



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

## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*



## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*

# Symbol tables

---

## Key-value pair abstraction.

- **Insert** a value with specified key.
- Given a key, **search** for the corresponding value.

### Ex. DNS lookup.

- Insert domain name with specified IP address.
- Given domain name, find corresponding IP address.

domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

↑  
key

↑  
value

# Symbol table applications

---

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address	domain name	IP address
reverse DNS	find domain name	IP address	domain name
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

# Basic symbol table API

---

**Associative array abstraction.** Associate one value with each key.

```
public class ST<Key, Value>
```

---

```
    ST()
```

*create a symbol table*

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

*put key-value pair into the table  
(remove key from table if value is null)*

← a[key] = val;

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

*value paired with key  
(null if key is absent)*

← a[key]

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

*remove key (and its value) from table*

```
    boolean contains(Key key)
```

*is there a value paired with key?*

```
    boolean isEmpty()
```

*is the table empty?*

```
    int size()
```

*number of key-value pairs in the table*

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

*all the keys in the table*

## Conventions

---

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

### Intended consequences.

- Easy to implement `contains()`.

```
public boolean contains(Key key)
{ return get(key) != null; }
```

- Can implement lazy version of `delete()`.

```
public void delete(Key key)
{ put(key, null); }
```

# Keys and values


---

**Value type.** Any generic type.

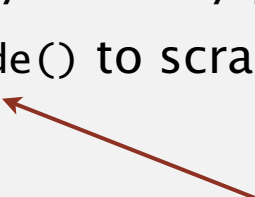
**Key type:** several natural assumptions.

- Assume keys are Comparable, use `compareTo()`.
- Assume keys are any generic type, use `equals()` to test equality.
- Assume keys are any generic type, use `equals()` to test equality; use `hashCode()` to scramble key.

specify Comparable in API.



built-in to Java  
(stay tuned)



**Best practices.** Use immutable types for symbol table keys.

- Immutable in Java: Integer, Double, String, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

# Equality test

---

All Java classes inherit a method `equals()`.

**Java requirements.** For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

} equivalence  
relation

do `x` and `y` refer to  
the same object?

**Default implementation.** `(x == y)`

**Customized implementations.** `Integer`, `Double`, `String`, `java.io.File`, ...

**User-defined implementations.** Some care needed.



# Implementing equals for user-defined types


---

Seems easy.

```
public class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that)
    {
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

check that all significant  
fields are the same



# Implementing equals for user-defined types

Seems easy, but requires some care.

typically unsafe to use equals() with inheritance  
(would violate symmetry)

```
public final class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Object y)
    {
        if (y == this) return true;

        if (y == null) return false;

        if (y.getClass() != this.getClass())
            return false;

        Date that = (Date) y;
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

must be Object.  
Why? Experts still debate.

optimize for true object equality

check for null

objects must be in the same class  
(religion: getClass() vs. instanceof)



cast is guaranteed to succeed

check that all significant  
fields are the same

# Equals design

---

## "Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against `null`.
- Check that two objects are of the same type and cast.
- Compare each significant field:
  - if field is a primitive type, use `==`
  - if field is an object, use `equals()`  apply rule recursively
  - if field is an array, apply to each entry  alternatively, use `Arrays.equals(a, b)` or `Arrays.deepEquals(a, b)`, but not `a.equals(b)`

## Best practices.

- No need to use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make `compareTo()` consistent with `equals()`.

 `x.equals(y)` if and only if `(x.compareTo(y) == 0)`

## ST test client for traces

---

Build ST by associating value  $i$  with  $i^{th}$  string from standard input.

```
public static void main(String[] args)
{
    ST<String, Integer> st = new ST<String, Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}
```

<b>keys</b>	S	E	A	R	C	H	E	X	A	M	P	L	E
<b>values</b>	0	1	2	3	4	5	6	7	8	9	10	11	12

**output**

A	8
C	4
E	12
H	5
L	11
M	9
P	10
R	3
S	0
X	7

# ST test client for analysis

---

**Frequency counter.** Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
```

```
% java FrequencyCounter 1 < tinyTale.txt
it 10
```

← tiny example  
(60 words, 20 distinct)

```
% java FrequencyCounter 8 < tale.txt
business 122
```

← real example  
(135,635 words, 10,769 distinct)

```
% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

← real example  
(21,191,455 words, 534,580 distinct)

# Frequency counter implementation

---

```
public class FrequencyCounter
{
    public static void main(String[] args)
    {
        int minlen = Integer.parseInt(args[0]);
        ST<String, Integer> st = new ST<String, Integer>();
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (word.length() < minlen) continue;
            if (!st.contains(word)) st.put(word, 1);
            else
                st.put(word, st.get(word) + 1);
        }
        String max = "";
        st.put(max, 0);
        for (String word : st.keys())
            if (st.get(word) > st.get(max))
                max = word;
        StdOut.println(max + " " + st.get(max));
    }
}
```

create ST

ignore short strings

read string and update frequency

print a string with max freq



## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*



## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*

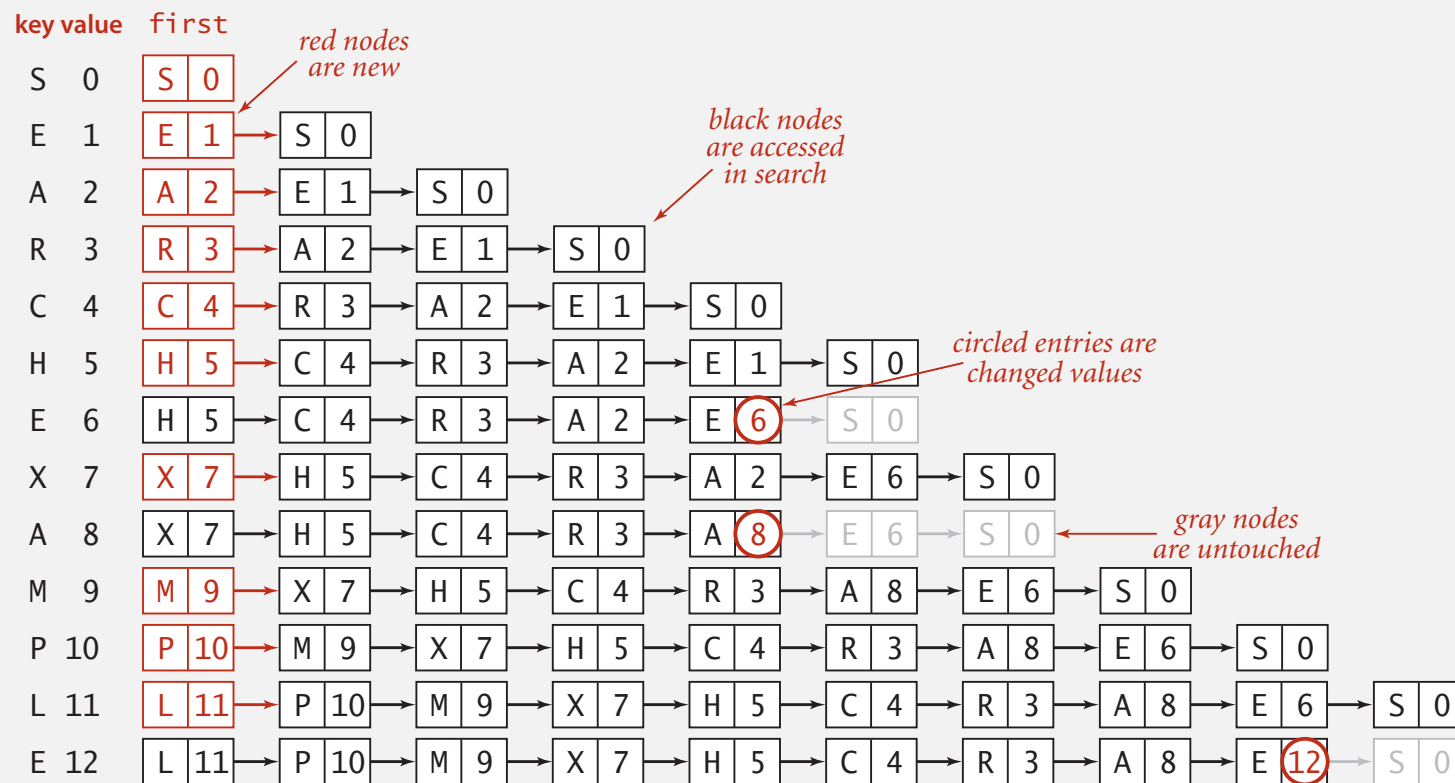


# Sequential search in a linked list

**Data structure.** Maintain an (unordered) linked list of key-value pairs.

**Search.** Scan through all keys until find a match.

**Insert.** Scan through all keys until find a match; if no match add to front.



Trace of linked-list ST implementation for standard indexing client

## Elementary ST implementations: summary

---

ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	$N / 2$	N	no	<code>equals()</code>

**Challenge.** Efficient implementations of both search and insert.

# Binary search in an ordered array

**Data structure.** Maintain an ordered array of key-value pairs.

**Rank helper function.** How many keys  $< k$ ?

			keys[]									
			0	1	2	3	4	5	6	7	8	9
successful search for P												
lo	hi	m										
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
6	6	6	A	C	E	H	L	M	P	R	S	X
unsuccessful search for Q												
lo	hi	m										
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
7	6	6	A	C	E	H	L	M	P	R	S	X

*entries in black are a[lo..hi]*

*entry in red is a[m]*

*loop exits with keys[m] = P: return 6*

*loop exits with lo > hi: return 7*

Trace of binary search for rank in an ordered array

## Binary search: Java implementation

---

```
public Value get(Key key)
{
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
}
```

```
private int rank(Key key)
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) hi = mid - 1;
        else if (cmp > 0) lo = mid + 1;
        else if (cmp == 0) return mid;
    }
    return lo;
}
```

number of keys < key

# Binary search: trace of standard indexing client

**Problem.** To insert, need to shift all greater keys over.

		keys[]											vals[]									
key	value	0	1	2	3	4	5	6	7	8	9	N	0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

entries in red  
were inserted

entries in gray  
did not move

entries in black  
moved to the right

circled entries are  
changed values

## Elementary ST implementations: summary

---

ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	$N / 2$	N	no	<code>equals()</code>
binary search (ordered array)	$\log N$	N	$\log N$	$N / 2$	yes	<code>compareTo()</code>

**Challenge.** Efficient implementations of both search and insert.



## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*



## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*



# Examples of ordered symbol table API

---

	<i>keys</i>	<i>values</i>
<code>min()</code> →	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
<code>get(09:00:13)</code> →	09:00:59	Chicago
	09:01:10	Houston
<code>floor(09:05:00)</code> →	09:03:13	Chicago
	09:10:11	Seattle
<code>select(7)</code> →	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
<code>keys(09:15:00, 09:25:00)</code> →	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
<code>ceiling(09:30:00)</code> →	09:35:21	Chicago
	09:36:14	Seattle
<code>max()</code> →	09:37:44	Phoenix

`size(09:15:00, 09:25:00)` is 5  
`rank(09:10:25)` is 7

# Ordered symbol table API

---

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

---

ST()	<i>create an ordered symbol table</i>
void put(Key key, Value val)	<i>put key-value pair into the table (remove key from table if value is null)</i>
Value get(Key key)	<i>value paired with key (null if key is absent)</i>
void delete(Key key)	<i>remove key (and its value) from table</i>
boolean contains(Key key)	<i>is there a value paired with key?</i>
boolean isEmpty()	<i>is the table empty?</i>
int size()	<i>number of key-value pairs</i>
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>
void deleteMin()	<i>delete smallest key</i>
void deleteMax()	<i>delete largest key</i>
int size(Key lo, Key hi)	<i>number of keys in [lo..hi]</i>
Iterable<Key> keys(Key lo, Key hi)	<i>keys in [lo..hi], in sorted order</i>
Iterable<Key> keys()	<i>all keys in the table, in sorted order</i>

## Binary search: ordered symbol table operations summary

---

	sequential search	binary search
search	$N$	$\lg N$
insert / delete	$N$	$N$
min / max	$N$	$1$
floor / ceiling	$N$	$\lg N$
rank	$N$	$\lg N$
select	$N$	$1$
ordered iteration	$N \lg N$	$N$

order of growth of the running time for ordered symbol table operations



## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*



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

## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*



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

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*



## 3.2 BINARY SEARCH TREES

---

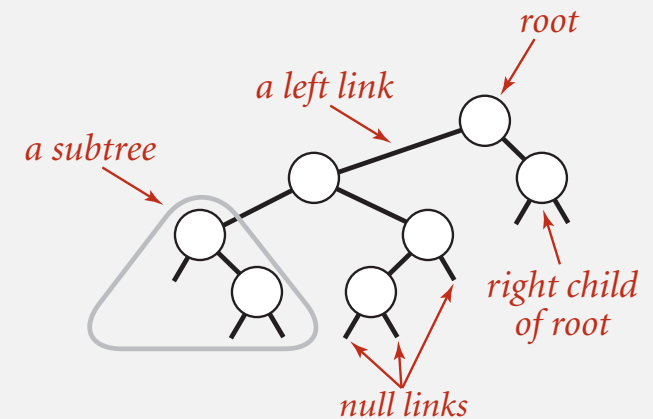
- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

# Binary search trees

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

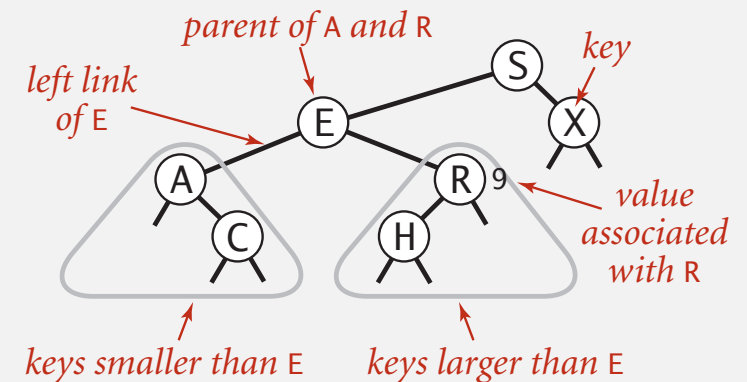
A binary tree is either:

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



**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.





# BST representation in Java

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

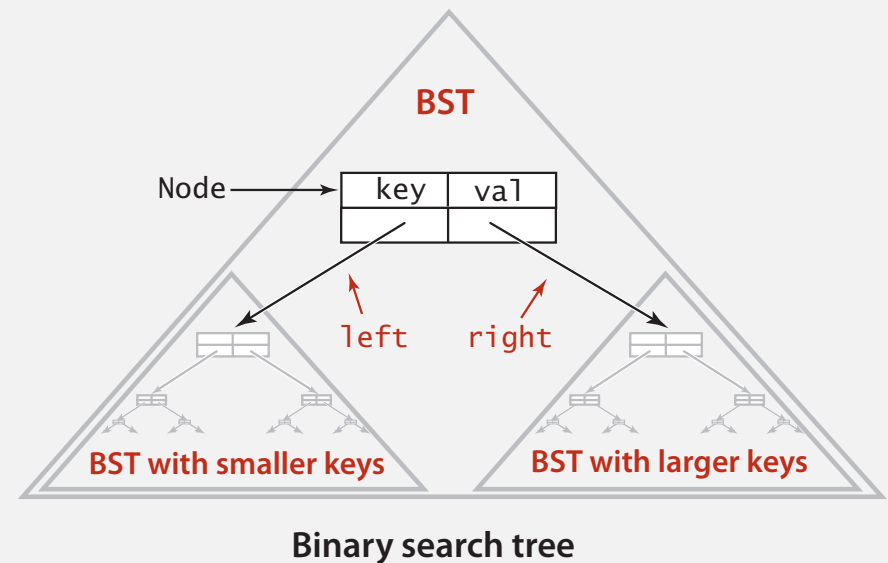
A Node is comprised of four fields:

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

↑ 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 */ }
```

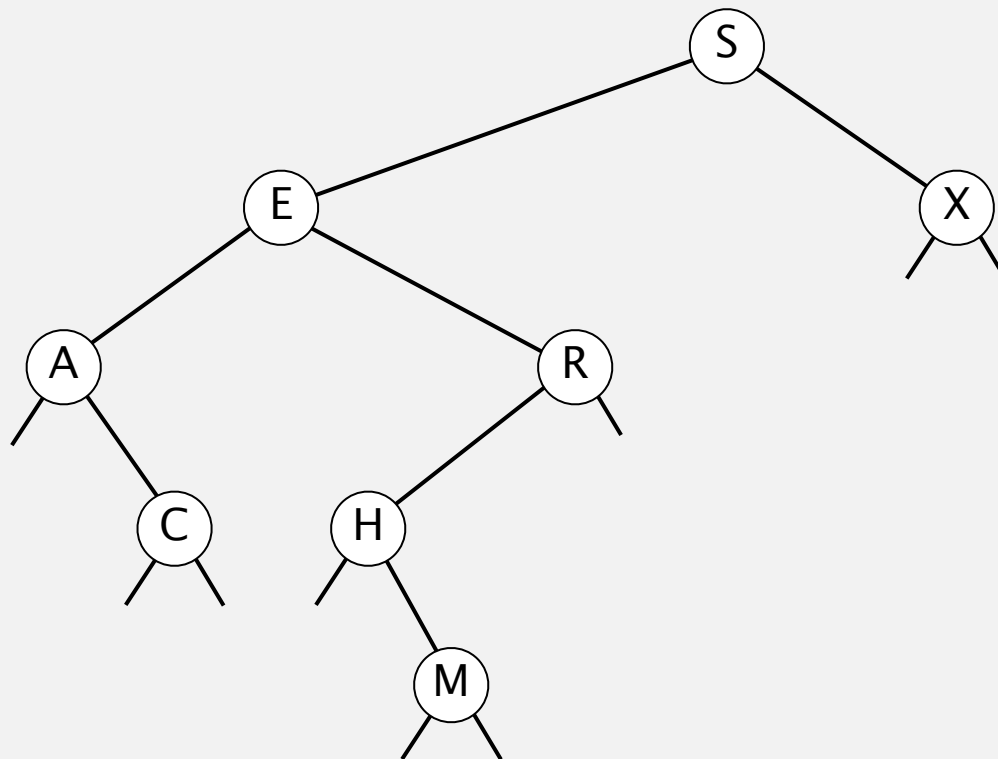
```
}
```

# Binary search tree demo

---

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

**successful search for H**

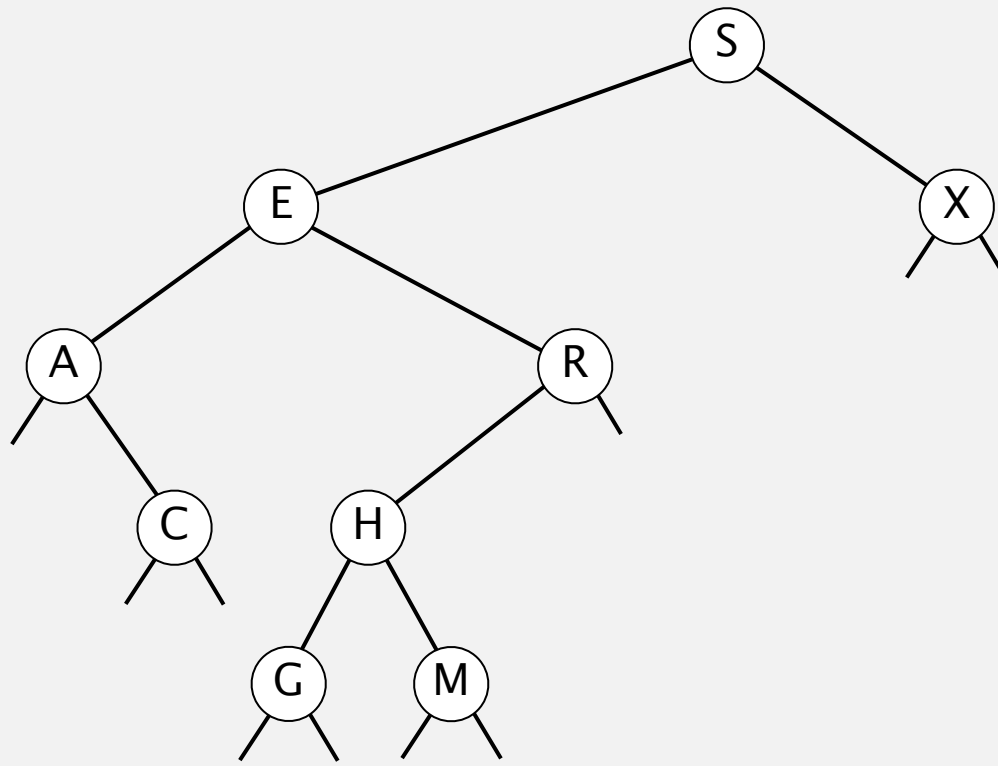


# Binary search tree demo

---

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

**insert G**



## 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 + depth of node.

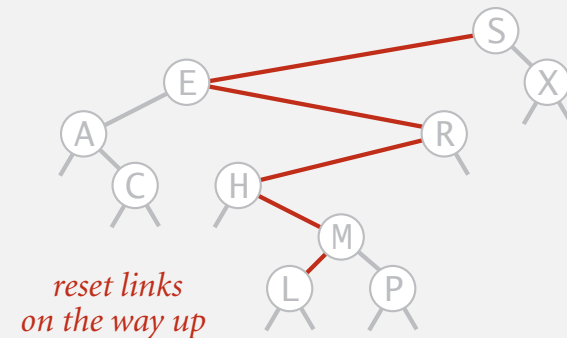
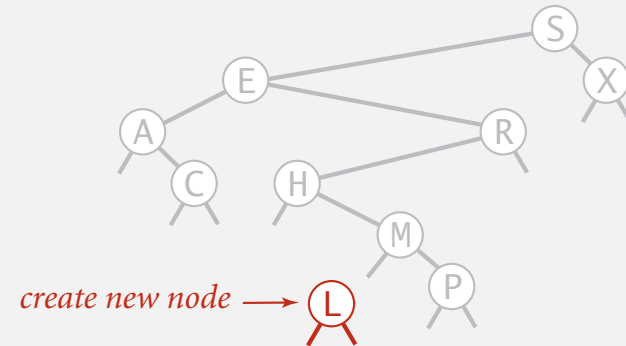
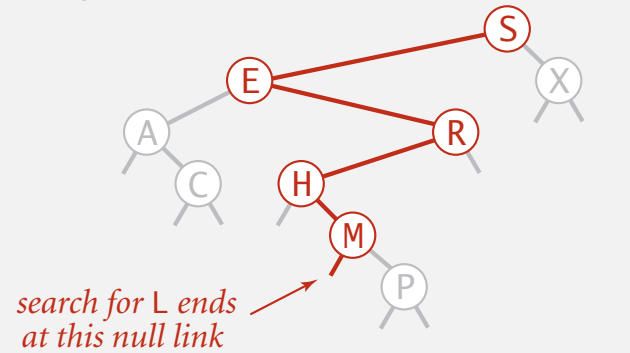
# BST insert

**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



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!

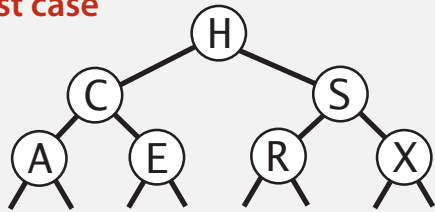
**Cost.** Number of compares is equal to 1 + depth of node.

# Tree shape

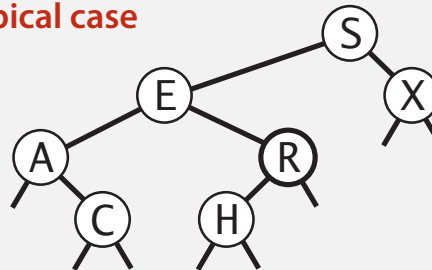
---

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

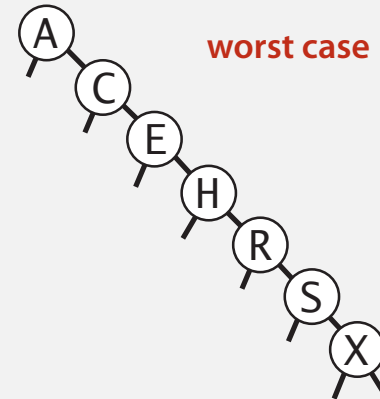
best case



typical case



worst case



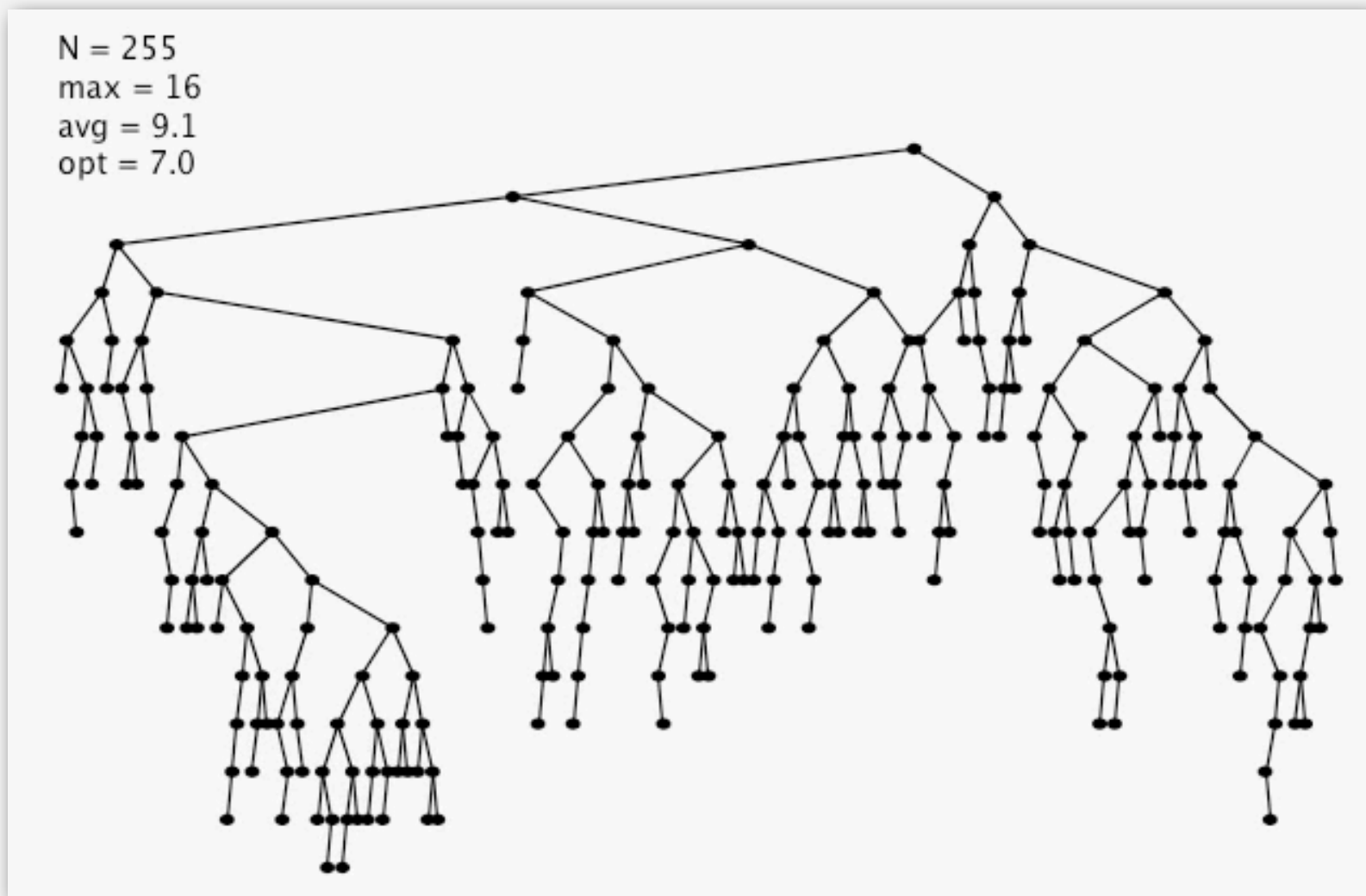
**Remark.** Tree shape depends on order of insertion.



## BST insertion: random order visualization

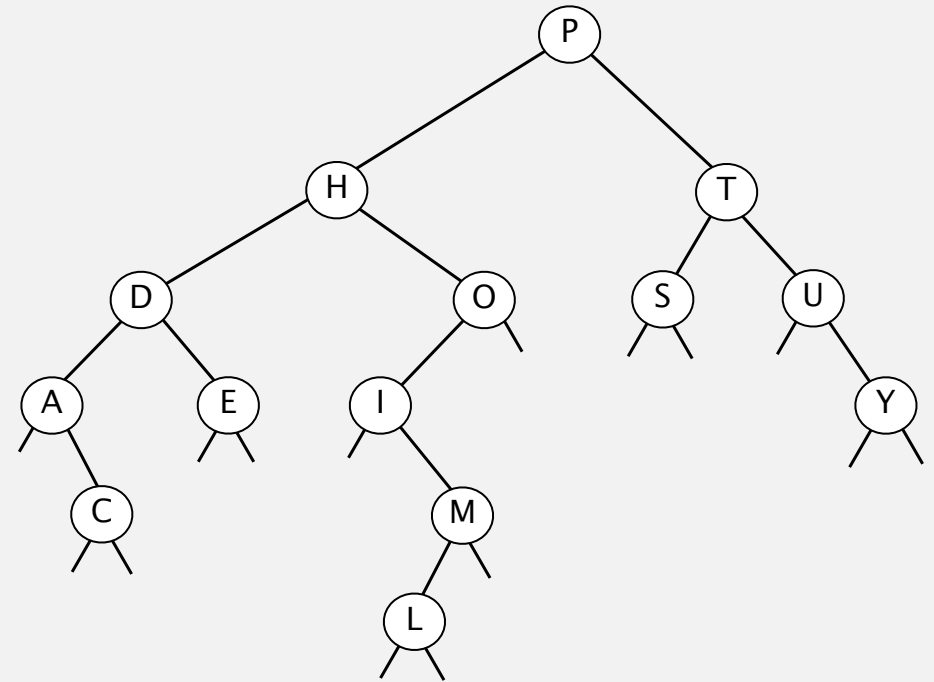
---

Ex. Insert keys in random order.



# 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



**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.

**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  $\mathbf{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$ . 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		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	equals()
binary search (ordered array)	lg N	N	lg N	N/2	yes	compareTo()
BST	N	N	1.39 lg N	1.39 lg N	next	compareTo()



## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*



## 3.2 BINARY SEARCH TREES

---

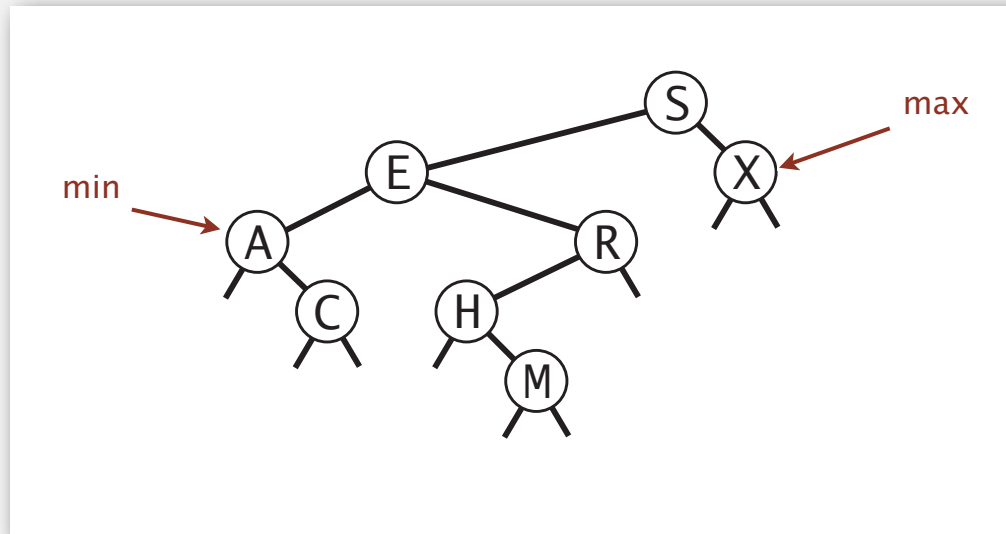
- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

# Minimum and maximum

---

**Minimum.** Smallest key in table.

**Maximum.** Largest key in table.



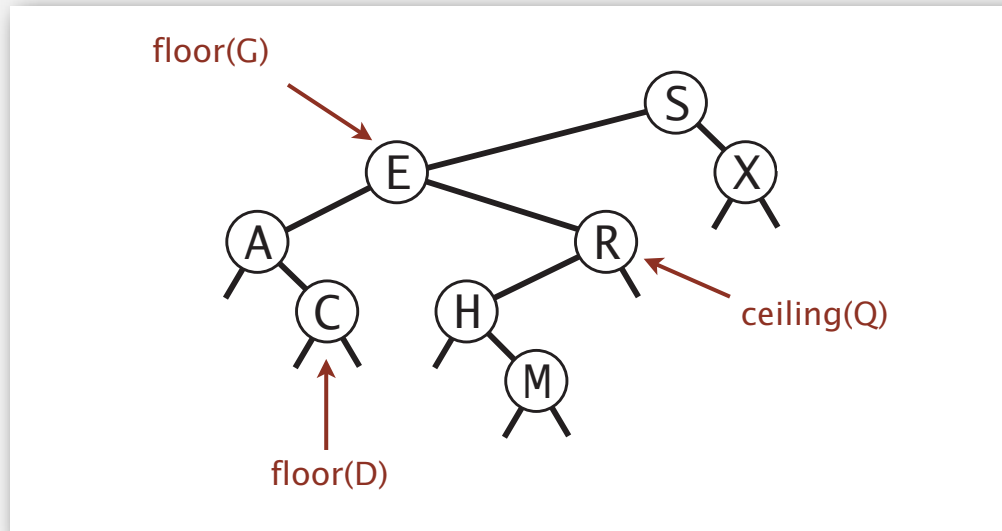
**Q.** How to find the min / max?

# 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 at root]

The floor of  $k$  is  $k$ .

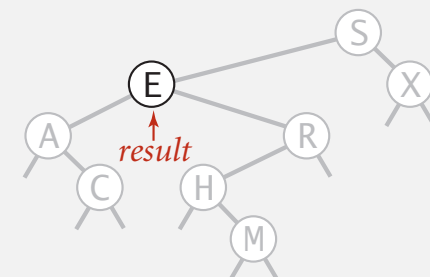
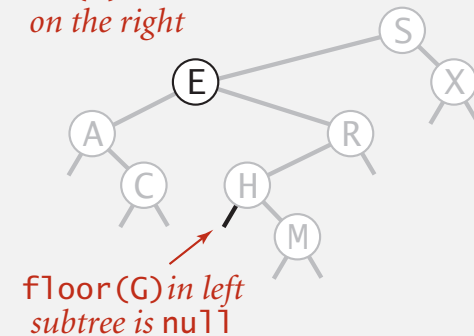
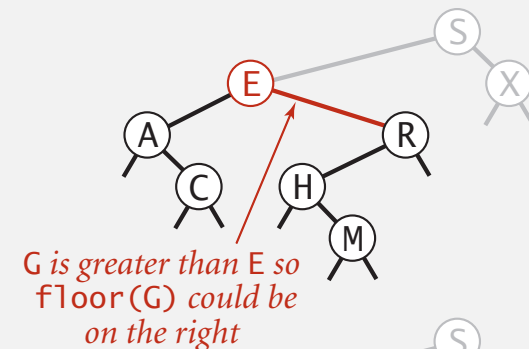
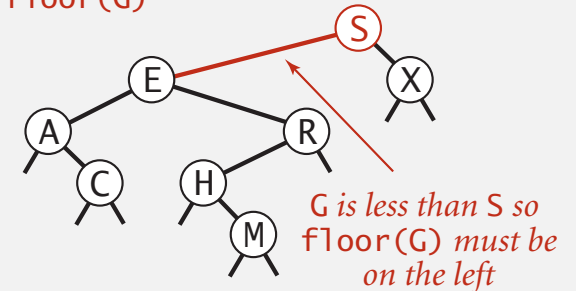
**Case 2.** [ $k$  is less than the key at root]

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

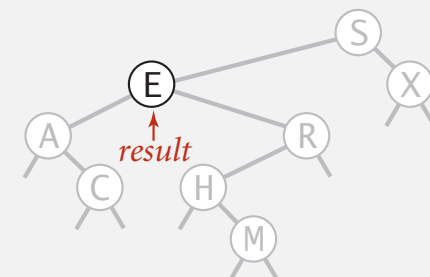
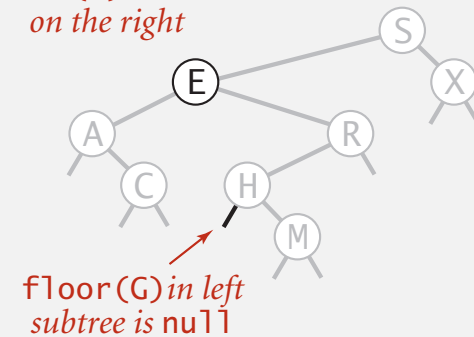
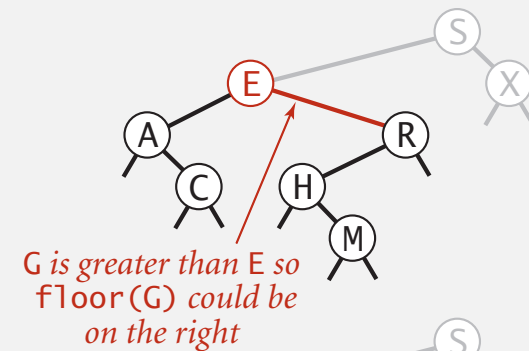
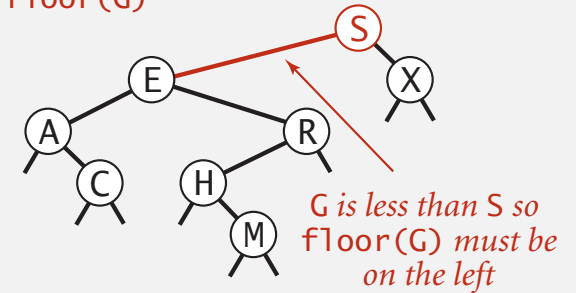
```
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    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;
}
```

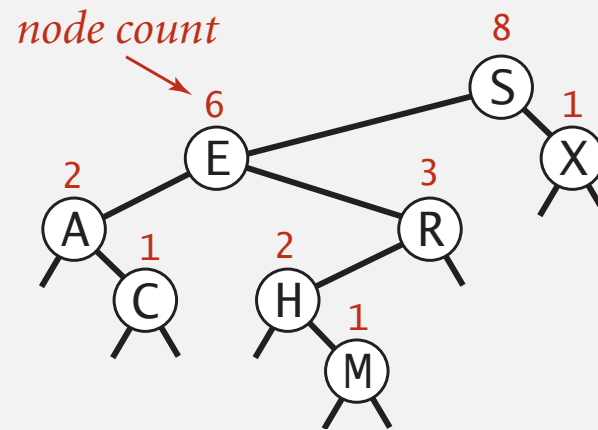
finding floor(G)



## 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 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

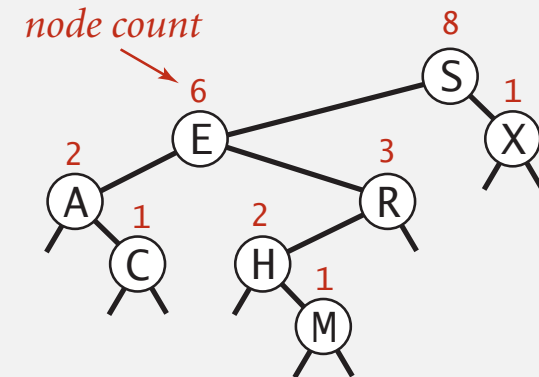


```
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else if (cmp == 0) x.val = val;
    x.count = 1 + size(x.left)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

# 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);
```

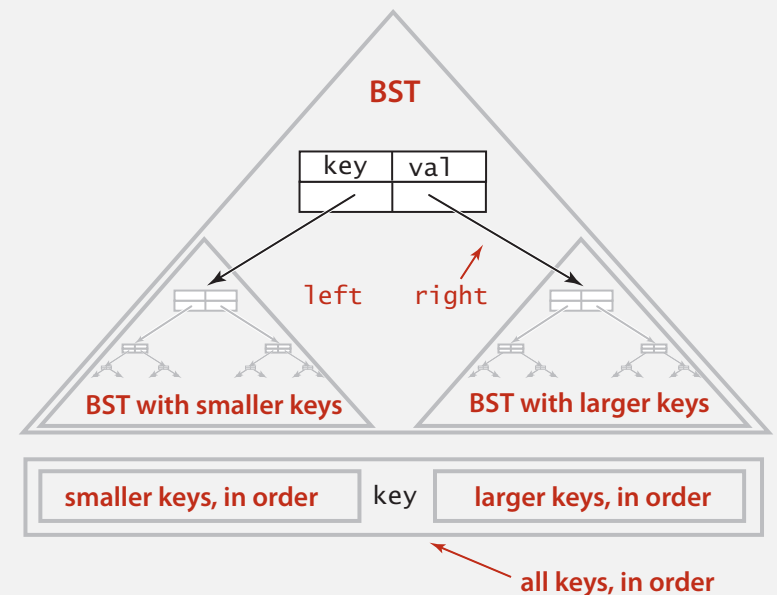
```
}
```

# 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);
}
```

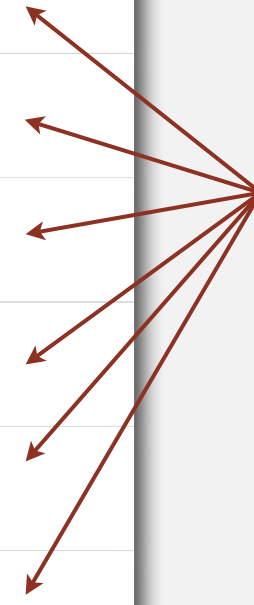


**Property.** Inorder traversal of a BST yields keys in ascending order.

# 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



## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*





## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

## ST implementations: summary

---

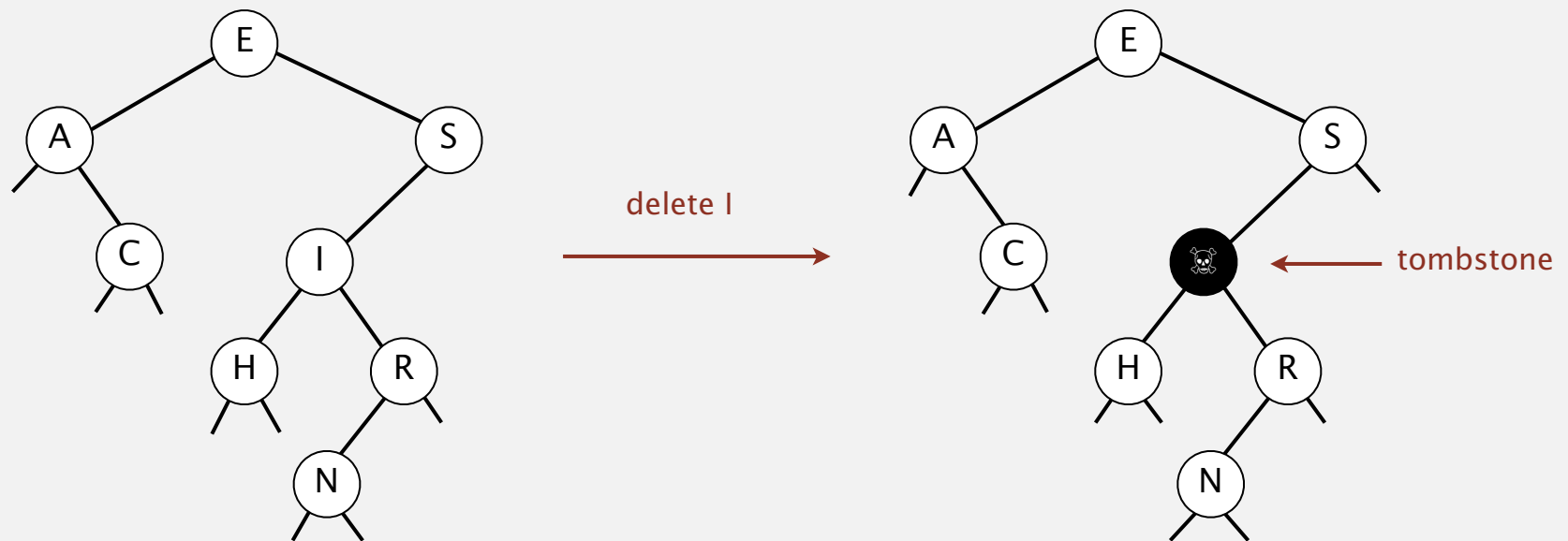
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	???	yes	compareTo()

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.

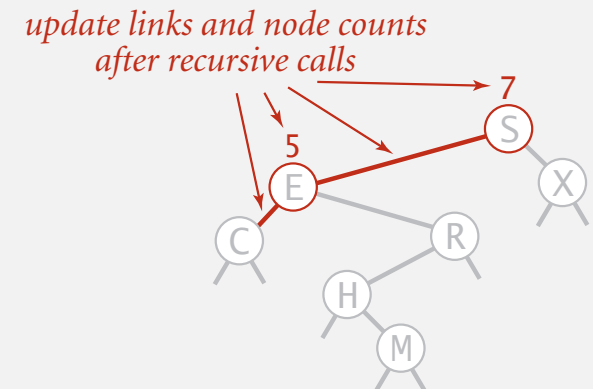
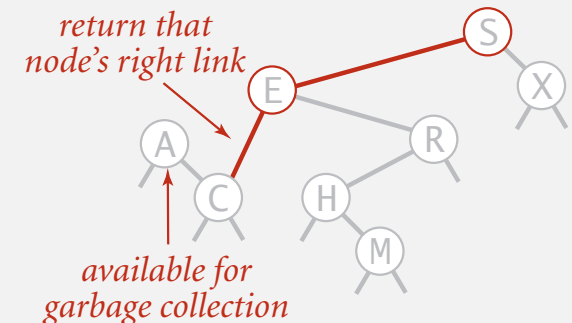
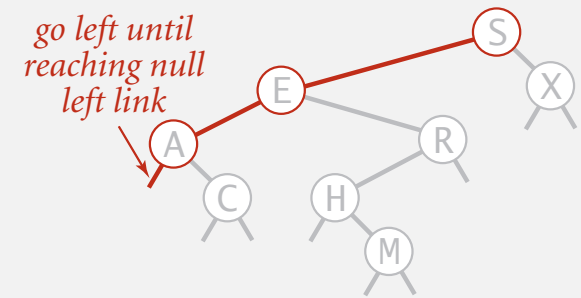
**Unsatisfactory solution.** Tombstone (memory) overload.

# 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;  
}
```

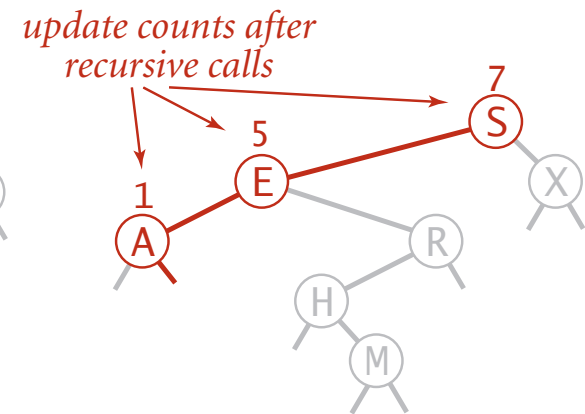
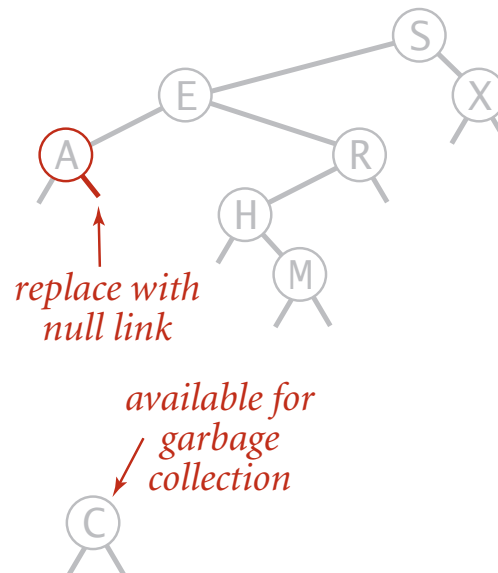
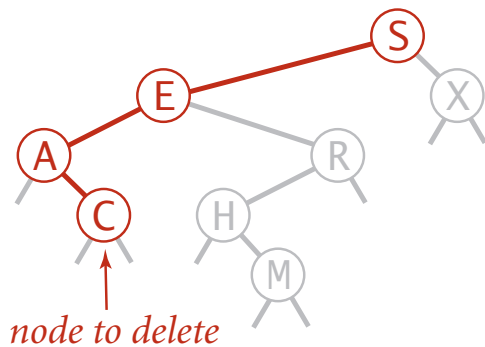


# Hibbard deletion

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.

deleting C

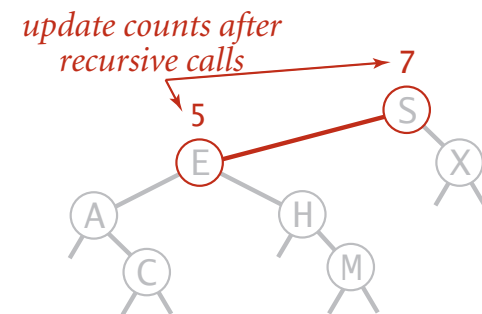
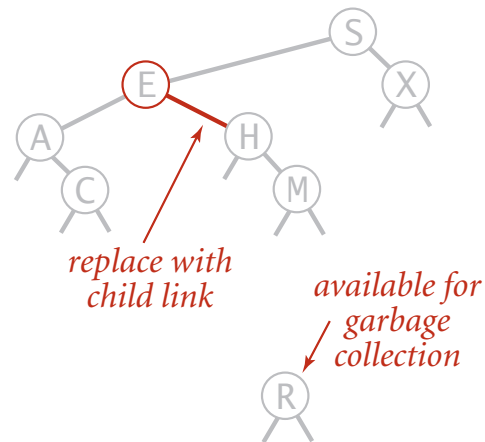
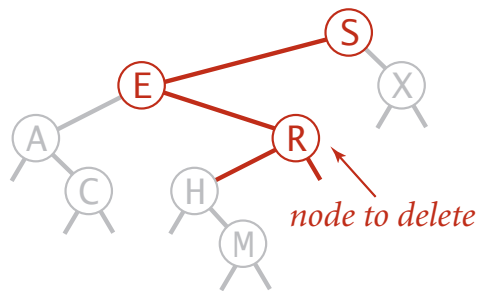


# Hibbard deletion

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

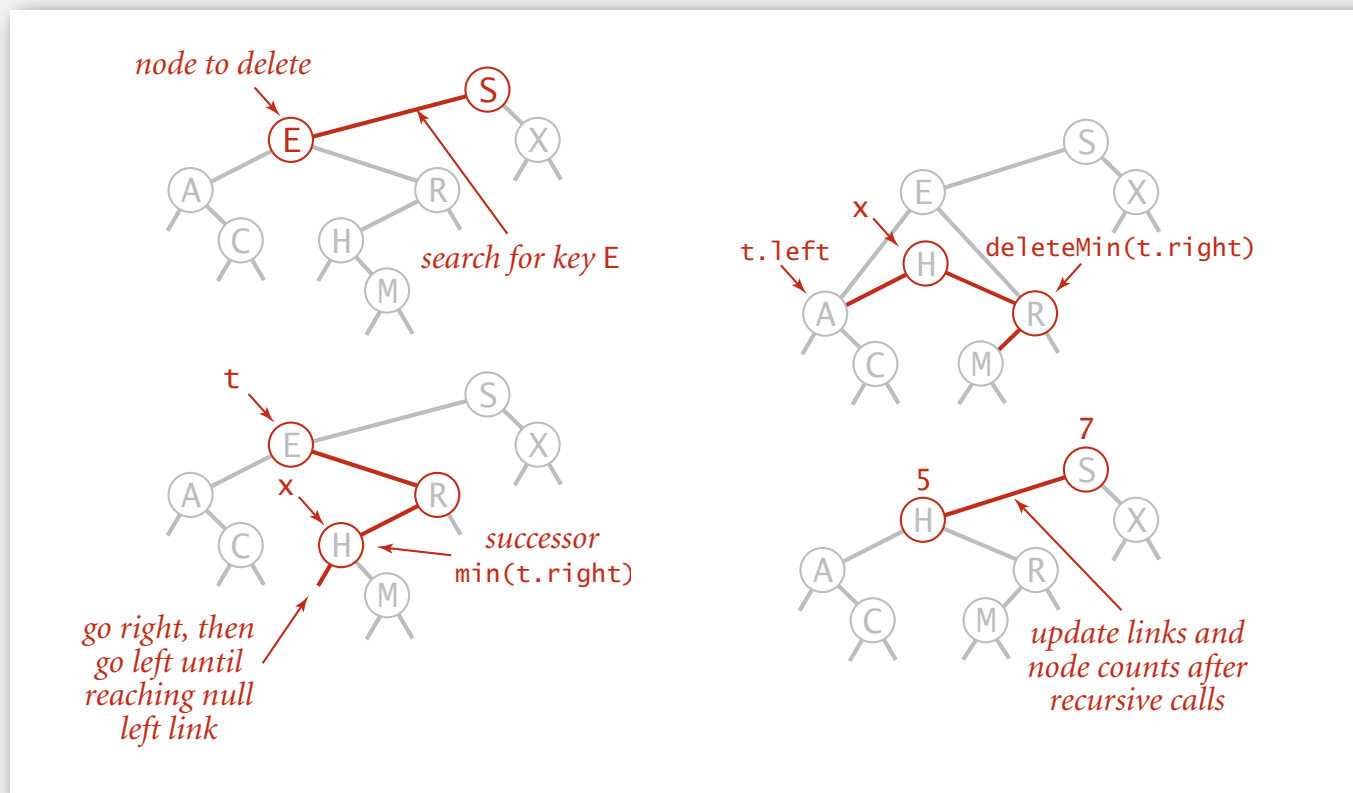


# Hibbard deletion

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;
```

```
        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

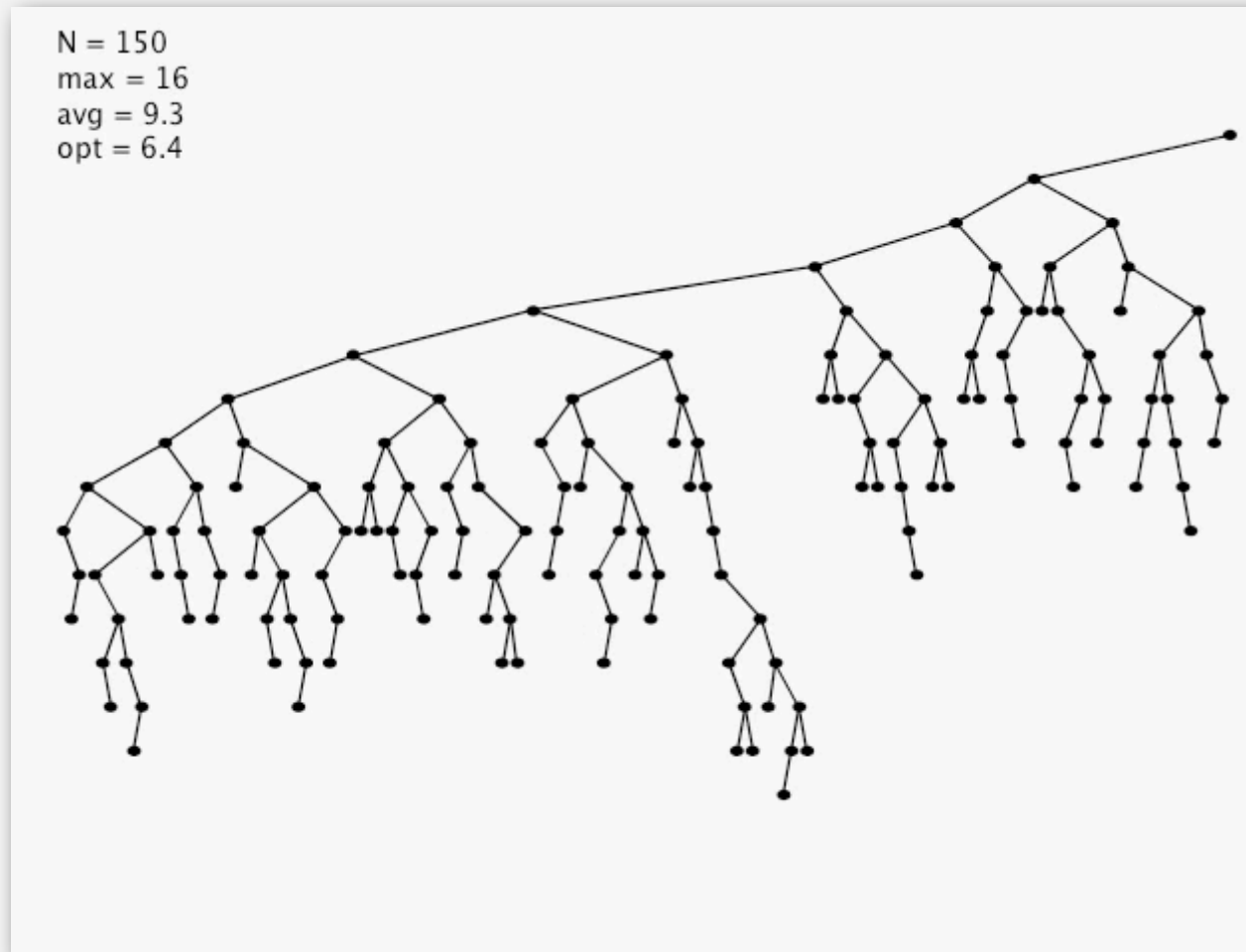
← update subtree  
counts



# Hibbard deletion: analysis

---

Unsatisfactory solution. Not symmetric.

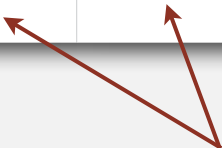


Surprising consequence. Trees not random (!)  $\Rightarrow \sqrt{N}$  per op.  
Longstanding open problem. Simple and efficient delete for BSTs.

# ST implementations: summary

---

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



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

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



## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*



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

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*