2.4 Priority Queues

- API and elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation (see videos)

https://algs4.cs.princeton.edu
2.4 **Priority Queues**

- *API and elementary implementations*
- *binary heaps*
- *heapsort*
- *event-driven simulation (see videos)*
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><strong>stack</strong></td>
<td>Push, Pop</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td><strong>queue</strong></td>
<td>Enqueue, Dequeue</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td><strong>priority queue</strong></td>
<td>Insert, Delete-Max</td>
<td>binary heap</td>
</tr>
<tr>
<td><strong>symbol table</strong></td>
<td>Put, Get, Delete</td>
<td>binary search tree, hash table</td>
</tr>
<tr>
<td><strong>set</strong></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
Collections allow adding and removing items. Which item to remove?

**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.
Priority queue API

**Requirement.** Keys are generic; they must also be 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 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]
- Data compression. [Huffman codes]
- Operating systems. [load balancing, interrupt handling]
- Graph searching. [Dijkstra’s algorithm, Prim’s algorithm]
- Number theory. [sum of powers]
- Spam filtering. [Bayesian spam filter]
- Statistics. [online median in data stream]
**Priority queue: elementary implementation**

**Exercise.** In the worst case, what are the running times for **INSERT** and **DELETE-MAX** for a priority queue implemented with

- an unordered array?
- an ordered array?

<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></td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td></td>
<td>2</td>
<td>E P</td>
<td>E P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></td>
<td>3</td>
<td>E P X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
<td>4</td>
<td>A E P X</td>
<td>A E P X</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
<td>5</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></td>
<td>4</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>5</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
<td>6</td>
<td>A E L M P</td>
<td>A E L M P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>7</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></td>
<td>6</td>
<td>E E M A P L</td>
<td>E E M A P L</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}? 

\begin{itemize}
  \item[A.] 1 and \( n \)
  \item[B.] 1 and \( \log n \)
  \item[C.] \( \log n \) and 1
  \item[D.] \( n \) and 1
\end{itemize}
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

what might this mean?

Solution. “Somewhat-ordered” array.
2.4 Priority Queues

- API and elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation

https://algs4.cs.princeton.edu
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.

Recursive definition

Property. Height of complete binary tree with \( n \) nodes is \( \lceil \log_2 n \rceil \).
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!
Which is the index of the parent of the item at index $k$ in a binary heap?

A. $k/2 - 1$
B. $k/2$
C. $k/2 + 1$
D. $2k$
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 $2k+1$. 

Heap representations
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**
**Binary heap: swim / promotion**

**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.

```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.
Binary heap: insertion

**Insert.** Add node at end in bottom level; then, swim it up.

**Cost.** At most $1 + \lg n$ compares.

```java
public void insert(Key x) {
    pq[++n] = x;
    swim(n);
}
```
**Binary heap: sink / 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.

```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.
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;
}
```
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; }
}
```

fixed capacity (for simplicity)
PQ ops
heap helper functions
array helper functions

https://algs4.cs.princeton.edu/24pq/MaxPQ.java.html
### 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
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.
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.

```
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 variables private and final (neither necessary nor sufficient, but good programming practice)

Defensive copy of mutable instance variables

Instance methods don’t change instance variables

Immutable in Java. String, Integer, Double, Color, File, ...
Mutable in Java. StringBuilder, Stack, URL, arrays, ...
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.
Binary heap: practical improvement

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
Priority queues: quiz 3

In the worst case, how many compares to INSERT and DELETE-MAX in a d-way heap?

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

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

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

D. $\sim d \log_d n$ and $\sim d \log_d n$
## 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&lt;sub&gt;d&lt;/sub&gt; n</td>
<td>d log&lt;sub&gt;d&lt;/sub&gt; n</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>log n †</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>

† amortized

- sweet spot: $d = 4$
- why impossible?

| order-of-growth of running time for priority queue with $n$ items |
Exercise.

- Assume there is a priority queue which makes a constant number of compares in the worst case for both \texttt{INSERT} and \texttt{DELETE-MAX}.
- Design a sorting algorithm that uses this priority queue.
- How many compares does it perform in the worst case?
2.4 Priority Queues

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

https://algs4.cs.princeton.edu
What are the properties of this sorting algorithm?

A. \( n \log n \) compares in the worst case.
B. In-place.
C. Stable.
D. All of the above.

```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();
}
```
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:

```
<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

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>A</th>
<th>E</th>
<th>E</th>
<th>L</th>
<th>M</th>
<th>O</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>X</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: heap construction**

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

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

**Key insight.**
After sink(a, k, n) completes, the subtree rooted at k is a heap.
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);
}
```

**Key insight.**
After each iteration, the array consists of a heap-ordered subarray followed by a sub-array in final order.
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

https://algs4.cs.princeton.edu/24pq/Heap.java.html
# Heapsort: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>initial values</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>SORTLXAMPLE</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORTLXAMPLE</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>SORTLXAMPLE</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>SORTLXAMPLE</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>SORTLXAMPLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>heap-ordered</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>XTSPLRAEMOE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sorted result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEEELMOPRSTX</td>
</tr>
</tbody>
</table>

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

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 makes $\leq n$ exchanges and $\leq 2n$ compares.

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

- algorithm can be improved to $\sim n \lg n$
  (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.

- can be improved using advanced caching tricks
# Sorting algorithms: summary

<table>
<thead>
<tr>
<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>$\frac{1}{2} n^2$</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>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td>✔</td>
<td>$\frac{1}{2} n \lg n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td>✔</td>
<td>$n \lg n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</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>
</tr>
<tr>
<td>heap</td>
<td>✔</td>
<td>✔</td>
<td>$3n$</td>
<td>$2 n \lg n$</td>
<td>$2 n \lg n$</td>
</tr>
<tr>
<td>?</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
</tr>
</tbody>
</table>