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.

```
operation  argument  return value
insert     P           Q
remove max X M         X
insert     P E         P
insert     Q E         E
insert     X M         M
insert     X P         X
insert     X E         E
remove max
```

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.

```
operation  argument  return value
insert     P           Q
remove max X M         X
insert     P E         E
insert     Q E         E
insert     X M         M
insert     X P         P
insert     X E         Q
remove max
```
**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.
- NSA detection: isolate $\$$ 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.deleteMin();
        pq now contains largest \( m \) items
    }
}
```
Priority queue: unordered and ordered array implementation

<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></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P Q</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>3</td>
<td>P Q E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>2</td>
<td>P E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>3</td>
<td>P E X</td>
<td>E</td>
<td></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></td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>4</td>
<td>P E M A</td>
<td>A E M P X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>5</td>
<td>P E M A P</td>
<td>A E M P P</td>
<td></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></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>7</td>
<td>P E M A P L E</td>
<td>A E E L M P P</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>E M A P L E</td>
<td>A E E L M P</td>
<td></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>

goal

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

Solution. Partially-ordered array.

2.4 Priority Queues

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 binary tree with n nodes is \( \lceil \log n \rceil \).

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.

heap ordered
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

S R O N P G A E I H

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.

```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.

Binary heap: insertion

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

Cost. At most 1 + \( \log n \) compares.

```java
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.

```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.
**Binary heap: delete the maximum**

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

**Cost.** At most $2 \log n$ compares.

```java
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

**Binary heap: Java implementation**

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

    public MaxPQ(int capacity) {
        pq = 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 i) {
        // 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;
    }
}
```

**DELETE-RANDOM FROM A BINARY HEAP**

**Goal.** Delete a random key from a binary heap in logarithmic time.
**DETERMINE FROM A BINARY HEAP**

**Goal.** Delete a random key from a binary heap in logarithmic time.

**Solution.**
- Pick a random index \( r \) between 1 and \( n \).
- Perform \( \text{exch}(r, n--) \).
- Perform either \( \text{sink}(r) \) or \( \text{swim}(r) \).

**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.  \( \longrightarrow \) only 1 compare per node
- Swim key back up.  \( \leftarrow \) some extra compares and exchanges
- Overall, fewer compares; more exchanges.
Binary heap: practical improvements

Multiway heaps.
- Complete $d$-way tree.
- Parent’s key no smaller than its children’s keys.

Fact. Height of complete $d$-way tree on $n$ nodes is $\sim \log_d n$.

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. $\sim \log_2 n$
B. $\sim \log_d n$
C. $\sim d \log_2 n$
D. $\sim d \log_d n$
E. I don’t know.

Priority queue: implementation cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Delete 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: 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.

- **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 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 variables** are `private` and `final` (neither necessary nor sufficient, but good programming practice).
- **Defensive copy** of mutable instance variables.

---

**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.

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.

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 demo

Heap construction. Build max heap using bottom-up method.

array in arbitrary order
Heapsort demo

**Sortdown.** Repeatedly delete the largest remaining item.

array in sorted order

A
E
E
L
M
O
P
R
S
T
X

Heapsort: sortdown

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

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

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: Java implementation

```java
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)

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

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

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

but convert from 1-based indexing to 0-base indexing
Heapsort: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Initial values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SORTEXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>SORTLXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORTLXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>SXTLXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>SXTPLXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>XTSPLXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sorted result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEELMOPRSTX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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.

**Proof sketch.** [assume $n = 2^{h+1} - 1$]

$$h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \ldots + 2^h(0) = 2^{h+1} - h - 2 = N - (h - 1) \leq N$$

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 $n = 16$.

In the wild. C++ STL, Microsoft .NET Framework.
### 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>½ $n^2$</td>
<td>½ $n^2$</td>
<td>$n^2$</td>
<td>$n$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>¼ $n^2$</td>
<td>½ $n^2$</td>
<td>use for small $n$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔</td>
<td></td>
<td>$n \log n$</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td></td>
<td>½ $n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td></td>
<td>$n \log n$</td>
<td>2 $n \ln n$</td>
<td>½ $n^2$</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>2 $n \ln n$</td>
<td>½ $n^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>heap</td>
<td>✔</td>
<td>✔</td>
<td>3 $n$</td>
<td>2 $n \log n$</td>
<td>2 $n \log n$</td>
<td>$n \log n$ guarantee; in-place</td>
</tr>
<tr>
<td>?</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</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.

![Simulation of hard discs](image)

**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.

**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)

\( dt \) too small: excessive computation
\( dt \) too large: may miss collisions
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?

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?

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?

\[
\begin{align*}
\mathbf{v}_i' &= \mathbf{v}_i + \frac{J_x}{m_i} \\
\mathbf{v}_j' &= \mathbf{v}_j + \frac{J_y}{m_j} \\
\mathbf{v}_j - \frac{J_x}{m_j} \\
\mathbf{v}_j - \frac{J_y}{m_j}
\end{align*}
\]

Newton's second law

(momentum form)

**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) { } predict collision
    public double timeToHitVerticalWall() { } with particle or wall
    public double timeToHitHorizontalWall() { }

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

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.

“potential” since collision is invalidated if some other collision intervenes

**Main loop.**
- Delete the impending event from PQ (min priority = i).
- If the event has been invalidated, ignore it.
- Advance all particles to time i, 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.
Particle collision simulation: example 1

% java CollisionSystem 100

Particle collision simulation: example 2

% java CollisionSystem < billiards.txt

Particle collision simulation: example 3

% java CollisionSystem < brownian.txt

Particle collision simulation: example 4

% java CollisionSystem < diffusion.txt