http://algs4.cs.princeton.edu

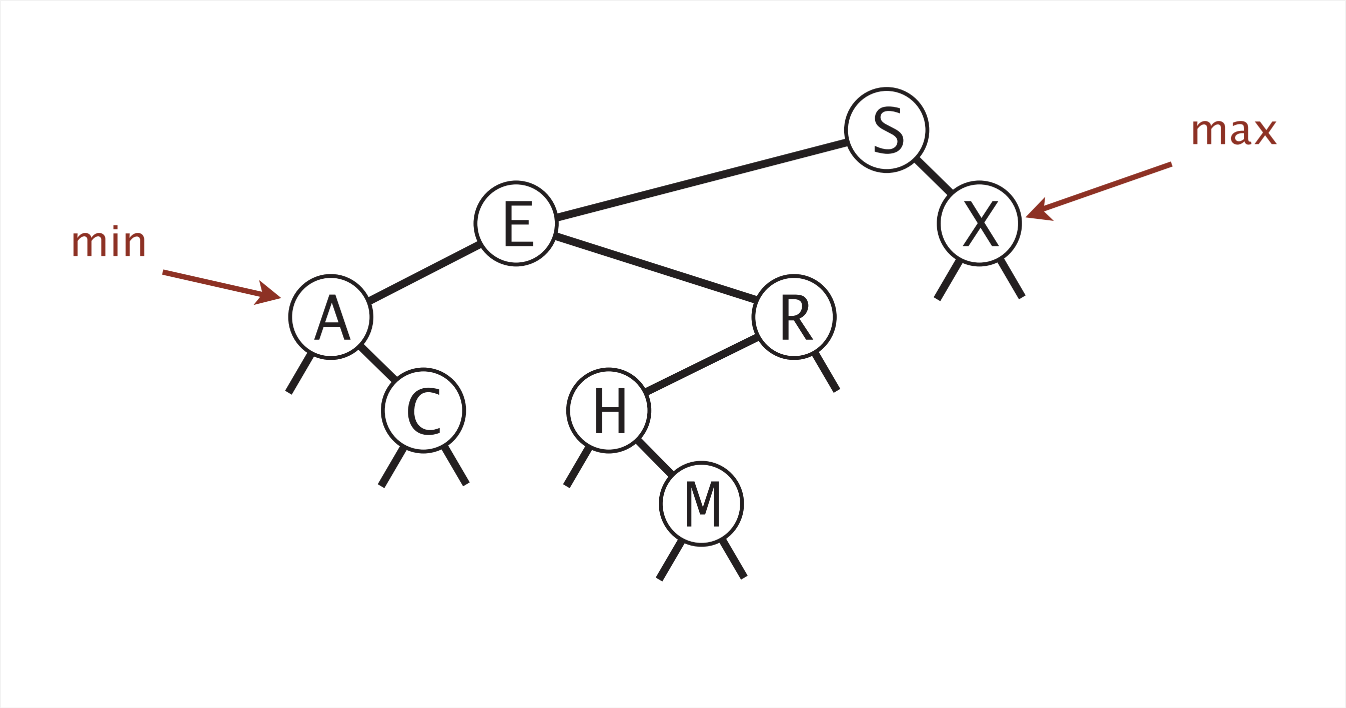
3.2 BINARY SEARCH TREES

- BSTs
- iteration
- ordered operations
- deletion

Minimum and maximum

Minimum. Smallest key in BST.

Maximum. Largest key in BST.

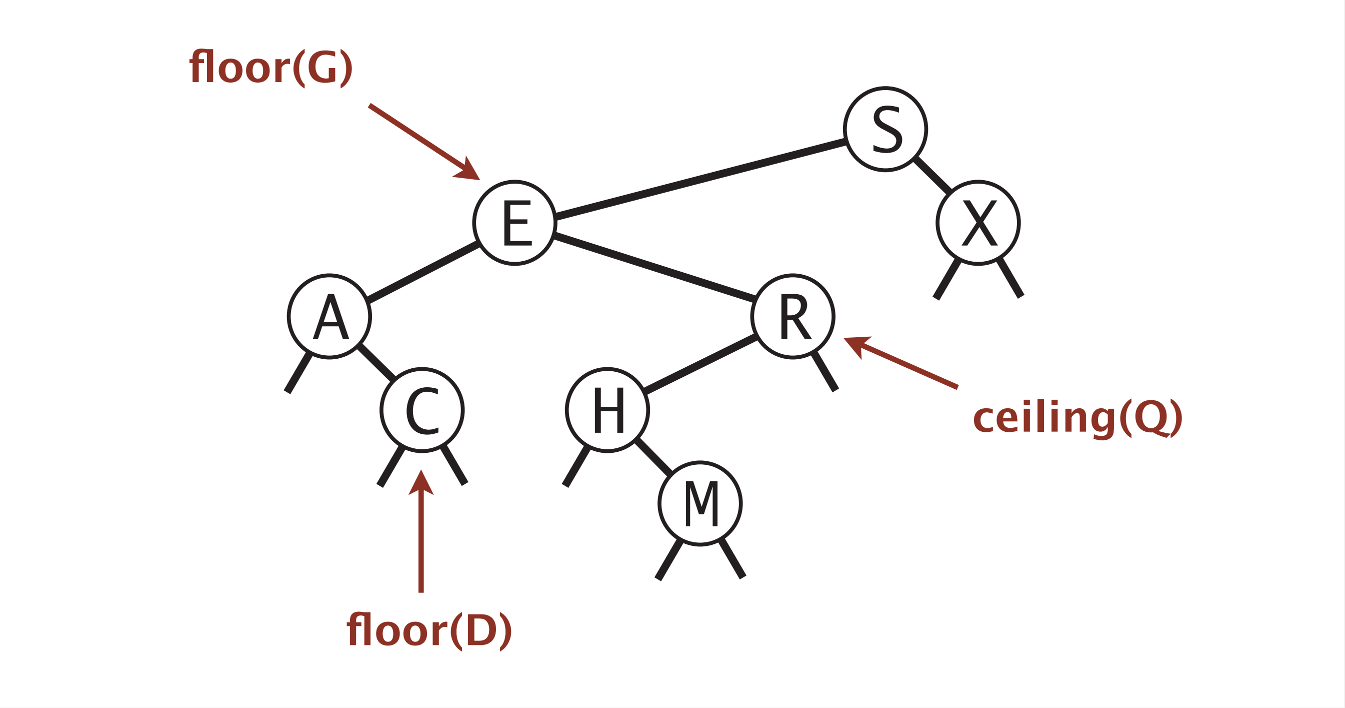


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key in BST $\leq$ query key.

Ceiling. Smallest key in BST $\geq$ query key.

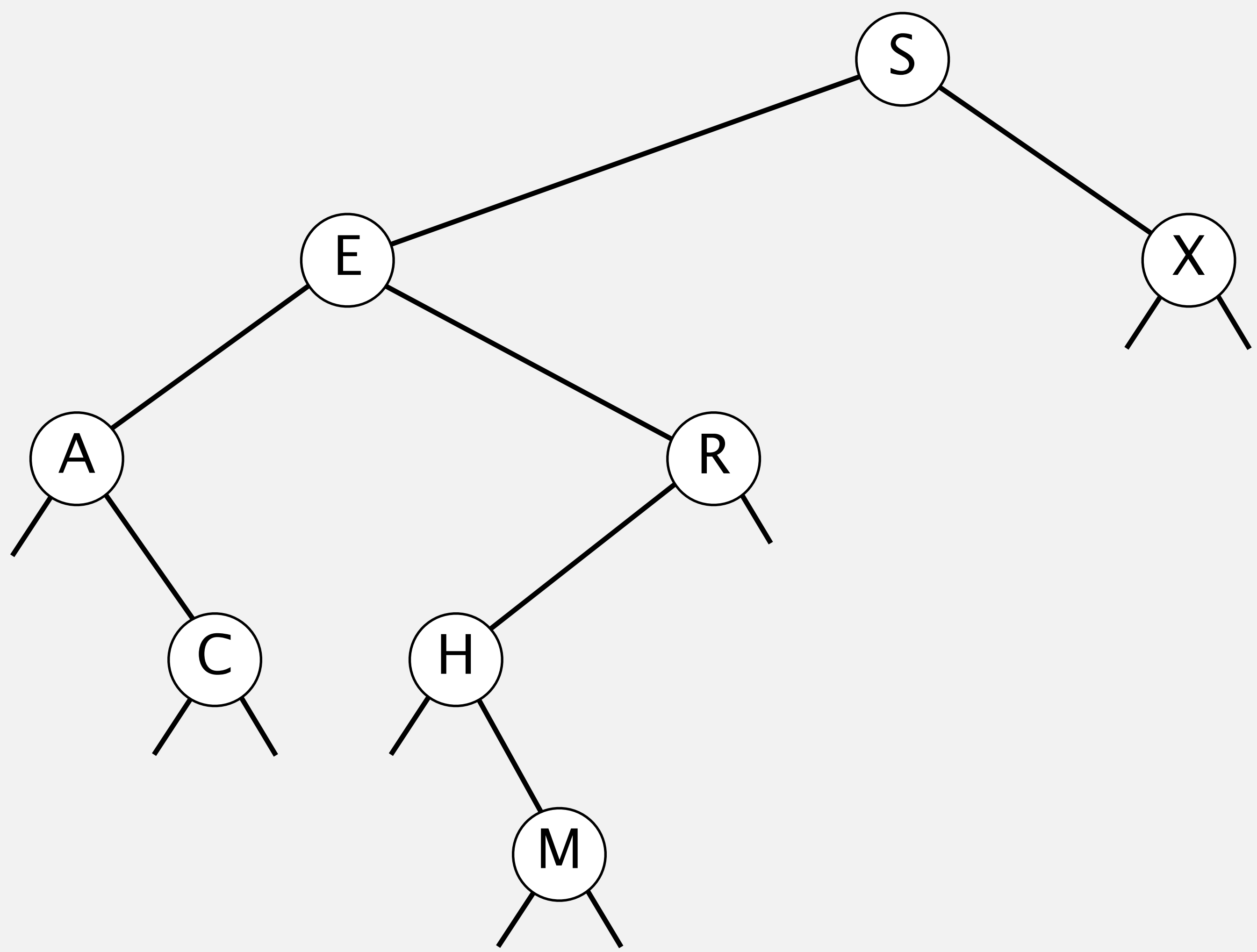


Q. How to find the floor / ceiling?

Floor in a BST demo

Floor. Find the largest key in a BST that is $\leq k$?

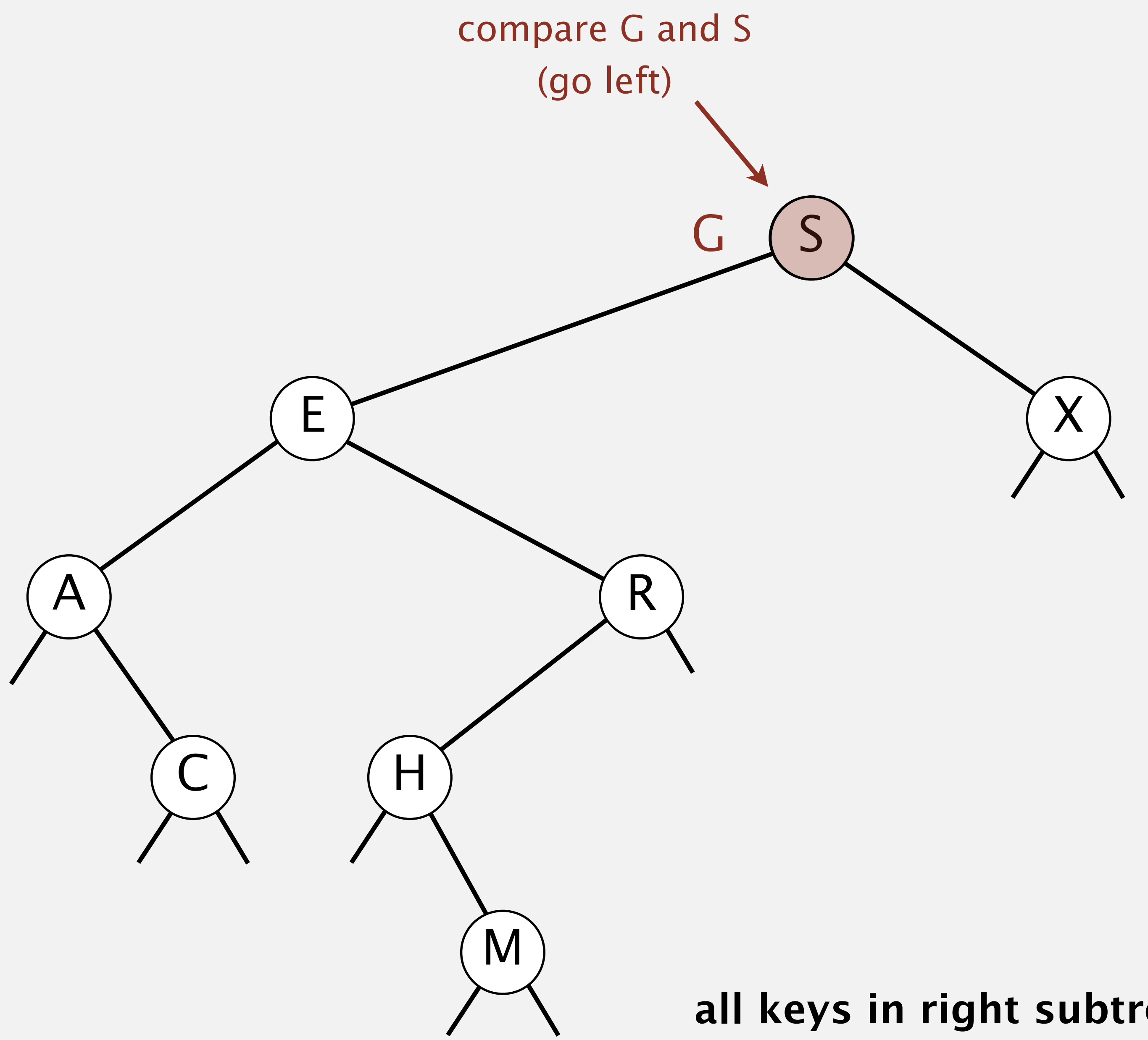
floor of G



Floor in a BST demo

Floor. Find the largest key in a BST that is $\leq k$?

floor of G



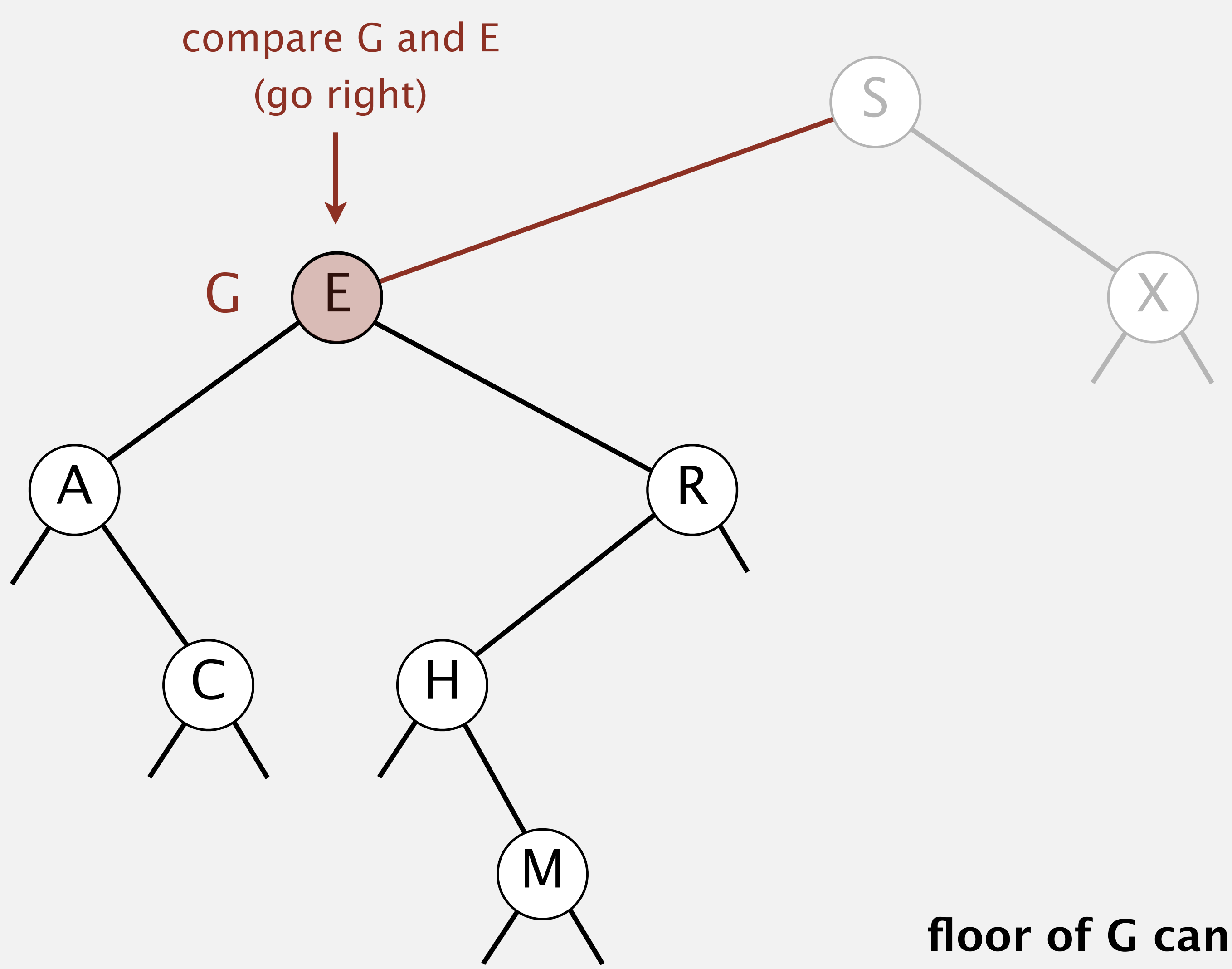
all keys in right subtree of S are greater than G
⇒ compute floor of G in left subtree

Floor in a BST demo

Floor. Find the largest key in a BST that is ≤ k ?

floor of G

E



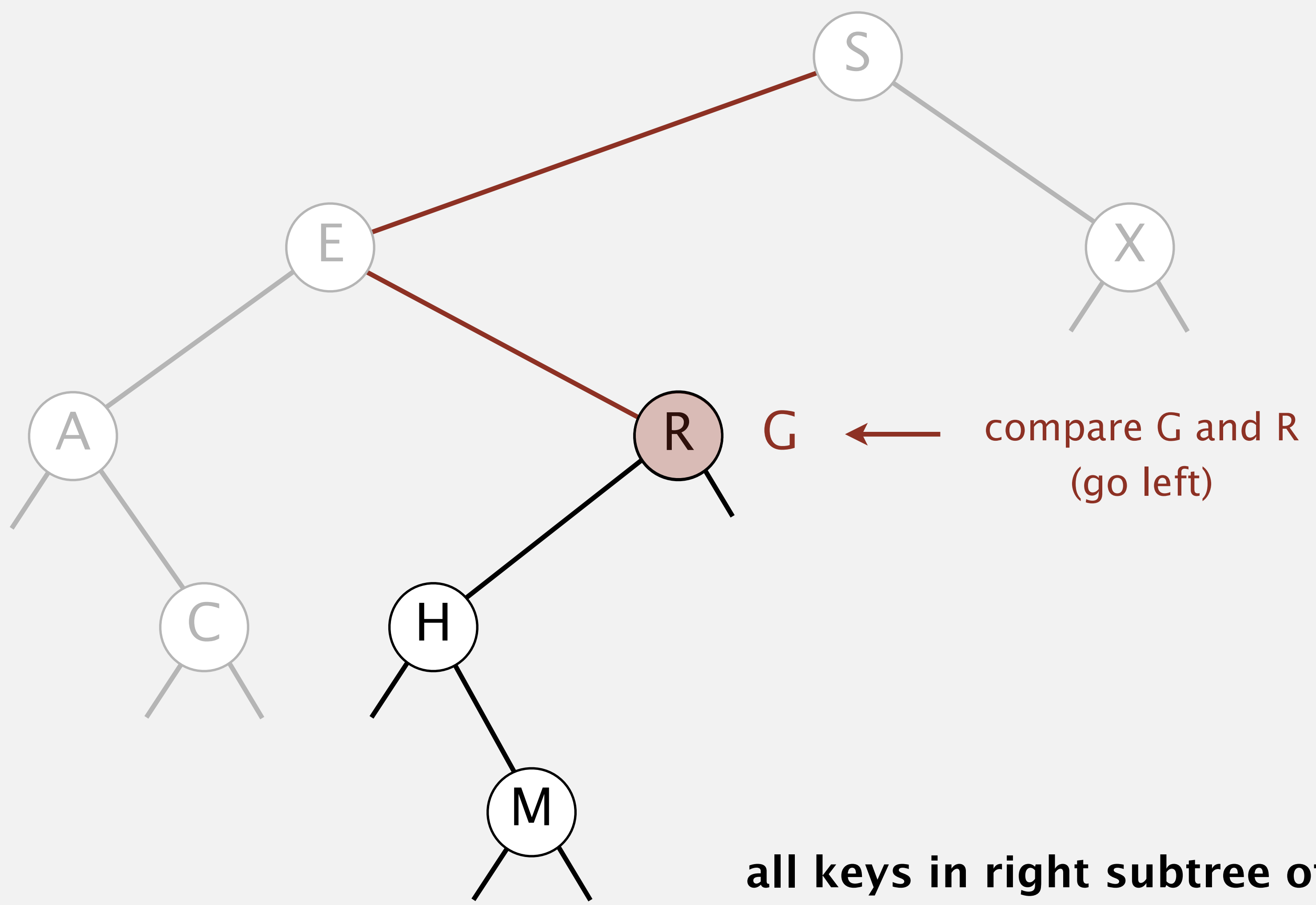
floor of G can't be in left subtree;
floor is either E or floor of G in right subtree

Floor in a BST demo

Floor. Find the largest key in a BST that is $\leq k$?

floor of G

E



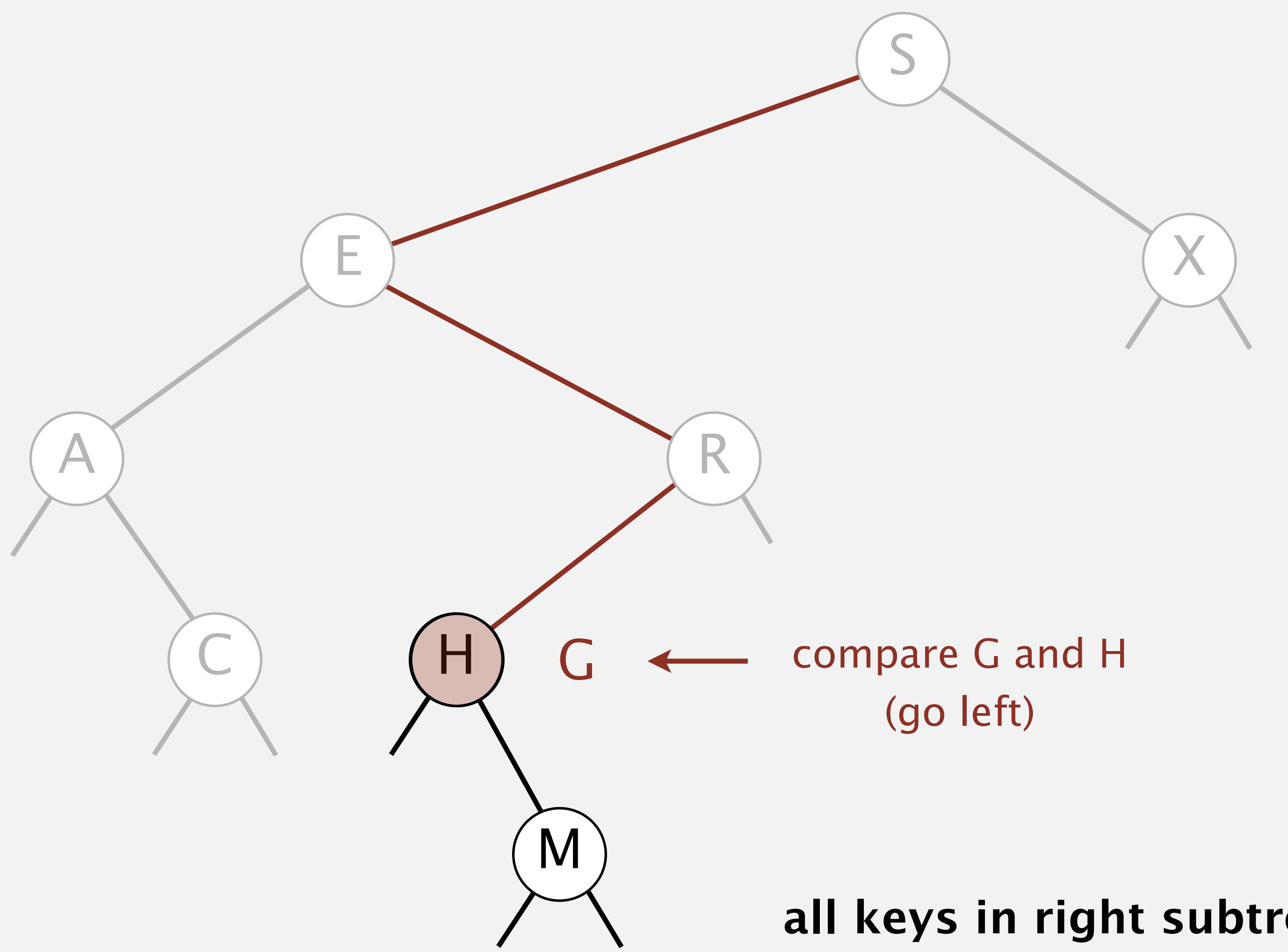
all keys in right subtree of R are greater than G
⇒ compute floor of G in left subtree

Floor in a BST demo

Floor. Find the largest key in a BST that is $\leq k$?

floor of G

E

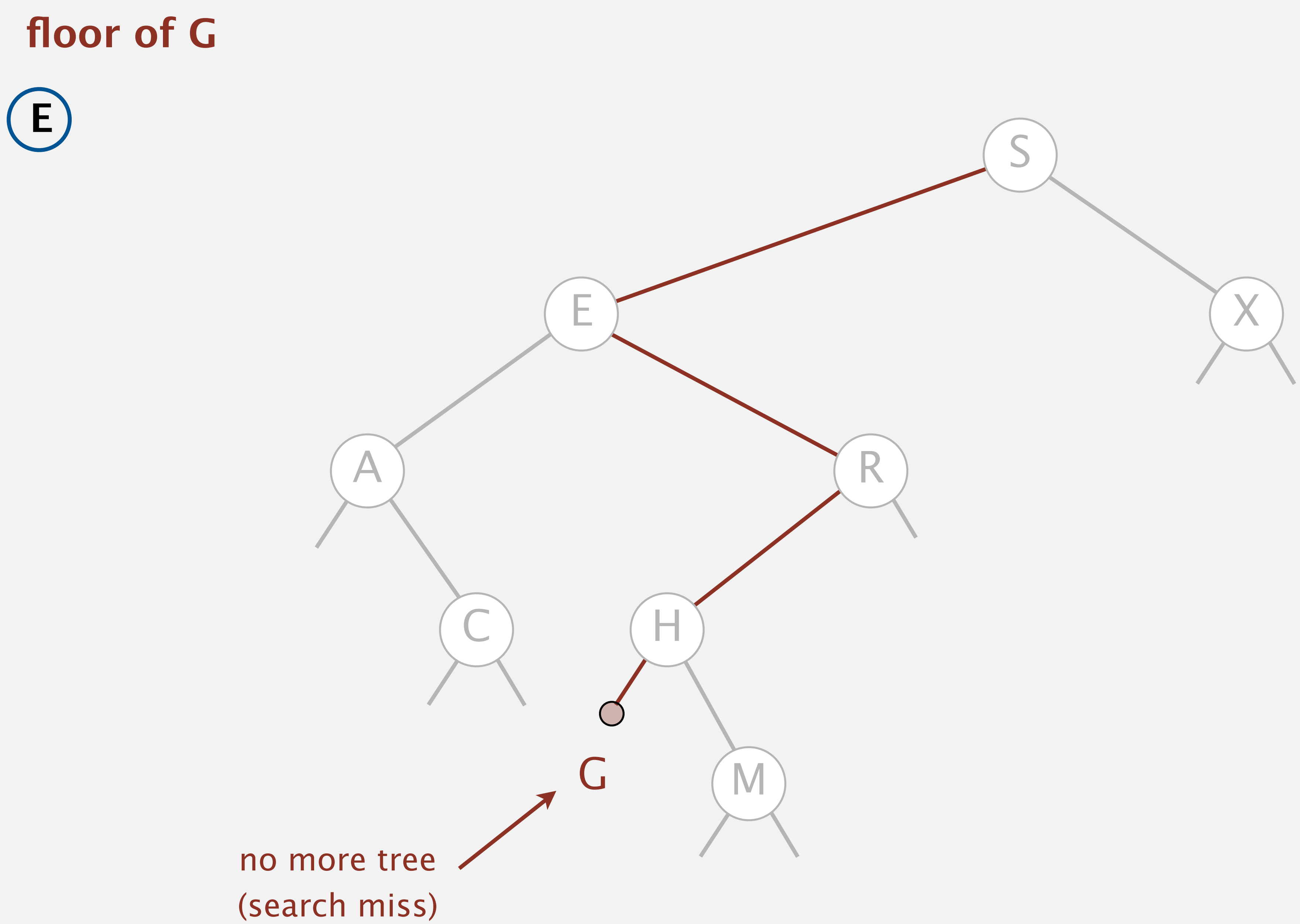


all keys in right subtree of H are greater than G

⇒ compute floor of G in left subtree

Floor in a BST demo

Floor. Find the largest key in a BST that is ≤ k ?



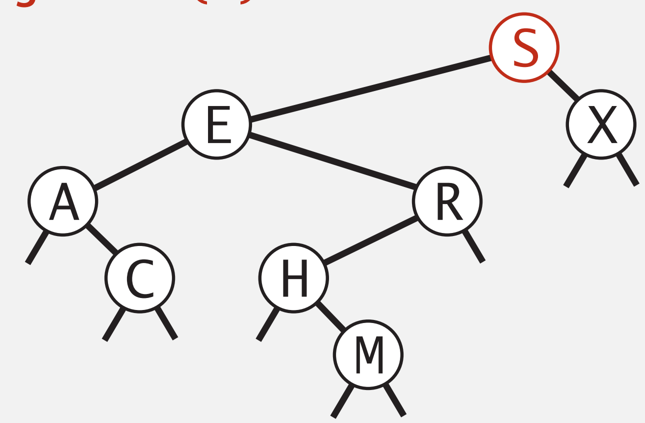
Computing the floor

Floor. Largest key in BST $\leq k$?

Case 1. [key in node $x = k$]

The floor of k is k.

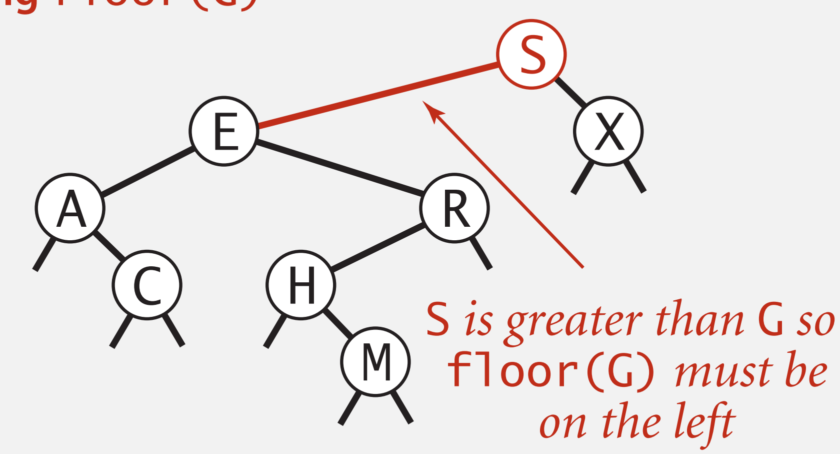
finding floor(S)



Case 2. [key in node $x > k$]

The floor of k is in the left subtree of x.

finding floor(G)

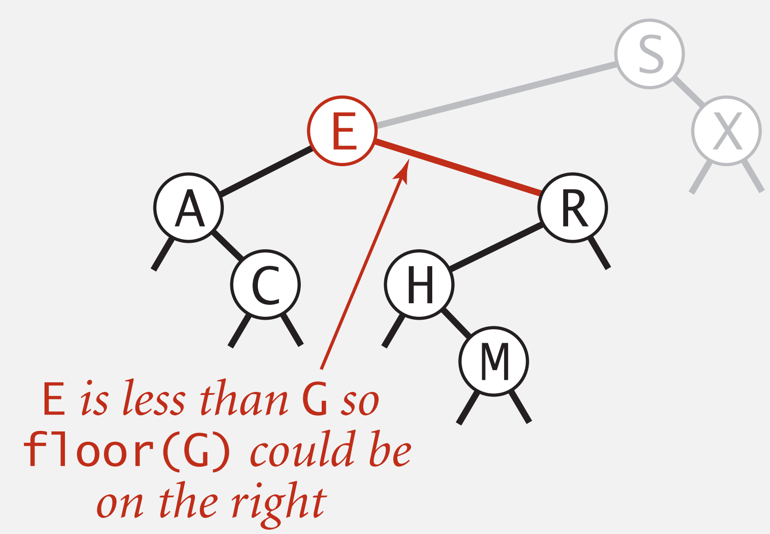


Case 3. [key in node $x < k$]

The floor of k can't be in left subtree of x:

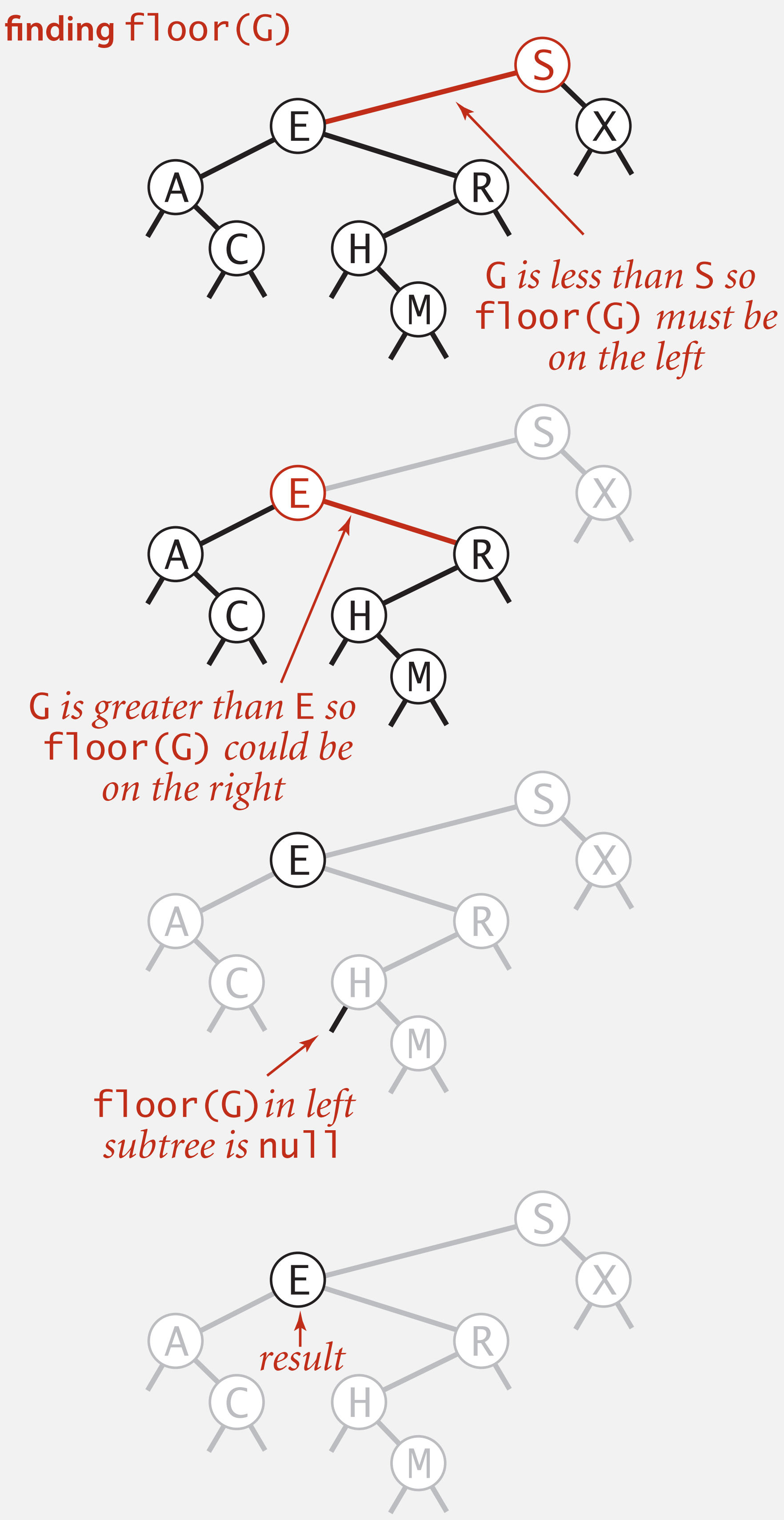
it is either in the right subtree of x or

it is the key in node x.



Computing the floor

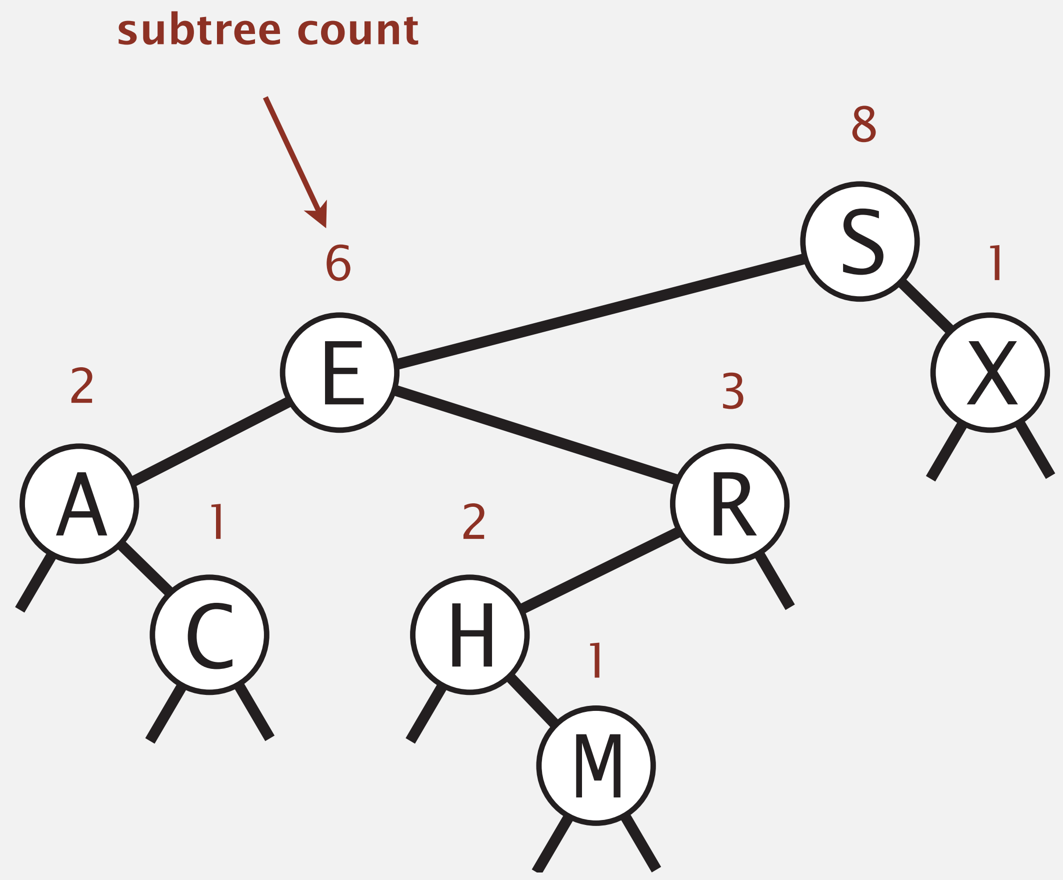
```
public Key floor(Key key)
{  return floor(root, key); }

private Key floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp  < 0) return floor(x.left, key);

   Key t = floor(x.right, key);
   if (t != null) return t;
   else           return x.key;

}
```



Rank and select

Q. How to implement `rank()` and `select()` efficiently for BSTs?

A. In each node, store the number of nodes in its subtree.



BST implementation: subtree counts

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int count;
}
```

number of nodes in subtree

```
public int size()
{  return size(root);  }

private int size(Node x)
{
   if (x == null) return 0;
   return x.count;
}
```

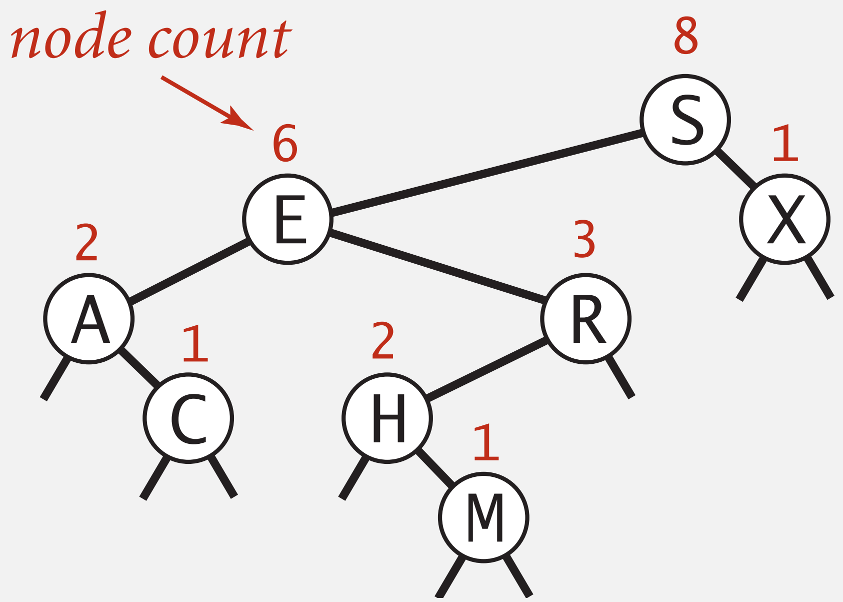
ok to call when x is null

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

initialize subtree count to 1

Computing the rank

Rank. How many keys in BST < k ?



Case 1. [k < key in node]

- Keys in left subtree? *count*
- Key in node? 0
- Keys in right subtree? 0

Case 2. [k > key in node]

- Keys in left subtree? *all*
- Key in node. 1
- Keys in right subtree? *count*

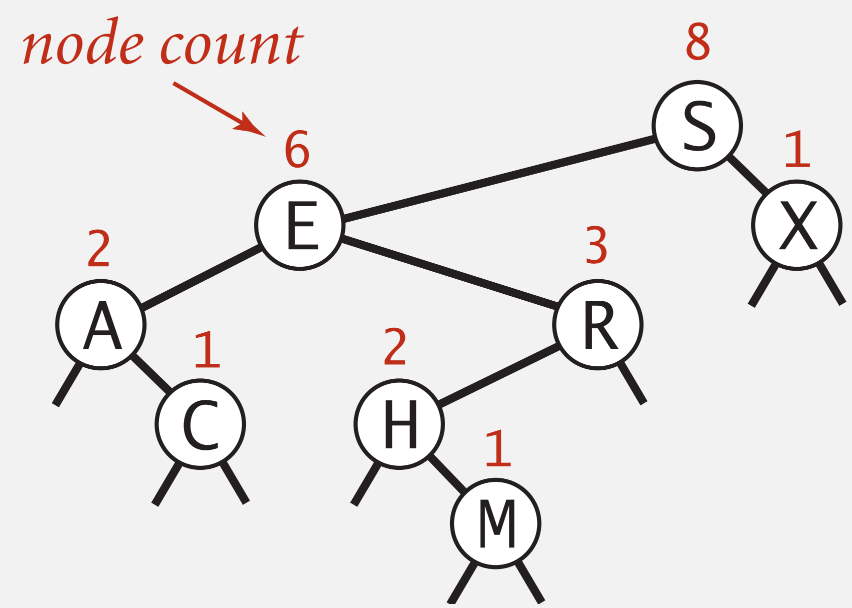
Case 3. [k = key in node]

- Keys in left subtree? *count*
- Key in node. 0
- Keys in right subtree? 0

Rank

Rank. How many keys in BST $< k$?

Easy recursive algorithm (3 cases!)

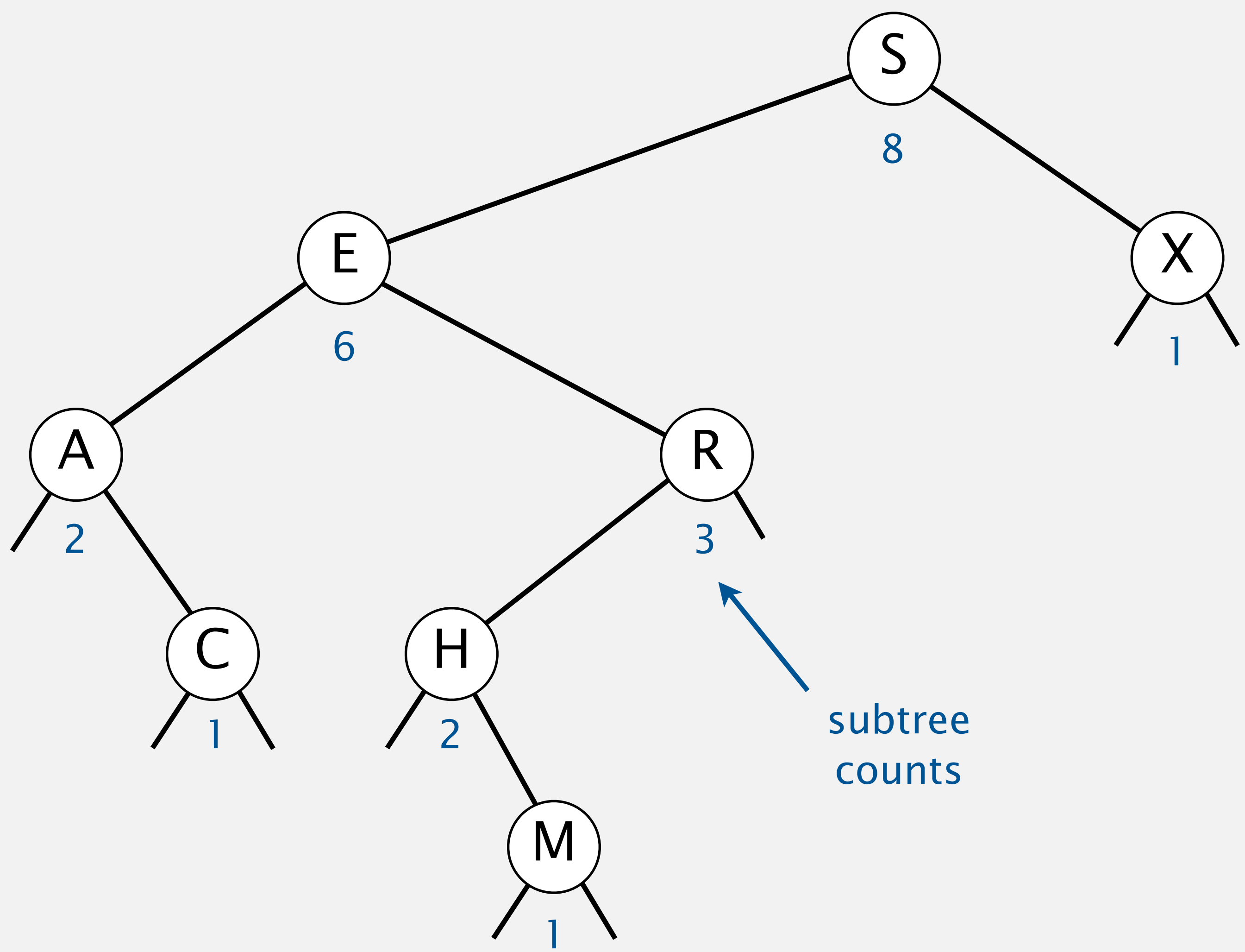



```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return rank(key, x.left);
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);
   else if (cmp == 0) return size(x.left);
}
```

Selection in a BST demo

Select. Find the key in a BST of rank k.

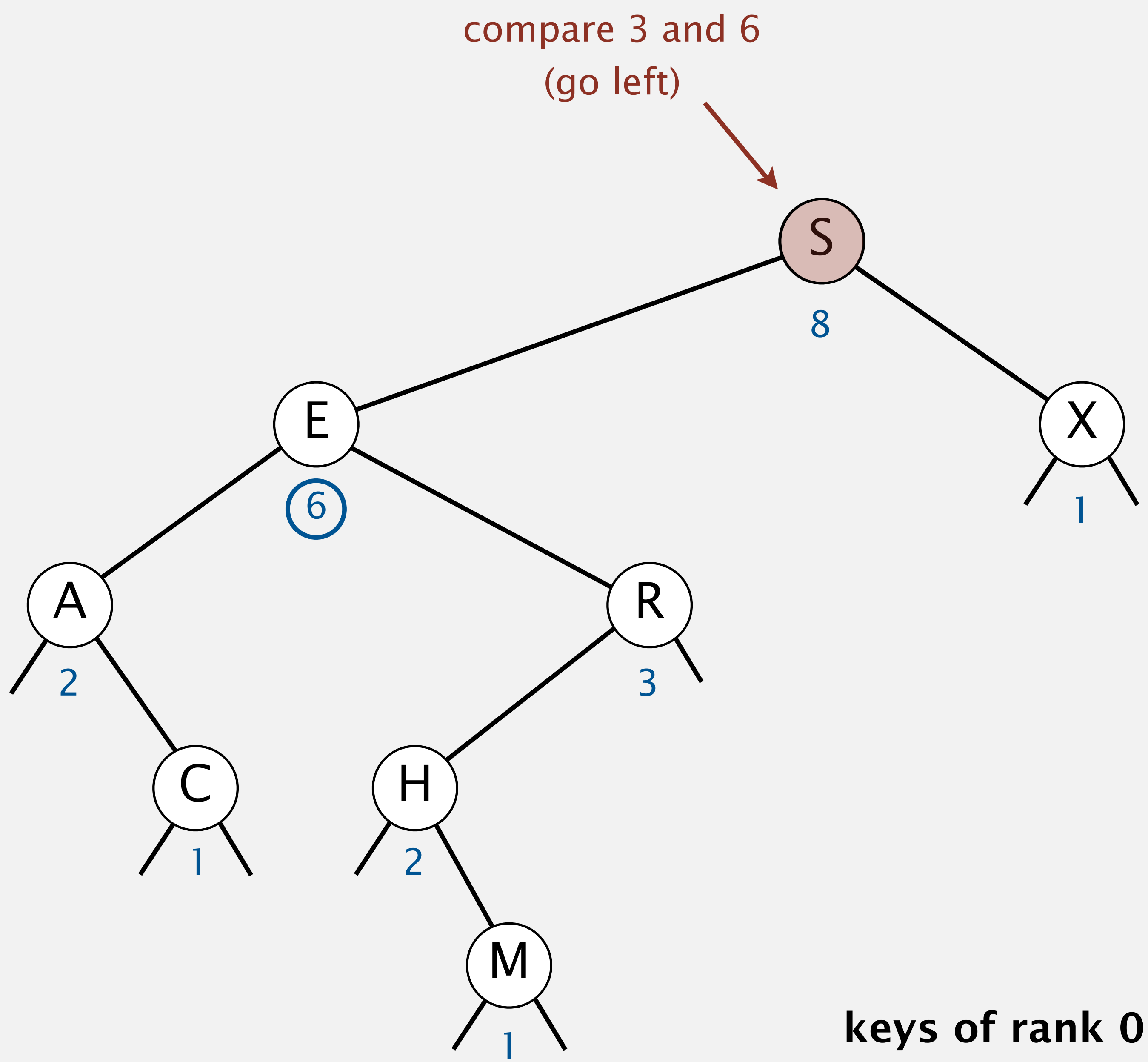
rank(S, 3)



Selection in a BST demo

Select. Find the key in a BST of rank k.

rank(S, 3)



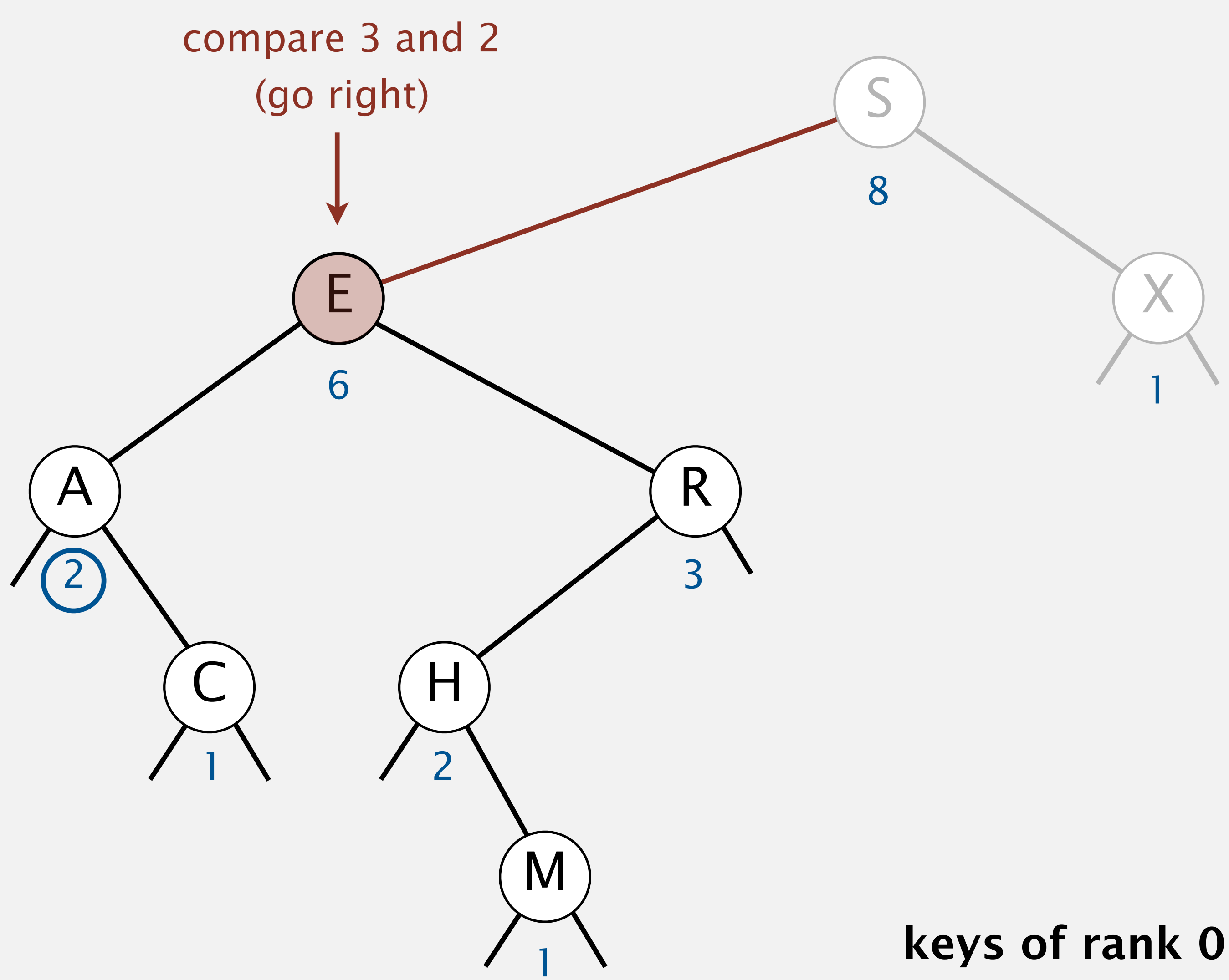
keys of rank 0–5 are in left subtree ⇒
find key of rank 3 in subtree rooted at E

Selection in a BST demo

Select. Find the key in a BST of rank k.

rank(S, 3)

rank(E, 3)



keys of rank 0–1 are in left subtree ⇒

find key of rank 0 in subtree rooted at R

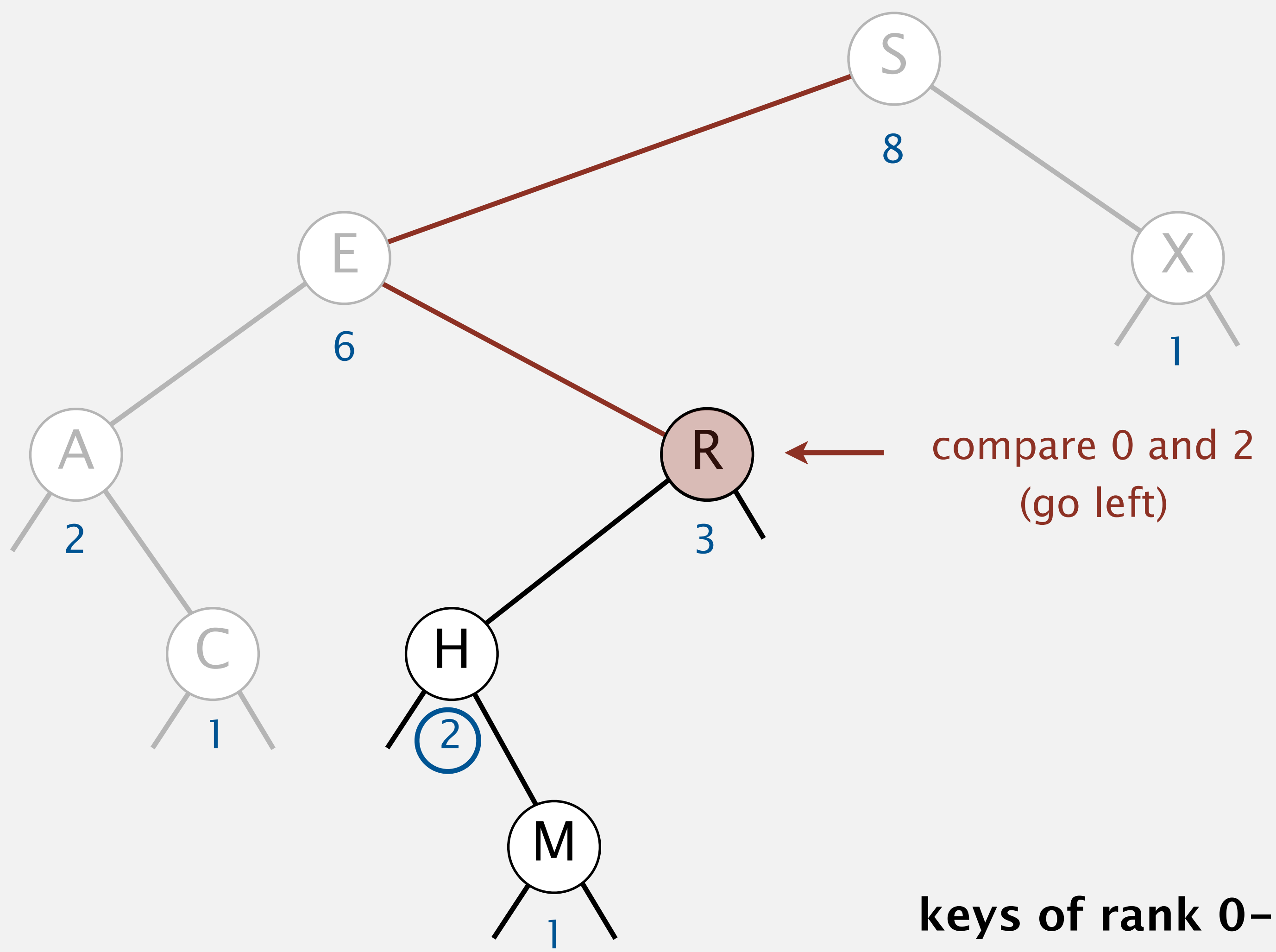
Selection in a BST demo

Select. Find the key in a BST of rank k.

rank(S, 3)

rank(E, 3)

rank(R, 0)



keys of rank 0–1 are in left subtree ⇒

find key of rank 0 in subtree rooted at H

Selection in a BST demo

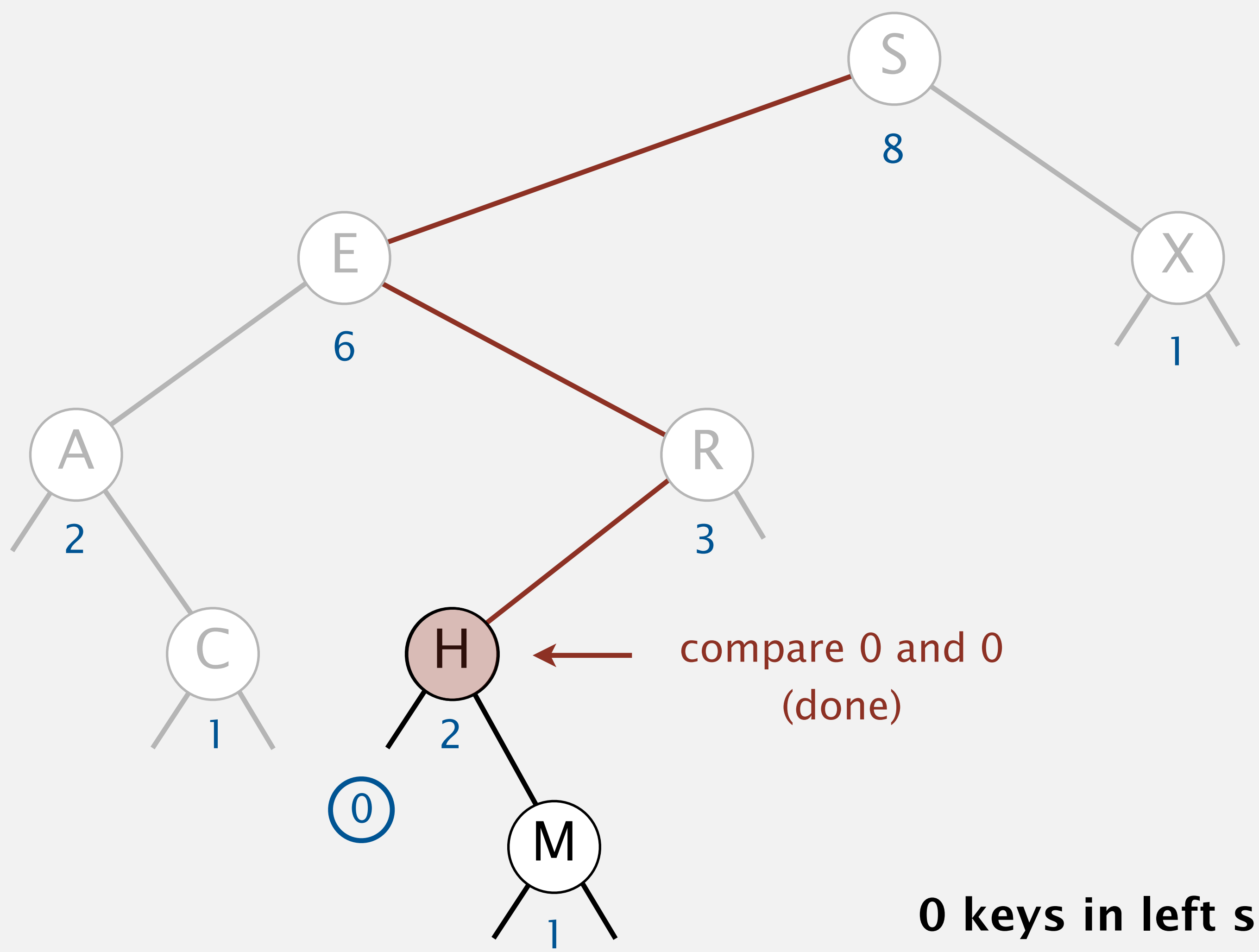
Select. Find the key in a BST of rank k.

rank(S, 3)

rank(E, 3)

rank(R, 0)

rank(H, 0)



0 keys in left subtree ⇒

key of rank 0 in subtree rooted at H is H

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\log N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\log N$	h
rank	N	$\log N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

h = height of BST
(proportional to log N
if keys inserted in random order)

order of growth of running time of ordered symbol table operations

ST implementations: summary

implementation	guarantee		average case		ordered ops?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N	N		`equals()`
binary search (ordered array)	$\log N$	N	$\log N$	N	✔	`compareTo()`
BST	N	N	$\log N$	$\log N$	✔	`compareTo()`
red-black BST	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`

Next lecture. Guarantee logarithmic performance for all operations.

http://algs4.cs.princeton.edu

3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
- *deletion*

ST implementations: summary

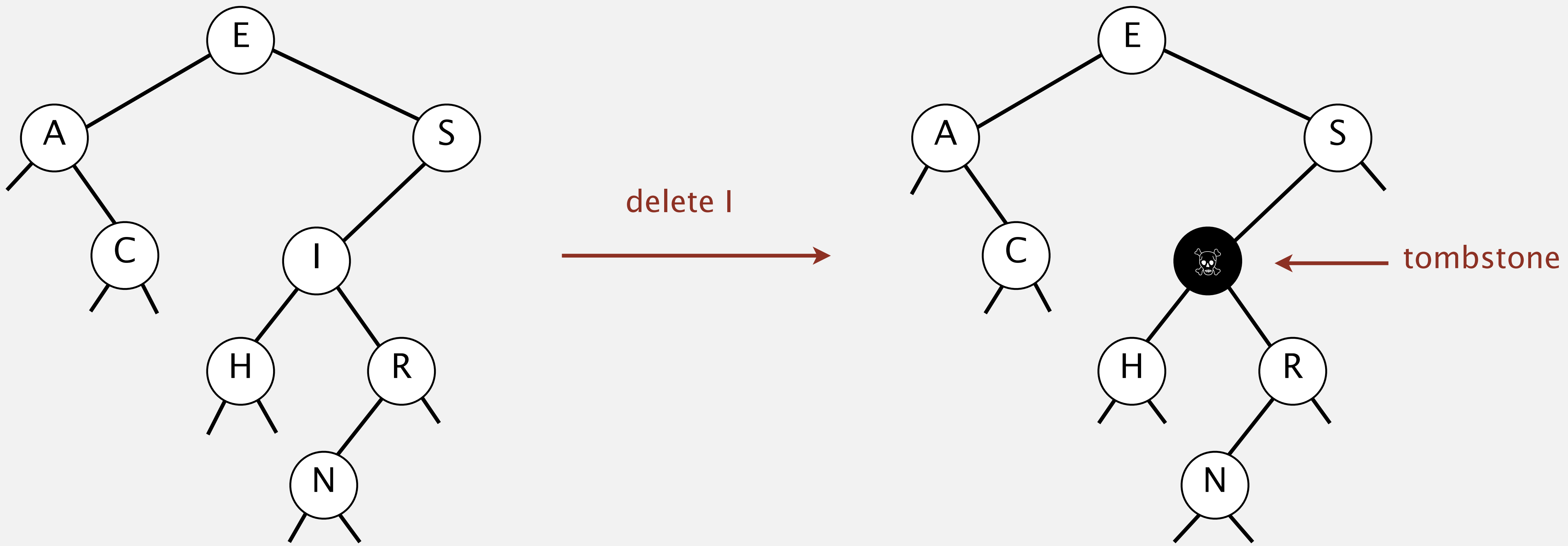
implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N	N	N		`equals()`
binary search (ordered array)	$\log N$	N	N	$\log N$	N	N	✔	`compareTo()`
BST	N	N	N	$\log N$	$\log N$	?	✔	`compareTo()`

Next. Deletion in BSTs.

BST deletion: lazy approach

To remove a node with a given key:

- Set its value to `null`.
- Leave key in tree to guide search (but don't consider it equal in search).



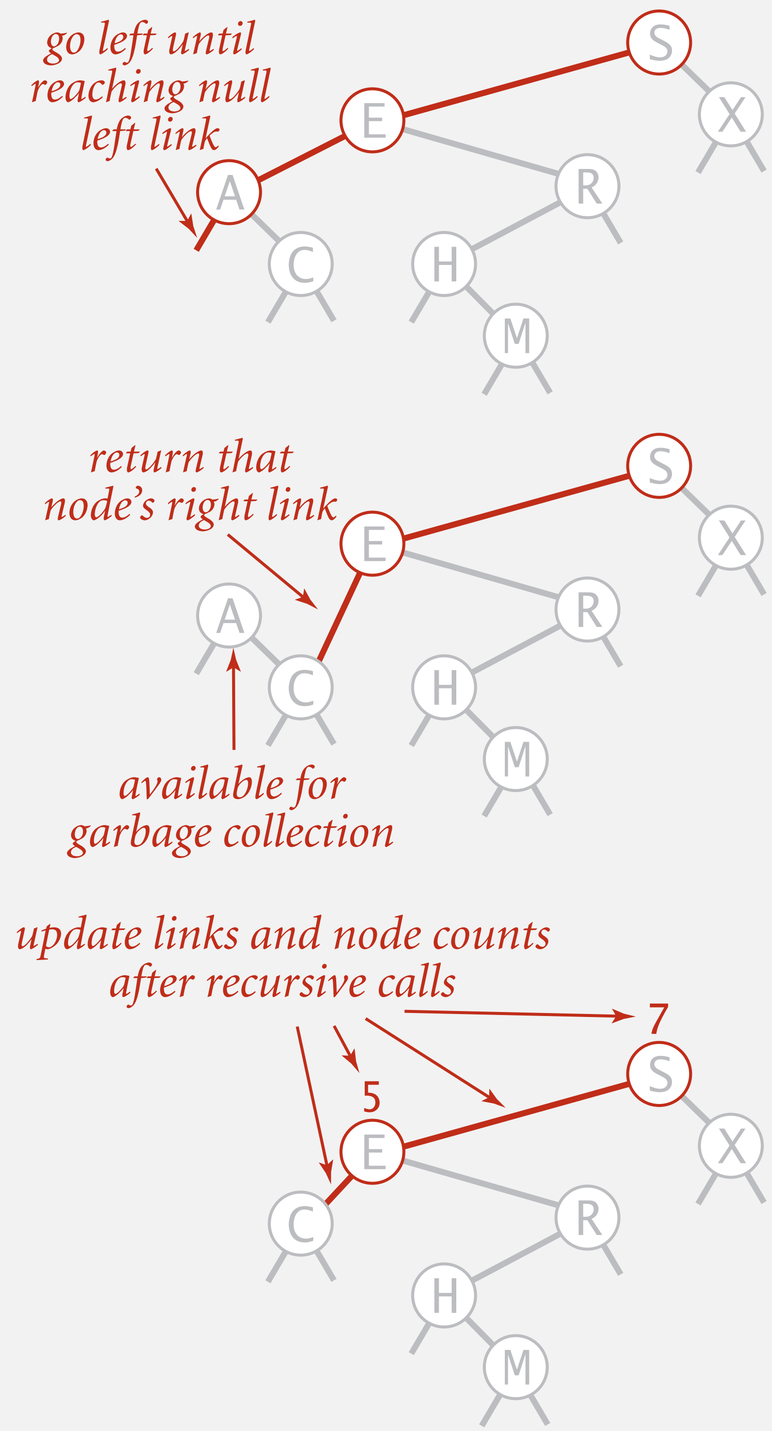
Cost. $\sim 2 \ln N'$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone (memory) overload.

Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

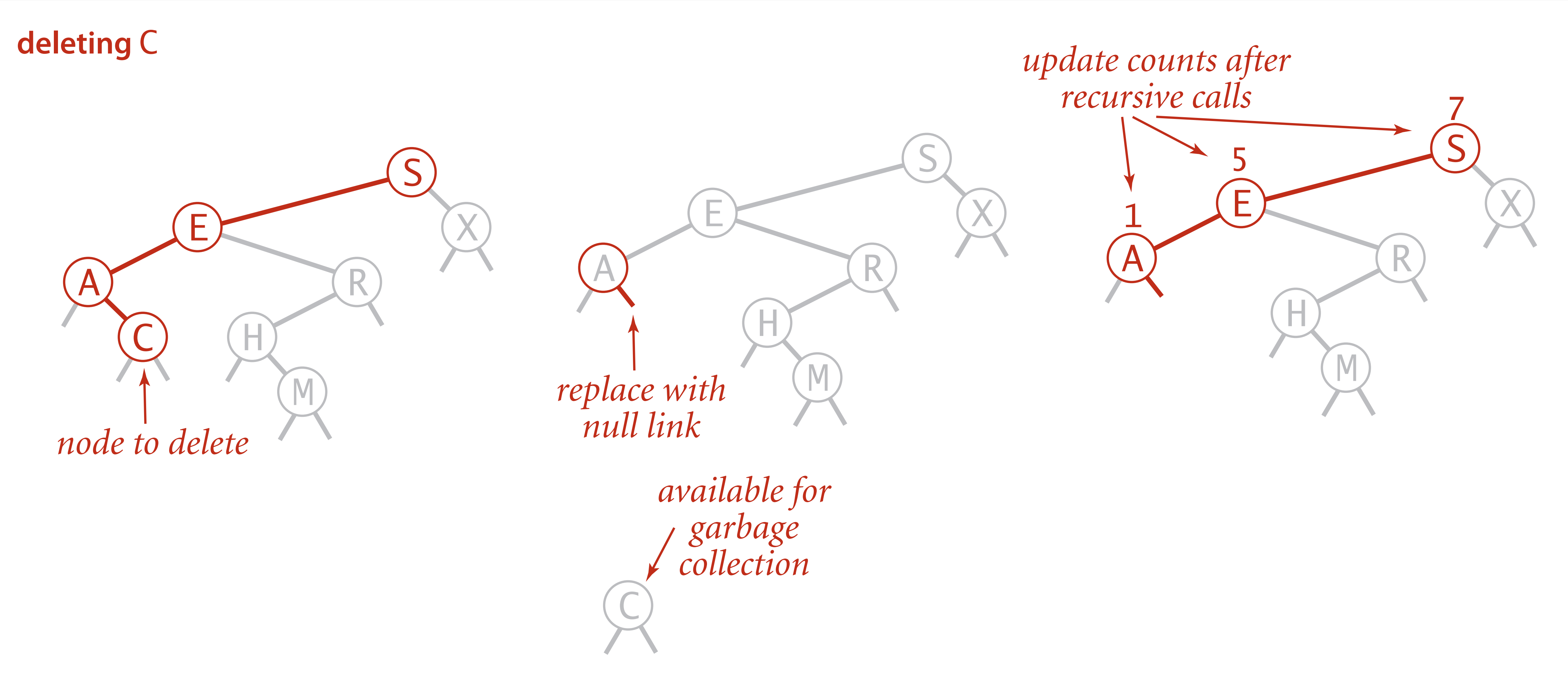



```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

Hibbard deletion

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

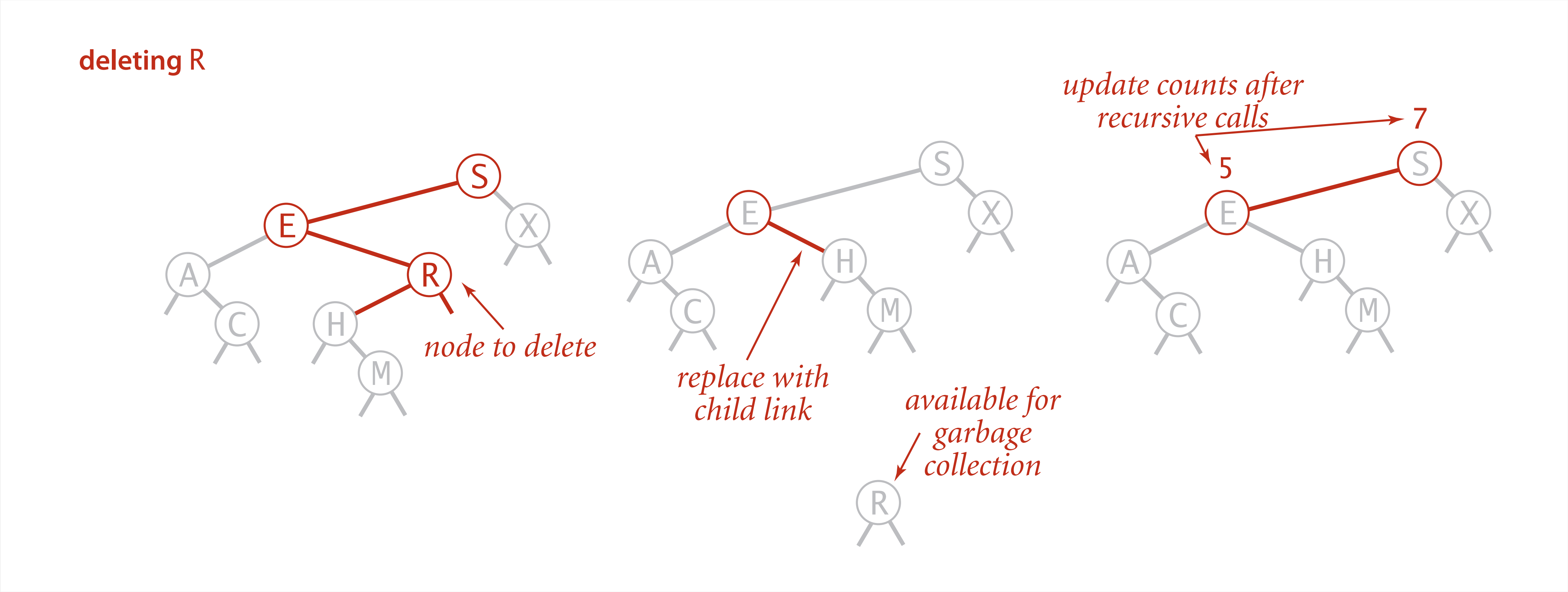
Case 0. [0 children] Delete `t` by setting parent link to null.



Hibbard deletion

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

Case 1. [1 child] Delete `t` by replacing parent link.

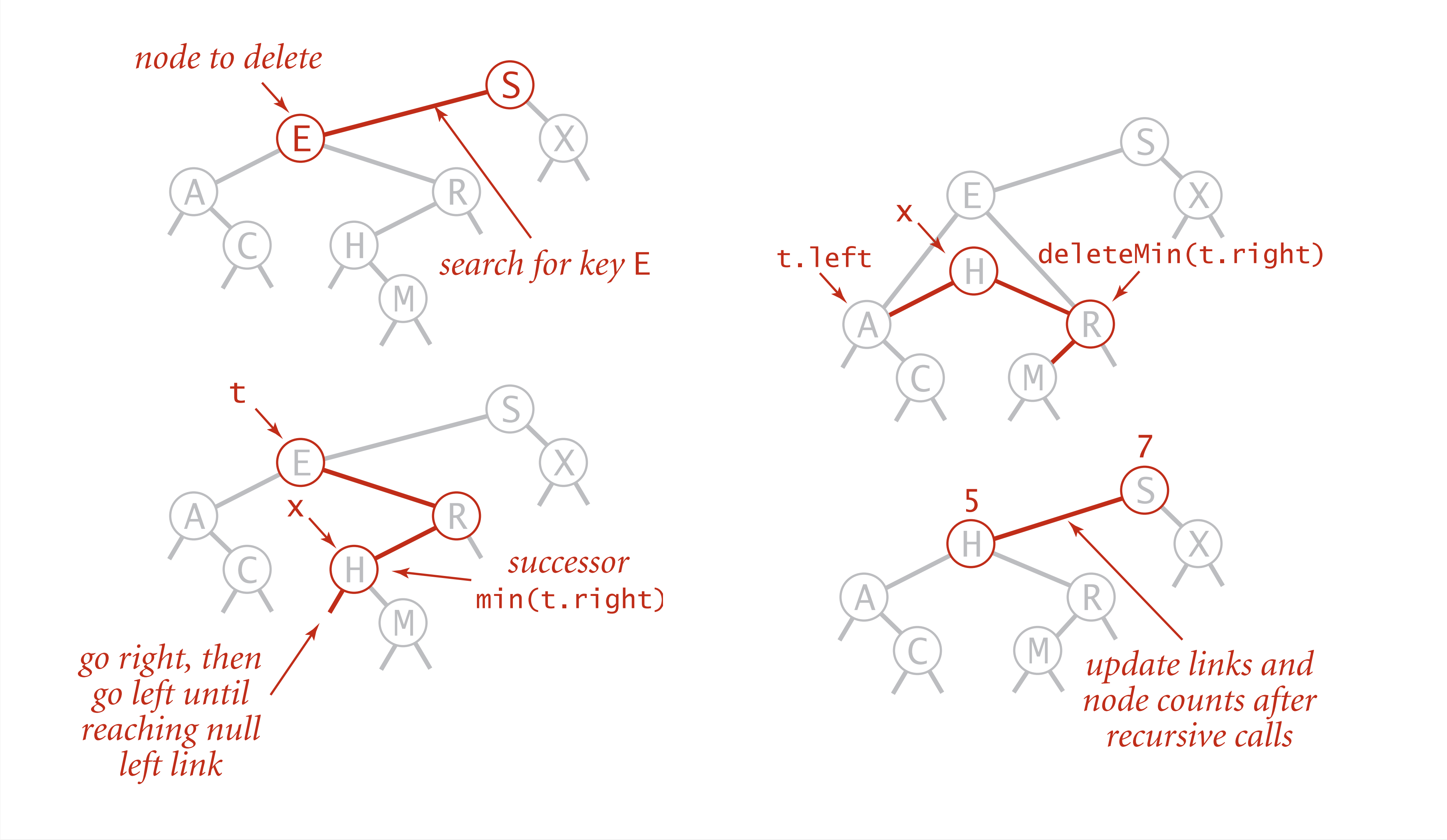


Hibbard deletion

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

Case 2. [2 children]

- Find successor `x` of `t`. ← x has no left child
- Delete the minimum in `t`'s right subtree. ← but don't garbage collect x
- Put `x` in `t`'s spot. ← still a BST



Hibbard deletion: Java implementation

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

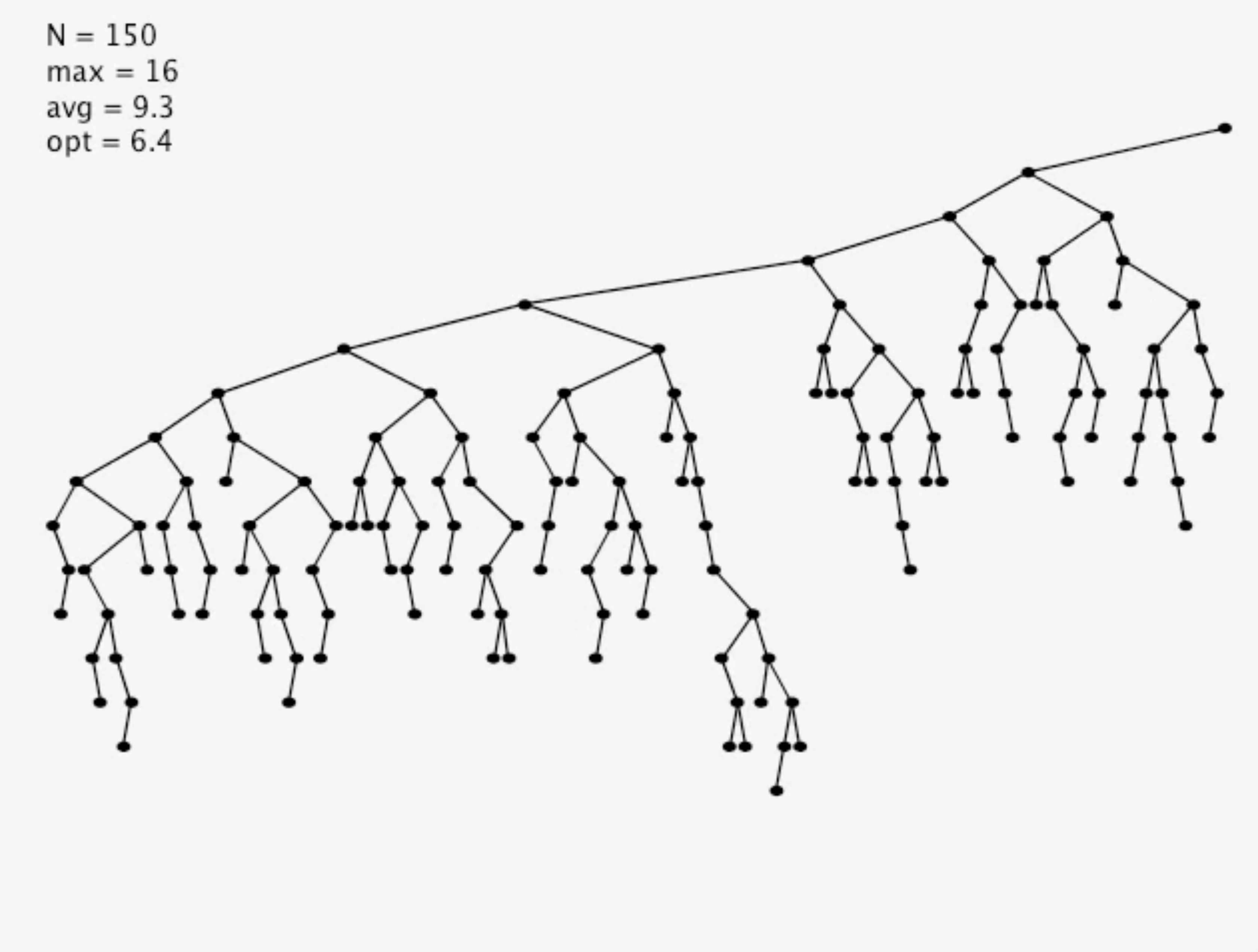
no left child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



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

Longstanding open problem. Simple and efficient delete for BSTs.

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()`

other operations also become √N
if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.