BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

Dept. of Computer Engineering

Priority Queues and Heapsort

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort

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.

operation	*argument*	*return value*
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

Priority queue API

Requirement. Generic items are `Comparable`.

Key must be Comparable (bounded type parameter)

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

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]
- Computational number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]
- 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 (N huge, M large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

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
Turing      10/12/1993   3532.36
Hoare        2/10/2005   4050.20
```

```
% 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 (N huge, M large).

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);
   if (pq.size() > M)
      pq.delMin();
}
```

pq contains largest M items

Transaction data type is Comparable (ordered by $$)

order of growth of finding the largest M 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

Priority Queues and Heapsort

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort

Priority queue: unordered and ordered array implementation

operation	*argument*	*return value*	*size*	*contents (unordered)*							*contents (ordered)*						
insert	P		1	P							P						
insert	Q		2	P	Q						P	Q					
insert	E		3	P	Q	E					E	P	Q				
remove max		Q	2	P	E						E	P					
insert	X		3	P	E	X					E	P	X				
insert	A		4	P	E	X	A				A	E	P	X			
insert	M		5	P	E	X	A	M			A	E	M	P	X		
remove max		X	4	P	E	M	A				A	E	M	P			
insert	P		5	P	E	M	A	P			A	E	M	P	P		
insert	L		6	P	E	M	A	P	L		A	E	L	M	P	P	
insert	E		7	P	E	M	A	P	L	E	A	E	E	L	M	P	P
remove max		P	6	E	M	A	P	L	E		A	E	E	L	M	P	

A sequence of operations on a priority queue

Priority queue: unordered array implementation

```
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;    // pq[i] = ith element on pq
   private int N;       // number of elements on pq

   public UnorderedMaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity];  }

   public boolean isEmpty()
   {  return N == 0; }

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

   public Key delMax()
   {
      int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}
```

no generic array creation

`less()` and `exch()` similar to sorting methods

null out entry to prevent loitering

Priority queue elementary implementations

Challenge. Implement all operations efficiently.

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

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

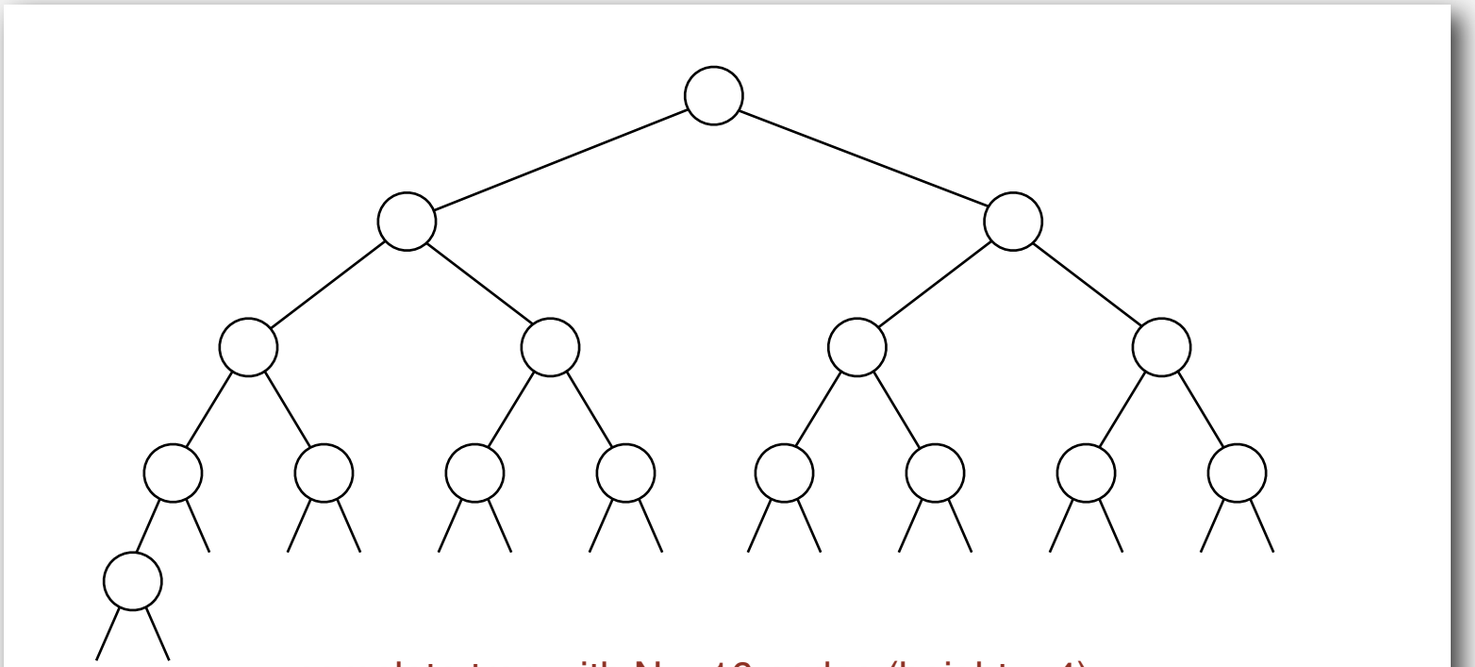
Priority Queues and Heapsort

- **Heapsort**
- API
- Elementary implementations
- **Binary heaps**
- Heapsort

Binary tree

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

Complete tree. Perfectly balanced, except for bottom level.



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

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

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

A complete binary tree in nature

Binary heap representations

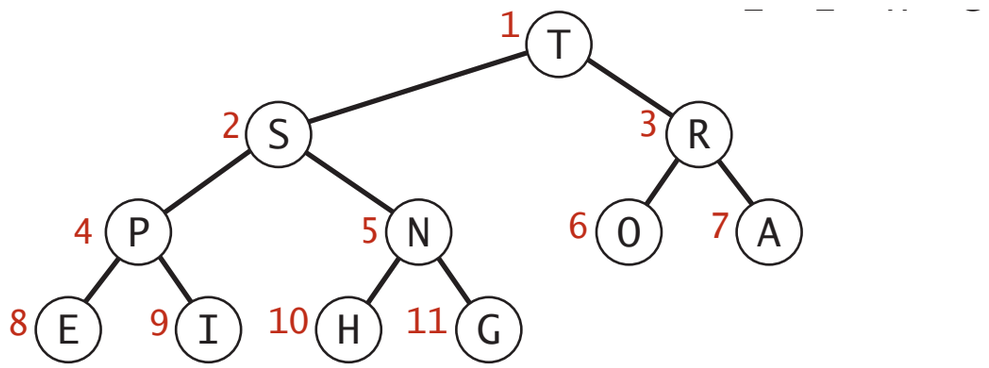
Binary heap. Array representation of a heap-ordered complete binary tree.

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!



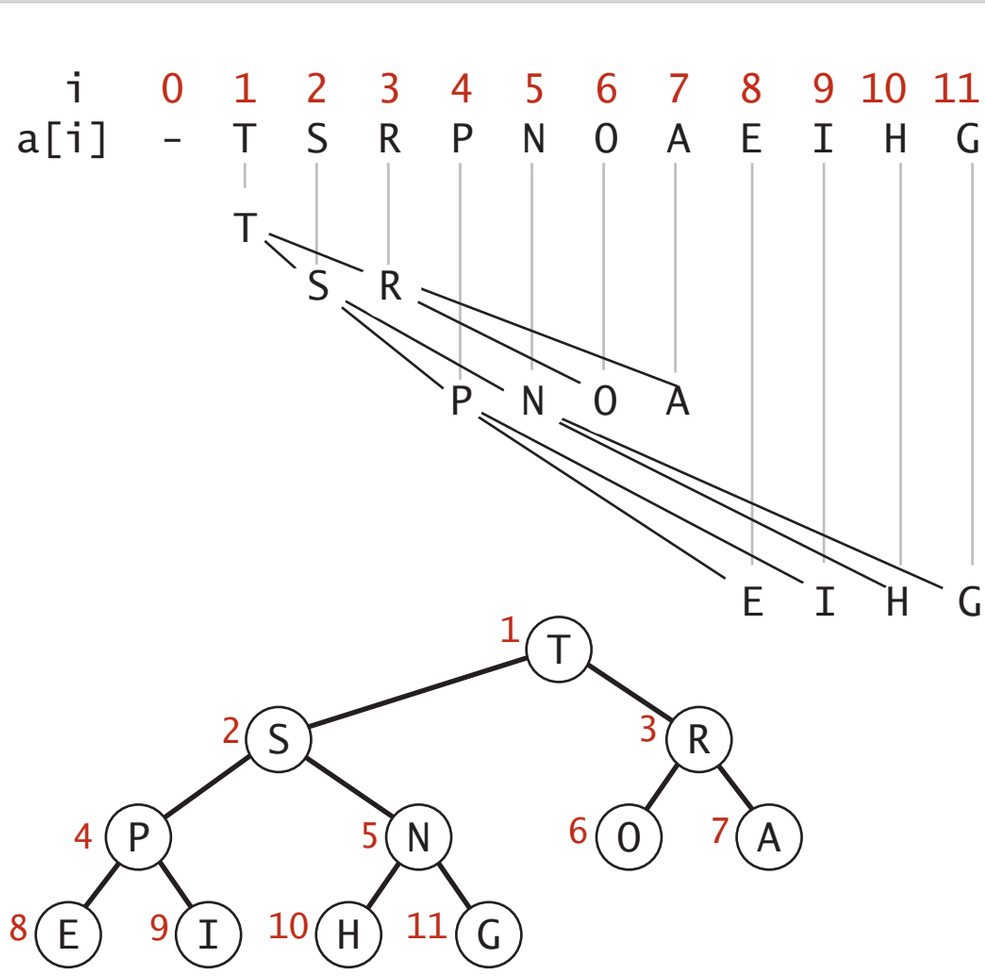
Heap representations

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`.



Heap representations

Promotion in a heap

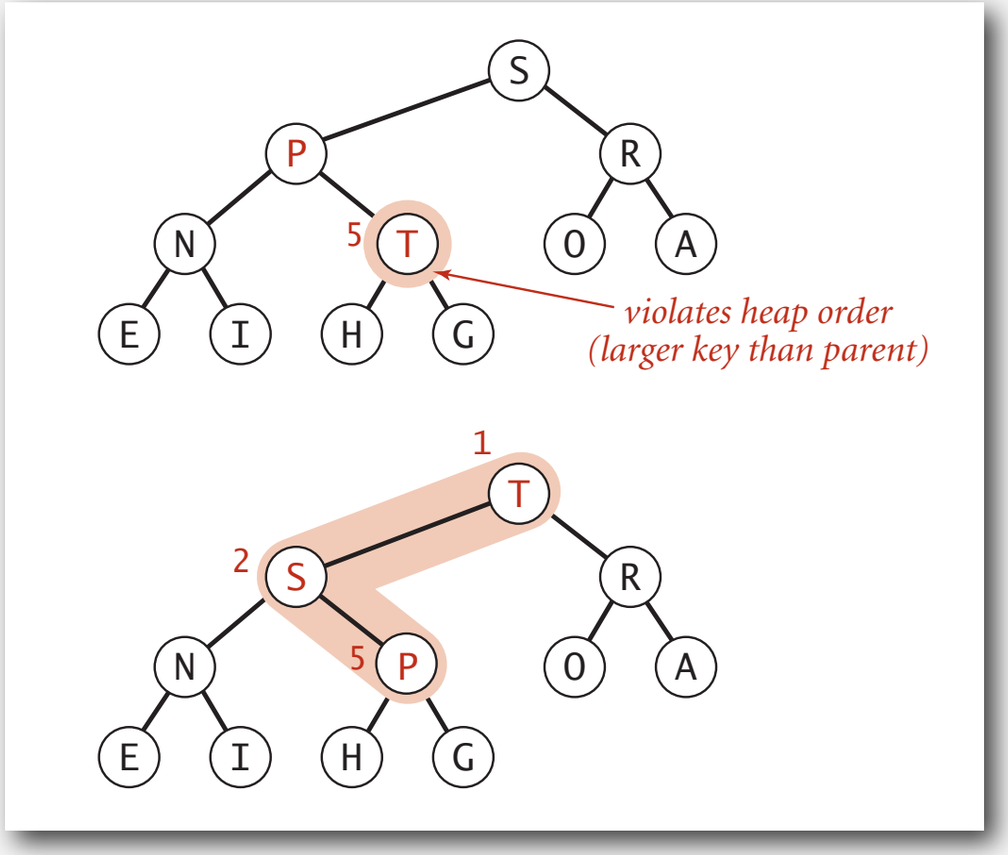
Scenario. Child's key becomes larger key than its parent's key.

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



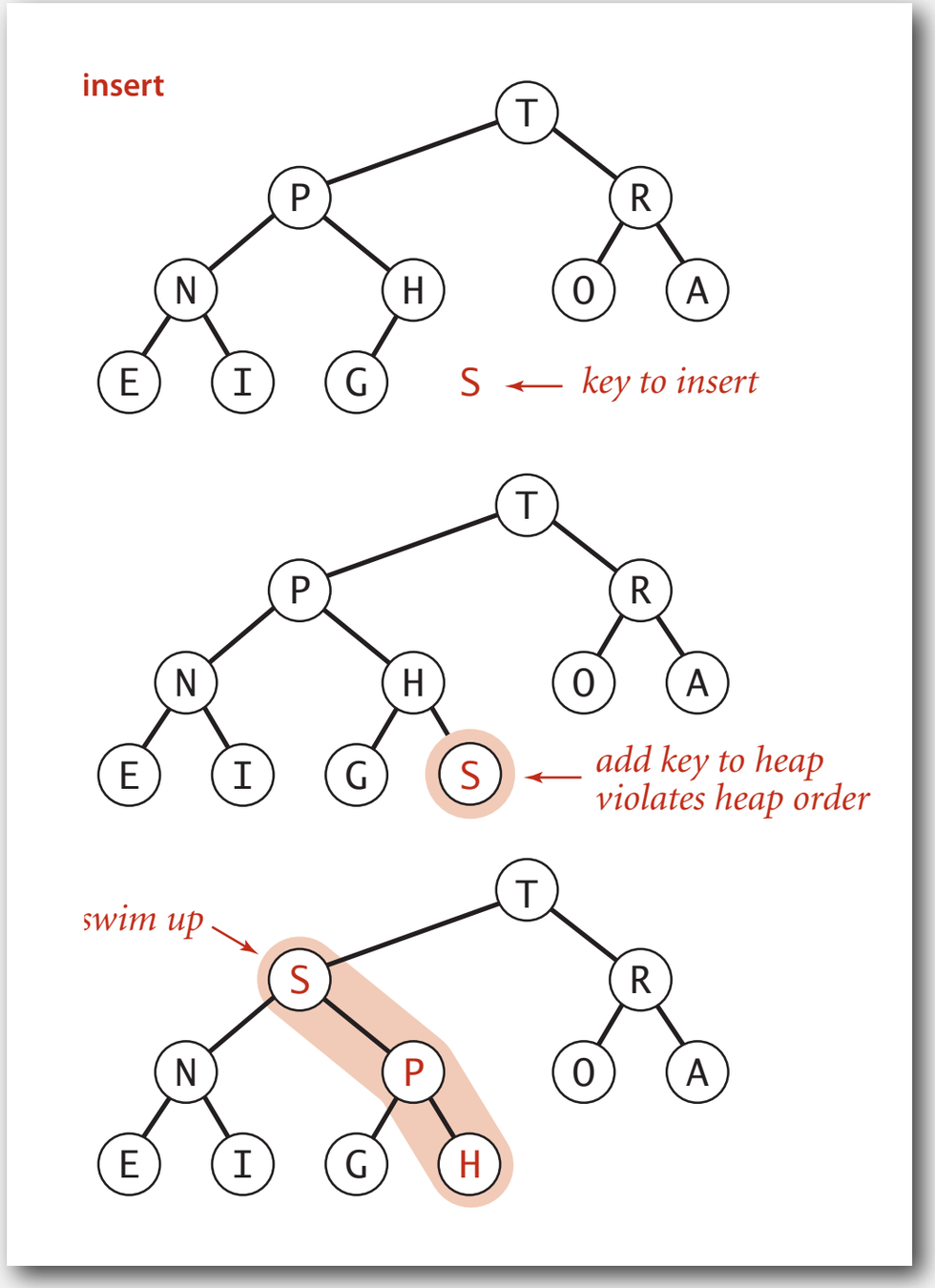
Peter principle. Node promoted to level of incompetence.

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 keys.

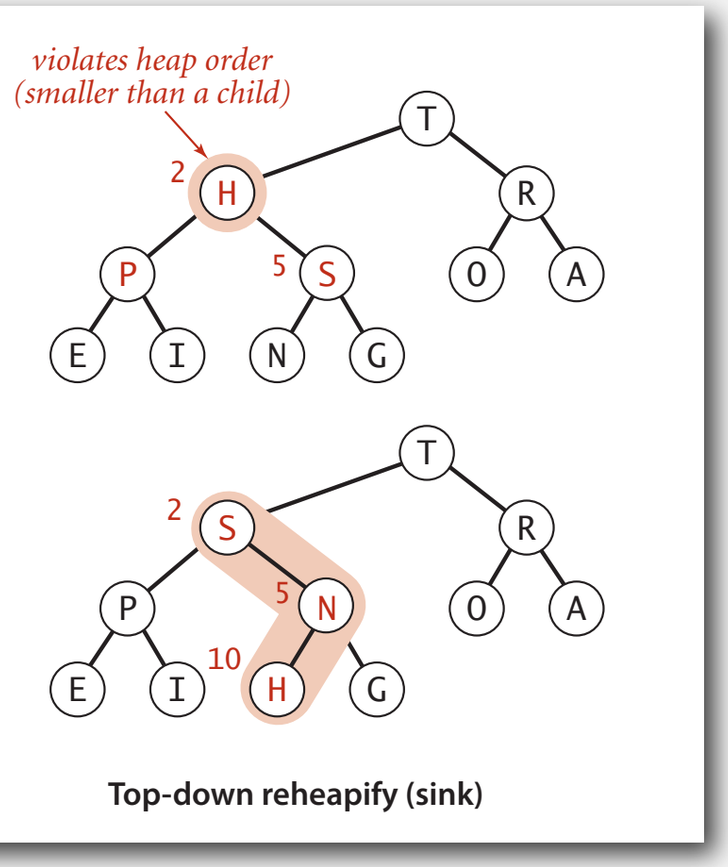
To eliminate the violation:

why not smaller child?

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

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

children of node at k are 2k and 2k+1



Top-down reheapify (sink)

Power struggle. Better subordinate promoted.

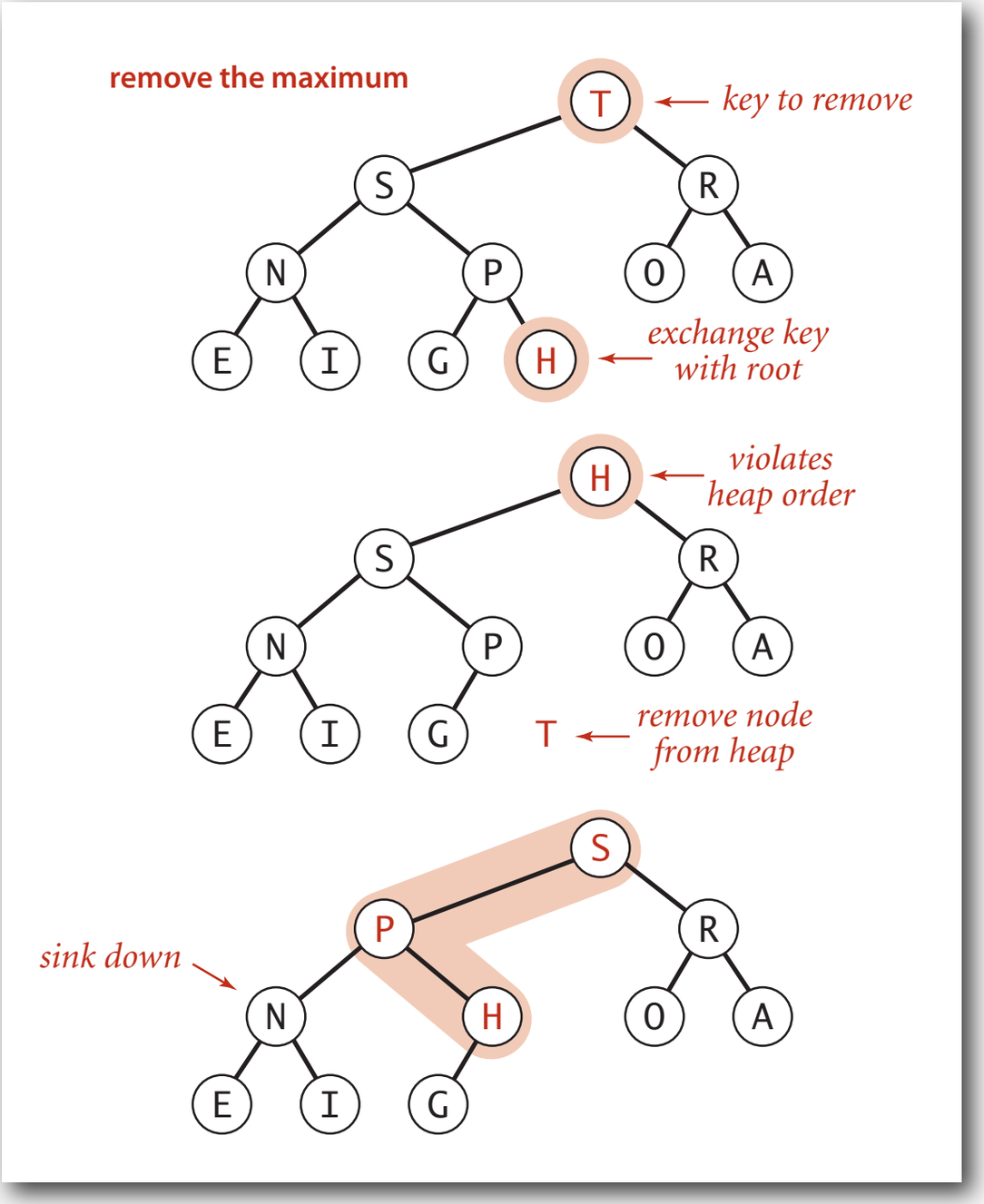
Delete the maximum in a heap

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

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

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

prevent loitering

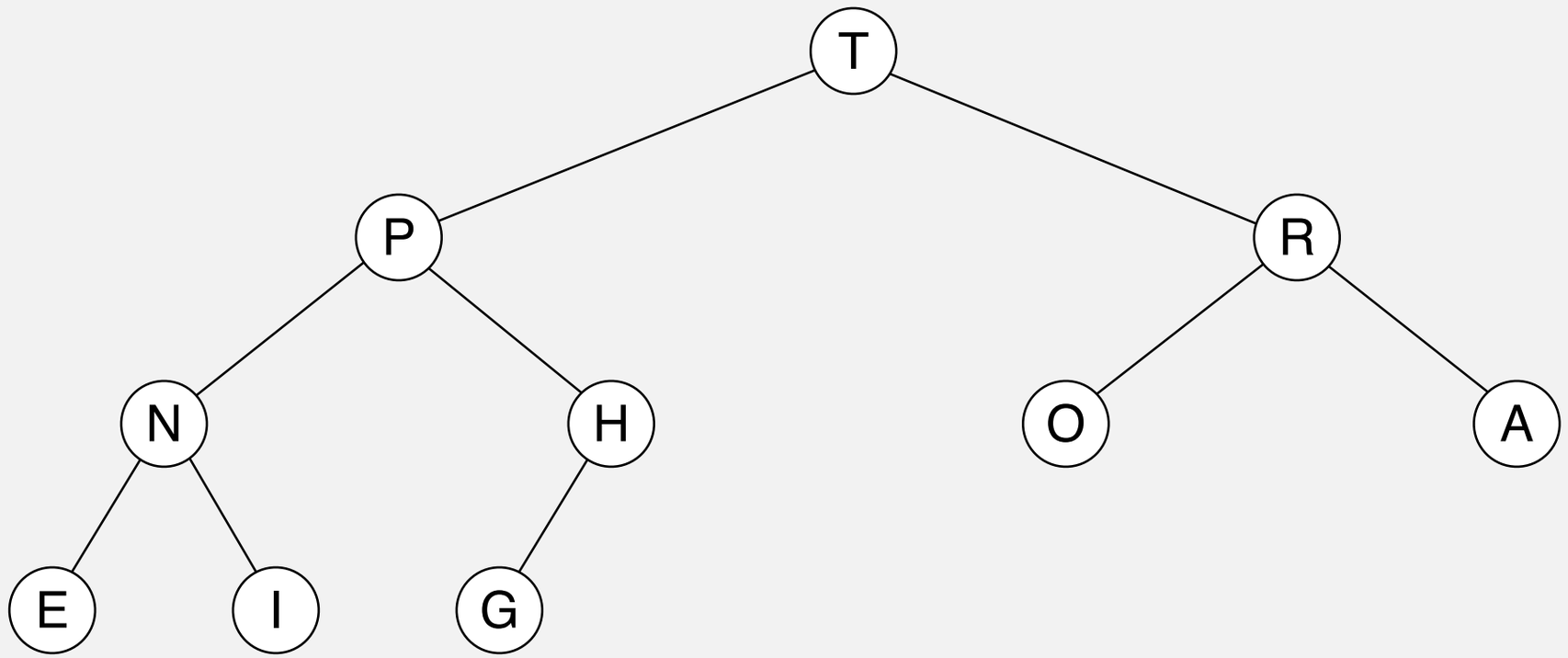


Binary heap operations

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

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

heap ordered



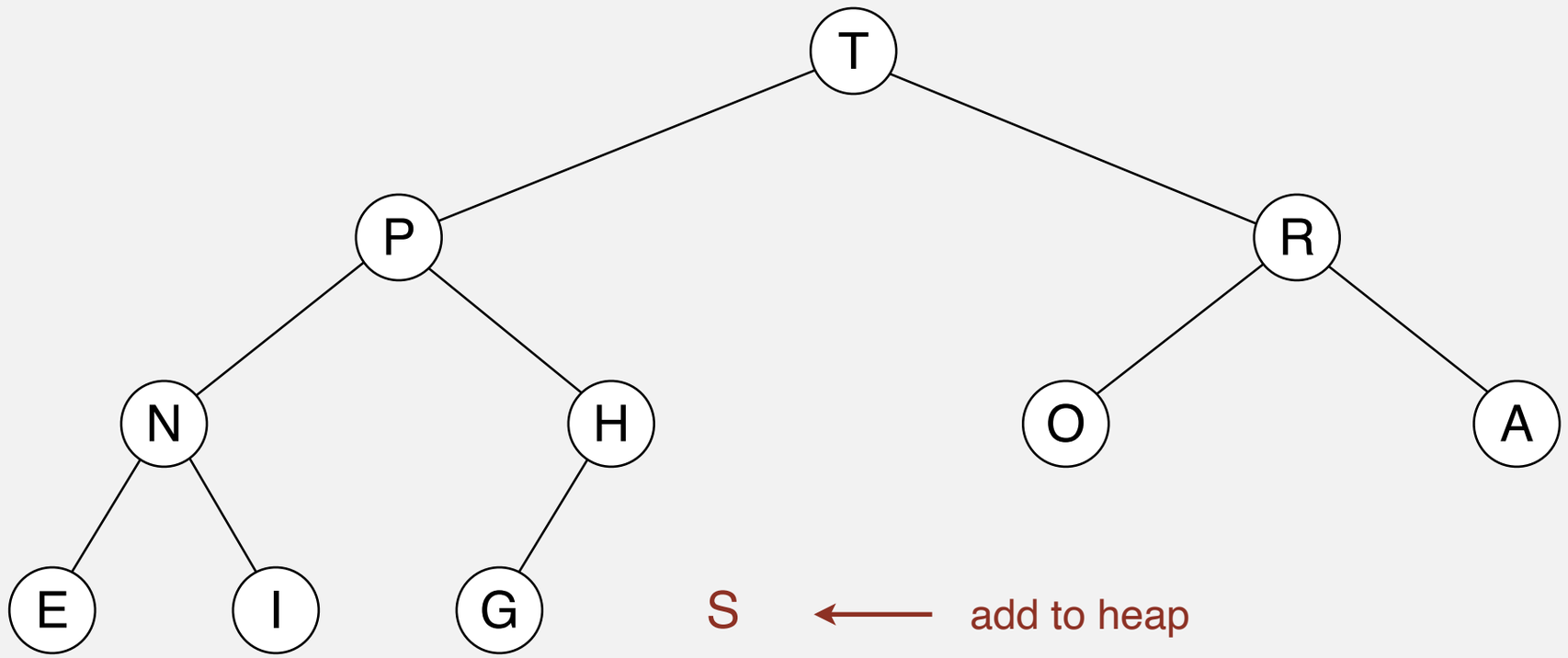
T	P	R	N	H	O	A	E	I	G	

Binary heap operations

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

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

insert S



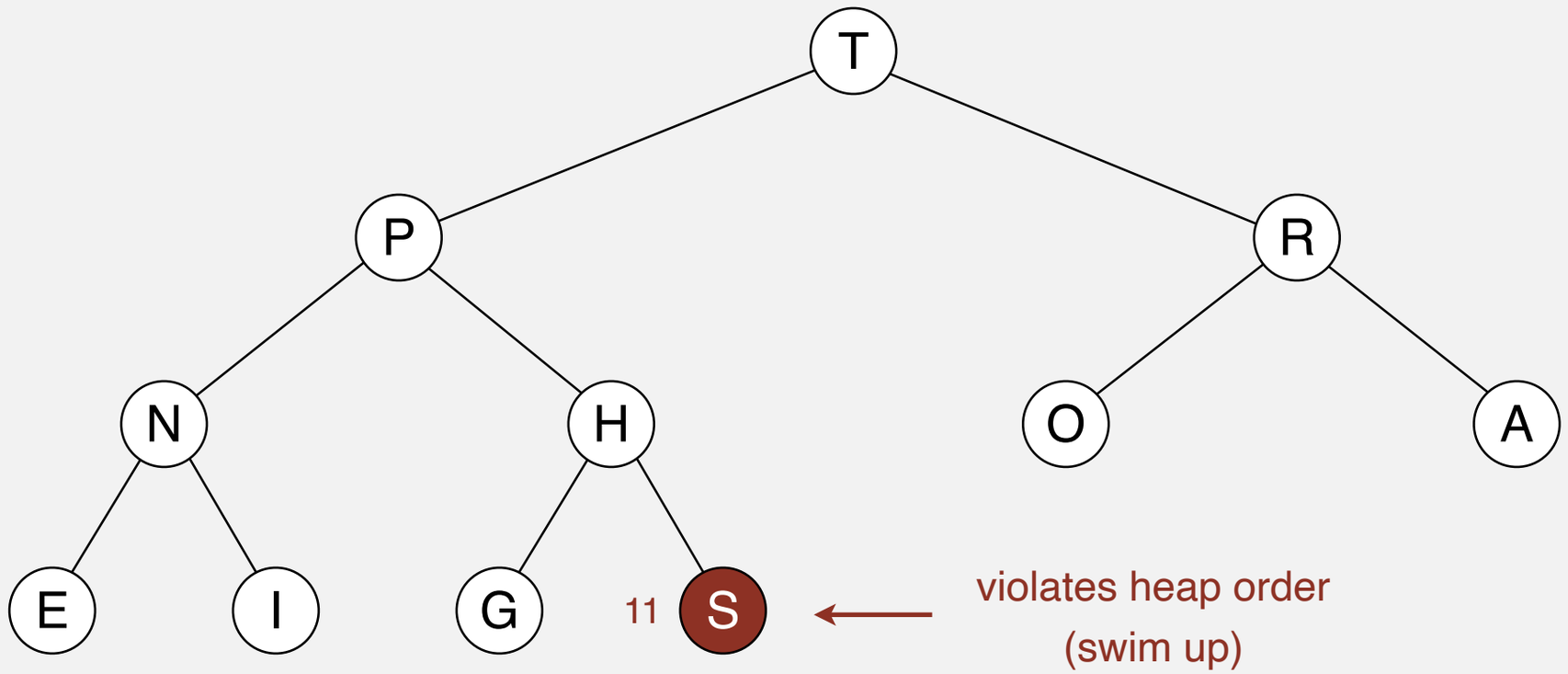
T	P	R	N	H	O	A	E	I	G	

Binary heap operations

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

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

insert S



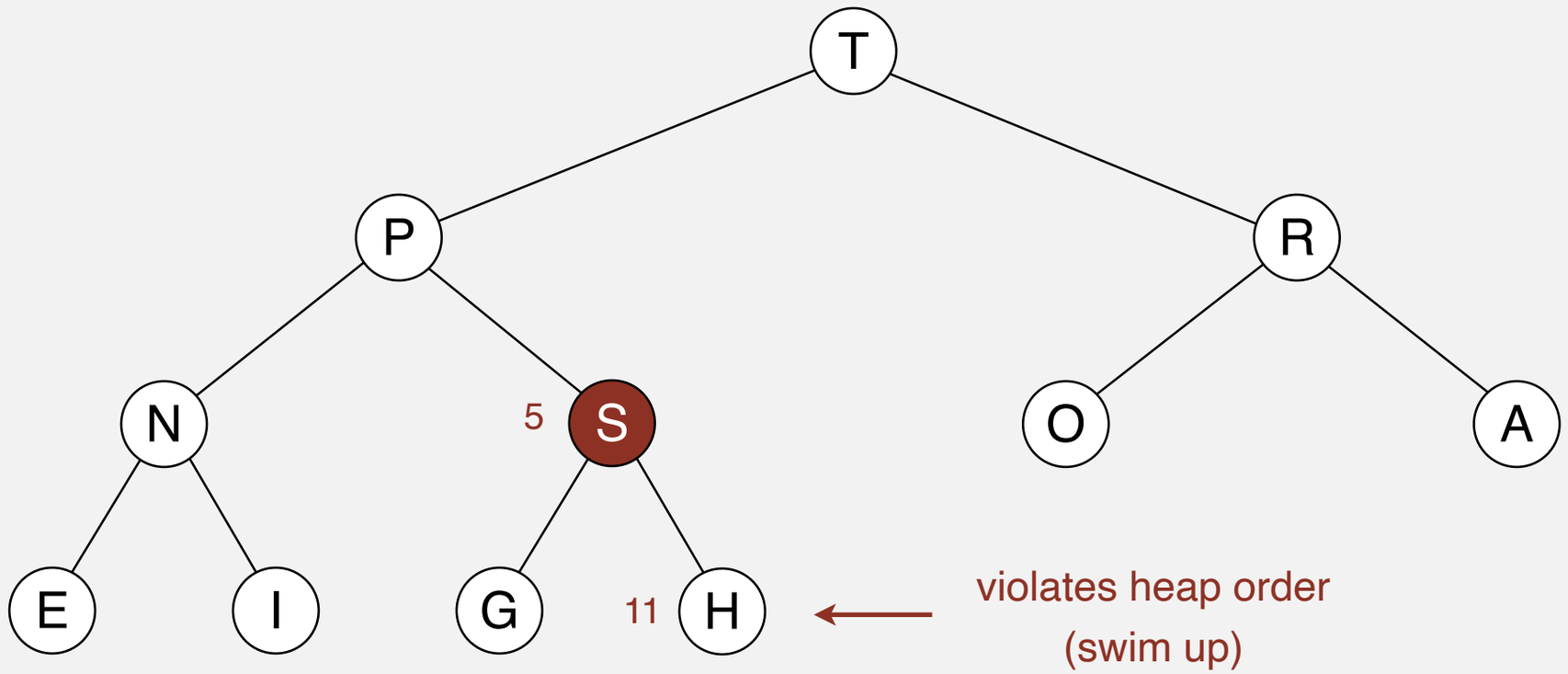
T	P	R	N	H	O	A	E	I	G	S
										11

Binary heap operations

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

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

insert S



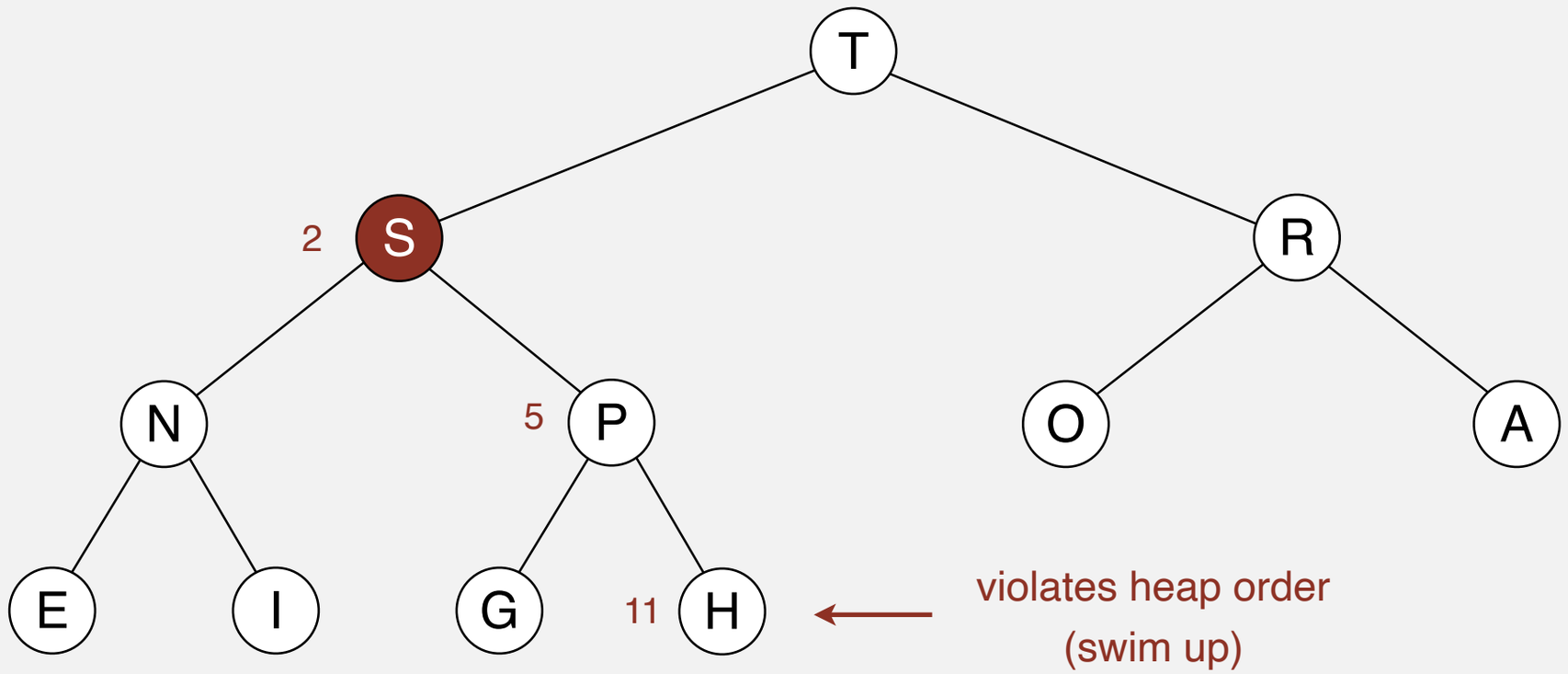
T	P	R	N	S	O	A	E	I	G	H
				5						11

Binary heap operations

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

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

insert S



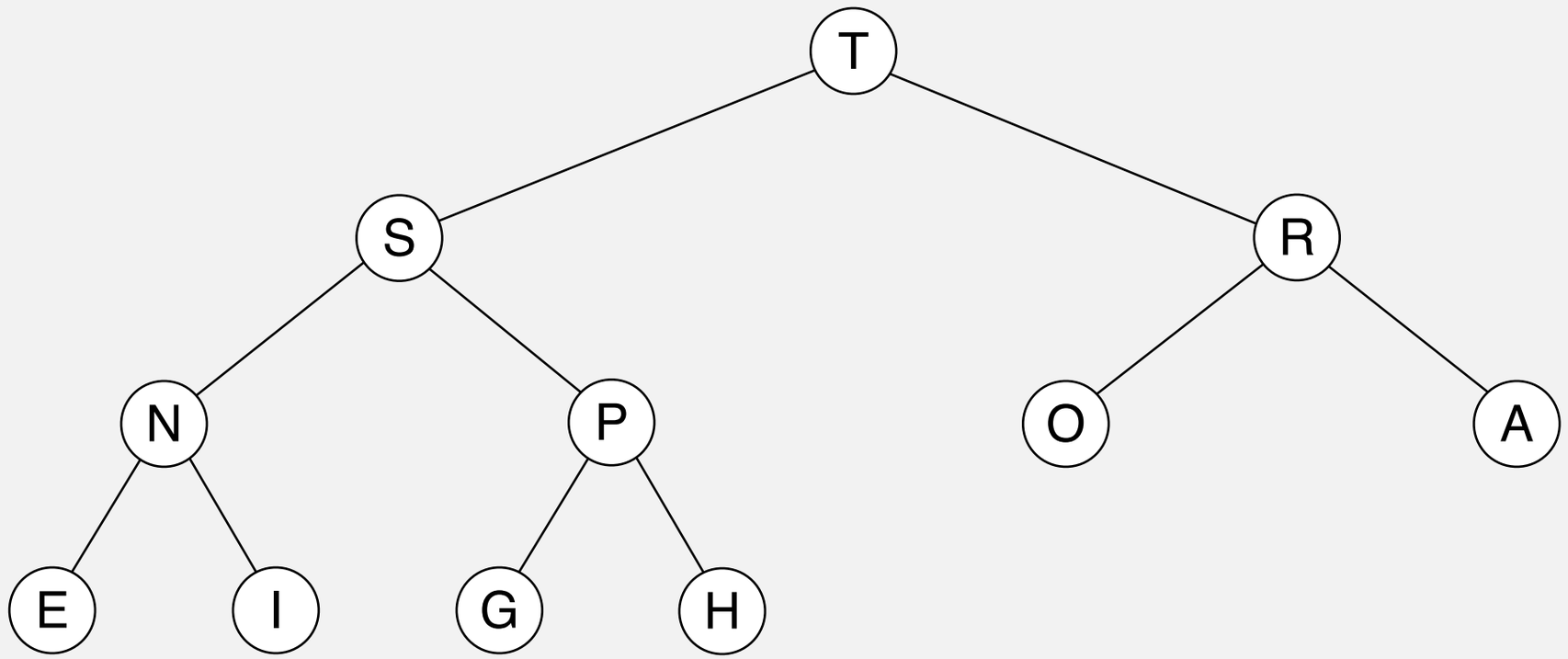
T	S	R	N	P	O	A	E	I	G	H
	2			5						11

Binary heap operations

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

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

heap ordered



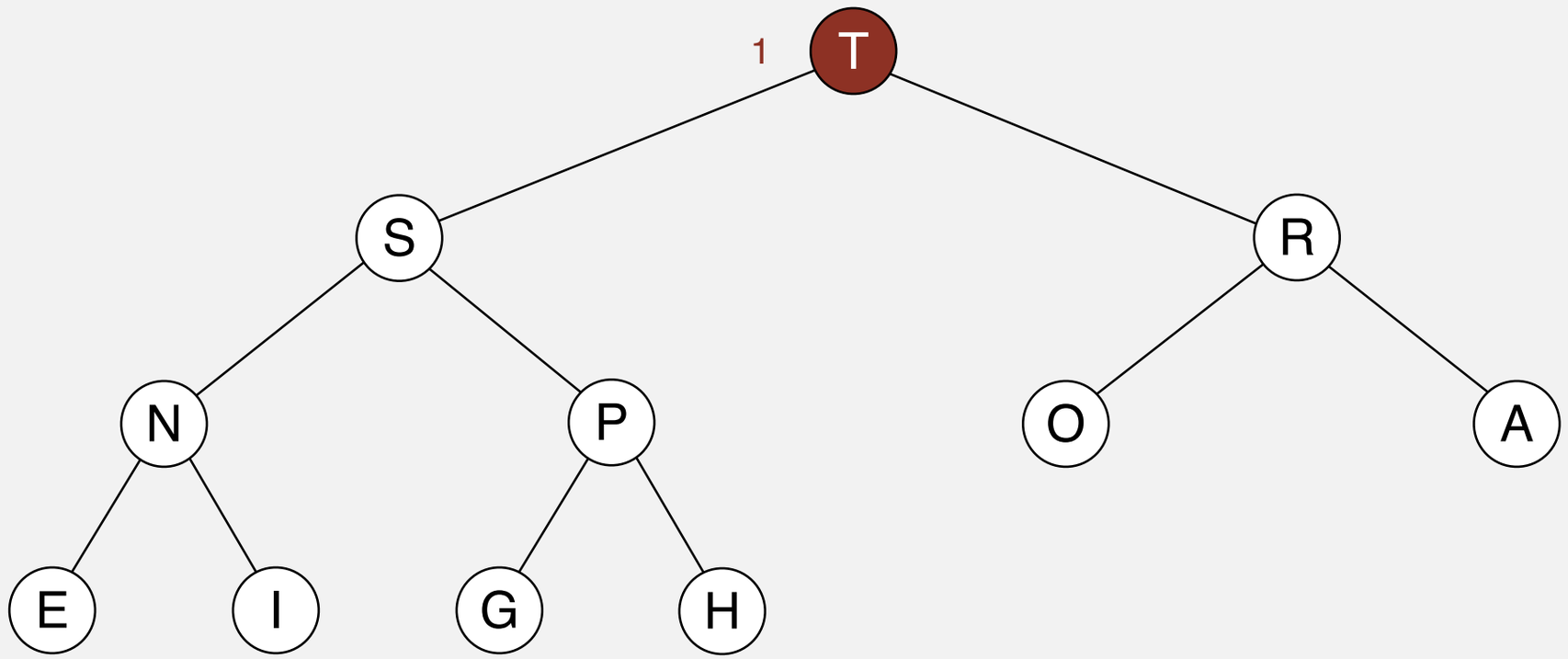
T	S	R	N	P	O	A	E	I	G	H

Binary heap operations

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

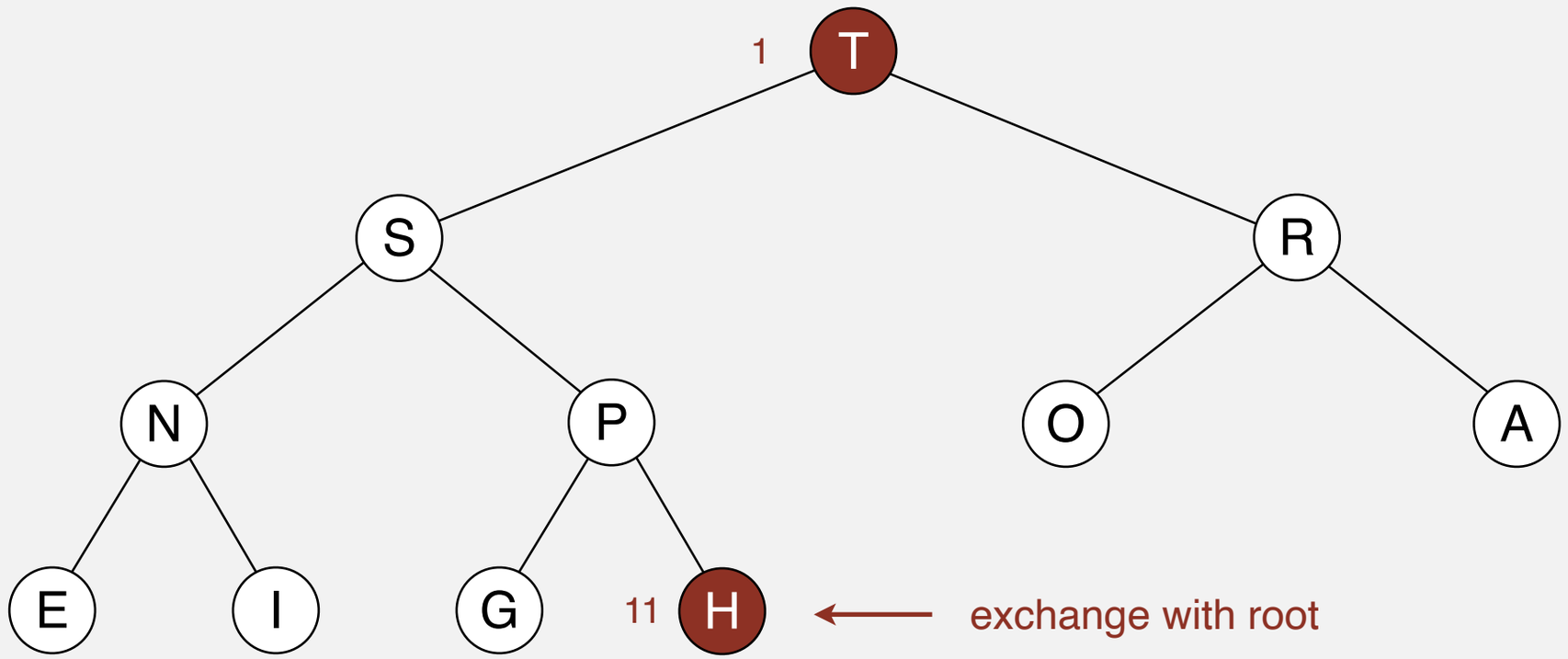


T	S	R	N	P	O	A	E	I	G	H
1										

Binary heap operations

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

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

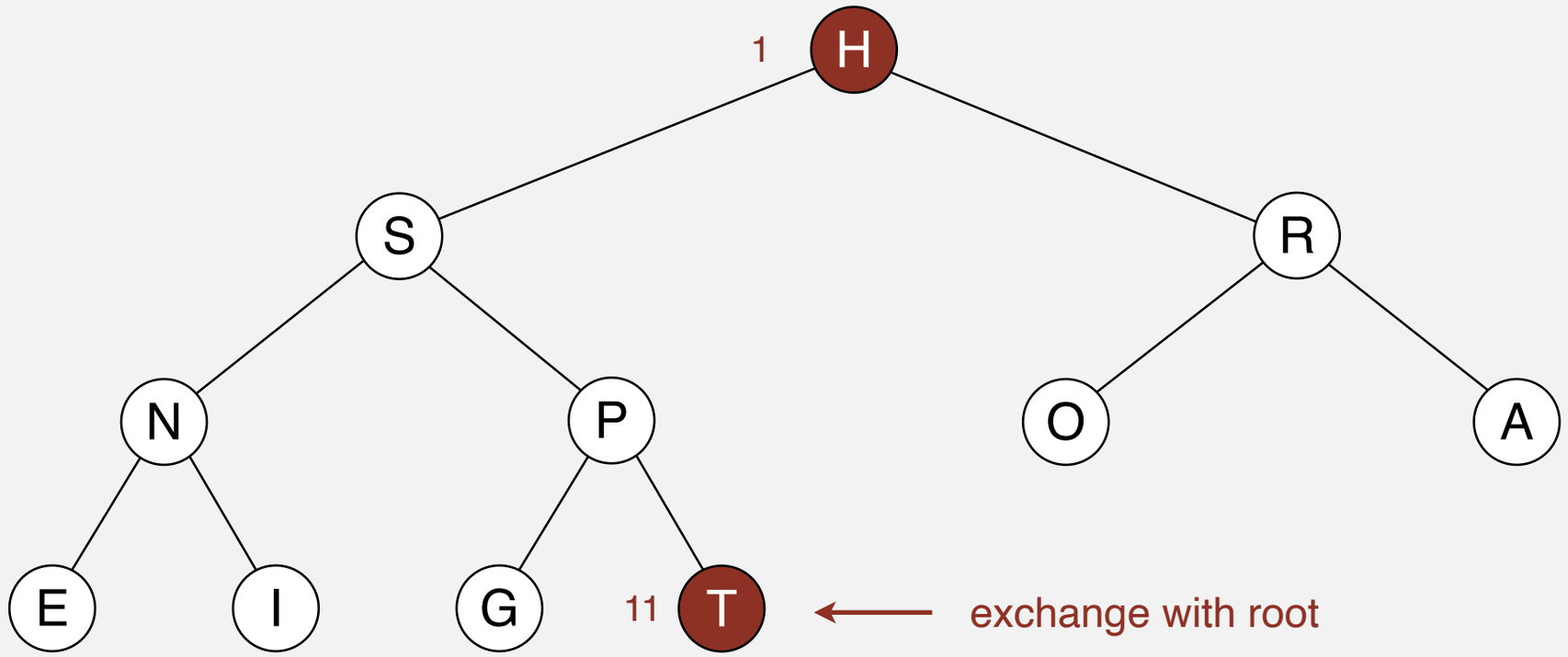


T	S	R	N	P	O	A	E	I	G	H
1										11

Binary heap operations

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

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

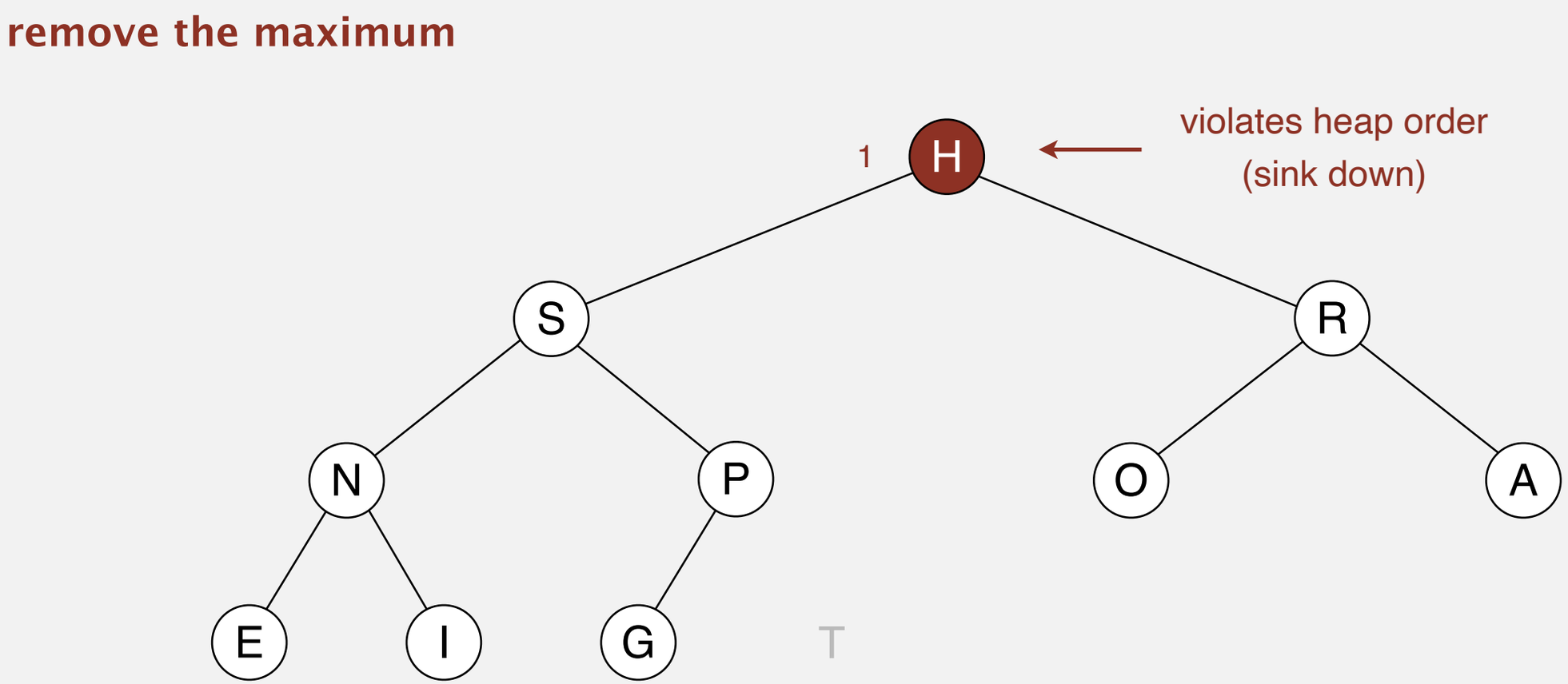


H	S	R	N	P	O	A	E	I	G	T
1										11

Binary heap operations

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

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



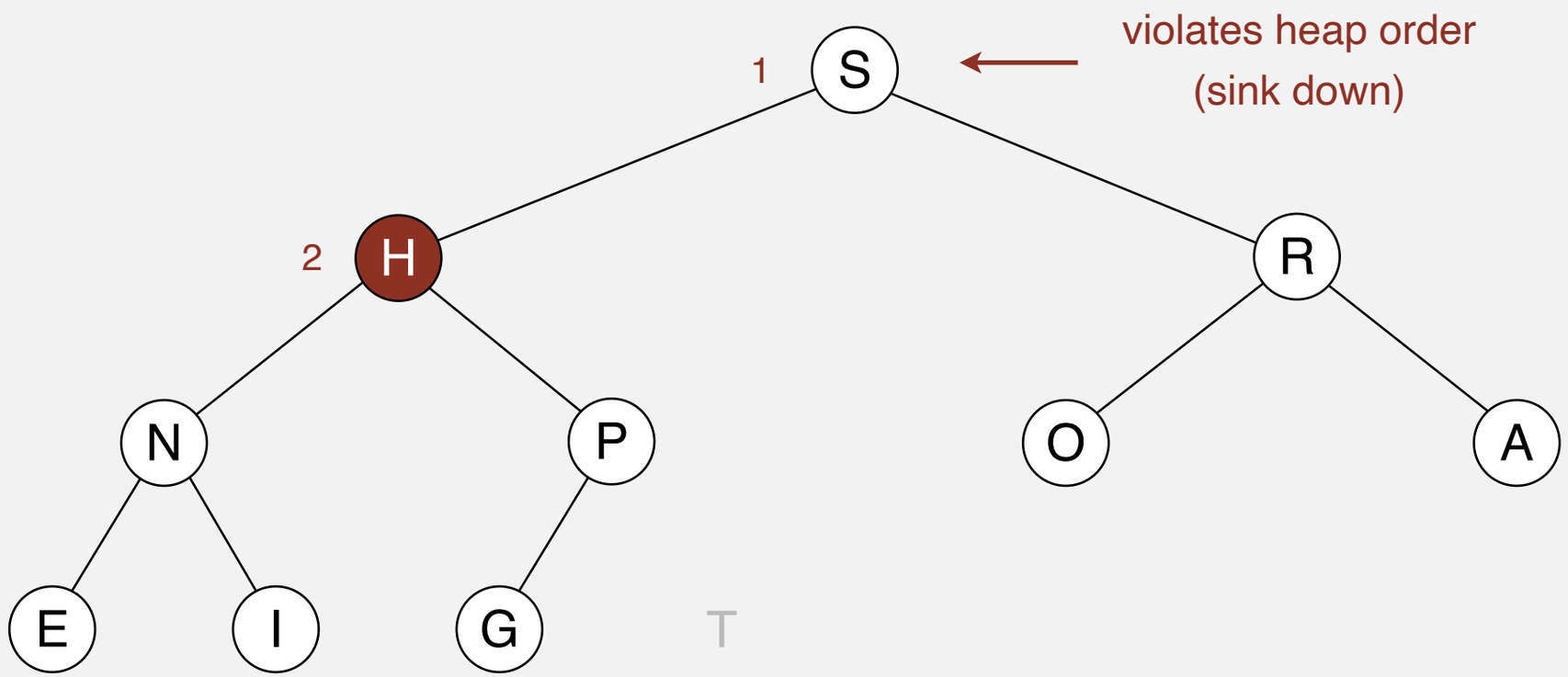
H	S	R	N	P	O	A	E	I	G	T
1										

Binary heap operations

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

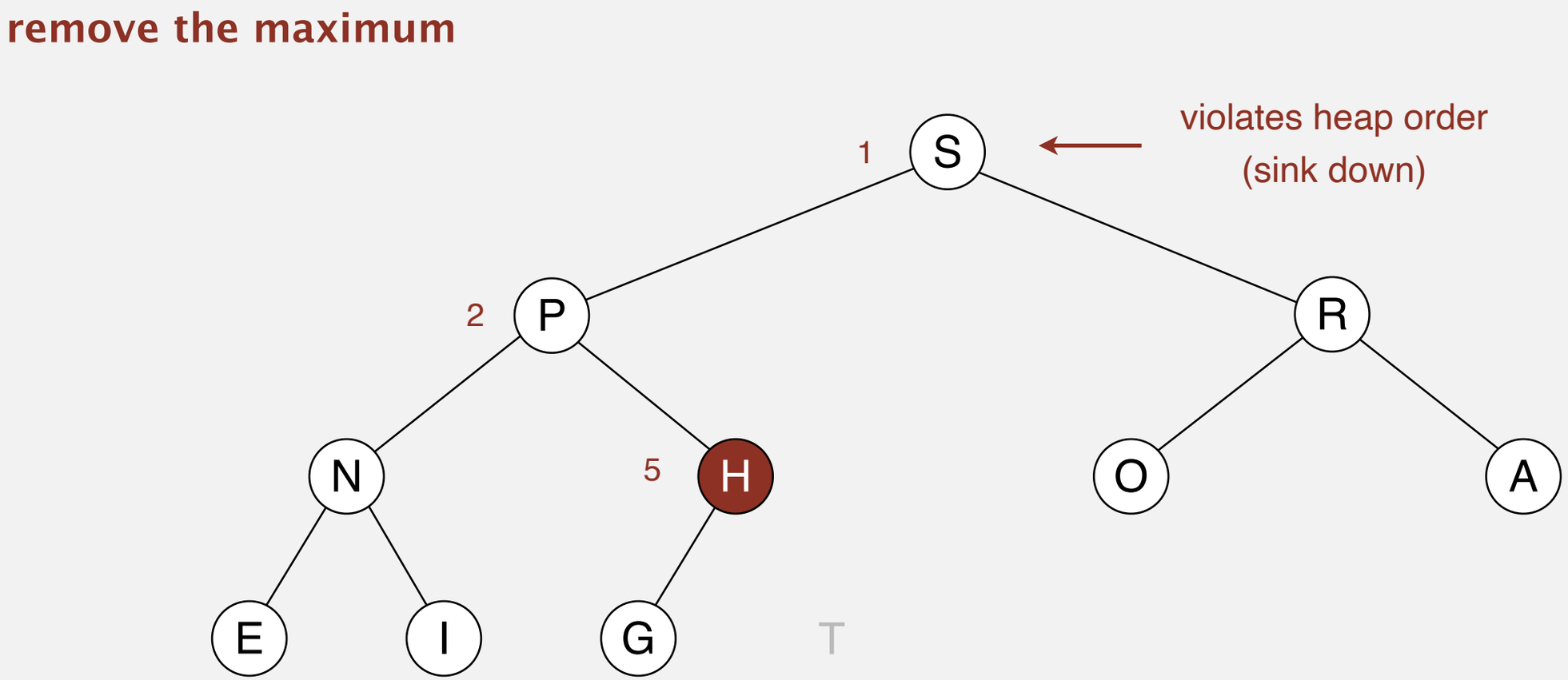


S	H	R	N	P	O	A	E	I	G	T
1	2									

Binary heap operations

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

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



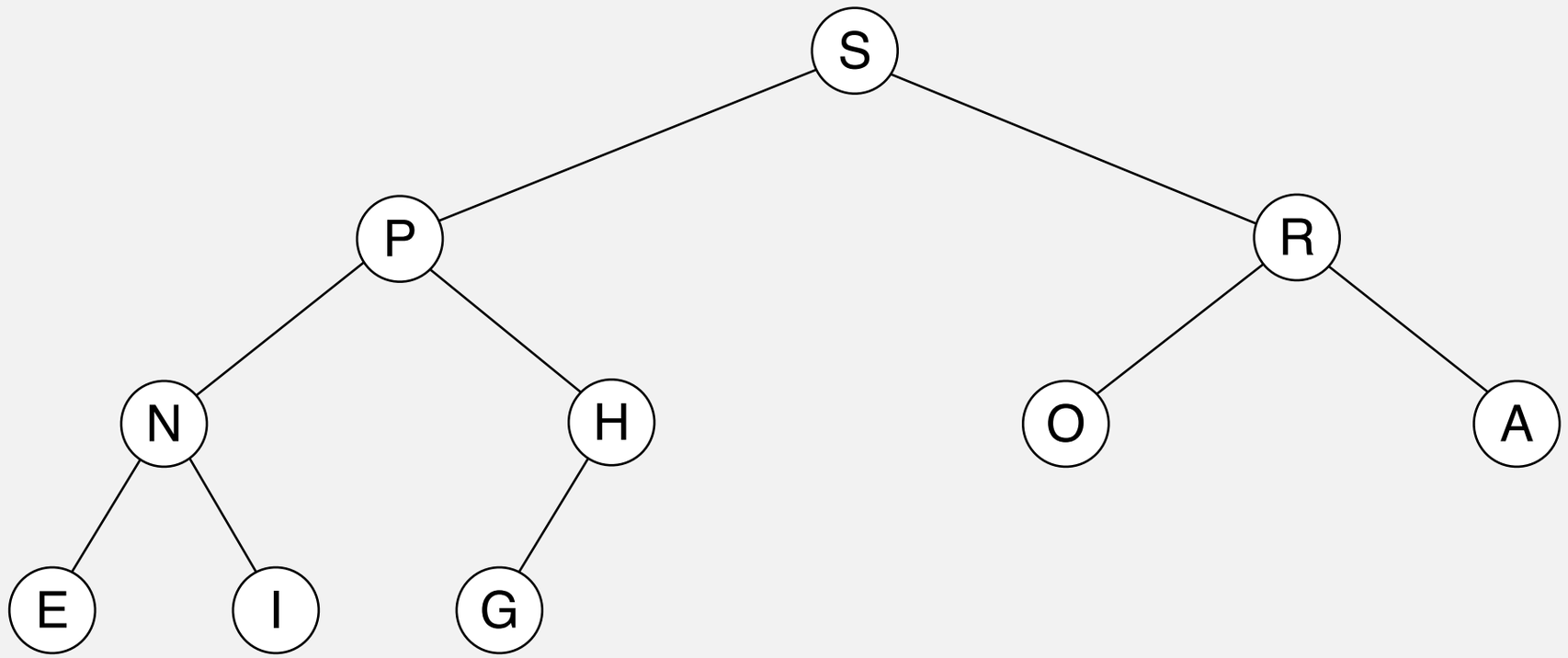
S	P	R	N	H	O	A	E	I	G	T
1	2			5						

Binary heap operations

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

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

heap ordered



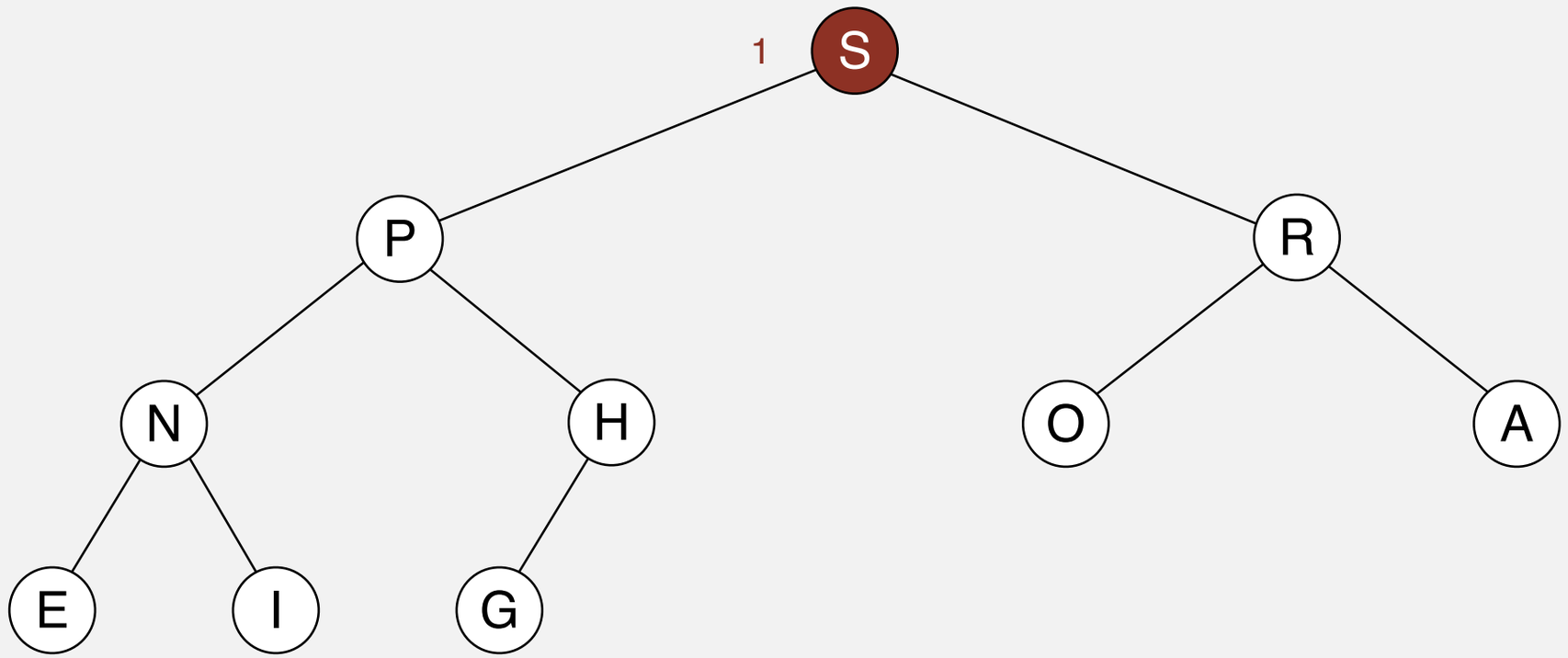
S	P	R	N	H	O	A	E	I	G	

Binary heap operations

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

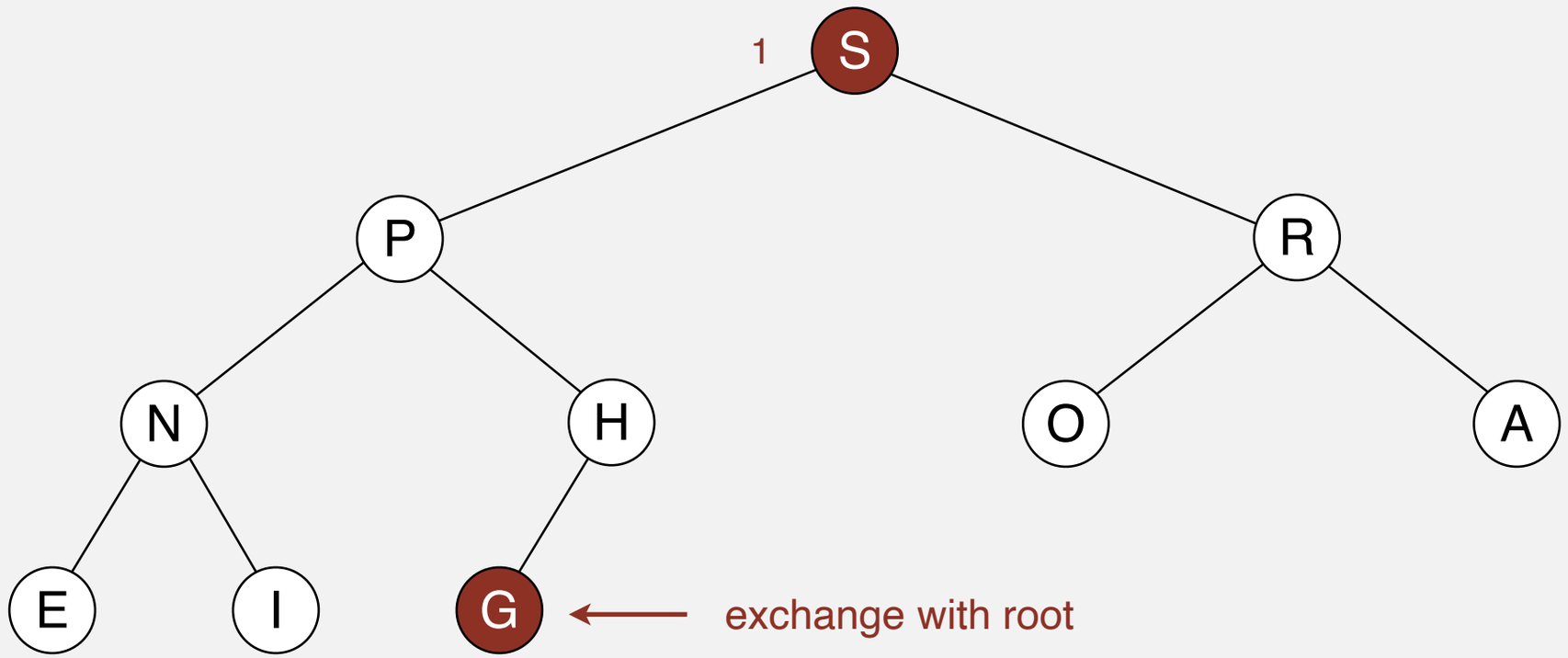


S	P	R	N	H	O	A	E	I	G	
1										

Binary heap operations

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

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

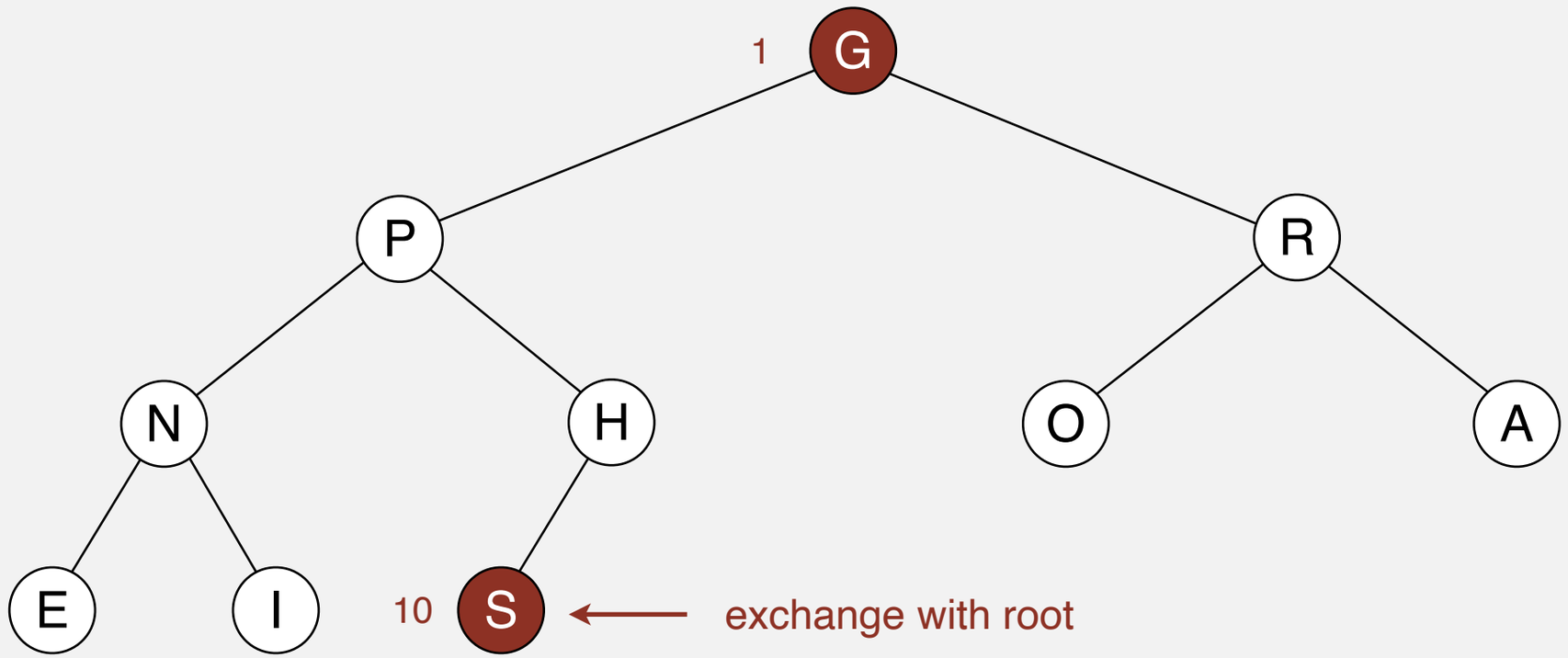


S	P	R	N	H	O	A	E	I	G	
1										

Binary heap operations

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

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

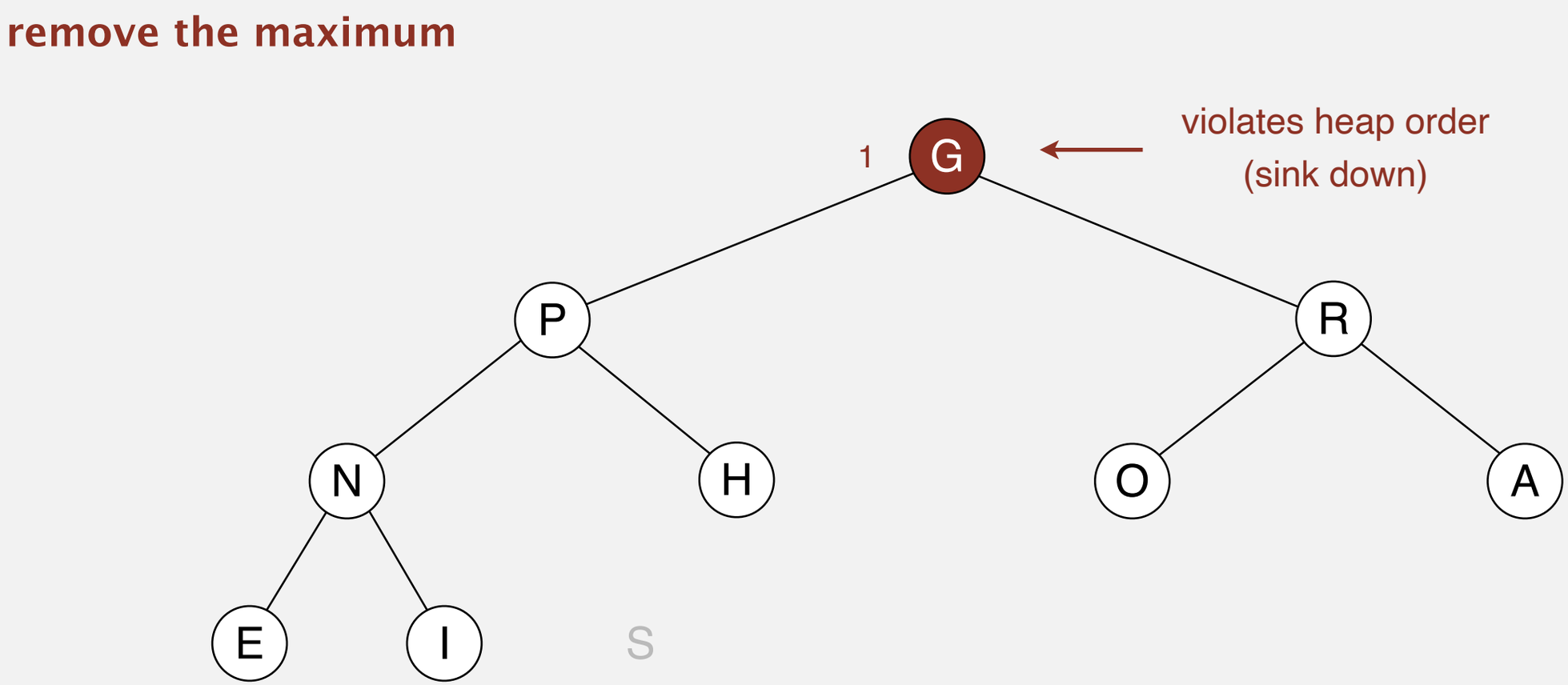


G	P	R	N	H	O	A	E	I	S	
1									10	

Binary heap operations

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

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



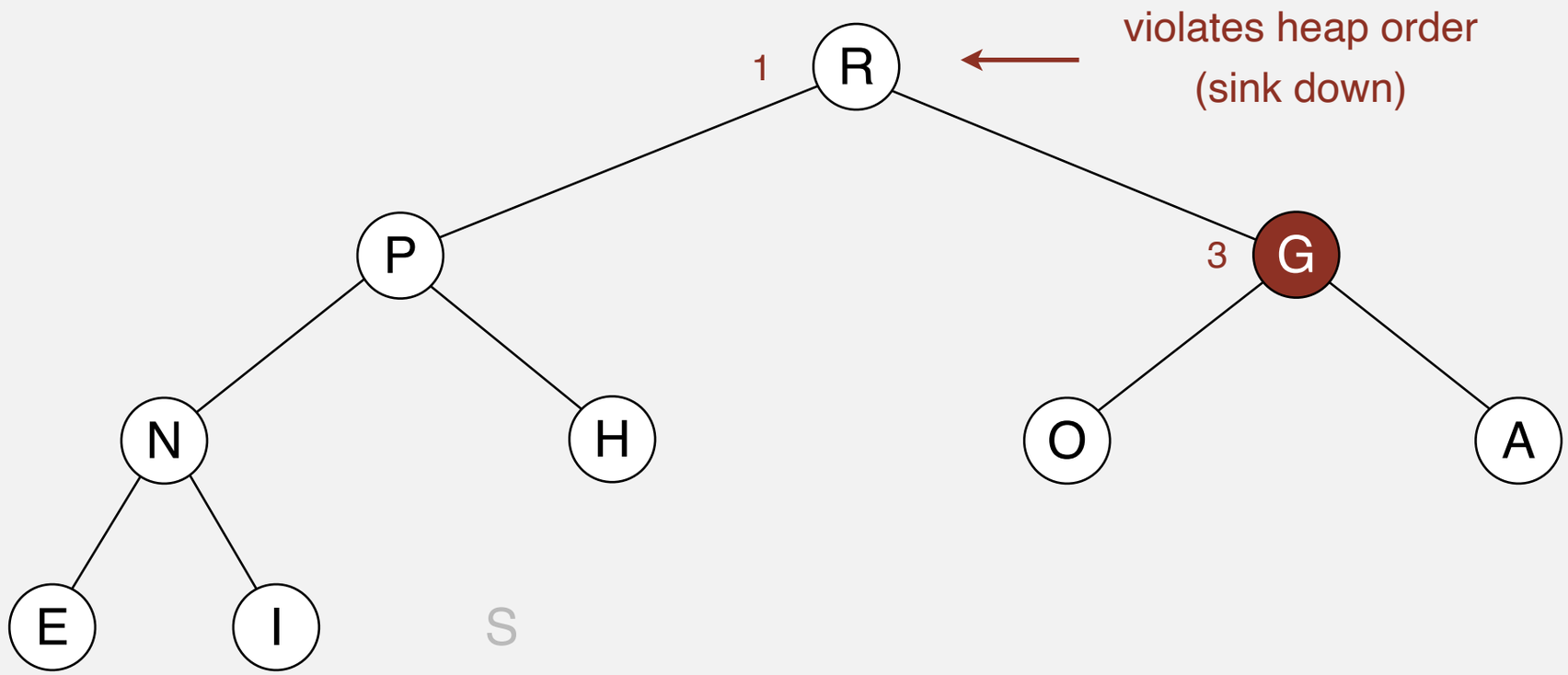
G	P	R	N	H	O	A	E	I	S	
1										

Binary heap operations

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



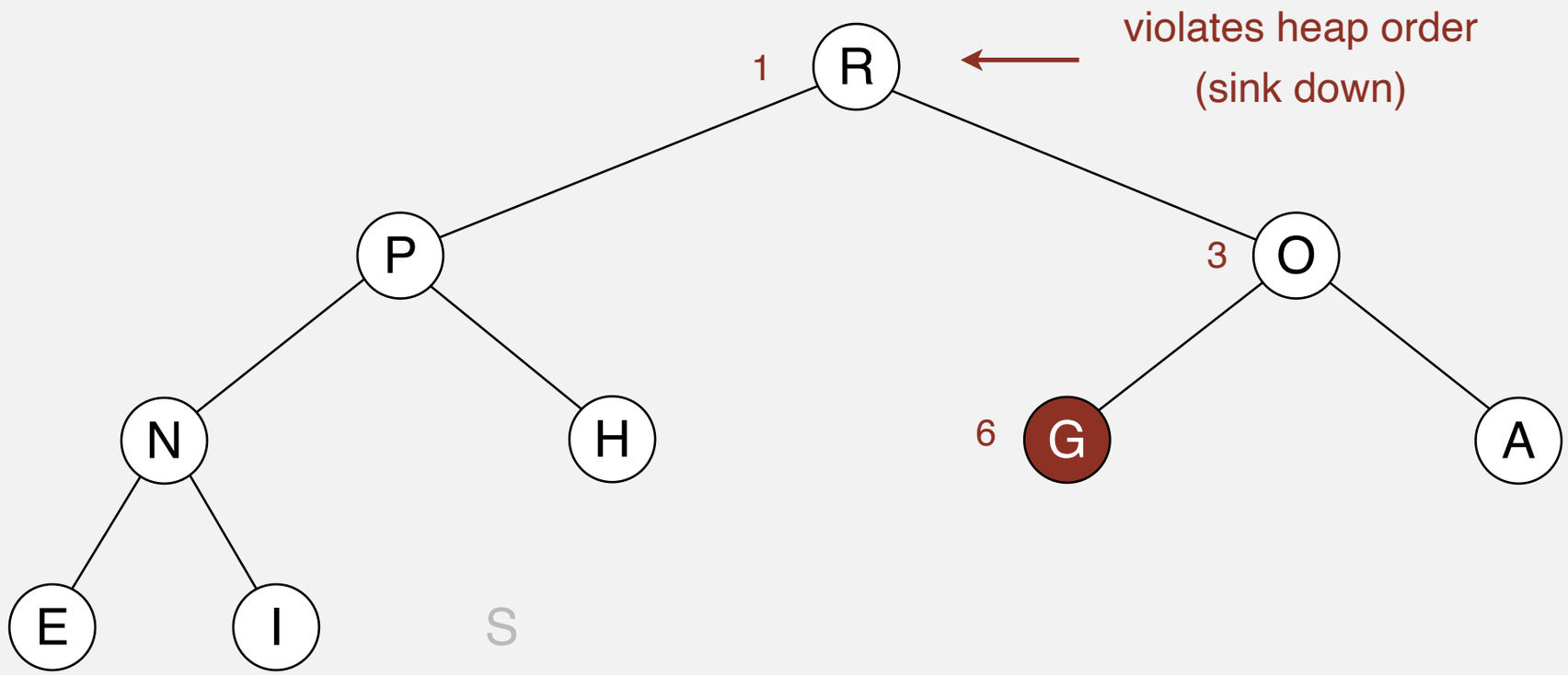
R	P	G	N	H	O	A	E	I	S	
1		3								

Binary heap operations

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



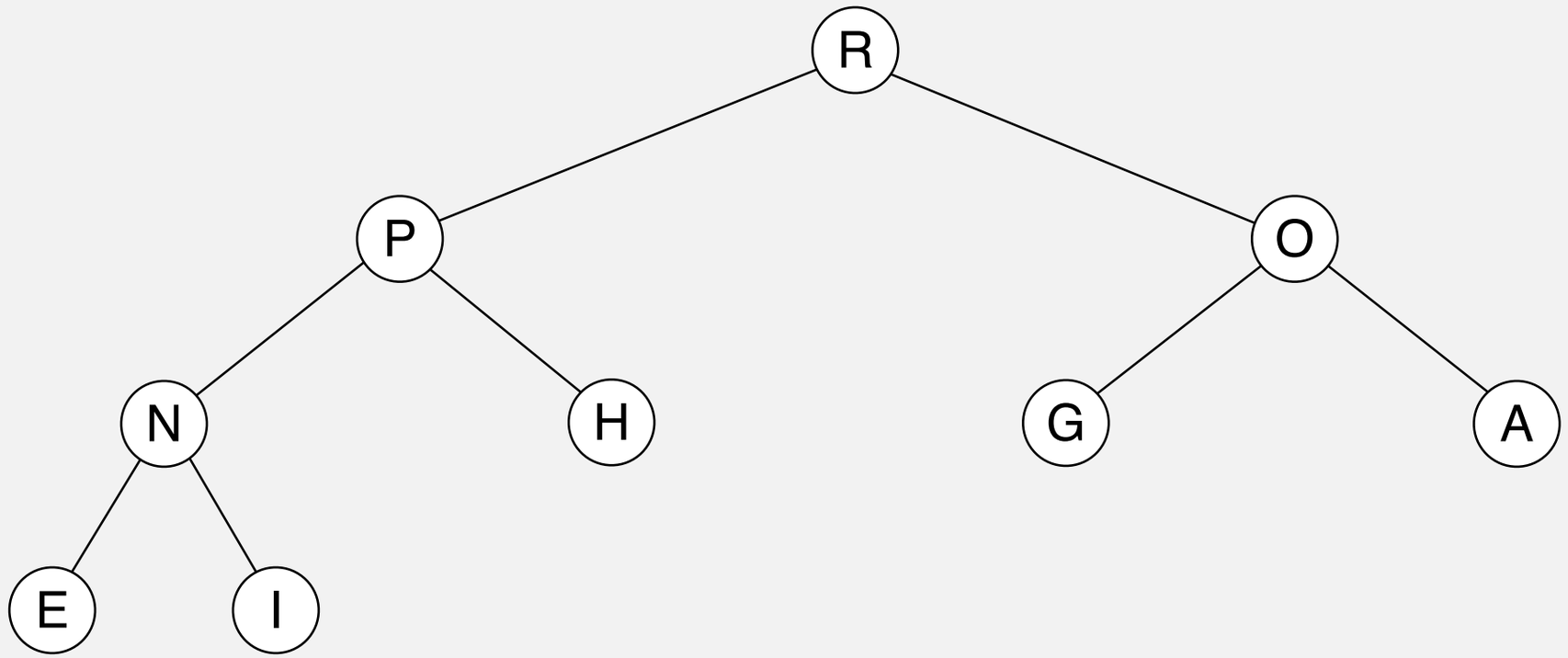
R	P	O	N	H	G	A	E	I	S	
1		3			6					

Binary heap operations

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

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

heap ordered



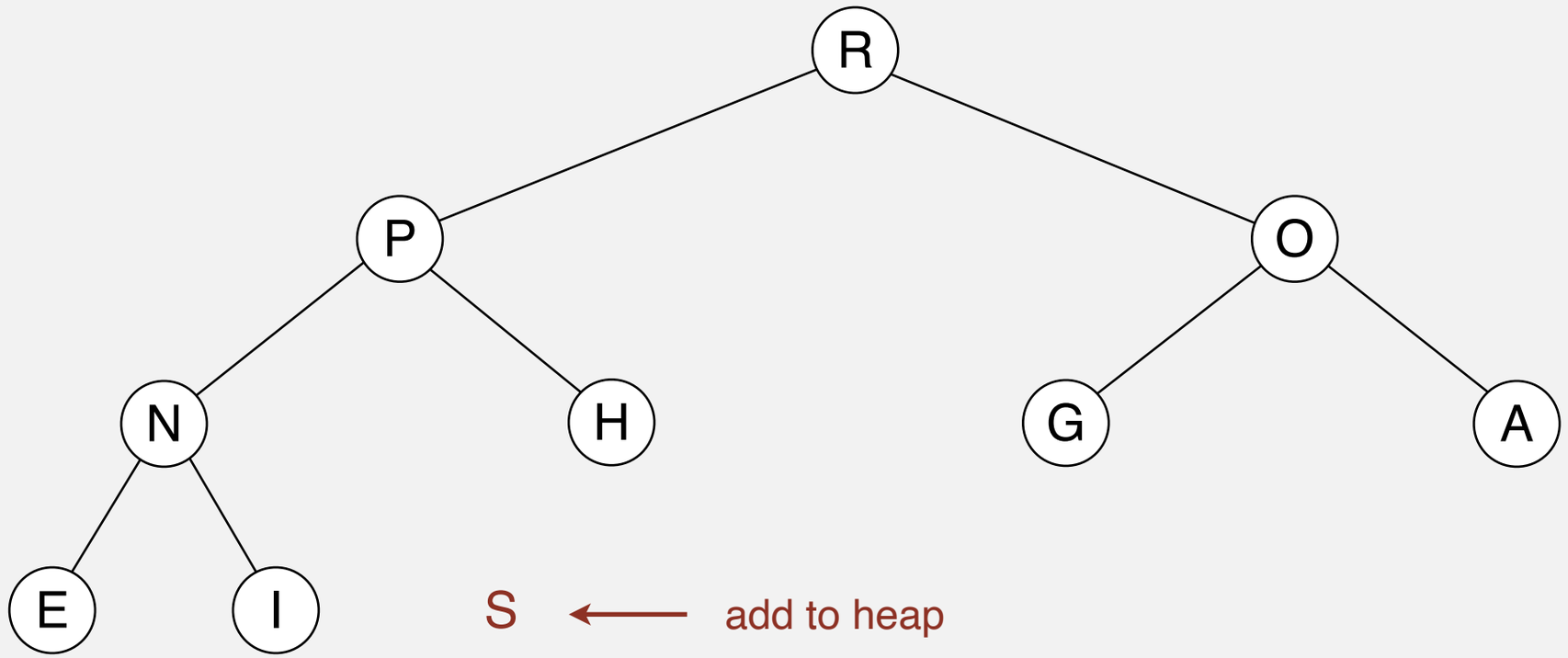
R	P	O	N	H	G	A	E	I		

Binary heap operations

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

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

insert S



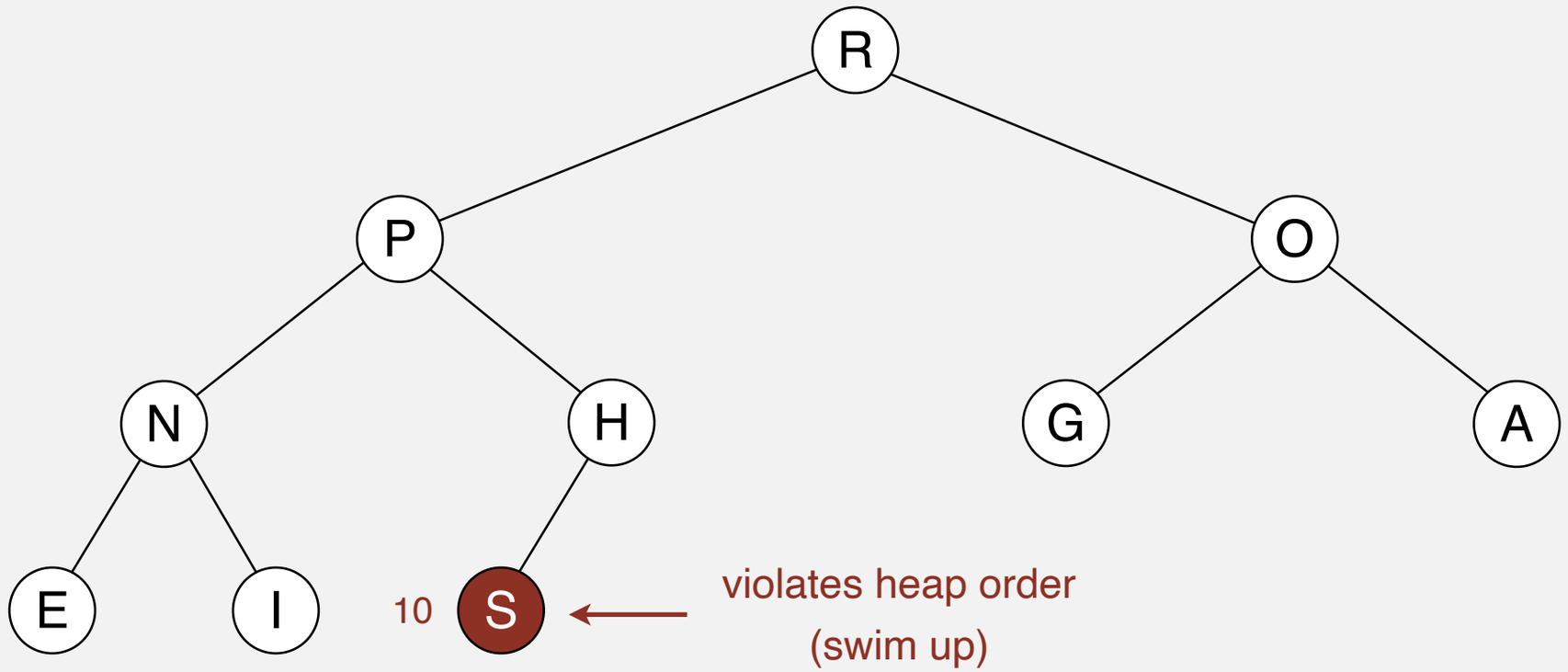
R	P	O	N	H	G	A	E	I	S	
									10	

Binary heap operations

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

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

insert S



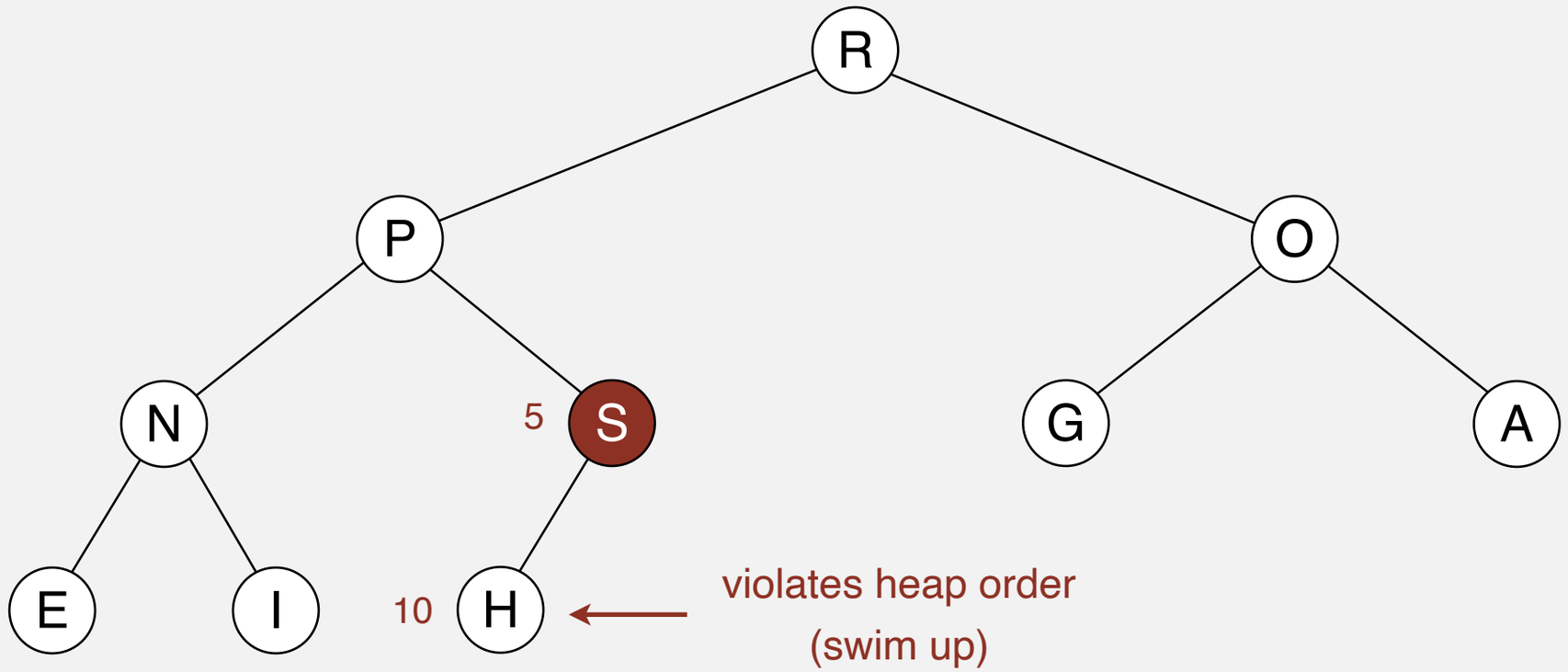
R	P	O	N	H	G	A	E	I	S	
									10	

Binary heap operations

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

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

insert S



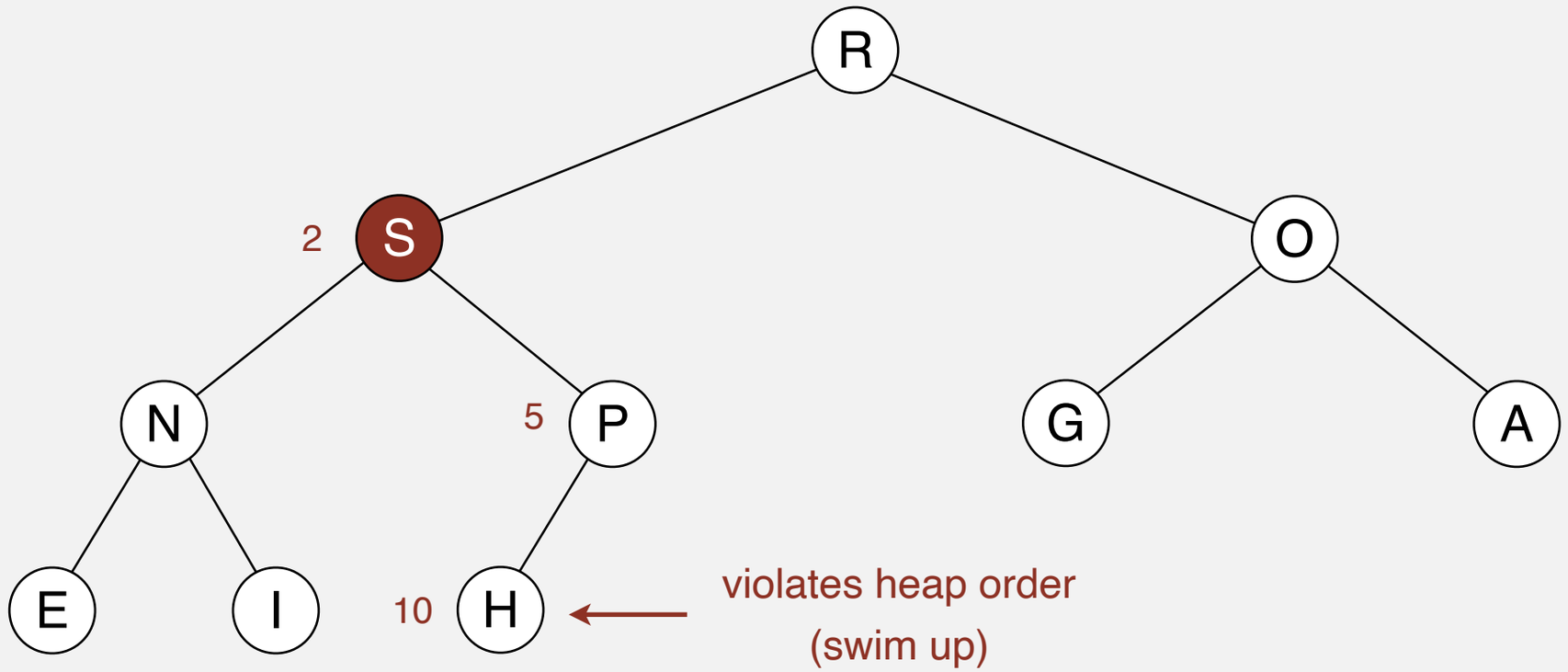
R	P	O	N	S	G	A	E	I	H	
				5					10	

Binary heap operations

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

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

insert S

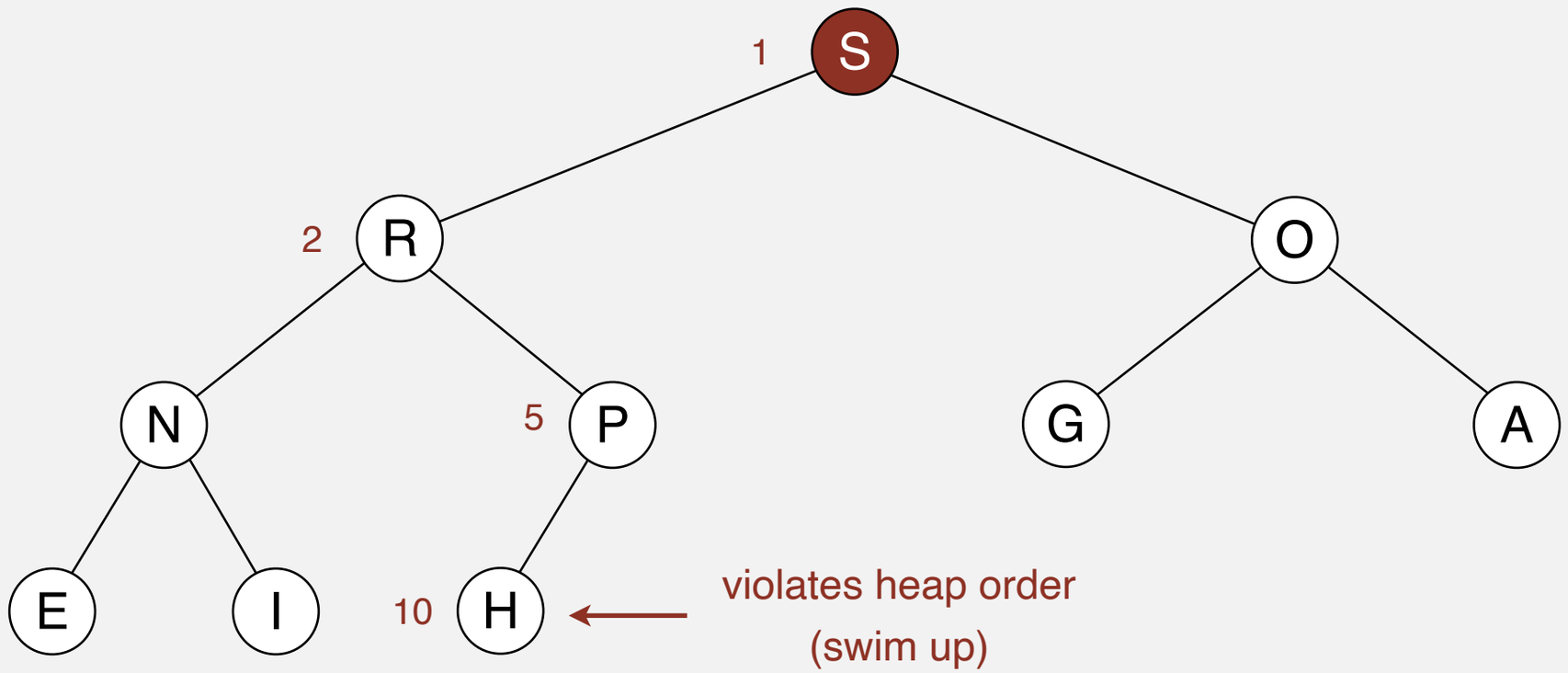


R	S	O	N	P	G	A	E	I	H	
	2			5					10	

Binary heap operations

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

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



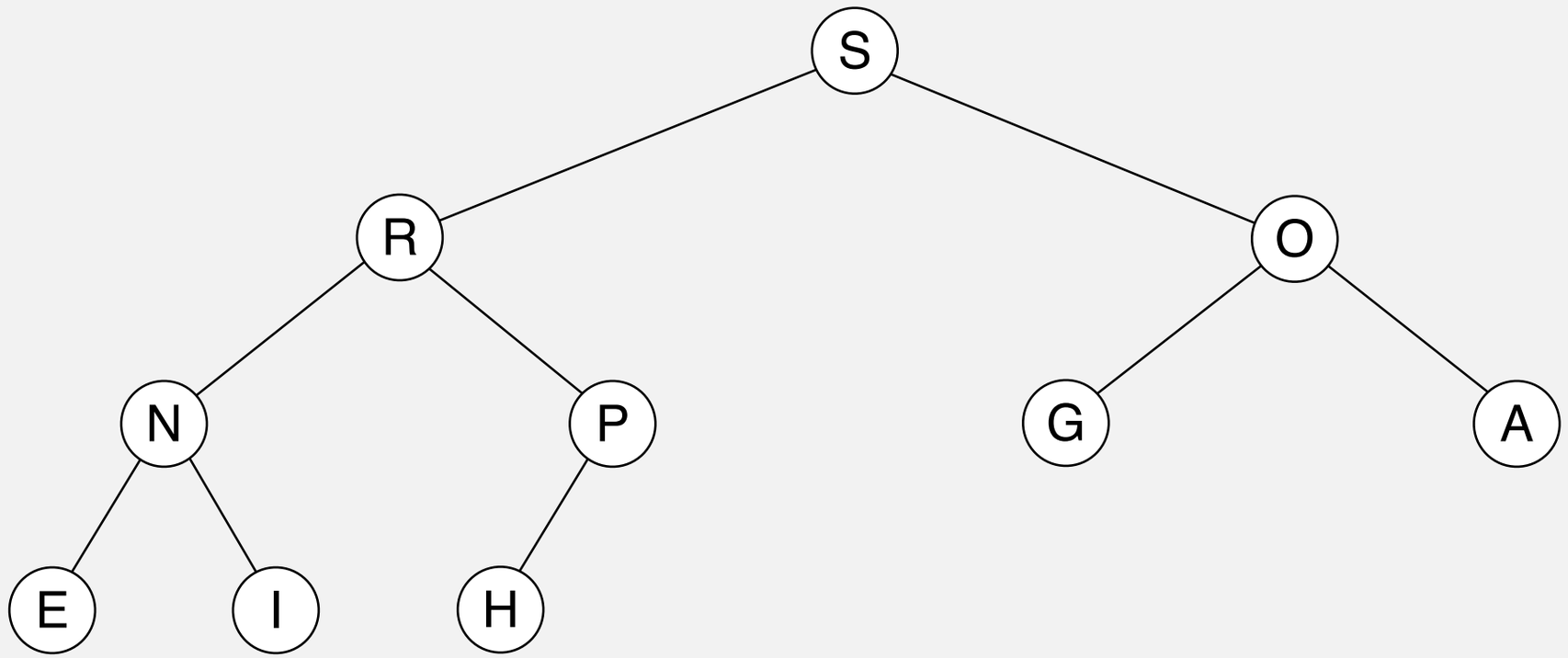
S	R	O	N	P	G	A	E	I	H	
1	2			5					10	

Binary heap operations

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	

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];  }

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

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

   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;  }
}
```

PQ ops

heap helper functions

array helper functions

Priority queues implementation cost summary

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

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	log N	log N	1
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	log N †	1
impossible	1	1	1

← why impossible?

† amortized

Binary heap considerations

Immutability of keys.

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

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 with `sink()` and `swim()` [stay tuned]

Immutability: implementing in Java

Data type. Set of values and operations on those values.

Immutable data type. Can't change the data type value once created.

```
public final class Vector {
   private final int N;
   private final double[] data;

   public Vector(double[] data) {
      this.N = data.length;
      this.data = new double[N];
      for (int i = 0; i < N; i++)
         this.data[i] = data[i];
   }

   ...
}
```

can't override instance methods

all instance variables private and final

defensive copy of mutable instance variables

instance methods don't change instance variables

Immutable. `String`, `Integer`, `Double`, `Color`, `Vector`, `Transaction`, `Point2D`.

Mutable. `StringBuilder`, `Stack`, `Counter`, Java array.

Immutability: properties

Data type. Set of values and operations on those values.

Immutable data type. Can't change the data type value once created.

Advantages.

- Simplifies debugging.
- Safer in presence of hostile code.
- Simplifies concurrent programming.
- Safe to use as key in priority queue or symbol table.

Disadvantage. Must create new object for each data type value.

> *" Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible. "*
>
> *— Joshua Bloch (Java architect)*

Priority Queues and Heapsort

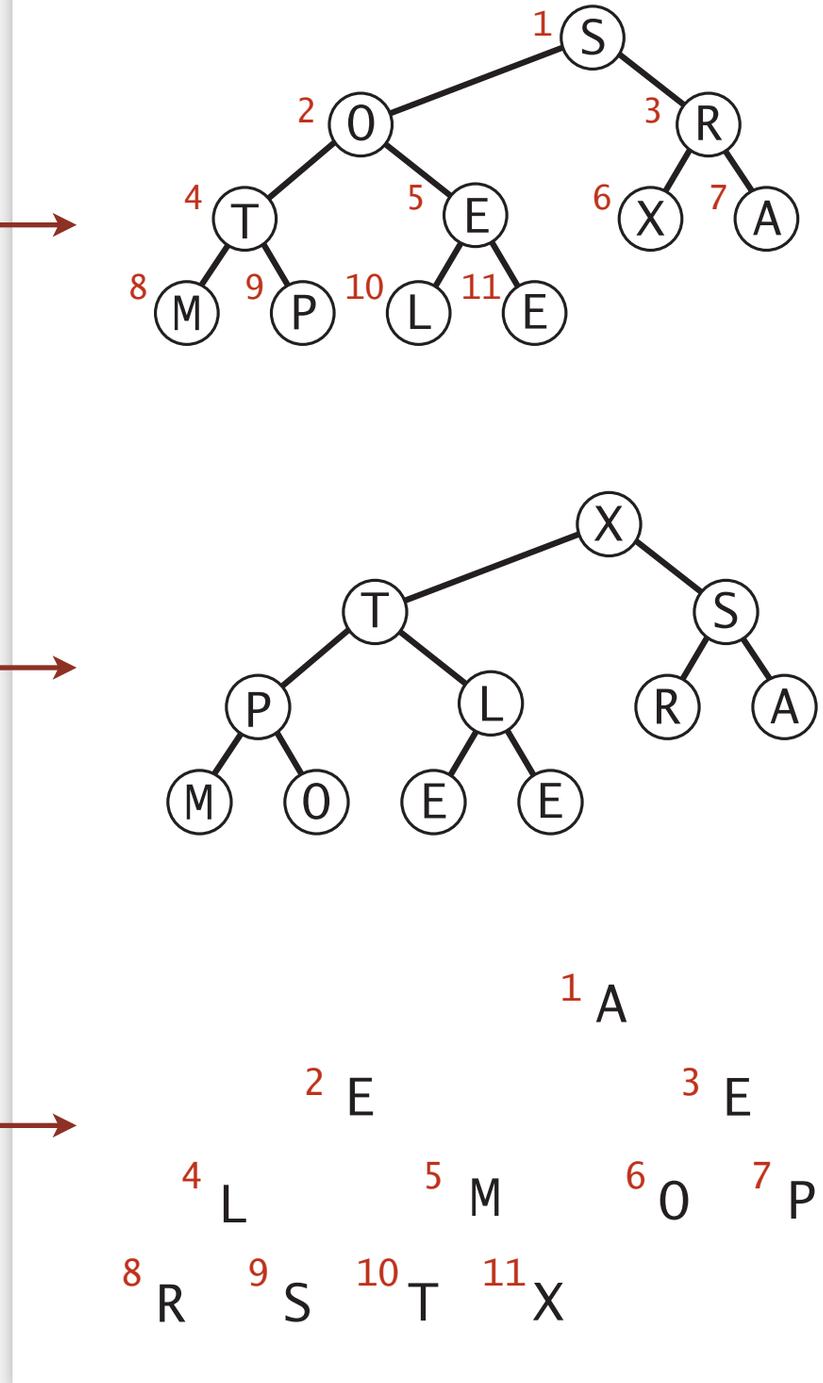
- **Heapsort**
- API
- Elementary implementations
- Binary heaps
- **Heapsort**

Heapsort

Basic plan for in-place sort.

- Create max-heap with all N keys.
- Repeatedly remove the maximum key.

start with array of keys in arbitrary order



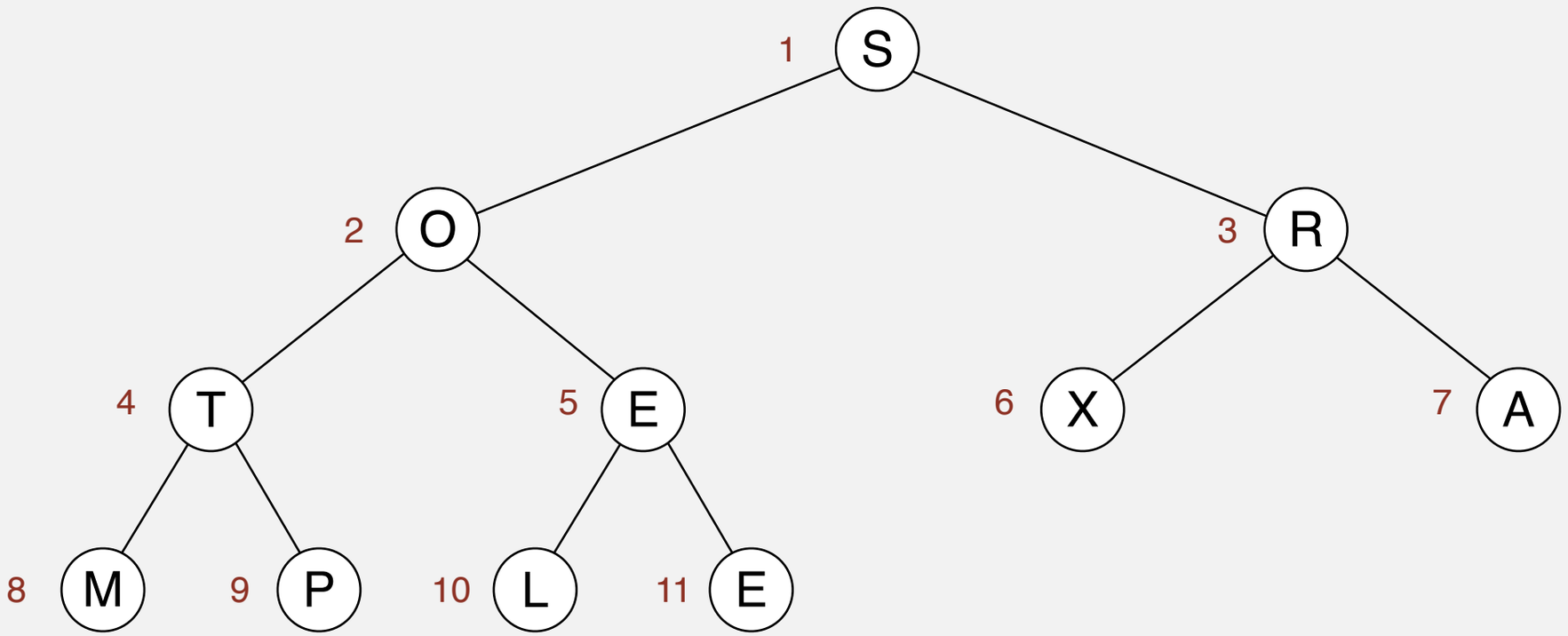
build a max-heap (in place)

sorted result (in place)

Heapsort

Starting point. Array in arbitrary order.

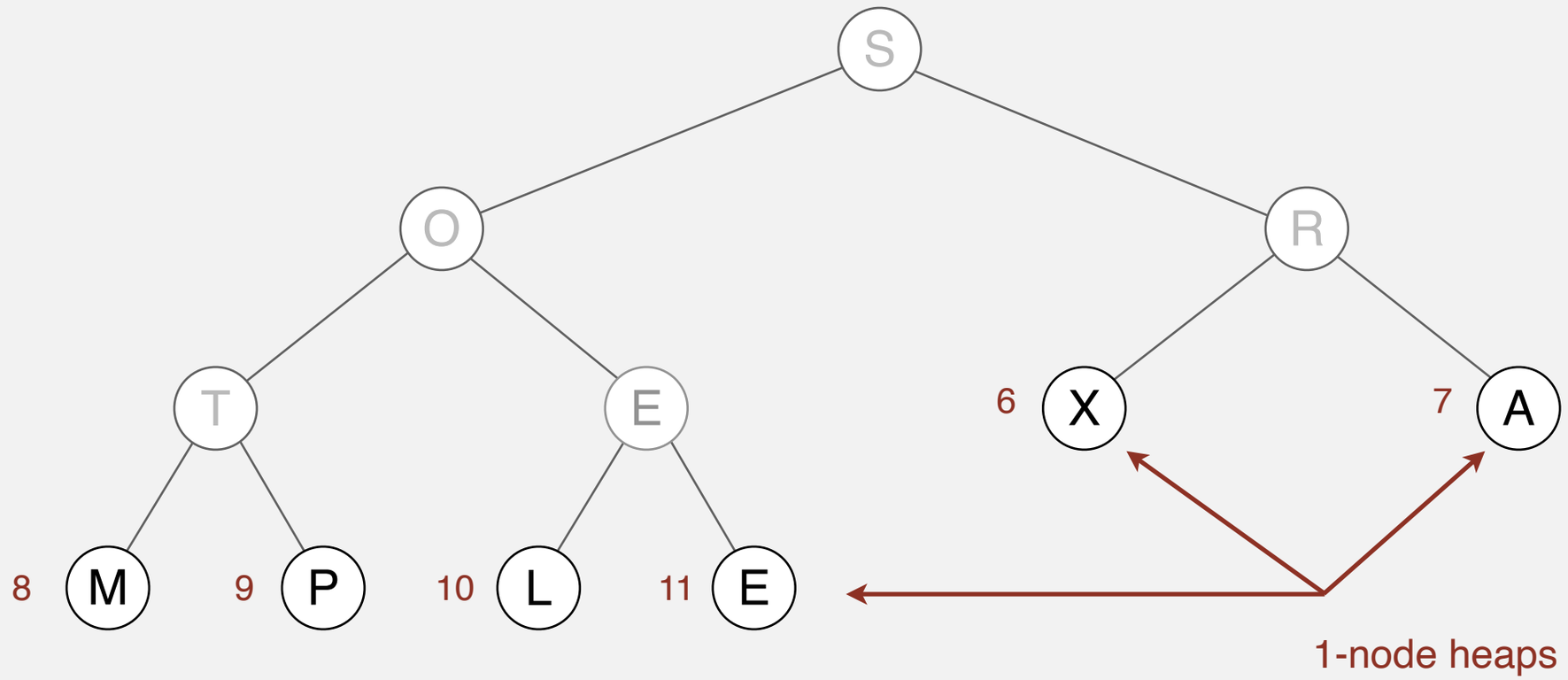
we assume array entries are indexed 1 to N



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

Heapsort

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

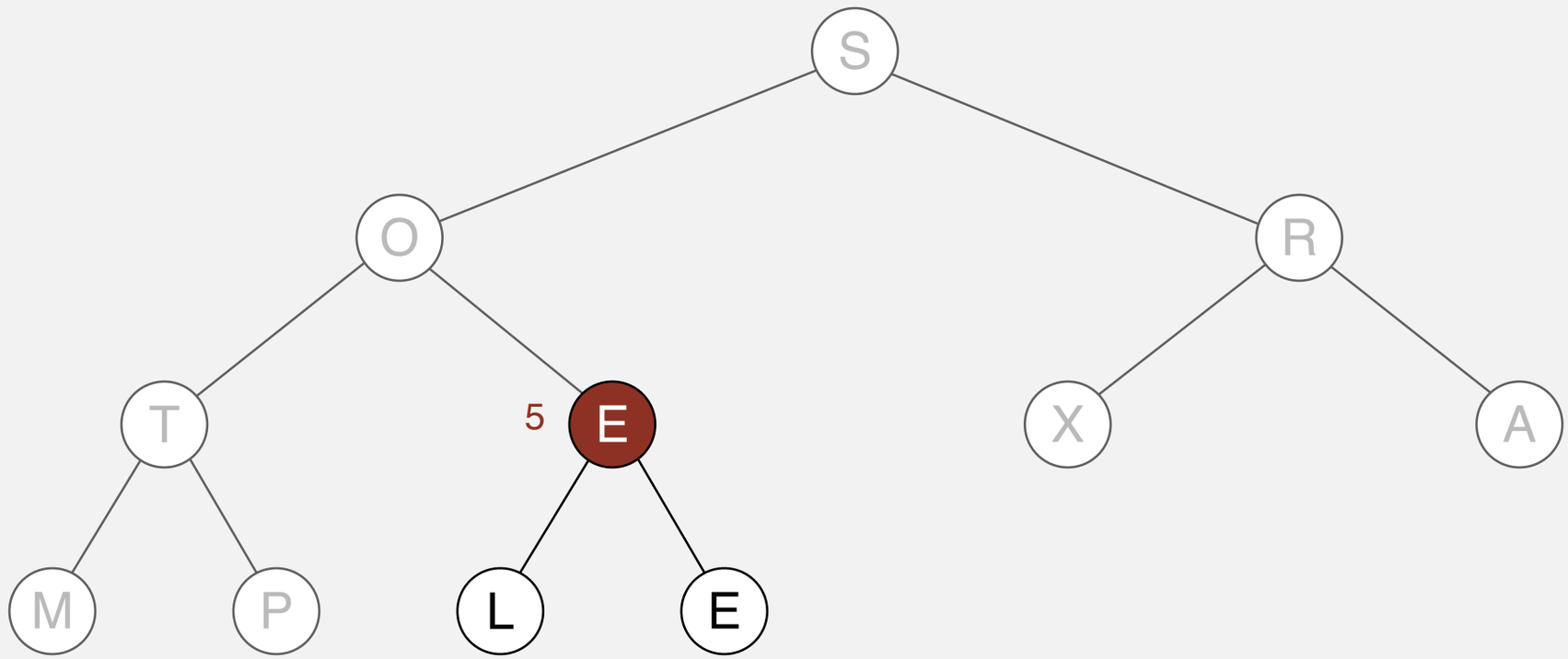


S	O	R	T	E	X	A	M	P	L	E
					6	7	8	9	10	11

Heapsort

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

sink 5

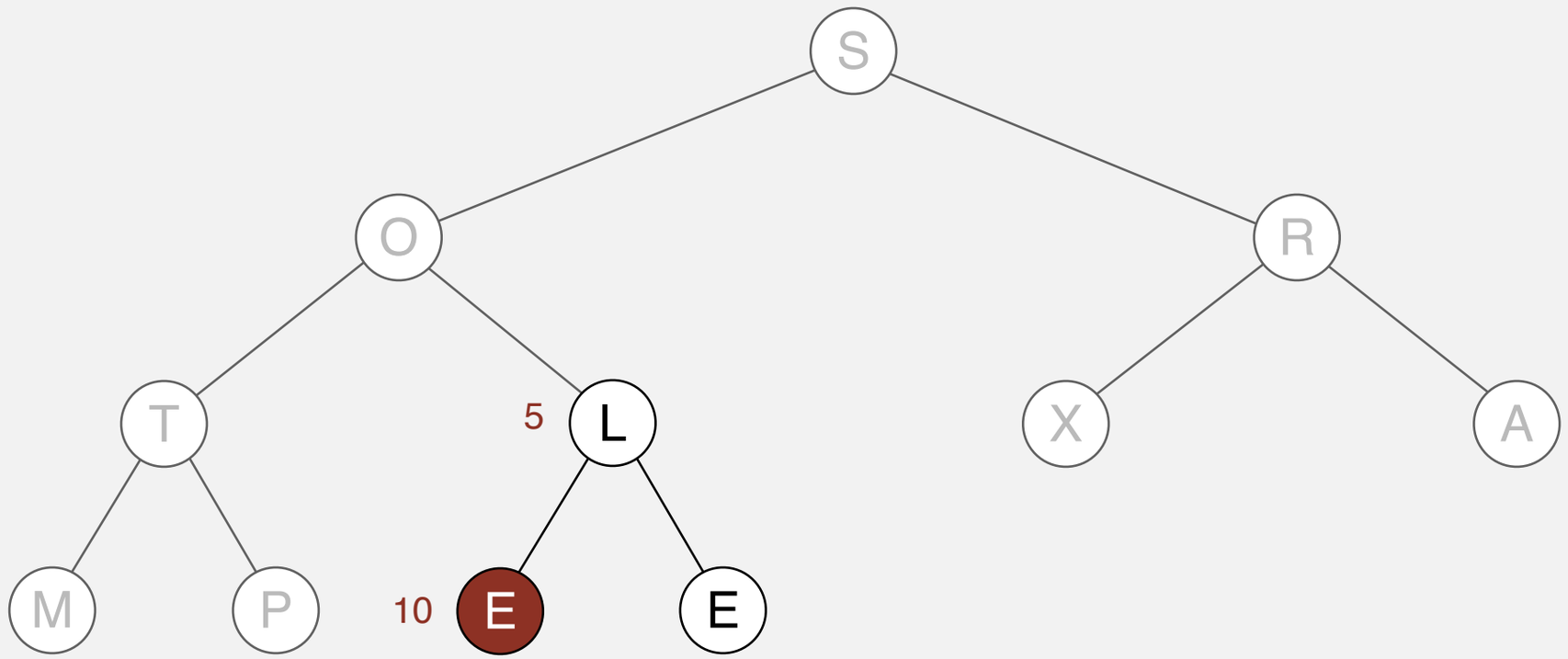


S	O	R	T	E	X	A	M	P	L	E
				5						

Heapsort

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

sink 5

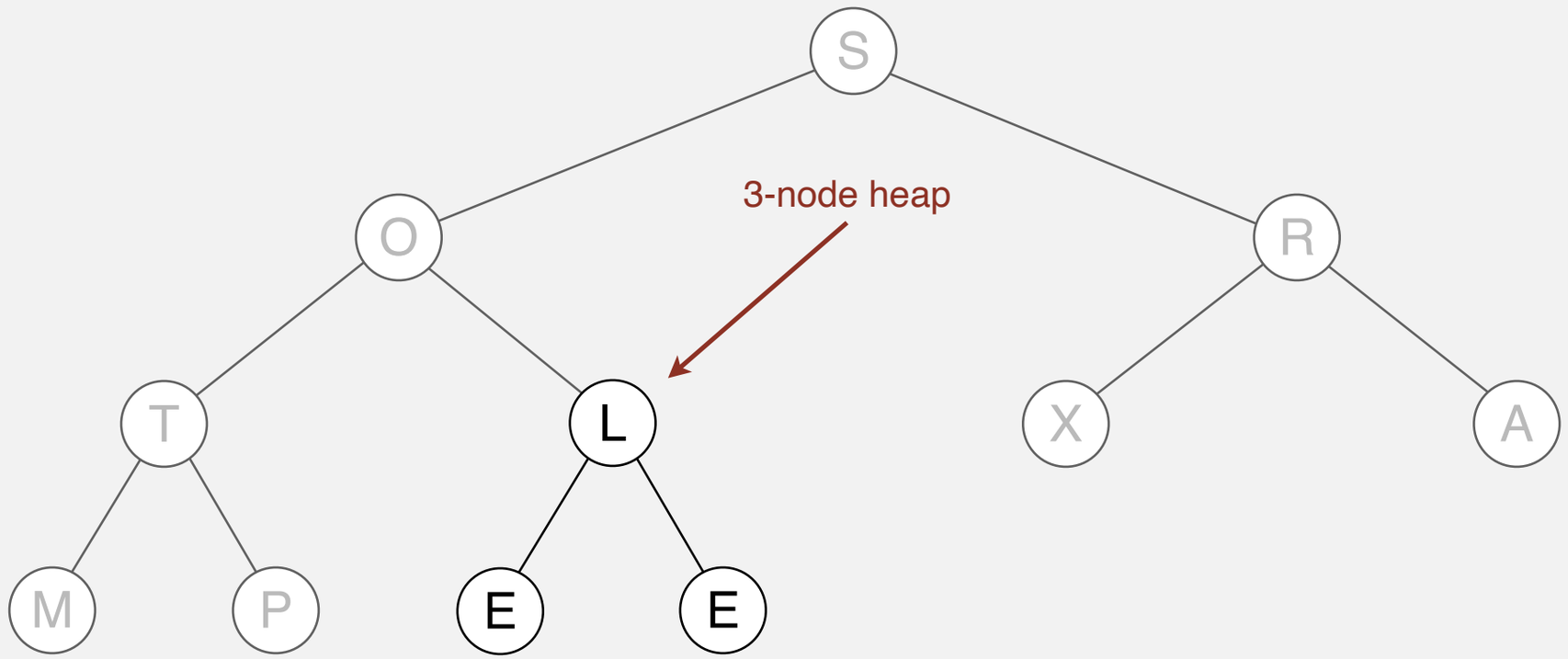


S	O	R	T	L	X	A	M	P	E	E
				5					10	

Heapsort

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

sink 5

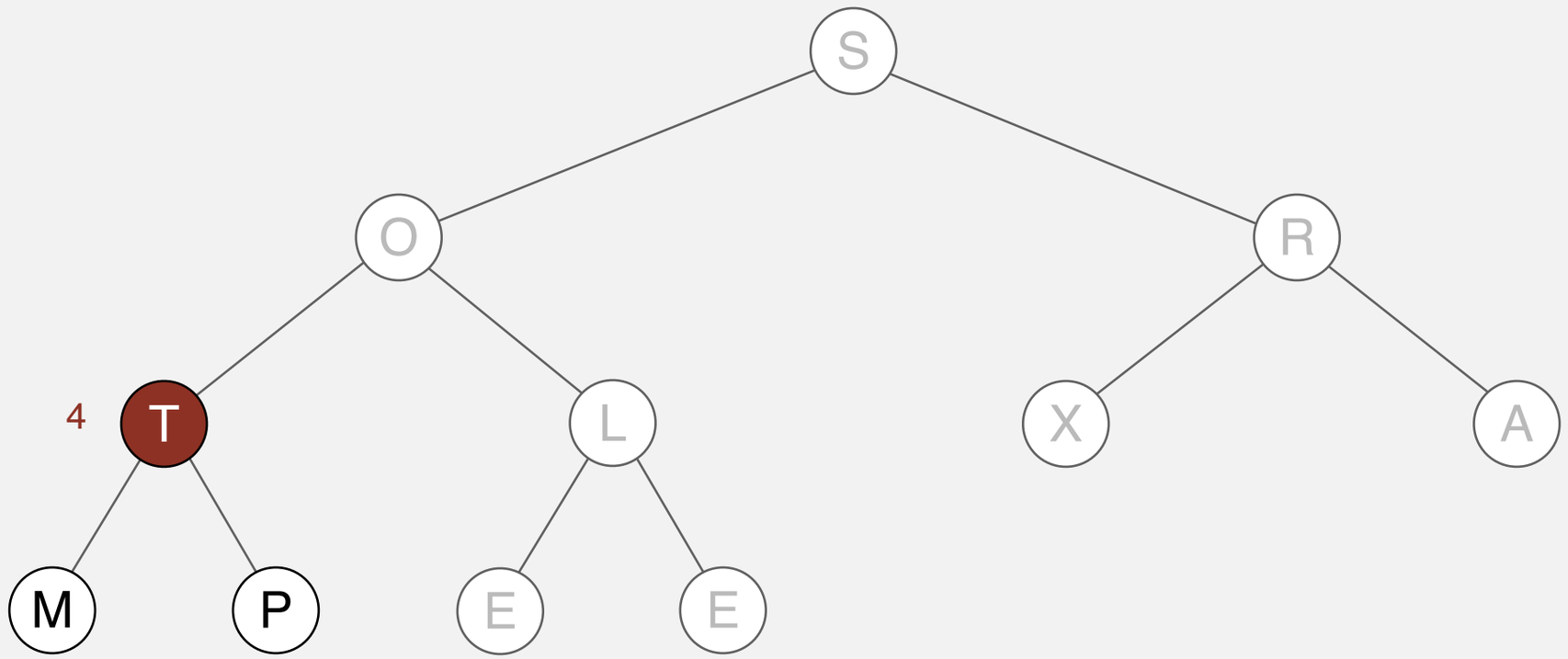


S	O	R	T	L	X	A	M	P	E	E

Heapsort

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

sink 4

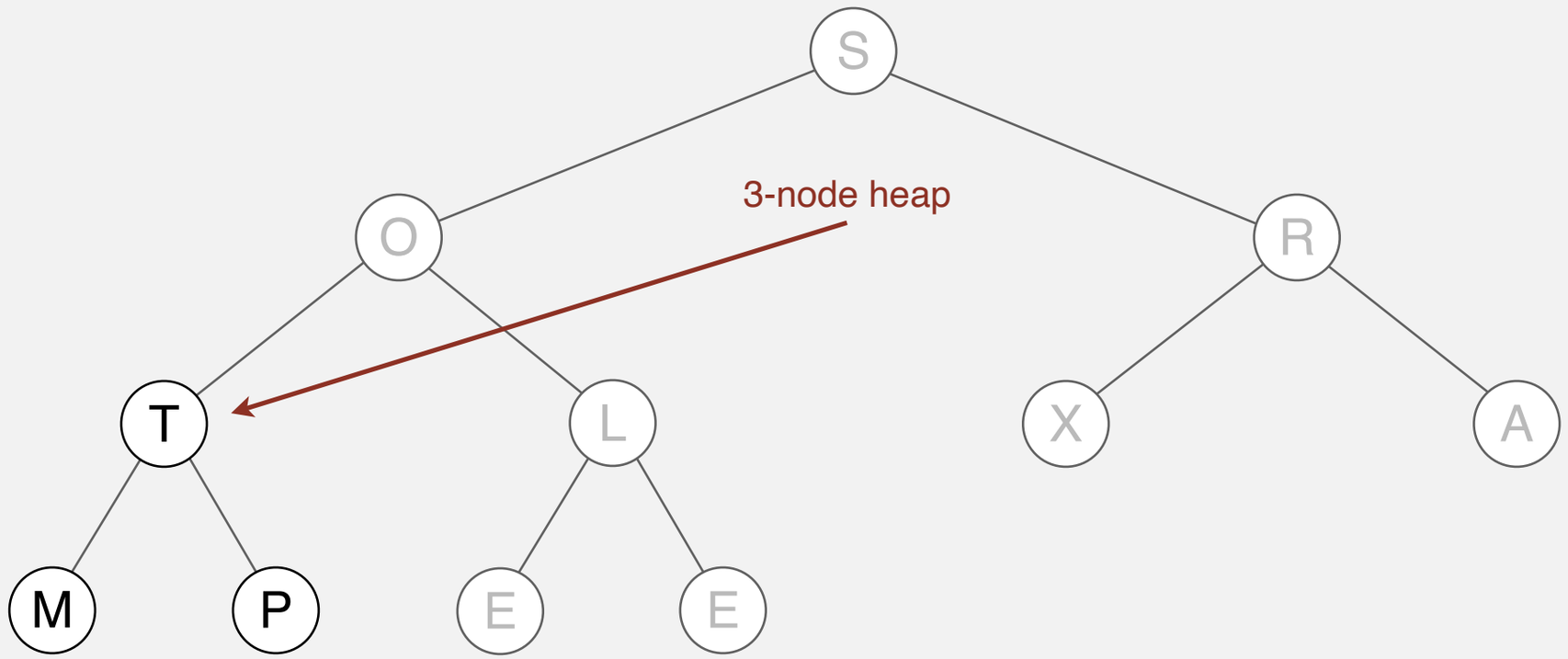


S	O	R	T	L	X	A	M	P	E	E
			4							

Heapsort

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

sink 4

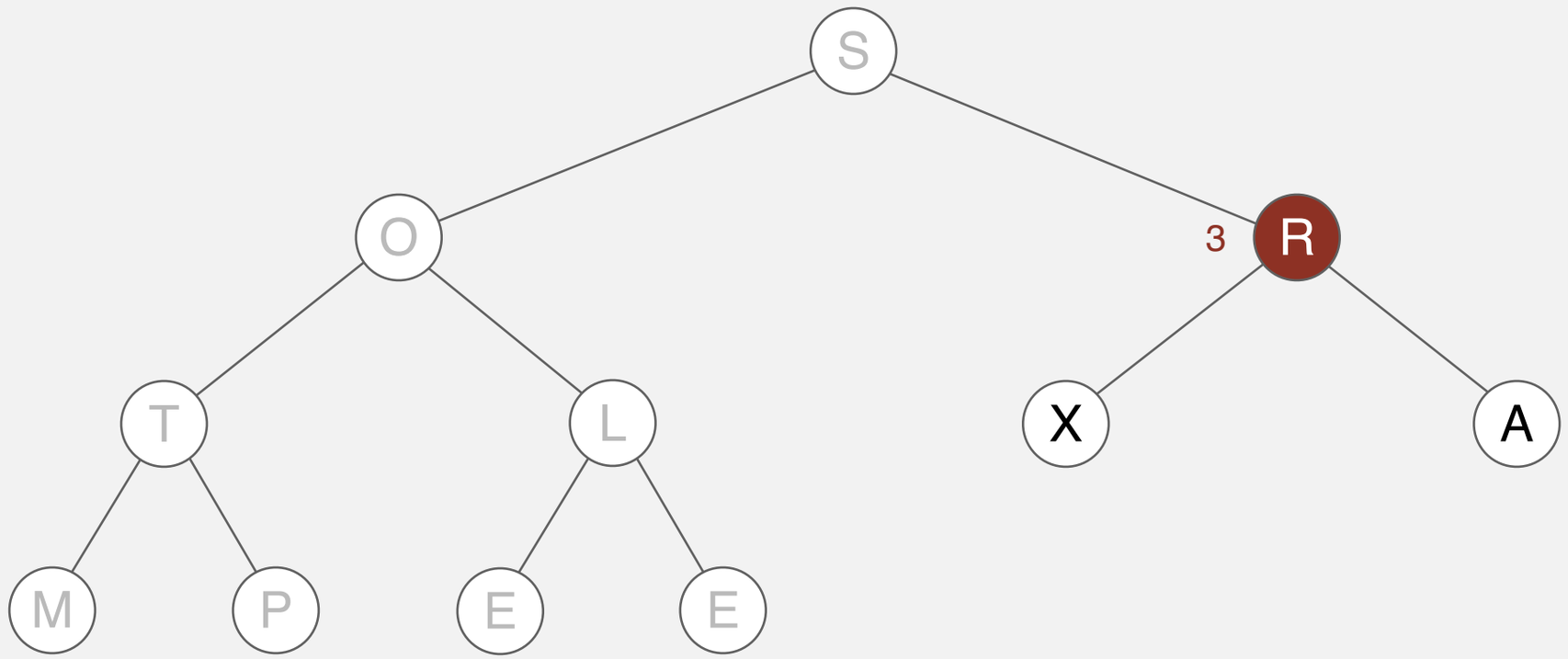


S	O	R	T	L	X	A	M	P	E	E

Heapsort

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

sink 3

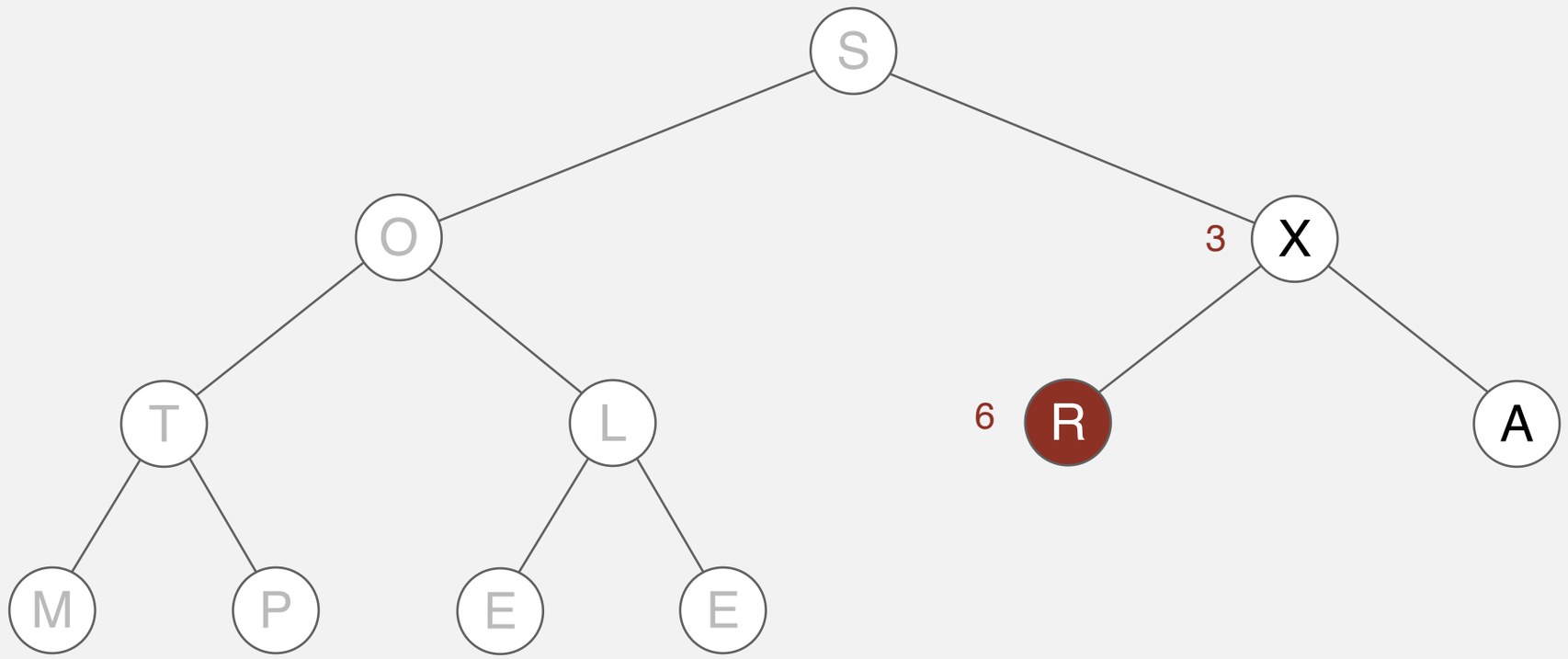


S	O	R	T	L	X	A	M	P	E	E
		3								

Heapsort

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

sink 3

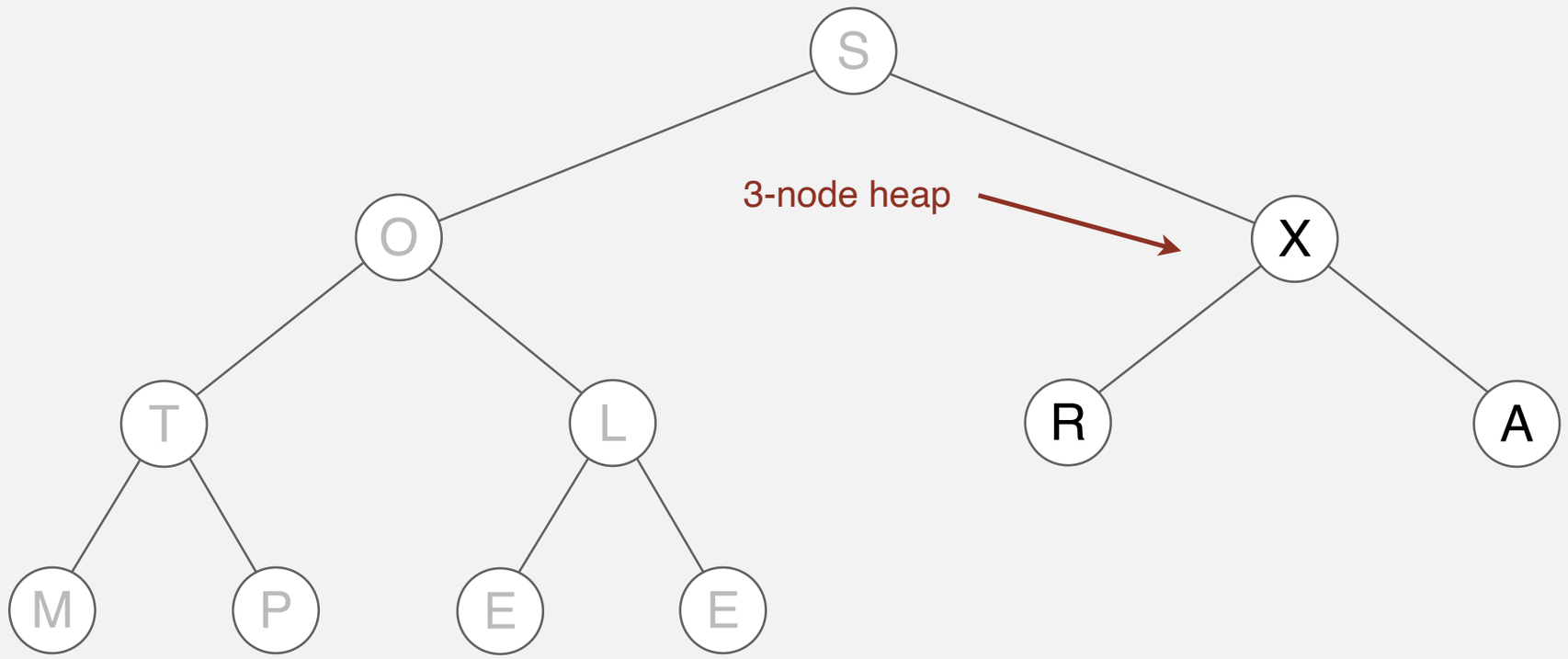


S	O	X	T	L	R	A	M	P	E	E
		3			6					

Heapsort

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

sink 3

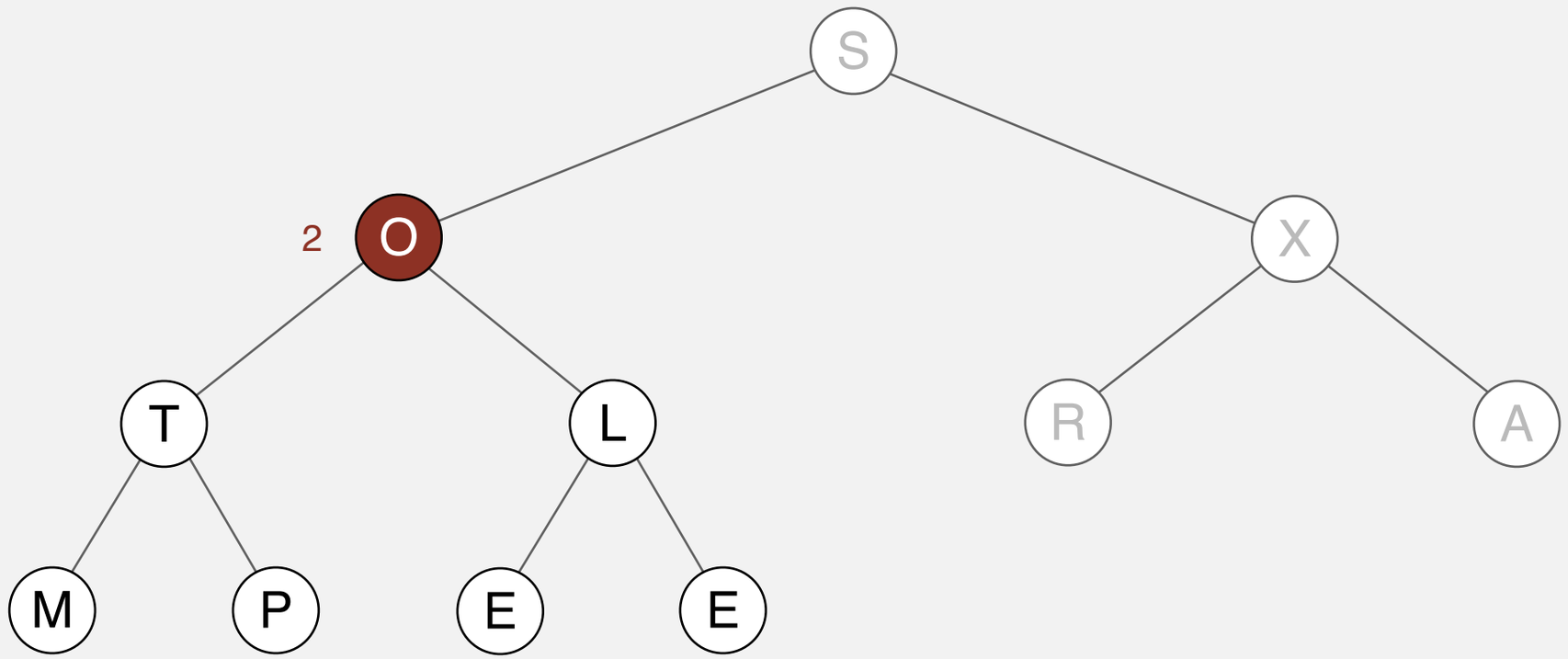


S	O	X	T	L	A	A	M	P	E	E

Heapsort

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

sink 2

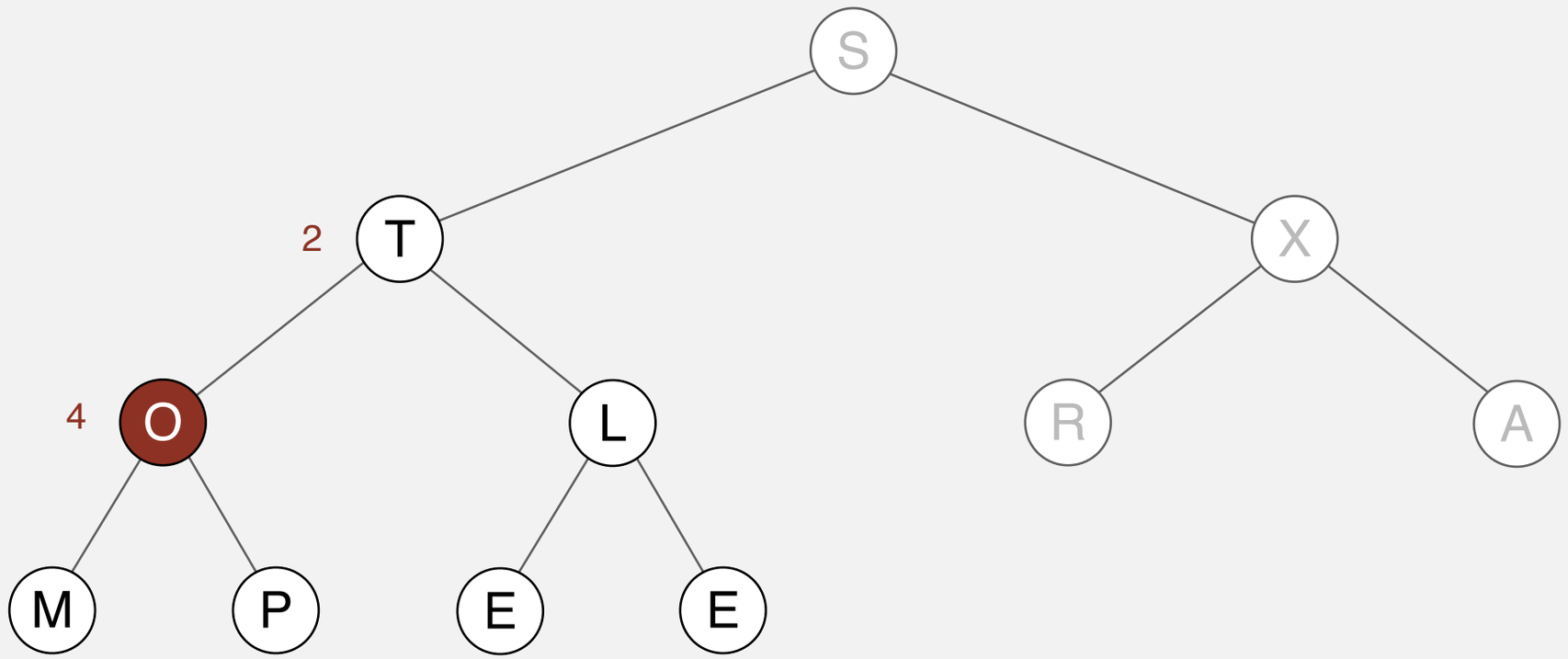


S	O	X	T	L	R	A	M	P	E	E
	2									

Heapsort

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

sink 2

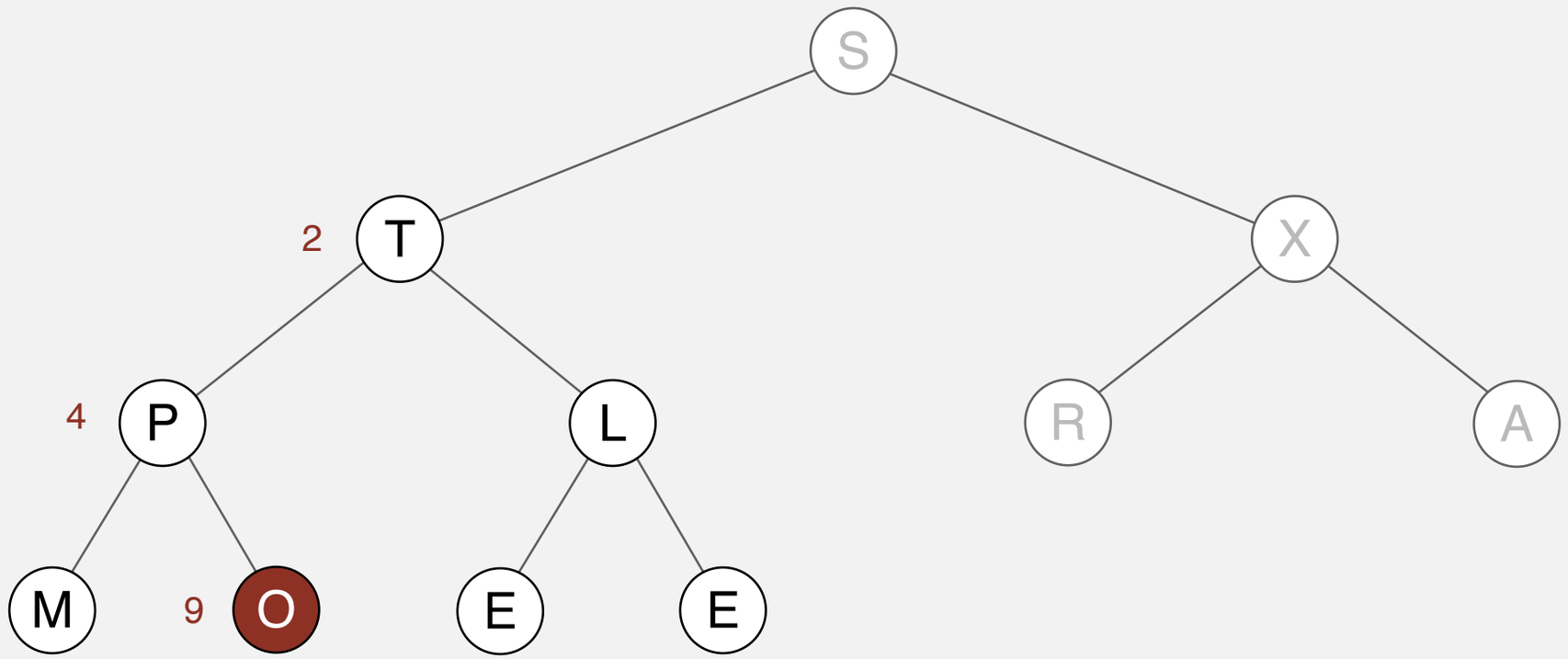


S	T	X	O	L	R	A	M	P	E	E
	2		4							

Heapsort

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

sink 2

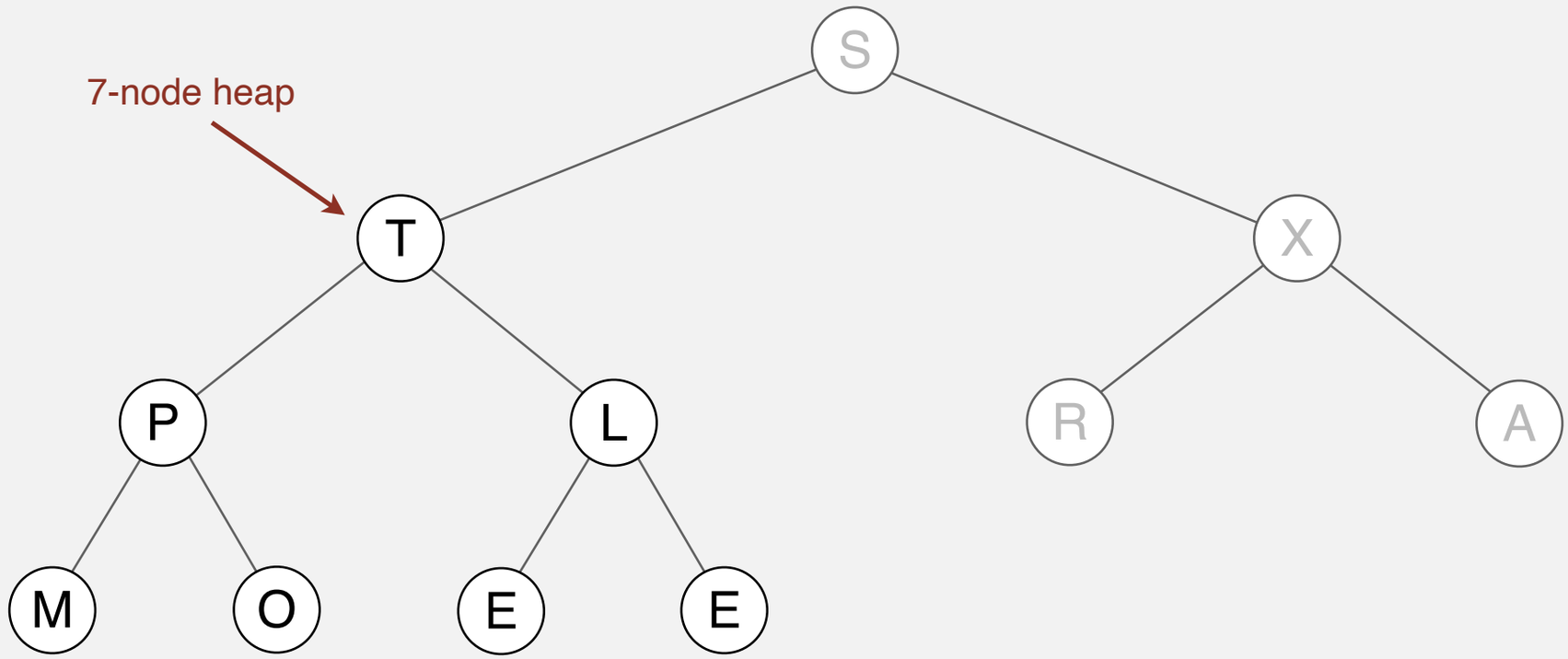


S	T	X	P	L	R	A	M	O	E	E
	2		4					9		

Heapsort

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

sink 2

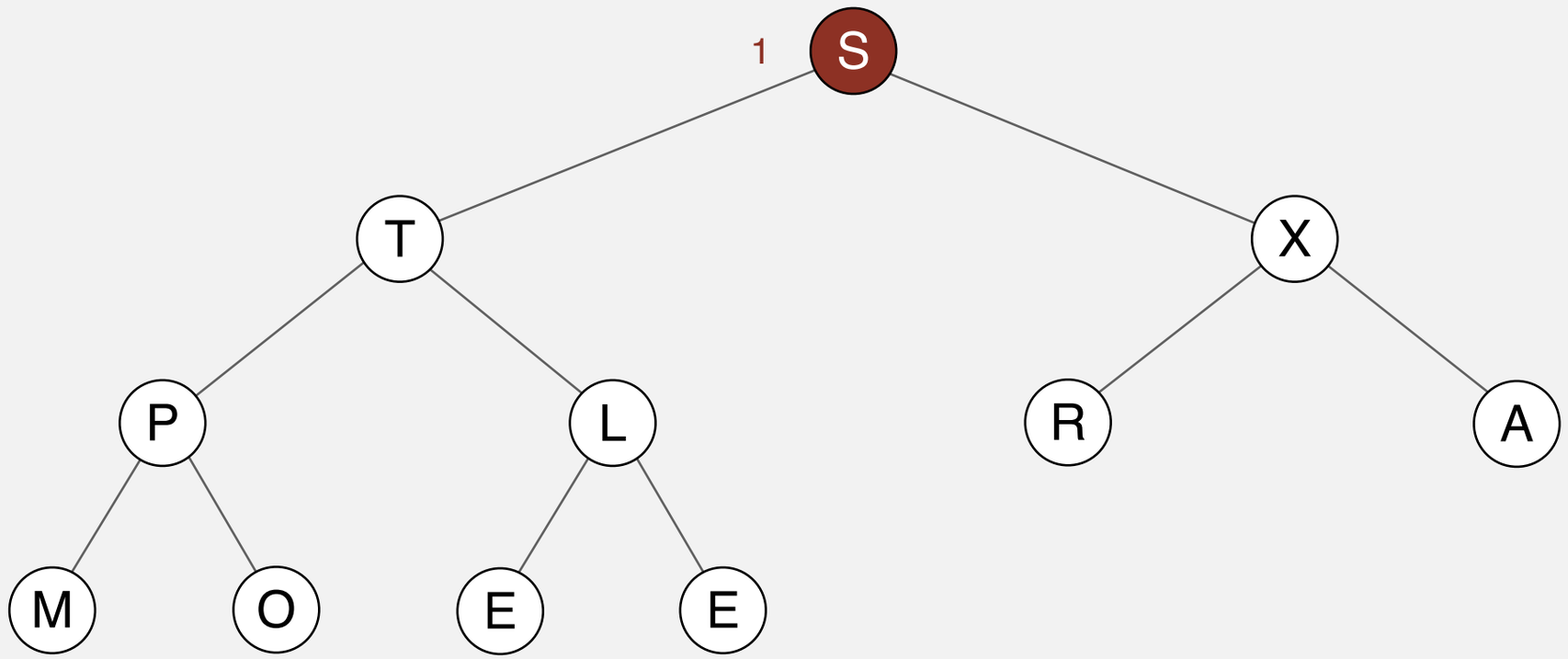


S	T	X	P	L	R	A	M	O	E	E

Heapsort

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

sink 1

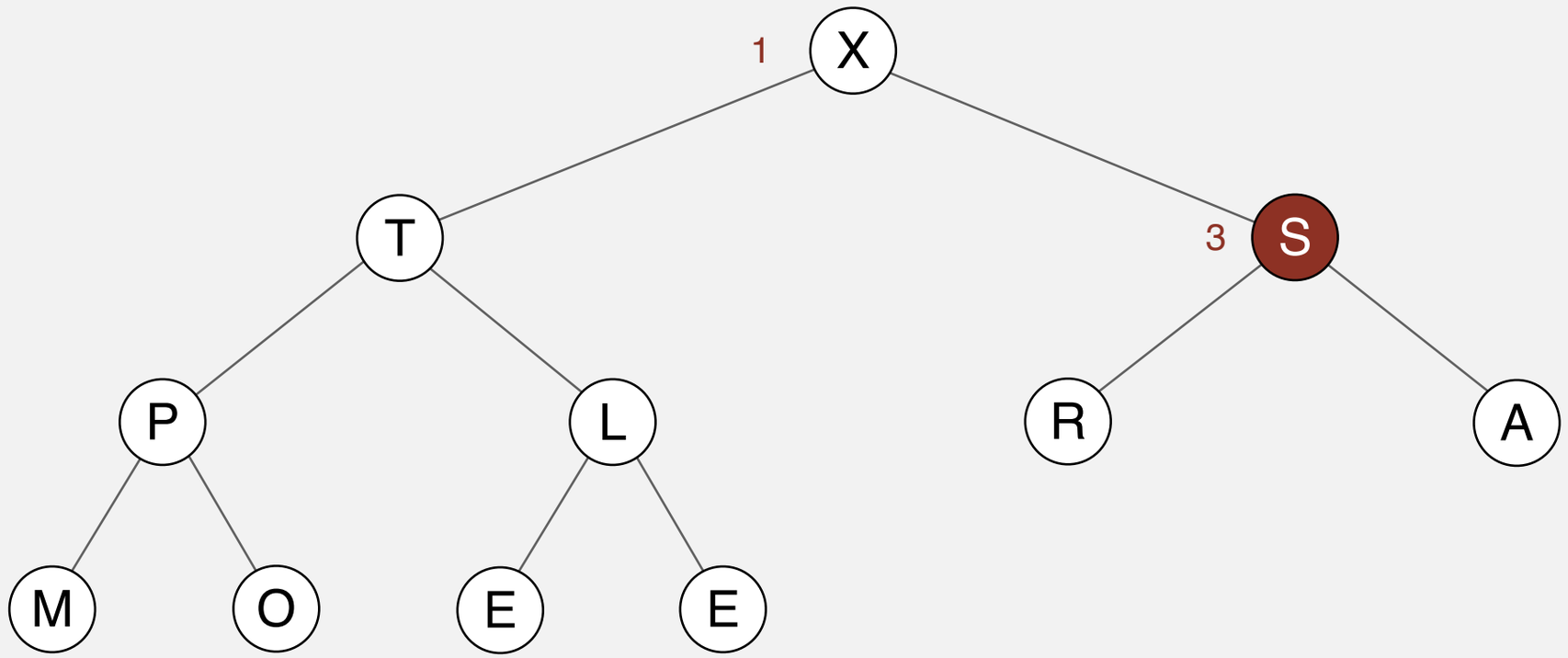


S	T	X	P	L	R	A	M	O	E	E
1										

Heapsort

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

sink 1

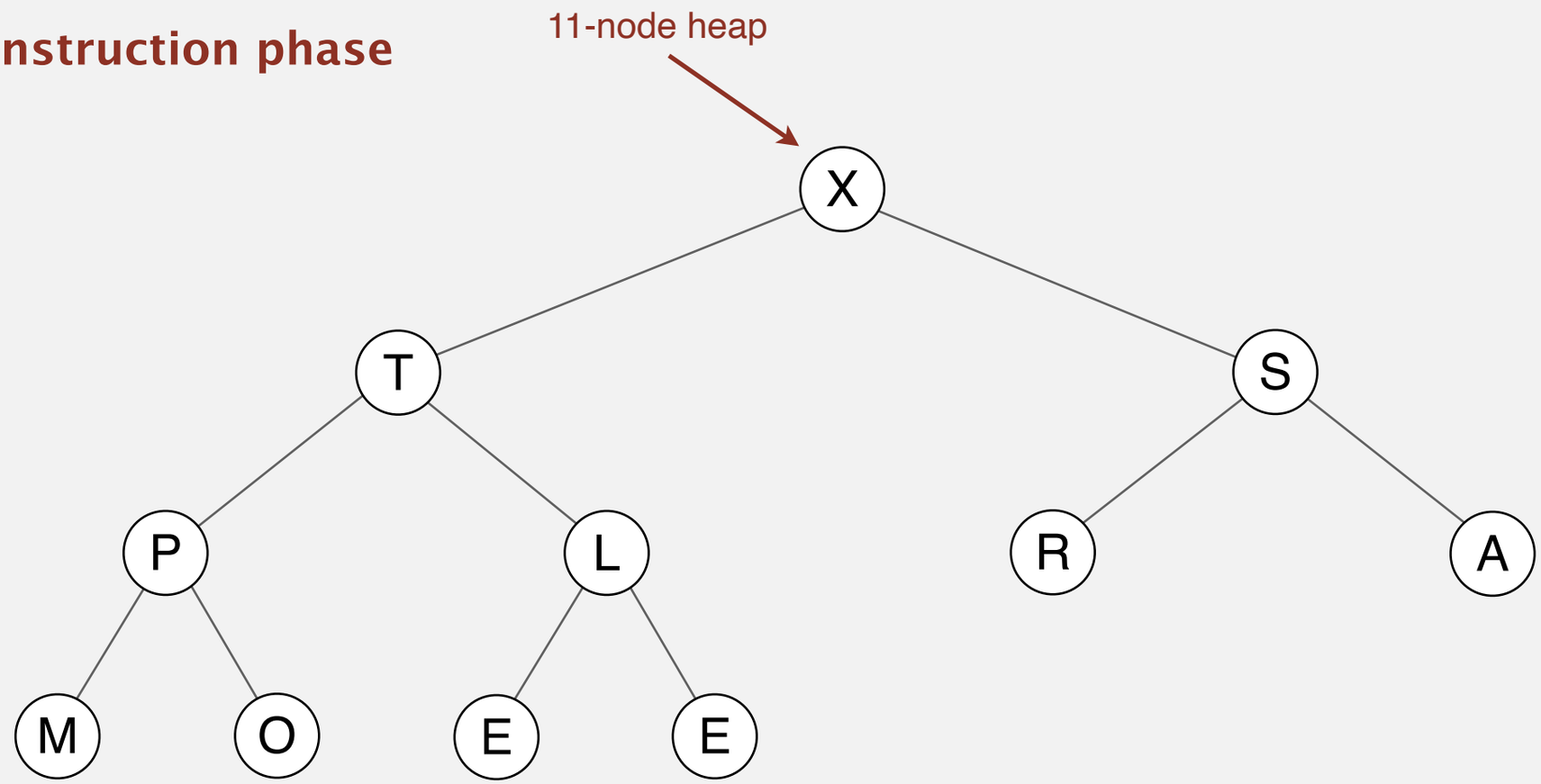


X	T	S	P	L	R	A	M	O	E	E
1		3								

Heapsort

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

end of construction phase

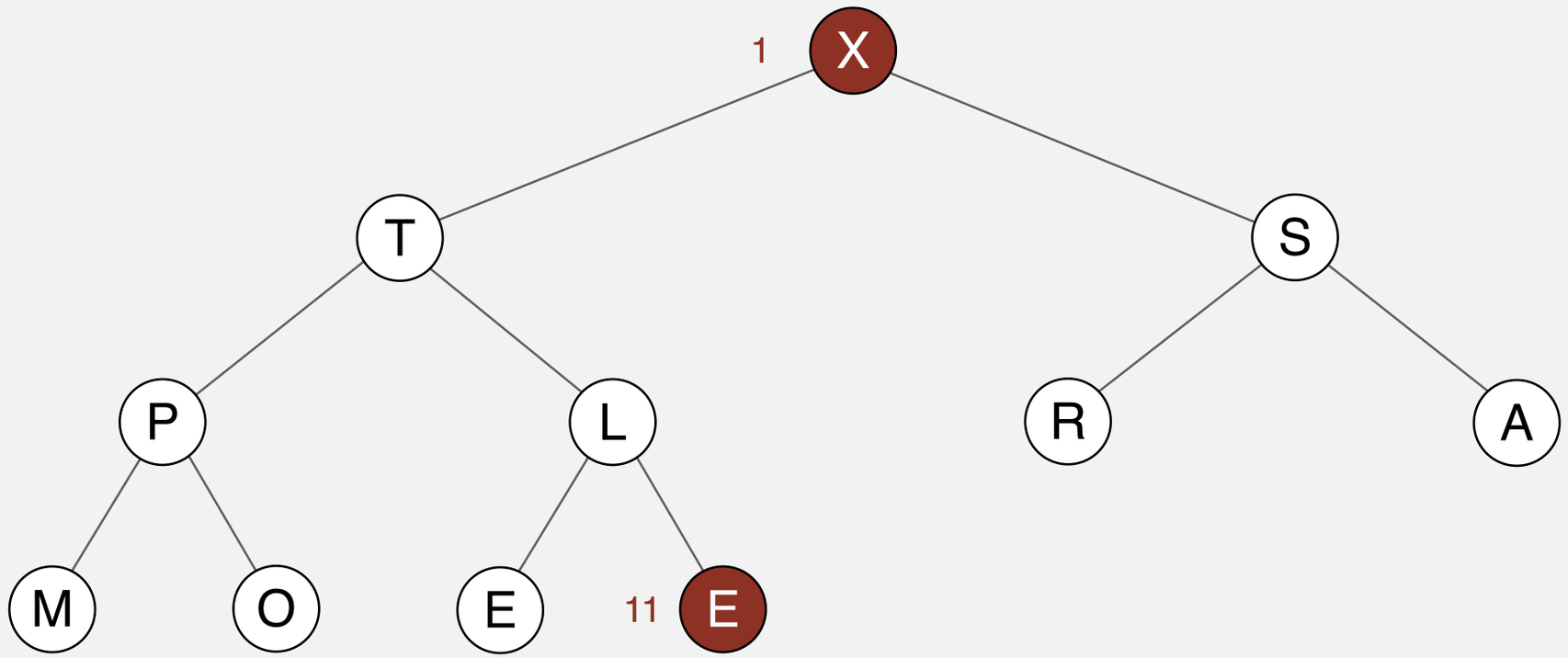


X	T	S	P	L	R	A	M	O	E	E

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

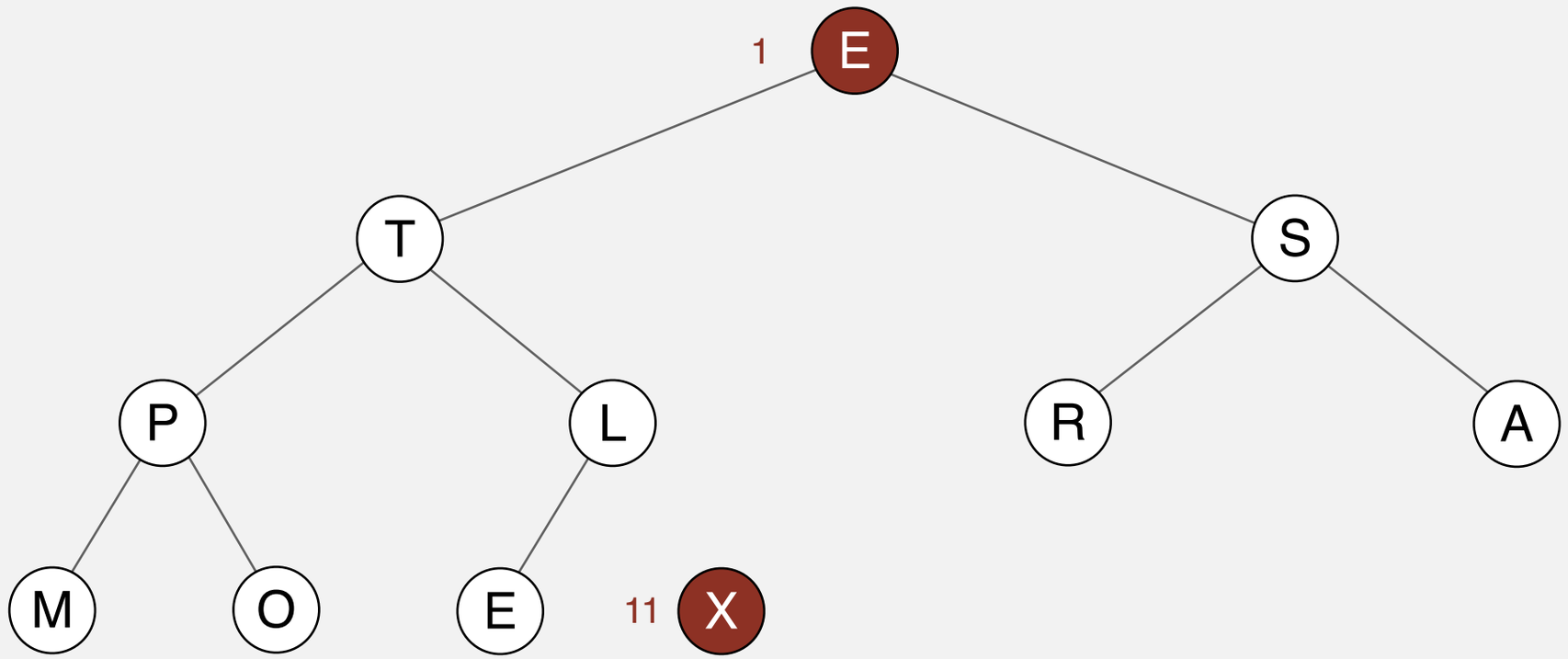


X	T	S	P	L	R	A	M	O	E	E
1										11

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

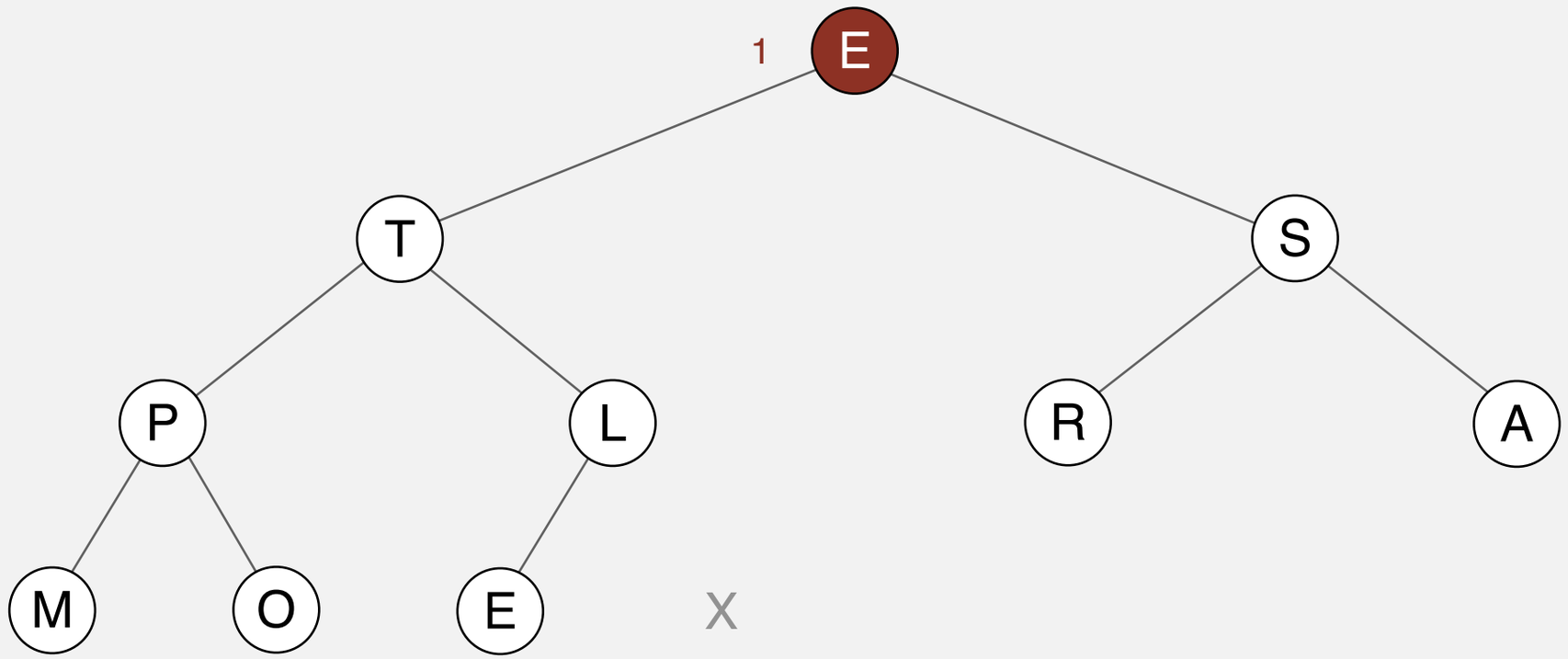


E	T	S	P	L	R	A	M	O	E	X
1										11

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

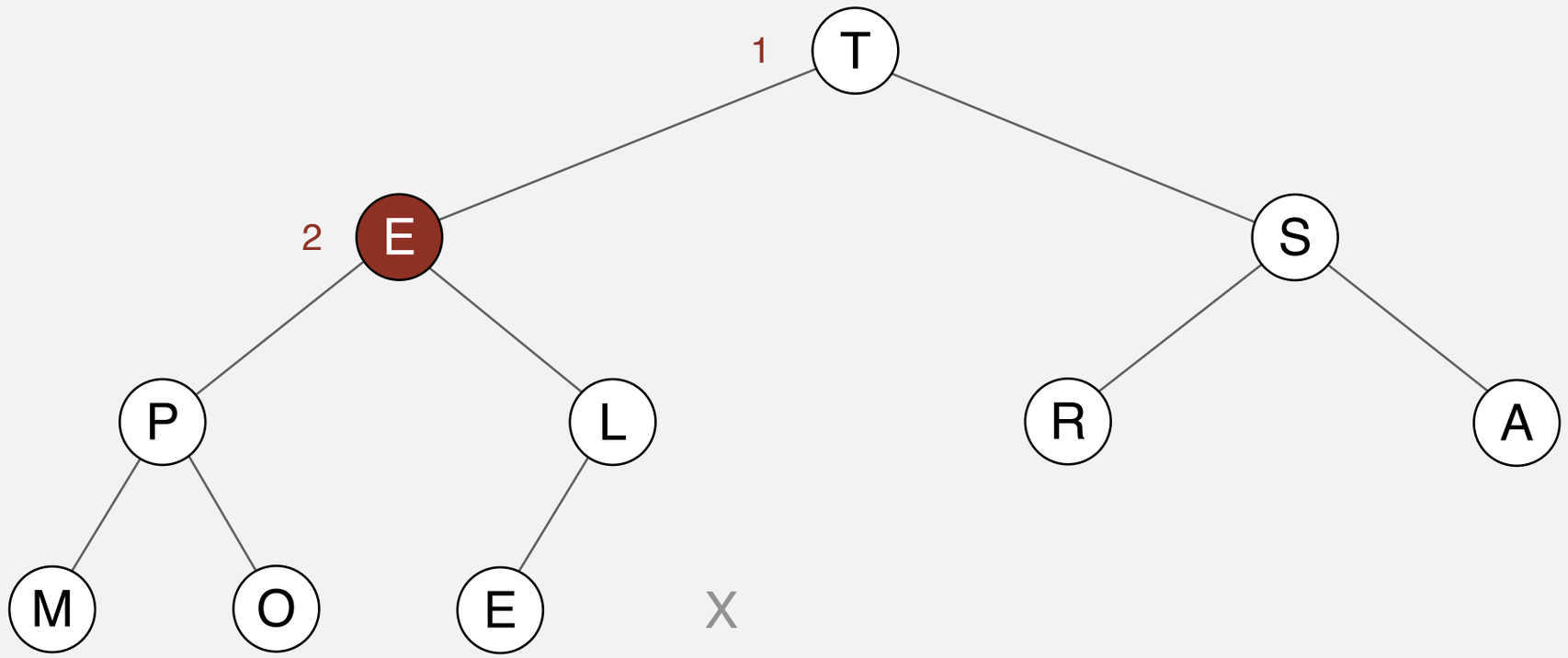


E	T	S	P	L	R	A	M	O	E	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

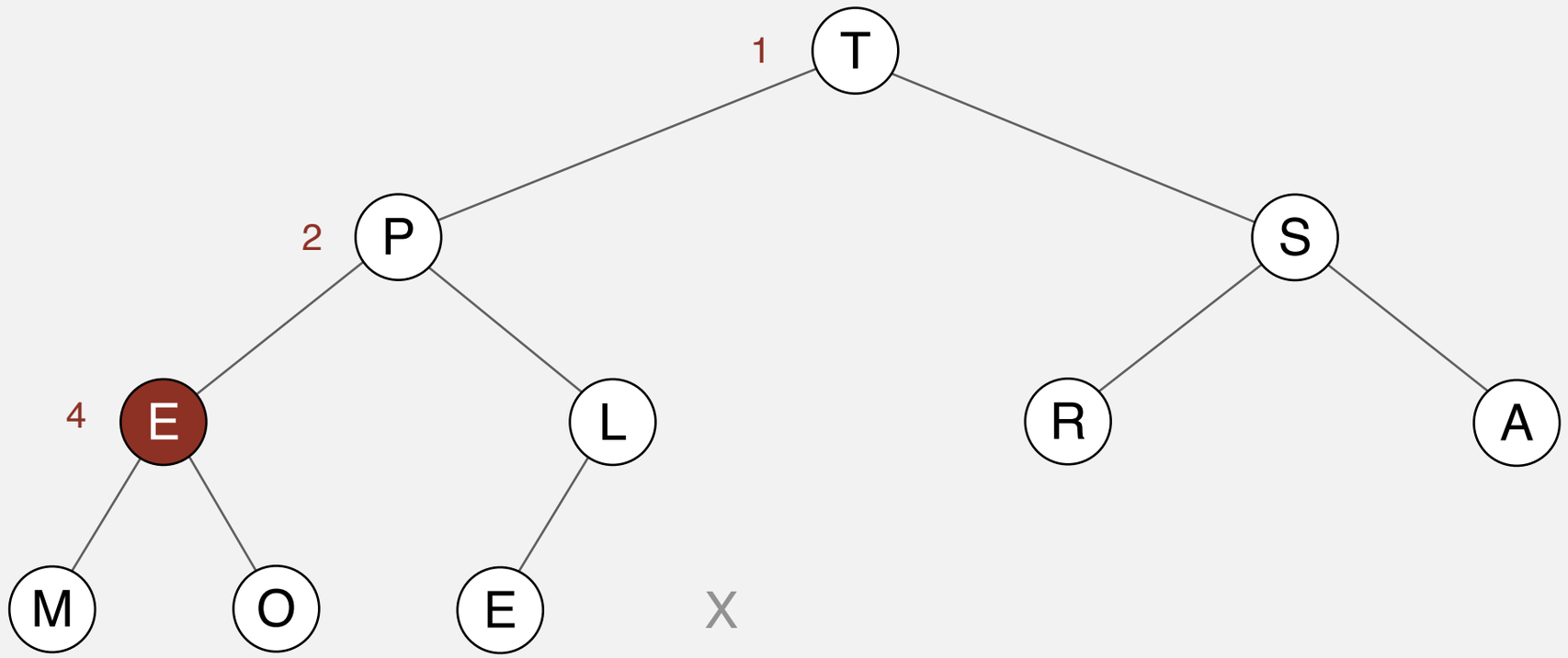


T	E	S	P	L	R	A	M	O	E	X
1	2									

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

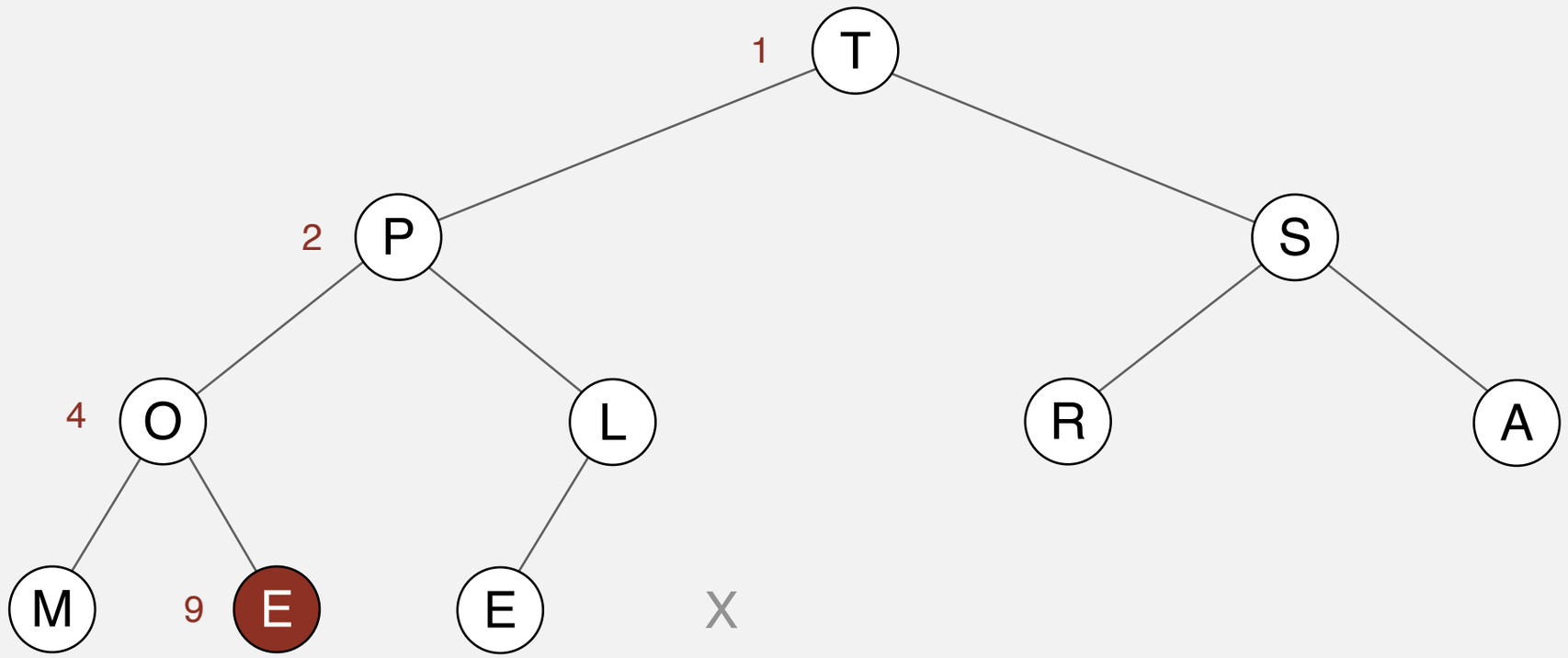


T	P	S	E	L	R	A	M	O	E	X
1	2		4							

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

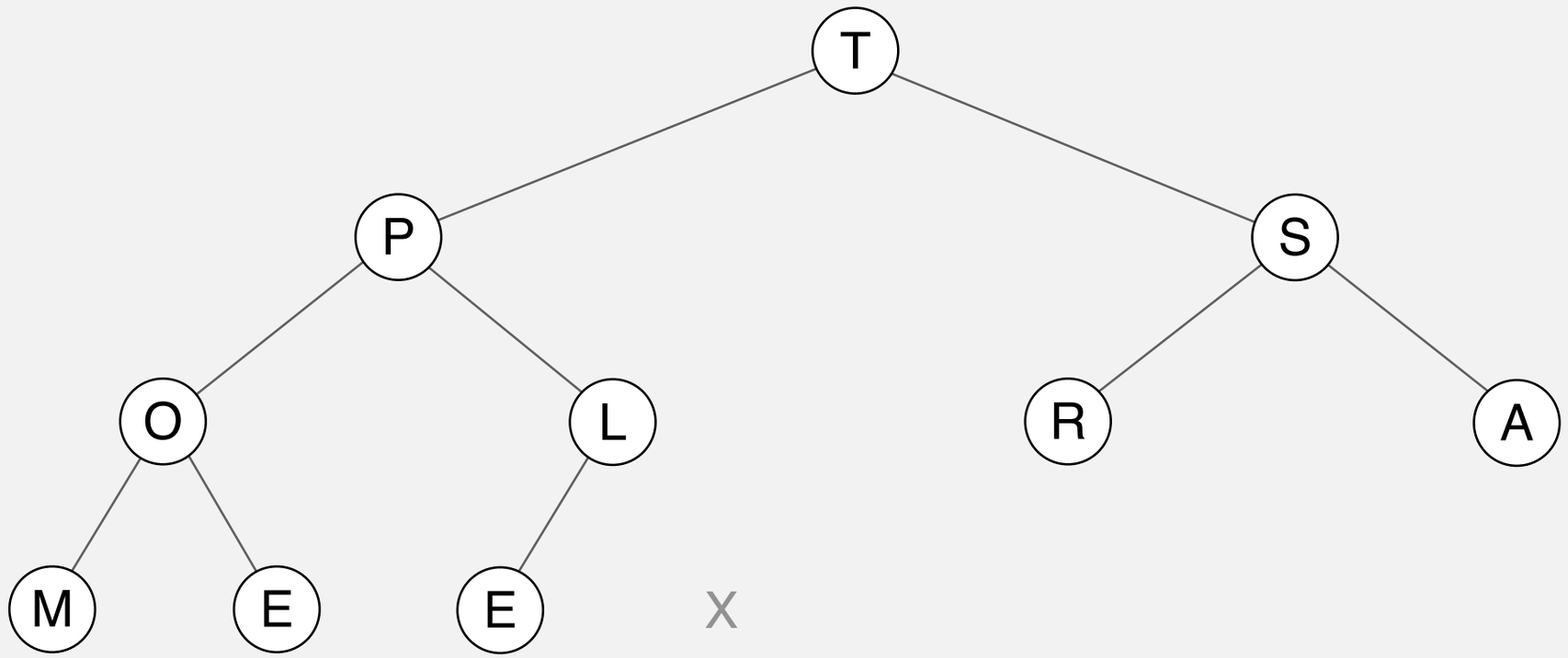
sink 1



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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

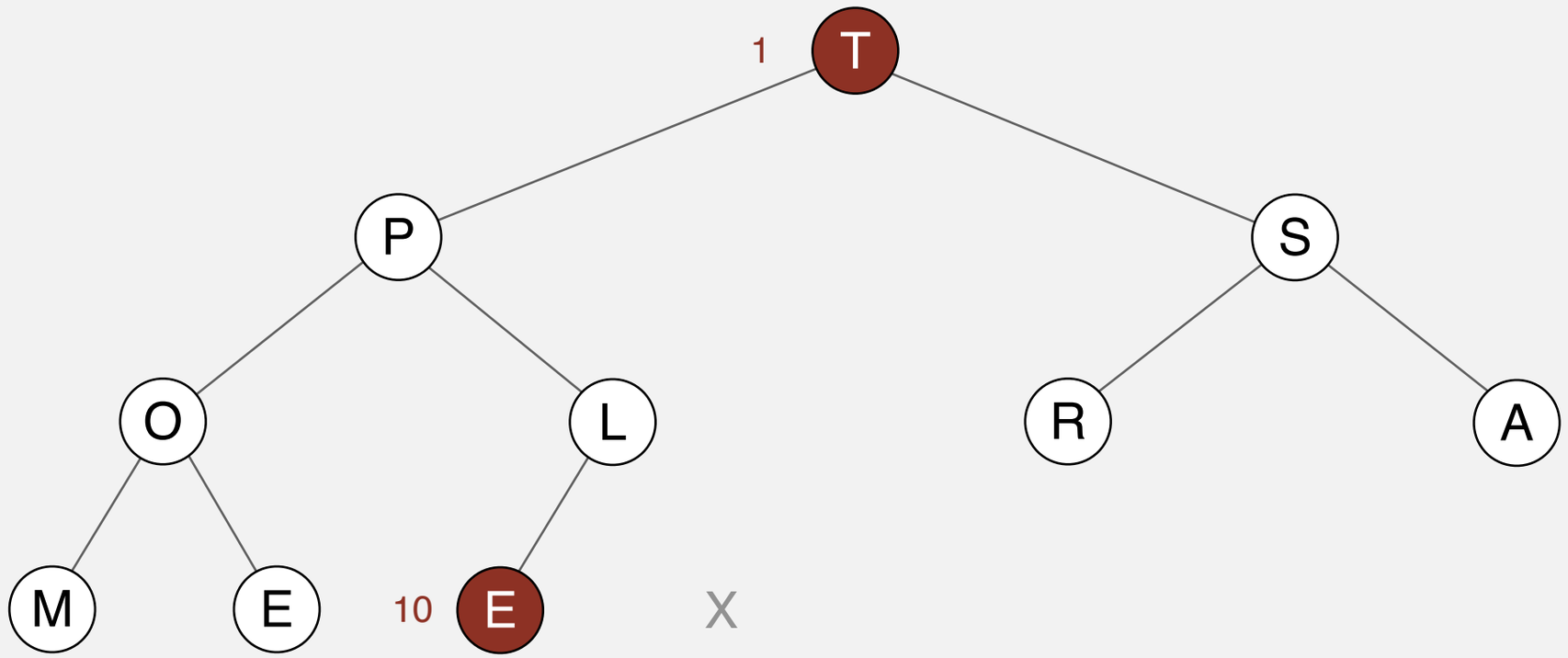


T	P	S	O	L	R	A	M	E	E	X

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 10

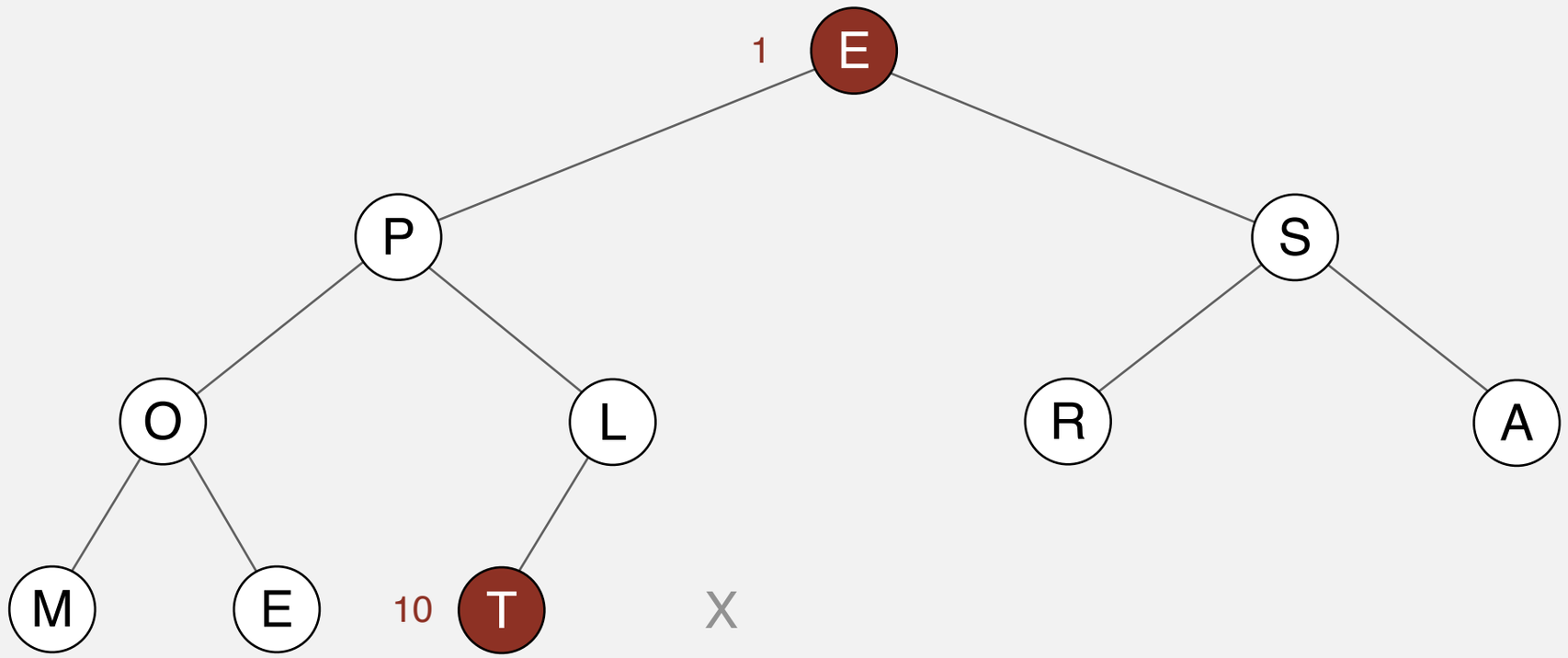


T	P	S	O	L	R	A	M	E	E	X
1									10	

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 10

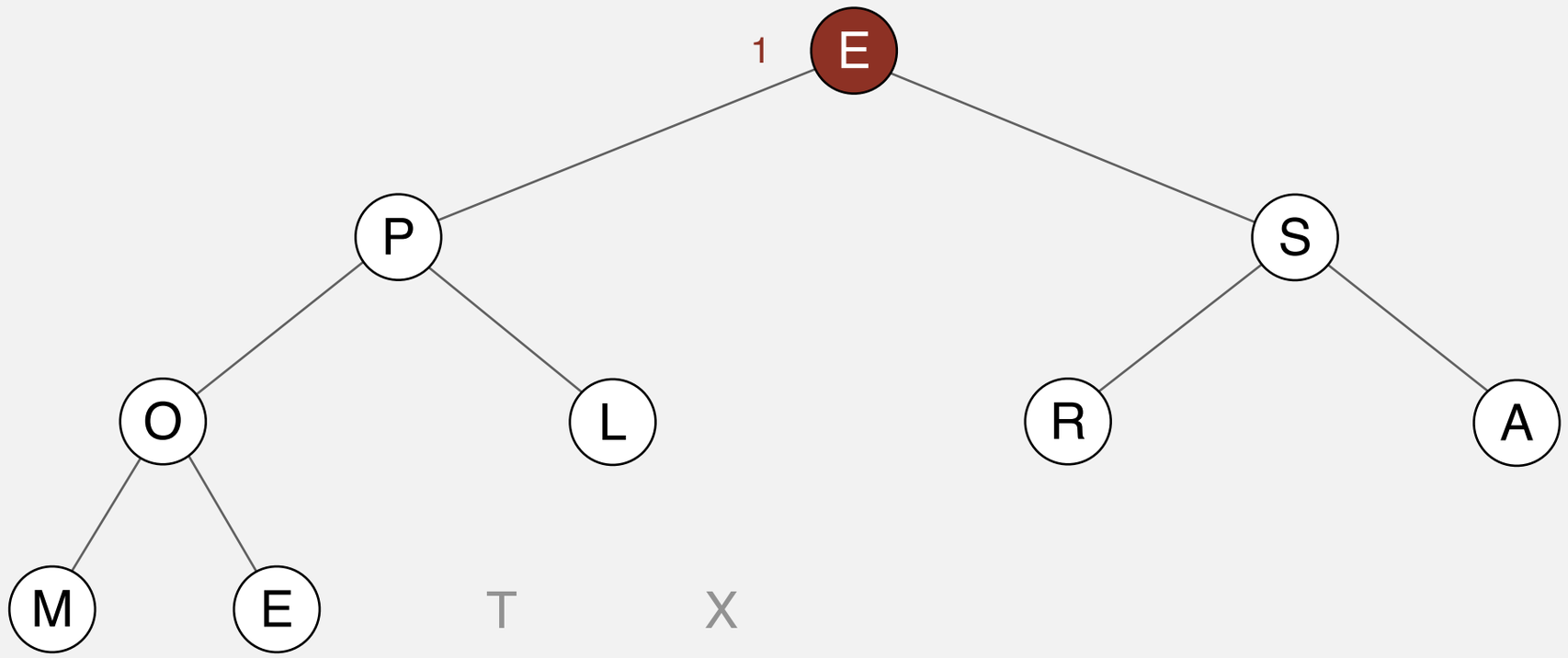


E	P	S	O	L	R	A	M	E	T	X
1									10	

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

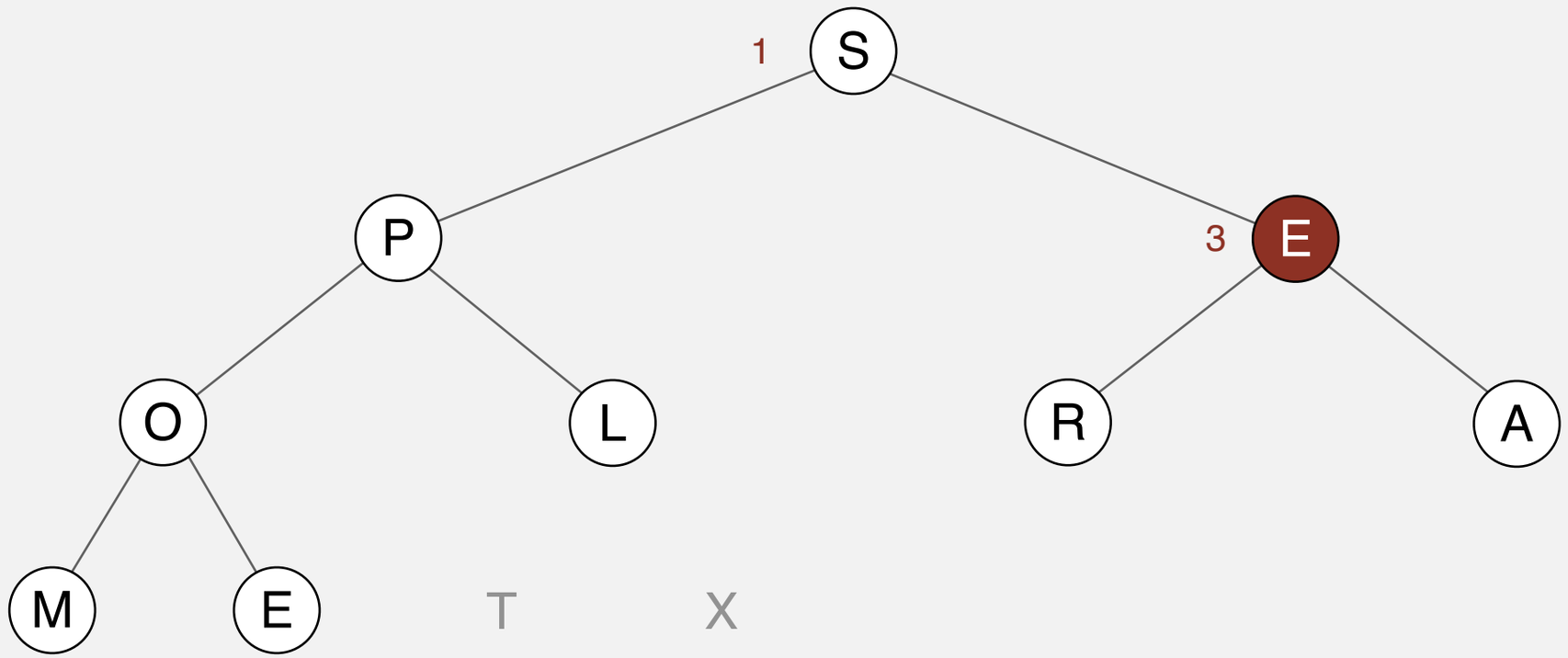


E	P	S	O	L	R	A	M	E	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

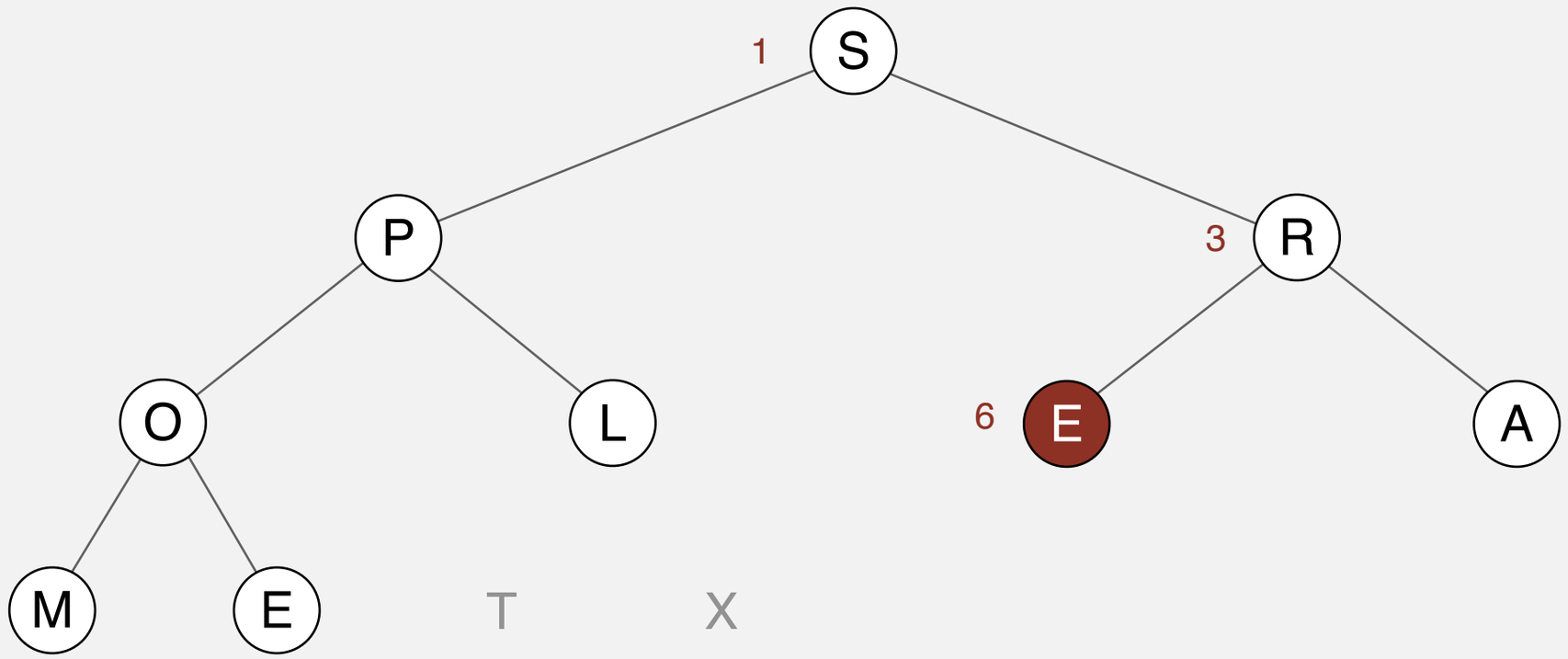


S	P	E	O	L	R	A	M	E	T	X
1		3								

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

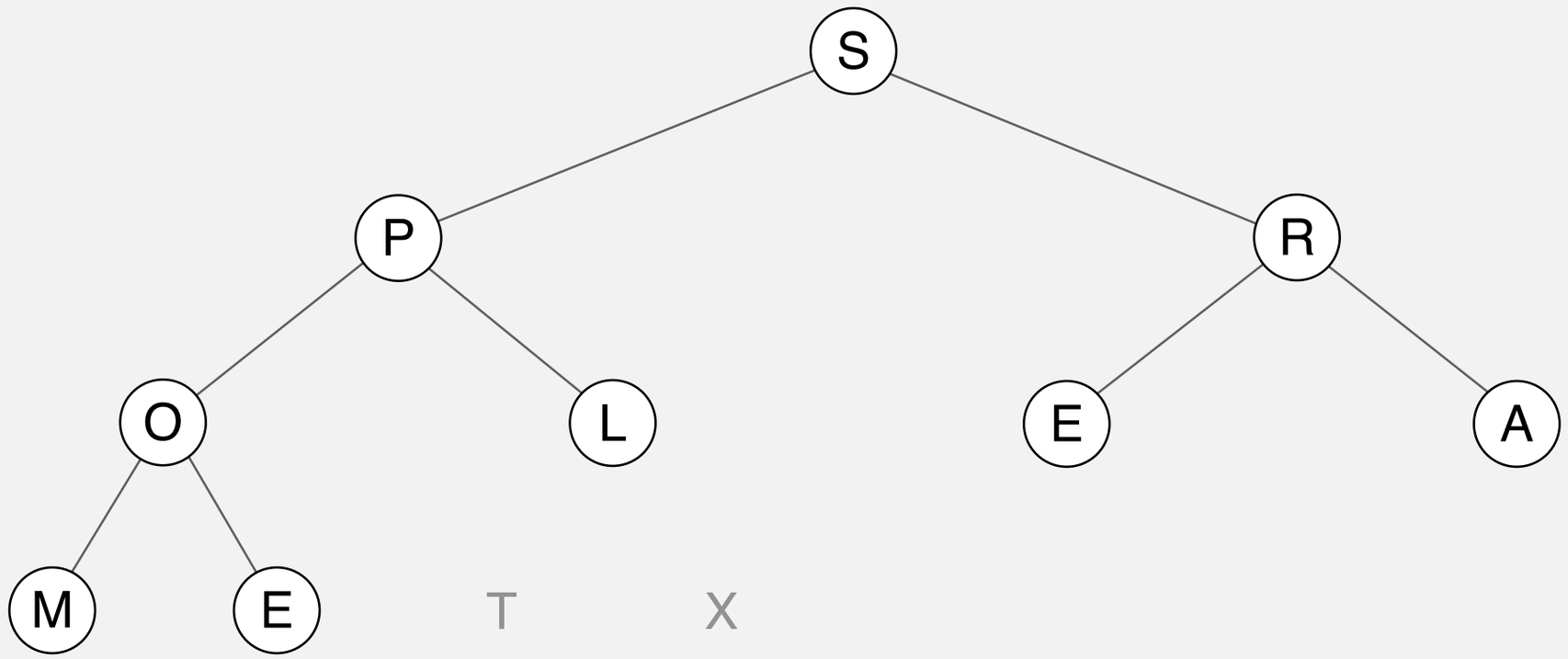
sink 1



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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

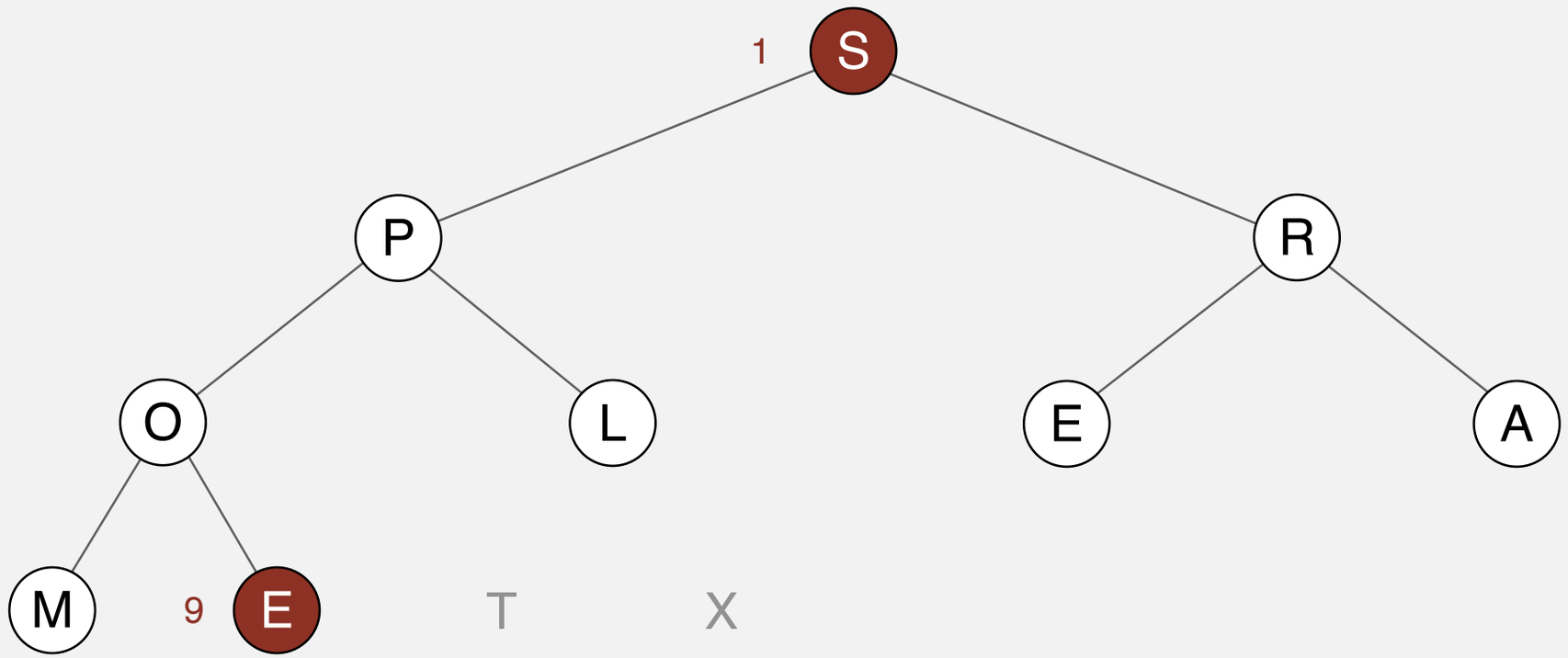


S	P	R	O	L	E	A	M	E	T	X

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

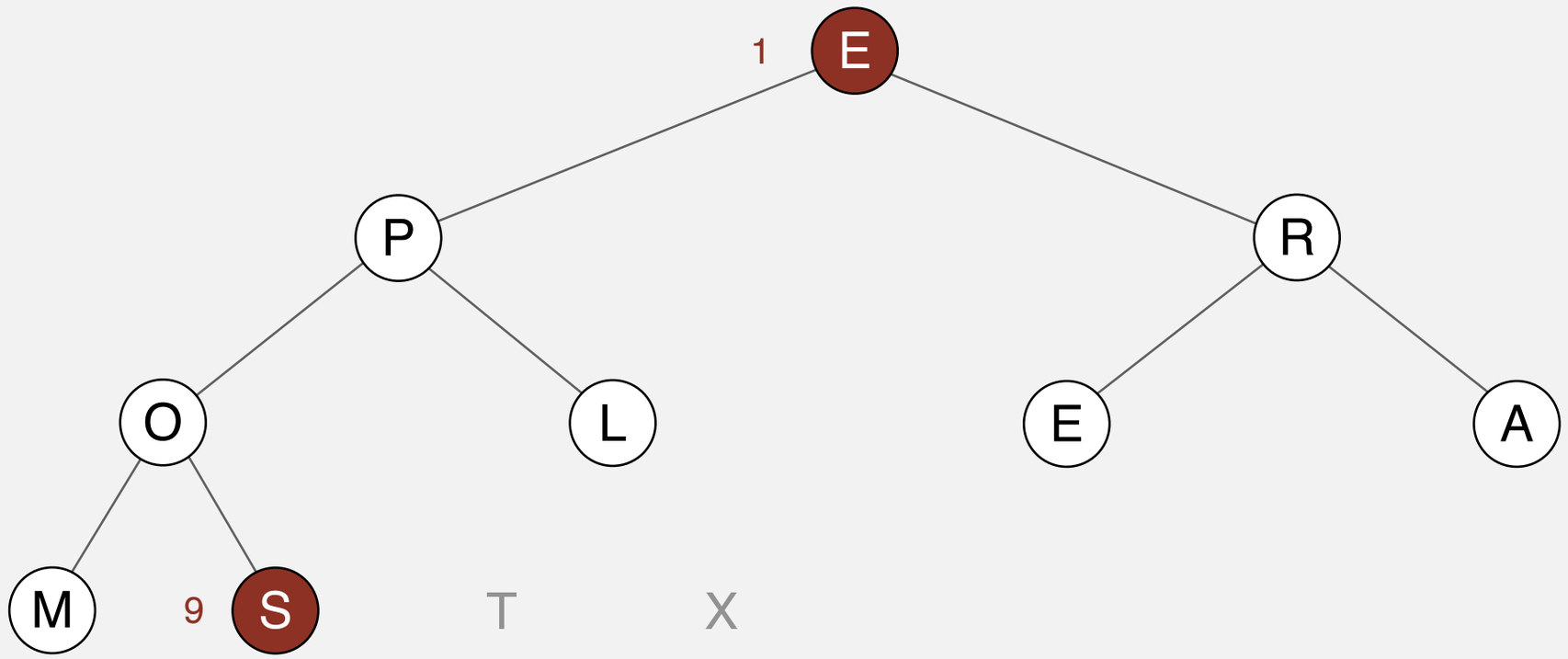


S	P	R	O	L	E	A	M	E	T	X
1								9		

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

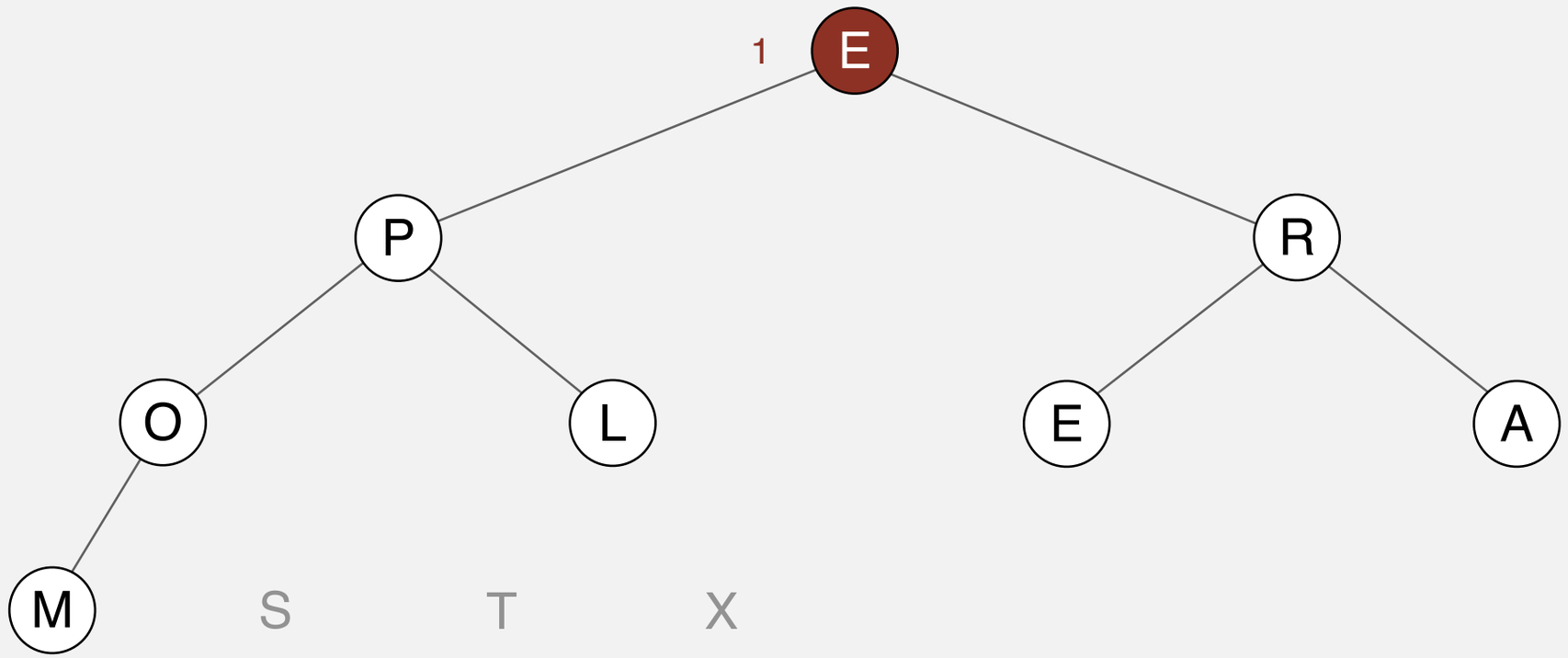


E	P	R	O	L	E	A	M	S	T	X
1								9		

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

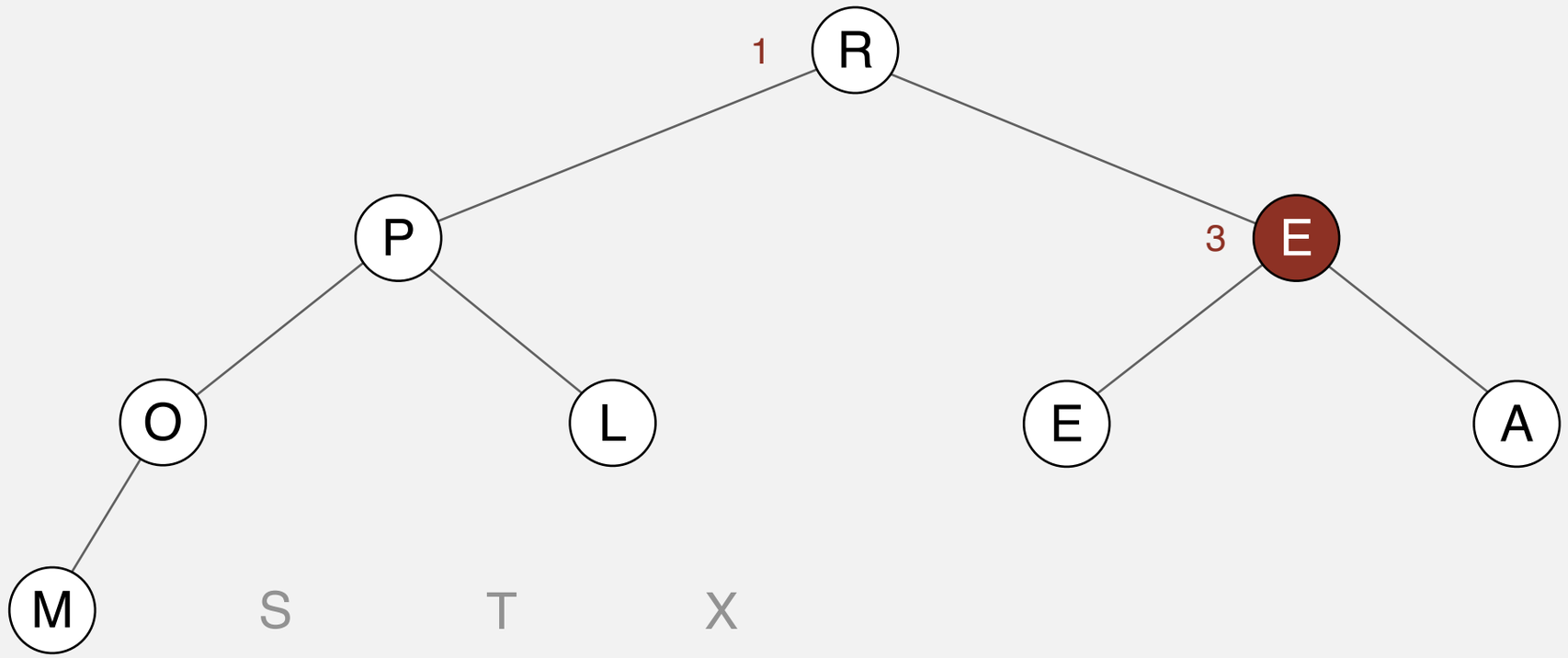


E	P	R	O	L	E	A	M	S	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

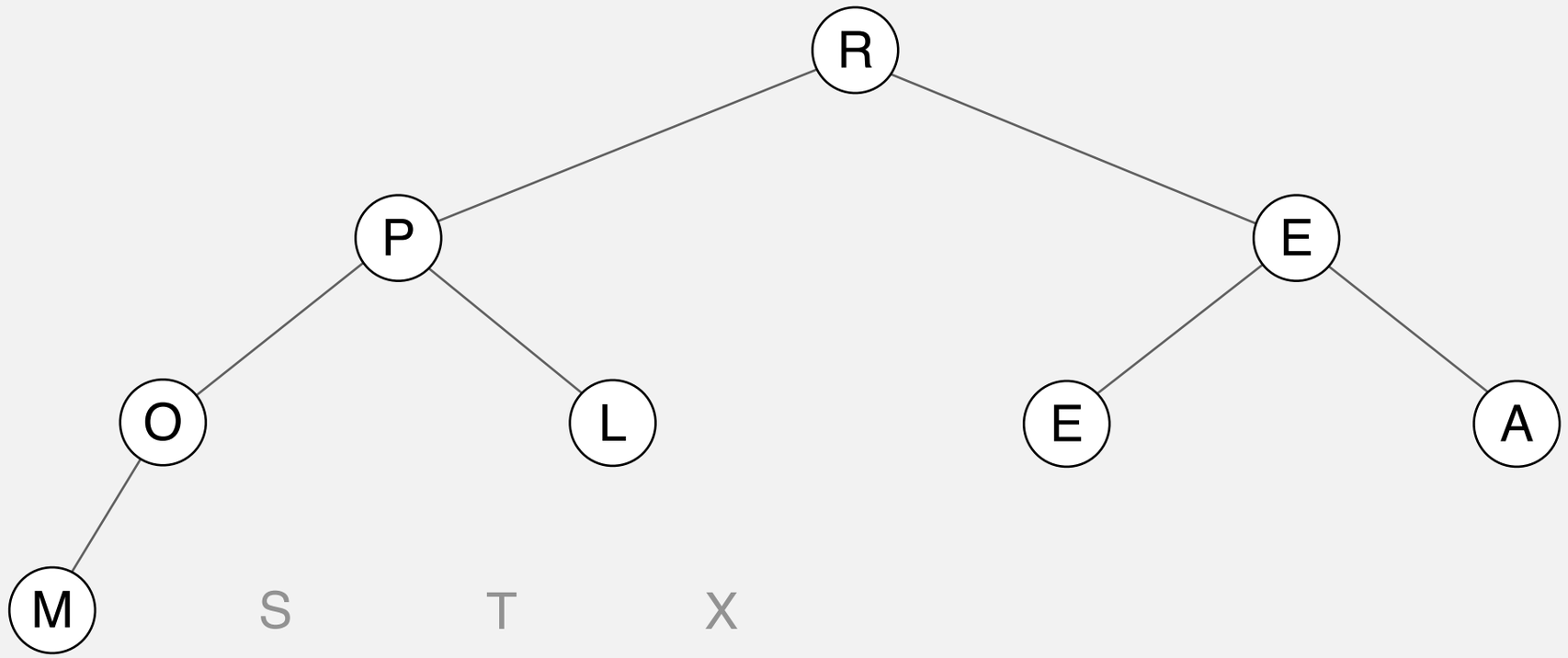
sink 1



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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

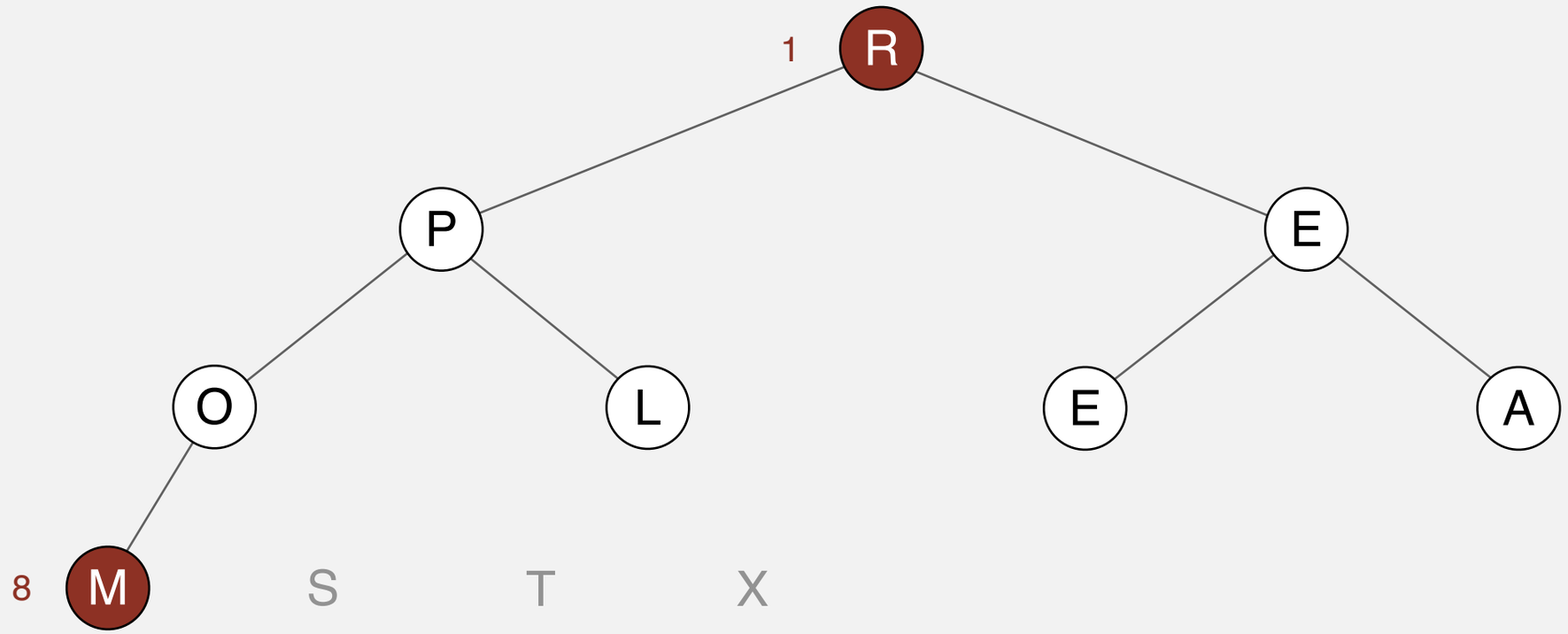


R	P	E	O	L	E	A	M	S	T	X

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

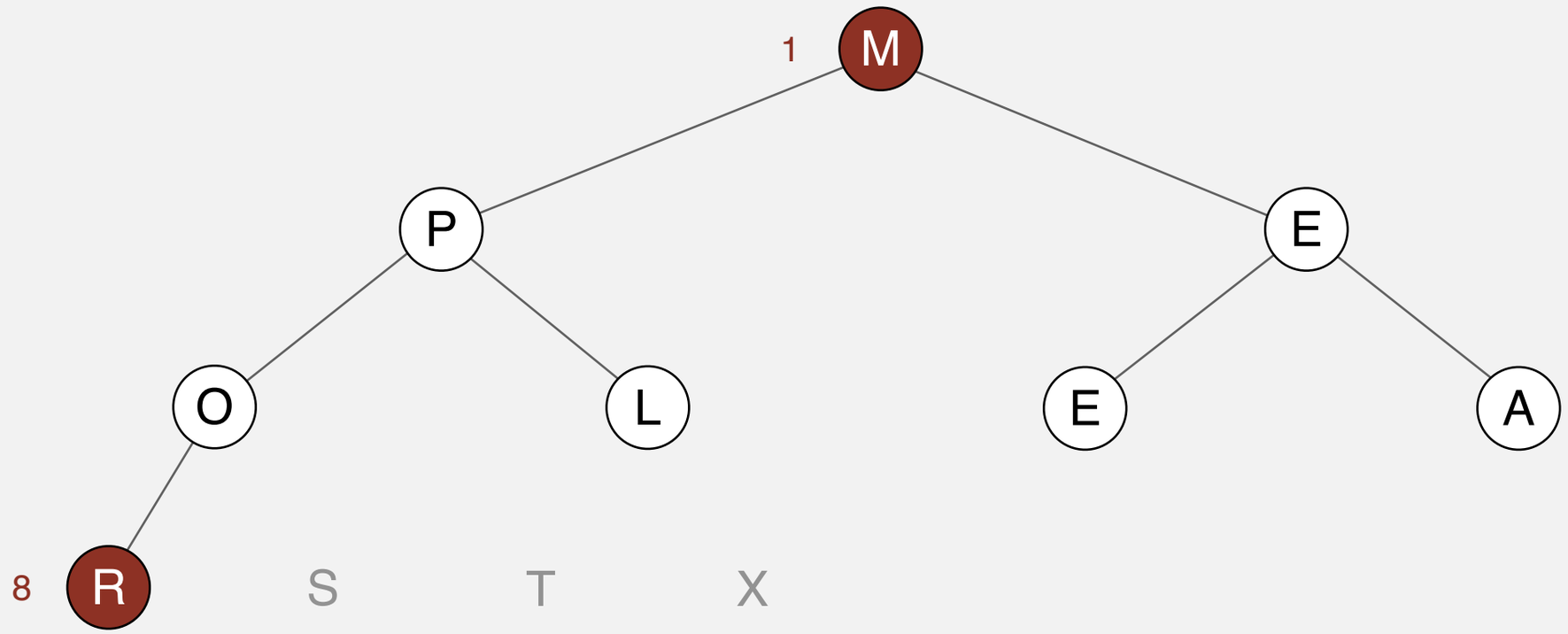


R	P	E	O	L	E	A	M	S	T	X
1							8			

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

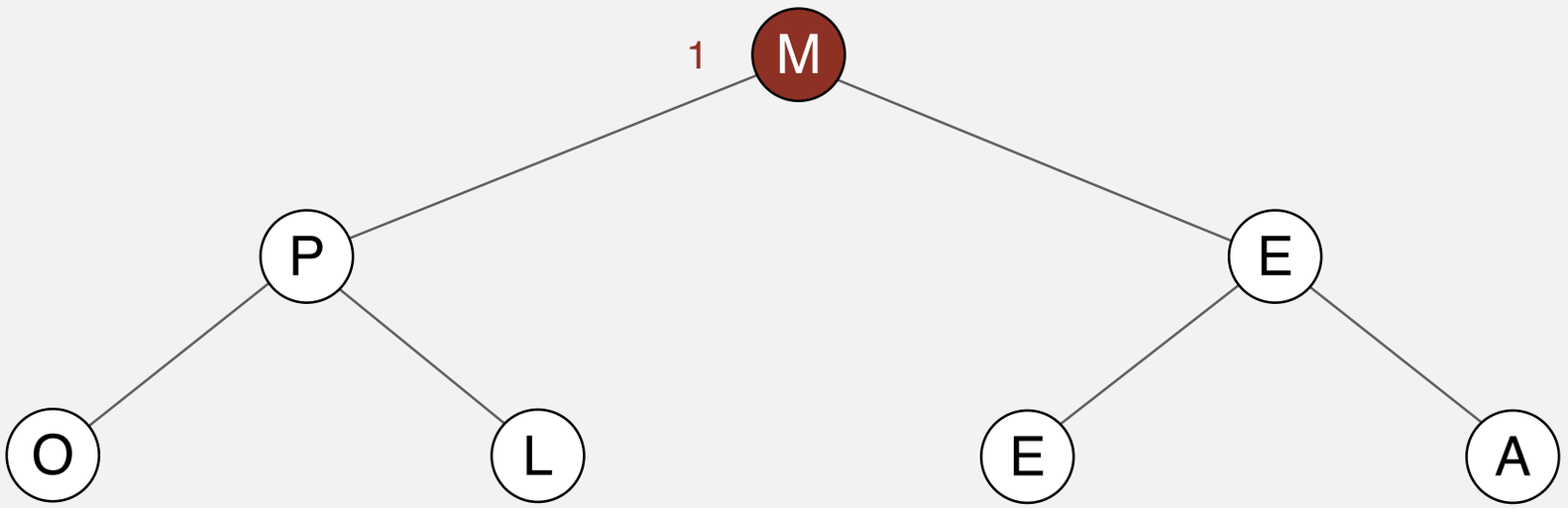


M	P	E	O	L	E	A	R	S	T	X
1							8			

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1



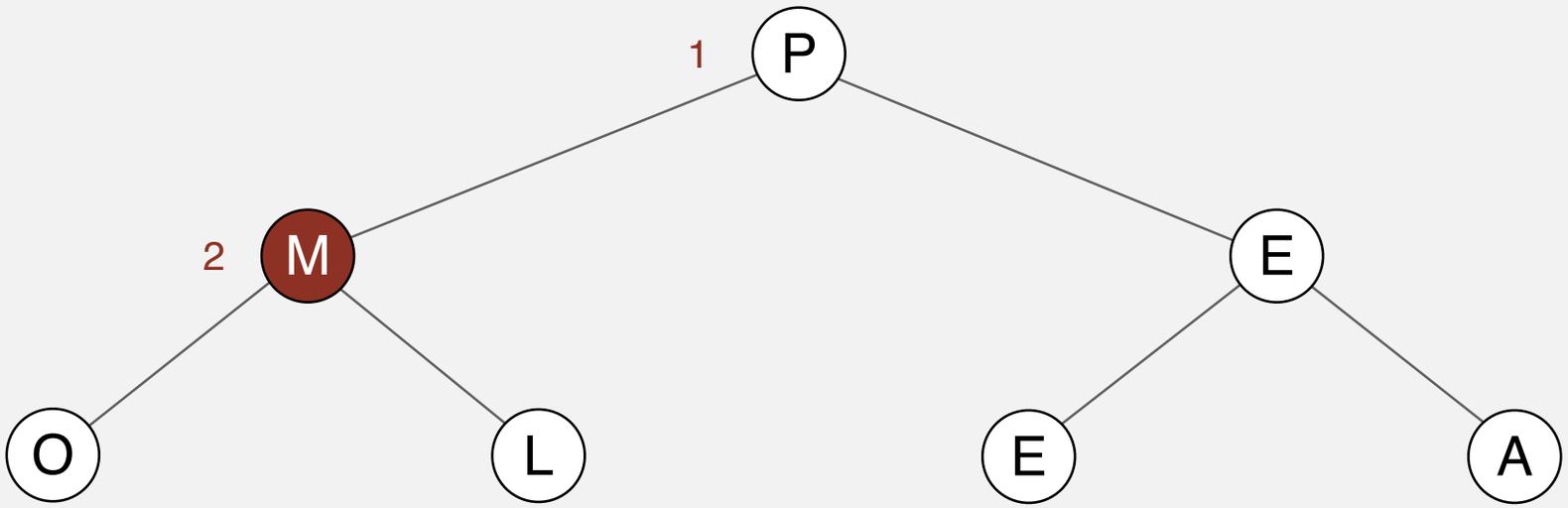
R S T X

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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1



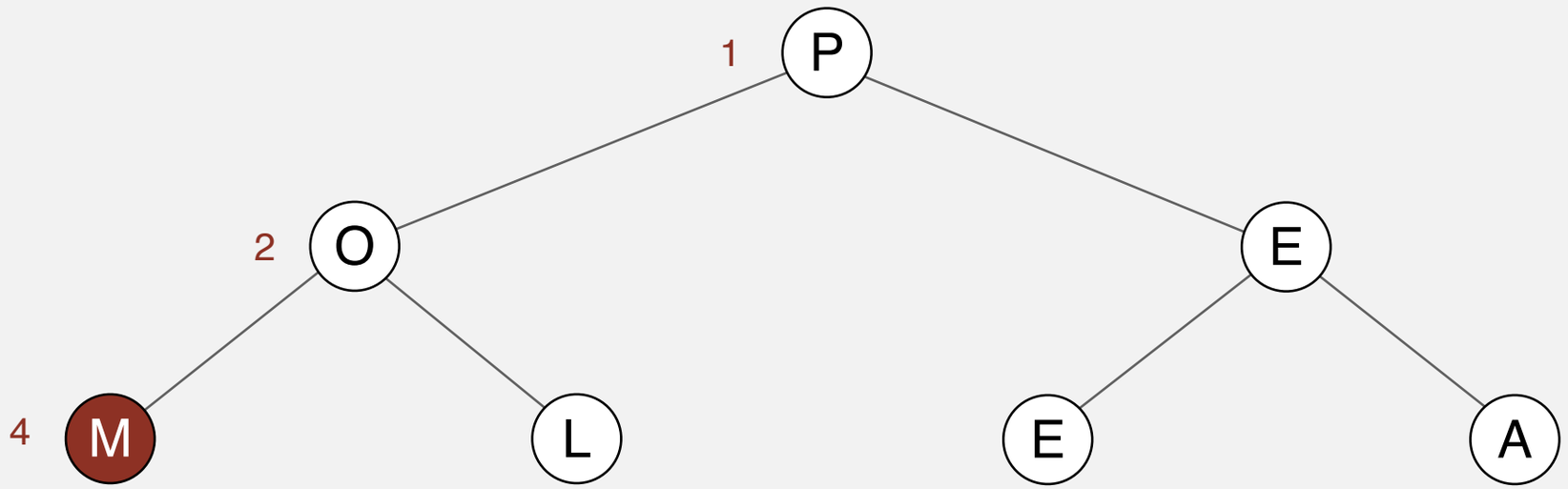
R S T X

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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

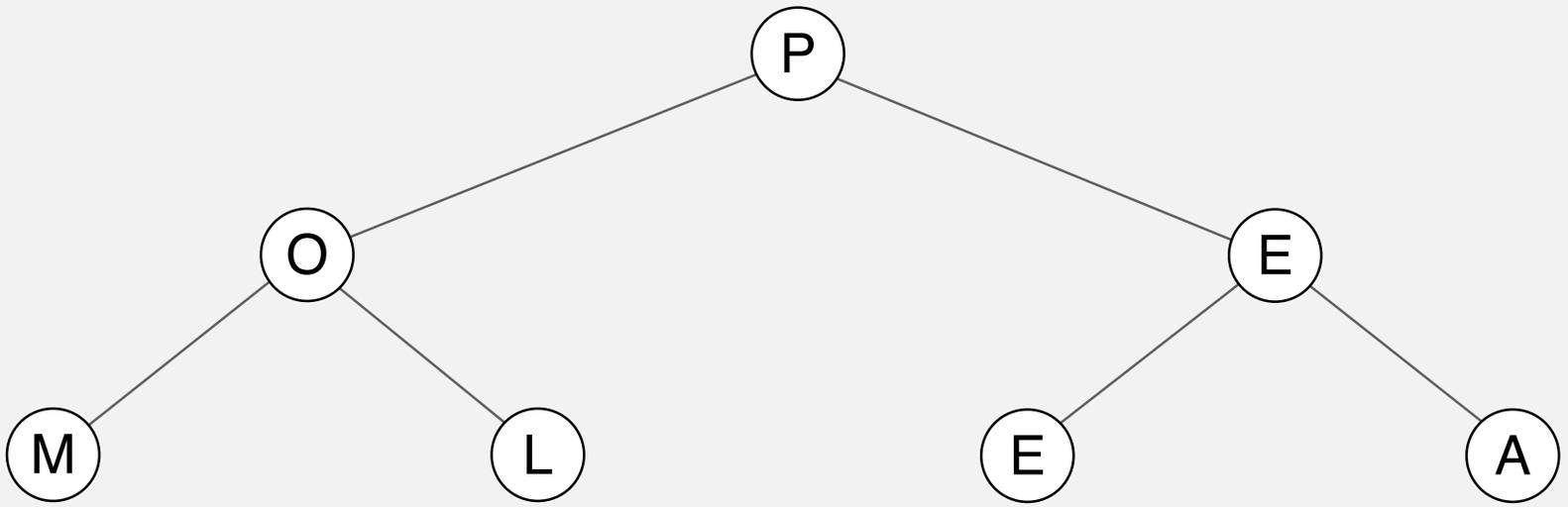


R S T X

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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.



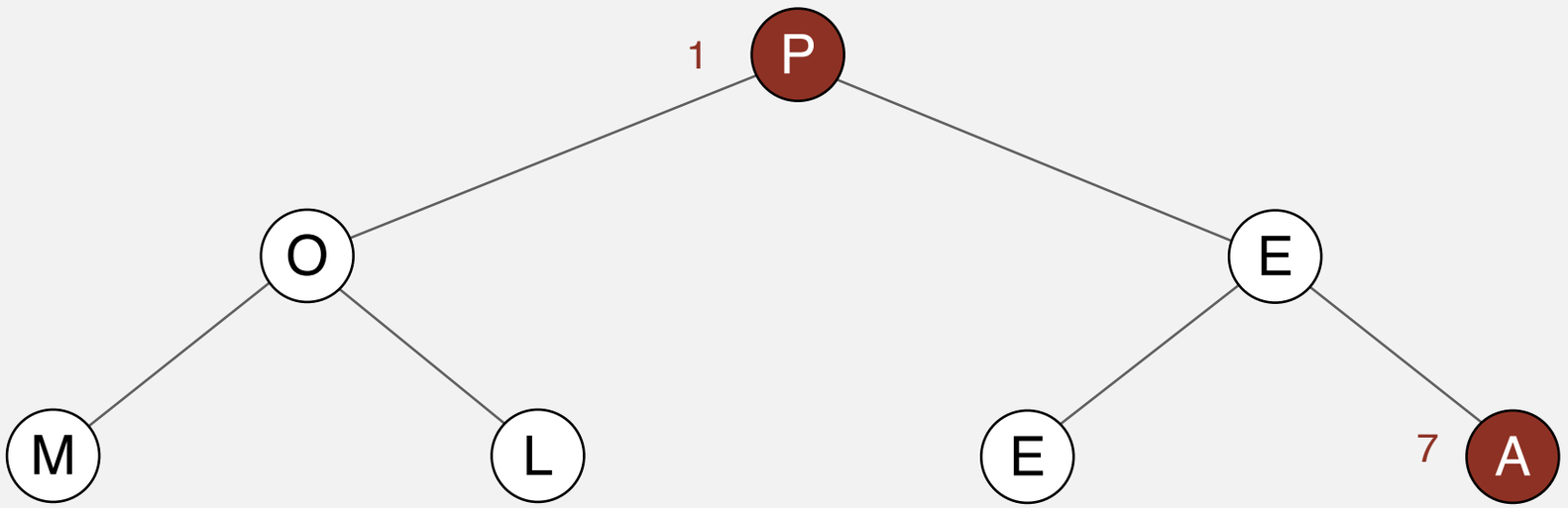
R S T X

P	O	E	M	L	E	A	R	S	T	X

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7



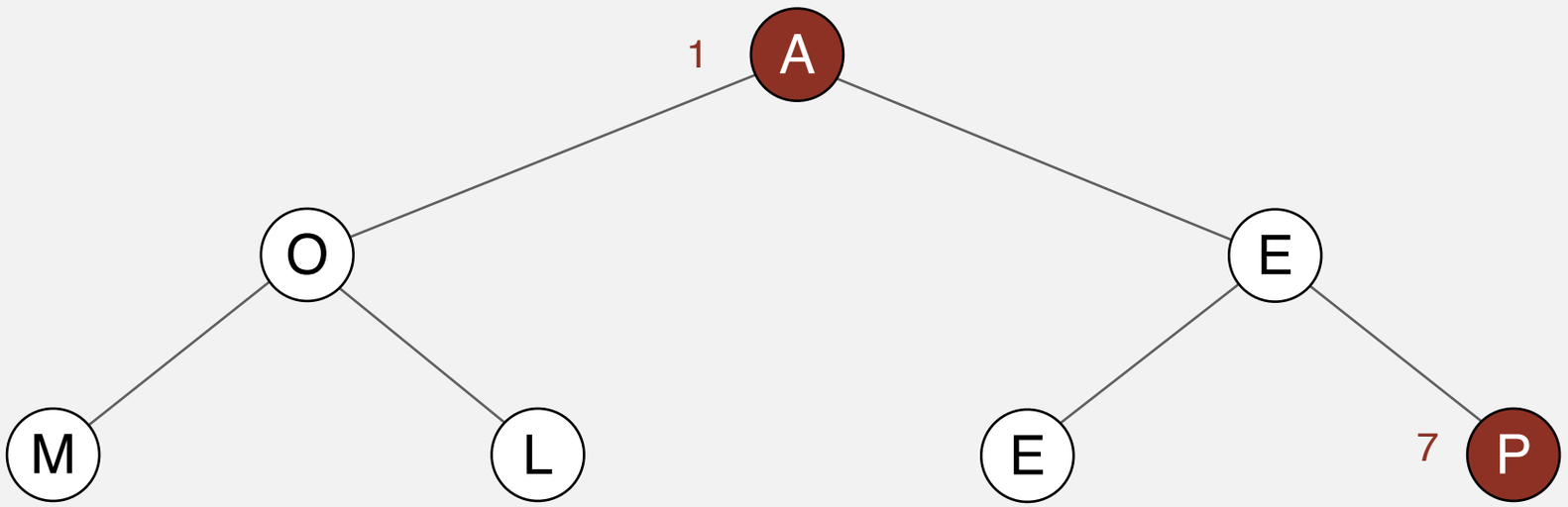
R S T X

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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7



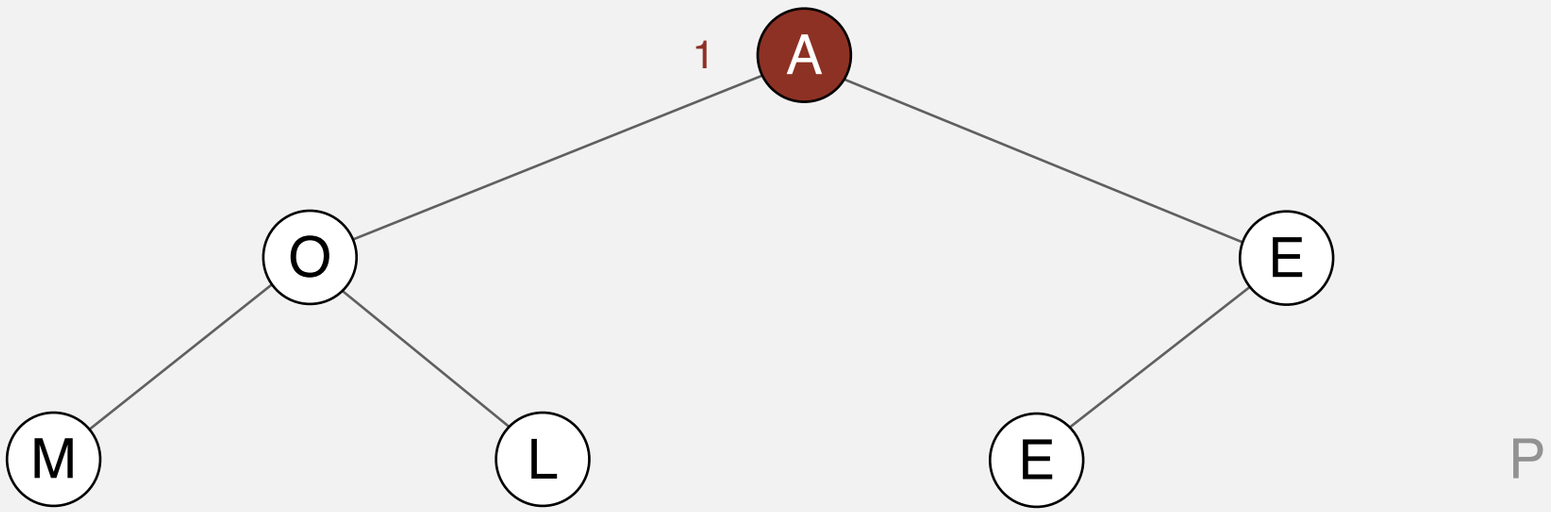
R S T X

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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

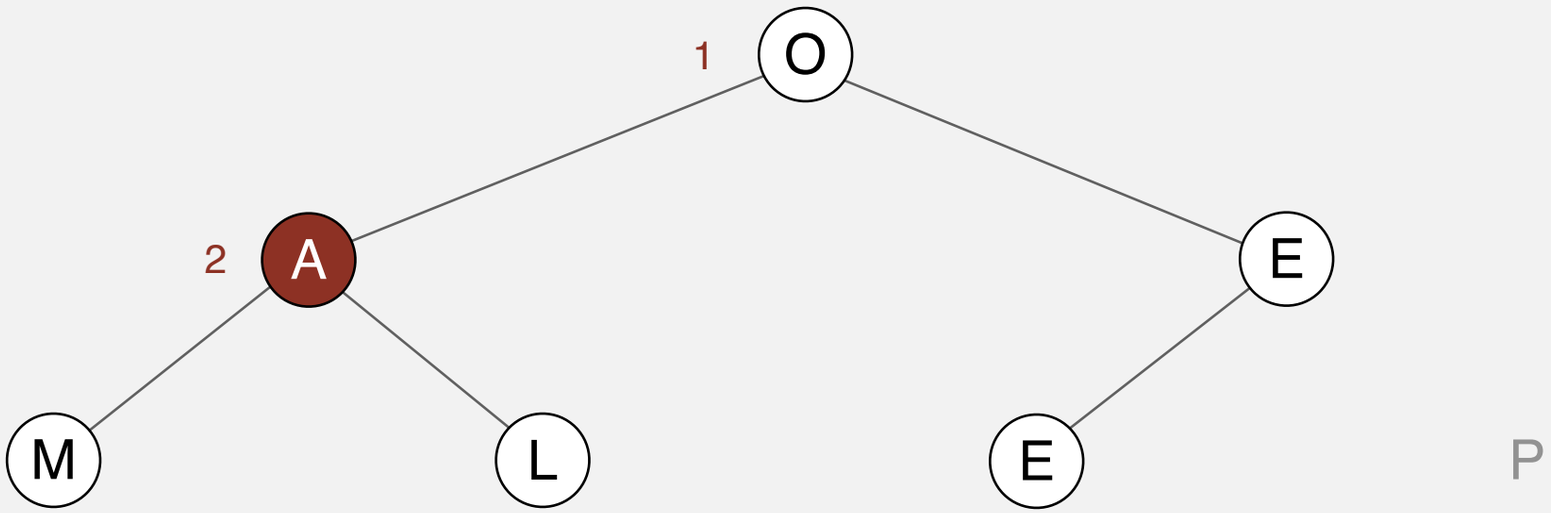


A	O	E	M	L	E	P	R	S	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

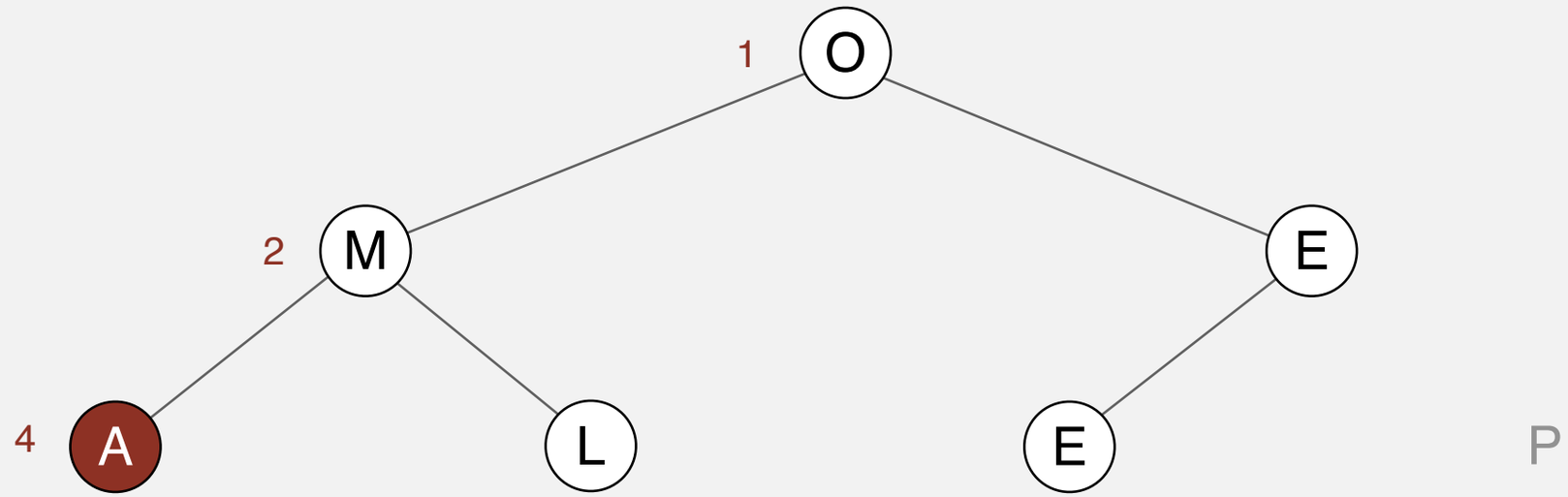


O	A	E	M	L	E	P	R	S	T	X
1	2									

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

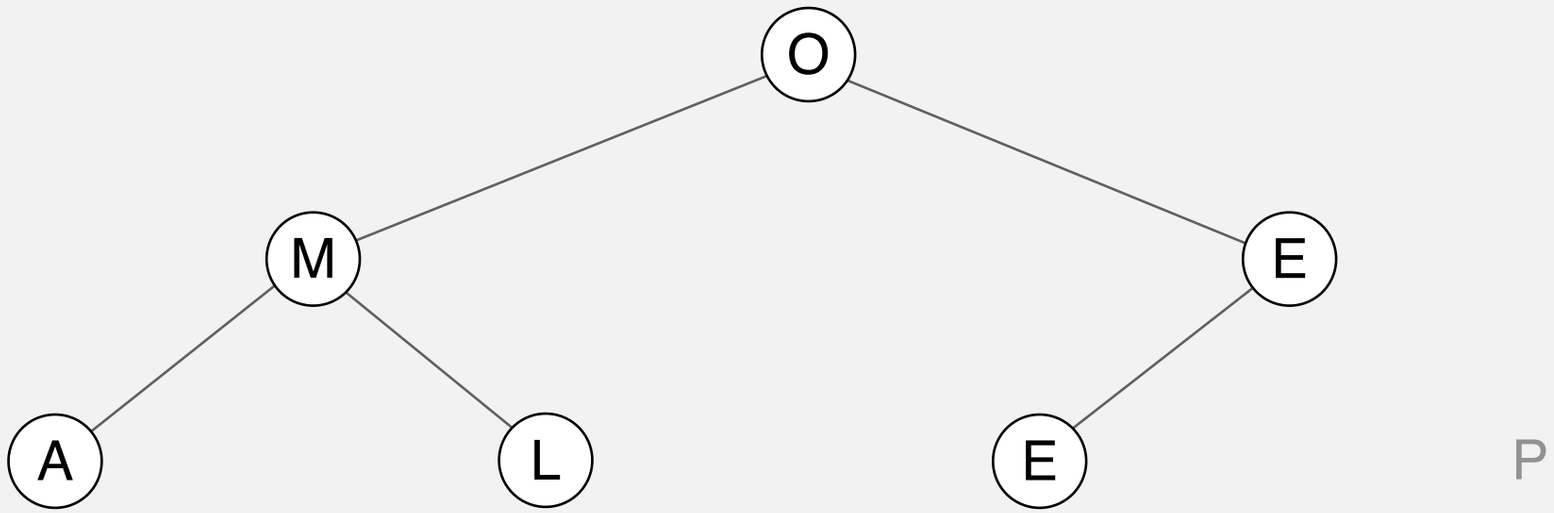


O	M	E	A	L	E	P	R	S	T	X
1	2		4							

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

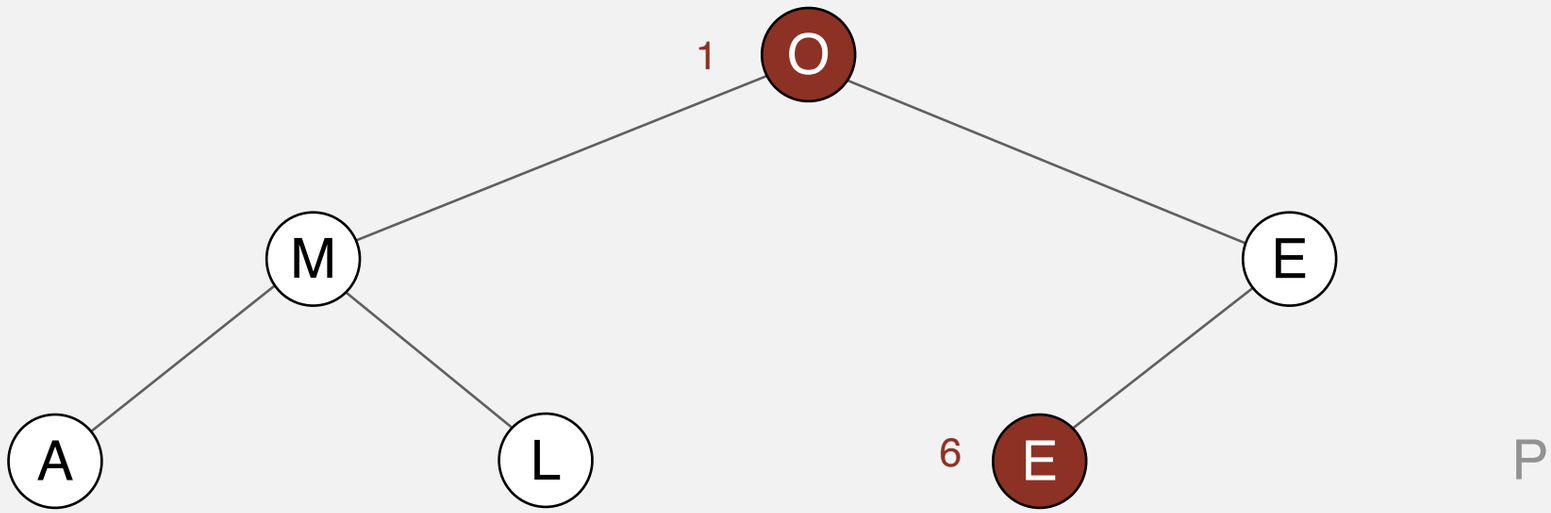


O	M	E	A	L	E	P	R	S	T	X

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

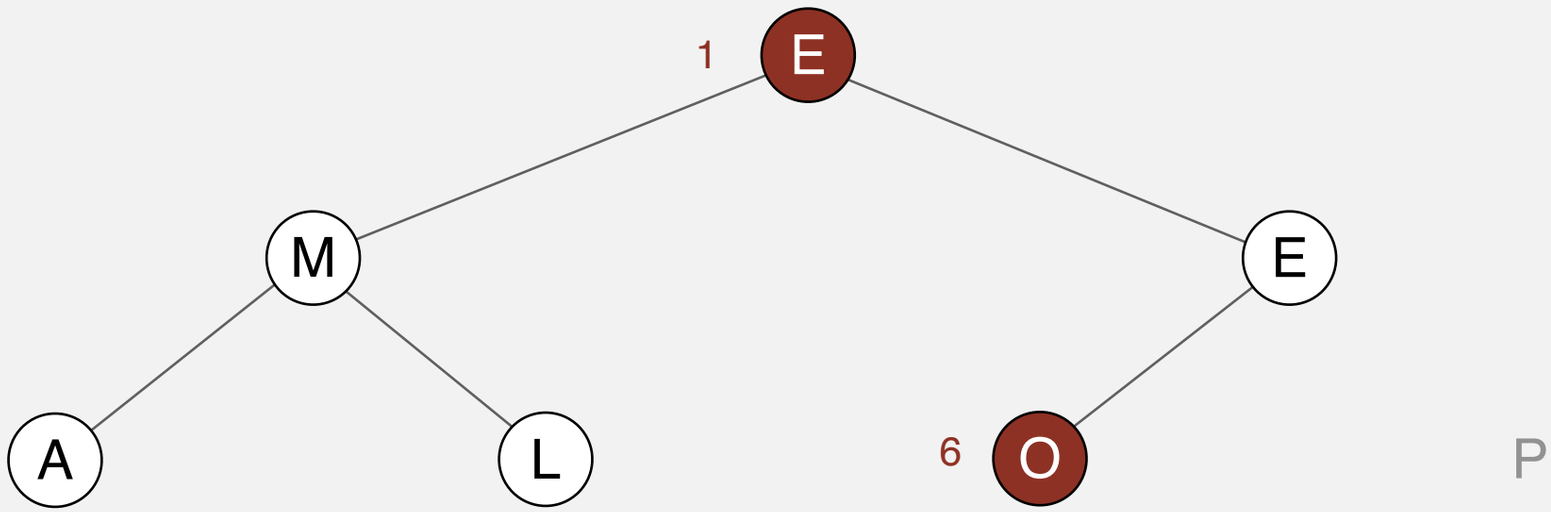


O	M	E	A	L	E	P	R	S	T	X
1					6					

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

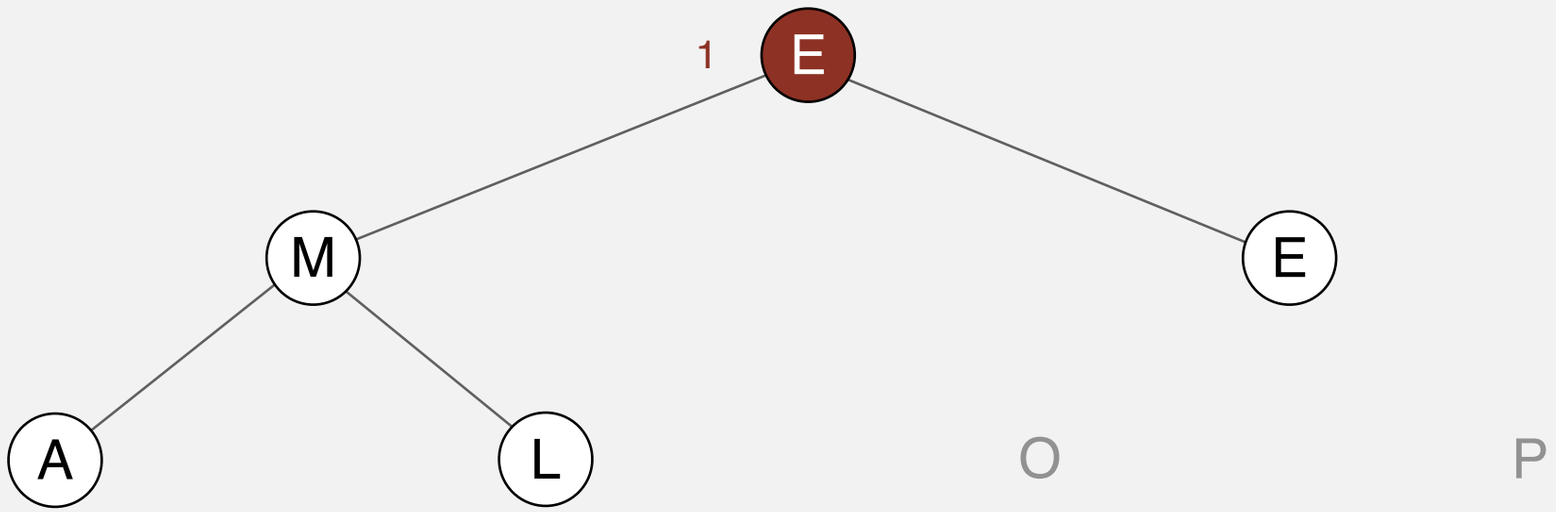


E	M	E	A	L	O	P	R	S	T	X
1					6					

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

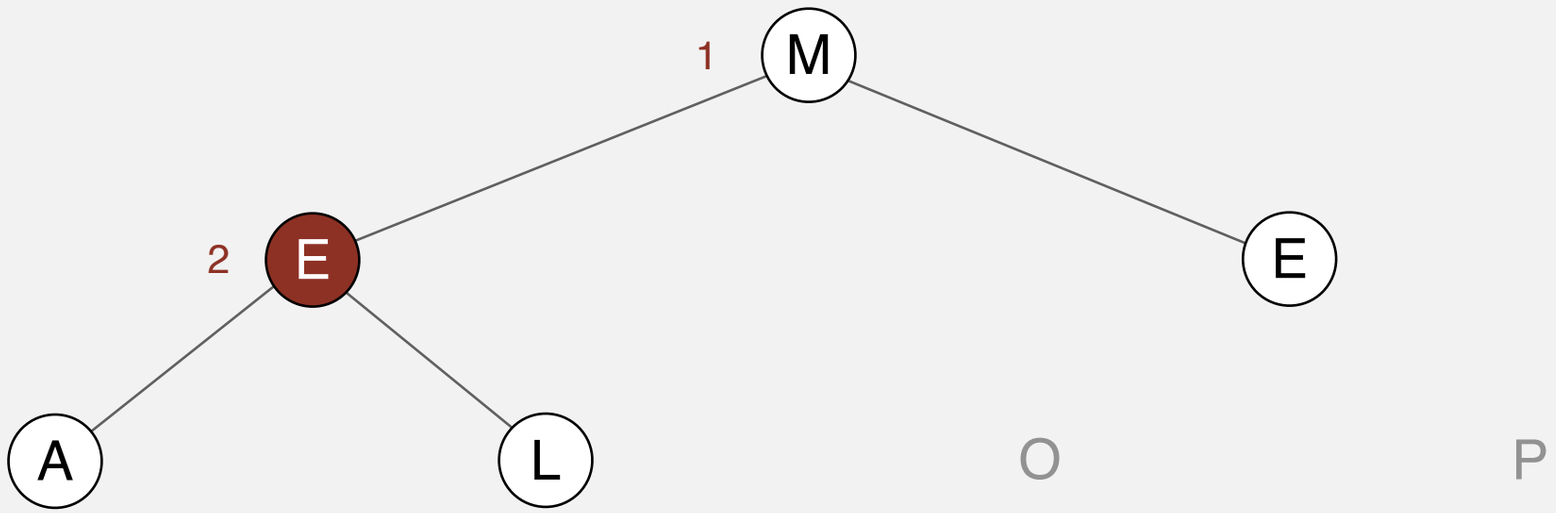


E	M	E	A	L	O	P	R	S	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

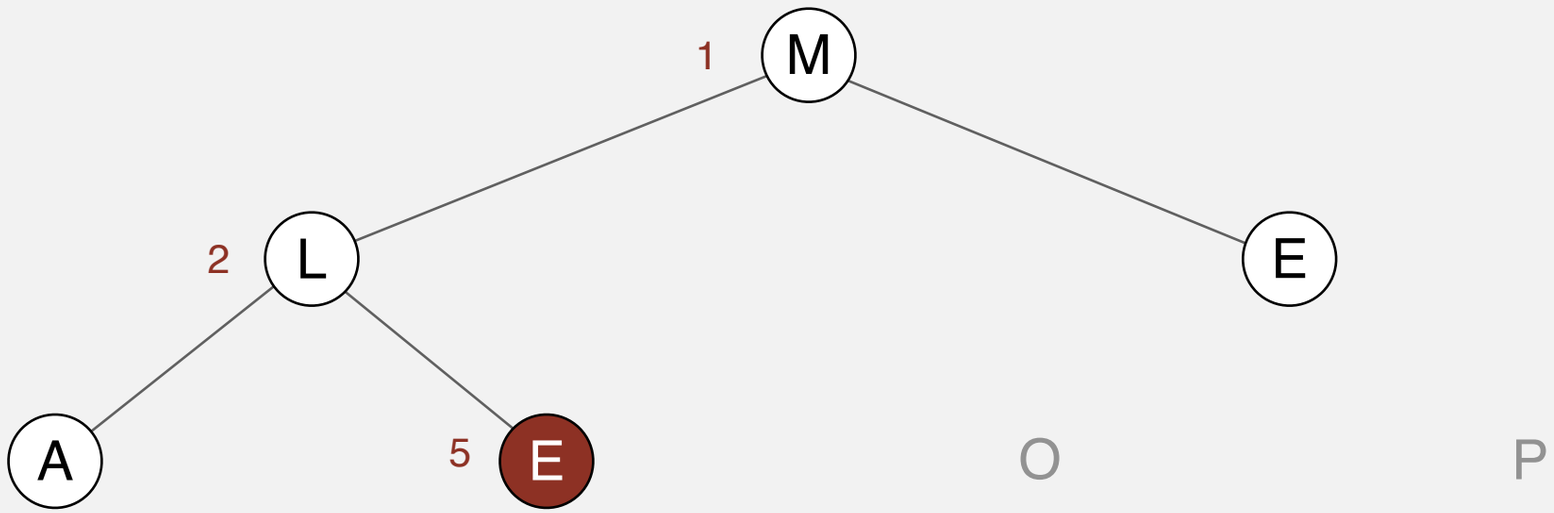


M	E	E	A	L	O	P	R	S	T	X
1	2									

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

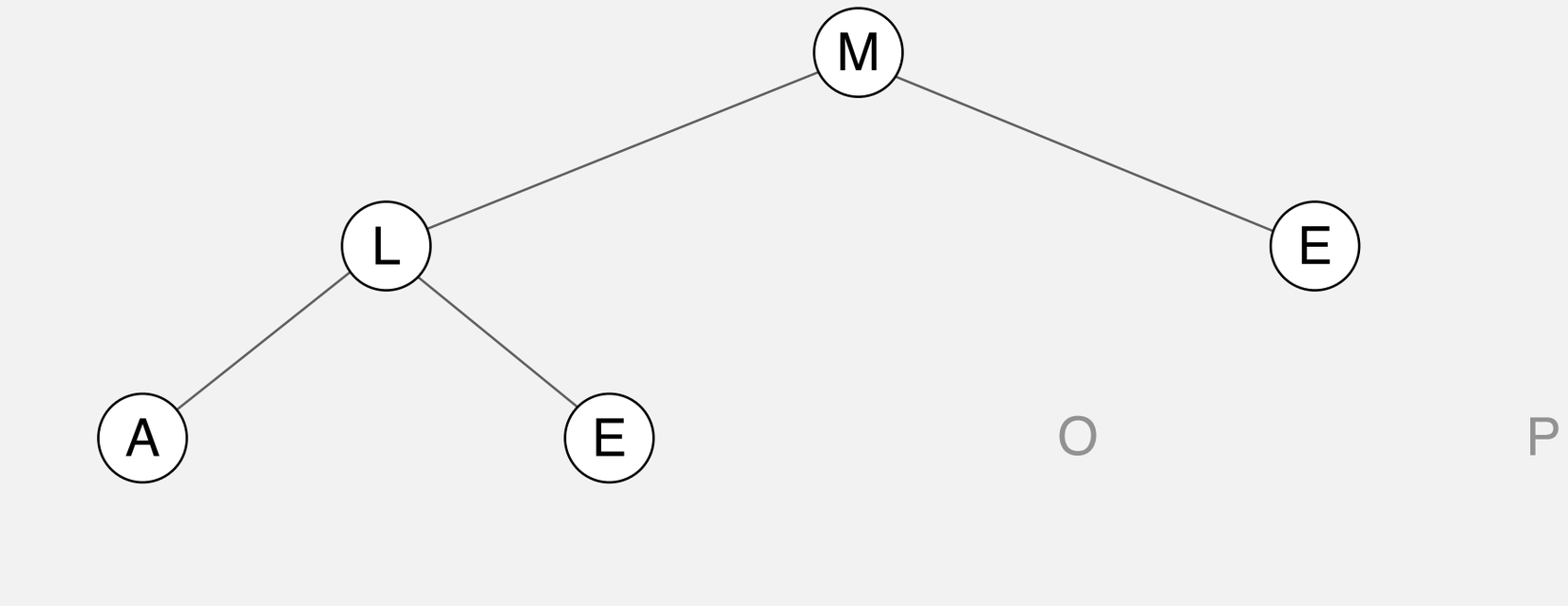
sink 1



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

Heapsort

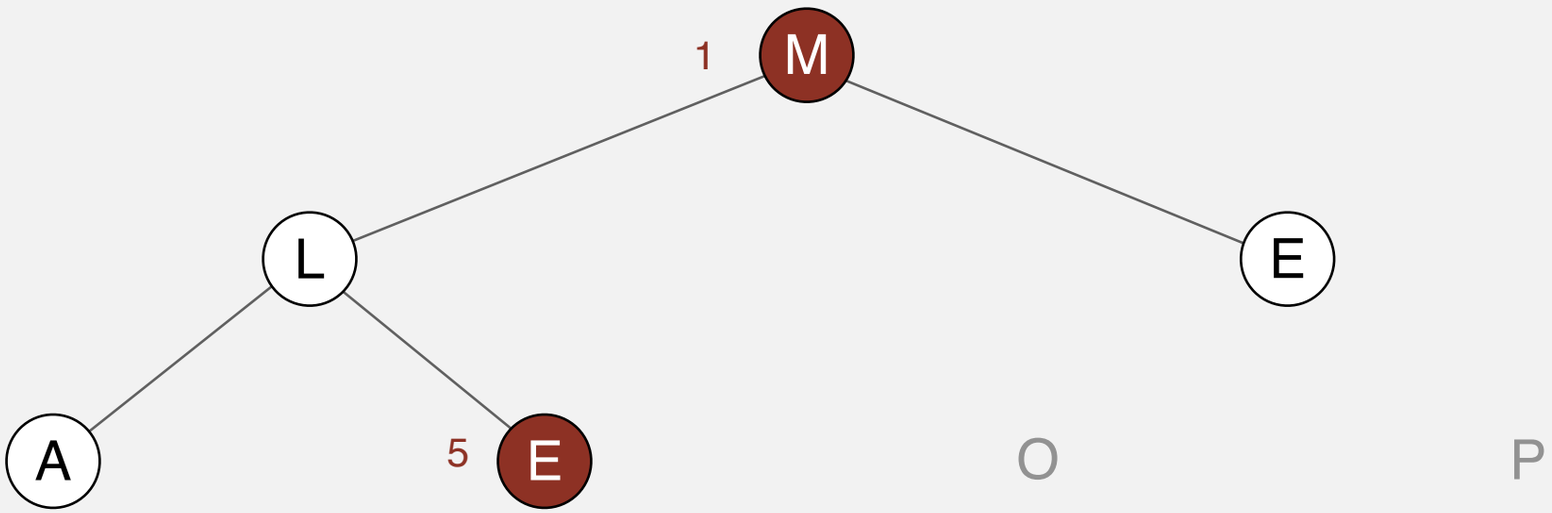
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5

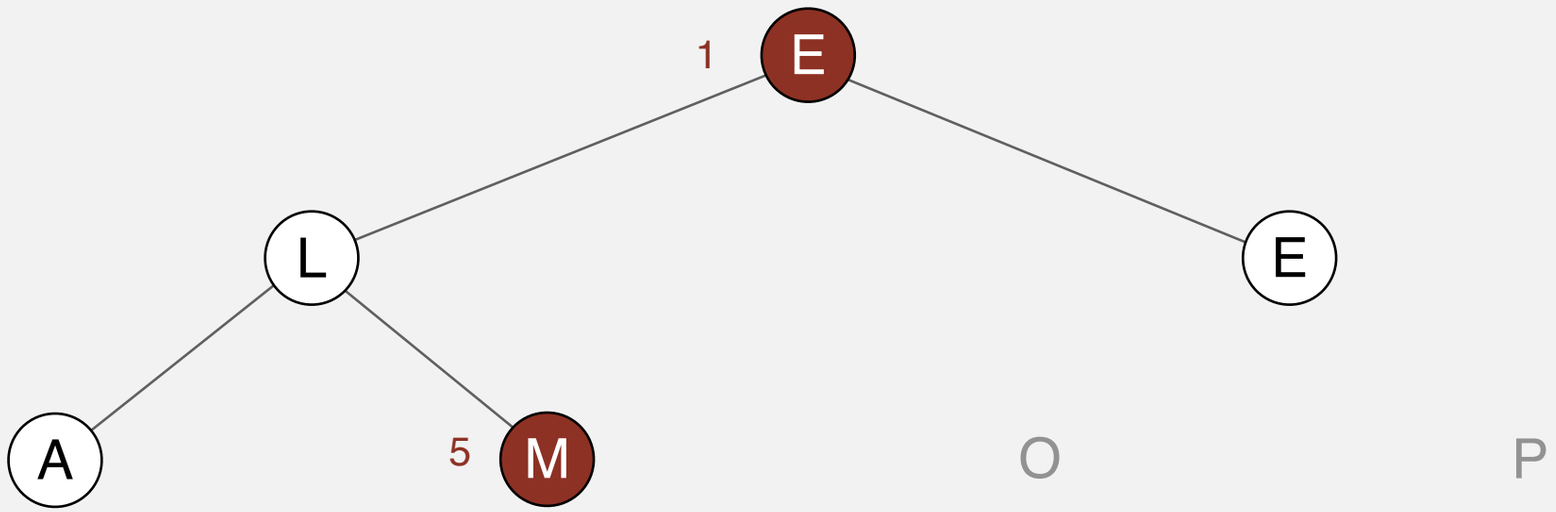


M	L	E	A	E	O	P	R	S	T	X
1				5						

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5

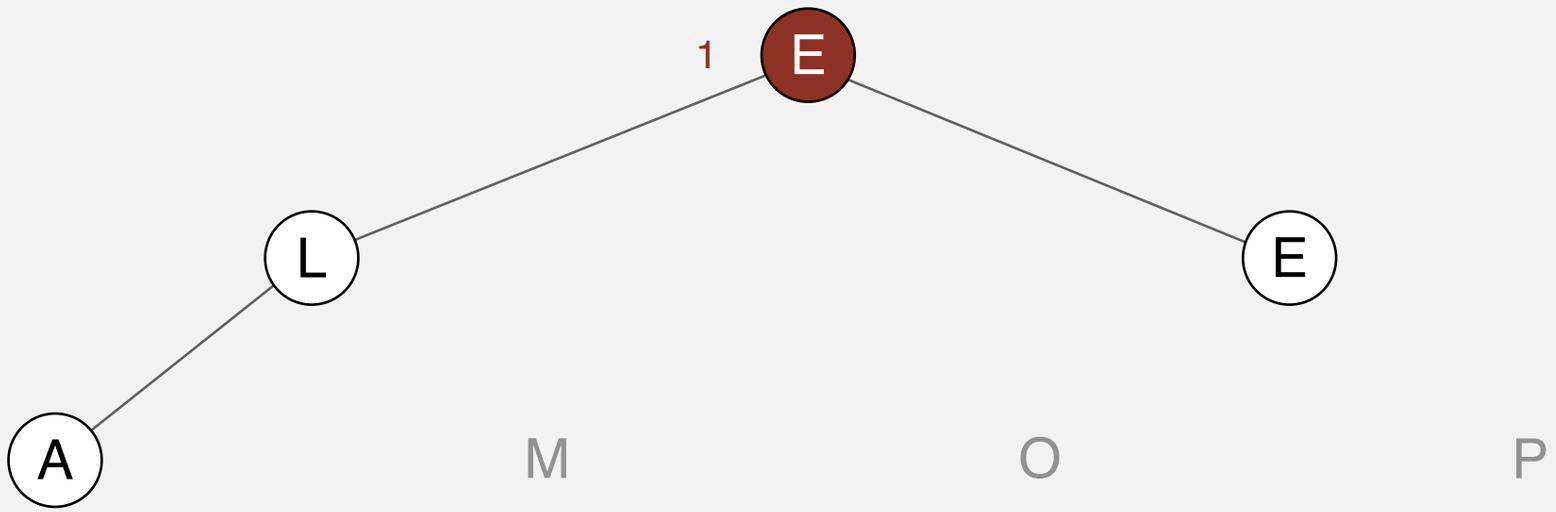


E	L	E	A	M	O	P	R	S	T	X
1				5						

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

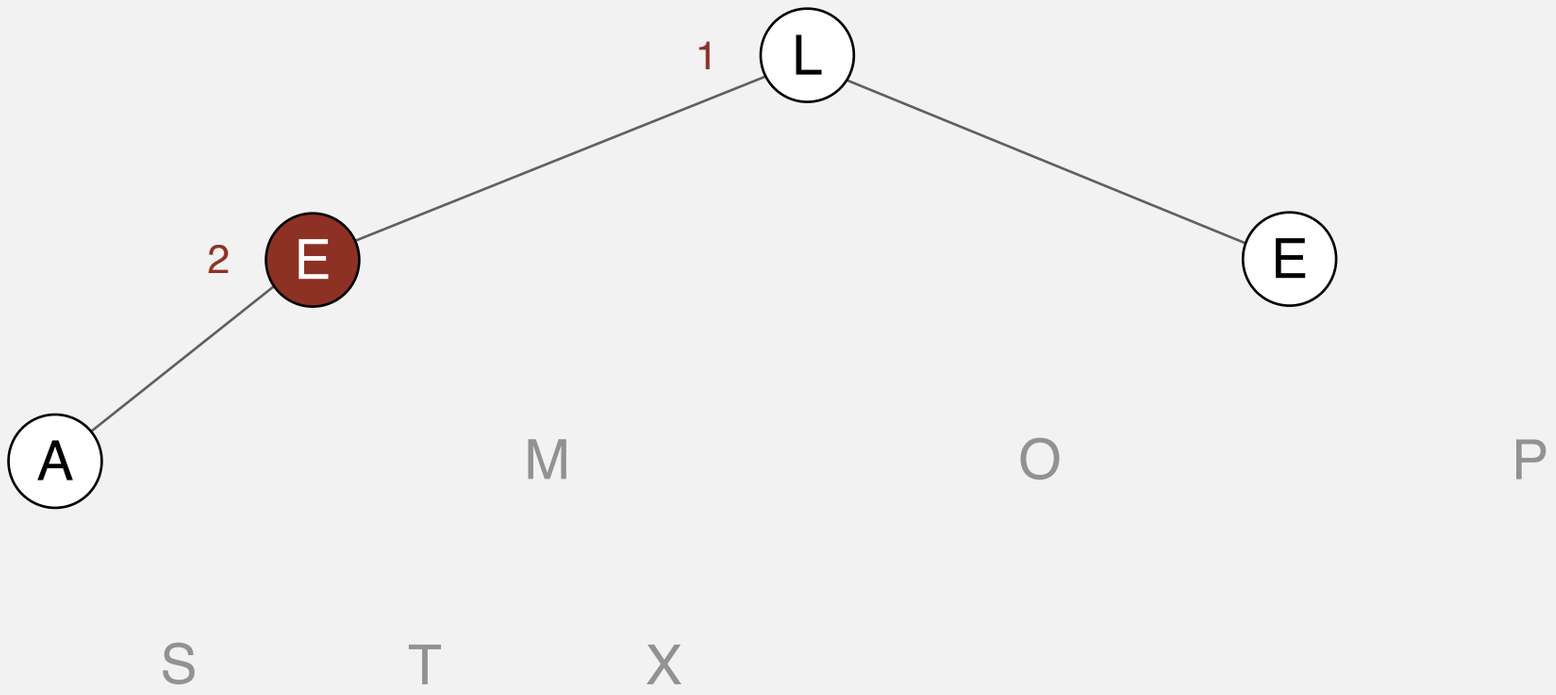


E	L	E	A	M	O	P	R	S	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

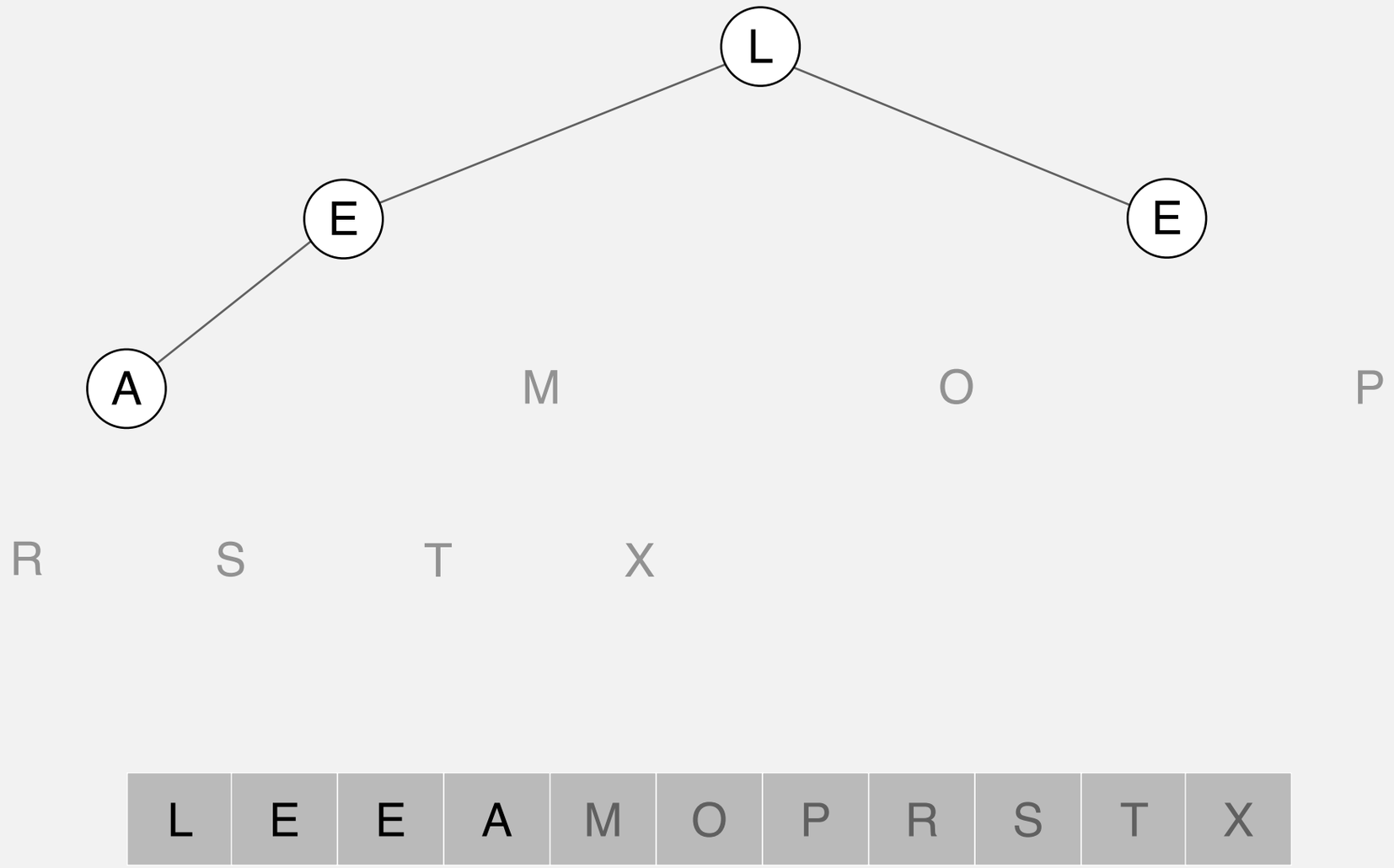
sink 1



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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

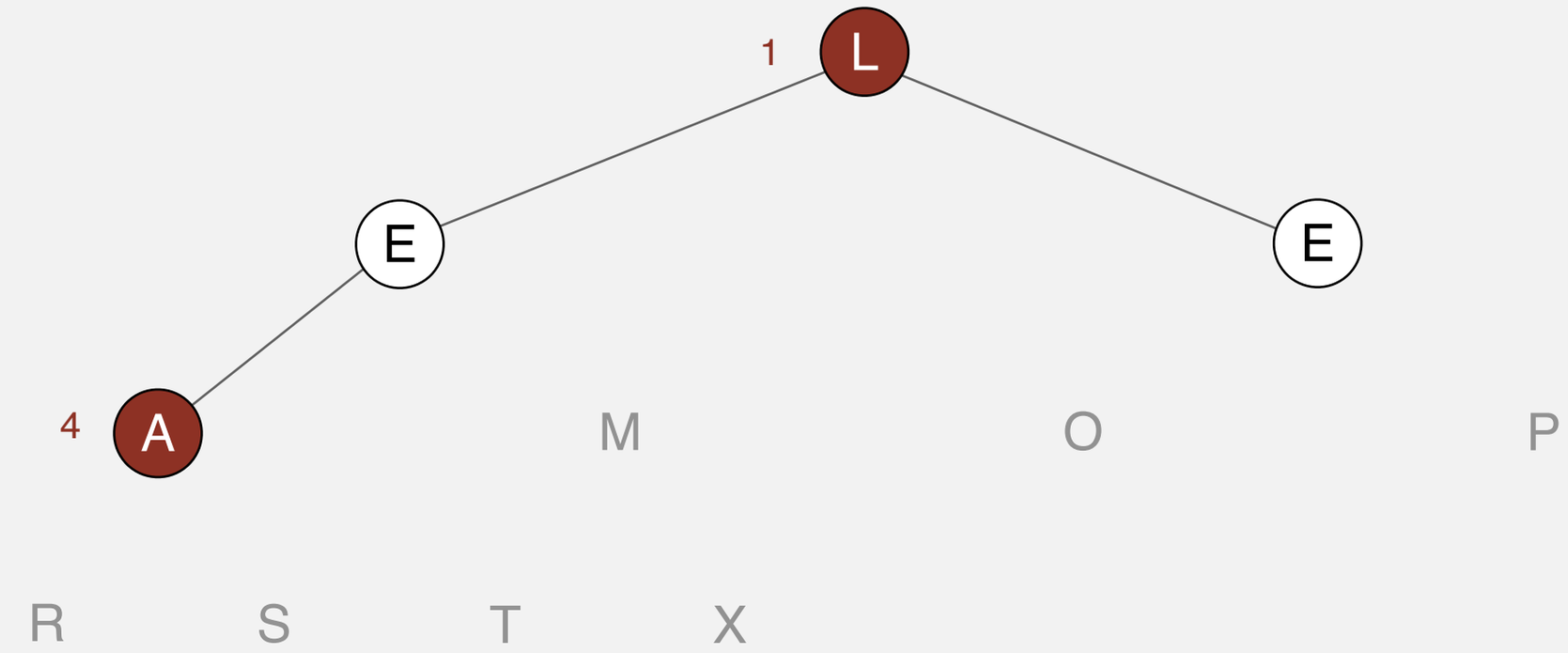


L	E	E	A	M	O	P	R	S	T	X

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

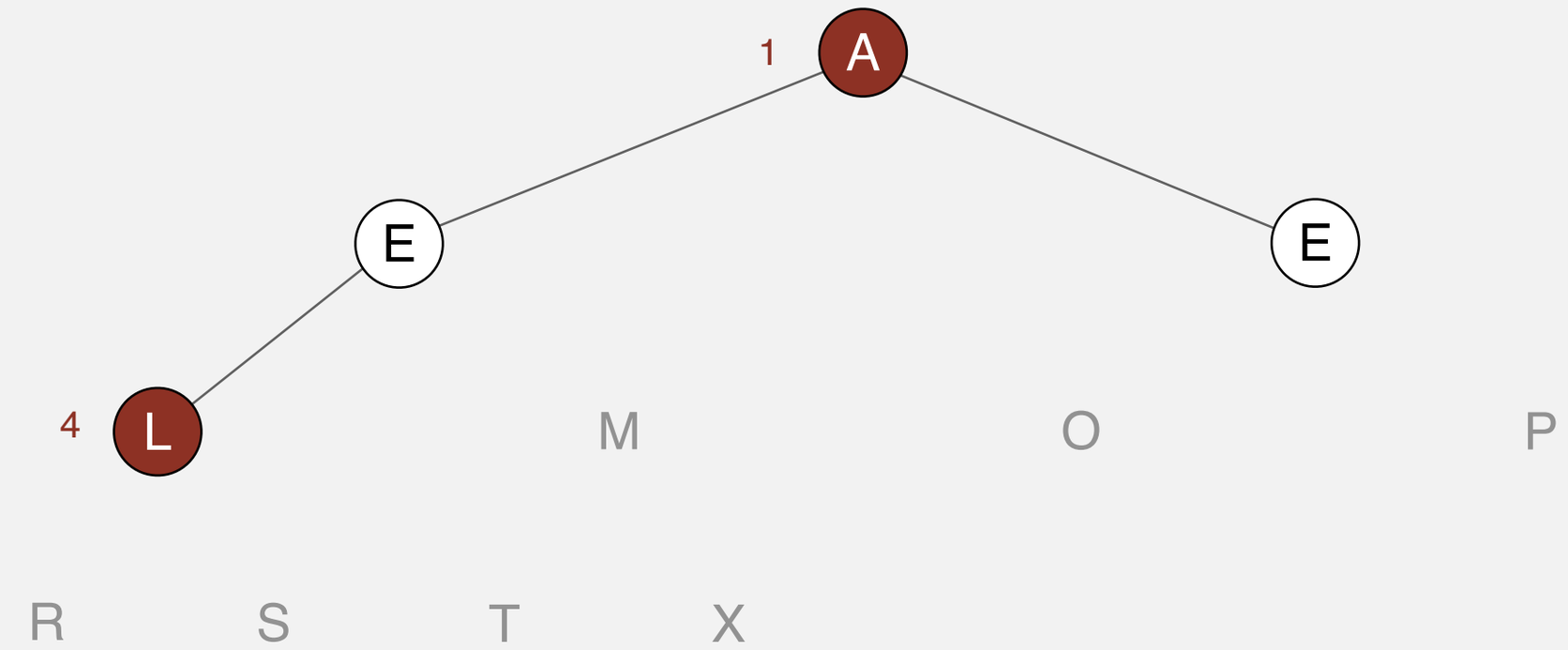


L	E	E	A	M	O	P	R	S	T	X
1			4							

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

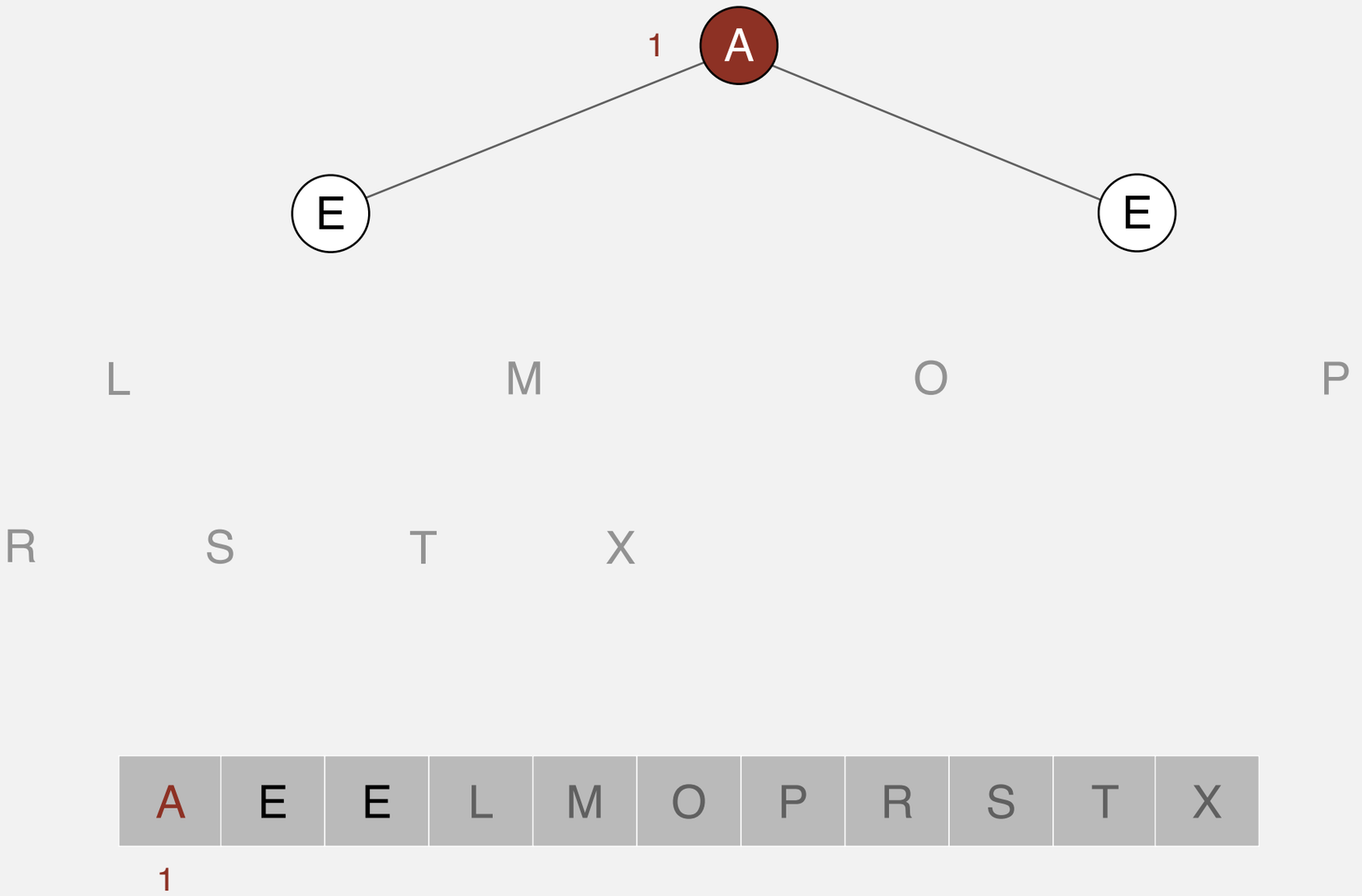


A	E	E	L	M	O	P	R	S	T	X
1			4							

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

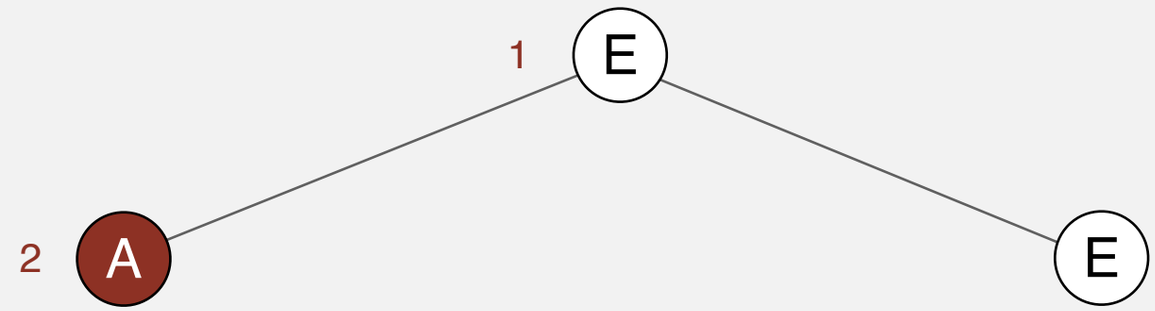
sink 1



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1



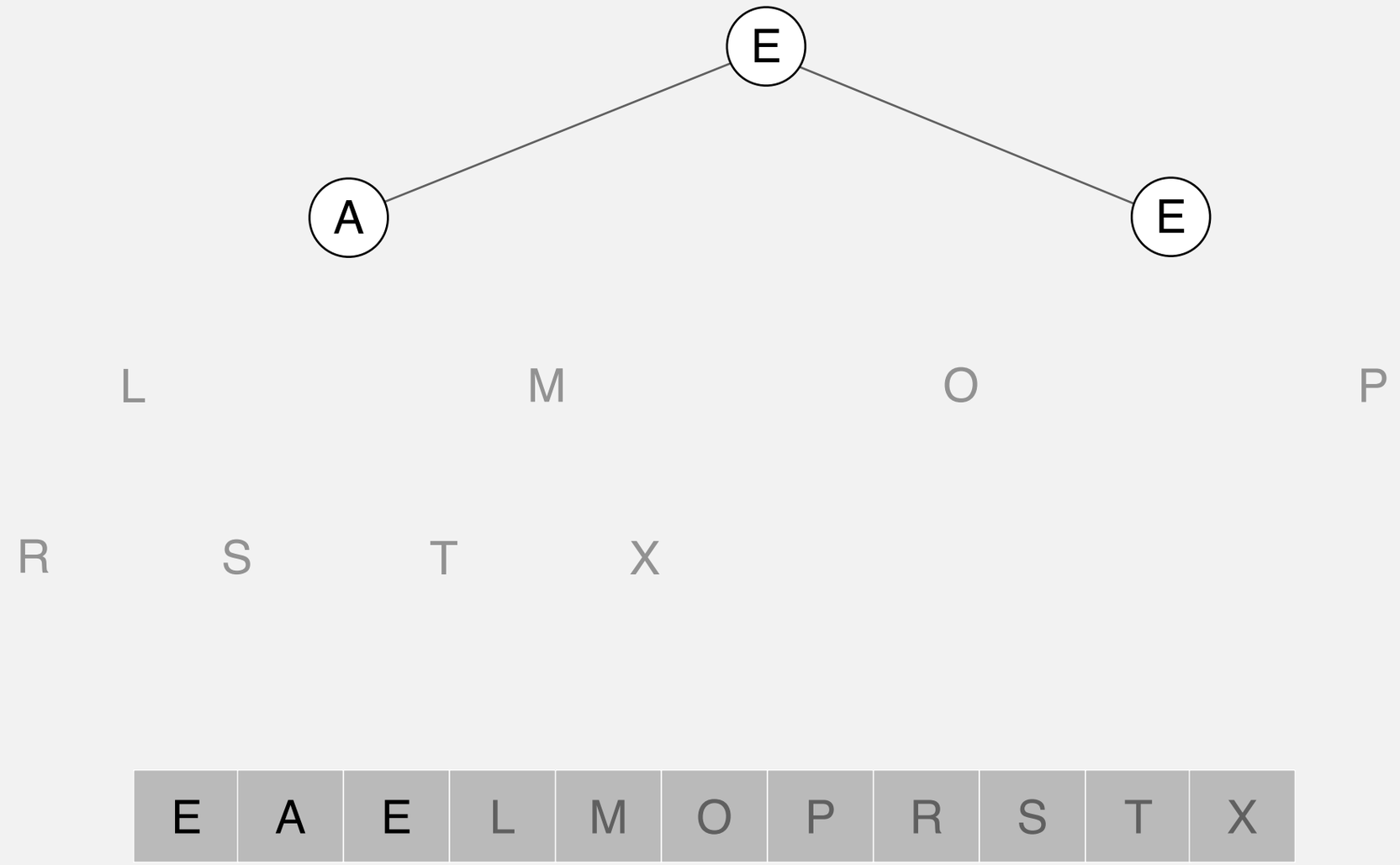
L M O P

R S T X

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

Heapsort

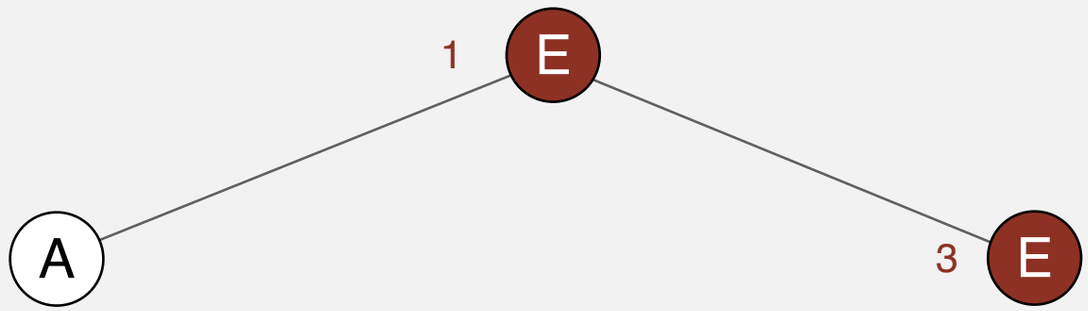
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3



L M O P

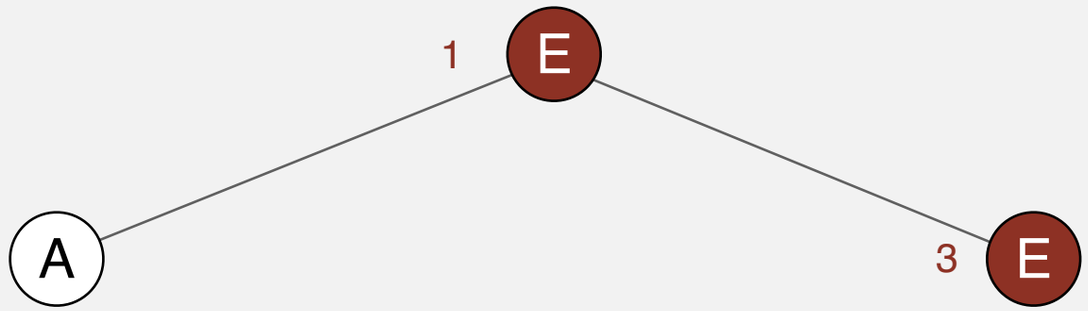
R S T X

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

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3



L M O P

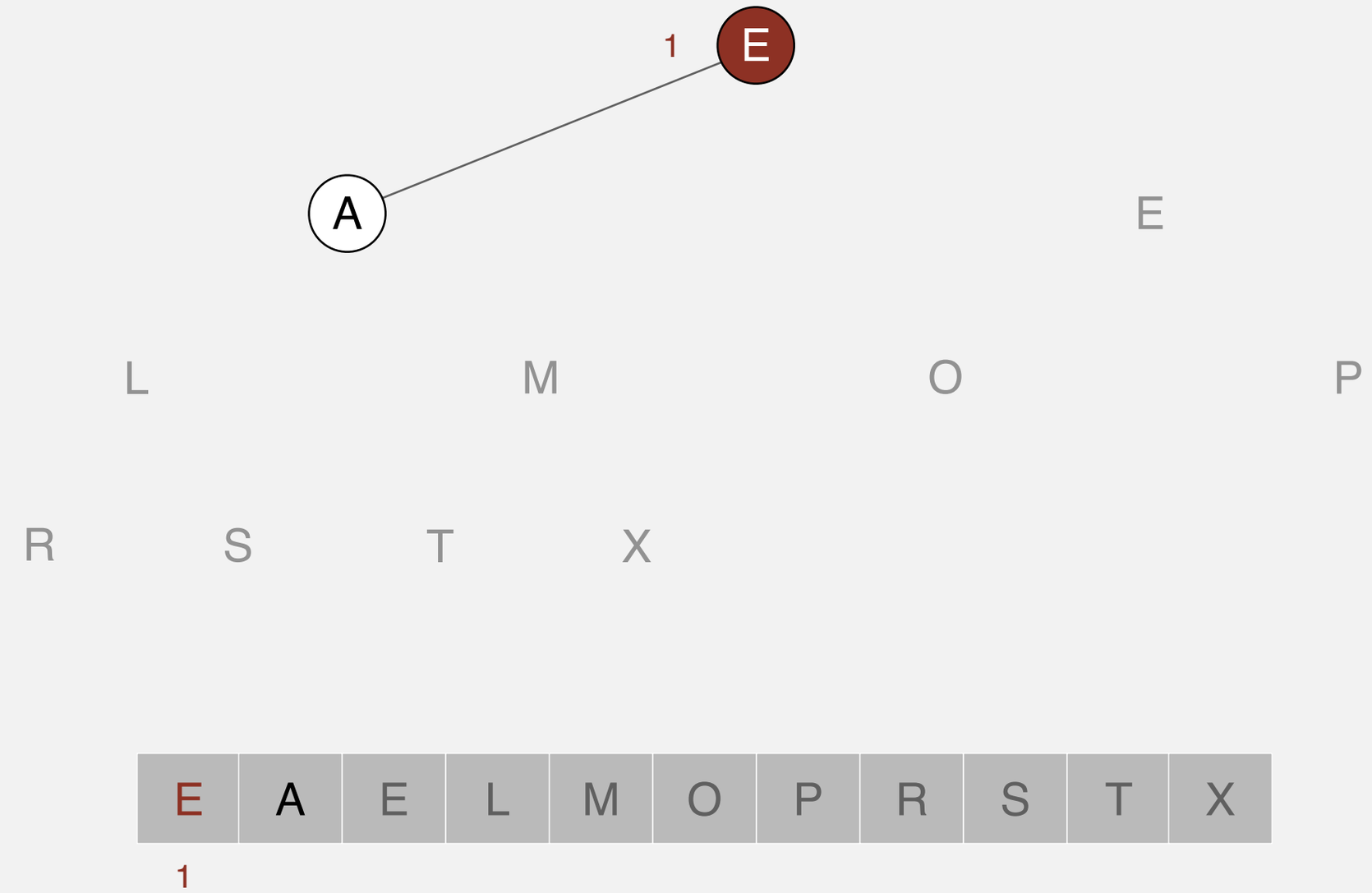
R S T X

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

Heapsort

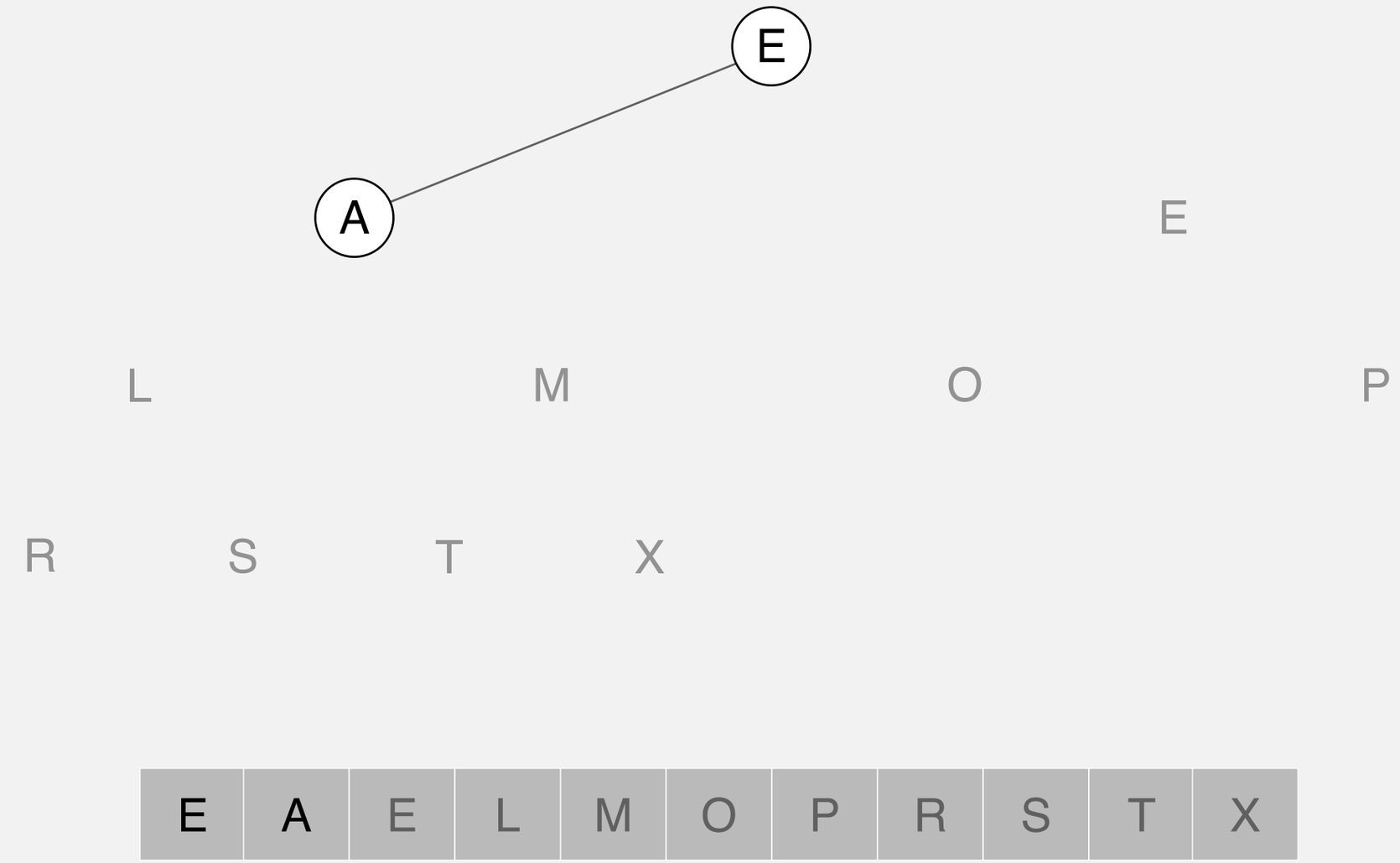
Sortdown. Repeatedly delete the largest remaining item.

sink 1



Heapsort

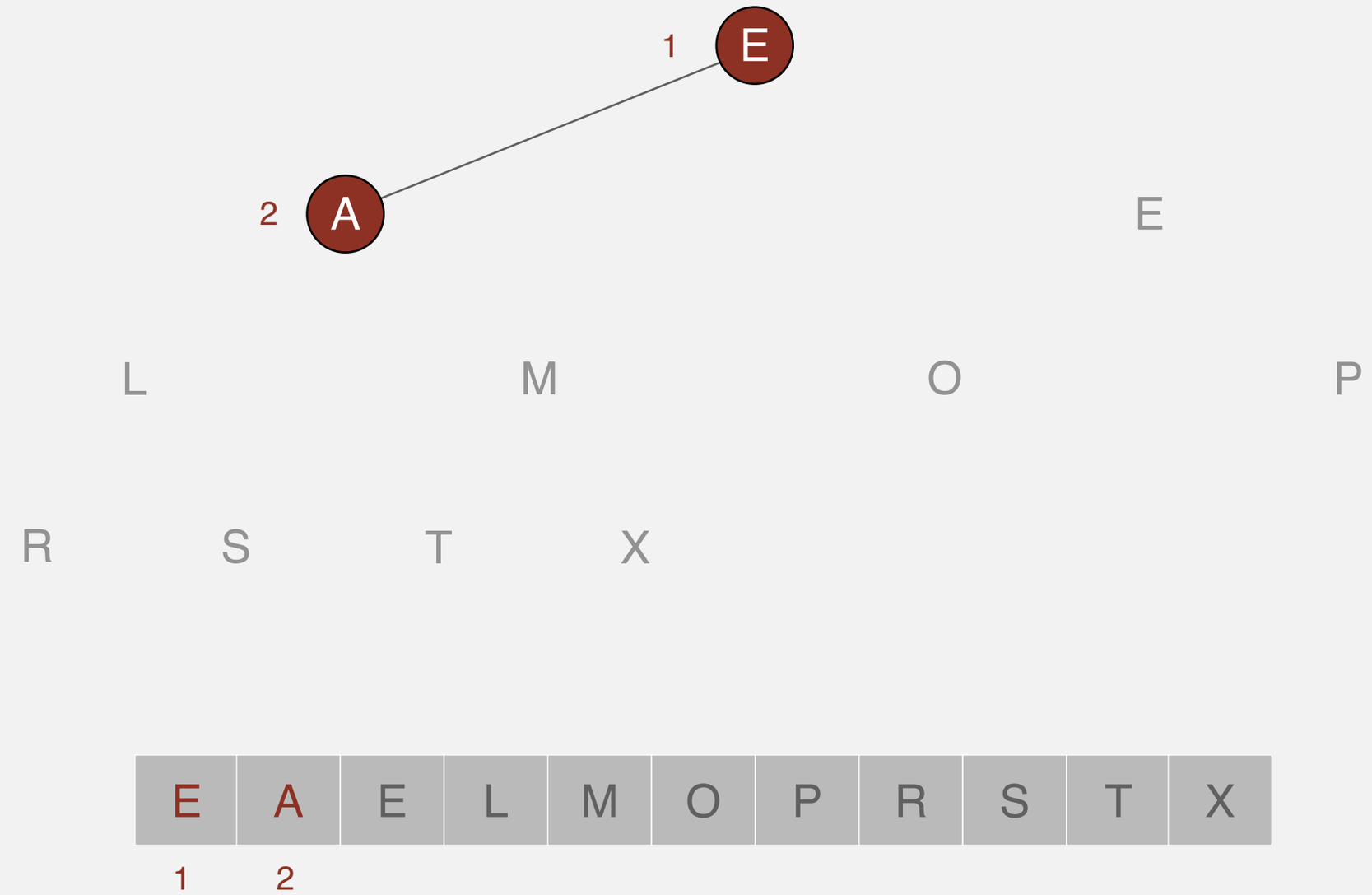
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

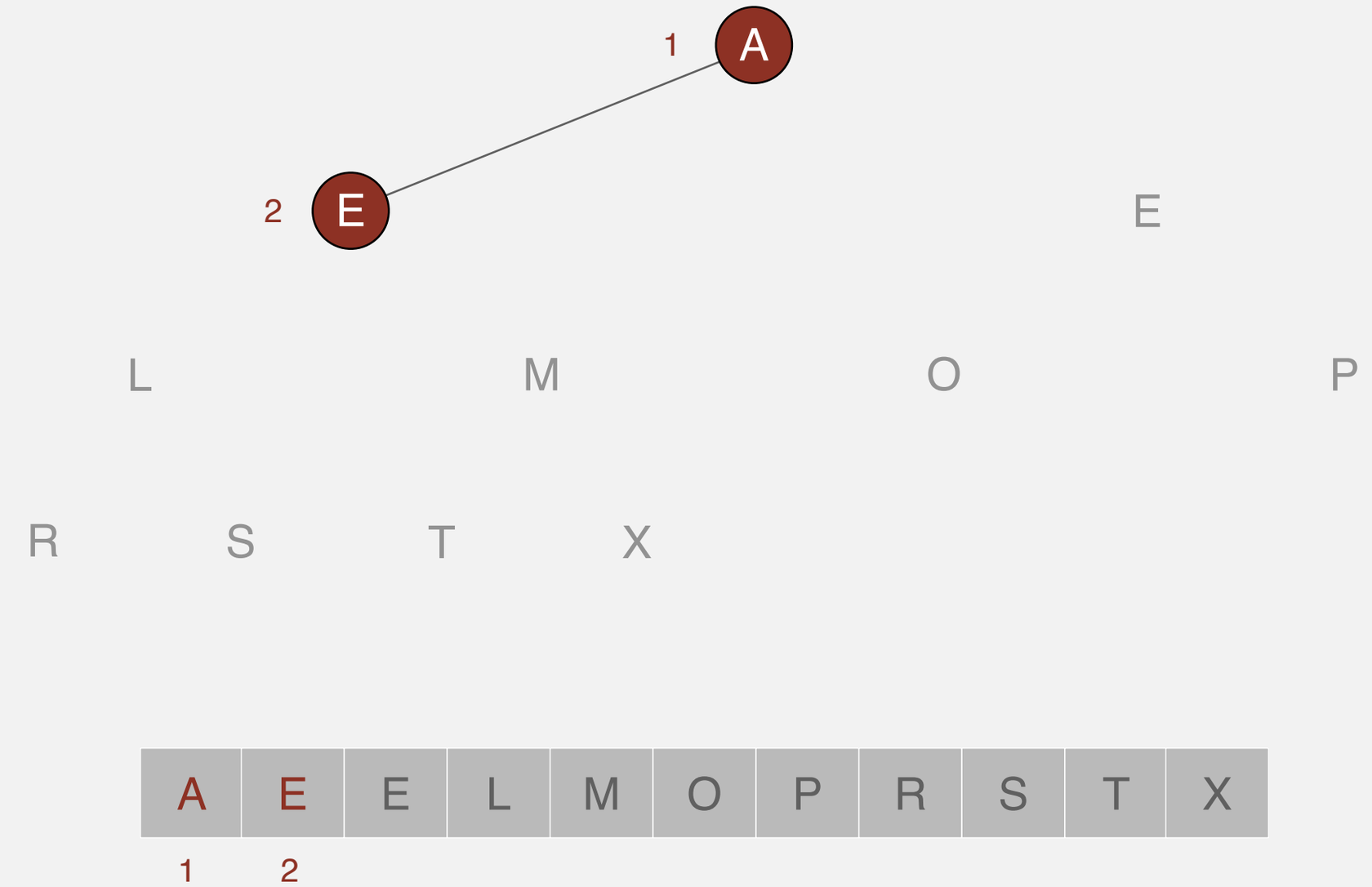
exchange 1 and 2



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

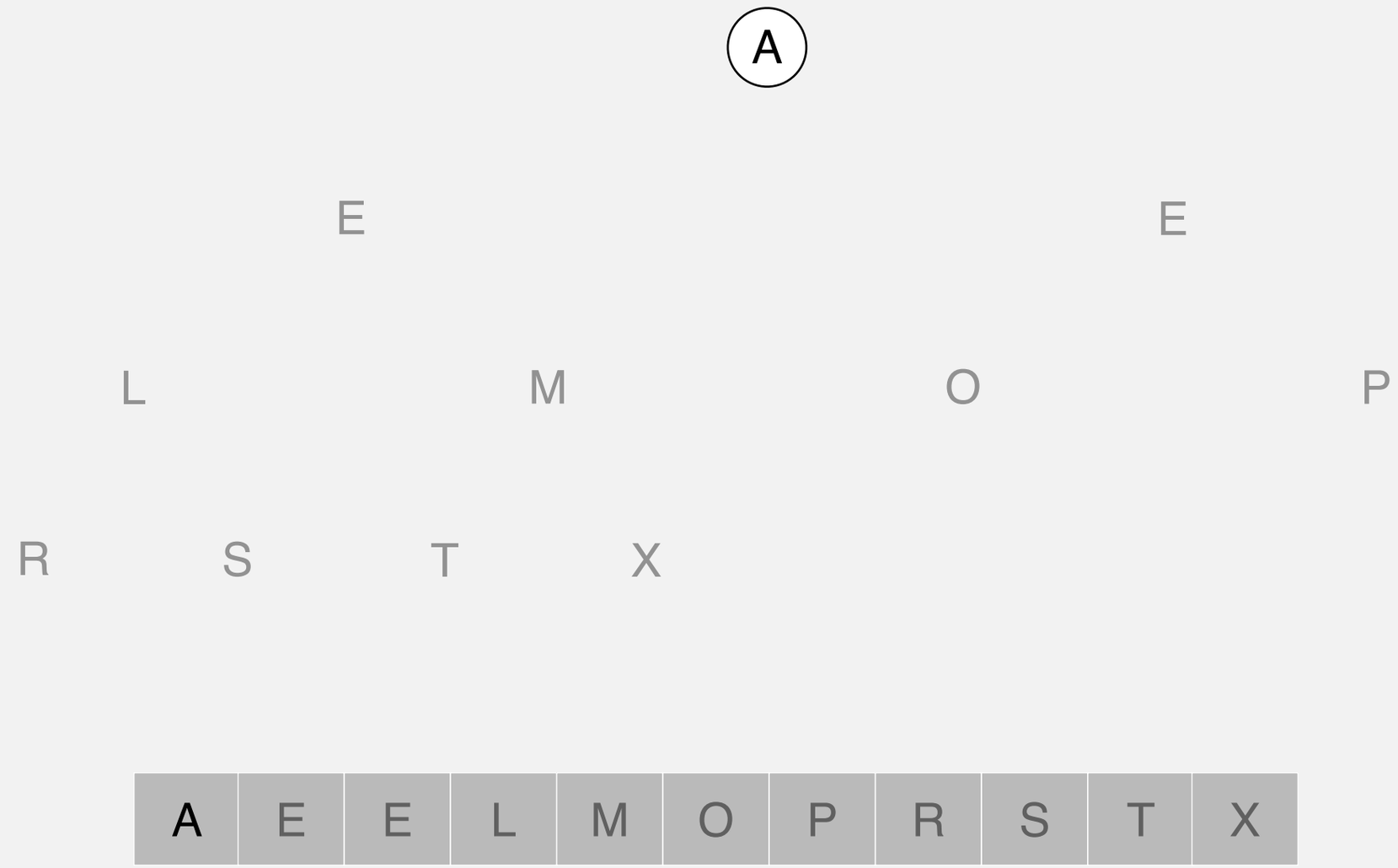
exchange 1 and 2



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

Heapsort

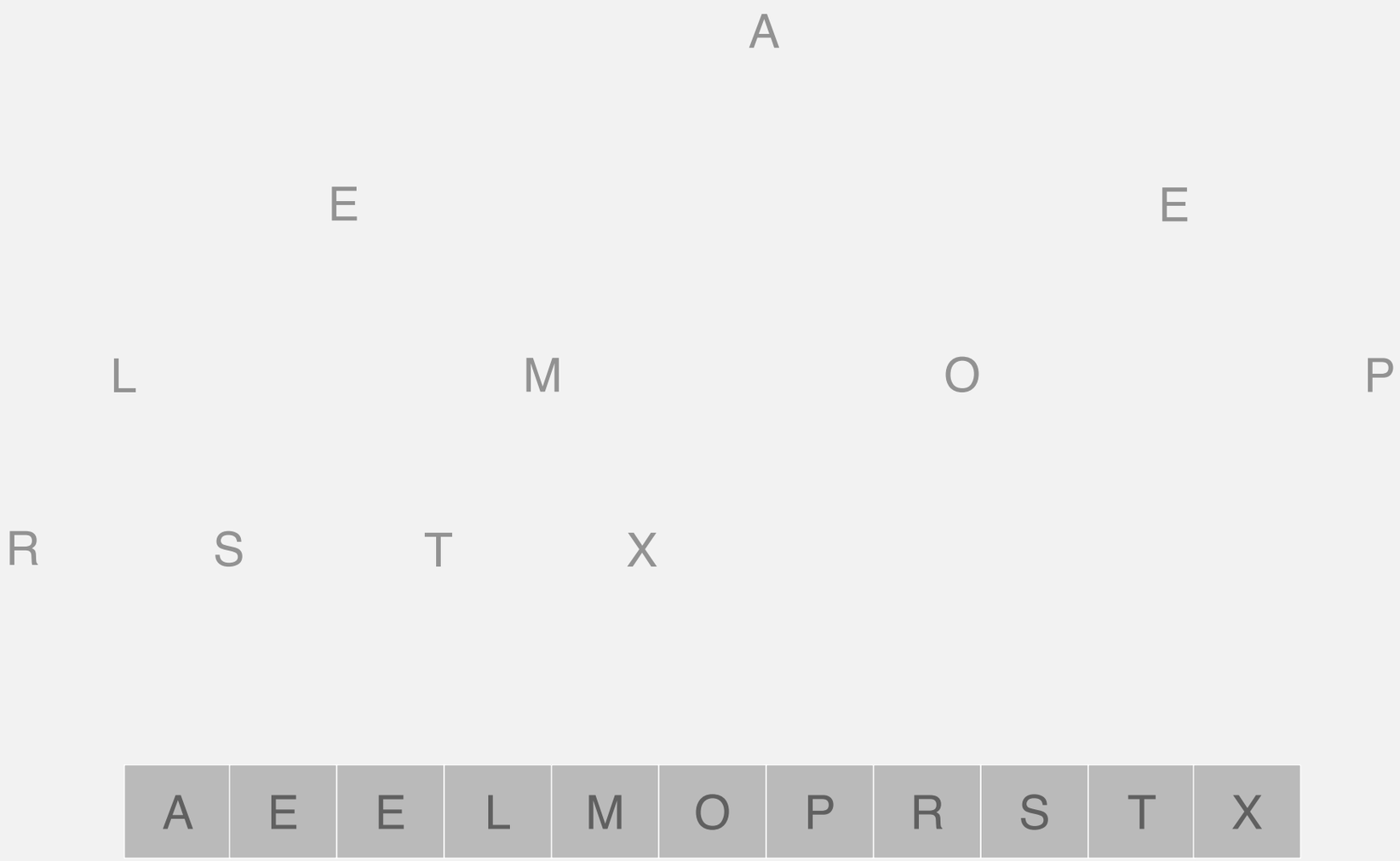
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

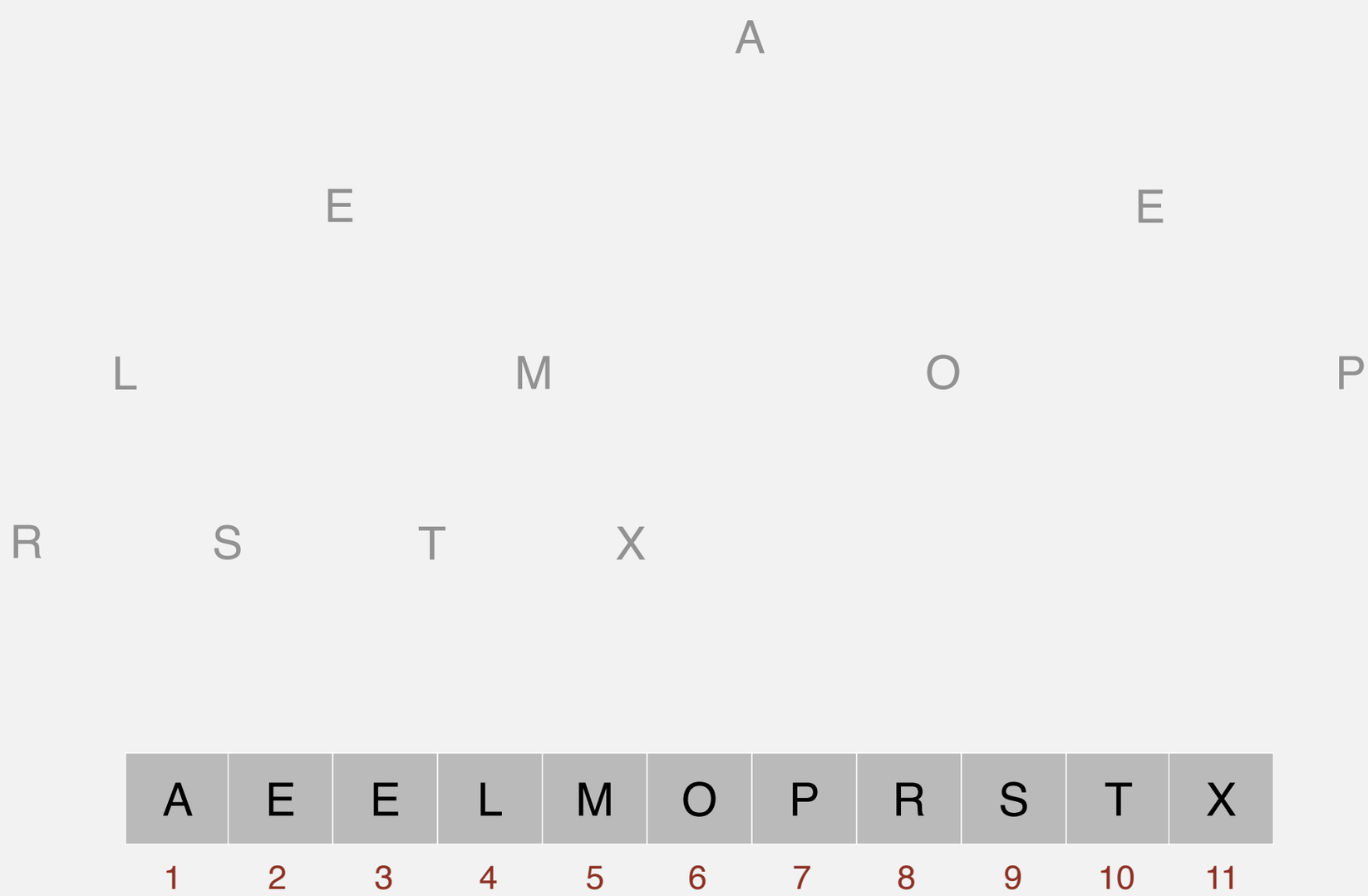
Sortdown. Repeatedly delete the largest remaining item.

end of sortdown phase



Heapsort

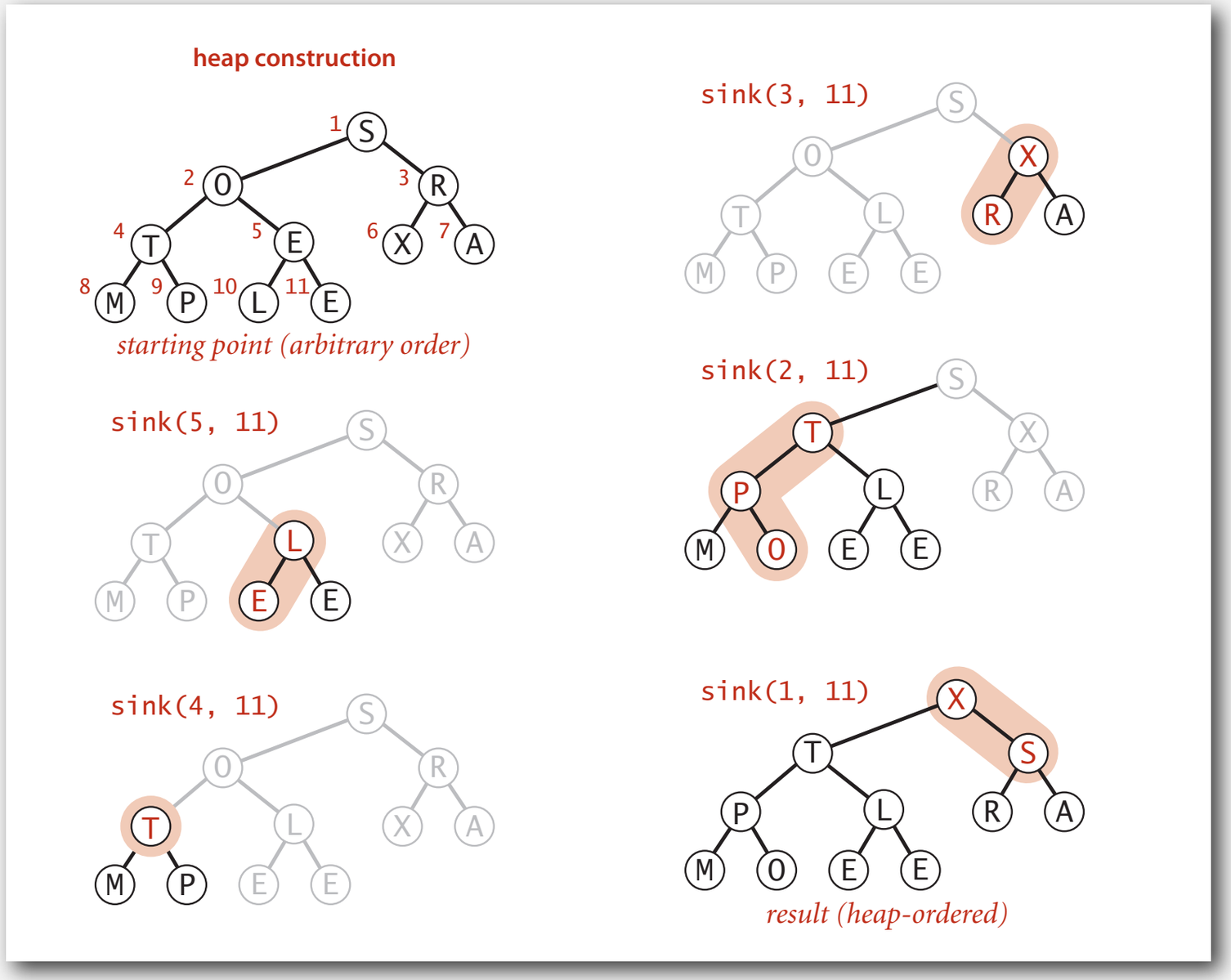
Ending point. Array in sorted order.



Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

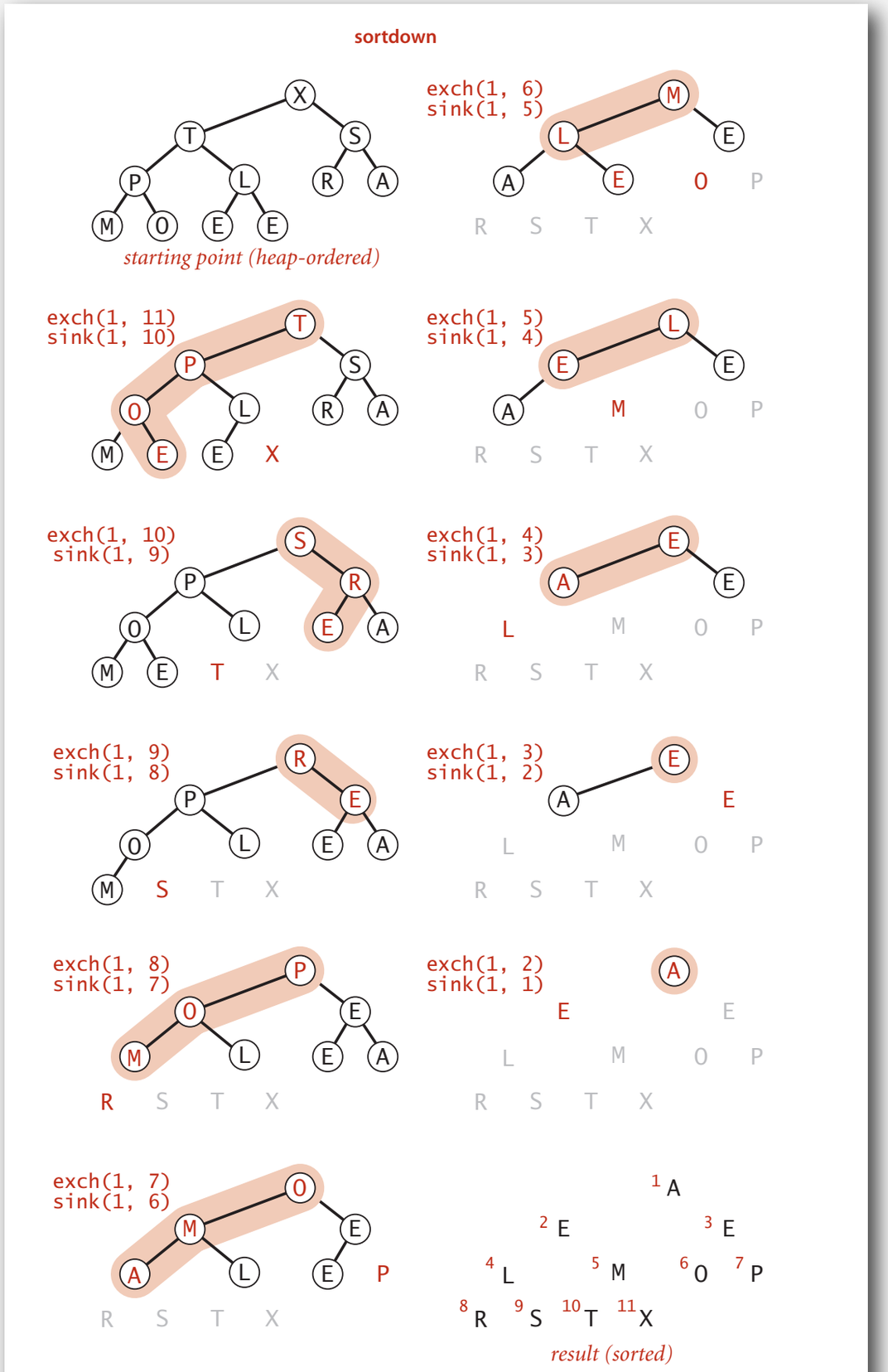


Heapsort: sortdown

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);
}
```



Heapsort: Java implementation

```
public class Heap
{
   public static void sort(Comparable[] pq)
   {
      int N = pq.length;
      for (int k = N/2; k >= 1; k--)
         sink(pq, k, N);
      while (N > 1)
      {
         exch(pq, 1, N);
         sink(pq, 1, --N);
      }
   }

   private static void sink(Comparable[] pq, int k, int N)
   {  /* as before */  }

   private static boolean less(Comparable[] pq, int i, int j)
   {  /* as before */  }

   private static void exch(Comparable[] pq, int i, int j)
   {  /* as before */  }

}
```

but convert from

1-based indexing to

0-base indexing

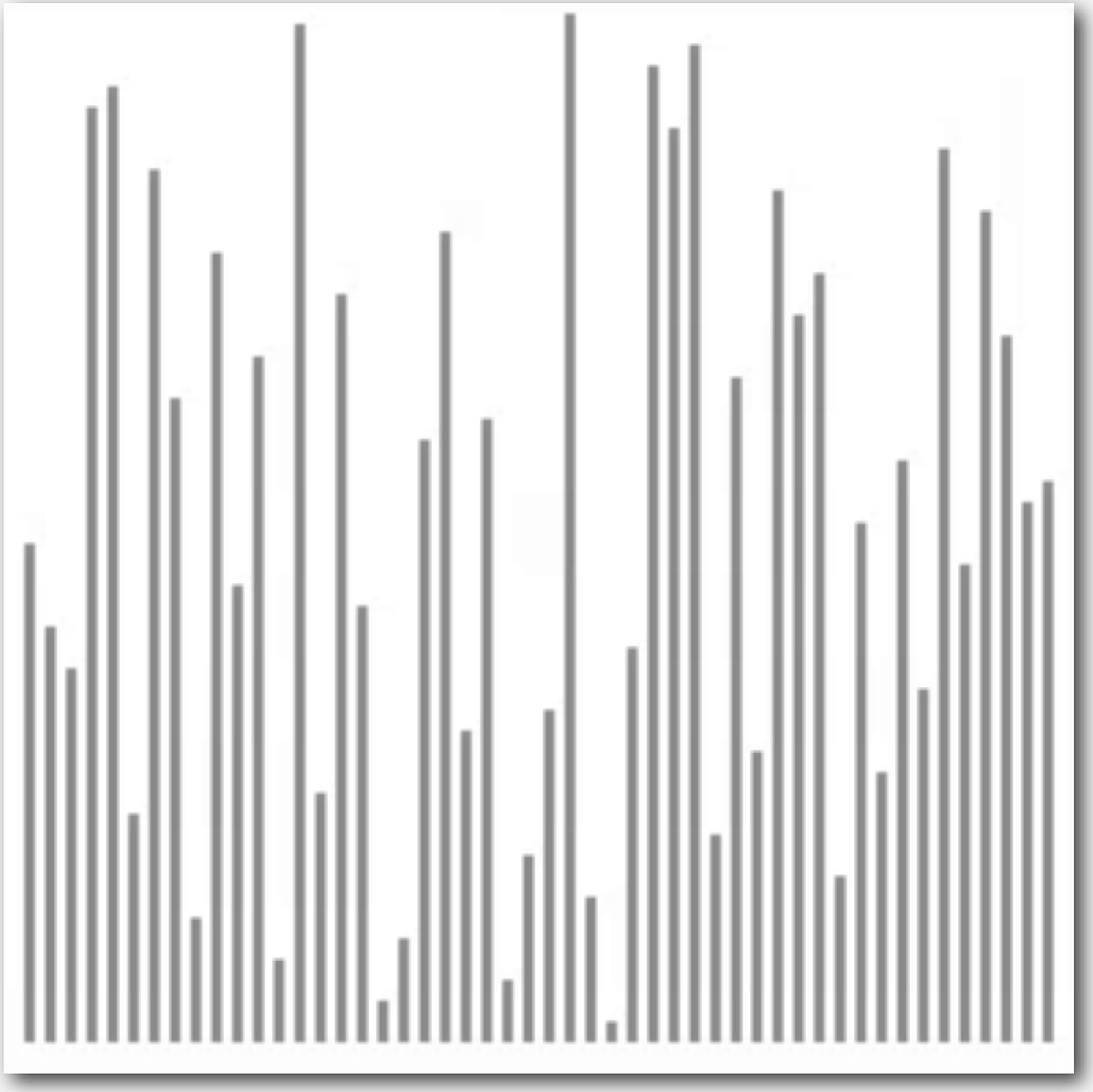
Heapsort: trace

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

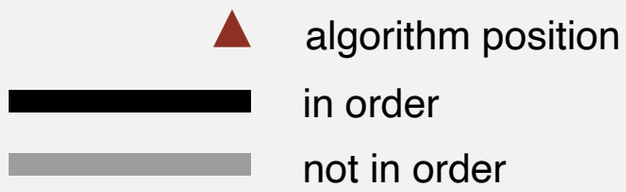
Heapsort trace (array contents just after each sink)

Heapsort animation

50 random items



http://www.sorting-algorithms.com/heap-sort



Heapsort: mathematical analysis

Proposition. Heap construction uses fewer than $2\,N$ compares and exchanges.

Proposition. Heapsort uses at most $2\,N \lg N$ compares and exchanges.

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 memory.
- Not stable.

Sorting algorithms: summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	N exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2 / 2$	2 N ln N	N lg N	N log N probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	2 N ln N	N	improves quicksort in presence of duplicate keys
merge		x	N lg N	N lg N	N lg N	N log N guarantee, stable
heap	x		2 N lg N	2 N lg N	N lg N	N log N guarantee, in-place
???	x	x	N lg N	N lg N	N lg N	holy sorting grail