

BINARY SEARCH



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

Binary search

To search in an ordered array of data in $\log(N)$ time

Idea: Compare to the middle element.

If the key of the middle element is smaller than that we search for continue search in right-hand (high indices) sub-array else continue search in left-hand (low indices) sub-array.

When the length of the sub-array is one we either have found the key or not.

More efficient to implement iteratively...

Binary search – pseudo code

```
int binSearch(Array arr[], Key key, int low, int high)
while(high >= 1)
    int mid = (low + high)/2
    if(key == arr[mid]) return mid
    if(key < arr[mid]) high = mid-1
    else low = mid + 1
return -1 //not found
```



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

3.2 BINARY SEARCH TREES

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



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

3.2 BINARY SEARCH TREES

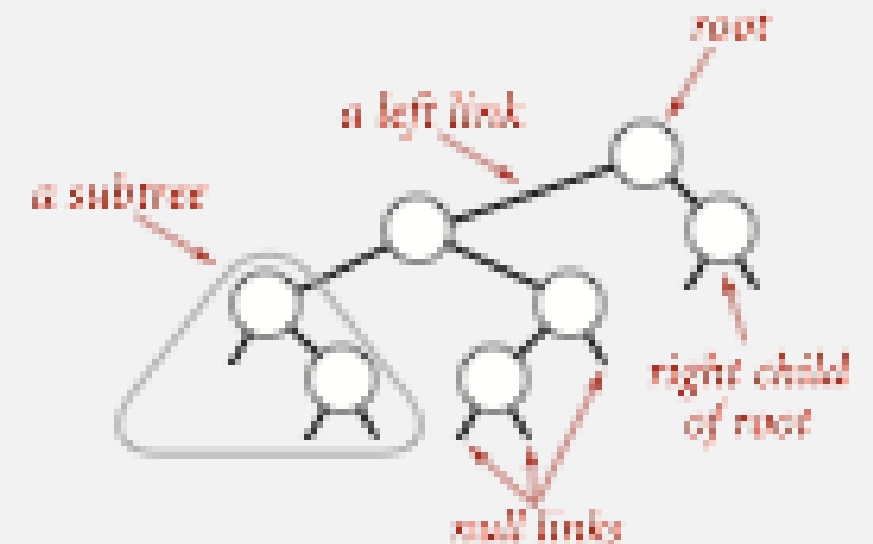
- *BSTs*
- *ordered operations*
- *deletion*

Binary search trees

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

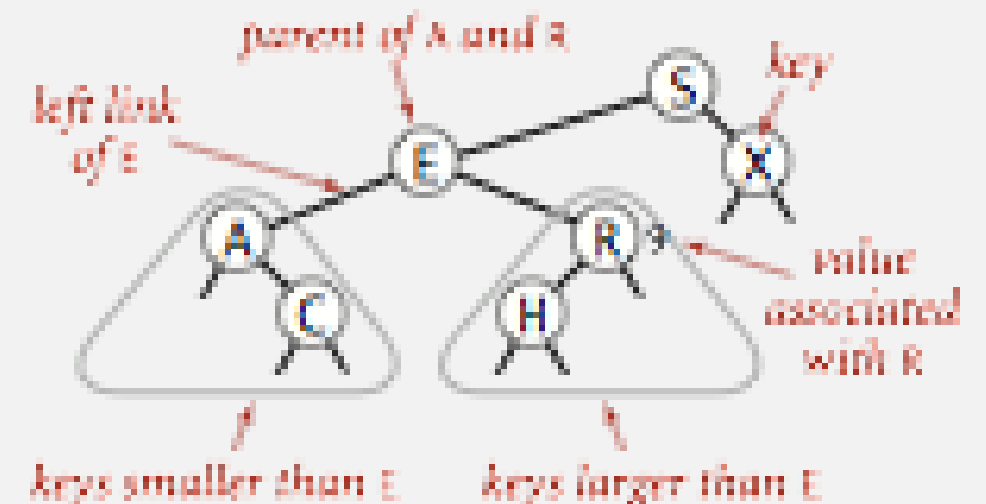
A binary tree is either:

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



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

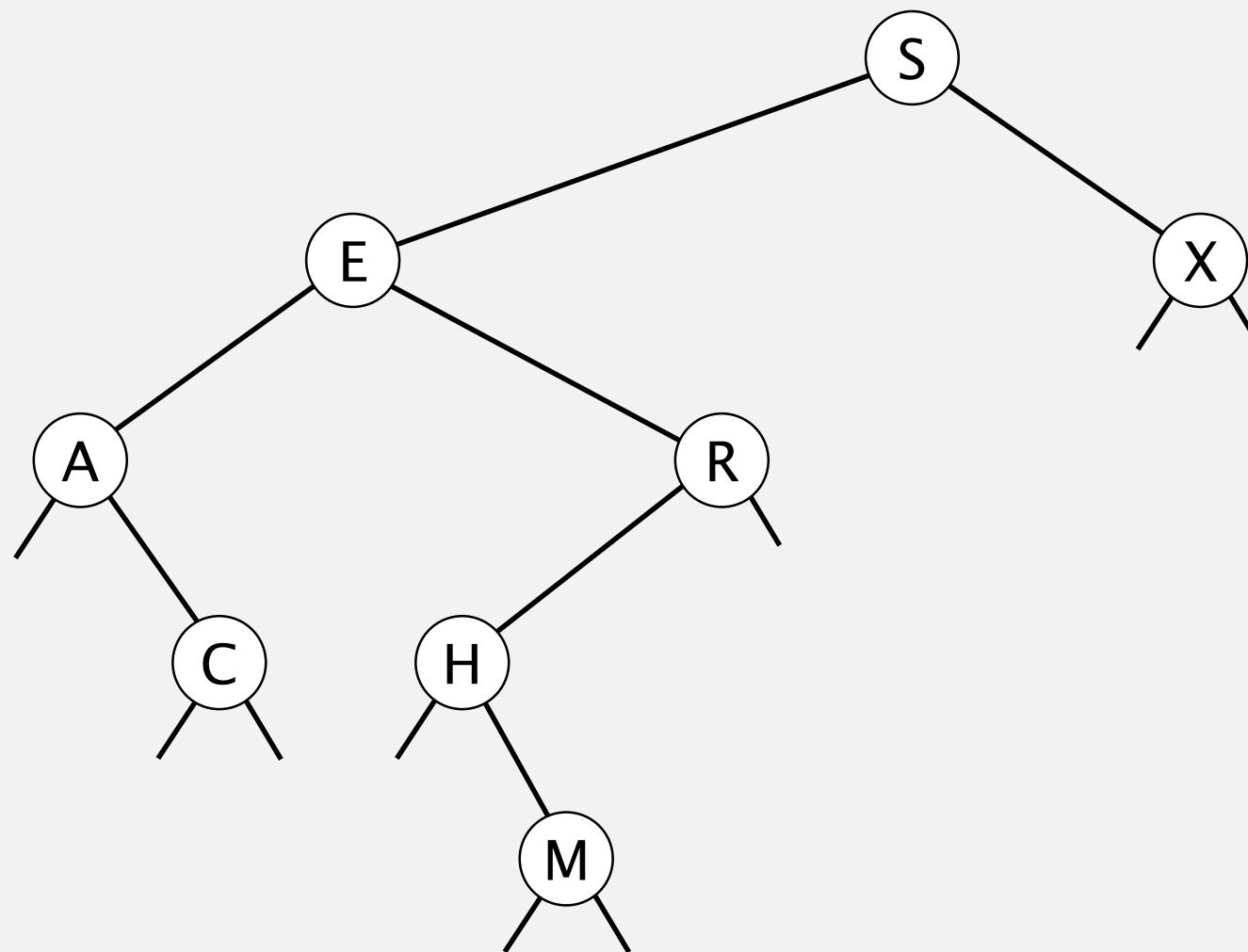
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Binary search tree demo

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

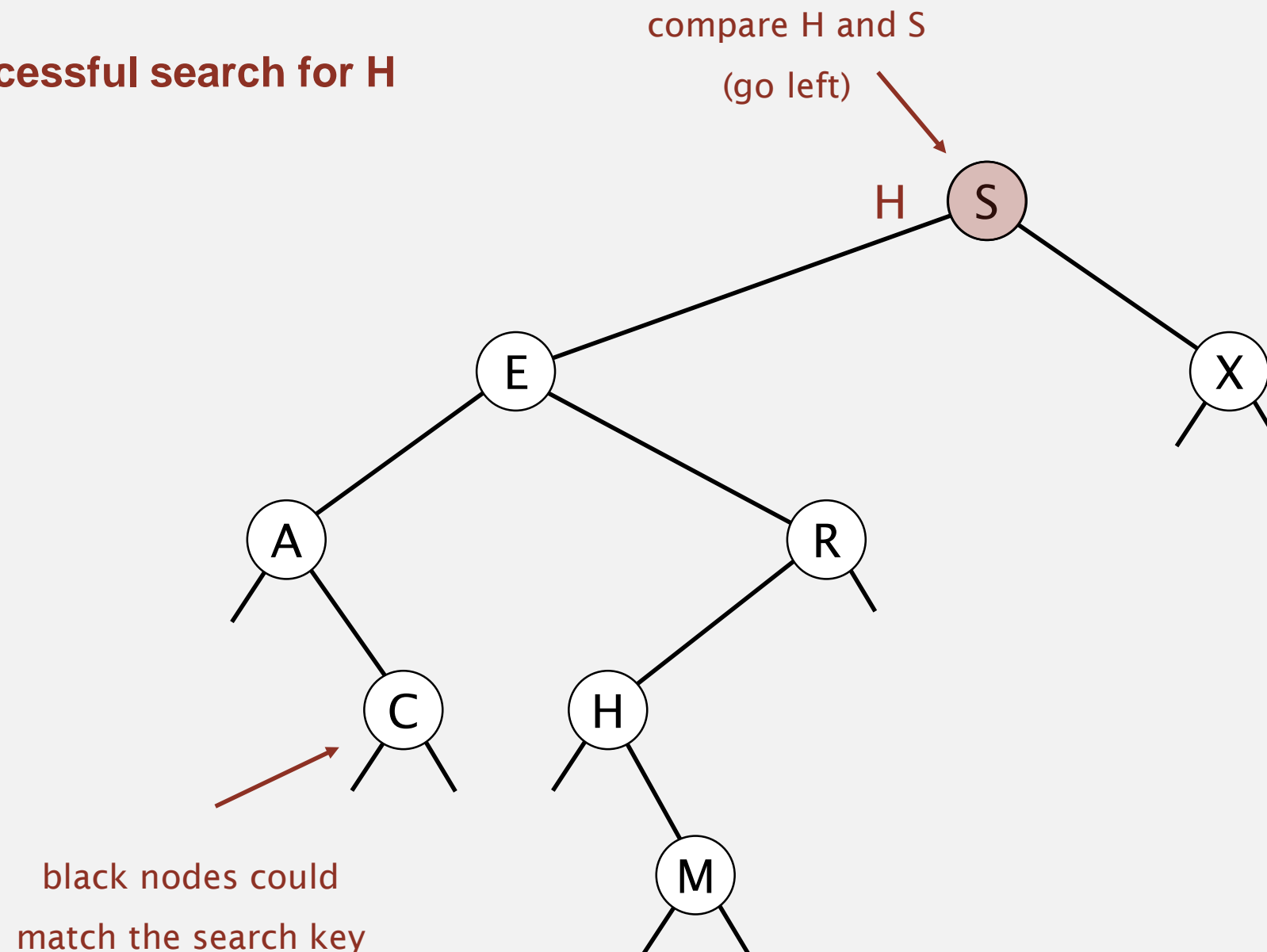
successful search for H



Binary search tree demo

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

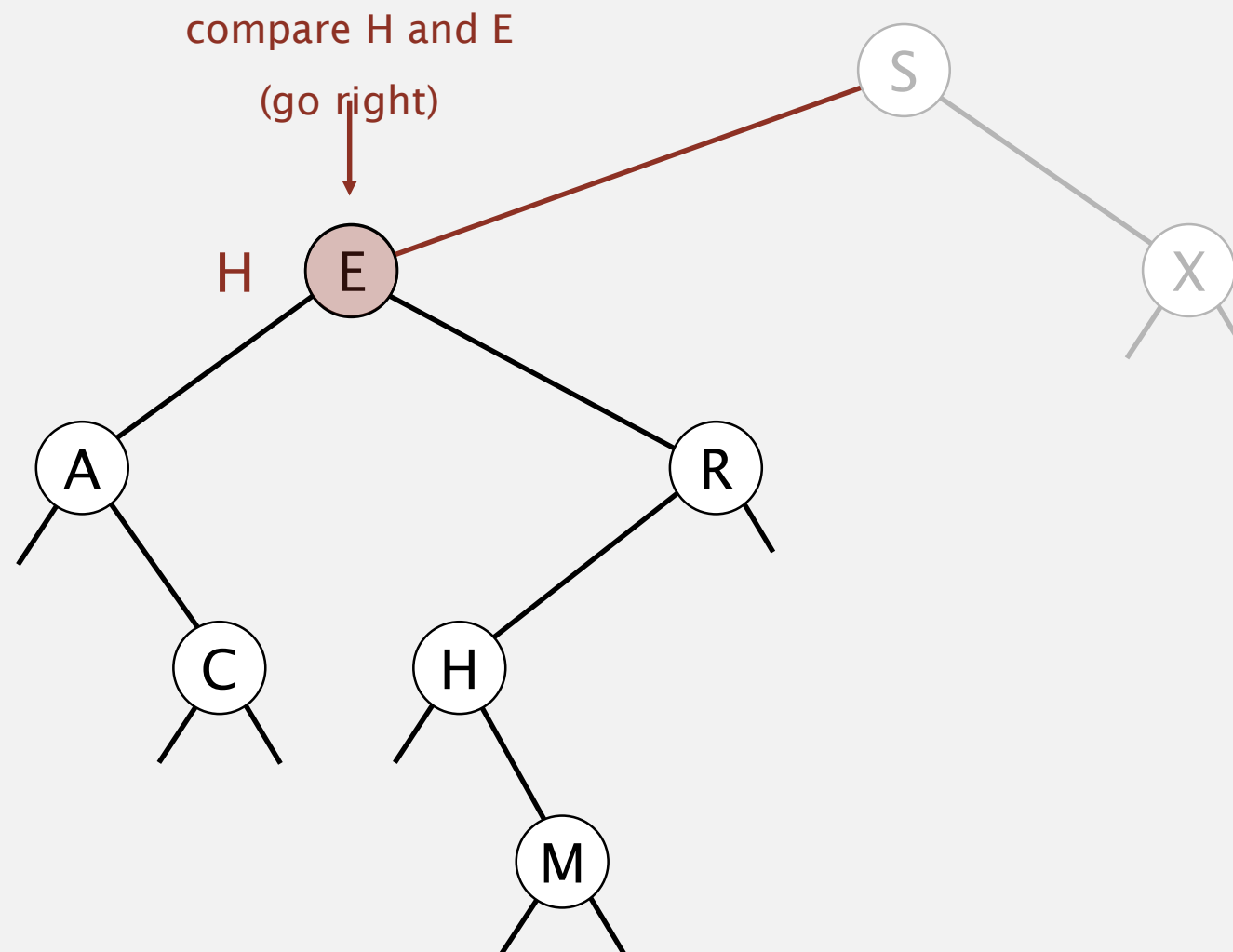
successful search for H



Binary search tree demo

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

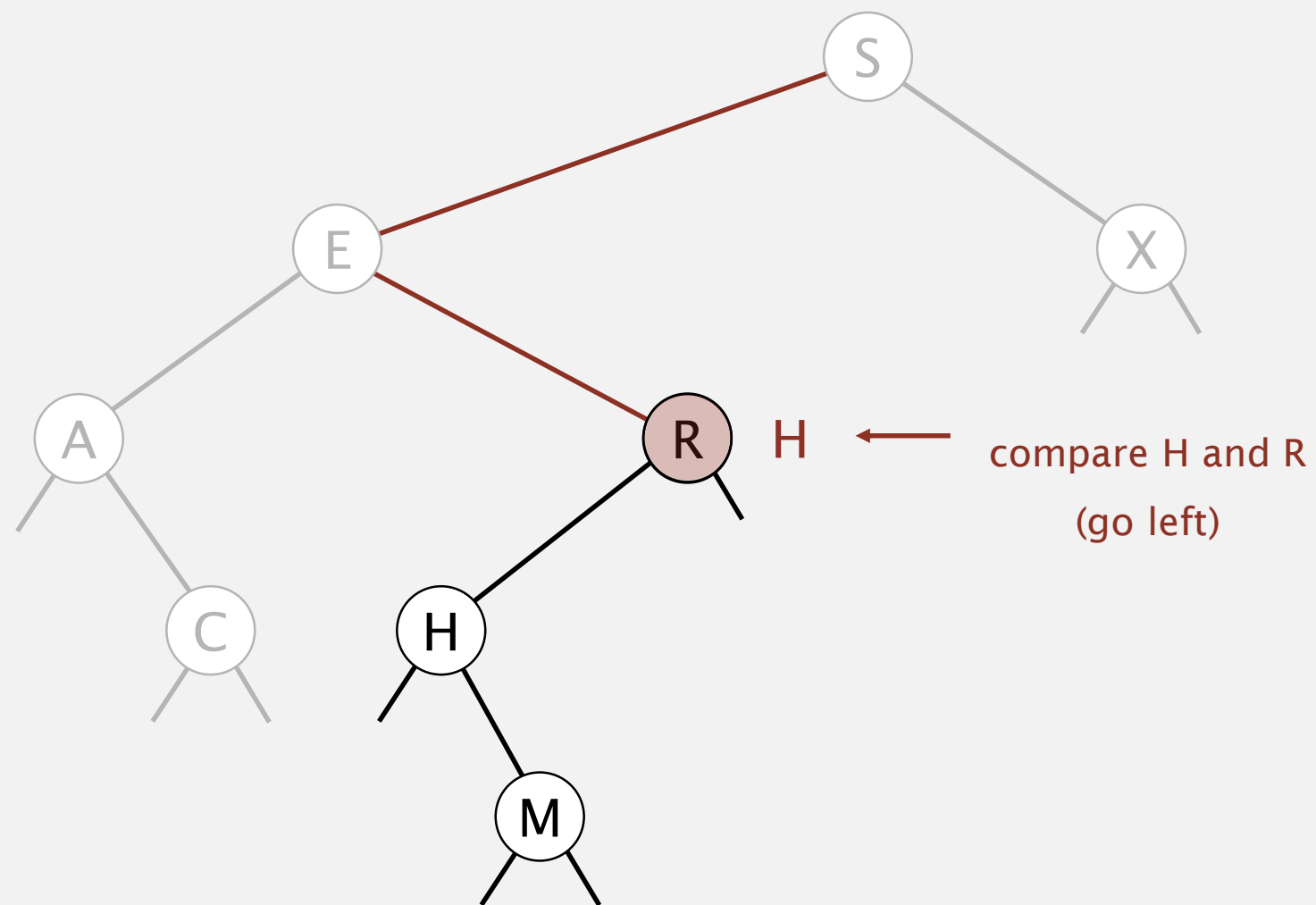
successful search for H



Binary search tree demo

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

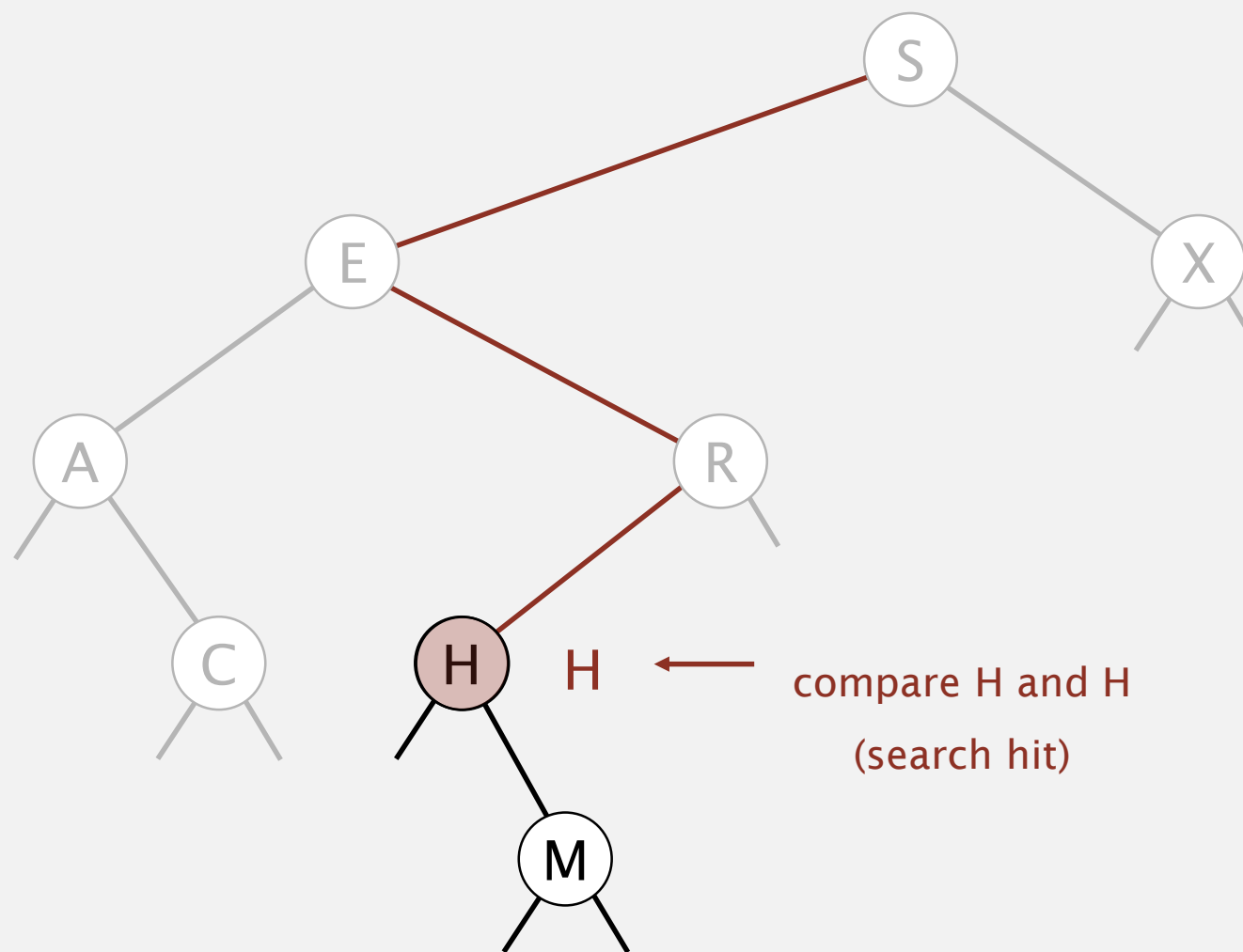
successful search for H



Binary search tree demo

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

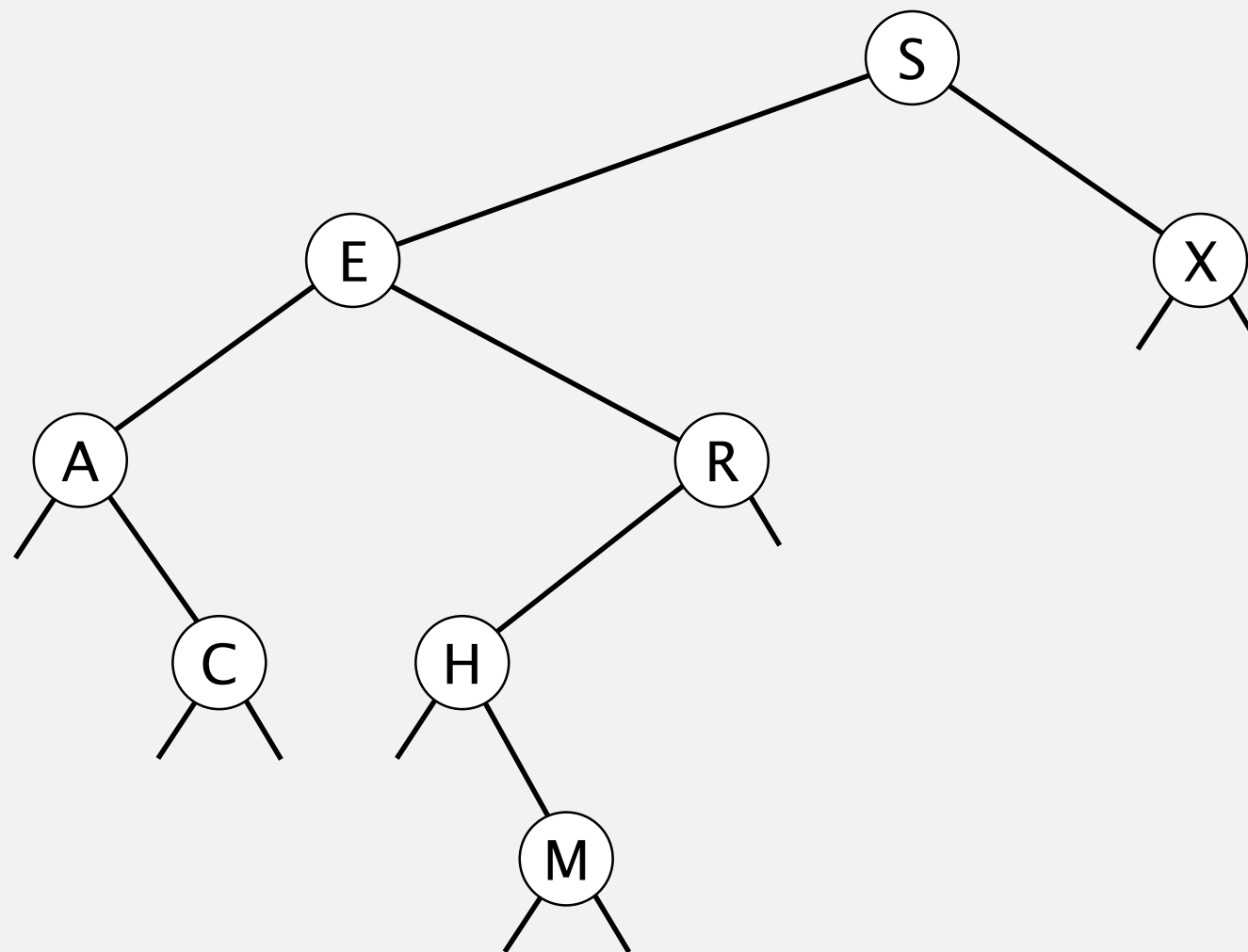
successful search for H



Binary search tree demo

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

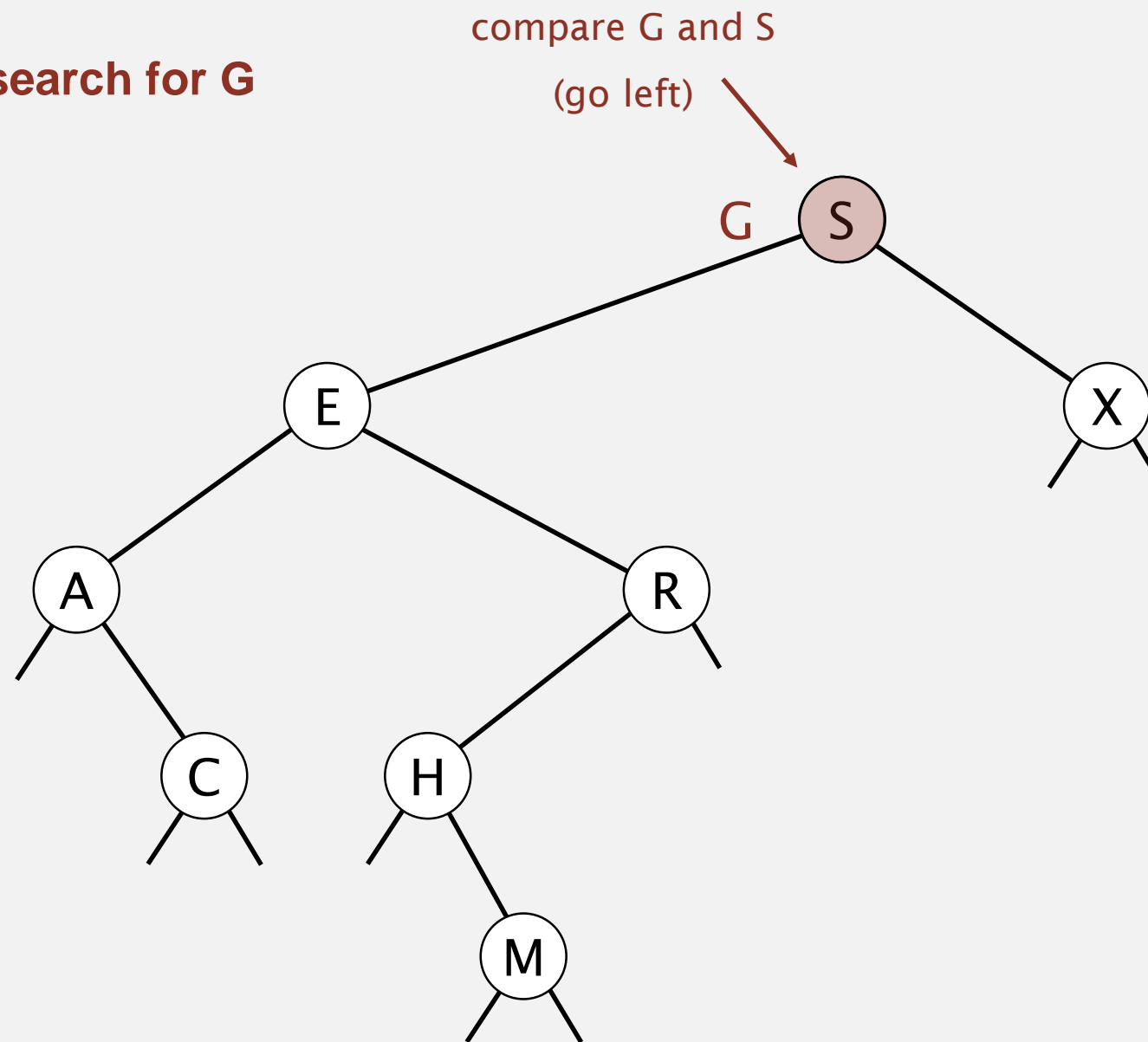
unsuccessful search for G



Binary search tree demo

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

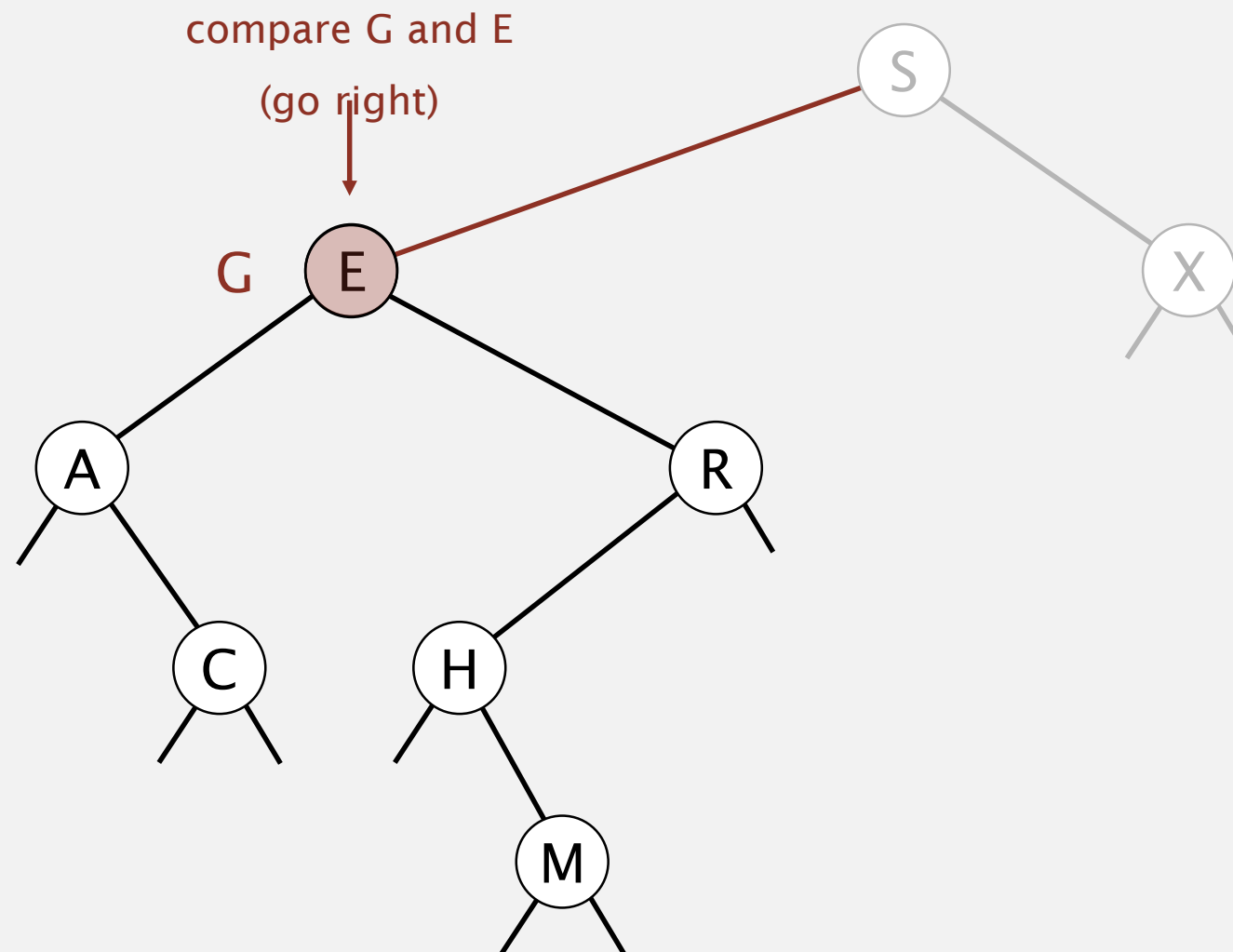
unsuccessful search for G



Binary search tree demo

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

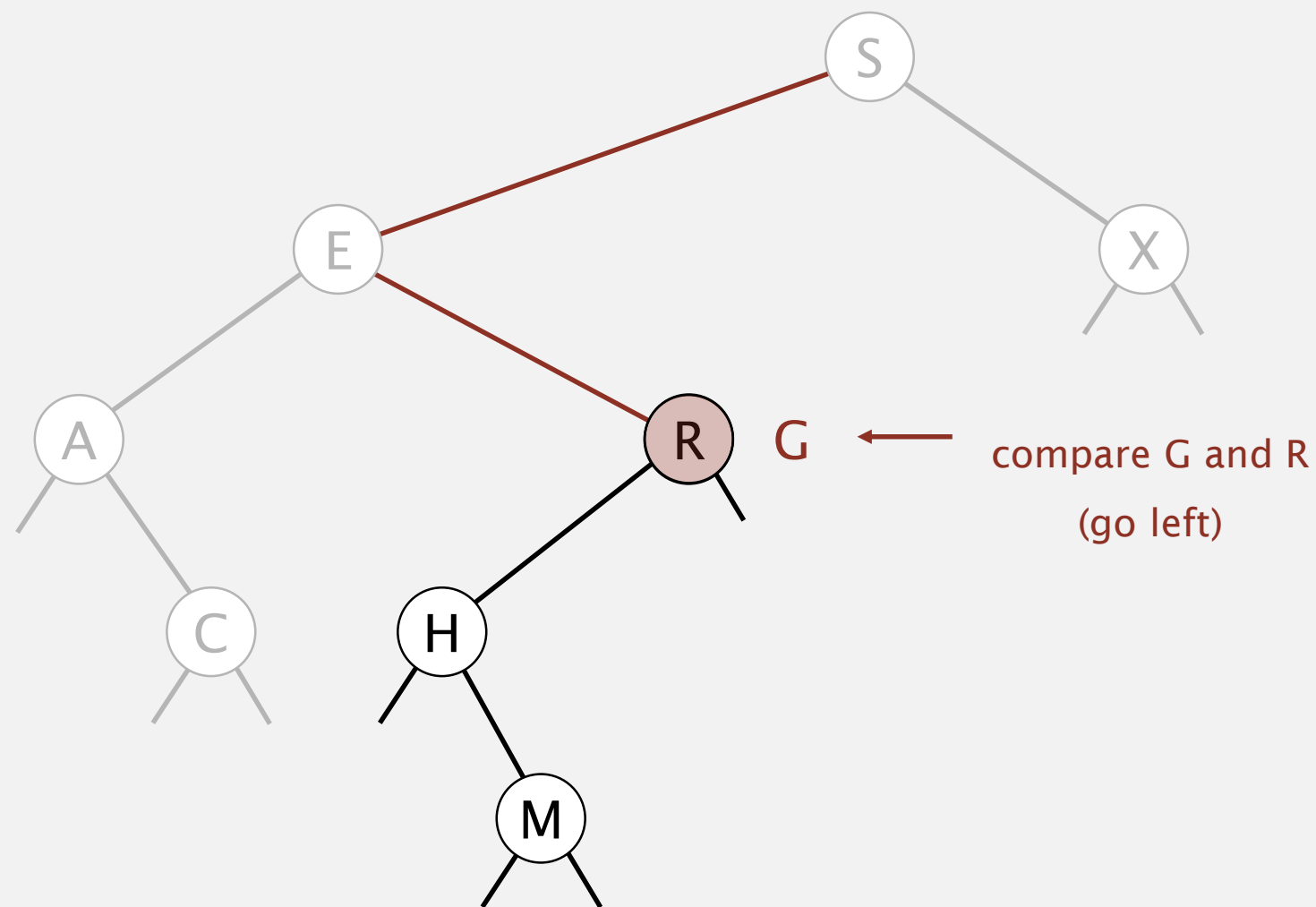
unsuccessful search for G



Binary search tree demo

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

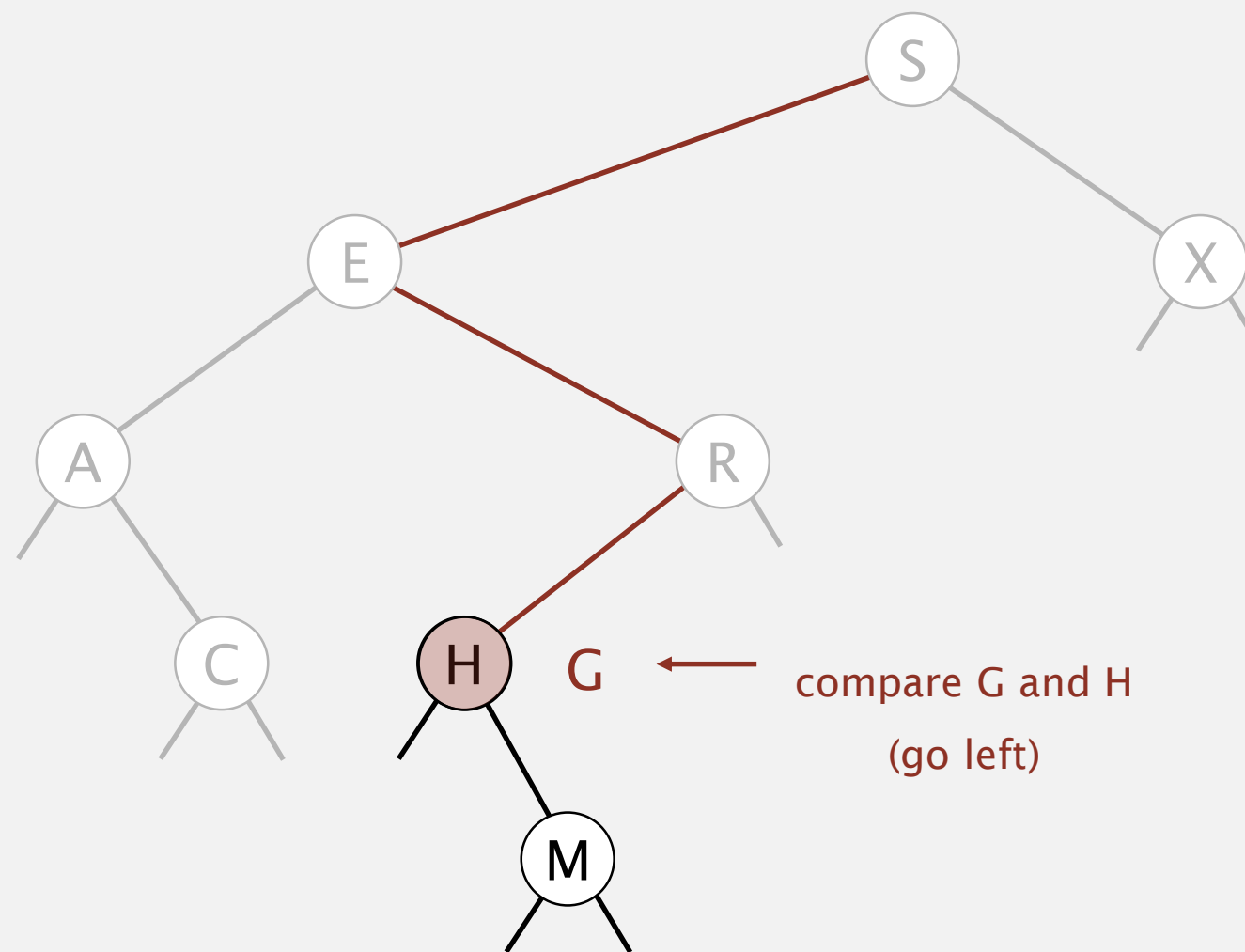
unsuccessful search for G



Binary search tree demo

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

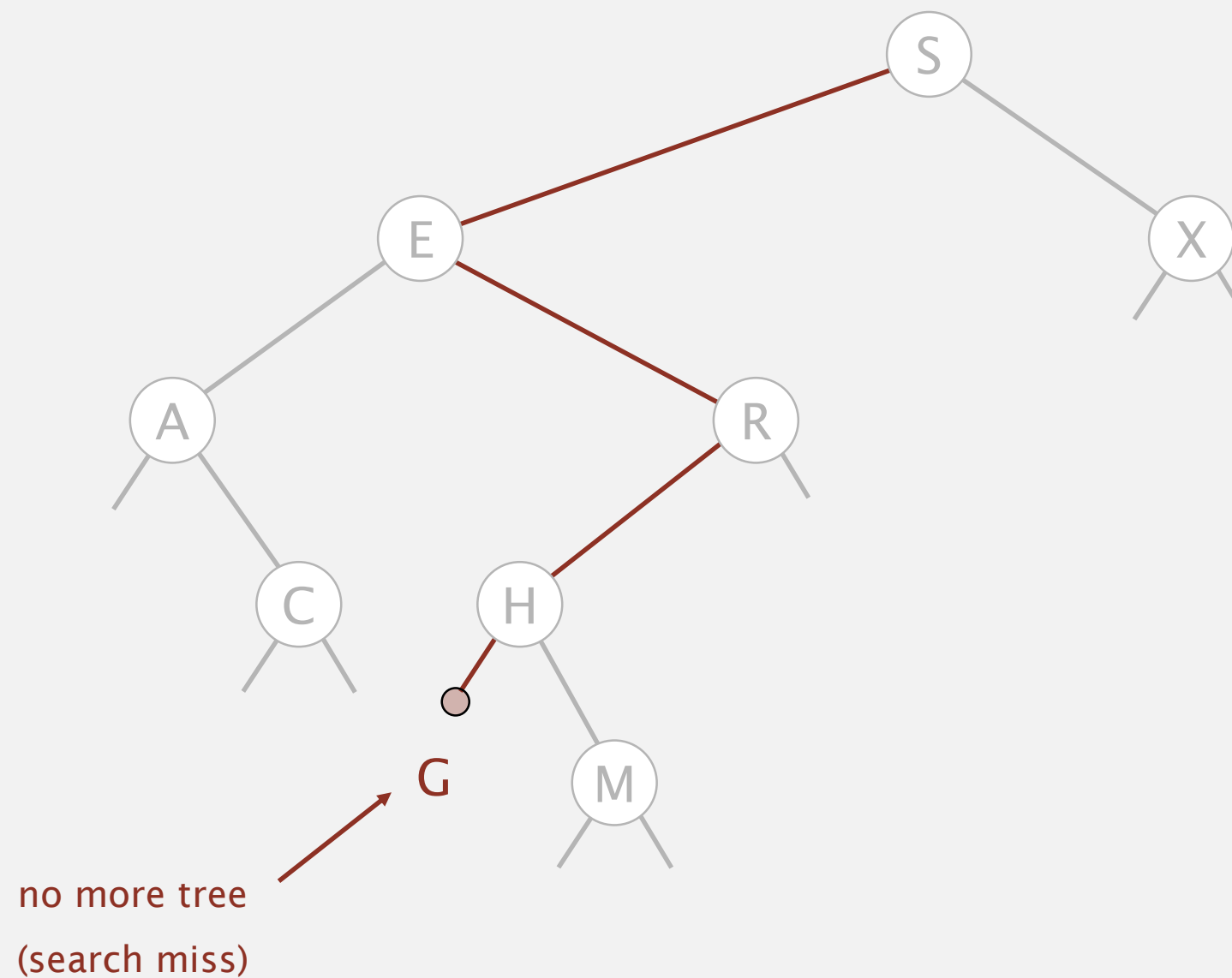
unsuccessful search for G



Binary search tree demo

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

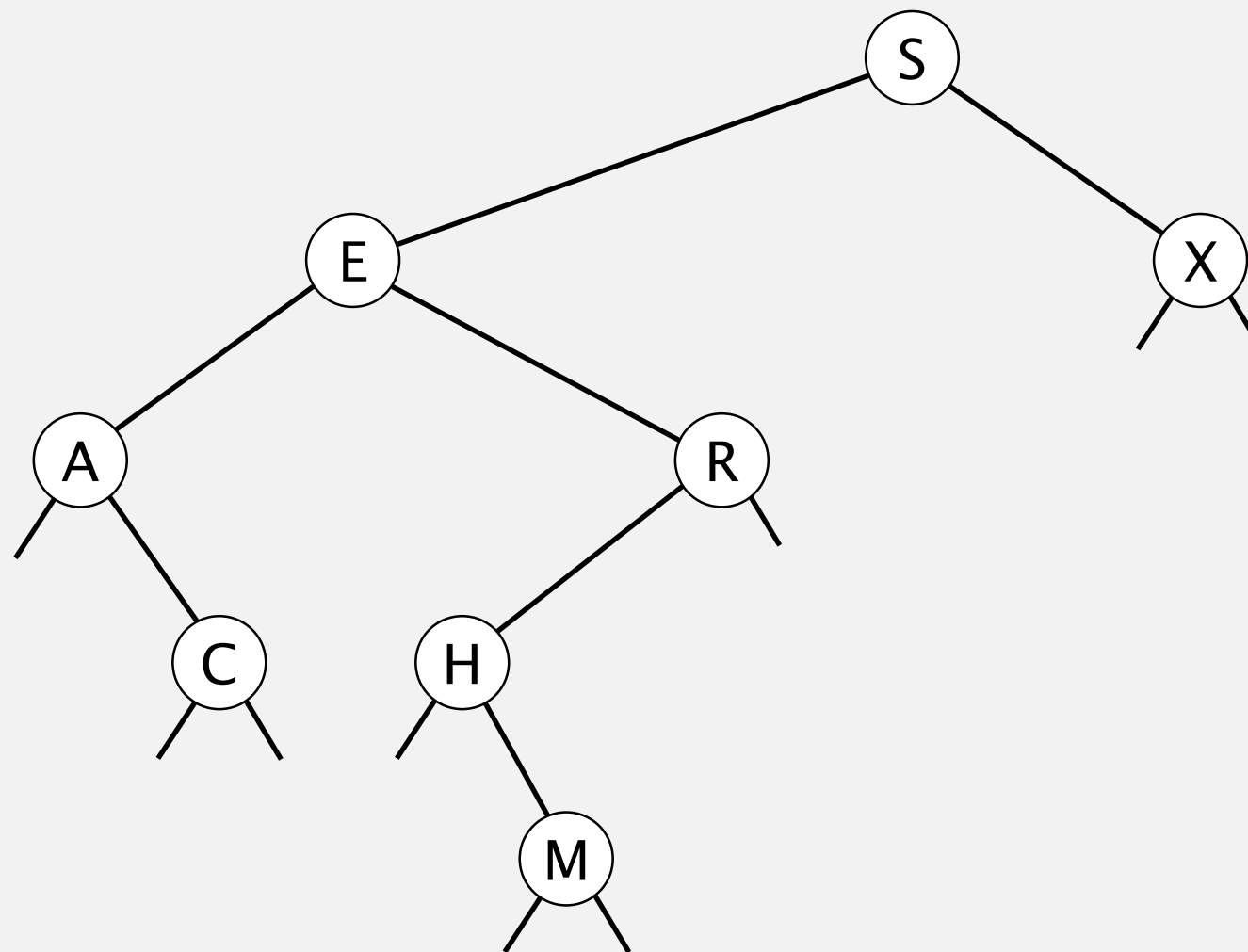
unsuccessful search for G



Binary search tree demo

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

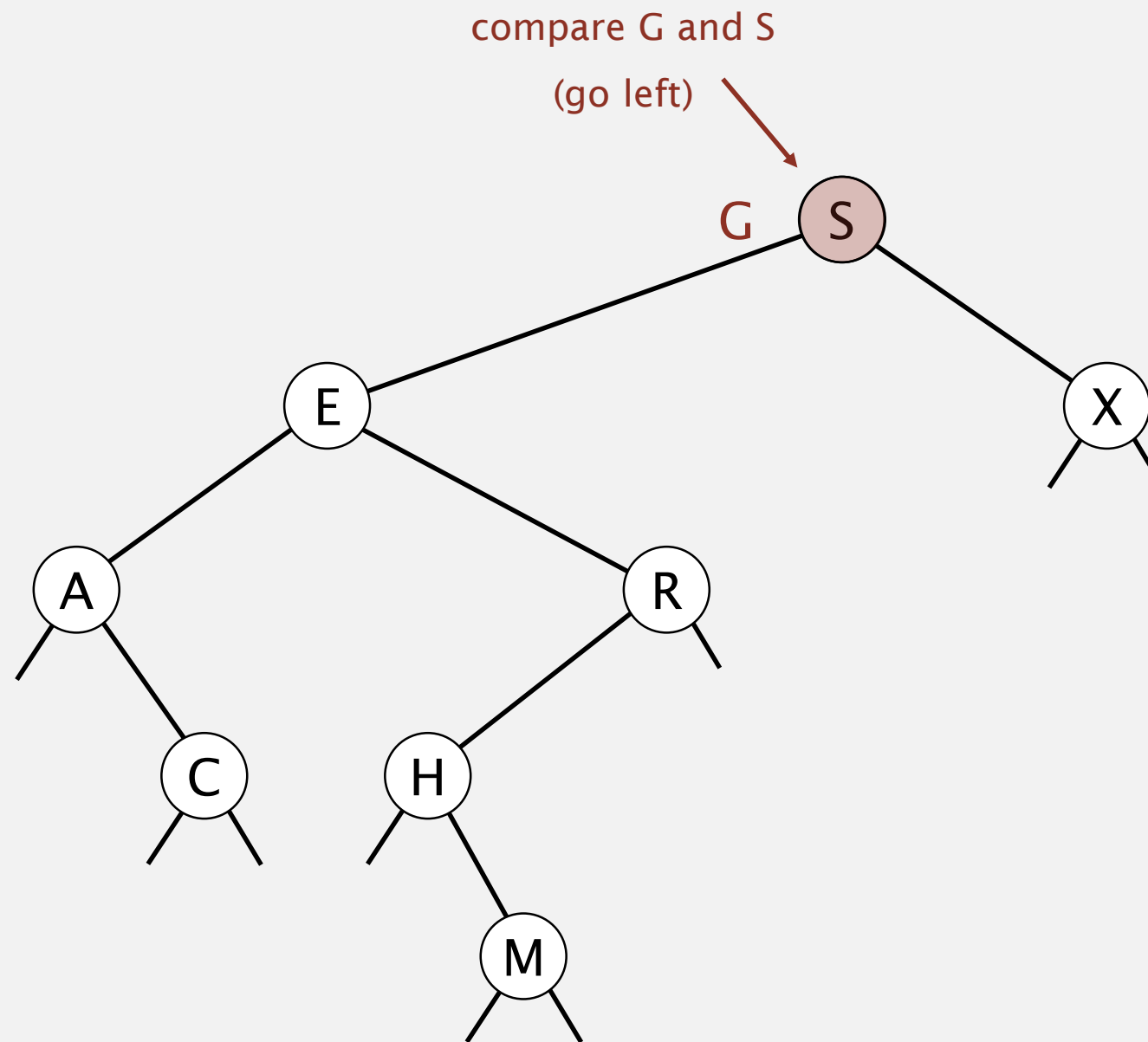
insert G



Binary search tree demo

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

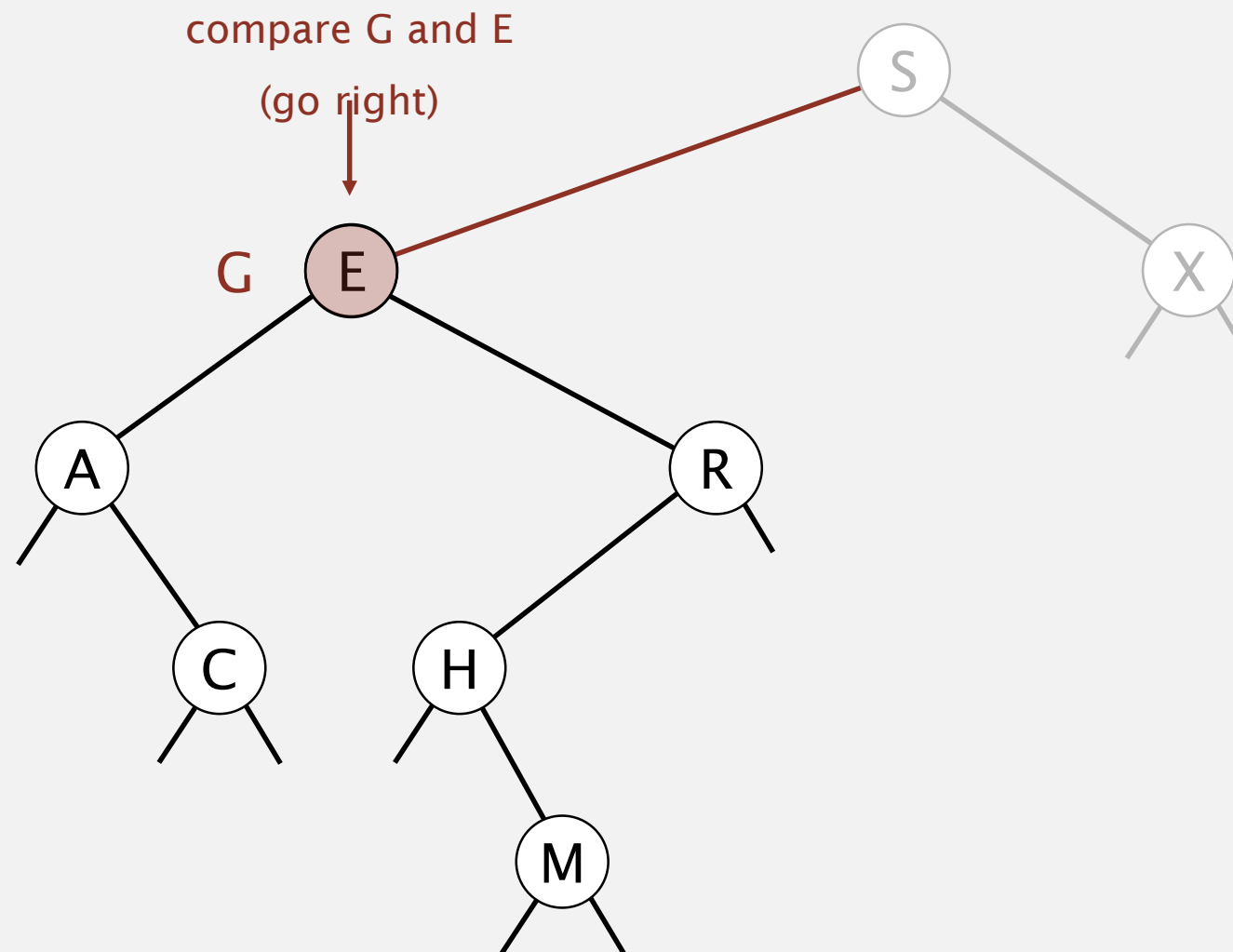
insert G



Binary search tree demo

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

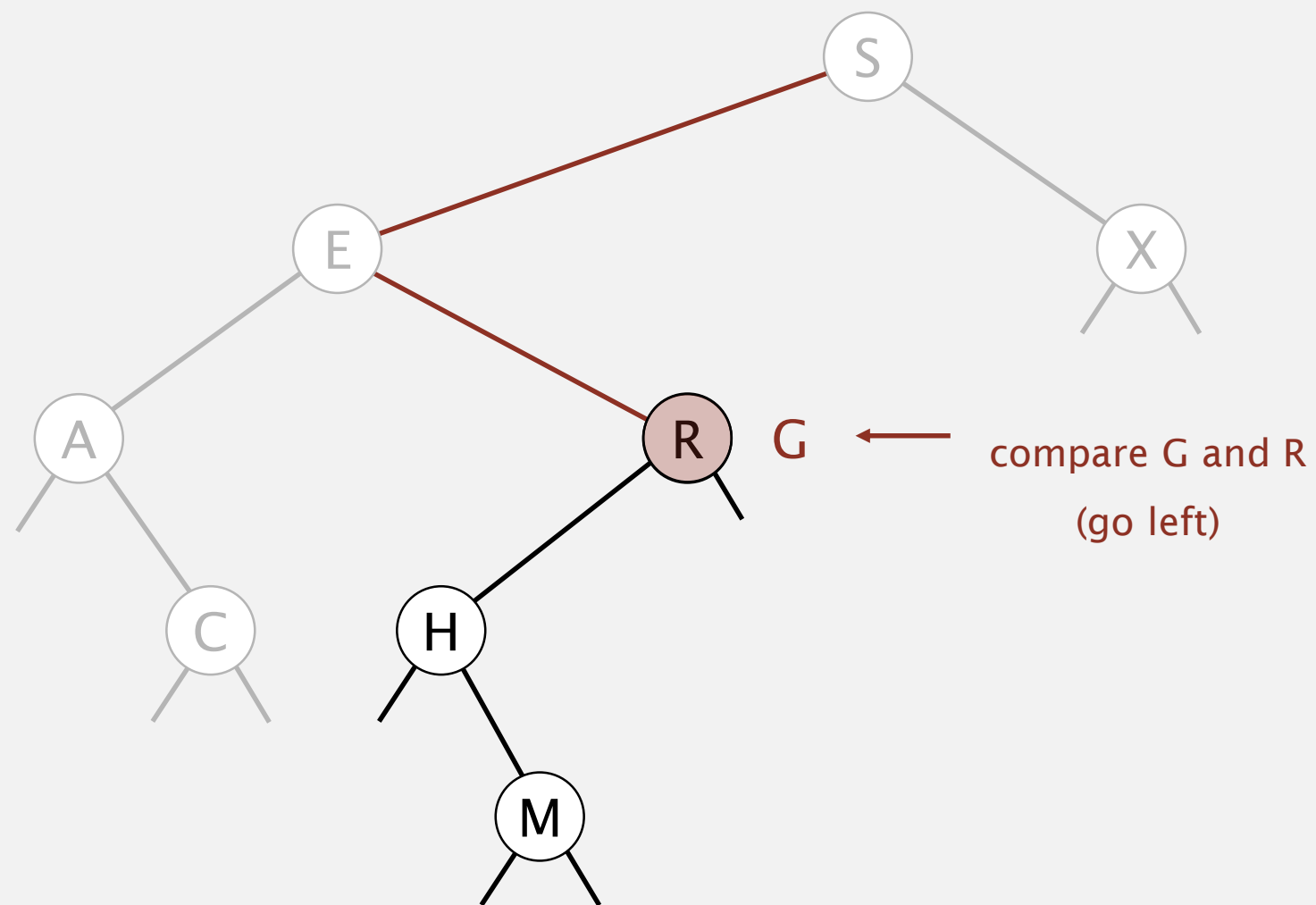
insert G



Binary search tree demo

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

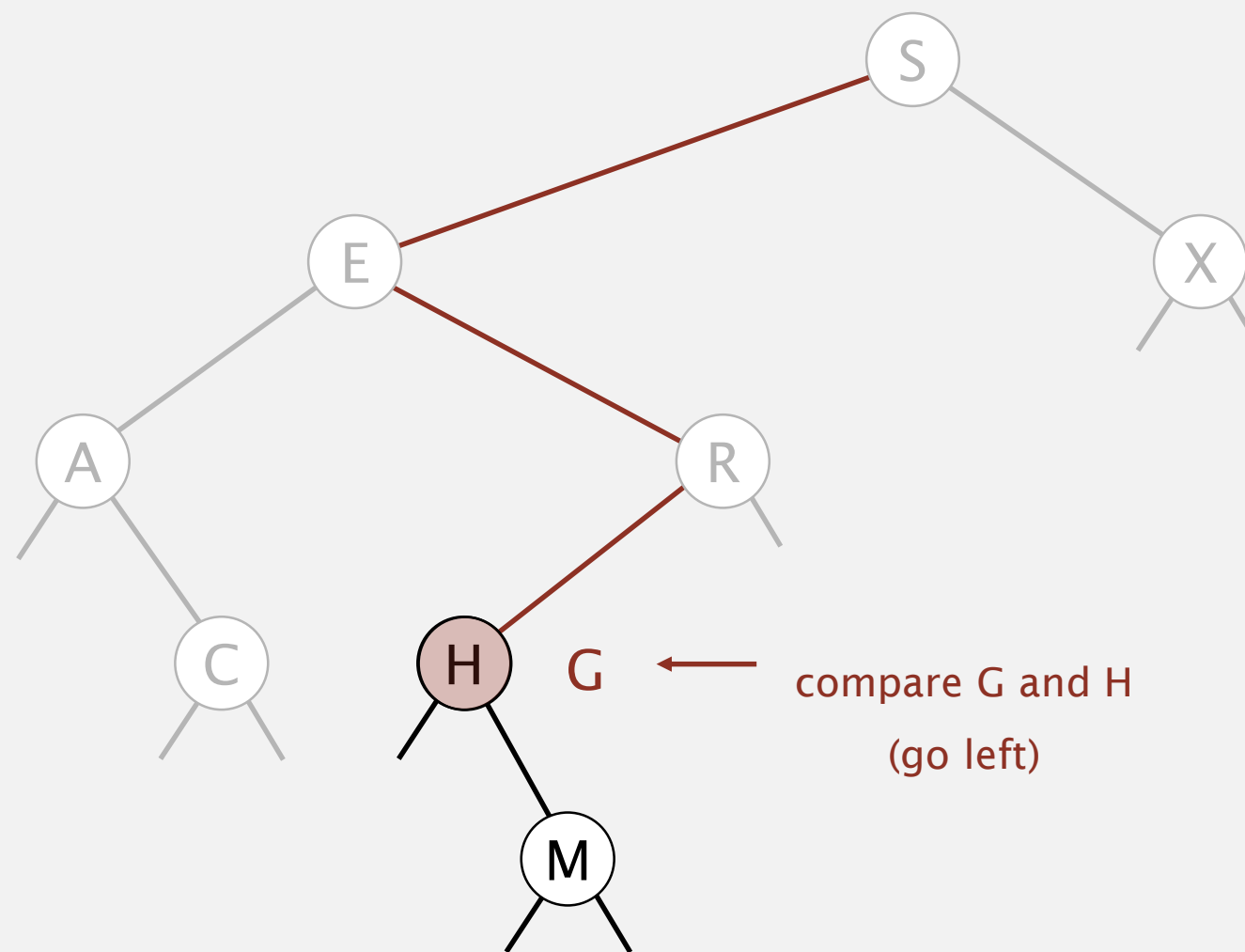
insert G



Binary search tree demo

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

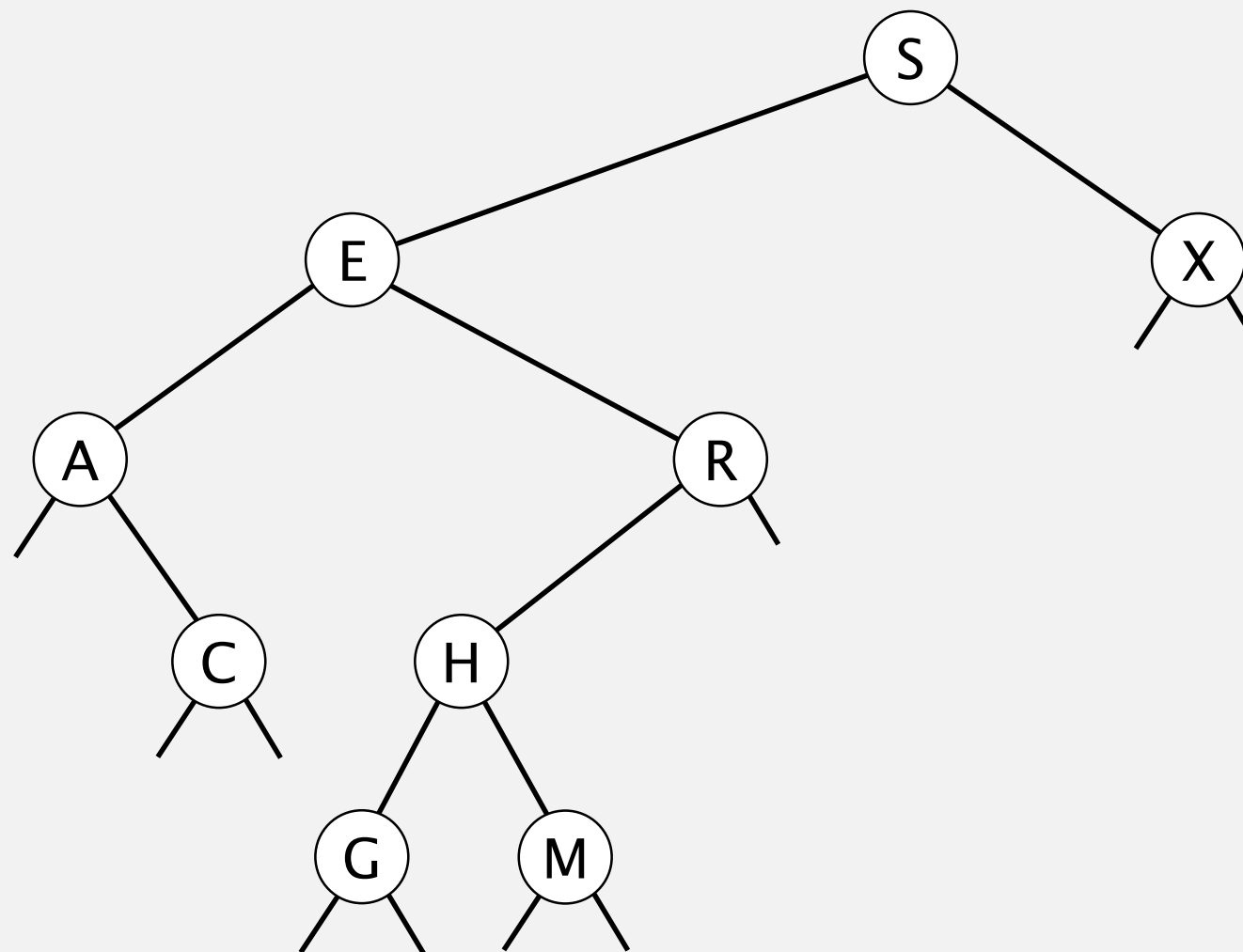
insert G



Binary search tree demo

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

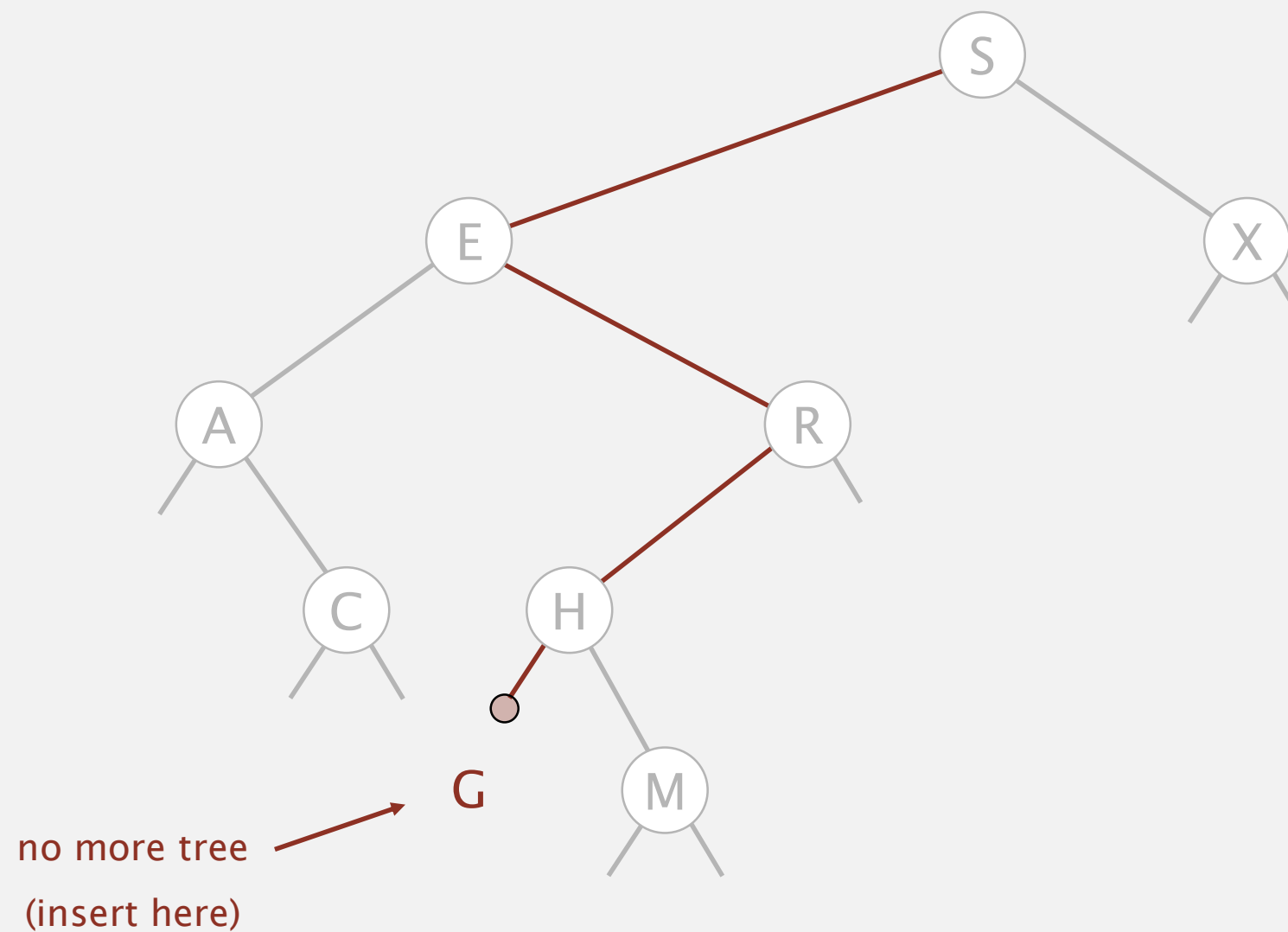
insert G



Binary search tree demo

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

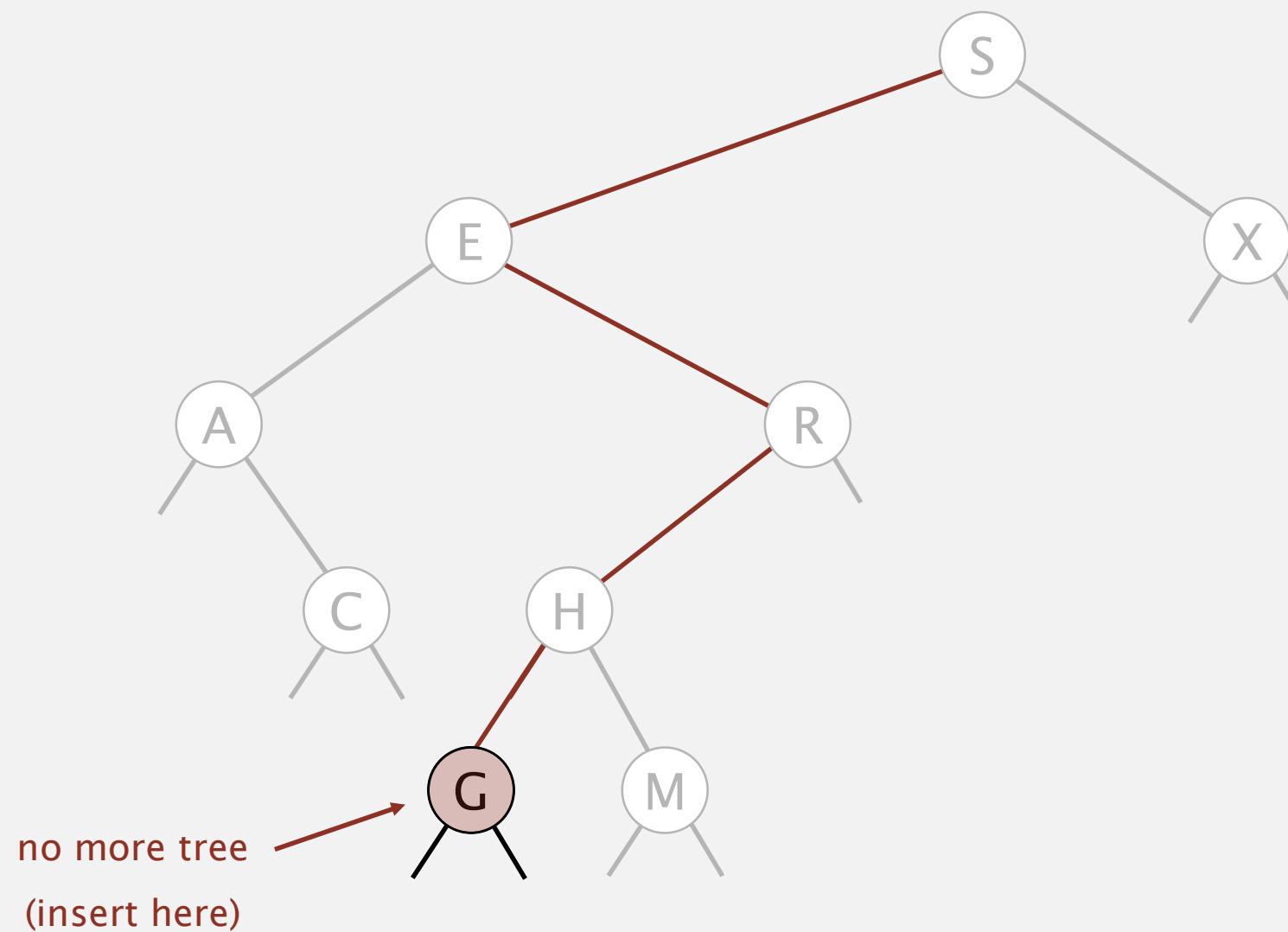
insert G



Binary search tree demo

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

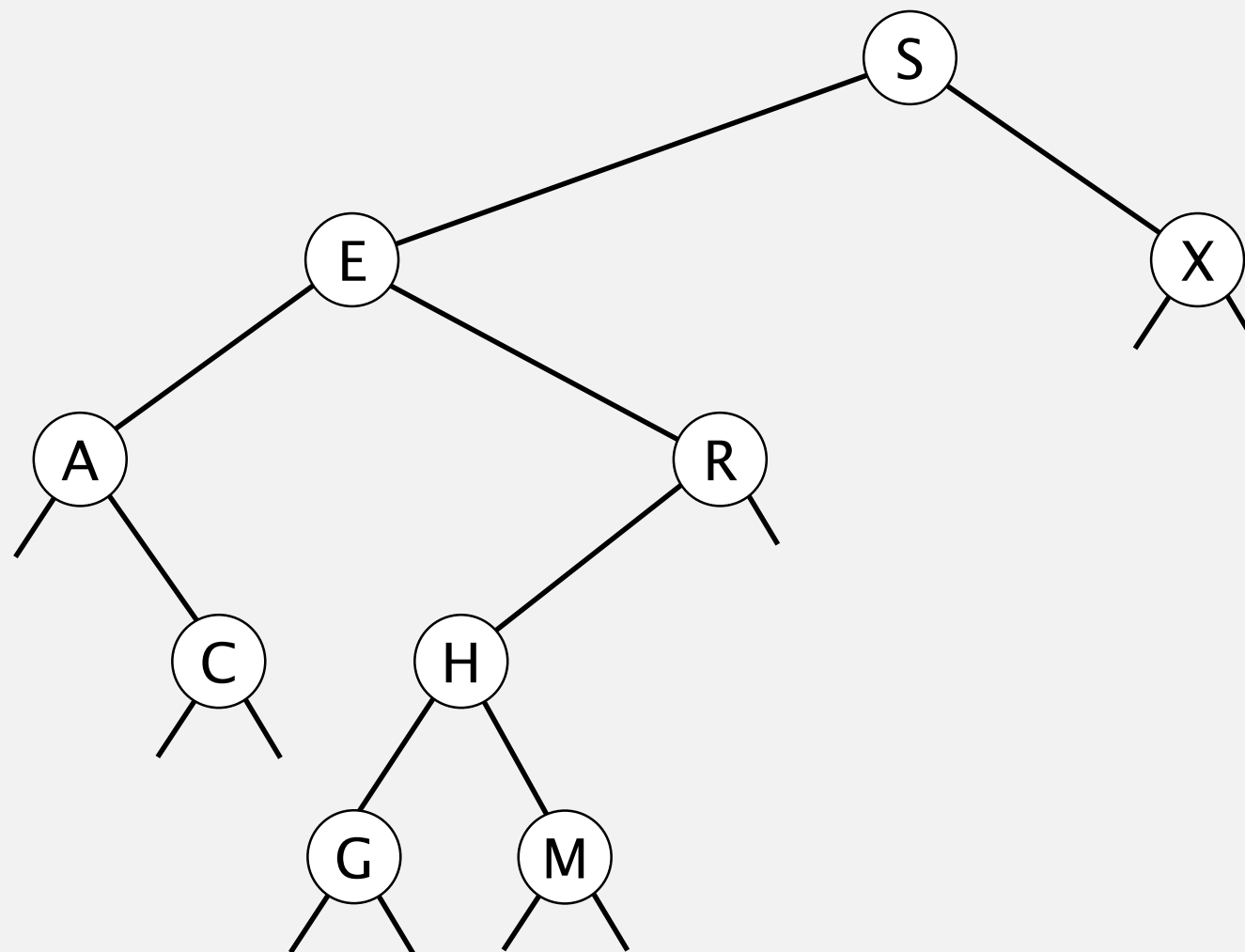
insert G



Binary search tree demo

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

insert G



BST representation in Java

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

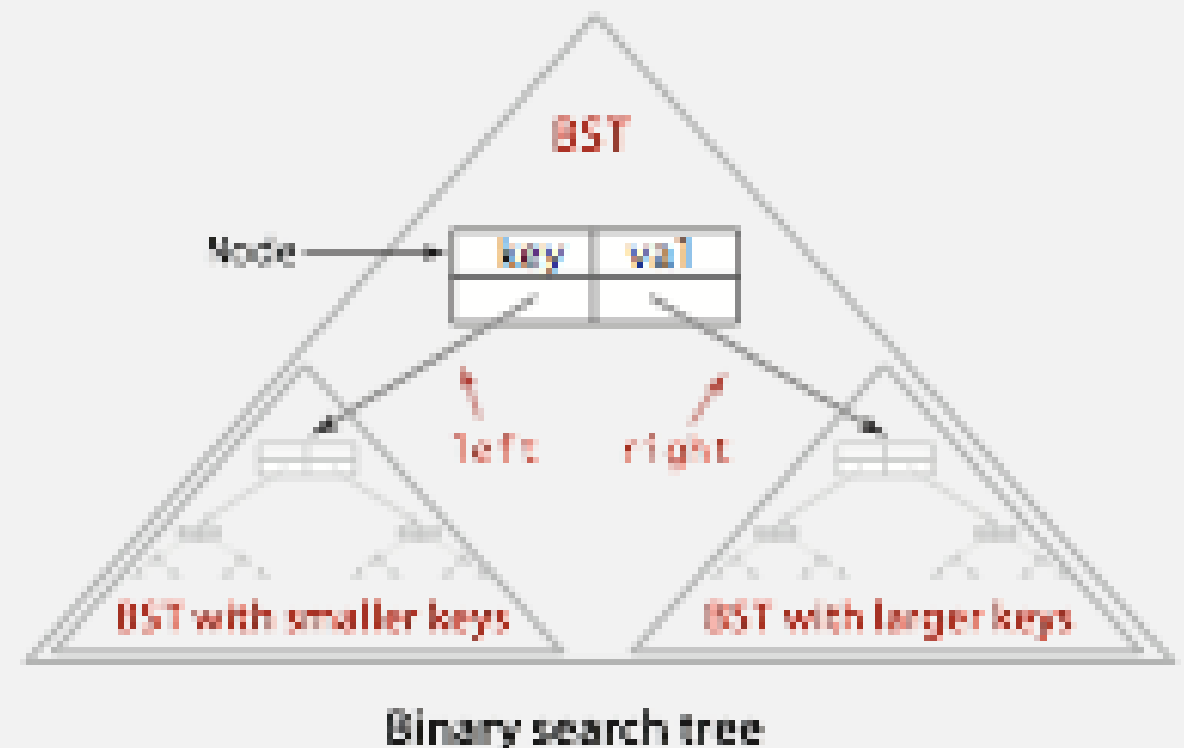
A Node is composed of four fields:

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

↑ ↑
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;


    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: 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.

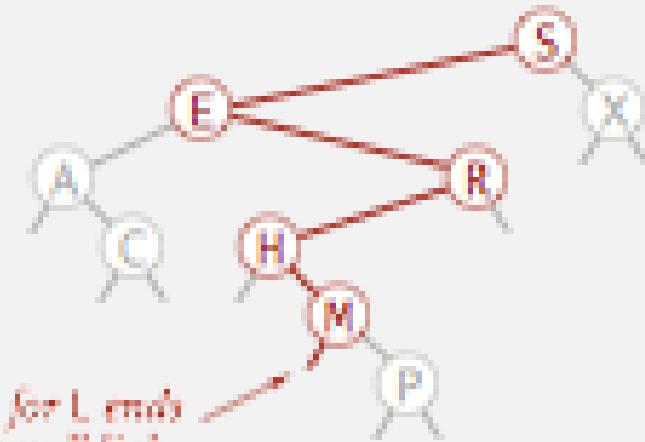
BST insert

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



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;
}
```

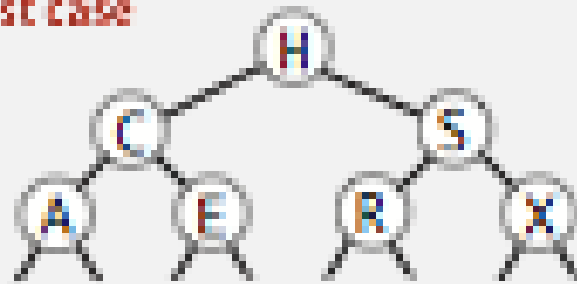
concise, but tricky,
recursive code;
read carefully!

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

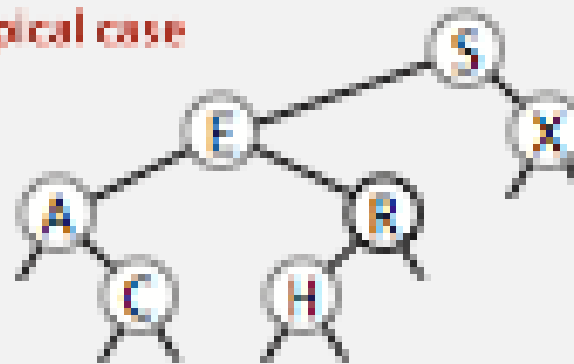
Tree shape

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

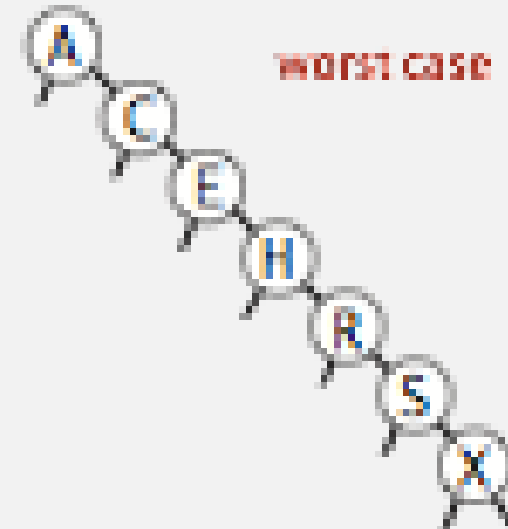
best case



typical case



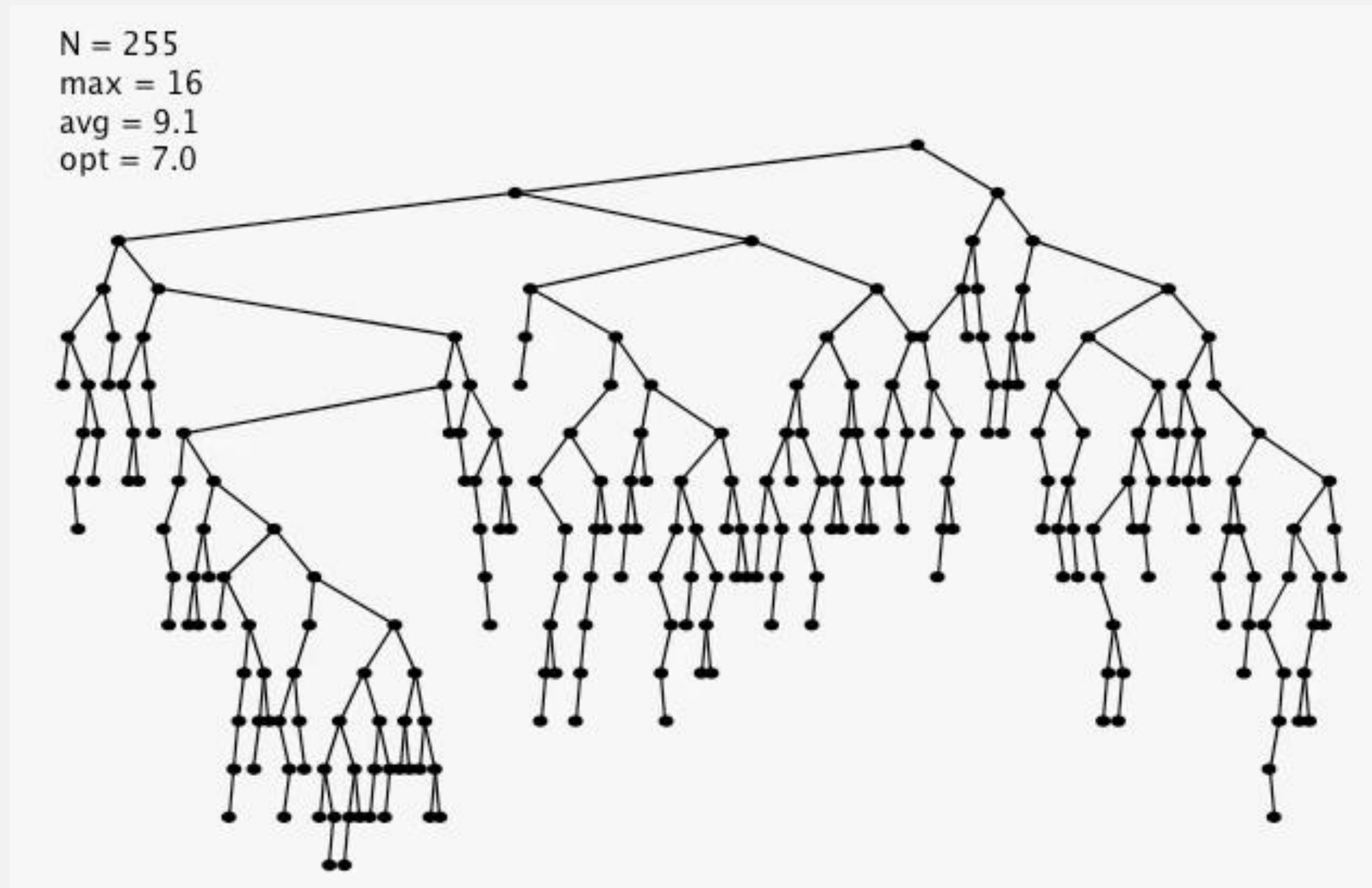
worst case



Bottom line. Tree shape depends on order of insertion.

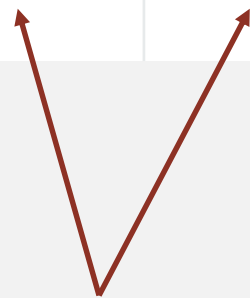
BST insertion: random order visualization

Ex. Insert keys 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	equals()
binary search (ordered array)	$\lg N$	N	$\lg N$	$\frac{1}{2} N$	compareTo()
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	compareTo()



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



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

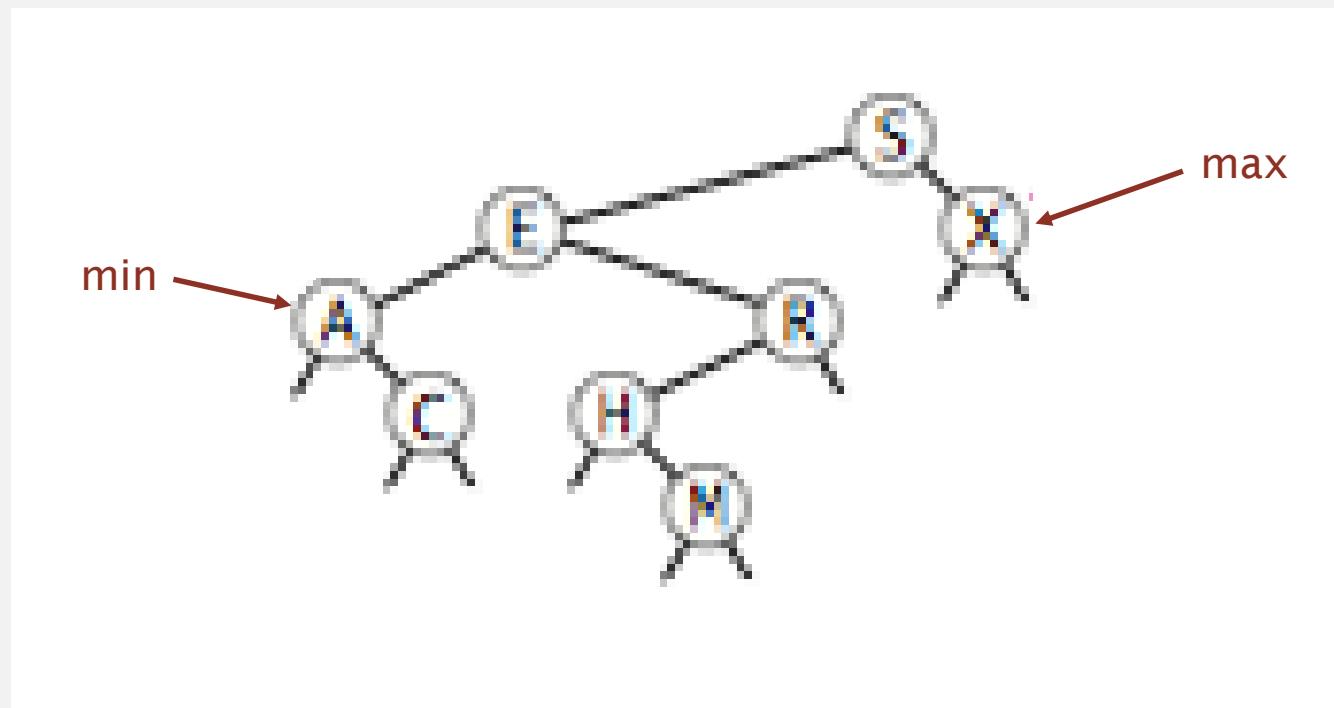
3.2 BINARY SEARCH TREES

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

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

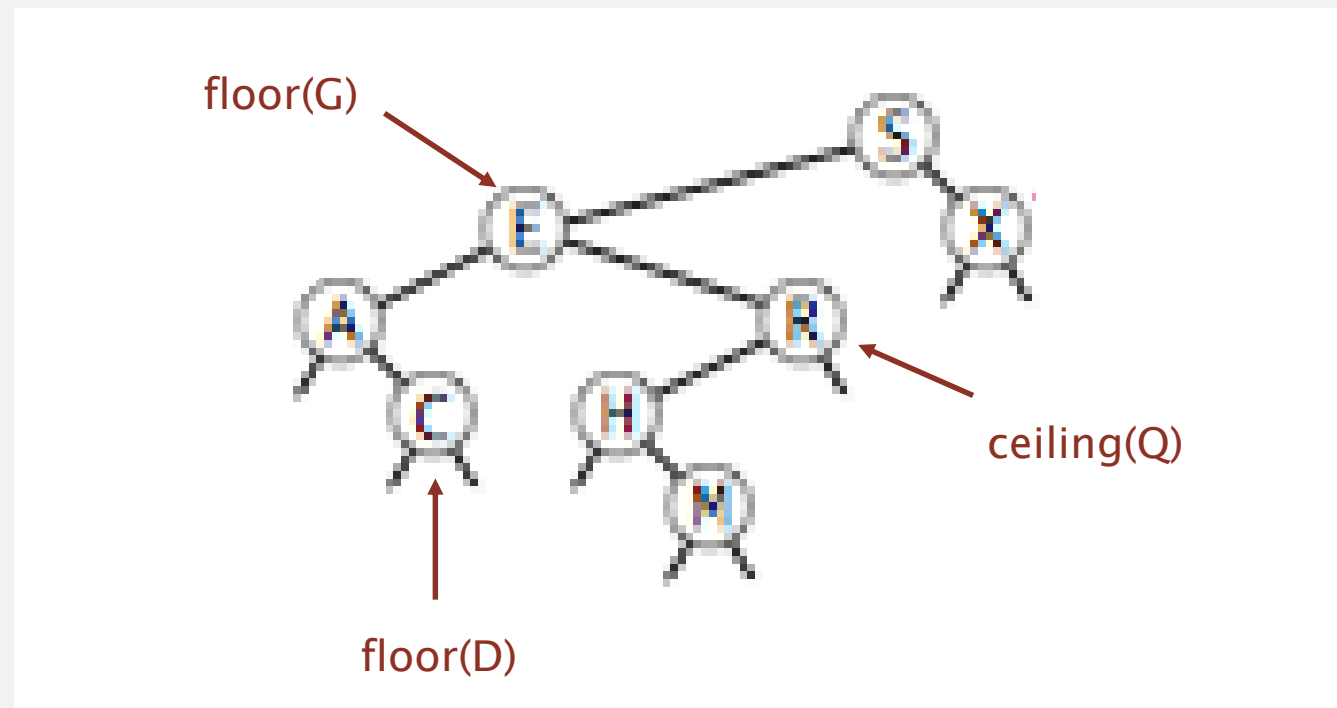


Q. How to find the min / max?

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 .

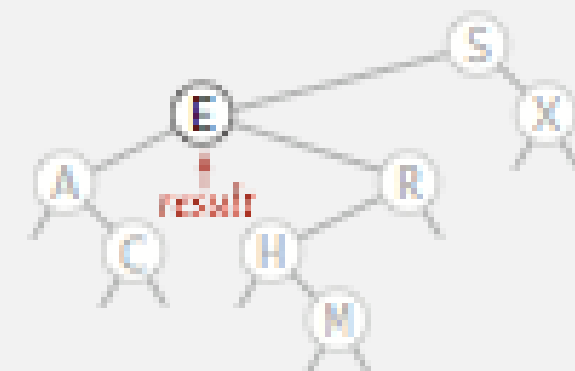
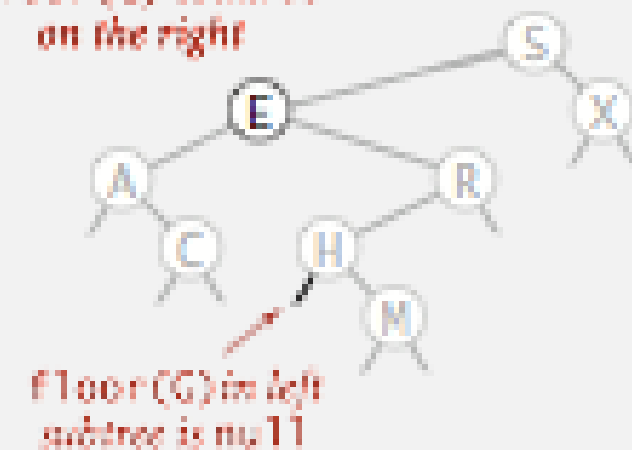
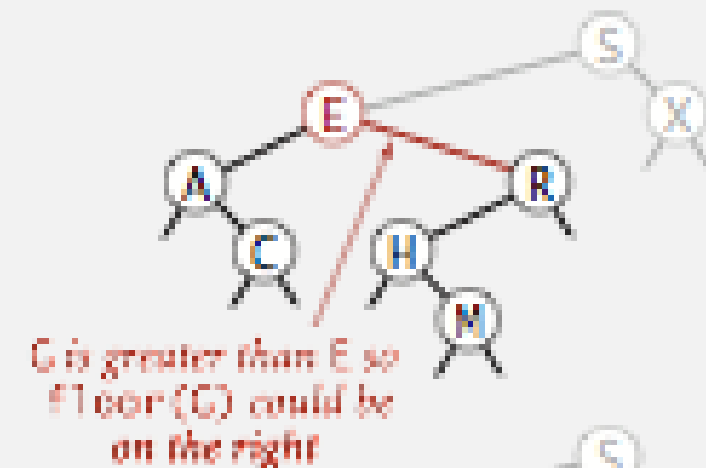
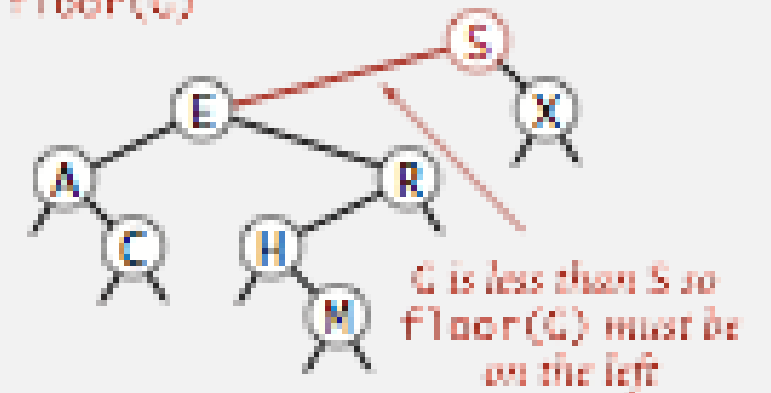
Case 2. [k is less than the key in the node]

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

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

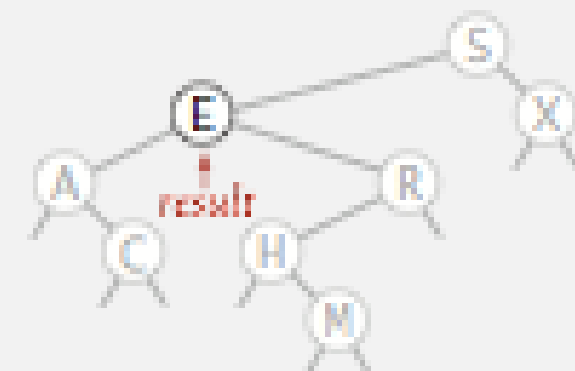
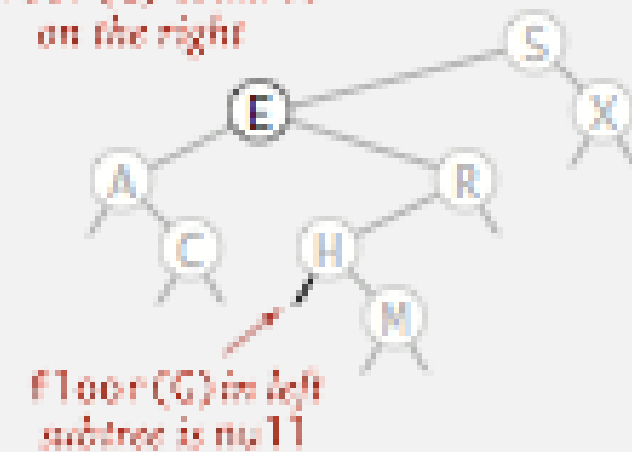
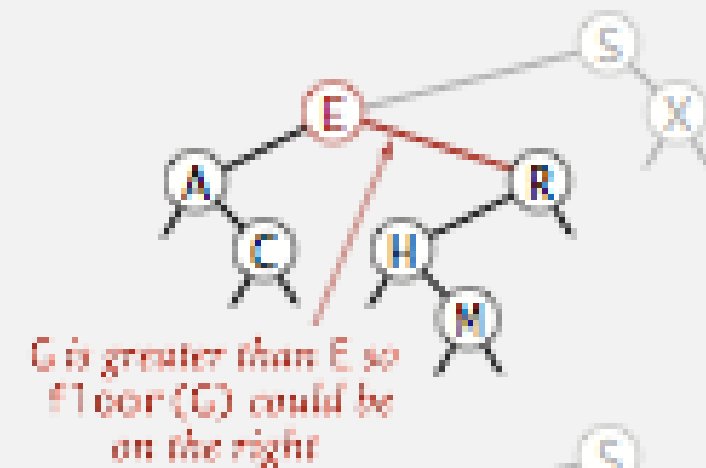
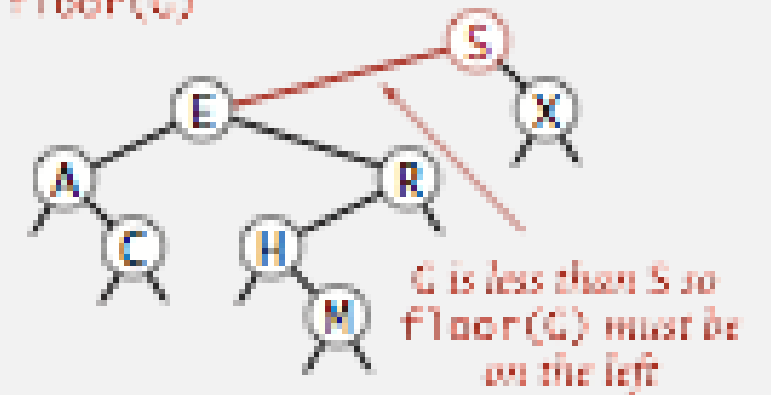
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    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)

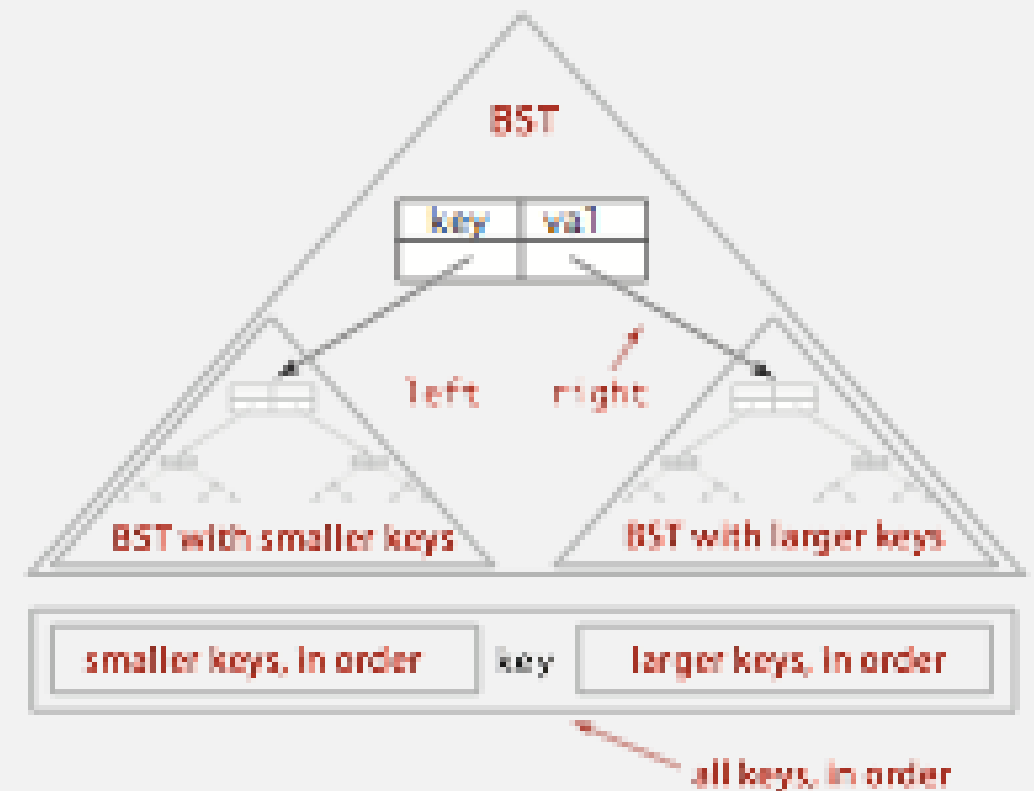


Inorder traversal

- 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 ^[SEP] search	binary ^[SEP] 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



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

3.2 BINARY SEARCH TREES

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

ST implementations: summary

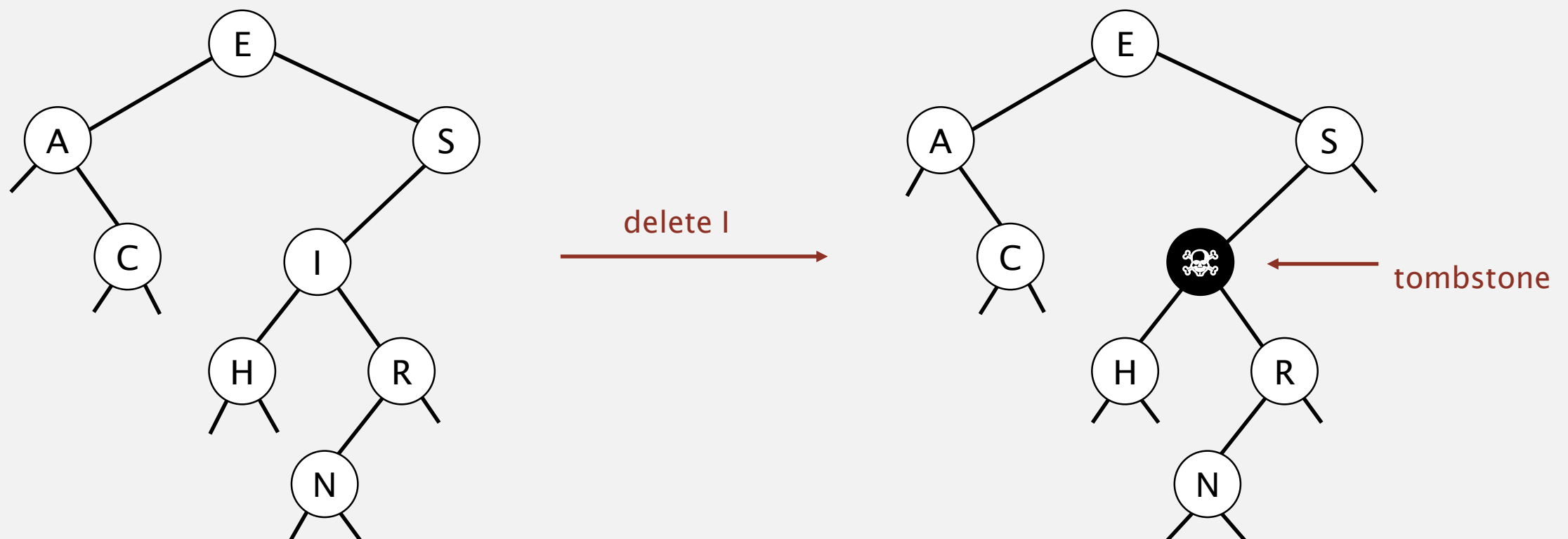
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search _[SEP] (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		equals()
binary search _[SEP] (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	compareTo()
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$???	✓	compareTo()

Next. Deletion in BSTs.

BST deletion: lazy approach

To remove a node with a given key:

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



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

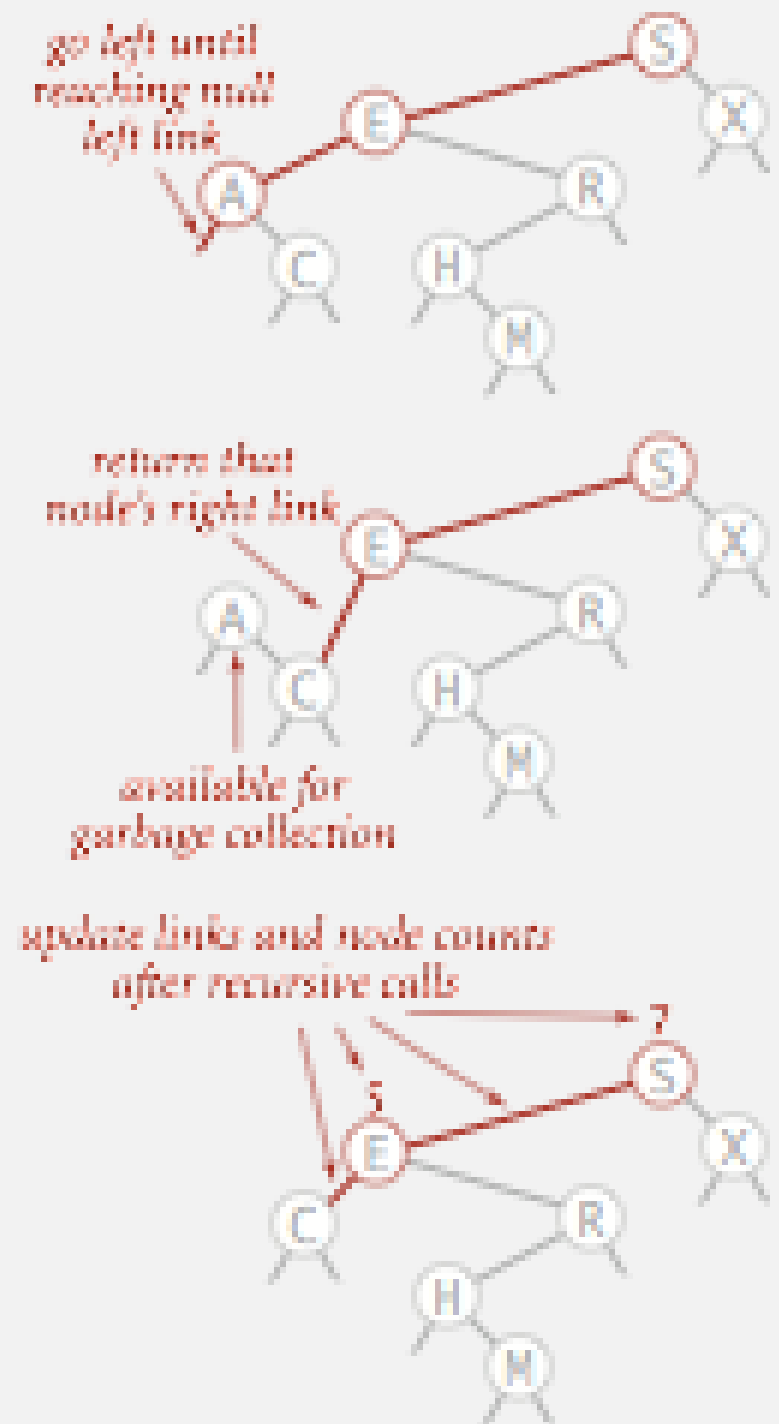
Unsatisfactory solution. Tombstone (memory) overload.

Deleting the minimum

To delete the minimum key:

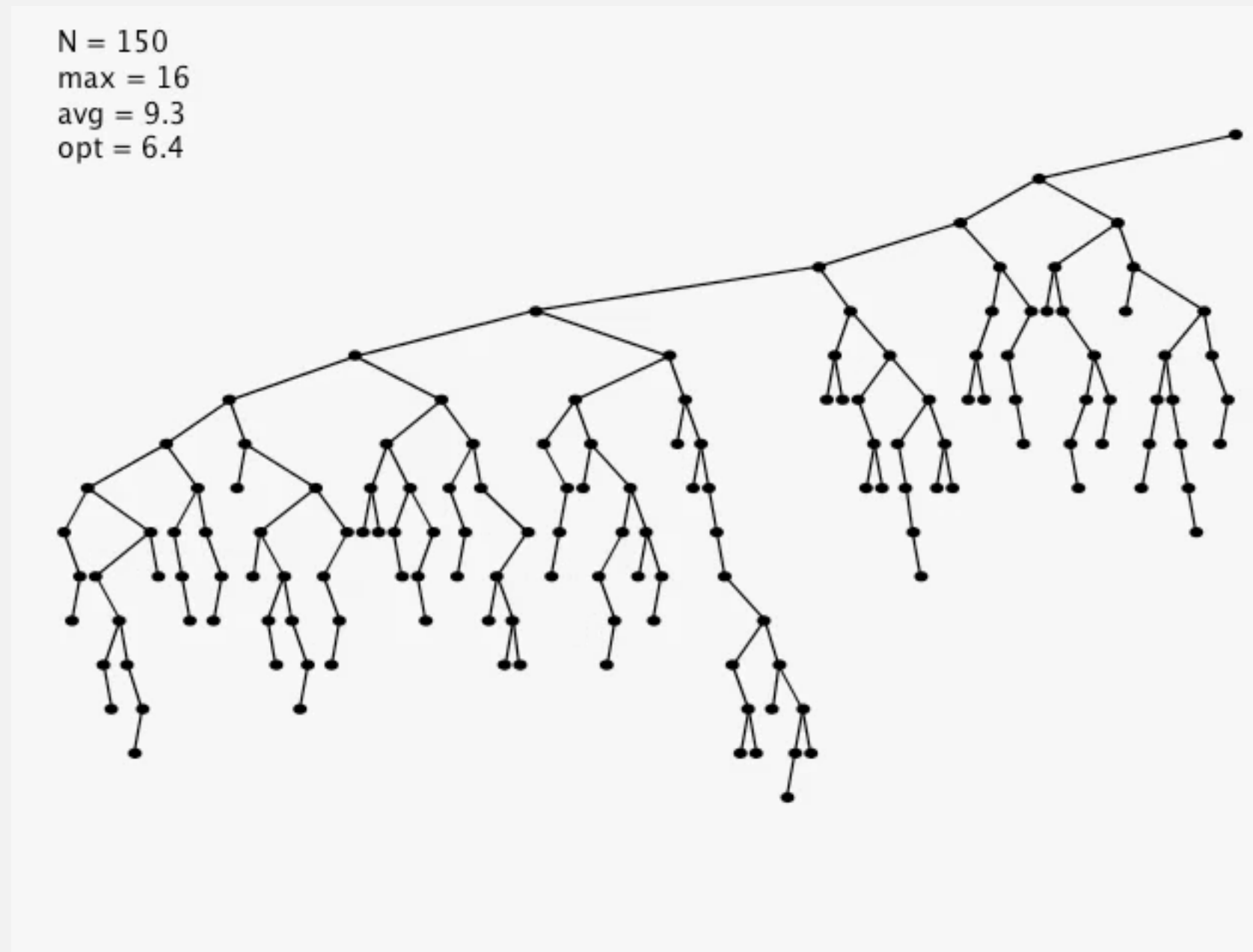
- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

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



Hibbard deletion: 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 _[SEP] (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		equals()
binary search _[SEP] (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	compareTo()
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	compareTo()

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

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

QUICKSORT



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

Quicksort

A binary search tree orders the elements by their key

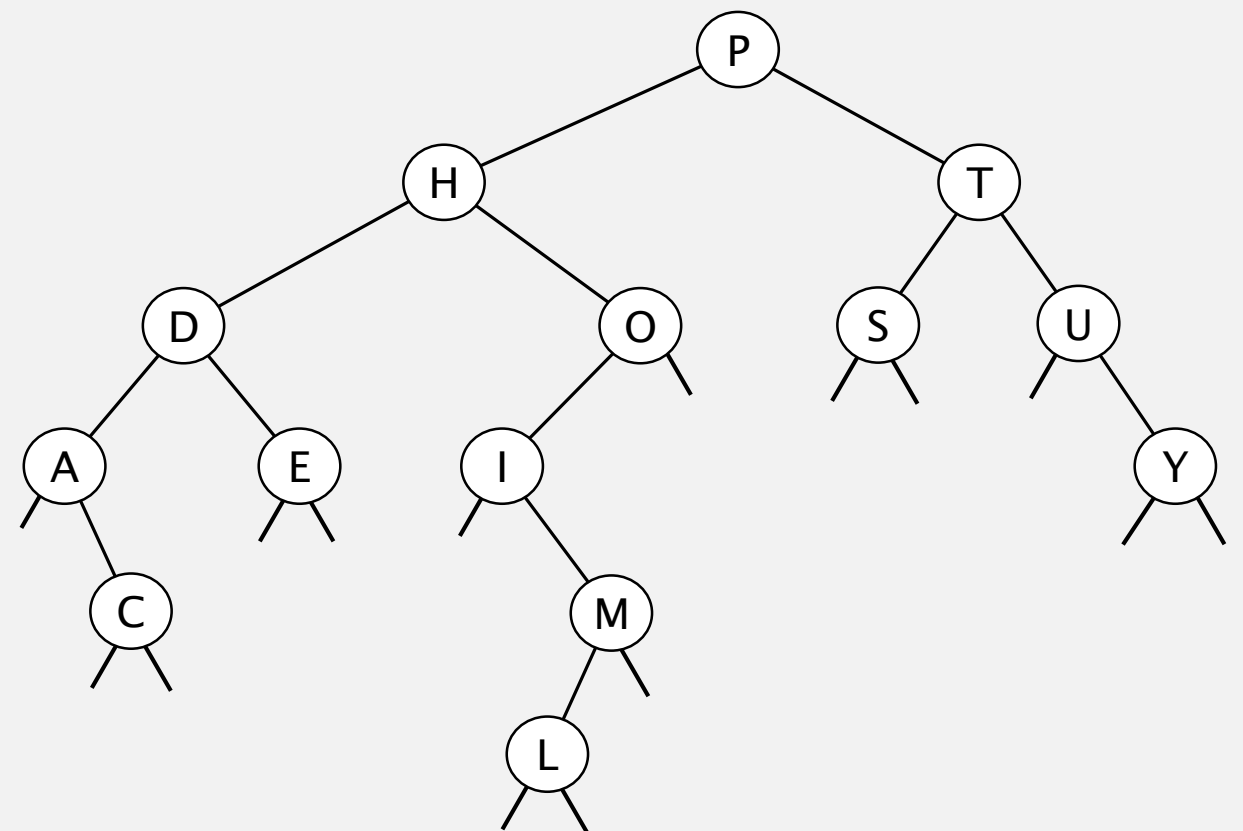
Elements with keys smaller than that in the node is in the left sub-tree while keys larger are in the right sub-tree

IDEA: A recursive sorting algorithm based on partitioning the elements in array so those smaller than the partitioning element falls to the left and those larger falls to the right

This is Quicksort

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
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.

Quicksort

Basic method.

- **Partition** so that for some value j
 - The element $a[j]$ is correctly placed (on its final place)
 - No element which is greater than $a[j]$ is to the left of j
 - No element which is less than $a[j]$ is to the right of j
- **Sort** all sub-arrays recursively.

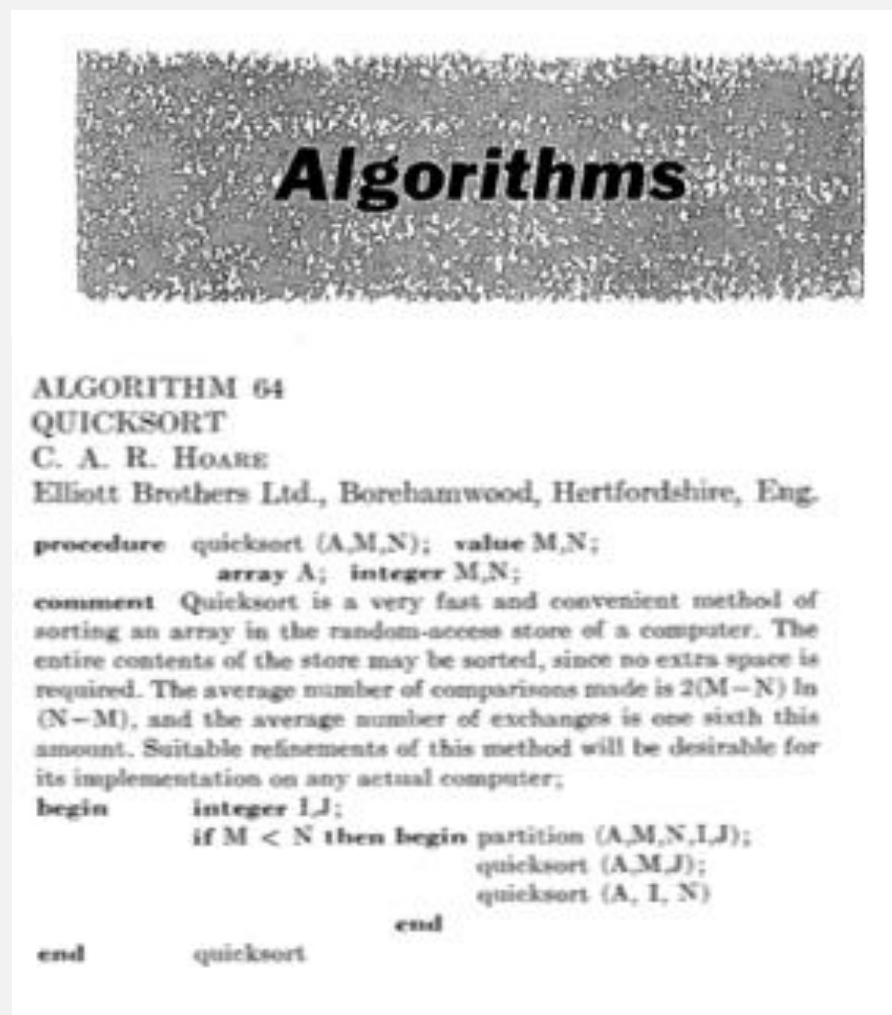


input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

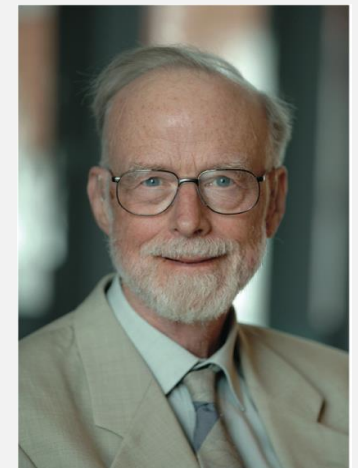
Quicksort overview

Tony Hoare

- Hoare invented quicksort to translate from russian to english.
- Originally implemented in Algol 60 1960 (with recursion).



Communications of the ACM (July 1961)



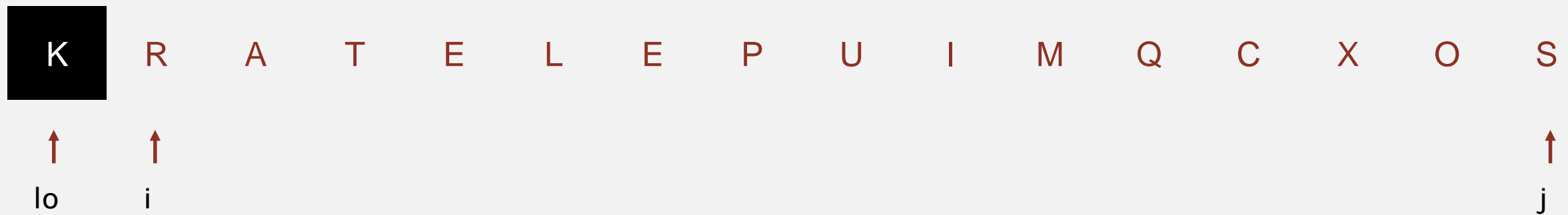
Tony Hoare
1980 Turing Award

“ There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. ”

Quicksort partitioning

Repeat until the indices i and j cross.

- Scan by i from left to right while $(a[i] < a[lo])$.
- Scan by j from right to left while $(a[j] > a[lo])$.
- Swap $a[i]$ and $a[j]$.



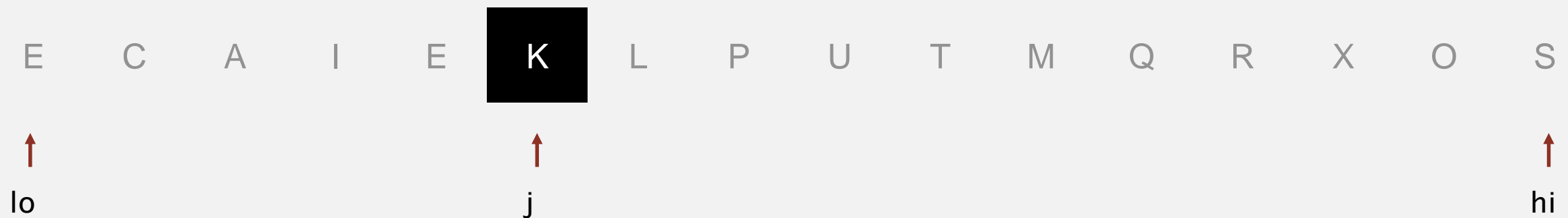
Quicksort partitioning

Repeat until the indices i and j cross.

- Scan by i from left to right while $(a[i] < a[lo])$.
- Scan by j from right to left while $(a[j] > a[lo])$.
- Swap $a[i]$ and $a[j]$.

When indices cross.

- Swap $a[lo]$ and $a[j]$ (i.e. place the partition element $a[lo]$ at its correct (final) place)



partitioned!

Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a,
int lo, int hi)
{
    int i = lo, j = hi+1;
    while (true)
    {
        while (less(a[++i], a[lo]))
            if (i == hi) break;

        while (less(a[lo], a[--j]))
            if (j == lo) break;

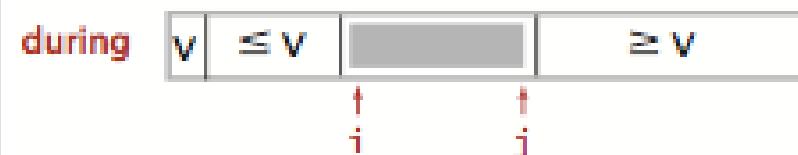
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

Find left hand element to swap

Find right hand element to swap

Check if indices i and j cross
swap

Swap the partitioning element to final
Return index of partitioning element (in
place
its correct place)



```
exch(a, lo, j);
```

```
return j;
```


Quicksort partitioning trace

	i	j	a[]															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	<u>R</u>	A	T	E	L	E	P	U	I	M	Q	<u>C</u>	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	<u>A</u>	<u>T</u>	E	L	E	P	U	<u>I</u>	<u>M</u>	<u>Q</u>	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	<u>E</u>	<u>L</u>	<u>E</u>	<u>P</u>	<u>U</u>	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	<u>E</u>	<u>L</u>	P	U	T	M	Q	R	X	O	S
final exchange	6	5	<u>E</u>	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

Quicksort: Java implementation

```
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    { /* see previous slide */ }

    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
```

Problem – partitioning element

For Quicksort to be effective we would like to select the partitioning element so that we halve the array

Select the partitioning element as the median of the first, last and middle element in the array to partition

(not perfect – but good enough and effective to implement)

Quicksort animation

<https://en.wikipedia.org/wiki/Quicksort>

Quicksort: implementation details

Partitioning in-place. One can use an auxiliary array to facilitate partitioning and make it stable – but the cost is too high.

To stop looping. The check to identify when the indices crosses are harder to implement correctly than what it may seem.

Duplicate keys. To avoid quadratic execution time stop scanning when elements equal to the partitioning element is found.

Preserve randomness. The initial ordering of the array elements need to be random for the average case performance guarantees to hold. Randomness of the left and right hand sub-arrays are preserved when swaping elements.

Equivalent alternative to shuffling. Randomly select the partitioning element in all (sub-) arrays.

Quicksort: empirical analysis (1961)

Execution time measurements and estimates:

- Algol 60 implementation.
- National-Elliott 405 computer.

Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

sorting N 6-word items with 1-word keys



**Elliott 405 magnetic disc
(16K words)**

Quicksort: empiric analysis

Estimated execution times:

- Home PC executes 10^8 comparisons/s.
- Super computer executes 10^{12} comparisons/s.

computer	insertion sort (N^2)			mergesort ($N \log N$)			quicksort ($N \log N$)		
	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than super computers.

Lesson 2. Excellent algorithms are better than good algorithms.

Quicksort: best-case analysis

Best case. The number of comparisons is $\sim N \lg N$.

			a[]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
input values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
Shuffle (already shuffled)			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: worst-case analysis

Worst case. The number of comparisons $\sim \frac{1}{2} N^2$.

			a[]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: summary of expected performance

Quicksort is a **randomized divide-and-conquer algorithm**.

- It is proven to be correct
- Execution time depends on the shuffle (randomness of the elements)

Average case. Expected number of comparisons are $\sim 1.39 N \lg N$.

- 39% higher than mergesort.
- Faster than mergesort as less elements are copied (swapped).

Best case. Number of comparisons is $\sim N \lg N$.

Worst case. Number of comparisons is $\sim \frac{1}{2} N^2$.



Quicksort properties

Proposition. Quicksort is an **in-place** sorting algorithm.

Proof.

- Partitioning: constant auxiliary space.
- Depth of recursion: logarithmic stack space (with high probability).

Proposition. Quicksort is **not stable**.

Proof by counter exemple.

i	j	0	1	2	3
		B ₁	C ₁	C ₂	A ₁
1	3	B ₁	C ₁	C ₂	A ₁
1	3	B ₁	A ₁	C ₂	C ₁
0	1	A ₁	B ₁	C ₂	C ₁

Quicksort: improvements

Insertion sort for short sub arrays.

- Recursive algorithms in general has too much overhead when sorting short arrays.
- Cutoff for insertion sort $H \approx 10$ elements.

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

- Why is it less efficient to run one insertion sort at the very end , sorting the partially sorted array?

Quicksort: improvements

Partitioning element: Median-of-three.

- Best partitioning element = median.
- Approximate the median by the median of a sample of keys.
- Median-of-3 random) elements (keys).

~ $12/7$ $N \ln N$ comparisons (14% reduction)

~ $12/35$ $N \ln N$ swaps (3% more)

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;

    int median = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, median);

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

Duplicate keys

Often we sort data with many similar or duplicate keys.

- Sort people by age.
- Remove duplicates from an E-mail list.
- Sort applicants by grades.

Common properties of such applications.

- Vast amount of elements (records).
- Relatively few keys.

```
Chicago 09:25:!!  
Chicago 09:03:!!  
Chicago 09:21:!!  
Chicago 09:19:!!  
Chicago 09:19:!!  
Chicago 09:00:!!  
Chicago 09:35:!!  
Chicago 09:00:!!  
Houston 09:01:!!  
Houston 09:00:!!  
Phoenix 09:37:!!  
Phoenix 09:00:!!  
Phoenix 09:14:!!  
Seattle 09:10:!!  
Seattle 09:36:!!  
Seattle 09:22:!!  
Seattle 09:10:!!  
Seattle 09:22:!!
```

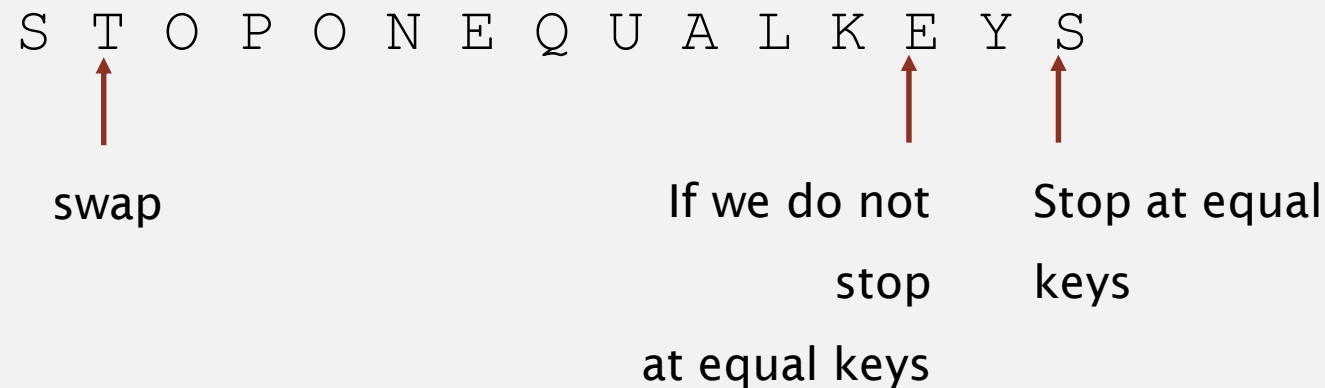
nyck

el

Quicksort and duplicate keys

Elements with equal keys (duplicate keys)

Quicksort and duplicate keys. Algorithm may be quadratic if one does not stop the partitioning when equal keys are compared!



Some implementations (both in books and in real life) is quadratic if many elements have equal keys.

(is mergesort sensitive to such cases?)

Partitioning element when equal keys

Where does the partitioning element end up when sorting equal keys?

A A A A A A A A A A A A A A A A

A A A A A A A A A A A A A A A A

A.

A A A A A A A A A A A A A A A

B.

A A A A A A A A A A A A A A A

C.

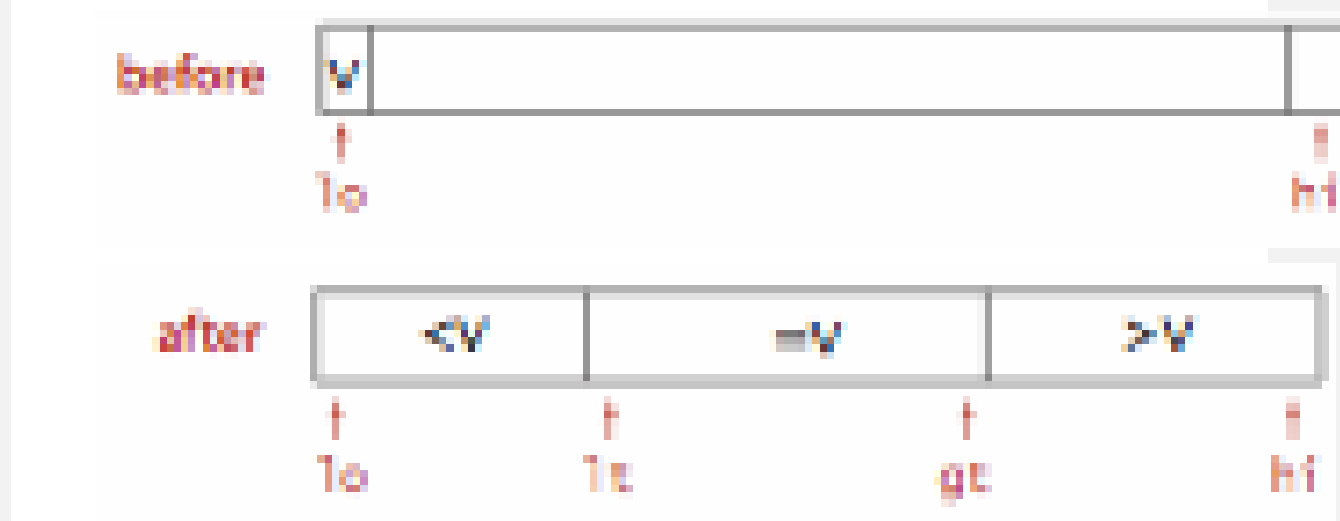
To partition an array with equal keys

		a[]															
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

3-way partitioning

Target. Partition the array in **three** partitions so that:

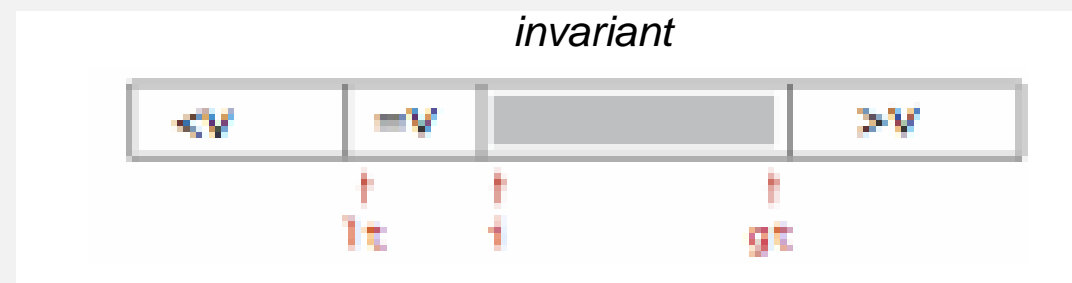
- Elements between lt and gt have keys equal to that of the partitioning element.
- No elements with greater keys are found to the left of lt .
- No elements with smaller keys are found to the right of gt .



- (Hard to implement efficiently in-place)

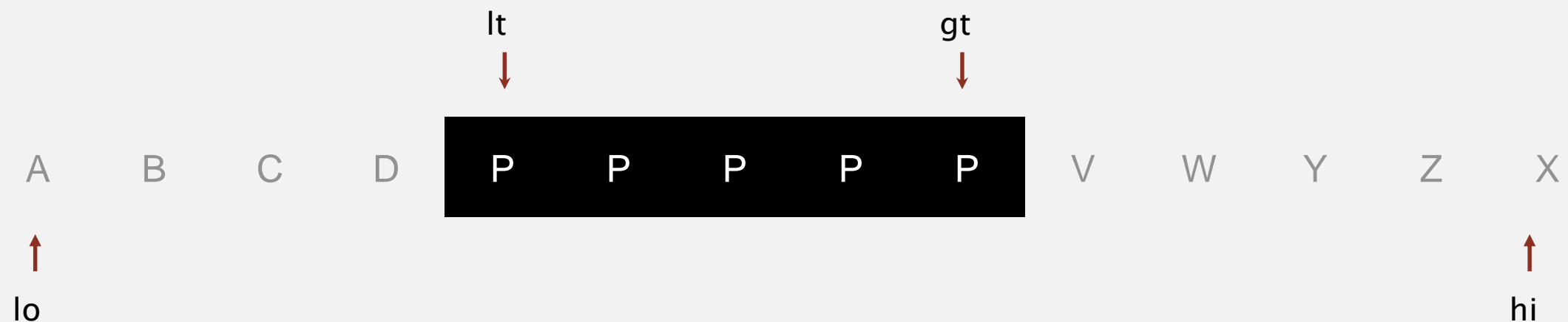
Dijkstra 3-way partitioning

- Let v be the partitioning element $a[lo]$.
- Scan by i from left to right.
 - $(a[i] < v)$: swap $a[lt]$ with $a[i]$; increment both lt and i
 - $(a[i] > v)$: swap $a[gt]$ with $a[i]$; decrement gt
 - $(a[i] == v)$: increment i

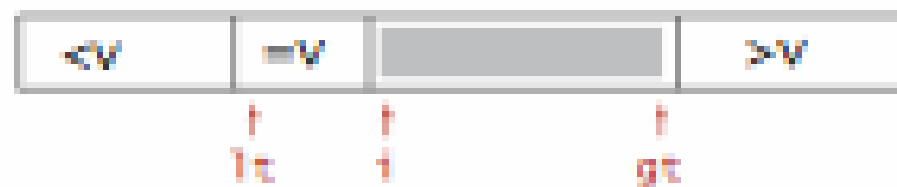


Dijkstra 3-way partitioning demo

- Let v be the partitioning element $a[lo]$.
- Scan by i from left to right.
 - $(a[i] < v)$: swap $a[lt]$ with $a[i]$; increment both lt and i
 - $(a[i] > v)$: swap $a[gt]$ with $a[i]$; decrement gt
 - $(a[i] == v)$: increment i



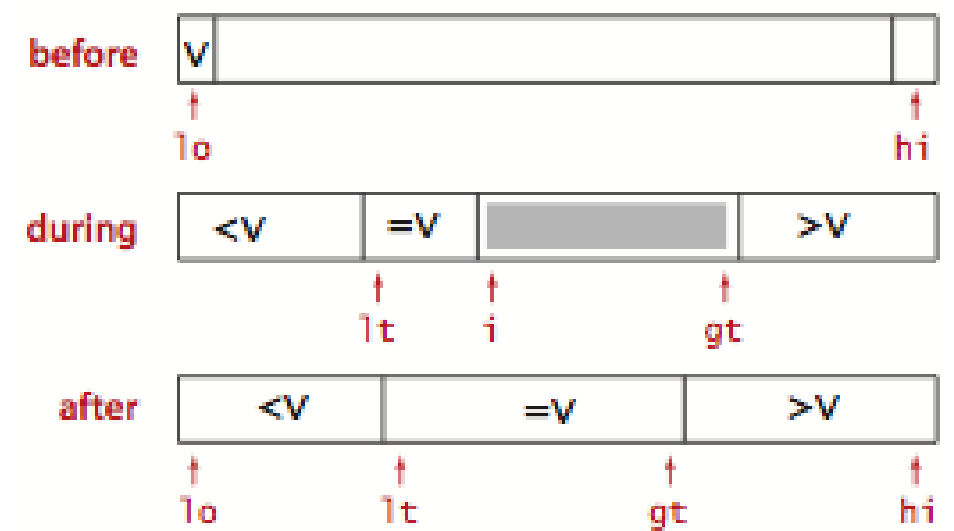
invariant



3-way quicksort: Java implementation

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt)
    {
        int cmp = a[i].compareTo(v);
        if (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else i++;
    }

    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
```

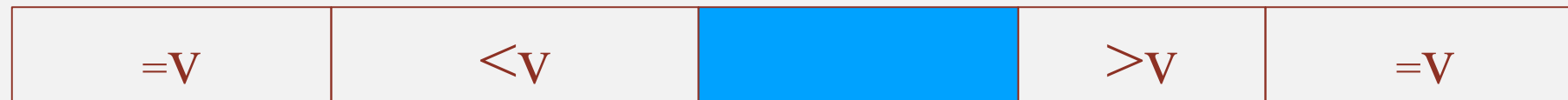


3-way partitioning overview

3-way partitioning improvement

The Dijkstra's 3-way partitioning is inefficient when there are few duplicate keys

J. Bentley and D. McIlroy proposed an entropy optimal 3-way partitioning (see problem 2.3.22)



Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✓	✓	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✓		$N \log_3 N$?	$c N^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
quicksort	✓		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-väggs quicksort	✓		N	$2 N \ln N$	$\frac{1}{2} N^2$	improves quicksort _[SEP] when duplicate keys
?	✓	✓	N	$N \lg N$	$N \lg N$	holy sorting grail

Which sorting algorithm to use?

There are many different sorting algorithms to choose from:

sorts	algorithms
elementary sorts	insertion sort, selection sort, bubblesort, shaker sort, ...
subquadratic sorts	quicksort, mergesort, heapsort, shellsort, samplesort, ...
system sorts	dual-pivot quicksort, timsort, introsort, ...
external sorts	Poly-phase mergesort, cascade-merge, psort,
radix sorts	MSD, LSD, 3-way radix quicksort, ...
parallel sorts	bitonic sort, odd-even sort, smooth sort, GPU sort, ...

Which sorting algorithm to use?

Applications have different requirements:

- Stable?
- Parallel?
- In-place?
- Deterministic?
- Duplicate keys?
- Different types of keys?
- Linked list or array?
- Large or small elements? (copy/swap)
- Order of the input?
- Performance guarantees?

Is "system sort" fast enough for common cases?

YES.

System sort in Java 7

`Arrays.sort()`.

- Has a method for objects that implement `Comparable`.
- Has an overloaded method for all primitive types.
- Has an overloaded method for objects implementing `Comparator`.
- Has overloaded methods to sort (short) sub-arrays.



Algorithms.

- Dual-pivot quicksort for primitive types.
- Timsort (mergesort clone) for reference types.

Why use different algorithms depending on the objects sorted are primitive or reference types?