

Symbol Tables

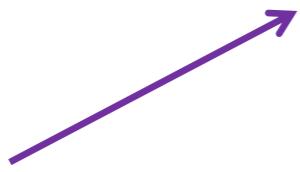
Binary Search Trees

Dr. 何明晰, He Mingxin, Max

program06 @ yeah.net

Email Subject: (L1-|L2-|L3-) + *last 4 digits of ID* + *Name: TOPIC*

Your Lab Class



Sakai: CS203B Fall 2022

数据结构与算法分析B

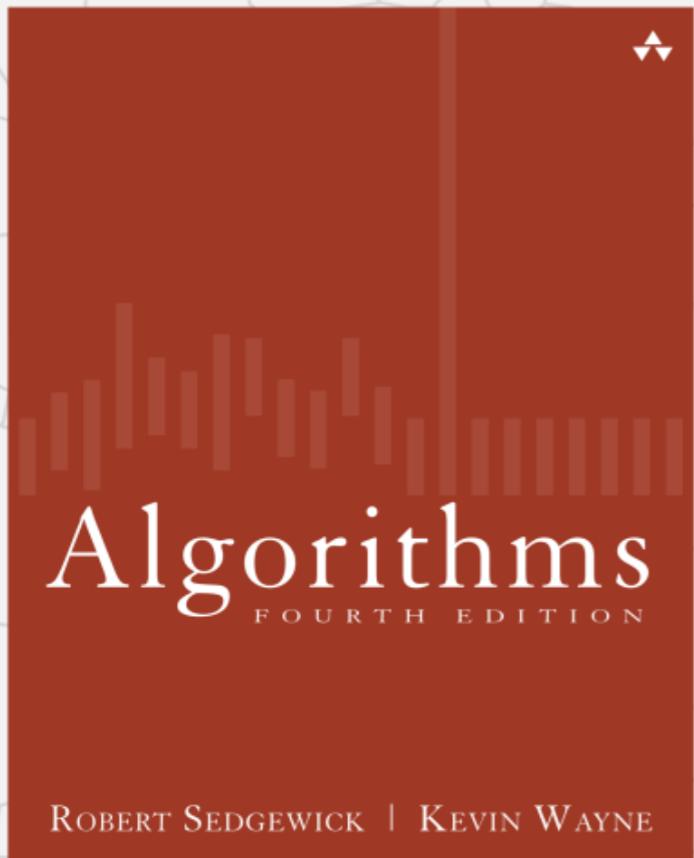
Data Structures and Algorithm Analysis

Lecture 10

- Symbol Tables (3.1 of Text A)
- Binary Search Trees (3.2 of Text A)

To be discussed in Lecture 11:

- Balanced Search Trees (3.3 of Text A)
- Hash Tables (3.4 of Text A)



ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

3.2 BINARY SEARCH TREES

- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

3.2 BINARY SEARCH TREES

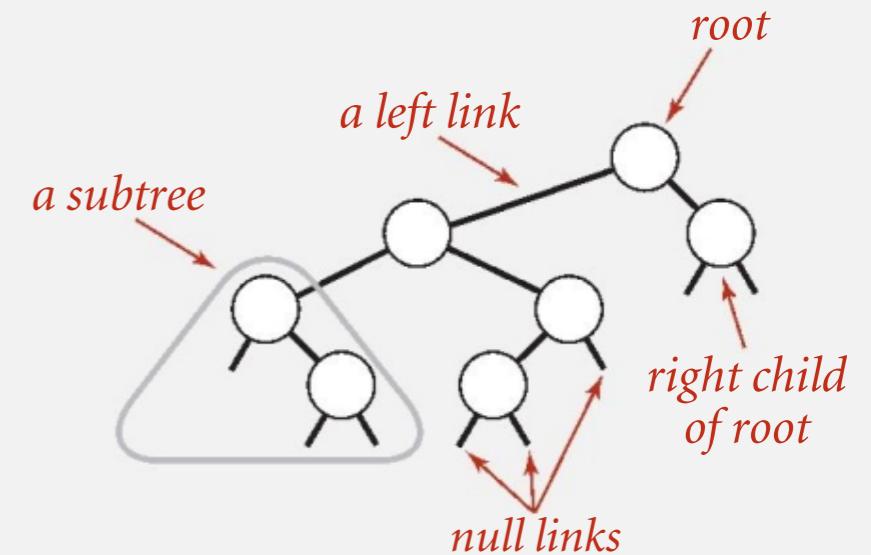
- ▶ BSTs → 允许有重复的键对.
- ▶ ordered operations
- ▶ deletion

Binary search trees

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

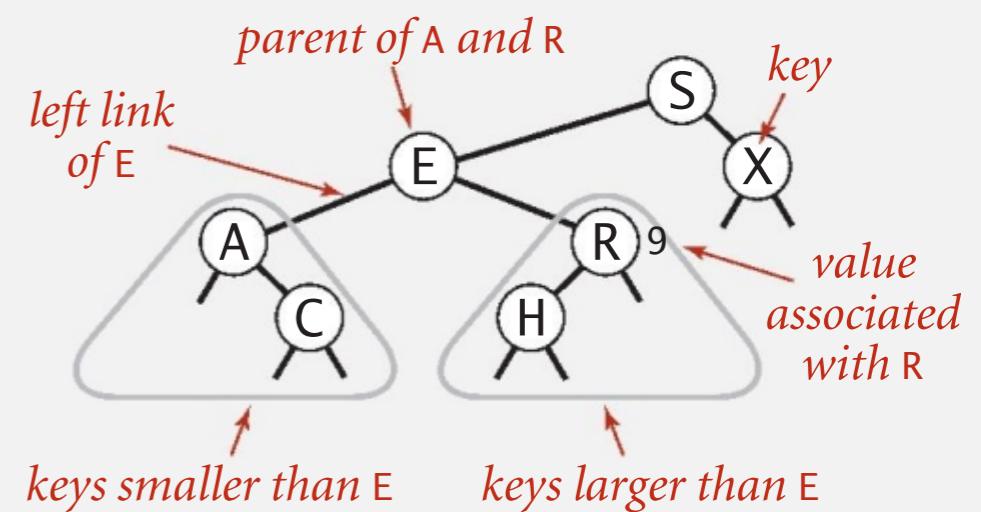
A binary tree is either:

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



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.

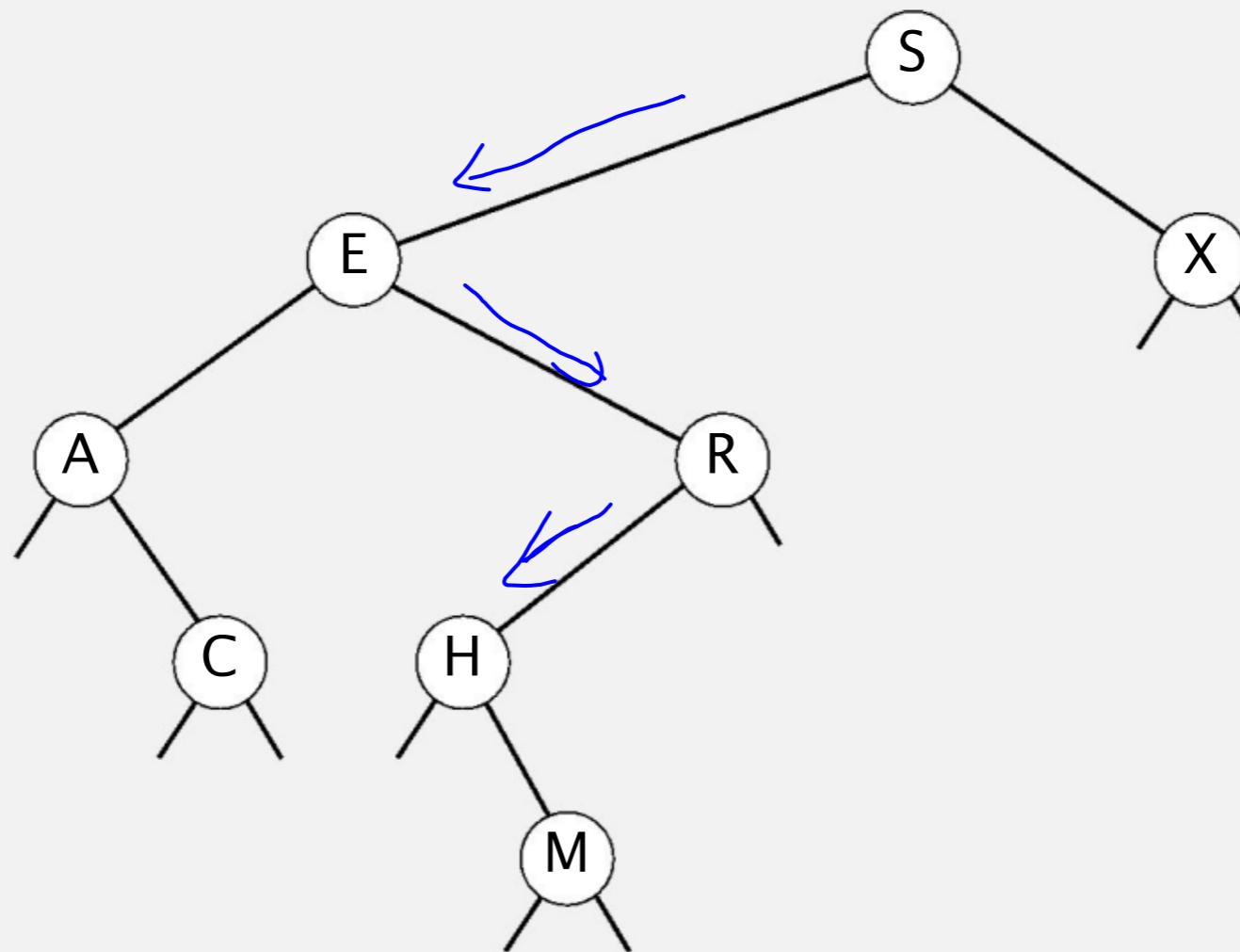


Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

<https://algs4.cs.princeton.edu/lectures/demo/32DemoBinarySearchTree.mov>

successful search for H

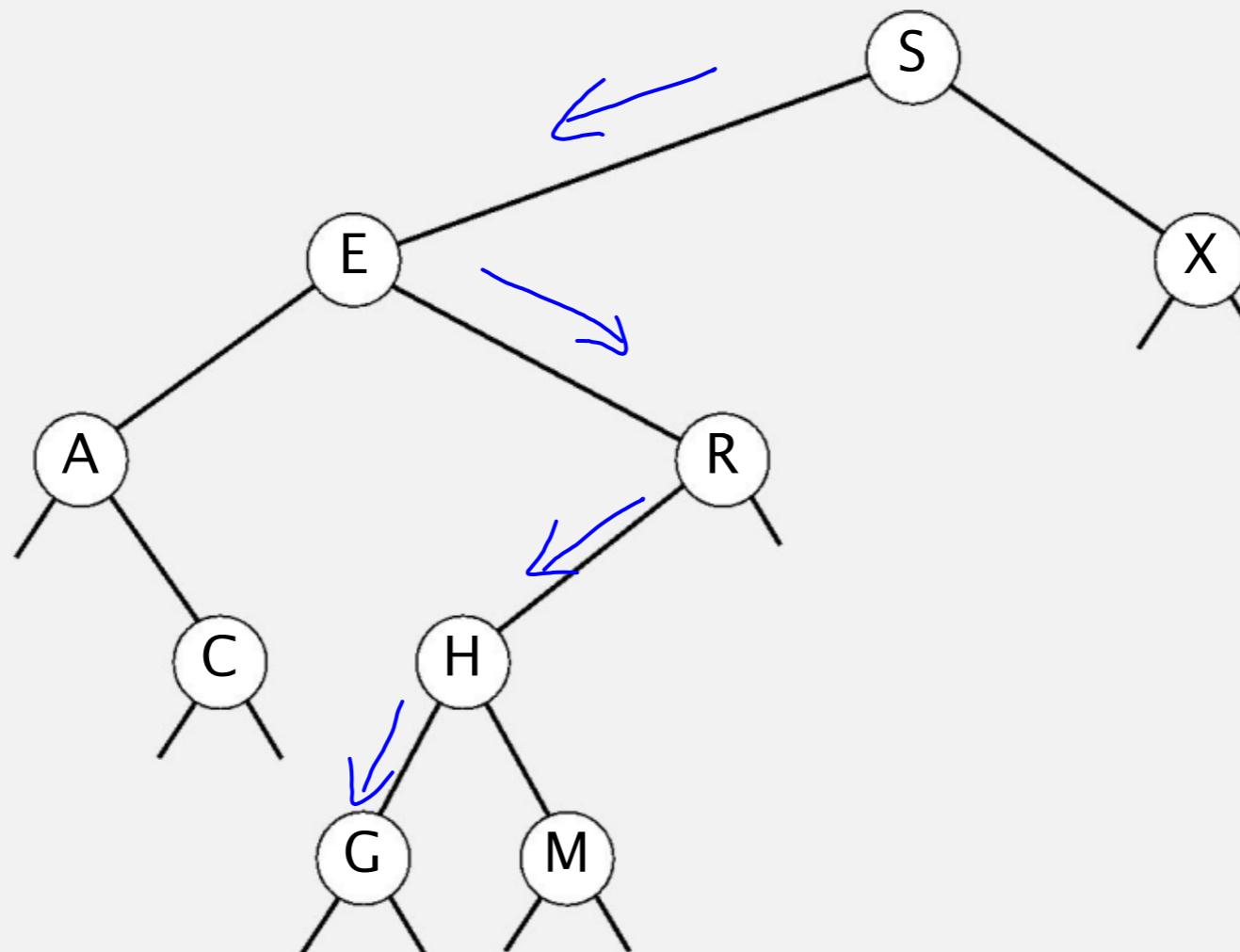


Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.

<https://algs4.cs.princeton.edu/lectures/demo/32DemoBinarySearchTree.mov>

insert G



BST representation in Java

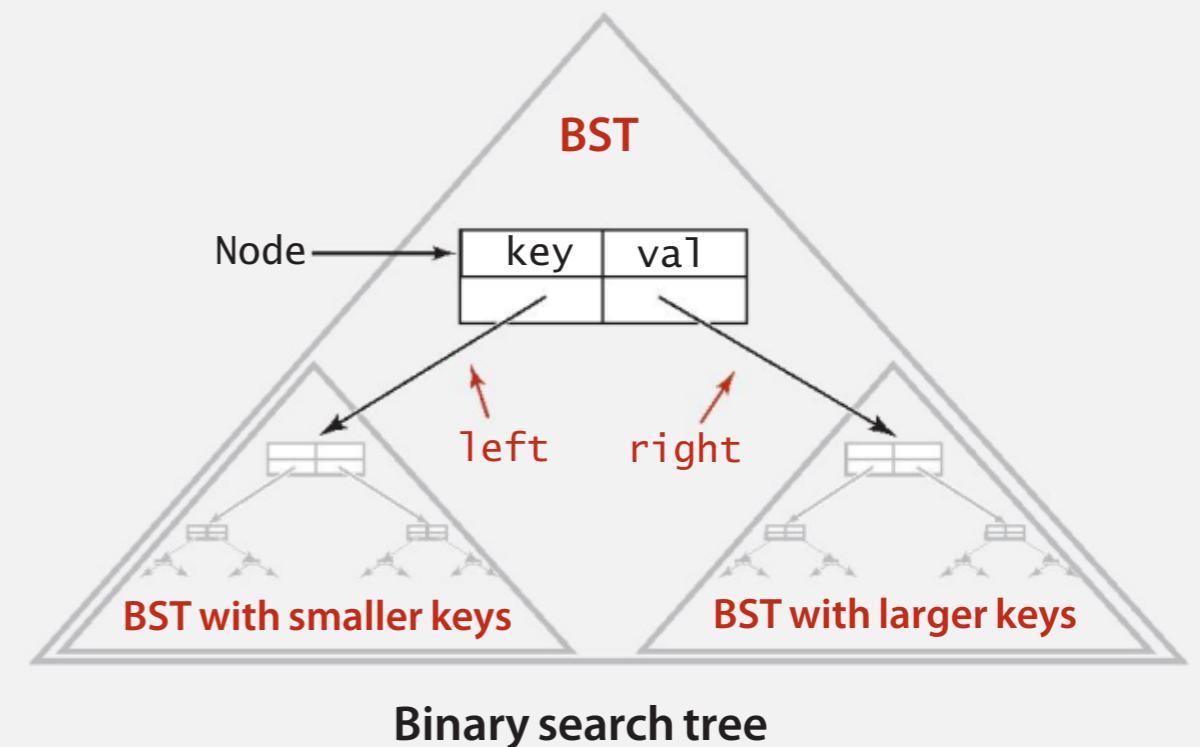
Java definition. A BST is a reference to a root Node.

A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree. 还没有 size n.

smaller keys larger keys

```
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;
```

← root of BST

```
    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 */ }
```

```
}
```

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 $1 + \text{depth of node.}$

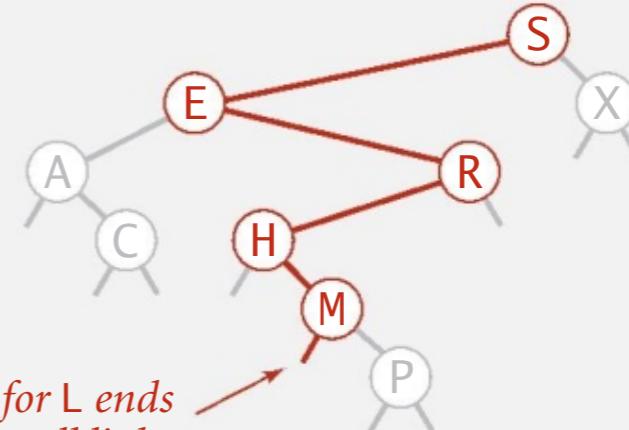
BST insert

Put. Associate value with key.

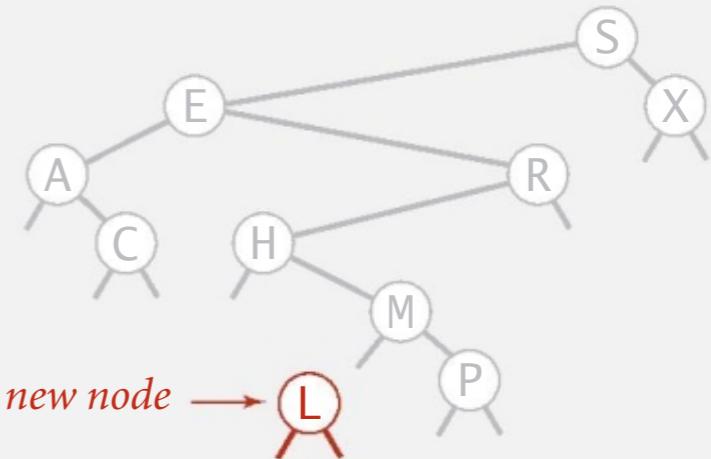
Search for key, then two cases:

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

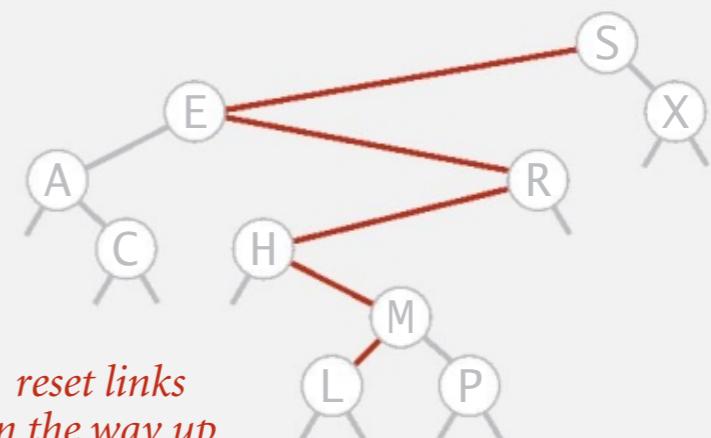
inserting L



*search for L ends
at this null link*



create new node



*reset links
on the way up*

Insertion into a BST

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;
}
```

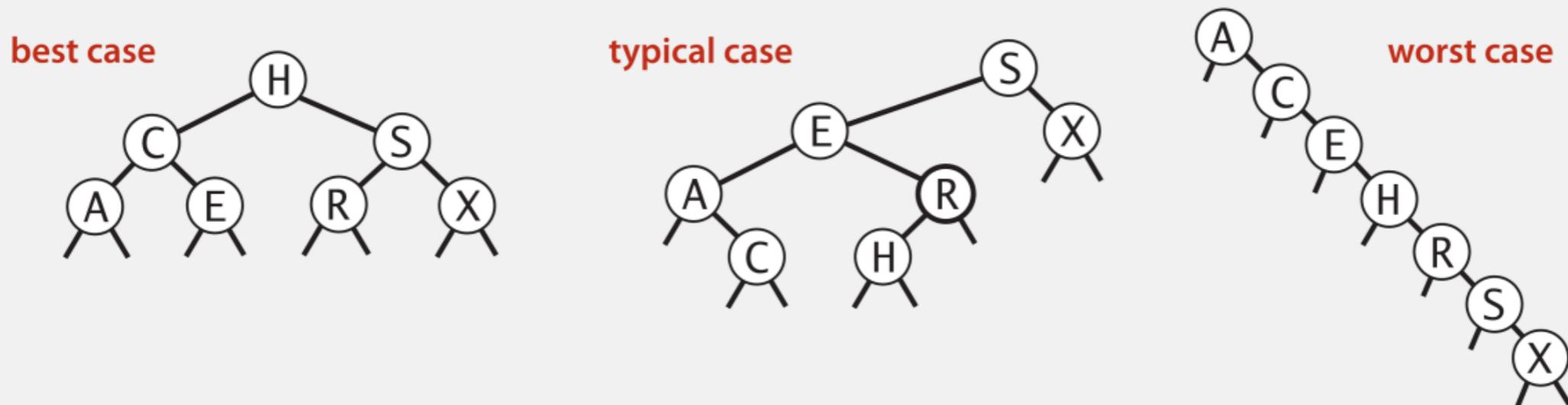
concise, but tricky,
recursive code;
read carefully!

→ 有点困难

Cost. Number of compares is equal to 1 + depth of node.

Tree shape

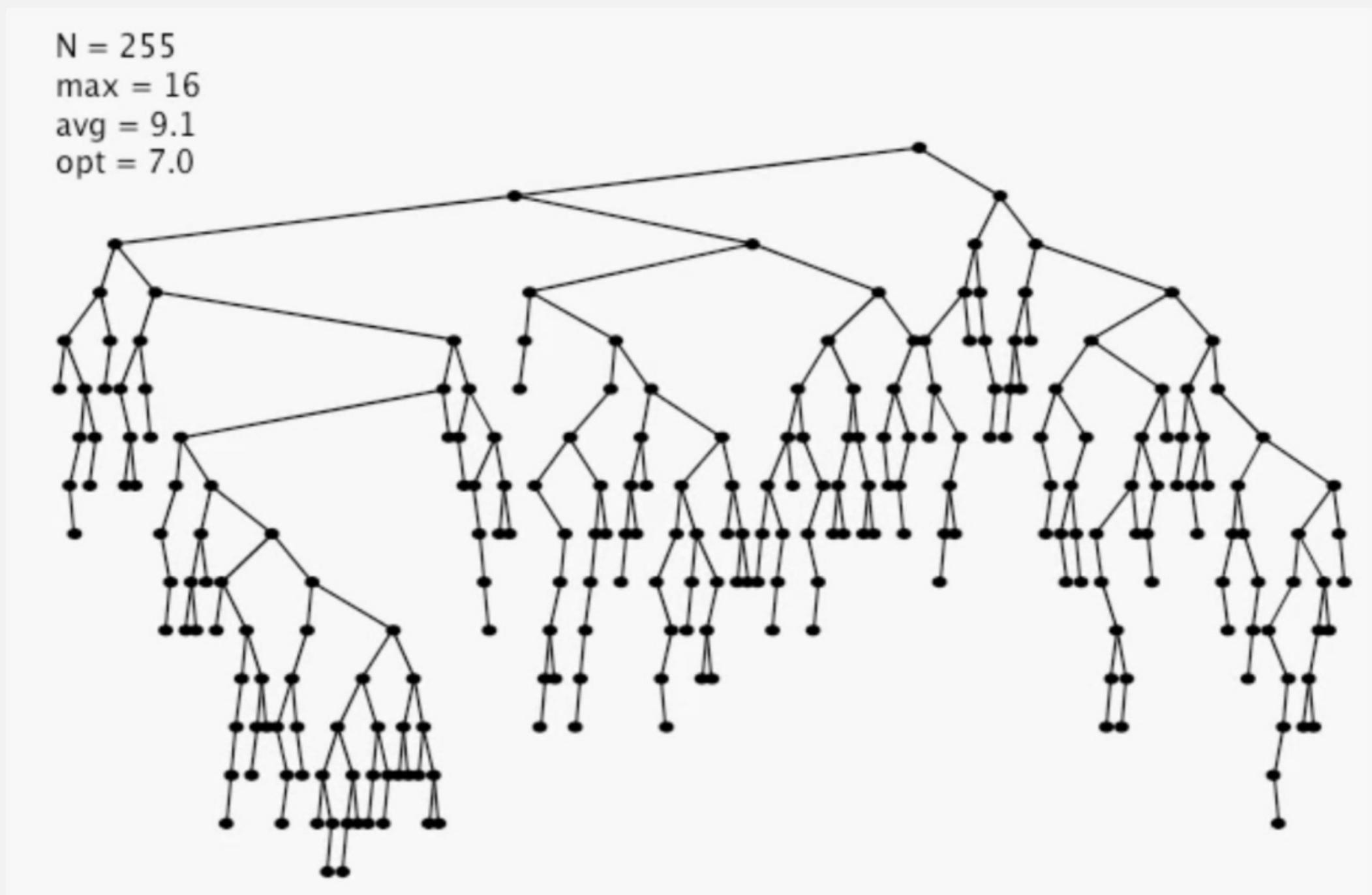
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.



Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

Ex. Insert keys in random order.



Sorting with a binary heap

Q. What is this sorting algorithm?

0. Shuffle the array of keys.
1. Insert all keys into a BST.
2. Do an inorder traversal of BST.

A. It's not a sorting algorithm (if there are duplicate keys)!

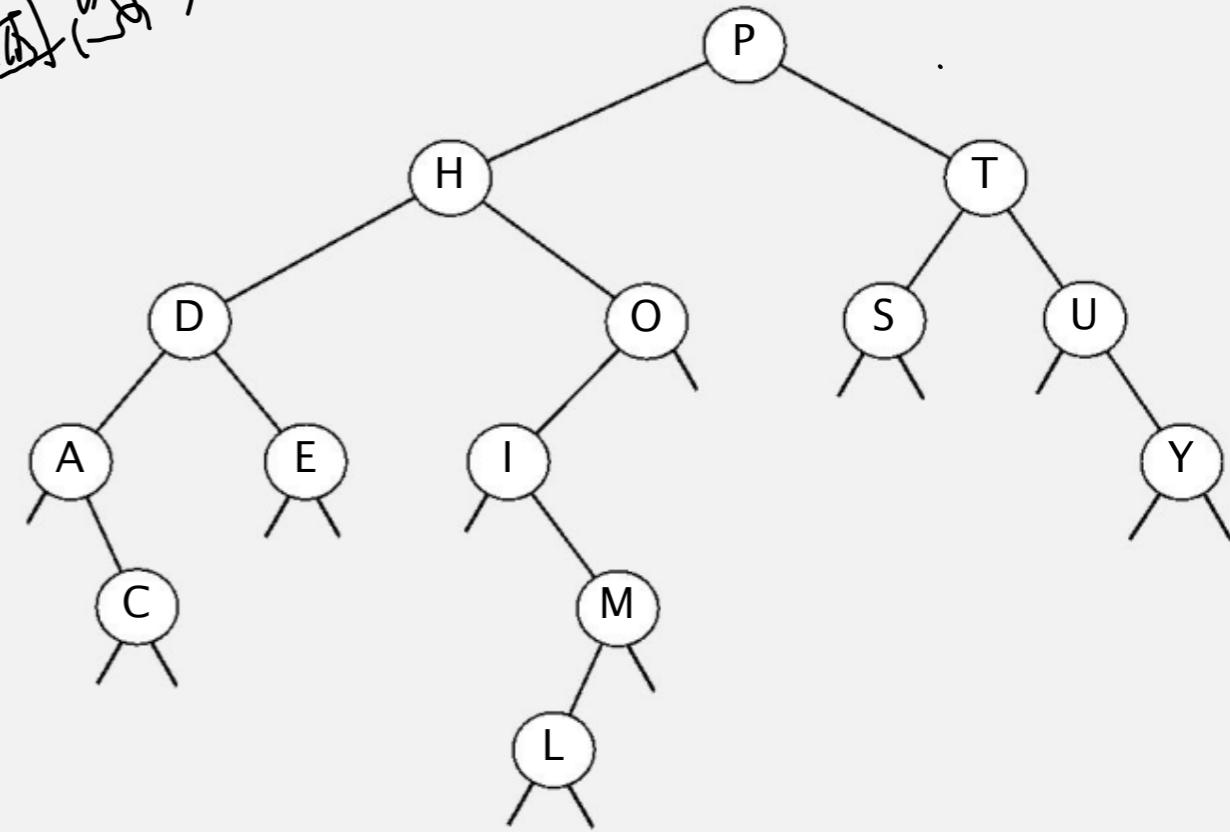
Q. OK, so what if there are no duplicate keys?

Q. What are its properties?

Correspondence between BSTs and quicksort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y

→ ~~13~~ 13



线序遍历

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

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.

1-1
对应

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

How Tall is a Tree?

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

ABSTRACT

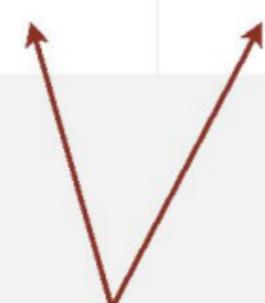
Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107 \dots$ and $\beta = 1.95 \dots$ such that $E(H_n) = \alpha \log n - \beta \log \log n + O(1)$, We also show that $\text{Var}(H_n) = O(1)$.

But... Worst-case height is N .

[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	$\frac{1}{2} N$	N	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	$\lg N$	$\frac{1}{2} N$	<code>compareTo()</code>
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	<code>compareTo()</code>



Why not shuffle to ensure a (probabilistic) guarantee of $4.311 \ln N$?

在由人手随机构造的二叉查找树中插入操作和查找命中平均所需的比较次数为 $2\ln N$

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

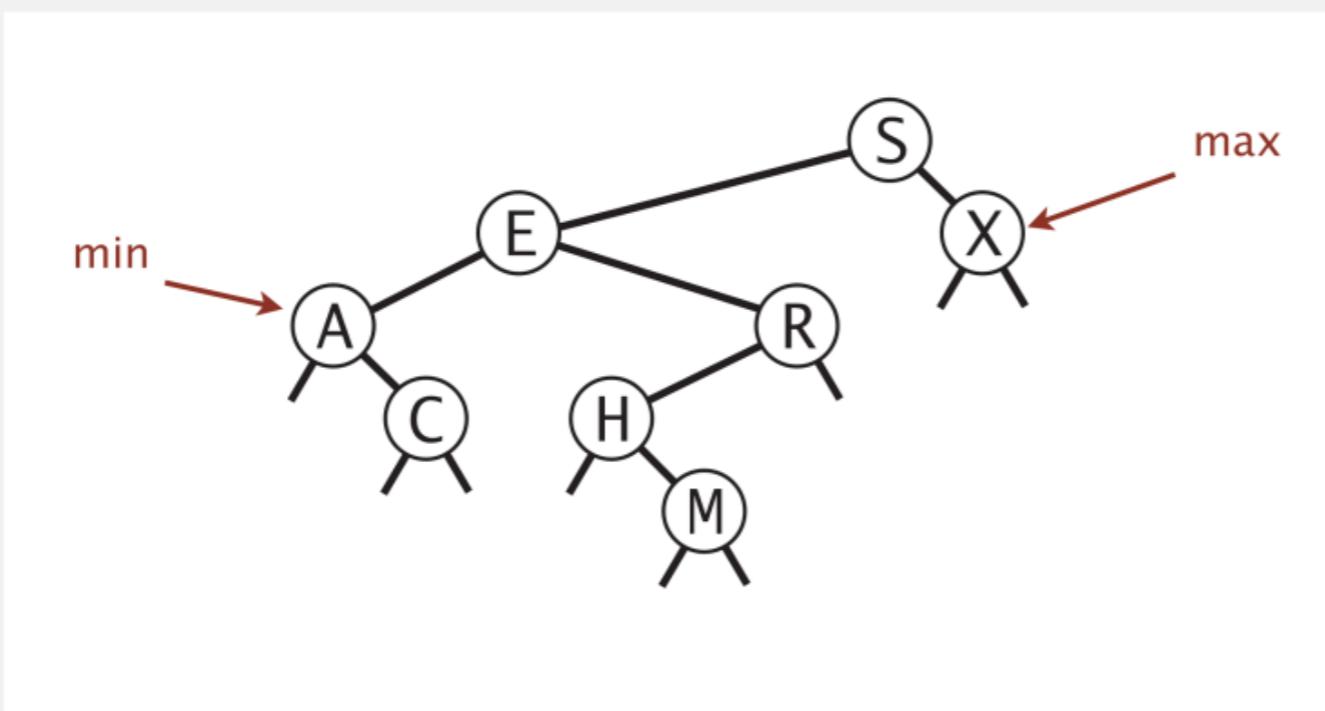
3.2 BINARY SEARCH TREES

- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

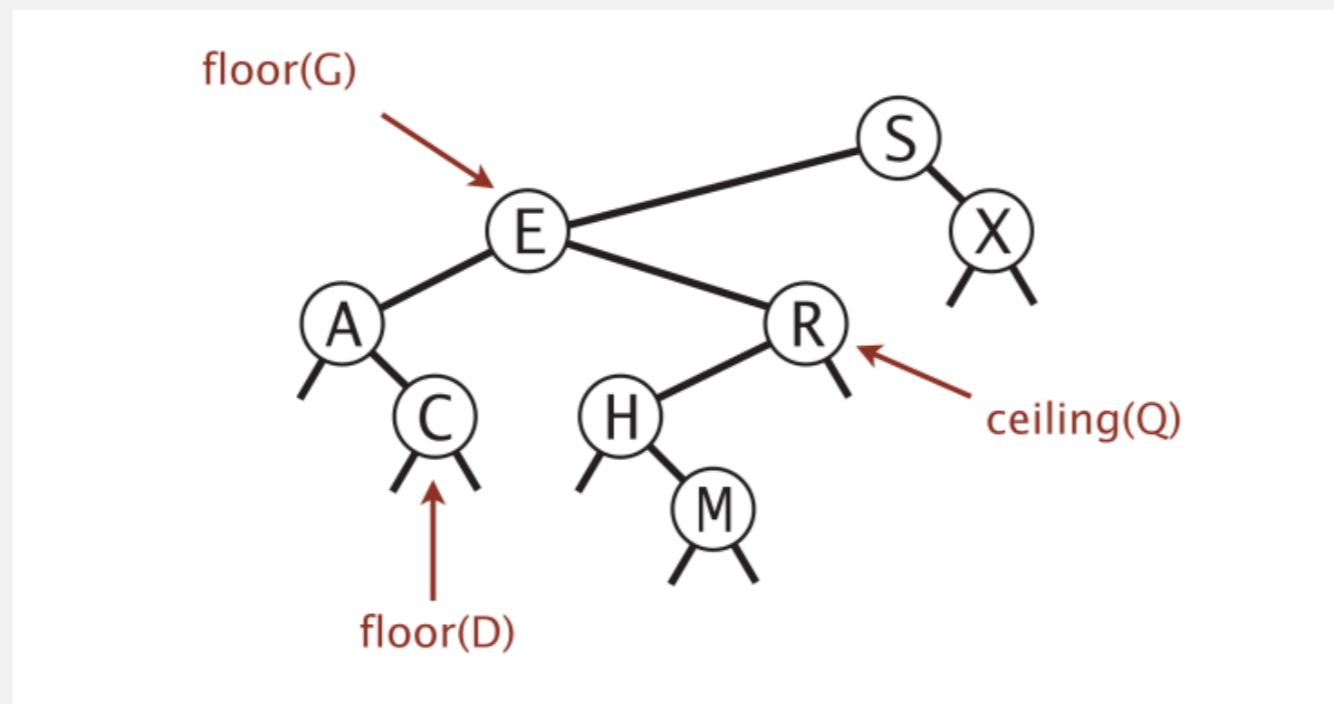


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key \leq a given key.

Ceiling. Smallest key \geq a given key.



Q. How to find the floor / ceiling?

Computing the floor

Case 1. [k equals the key in the node]

The floor of k is k .

Case 2. [k is less than the key in the node]

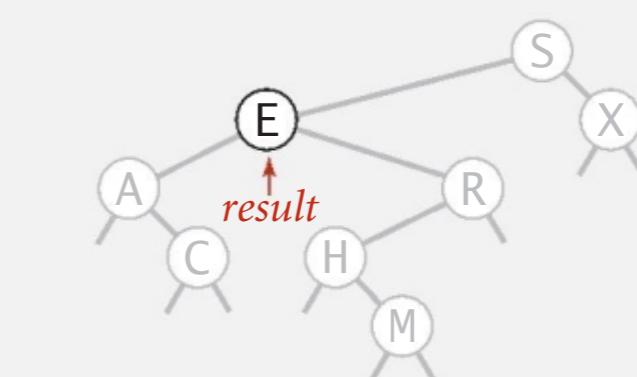
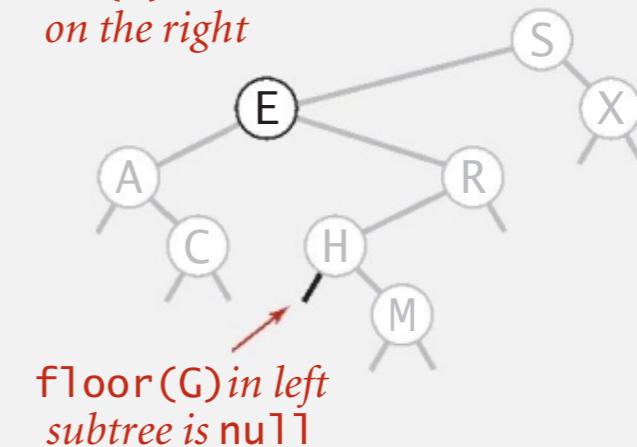
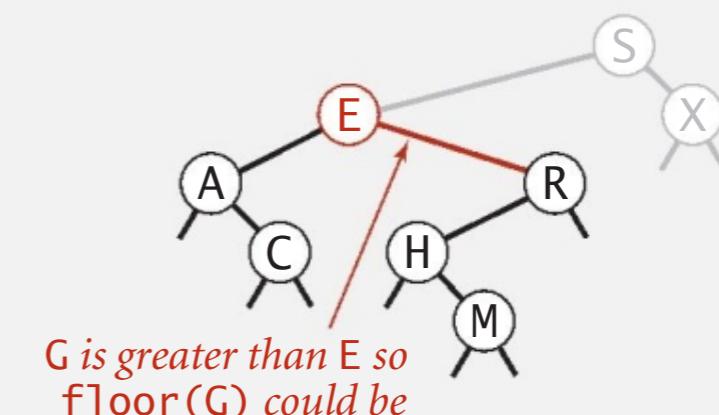
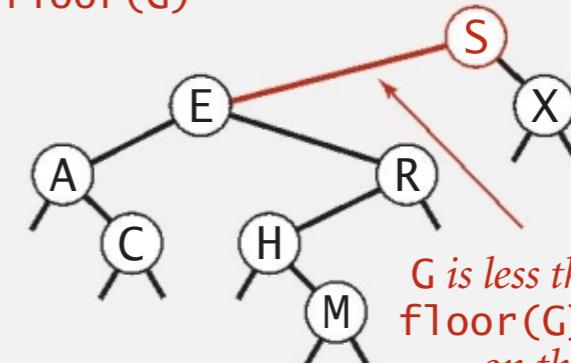
The floor of k is in the left subtree.

Case 3. [k is greater than the key in the node]

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 node.

finding floor(G)



Computing the floor

两个方法对应该有方法

方法一

递归

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

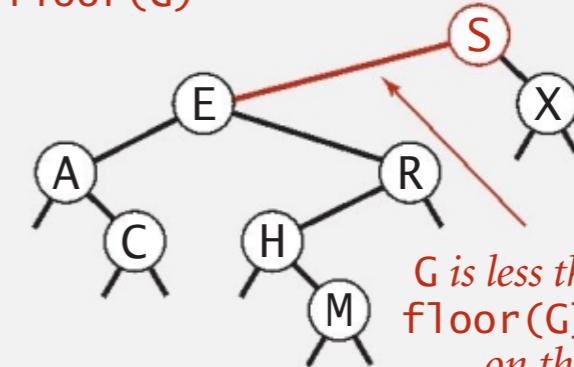
```
private Node floor (Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    if (cmp == 0) return x;

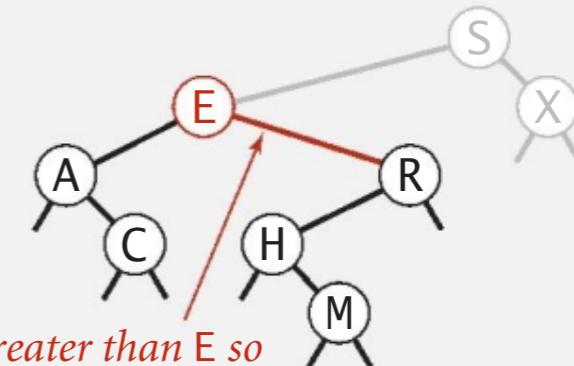
    if (cmp < 0) return floor(x.left, key);

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

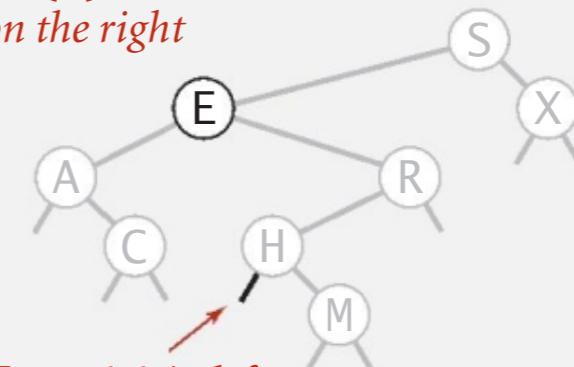
finding floor(G)



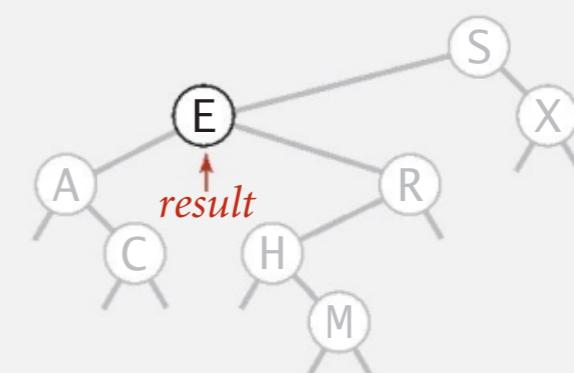
G is less than S so
floor(G) must be
on the left



G is greater than E so
floor(G) could be
on the right



floor(G) in left
subtree is null

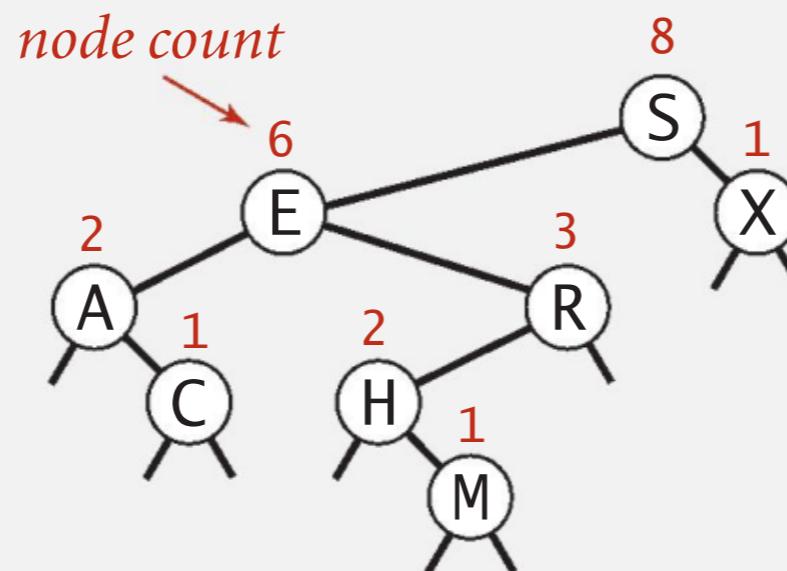


result

Rank and select

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

A. 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.



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

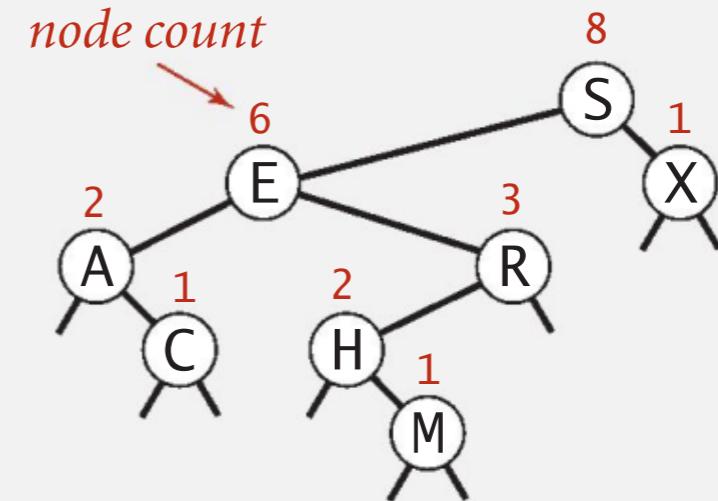
```
private Node put (Node x, Key key, Value val) {    initialize subtree  
                                                    count to 1  
  
    if (x == null) return new Node(key, val, 1);  
    int cmp = key.compareTo( x.key);  
    if      (cmp  < 0) x.left  = put( x.left,  key, val);  
    else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

看

Rank

Rank. How many keys $< k$?

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);
}
```

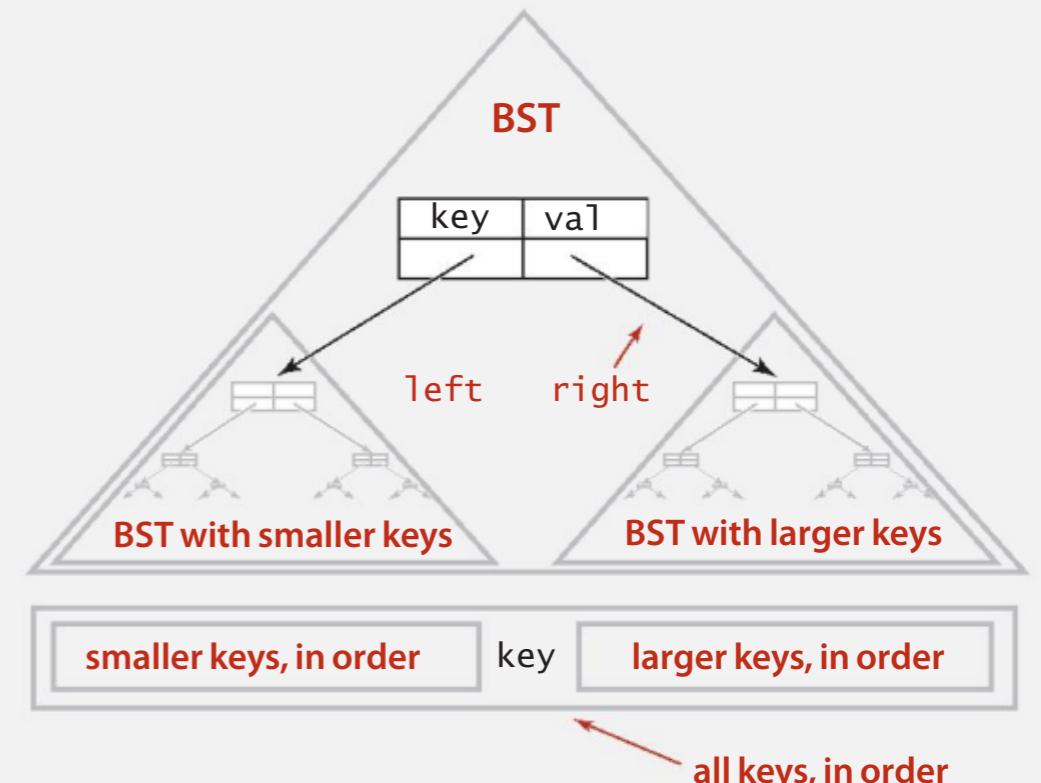
细节点..

Inorder traversal → 中序遍历.

- 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);  
}
```

仔细理解每行代码

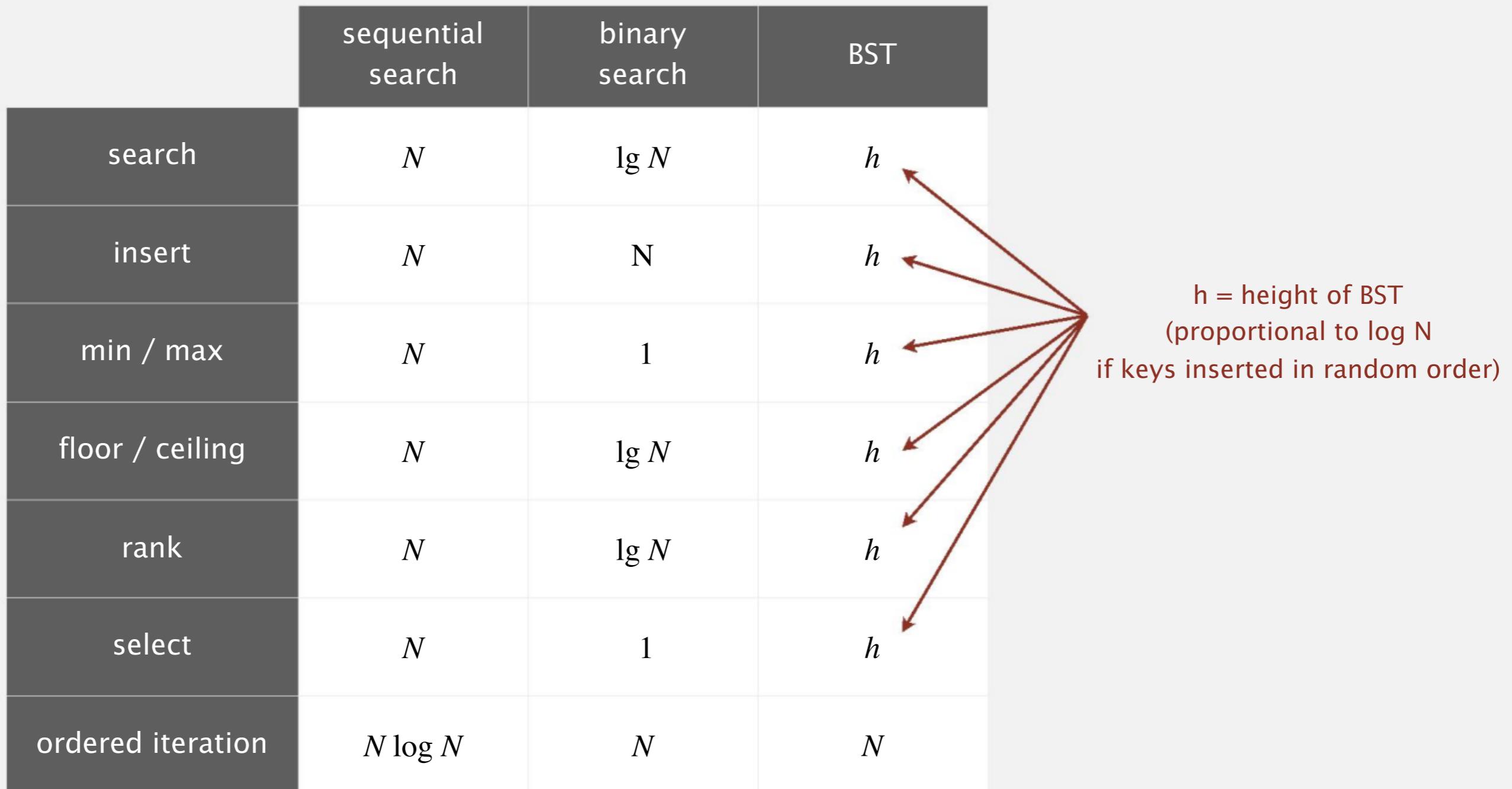


Property. Inorder traversal of a BST yields keys in ascending order.

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	N	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)



order of growth of running time of ordered symbol table operations

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

3.2 BINARY SEARCH TREES

- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

ST implementations: summary

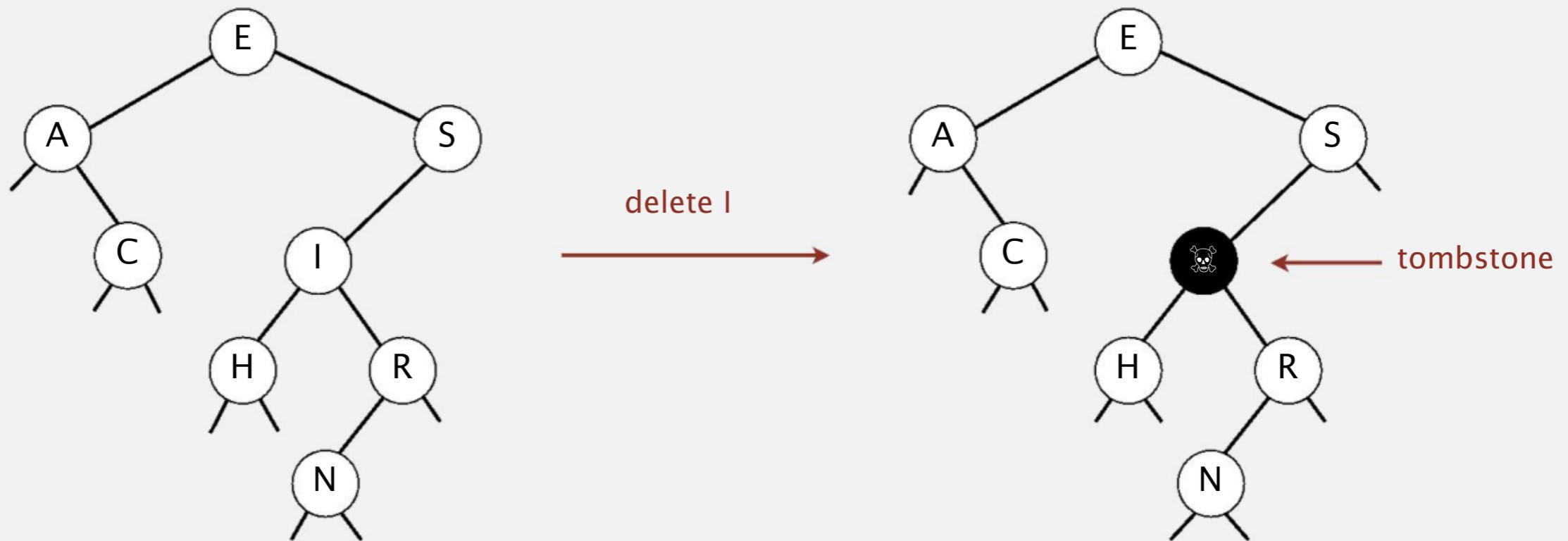
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$???	✓	<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 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.

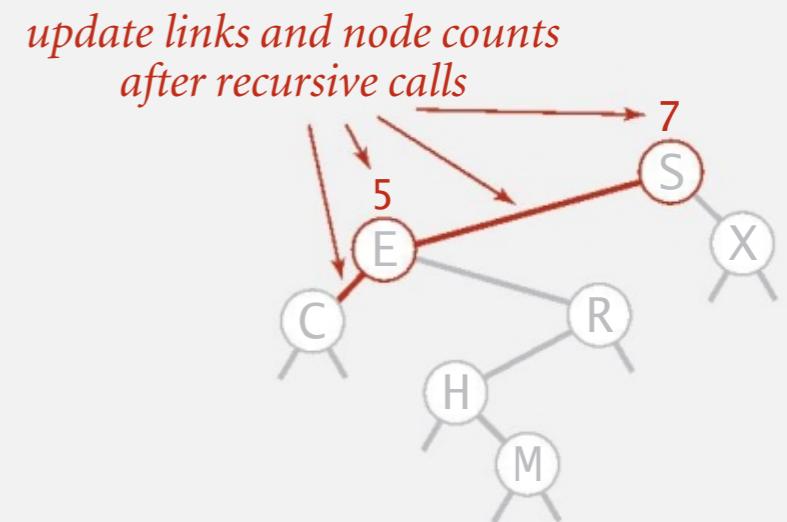
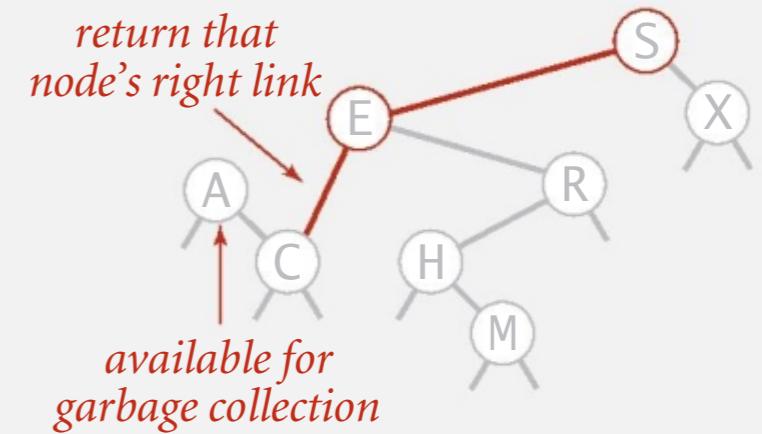
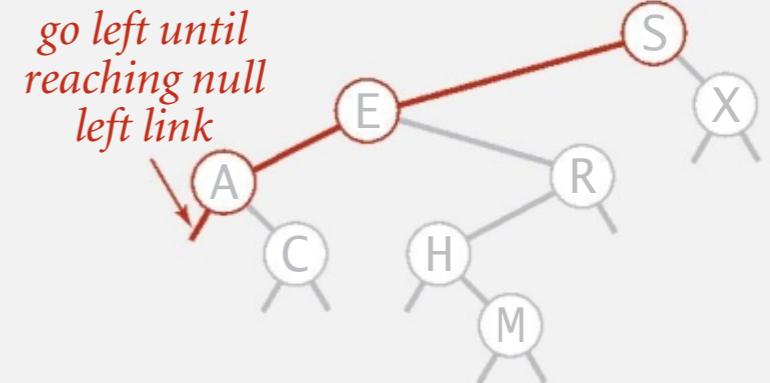
Unsatisfactory solution. Tombstone (memory) overload.

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.

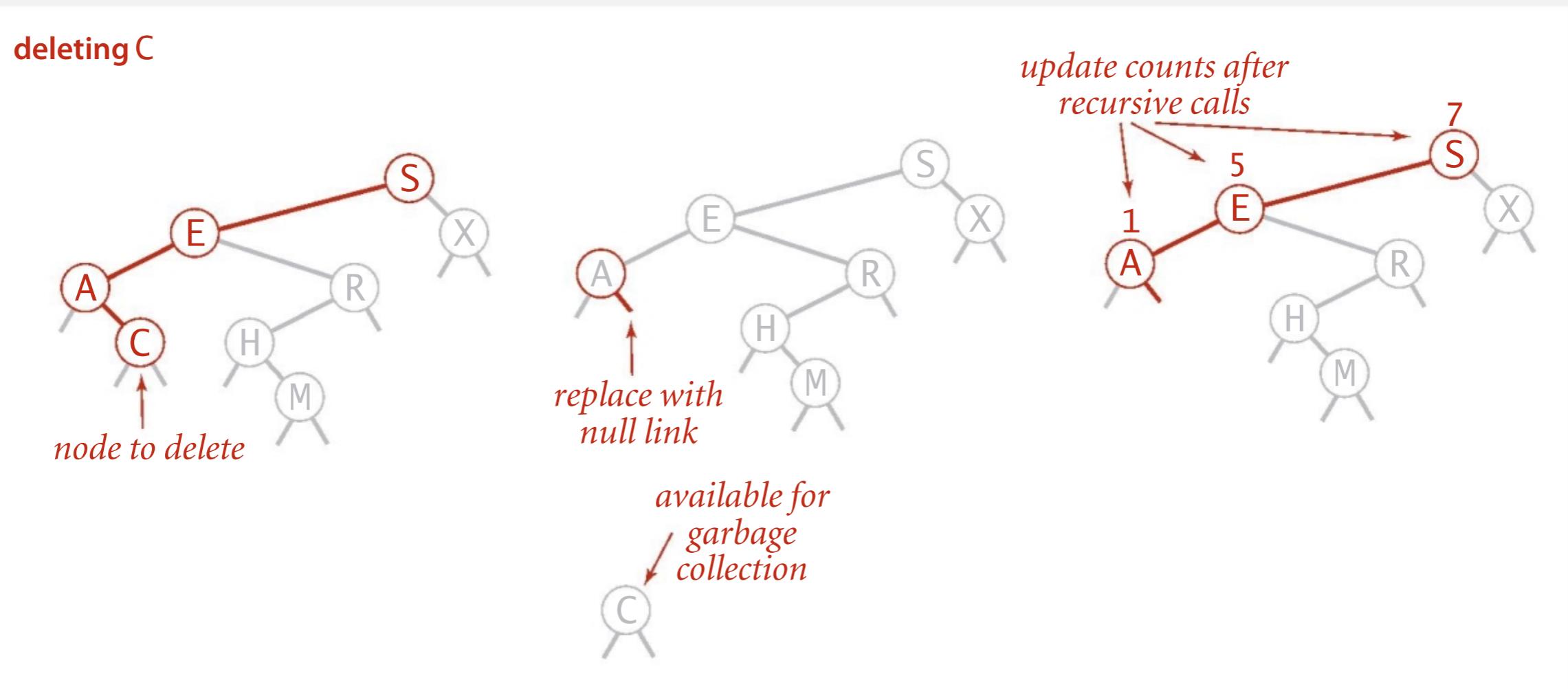
```
public void deleteMin()  
{  root = deleteMin(root);  }  
  
private Node deleteMin (Node x) {  
  
    if (x.left == null) return x.right;  
    x.left = deleteMin(x.left);  
    x.count = 1 + size(x.left) + size(x.right);  
    return x;  
}
```



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.

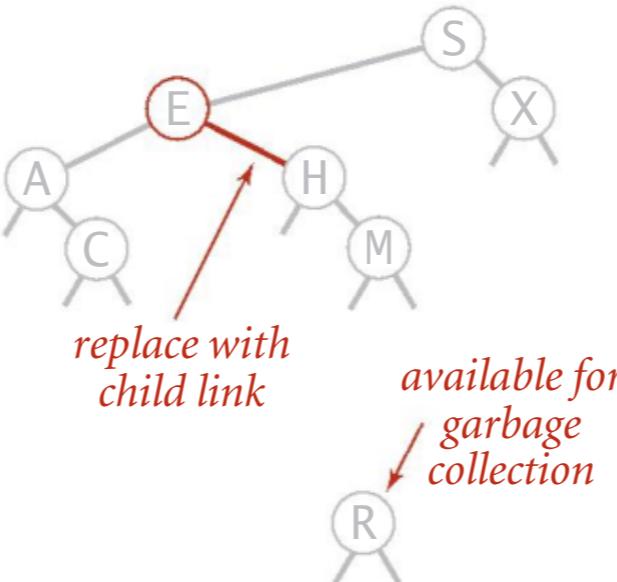
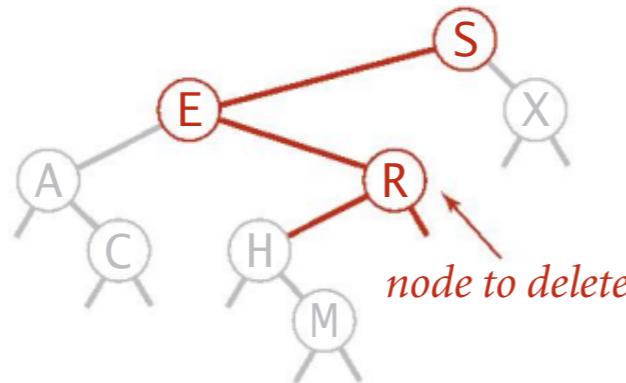


Hibbard deletion

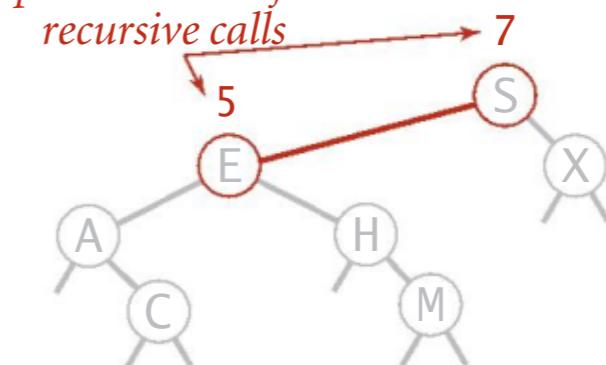
To delete a node with key k : search for node t containing key k .

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

deleting R



update counts after recursive calls

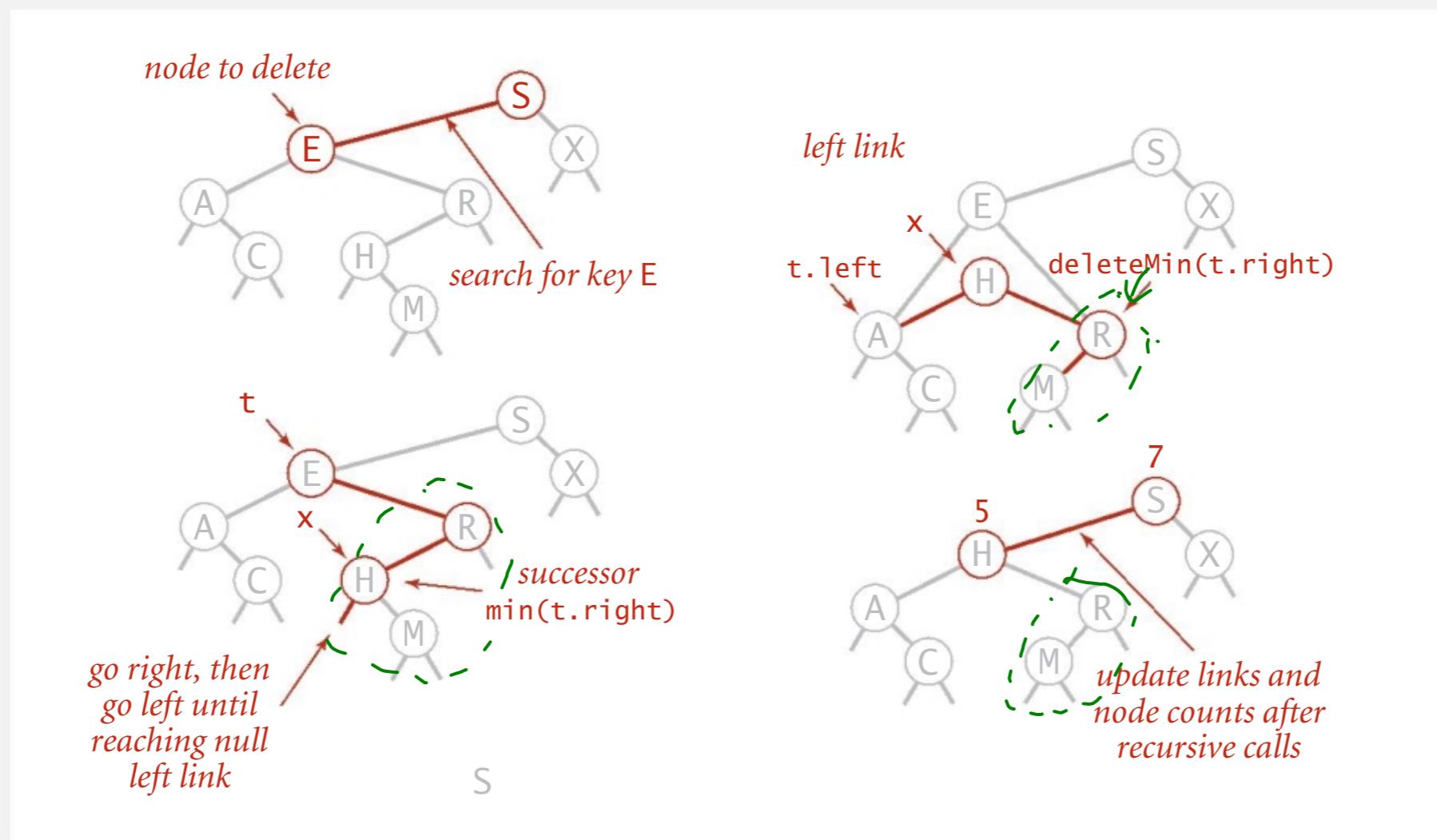


Hibbard deletion

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

细节理解

```
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

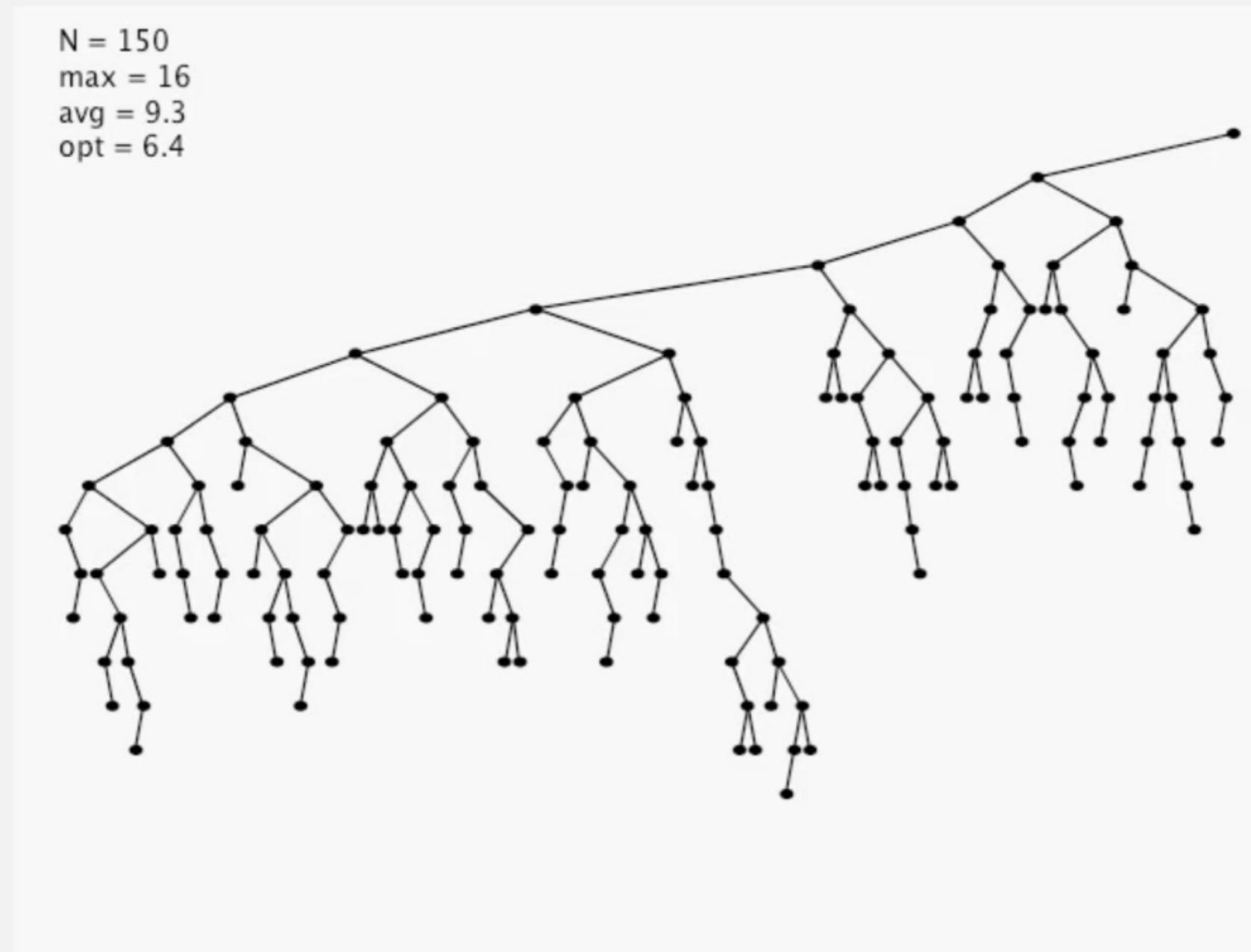
no left child

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?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>

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

Next lecture. **Guarantee** logarithmic performance for all operations.

Summary

- Symbol Tables (3.1 of Text A)
- Binary Search Trees (3.2 of Text A)

To be discussed in Lecture 11:

- Balanced Search Trees (3.3 of Text A)
- Hash Tables (3.4 of Text A)