Priority Queues and Heapsort

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

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.

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
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

### Challenge
Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

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

```java
topM $M$ < tinyBatch.txt
```

<table>
<thead>
<tr>
<th>Transaction data type is Comparable (ordered by $$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pq</td>
</tr>
</tbody>
</table>

```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 ($N$ huge, $M$ large).

<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>$M$</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 QUOTES AND HEAPSORT

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
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</td>
<td>P, Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>3</td>
<td>P, Q</td>
<td>E</td>
<td>E, Q</td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>2</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>3</td>
<td>E</td>
<td>X</td>
<td>E</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>5</td>
<td>E, M</td>
<td>A</td>
<td>E, M</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>5</td>
<td>E, M</td>
<td>A</td>
<td>E, M</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>6</td>
<td>E, M</td>
<td>P</td>
<td>E, M</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>4</td>
<td>E, M, P</td>
<td>A</td>
<td>E, M, P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>7</td>
<td>E, M, P</td>
<td>A</td>
<td>E, M, P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>E, M, P</td>
<td>A</td>
<td>E, M, P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue

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>N</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>heap</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

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 Queues and Heapsort**

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
**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 \( \lfloor \log_2 N \rfloor \).

**Pf.** Height only increases when \( N \) is a power of 2.

**Heap**

**Heap:** a heap is a specialised tree-based data structure that satisfies the heap property.

**Heap Property:**
- **min-heap property:** the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- **max-heap property:** the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

**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.

```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.

### Insertion in a heap

**Insert.** Add node at end, then swim it up.

**Cost.** At most $1 + \log N$ compares.

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

### Demotion in a heap

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

To eliminate the violation:
- Exchange key in parent with key in 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 sink it down.

**Cost.** At most $2 \lg N$ compares.

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

Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

Binomial heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.
**Binary heap operations**

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

1. **insert S**
   - Violates heap order (swim up)

2. **heap ordered**

3. **remove the maximum**

**Binary heap operations**

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

1. **insert S**
   - Violates heap order (swim up)

2. **heap ordered**

3. **remove the maximum**
Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum
Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

violates heap order (sink down)

heap ordered

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

exchange with root
Binary heap operations

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

*Remove the maximum*
Binary heap operations

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 operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

insert S

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

insert S

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

insert S

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.
Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

**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) {
        /* see previous code */
    }

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

    private void swim(int k) {
        /* see previous code */
    }

    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>Fibonacci</td>
<td>log N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Data type. Set of values and operations on those values.

Immutable data type. Can’t change the data type value once created.

Immutable. String, Integer, Double, Color, Vector, Transaction, Point2D.
Mutable. StringBuilder, Stack, Counter, JAVA ARRAY.

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.

### Starting point
Array in arbitrary order.

1. 2 3 4 5 6 7 8 9 10 11

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

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

sink 5

Heapsort

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

sink 5

Heapsort

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

sink 5

Heapsort

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

sink 4
Heapsort

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

sink 4

3-node heap

sink 3

3-node heap

sink 3

3-node heap
Heapsort

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

sink 2

7-node heap
Heapsort

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

sink 1

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

end of construction phase

Heapsort

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

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11
Heapsort

Sort down. Repeatedly delete the largest remaining item.

exchange 1 and 11

sink 1

sink 1

sink 1
Heapsort

Sort down. Repeatedly delete the largest remaining item.

sink 1

exchange 1 and 10

exchange 1 and 10
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

\[
\begin{array}{cccccc}
R & P & E & O & L & E & A \\
M & S & T & X \\
\end{array}
\]

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

\[
\begin{array}{cccccc}
R & P & E & O & L & E & A \\
M & S & T & X \\
\end{array}
\]

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

\[
\begin{array}{cccccc}
M & P & E & O & L & E & A \\
R & S & T & X \\
\end{array}
\]

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

\[
\begin{array}{cccccc}
M & P & E & O & L & E & A \\
R & S & T & X \\
\end{array}
\]
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

sink 1

exchange 1 and 7
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7

sink 1

sink 1

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

exchange 1 and 6

sink 1

exchange 1 and 6
Sortdown. Repeatedly delete the largest remaining item.

Heapsort

Sink 1

Heapsort

Exchange 1 and 5
**Heapsort**

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5

```
+----------------+-
| L               |
+---+---+---+---+
| A | S | M | O |
+---+---+---+---+
| R | S | T | X |
+---+---+---+---+
```

sink 1

```
+----------------+-
| 1 L  |
+---+---+---+---+
| 1 S | E | M | O |
+---+---+---+---+
| 1 R | S | T | X |
+---+---+---+---+
```

sink 1

```
+----------------+-
| 2 E  |
+---+---+---+---+
| 2 L | M | O | P |
+---+---+---+---+
| 2 R | S | T | X |
+---+---+---+---+
```

sink 1

```
+----------------+-
| L               |
+---+---+---+---+
| E | M | O | P |
+---+---+---+---+
| R | S | T | X |
+---+---+---+---+
```
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

1. Exchange 1 and 4

2. Sink 1

Exchange 1 and 4

Sink 1
**Heapsort**

*Sortdown.* Repeatedly delete the largest remaining item.

1. Initial array: A E L M O P R S T X
2. Swap A and E
3. Sink 1

1. Initial array: A E L M O P R S T X
2. Exchange 1 and 3
3. Sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2

exchange 1 and 2
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

end of sortdown phase

A
E
E
L
M
O
P
R
S
T
X

Heapsort: constructing (left) and sorting down (right) a heap

Heapsort

**Ending point.** Array in sorted order.

A
E
E
L
M
O
P
R
S
T
X

A
E
E
L
M
O
P
R
S
T
X

1
2
3
4
5
6
7
8
9
10
11

Heapsort: sortdown

**Second pass.**
- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

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

Heapsort: heap construction

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

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

A
E
E
L
M
O
P
R
S
T
X

Heapsort: sortdown

**Second pass.**
- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.
public class Heap {
    public static void sort(Comparable[] pq) {
        int N = pq.length;
        for (int k = N/2; k >= 1; k--)
            sink(pq, k, N);
        while (N > 1) {
            exch(pq, 1, N);
            sink(pq, 1, --N);
        }
    }
    private static void sink(Comparable[] pq, int k, int N) {
        /* as before */
    }
    private static boolean less(Comparable[] pq, int i, int j) {
        /* as before */
    }
    private static void exch(Comparable[] pq, int i, int j) {
        /* as before */
    }
}

Heapsort trace

Heapsort animation

Heapsort: mathematical analysis

**Proposition.** Heap construction uses fewer than $2N$ compares and exchanges.

**Proposition.** Heapsort uses at most $2N \log N$ compares and exchanges.

**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>Name</th>
<th>Stable</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>x</td>
<td>N / 2</td>
<td>N / 2</td>
<td>N / 2</td>
<td>N exchanges</td>
</tr>
<tr>
<td>Insertion</td>
<td>x</td>
<td>x</td>
<td>N / 2</td>
<td>N</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</td>
<td>2 N ln N</td>
<td>N lg N</td>
<td>N log N probabilistic guarantee, fasted in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>N / 2</td>
<td>2 N ln N</td>
<td>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 guarantee, stable</td>
</tr>
<tr>
<td>Heap</td>
<td>x</td>
<td>2 N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N log N guarantee, in-place</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</td>
<td>N lg N</td>
<td>N lg N</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>