http://algs4.cs.princeton.edu

3.3 BALANCED SEARCH TREES

- *2-3 search trees*
- *red-black BST introduction*
- *red-black BST insert*
- *B-trees*

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

- *2-3 search trees*
- *red-black BST introduction*
- *red-black BST insert*
- *B-trees*

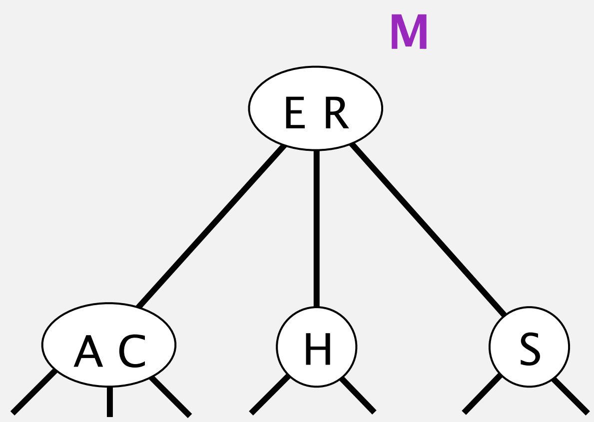
Symbol table review

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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.

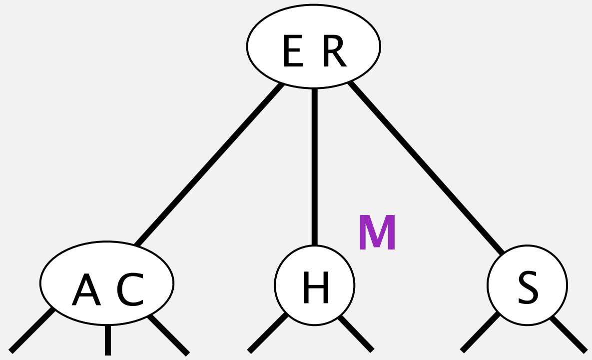
This lecture. 2-3 trees, left-leaning red-black BSTs, B-trees.

2-3 Trees (Turbo Edition)



Search for M

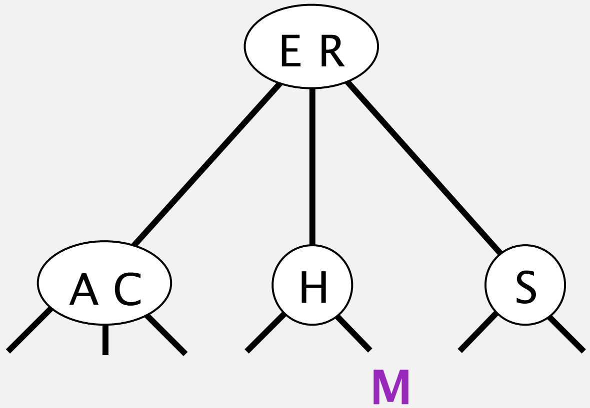
2-3 Trees (Turbo Edition)



Search for M

- Go middle

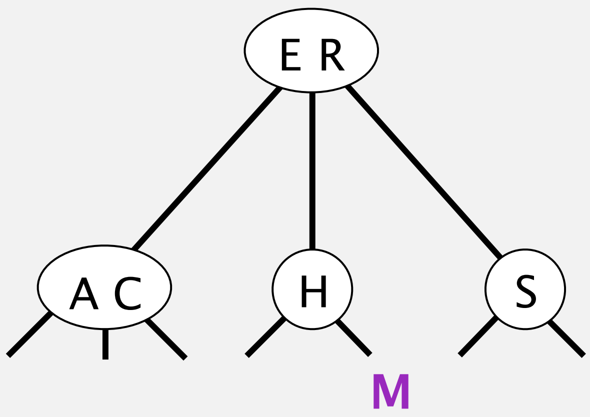
2-3 Trees (Turbo Edition)



Search for M

- Go middle
- Go right

2-3 Trees (Turbo Edition)



Search for M

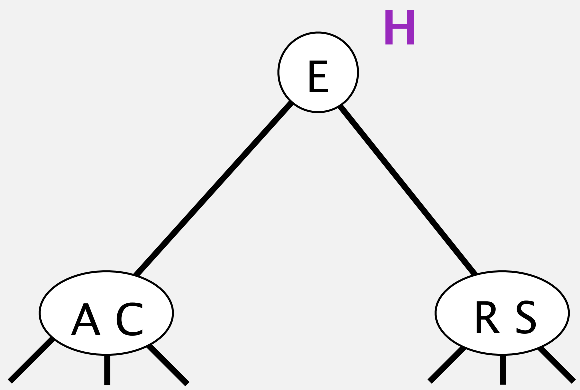
- Go middle
- Go right
  - null

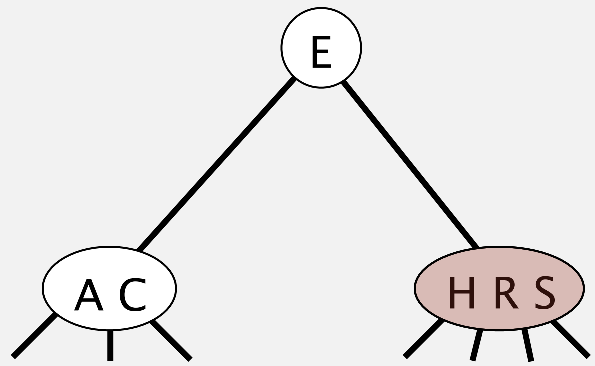
2-3 Trees (Turbo Edition)

Insert M (into 2-node)

- M is bigger than H, and H.right is null.
- M joins H.
  - **Important**: Never create new nodes at the bottom!

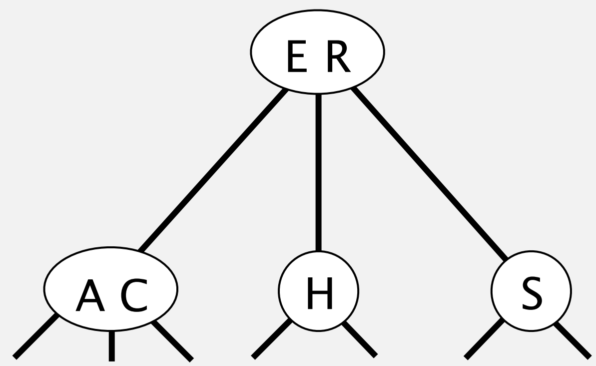
2-3 Trees (Turbo Edition)





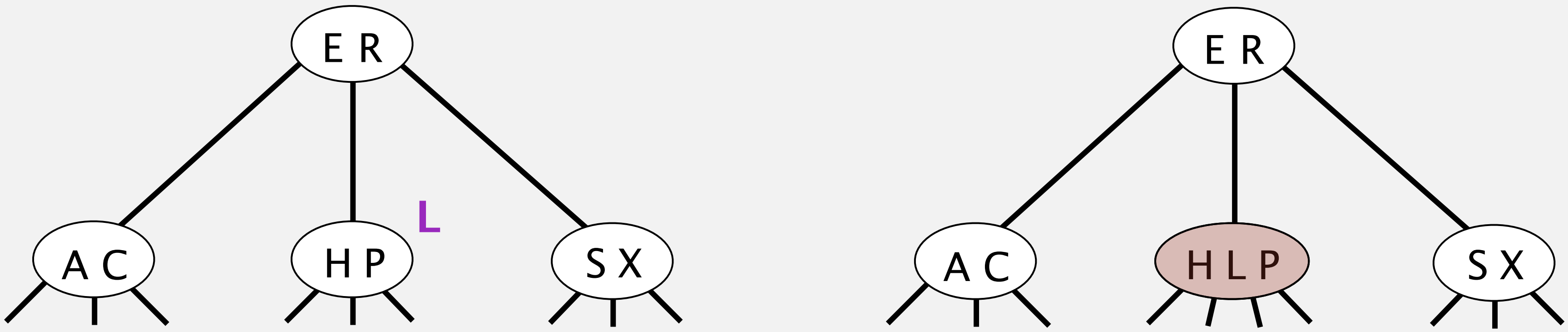
Insert H (into 3-node)

- H joins.
- [VIOLATION] 4 node created.
  - Send R to its parent.
  - Create two new 2-nodes from the debris.



- **Important**: Other than empty tree, only way to make new nodes.

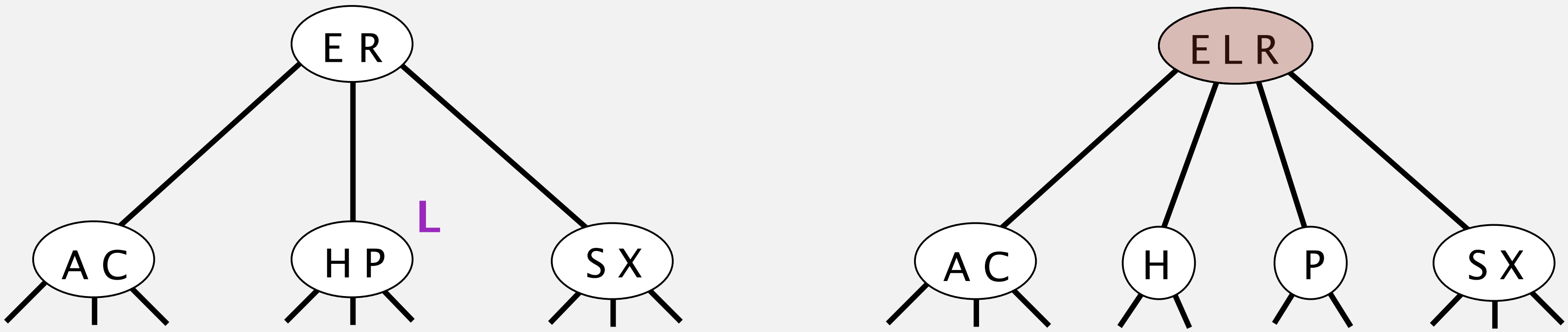
2-3 Trees (Turbo Edition)



Insert L (into 3-node with 3-node parent)

- [VIOLATION] HLP created.

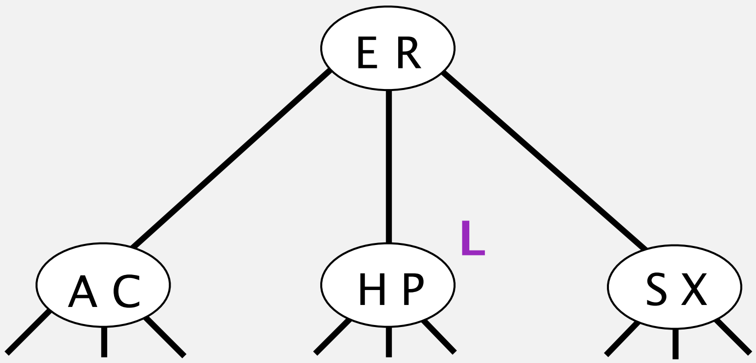
2-3 Trees (Turbo Edition)

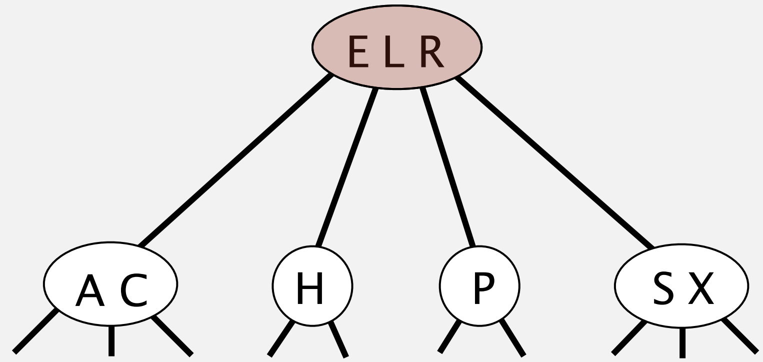


Insert L (into 3-node with 3-node parent)

- [VIOLATION] HLP created. Send L up, create H and P.
- [VIOLATION] ELR created.

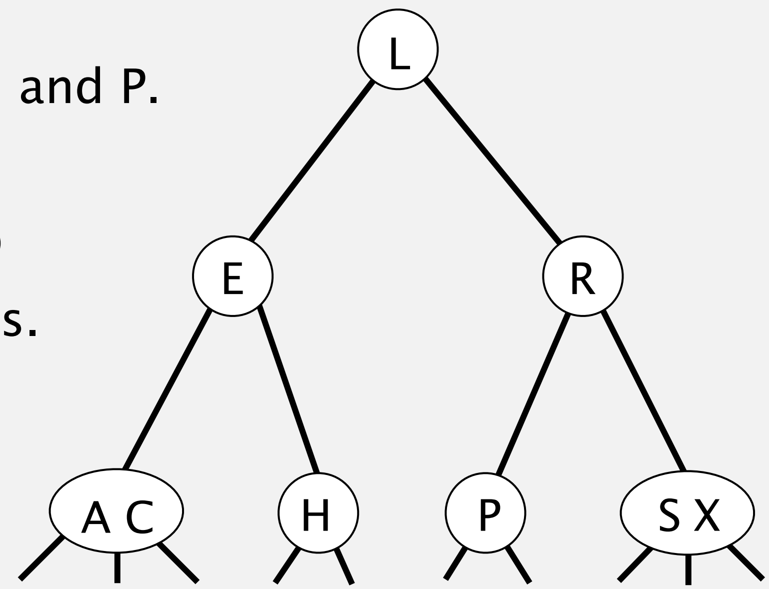
2-3 Trees (Turbo Edition)





Insert L (into 3-node with 3-node parent)

- [VIOLATION] HLP created. Send L up, create H and P.
- [VIOLATION] ELR created.
- Send L to join parent (no parent, so new root)
  - Create two new 2-nodes E-R from the debris.
  - Each gets custody of two nodes.



- **Important**: Only way to increase tree height is by splitting the root.

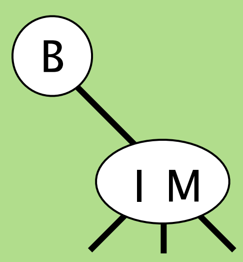
2-3 Tree Construction

Your turn.

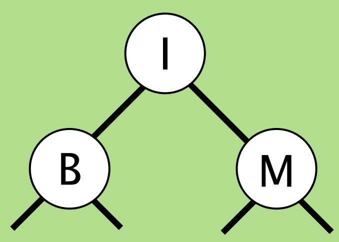
- Insert B, I, M. Which tree do you get?

pollEv.com/jhug text to **37607**

Which is the correct 2-3 tree?



[751394]



[751395]

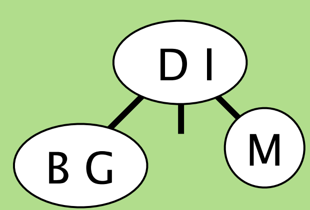
2-3 Tree Construction

One more.

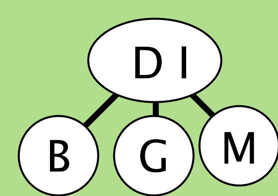
- Insert B, I, M, D, G. Which tree do you get?

pollEv.com/jhug text to **37607**

Which is the correct 2-3 tree?



[751422]



[751425]

Those Three Important Things Again

2-3 Tree

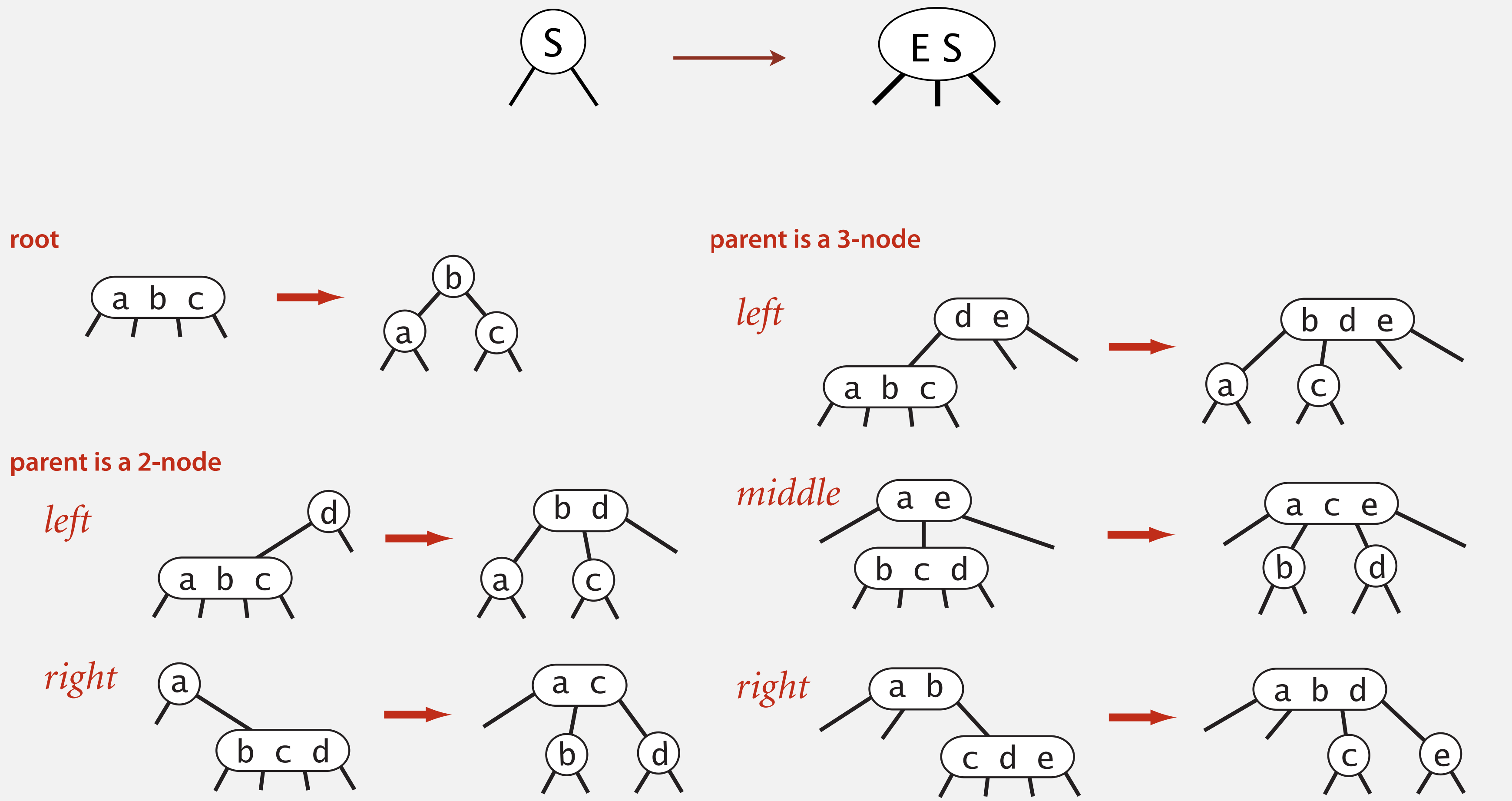
- Insert adds new keys into a leaf node instead of creating a new node at the bottom.
- New nodes only created when a 4-node is split.
- Height of tree only increases when root is split.

Stuff them til' they pop.

Global properties in a 2-3 tree

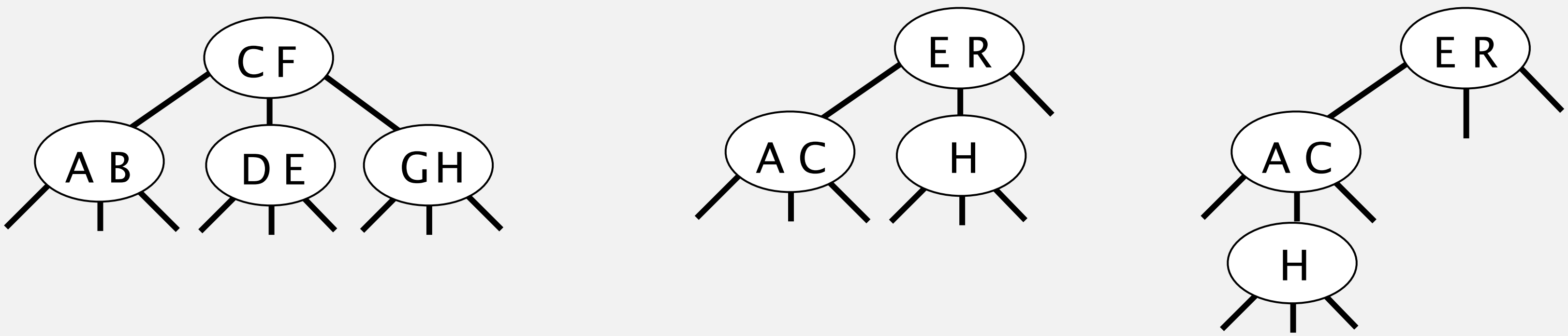
Invariants. Maintains symmetric order and perfect balance.

Pf. Adding a key to a leaf maintains symmetric order and perfect balance.
Splitting maintains symmetric order and perfect balance.



Group problems (groups of 3)

Which are valid 2-3 trees?



Are all 2-3 trees the same height for the same set of keys?

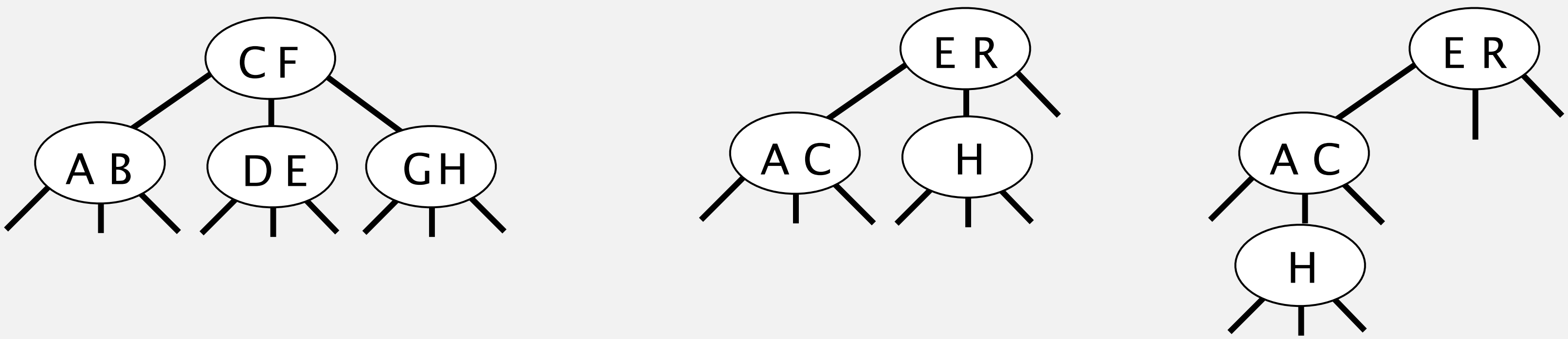
- If so: Why?
- If not: Give a counter example.

Bonus Questions

- Given N keys, describe a worst-case input sequence (greatest height).
- Given N keys, describe a best-case input sequence (smallest height).

Group problems (groups of 3)

Which are valid 2-3 trees?



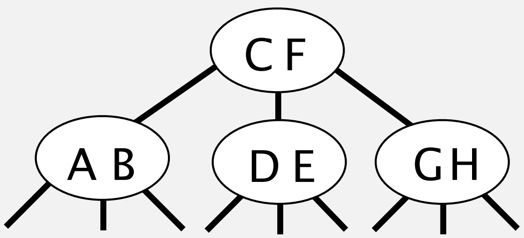
Perfect Balance

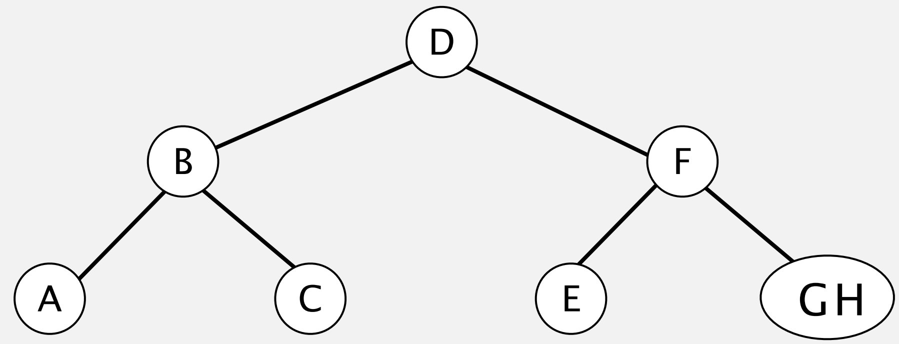
- Only leftmost tree achieves perfect balance.
- Perfect balance: Same number of nodes along every path from root to null.

Group problems (groups of 3)

Are all 2-3 trees the same height for the same set of keys?

- If so: Why?
- If not: Give a counter example.





Group problems (groups of 3)

Given N keys, describe a worst-case input sequence (greatest height).

- Worst case is all 2-nodes.
- Split as often as possible.
- Insert keys in ascending (or descending) order.

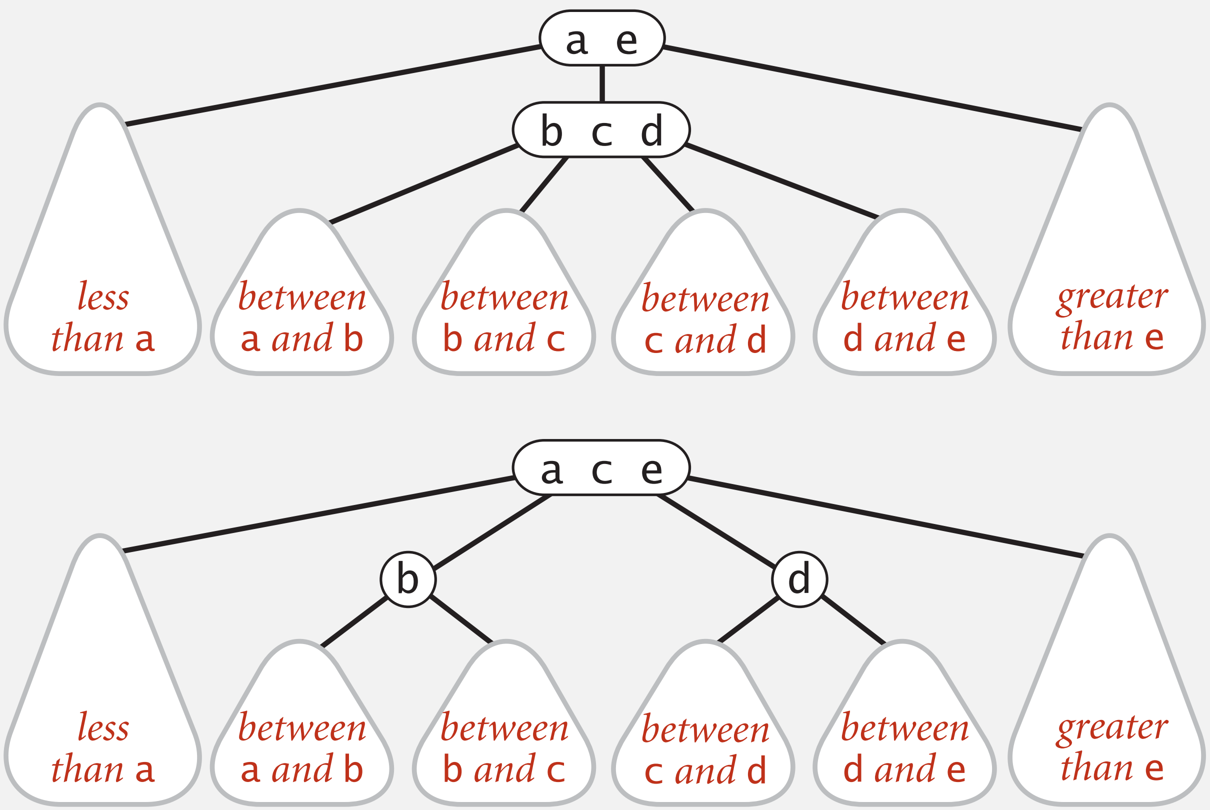
Given N keys, describe a best-case input sequence (smallest height).

- Best case is all 3-nodes.
- Split as infrequently as possible.
- ???

Performance: Local transformations in a 2-3 tree

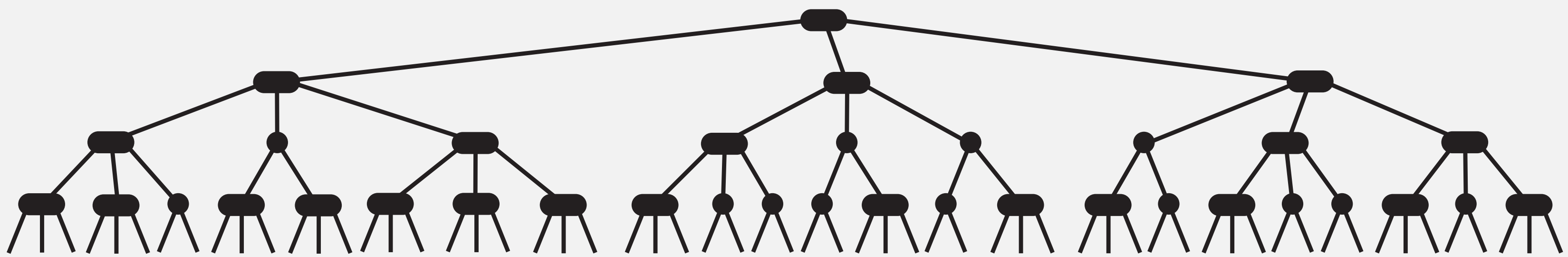
Splitting a 4-node is a local transformation: constant number of operations.

Bottom line: Splitting does not affect time complexity of insert.



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	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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()`

constants depend upon implementation

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

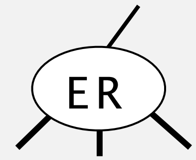
3.3 Balanced Search Trees

- 2-3 search trees
- red-black BST introduction
- red-black BST insert
- B-trees

The problem with 2-3 trees

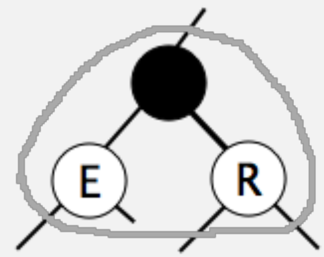
Hard to implement

- Multiple node types, 2-node, 3-node, 4-node
- Three children (leads to lots more cases)

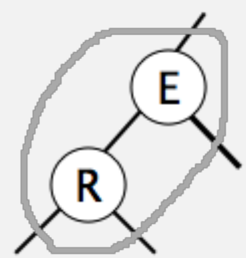


Goal: Represent as binary tree

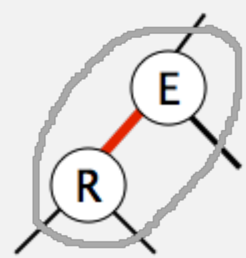
- Approach 1: Glue nodes.
  - Wasted space, wasted link.
  - Code probably messy.



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

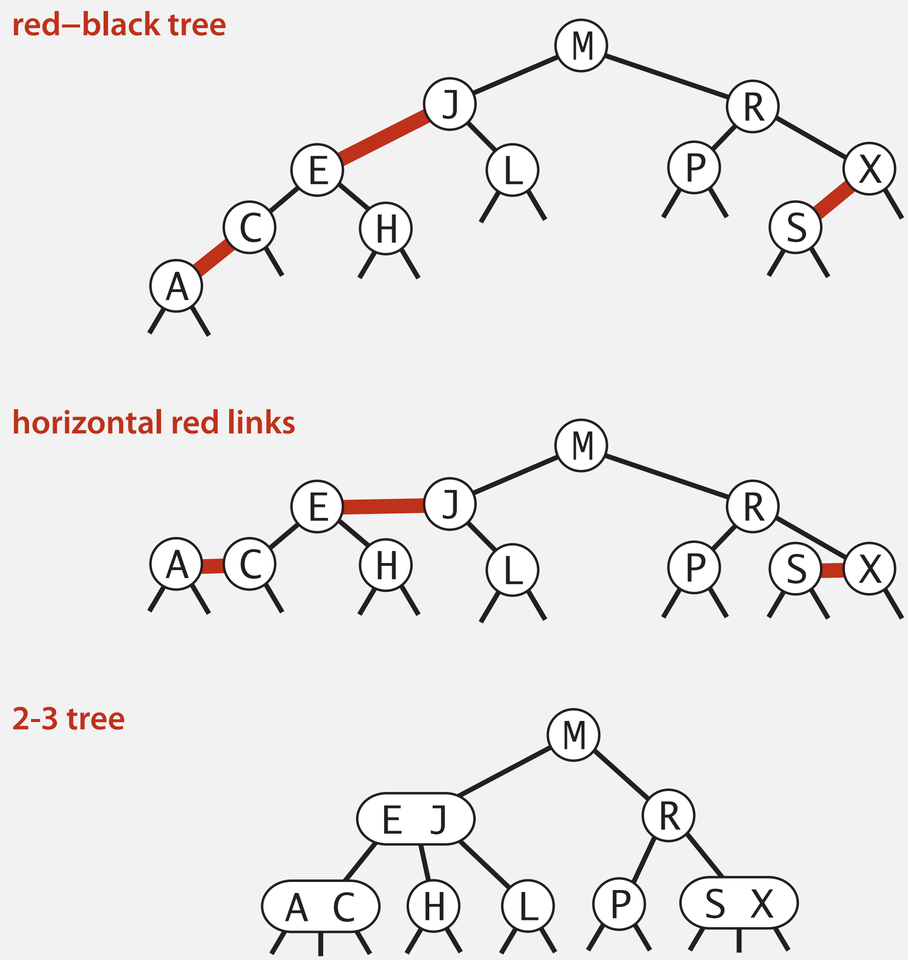


- Approach 3: BST with glue links.
  - Used widely in practice.
  - Arbitrary restriction: Red links lean left.



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

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



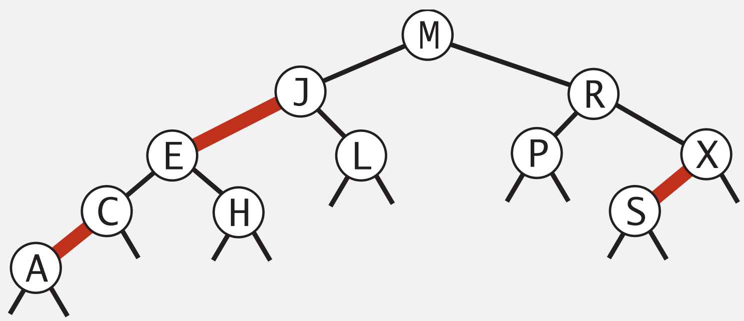
Not done. Need *search*, *insert*, and *delete* on an LLRB to mimic 2-3 trees.

Search implementation for red-black BSTs

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

but runs faster
because of better balance

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



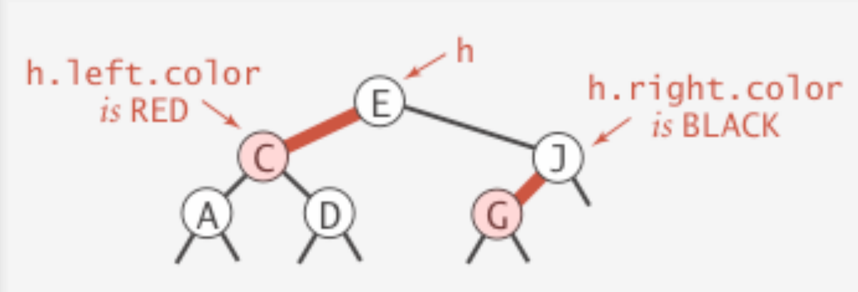
Remark. Most other ops (e.g., floor, iteration, selection) are also identical.

Red-black BST representation

Implementation Detail. Color is stored as a property of child node.

```
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



226 figures do not show node color!

Other possibilities for storing color.

As property of parent node (since all red links lean left).

As properties of links to children (two variables per parent).

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

- 2-3 search trees
- red-black BST introduction
- red-black BST insert
- B-trees

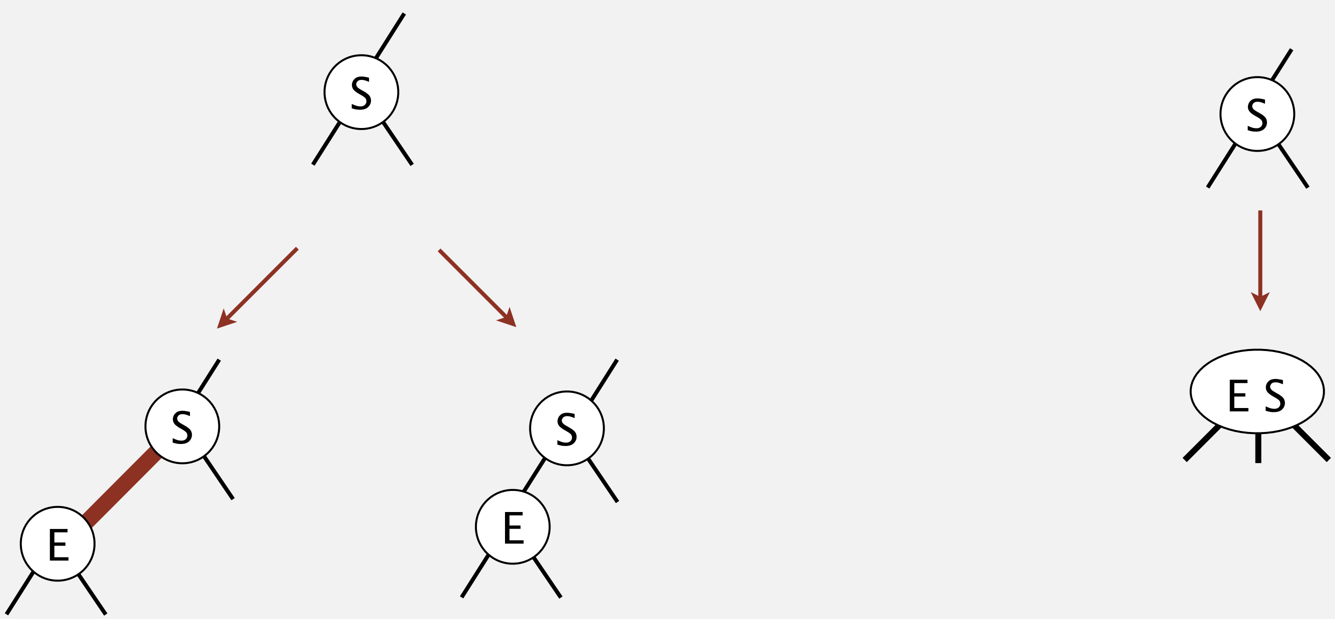
Easy Case 1: Inserting to the left of a 2-node

Should we use a red or a black link in this case?

- Red link.

What about in other cases?

- Red link.
  - Never create new nodes in a 2-3 tree except when splitting a 4 node.
  - Every path to null must have the same number of black links.



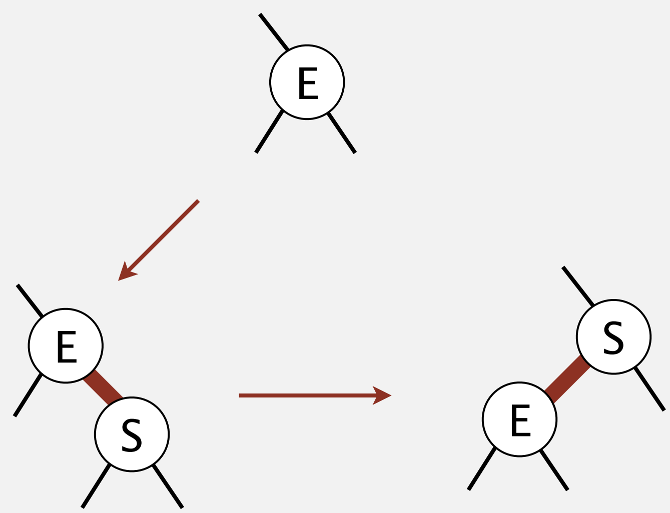
Easy Case 2: Inserting to the right of a 2-node

What is the problem here?

- Red links must lean left (by definition)

How do we fix the problem?

- Swap roles of S and E
  - Can generalize role-swapping for non-leaf nodes as *left rotation*.



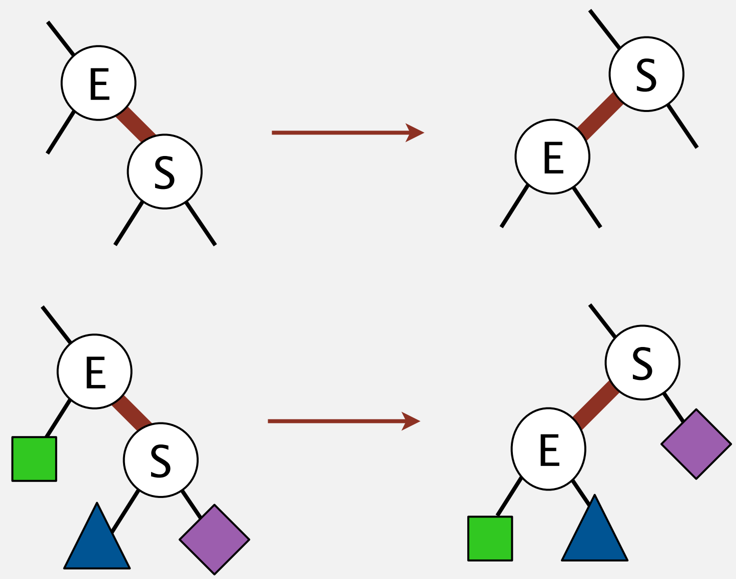
Easy Case 2: Inserting to the right of a 2-node

What is the problem here?

- Red links must lean left (by definition)

How do we fix the problem?

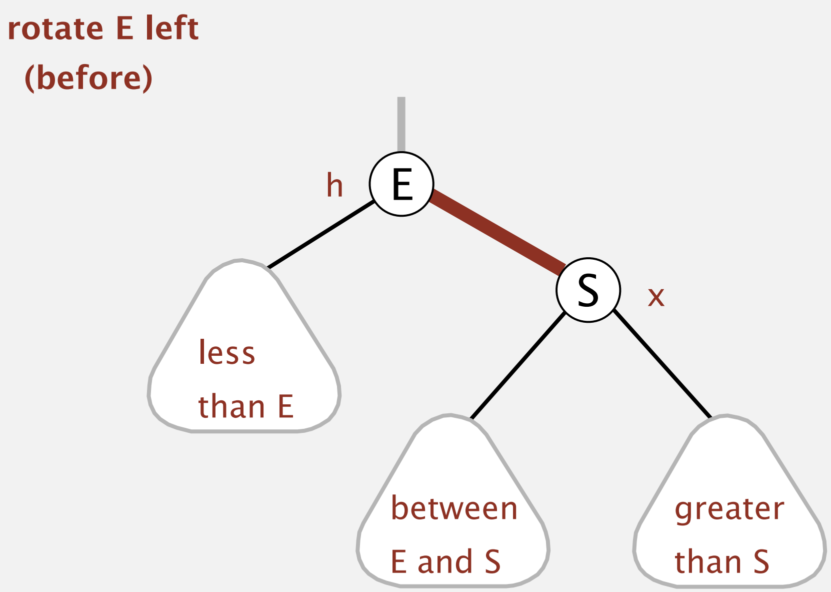
- Swap roles of S and E
  - Can generalize role-swapping for non-leaf nodes as *left rotation*.
  - Usefulness of rotation will become clear.



Elementary red-black BST operations

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

Rotate E left: Promote E's right child in the only sensible way.



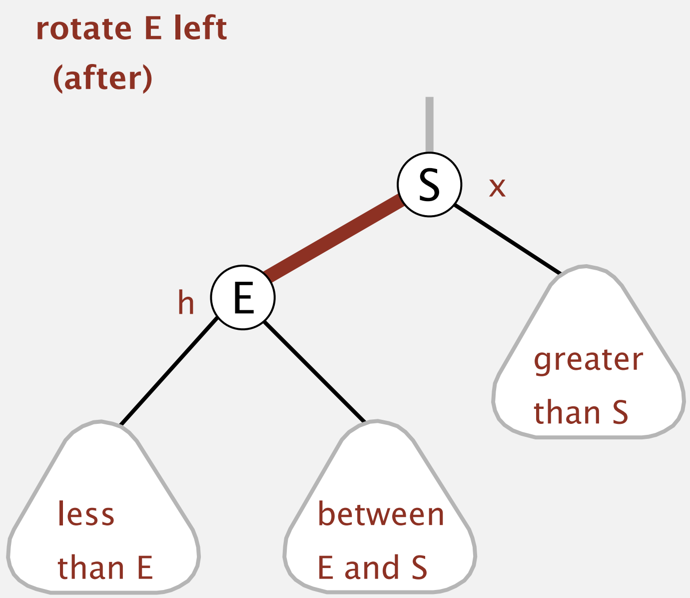

```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

Rotate E left: Promote E's right child in the only sensible way.



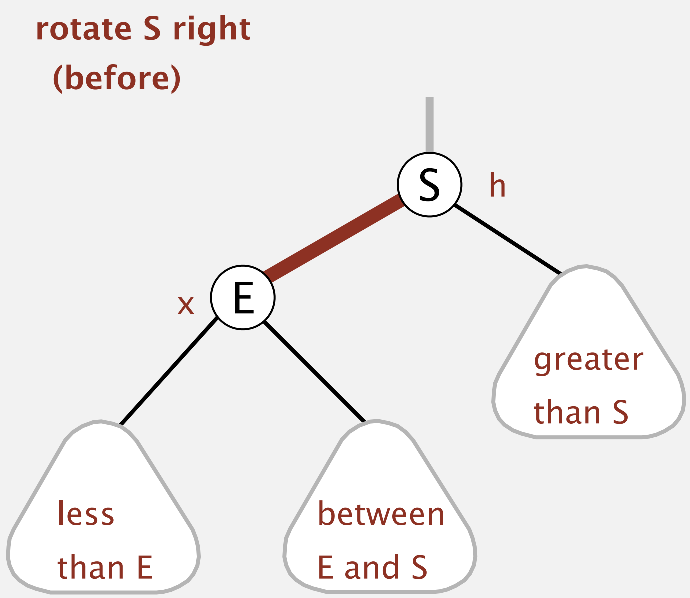

```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

Rotate S right: Promote S's left child in the only sensible way.



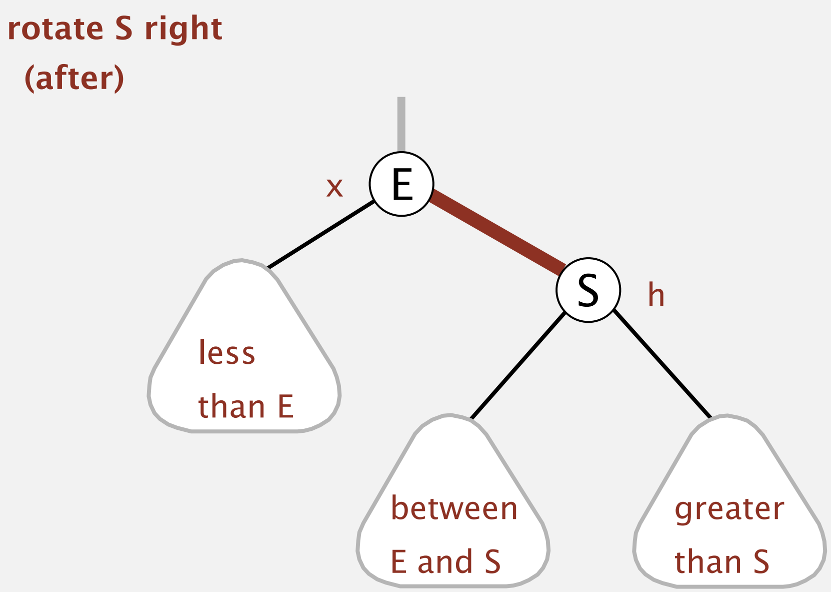

```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

Rotate S right: Promote S's left child in the only sensible way.



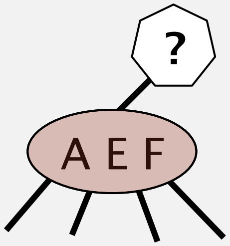

```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

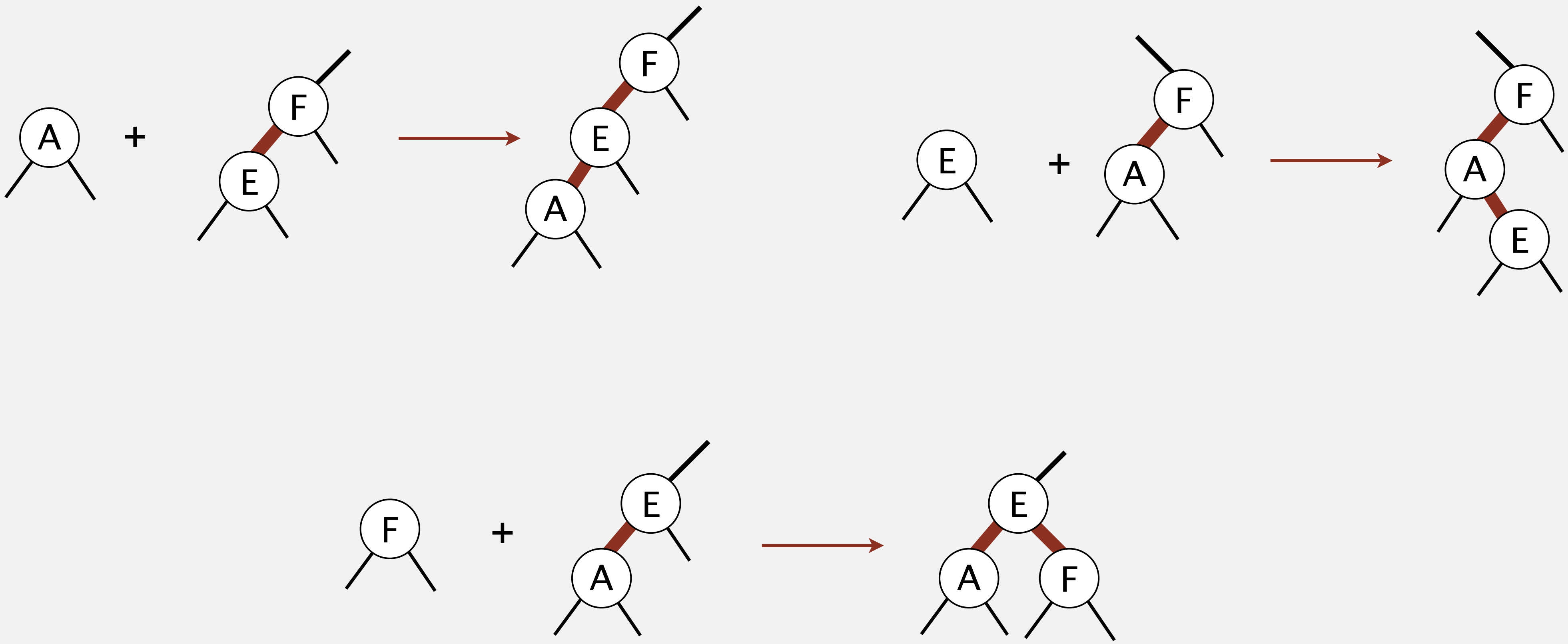
Invariants. Maintains symmetric order and perfect black balance.

Mimicking 2-3 Trees

Three problem cases when inserting into 3-node.

- Two consecutive red left children.
- Red right child.
- Two red children (easier to think of as separate case).





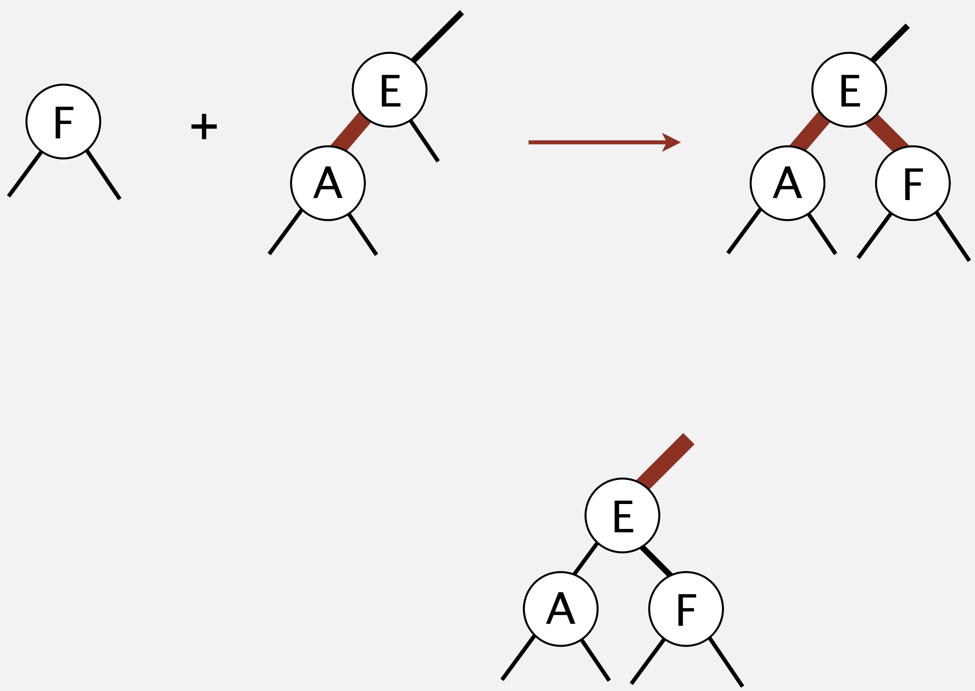
Case 1: Two red children

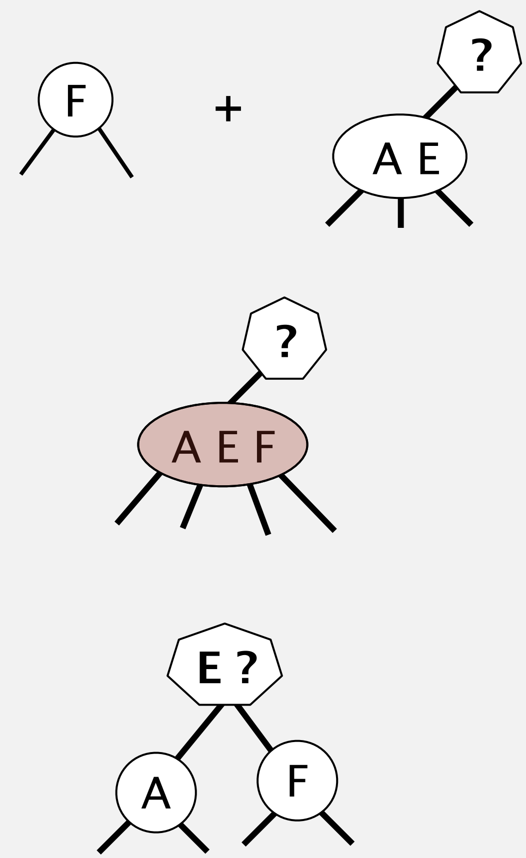
What is the problem here?

- Red links must lean left (by definition).
- [VIOLATION] 4 node.

How to Resolve?

- Color flip.
- Equivalent to splitting 4 node.





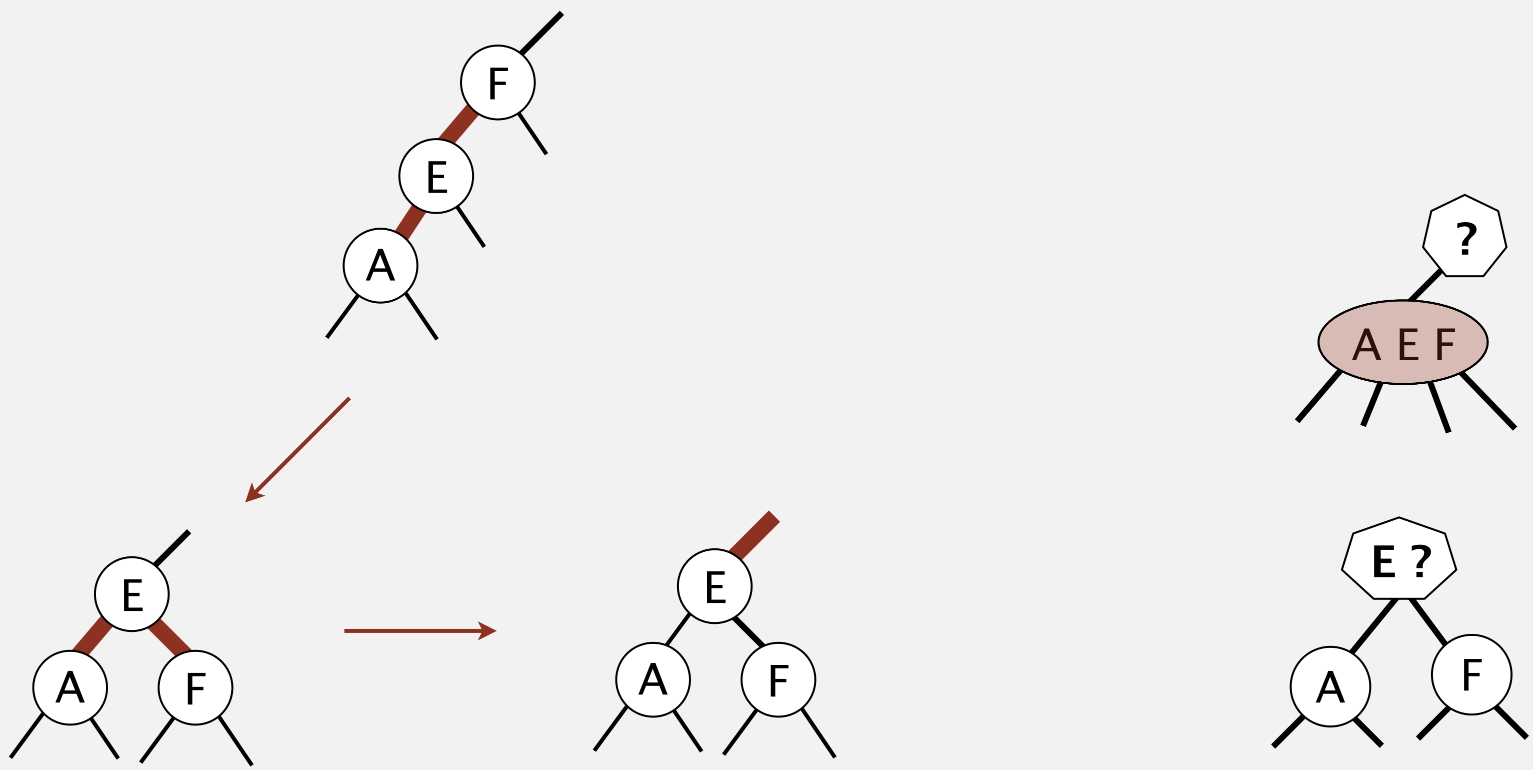
Case 2: Consecutive red left children

What is the problem here?

- Two red children in a row.
- [VIOLATION] 4 node.

How to Resolve?

- Rotate F right (back to case 1: two red children).



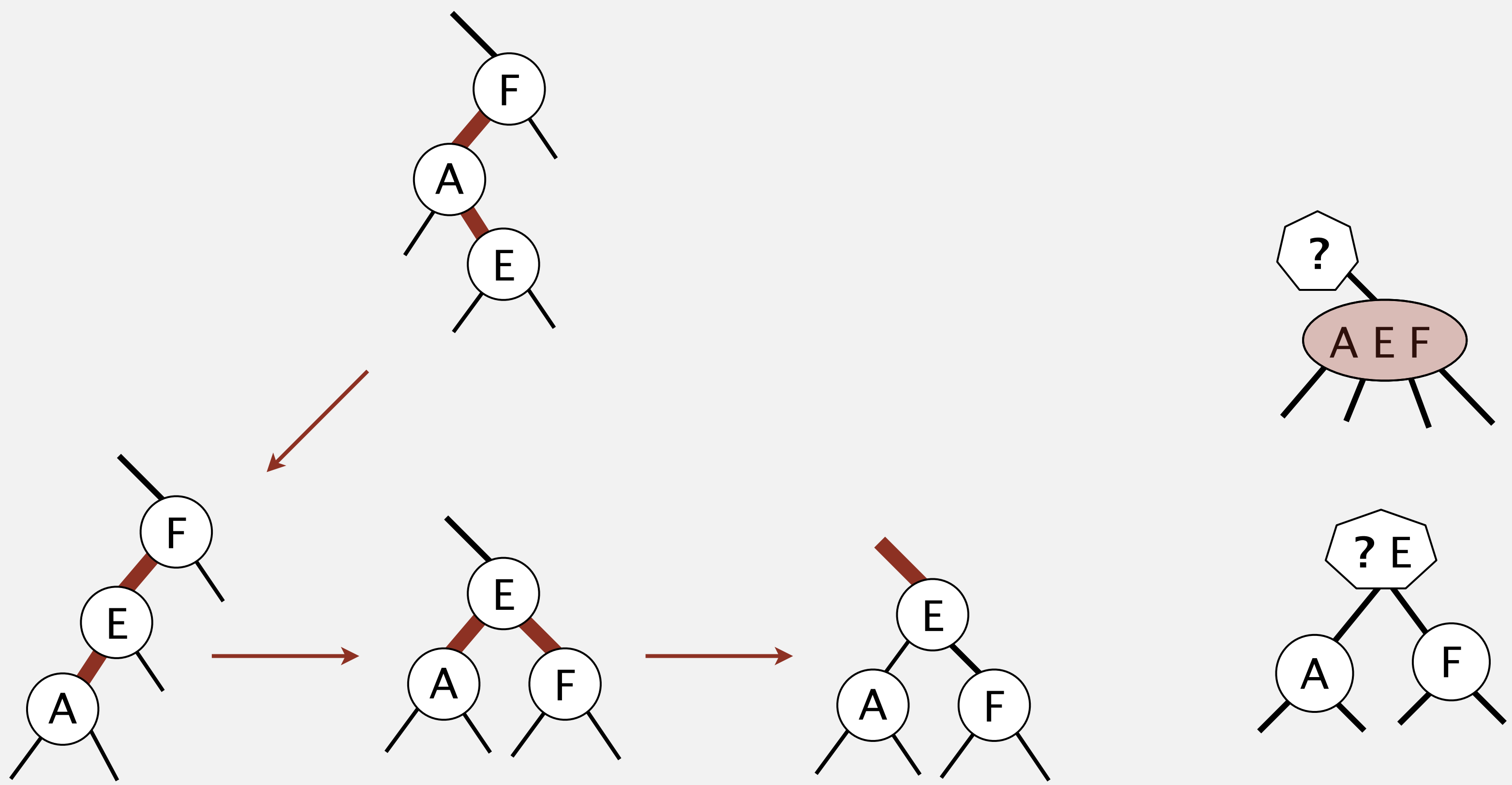
Case 3: Red right child and black left child

What is the problem here?

- Red links must lean left (by definition)
- [VIOLATION] Weird sort of 4-node.

How to Resolve?

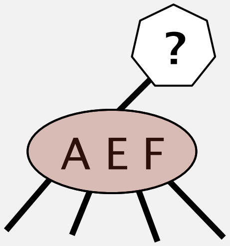
- Rotate A left. Either done or puts us right back into Case 2.

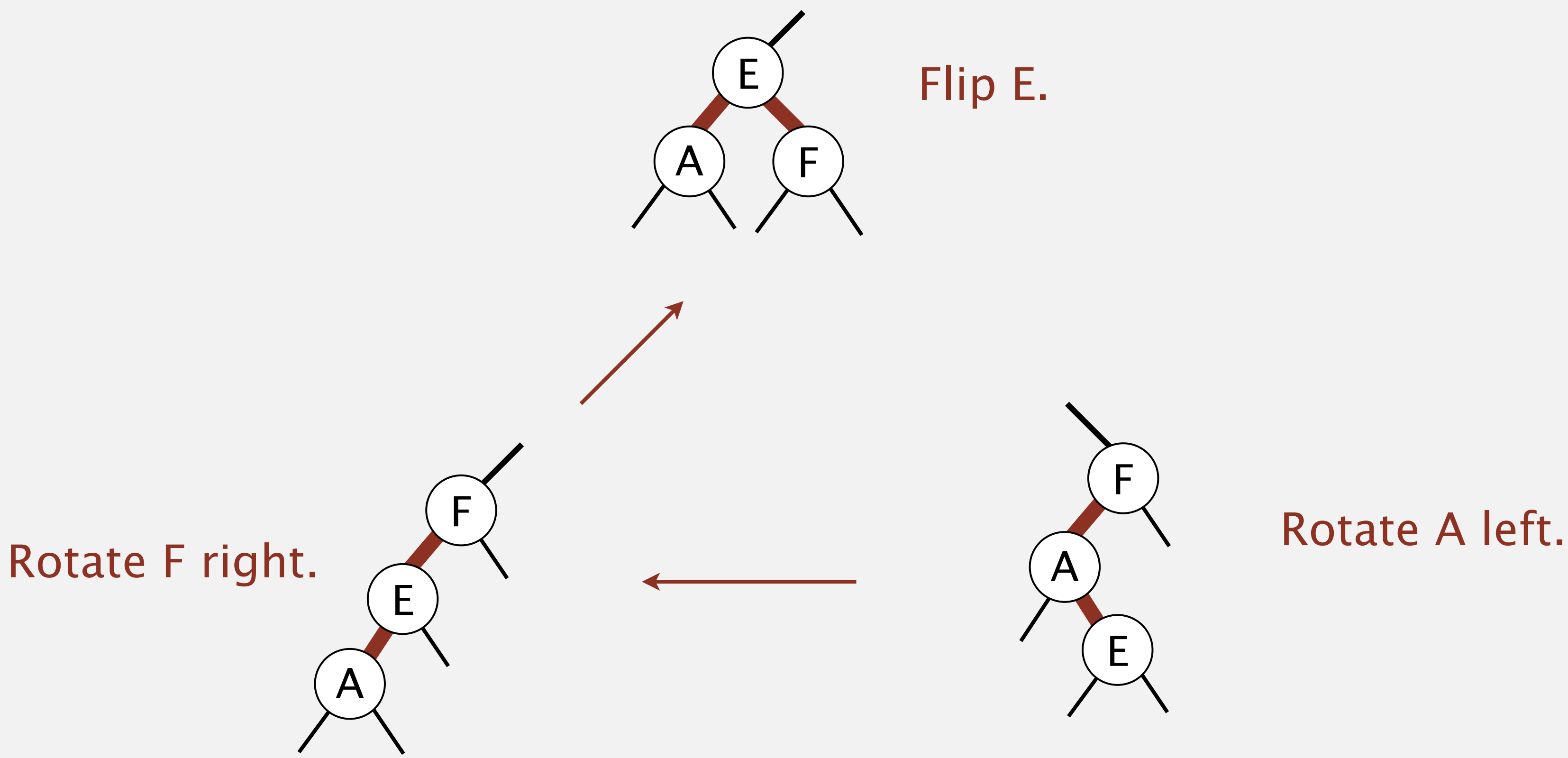


Summary: Mimicking 2-3 Trees

Three problem cases when inserting into 3-node.

- Two red children: Color flip
- Two consecutive red left children: Rotate right.
- Black left child, red right child: Rotate left.

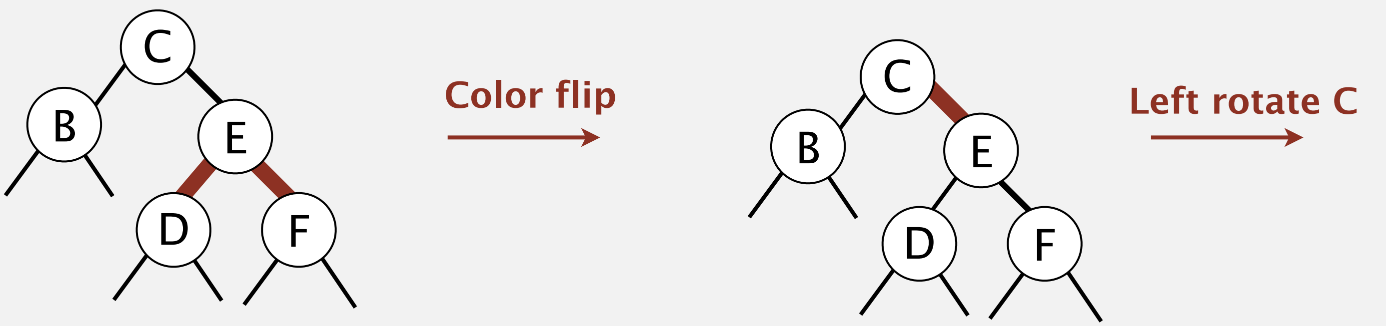




Color Flipping Dangers

What happens if a color flip leads to a violation?

- Repeat operations to preserve LLRB properties.



pollEv.com/jhug text to **37607**

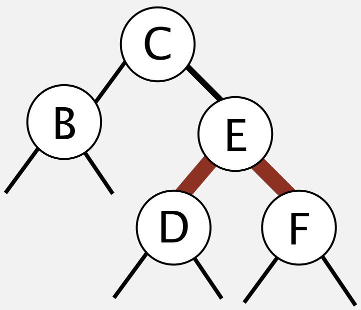
What is the correct operation to fix the tree?

```
A. Colorflip(C)     [189680]     D. LeftRotate(E)      [191780]
B. LeftRotate(C)    [189961]     E. RightRotate(E)     [191856]
C. RightRotate(C)   [190734]
```

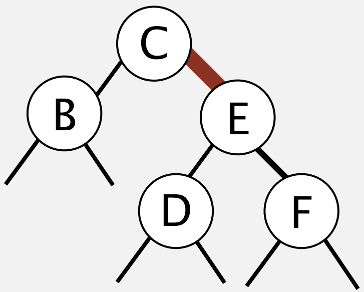
Color Flipping Dangers

What happens if a color flip leads to a violation?

- Repeat operations to preserve LLRB properties.



Color flip →



Left rotate C →

pollEv.com/jhug text to **37607**

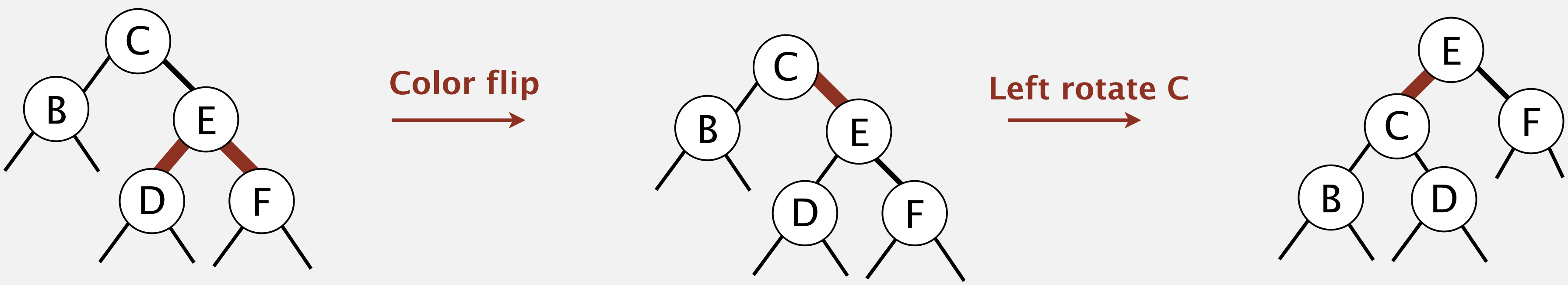
What is the level order traversal after left rotating C?

A. CBEDF [389282]
B. ECFBD [389283]
C. BCEDF [389284]
D. CDEBF [389288]

Color Flipping Dangers

What happens if a color flip leads to a violation?

- Repeat operations to preserve LLRB properties.

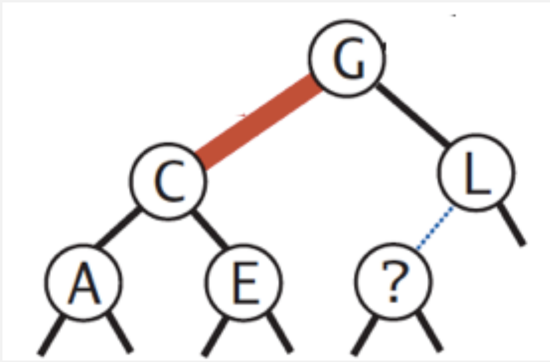


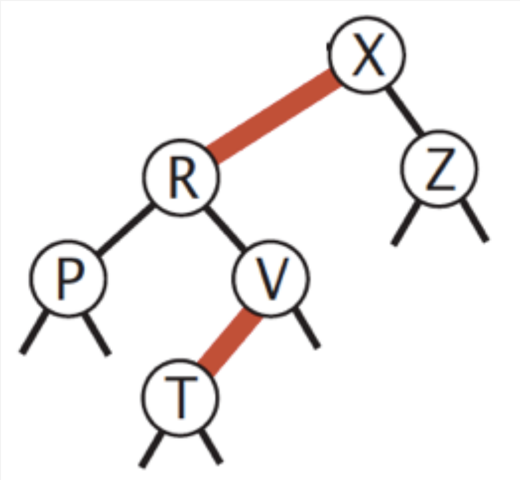
pollEv.com/jhug text to **37607**

What is the level order traversal after left rotating C?

B. ECFBD [389283]

Group Problems

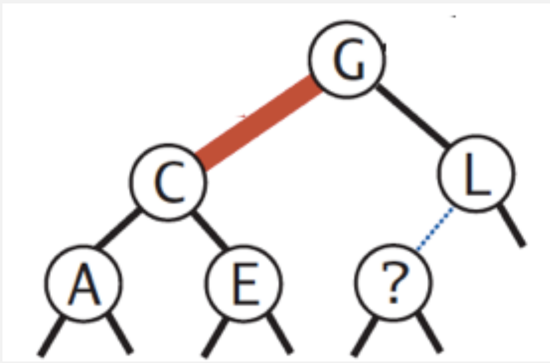


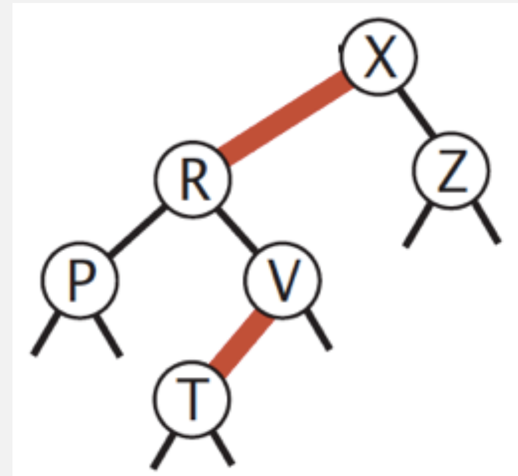


Groups of 3.

- What letters could possibly appear in the mystery node in the left tree?
- What color is the mystery node in the left tree?
- Give a very simple description of when we want to:
  - Rotate left:
  - Rotate right:
  - Color flip:
- If we insert W in the right tree, how many of the following must we perform:
  - Left rotation:
  - Right rotation:
  - Color flip:

Group Problems





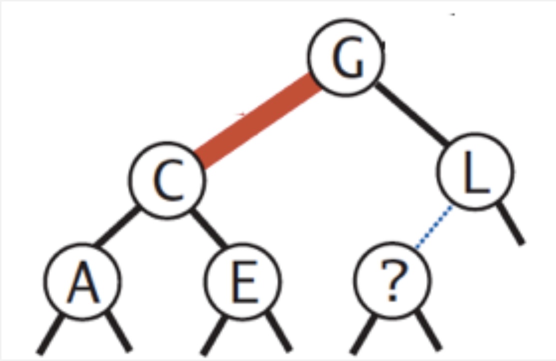
What letters could possibly appear in the mystery node?

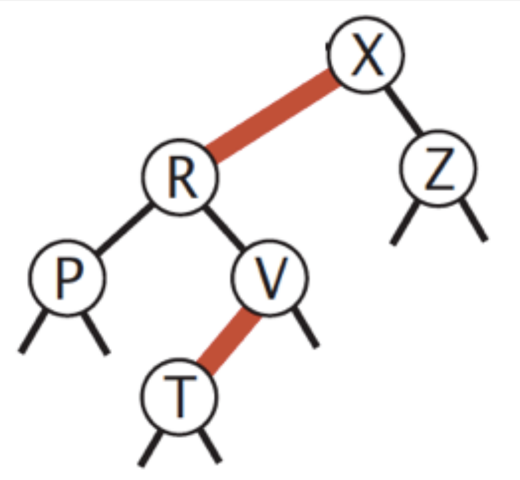
- Between G and L   [H I J K]

What color is the mystery node?

- red ?
  - NEED BALANCE
  - FIND OUR CENTER
  - LOOK TO THE HEAVENS
  - The path must be height 2 - Buddha

Group Problems





Give a very simple description of when we want to:

- Rotate left: Right red child
- Rotate right: Two consecutive left red children
- Color flip: Two red children

How many left, right, flip:

Left: 1

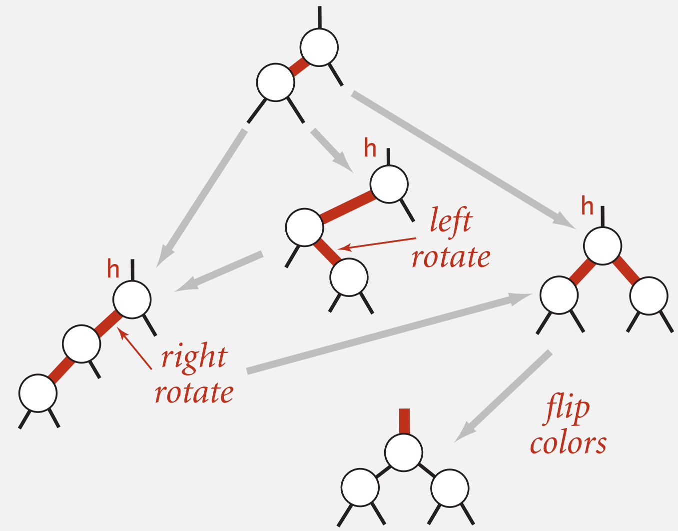
Right: 1

FLip: 2

Insertion in a LLRB tree: Java implementation

Same code for all cases.

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




```
private Node put(Node h, Key key, Value val)
{
   if (h == null) return new Node(key, val, RED);
   int cmp = key.compareTo(h.key);
   if      (cmp  < 0) h.left  = put(h.left,  key, val);
   else if (cmp  > 0) h.right = put(h.right, key, val);
   else if (cmp == 0) h.val = val;

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

   return h;
}
```

insert at bottom (and color it red)

lean left

balance 4-node

split 4-node

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

Insertion in a LLRB tree: Java implementation

```
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

Questions

- Why `isRed(h)` as opposed to `h.isRed()`?
- If `h.left` is null, will `h.left.left` throw an exception?

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

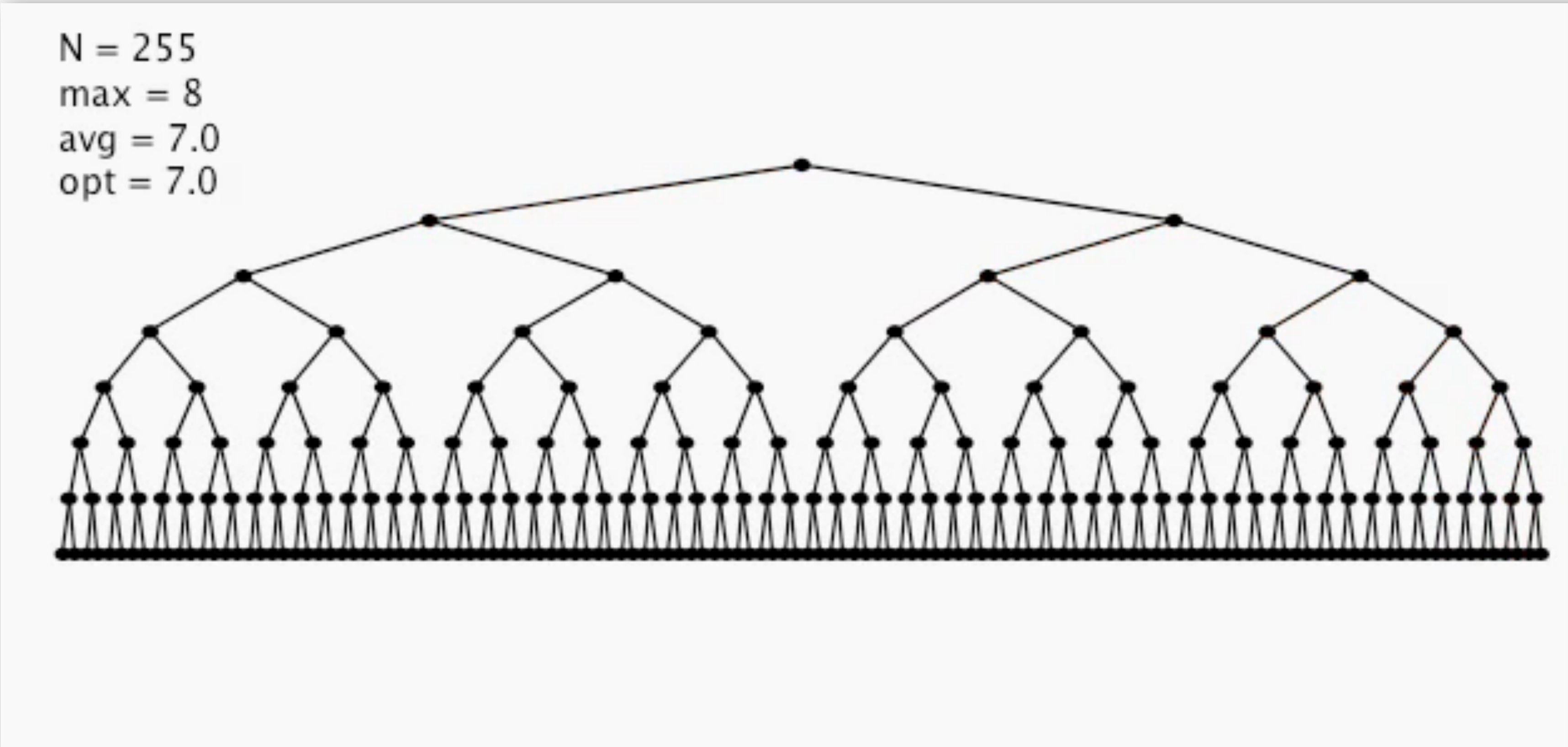
new code (that you have to learn)

```
public Node put(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);
   int cmp = kery.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

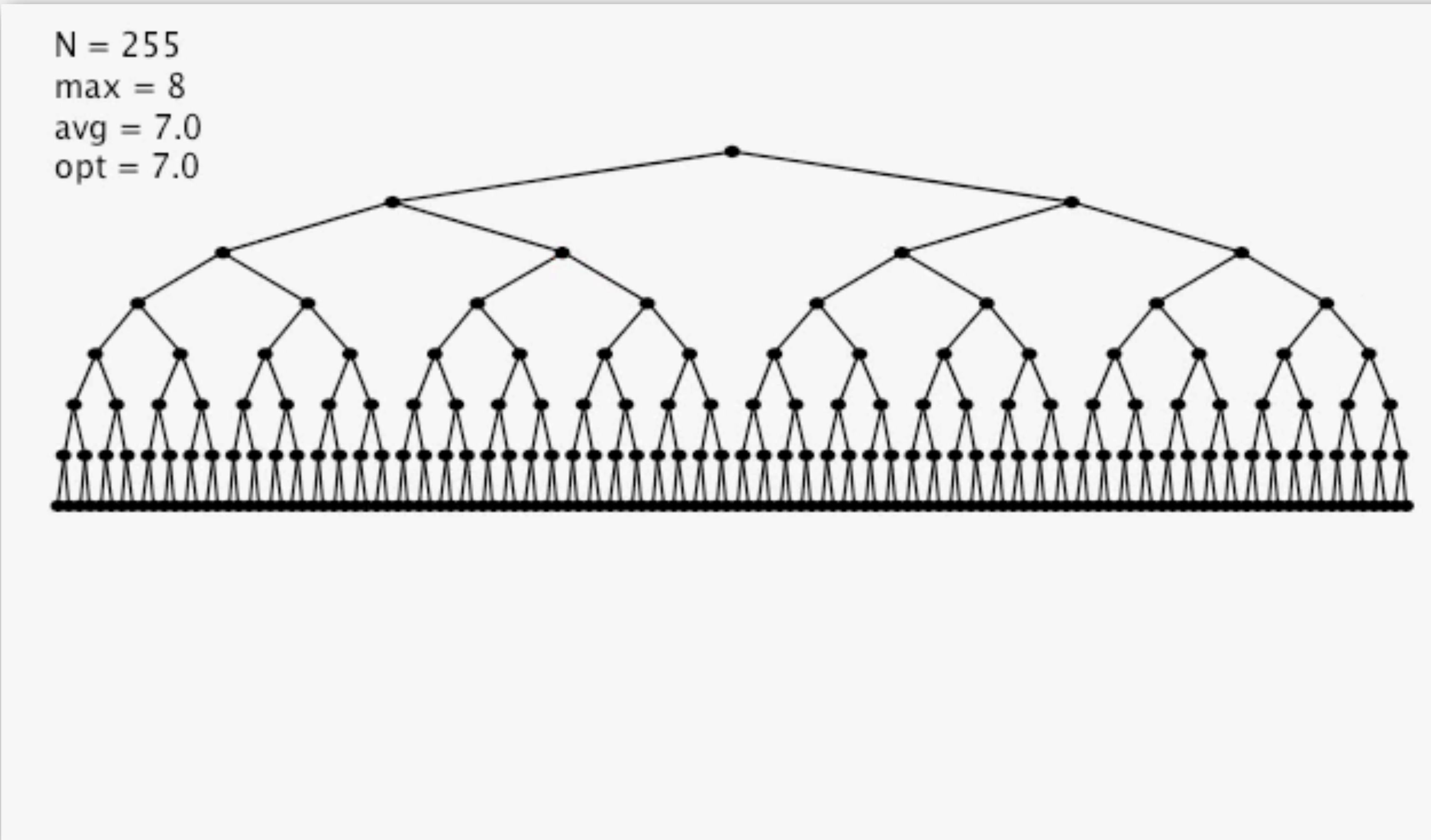
(if you've paid attention)

Insertion in a LLRB tree: visualization



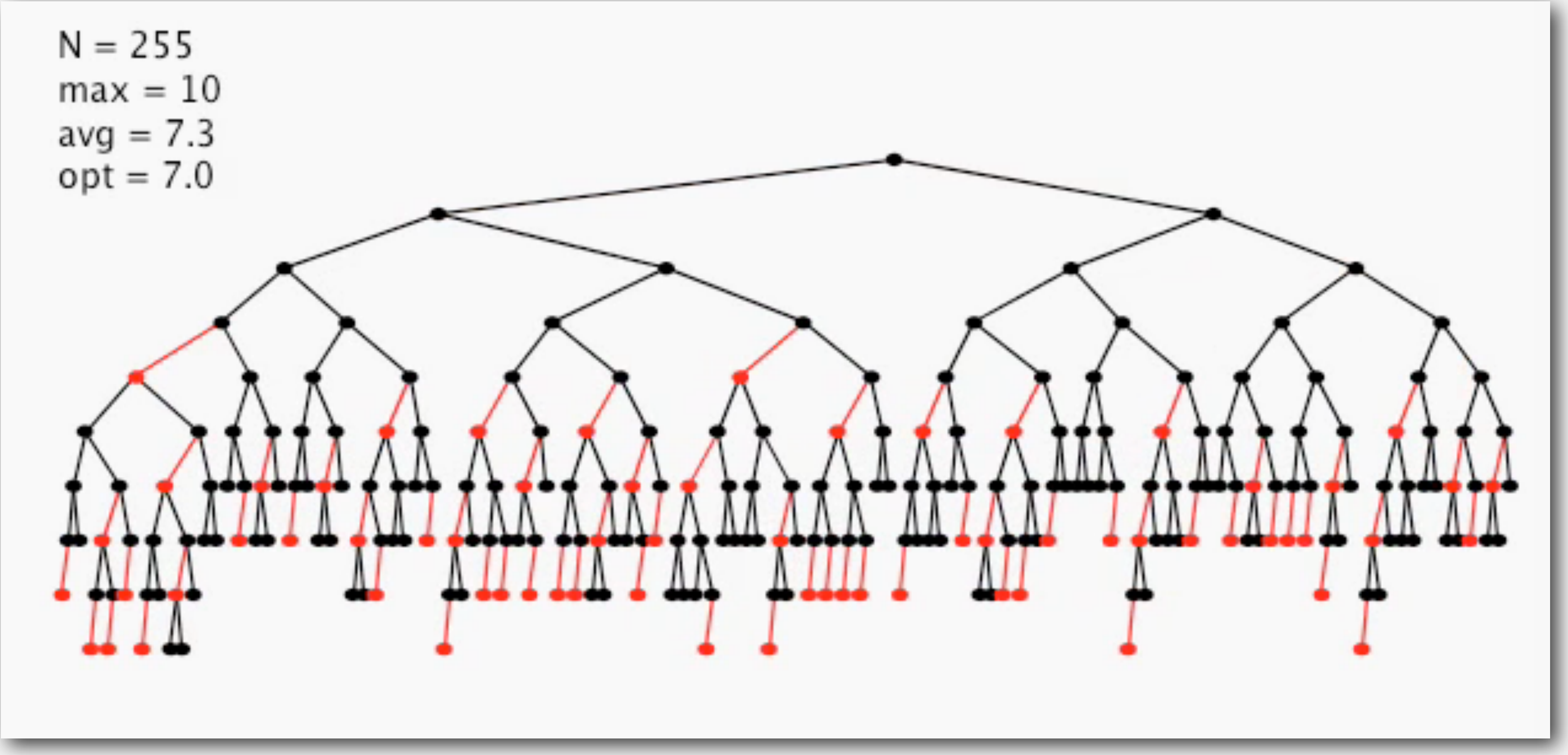
255 insertions in ascending order

Insertion in a LLRB tree: visualization



255 insertions in descending order

Insertion in a LLRB tree: visualization



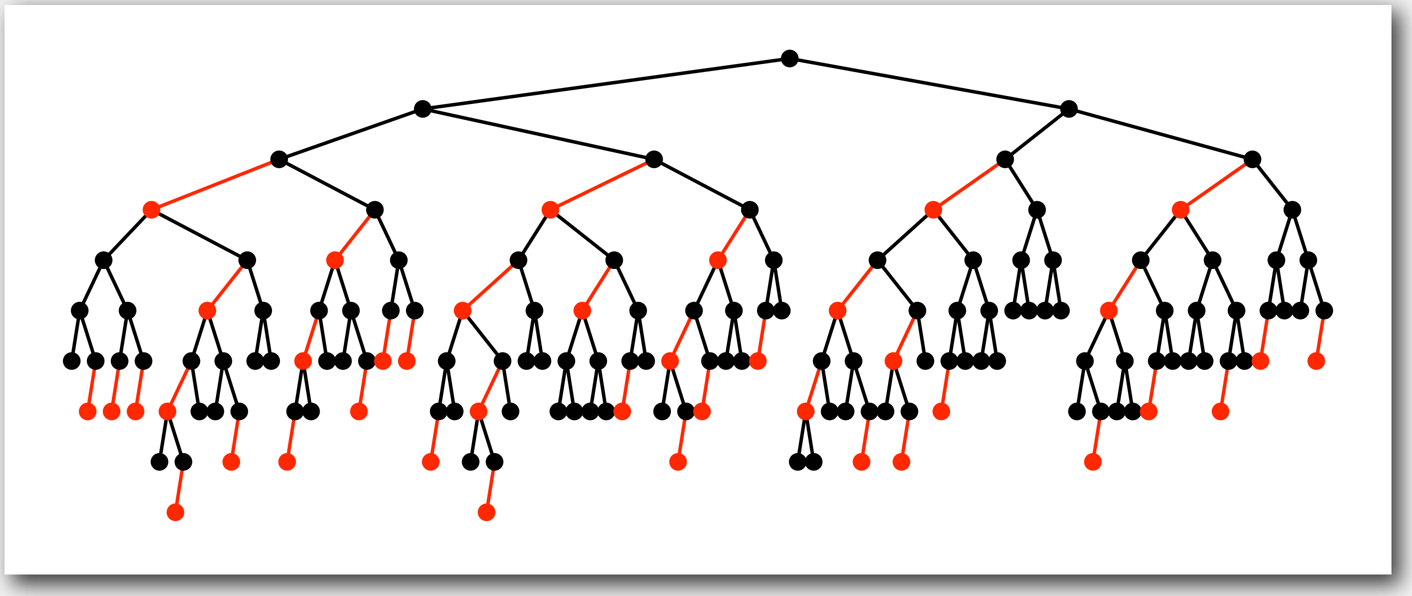
255 random insertions

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: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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

Logarithmic

Linear

- Billion items: 100 nanoseconds
- Trillion items: 0.1milliseconds
- 10^100 items: Long past the Stelliferous Era

Just how fast is logarithmic?

- Billion items: 100 nanoseconds
- Trillion items: 125 nanoseconds
- 10^100 items: 1 microsecond (if you had enough very tiny memory)

Can we do even better?

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	??	??	??	Constant*	Constant*	Constant*	no	??

* under a certain assumption

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

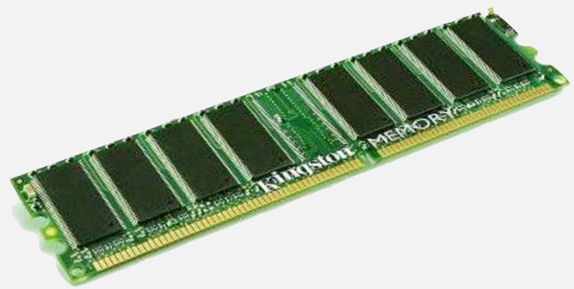
- 2-3 search trees
- red-black BST introduction
- red-black BST insert
- ***B-trees***

File system model

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

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

slow



fast

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

Cost model. Number of probes.

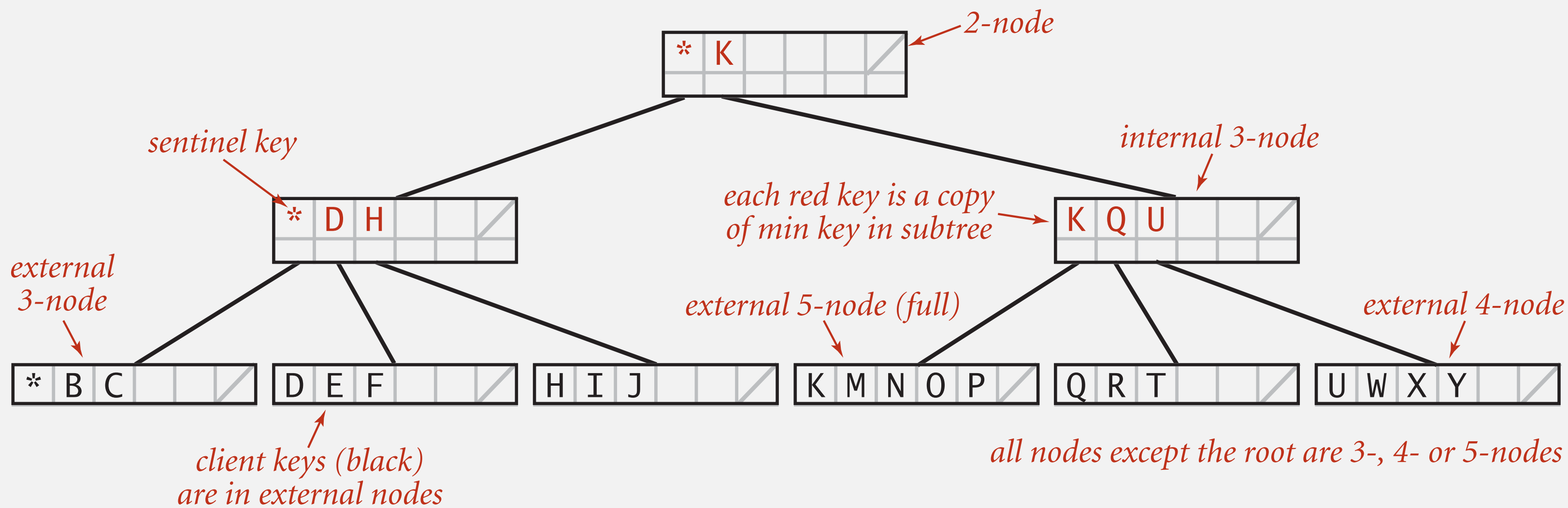
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

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

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

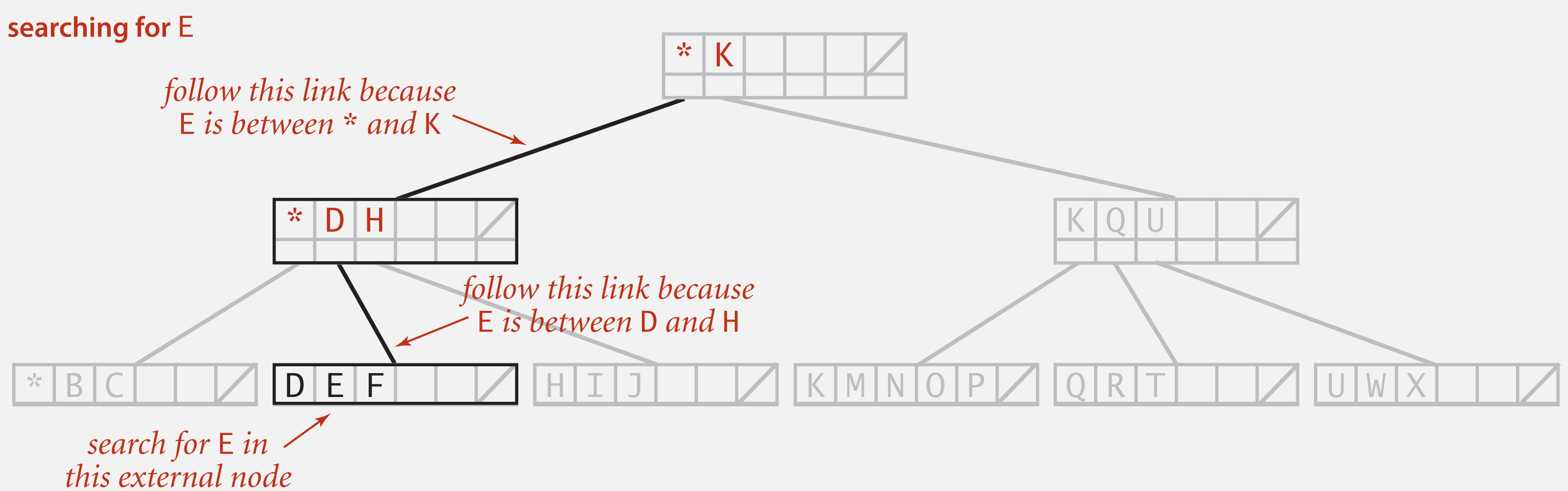
choose M as large as possible so that M links fit in a page, e.g., M = 1024



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

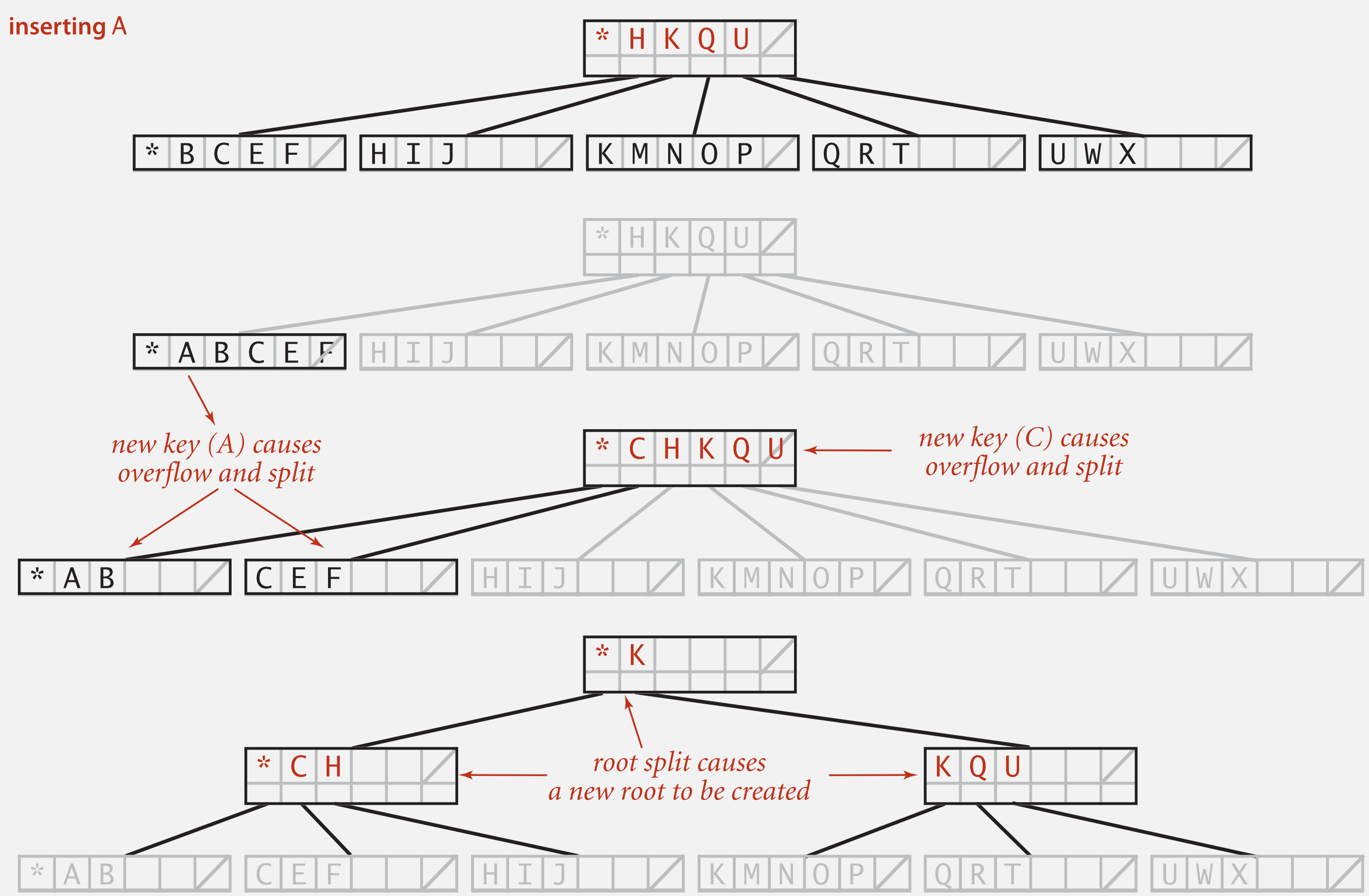
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Insertion in a B-tree

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



Inserting a new key into a B-tree set

Balance in B-tree

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

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

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

Optimization. Always keep root page in memory.

Building a large B tree

Balanced trees in the wild

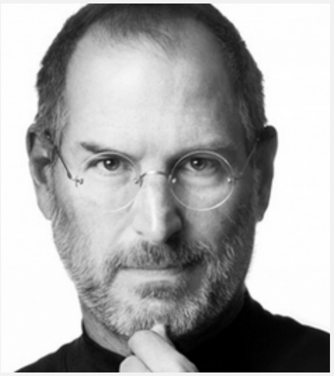
Red-black trees are widely used as system symbol tables.

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

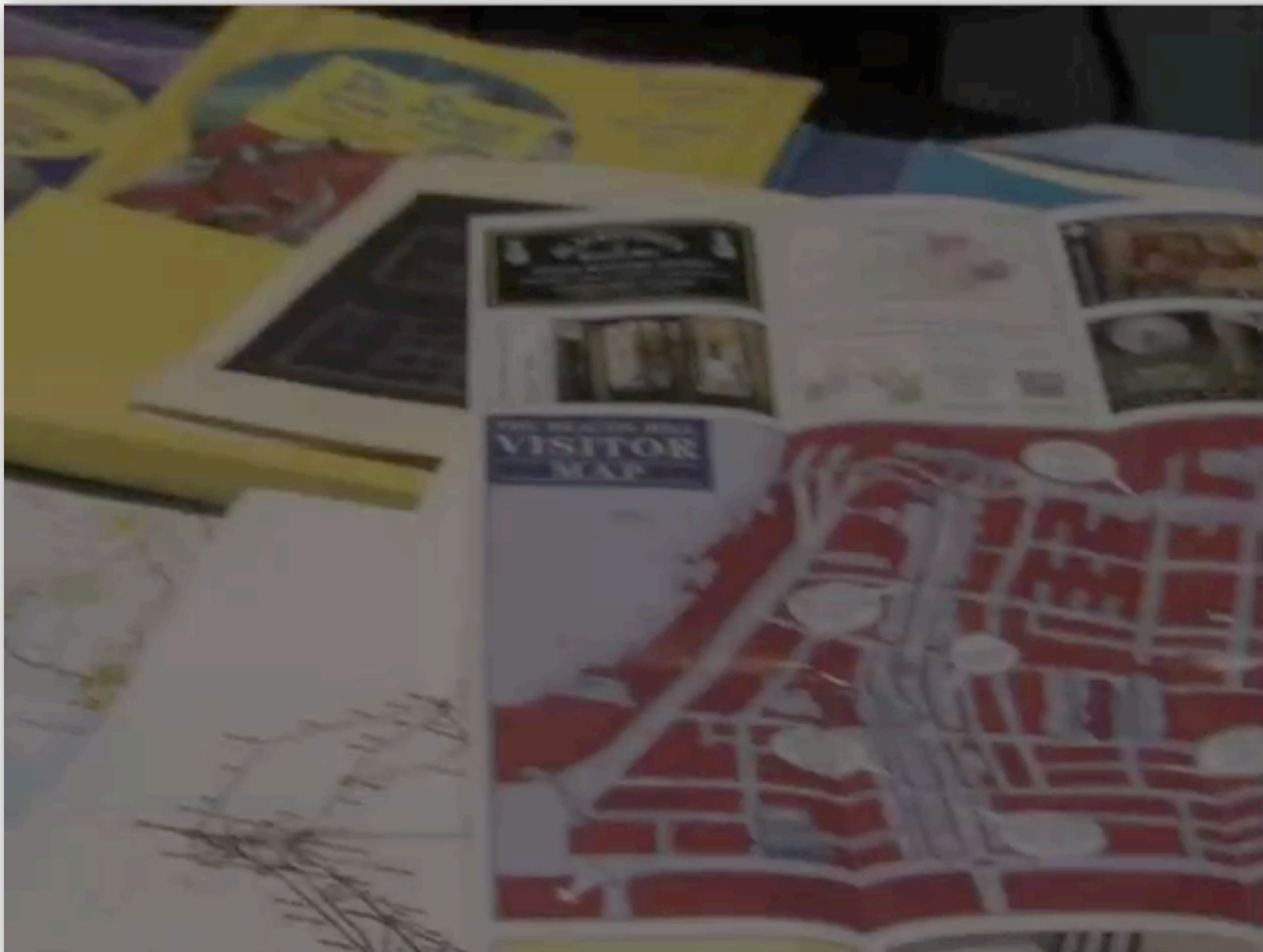
B-tree variants. B+ tree, B*tree, B# tree, …

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

- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.



Red-black BSTs in the wild



Common sense. Sixth sense.
Together they're the
FBI's newest team.

Red-black BSTs in the wild

ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?