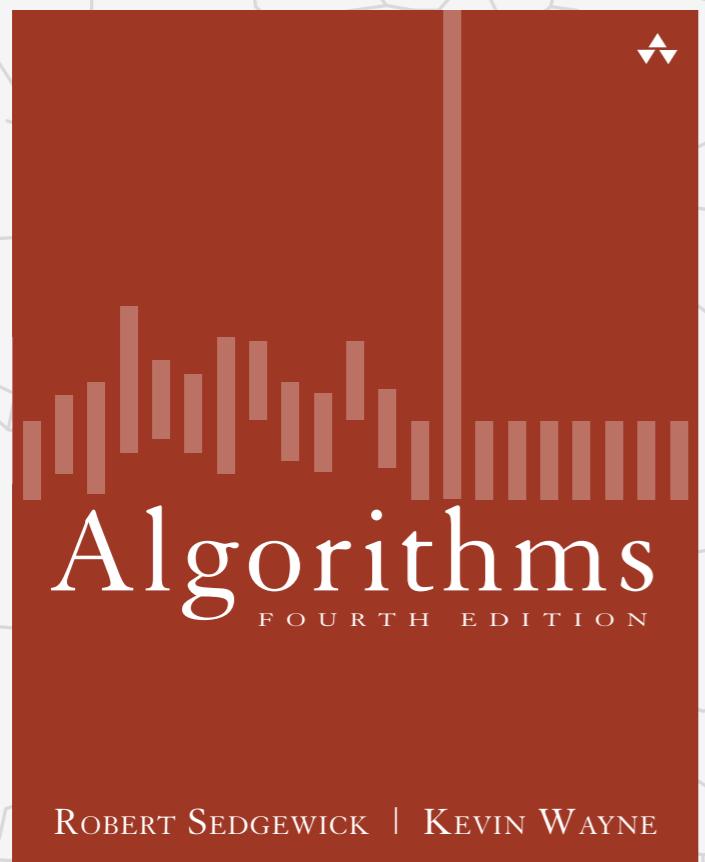


Algorithms

FARAAZ SARESHWALA



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

2.4 PRIORITY QUEUES

BASED ON SLIDES FROM ROBERT SEDGEWICK AND KEVIN WAYNE

Priority queue

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

Priority queue

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

Priority queue. Remove the **largest** (or **smallest**) item.

<i>operation</i>	<i>argument</i>	<i>return value</i>
<i>insert</i>	P	
<i>insert</i>	Q	
<i>insert</i>	E	
<i>remove max</i>		Q
<i>insert</i>	X	
<i>insert</i>	A	
<i>insert</i>	M	
<i>remove max</i>		X
<i>insert</i>	P	
<i>insert</i>	L	
<i>insert</i>	E	
<i>remove max</i>		P

Priority queue API

Requirement. Generic items are Comparable.

public class MaxPQ<Key extends Comparable<Key>>		Key must be Comparable (bounded type parameter)
MaxPQ()		<i>create an empty priority queue</i>
MaxPQ(Key[] a)		<i>create a priority queue with given keys</i>
void insert(Key v)		<i>insert a key into the priority queue</i>
Key delMax()		<i>return and remove the largest key</i>
boolean isEmpty()		<i>is the priority queue empty?</i>
Key max()		<i>return the largest key</i>
int size()		<i>number of entries in the priority queue</i>

Priority queue applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [online median in data stream]
- Operating systems. [load balancing, interrupt handling]
- Computer networks. [web cache]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue client example

Challenge. Find the largest M items in a stream of N items.

- Fraud detection: isolate \$\$ transactions.
- NSA monitoring: flag most suspicious documents.

```
% more tinyBatch.txt
Turing 6/17/1990 644.08
vonNeumann 3/26/2002 4121.85
Dijkstra 8/22/2007 2678.40
vonNeumann 1/11/1999 4409.74
Dijkstra 11/18/1995 837.42
Hoare 5/10/1993 3229.27
vonNeumann 2/12/1994 4732.35
Hoare 8/18/1992 4381.21
Turing 1/11/2002 66.10
Thompson 2/27/2000 4747.08
Turing 2/11/1991 2156.86
Hoare 8/12/2003 1025.70
vonNeumann 10/13/1993 2520.97
Dijkstra 9/10/2000 708.95
```

```
% java TopM 5 < tinyBatch.txt
Thompson 2/27/2000 4747.08
vonNeumann 2/12/1994 4732.35
vonNeumann 1/11/1999 4409.74
Hoare 8/18/1992 4381.21
vonNeumann 3/26/2002 4121.85
```

sort key ↑

Priority queue client example

Challenge. Find the largest M items in a stream of N items.

- Fraud detection: isolate \$\$ transactions.
- NSA monitoring: flag most suspicious documents.

N huge, M large

Constraint. Not enough memory to store N items.

```
% more tinyBatch.txt
Turing 6/17/1990 644.08
vonNeumann 3/26/2002 4121.85
Dijkstra 8/22/2007 2678.40
vonNeumann 1/11/1999 4409.74
Dijkstra 11/18/1995 837.42
Hoare 5/10/1993 3229.27
vonNeumann 2/12/1994 4732.35
Hoare 8/18/1992 4381.21
Turing 1/11/2002 66.10
Thompson 2/27/2000 4747.08
Turing 2/11/1991 2156.86
Hoare 8/12/2003 1025.70
vonNeumann 10/13/1993 2520.97
Dijkstra 9/10/2000 708.95
```

```
% java TopM 5 < tinyBatch.txt
Thompson 2/27/2000 4747.08
vonNeumann 2/12/1994 4732.35
vonNeumann 1/11/1999 4409.74
Hoare 8/18/1992 4381.21
vonNeumann 3/26/2002 4121.85
```

sort key

Priority queue client example

Challenge. Find the largest M items in a stream of N items.

- Fraud detection: isolate \$\$ transactions.
- NSA monitoring: flag most suspicious documents.

N huge, M large

Constraint. Not enough memory to store N items.

```
use a min-oriented pq
MinPQ<Transaction> pq = new MinPQ<Transaction>();

while (StdIn.hasNextLine())
{
    String line = StdIn.readLine();
    Transaction item = new Transaction(line);
    pq.insert(item); ← pq contains
    if (pq.size() > M)           largest M items
        pq.delMin();
}
```

Priority queue client example

Challenge. Find the largest M items in a stream of N items.

implementation	time	space
sort	$N \log N$	N
elementary PQ	$M N$	M
binary heap	$N \log M$	M
best in theory	N	M

order of growth of finding the largest M in a stream of N items

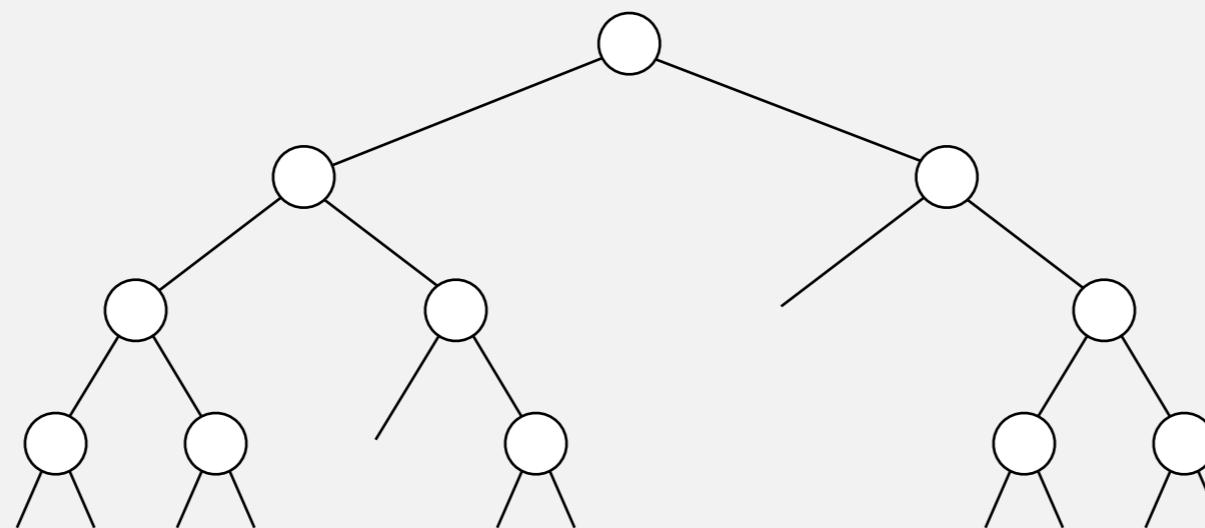
Priority queue elementary implementations

Challenge. Implement **all** operations efficiently.

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	$\log N$	$\log N$	$\log N$

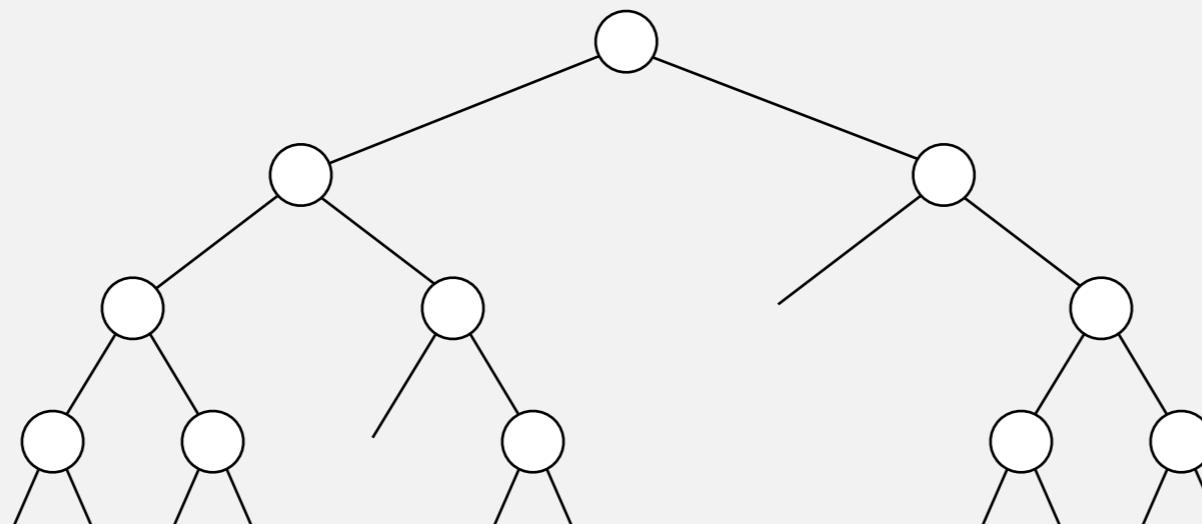
order of growth of running time for priority queue with N items

Complete binary tree



Complete binary tree

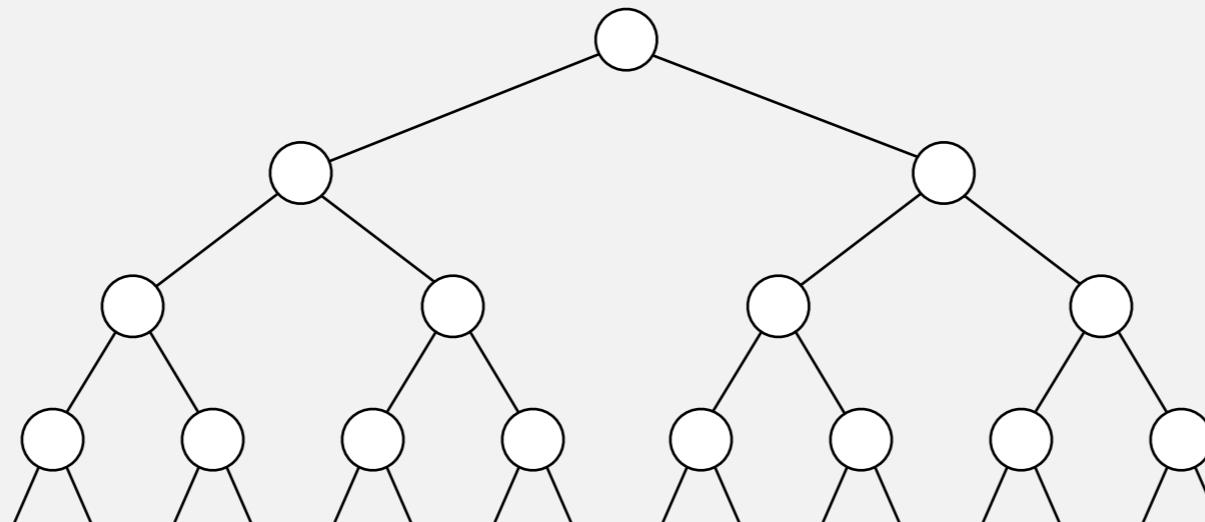
Binary tree. Empty or node with links to left and right binary trees.



Complete binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced

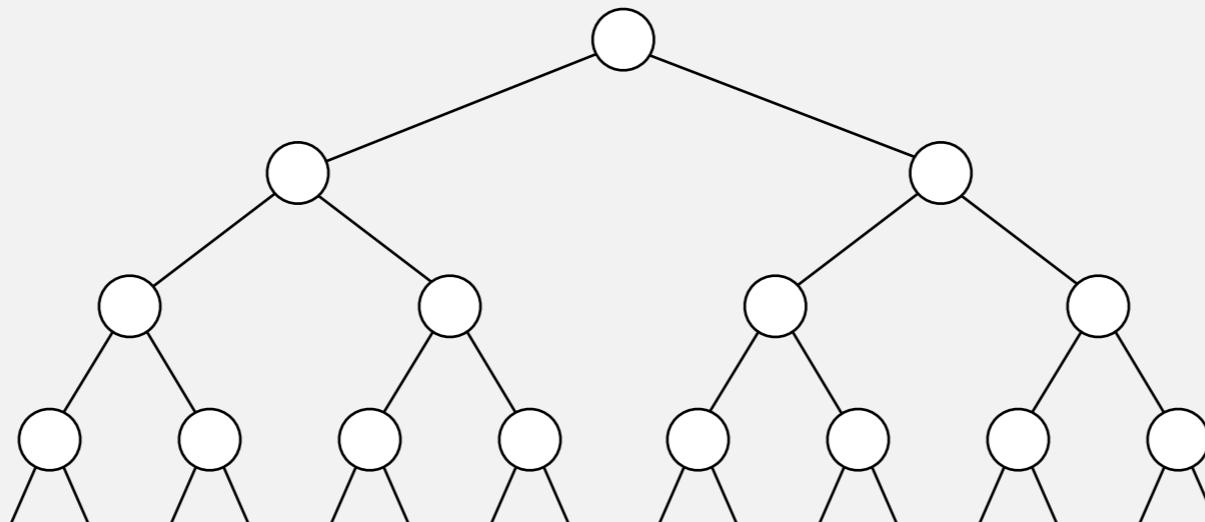


complete tree with $N = 16$ nodes (height = 4)

Complete binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced



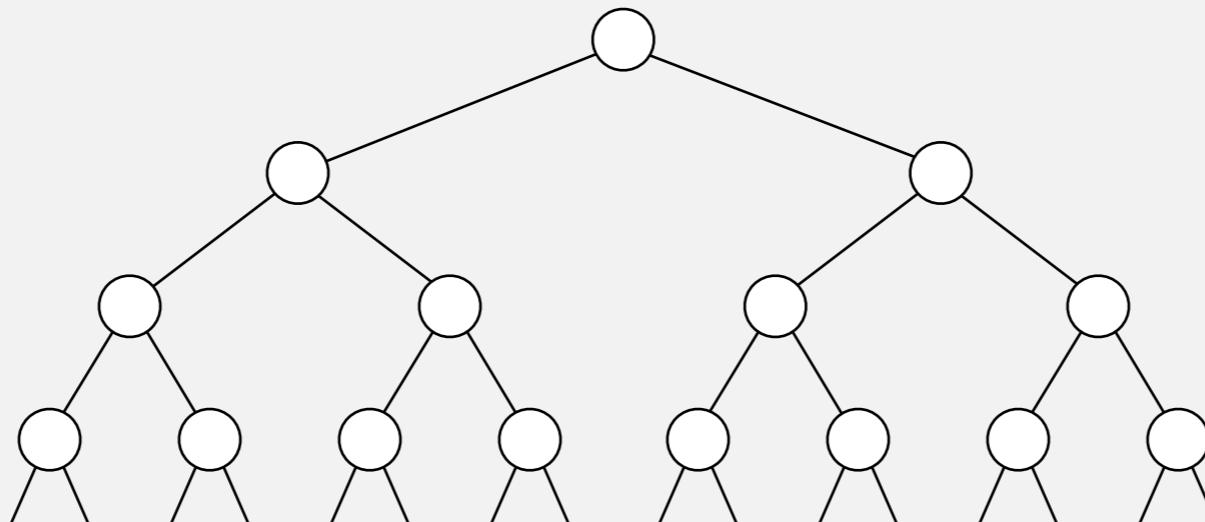
complete tree with $N = 16$ nodes (height = 4)

Property. Height of complete tree with N nodes is $\lfloor \log N \rfloor$.

Complete binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced



complete tree with $N = 16$ nodes (height = 4)

Property. Height of complete tree with N nodes is $\lfloor \log N \rfloor$.

Pf. Height increases only when N is a power of 2.

Binary heap representations

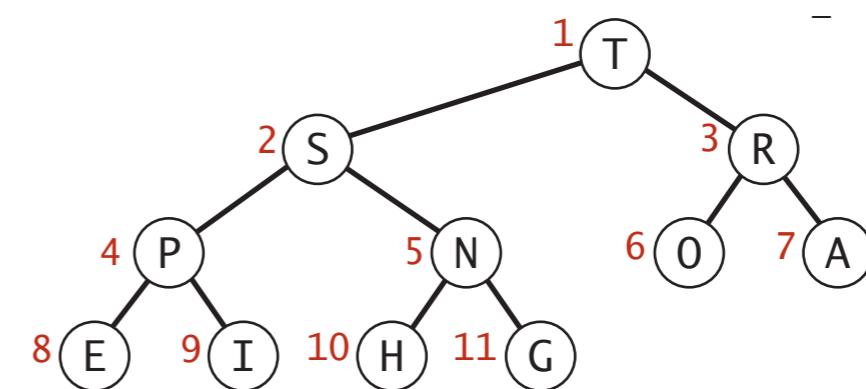
Binary heap. Array representation of a heap-ordered complete binary tree.

Binary heap representations

Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
- Parent's key no smaller than children's keys.



Heap representations

Binary heap representations

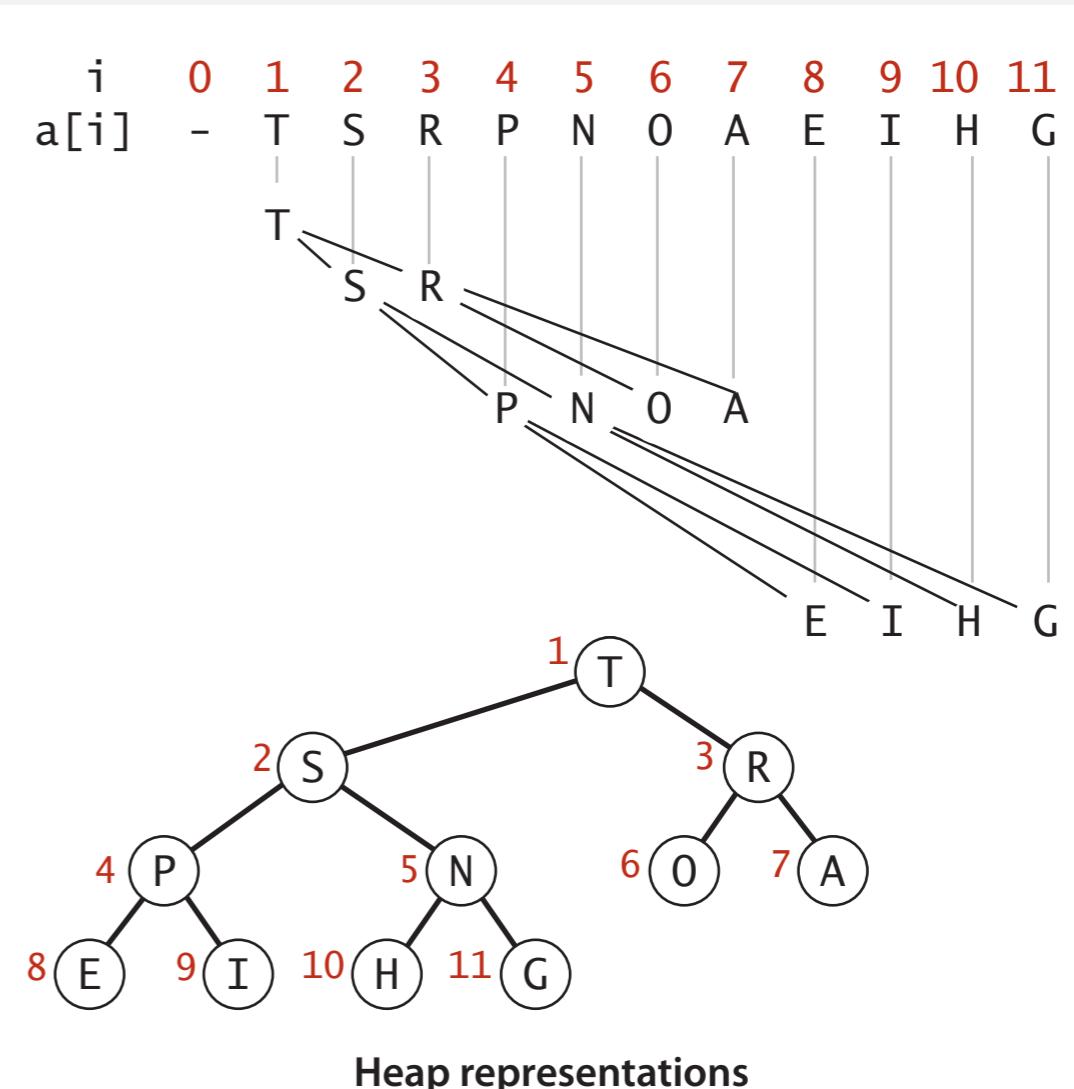
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
 - Parent's key no smaller than children's keys.

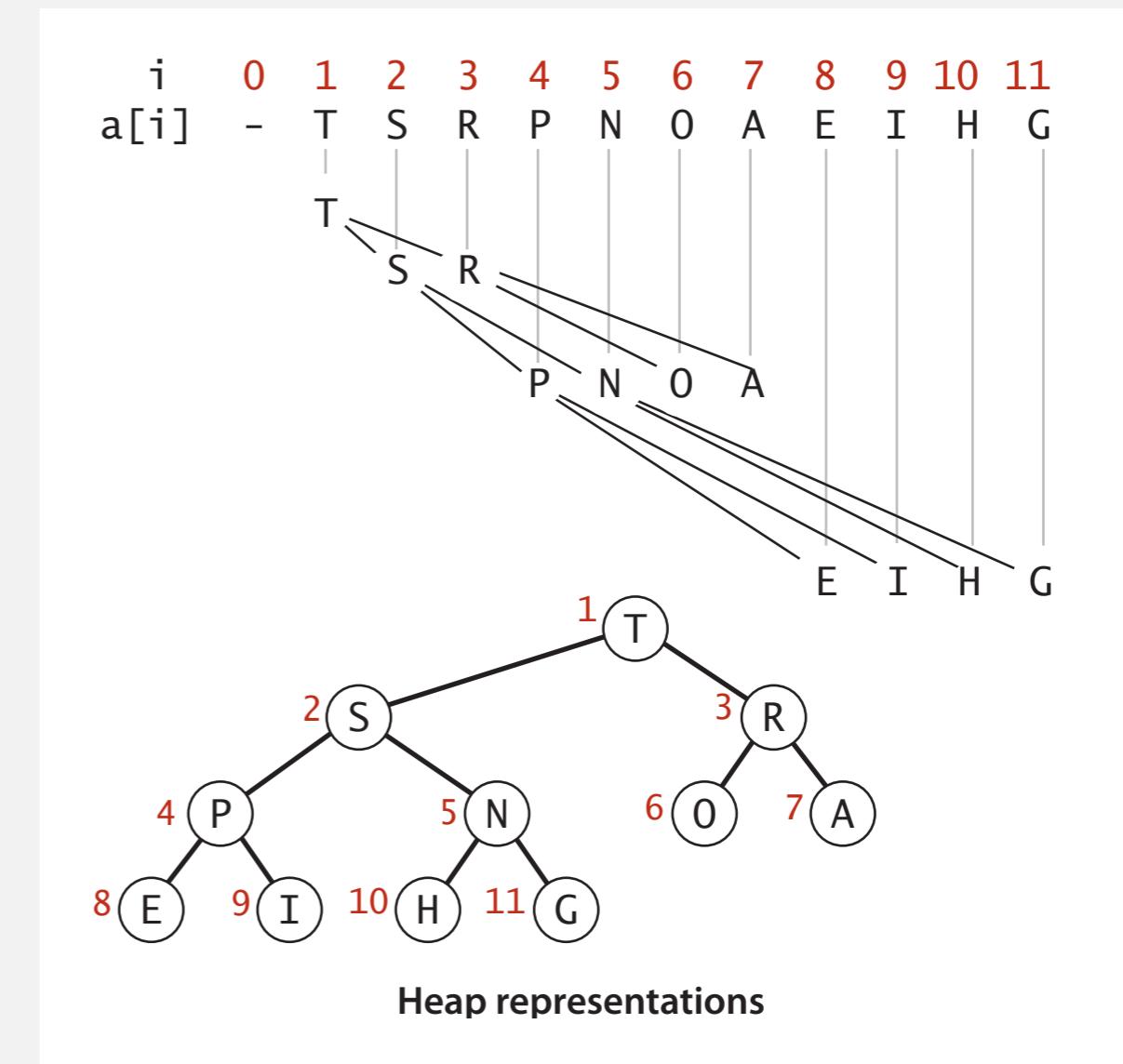
Array representation.

- Indices start at 1.
 - Take nodes in **level** order.
 - No explicit links needed!



Binary heap properties

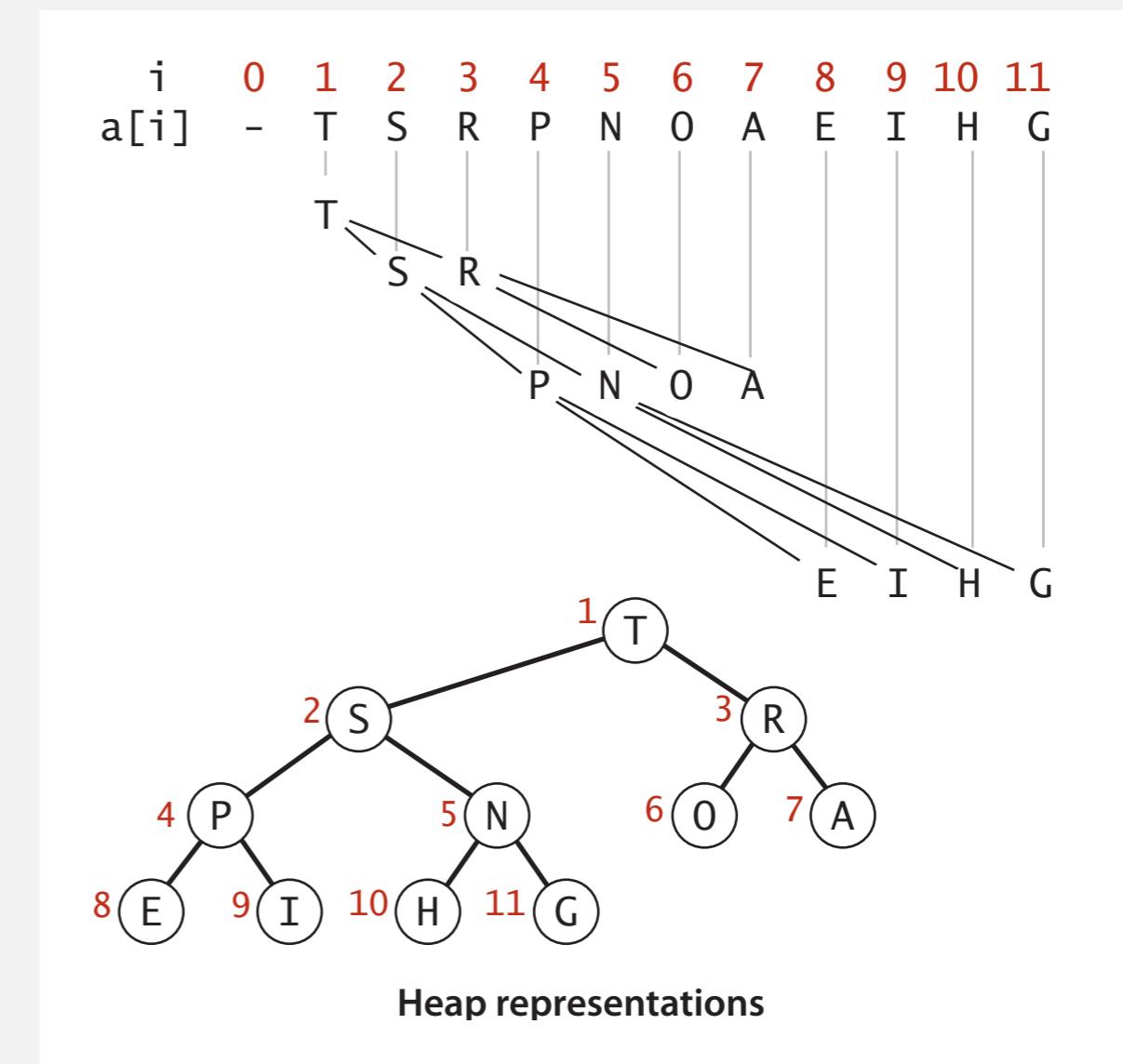
Proposition. Largest key is $a[1]$, which is root of binary tree.



Binary heap properties

Proposition. Largest key is $a[1]$, which is root of binary tree.

Proposition. Can use array indices to move through tree.

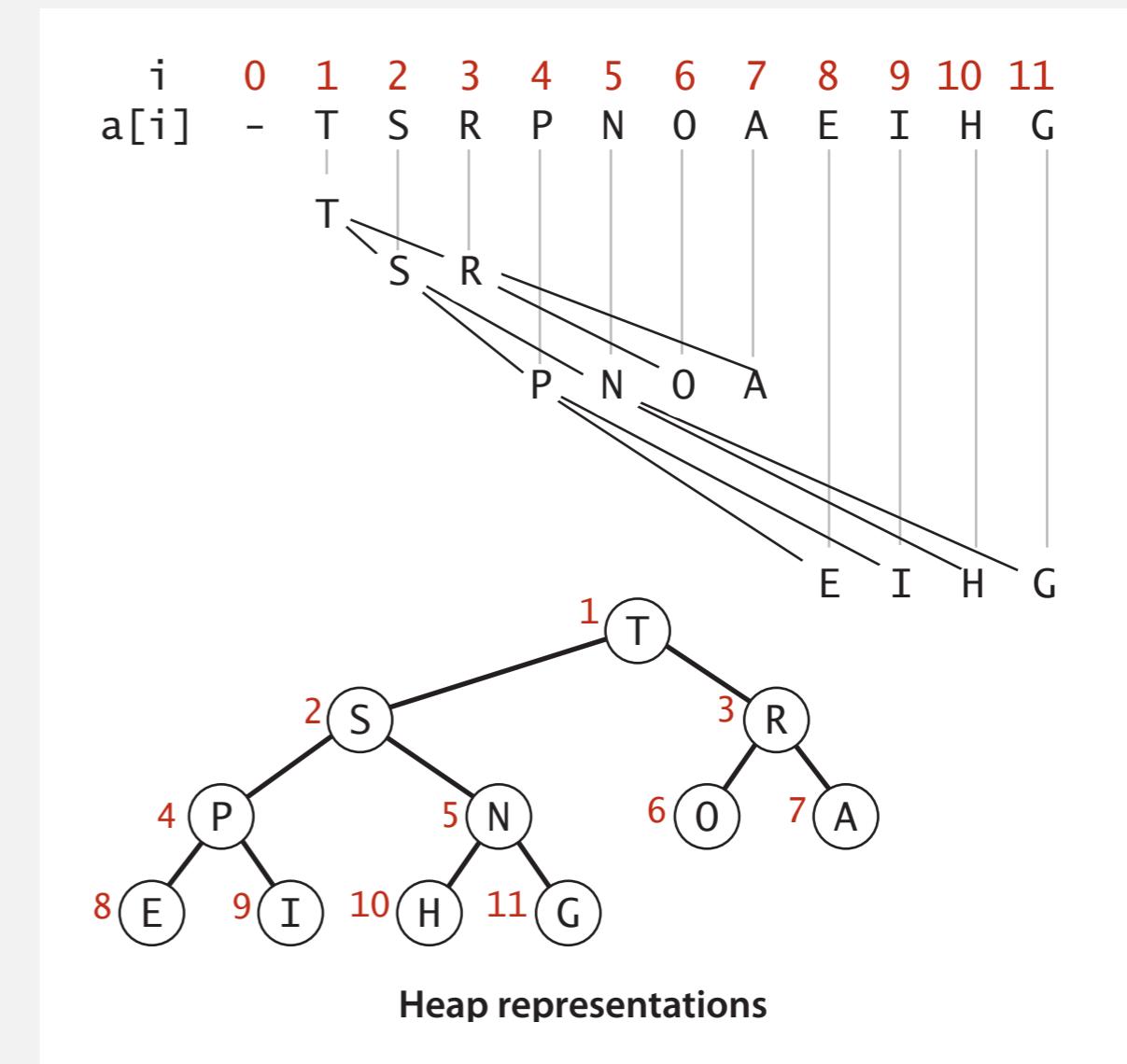


Binary heap properties

Proposition. Largest key is $a[1]$, which is root of binary tree.

Proposition. Can use array indices to move through tree.

- Parent of node at k is at $k/2$.

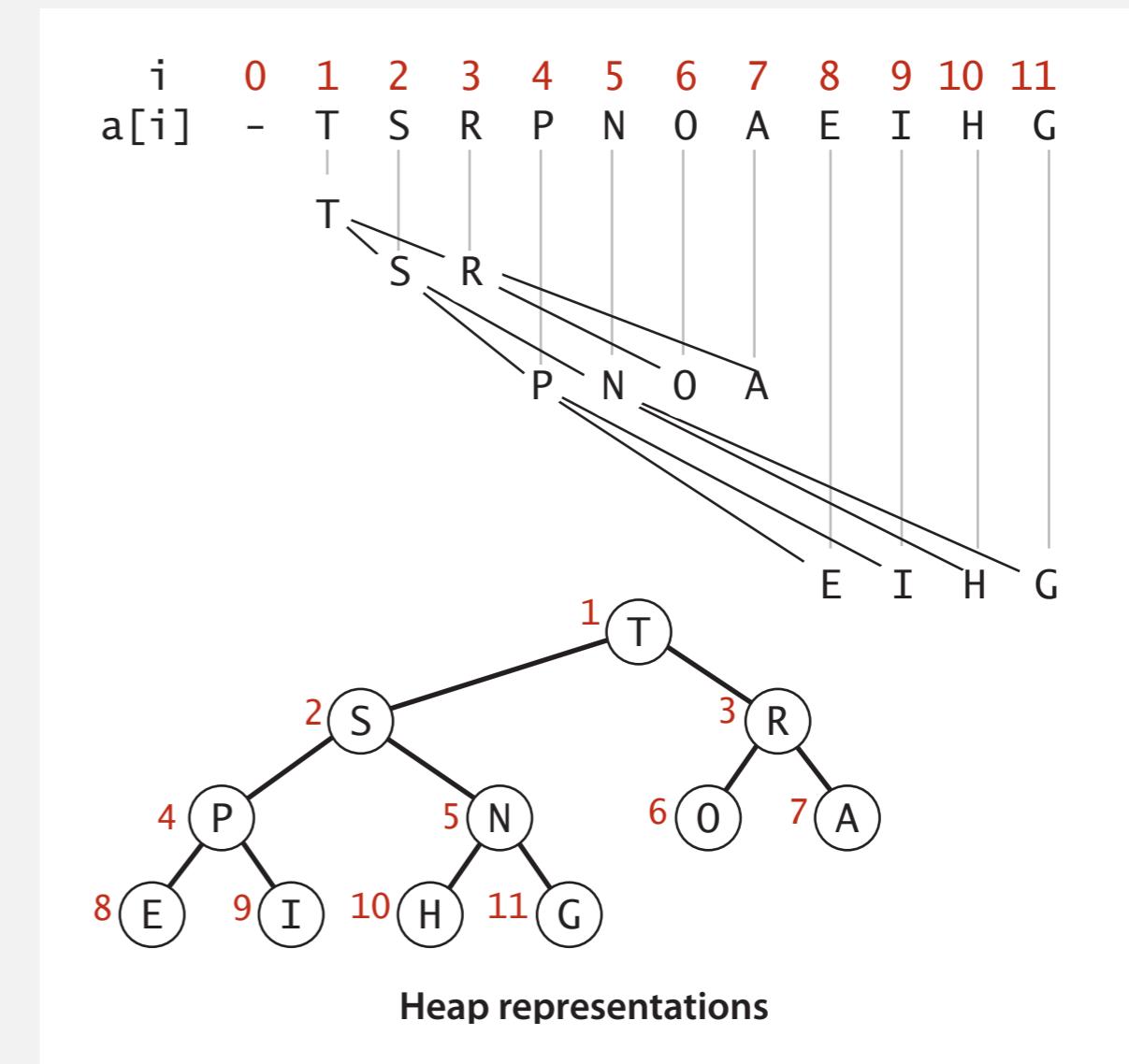


Binary heap properties

Proposition. Largest key is $a[1]$, which is root of binary tree.

Proposition. Can use array indices to move through tree.

- Parent of node at k is at $k/2$.
- Children of node at k are at $2k$ and $2k+1$.

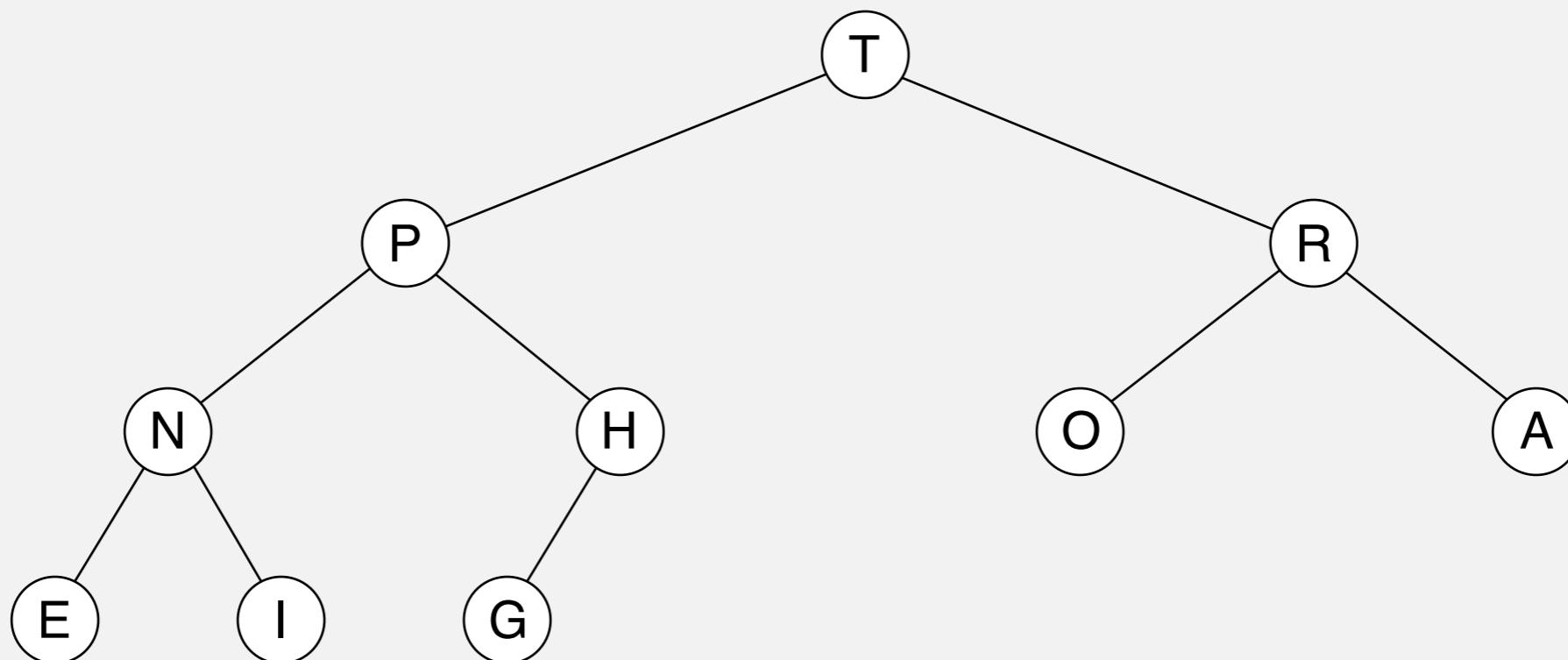


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



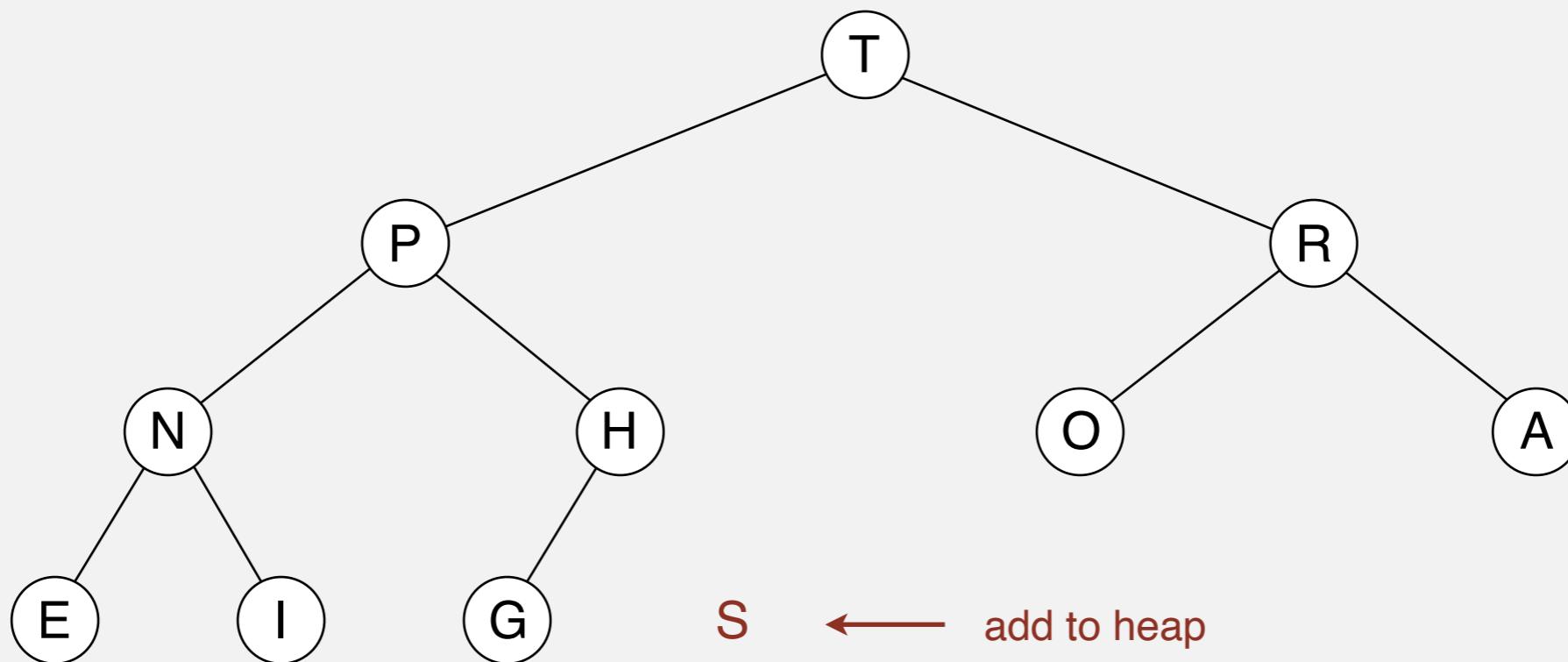
T	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



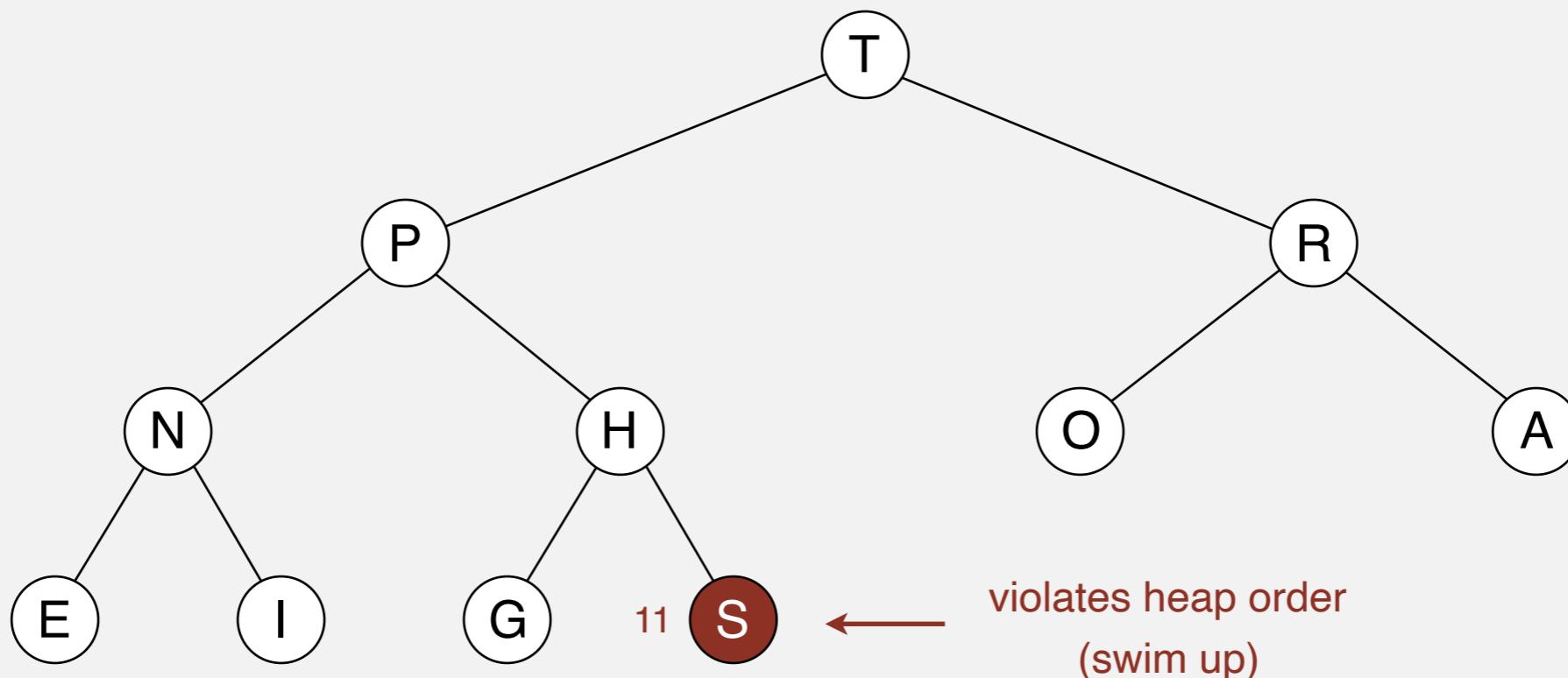
T	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



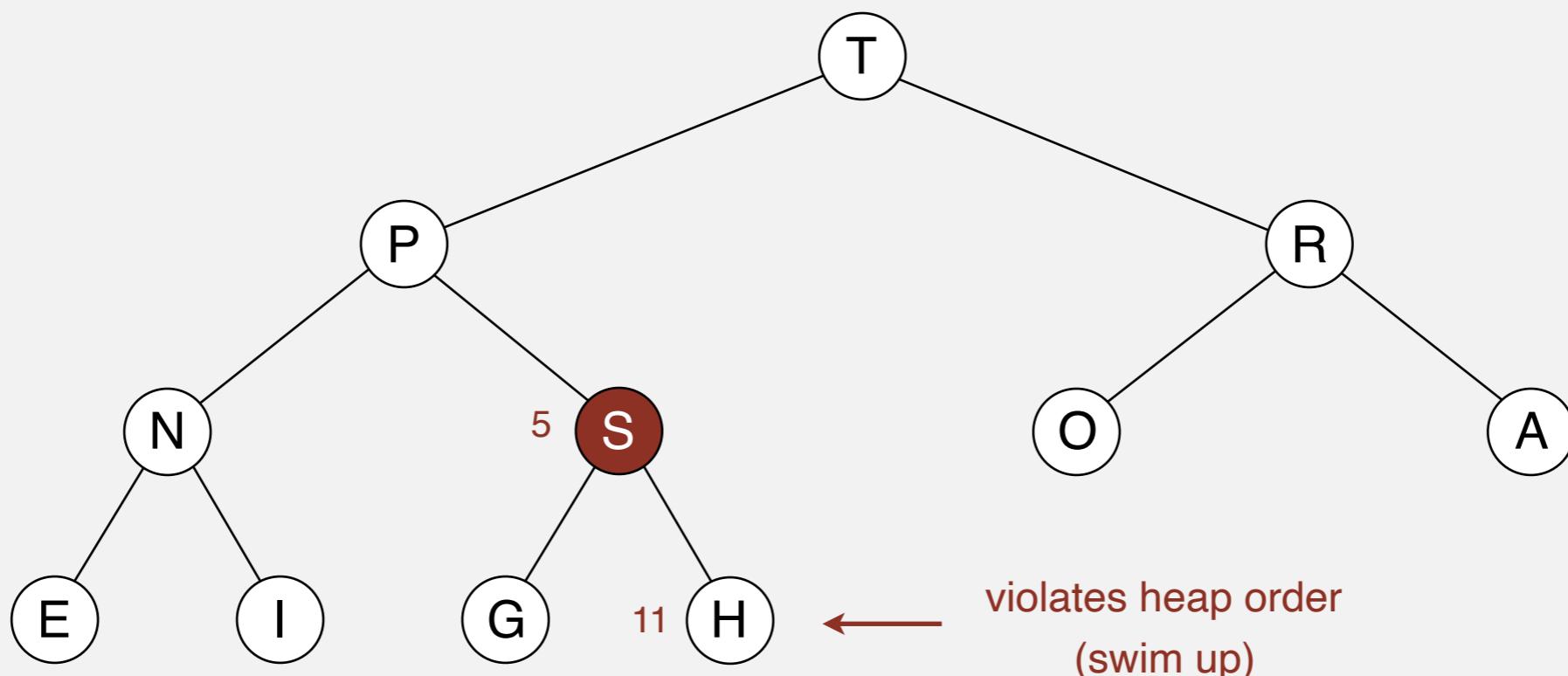
T	P	R	N	H	O	A	E	I	G	S
---	---	---	---	---	---	---	---	---	---	---

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



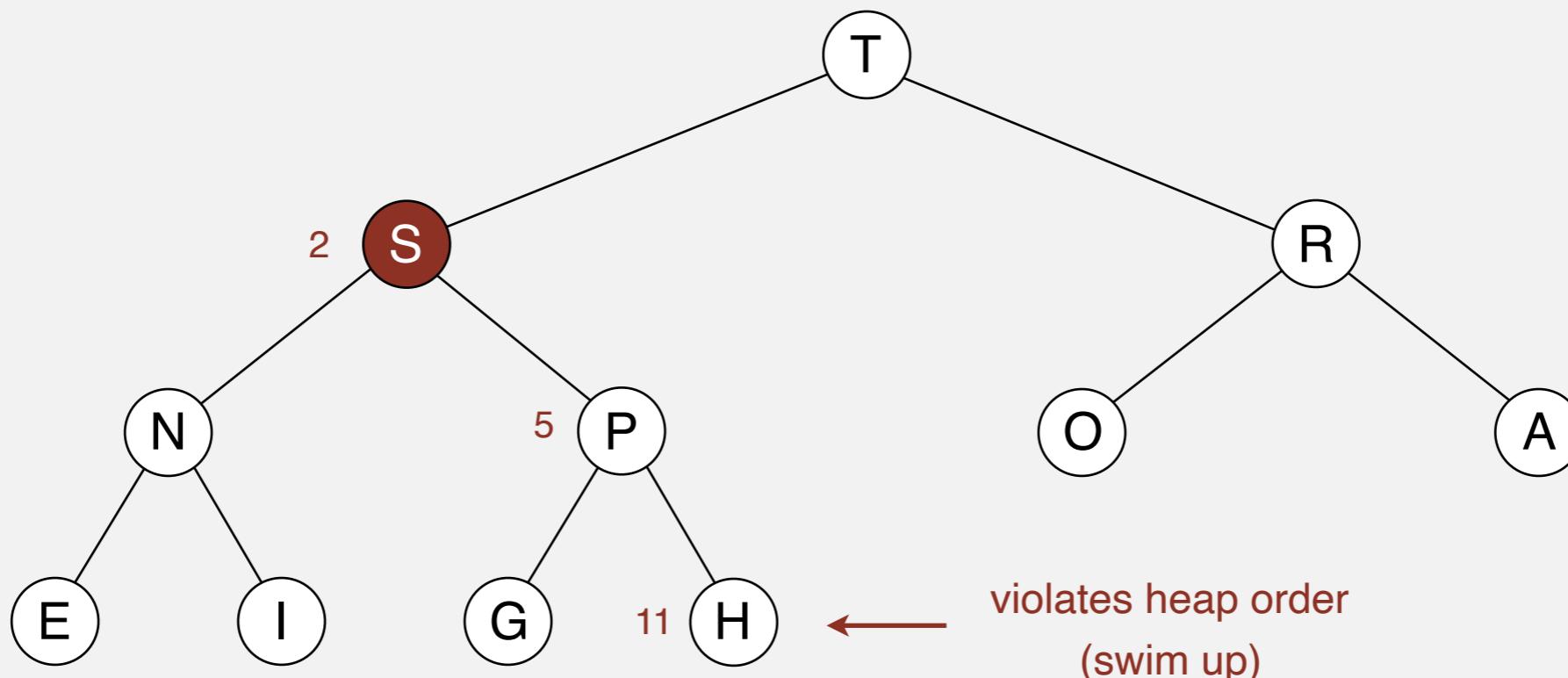
T P R N S O A E I G H
5 11

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



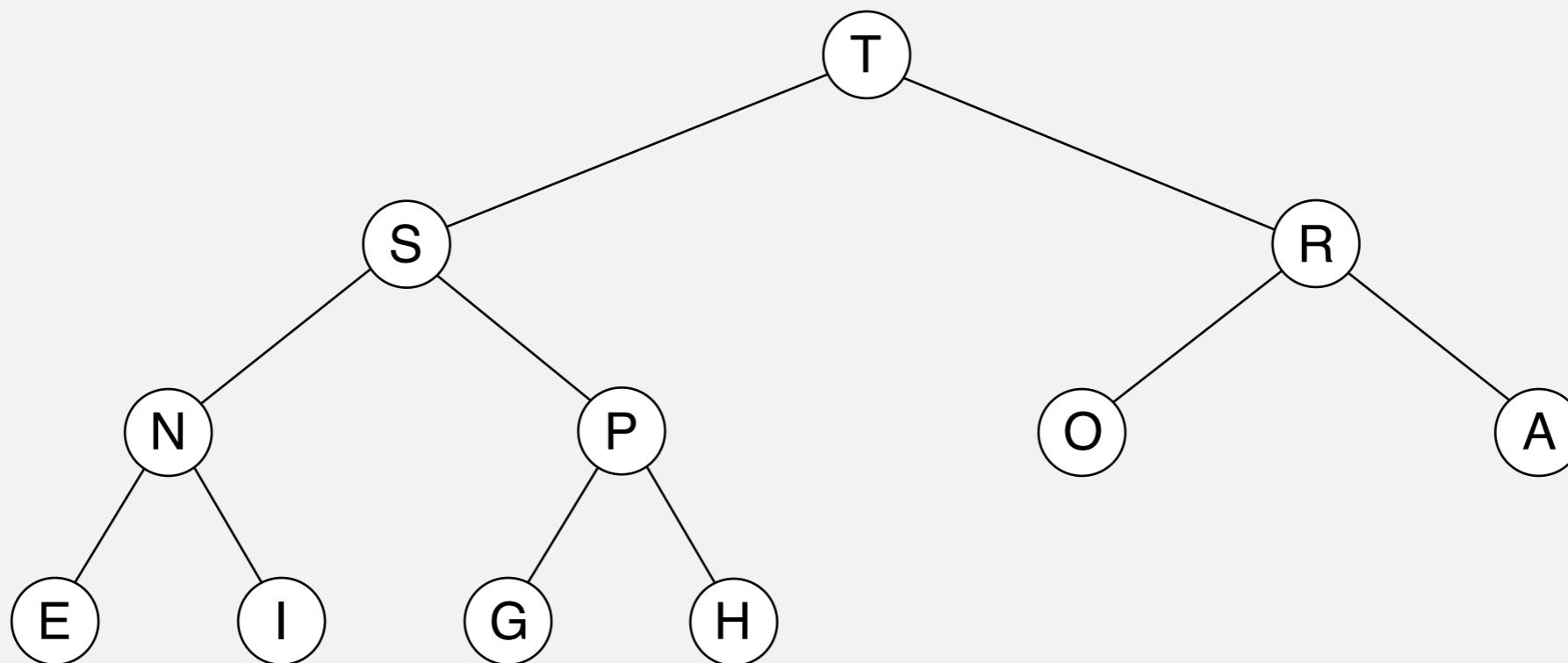
T	S	R	N	P	O	A	E	I	G	H
2	2	5	5	5	5	5	11	11	11	11

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



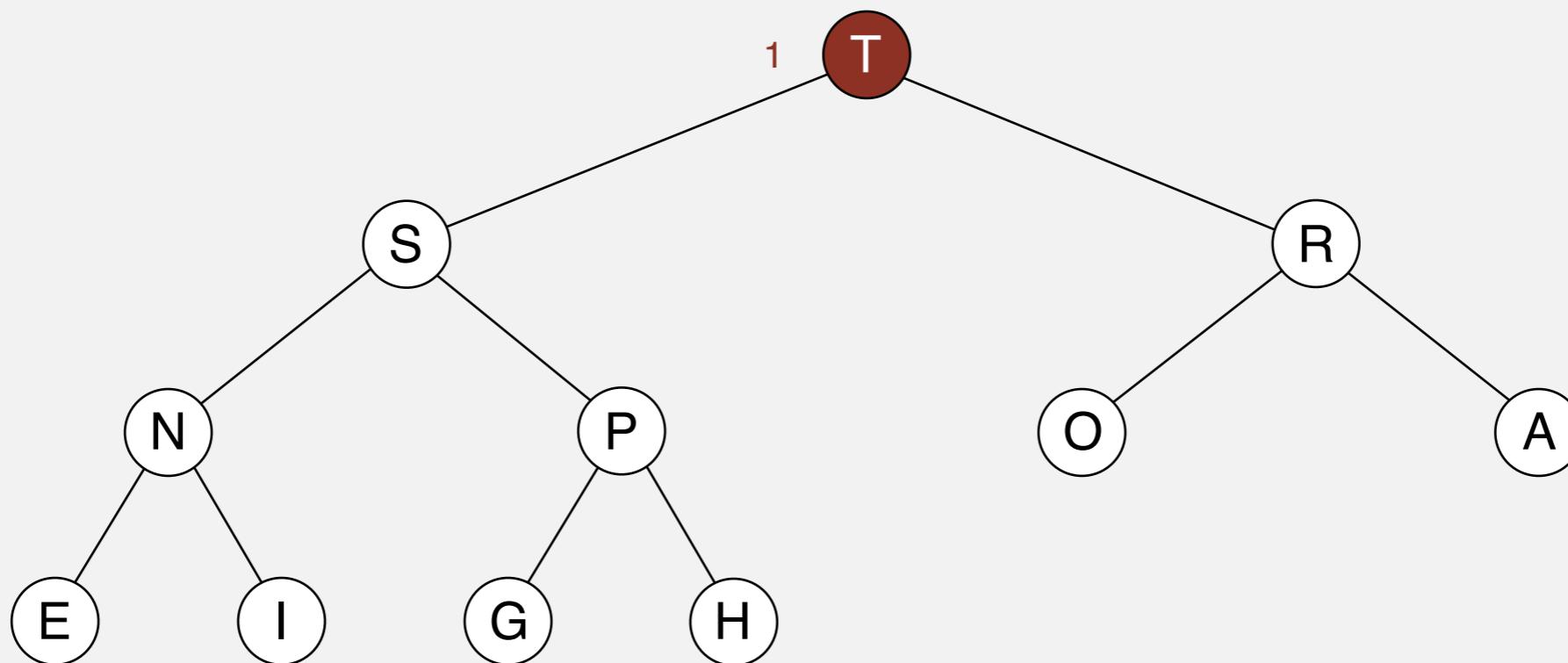
T	S	R	N	P	O	A	E	I	G	H
---	---	---	---	---	---	---	---	---	---	---

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



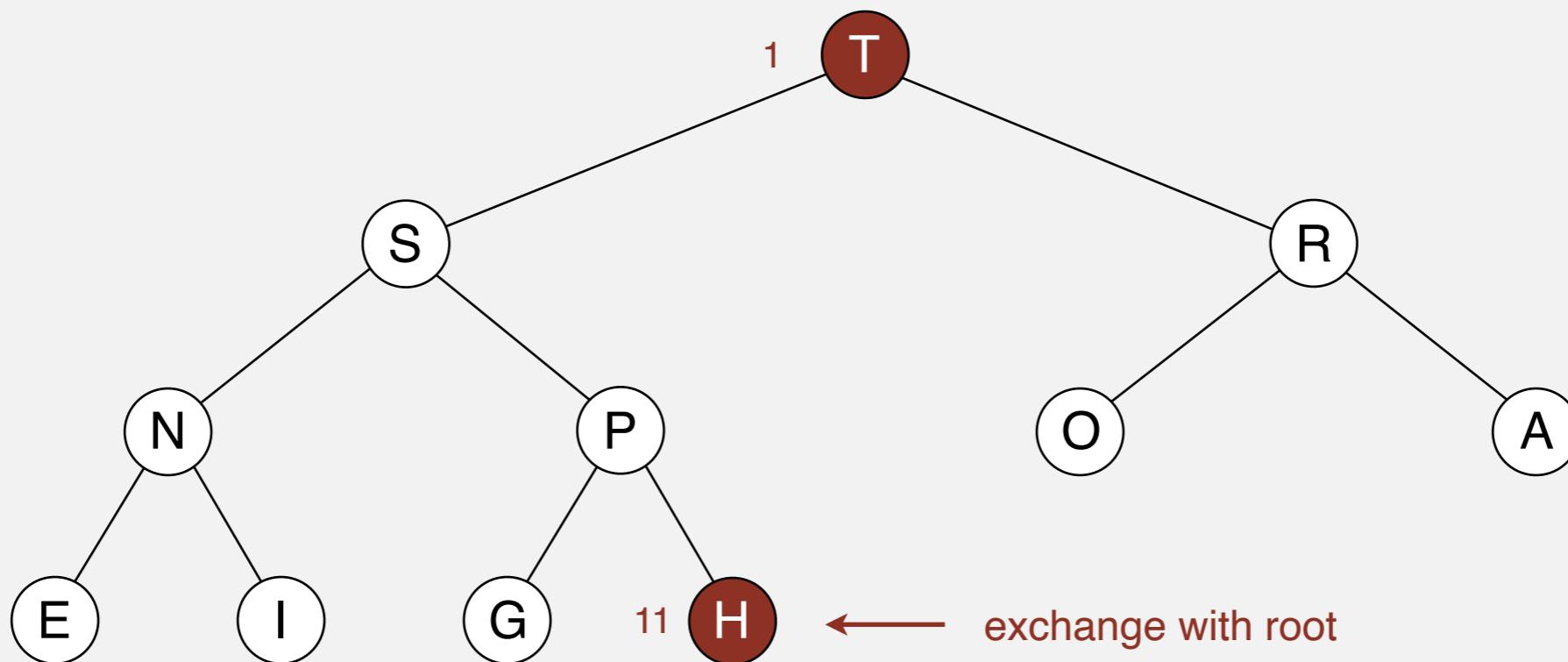
T	S	R	N	P	O	A	E	I	G	H
---	---	---	---	---	---	---	---	---	---	---

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



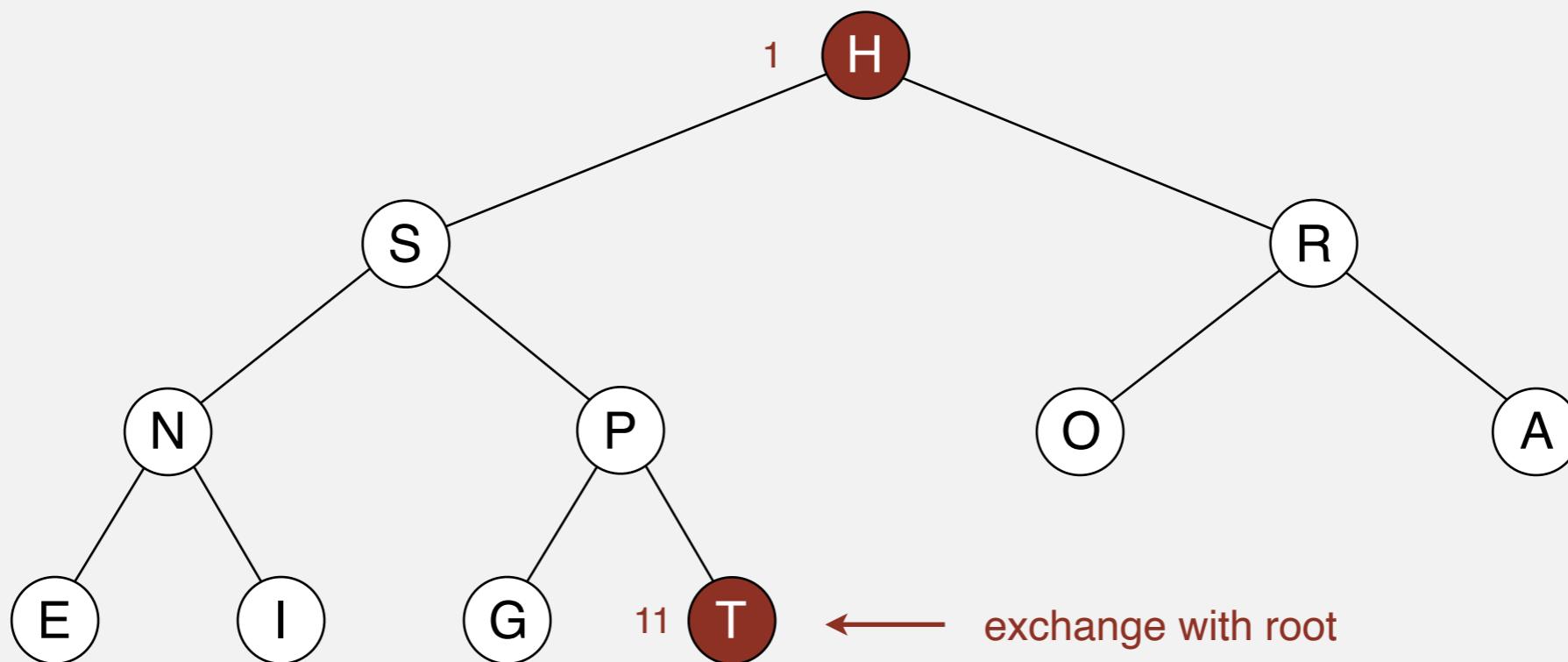
T S R N P O A E I G H

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



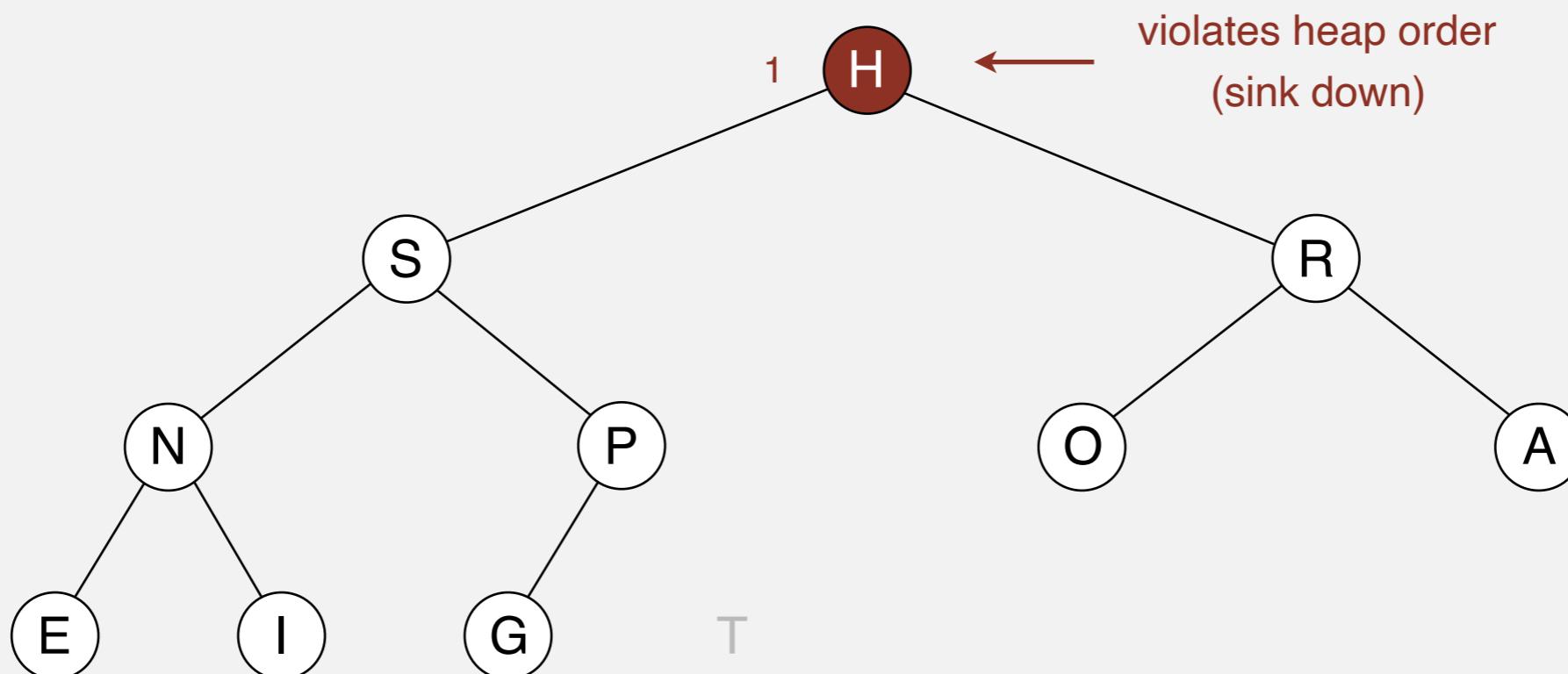
H S R N P O A E I G T

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



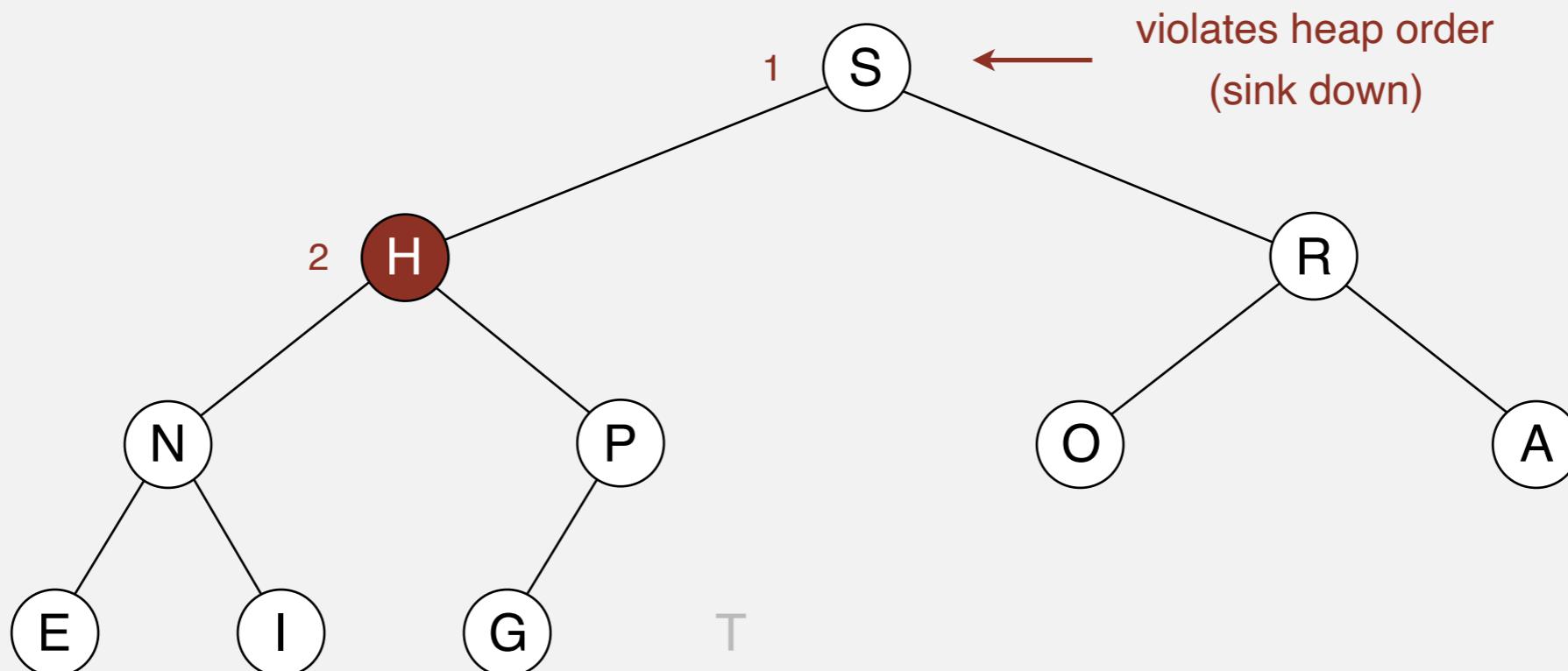
H	S	R	N	P	O	A	E	I	G	T
---	---	---	---	---	---	---	---	---	---	---

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



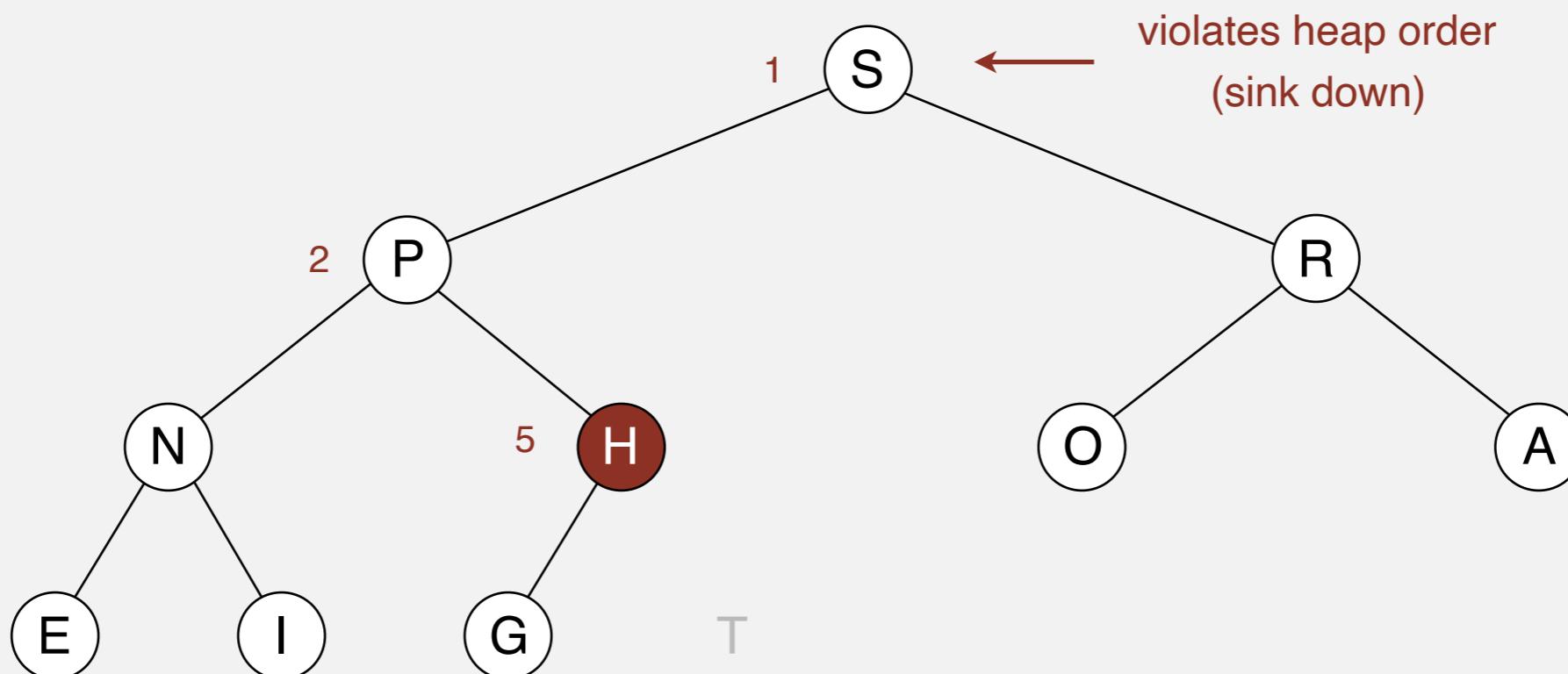
S H R N P O A E I G T
1 2

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



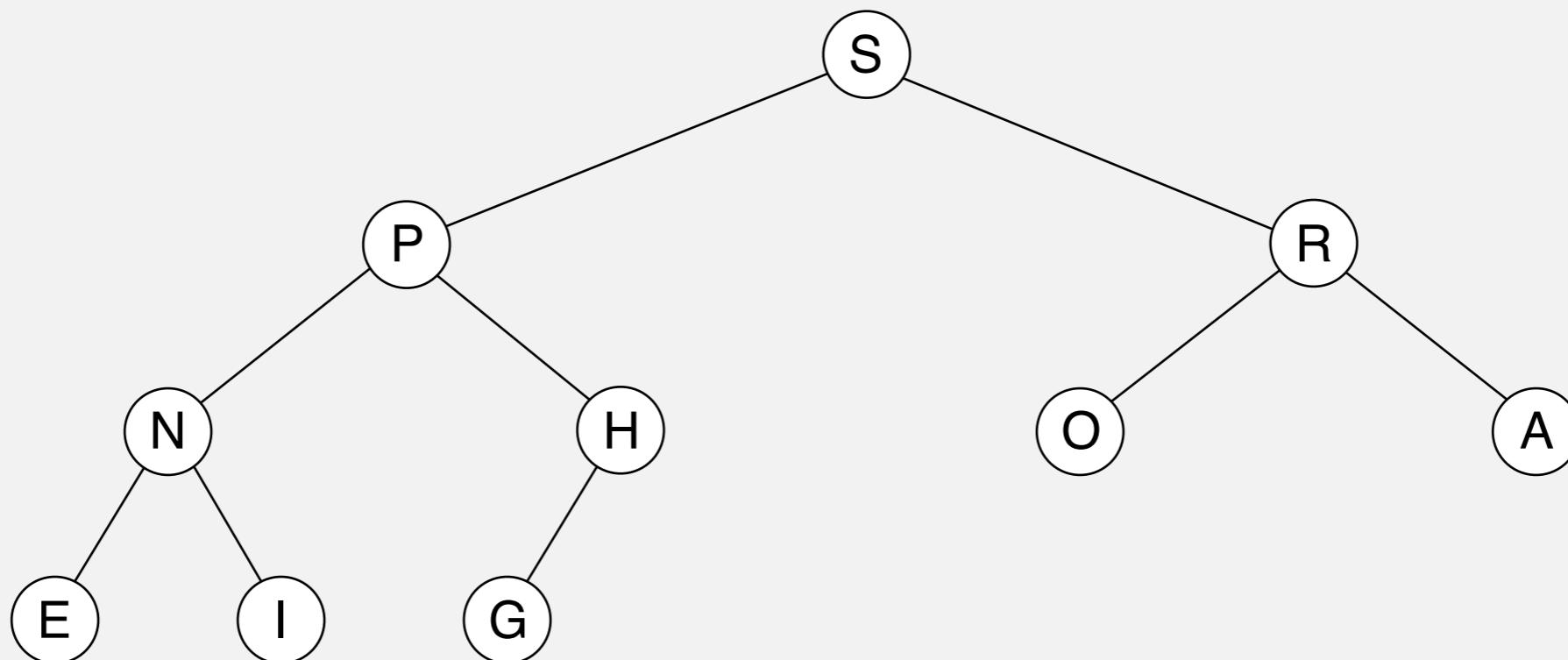
S	P	R	N	H	O	A	E	I	G	T
1	2			5						

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



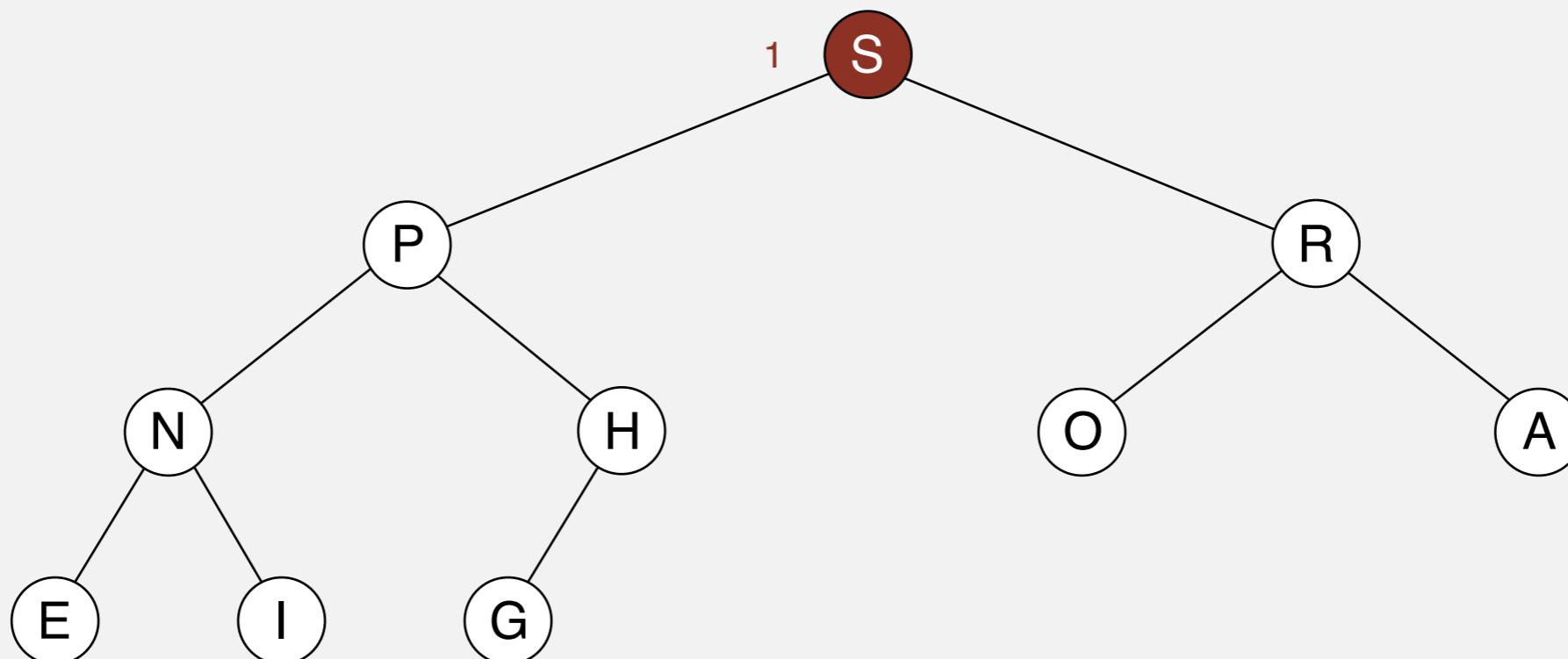
S	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



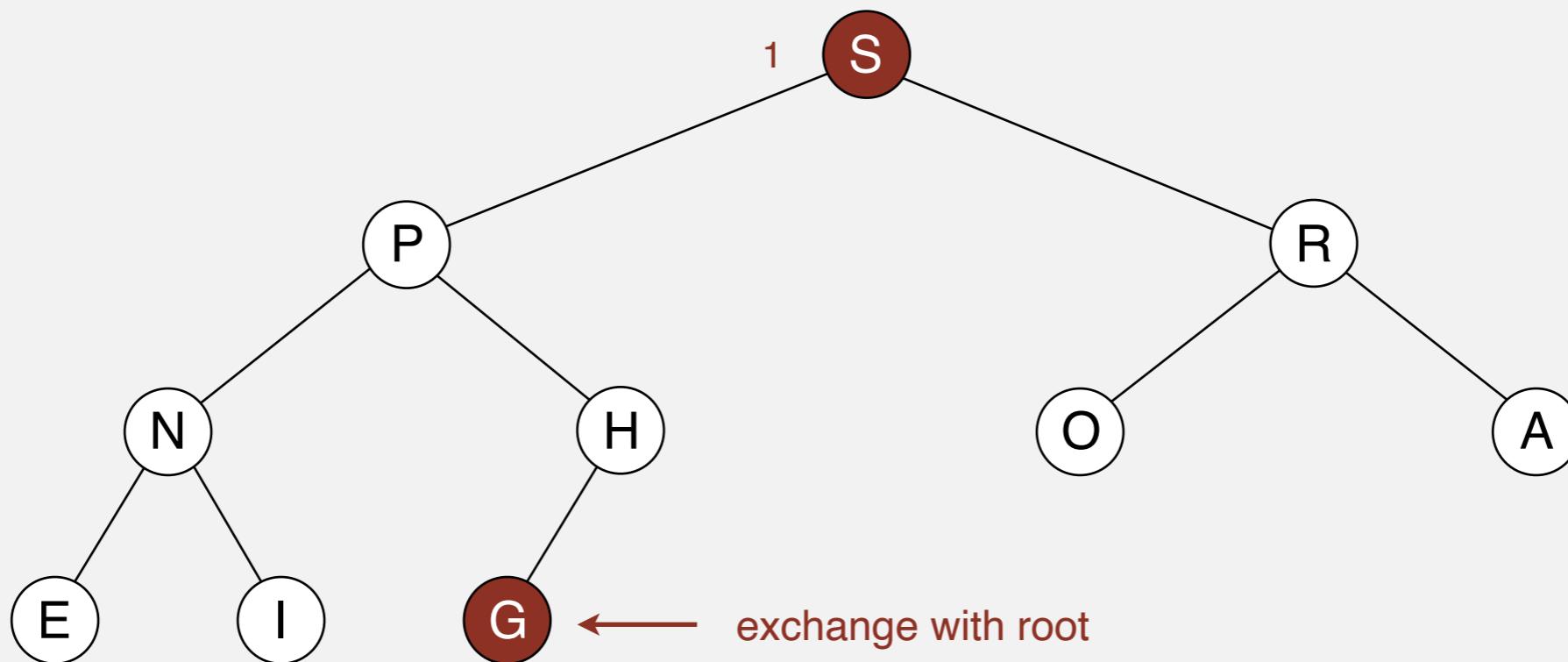
S	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



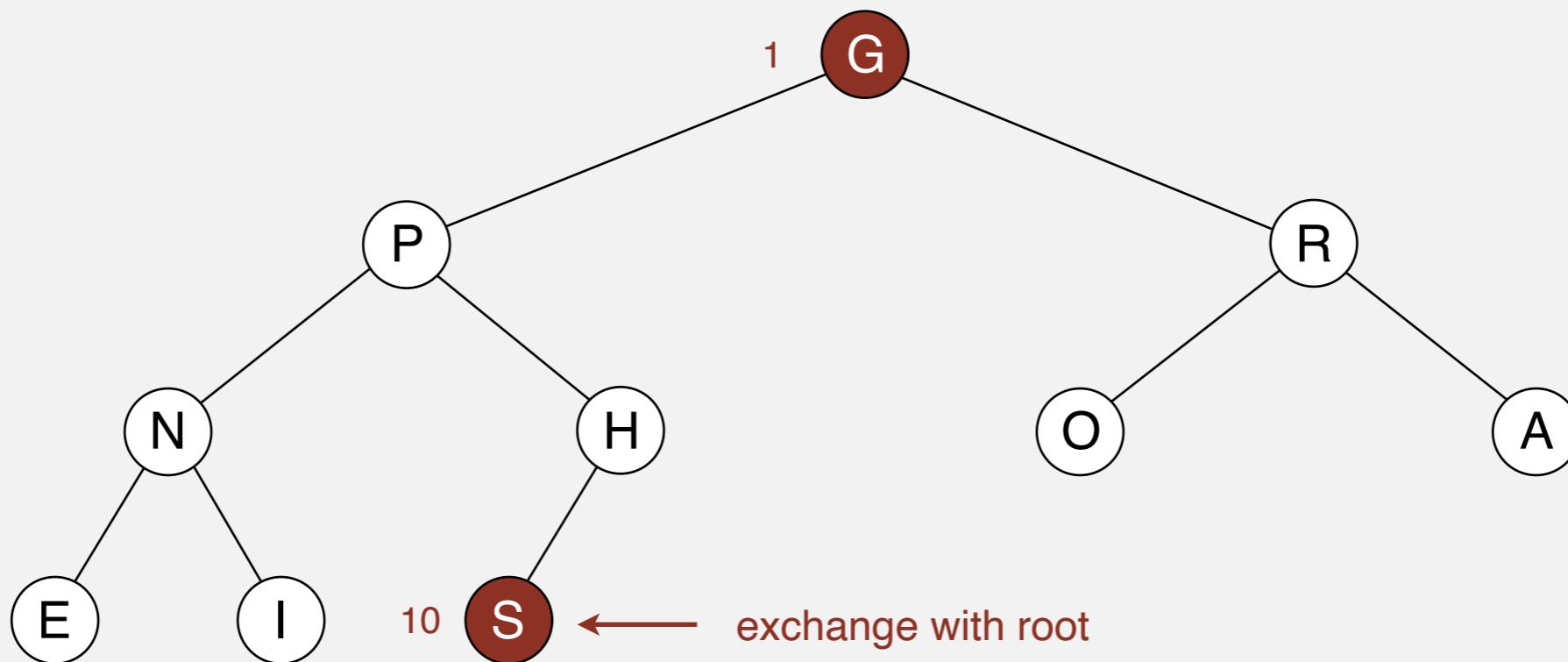
S	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

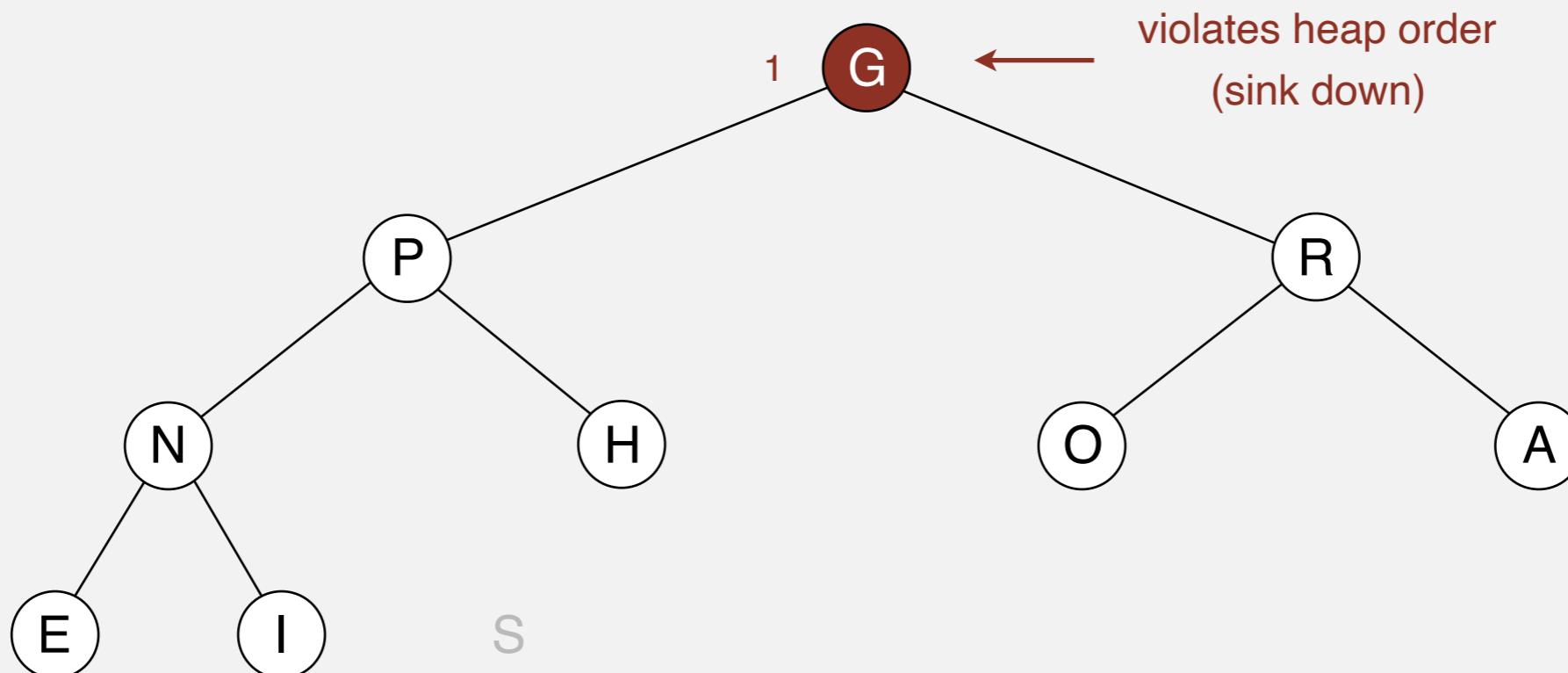


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



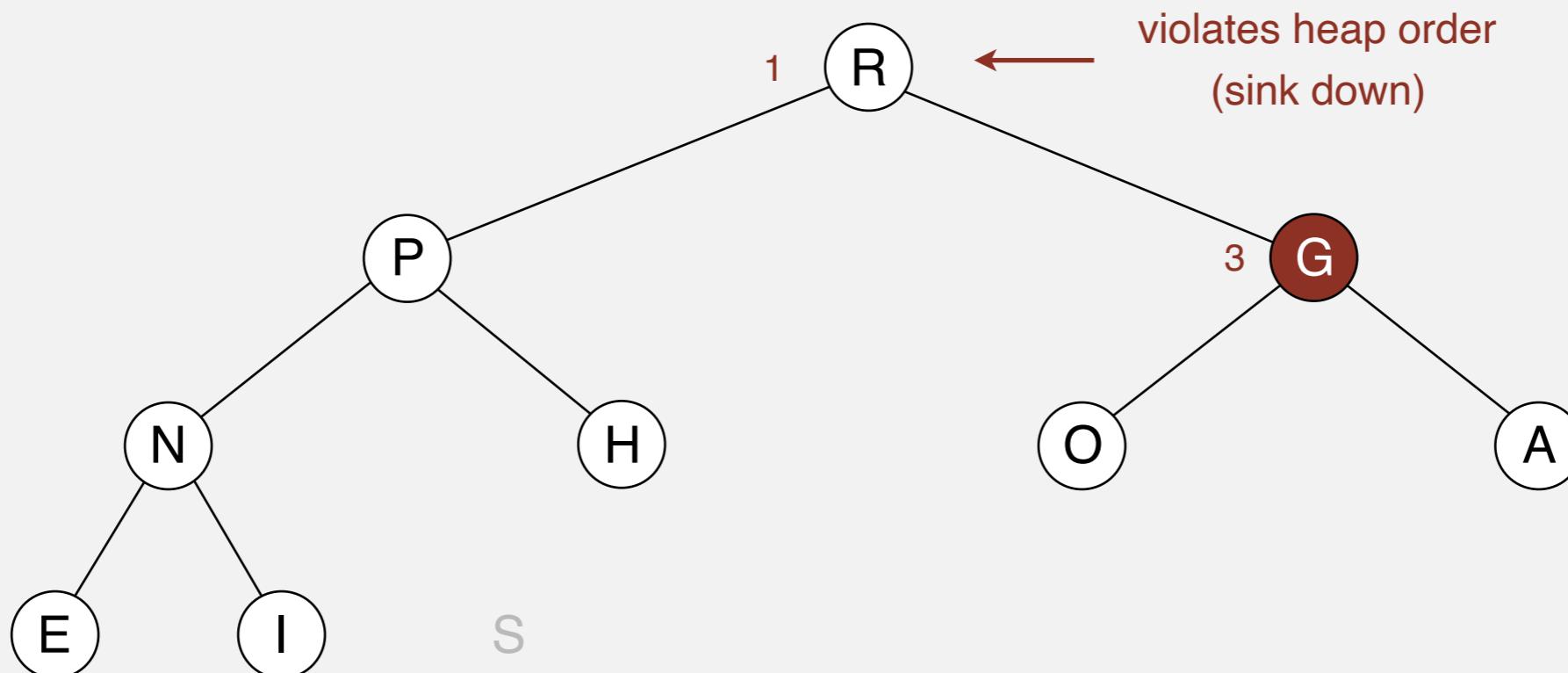
G	P	R	N	H	O	A	E	I	S
---	---	---	---	---	---	---	---	---	---

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



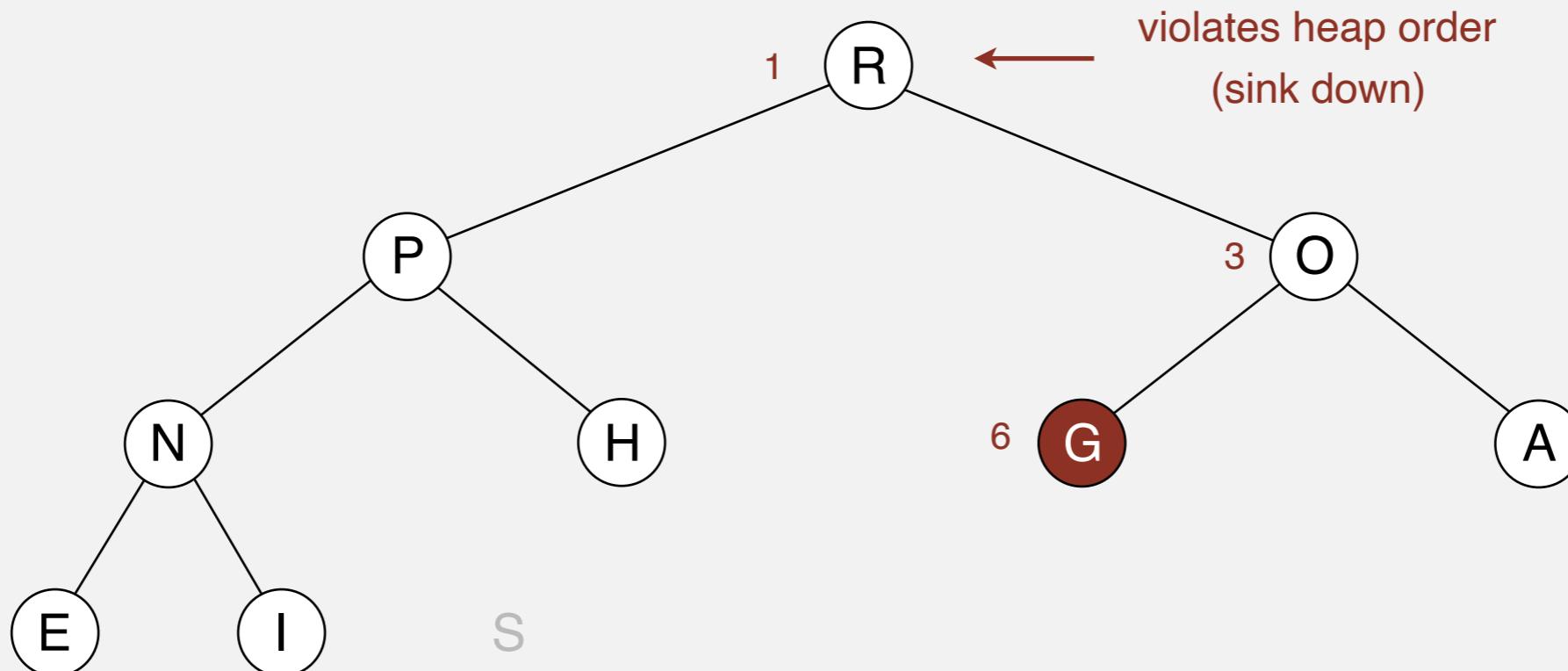
R	P	G	N	H	O	A	E	I	S
1		3							

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



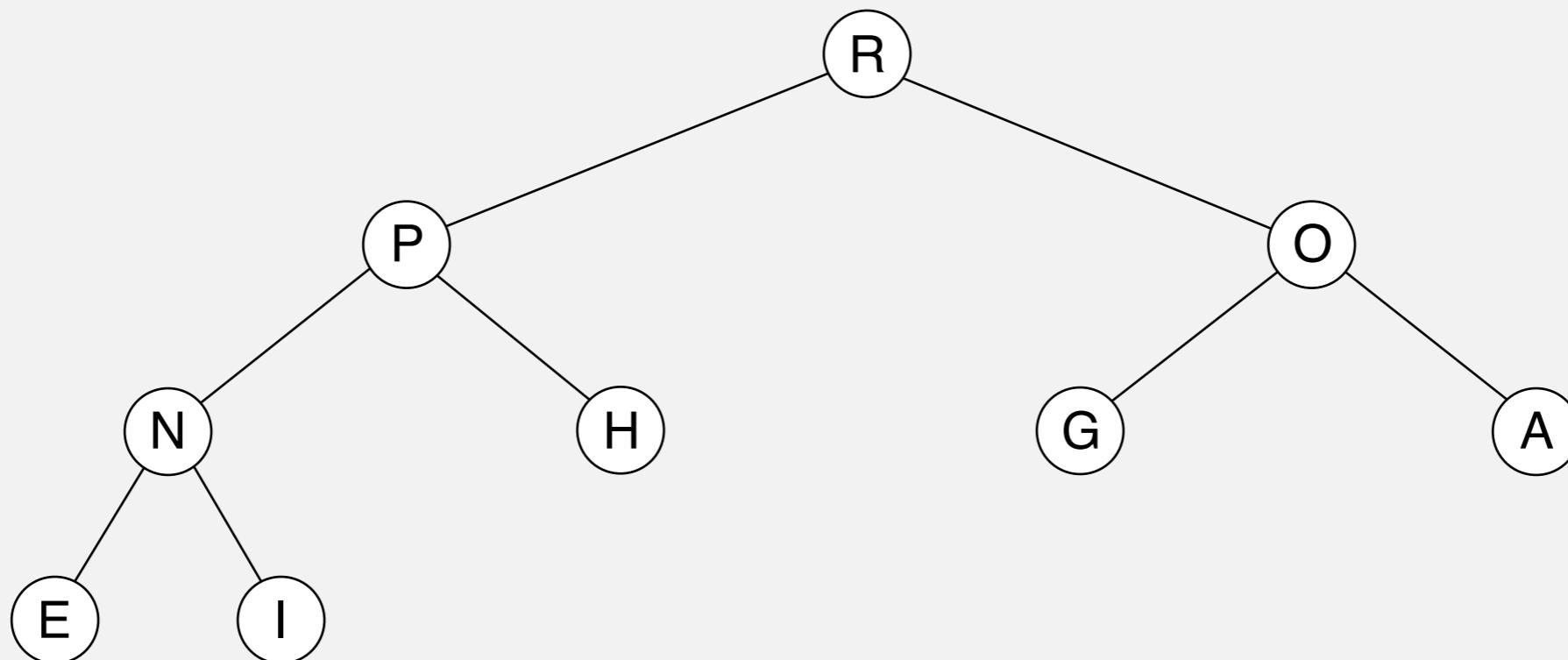
R	P	O	N	H	G	A	E	I	S
1	3	3	6		6				

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



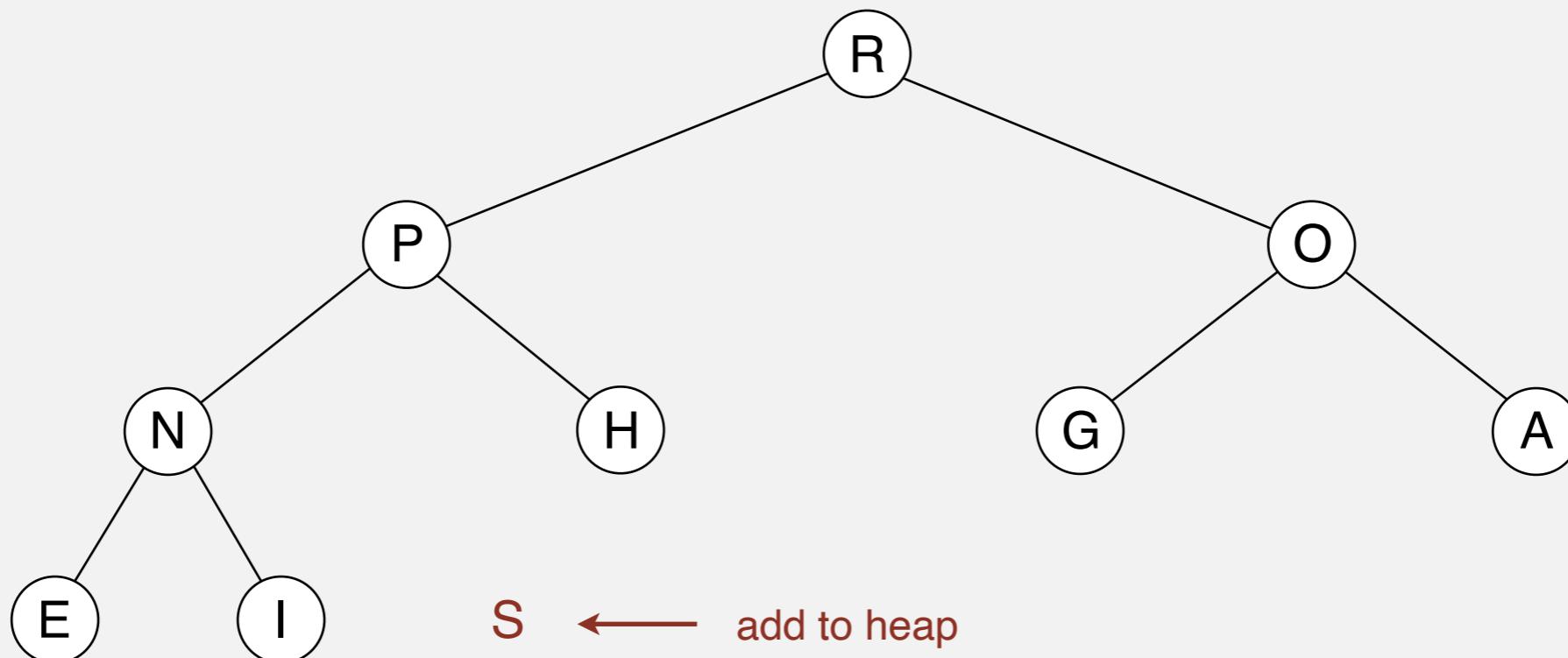
R	P	O	N	H	G	A	E	I		
---	---	---	---	---	---	---	---	---	--	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



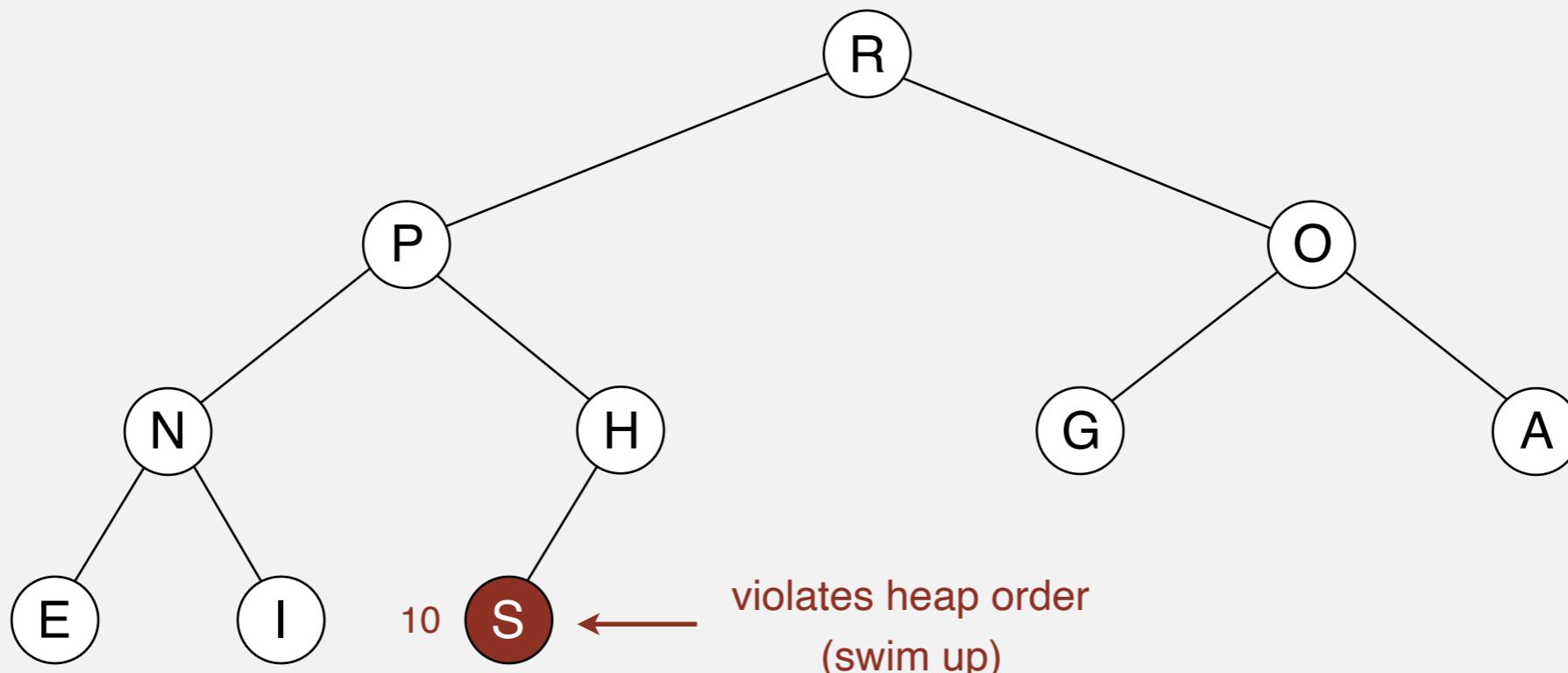
R	P	O	N	H	G	A	E	I	S	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



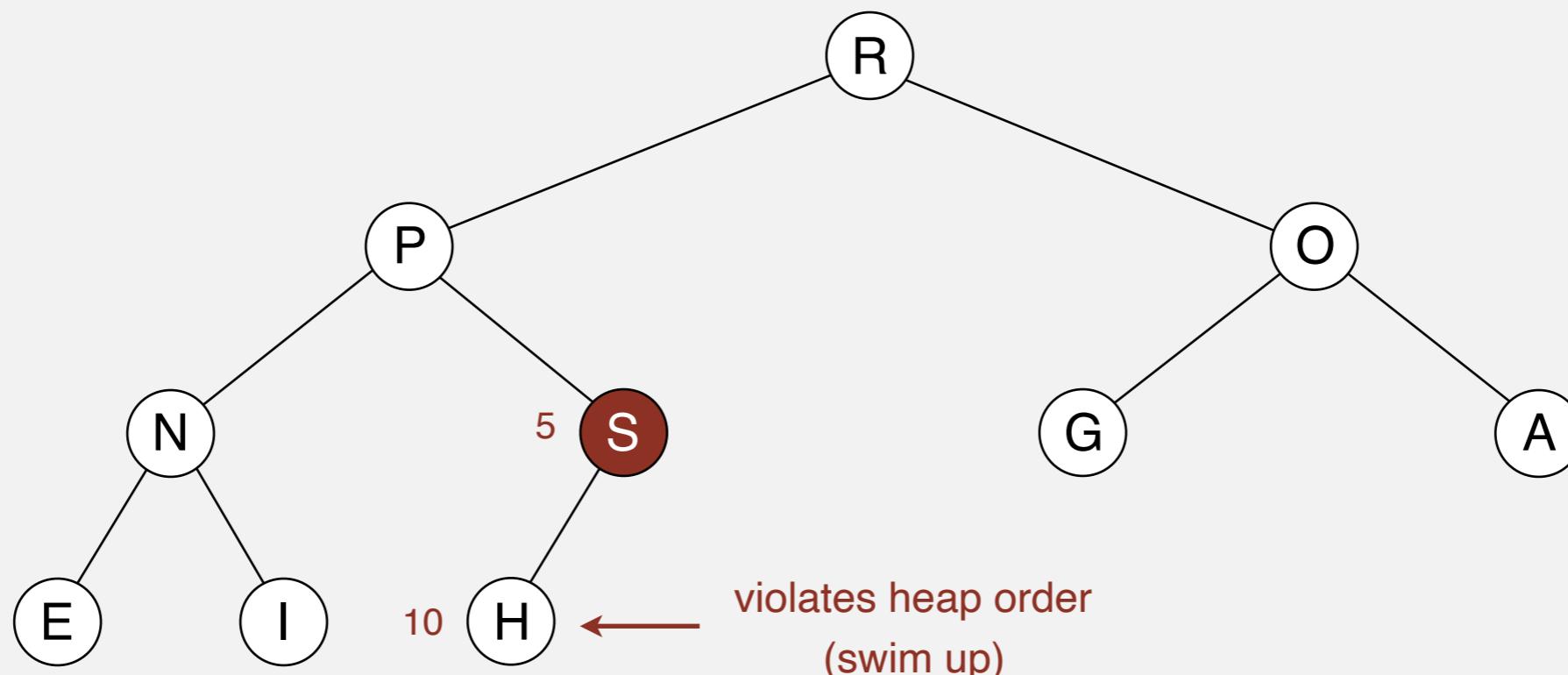
R	P	O	N	H	G	A	E	I	S	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



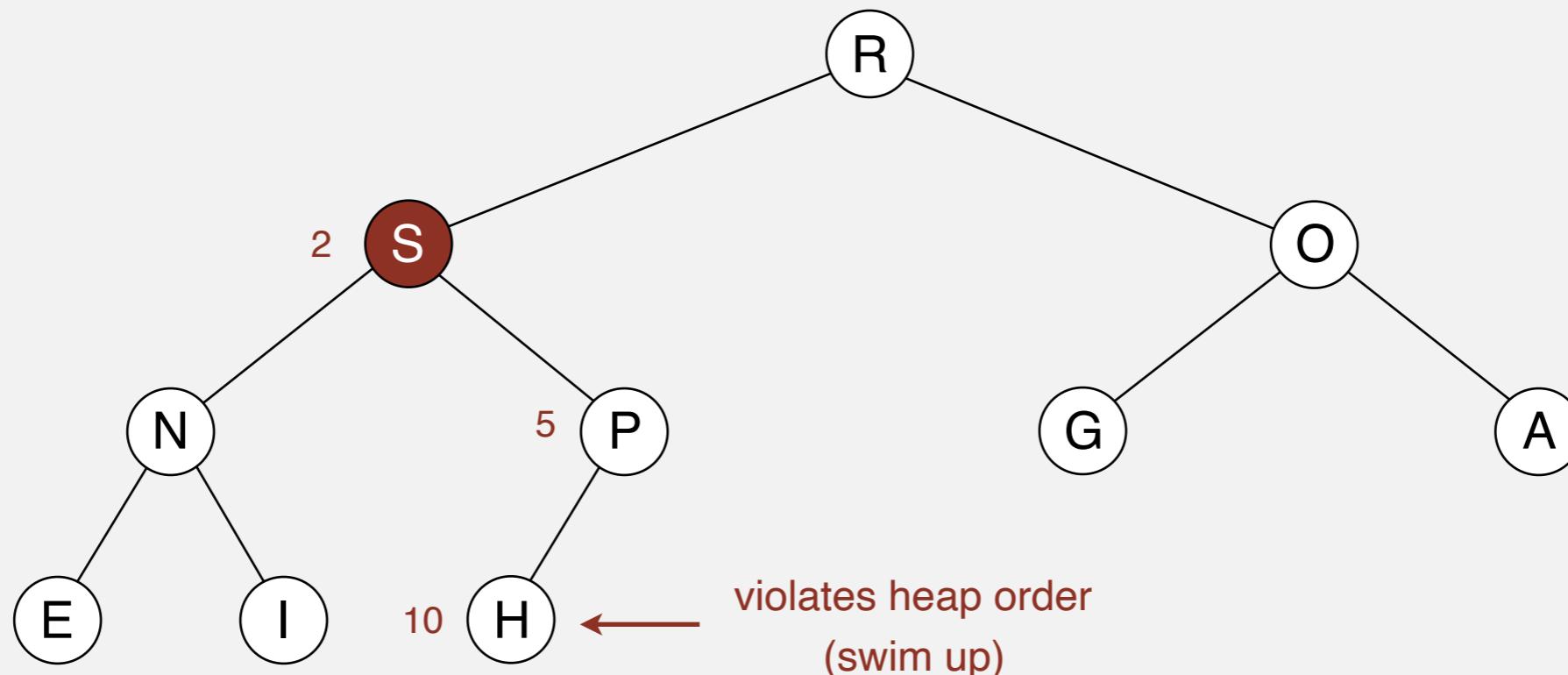
R	P	O	N	S	G	A	E	I	H	
5							10			

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



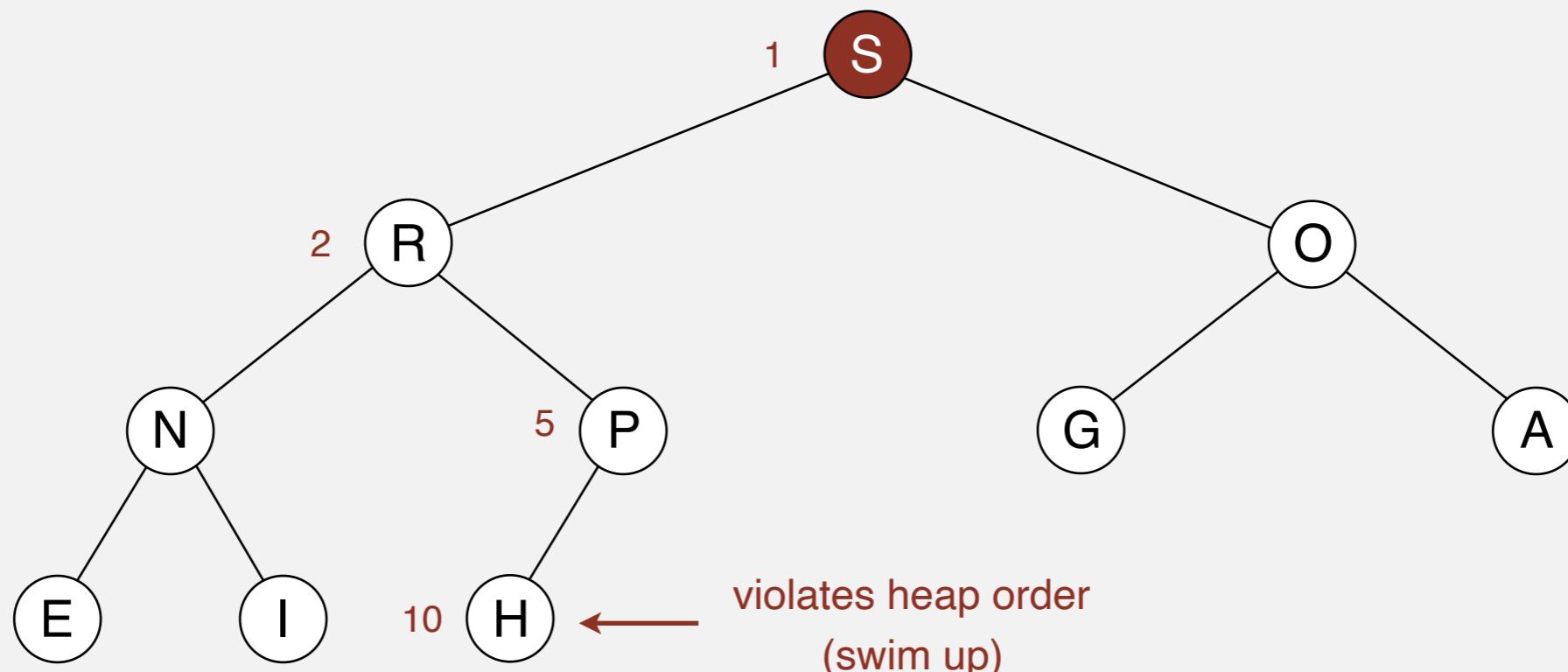
R	S	O	N	P	G	A	E	I	H	
2	2	5		5					10	

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



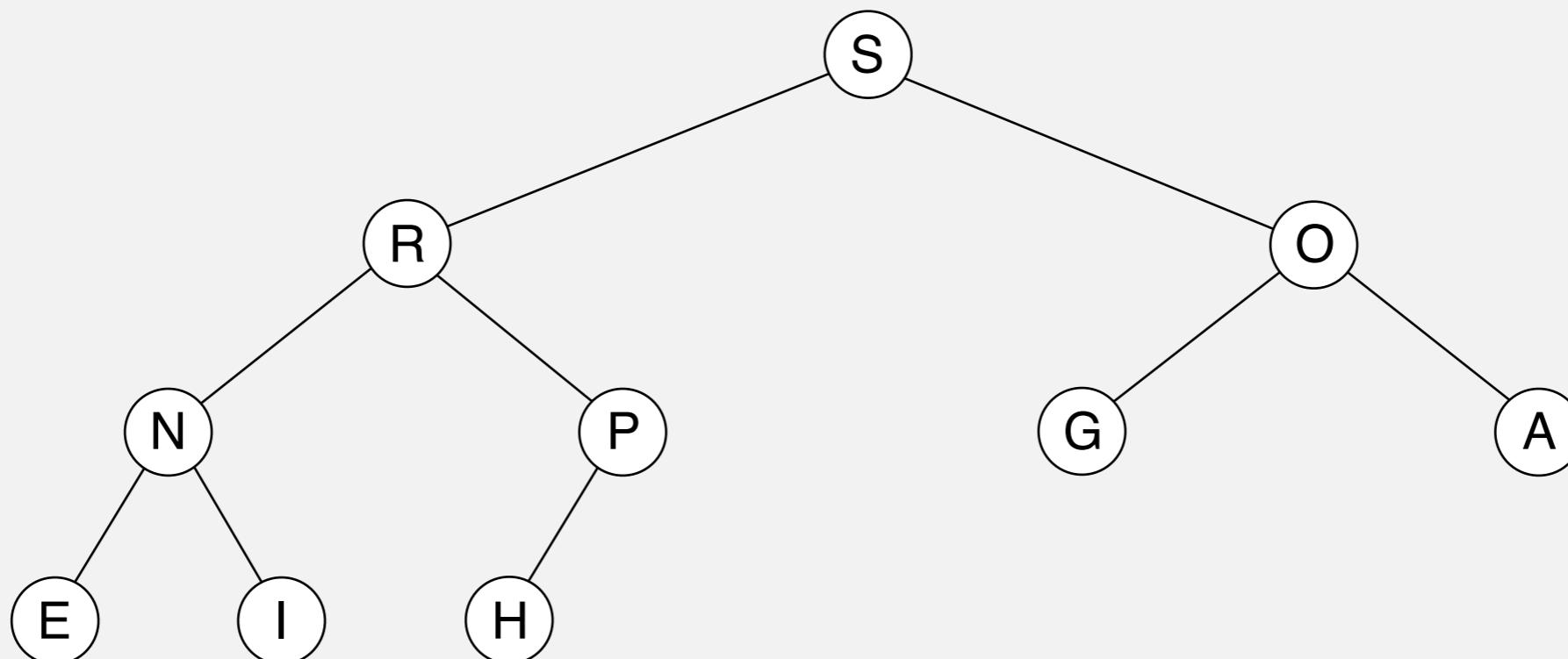
S	R	O	N	P	G	A	E	I	H	
1	2			5					10	

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



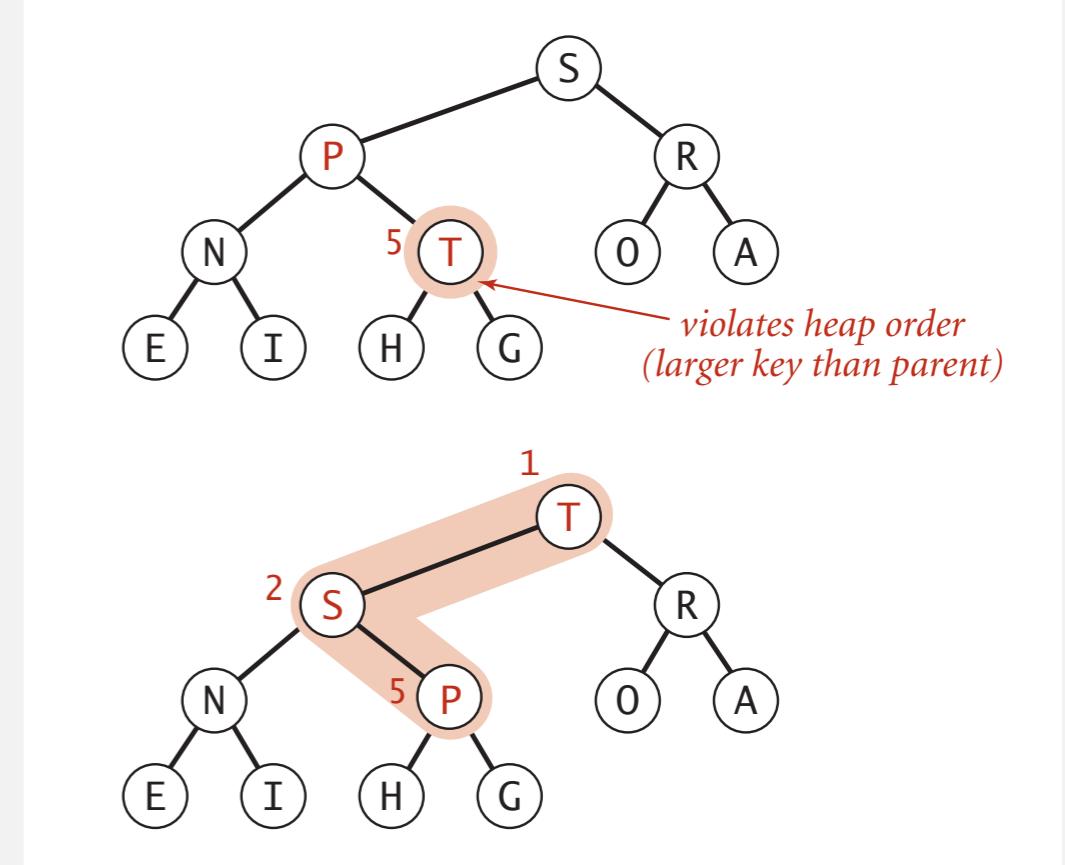
S	R	O	N	P	G	A	E	I	H	
---	---	---	---	---	---	---	---	---	---	--

Promotion in a heap

Scenario. Child's key becomes **larger** key than its parent's key.

To eliminate the violation:

- Exchange key in child with key in parent.
- Repeat until heap order restored.



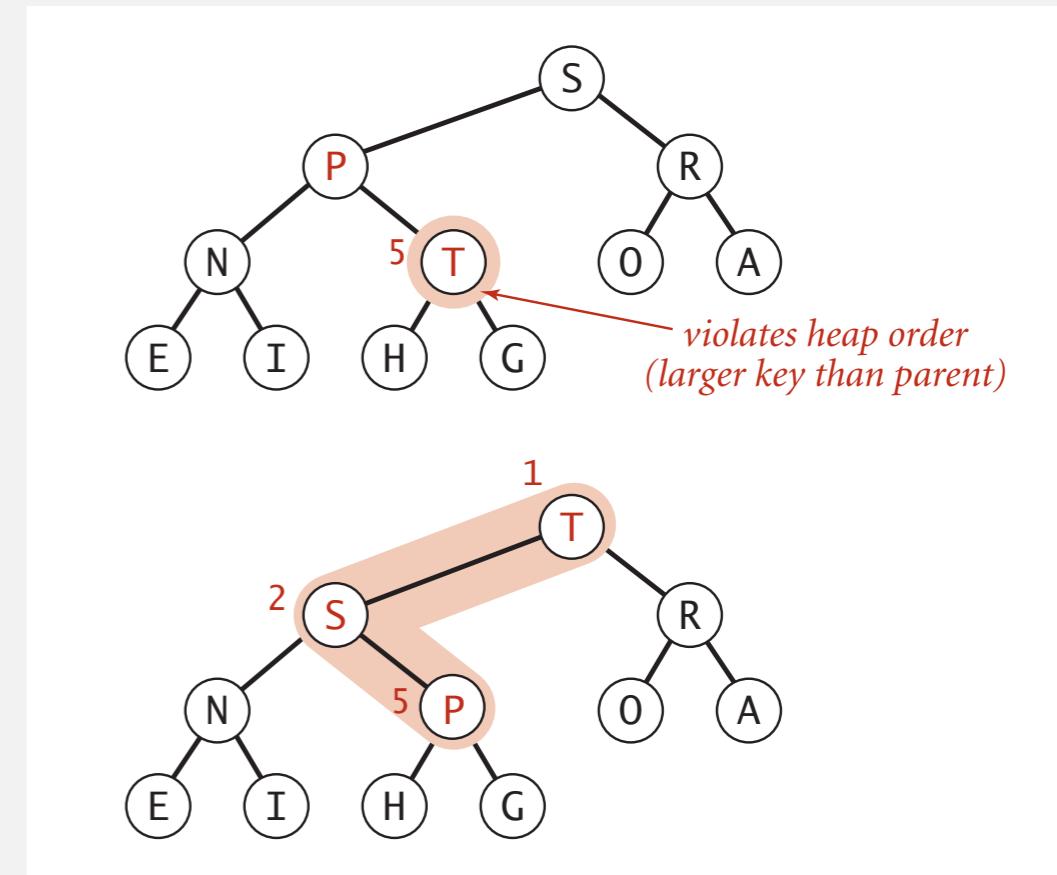
Promotion in a heap

Scenario. Child's key becomes **larger** key than its parent's key.

To eliminate the violation:

- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }    parent of node at k is at k/2
}
```



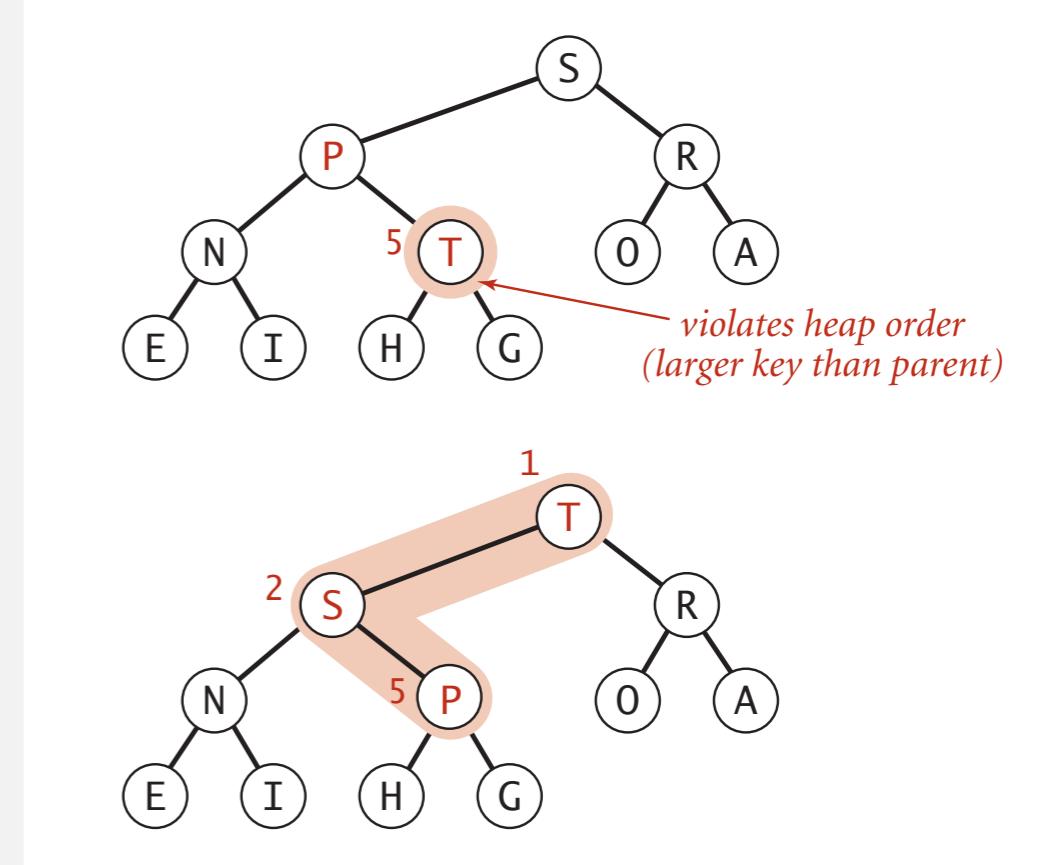
Promotion in a heap

Scenario. Child's key becomes **larger** key than its parent's key.

To eliminate the violation:

- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }  
    parent of node at k is at k/2
}
```



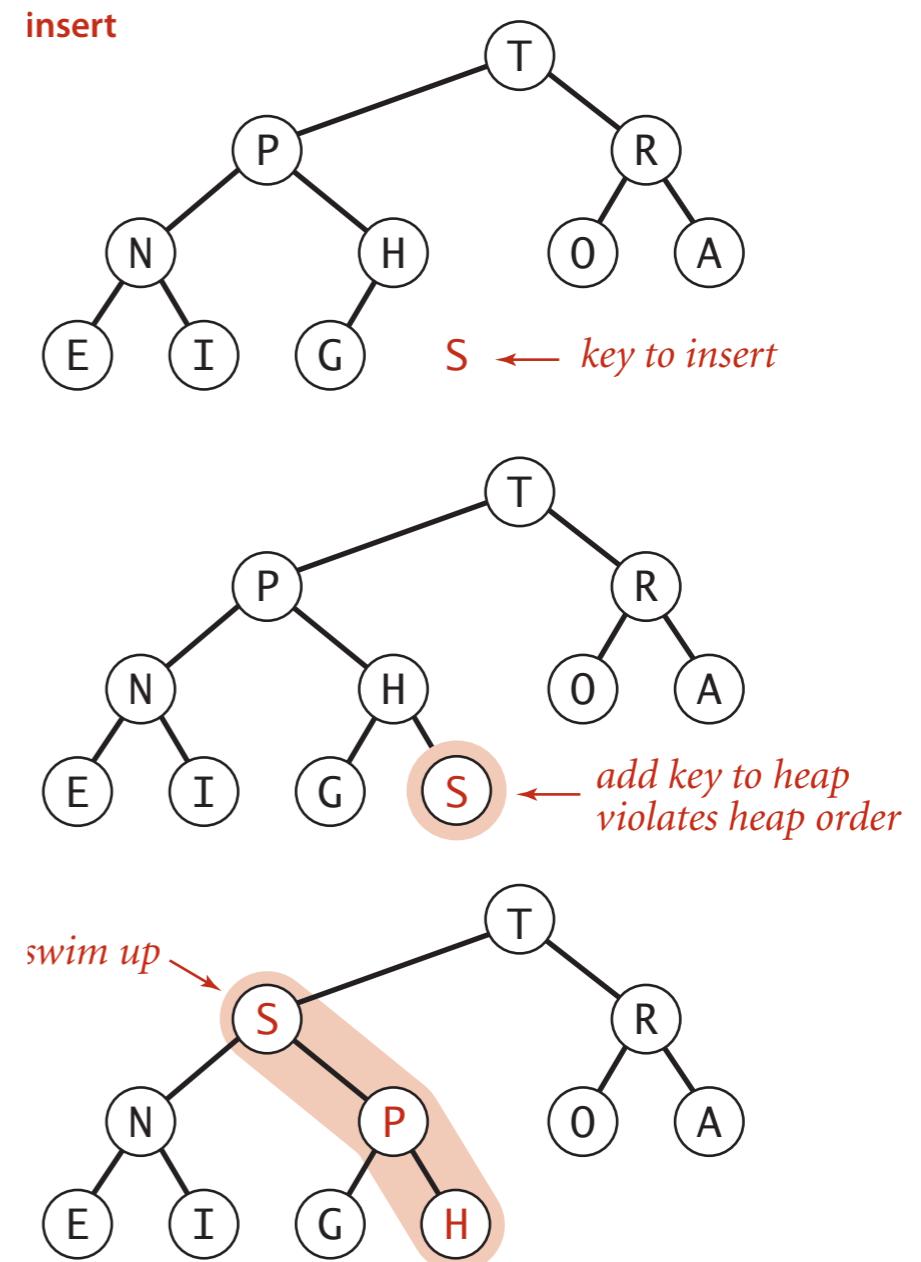
Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then swim it up.

Cost. At most $1 + \lg N$ compares.

```
public void insert(Key x)
{
    pq[++N] = x;
    swim(N);
}
```



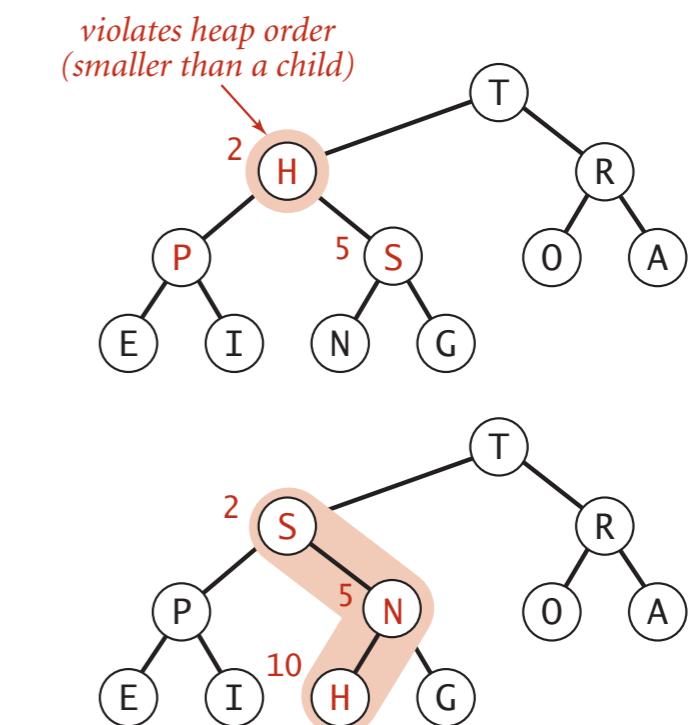
Demotion in a heap

Scenario. Parent's key becomes **smaller** than one (or both) of its children's.

To eliminate the violation:

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

why not smaller child?



Demotion in a heap

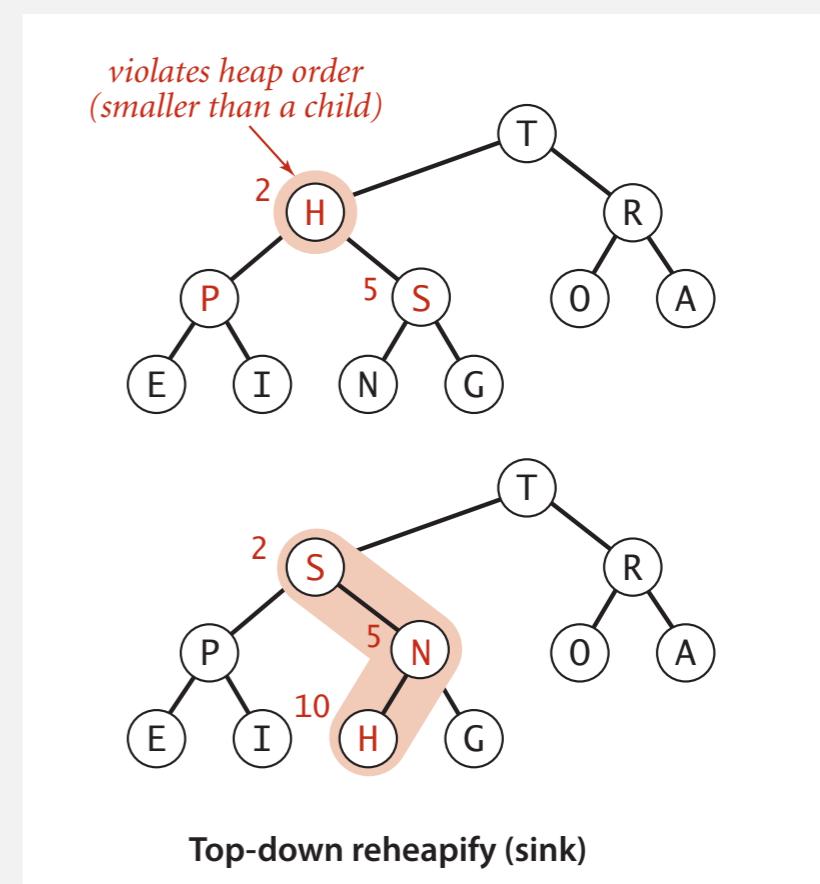
Scenario. Parent's key becomes **smaller** than one (or both) of its children's.

To eliminate the violation:

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

why not smaller child?

```
private void sink(int k)
{
    while (2*k <= N)
    {
        int j = 2*k;           children of node at k are
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```



Demotion in a heap

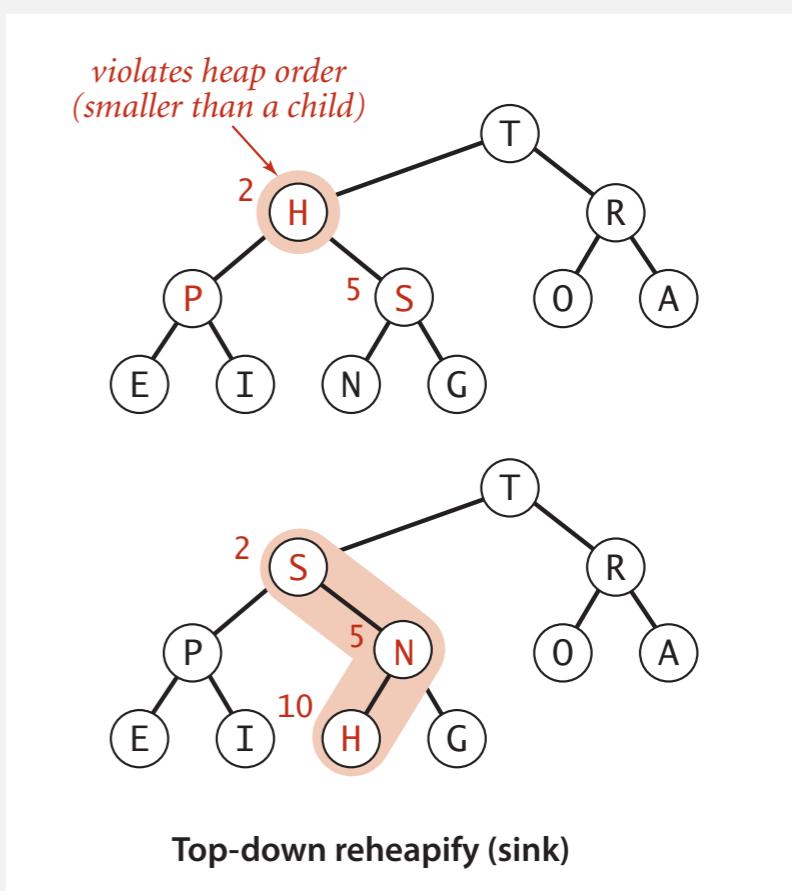
Scenario. Parent's key becomes **smaller** than one (or both) of its children's.

To eliminate the violation:

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

why not smaller child?

```
private void sink(int k)
{
    while (2*k <= N)
    {
        int j = 2*k;           children of node at k are
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

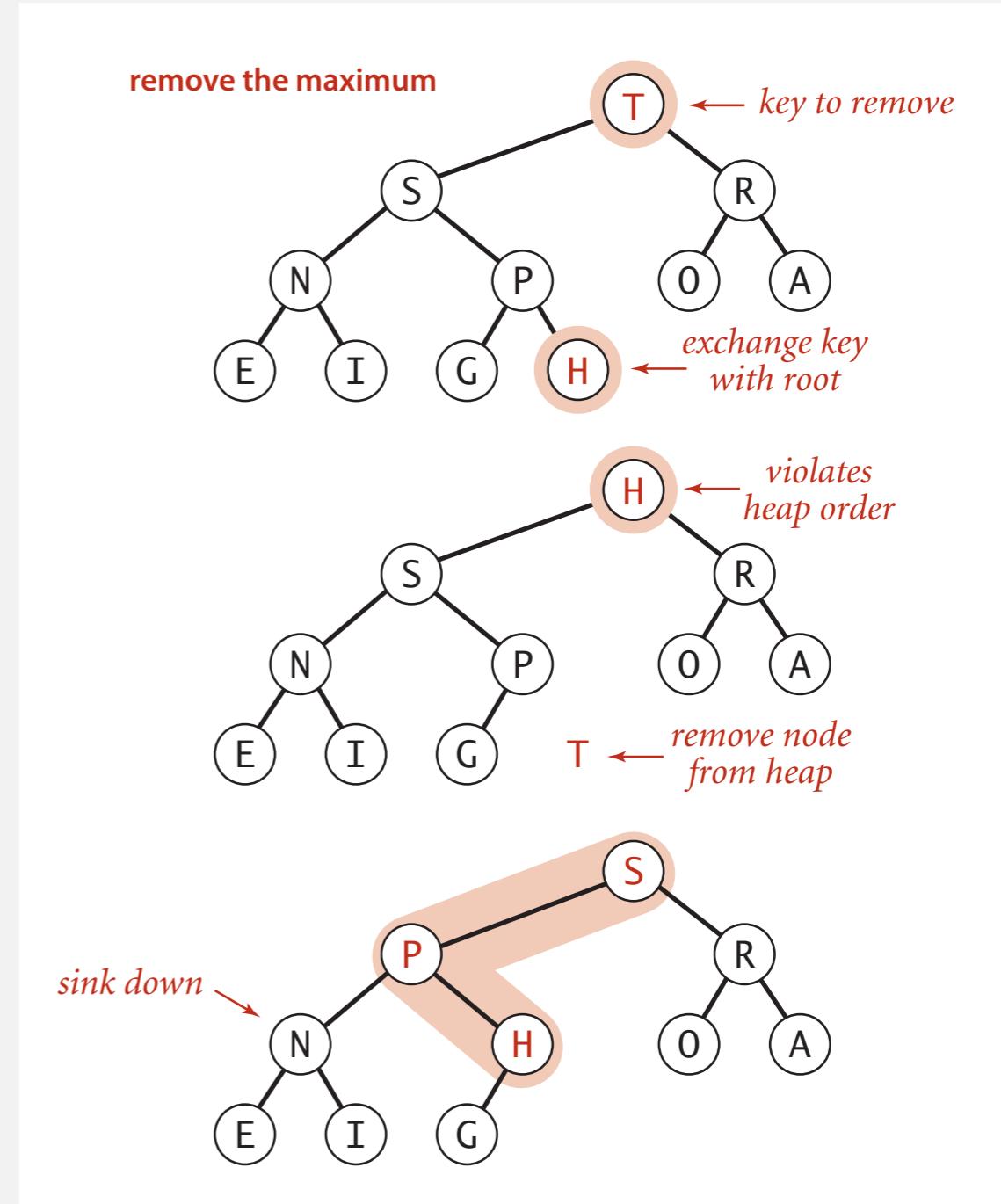


Power struggle. Better subordinate promoted.

Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most $2 \lg N$ compares.

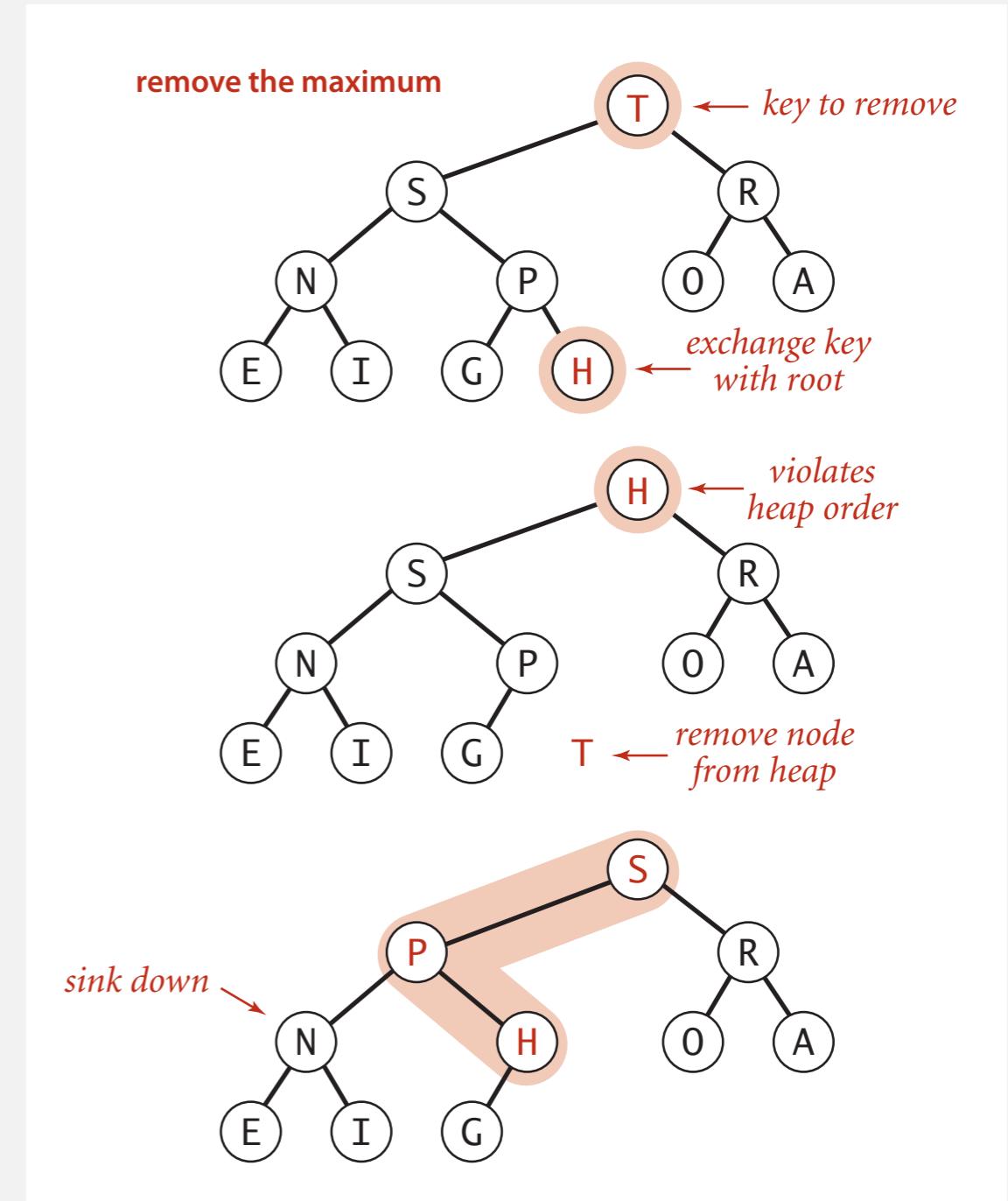


Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most $2 \lg N$ compares.

```
public Key delMax()
{
    Key max = pq[1];
    exch(1, N--);
    sink(1);
    pq[N+1] = null;
    return max;
}
```



Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>> {  
    private Key[] pq;  
    private int N;
```

```
    public MaxPQ(int capacity)  
    { pq = (Key[]) new Comparable[capacity+1]; }
```

fixed capacity
(for simplicity)

```
    public boolean isEmpty()  
    { return N == 0; }  
    public void insert(Key key)  
    public Key delMax()  
    { /* see previous code */ }
```

PQ ops

```
    private void swim(int k)  
    private void sink(int k)  
    { /* see previous code */ }
```

heap helper functions

```
    private boolean less(int i, int j)  
    { return pq[i].compareTo(pq[j]) < 0; }  
    private void exch(int i, int j)  
    { Key t = pq[i]; pq[i] = pq[j]; pq[j] = t; }  
}
```

array helper functions

Priority queues implementation cost summary

order-of-growth of running time for priority queue with N items

Priority queues implementation cost summary

implementation	insert	del max	max

order-of-growth of running time for priority queue with N items

Priority queues implementation cost summary

implementation	insert	del max	max
unordered array	1	N	N

order-of-growth of running time for priority queue with N items

Priority queues implementation cost summary

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1

order-of-growth of running time for priority queue with N items

Priority queues implementation cost summary

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	$\log N$	$\log N$	1

order-of-growth of running time for priority queue with N items

Binary heap considerations

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to log N
amortized time per op
(how to make worst case?)

Binary heap considerations

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to log N
amortized time per op
(how to make worst case?)

Minimum-oriented priority queue.

- Replace less() with greater().
- Implement greater().

Binary heap considerations

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to log N

amortized time per op

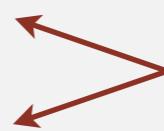
(how to make worst case?)

Minimum-oriented priority queue.

- Replace less() with greater().
- Implement greater().

Other operations.

- Remove an arbitrary item.
- Change the priority of an item.



can implement efficiently with sink() and swim()

[stay tuned for Prim/Dijkstra]

Binary heap considerations

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to log N

amortized time per op

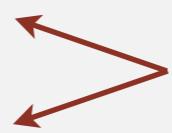
(how to make worst case?)

Minimum-oriented priority queue.

- Replace less() with greater().
- Implement greater().

Other operations.

- Remove an arbitrary item.
- Change the priority of an item.



can implement efficiently with sink() and swim()

[stay tuned for Prim/Dijkstra]

Immutability of keys.

- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

2.4 PRIORITY QUEUES

- ▶ API and elementary implementations
- ▶ binary heaps
- ▶ **heapsort**
- ▶ event-driven simulation

Sorting with a binary heap

Q. What is this sorting algorithm?

```
public void sort(String[] a)
{
    int N = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < N; i++)
        pq.insert(a[i]);
    for (int i = N-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

Sorting with a binary heap

Q. What is this sorting algorithm?

```
public void sort(String[] a)
{
    int N = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < N; i++)
        pq.insert(a[i]);
    for (int i = N-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

Q. What are its properties?

Sorting with a binary heap

Q. What is this sorting algorithm?

```
public void sort(String[] a)
{
    int N = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < N; i++)
        pq.insert(a[i]);
    for (int i = N-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

Q. What are its properties?

A. $N \log N$, extra array of length N , not stable.

Sorting with a binary heap

Q. What is this sorting algorithm?

```
public void sort(String[] a)
{
    int N = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < N; i++)
        pq.insert(a[i]);
    for (int i = N-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

Q. What are its properties?

A. $N \log N$, extra array of length N , not stable.

Heapsort intuition. A heap is an array; do sort in place.

Heapsort

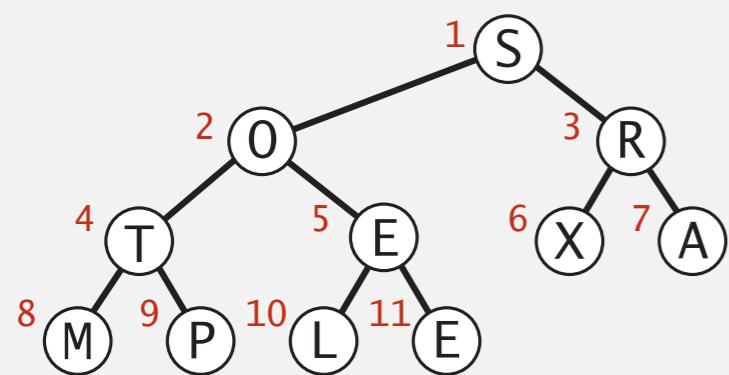
Basic plan for in-place sort.

Heapsort

Basic plan for in-place sort.

- View input array as a complete binary tree.

keys in arbitrary order



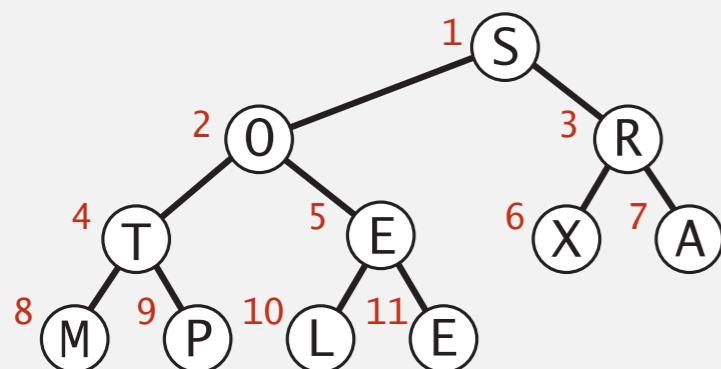
1	2	3	4	5	6	7	8	9	10	11
S	O	R	T	E	X	A	M	P	L	E

Heapsort

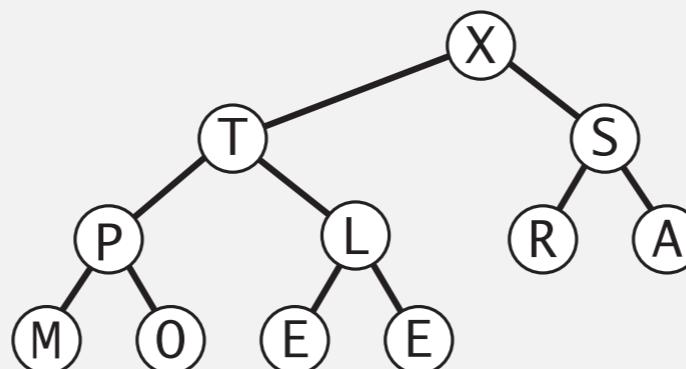
Basic plan for in-place sort.

- View input array as a complete binary tree.
- Heap construction: build a max-heap with all N keys.

keys in arbitrary order



build max heap
(in place)



1	2	3	4	5	6	7	8	9	10	11
S	O	R	T	E	X	A	M	P	L	E

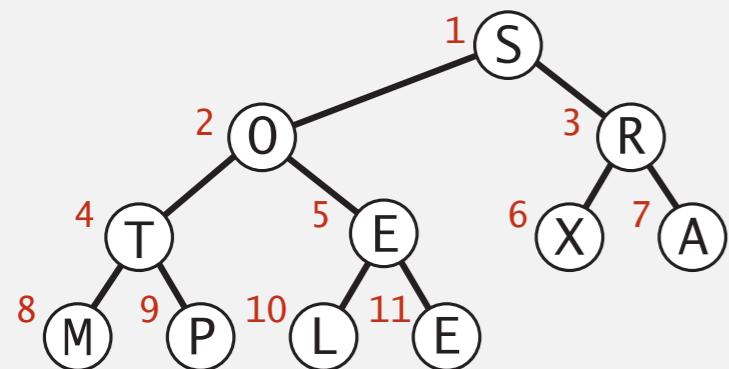
1	2	3	4	5	6	7	8	9	10	11
X	T	S	P	L	R	A	M	O	E	E

Heapsort

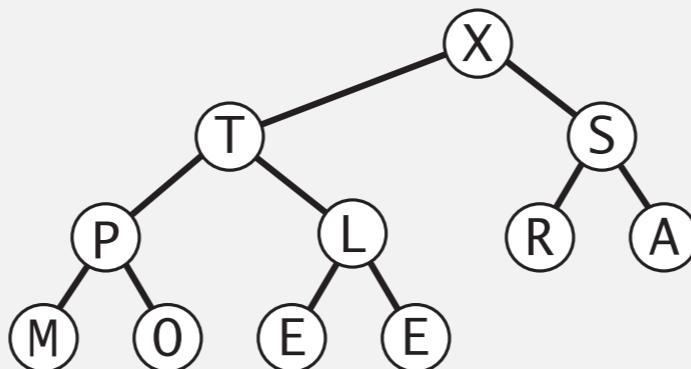
Basic plan for in-place sort.

- View input array as a complete binary tree.
- Heap construction: build a max-heap with all N keys.
- Sortdown: repeatedly remove the maximum key.

keys in arbitrary order



build max heap
(in place)



sorted result
(in place)



1	2	3	4	5	6	7	8	9	10	11
S	O	R	T	E	X	A	M	P	L	E

1	2	3	4	5	6	7	8	9	10	11
X	T	S	P	L	R	A	M	O	E	E

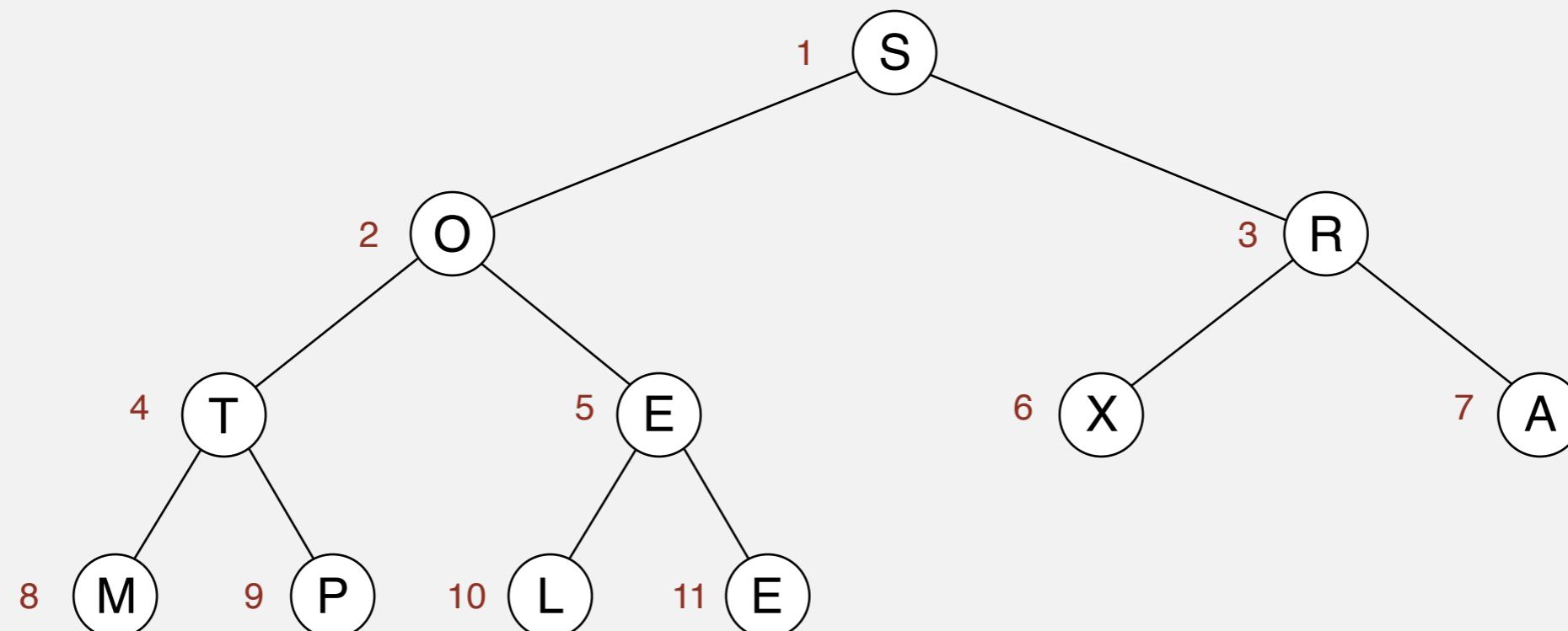
1	2	3	4	5	6	7	8	9	10	11
A	E	E	L	M	O	P	R	S	T	X

Heapsort demo

Heap construction. Build max heap using bottom-up method.

we assume array entries are indexed 1 to N

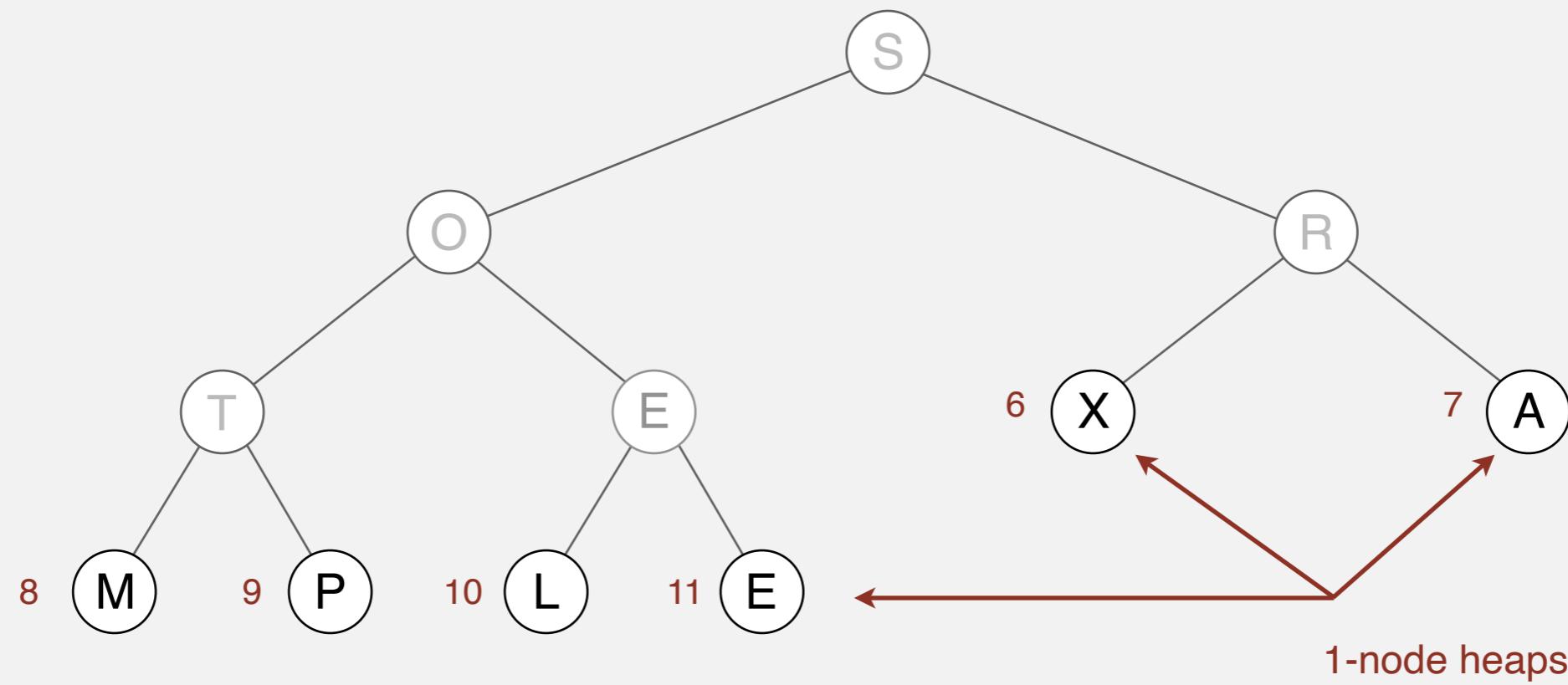
array in arbitrary order



S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort demo

Heap construction. Build max heap using bottom-up method.

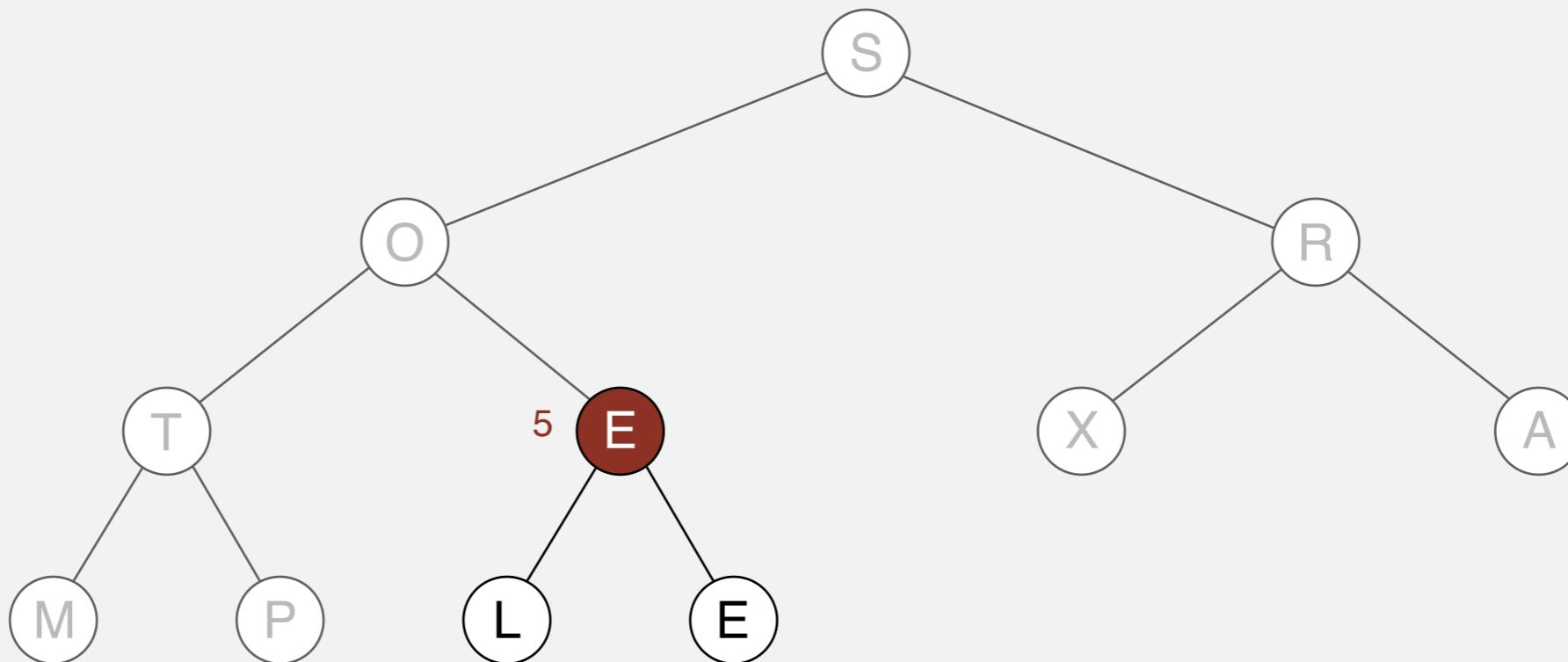


S	O	R	T	E	X	A	M	P	L	E
6	7	8	9	10	11					

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 5

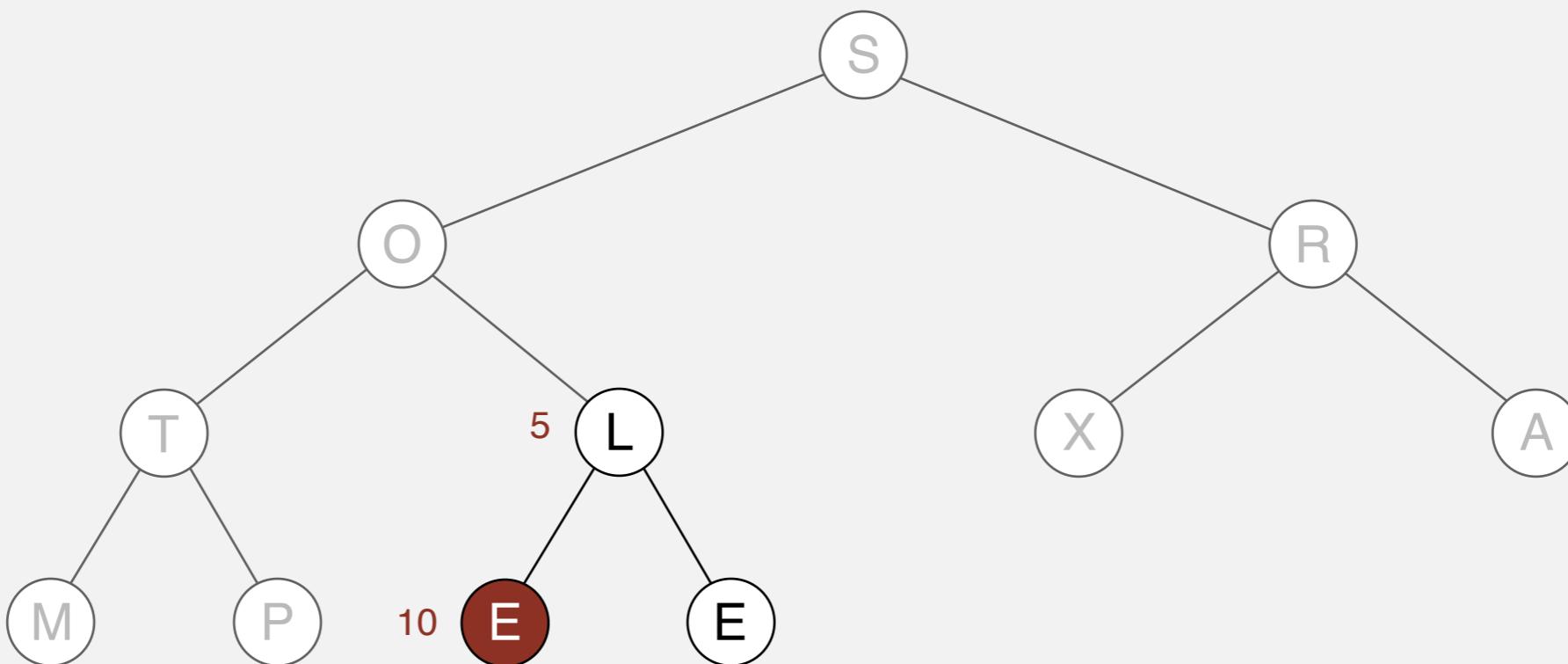


S	O	R	T	E	X	A	M	P	L	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 5

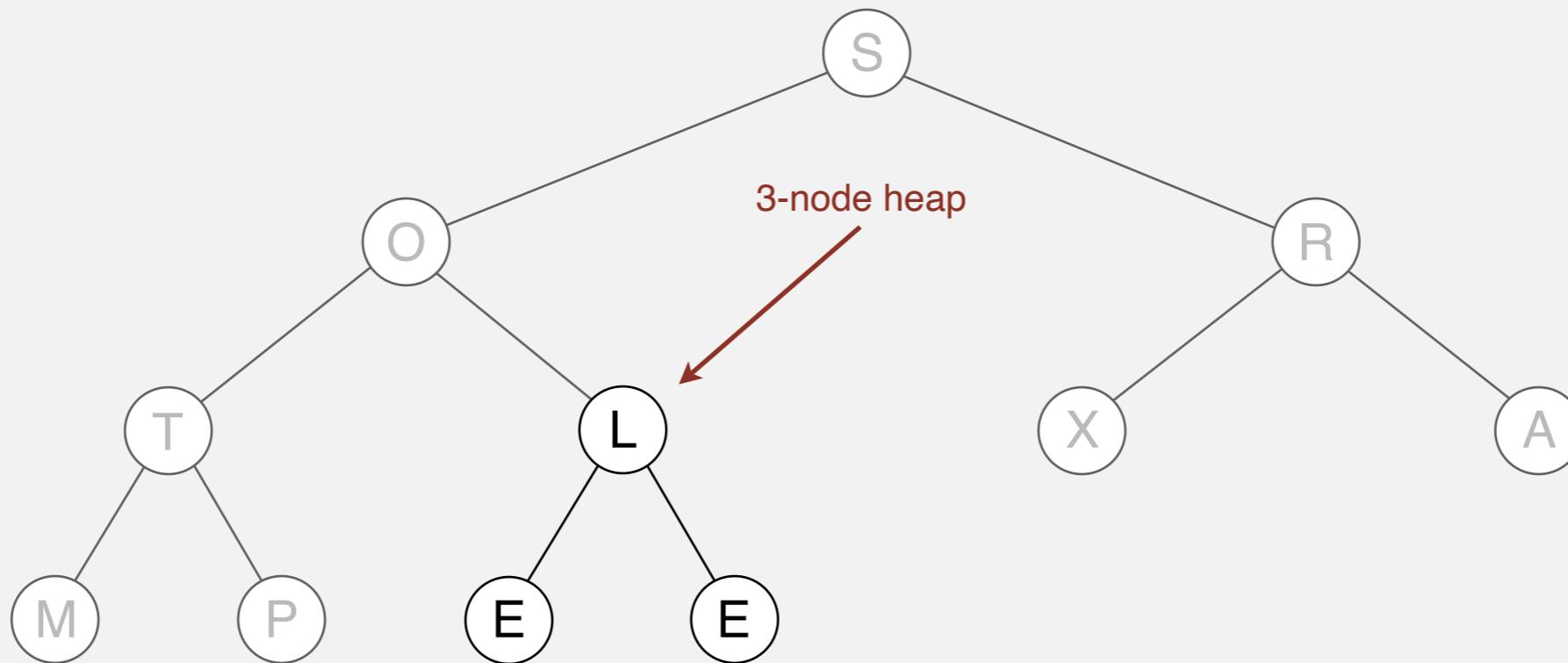


S O R T L X A M P E E
5 10

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 5

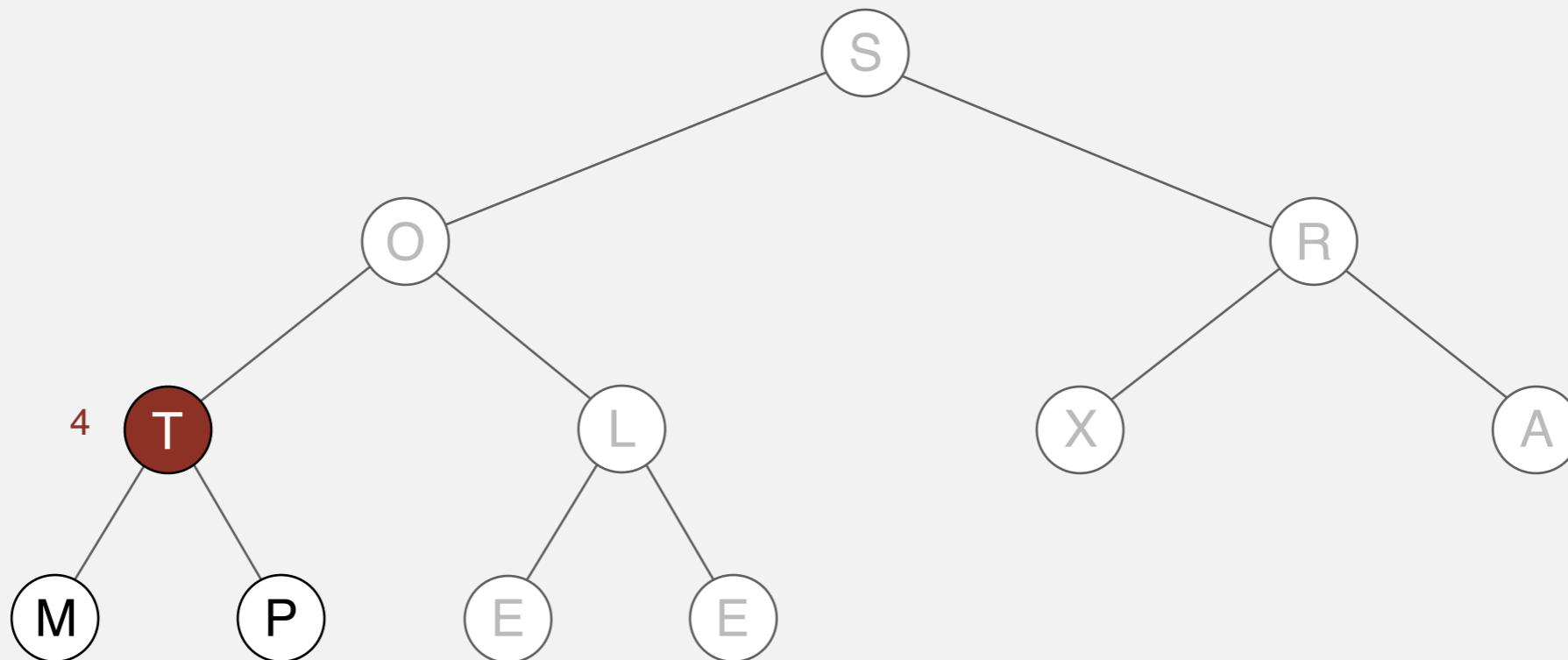


S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 4

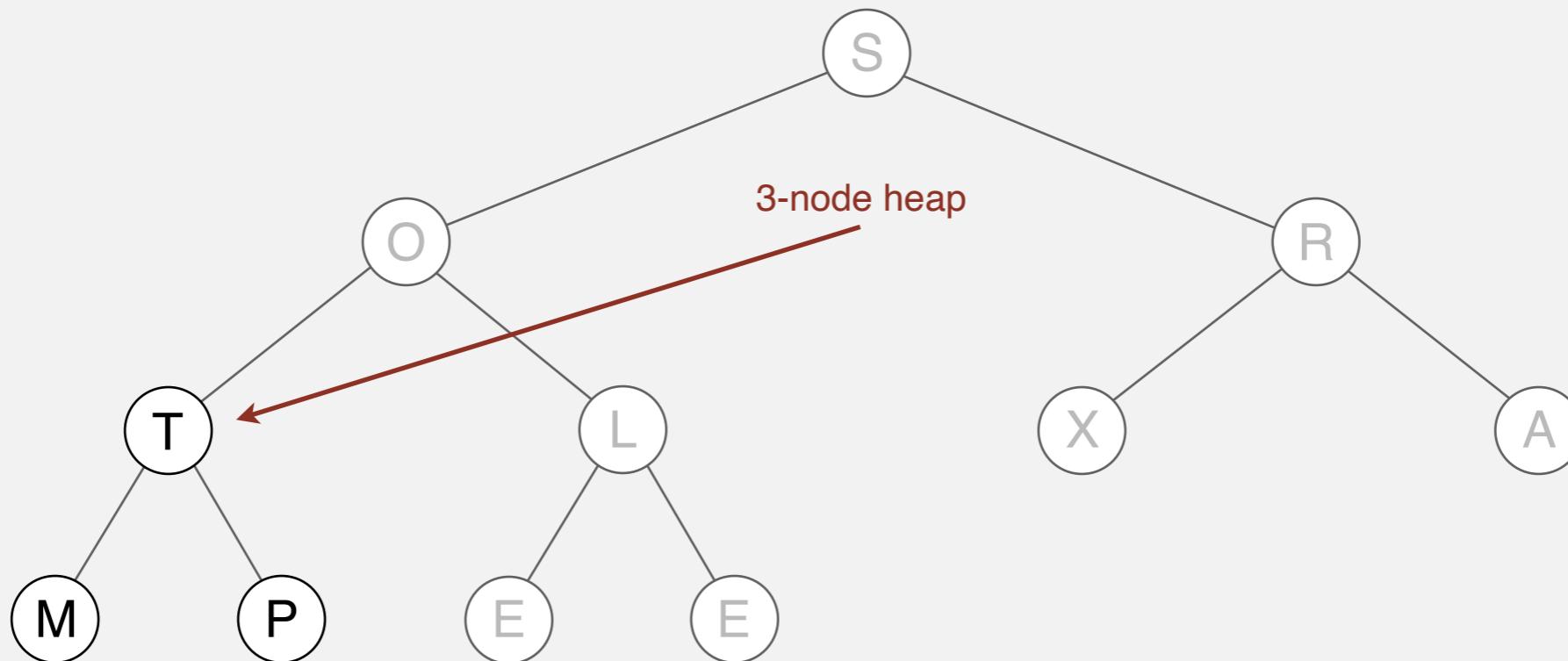


S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 4

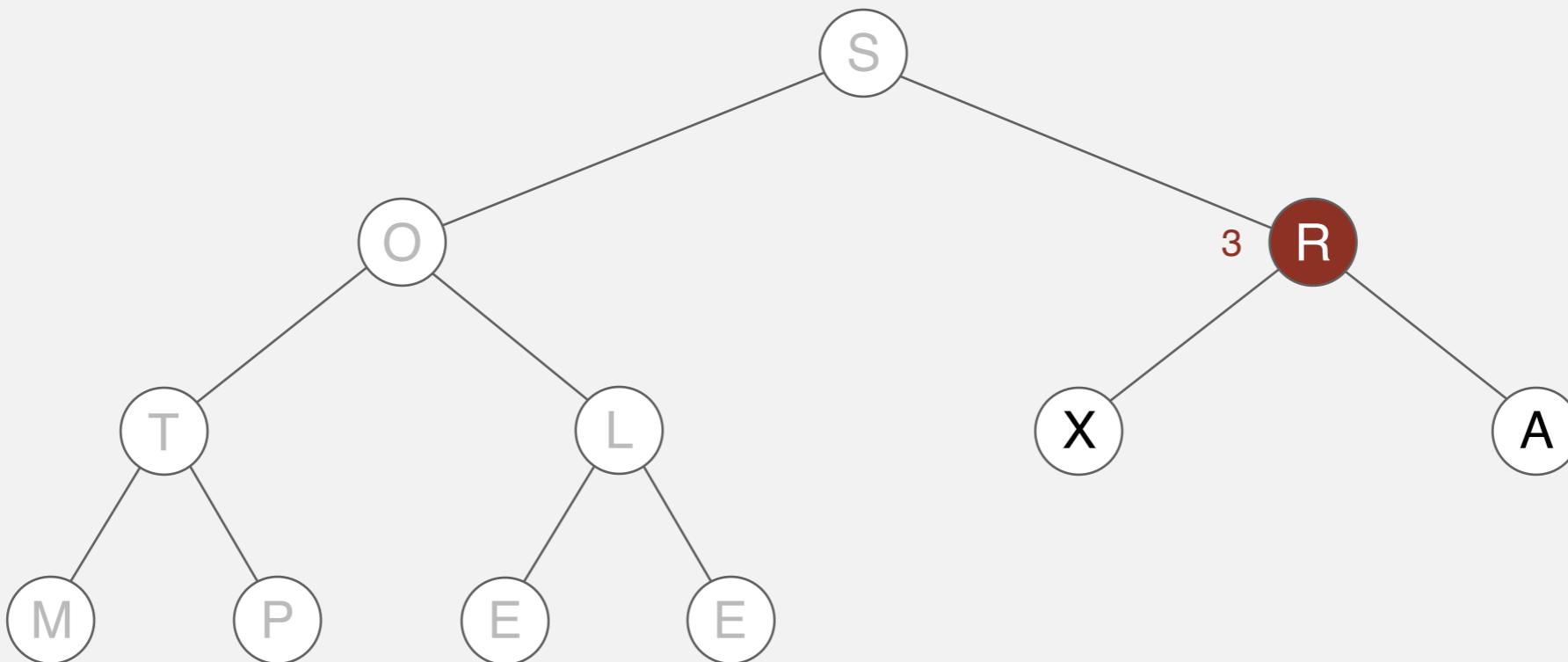


S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3

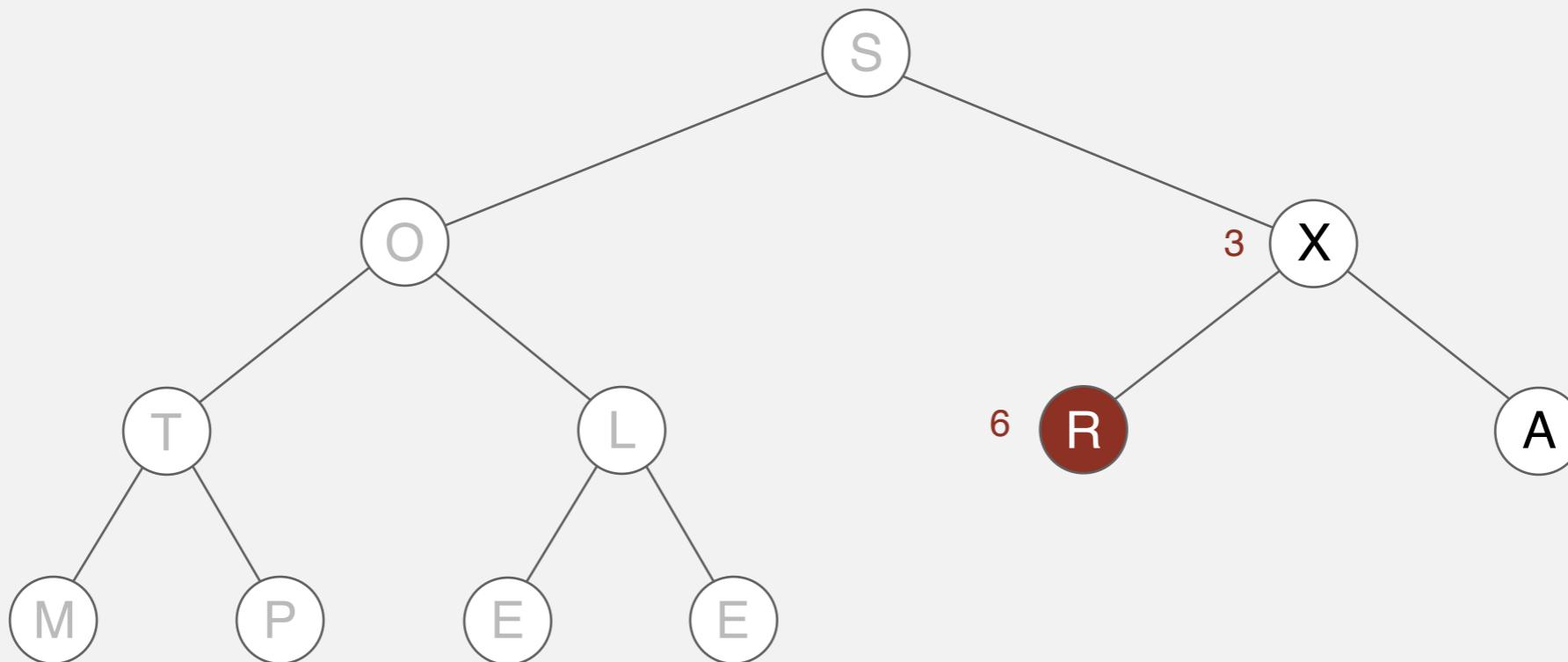


S O R T L X A M P E E
3

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3

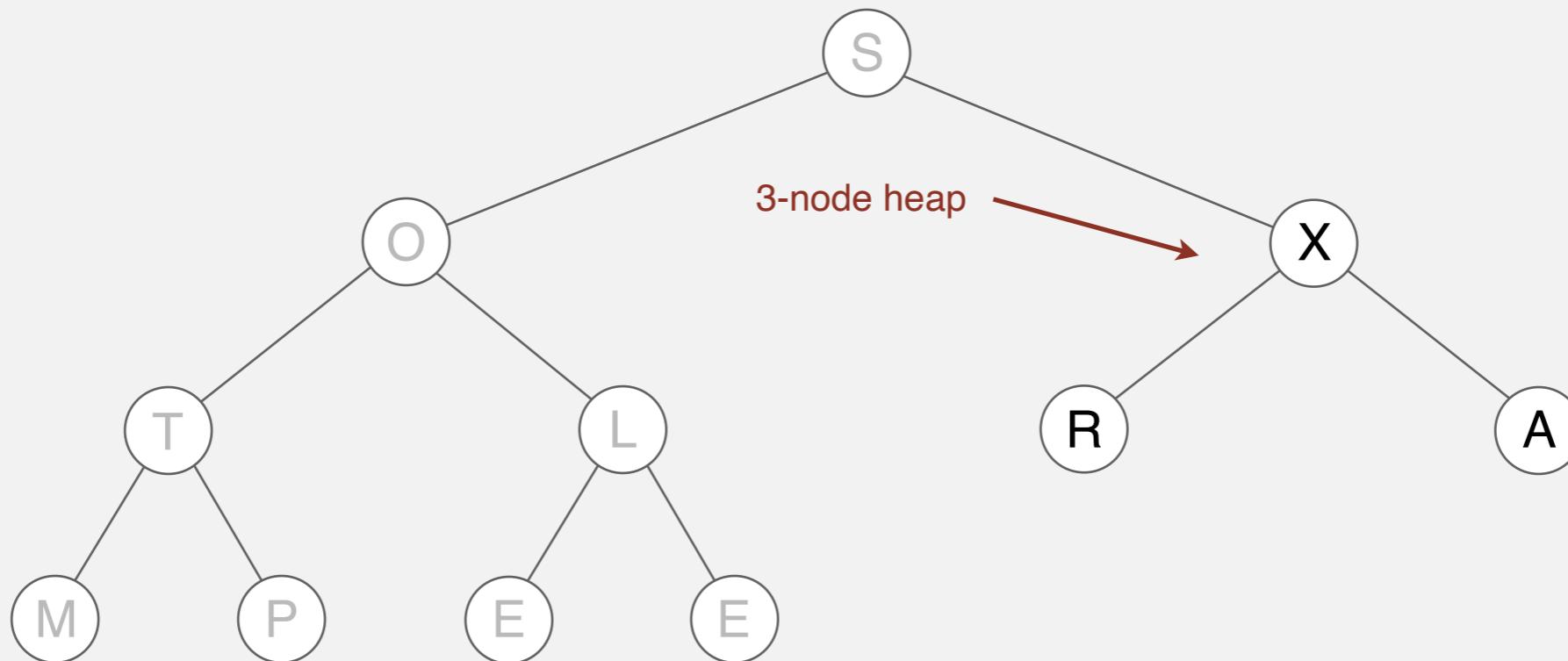


S	O	X	T	L	R	A	M	P	E	E
3					6					

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3

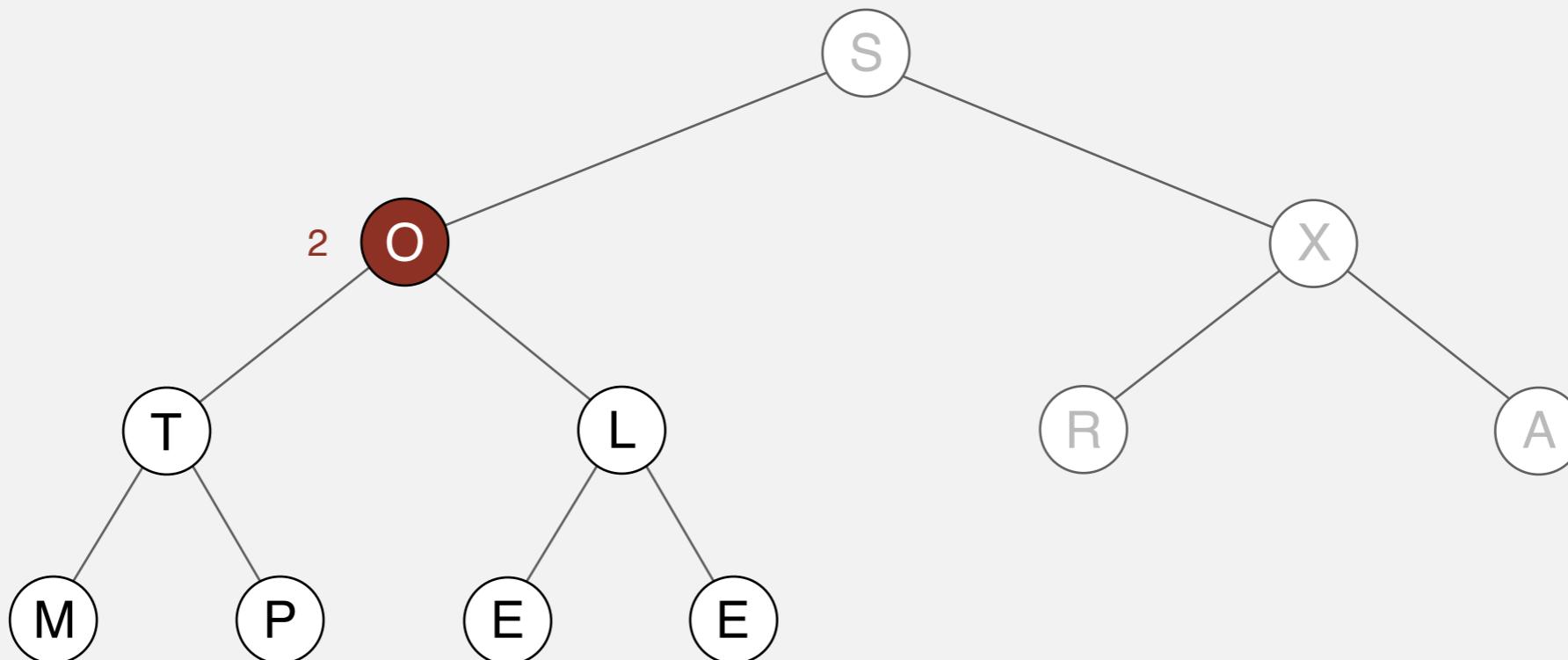


S	O	X	T	L	A	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 2

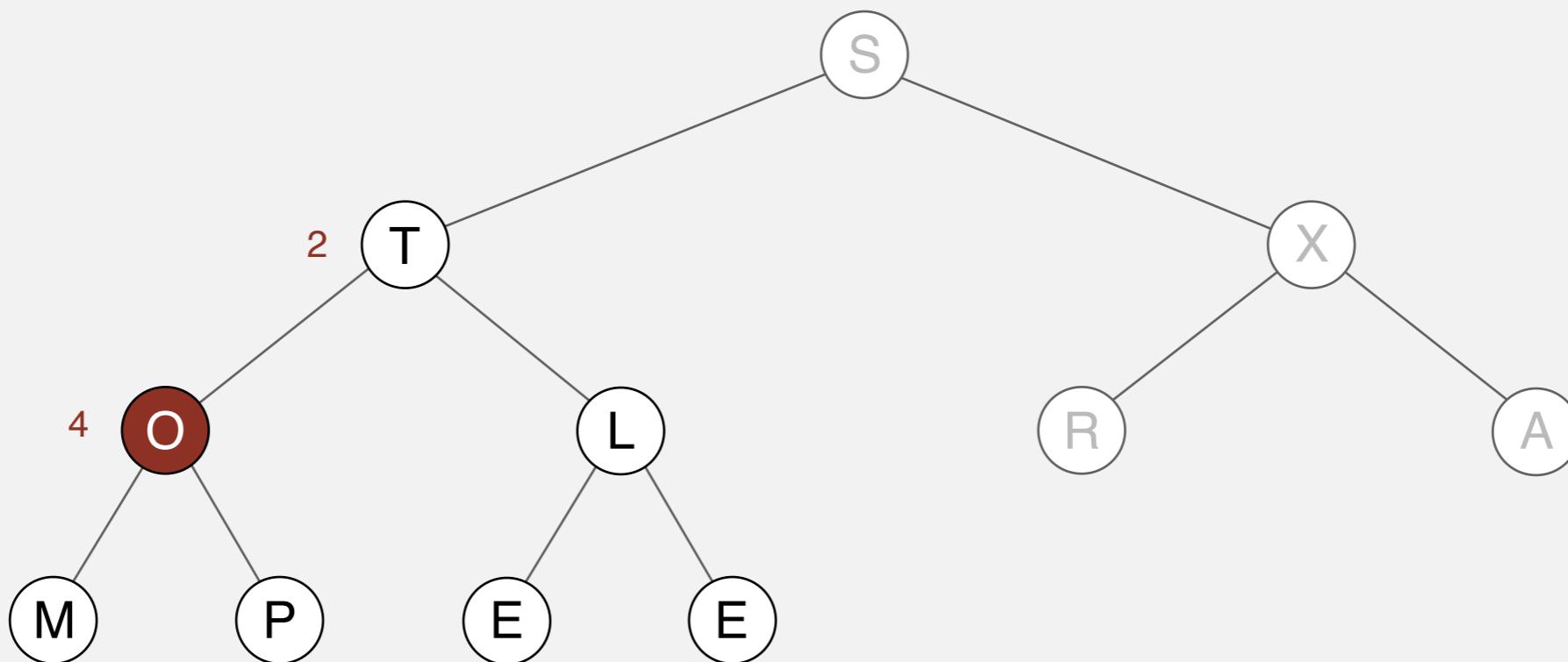


S	O	X	T	L	R	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 2

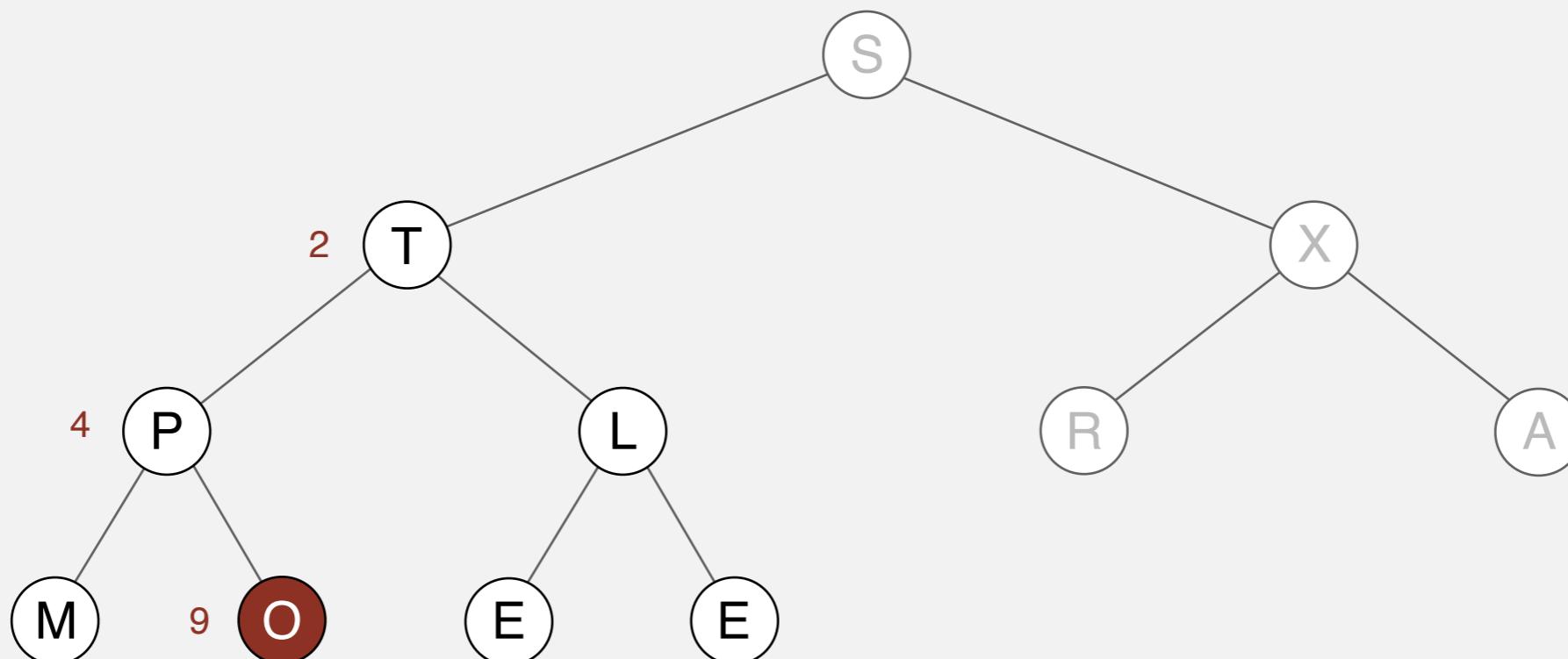


S T X O L R A M P E E
2 4

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 2

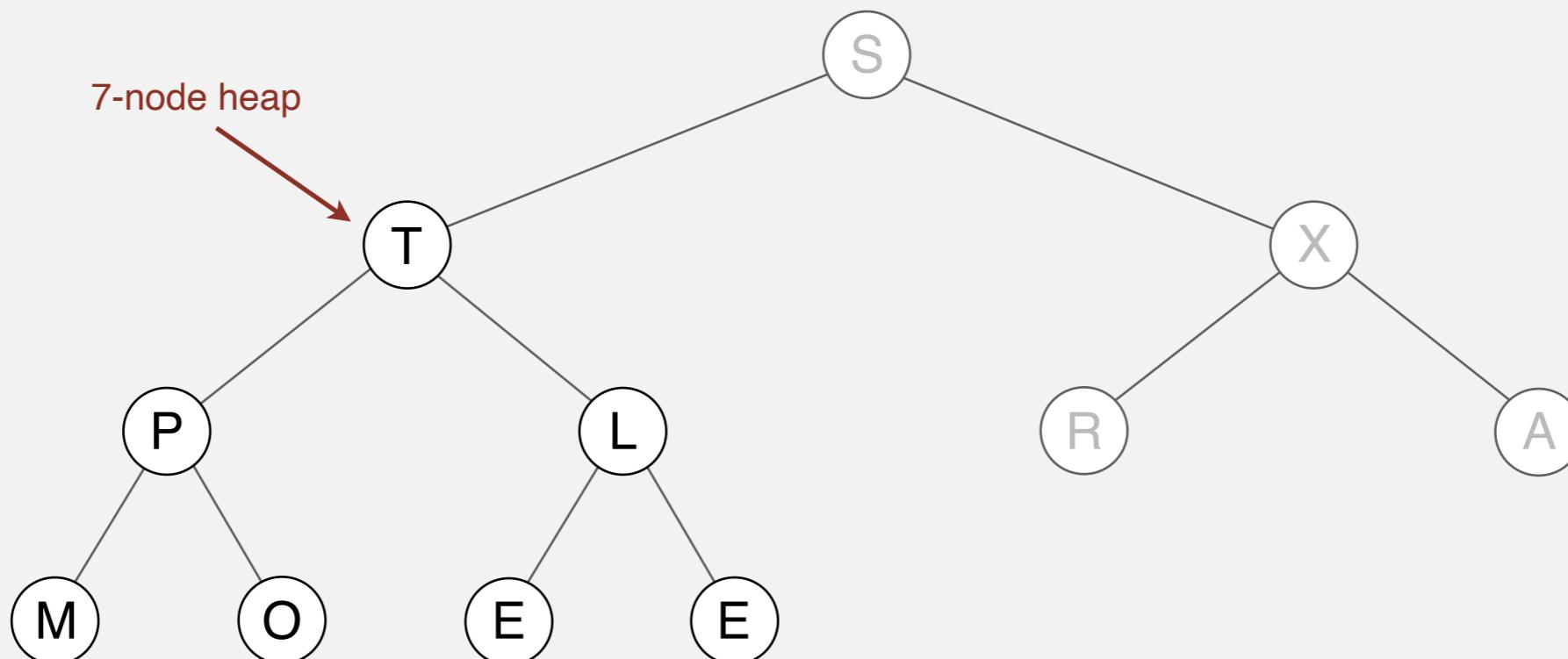


S	T	X	P	L	R	A	M	O	E	E
2	4			9						

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 2

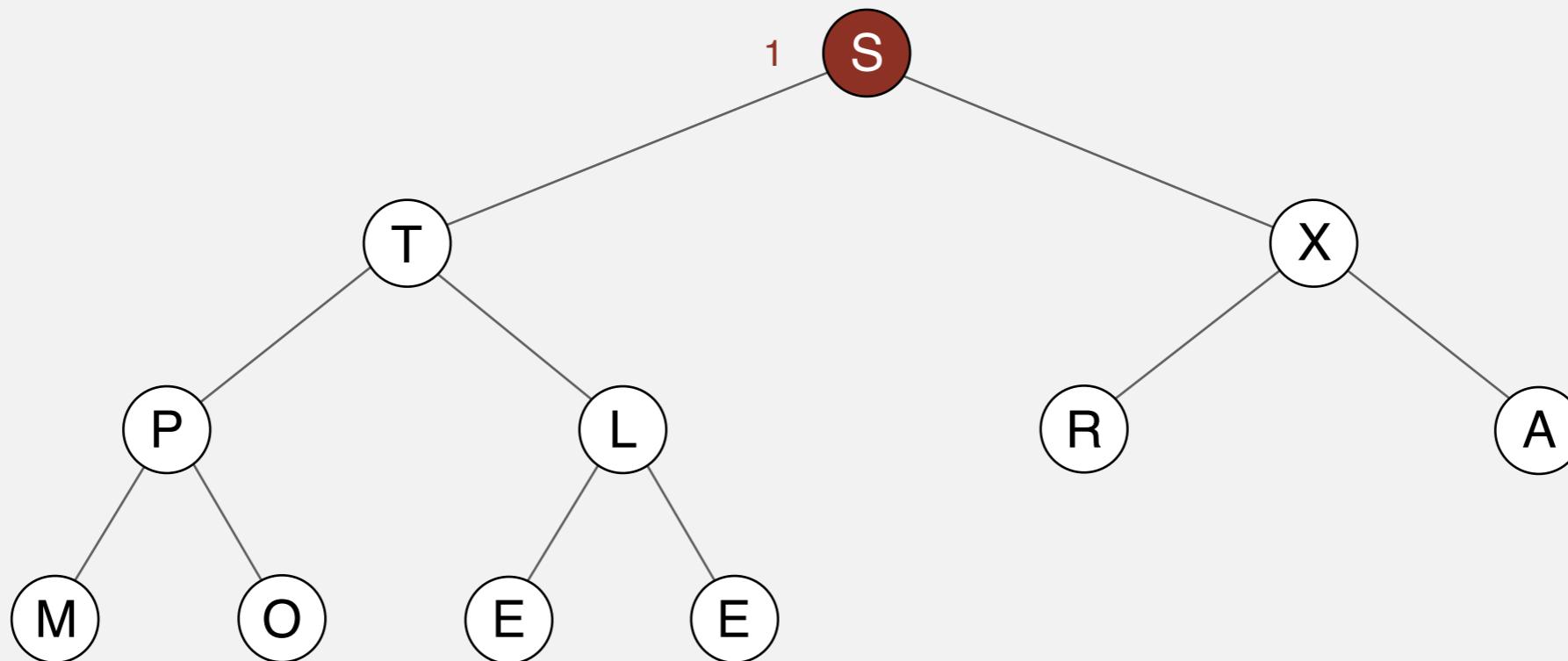


S	T	X	P	L	R	A	M	O	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 1

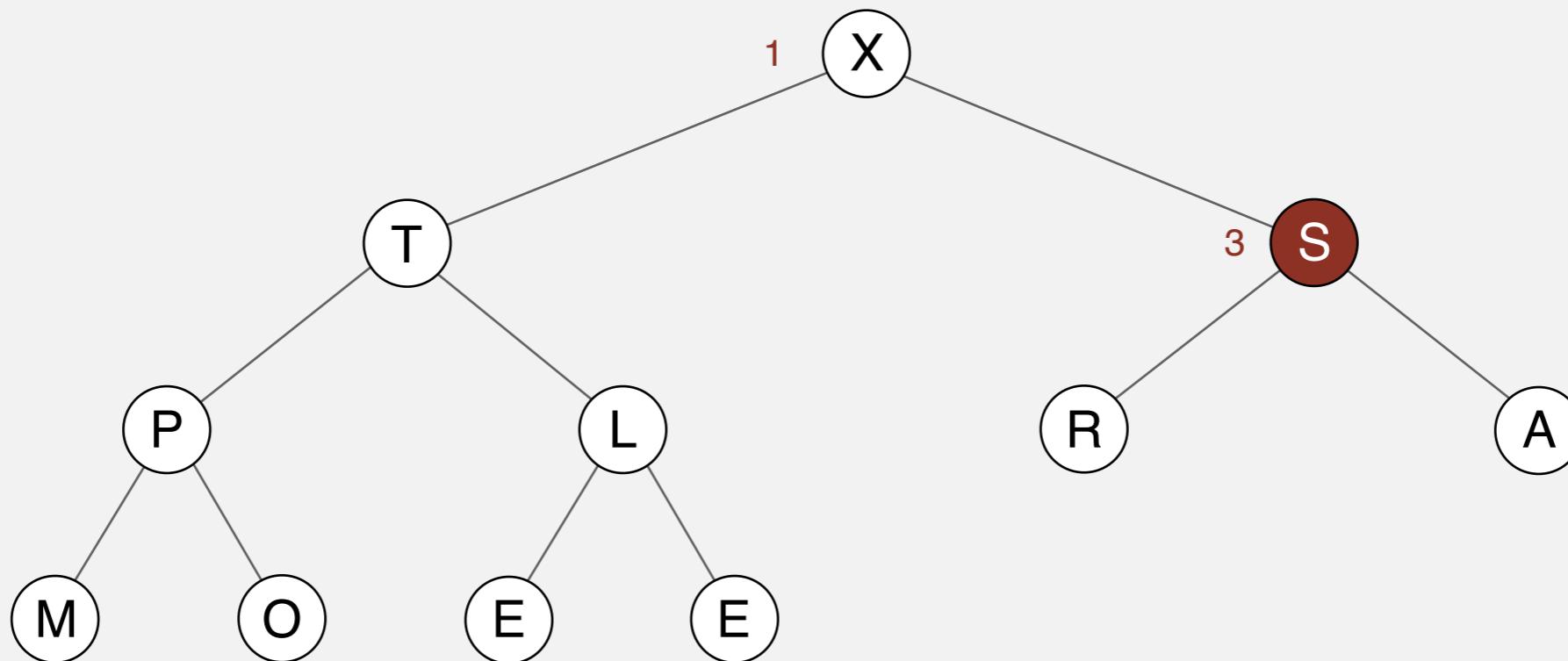


S	T	X	P	L	R	A	M	O	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Heap construction. Build max heap using bottom-up method.

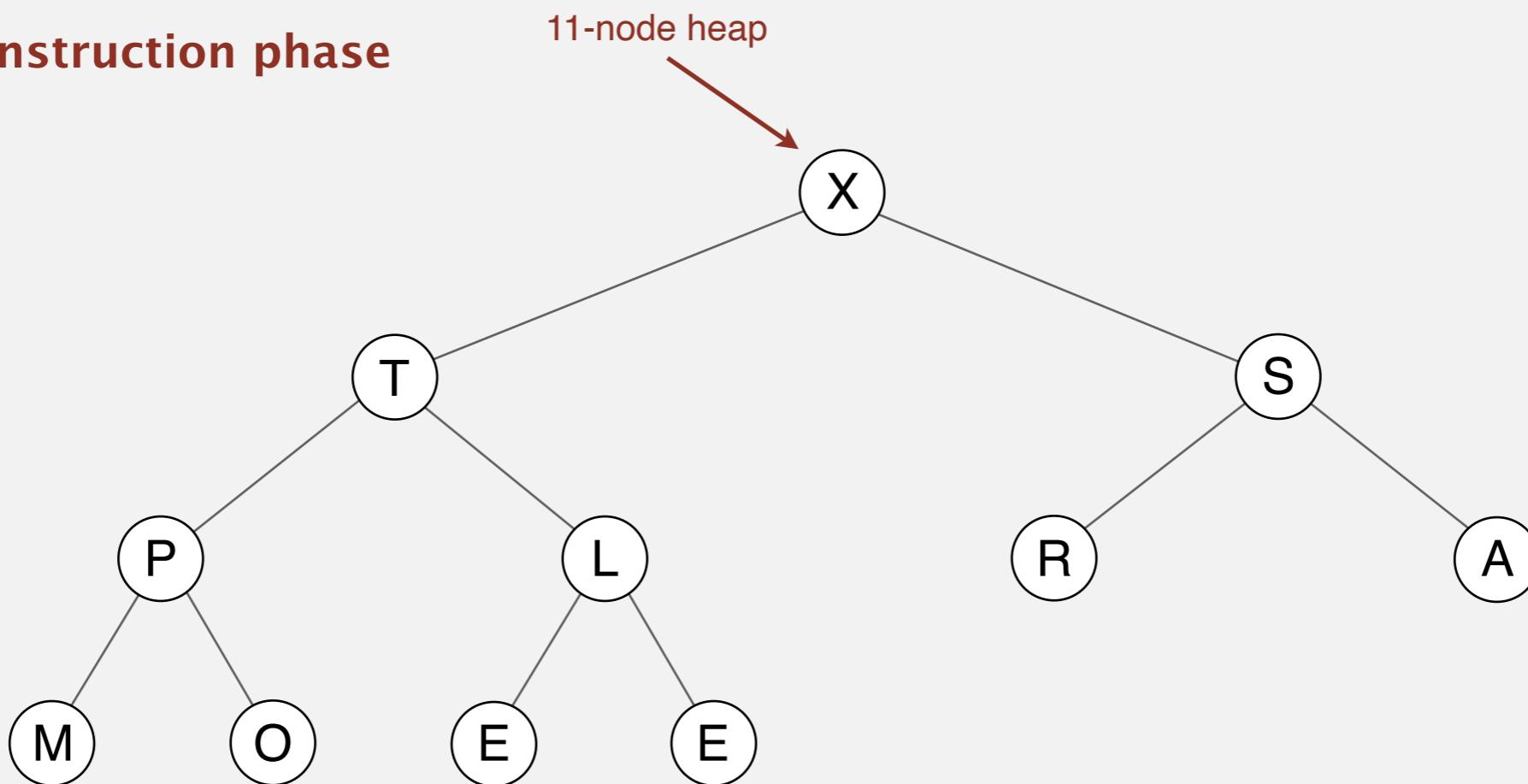
sink 1



Heapsort demo

Heap construction. Build max heap using bottom-up method.

end of construction phase

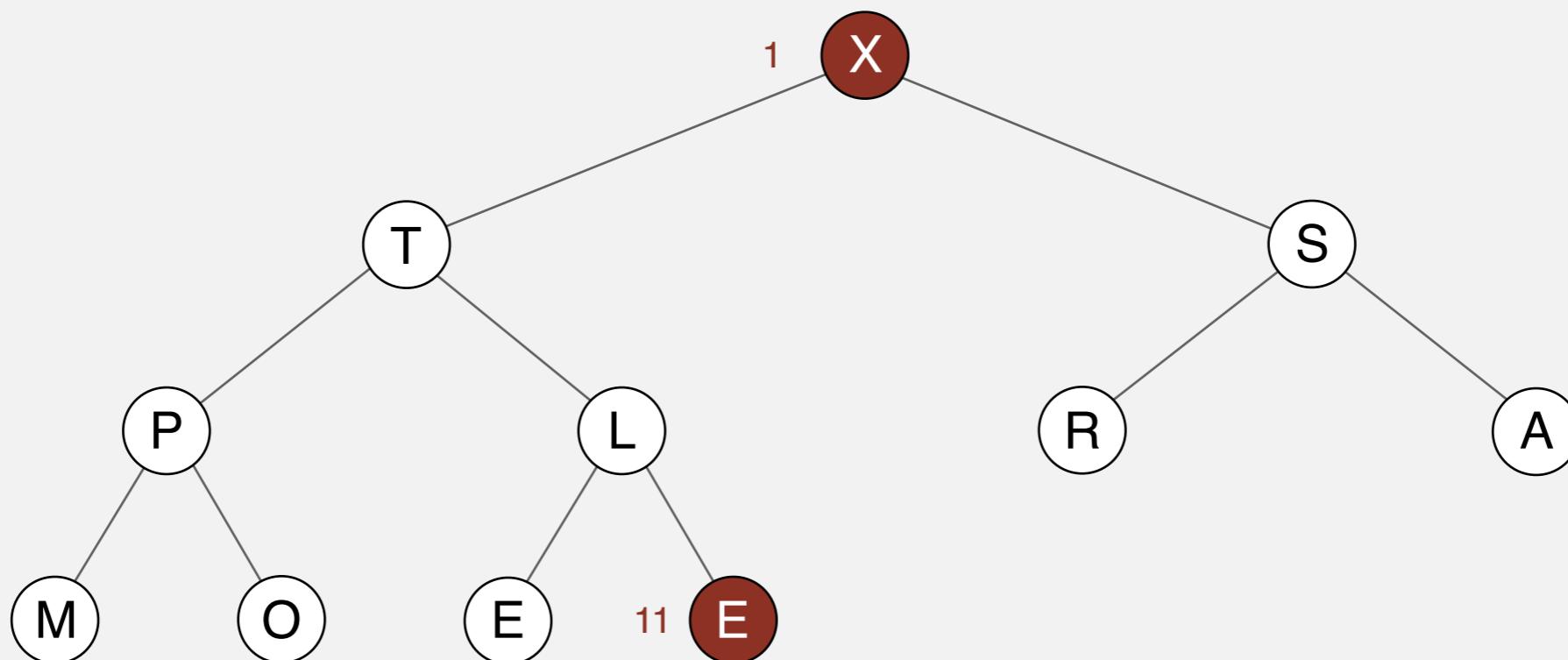


X	T	S	P	L	R	A	M	O	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

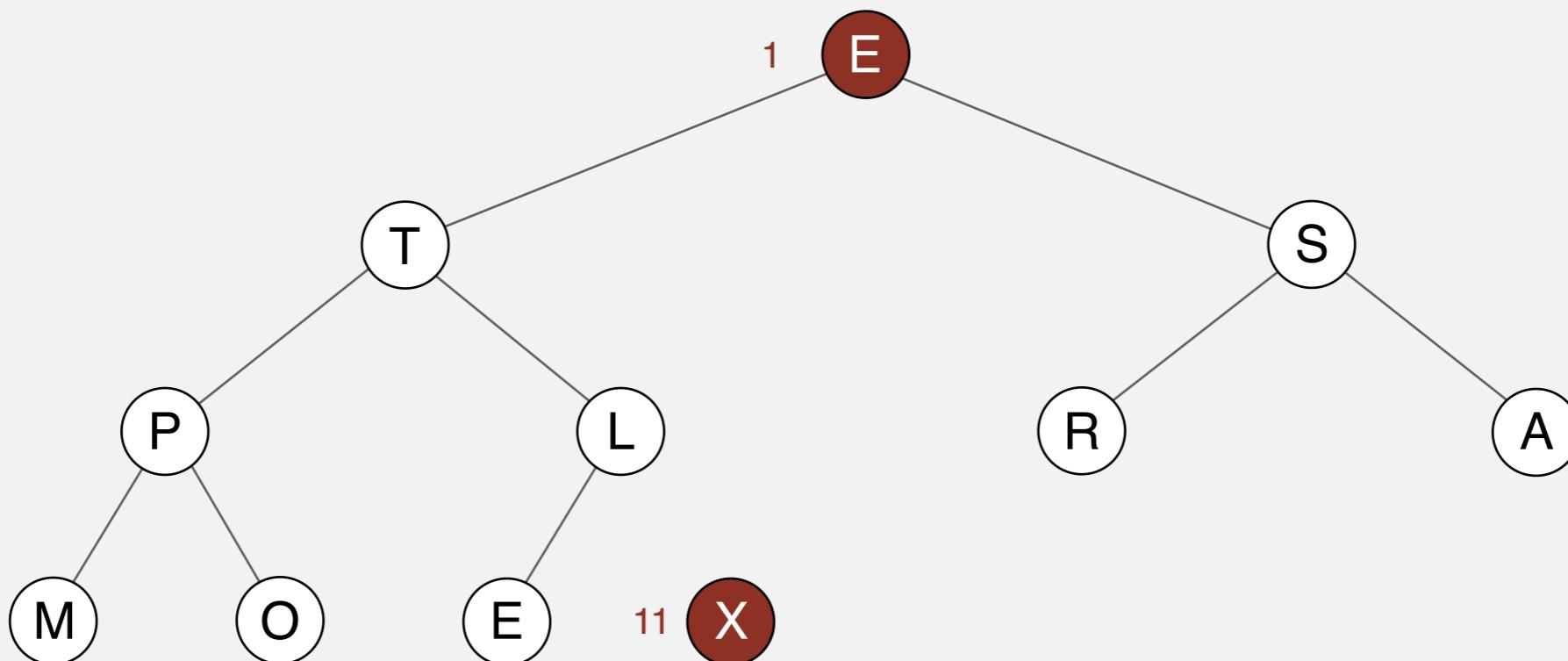


X T S P L R A M O E E

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

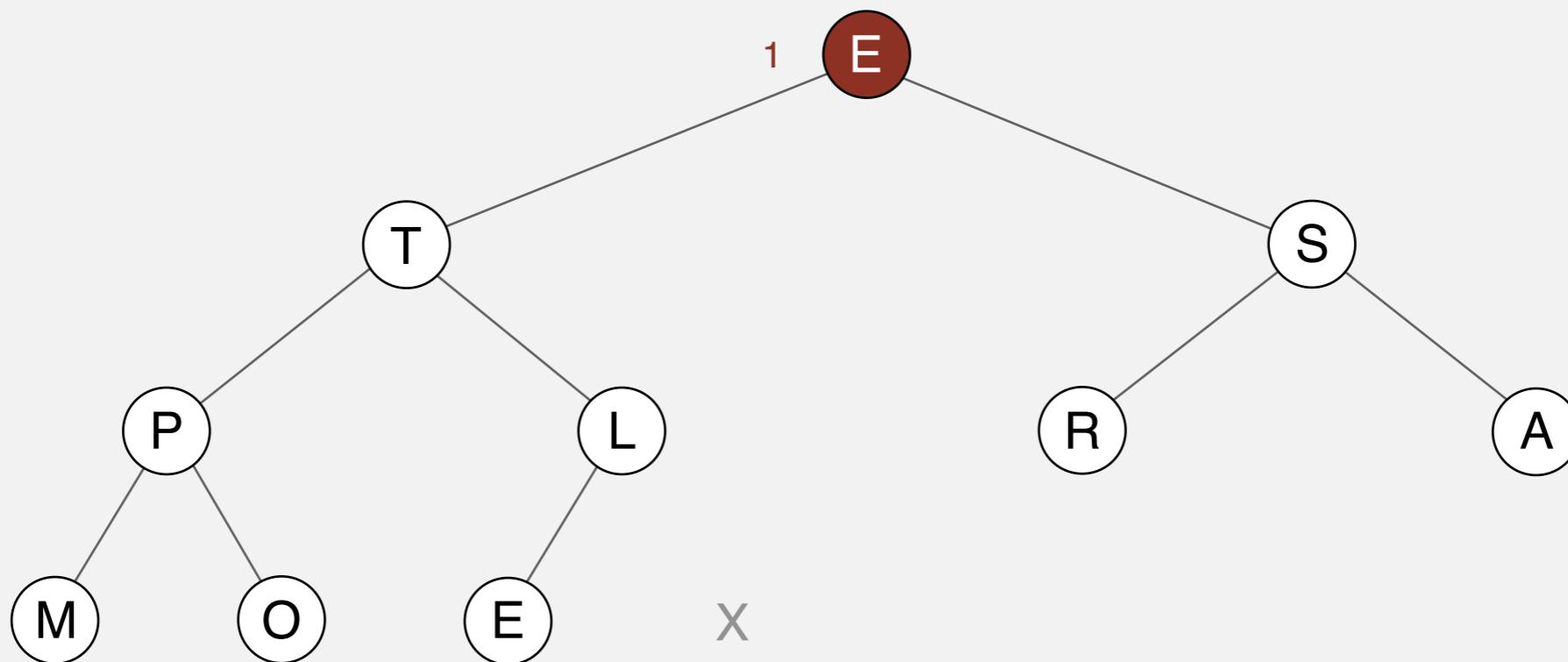


E T S P L R A M O E X
1 11

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

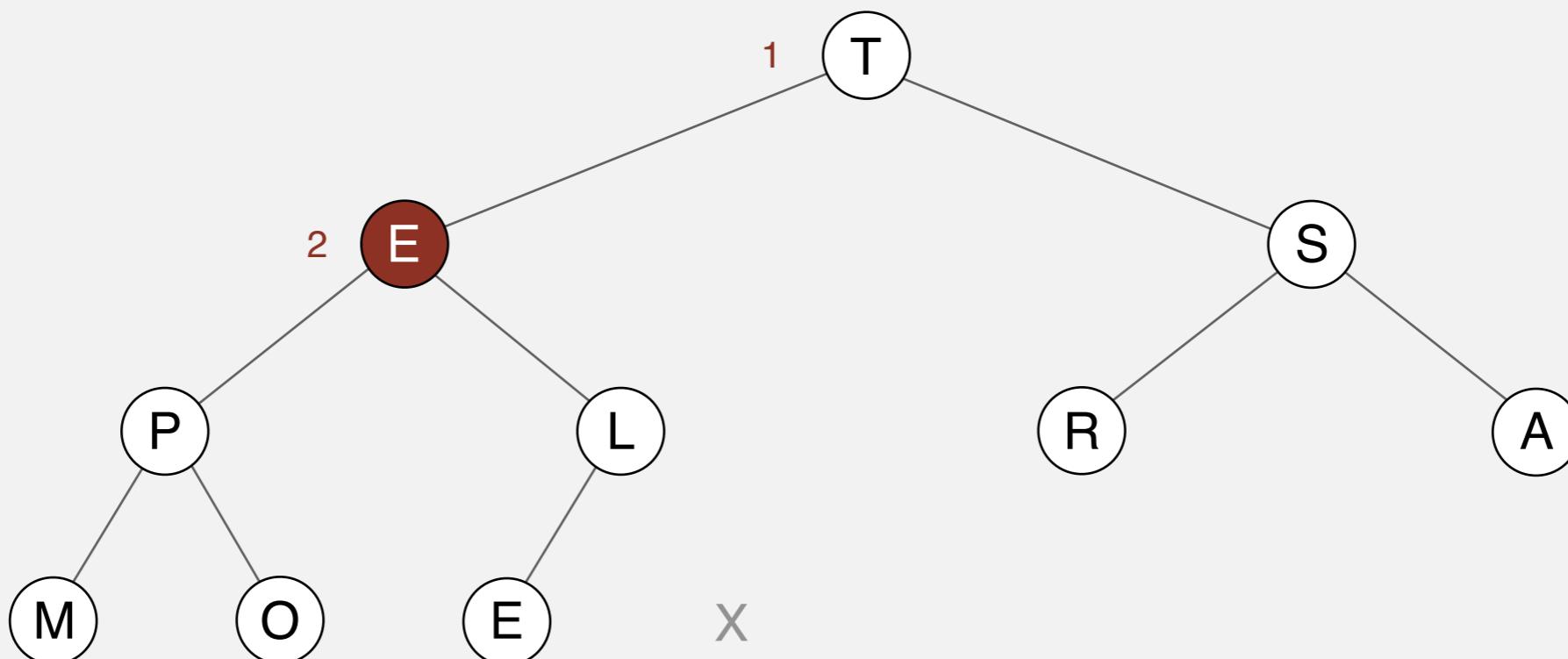


E	T	S	P	L	R	A	M	O	E	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

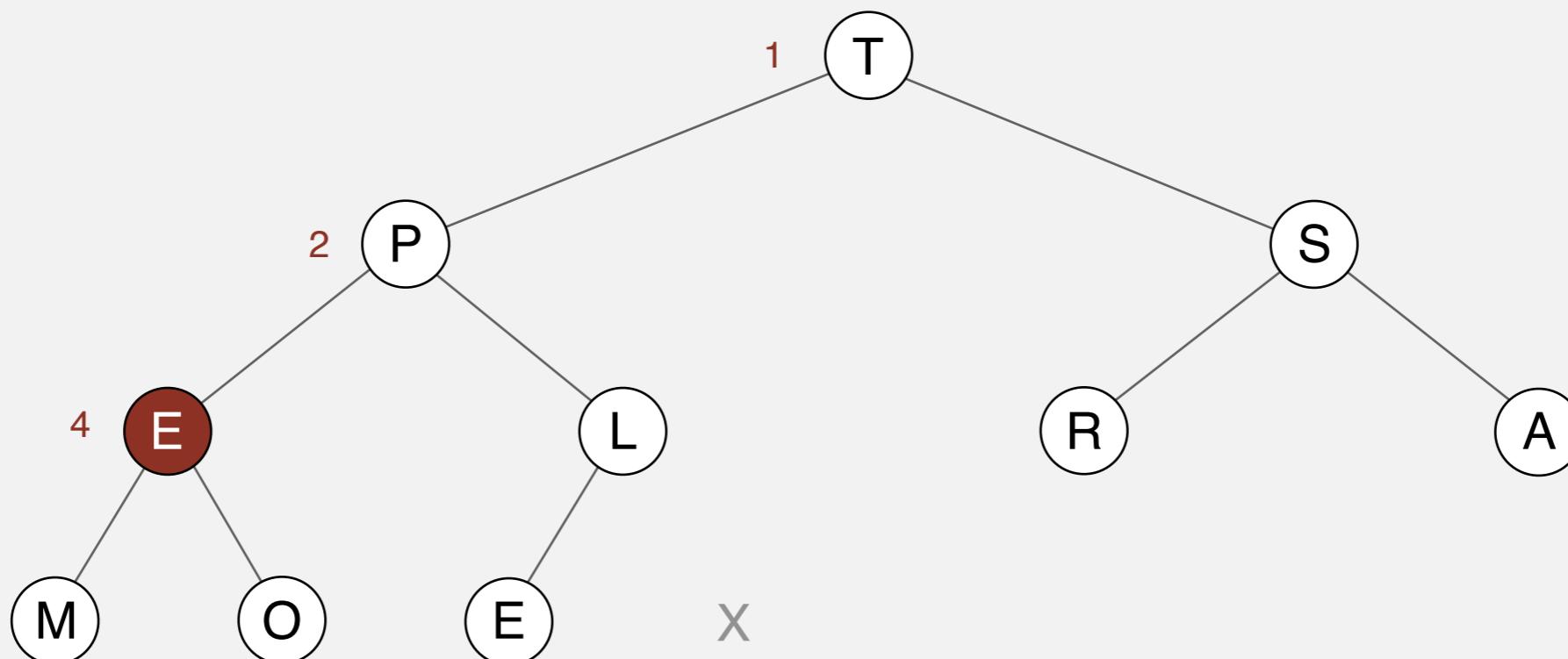


T E S P L R A M O E X
1 2

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

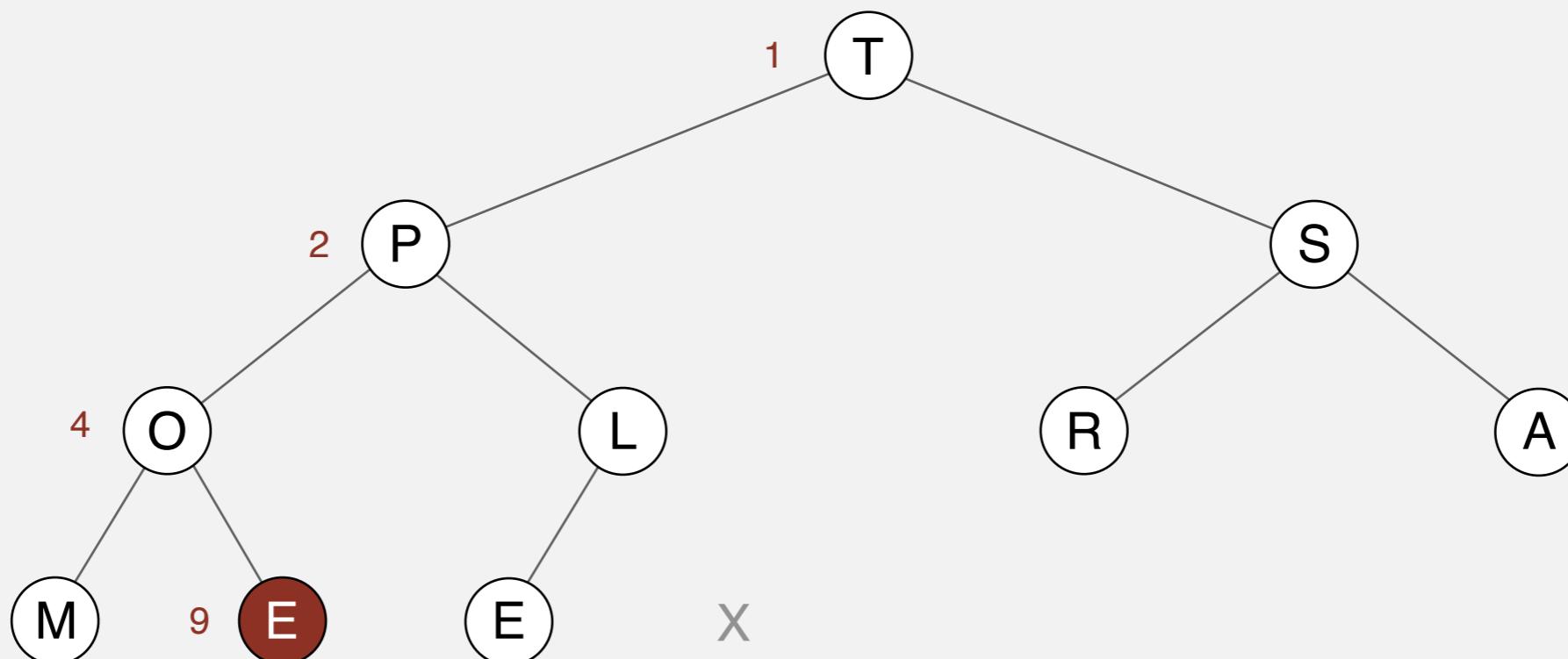


T	P	S	E	L	R	A	M	O	E	X
1	2		4							

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

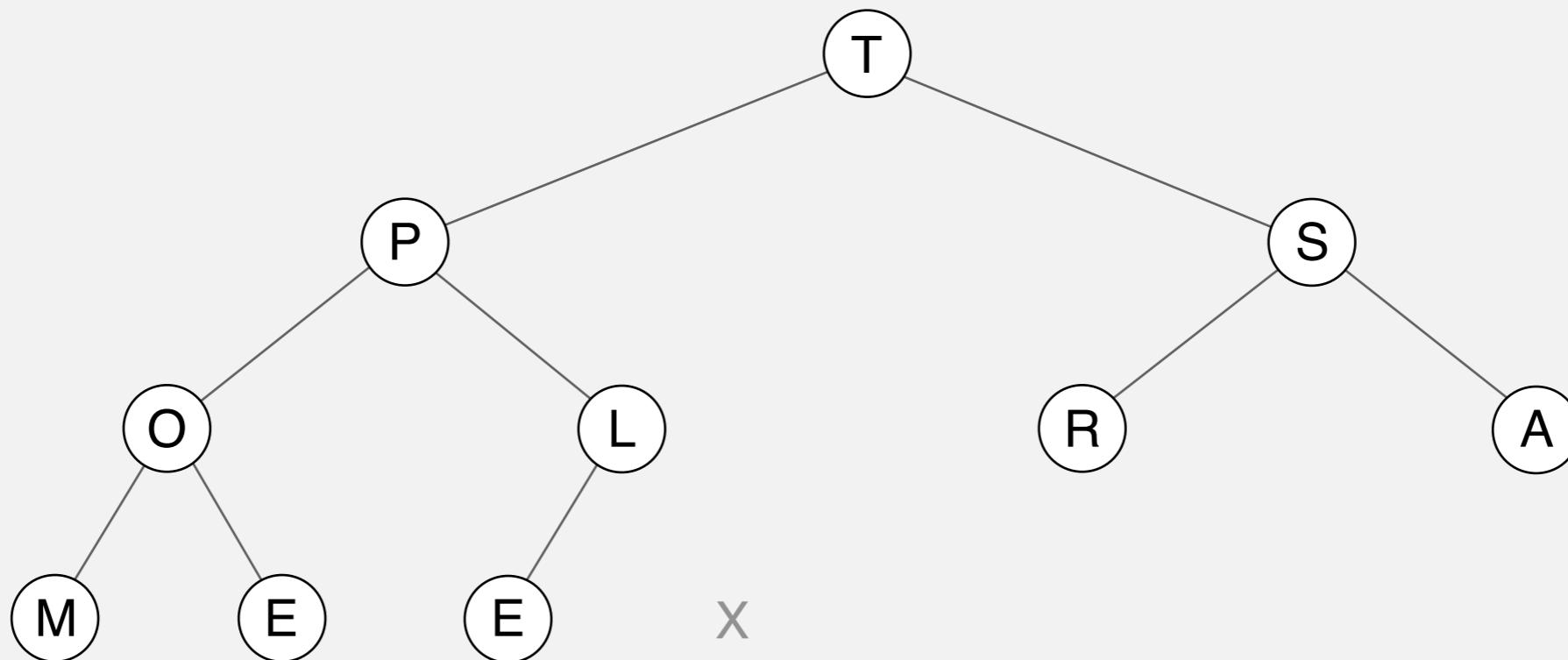
sink 1



T	P	S	O	L	R	A	M	E	E	X
1	2		4					9		

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

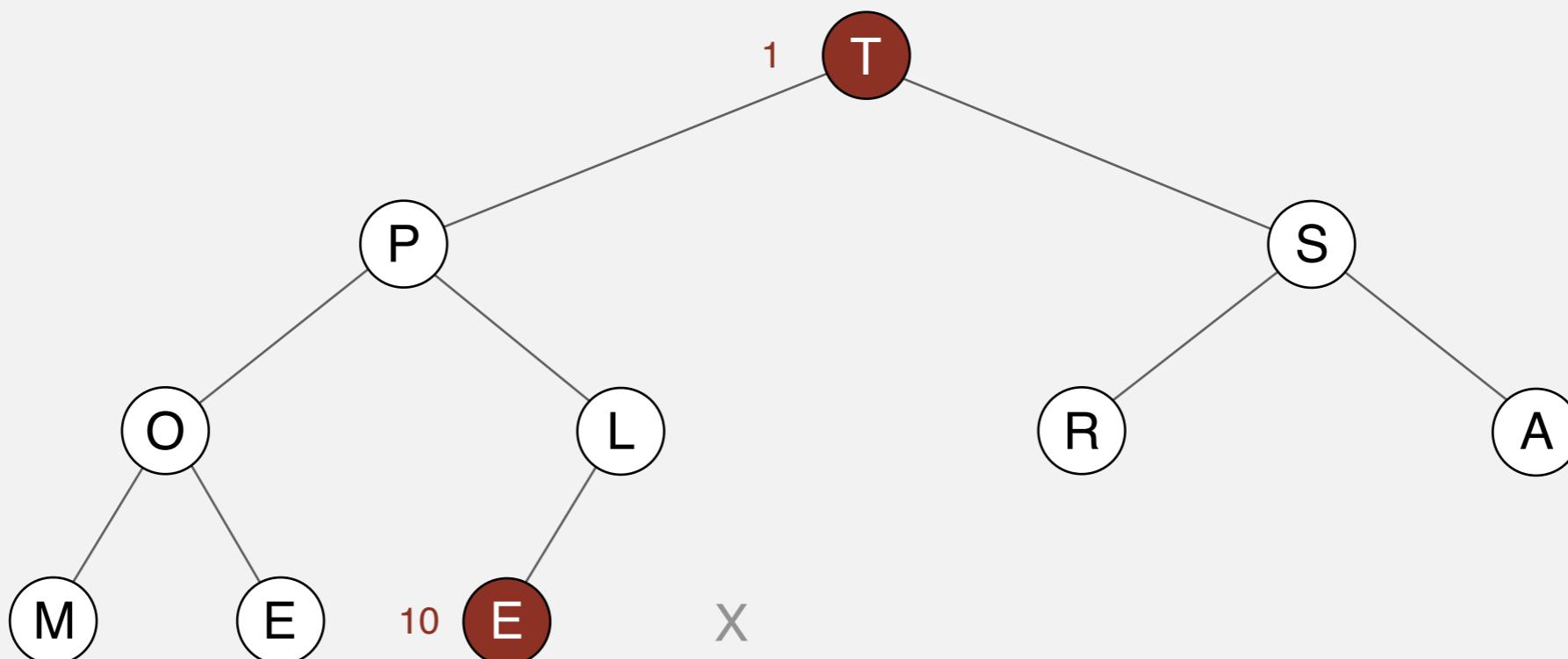


T	P	S	O	L	R	A	M	E	E	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 10



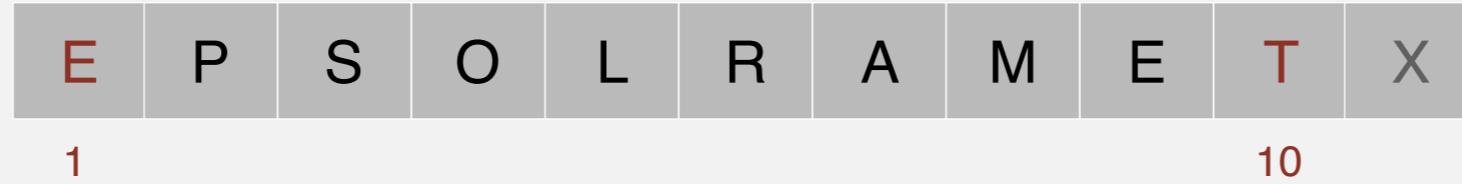
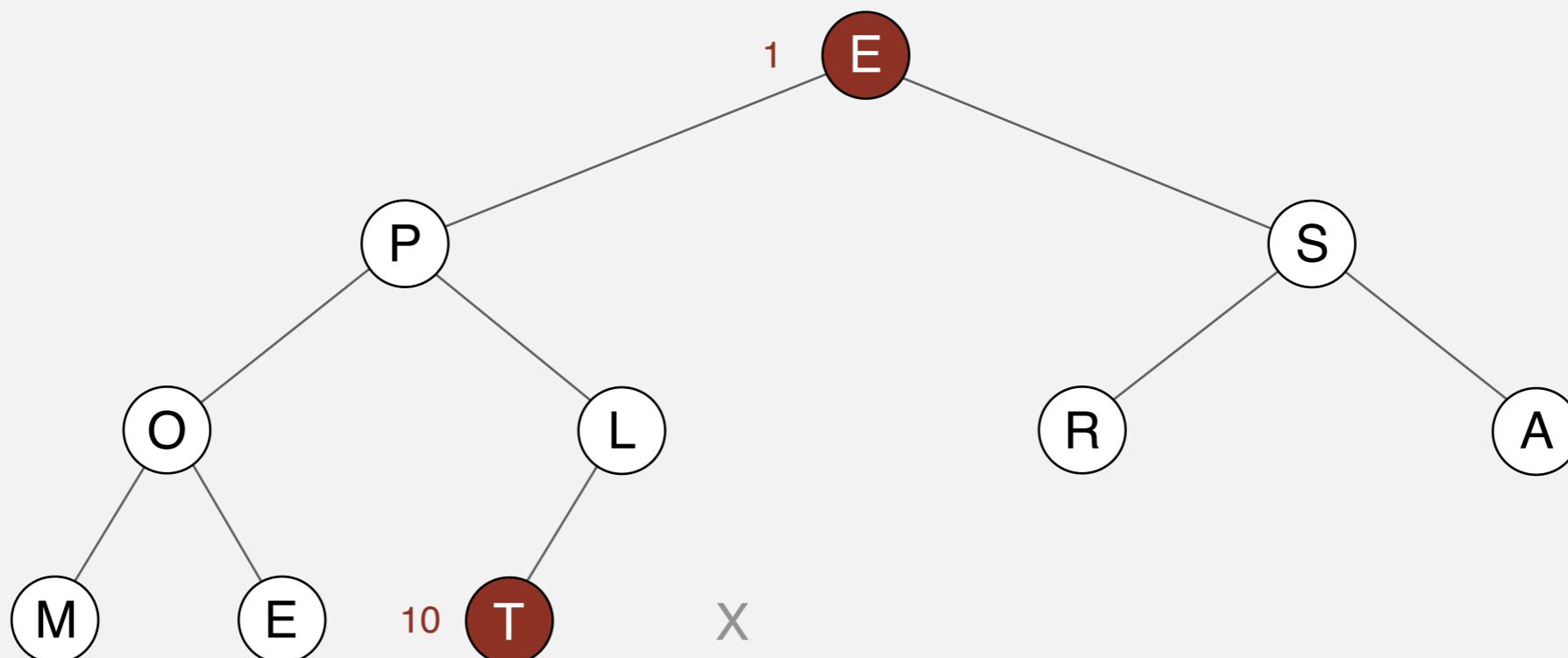
T P S O L R A M E E X

1 10

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

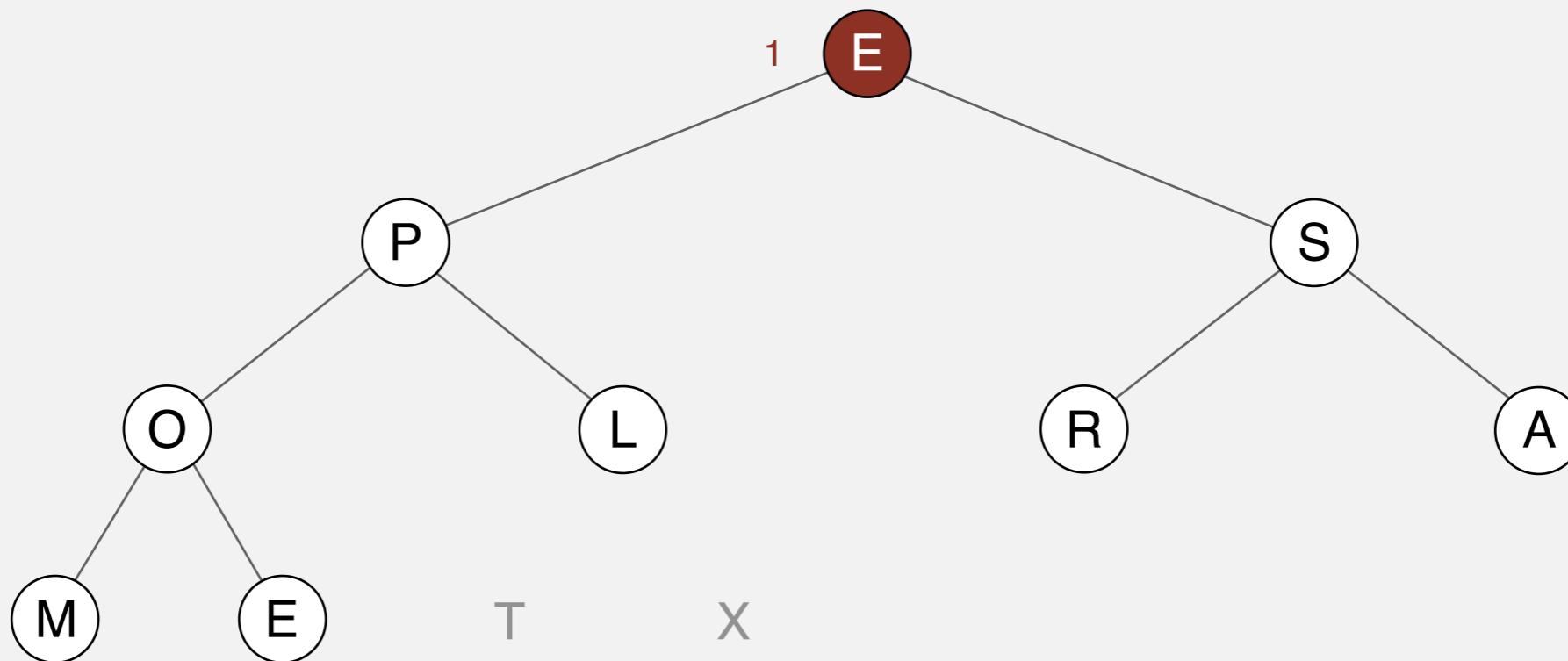
exchange 1 and 10



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

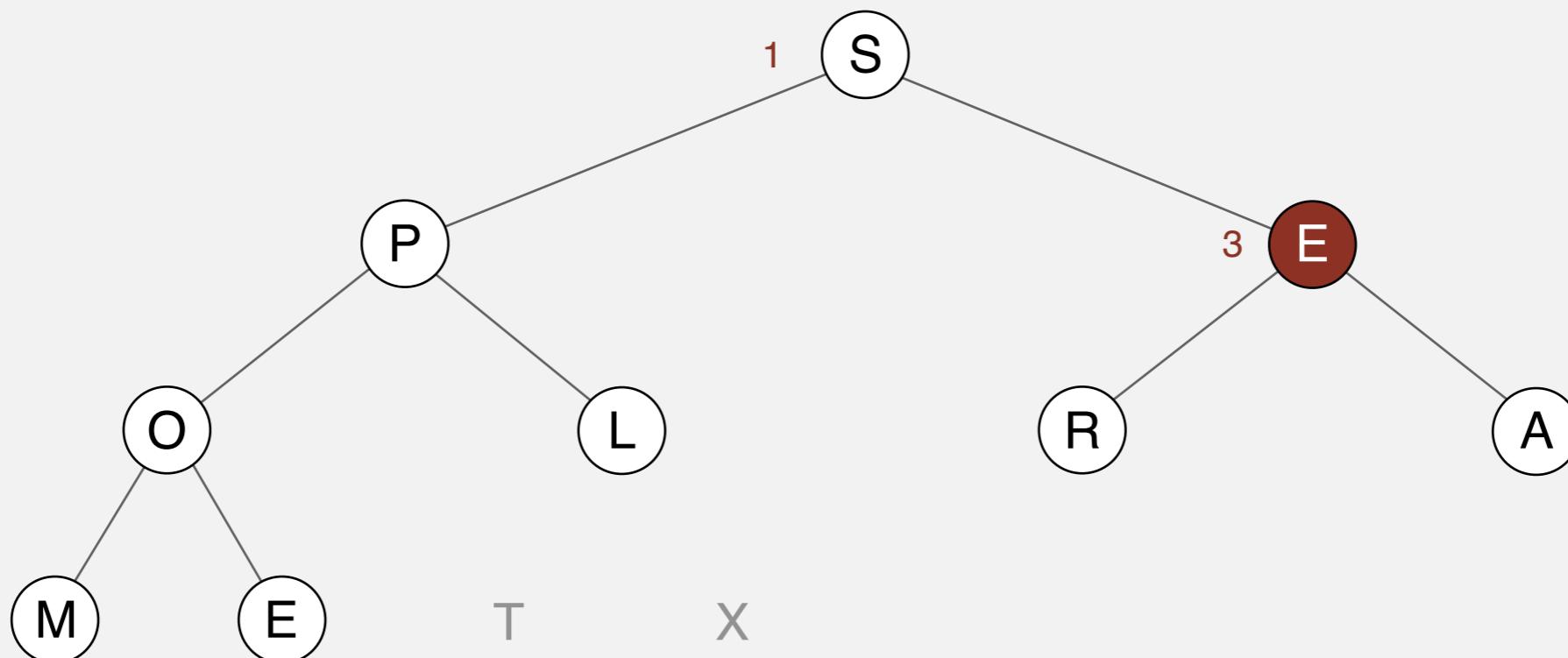


E	P	S	O	L	R	A	M	E	T	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

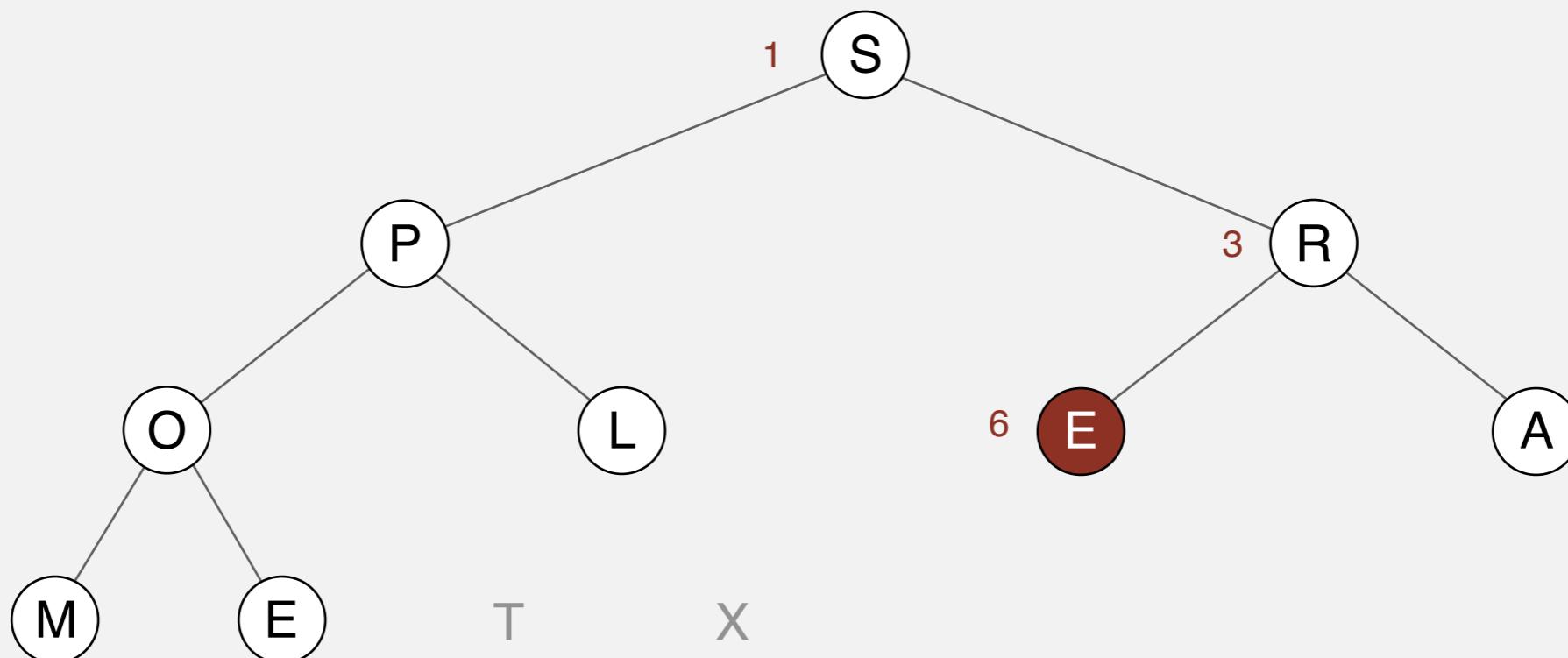


S	P	E	O	L	R	A	M	E	T	X
1		3								

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

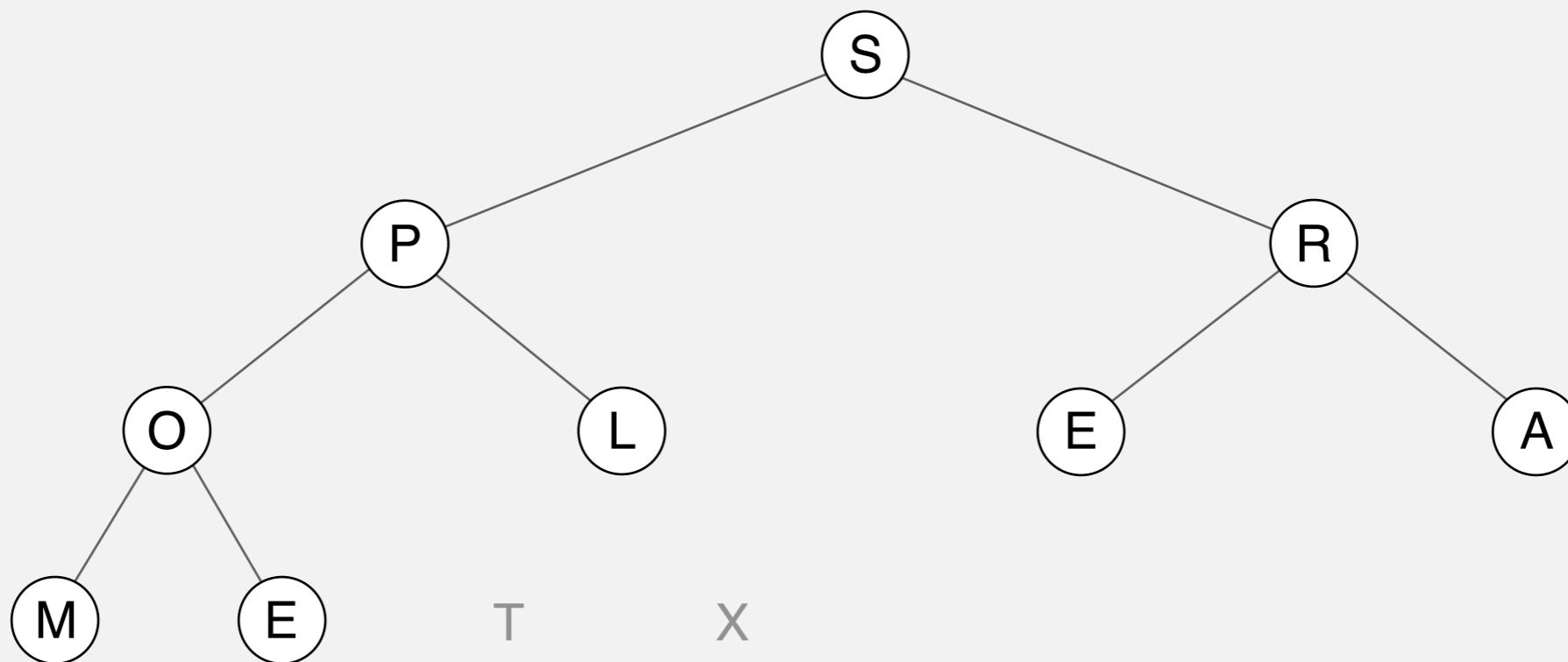
sink 1



S	P	R	O	L	E	A	M	E	T	X
1	3	3	T	X	6	T	X			

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

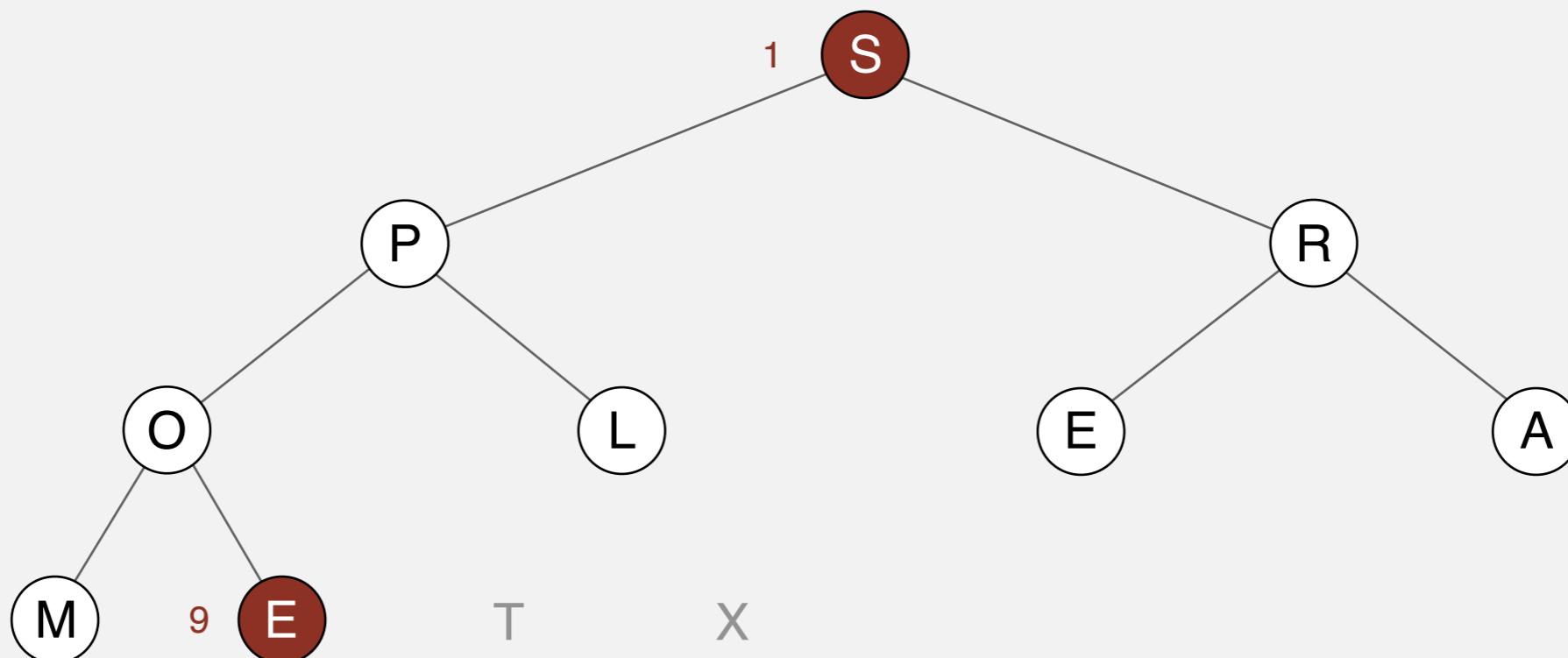


S	P	R	O	L	E	A	M	E	T	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

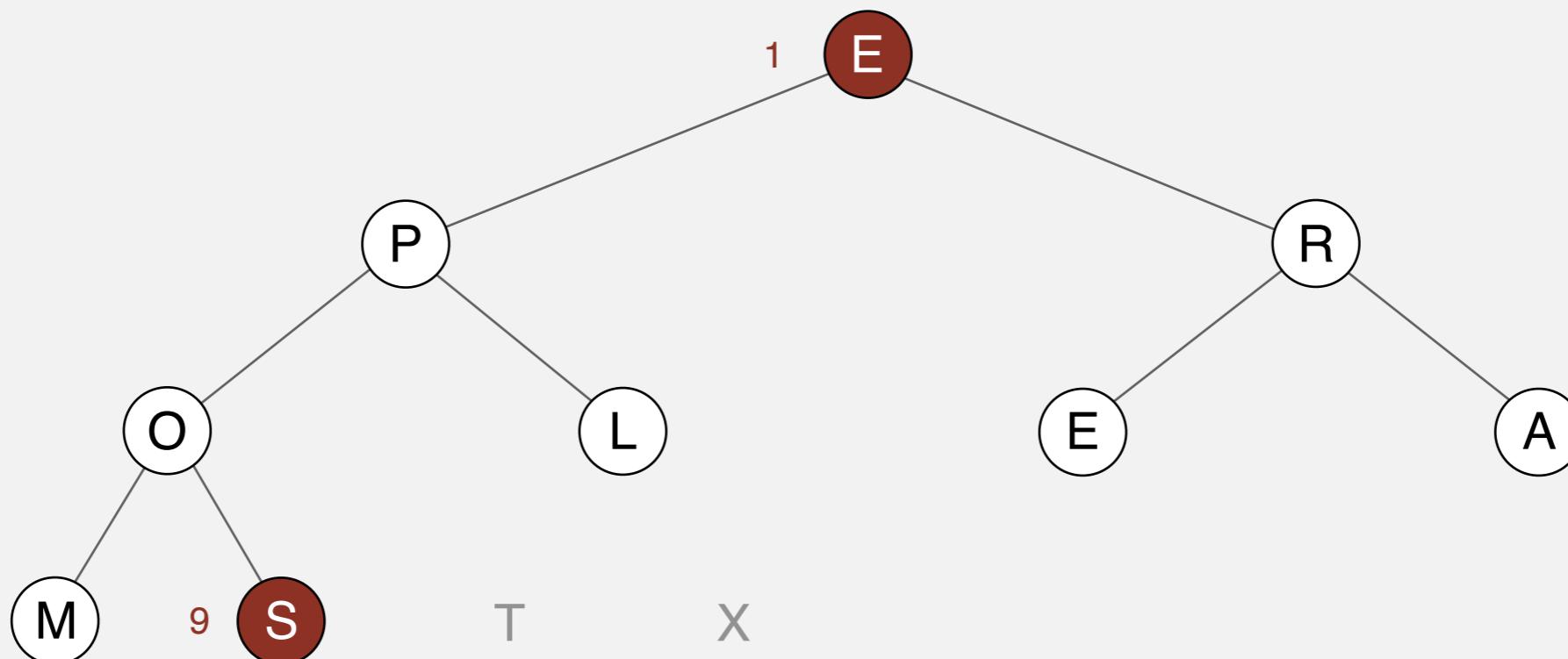


S	P	R	O	L	E	A	M	E	T	X
1								9		

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

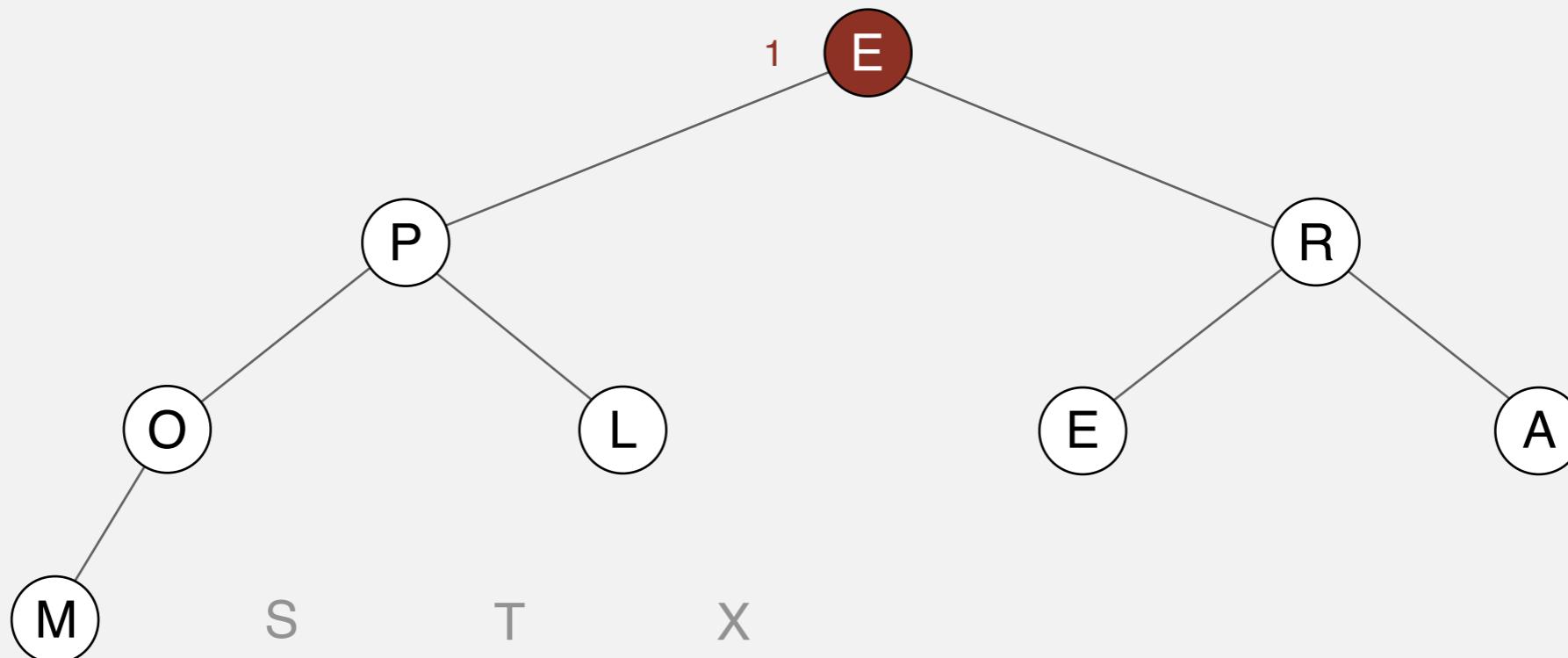


E	P	R	O	L	E	A	M	S	T	X
1								9		

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

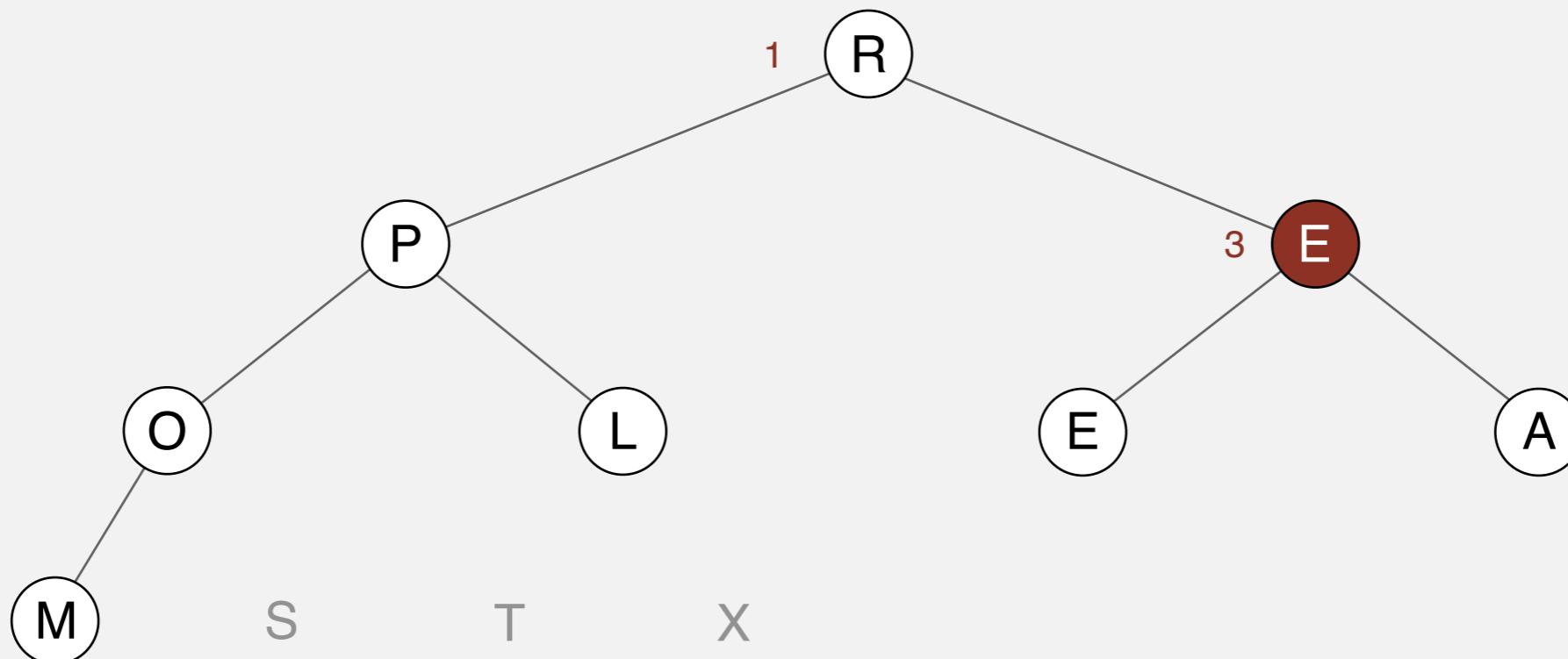


E	P	R	O	L	E	A	M	S	T	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

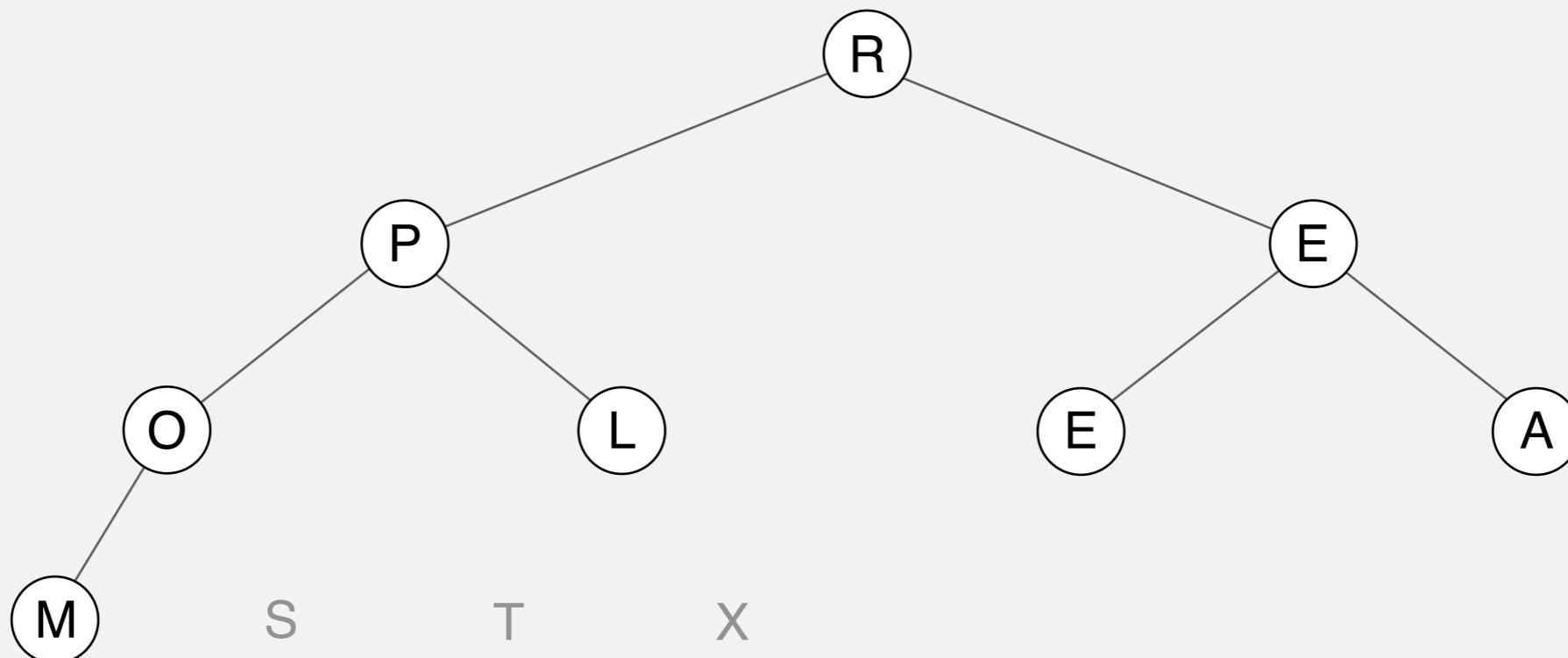
sink 1



R	P	E	O	L	E	A	M	S	T	X
1		3								

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

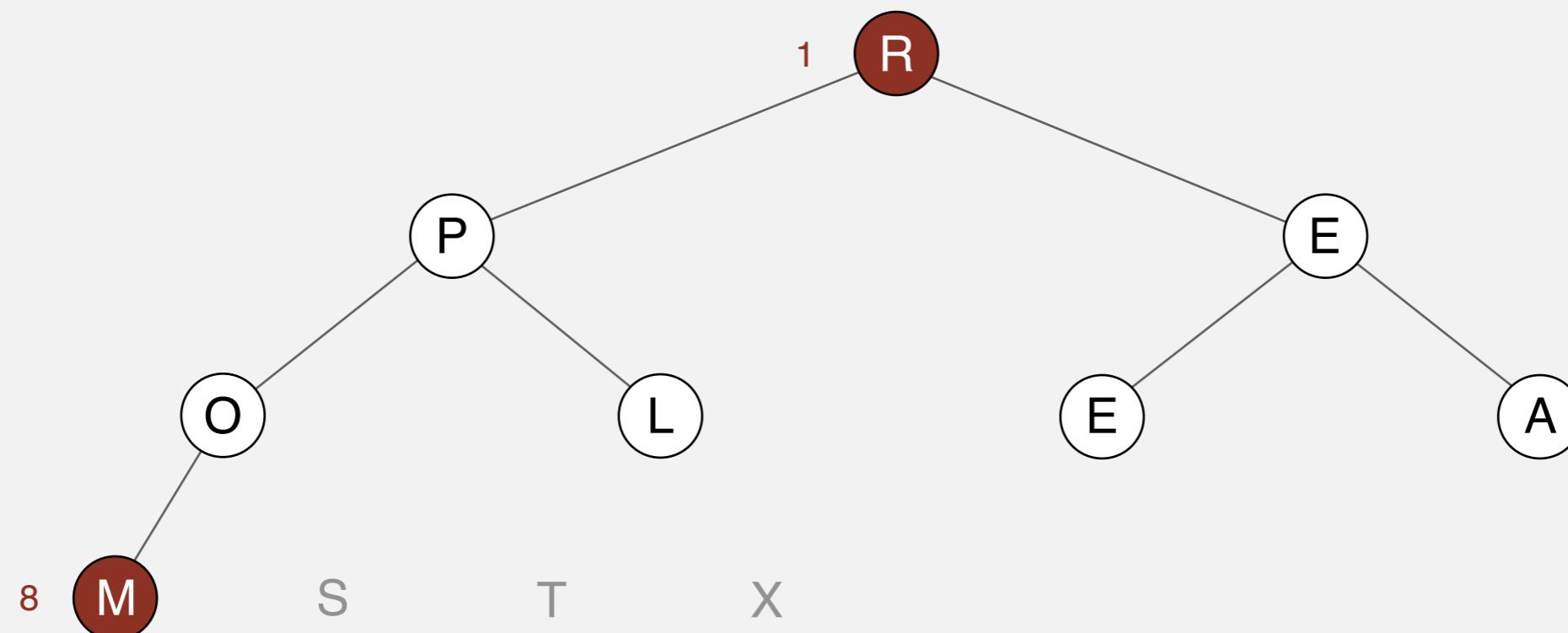


R	P	E	O	L	E	A	M	S	T	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

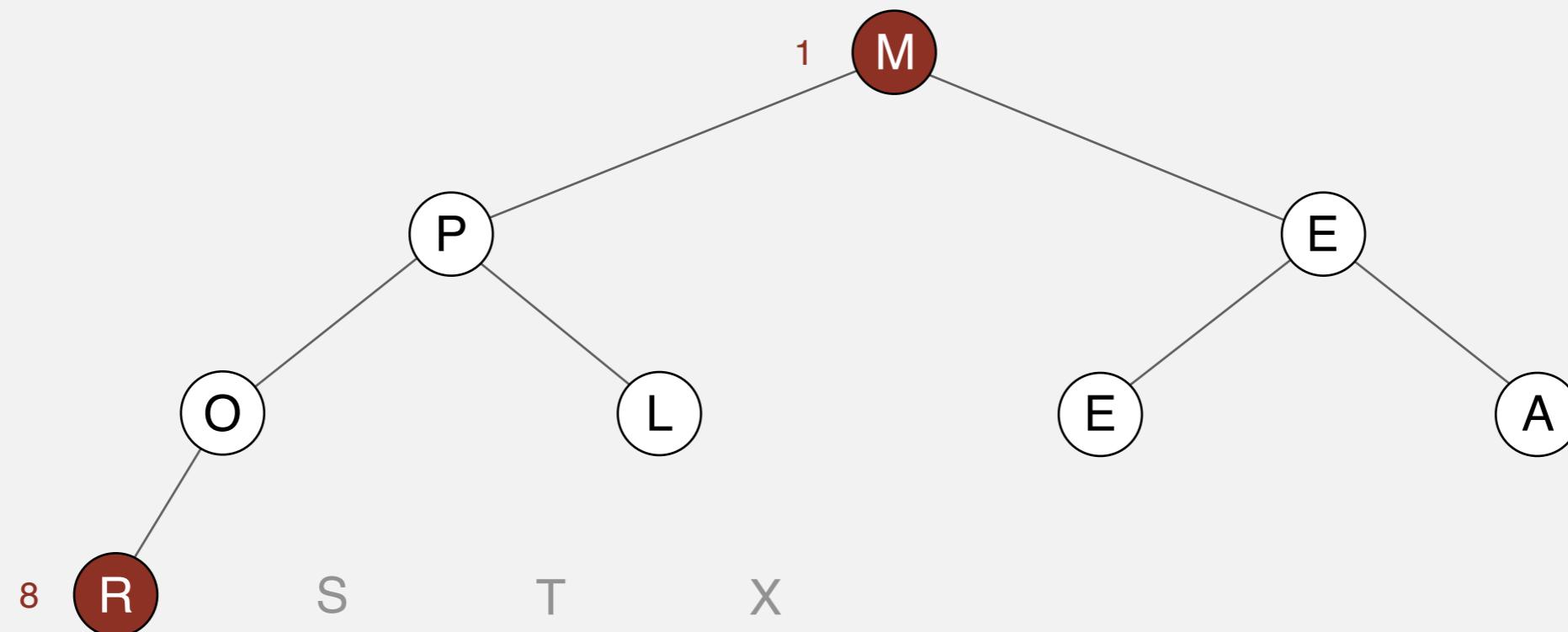


R	P	E	O	L	E	A	M	S	T	X
1							8			

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

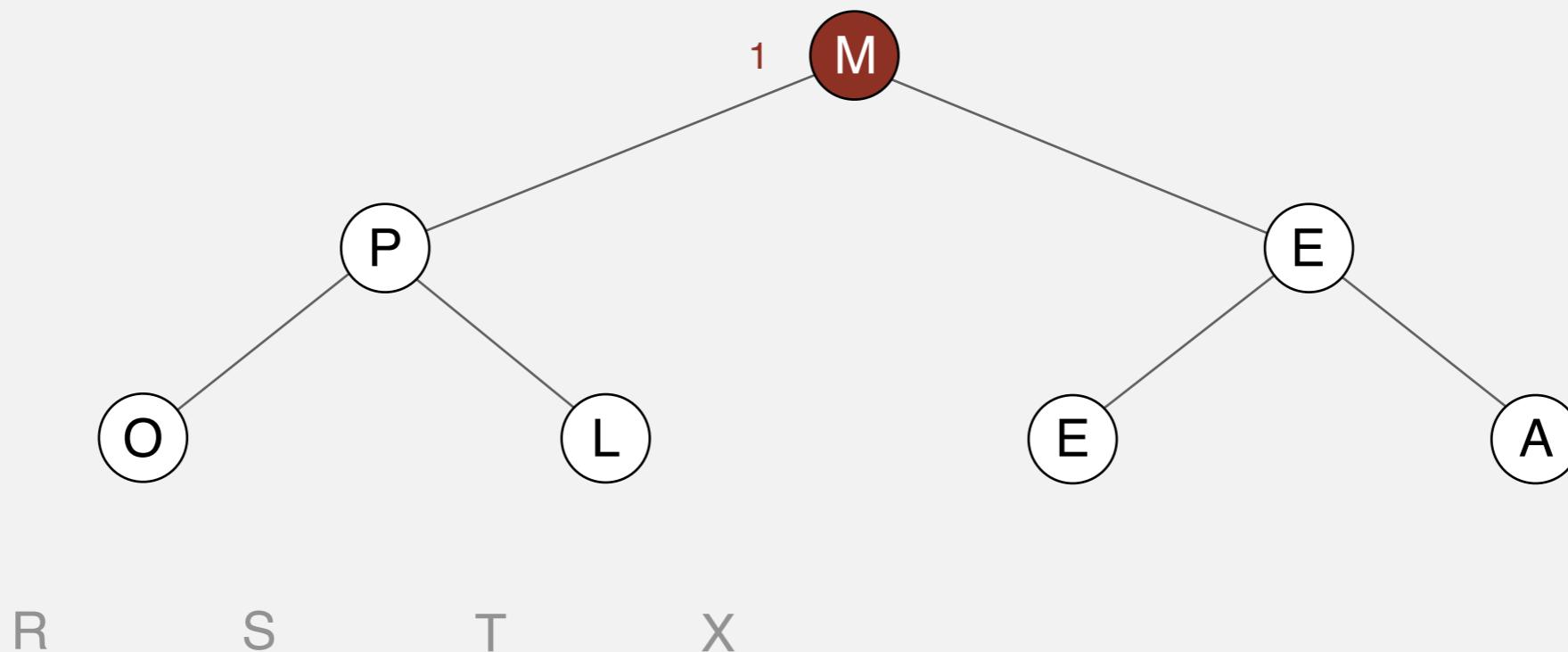


M	P	E	O	L	E	A	R	S	T	X
1							8			

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

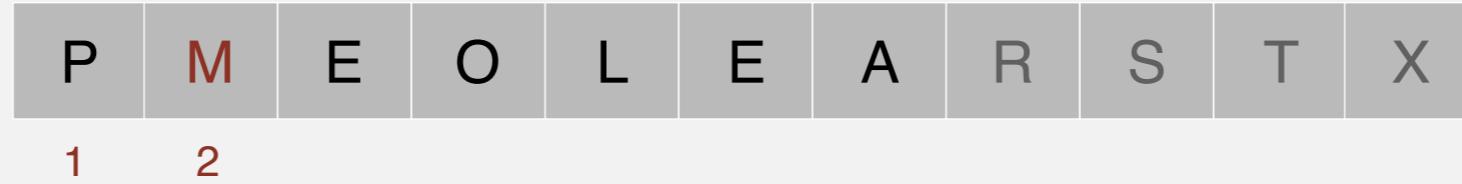
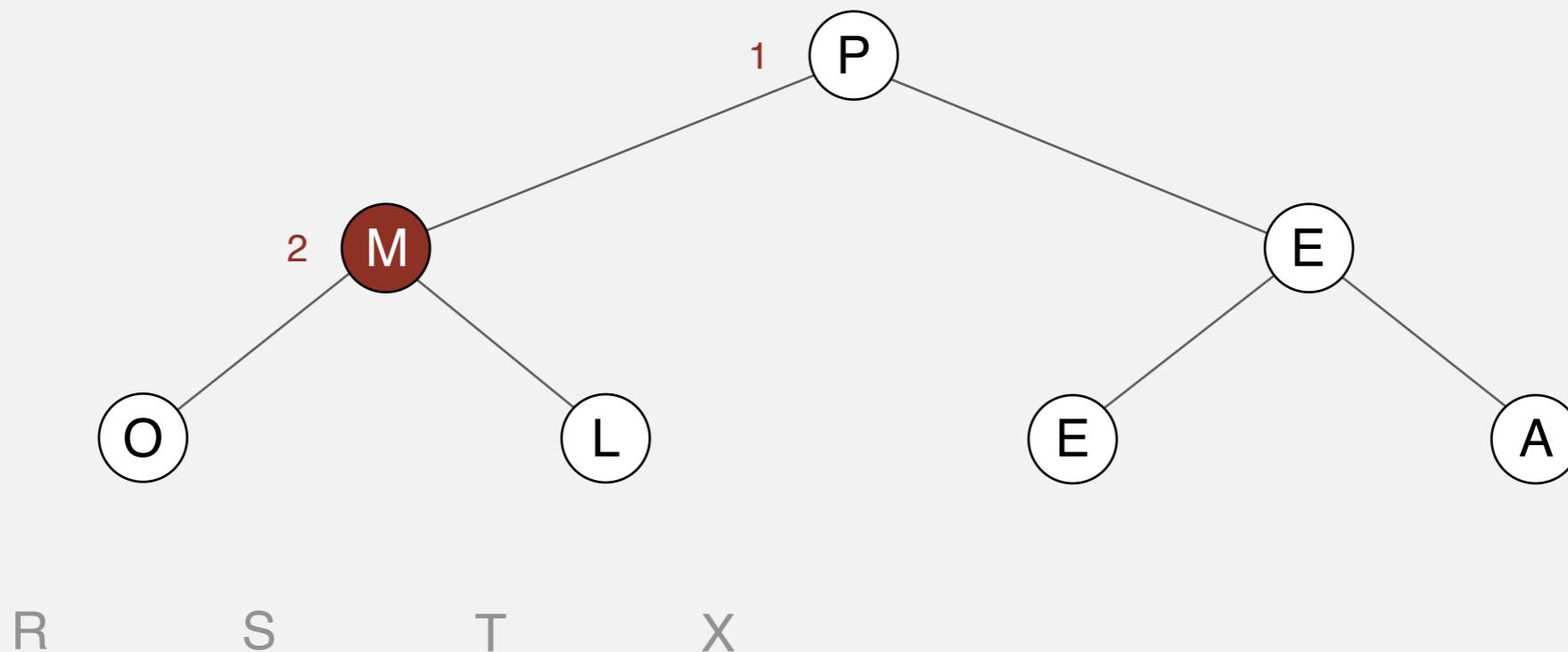


M P E O L E A R S T X

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

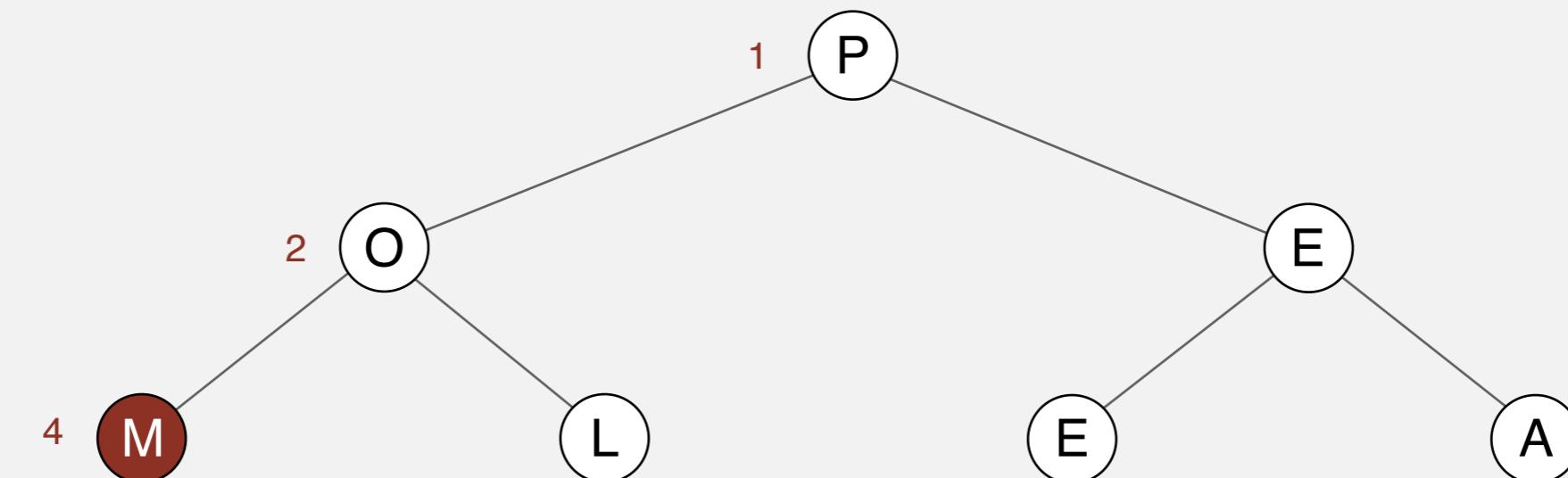
sink 1



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

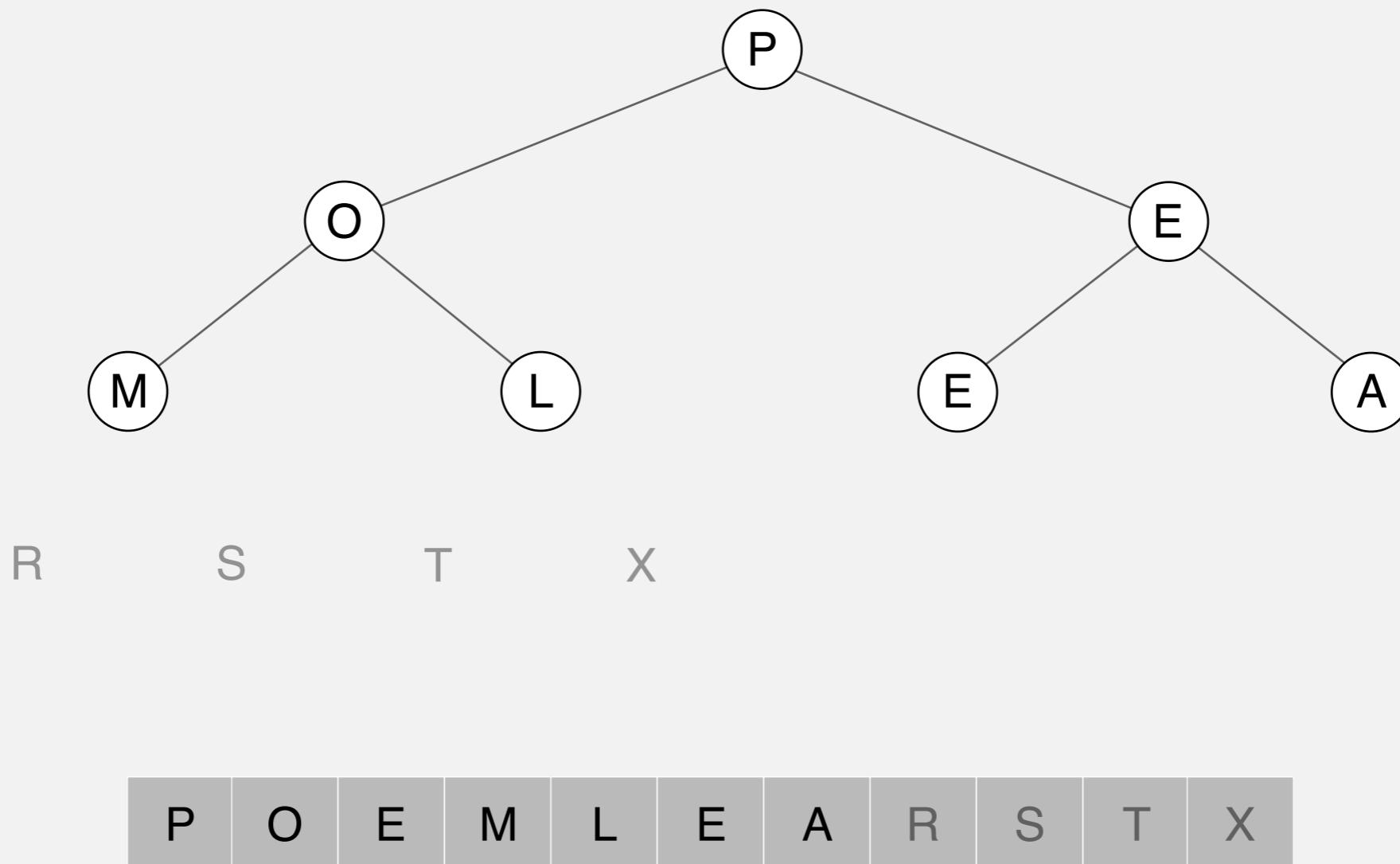


R S T X

P	O	E	M	L	E	A	R	S	T	X
1	2		4							

Heapsort demo

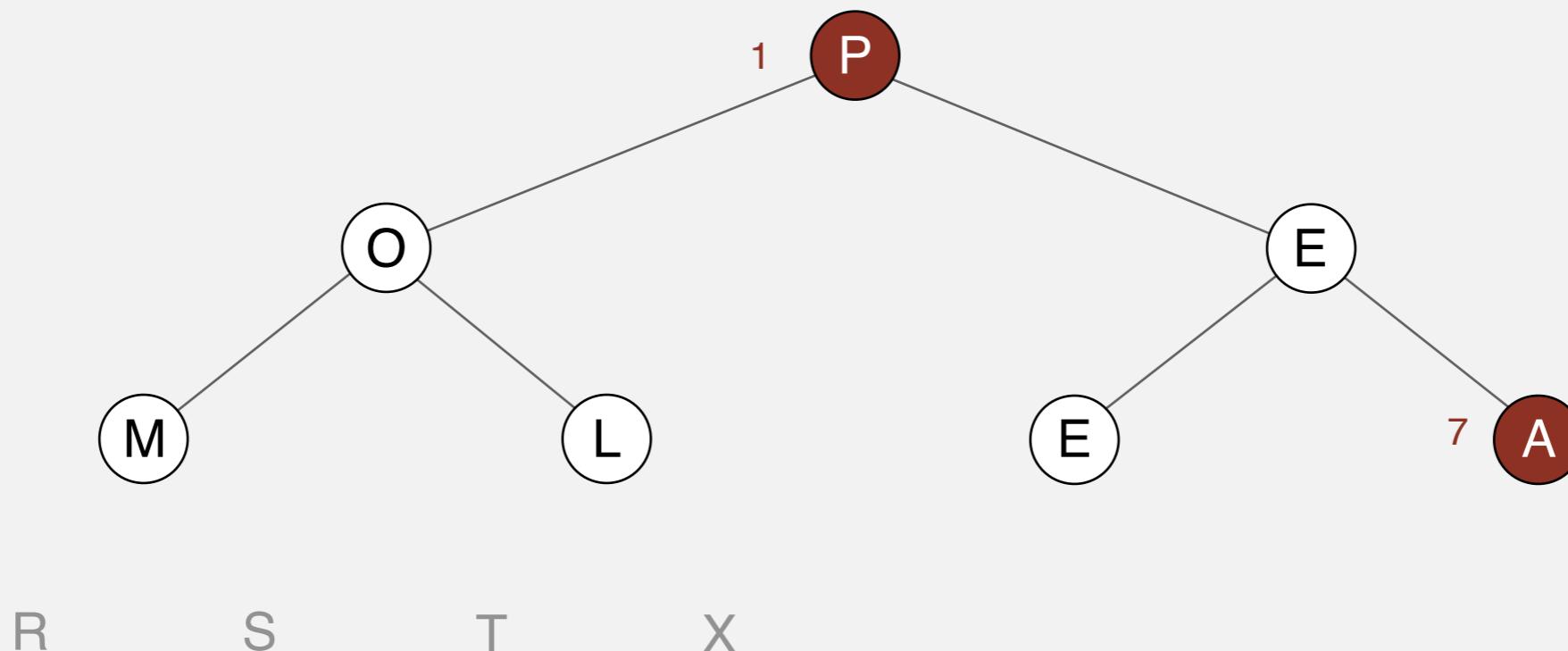
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7

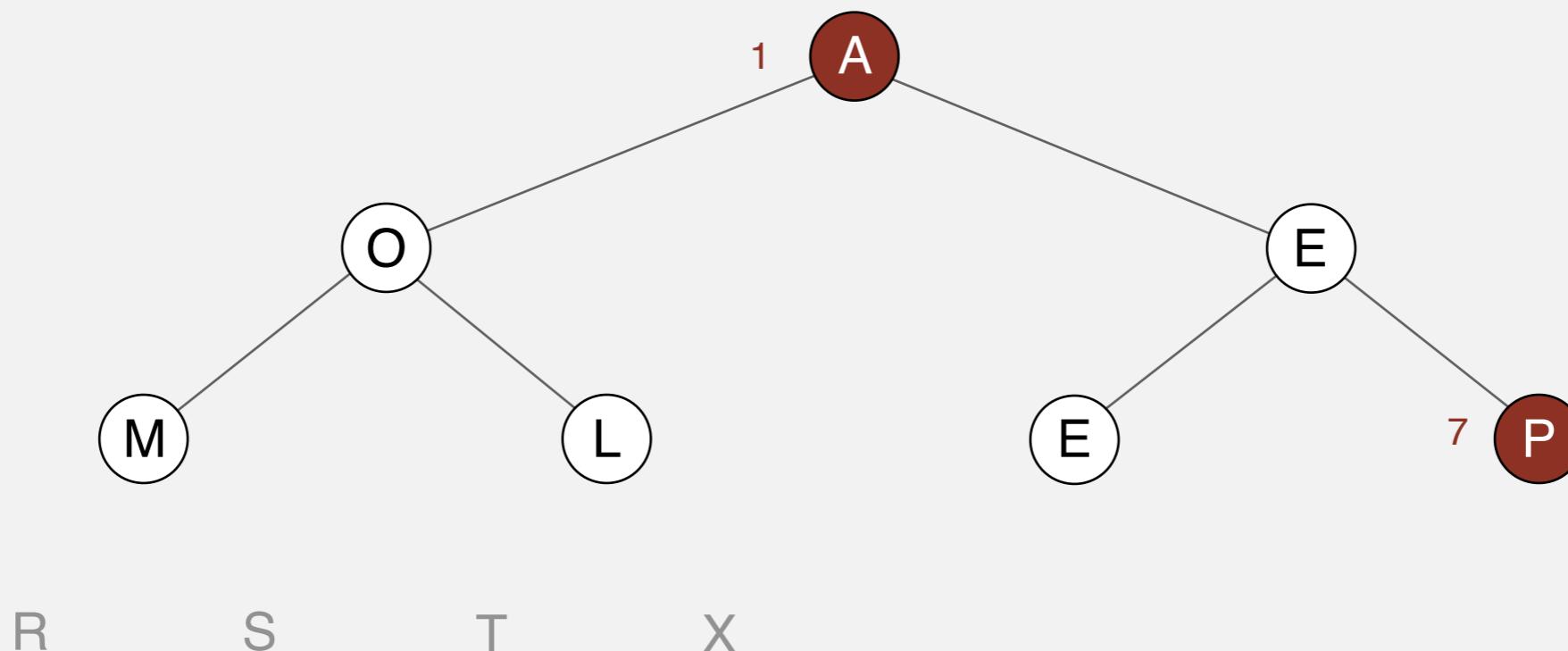


P	O	E	M	L	E	A	R	S	T	X
1						7				

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7

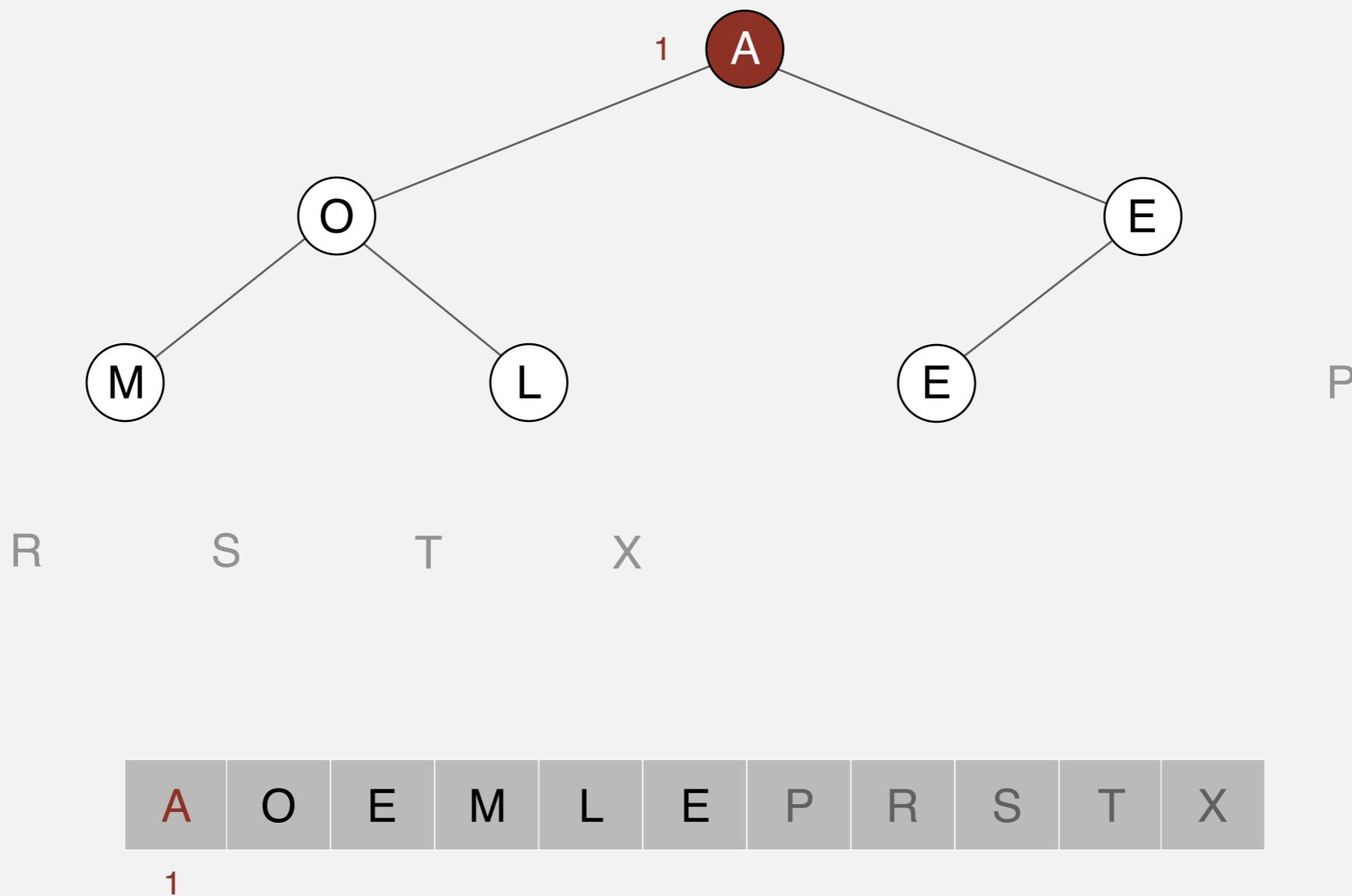


A	O	E	M	L	E	P	R	S	T	X
1						7				

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

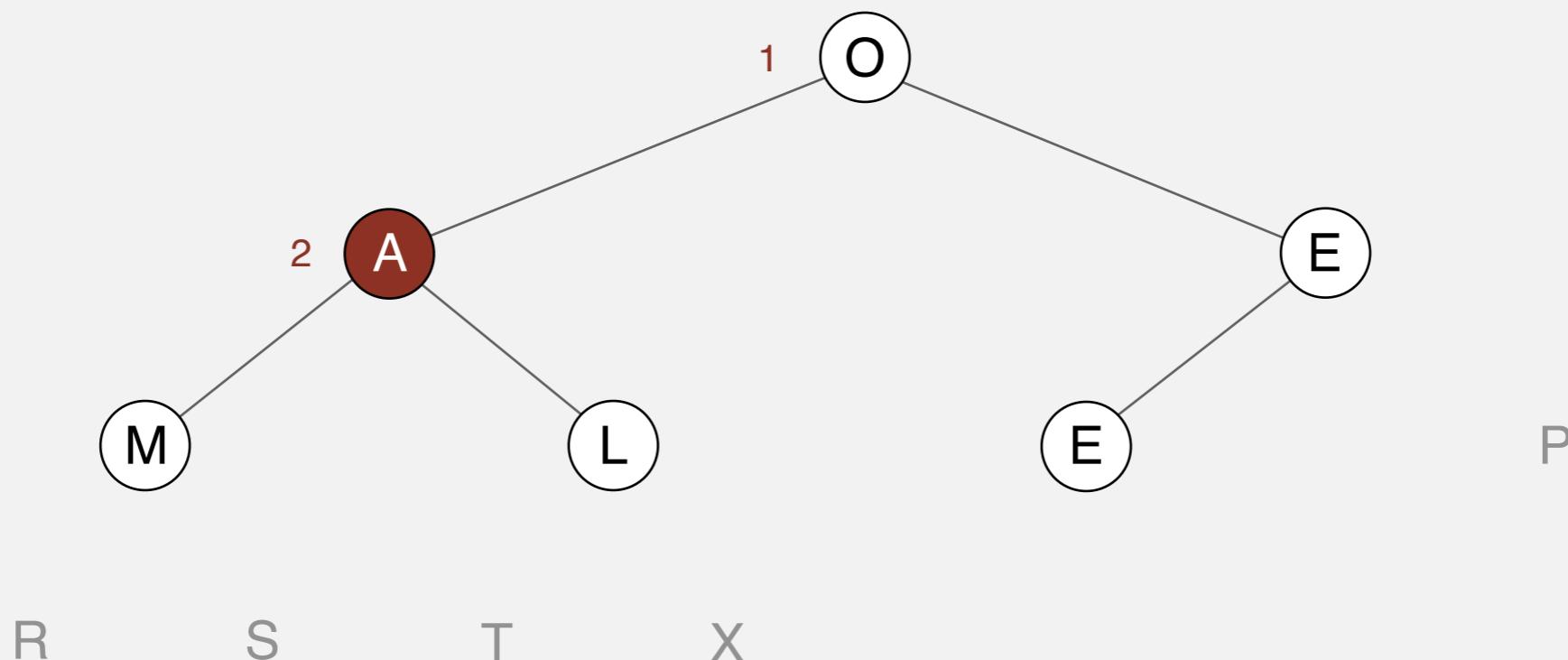
sink 1



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

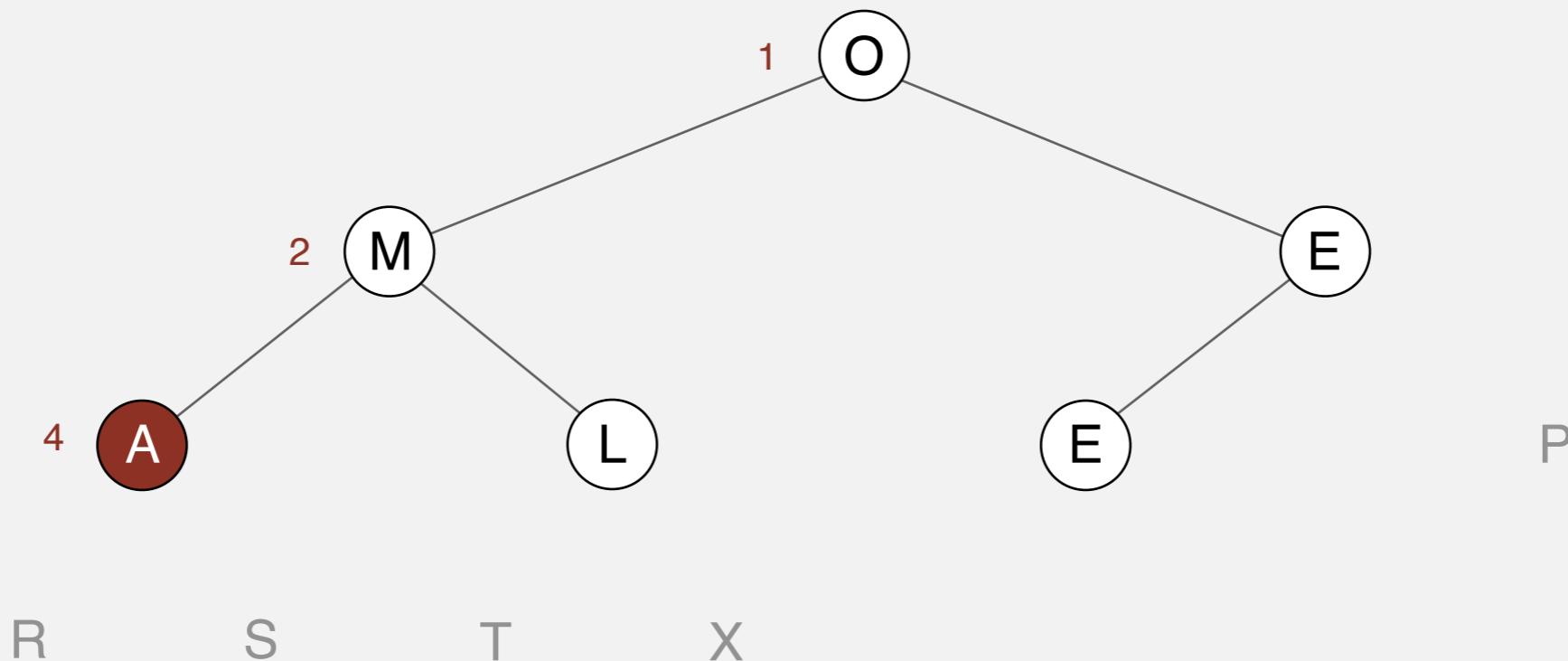


O A E M L E P R S T X
1 2

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

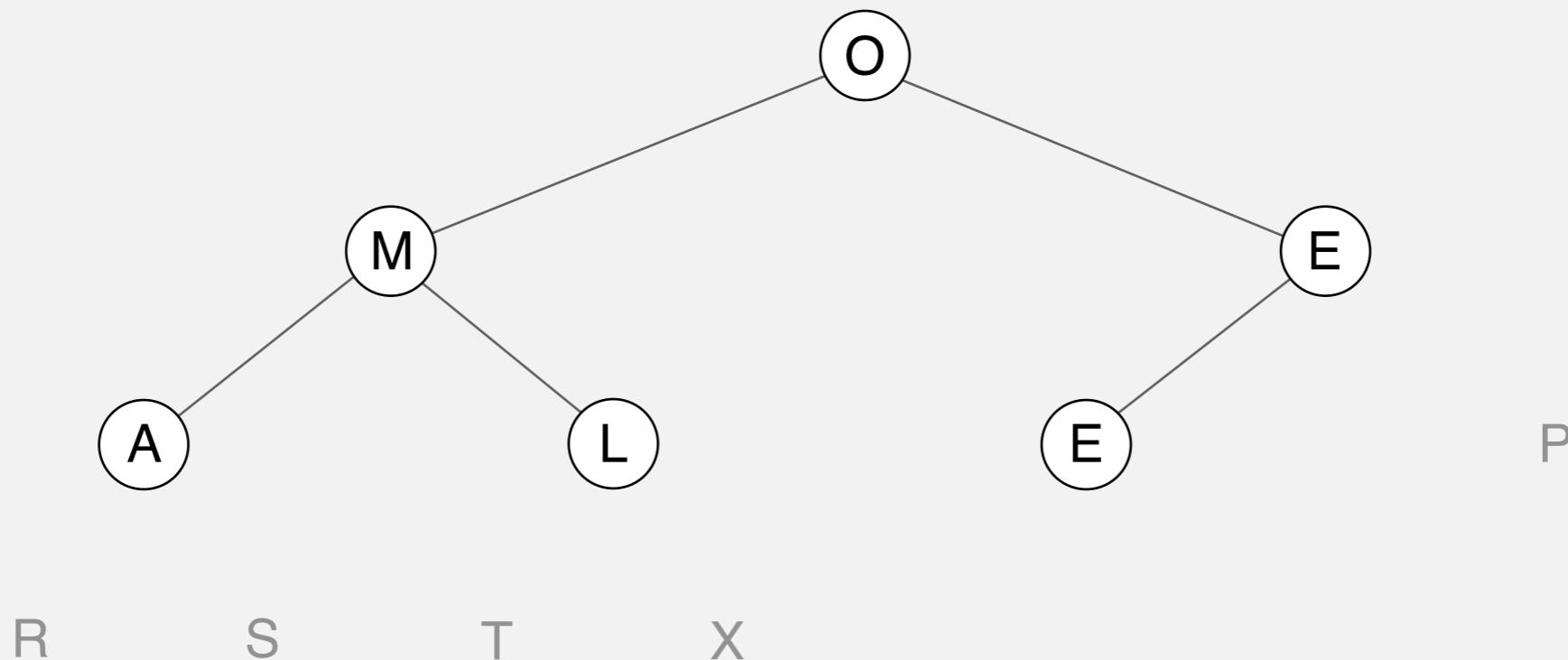


O	M	E	A	L	E	P	R	S	T	X
1	2		4							

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

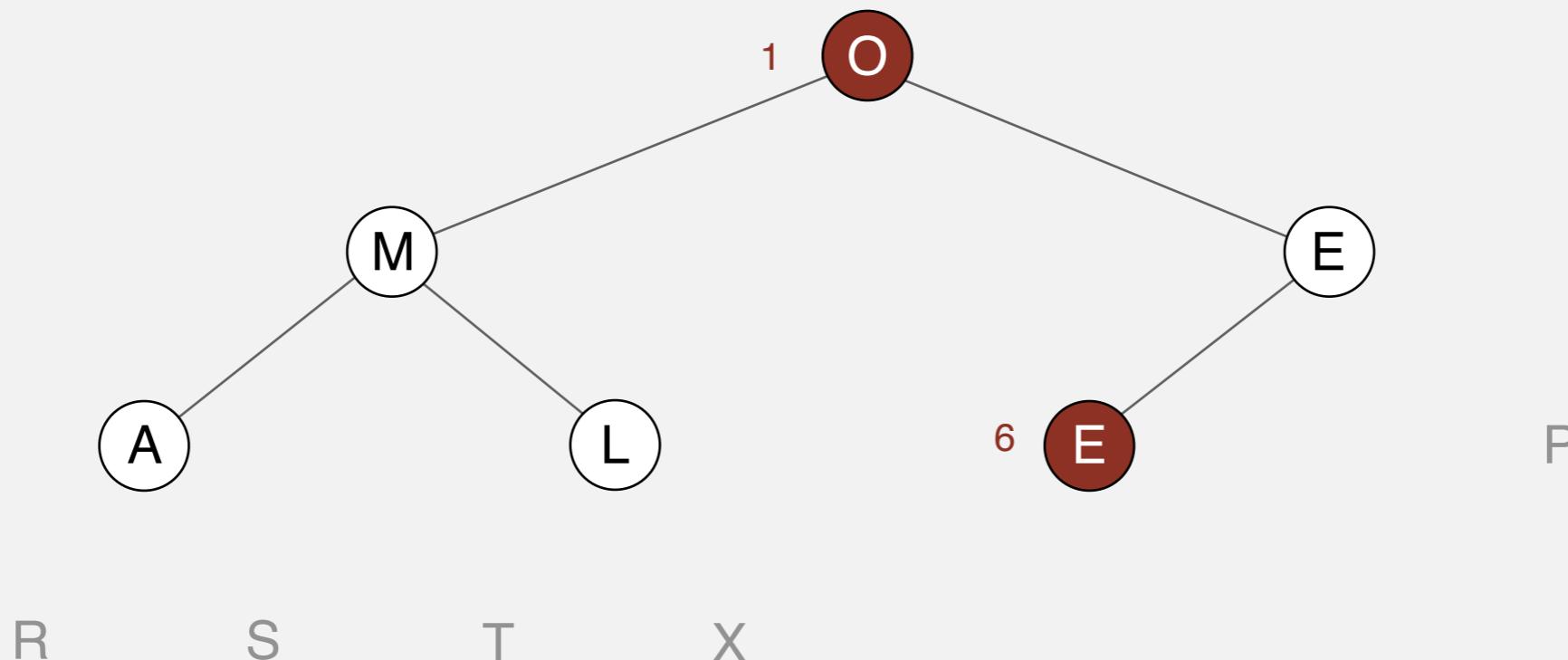


O	M	E	A	L	E	P	R	S	T	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

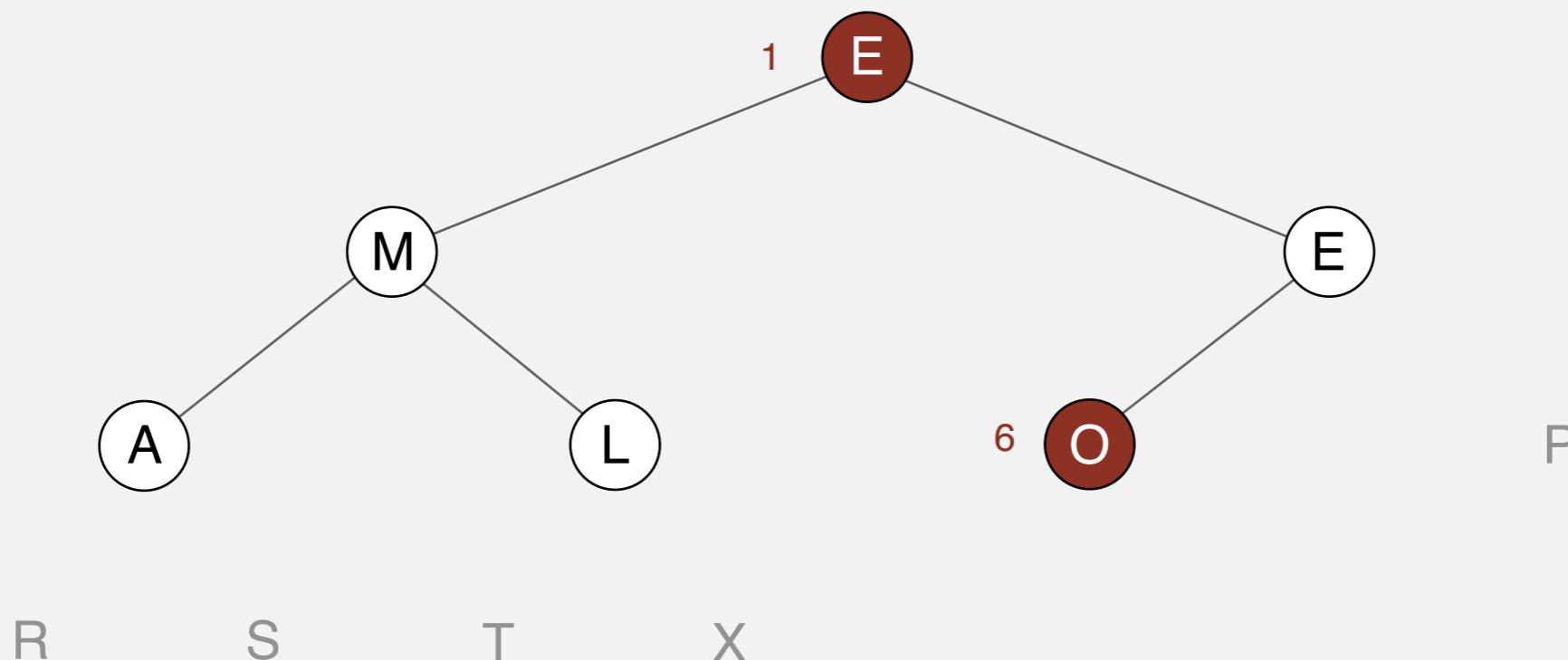


O	M	E	A	L	E	P	R	S	T	X
1					6					

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

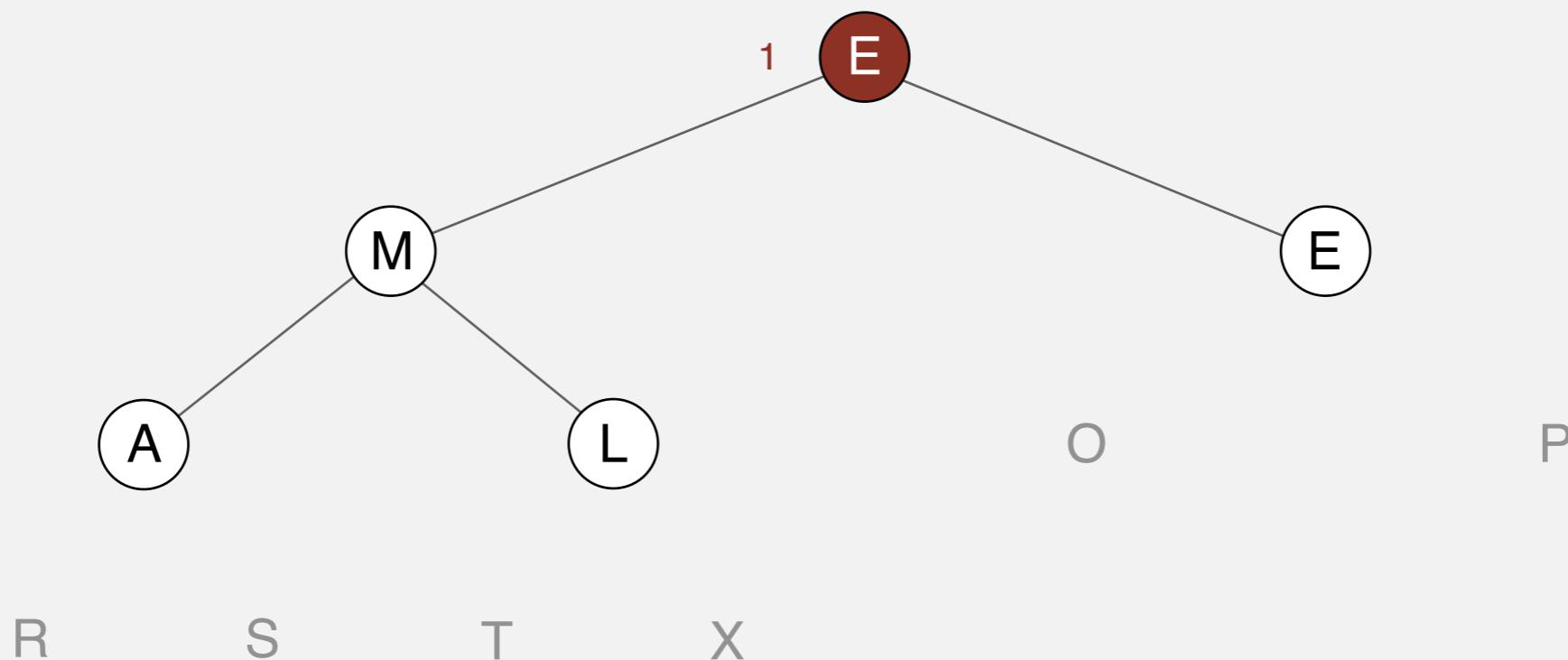


E	M	E	A	L	O	P	R	S	T	X
1					6					

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

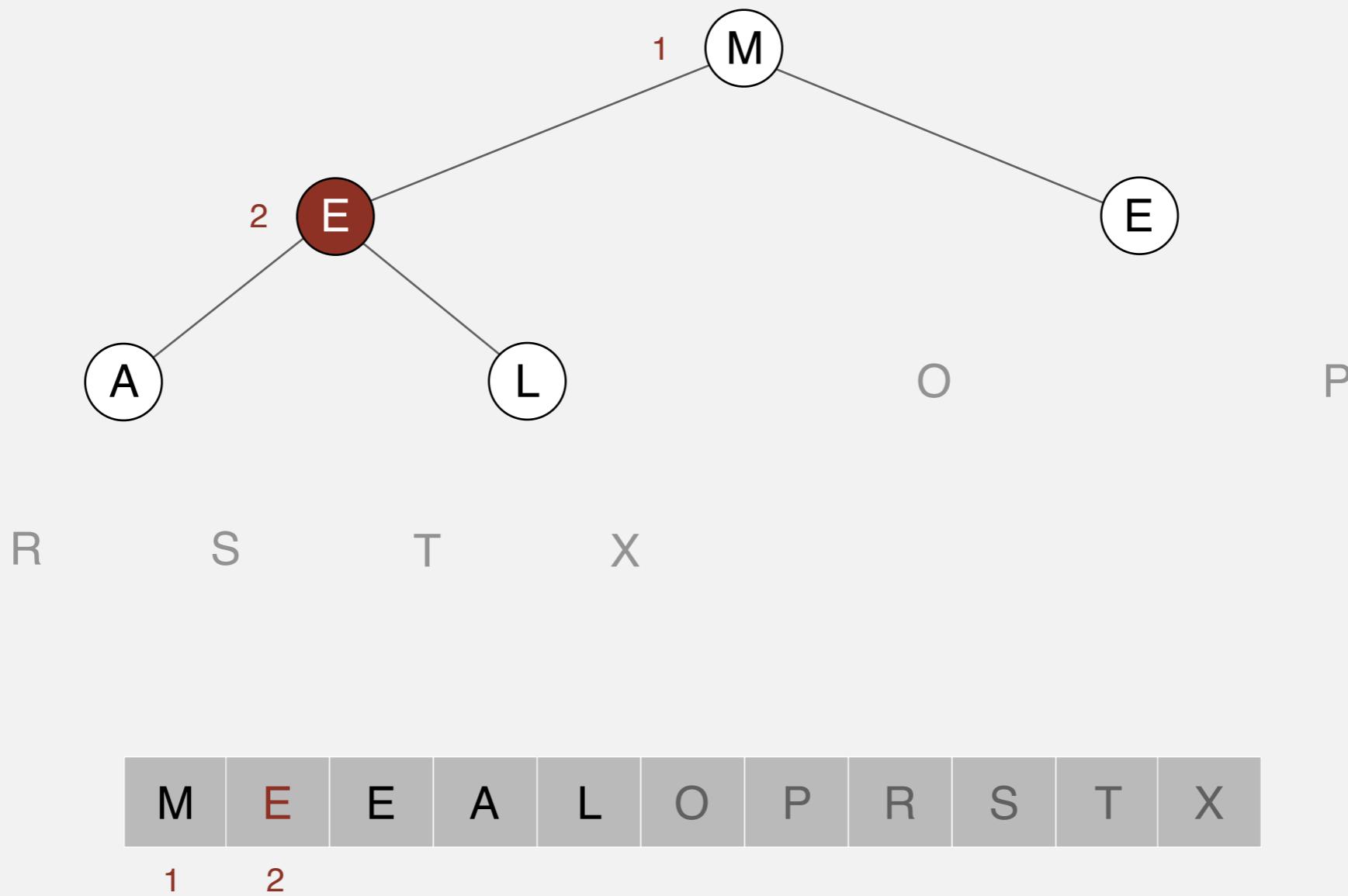


E M E A L O P R S T X
1

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

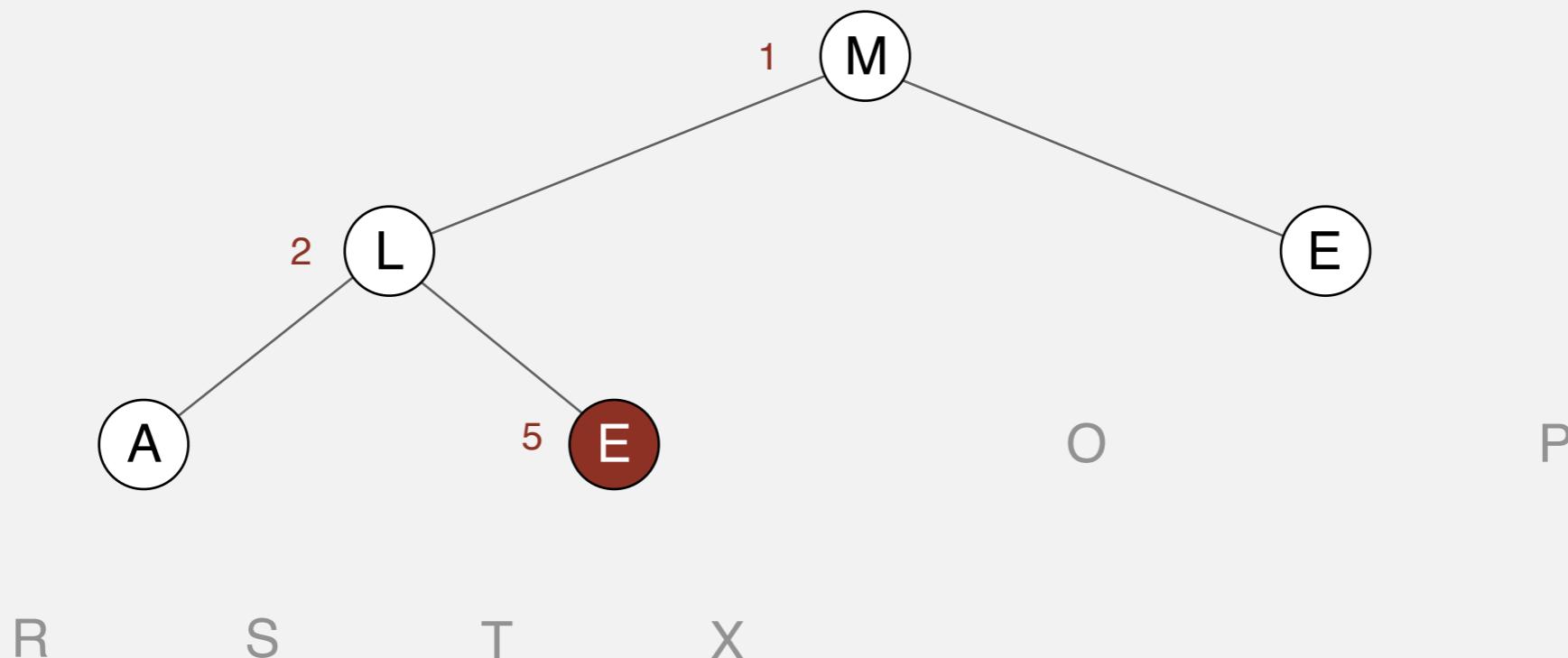
sink 1



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

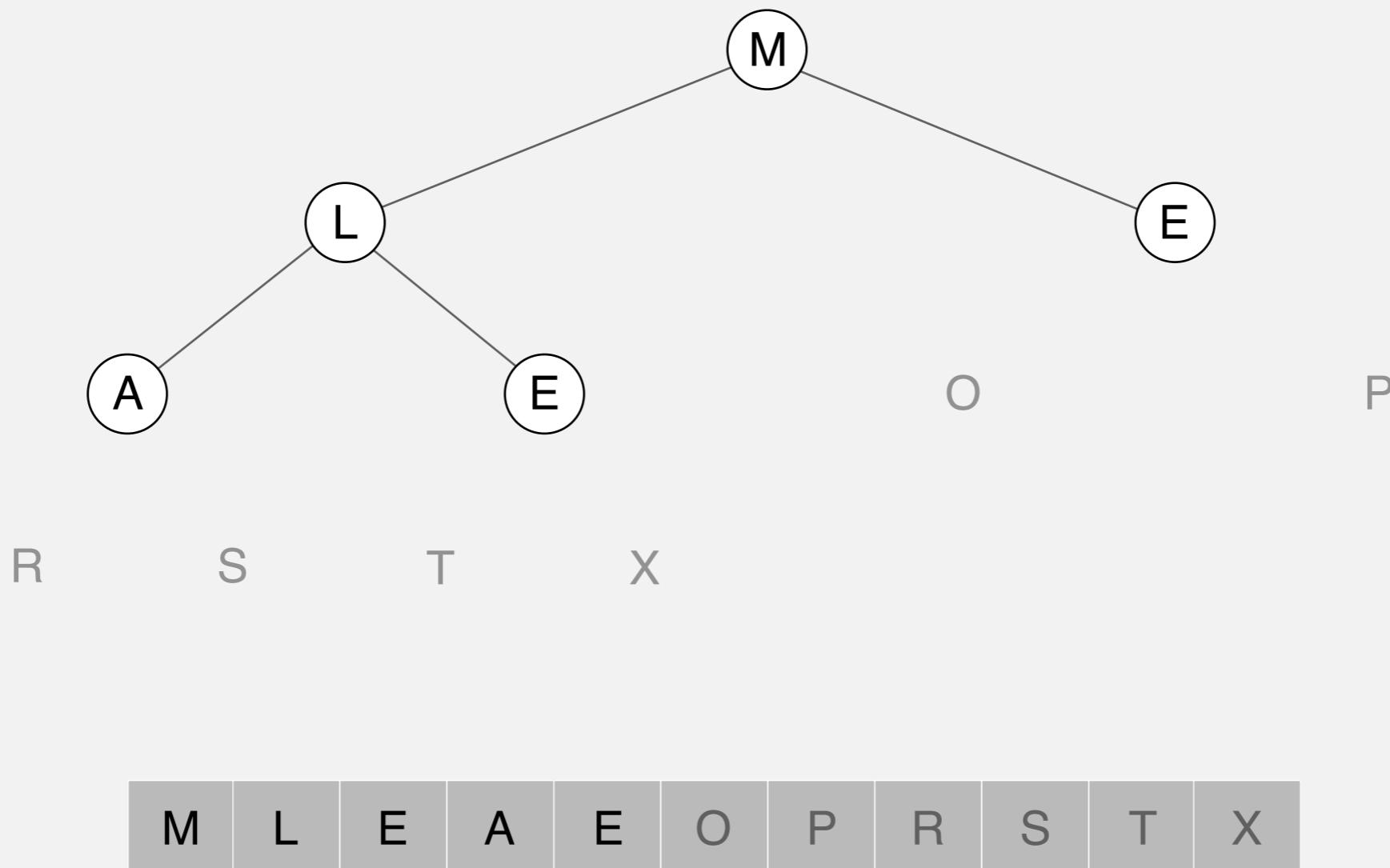
sink 1



M	L	E	A	E	O	P	R	S	T	X
1	2		4	5	7					

Heapsort demo

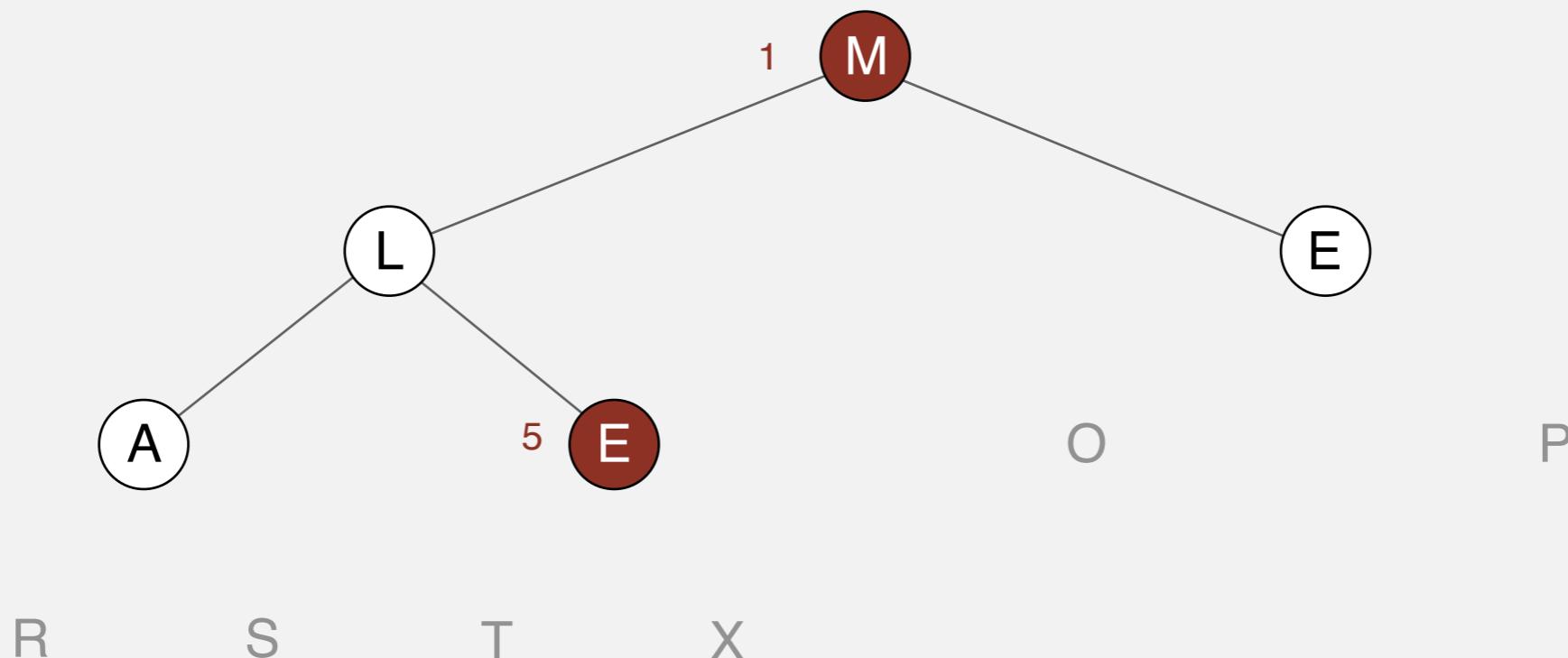
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5

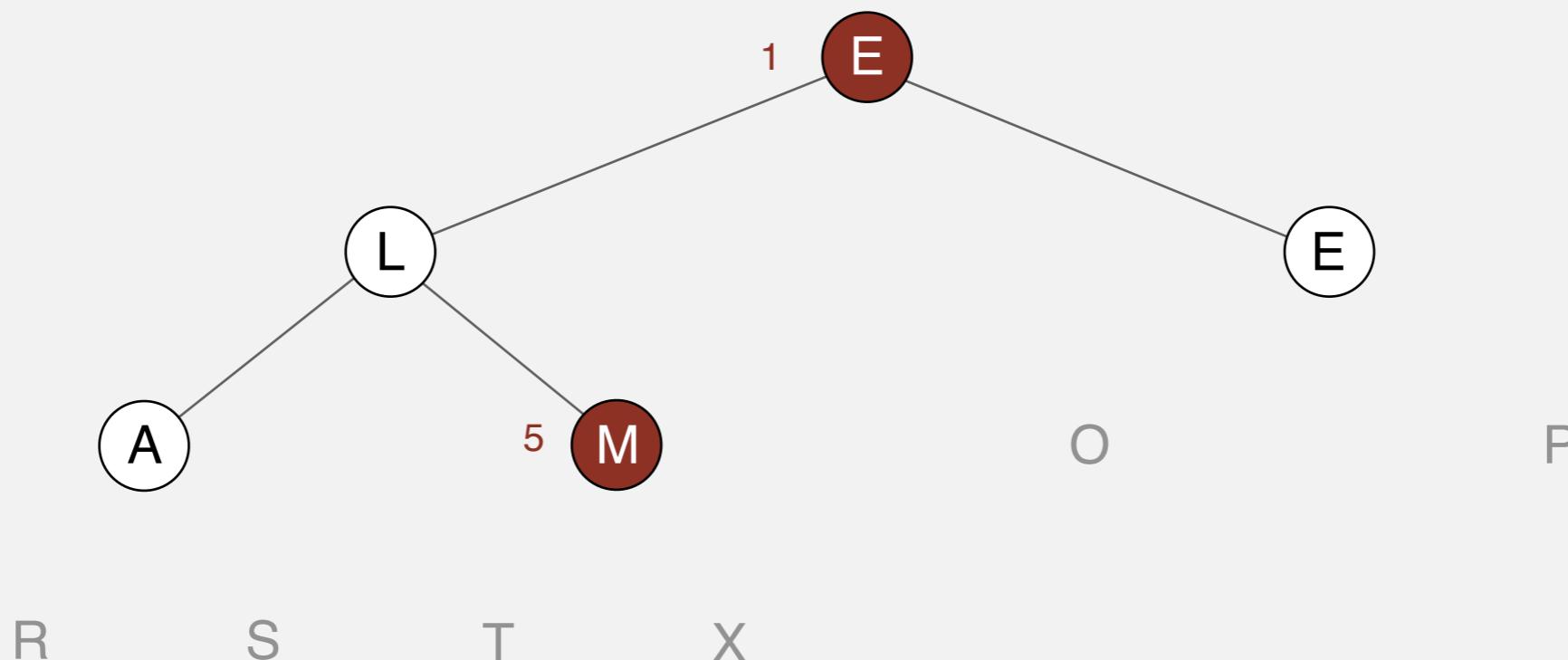


M	L	E	A	E	O	P	R	S	T	X
1				5						

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5

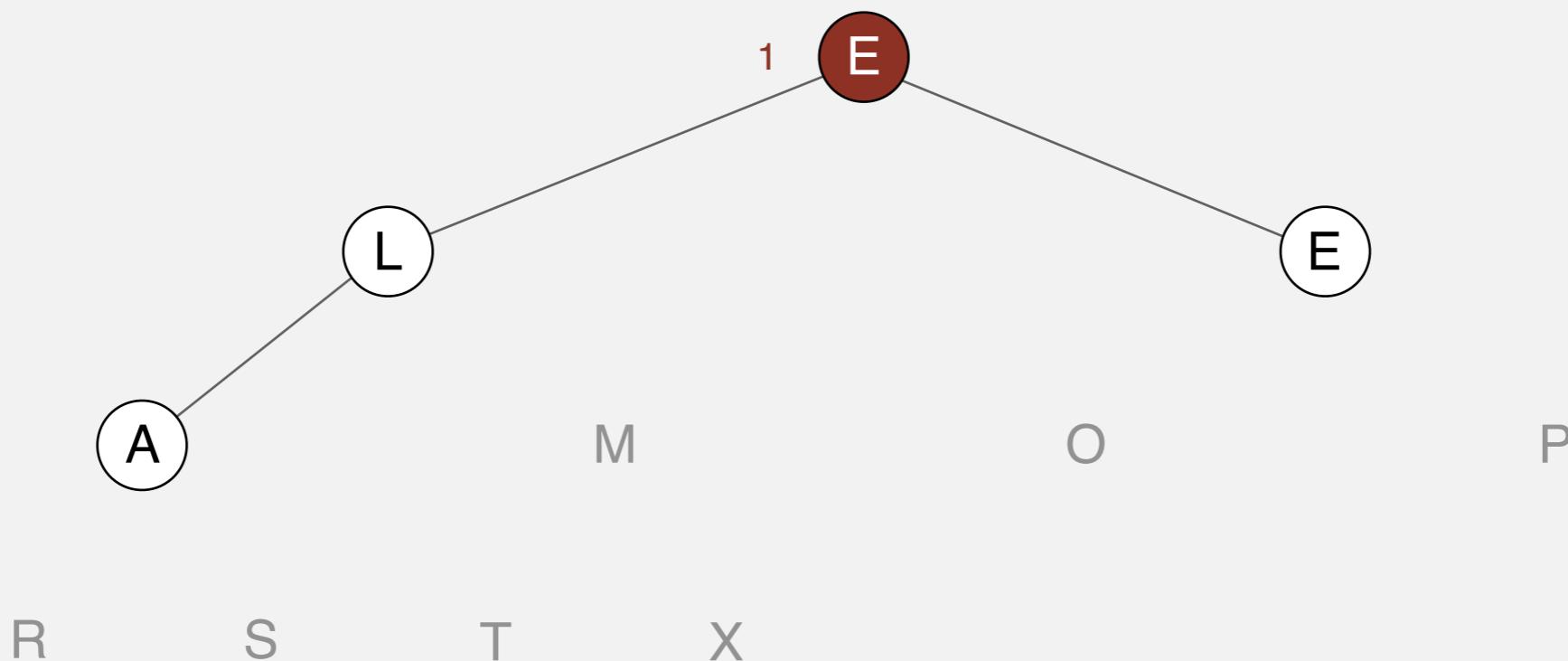


E	L	E	A	M	O	P	R	S	T	X
1				5						

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

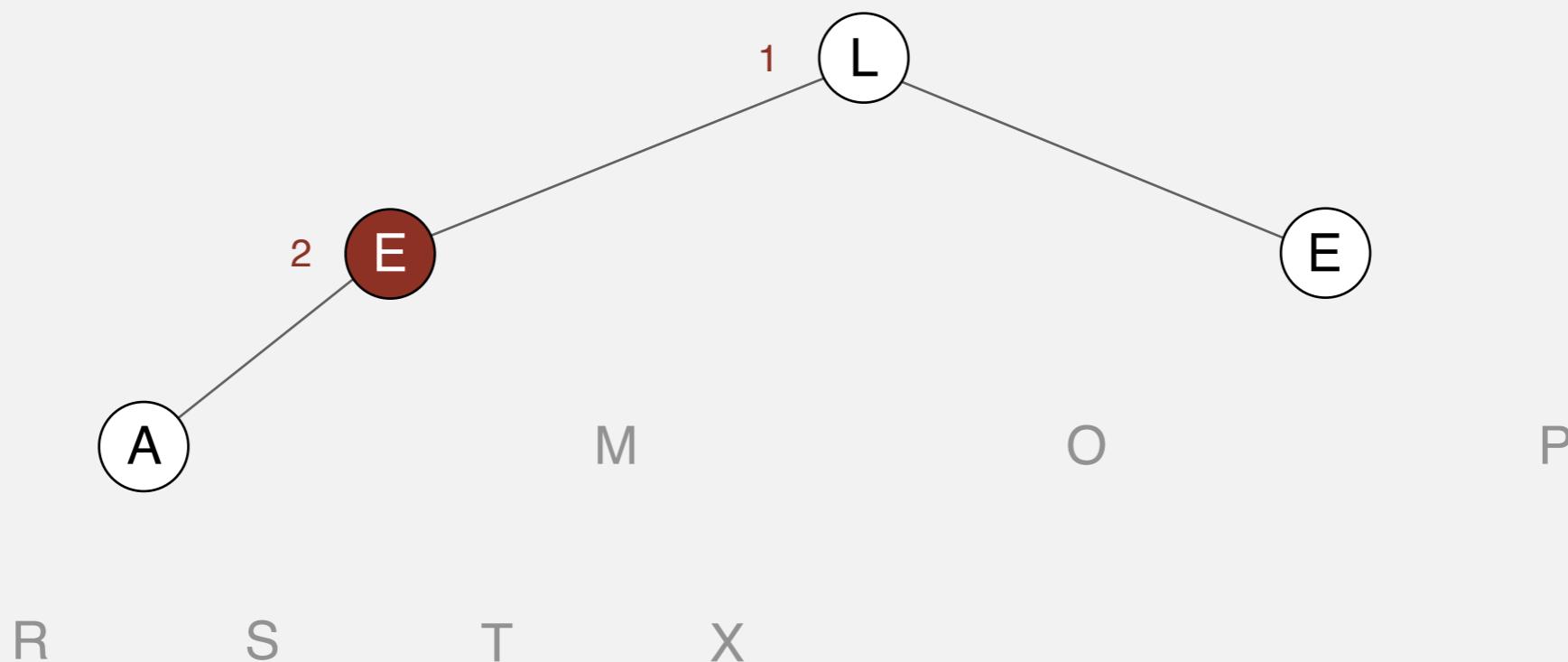


E L E A M O P R S T X
1

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

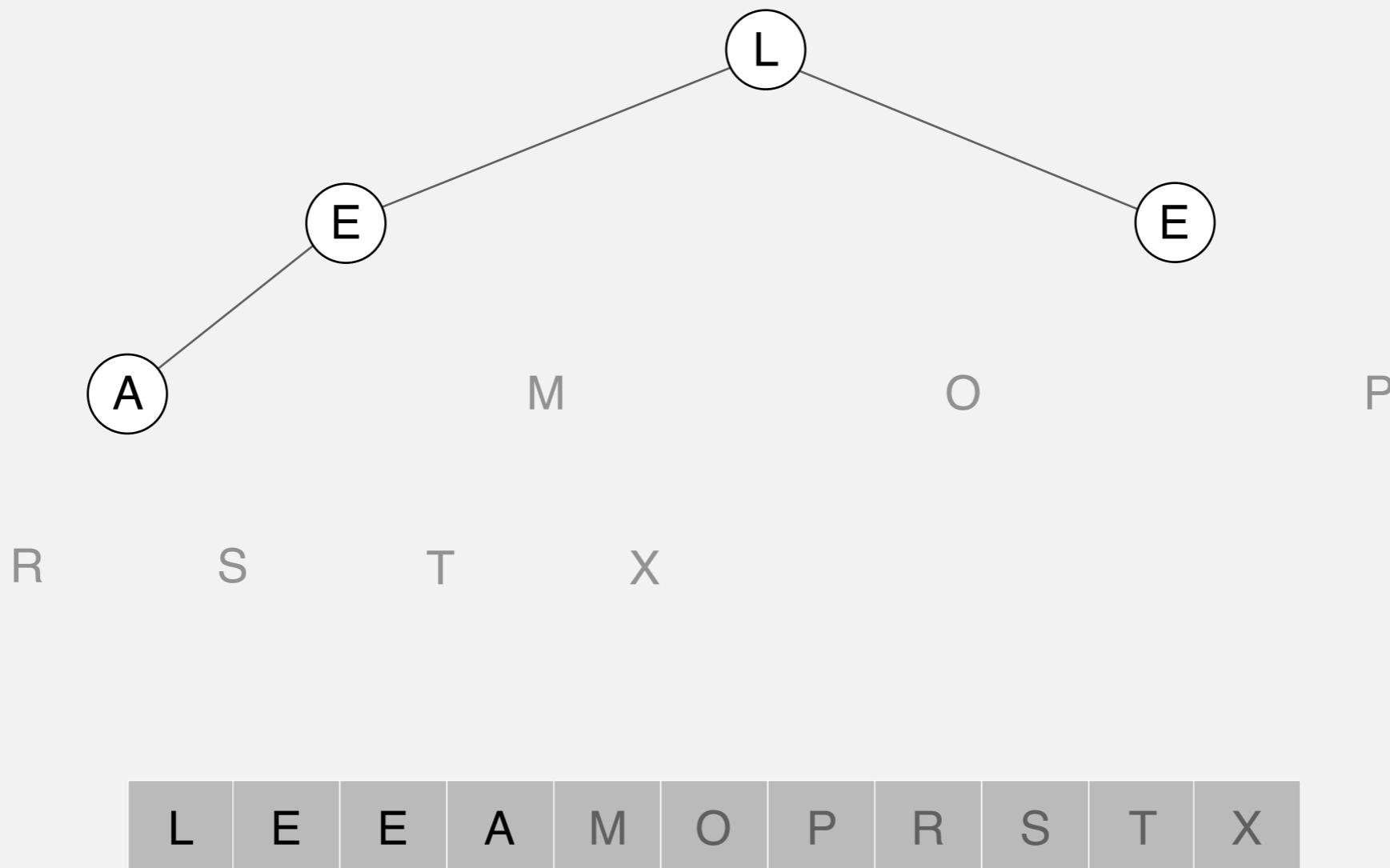
sink 1



L E E A M O P R S T X
1 2

Heapsort demo

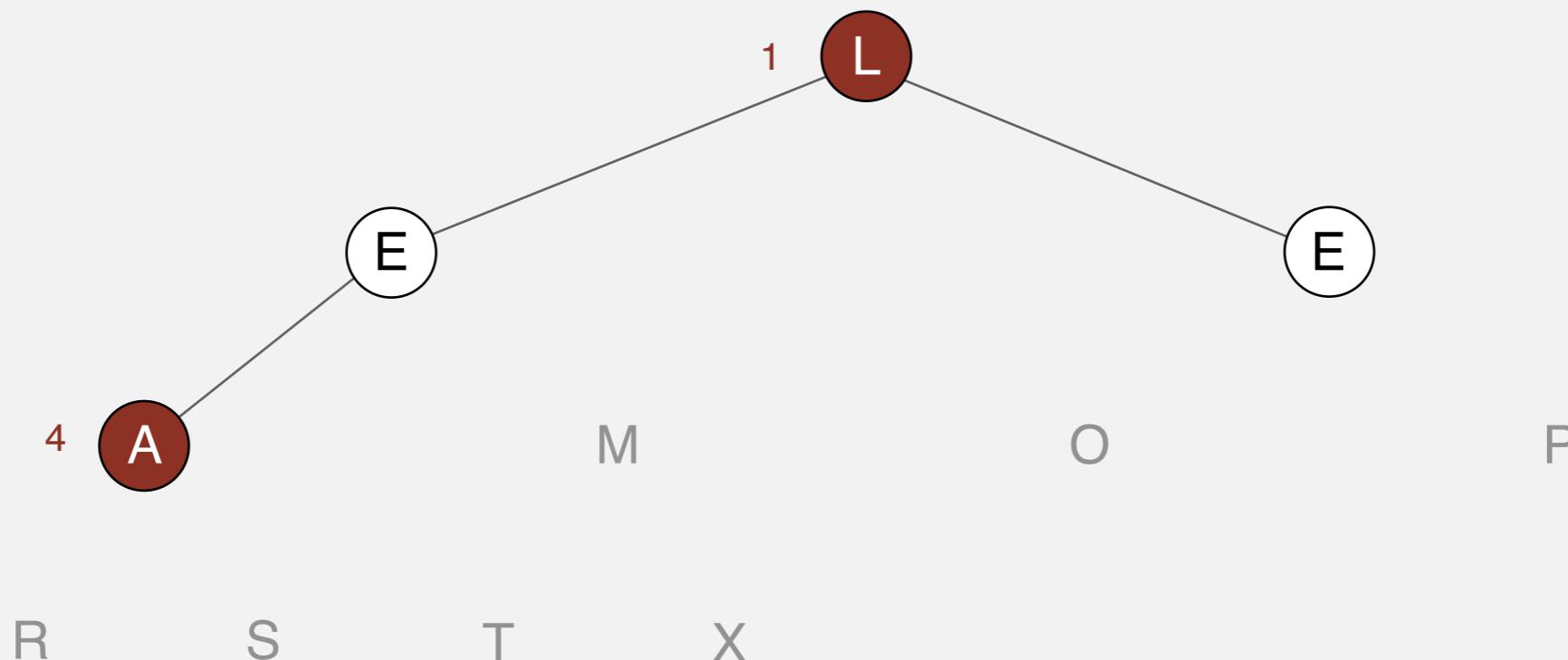
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

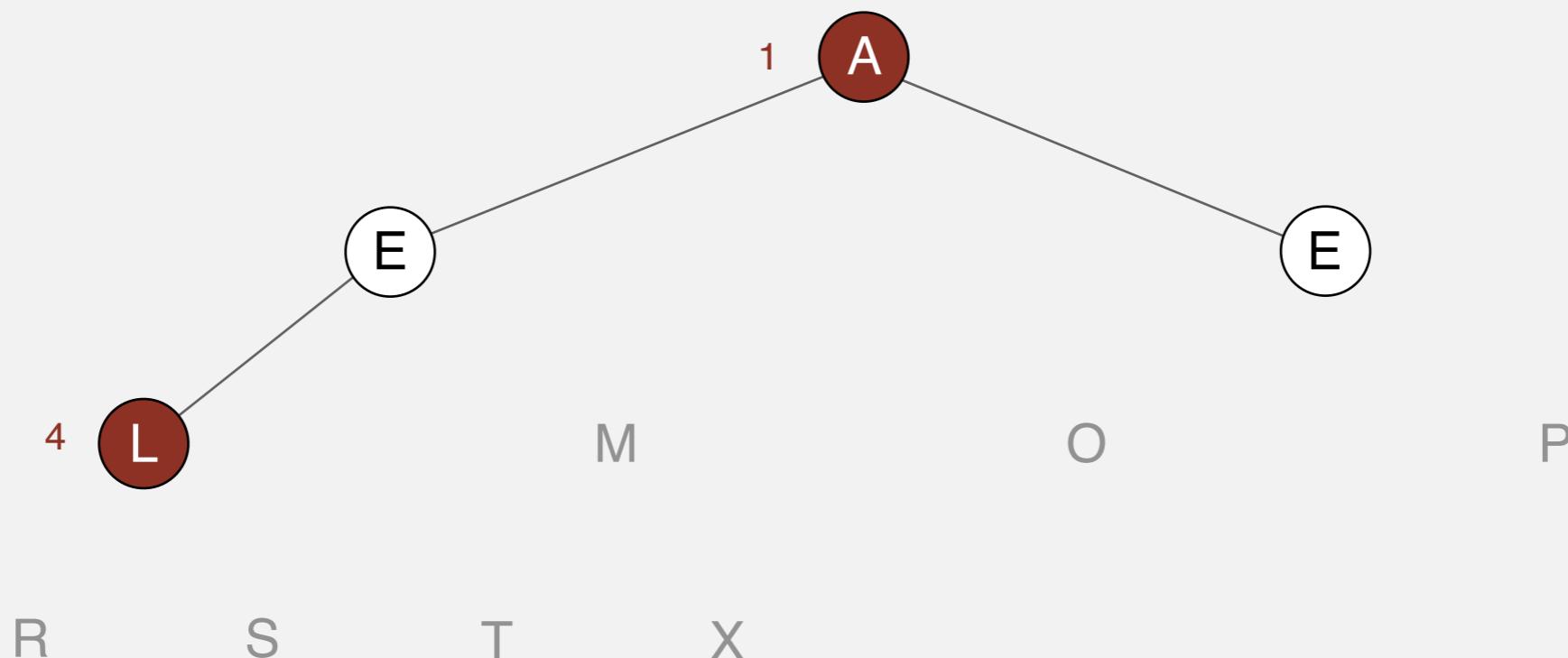


L	E	E	A	M	O	P	R	S	T	X
1			4							

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

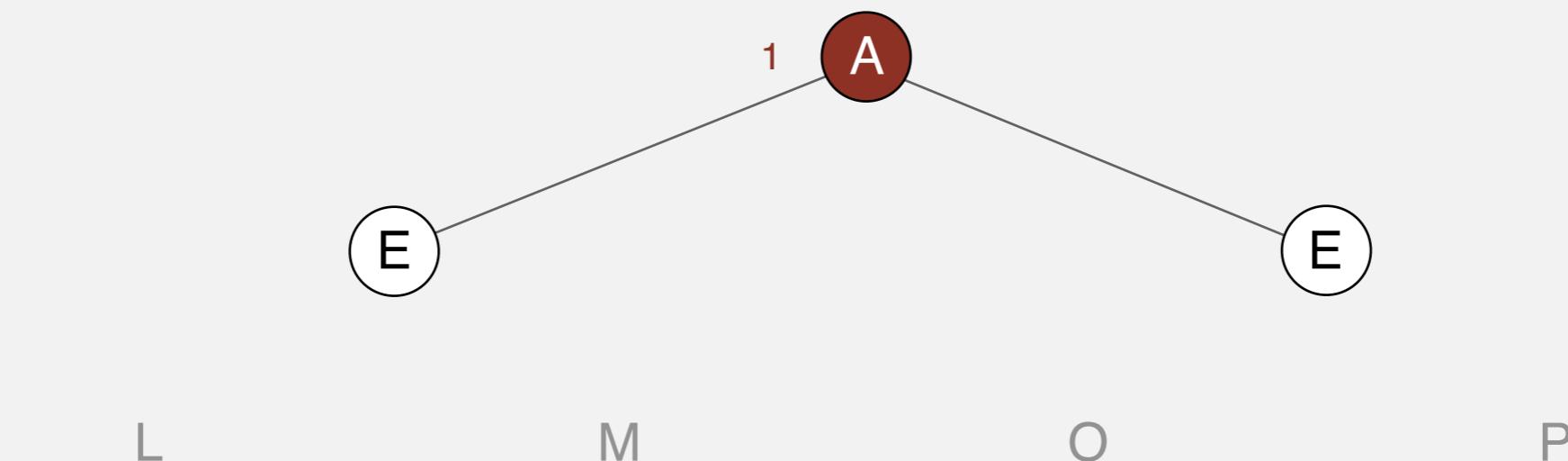


A	E	E	L	M	O	P	R	S	T	X
1			4							

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

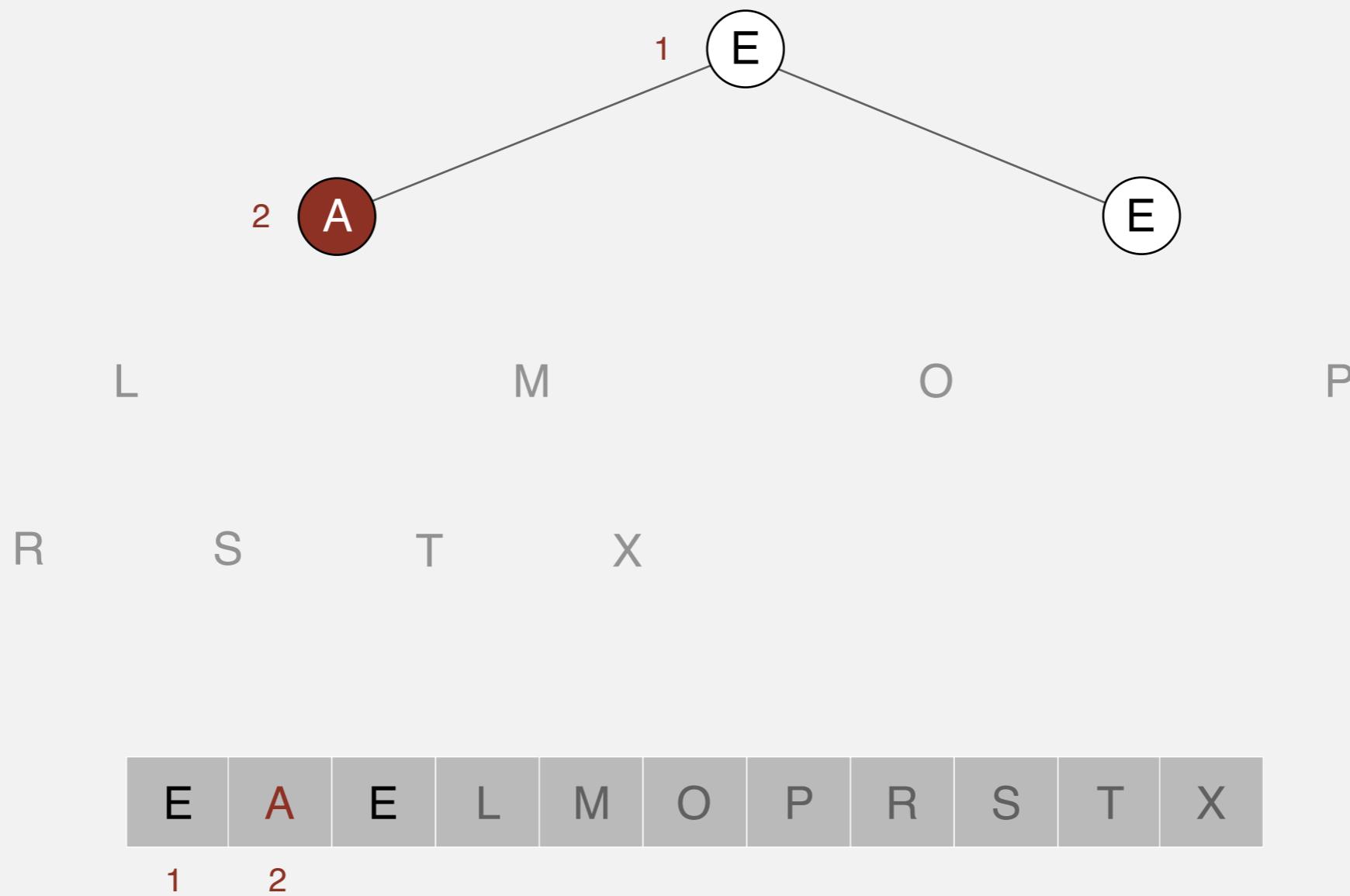


A E E L M O P R S T X
1

Heapsort demo

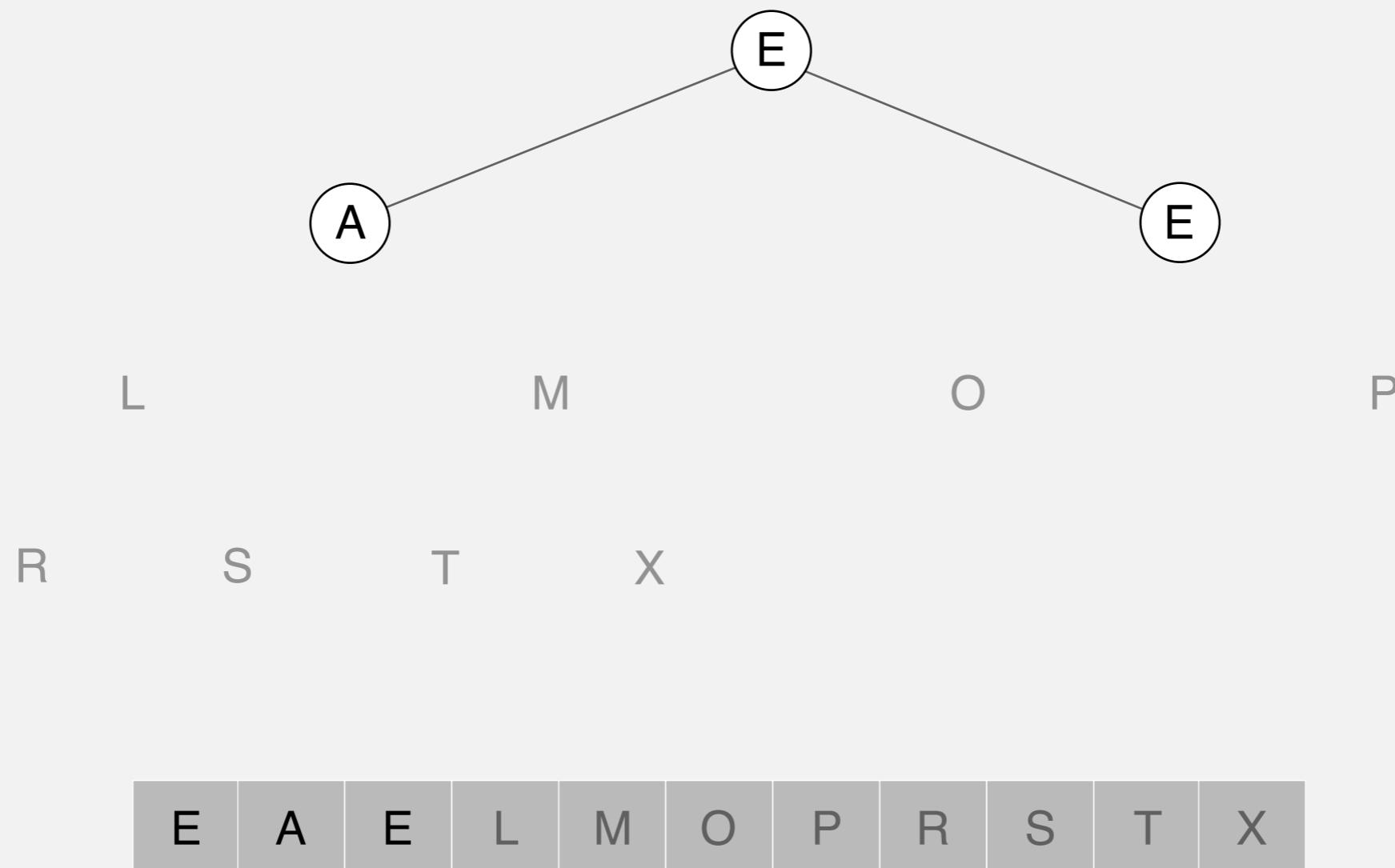
Sortdown. Repeatedly delete the largest remaining item.

sink 1



Heapsort demo

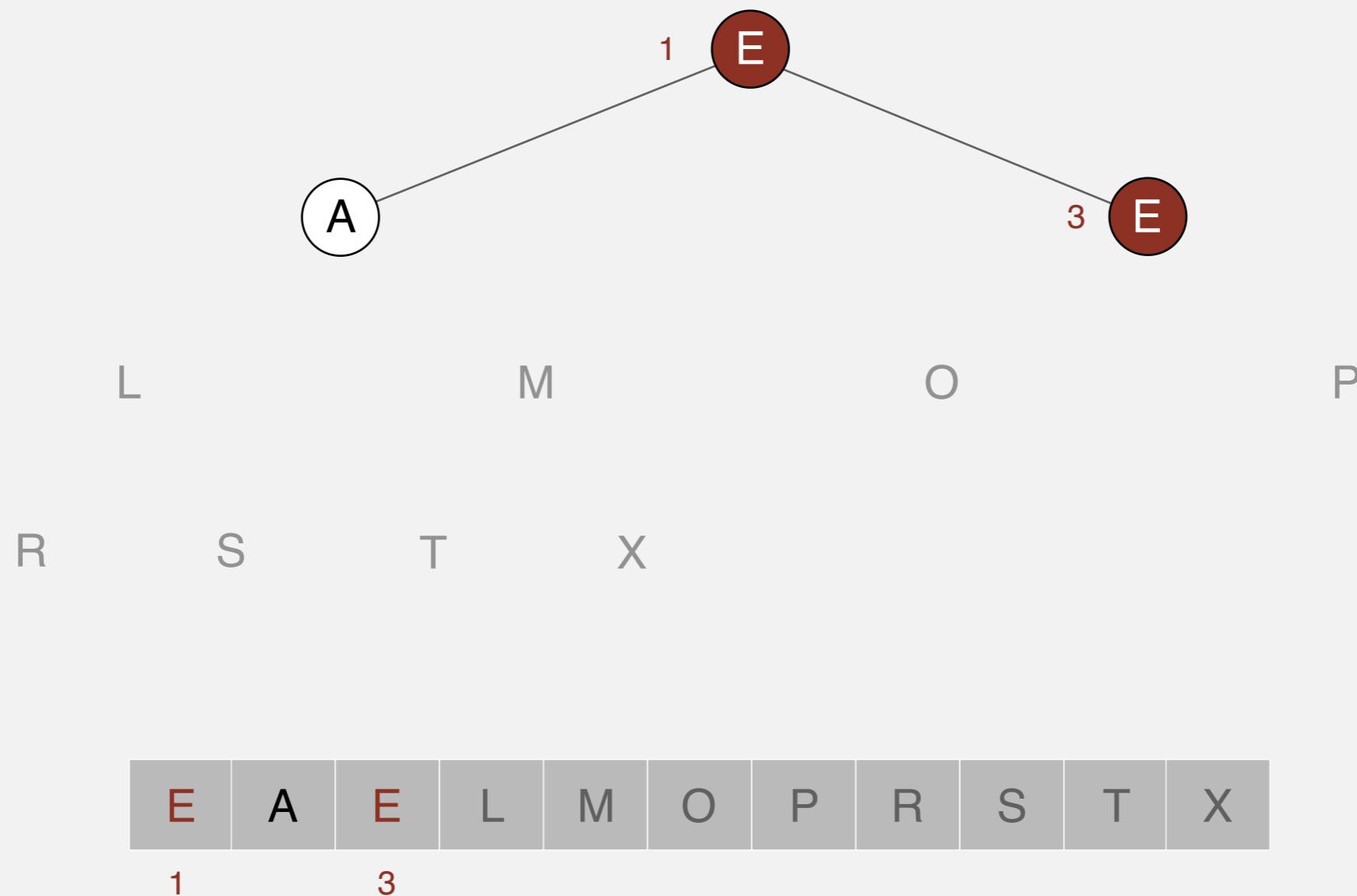
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

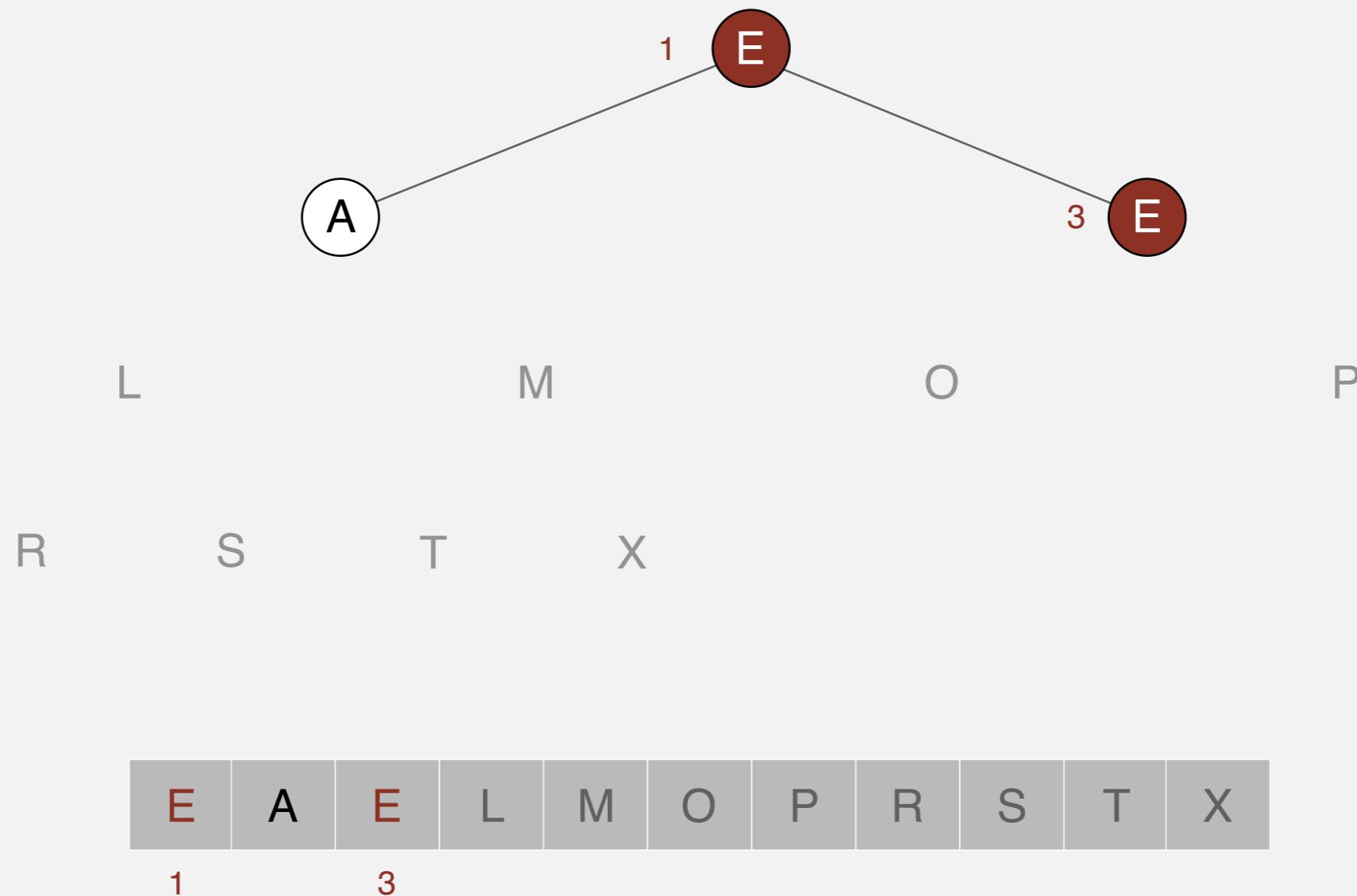
exchange 1 and 3



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

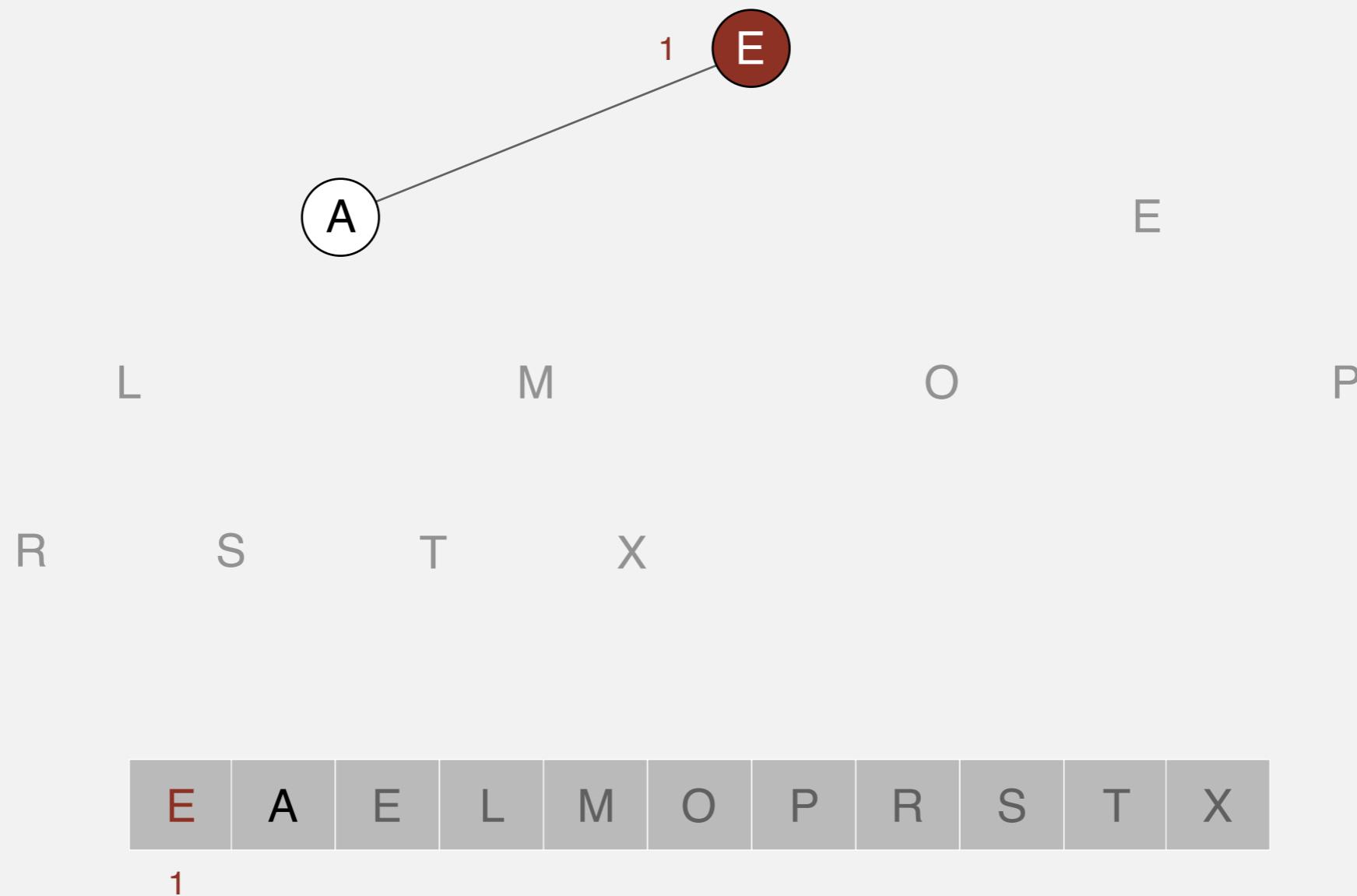
exchange 1 and 3



Heapsort demo

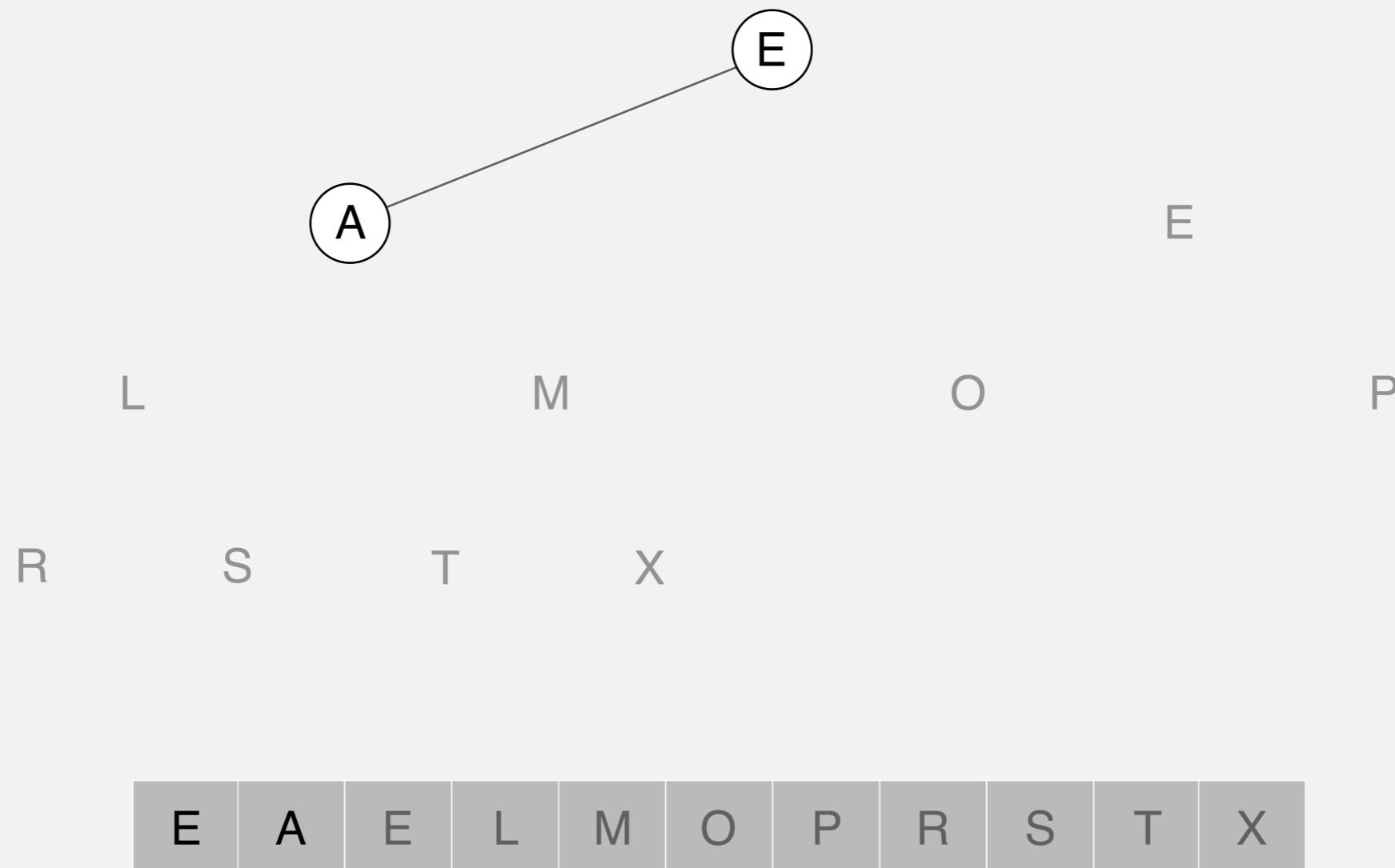
Sortdown. Repeatedly delete the largest remaining item.

sink 1



Heapsort demo

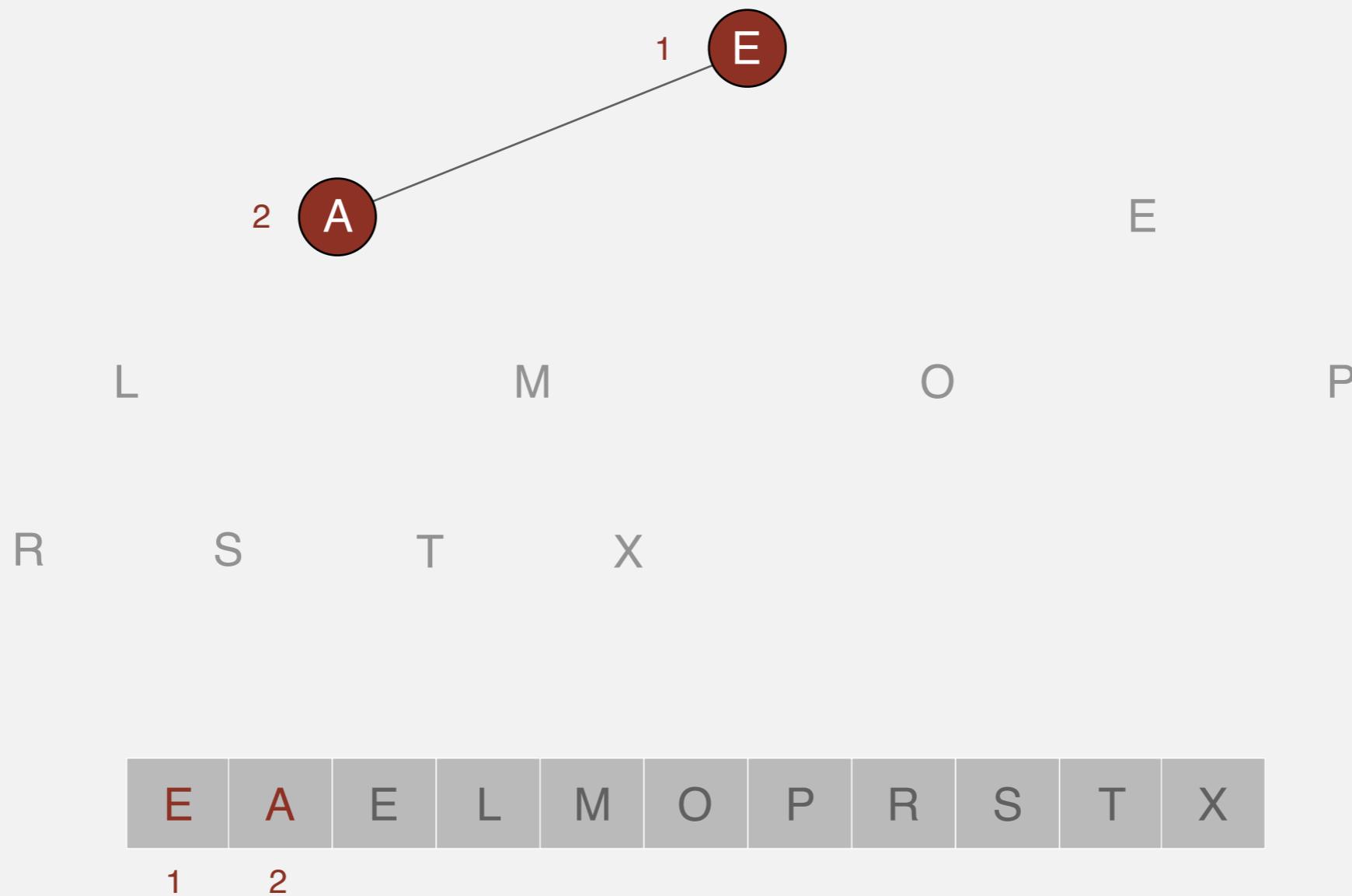
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

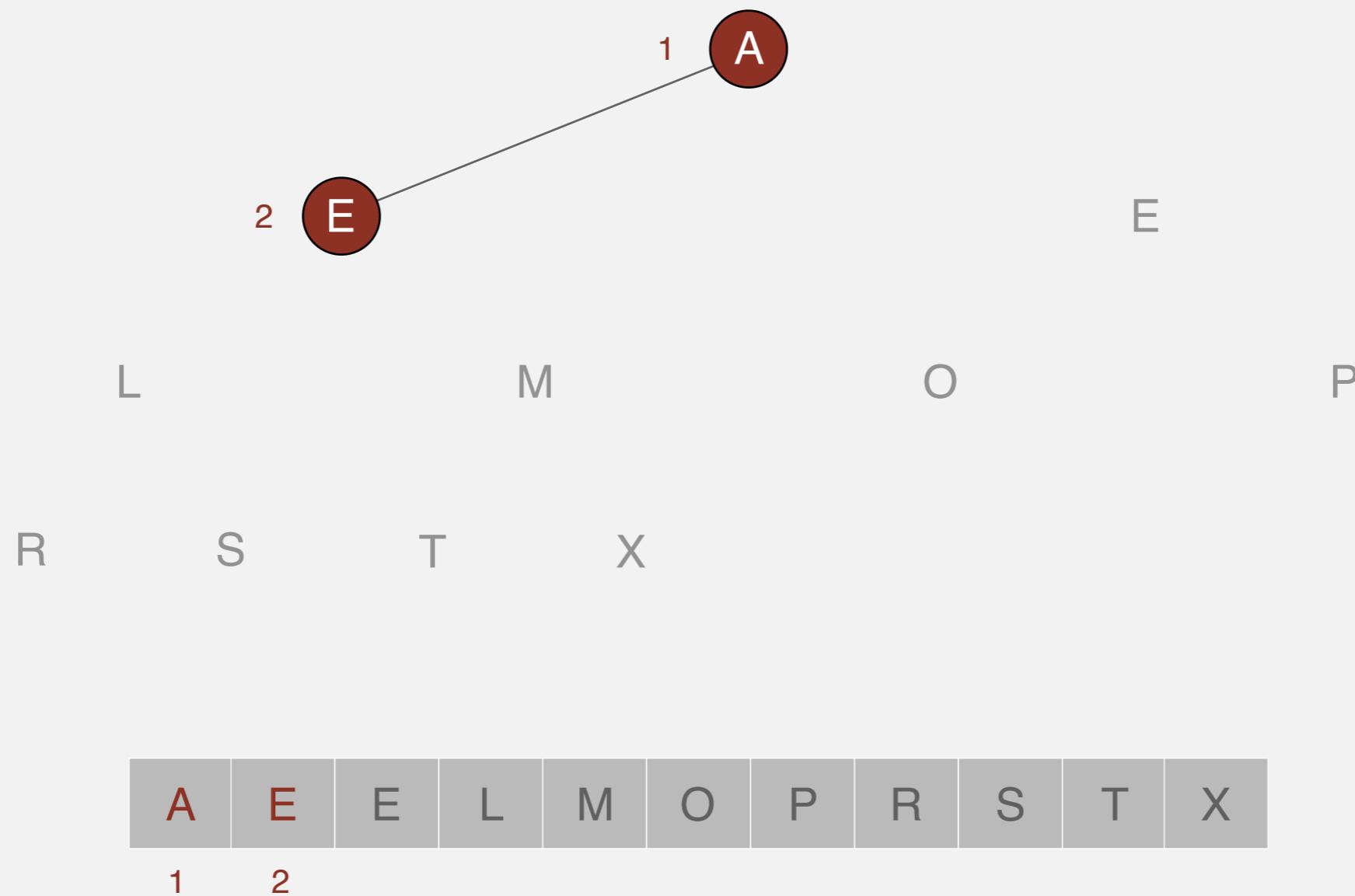
exchange 1 and 2



Heapsort demo

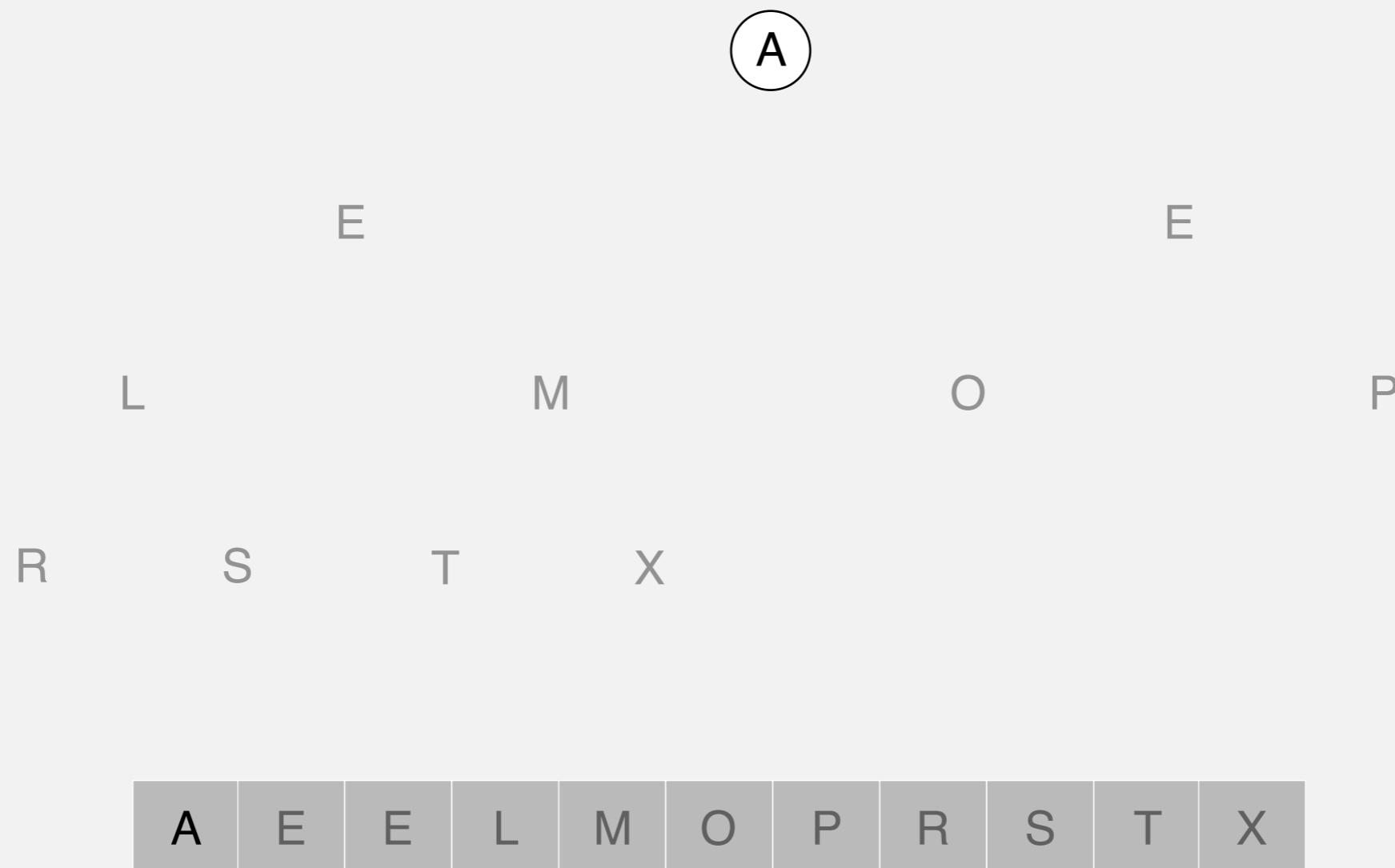
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

end of sortdown phase



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

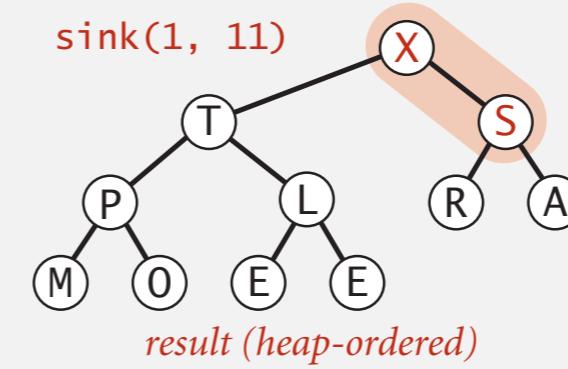
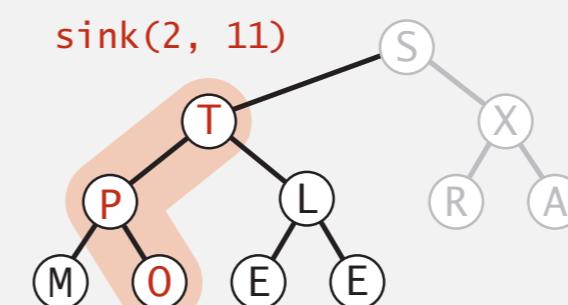
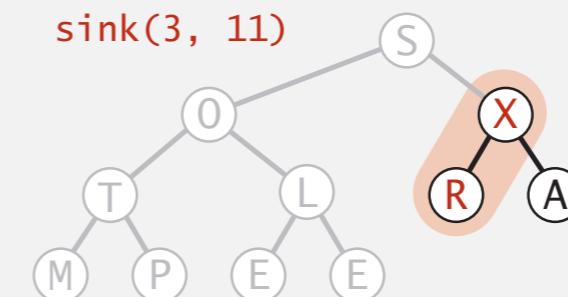
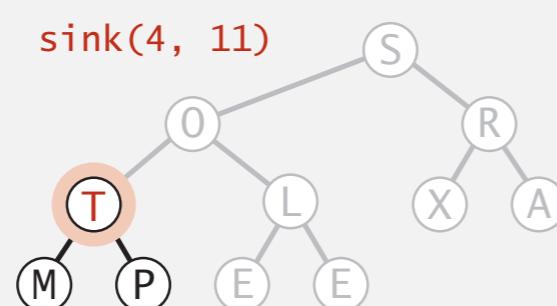
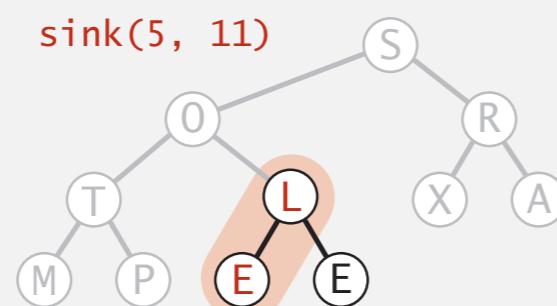
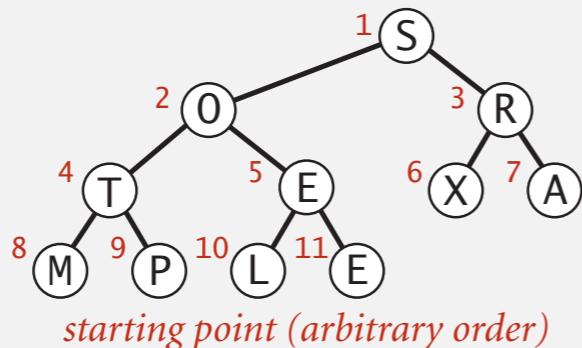
array in sorted order



Heapsort: heap construction

First pass. Build heap using bottom-up method.

```
for (int k = N/2; k >= 1; k--)  
    sink(a, k, N);
```

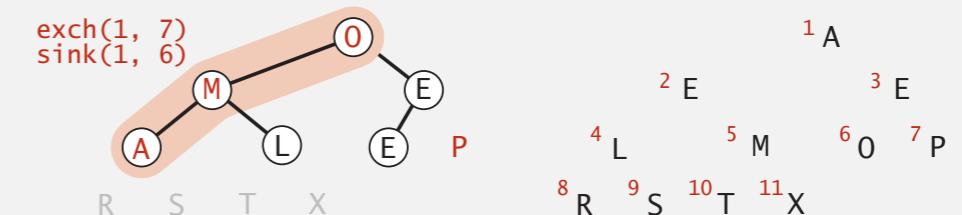
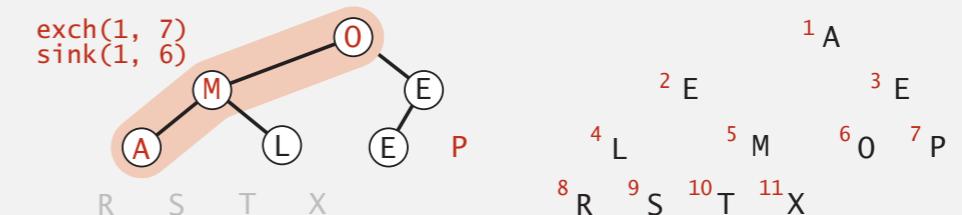
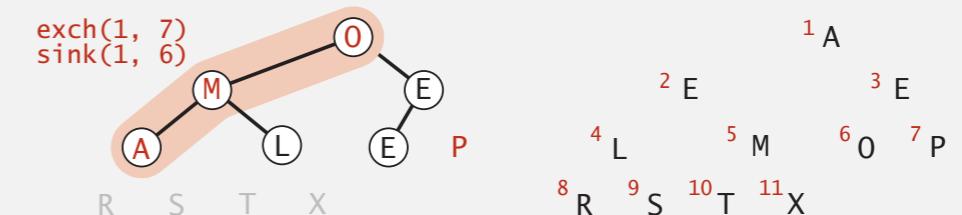
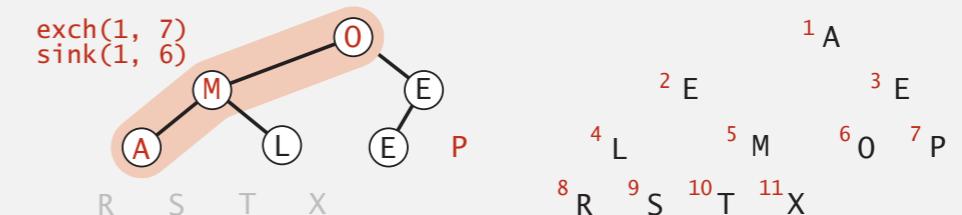
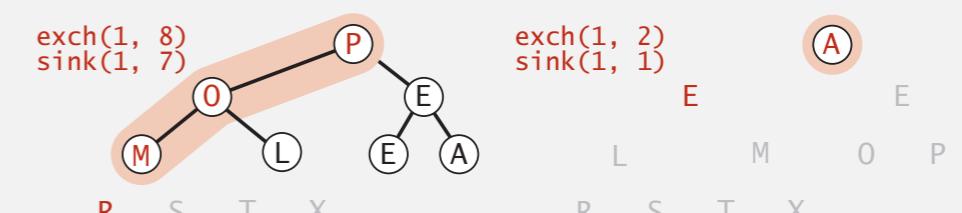
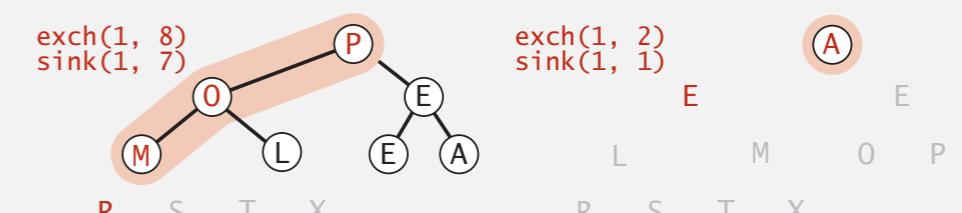
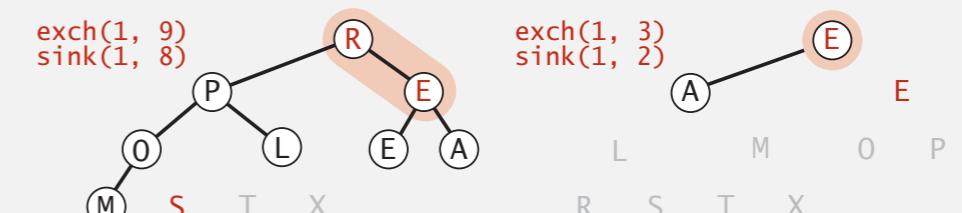
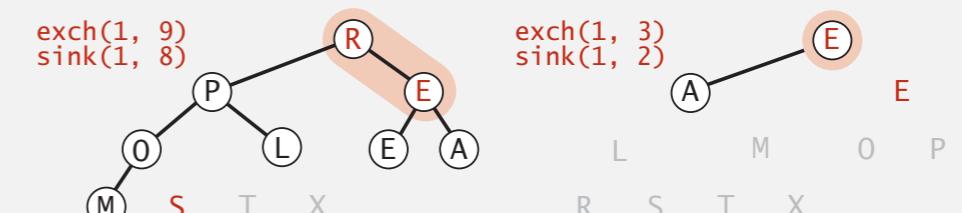
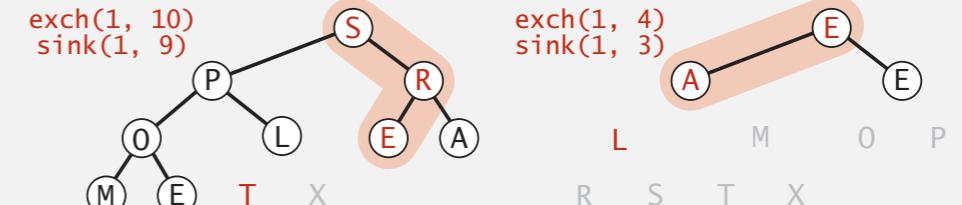
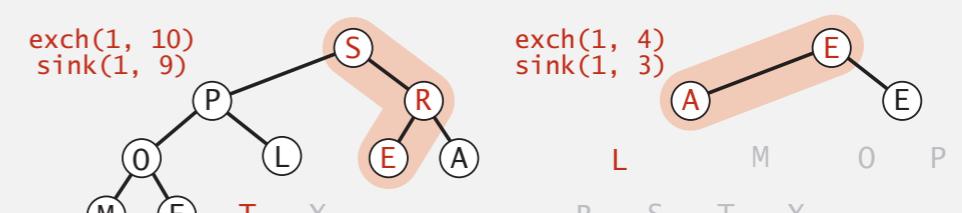
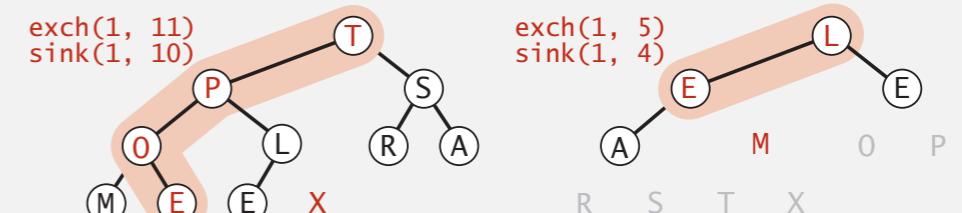
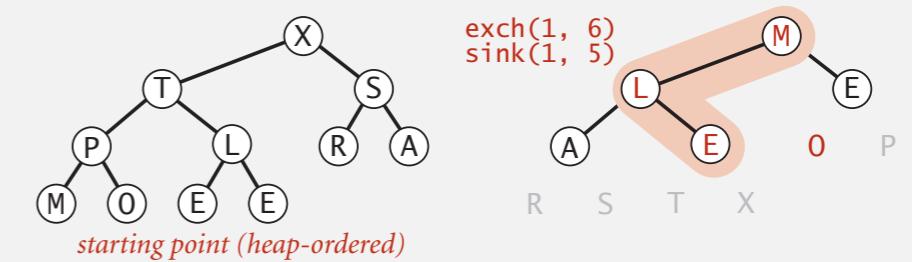


Heapsort: sortdown

Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```



result (sorted)

1 A
2 E
3 E
4 L
5 M
6 O
7 P
8 R
9 S
10 T
11 X

Heapsort: Java implementation

```
public class Heap {  
    public static void sort(Comparable[] a) {  
        int N = a.length;  
        for (int k = N/2; k >= 1; k--)  
            sink(a, k, N);  
        while (N > 1) {  
            exch(a, 1, N);  
            sink(a, 1, --N);  
        }  
    }  
  
    private static void sink(Comparable[] a, int k, int N)  
    { /* as before */ }  
  
    private static boolean less(Comparable[] a, int i, int j)  
    { /* as before */ }  
  
    private static void exch(Object[] a, int i, int j)  
    { /* as before */ }  
}
```

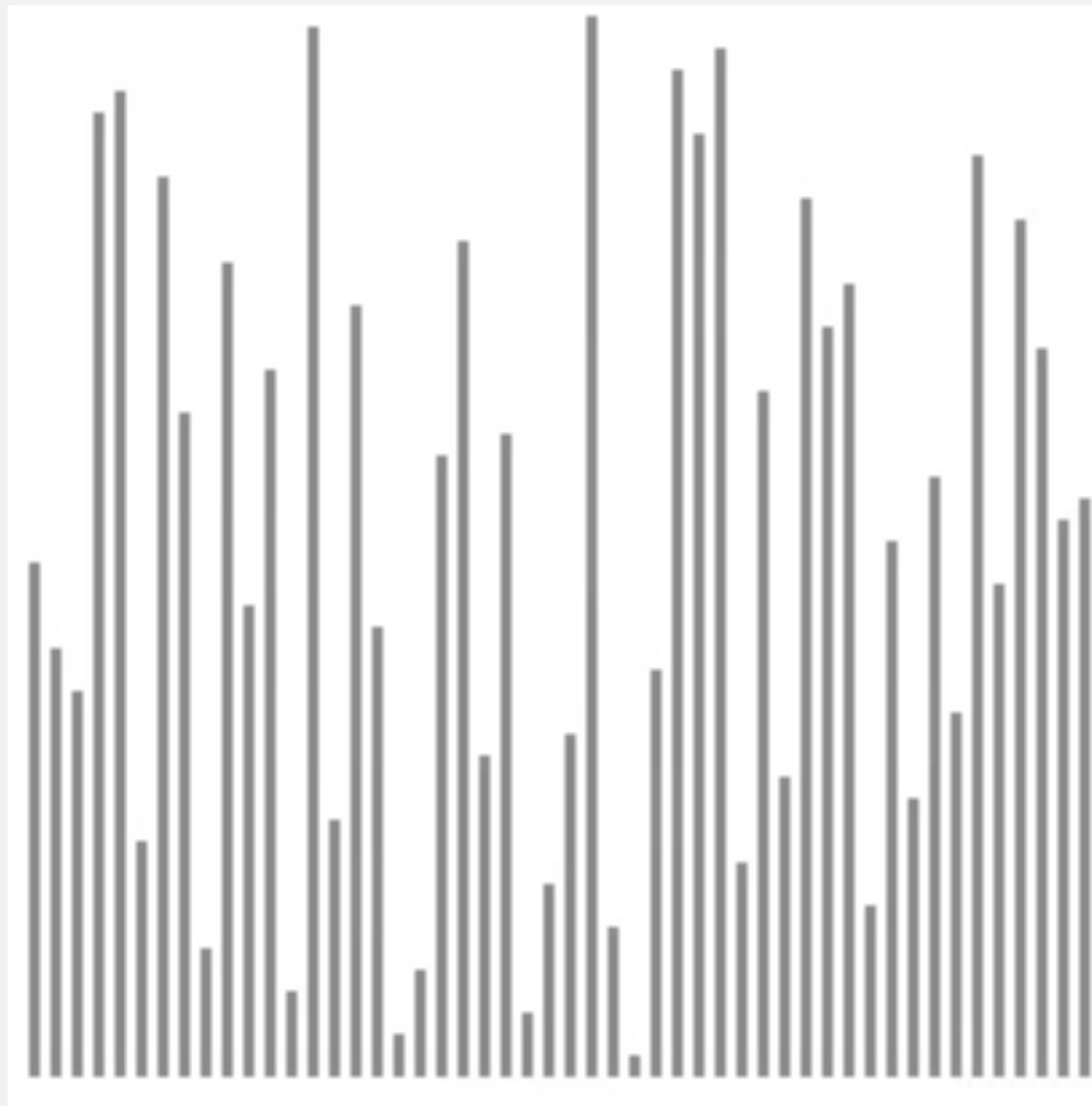
Heapsort: trace

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>		S	0	R	T	E	X	A	M	P	L	E	
11	5	S	0	R	T	L	X	A	M	P	E	E	
11	4	S	0	R	T	L	X	A	M	P	E	E	
11	3	S	0	X	T	L	R	A	M	P	E	E	
11	2	S	T	X	P	L	R	A	M	O	E	E	
11	1	X	T	S	P	L	R	A	M	O	E	E	
<i>heap-ordered</i>		X	T	S	P	L	R	A	M	O	E	E	
10	1	T	P	S	O	L	R	A	M	E	E	X	
9	1	S	P	R	O	L	E	A	M	E	T	X	
8	1	R	P	E	O	L	E	A	M	S	T	X	
7	1	P	O	E	M	L	E	A	R	S	T	X	
6	1	O	M	E	A	L	E	P	R	S	T	X	
5	1	M	L	E	A	E	O	P	R	S	T	X	
4	1	L	E	E	A	M	O	P	R	S	T	X	
3	1	E	A	E	L	M	O	P	R	S	T	X	
2	1	E	A	E	L	M	O	P	R	S	T	X	
1	1	A	E	E	L	M	O	P	R	S	T	X	
<i>sorted result</i>		A	E	E	L	M	O	P	R	S	T	X	

Heapsort trace (array contents just after each sink)

Heapsort animation

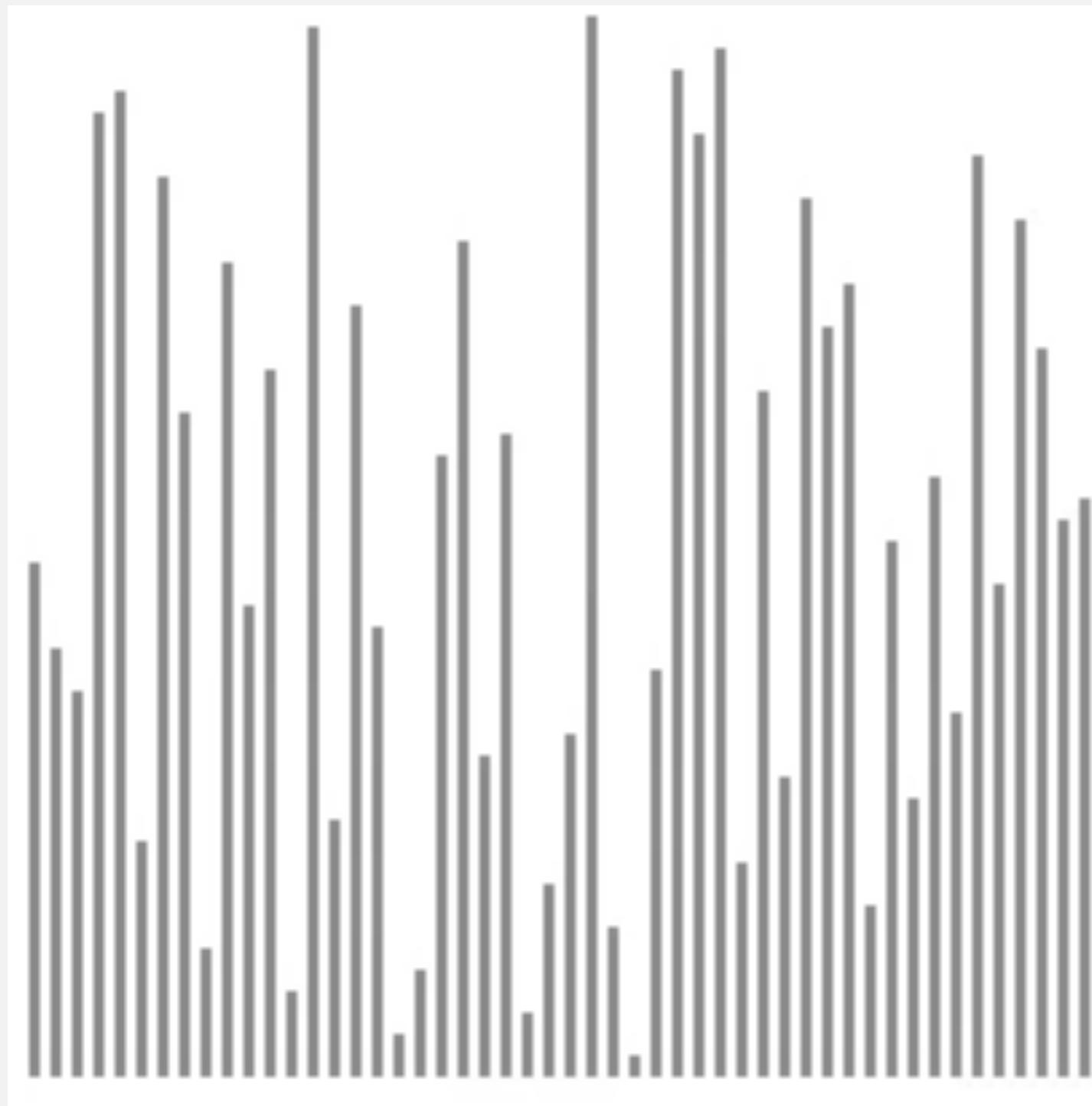
50 random items



- ▲ algorithm position
- in order
- not in order

Heapsort animation

50 random items



- ▲ algorithm position
- in order
- not in order

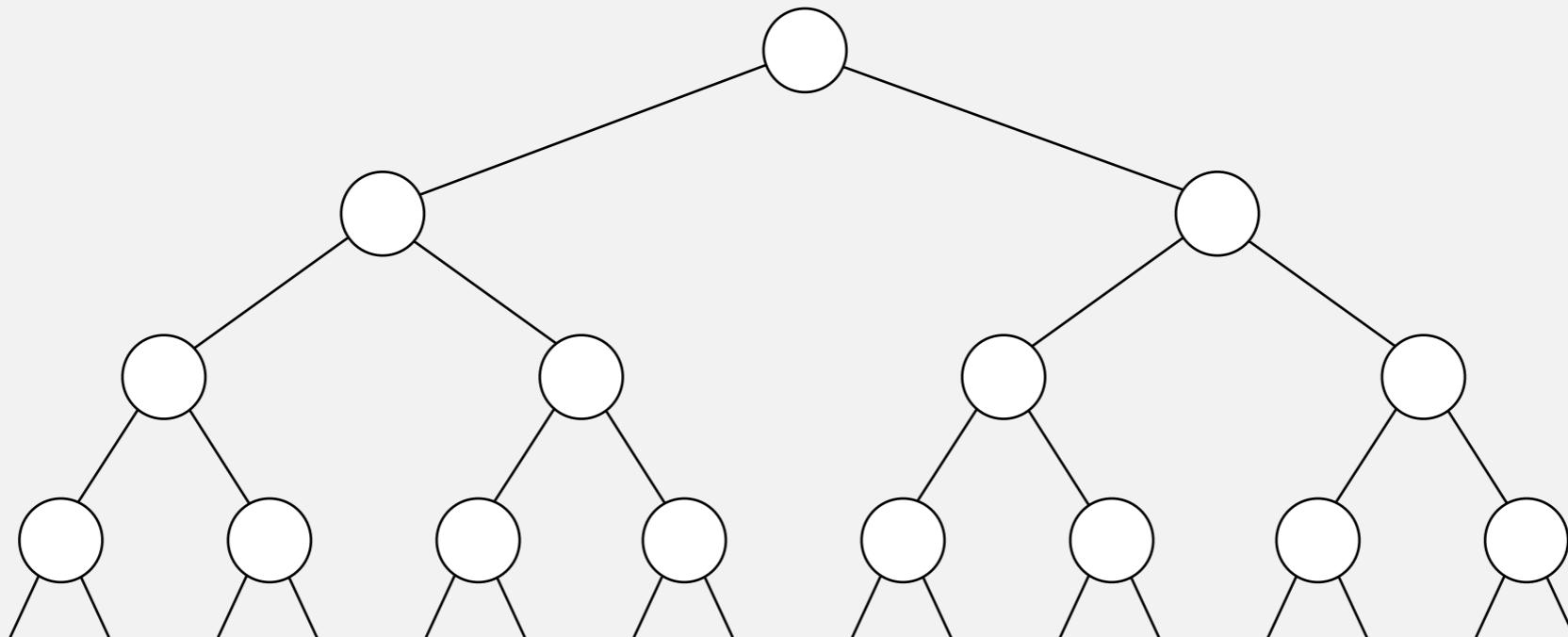
Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Pf sketch. [assume $N = 2^{h+1} - 1$]

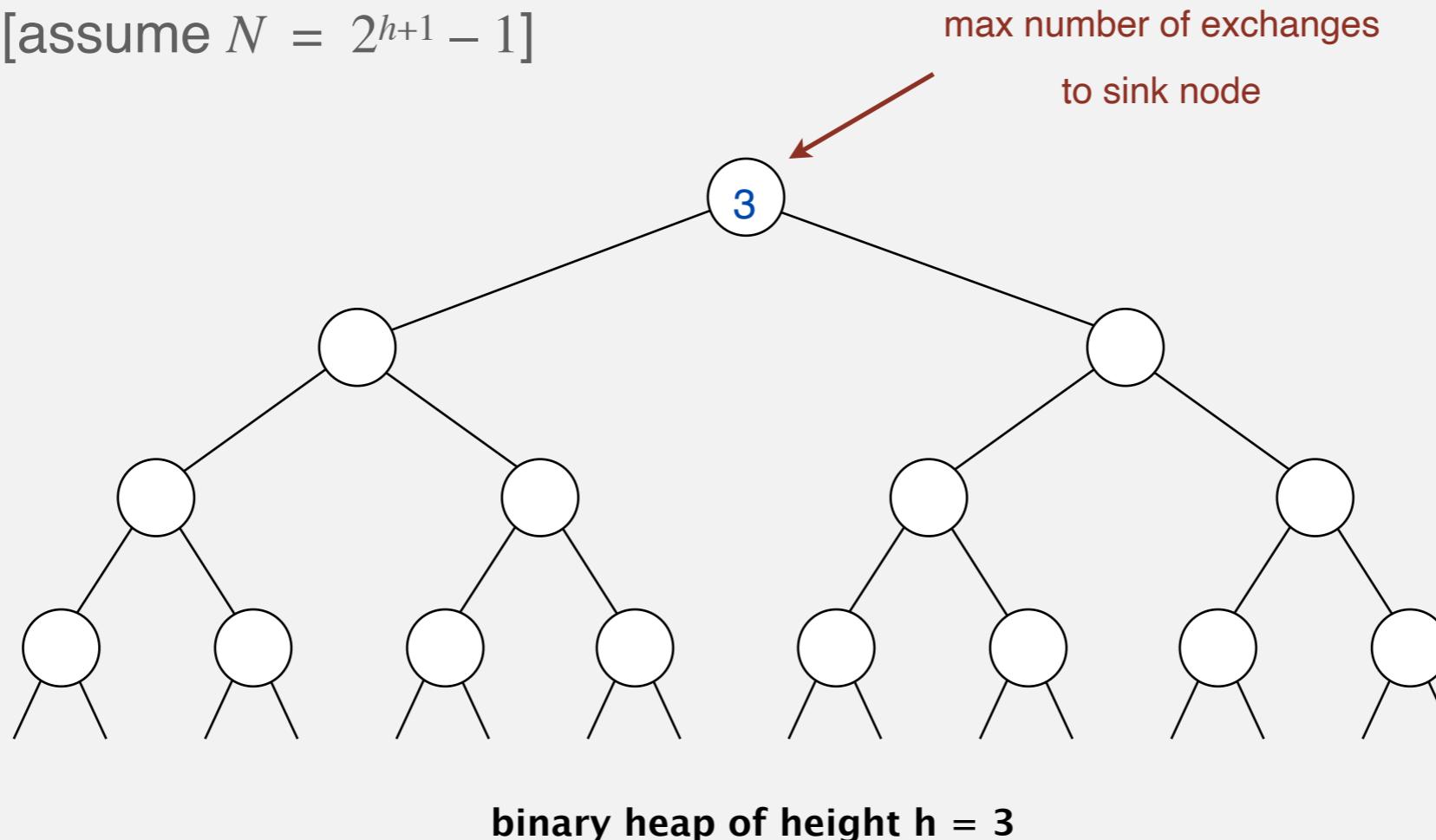


binary heap of height $h = 3$

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

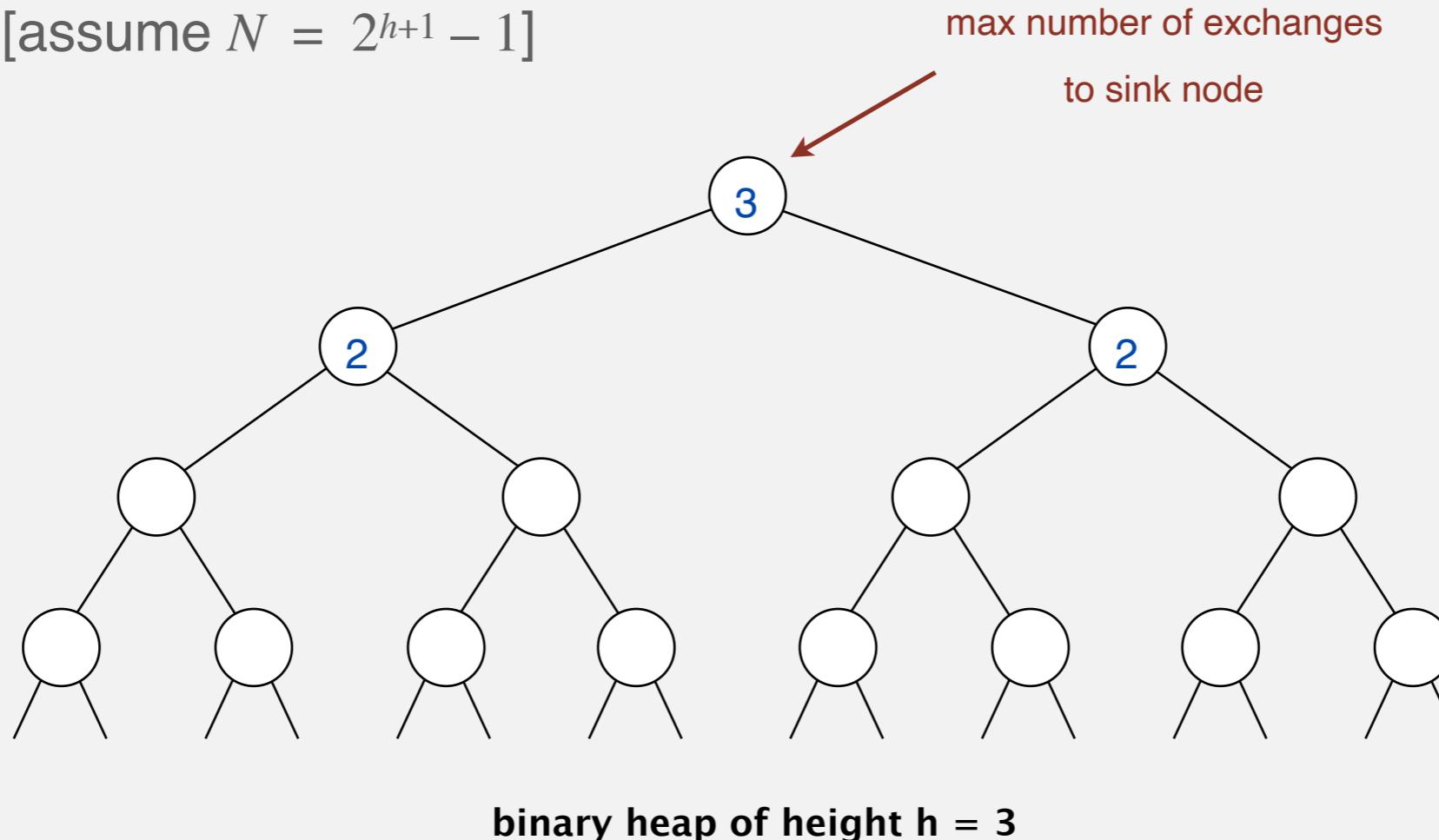
Pf sketch. [assume $N = 2^{h+1} - 1$]



Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

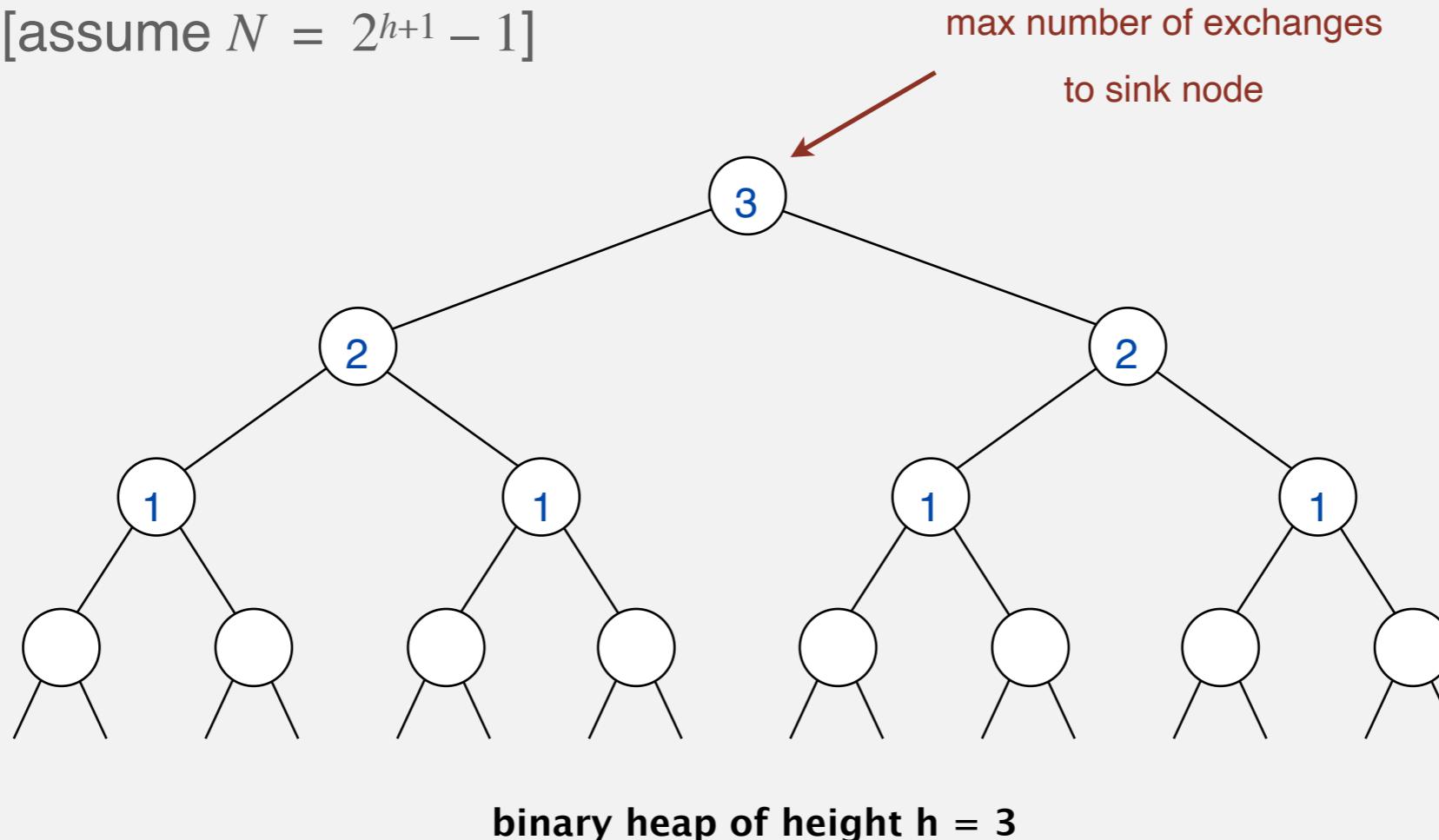
Pf sketch. [assume $N = 2^{h+1} - 1$]



Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

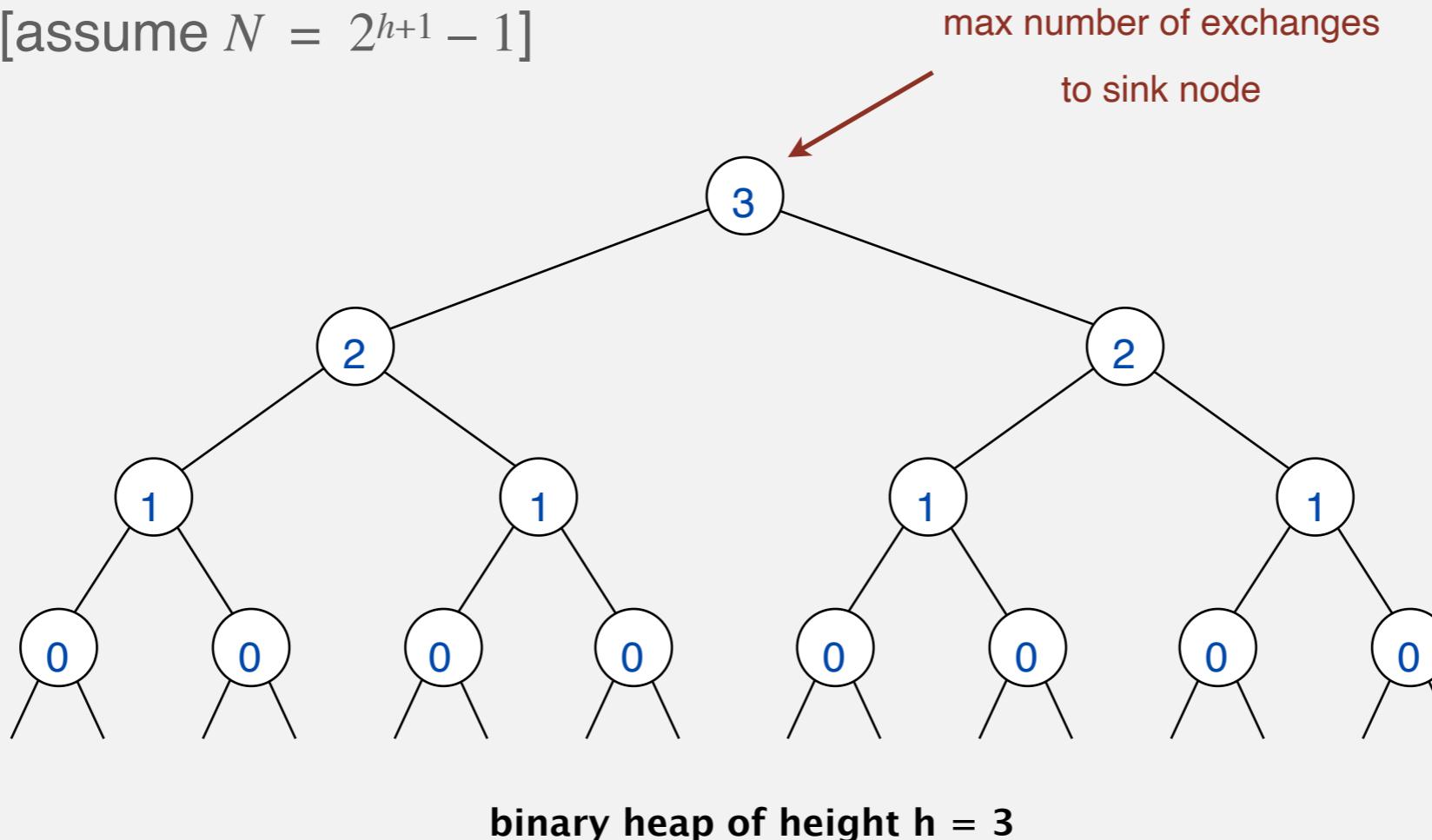
Pf sketch. [assume $N = 2^{h+1} - 1$]



Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

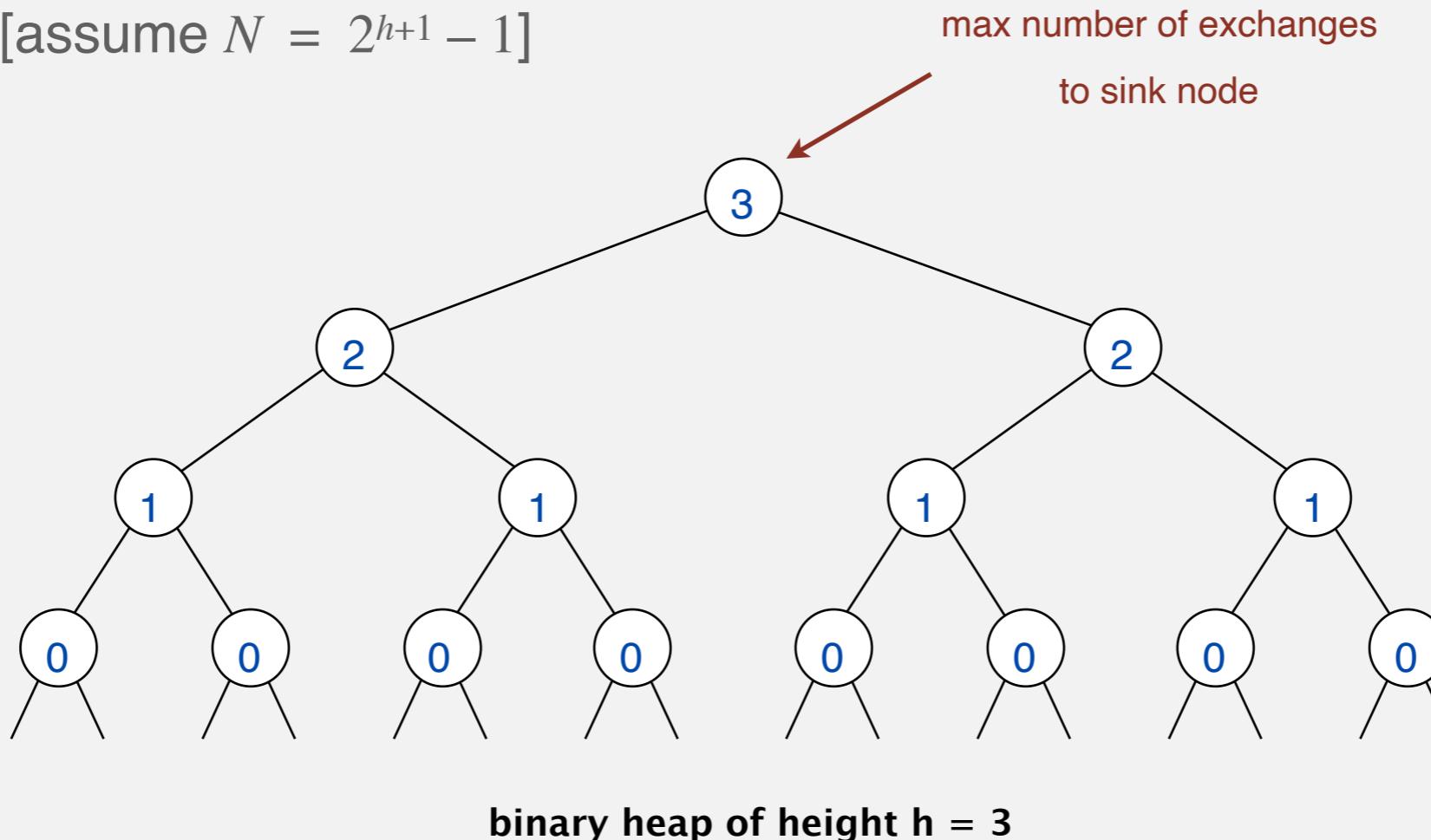
Pf sketch. [assume $N = 2^{h+1} - 1$]



Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Pf sketch. [assume $N = 2^{h+1} - 1$]

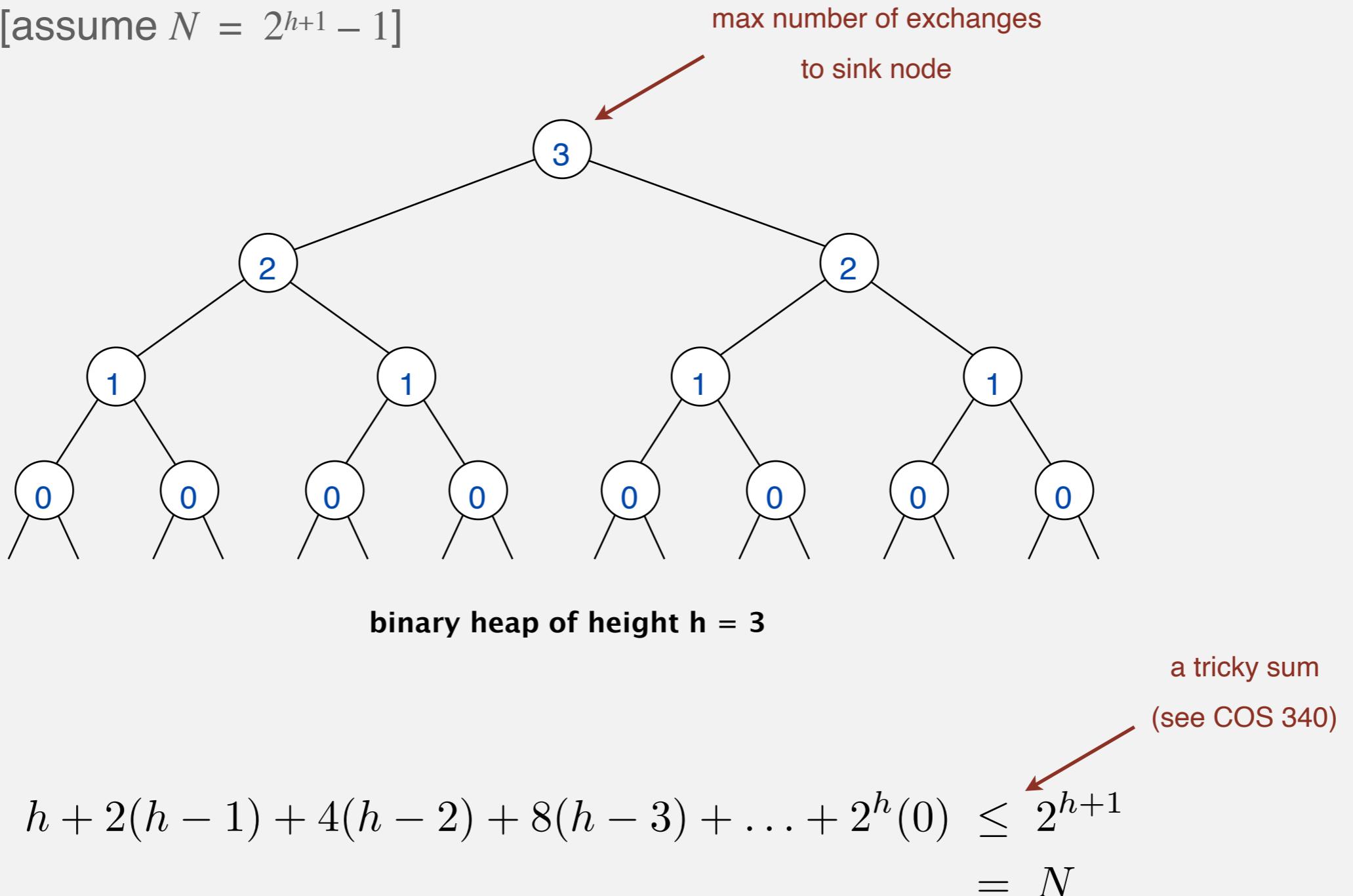


$$h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \dots + 2^h(0)$$

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Pf sketch. [assume $N = 2^{h+1} - 1$]



Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.



algorithm can be improved to $\sim 1 N \lg N$

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.



algorithm can be improved to $\sim 1 N \lg N$

Significance. In-place sorting algorithm with $N \log N$ worst-case.

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.



algorithm can be improved to $\sim 1 N \lg N$

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space.

← in-place merge possible, not practical

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.



algorithm can be improved to $\sim N \lg N$

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← $N \log N$ worst-case quicksort possible, not practical

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.



algorithm can be improved to $\sim N \lg N$

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← $N \log N$ worst-case quicksort possible, not practical
- Heapsort: yes!

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.



algorithm can be improved to $\sim N \lg N$

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← $N \log N$ worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, **but**:

- Inner loop longer than quicksort's.
- Makes poor use of cache.
- Not stable.



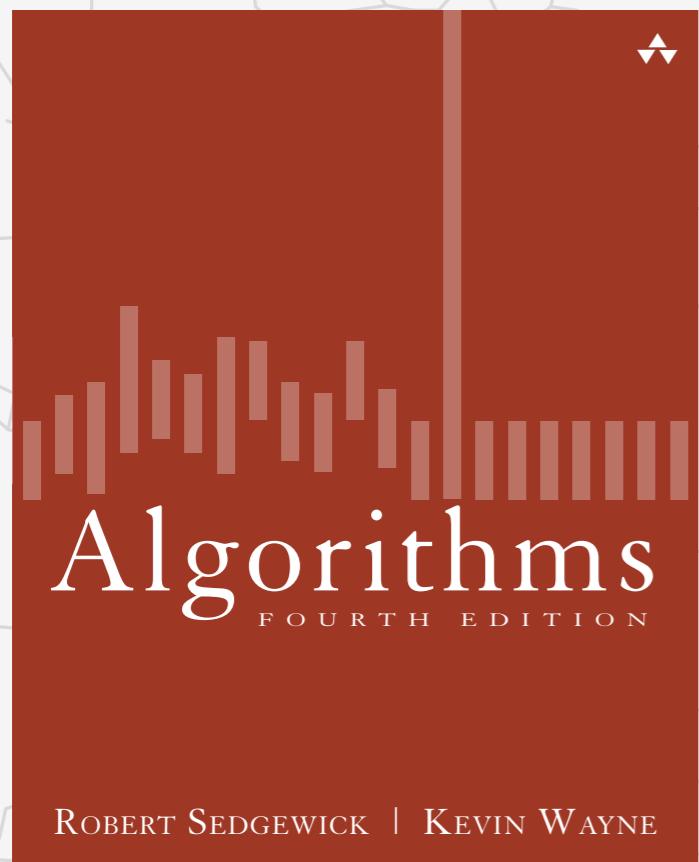
advanced tricks for improving

Sorting algorithms: 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
timsort		✓	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
quick	✓		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-way quick	✓		N	$2 N \ln N$	$\frac{1}{2} N^2$	improves quicksort when duplicate keys
heap	✓		N	$2 N \lg N$	$2 N \lg N$	$N \log N$ guarantee; in-place
?	✓	✓	N	$N \lg N$	$N \lg N$	holy sorting grail

Algorithms

FARAAZ SARESHWALA



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

3.1 SYMBOL TABLES

- ▶ API
- ▶ Elementary implementations
- ▶ Ordered operations

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

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.

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

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N - 1$.

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N - 1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

```
hasNiceSyntaxForAssociativeArrays["Python"] = True  
hasNiceSyntaxForAssociativeArrays["Java"] = False
```

legal Python code

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N - 1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an
associative array

```
hasNiceSyntaxForAssociativeArrays["Python"] = True  
hasNiceSyntaxForAssociativeArrays["Java"] = False
```

legal Python code

Symbol tables: context

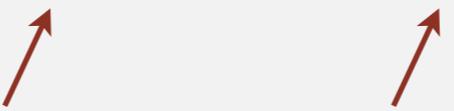
Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N - 1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an
associative array every object is an
associative array



```
hasNiceSyntaxForAssociativeArrays["Python"] = True  
hasNiceSyntaxForAssociativeArrays["Java"] = False
```

legal Python code

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N - 1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an
associative array every object is an
associative array table is the only
primitive data structure

```
hasNiceSyntaxForAssociativeArrays["Python"] = True  
hasNiceSyntaxForAssociativeArrays["Java"] = False
```

legal Python code

Basic symbol table API

Associative array abstraction. Associate one value with each key.

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

```
ST()
```

create an empty symbol table

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

put key-value pair into the table ← a[key] = val;

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

value paired with key ← a[key]

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

is there a value paired with key?

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

remove key (and its value) from table

```
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. ← Java allows null value
- Method get() returns null if key not present.
- Method put() overwrites old value with new value.

Conventions

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

Conventions

- Values are not null. ← Java allows null value
- 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; }
```

Conventions

- Values are not null. ← Java allows null value
- 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); }
```

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));
}
```

output

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

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

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
```



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

- ← tiny example
(60 words, 20 distinct)
- ← real example
(135,635 words, 10,769 distinct)
- ← real example
(21,191,455 words, 534,580 distinct)

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.1 SYMBOL TABLES

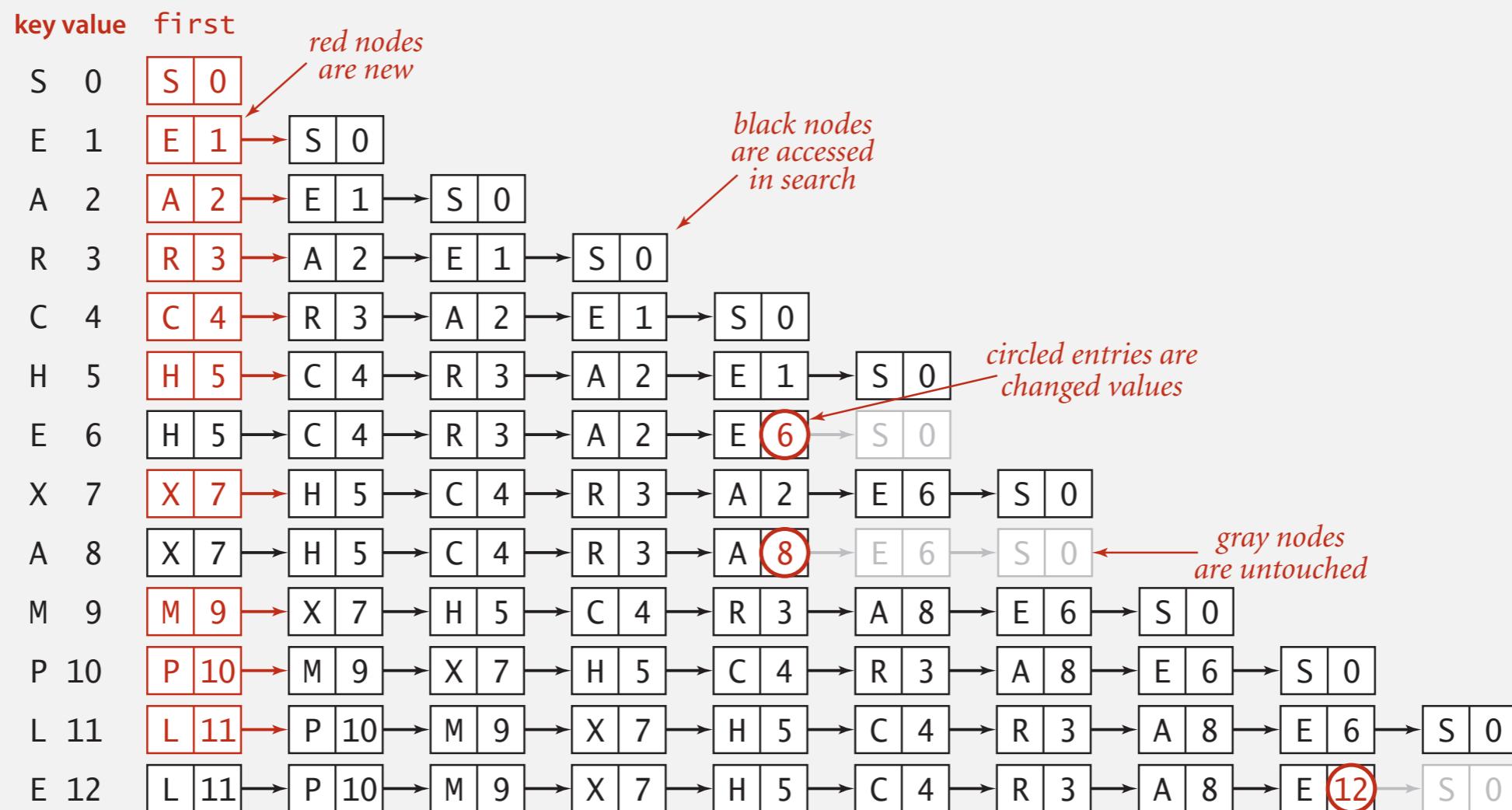
- ▶ API
- ▶ elementary implementations
- ▶ ordered operations

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	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	equals()

Challenge. Efficient implementations of both search and insert.

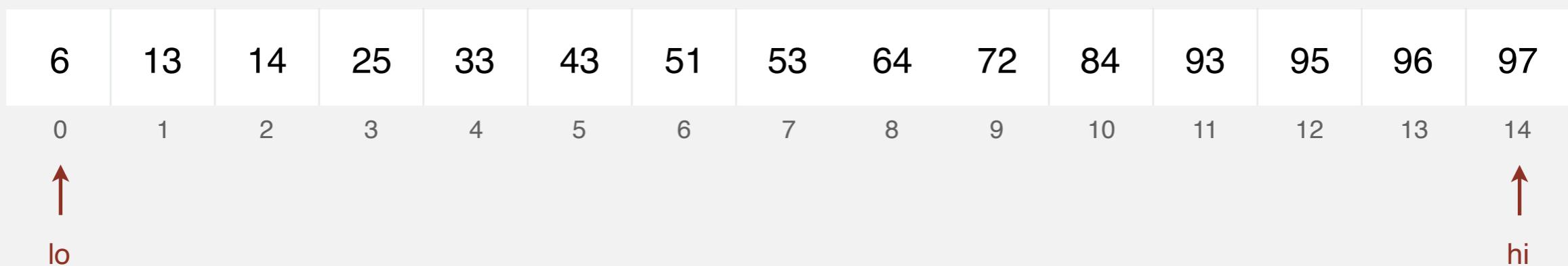
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



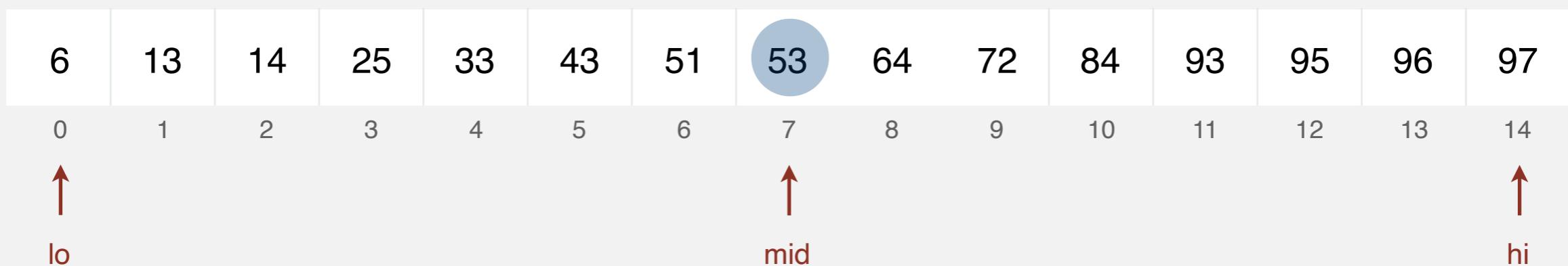
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
 - Too big, go right.
 - Equal, found.

successful search for 33



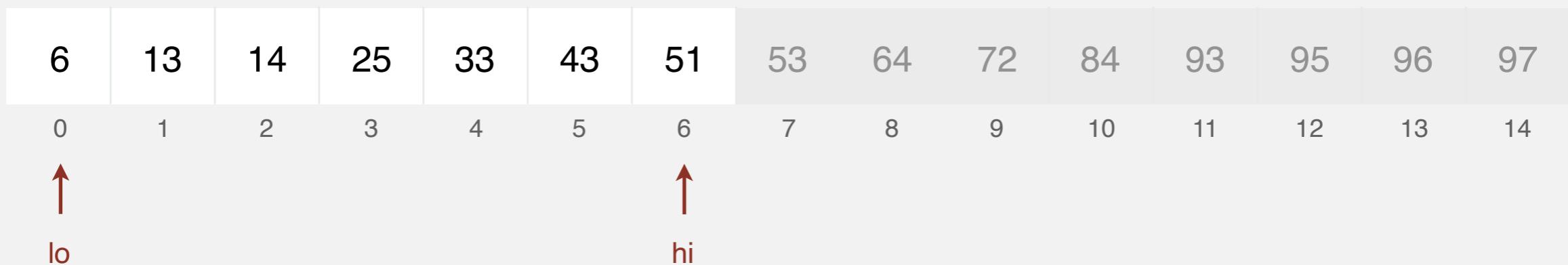
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
 - Too big, go right.
 - Equal, found.

successful search for 33



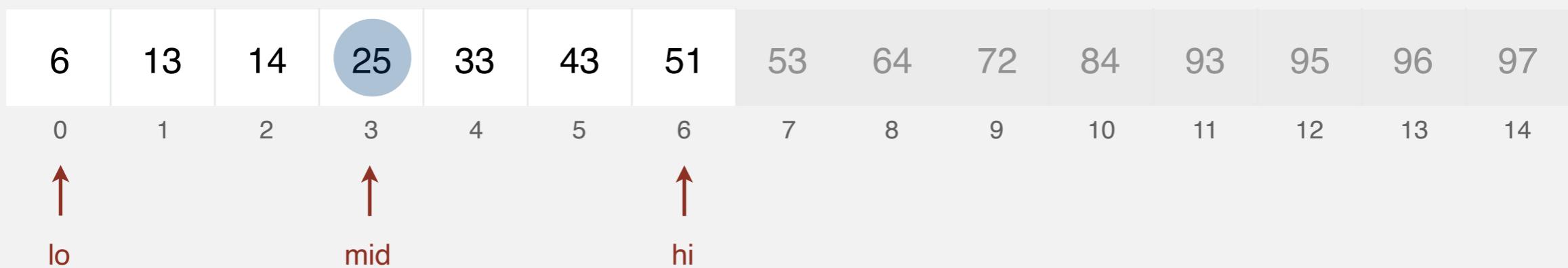
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



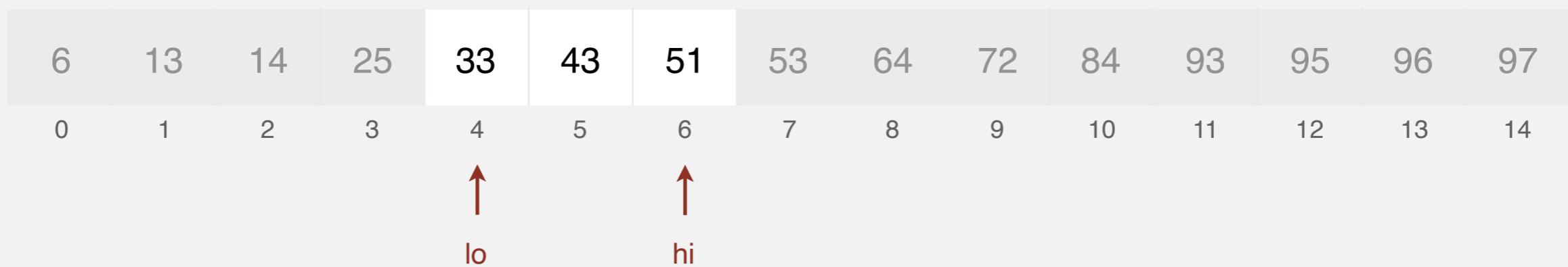
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



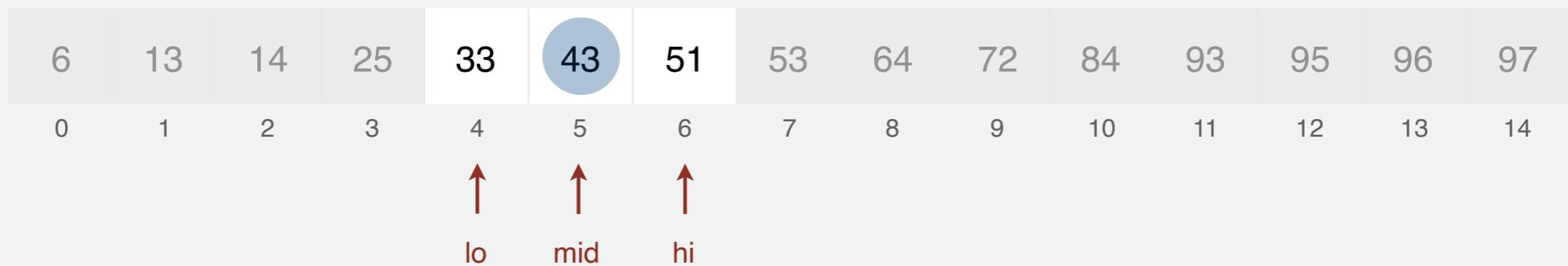
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



Binary search demo

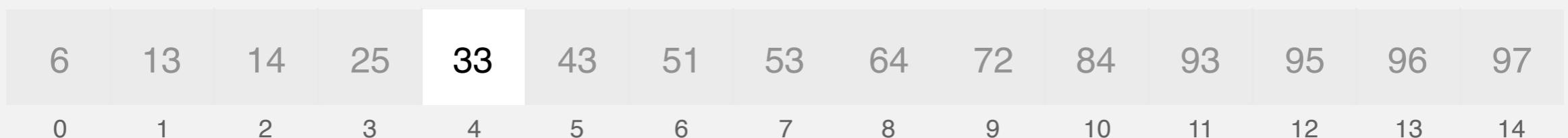
Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

lo = hi
↓



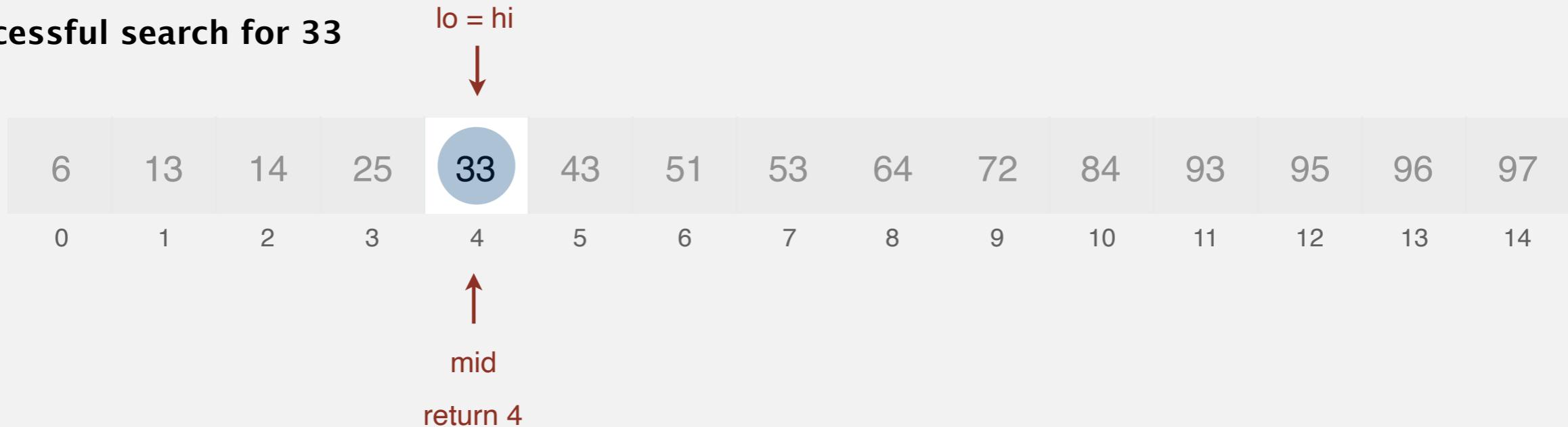
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



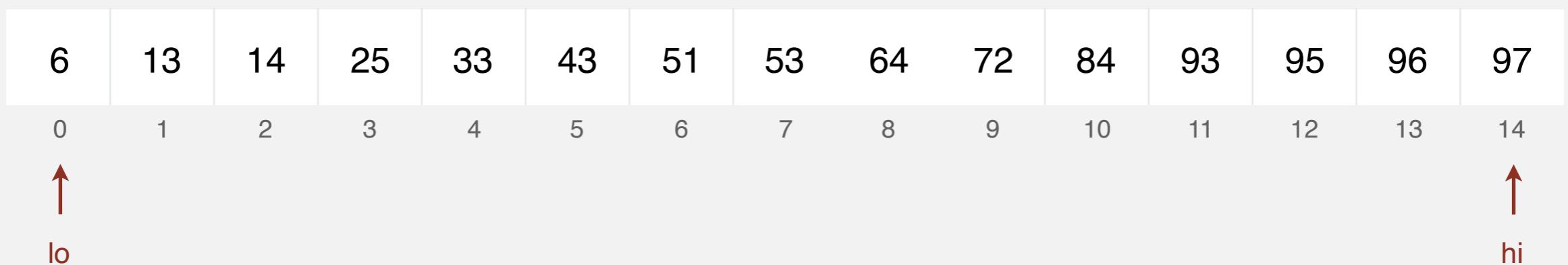
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



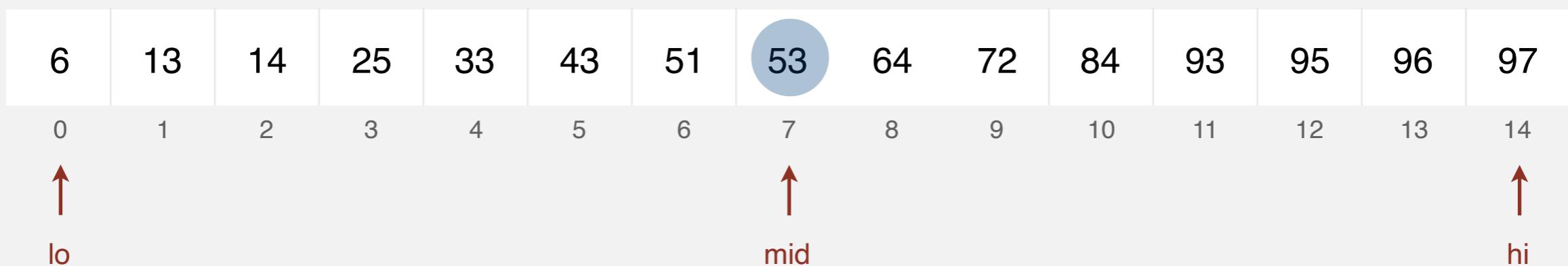
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
 - Too big, go right.
 - Equal, found.

unsuccessful search for 34



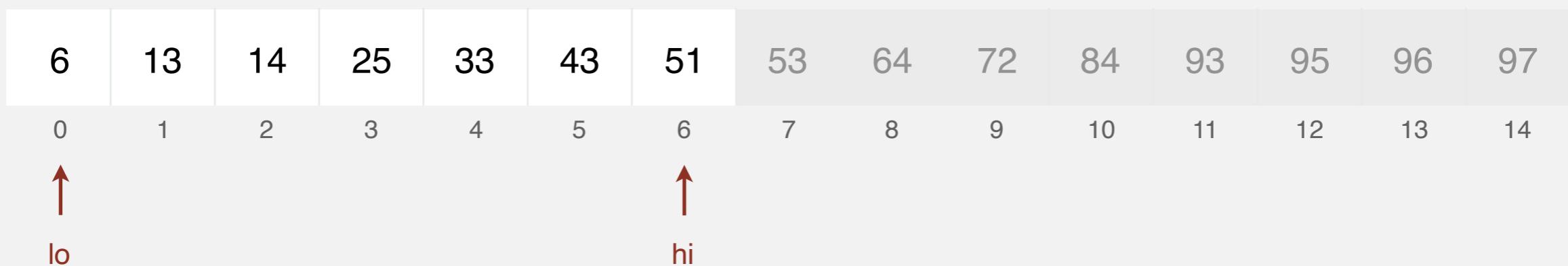
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



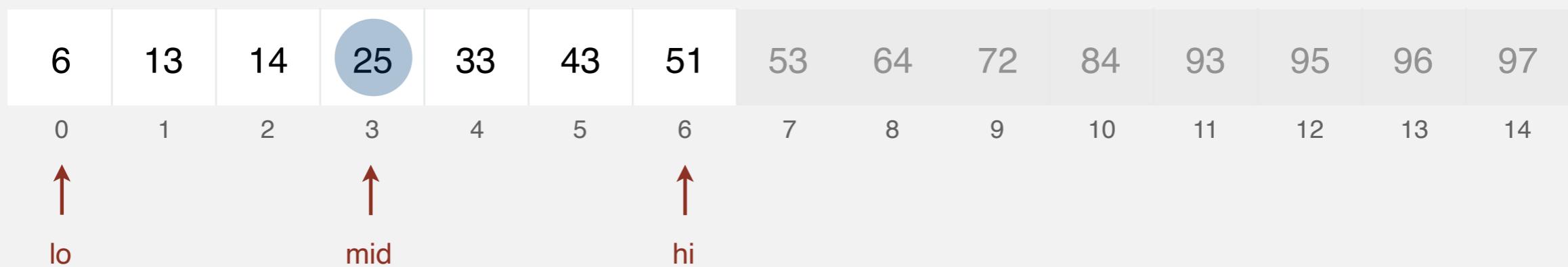
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



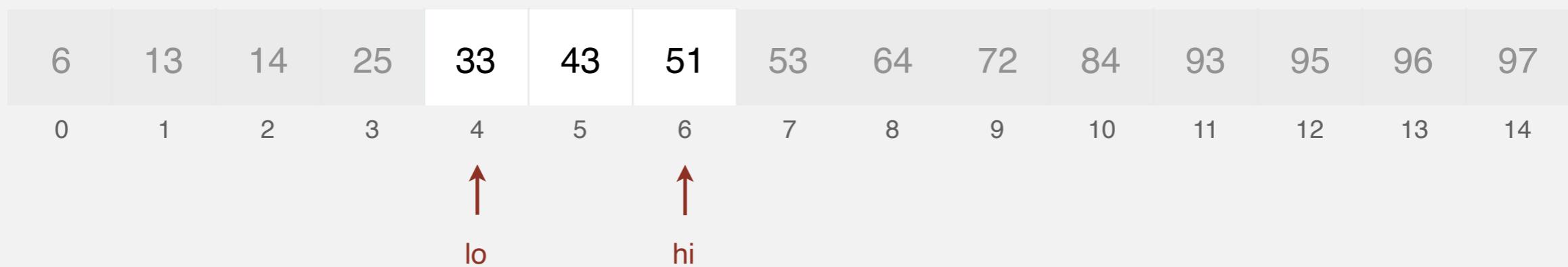
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



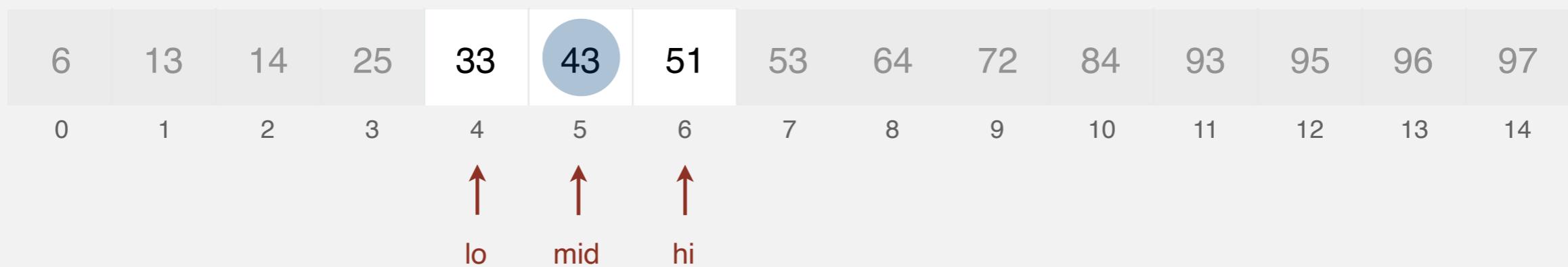
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



Binary search demo

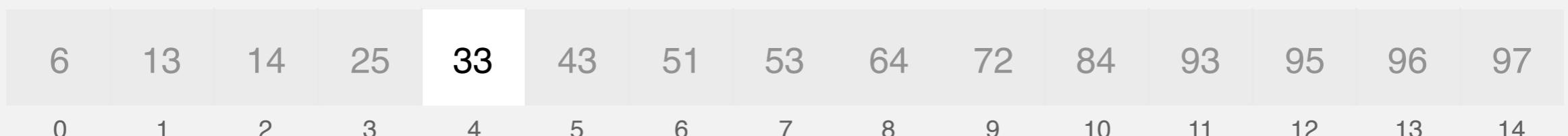
Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

lo = hi
↓



Binary search demo

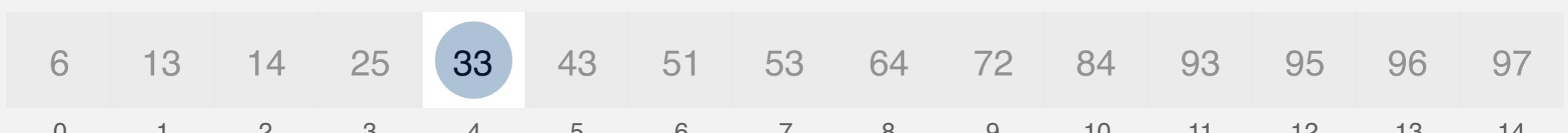
Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

lo = hi
↓



↑
mid

return -1

Binary search: Java implementation

Trivial to implement?

Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946.

Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.

Binary search: Java implementation

Trivial to implement?

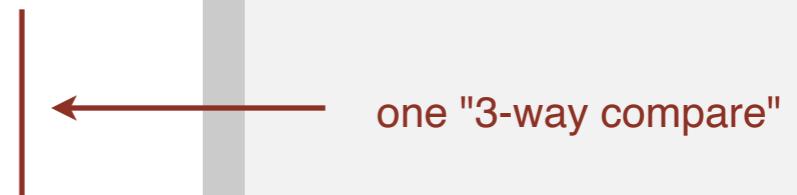
- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's Arrays.binarySearch() discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```



Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```



Invariant. If `key` appears in the array `a[]`, then $a[lo] \leq key \leq a[hi]$.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$T(N) \leq T(N/2) + 1 \quad [\text{ given }]$$

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [\text{ given }] \\ &\leq T(N/4) + 1 + 1 && [\text{ apply recurrence to first term }] \end{aligned}$$

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [\text{given}] \\ &\leq T(N/4) + 1 + 1 && [\text{apply recurrence to first term}] \\ &\leq T(N/8) + 1 + 1 + 1 && [\text{apply recurrence to first term}] \end{aligned}$$

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [\text{given}] \\ &\leq T(N/4) + 1 + 1 && [\text{apply recurrence to first term}] \\ &\leq T(N/8) + 1 + 1 + 1 && [\text{apply recurrence to first term}] \\ &\vdots \end{aligned}$$

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [\text{ given }] \\ &\leq T(N/4) + 1 + 1 && [\text{ apply recurrence to first term }] \\ &\leq T(N/8) + 1 + 1 + 1 && [\text{ apply recurrence to first term }] \\ &\vdots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && [\text{ stop applying, } T(1) = 1] \end{aligned}$$

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half
(floored division)



possible to implement with one
2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [\text{given}] \\ &\leq T(N/4) + 1 + 1 && [\text{apply recurrence to first term}] \\ &\leq T(N/8) + 1 + 1 + 1 && [\text{apply recurrence to first term}] \\ &\vdots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && [\text{stop applying, } T(1) = 1] \\ &= 1 + \lg N \end{aligned}$$

Binary search in an ordered array

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

keys []									
0	1	2	3	4	5	6	7	8	9
A	C	E	H	L	M	P	R	S	X

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
A	C	E	H	L	M	P	R	S	X

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			
A C E H L M P R S X												
successful search for P												
lo	hi	m	A	C	E	H	L	M	P	R	S	X
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										<i>entries in black are a[lo..hi]</i>		
										<i>entry in red is a[m]</i>		
										<i>loop exits with keys[m] = P: return 6</i>		

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
5	9	7	A	C	E	H	L	M	P	R
5	6	5	A	C	E	H	L	M	P	R
6	6	6	A	C	E	H	L	M	P	R
unsuccessful search for Q										
lo hi m										
0	9	4	A	C	E	H	L	M	P	R
5	9	7	A	C	E	H	L	M	P	R
5	6	5	A	C	E	H	L	M	P	R
7	6	6	A	C	E	H	L	M	P	R

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

Binary search: trace of standard indexing client

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

	keys[]										N	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	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

Elementary ST implementations: summary

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	<code>equals()</code>
binary search (ordered array)	$\log N$	N	$\log N$	$N / 2$	<code>compareTo()</code>

Challenge. Efficient implementations of both search and insert.

Ordered symbol table API

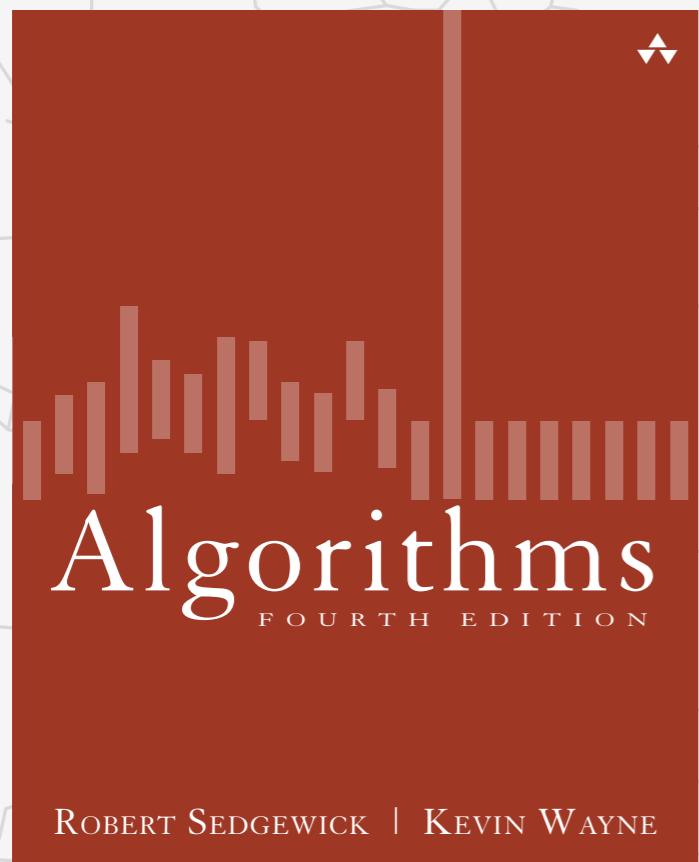
```
public class ST<Key extends Comparable<Key>, Value>
```

...

Key	min()	<i>smallest key</i>
Key	max()	<i>largest key</i>
Key	floor(Key key)	<i>largest key less than or equal to key</i>
Key	ceiling(Key key)	<i>smallest key greater than or equal to key</i>
int	rank(Key key)	<i>number of keys less than key</i>
Key	select(int k)	<i>key of rank k</i>
void	deleteMin()	<i>delete smallest key</i>
void	deleteMax()	<i>delete largest key</i>
int	size(Key lo, Key hi)	<i>number of keys between lo and hi</i>
Iterable<Key>	keys()	<i>all keys, in sorted order</i>
Iterable<Key>	keys(Key lo, Key hi)	<i>keys between lo and hi, in sorted order</i>

Algorithms

FARAAZ SARESHWALA



3.2 BINARY SEARCH TREES

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

BASED ON SLIDES FROM ROBERT SEDGEWICK AND KEVIN WAYNE

Binary search trees

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

Binary search trees

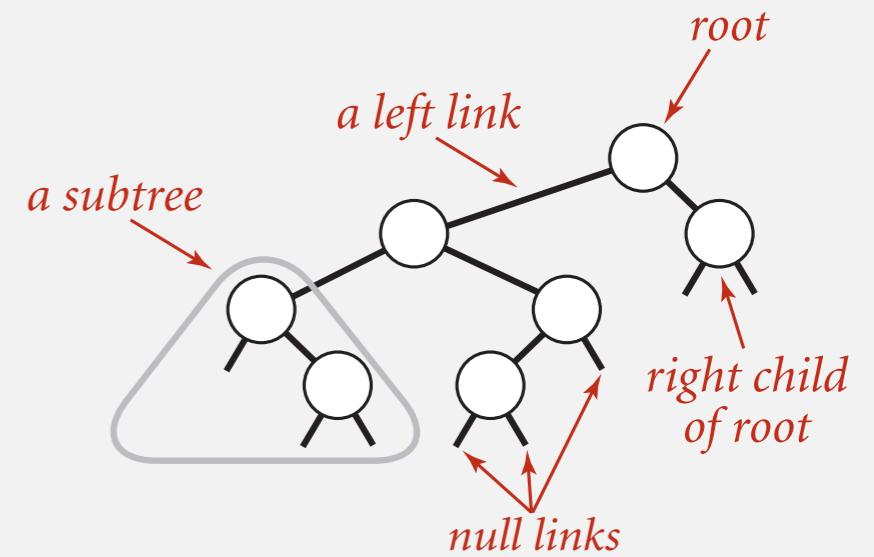
Definition. A BST is a **binary tree** in **symmetric order**.

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).

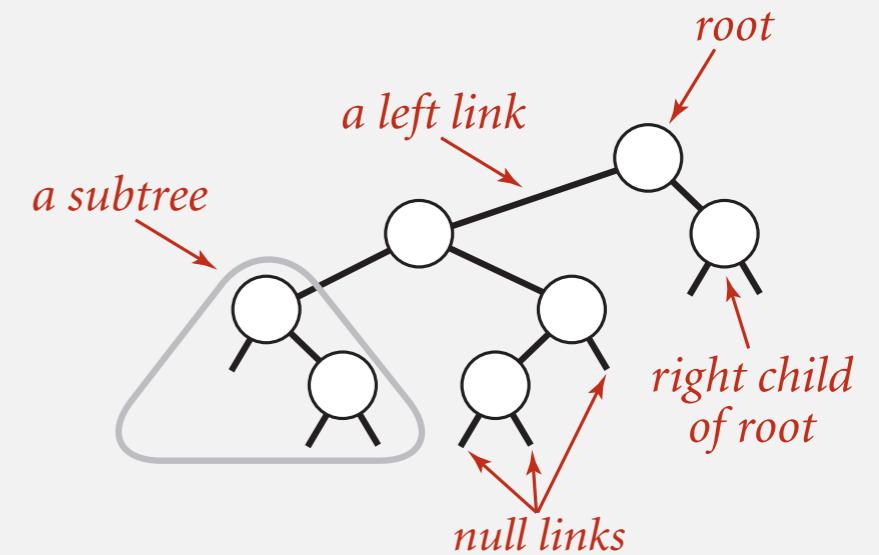


Binary search trees

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

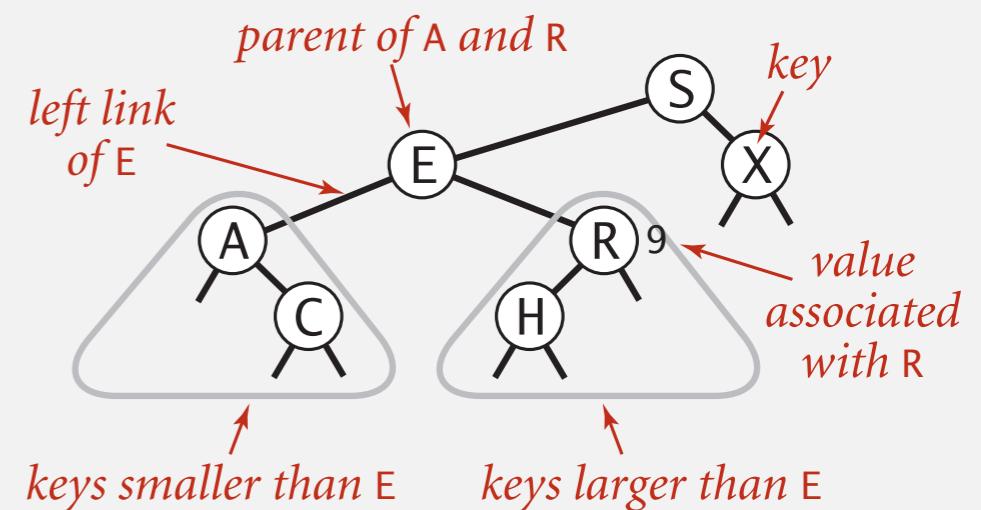
A binary tree is either:

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



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

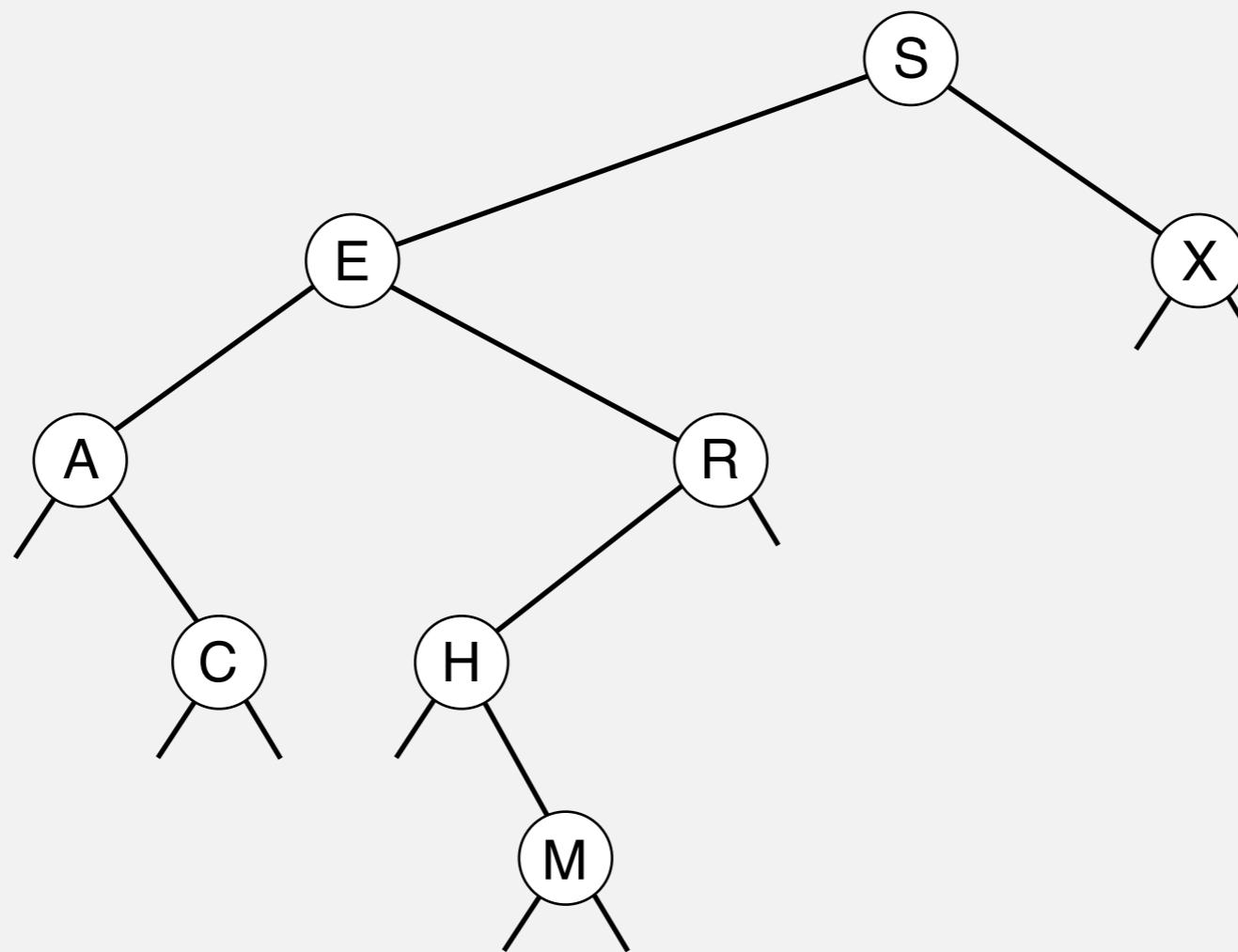
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Binary search tree demo

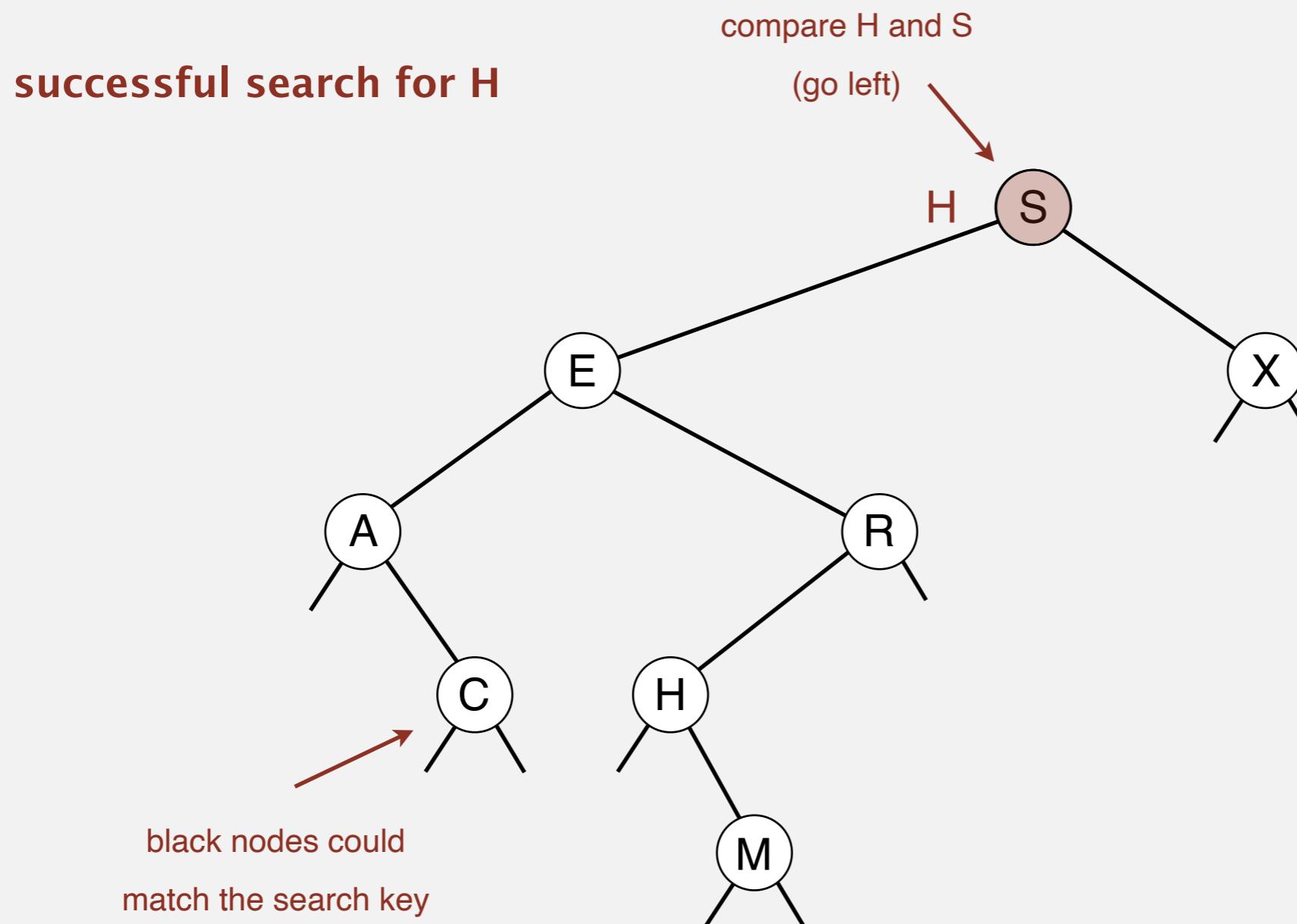
Search. If less, go left; if greater, go right; if equal, search hit.

successful search for H



Binary search tree demo

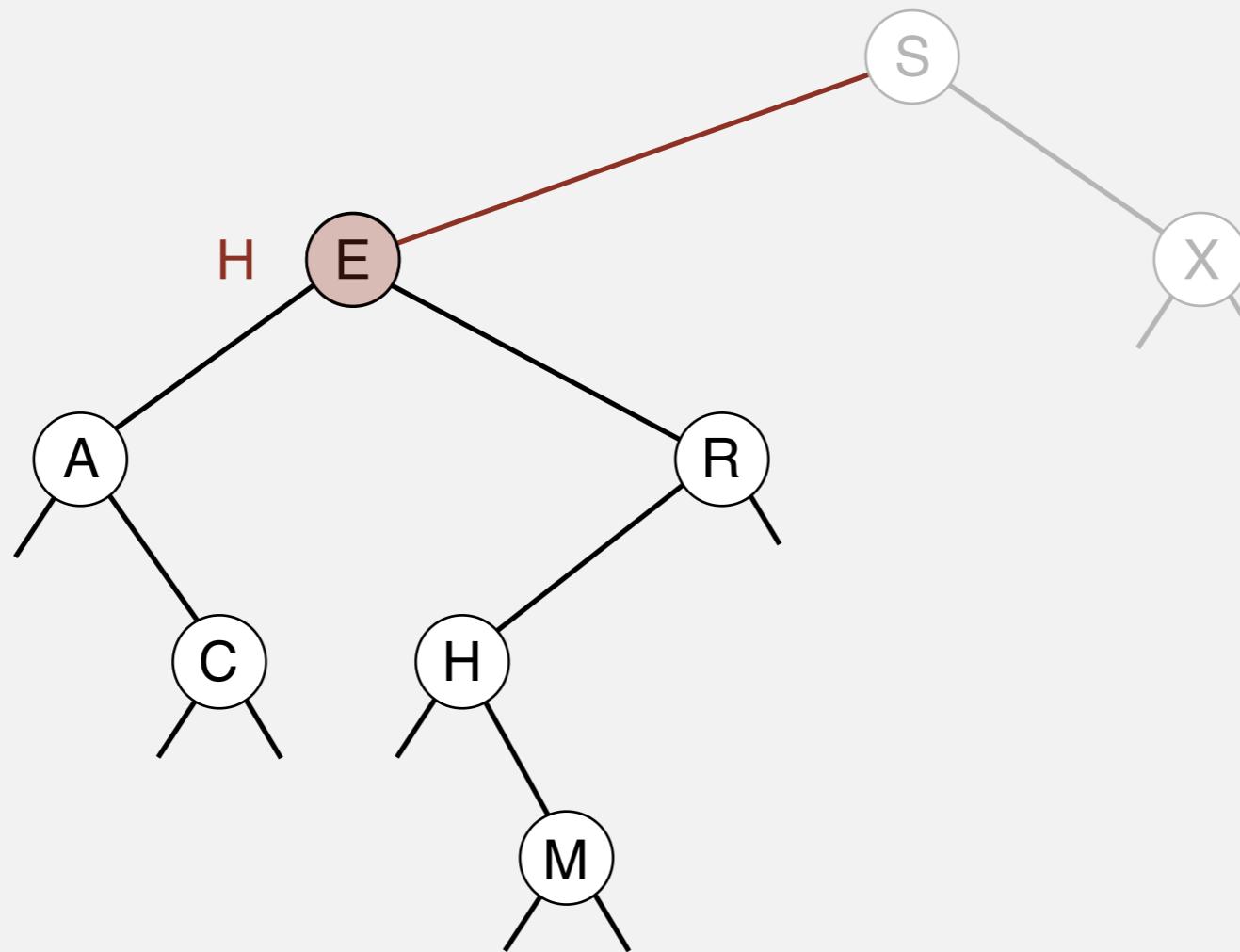
Search. If less, go left; if greater, go right; if equal, search hit.



Binary search tree demo

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

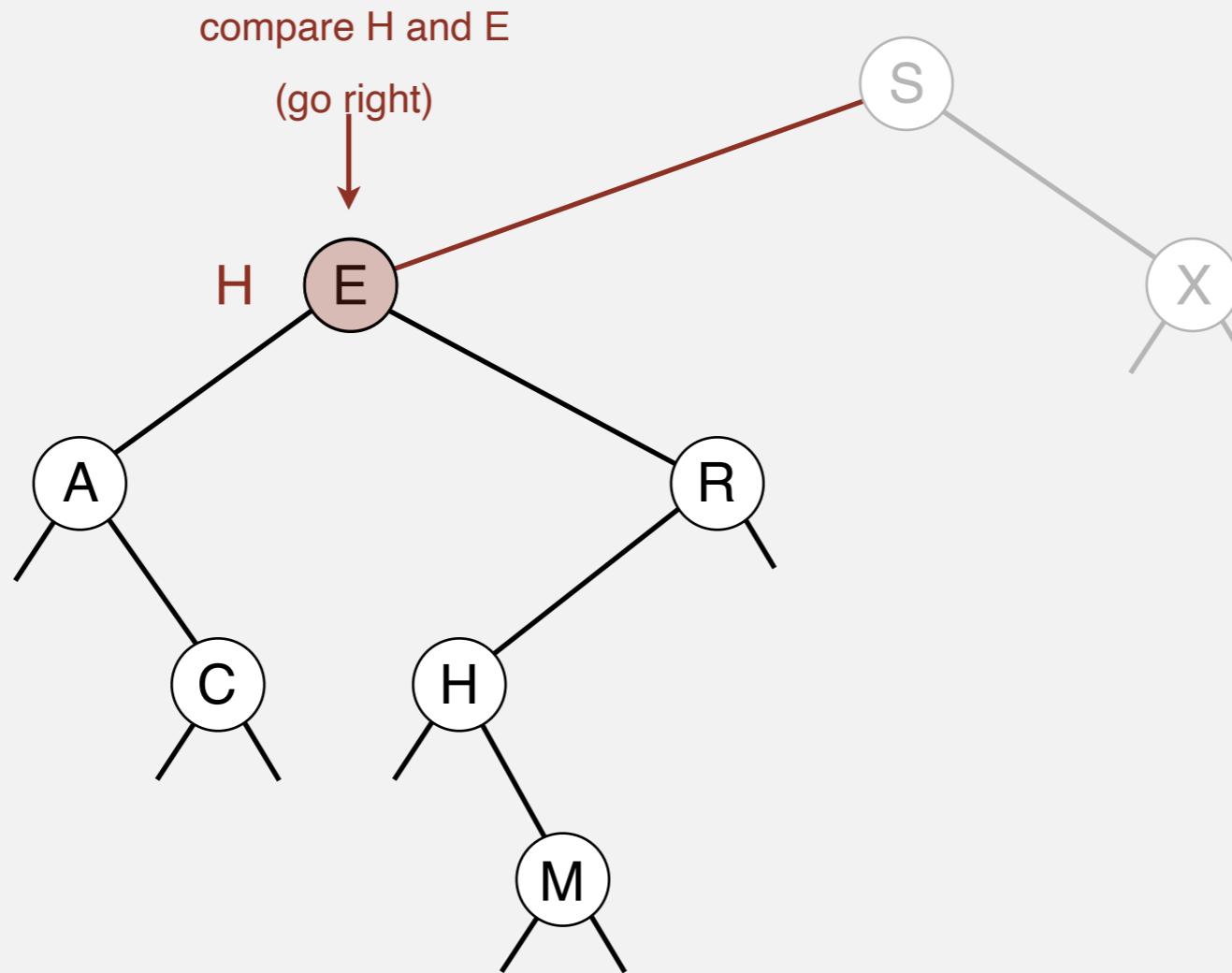
successful search for H



Binary search tree demo

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

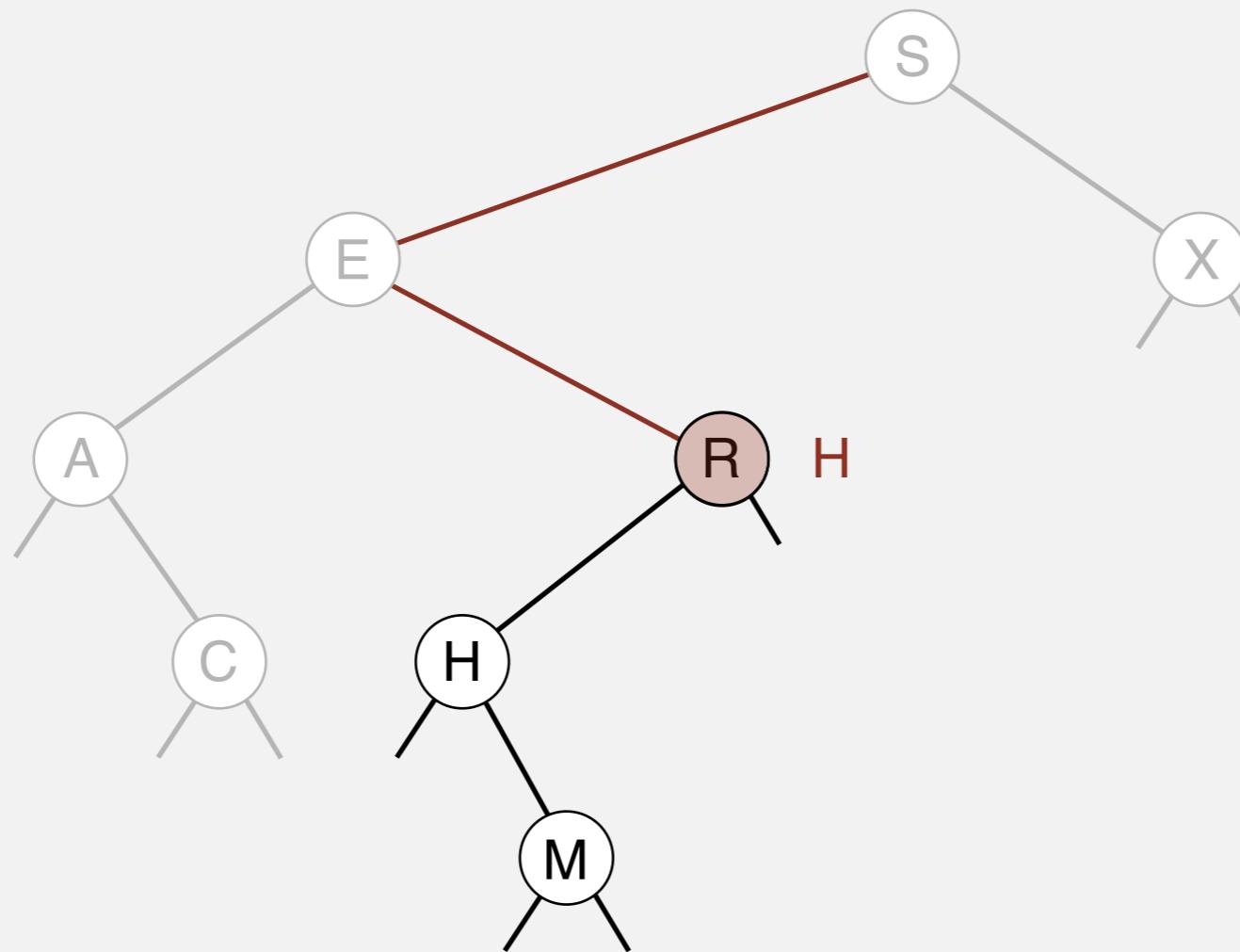
successful search for H



Binary search tree demo

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

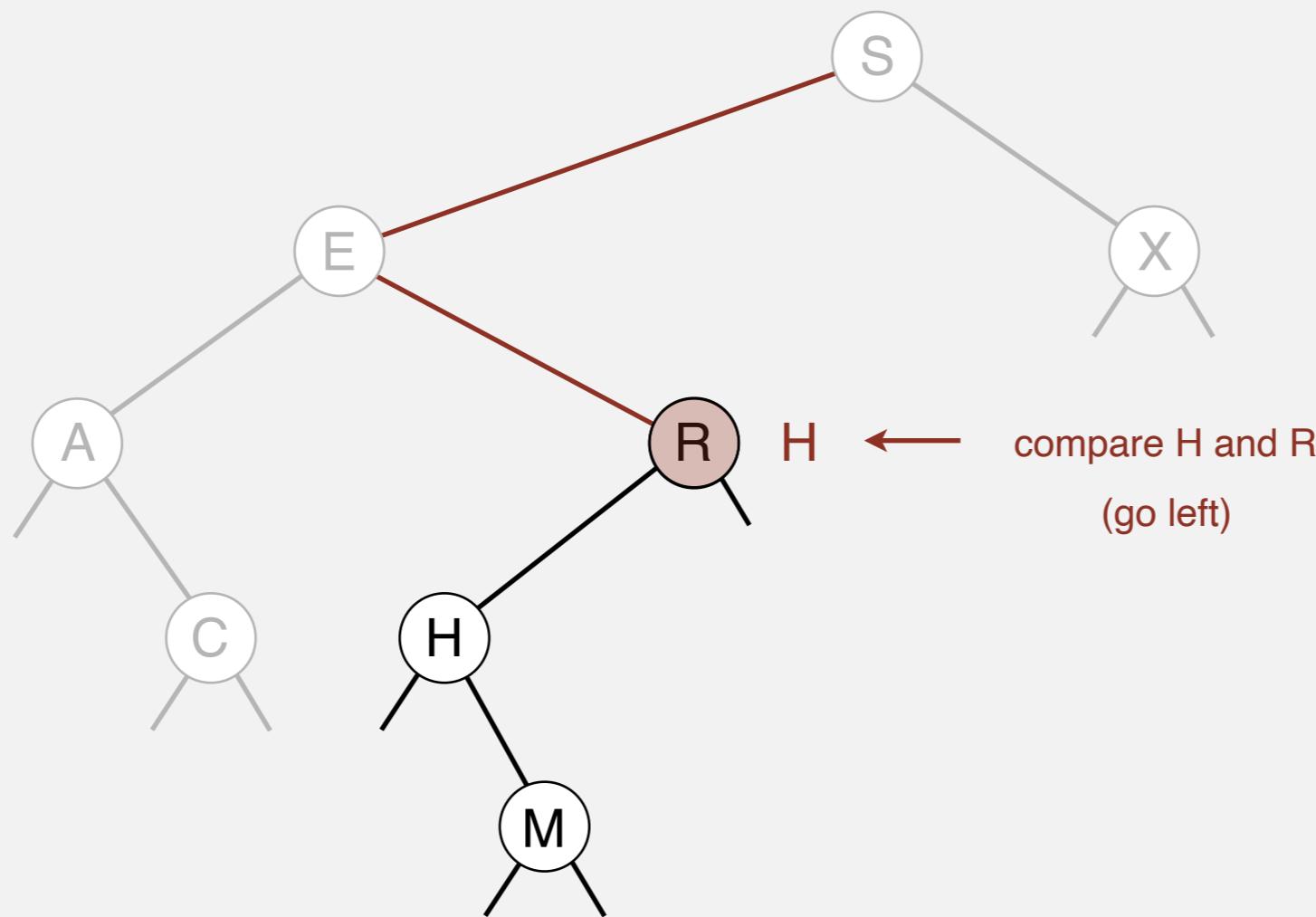
successful search for H



Binary search tree demo

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

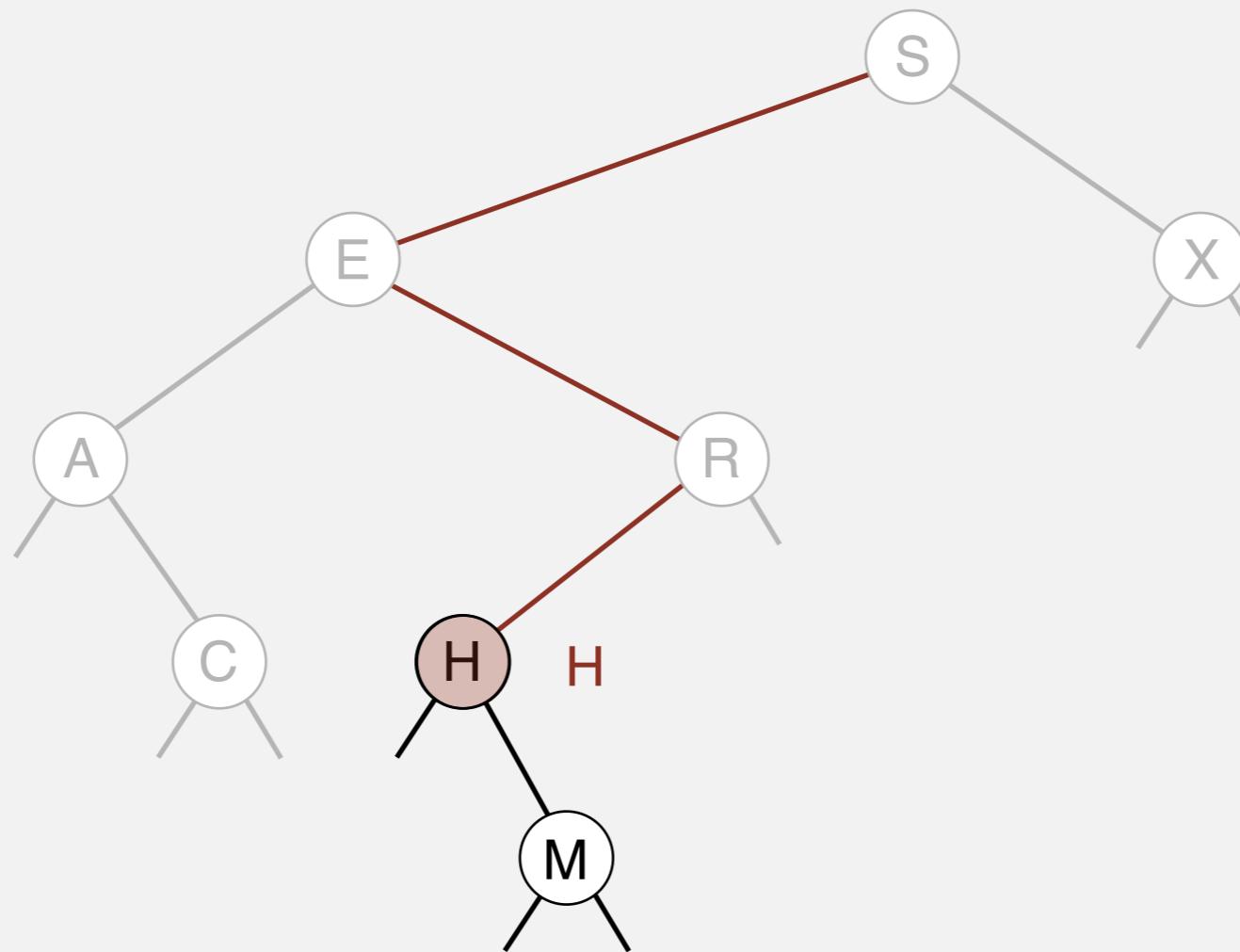
successful search for H



Binary search tree demo

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

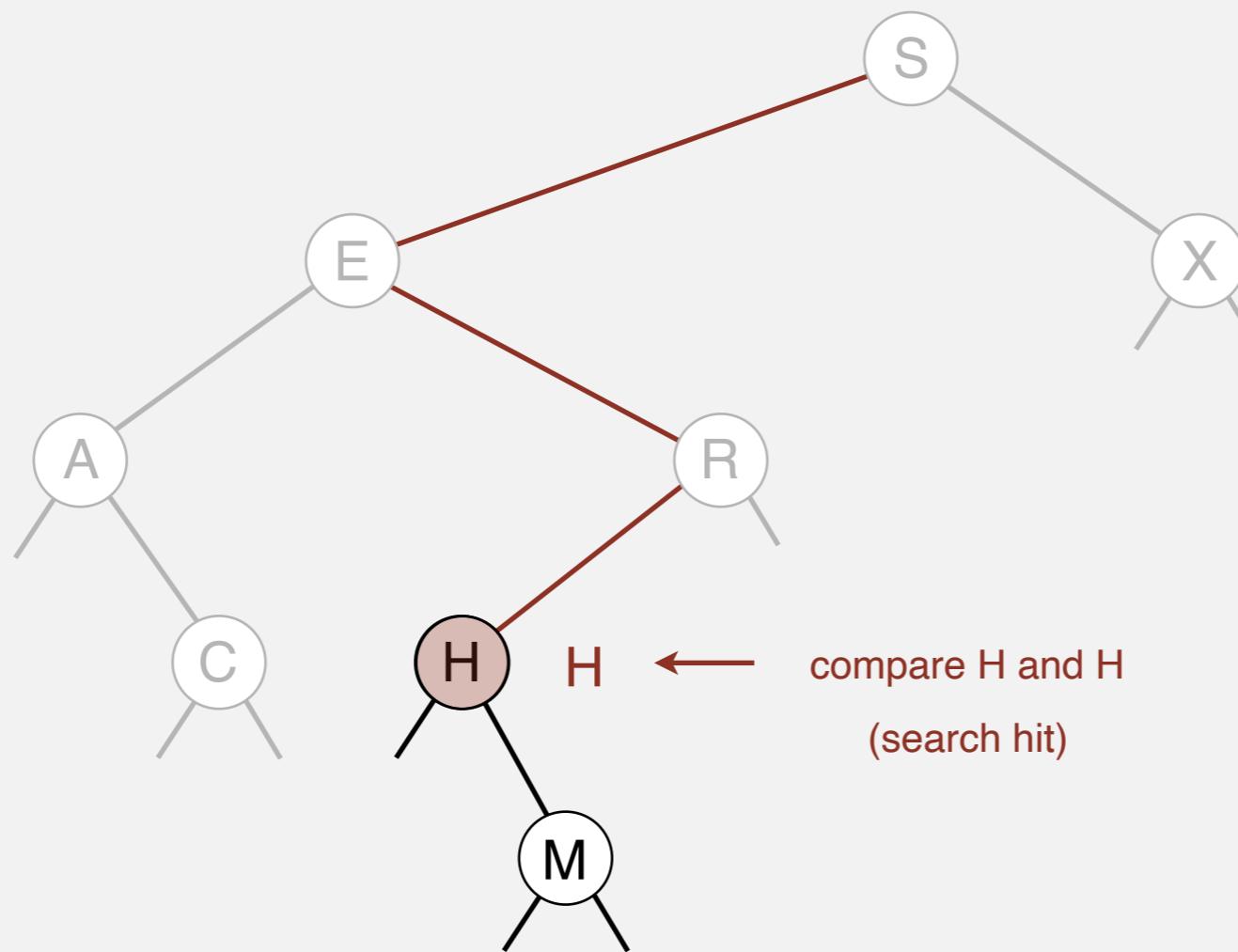
successful search for H



Binary search tree demo

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

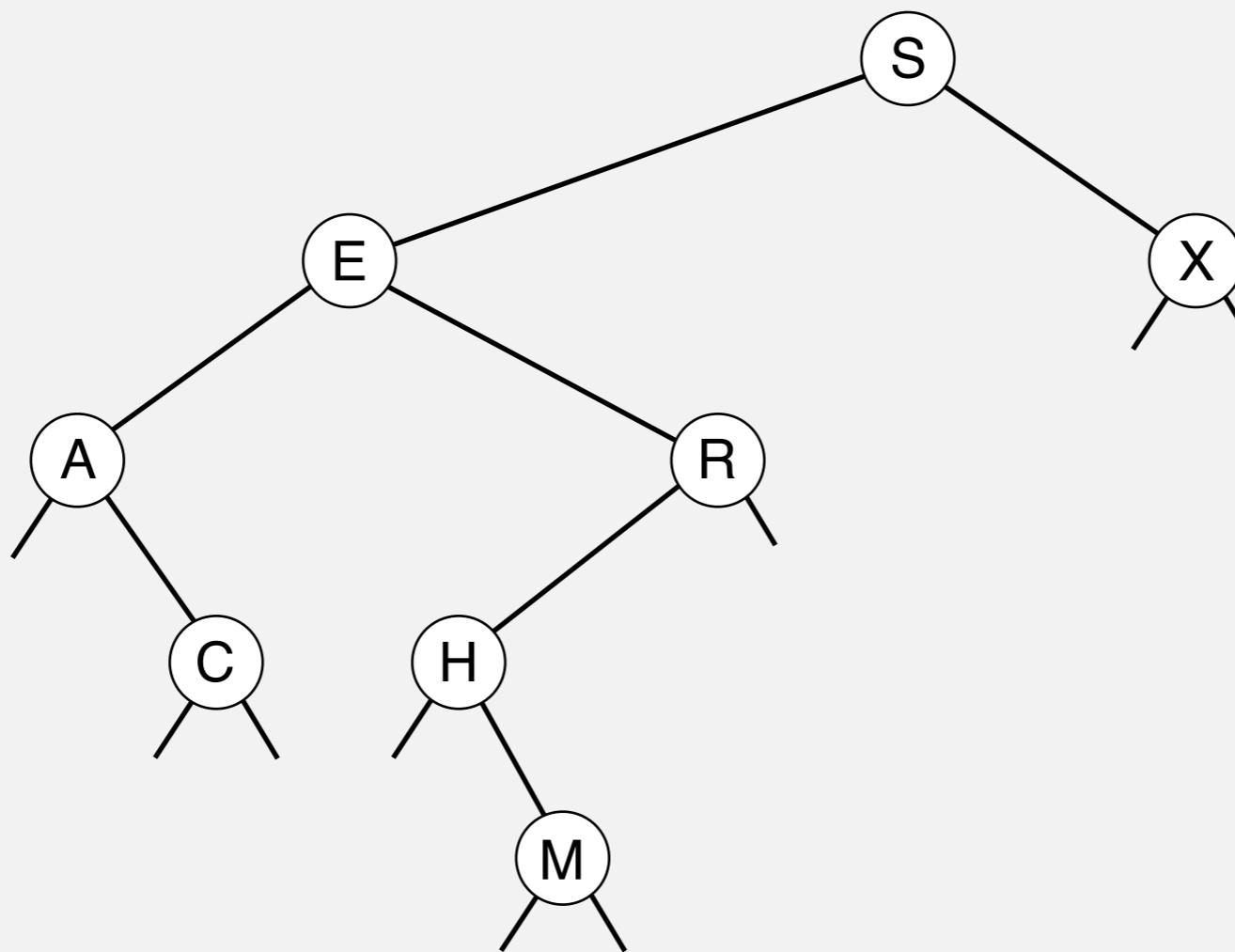
successful search for H



Binary search tree demo

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

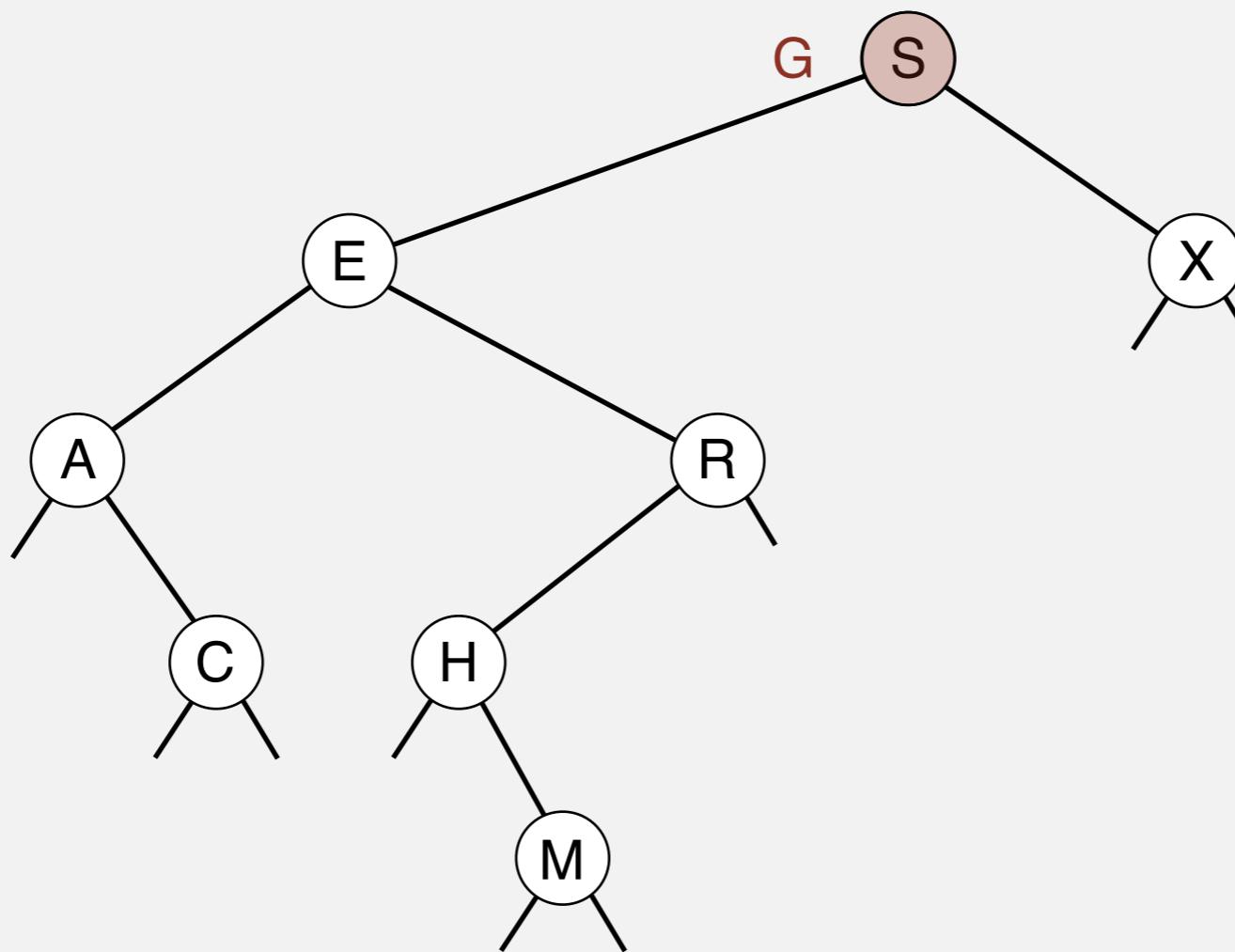
unsuccessful search for G



Binary search tree demo

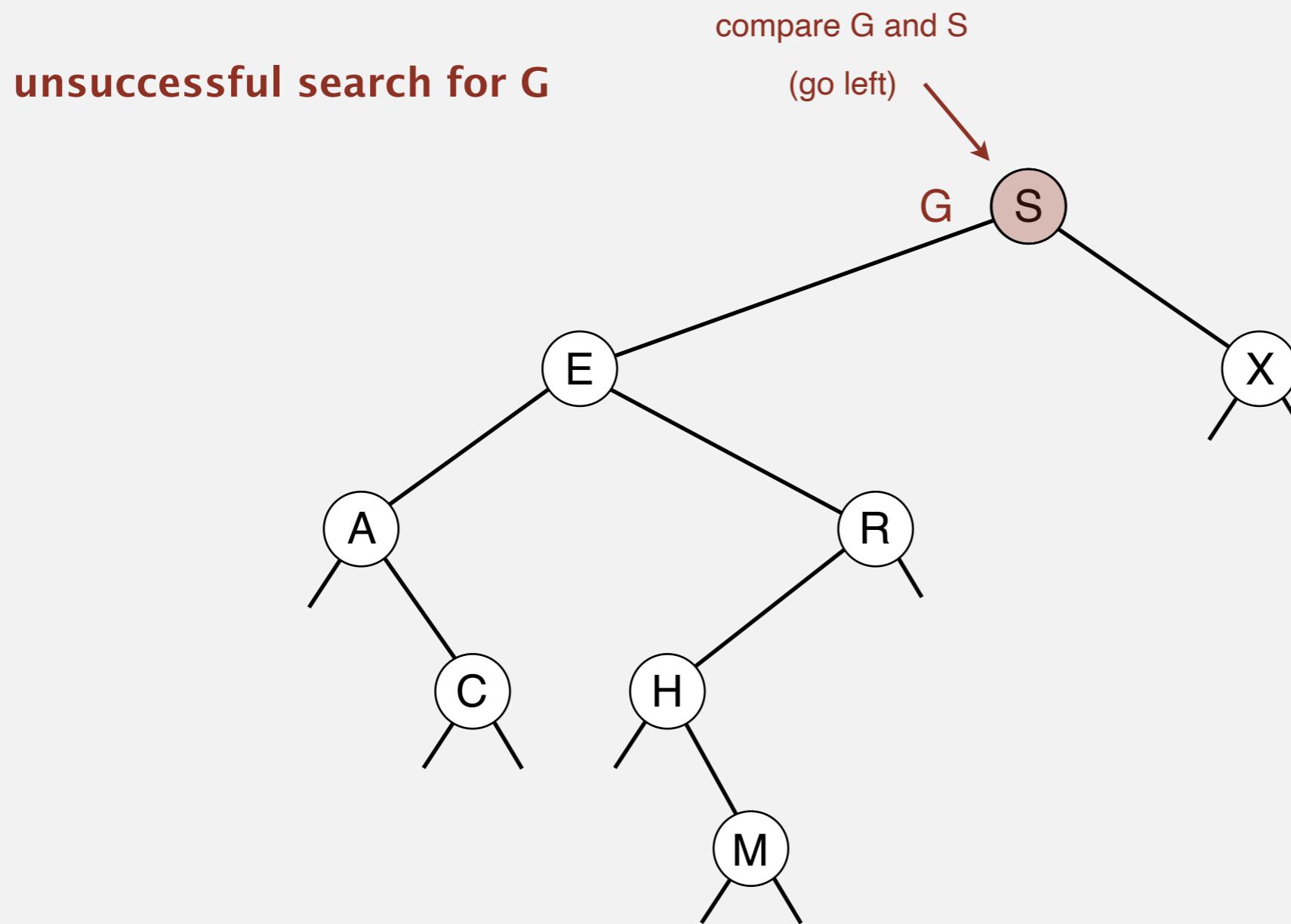
Search. If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G



Binary search tree demo

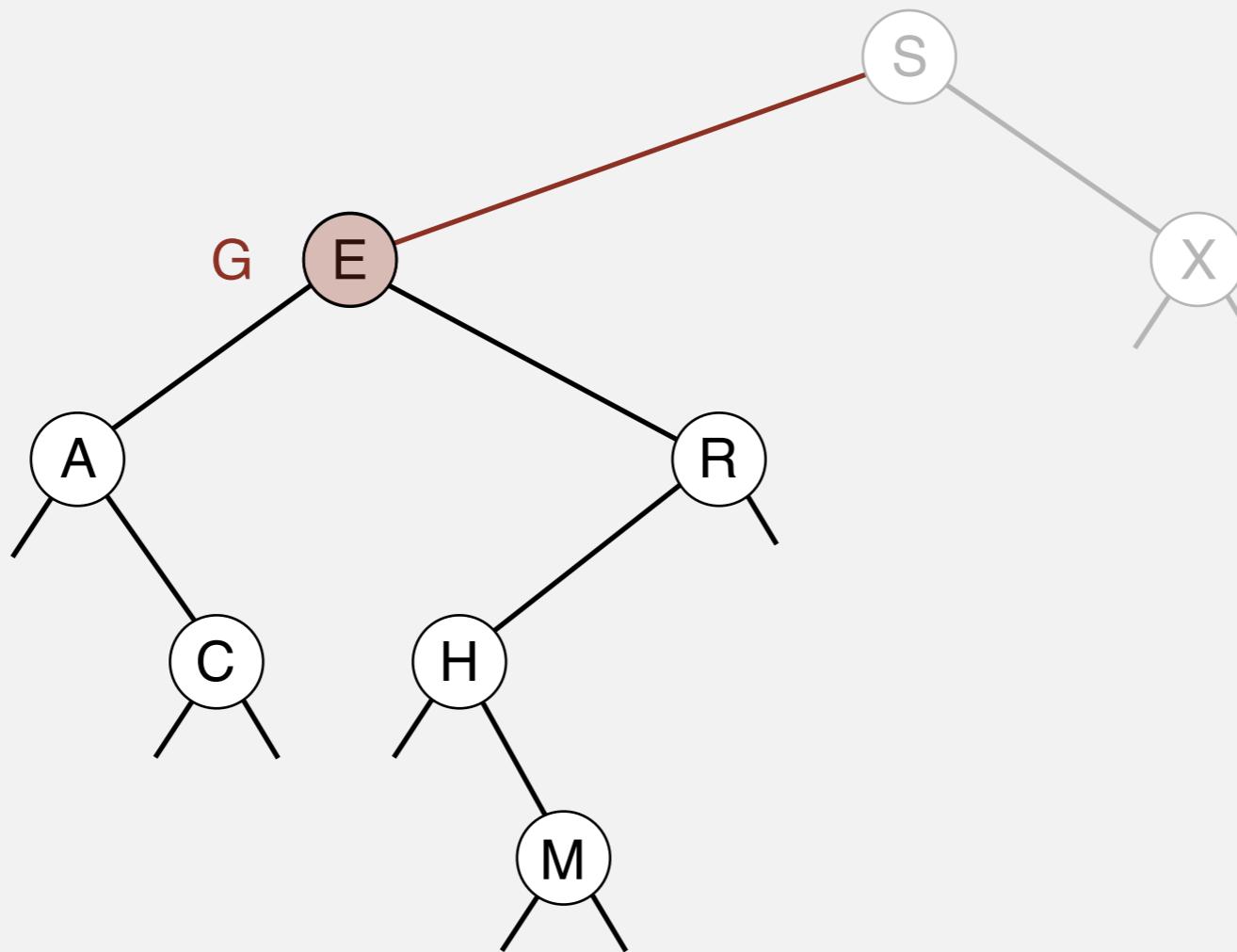
Search. If less, go left; if greater, go right; if equal, search hit.



Binary search tree demo

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

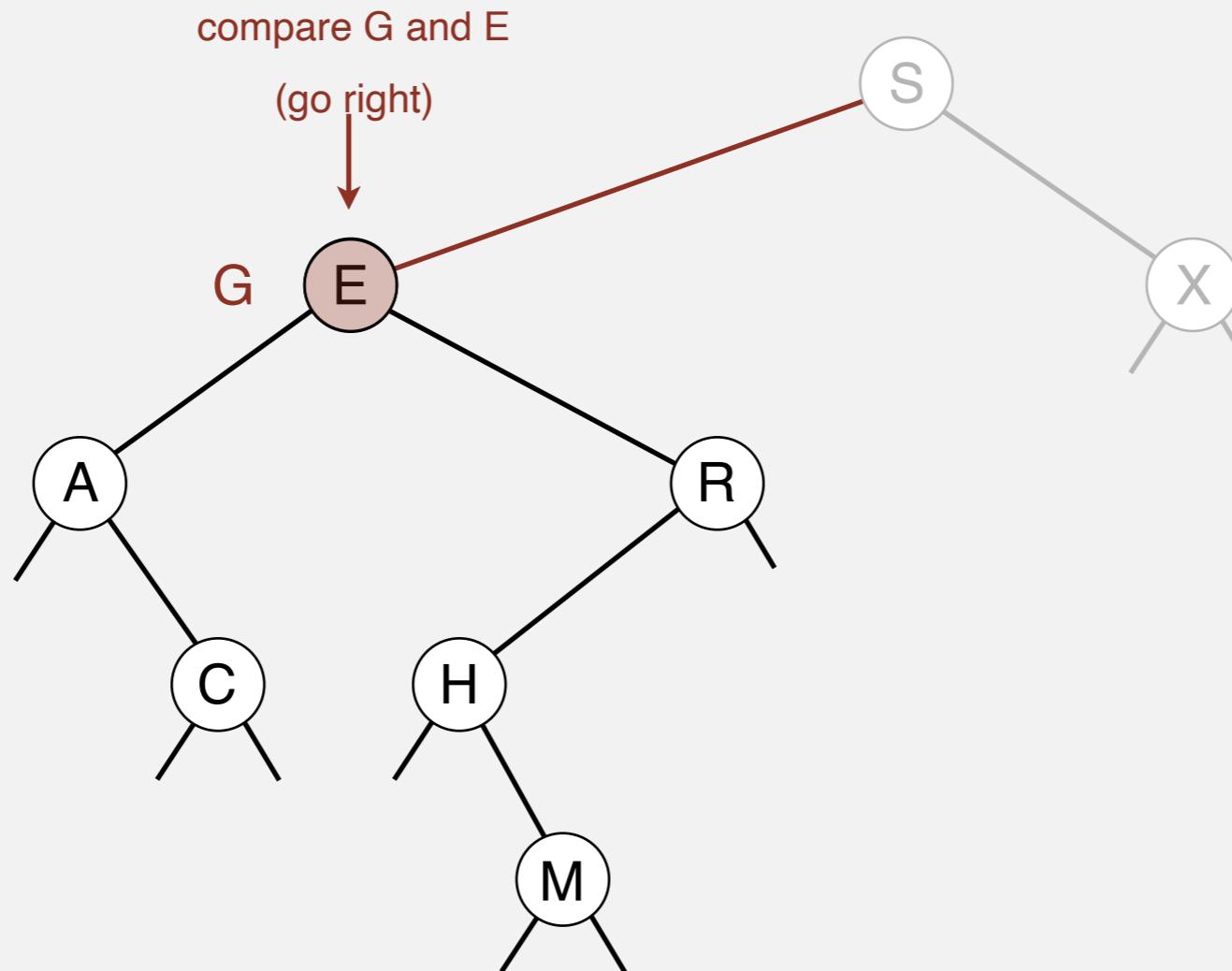
unsuccessful search for G



Binary search tree demo

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

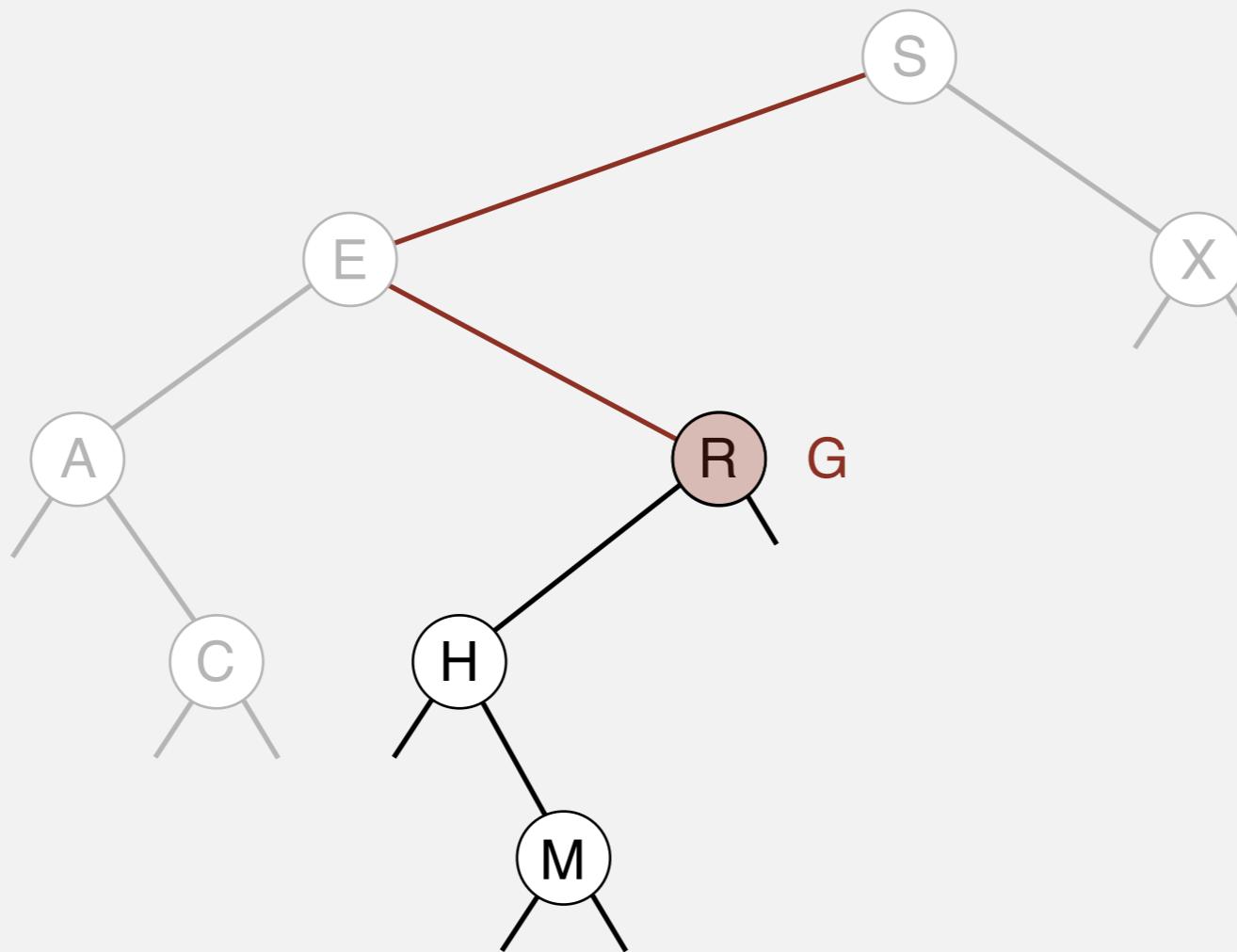
unsuccessful search for G



Binary search tree demo

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

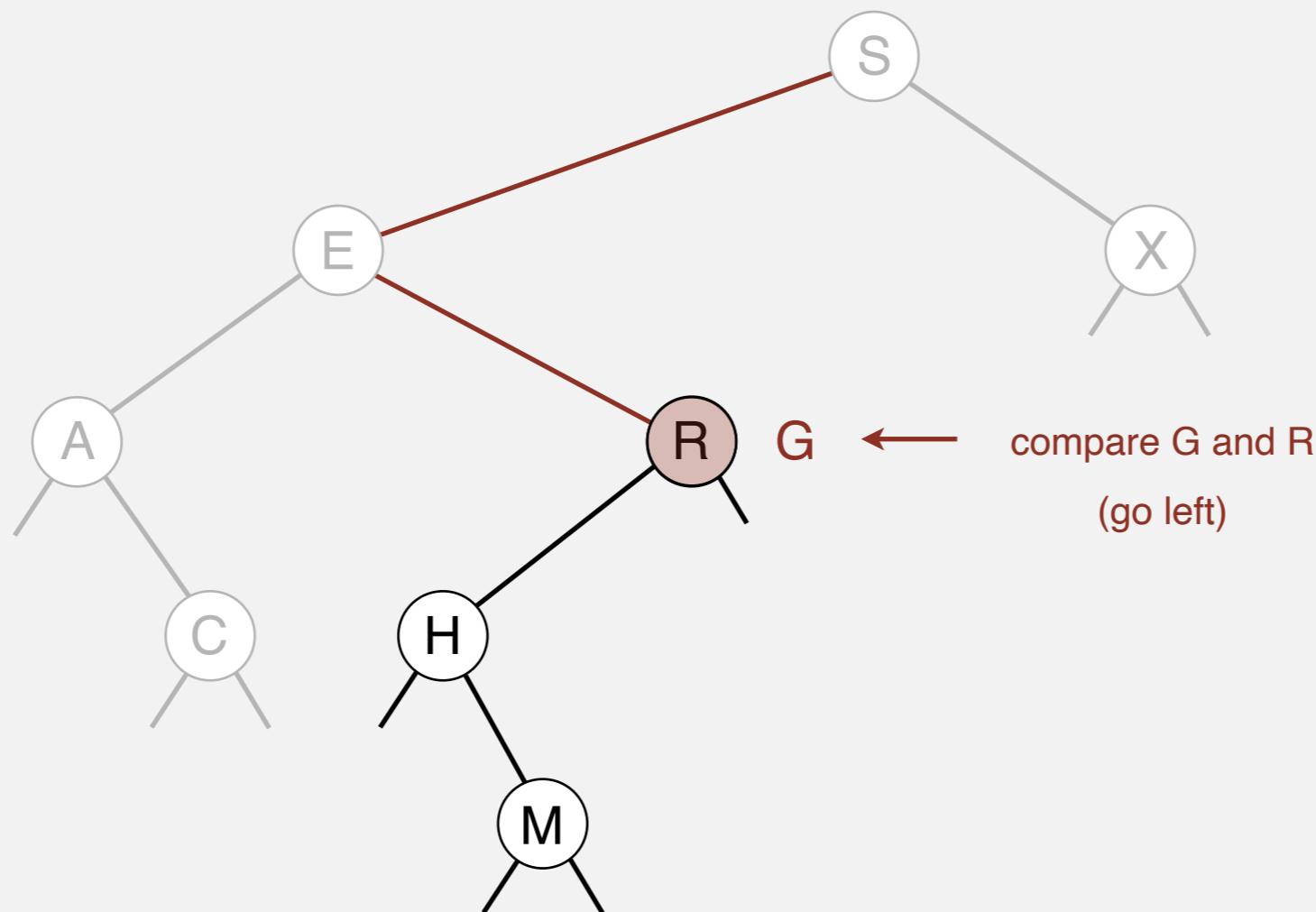
unsuccessful search for G



Binary search tree demo

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

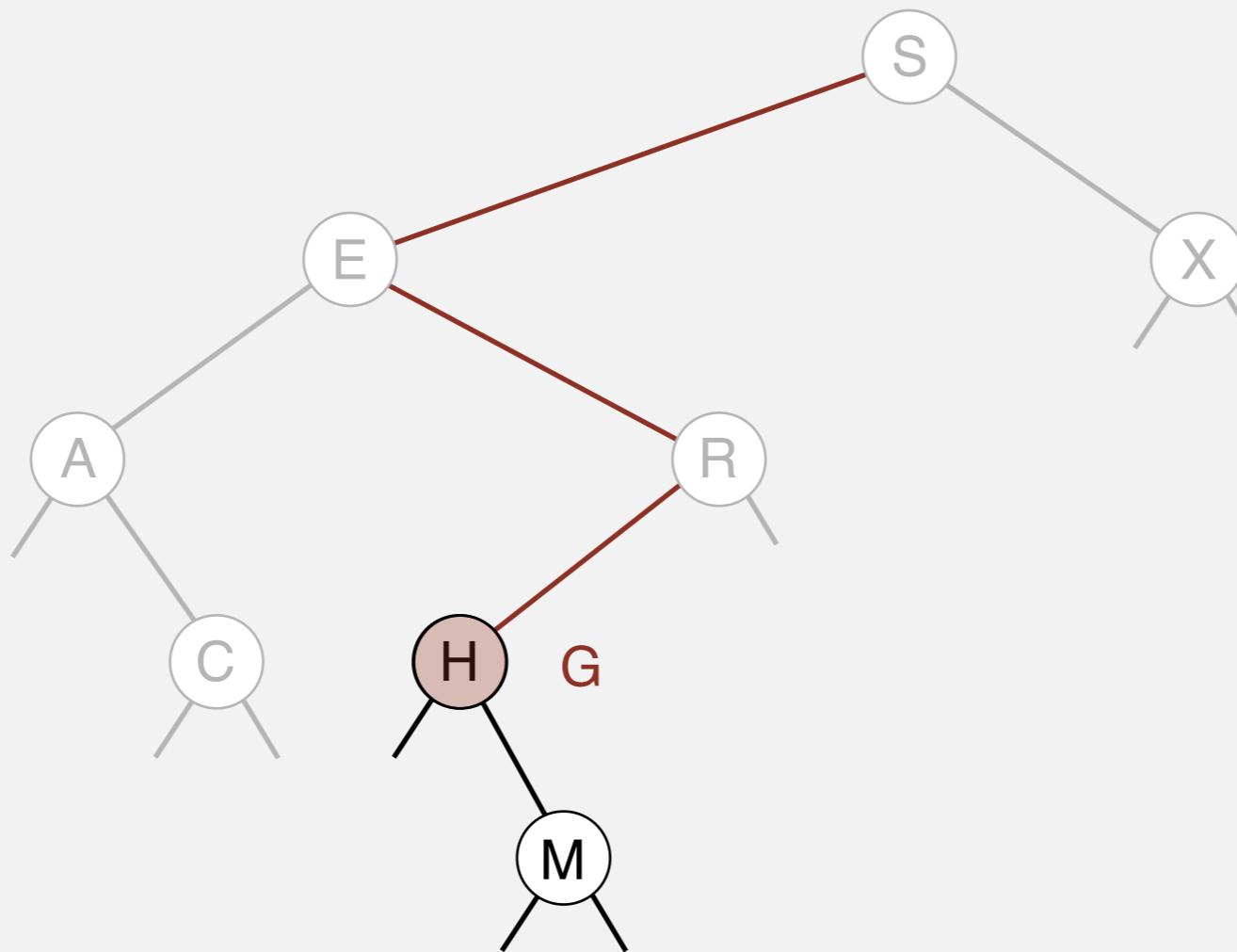
unsuccessful search for G



Binary search tree demo

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

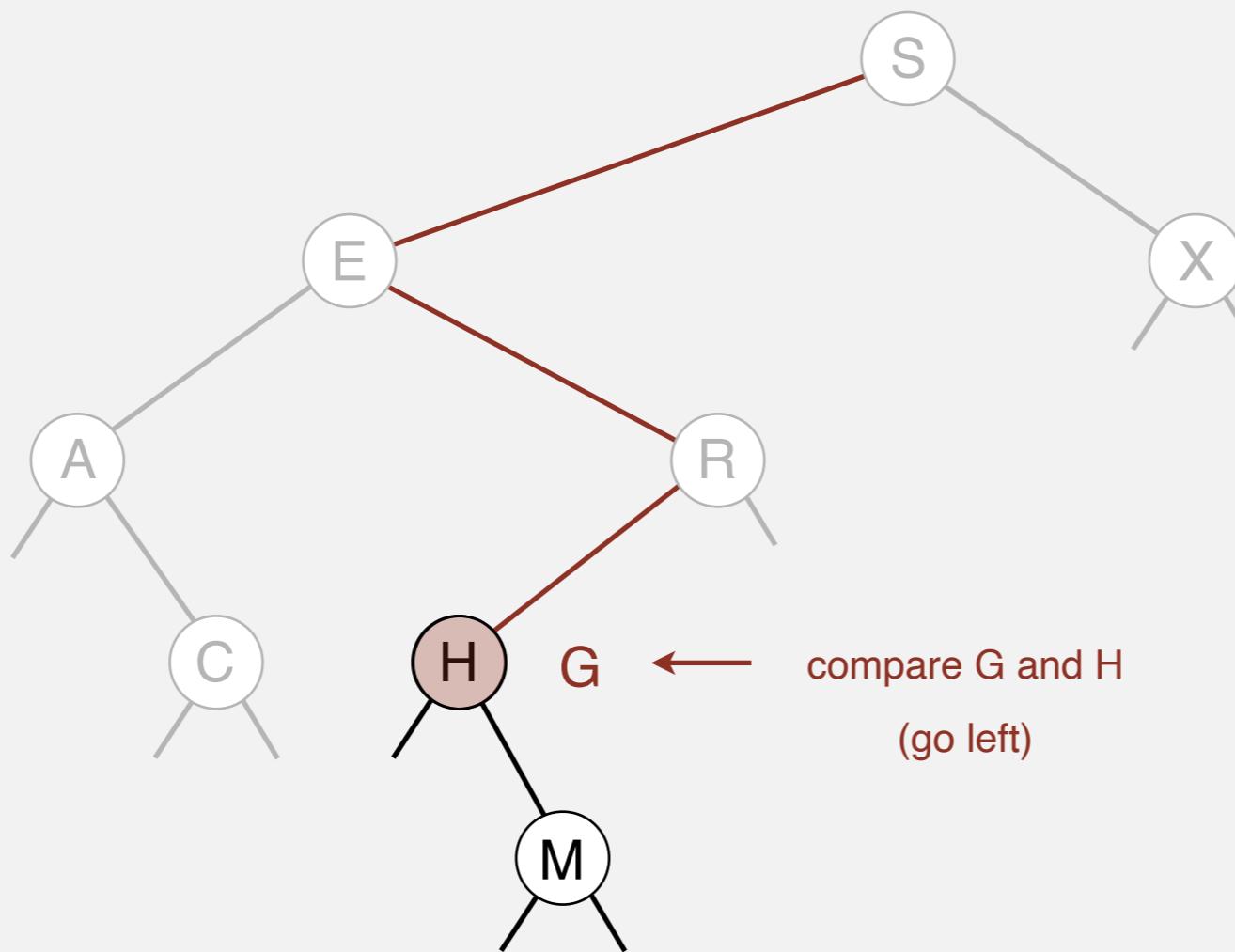
unsuccessful search for G



Binary search tree demo

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

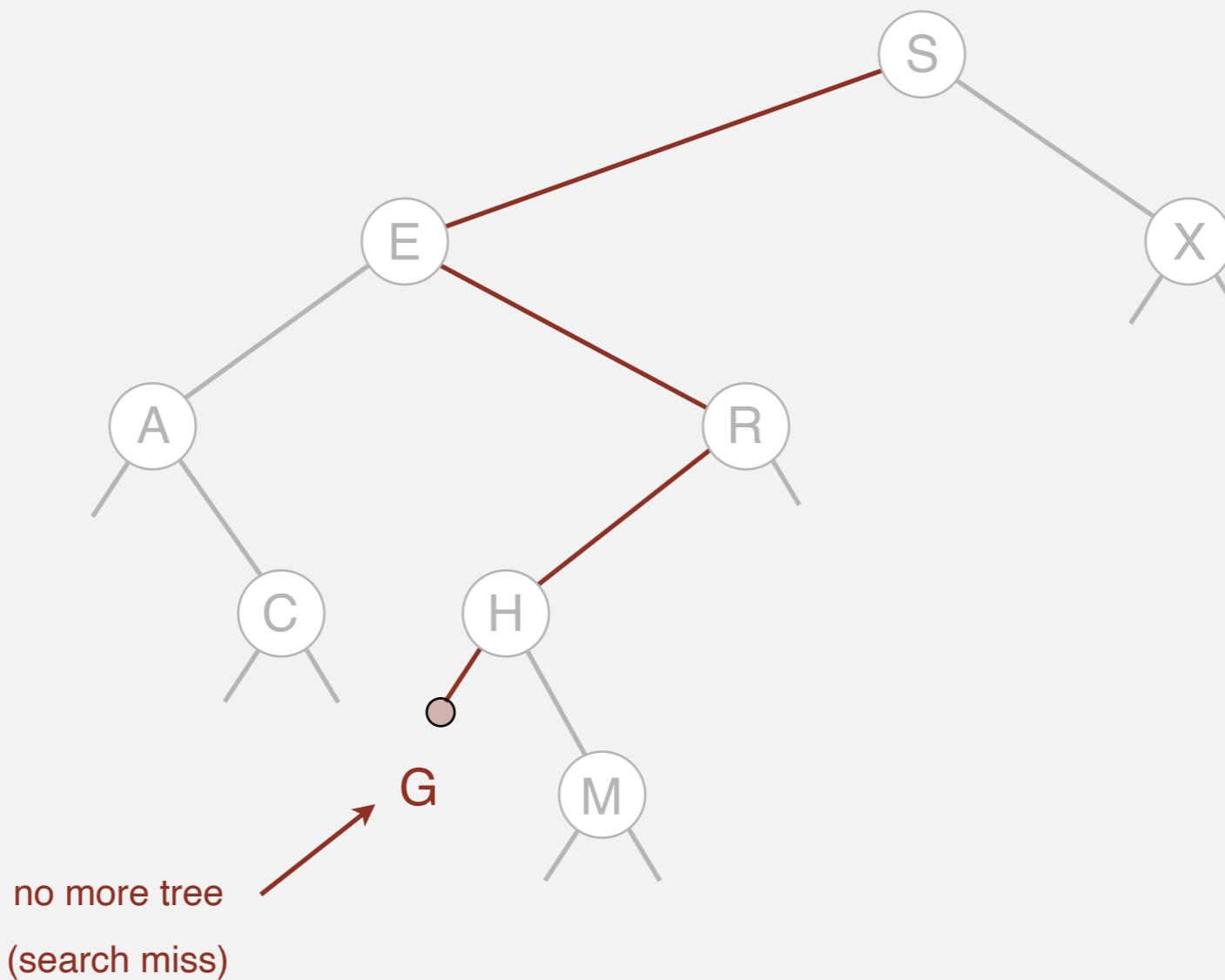
unsuccessful search for G



Binary search tree demo

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

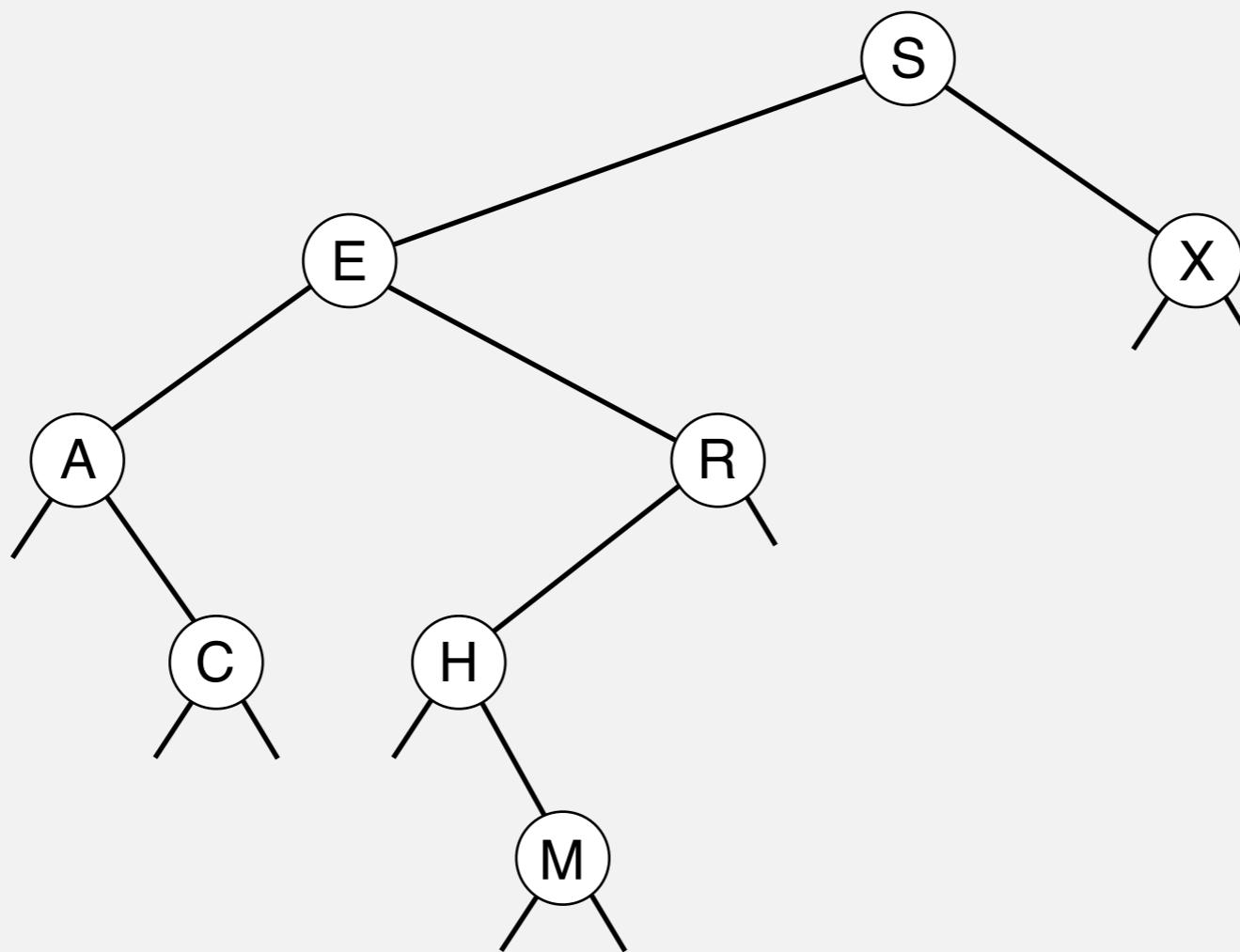
unsuccessful search for G



Binary search tree demo

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

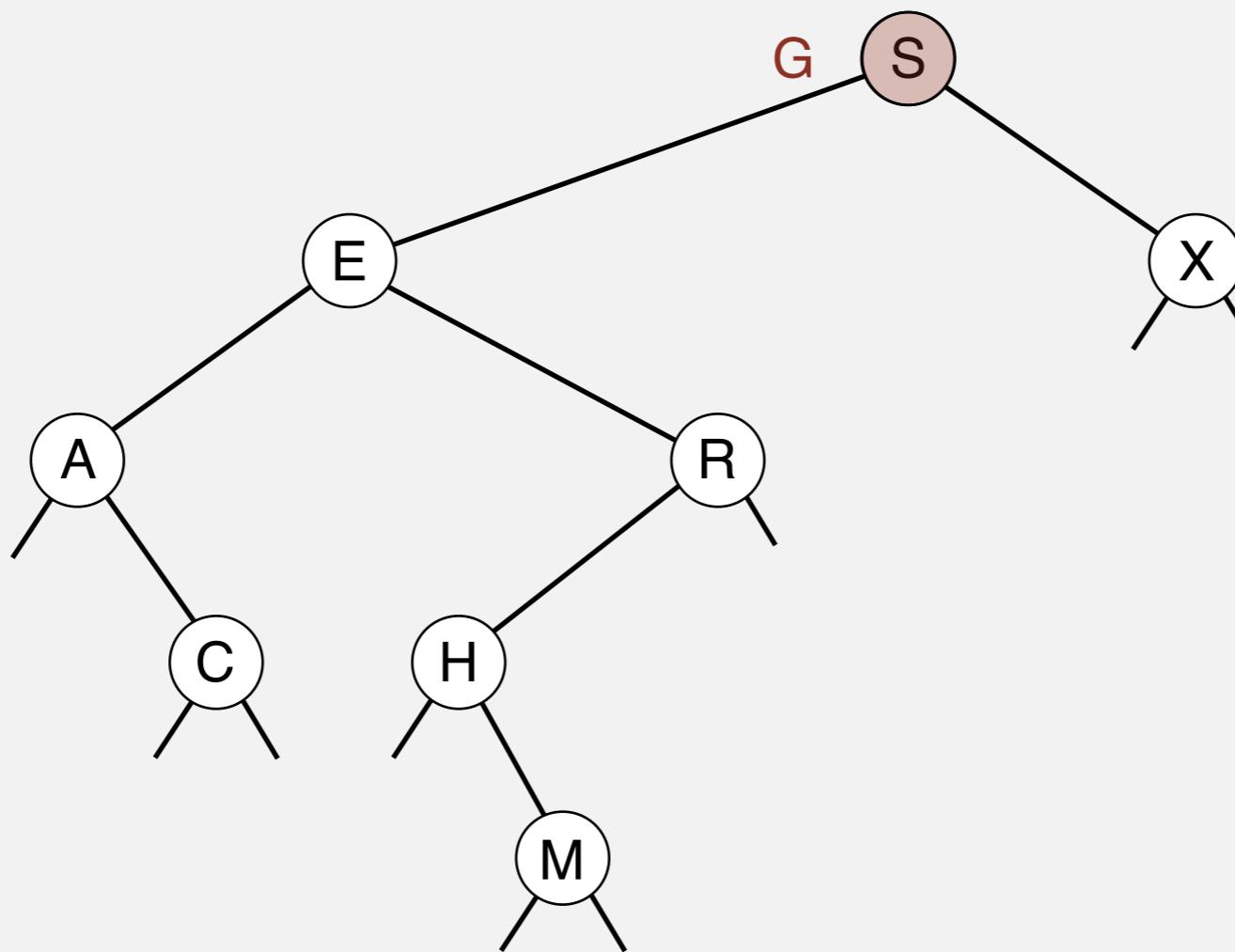
insert G



Binary search tree demo

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

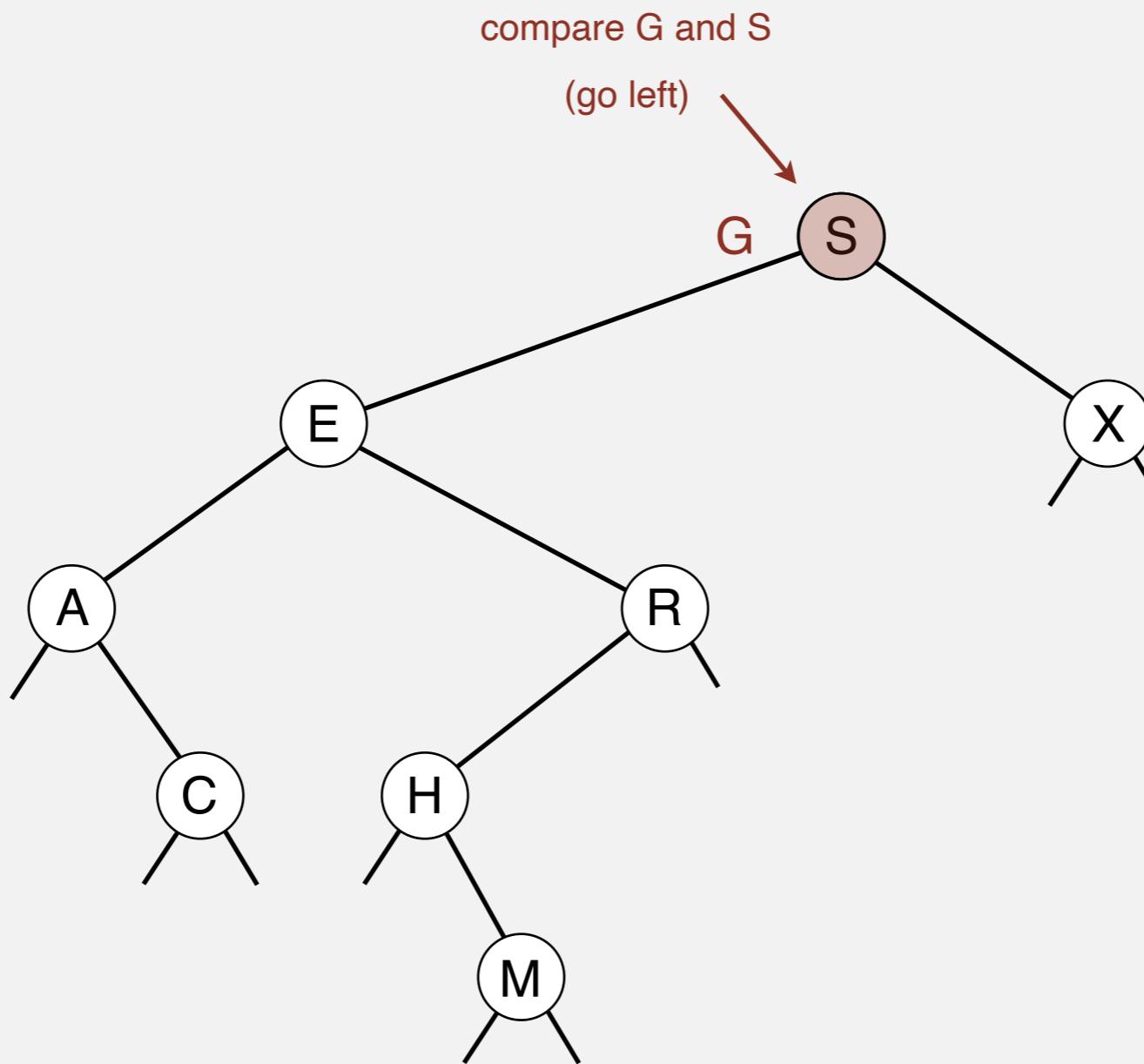
insert G



Binary search tree demo

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

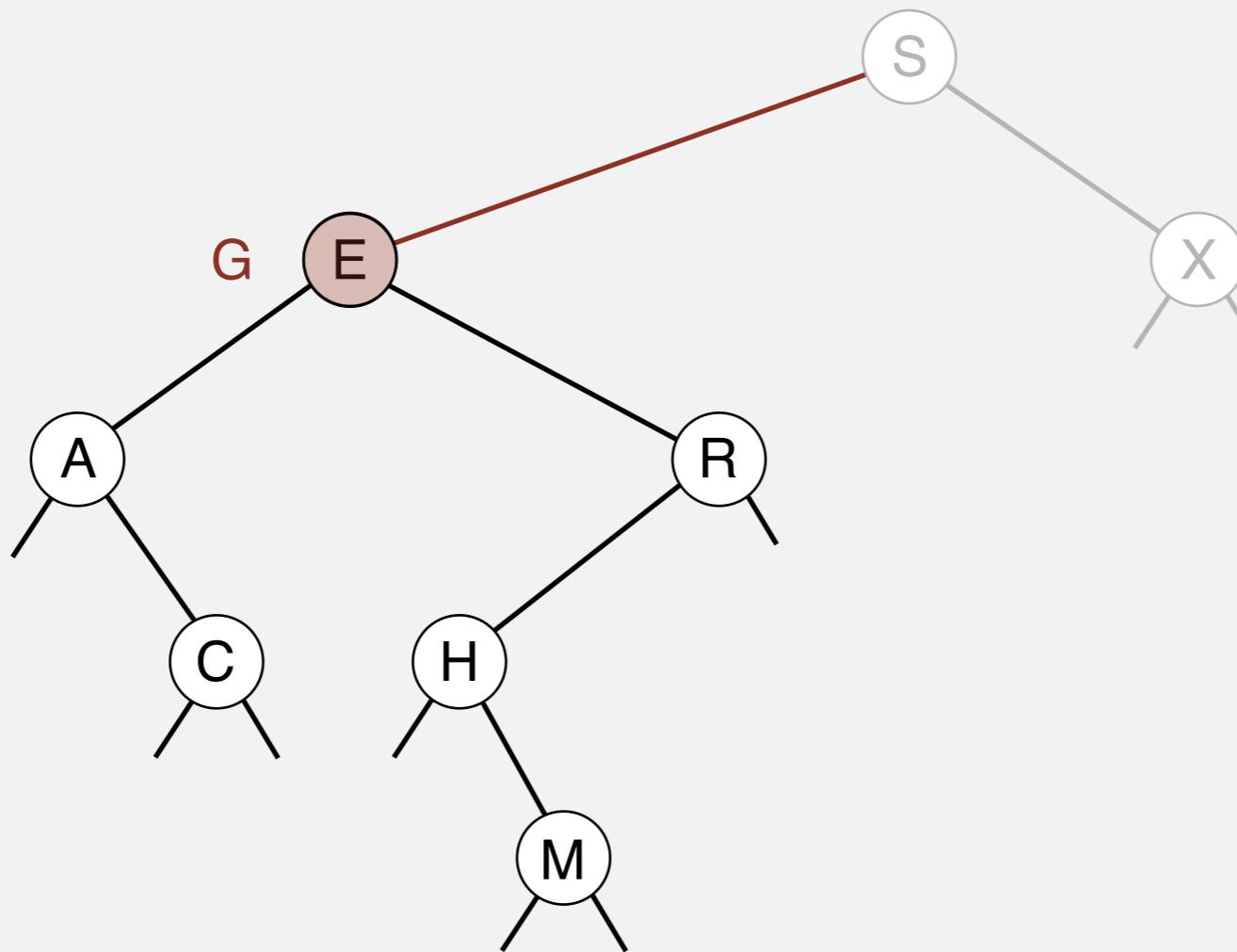
insert G



Binary search tree demo

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

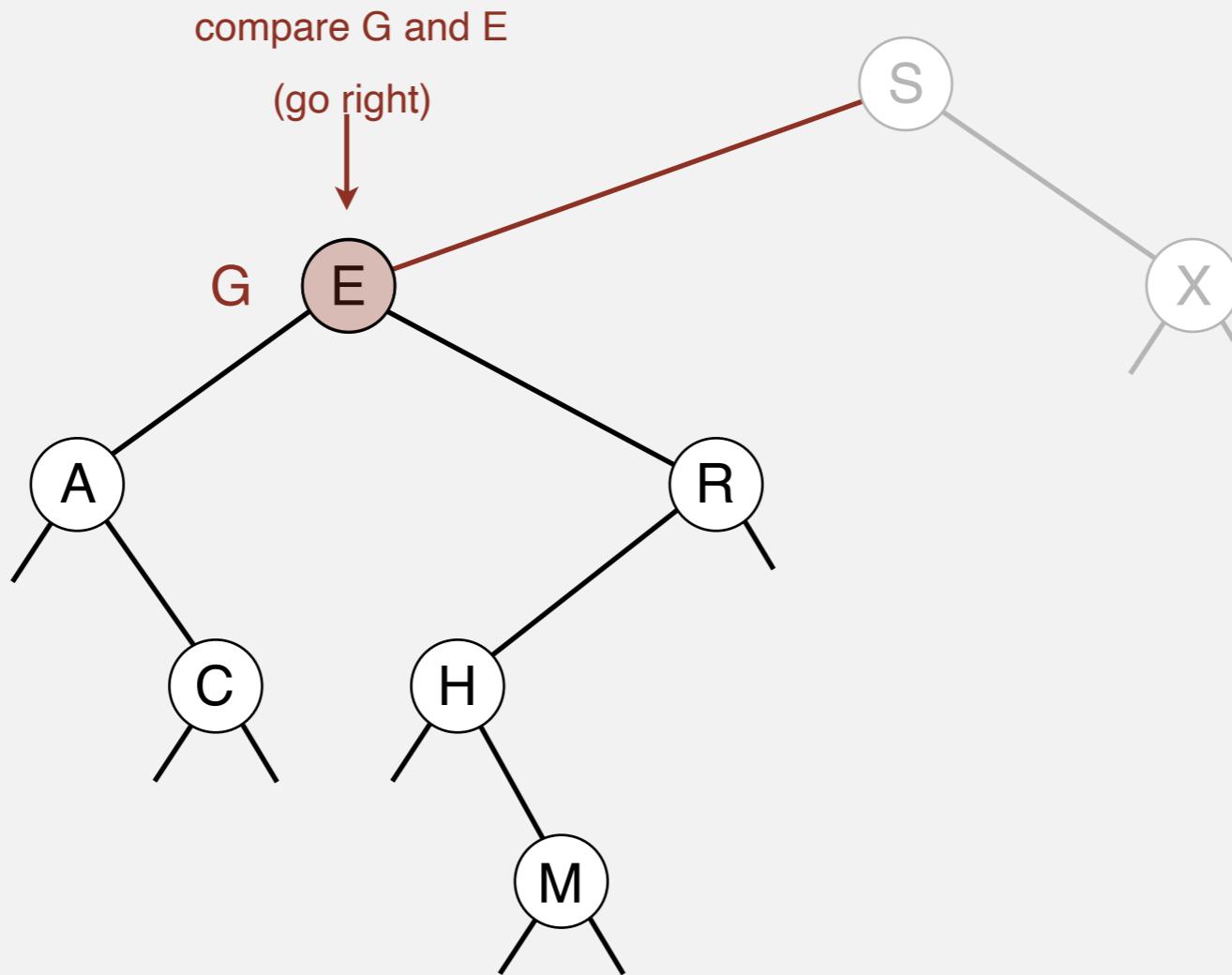
insert G



Binary search tree demo

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

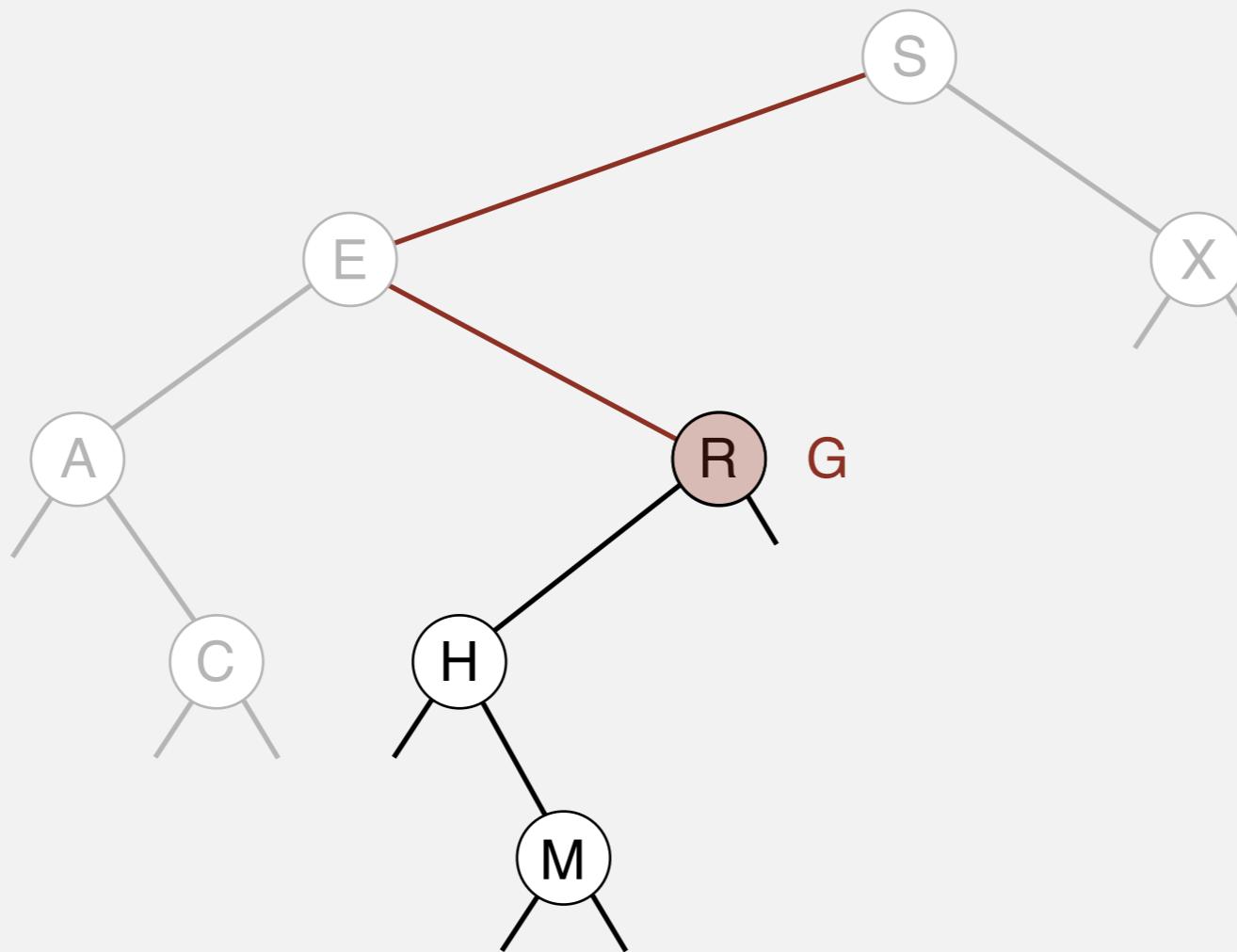
insert G



Binary search tree demo

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

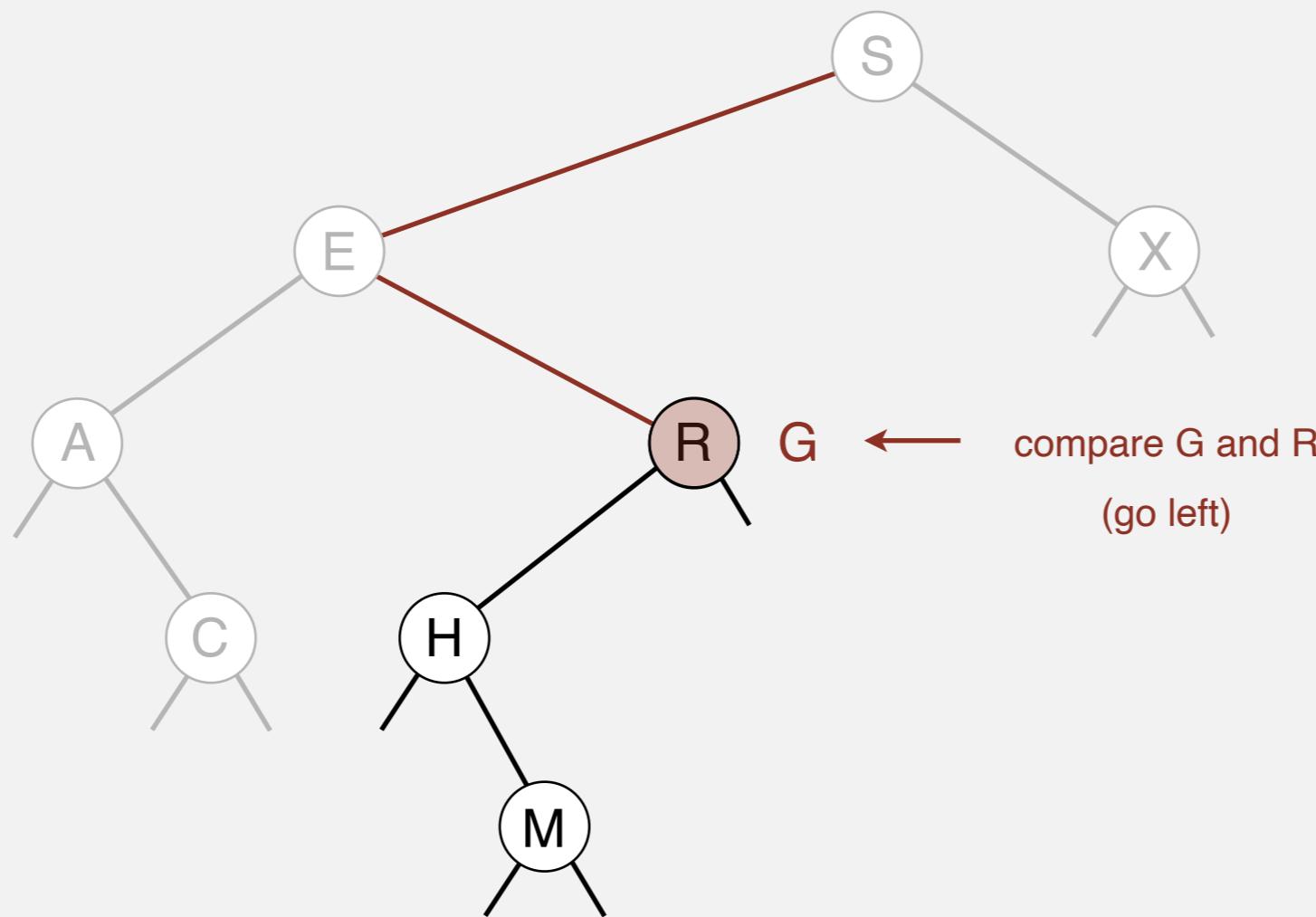
insert G



Binary search tree demo

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

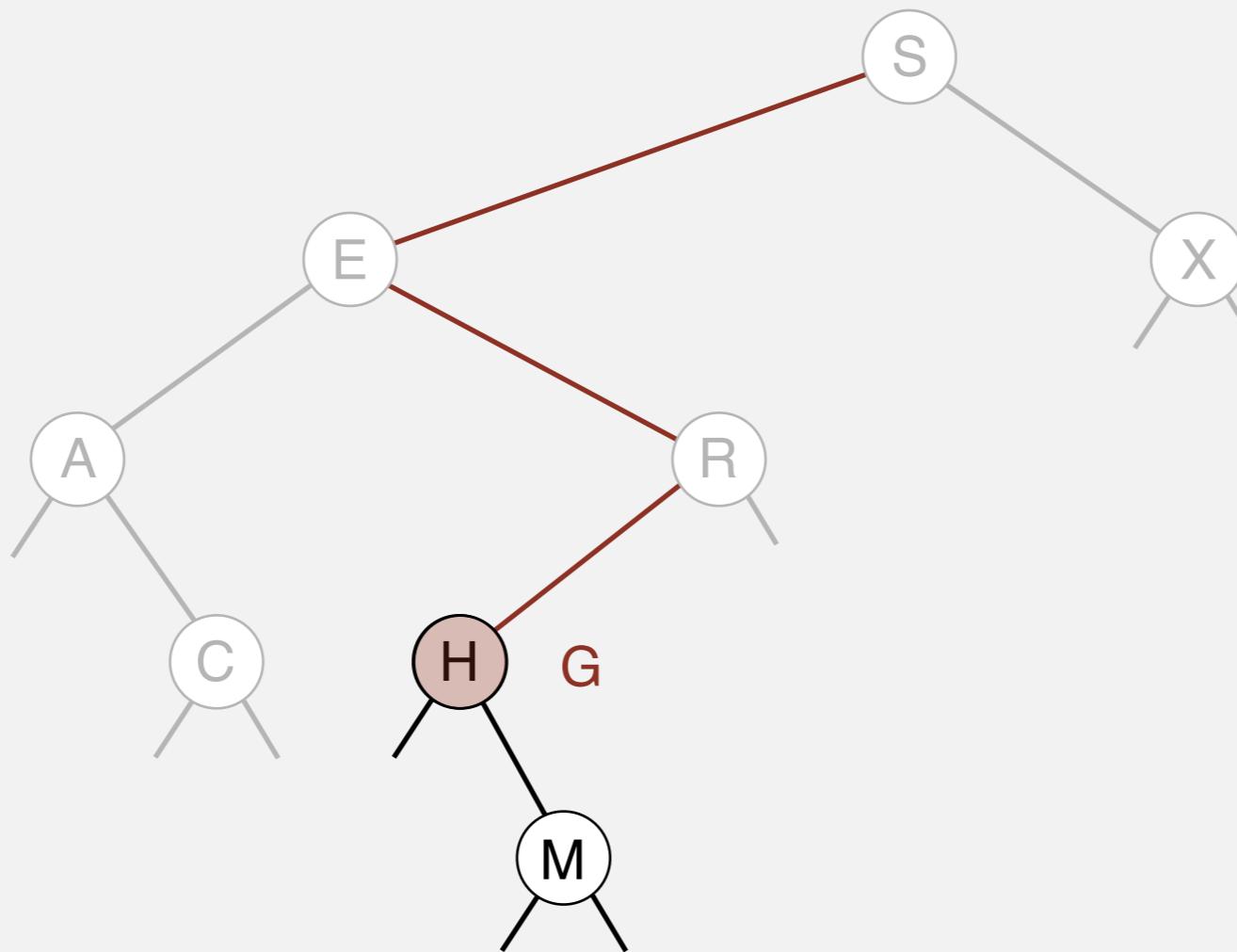
insert G



Binary search tree demo

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

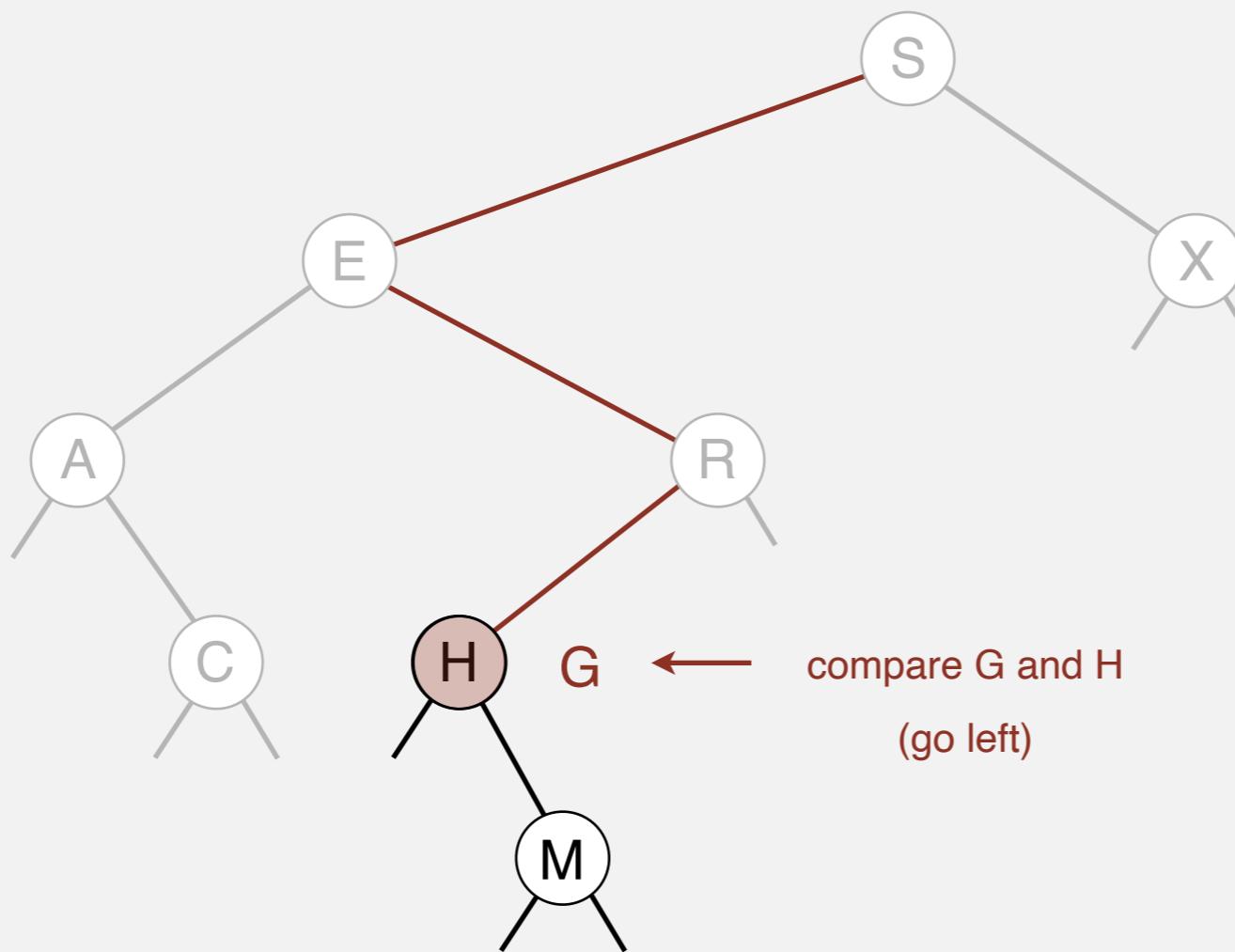
insert G



Binary search tree demo

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

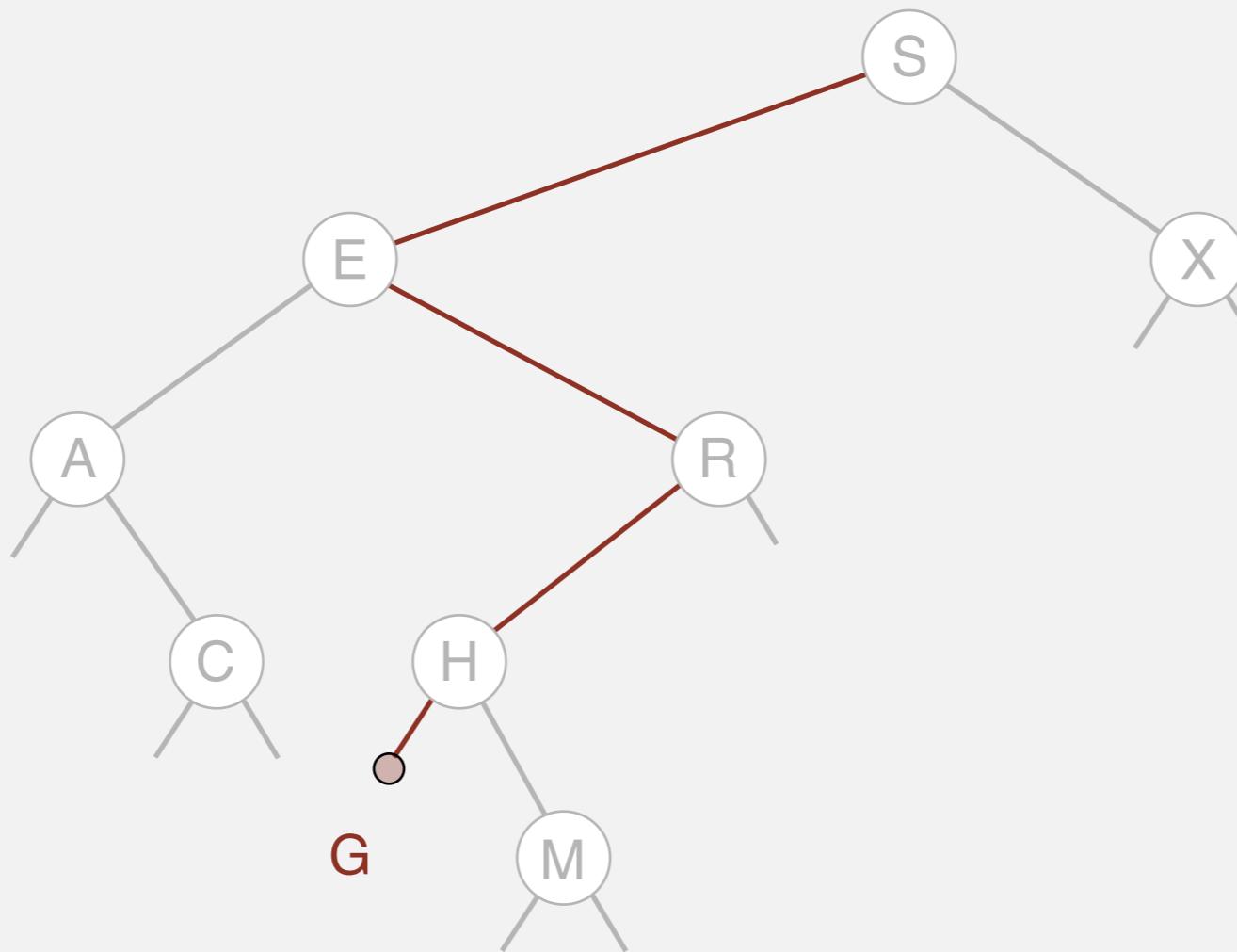
insert G



Binary search tree demo

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

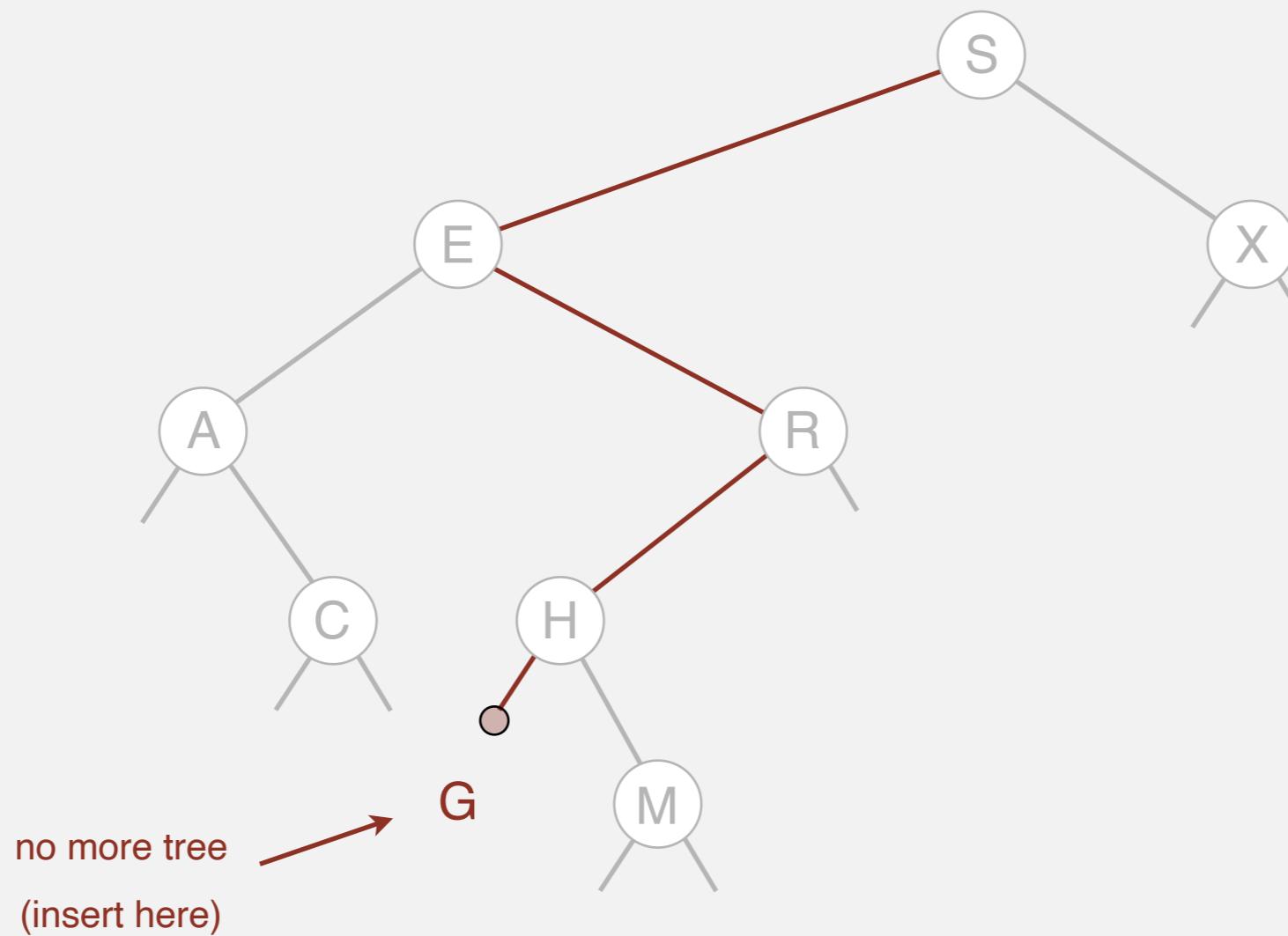
insert G



Binary search tree demo

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

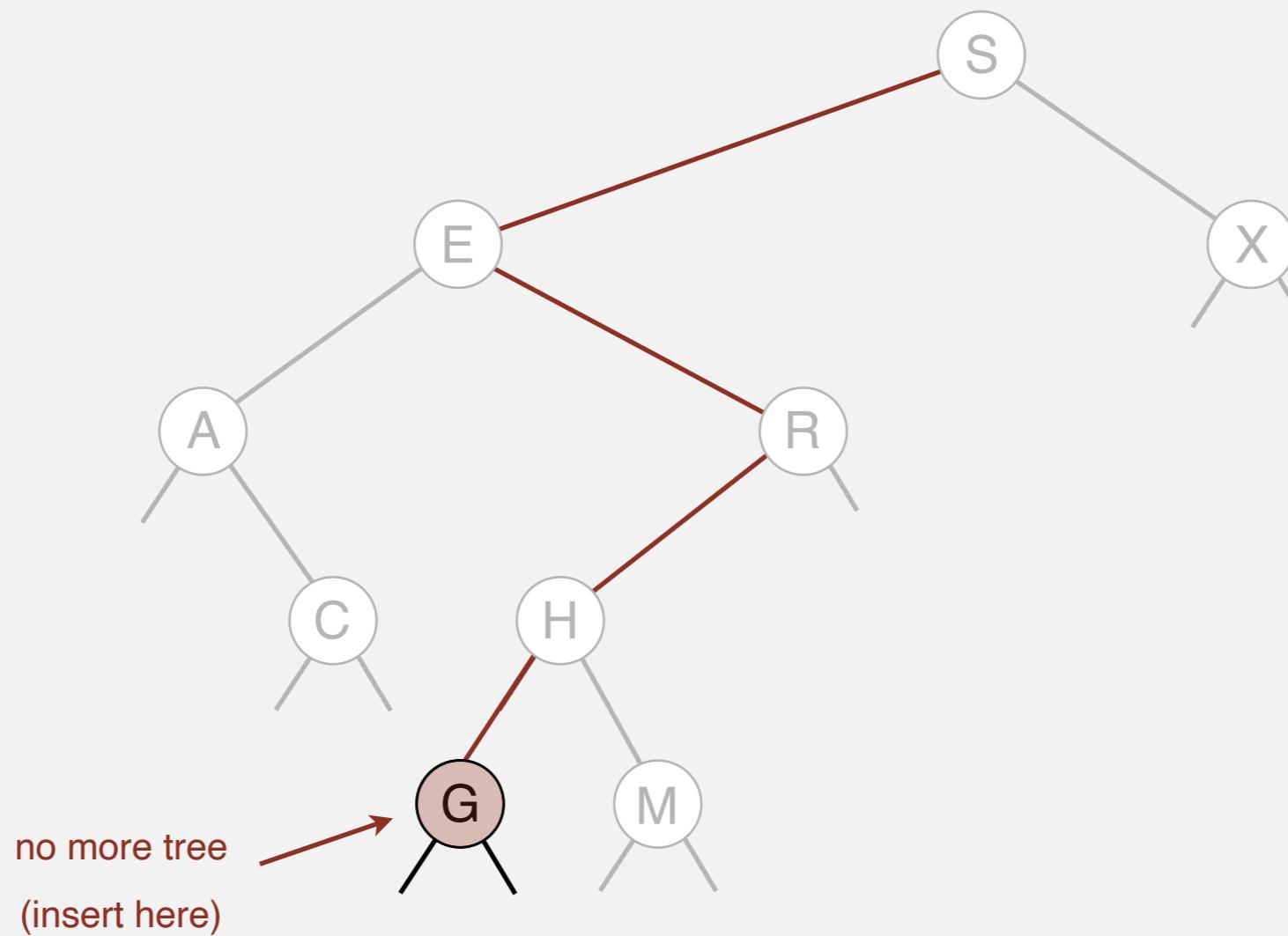
insert G



Binary search tree demo

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

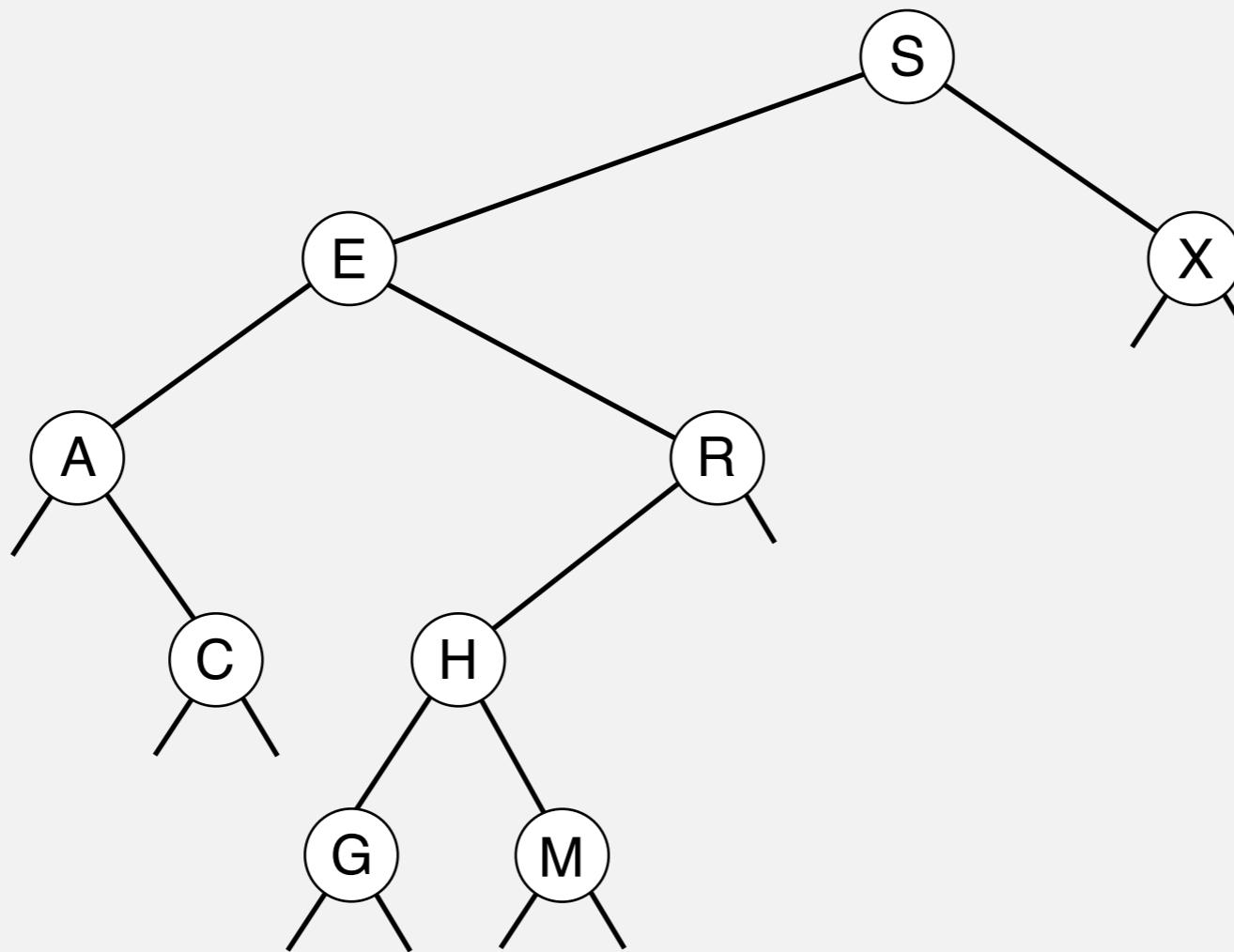
insert G



Binary search tree demo

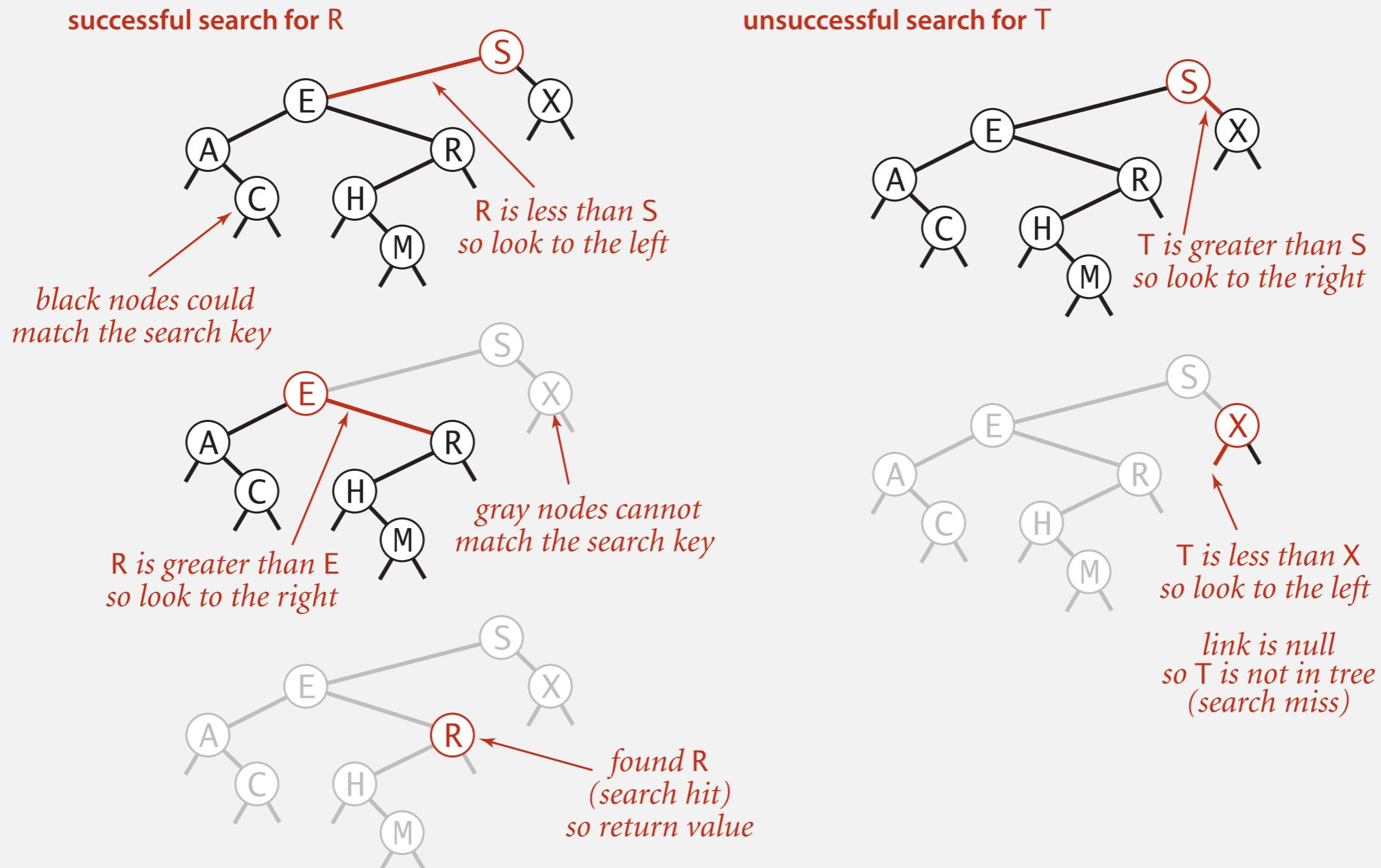
Insert. If less, go left; if greater, go right; if null, insert.

insert G



BST search

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

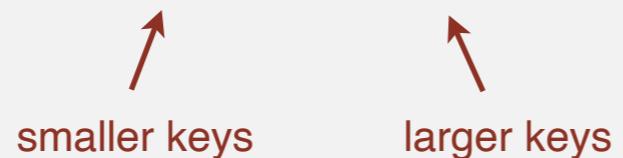


BST representation in Java

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.

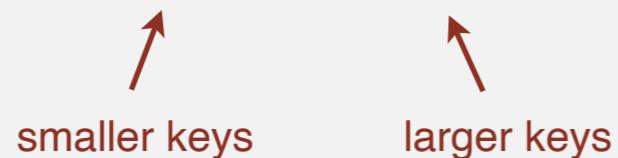


BST representation in Java

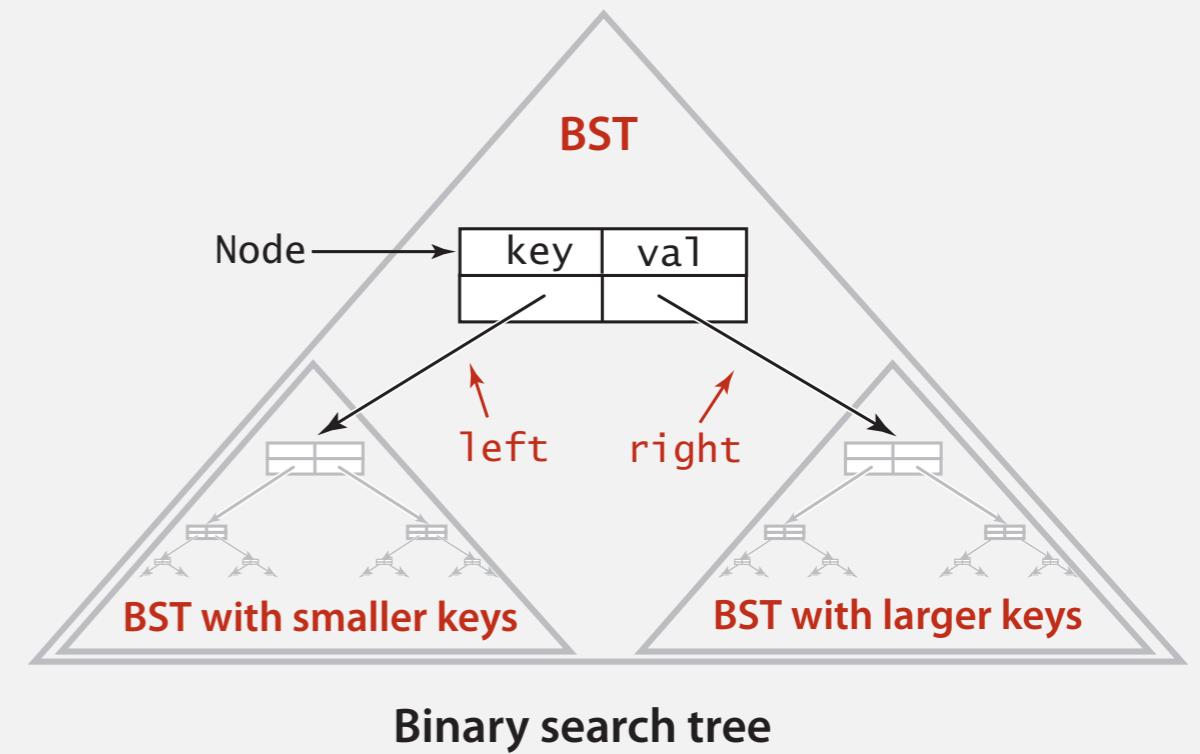
Java definition. A BST is a reference to a root Node.

A Node is composed of four fields:

- A Key and a Value. Key and Value are generic types; Key is Comparable
- A reference to the left and right subtree.



```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```



Binary search tree

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;
}
```

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

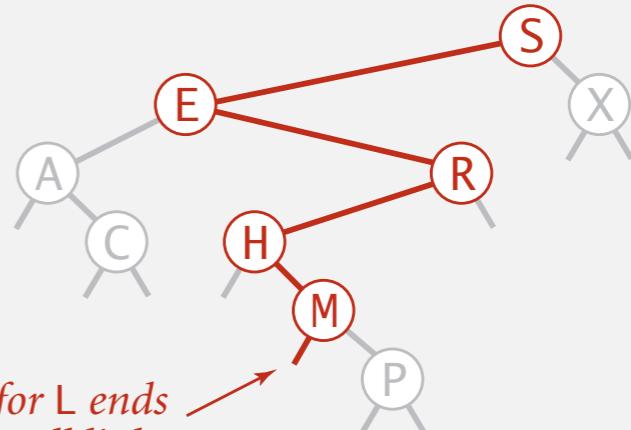
BST insert

Put. Associate value with key.

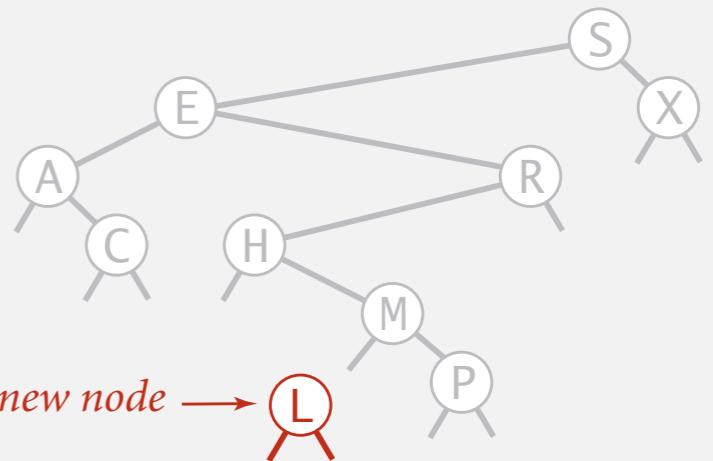
Search for key, then two cases:

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

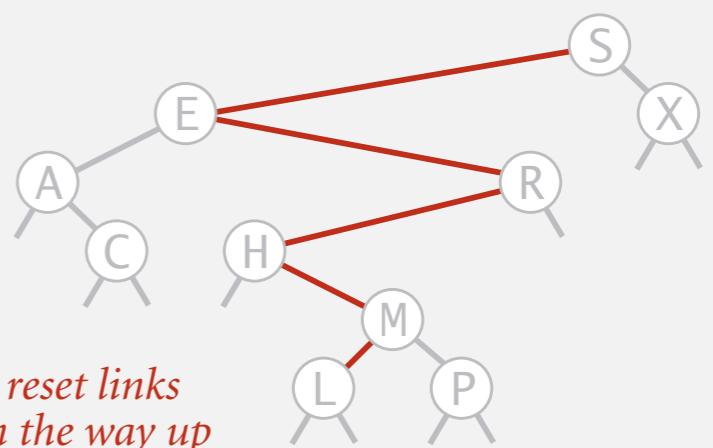
inserting L



*search for L ends
at this null link*



create new node → L



*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); }
```

concise, but tricky,
recursive code;
read carefully!

```
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;
}
```

BST insert: Java implementation

Put. Associate value with key.

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

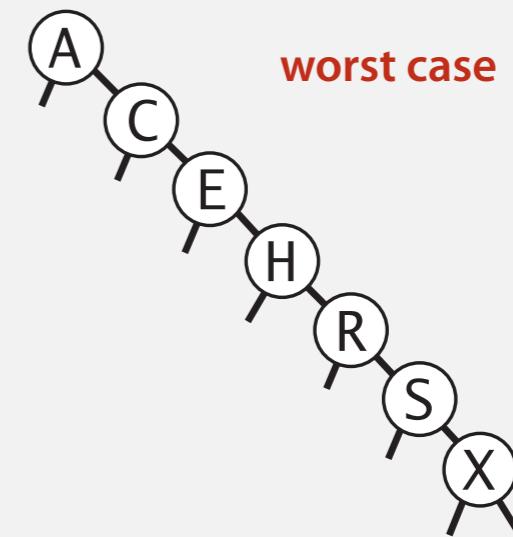
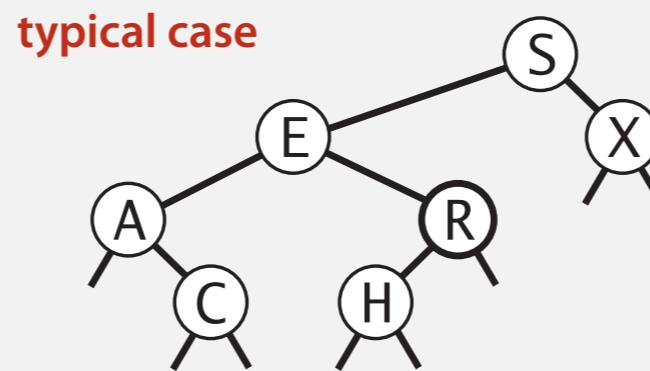
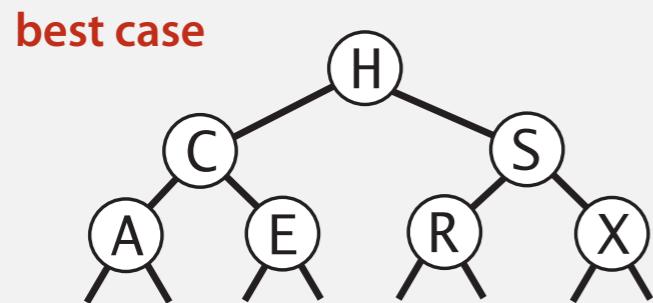
concise, but tricky,
recursive code;
read carefully!

```
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;
}
```

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

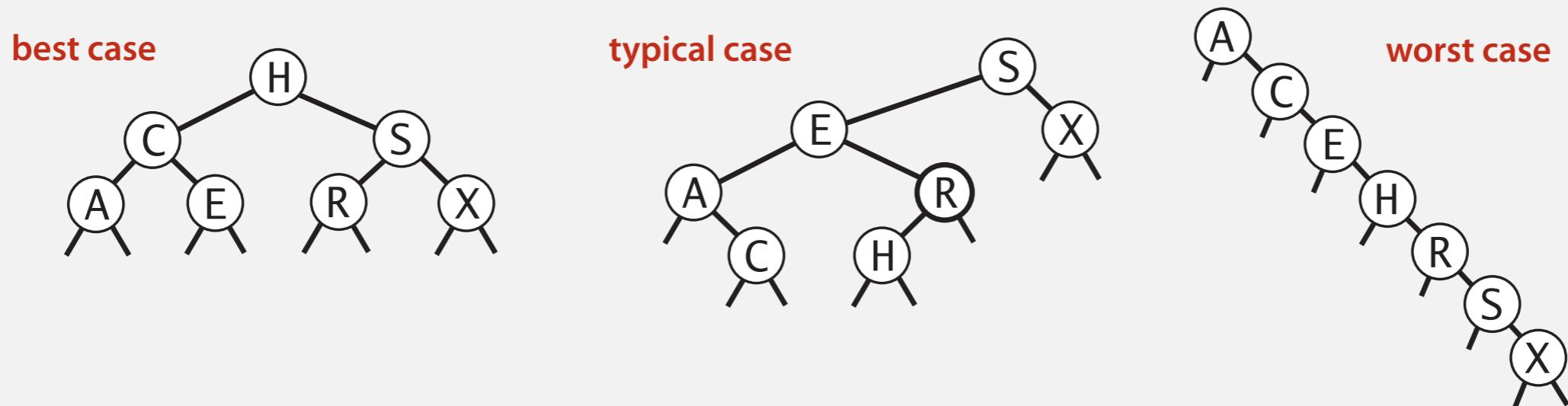
Tree shape

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



Tree shape

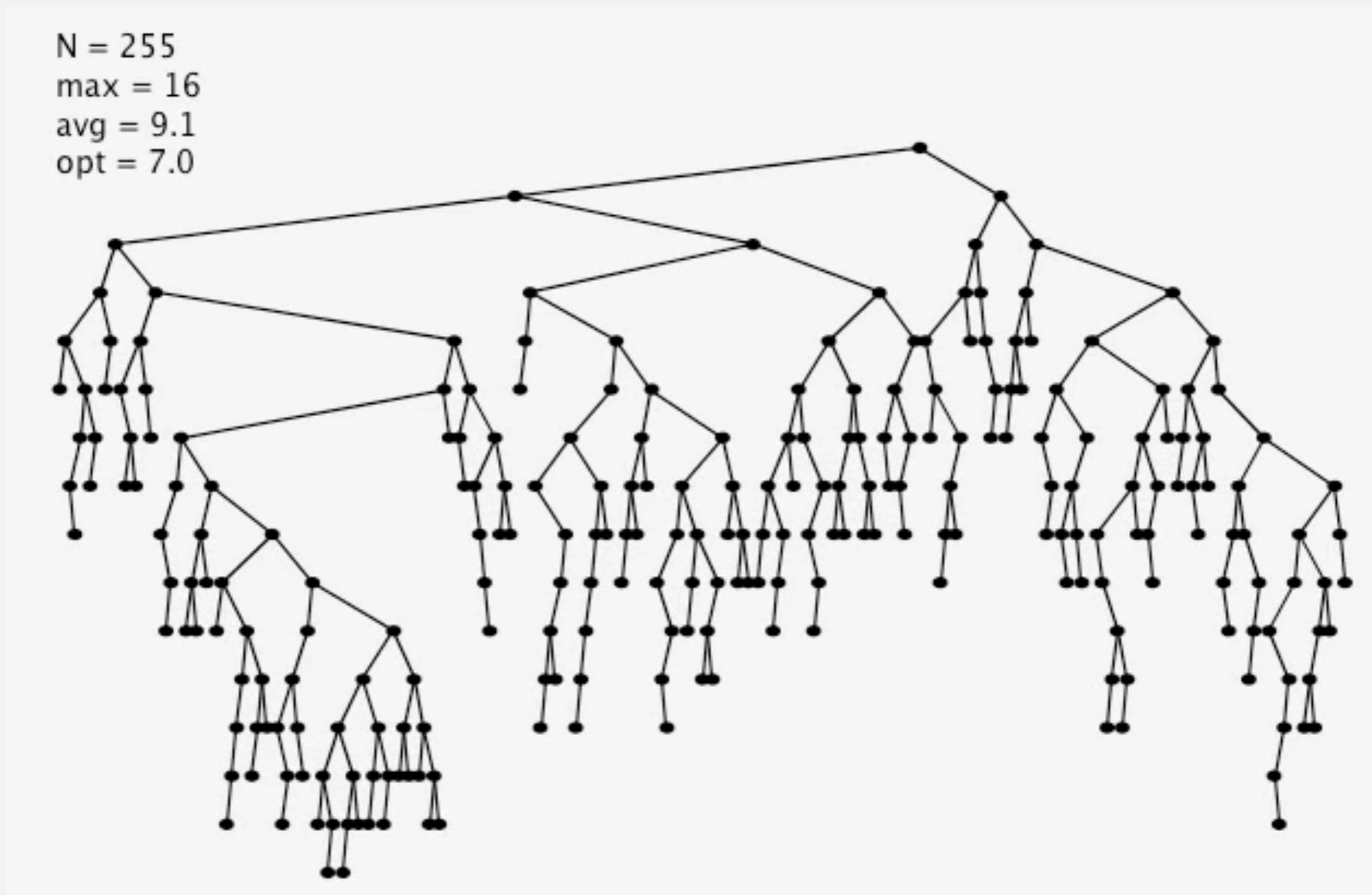
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to $1 + \text{depth of node}$.



Bottom line. Tree shape depends on order of insertion.

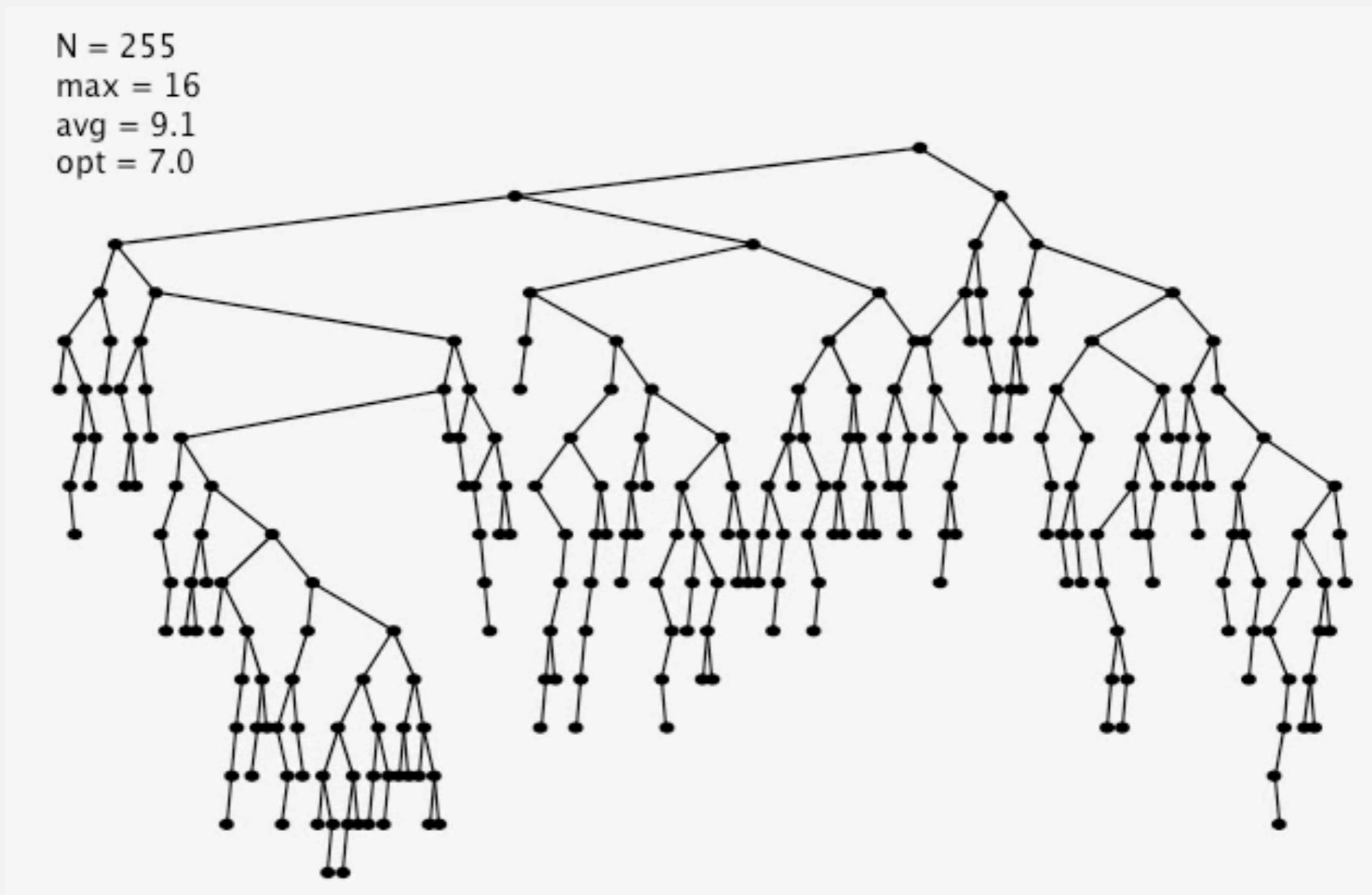
BST insertion: random order visualization

Ex. Insert keys in random order.



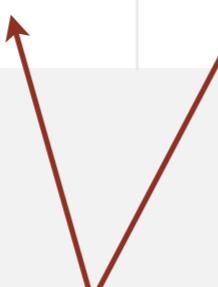
BST insertion: random order visualization

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	<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>

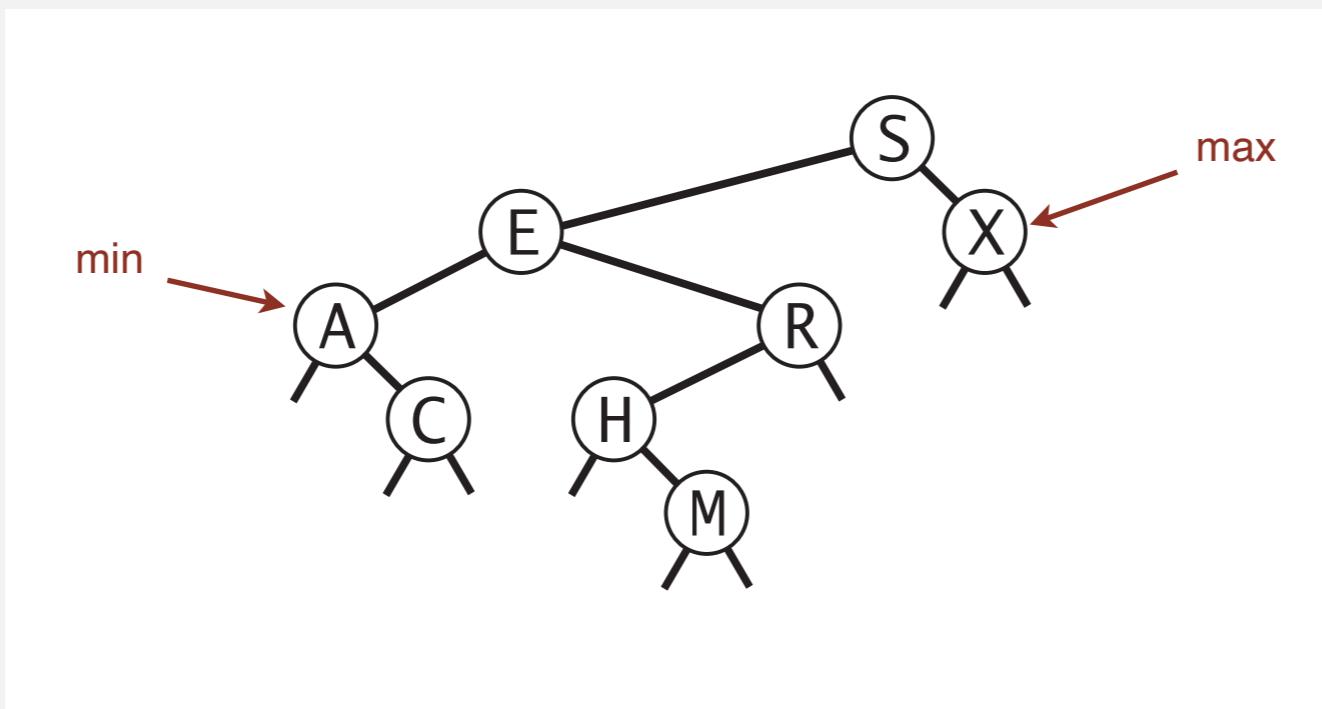


Why not shuffle to ensure a (probabilistic) guarantee of $4.311 \ln N$?

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.



Q. How to find the min / max?

Inorder traversal

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

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;
}

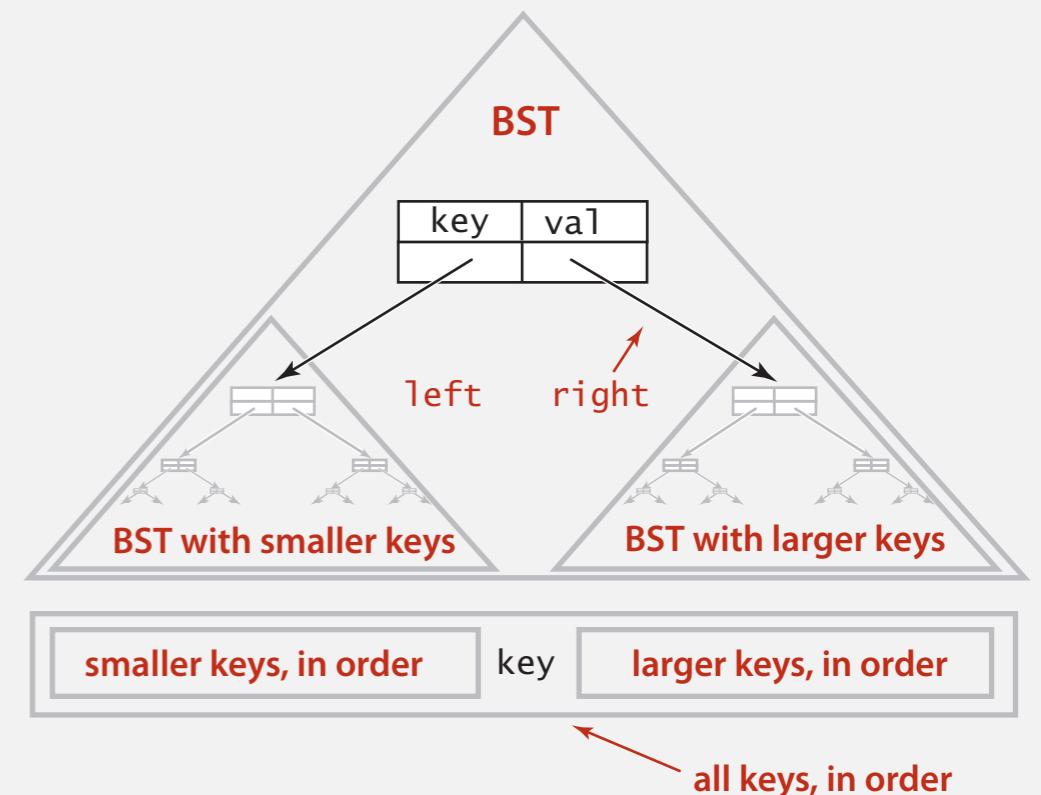
private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```

Inorder traversal

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

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

private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```



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

Inorder traversal

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

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

A
C
E

H
M
R
S
X

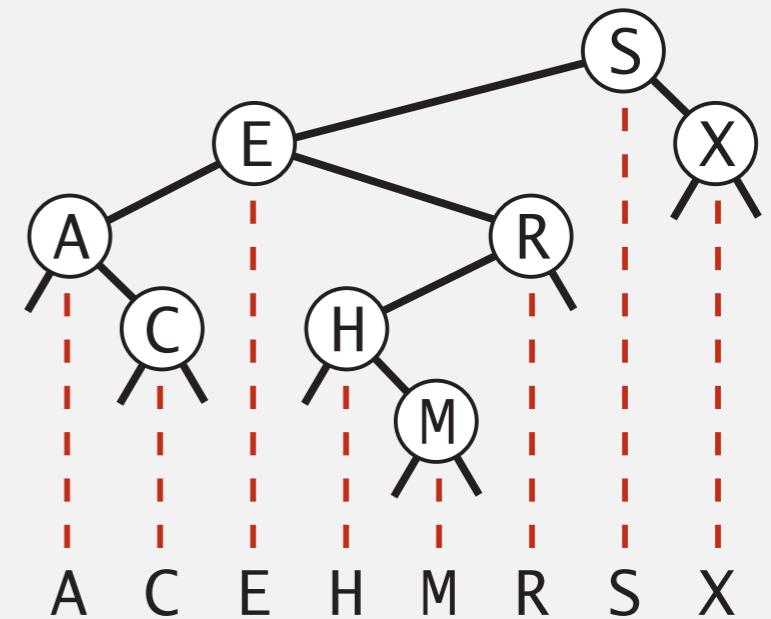
S
S E
S E A

S E A C

S E R
S E R H

S E R H M

S X



recursive calls

queue

function call stack

ST implementations: summary

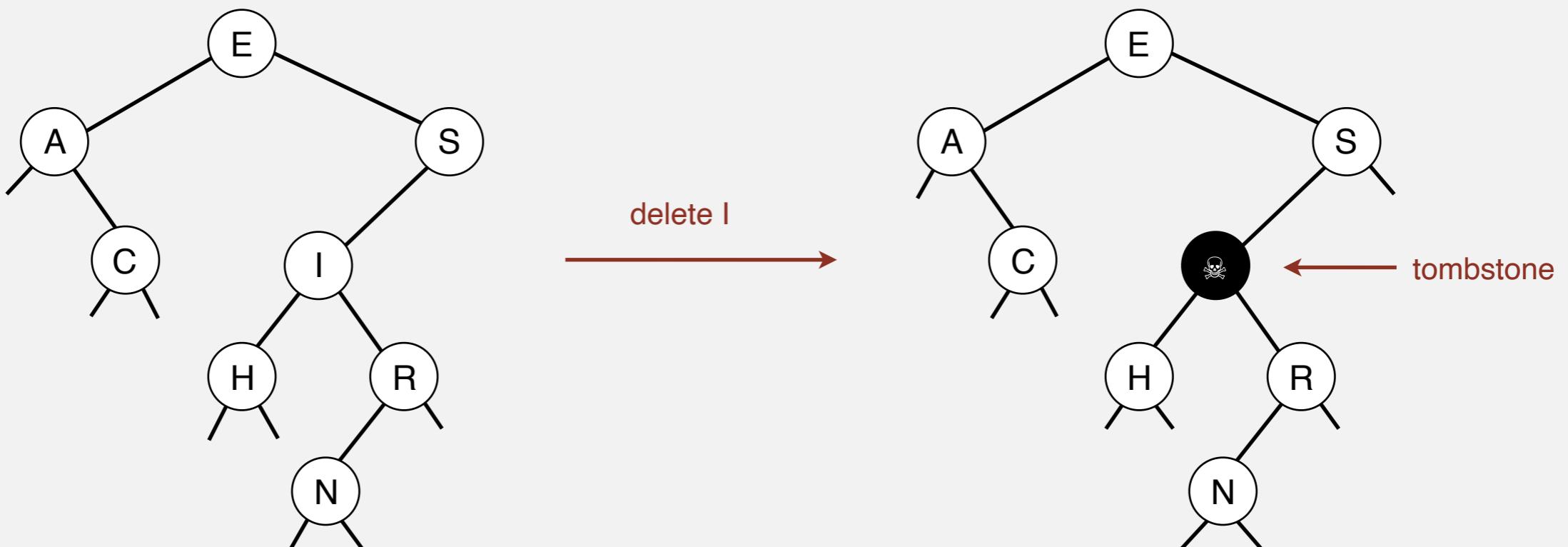
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()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

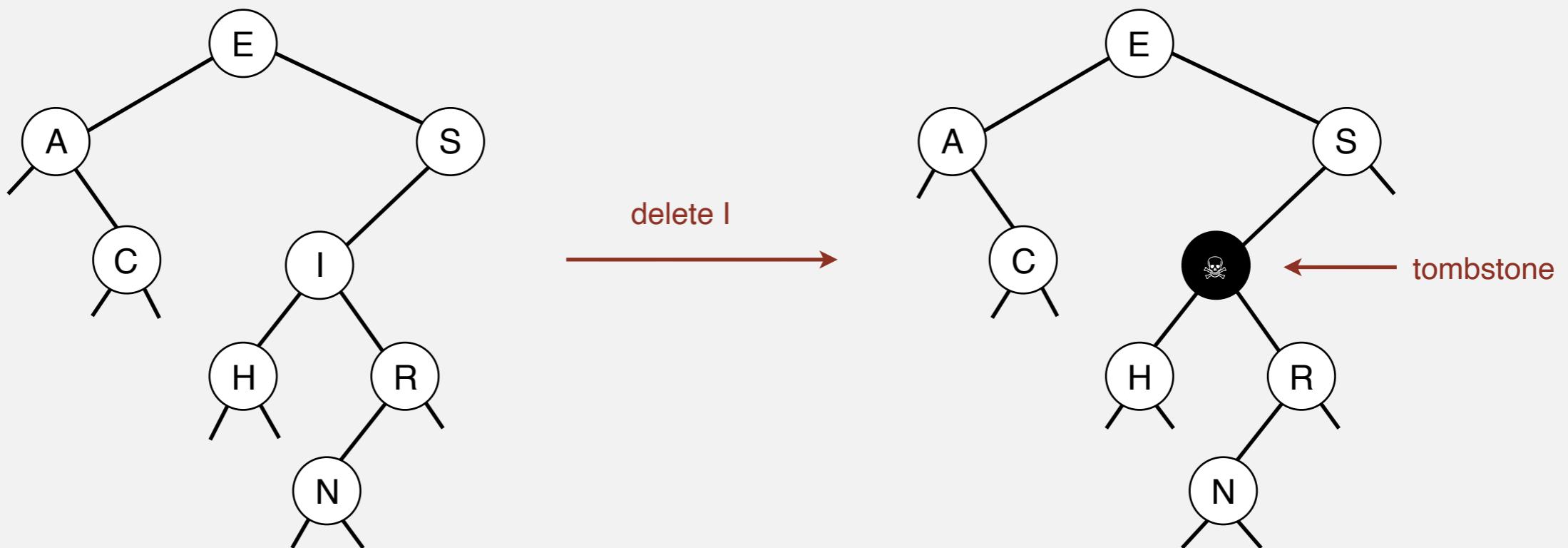
- Set its value to null.
- Leave key in tree to guide search (but don't consider it equal in search).



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).

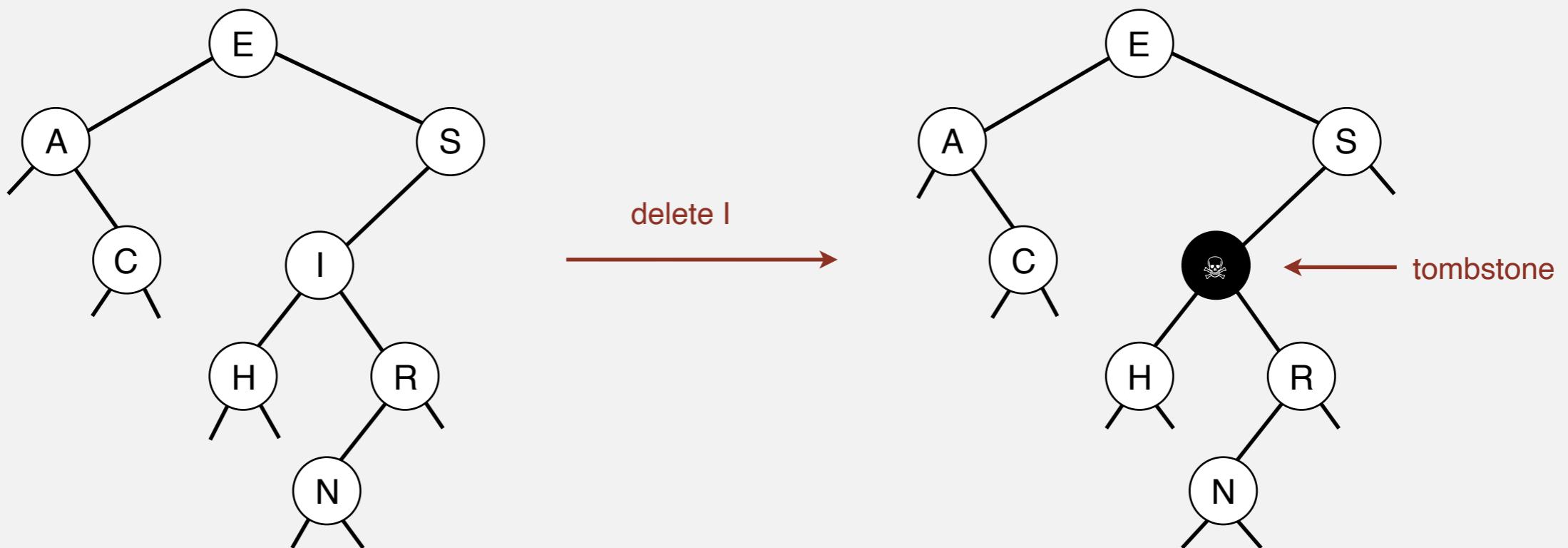


Cost. $\sim 2 \ln N'$ per insert, search, and delete (if keys in random order),
where N' is the number of key-value pairs ever inserted in the BST.

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).



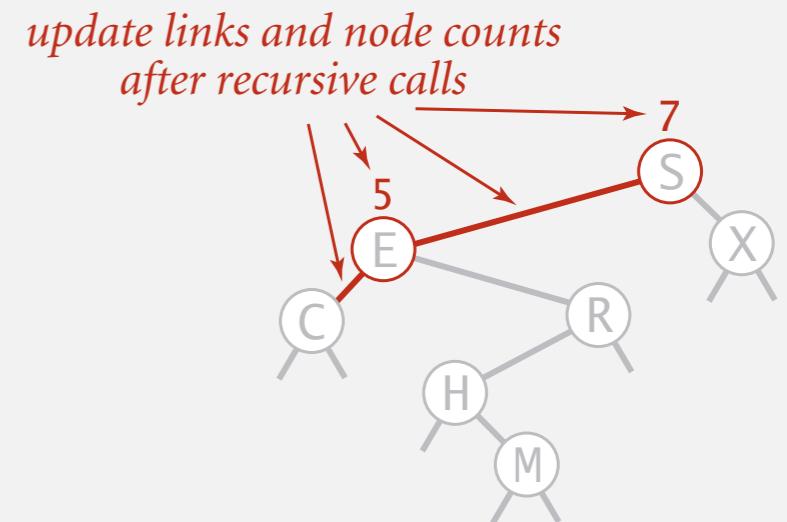
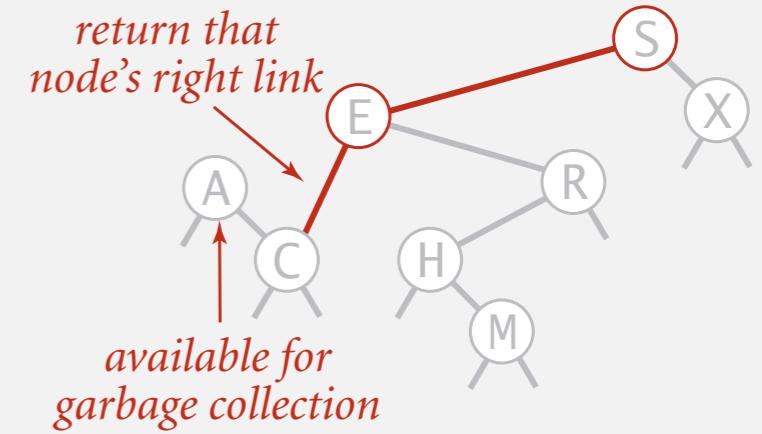
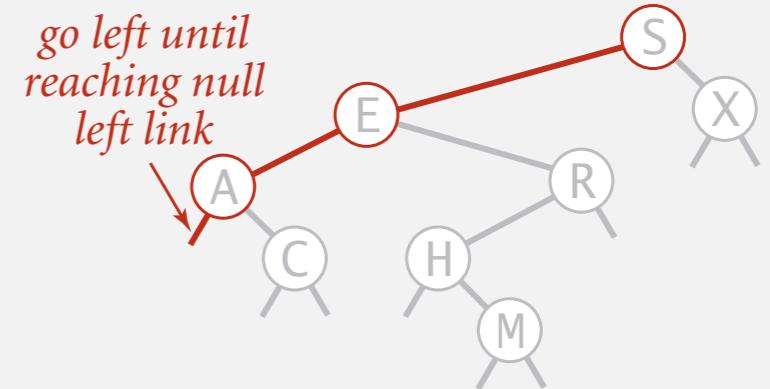
Cost. $\sim 2 \ln N'$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

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.

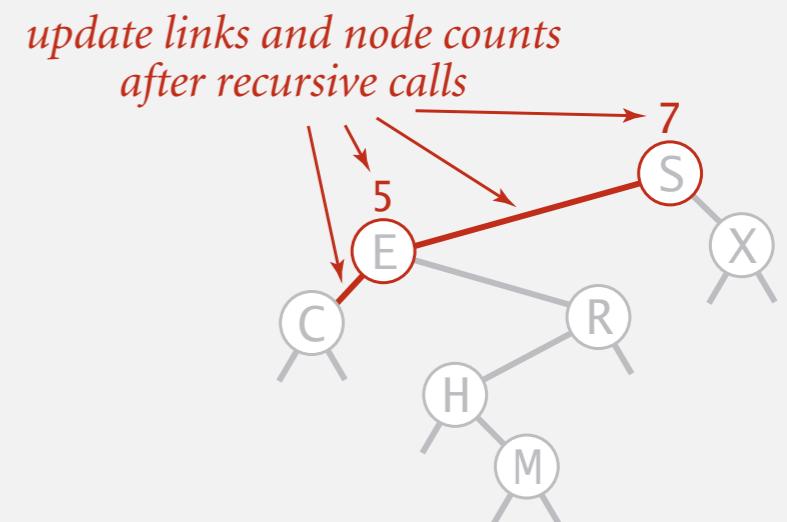
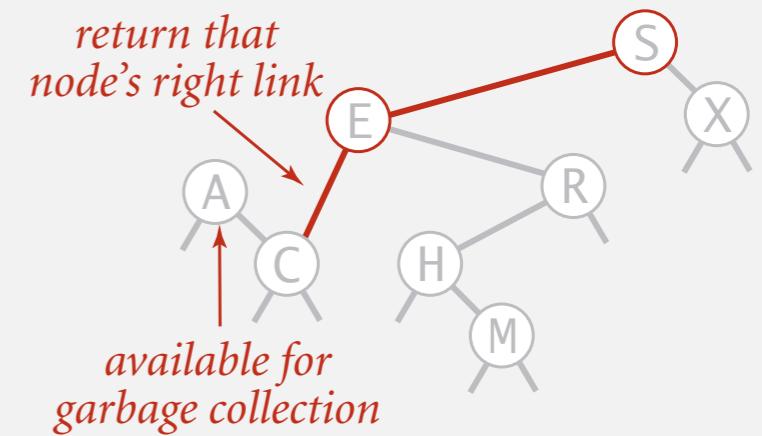
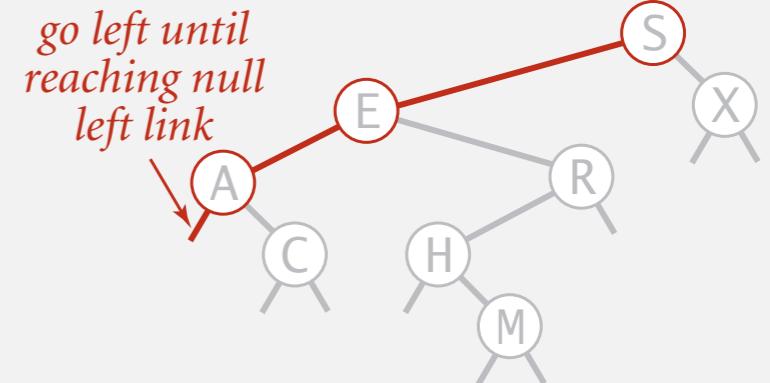


Deleting the minimum

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.

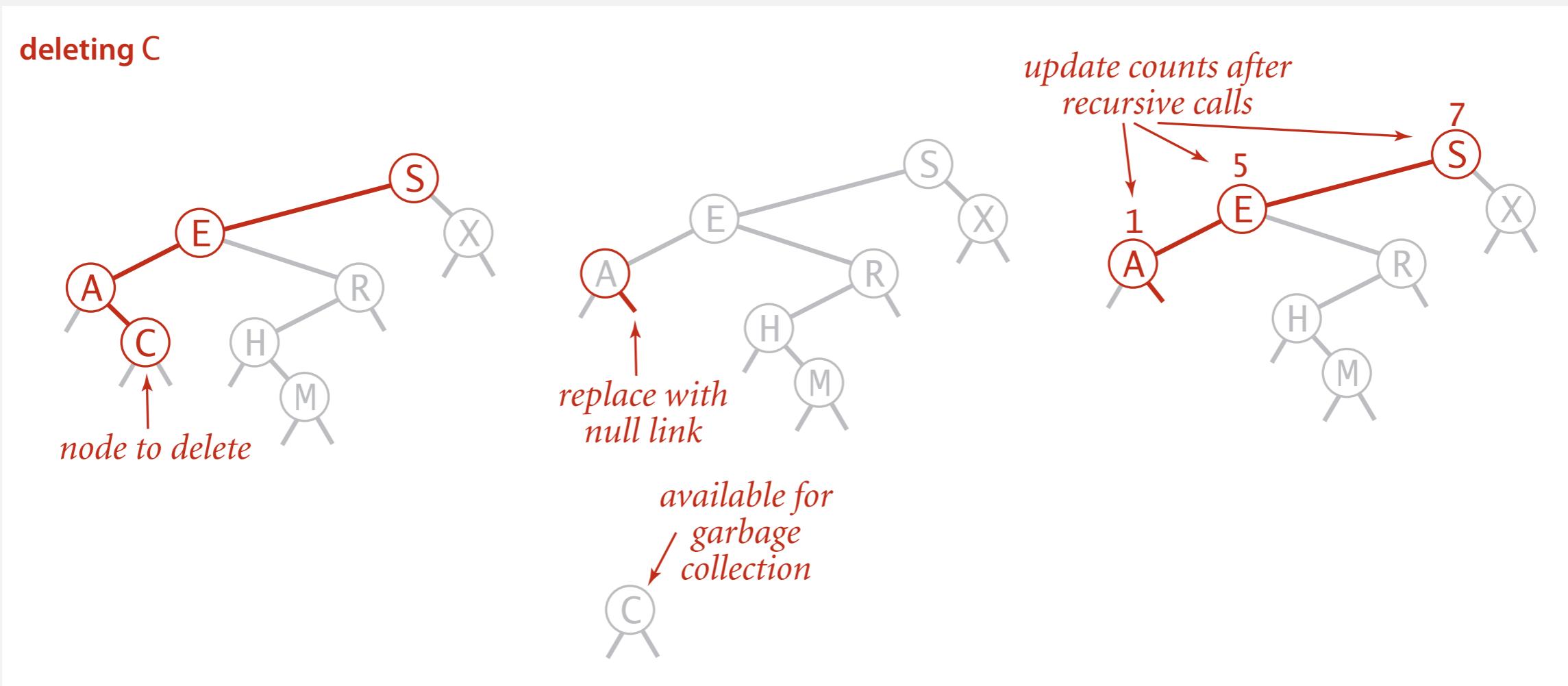
```
public void deleteMin()  
{ root = deleteMin(root); }  
  
private Node deleteMin(Node x)  
{  
    if (x.left == null) return x.right;  
    x.left = deleteMin(x.left);  
    x.count = 1 + size(x.left) + size(x.right);  
    return x;  
}
```



Hibbard deletion

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

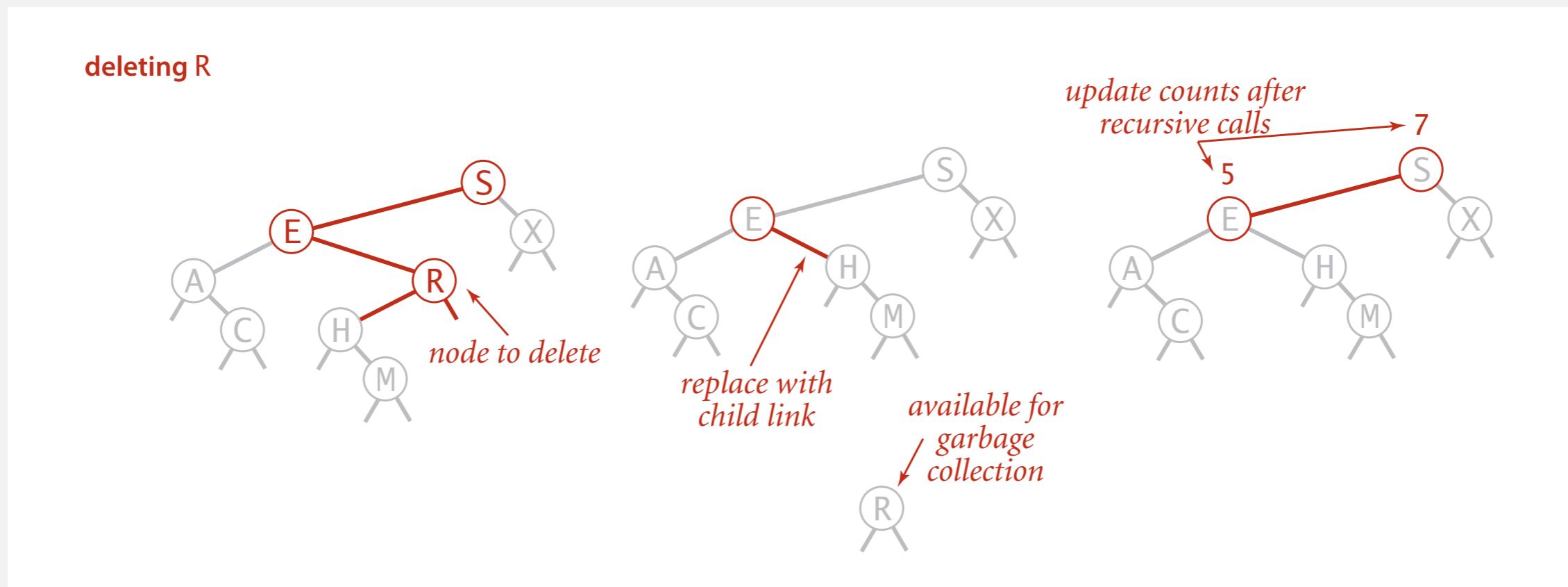
Case 0. [0 children] Delete t by setting parent link to null.



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.



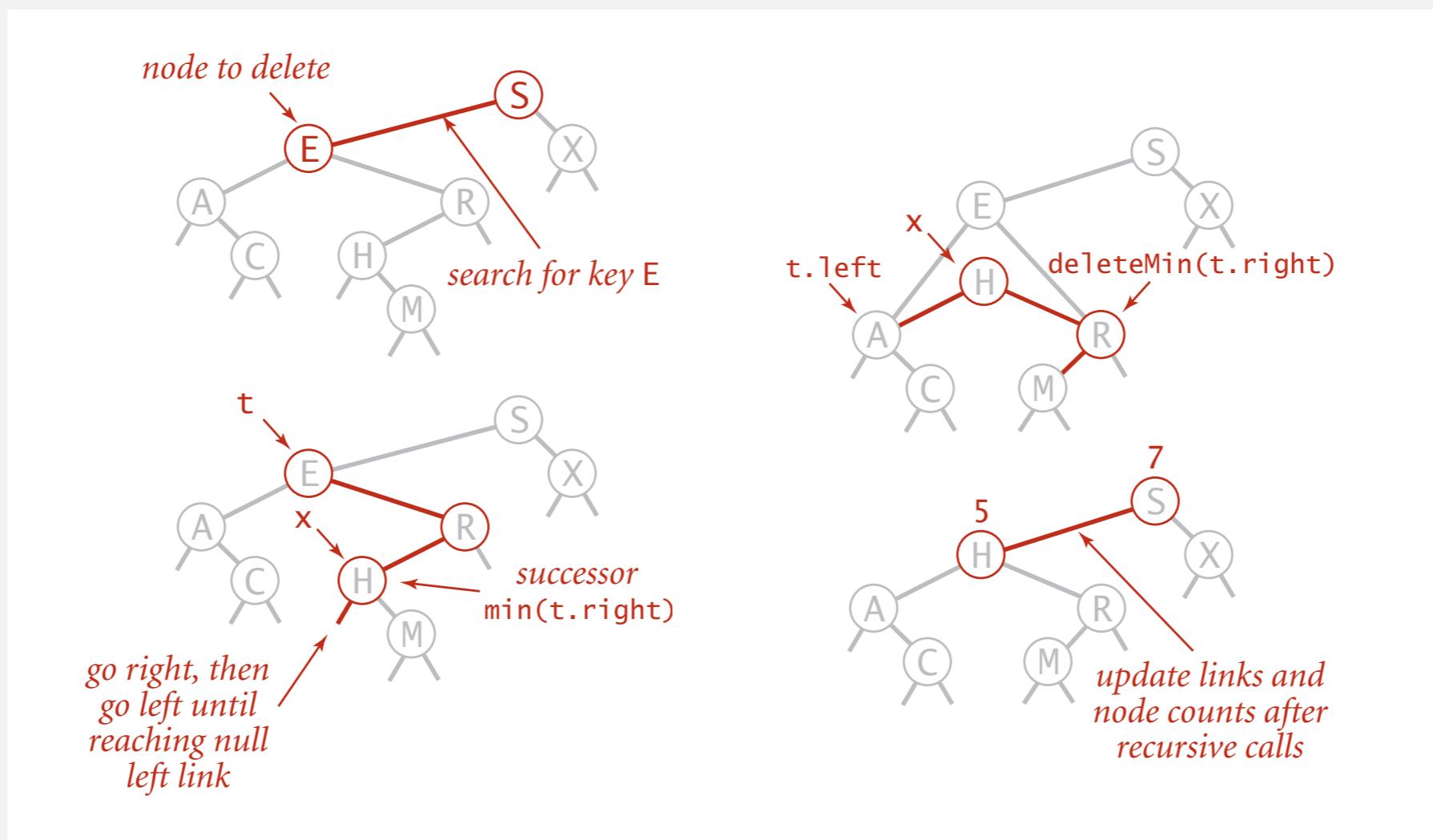
Hibbard deletion

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;
}
```

The diagram illustrates the search for a key and the replacement of a node with its successor during deletion. A vertical grey bar represents the current node being processed. Red arrows point from the right towards the node, indicating the search path. The code segments are annotated with these arrows:

- A red arrow points to the first if-statement (`x == null`) with the label "search for key".
- Two red arrows point to the assignments `x = min(t.right)` and `x.right = deleteMin(t.right)` with the label "no right child".
- Two red arrows point to the assignments `x = t` and `x.left = t.left` with the label "no left child".
- A red arrow points to the assignment `x = min(t.right)` with the label "replace with successor".

ST implementations: summary

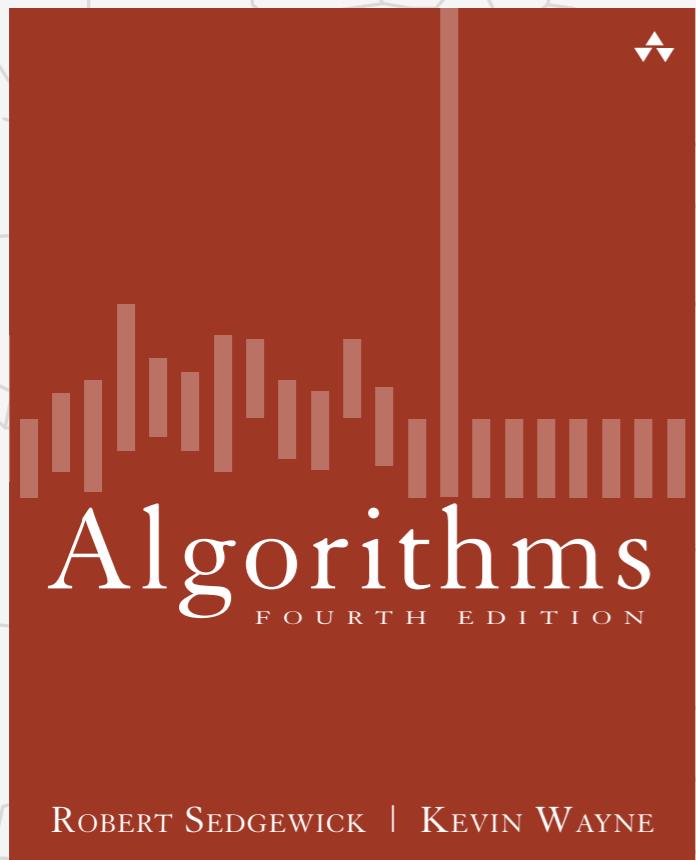
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()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

Algorithms

FARAAZ SARESHWALA



ROBERT SEDGEWICK | KEVIN WAYNE

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

3.4 HASH TABLES

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ context

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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>

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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>

Q. Can we do better?

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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>

Q. Can we do better?

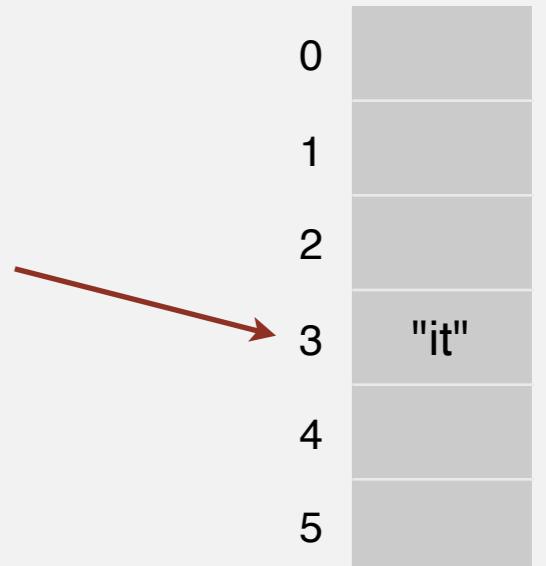
A. Yes, but with different access to the data.

Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.

$\text{hash("it")} = 3$

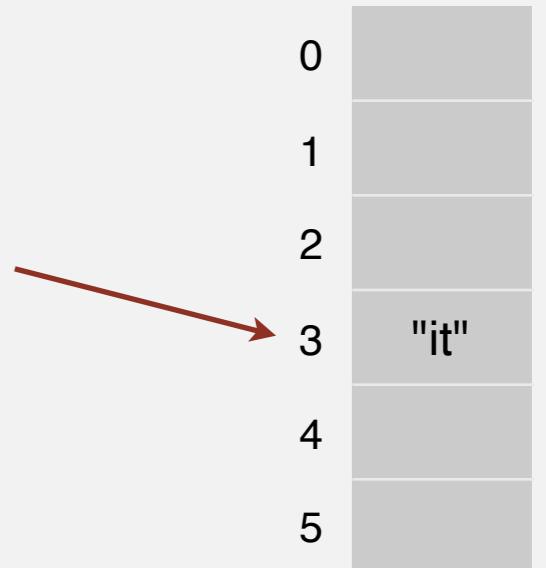


Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.

$\text{hash("it")} = 3$

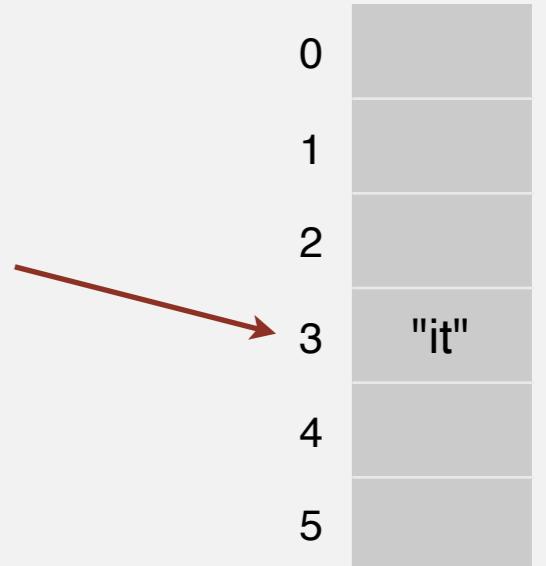


Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.

$\text{hash("it")} = 3$



Issues.

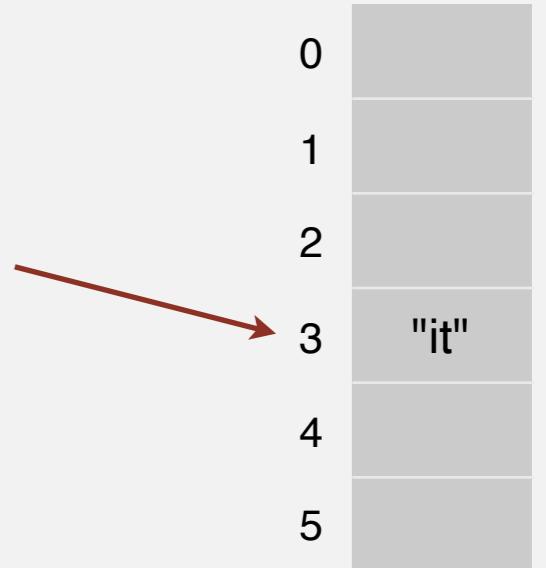
- Computing the hash function.

Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.

hash("it") = 3



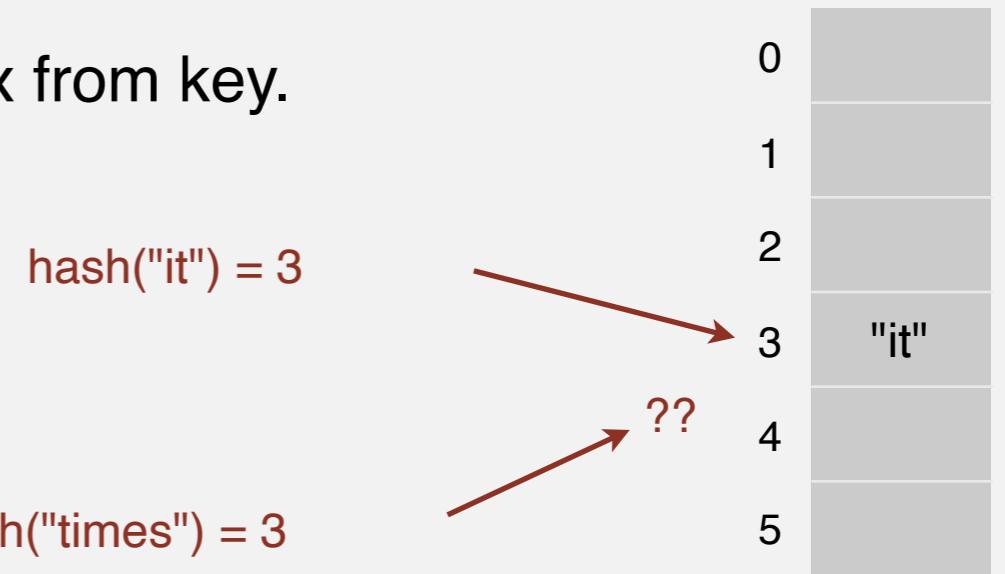
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.



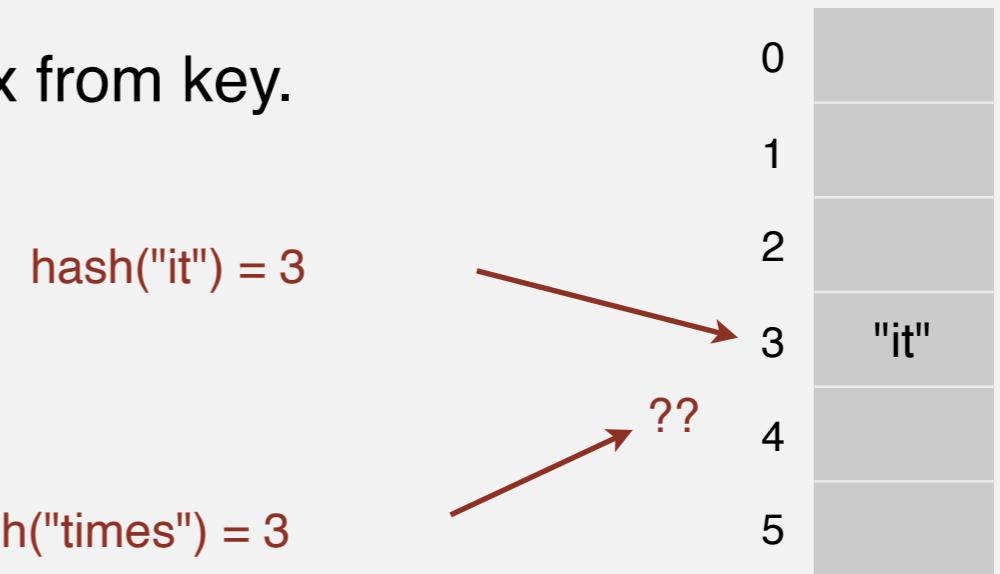
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

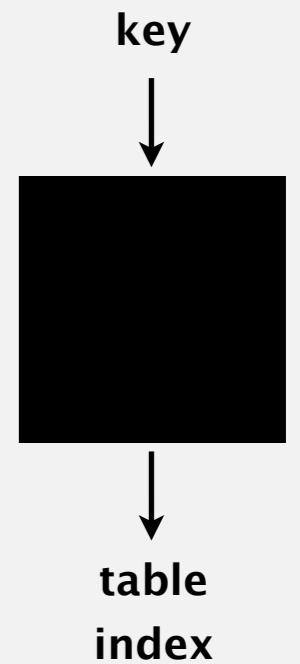
<http://algs4.cs.princeton.edu>

3.4 HASH TABLES

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ context

Computing the hash function

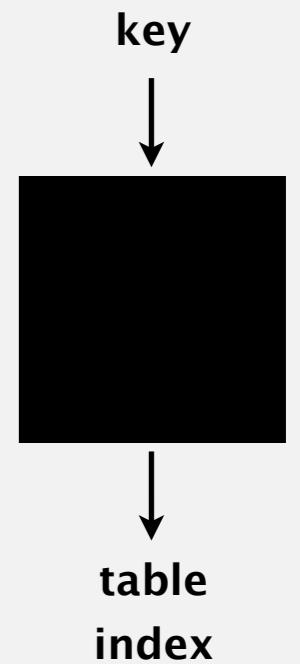
Idealistic goal. Scramble the keys uniformly to produce a table index.



Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.

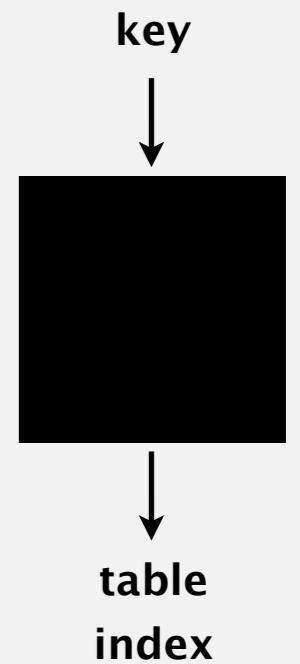


Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



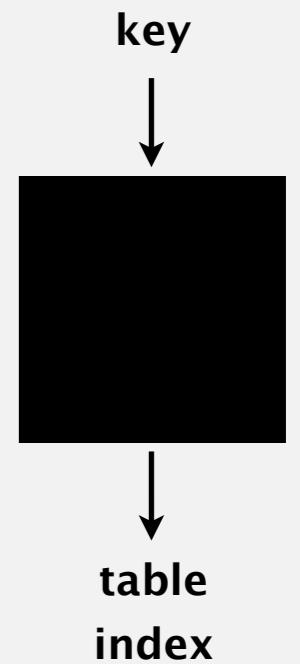
Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications

Ex 1. Phone numbers.

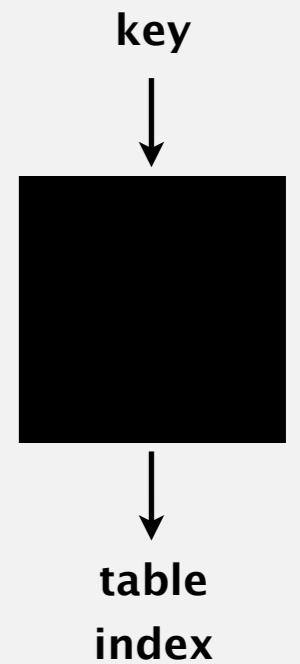


Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

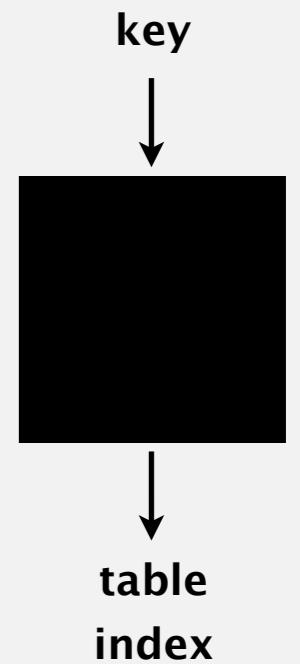
- Bad: first three digits.

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

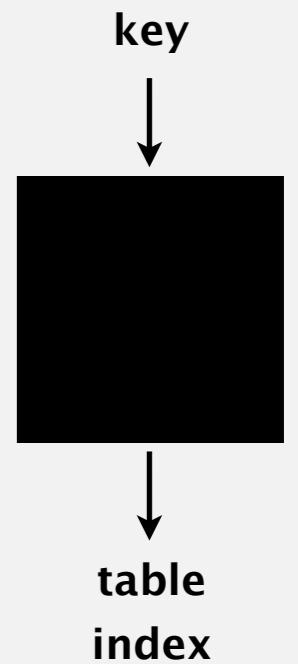
- Bad: first three digits.
- Better: last three digits.

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

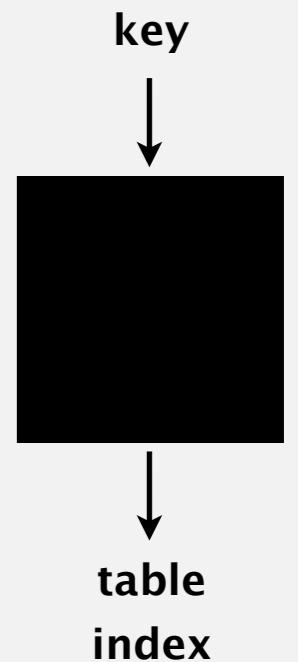
Ex 2. Social Security numbers.

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

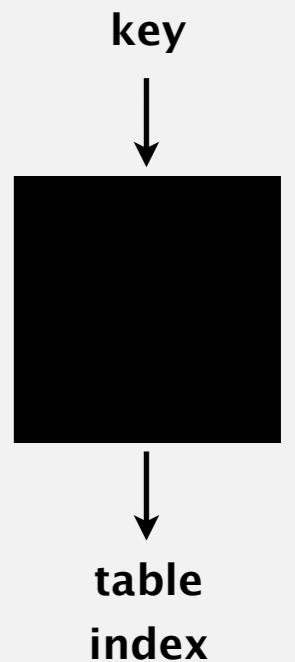
- Bad: first three digits. ← 573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

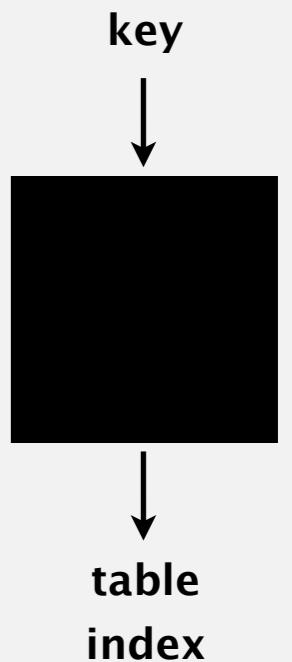
- Bad: first three digits. ← 573 = California, 574 = Alaska
(assigned in chronological order within geographic region)
- Better: last three digits.

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

- Bad: first three digits. ← 573 = California, 574 = Alaska
(assigned in chronological order within geographic region)
- Better: last three digits.

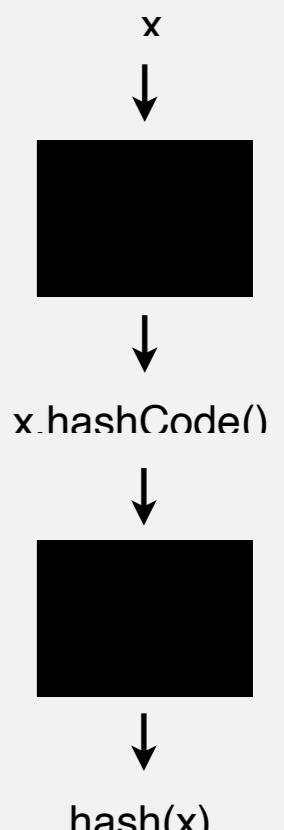
Practical challenge. Need different approach for each key type.

Modular hashing

Hash code. An int between -2^{31} and $2^{31} - 1$.

Hash function. An int between 0 and $M - 1$ (for use as array index).

typically a prime or power of 2



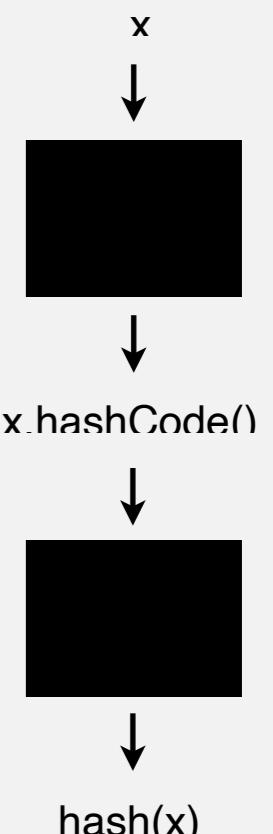
Modular hashing

Hash code. An int between -2^{31} and $2^{31} - 1$.

Hash function. An int between 0 and $M - 1$ (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)  
{ return key.hashCode() % M; }
```



Modular hashing

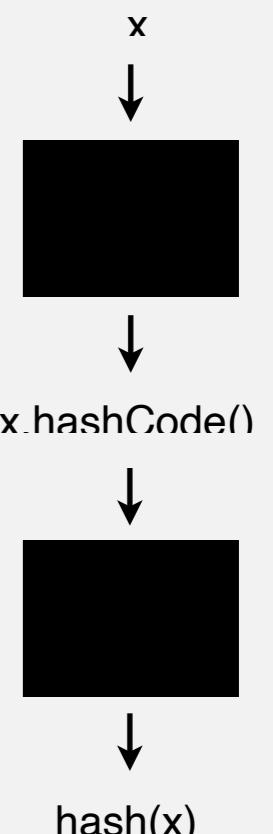
Hash code. An int between -2^{31} and $2^{31} - 1$.

Hash function. An int between 0 and $M - 1$ (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)  
{ return key.hashCode() % M; }
```

bug



Modular hashing

Hash code. An int between -2^{31} and $2^{31} - 1$.

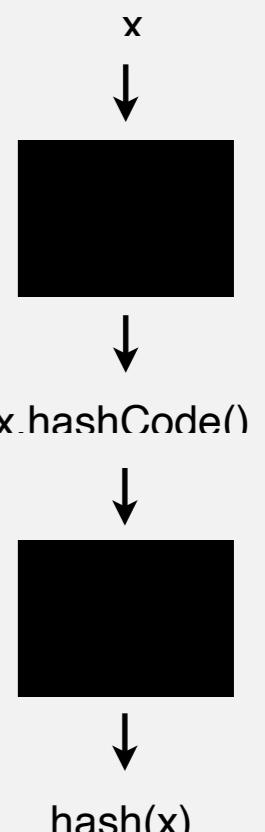
Hash function. An int between 0 and $M - 1$ (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)  
{ return key.hashCode() % M; }
```

bug

```
private int hash(Key key)  
{ return Math.abs(key.hashCode()) % M; }
```



Modular hashing

Hash code. An int between -2^{31} and $2^{31} - 1$.

Hash function. An int between 0 and $M - 1$ (for use as array index).

typically a prime or power of 2

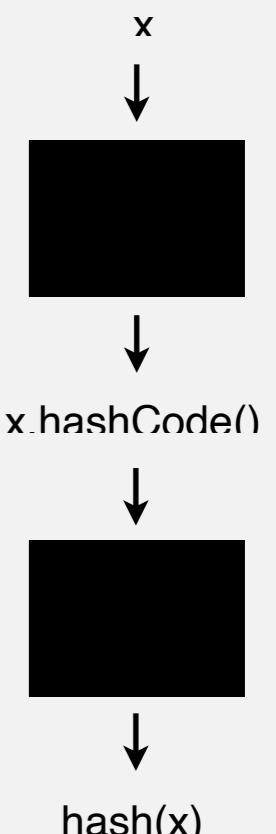
```
private int hash(Key key)  
{ return key.hashCode() % M; }
```

bug

```
private int hash(Key key)  
{ return Math.abs(key.hashCode()) % M; }
```

1-in-a-billion bug

hashCode() of "polygenelubricants" is -2^{31}



Modular hashing

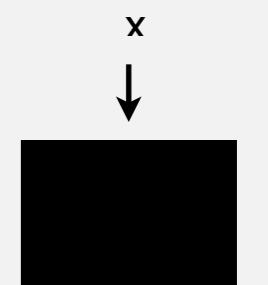
Hash code. An int between -2^{31} and $2^{31} - 1$.

Hash function. An int between 0 and $M - 1$ (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)  
{ return key.hashCode() % M; }
```

bug



`x.hashCode()`

```
private int hash(Key key)  
{ return Math.abs(key.hashCode()) % M; }
```

1-in-a-billion bug



`hash(x)`

hashCode() of "polygenelubricants" is -2^{31}

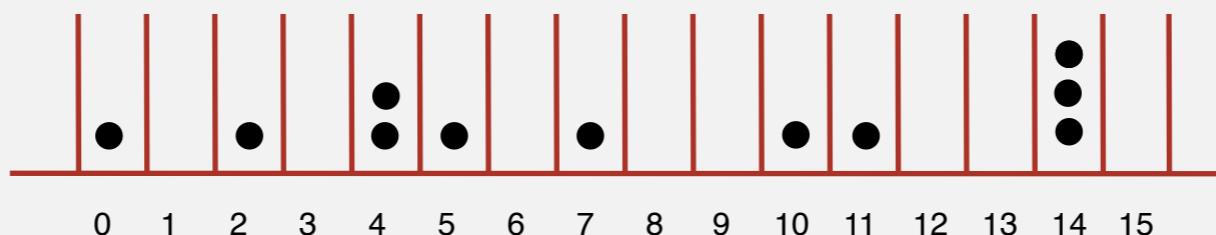
```
private int hash(Key key)  
{ return (key.hashCode() & 0xffffffff) % M; }
```

correct

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Java's String data uniformly distribute the keys of Tale of Two Cities

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

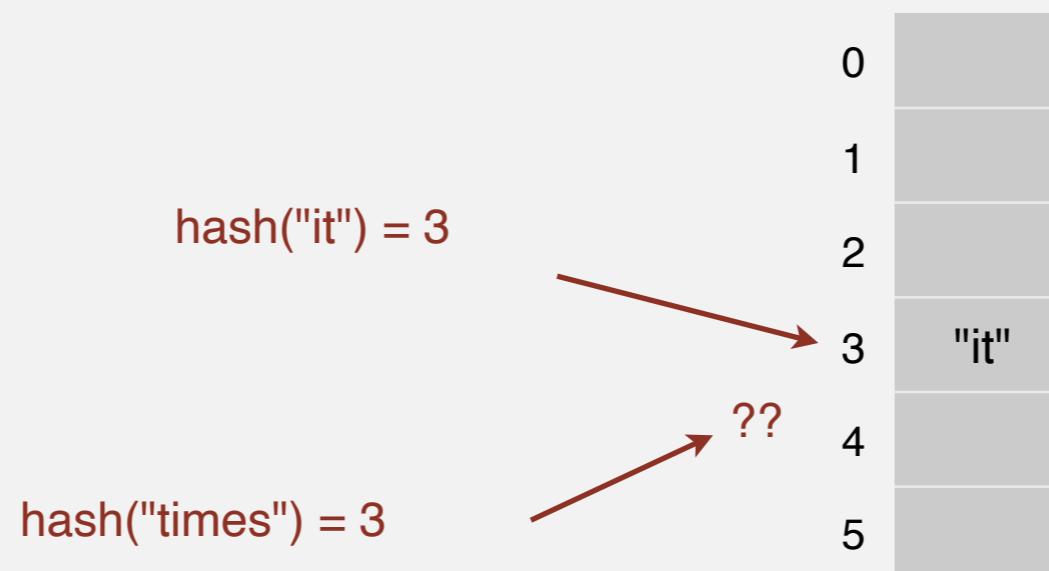
<http://algs4.cs.princeton.edu>

3.4 HASH TABLES

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ context

Collisions

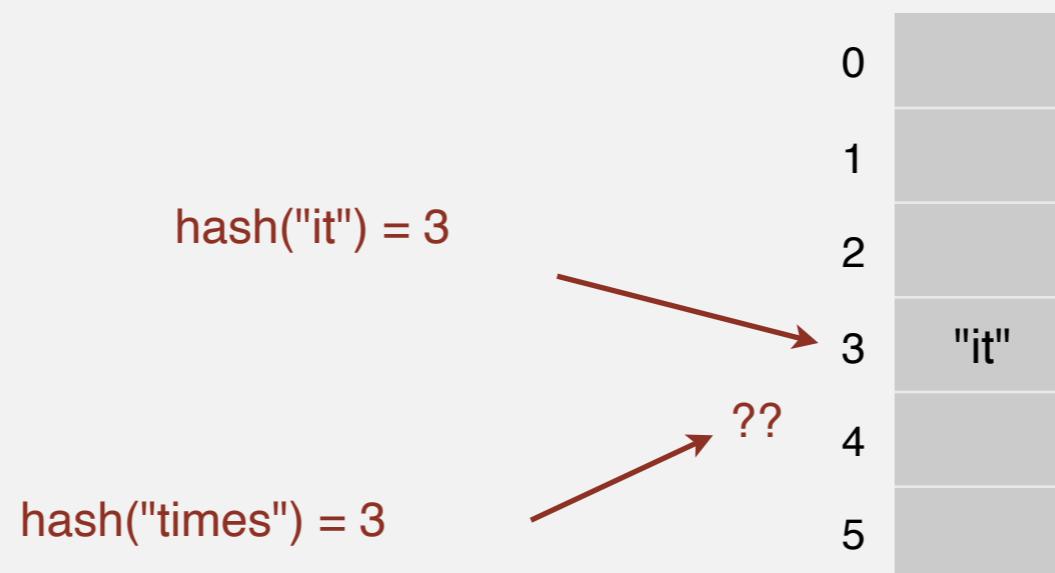
Collision. Two distinct keys hashing to same index.



Collisions

Collision. Two distinct keys hashing to same index.

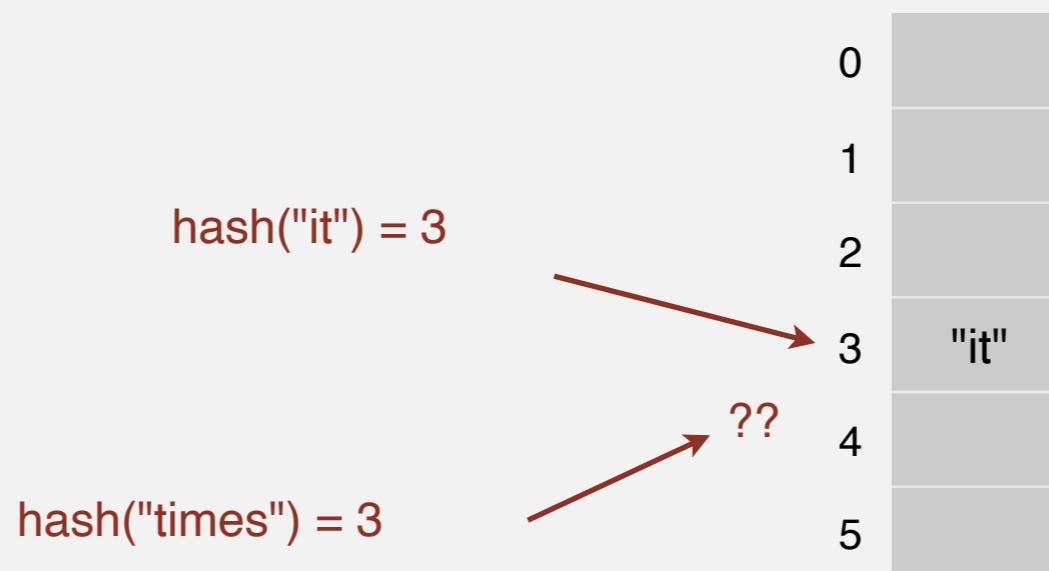
- Observation: can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.



Collisions

Collision. Two distinct keys hashing to same index.

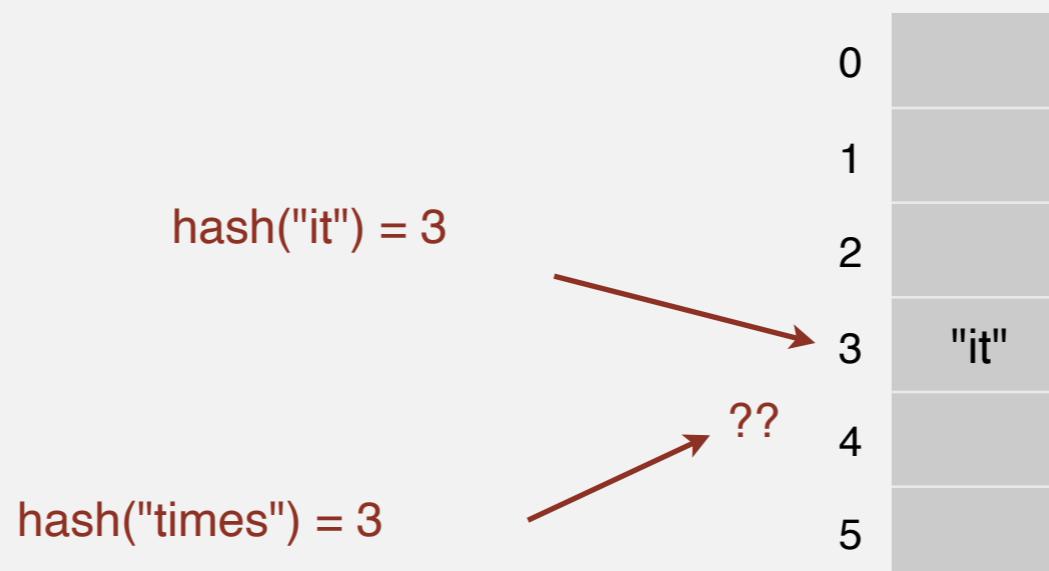
- Observation: can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Observation: collisions are evenly (uniformly) distributed if our hash function is.



Collisions

Collision. Two distinct keys hashing to same index.

- Observation: can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Observation: collisions are evenly (uniformly) distributed if our hash function is.

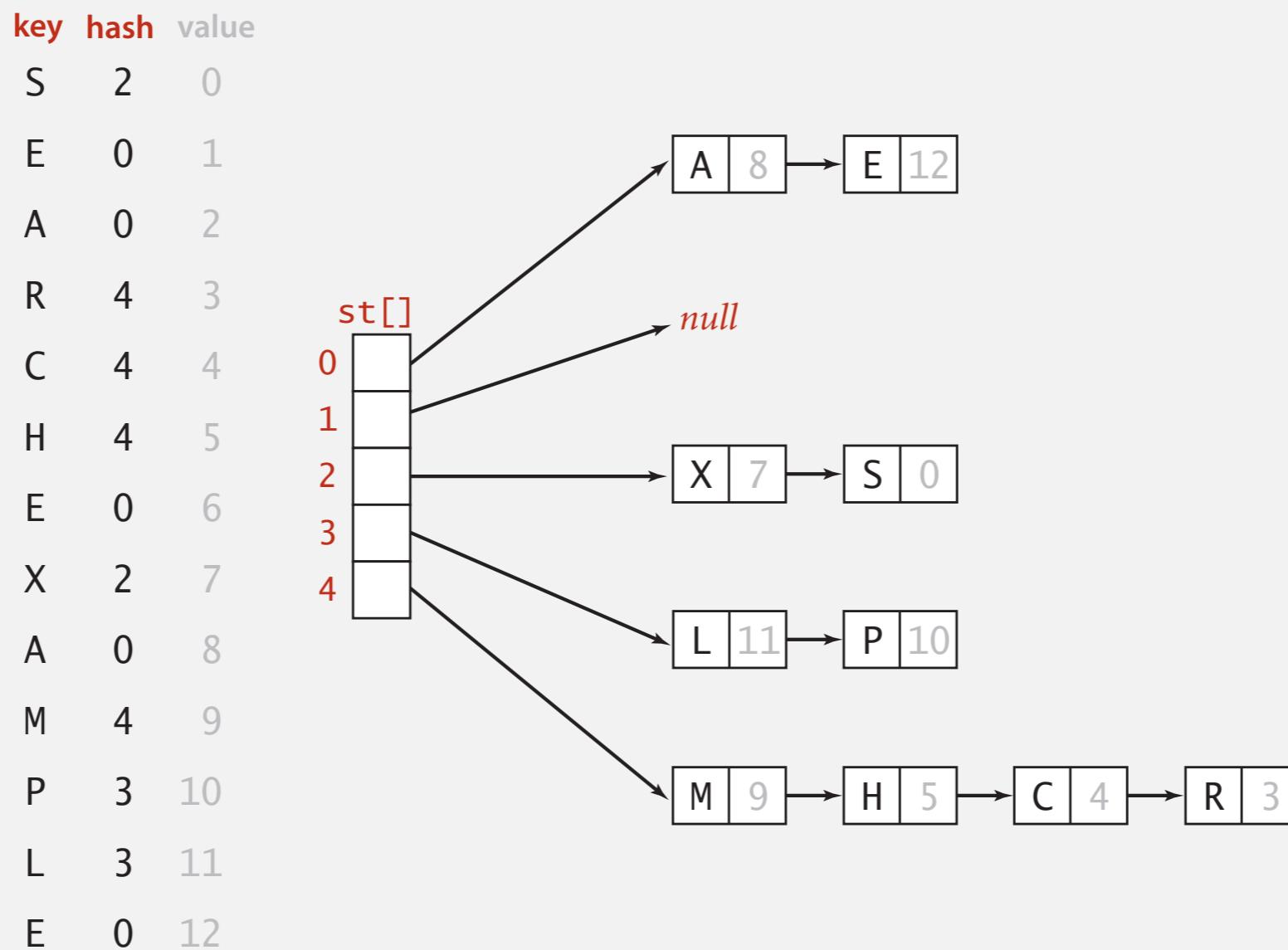


Challenge. Deal with collisions efficiently.

Separate-chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: need to search only i^{th} chain.



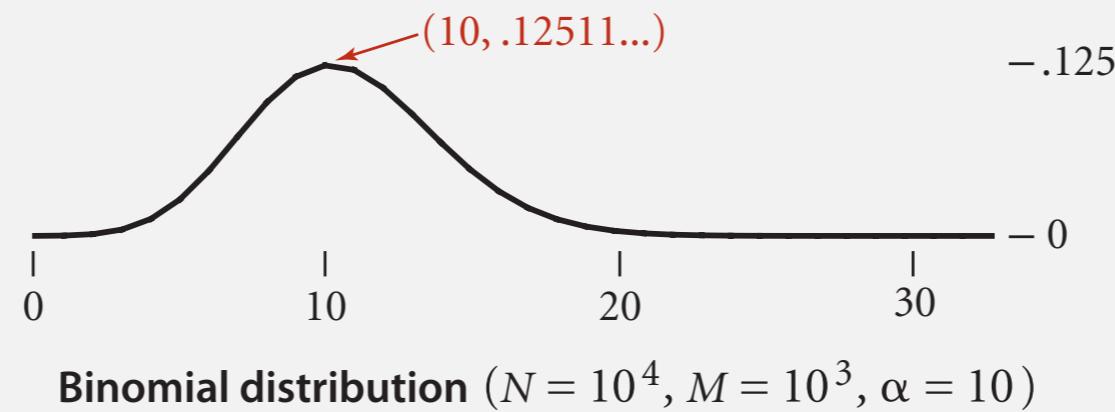
Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

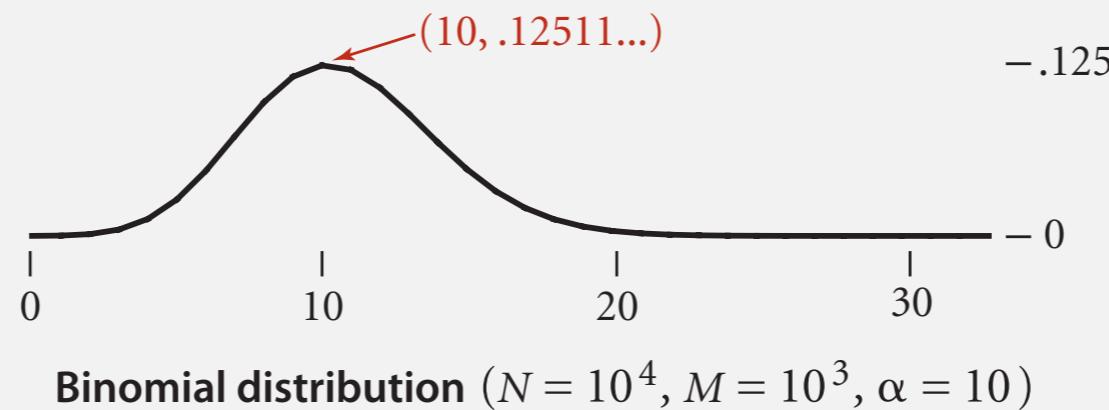
Pf sketch. Distribution of list size obeys a binomial distribution.



Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Consequence. Number of probes for search/insert is proportional to N / M .

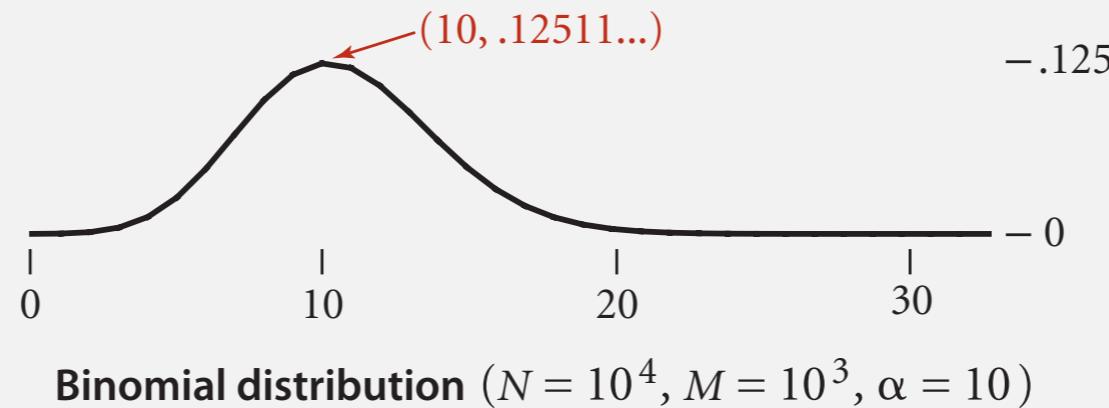
equals() and hashCode()

↑
M times faster than
sequential search

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Consequence. Number of probes for search/insert is proportional to N / M .

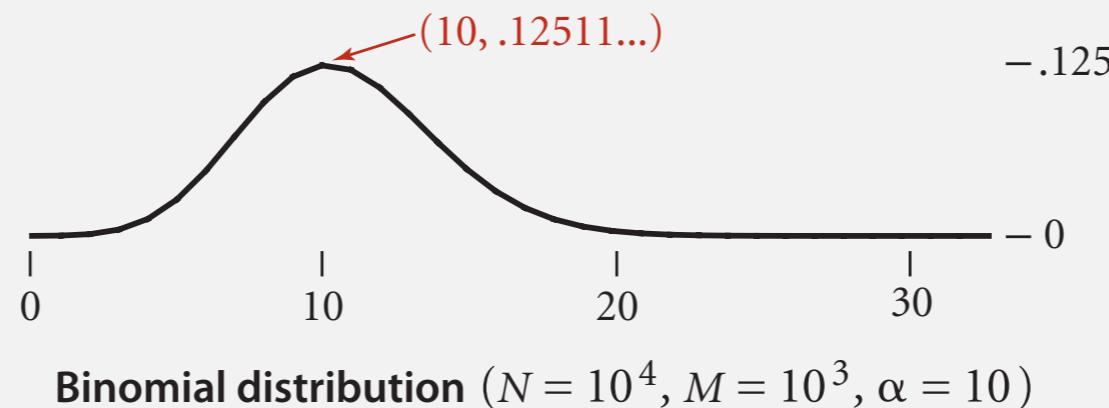
- M too large \Rightarrow too many empty chains.

\uparrow
M times faster than
sequential search

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Consequence. Number of probes for search/insert is proportional to N / M .

- M too large \Rightarrow too many empty chains.
- M too small \Rightarrow chains too long.

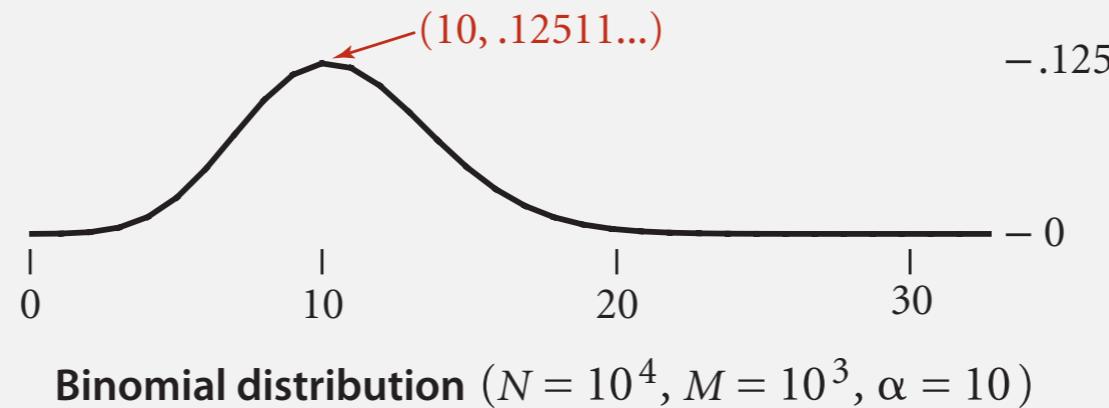
equals() and hashCode()

\uparrow
M times faster than
sequential search

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Consequence. Number of probes for search/insert is proportional to N / M .

- M too large \Rightarrow too many empty chains.
- M too small \Rightarrow chains too long.
- Typical choice: $M \sim N / 4 \Rightarrow$ constant-time ops.

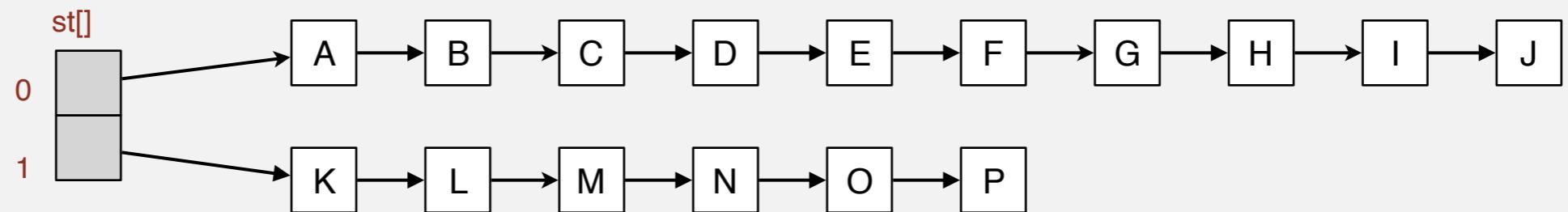
equals() and hashCode()

\uparrow
M times faster than
sequential search

Resizing in a separate-chaining hash table

Goal. Average length of list $N / M = \text{constant}$.

before resizing

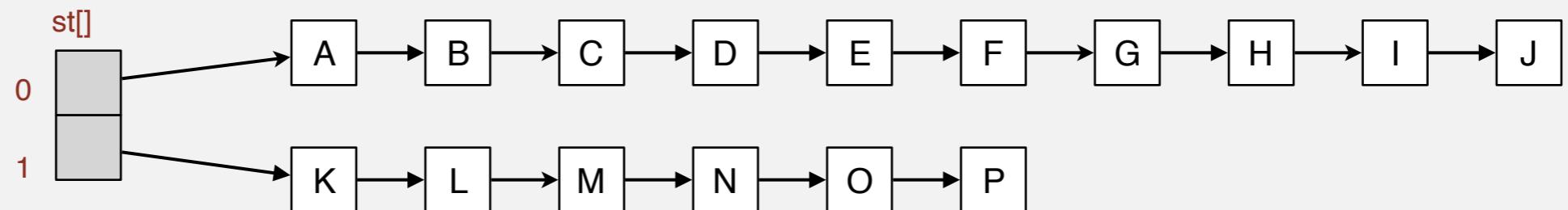


Resizing in a separate-chaining hash table

Goal. Average length of list $N / M = \text{constant}$.

- Double size of array M when $N / M \geq 8$.

before resizing

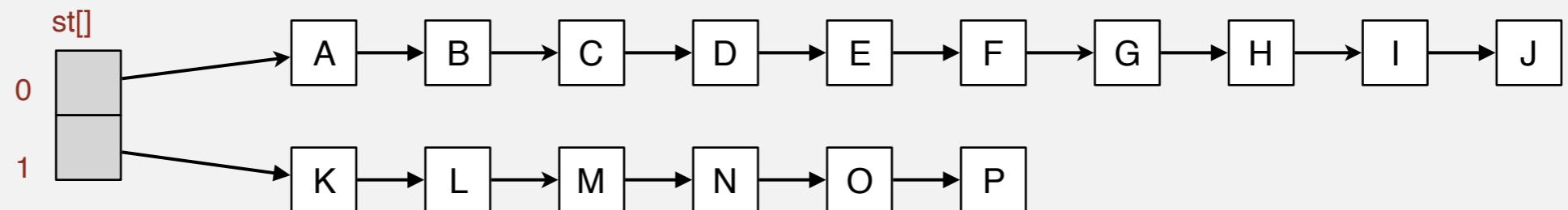


Resizing in a separate-chaining hash table

Goal. Average length of list $N / M = \text{constant}$.

- Double size of array M when $N / M \geq 8$.
- Halve size of array M when $N / M \leq 2$.

before resizing



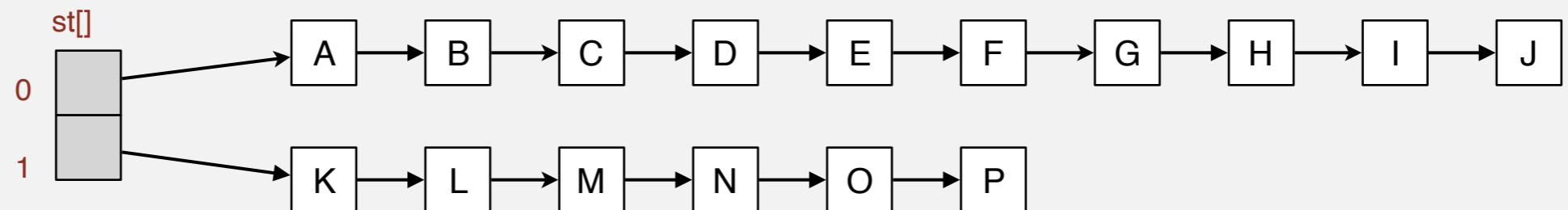
Resizing in a separate-chaining hash table

Goal. Average length of list $N / M = \text{constant}$.

- Double size of array M when $N / M \geq 8$.
- Halve size of array M when $N / M \leq 2$.
- Need to rehash all keys when resizing.

←
x.hashCode() does not change
but hash(x) can change

before resizing

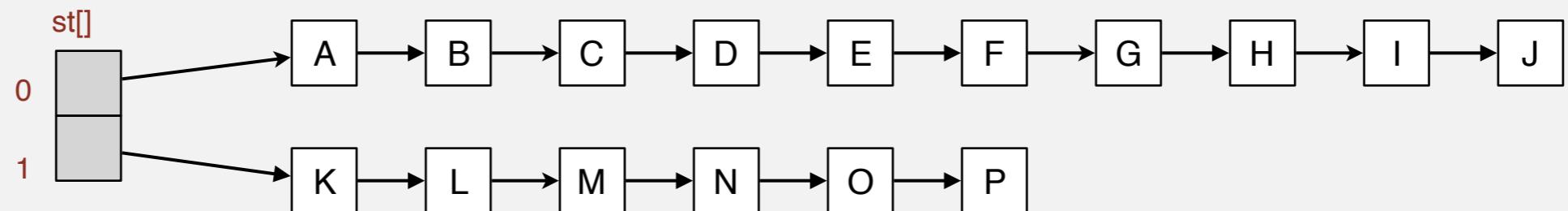


Resizing in a separate-chaining hash table

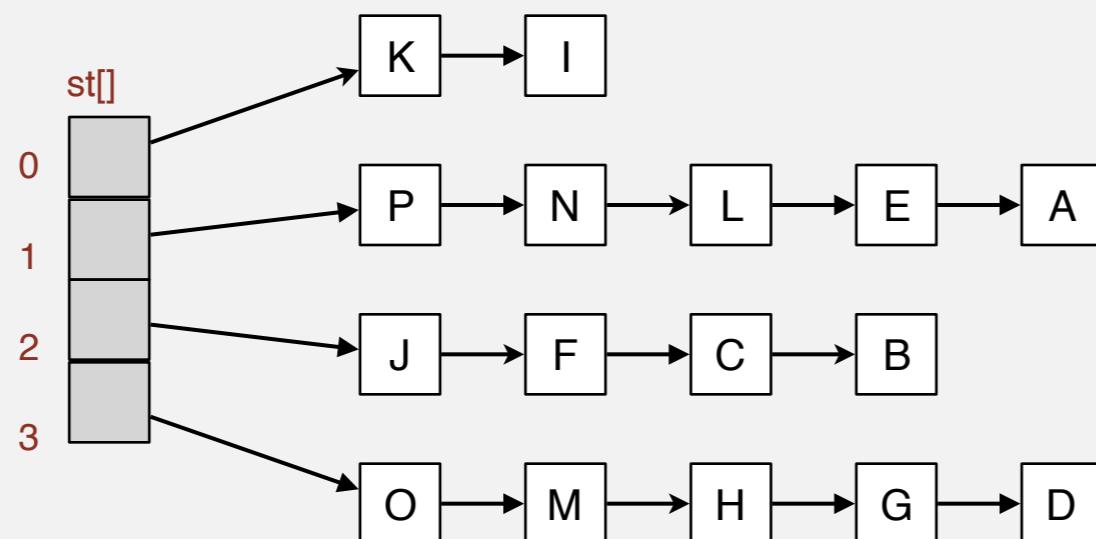
Goal. Average length of list $N / M = \text{constant}$.

- Double size of array M when $N / M \geq 8$.
- Halve size of array M when $N / M \leq 2$.
- Need to rehash all keys when resizing. ←
x.hashCode() does not change
but hash(x) can change

before resizing



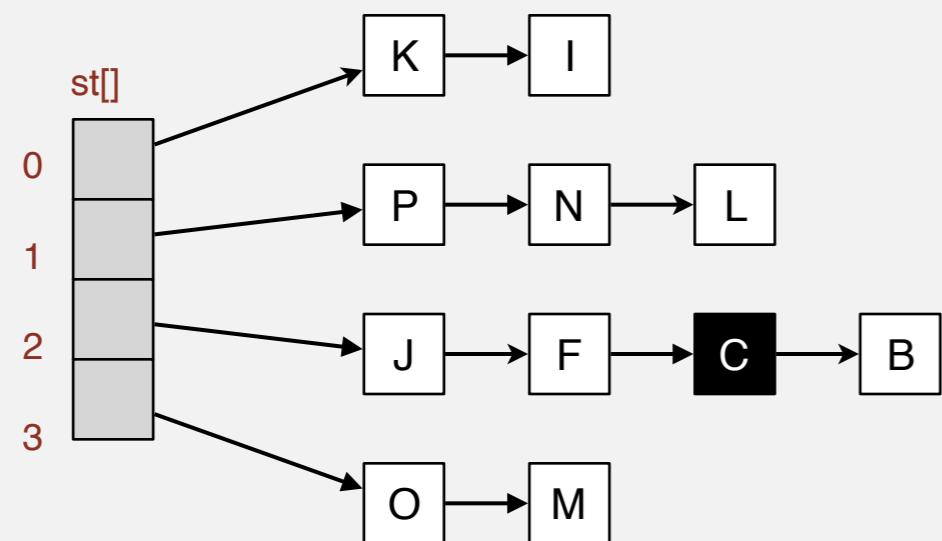
after resizing



Deletion in a separate-chaining hash table

Q. How to delete a key (and its associated value)?

before deleting C

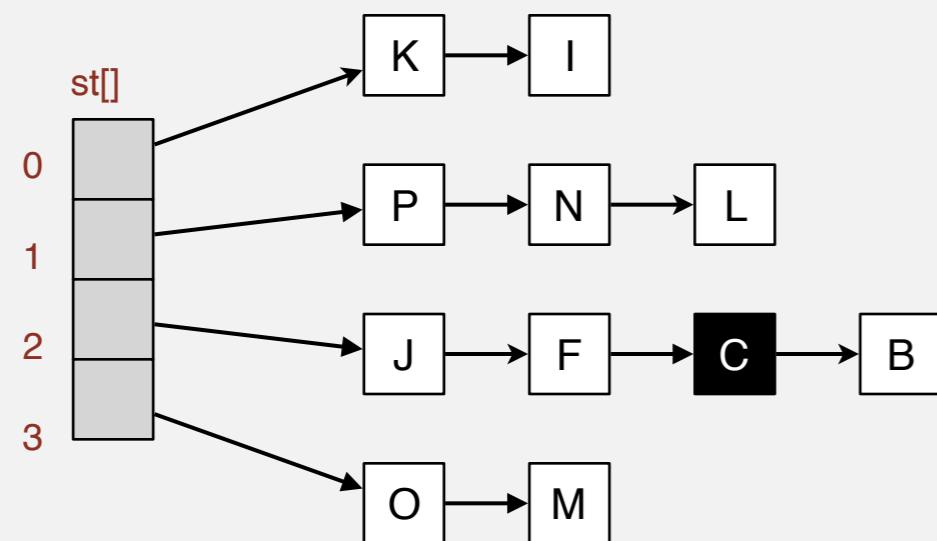


Deletion in a separate-chaining hash table

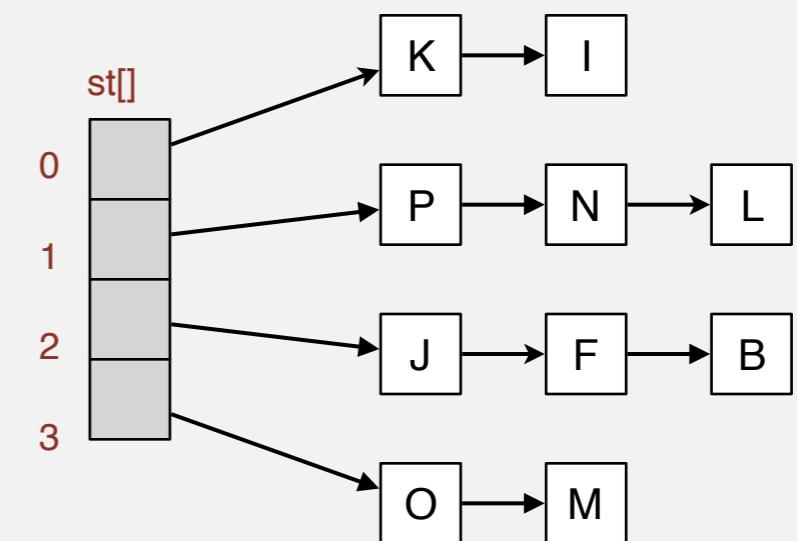
Q. How to delete a key (and its associated value)?

A. Easy: need only consider chain containing key.

before deleting C



after deleting C



Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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>
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		<code>equals()</code> <code>hashCode()</code>

* under uniform hashing assumption

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

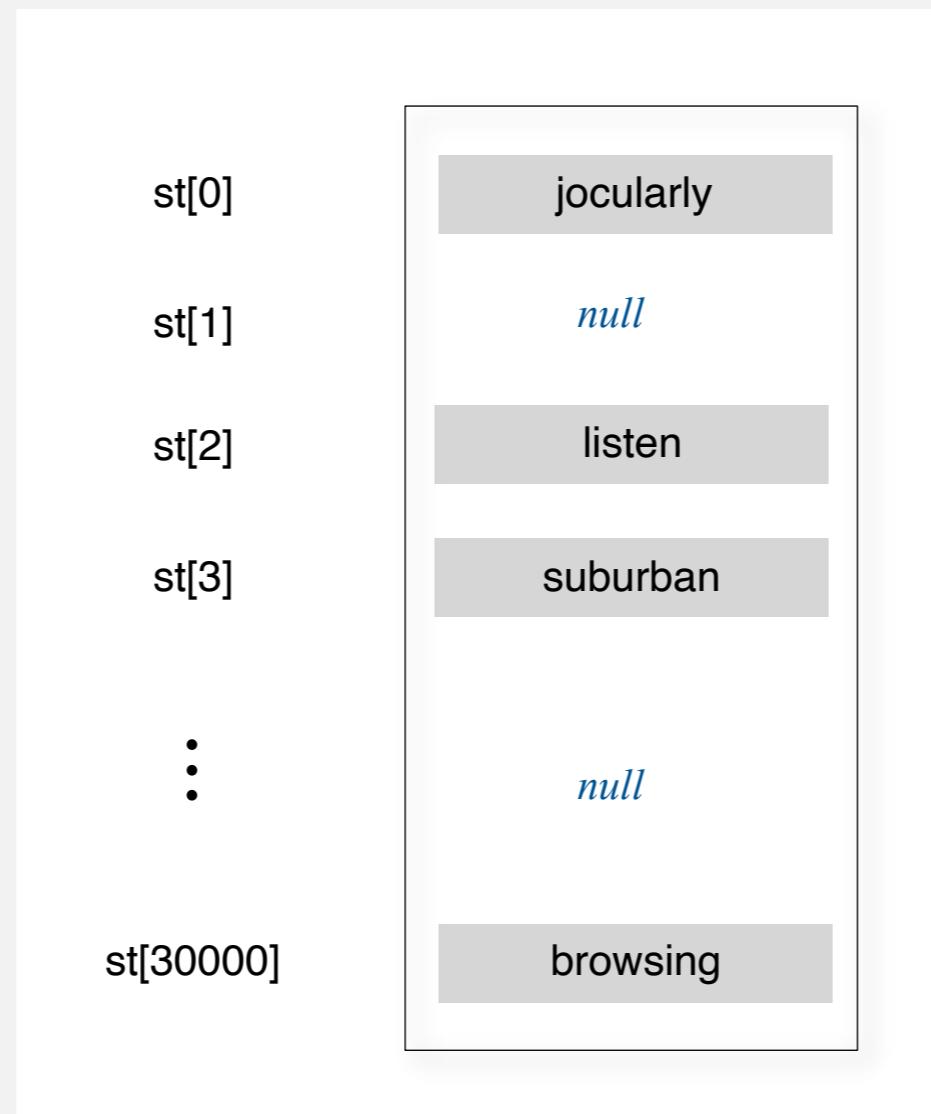
3.4 HASH TABLES

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ context

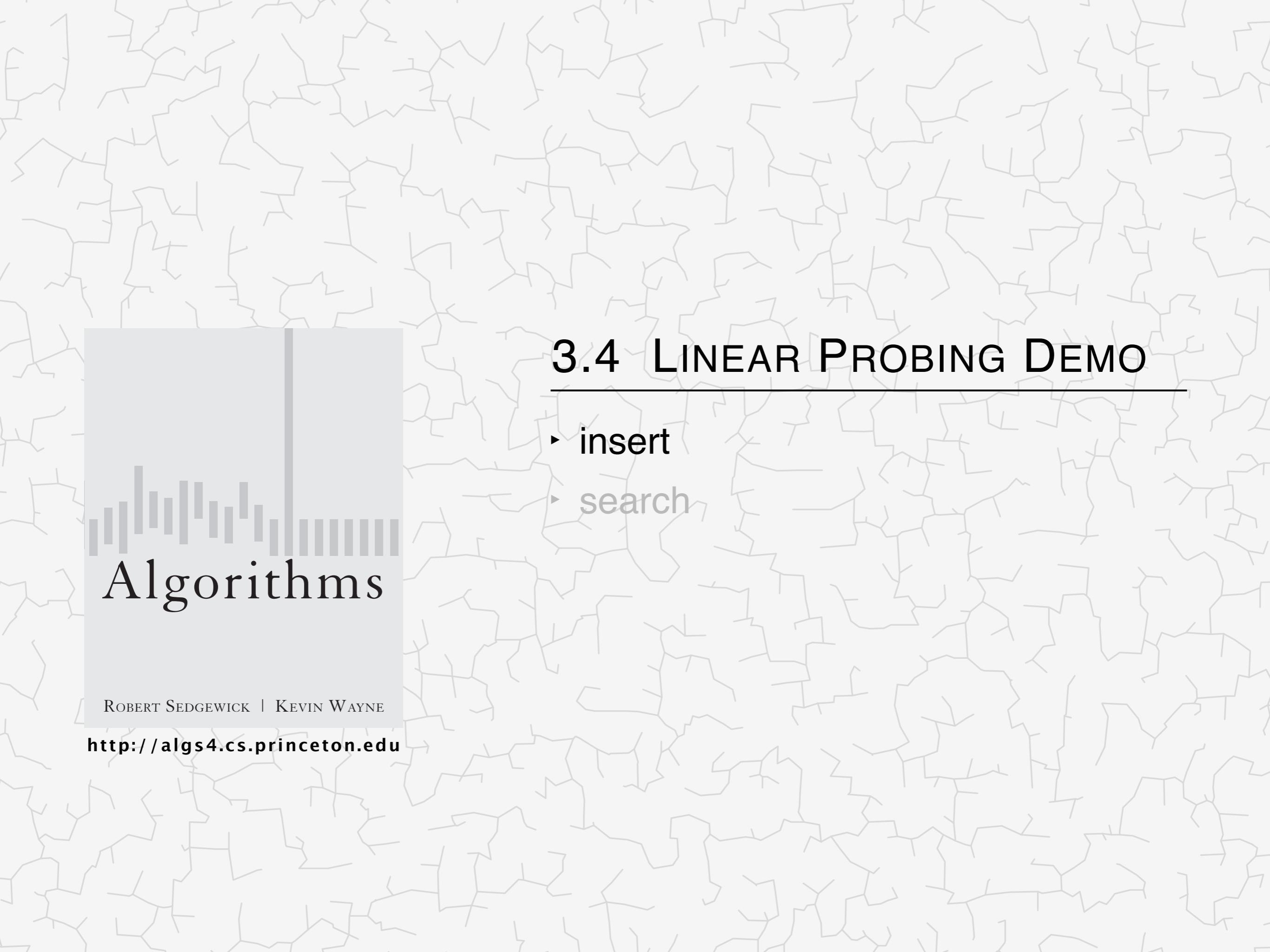
Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rochester-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



linear probing ($M = 30001$, $N = 15000$)



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.4 LINEAR PROBING DEMO

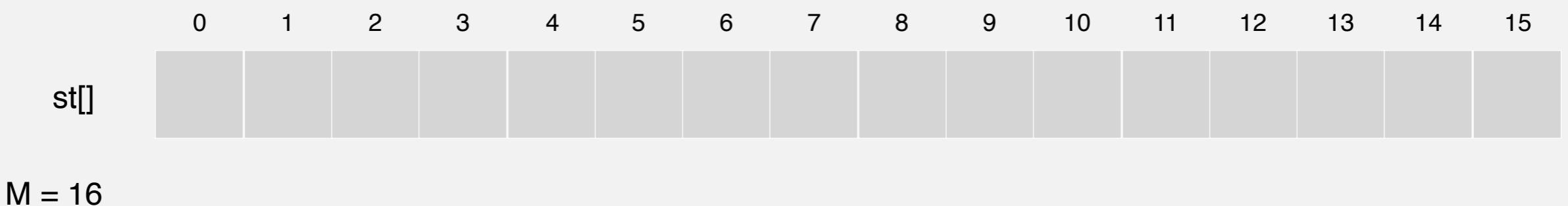
- ▶ insert
- ▶ search

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2$, etc.

linear-probing hash table



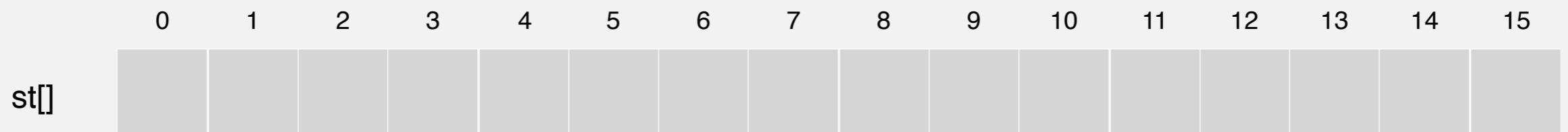
Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert

$\text{hash}(S) = 6$



$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert S

hash(S) = 6



$M = 16$

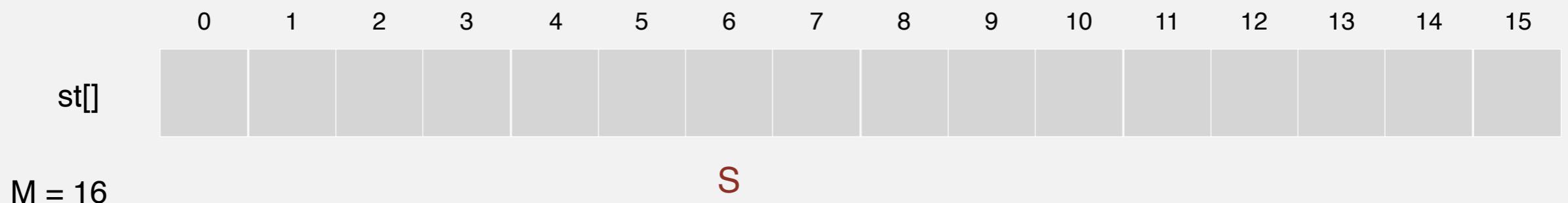
Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert S

hash(S) = 6



Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert S

hash(S) = 6



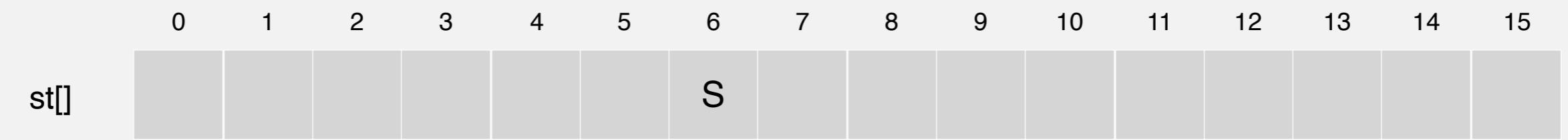
$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table



$M = 16$

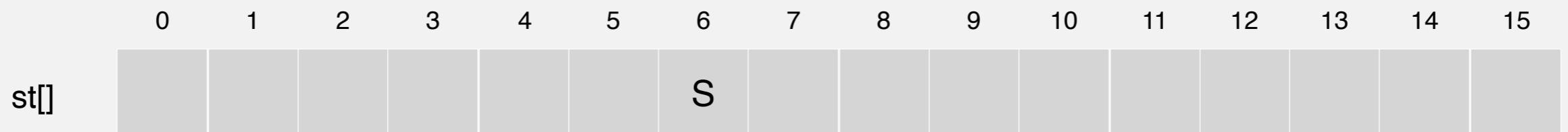
Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert E

$\text{hash}(E) = 10$



$M = 16$

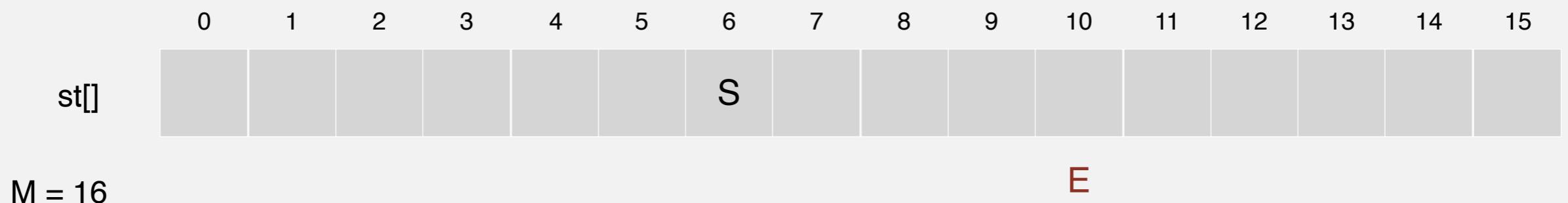
Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert E

$\text{hash}(E) = 10$



Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert E

$\text{hash}(E) = 10$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]						S					E					

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]						S				E						

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert A

hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]						S				E						

$M = 16$

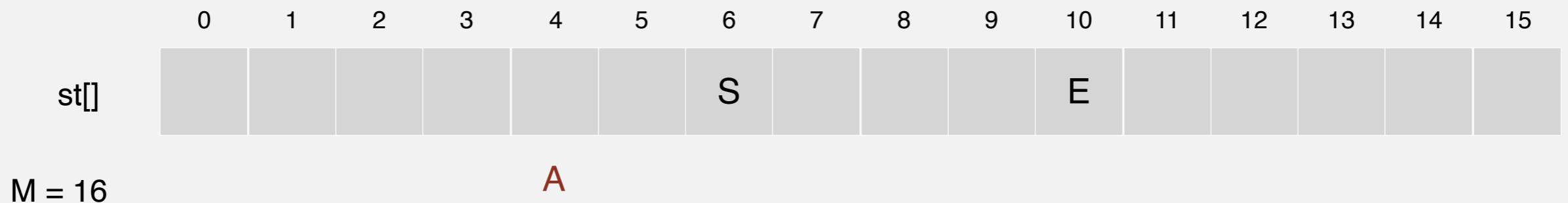
Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert A

hash(A) = 4



Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert A

hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	S				E						

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert R

hash(R) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

$M = 16$

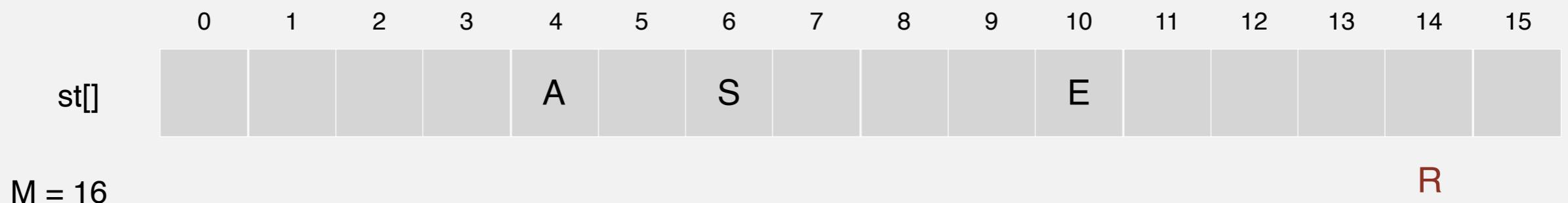
Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2$, etc.

insert R

hash(R) = 14



Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert R

hash(R) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	S				E				R		

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	S				E				R		

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert C

hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	S				E				R		

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert C

$$\text{hash}(C) = 5$$

The diagram shows a horizontal array of 16 cells, indexed from 0 to 15 above the cells. The cells contain the following values:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[0]				A	S			E					R		

Below the array, the value $M = 16$ is written in large red letters, indicating the total size of the array.

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert C

hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E			R		

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert H

$$\text{hash}(H) = 4$$

Diagram illustrating a 16-element array `st[]` with indices ranging from 0 to 15. The elements are represented by gray boxes. Specific elements are labeled:

- `st[4]`: A
- `st[5]`: C
- `st[6]`: S
- `st[9]`: E
- `st[14]`: R
- `st[11]`: H (highlighted in red)

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2$, etc.

insert H

$$\text{hash}(H) = 4$$

Diagram illustrating a character array `st[]` of size $M = 16$. The array elements are indexed from 0 to 15. The characters stored in the array are:

Index	Character
0	
1	
2	
3	
4	A
5	C
6	S
7	
8	
9	E
10	
11	H
12	
13	
14	R
15	

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert H

$$\text{hash}(H) = 4$$

The diagram shows a horizontal array of 16 cells, indexed from 0 to 15 above the array. The cells contain the following values:

Index	Value
0	
1	
2	
3	
4	A
5	C
6	S
7	H
8	
9	E
10	
11	
12	
13	
14	R
15	

Below the array, the label $M = 16$ is written in red.

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
M = 16												H				

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert X

hash(X) = 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2$, etc.

insert X

hash(X) = 15

Diagram illustrating a 16-element array `st[]` with indices ranging from 0 to 15. The array elements are as follows:

Index	Element Value
0	X
1	X
2	X
3	X
4	A
5	C
6	S
7	H
8	X
9	X
10	E
11	X
12	X
13	X
14	R
15	X

The label `M = 16` is located at the bottom right.

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert X

$\text{hash}(X) = 15$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	X

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E			R	X	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert M

hash(M) = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E			R	X	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert M

$$\text{hash}(M) = 1$$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

				A	C	S	H		E				R	X
--	--	--	--	---	---	---	---	--	---	--	--	--	---	---

st[]

M = 16 **M**

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert M

hash(M) = 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H		E			R	X	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E			R	X	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M		A	C	S	H		E				R	X	

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M		A	C	S	H		E				R	X	P

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M		A	C	S	H		E				R	X	P

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M		A	C	S	H		E				R	X	

M = 16 P

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H		E				R	X

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2$, etc.

insert L

$$\text{hash}(L) = 6$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H		E				R	X	

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
M = 16									L							

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
M = 16									L							

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.4 LINEAR PROBING DEMO

- ▶ insert
- ▶ search

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search E

$$\text{hash}(E) = 10$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1, i+2, \dots$

search E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16												E				

search hit
(return corresponding value)

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

L

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16												L				

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16												L				

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16									L							

search hit
(return corresponding value)

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16									K							

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

$$\text{hash}(K) = 5$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E			R	X	

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16												K				

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16												K				

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16												K				

Linear-probing hash table demo: search

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16											K					

search miss
(return null)

Linear-probing hash table summary

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2, \dots$

Search. Search table index i ; if occupied but no match, try $i+1, i+2, \dots$

Note. Array size M **must be** greater than number of key-value pairs N .

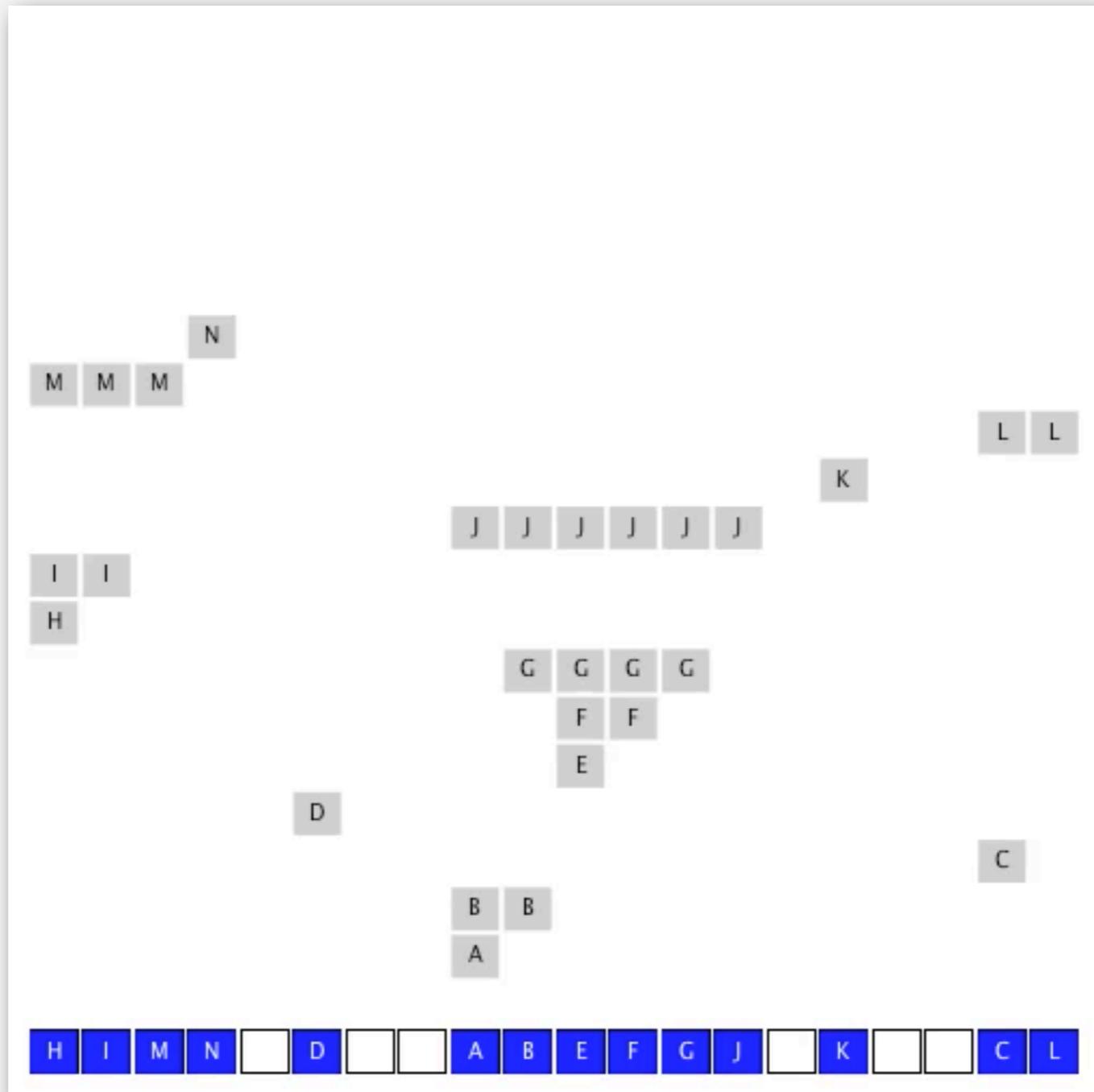
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Clustering

Cluster. A contiguous block of items.

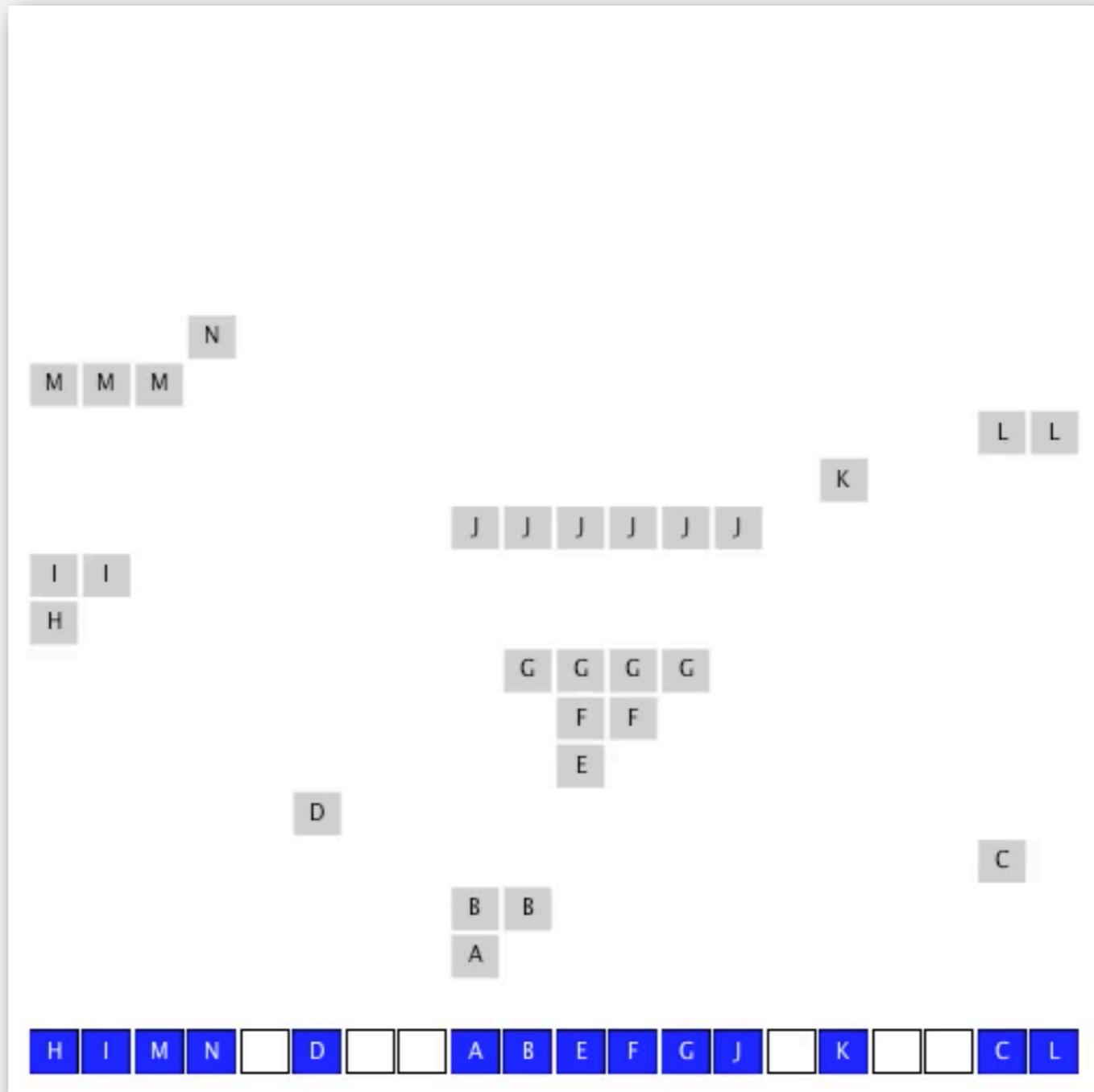
Observation. New keys likely to hash into middle of big clusters.



Clustering

Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of big clusters.

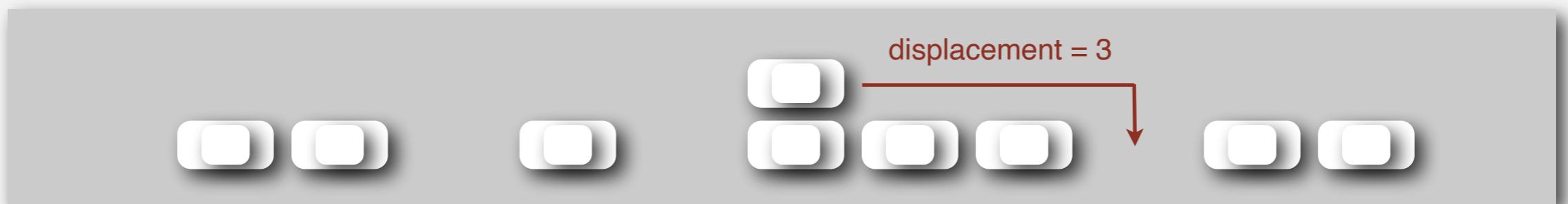


Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i + 1, i + 2$, etc.

Q. What is mean displacement of a car?

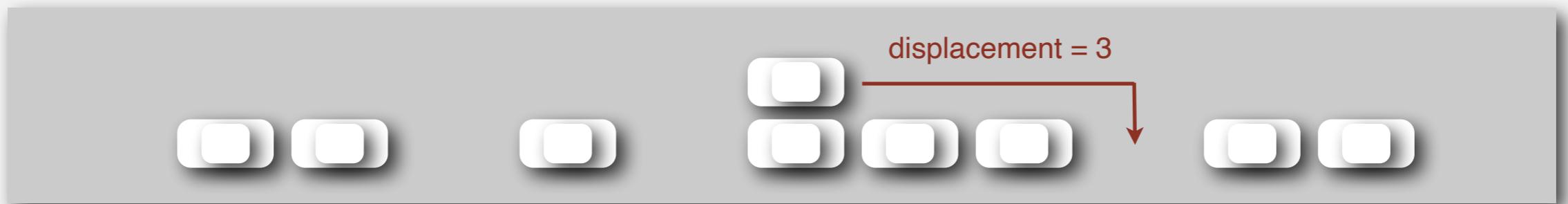


Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i + 1, i + 2$, etc.

Q. What is mean displacement of a car?



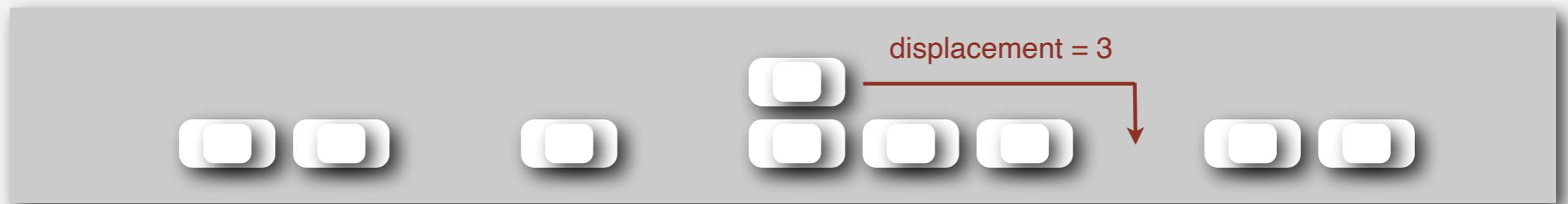
Half-full. With $M / 2$ cars, mean displacement is $\sim 3 / 2$.

Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i + 1, i + 2$, etc.

Q. What is mean displacement of a car?



Half-full. With $M / 2$ cars, mean displacement is $\sim 3 / 2$.

Full. With M cars, mean displacement is $\sim \pi\sqrt{M / 8}$.

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

search hit

$$\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

search miss / insert

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

search hit

$$\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

search miss / insert

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

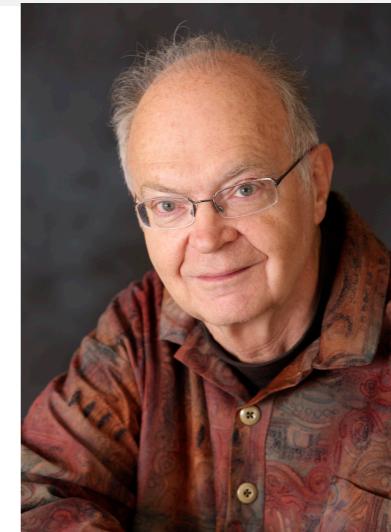
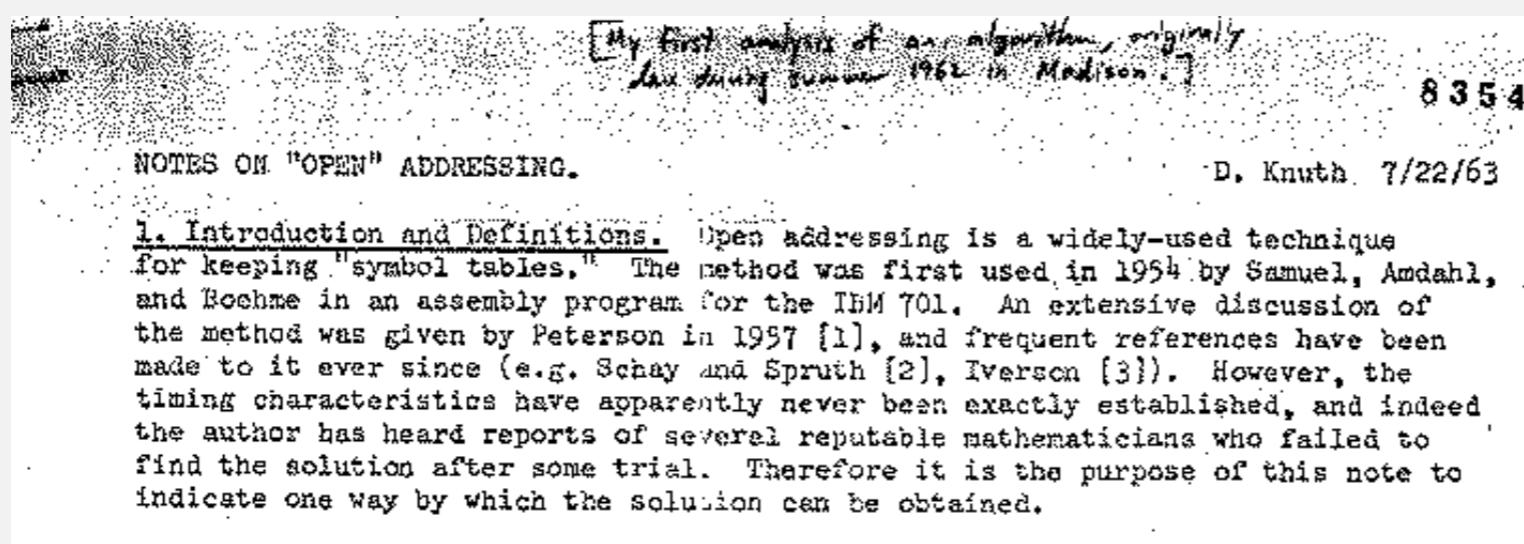
$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

search hit

$$\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

search miss / insert

Pf.



Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

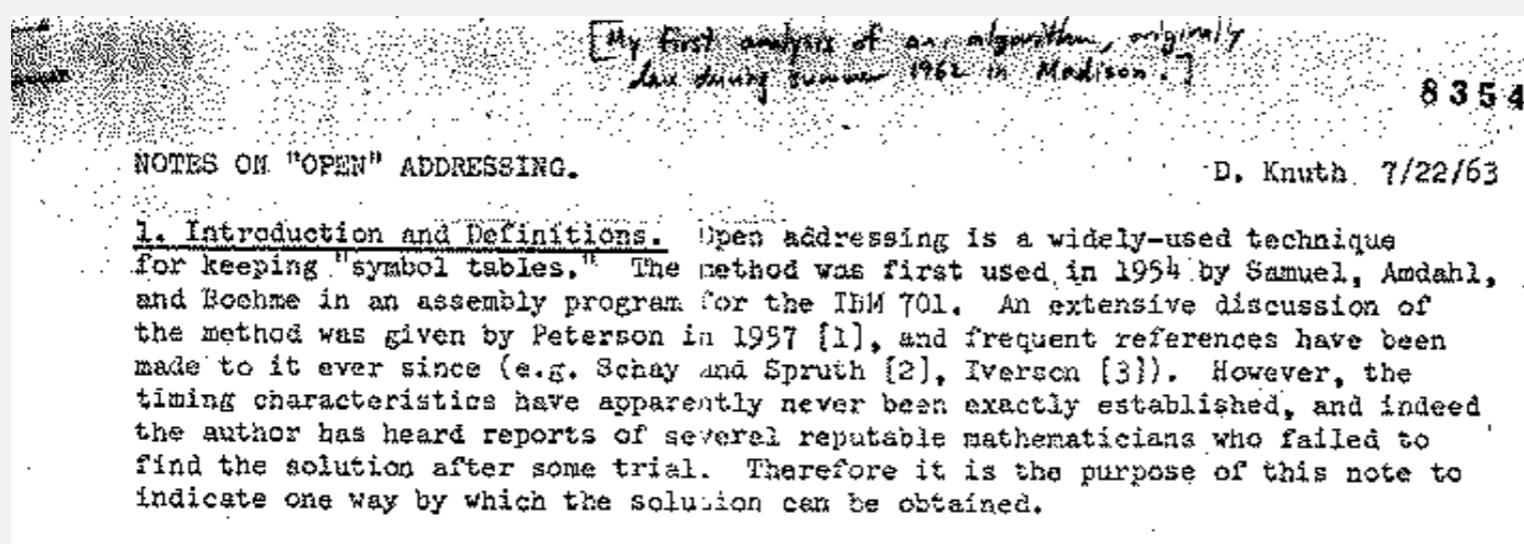
$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

search hit

$$\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

search miss / insert

Pf.



Parameters.

- M too large \Rightarrow too many empty array entries.
- M too small \Rightarrow search time blows up.
- Typical choice: $\alpha = N / M \sim 1/2$. \leftarrow # probes for search hit is about 3/2
probes for search miss is about 5/2

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq \frac{1}{2}$.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq \frac{1}{2}$.

- Double size of array M when $N / M \geq \frac{1}{2}$.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq \frac{1}{2}$.

- Double size of array M when $N / M \geq \frac{1}{2}$.
- Halve size of array M when $N / M \leq \frac{1}{8}$.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq \frac{1}{2}$.

- Double size of array M when $N / M \geq \frac{1}{2}$.
- Halve size of array M when $N / M \leq \frac{1}{8}$.
- Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq \frac{1}{2}$.

- Double size of array M when $N / M \geq \frac{1}{2}$.
- Halve size of array M when $N / M \leq \frac{1}{8}$.
- Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]					A		S			E				R		
vals[]					2		0			1				3		

Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

A. Requires some care: can't just delete array entries.

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

doesn't work, e.g., if $\text{hash}(H) = 4$

after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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>
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		<code>equals()</code> <code>hashCode()</code>
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		<code>equals()</code> <code>hashCode()</code>

* under uniform hashing assumption

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.4 HASH TABLES

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ context

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?



War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.



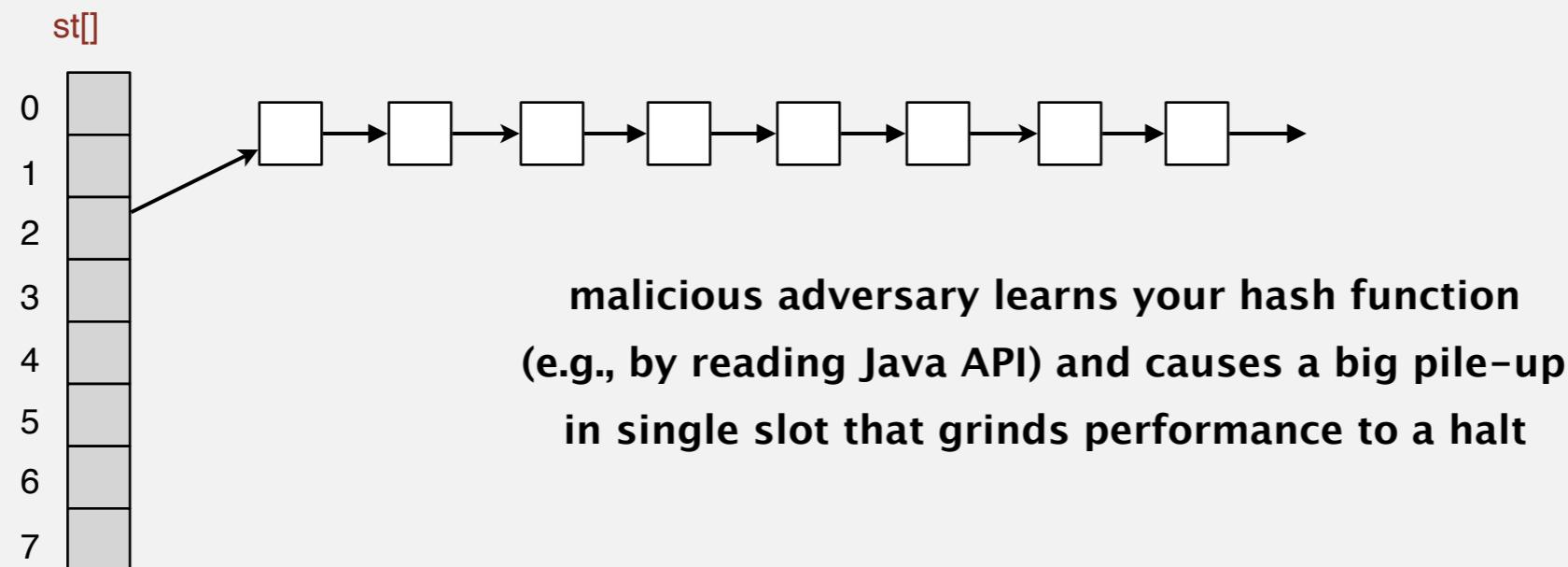
War story: algorithmic complexity attacks

- Q. Is the uniform hashing assumption important in practice?
- A. Obvious situations: aircraft control, nuclear reactor, pacemaker.
- A. Surprising situations: **denial-of-service** attacks.



War story: algorithmic complexity attacks

- Q. Is the uniform hashing assumption important in practice?
- A. Obvious situations: aircraft control, nuclear reactor, pacemaker.
- A. Surprising situations: denial-of-service attacks.



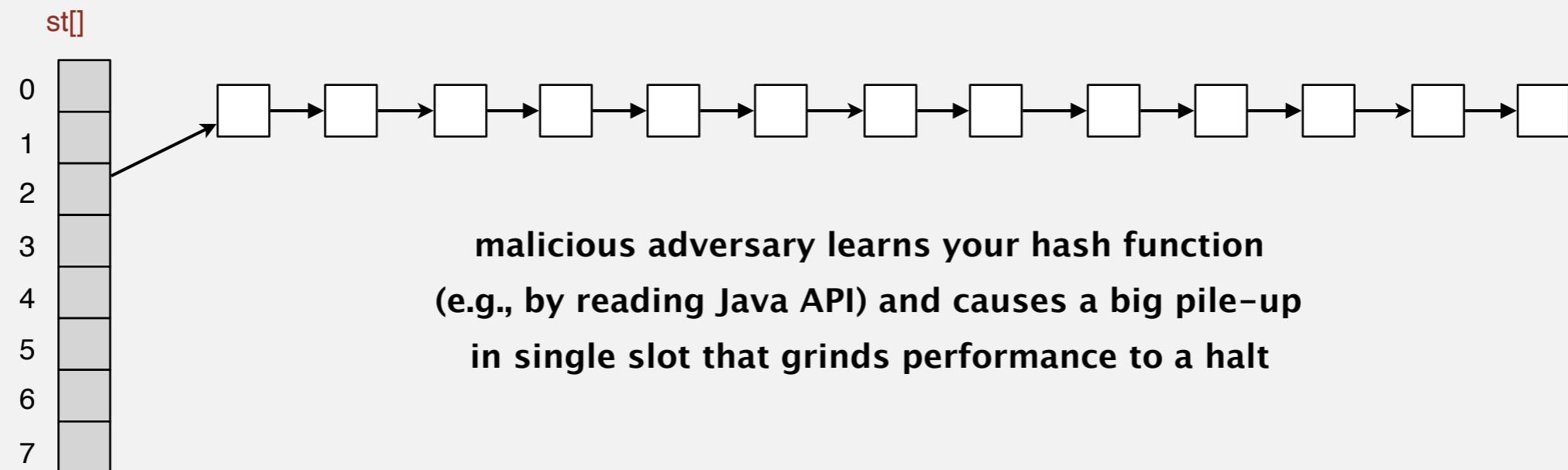
Real-world exploits. [Crosby-Wallach 2003]

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: **denial-of-service** attacks.



Real-world exploits. [Crosby-Wallach 2003]

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Diversion: one-way hash functions

One-way hash function. "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160,

known to be insecure

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Diversion: one-way hash functions

One-way hash function. "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160,

known to be insecure

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Applications. Digital fingerprint, message digest, storing passwords.

Diversion: one-way hash functions

One-way hash function. "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160,

known to be insecure

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

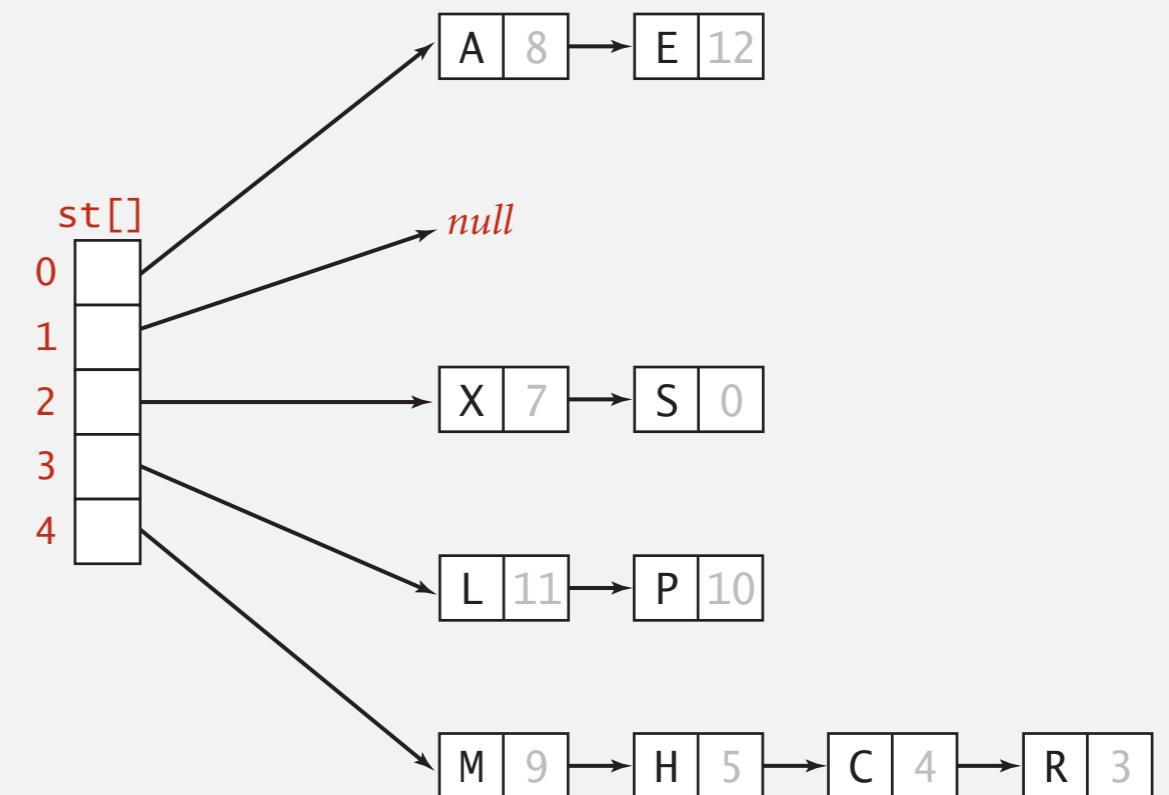
Applications. Digital fingerprint, message digest, storing passwords.

Caveat. Too expensive for use in ST implementations.

Separate chaining vs. linear probing

Separate chaining.

- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.



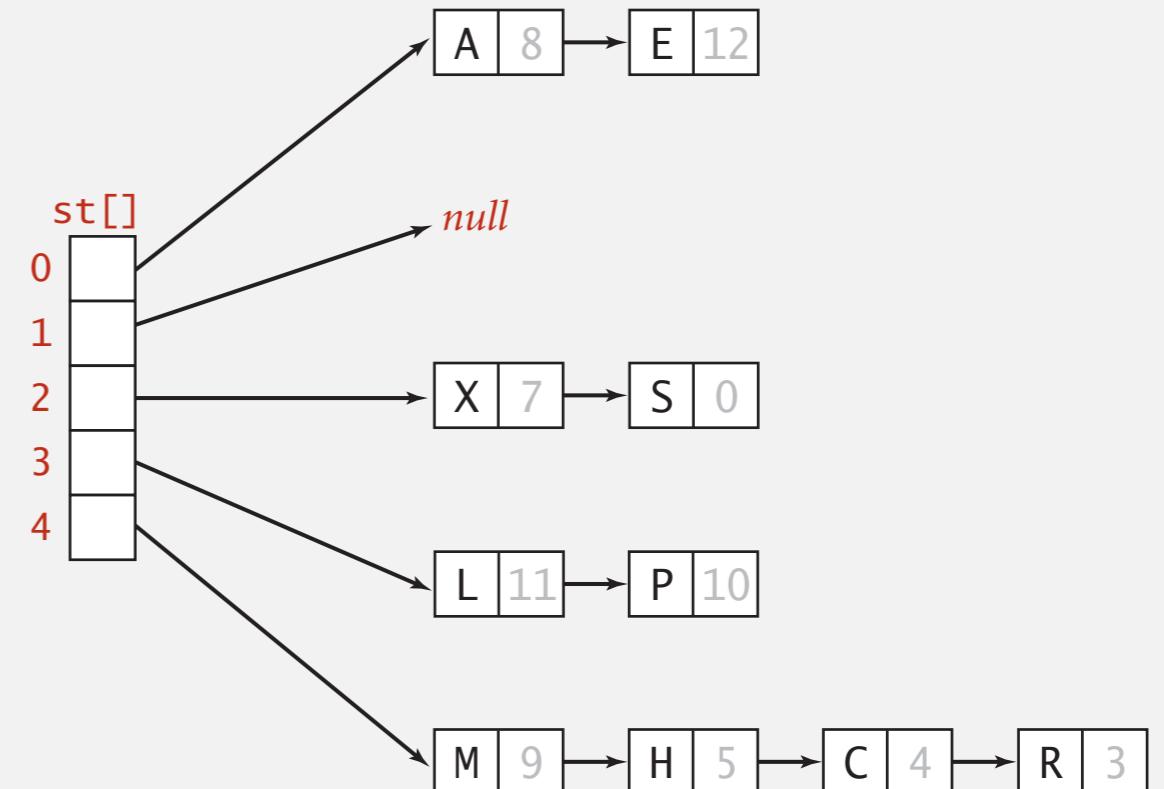
Separate chaining vs. linear probing

Separate chaining.

- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.



keys[]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
vals[]	P	M			A	C	S	H	L		E			R	X	
	10	9			8	4	0	5	11		12			3	7	

Hashing: variations on the theme

Many improved versions have been studied.

Hashing: variations on the theme

Many improved versions have been studied.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

Double hashing. [linear-probing variant]

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

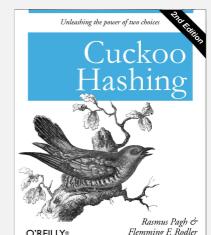
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

Double hashing. [linear-probing variant]

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst-case time for search.



Hash tables vs. balanced search trees

Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.