

## 2.4 PRIORITY QUEUES



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

## ▶ API

- ▶ elementary implementations
- ▶ binary heaps
- ▶ heapsort
- ▶ event-driven simulation

## Priority queue

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

**Stack.** Remove the item most recently added.

**Queue.** Remove the item least recently added.

**Randomized queue.** Remove a random item.

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

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

## Priority queue API

**Requirement.** Generic items are comparable.

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

---

    MaxPQ()                    *create a priority queue*

    MaxPQ(maxN)              *create a priority queue of initial capacity maxN*

void insert(Key v)          *insert a key into the priority queue*

Key max()                    *return the largest key*

Key delMax()                 *return and remove the largest key*

boolean isEmpty()            *is the priority queue empty?*

int size()                    *number of entries in the priority queue*

---

**API for a generic 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).

```
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();  
}
```

use a min-oriented pq

Transaction data type is Comparable

pq contains largest M items

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	$N$	$M$

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



## 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			
<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: 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()  
as for sorting

## Priority queue elementary implementations

Challenge. Implement **all** operations efficiently.

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

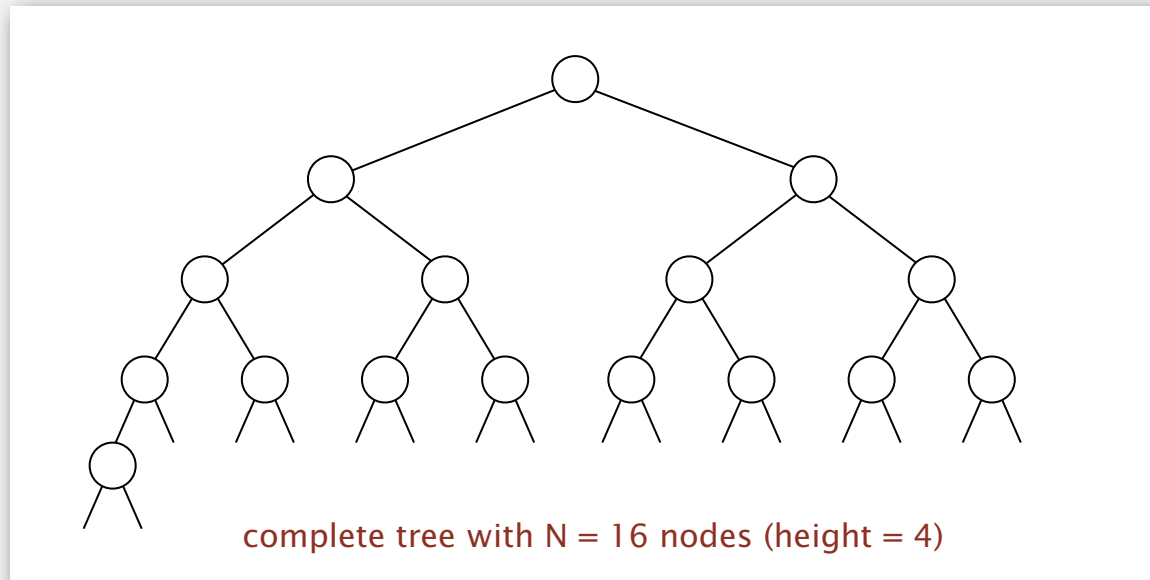
implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
<b>goal</b>	<b>log N</b>	<b>log N</b>	<b>log N</b>

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

## Binary tree

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

Complete tree. Perfectly balanced, except for bottom level.



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

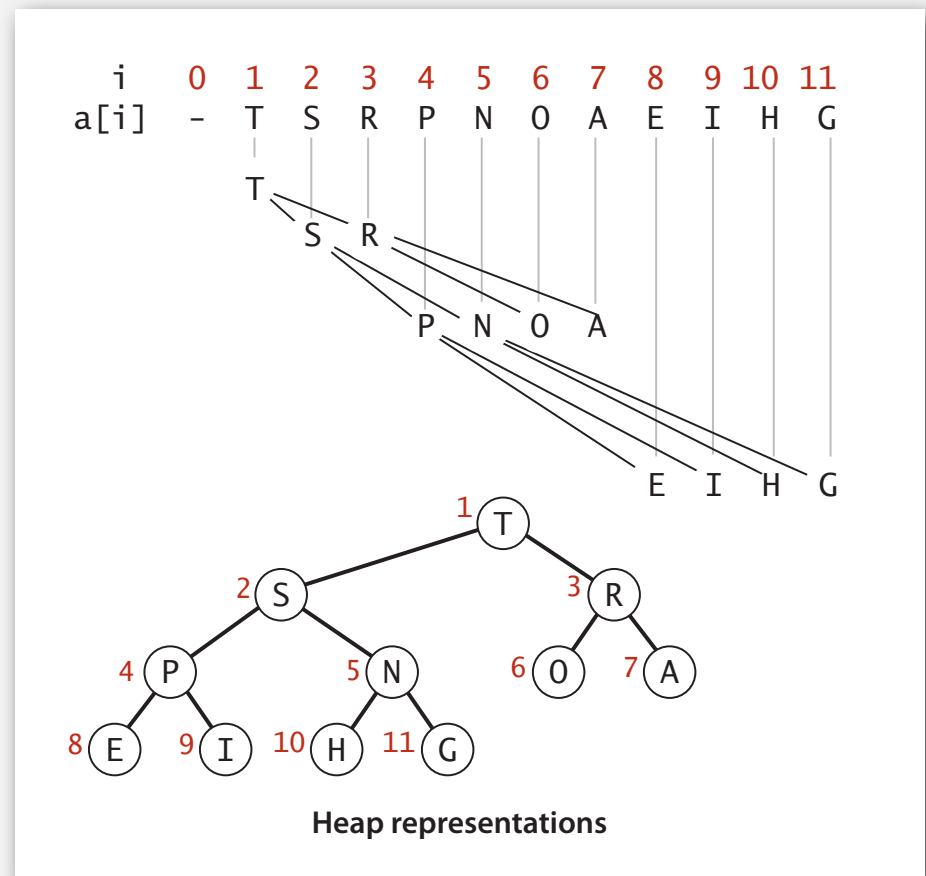
**Binary heap.** Array representation of a heap-ordered complete binary tree.

**Heap-ordered binary tree.**

- Keys in nodes.
- No smaller than children's keys.

**Array representation.**

- Take nodes in **level** order.
- No explicit links needed!

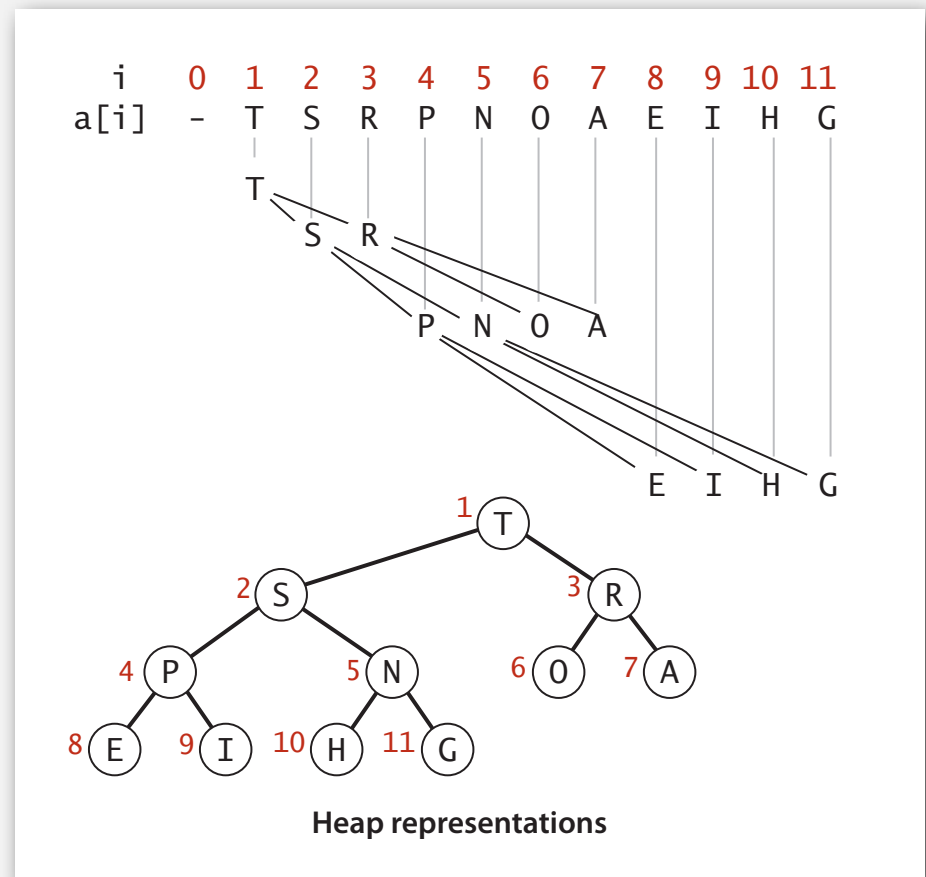


## Binary heap properties

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

**Proposition.** Can use array indices to move through tree. ↙ indices start at 1

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





## Promotion in a heap

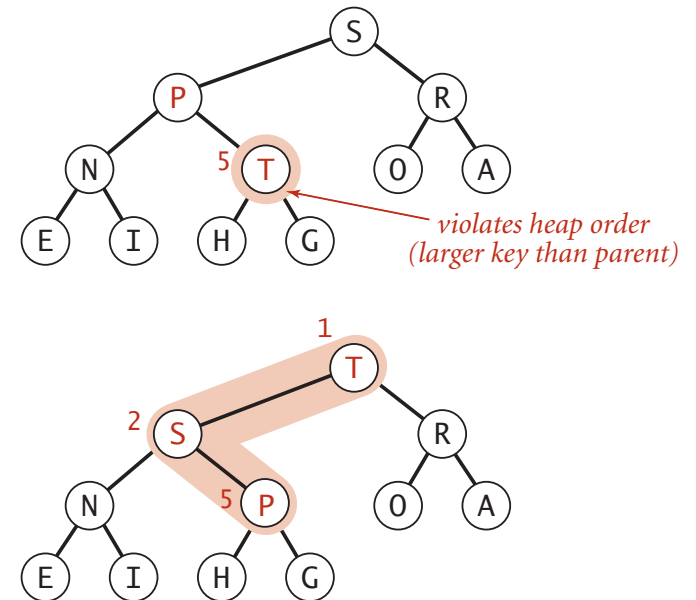
**Scenario.** Node's key becomes **larger** key than its parent's key.

To eliminate the violation:

- Exchange key in node 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



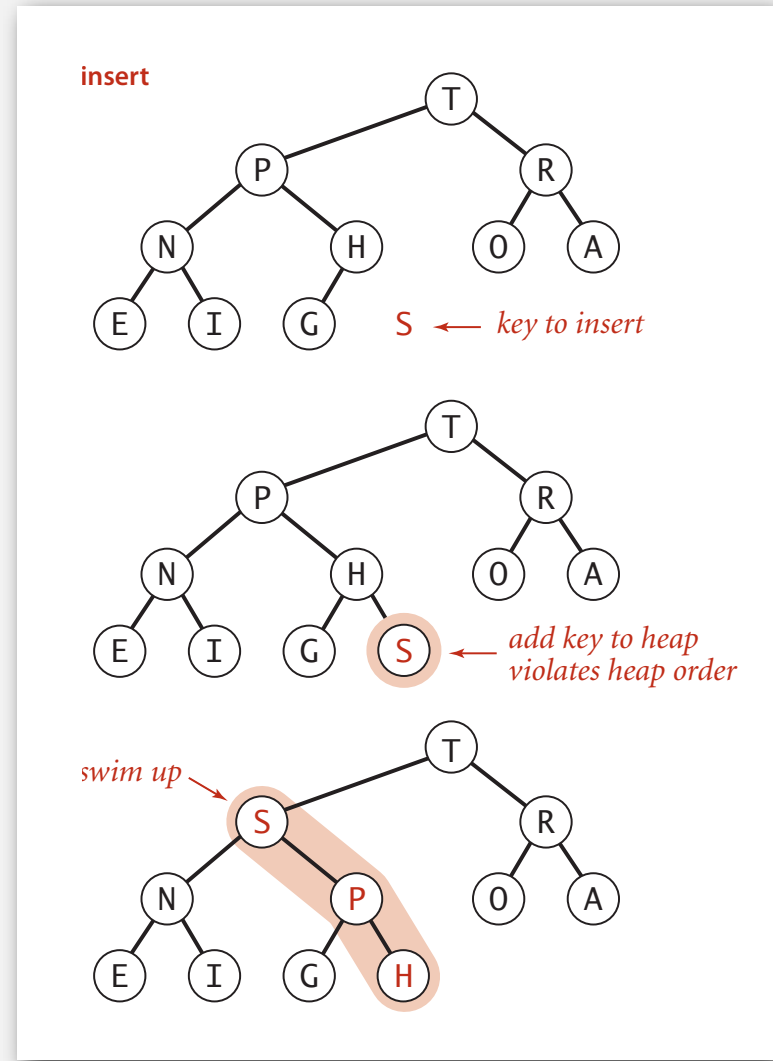
**Peter principle.** Node promoted to level of incompetence.

## 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

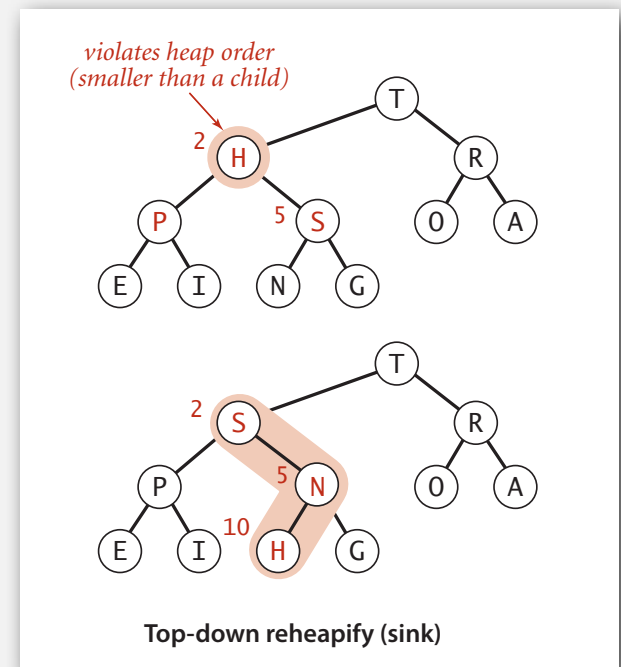
**Scenario.** Node's key becomes **smaller** than one (or both) of its children's keys.

To eliminate the violation:

- Exchange key in node 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



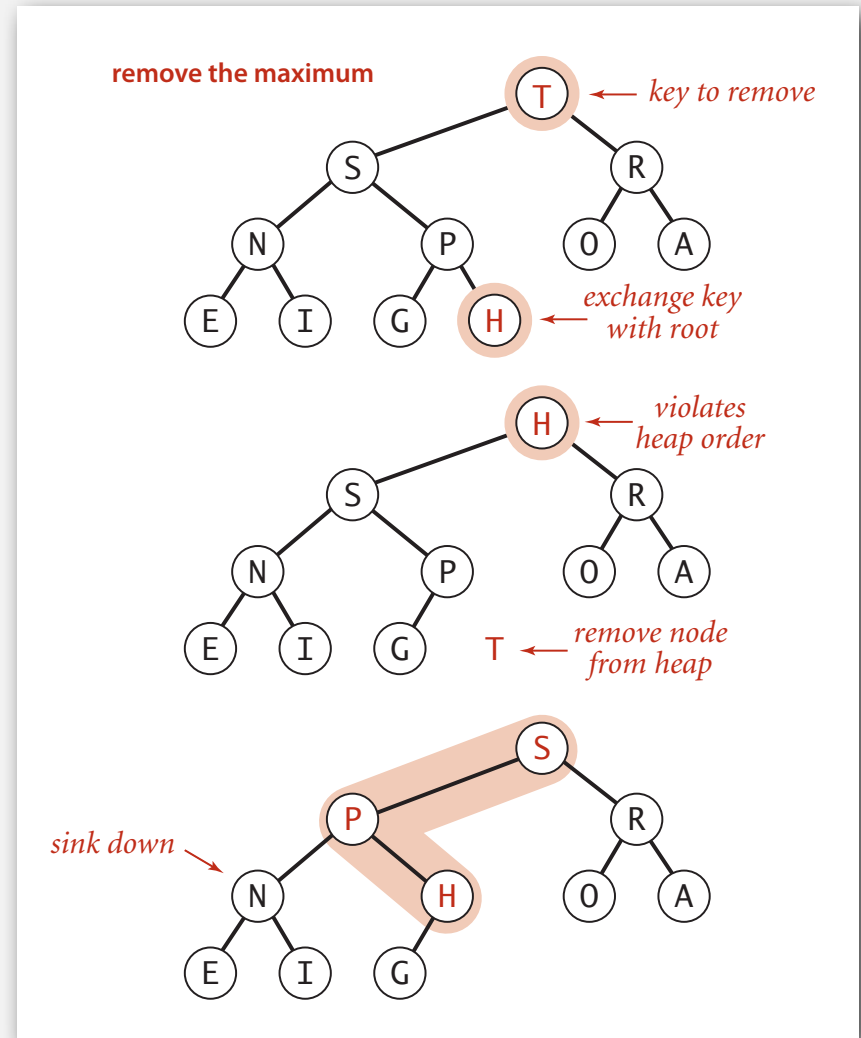
**Power struggle.** Better subordinate promoted.

## 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; ← prevent loitering
    return max;
}
```



## Binary heap demo

## 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




### Minimum-oriented priority queue.

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

### Other operations.

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

easy to 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.

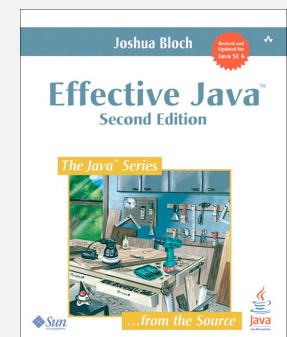
- Simplifies debugging.
- Safer in presence of hostile code.
- Simplifies concurrent programming.
- Safe to use as key in priority queue or symbol table.



**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)*



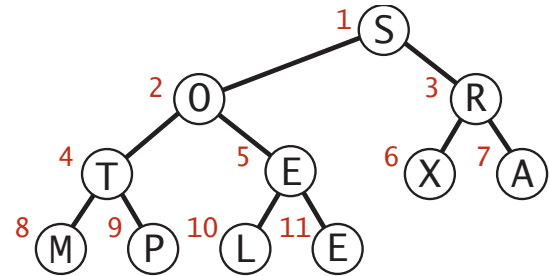
- ▶ API
- ▶ elementary implementations
- ▶ binary heaps
- ▶ **heapsort**
- ▶ event-driven simulation

# Heapsort

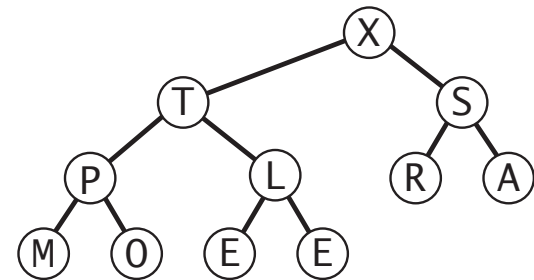
## Basic plan for in-place sort.

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

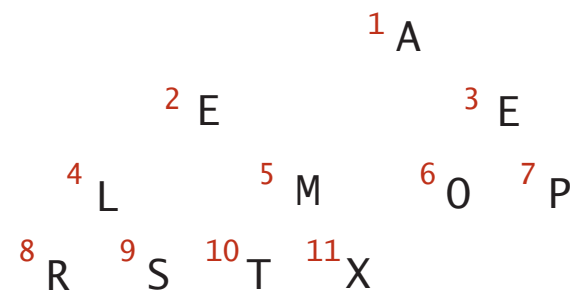
start with array of keys  
in arbitrary order



build a max-heap  
(in place)



sorted result  
(in place)

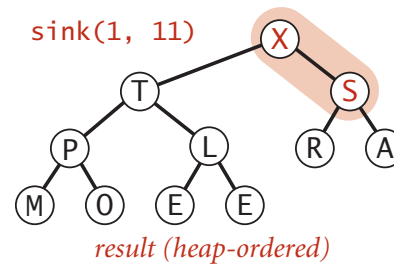
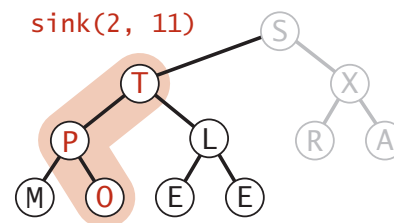
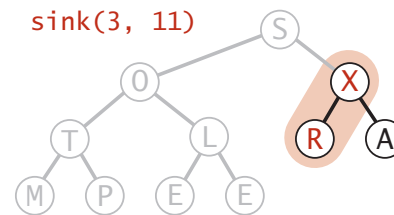
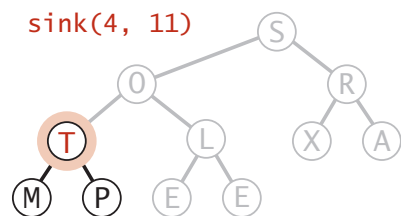
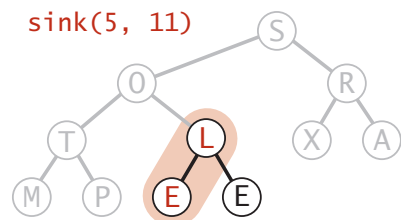
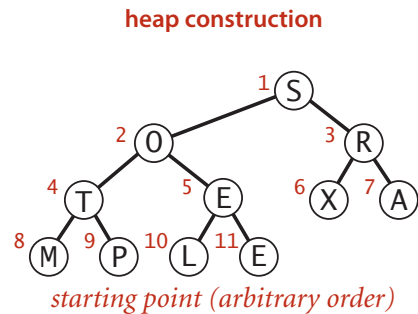


## Heapsort demo

# Heapsort: heap construction

First pass. Build heap using bottom-up method.

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



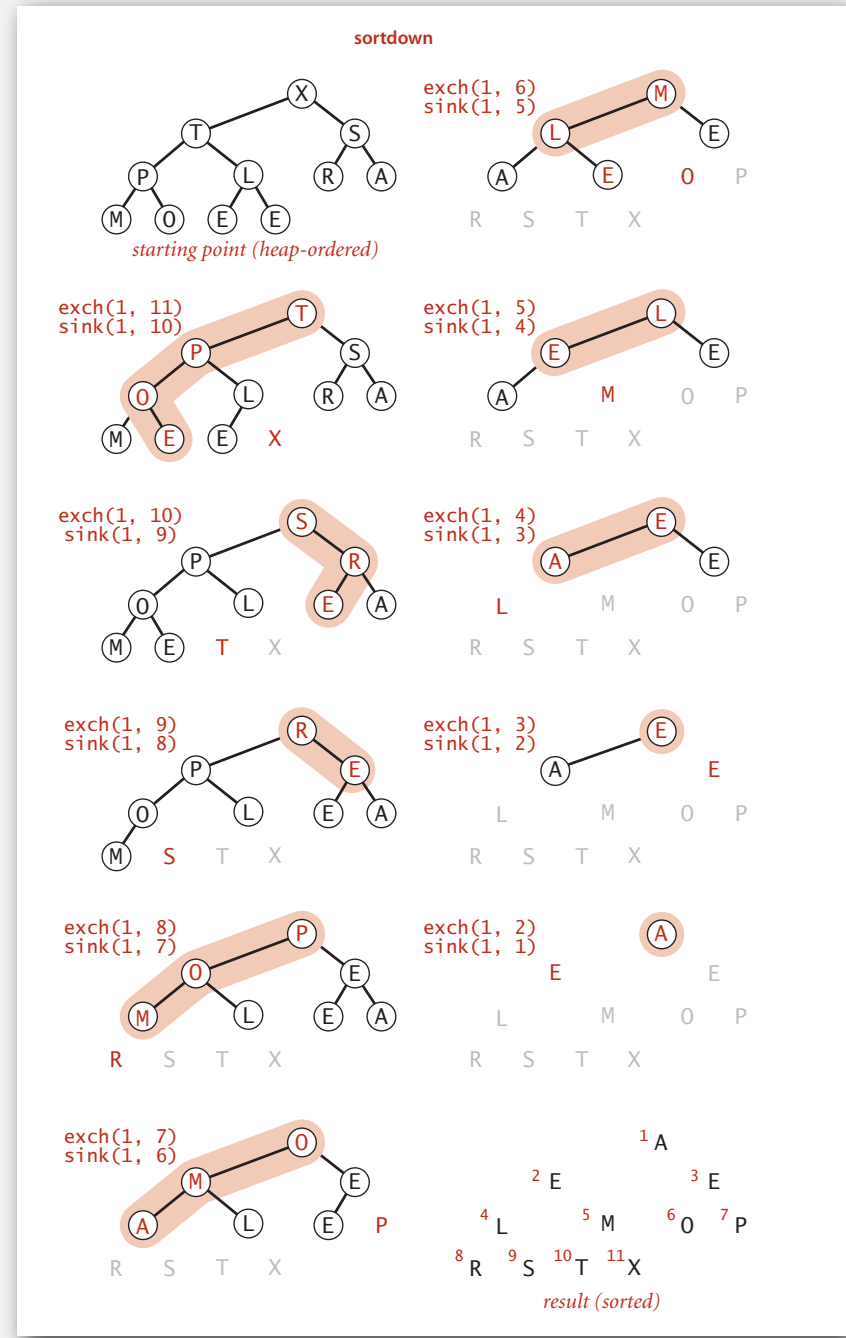
# Heapsort: sortdown

## 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 use 1-based 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: mathematical analysis

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

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

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

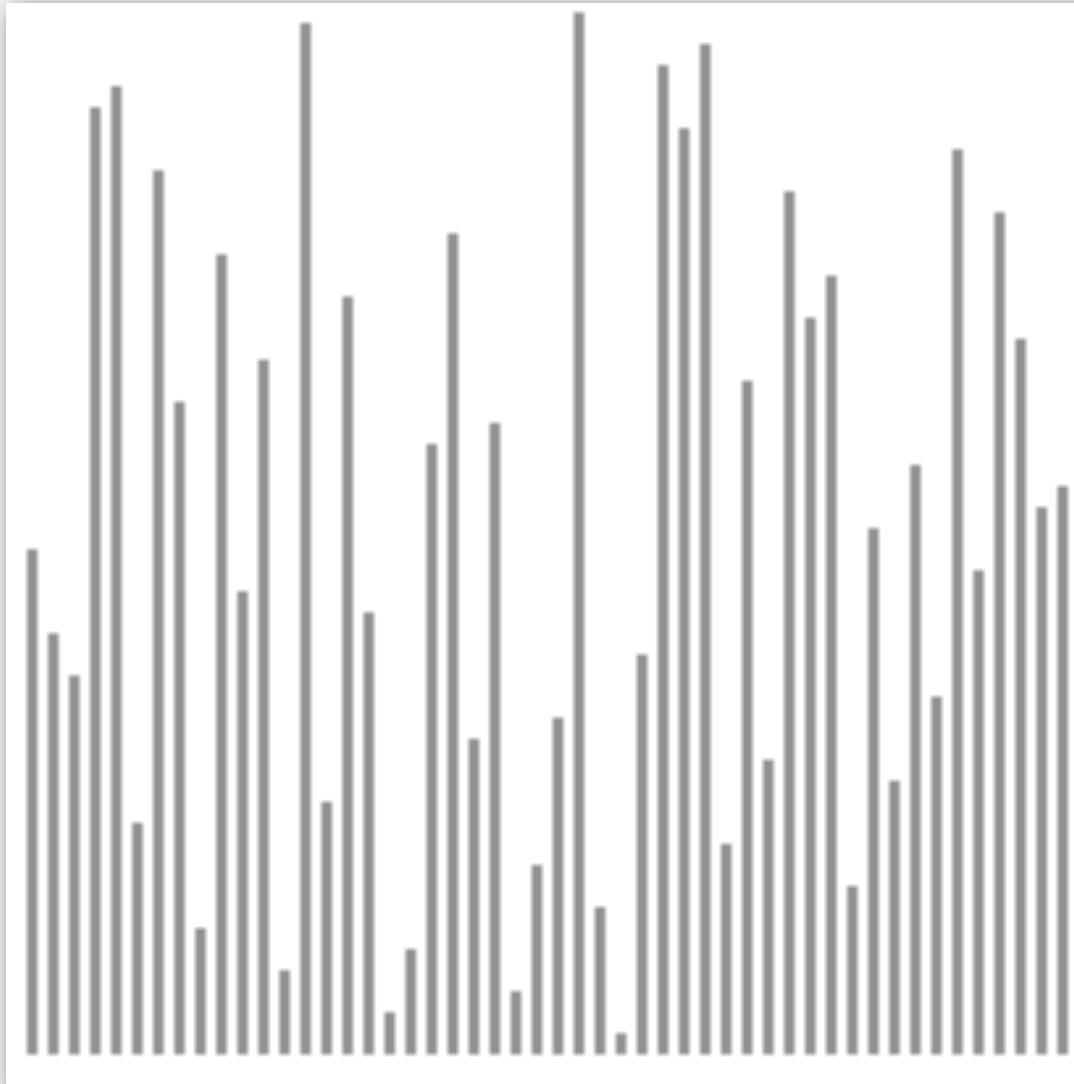
- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ←  $N \log N$  worst-case quicksort possible, not practical
- Heapsort: yes!

**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.

# Heapsort animation

50 random elements



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

- ▲ algorithm position
- ▬ in order
- ▬ not in order

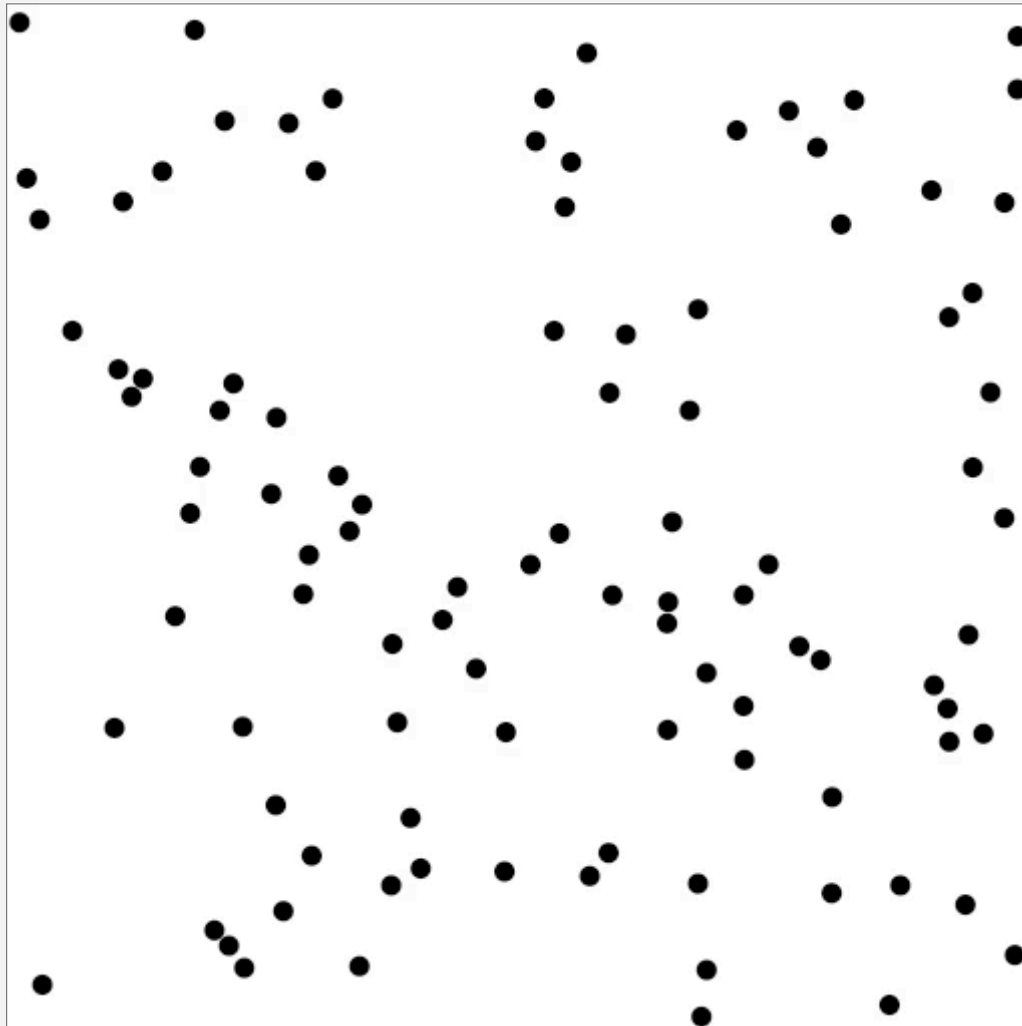
## Sorting algorithms: summary

	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

- ▶ API
- ▶ 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.

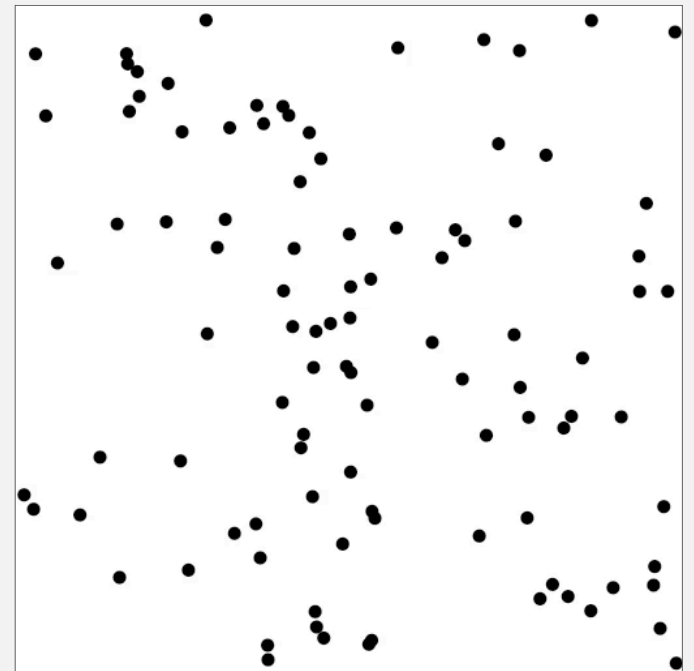
## Warmup: bouncing balls

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

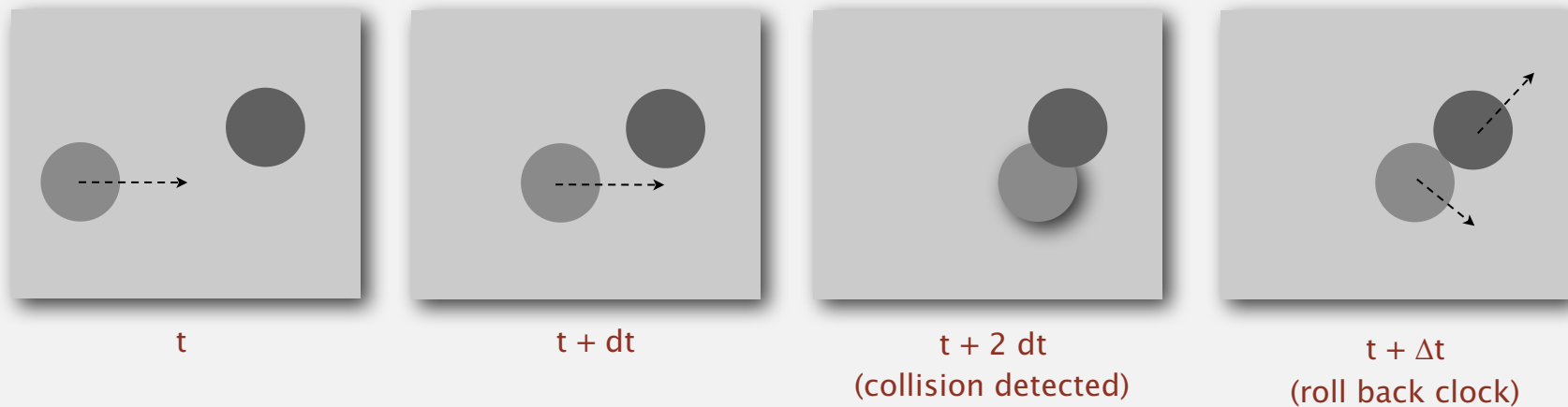


**Missing.** Check for balls colliding with **each other**.

- 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.

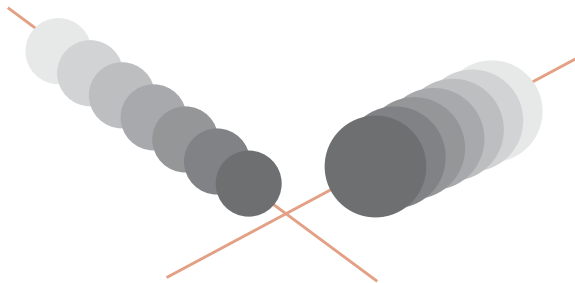


## Time-driven simulation

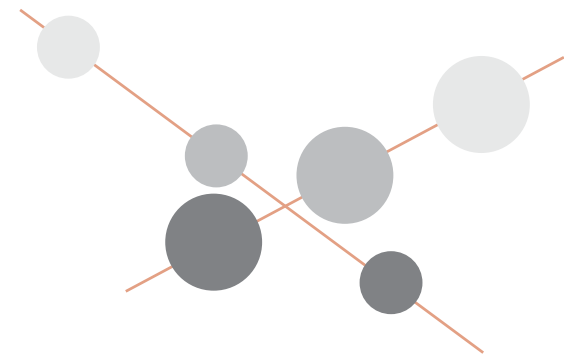
### Main drawbacks.

- $\sim N^2 / 2$  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)

**dt too small: excessive computation**



**dt too large: may miss collisions**



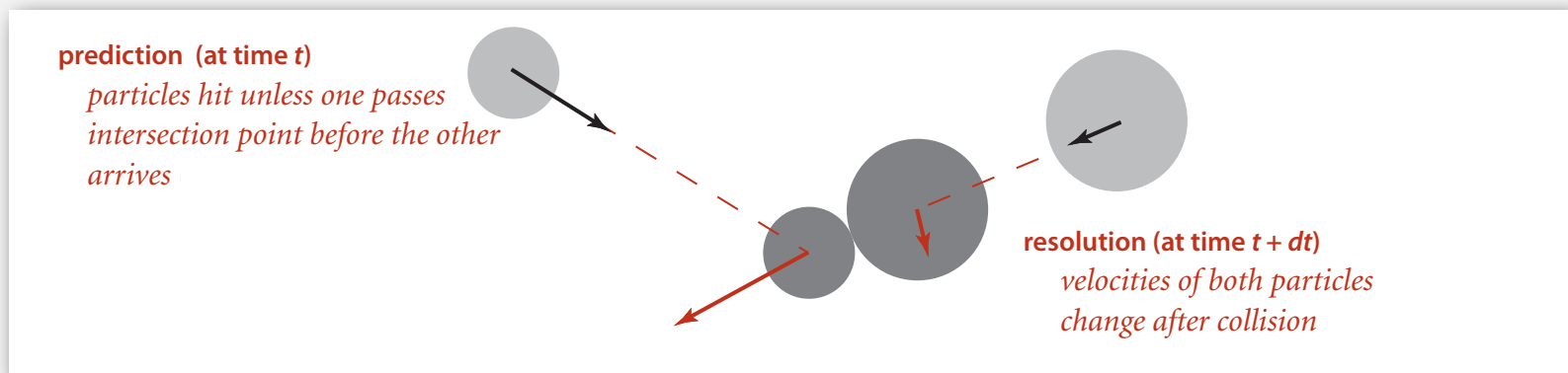
## Event-driven simulation

Change state only when something happens.

- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain **PQ** of collision events, prioritized by time.
- Remove the 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?

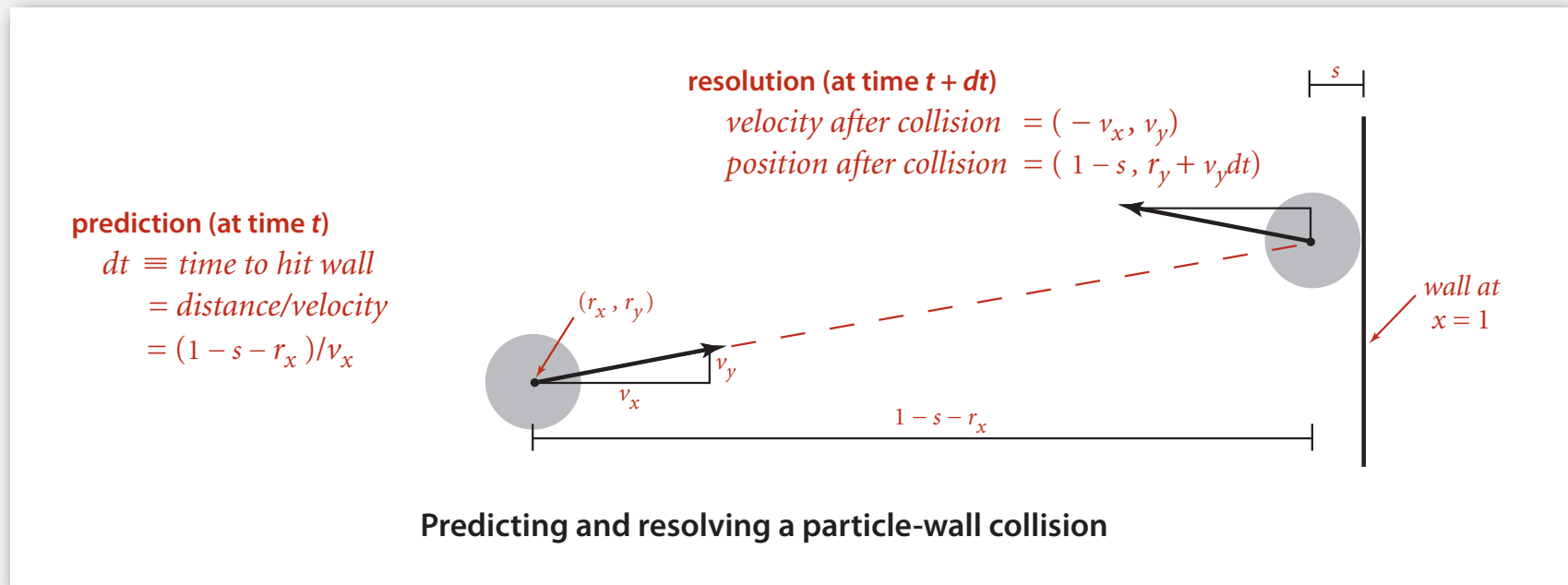
**Collision resolution.** If collision occurs, update colliding particle(s) according to laws of elastic collisions.



## Particle-wall collision

### Collision prediction and resolution.

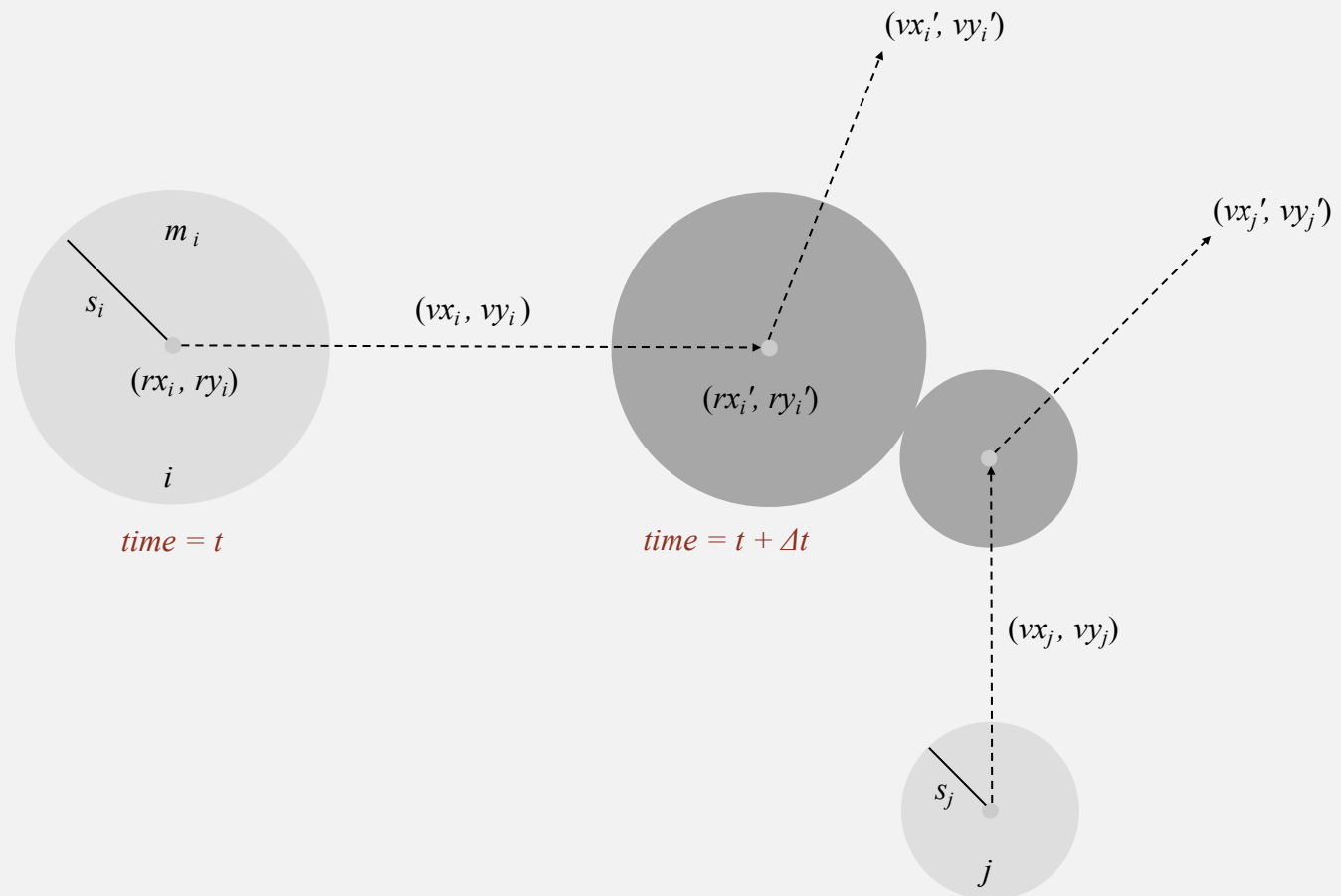
- Particle of radius  $s$  at position  $(r_x, r_y)$ .
- Particle moving in unit box with velocity  $(v_x, v_y)$ .
- Will it collide with a vertical wall? If so, when?



## 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?

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

$$d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v) (\Delta r \cdot \Delta r - \sigma^2) \quad \sigma = \sigma_i + \sigma_j$$

$$\Delta v = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j)$$

$$\Delta r = (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j)$$

$$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$$

$$\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2$$

$$\Delta v \cdot \Delta r = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)$$

**Important note: This is high-school 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)

$$Jx = \frac{J \Delta rx}{\sigma}, \quad Jy = \frac{J \Delta ry}{\sigma}, \quad J = \frac{2 m_i m_j (\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

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

**Important note: This is high-school 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

## Particle-particle collision and resolution implementation

```
public double timeToHit(Particle that)
{
    if (this == that) return INFINITY;
    double dx = that.rx - this.rx, dy = that.ry - this.ry;
    double dvx = that.vx - this.vx; dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    if( dvdr > 0) return INFINITY; ← no collision
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = this.radius + that.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY; ←
    return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

```
public void bounceOff(Particle that)
{
    double dx = that.rx - this.rx, dy = that.ry - this.ry;
    double dvx = that.vx - this.vx, dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    double dist = this.radius + that.radius;
    double J = 2 * this.mass * that.mass * dvdr / ((this.mass + that.mass) * dist);
    double Jx = J * dx / dist;
    double Jy = J * dy / dist;
    this.vx += Jx / this.mass;
    this.vy += Jy / this.mass;
    that.vx -= Jx / that.mass;
    that.vy -= Jy / that.mass;
    this.count++;
    that.count++;
} Important note: This is high-school physics, so we won't be testing you on it!
```

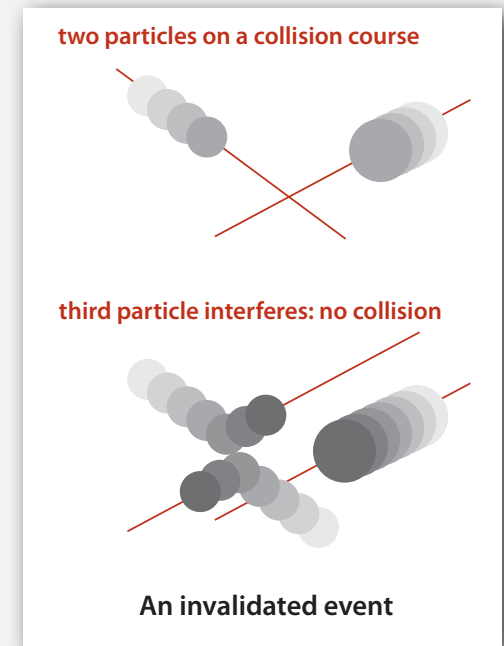
## Collision system: event-driven simulation main loop

### Initialization.

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



“potential” since collision may not happen 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 class Event implements Comparable<Event>
{
    private double time;           // time of event
    private Particle a, b;        // particles involved in event
    private int countA, countB;   // collision counts for a and b

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

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

    public boolean isValid()
    { }
}
```

← create event

← ordered by time

← invalid if  
intervening  
collision

## Collision system implementation: skeleton

```
public class CollisionSystem
{
    private MinPQ<Event> pq;           // the priority queue
    private double t = 0.0;           // simulation clock time
    private Particle[] particles;     // the array of particles

    public CollisionSystem(Particle[] particles) { }

    private void predict(Particle a)
    {
        if (a == null) return;
        for (int i = 0; i < N; i++)
        {
            double dt = a.timeToHit(particles[i]);
            pq.insert(new Event(t + dt, a, particles[i]));
        }
        pq.insert(new Event(t + a.timeToHitVerticalWall(), a, null));
        pq.insert(new Event(t + a.timeToHitHorizontalWall(), null, a));
    }

    private void redraw() { }

    public void simulate() { /* see next slide */ }
}
```

add to PQ all particle-wall and particle-particle collisions involving this particle

## Collision system implementation: main event-driven simulation loop

```
public void simulate()
{
```

```
    pq = new MinPQ<Event>();
    for(int i = 0; i < N; i++) predict(particles[i]);
    pq.insert(new Event(0, null, null));
```

initialize PQ with  
collision events and  
redraw event

```
    while(!pq.isEmpty())
    {
```

```
        Event event = pq.delMin();
        if(!event.isValid()) continue;
        Particle a = event.a;
        Particle b = event.b;
```

get next event

```
        for(int i = 0; i < N; i++)
            particles[i].move(event.time - t);
        t = event.time;
```

update positions  
and time

```
        if (a != null && b != null) a.bounceOff(b);
        else if (a != null && b == null) a.bounceOffVerticalWall();
        else if (a == null && b != null) b.bounceOffHorizontalWall();
        else if (a == null && b == null) redraw();
```

process event

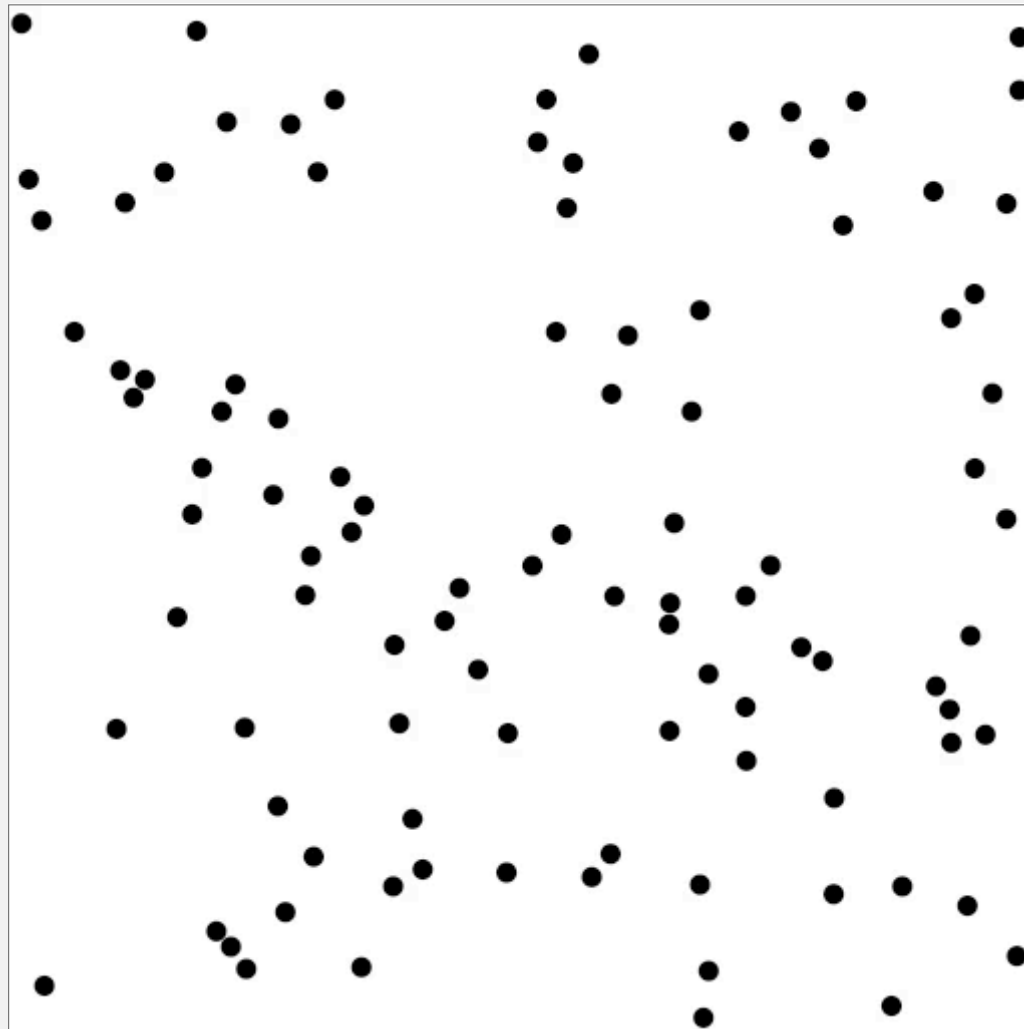
```
        predict(a);
        predict(b);
```

predict new events  
based on changes

```
    }
}
```

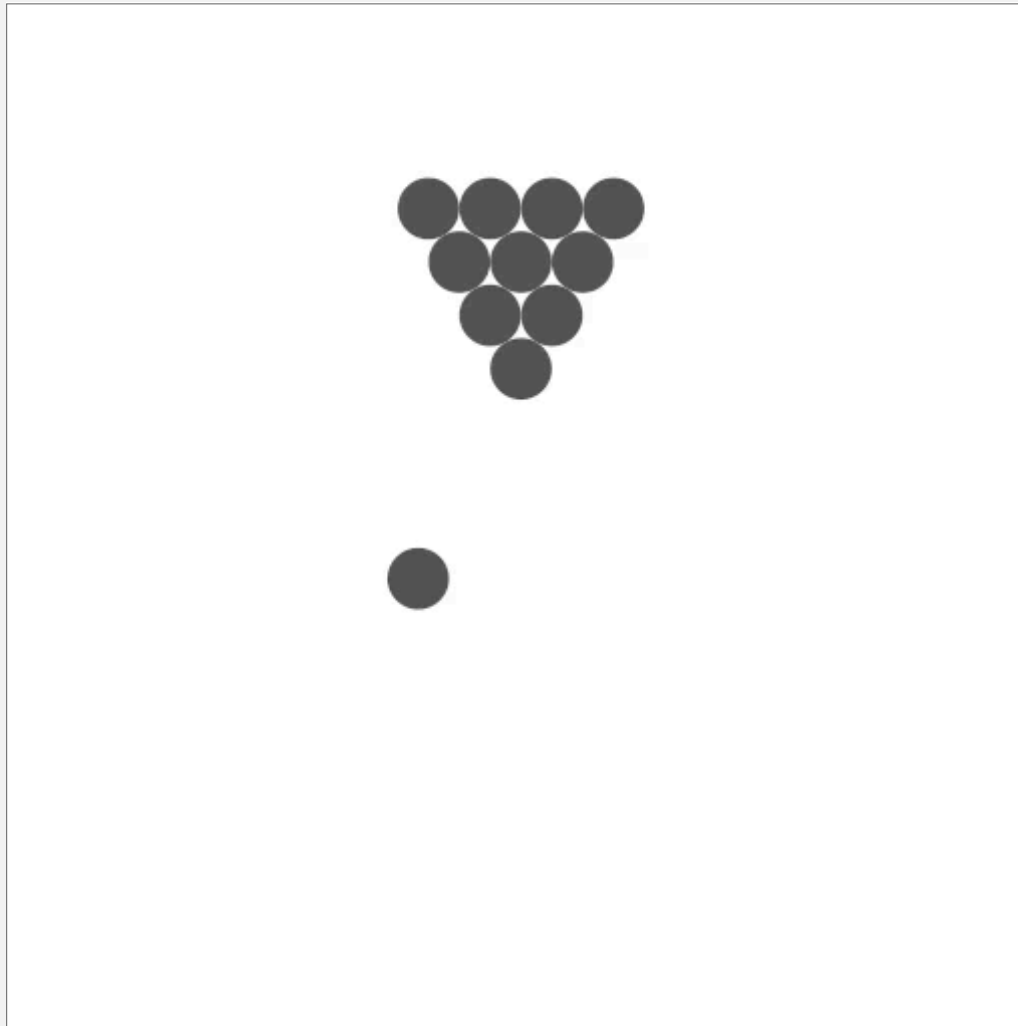
## Simulation example 1

```
% java CollisionSystem 100
```



## Simulation example 2

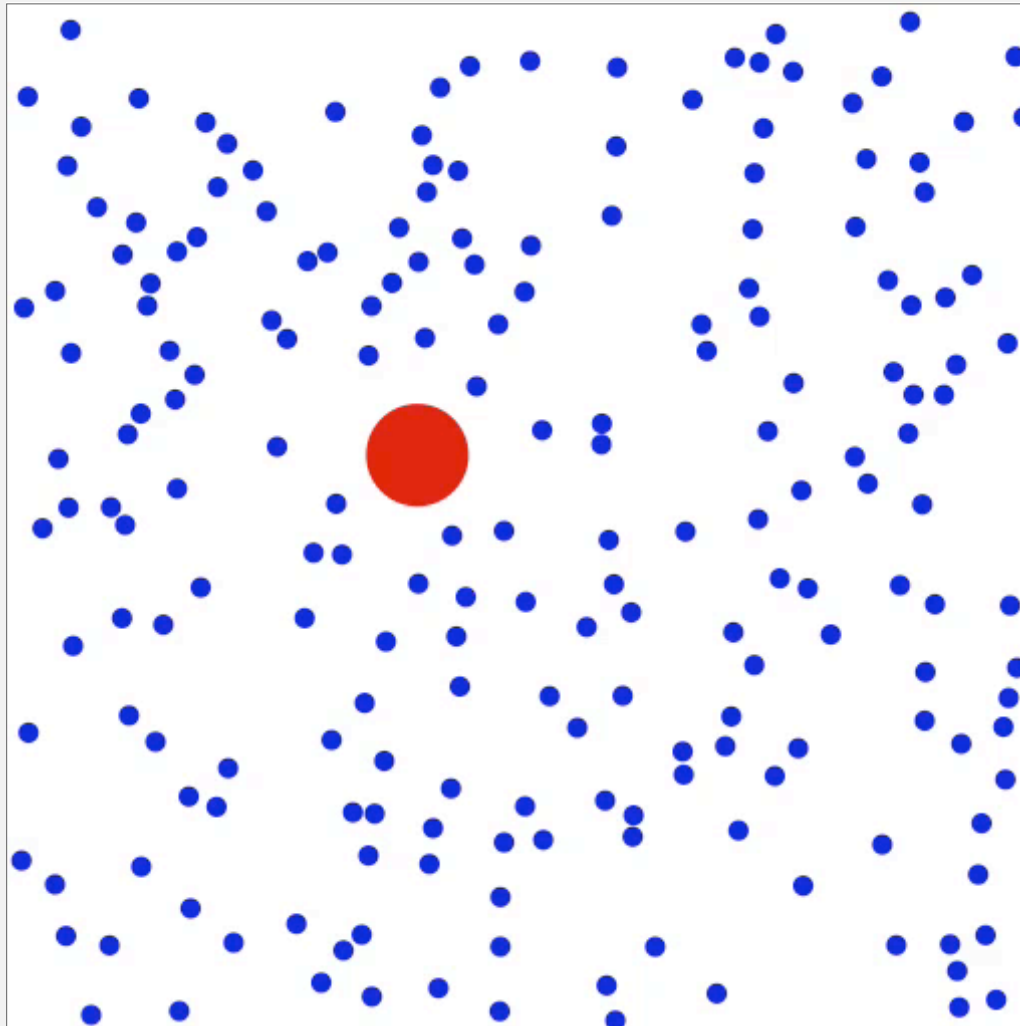
```
% java CollisionSystem < billiards.txt
```





## Simulation example 3

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



## Simulation example 4

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

