### 2.4 Priority Queues

- API and elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation

---

### 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.

### Priority Queue API

**Requirement.** Generic items are 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 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

**Key must be Comparable (bounded type parameter)**
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]
- 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 client example**

**Challenge.** Find the largest $M$ items in a stream of $N$ items.
- Fraud detection: isolate $S$ transactions.
- File maintenance: find biggest files or directories.

**Constraint.** Not enough memory to store $N$ items.

```java
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.delMin();
}
```

**Priority queue client example**

**Challenge.** Find the largest $M$ items in a stream of $N$ items.

**Constraint.** Not enough memory to store $N$ items.

**Order of growth of finding the largest $M$ 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>$MN$</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>
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>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
<td>E P Q</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P E</td>
<td>E P Q</td>
<td>E P Q</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>4</td>
<td>P E XA</td>
<td>A E P X</td>
<td>A E P X</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>5</td>
<td>P E XAM</td>
<td>A E M P X</td>
<td>A E M P X</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>4</td>
<td>P E M A</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>5</td>
<td>P E M A P</td>
<td>A E M P P</td>
<td>A E M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>L</td>
<td>6</td>
<td>P E M APL</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>E</td>
<td>7</td>
<td>P E M APLE</td>
<td>A E E L M P P</td>
<td>A E E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>E M APLE</td>
<td>A E E L M P</td>
<td>A E E L M P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue

Priority queue elementary implementations

**Challenge.** Implement all operations efficiently.

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

<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>

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 tree with $N$ nodes is $\lfloor \log N \rfloor$.

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

A complete binary tree in nature

---

Binary heap representations

**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 $2k+1$. 
Promotion in a heap

Scenario. Child’s key becomes larger key 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.

Demotion in a heap

Scenario. Parent’s 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.

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 swim it up.
Cost. At most 1 + \log N compares.

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 sink it down.
Cost. At most 2 \log N compares.

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

Binary heap: Java implementation

```java
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)
    public Key delMax()
    { /* see previous code */ }

    private void swim(int k)
    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 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>
<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

why impossible?
Binary heap considerations

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

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: implementing in Java

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)
Heapsort

Basic plan for in-place sort.
- Create max-heap with all N keys.
- Repeatedly remove the maximum key.

Heapsort demo

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

Sortdown. Repeatedly delete the largest remaining item.

Heapsort: heap construction

First pass. Build heap using bottom-up method.
Heapsort: mathematical analysis

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

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

*Algorithm be improved to \( \sim N \lg N \).*

**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 memory.
- Not stable.
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>$\frac{1}{2} N^2$</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>x x</td>
<td>$\frac{1}{2} N^2$</td>
<td>$\frac{1}{4} N^2$</td>
<td>N</td>
<td>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>$\frac{1}{2} N^2$</td>
<td>$2 \log N$</td>
<td>$\log N$</td>
<td>N log N probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>$\frac{1}{2} N^2$</td>
<td>$2 \log N$</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</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>$2 N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>N log N guarantee, in-place</td>
</tr>
<tr>
<td>??</td>
<td>x x</td>
<td>$N \log N$</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.

[Diagram of molecular dynamics simulation of hard discs]

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
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);
            StdDraw.show(50);
        }
    }
}
```

Main interaction loop

- 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.

Fundamental challenge for time-driven simulation

- $dt$ too small: excessive computation
- $dt$ too large: may miss collisions

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)

Physics problems:
- when? what effect?

CS problems:
- which object does the check? too many checks?
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?

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?

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?

Important note: This is high-school physics, so we won’t be testing you on it!
Particle-particle collision resolution

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

$$ v_x' = v_x + J_x \frac{m_j}{m_i} $$
$$ v_y' = v_y + J_y \frac{m_j}{m_i} $$
$$ v_x'' = v_x - J_x \frac{m_j}{m_j} $$
$$ v_y'' = v_y - J_y \frac{m_j}{m_j} $$

Newton’s second law (momentum form)

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

Important note: This is high-school 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) { }
    public double timeToHitVerticalWall() { }
    public double timeToHitHorizontalWall() { }

    public void bounceOff(Particle that) { }
    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.
- "potential" since collision may not happen 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.

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) {
        add to PQ all particle-wall and particle-particle collisions involving this particle
    }

    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() {} // invalidate
    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.de1Min();
        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 s = new CollisionSystem(100); % 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