2.4 PRIORITY QUEUES

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation

see videos
2.4 PRIORITY QUEUES

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation
A **collection** is a data type that stores a group of items.

<table>
<thead>
<tr>
<th>data type</th>
<th>core operations</th>
<th>data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>Push, Pop</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td>queue</td>
<td>Enqueue, Dequeue</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td>priority queue</td>
<td>Insert, Delete-Max</td>
<td>binary heap</td>
</tr>
<tr>
<td>symbol table</td>
<td>Put, Get, Delete</td>
<td>binary search tree, hash table</td>
</tr>
<tr>
<td>set</td>
<td>Add, Contains, Delete</td>
<td>binary search tree, hash table</td>
</tr>
</tbody>
</table>

“Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won’t usually need your code; it’ll be obvious.” — Fred Brooks
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.
Generalizes: stack, queue, randomized queue.

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></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>A</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>X</td>
</tr>
<tr>
<td>insert</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>P</td>
</tr>
</tbody>
</table>
## Priority queue API

### Requirement

Keys are generic; they must also be Comparable.

### Code Snippet

```java
public class MaxPQ<Key extends Comparable<Key>> {
    // Methods
    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 a largest key
    }
    boolean isEmpty() {
        // is the priority queue empty?
    }
    Key max() {
        // return a largest key
    }
    int size() {
        // number of entries in the priority queue
    }
}
```

### Note

Duplicate keys allowed; `delMax()` picks any maximum key.
Priority queue: applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Discrete optimization. [bin packing, scheduling]
- Artificial intelligence. [A* search]
- Computer networks. [web cache]
- Operating systems. [load balancing, interrupt handling]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra’s algorithm, Prim’s algorithm]
- Number theory. [sum of powers]
- Spam filtering. [Bayesian spam filter]
- Statistics. [online median in data stream]
Challenge. Find the largest $m$ items in a stream of $n$ items.
- Fraud detection: isolate $$ transactions.
- NSA monitoring: flag most suspicious documents.

Constraint. Not enough memory to store $n$ items.

```java
MinPQ<Double> pq = new MinPQ<Double>();

while (StdIn.isEmpty())
{
    double value = StdIn.readDouble();
    pq.insert(value);
    if (pq.size() > m)
        pq.delMin();
}
```

use a min-oriented pq

pq now contains largest $m$ numbers

n huge, m large
Challenge. Find the largest \( m \) items 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>( m \log 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>

order of growth of finding the largest \( m \) in a stream of \( n \) items
Priority queue: elementary 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>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</td>
<td>A E M 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</td>
<td>A E L M 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</td>
<td>A E E L M 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>
In the worst case, what are the running times for \texttt{INSERT} and \texttt{DELETE-MAX} for a priority queue implemented with an \textit{ordered array}?

A. 1 and 1

B. 1 and $n$

C. $n$ and 1

D. $n$ and $n$
Challenge. Implement all operations efficiently.

<table>
<thead>
<tr>
<th>implementation</th>
<th>INSERT</th>
<th>DELETE-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>

order of growth of running time for priority queue with \(n\) items

Solution. Partially ordered array.
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.** Every level (except possibly the last) is completely filled; the last level is filled from left to right.

Property. Height of complete binary tree with \( n \) nodes is \( \lceil \lg n \rceil \).

Pf. Height increases only when \( n \) is a power of 2.
A complete binary tree in nature

Hyphaene Compressa - Doum Palm

© Shlomit Pinter
Binary heap: representation

**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 \( 2k+1 \).
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
Scenario. A key becomes larger than its parent's key.

To eliminate the violation:
- Exchange key in child with key in parent.
- Repeat until heap order restored.

Peter principle. Node promoted to level of incompetence.
**Binary heap: insertion**

**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);
}
```
Binary heap: demotion

Scenario. A 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.
Binary heap: delete the maximum

**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;
}
```
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 queue: implementations cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>INSERT</th>
<th>DELETE-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 of running time for priority queue with $n$ items
**Priority Queue with Delete-Random**

**Goal.** Design an efficient data structure to support the following ops:

- **INSERT:** insert a key.
- **DELETE-MAX:** delete and return a max key.
- **SAMPLE:** return a random.
- **DELETE-RANDOM:** delete and return a random key.
Binary heap: considerations

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

leads to log n amortized time per op (how to make worst case?)

can implement efficiently with `sink()` and `swim()` [ stay tuned for Prim/Dijkstra ]
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.

```java
public final class Vector {
    private final int n;
    private final double[] data;

    public Vector(double[] data) {
        this.n = data.length;
        this.data = new double[n];
        for (int i = 0; i < n; i++)
            this.data[i] = data[i];
    }

    // ... instance methods don't change instance variables
}
```

**Immutable.** String, Integer, Double, Color, Vector, Point2D, ...

**Mutable.** StringBuilder, Stack, Java array types, java.util.Date, ...
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.
- Simplifies concurrent programming.
- More secure in presence of hostile code.
- 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)
Binary heap: practical improvements

Do “half exchanges” in sink and swim.

- Reduces number of array accesses.
- Worth doing.
Floyd’s "bounce" heuristic.

- Sink key at root all the way to bottom.  
  only 1 compare per node
- Swim key back up.  
  some extra compares and exchanges
- Overall, fewer compares; more exchanges.

R. W. Floyd
1978 Turing award
Binary heap: practical improvements

**Multiway heaps.**
- Complete $d$-way tree.
- Parent’s key no smaller than its children’s keys.

**Fact.** Height of complete $d$-way tree on $n$ nodes is $\sim \log_d n$. 

![3-way heap diagram](image)
In the worst case, how many compares to \textsc{Insert} and \textsc{Delete-Max} in a $d$-way heap?

A. $\sim \log_d n$ and $\log_d n$

B. $\sim \log_d n$ and $d \log_d n$

C. $\sim d \log_d n$ and $\log_d n$

D. $\sim d \log_d n$ and $d \log_d n$
### Priority queue: implementation cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>INSERT</th>
<th>DELETE-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 ) (\dagger)</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>

\(\dagger\) amortized

Order-of-growth of running time for priority queue with \( n \) items

Sweet spot: \(d = 4\)

Why impossible?
2.4 PRIORITY QUEUES

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation
What are the properties of this sorting algorithm?

```java
public void sort(String[] a) {
    int n = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < n; i++)
        pq.insert(a[i]);
    for (int i = n-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

A. \( n \log n \) compares in the worst case.

B. In-place.

C. Stable.

D. All of the above.
Heapsort

Basic plan for in-place sort.

- View input array as a complete binary tree.
- Heap construction: build a max-heap with all \( n \) keys.
- Sortdown: repeatedly remove the maximum key.
Heapsort demo

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

for now, assume array entries are indexed 1 to n

array in arbitrary order

```
S
  /\  \\  \\
O  \ / \ / \
  T  R
    \ / \ / \\
    M  X
      \ / \
      P  A
        \ / \\
        E
```

```
<table>
<thead>
<tr>
<th>S</th>
<th>O</th>
<th>T</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>E</th>
<th>P</th>
<th>L</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>S</th>
<th>O</th>
<th>R</th>
<th>T</th>
<th>E</th>
<th>X</th>
<th>A</th>
<th>M</th>
<th>P</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
```
Heapsort demo

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

array in sorted order

```
\[\begin{array}{cccccccccccc}
A & E & E & L & M & O & P & R & S & T & X \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}\]
```
Heapsort: heap construction

First pass. Build heap using bottom-up method.

for (int k = n/2; k >= 1; k--)
sink(a, k, n);
Heapsort: sortdown

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

```plaintext
while (n > 1) {
    exch(a, 1, n--);
    sink(a, 1, n);
}
```
Heapsort: Java implementation

```java
public class Heap {
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int k = n/2; k >= 1; k--)
            sink(a, k, n);
        while (n > 1) {
            exch(a, 1, n);
            sink(a, 1, --n);
        }
    }

    private static void sink(Comparable[] a, int k, int n) {
        /* as before */
    }

    private static boolean less(Comparable[] a, int i, int j) {
        /* as before */
    }

    private static void exch(Object[] a, int i, int j) {
        /* as before */
    }
}
```

but make static (and pass arguments)

but convert from 1-based indexing to 0-base indexing
Heapsort: trace

Heapsort trace (array contents just after each sink)
Proposition. Heap construction makes $\leq n$ exchanges and $\leq 2n$ compares.

Pf sketch. [assume $n = 2^{h+1} - 1$]

$$h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \ldots + 2^h(0) = 2^{h+1} - h - 2$$

$$= n - (h - 1)$$

$\leq n$
Heapsort: mathematical analysis

**Proposition.** Heap construction uses $\leq 2n$ compares and $\leq n$ exchanges.

**Proposition.** Heapsort uses $\leq 2n \lg n$ compares and exchanges.

algorithm can be improved to $\sim 1n \lg n$
(but no such variant is known to be practical)

**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.
- Not stable.

in-place merge possible, not practical

n log n worst-case quicksort possible, not practical

can be improved using advanced caching tricks
Introsort

**Goal.** As fast as quicksort in practice; $n \log n$ worst case, in place.

**Introsort.**
- Run quicksort.
- Cutoff to heapsort if stack depth exceeds $2 \log n$.
- Cutoff to insertion sort for $n = 16$.

**In the wild.** C++ STL, Microsoft .NET Framework.
### Sorting algorithms: summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td>✔️</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>✔️</td>
<td>✔️</td>
<td>$n$</td>
<td>$\frac{1}{4} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>use for small $n$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td></td>
<td>$n \log_3 n$</td>
<td>?</td>
<td>$c n^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</td>
<td>✔️</td>
<td>$\frac{1}{2} n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td>✔️</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td></td>
<td>$n \log n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$n \log n$ probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔️</td>
<td></td>
<td>$n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>heap</td>
<td>✔️</td>
<td>✔️</td>
<td>$3 n$</td>
<td>$2 n \log n$</td>
<td>$2 n \log n$</td>
<td>$n \log n$ guarantee; in-place</td>
</tr>
<tr>
<td>?</td>
<td>✔️</td>
<td>✔️</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>