

Priority Queues

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

References:

Algorithms in Java, Chapter 9

<http://www.cs.princeton.edu/introalgsds/34pq>

▶ API

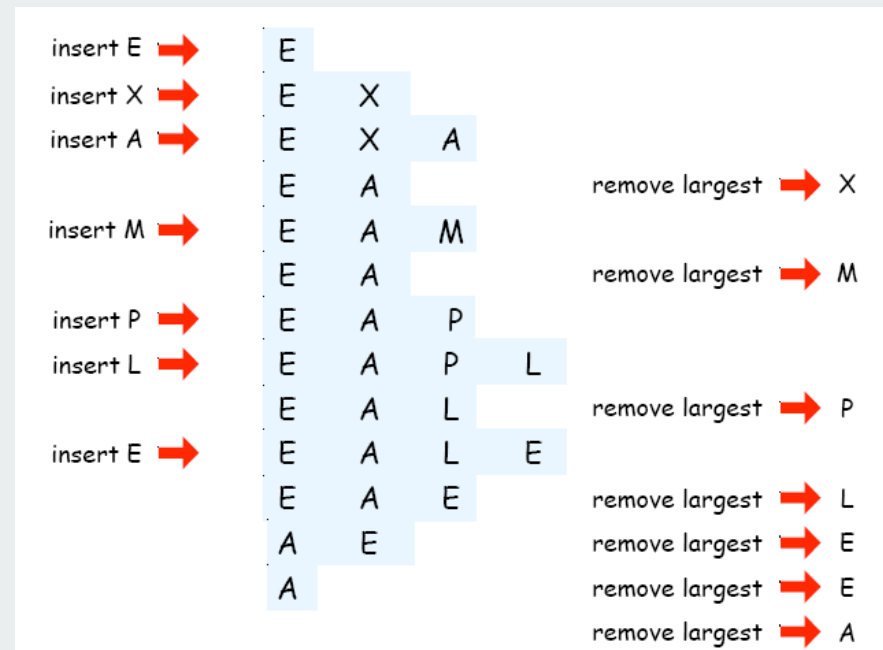
- ▶ elementary implementations
- ▶ binary heaps
- ▶ heapsort
- ▶ event-driven simulation

Priority Queues

Data. Items that can be compared.

Basic operations.

- **Insert.**
- **Remove largest.** defining ops
- **Copy.**
- **Create.**
- **Destroy.** generic ops
- **Test if empty.**



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

Problem: Find the largest M of a stream of N elements.

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

Constraint. Not enough memory to store N elements.

Solution. Use a priority queue.

| Operation | time | space |
|----------------|-----------|-------|
| sort | $N \lg N$ | N |
| elementary PQ | $M N$ | M |
| binary heap | $N \lg M$ | M |
| best in theory | N | M |

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

while(!StdIn.isEmpty())
{
    String s = StdIn.readLine();
    t = new Transaction(s);
    pq.insert(t);
    if (pq.size() > M)
        pq.delMin();
}

while (!pq.isEmpty())
    System.out.println(pq.delMin());
```

▶ API

▶ **elementary implementations**

▶ binary heaps

▶ heapsort

▶ event-driven simulation

Priority queue: unordered array implementation


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

    public UnorderedPQ(int maxN)
    {    pq = (Item[]) new Comparable[maxN];    }

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

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

    public Item 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

Priority queue elementary implementations

| Implementation | Insert | Del Max |
|-----------------|--------|---------|
| unordered array | 1 | N |
| ordered array | N | 1 |

worst-case asymptotic costs for PQ with N items

| | | |
|-------------------|--|--|
| <i>insert P</i> | <div>P</div> | <div>P</div> |
| <i>insert Q</i> | <div>P</div> <div>Q</div> | <div>P</div> <div>Q</div> |
| <i>insert E</i> | <div>P</div> <div>Q</div> <div>E</div> | <div>E</div> <div>P</div> <div>Q</div> |
| <i>delmax (Q)</i> | <div>P</div> <div>E</div> | <div>E</div> <div>P</div> |
| <i>insert X</i> | <div>P</div> <div>E</div> <div>X</div> | <div>E</div> <div>P</div> <div>X</div> |
| <i>insert A</i> | <div>P</div> <div>E</div> <div>X</div> <div>A</div> | <div>A</div> <div>E</div> <div>P</div> <div>X</div> |
| <i>insert M</i> | <div>P</div> <div>E</div> <div>X</div> <div>A</div> <div>M</div> | <div>A</div> <div>E</div> <div>M</div> <div>P</div> <div>X</div> |
| <i>delmax (X)</i> | <div>P</div> <div>E</div> <div>M</div> <div>A</div> | <div>A</div> <div>E</div> <div>M</div> <div>P</div> |
| | <i>unordered</i> | <i>ordered</i> |

Challenge. Implement **both** operations efficiently.

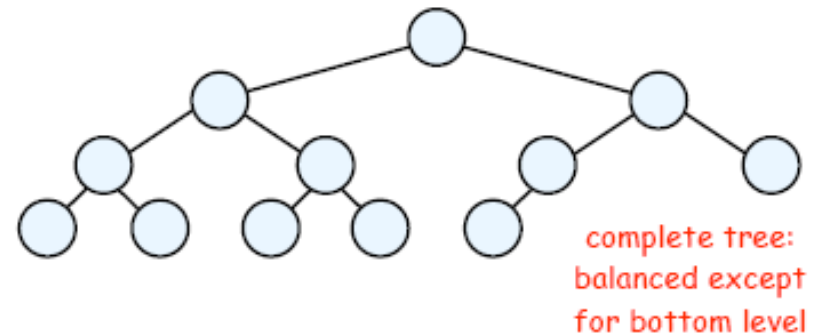
- ▶ API
- ▶ elementary implementations
- ▶ **binary heaps**
- ▶ heapsort
- ▶ event-driven simulation

Binary Heap

Heap: Array representation of a heap-ordered complete binary tree.

Binary tree.

- Empty **or**
- Node with links to left and right trees.



Binary Heap

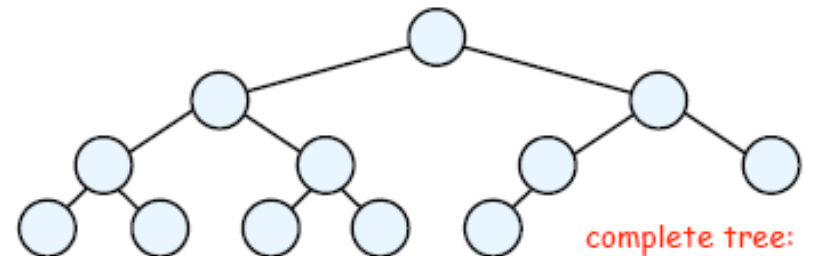
Heap: Array representation of a heap-ordered complete binary tree.

Binary tree.

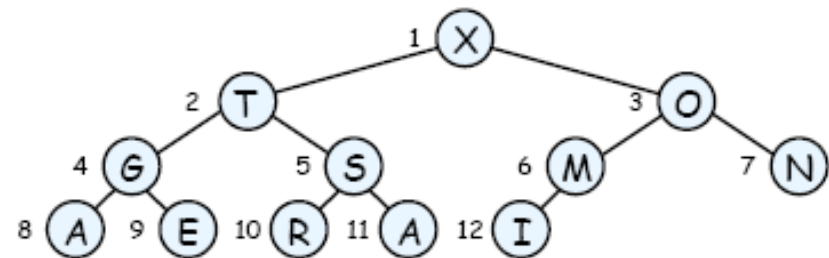
- Empty **or**
- Node with links to left and right trees.

Heap-ordered binary tree.

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



complete tree:
balanced except
for bottom level



Binary Heap

Heap: Array representation of a heap-ordered complete binary tree.

Binary tree.

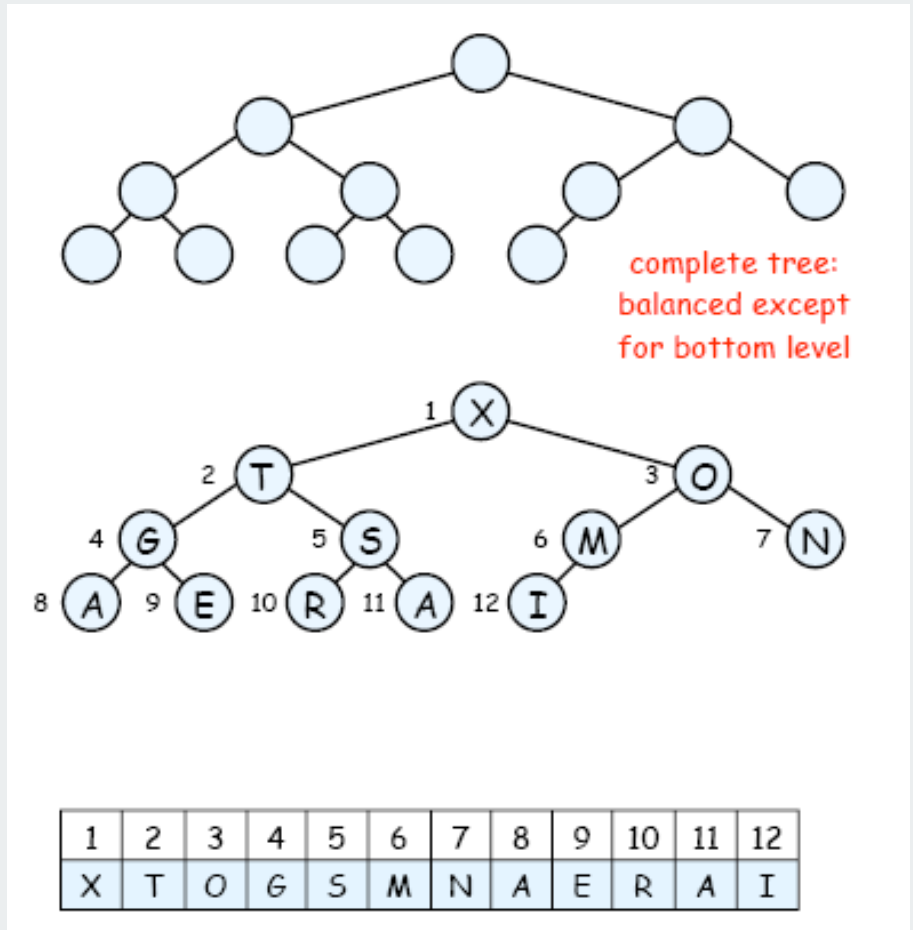
- Empty **or**
- Node with links to left and right trees.

Heap-ordered binary tree.

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

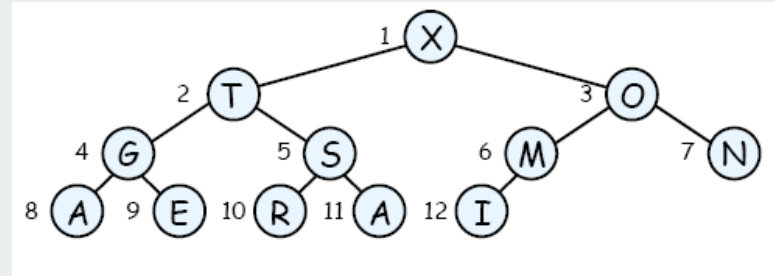
Array representation.

- Take nodes in **level** order.
- No explicit links needed since tree is complete.



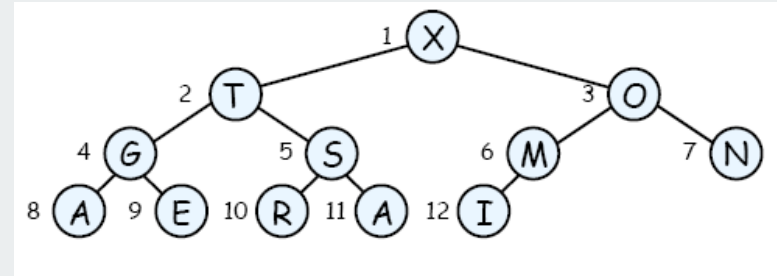
Binary Heap Properties

Property A. Largest key is at root.



Binary Heap Properties

Property A. Largest key is at root.



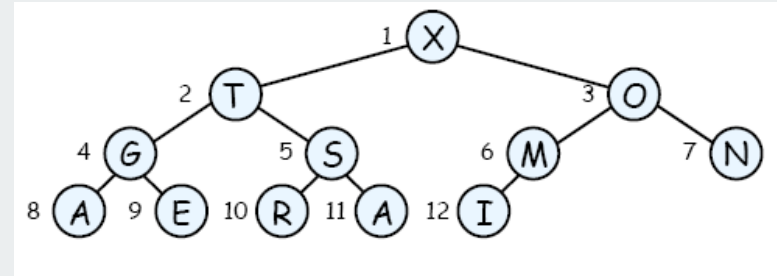
Property B. Can use array indices to move through tree.

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| X | T | O | G | S | M | N | A | E | R | A | I |

Binary Heap Properties

Property A. Largest key is at root.



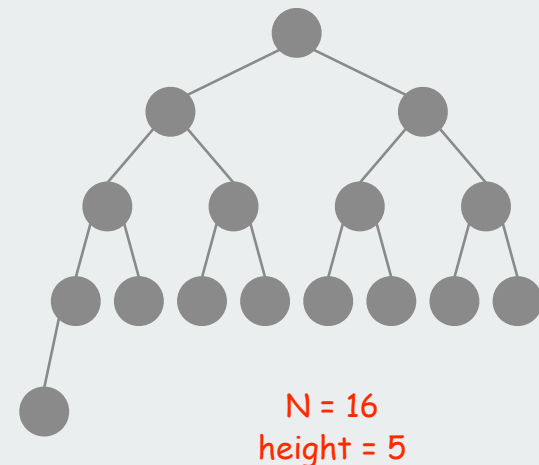
Property B. Can use array indices to move through tree.

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| X | T | O | G | S | M | N | A | E | R | A | I |

Property C. Height of N node heap is $1 + \lfloor \lg N \rfloor$.

↑
height increases only when
 N is a power of 2



Promotion In a Heap

Scenario. Exactly one node has a **larger** key than its parent.

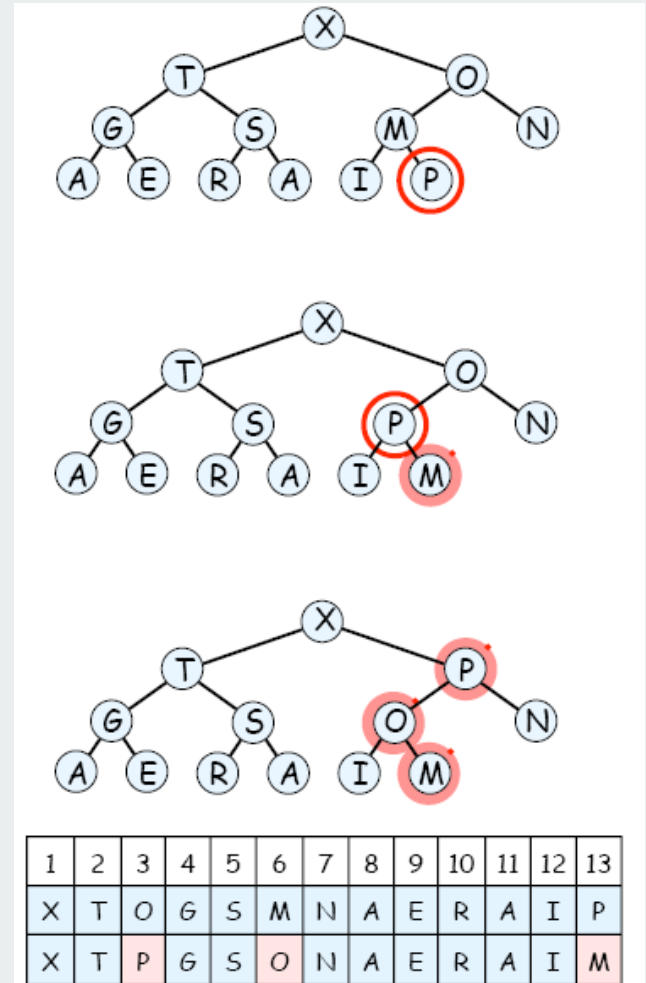
To eliminate the violation:

- Exchange with its 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

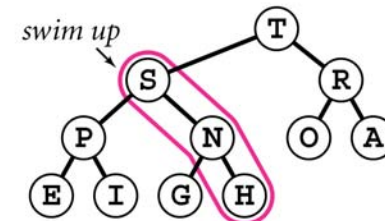
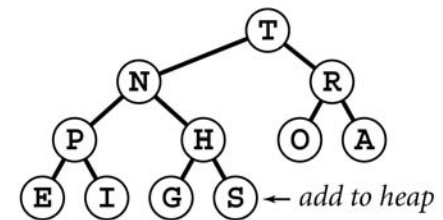
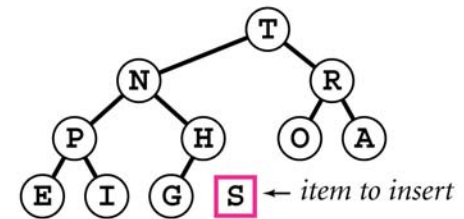
Peter principle: node promoted to level of incompetence.



Insert

Insert. Add node at end, then promote.

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



Demotion In a Heap

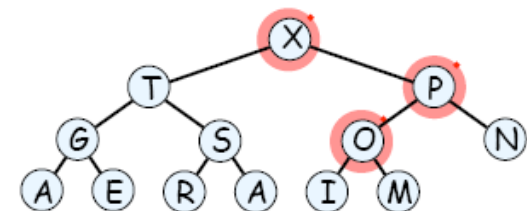
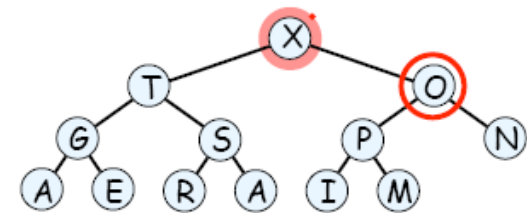
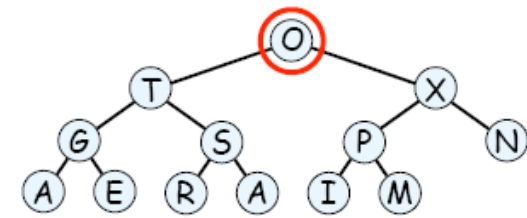
Scenario. Exactly one node has a **smaller** key than does a child.

To eliminate the violation:

- Exchange with 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



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| O | T | X | G | S | P | N | A | E | R | A | I | M |
| X | T | P | G | S | O | N | A | E | R | A | I | M |

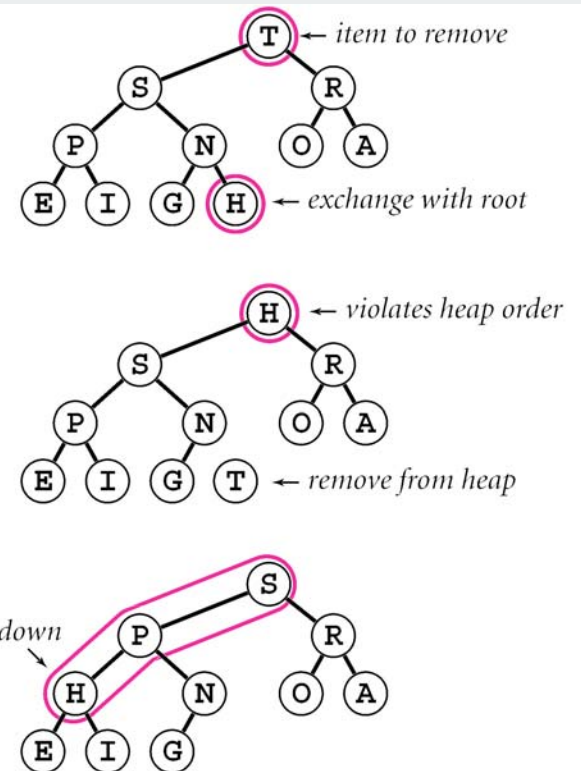
Power struggle: better subordinate promoted.

Remove the Maximum

Remove max. Exchange root with node at end, then demote.

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

← prevent loitering



Binary heap implementation summary

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

    public MaxPQ(int maxN)
    { . . . }
    public boolean isEmpty()
    { . . . }

    public void insert(Item x)
    { . . . }
    public Item delMax()
    { . . . }

    private void swim(int k)
    { . . . }
    private void sink(int k)
    { . . . }

    private boolean less(int i, int j)
    { . . . }
    private void      exch(int i, int j)
    { . . . }
}
```

← same as array-based PQ,
but allocate one extra element

← PQ ops

← heap helper functions

← array helper functions

Binary heap considerations

Minimum oriented priority queue

- replace `less()` with `greater()`
- implement `greater()`.

Array resizing

- add no-arg constructor
- apply repeated doubling.  leads to $O(\log N)$ amortized time per op

Immutability of keys.

- assumption: client does not change keys while they're on the PQ
- best practice: **use immutable keys**

Other operations.

- remove an arbitrary item.
 - change the priority of an item.
- 
- easy to implement with `sink()` and `swim()` [stay tuned]

Priority Queues Implementation Cost Summary

| Operation | Insert | Remove Max | Find Max |
|-----------------|---------|------------|----------|
| ordered array | N | 1 | 1 |
| ordered list | N | 1 | 1 |
| unordered array | 1 | N | N |
| unordered list | 1 | N | N |
| binary heap | $\lg N$ | $\lg N$ | 1 |

worst-case asymptotic costs for PQ with N items

Hopeless challenge. Make all ops $O(1)$.

Why hopeless?

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

Digression: Heapsort

First pass: build heap.

- Insert items into heap, one at a time.
- Or can use faster bottom-up method; see book.

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

Second pass: sort.

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

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

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

Property D. At most $2 N \lg N$ comparisons.

Significance of Heapsort

Q. Sort in $O(N \log N)$ worst-case without using extra memory?

A. Yes. Heapsort.

Not mergesort? Linear extra space.

← in-place merge possible, not practical

Not quicksort? Quadratic time in worst case.

← $O(N \log N)$ worst-case quicksort possible, not practical.

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

- inner loop longer than quicksort's.
- makes poor use of cache memory.

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 partly ordered |
| shell | x | | | | N | tight code |
| quick | x | | $N^2 / 2$ | $2N \ln N$ | $N \lg N$ | $N \log N$ probabilistic guarantee fastest in practice |
| merge | | x | $N \lg N$ | $N \lg N$ | $N \lg N$ | $N \log N$ guarantee, stable |
| heap | x | | $2N \lg N$ | $2N \lg N$ | $N \lg N$ | $N \log N$ guarantee, in-place |

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

Bouncing balls (COS 126)


```
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();
                balls[i].draw();
            }
            StdDraw.show(50);
        }
    }
}
```

Review

Bouncing balls (COS 126)

```
public class Ball
{
    private double rx, ry;    // position
    private double vx, vy;    // velocity
    private double radius;    // radius
    public Ball()
    { ... initialize position and velocity ... }
    public void move()
    {
        if ((rx + vx < radius) || (rx + vx > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy < radius) || (ry + vy > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx;
        ry = ry + vy;
    }
    public void draw()
    { StdDraw.filledCircle(rx, ry, radius); }
}
```

checks for
colliding with
walls



Missing: check for balls colliding with **each other**

- physics problems: when? what effect?
- CS problems: what object does the checks? too many checks?

Molecular dynamics simulation of hard spheres

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

Hard sphere model.

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

temperature, pressure,
diffusion constant

motion of individual
atoms and molecules

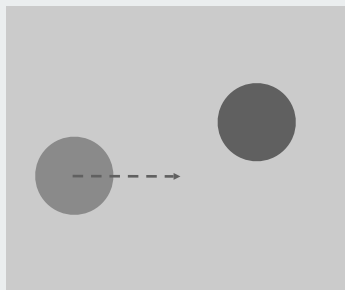
Significance. Relates **macroscopic** observables to **microscopic** dynamics.

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

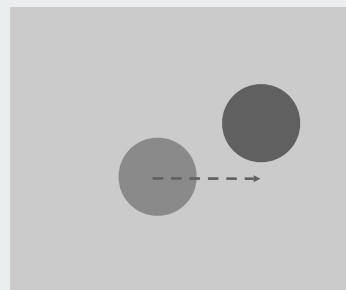
Time-driven simulation

Time-driven simulation.

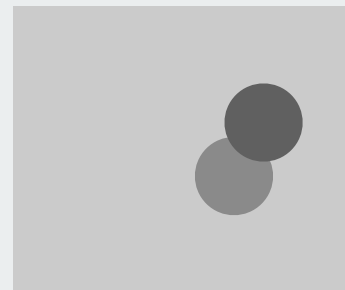
- Discretize time in quanta of size dt .
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.



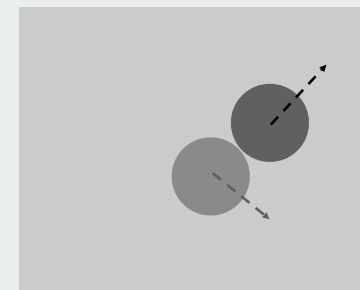
t



$t + dt$



$t + 2 dt$
(collision detected)

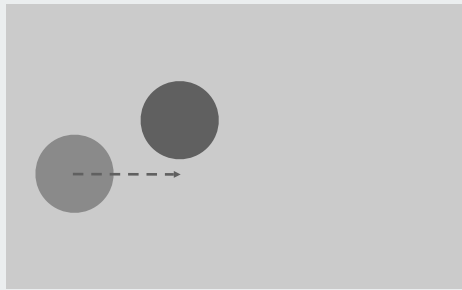


$t + \Delta t$
(roll back clock)

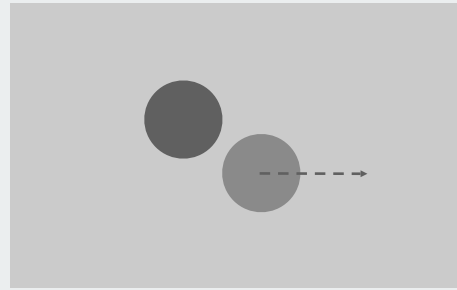
Time-driven simulation

Main drawbacks.

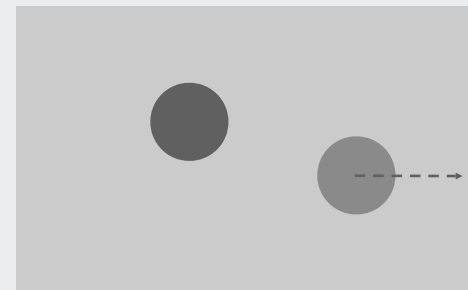
- N^2 overlap checks per time quantum.
- May miss collisions if dt is too large and colliding particles fail to overlap when we are looking.
- Simulation is too slow if dt is very small.



t



$t + dt$



$t + 2 dt$

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 **priority queue** of collision events, prioritized by time.
- Remove the minimum = 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.

Note: Same approach works for a broad variety of systems

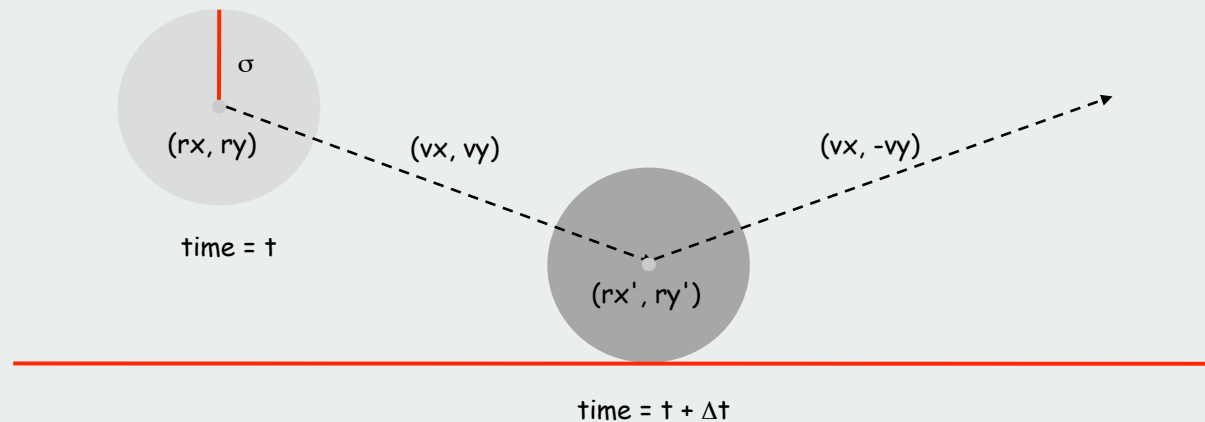
Particle-wall collision

Collision prediction.

- Particle of radius σ at position (rx, ry) .
- Particle moving in unit box with velocity (vx, vy) .
- Will it collide with a horizontal wall? If so, when?

$$\Delta t = \begin{cases} \infty & \text{if } vy = 0 \\ (\sigma - ry)/vy & \text{if } vy < 0 \\ (1 - \sigma - ry)/vy & \text{if } vy > 0 \end{cases}$$

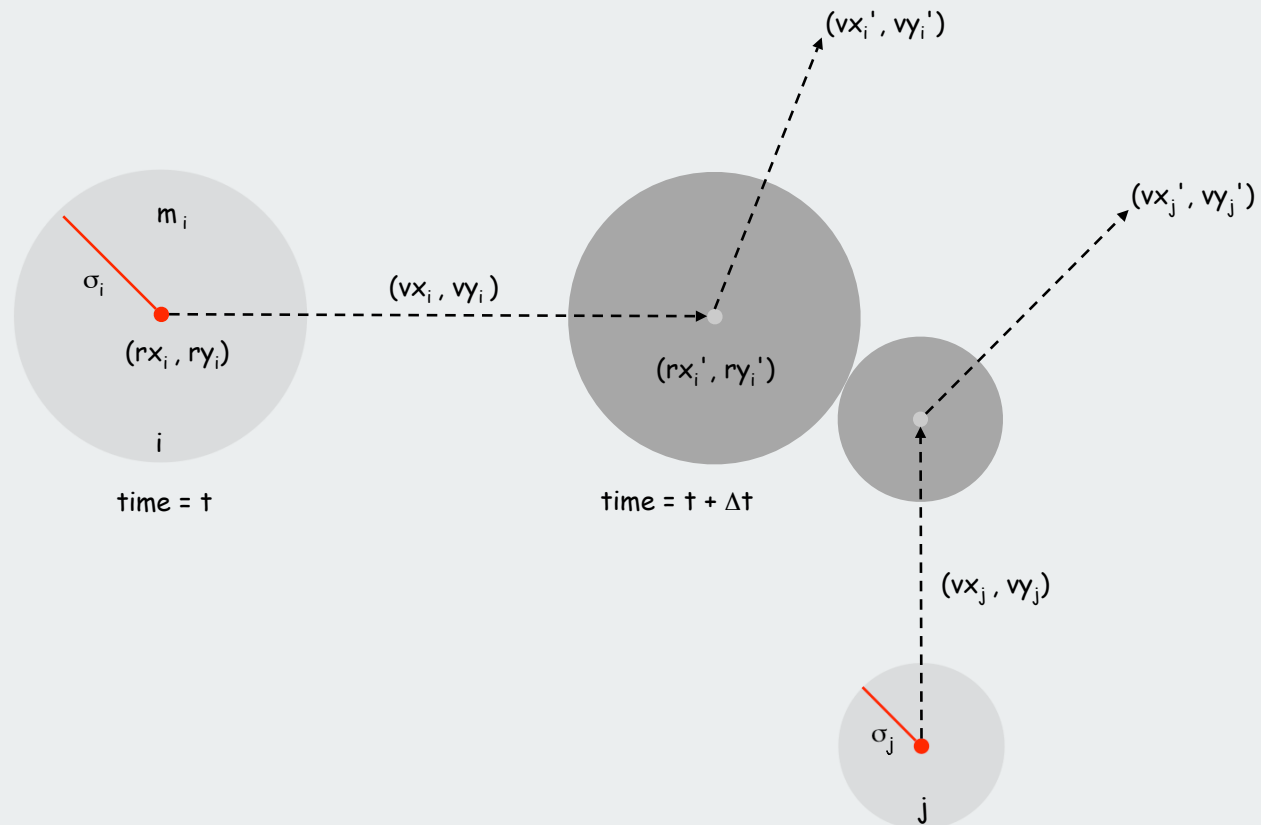
Collision resolution. $(vx', vy') = (vx, -vy)$.



Particle-particle collision prediction

Collision prediction.

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

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

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

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

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

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

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

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

Particle-particle collision prediction implementation

Particle has method to predict collision with another particle

```
public double dt(Particle b)
{
    Particle a = this;
    if (a == b) return INFINITY;
    double dx  = b.rx - a.rx;
    double dy  = b.ry - a.ry;
    double dvx = b.vx - a.vx;
    double dvy = b.vy - a.vy;
    double dvdr = dx*dvx + dy*dvy;
    if(dvdr > 0) return INFINITY;
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = a.radius + b.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY;
    return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

and methods `dtX()` and `dtY()` to predict collisions with walls

Particle-particle collision prediction implementation

CollisionSystem has method to predict all collisions

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

Particle-particle collision resolution

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

$$vx_i' = vx_i + Jx / m_i$$

$$vy_i' = vy_i + Jy / m_i$$

$$vx_j' = vx_j - Jx / m_j$$

$$vy_j' = vy_j - Jy / m_j$$

Newton's second law
(momentum form)

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

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

Particle-particle collision resolution implementation

Particle has method to resolve collision with another particle

```
public void bounce(Particle b)
{
    Particle a = this;
    double dx  = b.rx - a.rx;
    double dy  = b.ry - a.ry;
    double dvx = b.vx - a.vx;
    double dvy = b.vy - a.vy;
    double dvdr = dx*dvx + dy*dvy;
    double dist = a.radius + b.radius;
    double J = 2 * a.mass * b.mass * dvdr / ((a.mass + b.mass) * dist);
    double Jx = J * dx / dist;
    double Jy = J * dy / dist;
    a.vx += Jx / a.mass;
    a.vy += Jy / a.mass;
    b.vx -= Jx / b.mass;
    b.vy -= Jy / b.mass;
    a.count++;
    b.count++;
}
```

and methods `bounceX()` and `bounceY()` to resolve collisions with walls

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 is no longer valid, 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.

Collision system: main event-driven simulation loop implementation

```
public void simulate(double limit)
{
```

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

← initialize PQ with
collision events and
redraw event

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

← main event-driven
simulation loop

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

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

← update positions
and time

```
        if (a != null && b != null) a.bounce(b);
        else if (a != null && b == null) a.bounceX();
        else if (a == null && b != null) b.bounceY();
        else if (a == null && b == null)
        {
            StdDraw.clear(StdDraw.WHITE);
            for(int i = 0; i < N; i++) particles[i].draw();
            StdDraw.show(20);
            if (t < limit)
                pq.insert(new Event(t + 1.0 / Hz, null, null));
        }
```

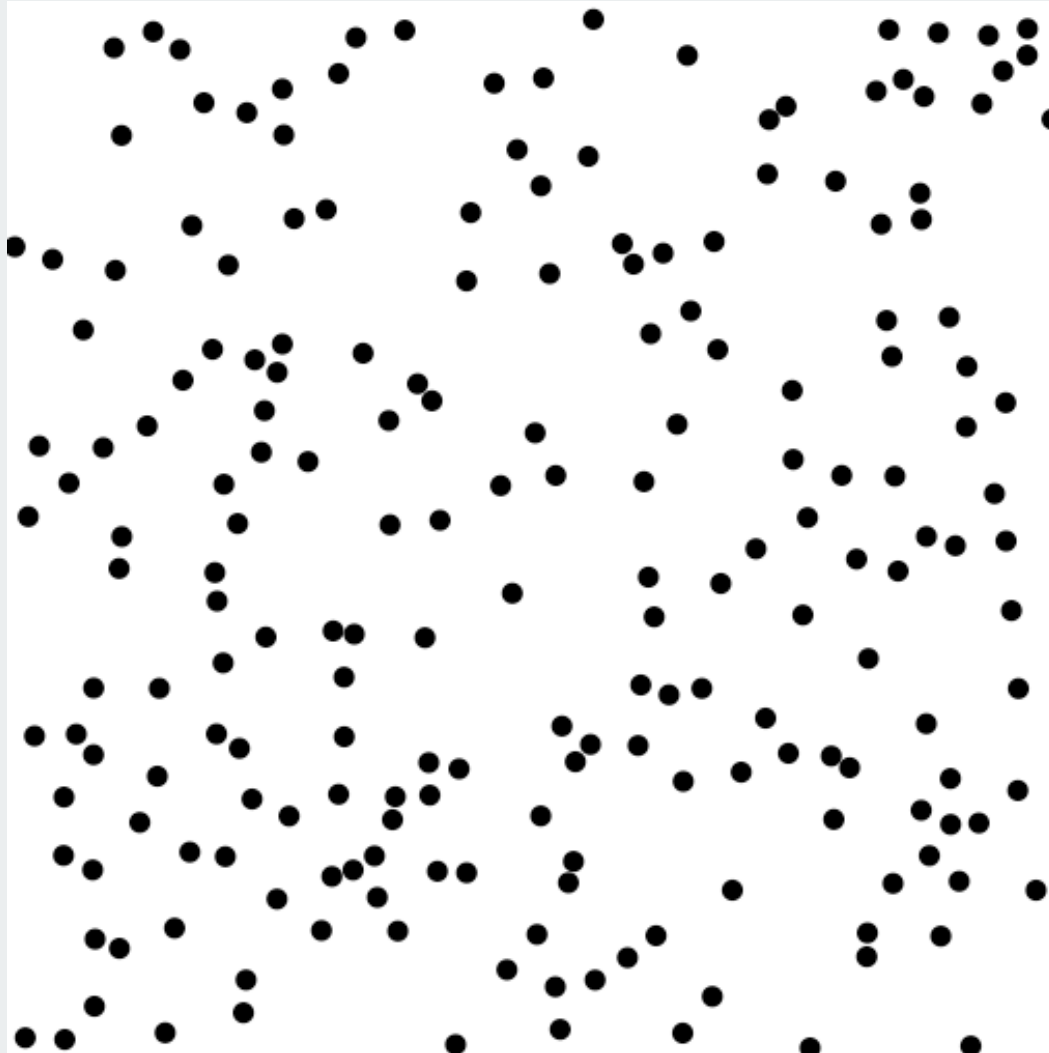
← process event

```
        predict(a, limit);
        predict(b, limit);
```

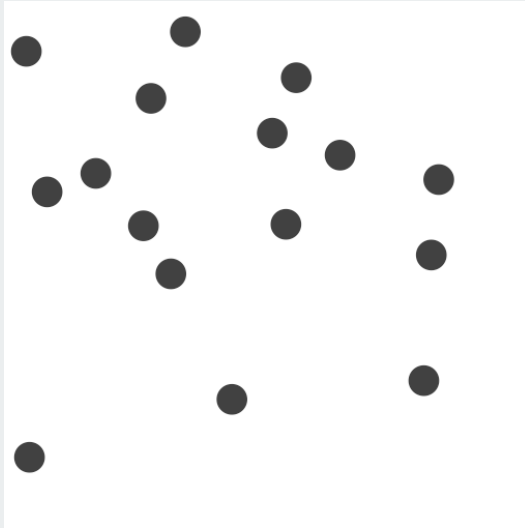
← predict new
events based on
changes

```
    }
```

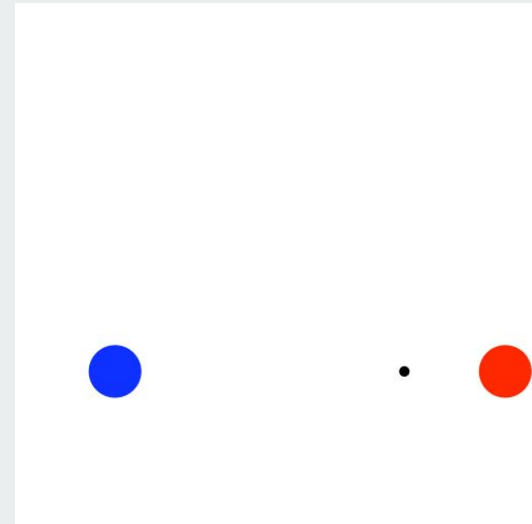
```
java CollisionSystem 200
```



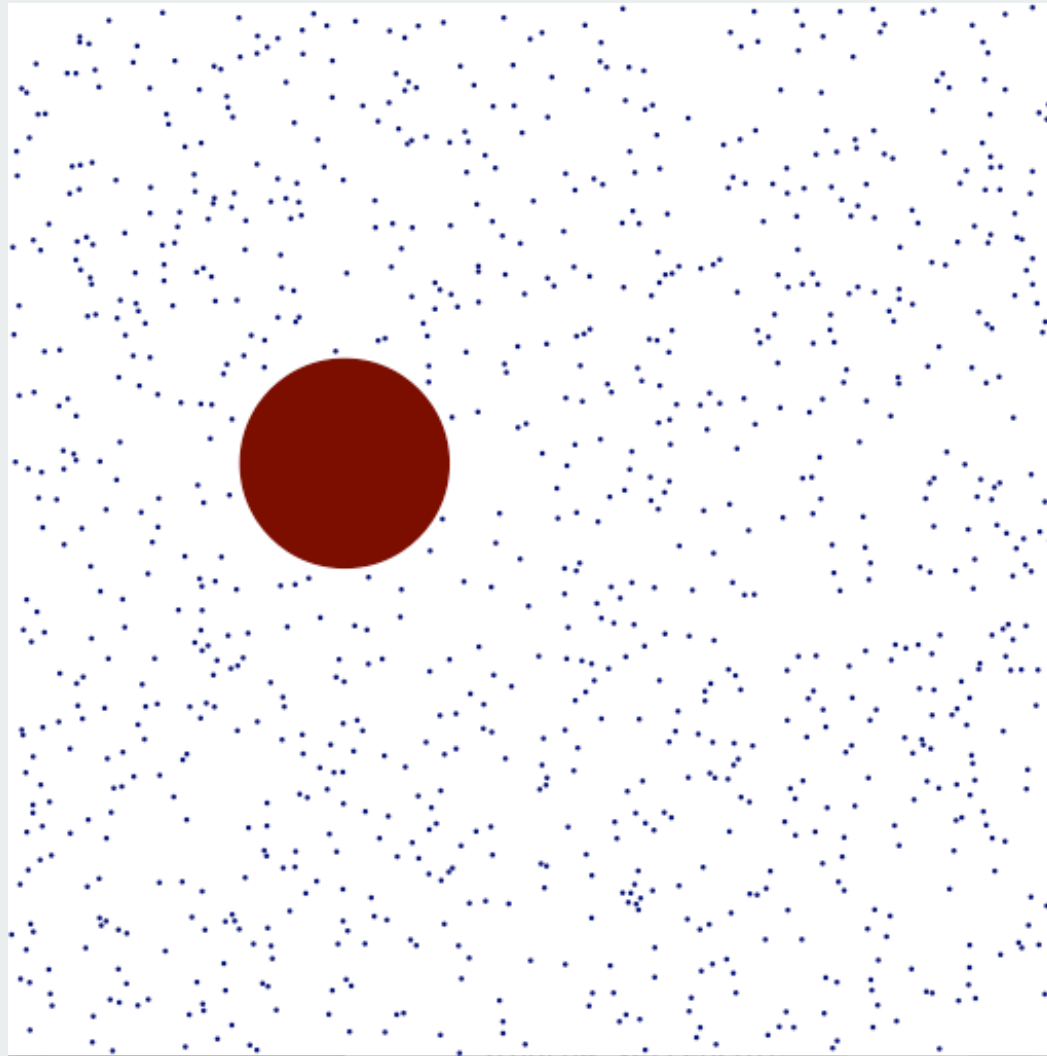
```
java CollisionSystem < billiards5.txt
```



```
java CollisionSystem < squeeze2.txt
```



```
java CollisionSystem < brownianmotion.txt
```



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

