Priority Queues

Priority Queue ADT
Binary heaps
Heapsort


Abstract Data Types

Idealized scenario.
- Design general-purpose ADT useful for many clients.
- Develop efficient implementation of all ADT functions.
- Each ADT provides a new level of abstraction.

Total cost depends on:
- ADT implementation.
- Client usage pattern.

Priority Queues

Records with keys (priorities) that can be compared.

Basic operations.
- Insert.
- Remove largest.
- Create.
- Test if empty.
- Copy.
- Destroy.

Not needed for one-time use, but critical in large systems when writing in C or C++.
Priority Queue Applications

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

customers in a line, colliding particles
reducing roundoff error
Huffman codes
shortest path, MST
sum of powers
A* search
maintain largest M values in a sequence
task scheduling, interrupt handling
bin packing heuristics
Bayesian spam filter

Priority Queue Client Example

Problem: Find the largest M of a stream of N elements.
Ex 1: Fraud detection - isolate $ transactions.
Ex 2: File maintenance - find biggest files or directories.
Possible constraint: may not have enough memory to store N elements.
Solution: Use a priority queue.

Ex: top 10,000 in a stream of 1 billion.
- Not possible without good algorithm.

Unordered Array Priority Queue Implementation

public class PQ {
    private Comparable[] pq;   // pq[i] = ith element
    private int N;             // number of elements on PQ
    public PQ() { pq = new Comparable[8]; }
    public boolean isEmpty() { return N == 0; }
    public void insert(Comparable x) {
        pq[N++] = x;
    }
    public Comparable delMax() {
        int max = 0;
        for (int i = 1; i < N; i++)
            if (less(pq[max], pq[i])) max = i;
        exch(pq, max, N-1);
        return pq[--N];
    }
}

Implementation Details

What if I don’t know the max capacity of the PQ ahead of time?
- Double the size of the array as needed.
- Add following code to insert before updating array.

if (N >= pq.length) {
    Comparable[] temp = new Comparable[2*N];
    for (int i = 0; i < N; i++)
        temp[i] = pq[i];
    pq = temp;
}

Memory leak.
- Garbage collector only reclaims memory if there is no outstanding reference to it.
- When deleting element N-1 from the priority queue, set:

    pq[N-1] = null;
Priority Queues Implementation Cost Summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Insert</th>
<th>Remove Max</th>
<th>Find Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>unordered list</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Can we implement all operations efficiently?

Heap

Heap: Array representation of a heap-ordered complete binary tree.

Binary tree.
- null or
- Node with links to left and right trees.

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

Array representation.
- Take nodes in level order.
- No explicit links needed since tree is complete.

Heap Properties

Largest key is at root.

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.

Length of path in N-node heap is at most \( \sim \lg N \).
- \( n \) levels when \( 2^n \leq N < 2^{n+1} \).
- \( n \leq \lg N < n+1 \).
- \( \sim \lg N \) levels.

Promotion (Bubbling Up) In a Heap

Suppose that exactly one node is bigger than its parent.

To eliminate the violation:
- Exchange with its parent.
- Repeat until heap order restored.

\[
\text{private void swim(int k) \{}
\text{while (k > 1 &\& less(k/2, k)) {}
\text{exch(k, k/2);}
\text{k = k/2;}
\text{}}
\text{}}
\text{parent of node at k is at k/2}
\]

Peter principle: node promoted to level of incompetence.
Demoting (Sifting Down) In a Heap

Suppose that exactly one node is smaller than a child.

To eliminate the violation:
1. Exchange with larger child.
2. Repeat until heap order restored.

```
private void sink(int k, int N) {
    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.

Insert and Delete Max

Insert. Add node at end, then promote.

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

Remove largest. Exchange root with node at end, then sift down.

```
public Comparable delMax() {
    exch(1, N);
    sink(1, N-1);
    return pq[N--];
}
```

Heap Based Priority Queue in Java

```
public class PQ {
    private Comparable[] pq; // exactly as in array-based PQ
    private int N;

    public PQ() {} // same as array-based PQ, but allocate one extra element in array
    public boolean isEmpty() {} // PQ ops
    public int size() {} // PQ ops
    public void insert(Comparable x) { } // PQ ops
    public Comparable delMax() { } // PQ ops
    private void swim(int k) {} // heap helper functions
    private void sink(int k, int N) {} // helper functions
    private boolean less(int i, int j) {} // helper functions
    private void exch(int i, int j) {} // helper functions
}
```

Expansion: double size of array as needed.
Memory leak: when deleting element N, set pq[N] = null.

Priority Queues Implementation Cost Summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Insert</th>
<th>Remove Max</th>
<th>Find Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>unordered list</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>heap</td>
<td>Ig N</td>
<td>Ig N</td>
<td>1</td>
</tr>
</tbody>
</table>

Hopeless challenge: get all ops O(1).
Digression: Heapsort

First pass: build heap.
- Add item to heap at each iteration, then sift up.
- Or can use faster bottom-up method; see book.

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

Second pass: sort.
- Remove maximum at each iteration.
- Exchange root with node at end, then sift down.

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

Significance of Heapsort

Q: Sort in $N \log N$ worst-case without using extra memory?
A: Yes. Heapsort.

Not mergesort? Linear extra space. challenge for bored: design in-place merge
Not quicksort? Quadratic in worst case. challenge for bored: design $O(N \log N)$ worst-case quicksort

Heapsort is OPTIMAL for both time and space, BUT
- Inner loop longer than quicksort's.
- Makes poor use of cache memory.

In the wild: g++ STL uses introsort.

challenge for bored: design in-place merge
combo of quicksort, heapsort, and insertion

Sorting Summary

<table>
<thead>
<tr>
<th></th>
<th>In-Place</th>
<th>Stable</th>
<th>Worst</th>
<th>Average</th>
<th>Best</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>X</td>
<td></td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>N</td>
<td>never use it</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>X</td>
<td></td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>N</td>
<td>N exchanges</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>X</td>
<td></td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>N</td>
<td>use as cutoff for small N</td>
</tr>
<tr>
<td>Shellsort</td>
<td>X</td>
<td></td>
<td>$N^{1/2}$</td>
<td>$N^{1/2}$</td>
<td>N</td>
<td>with Knuth sequence</td>
</tr>
<tr>
<td>Quicksort</td>
<td>X</td>
<td></td>
<td>$N^2/2$</td>
<td>$2N \log N$</td>
<td>$N \log N$</td>
<td>fastest in practice</td>
</tr>
<tr>
<td>Mergesort</td>
<td>X</td>
<td></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>Heapsort</td>
<td>X</td>
<td></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>
</tbody>
</table>

Sam Loyd’s 15-Slider Puzzle

15 puzzle.
- Legal move: slide neighboring tile into blank square.
- Challenge: sequence of legal moves to put tiles in increasing order.
- Win $1000 prize for solution.

http://www.javaonthebrain.com/java/puzz15/
### Breadth First Search of 8-Puzzle Game Tree

#### Initial State

```
1 2 3
4 5 6
7 8 5
```

#### Goal State

```
1 2 3
4 5 6
7 8 5
```

### A* Search of 8-Puzzle Game Tree

**Priority first search.**

- **Basic idea:** explore positions in more intelligent order.
- **Ex 1:** number of tiles out of order.
- **Ex 2:** sum of Manhattan distances + depth.

Implement A* algorithm with PQ.

### Event-Based Simulation

**Challenge:** animate N moving particles.

- Each has given velocity vector.
- Bounce off edges and one another upon collision.

**Example applications:** molecular dynamics, traffic, ...

**Naive approach:** t times per second

- Update particle positions.
- Check for collisions, update velocities.
- Redraw all particles.

**Problems:**

- $N^2 t$ collision checks per second.
- May miss collisions!

**Approach:** use PQ of events with time as key,

- Put collision event on PQ for each particle (calculate time of next collision as priority)
- Put redraw events on PQ (t per second)

**Main loop:** remove next event from PQ.

- Redraw: update positions and redraw.
- Collision: update velocity of affected particles and put new collision events on PQ.

**More PQ operations needed:**

- may need to remove items from PQ.
- may want to join PQs for different sets of events (Ex: join national for air traffic control).

**More sophisticated PQ interface needed**
More Priority Queue Operations

Indirect priority queue.
- Supports deletion of arbitrary elements.
- Use symbol table to access binary heap node, given element to delete.

Binomial queue.
- Supports fast join.
- Slightly relaxes heap property to gain flexibility.

Priority Queues Implementation Cost Summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Insert</th>
<th>Remove Max</th>
<th>Find Max</th>
<th>Change Key</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>unordered list</td>
<td>1</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>heap</td>
<td>lg N</td>
<td>lg N</td>
<td>1</td>
<td>lg N</td>
<td>N</td>
</tr>
<tr>
<td>binomial queue</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
</tr>
<tr>
<td>best in theory</td>
<td>1</td>
<td>lg N</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>