2.4 Priority Queues

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

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.

Priority queue applications

- Event-driven simulation.
- Numerical computation.
- Data compression.
- Graph searching.
- Computational number theory.
- Artificial intelligence.
- Statistics.
- Operating systems.
- Discrete optimization.
- Spam filtering.

Generalizes: stack, queue, randomized queue.
Problem. Find the largest $M$ in a stream of $N$ elements ($N$ huge, $M$ large).

- 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.

Time. Proportional to $N \log M$ (stay tuned).

```java
public class TopM{
    public static void main(String[] args){
        // Print the top $M$ lines in the input stream.
        int M = Integer.parseInt(args[0]);
        MinPQ<Transaction> pq = new MinPQ<Transaction>(M+1);
        while (StdIn.hasNextLine()) {
            // Create an entry from the next line and put on the PQ.
            pq.insert(new Transaction(StdIn.readLine()));
            if (pq.size() > M) { pq.delMin(); } // Remove minimum if $M+1$ entries on the PQ.
        }
        // Top $M$ entries are on the PQ.
        // Smallest is first out---put on stack to get descending order.
        Stack<Transaction> stack = new Stack<Transaction>();
        while (!pq.isEmpty()) stack.push(pq.delMin());
        for (Transaction t : stack) StdOut.println(t);
    }
}
```

Priority queue client example

% more tinyBatch.txt
Turing 6/17/1990 644.08
vonNeumann 3/26/2002 4121.85
Dijkstra 11/18/2007 2678.40
Hoare 8/18/1992 4381.21
Turing 1/11/2002 66.10
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

% java TopM 5 < tinyBatch.txt
Turing 6/17/1990 644.08
vonNeumann 3/26/2002 4121.85
Dijkstra 11/18/2007 2678.40
vonNeumann 1/11/1999 4409.74
Hoare 8/18/1992 4381.21

Priority queue: unordered and ordered array implementation

A sequence of operations on a priority queue

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
<th>size</th>
<th>contents (unordered)</th>
<th>contents (ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>2</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>3</td>
<td>P E X</td>
<td>E P X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>4</td>
<td>P E X A</td>
<td>A E P X</td>
<td>A E P X</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>5</td>
<td>P E X A M</td>
<td>A E M P X</td>
<td>A E M P X</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>4</td>
<td>P E M A</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>5</td>
<td>P E M A P</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>6</td>
<td>P E M A P L</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>E</td>
<td>7</td>
<td>P E M A P L E</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>E M A P L E</td>
<td>A E L M P</td>
<td>A E L M P</td>
</tr>
</tbody>
</table>
Priority queue: unordered array implementation

```java
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];
    }
}
```

Priority queue elementary implementations

**Challenge.** Implement all operations efficiently.

**Order-of-growth of running time for priority queue with N items**

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Goal**

<table>
<thead>
<tr>
<th></th>
<th>log N</th>
<th>log N</th>
<th>log N</th>
</tr>
</thead>
</table>

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 \(1 + \lfloor \lg N \rfloor\).

**Pf.** Height only increases when \(N\) is a power of 2.
### 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.
- Parent of node at $k$ is at $k/2$.
- Children of node at $k$ are at $2k$ and $2k+1$.

```java
private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}
```

### 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.

```java
private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = 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 $\lg N$ compares.

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

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.

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

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.

```java
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;
    }
}
```

Heap operations

- **Insertion in a heap**: Add node at end, then swim it up.
- **Delete the maximum**: Exchange root with node at end, then sink it down.
- **Demotion**: Exchange key in node with key in larger child. Repeat until heap order restored.
### Binary heap: Java implementation

```java
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;
    }
}
```

### Priority queues implementation cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Del max</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>( O(1) )</td>
<td>( O(N) )</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>ordered array</td>
<td>( O(N) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( O(\log N) )</td>
<td>( O(\log N) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>( O(\log d) )</td>
<td>( O(\log d) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>( O(1) )</td>
<td>( O(\log N) )</td>
<td>( \dagger )</td>
</tr>
</tbody>
</table>

\( \dagger \) amortized

### Hopeless challenge

Make all operations constant time.

Q. Why hopeless?

### 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.
  
**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.
  
  lead to \( O(\log N) \) amortized time per op
Heapsort

Basic plan for in-place sort:
- Create max-heap with all \( N \) keys.
- Repeatedly remove the maximum key.

Heapsort: heap construction

First pass. Build heap using bottom-up method.

\[
\text{for} \ (\text{int } k = N/2; k >= 1; k--) \\
\quad \text{sink}(a, k, N); \\
\]

Heapsort: sortdown

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

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 i, 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

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[k]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>SORT L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORT L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>S O R T E X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S O R T E X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X S O R T E A M P L E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>T P S O L R A M E X</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>S P R O L E A M E T X</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>R P E O L E A M S T X</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>P O E M L E A R S T X</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>O M E A L E P R S T X</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>M E A L E O P R S T X</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>L E A M O P R S T X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A E E L M O P R S T X</td>
</tr>
</tbody>
</table>

*Heapsort trace (array contents just after each sink)*

### Heapsort: mathematical analysis

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

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

- Mergesort: no, linear extra space.
- Quicksort: no, quadratic time in worst case.
- 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


### Sorting algorithms: summary

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

**Significance.** Relates macroscopic observables to microscopic dynamics.
- Einstein: explain Brownian motion of pollen grains.

```java
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();
        }
    }
}
```

Warmup: bouncing balls

**Time-driven simulation.** \( N \) bouncing balls in the unit square.
**Warmup: bouncing balls**

```java
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);
    }
}
```

**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.

**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.

**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?

Collision prediction.
• Particle moving in unit box with velocity \((v_x, v_y)\).

Collision prediction and resolution.

Particle-wall collision resolution

\[
\Delta t = \begin{cases} 
\infty & \text{if } \Delta v \cdot \Delta r \leq 0 \\
\infty & \text{if } d > 0 \\
\frac{\Delta v \cdot \Delta r}{\Delta v \cdot \Delta r} & \text{otherwise} 
\end{cases}
\]

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

\[
\Delta v = (\Delta v_x, \Delta v_y) = (v_{x_f} - v_{x_i}, v_{y_f} - v_{y_i})
\]

\[
\Delta r = (\Delta r_x, \Delta r_y) = (r_{x_f} - r_{x_i}, r_{y_f} - r_{y_i})
\]

\[
\Delta v \cdot \Delta r = (\Delta v_x \Delta r_x + \Delta v_y \Delta r_y)
\]

Important note: This is high-school physics, so we won’t be testing you on it!

Particle-particle collision prediction

Collision prediction.
• Particle \(i\): radius \(s_i\), position \((r_{x_i}, r_{y_i})\), velocity \((v_{x_i}, v_{y_i})\).
• Particle \(j\): radius \(s_j\), position \((r_{x_j}, r_{y_j})\), velocity \((v_{x_j}, v_{y_j})\).
• Will particles \(i\) and \(j\) collide? If so, when?

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

\[
\begin{align*}
v_{x_i}' &= v_{x_i} + J_x/m_i \\
v_{y_i}' &= v_{y_i} + J_y/m_i \\
v_{x_j}' &= v_{x_j} - J_x/m_j \\
v_{y_j}' &= v_{y_j} - J_y/m_j
\end{align*}
\]

\[
J_x = \frac{J \Delta x}{\sigma}, \quad J_y = \frac{J \Delta y}{\sigma}, \quad J = \frac{2 m_i m_j (\Delta v \cdot \Delta r)}{\sigma (m_i + m_j)}
\]

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

Important note: This is high-school physics, so we won’t be testing you on it!
Particle data type skeleton

```java
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()   { }
}
```

Particle-particle collision and resolution implementation

```java
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;
    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++;
}
```

Collision system: event-driven simulation main loop

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

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.

```java
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() {   }
}
```
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 */
    }
}

Collision system implementation: skeleton

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.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();
            predict(a);
            predict(b);
        }
    }
}

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