Priority Queues

- API
- elementary implementations
- binary heaps
- heapsort
- event-based simulation

Priority queue API

Keys. Items that can be compared.

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

	`MaxPQ()`	*create an empty priority queue*
`boolean`	`isEmpty()`	*is the priority queue empty*
`void`	`insert(Key key)`	*insert a key*
`Key`	`delMax()`	*delete and return the maximum key*
`Key`	`max()`	*return the maximum key*
`int`	`size()`	*return the number of keys*

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

Priority queue 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 min-oriented priority queue.

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

cost of finding the largest M in a stream of N items

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

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

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

- API
- **elementary implementations**
- binary heaps
- heapsort
- event-based simulation

Priority queue: unordered and ordered array implementation

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

A sequence of operations on a priority queue

Priority queue: unordered array implementation

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

   public UnorderedPQ(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.

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

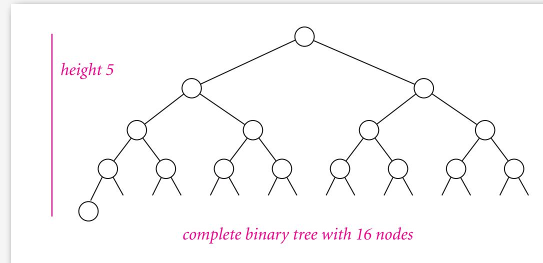
order-of-growth running time for PQ with N items

- API
- elementary implementations
- **binary heaps**
- heapsort
- event-based simulation

Binary tree

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

Complete tree. Balanced except for bottom level.



Property. Height of binary heap with N nodes is $1 + \lfloor \lg N \rfloor$.

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

Binary heap

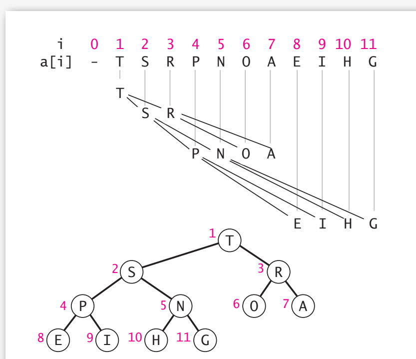
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

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

Array representation.

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

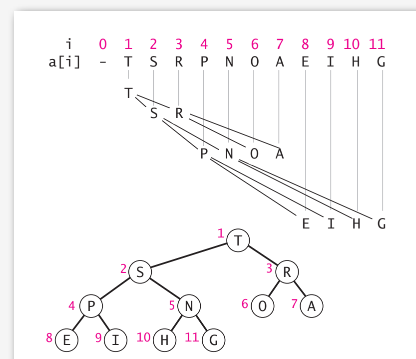


Binary heap properties

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.



Promotion in a heap

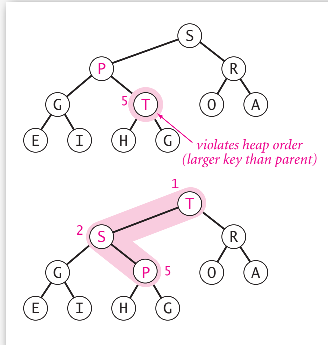
Scenario. Exactly one node has a larger key than its parent.

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

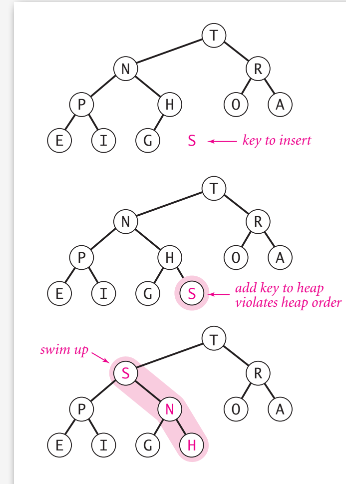


Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then promote.

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



Demotion in a heap

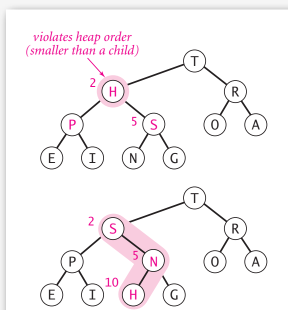
Scenario. Exactly one node has a smaller key than does a child.

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



Top-down heapify (sink)

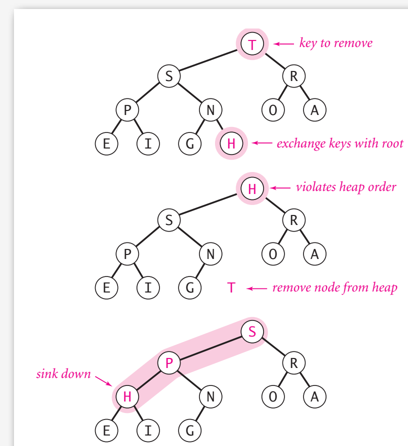
Power struggle. Better subordinate promoted.

Delete the maximum in a heap

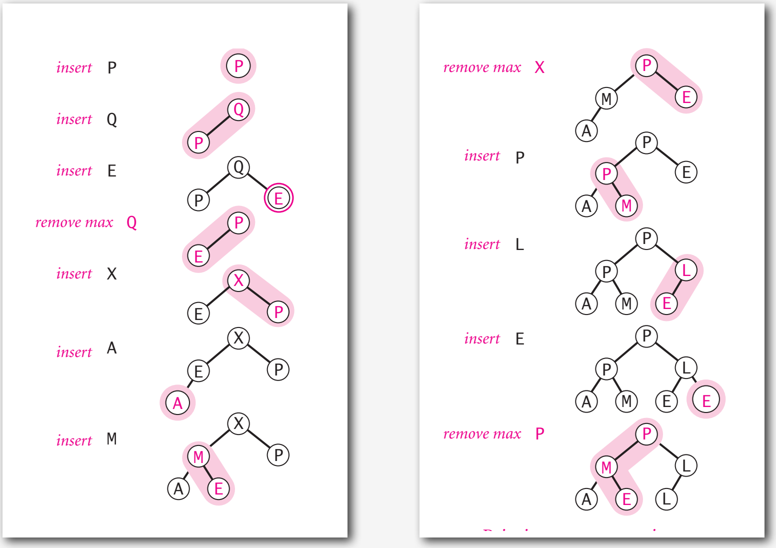
Delete max. Exchange root with node at end, then demote.

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

prevent loitering



Heap operations



Binary heap: Java implementation

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

   public MaxPQ(int capacity)
   {   ...   }

   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;   }
}
```

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()`.

Dynamic array resizing.
- Add no-arg constructor.
- Apply repeated doubling and shrinking. ← 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

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

order-of-growth running time for PQ with N items

Hopeless challenge. Make all operations constant time.

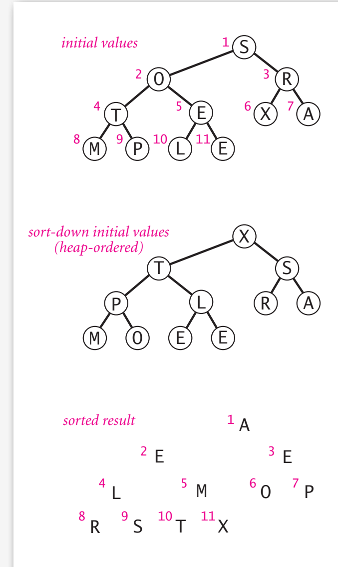
Q. Why hopeless?

- API
- elementary implementations
- binary heaps
- **heapsort**
- event-based simulation

Heapsort

Basic plan for in-place sort.

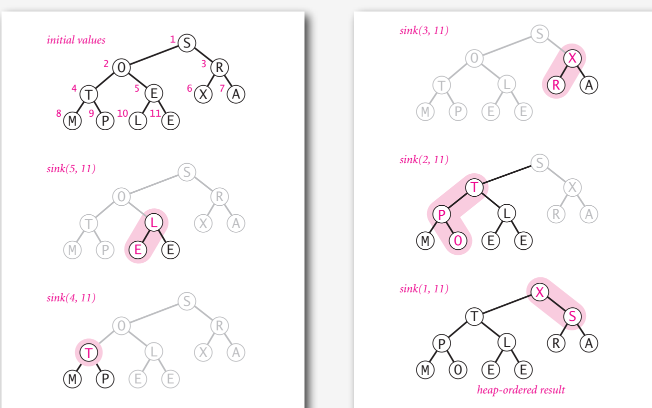
- Create max-heap with all N keys.
- Repeatedly remove the maximum key.



Heapsort

First pass. Build heap using bottom-up method.

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

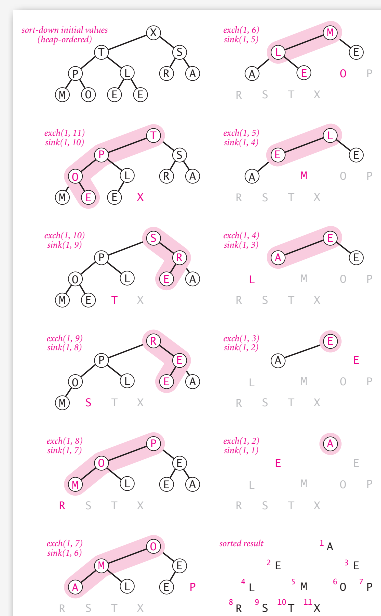


Heapsort

Second pass. Sort.

- 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

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

entries in black are compared

entries in red are exchanged

entries in gray are not touched

Heapsort trace (array contents after each sink)

Heapsort: mathematical analysis

Property D. At most 2 N lg N compares.

Significance. Sort in N log N worst-case without using extra memory.

- 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.

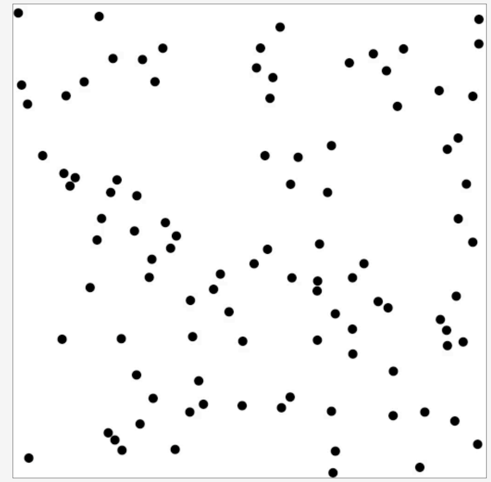
Sorting algorithms: summary

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

- API
- elementary implementations
- binary heaps
- heapsort
- **event-based 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 are exerted.

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.

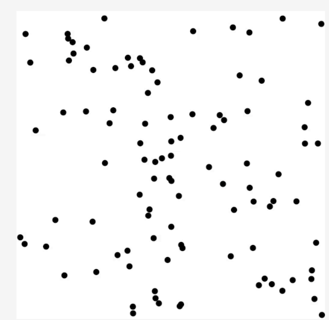
Warmup: bouncing balls

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);  }
}
```

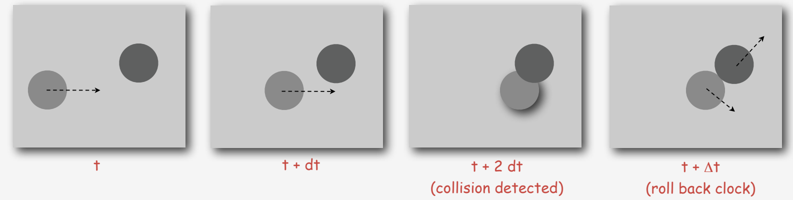
check for collision with walls

Missing. Check for balls colliding with each other.

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

Time-driven simulation

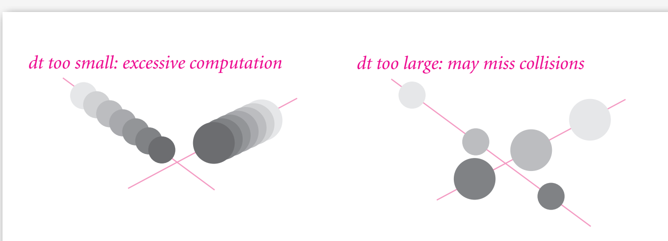
- Discretize time in quanta of size dt.
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.



Time-driven simulation

Main drawbacks.

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



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.

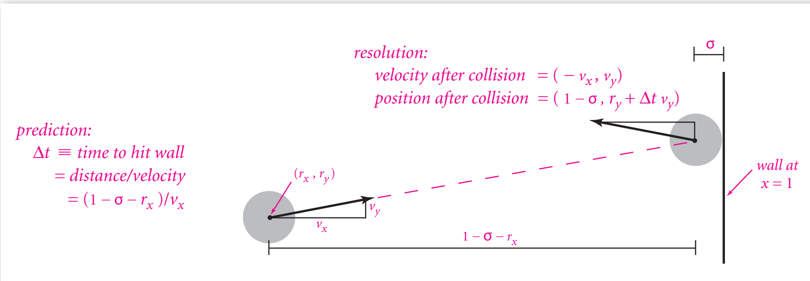
Collision prediction. Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

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

Particle-wall collision

Collision prediction and resolution.

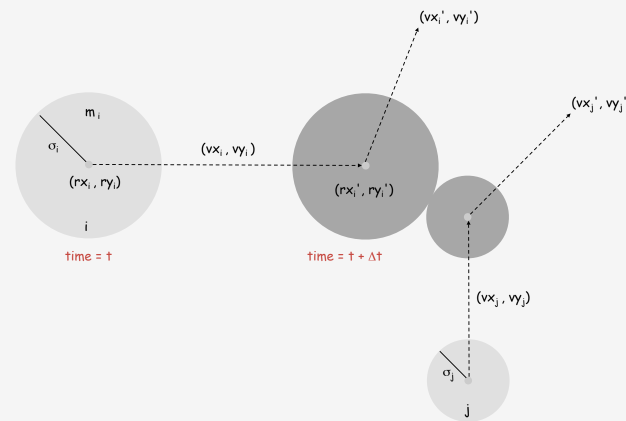
- Particle of radius σ at position (rx, ry).
- Particle moving in unit box with velocity (vx, vy).
- Will it collide with a vertical wall? If so, when?



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?



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?

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

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

$$\Delta v = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j)$$
$$\Delta r = (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j)$$

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

Particle-particle collision resolution

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

$$vx_i' = vx_i + Jx / m_i$$
$$vy_i' = vy_i + Jy / m_i$$
$$vx_j' = vx_j - Jx / m_j$$
$$vy_j' = vx_j - Jy / m_j$$

← 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)

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 dt(Particle that) { }
    public double dtX() { }
    public double dtY() { }

    public void bounce(Particle that) { }
    public void bounceX() { }
    public void bounceY() { }

}
```

- `dt`, `dtX`, `dtY`: predict collision with particle or wall
- `bounce`, `bounceX`, `bounceY`: resolve collision with particle or wall

Particle-particle collision and resolution implementation

```
public double dt(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;
   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;
}
```

no collision

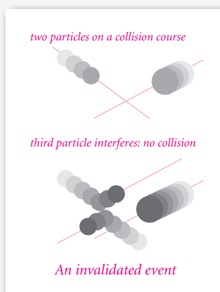
```
public void bounce(Particle that)
{
   double dx  = that.rx - this.rx, dy  = that.ry - this.ry;
   double dvx = that.vx - this.vx; dvy = that.vy - this.vy;
   double dvdr = dx*dvx + dy*dvy;
   double dist = this.radius + that.radius;
   double J = 2 * this.mass * that.mass * dvdr / ((this.mass +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


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



An invalidated event

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` ⇒ particle-particle collision.
- One particle `null` ⇒ particle-wall collision.
- Both particles `null` ⇒ redraw event.

```
public 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 double time()     { return time; }
    public Particle a()      { return a;    }
    public Particle b()      { return b;    }

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

    public boolean isValid()
    {   }
}
```

- `Event(...)`: create event
- `time()`, `a()`, `b()`: accessor methods
- `compareTo`: ordered by time
- `isValid`: 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.dt(particles[i]);
          pq.insert(new Event(t + dt, a, particles[i]));
       }
       pq.insert(new Event(t + a.dtX(), a, null));
       pq.insert(new Event(t + a.dtY(), null, a));
 }

    private void redraw()  { }

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

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

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));

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

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

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

initialize PQ with collision events and redraw event

get next event

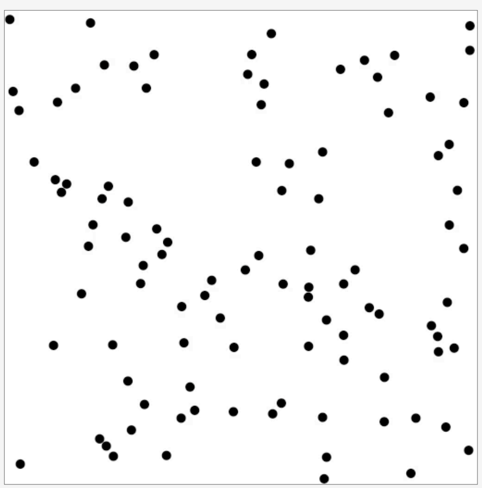
update positions and time

process event

predict new events based on changes

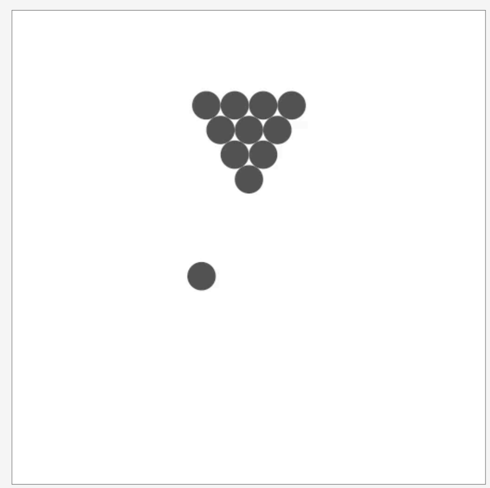
Simulation example 1

```
% java CollisionSystem 100
```



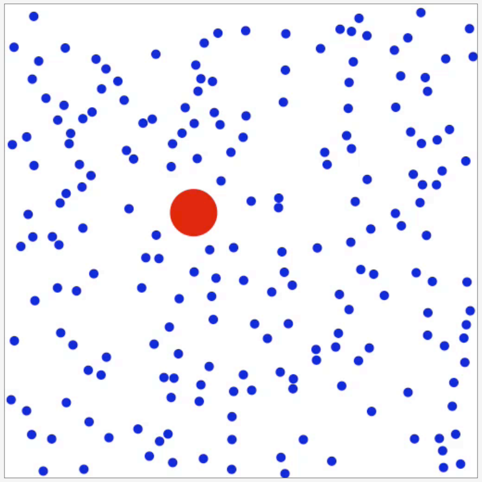
Simulation example 2

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



Simulation example 3

`% java CollisionSystem < brownian.txt`



Simulation example 4

`% java CollisionSystem < diffusion.txt`