

INTRODUCTION TO DATA STRUCTURES
and
ALGORITHMS

Rutgers University

BINARY SEARCH TREES

- *Symbol Table*
- *BSTs*
- *Ordered operations*
- *Iteration*
- *Deletion*



BINARY SEARCH TREES

- *Symbol Table*
- *BSTs*
- *Ordered operations*
- *Iteration*
- *Deletion*

Motivation – Symbol Table

LO 5.1

Telephone books are obsolete

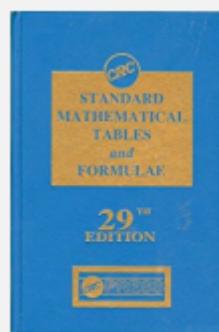


Unsupported Operations

- Add a new name and associated number
- Remove a given name and associated number
- Change the number associated with a given name

key = name
value = phone number

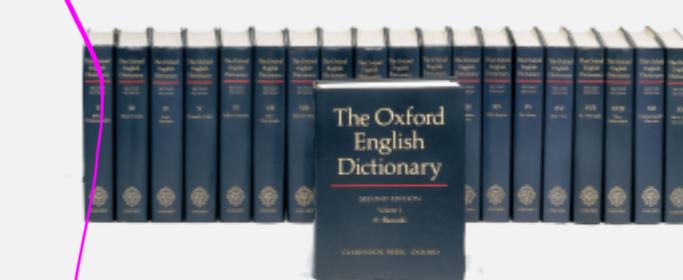
Idea. Store data as (key, value) pairs



key = math function and input
value = function output



key = term
value = article



key = word
value = definition

A Symbol Table API

LO 5.1

two generic type parameters
public class ST<Key extends Comparable<Key>, Value>

ST()

create an empty symbol table

void put(Key key, Value val)

insert key-value pair

← a[key] = val;

Value get(Key key)

value paired with key

← a[key]

boolean contains(Key key)

is there a value paired with key?

Iterable<Key> keys()

all the keys in the symbol table

void delete(Key key)

remove key (and associated value)

boolean isEmpty()

is the symbol table empty?

int size()

number of key-value pairs

An **API** (Application Programming Interface) defines what kind of operations are available to client programs.

Goal – Design a Symbol Table

LO 5.1

Design a symbol table that can support ordered operations

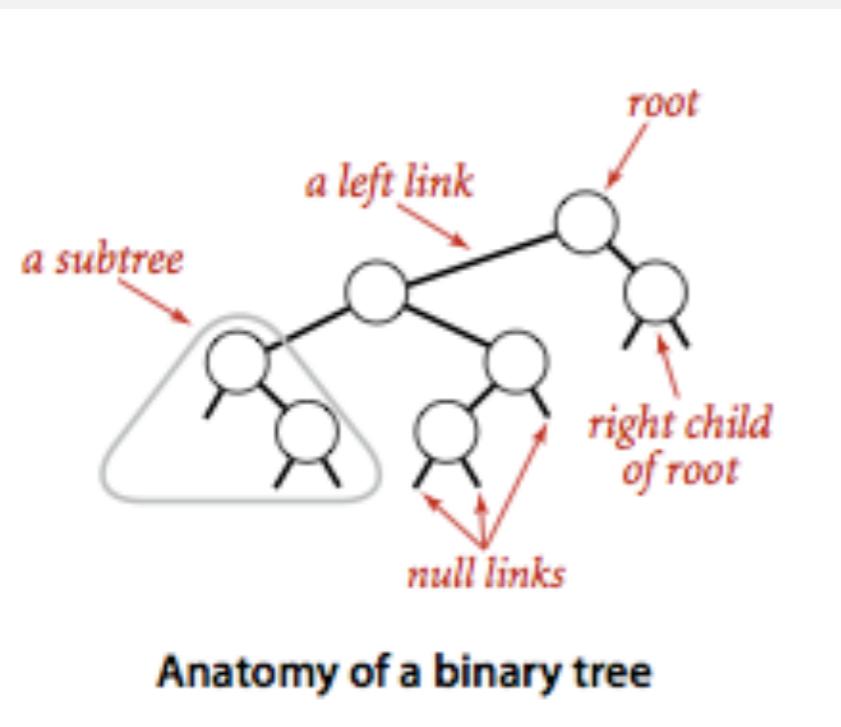
	keys	values
$\min()$ →	9:00:00	Chicago
	9:00:03	Phoenix
	9:00:13	Houston ← get(9:00:13)
	9:00:59	Chicago
	9:01:10	Houston
$\text{floor}(9:05:00)$ →	9:03:13	Chicago
	9:10:11	Seattle
$\text{select}(7)$ →	9:10:25	Seattle
$\text{rank}(9:10:25) = 7$	9:14:25	Phoenix
	9:19:32	Chicago
	9:19:46	Chicago
	9:21:05	Chicago
	9:22:43	Seattle
	9:22:54	Seattle
	9:25:52	Chicago
$\text{ceiling}(9:30:00)$ →	9:35:21	Chicago
	9:36:14	Seattle
$\max()$ →	9:37:44	Phoenix

Ordered Symbol Table API

LO 5.1

public class ST<Key extends Comparable<Key>, Value>	
:	
Key min()	<i>smallest key</i>
Key max()	<i>largest key</i>
Key floor(Key key)	<i>largest key less than or equal to key</i>
Key ceiling(Key key)	<i>smallest key greater than or equal to key</i>
int rank(Key key)	<i>number of keys less than key</i>
Key select(int k)	<i>key of rank k</i>
:	

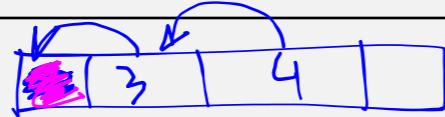
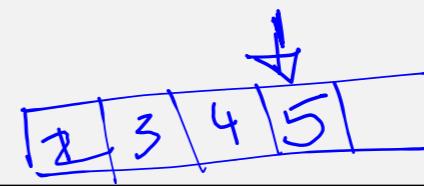
Goal. Implement ordered symbol using appropriate data structure that can support ordered operations, **efficiently**



BINARY SEARCH TREES

- *Symbol Table*
- ***BSTs***
- *Ordered operations*
- *Iteration*
- *Deletion*

Binary Search Tree (BST) Motivation



Data Structure	Insert		Delete		Search		
	Best	Worst	Best	Worst	Best	Worst	
Unsorted	Array	<u>O(1)</u>	<u>O(1)</u>	<u>O(1)</u>	<u>O(n)</u>	<u>O(1)</u>	<u>O(n)</u>
	Linked List	<u>O(1)</u>	<u>O(1)</u>	<u>O(1)</u>	<u>O(n)</u>	<u>O(1)</u>	<u>O(n)</u>
Sorted	Array	<u>O(n)</u>	<u>O(n)</u>	<u>O(1)</u>	<u>O(n)</u>	<u>O(1)</u>	<u>O(log n)</u>
	Linked List	<u>O(1)</u>	<u>O(n)</u>	<u>O(1)</u>	<u>O(n)</u>	<u>O(1)</u>	<u>O(n)</u>

Only if array is kept sparse (with empty cells)

$$\lceil \frac{m+1}{2} \rceil \quad \lfloor \frac{m}{2} \rfloor$$

$\hookrightarrow a \rightarrow b \rightarrow c \rightarrow d$

$\hookrightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g$

$\lceil \frac{m+3}{2} \rceil \quad \lfloor \frac{m}{2} \rfloor$

BST is a dynamic structure that combines the *flexibility of insertion in a linked structure* with the *efficiency of search in an ordered array*.

Binary search tree (BST)

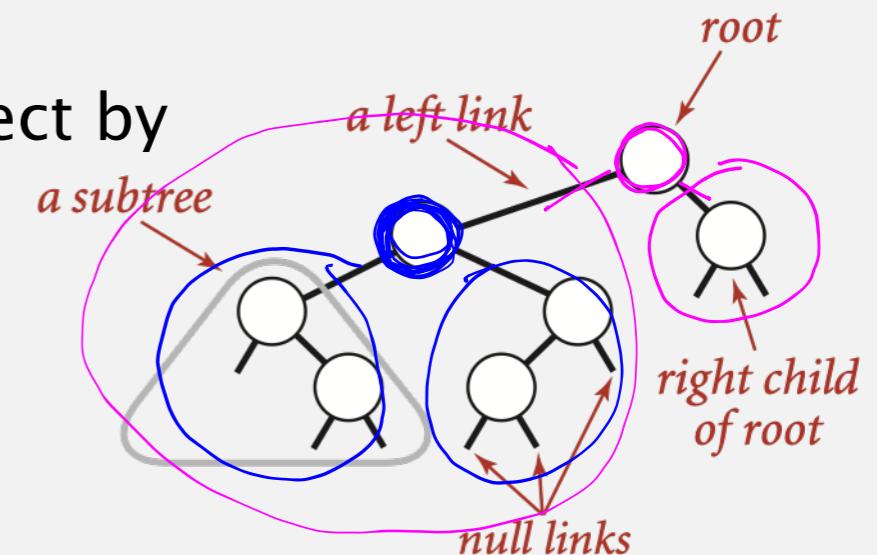
Definition. A BST is a binary tree in symmetric order.

sorted



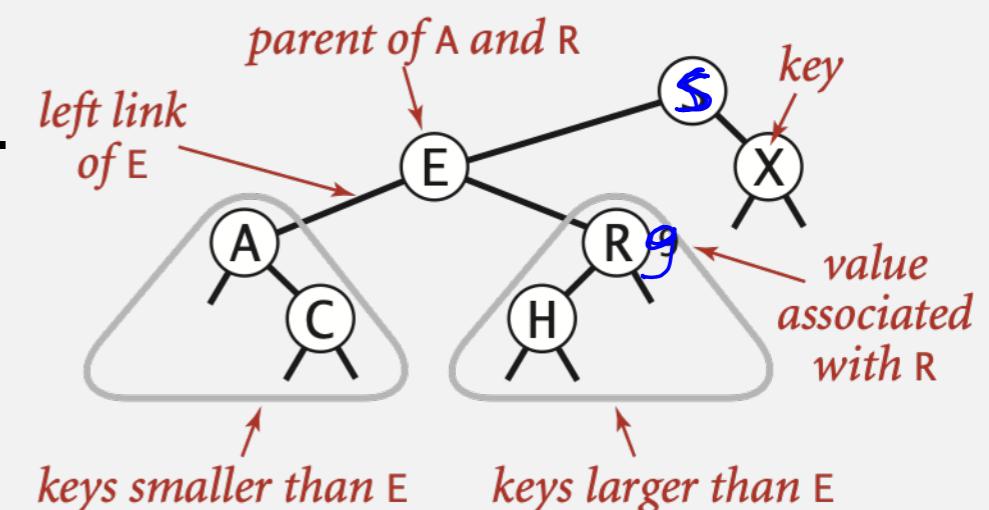
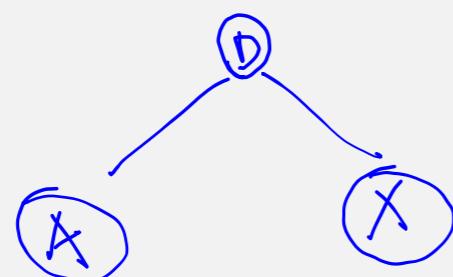
A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right) connect by a root node.



Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.

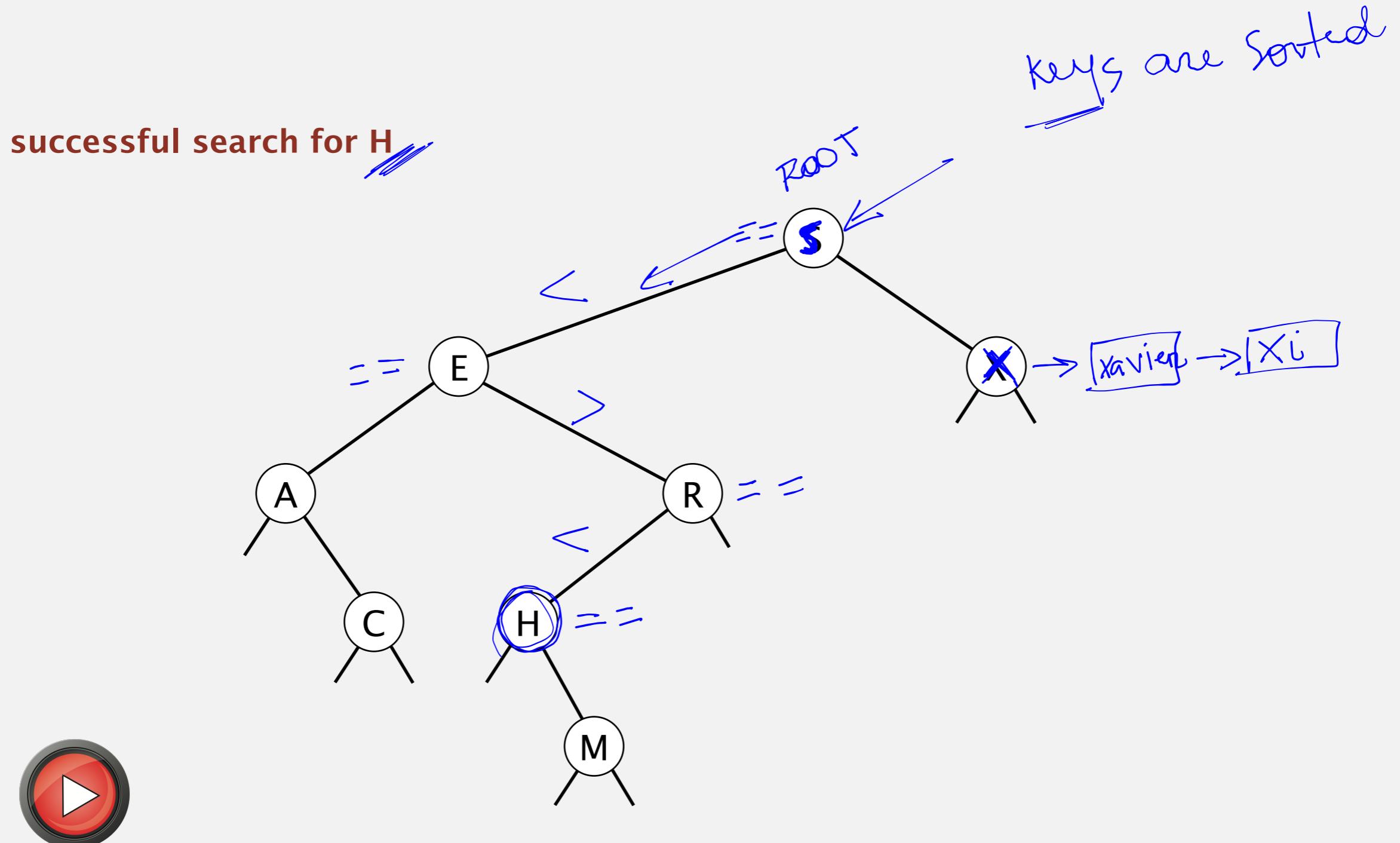


Binary search tree demo

LO 5.3

Search.

Start at the root; If less, go left; if greater, go right; if equal, search hit.



Binary search tree demo

LO 5.3

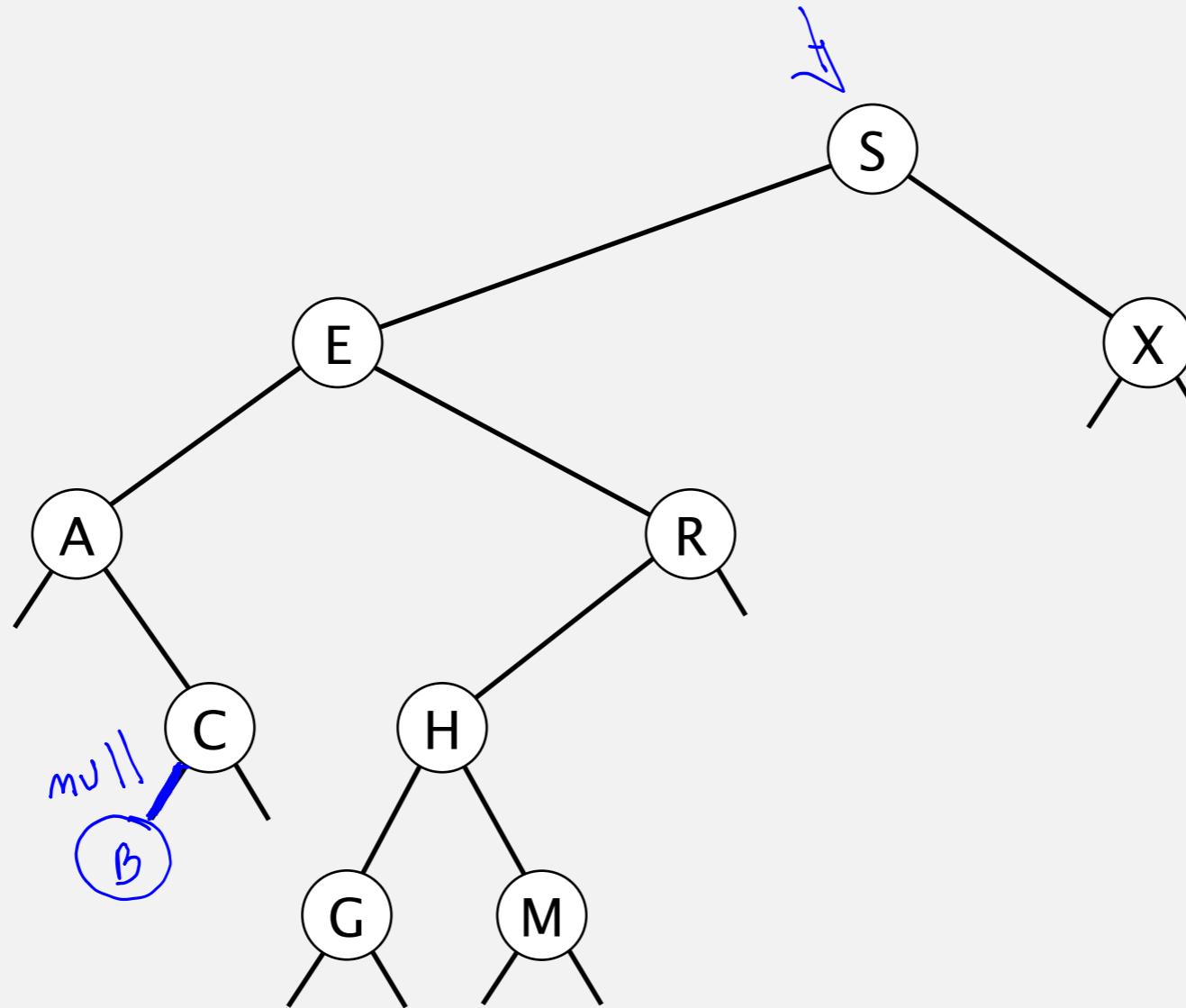
Insert.

Start at the root; If less, go left; if greater, go right; if null, insert.

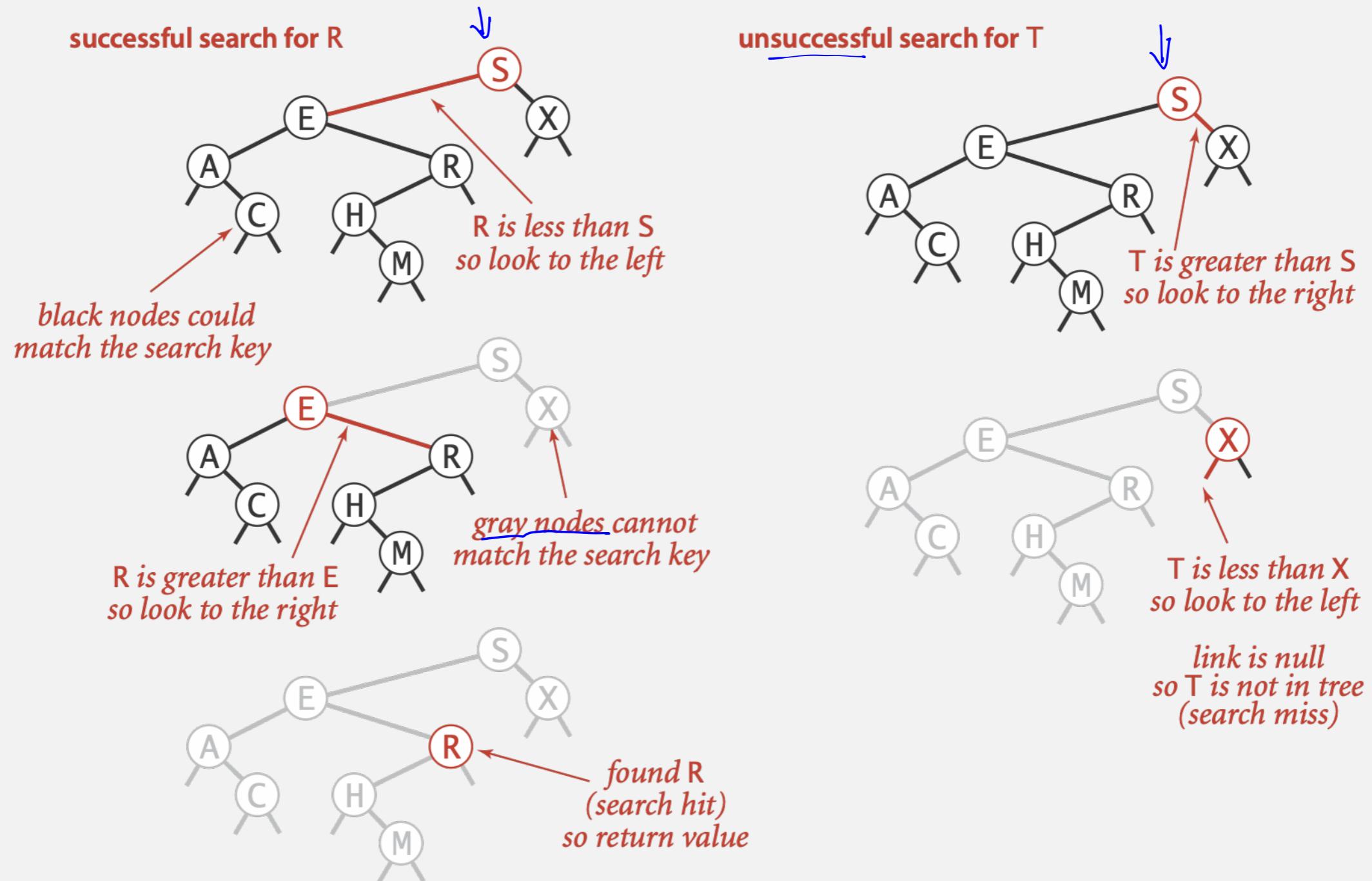
Intuition: search until it fails, insert at failure point.

insert G

insert B



Get. Return value corresponding to given key, or null if no such key.



BST representation in Java

LO 5.3

Java definition. A BST is a reference to a root Node.

A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree.

smaller keys larger keys

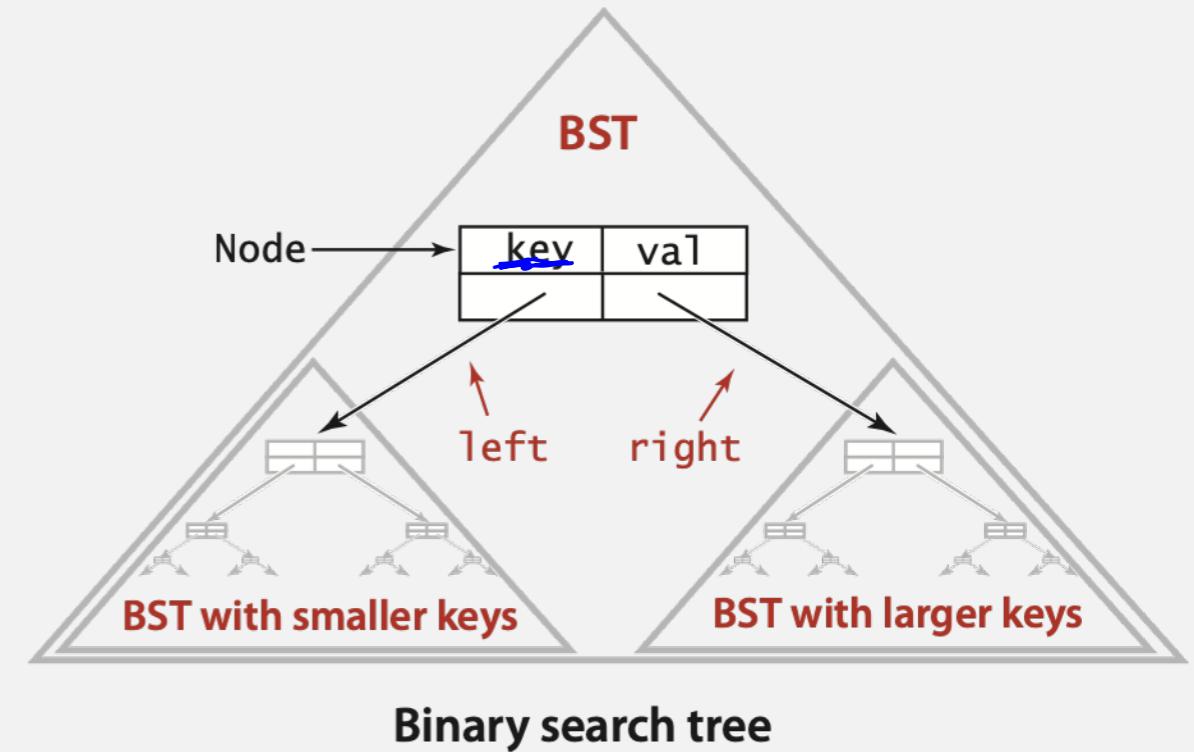
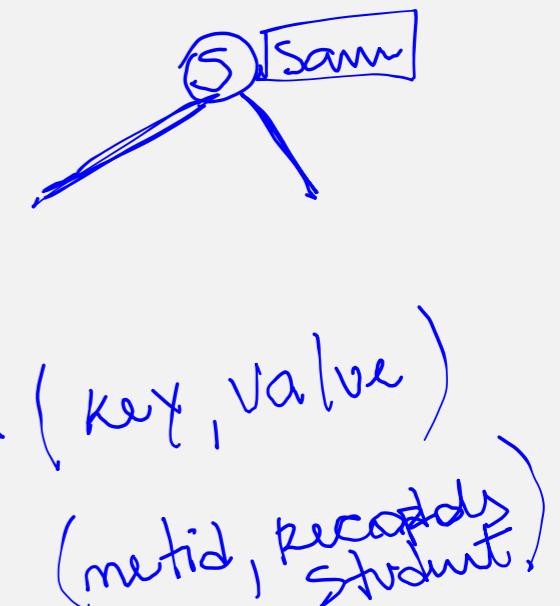
```
private class Node<Key, Value>
```

```
{  
    private Key key;  
    private Value val;  
    private Node left, right;
```

```
public Node(Key key, Value val)
```

```
{  
    this.key = key;  
    this.val = val;  
}
```

```
}      Key and Value are generic types; Key is Comparable
```



BST implementation (skeleton)

LO 5.3

Key class must implement compareTo()

```
public class BST<Key extends Comparable<Key>, Value>
```

```
{
```

```
    private Node root;
```

← root of BST

```
    private class Node
```

```
    { /* see previous slide */ }
```

```
    public void put(Key key, Value val)
```

```
    { /* see next slide */ }
```

```
    public Value get(Key key)
```

```
    { /* see next slide */ }
```

```
    public Iterable<Key> keys()
```

```
    { /* see slides in next section */ }
```

```
    public void delete(Key key)
```

```
    { /* see textbook */ }
```

```
}
```

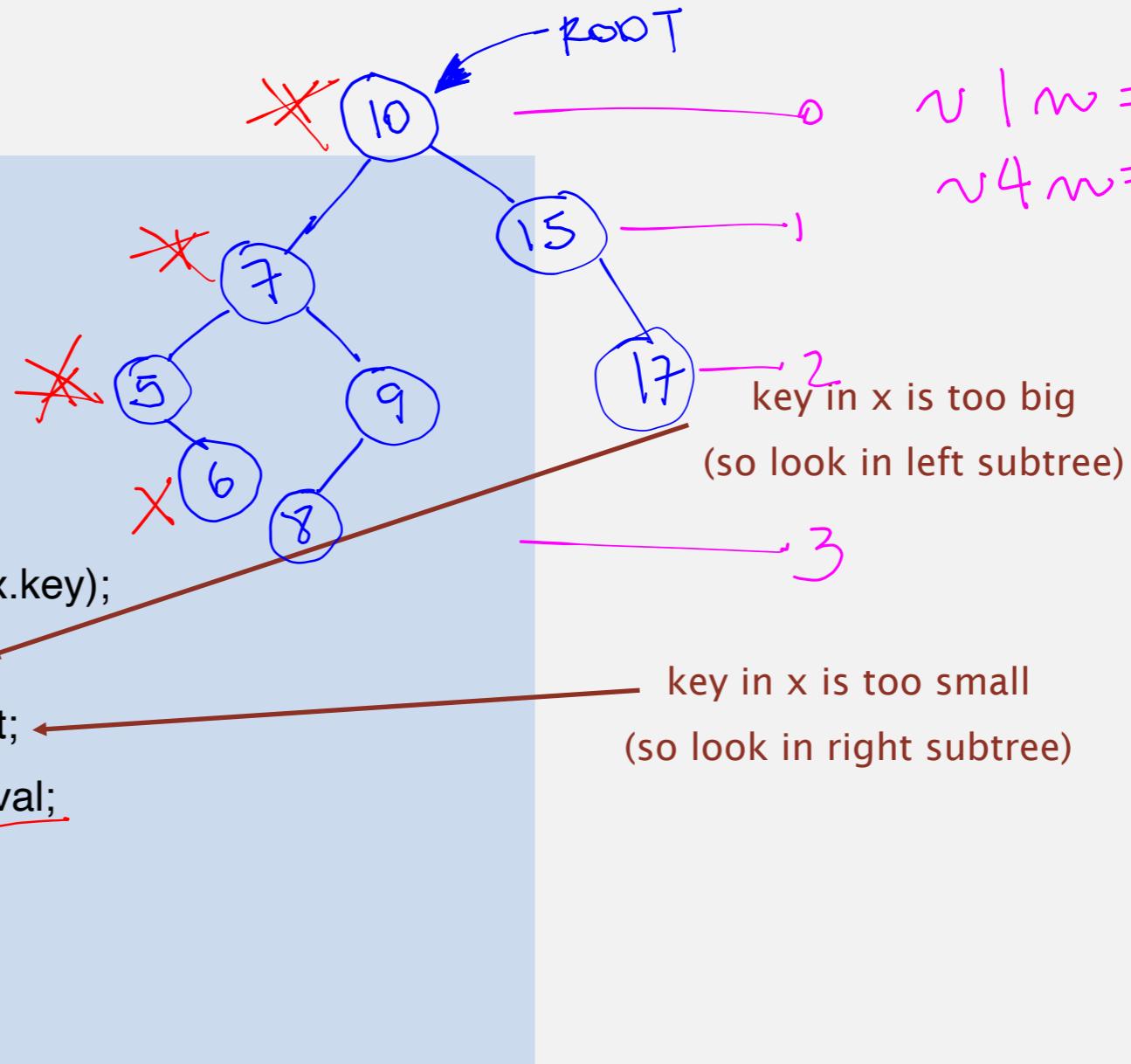
BST search: Java implementation

LO 5.3

Get. Return value corresponding to given key, or null if no such key.

key = 6

```
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;
}
```



$\sim \ln n \Rightarrow O(n)$
 $\sim 4 \ln n \Rightarrow O(n)$

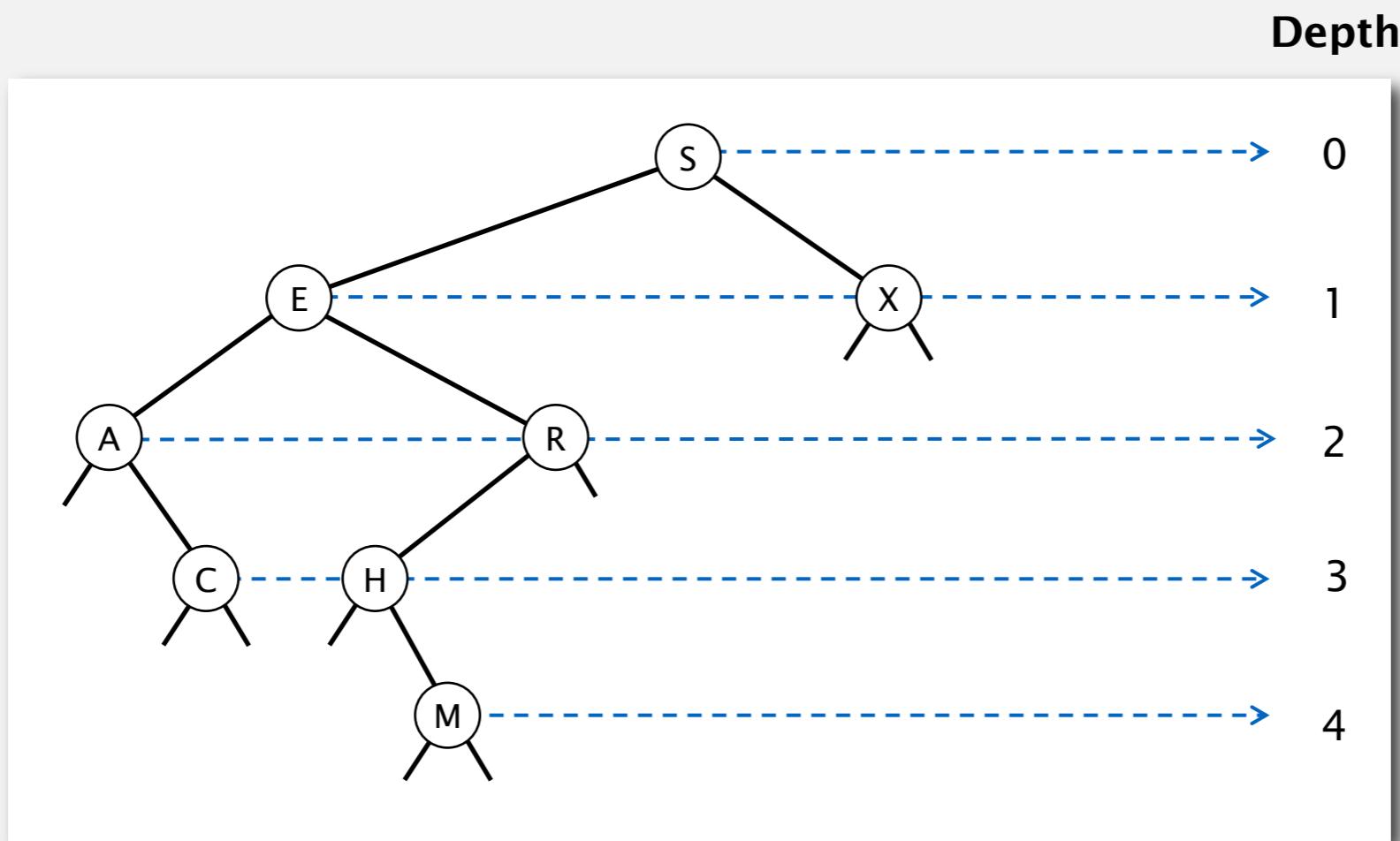
Cost. Number of compares = 1 + depth of node.

= 4 key = 6

Depth of a node in a tree and the height of a tree

The depth of a node is the number of links on the path from it to the root.

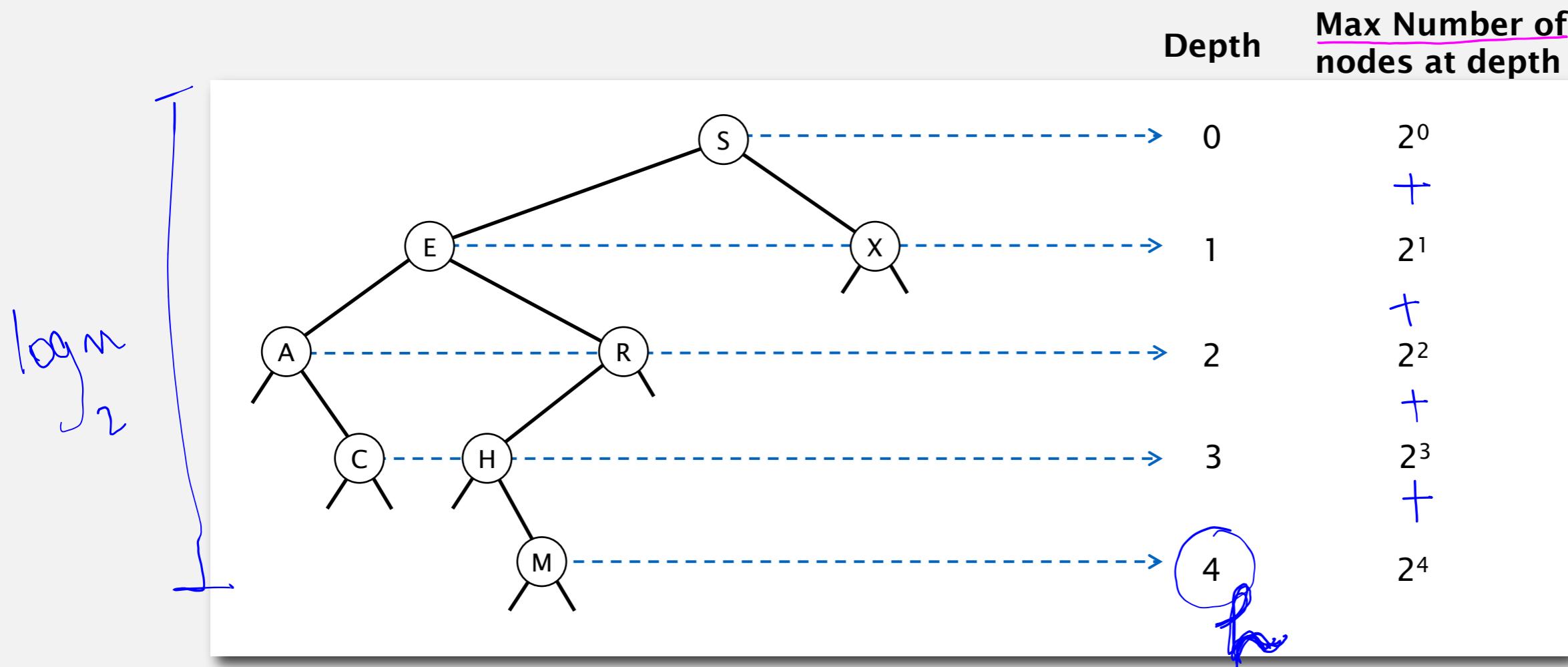
Root's depth is 0



The height of a tree is the maximum depth among its nodes.

Height of a tree

The height (h) of a tree is the maximum depth among its nodes.



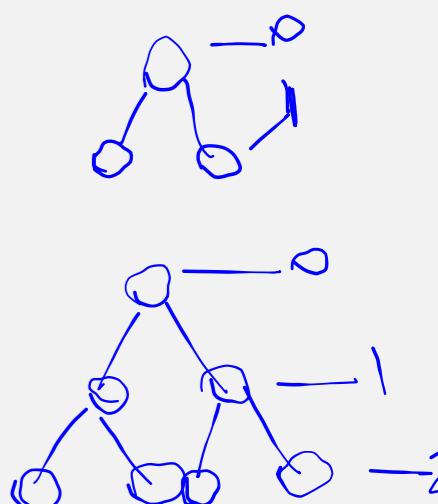
The last level, deepest depth, may have 2^h nodes

Let N_{max} be the maximum number of nodes in a tree of height h

$$N_{max} = 2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{h+1} - 1$$

$$N_{max} + 1 = 2^{h+1}$$

$$h = \log(N_{max} + 1) - 1 \rightarrow O(\log N_{max})$$



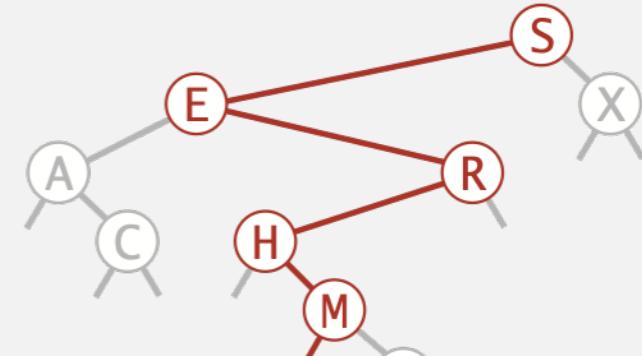
BST insert

Put. Associate value with key.

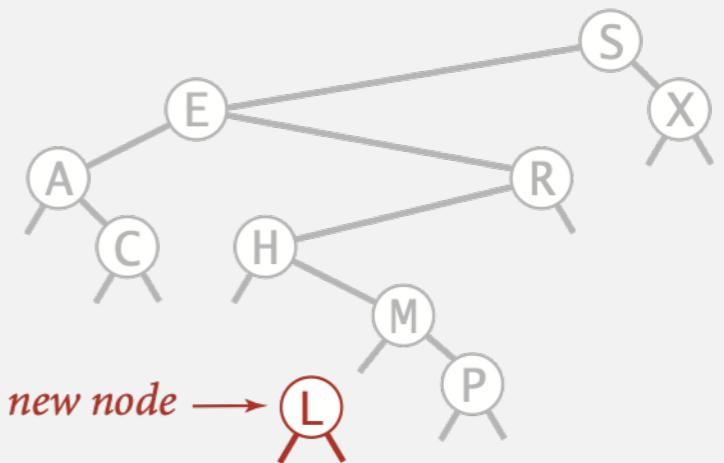
Search for key, then two cases:

- Key in tree \Rightarrow reset value.
- Key not in tree \Rightarrow add new node.

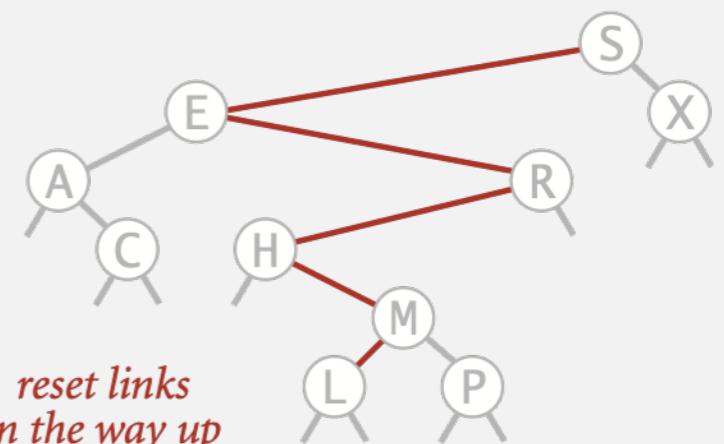
inserting L



*search for L ends
at this null link*



create new node → L



*reset links
on the way up*

Insertion into a BST

BST insert: Java implementation

LO 5.3

Put. Associate value with key.

put(9, record)

```
public void put (Key key, Value val)
{   root = put (root, key, val); }

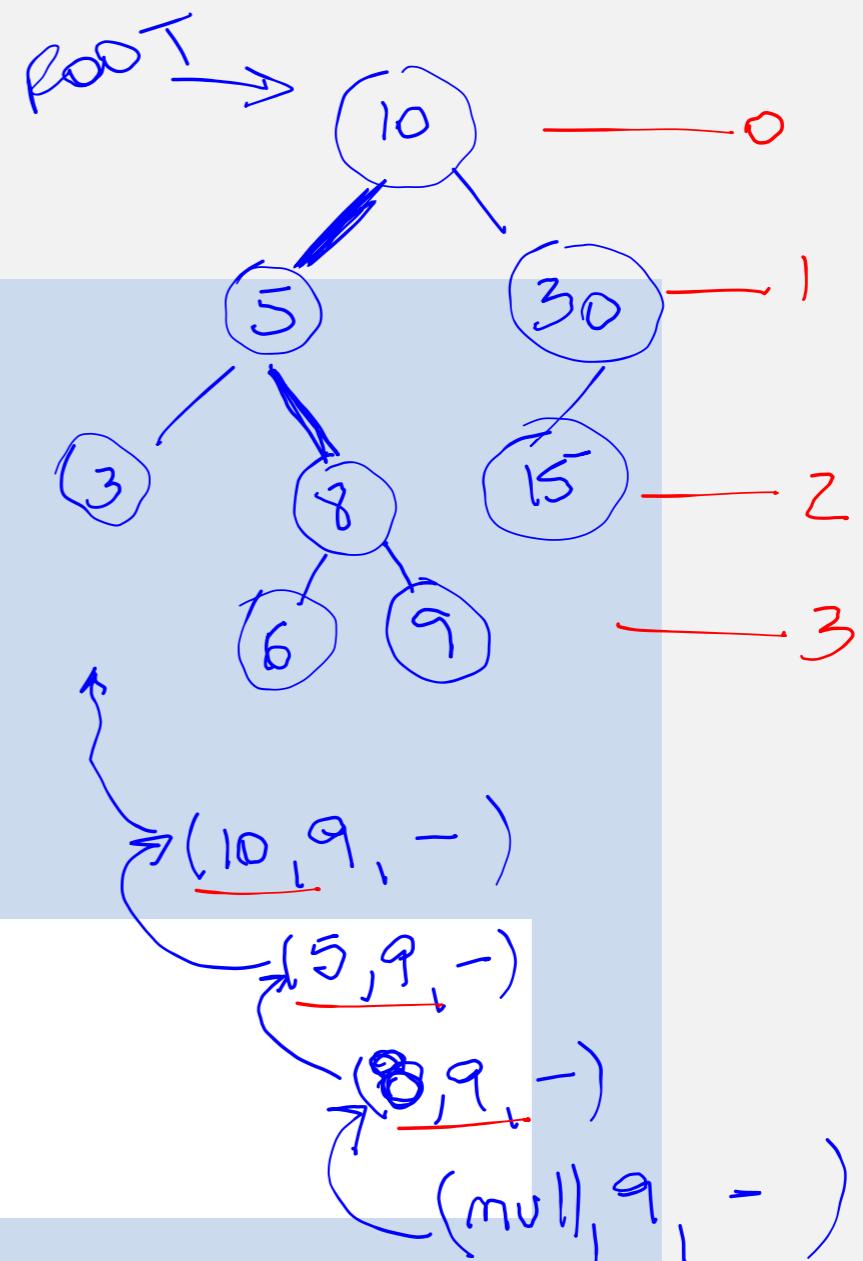
private Node put (Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);

    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;

    return x;
}
```

Count
this line

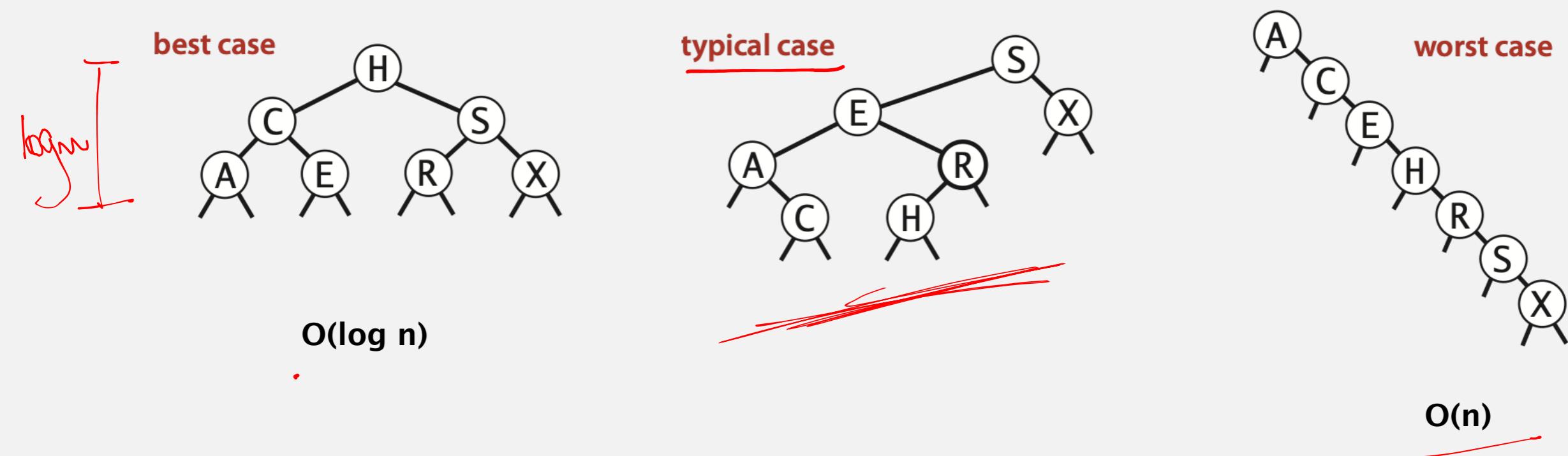
⚠ Warning: concise but tricky code; read carefully!



Cost. Number of compares = 1 + depth of node. Count compare To

Tree shape

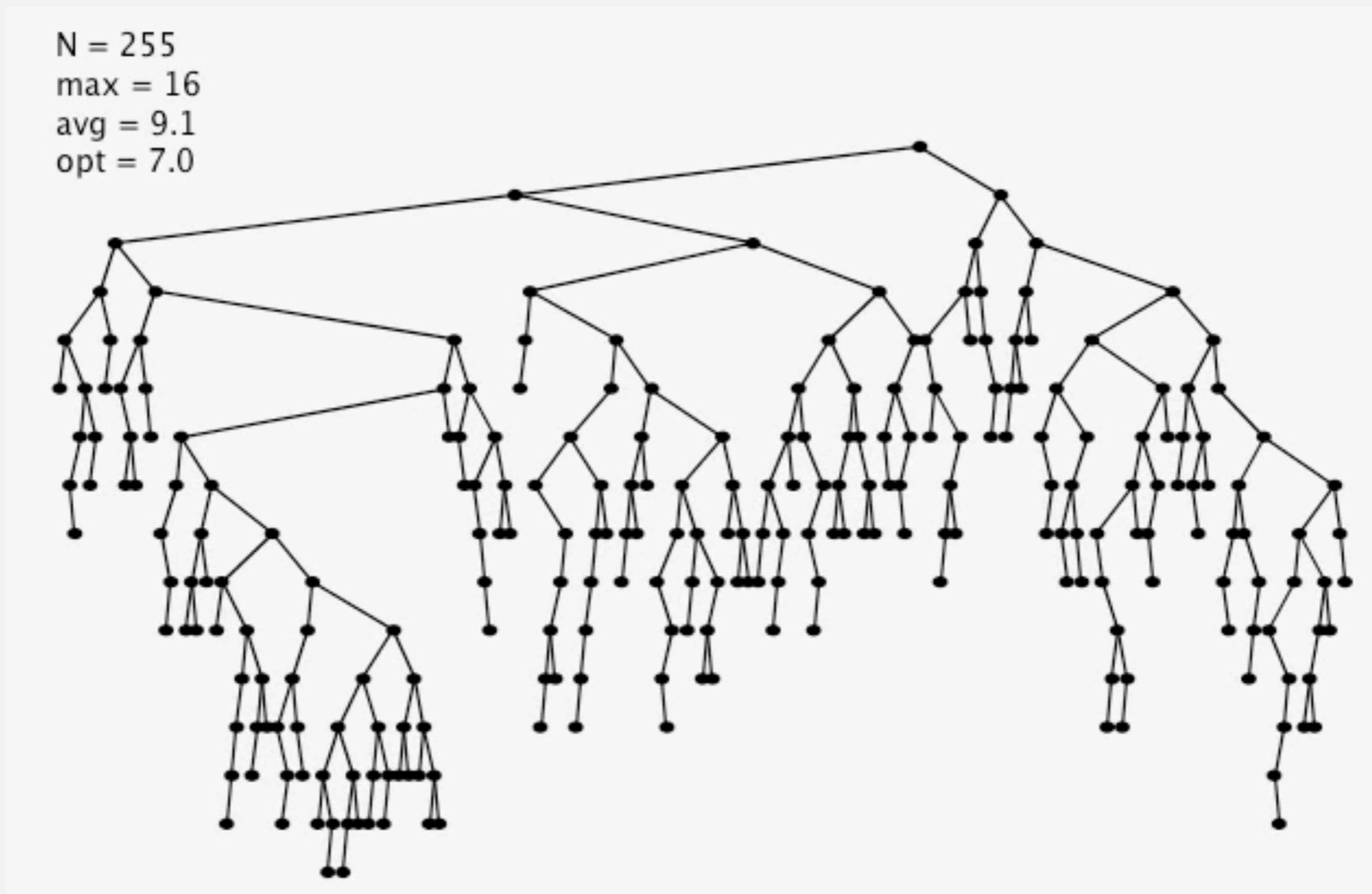
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert = 1 + depth of node.



Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

Ex. Insert keys in random order.

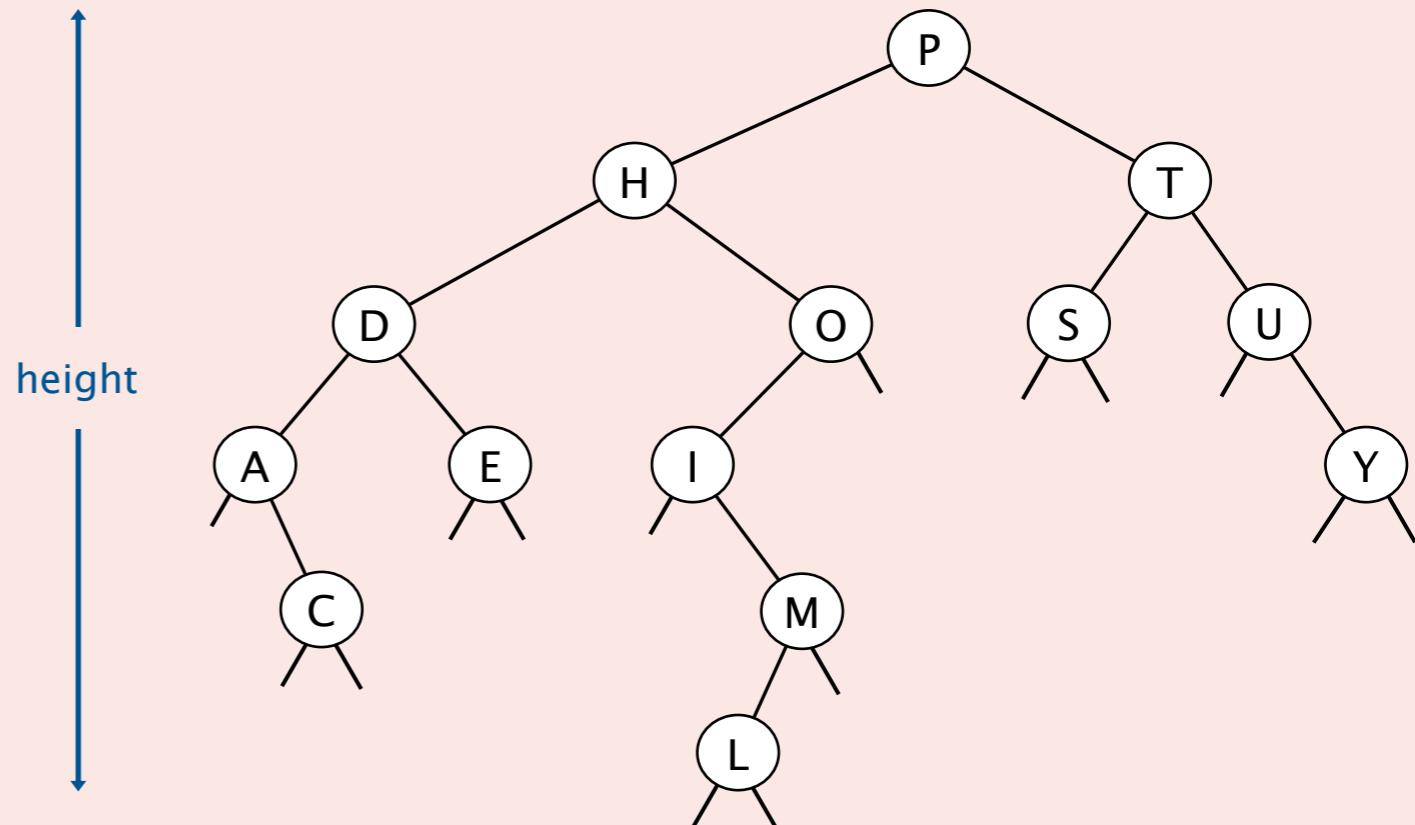


Binary search trees: quiz 1

Suppose that you insert n keys in random order into a BST.

What is the expected height of the resulting BST?

- A. $\sim \lg n$
- B. $\sim \ln n$
- C. $\sim 2 \lg n$
- D. $\sim 2 \ln n$
- E. $\sim 4.31107 \ln n$



BSTs: mathematical analysis

Proposition. If n distinct keys are inserted into a BST in **random** order, the expected number of compares for a search/insert is $\sim 2 \ln n$.

Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If n distinct keys are inserted into a BST in random order, the expected height is $\sim 4.311 \ln n$.

expected depth of
function-call stack in quicksort

How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

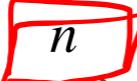
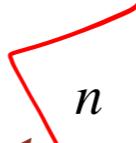
ABSTRACT

Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\dots$ and $\beta = 1.95\dots$ such that $E(H_n) = \alpha \log n - \beta \log \log n + O(1)$. We also show that $\text{Var}(H_n) = O(1)$.

But... Worst-case height is $n - 1$.

[exponentially small chance when keys are inserted in random order]

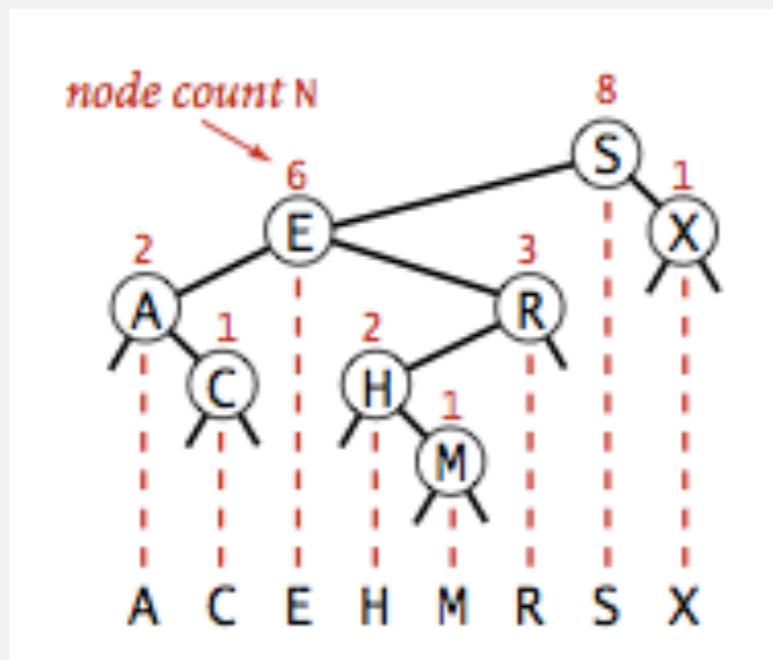
ST implementations: summary

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	n	n	n	n	<code>equals()</code>
binary search (ordered array)	$\log n$	n	$\log n$	n	<code>compareTo()</code>
BST					<code>compareTo()</code>

Why not shuffle to ensure a (probabilistic) guarantee of $\log n$?

- 1) Cannot assume all data is available up front
- 2) $\sim 4.311 \ln n$ is not guaranteed because there is an assumption about the input distribution

BINARY SEARCH TREES

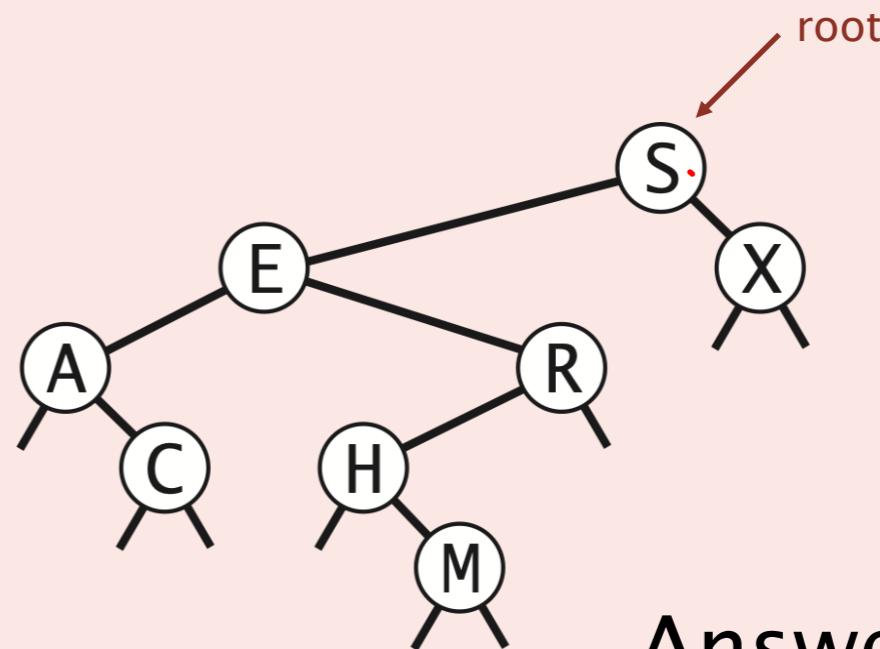


- *Symbol Table*
- *BSTs*
- *Iteration*
- *Ordered operations*
- *Deletion*

In which order does `traverse(root)` print the keys in the BST?

```
private void traverse(Node x)
{
    if (x == null) return;
    traverse(x.left);
    → StdOut.println(x.key);
    R → traverse(x.right);
}
```

- A. A C E H M R S X
- B. S E A C R H M X
- C. C A M H R E X S
- D. S E X A R C H M



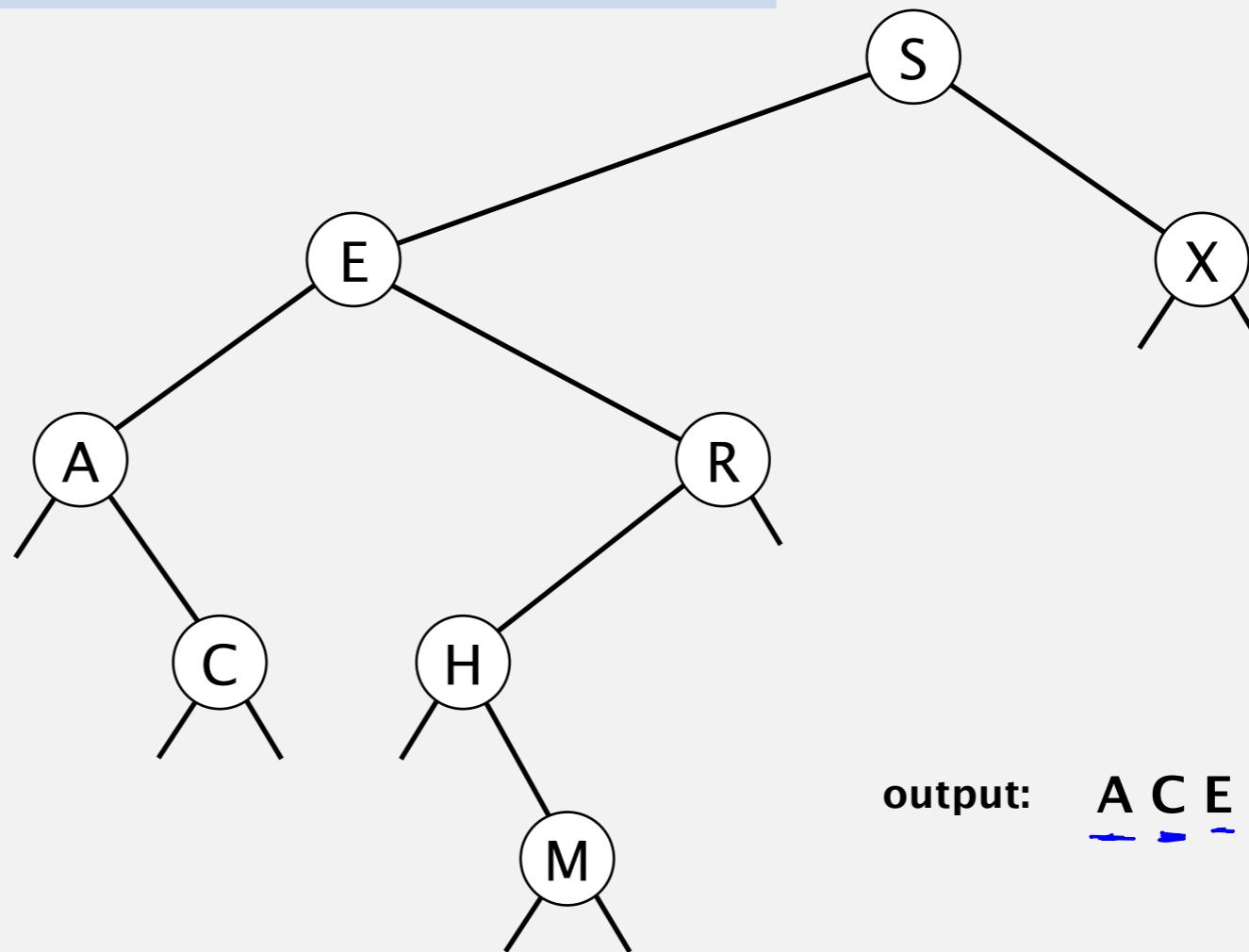
Answer: A

Inorder traversal

LO 5.8

- ① inorder(S)
- inorder(E)
- inorder(A)
- print A
- inorder(C)
- print C
- done C
- done A
- print E
- inorder(R)
- inorder(H)
- print H
- inorder(M)
- print M
- done M
- done H
- print R
- done R
- done E
- print S
- inorder(X)
- print X
- done X
- done S

```
inorder  
private void traverse(Node x)  
{  
    if (x == null) return; → Base Case  
    inorder traverse(x.left);  
    inorder StdOut.println(x.key);  
    inorder traverse(x.right);  
}
```



output: A C E H M R S X

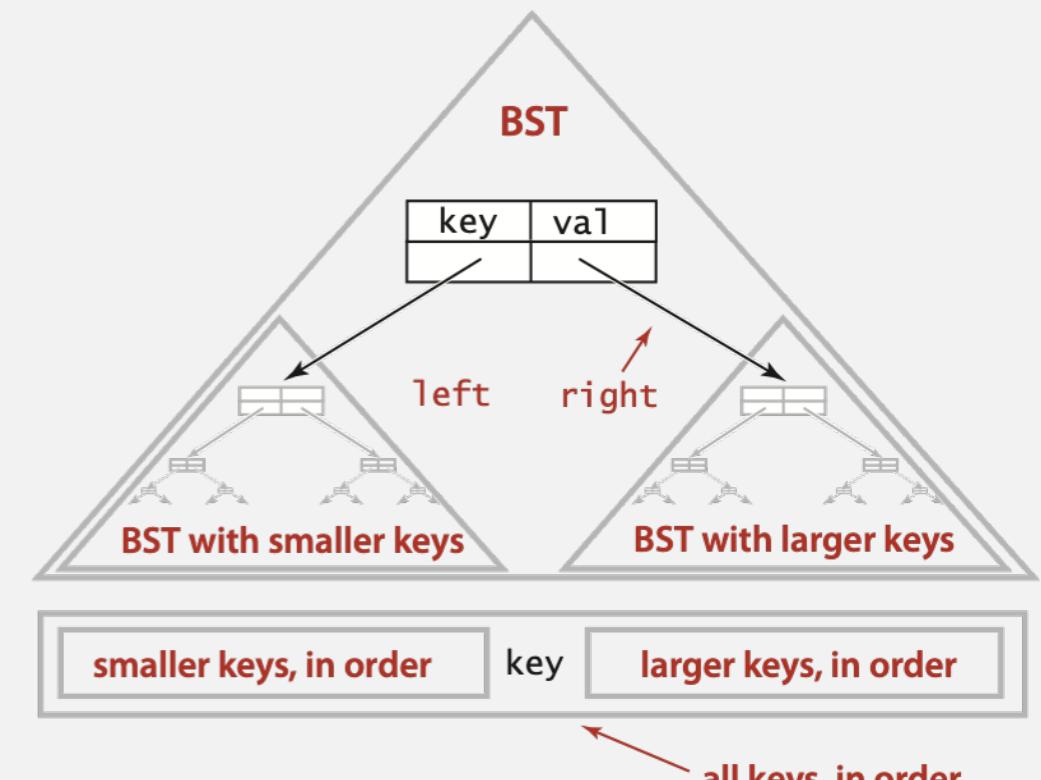
Inorder traversal

LO 5.8

1. Traverse left subtree.
2. Enqueue key.
3. Traverse right subtree.

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);      A C E H M R S X
    inorder(x.right, q);
}
```



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

Preorder traversal

LO 5.8

1. Enqueue key.
2. Traverse left subtree.
3. Traverse right subtree.

```
public Iterable<Key> keys()
{
```

```
    Queue<Key> q = new Queue<Key>();
```

```
    preorder(root, q);
```

```
    return q;
```

```
}
```

```
private void preorder(Node x, Queue<Key> q)
```

```
{
```

```
    if (x == null) return;
```

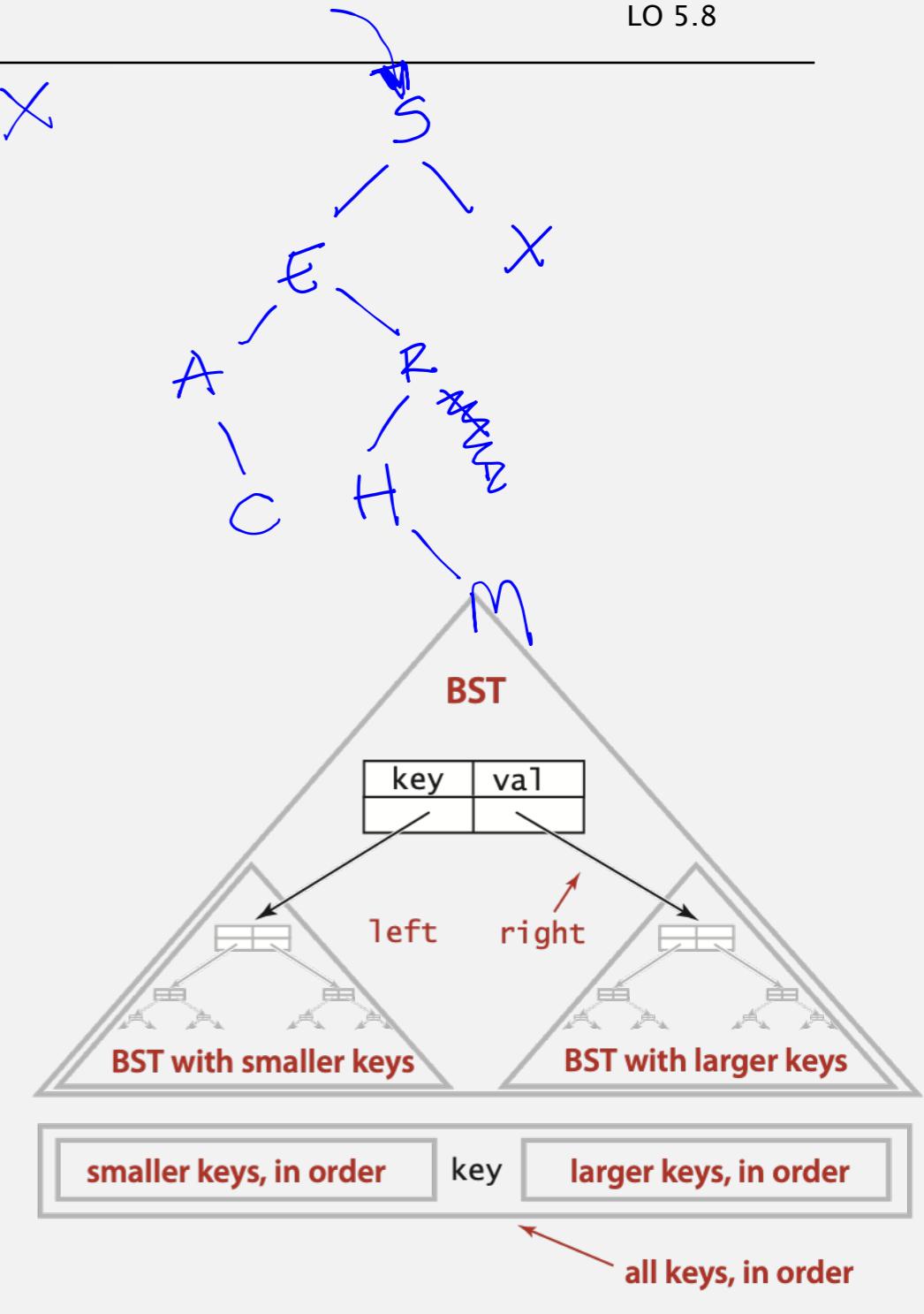
```
    q.enqueue(x.key);
```

```
    preorder(x.left, q);
```

```
    preorder(x.right, q);
```

```
}
```

S E A C & H M X



postorder traversal

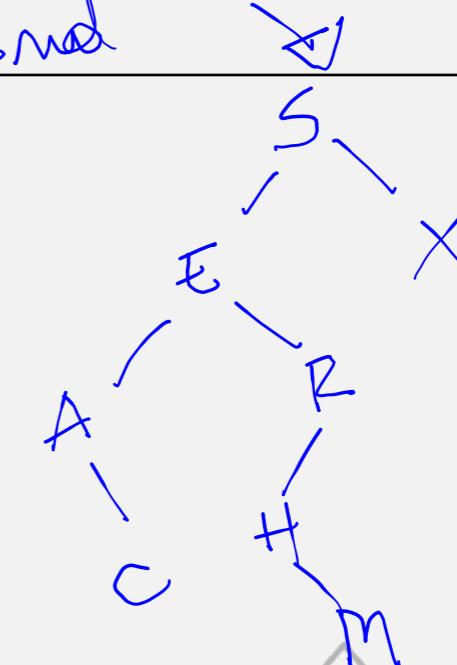
start

end

LO 5.8

1. Traverse left subtree.
2. Traverse right subtree.
3. Enqueue key.

C A M H R E X S
 $O(n)$



```
public Iterable<Key> keys()  
{
```

```
    Queue<Key> q = new Queue<Key>();  
    postorder(root, q);
```

```
    return q;
```

```
}
```

private void ~~postorder~~(Node x, Queue<Key> q)

```
{
```

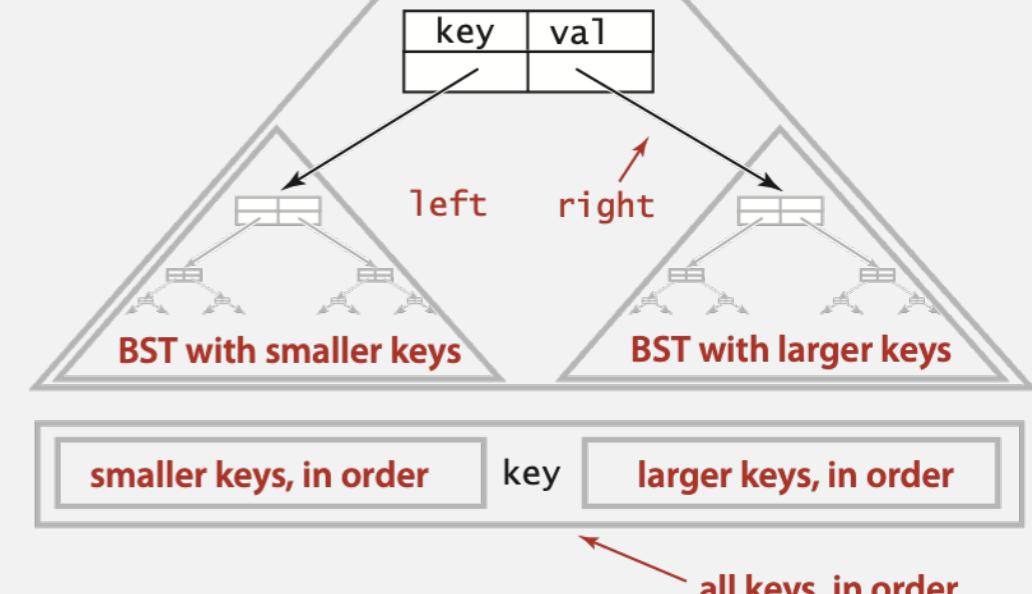
```
    if (x == null) return;
```

```
    postorder(x.left, q);
```

```
    postorder(x.right, q);
```

~~```
 q.enqueue(x.key);
```~~

```
}
```



# Running time

---

Property. Inorder traversal of a BST takes linear time  $O(n)$ .

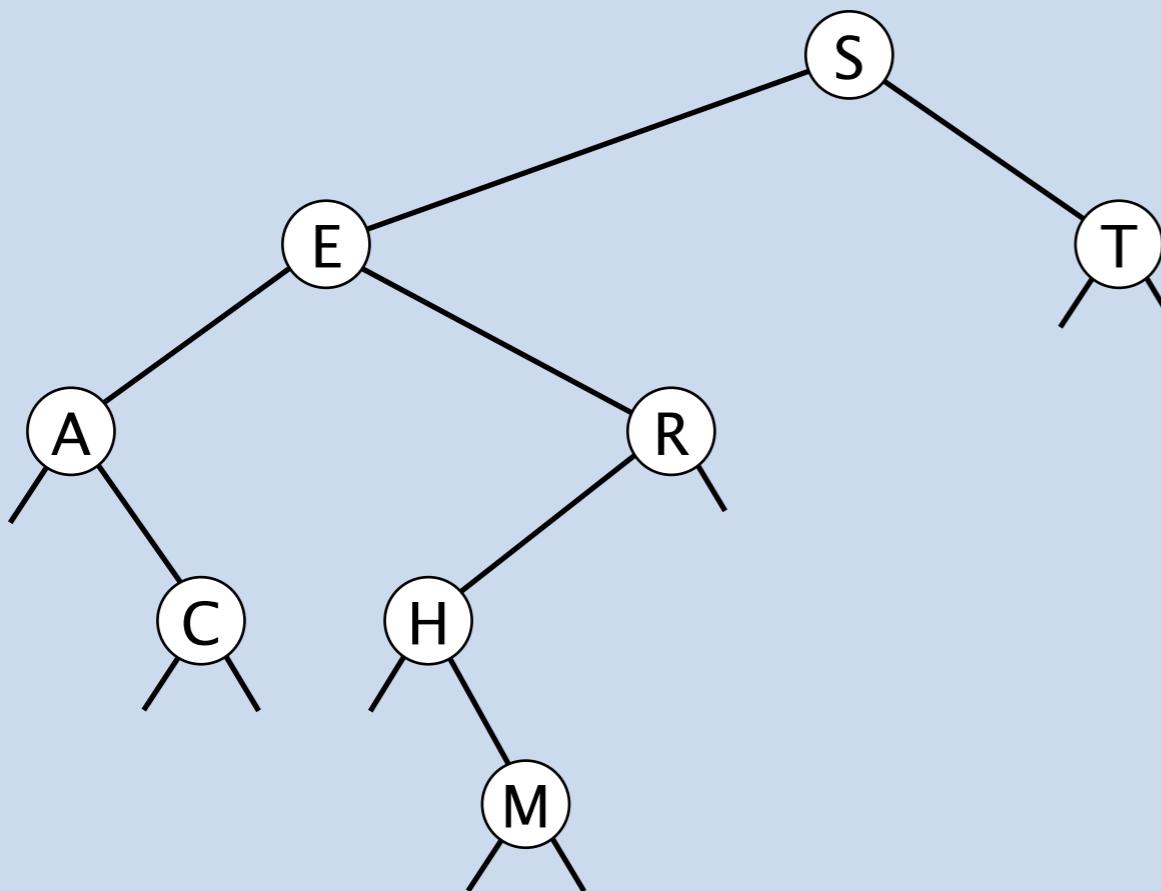


Silicon Valley, Season 4, Episode 5

# LEVEL-ORDER TRAVERSAL

Level-order traversal of a binary tree.

- Process root.
- Process children of root, from left to right.
- Process grandchildren of root, from left to right.
- ...



level-order traversal: **S E T A R C H M**

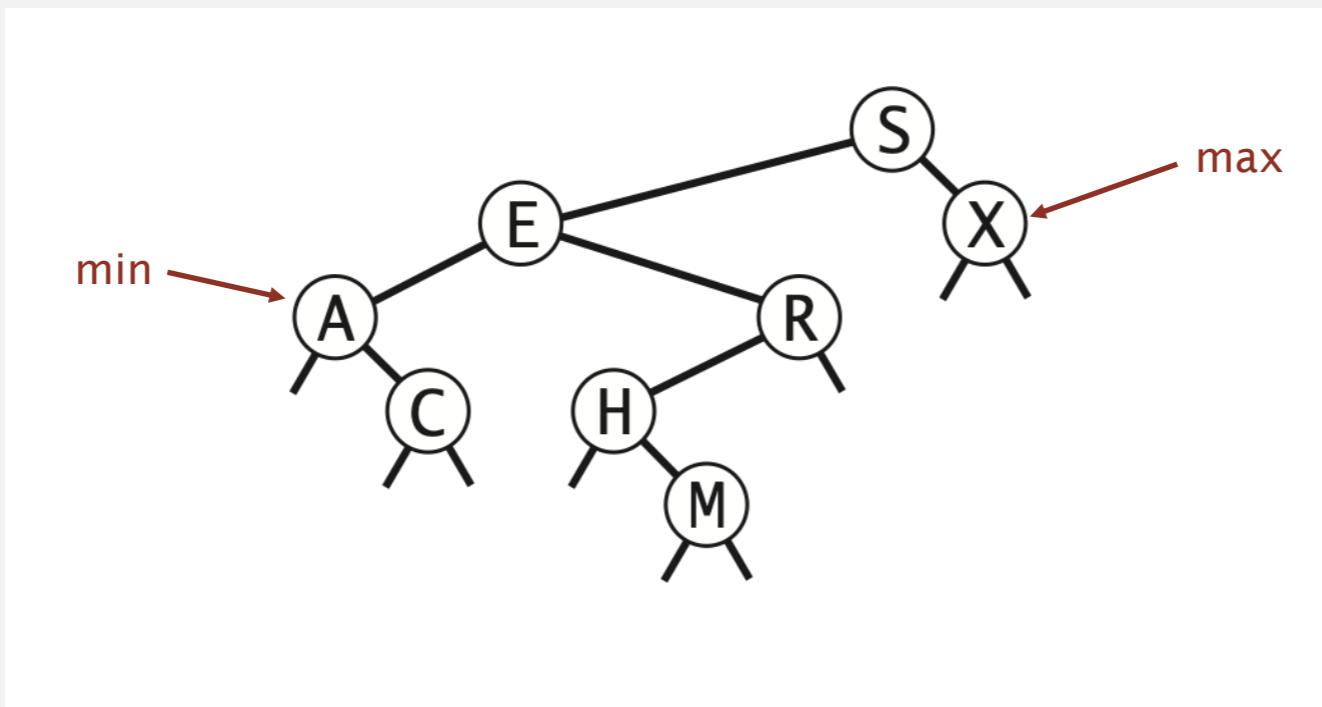
# BINARY SEARCH TREES

---

- *Symbol Table*
- *BSTs*
- *Iteration*
- *Ordered operations*
- *Deletion*

**Minimum.** Smallest key in BST.

**Maximum.** Largest key in BST.



**Q.** How to find the min / max?

**A.** Follow leftmost/rightmost spine

# Implementing min() and max()

LO 5.4

```
public Key max() {
 if (isEmpty()) throw new NoSuchElementException("calls max() with empty symbol table");
 return max(root).key;
}

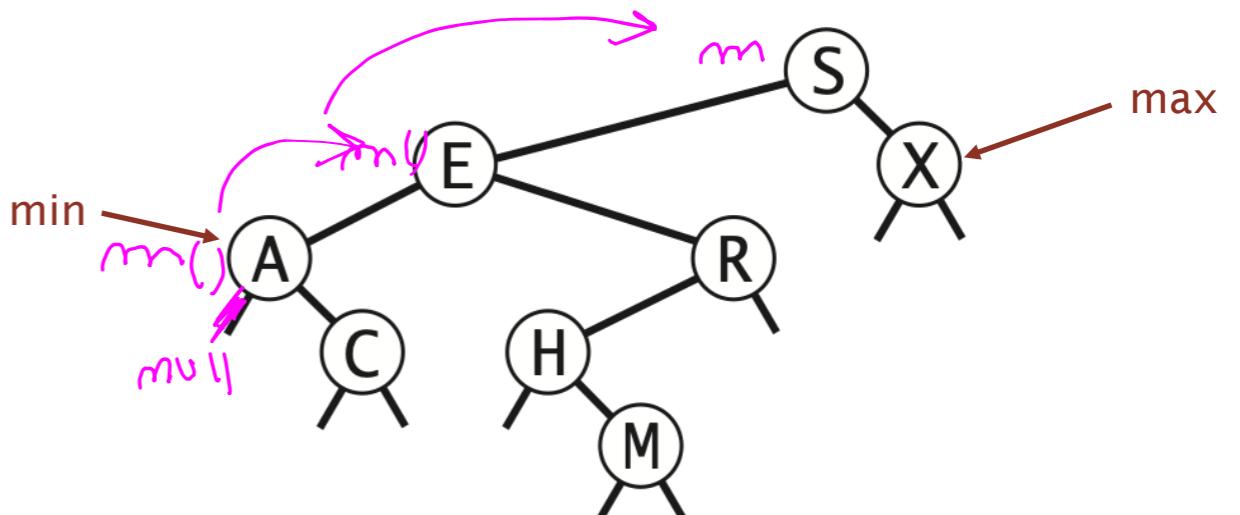
private Node max(Node x) {
 if (x.right == null) return x;
 else return max(x.right);
}
```

```
public Key min() {
 if (isEmpty()) throw new NoSuchElementException("calls min() with empty symbol table");
 return min(root).key;
}

private Node min(Node x) {
 if (x.left == null) return x;
 else return min(x.left);
}
```

null

min



# Floor and ceiling

LO 5.9

Floor. Largest key in BST  $\leq$  query key.  $\lfloor \cdot \rfloor$   $\lfloor 3.7 \rfloor = 3$   $\lfloor 2.1 \rfloor = 2$

Ceiling. Smallest key in BST  $\geq$  query key.  $\lceil \cdot \rceil$   $\lceil 3.7 \rceil = 4$   $\lceil 2.1 \rceil = 3$

Q. How to find the floor and ceiling?

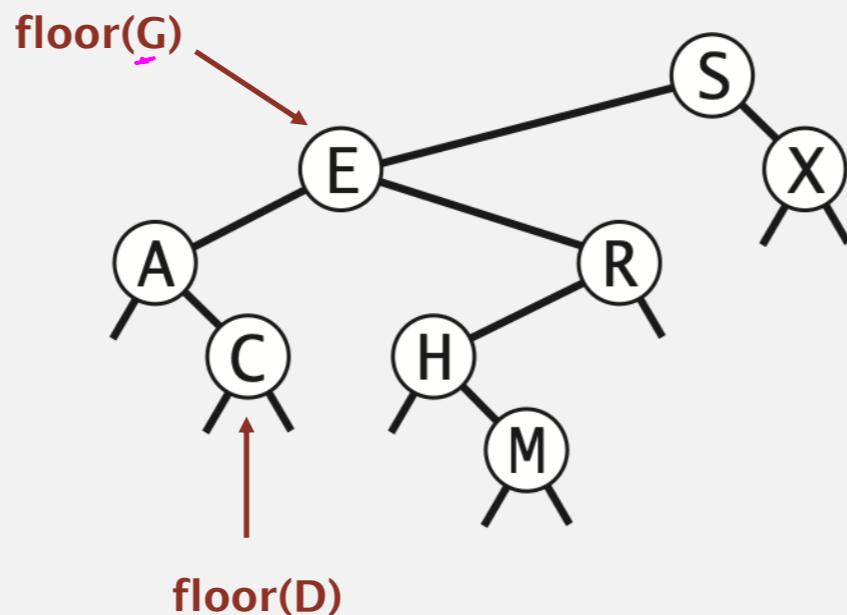
# Computing the floor

LO 5.9

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

Key idea.

- To compute  $\text{floor}(\text{key})$ , search for key.
- On search path, must encounter  $\text{floor}(\text{key})$  and  $\text{ceiling}(\text{key})$ . Why?



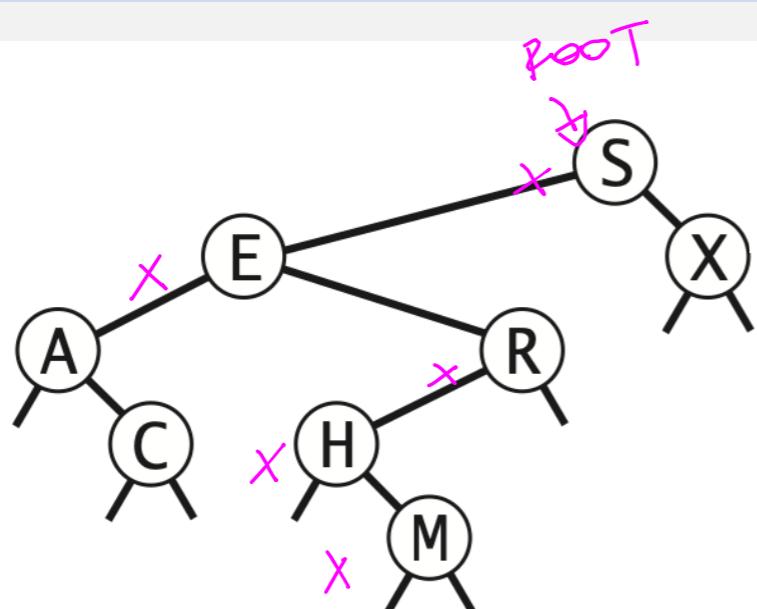
# Computing the floor

floor  
G H I J K L M ceiling

LO 5.9

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

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



$\text{H} \rightarrow \text{floor}(j)$   
 $\text{H} \rightarrow \text{floor}(s, j, \text{l})$   
 $\text{H} \rightarrow \text{floor}(t, j, \text{l})$   
 $\text{H} \rightarrow \text{floor}(r, j, e)$   
 $\text{H} \rightarrow \text{floor}(h, j, e)$   
 $\text{H} \rightarrow \text{floor}(m, j, h)$   
 $\text{H} \rightarrow \text{floor}(\text{l}, j, h)$

key in node is too big  
(so look in left subtree)

$\text{A} \rightarrow \text{Floor}(B)$   
 $\text{A} \rightarrow \text{Floor}(S, B, \text{l})$   
 $\text{A} \rightarrow \text{Floor}(T, B, \text{l})$   
 $\text{A} \rightarrow \text{Floor}(A, B, \text{l})$   
 $\text{A} \rightarrow \text{Floor}(C, B, A)$   
 $\text{A} \rightarrow \text{Floor}(\text{l}, B, A)$

key in node is best candidate for floor  
(but maybe better one in right subtree)

# Rank and select

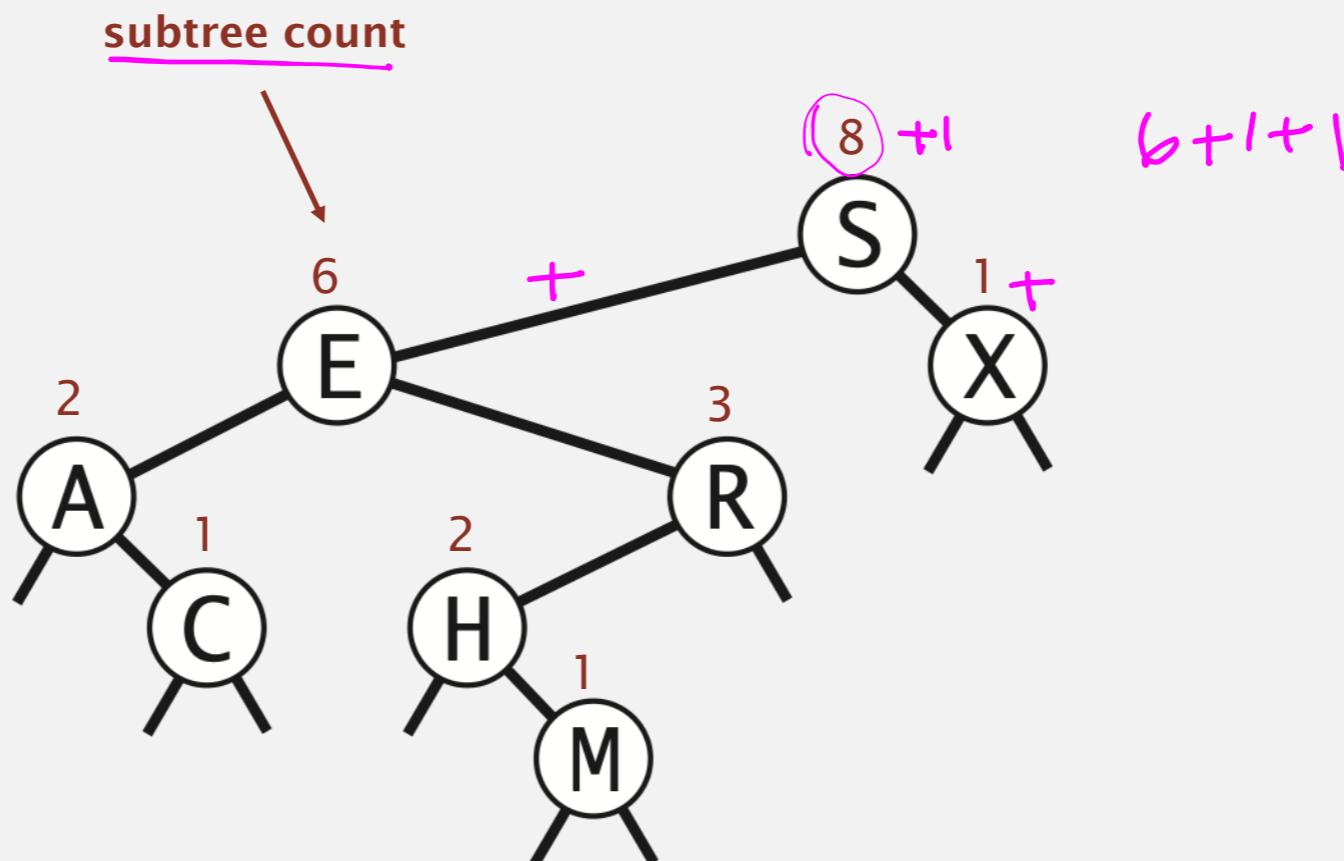
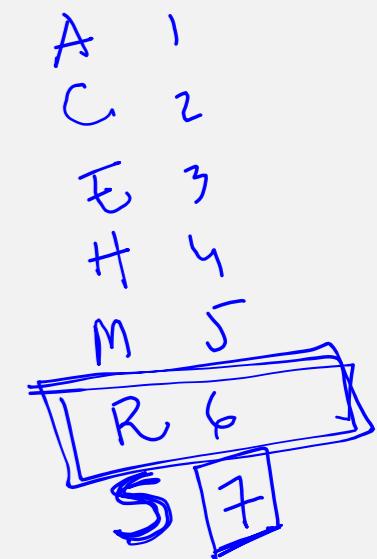
LO 5.9

Rank. How many keys < key?

Select. Key of rank  $k$ .

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

LO 5.9

```
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); // initialize subtree count to 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 = size(x.left) + size(x.right);
}
return x;
```

# Computing the rank

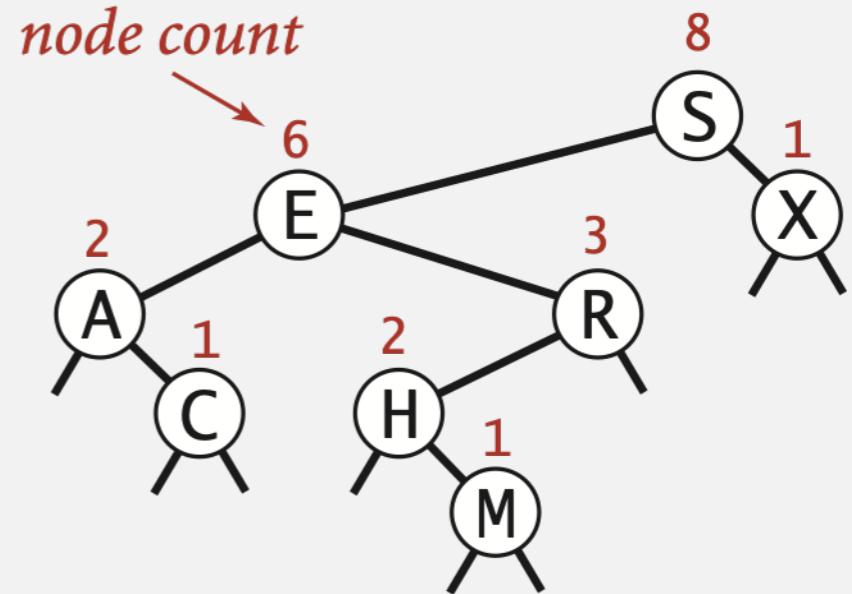
Rank E is 2

LO 5.9

Rank. How many keys  $< key$ ?

Case 1. [  $key < \text{key in node}$  ]

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



Case 2. [  $key > \text{key in node}$  ]

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

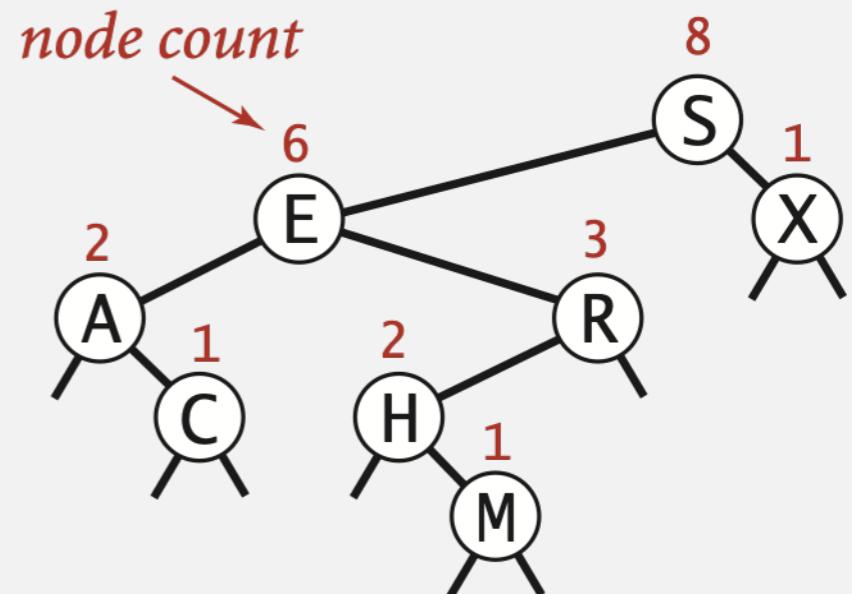
Case 3. [  $key = \text{key in node}$  ]

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



Rank. How many keys  $< key$ ?

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);
}
```

# BST: ordered symbol table operations summary

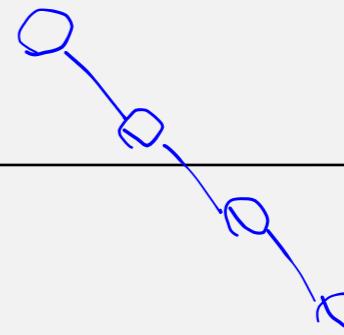
|                   | sequential search | binary search | BST |
|-------------------|-------------------|---------------|-----|
| search            | $n$               | $\log n$      | $h$ |
| insert            | $n$               | $n$           | $h$ |
| min / max         | $n$               | 1 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

Diagram illustrating the height  $h$  of a BST:

The diagram shows seven blue circles, each containing the letter 'h'. Six arrows point from these circles to a single point on the right side of the table, which is labeled 'h = height of BST'.

order of growth of running time of ordered symbol table operations

# ST implementations: summary



| implementation                    | guarantee |          | search hit | insert   | ordered ops? | key interface |
|-----------------------------------|-----------|----------|------------|----------|--------------|---------------|
|                                   | search    | 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 week. Guarantee logarithmic performance for all operations.

# BINARY SEARCH TREES

---

- *Symbol Table*
- *BSTs*
- *Iteration*
- *Ordered operations*
- *Deletion*

# ST implementations: summary

---

| implementation                           | guarantee |        |        | average case |          |        | ordered ops? | key interface            |
|------------------------------------------|-----------|--------|--------|--------------|----------|--------|--------------|--------------------------|
|                                          | search    | insert | delete | search hit   | insert   | delete |              |                          |
| <b>sequential search(unordered list)</b> | $n$       | $n$    | $n$    | $n$          | $n$      | $n$    |              | <code>equals()</code>    |
| <b>binary search(ordered array)</b>      | $\log n$  | $n$    | $n$    | $\log n$     | $n$      | $n$    | ✓            | <code>compareTo()</code> |
| <b>BST</b>                               | $n$       | $n$    | $n$    | $\log n$     | $\log n$ | ?      | ✓            | <code>compareTo()</code> |

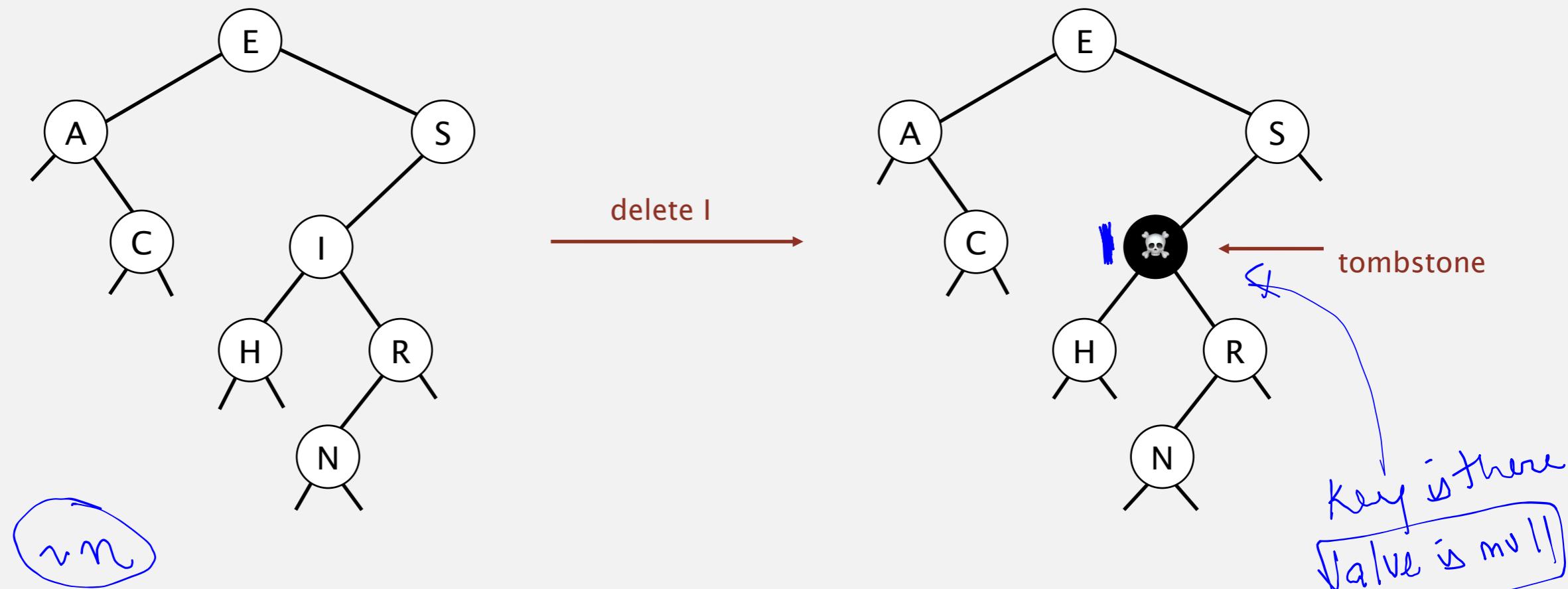
Next. Deletion in BSTs.

# BST deletion: lazy approach

LO 5.3

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  $\underline{n'}$  is the number of key-value pairs ever inserted in the BST.

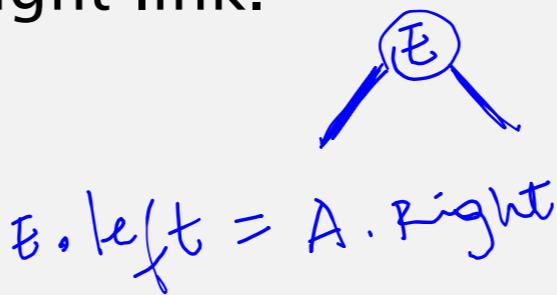
Unsatisfactory solution. Tombstone (memory) overload.

# Deleting the minimum

LO 5.3

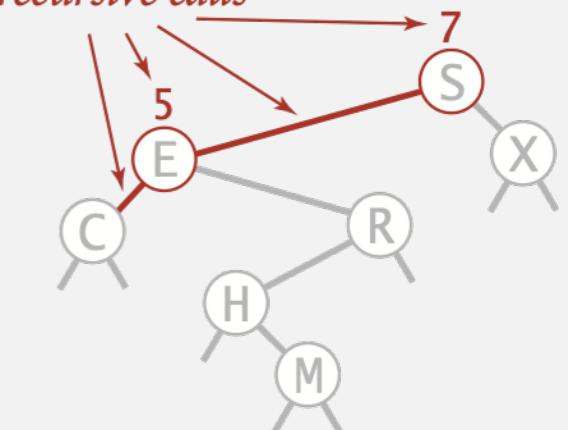
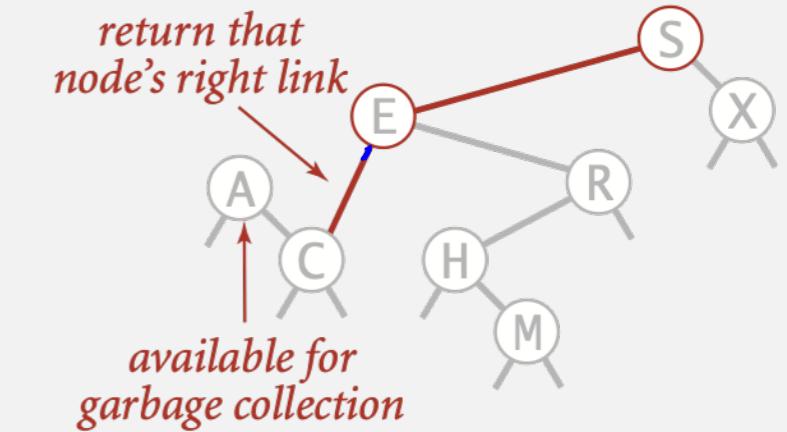
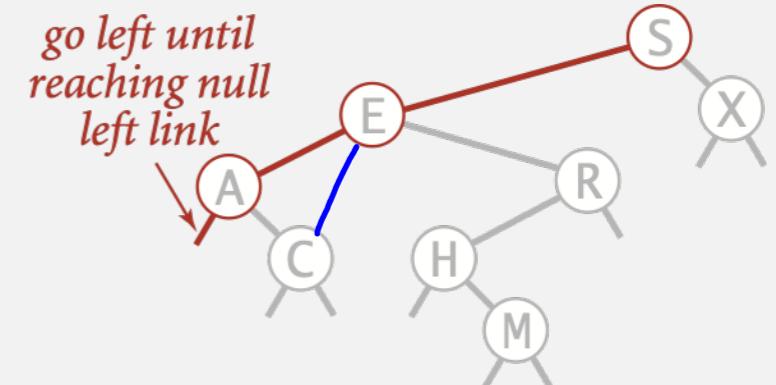
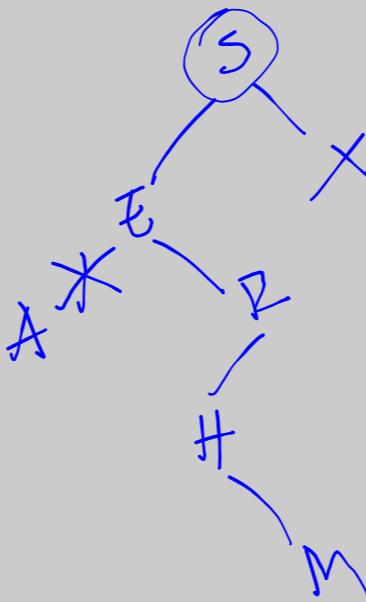
To delete the minimum key:

- 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;
}
```

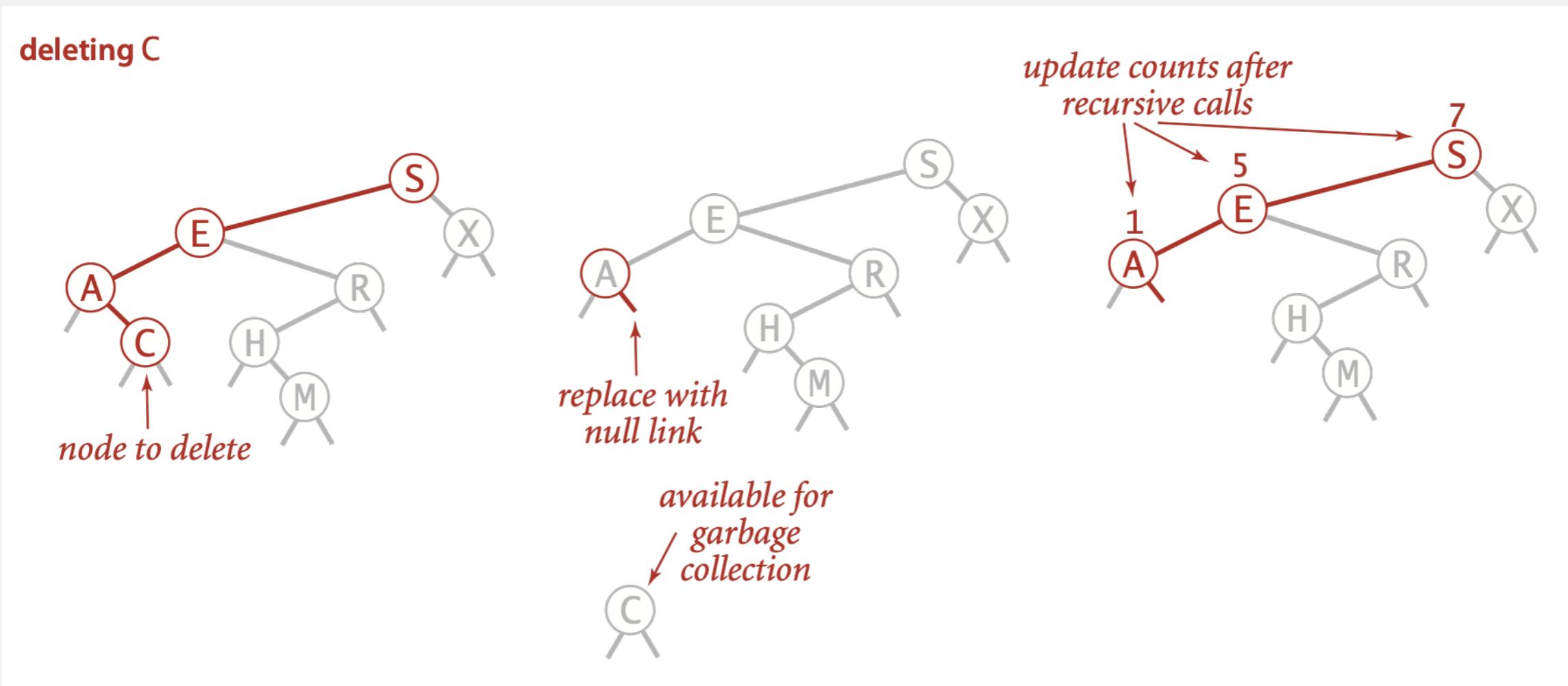


# Hibbard deletion

LO 5.3

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.

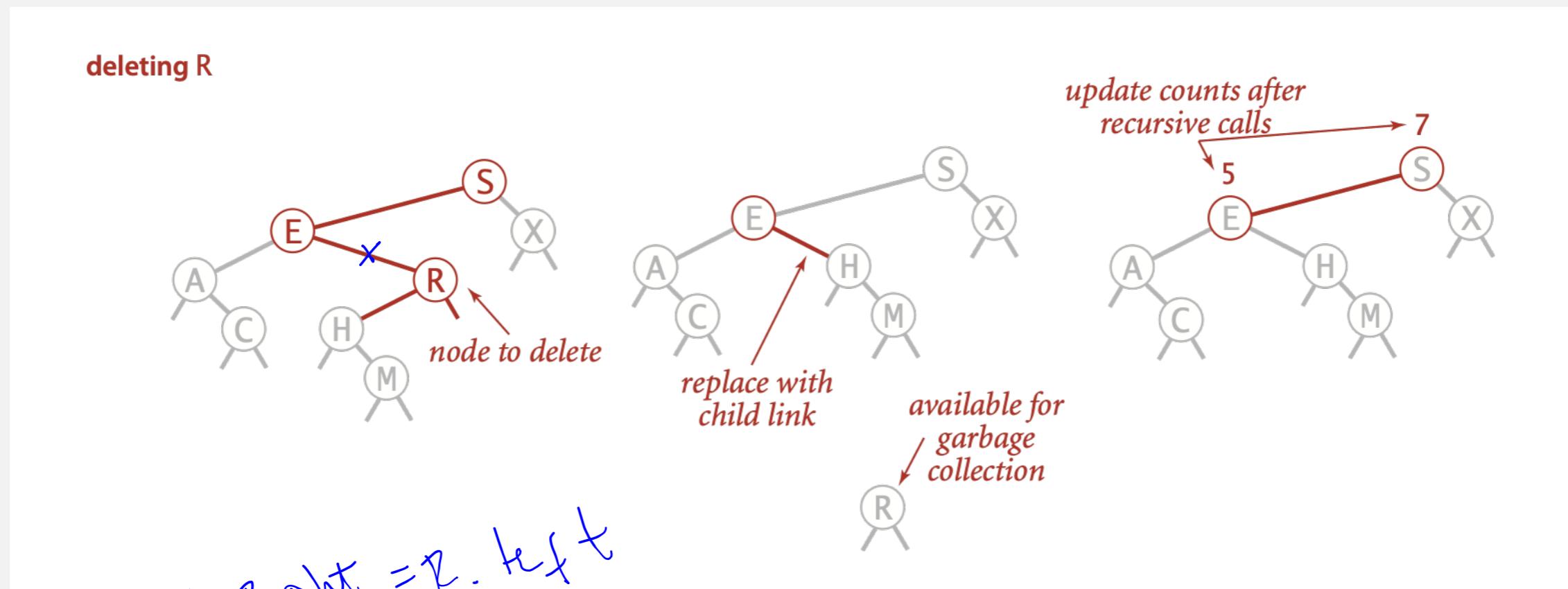


# Hibbard deletion

LO 5.3

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

LO 5.3

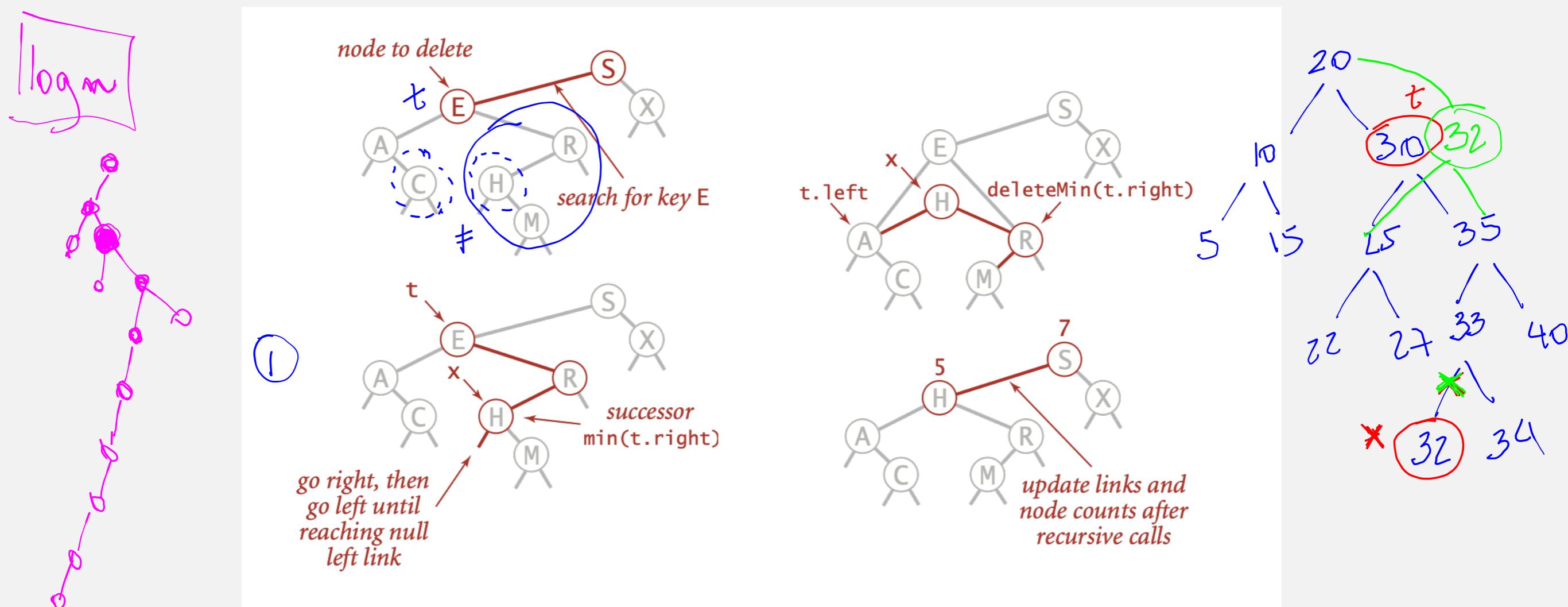
To delete a node with key  $k$ : search for node  $t$  containing key  $k$ .



## Case 2. [2 children]

- ① Find successor  $x$  of  $t$ .
- ② Delete the minimum in  $t$ 's right subtree.
- ③ Put  $x$  in  $t$ 's spot.

←  $x$  has no left child  
← but don't garbage collect  $x$   
← still a BST



# Hibbard deletion: Java implementation

LO 5.3

```
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

# INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

## BINARY SEARCH TREES

- *BSTs*
- *Ordered operations*
- *Iteration*
- *Deletion*

$$\begin{array}{c} O(\log n) \\ \downarrow \\ O(n) \end{array}$$

