Priority Queues

- API
- elementary implementations
- binary heaps
- heapsort
Priority queues

• Application:
  – Collect records
  – Remove the one with maximum (minimum) key for processing
  – Continue collection
  – Remove the one with maximum (minimum) key for processing
  – etc…

• Generalization of
  – Stack: remove the newest
  – Queue: remove the oldest
Applications

- Events to be processed (e.g. chronological order)
- Job scheduling in computer systems (e.g. priority)
- Errors to be handled (bigger first)
- With sorting algorithms: remove largest element of sorted array
Priority queue API

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

```java
public class MaxPQ<Key extends Comparable<Key>>
{
    MaxPQ() // create an empty priority queue
    boolean isEmpty() // is the priority queue empty
    void insert(Key key) // insert a key
    Key delMax() // delete and return the maximum key
    Key max() // return the maximum key
    int size() // return the number of keys
}
```

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

```
Example

**Problem.** Find the largest $M$ of a stream of $N$ elements.
- Fraud detection: isolate $\$\$ transactions.
- File maintenance: find biggest files or directories.

<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>$M \log N$</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>

Cost of finding the largest $M$ in a stream of $N$ items
Priority queue client example

**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());
```
API
- elementary implementations
- binary heaps
- heapsort
- event-based simulation
Efficiency issues

• Easy to find implementations where *either* the insert or remove operation take constant time

• Both operations fast : difficult problem
Efficiency issues

• Ordered list
  – Fast \textit{find\_the\_maximum} operations
  – Slow \textit{insert} operations

• Unordered list
  – Slow \textit{find\_the\_maximum} operations
  – Fast \textit{insert} operations

• Which should be selected?
Priority queue: unordered and ordered array implementation

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>X</td>
<td>4</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>P</td>
<td>6</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>remove max</td>
<td>Q</td>
<td>2</td>
<td>P E</td>
<td>E P</td>
<td>E P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>3</td>
<td>P E X</td>
<td>E P X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>4</td>
<td>P E X A</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 X A M</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>insert</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>insert</td>
<td>L</td>
<td>6</td>
<td>P E M A P L</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>7</td>
<td>P E M A P L E</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 A P L E</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

---

For a better understanding, we can break down the sequence of operations and their outcomes:

**Insert operations**:
- **Insert (P)**: Inserts a new element with the value `P` into the priority queue. The queue becomes `[P]`.
- **Insert (Q)**: Inserts a new element with the value `Q` into the priority queue. The queue becomes `[P, Q]`.
- **Insert (E)**: Inserts a new element with the value `E` into the priority queue. The queue becomes `[P, Q, E]`.
- **Insert (X)**: Inserts a new element with the value `X` into the priority queue. The queue becomes `[P, Q, E, X]`.
- **Insert (A)**: Inserts a new element with the value `A` into the priority queue. The queue becomes `[P, Q, E, X, A]`.
- **Insert (M)**: Inserts a new element with the value `M` into the priority queue. The queue becomes `[P, Q, E, X, A, M]`.
- **Insert (L)**: Inserts a new element with the value `L` into the priority queue. The queue becomes `[P, Q, E, X, A, M, L]`.
- **Insert (E)**: Inserts a new element with the value `E` into the priority queue. The queue becomes `[P, Q, E, X, A, M, L, E]`.
- **Insert (P)**: Inserts a new element with the value `P` into the priority queue. The queue becomes `[E, M, A, P, L, E, P]`.

**Remove Max operations**:
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.
- **Remove Max**: Removes the maximum element from the priority queue. The queue becomes `[E, M, A, P, L]`.

The table shows how the priority queue changes with each operation, maintaining the priority order of elements.
Priority queue: unordered array implementation

```java
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
- `less()` and `exch()` as for sorting
 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>1</td>
</tr>
<tr>
<td>?</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

order-of-growth running time for PQ with N items
API
• elementary implementations
• **binary heaps**
• heapsort
• event-based simulation
Binary tree

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

Complete tree. Balanced except for bottom level.

Property. Height of binary heap 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 at root.

Property B. Can 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 $2k+1$. 
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.

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 promote.

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert P</td>
<td>![Insert P Diagram]</td>
</tr>
<tr>
<td>Insert Q</td>
<td>![Insert Q Diagram]</td>
</tr>
<tr>
<td>Insert E</td>
<td>![Insert E Diagram]</td>
</tr>
<tr>
<td>Remove max Q</td>
<td>![Remove max Q Diagram]</td>
</tr>
<tr>
<td>Insert X</td>
<td>![Insert X Diagram]</td>
</tr>
<tr>
<td>Insert A</td>
<td>![Insert A Diagram]</td>
</tr>
<tr>
<td>Insert M</td>
<td>![Insert M Diagram]</td>
</tr>
</tbody>
</table>

Priority queue operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert P</td>
<td>![Insert P Diagram]</td>
</tr>
<tr>
<td>Insert L</td>
<td>![Insert L Diagram]</td>
</tr>
<tr>
<td>Insert E</td>
<td>![Insert E Diagram]</td>
</tr>
<tr>
<td>Remove max P</td>
<td>![Remove max P Diagram]</td>
</tr>
</tbody>
</table>
Binary heap: Java implementation

```java
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 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;
    }
}
```
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.  
   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.  
• Change the priority of an item.  
   easy to implement with \texttt{sink()} and \texttt{swim()} [stay tuned]
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>
</tbody>
</table>

Order-of-growth running time for PQ with N items
API
- elementary implementations
- binary heaps
- heapsort
- event-based simulation
Heapsort

**Basic plan for in-place sort.**
- Create max-heap with all N keys.
- Repeatedly remove the maximum key.
First pass. Build heap using bottom-up method.

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

Second pass. Sort.
• 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);
}
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 */
    }
}

but use 1-based indexing
Heapsort: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>S O R T L X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>S O R T L X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>S O X T L R A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S T X P L R A M O E E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X T S P L R A M O E E</td>
</tr>
</tbody>
</table>

Initial values

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>T P S O L R A M E E X</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>S P R O L E A M E T X</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>R P E O L E A M S T X</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>P O E M L E A R S T X</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>O M E A L E P R S T X</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>M L E A E O P R S T X</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>L E E A M O P R S T X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A E E L M O P R S T X</td>
</tr>
</tbody>
</table>

Heap-ordered

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>T P S O L R A M E E X</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>S P R O L E A M E T X</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>R P E O L E A M S T X</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>P O E M L E A R S T X</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>O M E A L E P R S T X</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>M L E A E O P R S T X</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>L E E A M O P R S T X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A E E L M O P R S T X</td>
</tr>
</tbody>
</table>

Sorted result

A E E L M O P R S T X

Heapsort trace (array contents after each sink)
Heapsort: mathematical analysis

Property D. At most $2N \lg N$ 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.
## Sorting algorithms: summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>selection</strong></td>
<td>×</td>
<td></td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td><strong>insertion</strong></td>
<td>×</td>
<td>×</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><strong>shell</strong></td>
<td>×</td>
<td></td>
<td>?</td>
<td>?</td>
<td>$N$</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td><strong>quick</strong></td>
<td>×</td>
<td></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><strong>3-way quick</strong></td>
<td>×</td>
<td></td>
<td>$N^2/2$</td>
<td>$2N\ln N$</td>
<td>$N$</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td><strong>merge</strong></td>
<td></td>
<td>×</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><strong>heap</strong></td>
<td>×</td>
<td></td>
<td>$2N\lg N$</td>
<td>$2N\lg N$</td>
<td>$N\lg N$</td>
<td>$N\log N$ guarantee, in-place</td>
</tr>
<tr>
<td><strong>??</strong></td>
<td>×</td>
<td>×</td>
<td>$N\lg N$</td>
<td>$N\lg N$</td>
<td>$N\lg N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>