2.4 PRIORITY QUEUES

- APIs
- elementary implementations
- binary heaps
- heapsort
- event-driven simulation

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

- APIs
  - elementary implementations
  - binary heaps
  - heapsort
  - event-driven simulation

https://algs4.cs.princeton.edu
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><strong>Push, Pop</strong></td>
<td><strong>linked list</strong></td>
</tr>
<tr>
<td>queue</td>
<td><strong>Enqueue, Dequeue</strong></td>
<td><strong>resizing array</strong></td>
</tr>
<tr>
<td>priority queue</td>
<td><strong>Insert, Delete-Max</strong></td>
<td><strong>binary heap</strong></td>
</tr>
<tr>
<td>symbol table</td>
<td><strong>Put, Get, Delete</strong></td>
<td><strong>binary search tree</strong></td>
</tr>
<tr>
<td>set</td>
<td><strong>Add, Contains, Delete</strong></td>
<td><strong>hash table</strong></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.

A sequence of operations on a priority queue
triage in an emergency room (priority = urgency of wound/illness)

<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>M</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>M</td>
<td>X</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>P</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>
## Max-oriented priority queue API

### Requirement.
Must insert keys of the same (generic) type; moreover, keys must be Comparable.

```java
public class MaxPQ<Key extends Comparable<Key>> {
    MaxPQ() {
        // create an empty priority queue
    }
    void insert(Key v) {
        // insert a key
    }
    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.
Min-oriented priority queue API

Analogous to MaxPQ.

```java
public class MinPQ<Key extends Comparable<Key>> {
    MinPQ() {
        // create an empty priority queue
    }
    void insert(Key v) {
        // insert a key
    }
    Key delMin() {
        // return and remove a smallest key
    }
    boolean isEmpty() {
        // is the priority queue empty?
    }
    Key min() {
        // return a smallest key
    }
    int size() {
        // number of entries in the priority queue
    }
}
```

Warmup client. Sort a stream of integers from standard input.
 Priority queue: applications

- Event-driven simulation. [customers in a line, colliding particles]
- 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 = length of best known path

priority = “distance” to goal board

priority = event time
2.4 PRIORITY QUEUES

- APIs
- elementary implementations
- binary heaps
- heapsort
- event-driven simulation
Priority queue: elementary implementations

**Unordered list.** Store keys in a linked list.

![Linked list diagram]

**Performance.** **INSERT** takes $\Theta(1)$ time; **DELETE-MAX** takes $\Theta(n)$ time.
**Priority queue: elementary implementations**

**Ordered array.** Store keys in an array in ascending (or descending) order.

<table>
<thead>
<tr>
<th>a[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ordered array implementation of a MaxPQ
What are the worst-case running times for **INSERT** and **DELETE-MAX**, respectively, for a MaxPQ implemented with an **ordered array**?

A. $\Theta(1)$ and $\Theta(n)$
B. $\Theta(1)$ and $\Theta(\log n)$
C. $\Theta(\log n)$ and $\Theta(1)$
D. $\Theta(n)$ and $\Theta(1)$
Priority queue: implementations cost summary

**Elementary implementations.** Either `INSERT` or `DELETE-MAX` takes $\Theta(n)$ time.

<table>
<thead>
<tr>
<th>implementation</th>
<th>INSERT</th>
<th>DELETE-MAX</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered list</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><strong>goal</strong></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

**Challenge.** Implement both core operations efficiently.

**Solution.** “Somewhat-ordered” array.
2.4 PRIORITY QUEUES

- API
- 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.

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

Pf. Height increases (by 1) 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 tree.
- Keys in nodes.
- Child’s key no larger than parent’s key.

Array representation.
- Indices start at 1.
- Take nodes in level order.
- No explicit links!
Consider the node at index $k$ in a binary heap. Which Java expression gives the index of its parent?

A. $(k - 1) / 2$
B. $k / 2$
C. $(k + 1) / 2$
D. $2 * k$
Binary heap: properties

**Proposition.** Largest key is at index 1, which is root of binary tree.

**Proposition.** Can use array indices to move through tree.
- Parent of key at index \( k \) is at index \( \frac{k}{2} \).
- Children of key at index \( k \) are at indices \( 2k \) 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
**Binary heap: 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: demotion

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

**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 max. Exchange root with node at end; then, sink it down.

Cost. At most $2 \log n$ compares.

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[] a;
    private int n;

    public MaxPQ(int capacity)
    { a = (Key[]) new Comparable[capacity+1]; }  // see previous code

    public boolean isEmpty()
    { return n == 0; }  // see previous code

    public void insert(Key key)
    { return; }  // see previous code

    public Key delMax()
    { return; }  // see previous code

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

    private void sink(int k)
    { return; }  // see previous code

    private boolean less(int i, int j)
    { return a[i].compareTo(a[j]) < 0; }  // see previous code

    private void exch(int i, int j)
    { Key t = a[i]; a[i] = a[j]; a[j] = t; }  // see previous code

}
Goal. Implement both **INSERT** and **DELETE-MAX** in $\Theta(\log n)$ time.

<table>
<thead>
<tr>
<th>implementation</th>
<th><strong>INSERT</strong></th>
<th><strong>DELETE-MAX</strong></th>
<th><strong>MAX</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered list</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>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.

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

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

immutable in Java: String, Integer, Double
Goal. Design an efficient data structure to support the following ops:

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

Multiway heaps.
- Complete $d$-way tree.
- Child’s key no larger than parent’s key.

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

3-way heap
In the worst case, how many compares to \textbf{INSERT} and \textbf{DELETE-MAX} in a $d$-way heap as function of $n$ and $d$?

\begin{enumerate}
\item \(\sim \log_d n\) and \(\sim \log_d n\)
\item \(\sim \log_d n\) and \(\sim d \log_d n\)
\item \(\sim d \log_d n\) and \(\sim \log_d n\)
\item \(\sim d \log_d n\) and \(\sim d \log_d n\)
\end{enumerate}
## 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 $n$†</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

Order-of-growth of running time for priority queue with $n$ items

- sweet spot: $d = 4$
- see COS 423
- why impossible?
2.4 PRIORITY QUEUES

- APIs
- elementary implementations
- binary heaps
- heapsort
- event-driven simulation

https://algs4.cs.princeton.edu
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. $\Theta(n \log n)$ 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.
Heap construction

**Top-down approach.** Insert keys into a max-oriented heap, one at a time.
- Intuitive swim-based approach.
- $\Theta(n \log n)$ compares in worst case.

**Bottom-up approach.** Successively build larger heap from smaller ones.
- Clever sink-based alternative.
- $\Theta(n)$ compares. [stay tuned]
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
M

R
X
A

S O R T E X A M P L E
1 2 3 4 5 6 7 8 9 10 11
```
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

array in sorted order

A

E

L

R

1 2 3 4 5 6 7 8 9 10 11
Heapsort: heap construction

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

**Invariant.** After calling \( \text{sink}(a, k, n) \), trees rooted at \( k \) to \( n \) are heap-ordered.

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

![Heap construction diagram](image-url)
Heapsort: sortdown

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

Invariants. After calling `sink(a, 1, k)
- `a[k..n]` are in final sorted order.
- `a[1..k-1]` is a heap.

```java
int k = n;
while (k > 1)
{
    exch(a, 1, k--);
    sink(a, 1, k);
}
```
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);
        int k = n;
        while (k > 1)
        {
            exch(a, 1, k--);
            sink(a, 1, k);
        }
    }
}

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

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

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

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

Heapsort trace (array contents just after each sink)
Heapsort animation

50 random items

https://www.toptal.com/developers/sorting-algorithms/heap-sort
**Heapsort: mathematical analysis**

**Proposition.** Heap construction makes \( \leq n \) exchanges and \( \leq 2n \) compares.

**Pf sketch.** [assume \( n = 2^{h+1} - 1 \)]

\[
\begin{align*}
  h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^h(0) &= 2^{h+1} - h - 2 \\
  &= n - (h - 1) \\
  &\leq n
\end{align*}
\]
Heapsort: mathematical analysis

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

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

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

**Significance.** In-place sorting algorithm with $\Theta(n \log n)$ worst-case.

- Mergesort: no, $\Theta(n)$ extra space. in-place merge possible, not practical
- Quicksort: no, $\Theta(n^2)$ time in worst case. $\Theta(n \log n)$ worst-case quicksort possible, not practical
- 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
**Goal.** As fast as quicksort in practice; $\Theta(n \log n)$ worst case; in place.

**Introsort.**

- Run quicksort.
- Cutoff to heapsort if stack depth exceeds $2 \lg n$.
- 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>merge</td>
<td>✔️</td>
<td></td>
<td>$\frac{1}{2} n \lg n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>$n \log n$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td>✔️</td>
<td>$n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>improves mergesort when preexisting order</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>
<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>$2 n \lg n$</td>
<td>$2 n \lg n$</td>
<td>$n \log n$ guarantee; in-place</td>
</tr>
<tr>
<td>?</td>
<td>✔️</td>
<td>✔️</td>
<td>$n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>