2.4 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 client example

Problem. Find the largest M in 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());
```

Priority queue API

```java
public class MaxPQ<Key extends Comparable<Key>>{
    // API for a generic priority queue
    int size(); // number of entries in the priority queue
    Key max(); // return the largest key
    Key delMax(); // return and remove the largest key
    boolean isEmpty(); // is the priority queue empty?
    void insert(Key v); // insert a key into the priority queue
    int maxN(); // create a priority queue of initial capacity maxN
    MaxPQ(); // create a priority queue
}
```

Implementation:

<table>
<thead>
<tr>
<th>data type</th>
<th>delete</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>N log N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>elementary PQ</td>
<td>M N M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary heap</td>
<td>N log M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>best in theory</td>
<td>N M</td>
<td></td>
<td></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></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>3</td>
<td>P Q E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>2</td>
<td>P E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>3</td>
<td>P E X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>4</td>
<td>P E X A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>5</td>
<td>P E X A M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>4</td>
<td>P E X A M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>6</td>
<td>P E M A P L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>7</td>
<td>P E M A P L E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>P E M A P L E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue

Priority queue: unordered array implementation

```java
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];
    }
}
```

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></td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>
Binary tree

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

**Complete tree.** Perfectly balanced, except for bottom level.

![](image)

Property. Height of complete tree with $N$ nodes is $1 + \lfloor \lg N \rfloor$.

**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 \( a[1] \), which is root of binary tree.

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 \( 2k+1 \).

Insertion in a heap

Insert. Add node at end, then swim it up.
Running time. At most \( \log N \) compares.

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

Promotion in a heap

Scenario. Node’s key becomes larger key than its parent’s key.

To eliminate the violation:
- Exchange key in node 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;
    }
}

Demotion in a heap

Scenario. Node’s key becomes smaller than one (or both) of its children’s keys.

To eliminate the violation:
- Exchange key in node 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.
Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Running time. At most \( \sim 2 \log N \) compares.

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

Heap operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unordered Array</th>
<th>Ordered Array</th>
<th>Binary Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( \log N )</td>
</tr>
<tr>
<td>Delete Max</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Max</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

Priority queues implementation cost summary

Hopeless challenge. Make all operations constant time.

Q. Why hopeless?
Binary heap considerations

Minimum-oriented priority queue.
- Replace \texttt{less()} with \texttt{greater().}
- Implement \texttt{greater().}

Dynamic array resizing.
- Add no-arg constructor.
- Apply repeated doubling and shrinking. \( \longrightarrow \) leads to \( O(\log N) \) amortized time per op

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. \( \longrightarrow \) easy to implement with \texttt{sink()} and \texttt{swim()} [stay tuned]
- Change the priority of an item.

Heapsort

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

Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

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 trace (array contents just after each sink)

<table>
<thead>
<tr>
<th>N</th>
<th>Initial</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>EXAMPLE</td>
<td>E</td>
</tr>
<tr>
<td>11</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>11</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>11</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>11</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>11</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>11</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

Heapsort: mathematical analysis

**Proposition Q.** At most $2N \log N$ compares and exchanges.

**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.
- Not stable.
### Heapsort animation

[Animation of Heapsort with 50 random elements.](http://www.sorting-algorithms.com/heap-sort)

### 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>$N^2/2$</td>
<td>$N^2/4$</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>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N \lg N$</td>
<td>$N \log N$, probabilistic guarantee, fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>$N^2/2$</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>$N \log N$, guaranteed, 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>$N \log N$, guaranteed, 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>

### 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);     }   }}
```

**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.

```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?
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)

Time-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.

Event-driven simulation

Particle-wall collision

Collision prediction and resolution.
• Particle of radius \( s \) at position \((x, y)\).
• 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 \((x_i, y_i)\), velocity \((v_{xi}, v_{yi})\).
• Particle \( j \): radius \( s_j \), position \((x_j, y_j)\), velocity \((v_{xj}, v_{yj})\).
• Will particles \( i \) and \( j \) collide? If so, when?
Particle data type skeleton

public class Particle
{
    private double rx, ry;  // position
    private double vx, vy;  // velocity
    private final double mass;  // mass
    private final double radius;  // radius
    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() { }
}

Important note: This is high-school 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.

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 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);
    }
}
```
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