Priority Queues

- API
- elementary implementations
- binary heaps
- heapsort
- event-based simulation

Priority queue applications

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

Generalizes: stack, queue, randomized queue.

Priority queue API

Keys. Items that can be compared.

```
public class MaxPQ<Key extends Comparable<Key>>
MaxPQ() create an empty priority queue
boolean isEmpty() is the priority queue empty
void insert(Key key) insert a key
Key delMax() delete and return the maximum key
Key max() return the maximum key
int size() return the number of keys
```

Priority queue client example

Problem. Find the largest M of a stream of N elements.
- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N elements.

Solution. Use a min-oriented priority queue.

```
MinPQ<String> pq = new MinPQ<String>();
while (!StdIn.isEmpty())
{
   String s = StdIn.readString();
   pq.insert(s);
   if (pq.size() > M)
      pq.delMin();
}
while (!pq.isEmpty())
   System.out.println(pq.delMin());
```
API
• elementary implementations
• binary heaps
• heapsort
• event-based simulation

Priority queue: unordered and ordered array implementation

A sequence of operations on a priority queue

<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>P</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>P Q</td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>E P Q</td>
<td>3</td>
<td>E P Q</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>P E</td>
<td>2</td>
<td>P E</td>
<td>E P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>E P X</td>
<td>3</td>
<td>E P X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>A E P X</td>
<td>4</td>
<td>A E P X</td>
<td>A E P X</td>
</tr>
<tr>
<td>remove max</td>
<td>M</td>
<td>A E M P</td>
<td>5</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>A E M P P</td>
<td>5</td>
<td>A E M P P</td>
<td>A E M P P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>A E L M P P</td>
<td>6</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>E M A P L E</td>
<td>6</td>
<td>E M A P L E</td>
<td>A E E L M P</td>
</tr>
</tbody>
</table>

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>

order-of-growth running time for PQ with N items
Binary tree

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

Complete tree. Balanced except for bottom level.

Property. Height of binary heap with N nodes is \(1 + \lceil \log N \rceil\).

Pf. Height only increases when N is exactly a power of 2.

Binary heap

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

Heap-ordered binary tree.
• Keys in nodes.
• No smaller than children’s keys.

Array representation.
• Take nodes in level order.
• No explicit links needed!

Binary heap properties

Property A. Largest key is at root.

Property B. Can use array indices to move through tree.
• Note: indices start at 1.
• Parent of node at k is at \(k/2\).
• Children of node at k are at \(2k\) and \(2k+1\).
Promotion in a heap

Scenario. Exactly one node has a larger key than its parent.

To eliminate the violation:
• Exchange with its 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.

Demotion in a heap

Scenario. Exactly one node has a smaller key than does a child.

To eliminate the violation:
• Exchange with 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 demote.

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

Priority queue operations (heap implementation)

- insert
- insert
- insert
- remove max
- insert
- insert
- insert
- remove max
- insert
- insert
- insert
- remove max

Binary heap considerations

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

Dynamic array resizing.
- Add no-arg constructor.
- Apply repeated doubling and shrinking.

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

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

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 running time for PQ with N items

Hopeless challenge. Make all operations constant time.

Q. Why hopeless?
Heapsort

**First pass.** Build heap using bottom-up method.

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

**Second pass.** Sort.

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

Basic plan for in-place sort.

- Create max-heap with all $N$ keys.
- Repeatedly remove the maximum key.
Heapsort: Java implementation

```java
public class Heap {
    public static void sort(Comparable[] pq) {
        int N = pq.length;
        for (int k = N/2; k > 0; k--) {
            sink(pq, k, --N);
        }
        for (int k = N/2; k >= 1; k--)
            sink(pq, k, N);
    }
    private static void exch(Comparable[] pq, int i, int j) {
        /* as before */
    }
    private static void sink(Comparable[] pq, int k, int N) {
        /* as before */
    }
    private static boolean less(Comparable[] pq, int i, int j) {
        /* as before */
    }
}
```

but use 1-based indexing

Heapsort: mathematical analysis

Property D. At most \(2 \cdot N \lg N\) compares.

Significance. Sort in \(N \log N\) worst-case without using extra memory.
- 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 memory.

Sorting algorithms: summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>x</td>
<td>(N^{3/2})</td>
<td>(N^{3/2}/2)</td>
<td>(N^{3/2}/2)</td>
<td>(N) exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>(N^{3/2})</td>
<td>(N^{3/4})</td>
<td>(N) use for small (N) or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>(N)</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td>(N^{3/2})</td>
<td>(2N\ln N)</td>
<td>(N\lg N)</td>
<td>(N\lg N) probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>(N^{3/2})</td>
<td>(2N\ln N)</td>
<td>(N\lg N)</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>(N\lg N)</td>
<td>(N\lg N)</td>
<td>(N\lg N)</td>
<td>(N\lg N) guarantee, stable</td>
</tr>
<tr>
<td>heap</td>
<td>x</td>
<td>(2N\lg N)</td>
<td>(2N\lg N)</td>
<td>(N\lg N)</td>
<td>(N\lg N) guarantee, in-place</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>(N\lg N)</td>
<td>(N\lg N)</td>
<td>(N\lg N)</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>

Heapsort trace (array contents after each sink)
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 are exerted.

**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();
   }
}
```

% java BouncingBalls 100

Main simulation loop
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.

Main drawbacks.
• \( N^2/2 \) overlap checks per time quantum.
• Simulation is too slow if dt is very small.
• May miss collisions if dt is too large and 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 $\sigma$ 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?

$$
\Delta t = \frac{1 - \sigma - r_x}{v_x}
$$

Collision prediction.
- Particle $i$: radius $\sigma_i$, position $(r_{x_i}, r_{y_i})$, velocity $(v_{x_i}, v_{y_i})$.
- Particle $j$: radius $\sigma_j$, position $(r_{x_j}, r_{y_j})$, velocity $(v_{x_j}, v_{y_j})$.
- Will particles $i$ and $j$ collide? If so, when?

$$
\Delta t = \frac{d}{v_x}, \quad d = \frac{(\Delta v \cdot \Delta r)^2}{\Delta v \cdot \Delta v}
$$

$$
\Delta v = (\Delta v_x, \Delta v_y) = (v_{x_i} - v_{x_j}, v_{y_i} - v_{y_j})
$$

$$
\Delta r = (\Delta r_x, \Delta r_y) = (r_{x_i} - r_{x_j}, r_{y_i} - r_{y_j})
$$

$$
\Delta v \cdot \Delta v = (\Delta v x)^2 + (\Delta v y)^2
$$

$$
\Delta r \cdot \Delta r = (\Delta r x)^2 + (\Delta r y)^2
$$

Newton's second law (momentum form)

$$
J_x = \frac{f \Delta r x}{m_i}, \quad J_y = \frac{f \Delta r y}{m_j}, \quad f = \frac{2m_i m_j (\Delta v \cdot \Delta r)}{\sigma (m_i + m_j)}
$$

Impulse due to normal force
(conservation of energy, conservation of momentum)
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 dt(Particle that) { }
    public double dtX() { }
    public double dtY() { }
    public void bounce(Particle that) { }
    public void bounceX() { }
    public void bounceY() { }
}
```

Particle-particle collision and resolution implementation

```java
public double dt(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 dvdr = dx*dvx + dy*dvy;
    if( dvdr > 0) return INFINITY;
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = this.radius + that.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY;
    return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

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

```java
public 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 double time()   { return time; }
    public Particle a()    { return a;    }
    public Particle b()    { return b;    }
    public int compareTo(Event that) {   return this.time - that.time;   }
    public boolean isValid() {   }
}
```
public class CollisionSystem
{
    private MinPQ<Event> pq;        // the priority queue
    private double t  = 0.0;  ... + a.dtY(), null, a));
    private void redraw()  { }
    public void simulate() {  /* see next slide */  }
}

Collision system implementation: skeleton

Simulation example 1
% java CollisionSystem 100

Simulation example 2
% java CollisionSystem < billiards.txt
Simulation example 3

% java CollisionSystem < brownian.txt

Simulation example 4

% java CollisionSystem < diffusion.txt