### Priority Queues

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

### Priority queue applications

- Event-driven simulation.  [customers in a line, colliding particles]
- Numerical computation.  [reducing roundoff error]
- Data compression.  [Huffman codes]
- Graph searching.  [Dijkstra’s algorithm, Prim’s algorithm]
- Computational number theory.  [sum of powers]
- Artificial intelligence.  [A* search]
- Statistics.  [maintain largest M values in a sequence]
- Operating systems.  [load balancing, interrupt handling]
- Discrete optimization.  [bin packing, scheduling]
- Spam filtering.  [Bayesian spam filter]

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

### Priority queue API

**Keys.** Items that can be compared.

**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.

```java
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());
```

### Cost of finding the largest M in a stream of N items

- **operation**
- **argument**
- **return value**

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

### Problem

Find the largest M of a stream of N elements.

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.
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></td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td></td>
<td>3</td>
<td>P E X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
<td>4</td>
<td>P E X A</td>
<td>A E P X</td>
</tr>
<tr>
<td>remove max</td>
<td>M</td>
<td></td>
<td>5</td>
<td>P E X A M</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>5</td>
<td>P E M A P</td>
<td>A E M P P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
<td>6</td>
<td>P E M A P L</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td></td>
<td>6</td>
<td>P E M A P L E</td>
<td>A E E L M P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue

Priority queue: unordered array implementation

public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;   // pq[i] = ith element on pq
   private int N;      // number of elements on pq

   public UnorderedMaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity];  }

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

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

   public Key delMax()
   {  int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}

no generic array creation

Challenge. Implement all operations efficiently.

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

goal  

log N  

log N  

log N  

order-of-growth running time for PQ with N items
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 binary heap with N nodes is \(1 + \lfloor \log_2 N \rfloor\).

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.
• 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.

**Insertion in a heap**

**Insert.** Add node at end, then promote.

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

**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
- Remove max
- Insert
- Remove max
- Insert
- Remove max
- 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

- Unordered array: Insert N, Del max N, Max N
- Ordered array: Insert N, Del max 1, Max 1
- Binary heap: Insert log N, Del max log N, Max 1

Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;
    private int N;
    public MaxPQ(int capacity)
    {   ... }
    public boolean isEmpty()
    {   return N == 0;   }
    public void insert(Key key)
    {   /* see previous code */ }
    public Key delMax()
    {   /* see previous code */ }
    private void exch(int i, int j)
    {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;   }
    private void swim(int k)
    {   /* see previous code */ }
    private void sink(int i, int j)
    {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;   }
}
```

Hopeless challenge. Make all operations constant time.

Q. Why hopeless?
Heapsort

First pass. Build heap using bottom-up method.

\[\text{for (int } k = \text{N/2}; k \geq 1; k--)\]
\[\text{sink(a, k, N);}\]

Second pass. Sort.

\[\text{while (N > 1)}\]
\[\{\]
\[\text{exch(a, 1, N--);}\]
\[\text{sink(a, 1, N);}\]
\[\}\]

Heapsort

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);
        }
        while (N > 1) {
            exch(pq, 1, N);
            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) {
        if (N > 1) {
            exchange(pq, 1, N);
            sink(pq, k, N);
        }
    }

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

    private static void exchange(Comparable[] pq, int i, int j) {
        /* as before */
    }
}
```

but use 1-based indexing

Heapsort: trace

```
0 1 2 3 4 5 6 7 8 9 10 11
S O R T E X A M P L E
```

property D. At most $2N \log N$ 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>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$N^2/4$</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>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N \log N$</td>
<td>$N \log N$ probabilistic guarantee</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N$</td>
<td>fastest in practice</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee, stable</td>
</tr>
<tr>
<td>heap</td>
<td>x</td>
<td>$2N \log N$</td>
<td>$2N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee, in-place</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</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.

**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();
      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: what object does the checks? 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 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 a 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?

Collision resolution:
- Velocity after collision: $(-v_x, v_y)$
- Position after collision: $(1 - \sigma - r_x)/v_x, r_y + \Delta t v_y$.

Wall at $x = 1$

Particle-particle collision prediction

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

Collision prediction:
- $\Delta t = \text{time to hit wall} = \frac{\text{distance/velocity}}{\text{distance}} = (1 - \sigma - r_x)/v_x$.

Particle-particle collision resolution

Collision resolution. When two particles collide, how does velocity change?
- $v_{xi}' = v_{xi} + J_x/m_i$
- $v_{yi}' = v_{yi} + J_y/m_i$
- $v_{xj}' = v_{xj} - J_x/m_j$
- $v_{yj}' = v_{yj} - J_y/m_j$

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

$J_x = \frac{1}{\sigma} \Delta r_x$, $J_y = \frac{1}{\sigma} \Delta r_y$, $J = \frac{2 m_i m_j(\Delta v \cdot \Delta r)}{\sigma (m_i + m_j)}$
**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 void bounce(Particle that) { }
    public void bounceX() { }
    public void bounceY() { }
}
```

**Particle-particle collision and resolution implementation**

```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() { }
}
```

**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.
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.dt(particles[i]);
            pq.insert(new Event(t + dt, a, particles[i]));
        }
        pq.insert(new Event(t + a.dtX(), a, null));
        pq.insert(new Event(t + a.dtY(), null, a));
    }

    private void redraw() { }
    public void simulate() { /* see next slide */ }
}
```

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.bounce(b);
        else if (a != null && b == null) a.bounceX();
        else if (a == null && b != null) b.bounceY();
        else if (a == null && b == null) redraw();
        predict(a);
        predict(b);
    }
}
```

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