2.4 PRIORITY QUEUES

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

see chapter 6

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

- APIs
  - 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><strong>Push, Pop</strong></td>
<td>singly linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>resizing array</td>
</tr>
<tr>
<td>queue</td>
<td><strong>Enqueue, Dequeue</strong></td>
<td></td>
</tr>
<tr>
<td>deque</td>
<td><strong>Add–First, Remove–First, Add–Last, Remove–Last</strong></td>
<td>doubly linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>resizing array</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></td>
<td></td>
<td><strong>hash table</strong></td>
</tr>
<tr>
<td>set</td>
<td><strong>Add, Contains, Delete</strong></td>
<td></td>
</tr>
</tbody>
</table>
Priority queue

Collections. Insert and remove 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.

A sequence of operations on a priority 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>Q</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></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>X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></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>P</td>
<td></td>
</tr>
</tbody>
</table>

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

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

```
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

    Key max() return a largest key

    boolean isEmpty() is the priority queue empty?

    int size() number of keys in the priority queue
```

Note. Duplicate keys allowed; delMax() removes and returns 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

    Key min()
    return a smallest key

    boolean isEmpty()
    is the priority queue empty?

    int size()
    number of keys 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
Unordered list. Store keys in a linked list.

Performance. \textsc{Insert} takes $\Theta(1)$ time; \textsc{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>44</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 \texttt{INSERT} and \texttt{DELETE-MAX}, respectively, in a \texttt{MaxPQ} implemented with an \texttt{ordered array}?

\begin{itemize}
  \item \textbf{A.} $\Theta(1)$ and $\Theta(n)$
  \item \textbf{B.} $\Theta(1)$ and $\Theta(\log n)$
  \item \textbf{C.} $\Theta(\log n)$ and $\Theta(1)$
  \item \textbf{D.} $\Theta(n)$ and $\Theta(1)$
\end{itemize}
Priority queue: implementations cost summary

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

<table>
<thead>
<tr>
<th>implementation</th>
<th><strong>INSERT</strong></th>
<th><strong>DELETE-MAX</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered list</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td><strong>goal</strong></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 **INSERT** and **DELETE-MAX** 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 two disjoint binary trees (left and right subtrees).

**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 \( \lfloor \log_2 n \rfloor \).

**Pf.** As you successively add nodes, height increases (by 1) only when \( n \) is a power of 2.
A complete binary tree in nature (of height 4)
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!

```
a[]  0  1  2  3  4  5  6  7  8  9  10  11
   - T S R P N O A E I H G
```
Consider the node at index \( k \) in a binary heap. Which Java expression produces the index of its parent?

A. \((k - 1) / 2\)
B. \(k / 2\)
C. \((k + 1) / 2\)
D. \(2 * k\)
Proposition. Largest key is at index 1, which is root of binary tree.

Proposition. Can use array indices to move up or down tree.
- Parent of key at index \( k \) is at index \( k/2 \).
- Children of key at index \( k \) are at indices \( 2k \) and \( 2k + 1 \).

```
 0 1 2 3 4 5 6 7 8 9 10 11
```

```
T S R P N O A E I H G
```
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

```
T  P  R  N  H  O  A  E  I  G
```

```
T
 |
 ---
P
 |   
 N   H
 |
 ---
E   I
 |   
 G
```

```
T
 |
 ---
R
 |   
 O
 |
 ---
A
```
Binary heap: promotion

**Scenario.** Key in node becomes *larger* than key in parent’s node.

**To eliminate the violation:**
- Exchange key in child node with key in parent node.
- 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.
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);
}
```
Binary heap: demotion

**Scenario.** Key in node becomes *smaller* than one (or both) of keys in childrens' nodes.

To eliminate the violation:

- Exchange key in parent node with key in larger child’s node.
- 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 $2 \log_2 n$ compares.

```java
public Key delMax() {
    Key max = pq[1];
    exch(1, n--);
    sink(1);
    pq[n+1] = null;
    return max;
}
```

prevent loitering
public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] a;
    private int n;

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

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

<table>
<thead>
<tr>
<th>implementation</th>
<th>\texttt{INSERT}</th>
<th>\texttt{DELETE-MAX}</th>
<th>\texttt{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 `less()` with `greater()`.
- Implement `greater()`.

Other operations.
- Remove an arbitrary item.
- Change the priority of an item.
  can implement efficiently with `sink()` and `swim()`
  [stay tuned for Prim/Dijkstra]

Immutability of keys.
- Assumption: client does not change keys while they’re on the PQ.
- Best practice: use immutable keys.
  immutable in Java: String, Integer, Double, ...
**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.

Property. Height of complete $d$–way tree on $n$ nodes is $\sim \log_d n$.

Property. Children of key at index $k$ at indices $3k - 1$, $3k$, and $3k + 1$; parent at $\left\lceil (k + 1) / 3 \right\rceil$.

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 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>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>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

```java
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.  
  we'll assume 1-indexed for now
- Phase 1 (heap construction): build a max-oriented heap.
- Phase 2 (sortdown): repeatedly remove the maximum key.  
  a version of selection sort
Heapsort: top-down heap construction

Phase 1 (top-down heap construction).

- View input array as complete binary tree.
- Insert keys into a max heap, one at a time.

**before inserting X**

<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>S</td>
<td>R</td>
<td>O</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
</tbody>
</table>

**after inserting X**

<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>S</td>
<td>T</td>
<td>O</td>
<td>E</td>
<td>R</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
</tbody>
</table>

**max heap**

**untouched**

swim(6)
Heapsort: sortdown

Phase 2 (sortdown).

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

before deleting P

```
    O
   /|
  P  L
 /|
M E A
```

after deleting P

```
    O
   /|
  M  L
 /|
A E P
```

max heap final sorted order

max heap final sorted order

exch(1, 7)
sink(1)
public class HeapTopDown
{
    public static void sort(Comparable[] a)
    {
        // top-down heap construction
        int n = a.length;
        for (int k = 1; k <= n; k++)
            swim(a, k);

        // sortdown
        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 */ }

private static void swim(Comparable[] a, int k)
{ /* 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 */ }

...
Heapsort: mathematical analysis

**Proposition.** Heapsort uses only $\Theta(1)$ extra space.

**Proposition.** Heapsort makes $\leq 3n \log_2 n$ compares (and $\leq 2n \log_2 n$ exchanges).
- Top–down heap construction: $\leq n \log_2 n$ compares (and exchanges).
- Sortdown: $\leq 2n \log_2 n$ compares (and $\leq n \log_2 n$ exchanges).

**Bottom–up heap construction.** [see book] Successively building larger heap from smaller ones.

**Proposition.** Makes $\leq 2n$ compares (and $\leq n$ exchanges).

---

**Diagram:**

- Goal: 7-node heap
- 3-node heap
- Successively building larger heap from smaller ones

---
Heapsort: context

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

- Mergesort: no, $\Theta(n)$ extra space.  \(\leftarrow\) in-place merge possible; not practical
- Quicksort: no, $\Theta(n^2)$ time in worst case.  \(\leftarrow\) $\Theta(n \log n)$ 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.
- Not stable.
Introsort

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

**Introsort.**

- Run quicksort.
- Cutoff to heapsort if function-call stack depth exceeds $2 \log_2 n$.
- Cutoff to insertion sort for $n \leq 16$.

**In the wild.** C++ STL, Microsoft .NET Framework, Go.
## 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><em>n exchanges</em></td>
</tr>
<tr>
<td>insertion</td>
<td>✔️✔️</td>
<td>$n$</td>
<td>$\frac{1}{4} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td><em>use for small n or partially ordered</em></td>
</tr>
<tr>
<td>merge</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>$n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td><em>improves mergesort when pre-existing order</em></td>
</tr>
<tr>
<td>quick</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>$n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</td>
<td><em>improves quicksort when duplicate keys</em></td>
</tr>
<tr>
<td>heap</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>$n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td><em>holy sorting grail</em></td>
</tr>
</tbody>
</table>

Number of compares to sort an array of n elements.