Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
† Heapsort
† API
† Elementary implementations
† Binary heaps
† Heapsort
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.

<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></td>
<td>Q</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></td>
<td>X</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></td>
<td>P</td>
</tr>
</tbody>
</table>
## Priority queue API

### Requirement

Generic items are `Comparable`. The `MaxPQ` class must be `Comparable` (bounded type parameter).

```java
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>MaxPQ(Key[] a)</code></td>
<td>create a priority queue with given keys</td>
</tr>
<tr>
<td><code>void insert(Key v)</code></td>
<td>insert a key into the priority queue</td>
</tr>
<tr>
<td><code>Key delMax()</code></td>
<td>return and remove the 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 the largest key</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>number of entries in the priority queue</td>
</tr>
</tbody>
</table>
Priority queue applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Computational number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

**Generalizes:** stack, queue, randomized queue.
Challenge. Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store $N$ items.

Priority queue client example

```
% more tinyBatch.txt
Turing       6/17/1990   644.08
vonNeumann   3/26/2002  4121.85
Dijkstra     8/22/2007  2678.40
vonNeumann   1/11/1999  4409.74
Dijkstra     11/18/1995 837.42
Hoare        5/10/1993  3229.27
vonNeumann   2/12/1994  4732.35
Hoare        8/18/1992  4381.21
Turing       1/11/2002  66.10
Thompson     2/27/2000  4747.08
vonNeumann   2/12/1994  4732.35
vonNeumann   1/11/1999  4409.74
Hoare        8/18/1992  4381.21
vonNeumann   3/26/2002  4121.85
```

```
% java TopM 5 < tinyBatch.txt
Thompson     2/27/2000  4747.08
vonNeumann   2/12/1994  4732.35
vonNeumann   1/11/1999  4409.74
Hoare        8/18/1992  4381.21
vonNeumann   3/26/2002  4121.85
```
Challenge. Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

```java
MinPQ<Transaction> pq = new MinPQ<Transaction>();
while (StdIn.hasNextLine())
{
    String line = StdIn.readLine();
    Transaction item = new Transaction(line);
    pq.insert(item);
    if (pq.size() > M)
        pq.delMin();
}
```

use a min-oriented pq

Transaction data type is Comparable (ordered by $\leq$)

pq contains largest $M$ items

order of growth of finding the largest $M$ in a stream of $N$ items

<table>
<thead>
<tr>
<th>implementation</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>elementary PQ</td>
<td>$MN$</td>
<td>$M$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$N \log M$</td>
<td>$M$</td>
</tr>
<tr>
<td>best in theory</td>
<td>$N$</td>
<td>$M$</td>
</tr>
</tbody>
</table>
Priority Queues and Heapsort

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
## Priority queue: unordered and ordered array implementation

A sequence of operations on a priority queue

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

**Operation**: action performed on the priority queue

**Argument**: element added to the priority queue

**Return Value**: element removed from the priority queue (if applicable)

**Size**: current size of the priority queue

**Contents**: order of elements in the priority queue

**Unordered**: order of elements not explicitly specified

**Ordered**: order of elements specified explicitly
Priority queue: unordered array implementation

```java
public class UnorderedMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq; // pq[i] = ith element on pq
    private int N; // number of elements on pq

    public UnorderedMaxPQ(int capacity)
    {  pq = (Key[]) new Comparable[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void insert(Key x)
    {  pq[N++] = x;  }

    public Key delMax()
    {  int max = 0;
        for (int i = 1; i < N; i++)
            if (less(max, i)) max = i;
        exch(max, N-1);
        return pq[--N];
    }
}
```

- no generic array creation
- less() and exch() similar to sorting methods
- null out entry to prevent loitering
Priority queue elementary implementations

**Challenge.** Implement **all** operations efficiently.

---

**Order-of-growth of running time for priority queue with N items**

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del 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>
Heapsort

API

Elementary implementations

Binary heaps

Heapsort
Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced, except for bottom level.

Property. Height of complete tree with $N$ nodes is $\lceil \log N \rceil$.

Pf. Height only increases when $N$ is a power of 2.
A complete binary tree in nature
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!
**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 $2k+1$.
Promotion in a heap

**Scenario.** Child's key becomes larger key 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.
Insertion in a heap

**Insert.** Add node at end, then swim it up.

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

```java
public void insert(Key x) {
    pq[++N] = x;
    swim(N);
}
```
### Demotion in a heap

**Scenario.** Parent's key becomes *smaller* than one (or both) of its children's keys.

**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 the maximum in a heap

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

Cost. At most \(2 \log N\) compares.

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

**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 operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

```
insert S

T       P       R
N       H       O
E       I       A
        G       S

violates heap order (swim up)
```

```
T P R N H O A E I G S
11
```
Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

insert $S$
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

```

```

T | S | R | N | P | O | A | E | I | G | H
---|---|---|---|---|---|---|---|---|---|---
2  | 5 | 11| 2 | 5 | 11| 11| 11| 2 | 5 | 11

violates heap order
(swap up)
Binary heap operations

**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 operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

remove the maximum
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.
Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

Binary heap operations
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

---

Remove the maximum

```
S H R N P O A E I G T
1 2
```

violates heap order (sink down)
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

```
S   P   R   N   H   O   A   E   I   G   T
1   2   5
```
Binary heap operations

**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 operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

remove the maximum

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

```
1
```

```
E I G
```

```
N H
```

```
P
```

```
S
```

```
R
```

```
O A
```

```
1
```
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**remove the maximum**

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

```
1
```
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

remove the maximum

![Binary heap diagram]

- **Insert:** Add node at end, then swim it up.
- **Remove the maximum:** Exchange root with node at end, then sink it down.
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

---

**remove the maximum**

![Binary heap diagram](image)

---

<table>
<thead>
<tr>
<th>R</th>
<th>P</th>
<th>G</th>
<th>N</th>
<th>H</th>
<th>O</th>
<th>A</th>
<th>E</th>
<th>I</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

---

**remove the maximum**

---

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 operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

```
R
P
N
E
I
H
S
```

10 violates heap order (swim up)
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

```
R
  /  \
P    O
  / \
N    G
 /  \
E    S
   /  \  
  I   H
```

Violates heap order (swim up)
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

[Diagram of a binary heap with nodes labeled R, S, O, N, P, G, A, E, I, H. Node S is added to the heap, violating the heap order, and is then swam up to maintain the heap property.]
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

- Initial heap:
  - `S` (1) at the root
  - `R` (2) as the parent of `N` (5) and `P` (10)
  - `N` (5) is above `I` (10) and `E` (2)
  - `P` (10) is above `H` (5)

- After inserting `S`:
  - The root is now `S` (1), which violates the heap order.
  - To fix this, `S` (1) must be "swum up" to the root, causing the node with value 10 to sink down.

- The heap after insertion:
  - `S` (1) at the root
  - `R` (2) as the parent of `N` (5) and `P` (10)
  - `N` (5) is above `I` (10) and `E` (2)
  - `P` (10) is above `H` (5)

- The heap order is now restored.

- The heap values:
  - `S`: 1
  - `R`: 2
  - `O`: 5
  - `N`: 10
  - `P`: 5
  - `G`: 10
  - `A`: 1
  - `E`: 2
  - `I`: 10
  - `H`: 1
**Binary heap operations**

**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: 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;
    }
}
```
Priority queues implementation cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del 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>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† amortized

Why impossible?
Binary heap considerations

Immutability of keys.
• Assumption: client does not change keys while they're on the PQ.
• Best practice: use immutable keys.

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 with sink() and swim() [stay tuned]
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. String, Integer, Double, Color, Vector, Transaction, Point2D.
Mutable. StringBuilder, Stack, Counter, Java array.
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.
• Safer in presence of hostile code.
• Simplifies concurrent programming.
• 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)
Priority Queues and Heapsort

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
Heapsort

Basic plan for in-place sort.
- Create max-heap with all $N$ keys.
- Repeatedly remove the maximum key.
Starting point. Array in arbitrary order.

we assume array entries are indexed 1 to N
Heap construction. Build max heap using bottom-up method.
Heapsort

Heap construction. Build max heap using bottom-up method.

sink 5
Heap construction. Build max heap using bottom-up method.

sink 5
Heap construction. Build max heap using bottom-up method.

sink 5
Heap construction. Build max heap using bottom-up method.

sink 4
Heap construction. Build max heap using bottom-up method.

sink 4
Heap construction. Build max heap using bottom-up method.

sink 3
Heapsort

Heap construction. Build max heap using bottom-up method.

sink 3
Heap construction. Build max heap using bottom-up method.

sink 3
Heapsort

Heap construction. Build max heap using bottom-up method.

sink 2
Heap construction. Build max heap using bottom-up method.

sink 2
Heap construction. Build max heap using bottom-up method.

sink 2
Heap construction. Build max heap using bottom-up method.

sink 2

7-node heap

S T X P L R A M O E E
Heap construction. Build max heap using bottom-up method.
Heap construction. Build max heap using bottom-up method.
Heap construction. Build max heap using bottom-up method.

end of construction phase
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 11
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

交换 1 和 10

交换 1 和 10
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 10
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1

```
          S
         /\  \
        P   E
      /   /   \   \  
     O   L   R   A
   /   /     /     /   
  M   E     T     X
```

```
<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>E</th>
<th>O</th>
<th>L</th>
<th>R</th>
<th>A</th>
<th>M</th>
<th>E</th>
<th>T</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 9
Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 8
Sortdown. Repeatedly delete the largest remaining item.
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

![Diagram of a heap](image)

- **sink 1**

  - A
    - O
      - M
    - L
    - E
  - E

<table>
<thead>
<tr>
<th>A</th>
<th>O</th>
<th>E</th>
<th>M</th>
<th>L</th>
<th>E</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 6
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 6

![Tree diagram]

E M E A L O P R S T X

1 6
Sortdown. Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
**Heapsort**

*Sortdown.* Repeatedly delete the largest remaining item.

*exchange 1 and 5*
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 5
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1

![Heap diagram](image-url)
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 4
**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 4
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Sortdown. Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

echange 1 and 3
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Sortdown. Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 2
Sortdown. Repeatedly delete the largest remaining item.
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

**end of sortdown phase**
Ending point. Array in sorted order.
Heapsort: heap construction

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

```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.

while (N > 1)
{
  exch(a, 1, N--);
  sink(a, 1, N);
}
public class Heap
{
    public static void sort(Comparable[] pq)
    {
        int N = pq.length;
        for (int k = N/2; k >= 1; k--)
            sink(pq, k, N);
        while (N > 1)
        {
            exch(pq, 1, N);
            sink(pq, 1, --N);
        }
    }

    private static void sink(Comparable[] pq, int k, int N)
    {
    }

    private static boolean less(Comparable[] pq, int i, int j)
    {
    }

    private static void exch(Comparable[] pq, int i, int j)
    {
    }
}

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

<table>
<thead>
<tr>
<th>initial values</th>
<th>heap-ordered</th>
<th>sorted result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N k</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11 5</td>
<td>11 4</td>
<td>11 3</td>
</tr>
<tr>
<td>11 2</td>
<td>11 1</td>
<td>11 1</td>
</tr>
<tr>
<td>10 1</td>
<td>9 1</td>
<td>8 1</td>
</tr>
<tr>
<td>7 1</td>
<td>6 1</td>
<td>5 1</td>
</tr>
<tr>
<td>4 1</td>
<td>3 1</td>
<td>2 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

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

http://www.sorting-algorithms.com/heap-sort
Heapsort: mathematical analysis

**Proposition.** Heap construction uses fewer than $2N$ compares and exchanges.

**Proposition.** Heapsort uses at most $2N \lg N$ compares and exchanges.

**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 memory.
- Not stable.
## Sorting algorithms: summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>x</td>
<td></td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>$N$</td>
<td>use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td></td>
<td>?</td>
<td>?</td>
<td>$N$</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td></td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td></td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N$</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td>x</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ guarantee, stable</td>
</tr>
<tr>
<td>heap</td>
<td>x</td>
<td></td>
<td>$2N \lg N$</td>
<td>$2N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ guarantee, in-place</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>