BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

BALANCED TREES

Balanced Search Trees

- ‣ **2-3 search trees**
- ‣ **Red-black BSTs**
- ‣ **B-trees**
- ‣ **Geometric applications of BSTs**

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()`

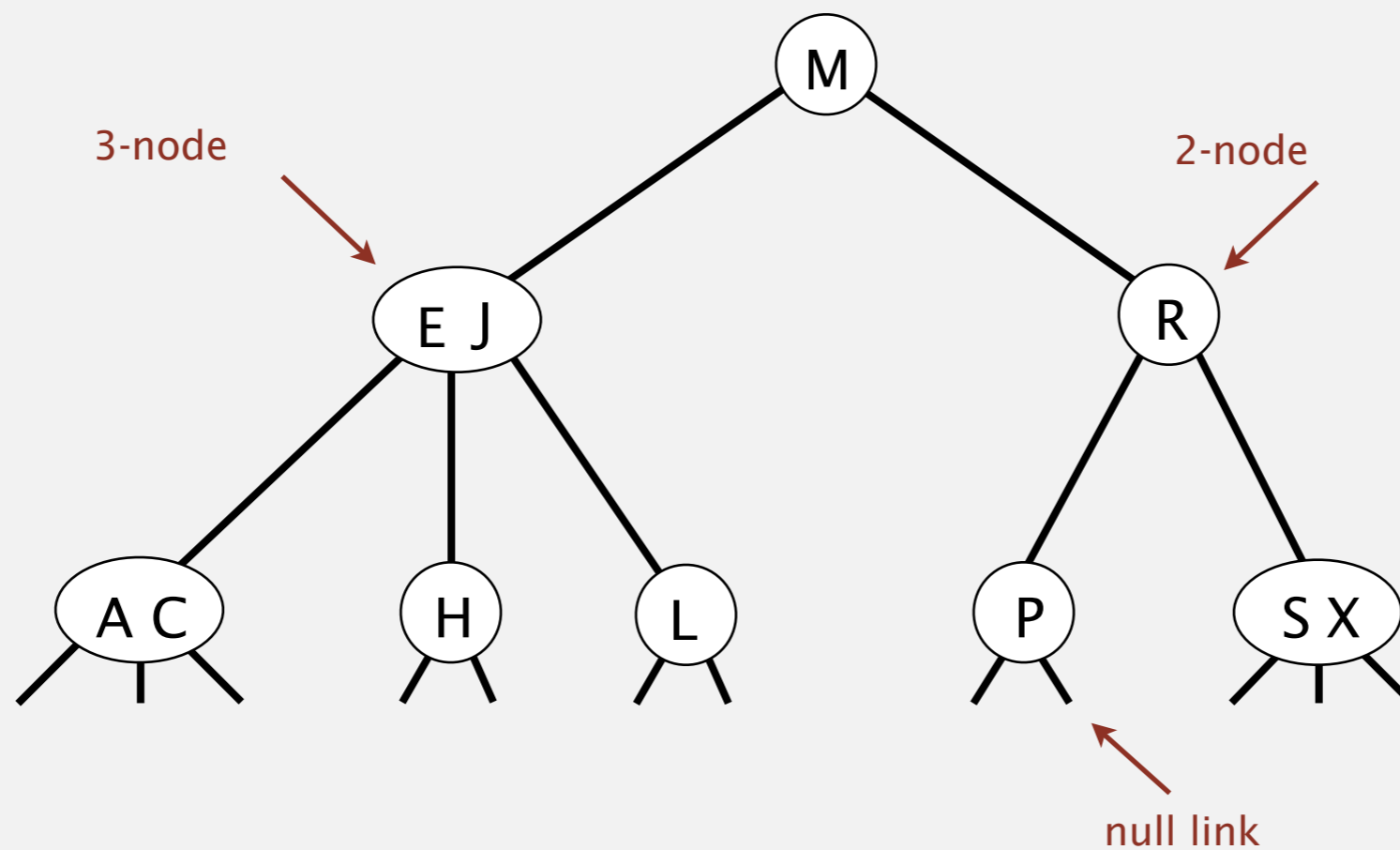‣ Challenge. Guarantee performance.

Balanced Search Trees

- ▸ **2-3 search trees**
- ▸ Red-black BSTs
- ▸ B-trees
- ▸ Geometric applications of BSTs

2-3 tree

You can read it as 2 or 3 children tree

Allow 1 or 2 keys per node.

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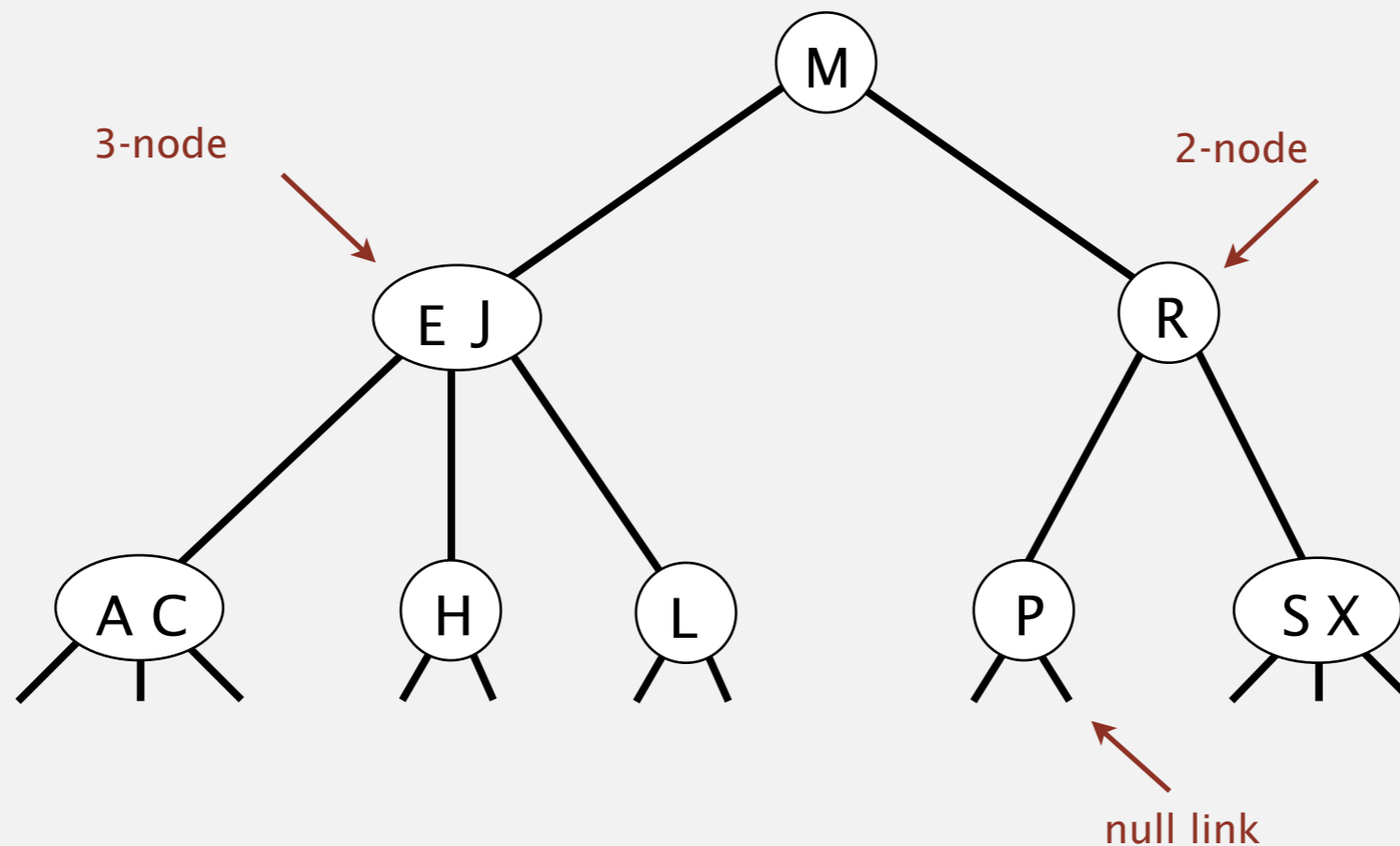- 3-node: two keys, three children.

2-3 tree

Allow 1 or 2 keys per node.

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

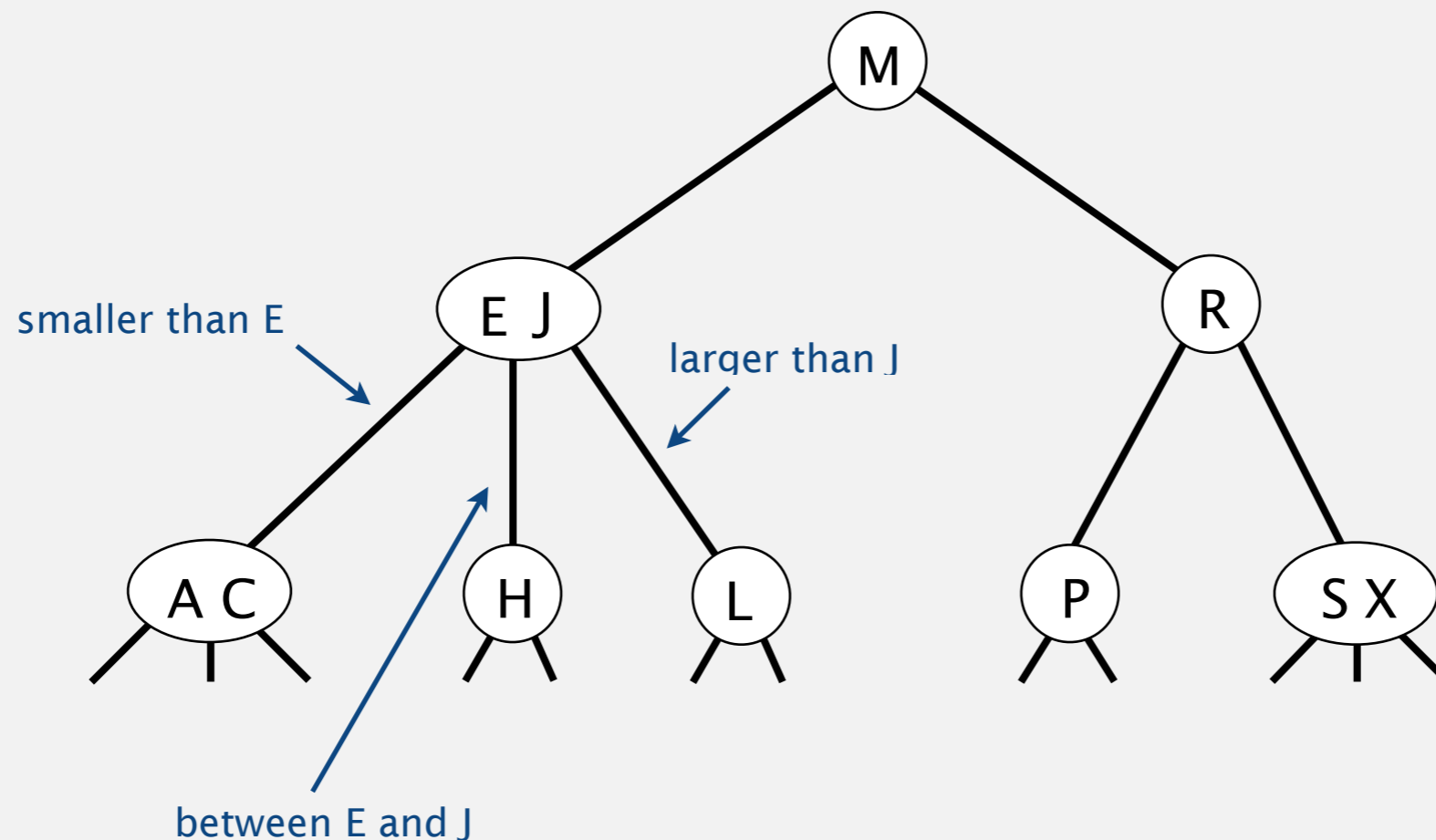Our Aim is Perfect balance. Every path from root to null link has same length.

2-3 tree

Allow 1 or 2 keys per node.
- 2-node: one key, two children.
- 3-node: two keys, three children.

Perfect balance. Every path from root to null link has same length.
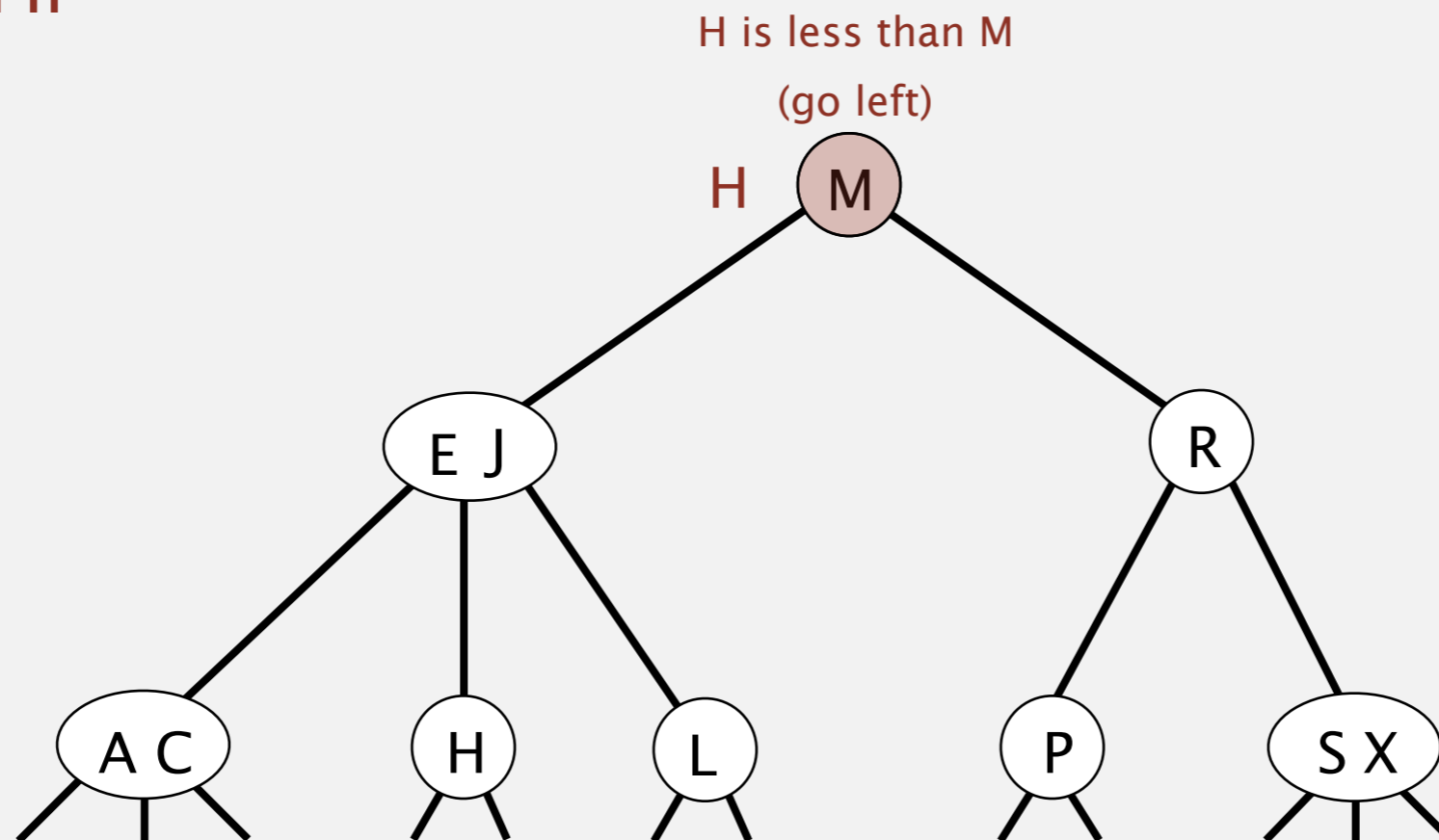Symmetric order. Inorder traversal yields keys in ascending order.

smaller than E

larger than J

between E and J

2-3 tree demo

Search.

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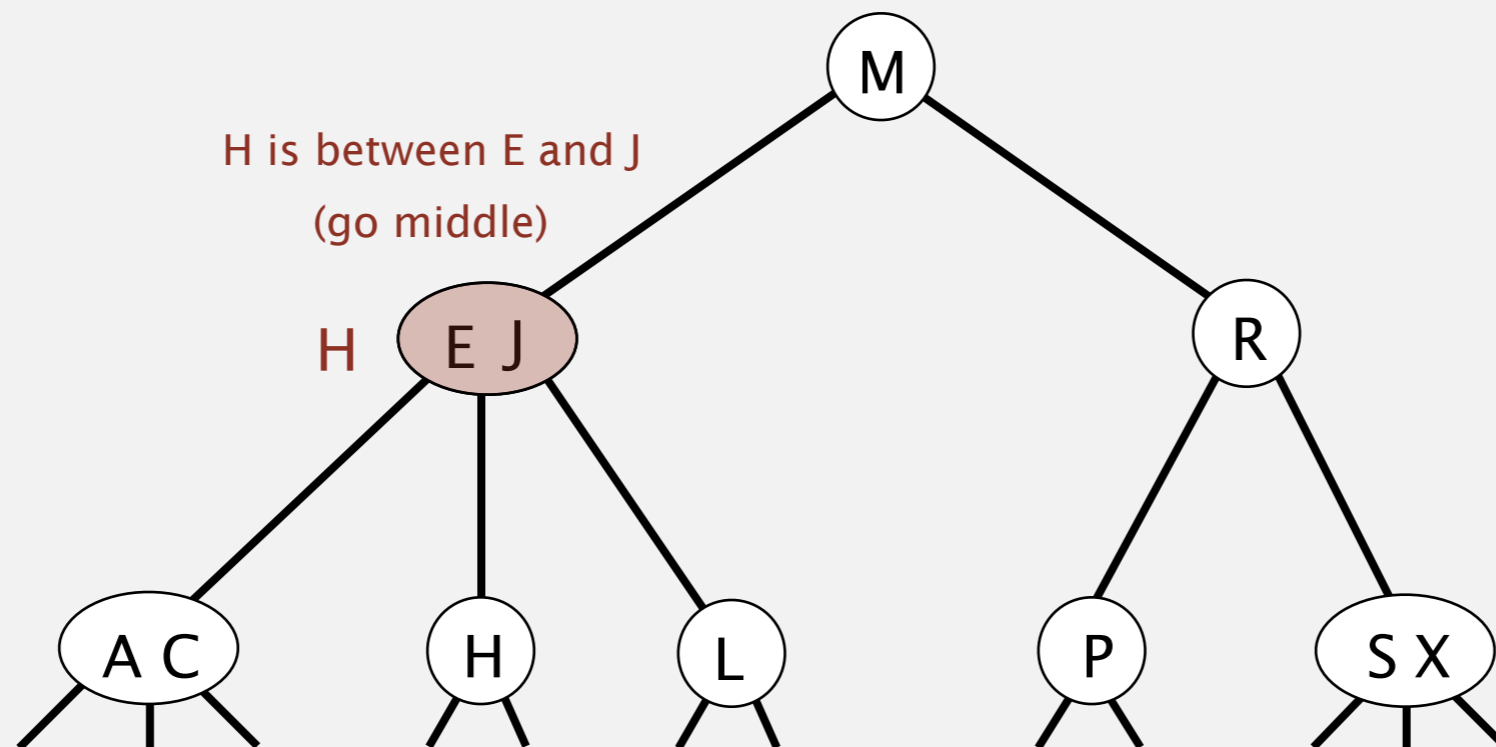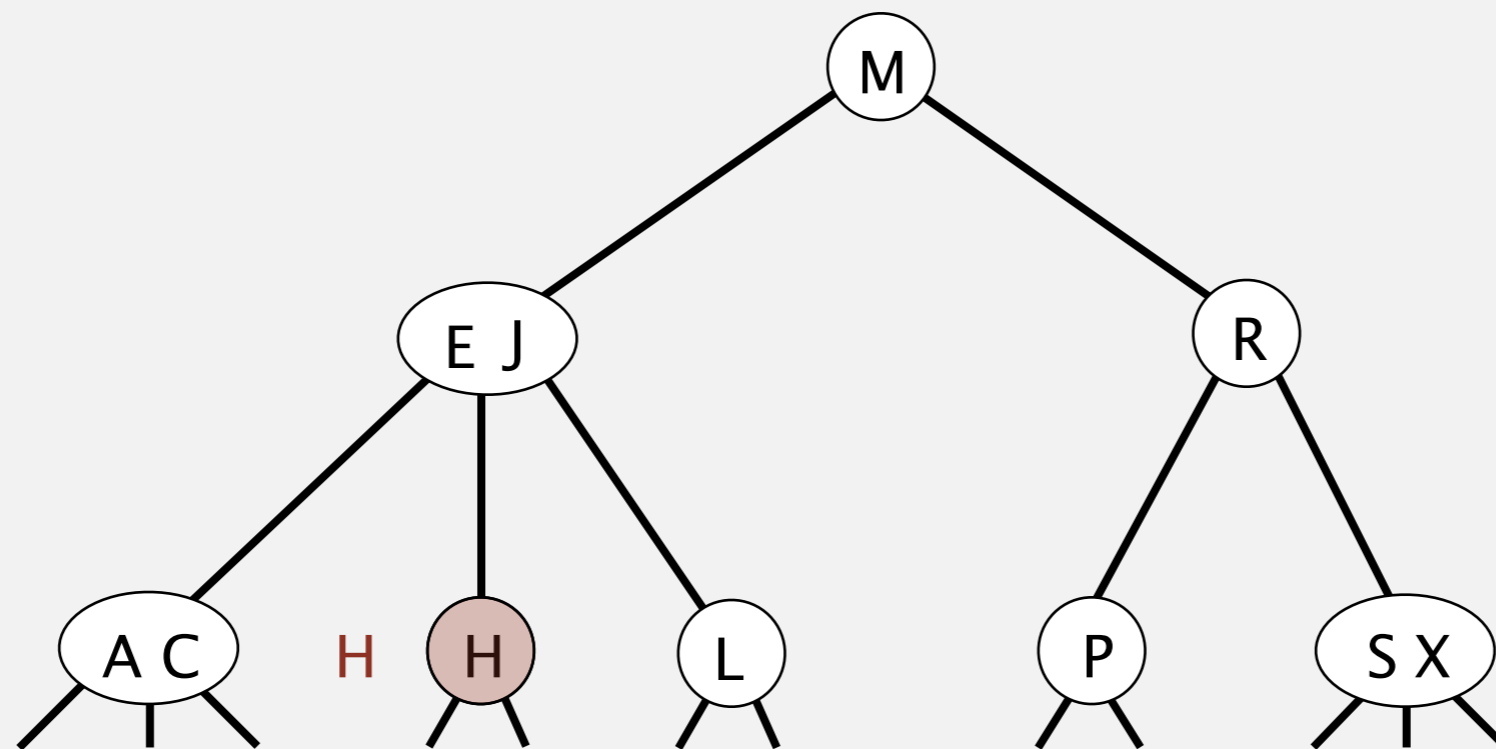- Follow associated link (recursively).

search for H

H is less than M
(go left)

2-3 tree demo

Search.

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

search for H

H is between E and J
(go middle)

2-3 tree demo

Search.

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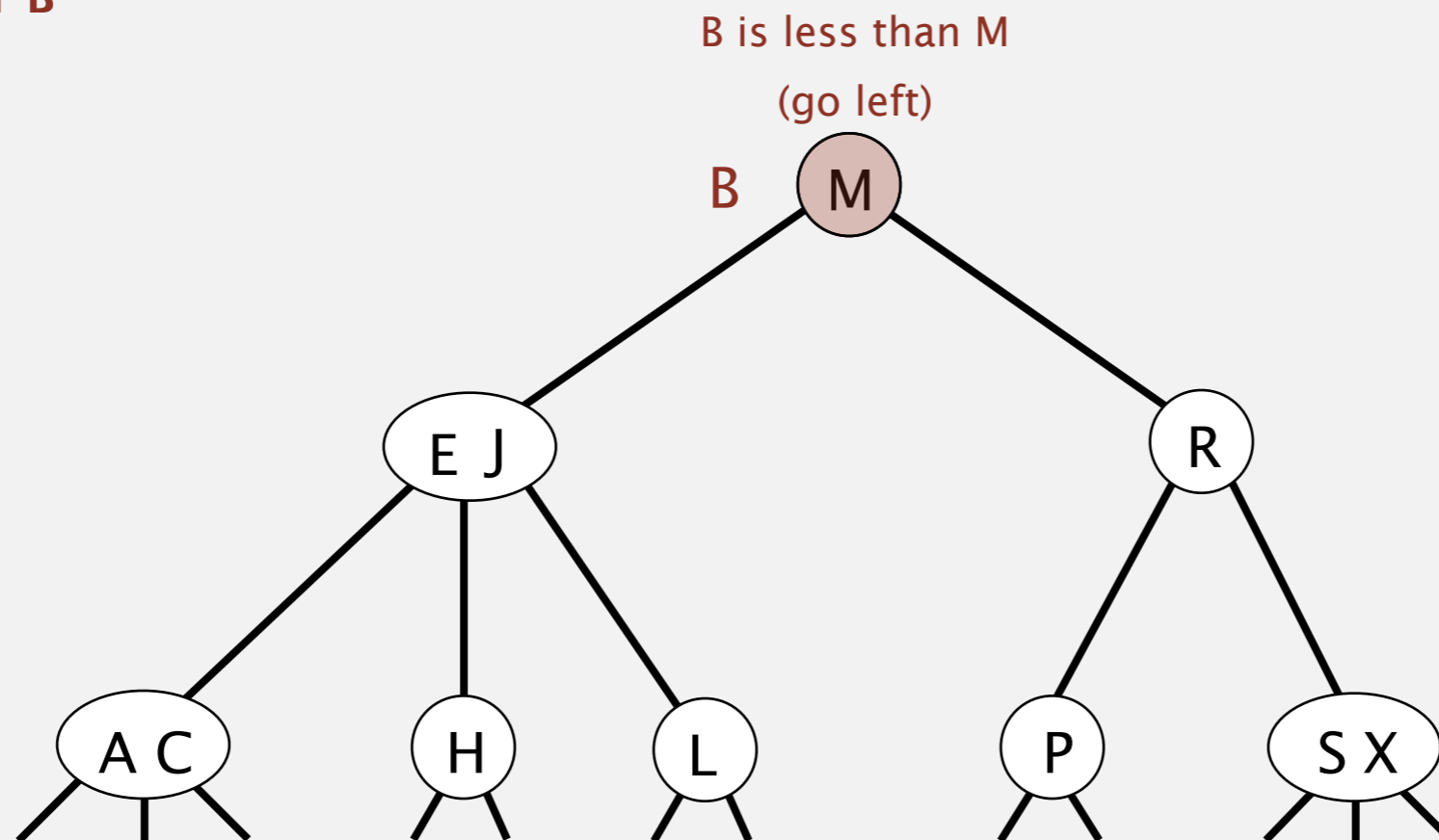- Follow associated link (recursively).
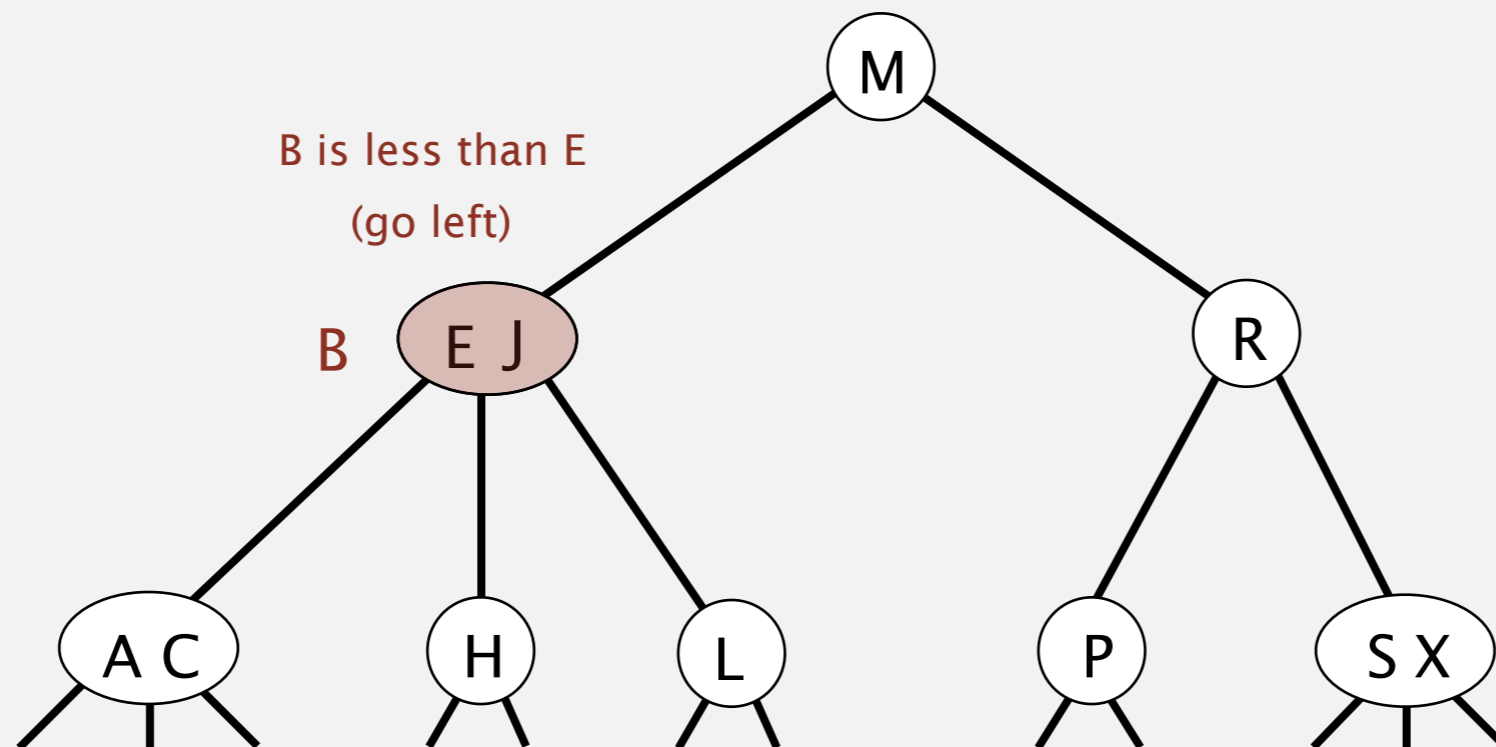
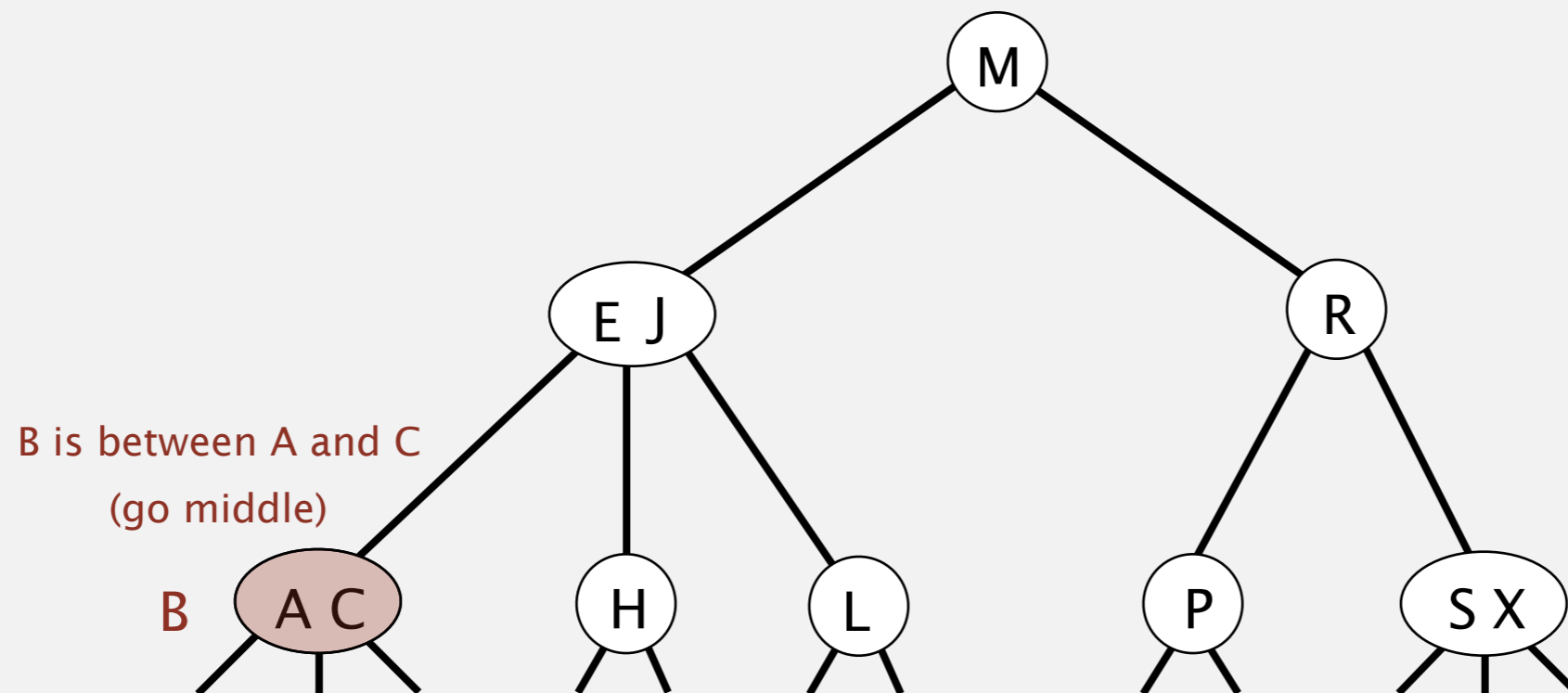**search for H**

found H
(search hit)

2-3 tree demo

Search.

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

search for B

B is less than M
(go left)

2-3 tree demo

Search.

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

search for B

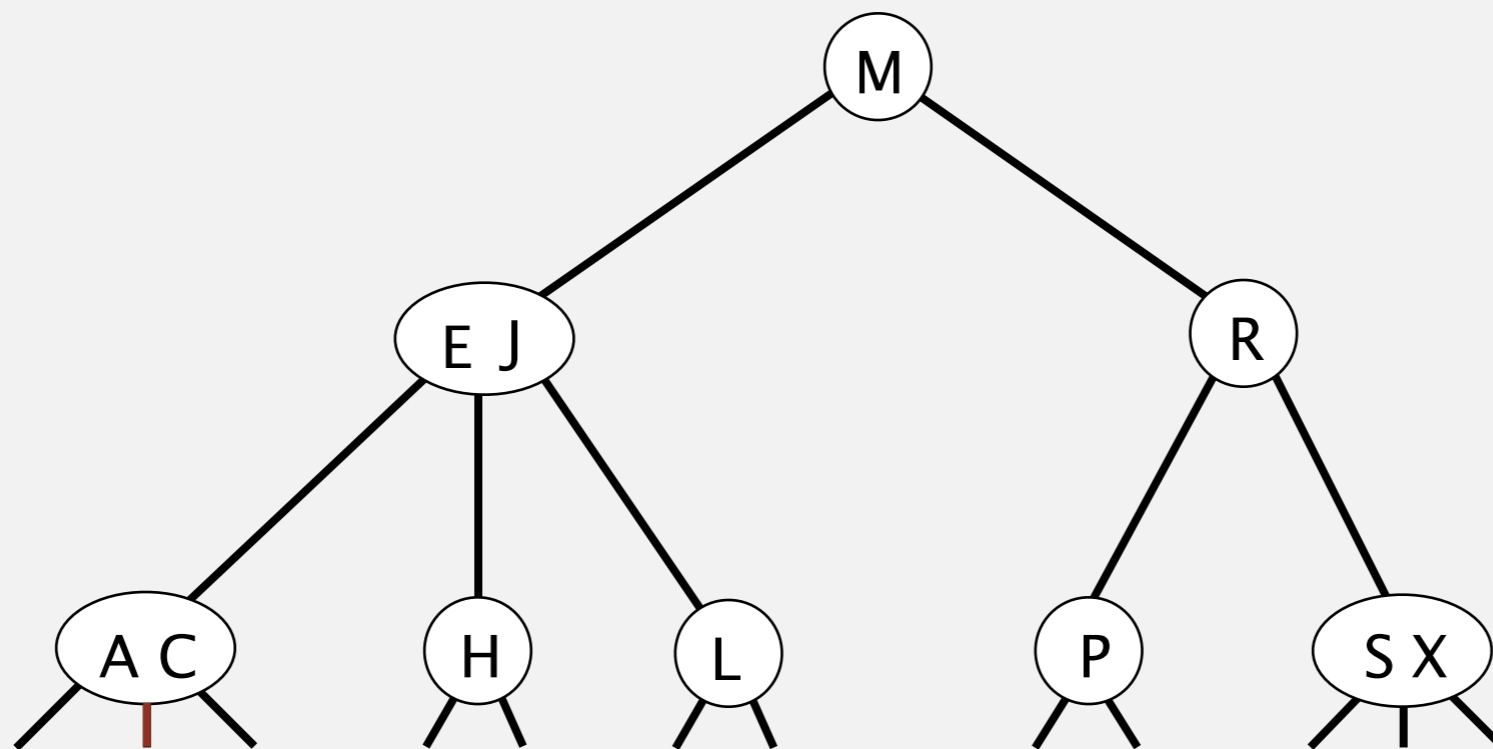B is less than E
(go left)

B

2-3 tree demo

Search.

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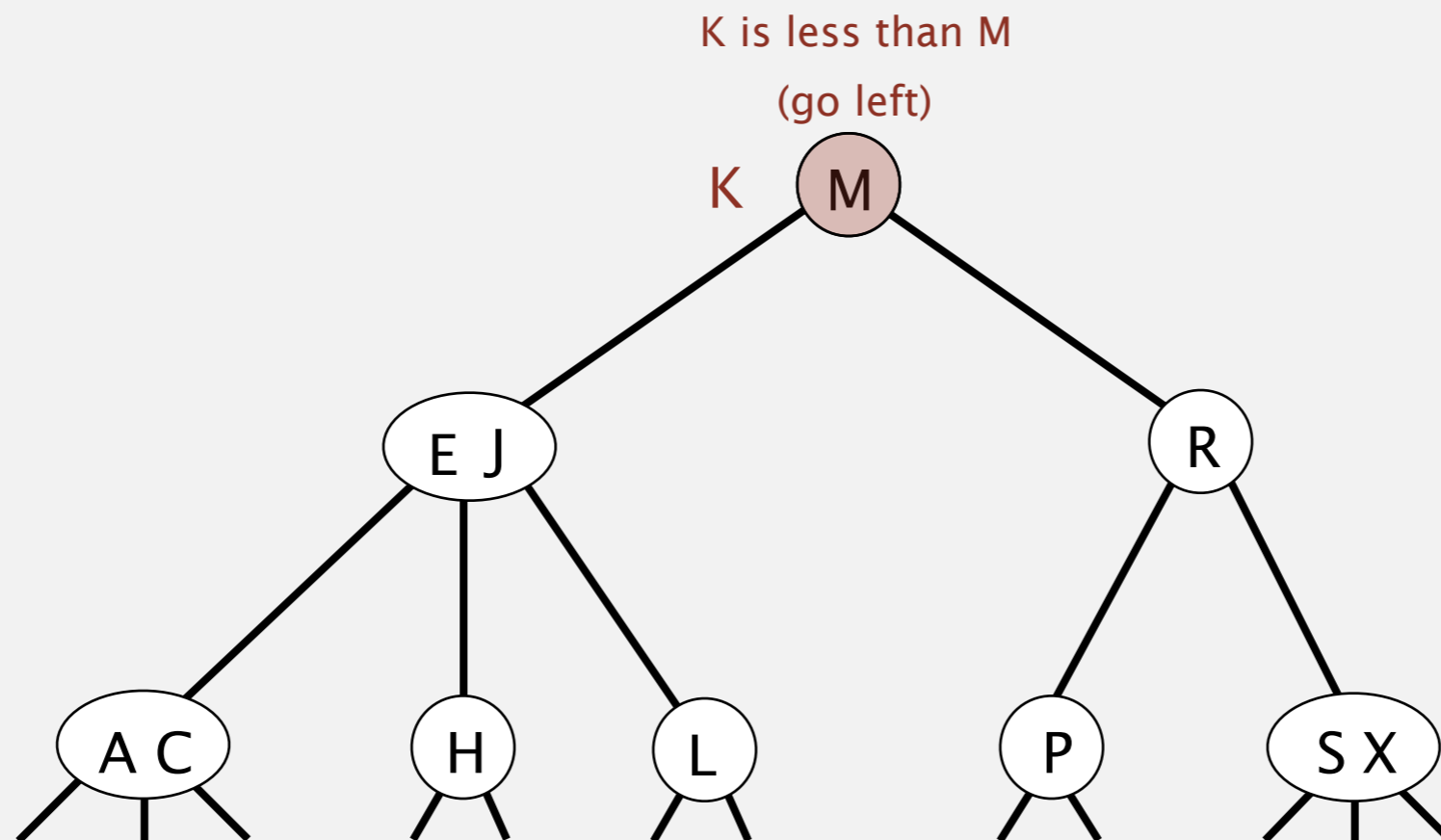- Follow associated link (recursively).

search for B

M

E J   R

B is between A and C
(go middle)

B   A C   H   L   P   S X

2-3 tree demo

Search.

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

search for B

M

E J

R

A C

H

L

P

S X

B

link is null
(search miss)

2-3 tree demo

Insert into a 2-node at bottom.

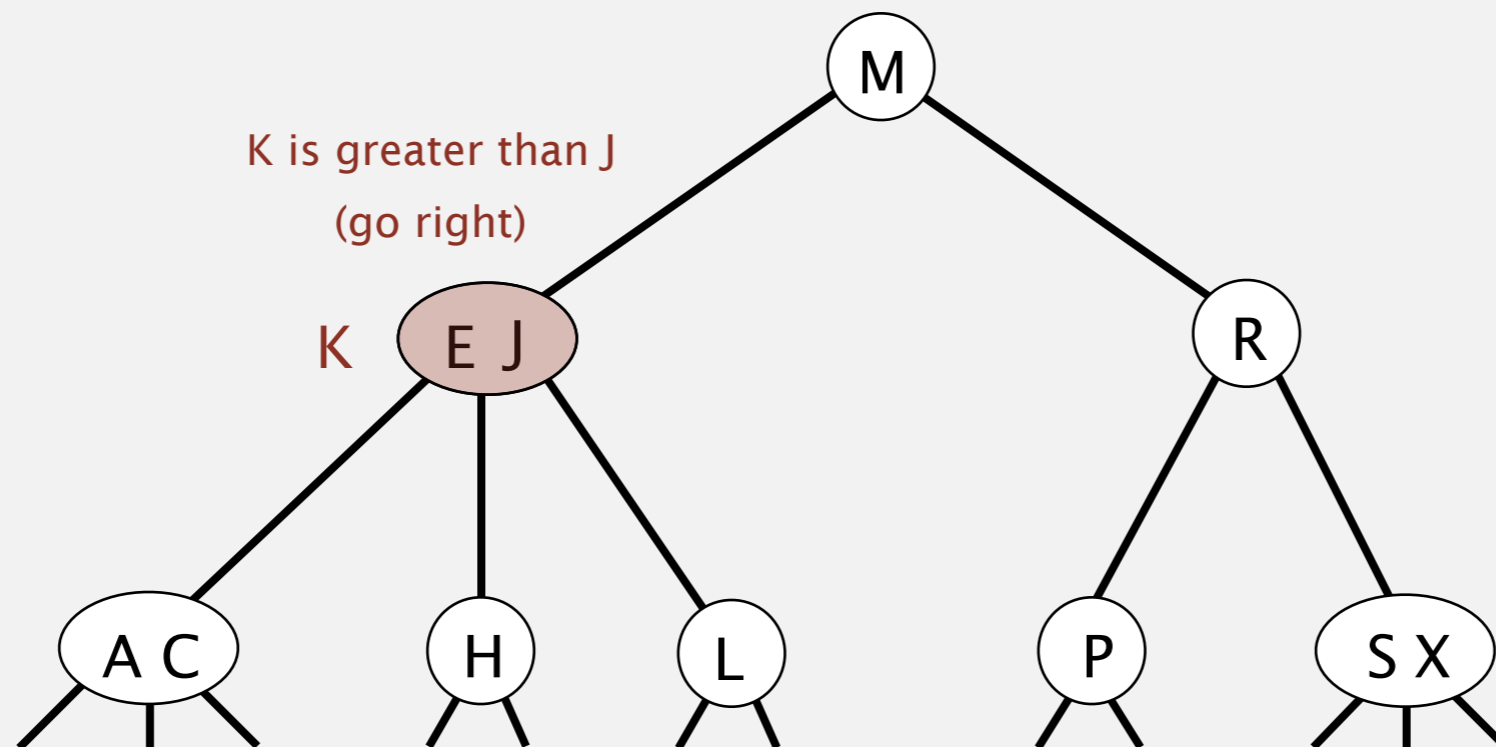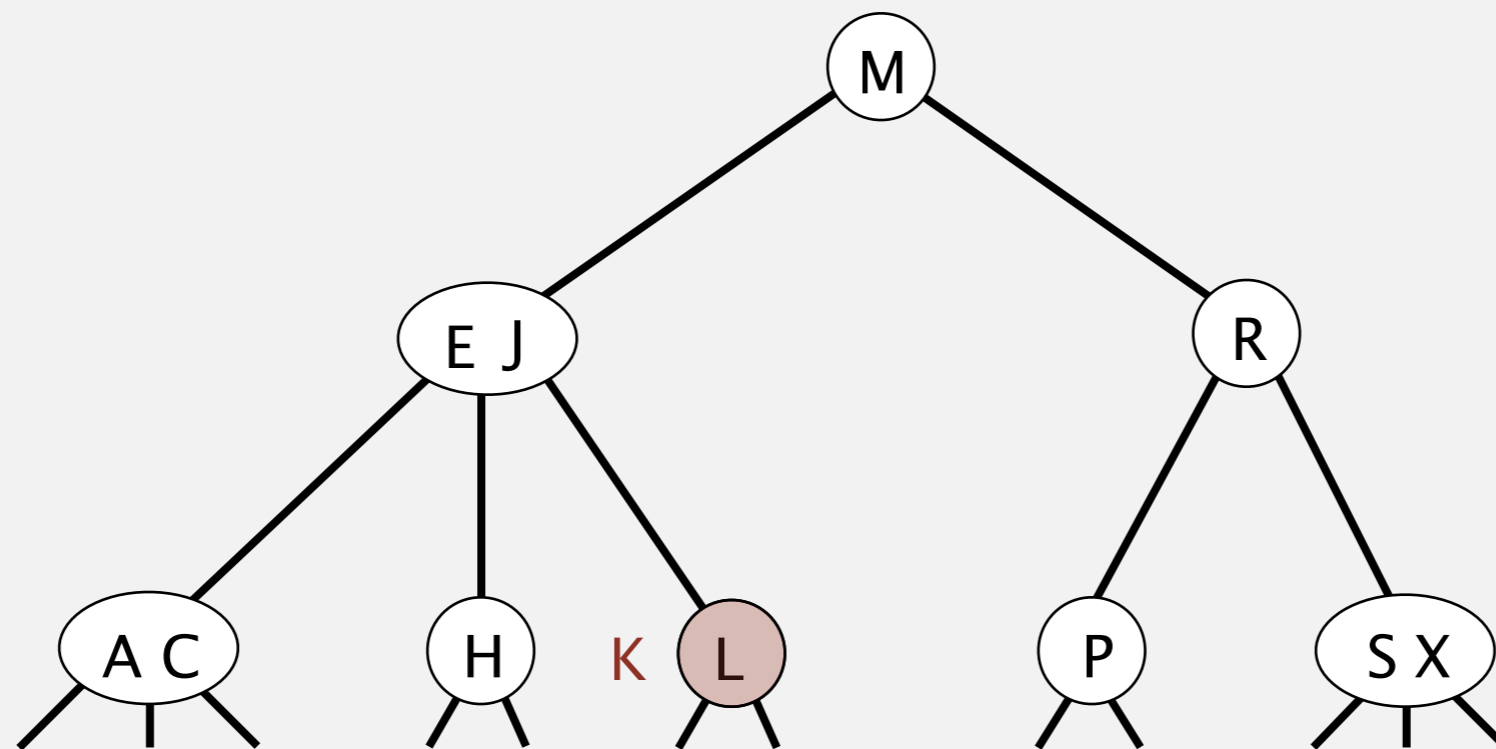- Search for key, as usual.
- Replace 2-node with 3-node.

insert K

K is less than M
(go left)

K M

E J   R

A C   H   L   P   S X

2-3 tree demo

Insert into a 2-node at bottom.

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

insert K

K is greater than J
(go right)

K

2-3 tree demo

Insert into a 2-node at bottom.

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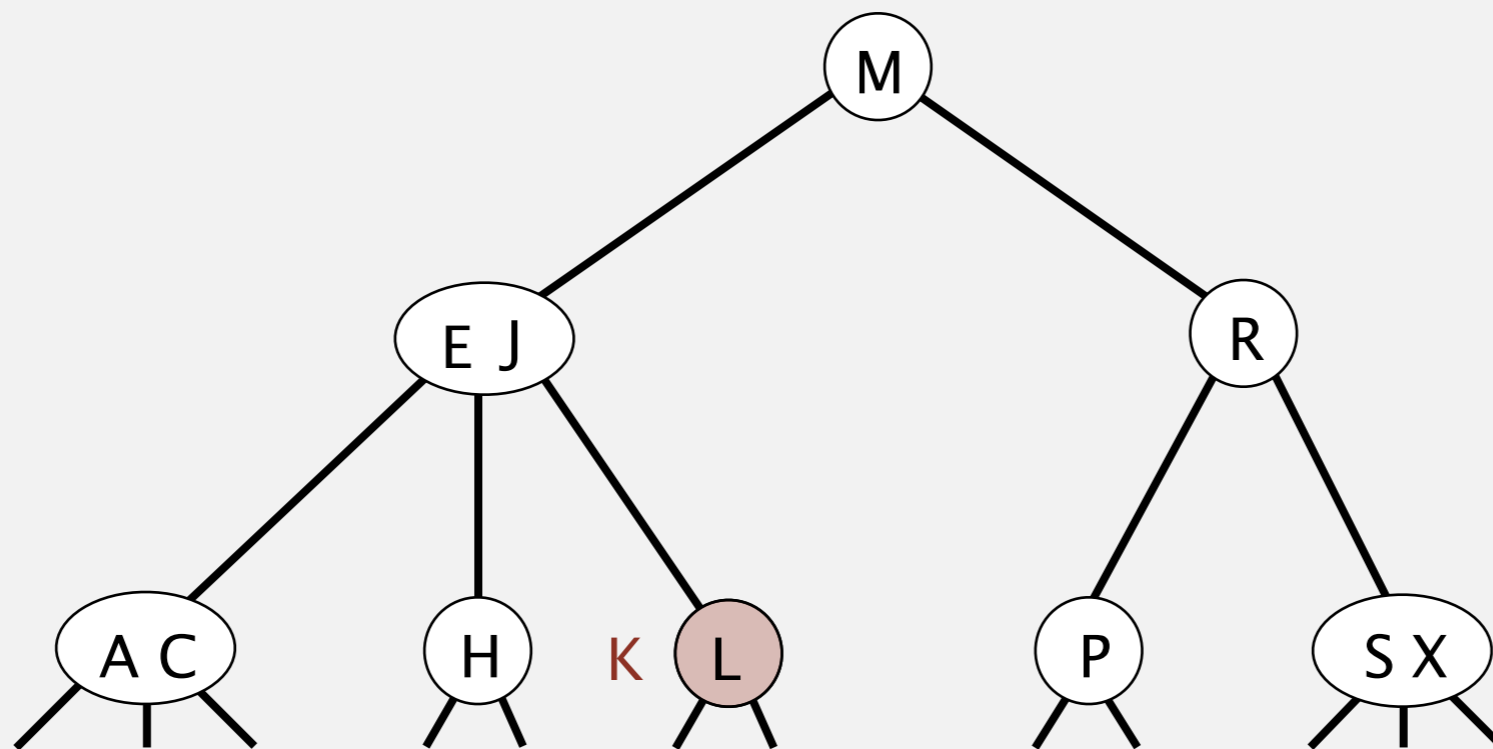- Replace 2-node with 3-node.

insert K

search ends here

2-3 tree demo

Insert into a 2-node at bottom.

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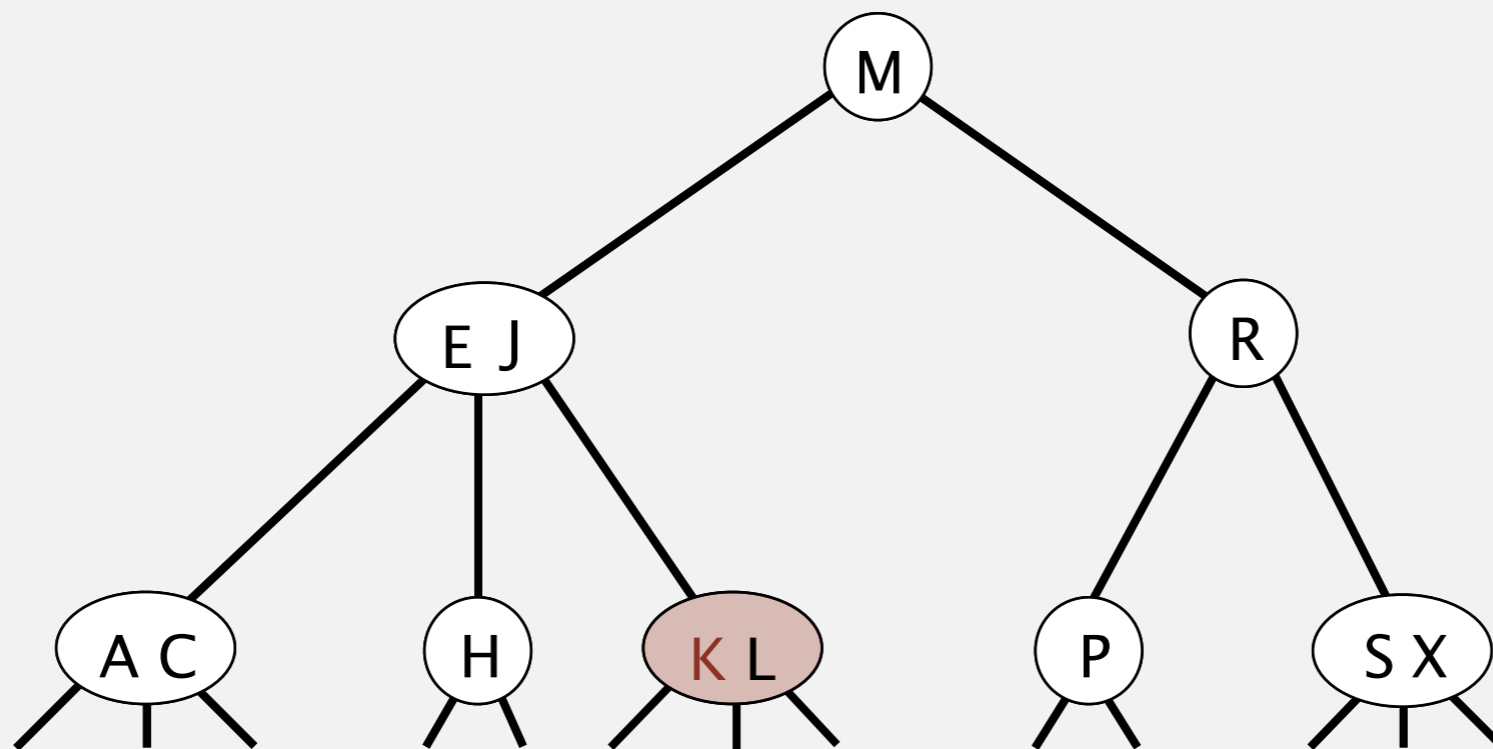- Replace 2-node with 3-node.

insert K

replace 2-node with
3-node containing K

2-3 tree demo

Insert into a 2-node at bottom.

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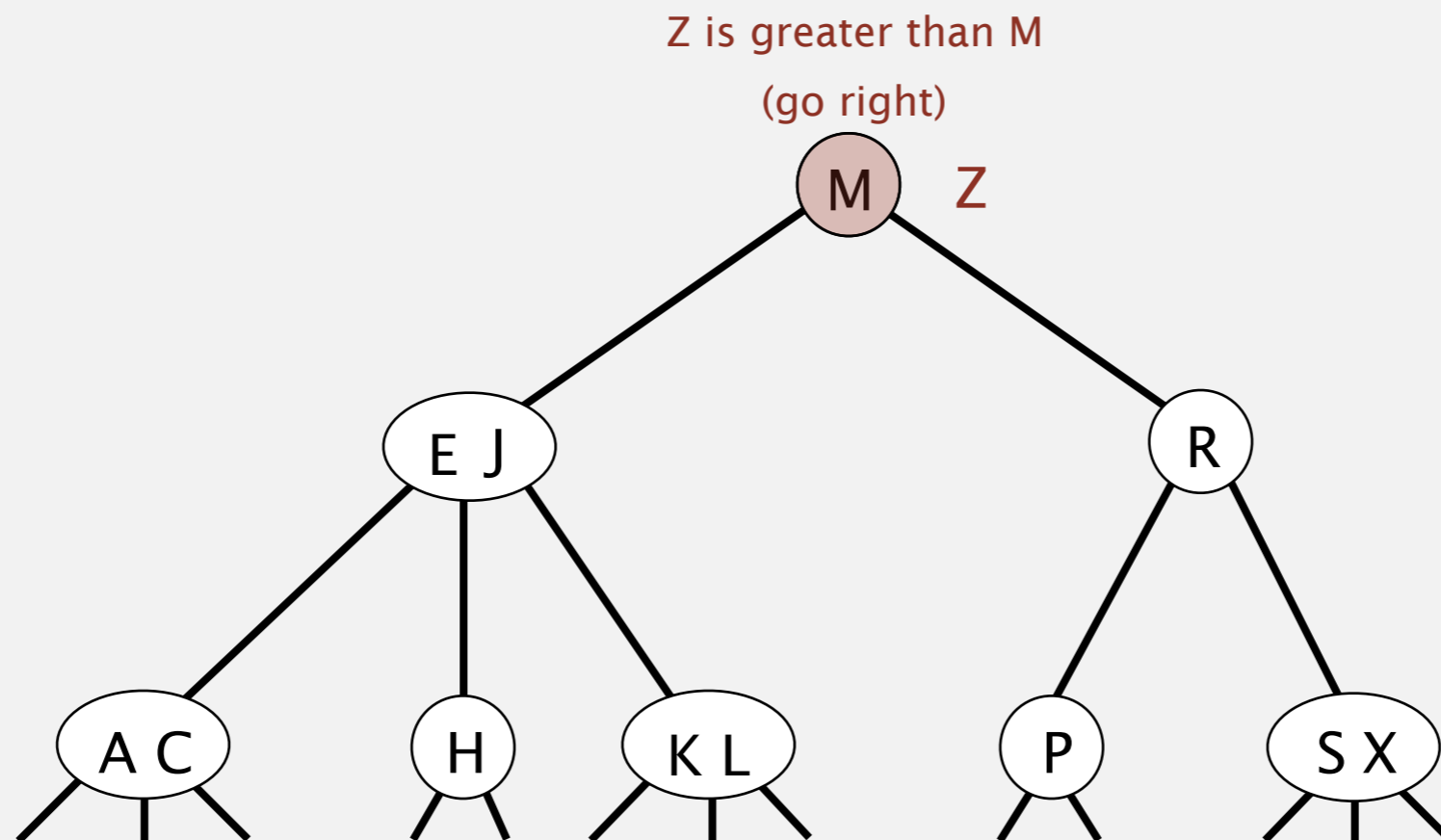- Replace 2-node with 3-node.

insert K

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

Z is greater than M
(go right)

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
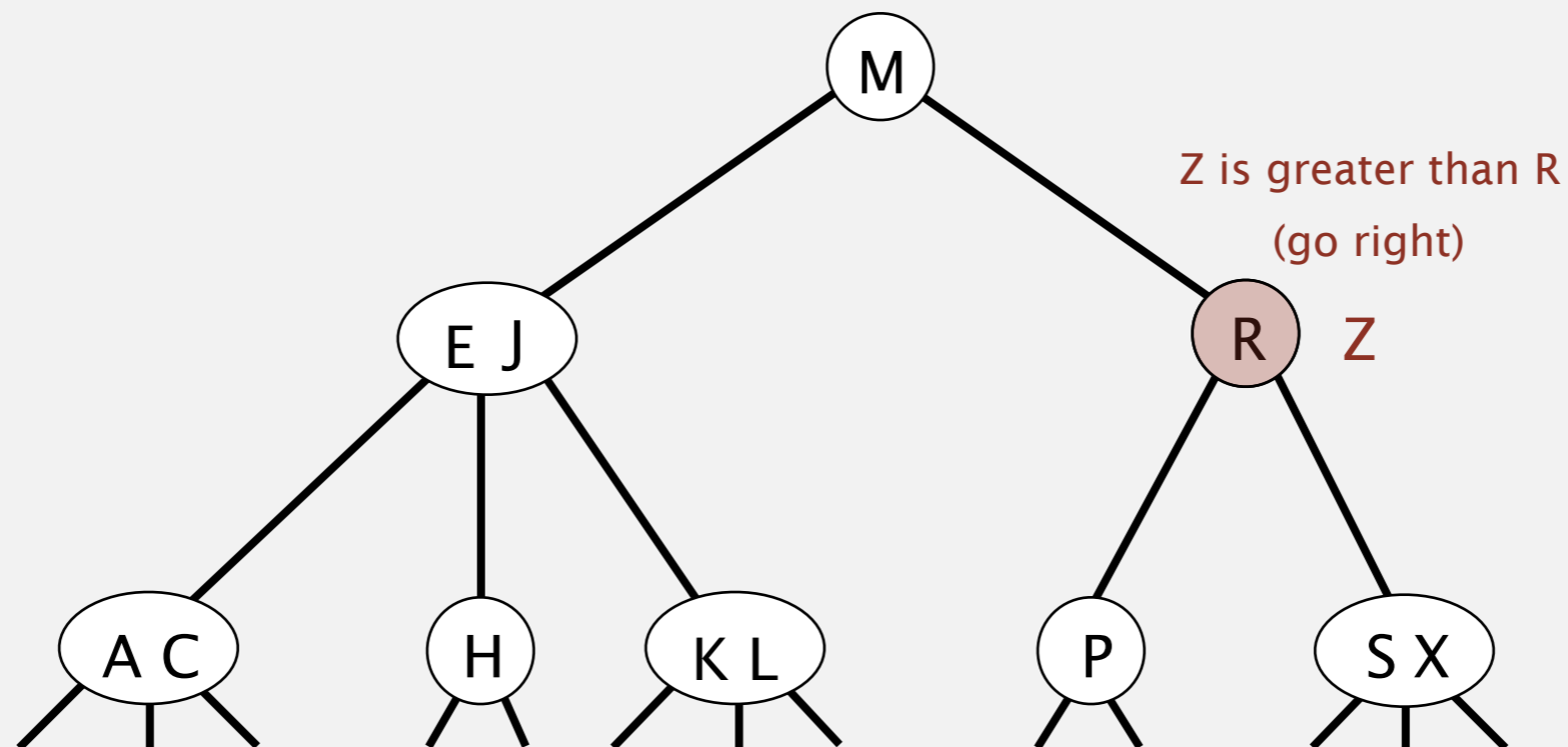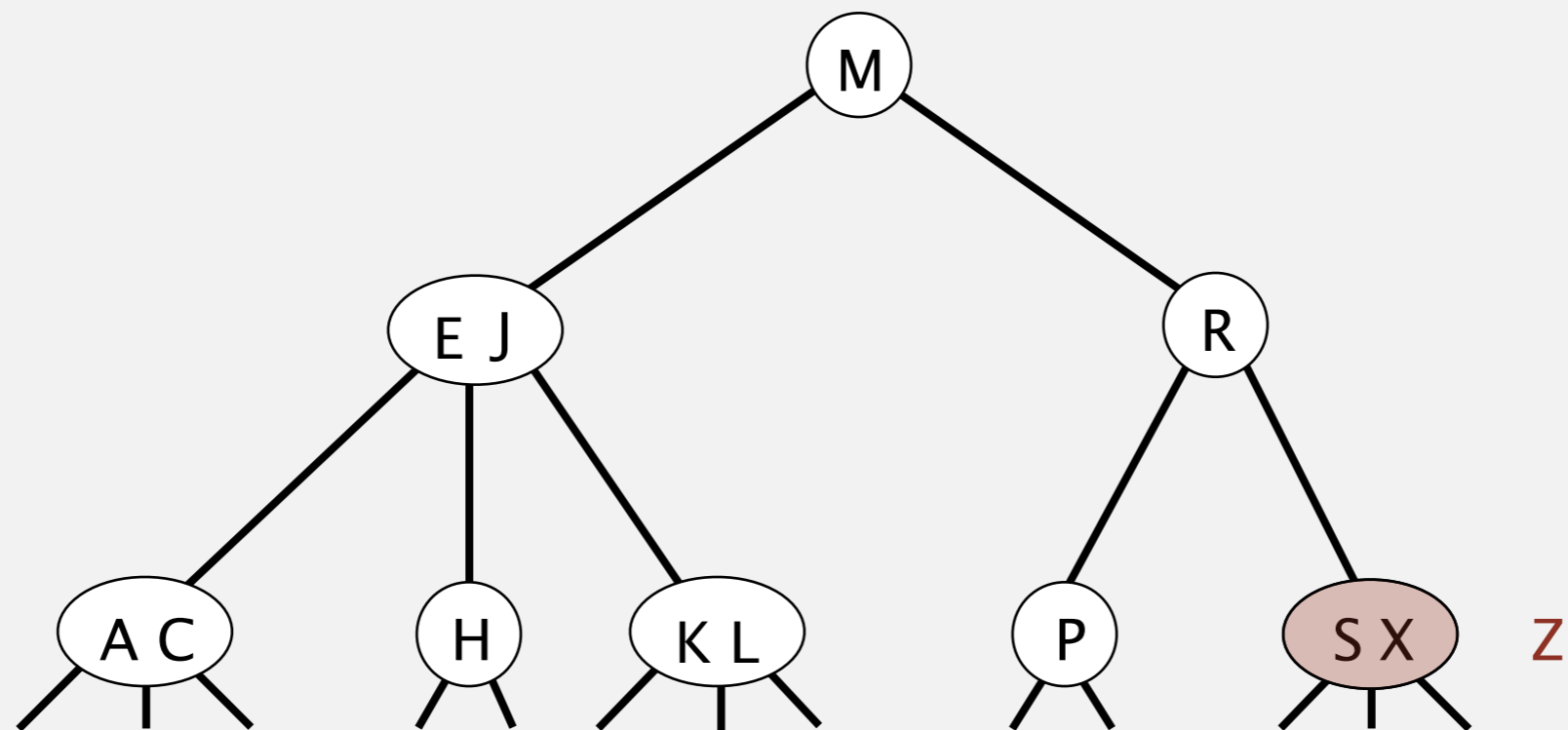- Move middle key in 4-node into parent.
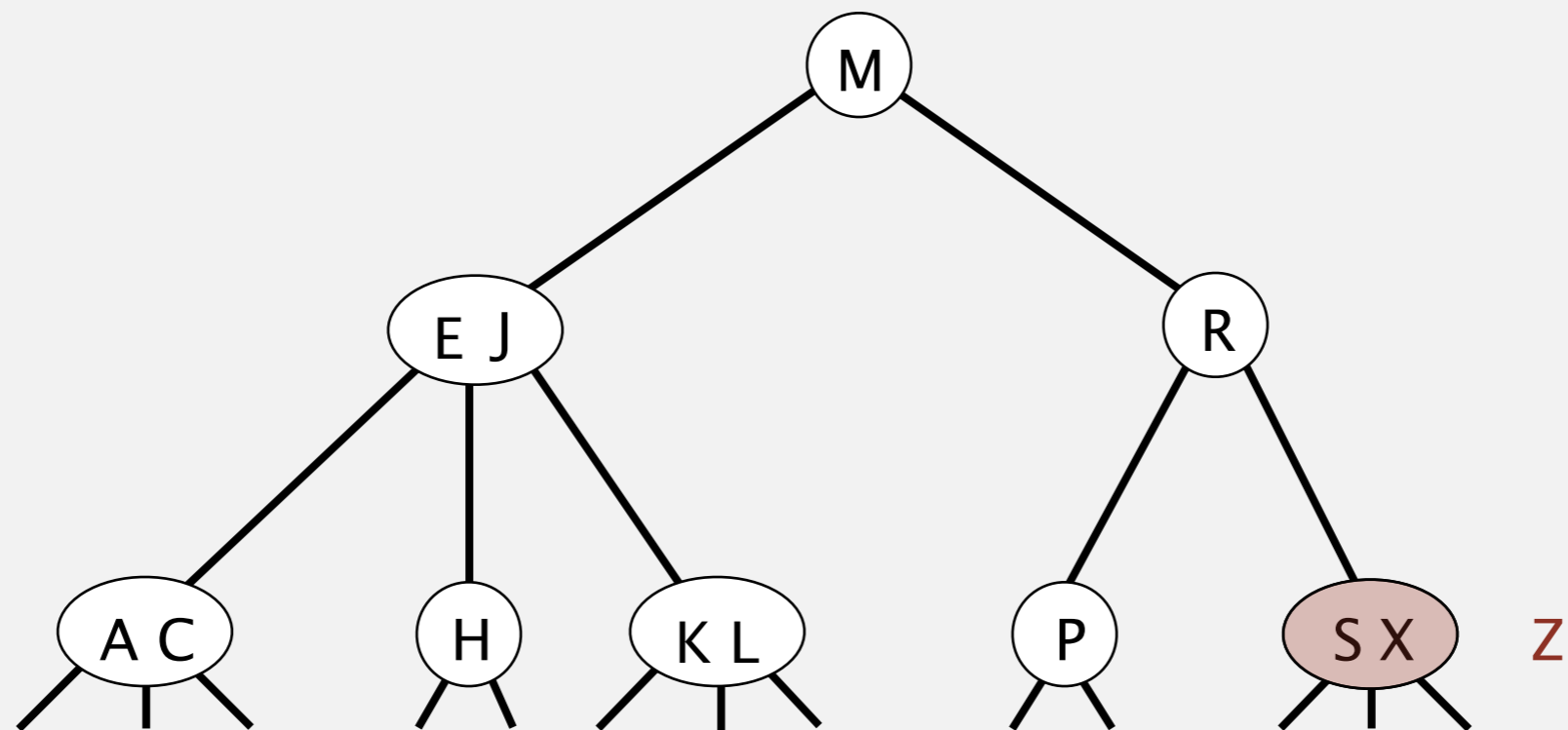
insert Z

Z is greater than R
(go right)

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

search ends here

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
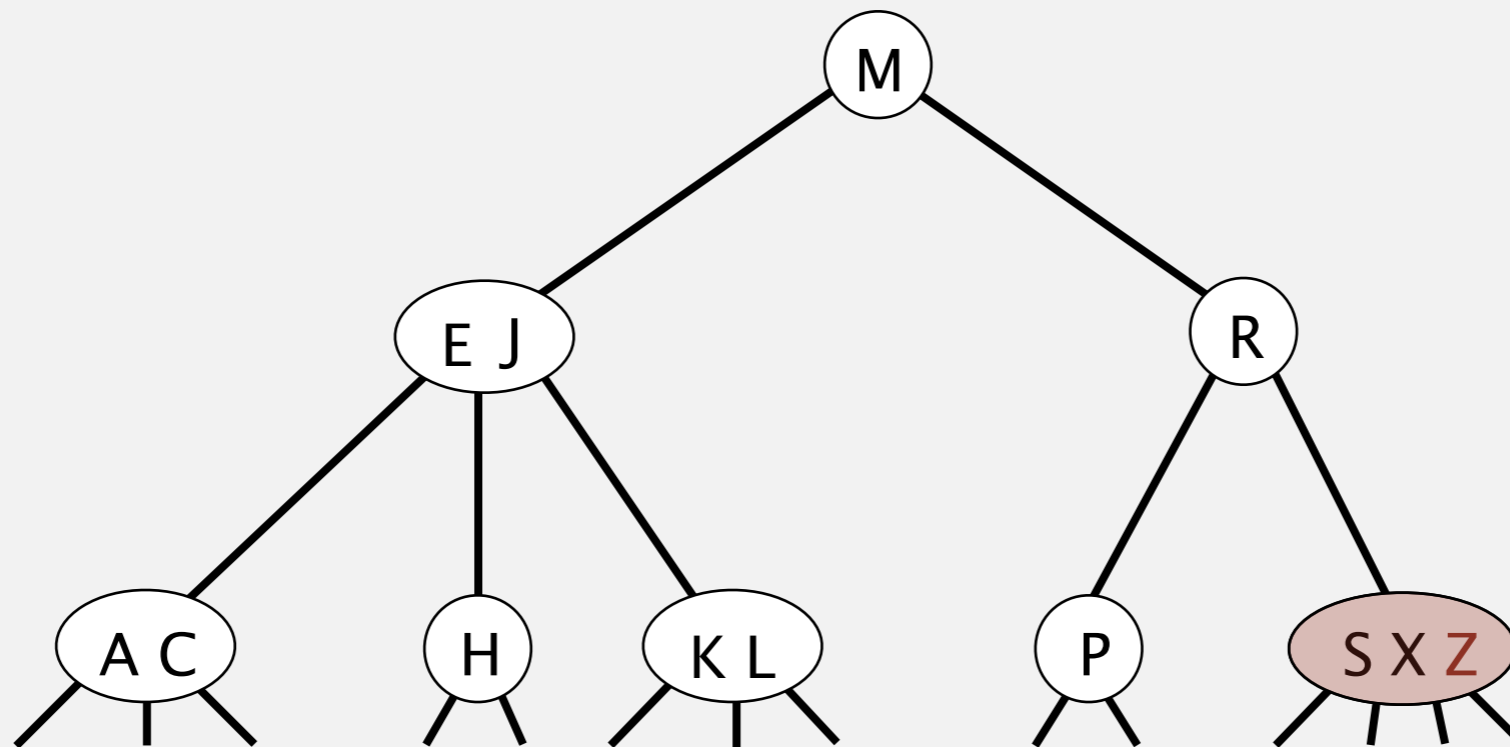- Move middle key in 4-node into parent.

insert Z

replace 3-node with
temporary 4-node containing Z

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
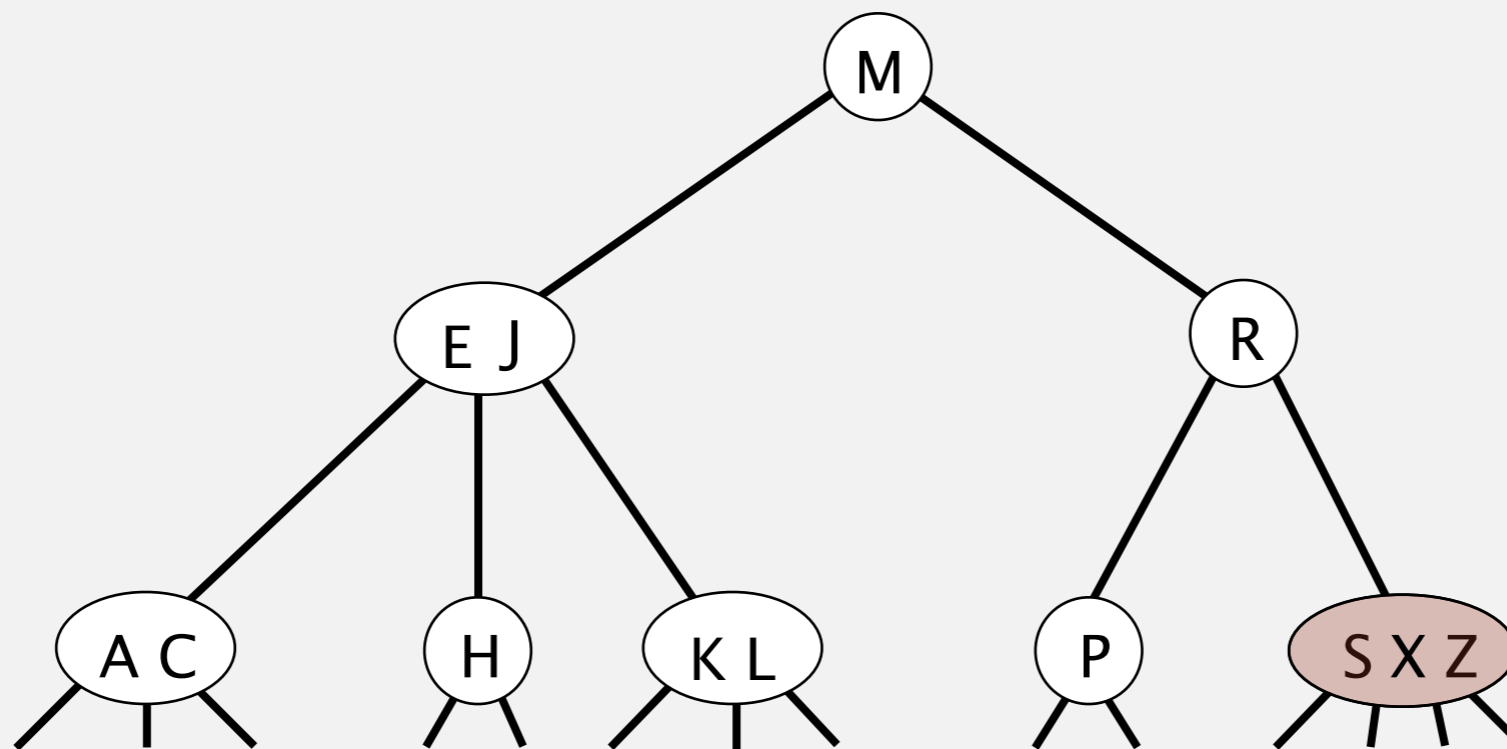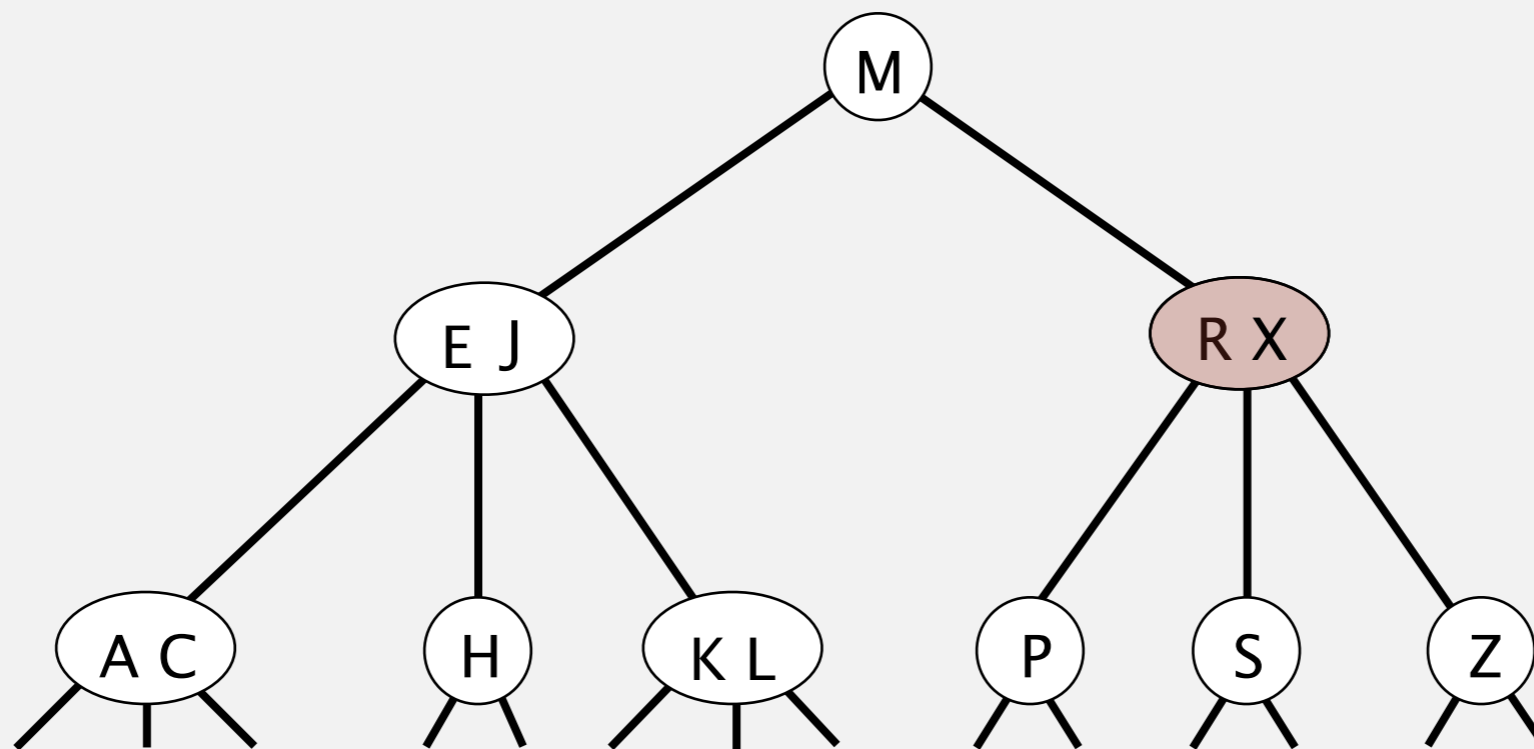- Move middle key in 4-node into parent.

insert Z

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

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

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
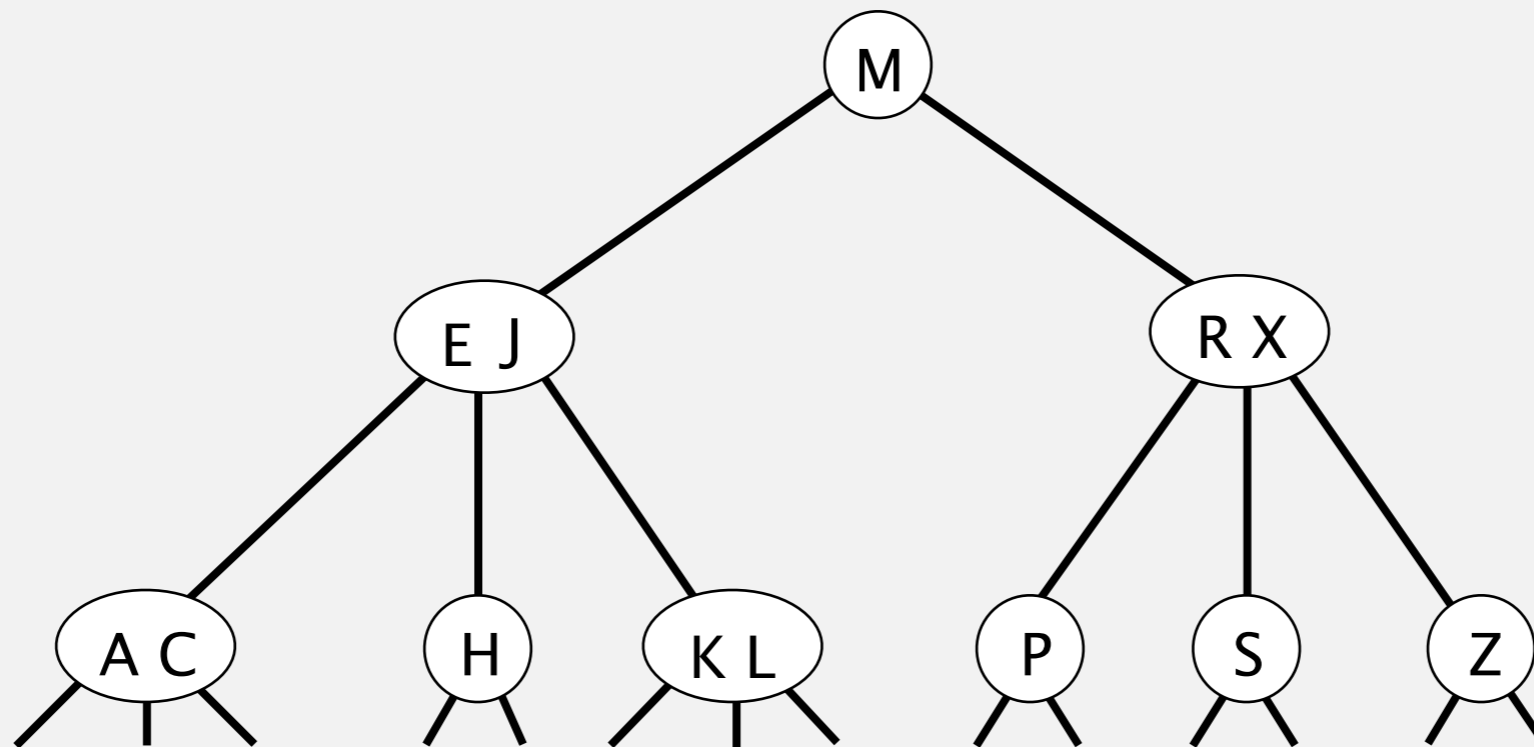- Move middle key in 4-node into parent.

insert Z

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
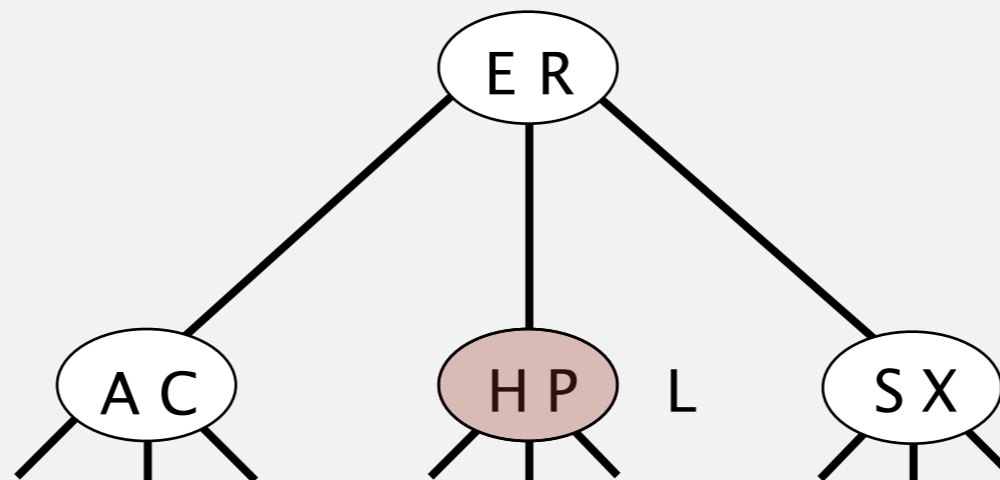- Move middle key in 4-node into parent.

insert Z

2-3 tree demo

Insert into a 3-node at bottom.

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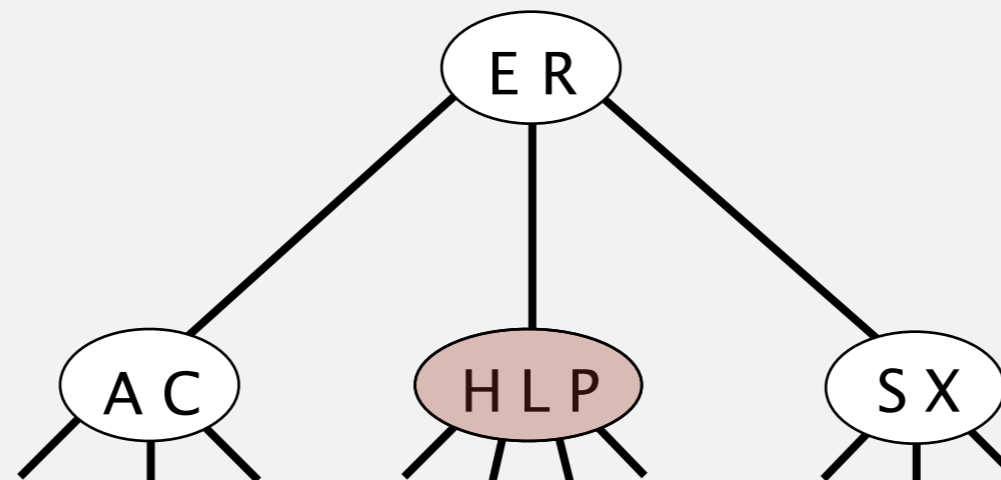
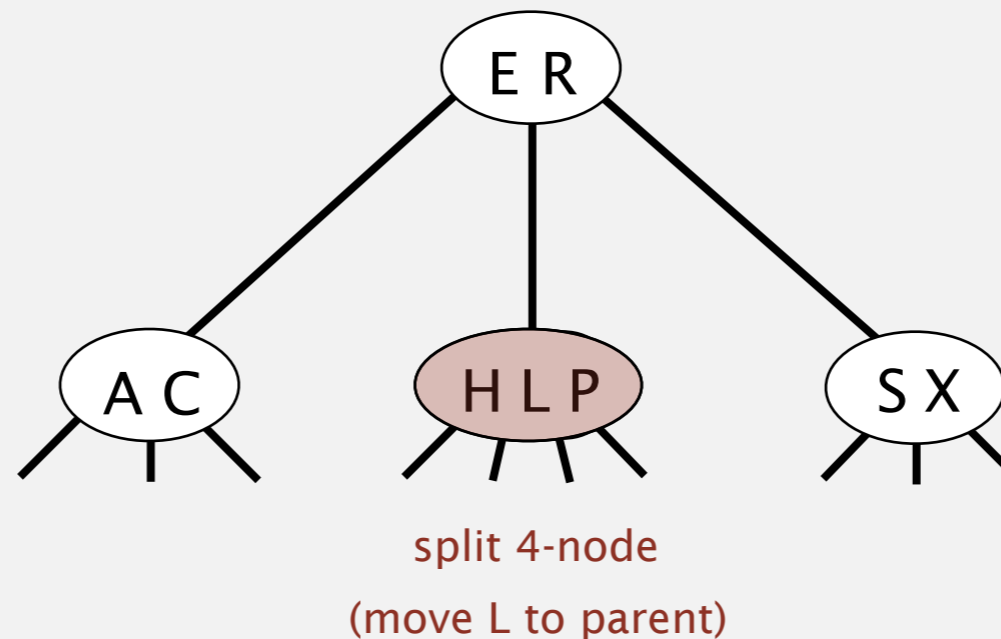**insert L**

convert 3-node into 4-node

2-3 tree demo

Insert into a 3-node at bottom.

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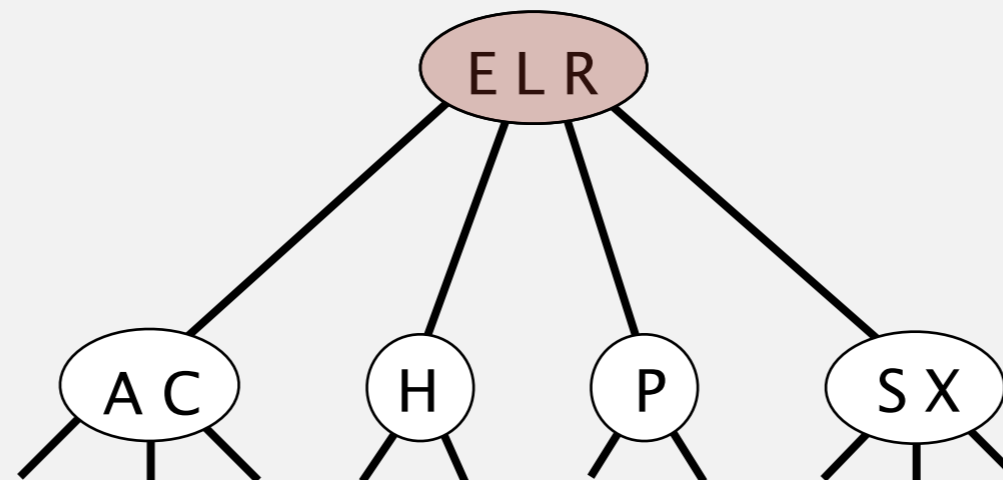
insert L

2-3 tree demo

Insert into a 3-node at bottom.

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

insert L
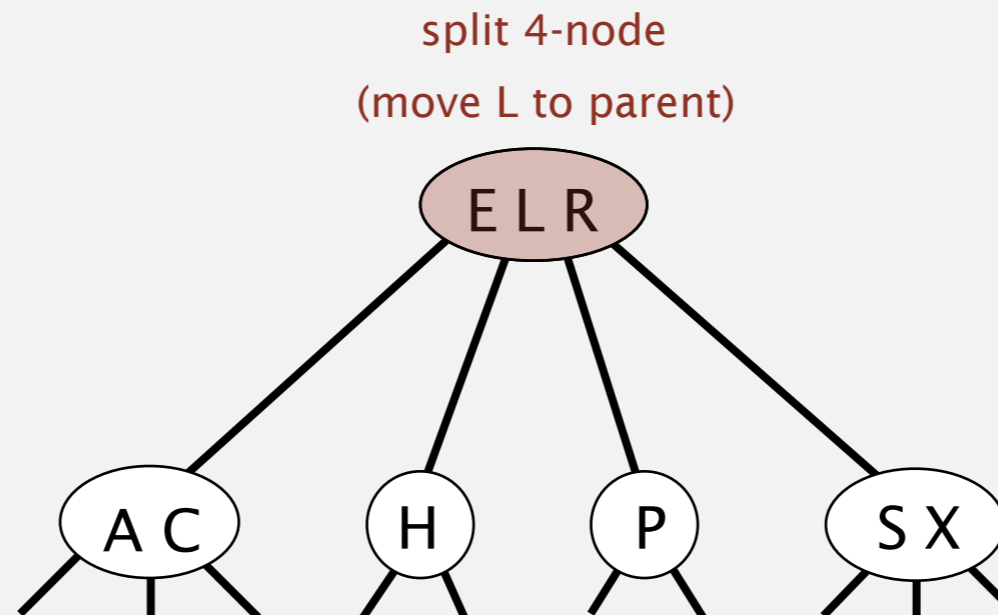

split 4-node

(move L to parent)

2-3 tree demo

Insert into a 3-node at bottom.

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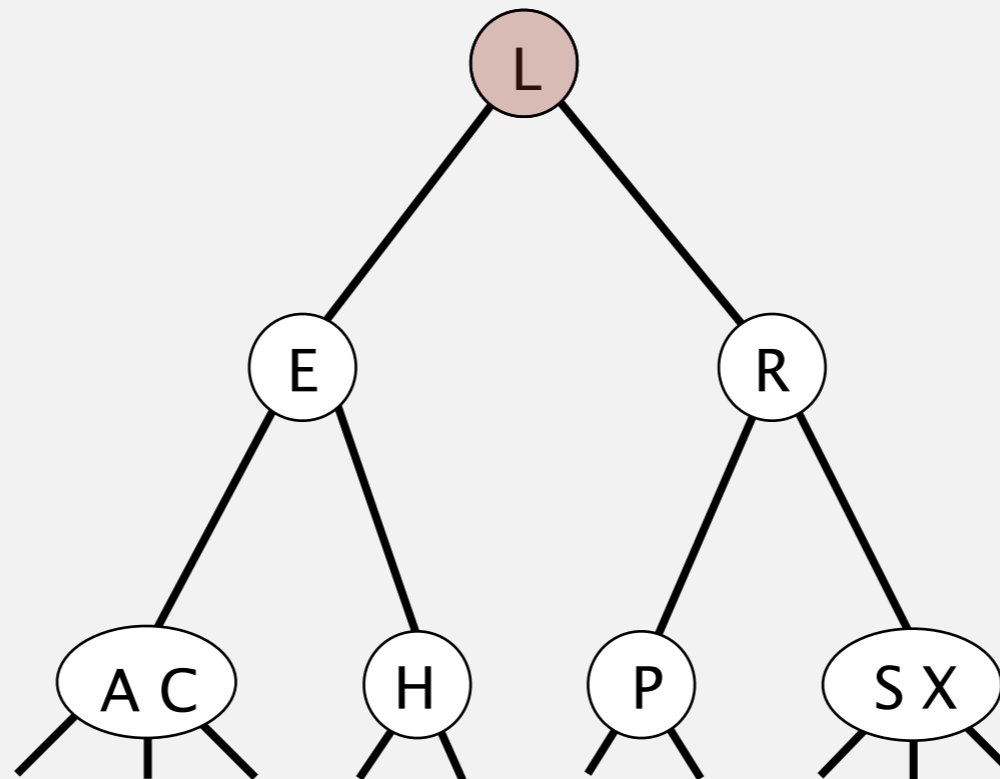
insert L

2-3 tree demo

Insert into a 3-node at bottom.

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

insert L

split 4-node
(move L to parent)
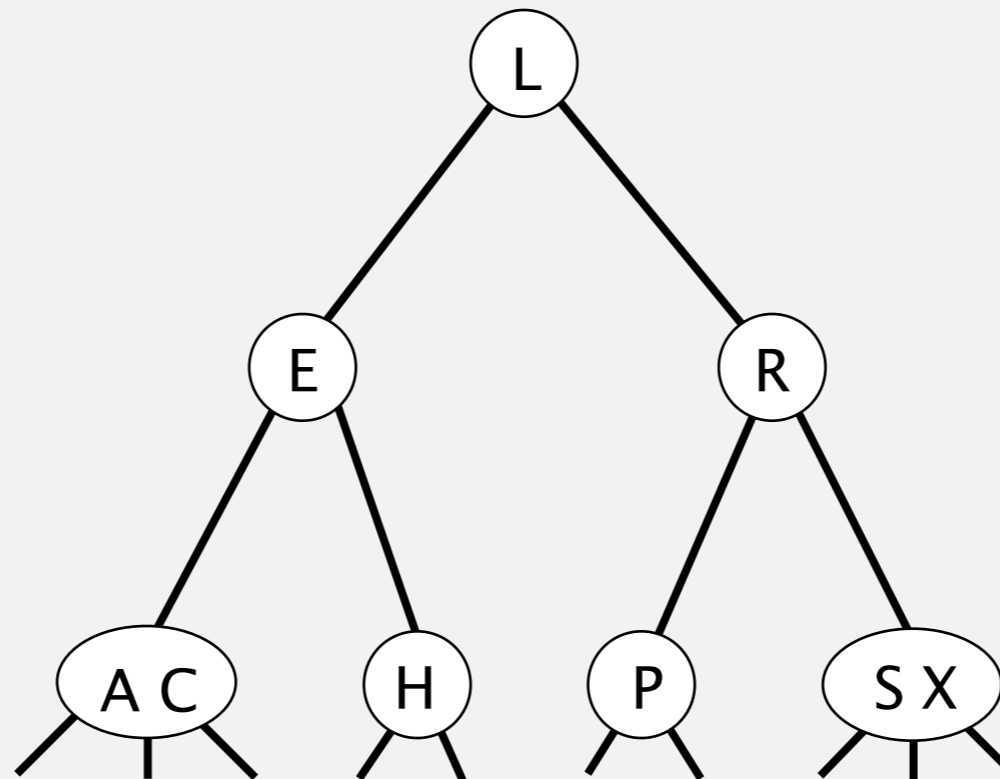
E L R

A C   H   P   S X

2-3 tree demo

Insert into a 3-node at bottom.

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

height of tree increases by 1

insert L

2-3 tree demo

Insert into a 3-node at bottom.

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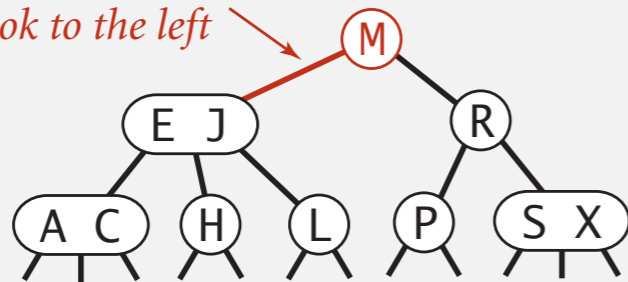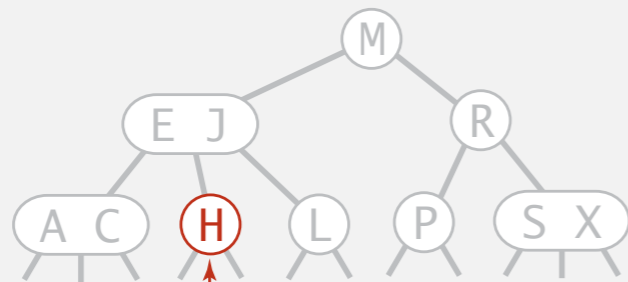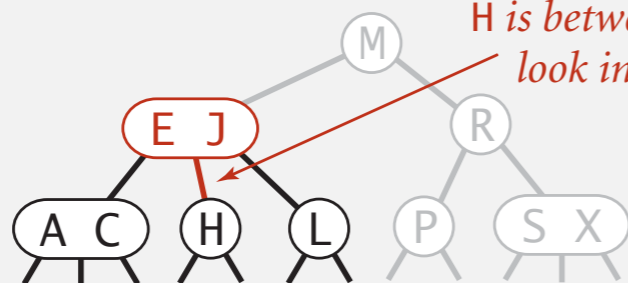
insert L

Search in a 2-3 tree

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

successful search for H
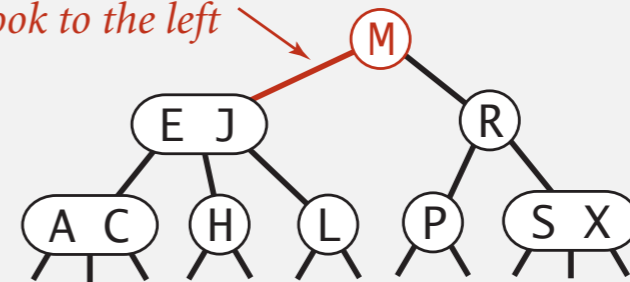
H *is less than* M *so look to the left*

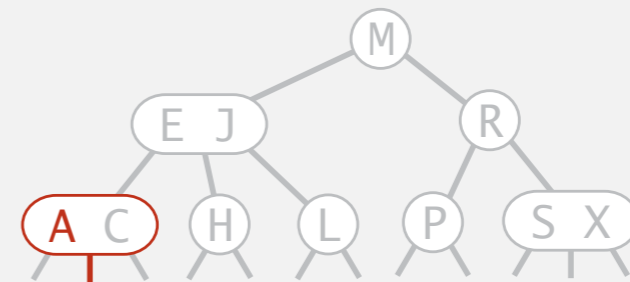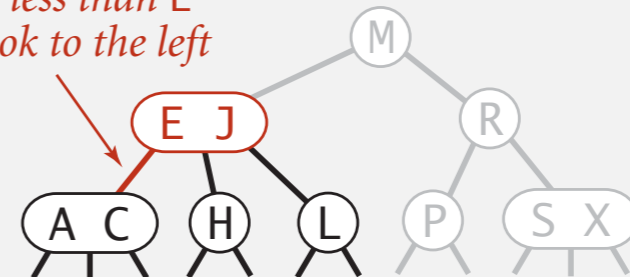H *is between* E *and* L *so look in the middle*

found H *so return value (search hit)*

unsuccessful search for B

B *is less than* M *so look to the left*
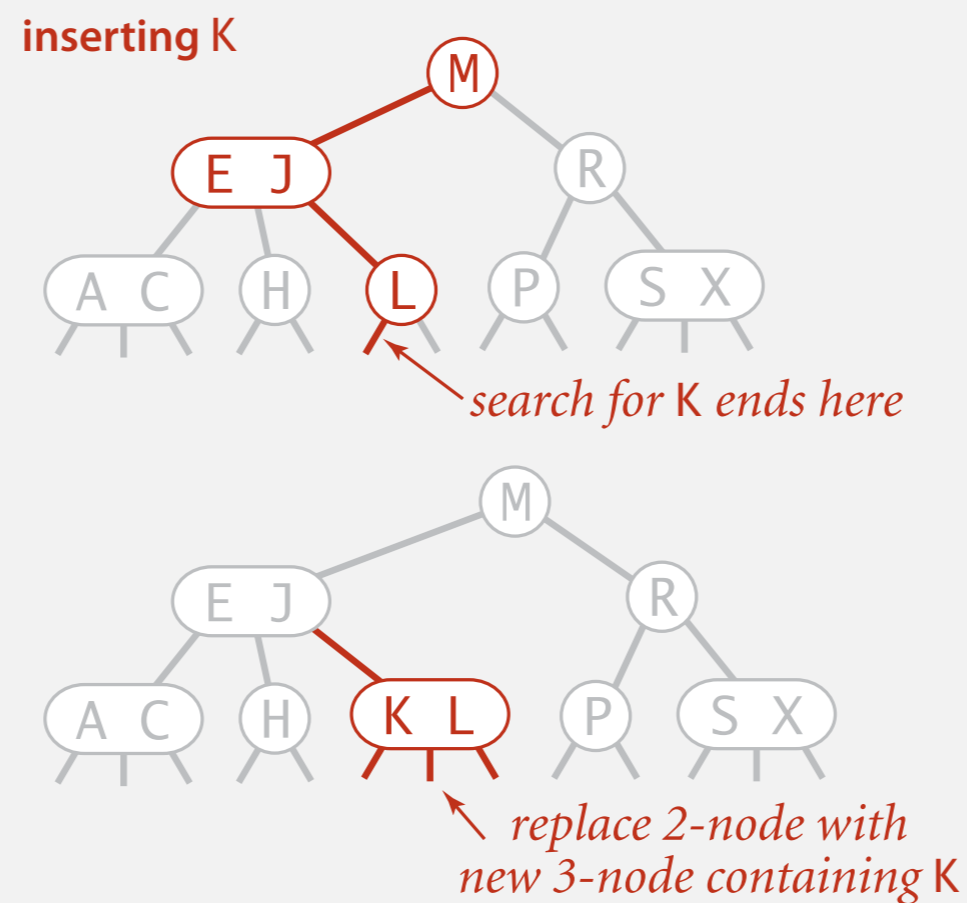
B *is less than* E *so look to the left*

B *is between* A *and* C *so look in the middle*
link is null so B *is not in the tree (search miss)*

Insertion in a 2-3 tree

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

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

inserting K

search for K ends here

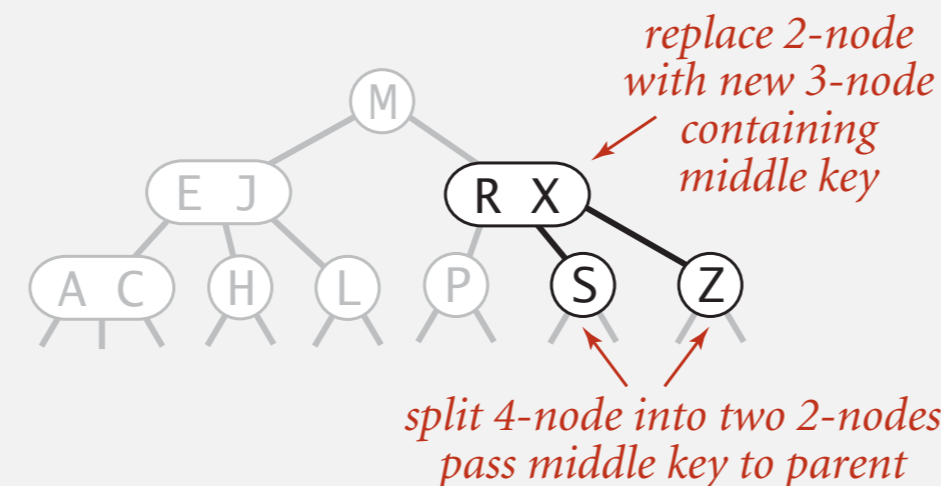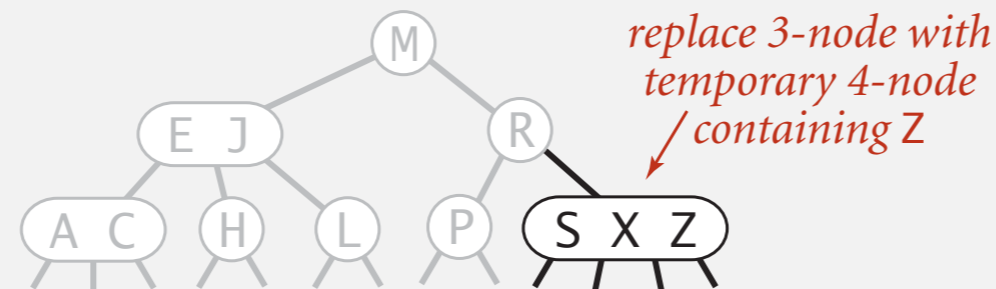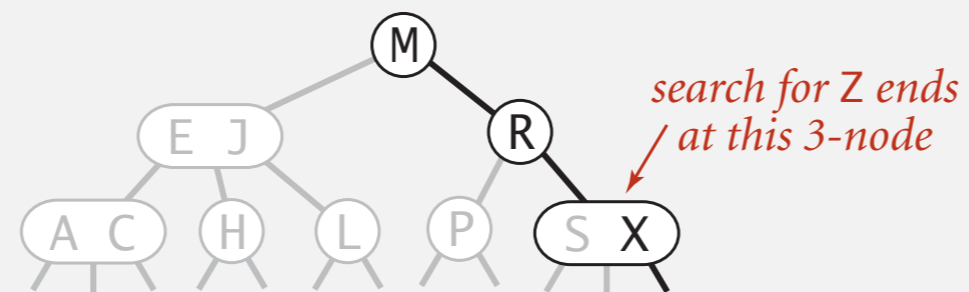*replace 2-node with new 3-node containing K*

Insertion in a 2-3 tree

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

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

inserting Z

search for Z ends at this 3-node

replace 3-node with temporary 4-node containing Z

replace 2-node with new 3-node containing middle key

split 4-node into two 2-nodes pass middle key to parent
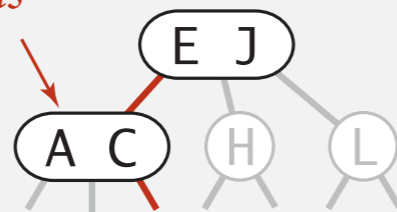
Insertion in a 2-3 tree

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

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
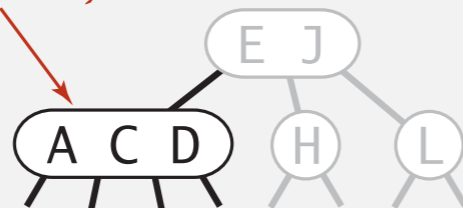- If you reach the root and it's a 4-node, split it into three 2-nodes.
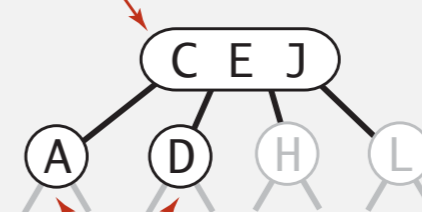
increases height by 1

inserting D

search for D ends
at this 3-node

E J

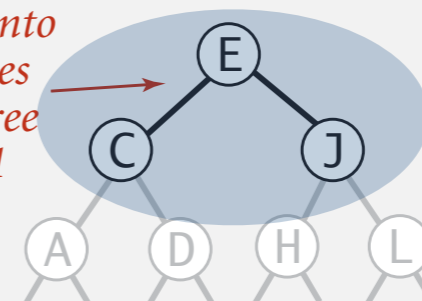A C   H   L

add new key D to 3-node
to make temporary 4-node

E J

A C D   H   L

add middle key C to 3-node
to make temporary 4-node

C E J

A   D   H   L
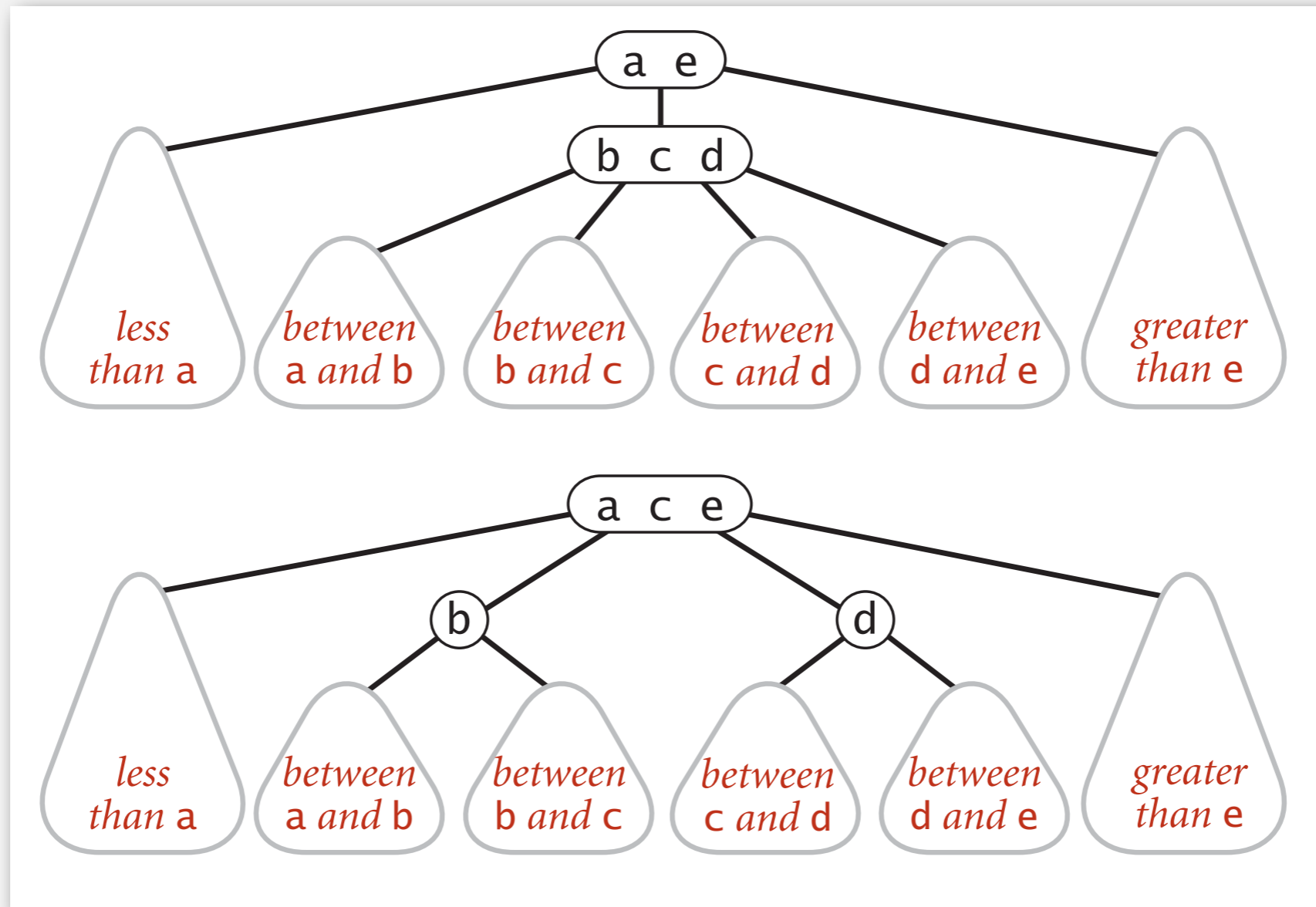
split 4-node into two 2-nodes
pass middle key to parent

split 4-node into
three 2-nodes
increasing tree
height by 1

E

C   J

A   D   H   L

Local transformations in a 2-3 tree

Splitting a 4-node is a local transformation: constant number of operations.
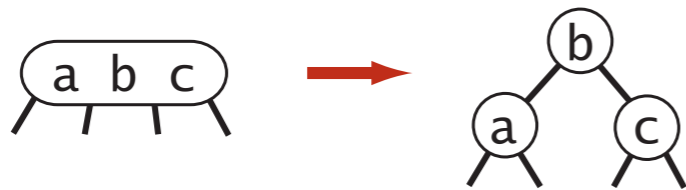
Global properties in a 2-3 tree

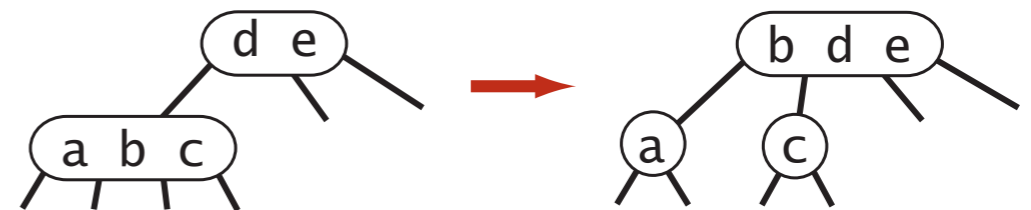Invariants.   Maintains symmetric order and perfect balance.

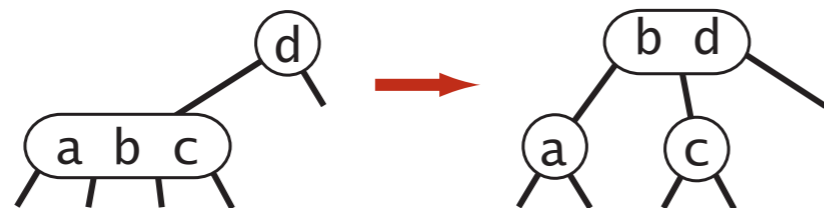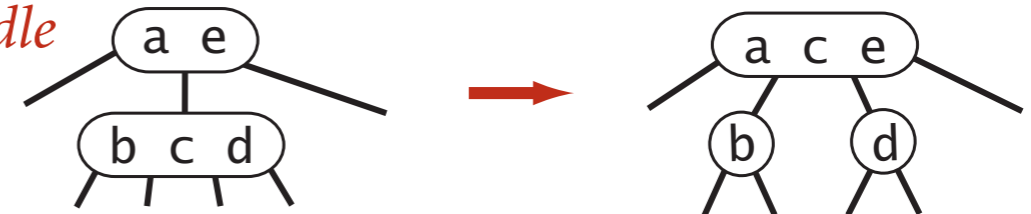Pf.   Each transformation maintains symmetric order and perfect balance.

2-3 tree: performance

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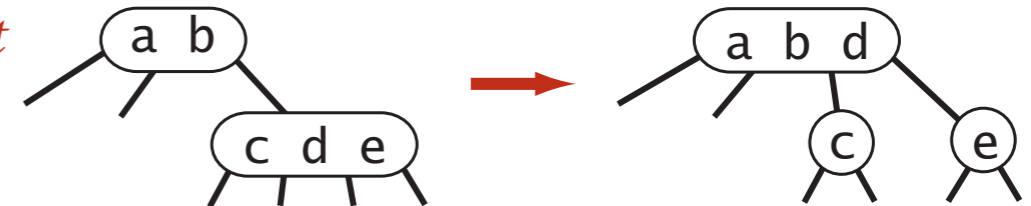

Tree height.
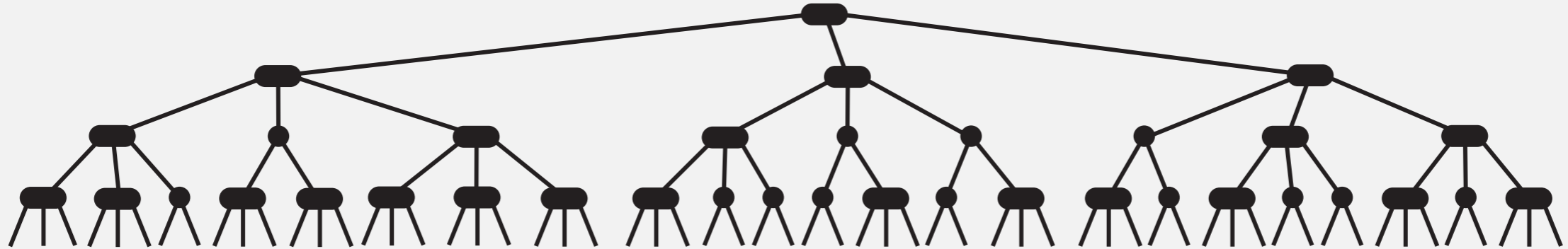- Worst case:
- Best case:

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()`
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.
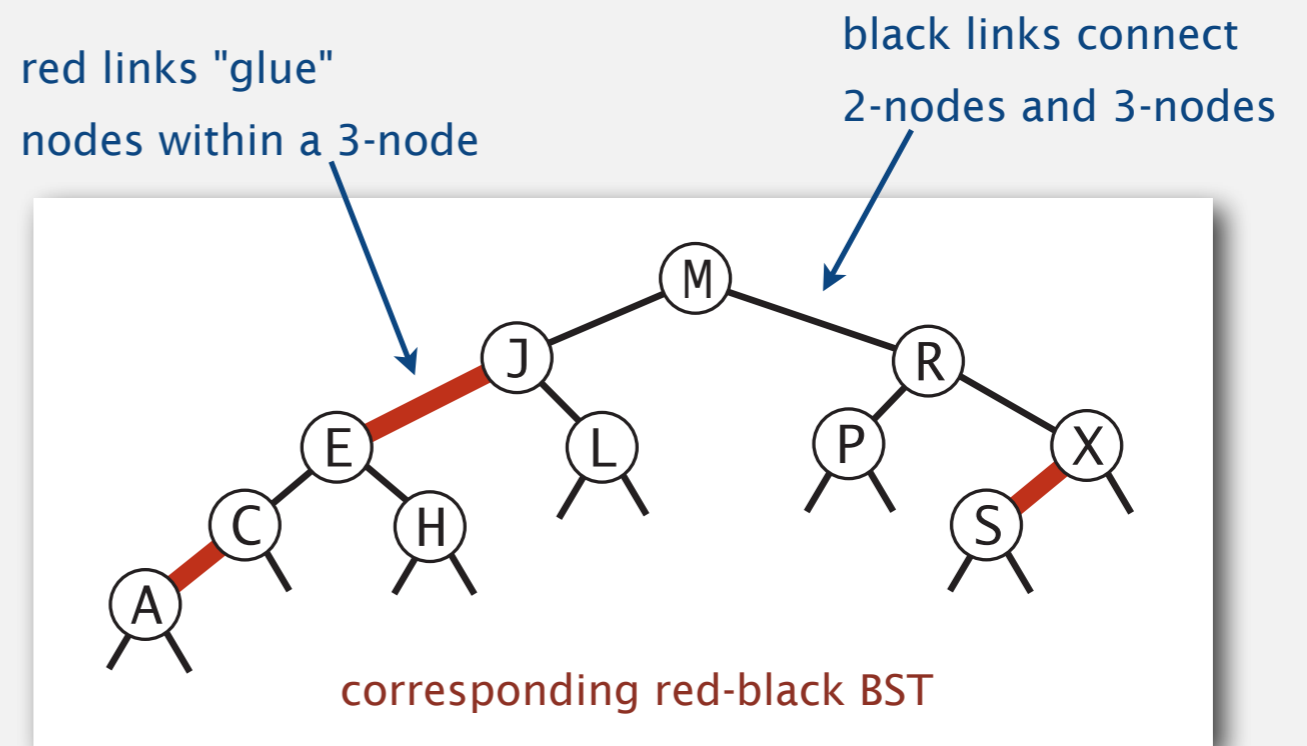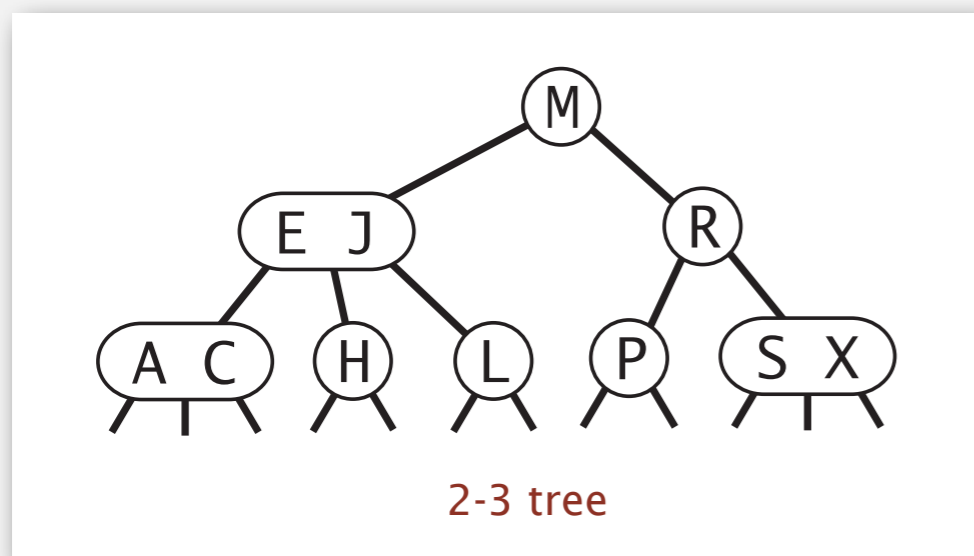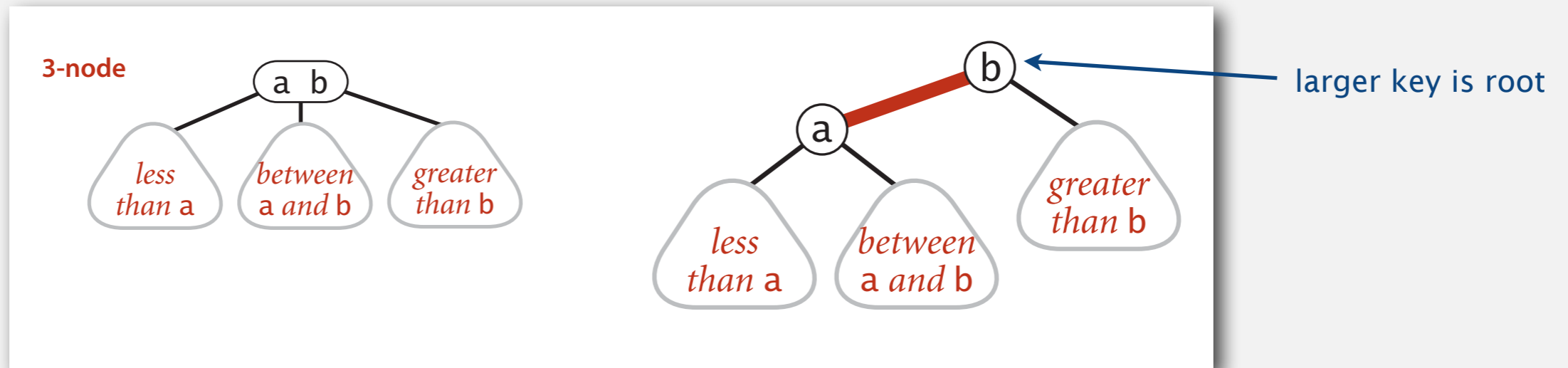- Large number of cases for splitting.

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

BALANCED SEARCH TREES

- ‣ 2-3 search trees
- ‣ **Red-black BSTs**
- ‣ B-trees
- ‣ Geometric applications of BSTs

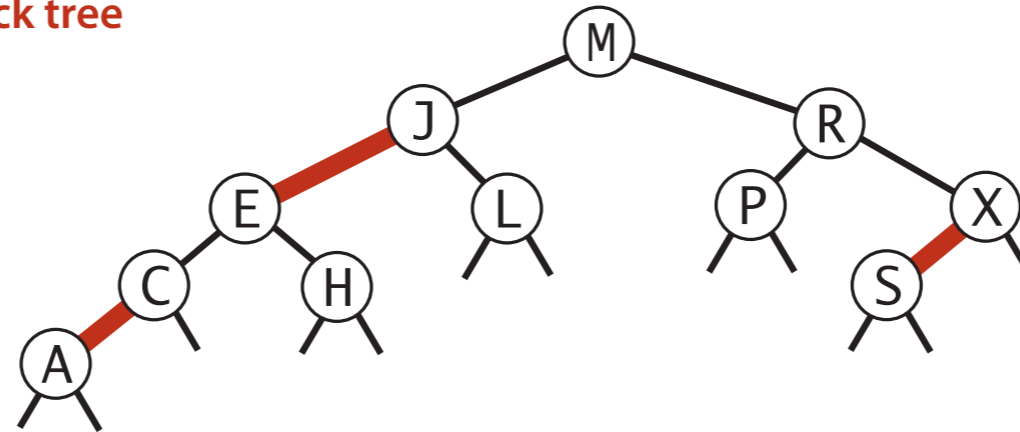# Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

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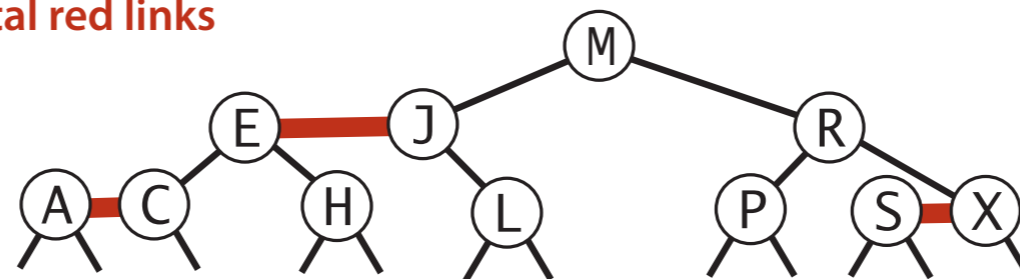2. Use "internal" left-leaning links as "glue" for 3–nodes.

larger key is root

red links "glue" nodes within a 3-node

black links connect 2-nodes and 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.
  - We will only allow one red link to simulate 2 keys in node
  - A node with two red links would be the same as having 3 keys

  "perfect black balance"

- Red links lean left (correct ordering)

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

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

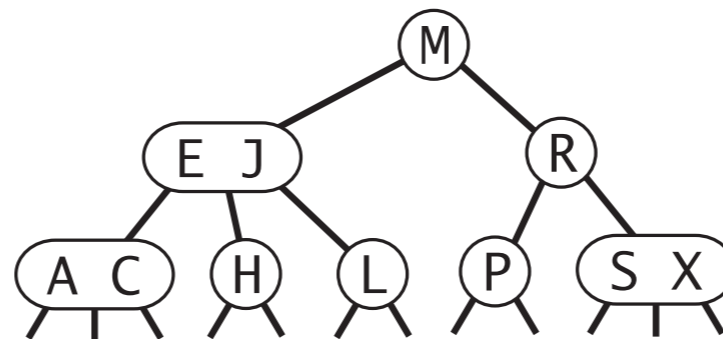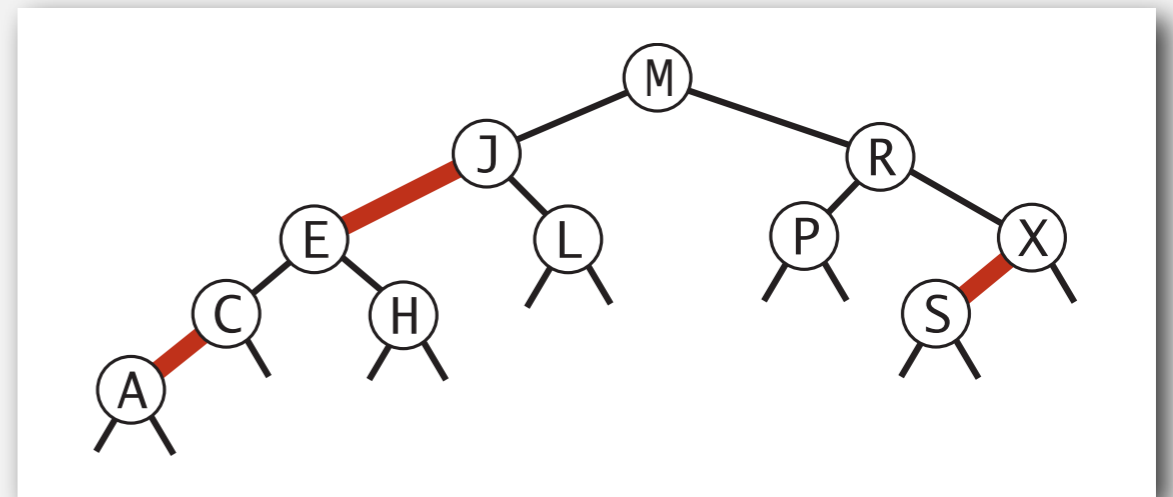Observation. Search is the same as for elementary BST (ignore color).

↑

but runs faster because of better balance

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


Remark. Most other ops (e.g., ceiling, selection, iteration) are also identical.
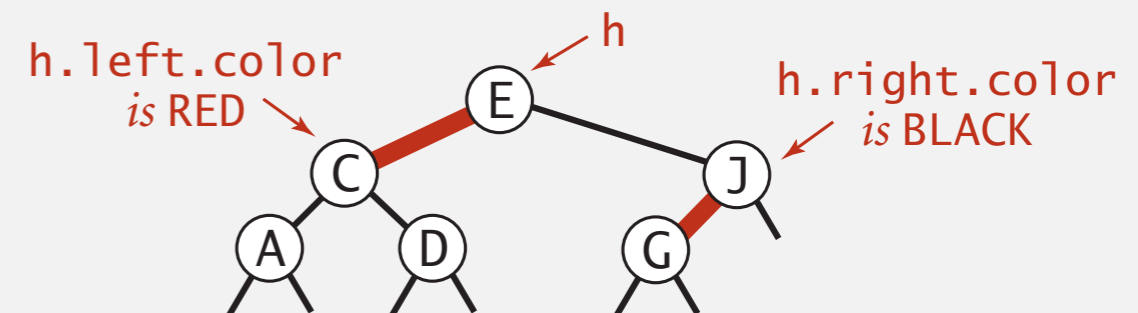
Red-black BST representation

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

```java
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;
}
```
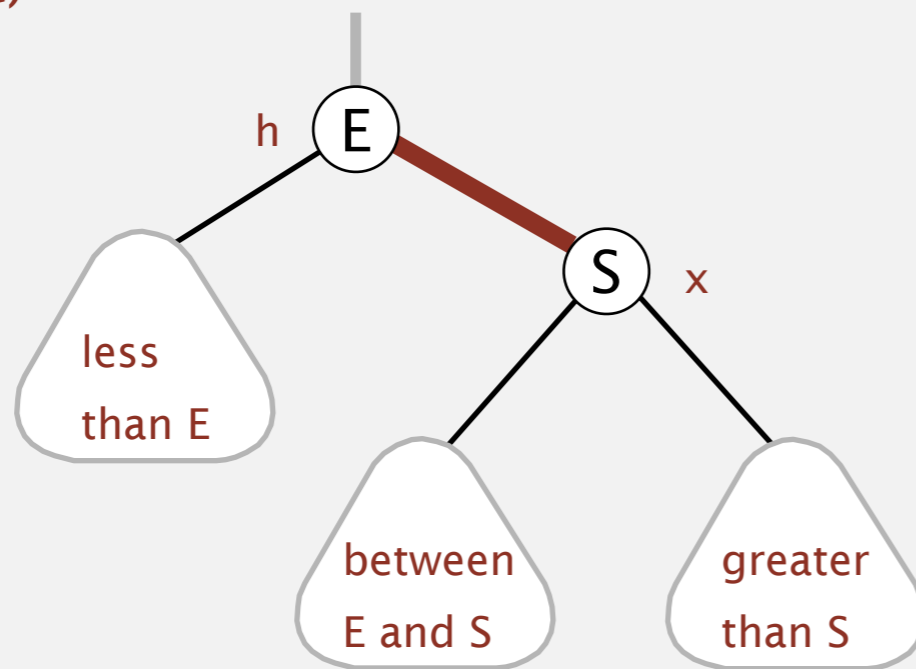
null links are black

h.left.color *is* RED

h

h.right.color *is* BLACK

Elementary red-black BST operations

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

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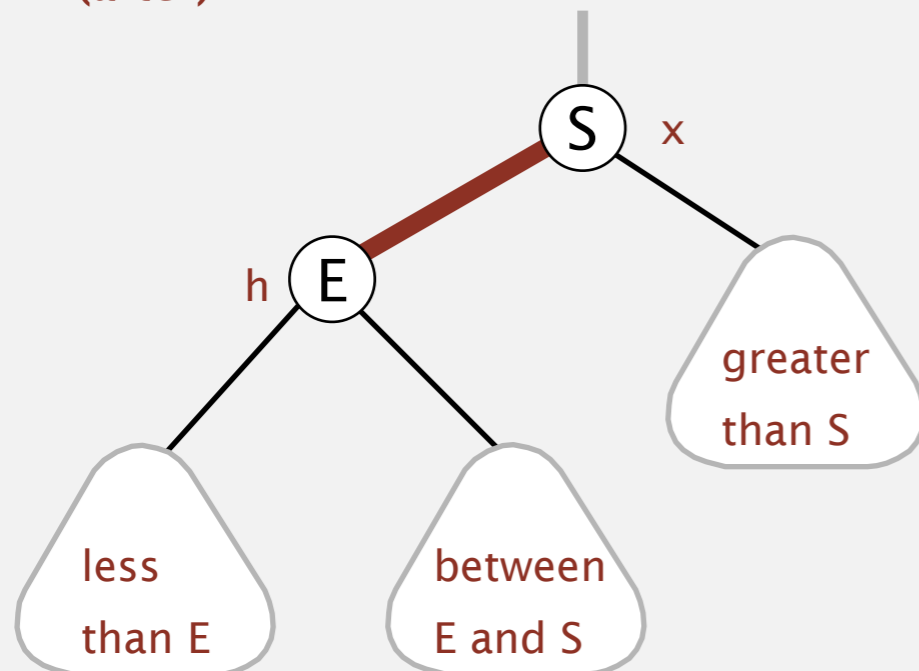 (before)
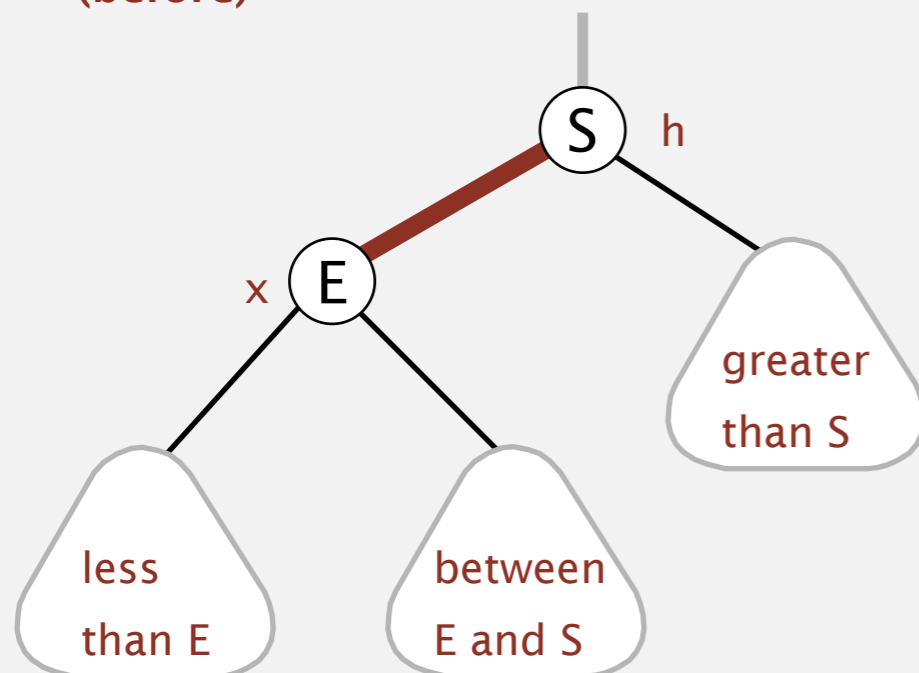
```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

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 (after)

```
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
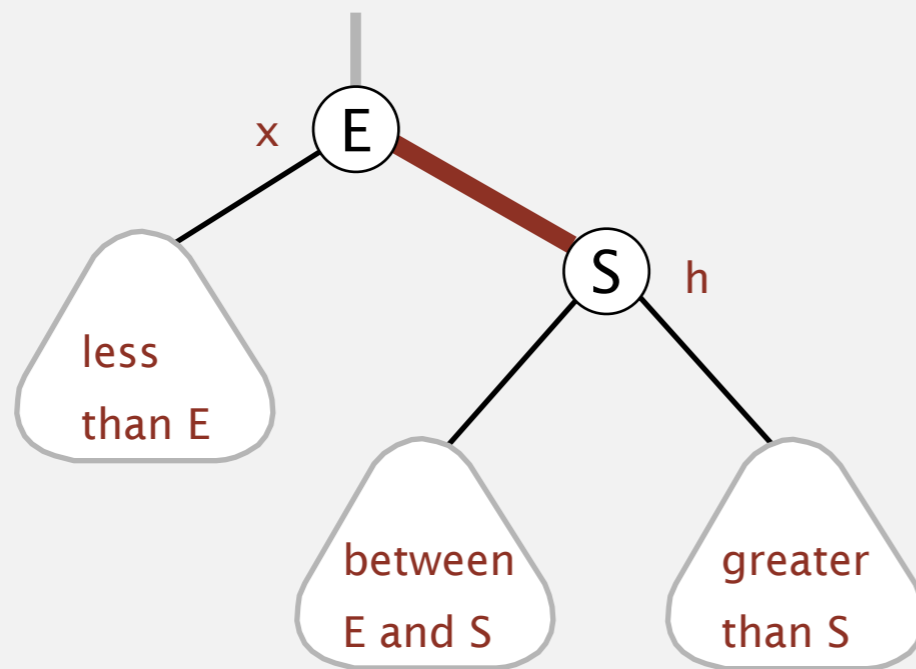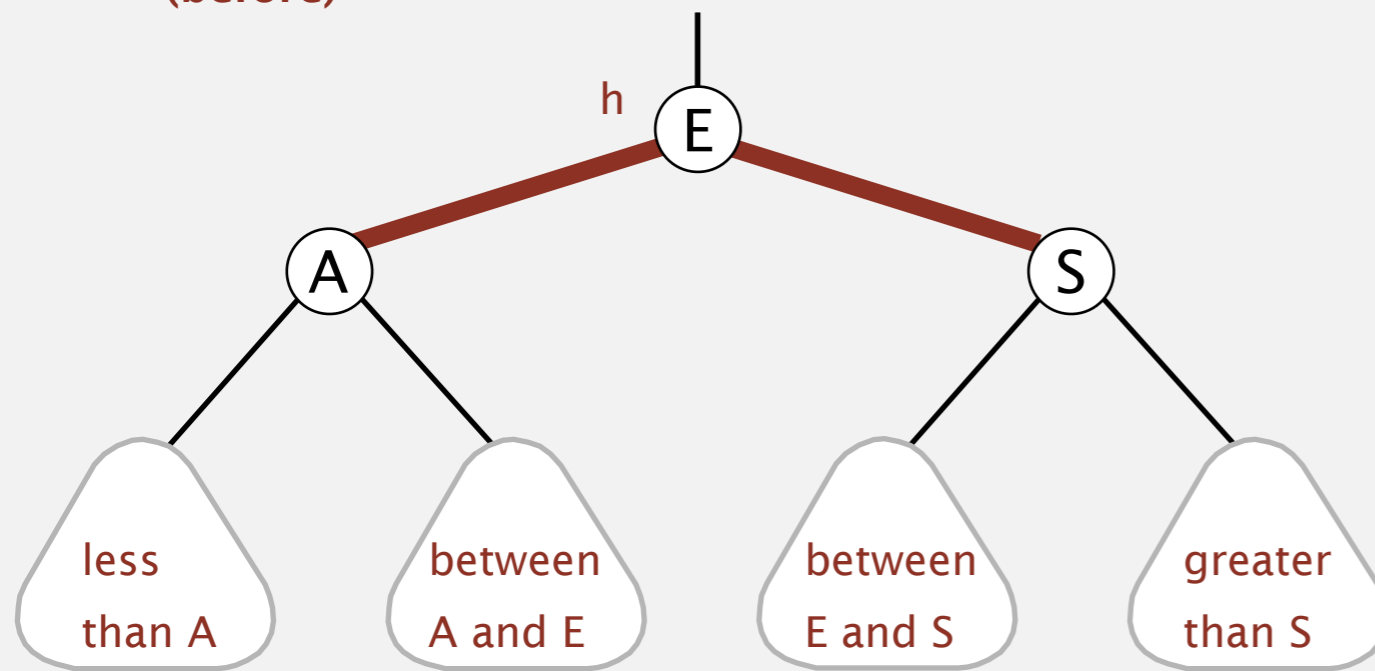(before)

```
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
 (after)


```
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
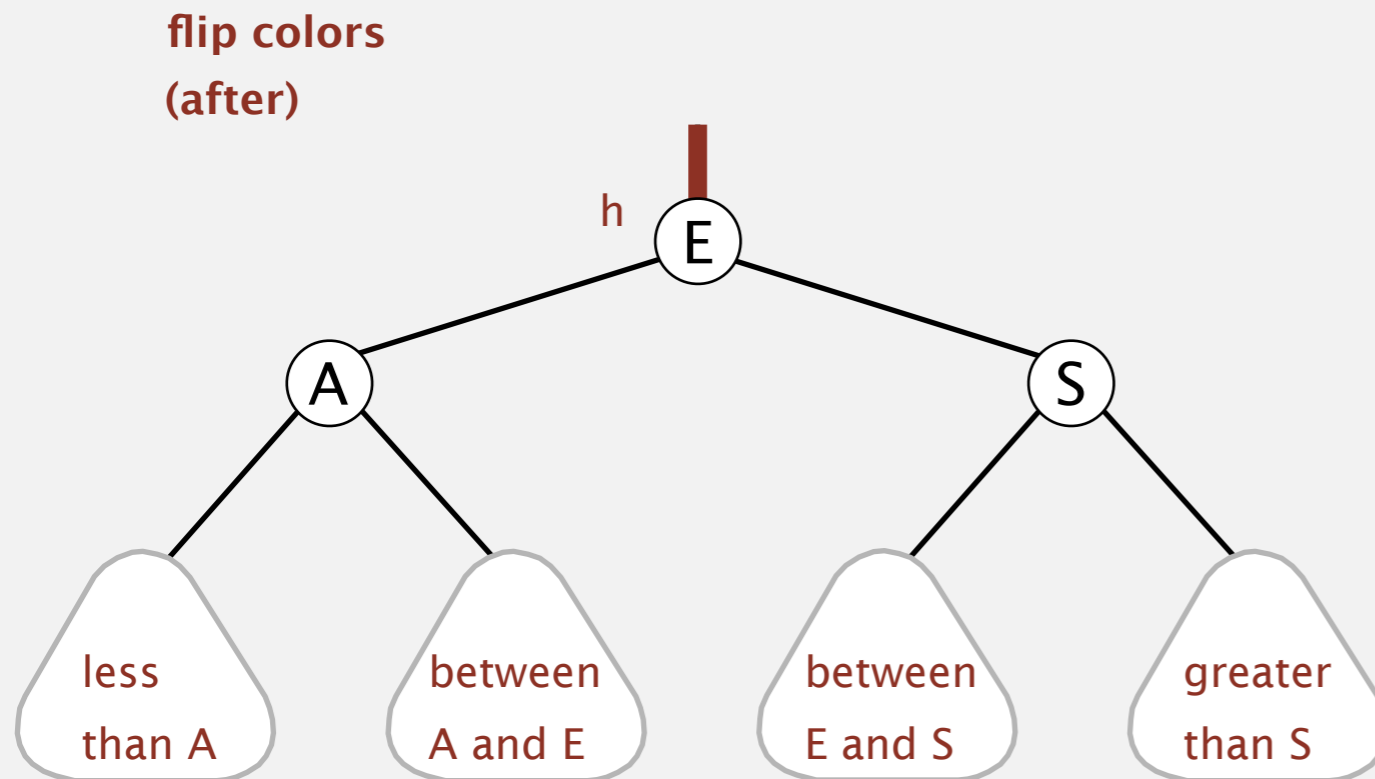
Color flip. Recolor to split a (temporary) 4-node.
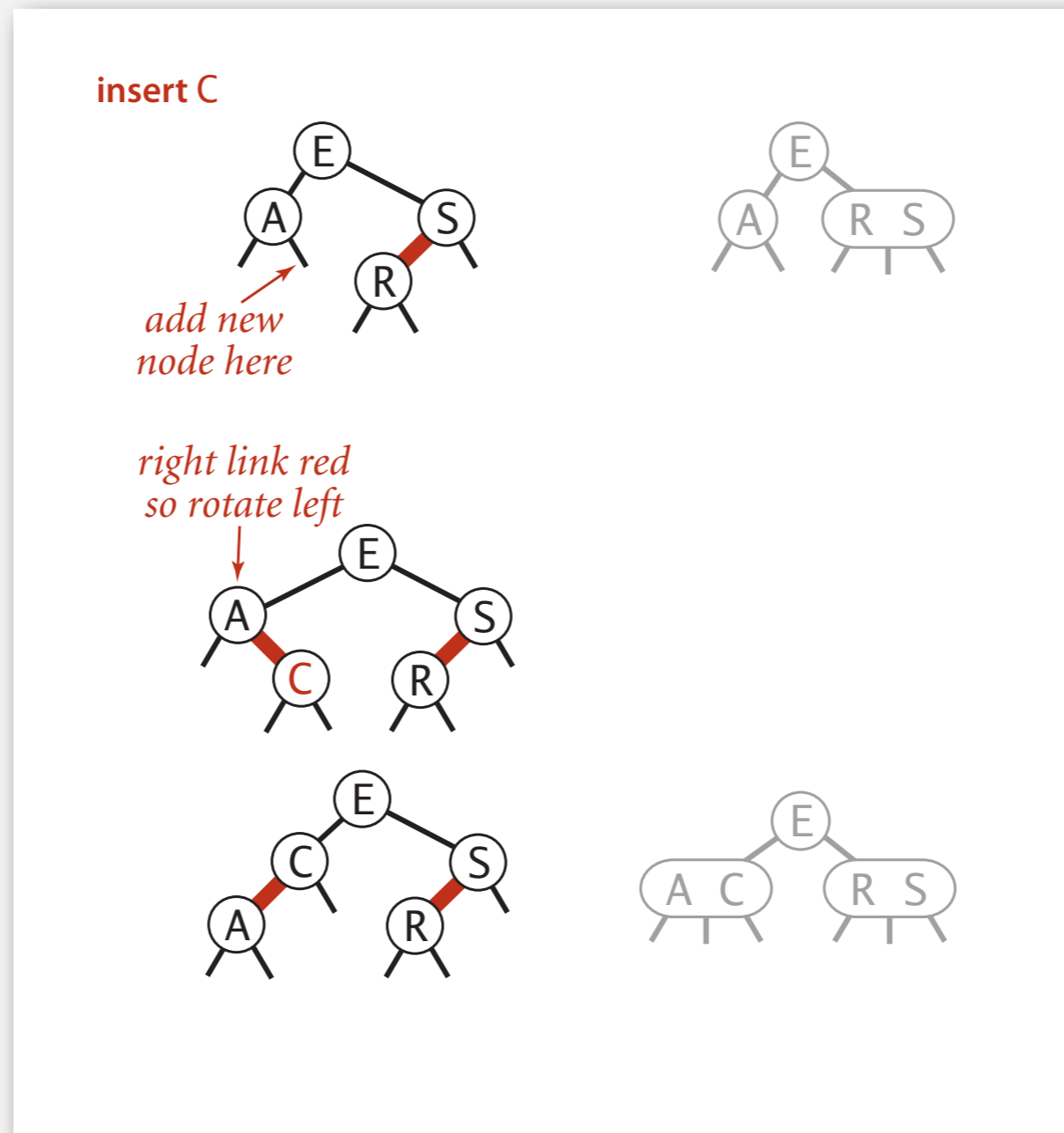
**flip colors
(before)**


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

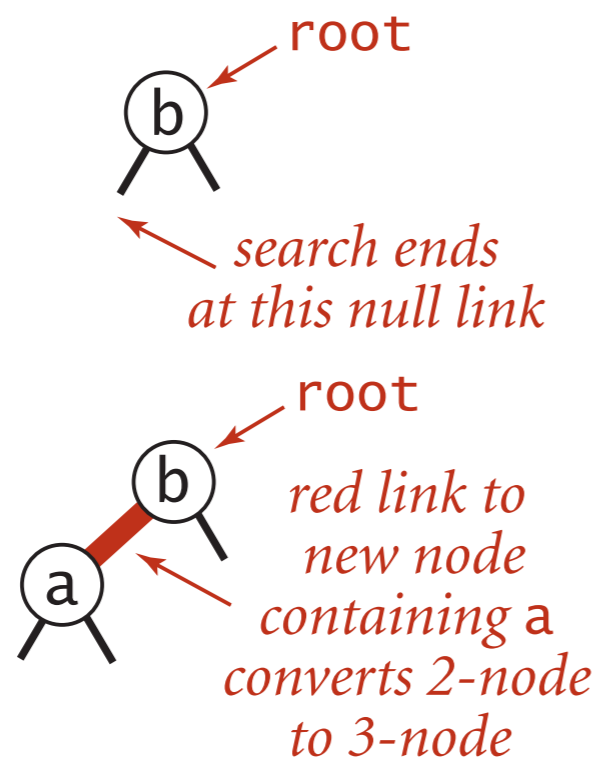Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

**flip colors
(after)**

h   E

A   S

less   between   between   greater
than A   A and E   E and S   than S

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

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree: overview

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

Insertion in a LLRB tree

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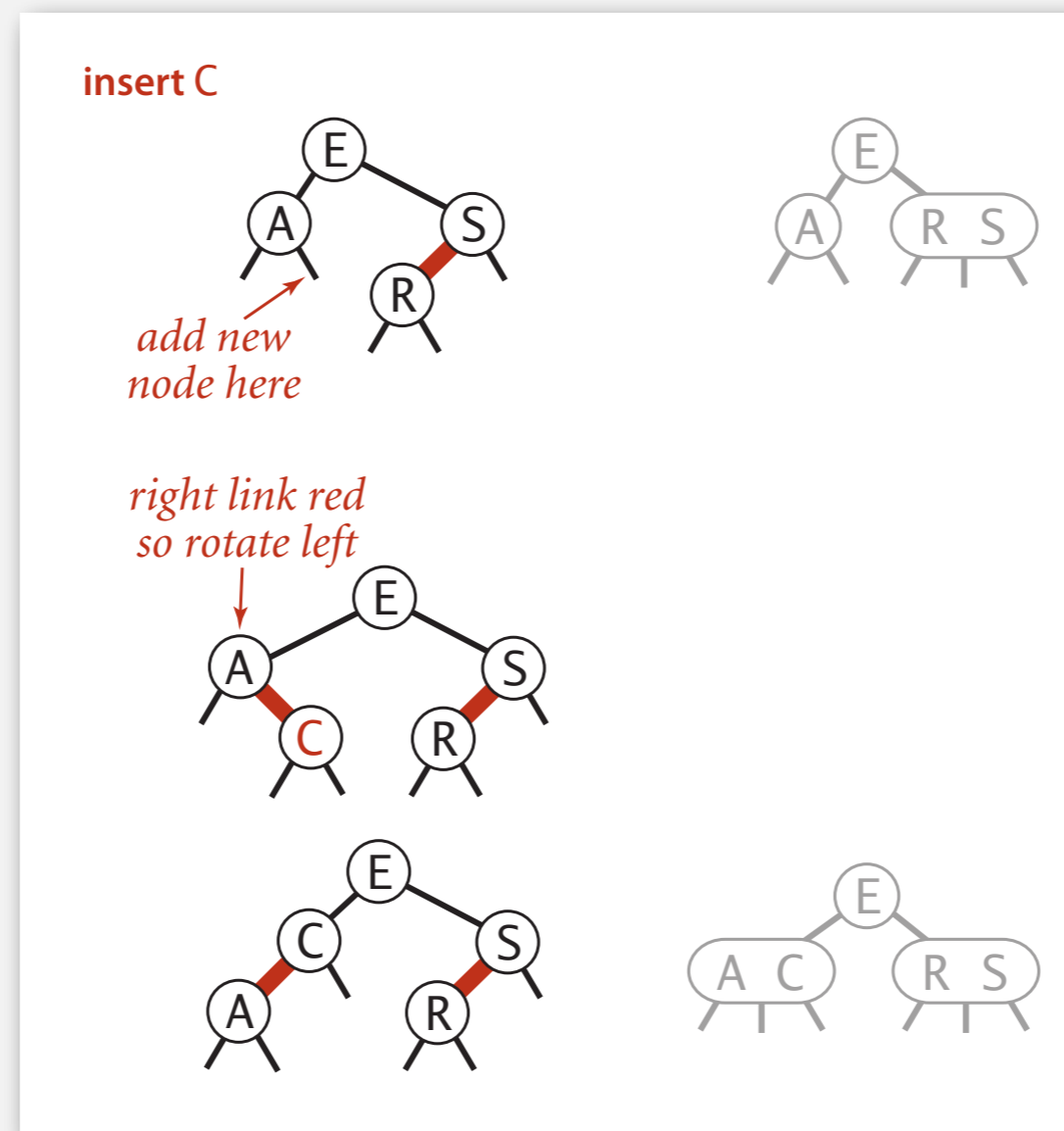

left

root

search ends at this null link

root

red link to new node containing a *converts 2-node to 3-node*

right

root

search ends at this null link

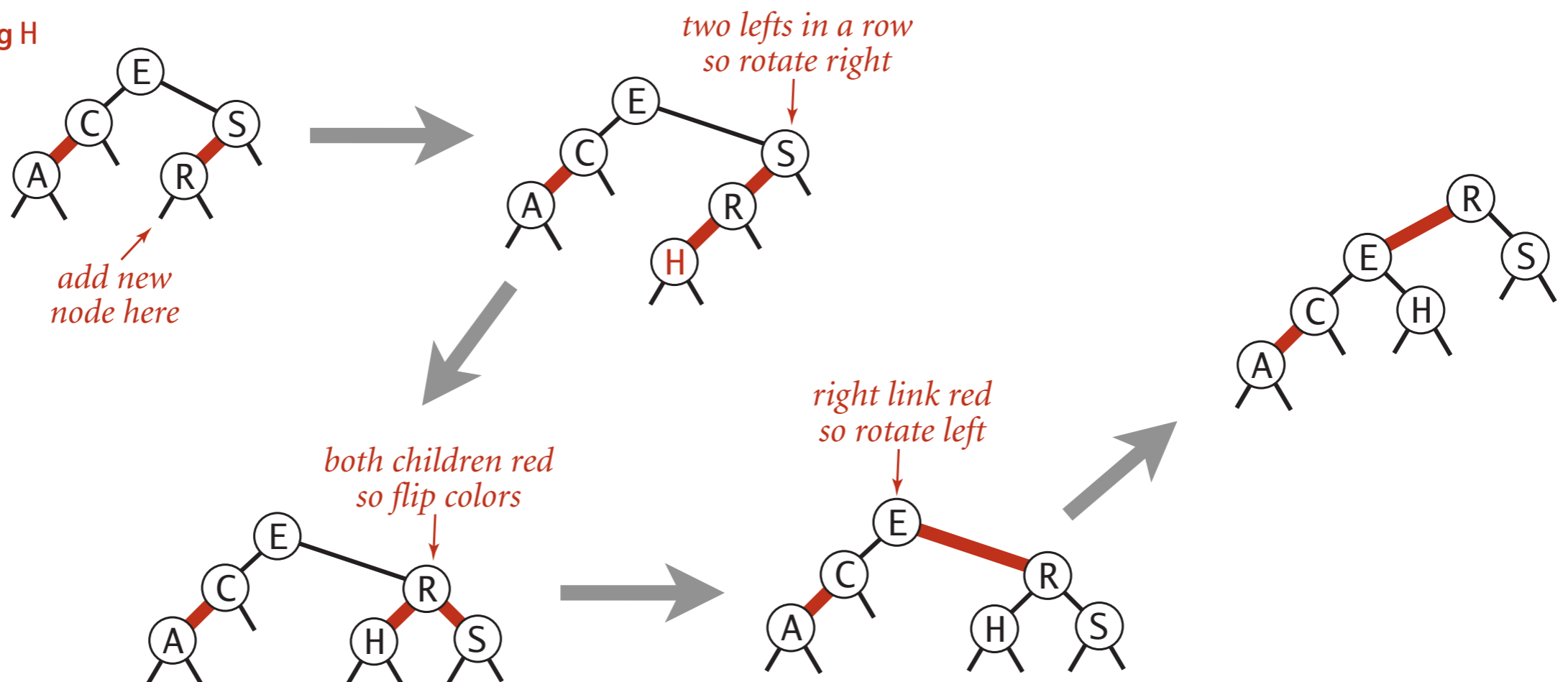*attached new node with red link*

root

rotated left to make a legal 3-node

Insertion in a LLRB tree

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

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

Insertion in a LLRB tree

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

Think of this as a split in 2-3 tree

larger

search ends at this null link

attached new node with red link

colors flipped to black

smaller

search ends at this null link

attached new node with red link

rotated right

colors flipped to black

between

search ends at this null link

attached new node with red link

rotated left

rotated right

colors flipped to black

Insertion in a LLRB tree

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

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

As with 2-3 Trees
we have to update parents,
bottom-to-top if we violate the
conditions

inserting H

add new
node here

two lefts in a row
so rotate right

both children red
so flip colors

right link red
so rotate left

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

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

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

Red-black BST insertion

insert S

Red-black BST insertion

Red-black BST insertion

insert A

insert A

two left reds in a row

(rotate S right)

Red-black BST insertion

both children red
(flip colors)

Red-black BST insertion

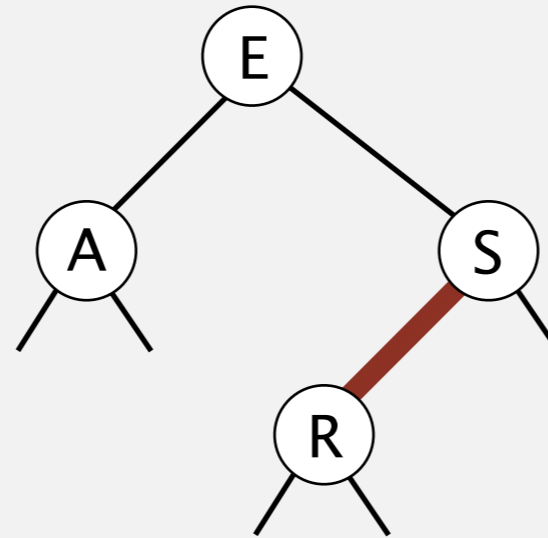both children red
(flip colors)

red-black BST
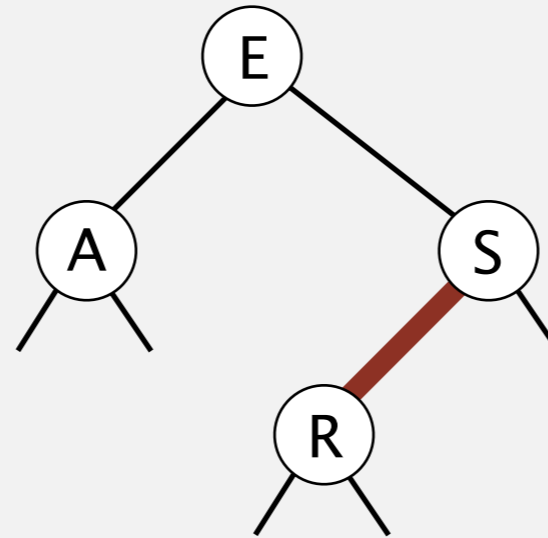
Red-black BST insertion

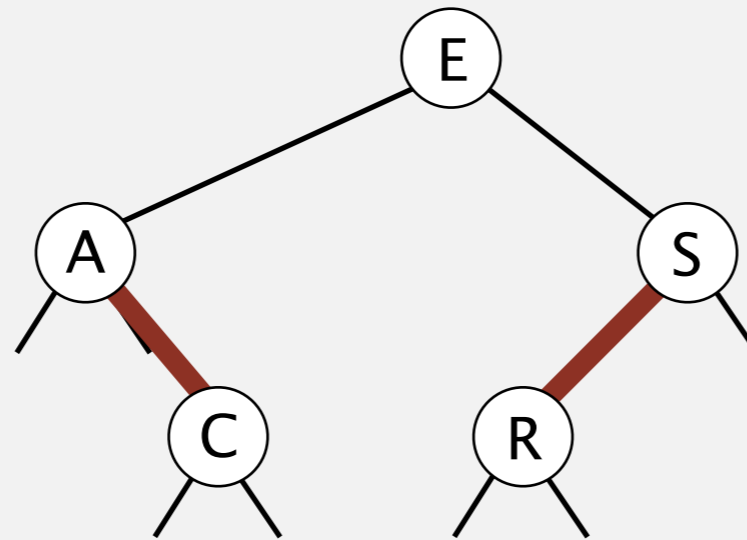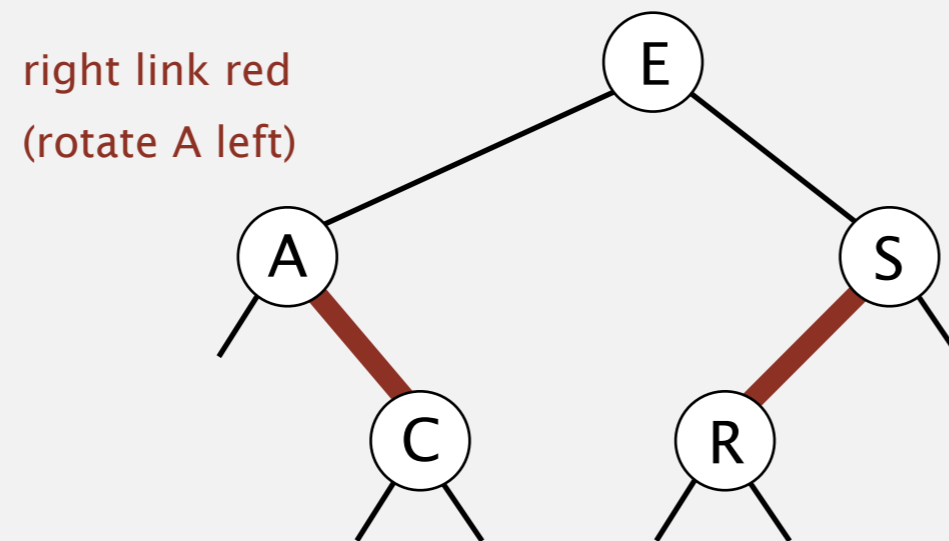**red–black BST**

Red-black BST insertion

insert R

Red-black BST insertion

red–black BST

Red-black BST insertion

Red-black BST insertion

insert C

Red-black BST insertion

right link red
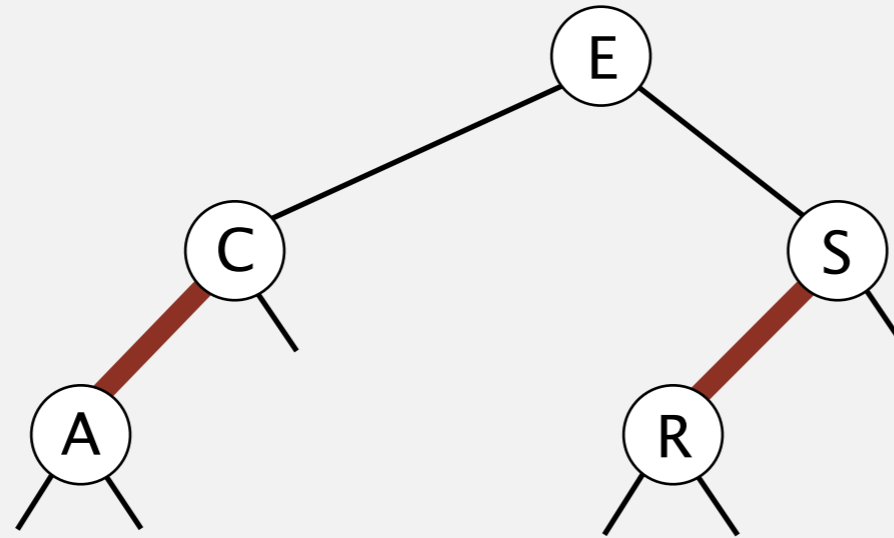(rotate A left)
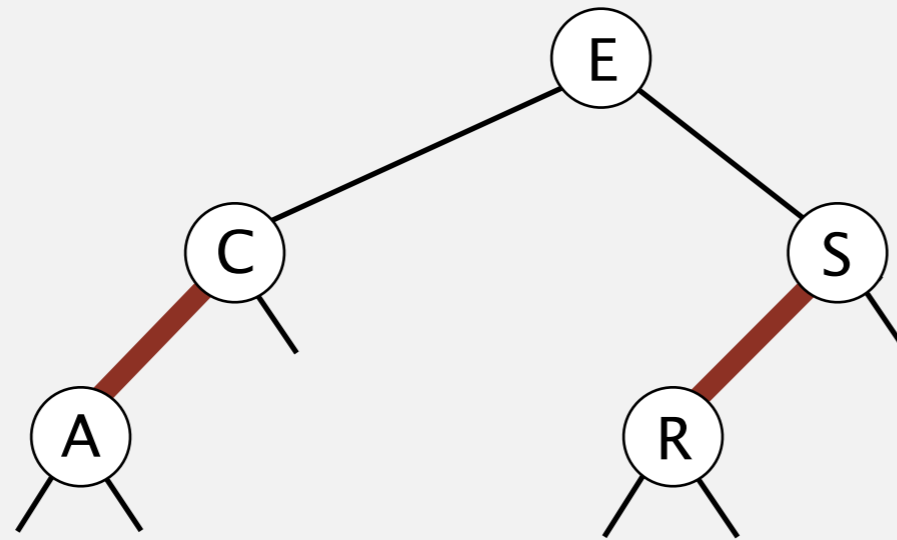
Red-black BST insertion
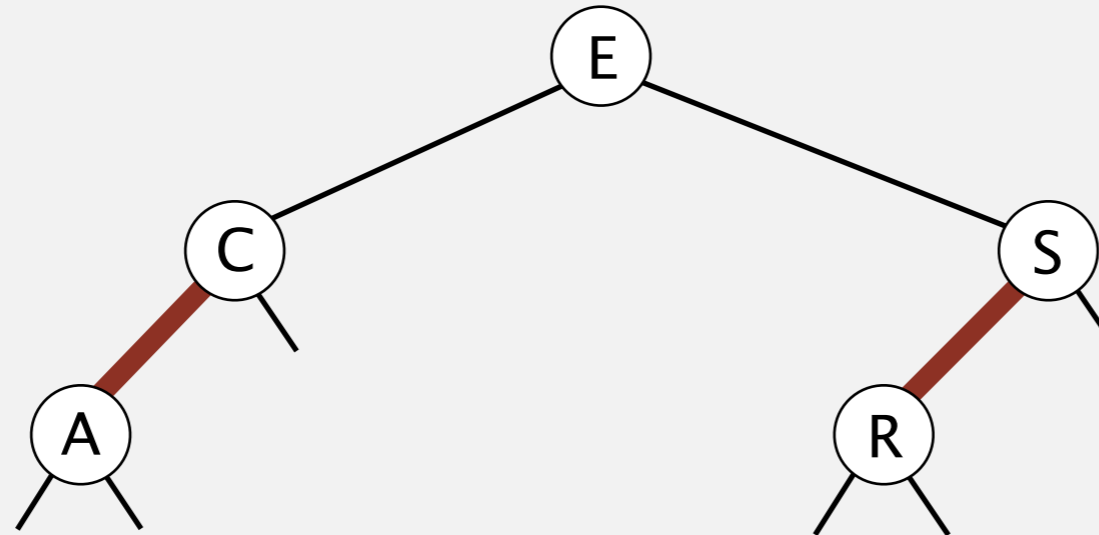
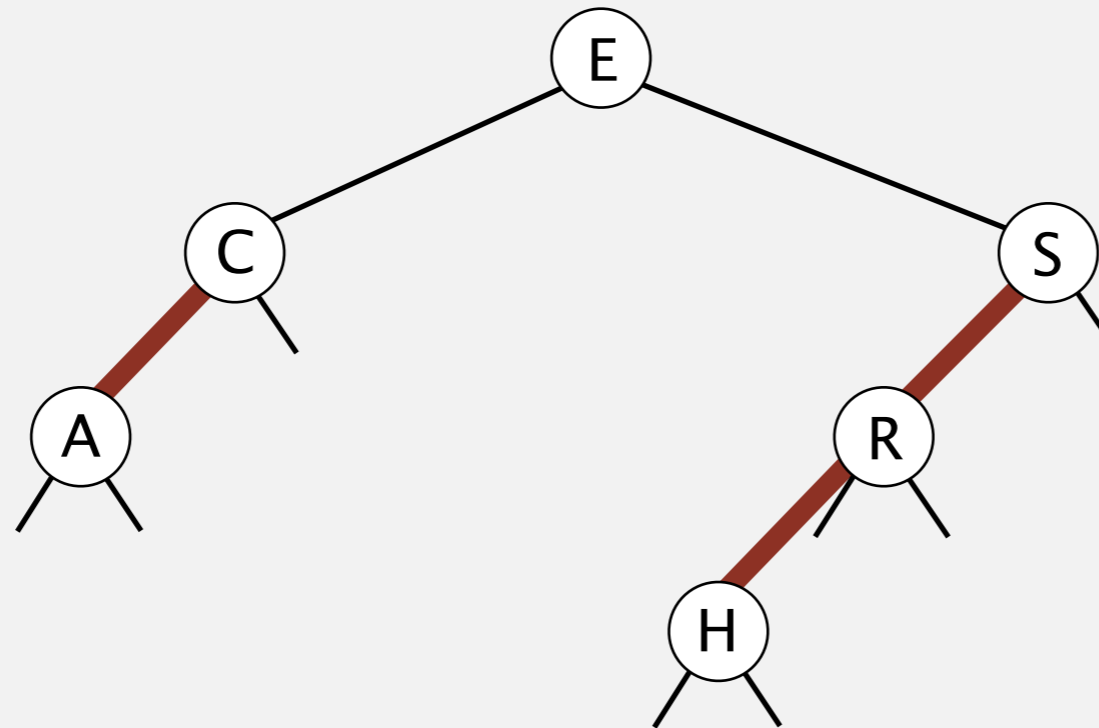**red-black BST**

Red-black BST insertion
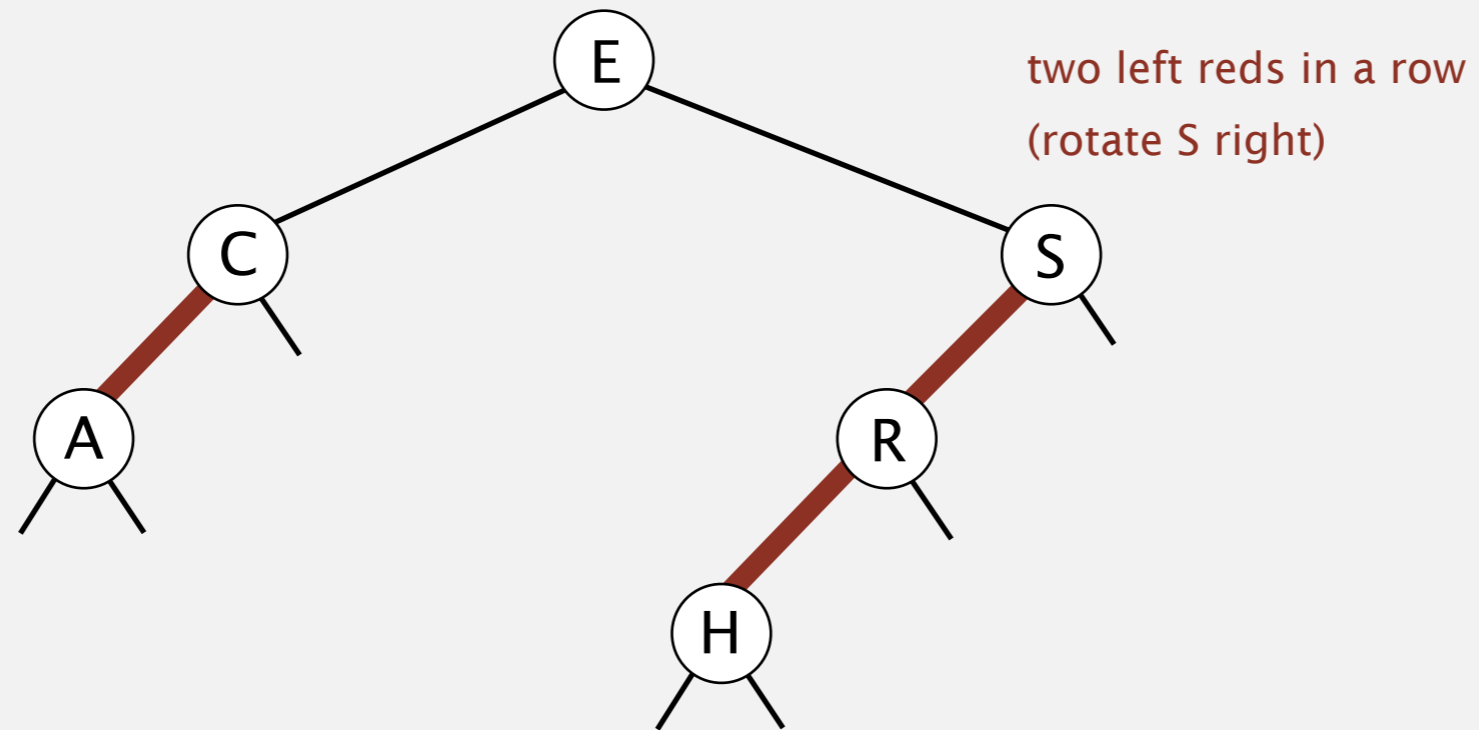
red-black BST

Red-black BST insertion

red-black BST

Red-black BST insertion

insert H

Red-black BST insertion

two left reds in a row
(rotate S right)

Red-black BST insertion

both children red
(flip colors)

Red-black BST insertion

both children red
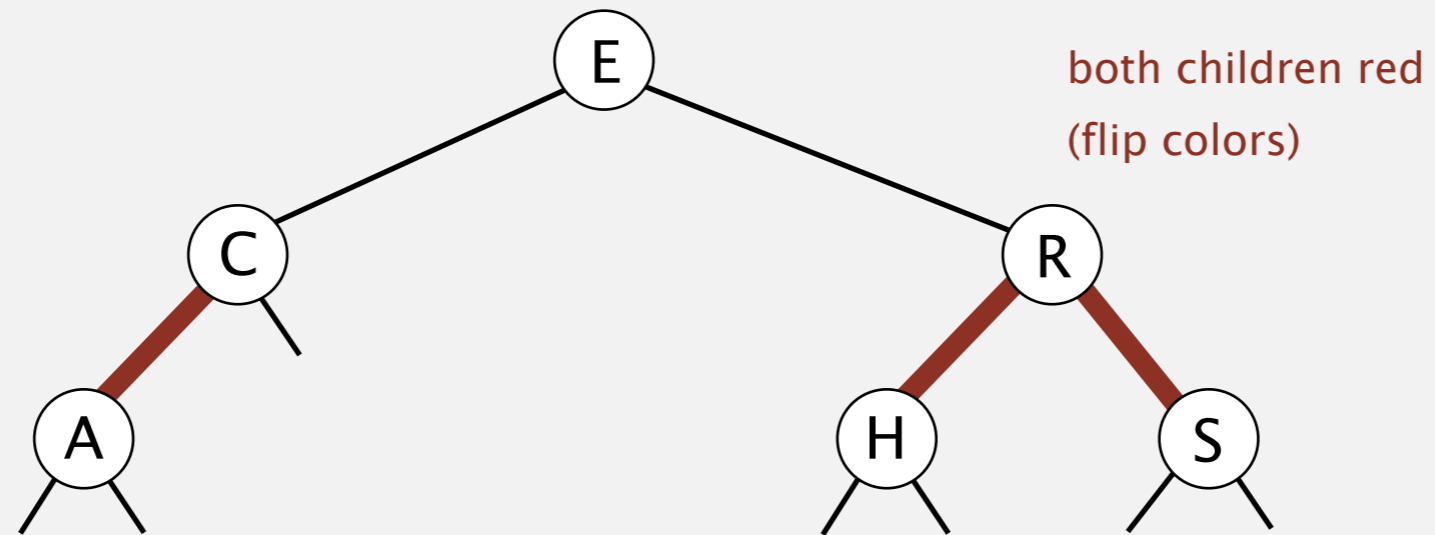(flip colors)

Red-black BST insertion

right link red
(rotate E left)

Red-black BST insertion

red-black BST

Red-black BST insertion

red-black BST

Red-black BST insertion
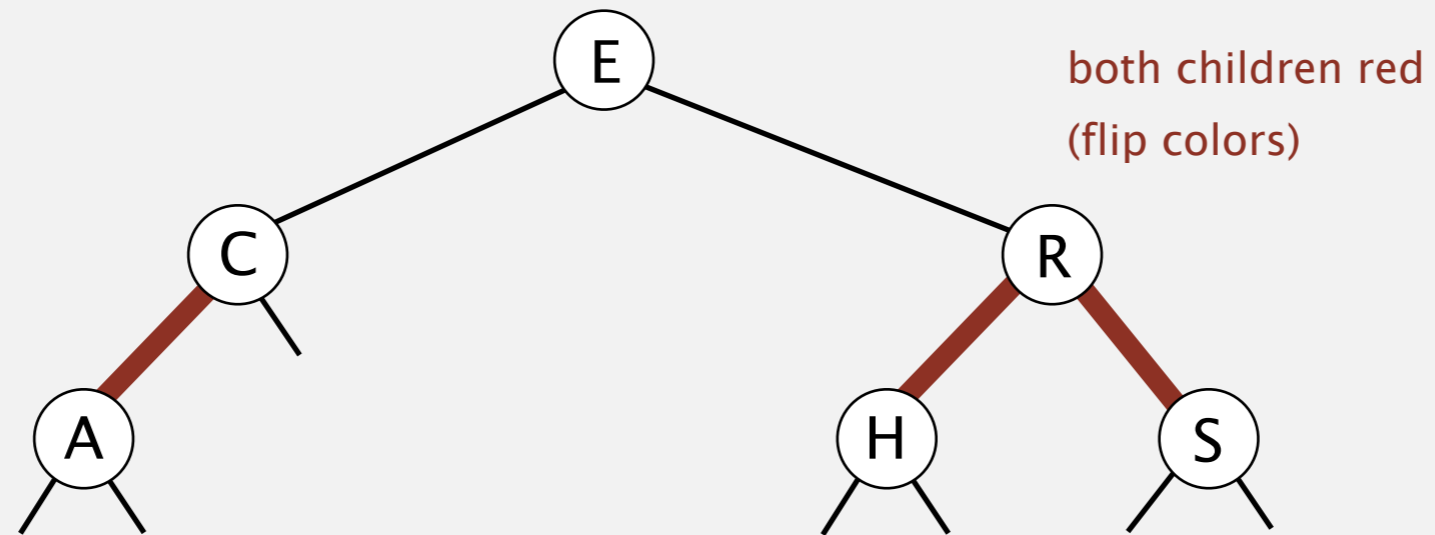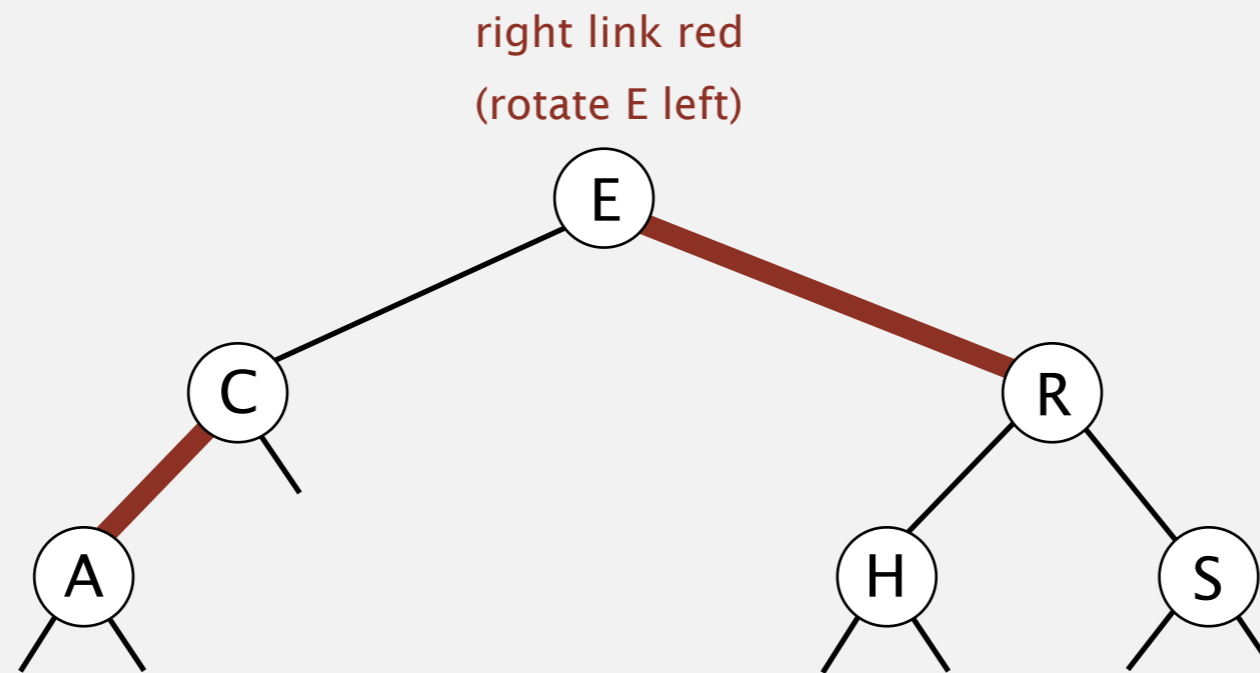
red–black BST

Red-black BST insertion

insert X

Red-black BST insertion

insert X

right link red
(rotate S left)

Red-black BST insertion

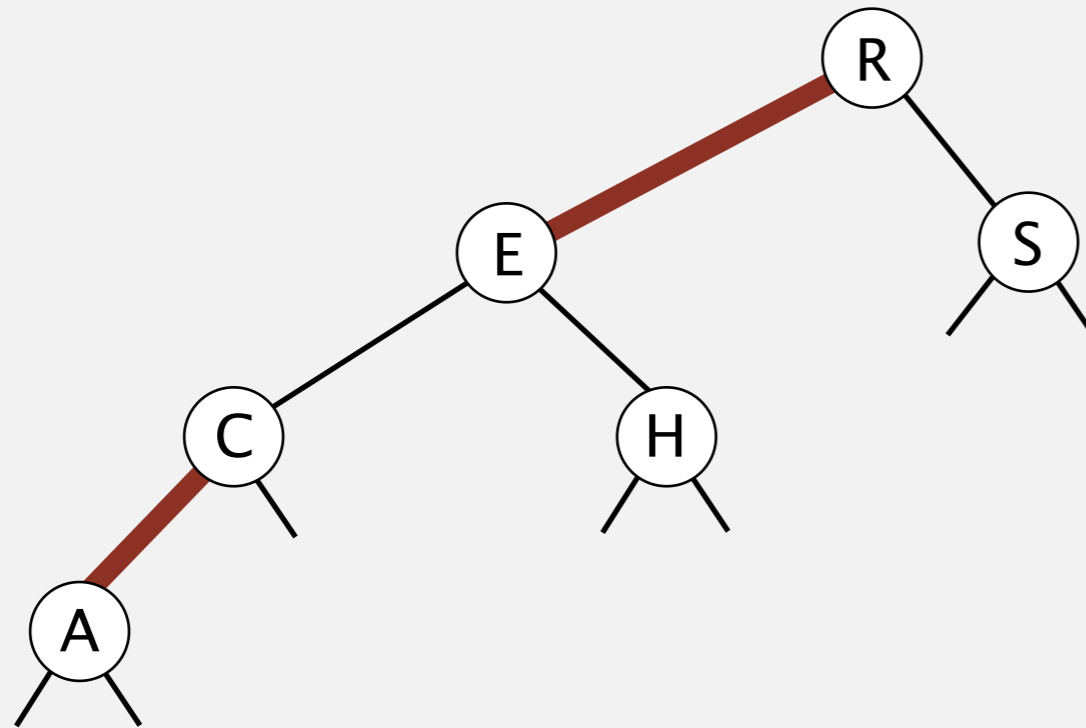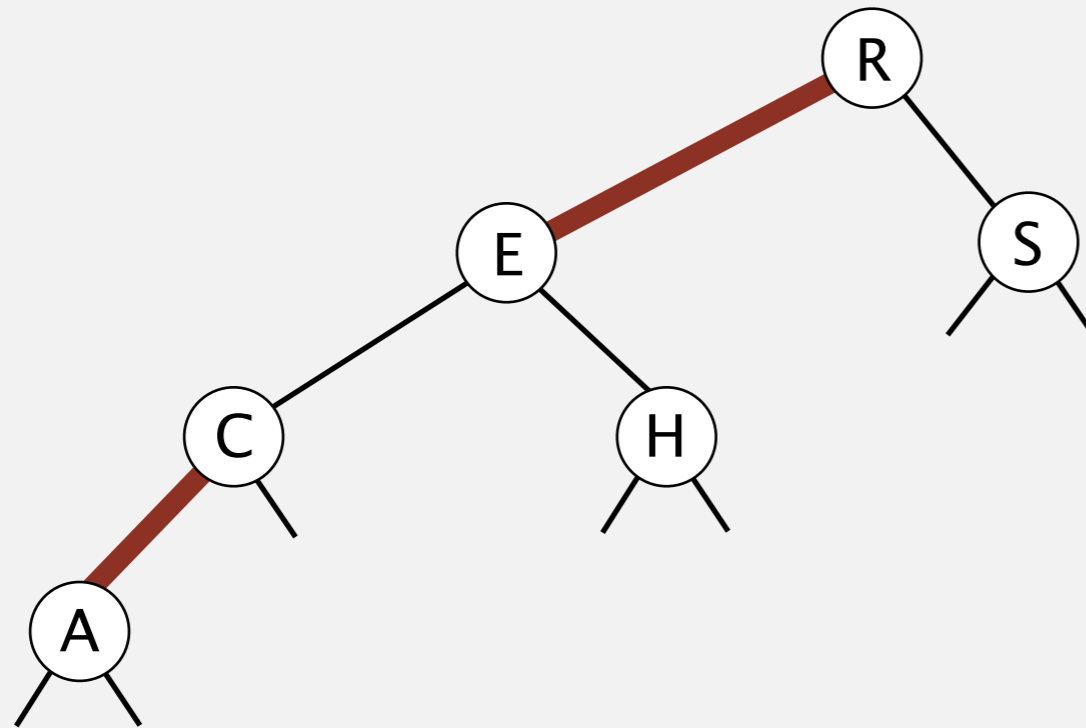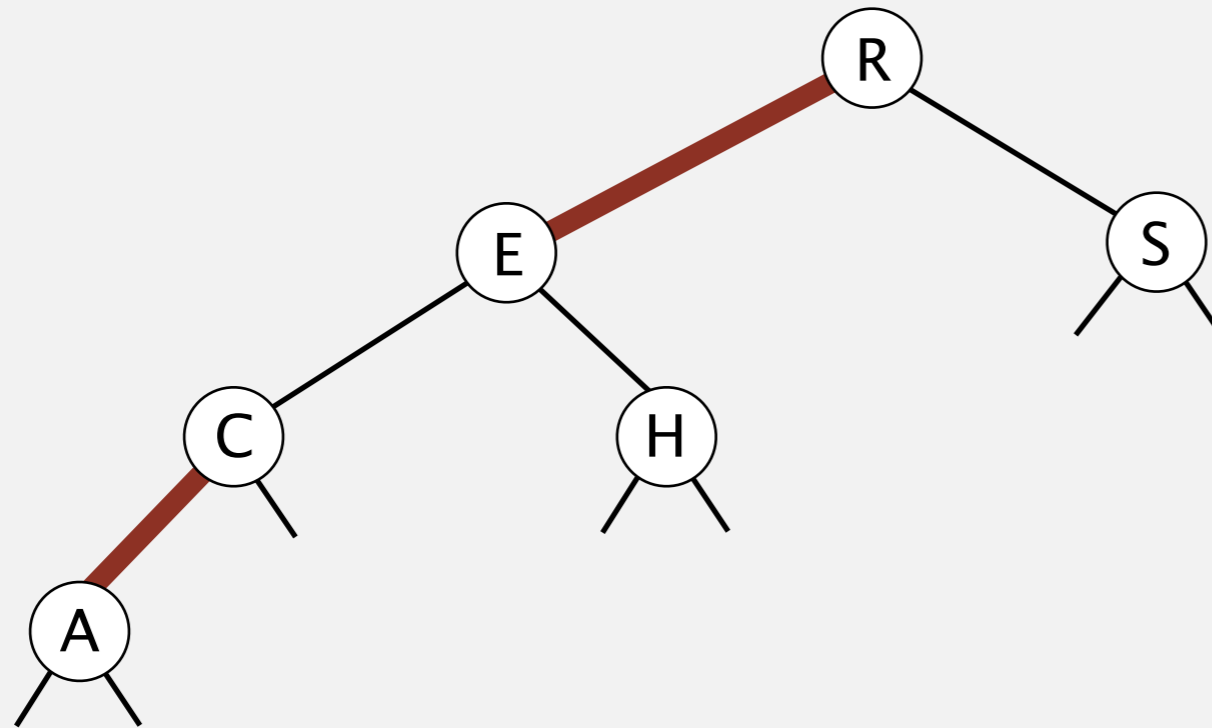**red–black BST**

Red-black BST insertion

red-black BST

Red-black BST insertion

red-black BST

Red-black BST insertion

insert M

Red-black BST insertion

insert M

right link red
(rotate H left)

Red-black BST insertion

red–black BST

insert P

Red-black BST insertion

insert P

two red children
(flip colors)

Red-black BST insertion

insert P

two red children
(flip colors)

Red-black BST insertion

right link red
(rotate E left)

Red-black BST insertion

two left reds in a row

(rotate R right)

Red-black BST insertion

two red children
(flip colors)

Red-black BST insertion

two red children
(flip colors)

Red-black BST insertion

red-black BST
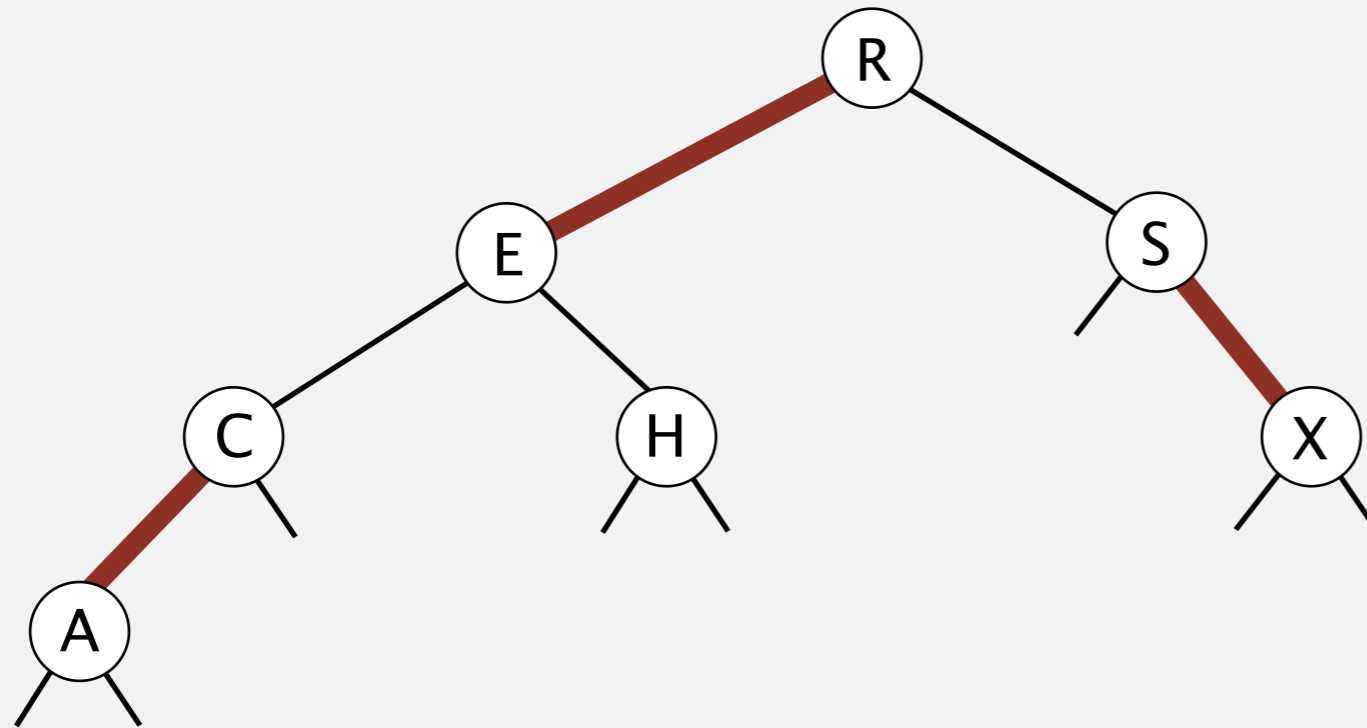
Red-black BST insertion

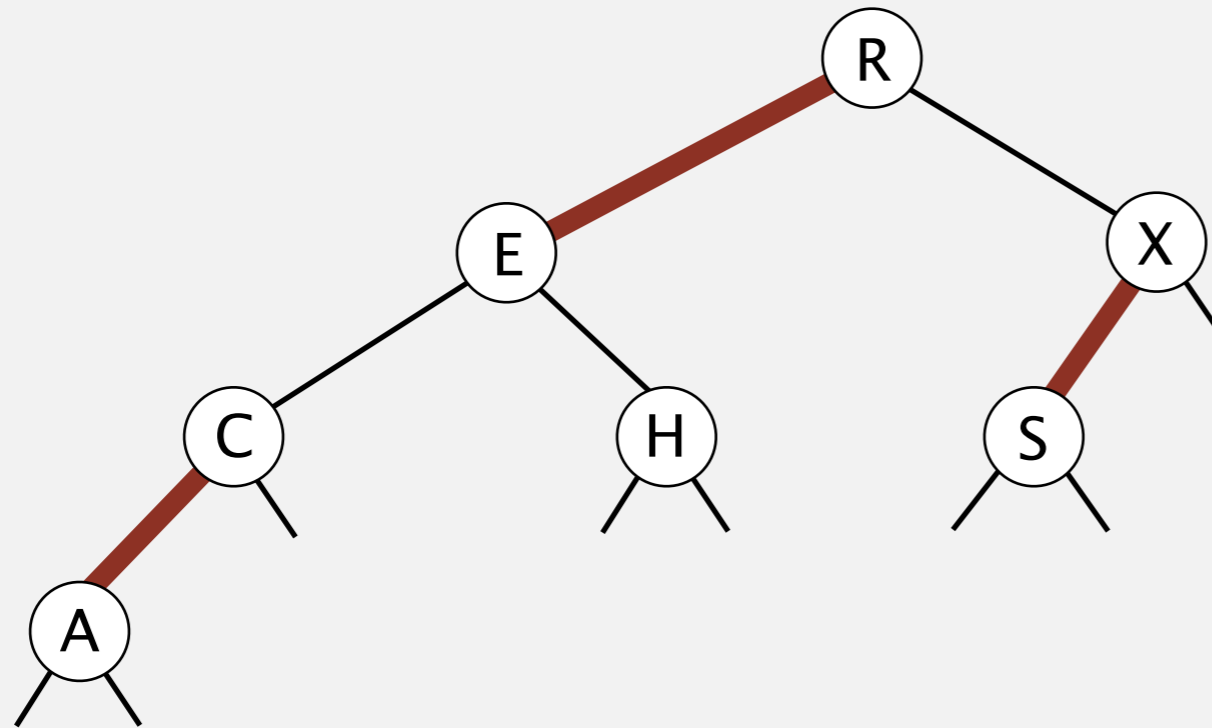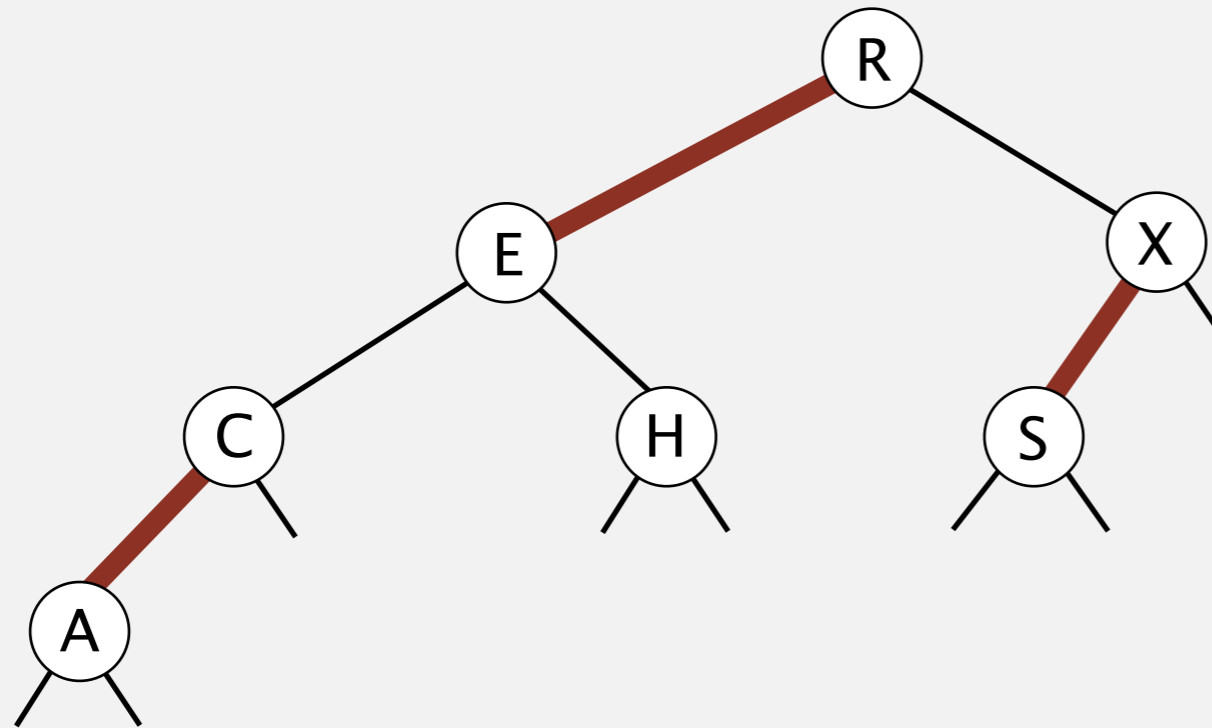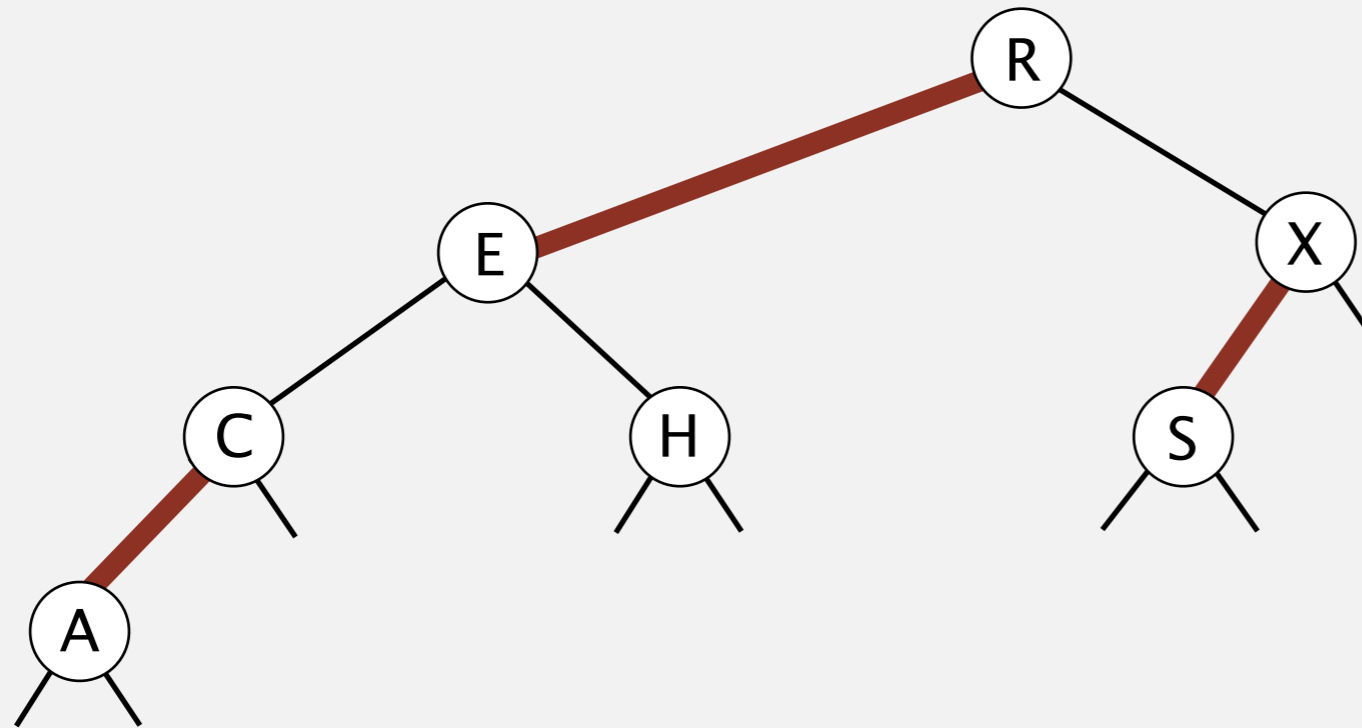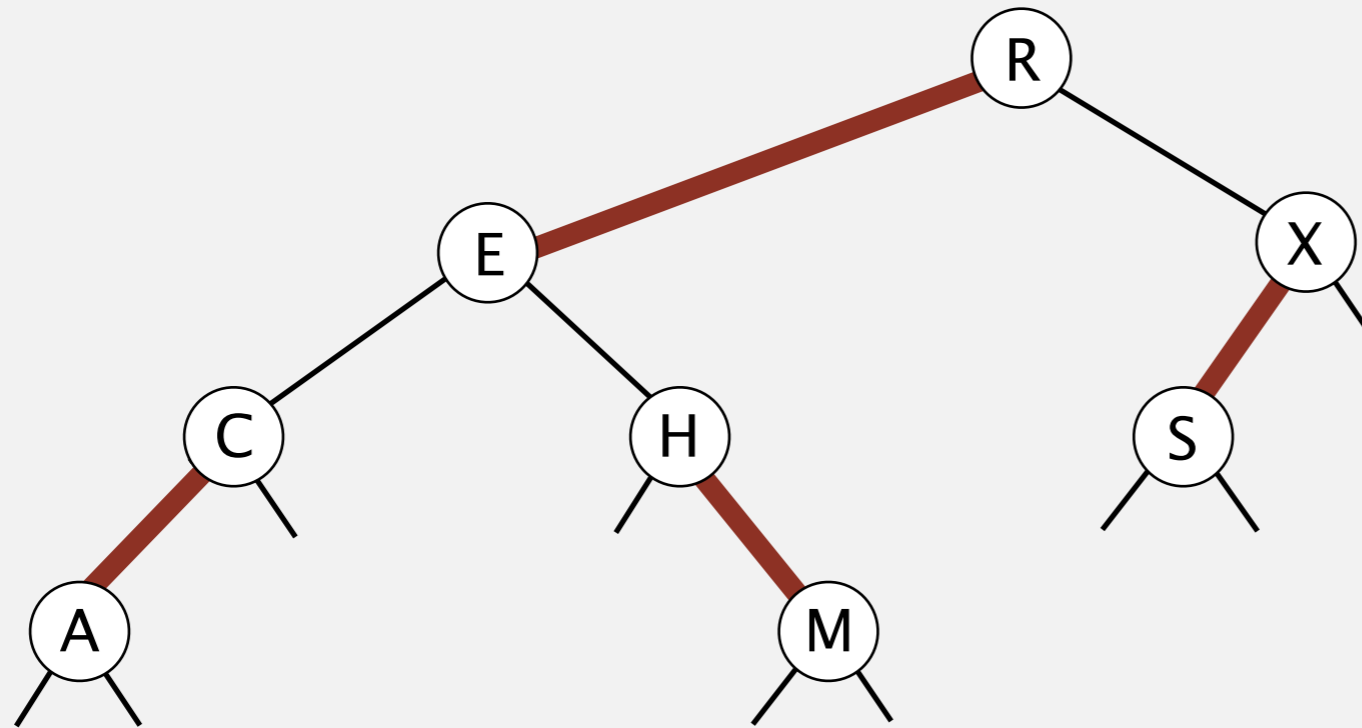**red–black BST**

Red-black BST insertion
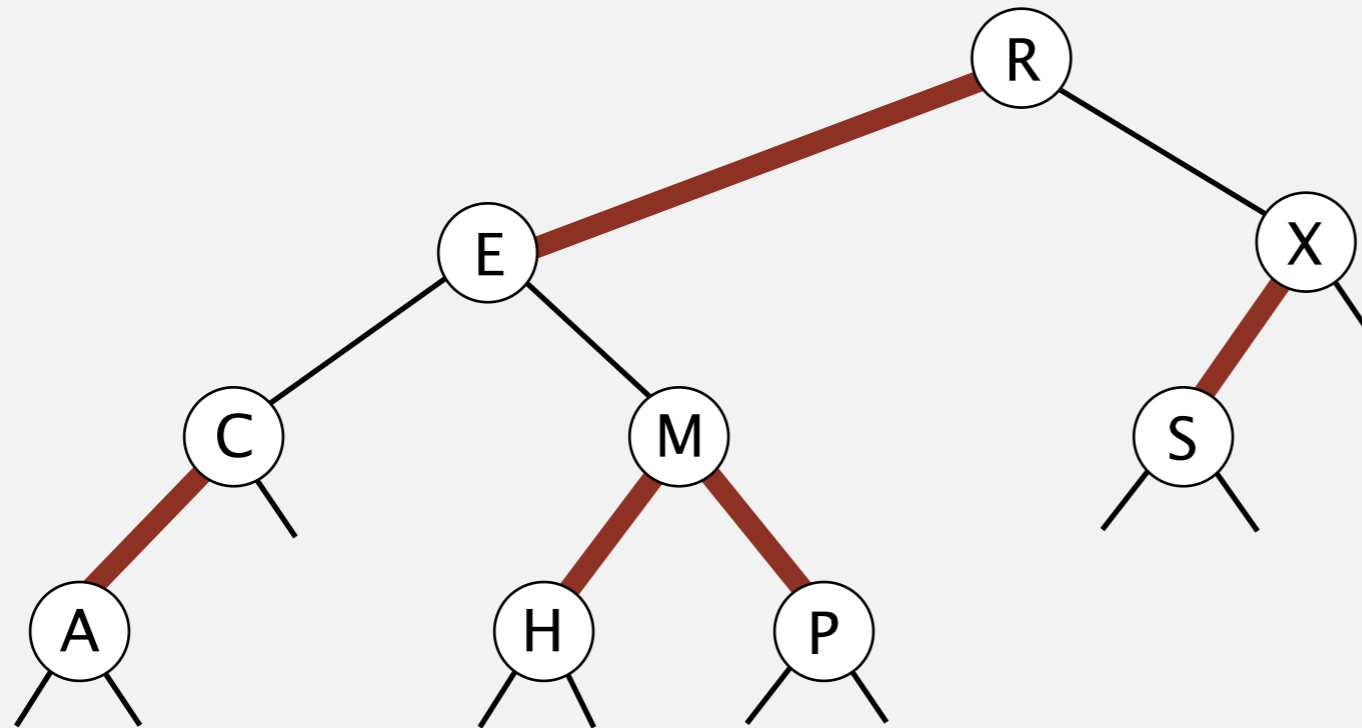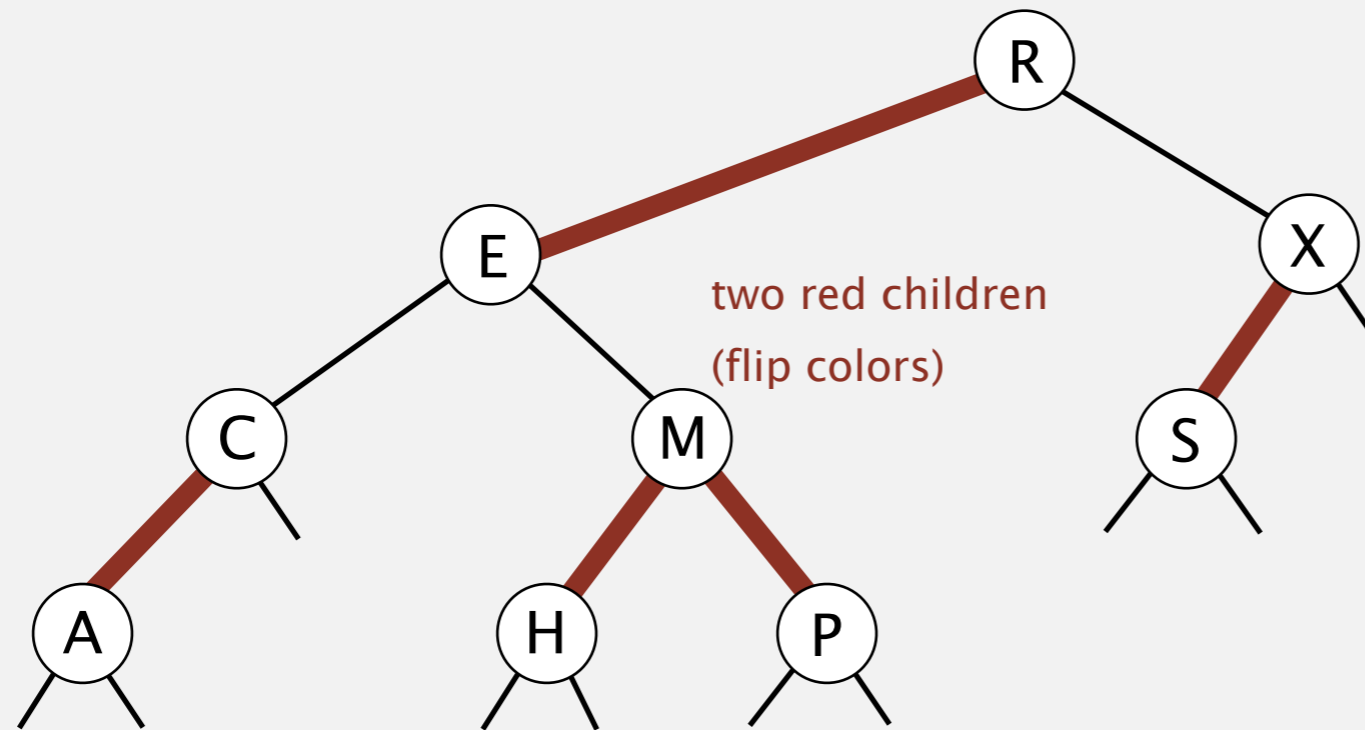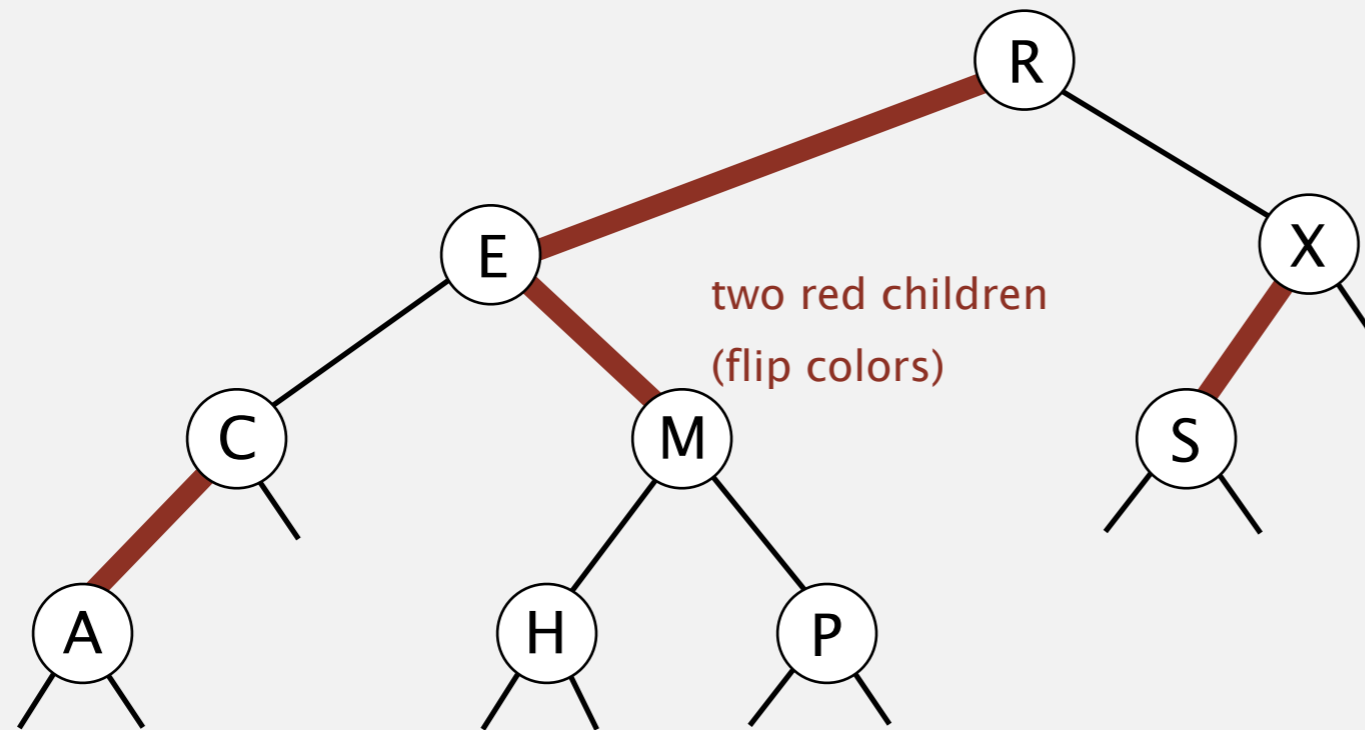
red-black BST

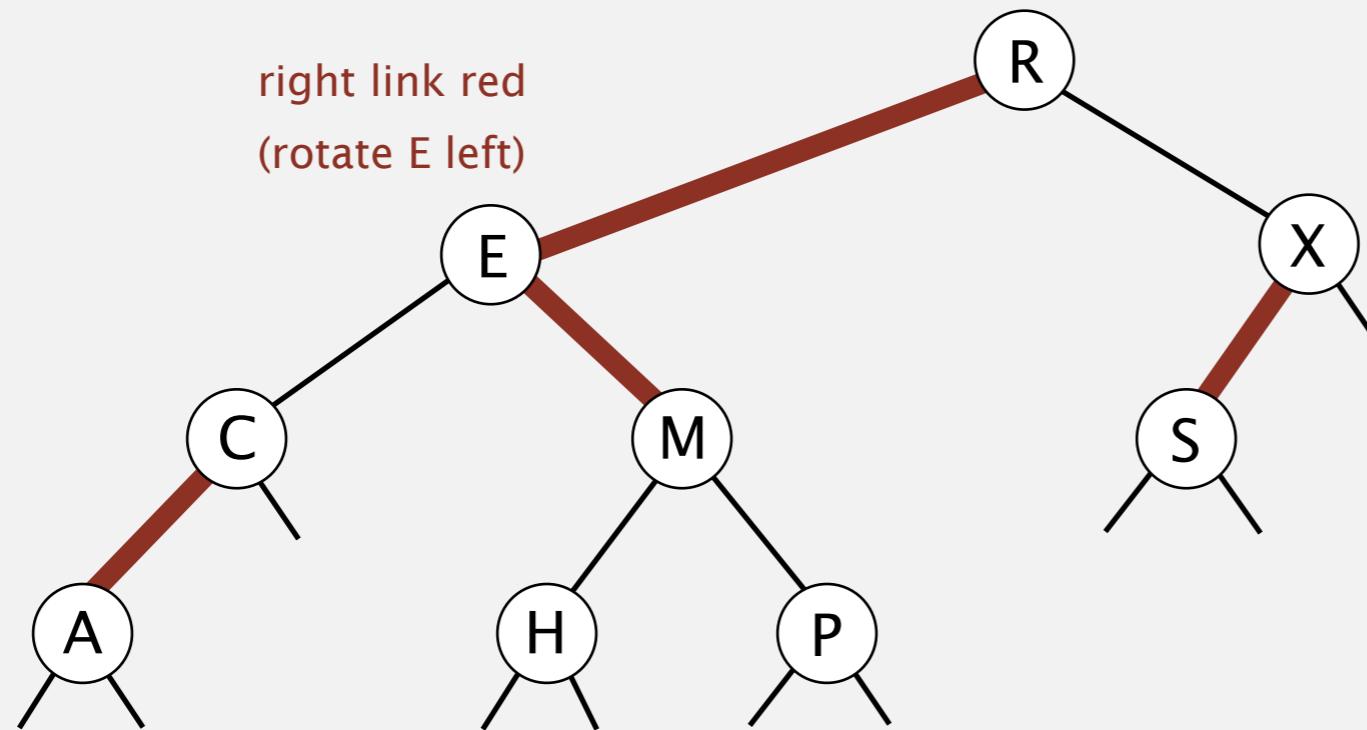# Red-black BST insertion

insert L

Red-black BST insertion

insert L

right link red
(rotate H left)

Red-black BST insertion

red-black BST

LLRB tree insertion trace

Standard indexing client.

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

LLRB tree insertion trace

Standard indexing client (continued).

red–black BST   corresponding 2–3 tree

Insertion in a LLRB tree: Java implementation

Same code for both cases.

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


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

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

    return h;
}
```

insert at bottom (and color red)

lean left
balance 4-node
split 4-node

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

Insertion in a LLRB tree: visualization

N = 255
max = 8
avg = 7.0
opt = 7.0

255 insertions in ascending order

Insertion in a LLRB tree: visualization

Remark. Only a few extra lines of code to standard BST insert.

N = 255
max = 8
avg = 7.0
opt = 7.0

255 insertions in descending order

Insertion in a LLRB tree: visualization

Remark. Only a few extra lines of code to standard BST insert.

N = 255
max = 10
avg = 7.3
opt = 7.0

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: frequency counter

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

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

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

BALANCED SEARCH TREES

- ‣ 2-3 search trees
- ‣ Red-black BSTs
- ‣ **B-trees**
- ‣ Geometric applications of BSTs

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

2-node

sentinel key

internal 3-node

each red key is a copy
of min key in subtree

external
3-node

external 5-node (full)

external 4-node

* K

* D H

K Q U

* B C   D E F   H I J   K M N O P   Q R T   U W X Y

client keys (black)
are in external nodes

all nodes except the root are 3-, 4- or 5-nodes

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 for E

*follow this link because E is between * and K*

follow this link because E is between D and H

search for E in this 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.

each line shows the result
of inserting one key
in some page

white: unoccupied portion of page

black: occupied portion of page

full page, about to split

full page splits into
two half-full pages
then a new key is added
to one of them

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.`

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.

Balanced Search Trees

- ‣ 2-3 search trees
- ‣ Red-black BSTs
- ‣ B-trees
- ‣ **Geometric applications of BSTs**

‣ **kd trees**

2-d orthogonal range search

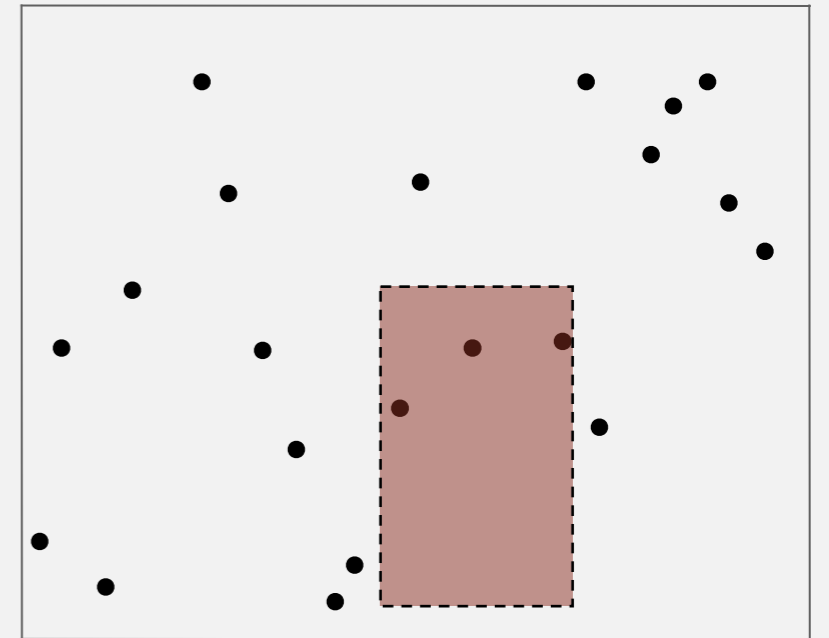Extension of ordered symbol-table to 2d keys.

- Insert a 2d key.
- Delete a 2d key.
- Search for a 2d key.
- Range search: find all keys that lie in a 2d range.
- Range count: number of keys that lie in a 2d range.

Geometric interpretation.

- Keys are point in the plane.
- Find/count points in a given $h-v$ rectangle.

rectangle is axis-aligned

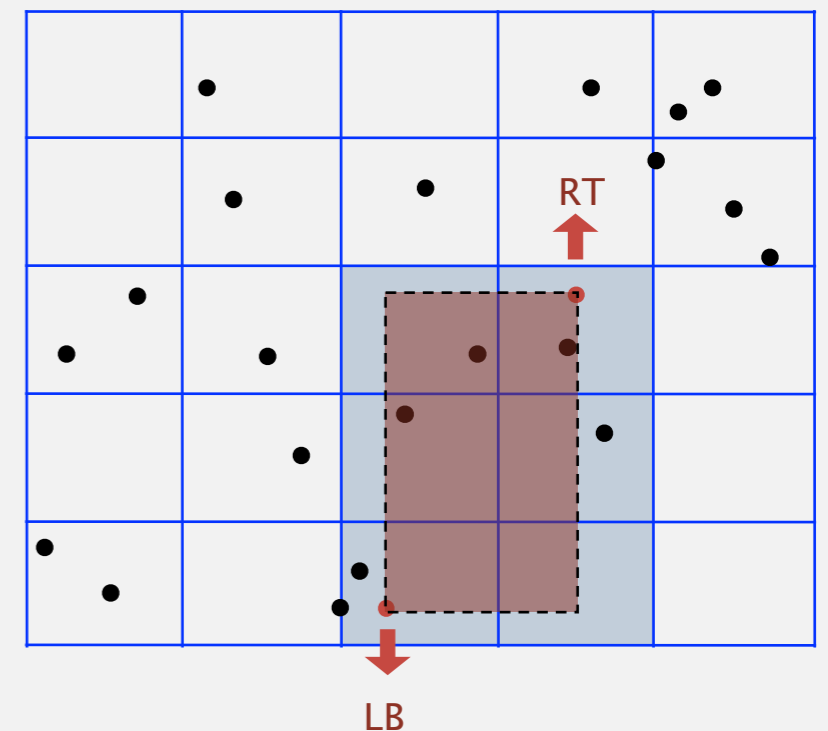Applications. Networking, circuit design, databases,...

2d orthogonal range search: grid implementation

Grid implementation.

- Divide space into M-by-M grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add (x, y) to list for corresponding square.
- Range search: examine only those squares that intersect 2d range query.

2d orthogonal range search: grid implementation costs
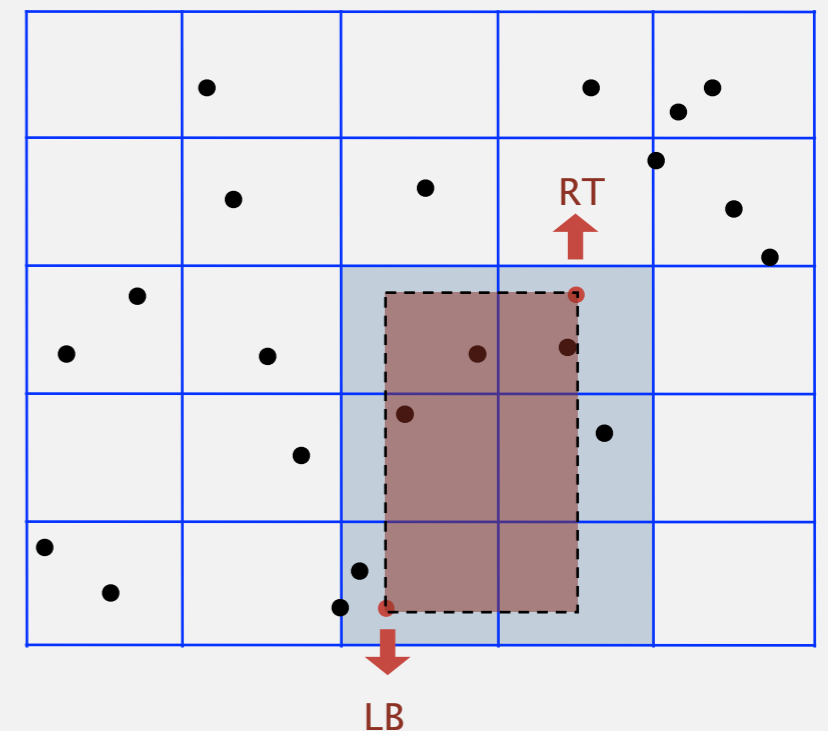
Space-time tradeoff.

- Space: $M^2 + N$.
- Time: $1 + N/M^2$ per square examined, on average.

Choose grid square size to tune performance.

- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: $\sqrt{N}$-by-$\sqrt{N}$ grid.

Running time. [if points are evenly distributed]

- Initialize data structure: N.
- Insert point: 1.
- Range search: 1 per point in range.

choose M ~ √N

Clustering

Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.
- Lists are too long, even though average length is short.
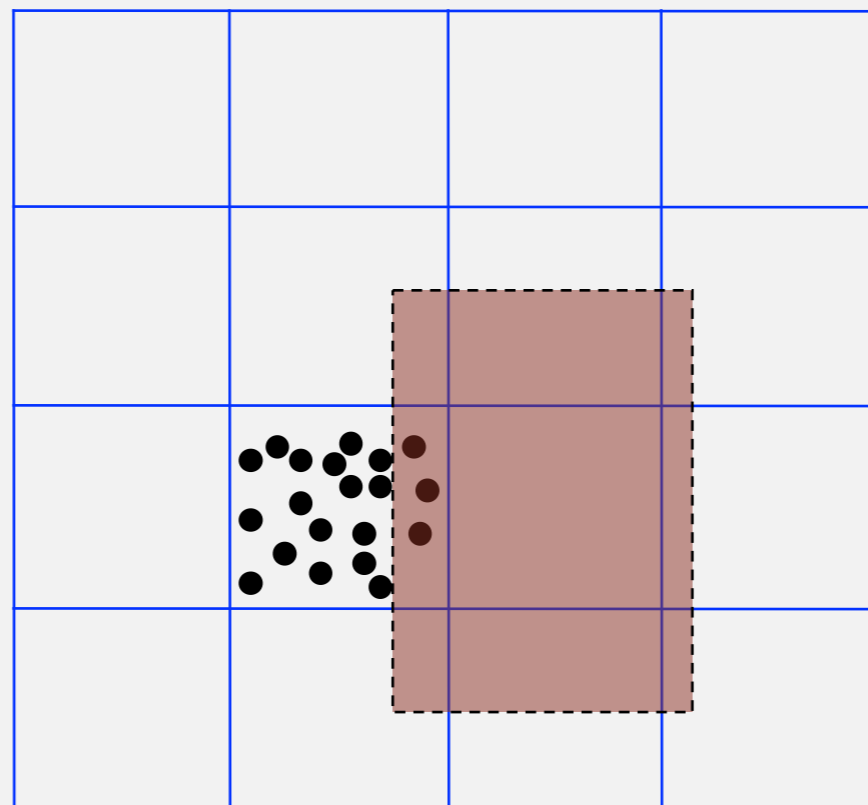- Need data structure that gracefully adapts to data.

Clustering

Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.

13,000 points, 1000 grid squares

half the squares are empty

half the points are
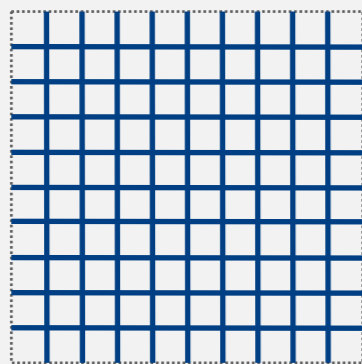in 10% of the squares

Space-partitioning trees

Use a tree to represent a recursive subdivision of 2d space.

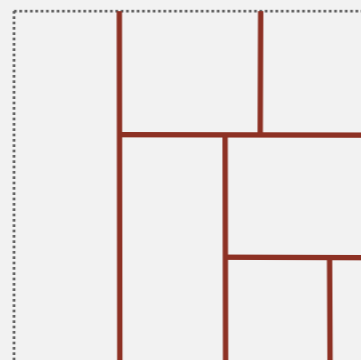Grid. Divide space uniformly into squares.

2d tree.   Recursively divide space into two halfplanes.

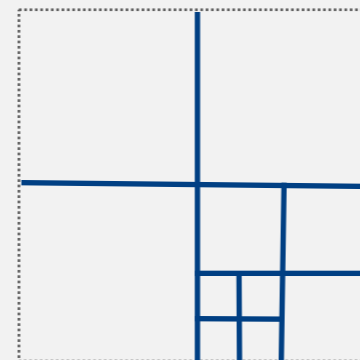Quadtree. Recursively divide space into four quadrants.

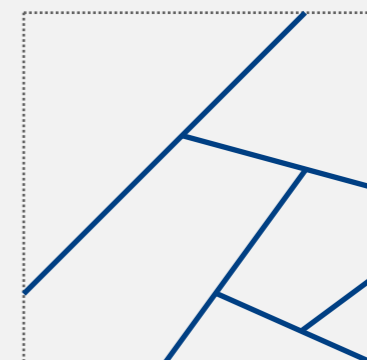BSP tree. Recursively divide space into two regions.

Grid   **2d tree**   **Quadtree**   **BSP tree**
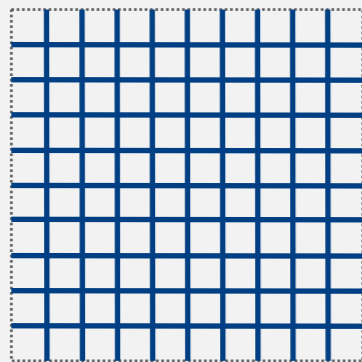
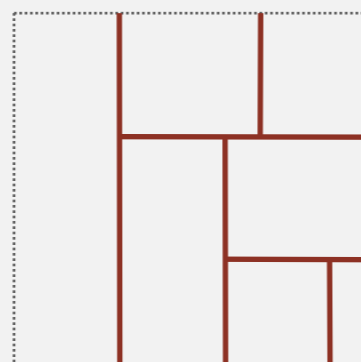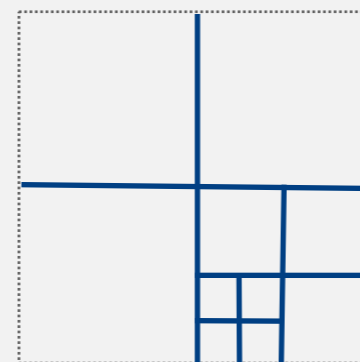# Space-partitioning trees: applications

Applications.

- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.

Grid   **2d tree**   **Quadtree**   **BSP tree**

Kd tree

Kd tree. Recursively partition k-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.

level ≡ i (mod k)

points whose ith coordinate is less than p's

points whose ith coordinate is greater than p's

Efficient, simple data structure for processing k-dimensional data.
• Widely used.
• Adapts well to high-dimensional and clustered data.
• Discovered by an undergrad in an algorithms class!

Jon Bentley

N-body simulation

Goal. Simulate the motion of N particles, mutually affected by gravity.

Brute force. For each pair of particles, compute force. $F = \dfrac{G\,m_1\,m_2}{r^2}$

Appel algorithm for N-body simulation

Key idea. Suppose particle is far, far away from cluster of particles.

- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and center of mass of aggregate particle.

Appel algorithm for N-body simulation

- Build 3d-tree with N particles as nodes.
- Store center-of-mass of subtree in each node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to subdivision is sufficiently large.

AN EFFICIENT PROGRAM FOR MANY-BODY SIMULATION*

ANDREW W. APPEL†

Abstract. The simulation of N particles interacting in a gravitational force field is useful in astrophysics, but such simulations become costly for large N. Representing the universe as a tree structure with the particles at the leaves and internal nodes labeled with the centers of mass of their descendants allows several simultaneous attacks on the computation time required by the problem. These approaches range from algorithmic changes (replacing an $O(N^2)$ algorithm with an algorithm whose time-complexity is believed to be $O(N \log N)$) to data structure modifications, code-tuning, and hardware modifications. The changes reduced the running time of a large problem ($N = 10{,}000$) by a factor of four hundred. This paper describes both the particular program and the methodology underlying such speedups.

Impact. Running time per step is $N \log N$ instead of $N^2 \Rightarrow$ enables new research.