2.4 Priority Queues

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation
2.4 PRIORITY QUEUES

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation
Collections

A **collection** is a data type that stores a group of items.

<table>
<thead>
<tr>
<th>data type</th>
<th>core operations</th>
<th>data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td><strong>Push, Pop</strong></td>
<td><strong>linked list, resizing array</strong></td>
</tr>
<tr>
<td>queue</td>
<td><strong>Enqueue, Dequeue</strong></td>
<td><strong>linked list, resizing array</strong></td>
</tr>
<tr>
<td>priority queue</td>
<td><strong>Insert, Delete-Max</strong></td>
<td><strong>binary heap</strong></td>
</tr>
<tr>
<td>symbol table</td>
<td><strong>Put, Get, Delete</strong></td>
<td><strong>binary search tree, hash table</strong></td>
</tr>
<tr>
<td>set</td>
<td><strong>Add, Contains, Delete</strong></td>
<td><strong>binary search tree, hash table</strong></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.

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

**Requirement.** Items are generic; they must also be Comparable.

```java
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 a largest key

boolean isEmpty()
is the priority queue empty?

Key max()
return a largest key

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

**Note.** Duplicate keys allowed; `delMax()` picks any maximum key.
Priority queue: applications

- Event-driven simulation. [ customers in a line, colliding particles ]
- Numerical computation. [ reducing roundoff error ]
- Discrete optimization. [ bin packing, scheduling ]
- Artificial intelligence. [ A* search ]
- Computer networks. [ web cache ]
- Operating systems. [ load balancing, interrupt handling ]
- Data compression. [ Huffman codes ]
- Graph searching. [ Dijkstra's algorithm, Prim's algorithm ]
- Number theory. [ sum of powers ]
- Spam filtering. [ Bayesian spam filter ]
- Statistics. [ online median in data stream ]
Challenge. Find the largest $M$ items in a stream of $N$ items, where $N \gg M$.
- Fraud detection: isolate $\$$ transactions.
- NSA monitoring: flag most suspicious documents.

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

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

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

use a min-oriented pq

pq now contains largest $M$ items

N huge, M large
Priority queue: client example

**Challenge.** Find the largest $M$ items in a stream of $N$ items, where $N \gg M$.

<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>$M \cdot N$</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>

order of growth of finding the largest $M$ in a stream of $N$ items
## 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>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 X A 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 M A</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 M A P</td>
<td>A E M P P</td>
<td>A E M P P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>6</td>
<td>P E M A P 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 M A P 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 M A P L E</td>
<td>A E E L M P</td>
<td>A E E L M P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue
Priority queue: implementations cost summary

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>

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
Complete 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 binary tree with \( N \) nodes is \( \lfloor \lg N \rfloor \).

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

Hyphaene Compressa - Doum Palm

© Shlomit Pinter
Binary heap: representation

**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 $2k+1$. 

---

**Heap representations**
Binary heap demo

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

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

**heap ordered**

```
T
/   \
/     \
P      R
|      |
N      O
|      |
E      A
|      |
I      G
|      |
H

T | P | R | N | H | O | A | E | I | G
```
Binary heap demo

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

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

**insert S**

```
T
 /  \
|   |
P  /\  /
|  |  |
N  H  R
 /  \  /
E  I  O
 /  \
G  A
```

S add to heap
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.

**insert S**

---

**Diagram:**

- Insert: Add node S at the end, then swim upward.
- Maximum removal: Exchange the root with the node at the end, then sink downward.

---

**Heap Structure:**

- **T** (root)
- **P**
  - **N**
    - **E**
    - **I**
  - **H**
    - **G**
    - **S** (11)
- **R**
  - **O**
  - **A**

---

**Heap Order Violation:**

Node S violates heap order (swim up).
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.

**insert S**
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.

**insert S**

```
T  S  R  N  P  O  A  E  I  G  H
2  5  11
```
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
**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.
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.

remove the maximum
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.

**remove the maximum**

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

exchange with root
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.

---

**remove the maximum**

```
 violations heap order
(sink down)
```

---

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

```
   O
 R   A
```

---

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

### remove the maximum

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

- **Node H** violates heap order (sink down)
**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.

- remove the maximum

```
remove the maximum
```

Violates heap order (sink down)

```
Violates heap order (sink down)
```
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

```
S
 / \  
P   H
 /   |
N    R
 /    |
E    O
    /  |
   I    A
```

```
S P R N H O A E I G
```
**Binary heap demo**

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

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

### remove the maximum

```
remove the maximum

S  P  R  N  H  O  A  E  I  G
```
Binary heap demo

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

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

---

remove the maximum

---

![Binary heap diagram](image)

**Exchange with root**

---

<table>
<thead>
<tr>
<th>S</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>G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

remove the maximum

```
G  P  R  N  H  O  A  E  I  S
1          10
```

Exchange with root
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.

---

**remove the maximum**

```
  P
 /|
N H
 /|
E I
```

---

Violates heap order (sink down)
**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.

**remove the maximum**

![Binary heap diagram](image)
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.

---

remove the maximum

---

![Binary heap diagram](image-url)
**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**
**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.

*insert S*
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.

**insert S**

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

10 violates heap order
(swim up)

---

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

10
**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.

---

**insert S**

```
R  P  O  N  S  G  A  E  I  H
5  10
```

![Diagram of a binary heap with nodes E, I, H, S, O, G, A, R. The node S is the maximum, and S is added to the heap, causing a violation of the heap order.](image)

---

The node S is added to the heap, causing a violation of the heap order. To fix this, the node S is swum up, moving it and any other nodes necessary to maintain the heap order.
**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.

**insert S**

![Binary heap diagram]

- Insert `S` into the heap.
- The root `R` (value 2) violates the heap order.
- E (value 10) moves up to replace `S` (swim up).

- The heap order is restored.

---

<table>
<thead>
<tr>
<th>R</th>
<th>S</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>2</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
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.

**insert S**

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

Violates heap order (swim up)

<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>10</td>
</tr>
</tbody>
</table>
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
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

```
S   R   O   N   P   G   A   E   I   H
```
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;
    }
}
```
Binary heap: insertion

**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);
}
```
**Binary heap: demotion**

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

**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 $2 \log N$ 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[] 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 queue: implementations 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>
</tbody>
</table>

Order-of-growth of running time for priority queue with $N$ items
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
## Priority queue: 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>log₃ N</td>
<td>d log₃ N</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>log N †</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

order-of-growth of running time for priority queue with N items

sweet spot: $d = 4$

why impossible?
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.

leads to log N amortized time per op (how to make worst case?)

can implement efficiently with sink() and swim() [ stay tuned for Prim/Dijkstra ]
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.
- 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)
2.4 PRIORITY QUEUES

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation
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)
Heap sort demo

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

we assume array entries are indexed 1 to N

array in arbitrary order

```
1
  2
  3
  4 5
  6

M P L E

S R X A

SORT
1 2 3 4 5 6 7 8 9 10 11
```
Heap construction. Build max heap using bottom-up method.
Heapsort demo

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

sink 5

![Heap diagram with nodes labeled S, O, R, T, E, X, A, M, P, L, E and the value 5 at certain positions.](image)
Heapsort demo

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

sink 5
Heapsort demo

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

sink 5

```
S
 /  \
O   R
 /     /
T     X
 /  \
M   E
 /  \
E   E
```

3-node heap
Heapsort demo

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

sink 4

Heapsort demo

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

sink 4

3-node heap
Heapsort demo

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

sink 3
Heapsort demo

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

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

sink 3
Heapsort demo

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

sink 2
Heapsort demo

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

sink 2
Heapsort demo

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
Heapsort demo

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

sink 1
Heapsort demo

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

sink 1
Heapsort demo

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

end of construction phase
Heapsort demo

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

**exchange 1 and 11**
Heapsort demo

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

**exchange 1 and 11**
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort demo

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

```plaintext
sink 1
```

```
<table>
<thead>
<tr>
<th>T</th>
<th>P</th>
<th>S</th>
<th>E</th>
<th>L</th>
<th>R</th>
<th>A</th>
<th>M</th>
<th>O</th>
<th>E</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.
Heapsort demo

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

**exchange 1 and 10**
Heapsort demo

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

exchange 1 and 10
Heapsort demo

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

---

**sink 1**

```
  E  
 /   
P   
|   |   
O   L  
|   |   
M   E  
|   |
T   X  
  
```
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.
Heapsort demo

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 9
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9
Heapsort demo

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

**sink 1**
Heapsort demo

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 8
**Heapsort demo**

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

**exchange 1 and 8**

```
M   P   E   O   L   E   A   R   S   T   X
1   8
```
Heapsort demo

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

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

sink 1

```
     P
    /  \
   M    E
  / \
 O  L  E
```

R S T X
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

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

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7
Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort demo

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

sink 1

![Heap diagram]

<table>
<thead>
<tr>
<th>O</th>
<th>A</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>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Heapsort demo

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

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

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6
Heapsort demo

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

exchange 1 and 6
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

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

exchange 1 and 5
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

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

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

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

exchange 1 and 4
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

```
        A
       /\  
      E   E
    /\    /\  
   4 L  M  O  P
  /\    /\  /\  
 R S   T  X
```
Sortdown. Repeatedly delete the largest remaining item.

sink 1
**Heapsort demo**

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

**sink 1**
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.
Heapsort demo

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

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

exchange 1 and 3
Heapsort demo

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

```plaintext
sink 1

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

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.
Heapsort demo

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

end of sortdown phase
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

array in sorted order

A
E
L

M
O
P

R
S
T
X

<table>
<thead>
<tr>
<th>A</th>
<th>E</th>
<th>E</th>
<th>L</th>
<th>M</th>
<th>O</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
**Heapsort: heap construction**

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

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

![Heap construction diagram](image)
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 */
    }
}
```

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>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>11</td>
<td>5</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>L</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>L</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>S</td>
<td>O</td>
<td>X</td>
<td>T</td>
<td>L</td>
<td>R</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>P</td>
<td>L</td>
<td>R</td>
<td>A</td>
<td>M</td>
<td>O</td>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X</td>
<td>T</td>
<td>S</td>
<td>P</td>
<td>L</td>
<td>R</td>
<td>A</td>
<td>M</td>
<td>O</td>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>T</td>
<td>P</td>
<td>S</td>
<td>O</td>
<td>L</td>
<td>R</td>
<td>A</td>
<td>M</td>
<td>E</td>
<td>E</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>S</td>
<td>P</td>
<td>R</td>
<td>O</td>
<td>L</td>
<td>E</td>
<td>A</td>
<td>M</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>R</td>
<td>P</td>
<td>E</td>
<td>O</td>
<td>L</td>
<td>E</td>
<td>A</td>
<td>M</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>P</td>
<td>O</td>
<td>E</td>
<td>M</td>
<td>L</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>O</td>
<td>M</td>
<td>E</td>
<td>A</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>M</td>
<td>L</td>
<td>E</td>
<td>A</td>
<td>E</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>A</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>E</td>
<td>A</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>E</td>
<td>A</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Heapsort trace (array contents just after each sink)
Heapsort: mathematical analysis

**Proposition.** Heap construction uses \( \leq 2N \) compares and \( \leq N \) exchanges.

**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
## Sorting algorithms: summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td>$\frac{1}{2}N^2$</td>
<td>$\frac{1}{2}N^2$</td>
<td>$\frac{1}{2}N^2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔️ ✔️</td>
<td>$N$</td>
<td>$\frac{1}{4}N^2$</td>
<td>$\frac{1}{2}N^2$</td>
<td>use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td>$N \log_3 N$</td>
<td>?</td>
<td>$cN^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</td>
<td>$\frac{1}{2}N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td>$N \log N$</td>
<td>$2N \log N$</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>$N$</td>
<td>$2N \log N$</td>
<td>$\frac{1}{2}N^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>heap</td>
<td>✔️</td>
<td>$3N$</td>
<td>$2N \log N$</td>
<td>$2N \log N$</td>
<td>$N \log N$ guarantee; in-place</td>
</tr>
<tr>
<td>?</td>
<td>✔️ ✔️</td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>