Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
TODAY

- Heapsort
- API
  - Elementary implementations
  - Binary heaps
  - Heapsort
Priority queue

- A stack is first in, last out
- A queue is first in, first out
- A priority queue is least-first-out
  - The “smallest” element is the first one removed (You could also define a largest-first-out priority queue)
  - The definition of “smallest” is up to the programmer (for example, you might define it by implementing Comparator or Comparable)
  - If there are several “smallest” elements, the implementer must decide which to remove first
    - Remove any “smallest” element (don’t care which)
    - Remove the first one added
Priority queue

• Here is one possible ADT:

  PriorityQueue(): a constructor
  void add(Comparable o): inserts o into the priority queue
  Comparable removeLeast(): removes and returns the least element
  Comparable getLeast(): returns (but does not remove) the least element
  boolean isEmpty(): returns true iff empty
  int size(): returns the number of elements
  void clear(): discards all elements
Evaluating implementations

• When we choose a data structure, it is important to look at usage patterns.

• If we load an array once and do thousands of searches on it, we want to make searching fast—so we would probably sort the array.

• If we load a huge array and expect to do only a few searches, we probably don’t want to spend time sorting the array.

• For almost all uses of a queue (including a priority queue), we eventually remove everything that we add.

• Hence, when we analyze a priority queue, neither “add” nor “remove” is more important—we need to look at the timing for “add + remove”.
Array implementations

• A priority queue could be implemented as an *unsorted* array (with a count of elements)
  - Adding an element would take $O(\ )$ time (why?)
  - Removing an element would take $O(\ )$ time (why?)
  - Hence, adding *and* removing an element takes $O(\ )$ time
  - This is an inefficient representation

• A priority queue could be implemented as a *sorted* array (again, with a count of elements)
  - Adding an element would take $O(\ )$ time (why?)
  - Removing an element would take $O(\ )$ time (why?)
  - Hence, adding *and* removing an element takes $O(\ )$ time
  - Again, this is inefficient
Array implementations

- A priority queue could be implemented as an *unsorted* array (with a count of elements)
  - Adding an element would take $O(1)$ time (why?)
  - Removing an element would take $O(n)$ time (why?)
  - Hence, adding *and* removing an element takes $O(n)$ time
  - This is an inefficient representation

- A priority queue could be implemented as a *sorted* array (again, with a count of elements)
  - Adding an element would take $O(n)$ time (why?)
  - Removing an element would take $O(1)$ time (why?)
  - Hence, adding *and* removing an element takes $O(n)$ time
  - Again, this is inefficient
Linked list implementations

• A priority queue could be implemented as an *unsorted* linked list
  - Adding an element would take \( \mathcal{O}(\; ) \) time (why?)
  - Removing an element would take \( \mathcal{O}(\; ) \) time (why?)

• A priority queue could be implemented as a *sorted* linked list
  - Adding an element would take \( \mathcal{O}(\; ) \) time (why?)
  - Removing an element would take \( \mathcal{O}(\; ) \) time (why?)

• As with array representations, adding *and* removing an element takes 
  \( \mathcal{O}(\; ) \) time
  - Again, these are inefficient implementations
Linked list implementations

• A priority queue could be implemented as an *unsorted* linked list
  - Adding an element would take $O(1)$ time (why?)
  - Removing an element would take $O(n)$ time (why?)

• A priority queue could be implemented as a *sorted* linked list
  - Adding an element would take $O(n)$ time (why?)
  - Removing an element would take $O(n)$ time (why?)

• As with array representations, adding *and* removing an element takes $O(n)$ time
  - Again, these are inefficient implementations
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.

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

Key must be Comparable (bounded type parameter)
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.

```
% 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();
}
```

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

Use a min-oriented pq

PQ client example implementation time space

<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>$M \times N$</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

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

<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>
Priority Queues and Heapsort

- 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 $\lceil \lg N \rceil$.

Pf. Height only increases when $N$ is a power of 2.
A complete binary tree in nature

Hyphaene Compressa - Doum Palm

© Shlomit Pinter
**Heap**

*a heap is a specialised tree-based data structure that satisfies the heap property.

**Heap Property:**

*min-heap property:* the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

*max-heap property:* the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.
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.
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);
}
```
**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

T P R N H O A E I G

T --- P --- N --- H
     /   |   |
    /    |   |
   /     |   |
  E I G  O A

S ← add to heap
**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**
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**
- **N**
- **E**
- **I**
- **G**
- **P**
- **H**
- **R**
- **O**
- **A**

The node labeled `S` violates heap order and needs to be swum 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
```

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

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

![Binary heap diagram](image-url)
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

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.

---

**remove the maximum**

---

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

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.

*heap ordered*
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)

Exchange with root

---

<table>
<thead>
<tr>
<th>G</th>
<th>P</th>
<th>R</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</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**

```
GBPRNHOUAEIS
```

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.

**remove the maximum**

```
R P G N H O A E I S
1 3
```

```
R
|
+---P---+
|       |
|       +---N---E
|           |
|           +---I
|
H
|
+---O---+
|       |
|       +---G---A
|           S
```

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.

---

**remove the maximum**

---

![Binary heap diagram](image)
### 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**

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

S ← add to heap
**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:**
  - Node `S` is inserted at the end.
  - Node `S` violates the heap order (swim up).
  - The heap order is maintained after the operation.

- **Row:**
  - The value `10` is highlighted to denote the insertion point.

- **Note:** The operation is performed on a binary heap, ensuring the heap property is preserved after each operation.
**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**
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 diagram]

The diagram shows a binary heap with nodes labeled S, R, O, N, P, G, A, E, I, H. The root node S has a value of 10, which violates the heap order and requires a swim up operation. The values in the heap are shown in the table below:

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>O</th>
<th>N</th>
<th>P</th>
<th>G</th>
<th>A</th>
<th>E</th>
<th>I</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The heap is adjusted by swimming up the node with value 10, ensuring it satisfies 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.

*heap ordered*
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\textsubscript{d} N</td>
<td>d log\textsubscript{d} N</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>log N\textsuperscript{†}</td>
<td>1</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(\textsuperscript{†}\) 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.
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];
    }

    ...
}
```

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
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.
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

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

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

sink 3
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.

sink 1
Heap construction. Build max heap using bottom-up method.
Heap construction. Build max heap using bottom-up method.

end of construction phase

11-node heap

X

T

P

M

O

E

L

E

S

R

A

X

T

S

P

L

R

A

M

O

E

E
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11
**Sortdown.** Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.

sink 1
**Heapsort**

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

```
sink 1
```

```
T  P  S  E  L  R  A  M  O  E  X
1  2  4
```
**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 10
**Heapsort**

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

*exchange 1 and 10*
**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

S

P

O

M

E

L

T

X

R

E

A

E

M

E

T

X

1 3 6
```

```
S P R O L E A M E T X
1 3 6
```
Sortdown. Repeatedly delete the largest remaining item.
Heapsort

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

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

**exchange 1 and 9**

```
1  E
  P  R
   O   L
    M   T
       9
```

```
9  S
  E
  A
 ```
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.

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

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

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

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

```
sink 1
```

```
P
  O
    M
    4

E
  2
    O
    M

R S T X
```

```
1 2 4
```

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

```
1 2 4
```
Sortdown. Repeatedly delete the largest remaining item.
**Heapsort**

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

exchange 1 and 7

```
[1] P O E M L E A R S T X
```
**Heapsort**

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

**exchange 1 and 7**

```
1   1
A   E
O   L
M   E
R   S   T   X
```

```
    7
  P
  E
    7
```

```
A   O   E   M   L   E   P   R   S   T   X
  1
  7
```
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.

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

exchange 1 and 6
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6
**Heapsort**

*Sortdown.* Repeatedly delete the largest remaining item.
**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 5
**Heapsort**

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

exchange 1 and 5
**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
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.

**exchange 1 and 4**
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4
**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 3
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3
**Sortdown.** Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.

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

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

end of sortdown phase
Heapsort

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.

```java
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```
Heapsort: Java implementation

```java
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

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

**initial values**

Heapsort trace (array contents just after each sink)

**heap-ordered**

**sorted result**
Heapsort animation

50 random items

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

**Proposition.** Heap construction uses fewer than \(2^N\) compares and exchanges.

**Proposition.** Heapsort uses at most \(2N \log N\) 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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>