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.

Priority queue API

Requirement. Generic items are Comparable.

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P Q E X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>A M</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>L E</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>E</td>
</tr>
<tr>
<td>remove max</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Key must be Comparable

(public class MaxPQ<Key extends Comparable<Key>>
MaxPQ() create an empty priority queue
MaxPQ(Key[] a) create a priority queue with given keys
void insert(Key v) insert a key into the priority queue
Key delMax() return and remove the largest key
boolean isEmpty() is the priority queue empty?
Key max() return the largest key
int size() number of entries in the priority queue)
Priority queue applications

- Event-driven simulation.
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching.
- 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.

Priority queue client example

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.

```java
% 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
Turing 2/11/1991 2156.86
Hoare 8/12/2003 1025.70
vonNeumann 10/13/1993 2520.97
Dijkstra 9/10/2000 708.95
Turing 10/12/1993 3532.36
Hoare 2/10/2005 4050.20
%

% 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
```

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>$M \ N$</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

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

A sequence of operations on a priority queue

Priority queue elementary implementations

Challenge. Implement all operations efficiently.

<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>
</tbody>
</table>

goal log N log N log N

Priority queue: unordered array implementation

```
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];
  }
  ...
}
```

null out entry to prevent loitering

Priority queues and Heapsort

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

- **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.

Complete tree with $N = 16$ nodes (height = 4)

Binary heap representations

- **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.

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.

Insertion in a heap

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

**Cost.** At most \(1 + \log N\) compares.

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

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.

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.
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.

**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.
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.
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.

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.

- heap ordered
- remove the maximum
- exchange with root

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

<table>
<thead>
<tr>
<th>R</th>
<th>P</th>
<th>O</th>
<th>N</th>
<th>H</th>
<th>G</th>
<th>A</th>
<th>E</th>
<th>I</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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: 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
```

```
insert S
```

```
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.

---

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</td>
<td>d</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>

† 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.

Minimum-oriented priority queue.
- Replace `less()` with `greater()`.
- Implement `greater()`.

Other operations.
- Remove an arbitrary item.
- Change the priority of an item.

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.

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)
Heapsort

Basic plan for in-place sort.
- Create max-heap with all \( N \) keys.
- Repeatedly remove the maximum key.

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

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

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

sink 5

sink 4

sink 5

sink 4
Heapsort

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

sink 3

1. S
   2. T
   3. L
   4. X
   5. A

S O R T L X A M P E E

sink 3

1. S
   2. T
   3. L
   4. X
   5. A

S O X T L R A M P E E

sink 3

1. S
   2. T
   3. L
   4. X
   5. A

S O X T L A M P E E

sink 2

1. S
   2. T
   3. L
   4. X
   5. A

S O X T L R A M P E E
Heapsort

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

sink 2

sink 1
**Heapsort**

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

sink 1

![Heap construction diagram](image1)

**Heapsort**

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

end of construction phase

11-node heap

![Heap construction diagram](image2)

**Heapsort**

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

exchange 1 and 11

![Sortdown diagram](image3)

**Heapsort**

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

exchange 1 and 11

![Sortdown diagram](image4)
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

E  T  S  P  L  R  A  M  O  E  X

1  2  4

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

E  T  S  P  L  R  A  M  O  E  X

1  2  4  9

sink 1

E  T  S  P  L  R  A  M  O  E  X

1  2  4  9

sink 1

E  T  S  P  L  R  A  M  O  E  X

1  2  4  9

sink 1

E  T  S  P  L  R  A  M  O  E  X

1  2  4  9

sink 1

E  T  S  P  L  R  A  M  O  E  X

1  2  4  9

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 10

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

```
SPROLEAMETX
1 3
```

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

```
SPROLEAMETX
1 3 6
```

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

```
SPROLEAMETX
1 9
```
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

POEMLEARSTX
1 2 4

POEMLEARSTX
1 2 4

exchange 1 and 7

POEMLEARSTX
1 7

POEMLEARSTX
1 7
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

\[
\begin{array}{cccccc}
 & & & 1 & \text{A} & \\
 & O & & & E & \\
\end{array}
\]

sink 1

\[
\begin{array}{cccccc}
 & & & 2 & \text{A} & \\
 & O & & & E & \\
\end{array}
\]

sink 1

\[
\begin{array}{cccccc}
 & & & 4 & \text{A} & \\
 & O & & & E & \\
\end{array}
\]

sink 1

\[
\begin{array}{cccccc}
 & & & 1 & \text{A} & \\
 & O & & & E & \\
\end{array}
\]

sink 1

\[
\begin{array}{cccccc}
 & & & 2 & \text{A} & \\
 & O & & & E & \\
\end{array}
\]

sink 1

\[
\begin{array}{cccccc}
 & & & 4 & \text{A} & \\
 & O & & & E & \\
\end{array}
\]

sink 1

\[
\begin{array}{cccccc}
 & & & 1 & \text{A} & \\
 & O & & & E & \\
\end{array}
\]
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

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.

sink 1

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

1

L
E
1

E

A
M
O
P
R
S
T
X

E
L
E
A
M
O
P
R
S
T
X

1

2

L

E

A
M
O
P
R
S
T
X

1
2

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

1

4

L

E

E

A
M
O
P
R
S
T
X

L
E
E
A
M
O
P
R
S
T
X

1
4
Heapsort

Sort down. Repeatedly delete the largest remaining item.

exchange 1 and 4

sink 1

sink 1
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2

A

E

1

2

L M O P

R S T X

E A E L M O P R S T X

1 2

end of sortdown phase
Heapsort

Ending point. Array in sorted order.

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

A   E   L   M   O   P   R   S   T   X

1   2   3   4   5   6   7   8   9   10   11

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: heap construction

First pass. Build heap using bottom-up method.

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

Heapsort: Java implementation

```
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)
    { /* as before */
    }

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

    private static void exch(Comparable[] pq, int i, int j)
    { /* as before */
        but convert from
        1-based indexing to
        0-base indexing
    }
```
Heapsort: trace

a[1]

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Initial values: SORT EXAMP LE

Sorted result: A E E L M O P R S T X

Heap-ordered: X T S P L R A M O E E

Heapsort trace (array contents just after each sink)

Heapsort animation

50 random items

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>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td>N exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>N</td>
<td>N</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>N</td>
<td></td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td>N</td>
<td>2 N ln N</td>
<td>N lg N</td>
<td></td>
<td>(N \log N) probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>N</td>
<td>2 N ln N</td>
<td>N</td>
<td></td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td></td>
<td>(N \log N) guarantee, stable</td>
</tr>
<tr>
<td>heap</td>
<td>x</td>
<td>2 N lg N</td>
<td>2 N lg N</td>
<td>N lg N</td>
<td></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>