



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

## 2.4 PRIORITY QUEUES

---

- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ ~~*event-driven simulation*~~  
*See video/book/booksite*



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

## 2.4 PRIORITY QUEUES

---

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

# Collections

---

A **collection** is a data type that stores a group of items.

data type	core operations	data structure
<b>stack</b>	PUSH, POP	<i>linked list, resizing array</i>
<b>queue</b>	ENQUEUE, DEQUEUE	<i>linked list, resizing array</i>
<b>priority queue</b>	INSERT, DELETE-MAX	<i>binary heap</i>
<b>symbol table</b>	PUT, GET, DELETE	<i>binary search tree, hash table</i>
<b>set</b>	ADD, CONTAINS, DELETE	<i>binary search tree, hash table</i>



# 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.

**Generalizes:** stack, queue, randomized queue.


<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.** Items are generic; they must also be Comparable.

Key must be Comparable  
(bounded type parameter)



```
public class MaxPQ<Key extends Comparable<Key>>
```

---

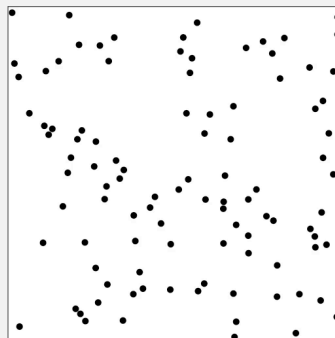
MaxPQ()	<i>create an empty priority queue</i>
MaxPQ(Key[] a)	<i>create a priority queue with given keys</i>
void insert(Key v)	<i>insert a key into the priority queue</i>
Key delMax()	<i>return and remove a largest key</i>
boolean isEmpty()	<i>is the priority queue empty?</i>
Key max()	<i>return a largest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

**Note.** Duplicate keys allowed; delMax() picks any maximum key.

# Priority queue: applications

---

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



8	4	7
1	5	6
3	2	

# 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.

**Q.** Would you use a MaxPQ or a MinPQ?

*Transaction data  
type is Comparable  
(ordered by \$\$)*

```
MinPQ<Transaction> pq = new MinPQ<Transaction>();

while (StdIn.hasNextLine())
{
    String line = StdIn.readLine();
    Transaction transaction = new Transaction(line);
    pq.insert(transaction);
    if (pq.size() > M)
        pq.delMin();
}

```

*← pq now contains  
largest M items*

# Priority queue: unordered and ordered array implementation

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

A sequence of operations on a priority queue



# Priority queue: implementations cost summary

---

**Challenge.** Implement **all** operations efficiently.

implementation	insert	del max	max
<b>unordered array</b>	1	$N$	$N$
<b>ordered array</b>	$N$	1	1
<b>goal</b>	$\log N$	$\log N$	$\log N$

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



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

## 2.4 PRIORITY QUEUES

---

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



# A complete binary tree in nature

---



Hyphaene Compressa - Doum Palm

© Shlomit Pinter

# Binary heap: representation

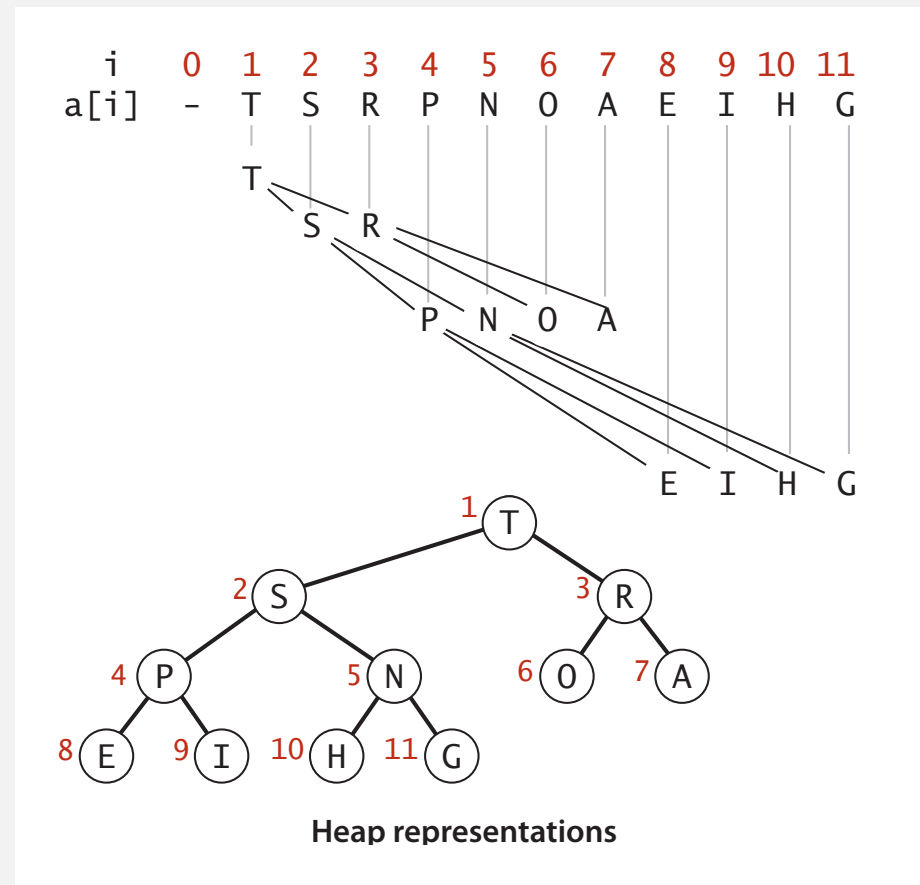
**Binary heap.** Array representation of a heap-ordered complete binary tree.

**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!

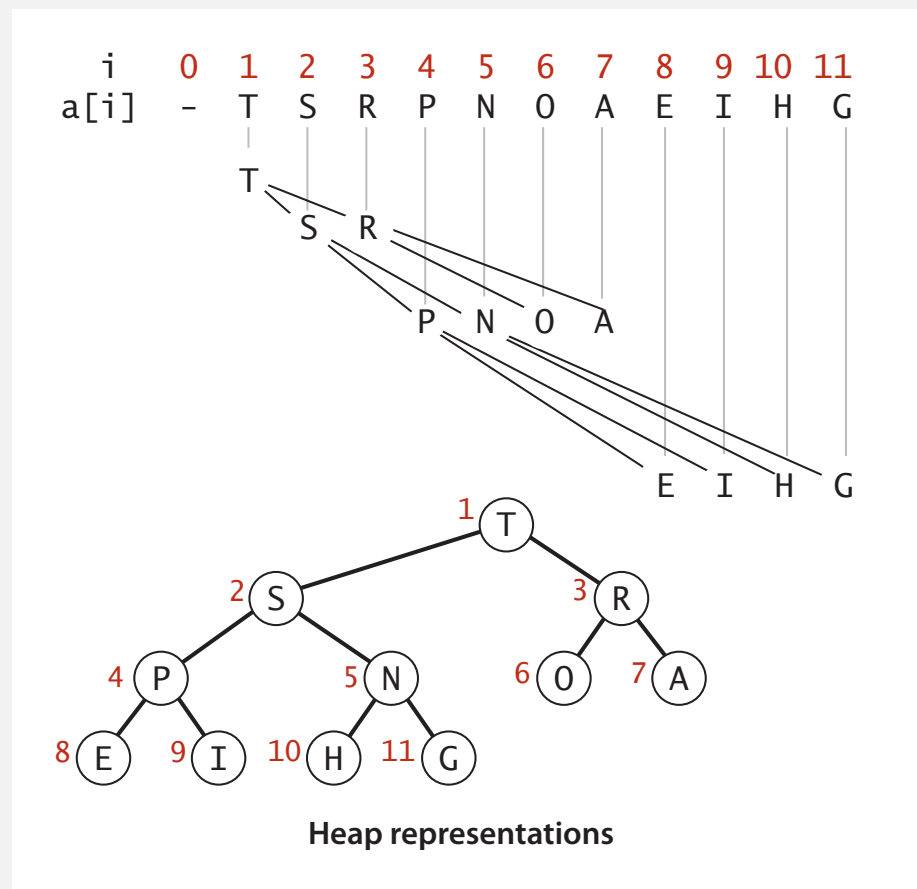


# Binary heap: properties

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

**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  $2k+1$ .



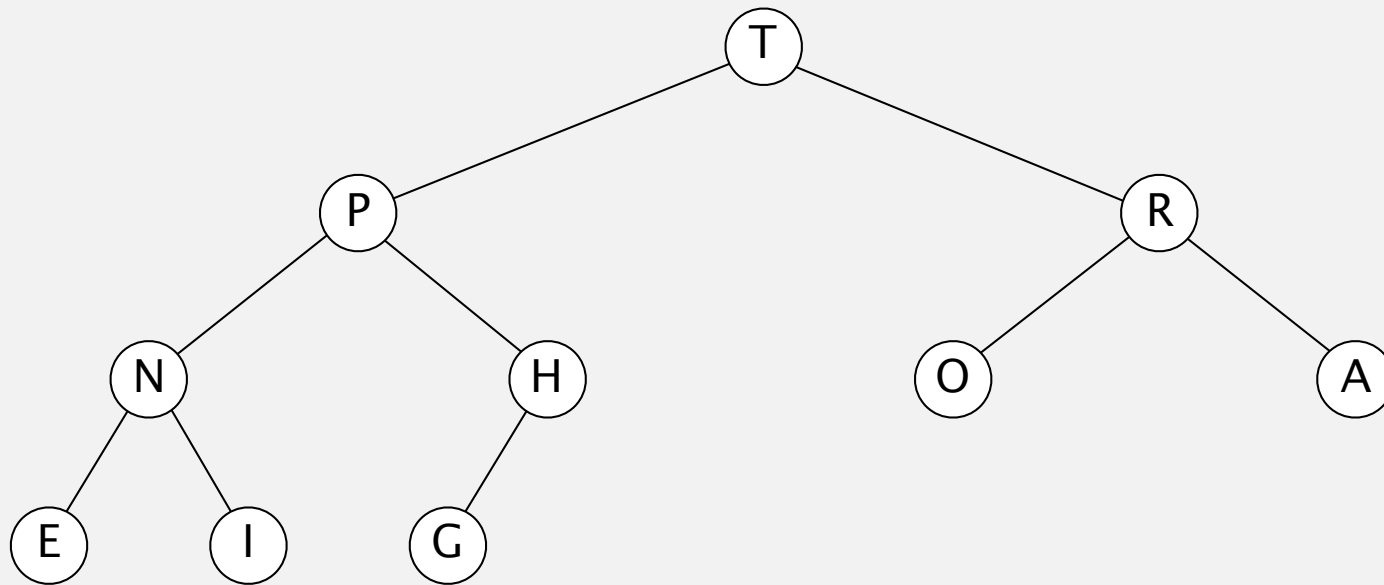
# Binary heap demo

---

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

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

heap ordered



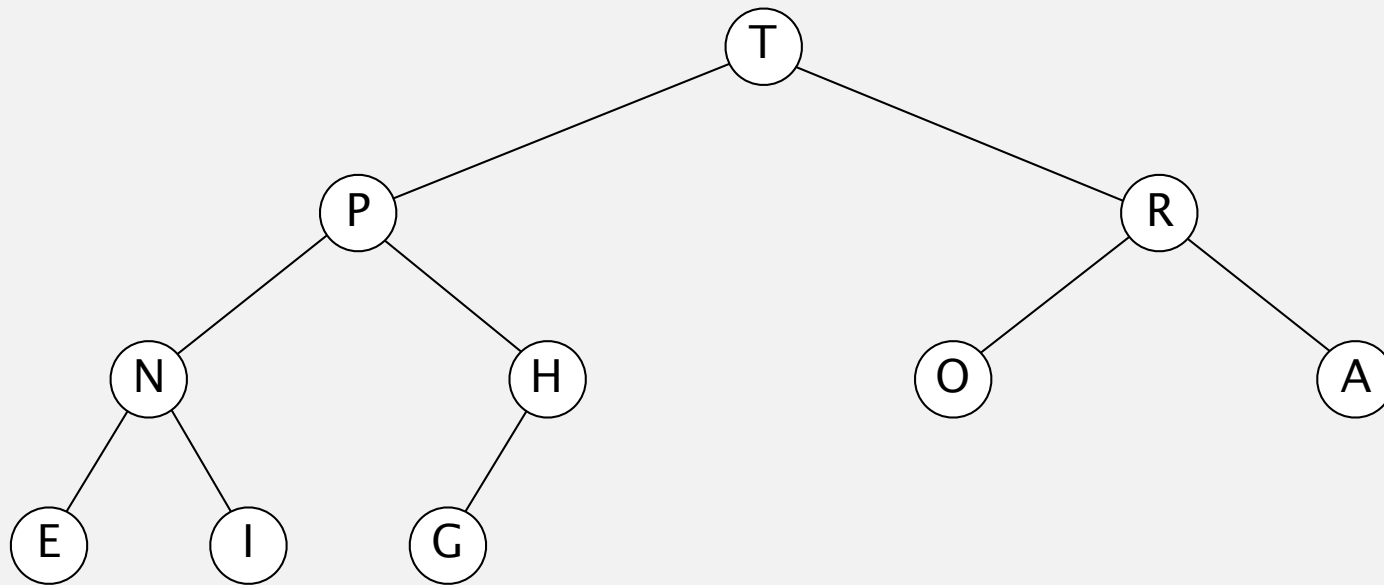
# Binary heap demo

---

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

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

heap ordered





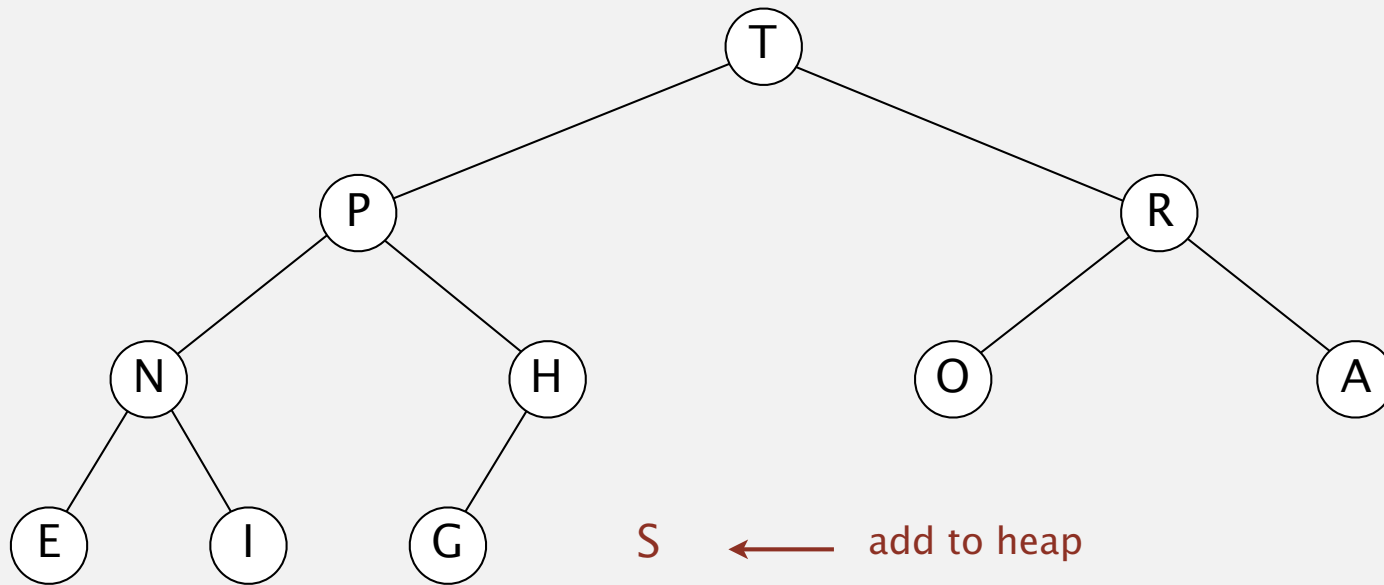
# Binary heap demo

---

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

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

**insert S**



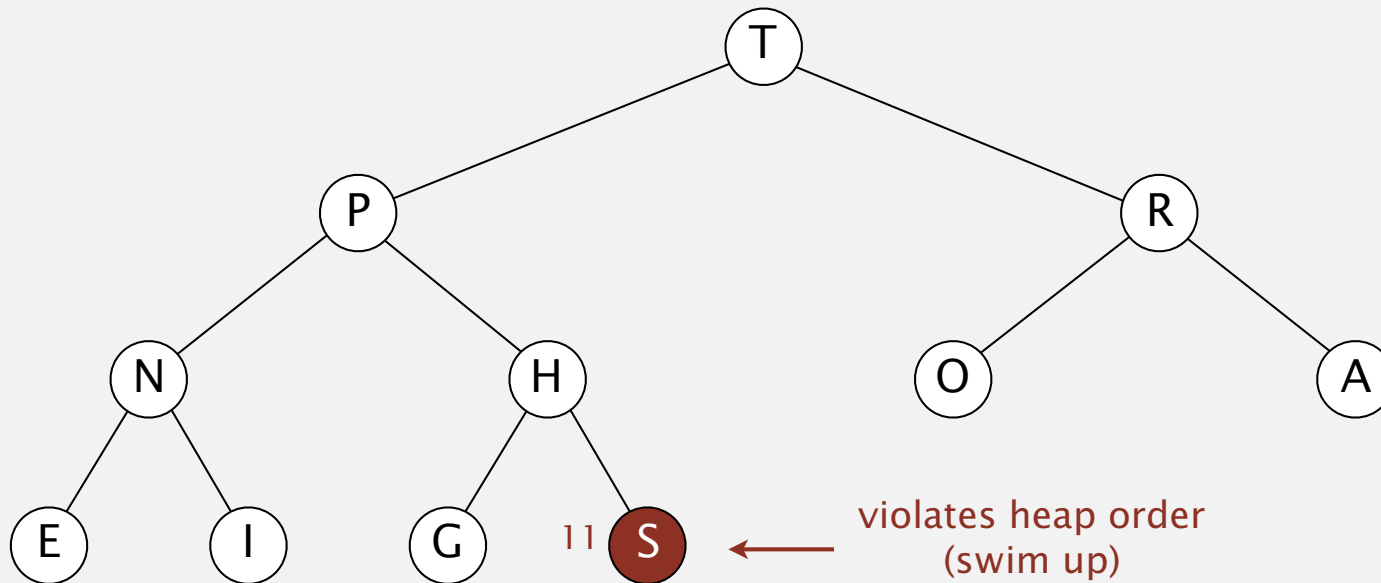
# Binary heap demo

---

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

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

insert S



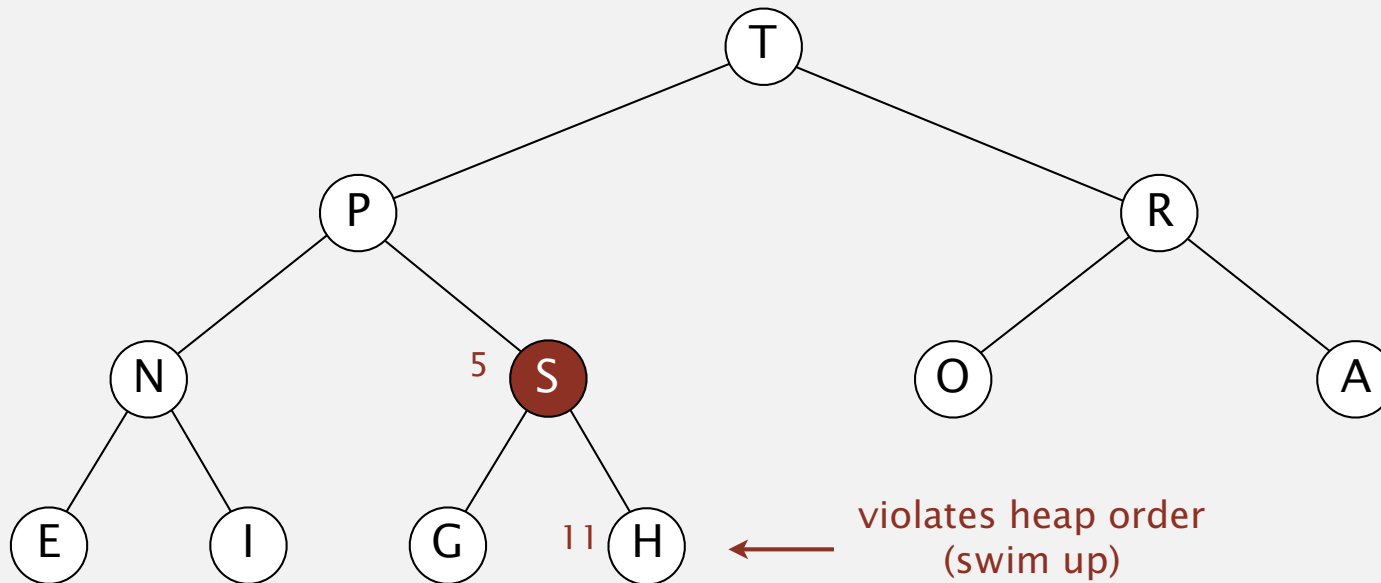
# Binary heap demo

---

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

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

insert S



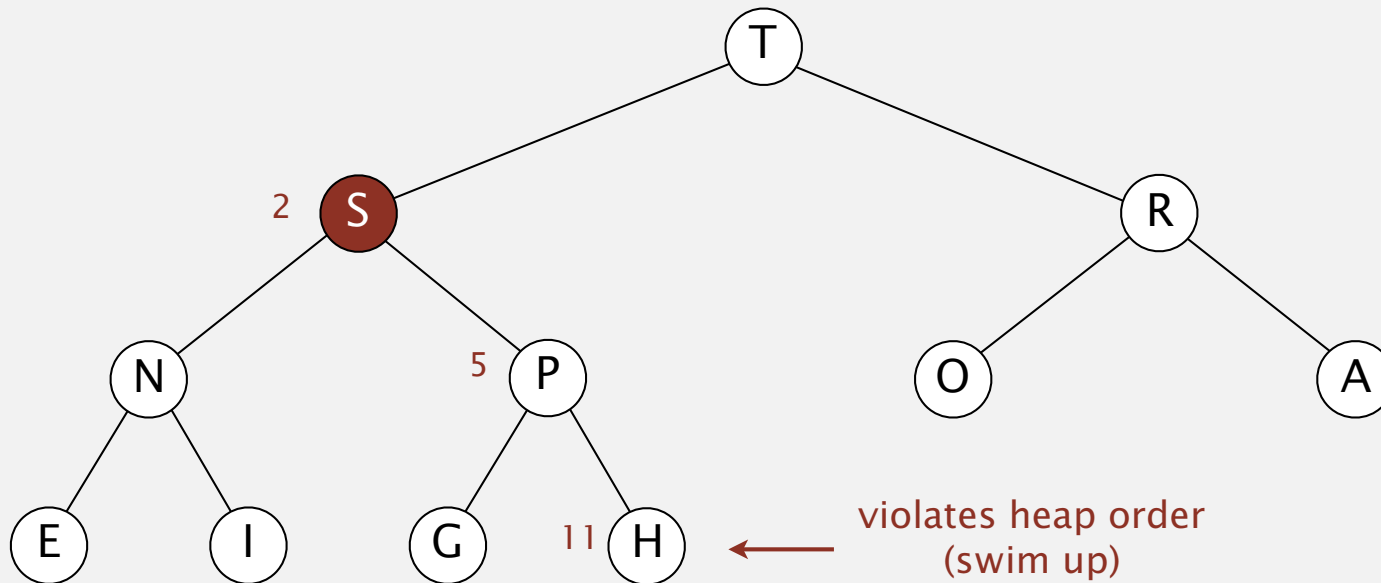
# Binary heap demo

---

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

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

insert S



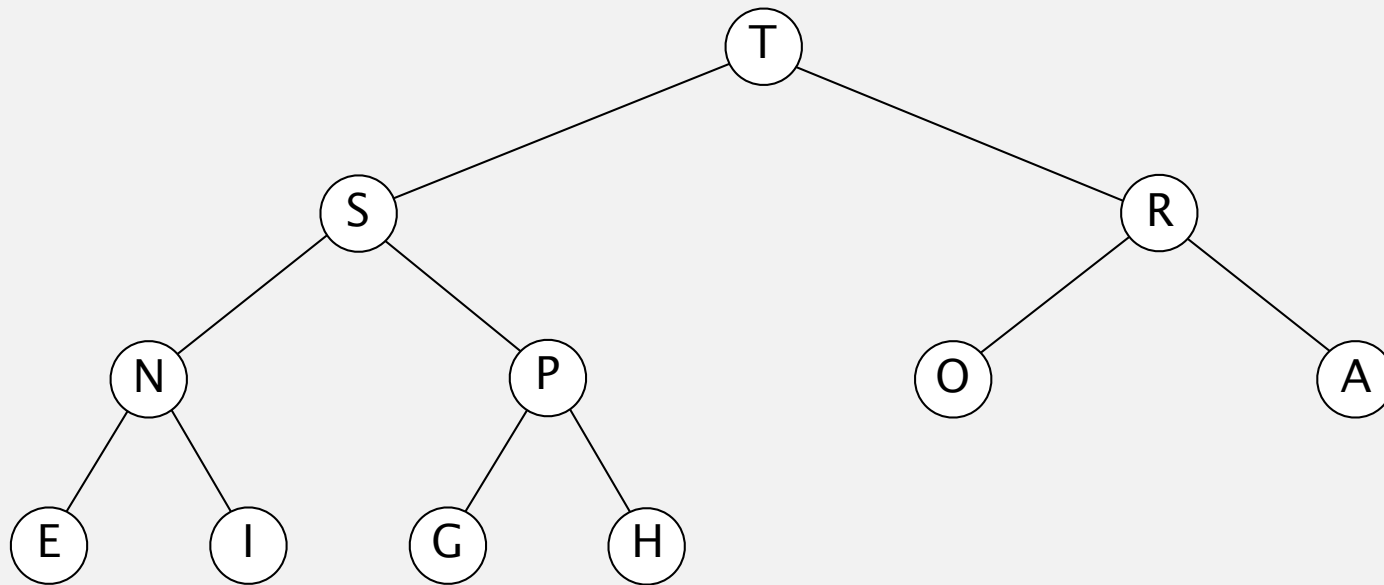
# Binary heap demo

---

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

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

heap ordered



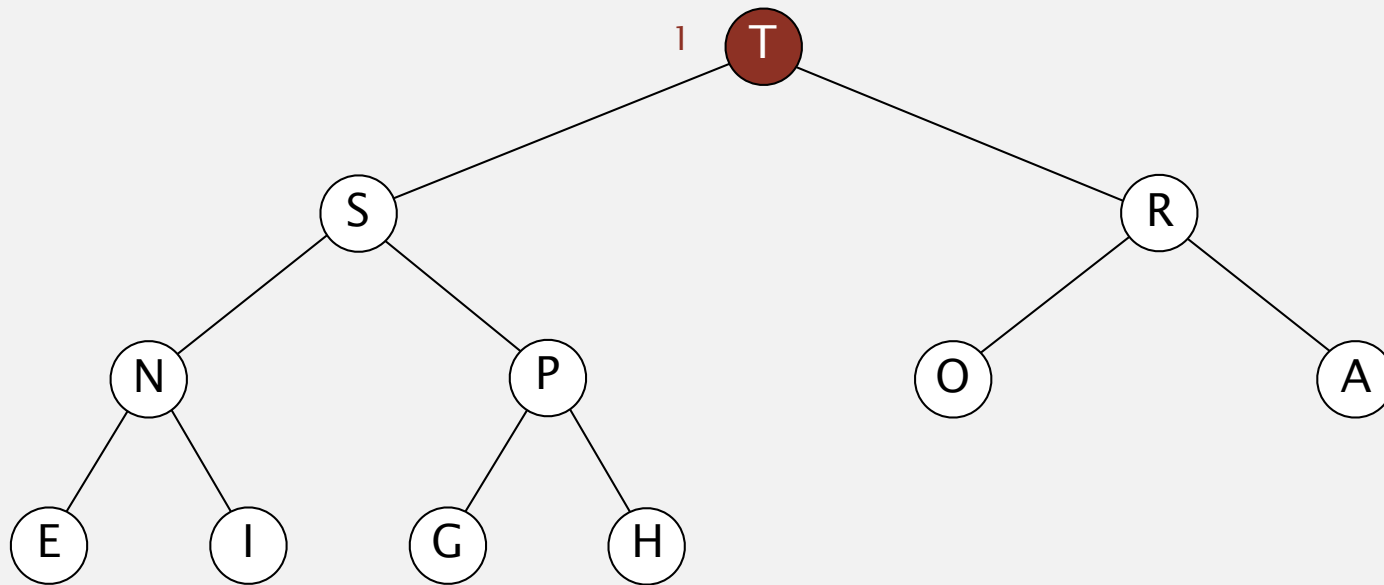
# Binary heap demo

---

**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**









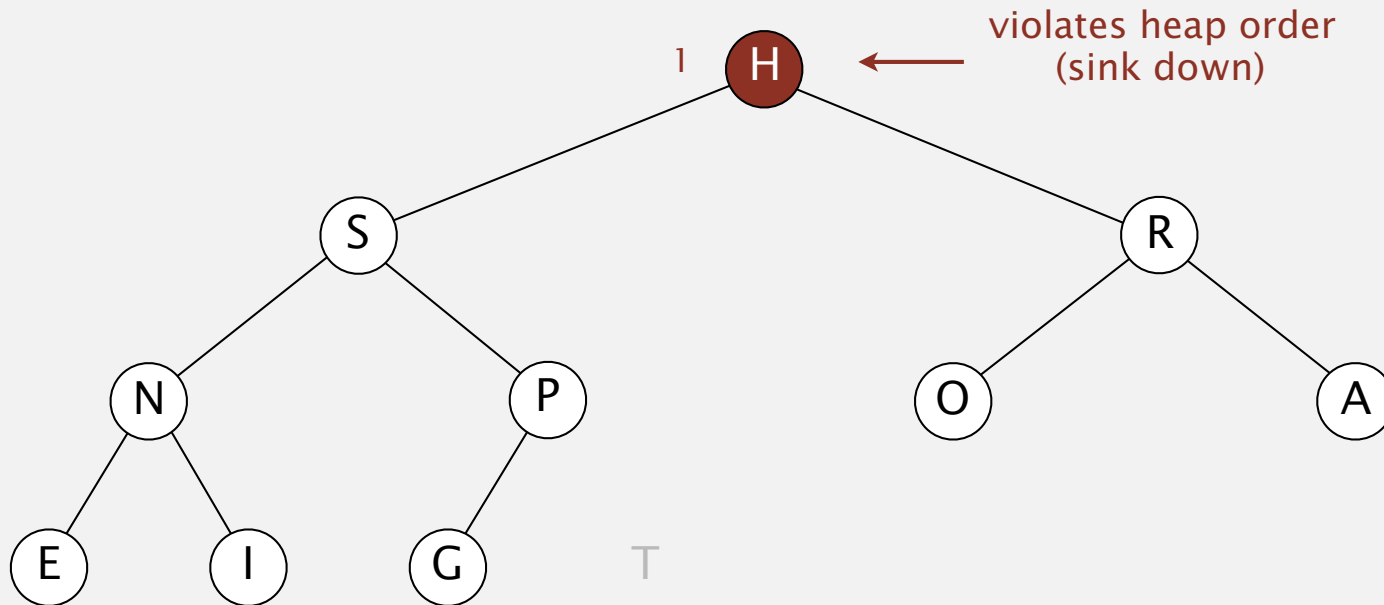
# Binary heap demo

---

**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**





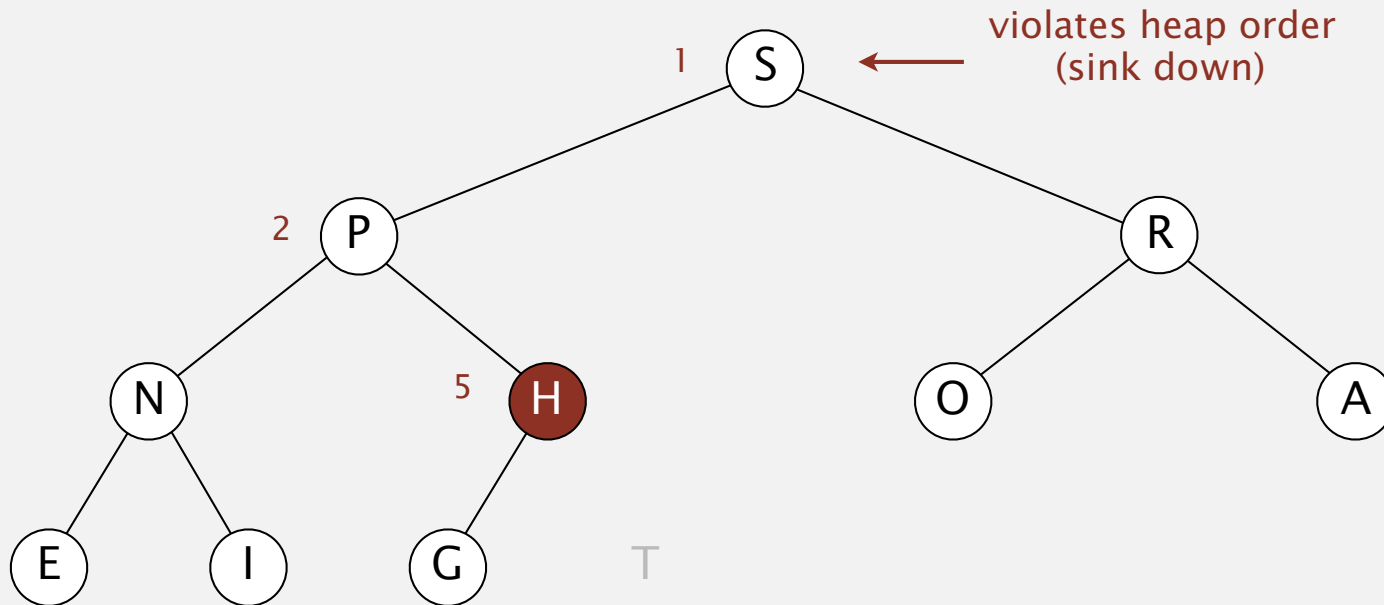
# Binary heap demo

---

**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**



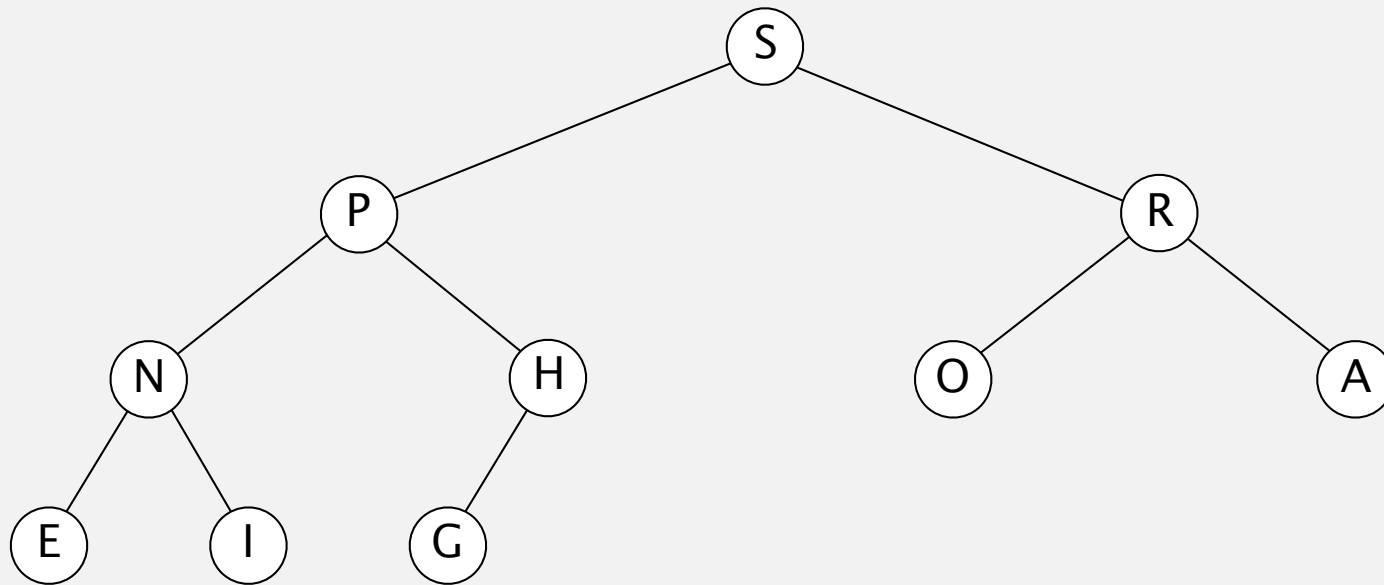
# Binary heap demo

---

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

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

heap ordered



# Binary heap: promotion

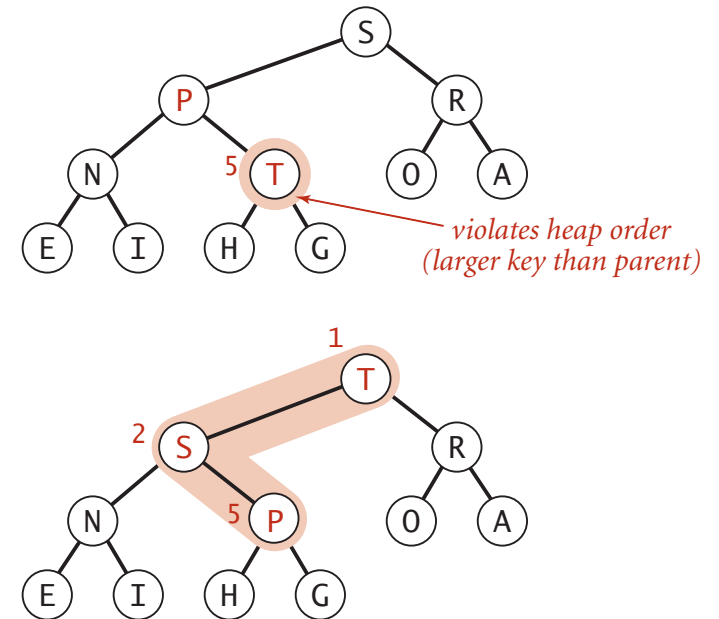
**Scenario.** A key becomes **larger** than its parent's key.

**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



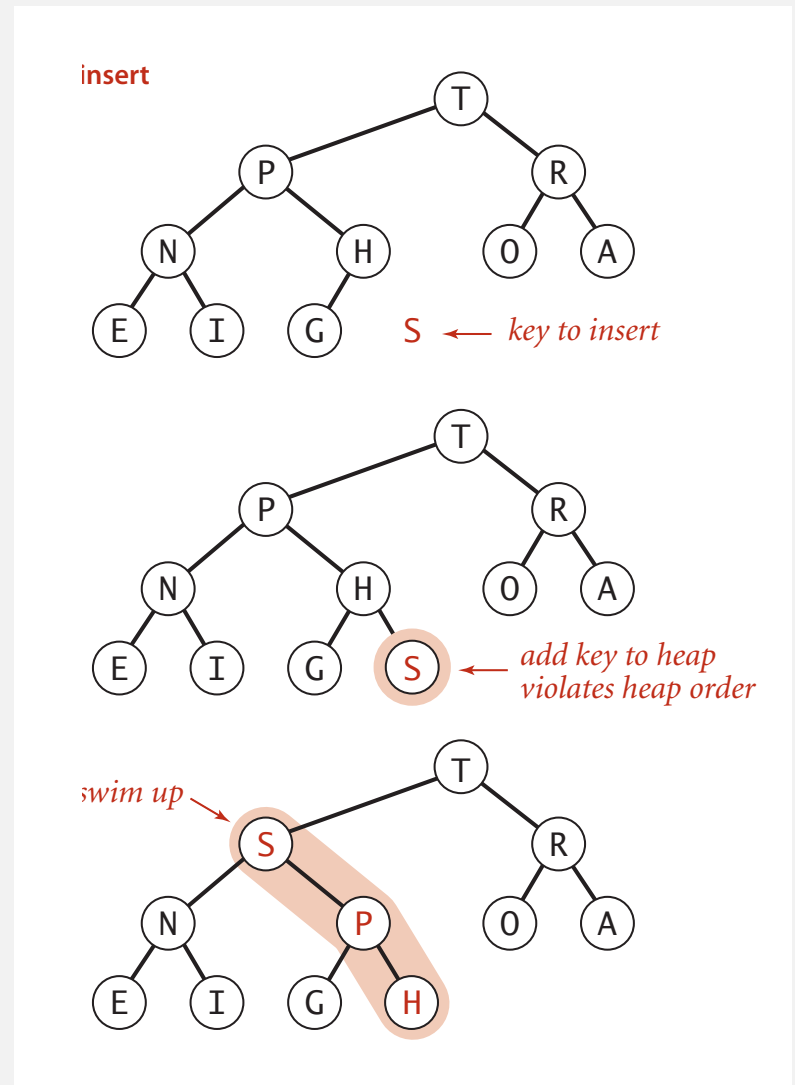
**Peter principle.** Node promoted to level of incompetence.

# Binary heap: insertion

**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);
}
```



# Binary heap: demotion

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

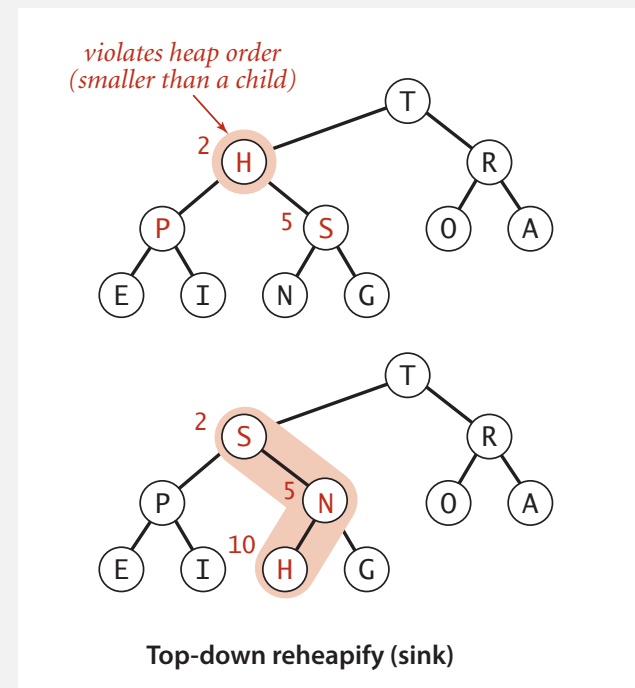
To eliminate the violation:

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  $2*k$  and  $2*k+1$



**Power struggle.** Better subordinate promoted.

## Binary heap: demotion

---

Q. Write a recursive version of sink

```
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;
    }
}
```

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

This is just an exercise. No particular reason to implement this recursively.

In fact, many compilers will *automatically* convert the recursive version to the iterative one. This is called tail-call elimination or tail-call optimization.

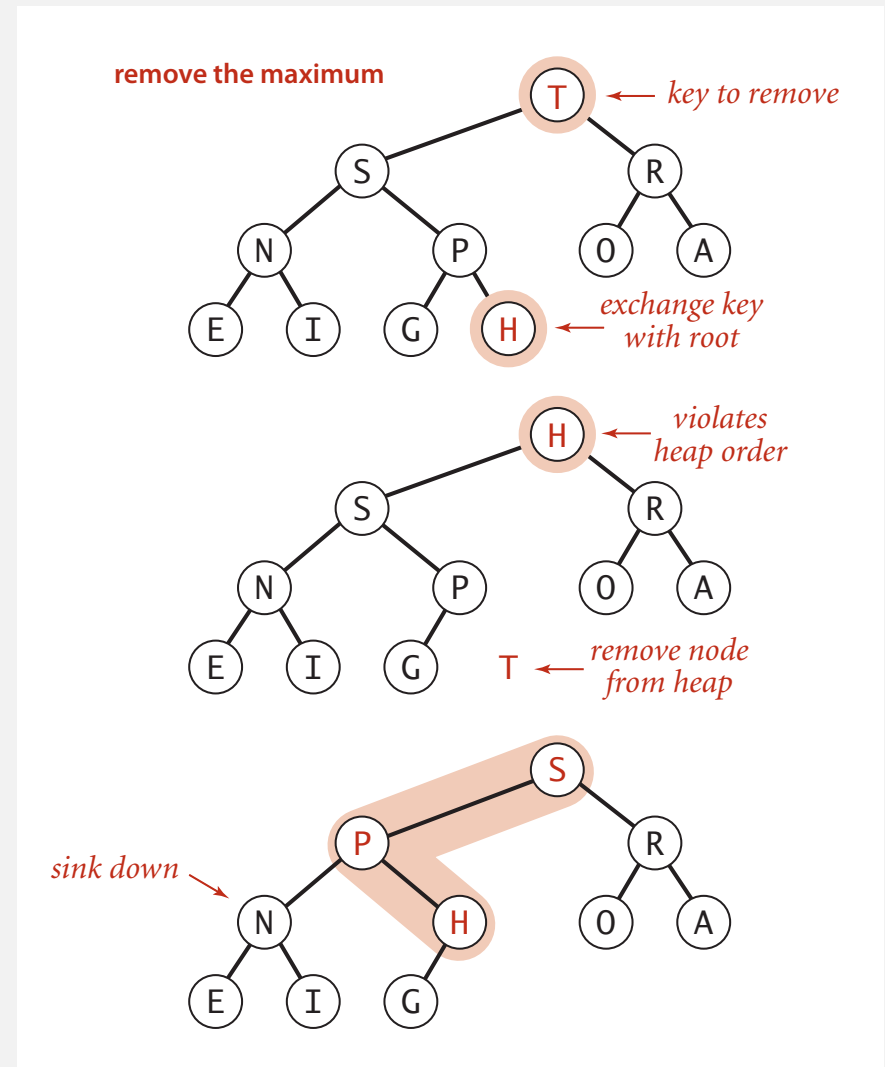


# Binary heap: delete the maximum

**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; ← prevent loitering
    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) // see previous code
    public Key delMax() // see previous code
```

← PQ ops

```
    private void swim(int k) // see previous code
    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 queue: implementations cost summary

---

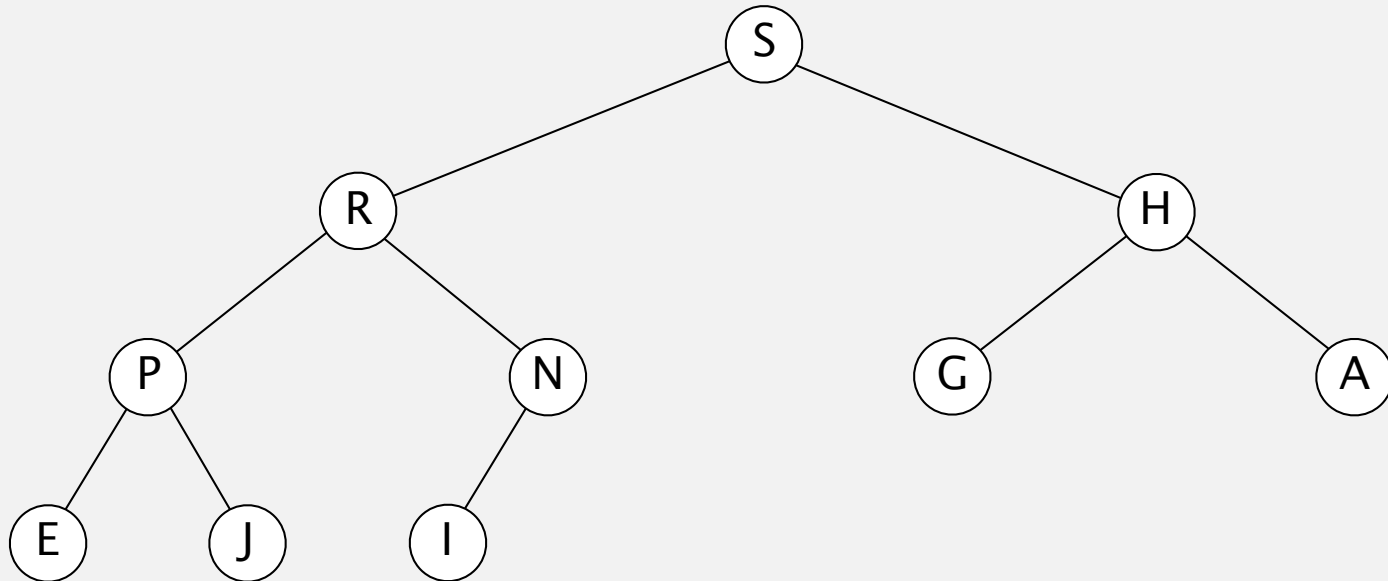
implementation	insert	del max	max
<b>unordered array</b>	1	$N$	$N$
<b>ordered array</b>	$N$	1	1
<b>binary heap</b>	$\log N$	$\log N$	1

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

# Delete-random from a binary heap

---

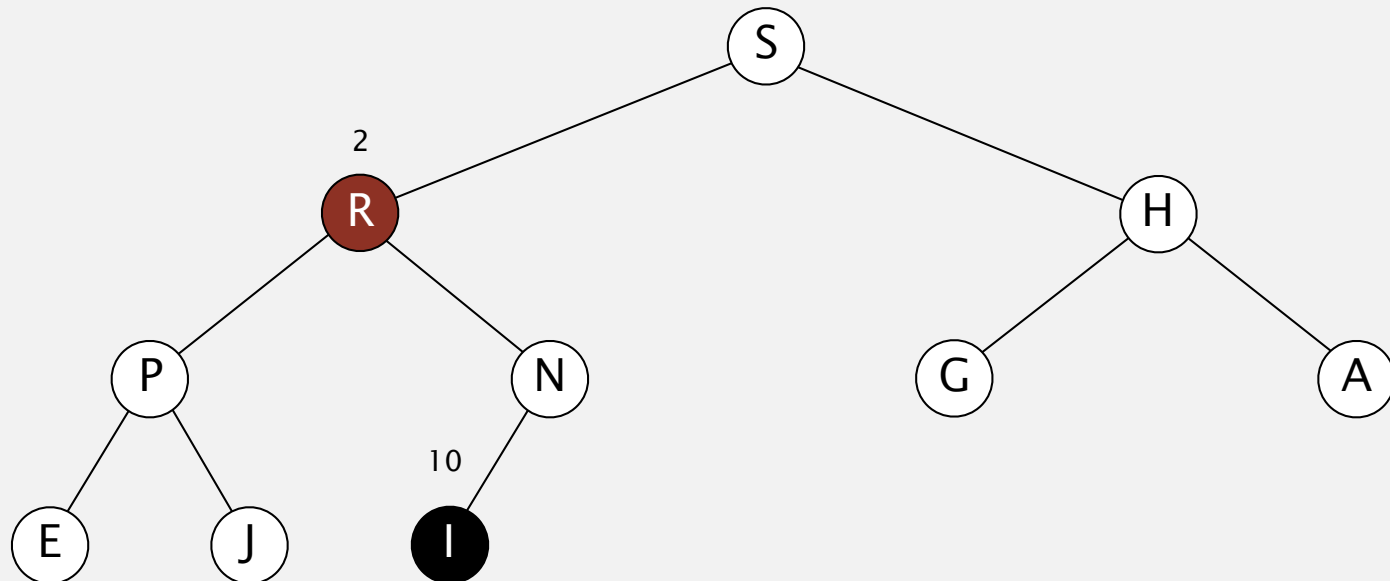
**Problem.** Delete a random key from a binary heap in logarithmic time.



# Delete-random from a binary heap

---

**Problem.** Delete a random key from a binary heap in logarithmic time.



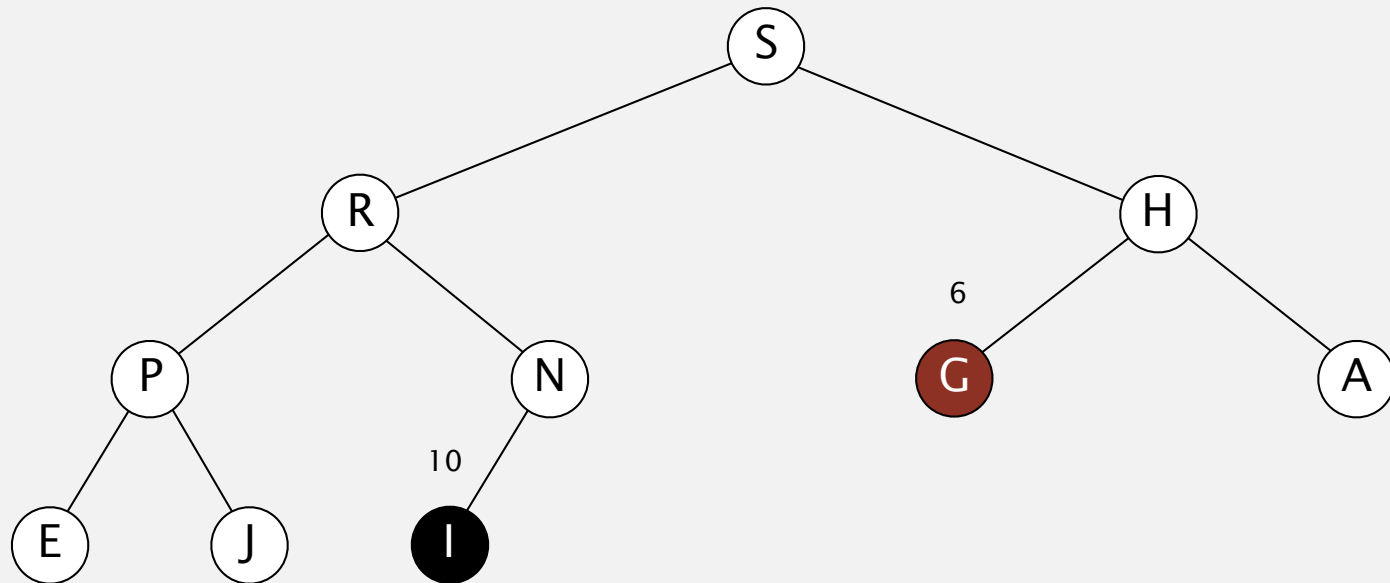
**Solution.**

- Pick a random index  $r$  between 1 and  $N$ .
- Perform  $\text{exch}(r, N--)$ .
- Perform either  $\text{sink}(r)$  or  $\text{swim}(r)$ .

# Delete-random from a binary heap

---

**Problem.** Delete a random key from a binary heap in logarithmic time.



**Solution.**

- Pick a random index  $r$  between 1 and  $N$ .
- Perform  $\text{exch}(r, N--)$ .
- Perform either  $\text{sink}(r)$  or  $\text{swim}(r)$ .

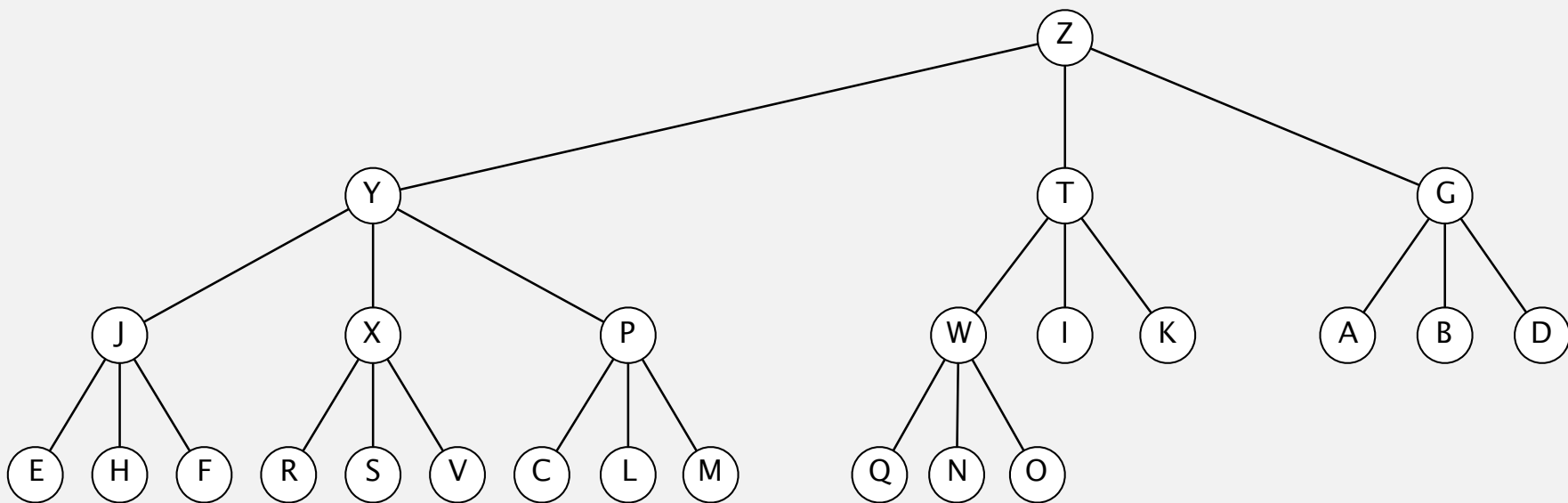
# Binary heap: practical improvements

---

## Multiway heaps.

- Complete  $d$ -way tree.
- Parent's key no smaller than its children's keys.

**Fact.** Height of complete  $d$ -way tree on  $N$  nodes is  $\sim \log_d N$ .



3-way heap

# Priority queue: implementation cost summary

---

implementation	insert	del max	max
<b>unordered array</b>	1	$N$	$N$
<b>ordered array</b>	$N$	1	1
<b>binary heap</b>	$\log N$	$\log N$	1
<b>d-ary heap</b>	$\log_d N$	$d \log_d N$	1
<b>Fibonacci</b>	1	$\log N^\dagger$	1
<b>Brodal queue</b>	1	$\log N$	1
<b>impossible</b>	1	1	1

← sweet spot:  $d = 4$

← why impossible?

† amortized

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: use resizing array.

## 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()`

## Immutability of keys.

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

# Immutability: implementing in Java

---

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

**Examples:** String, Integer, Double, Color, Vector, Transaction, Point2D.

**Mutable:** StringBuilder, Stack, Counter, Java array.

To create your own immutable data types:

- Make defensive copy of client-provided mutable variables in constructor
- Don't change instance variables in instance methods

# 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.

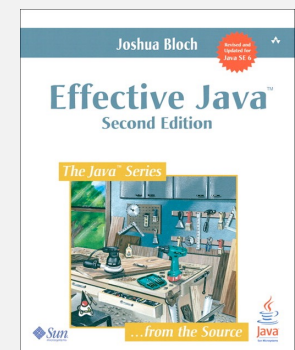
- Simplifies debugging.
- Simplifies concurrent programming.
- More secure in presence of hostile code.
- Safe to use as key in priority queue or symbol table.



**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)*





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

## 2.4 PRIORITY QUEUES

---

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

## Priority queues: quiz 4

---

Verify that this is a correct sorting algorithm. What are its properties?

```
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();
}
```

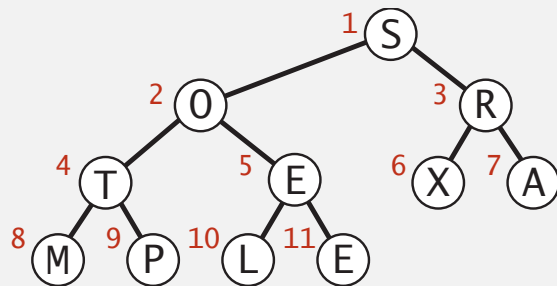
- A.**  $N \log N$  compares in the worst case.
- B.** In-place.
- C.** Stable.
- D.** *All of the above.*
- E.** *I don't know.*

# Heapsort

## Basic plan for in-place sort.

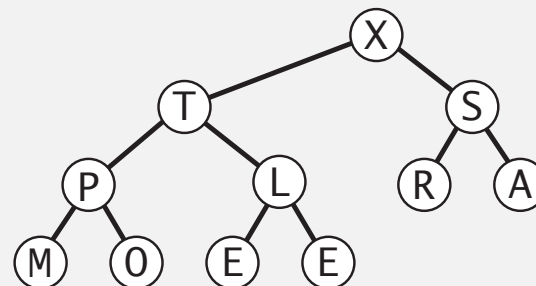
- View input array as a complete binary tree.
- Heap construction: build a max-heap with all  $N$  keys.
- Sortdown: repeatedly remove the maximum key.

keys in arbitrary order



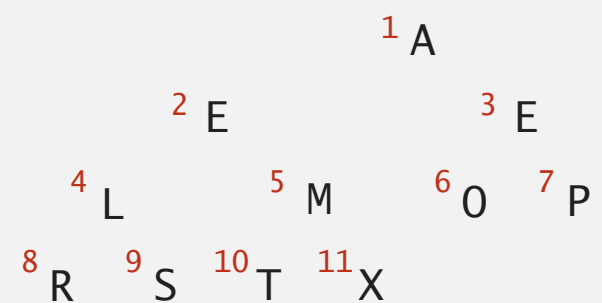
1	2	3	4	5	6	7	8	9	10	11
S	O	R	T	E	X	A	M	P	L	E

build max heap  
(in place)



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

sorted result  
(in place)



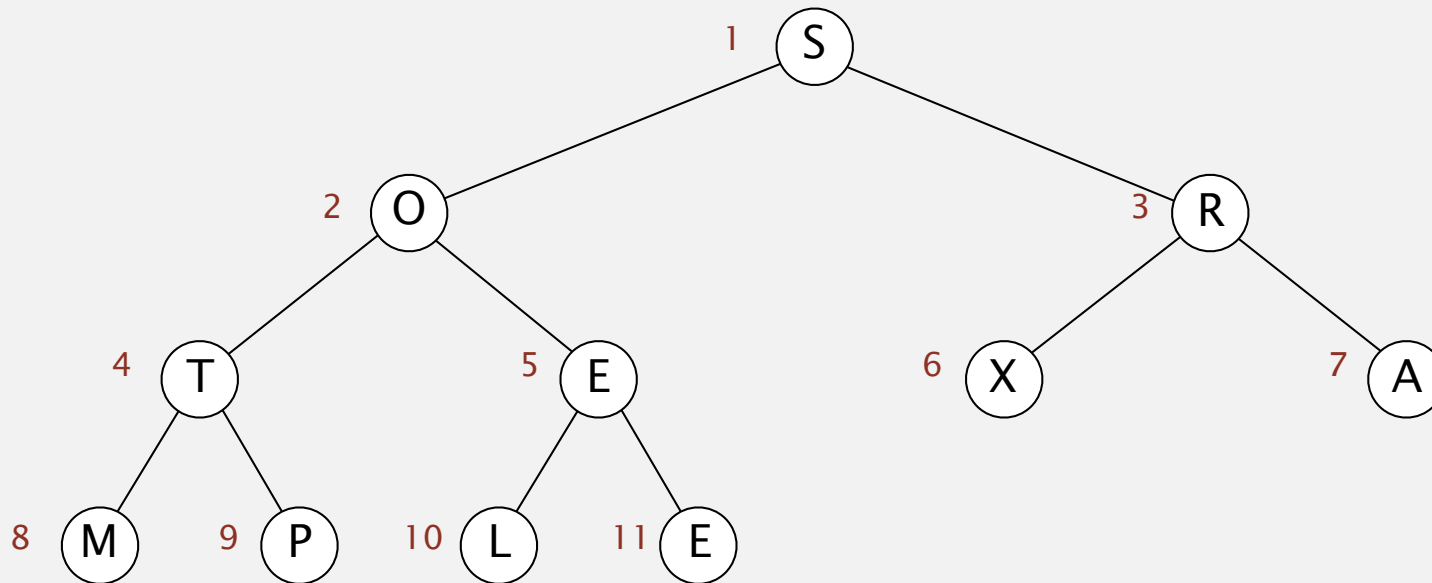
1	2	3	4	5	6	7	8	9	10	11
A	E	E	L	M	O	P	R	S	T	X

# Heapsort demo

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

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

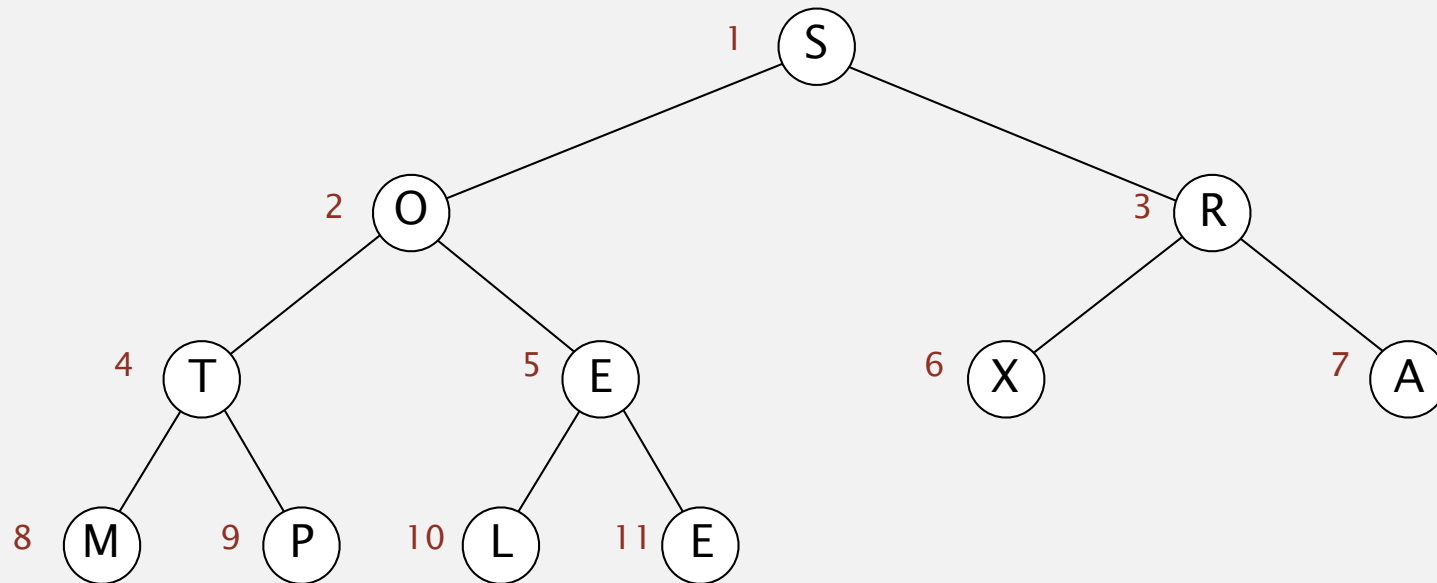
---

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



we assume array entries are indexed 1 to N

array in arbitrary order



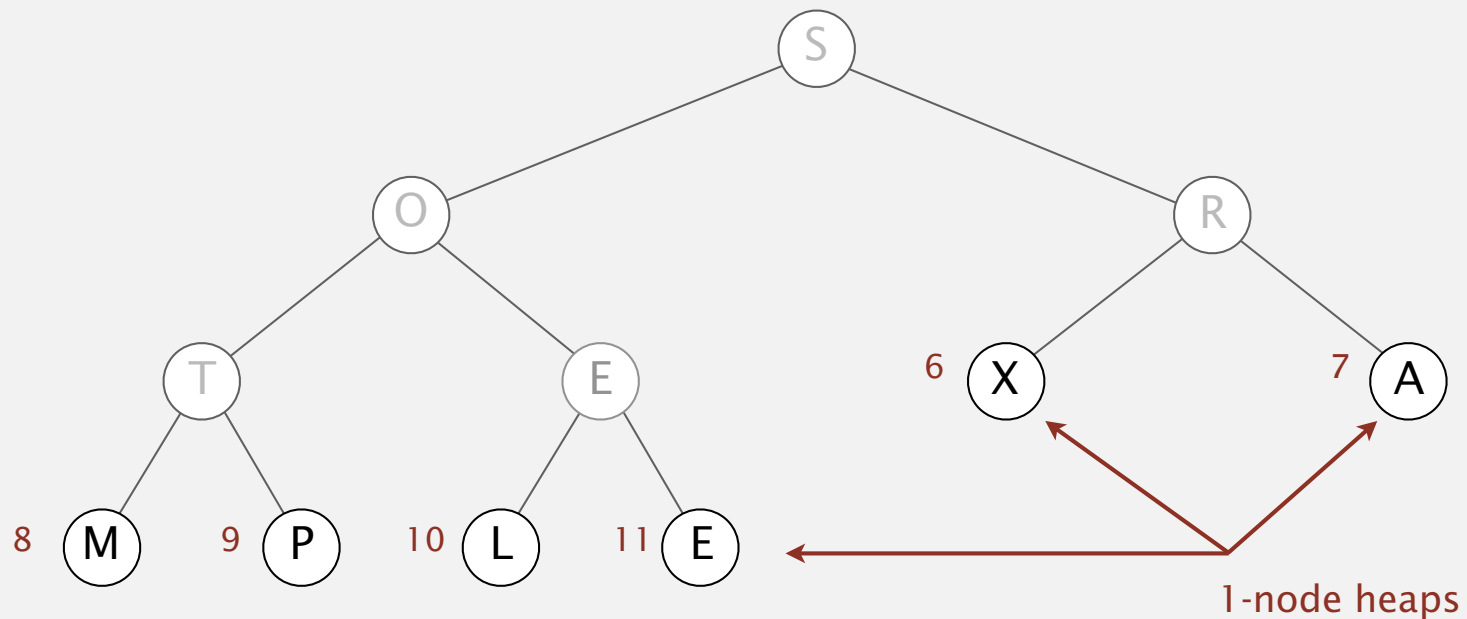
S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11



# Heapsort demo

---

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

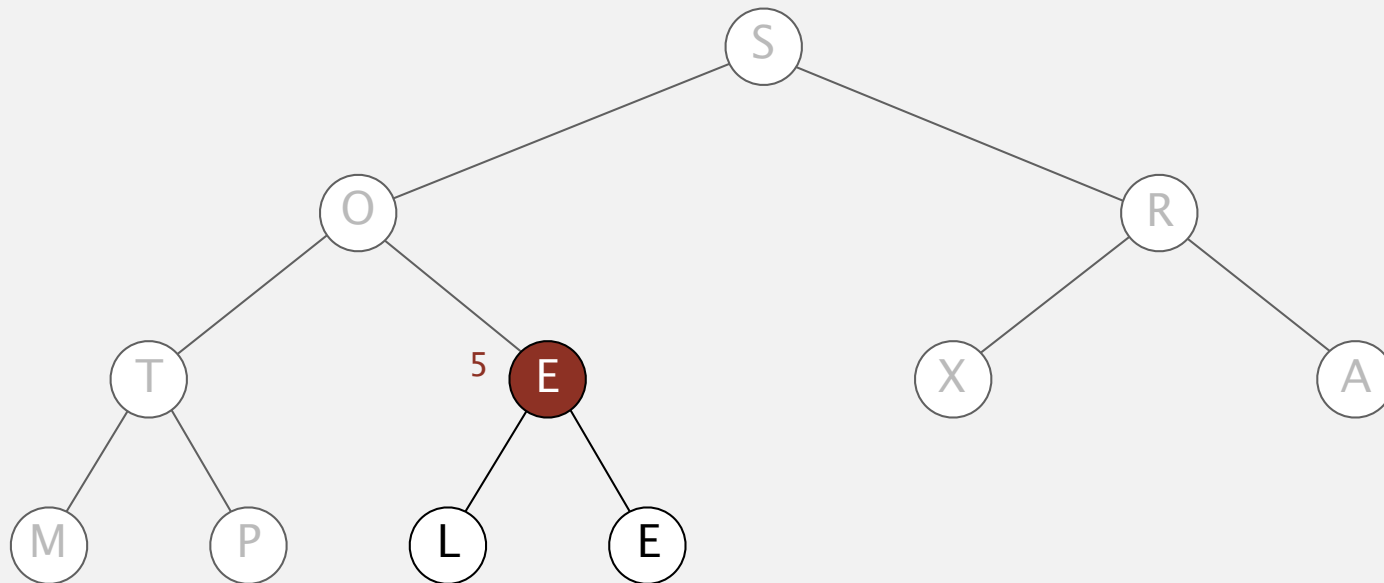


# Heapsort demo

---

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

sink 5

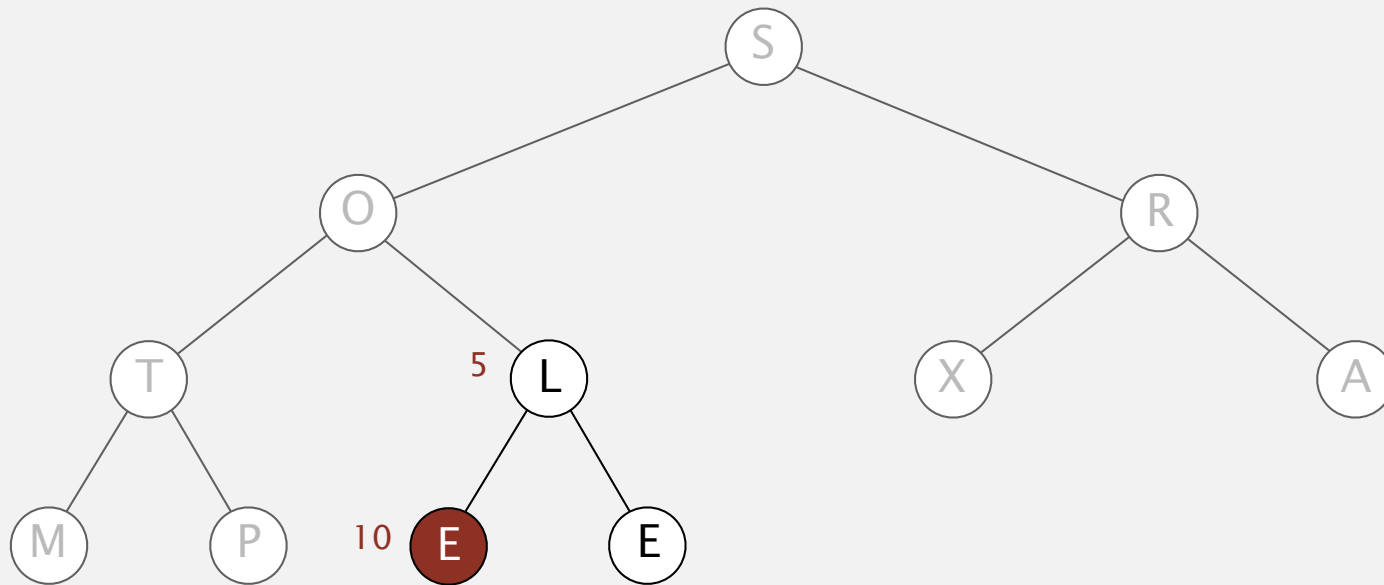


# Heapsort demo

---

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

sink 5

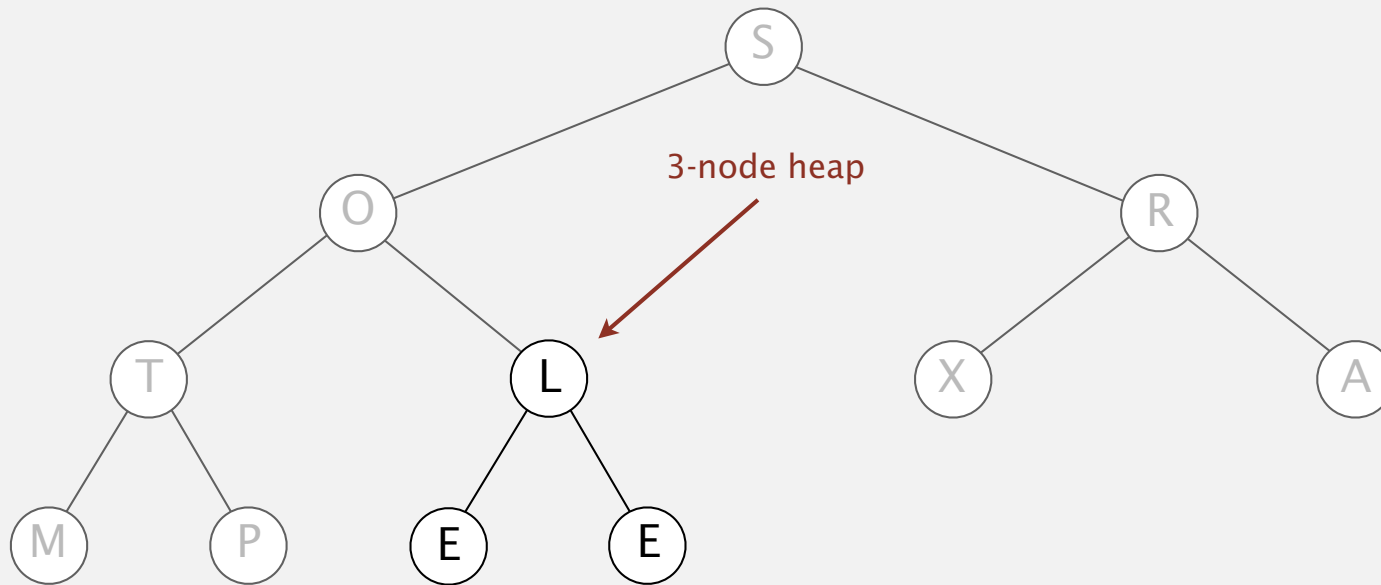


# Heapsort demo

---

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

sink 5



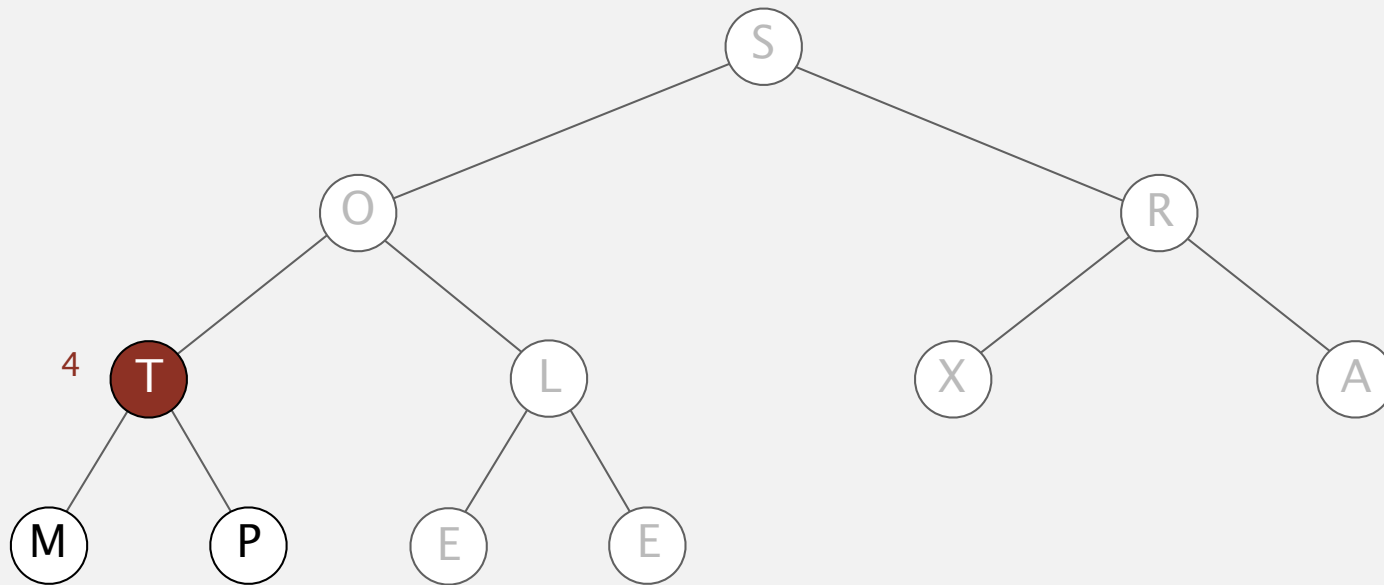
S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

# Heapsort demo

---

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

sink 4



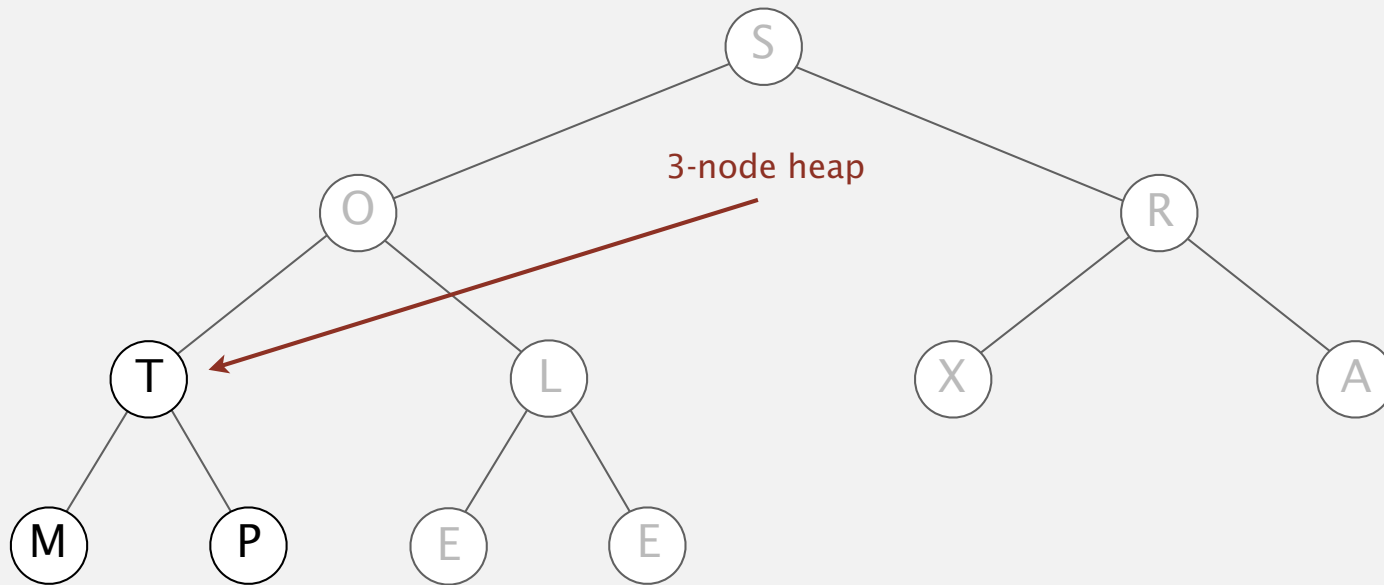
4

# Heapsort demo

---

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

sink 4



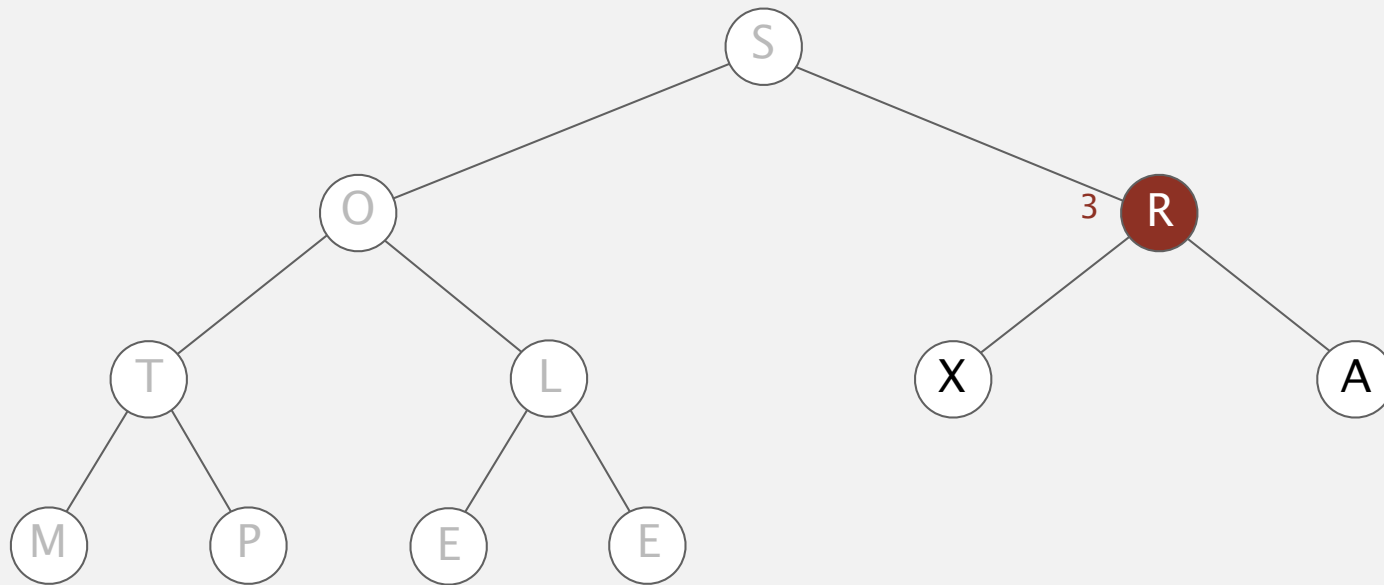
S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

# Heapsort demo

---

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

sink 3

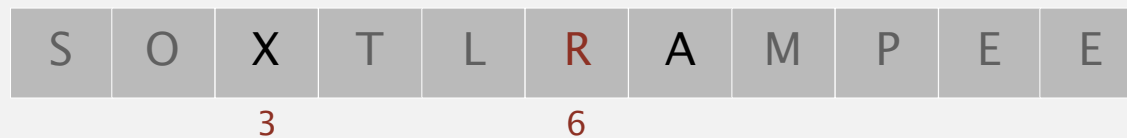
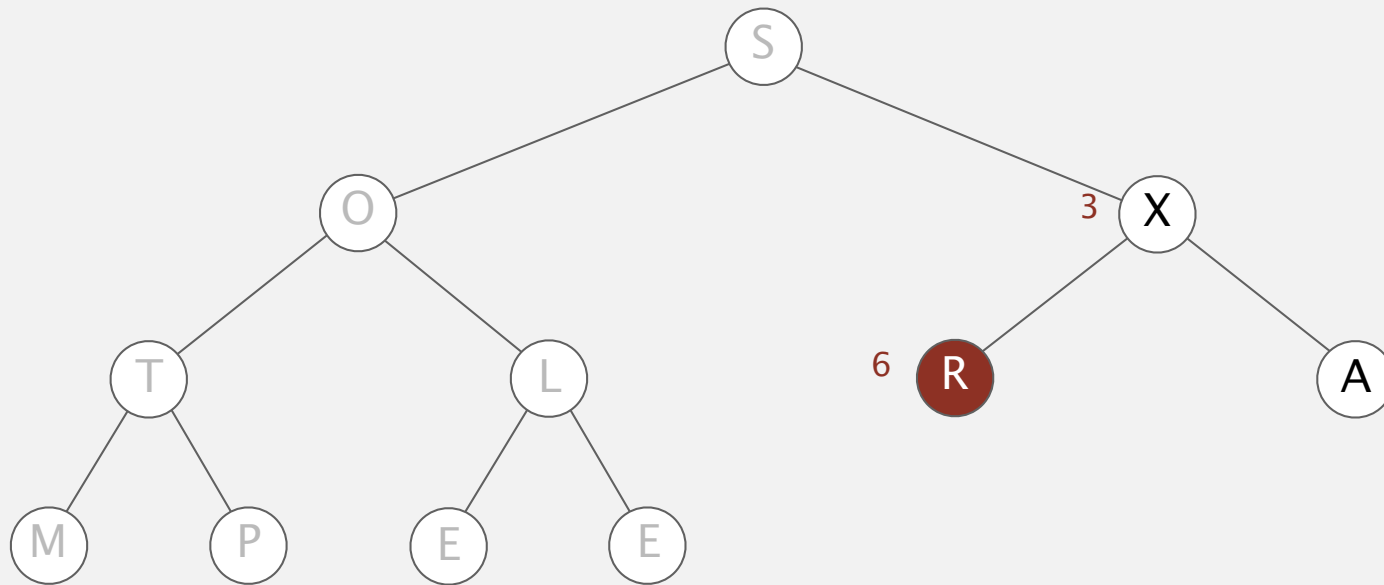


# Heapsort demo

---

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

sink 3



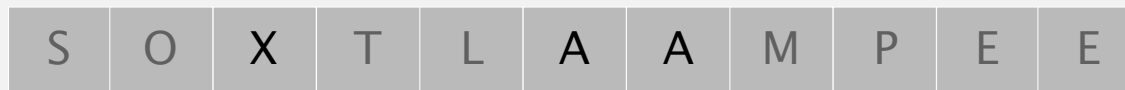
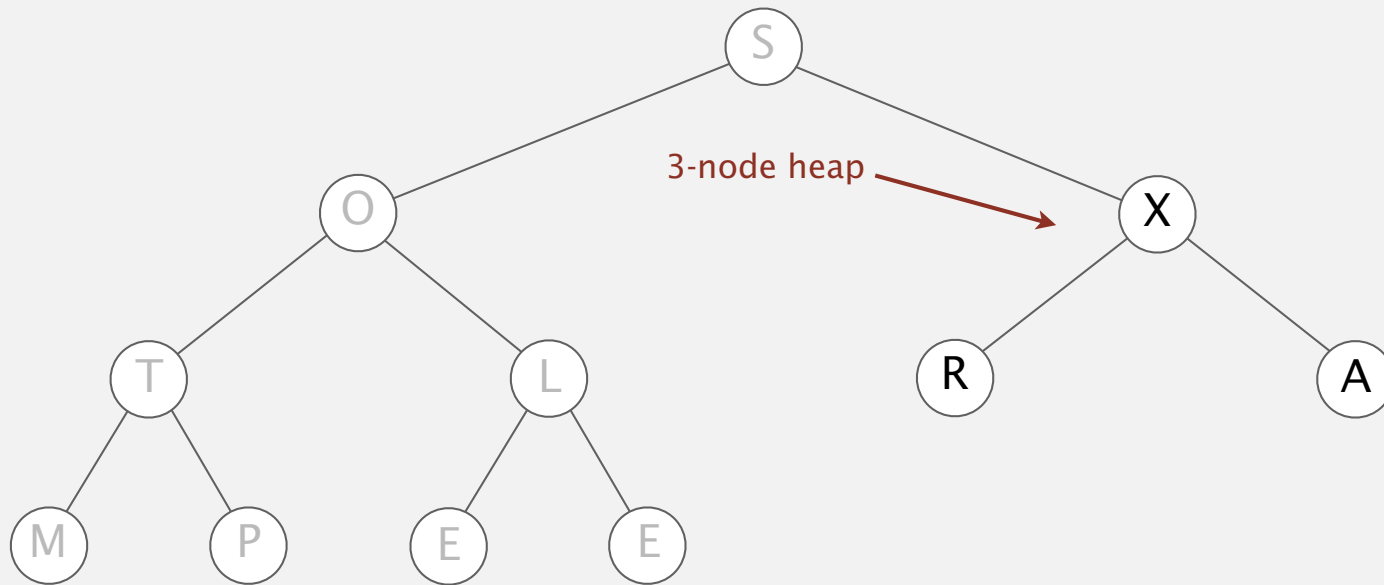


# Heapsort demo

---

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

sink 3

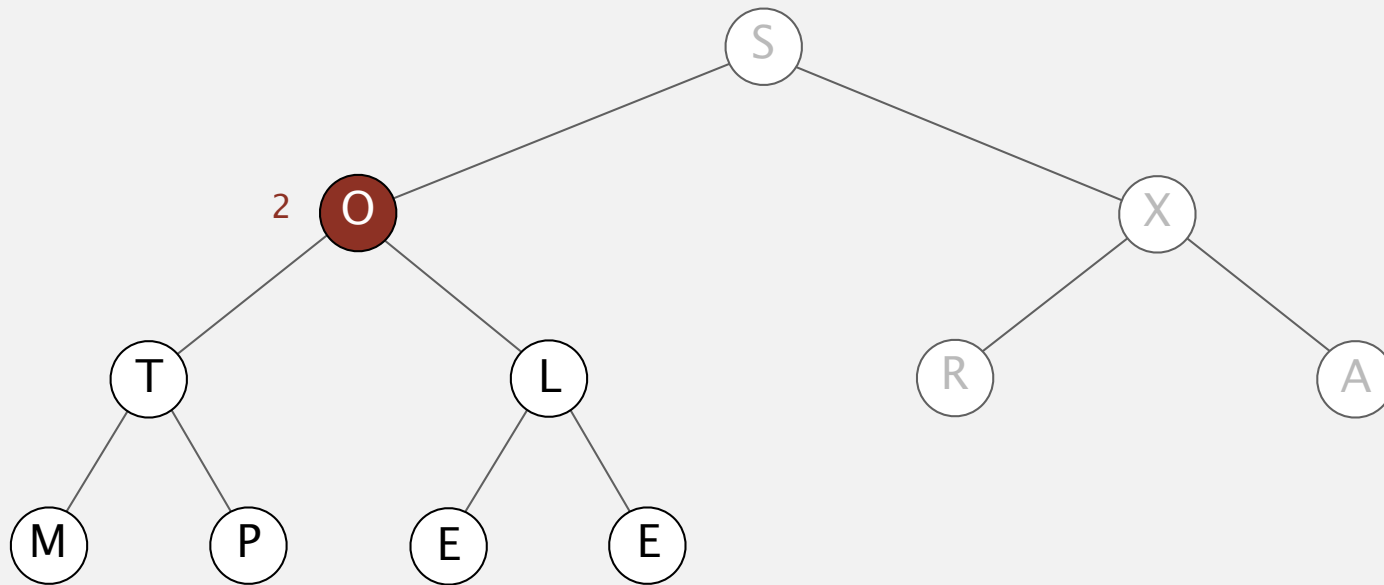


# Heapsort demo

---

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

sink 2

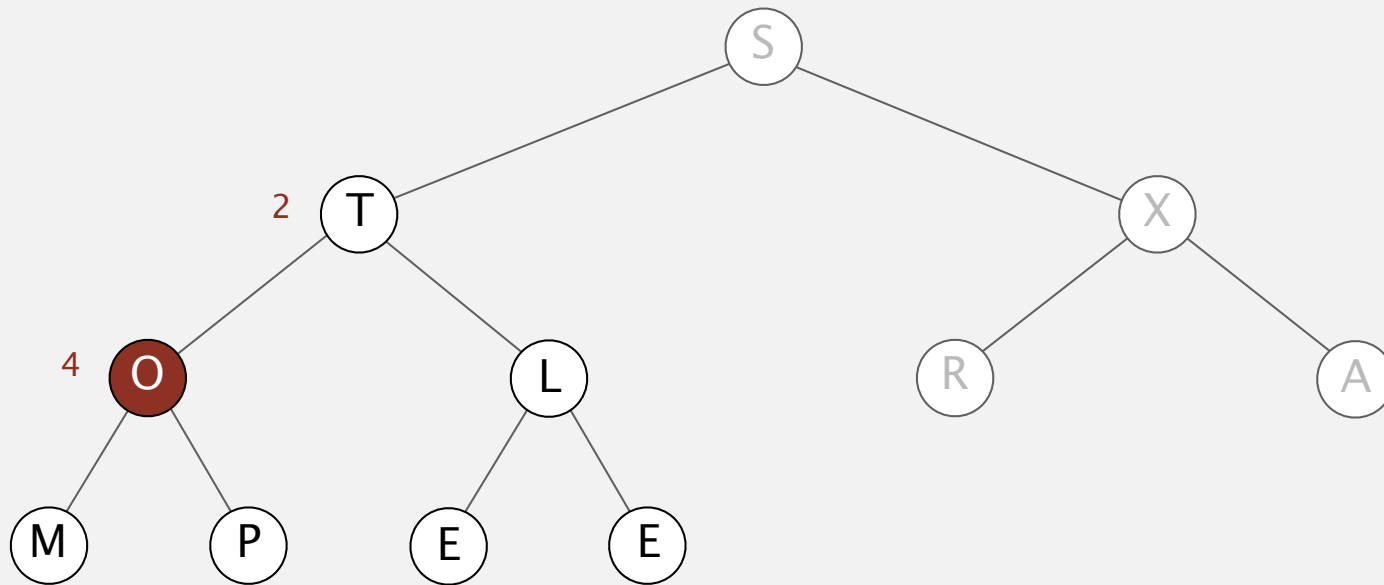


# Heapsort demo

---

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

sink 2

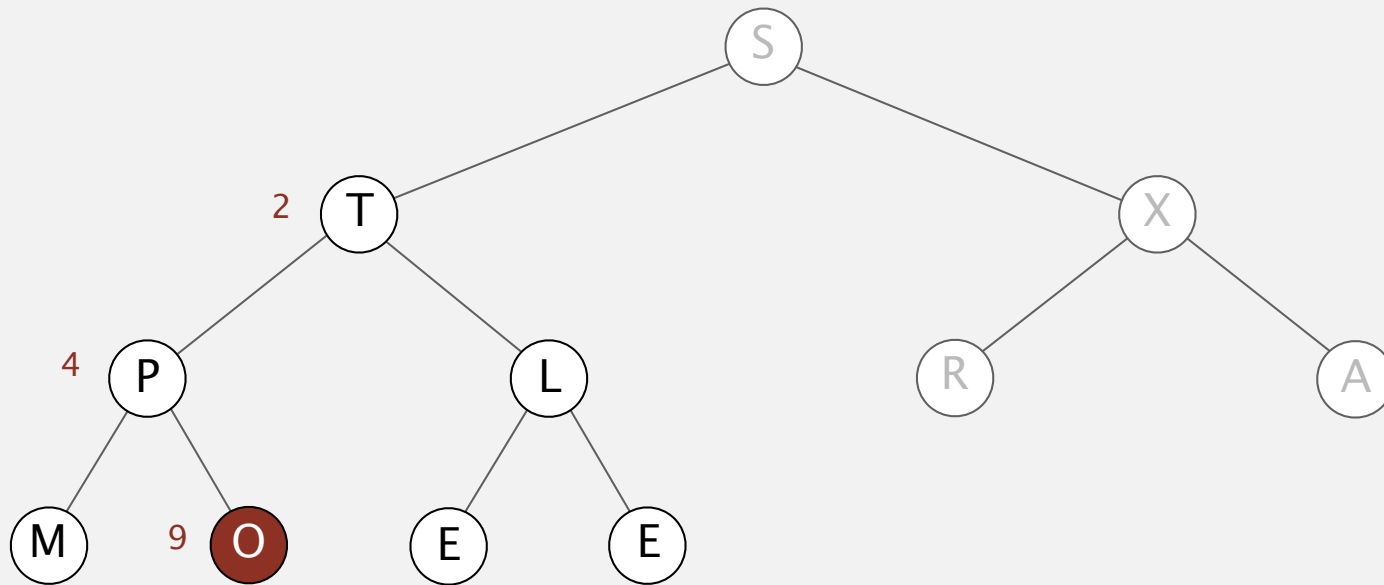


# Heapsort demo

---

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

sink 2

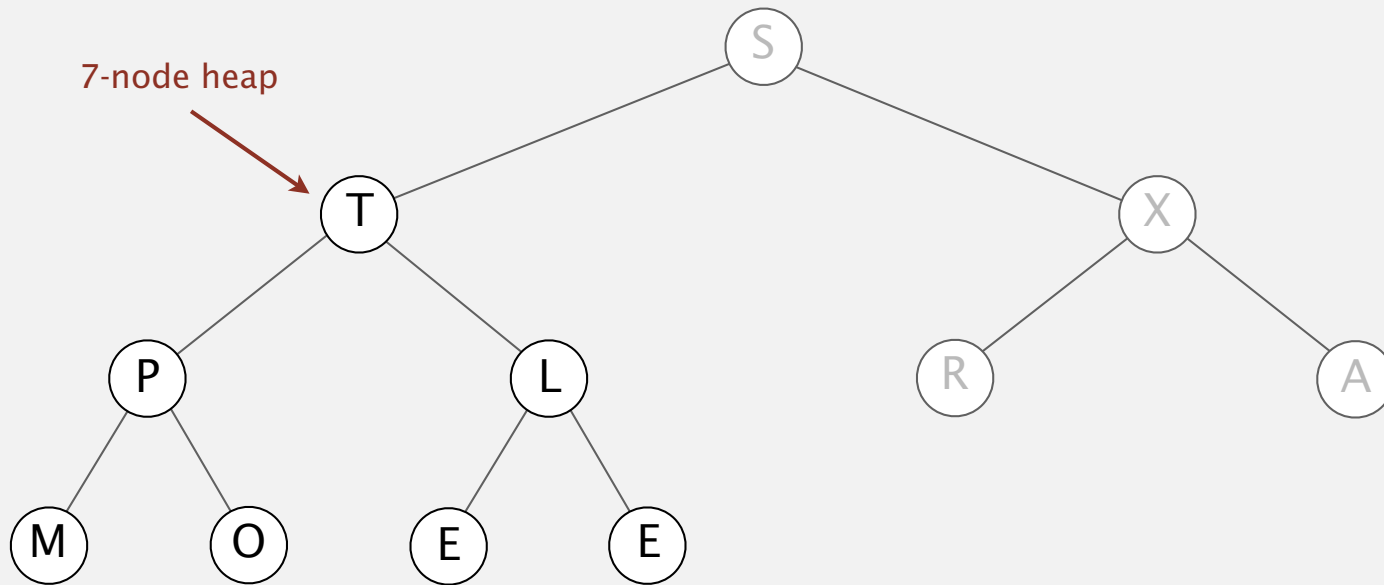


# Heapsort demo

---

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

sink 2

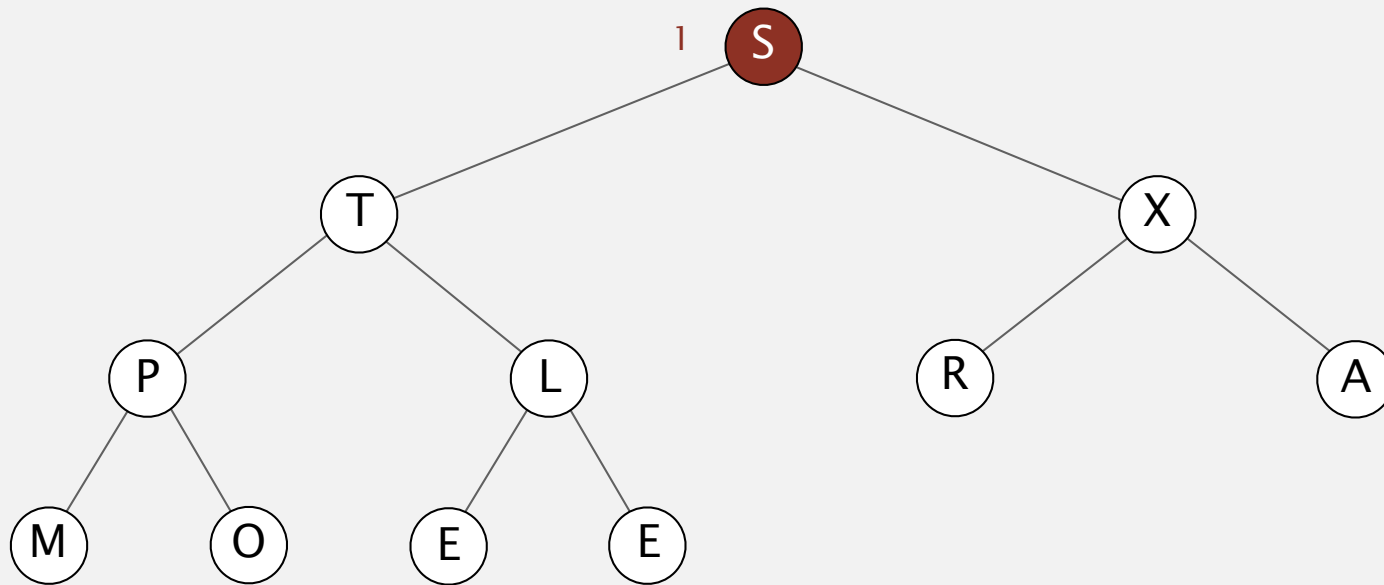


# Heapsort demo

---

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

sink 1

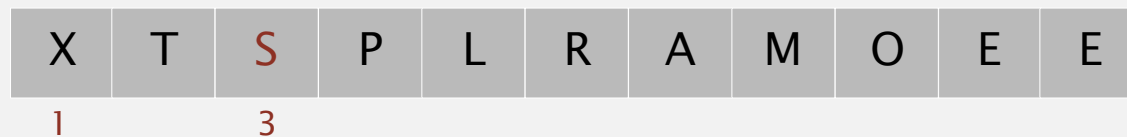
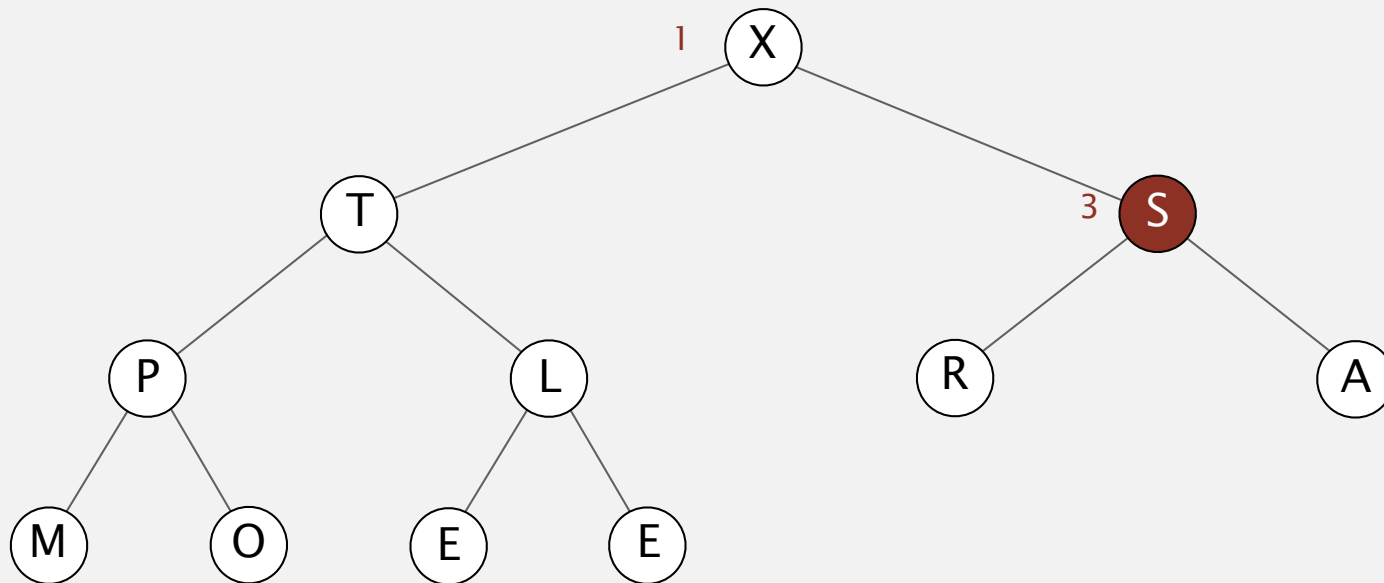


# Heapsort demo

---

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

sink 1



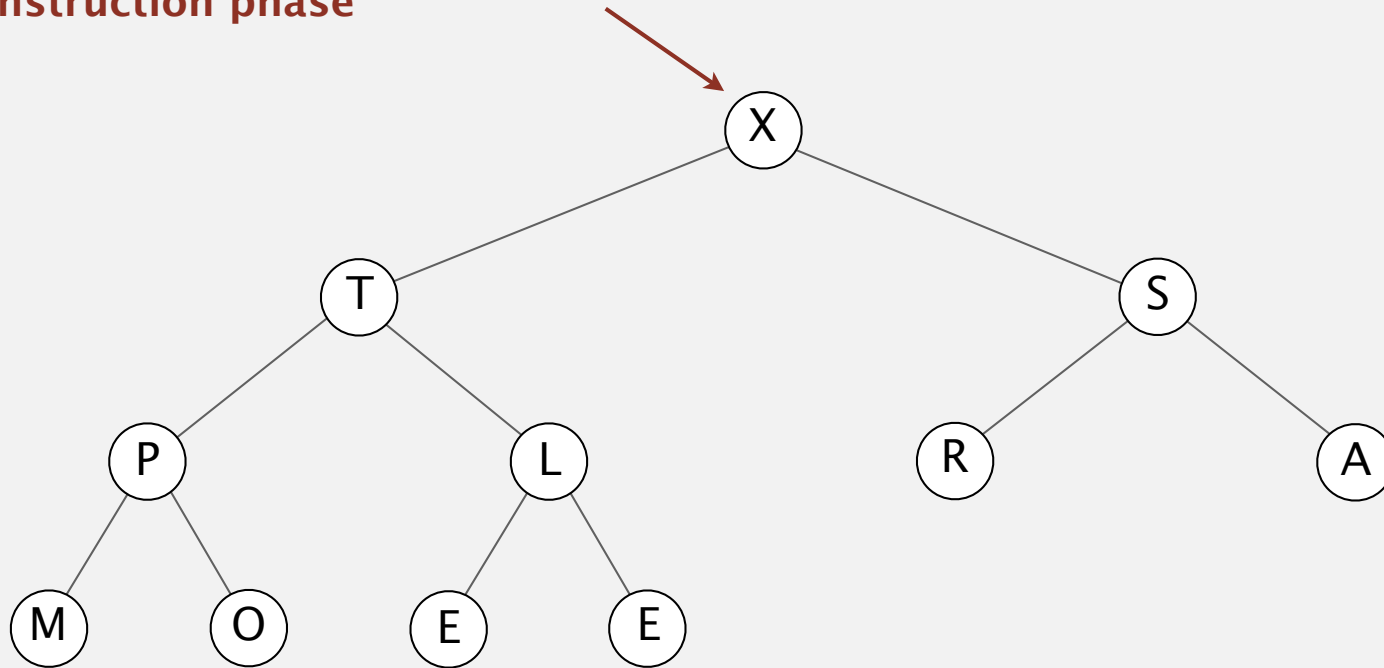
# Heapsort demo

---

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

end of construction phase

11-node heap







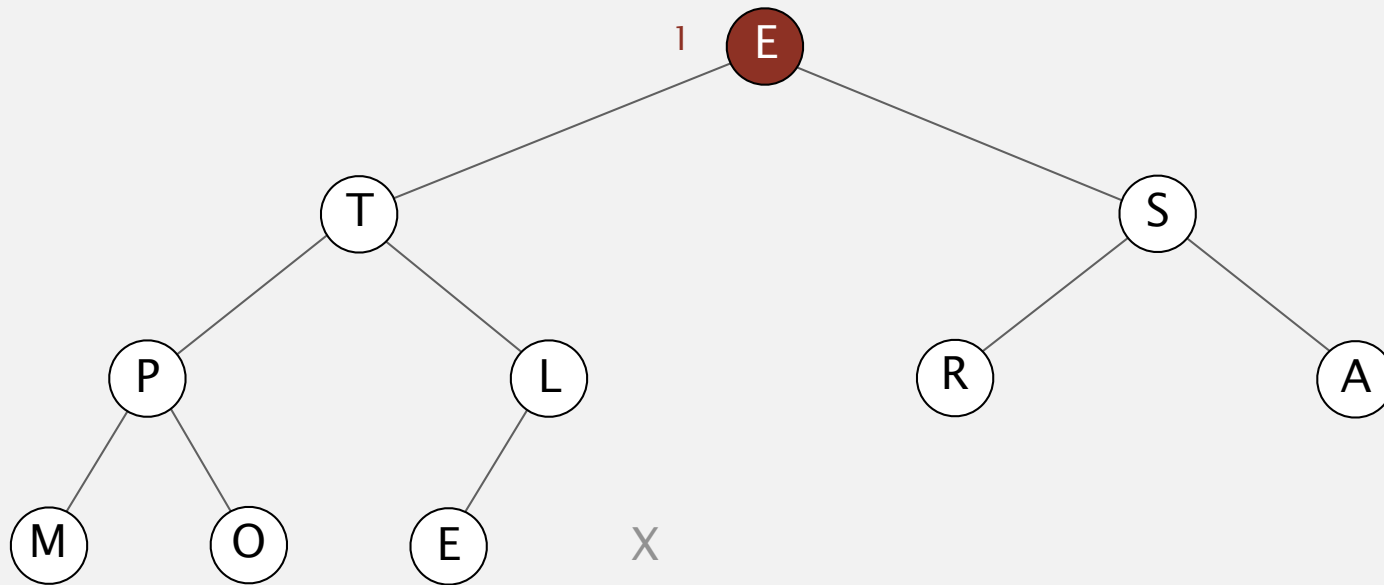


# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1



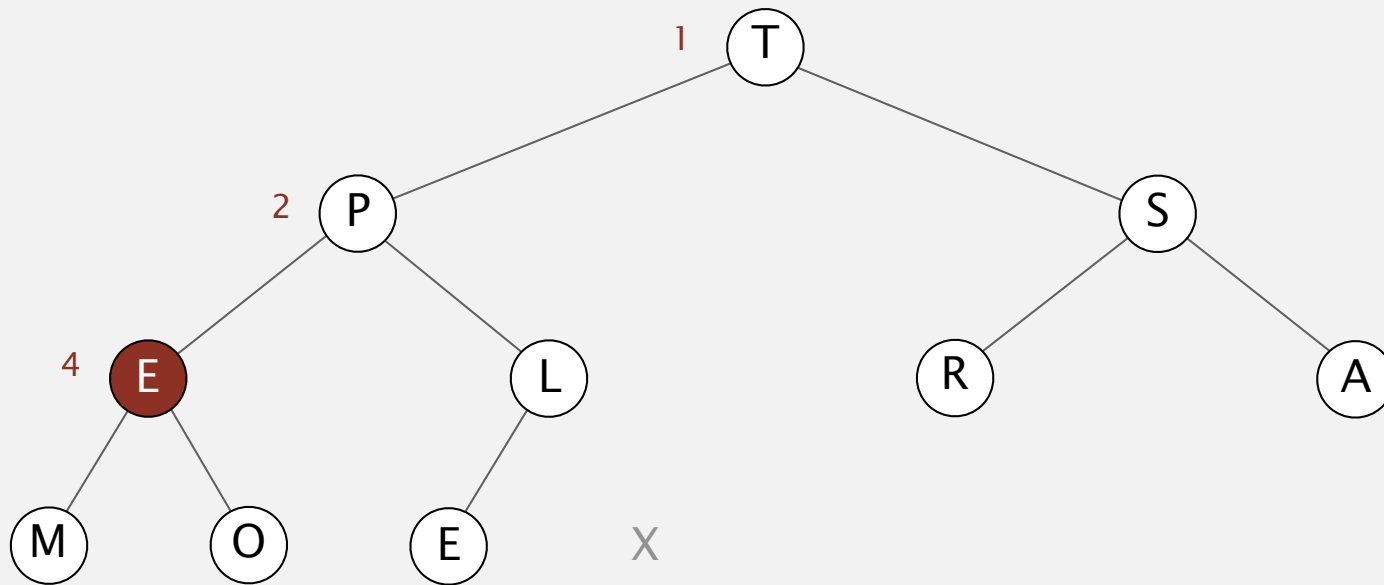


# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1

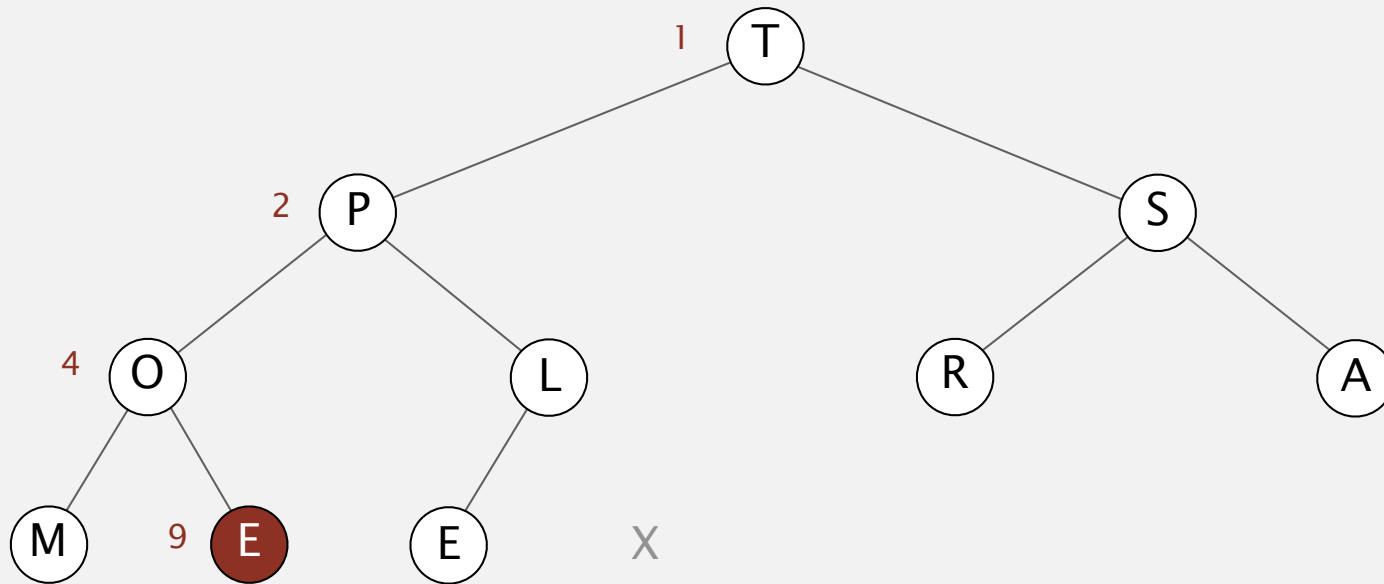


# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.

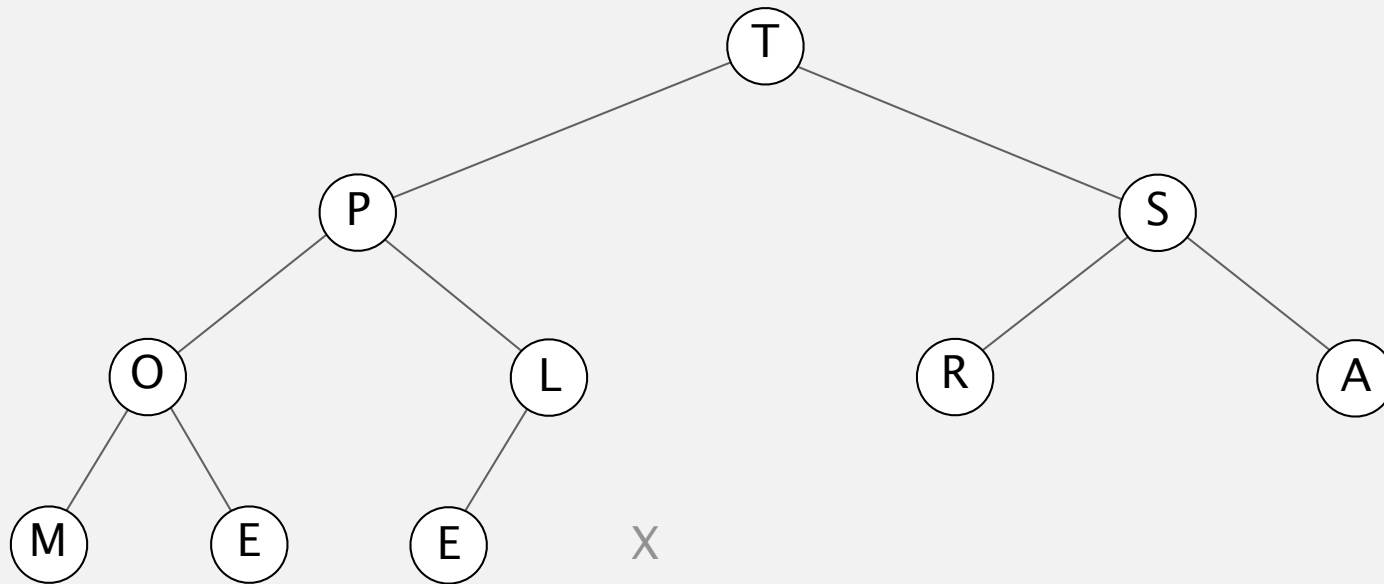
sink 1



# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.



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





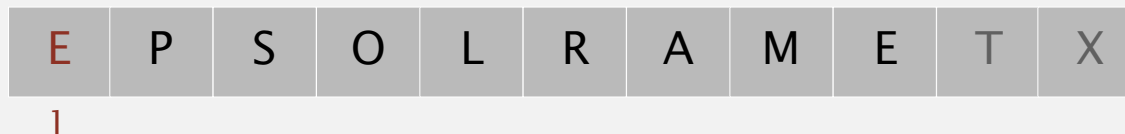
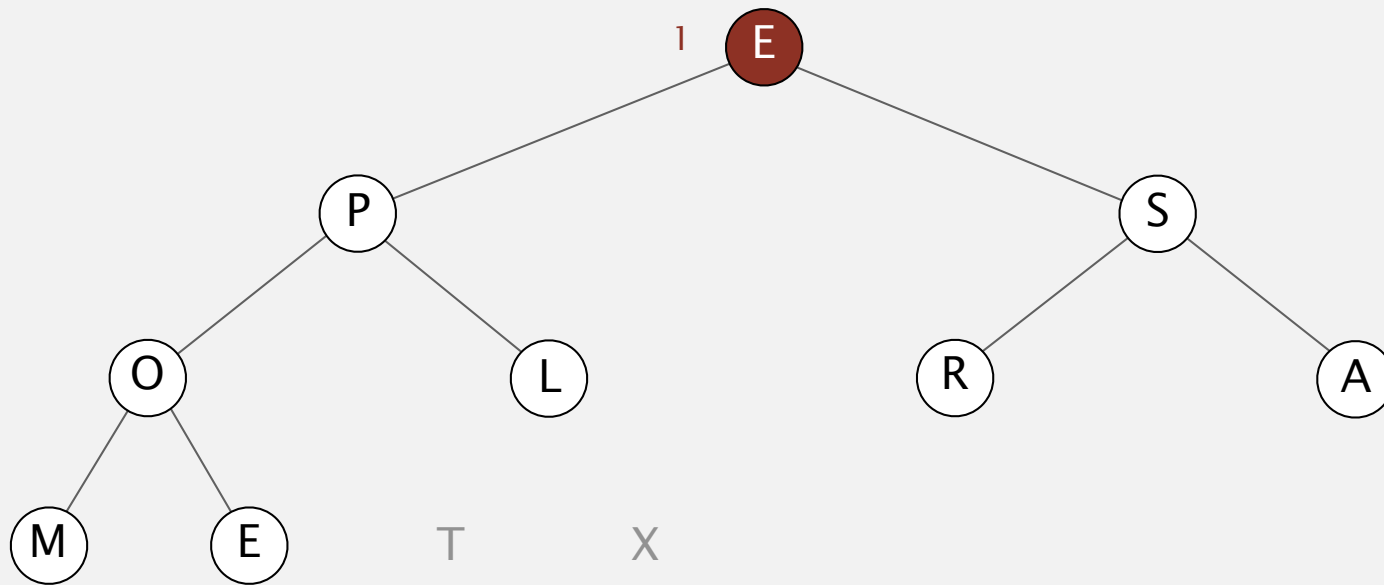


# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1

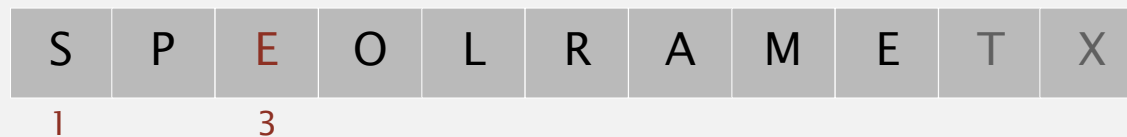
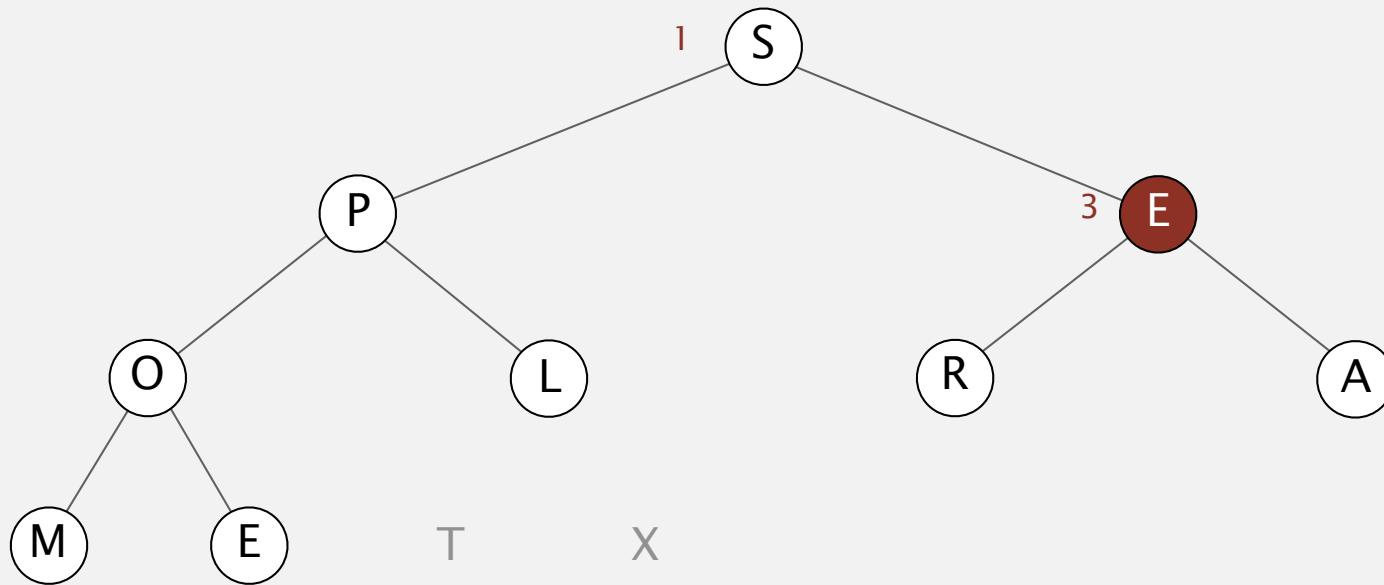


# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1

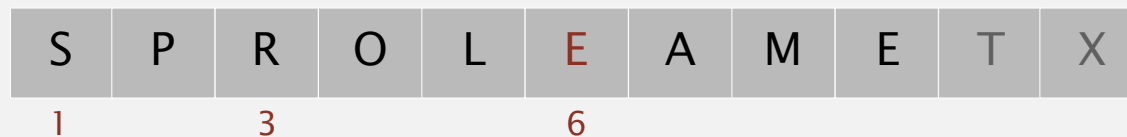
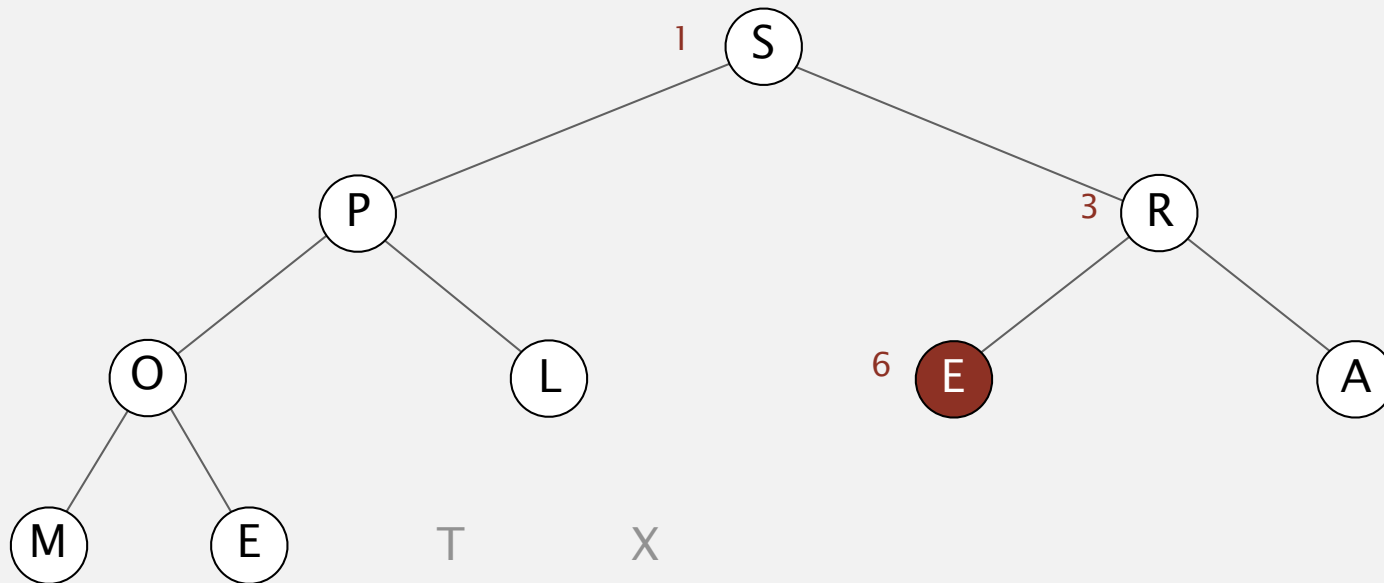


# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.

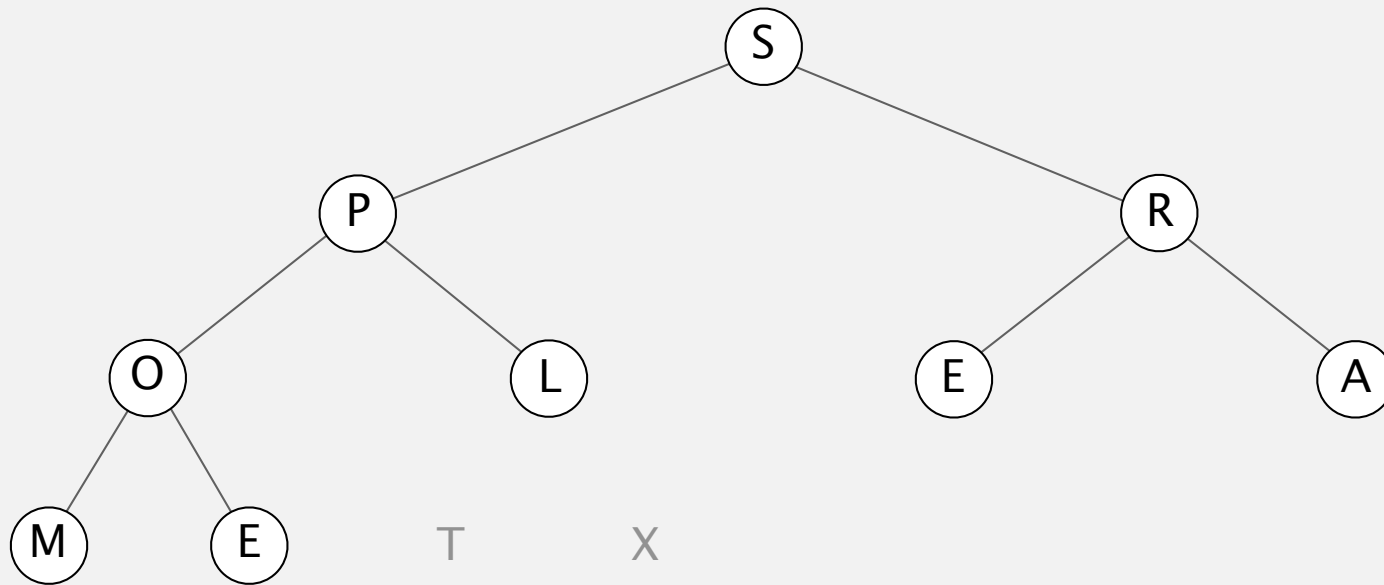
sink 1



# 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.

**array in sorted order**

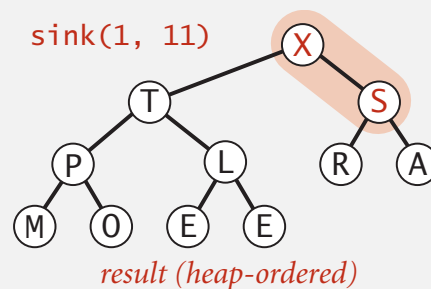
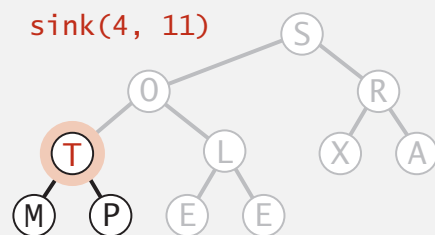
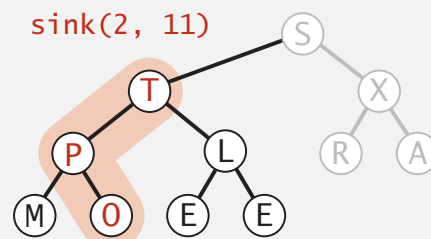
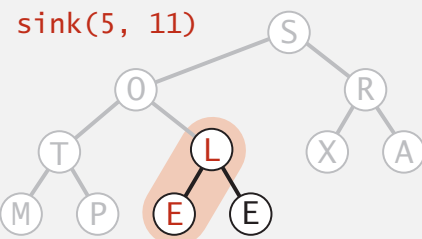
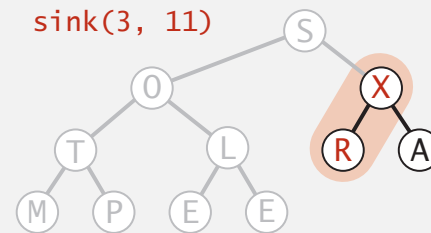
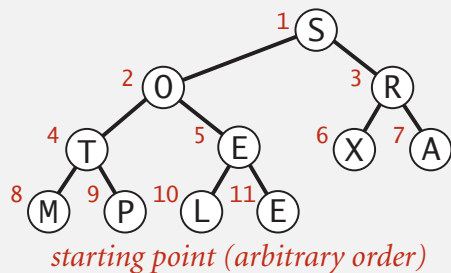


A	E	E	L	M	O	P	R	S	T	X
1	2	3	4	5	6	7	8	9	10	11

# Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

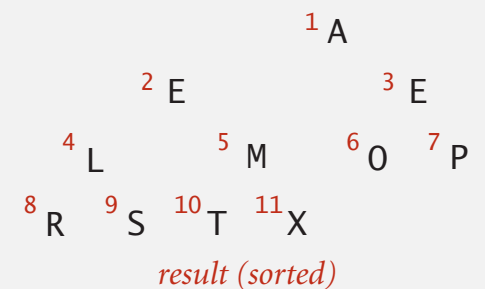
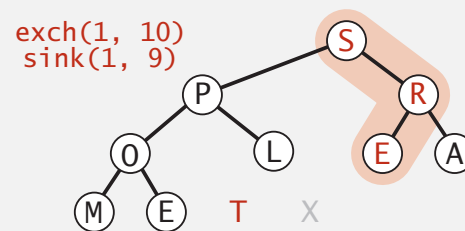
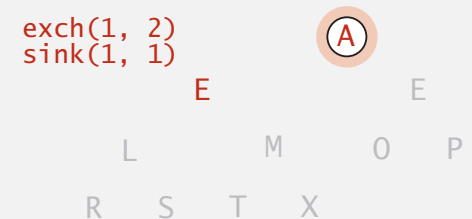
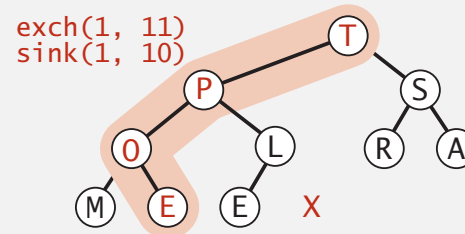
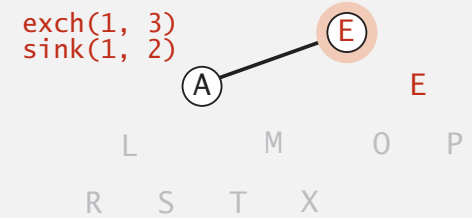
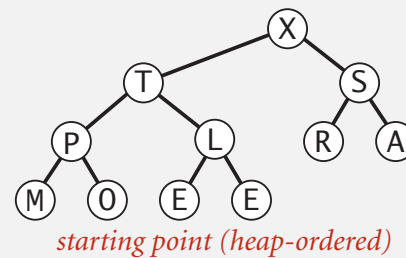


# Heapsort: sortdown

## 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[] 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 */ }
}
```

but make static (and pass arguments)

but convert from 1-based indexing to 0-base indexing

# Heapsort: trace

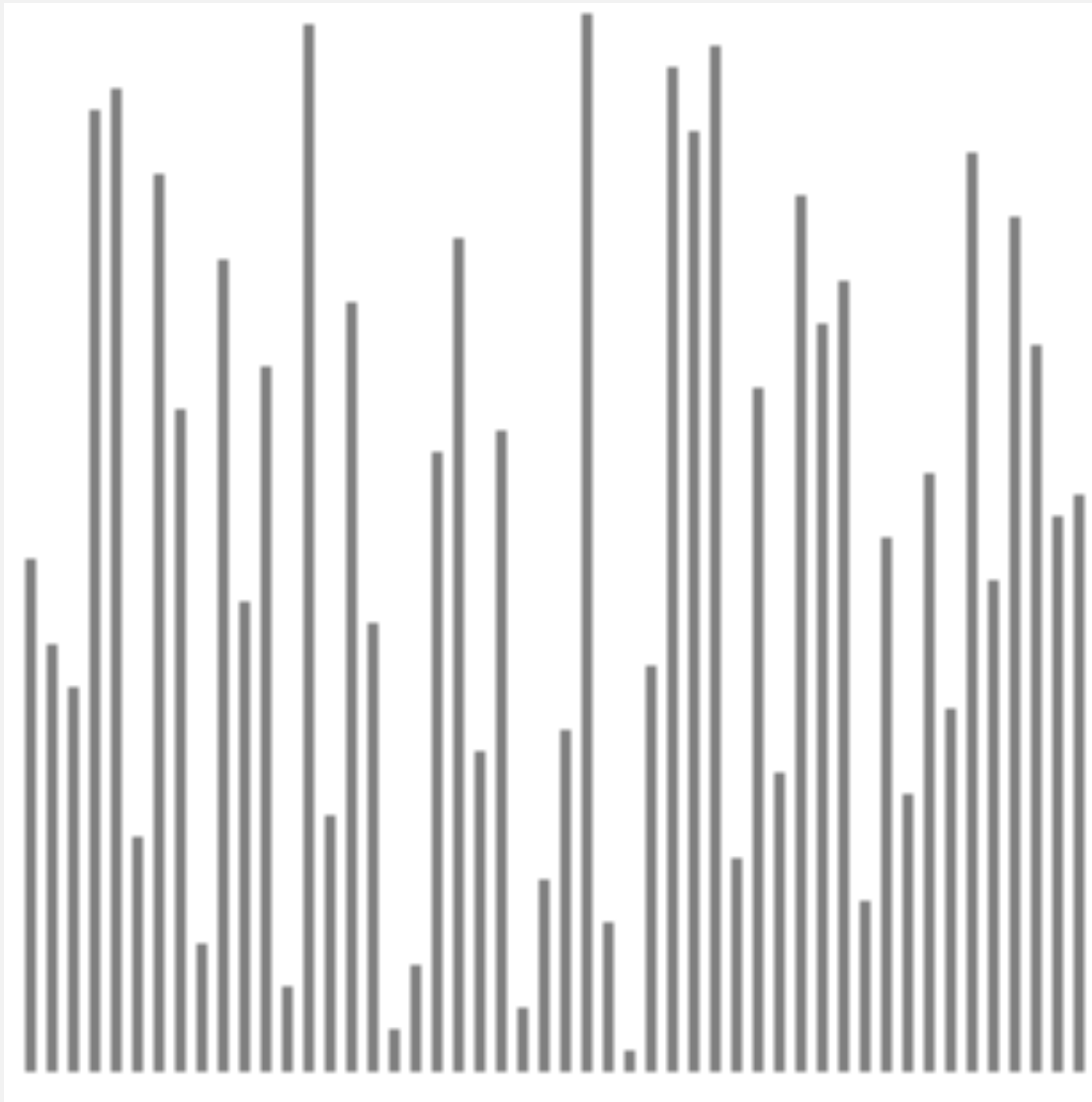
		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
<i>heap-ordered</i>			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E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



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

- ▲ algorithm position
- █ in order
- █ not in order

# 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  $\sim 1 N \lg N$   
(but no such variant is known to be practical)

**Significance.** Deterministic in-place  $N \log N$  sorting algorithm.

- Mergesort: no, linear extra space.
- Quicksort: no, randomized.
- Heapsort: yes!

← in-place merge possible, not practical

← deterministic  $N \log N$  quicksort possible, not practical

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

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



can be improved using  
advanced caching tricks

# 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
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	✓		$N \lg N$	$2 N \ln N$ (expected)	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-way quick	✓		$N$	$2 N \ln N$ (expected)	$\frac{1}{2} N^2$	improves quicksort when duplicate keys
heap	✓		$3 N$	$2 N \lg N$	$2 N \lg N$ $N$	$N \log N$ guarantee; in-place
?	✓	✓	$N$	$N \lg N$	$N \lg N$	holy sorting grail



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

## 2.4 PRIORITY QUEUES

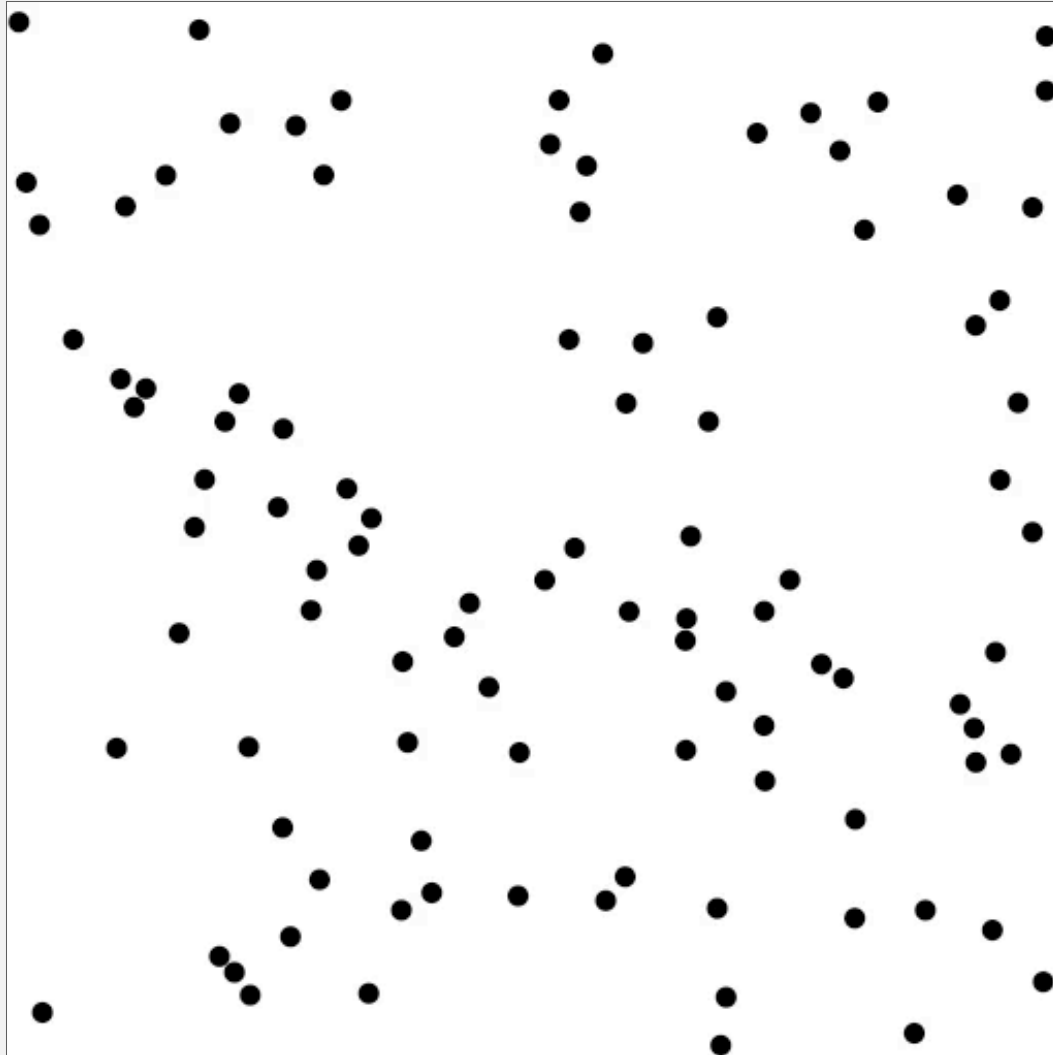
---

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

# Molecular dynamics simulation of hard discs

---

**Goal.** Simulate the motion of  $N$  moving particles that behave according to the laws of elastic collision.



# Molecular dynamics simulation of hard discs

---

**Goal.** Simulate the motion of  $N$  moving particles that behave according to the laws of elastic collision.

## Hard disc model.

- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.

temperature, pressure,  
diffusion constant

motion of individual  
atoms and molecules

**Significance.** Relates macroscopic observables to microscopic dynamics.

- Maxwell-Boltzmann: distribution of speeds as a function of temperature.
- Einstein: explain Brownian motion of pollen grains.



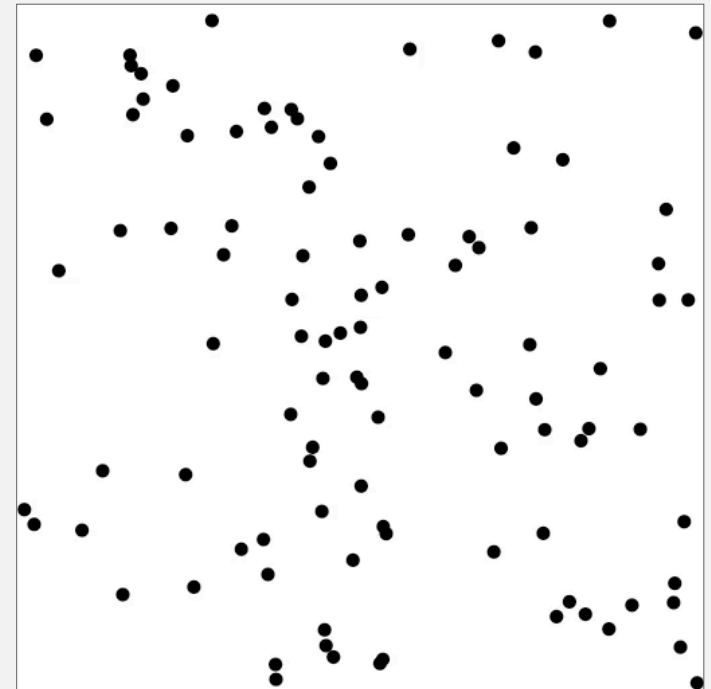
# Warmup: bouncing balls

Time-driven simulation.  $N$  bouncing balls in the unit square.

```
public class BouncingBalls
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        Ball[] balls = new Ball[N];
        for (int i = 0; i < N; i++)
            balls[i] = new Ball();
        while(true)
        {
            StdDraw.clear();
            for (int i = 0; i < N; i++)
            {
                balls[i].move(0.5);
                balls[i].draw();
            }
            StdDraw.show(50);
        }
    }
}
```

↑  
main simulation loop

```
% java BouncingBalls 100
```




# Warmup: bouncing balls

---

```
public class Ball
{
    private double rx, ry;        // position
    private double vx, vy;        // velocity
    private final double radius;  // radius
    public Ball(...)
    { /* initialize position and velocity */ }

    public void move(double dt)
    {
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }
    public void draw()
    { StdDraw.filledCircle(rx, ry, radius); }
}
```

check for collision with walls



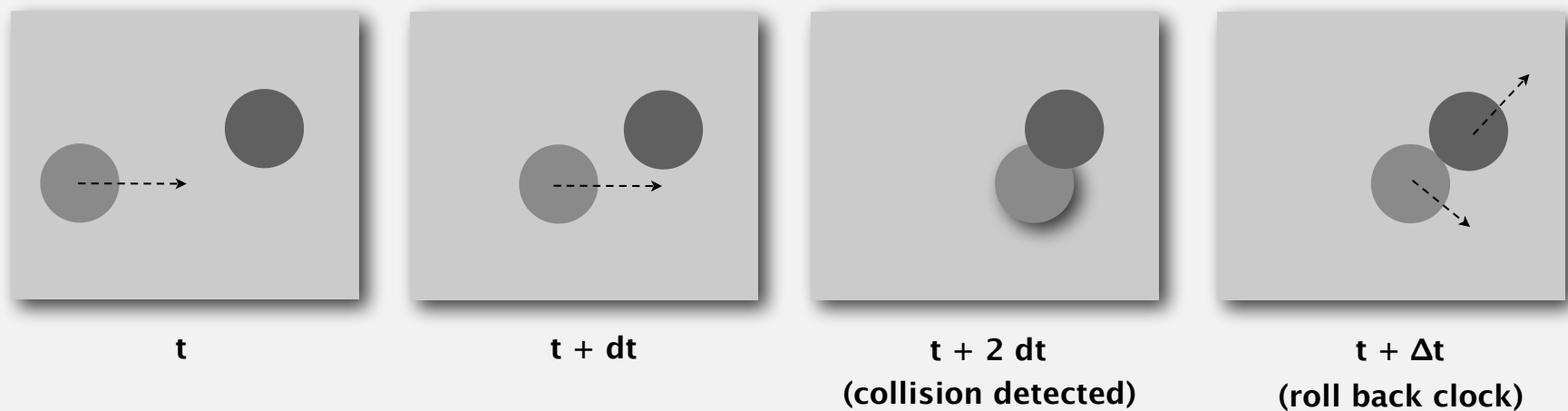
**Missing.** Check for balls colliding with **each other**.

- Physics problems: when? what effect?
- CS problems: which object does the check? too many checks?

# Time-driven simulation

---

- Discretize time in quanta of size  $dt$ .
- Update the position of each particle after every  $dt$  units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

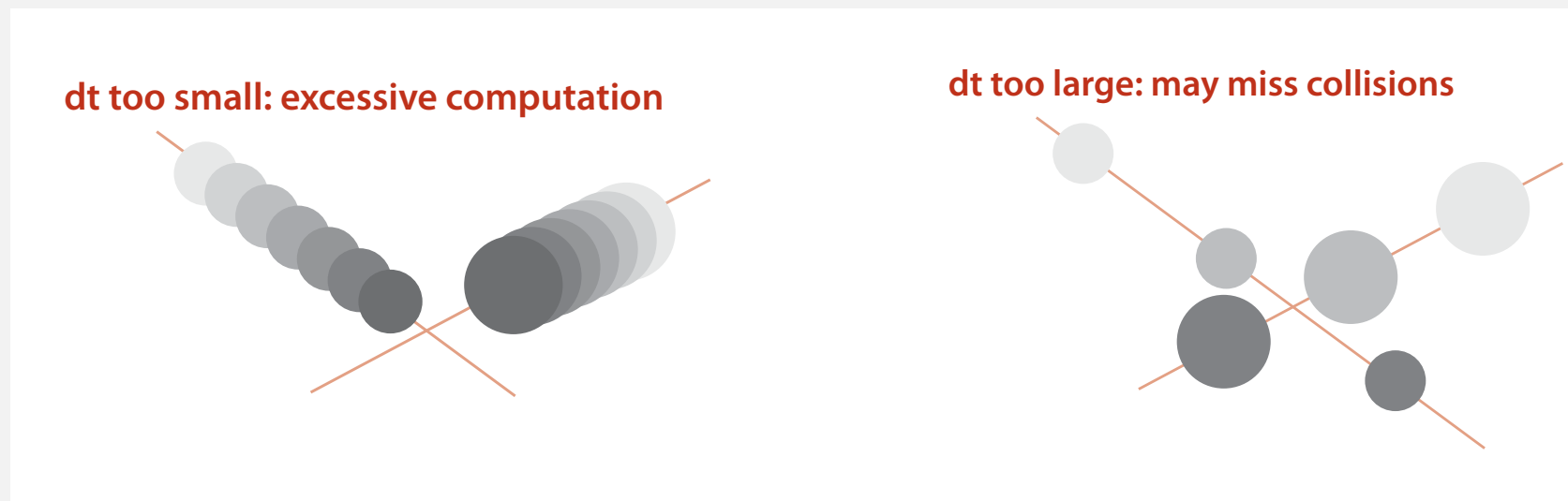


# Time-driven simulation

---

## Main drawbacks.

- $\sim N^2 / 2$  overlap checks per time quantum.
- Simulation is too slow if  $dt$  is very small.
- May miss collisions if  $dt$  is too large.  
(if colliding particles fail to overlap when we are looking)



# Event-driven simulation

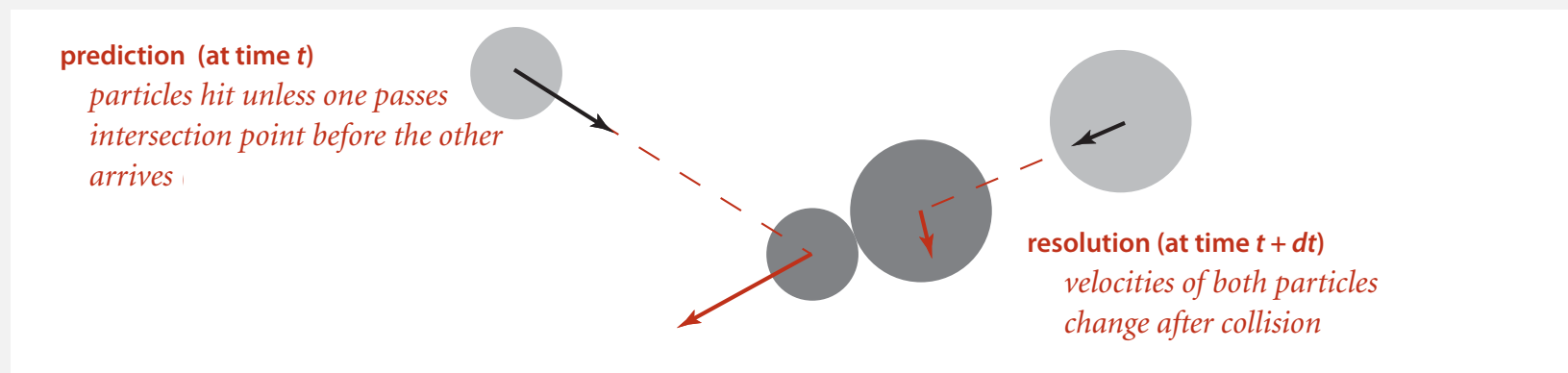
---

Change state only when something interesting happens.

- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain **PQ** of collision events, prioritized by time.
- Delete min = get next collision.

**Collision prediction.** Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

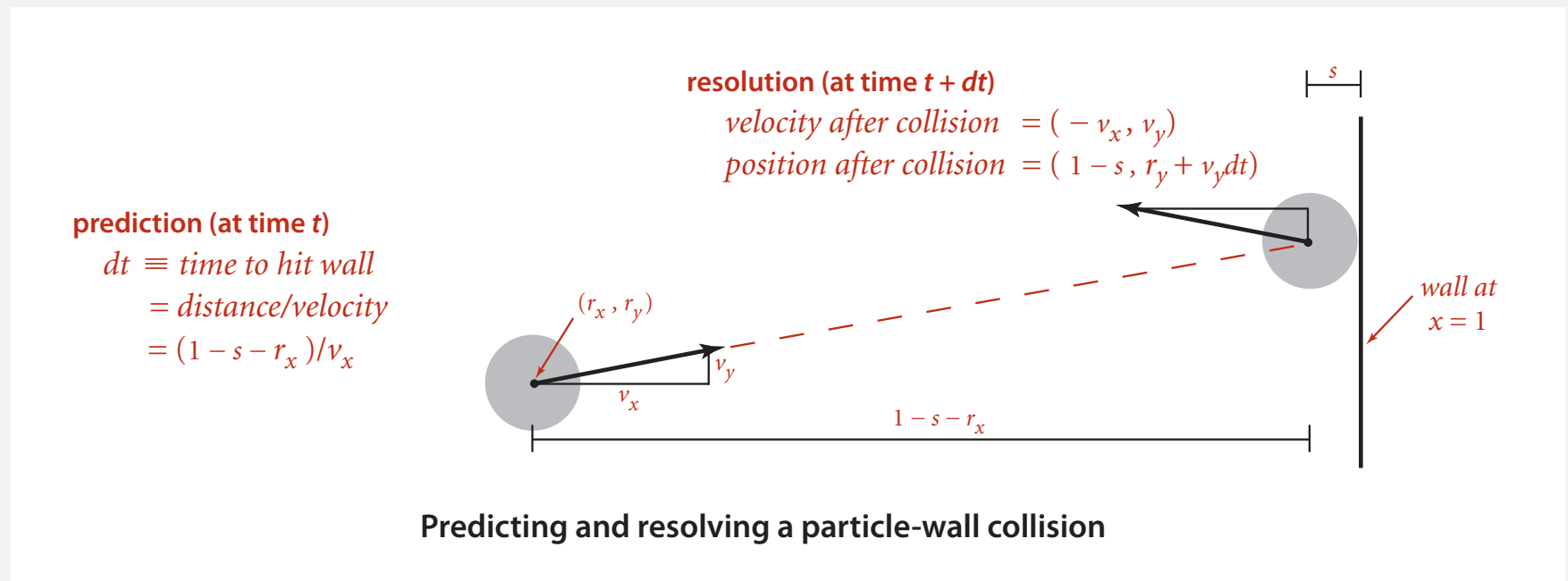
**Collision resolution.** If collision occurs, update colliding particle(s) according to laws of elastic collisions.



# Particle-wall collision

## Collision prediction and resolution.

- Particle of radius  $s$  at position  $(r_x, r_y)$ .
- Particle moving in unit box with velocity  $(v_x, v_y)$ .
- Will it collide with a vertical wall? If so, when?

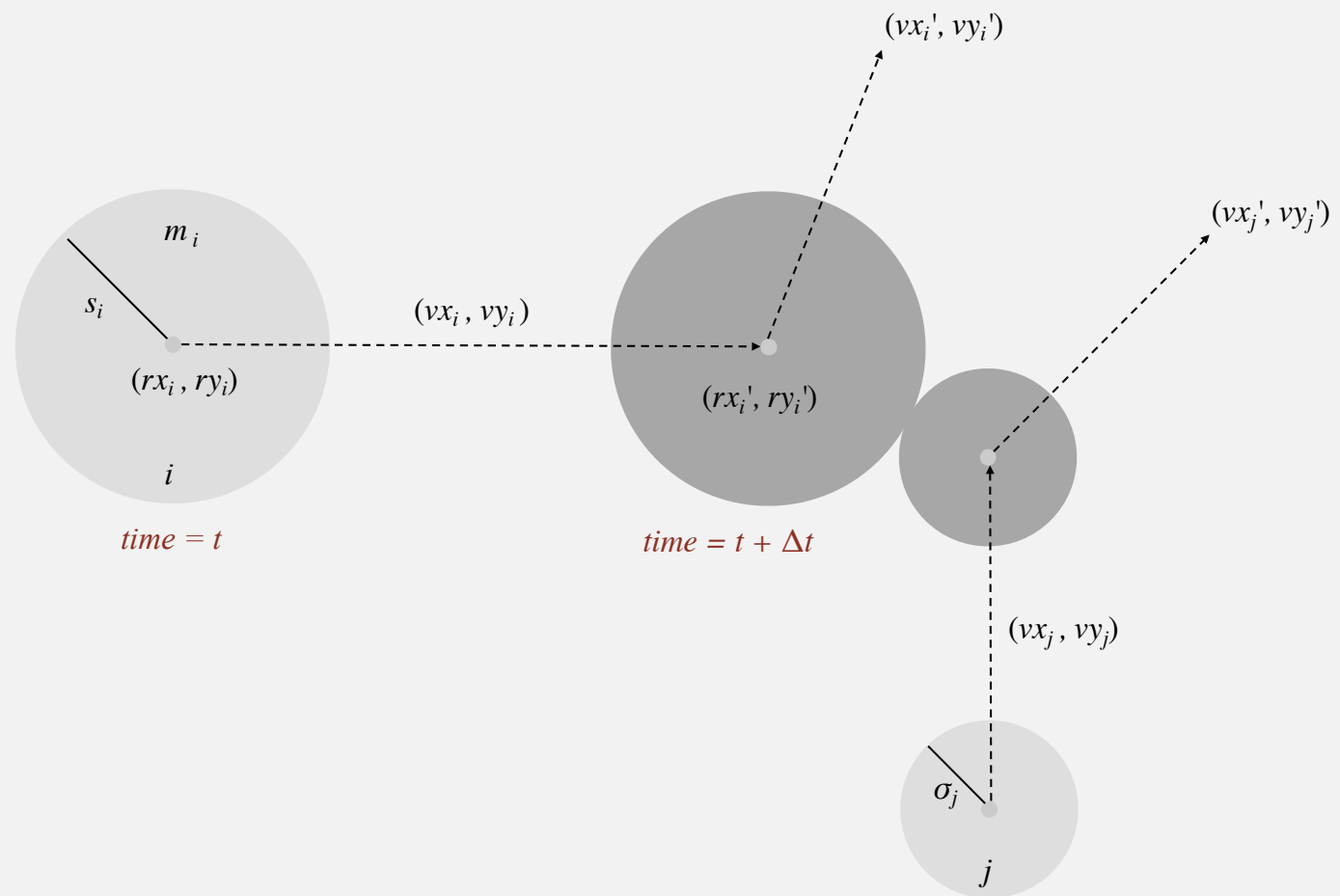


# Particle-particle collision prediction

---

## Collision prediction.

- Particle  $i$ : radius  $s_i$ , position  $(rx_i, ry_i)$ , velocity  $(vx_i, vy_i)$ .
- Particle  $j$ : radius  $s_j$ , position  $(rx_j, ry_j)$ , velocity  $(vx_j, vy_j)$ .
- Will particles  $i$  and  $j$  collide? If so, when?



# Particle-particle collision prediction

---

## Collision prediction.

- Particle  $i$ : radius  $s_i$ , position  $(rx_i, ry_i)$ , velocity  $(vx_i, vy_i)$ .
- Particle  $j$ : radius  $s_j$ , position  $(rx_j, ry_j)$ , velocity  $(vx_j, vy_j)$ .
- Will particles  $i$  and  $j$  collide? If so, when?

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0, \\ \infty & \text{if } d < 0, \\ -\frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise} \end{cases}$$

$$d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v) (\Delta r \cdot \Delta r - s^2), \quad s = s_i + s_j$$

$$\begin{aligned} \Delta v &= (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j) & \Delta v \cdot \Delta v &= (\Delta vx)^2 + (\Delta vy)^2 \\ \Delta r &= (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j) & \Delta r \cdot \Delta r &= (\Delta rx)^2 + (\Delta ry)^2 \\ & & \Delta v \cdot \Delta r &= (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry) \end{aligned}$$

**Important note: This is physics, so we won't be testing you on it!**




# Particle-particle collision resolution

---

**Collision resolution.** When two particles collide, how does velocity change?

$$\begin{aligned}vx_i' &= vx_i + Jx / m_i \\vy_i' &= vy_i + Jy / m_i \\vx_j' &= vx_j - Jx / m_j \\vy_j' &= vy_j - Jy / m_j\end{aligned}$$

Newton's second law  
(momentum form)



$$Jx = \frac{J \Delta r_x}{s}, \quad Jy = \frac{J \Delta r_y}{s}, \quad J = \frac{2 m_i m_j (\Delta v \cdot \Delta r)}{s (m_i + m_j)}$$

impulse due to normal force  
(conservation of energy, conservation of momentum)

**Important note: This is physics, so we won't be testing you on it!**

# Particle data type skeleton

---

```
public class Particle
{
    private double rx, ry;          // position
    private double vx, vy;          // velocity
    private final double radius;    // radius
    private final double mass;      // mass
    private int count;              // number of collisions

    public Particle( ... )          { ... }

    public void move(double dt) { ... }
    public void draw()           { ... }

    public double timeToHit(Particle that) { }
    public double timeToHitVerticalWall() { }
    public double timeToHitHorizontalWall() { }

    public void bounceOff(Particle that) { }
    public void bounceOffVerticalWall() { }
    public void bounceOffHorizontalWall() { }

}
```

predict collision  
with particle or wall

resolve collision  
with particle or wall

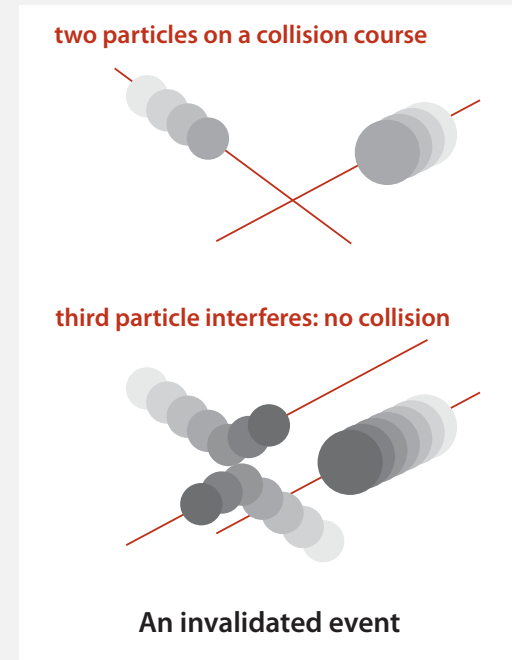
# Collision system: event-driven simulation main loop

---

## Initialization.

- Fill PQ with all potential particle-wall collisions.
- Fill PQ with all potential particle-particle collisions.

“potential” since collision is invalidated  
if some other collision intervenes



## Main loop.

- Delete the impending event from PQ (min priority =  $t$ ).
- If the event has been invalidated, ignore it.
- Advance all particles to time  $t$ , on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

# Event data type

---

## Conventions.

- Neither particle null  $\Rightarrow$  particle-particle collision.
- One particle null  $\Rightarrow$  particle-wall collision.
- Both particles null  $\Rightarrow$  redraw event.

```
private static class Event implements Comparable<Event>
{
    private final double time;           // time of event
    private final Particle a, b;        // particles involved in event
    private final int countA, countB;   // collision counts of a and b

    public Event(double t, Particle a, Particle b)           create event
    { ... }

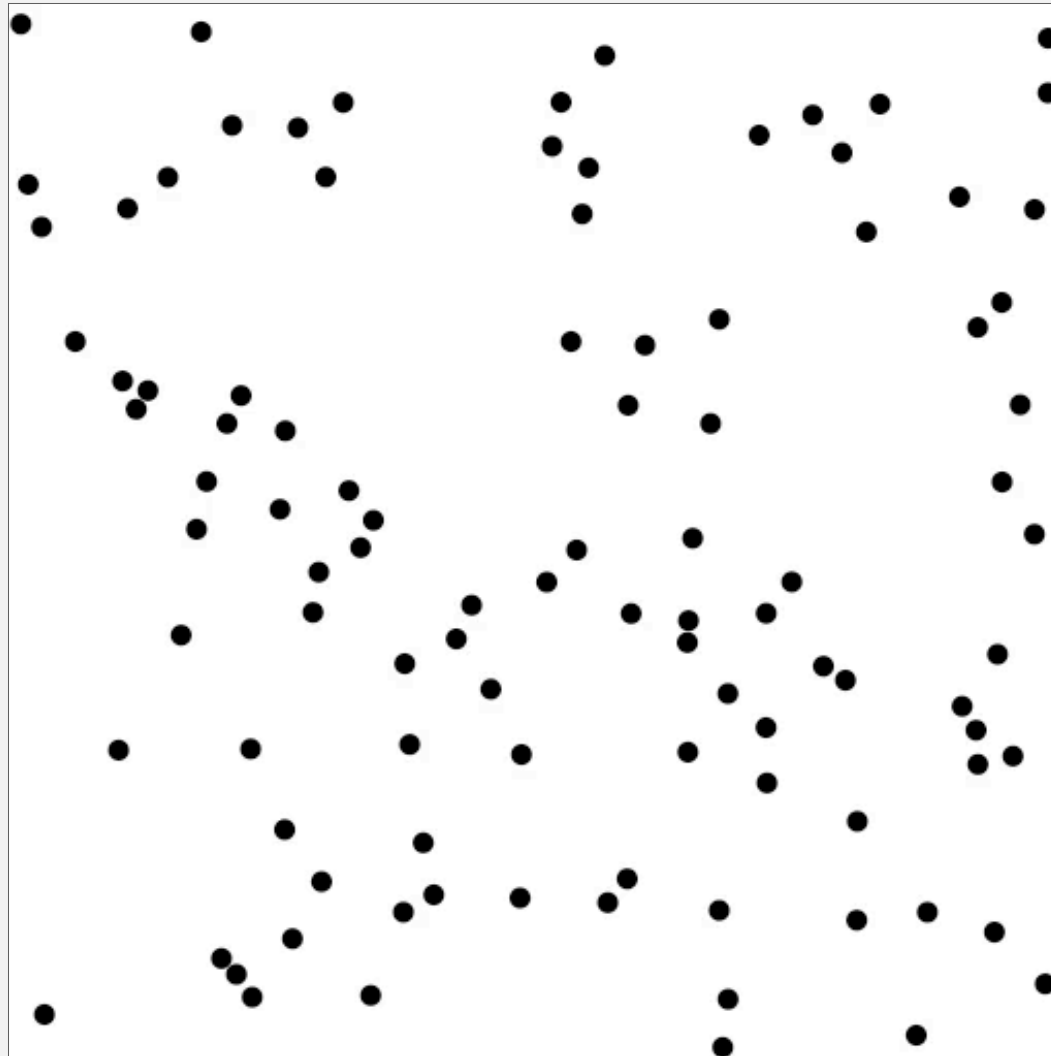
    public int compareTo(Event that)                         ordered by time
    { return this.time - that.time; }

    public boolean isValid()                                  valid if no intervening collisions
    { ... }                                                    (compare collision counts)
}
```

# Particle collision simulation: example 1

---

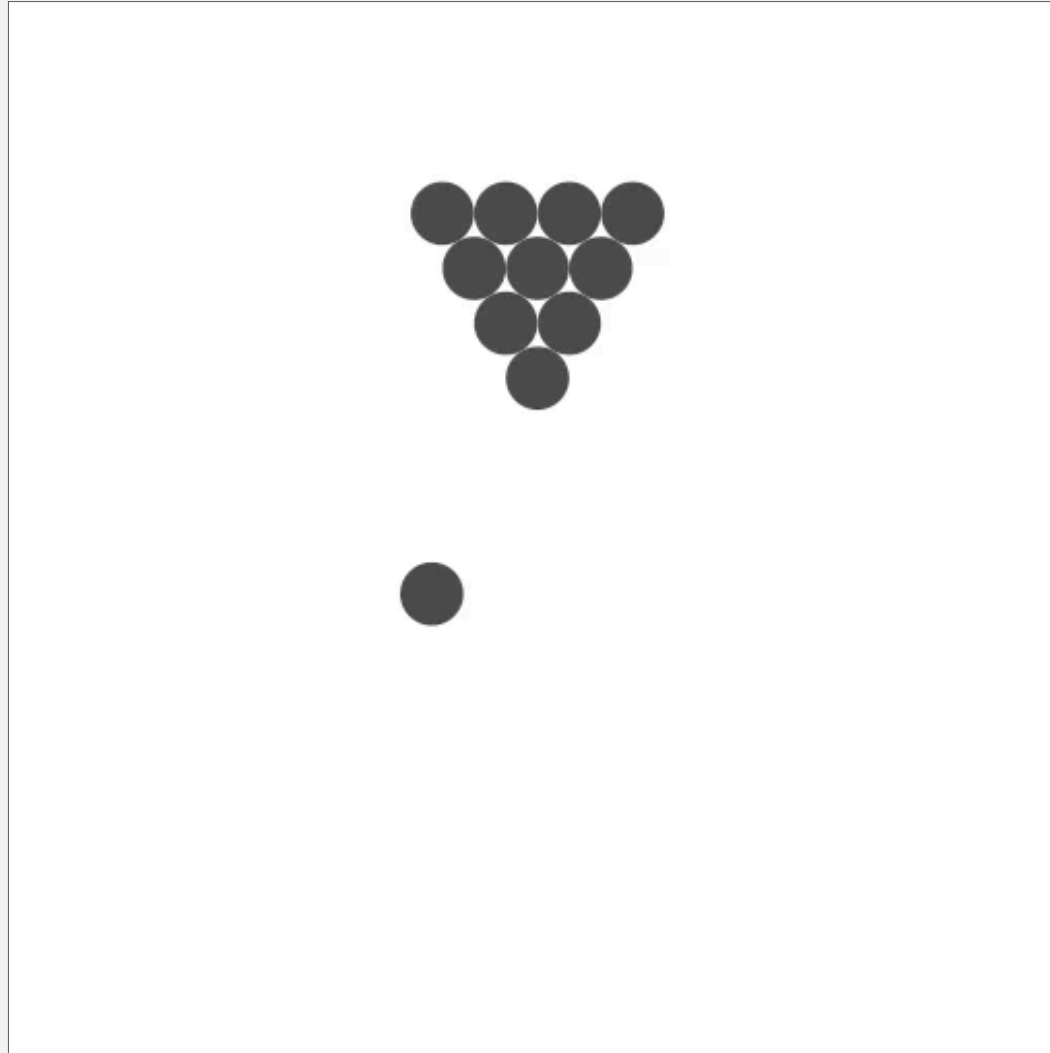
```
% java CollisionSystem 100
```



## Particle collision simulation: example 2

---

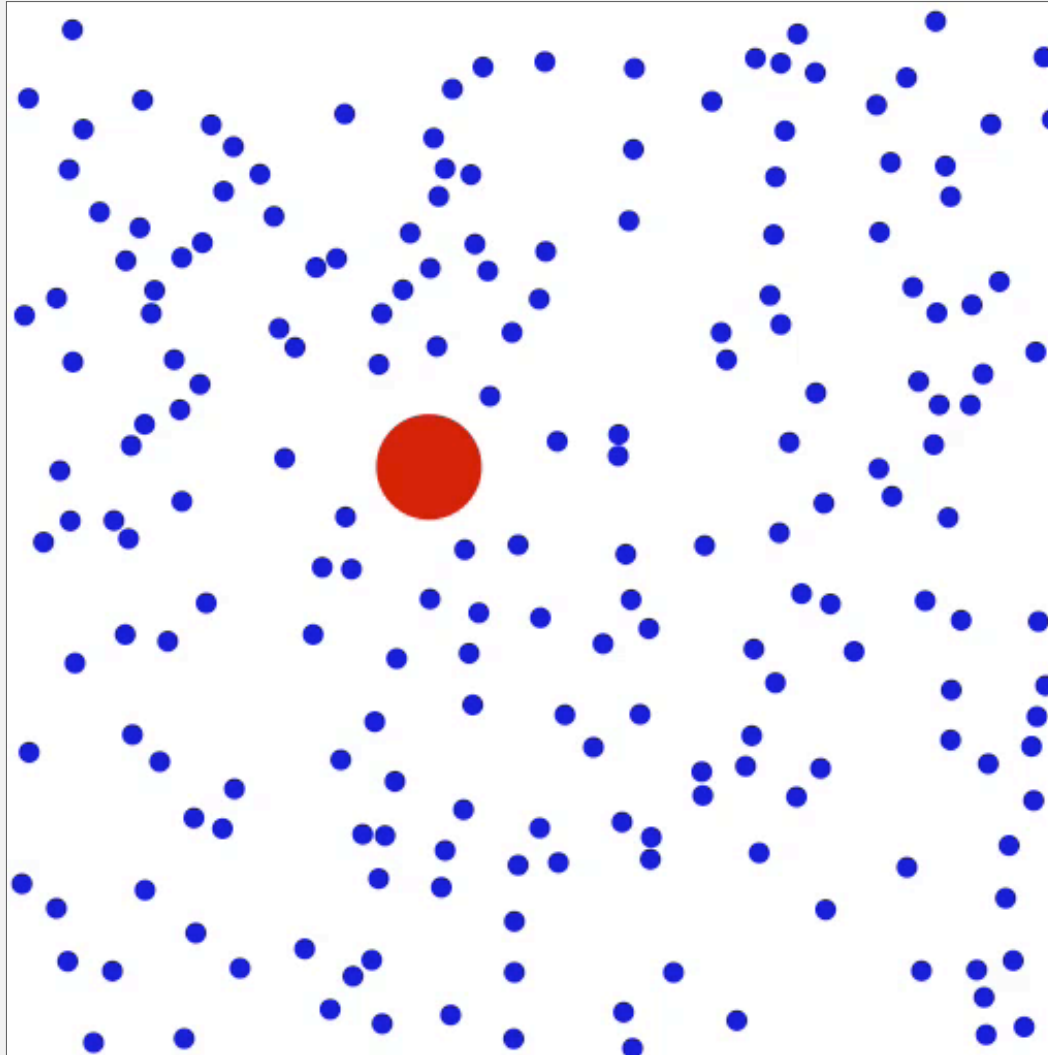
```
% java CollisionSystem < billiards.txt
```



## Particle collision simulation: example 3

---

```
% java CollisionSystem < brownian.txt
```



# Particle collision simulation: example 4

---

```
% java CollisionSystem < diffusion.txt
```

