Midterm exam

In-class midterm. 11-12:20pm on Tuesday, October 22.

- Rooms TBA.
- No makeups.

Rules.

including associated readings and assignments (but no serious Java programming)

- Closed book, closed note.
- Covers all material thru Lecture 10 (hashing).
- No computers or other computational devices.
- 8.5-by-11 cheatsheet (one side, in your own handwriting).

Midterm preparation.

Even better, warn us by email before asking so we can think about them ahead of time!

- **Study guide.** Especially the old midterm and final problems!
  - Form a study group! See "Search for Teammates" on Piazza.
- Leave A level problems to me, Bob or Guna. Some are very hard!
- Bring questions to precept, office hours, or review session. TBA
- No assignment this week specifically so you can start studying!

http://algs4.cs.princeton.edu

3.2 Binary Search Trees

- *BSTs*
- *deletion*
- *ordered operations (optional)*
- *inorder traversal (oops!)*

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

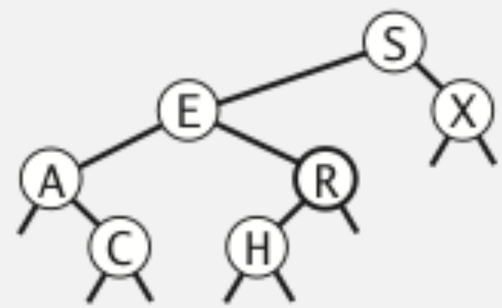
3.2 Binary Search Trees

- BSTs
- deletion
- ordered operations (optional)
- inorder traversal (oops!)

Inorder traversal

- Traverse left subtree.
- Print key.
- Traverse right subtree.

```
private void inorder(Node x)
{
   if (x == null) return;
   inorder(x.left, q);
   System.out.print(x.key + " ");
   inorder(x.right, q);
}
```



A C E H R S X

Interpretation:

- Crawl around the graph counterclockwise, and yell when you see the underside of a node.

Property. Inorder traversal of a BST yields keys in ascending order.

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

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

Symbol table review

Last time:

- BSTs are a huge step forward from an array or linked list.

Performance issues with BSTs:

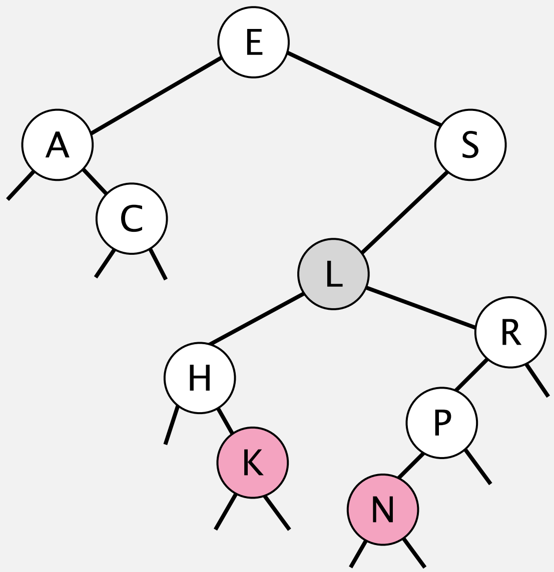
- 1:
- 2:

Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 2. [2 children] Delete `t` by replacing parent link.

Example. `delete(L)`



Choosing a replacement.

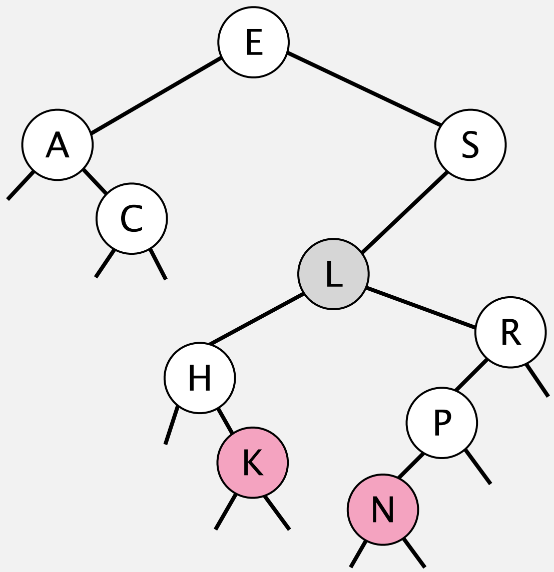
- Successor: N
- Predecessor: K

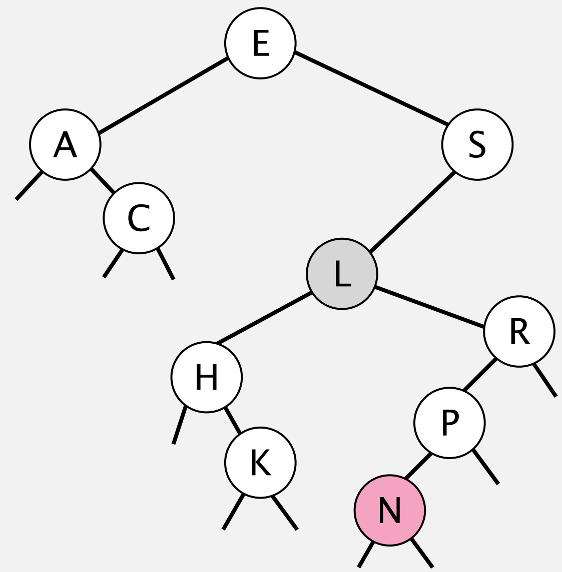
Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 2. [2 children] Delete `t` by replacing parent link.

Example. `delete(L)`





Choosing a replacement.

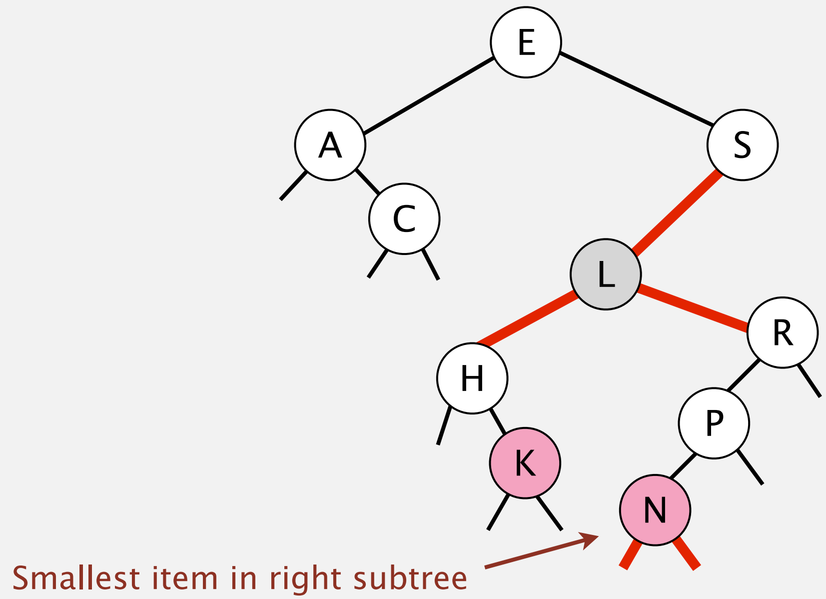
- Successor: N [by convention]
- ~~Predecessor: K~~

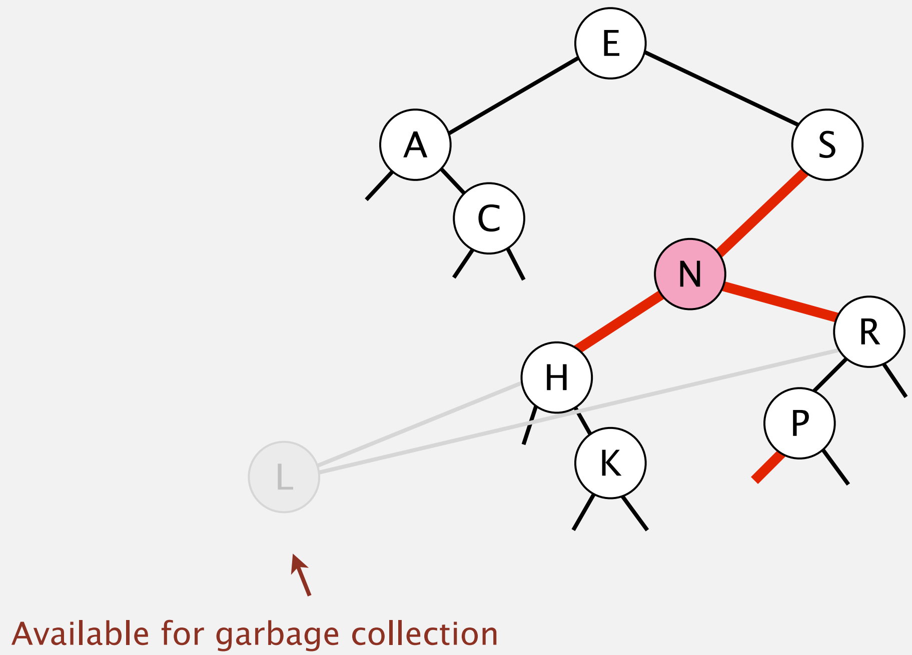
Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 2. [2 children] Delete `t` by replacing parent link.

Example. `delete(L)`



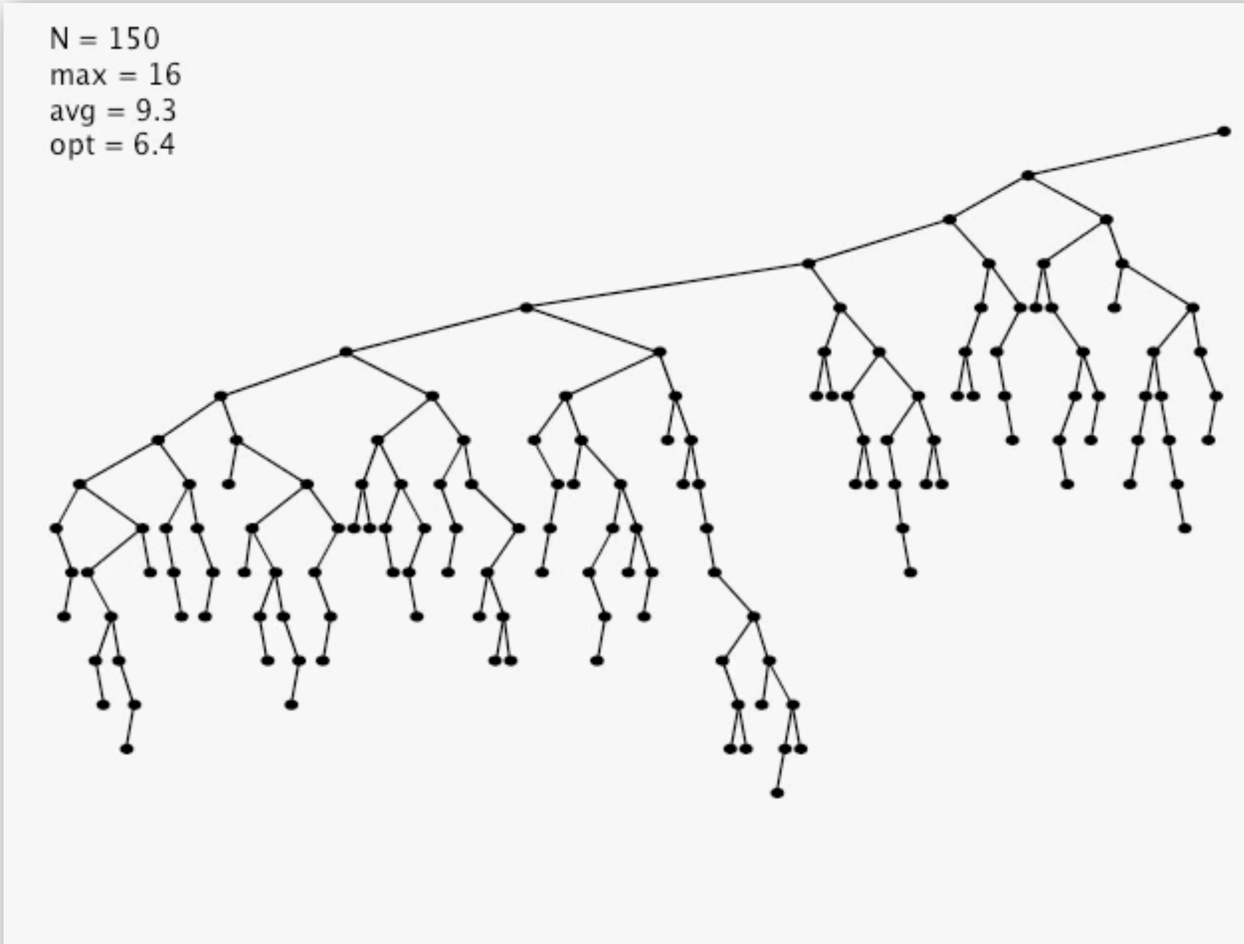


Four pointers must change.

- Parent of deleted node
- Parent of successor
- Left child of successor
- Right child of successor

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) $\Rightarrow$ $\text{sqrt}\,(N)$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

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()`
BST, after many deletes	N	N	N	√N	√N	√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 (optional).

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

- 2-3 search trees
- red-black BSTs
- B-trees (optional)

2-3 tree

Allow 1 or 2 keys per node.

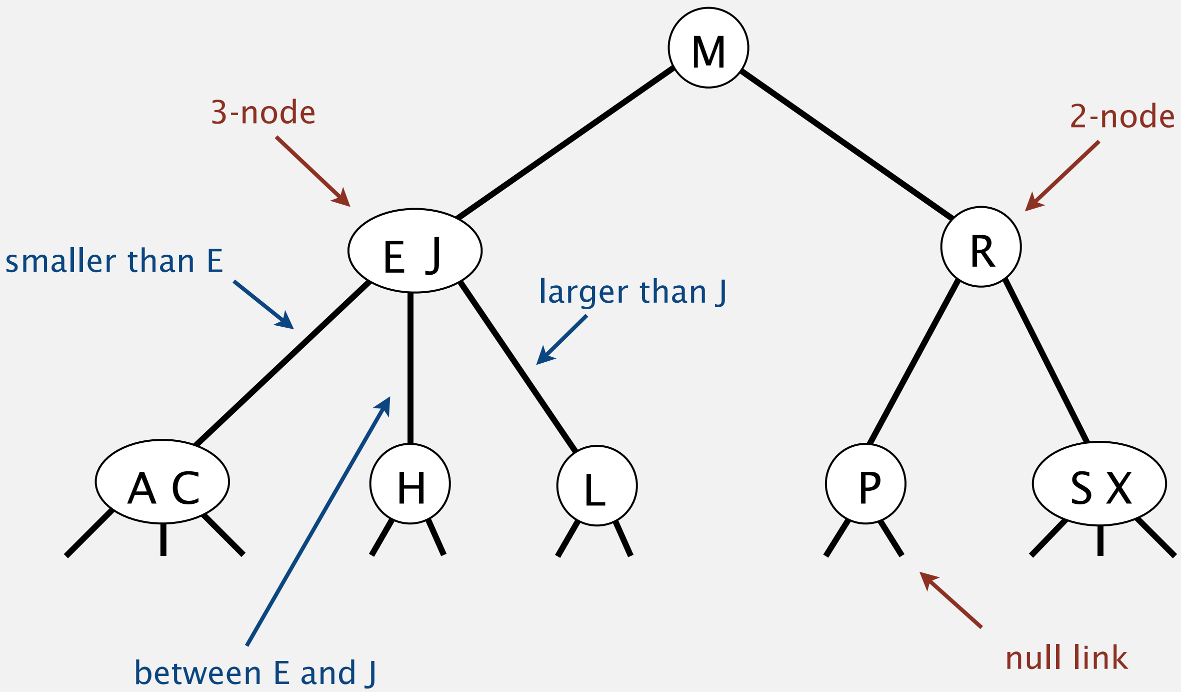
- 2-node: one key, two children.
- 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.

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

How???

All shall be revealed.

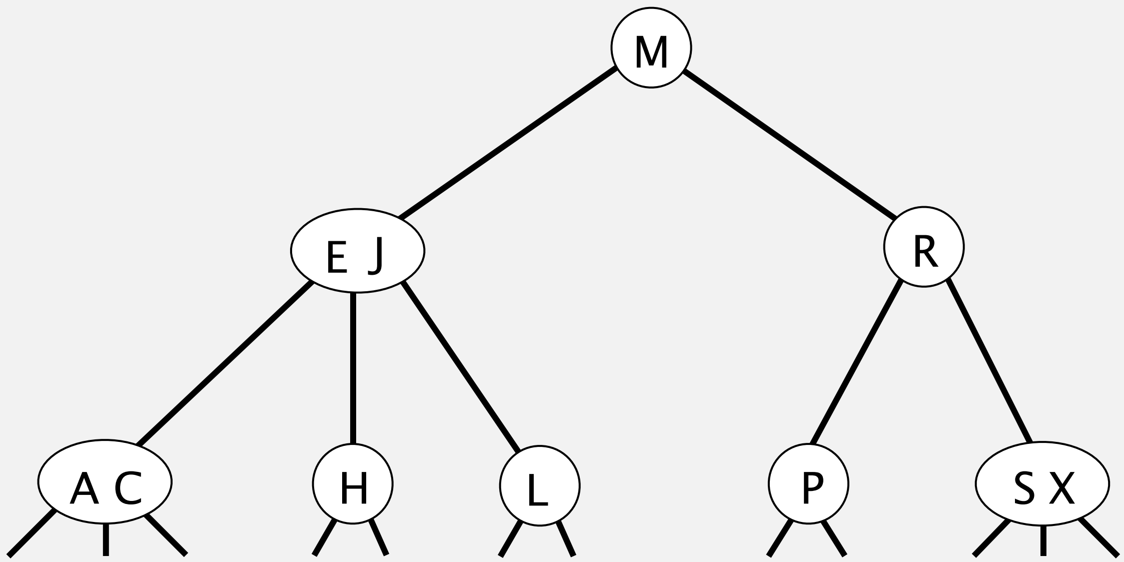


2-3 tree demo

Search.

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

search for H

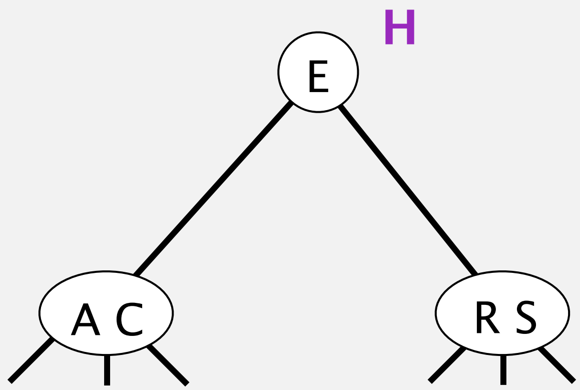


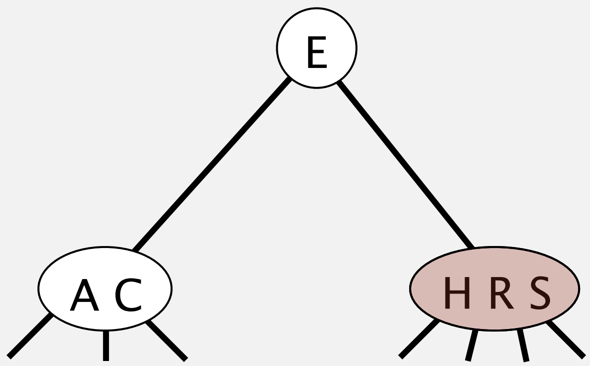
2-3 Trees

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!

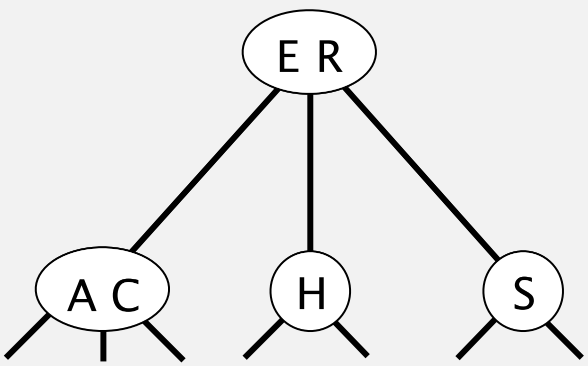
2-3 Trees





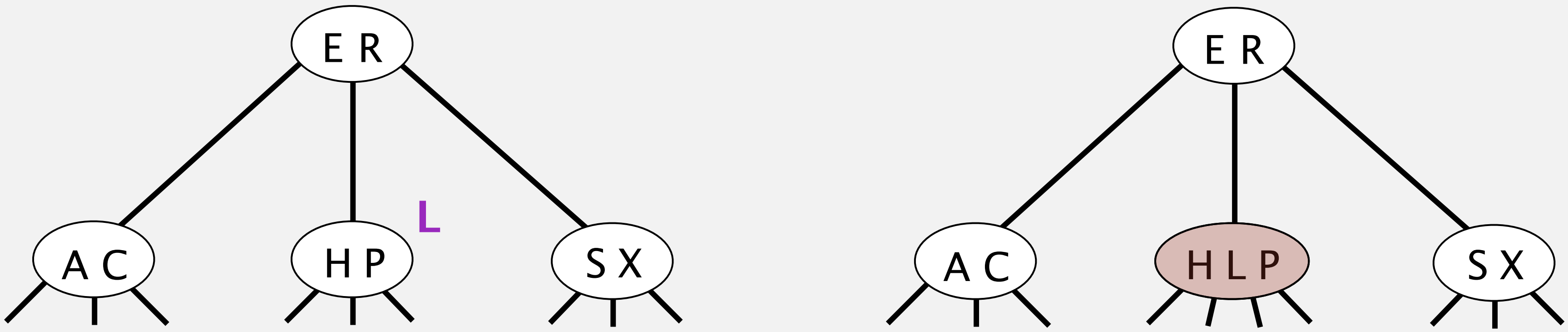
Insert H (into 3-node)

- 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.

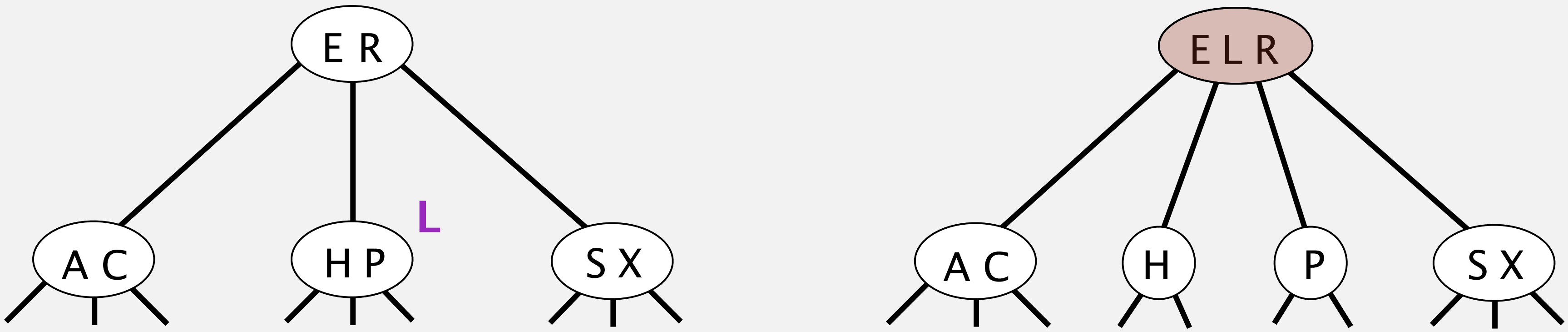
2-3 Trees



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

- [VIOLATION] HLP created.

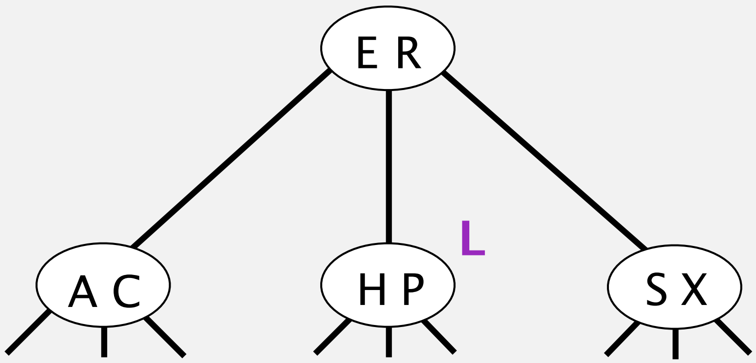
2-3 Trees

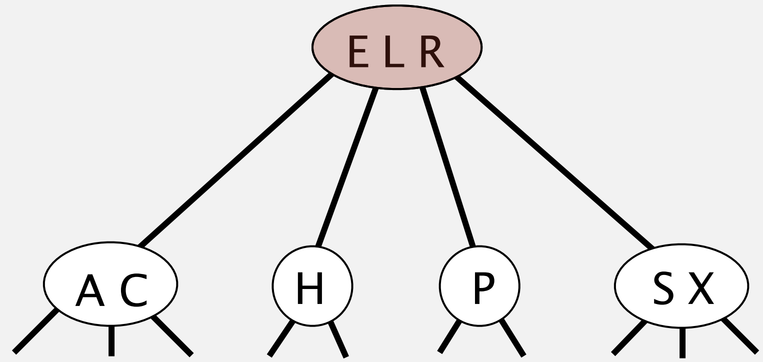


Insert L (into 3-node with 3-node parent)

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

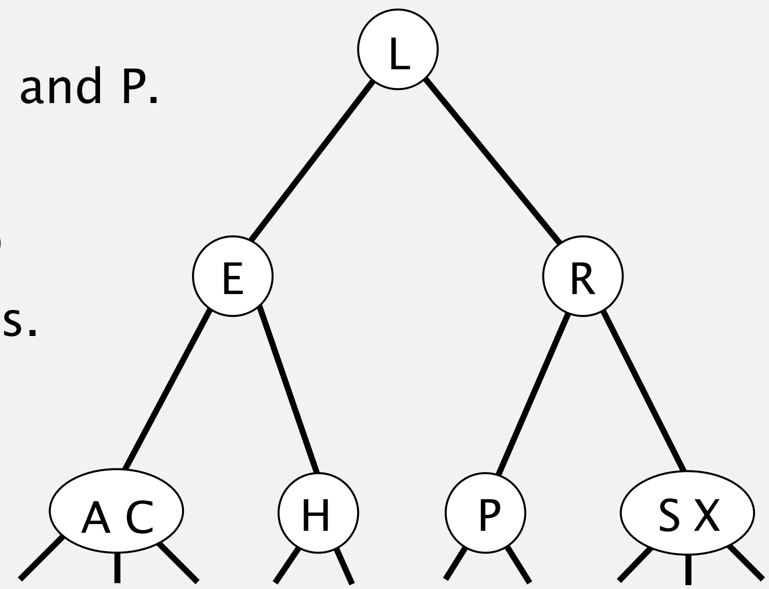
2-3 Trees





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

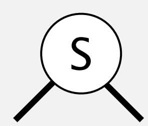
- [VIOLATION] HLP created. Send L up, create H and P.
- [VIOLATION] ELR created.
- Send L to join parent (no parent, so new root)
  - Create two new 2-nodes E-R from the debris.
  - Each gets custody of two nodes.



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

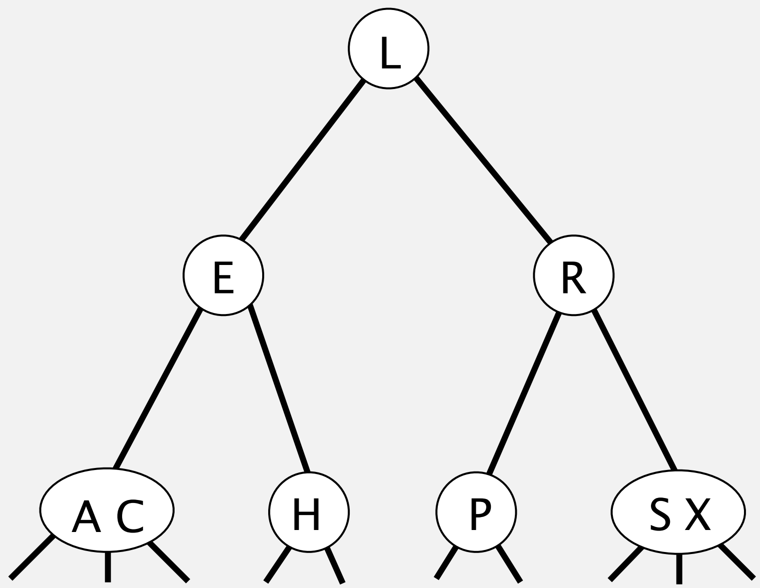
2-3 tree construction demo

insert S



2-3 tree construction demo

2-3 tree



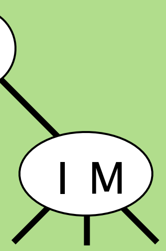
2-3 Tree Construction

Your turn.

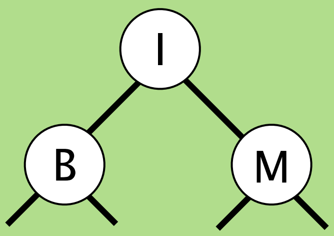
- Insert B, I, M. Which tree do you get?

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

Which is the correct 2-3 tree?



[484710]

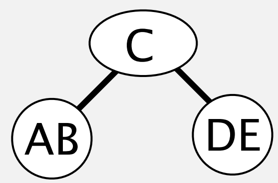


[484711]

2-3 Tree Construction

One more.

- Suppose we insert 5 nodes and get the tree shown below:

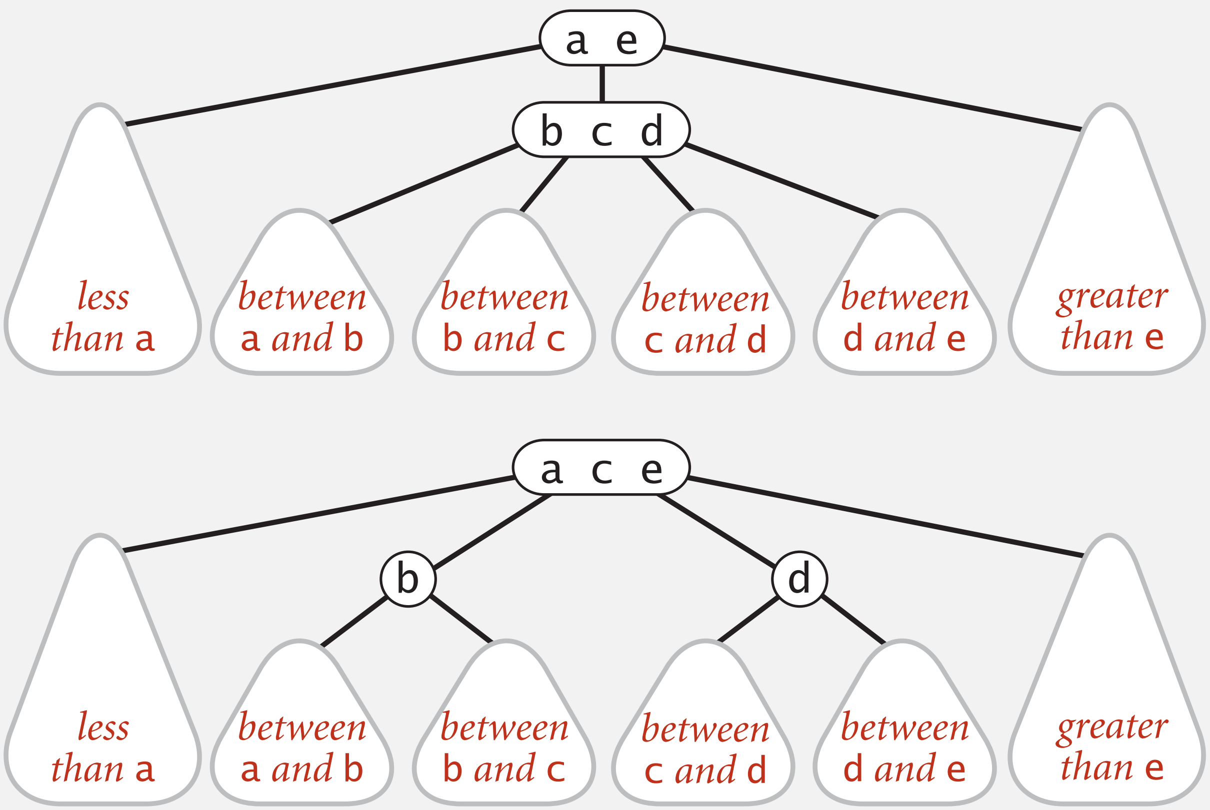


pollEv.com/jhug text to **37607**

```
Which insertion sequence resulted in the tree above?
1. ABCDE   [489691]
2. CABDE   [489700]
3. ACEDB   [489895]
4. None of these and the 2-3 tree is valid.      [489896]
5. None of these and the 2-3 tree is invalid.    [489897]
```

Local transformations in a 2-3 tree

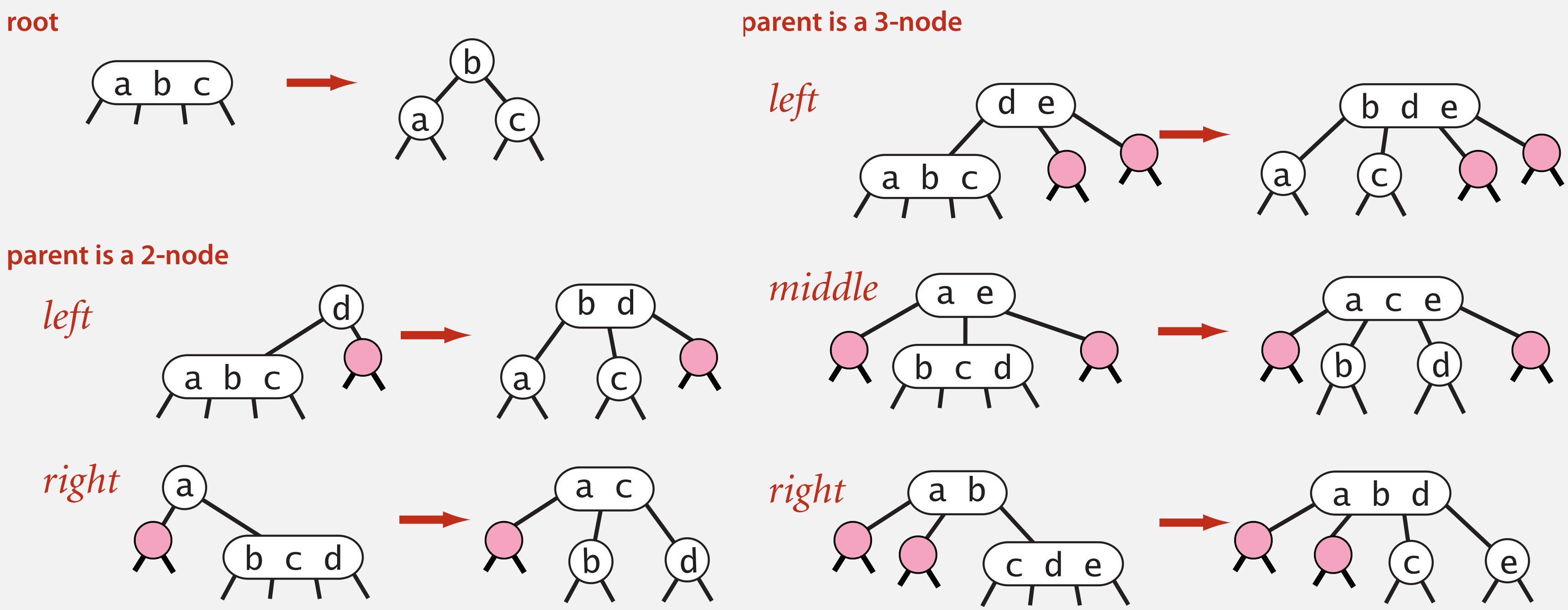
Splitting a 4-node is a local transformation: constant number of operations.



Global properties in a 2-3 tree

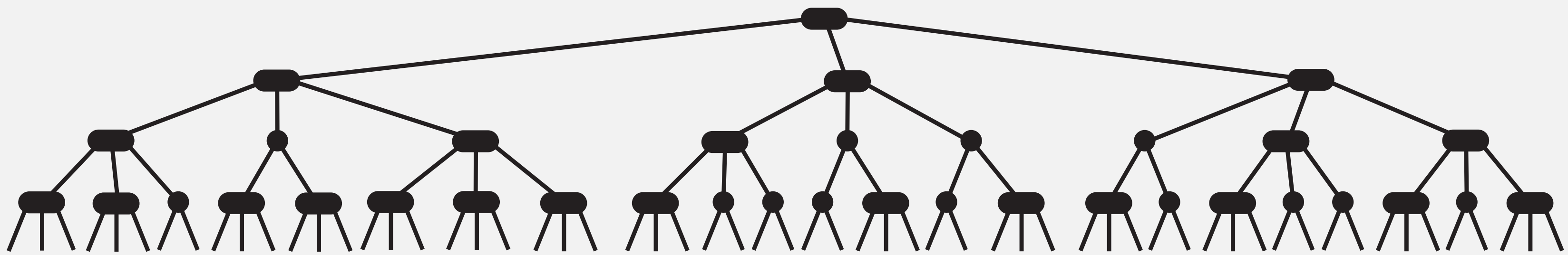
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.



2-3 tree: performance

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

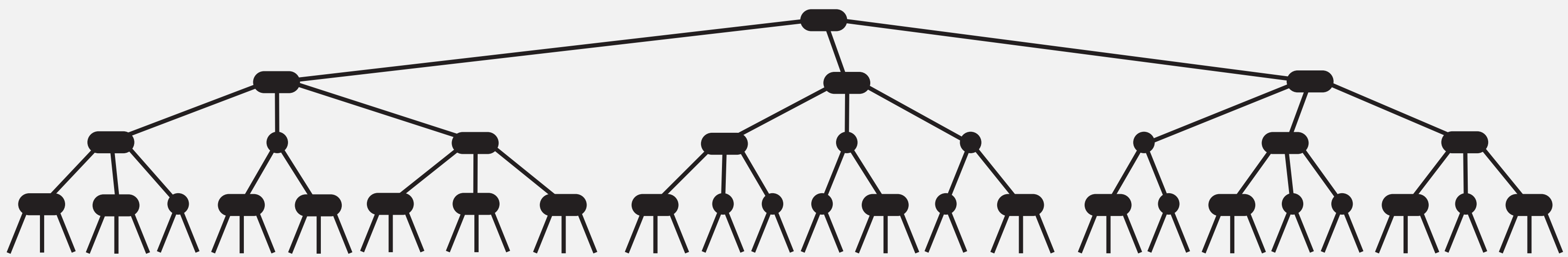


Tree height.

- Worst case:
- Best case:

2-3 tree: performance

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



Tree height.

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

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()`
BST, after many deletes	N	N	N	$\sqrt{N}$	$\sqrt{N}$	$\sqrt{N}$	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

2-3 tree: implementation?

Direct implementation is complicated, because:

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

fantasy code

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

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

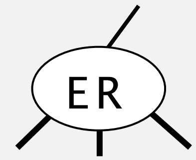
3.3 BALANCED SEARCH TREES

- 2-3 search trees
- red-black BSTs
- B-trees (optional)

The problem with 2-3 trees

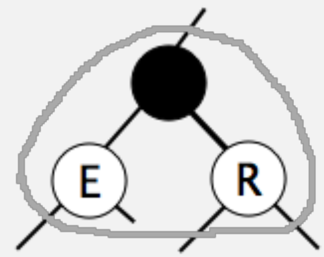
Hard to implement

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

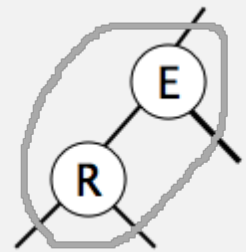


Goal: Represent as binary tree

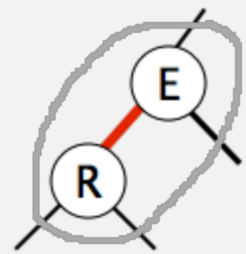
- Approach 1: Glue nodes.
  - Wasted space, wasted link.
  - Code probably messy.



- 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.

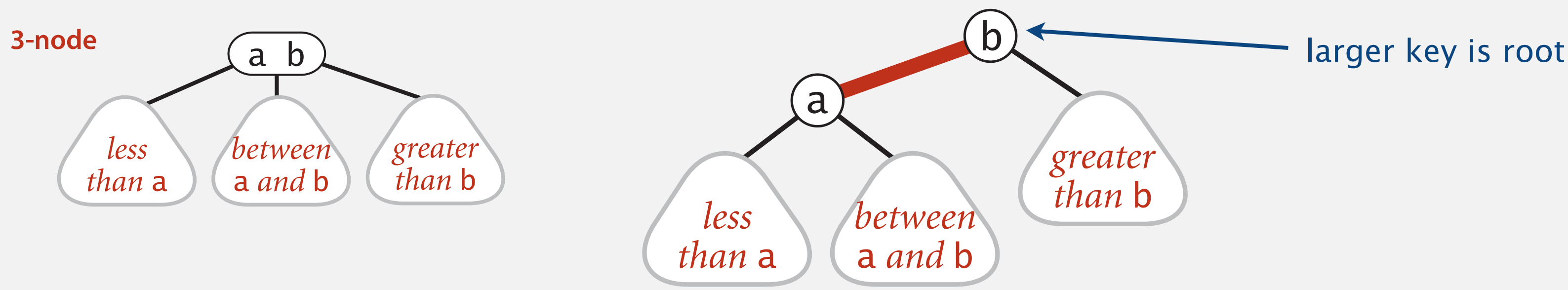


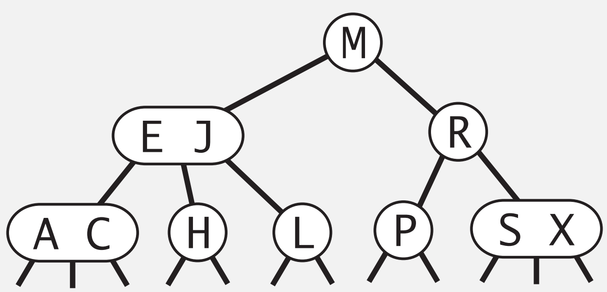
- Approach 3: BST with glue links.
  - Used widely in practice.
  - Arbitrary restriction: Red links lean left.



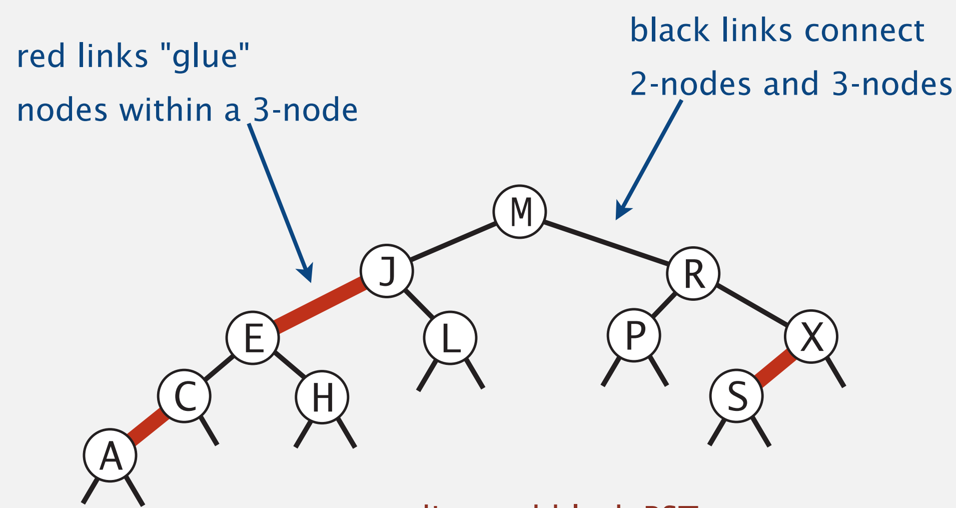
Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

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





2-3 tree



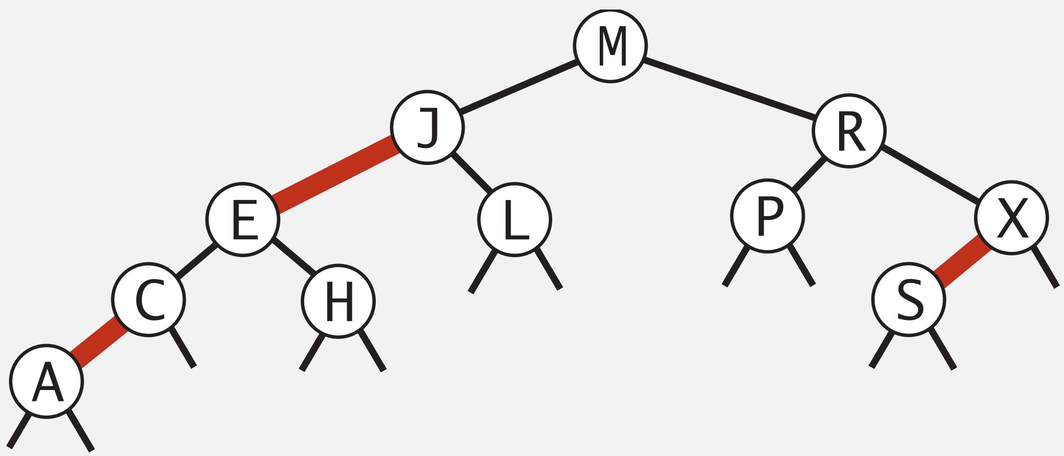
corresponding red-black BST

An equivalent definition

A BST such that:

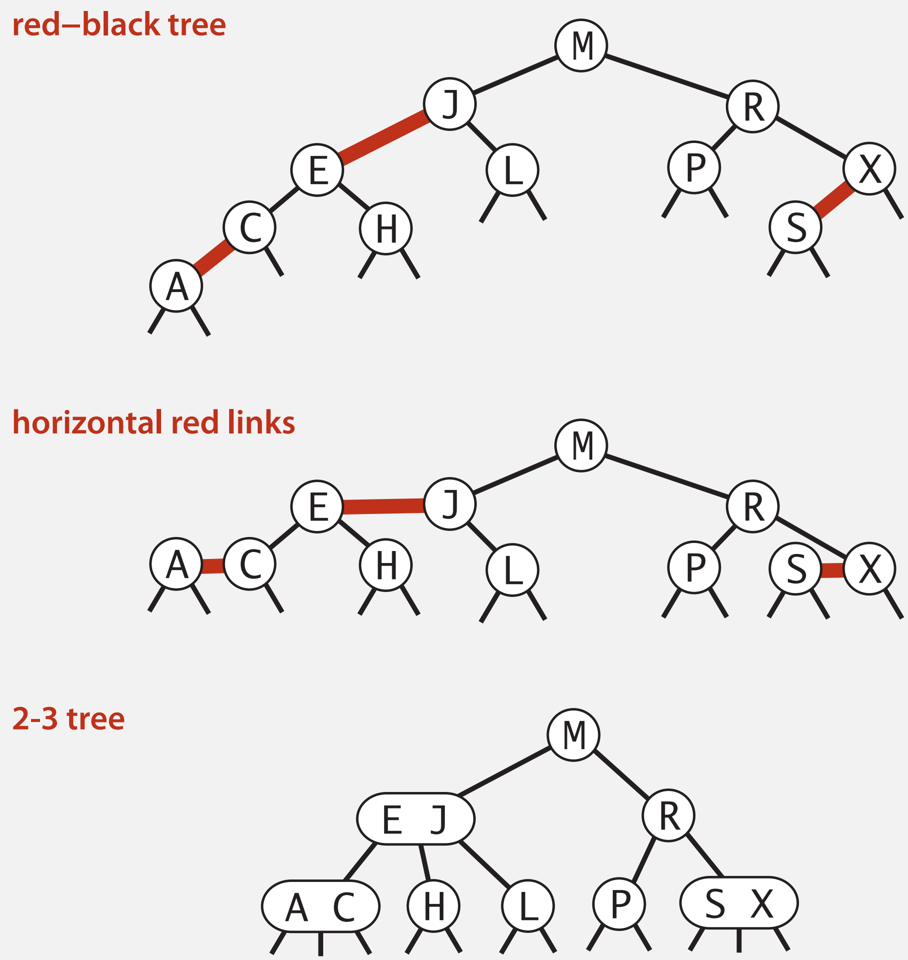
- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

"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.

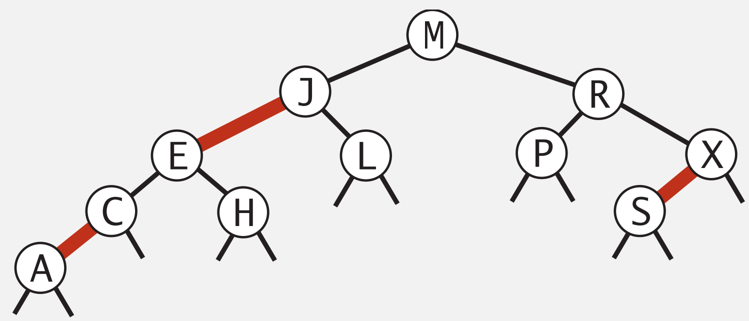


Search implementation for red-black BSTs

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

but runs faster
because of better balance

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



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

Red-black BST representation

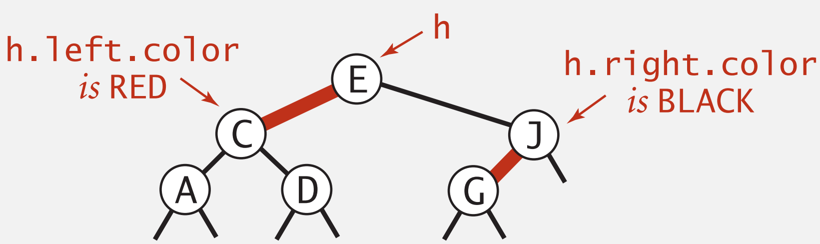
Each node is pointed to by precisely one link (from its parent) $\Rightarrow$ can encode color of links in nodes.

```
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



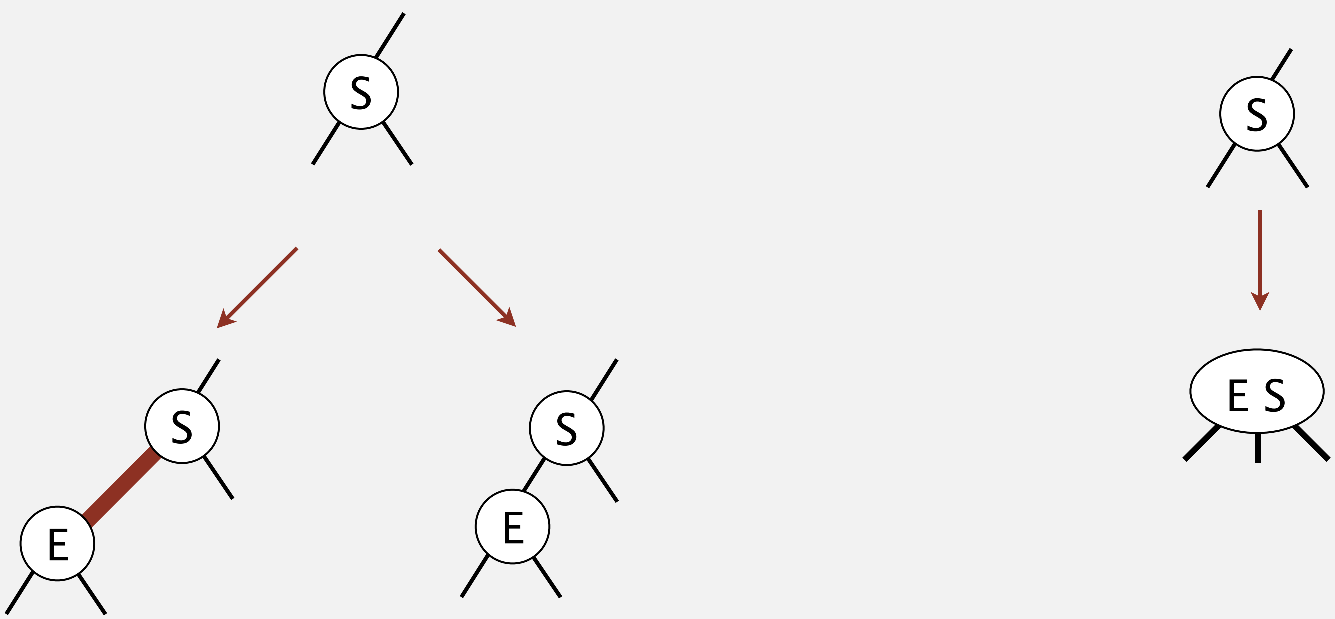
Thought experiment on link color for new nodes

Should we use a red or a black link when inserting to the left of a 2-node?

- Red link.

What about in other cases (right of a 2-node, into a 3-node)?

- Red link. Because:
  - 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.



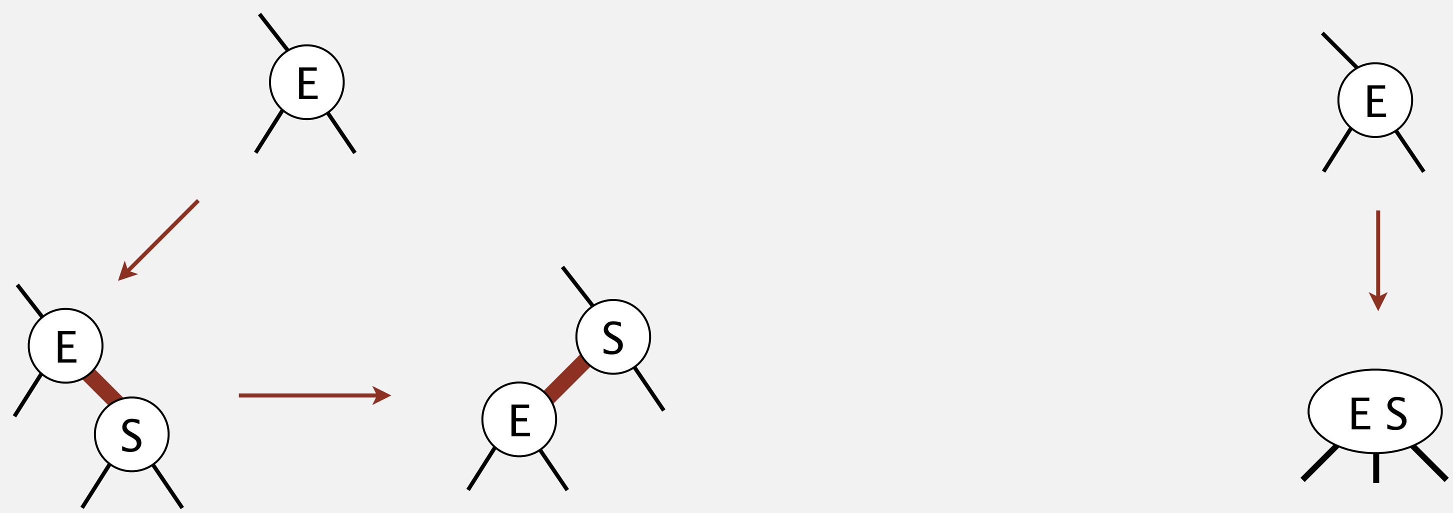
Thought experiment on right insertions

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*.



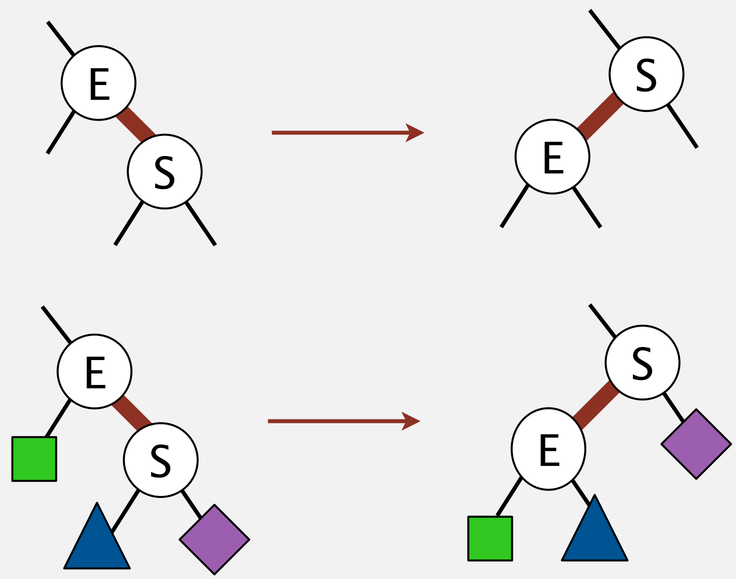
Easy Case 2: Inserting to the right of a 2-node

What is the problem here?

- Red links must lean left (by definition)

How do we fix the problem?

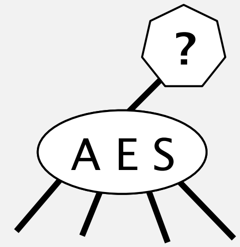
- Swap roles of S and E
  - Can generalize role-swapping for non-leaf nodes as *left rotation*.
  - Usefulness of rotation will become clear.



More general approach

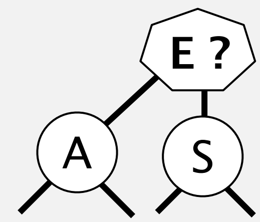
2-3 Tree Violations

- Existence of 4-nodes.



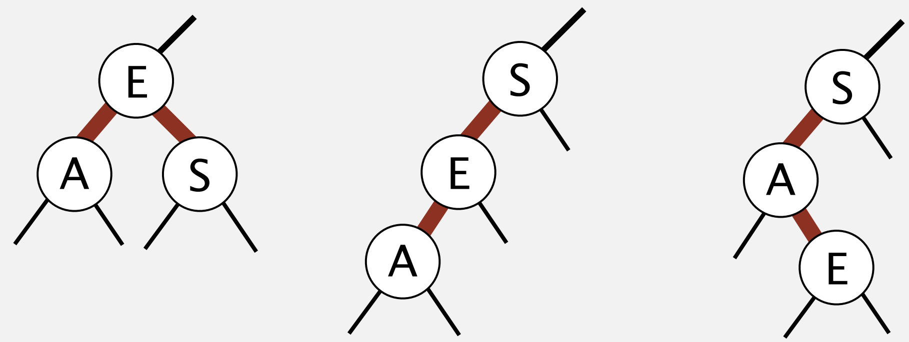
Operations for fixing 2-3 tree violations

- Splitting a 4 node.



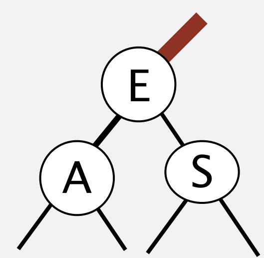
LLRB Violations

- Two red children.
- Two consecutive red links.
- Right red child (breaks LL rule).



Operations for fixing LLRB violations

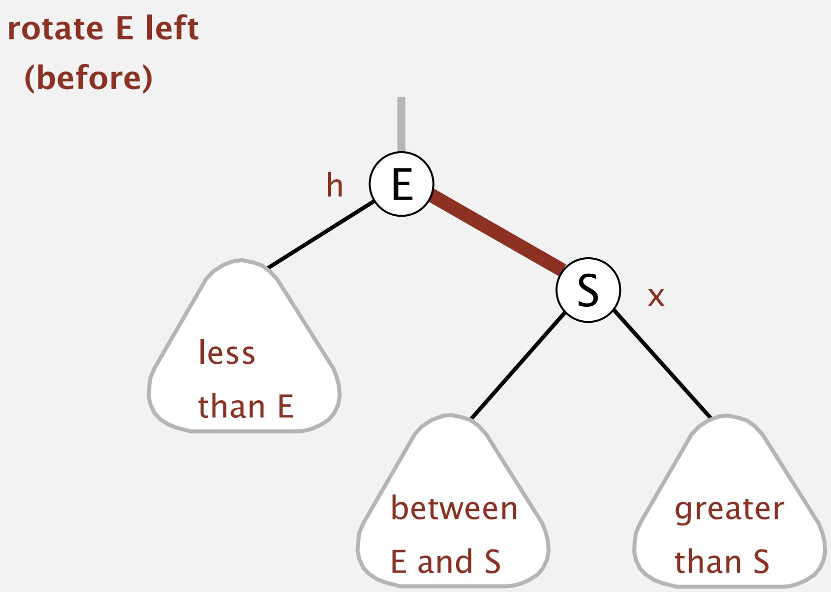
- Left rotation.
- Right rotation.
- Color flipping.



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

Elementary red-black BST operations

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

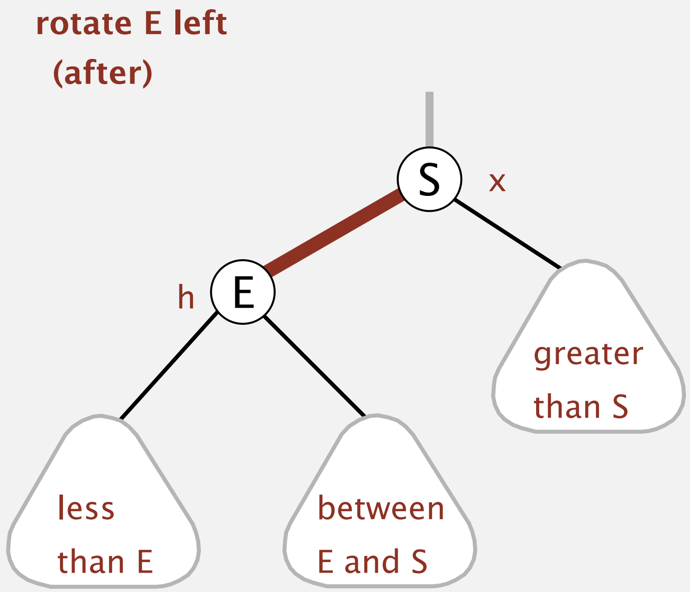



```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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.

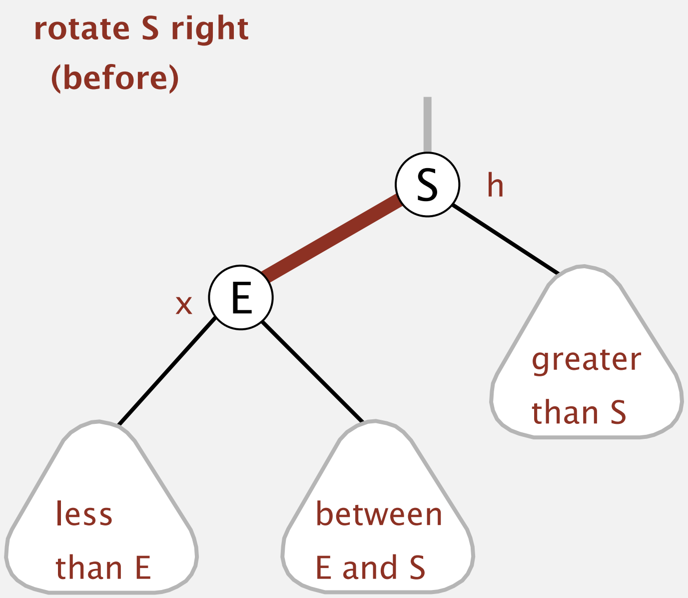



```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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.

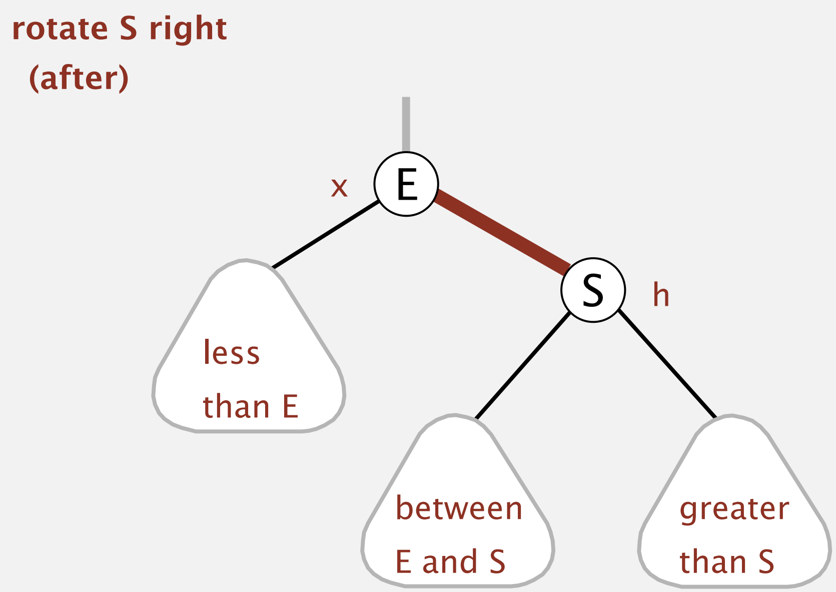



```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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.



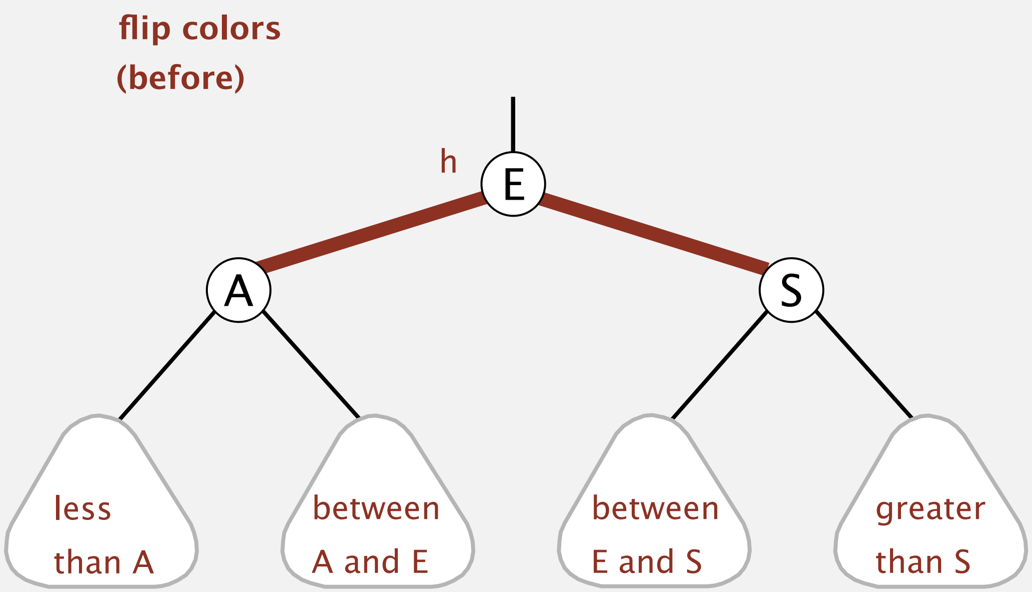

```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

The node being rotated always ends up lower!

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

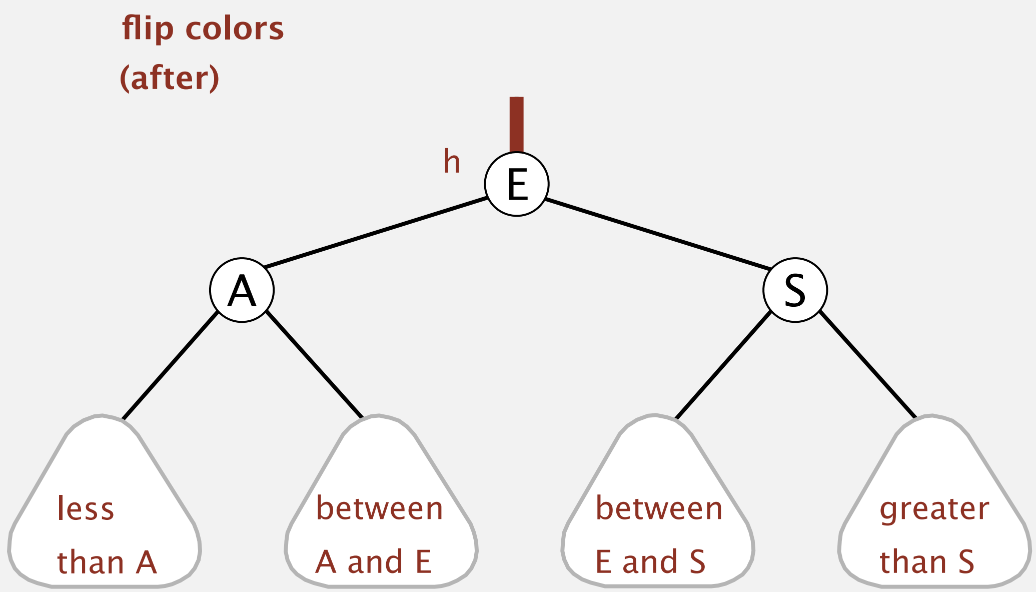



```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

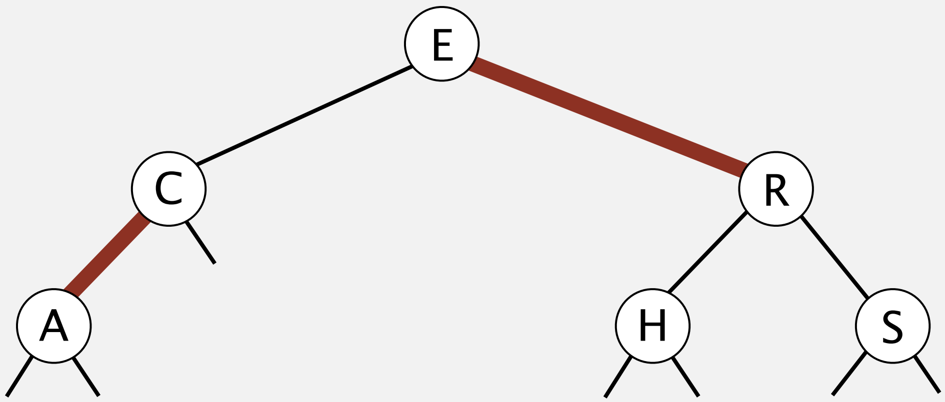
Color flip. Recolor to split a (temporary) 4-node.




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

Invariants. Maintains symmetric order and perfect black balance.

right link red
(rotate E left)



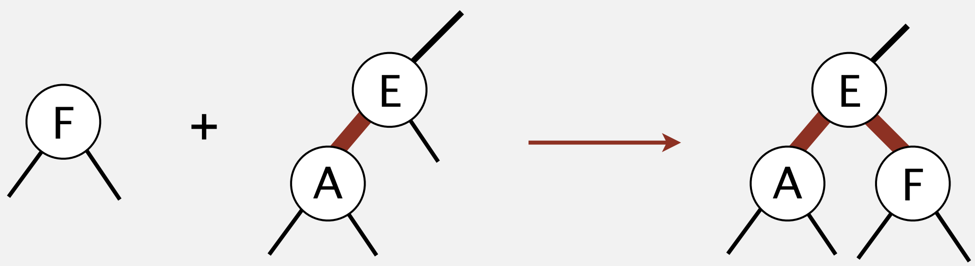
Case 1: Two red children

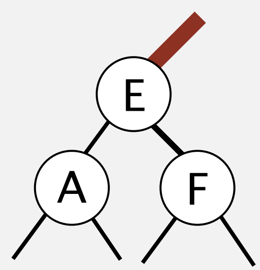
What is the problem here?

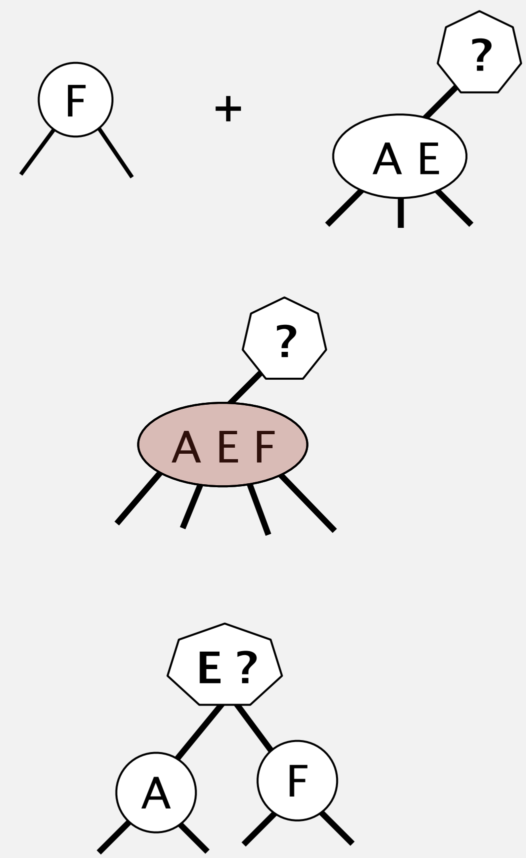
- [LLRB VIOLATION] Two red links touching a single node.
- [2-3 VIOLATION] 4 node.

How to Resolve?

- Color flip.
- Equivalent to splitting 4 node.







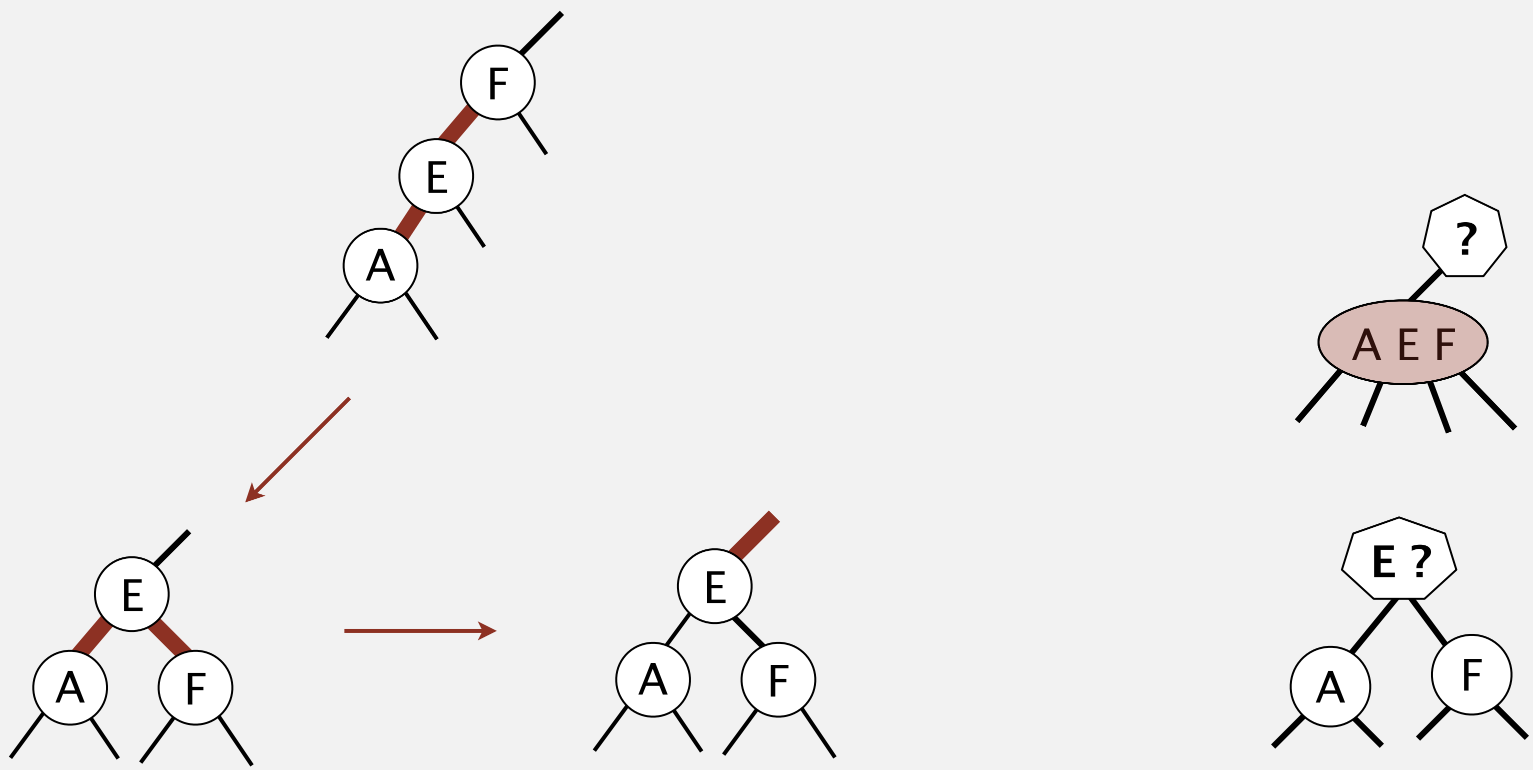
Case 2: Consecutive red left children

What is the problem here?

- [LLRB VIOLATION] Two red links touching a single node.
- [2-3 VIOLATION] 4 node.

How to Resolve?

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



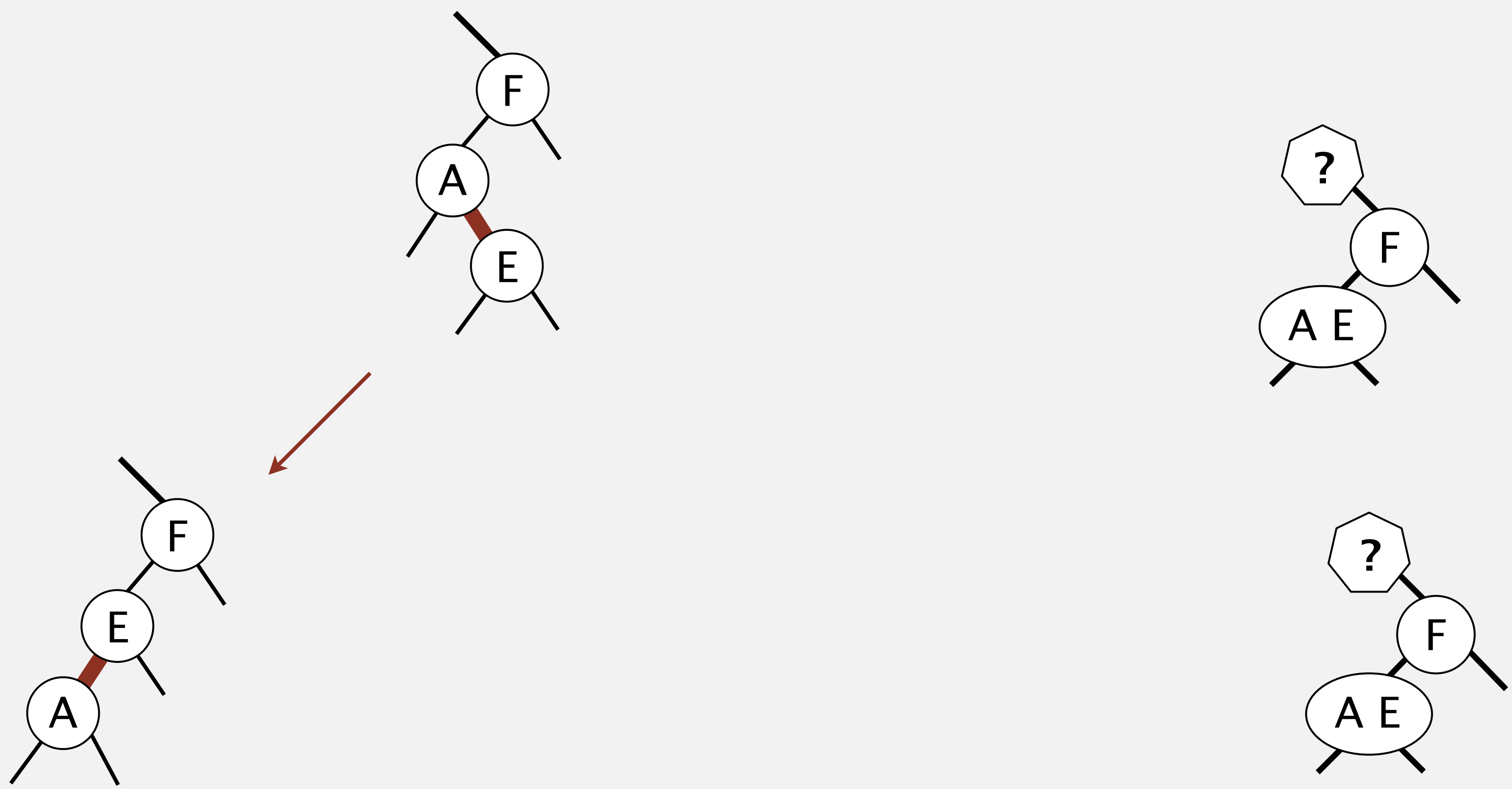
Case 3a: Red right child and black left child (alternate)

What is the problem here?

- [LLRB VIOLATION] Red link leans right.
- No 2-3 violation.

How to Resolve?

- Rotate A left. Done.



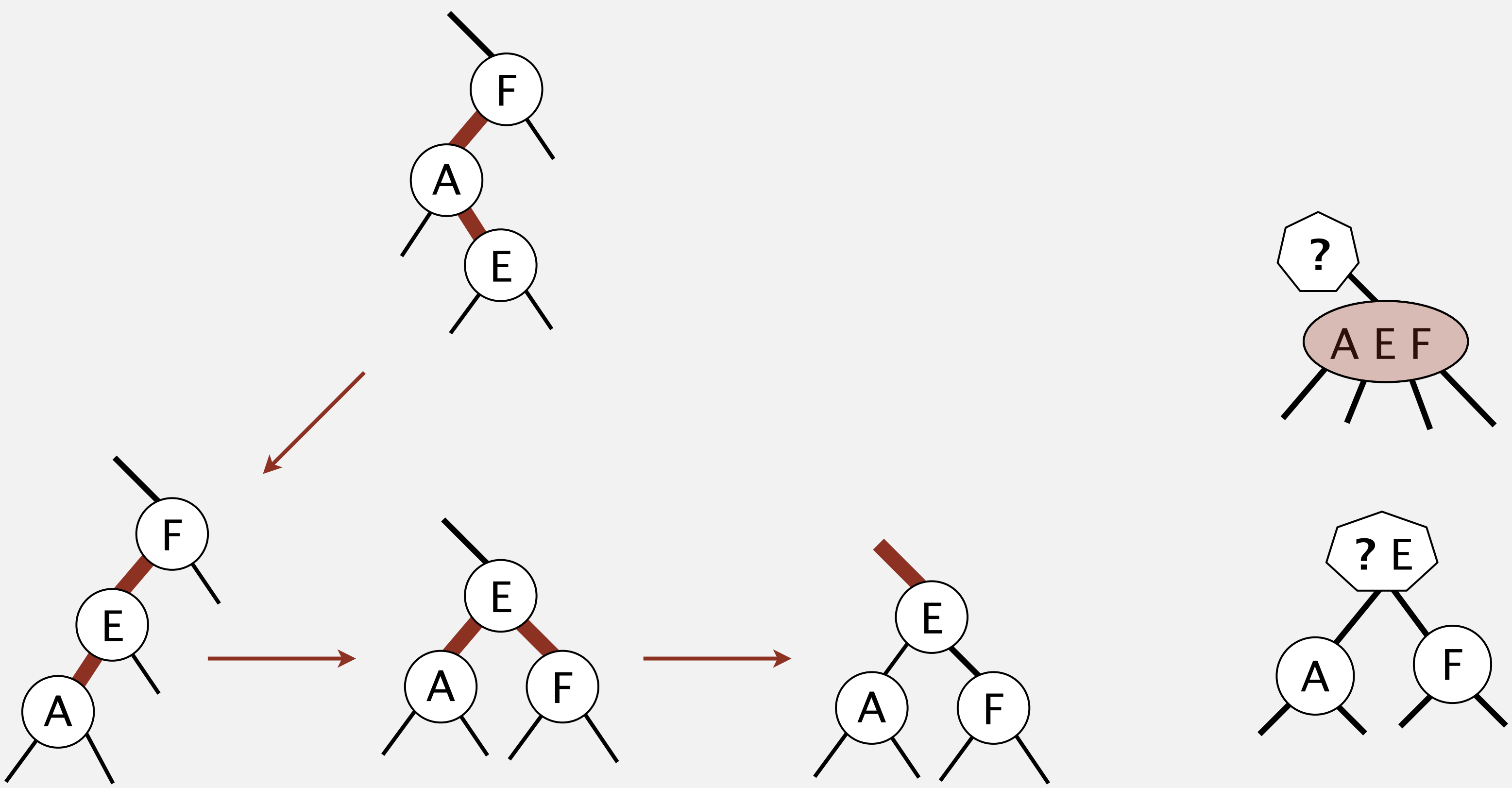
Case 3b: Red right child and black left child

What is the problem here?

- [LLRB VIOLATION] Two red links touching a single node.
- [2-3 VIOLATION] 4 node.

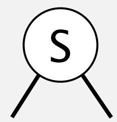
How to Resolve?

- Rotate A left. Puts us right back into Case 2.

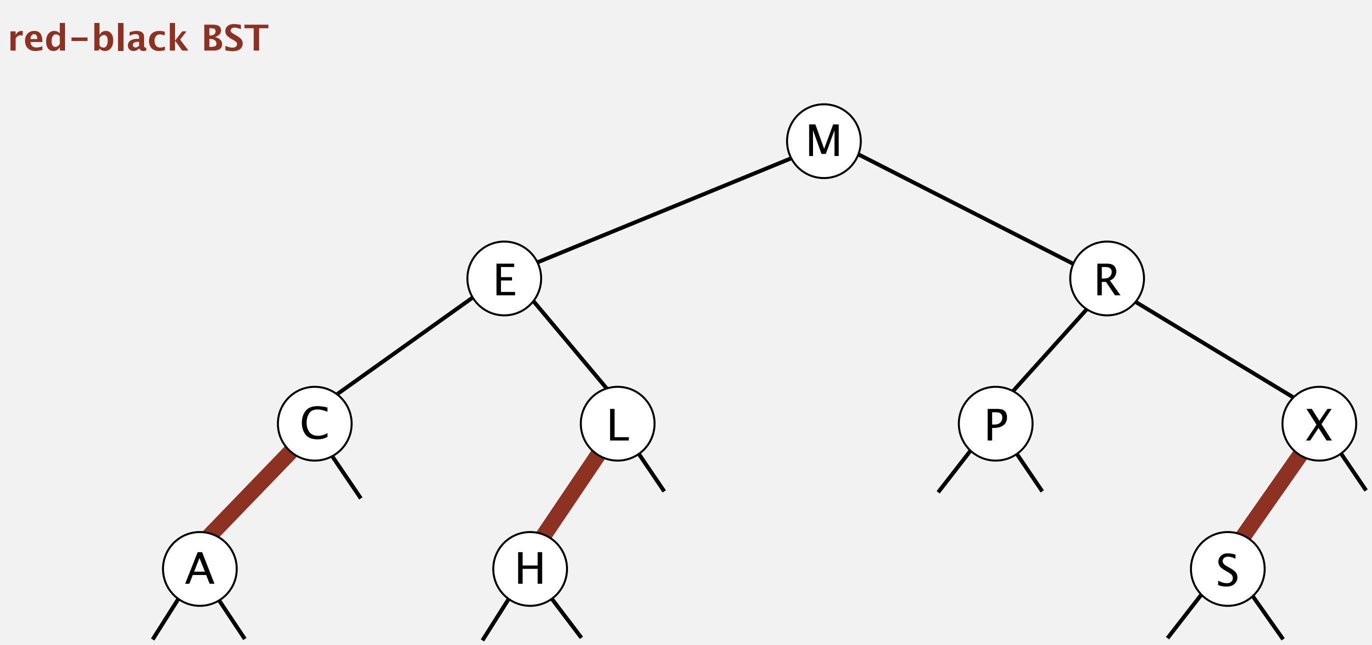


Red-black BST construction demo

insert S



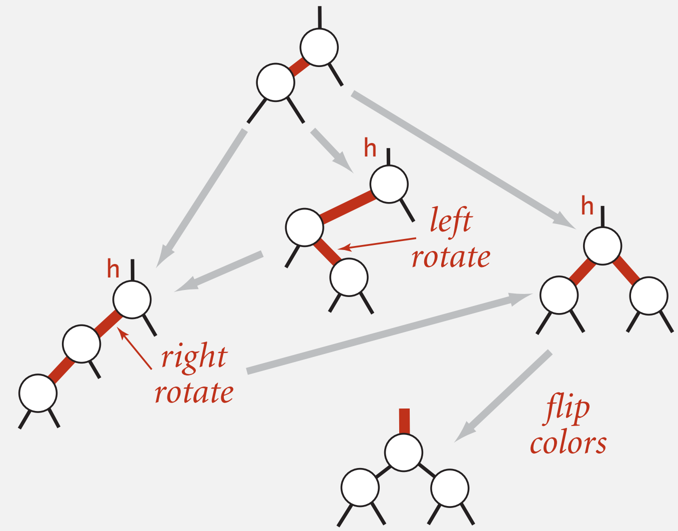
Red-black BST construction demo



Insertion in a LLRB tree: Java implementation

Same code for all cases.

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




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

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

   return h;
}
```

insert at bottom (and color it red)

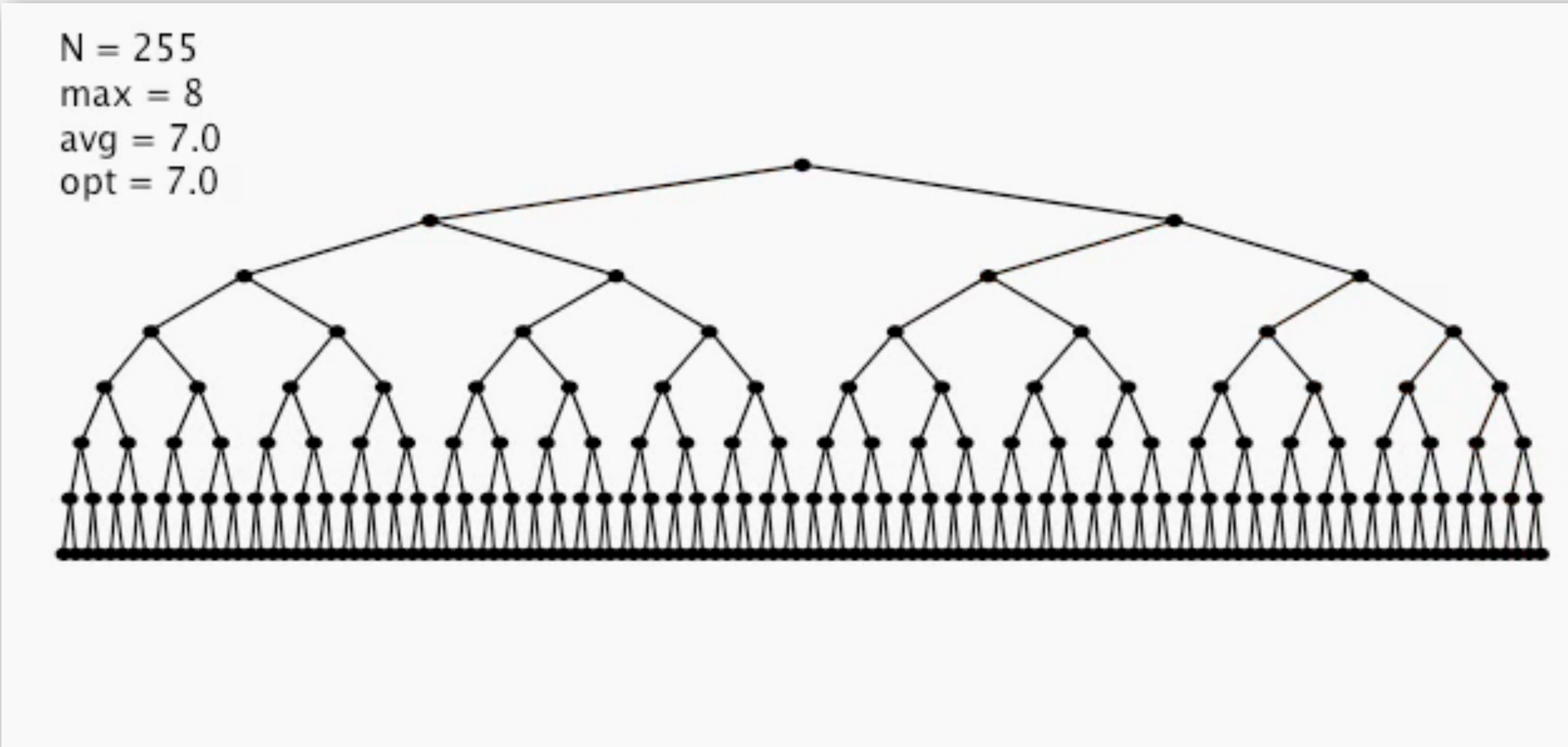
lean left

balance 4-node

split 4-node

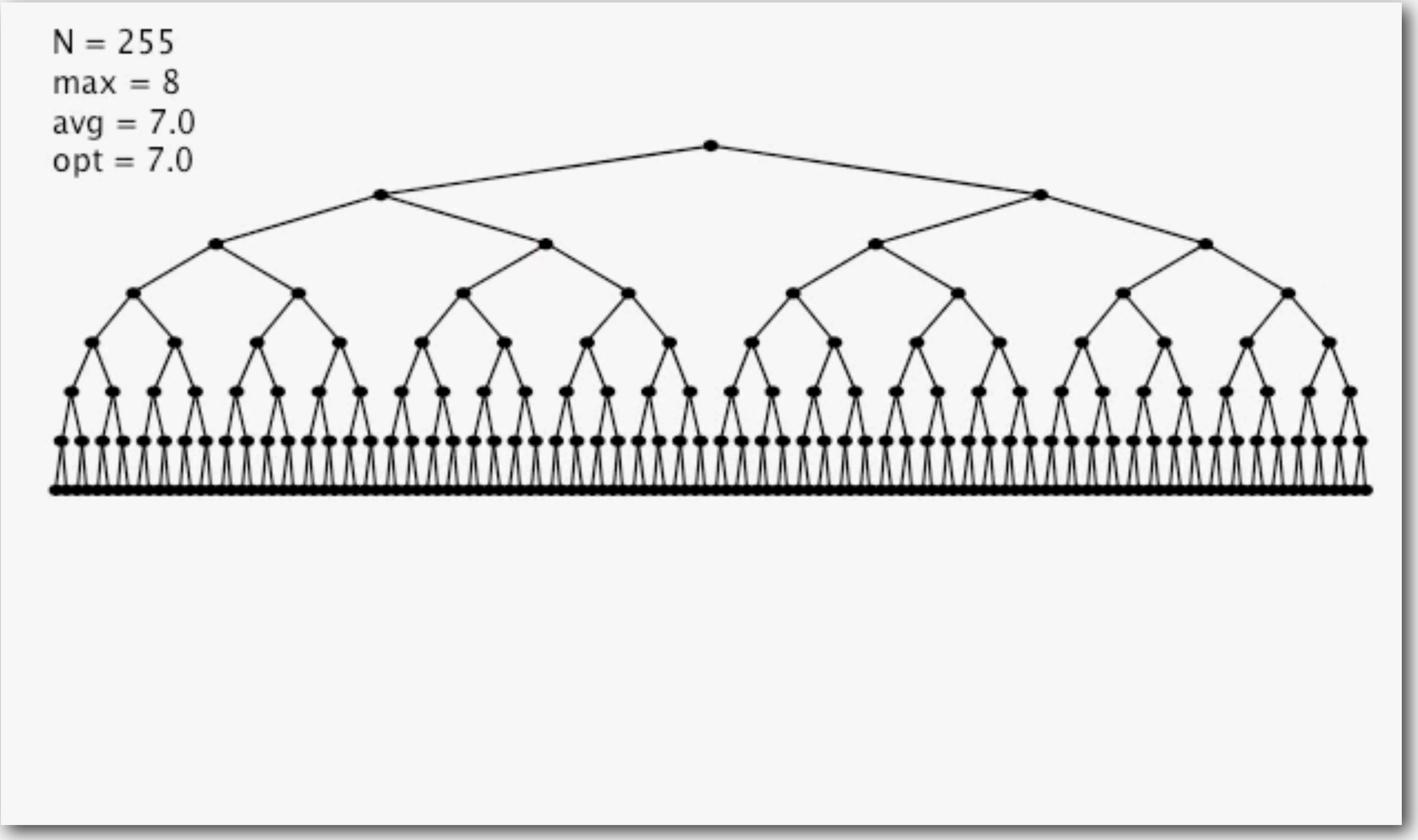
only a few extra lines of code provides near-perfect balance

Insertion in a LLRB tree: visualization



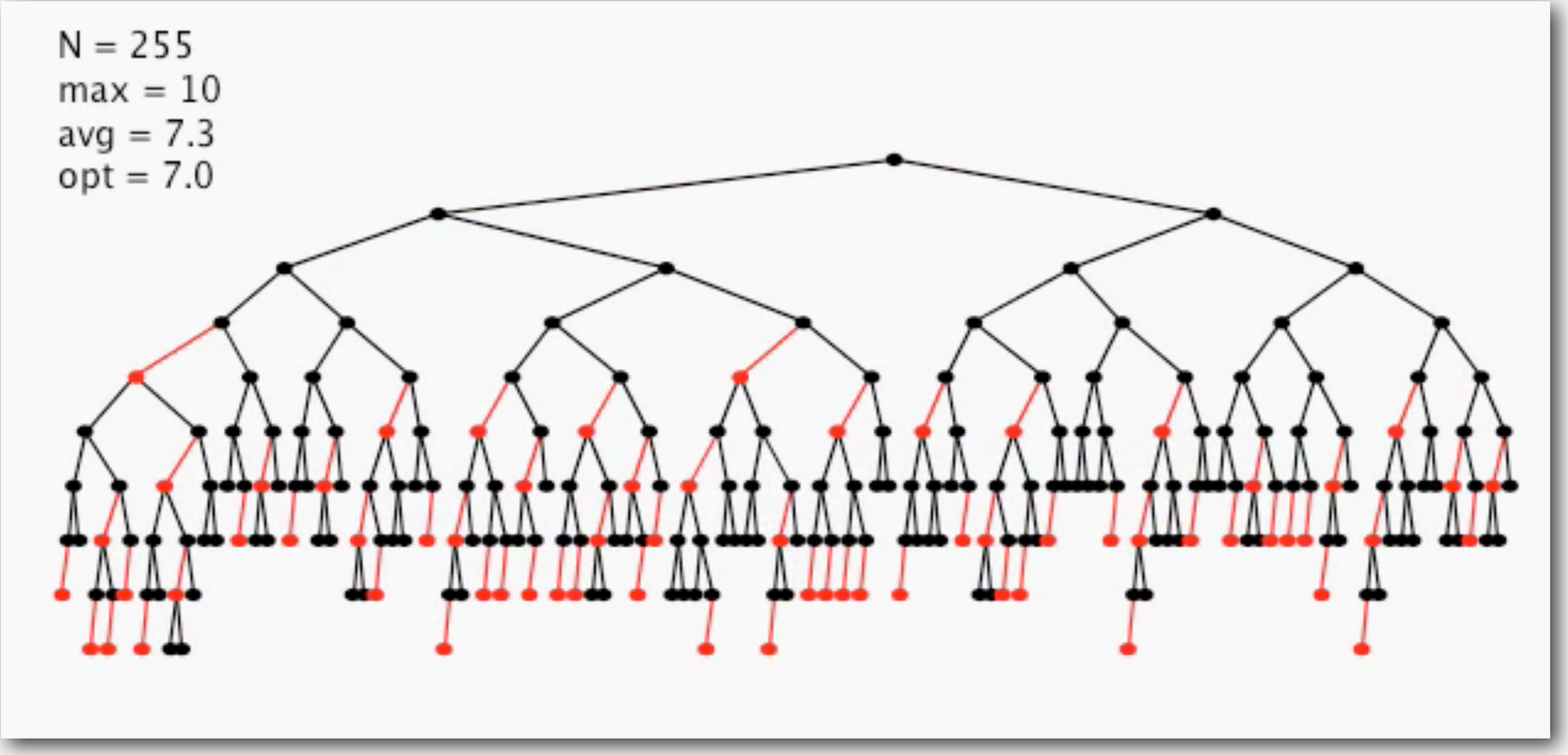
255 insertions in ascending order

Insertion in a LLRB tree: visualization



255 insertions in descending order

Insertion in a LLRB tree: visualization



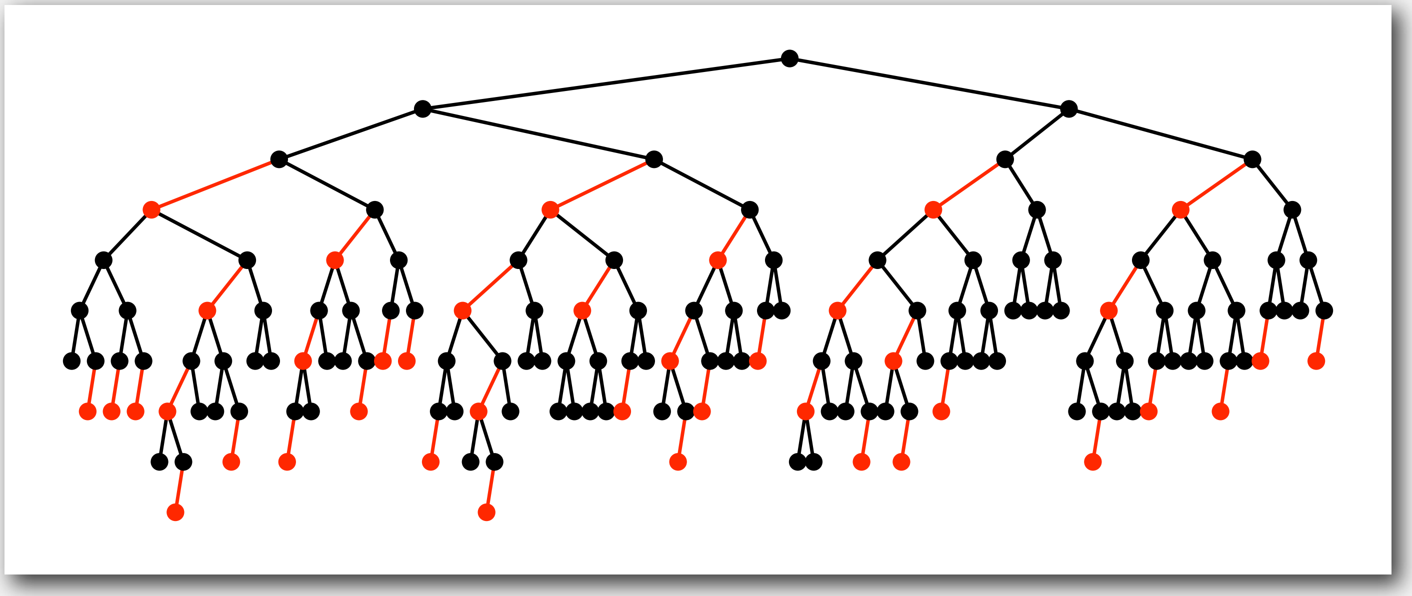
255 random insertions

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 (no deletes)	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

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.

Hibbard deletion was the problem

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


[redacted]

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

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

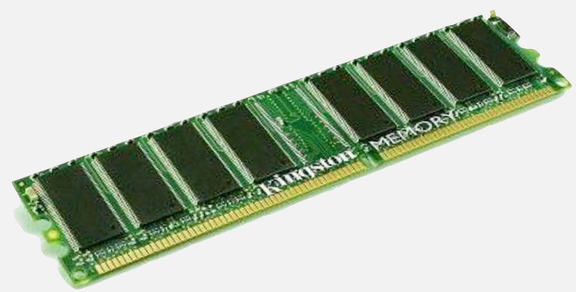
- 2-3 search trees
- red-black BSTs
- *B-trees (optional)*

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.

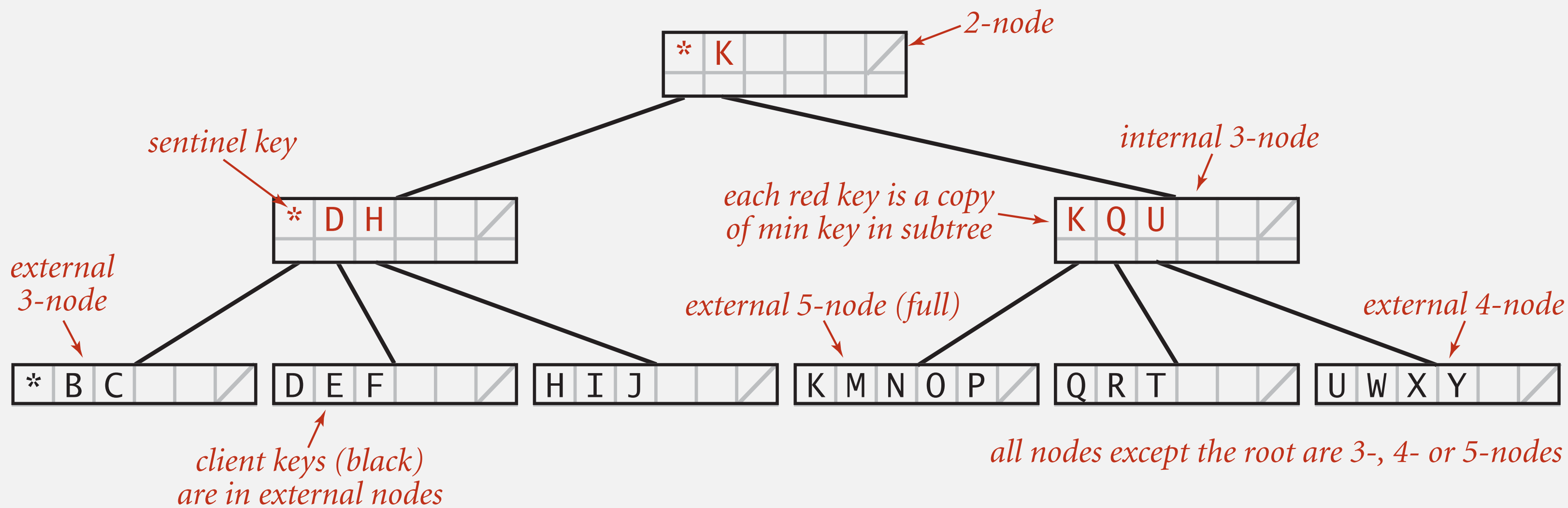
Goal. Access data using minimum number of probes.

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.

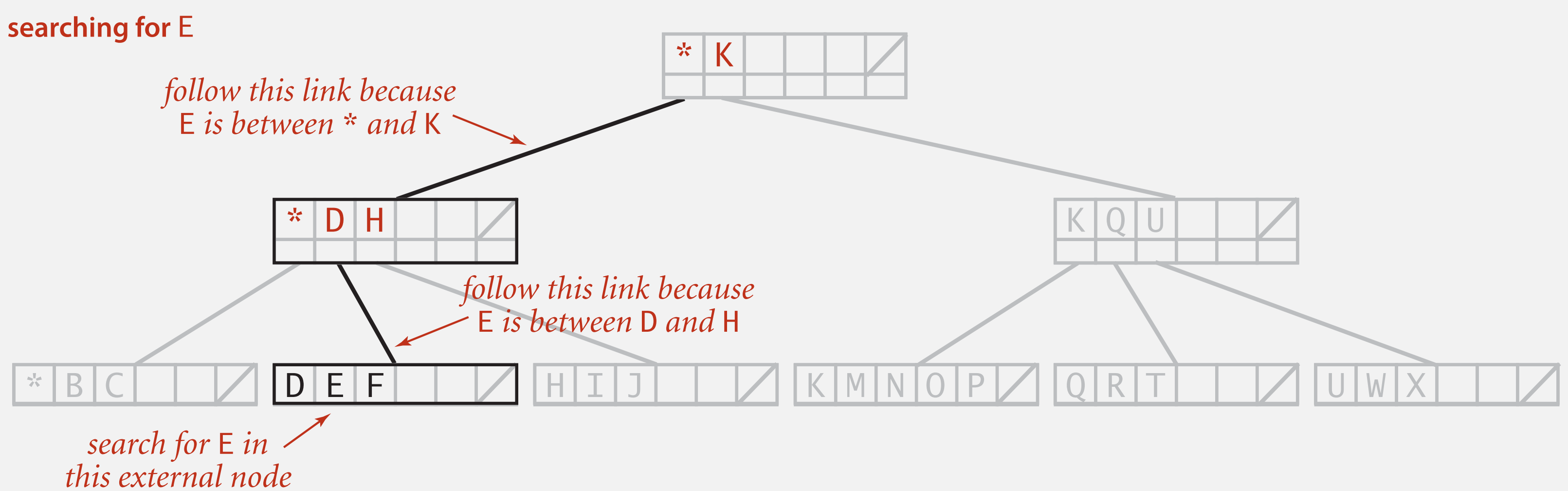
choose M as large as possible so that M links fit in a page, e.g., M = 1024



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

Searching in a B-tree

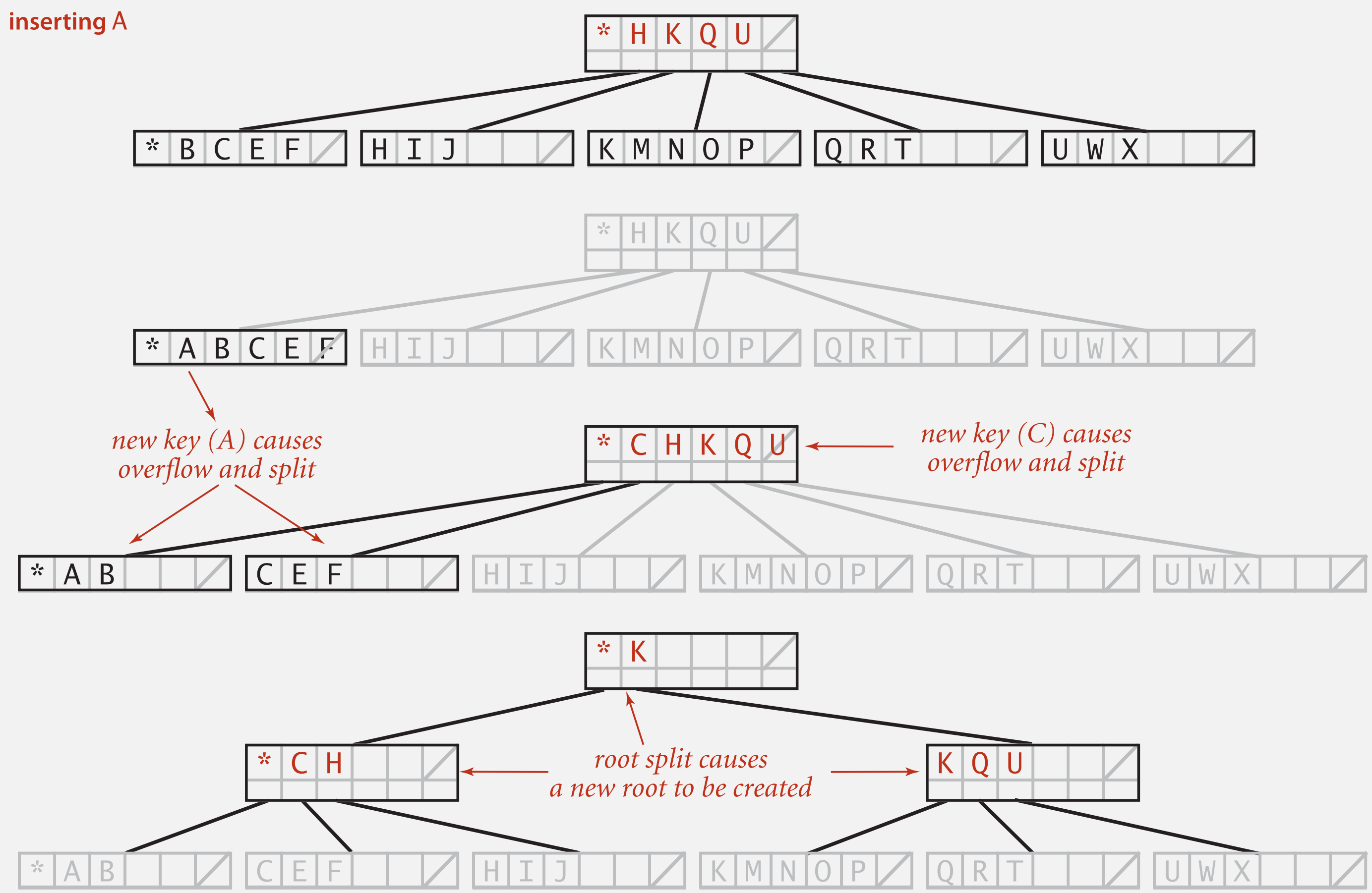
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



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.

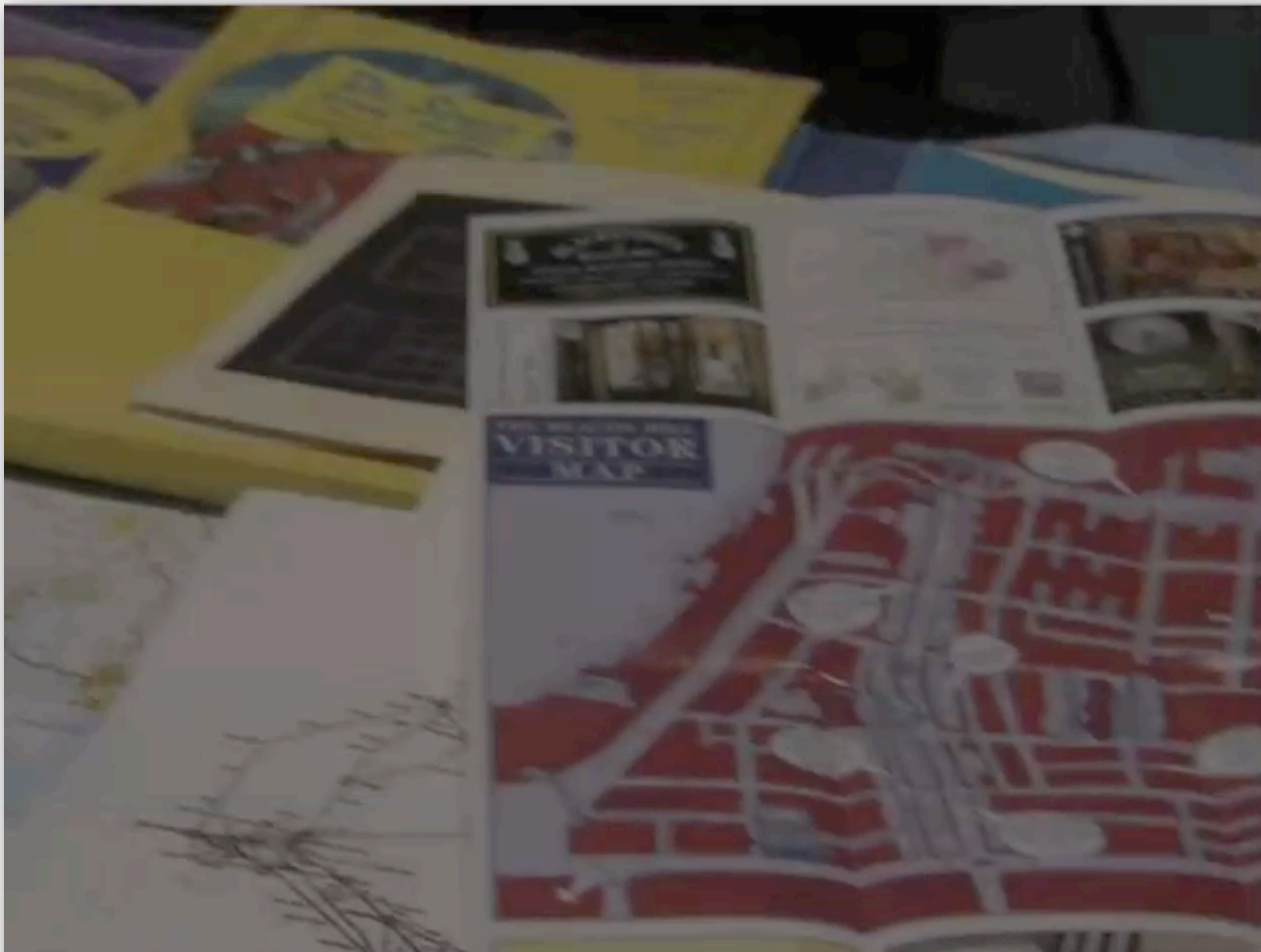
- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.
- Emacs: conservative stack scanning.

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.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.

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?