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>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>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.

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

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

```
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 the largest key
- `boolean isEmpty()` is the priority queue empty?
- `Key max()` return the largest key
- `int size()` number of entries in the priority queue

**Priority queue: applications**

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

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

**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.

```
% more transactions.txt
Turing 6/17/1990 644.08
vonNeumann 3/26/2002 4121.85
Dijkstra 8/22/2007 2678.40
vonNeumann 1/11/1999 4409.74
Dijkstra 11/18/1995 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
vonNeumann 2/12/1994 4732.35
vonNeumann 1/11/1999 4409.74
Hoare 8/18/1992 4381.21
vonNeumann 3/26/2002 4121.85
```

```
% java TopM S < transactions.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
Turing 1/11/2002 66.10
Thompson 2/27/2000 4747.08
Turing 2/11/1991 2556.86
Hoare 8/12/2003 1023.70
vonNeumann 10/13/1993 2520.97
Dijkstra 9/10/2000 708.95
Turing 10/12/1993 3532.36
Hoare 2/10/2005 4050.20
```

**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.

```
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 data type is Comparable (ordered by $SS$)
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>$N \log M$</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>size</th>
<th>contents (unordered)</th>
<th>contents (ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td>P</td>
<td>1</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>P</td>
<td>2</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>P</td>
<td>3</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>P</td>
<td>2</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>P</td>
<td>3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>E</td>
<td>4</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>P</td>
<td>3</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>P</td>
<td>5</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>E</td>
<td>6</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>L</td>
<td>7</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>M</td>
<td>6</td>
<td>L</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>
<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

---

### 2.4 Priority Queues

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

http://algs4.cs.princeton.edu
**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 16 nodes (height = 4)](image)

**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.

---

**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!

![Heap representations](image)

---

**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 $2k+1$. 

![Heap representations](image)
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

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.
Remove the maximum. Exchange root with node at end, then sink it down.
**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*
**DELETE-RANDOM FROM A BINARY HEAP**

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

```
S  R  H  P  N  G  A  E  J  I
```

---

**Binary heap: practical improvements**

**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.
- Worthwhile depending on cost of compare and exchange.

R. W. Floyd
1978 Turing award

---

**Binary heap memory layout (page size = 8 nodes)**

**Caching.** Binary heap is not cache friendly.
**Binary heap: practical improvements**

**Caching.** Binary heap is not cache friendly.
- Cache-aligned $d$-heap.
- Funnel heap.
- B-heap.
- ...

B-heap memory layout (page size = 8 nodes)

---

**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)</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>

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

† amortized

---

**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 \(\text{less}()\) with \(\text{greater}()\).
- Implement \(\text{greater}()\).

**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)
2.4 PRIORITY QUEUES

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

---

Priority queues: quiz 2

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 3

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.

---
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

Heapsort demo

First pass. Build heap using bottom-up method.

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

Heapsort: heap construction

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: sortdown
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);
    }
  }

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

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>SORT</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORT</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>SOXTL</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>STXPL</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>XTSPLR</td>
</tr>
</tbody>
</table>

 initial values

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>TPSOLRAMEEX</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>SPROLEAMETX</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>RPEOLEAMESTX</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>POEMLEARSTX</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>OMEAEPRSTX</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>MLEAEOPRSTX</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>LEAMEOPRSTX</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>EAELOPRSTX</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>EAELMOPRSTX</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>AEELMOPRSTX</td>
</tr>
</tbody>
</table>

sorted result

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>AEELMOPRSTX</td>
</tr>
</tbody>
</table>

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

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

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

- **max number of exchanges to sink node**
- **binary heap of height \( h = 3 \)**
- **a tricky sum** (see COS 340)

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

Heapsort: mathematical analysis

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

**Proposition.** Heapsort uses \( \leq 2N \lg N \) 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.

- can be improved using advanced caching tricks

- in-place merge possible, not practical

- \( N \log N \) worst-case quicksort possible, not practical
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.

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>$c \cdot N^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</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>$N$</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>$N \lg N$</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>$N$</td>
<td>$2 N \lg N$</td>
<td>$2 N \lg N$</td>
<td>$N \log N$ guarantee; in-place</td>
</tr>
<tr>
<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>

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.

**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);
        }
    }
}
```

**Warmup: bouncing balls**

```java
public class Ball {
    private double rx, ry; // position
    private double vx, vy; // velocity
    private final double radius; // radius
    public Ball(...) {
        // initialize position and velocity */
    }
    public void move(double dt) {
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }
    public void draw() {
        StdDraw.filledCircle(rx, ry, radius); }
}
```

**Time-driven simulation**
- Discretize time in quanta of size $dt$.
- Update the position of each particle after every $dt$ units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

**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**

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)

![Diagram](image1)  
**dt too small: excessive computation**  
**dt too large: may miss collisions**

---

**Event-driven simulation**

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

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

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

---

**Particle-wall collision**

Collision prediction and resolution.
- Particle of radius \( s \) at position \((r_x, r_y)\).
- Particle moving in unit box with velocity \((v_x, v_y)\).
- Will it collide with a vertical wall? If so, when?

![Diagram](image2)  
**Prediction**  
\( \Delta t = \text{time to hit wall} = \text{distance/velocity} = (1 - s - r_x)v_x \)  
**Resolution**  
\( \text{velocity after collision} = (-v_x, v_y) \)  
\( \text{position after collision} = (1 - s - r_x + v_y \Delta t) \)

---

**Particle-particle collision prediction**

Collision prediction.
- Particle \( i \): radius \( s_i \), position \((r_{xi}, r_{yi})\), velocity \((v_{xi}, v_{yi})\).
- Particle \( j \): radius \( s_j \), position \((r_{xj}, r_{yj})\), velocity \((v_{xj}, v_{yj})\).
- Will particles \( i \) and \( j \) collide? If so, when?

![Diagram](image3)  
**Collision prediction**  
\( v_{xi}, v_{yi} \)  
\( v_{xj}, v_{yj} \)  
\( r_{xi}, r_{yi} \)  
\( r_{xj}, r_{yj} \)  
\( v_{xi}', v_{yi}' \)  
\( v_{xj}', v_{yj}' \)  
\( r_{xi}', r_{yi}' \)  
\( r_{xj}', r_{yj}' \)  
\( t \)  
\( t + \Delta t \)
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?

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

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

$\Delta v = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j)$
$\Delta r = (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j)$
$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$
$\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2$
$\Delta r \cdot \Delta v = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)$

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

Particle-particle collision and resolution implementation

```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) {
        if (this == that) return INFINITY;
        double dx = that.rx - this.rx, dy = that.ry - this.ry;
        double dvx = that.vx - this.vx, dvy = that.vy - this.vy;
        double ddvr = dx*dvx + dy*dvy;
        if (ddvr > 0) return INFINITY;
        double d = (dvx*dvx + dvy*dvy);
        double dr = dx + dy;
        double s = this.radius + that.radius;
        double d = (ddvr*dvr) - dvx*dvy * (dr*r) / (s*s);
        if (d < 0) return INFINITY;
        return -(dvx*dr + Math.sqrt(d))/dvy;
    }
    public void bounceOff(Particle that) {
        double dx = that.rx - this.rx, dy = that.ry - this.ry;
        double dvx = that.vx - this.vx, dvy = that.vy - this.vy;
        double dr = dx + dy;
        double s = this.radius + that.radius;
        double d = (dvx*dvx + dvy*dvy);
        double dx = d * dx / s;
        double dy = d * dy / s;
        this.vx = dx / this.mass;
        this.vy = dy / this.mass;
    }
}
```

Particle-particle collision resolution

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

$$v_i' = v_i + s_i \cdot \frac{J_x}{m_i}$$
$$v_j' = v_j + s_j \cdot \frac{J_y}{m_j}$$
$$J_x = \frac{J \cdot \Delta r_x}{s}, \ J_y = \frac{J \cdot \Delta r_y}{s}, \ J = \frac{2 m_i m_j (\Delta v \cdot \Delta r)}{s (m_i + m_j)}$$

Important note: This is physics, so we won’t be testing you on it!
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 may not happen if some other collision intervenes

**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.

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) {
    }

    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 */ }
}
```

Event data type

**Conventions.**
- Neither particle null ⇒ particle-particle collision.
- One particle null ⇒ particle-wall collision.
- Both particles null ⇒ redraw event.

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

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

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

    public boolean isValid() { ... }
}
```

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.bounceOffHorizontalWall();
        else if (a == null && b == null) redraw();

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