2.4 PRIORITY QUEUES

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

Priority queue

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

<table>
<thead>
<tr>
<th>data type</th>
<th>key 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>BST, hash table</td>
</tr>
<tr>
<td>set</td>
<td>ADD, CONTAINS, DELETE</td>
<td>BST, 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

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 API

Requirement. Generic items are Comparable.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxPQ()</td>
<td>create an empty priority queue</td>
</tr>
<tr>
<td>MaxPQ(Key[] a)</td>
<td>create a priority queue with given keys</td>
</tr>
<tr>
<td>void insert(Key v)</td>
<td>insert a key into the priority queue</td>
</tr>
<tr>
<td>Key delMax()</td>
<td>return and remove the largest key</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the priority queue empty?</td>
</tr>
<tr>
<td>Key max()</td>
<td>return the largest key</td>
</tr>
<tr>
<td>int size()</td>
<td>number of entries in the priority queue</td>
</tr>
</tbody>
</table>

Key must be Comparable (bounded type parameter)

Priority queue API

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]
- Number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [online median in data stream]
- Operating systems. [load balancing, interrupt handling]
- Computer networks. [web cache]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue client example

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

Constraint. Not enough memory to store \( N \) items.

```
% more tinyBatch.txt
Turing 6/17/1990 644.08
vonNeumann 3/26/2002 4212.85
Dijkstra 8/22/2007 2678.40
vonNeumann 1/11/1999 4409.74
Dijkstra 11/8/1995 837.42
Hoare 5/10/1993 3229.27
vonNeumann 2/12/1994 4732.35
Hoare 8/18/1992 4381.21
Turing 1/11/2002 66.10
Thompson 2/27/2000 4747.08
Turing 2/11/1991 2156.86
Hoare 8/12/2003 1025.70
vonNeumann 10/13/1993 2520.97
Dijkstra 9/10/2000 708.95
Turing 10/12/1993 3532.36
Hoare 2/10/2005 4050.20
```

```
% java TopM S < tinyBatch.txt
Thompson 2/27/2000 4747.08
vonNeumann 2/12/1994 4732.35
vonNeumann 1/11/1999 4409.74
Hoare 8/18/1992 4381.21
```

Transaction data type is Comparable (ordered by \$\$)

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

Transaction contains largest \( M \) items
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>$M N$</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 elementary implementations

**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>

Goal: $\log N$ $\log N$ $\log N$

Order of growth of running time for priority queue with $N$ items
2.4 Priority Queues

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

Complete 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 $\lceil \log N \rceil$.

Proof. Height increases only when $N$ is a power of 2.

A complete binary tree in nature

Binary heap representations

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

Heap-ordered binary tree.
- Keys in nodes.
- 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.

Heap ordered

Promotion in a heap

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

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

**Peter principle.** Node promoted to level of incompetence.
**Insertion in a heap**

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

Cost. At most 1 + lg N compares.

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

**Demotion in a heap**

Scenario. Parent's 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.

```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;
    }
}
```

Power struggle. Better subordinate promoted.

**Delete the maximum in a heap**

Delete max. Exchange root with node at end, then sink it down.

Cost. At most 2 lg N compares.

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

**Binary heap: Java implementation**

Public class MaxPQ<Key extends Comparable<Key>>

```java
public MaxPQ(int capacity) {
    pq = new Comparable[capacity+1];
}
```

PQ ops

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

heap helper functions

```java
public void insert(Key key) {
    /* see previous code */
}
```

```java
public void delMax() {
    /* see previous code */
}
```

```java
private boolean less(int i, int j) {
    return pq[i].compareTo(pq[j]) < 0;
}
```

```java
private void exch(int i, int j) {
    Key t = pq[i];
    pq[i] = pq[j];
    pq[j] = t;
}
```

array helper functions

```java
private void swim(int k) {
    /* see previous code */
}
```

```java
private void sink(int k) {
    /* see previous code */
}
```

```java
public boolean delMin() {
    /* see previous code */
}
```
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>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

Binary heap: practical improvements

Floyd’s sink-to-root trick.
- Sink key at root all the way to bottom. ← 1 compare per node
- Swim key back up. ← some extra compares and exchanges
- Fewer compares; more exchanges.
- Worthwhile depending on cost of compare and exchange.

Binary heap: practical improvements

Half-exchanges in sink and swim.
- Reduces number of array accesses.
- Worth doing.

Multiway heaps.
- Complete d-way tree.
- Parent’s key no smaller than its children’s keys.
- Swim takes \( \log_d N \) compares; sink takes \( d \log_d N \) compares.
- Sweet spot: \( d = 4 \).
Binary heap: practical improvements

Caching. Binary heap is not cache friendly.

![Diagram of a binary heap]

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>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

Binary heap: practical improvements

Caching. Binary heap is not cache friendly.

- Cache-aligned d–heap.
- Funnel heap.
- B–heap.
- ...

Figure 5: The layout of a

Figure 6: Graph of

Priority queues implementation cost summary

- 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.

- 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.

```java
public final class Vector {
    private final int N;
    private final double[] data;
    public Vector(double[] data) {
        this.N = data.length;
        this.data = new double[N];
        for (int i = 0; i < N; i++)
            this.data[i] = data[i];
    }
    ...
}
```

Immutable. String, Integer, Double, Color, Vector, Transaction, Point2D.
Mutable. StringBuilder, Stack, Counter, Java array.

Immutability: properties

Data type. Set of values and operations on those values.

Immutable data type. Can’t change the data type value once created.

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

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

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

Sorting with a binary heap

Q. What is this sorting algorithm?

```java
public void sort(String[] a) {
    int N = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < N; i++)
        pq.insert(a[i]);
    for (int i = N-1; i >= 0; i--)
        a[i] = pq.deMax();
}
```

Q. What are its properties?
A. $N \log N$, extra array of length $N$, not stable.

Heapsort intuition. A heap is an array; do sort in place.
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.

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

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: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>S O R T E X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>S O R T L X A M P E 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></td>
<td>10</td>
<td>T P S O L R A M E E X</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>S P R O L E A M E T X</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>R P E O L E A M S T X</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>P O E M L E A R S T X</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>O M E A L E P R S T X</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>M L E A E O P R S T X</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>L E E A M O P R S T X</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>A E E L M O P R S T X</td>
</tr>
</tbody>
</table>

initial values

heap-ordered

sorted result

A E E L M O P R S T X

Heapsort trace (array contents just after each sink)

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);
            }
    }

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

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

    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: mathematical analysis

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

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

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

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

**Proposition.** Heapsort uses \( \leq 2N \lg N \) 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.
- Not stable.

---

### Sorting algorithms: summary

<table>
<thead>
<tr>
<th></th>
<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></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>
<td>use for small ( N ) or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔</td>
<td></td>
<td>( N \log_2 N )</td>
<td>?</td>
<td>( c N^{3/2} )</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td></td>
<td>( \frac{1}{2} N \lg N )</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N \log N ) guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔</td>
<td>✔</td>
<td>( N )</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td>✔</td>
<td>( N \lg N )</td>
<td>( 2N \ln N )</td>
<td>( \frac{1}{2} N^2 )</td>
<td>( N \log N ) probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔ ✔</td>
<td>✔</td>
<td>( N )</td>
<td>( 2N \ln N )</td>
<td>( \frac{1}{2} N^2 )</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>heap</td>
<td>✔</td>
<td>✔</td>
<td>( N )</td>
<td>( 2N \lg N )</td>
<td>( 2N \lg N )</td>
<td>( N \log N ) guarantee; in-place</td>
</tr>
<tr>
<td>?</td>
<td>✔ ✔</td>
<td>✔</td>
<td>( N )</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>

---

**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 \( 2 \lg N \).
- Cutoff to insertion sort for \( N = 16 \).

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

---

2.4 **Priority Queues**

- API and elementary implementations.
- Binary heaps.
- Heapsort.
- Event-driven simulation.
Molecular dynamics simulation of hard discs

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

![Diagram of hard discs](image)

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

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

**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.

**Main drawbacks.**
- $\sim N^2/2$ overlap checks per time quantum.
- Simulation is too slow if $dt$ is very small.
- May miss collisions if $dt$ is too large.
  (if colliding particles fail to overlap when we are looking)

**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 $(rx, ry)$.
- 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**

<table>
<thead>
<tr>
<th>Prediction (at time $t$)</th>
<th>Resolution (at time $t + dt$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>particles hit unless one passes intersection point before the other arrives</td>
<td>velocities of both particles change after collision</td>
</tr>
<tr>
<td>$dr = \frac{distance}{velocity} = (1 - s - rx)vx$</td>
<td>$v_{x_{\text{after}}, y_{\text{after}}} = (1 - s, r_y + vy dt)$</td>
</tr>
<tr>
<td>$t + dr = t_{\text{wall}}$</td>
<td>$t_{\text{next collision}} = t + \Delta t$</td>
</tr>
</tbody>
</table>
### Particle-particle collision prediction

**Collision prediction.**
- Particle $i$: radius $s_i$, position $(rx_i, ry_i)$, velocity $(vx_i, vy_i)$.
- Particle $j$: radius $s_j$, position $(rx_j, ry_j)$, velocity $(vx_j, vy_j)$.
- Will particles $i$ and $j$ collide? If so, when?

#### When two particles collide

- Impulse due to normal force:
  \[ J_x = \frac{J \Delta rx}{\sigma}, \quad J_y = \frac{J \Delta ry}{\sigma}, \quad J = \frac{2m_i m_j (\Delta v \cdot \Delta s)}{\sigma (m_i + m_j)} \]

- Conservation of energy, conservation of momentum:
  \[ \Delta v = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j) \]
  \[ \Delta r = (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j) \]
  \[ \Delta v \cdot \Delta r = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry) \]

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

### Particle-particle collision resolution

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

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

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

**Important note:** This is 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() { }
}
```

**Important note:** This is physics, so we won’t be testing you on it!
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 dvr = dx*dvr + dy*dvy;
    if (dvr > 0) return INFINITY;
    double dvx = dx*v + dy*v; dvx = sqrt(dx*dx + dy*dy);
    double d = sqrt(dx*dx + dy*dy);
    if (d > 0) return INFINITY;
    return -(dvx + Math.sqrt(dx*dx + dy*dy)) / dvx;
}
```

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

Collision system implementation: skeleton

```java
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) {
        // add to PQ all particle-wall and particle-particle collisions involving this particle
        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 */ }
    }
```

- Important note: This is physics, so we won't be testing you on it!
Collision system implementation: main event-driven simulation loop

```java
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.bounceOffVerticalWall();
        else if (a == null && b == null) redraw();

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

Particle collision simulation example 1

Particle collision simulation example 2

Particle collision simulation example 3
Particle collision simulation example 4

% java CollisionSystem < diffusion.txt