

Binary Search Trees

- Trees
- BSTs
- ordered operations
- deletion

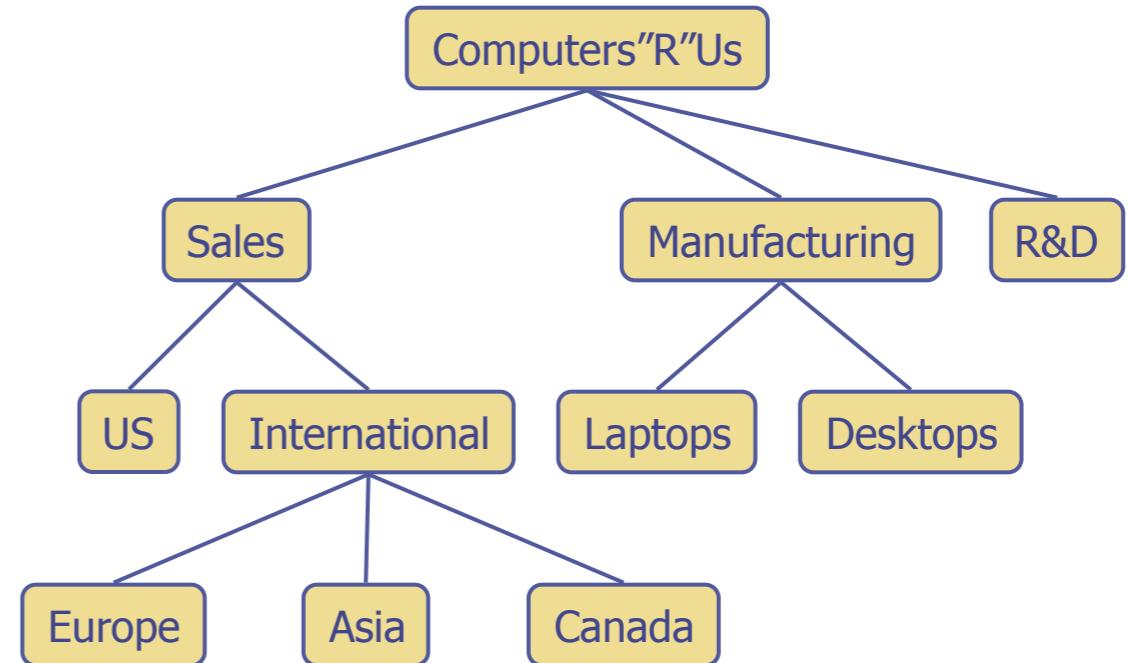
What is a tree?

In computer science, a tree is an abstract model of a hierarchical structure

A tree consists of nodes with a parent-child relation

Applications:

- Organization charts
- File systems
- Programming environments



Tree terminology

Root:

- Node without parent (e.g., A)

Internal node:

- Node with at least one child
(e.g., A, B, C, F)

External node:

- Node without children
(e.g., E, I, J, K, G, H, D)
- Also known as **leaf node**

Ancestors of a node:

- Parent, grandparent, grand-grandparent, etc.

Descendant of a node:

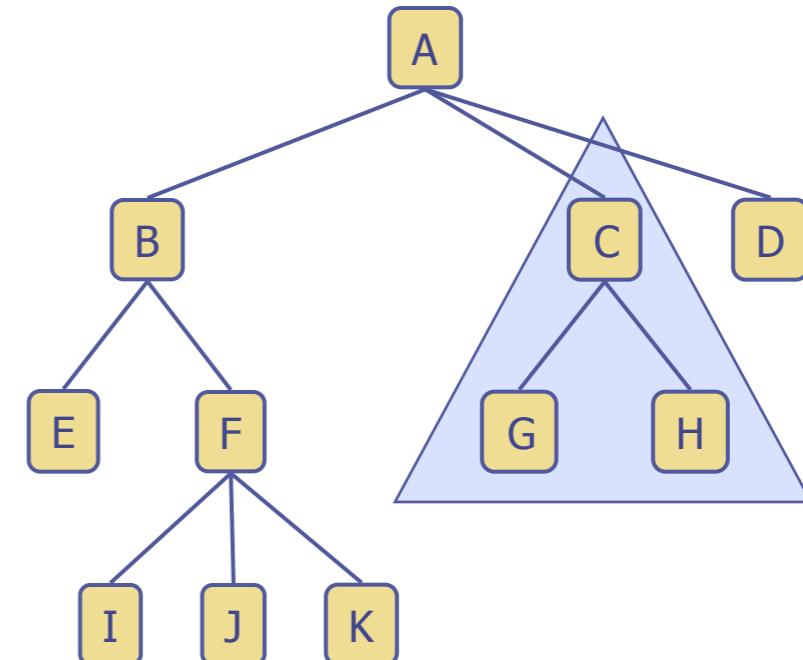
- Child, grandchild, grand-grandchild, etc.

Depth of a node:

- Number of ancestors

Height of a tree:

- Maximum depth of any node (e.g., 3)



Subtree:

- Tree consisting of a node and its descendants

Tree: abstract data type

We use positions to abstract nodes

Generic methods:

- integer size()
- boolean isEmpty()
- Iterator iterator()
- Iterable positions()

Accessor methods:

- position root()
- position parent(p)
- Iterable children(p)

Query methods:

- boolean isInternal(p)
- boolean isExternal(p)
- boolean isRoot(p)

Update method:

- element replace (p, o)

Additional update methods
may be defined by data
structures implementing the
Tree ADT

Depth and height

Define depth recursively

- If v is the root, then the depth of v is 0
- Otherwise, the depth of v is one plus the depth of the parent of v .

Define height recursively

- If v is an external node, then the height of v is 0
- Otherwise, the height of v is one plus the maximum height of a child of v .

Preorder traversal

A traversal visits the nodes of a tree in a systematic manner

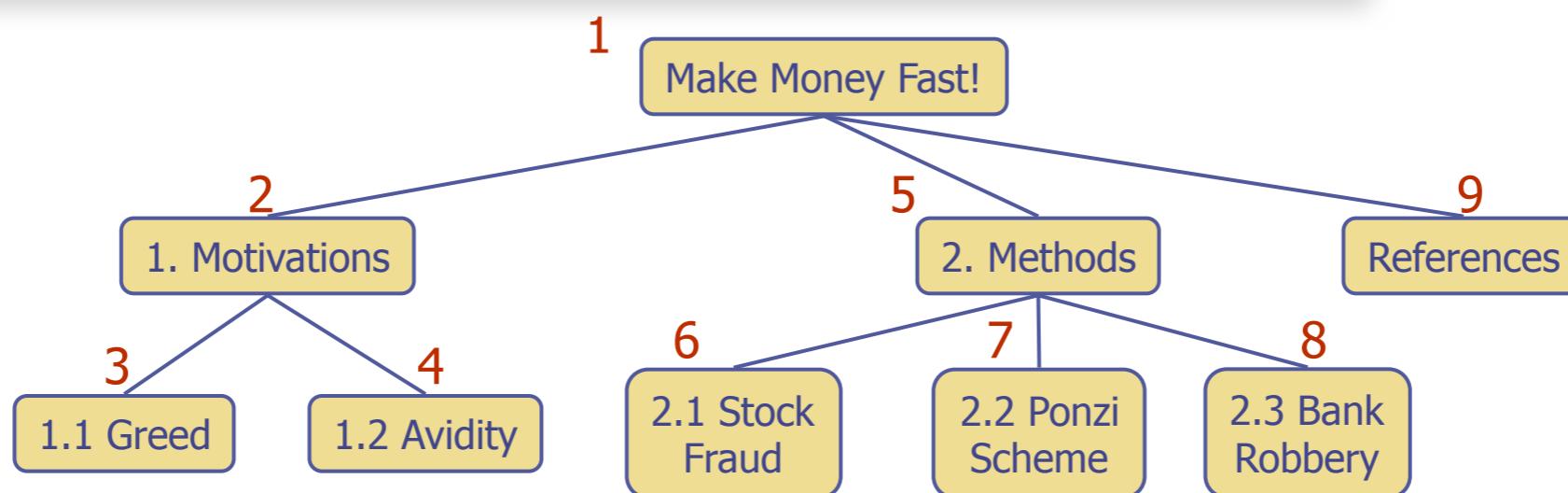
In a **preorder** traversal, a node is visited **before** its descendants

- Application: print a structured document

```
Algorithm preOrder(v)
```

- visit(v)
- for each child w of v

```
    preorder (w)
```



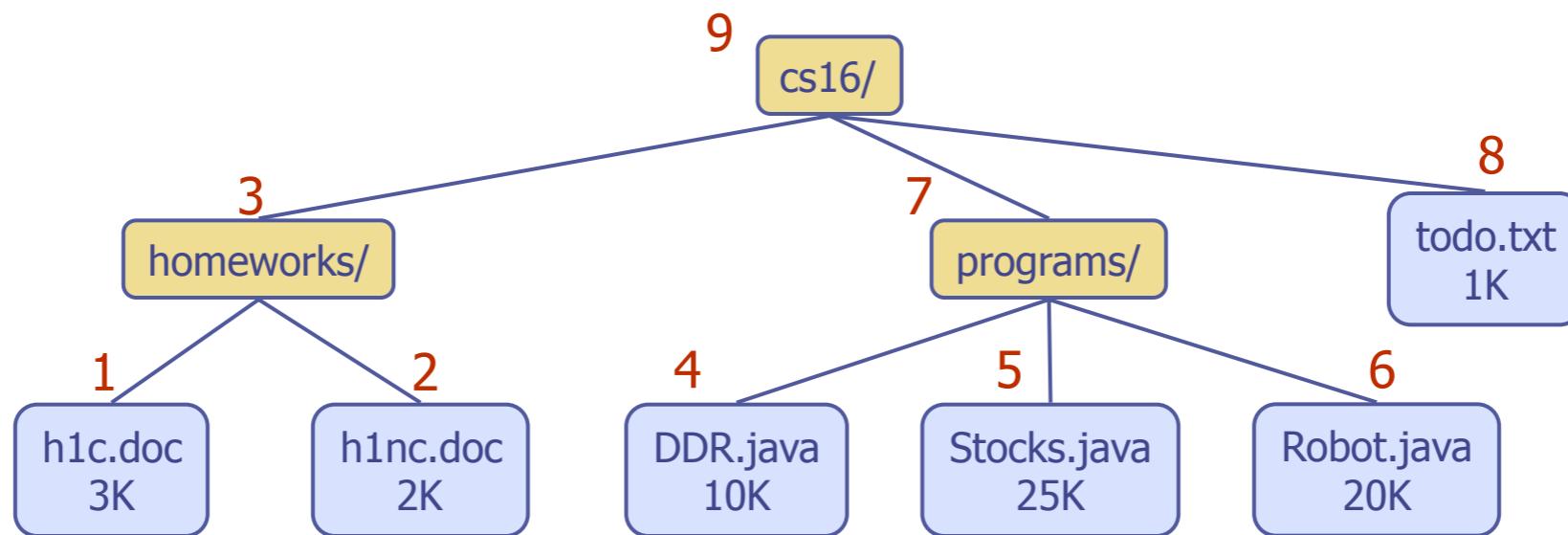
Postorder traversal

In a **postorder** traversal, a node is visited **after** its descendants

- Application: compute space used by files in a directory and its subdirectories

```
Algorithm postOrder(v)
```

- for each child w of v
 - postOrder (w)
- visit(v)



Binary trees

A binary tree is a tree with the following properties:

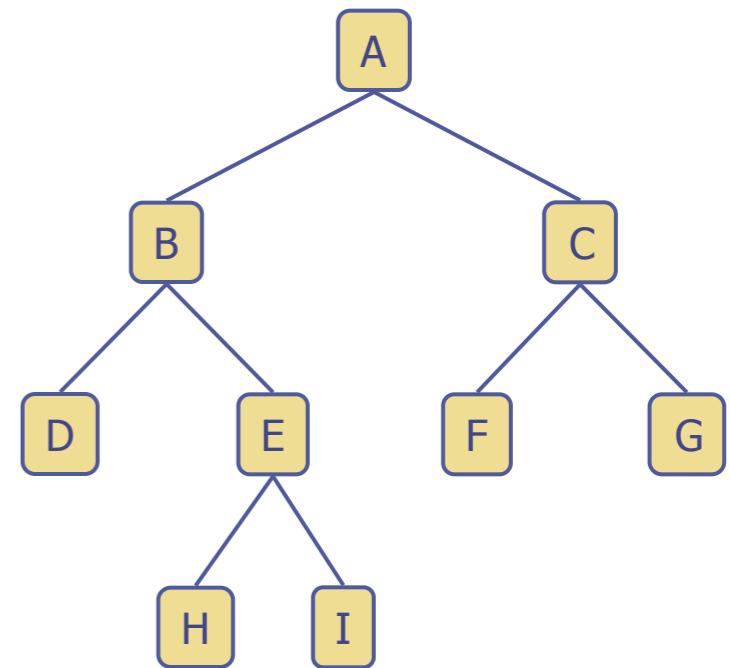
- Each internal node has at most two children
(exactly two for proper binary trees)
- The children of a node are an ordered pair
- We call the children of an internal node left child
and right child

Alternative recursive definition: a binary tree is either

- Tree consisting of a single node, or
- Tree whose root has an ordered pair of children,
each of which is a binary tree

Applications:

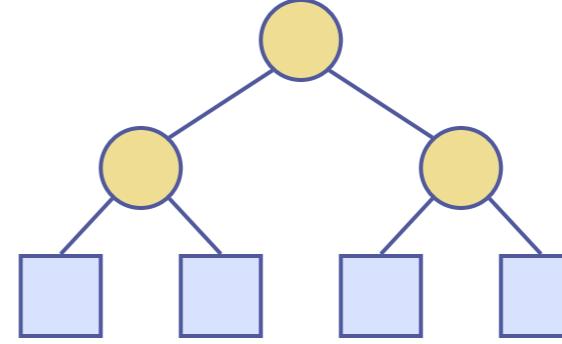
- Arithmetic expressions
- Decision processes
- Searching



Properties of binary trees

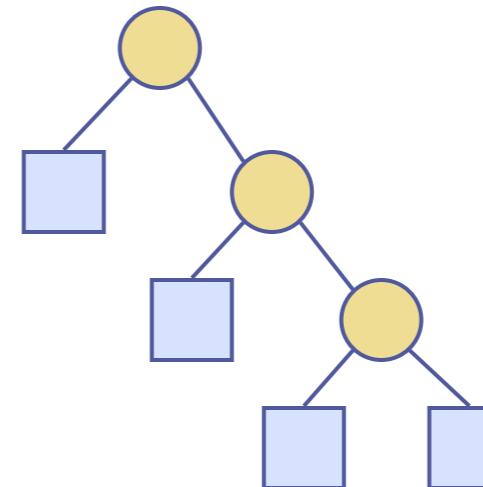
Notation

- n : number of nodes
- e : number of external nodes
- i : number of internal nodes
- h : height



Properties:

- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2(n + 1) - 1$

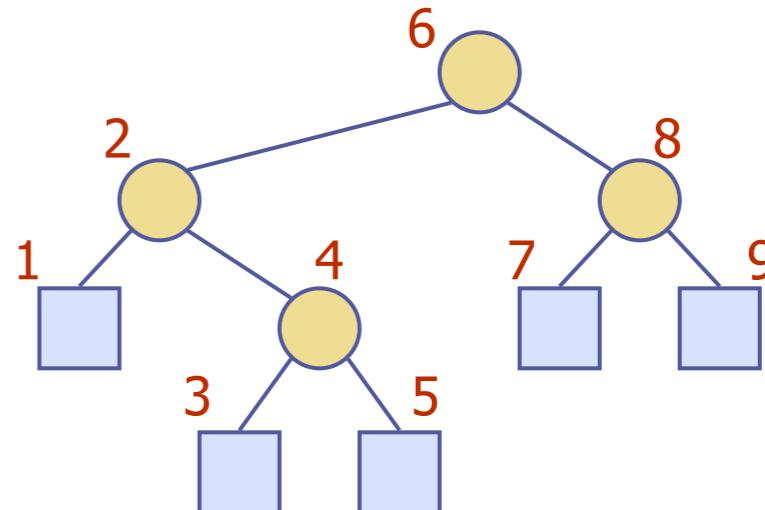


Inorder traversal

In an **inorder** traversal a node is visited **after** its left subtree and **before** its right subtree

- Application: draw a binary tree
 - $x(v)$ = inorder rank of v
 - $y(v)$ = depth of v

```
Algorithm inOrder (v)
    • if hasLeft (v)
        inOrder (left (v) )
    • visit(v)
    • if hasRight (v)
        inOrder (right (v) )
```

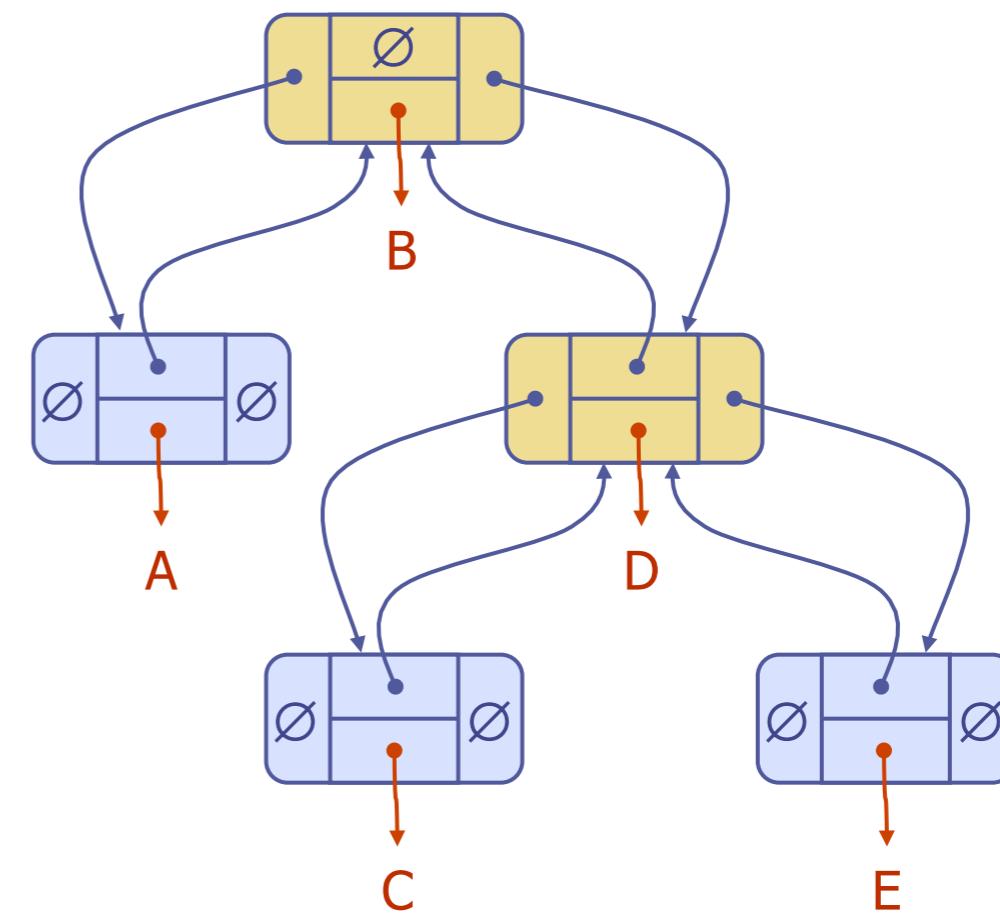
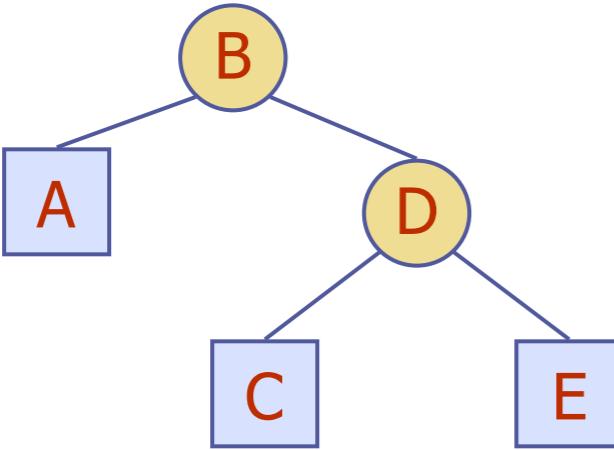


Linked structure for binary trees

A node is represented by an object storing

- Element
- Parent node
- Left child node
- Right child node

Node objects implement the Position ADT

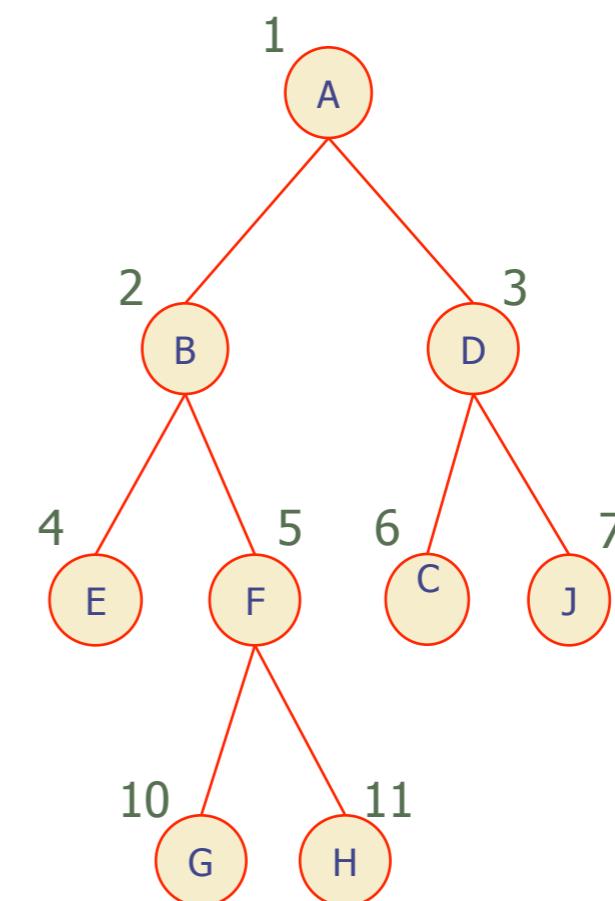


Array-based representation of binary trees

Nodes are stored in an array A

Node v is stored at $A[\text{rank}(v)]$

- $\text{rank}(\text{root}) = 1$
- if node is the left child of $\text{parent}(\text{node})$
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node}))$
- if node is the right child of $\text{parent}(\text{node})$
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node})) + 1$



Binary Search Trees

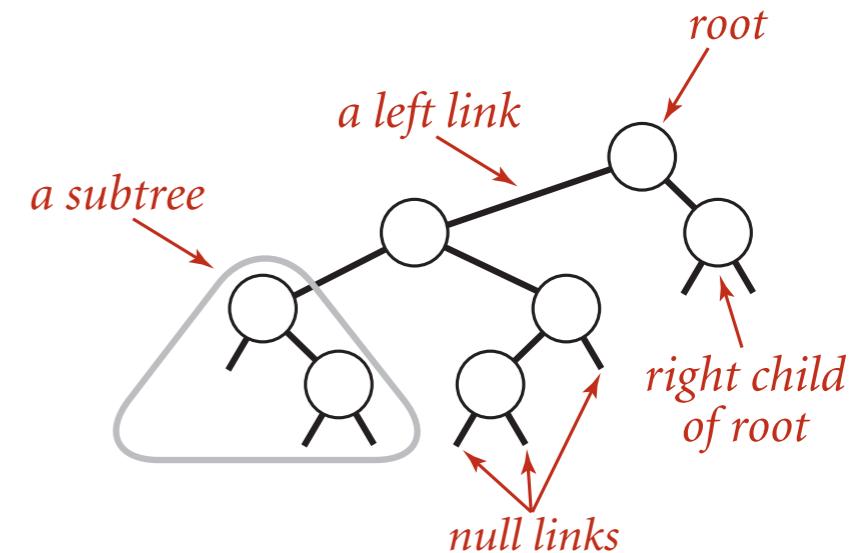
- Trees
- **BSTs**
- ordered operations
- deletion

Binary search trees

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

A binary tree is either:

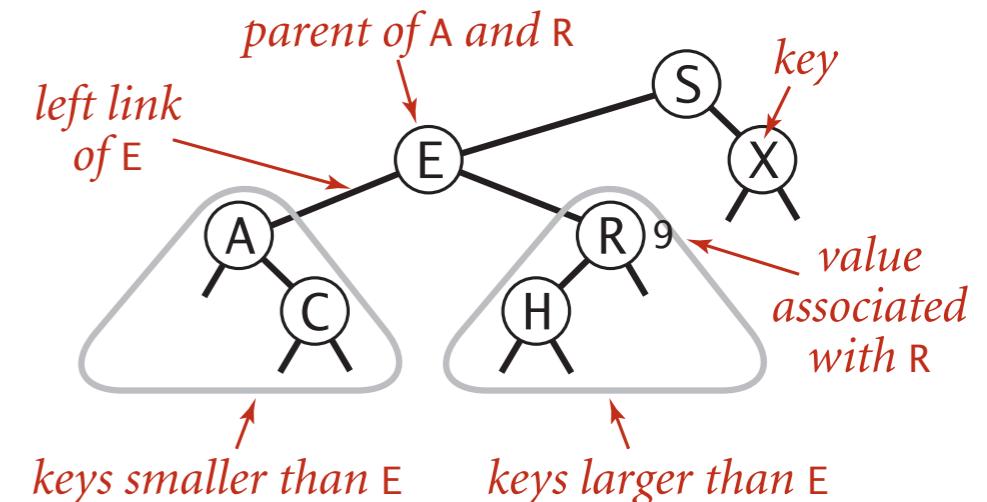
- Empty.
- Two disjoint binary trees (left and right).



Anatomy of a binary tree

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.



Anatomy of a binary search tree

BST representation in Java

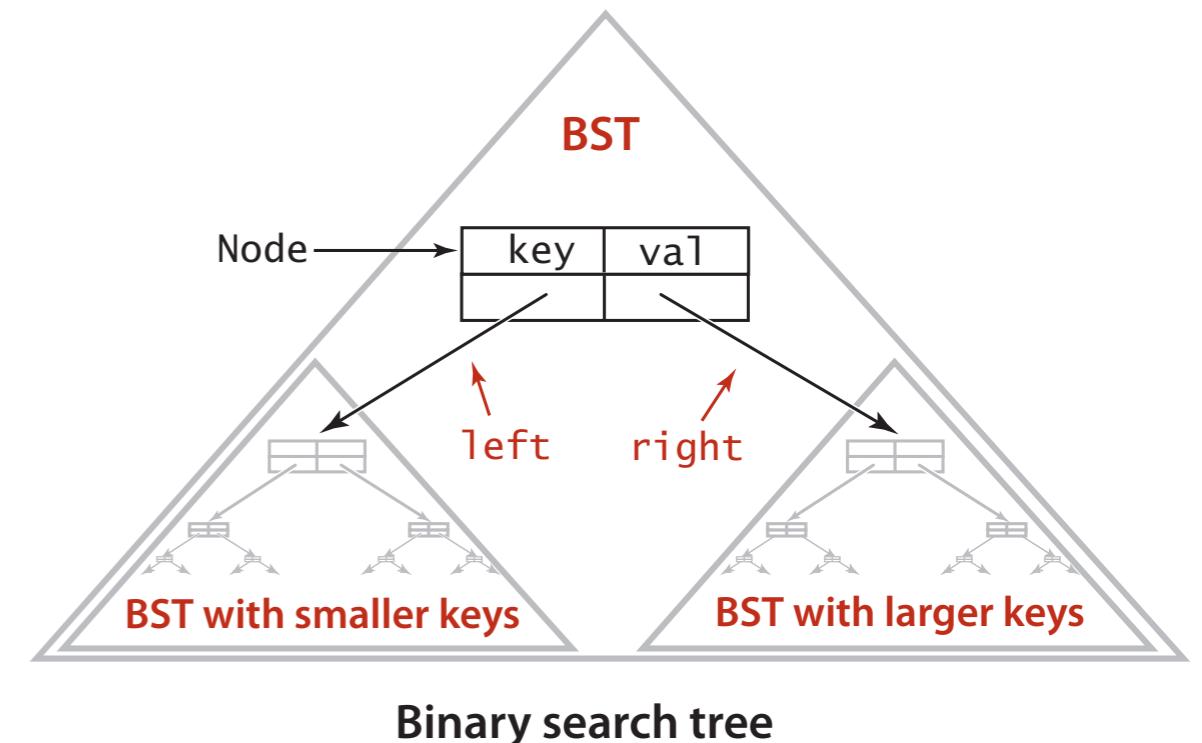
Java definition. A BST is a reference to a root **Node**.

A **Node** is comprised of four fields:

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



```
private class Node
{
    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)

```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;

    private class Node
    { /* see previous slide */ }

    public void put(Key key, Value val)
    { /* see next slides */ }

    public Value get(Key key)
    { /* see next slides */ }

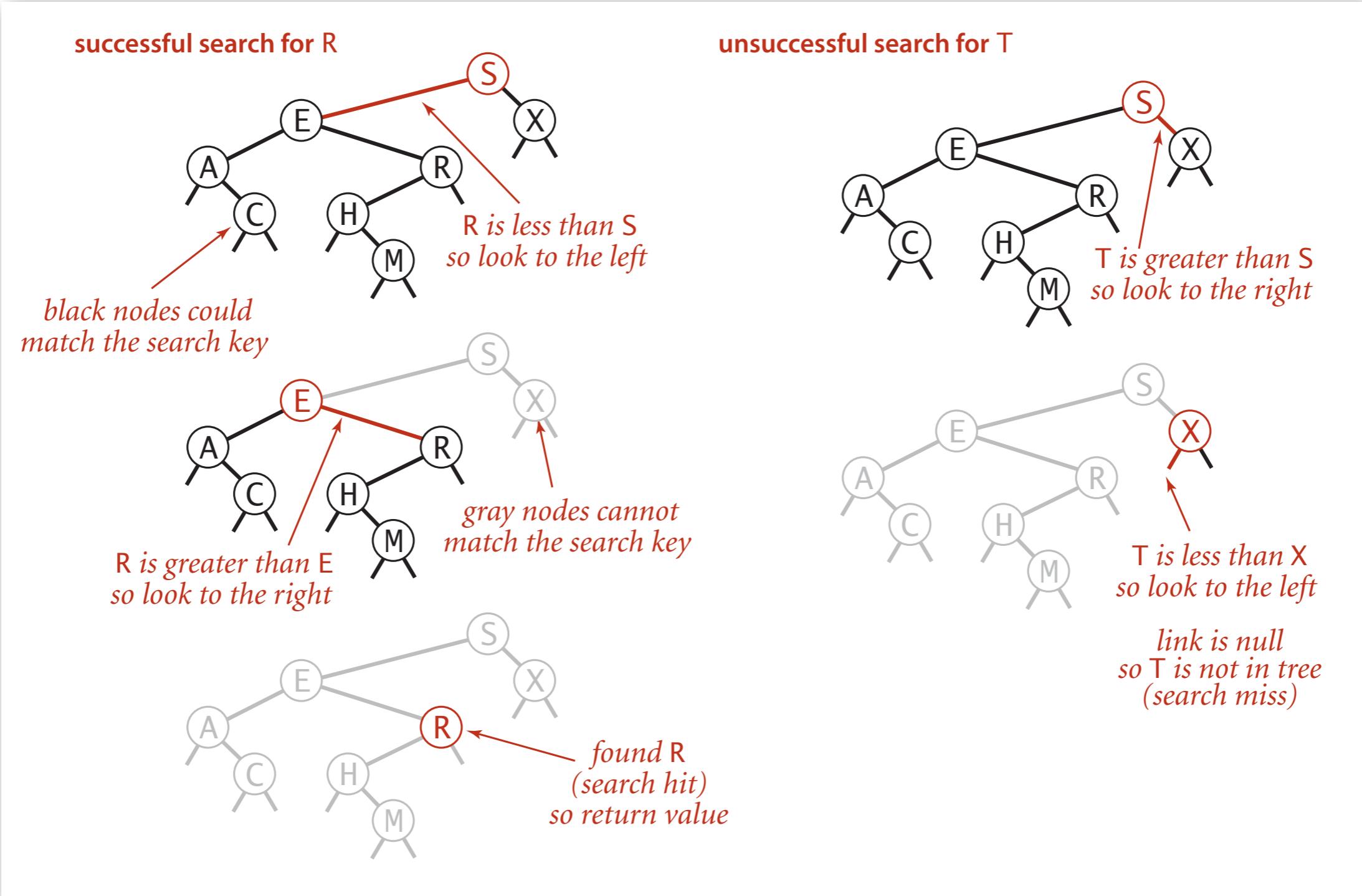
    public void delete(Key key)
    { /* see next slides */ }

    public Iterable<Key> iterator()
    { /* see next slides */ }
}
```

← root of BST

BST search

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



BST search: Java implementation

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

```
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;
}
```

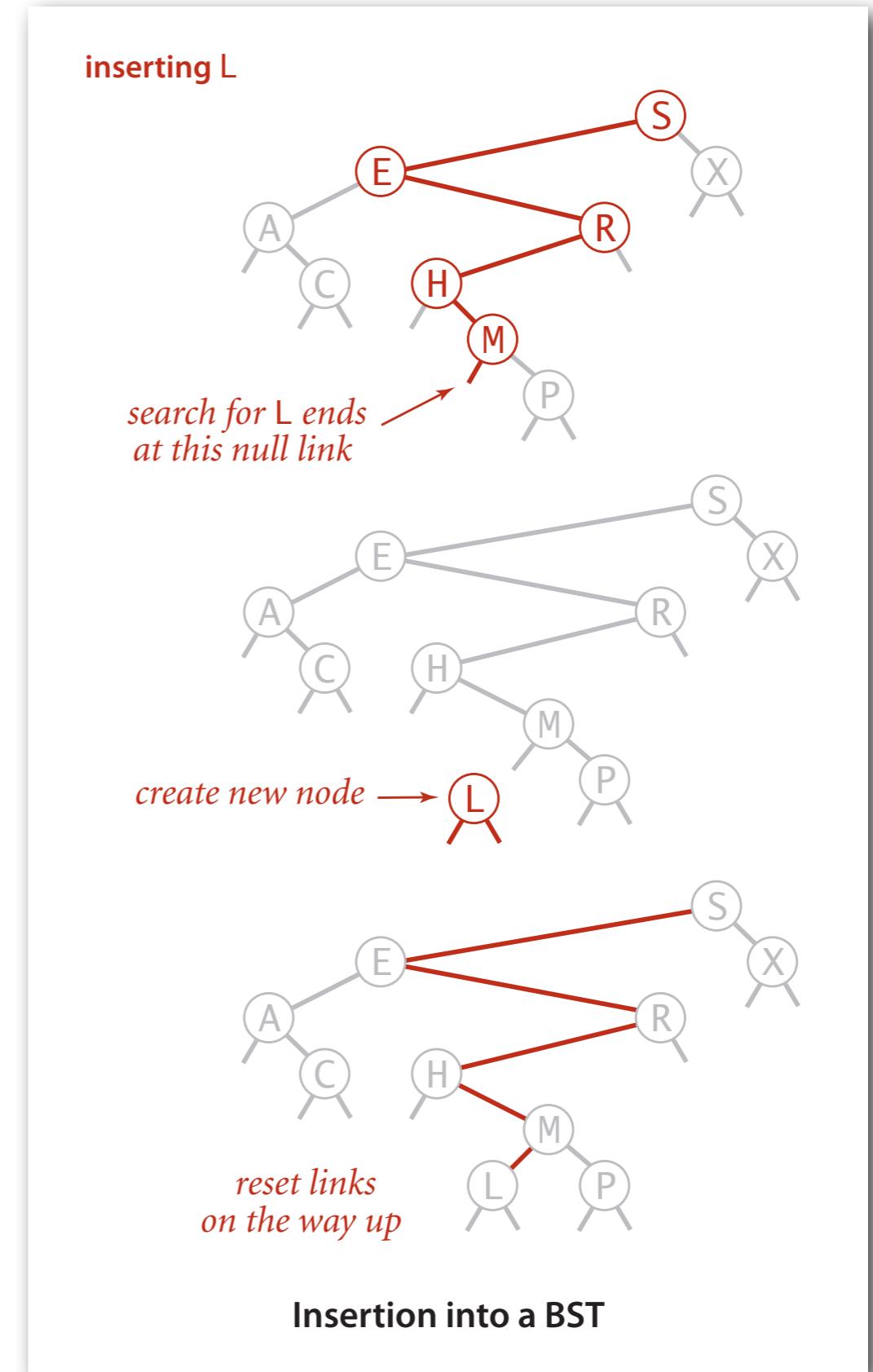
Cost. Number of compares is equal to depth of node.

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.



BST insert: Java implementation

Put. Associate value with key.

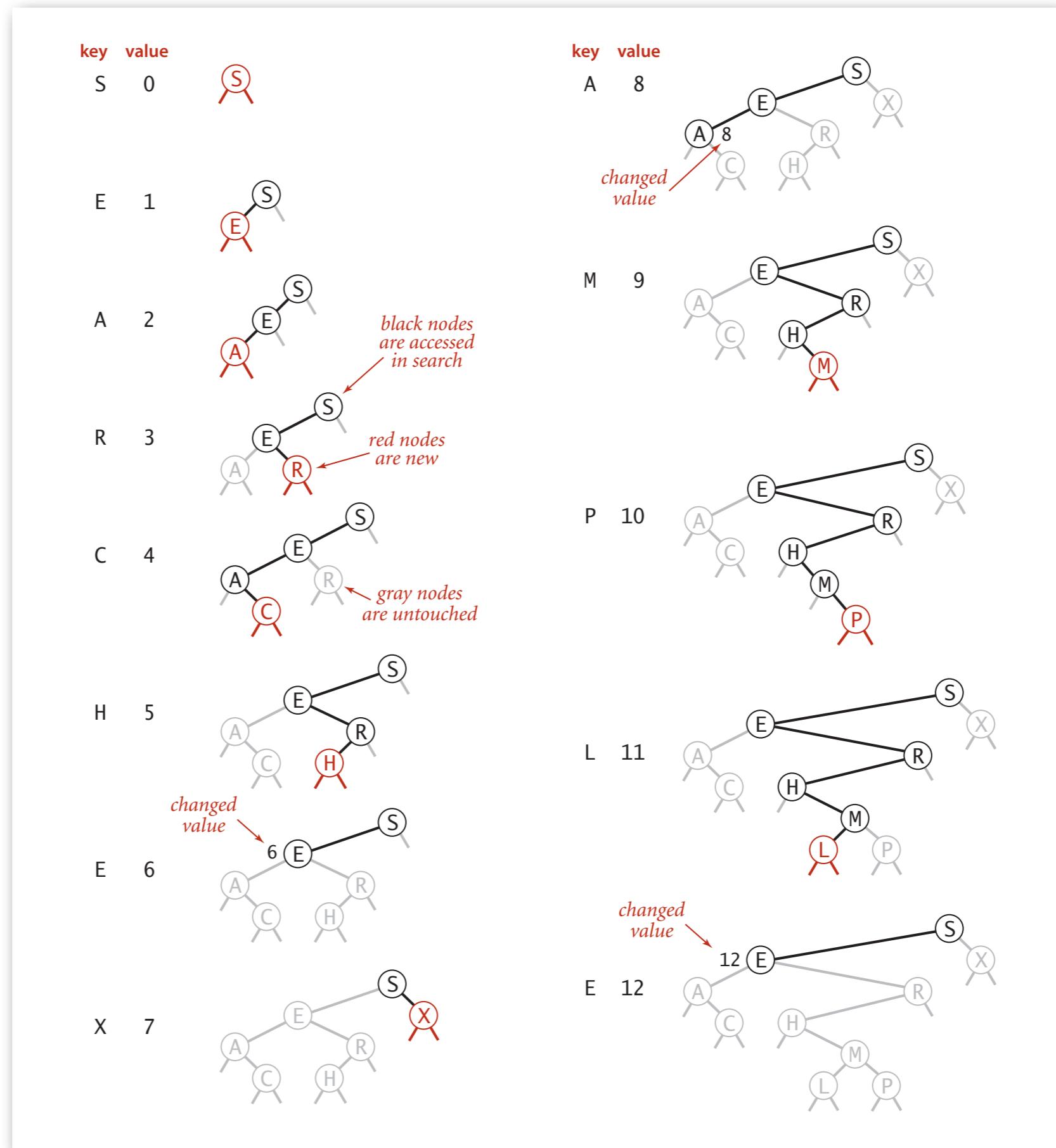
```
public void put(Key key, Value val)
{   root = put(root, key, val); }

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;
}
```

concise, but tricky,
recursive code;
read carefully!

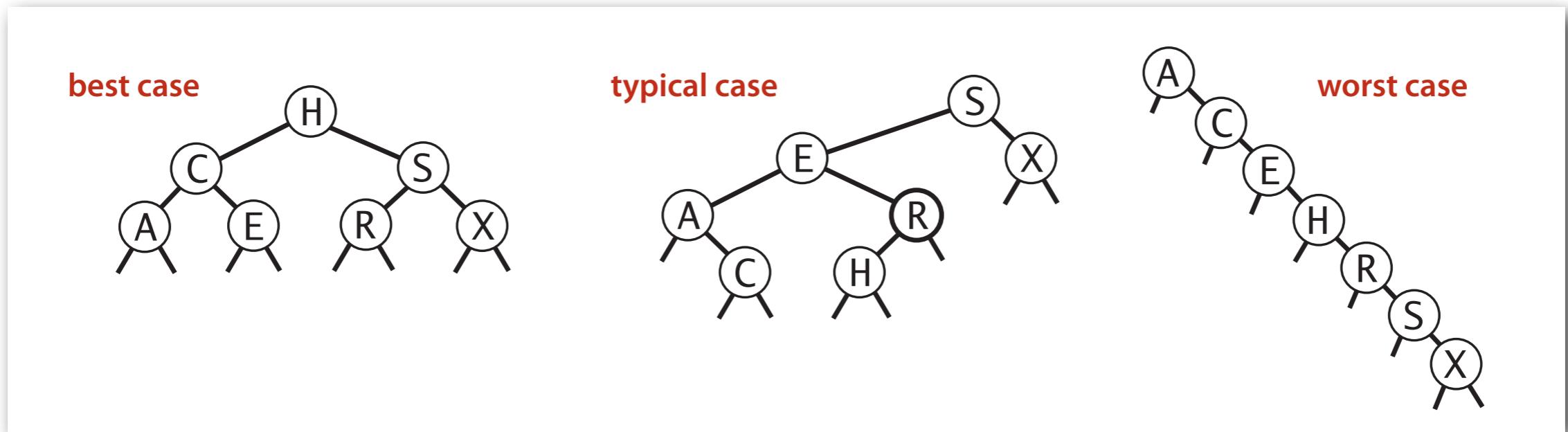
Cost. Number of compares is equal to depth of node.

BST trace: standard indexing client



Tree shape

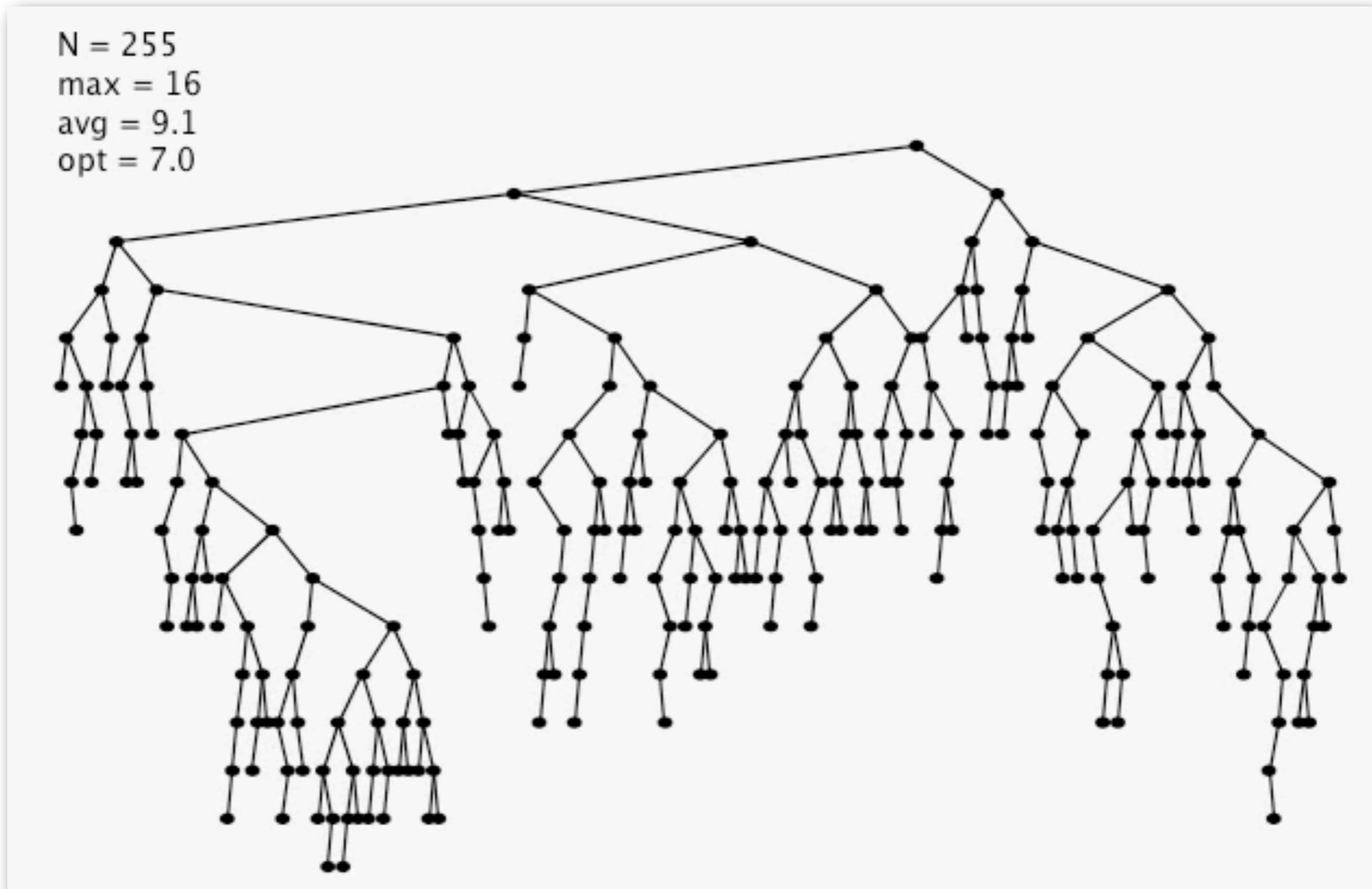
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to depth of node.



Remark. Tree shape depends on order of insertion.

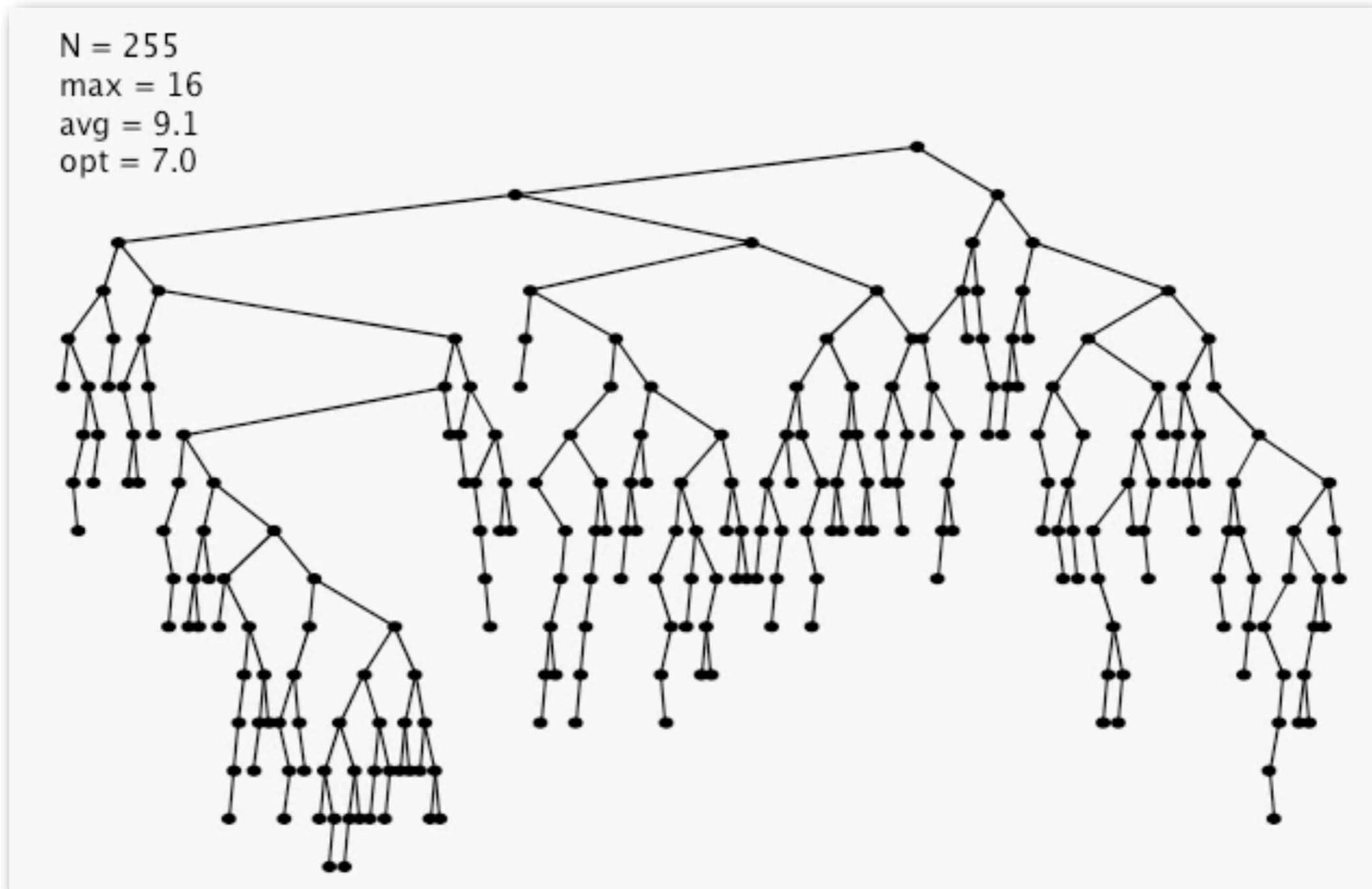
BST insertion: random order

Observation. If keys inserted in random order, tree stays relatively flat.

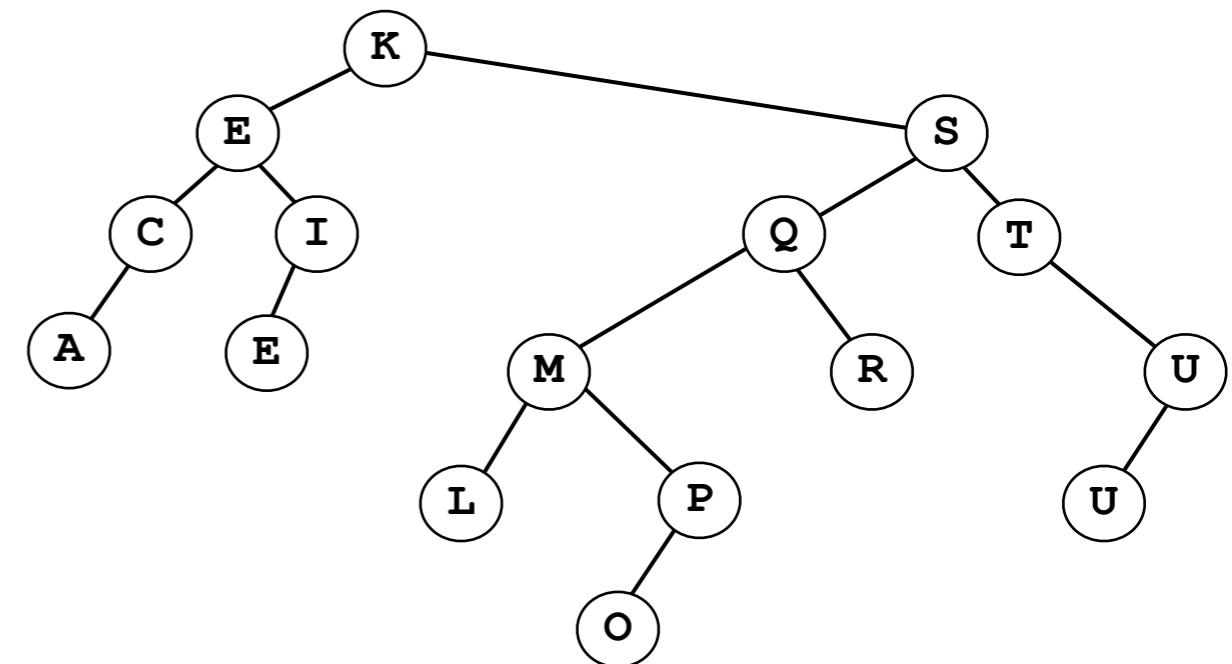
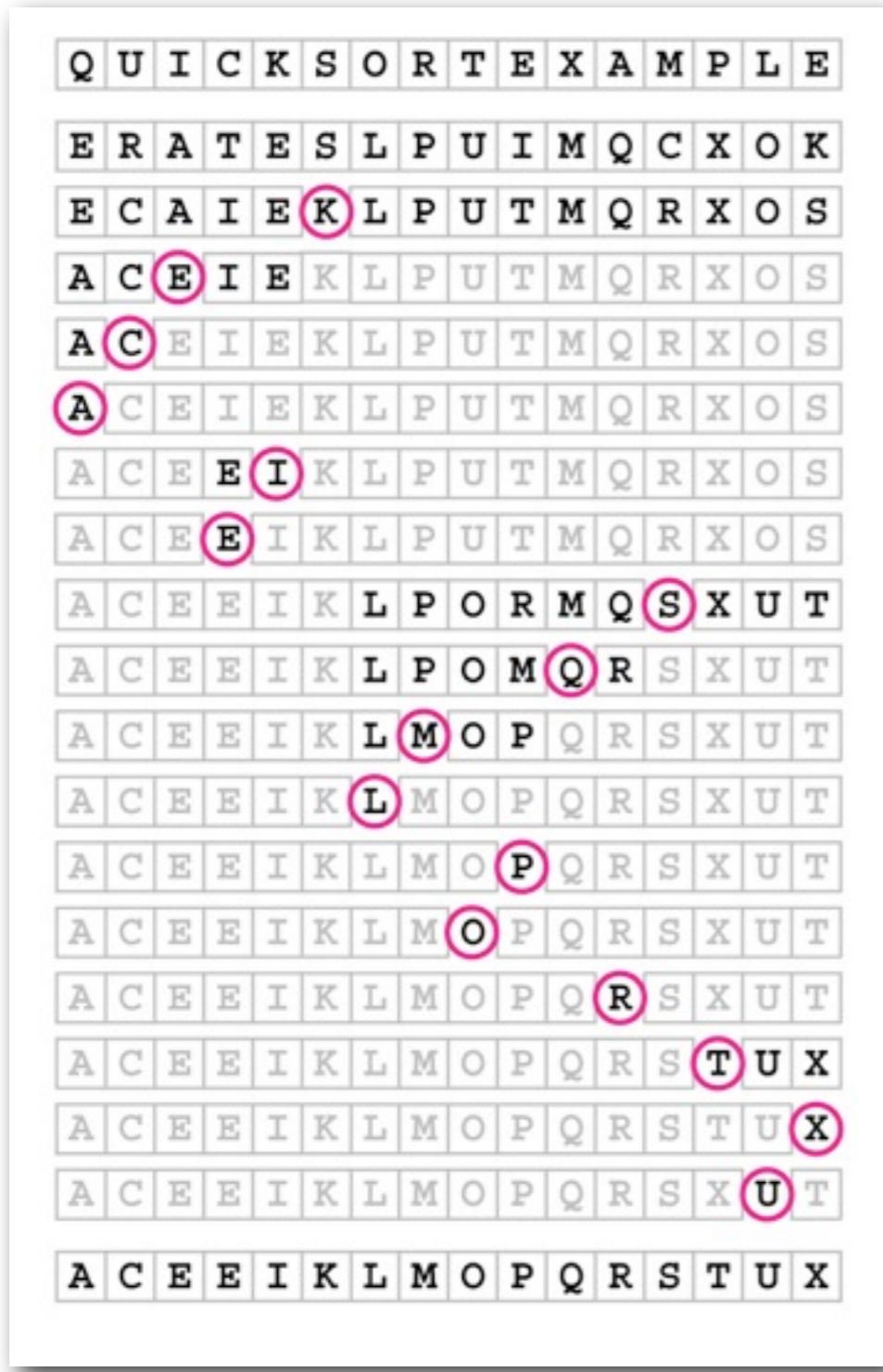


BST insertion: random order visualization

Ex. Insert keys in random order.



Correspondence between BSTs and quicksort partitioning



Remark. Correspondence is 1-1 if array has no duplicate keys.

BSTs: mathematical analysis

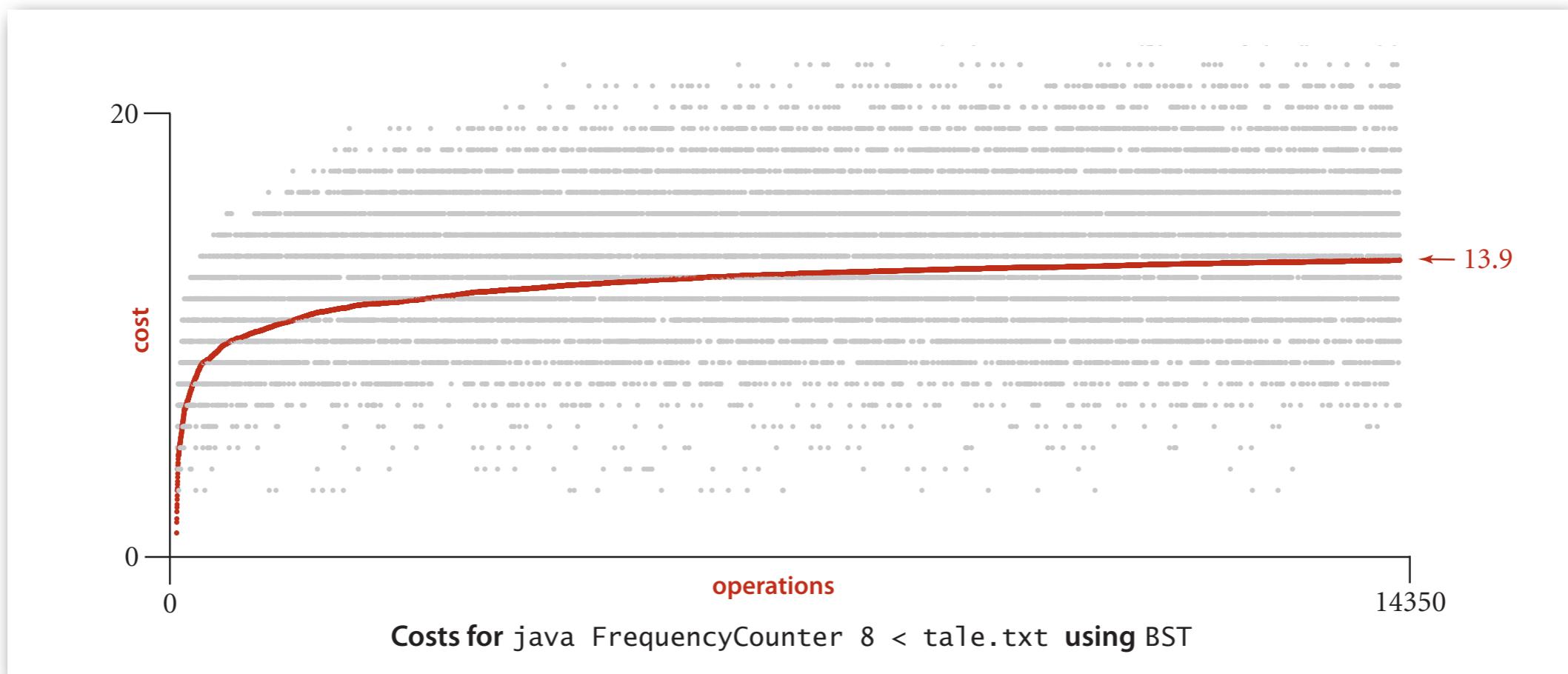
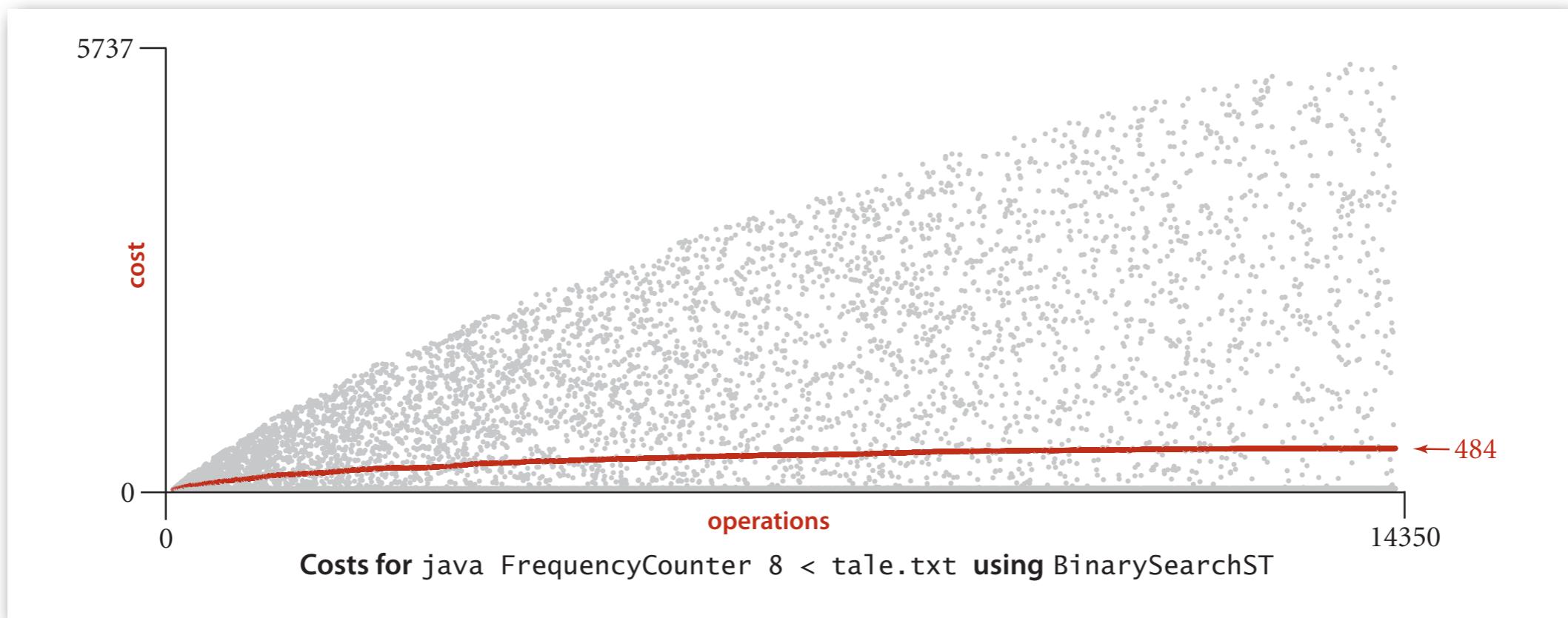
Proposition. If keys are inserted in **random** order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

But... Worst-case height is N .
(exponentially small chance when keys are inserted in random order)

ST implementations: frequency counter



ST implementations: summary

implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	$\lg N$	N/2	yes	<code>compareTo()</code>
BST	N	N	$1.39 \lg N$	$1.39 \lg N$?	<code>compareTo()</code>

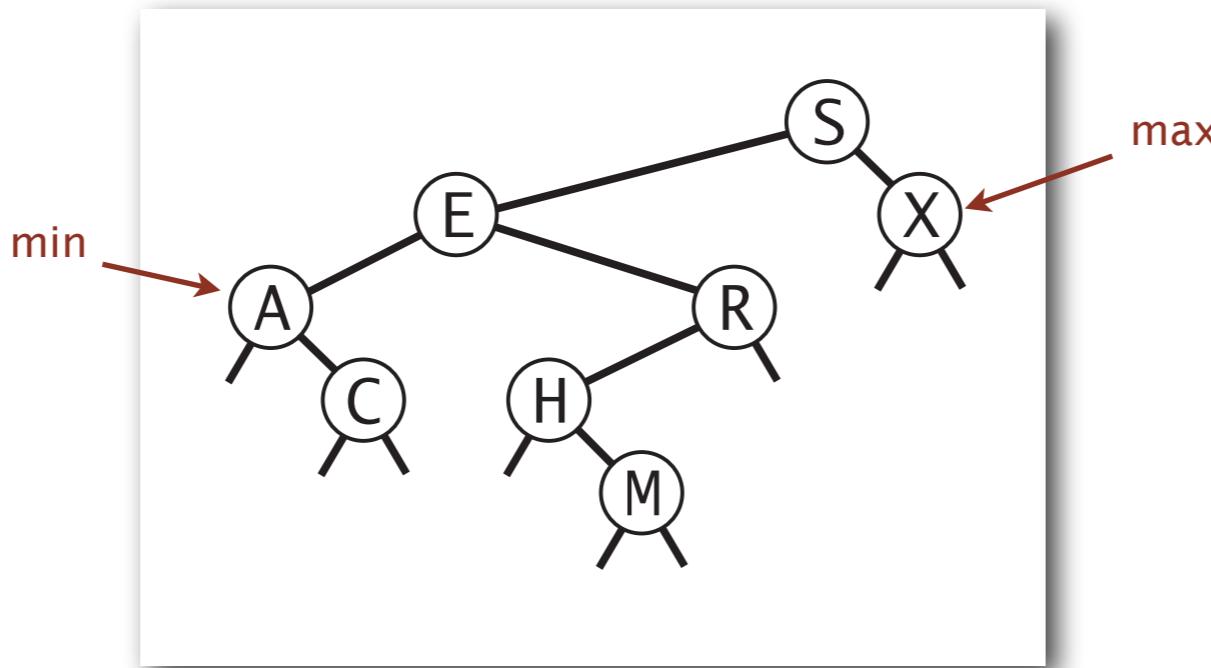
Binary Search Trees

- Trees
- BSTs
- ordered operations
- deletion

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

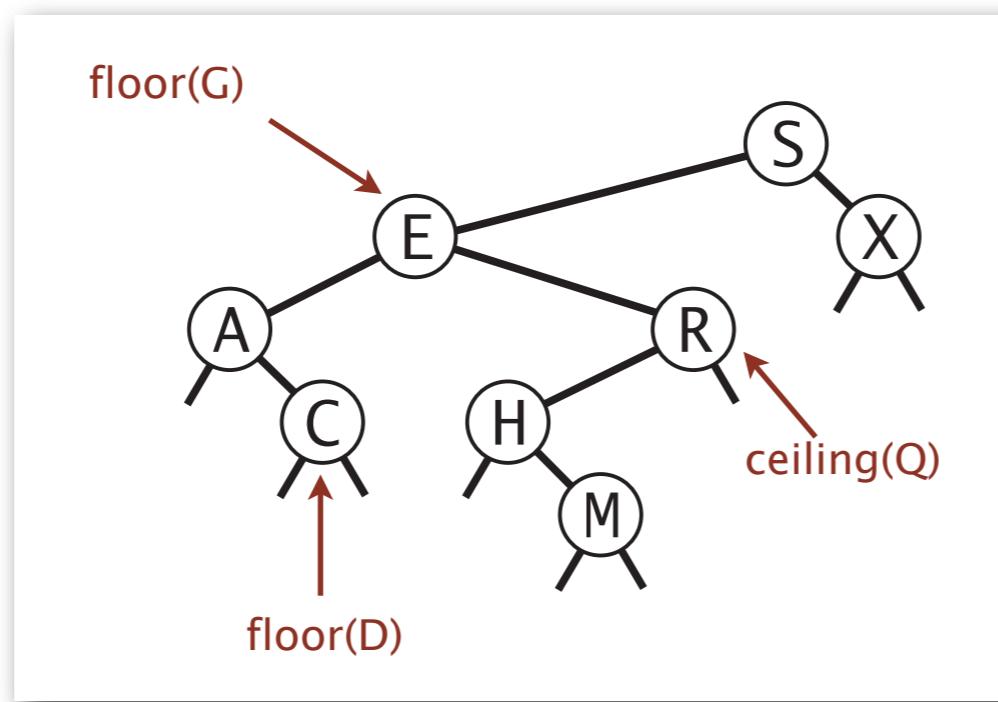


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key \leq to a given key.

Ceiling. Smallest key \geq to a given key.



Q. How to find the floor /ceiling?

Computing the floor

Case 1. [k equals the key at root]

The floor of k is k .

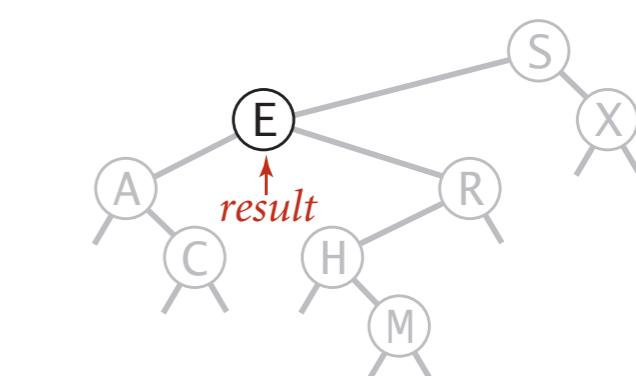
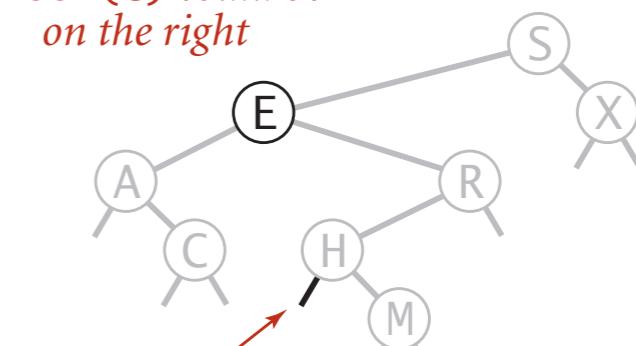
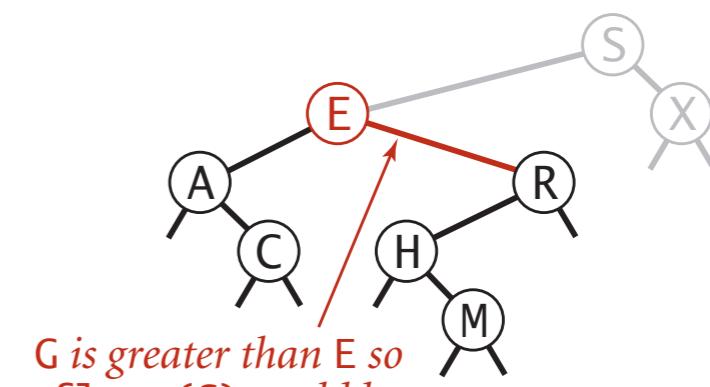
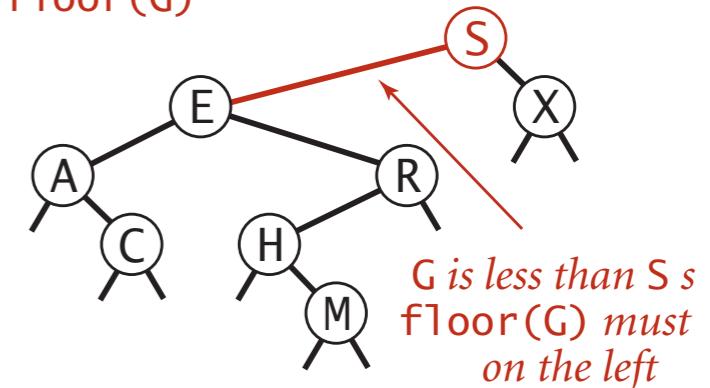
Case 2. [k is less than the key at root]

The floor of k is in the left subtree.

Case 3. [k is greater than the key at root]

The floor of k is in the right subtree
(if there is any key $\leq k$ in right subtree);
otherwise it is the key in the root.

finding floor(G)



Computing the floor

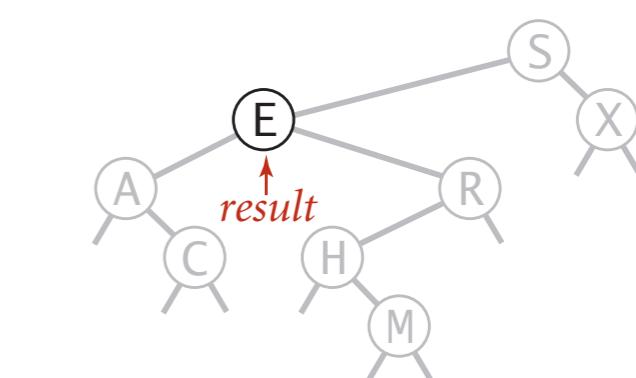
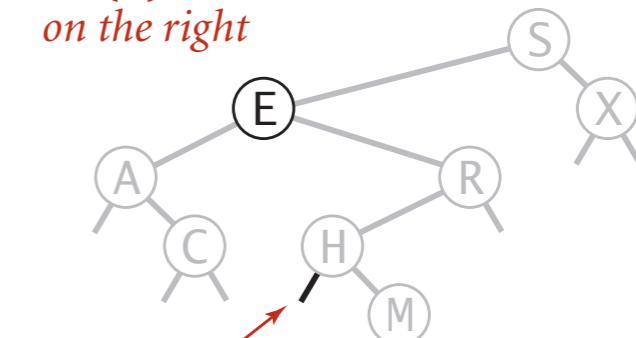
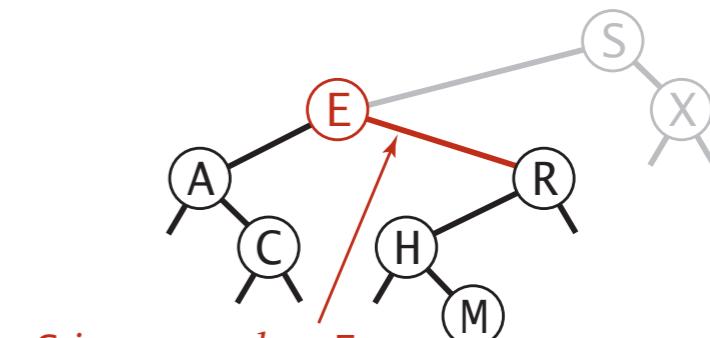
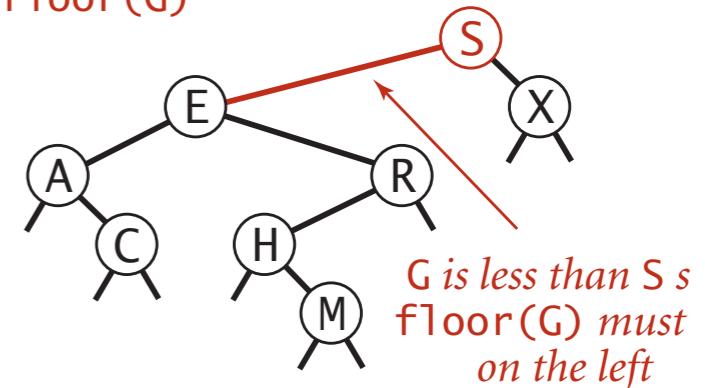
```
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    if (cmp == 0) return x;

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

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

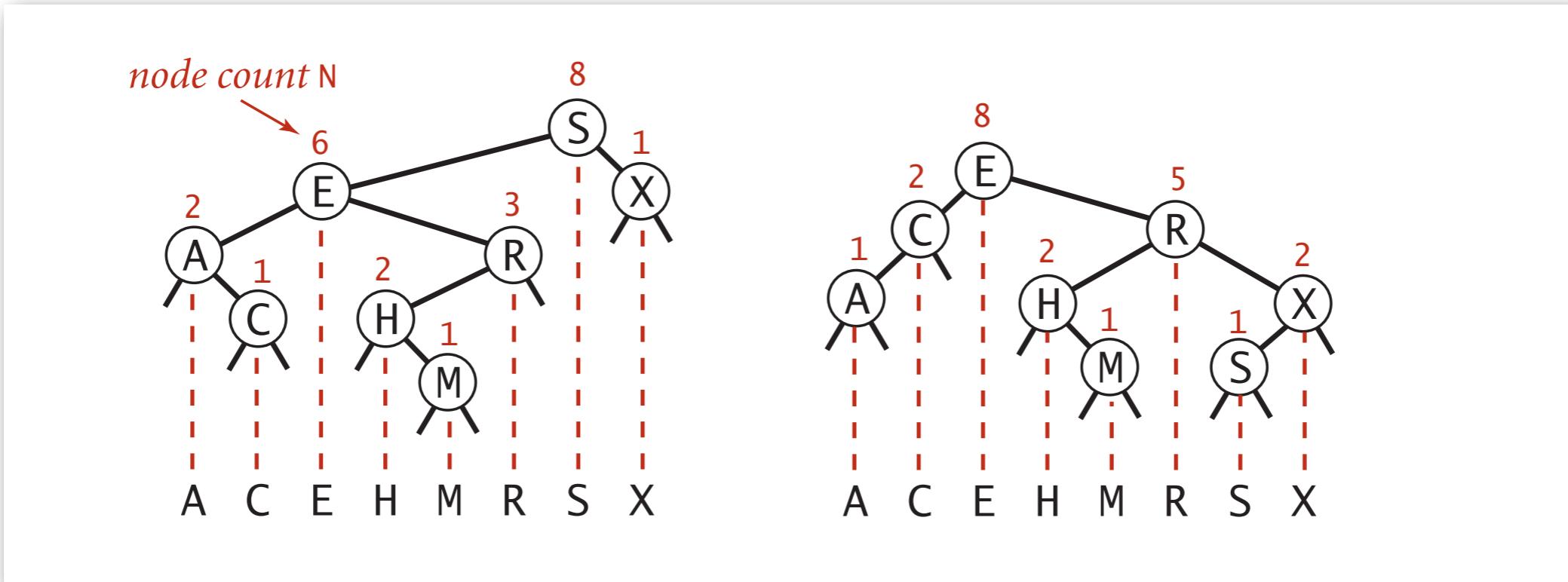
finding floor(G)



Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node.

To implement `size()`, return the count at the root.



Remark. This facilitates efficient implementation of `rank()` and `select()`.

BST implementation: subtree counts

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

nodes in subtree

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

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

```
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;  
    x.N = 1 + size(x.left) + size(x.right);  
    return x;  
}
```

Rank

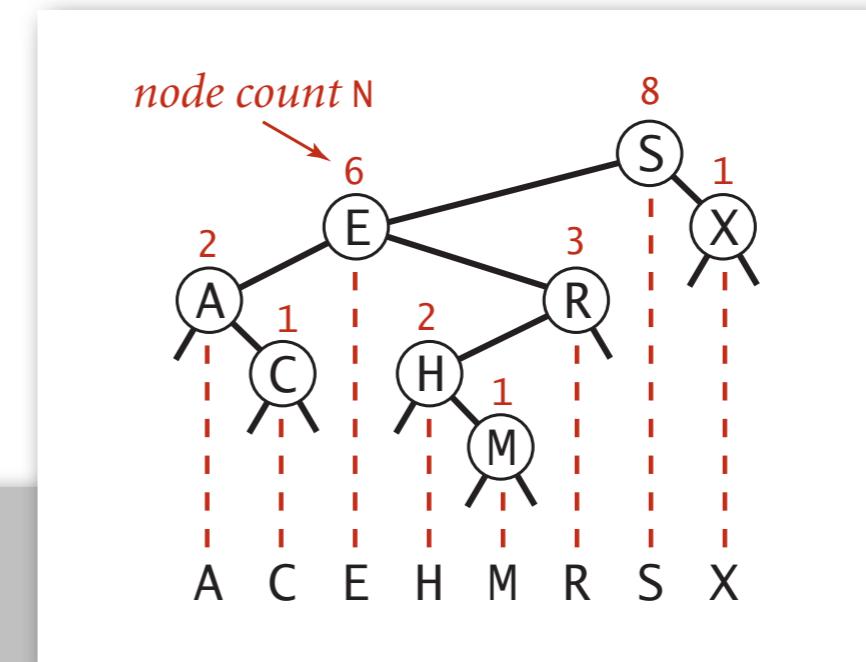
Rank. How many keys $< k$?

Easy recursive algorithm (4 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);
}
```

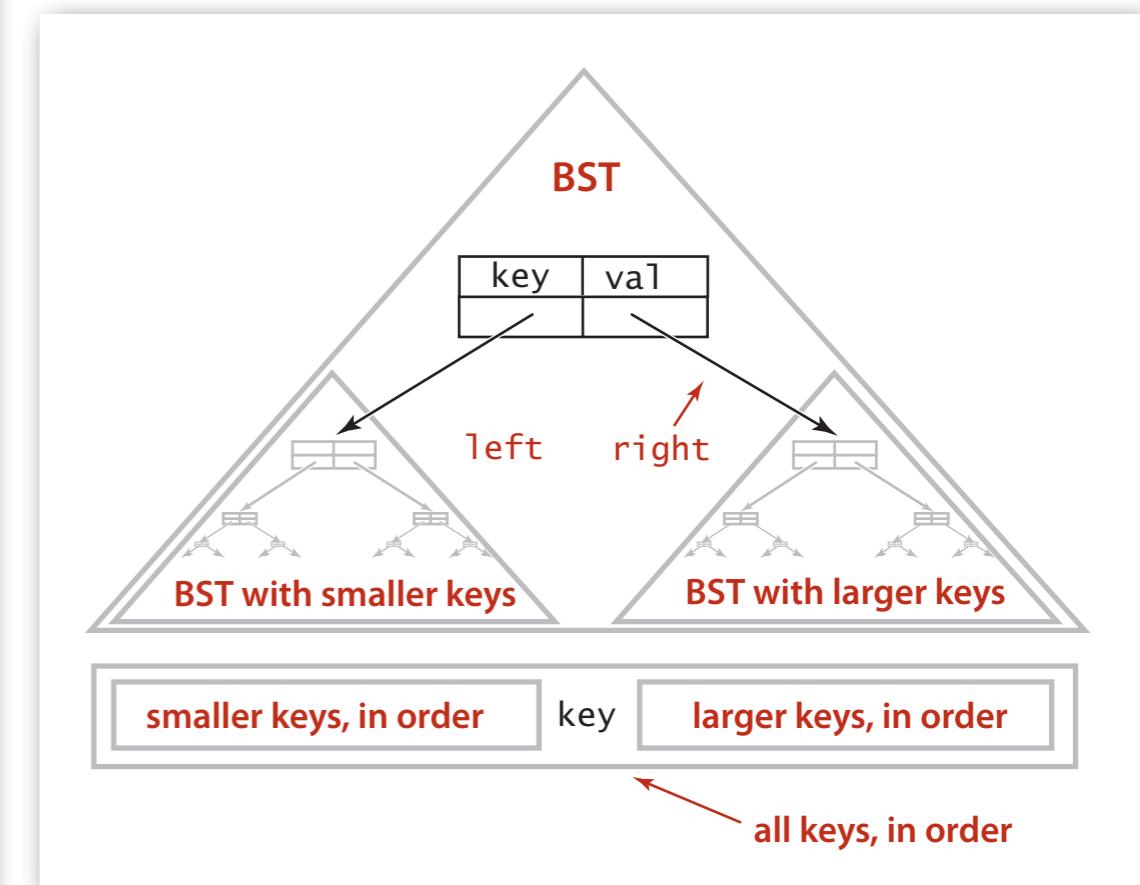


Inorder traversal

- Traverse left subtree.
- Enqueue key.
- 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);
    inorder(x.right, q);
}
```



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

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
inorder(S)
    inorder(E)
        inorder(A)
            enqueue A
        inorder(C)
            enqueue C
        enqueue E
    inorder(R)
        inorder(H)
            enqueue H
        inorder(M)
            enqueue M
    print R
    enqueue S
inorder(X)
    enqueue X
```

A
C
E

H
M
R
S

X

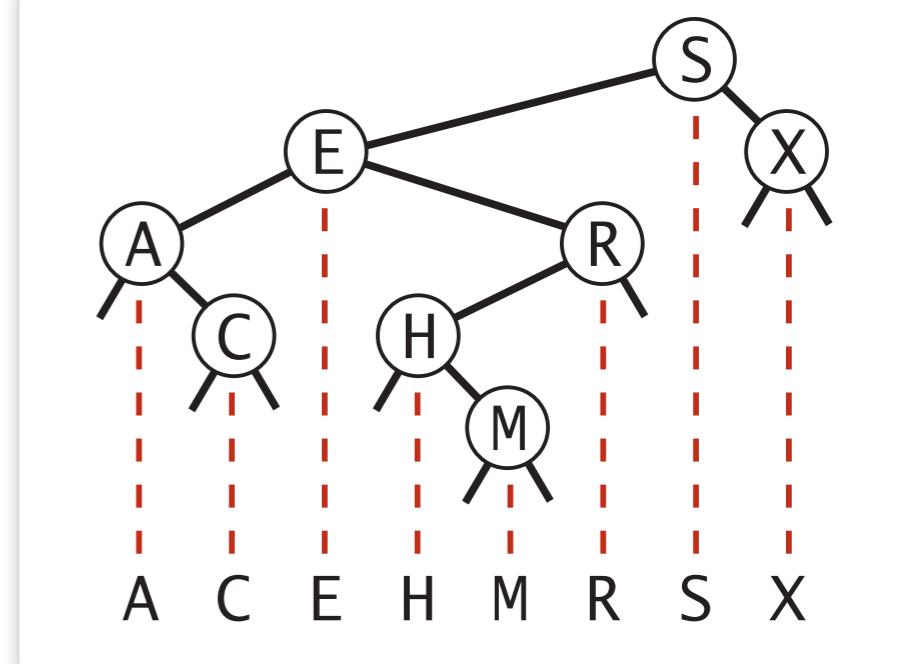
S
S E
S E A

S E A C

S E R
S E R H

S E R H M

S X



recursive calls

queue

function call stack

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	1	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	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)

worst-case running time of ordered symbol table operations

Binary Search Trees

- Trees
- BSTs
- ordered operations
- **deletion**

ST implementations: summary

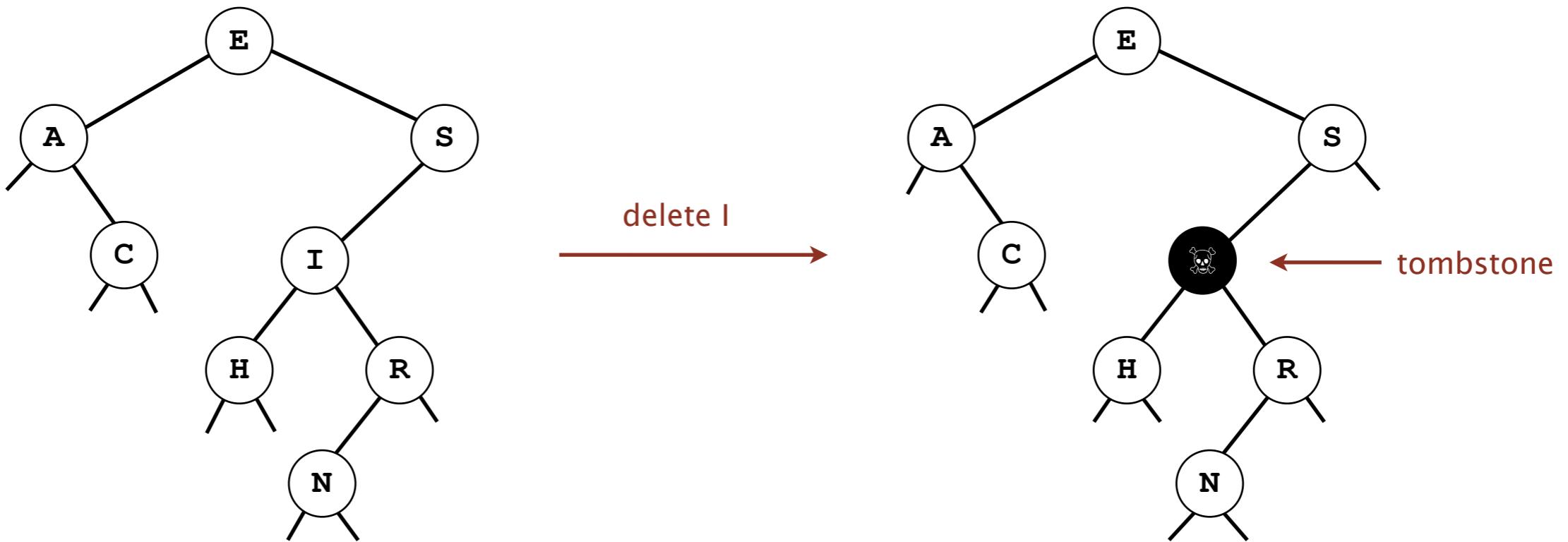
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$???	yes	<code>compareTo()</code>

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 searches (but don't consider it equal to search key).



Cost. $2 \ln N'$ 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 overload.

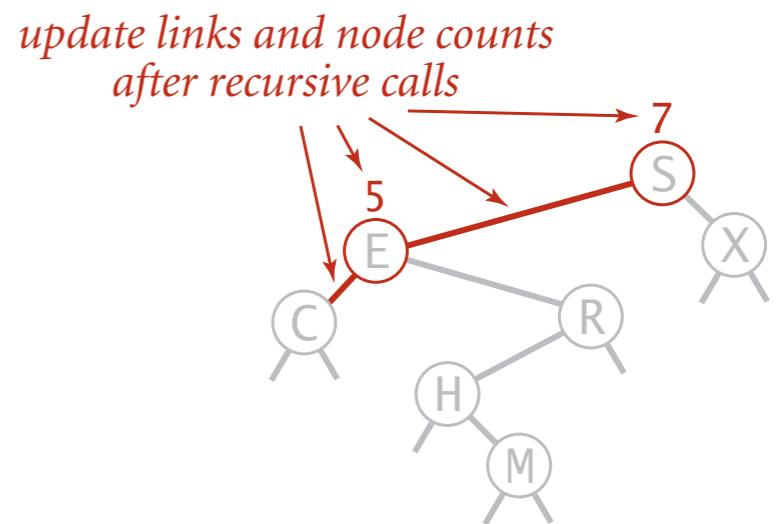
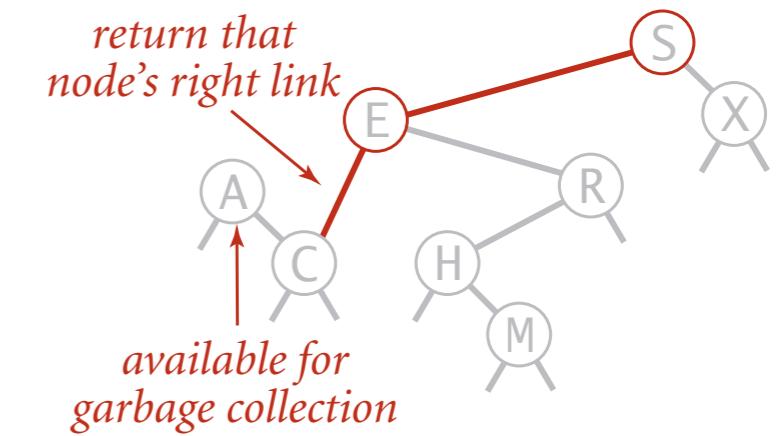
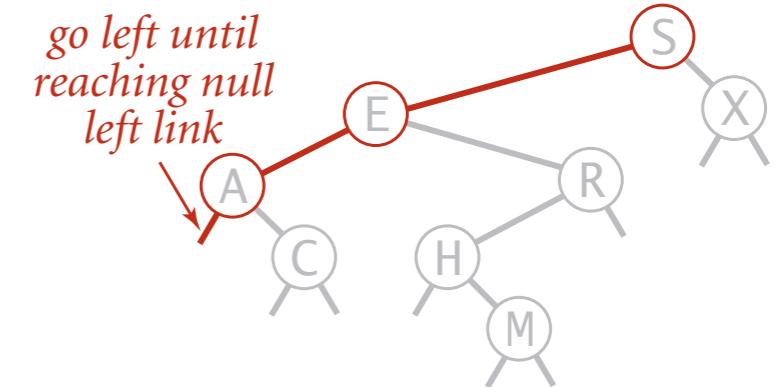
Deleting the minimum

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);   }

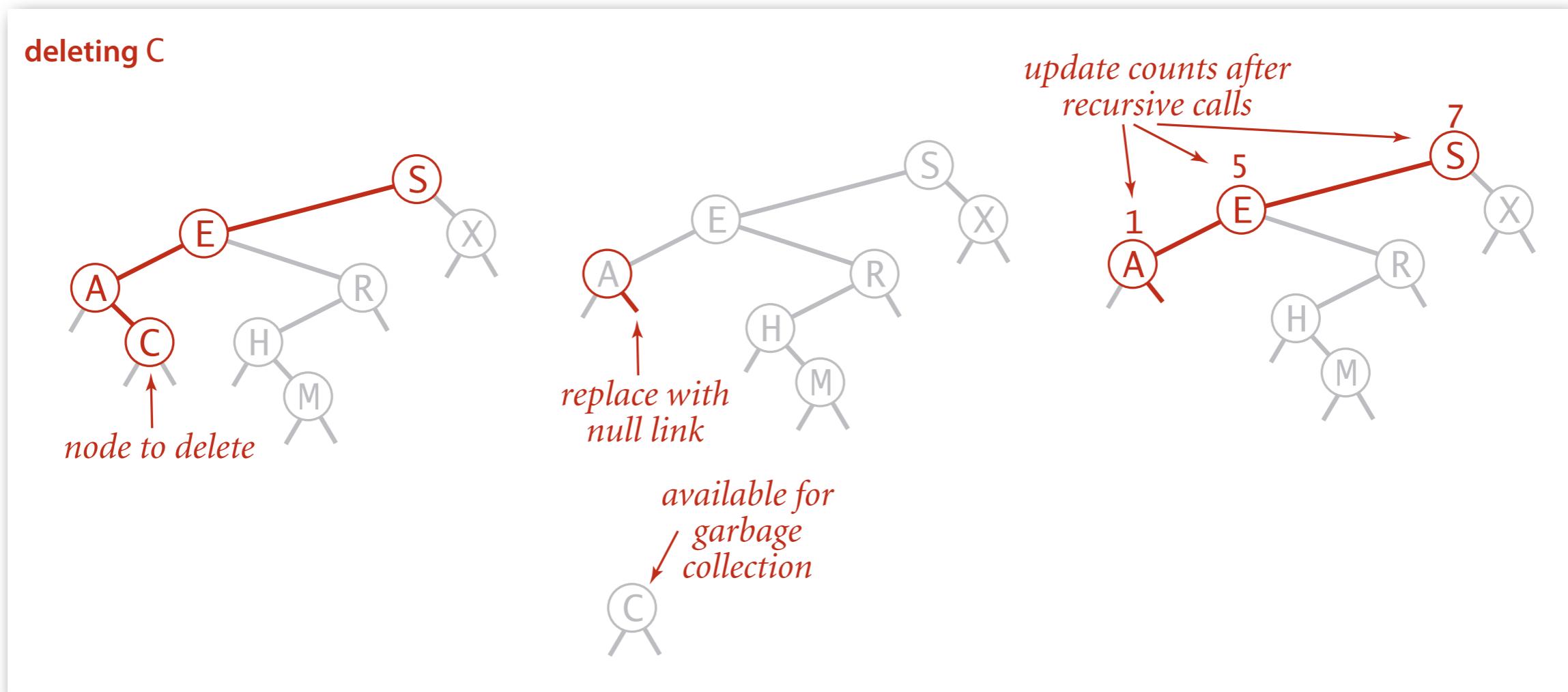
private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```



Hibbard deletion

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

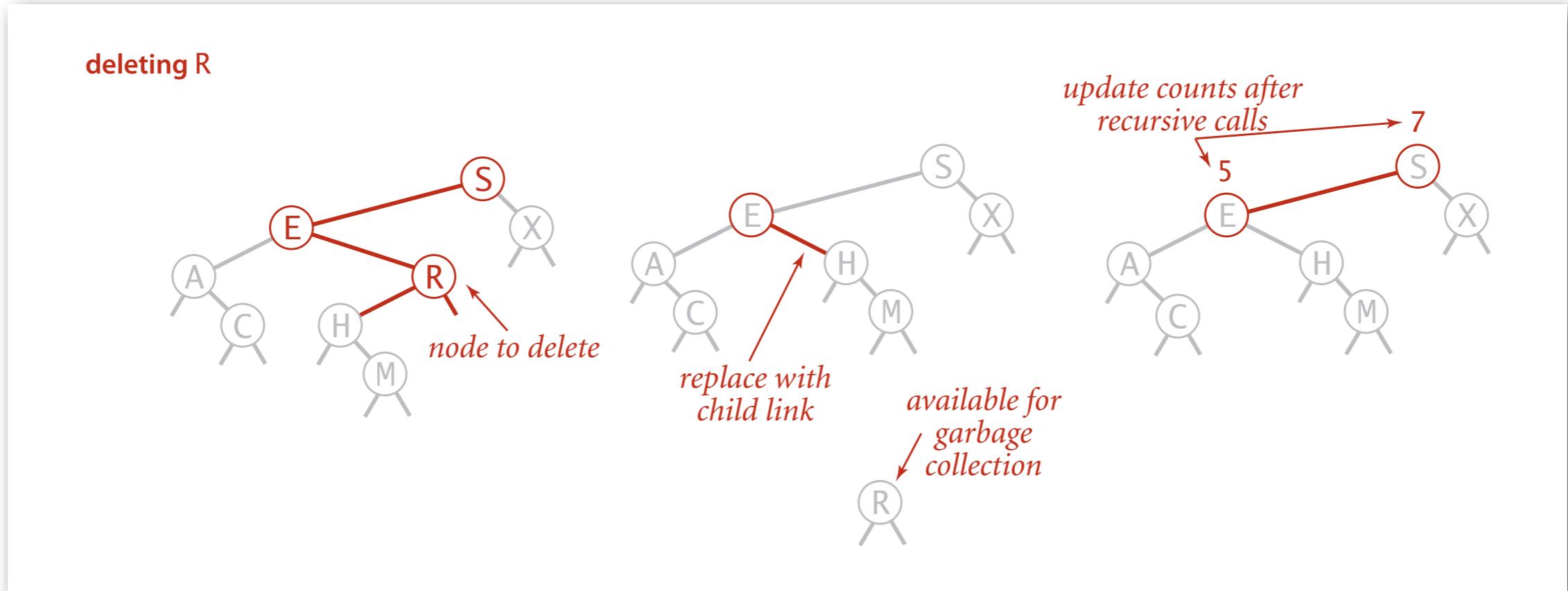
Case 0. [0 children] Delete t by setting parent link to null.



Hibbard deletion

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

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



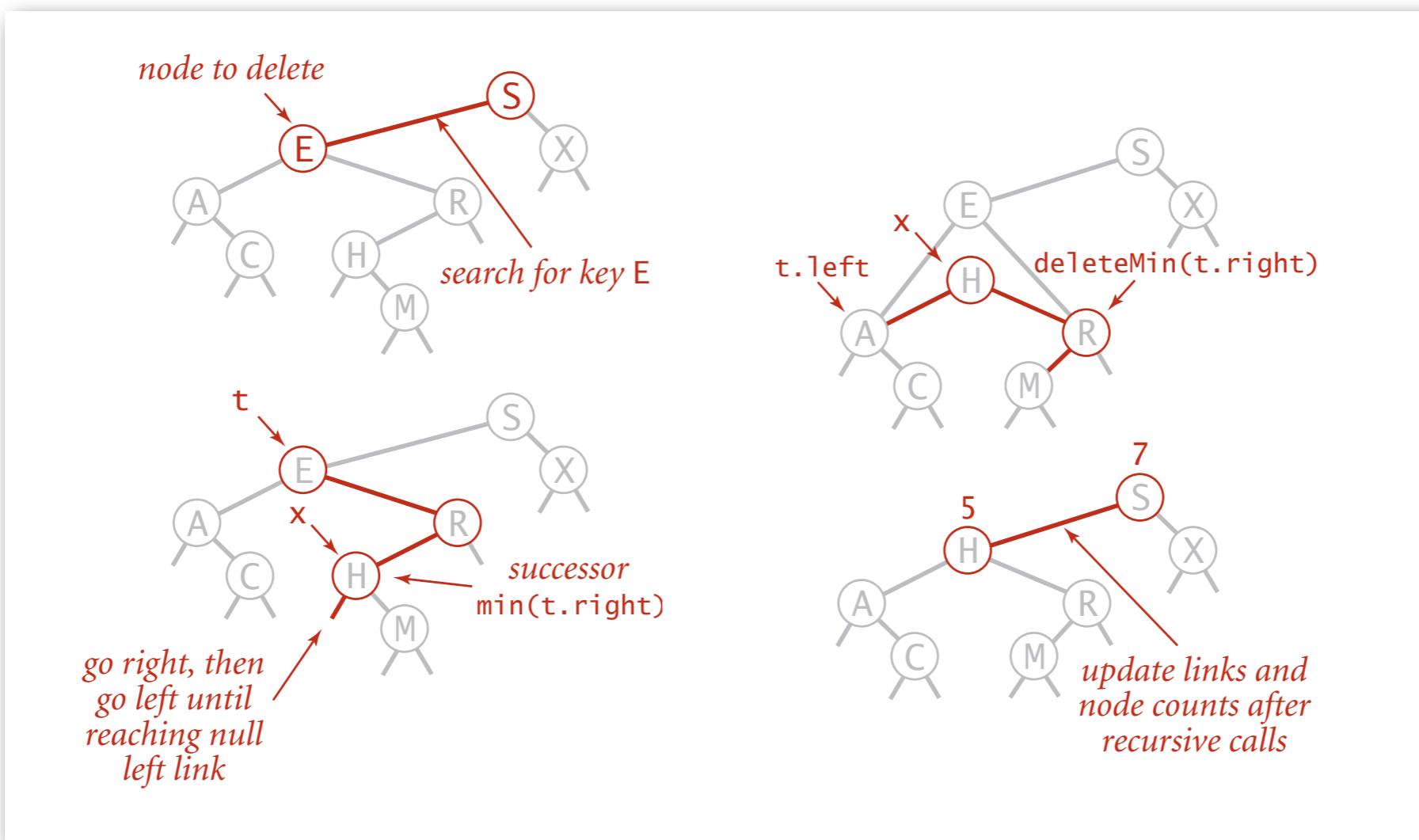
Hibbard deletion

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

Case 2. [2 children]

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

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



Hibbard deletion: Java implementation

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

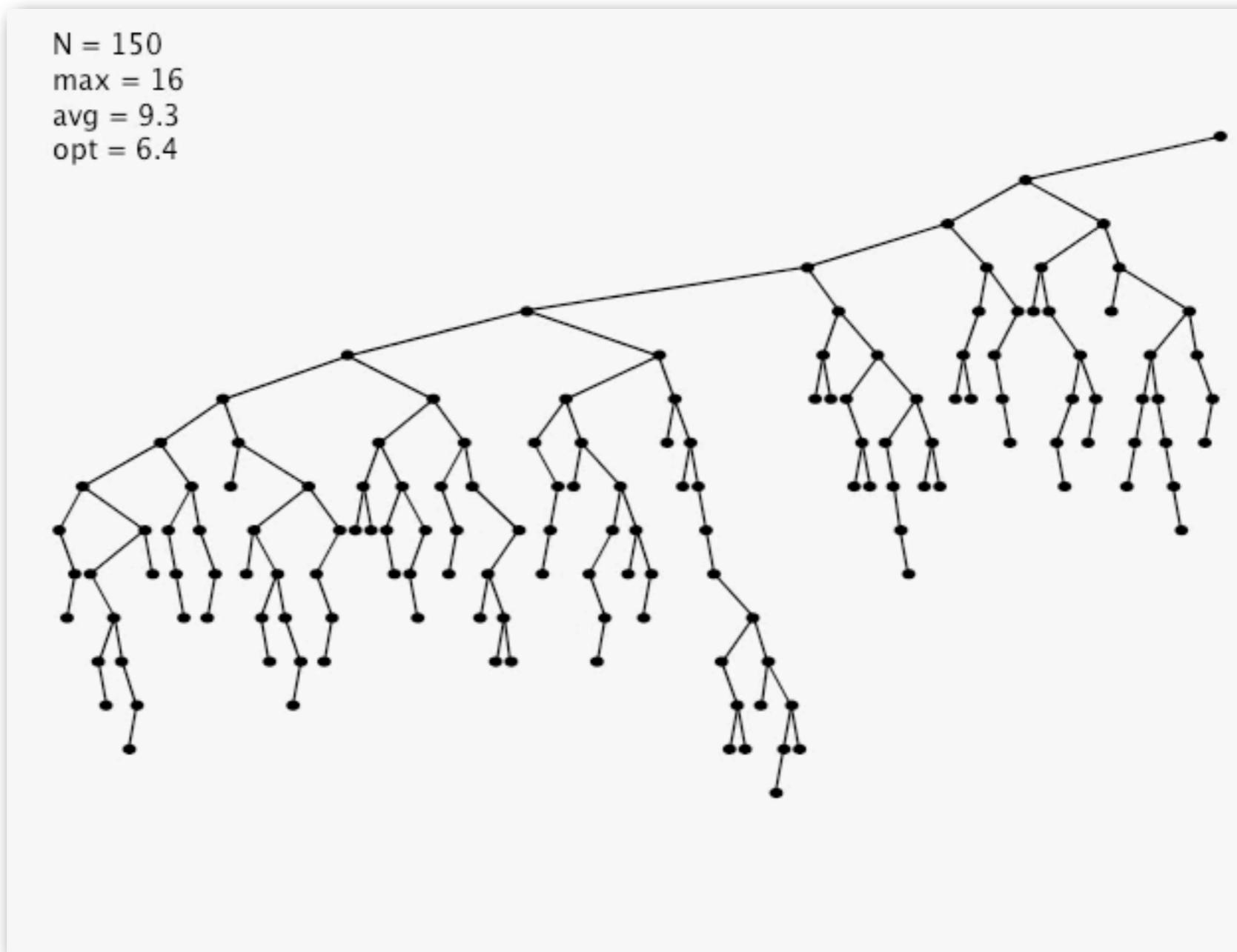
private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left; ← no right child
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right); ← replace with successor
        x.left = t.left;
    }
    x.N = size(x.left) + size(x.right) + 1; ← update subtree counts
    return x;
}
```

The code is annotated with red arrows pointing to specific lines of code:

- A red arrow points to the line `x.right = delete(x.right, key);` with the label "search for key".
- A red arrow points to the line `if (x.right == null) return x.left;` with the label "no right child".
- A red arrow points to the line `x.right = deleteMin(t.right);` with the label "replace with successor".
- A red arrow points to the line `x.N = size(x.left) + size(x.right) + 1;` with the label "update subtree counts".

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.

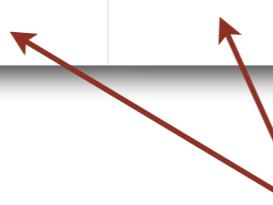


Surprising consequence. Trees not random (!) $\Rightarrow \sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	yes	<code>compareTo()</code>



other operations also become \sqrt{N}
if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.