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
**Collections**

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</td>
</tr>
<tr>
<td>queue</td>
<td>Enqueue, Dequeue</td>
<td>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</td>
</tr>
<tr>
<td>set</td>
<td>Add, Contains, Delete</td>
<td>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.

---

Triage in an emergency room
(priority = urgency of wound/illness)
Max-oriented priority queue API

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

```
public class MaxPQ<Key extends Comparable<Key>>
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MaxPQ()</code></td>
<td>create an empty priority queue</td>
</tr>
<tr>
<td><code>void insert(Key v)</code></td>
<td>insert a key</td>
</tr>
<tr>
<td><code>Key delMax()</code></td>
<td>return and remove a largest key</td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>is the priority queue empty?</td>
</tr>
<tr>
<td><code>Key max()</code></td>
<td>return a largest key</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>number of entries in the priority queue</td>
</tr>
</tbody>
</table>

**Note.** Duplicate keys allowed; `delMax()` removes and returns any maximum key.
Min-oriented priority queue API

Analogous to MaxPQ.

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.

![Unordered 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.

```
a[]
```

```
0  1  2  3  4  5  6  7  8  9  10
11  22  33  33  44  55  99
n
```

ordered array implementation of a MaxPQ
What are the worst-case running times for \textsc{Insert} and \textsc{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
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.

![Complete binary tree with 16 nodes](image)

**Property.** Height of complete binary tree with $n$ nodes is $\lceil \log_2 n \rceil$.

**Pf.** As you add nodes, 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. \( \frac{k - 1}{2} \)

B. \( \frac{k}{2} \)

C. \( \frac{k + 1}{2} \)

D. \( 2 \times k \)
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 \( k/2 \).
- Children of key at index \( k \) are at indices \( 2^*k \) and \( 2^*k + 1 \).
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 $1 + \log_2 n$ compares.

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

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 $2 \log_2 n$ 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[] a;
    private int n;

    public MaxPQ(int capacity)
    {
        a = (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 a[i].compareTo(a[j]) < 0;
    }

    private void exch(int i, int j)
    {
        Key temp = a[i]; a[i] = a[j]; a[j] = temp;
    }
}

https://algs4.cs.princeton.edu/24pq/MaxPQ.java.html
Priority queue: implementations cost summary

Goal. Implement both `INSERT` and `DELETE-MAX` in $\Theta(\log n)$ 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>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 \texttt{less()} with \texttt{greater()}.  
- Implement \texttt{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.
**Goal.** Design an efficient data structure to support the following API:

- **INSERT:** insert a key.
- **DELETE-MAX:** return and remove a largest key.
- **SAMPLE:** return a random key.
- **DELETE-RANDOM:** return and remove a random key.
Goal. Delete a random key from a binary heap in $O(\log n)$ time.
Multiway heaps

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 **INSERT** and **DELETE-MAX** in a $d$-way heap as function of both $n$ and $d$?

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_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

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

Order of growth of running time for priority queue with $n$ items
2.4 Priority Queues

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

```
public void sort(String[] a)
{
    int n = a.length;
    MinPQ<String> pq = new MinPQ<String>();
    
    for (int i = 0; i < n; i++)
        pq.insert(a[i]);
    
    for (int i = 0; i < n; i++)
        a[i] = pq.delMin();
}
```

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-oriented 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.

$$\log_2 1 + \log_2 2 + \ldots + \log_2 n = \log_2 (n!) \sim n \log_2 n$$

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

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

**Invariant.** After calling 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);
```
Heapsort: sortdown

Second pass.

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

Invariants. After calling $\text{sink}(a, 1, k)$

- $a[k..n]$ are in final sorted order.
- $a[1..k-1]$ is a heap.

```c
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 */ }  // but convert from 1-based indexing to 0-base indexing
}
Heapsort trace

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>k</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11</td>
</tr>
<tr>
<td>-----</td>
<td>---</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initial values</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>SORT L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORT L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>SORT L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>SX PL RAM O E E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>SX PL RAM O E E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>heap-ordered</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>TPS O L R A M E E X</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>SPR O L E A M E T X</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>RPE O L E A M S T X</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>POE M L E A R S T X</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>OME A L E P R S T X</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>MLE A E O P R S T X</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>LE A M O P R S T X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>EAE L M O P R S T X</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>EAE L M O P R S T X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>AE E L M O P R S T X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sorted result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AE E L M O P R S T X</td>
</tr>
</tbody>
</table>

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 \( n = 2^{h+1} - 1 \)]

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

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

**Proposition.** Heapsort makes \( \leq 2n \log_2 n \) compares and exchanges.

algorithm can be improved to \( \sim n \log_2 n \)

(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.
- **Quicksort:** no, \( \Theta(n^2) \) 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.

\( \Theta(n \log n) \) worst-case quicksort possible, not practical

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 $2 \log_2 n$.
- 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>$\frac{1}{2} n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$\Theta(n \log n)$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td></td>
<td>$n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>improves mergesort when pre-existing order</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td></td>
<td>$n \log_2 n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\Theta(n \log n)$ 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>$2 n \log_2 n$</td>
<td>$2 n \log_2 n$</td>
<td>$\Theta(n \log n)$ guarantee; in-place</td>
</tr>
<tr>
<td>?</td>
<td>✔️ ✔️</td>
<td></td>
<td>$n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>

number of compares to sort an array of $n$ elements