http://algs4.cs.princeton.edu

3.3 BALANCED SEARCH TREES

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

Last updated on 2/28/16 9:33 PM

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	N	N	N		`equals()`
binary search (ordered array)	$\log N$	N	N	$\log N$	N	N	✔	`compareTo()`
BST	N	N	N	$\log N$	$\log N$	$\sqrt{N}$	✔	`compareTo()`
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`

Challenge. Guarantee performance.

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

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

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

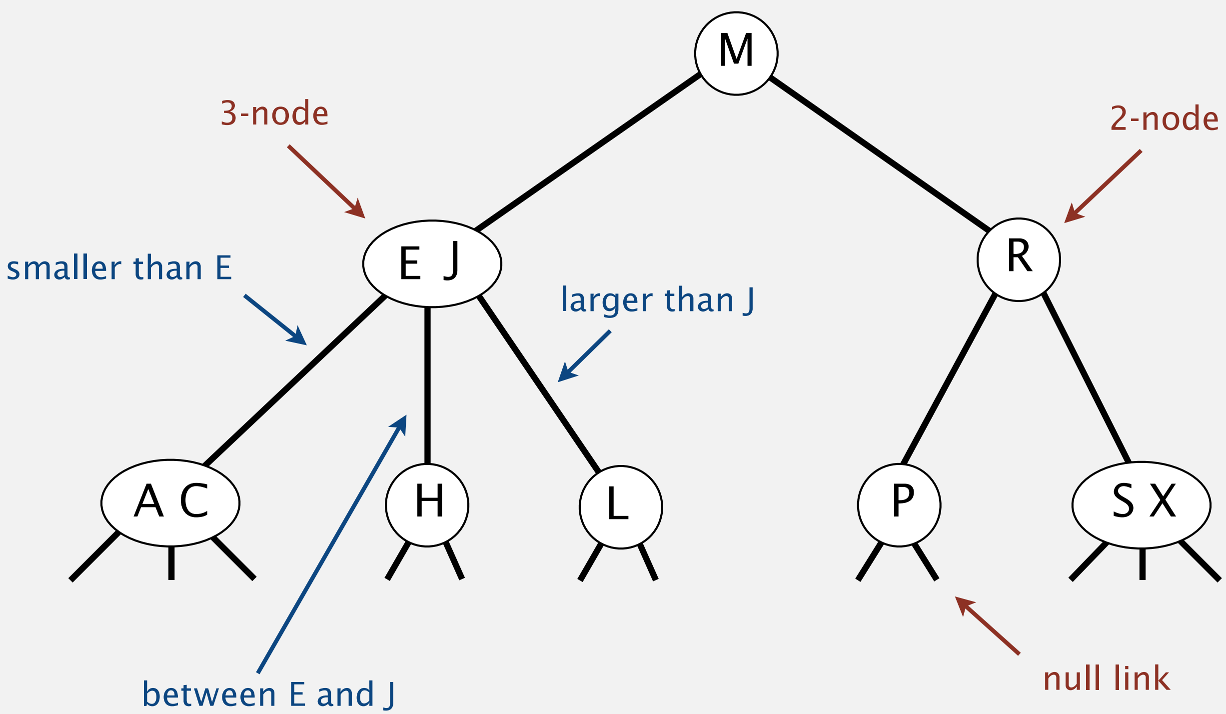
2–3 tree

Symmetric order. Inorder traversal yields keys in ascending order.

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

Allow 1 or 2 keys per node.

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

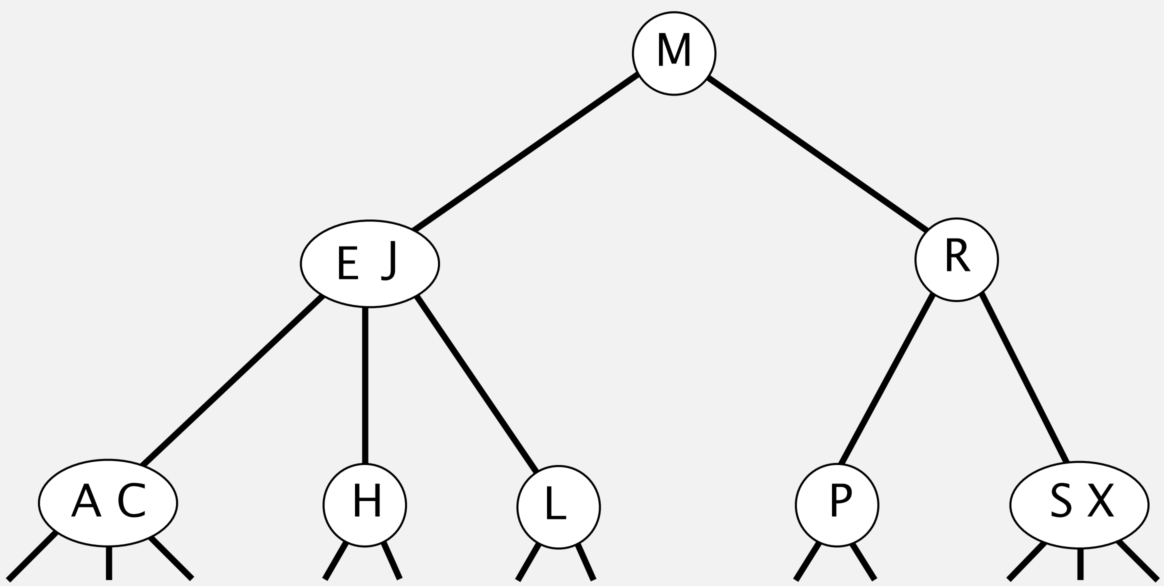


2-3 tree demo

Search.

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

search for H

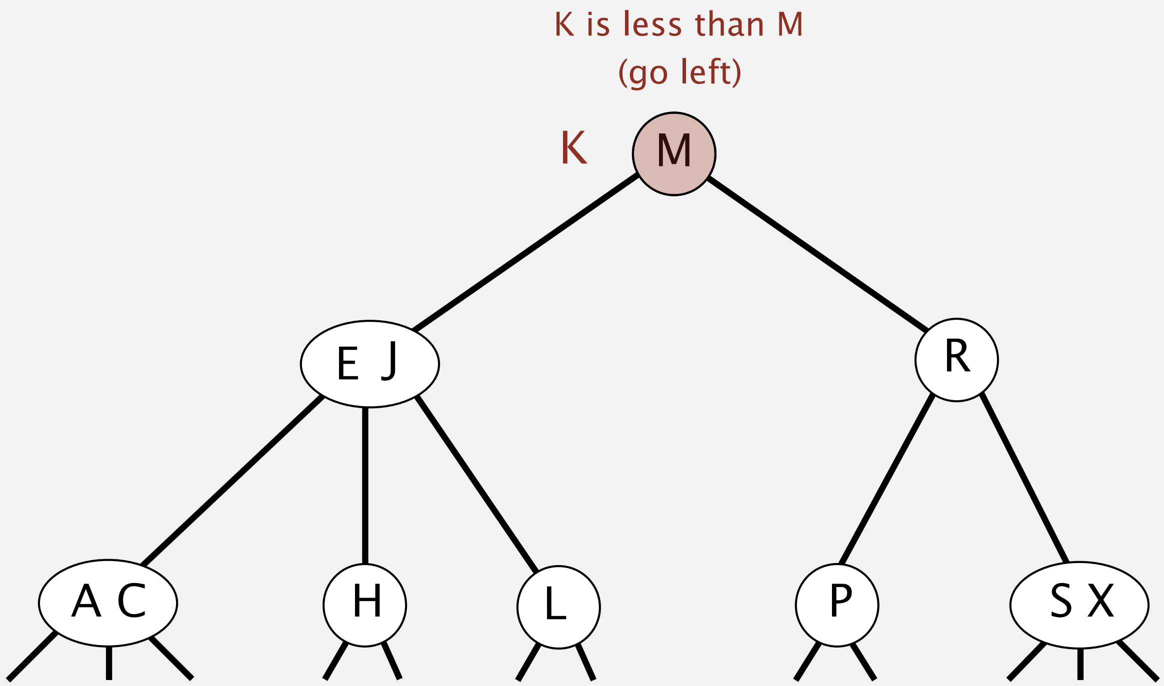


2-3 tree demo: insertion

Insert into a 2-node at bottom.

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

insert K

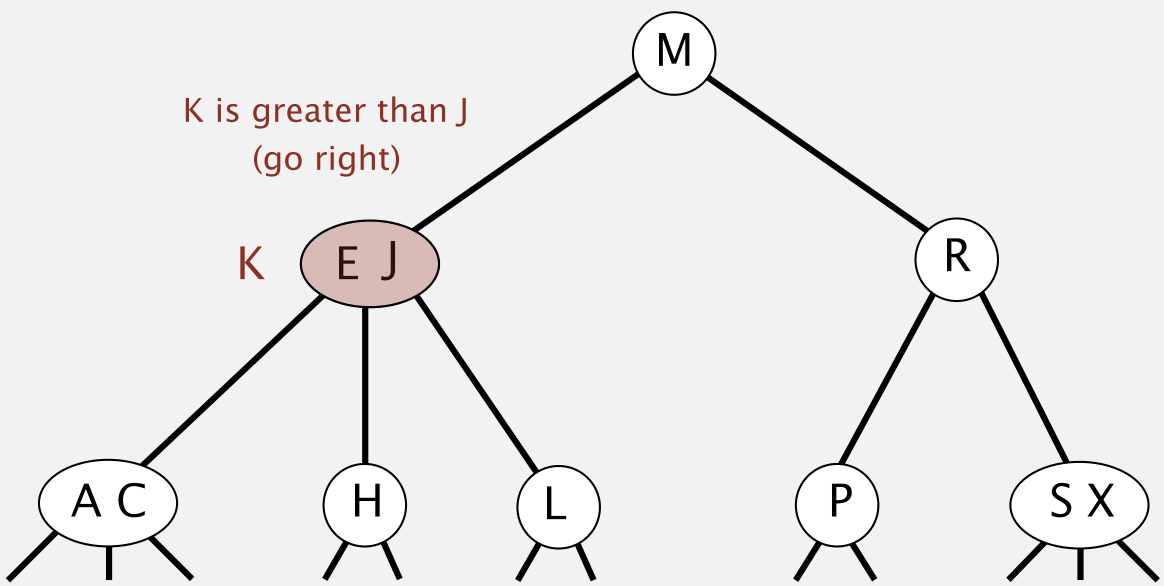


2–3 tree demo: insertion

Insert into a 2-node at bottom.

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

insert K

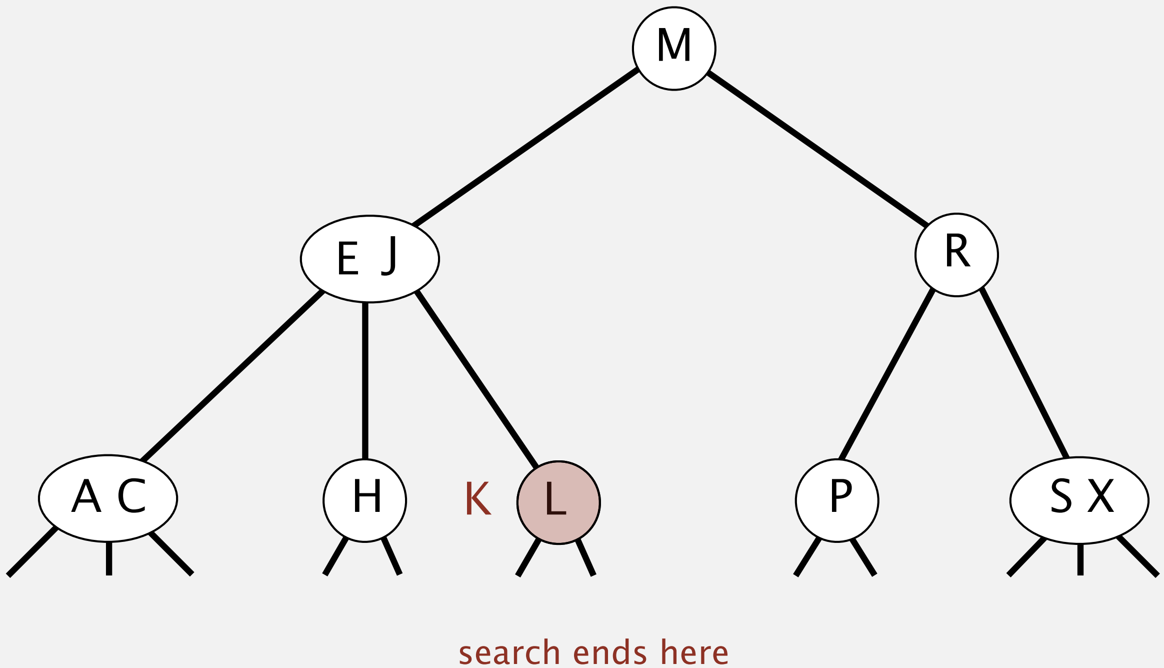


2-3 tree demo: insertion

Insert into a 2-node at bottom.

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

insert K

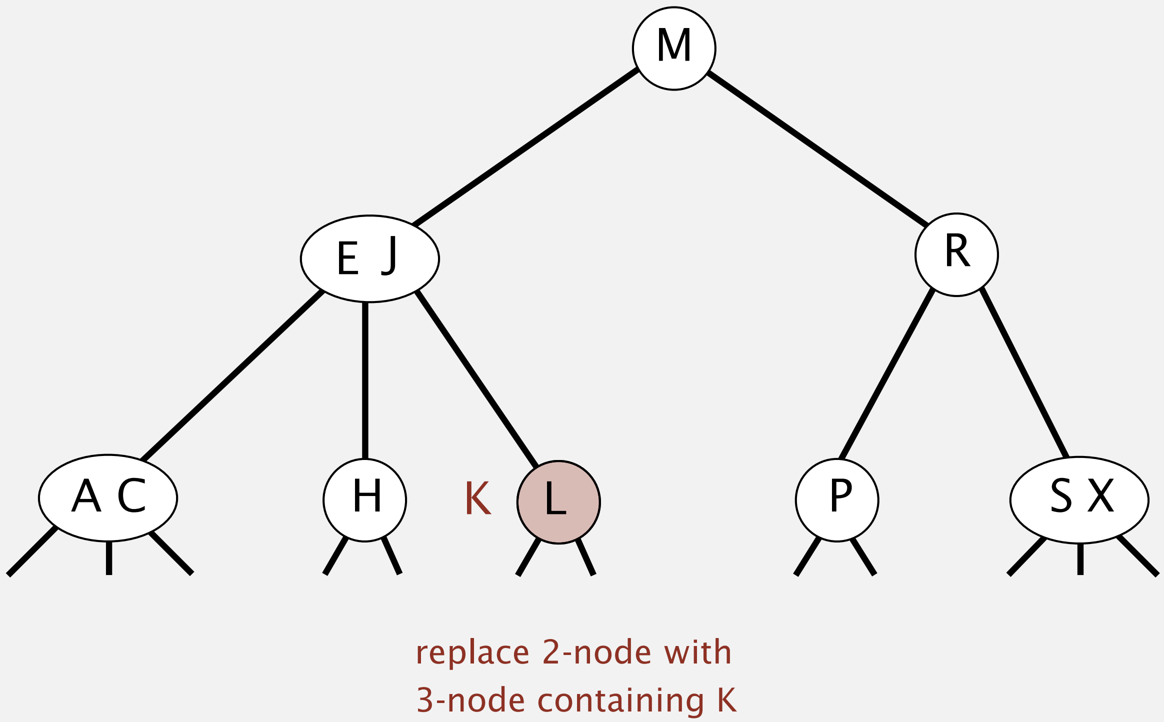


2–3 tree demo: insertion

Insert into a 2-node at bottom.

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

insert K

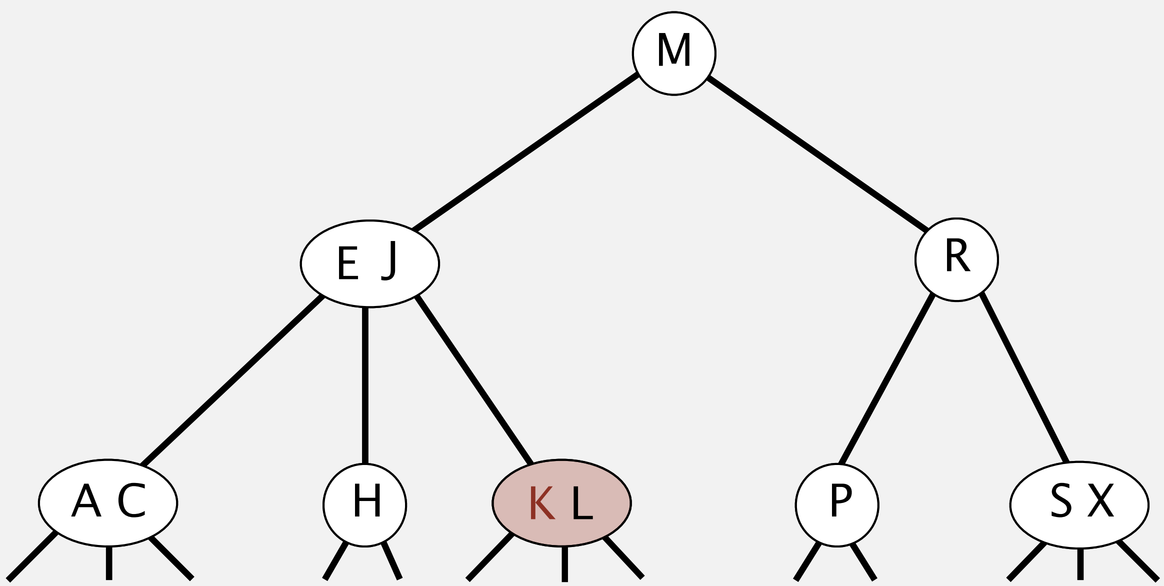


2–3 tree demo: insertion

Insert into a 2-node at bottom.

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

insert K

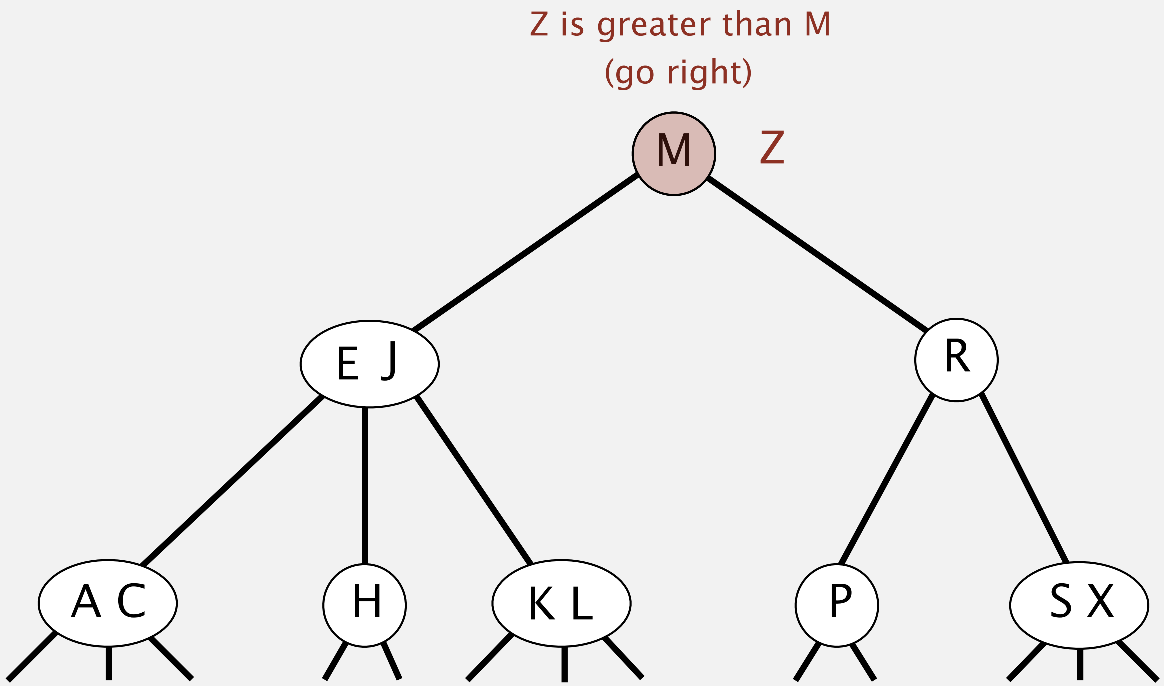


2-3 tree demo: insertion

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.

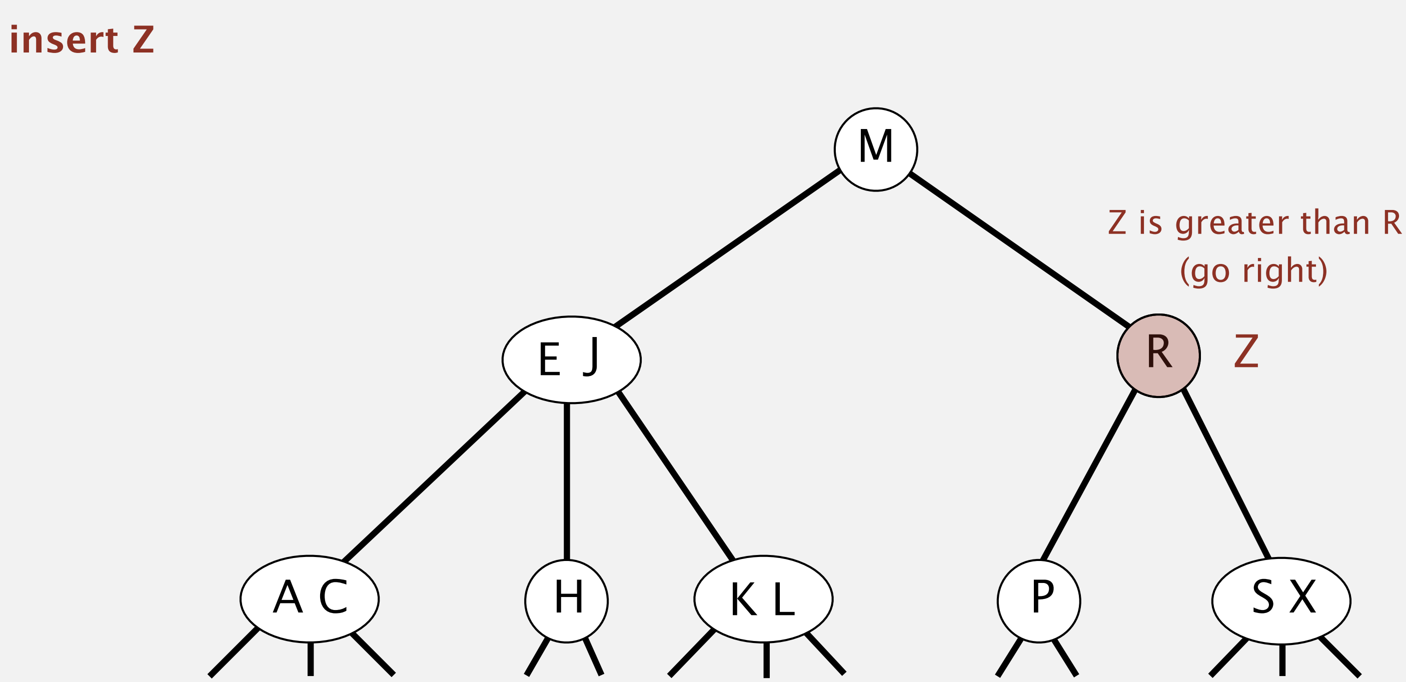
insert Z



2-3 tree demo: insertion

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.

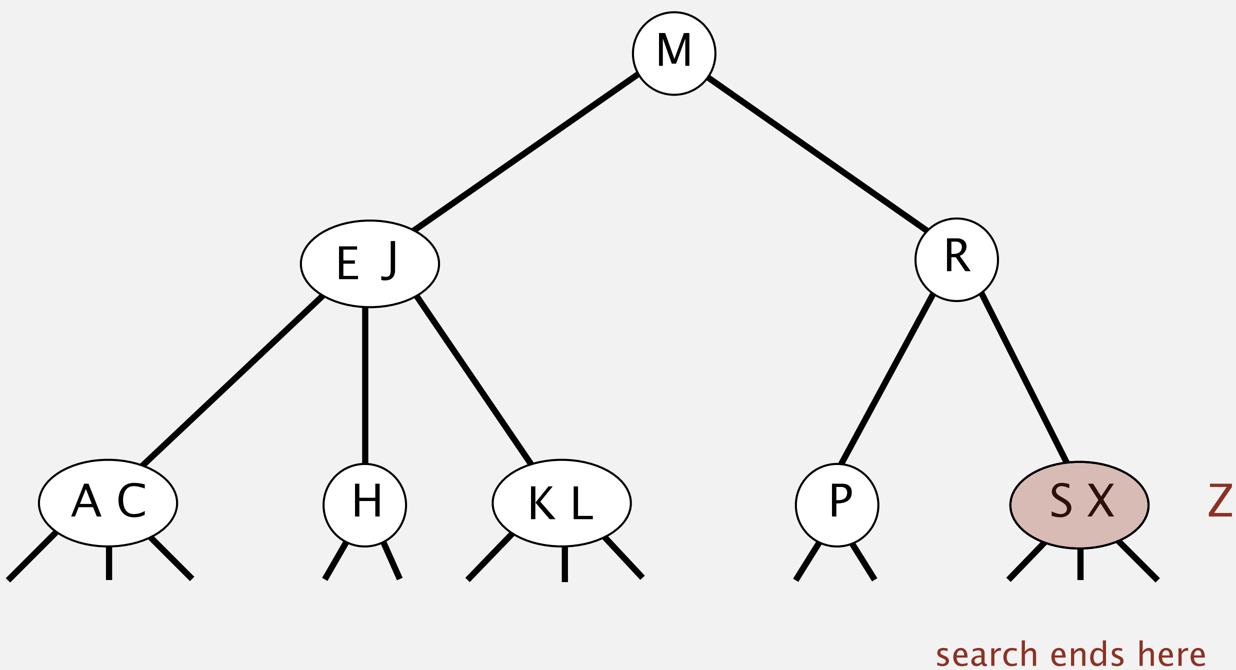


2-3 tree demo: insertion

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.

insert Z

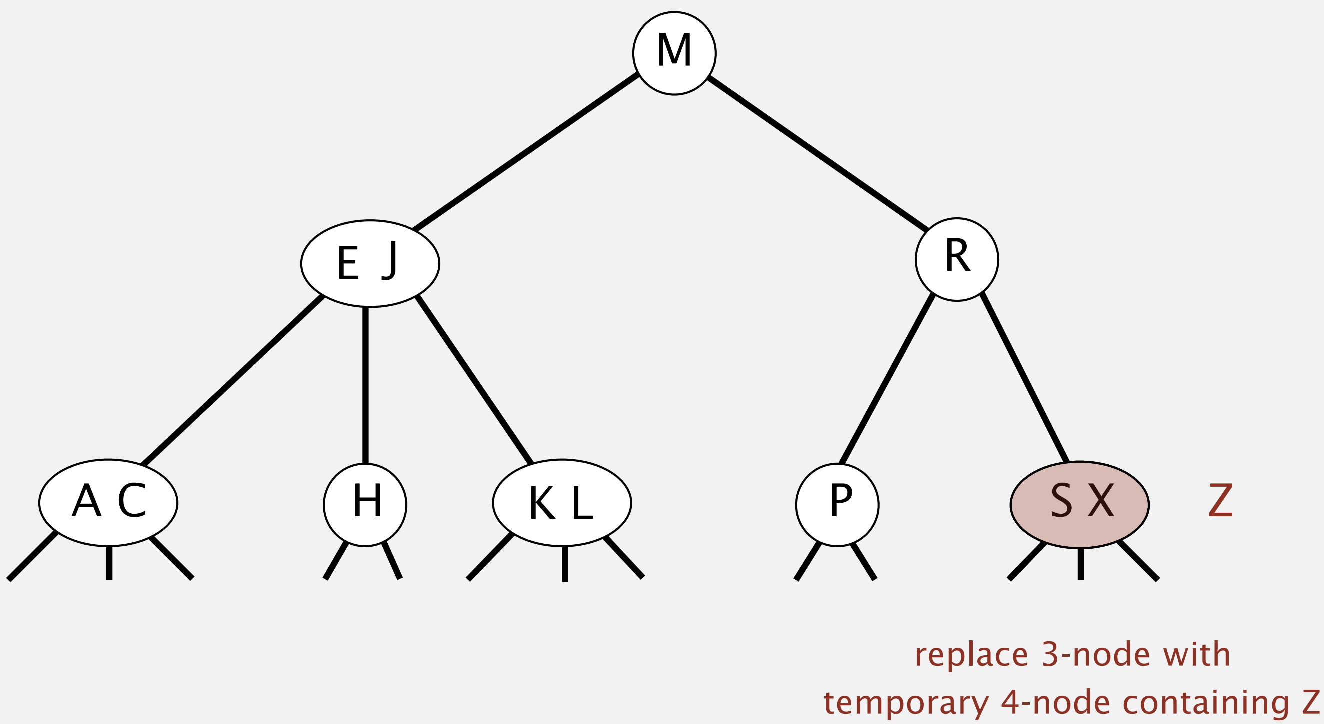


2-3 tree demo: insertion

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.

insert Z

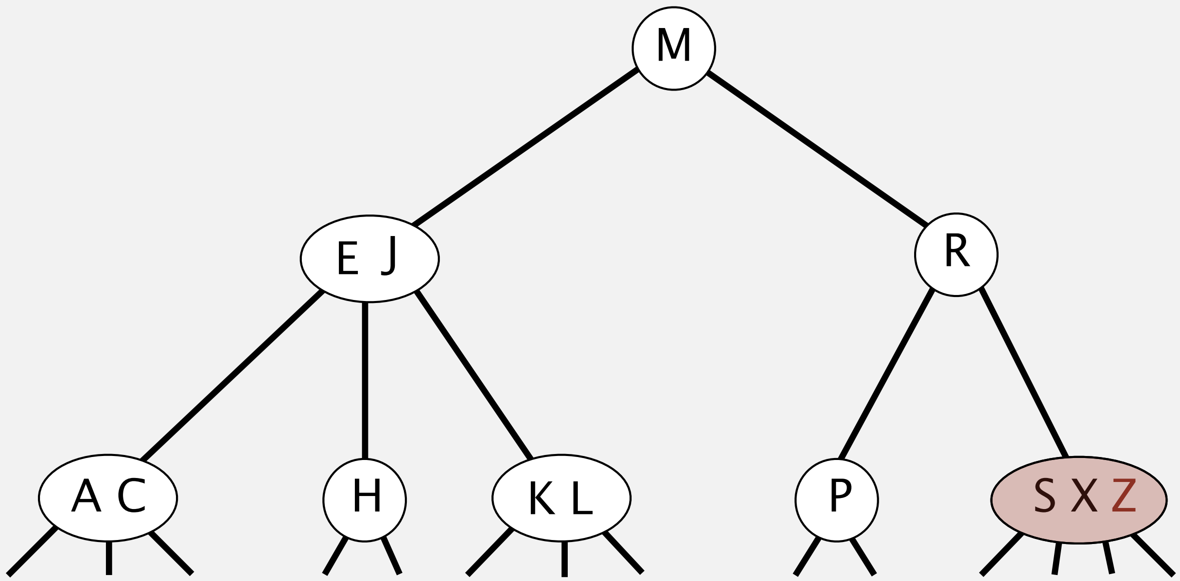


2–3 tree demo: insertion

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.

insert Z

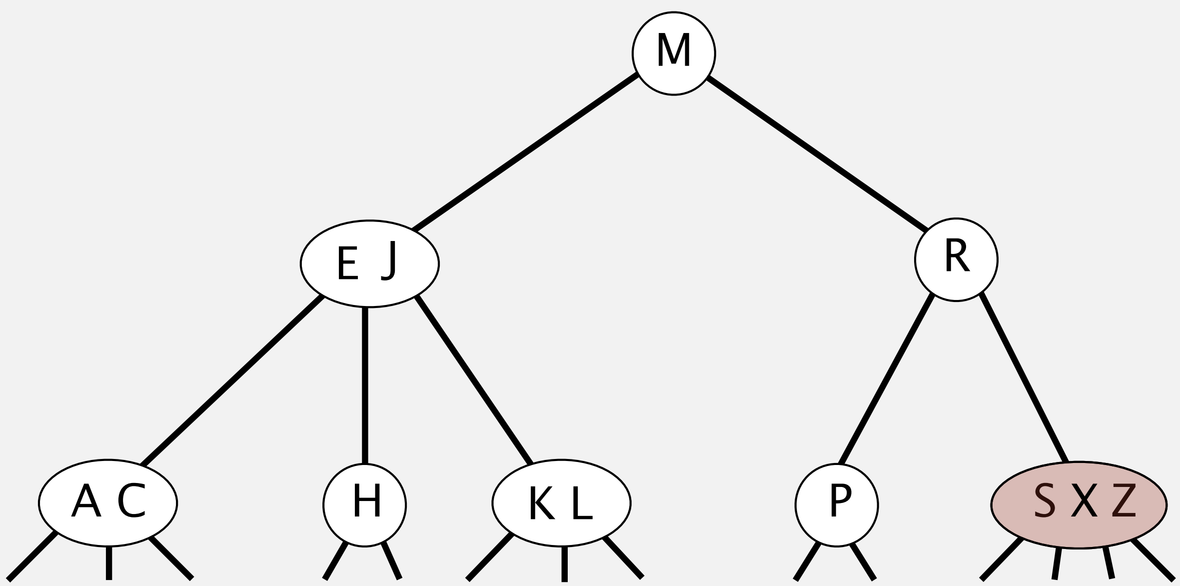


2–3 tree demo: insertion

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.

insert Z



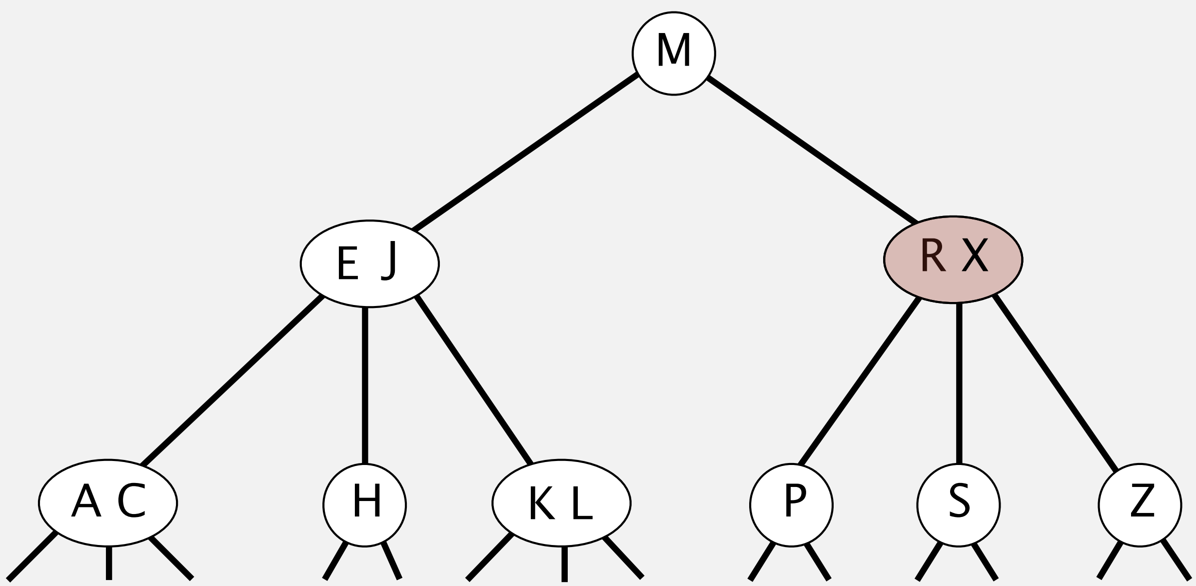
split 4-node into two 2-nodes
(pass middle key to parent)

2-3 tree demo: insertion

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.

insert Z

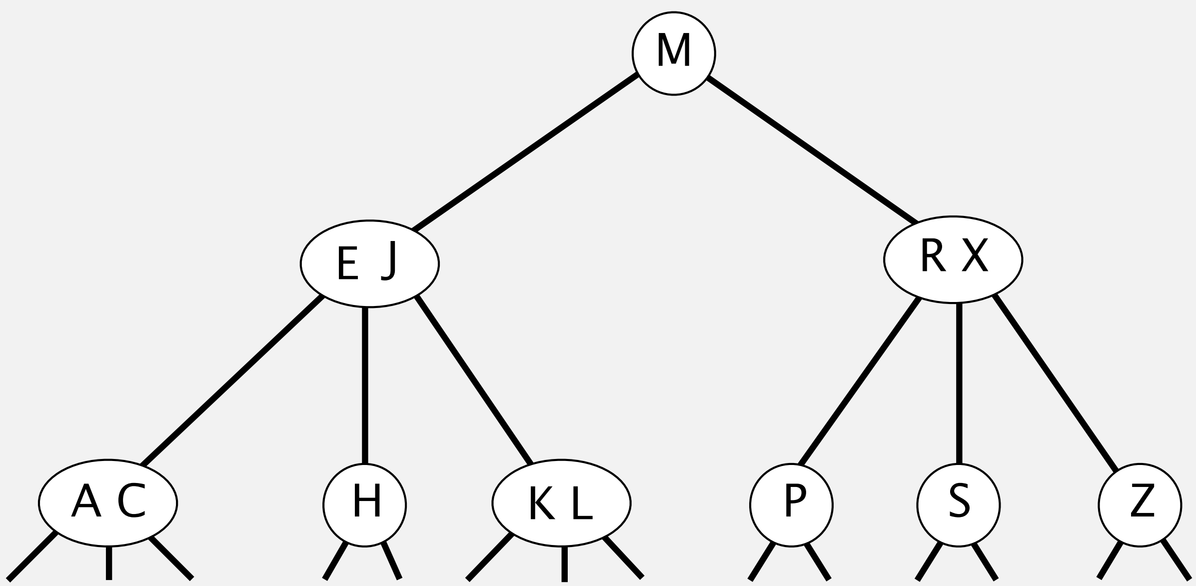


2-3 tree demo: insertion

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.

insert Z

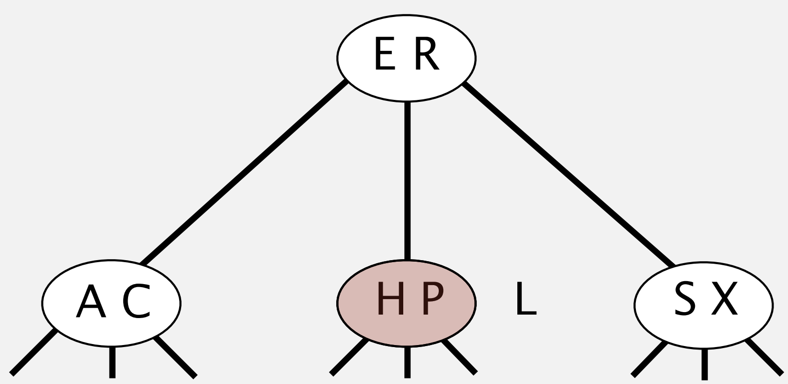


2-3 tree demo: insertion

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.

insert L



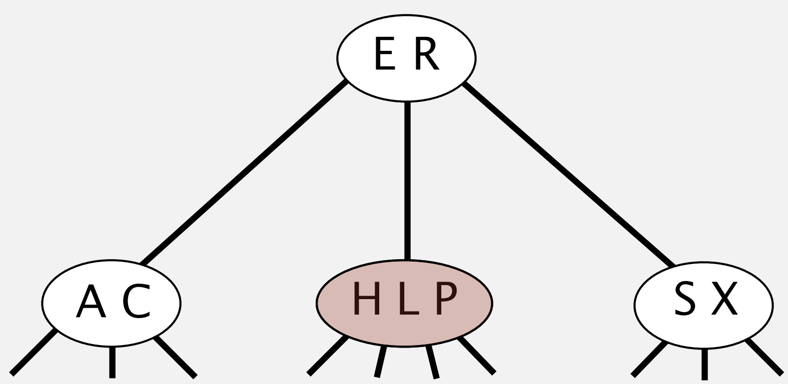
convert 3-node into 4-node

2–3 tree demo: insertion

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.

insert L

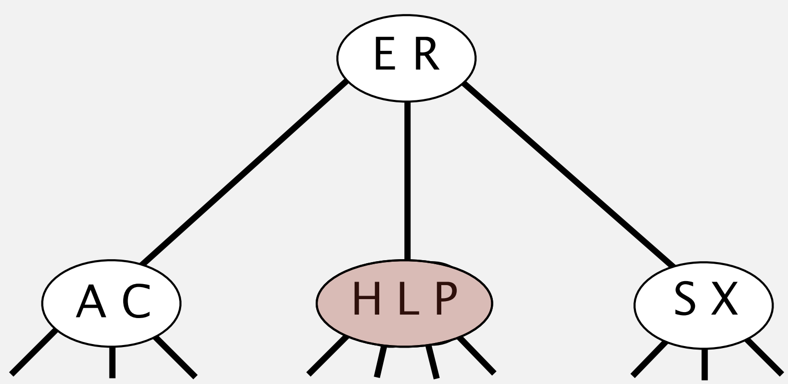


2–3 tree demo: insertion

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.

insert L



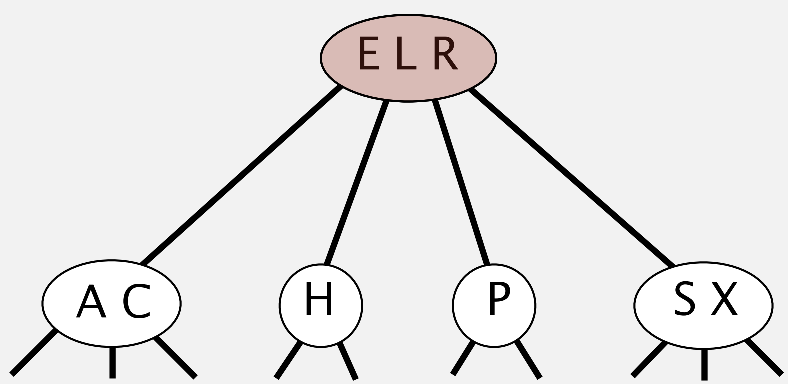
split 4-node
(move L to parent)

2-3 tree demo: insertion

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.

insert L

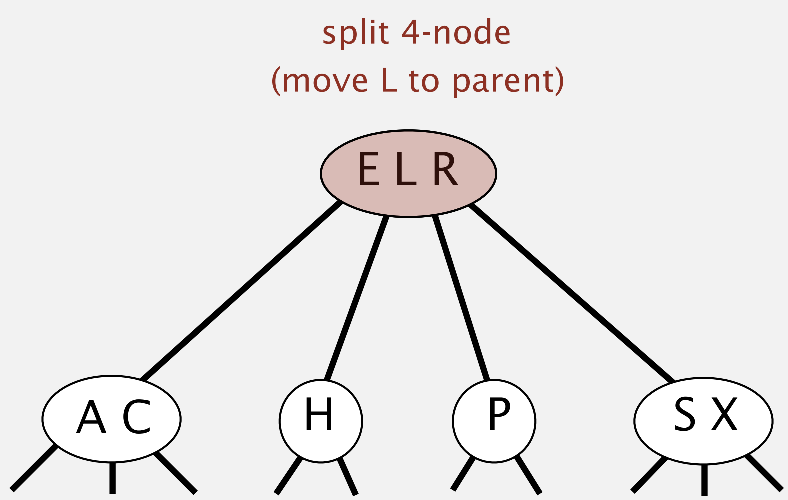


2–3 tree demo: insertion

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.

insert L



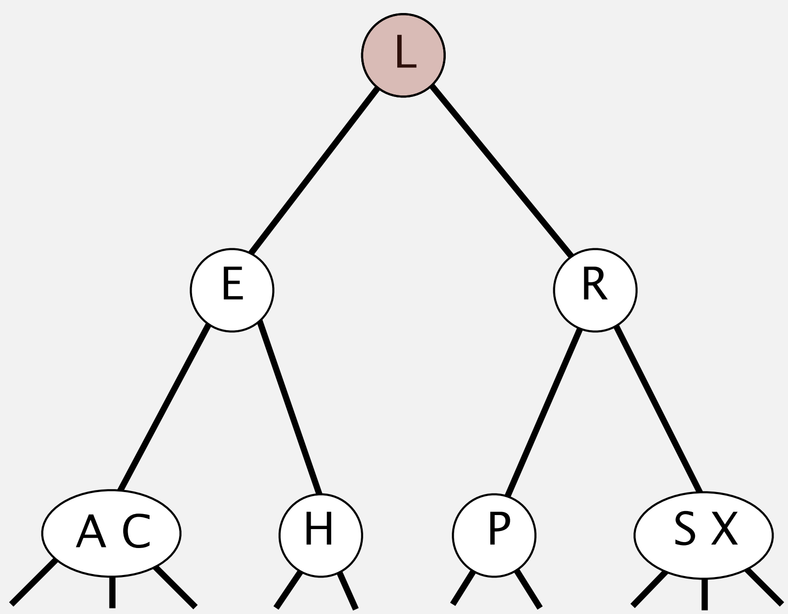
2-3 tree demo: insertion

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.

height of tree increases by 1

insert L

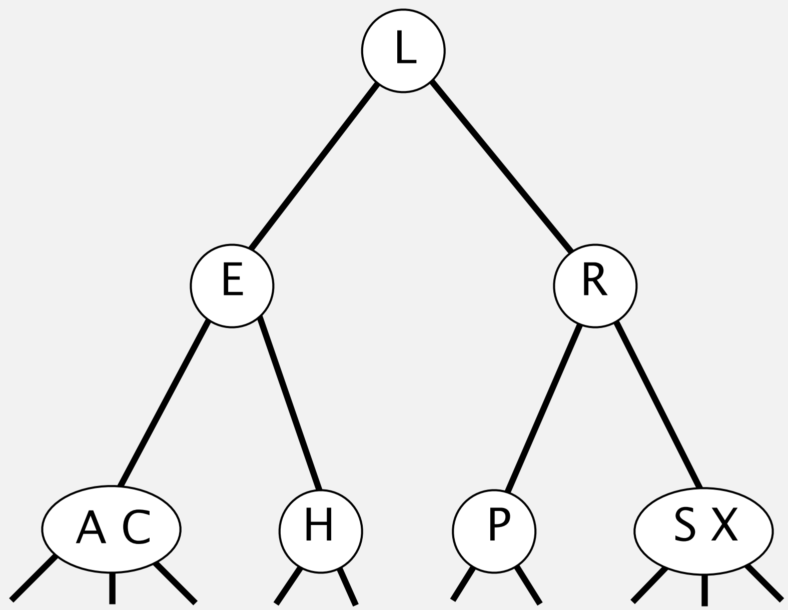


2–3 tree demo: insertion

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.

insert L



2–3 tree: insertion

Insertion into a 2-node at bottom.

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

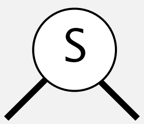
Insertion into a 3-node at bottom.

- 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.

Practice: draw the 2-3 tree construction for SEARCH

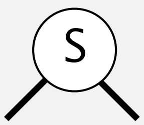
2-3 tree demo: construction

insert S



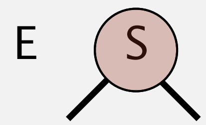
2-3 tree demo: construction

2-3 tree



2-3 tree demo: construction

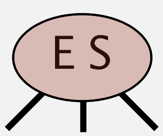
insert E



convert 2-node into 3-node

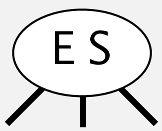
2–3 tree demo: construction

insert E



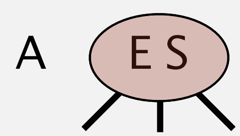
2-3 tree demo: construction

2-3 tree



2-3 tree demo: construction

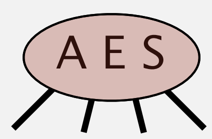
insert A



convert 3-node into 4-node

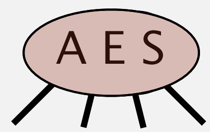
2-3 tree demo: construction

insert A



2-3 tree demo: construction

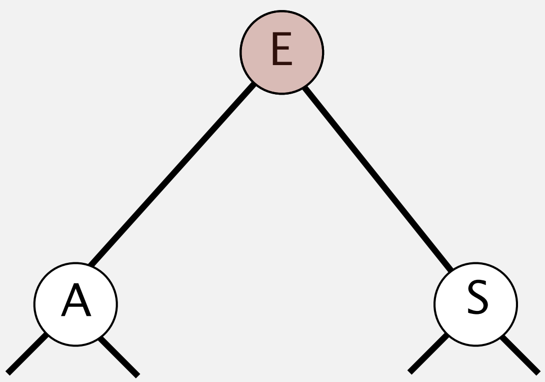
insert A



split 4-node
(move E to parent)

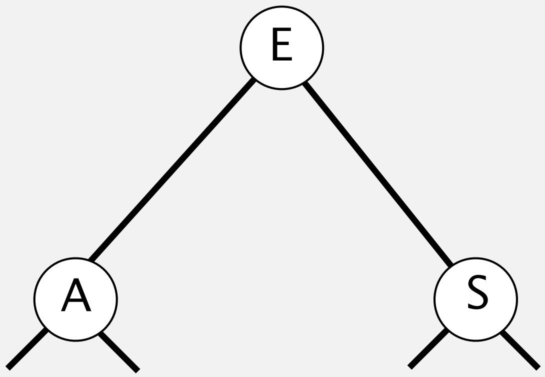
2-3 tree demo: construction

insert A



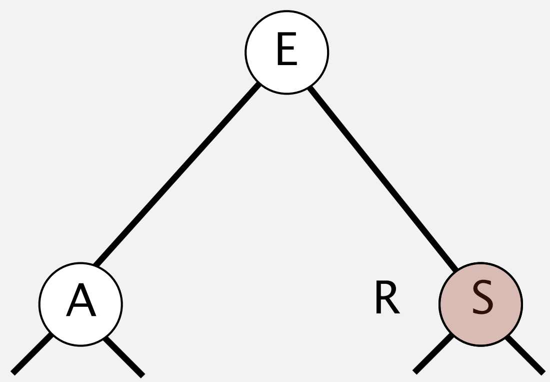
2-3 tree demo: construction

2-3 tree



2-3 tree demo: construction

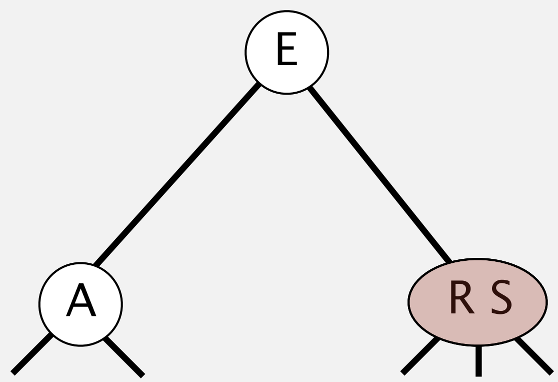
insert R



convert 2-node into 3-node

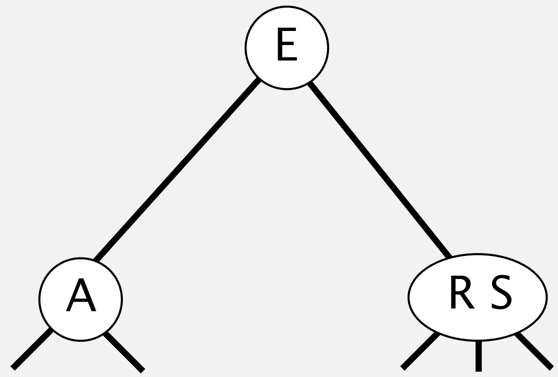
2-3 tree demo: construction

insert R



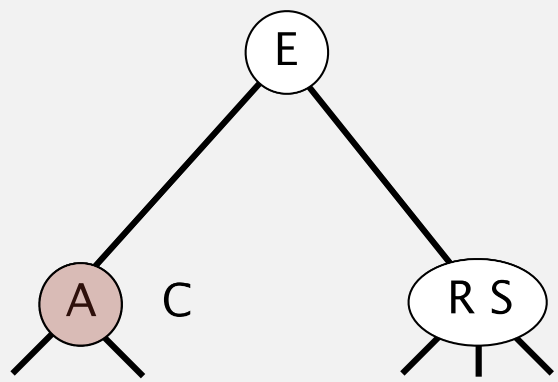
2-3 tree demo: construction

2-3 tree



2-3 tree demo: construction

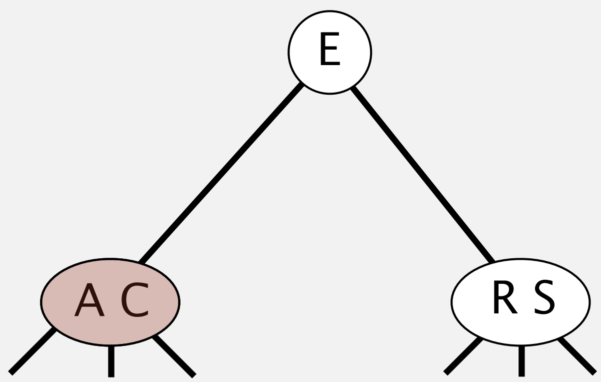
insert C



convert 2-node into 3-node

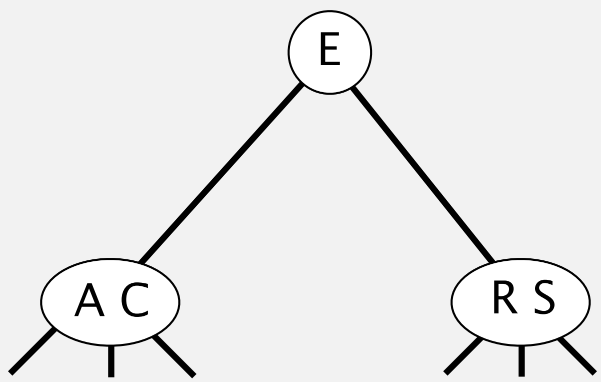
2-3 tree demo: construction

insert C



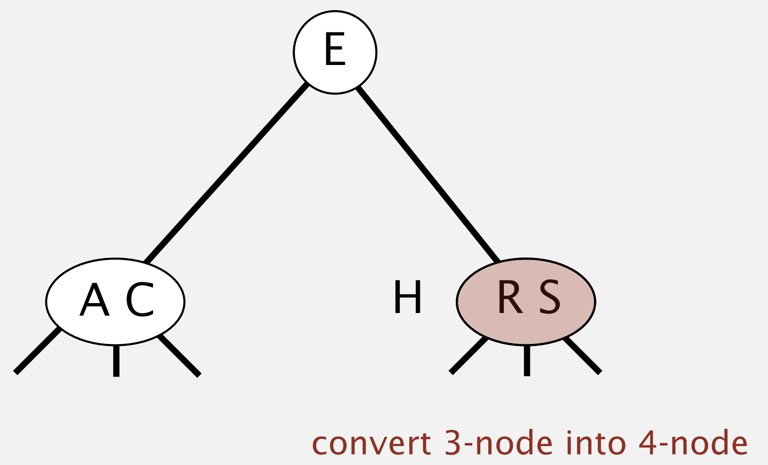
2-3 tree demo: construction

2-3 tree



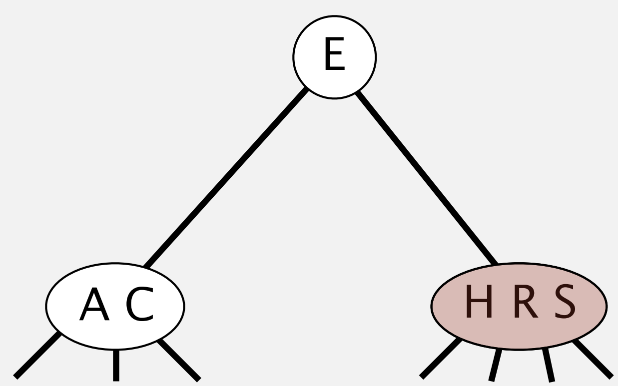
2-3 tree demo: construction

insert H



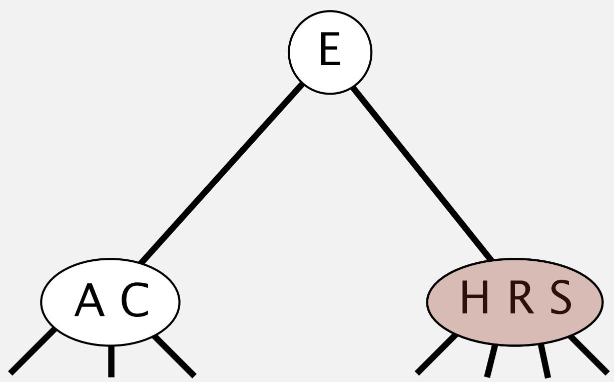
2-3 tree demo: construction

insert H



2-3 tree demo: construction

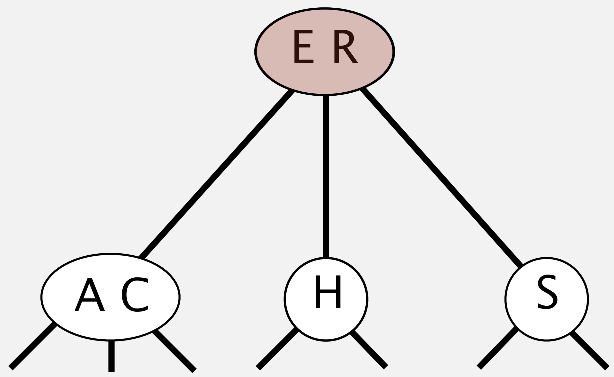
insert H



split 4-node
(move R to parent)

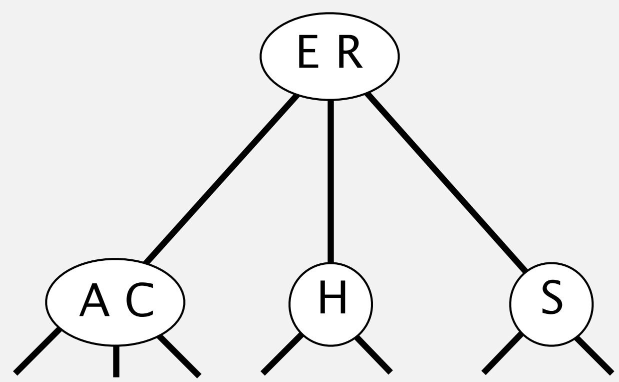
2-3 tree demo: construction

insert H



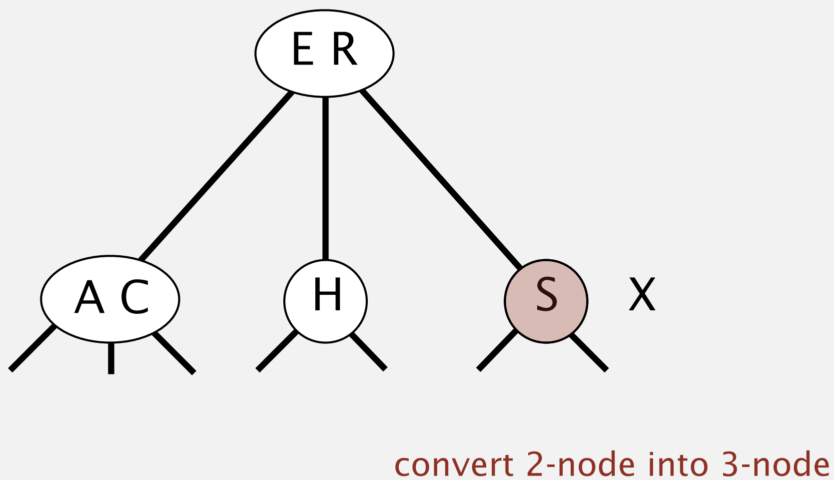
2-3 tree demo: construction

2-3 tree



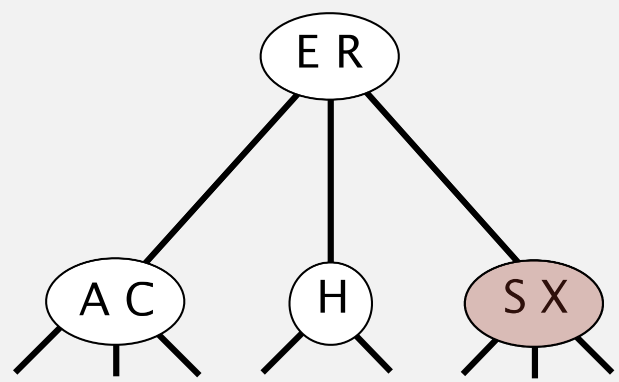
2-3 tree demo: construction

insert X



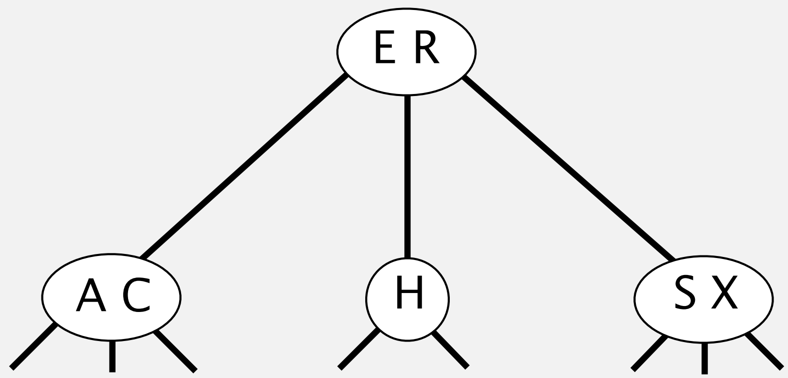
2-3 tree demo: construction

insert X



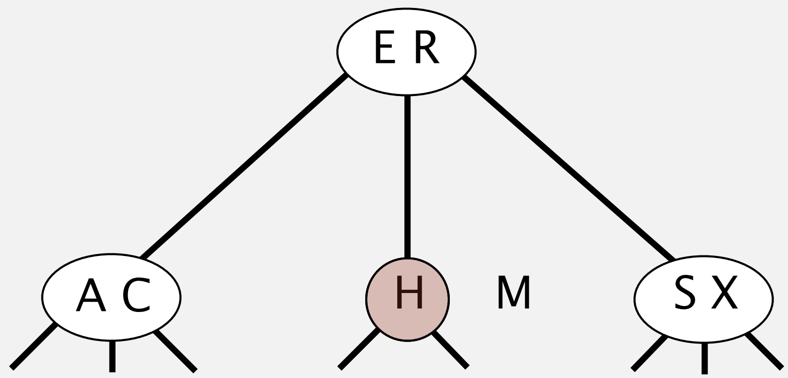
2-3 tree demo: construction

2-3 tree



2-3 tree demo: construction

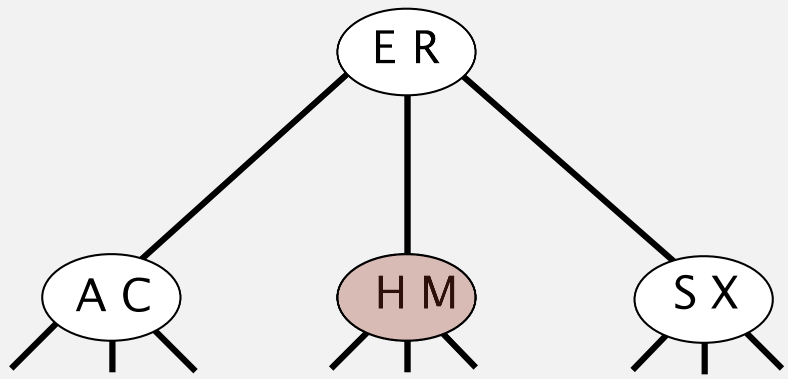
insert M



convert 2-node into 3-node

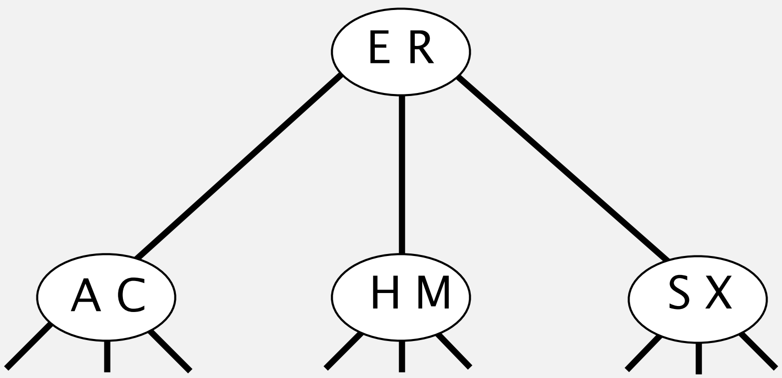
2-3 tree demo: construction

insert M



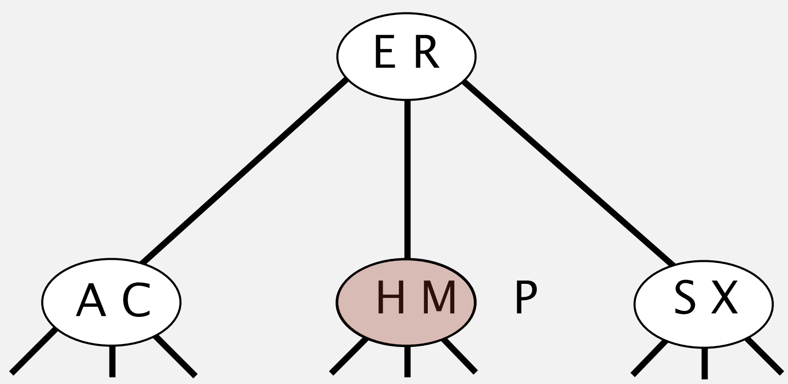
2-3 tree demo: construction

2-3 tree



2-3 tree demo: construction

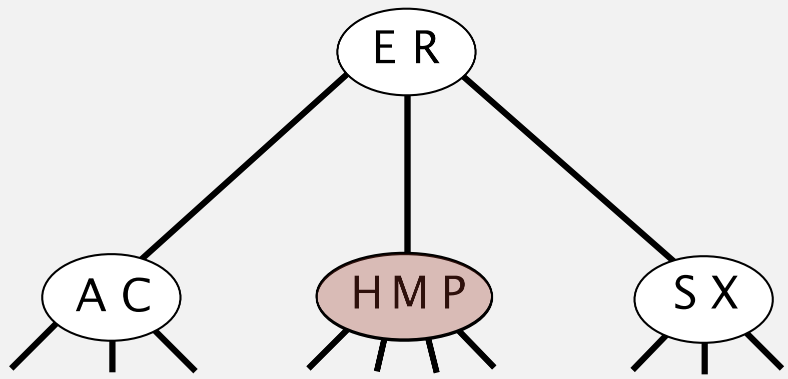
insert P



convert 3-node into 4-node

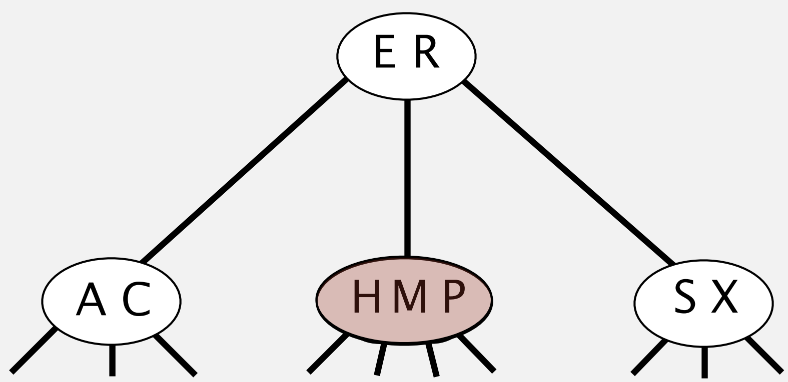
2-3 tree demo: construction

insert P



2-3 tree demo: construction

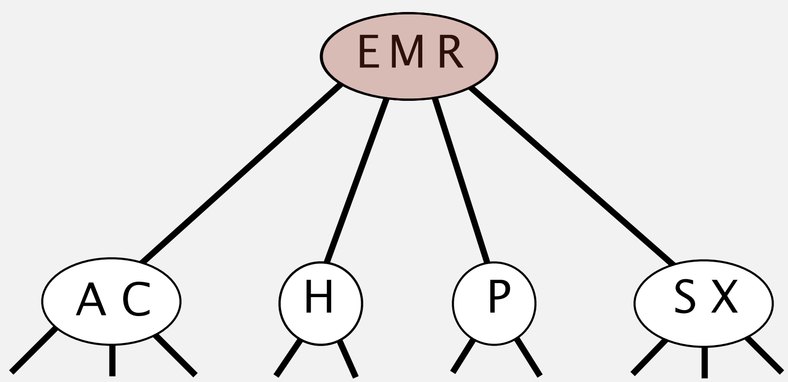
insert P



split 4-node
(move L to parent)

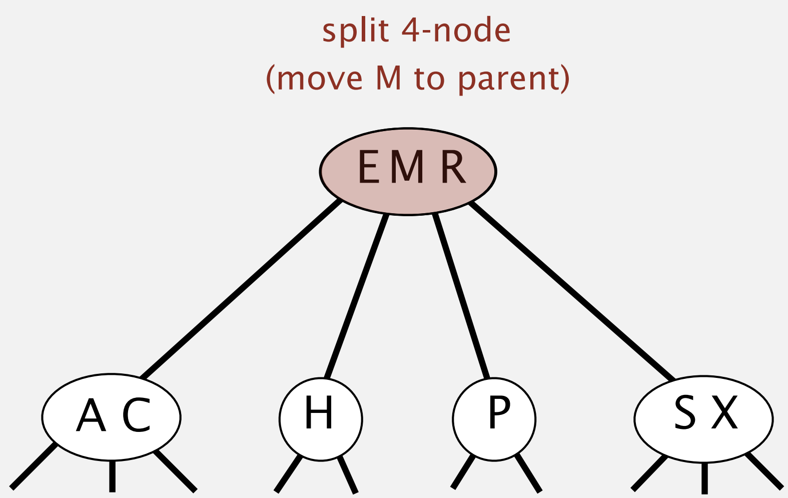
2-3 tree demo: construction

insert P



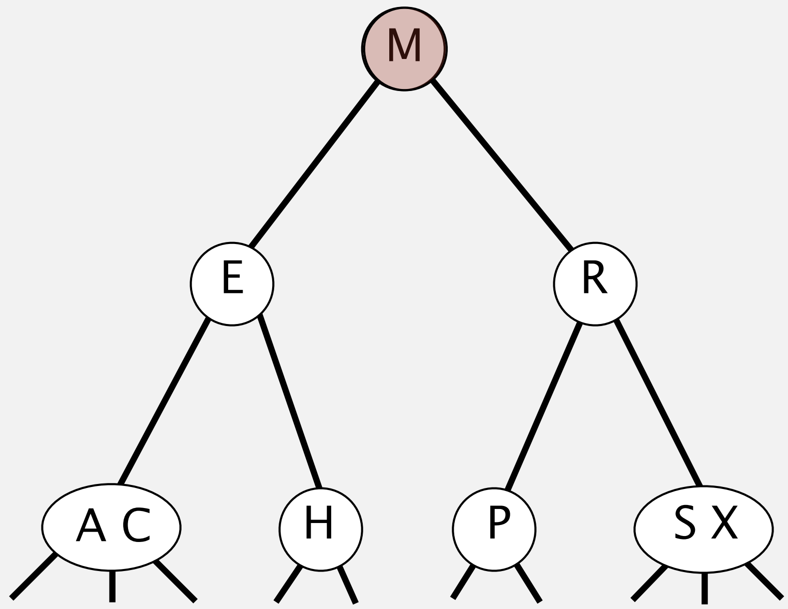
2-3 tree demo: construction

insert P



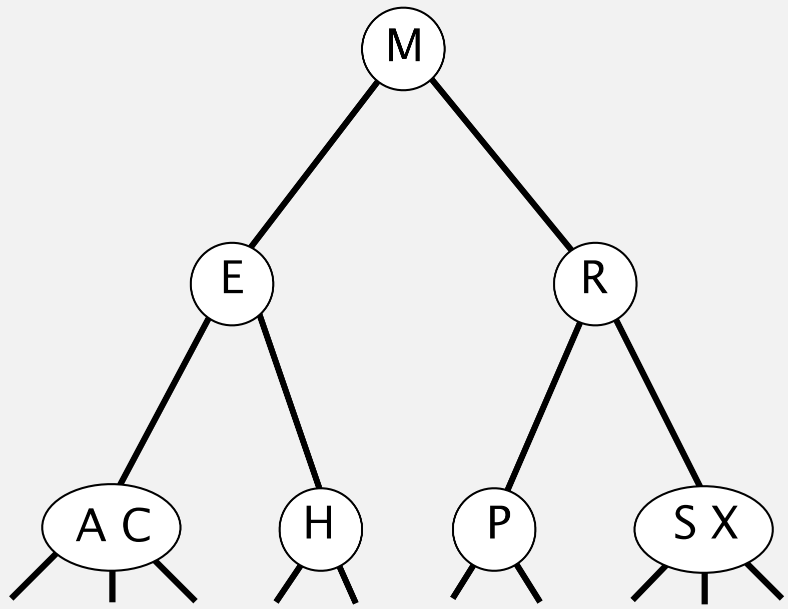
2-3 tree demo: construction

insert P



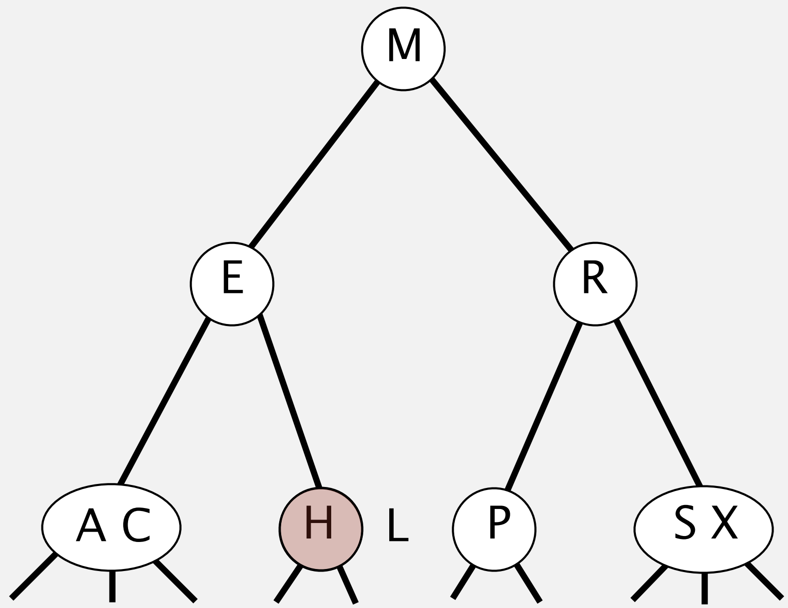
2-3 tree demo: construction

2-3 tree



2-3 tree demo: construction

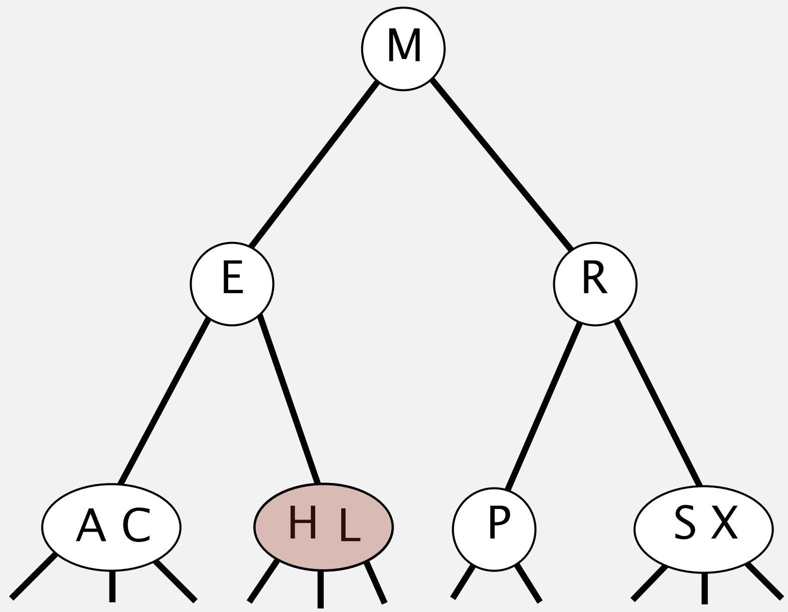
insert L



convert 2-node into 3-node

2-3 tree demo: construction

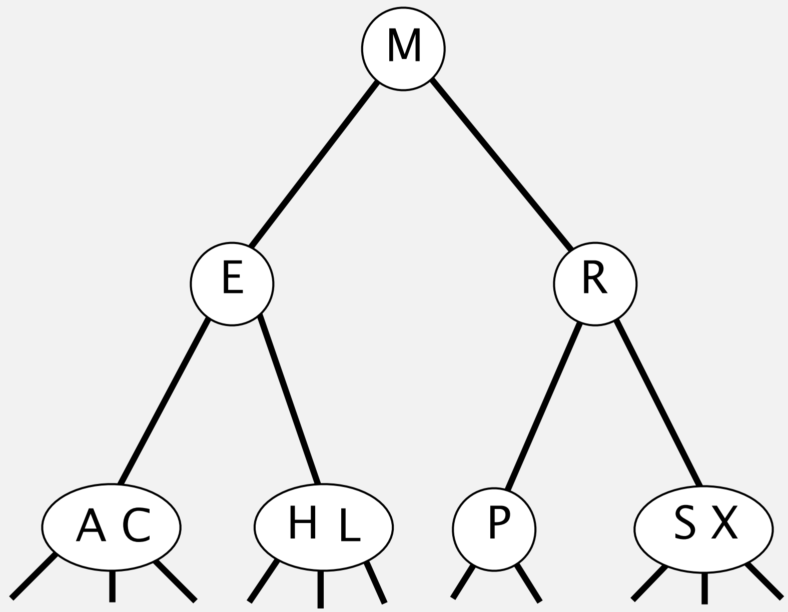
insert L



convert 2-node into 3-node

2-3 tree demo: construction

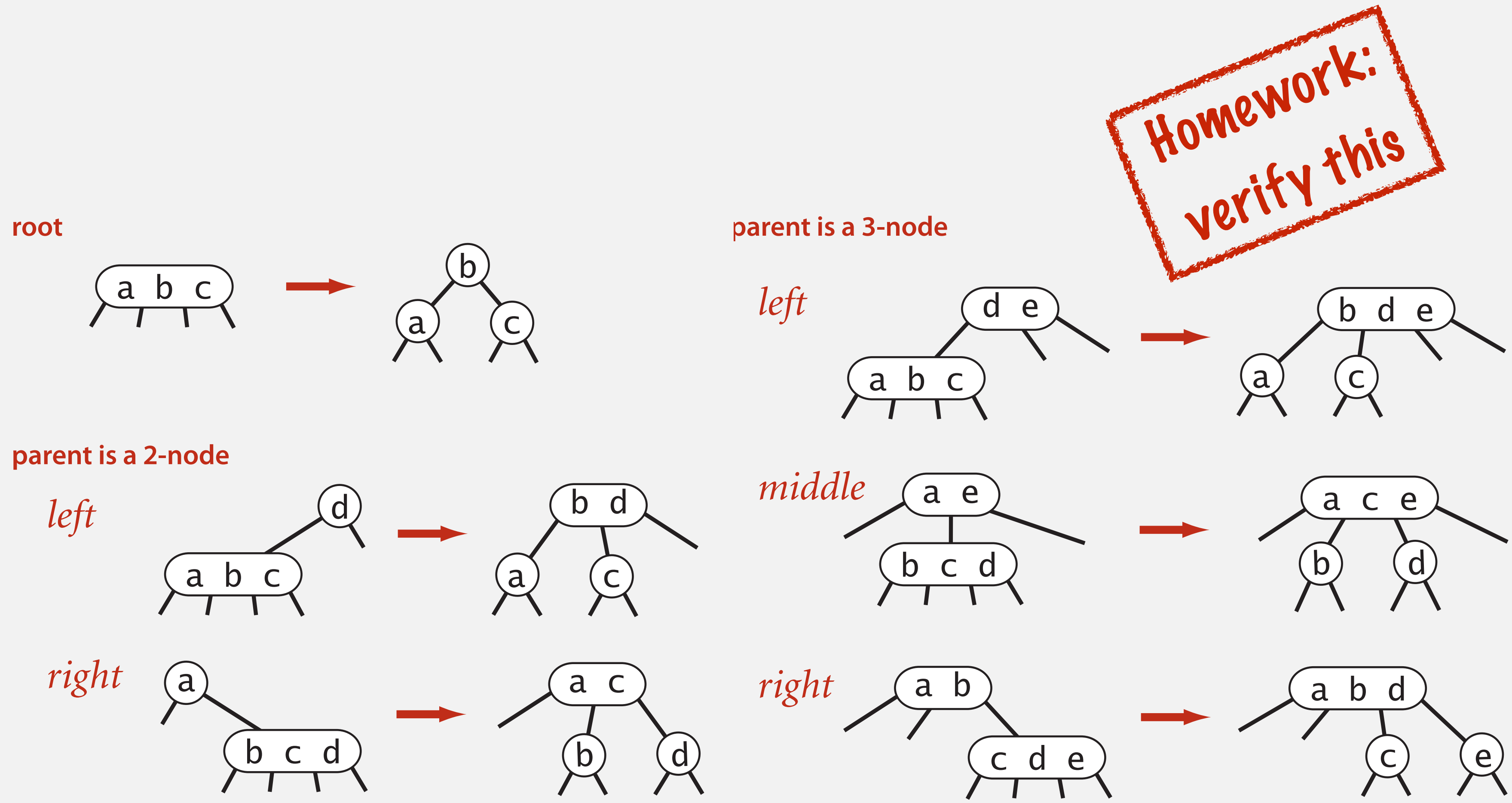
2-3 tree



2-3 tree: global properties

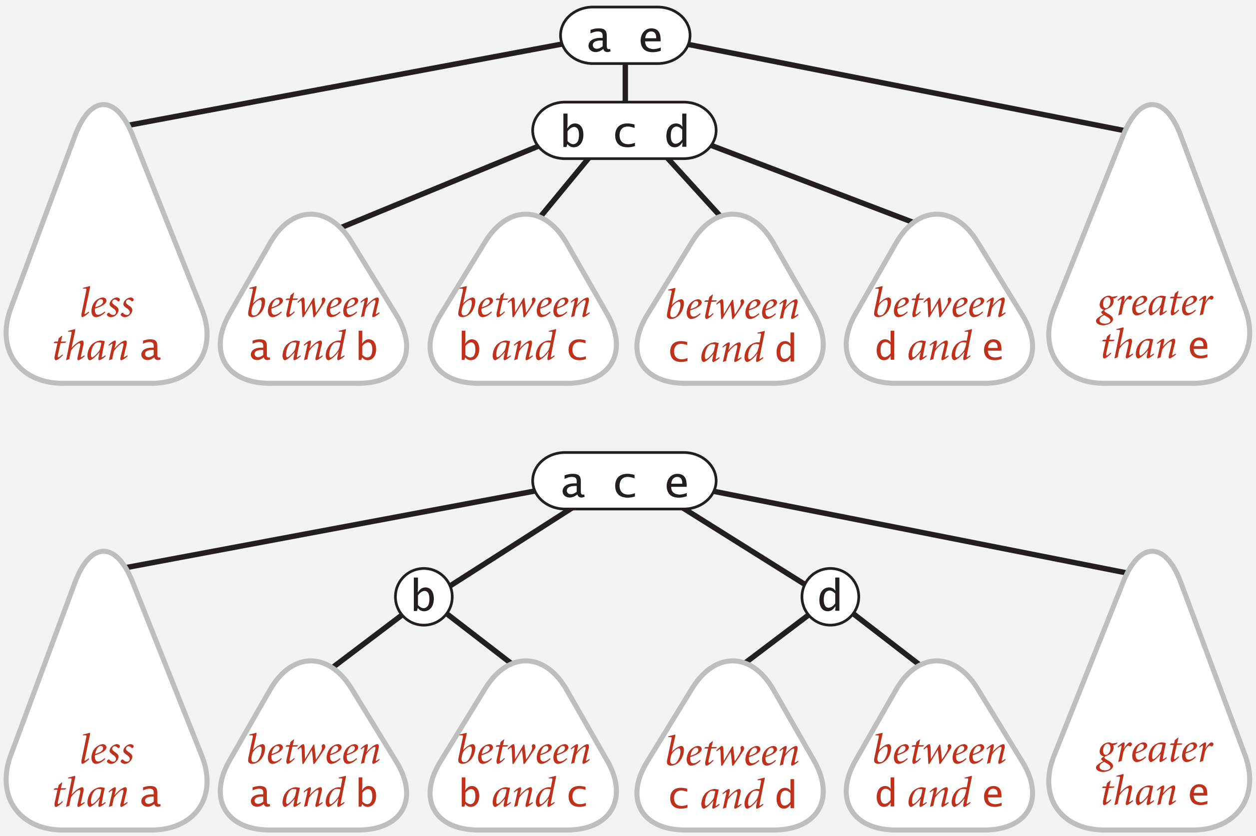
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.



2–3 tree: performance

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



Balanced search trees: quiz 1

What is the height of a 2–3 tree with N keys in the worst case?

A. $\sim \log_3 N$

B. $\sim \log_2 N$

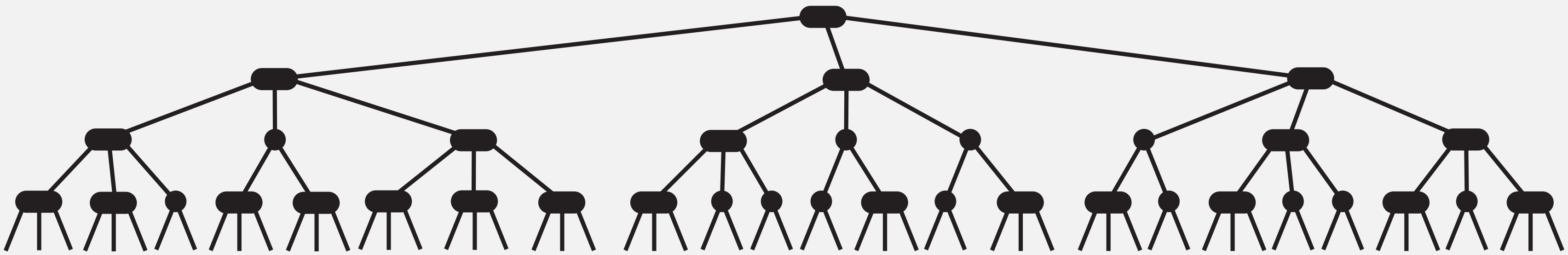
C. $\sim 2 \log_2 N$

D. $\sim N$

E. *I don't know.*

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	N	N	N		`equals()`
binary search (ordered array)	$\log N$	N	N	$\log N$	N	N	✔	`compareTo()`
BST	N	N	N	$\log N$	$\log N$	$\sqrt{N}$	✔	`compareTo()`
2-3 tree	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`

but hidden constant is large
(depends 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.

> *" Beautiful algorithms are not always the most useful. "*
>
> *— Donald Knuth*

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

http://algs4.cs.princeton.edu

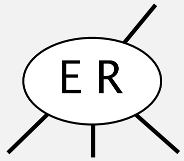
3.3 Balanced Search Trees

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

left-leaning version optimized for teaching and coding;
developed by Bob Sedgewick in creating this course!

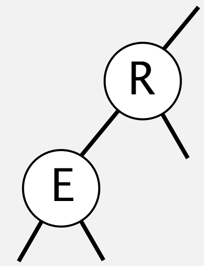
How to implement 2–3 trees with binary trees?

Challenge. How to represent a 3 node?



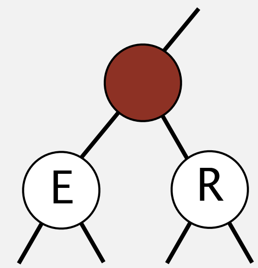
Approach 1. Regular BST.

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



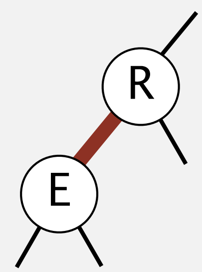
Approach 2. Regular BST with red "glue" nodes.

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



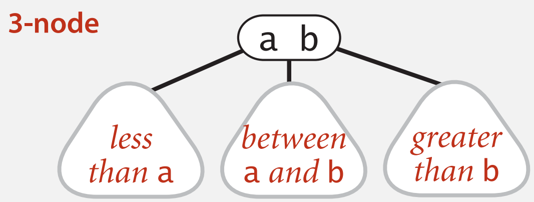
Approach 3. Regular BST with red "glue" links.

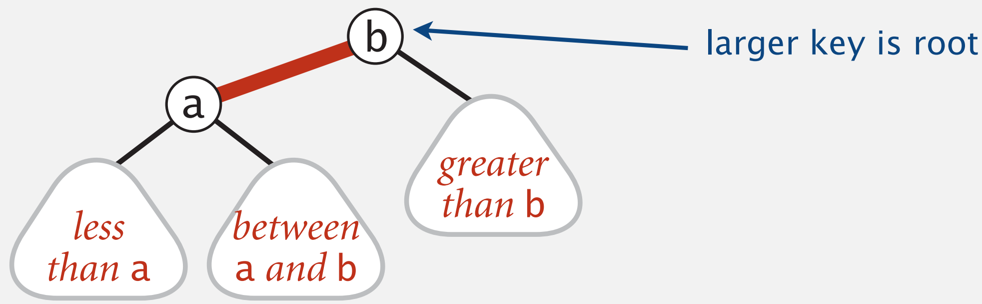
- Widely used in practice.
- Arbitrary restriction: red links lean left.

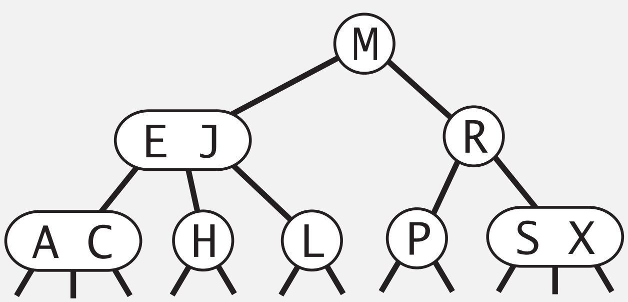


Left-leaning red–black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

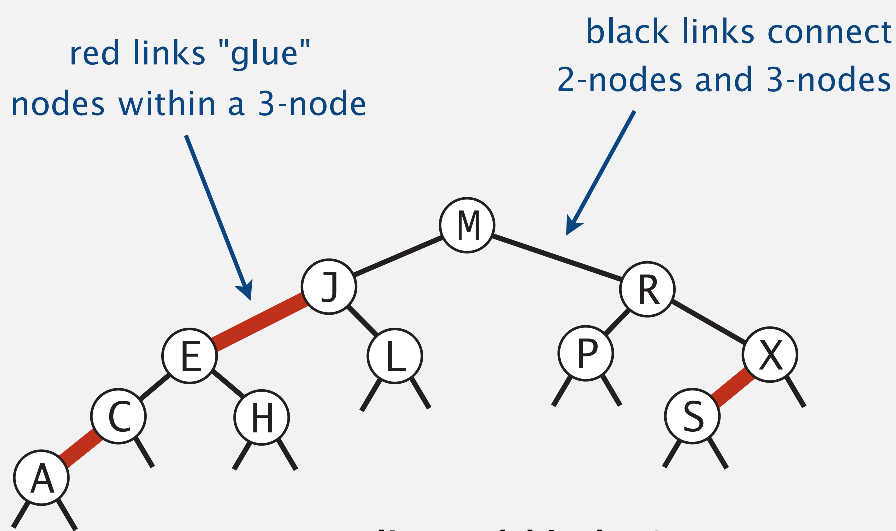
1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.







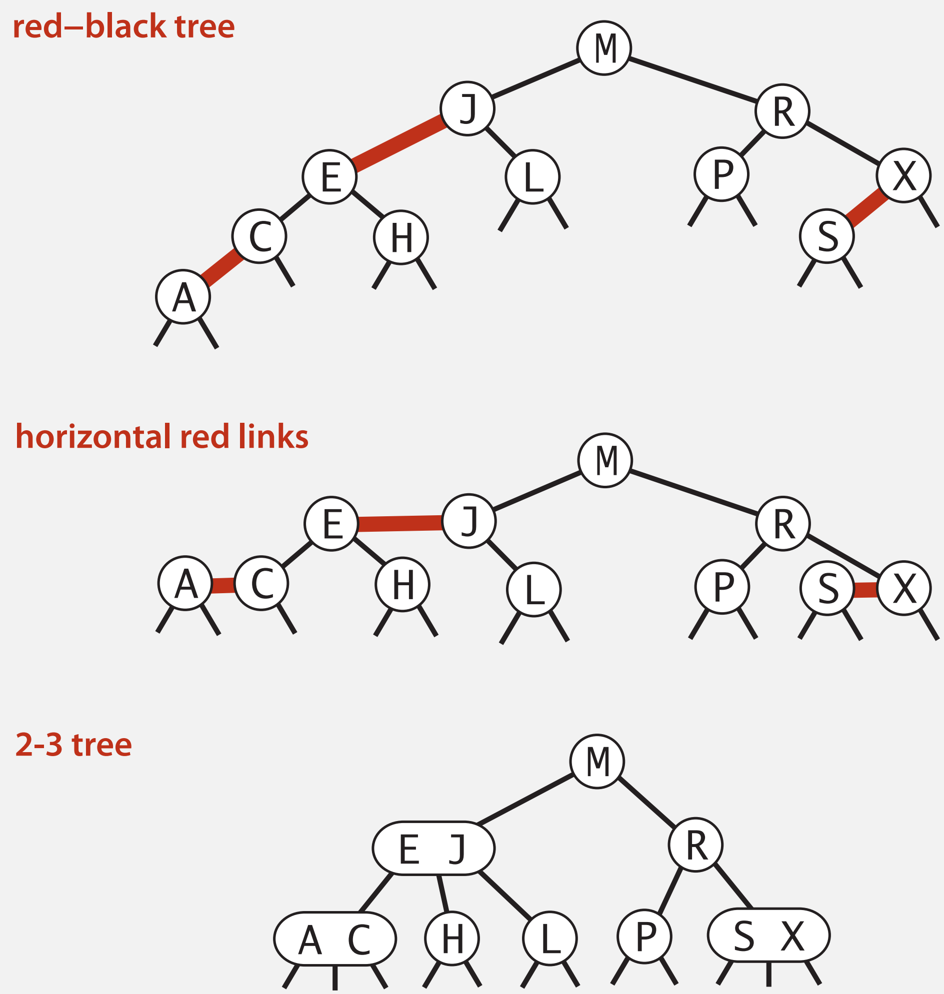
2–3 tree



corresponding red–black BST

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

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

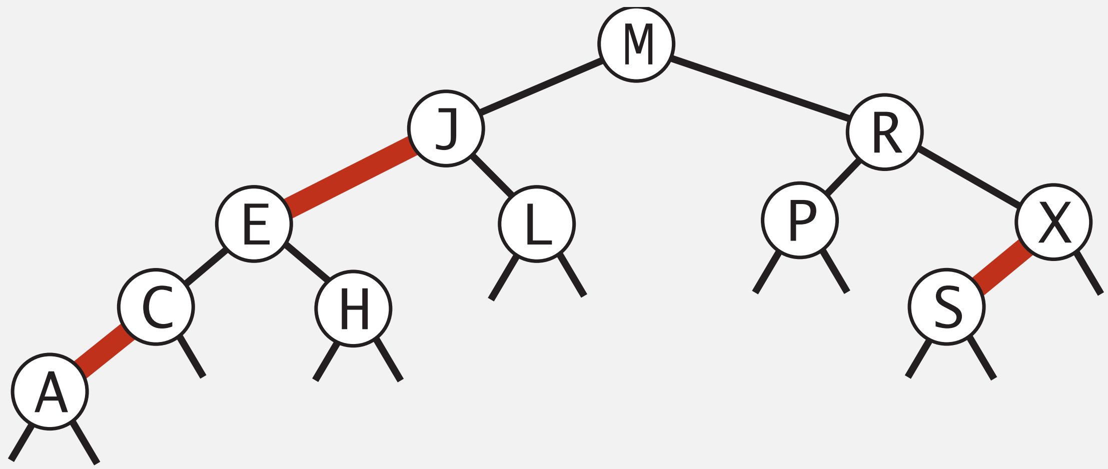


Definition of left-leaning red-black tree

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"

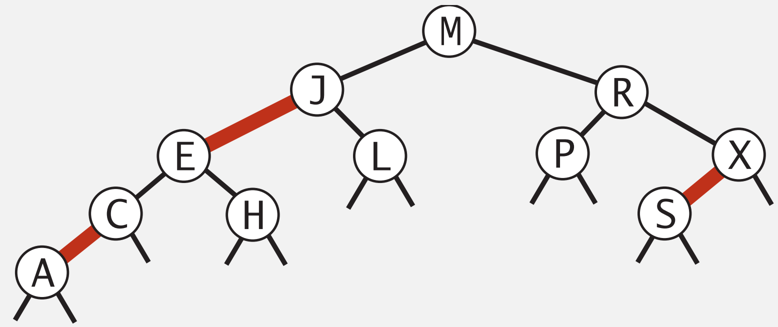


Search implementation for red-black BSTs

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

but runs faster because
of better balance

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



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

Red-black BST representation

Q. How to represent color of links in Java data structure?

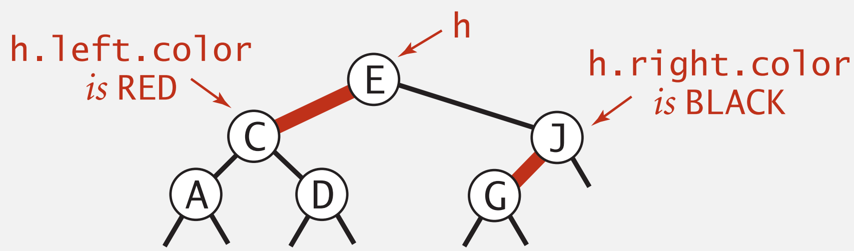
Each node is pointed to by precisely one link (from its parent) $\Rightarrow$ can encode color of links in nodes.

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

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

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

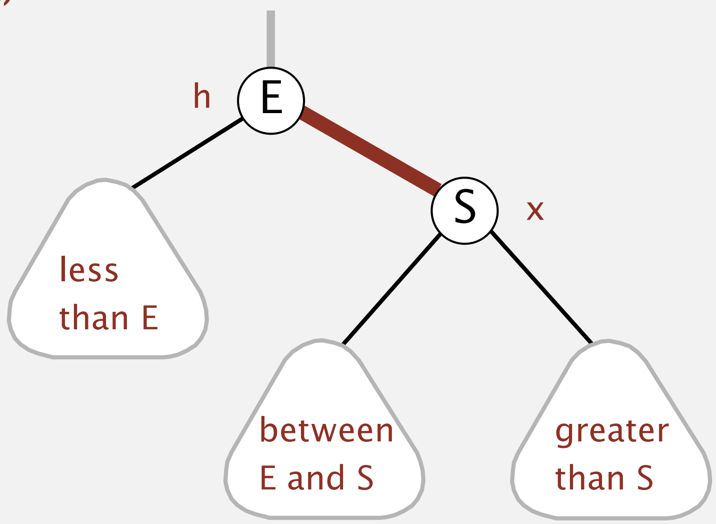
null links are black



Elementary red-black BST operations

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

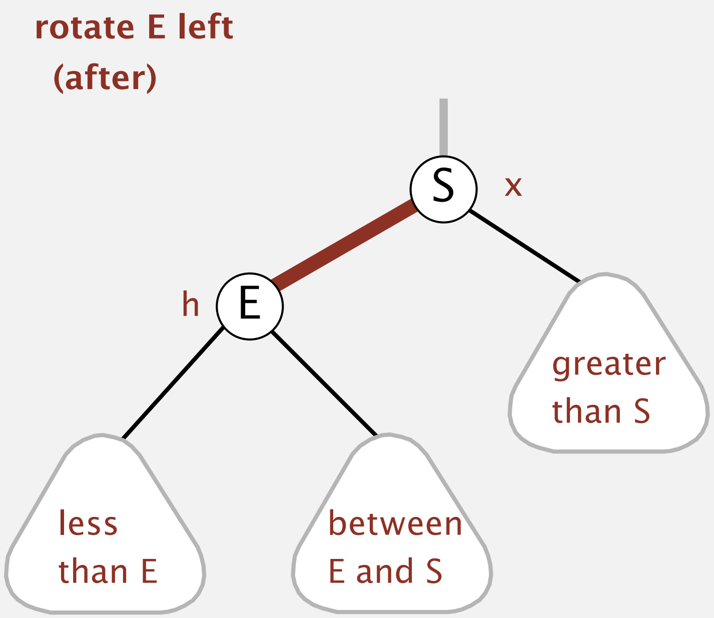
rotate E left
(before)



Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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



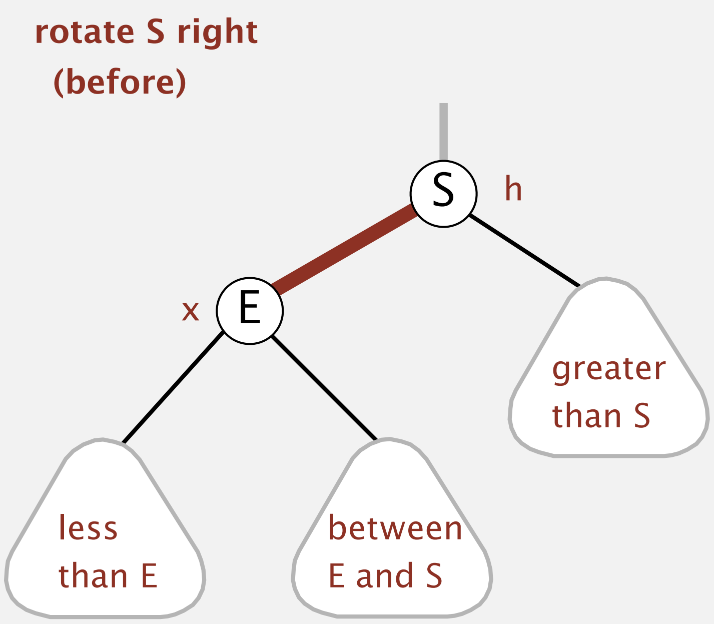

```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Skipped in class

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

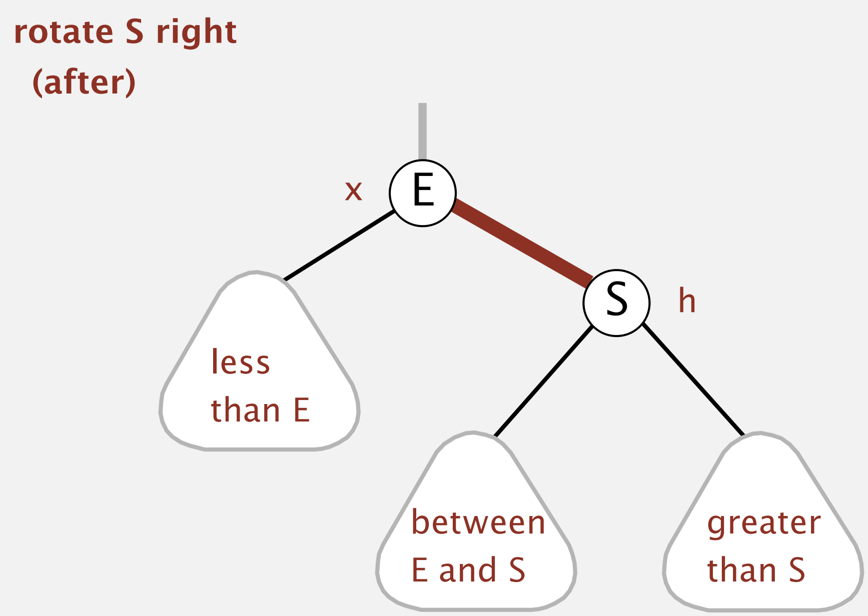
Right rotation. Orient a left-leaning red link to (temporarily) lean right.



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.



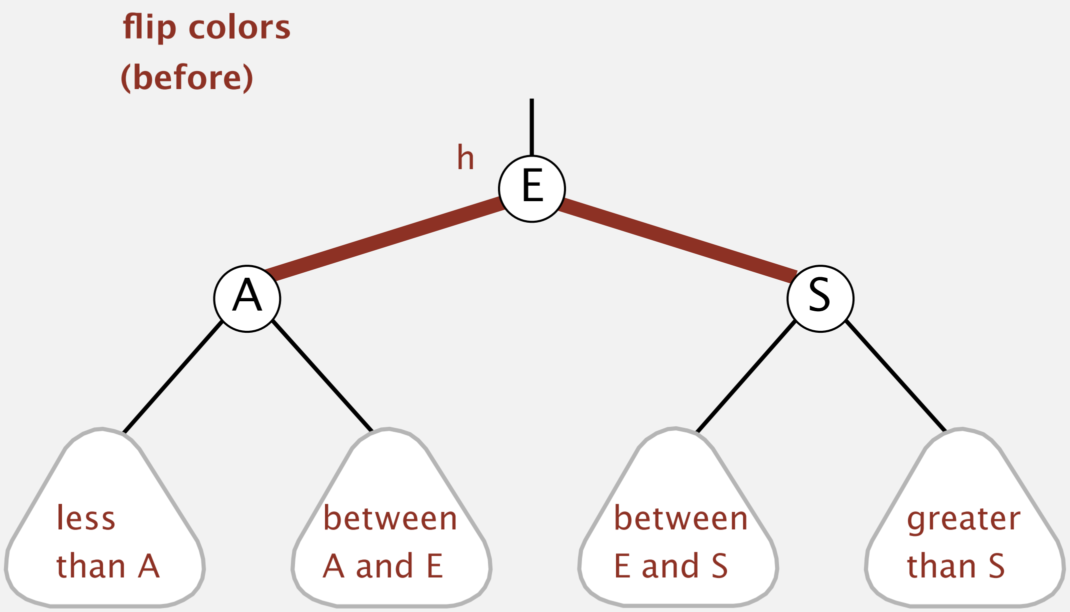

```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Skipped in class

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

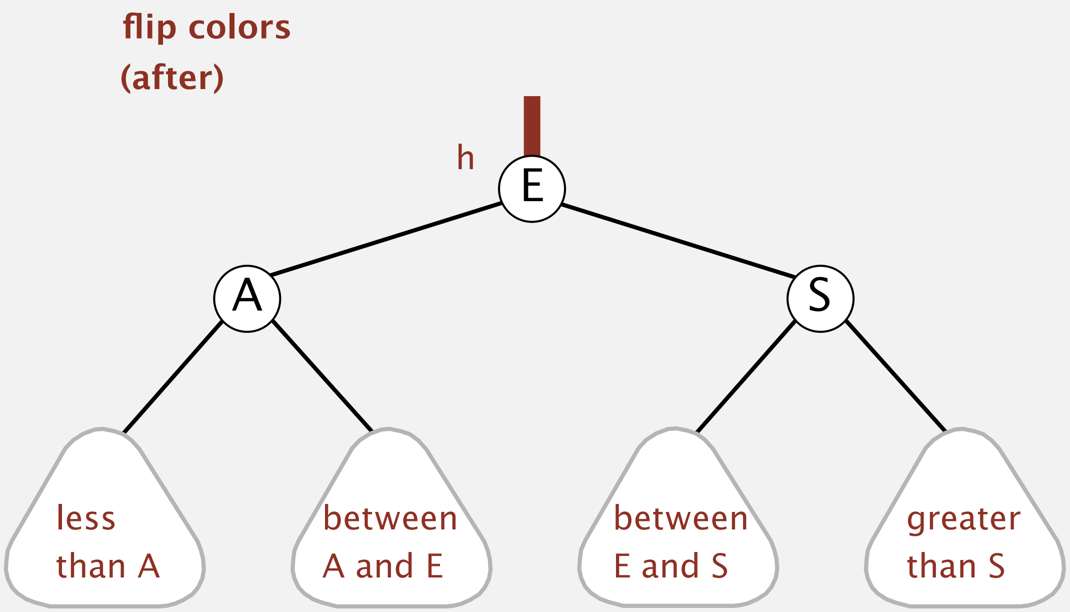
Color flip. Recolor to split a (temporary) 4-node.



Invariants. Maintains symmetric order and perfect black balance.

Elementary red–black BST operations

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



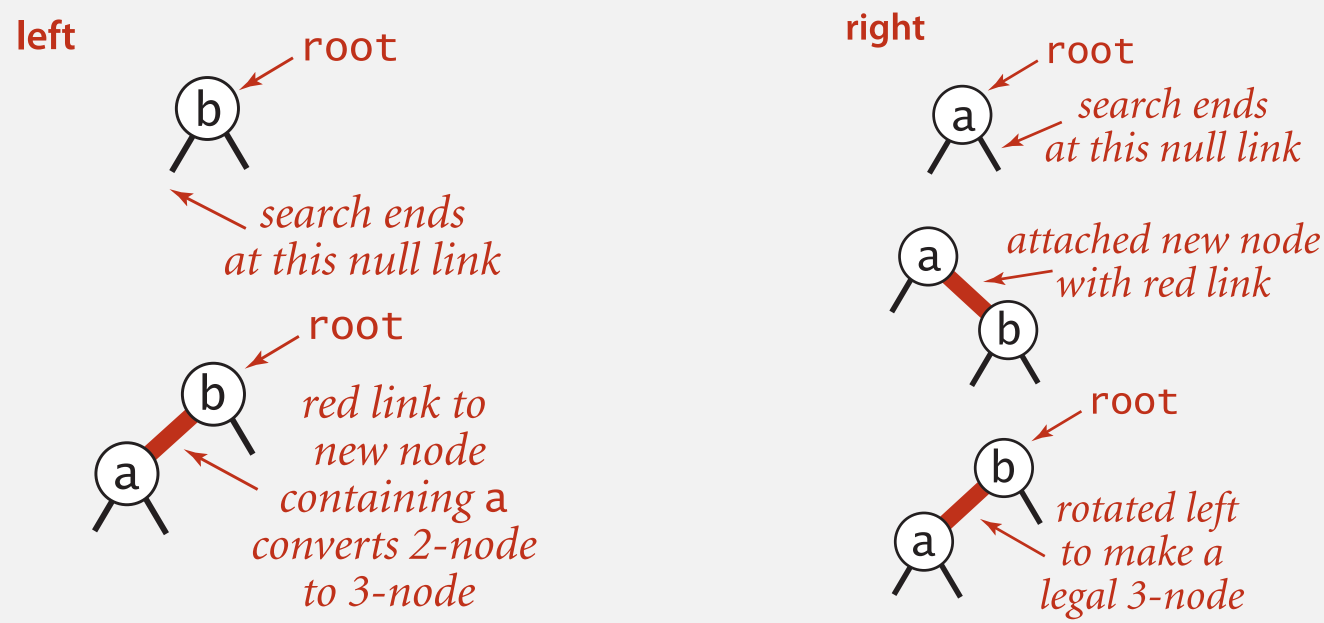

```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Skipped in class

Invariants. Maintains symmetric order and perfect black balance.

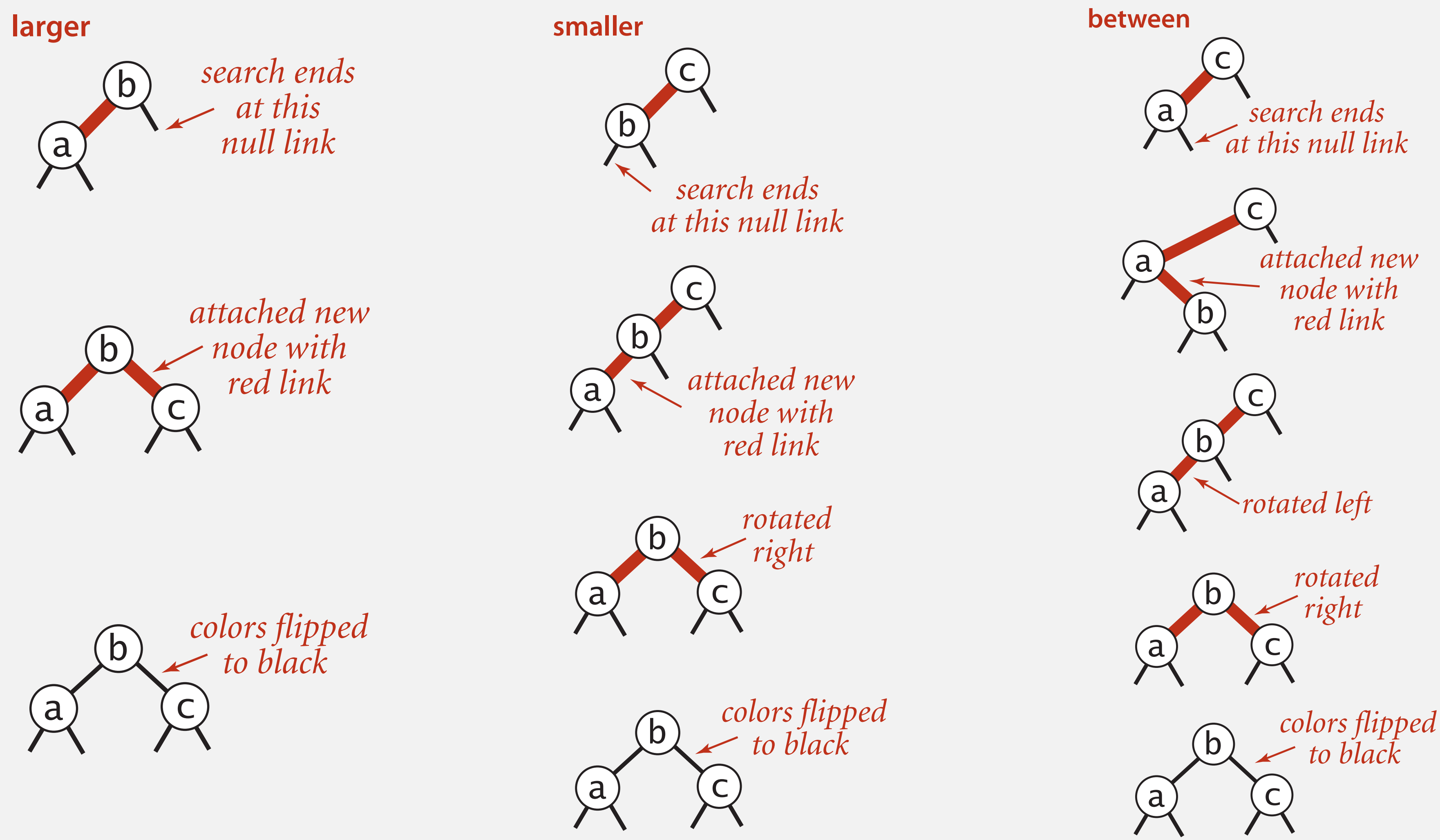
Insertion into a LLRB tree

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



Insertion into a LLRB tree

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

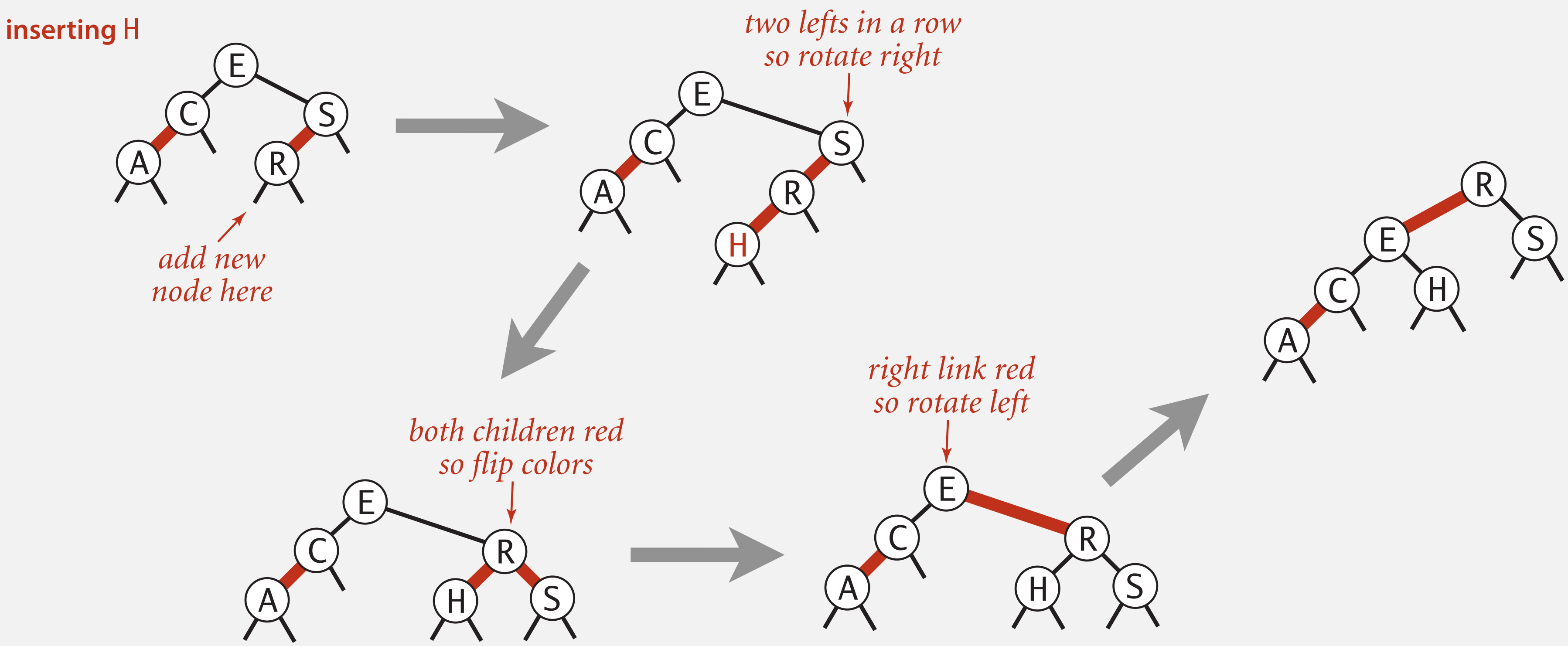


Insertion into a LLRB tree

General case.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left red links in a row: rotate right.
  - Both children red: flip colors.

← to fix color invariants

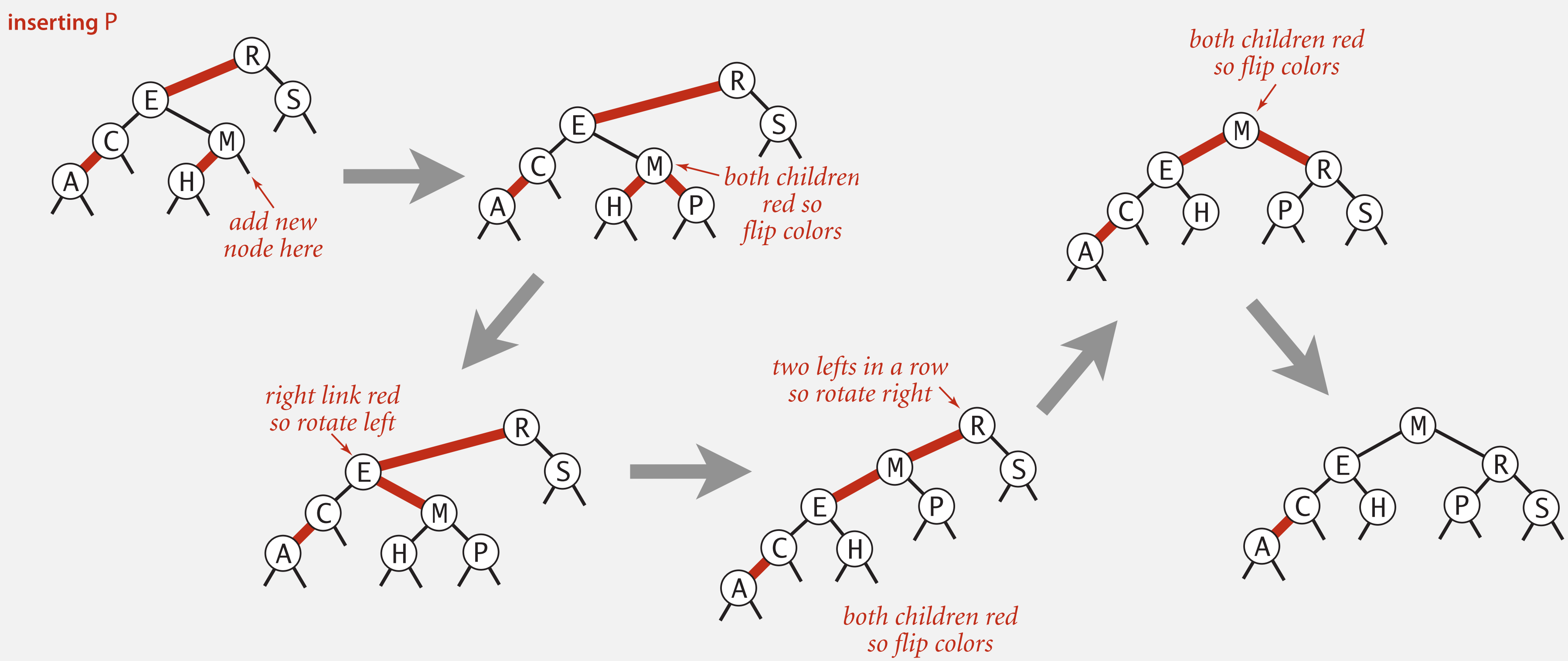


Insertion into a LLRB tree: passing red links up the tree

General case.

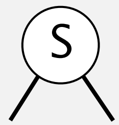
- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left red links in a row: rotate right.
  - Both children red: flip colors.

  ← to fix color invariants



Red-black BST construction practice: SEARCH

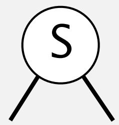
insert S



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

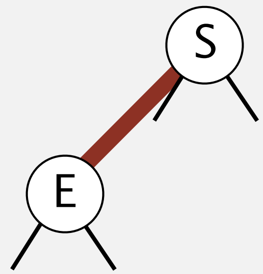
insert S



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

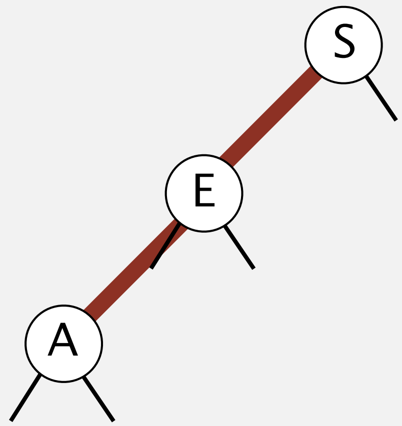
insert E



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

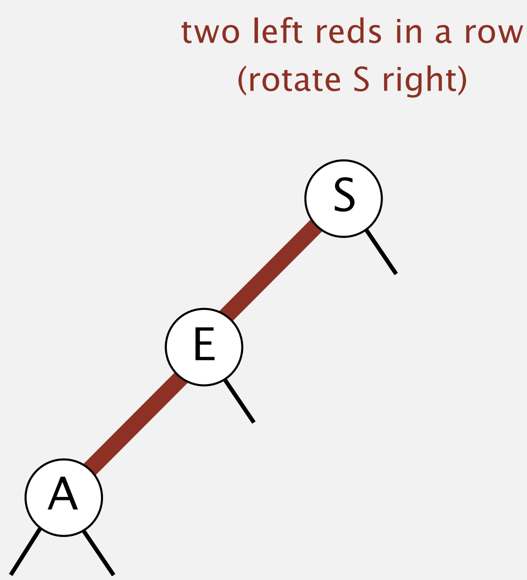
insert A



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

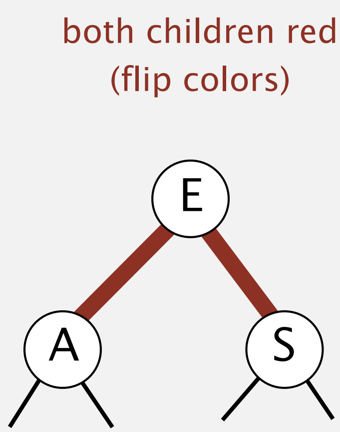
Red-black BST construction demo

insert A



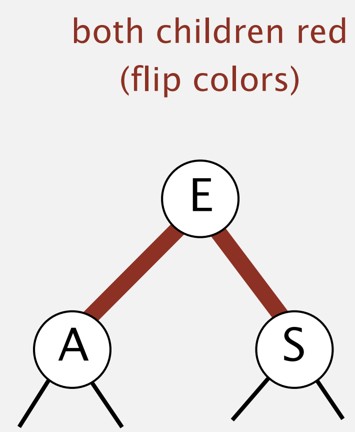
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

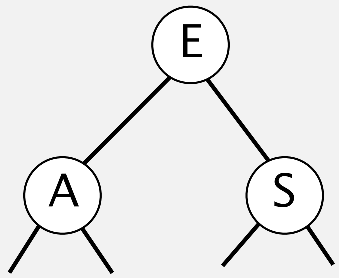
Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

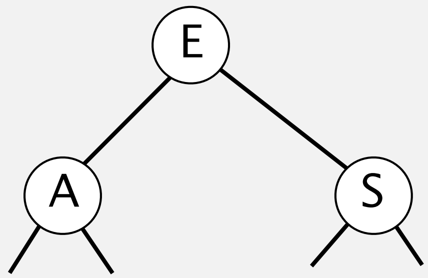
red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

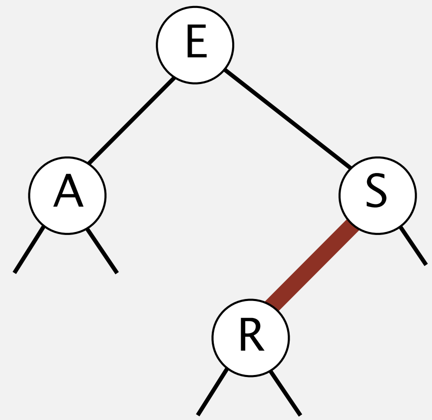
red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

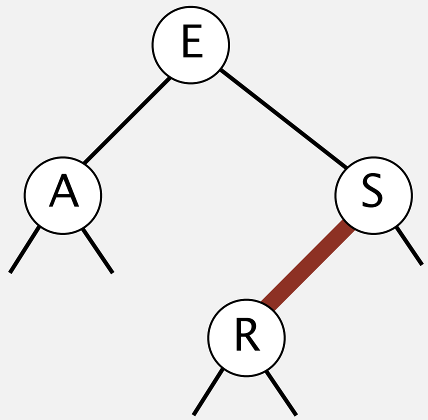
insert R



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

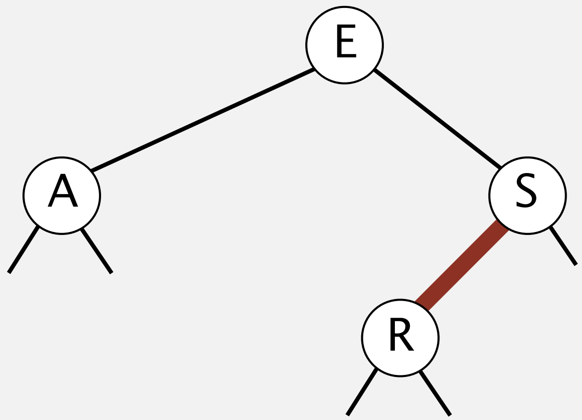
red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

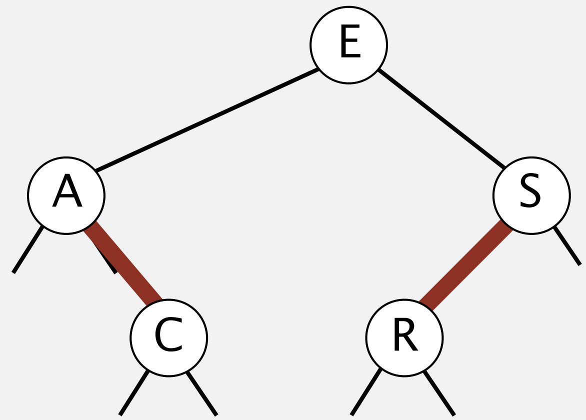
red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

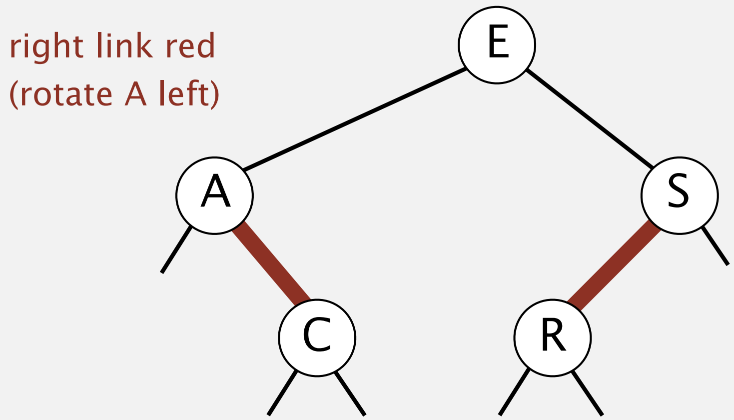
Red-black BST construction demo

insert C



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

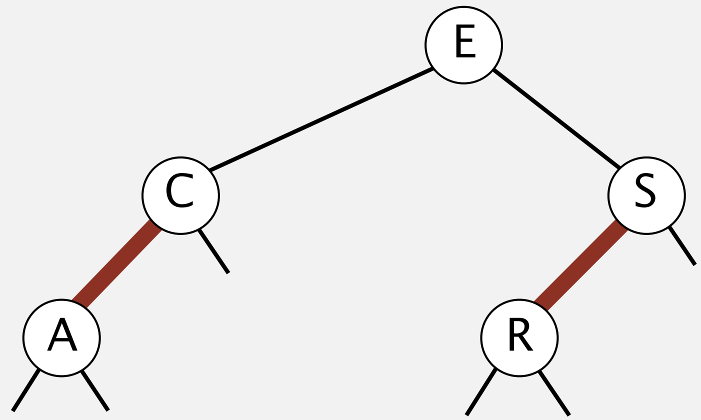
Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

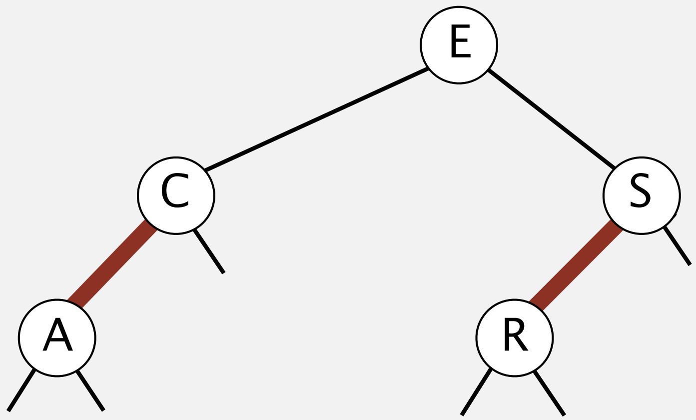
red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

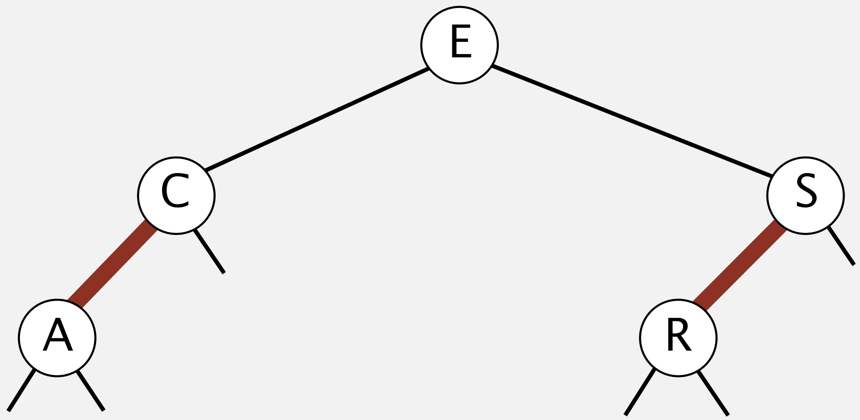
red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

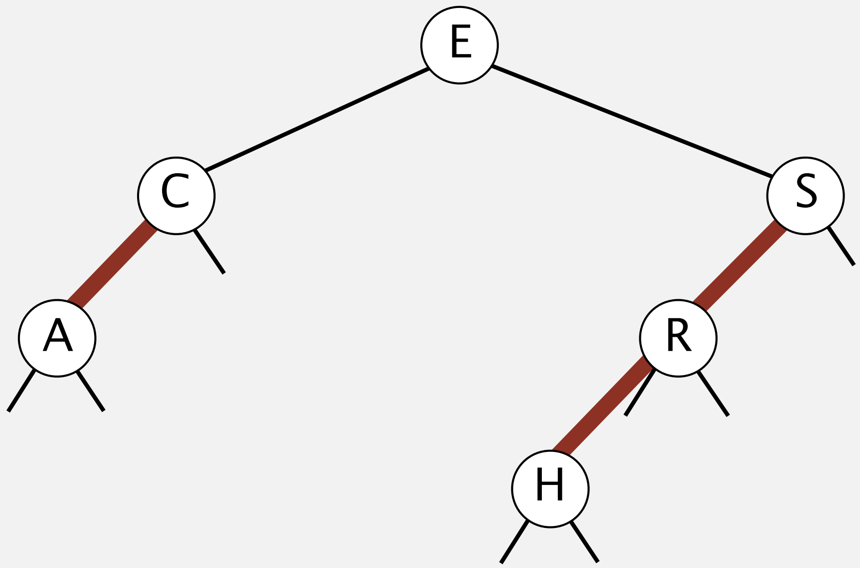
red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

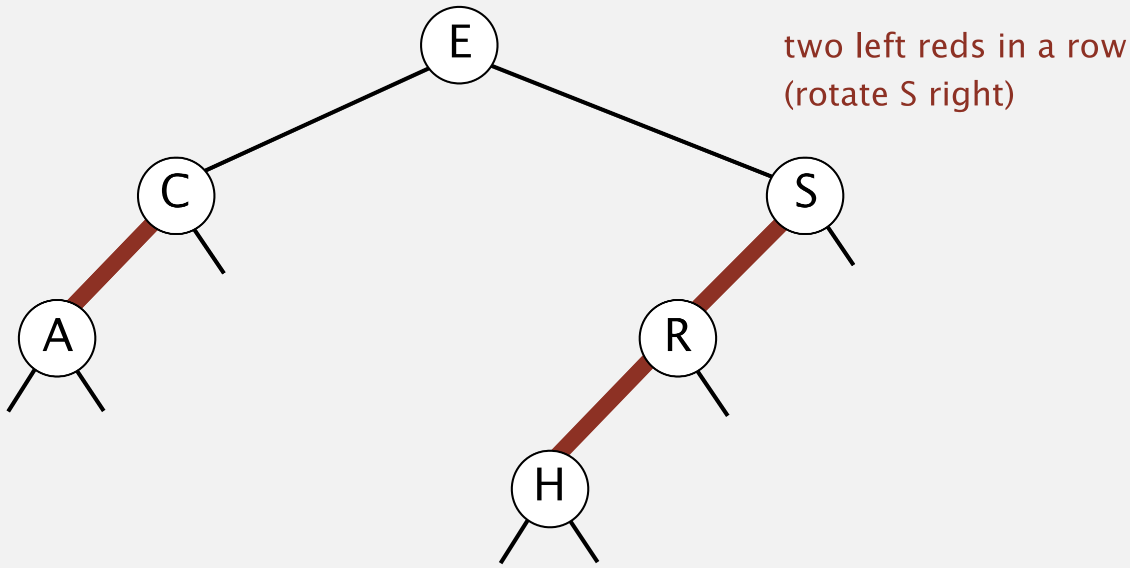
Red-black BST construction demo

insert H



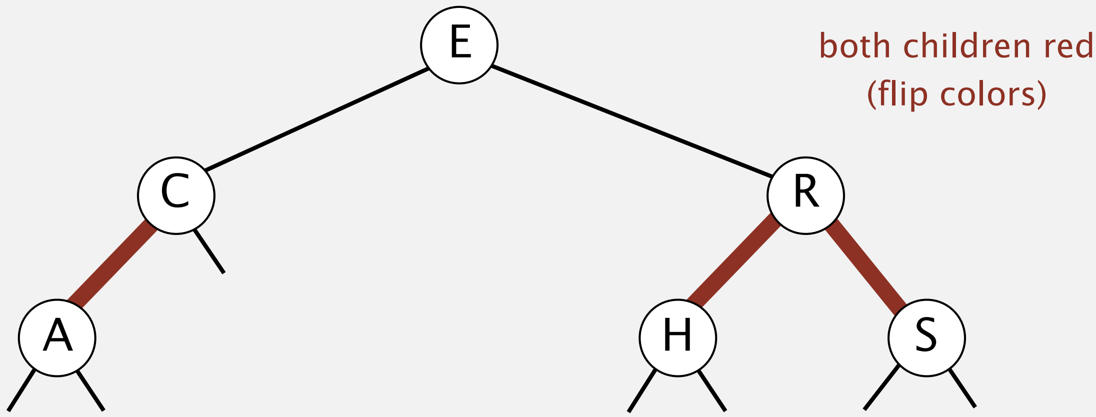
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



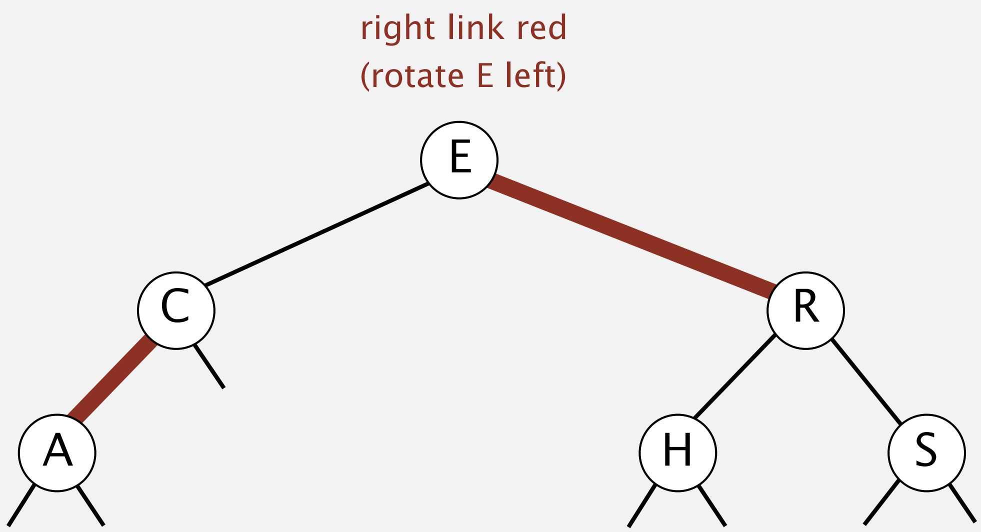
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

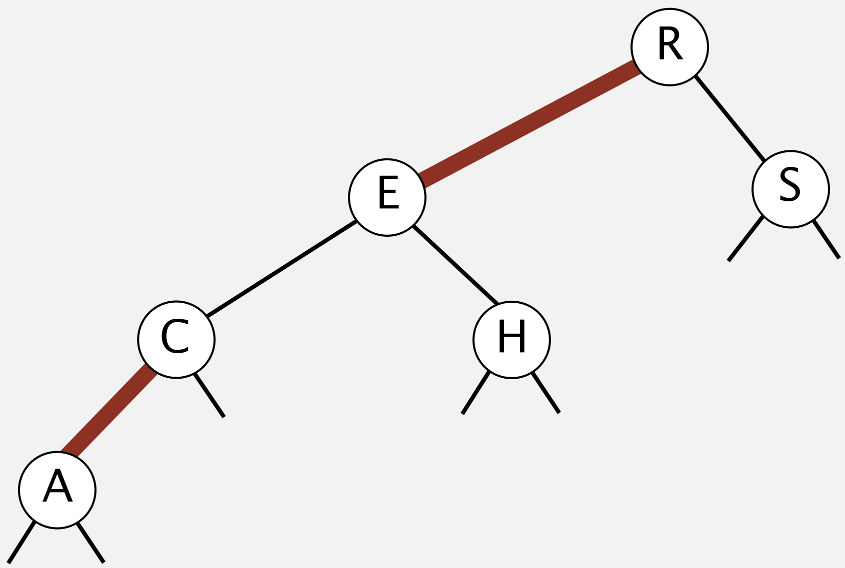
Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

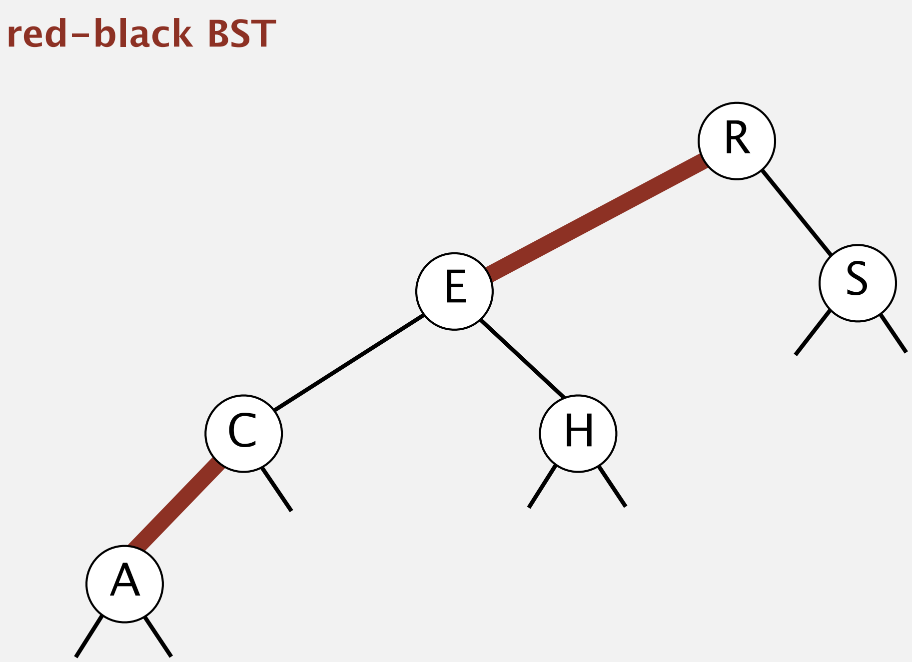
Red-black BST construction demo

red–black BST



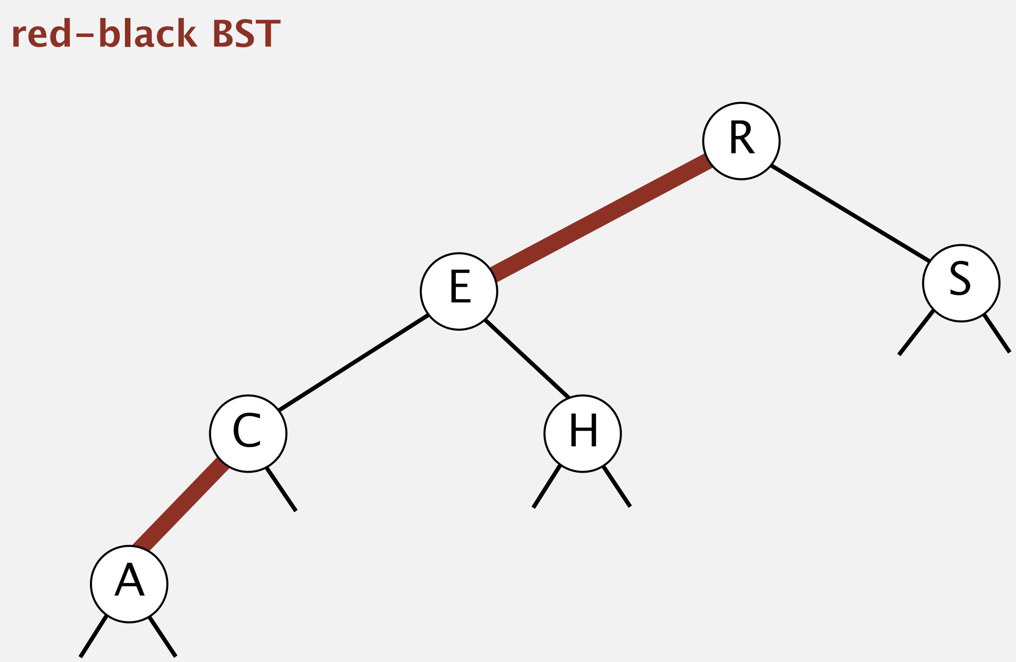
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

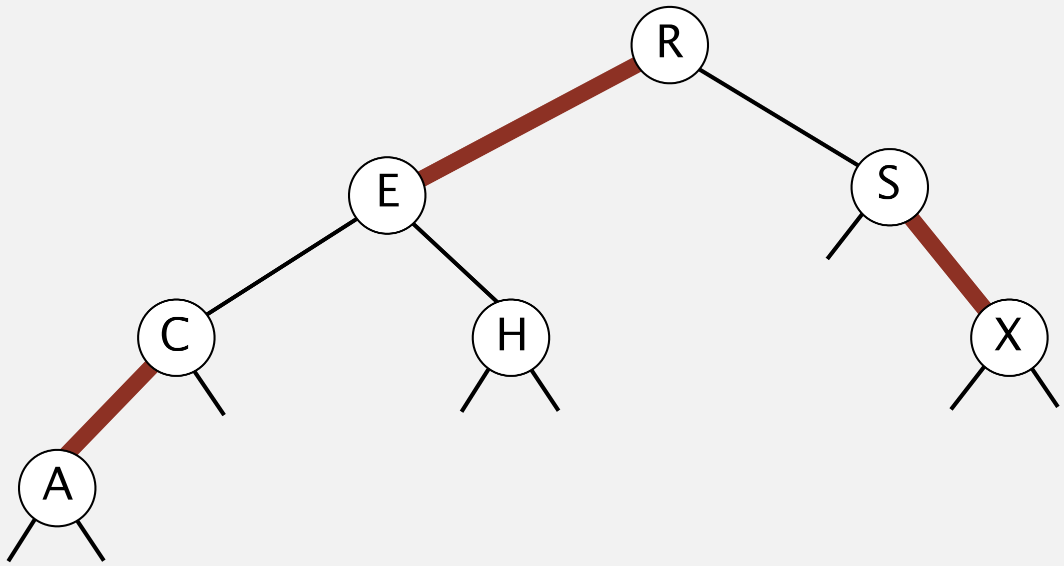
Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

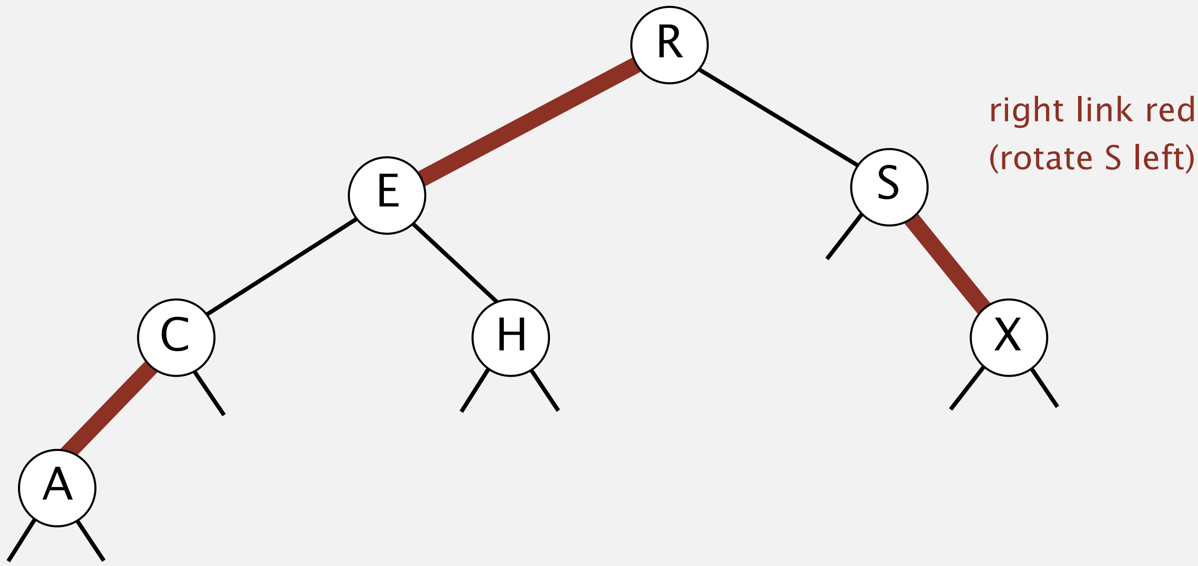
insert X



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

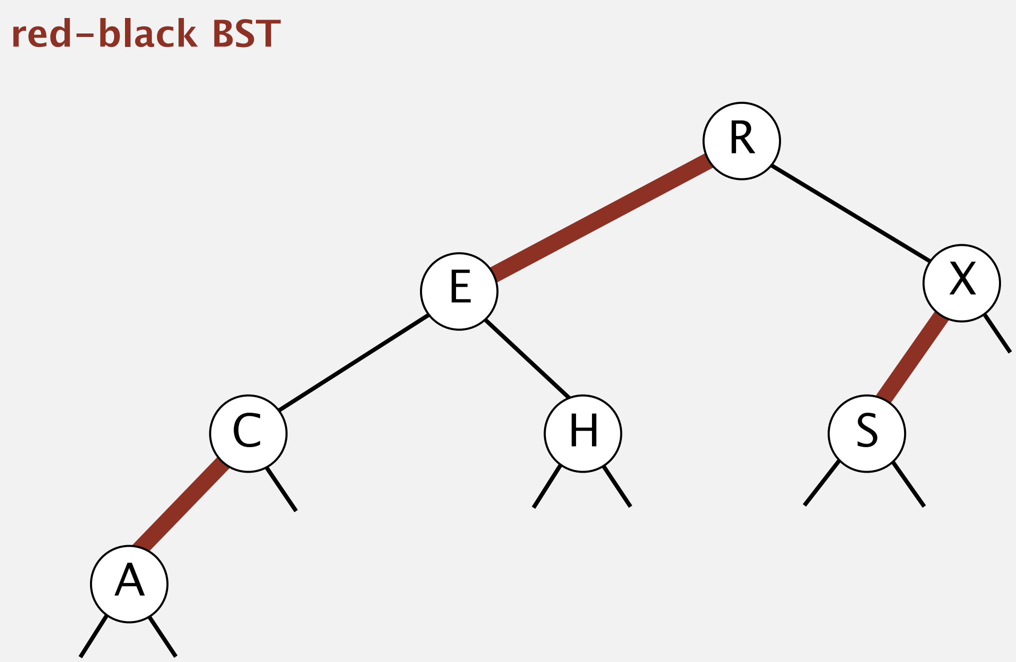
Red-black BST construction demo

insert X



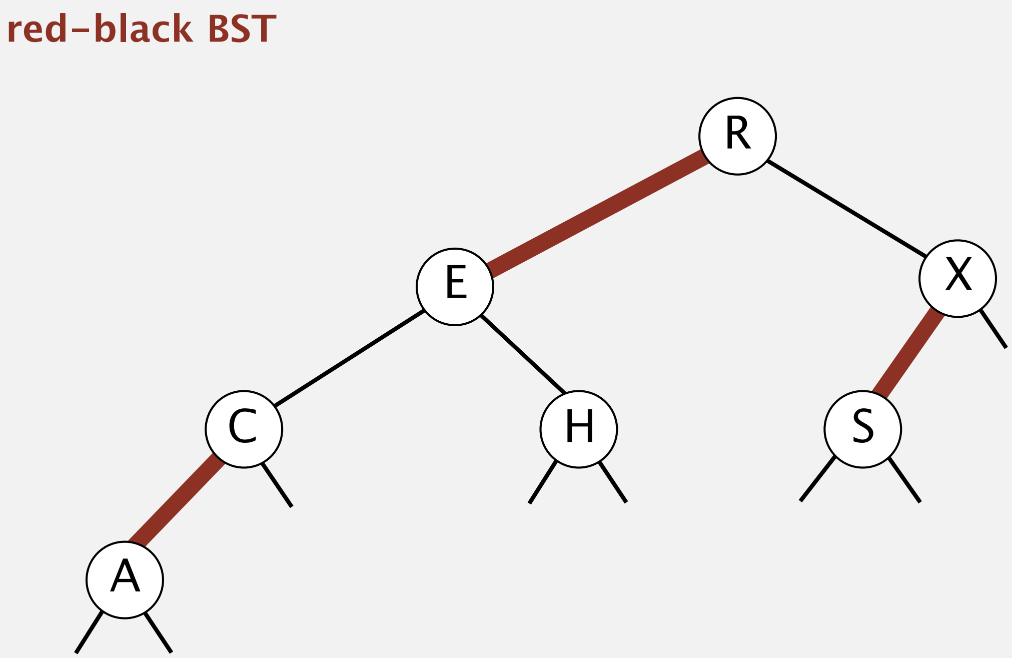
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



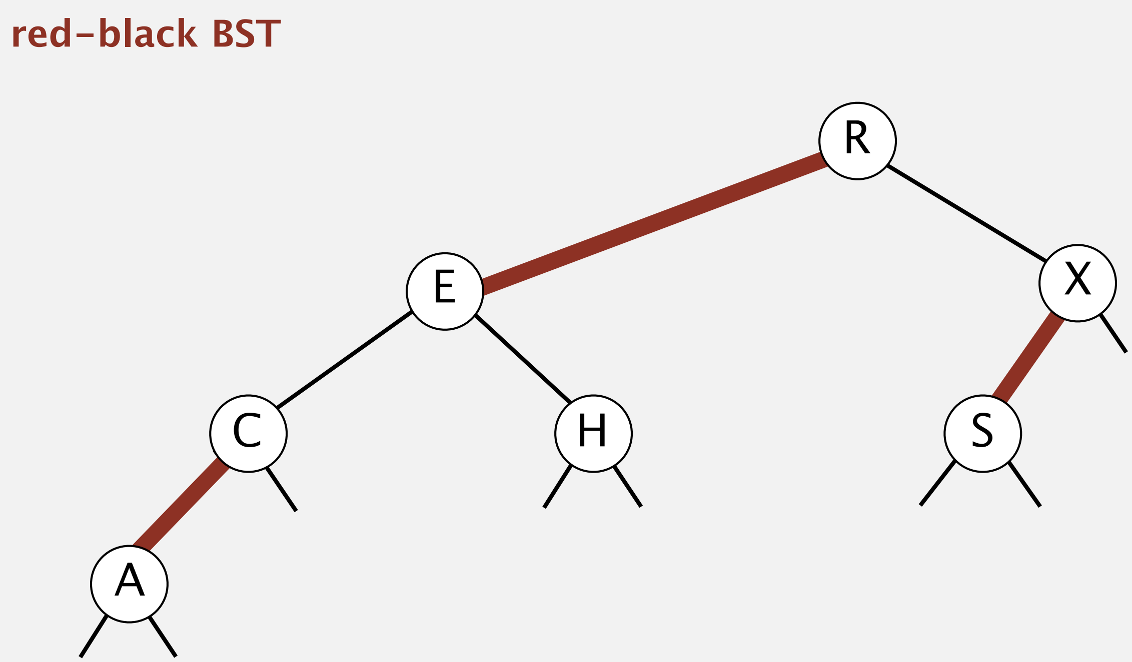
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

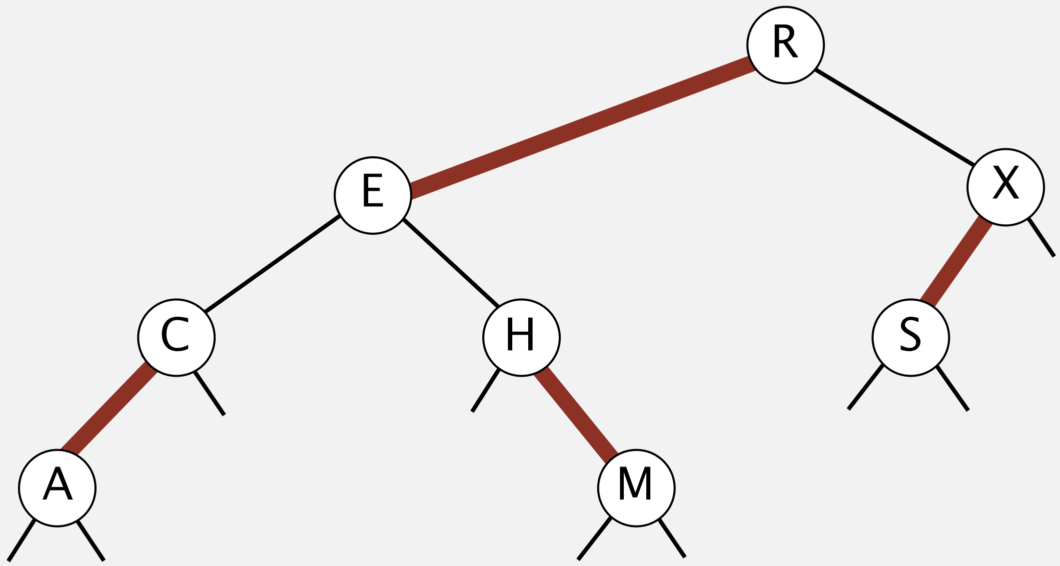
Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo

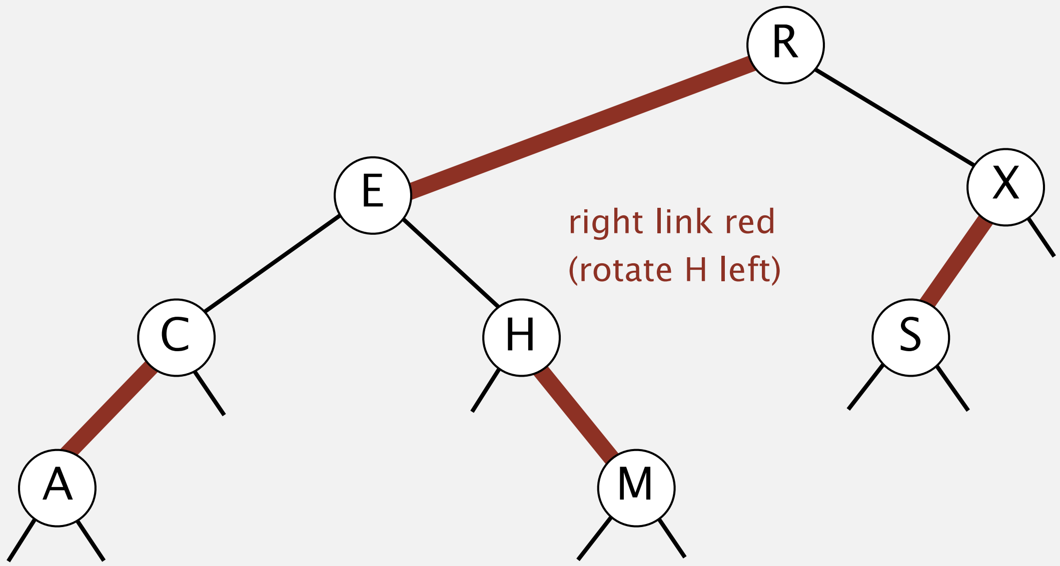
insert M



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

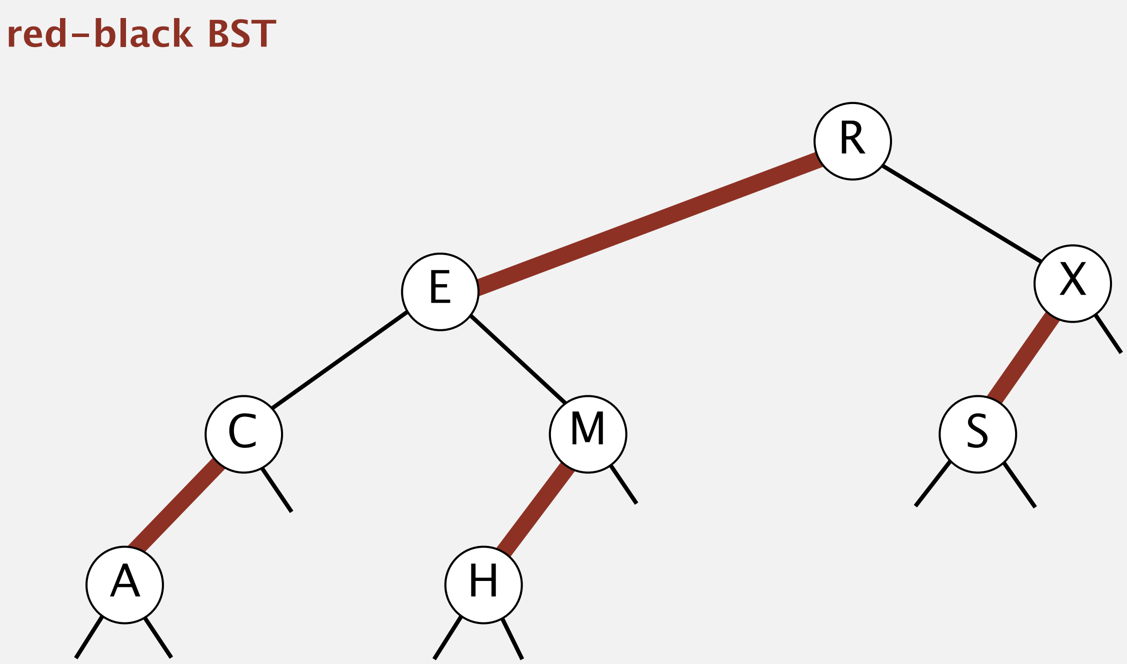
Red-black BST construction demo

insert M



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

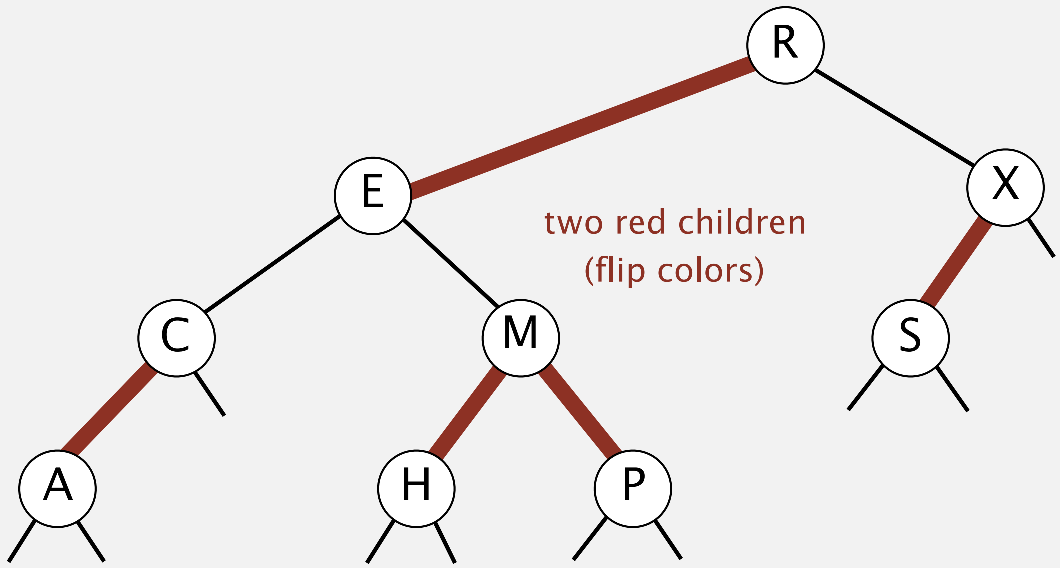
Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

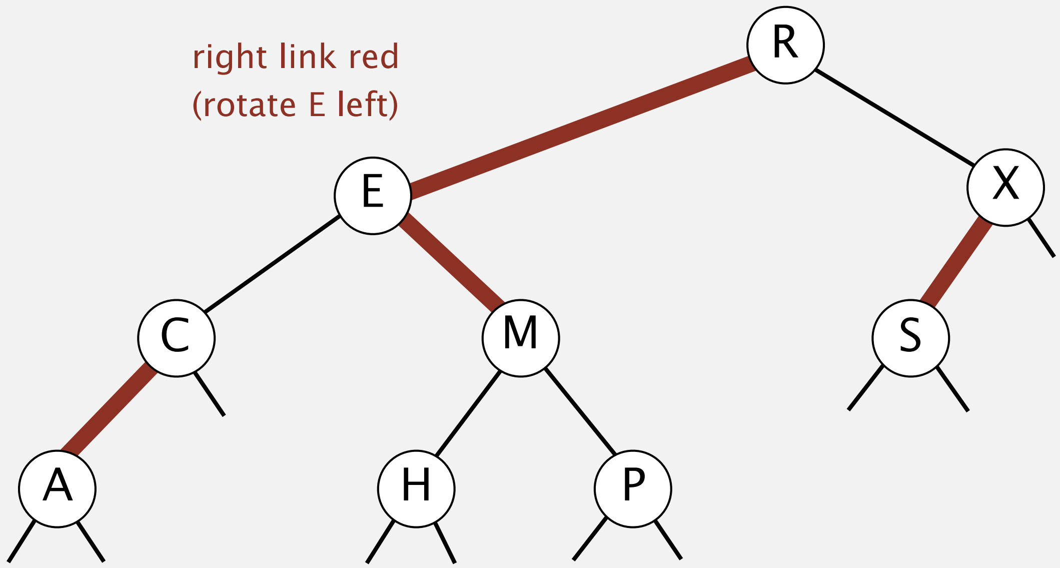
Red-black BST construction demo

insert P



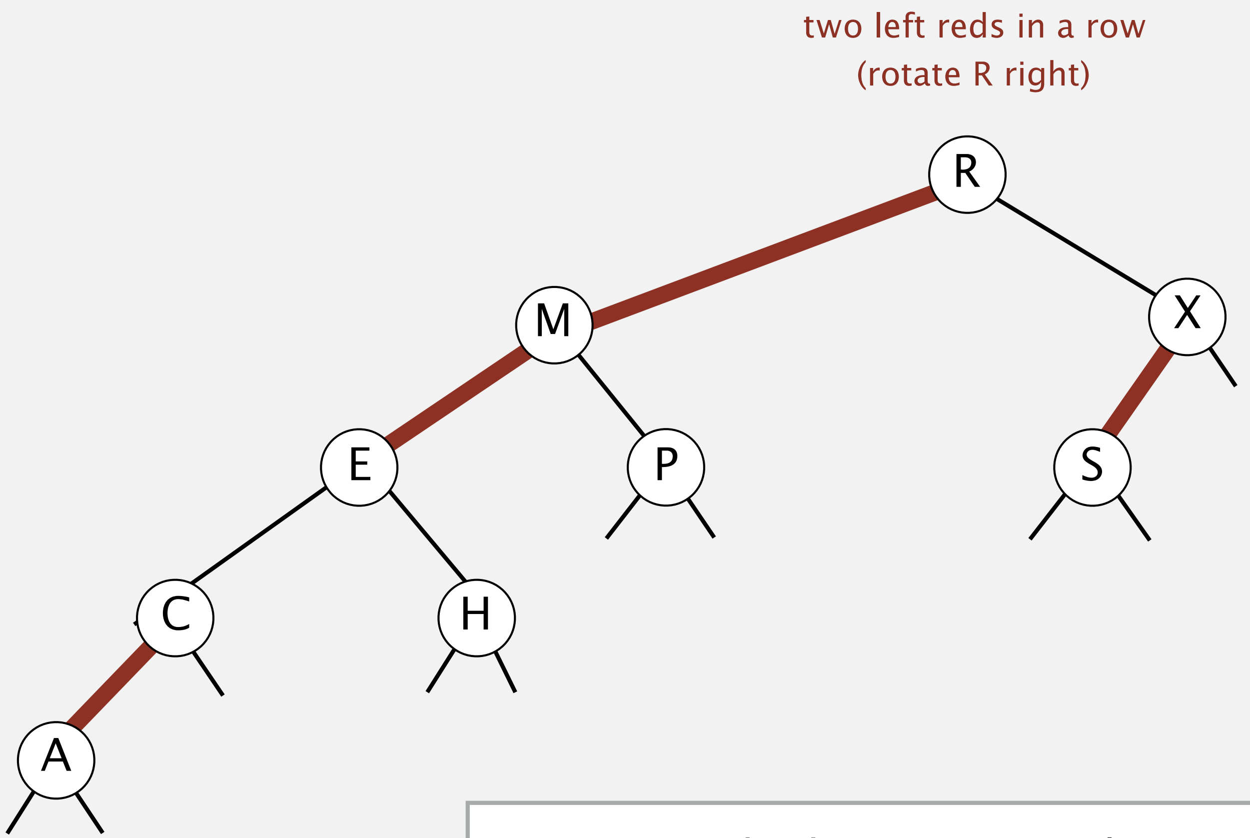
- Do standard BST insert; color new link red.
- Repeat until needed:
  – (Only) right link red: rotate left.
  – Two left reds in a row: rotate right.
  – Both children red: flip colors.

Red-black BST construction demo



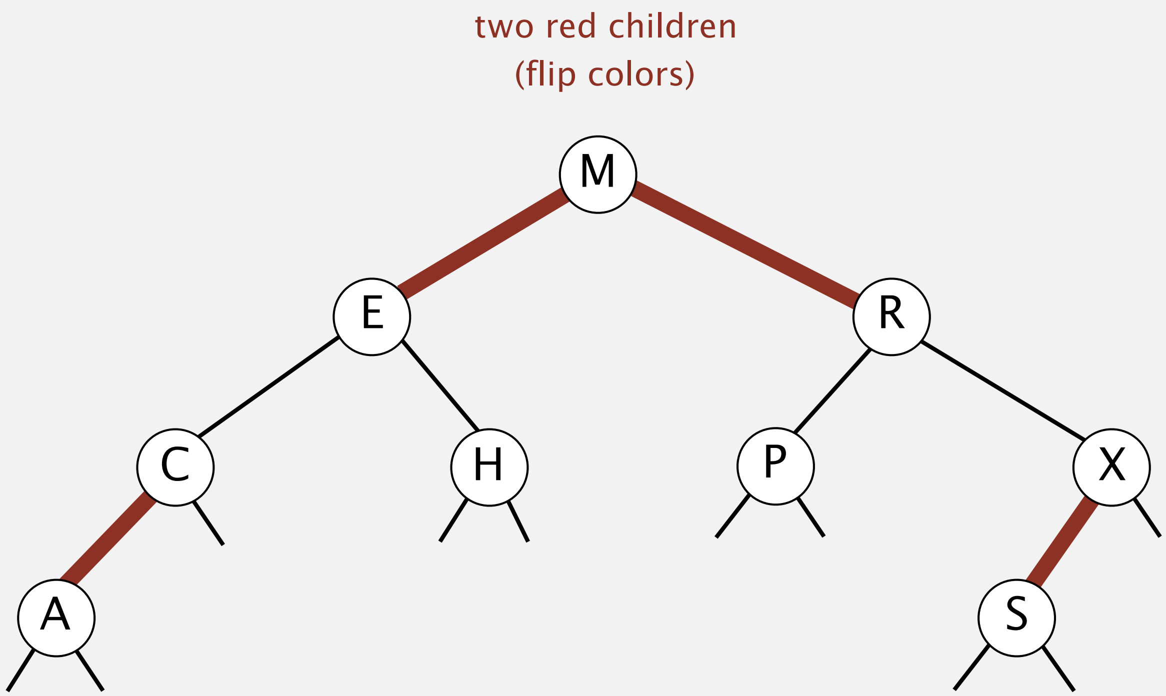
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



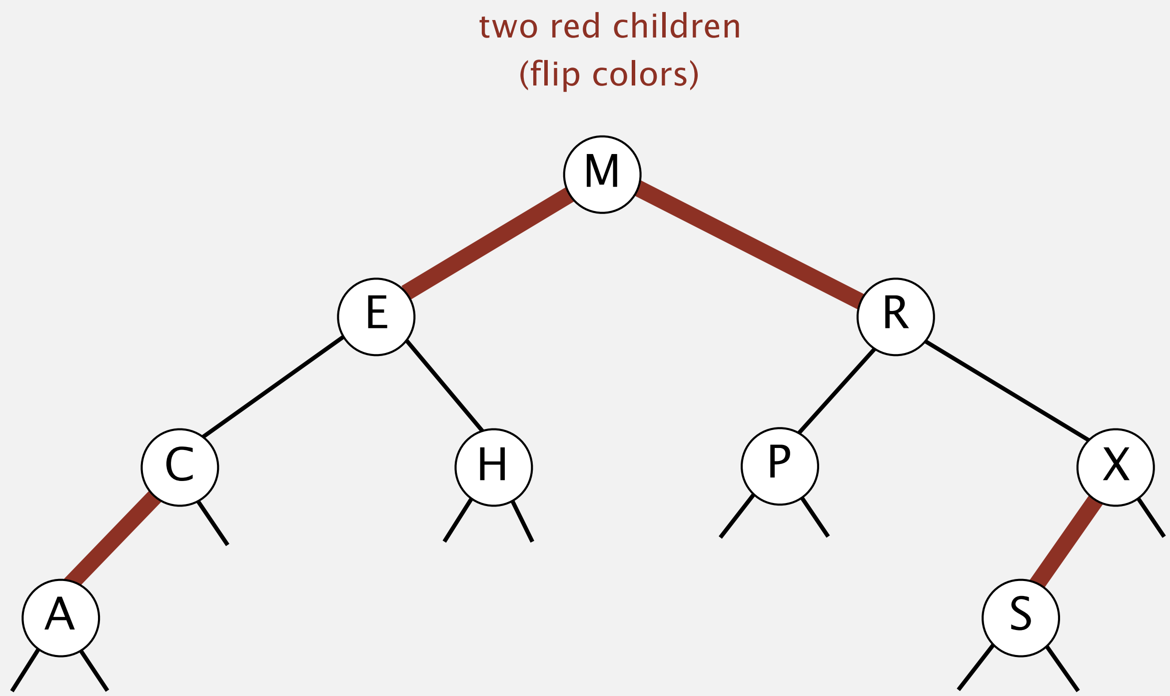
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



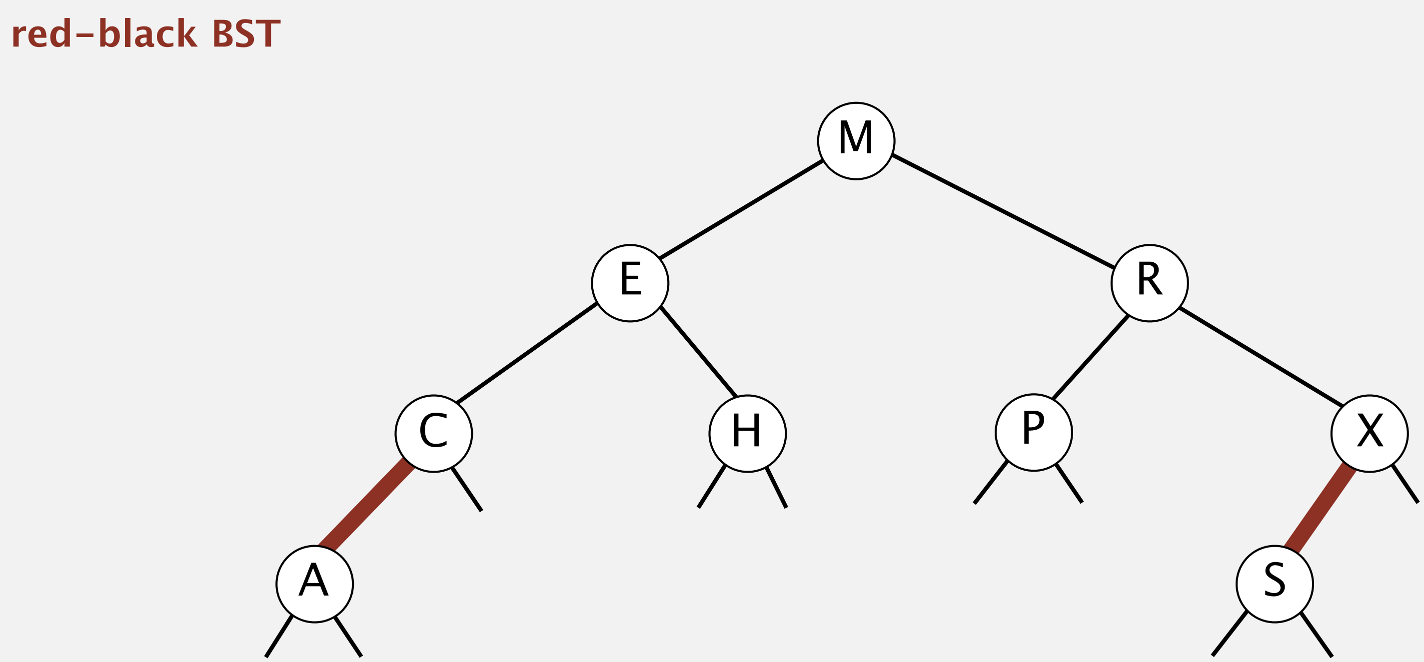
- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

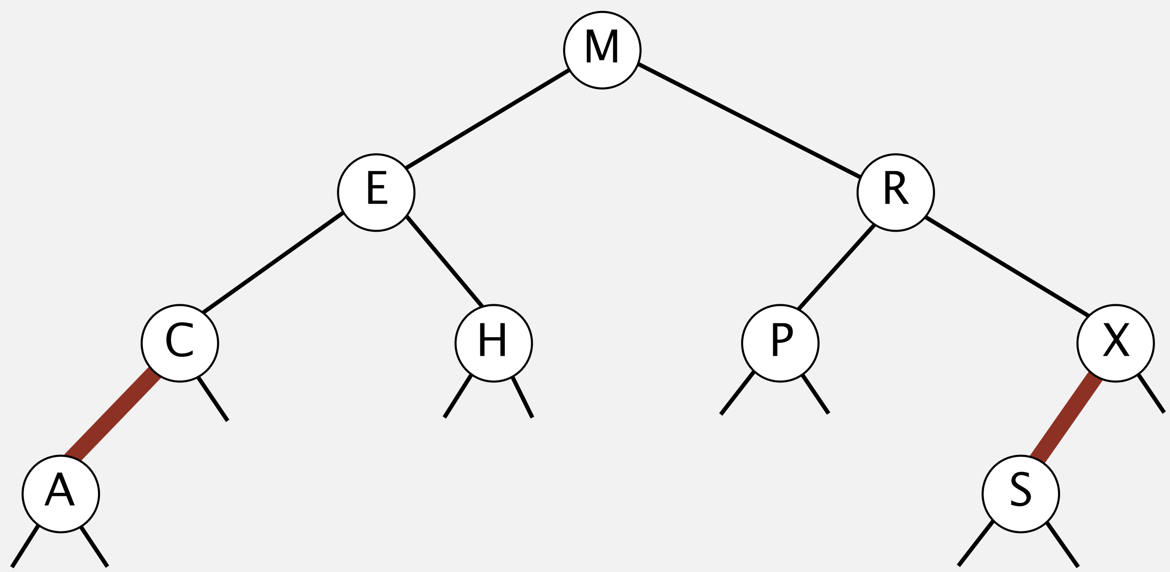
Red-black BST construction demo



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

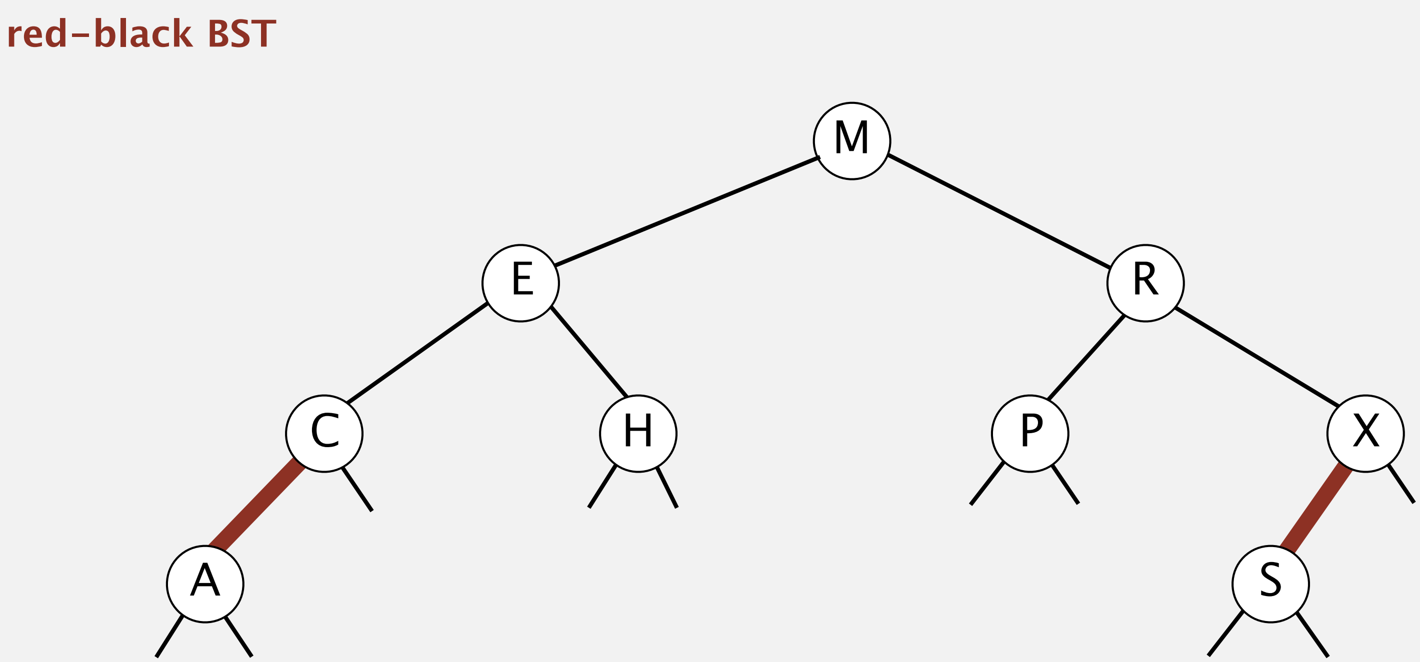
Red-black BST construction demo

red–black BST



- Do standard BST insert; color new link red.
- Repeat until needed:
  - (Only) right link red: rotate left.
  - Two left reds in a row: rotate right.
  - Both children red: flip colors.

Red-black BST construction demo



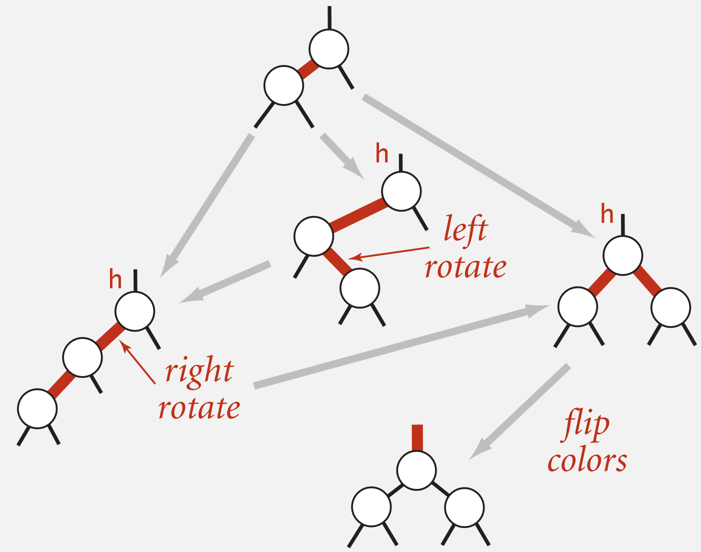
- Do standard BST insert; color new link red.
- Repeat until needed:
  – (Only) right link red: rotate left.
  – Two left reds in a row: rotate right.
  – Both children red: flip colors.

Insertion into 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.

Skipped in class




```
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)

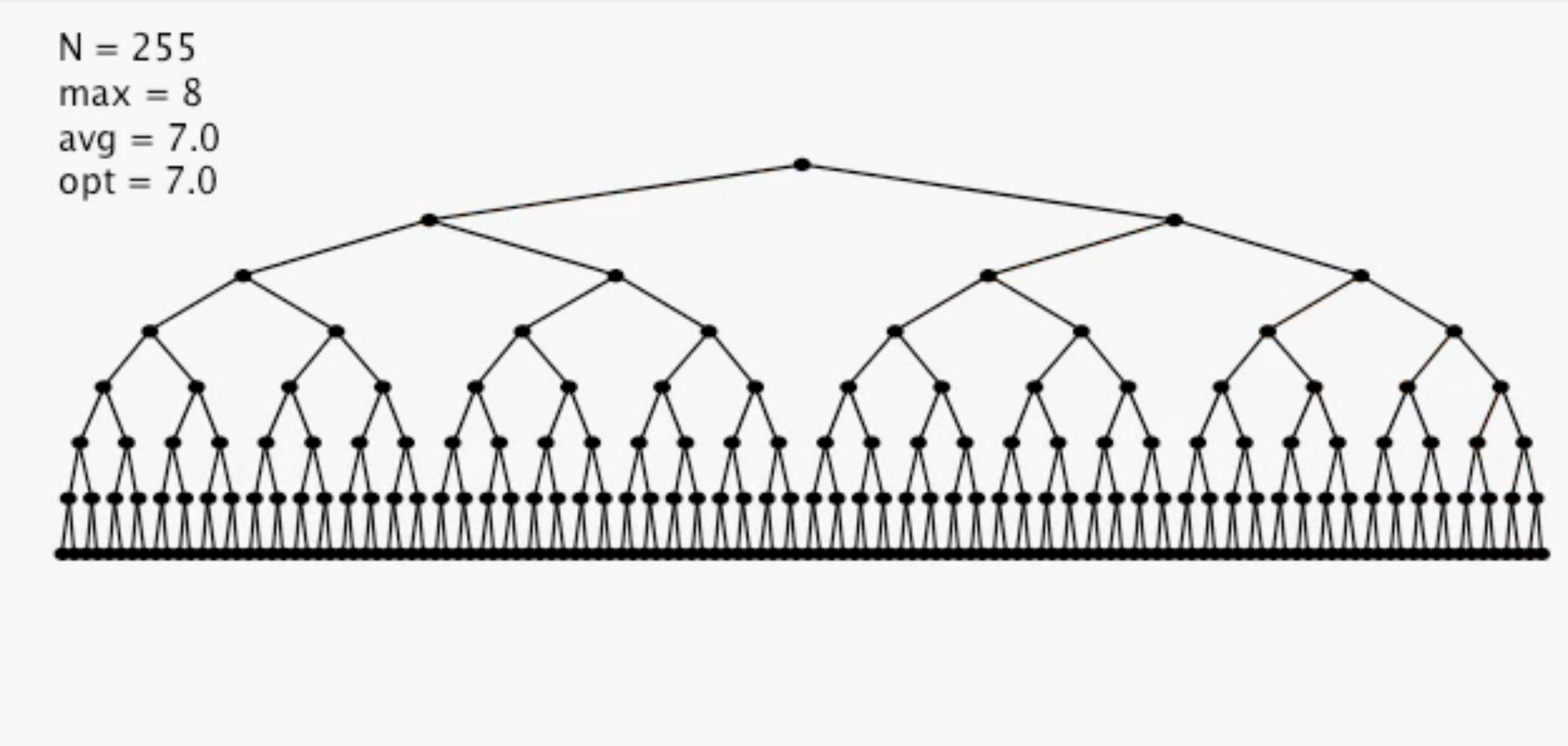
lean left

balance 4-node

split 4-node

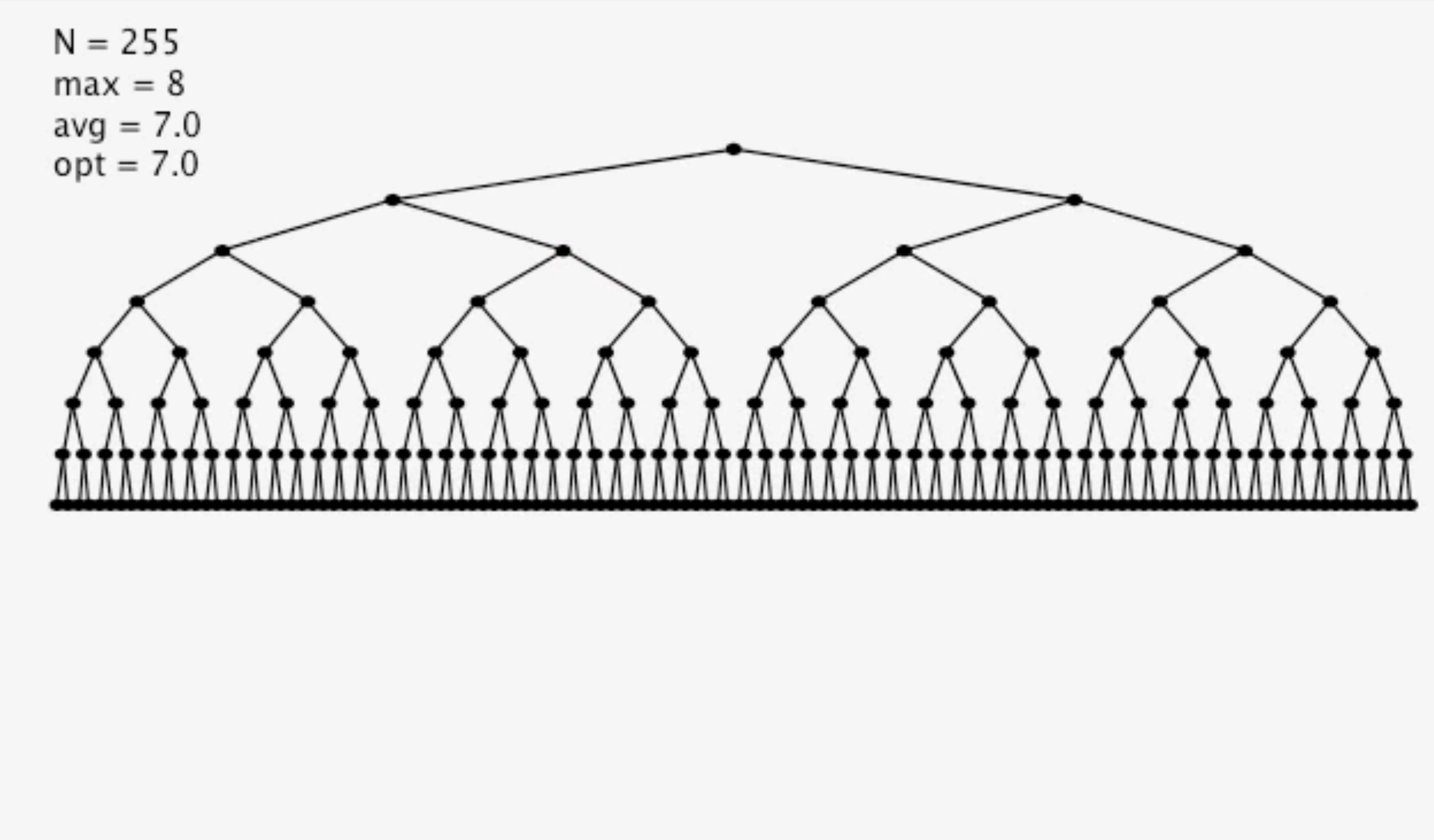
only a few extra lines of code provides near-perfect balance

Insertion into a LLRB tree: visualization



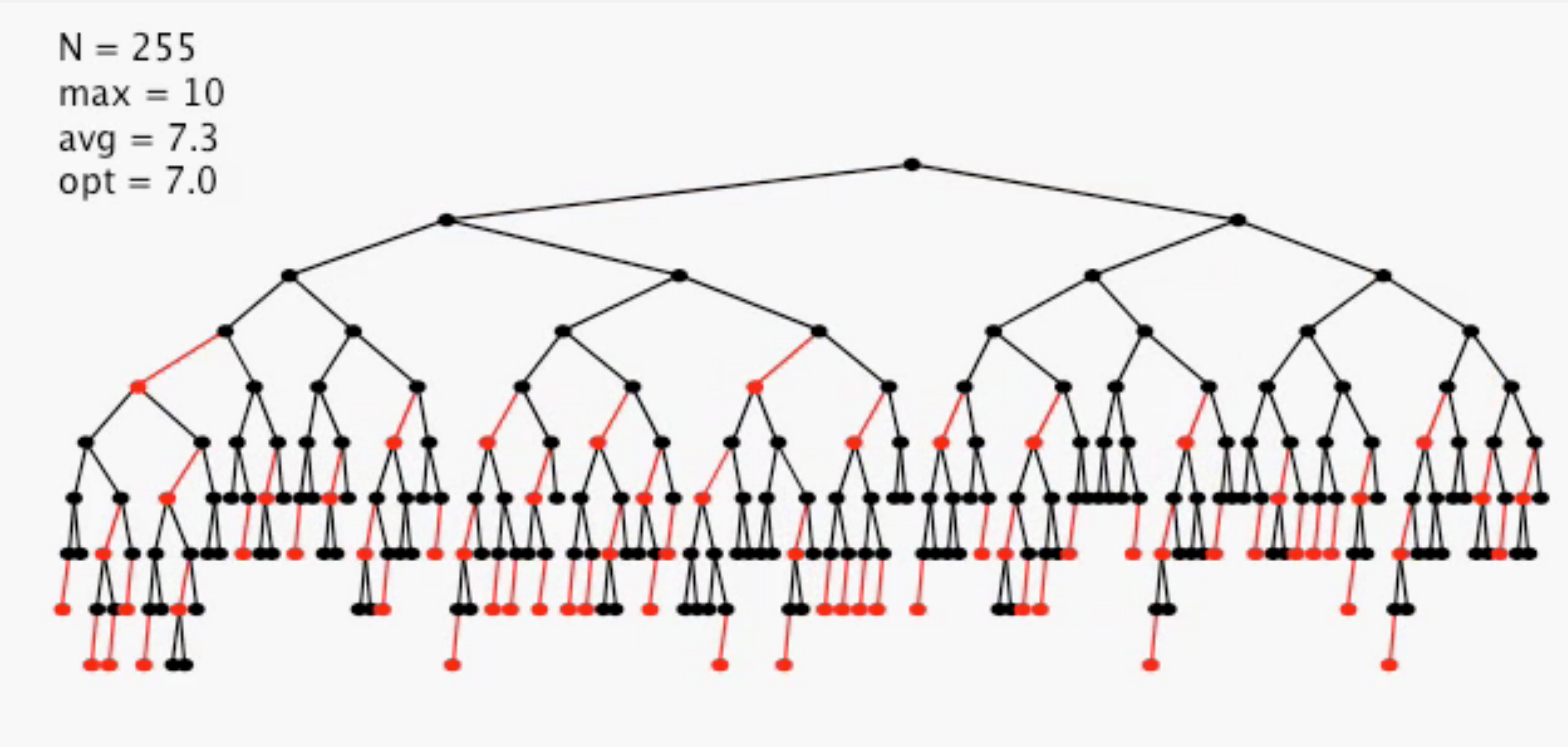
255 insertions in ascending order

Insertion into a LLRB tree: visualization



255 insertions in descending order

Insertion into a LLRB tree: visualization



255 random insertions

Balanced search trees: quiz 2

What is the height of a LLRB tree with N keys in the worst case?

A. $\sim \log_3 N$

B. $\sim \log_2 N$

C. $\sim 2 \log_2 N$

D. $\sim N$

E. *I don't know.*

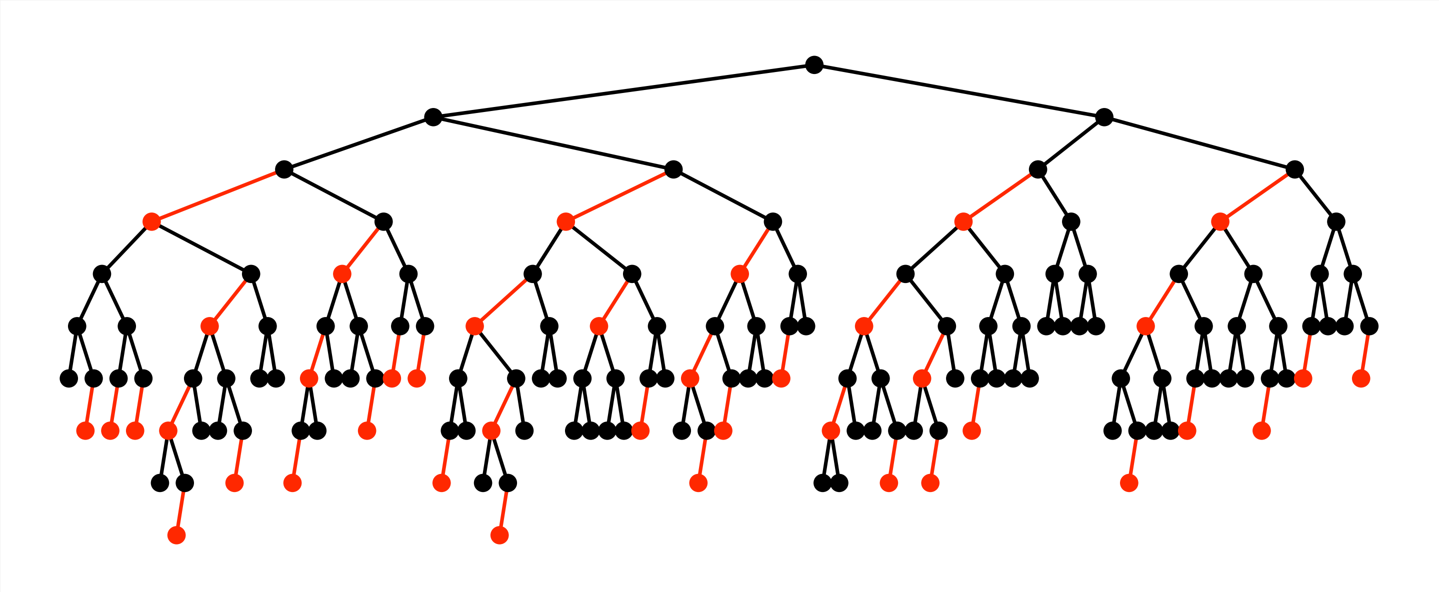
Ran out of time about here

Balance in LLRB trees

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

Pf.

- Black height = height of corresponding 2–3 tree $\leq \lg N$.
- 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	N	N	N		`equals()`
binary search (ordered array)	$\log N$	N	N	$\log N$	N	N	✔	`compareTo()`
BST	N	N	N	$\log N$	$\log N$	$\sqrt{N}$	✔	`compareTo()`
2-3 tree	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`
red-black BST	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`

hidden constant c is small
(at most 2 lg N compares)

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.
- Exceeding height limit of 80 triggered error-recovery process.

show allow for for up to 2^{40} keys

Extended telephone service outage.

did not rebalance BST during delete

- Main cause = height bound 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*


[redacted]

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

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

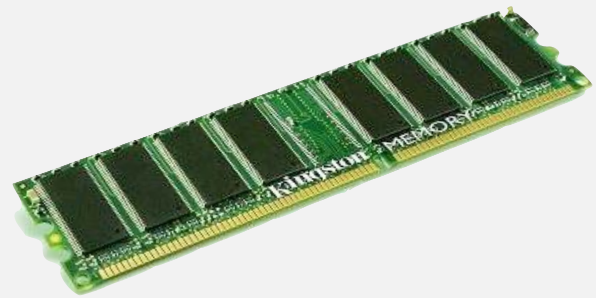
A type of **B**alanced tree (co-)invented by Rudolf **B**ayer while working at **B**oeing

File system model

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

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

slow



fast

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

Cost model. Number of probes.

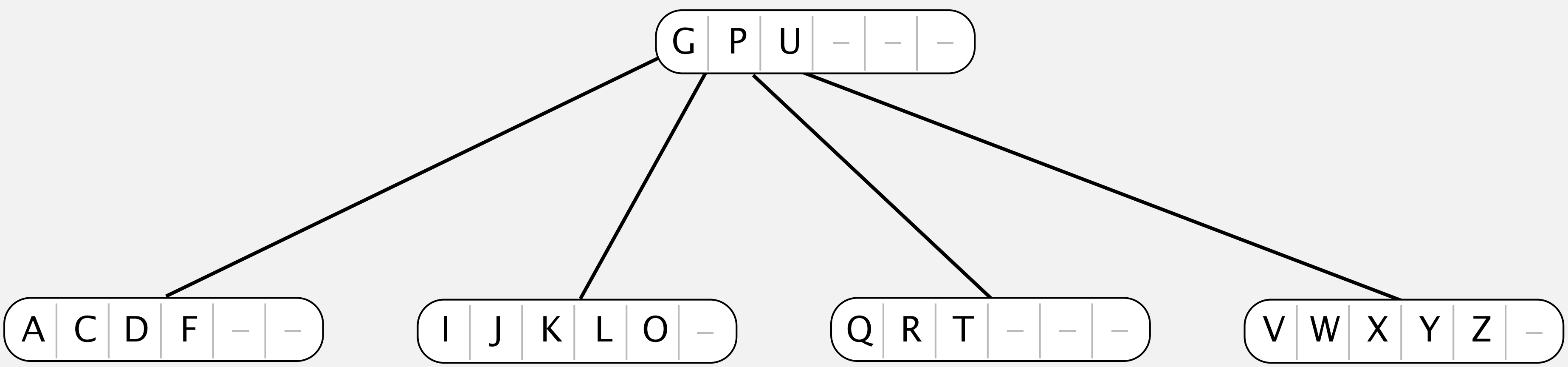
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2–3 trees by allowing up to M keys per node.

- At least $\lfloor M/2 \rfloor$ keys in all nodes (except root).
- Every path from root to leaf has same number of links.

choose M as large as possible so that M keys fit in a page (M = 1,024 is typical)

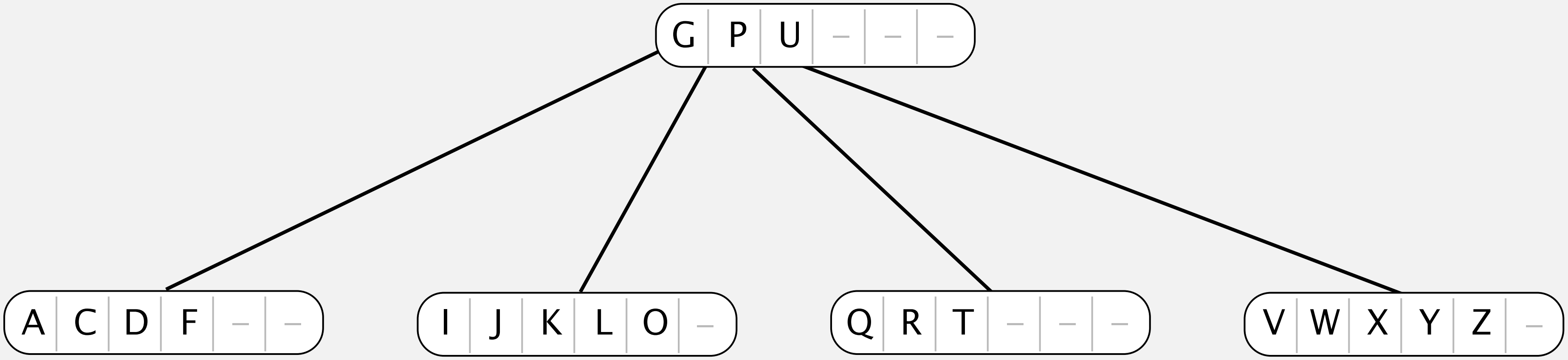


a B–tree (M = 6)

Search in a B-tree

- Start at root.
- Check if node contains key.
- Otherwise, find interval for search key and take corresponding link.

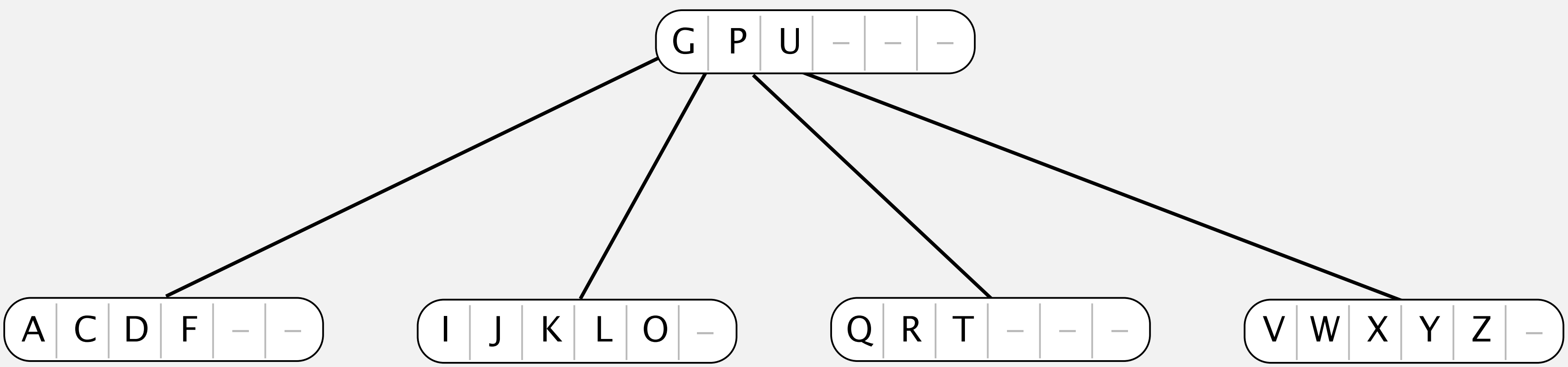
could use binary search
(but all ops are considered free)



a B-tree (M = 6)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with $M+1$ keys on the way back up the B-tree (moving middle key to parent).



a B–tree (M = 6)

Balance in B-tree

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

Pf. All nodes (except possibly root) have between $\lfloor M / 2 \rfloor$ and M keys.

In practice. Number of probes is at most 4.

M = 1024; N = 62 billion
$\log_{M/2} N \leq 4$

Balanced search trees: quiz 3

What of the following does the B in B-tree not mean?

A. Bayer

B. Balanced

C. Binary

D. Boeing

E. *I don't know.*

" the more you think about what the B in B-trees could mean, the more you learn about B-trees and that is good. "

– Rudolph Bayer

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 cousins. B+ tree, B*tree, B# tree, …

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

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