INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

BALANCED SEARCH TREES

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



http://ds.cs.rutgers.edu

Some slides Adopted and modified from Sedgewick and Wayne

Last updated: Sun 9/11/22 1:36 PM EDT

Motivation - Symbol table review (Optional)

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 and left-leaning red-black BSTs.

optimized for teaching and coding;
introduced to the world in this course!

simplified RBT proposed in 2008 by Sedgewick

BALANCED SEARCH TREES

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

2–3 tree

Allow 1 or 2 keys per node.

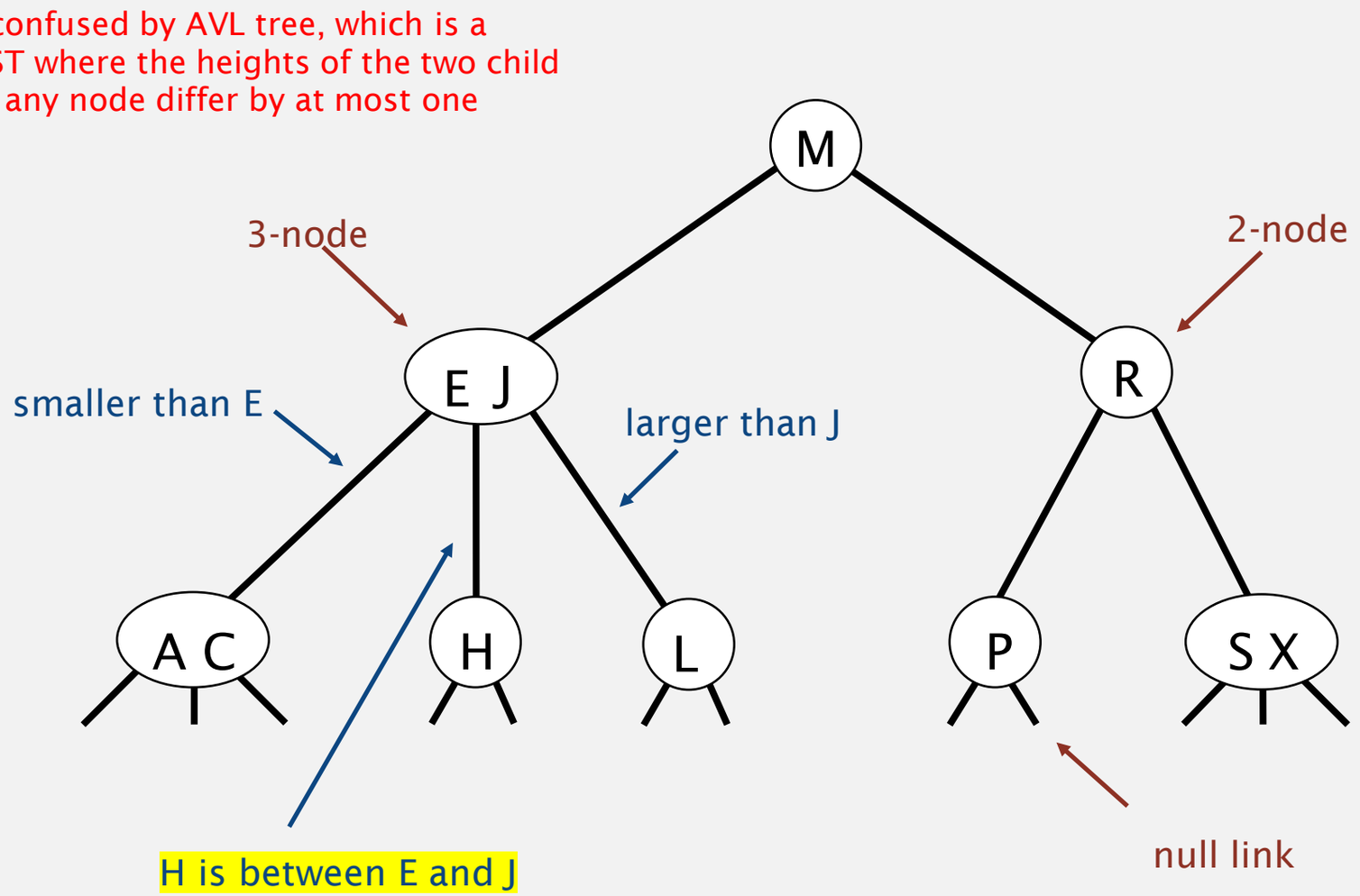
- 2-node: one key, two children.
- 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.

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

NOT to be confused by AVL tree, which is a balanced BST where the heights of the two child subtrees of any node differ by at most one

how to maintain?



2–3 tree demo

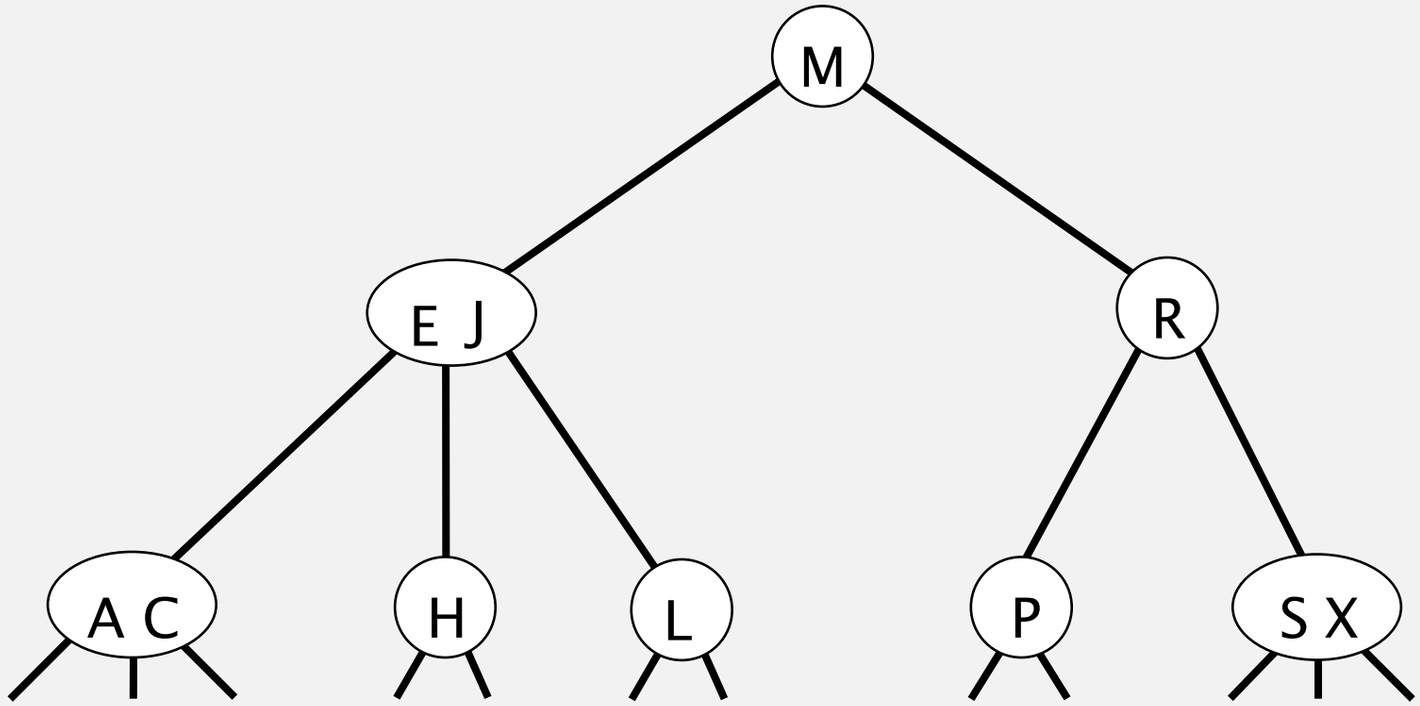
Search.

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

search for H //search hit

search for B //search miss

start at root node

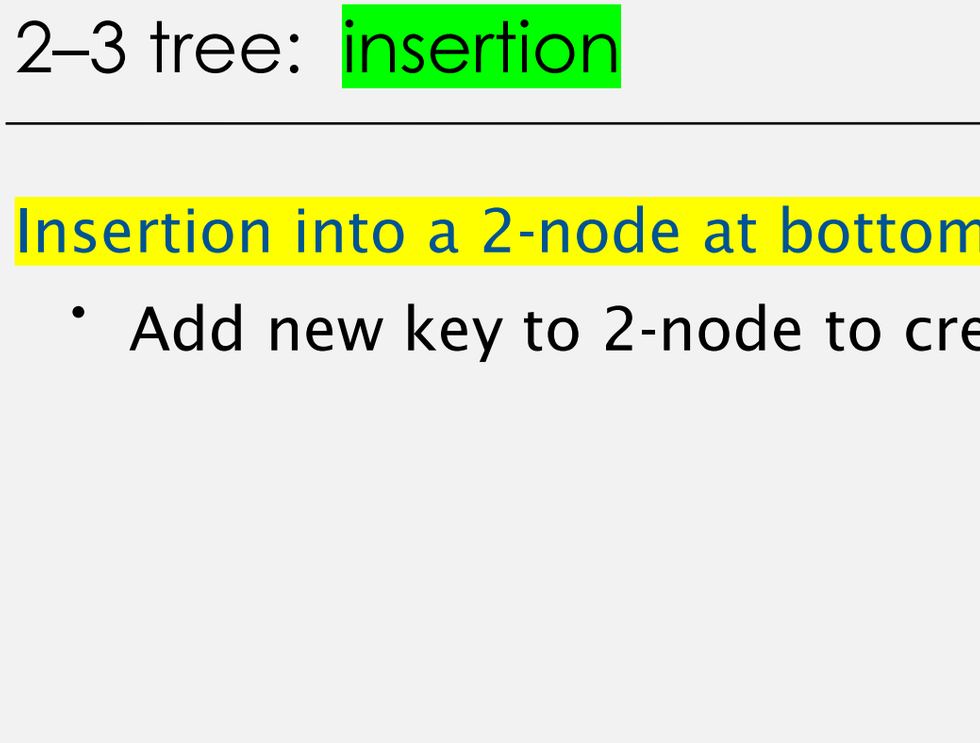


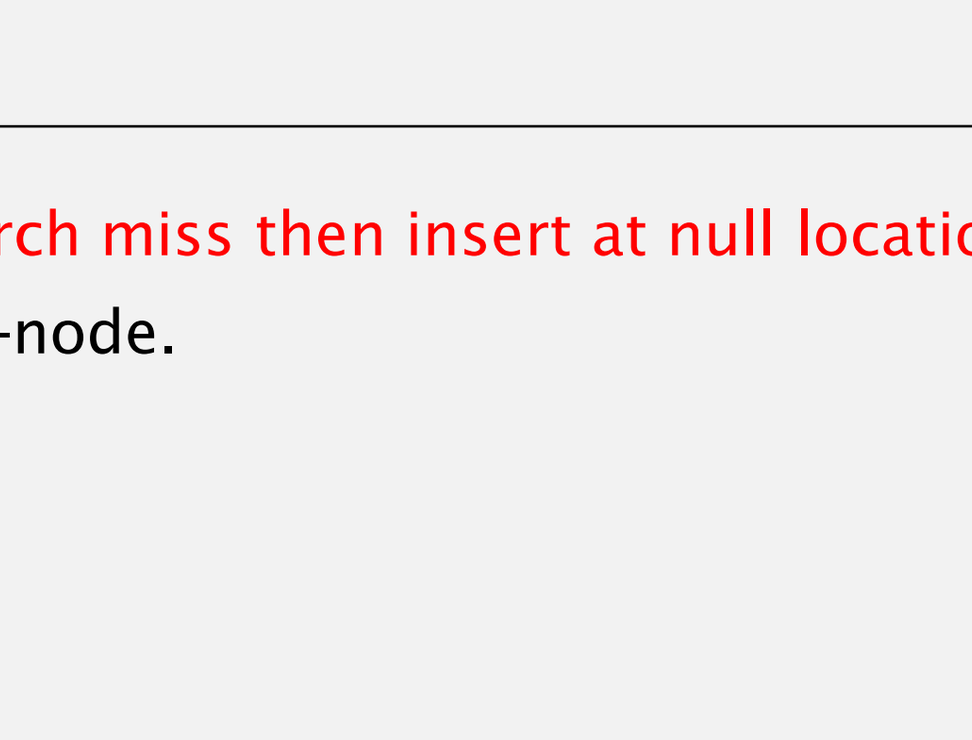
2–3 tree: insertion

Insertion into a 2-node at bottom. //search miss then insert at null location

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

insert G



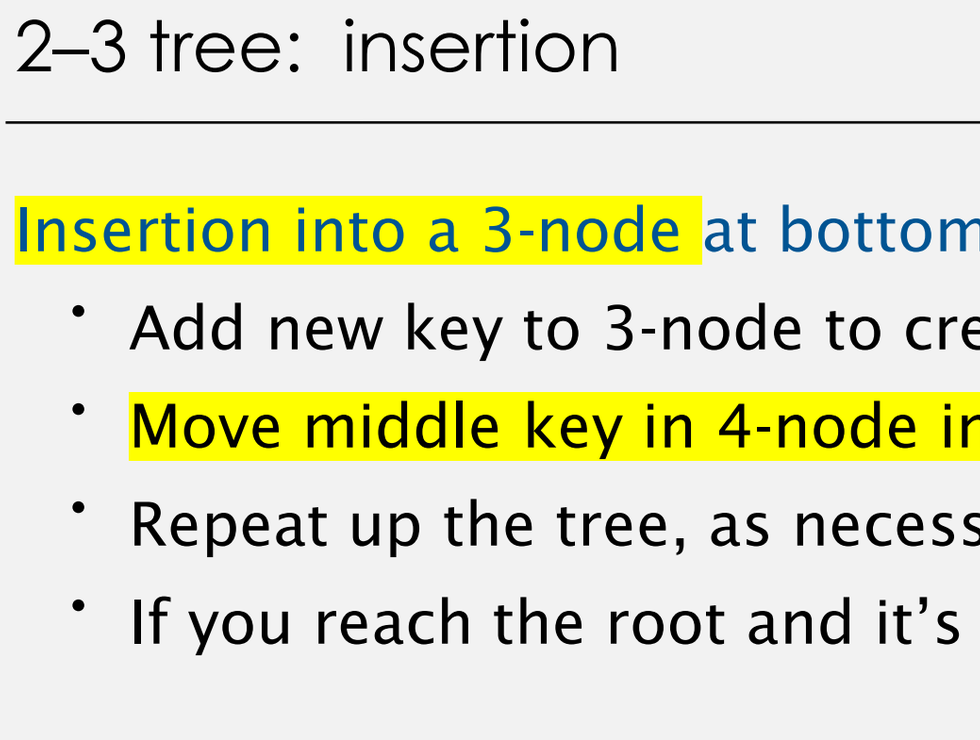


2–3 tree: insertion

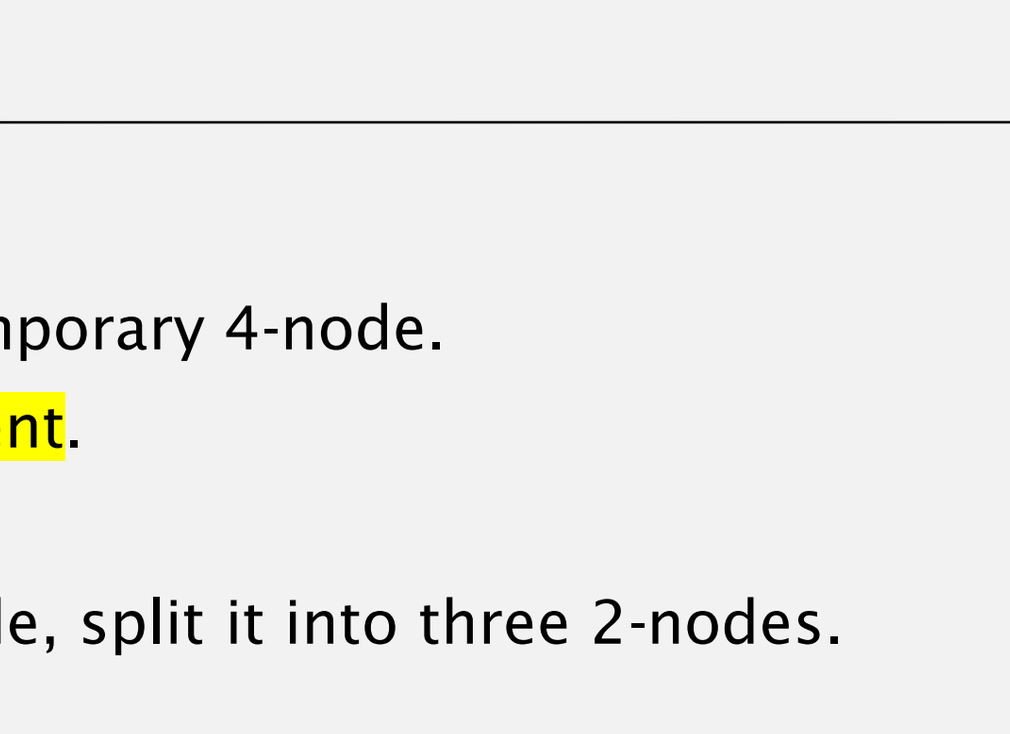
Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

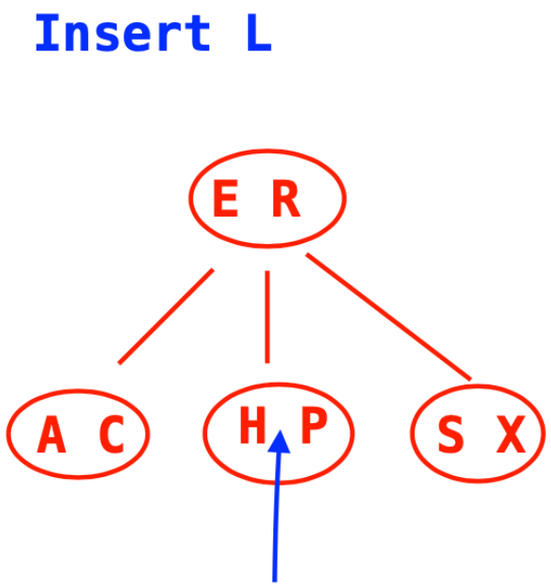
insert Z



search miss



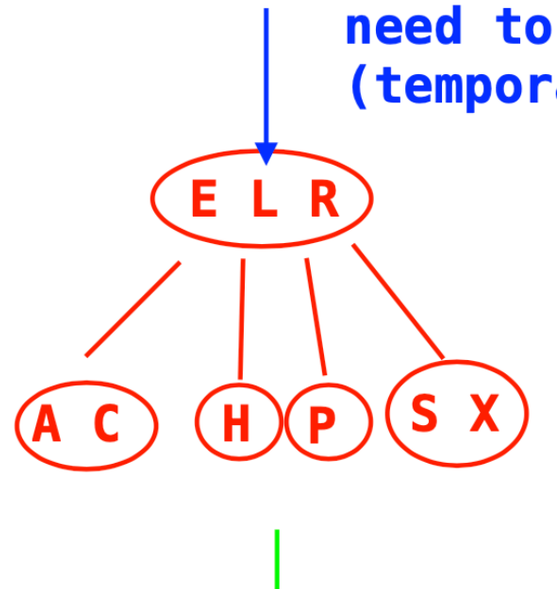
Another Example



L goes here, but need to move up

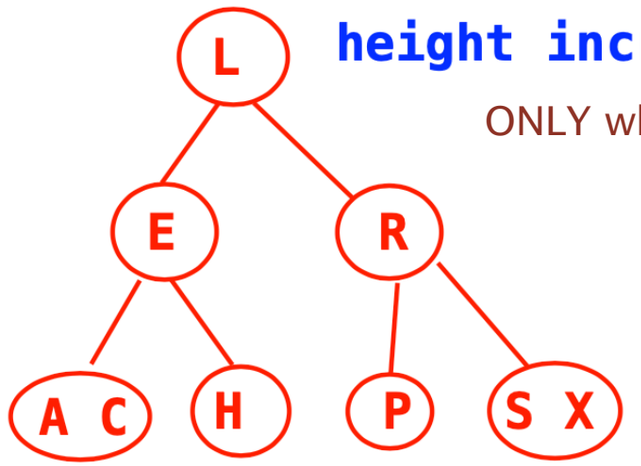
split H and P,

move up L



split E and R,

move up L



height increases by 1

ONLY when the root split

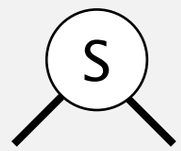
tree height grows BOTTOM-UP

2–3 tree construction demo

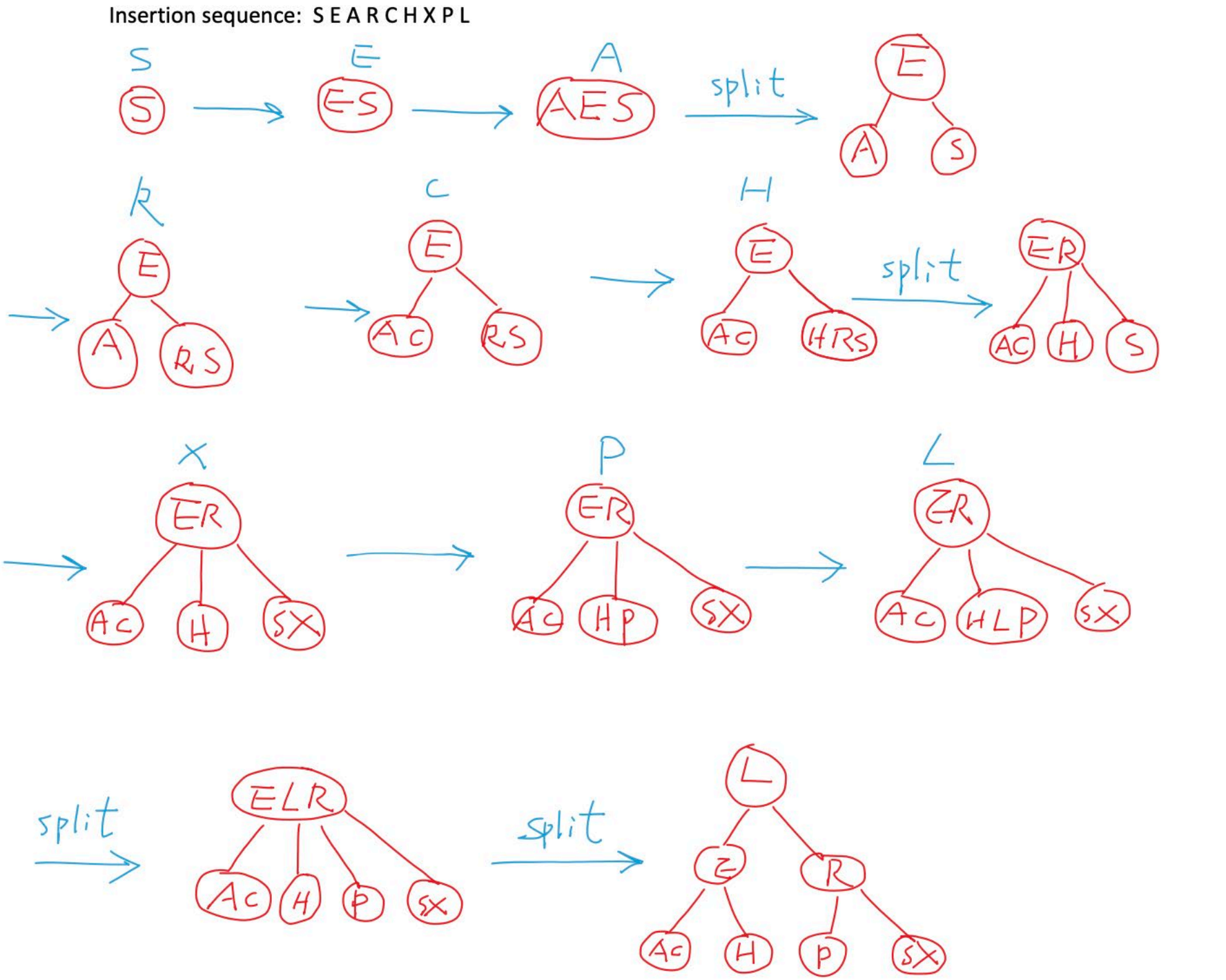


insert S

Insertion Sequence: S E A R C H X P L

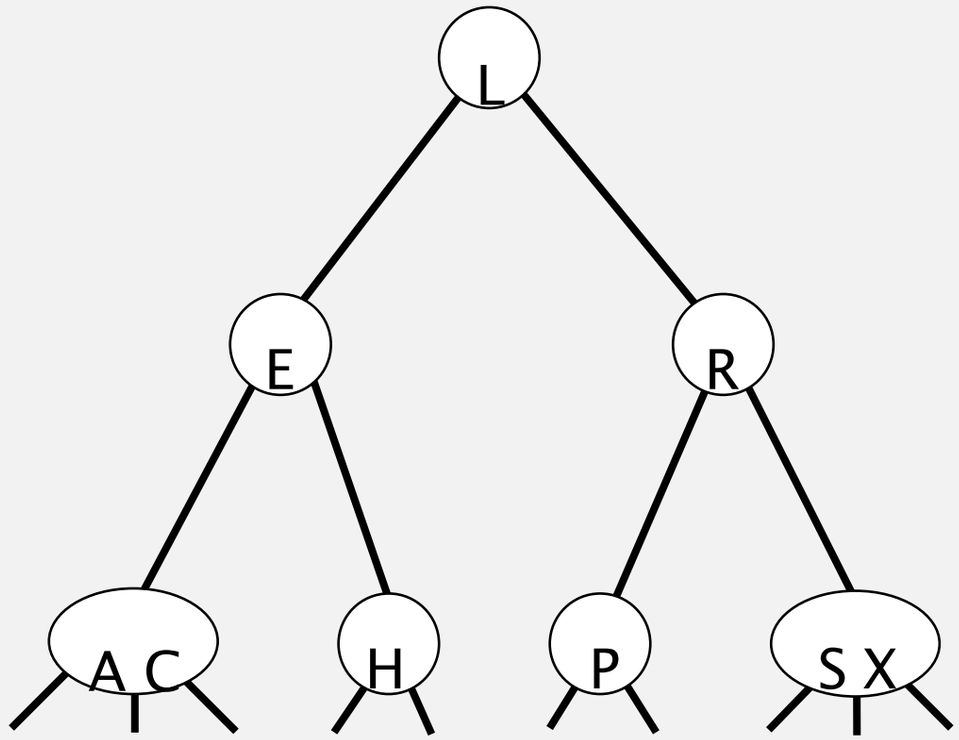


Example -- Building a 2-3 Tree



2–3 tree construction demo

2–3 tree



2–3 tree: global properties

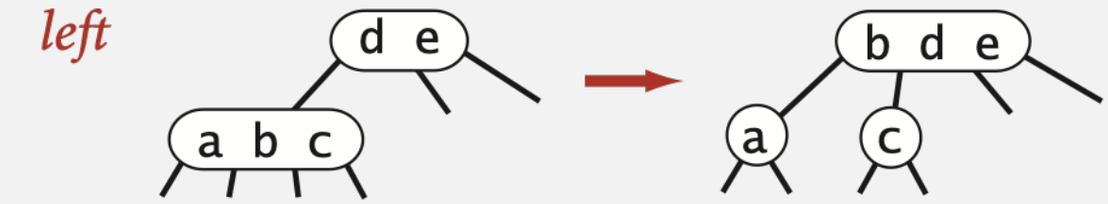
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.

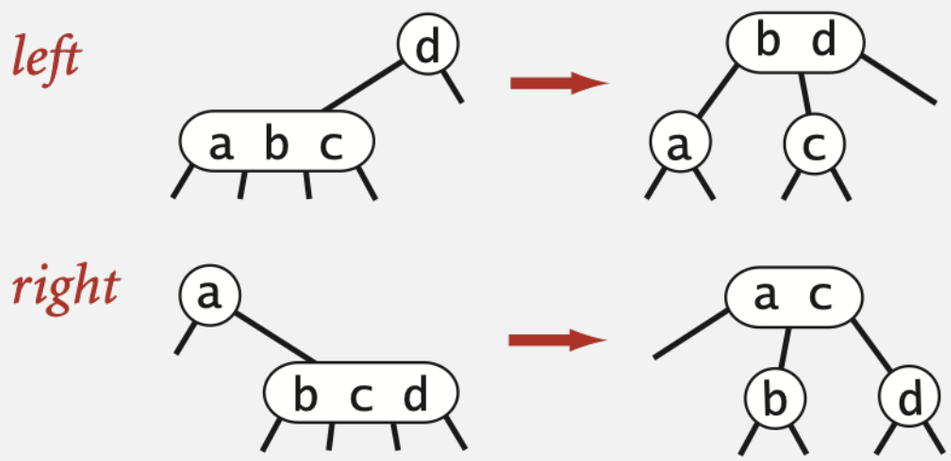
root

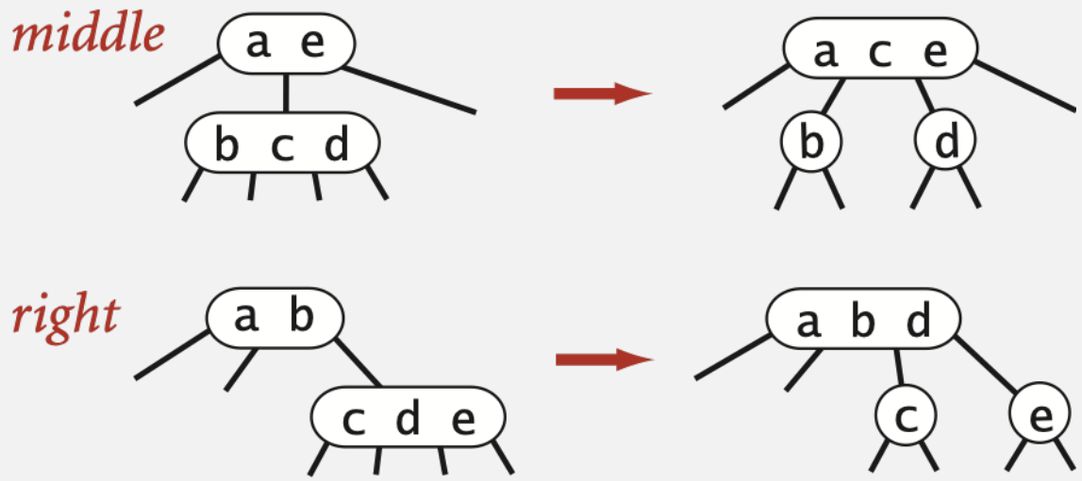


parent is a 3-node



parent is a 2-node

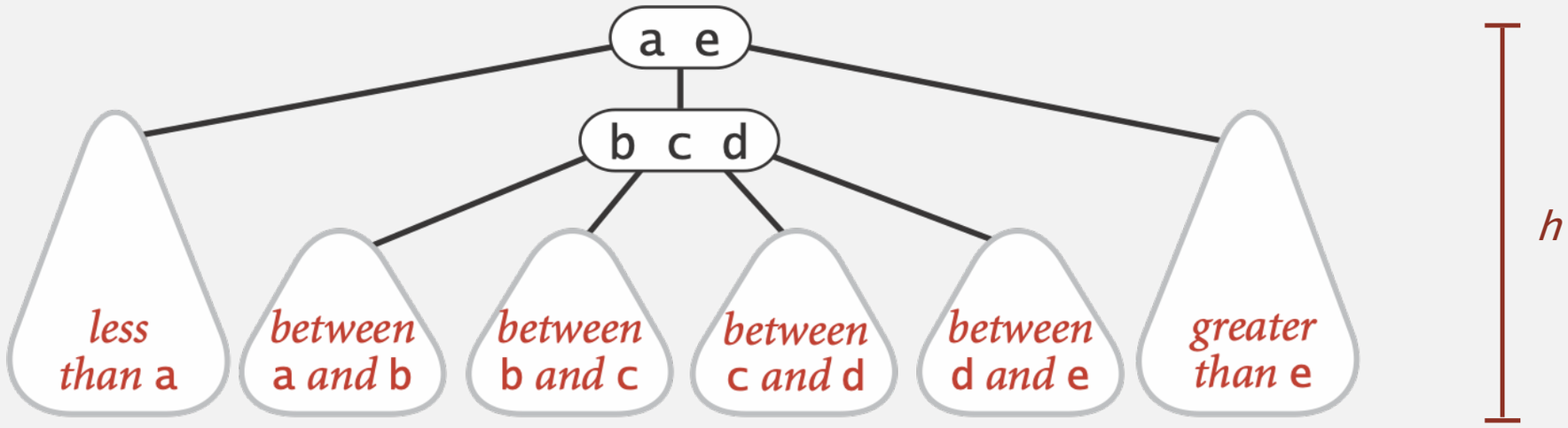


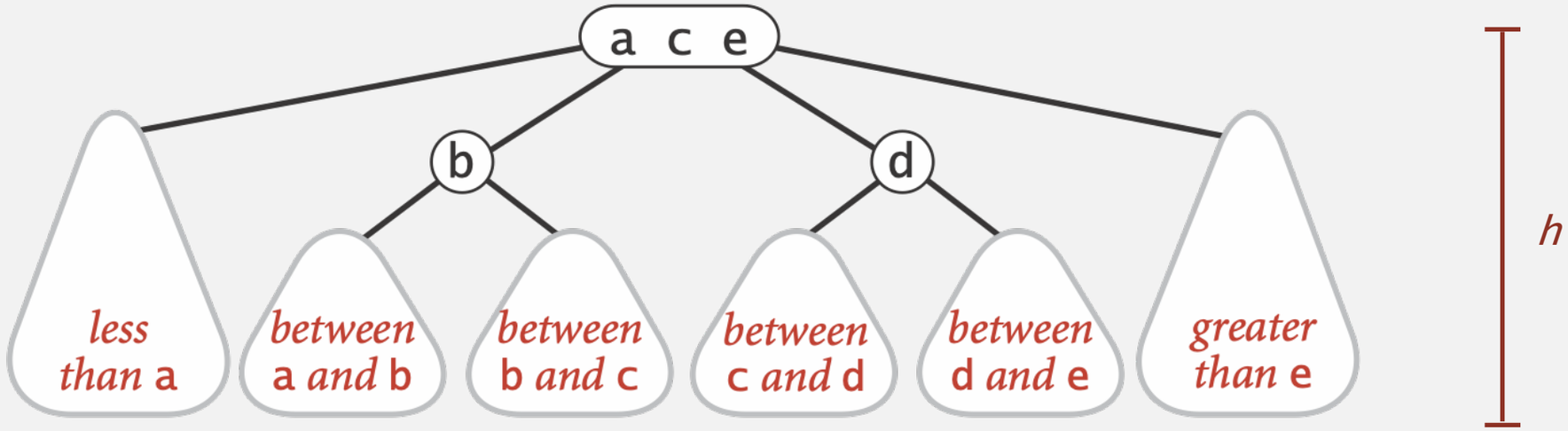


2–3 tree: performance

does not change the global property "perfect balance"

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





The length from the root to any null link doesn't change after the transformation.

Only when the root splits into three 2-nodes, the length of every path from the root to a null link increases by 1 (see previous slide).

Balanced search trees: quiz 1

What is the maximum height of a 2–3 tree with n keys?

A. $\sim \log_3 n$

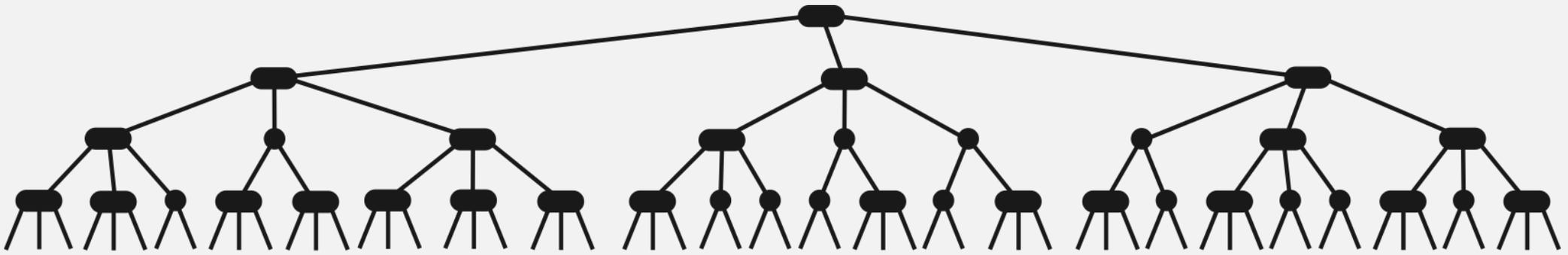
B. $\sim \log_2 n$

C. $\sim 2 \log_2 n$

D. $\sim n$

2–3 tree: performance

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



Tree height.

- Worst case: $\log_2 n$. [all 2-nodes]
- Best case: $\log_3 n \approx .631 \log_2 n$. [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 c 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. //tree grows from bottom up
- Large number of cases for splitting.

fantasy code

```
public void put(Key key, Value val) //insertion
{
   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);
}
```

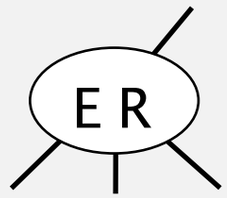
Bottom line. Could do it, but there's a better way.

BALANCED SEARCH TREES

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

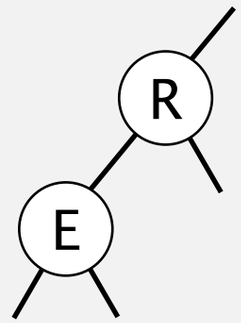
How to implement 2–3 trees with binary trees?

Challenge. How to represent a 3 node?



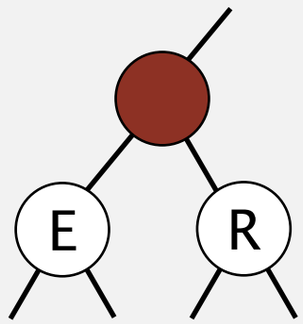
Approach 1. Regular BST.

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



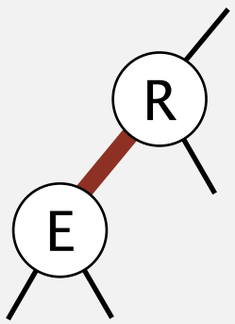
Approach 2. Regular BST with red "glue" nodes.

- Wastes space for extra node.
- Code probably messy.



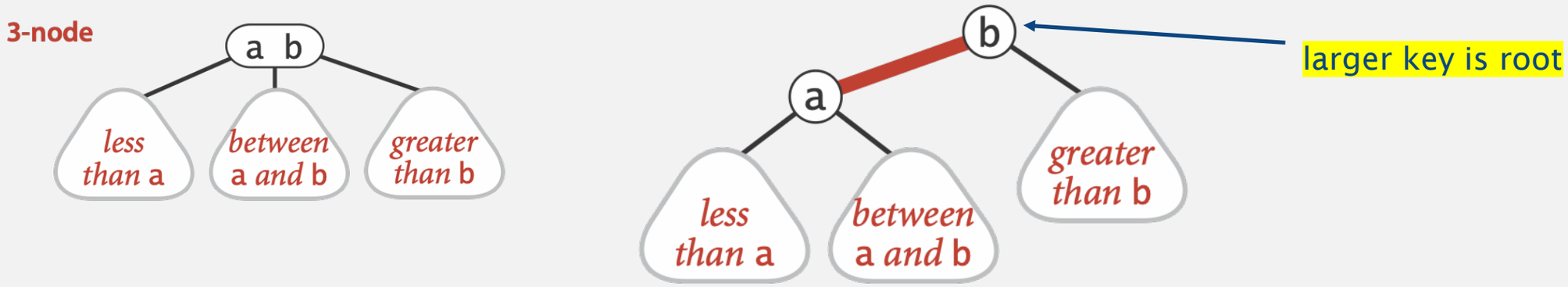
Approach 3. Regular BST with red "glue" links.

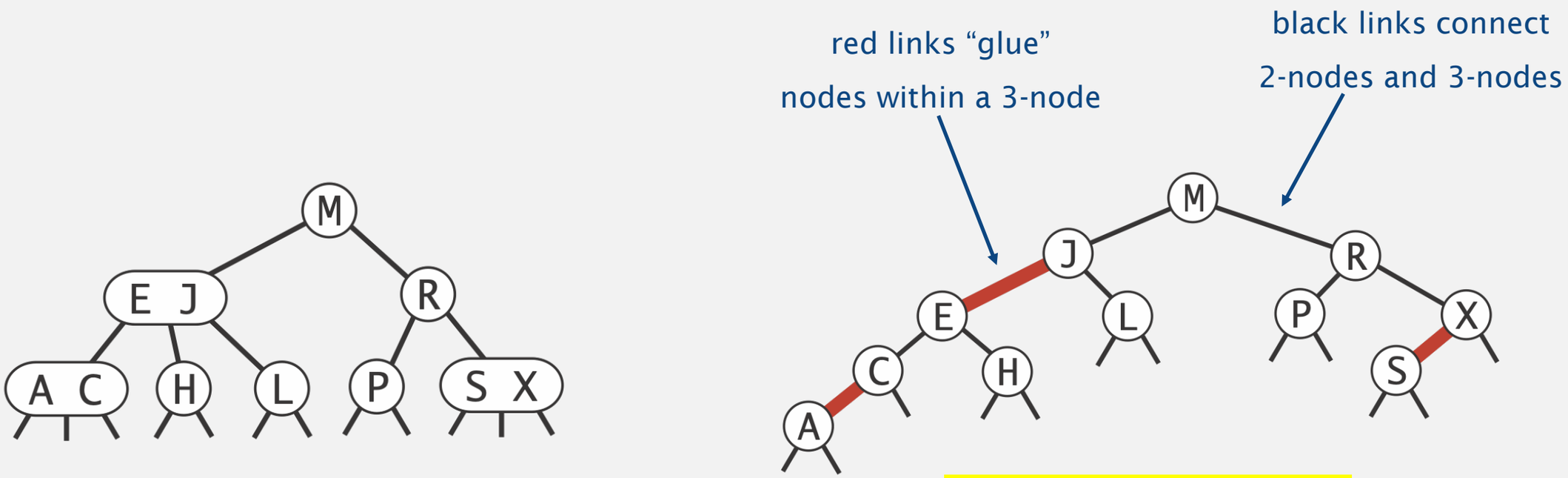
- Widely used in practice.
- Arbitrary restriction: red links lean left.



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.





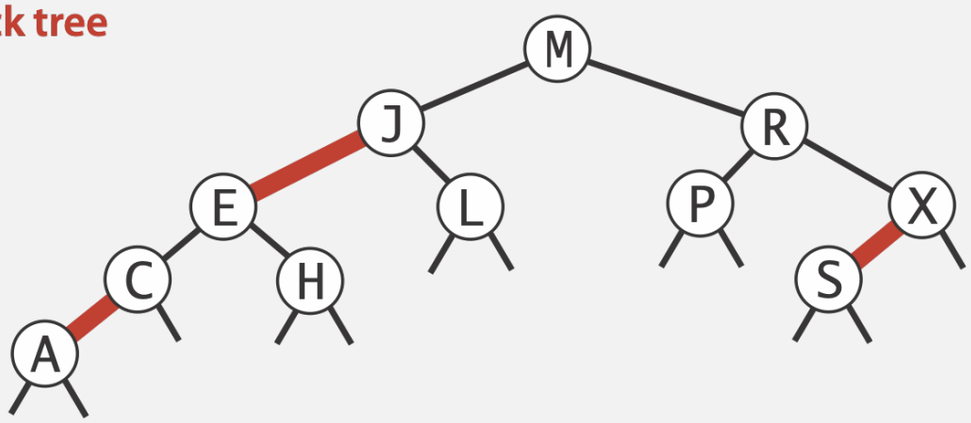
2–3 tree

corresponding red–black BST

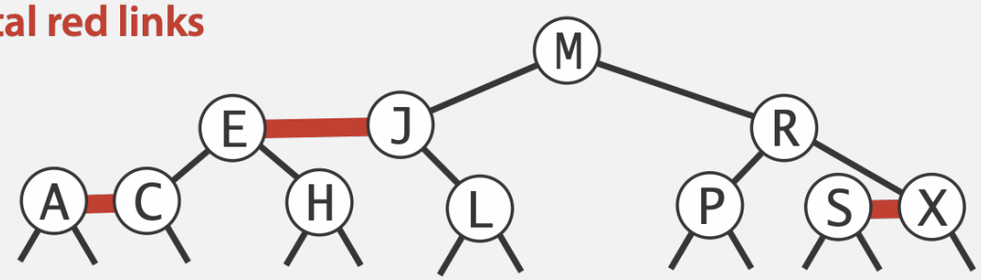
Left-leaning red–black BSTs: 1–1 correspondence with 2–3 trees

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

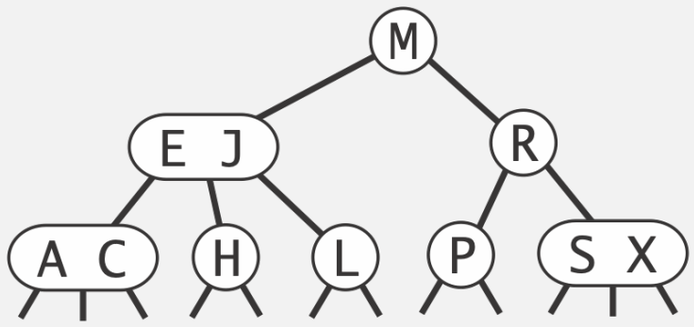
red–black tree



horizontal red links



2-3 tree



If red links are drawn horizontally, all the null links are the same distance from the root.

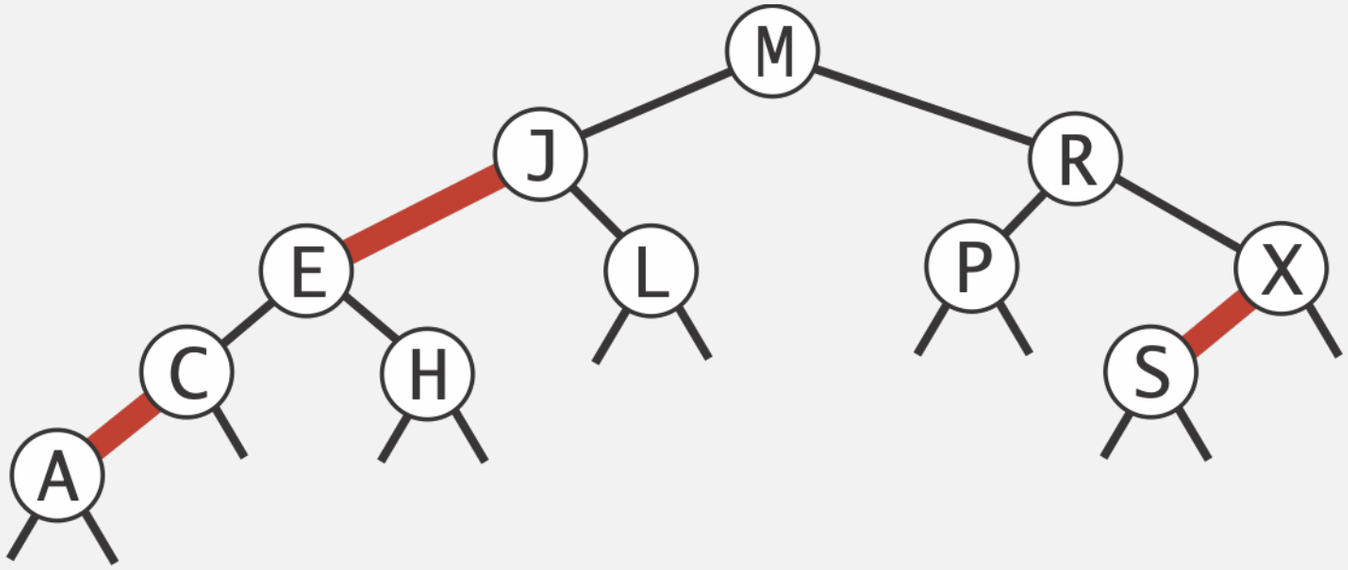
- if we then collapse together the nodes connected by red links, the result is a 2-3 tree.
- Conversely, if we draw 3-nodes in a 2-3 tree as 2-nodes connected by a red link that leans left, then no nodes has two red links connected to it, and the tree has perfect black balance.

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.

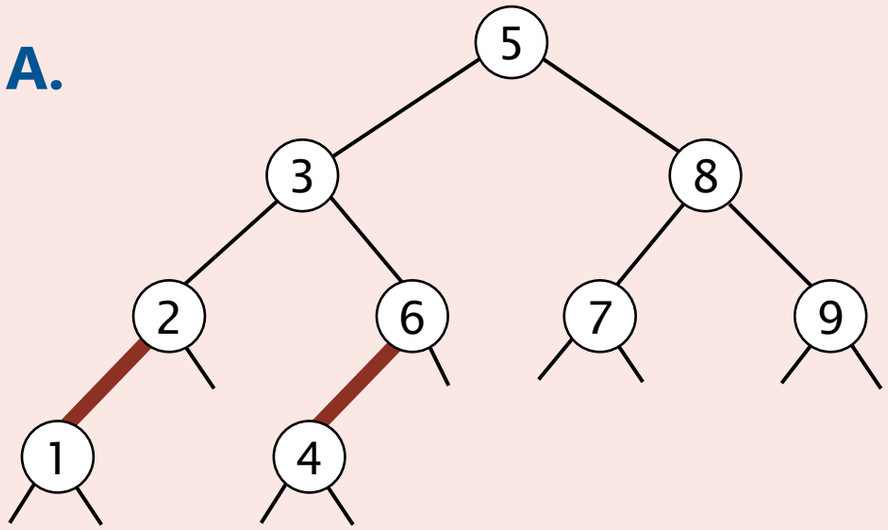
"perfect black balance"



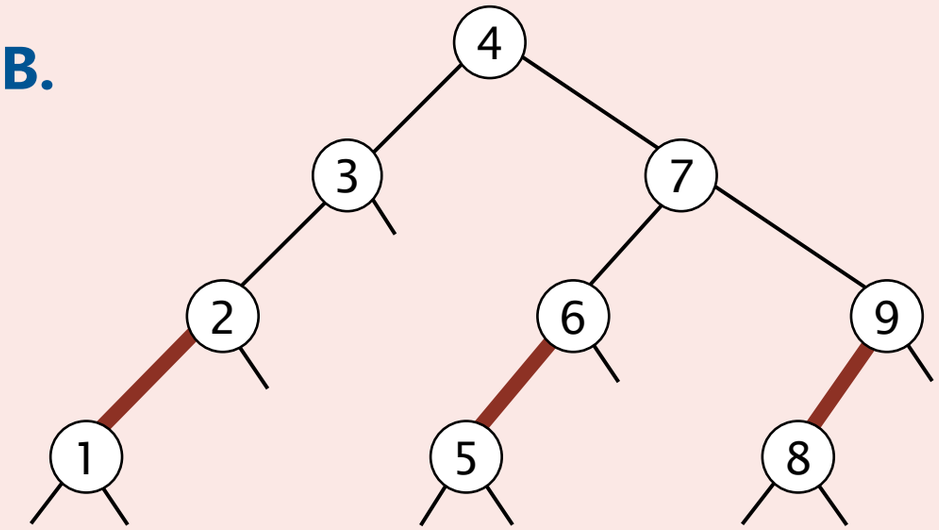
Balanced search trees: quiz 2

Which one of the following is a red–black BST?

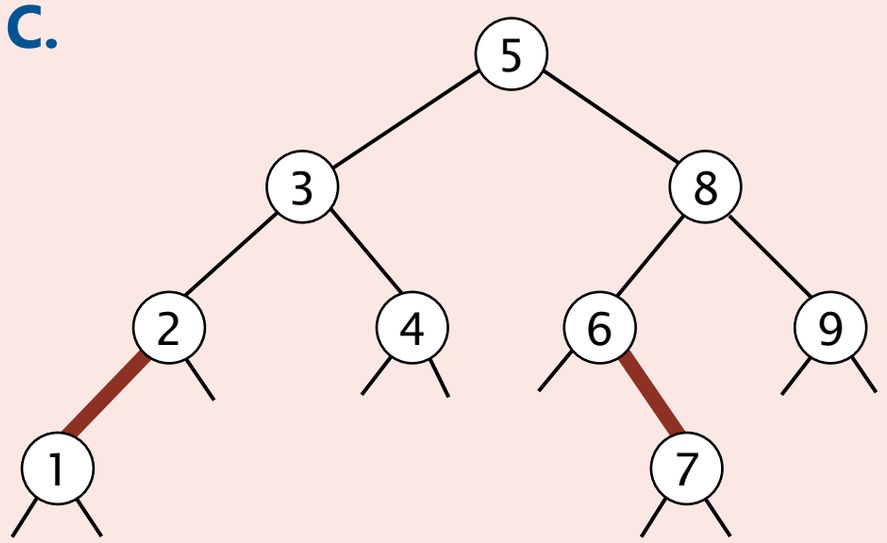
A.



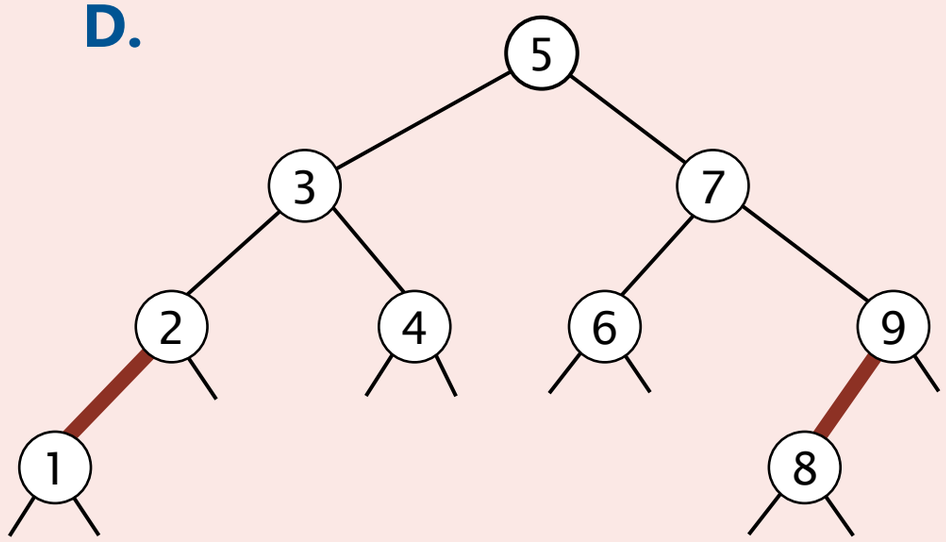
B.



C.



D.

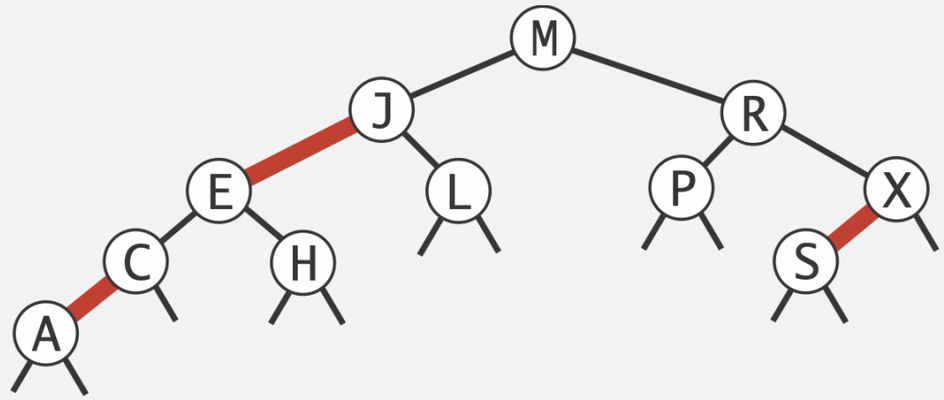


Search implementation for red–black BSTs

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

Red–black BST representation

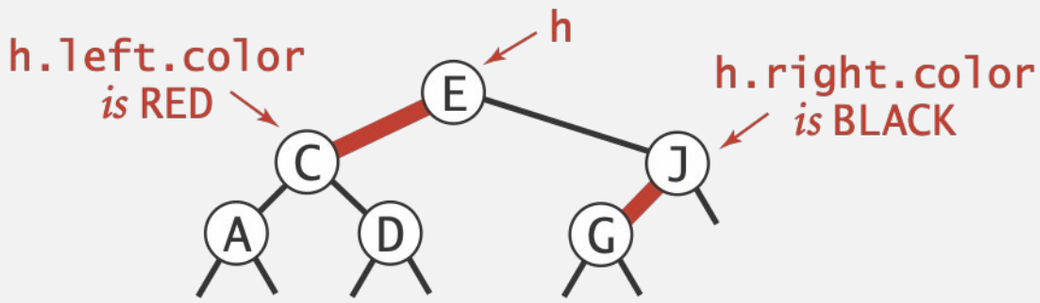
Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

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

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

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

null links are black



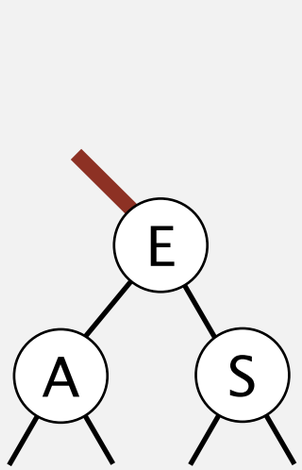
Insertion into a LLRB tree: overview

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

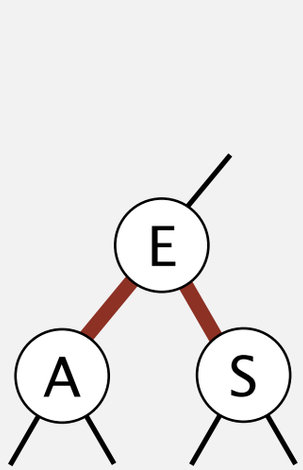
During internal operations, maintain: (could violate the color invariant)

- Symmetric order.
- Perfect black balance.

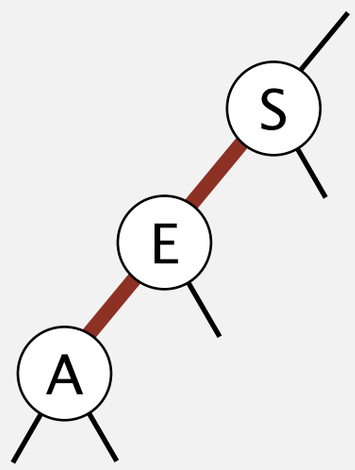
  [but not necessarily color invariants]



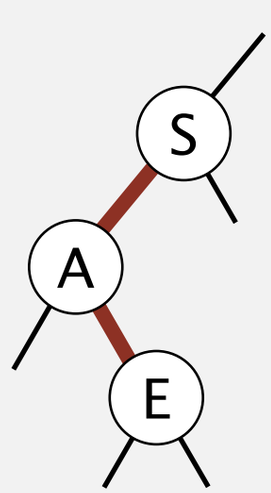
right-leaning red link



two red children (a temporary 4-node)



left-left red (a temporary 4-node)

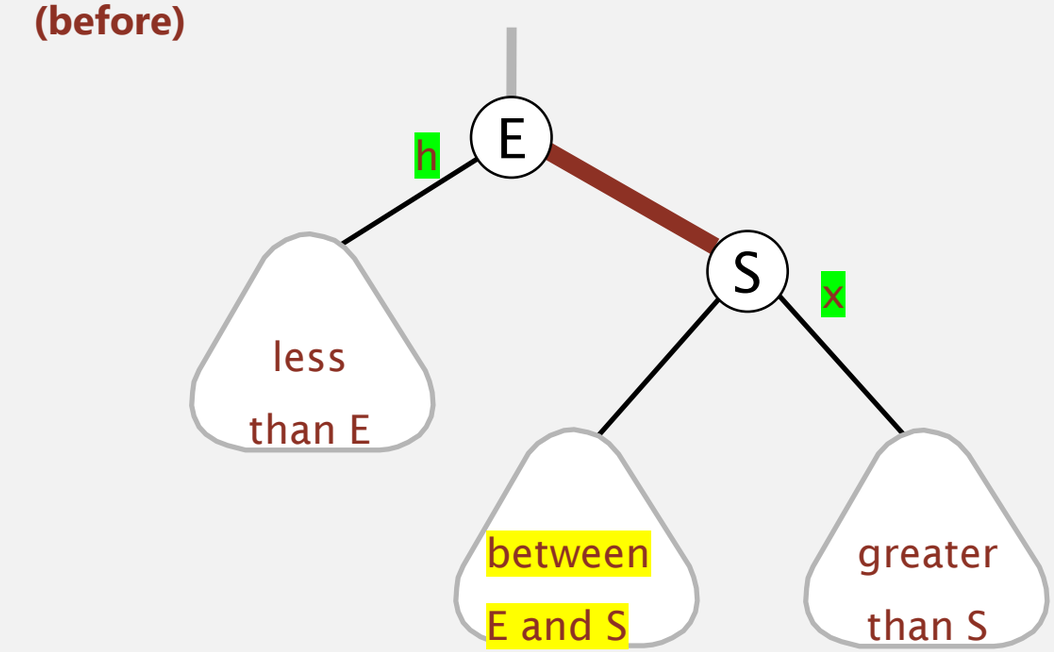


left-right red (a temporary 4-node)

To restore color invariant: apply elementary ops (rotations and color flips).

Elementary red–black BST operations

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



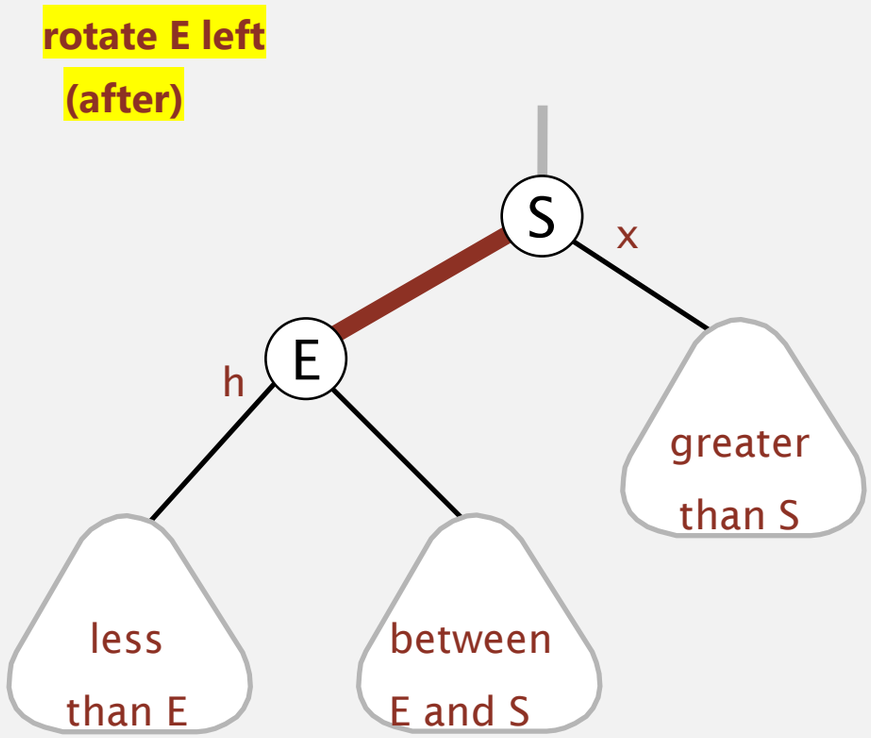
We are switching from having the smaller of the two keys at the root to having the larger of the two keys at the root.

```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right; //new parent node S
   h.right = x.left; //make E.right to S.left
   x.left = h; //make E left of S
   x.color = h.color;   <-- Preserve link 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.



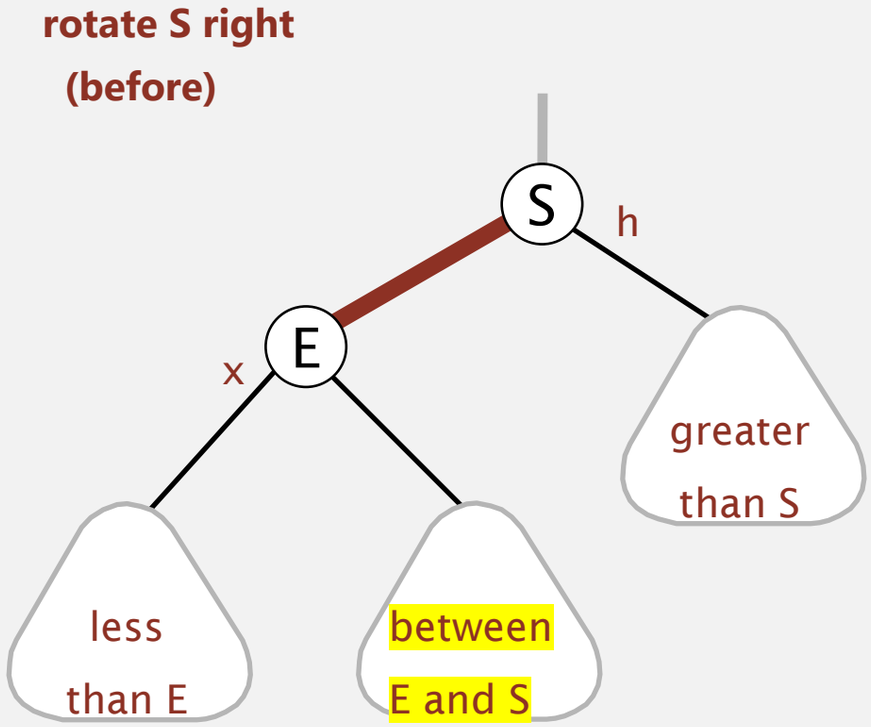

```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Returns new root to update parent.

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.

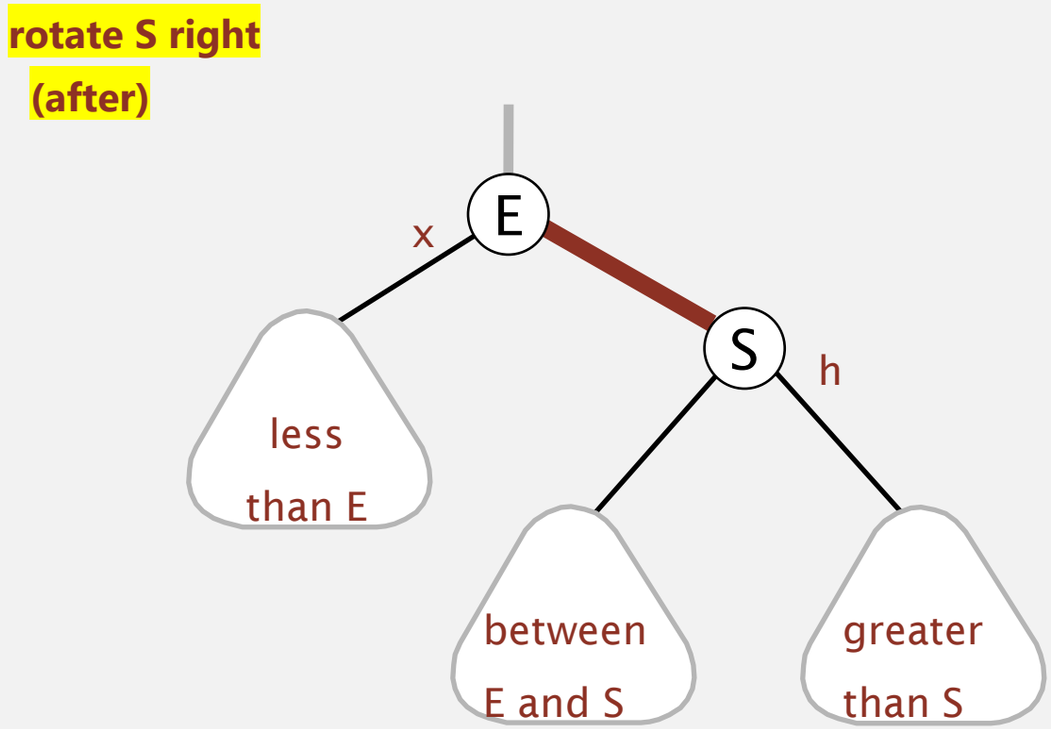



```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;  //new parent node E
   h.left = x.right;  //make S.left to E.right
   x.right = h;  //make E.right to S
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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.

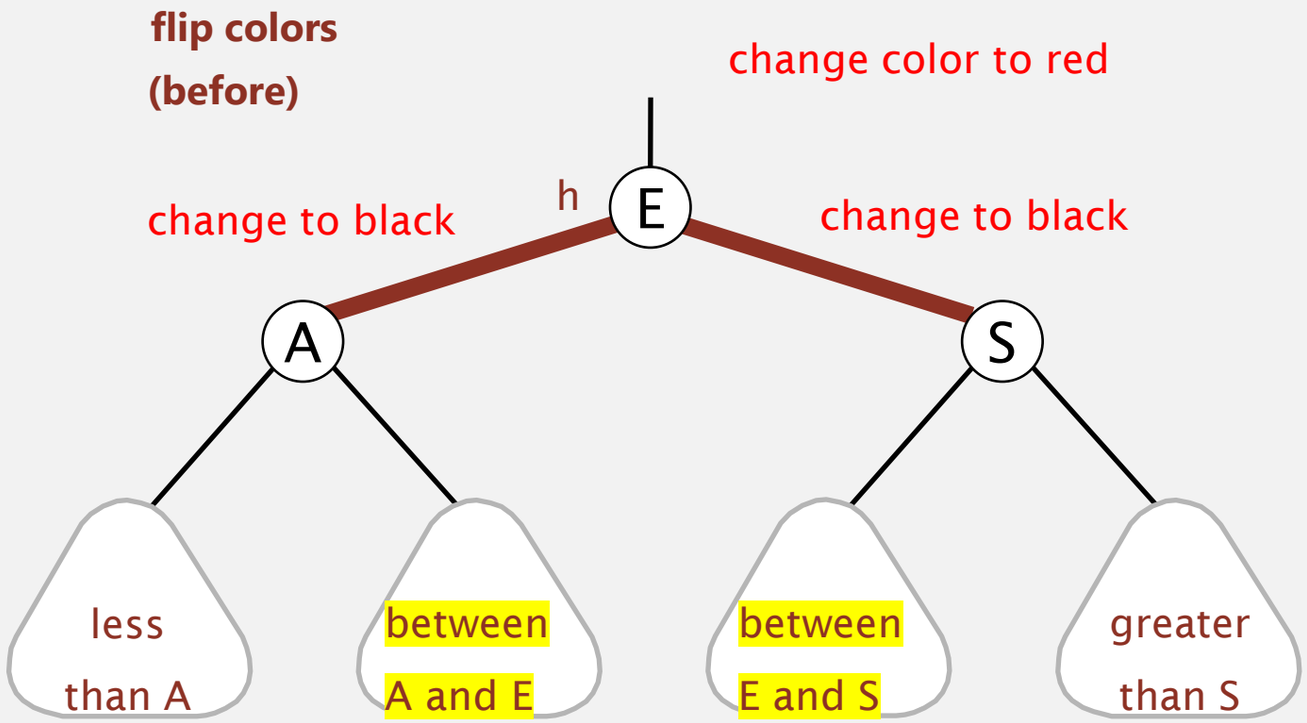



```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red–black BST operations

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



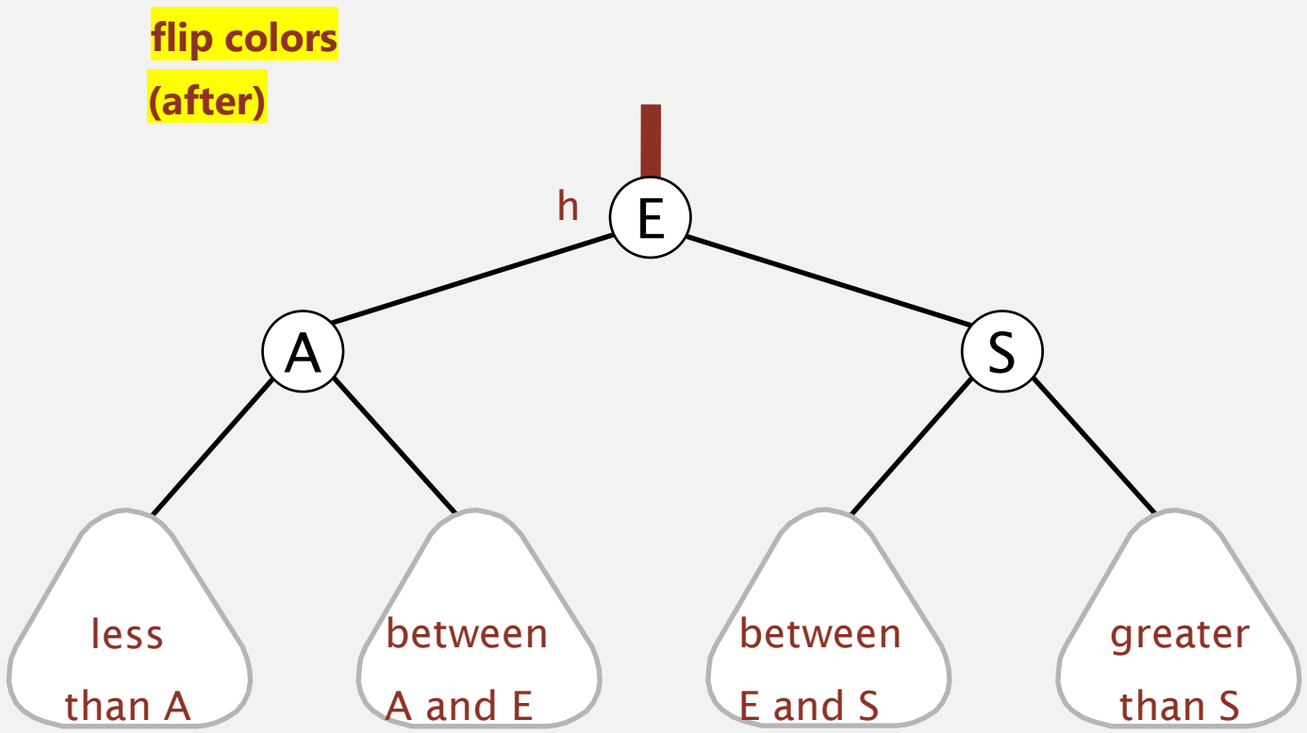
This situation arises when a node is inserted into a 3-node

```
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.



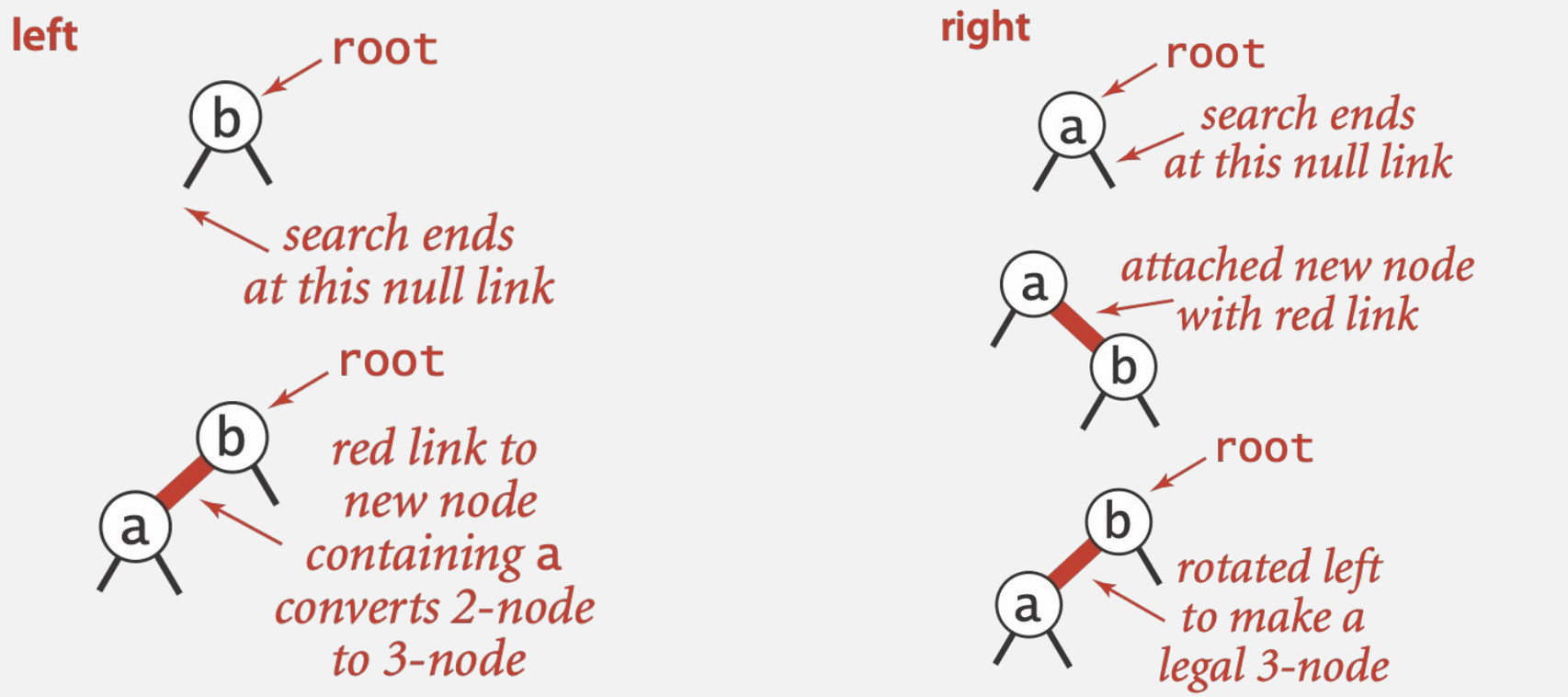
Flipping the colors the middle key is passed to its parent.

```
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.

Insertion into a LLRB tree

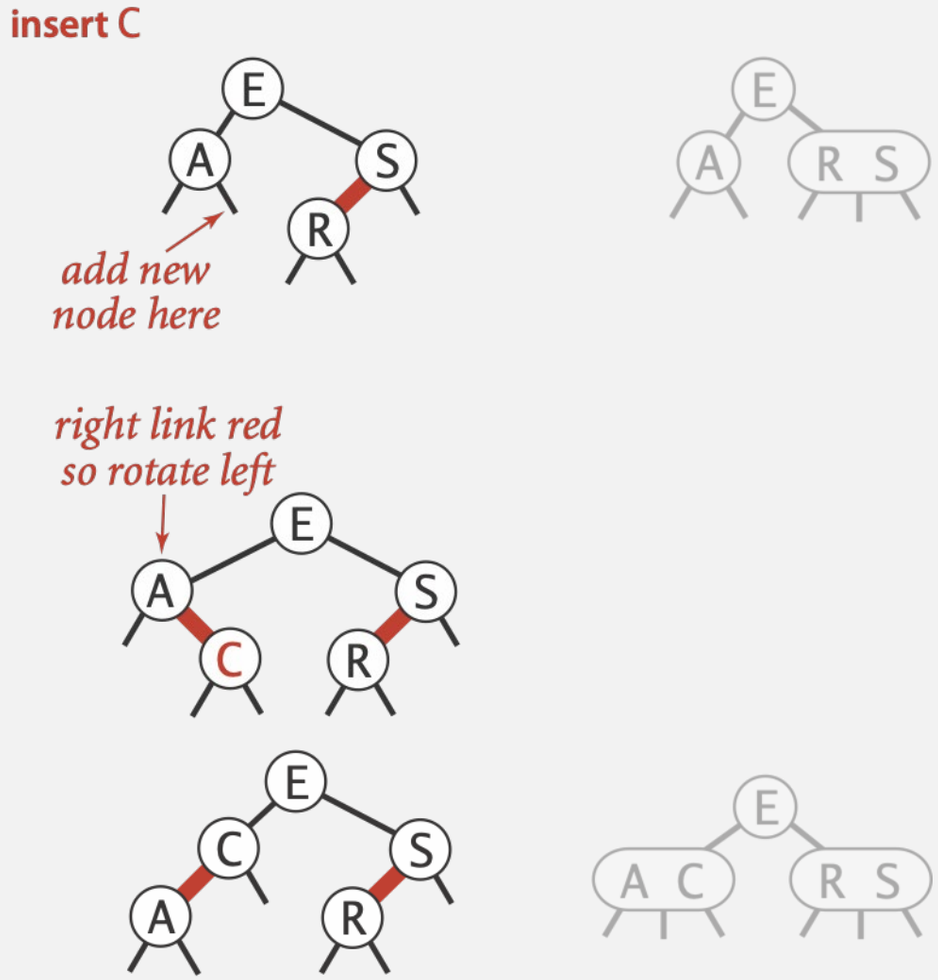
Warmup 1. Insert into a tree with exactly 1 node.



Insertion into a LLRB tree

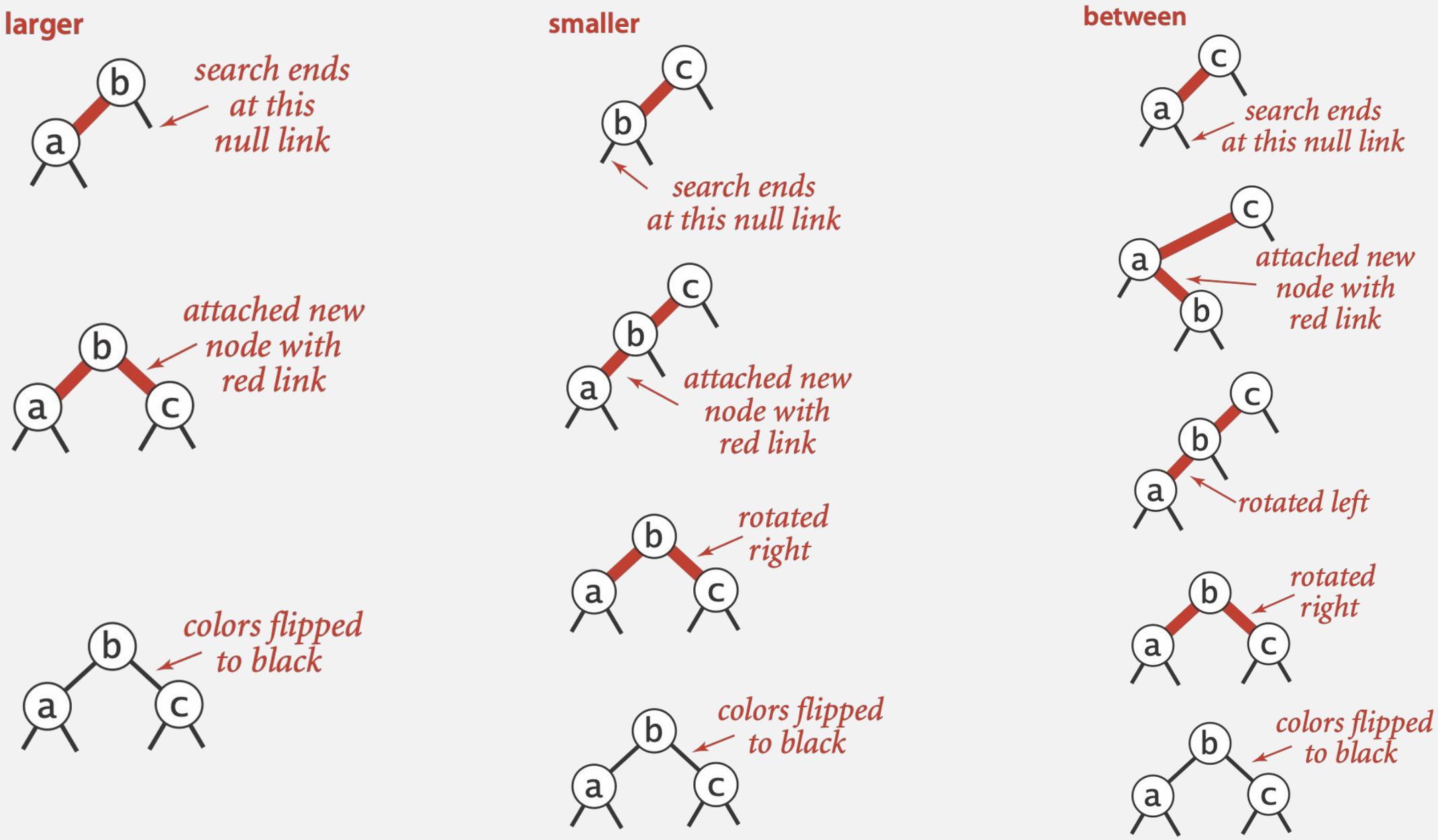
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- If new red link is a right link, rotate left. ← to restore color invariants



Insertion into a LLRB tree

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

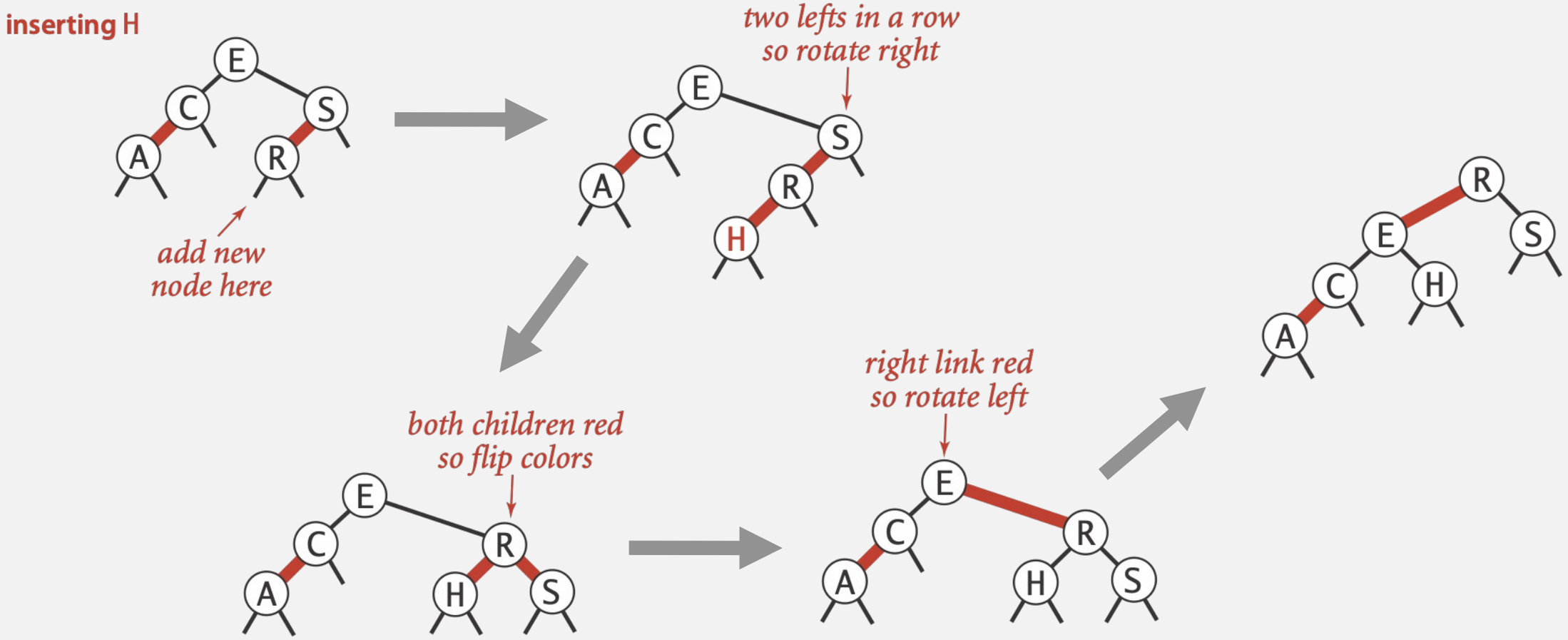


Achieve the desired result by doing zero, one, or two rotations followed by flipping of the colors of the two children of the root.

Insertion into a LLRB tree

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

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed). ← to restore color invariants

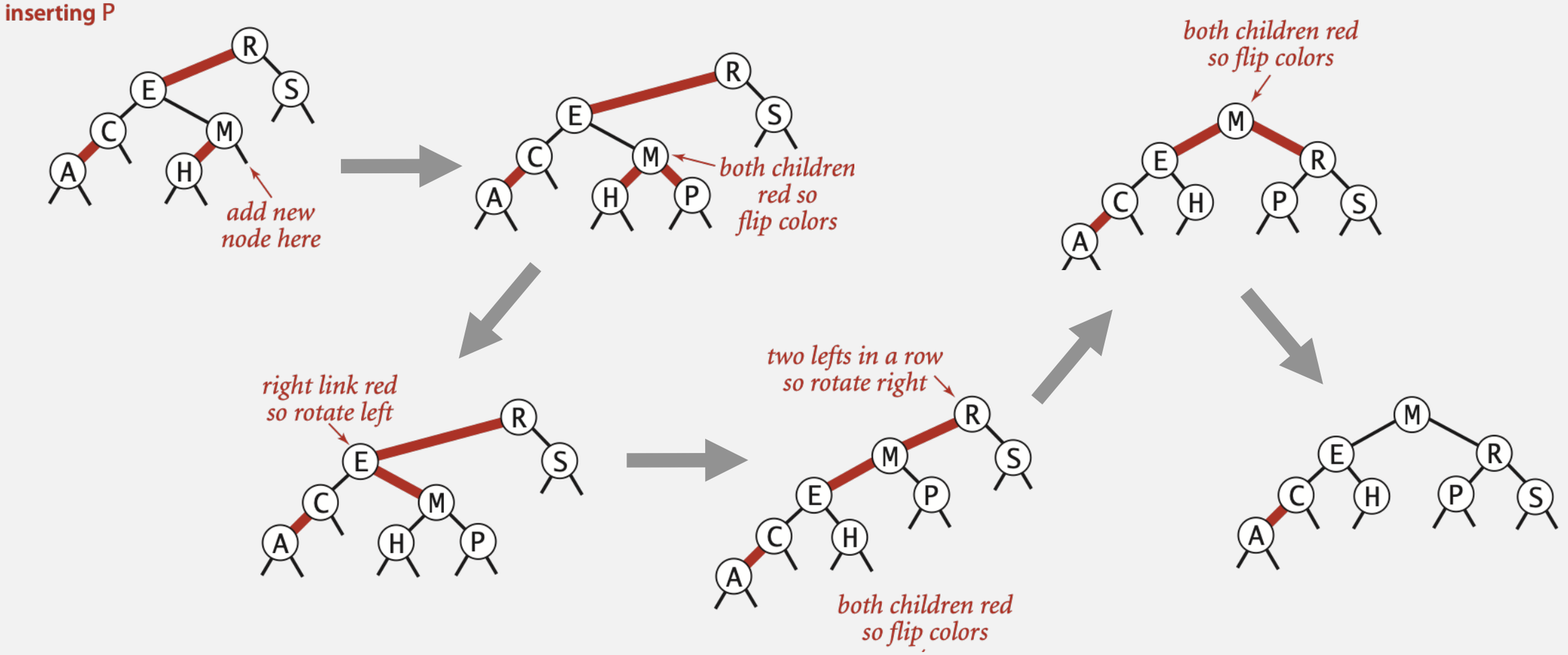


Insertion into a LLRB tree: passing red links up the tree

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

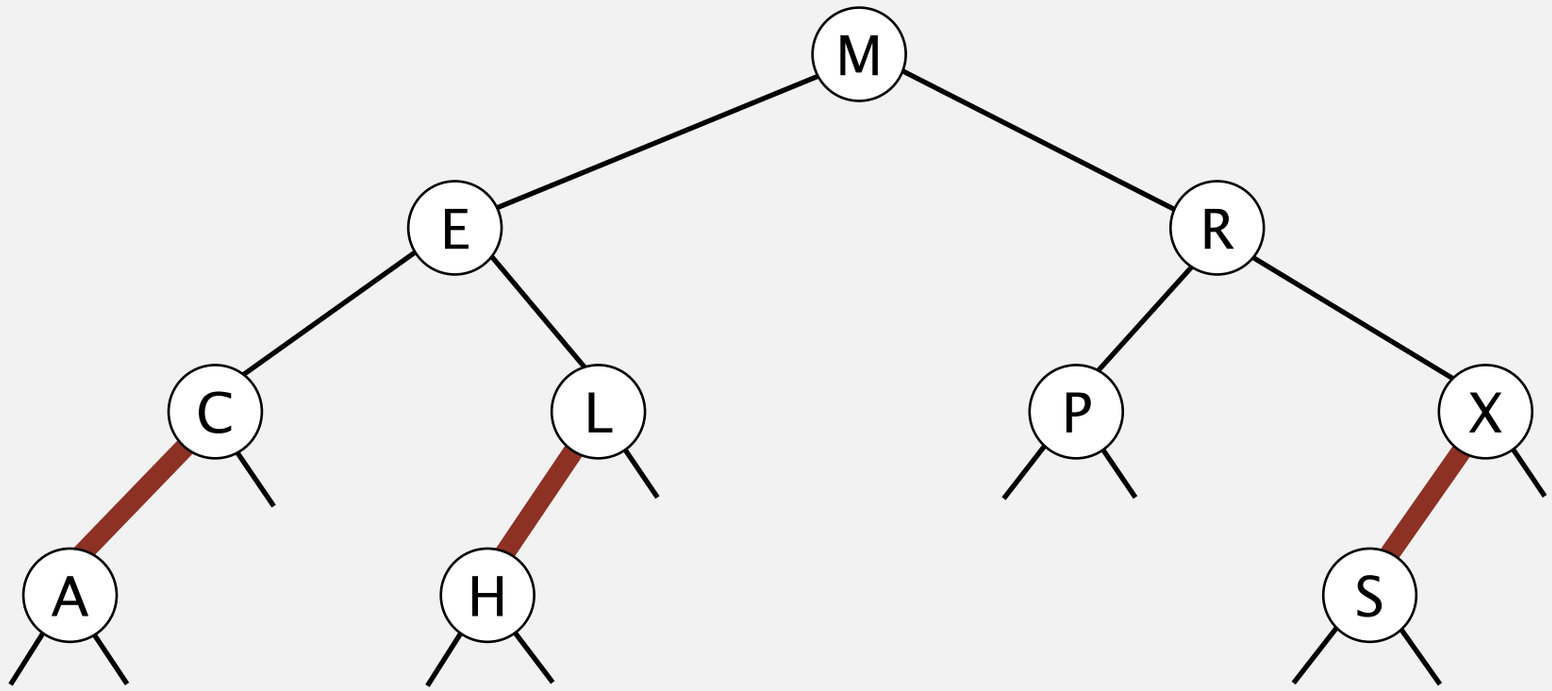
- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat case 1 or case 2 up the tree (if needed).

← to restore color invariants

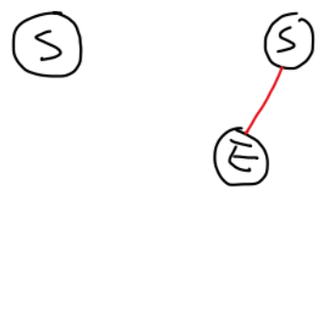


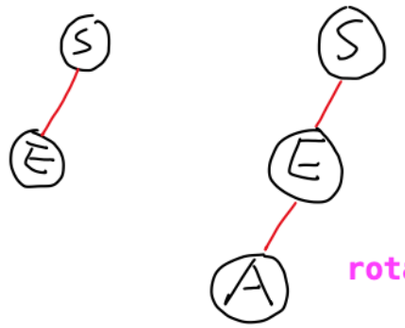
Red–black BST construction demo

insert S E A R C H X M P L

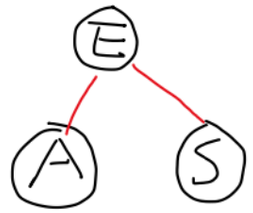




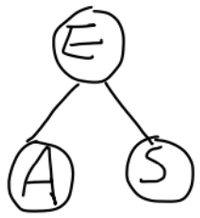


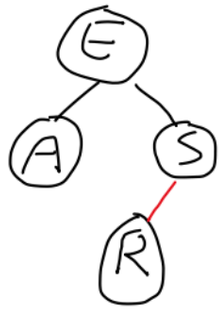


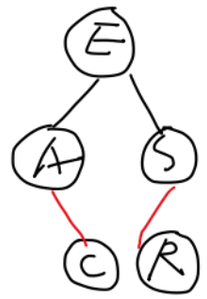
rotate R



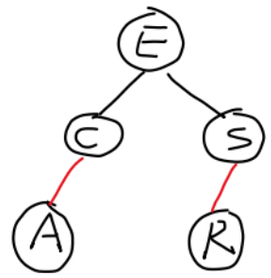
flip colors

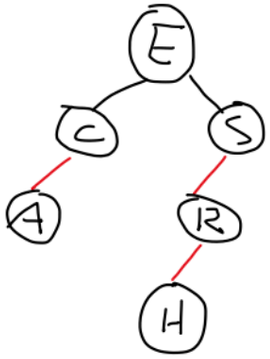




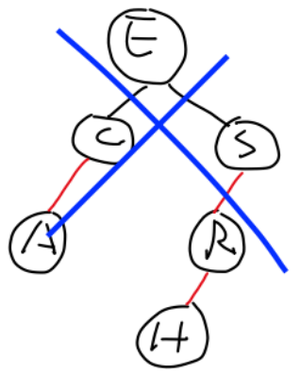


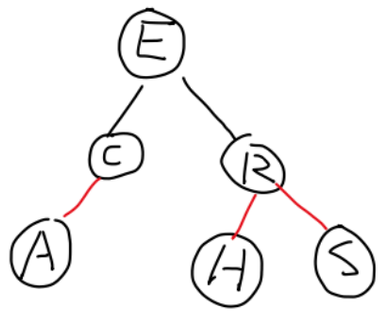
rotate L



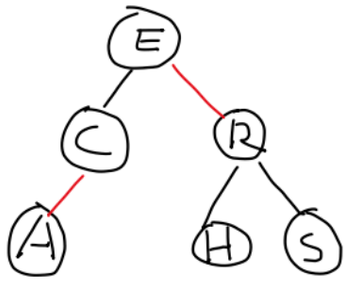


rotate R

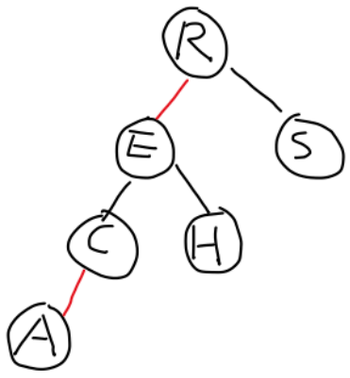


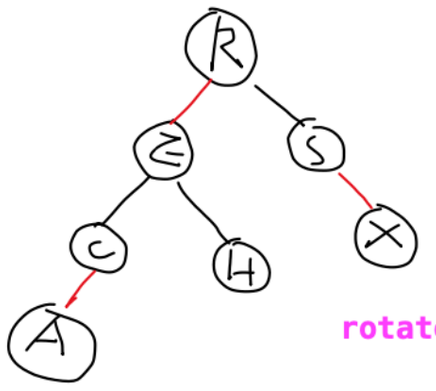


flip colors

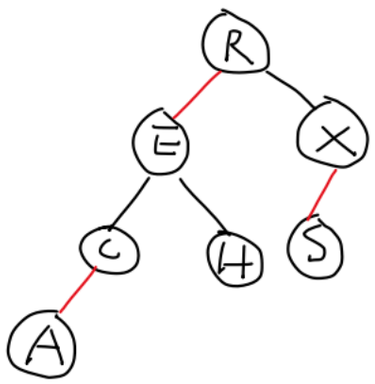


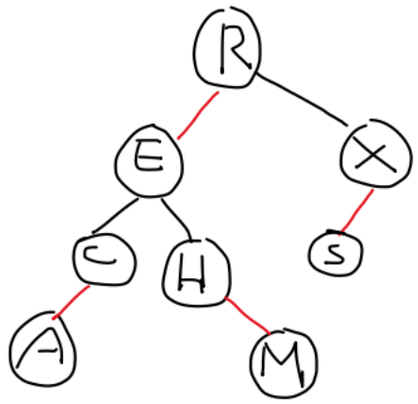
rotate L

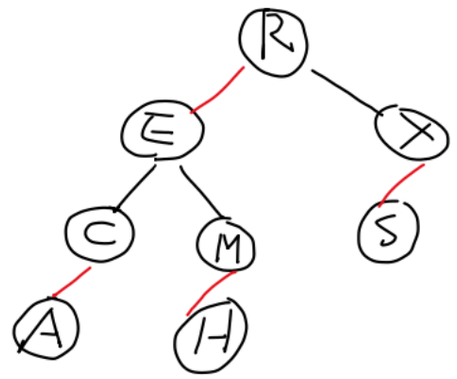




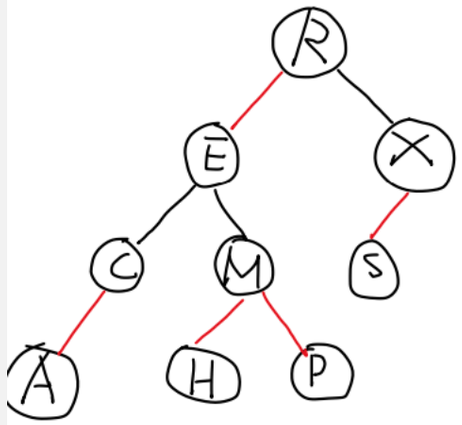
rotate L



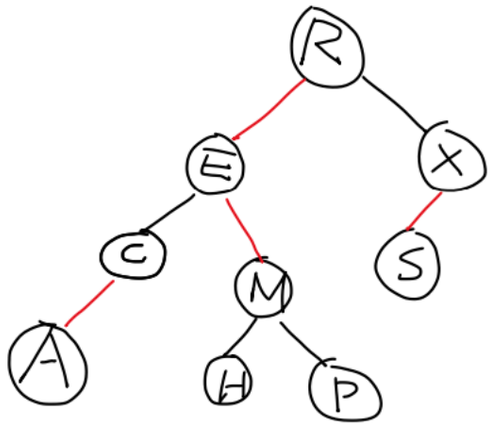




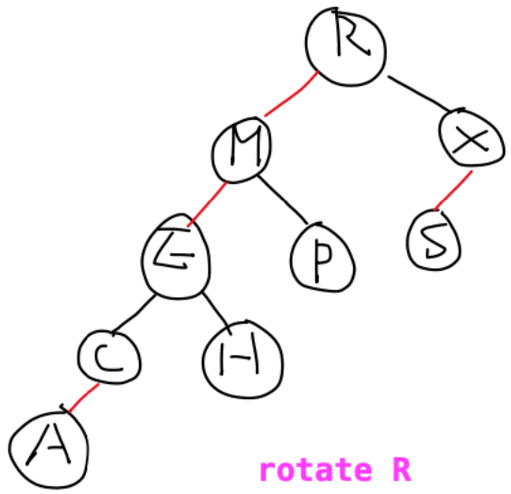
rotate L



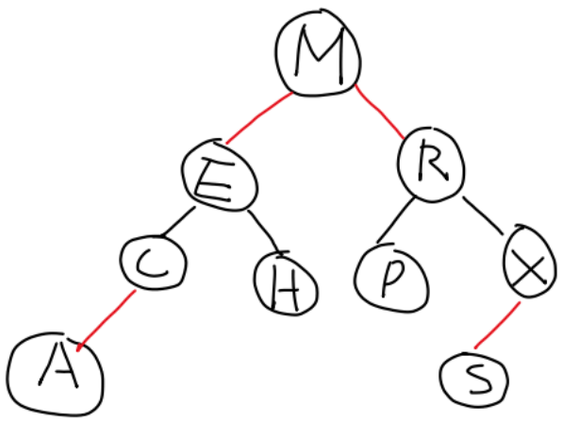
flip colors



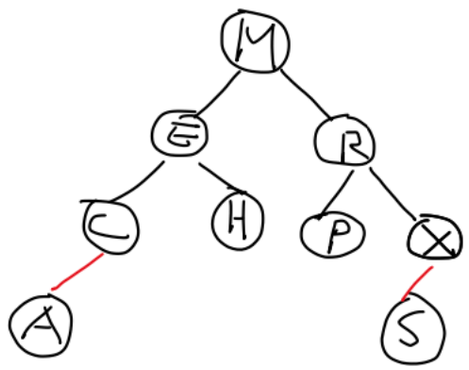
rotate L

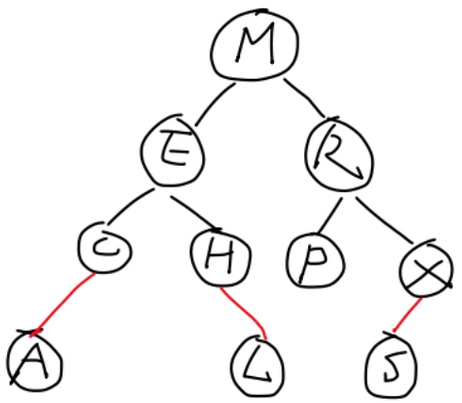


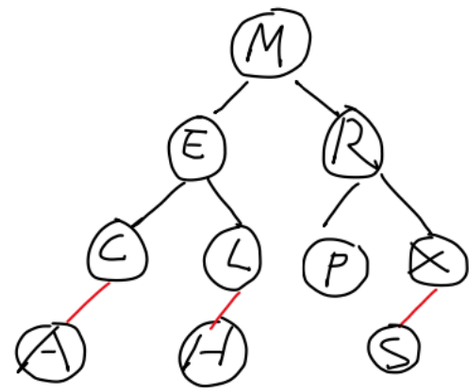
rotate R



flip colors





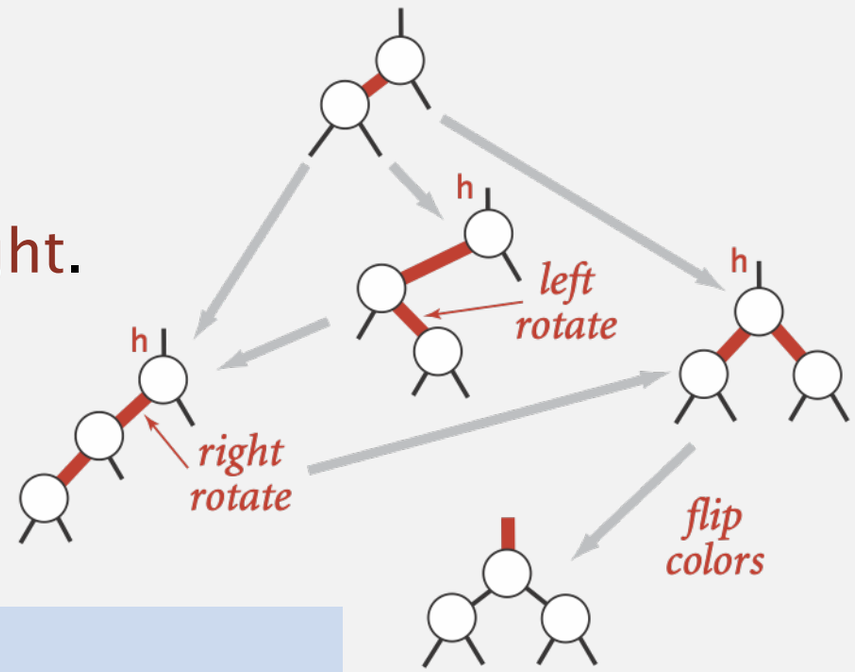


rotate L

Insertion into a LLRB tree: Java implementation

Can distill down to three cases!

- Right child red; left child black: rotate left.
- Left child red; 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)

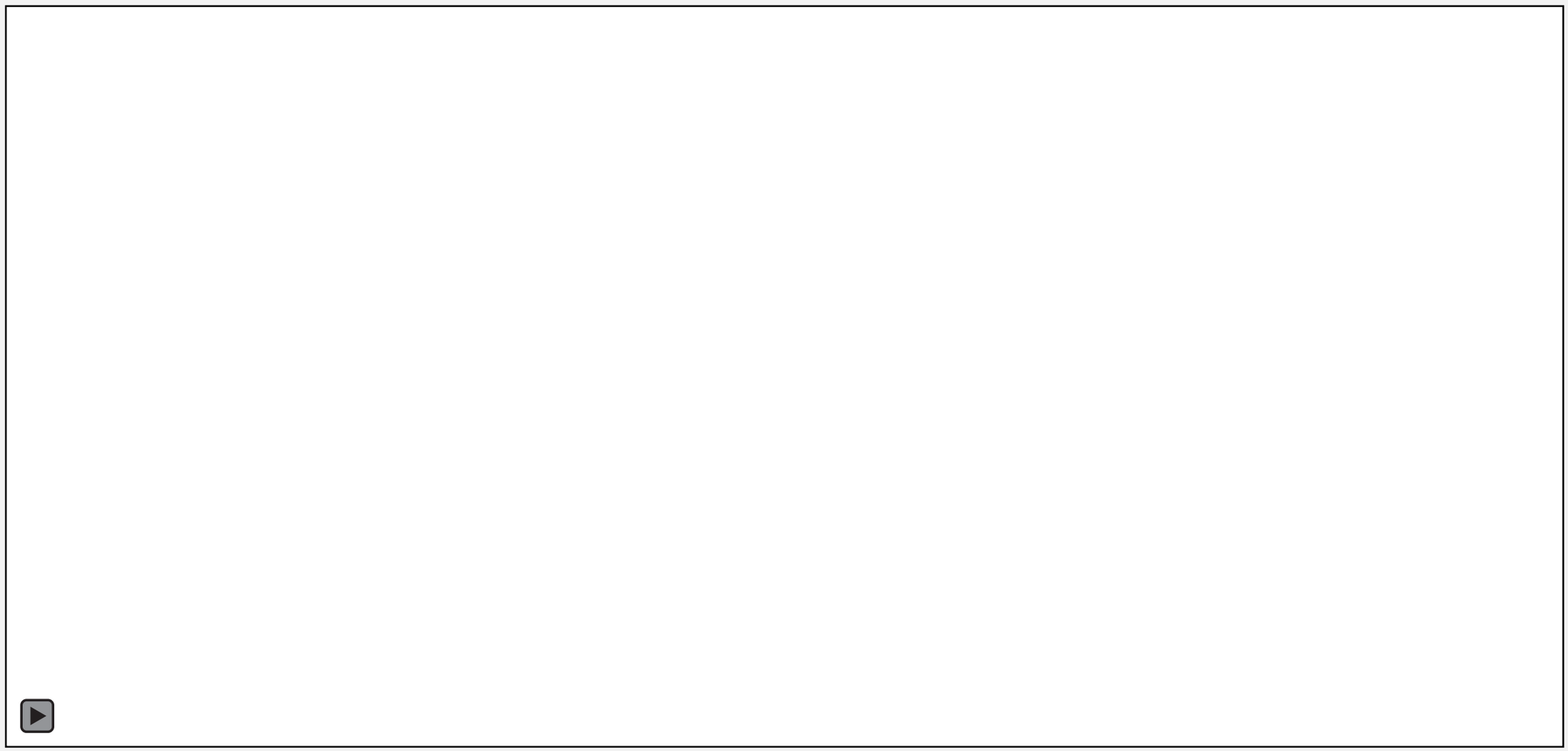
lean left

balance 4-node

split 4-node

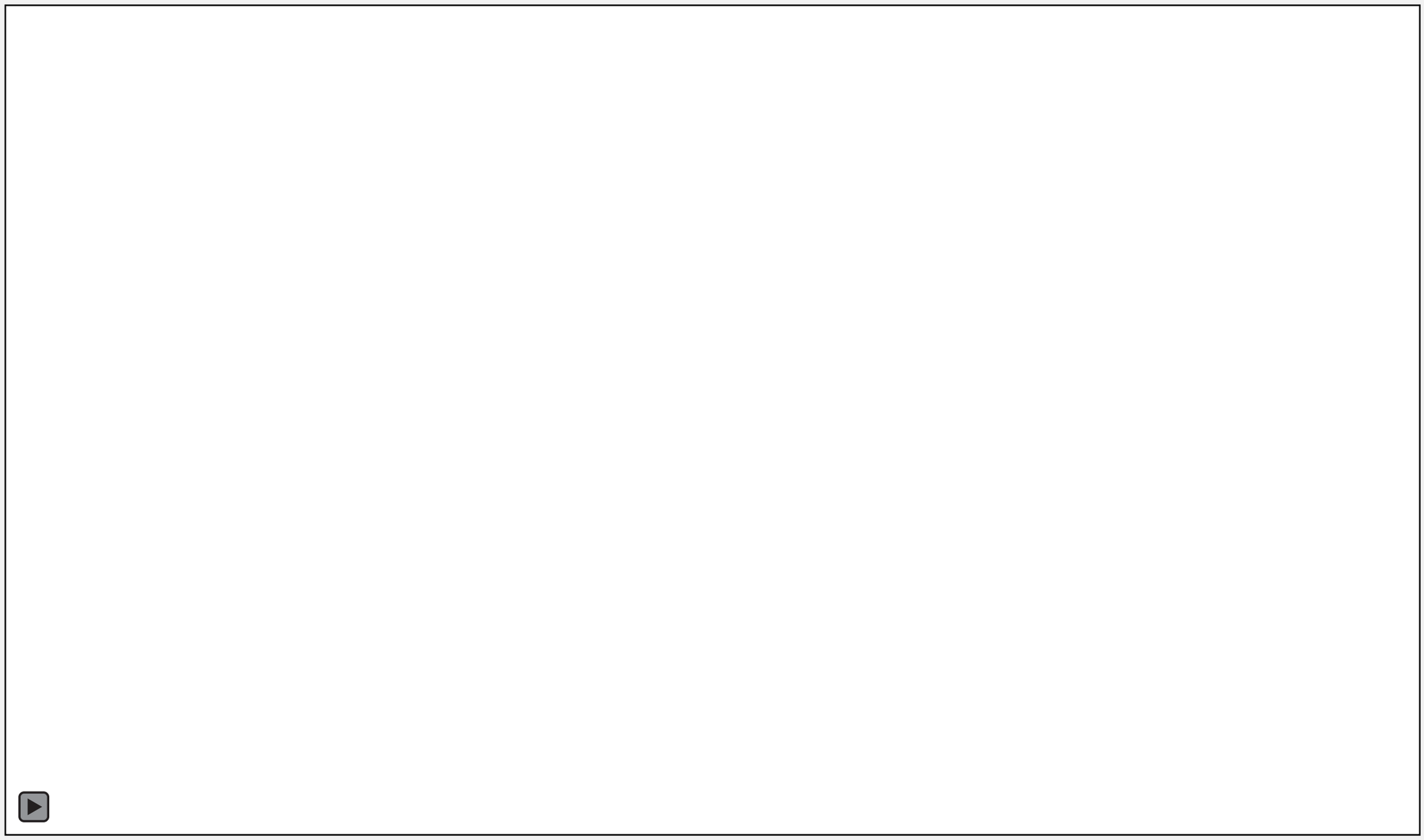
only a few extra lines of code provides near-perfect balance

Insertion into a LLRB tree: visualization



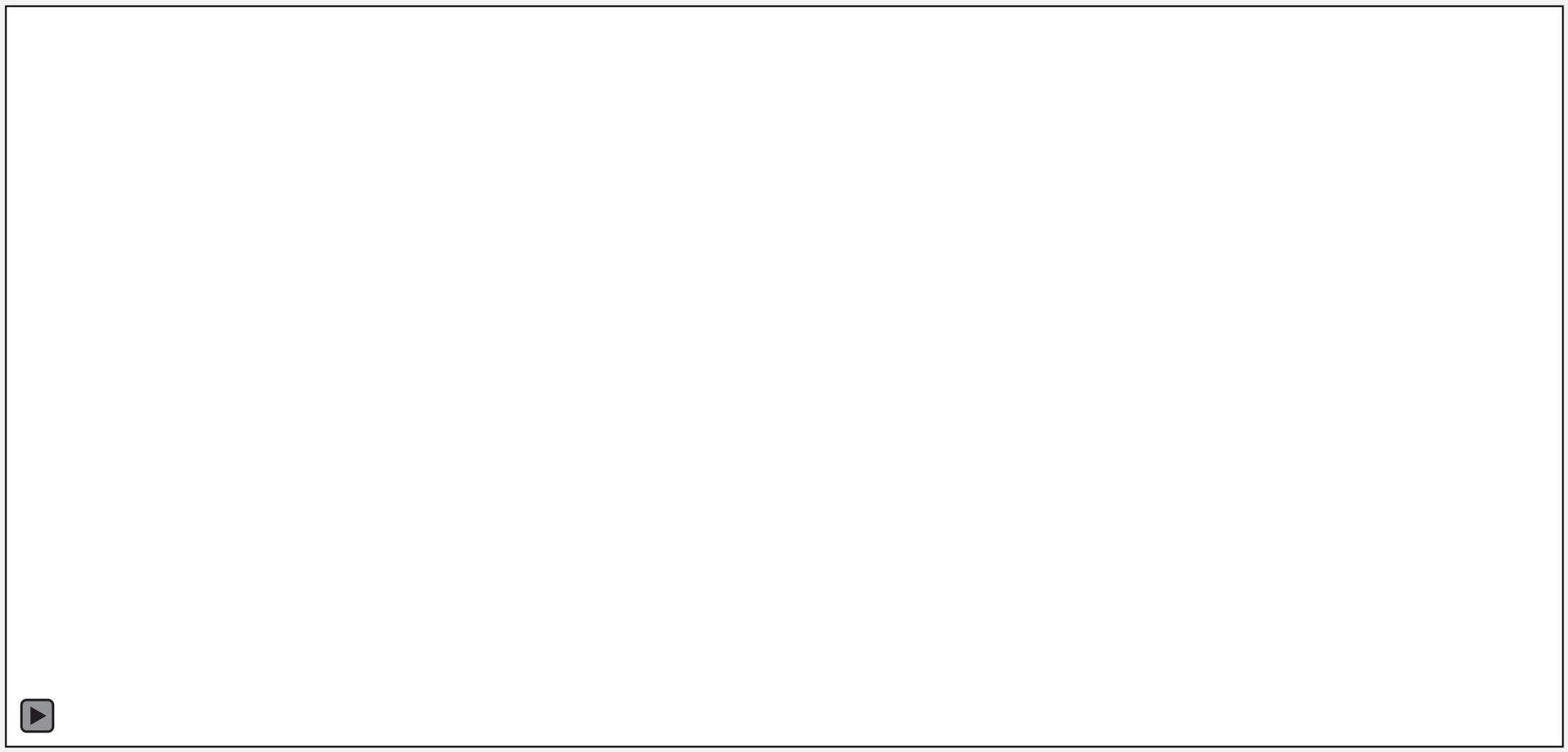
255 insertions in ascending order

Insertion into a LLRB tree: visualization



255 insertions in descending order

Insertion into a LLRB tree: visualization



255 random insertions

Balanced search trees: quiz 4

What is the maximum height of a LLRB tree with n keys?

A. $\sim \log_3 n$

B. $\sim \log_2 n$

C. $\sim 2 \log_2 n$

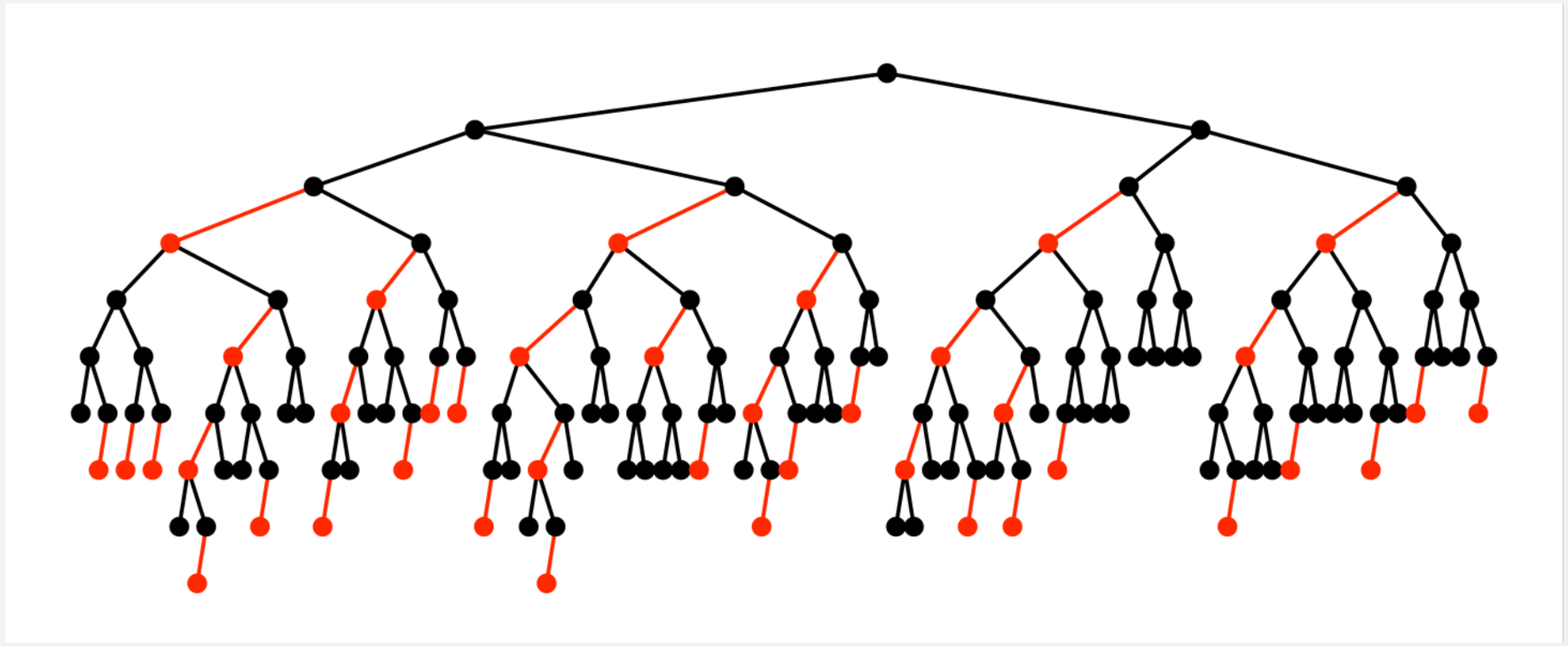
D. $\sim n$

Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \log_2 n$ in the worst case.

Pf.

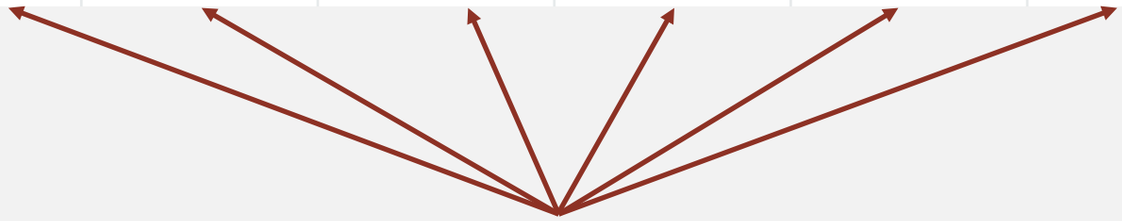
- Black height = height of corresponding 2-3 tree $\leq \log_2 n$.
- Never two red links in-a-row.



Empirical observation. Height of tree is $\sim 1.0 \log_2 n$ 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) red links and black links

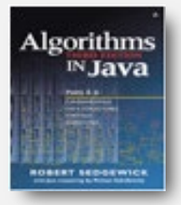
Why left-leaning trees?

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)



Why left-leaning red–black BSTs?

Simplified code.

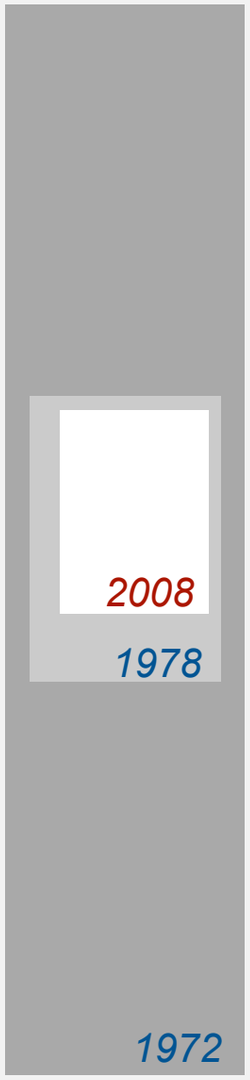
- Left-leaning restriction reduces number of cases.
- Short inner loop.

Same ideas simplify implementation of other operations.

- Delete min/max.
- Arbitrary delete.

Improves widely-used balanced search trees.

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



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

War story: why red–black?

Xerox PARC innovations. [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- Laser printing.
- Bitmapped display.
- WYSIWYG text editor.
- ...

XEROX PARC



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.

should allow for $\leq 2^{40}$ keys

Extended telephone service outage.

- 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 $2 \log_2 n$. "* — *expert witness*

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.

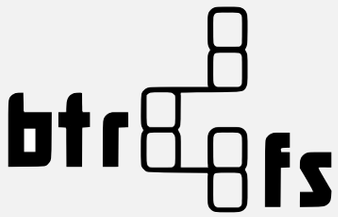
Other balanced BSTs. AVL trees, splay trees, randomized BSTs, ….

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.









BALANCED SEARCH TREES

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

B-trees

The balanced-tree algorithms we have learned so far deal with small symbol tables that can fit into addressable memory.

Large symbol tables (trillions of items) are kept on a disk or on the web.

- We are not only searching local files but web pages as well
- Files and web pages may be stored on a remote computer
- The amount of data that we might wish to search is virtually unlimited

B-trees are an extension of the balanced-tree algorithms that can support external search in symbol tables that are kept on a disk or on the web.

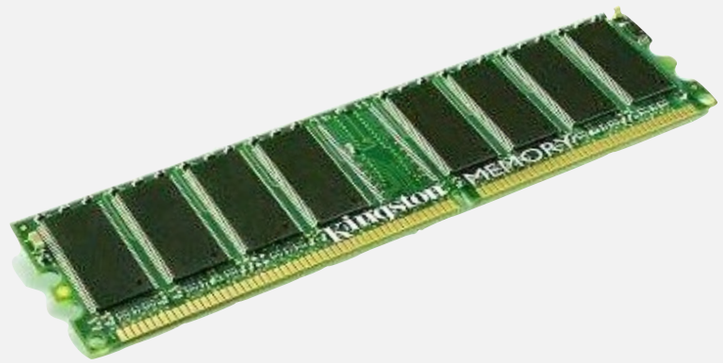
B-trees support search and insert operations on symbol tables containing trillions of items using only four or five references to small blocks of data.

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).





May be a file on your computer or a web page on a remote computer, or a file on a server.

slow fast

Property. Time required for a probe is much larger than time to access data within a page. (Typically, 4096 bytes per page)

Cost model. Number of probes.

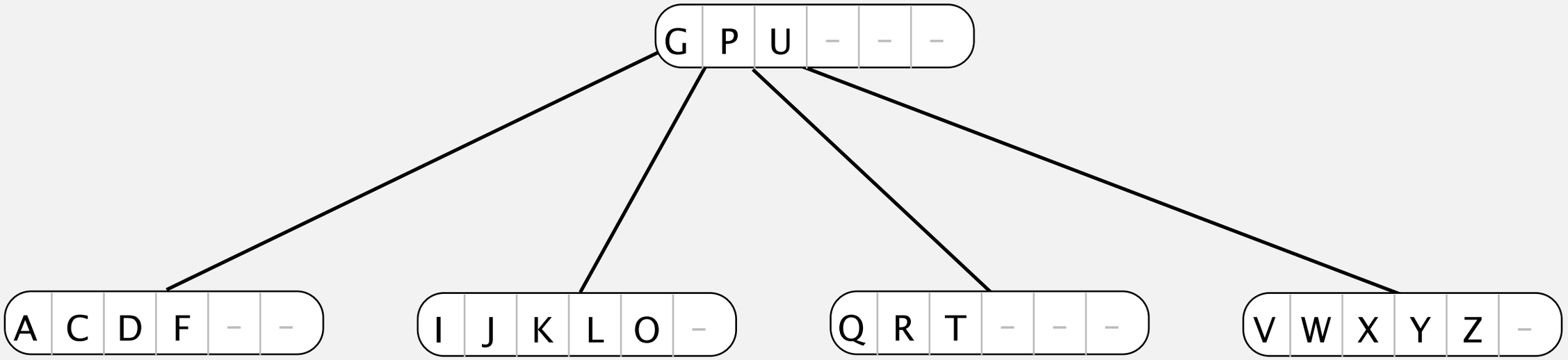
Goal. Access data using minimum number of probes.

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)

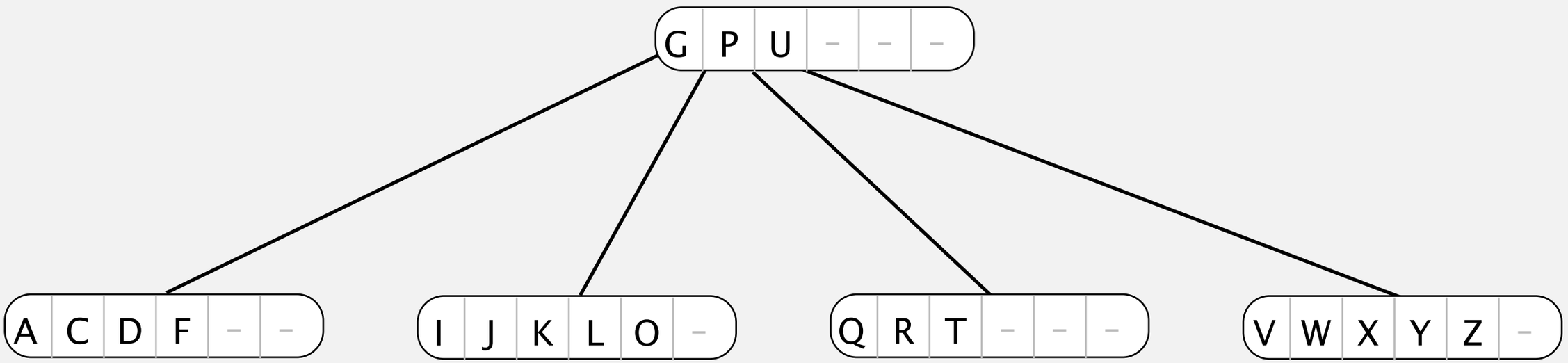


a B-tree (M = 6)

Search in a B-tree

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

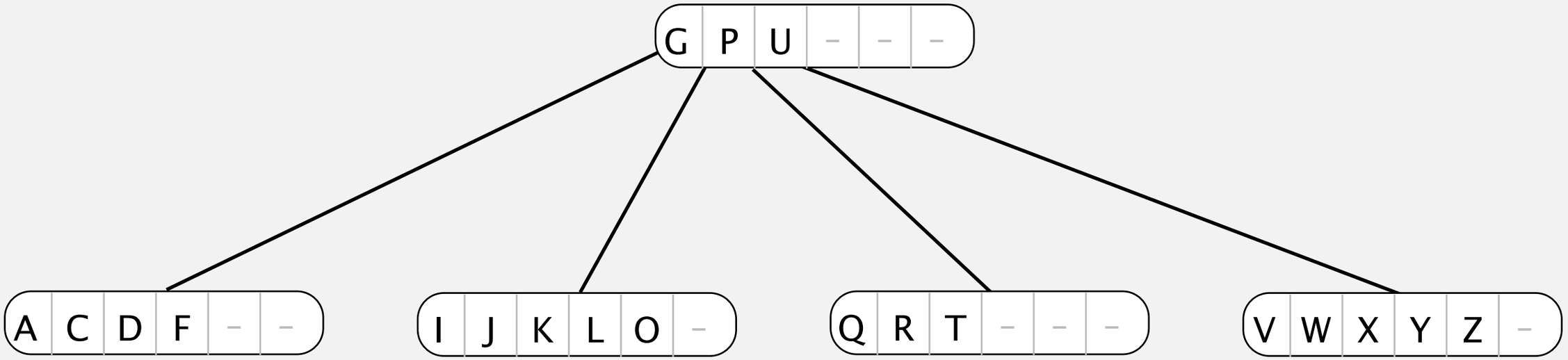
could use binary search
(but all ops are considered free)



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

Which of the following does the 'B' in B-tree not mean?

A. Bayer

B. Balanced

C. Binary

D. Boeing

> *" 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*

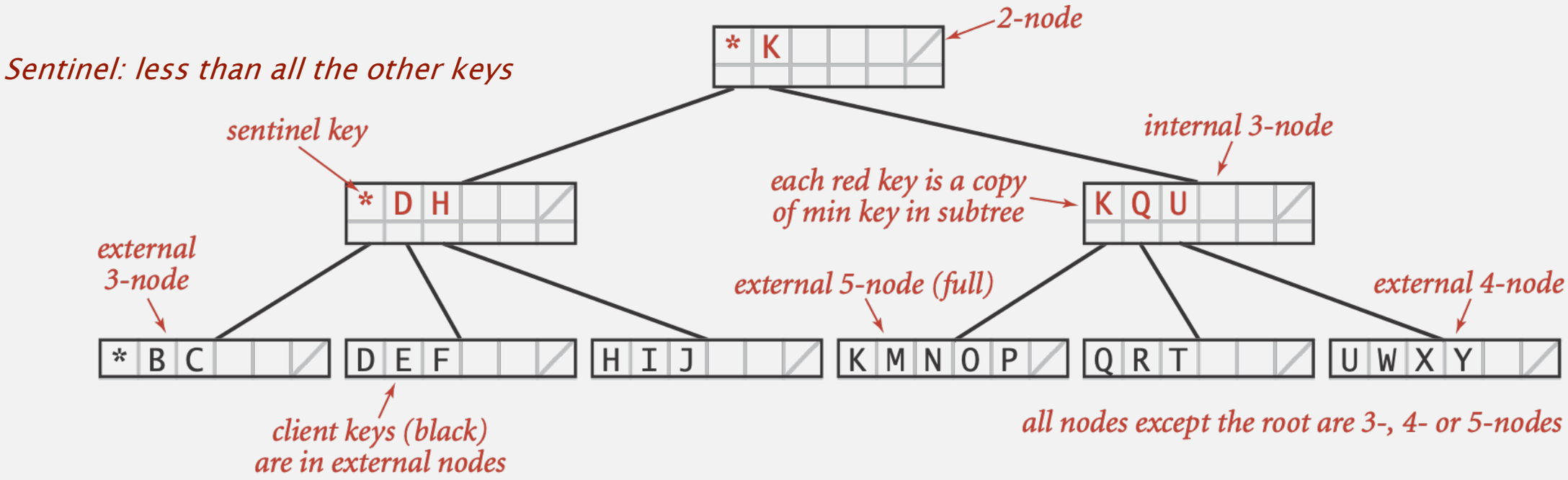


B-trees (Bayer–McCreight, 1972)

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.

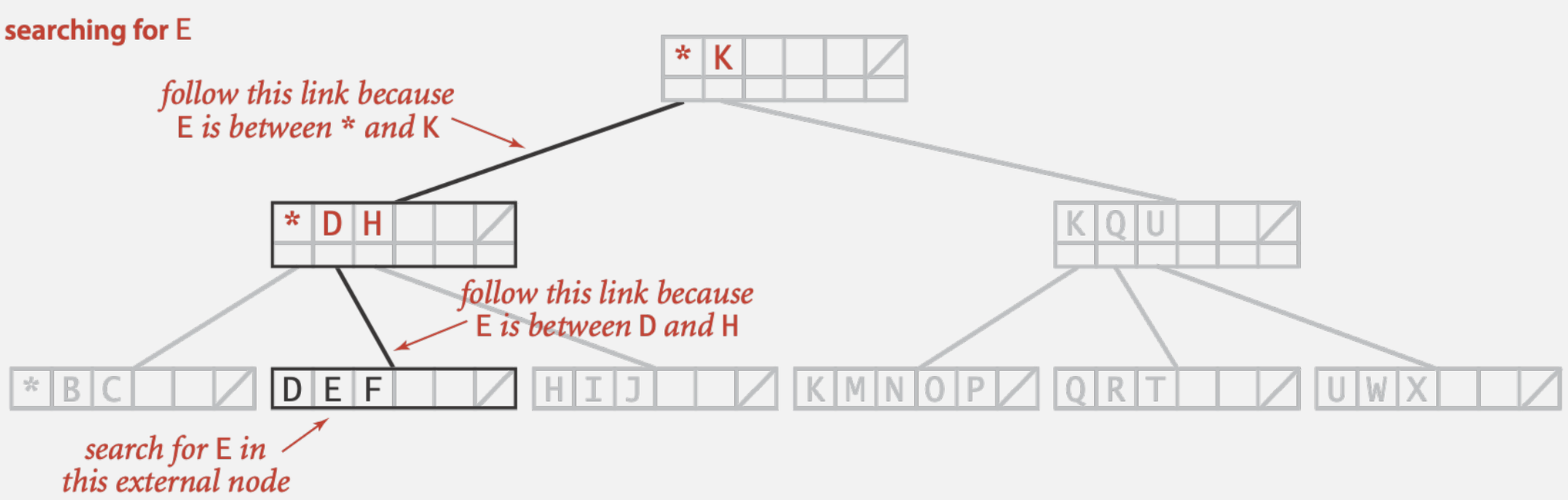
choose M as large as possible so that M links fit in a page, e.g., M = 1024



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

Searching in a B-tree

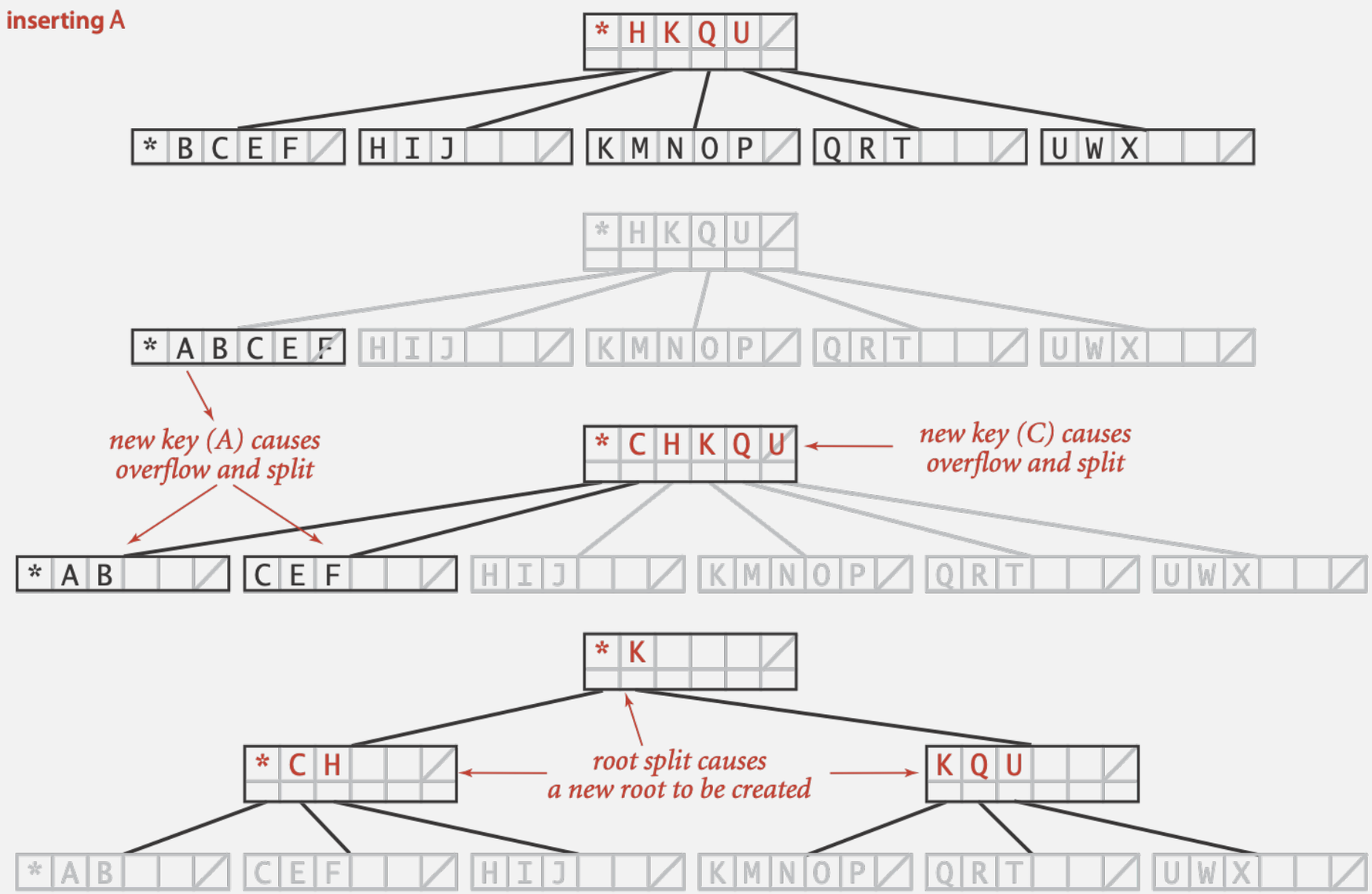
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



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

Building a large B-tree



INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

BALANCED SEARCH TREES

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



http://ds.cs.rutgers.edu

Some slides Adopted and modified from Sedgewick and Wayne

Last updated: Sep 11, 2018 1:36 PM EDT.