2.4 Priority Queues

- API and elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation

Collections

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

<table>
<thead>
<tr>
<th>data type</th>
<th>core operations</th>
<th>data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>Push, Pop</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td>queue</td>
<td>Enqueue, Dequeue</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td>priority queue</td>
<td>insert, Delete-Max</td>
<td>binary heap</td>
</tr>
<tr>
<td>symbol table</td>
<td>Put, Get, Delete</td>
<td>binary search tree, hash table</td>
</tr>
<tr>
<td>set</td>
<td>Add, Contains, Delete</td>
<td>binary search tree, hash table</td>
</tr>
</tbody>
</table>

“Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won’t usually need your code; it’ll be obvious.” — Fred Brooks

Priority queue

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

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

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

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

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

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

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

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

**Priority queue: client example**

**Challenge.** Find the largest $M$ items in a stream of $N$ items.
- Fraud detection: isolate $SS$ transactions.
- NSA monitoring: flag most suspicious documents.

**Constraint.** Not enough memory to store $N$ items.

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

**Priority queue: applications**

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

**Priority queue: client example**

**Challenge.** Find the largest $M$ items in a stream of $N$ items.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>elementary PQ</td>
<td>$MN$</td>
<td>$M$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$N \log M$</td>
<td>$M$</td>
</tr>
<tr>
<td>best in theory</td>
<td>$N$</td>
<td>$M$</td>
</tr>
</tbody>
</table>

Order of growth of finding the largest $M$ in a stream of $N$ items.
**Priority queue: unordered and ordered array implementation**

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return</th>
<th>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>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
<td>P</td>
<td>P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>E</td>
<td>3</td>
<td>P Q E</td>
<td>E P</td>
<td>P</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>4</td>
<td>P E X</td>
<td>E P X</td>
<td>P</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>5</td>
<td>P E X A</td>
<td>A E P X</td>
<td>P</td>
<td>P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>6</td>
<td>P E X A M</td>
<td>A E M P X</td>
<td>P</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>7</td>
<td>P E X A M</td>
<td>A E M P X</td>
<td>P</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>8</td>
<td>P E X A M</td>
<td>A E M P X</td>
<td>P</td>
<td>P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>E</td>
<td>9</td>
<td>P E X A M</td>
<td>A E M P X</td>
<td>P</td>
<td>P Q</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue

**Priority queue: implementations cost summary**

**Challenge.** Implement all operations efficiently.

<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>

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

**Complete binary tree**

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

**Complete tree.** Perfectly balanced, except for bottom level.

![Complete binary tree with N = 16 nodes (height = 4)](image)

**Property.** Height of complete binary tree with N nodes is \([\log N]\).

**Pf.** Height increases only when \(N\) is a power of 2.
A complete binary tree in nature

Binary heap: representation

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

**Heap-ordered binary tree.**
- Keys in nodes.
- Parent's key no smaller than children’s keys.

**Array representation.**
- Indices start at 1.
- Take nodes in level order.
- No explicit links needed!

Binary heap: properties

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

**Proposition.** Can use array indices to move through tree.
- Parent of node at $k$ is at $k/2$.
- Children of node at $k$ are at $2k$ and $2k+1$.

Binary heap demo

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

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

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

Binary heap: promotion

Scenario. A key becomes larger than its parent’s key.

To eliminate the violation:
• Exchange key in child with key in parent.
• Repeat until heap order restored.

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

Peter principle. Node promoted to level of incompetence.

Binary heap: insertion

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

Cost. At most 1 + \lg N compares.

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

Binary heap: demotion

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

To eliminate the violation:
• Exchange key in parent with key in larger child.
• Repeat until heap order restored.

private void sink(int k) {
    while (2^k <= N) {
        int j = 2^k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}

Power struggle. Better subordinate promoted.
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;
    return max;
}

---

**Priority queue: implementations 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>$1$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>binary heap</td>
<td>log $N$</td>
<td>log $N$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

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

---

**DELETÉ-RANDOM FROM A BINARY HEAP**

**Goal.** Delete a random key from a binary heap in logarithmic time.
**Binary heap: practical improvements**

Do "half-exchanges" in sink and swim.
- Reduces number of array accesses.
- Worth doing.

Floyd’s "bounce" heuristic.
- Sink key at root all the way to bottom.  
- Swim key back up.  
- Overall, fewer compares; more exchanges.
- Worthwhile depending on cost of compare and exchange.

**Priority queues: quiz 1**

How many compares (in the worst case) to insert in a $d$-way heap?

A. $\sim \log_2 N$
B. $\sim \log_d N$
C. $\sim d \log_2 N$
D. $\sim d \log_d N$
E. I don’t know.
Priority queues: quiz 2

How many compares (in the worst case) to delete-max in a $d$-way heap?

A. $\approx \log_2 N$
B. $\approx \log_d N$
C. $\approx d \log_2 N$
D. $\approx d \log_d N$
E. I don’t know.

Priority queue: 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>1</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>binary heap</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>1</td>
</tr>
<tr>
<td>$d$-ary heap</td>
<td>$\log_d N$</td>
<td>$d \log_d N$</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>$\log N$†</td>
<td>1</td>
</tr>
<tr>
<td>Brodal queue</td>
<td>1</td>
<td>$\log N$†</td>
<td>1</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† amortized

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

sweet spot: $d = 4$

why impossible?

Binary heap: considerations

Underflow and overflow.
- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

Minimum-oriented priority queue.
- Replace less() with greater().
- Implement greater().

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

can implement efficiently with sink() and swim() [stay tuned for Prim/Dijkstra]

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

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

Disadvantage. Must create new object for each data type value.

“Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible.”
— Joshua Bloch (Java architect)

Priority queues: quiz 3

What is the name of this sorting algorithm?

A. Insertion sort.
B. Mergesort.
C. Quicksort.
D. None of the above.
E. I don’t know.

Priority queues: quiz 4

What are its properties?

A. \( N \log N \) compares in the worst case.
B. In-place.
C. Stable.
D. All of the above.
E. I don’t know.
Heapsort

Basic plan for in-place sort.
- View input array as a complete binary tree.
- Heap construction: build a max-heap with all N keys.
- Sortdown: repeatedly remove the maximum key.

keys in arbitrary order

build max heap (in place)

sorted result (in place)

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

array in sorted order

Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

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

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

Heapsort: Java implementation

```
public class Heap {
  public static void sort(Comparable[] a) {
    int N = a.length;
    for (int k = N/2; k >= 1; k--)
      sink(a, k, N);
    while (N > 1)
    {
      exch(a, 1, N);
      sink(a, 1, --N);
    }
  }
}
```

but make static (and pass arguments)

```
private static void sink(Comparable[] a, int k, int N) {
  /* as before */
}
```

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

```
private static boolean less(Comparable[] a, int i, int j) {
  /* as before */
}
```

```
private static void exch(Object[] a, int i, int j) {
  /* as before */
}
```

Heapsort: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>S O R T E X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>S O R T L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>S O X T L R A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S T X P L R A M O E E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X T S P L R A M O E E</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>T P S O L R A M E 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 L E A 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>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.** Heap construction makes \( \leq N \) exchanges and \( \leq 2N \) compares.

**Pf sketch.** [assume \( N = 2^{h+1} - 1 \)]

```
max number of exchanges
to sink node
```

```
binary heap of height \( h = 3 \)
```

\[
h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \ldots + 2^h(0) \leq 2^{h+1} - 1 = N
\]
Heapsort: mathematical analysis

**Proposition.** Heap construction uses \( \leq 2N \) compares and \( \leq N \) exchanges.

**Proposition.** Heapsort uses \( \leq 2N \log N \) compares and exchanges.

\[ \text{algorithm can be improved to } \sim N \log N \]

(but no such variant is known to be practical)

**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.
- Not stable.

\[ \text{can be improved using advanced caching tricks} \]

---

**Introsort**

**Goal.** As fast as quicksort in practice; \( N \log N \) worst case, in place.

**Introsort.**

- Run quicksort.
- Cutoff to heapsort if stack depth exceeds \( 2 \log N \).
- Cutoff to insertion sort for \( N = 16 \).

---

**Sorting algorithms: summary**

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td>( \frac{1}{2} N^2 )</td>
<td>( \frac{1}{2} N^2 )</td>
<td>( \frac{1}{2} N^2 )</td>
<td>( N ) exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔️</td>
<td>✔️</td>
<td>( N )</td>
<td>( \frac{1}{4} N^2 )</td>
<td>( \frac{1}{2} N^2 )</td>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td>✔️</td>
<td>( N \log N )</td>
<td>✔️</td>
<td>( N \log N )</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</td>
<td>✔️</td>
<td>( \frac{1}{2} N \log N )</td>
<td>( N \log N )</td>
<td>( N \log N )</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td>✔️</td>
<td>( N \log N )</td>
<td>✔️</td>
<td>( N \log N )</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td>✔️</td>
<td>( N \log N )</td>
<td>( 2N \ln N )</td>
<td>( \frac{1}{2} N^2 )</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔️</td>
<td>✔️</td>
<td>( 2N \ln N )</td>
<td>( N \log N )</td>
<td>( \frac{1}{2} N^2 )</td>
</tr>
<tr>
<td>heap</td>
<td>✔️</td>
<td>✔️</td>
<td>( N \log N )</td>
<td>( 2N \log N )</td>
<td>( 2N \log N )</td>
</tr>
<tr>
<td>?</td>
<td>✔️</td>
<td>✔️</td>
<td>( N \log N )</td>
<td>( N \log N )</td>
<td>✔️</td>
</tr>
</tbody>
</table>

---

In the wild. C++ STL, Microsoft .NET Framework.

---

2.4 **Priority Queues**

- API and elementary implementations
- binary heaps
- heap sort
- 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.

---

Warmup: bouncing balls

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

```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);
            StdDraw.show(50); // main simulation loop
        }
    }
}
```

---

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.

---

Warmup: bouncing balls

```java
public class BouncingBalls {
    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.

Event-driven simulation

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

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

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

Particle-wall collision

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

Prediction and resolving a particle-wall collision.
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?

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

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

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

Newton’s second law
(momentum form)

\[
\begin{align*}
\Delta v_x &= (v_{x_i} - r_{x_i}, v_{y_i} - r_{y_i}) \\
\Delta v_y &= (r_{x_i} - r_{x_j}, r_{y_i} - r_{y_j}) \\
\Delta v &= (\Delta v_x, \Delta v_y)
\end{align*}
\]

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) { } // predict collision with particle or wall
    public double timeToHitVerticalWall() { } // predict collision with vertical wall
    public double timeToHitHorizontalWall() { } // predict collision with horizontal wall

    public void bounceOff(Particle that) { } // resolve collision with particle or wall
    public void bounceOffVerticalWall() { } // resolve collision with vertical wall
    public void bounceOffHorizontalWall() { } // resolve collision with horizontal wall
}

http://algs4.cs.princeton.edu/61event/Particle.java.html
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 = i).
- If the event has been invalidated, ignore it.
- Advance all particles to time t, on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

Event data type

Conventions.
- Neither particle null \(\Rightarrow\) particle-particle collision.
- One particle null \(\Rightarrow\) particle-wall collision.
- Both particles null \(\Rightarrow\) redraw event.

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

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

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

  public boolean isValid()
  { ... }
}

Particle collision simulation: example 1

Particle collision simulation: example 2
Particle collision simulation: example 3

% java CollisionSystem < brownian.txt

Particle collision simulation: example 4

% java CollisionSystem < diffusion.txt