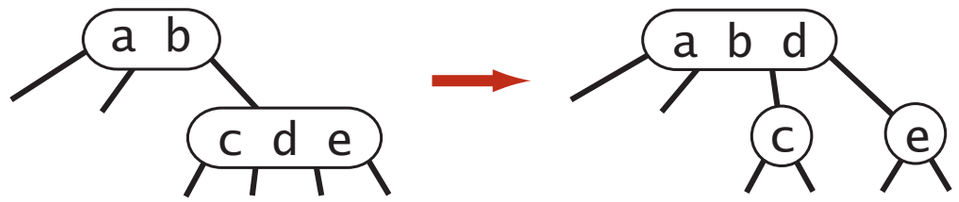
left



middle

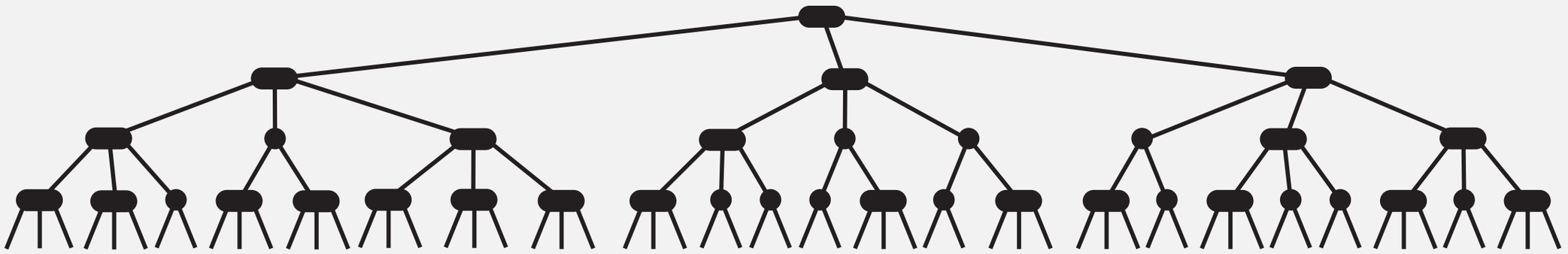


right



2-3 tree: performance

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

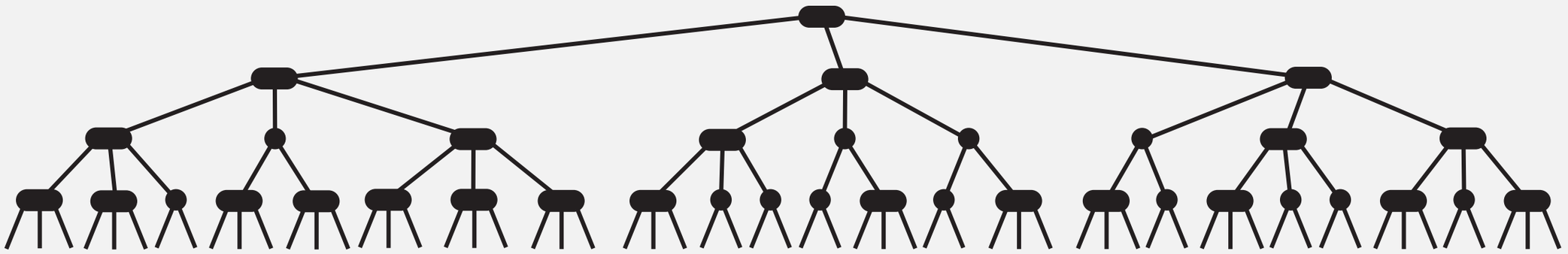


Tree height.

- 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.

Guaranteed logarithmic performance for search and insert.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`

2-3 tree: implementation?

Direct implementation is complicated, because:

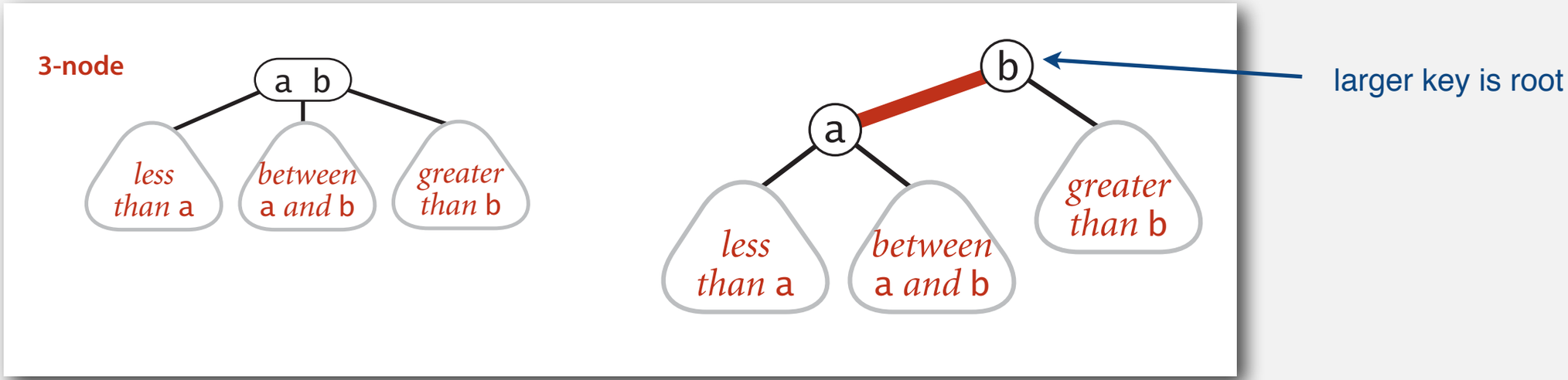
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

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

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

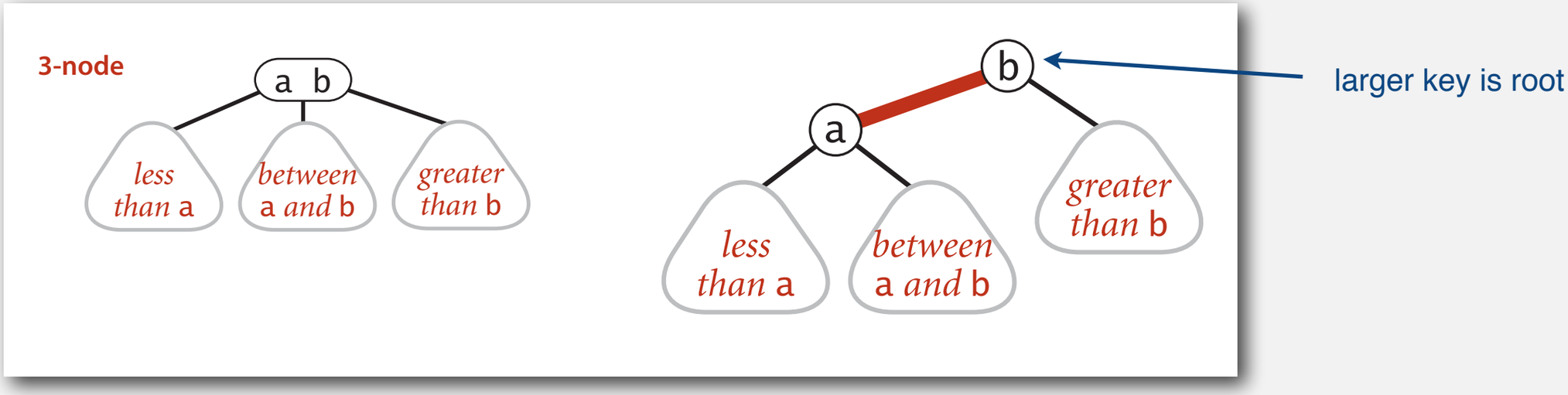
Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

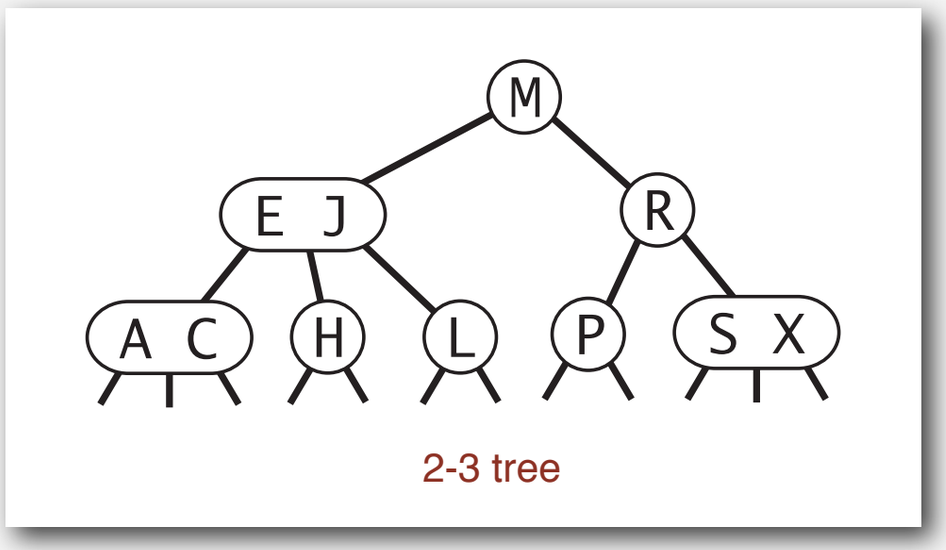
1. Represent 2-3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3-nodes.

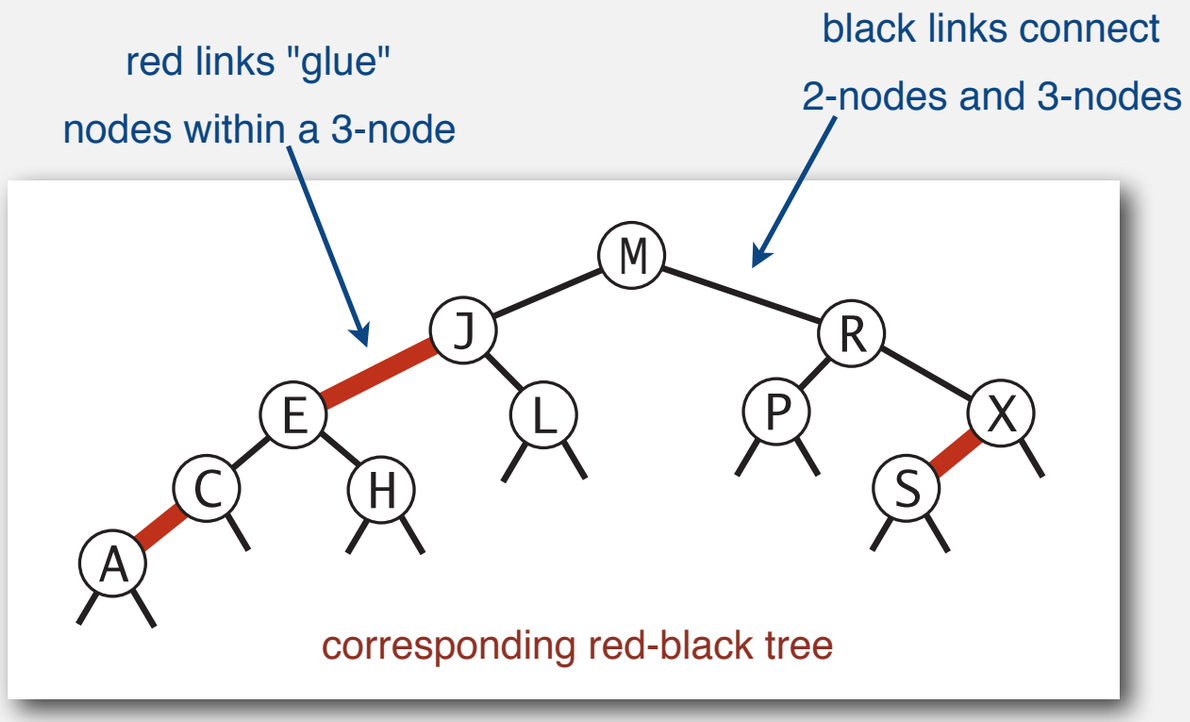


Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

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





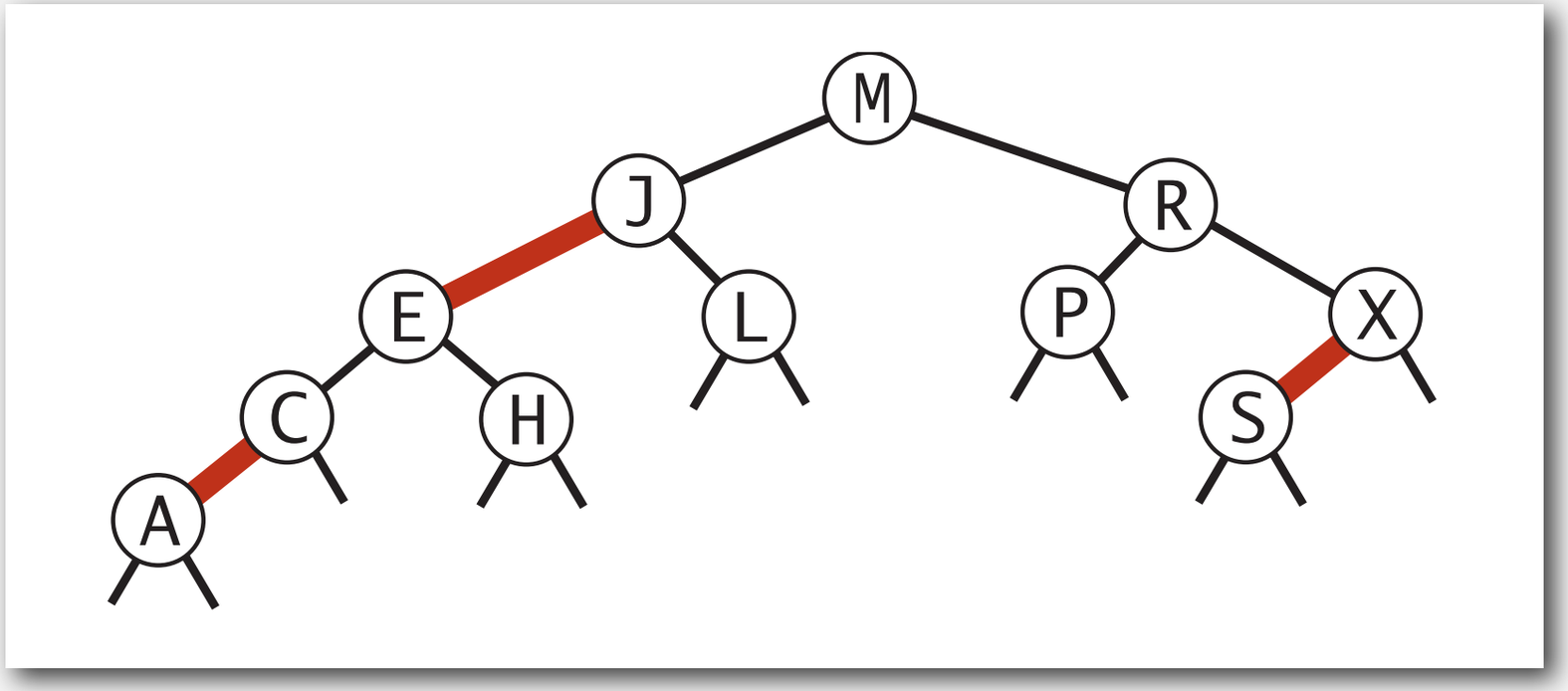


An equivalent definition

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.

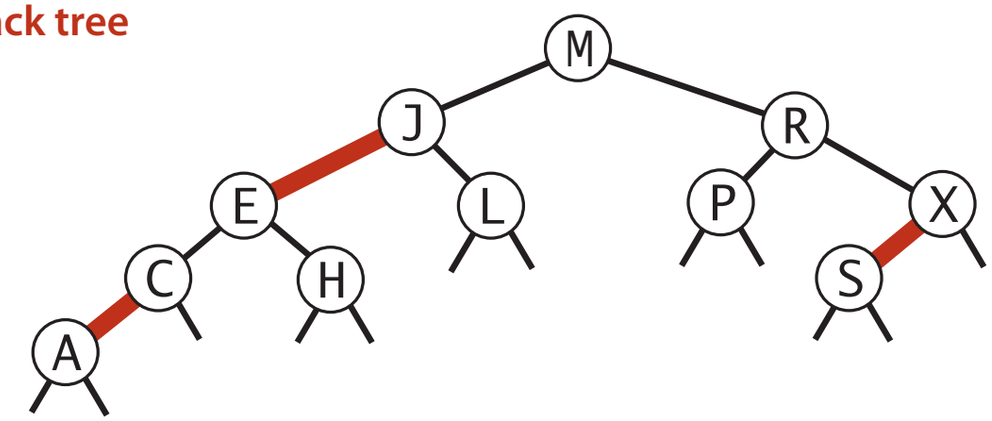
"perfect black balance"



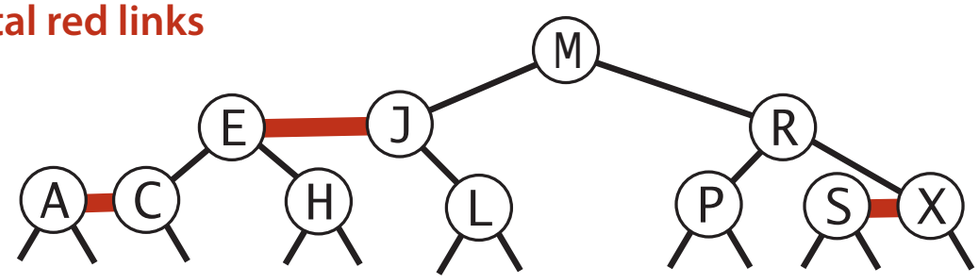
Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

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

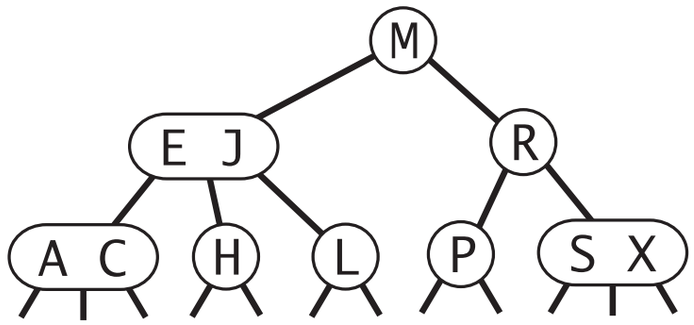
red–black tree



horizontal red links



2-3 tree

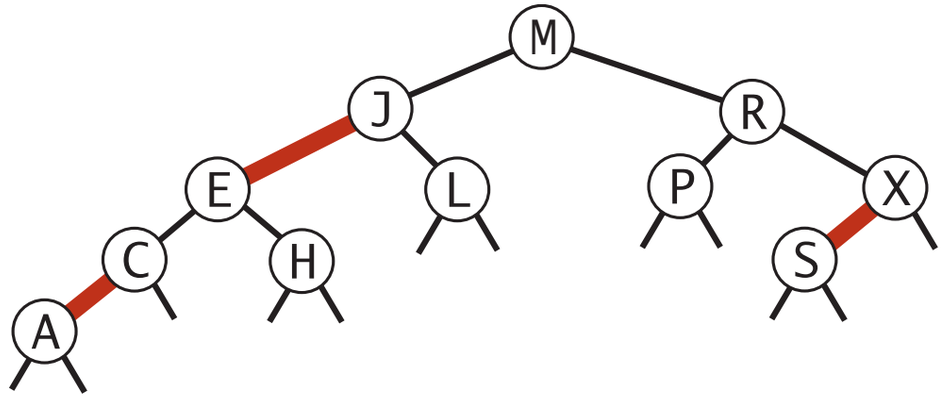


Search implementation for red-black BSTs

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;
}
```



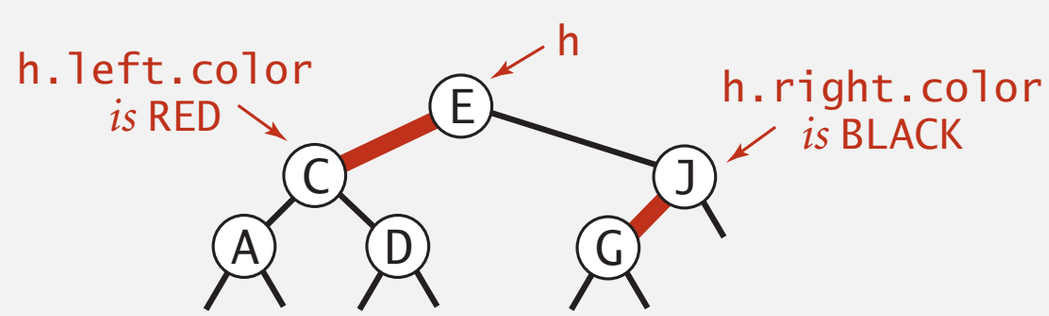
Remark. Most other ops (e.g., ceiling, selection, iteration) are also identical.

Red-black BST representation

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

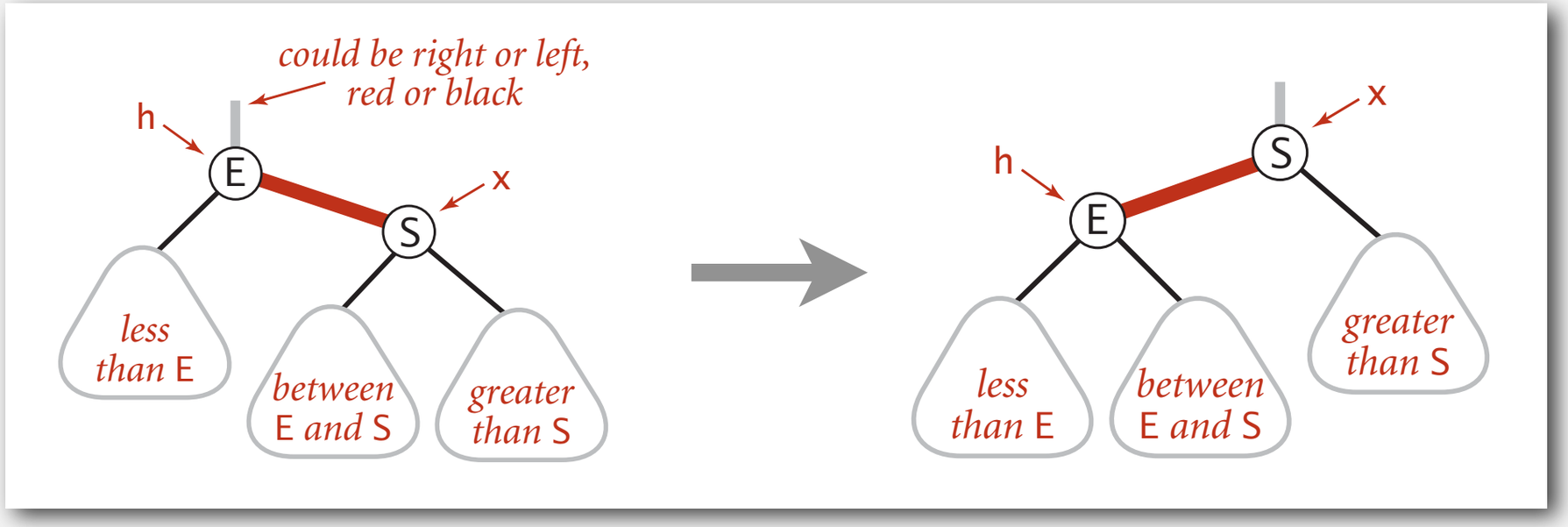
```
private static final boolean RED   = true;
private static final boolean BLACK = false;

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



Elementary red-black BST operations

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

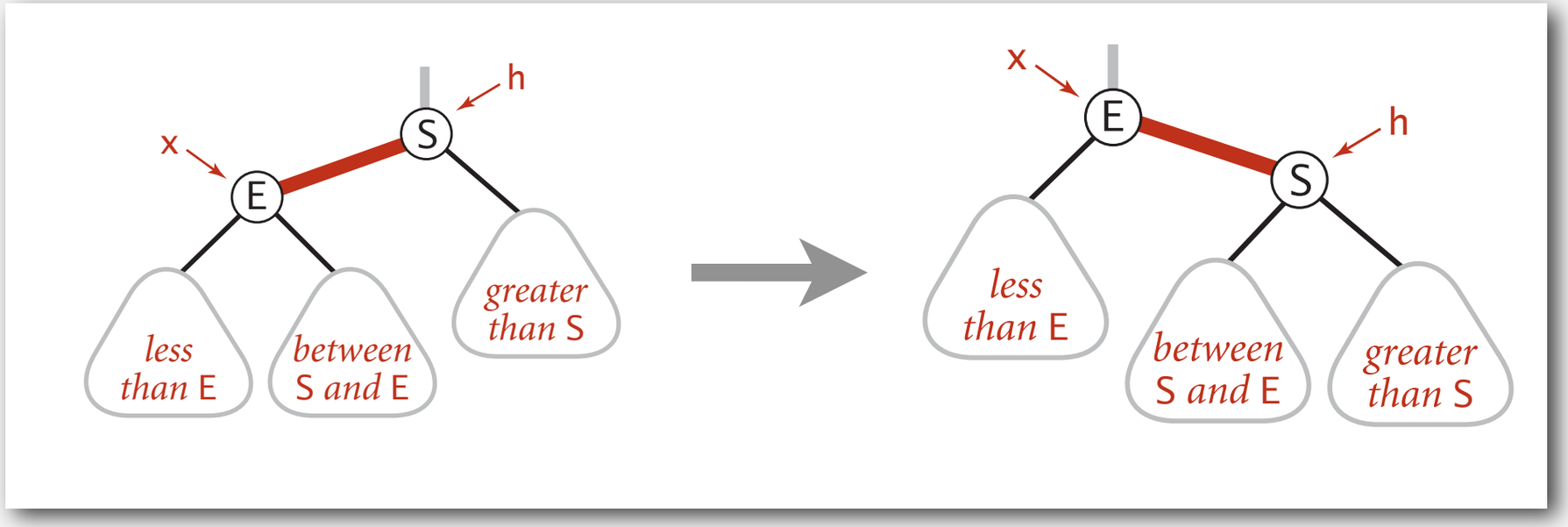



```
private Node rotateLeft(Node h)
{
   assert (h != null) && isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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.

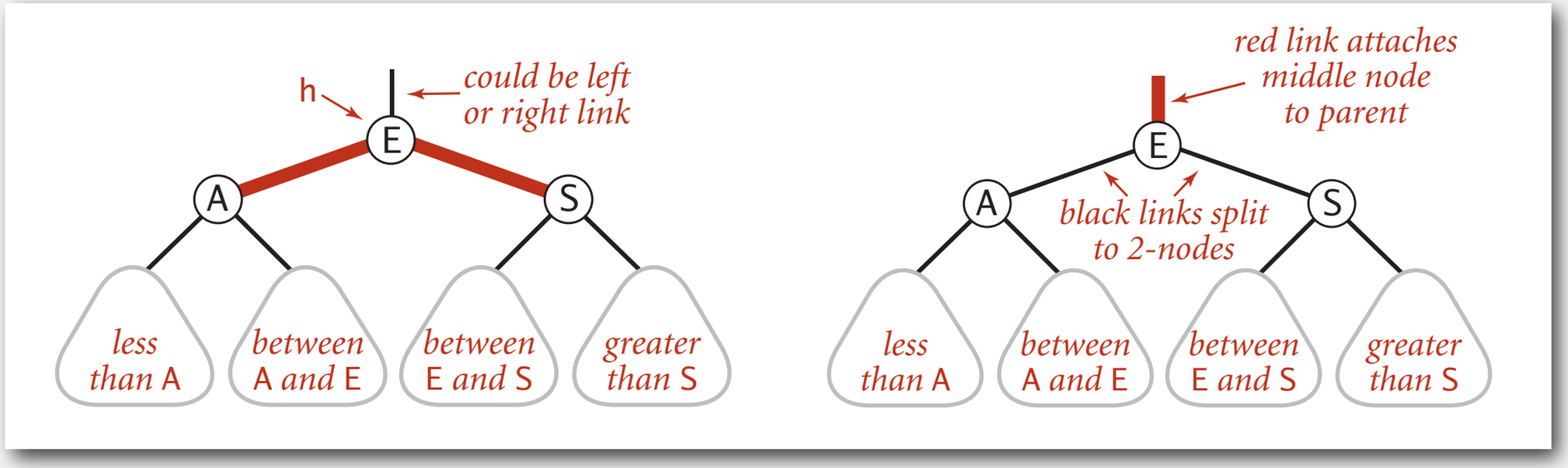



```
private Node rotateRight(Node h)
{
   assert (h != null) && isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

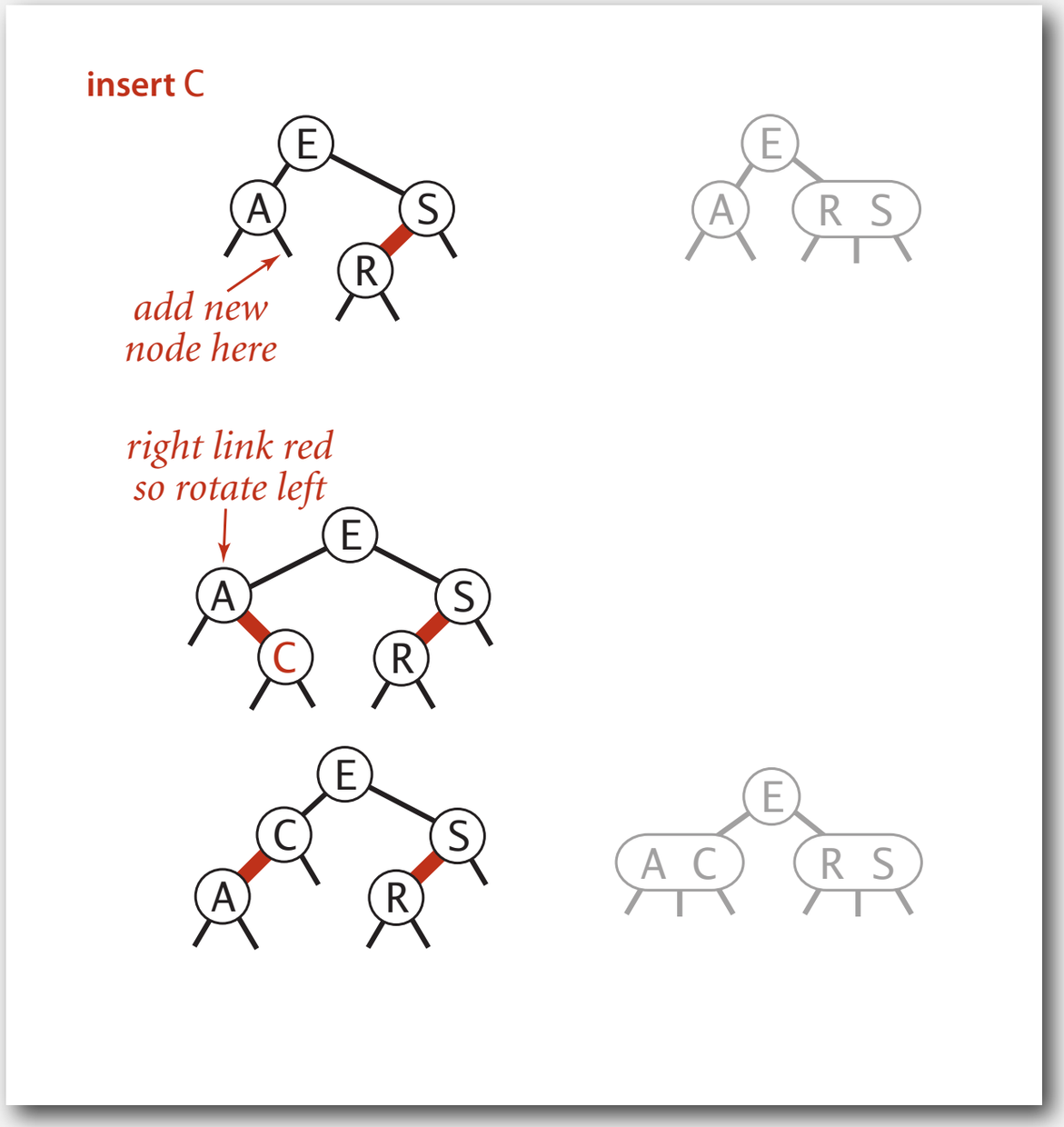



```
private void flipColors(Node h)
{
   assert !isRed(h) && isRed(h.left) && isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

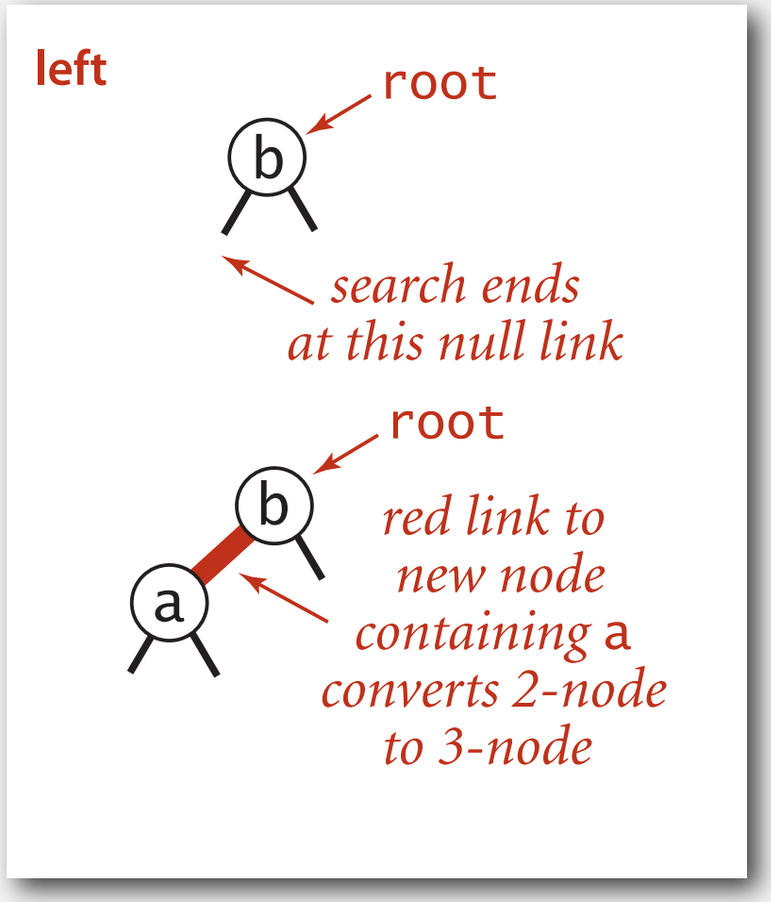
Insertion in a LLRB tree: overview

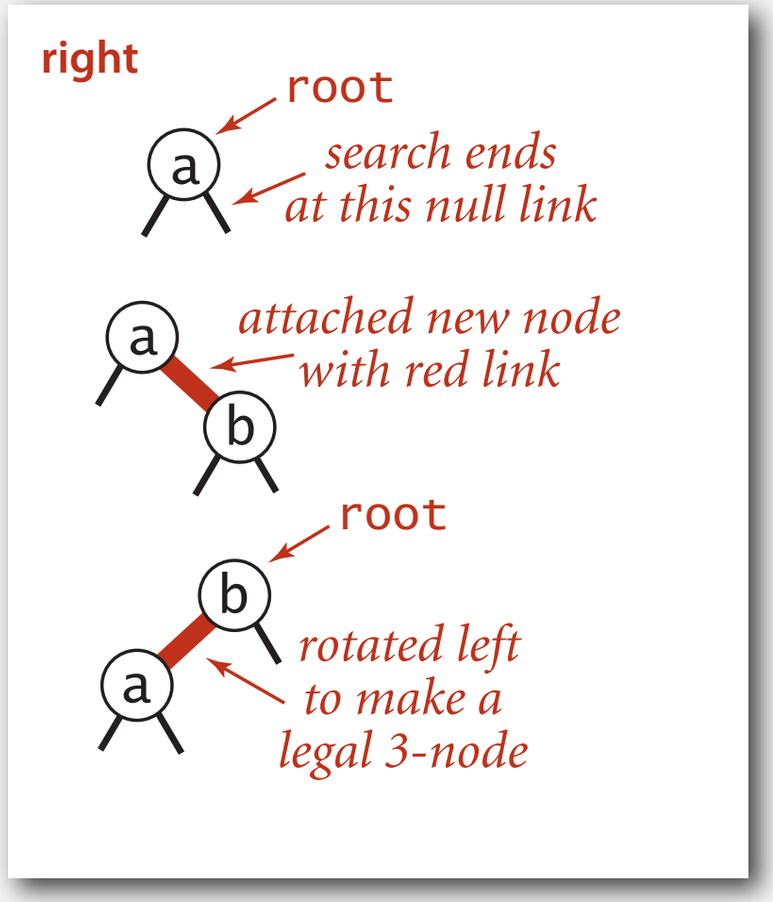
Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black tree operations.



Insertion in a LLRB tree

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

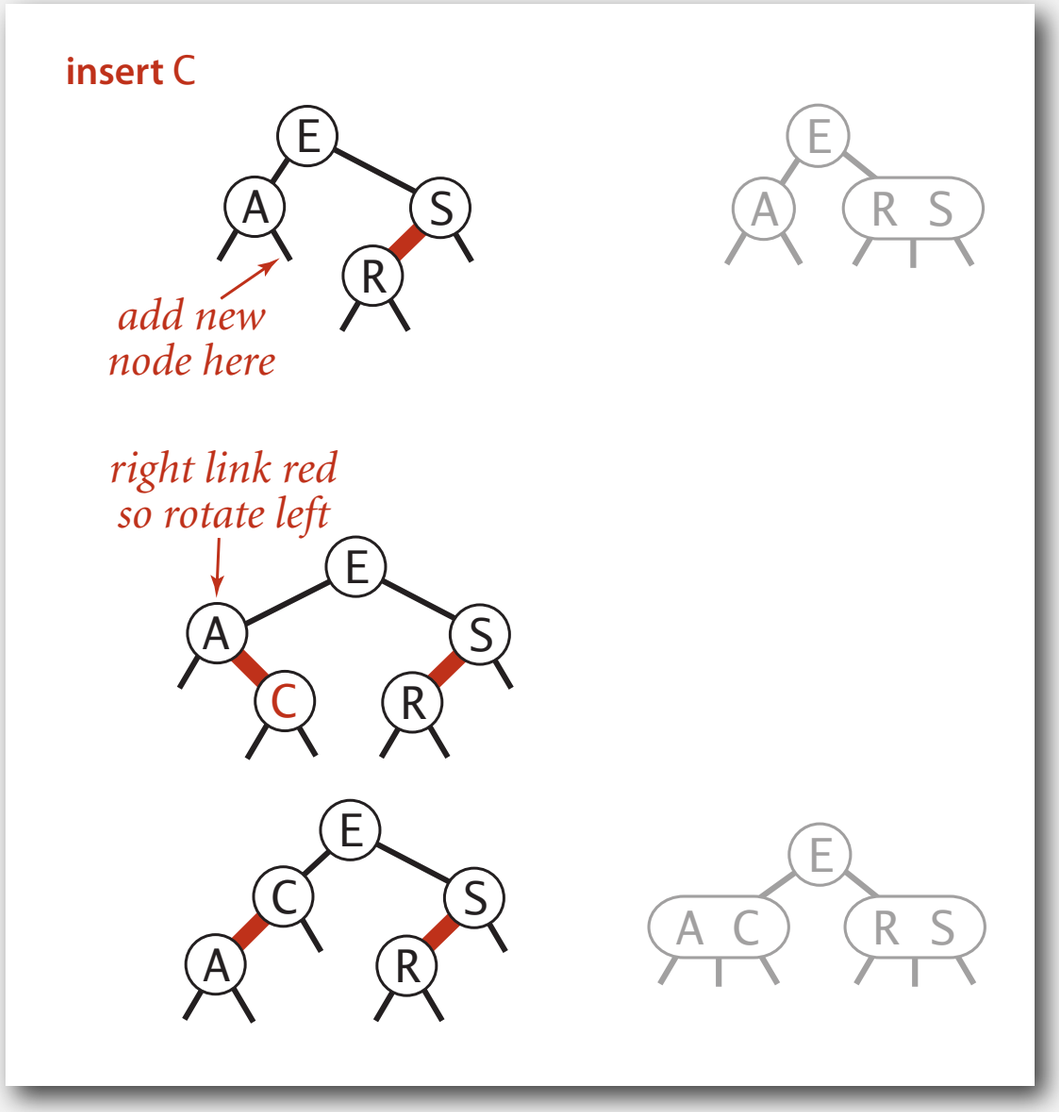




Insertion in a LLRB tree

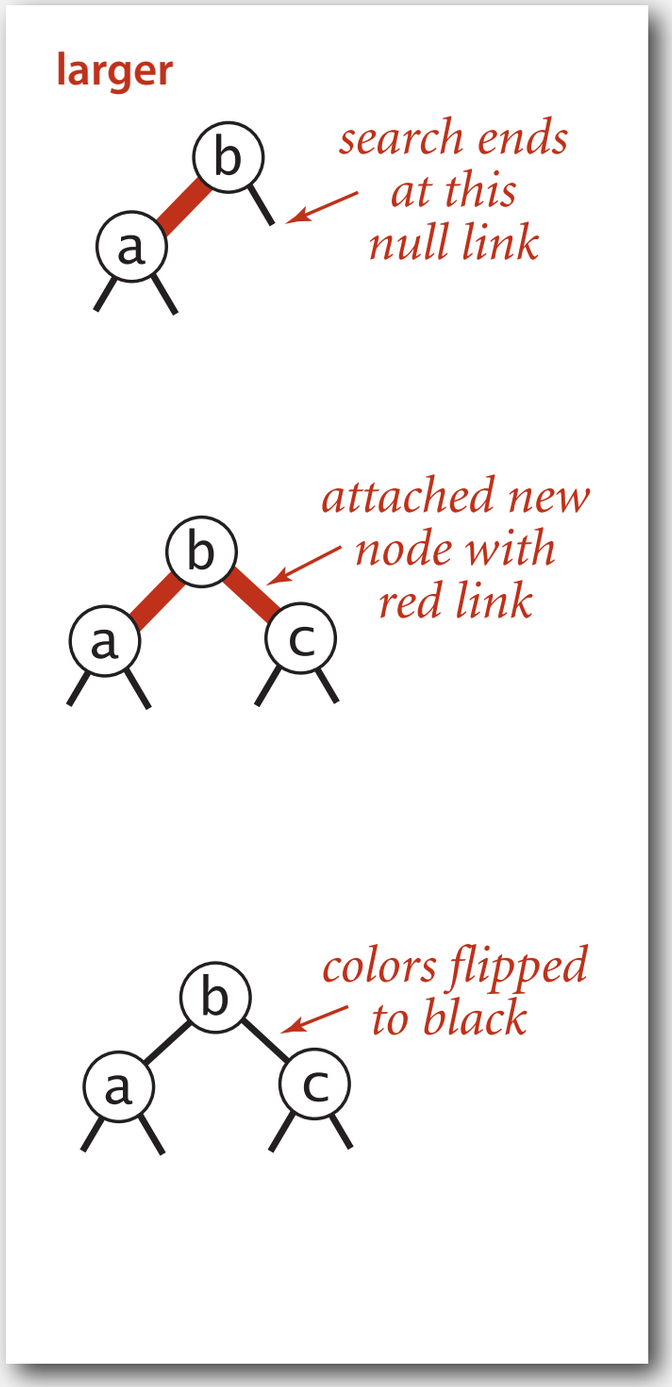
Case 1. Insert into a 2-node at the bottom.

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



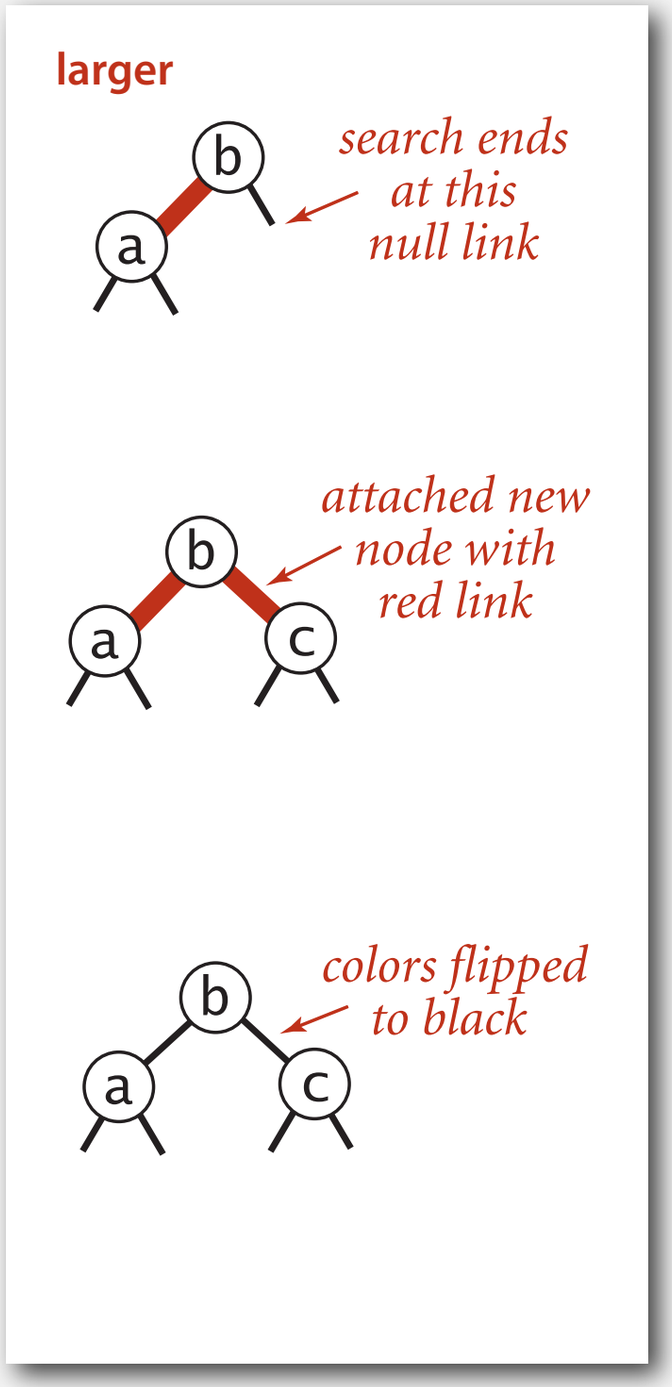
Insertion in a LLRB tree

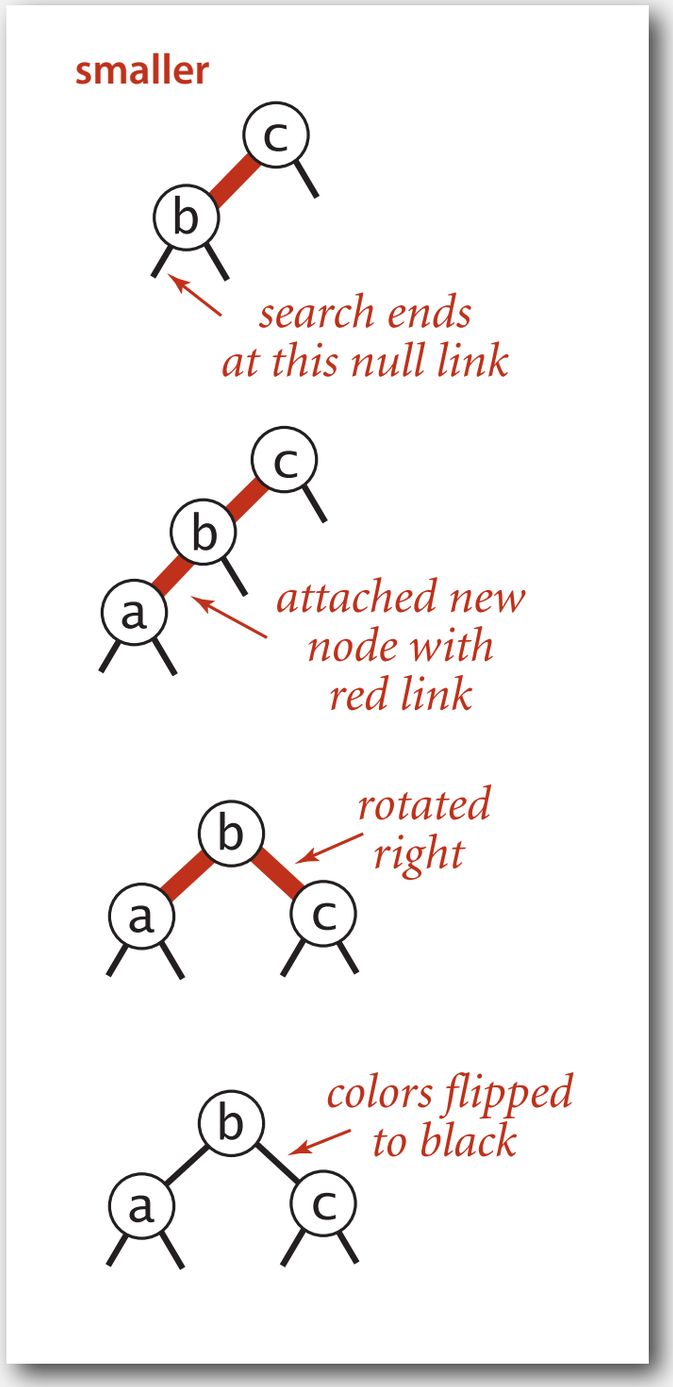
Warmup 2. Insert into a tree with exactly 2 nodes.



Insertion in a LLRB tree

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

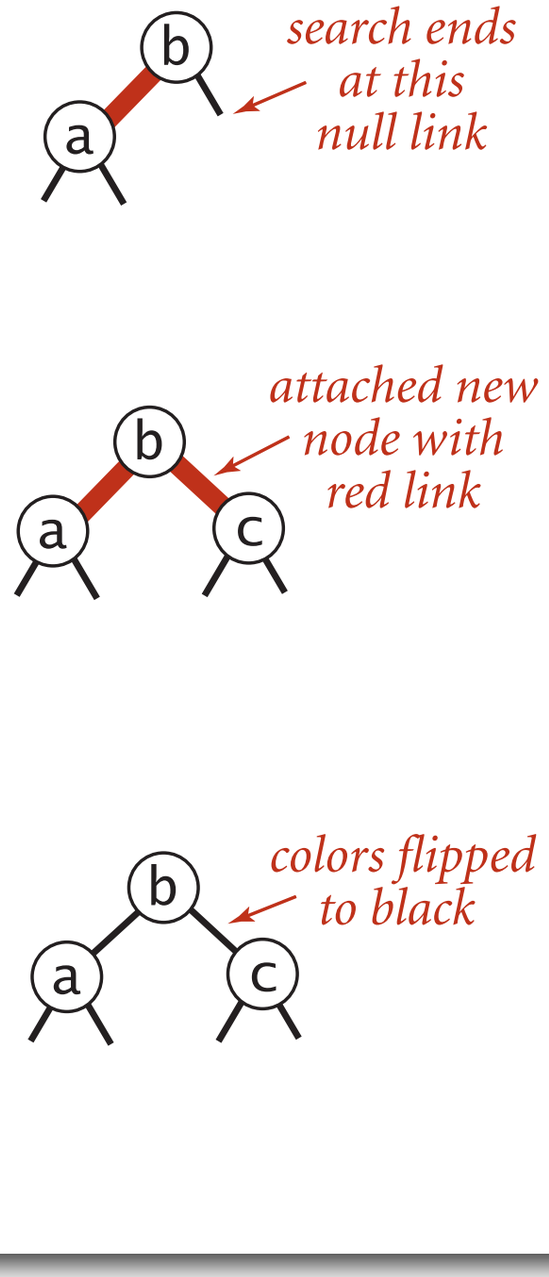




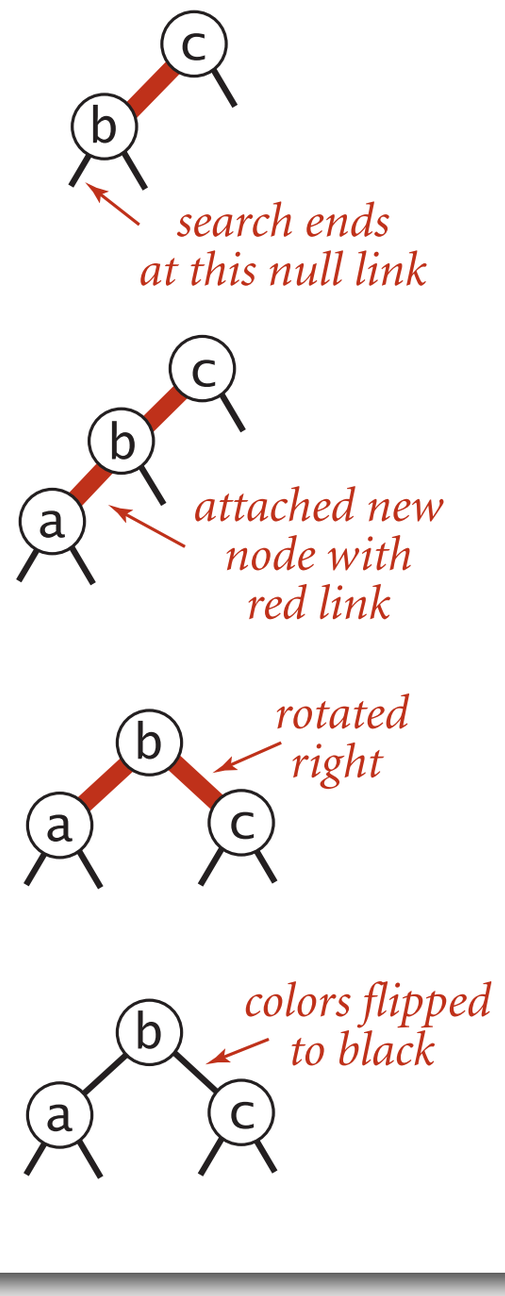
Insertion in a LLRB tree

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

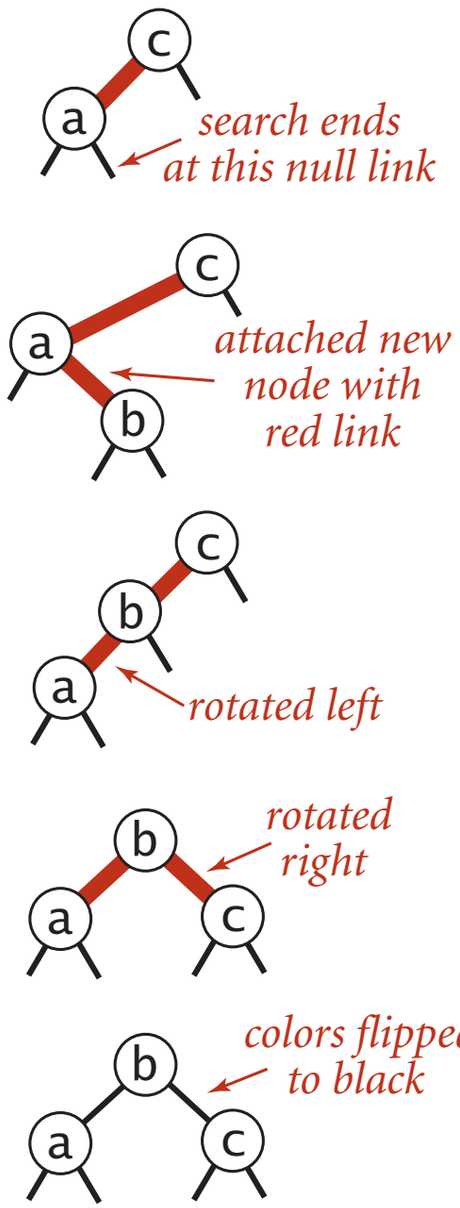
larger



smaller



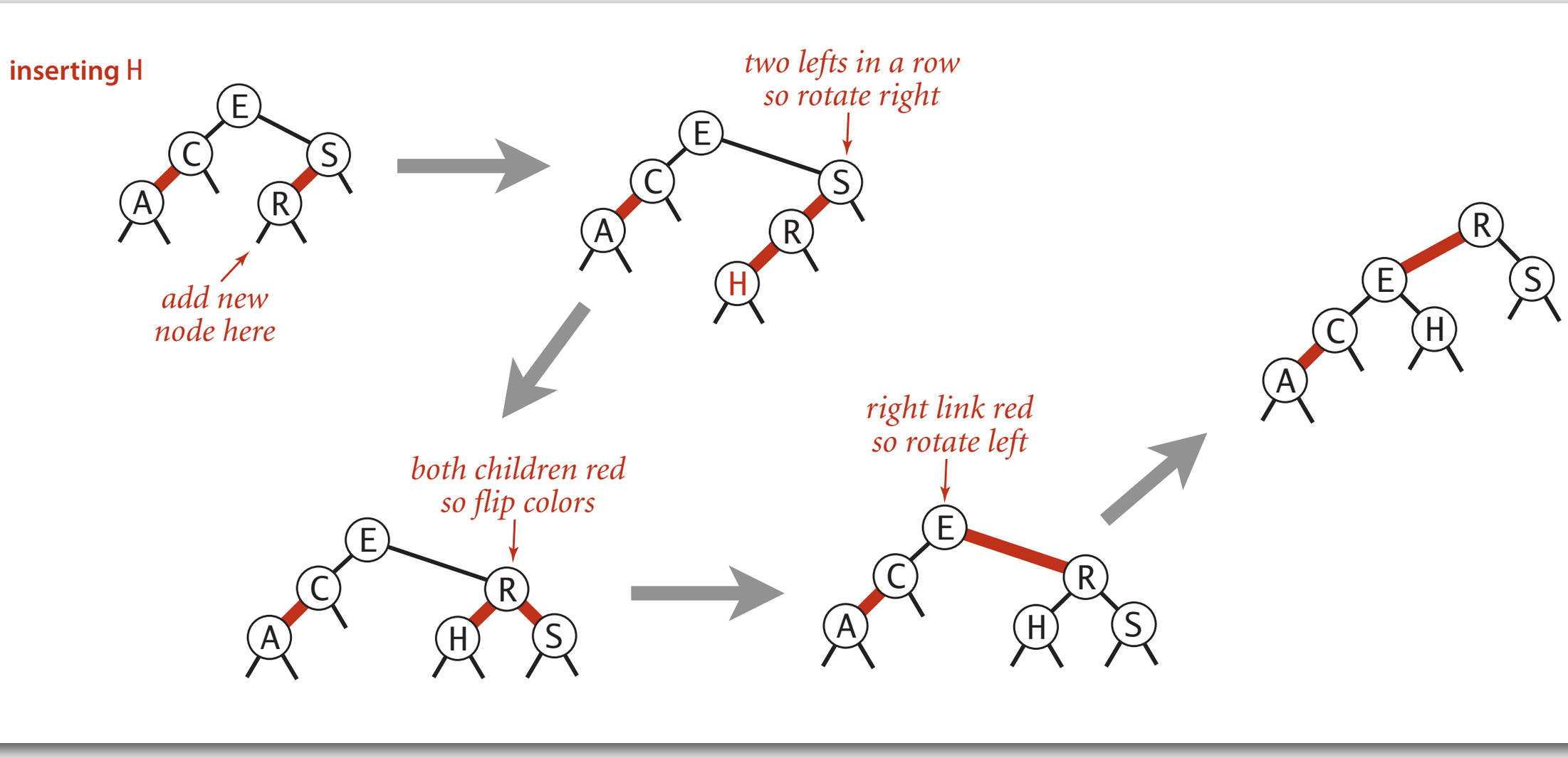
between



Insertion in a LLRB tree

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

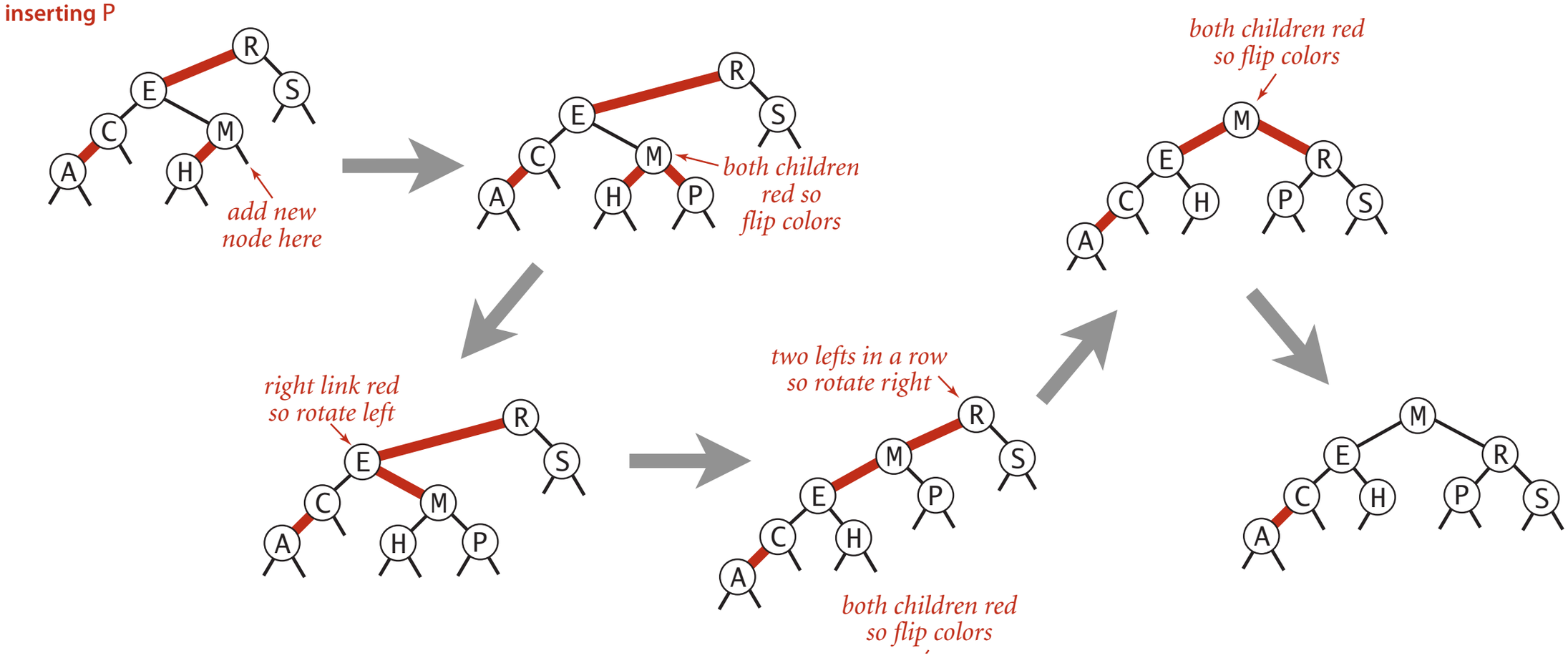
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).



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

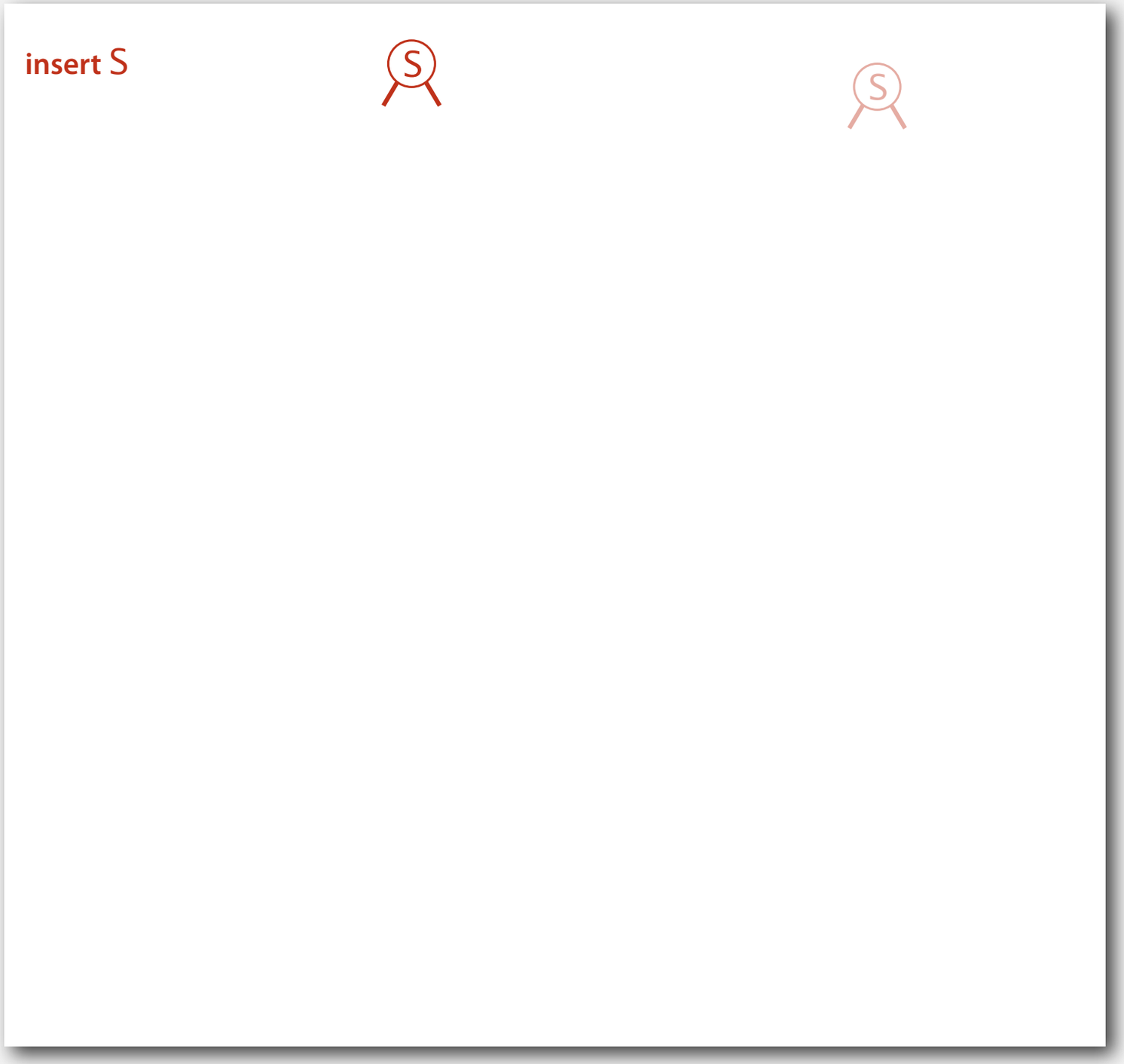
Case 2. Insert into a 3-node at the bottom.

- 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).



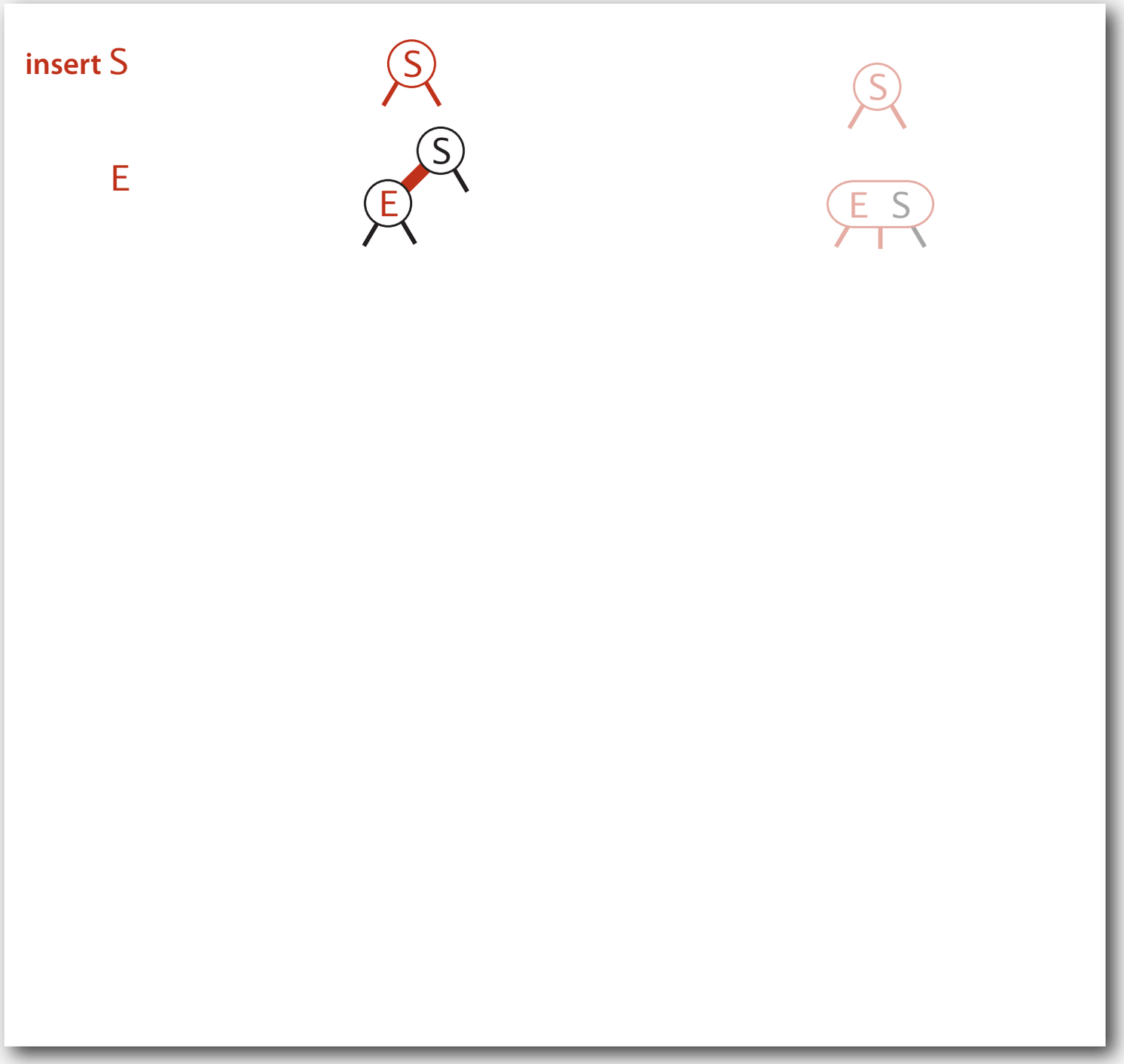
LLRB tree construction trace

Standard indexing client.



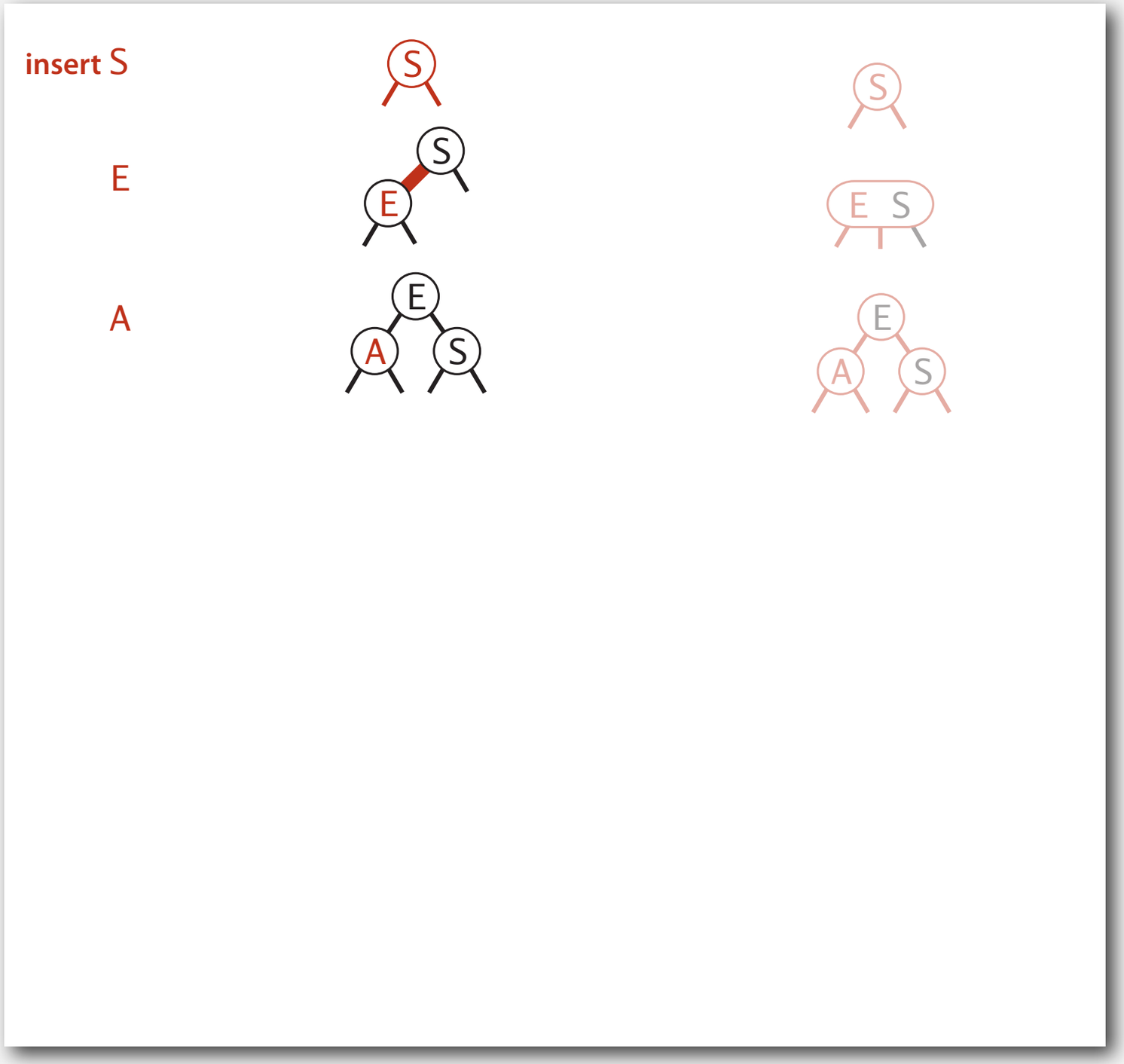
LLRB tree construction trace

Standard indexing client.



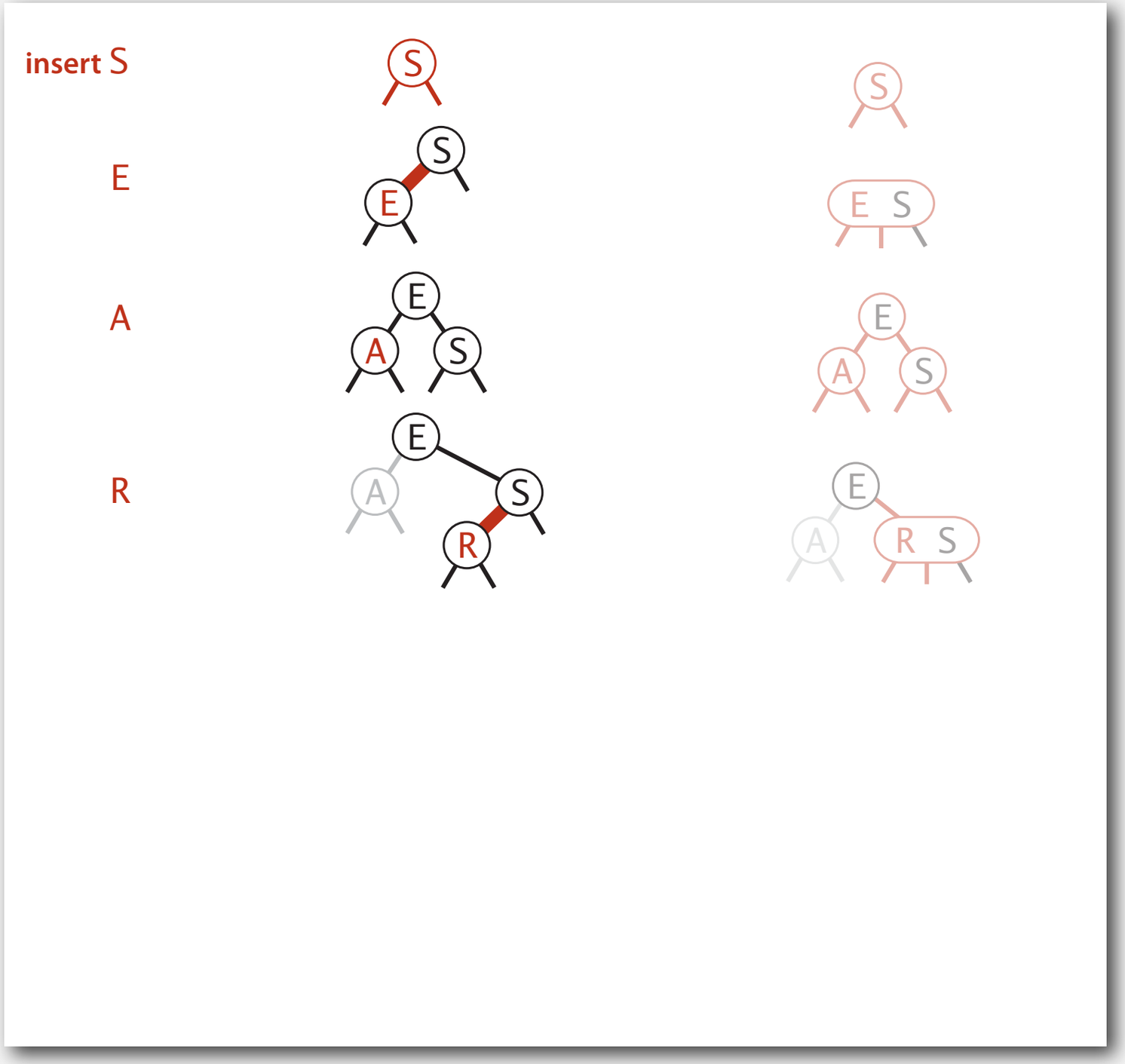
LLRB tree construction trace

Standard indexing client.



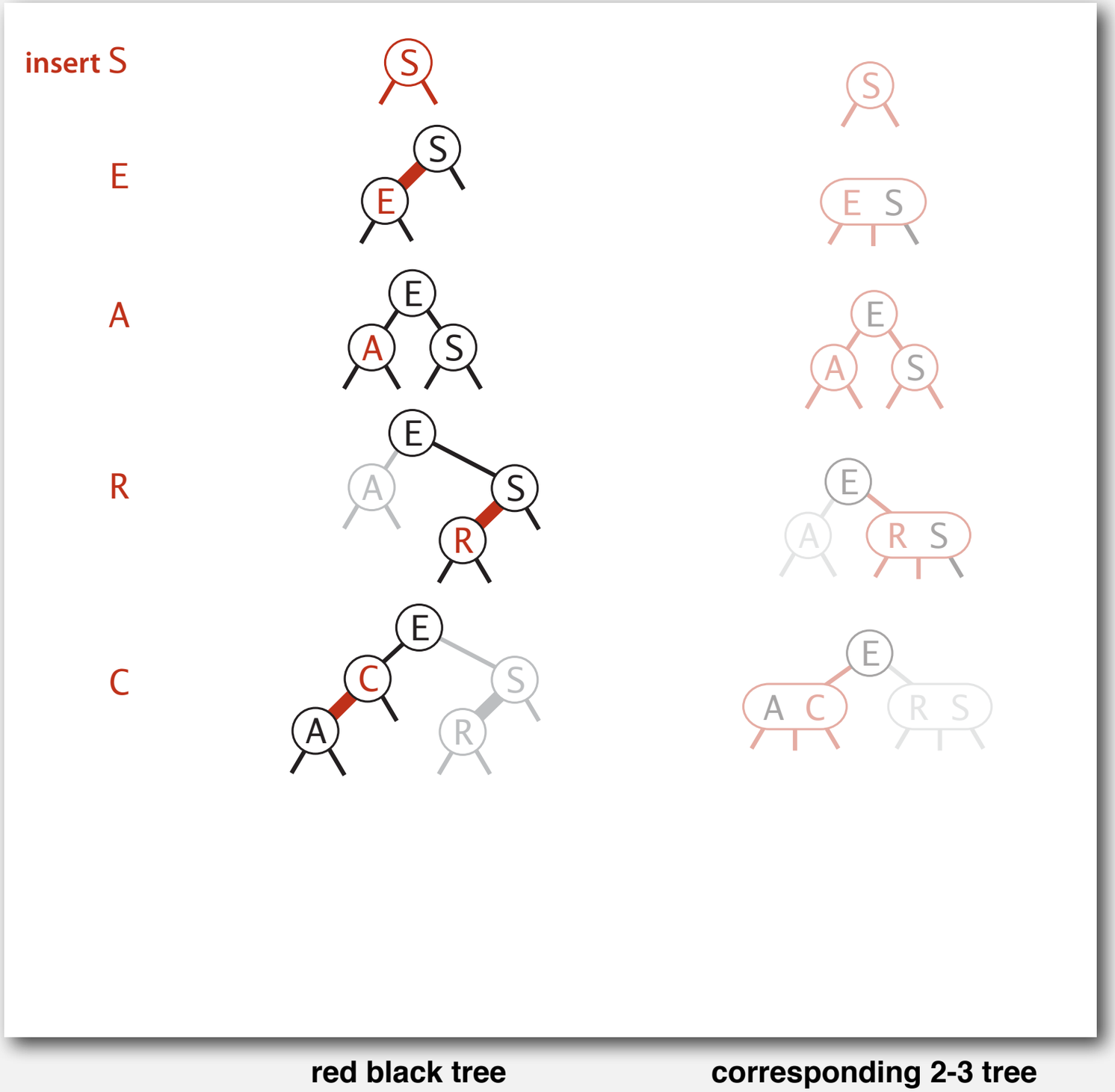
LLRB tree construction trace

Standard indexing client.



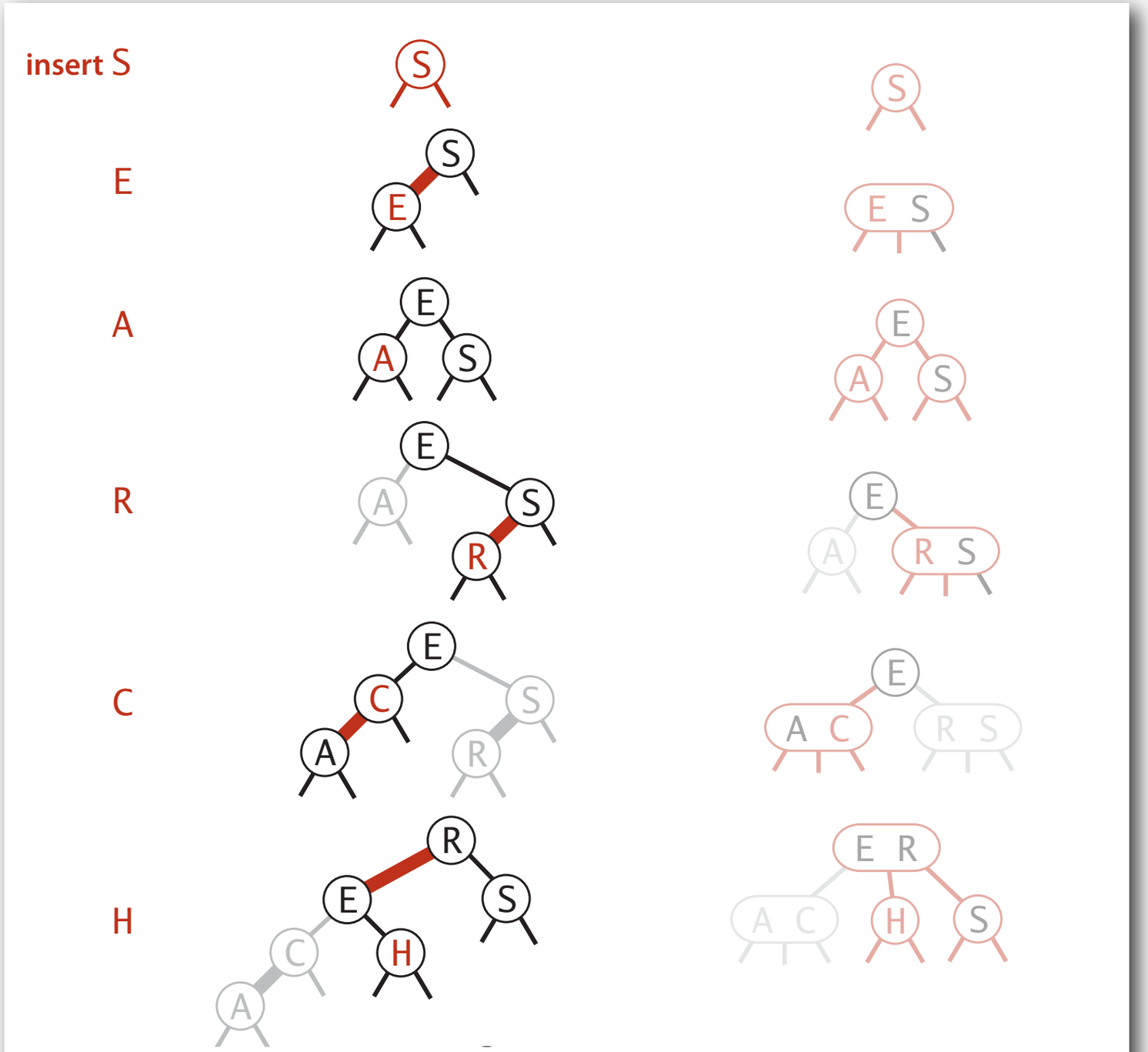
LLRB tree construction trace

Standard indexing client.



LLRB tree construction trace

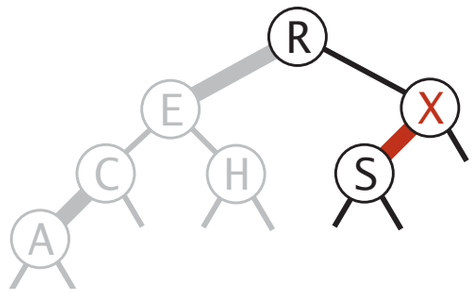
Standard indexing client.

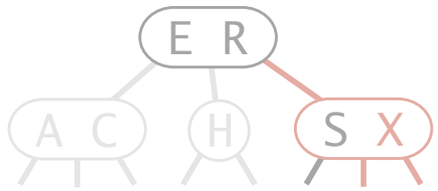


LLRB tree construction trace

Standard indexing client (continued).

X



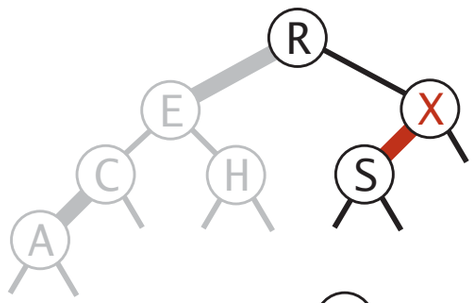


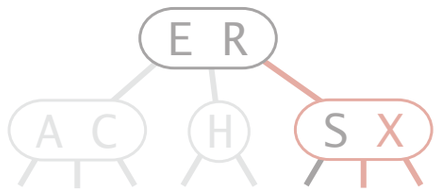
red black tree **corresponding 2-3 tree**

LLRB tree construction trace

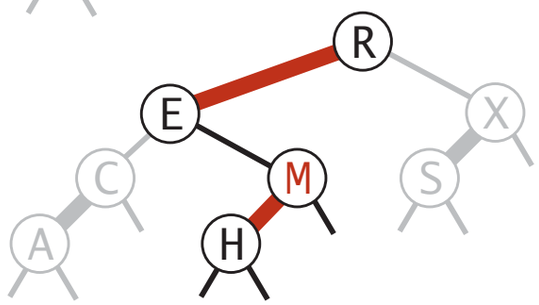
Standard indexing client (continued).

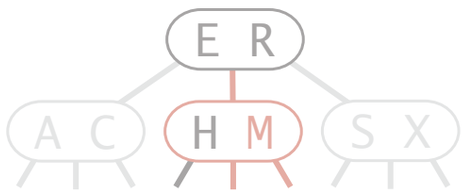
X





M



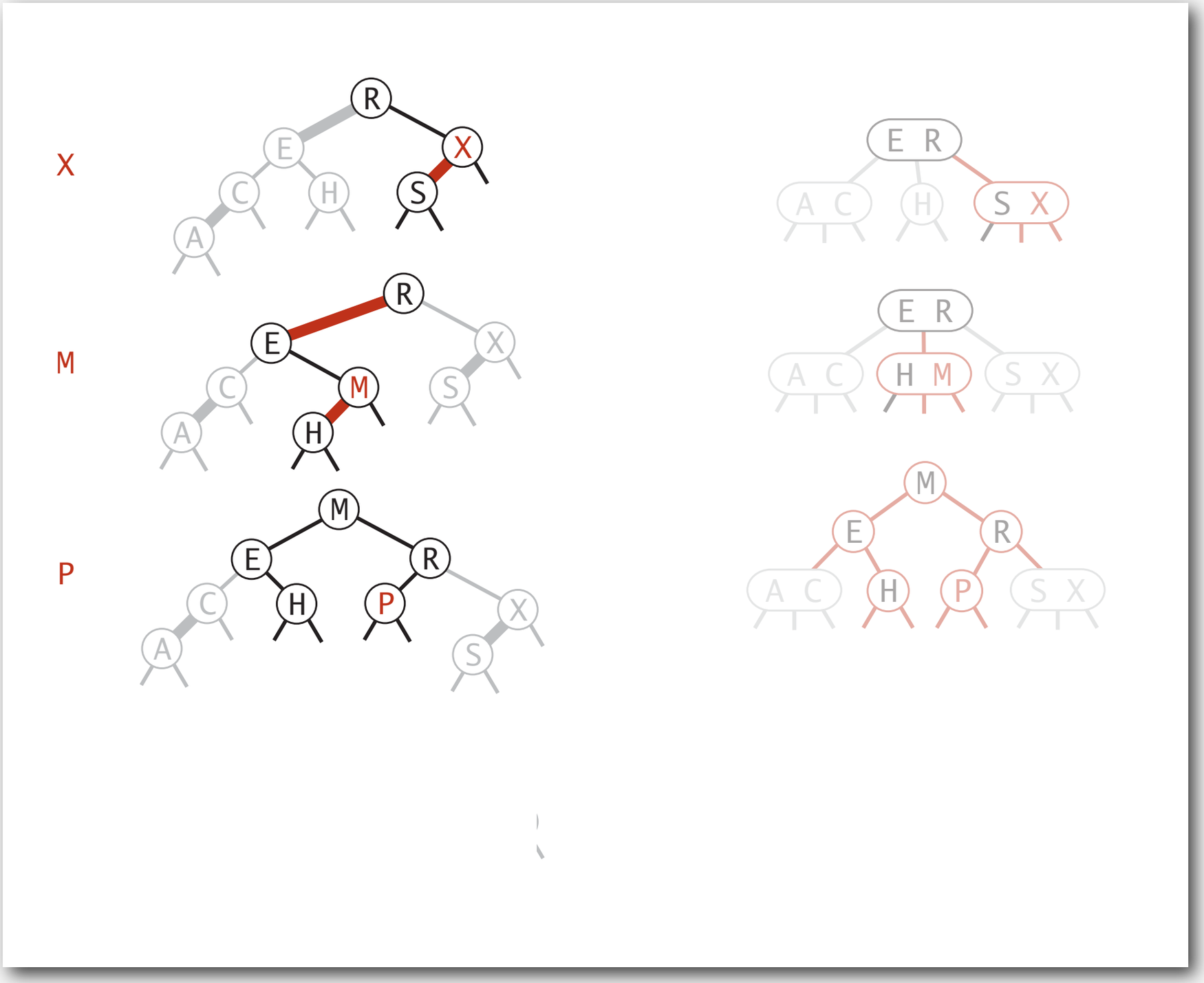


red black tree

corresponding 2-3 tree

LLRB tree construction trace

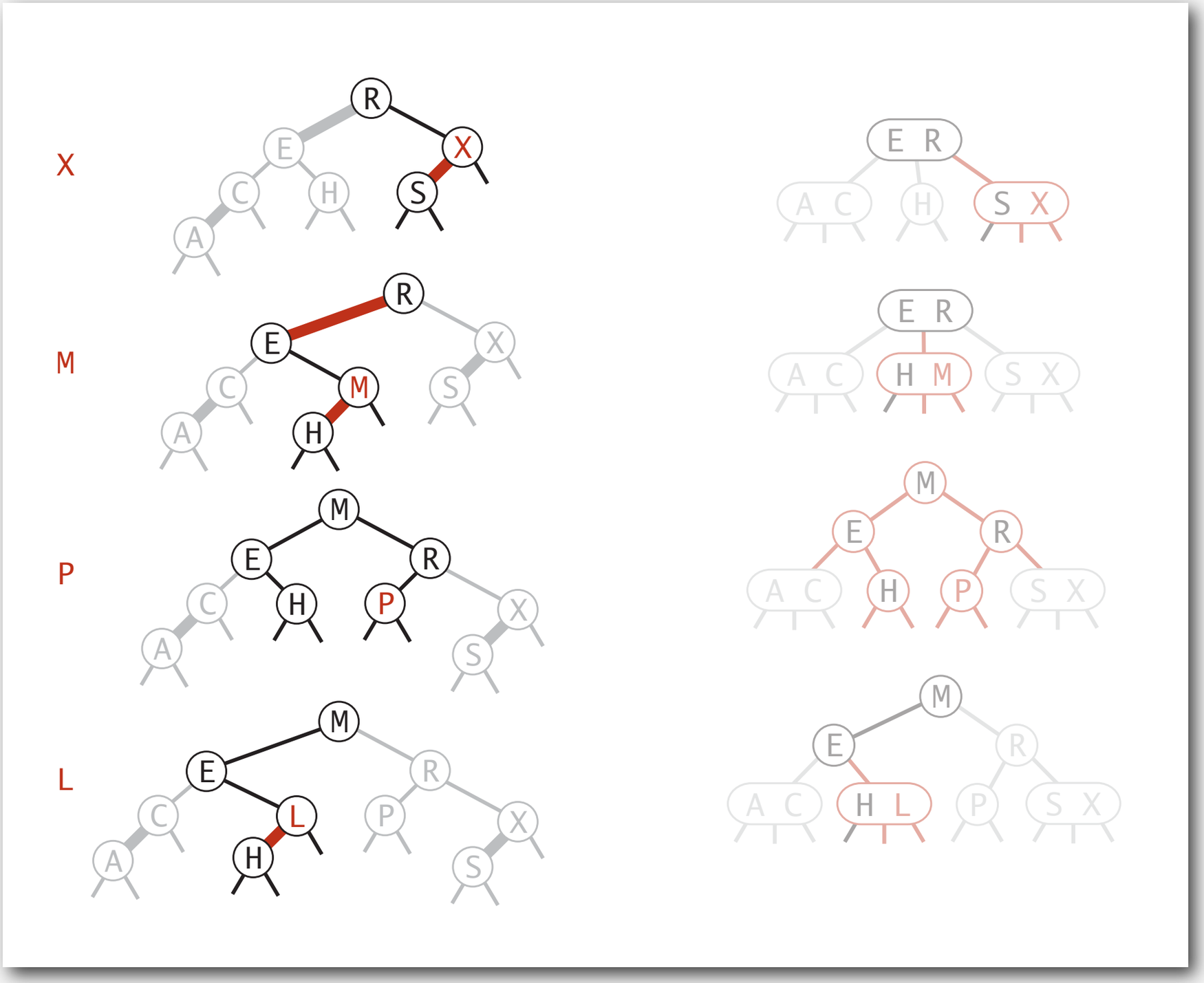
Standard indexing client (continued).



red black tree **corresponding 2-3 tree**

LLRB tree construction trace

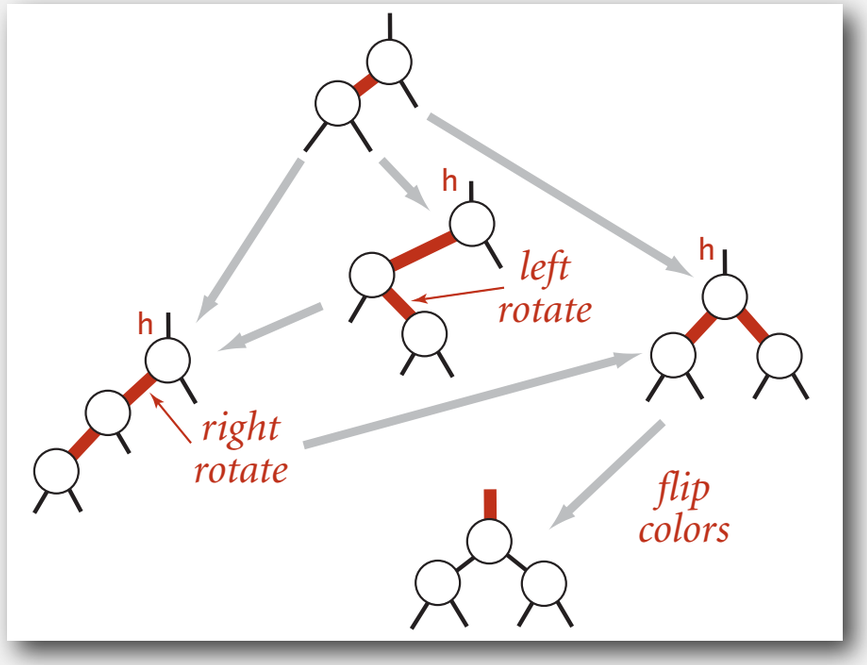
Standard indexing client (continued).



Insertion in a LLRB tree: Java implementation

Same code for both 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 red)

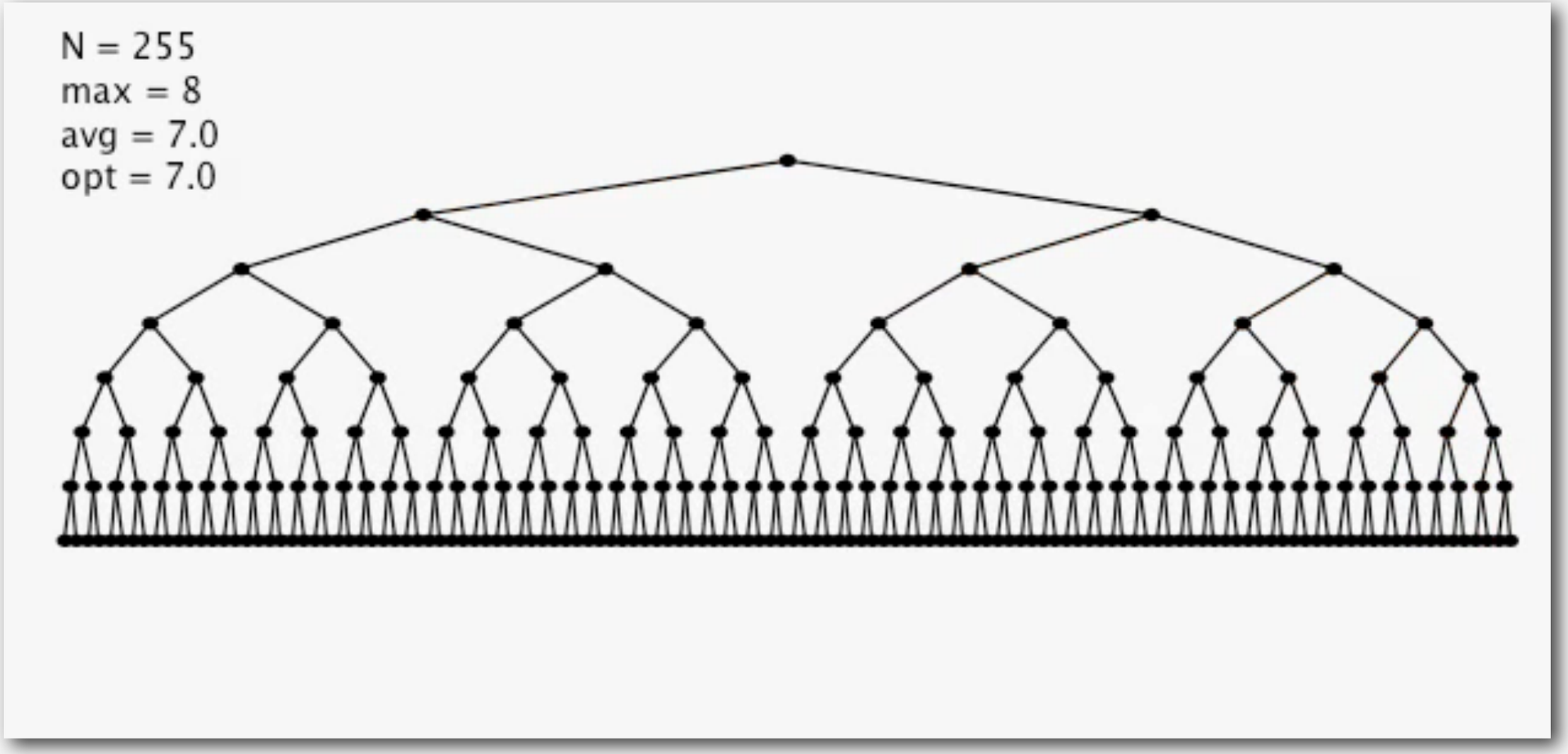
lean left

balance 4-node

split 4-node

only a few extra lines of code to provide near-perfect balance

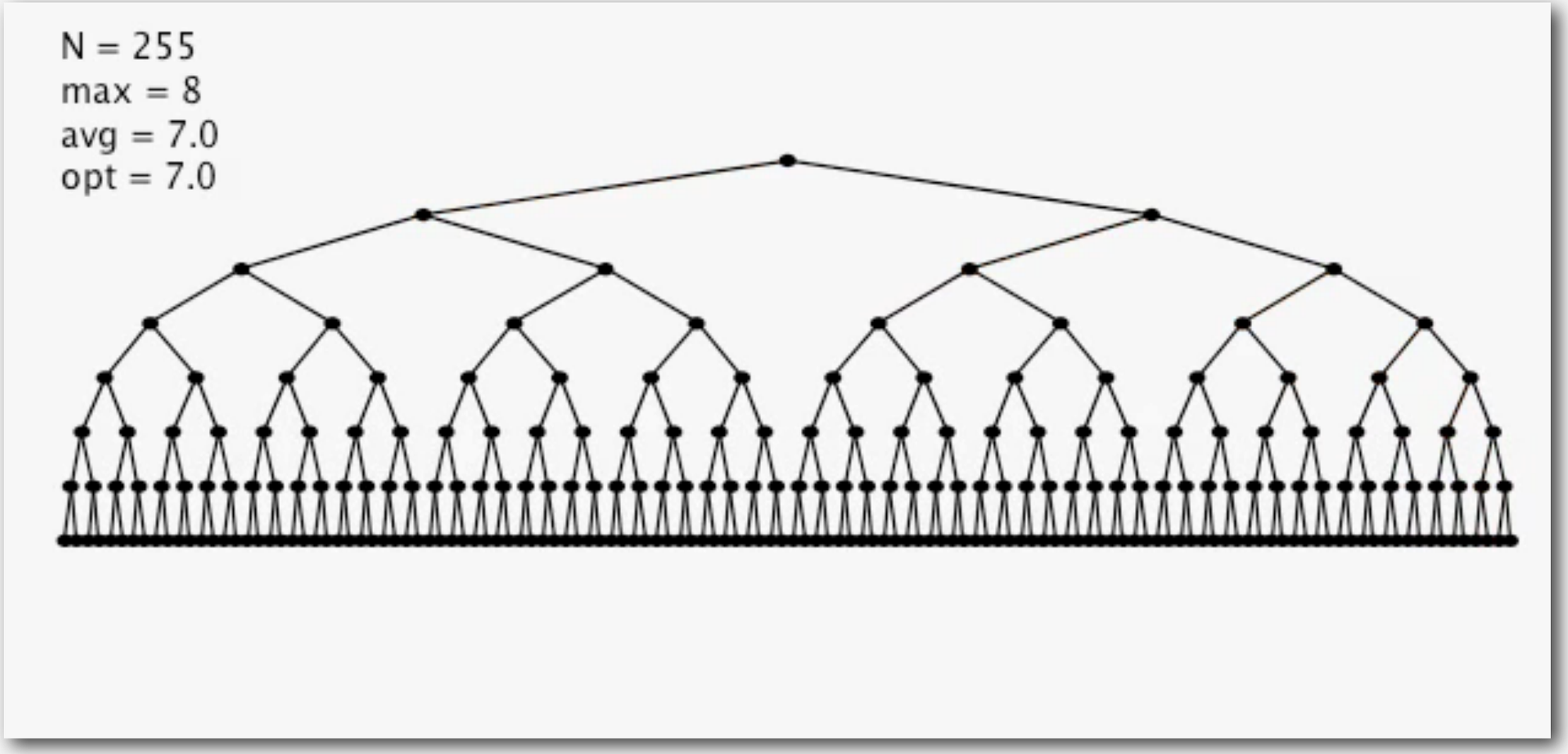
Insertion in a LLRB tree: visualization



255 insertions in ascending order

https://www.cs.purdue.edu/homes/cs251/slides/media/redblack-255ascending.mov

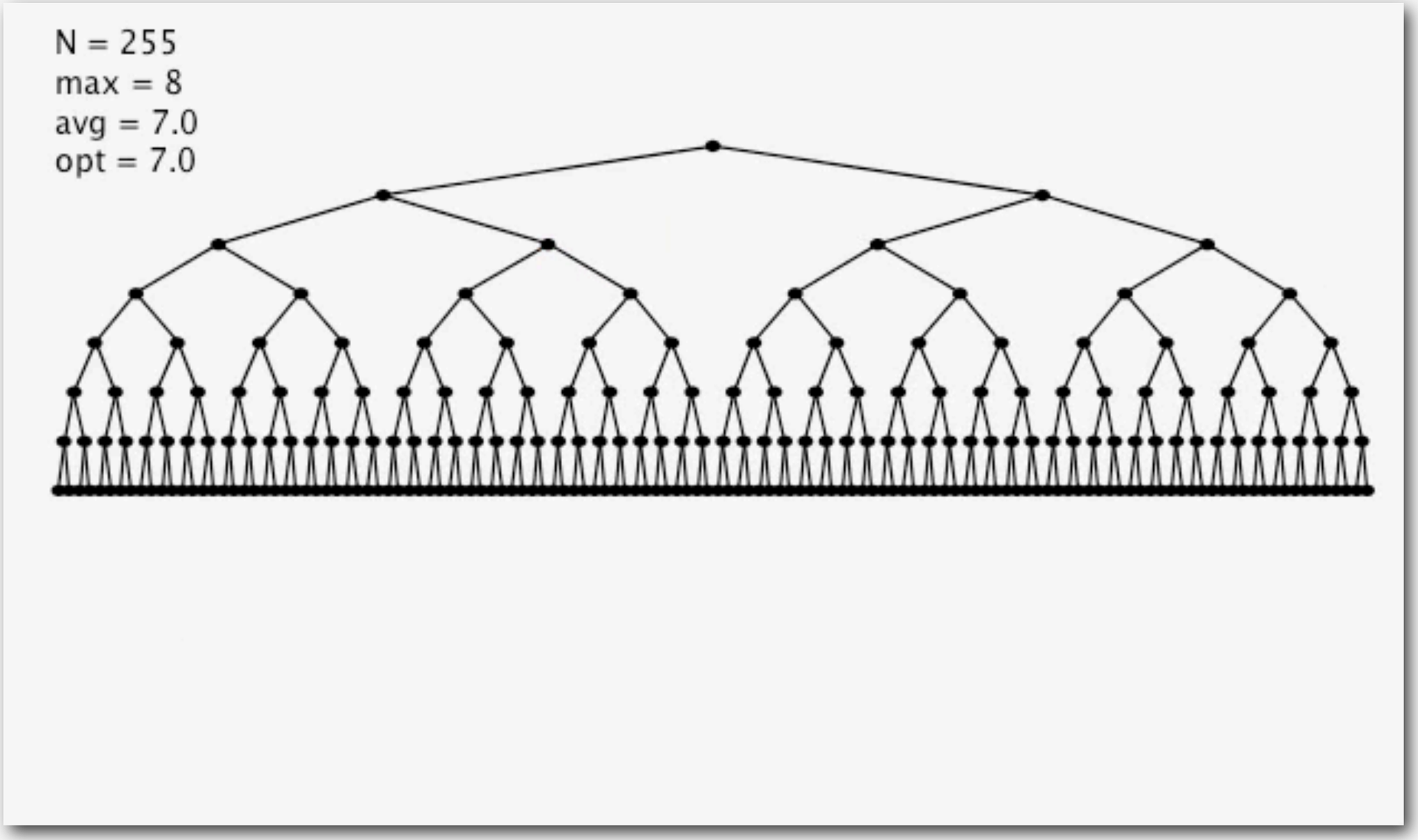
Insertion in a LLRB tree: visualization



255 insertions in ascending order

https://www.cs.purdue.edu/homes/cs251/slides/media/redblack-255ascending.mov

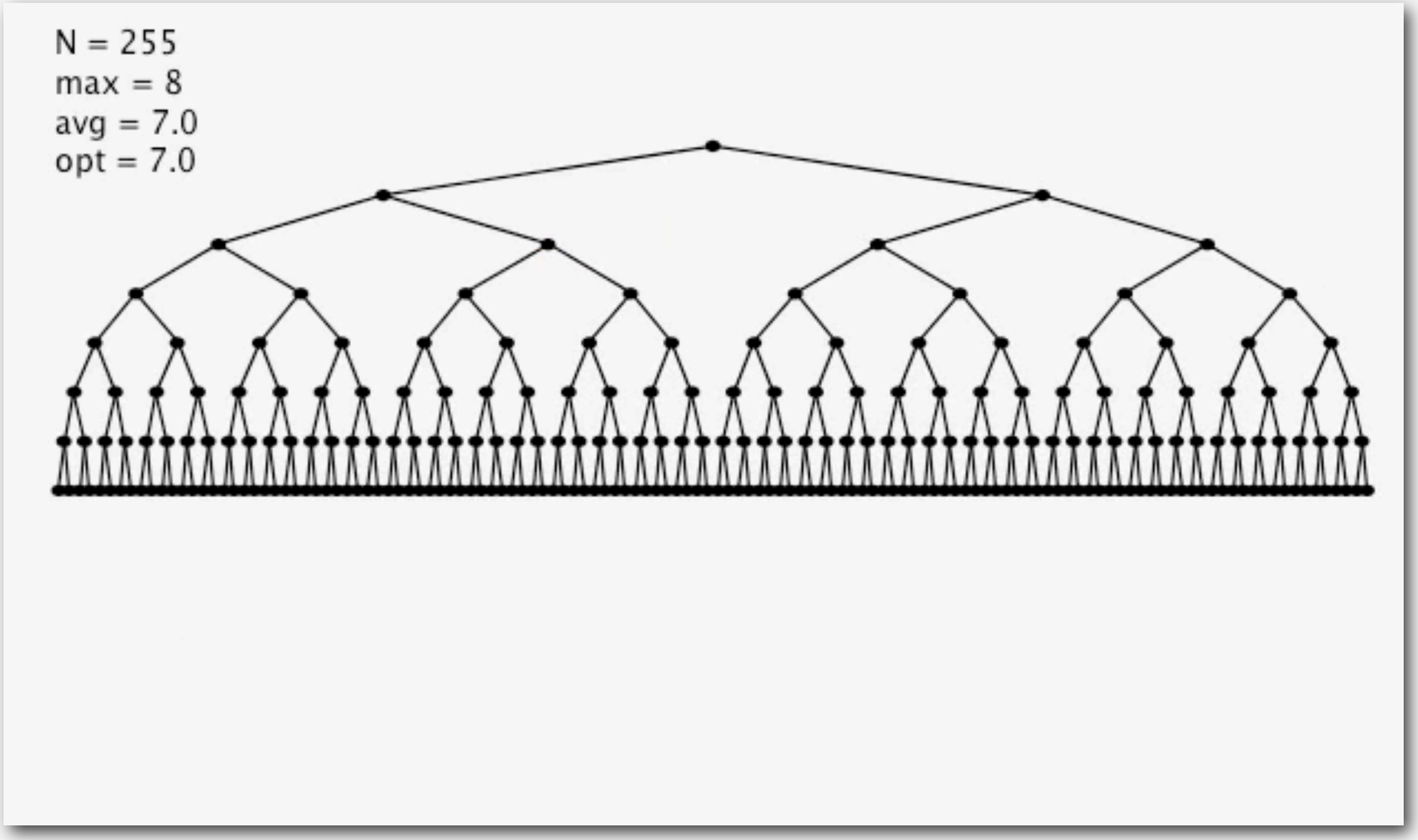
Insertion in a LLRB tree: visualization



255 insertions in descending order

https://www.cs.purdue.edu/homes/cs251/slides/media/redblack-255descending.mov

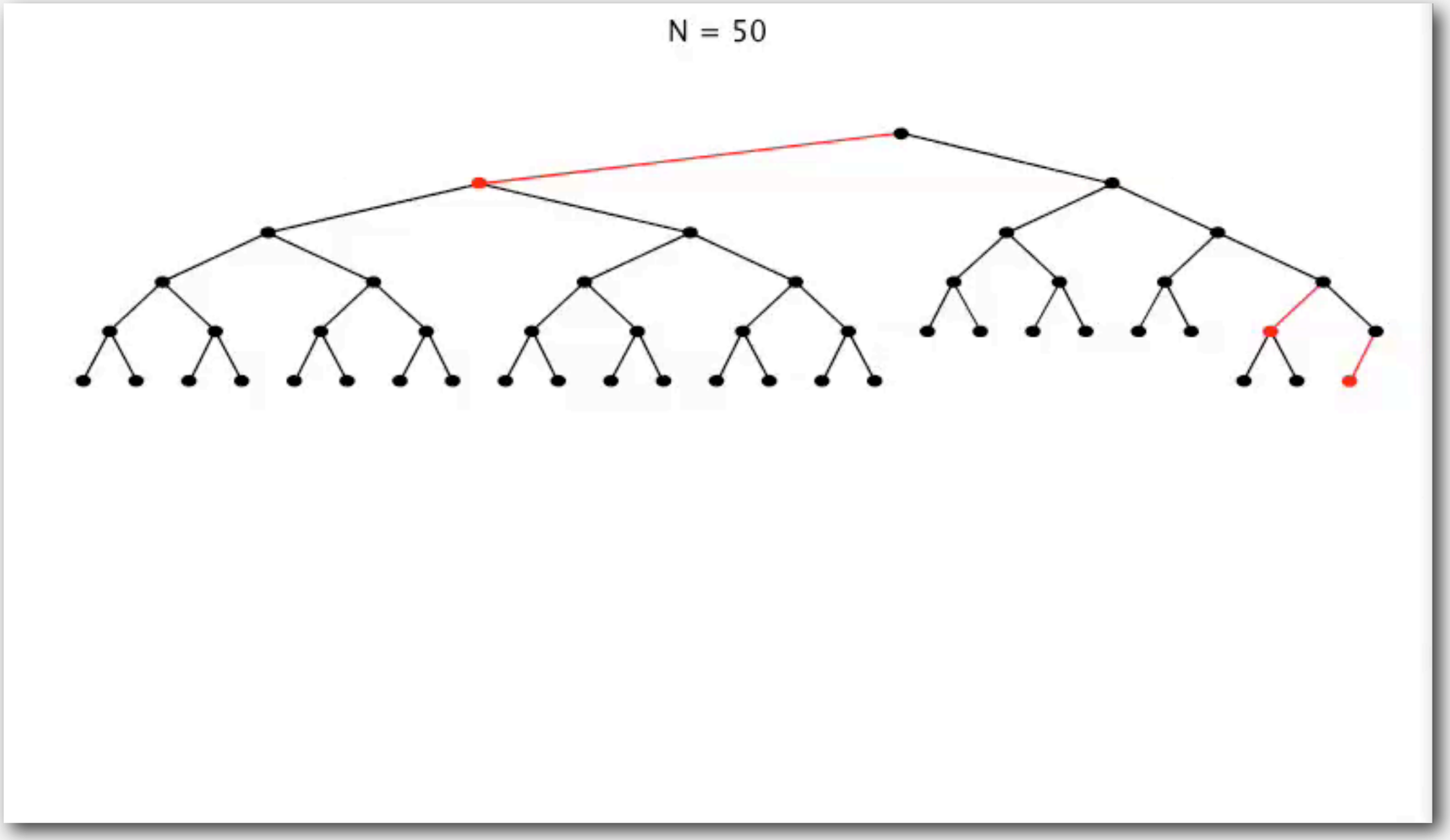
Insertion in a LLRB tree: visualization



255 insertions in descending order

https://www.cs.purdue.edu/homes/cs251/slides/media/redblack-255descending.mov

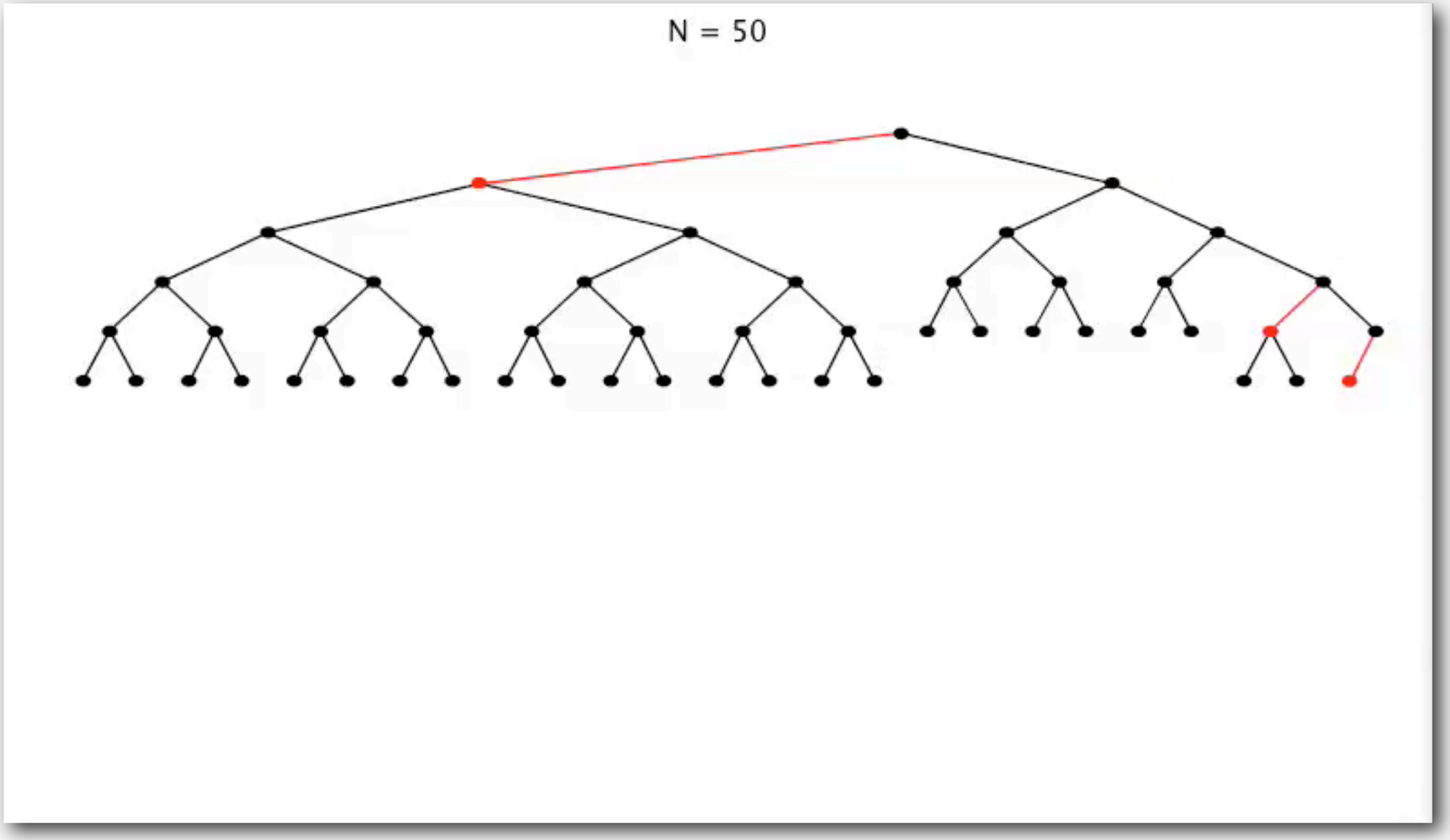
Insertion in a LLRB tree: visualization



50 random insertions

https://www.cs.purdue.edu/homes/cs251/slides/media/redblack50-random.mov

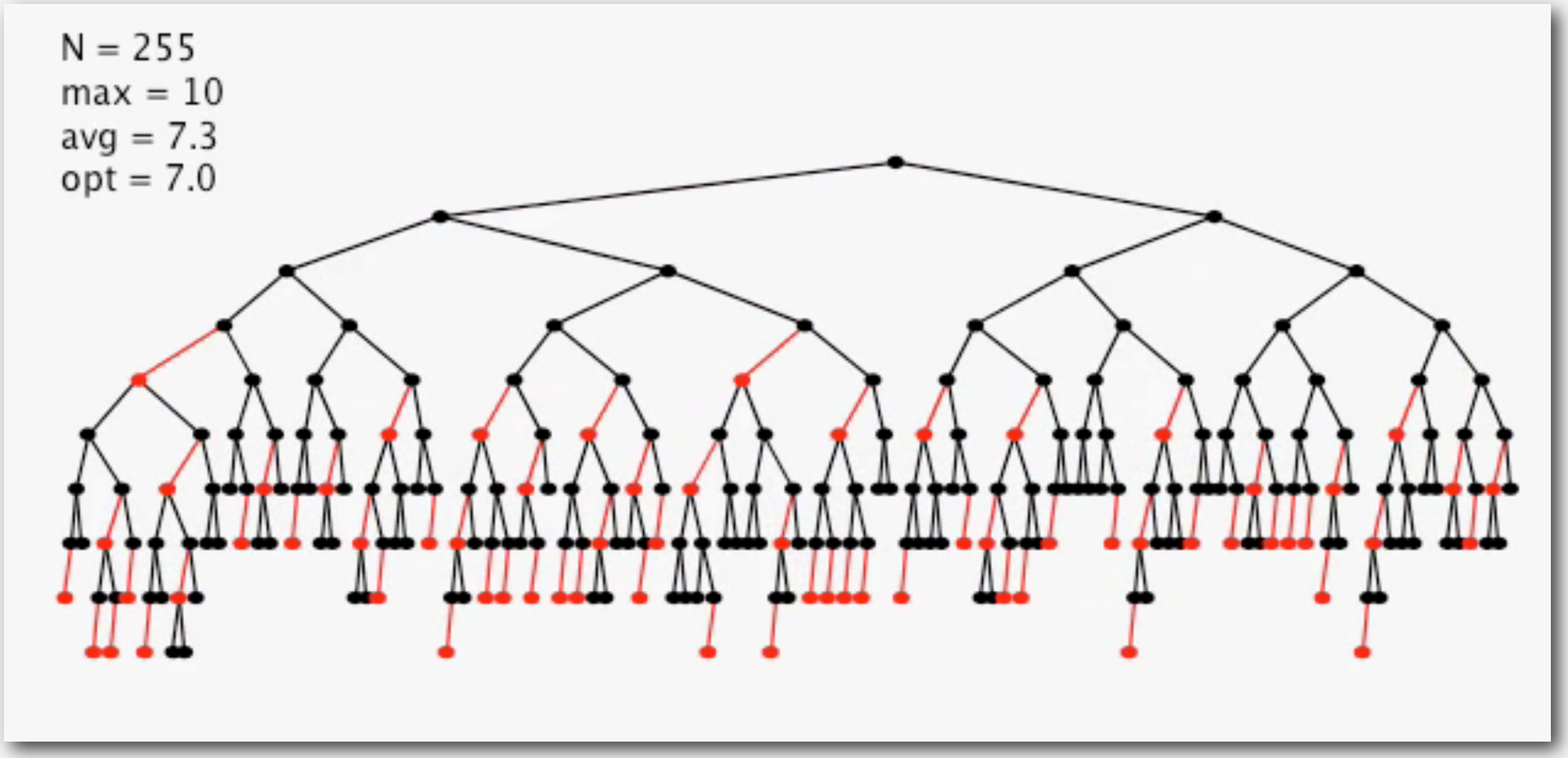
Insertion in a LLRB tree: visualization



50 random insertions

https://www.cs.purdue.edu/homes/cs251/slides/media/redblack50-random.mov

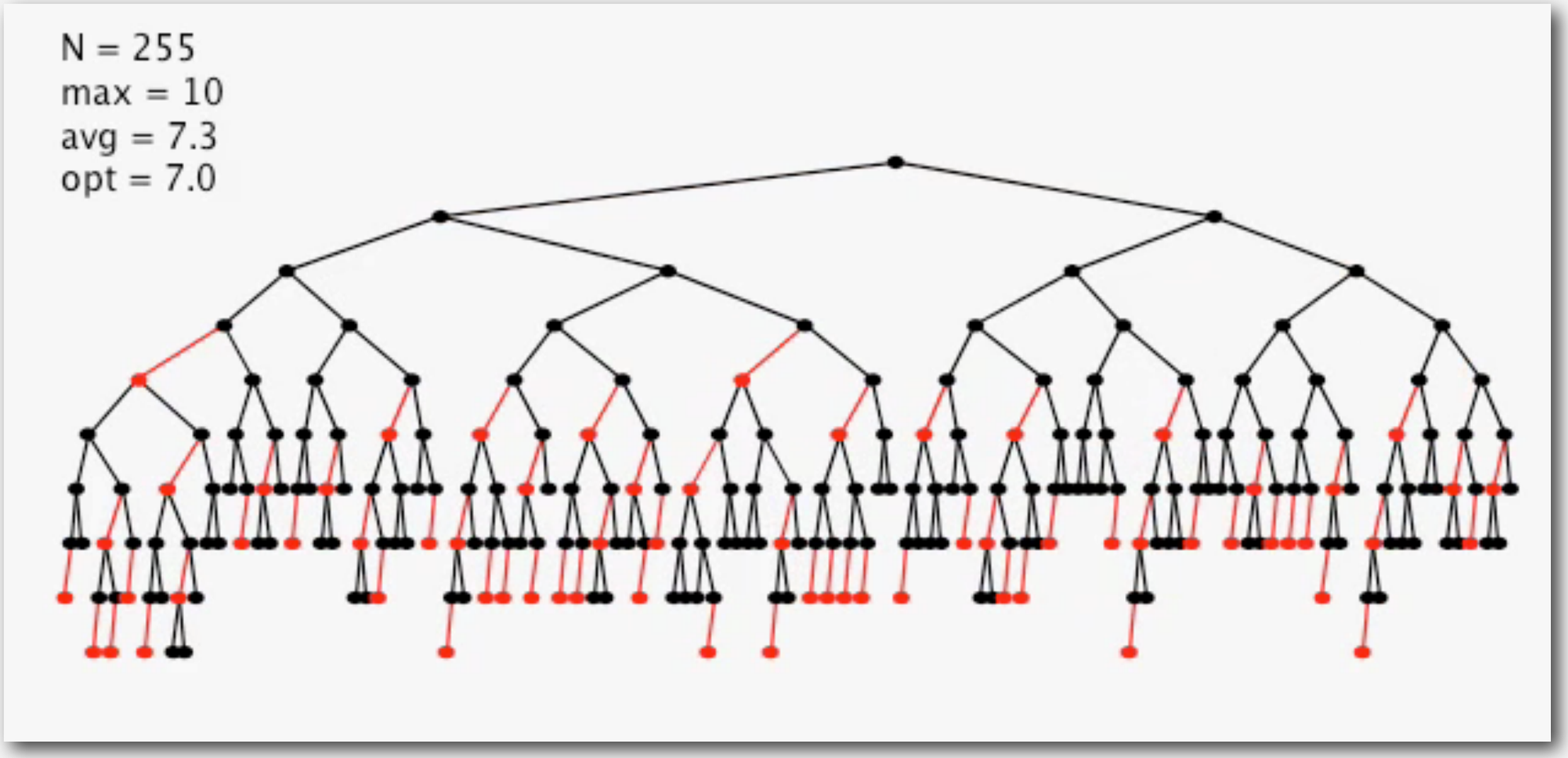
Insertion in a LLRB tree: visualization



255 random insertions

https://www.cs.purdue.edu/homes/cs251/slides/media/redblack-255random.mov

Insertion in a LLRB tree: visualization



255 random insertions

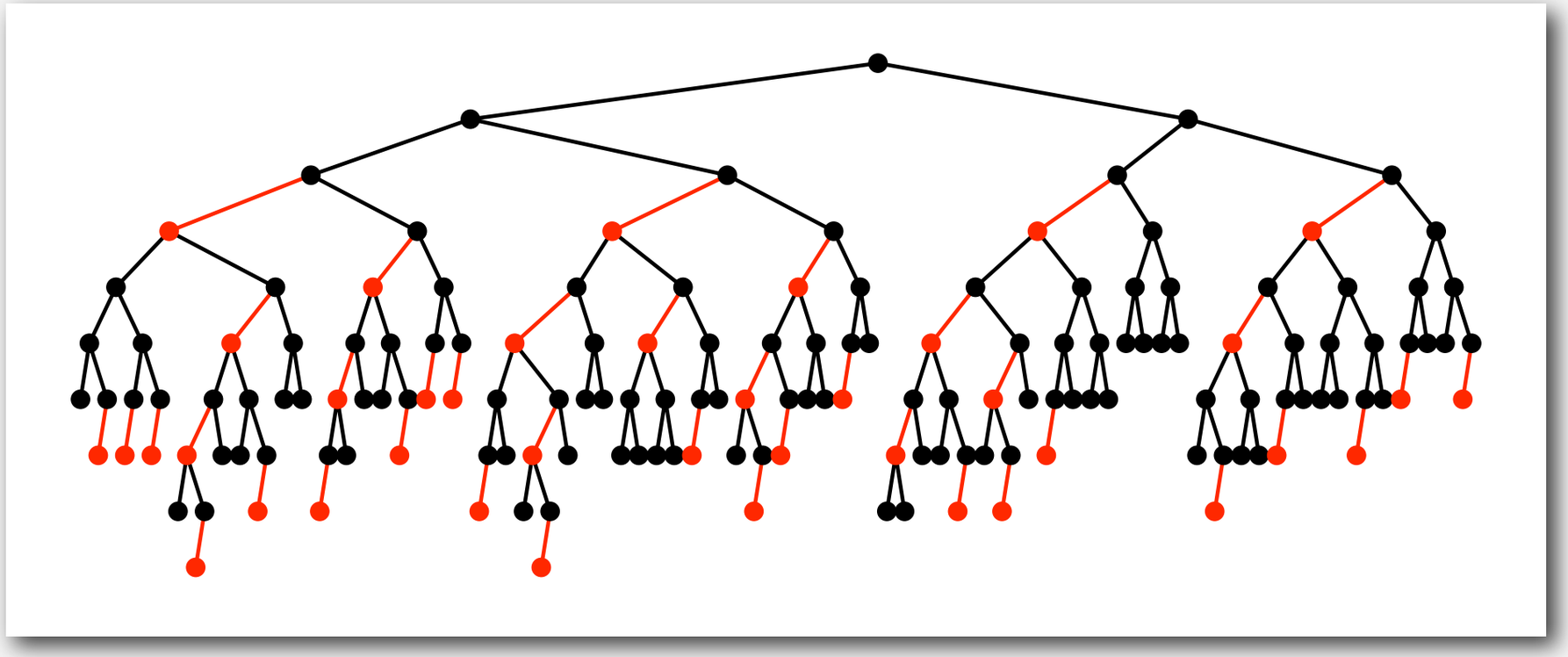
https://www.cs.purdue.edu/homes/cs251/slides/media/redblack-255random.mov

Balance in LLRB trees

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

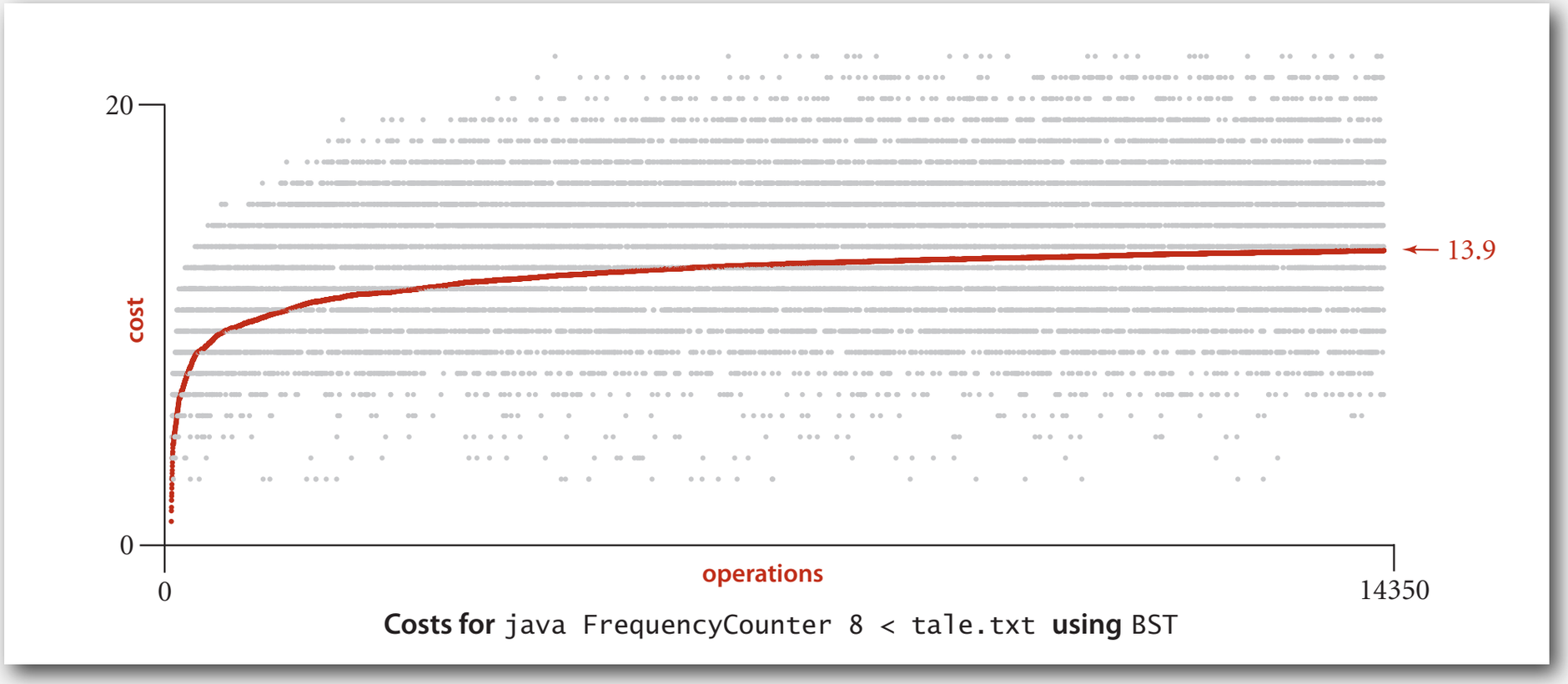
Pf.

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

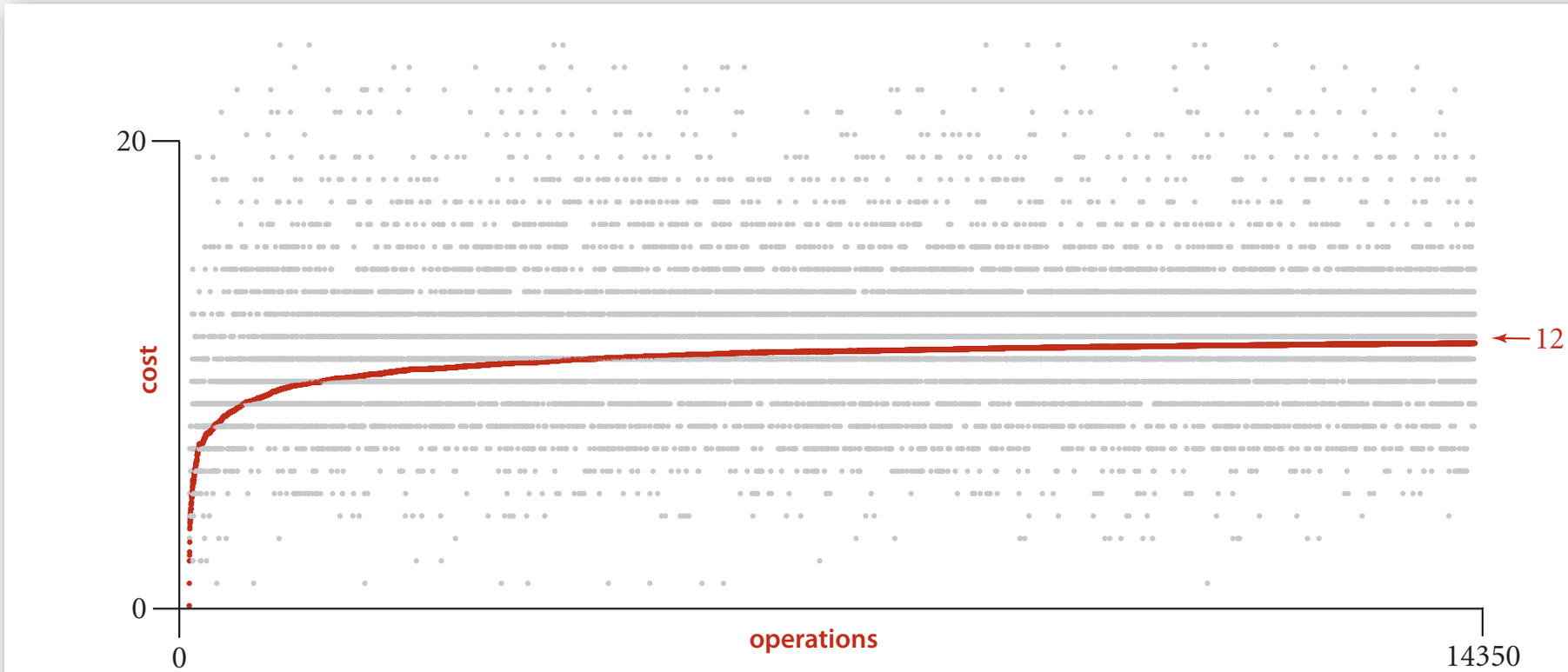


Property. Height of tree is $\sim 1.00 \lg N$ in typical applications.

ST implementations: frequency counter



Costs for `java FrequencyCounter 8 < tale.txt` **using** `BST`



Costs for `java FrequencyCounter 8 < tale.txt` **using** `RedBlackBST`

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	`compareTo()`

* exact value of coefficient unknown but extremely close to 1

Why left-leaning trees?

old code (that students had to learn in the past)

```
private Node put(Node x, Key key, Value val, boolean sw)
{
   if (x == null)
      return new Node(key, value, RED);
   int cmp = key.compareTo(x.key);

   if (isRed(x.left) && isRed(x.right))
   {
      x.color = RED;
      x.left.color  = BLACK;
      x.right.color = BLACK;
   }
   if (cmp < 0)
   {
      x.left = put(x.left, key, val, false);
      if (isRed(x) && isRed(x.left) && sw)
         x = rotateRight(x);
      if (isRed(x.left) && isRed(x.left.left))
      {
         x = rotateRight(x);
         x.color = BLACK; x.right.color = RED;
      }
   }
   else if (cmp > 0)
   {
      x.right = put(x.right, key, val, true);
      if (isRed(h) && isRed(x.right) && !sw)
         x = rotateLeft(x);
      if (isRed(h.right) && isRed(h.right.right))
      {
         x = rotateLeft(x);
         x.color = BLACK; x.left.color = RED;
      }
   }
   else x.val = val;
   return x;
}
```

extremely tricky

new code (that you have to learn)

```
public 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;
}
```

straightforward

(if you've paid attention)

Why left-leaning red-black BSTs?

Simplified code.

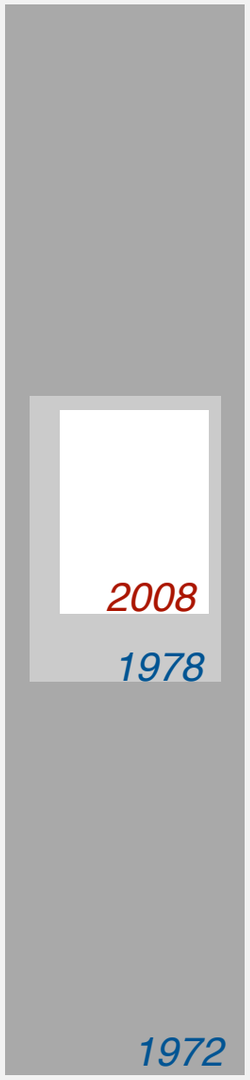
- Left-leaning restriction reduces number of cases.
- Short inner loop.

Same ideas simplify implementation of other operations.

- Delete min/max.
- Arbitrary delete.

Improves widely-used balanced search trees.

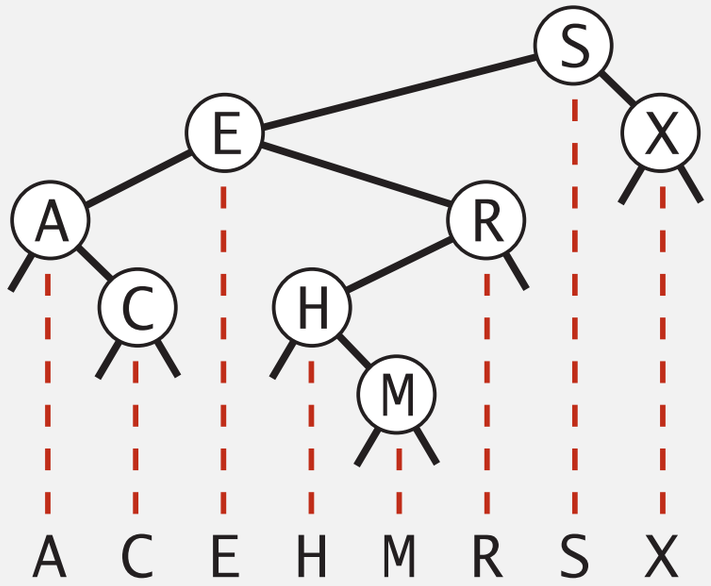
- AVL trees, splay trees, randomized BSTs, ...
- 2-3 trees, 2-3-4 trees.
- Red-black BSTs.



Bottom line. Left-leaning red-black BSTs are among the simplest balanced BSTs to implement and among the fastest in practice.

Quiz

What is the height of this tree



A. 2

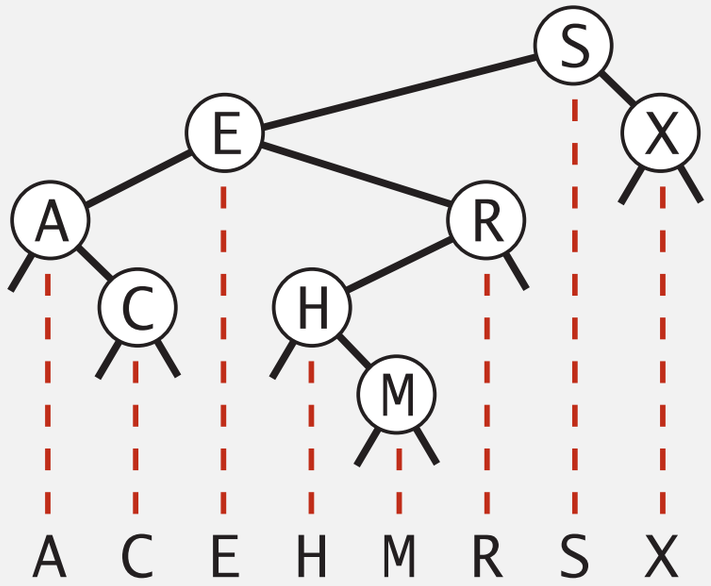
B. 3

C. 4

D. 5

Quiz

Which of the following input orders could have produced this BST



A. S, E, X, R, A, C, H, M

B. S, X, E, A, H, R, C, M

C. A & B

D. None

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

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

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:

> *" If implemented properly, the height of a red-black BST with N keys is at most 2 lg N. "* — *expert witness*