2.4 **Priority Queues**

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

Visit [https://algs4.cs.princeton.edu](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>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.

triage in an emergency room
(priority = urgency of wound/illness)
Priority queue API

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

```
public class MaxPQ<Key extends Comparable<Key>> {
    MaxPQ();
    void insert(Key v);
    Key delMax();
    boolean isEmpty();
    Key max();
    int size();
}
```

- `create an empty priority queue`
- `insert a key into the priority queue`
- `return and remove a largest key`
- `is the priority queue empty?`
- `return a largest key`
- `number of entries in the priority queue`

**Note.** Duplicate keys allowed; `delMax()` picks any maximum key.

**Warmup client.** Reverse sort sequence 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)
Priority queue: elementary implementation

Unordered list. Store keys in a linked list.

Ordered array. Store keys in an array in ascending order.

<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 XA</td>
<td>A E P X</td>
<td>A E P X</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>5</td>
<td>P E XA M</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>4</td>
<td>P E MA</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 MA 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 MAP L</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>7</td>
<td>P E MAP L E</td>
<td>A E E L M P P</td>
<td>A E E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>E EM A P L</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}, respectively, 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}
Priority queue: implementations cost summary

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 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>log n</td>
</tr>
</tbody>
</table>

order of growth of running time for priority queue with n items

Solution. “Somewhat-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.

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

**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
Binary heap: representation

Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered tree.
- Keys in nodes.
- Parent’s key no smaller than children’s keys.

Array representation.
- Indices start at 1.
- Take nodes in level order.
- Don’t need explicit links!
Consider the key at index $k$ in a binary heap. What is index of its parent?

A. $k/2 - 1$
B. $k/2$
C. $k/2 + 1$
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 \( k/2 \).
- Children of key at index \( k \) are at indices \( 2\times k \) and \( 2\times k + 1 \).
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

- T
- P
- N
  - E
  - I
- H
- R
- O
- A

T | P | R | N | H | O | A | E | I | G
**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 $1 + \log n$ 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.
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[] 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 t = a[i]; a[i] = a[j]; a[j] = t; }
}

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 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
**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.
**Goal.** Delete a random key from a binary heap in logarithmic time.
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.

```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];
    }
}
```

**Immutable in Java.** String, Integer, Double, Color, File, ...

**Mutable in Java.** StringBuilder, Stack, URL, arrays, ...

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
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.
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
In the worst case, how many compares to \texttt{INSERT} and \texttt{DELETE-MAX} in a \(d\)-way heap as function of \(n\) 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>

† amortized

sweet spot: $d = 4$

why impossible?

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

- API and 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;
    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.

```
keys in arbitrary order
```

```
build max heap (in place)
```

```
sorted result (in place)
```
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 R T 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
**Heapsort: heap construction**

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

**Key property.** After $\text{sink}(a, k, n)$ completes, tree rooted at $k$ is a heap.

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

Starting point (arbitrary order)

Result (heap-ordered)
Heapsort: sortdown

Second pass.

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

```
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 */
    }
}

https://algs4.cs.princeton.edu/24pq/Heap.java.html
```

Heapsort: Java implementation

but make static (and pass arguments)

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

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>S O R T E X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>S O R T L X A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>S O X T L R A M P L E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S T X P L R A M O E E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X T S P L R A M O E E</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>T P S O L R A M E E X</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>S P R O L E A M E T X</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>R P E O L E A M S T X</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>P O E M L E A R S T X</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>O M E A L E P R S T X</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>M L E A E O P R S T X</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>L E E A M O P R S T X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>E A E L M O P R S T X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A E E L M O P R S T X</td>
</tr>
</tbody>
</table>

*initial values*

*heap-ordered*

*sorted result*

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 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
Introsort

Goal. As fast as quicksort in practice; $\log n$ 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><strong>selection</strong></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><strong>insertion</strong></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><strong>merge</strong></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><strong>timsort</strong></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><strong>quick</strong></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><strong>heap</strong></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><strong>?</strong></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>