2.4 PRIORITY QUEUES

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation
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><strong>Push, Pop</strong></td>
<td><em>linked list, resizing array</em></td>
</tr>
<tr>
<td>queue</td>
<td><strong>ENQUEUE, DEQUEUE</strong></td>
<td><em>linked list, resizing array</em></td>
</tr>
<tr>
<td>priority queue</td>
<td><strong>INSERT, DELETE-MAX</strong></td>
<td><em>binary heap</em></td>
</tr>
<tr>
<td>symbol table</td>
<td><strong>PUT, GET, DELETE</strong></td>
<td><em>binary search tree, hash table</em></td>
</tr>
<tr>
<td>set</td>
<td><strong>ADD, CONTAINS, DELETE</strong></td>
<td><em>binary search tree, hash table</em></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.

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>P</td>
</tr>
</tbody>
</table>
Priority queue API

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

<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 a largest key</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the priority queue empty?</td>
</tr>
<tr>
<td>Key max()</td>
<td>return a largest key</td>
</tr>
<tr>
<td>int size()</td>
<td>number of entries in the priority queue</td>
</tr>
</tbody>
</table>

Note. Duplicate keys allowed; `delMax()` picks any maximum key.
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.
- Fraud 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.delMin();
    }
}
```

Transaction data type is Comparable (ordered by $$)

use a min-oriented pq

pq now contains largest m items

$n$ huge, $m$ large
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>$m \cdot n$</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
A sequence of operations on a priority queue
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>
<tr>
<td>goal</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

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

Solution. Partially-ordered array.
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 binary tree with \( n \) nodes is \([\lg n]\).

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

Hyphaene Compressa - Doum Palm

© Shlomit Pinter
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

```
T
/  \
P   R
|   |
N   H
|   |
E   I
|   |
G
```

```
T P R N H O A E I G
```
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.

```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 $1 + \lg n$ 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;
}
```
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 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><strong>binary heap</strong></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**
Goal. Delete a random key from a binary heap in logarithmic time.
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 $\text{swim}(r)$.
**DELETE-RANDOM 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 $\text{swim}(r)$.
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.  ⟷ only 1 compare per node
- Swim key back up. ⟷ some extra compares and exchanges
- Overall, fewer compares; more exchanges.

R. W. Floyd
1978 Turing award
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>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>$\log n \dagger$</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.

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

leads to log n amortized time per op (how to make worst case?)

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

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)
2.4 Priority Queues

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation
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.
What are its properties?

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.delMax();
}

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

we assume array entries are indexed 1 to n

array in arbitrary order
Heapsort demo

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

array in sorted order

<table>
<thead>
<tr>
<th>A</th>
<th>E</th>
<th>E</th>
<th>L</th>
<th>M</th>
<th>O</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

starting point (arbitrary order)
sink(5, 11)
sink(4, 11)
sink(3, 11)
sink(1, 11)

result (heap-ordered)
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);
}
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 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 */ }

but make static (and pass arguments)

but convert from 1-based indexing to 0-base indexing
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 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>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 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 $n = 2^{h+1} - 1$]

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

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

**Proposition.** Heapsort uses \( \leq 2n \lg n \) compares and exchanges.

algorithm can be improved to \( \sim n \lg n \)
(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.

in-place merge possible, not practical
\( n \log n \) worst-case quicksort possible, not practical

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 $n = 16$.

In the wild. C++ STL, Microsoft .NET Framework.
## 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>$n$</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>$n \log_3 n$</td>
<td>?</td>
<td>$c n^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td>$\frac{1}{2} n \log n$</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>$n$</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>$n \log n$</td>
<td>$2 n \ln n$</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>$n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>heap</td>
<td>✔</td>
<td>$3 n$</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>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
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.
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);
        }
    }
}
```

% java BouncingBalls 100

main simulation loop
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);
    }
}
```

**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.
Time-driven 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 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?

\[
\text{prediction (at time } t \text{)} \\
\text{ } dt = \text{time to hit wall} \\
\text{ } = \text{distance/velocity} \\
\text{ } = \frac{(1-s-r_x)}{v_x}
\]

\[
\text{resolution (at time } t + dt \text{)} \\
\text{velocity after collision} = (-vx, vy) \\
\text{position after collision} = (1-s, ry+vydt)
\]

Finding and resolving a particle-wall collision
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?
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?

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

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

$$
\begin{align*}
\Delta v &= (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j) \\
\Delta r &= (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j) \\
\Delta v \cdot \Delta v &= (\Delta vx)^2 + (\Delta vy)^2 \\
\Delta r \cdot \Delta r &= (\Delta rx)^2 + (\Delta ry)^2 \\
\Delta v \cdot \Delta r &= (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)
\end{align*}
$$

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?

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

Newton's second law (momentum form)

\[
J_x = \frac{J \Delta r_x}{s}, \quad J_y = \frac{J \Delta r_y}{s}, \quad J = \frac{2 m_i m_j (\Delta v \cdot \Delta r)}{s (m_i + m_j)}
\]

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

Important note: This is physics, so we won’t be testing you on it!
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() { }  
    public double timeToHitHorizontalWall() { }  

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

declared in

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()
    {
        return true;  
    }
}

create event

ordered by time

valid if no intervening collisions (compare collision counts)
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
% javaCollisionSystem < diffusion.txt
```