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
Event-driven simulation
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>E</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>M</td>
<td>A</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>P</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.** Generic items are *Comparable*.

```java
public class MaxPQ<Key extends Comparable<Key>>
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxPQ()</td>
<td>create an empty priority queue</td>
</tr>
<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
```

sort key
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 $$)

pq contains largest $M$ items

use a min-oriented pq

The 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>$M \cdot N$</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>
Priority Queues and Heapsort

- Heapsort
  - API
  - Elementary implementations
  - Binary heaps
  - Heapsort
  - Event-driven simulation
## 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></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>2</td>
<td>P E</td>
<td>E P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>3</td>
<td>P E X</td>
<td>E P X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>4</td>
<td>P E X</td>
<td>A E P X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>5</td>
<td>P E X</td>
<td>A E M P X</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>4</td>
<td>P E M</td>
<td>A E M P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>5</td>
<td>P E M</td>
<td>A E M P P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>6</td>
<td>P E M</td>
<td>A E L M P P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>7</td>
<td>P E M</td>
<td>A E E L M P P</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>E M A</td>
<td>A E E L M P</td>
<td></td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue
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
- null out entry to prevent loitering
- less() and exch() similar to sorting methods
**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>
Heapsort
API
Elementary implementations
Binary heaps
Heapsort
Event-driven simulation
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 $\lfloor \lg N \rfloor$.

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

Hyphaene Compressa - Doum Palm

© Shlomit Pinter
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
defined 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 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.

`insert S`

 violates heap order
(swim up)
Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

insert S

The operation to insert S violates the heap order and needs to be.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**

The diagram shows a binary heap with nodes labeled T, S, R, N, P, O, A, E, I, G, and H. The node S is inserted at the end of the heap, which violates the heap order. The node S is then swum up to its correct position in the 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.

heap ordered

```
heap ordered

T
S
N E I
P G
R O

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

```
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.

**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.

![Diagram of a binary heap]

**remove the maximum**

1. 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]

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

**Violates heap order** (sink down)

<table>
<thead>
<tr>
<th>S</th>
<th>H</th>
<th>R</th>
<th>N</th>
<th>P</th>
<th>O</th>
<th>A</th>
<th>E</th>
<th>I</th>
<th>G</th>
<th>T</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>
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)
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
```

![Binary heap diagram]

```
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.

**remove the maximum**

![Binary heap diagram]

- **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**

![Binary heap diagram]

The heap 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**

- **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

[Diagram of a binary heap with nodes labeled R, O, N, H, P, G, A, I, E, S. Node R is the root, and node G is marked with a red arrow indicating it violates heap order (sink down). The heap order is shown in the bottom row with values 1, 3, 6.]
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**

![Binary heap diagram with 'S' inserted and violating heap order](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.

**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**

- Insert 10: 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></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.

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_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

**order-of-growth of running time for priority queue with \(N\) items**

**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 \texttt{less()} with \texttt{greater()}.
- Implement \texttt{greater()}.

Other operations.
- Remove an arbitrary item.
- Change the priority of an item.
  
  can implement with \texttt{sink()} and \texttt{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
- Event-driven simulation
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
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
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.
Heap construction. Build max heap using bottom-up method.

sink 2

7-node heap

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

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

sink 1
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
Heapsort

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

*exchange 1 and 11*
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.

sink 1
Heapsort

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

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

```
exchange 1 and 10
```

```
T  P  S  O  L  R  A  M  E  E  X
1   10
```
**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
1  3
```
**Heapsort**

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

sink 1

```
S   P   R   L   E   A   M   E   T   X
1   3   6
```
Heapsort

**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

```
E  P  R  O  L  E  A  M  S  T  X
1   9
```
**Heapsort**

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

sink 1

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

```
M  S  T  X
```

```
P
  O
  |
M
```

```
E
  |
  R
  |
E
  |
A
```

```
E
```

```
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

```
R
P
O
M
S
T
X
R
P
E
O
L
E
A
M
S
T
X
1
8
```
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8
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.
Heapsort

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

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

exchange 1 and 7
**Heapsort**

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

sink 1
**Heapsort**

*Sortdown.* Repeatedly delete the largest remaining item.

```
sink 1
```

```
M     L     E
O---A---E
  |  |  |
  R S T X
```

```
<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**

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

```
sink 1
```

```
R   S   T   X
```

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

```
1   2   4
```
Heapsort

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

Sortdown. Repeatedly delete the largest remaining item.

sink 1

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

```
R   S   T   X
```

```
M  L  E  A  E  O  P  R  S  T  X
1  2  5
```
**Heapsort**

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

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

**exchange 1 and 5**

```
E L E A M O P R S T X
  1  5
```
Sortdown. Repeatedly delete the largest remaining item.

sink 1
**Heapsort**

**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 4
Heapsort

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

exchange 1 and 4

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

```
4 1
L 4
```
**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.

exchange 1 and 3
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3
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 2

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

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

exchange 1 and 2

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

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

```plaintext
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>(a[i])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{initial values})</td>
<td></td>
<td>SORT EXAMPLE</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>SORTLXAMPLEE</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORTLXAMPLEE</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>SORTLXAMPLEE</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>SORTLXAMPLEE</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>SOTLXAMPLEE</td>
</tr>
<tr>
<td>(\text{heap-ordered})</td>
<td></td>
<td>XTLSAMPLEE</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>TPSOLRAMEX</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>SPROLEAMETX</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>REProleAMSTX</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>POEMLEARSETX</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>OMEALEPSTX</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>MLEAEOPRSTX</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>LEAMOPRSTX</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>EAELMOPRSTX</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>EAELMOPRSTX</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>EAELMOPRSTX</td>
</tr>
<tr>
<td>(\text{sorted result})</td>
<td></td>
<td>AEELMOPSTX</td>
</tr>
</tbody>
</table>

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></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>
Heapsort

API

Elementary implementations

Binary heaps

Heapsort

Event-driven simulation
Molecular dynamics simulation of hard discs

**Goal.** Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision.
Molecular dynamics simulation of hard discs

**Goal.** Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision.

**Hard disc model.**
- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.

**Significance.** Relates macroscopic observables to microscopic dynamics.
- Einstein: explain Brownian motion of pollen grains.
Warmup: bouncing balls

Time-driven simulation. $N$ bouncing balls in the unit square.

```java
public class BouncingBalls {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        Ball[] balls = new Ball[N];
        for (int i = 0; i < N; i++)
            balls[i] = new Ball();
        while(true)
            {
                StdDraw.clear();
                for (int i = 0; i < N; i++)
                    {
                        balls[i].move(0.5);
                        balls[i].draw();
                    }
                StdDraw.show(50);
            }
}
```
Warmup: bouncing balls

public class Ball
{
    private double rx, ry;       // position
    private double vx, vy;       // velocity
    private final double radius; // radius
    public Ball()
    { /* initialize position and velocity */ }

    public void move(double dt)
    {
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }

    public void draw()
    { StdDraw.filledCircle(rx, ry, radius); }
}

Missing. Check for balls colliding with each other.

• Physics problems: when? what effect?
• CS problems: which object does the check? too many checks?
Time-driven simulation

- Discretize time in quanta of size $dt$.
- Update the position of each particle after every $dt$ units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.
Main drawbacks.

- ~ $N^2/2$ overlap checks per time quantum.
- Simulation is too slow if $dt$ is very small.
- May miss collisions if $dt$ is too large.
  (if colliding particles fail to overlap when we are looking)
Event-driven simulation

Change state only when something happens.
• Between collisions, particles move in straight-line trajectories.
• Focus only on times when collisions occur.
• Maintain PQ of collision events, prioritized by time.
• Remove the min = get next collision.

Collision prediction. Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.

prediction (at time t)
particles hit unless one passes intersection point before the other arrives

resolution (at time t + dt)
velocities of both particles change after collision
Collision prediction and resolution.

- Particle of radius $s$ at position $(r_x, r_y)$.
- Particle moving in unit box with velocity $(v_x, v_y)$.
- Will it collide with a vertical wall? If so, when?

Prediction (at time $t$)

$\Delta t = \text{time to hit wall} = \frac{\text{distance/velocity}}{v_x} = \frac{(1 - s - r_x)}{v_x}$

Resolution (at time $t + \Delta t$)

velocity after collision $= (-v_x, v_y)$
position after collision $= (1 - s, r_y + v_y \Delta t)$

Predicting and resolving a particle-wall collision
Collision prediction.

- Particle $i$: radius $s_i$, position $(rx_i, ry_i)$, velocity $(vx_i, vy_i)$.
- Particle $j$: radius $s_j$, position $(rx_j, ry_j)$, velocity $(vx_j, vy_j)$.
- Will particles $i$ and $j$ collide? If so, when?
Collision prediction.

- Particle $i$: radius $s_i$, position $(r_{xi}, r_{yi})$, velocity $(v_{xi}, v_{yi})$.
- Particle $j$: radius $s_j$, position $(r_{xj}, r_{yj})$, velocity $(v_{xj}, v_{yj})$.
- Will particles $i$ and $j$ collide? If so, when?

\[
\Delta t = \begin{cases} 
\infty & \text{if } \Delta v \cdot \Delta r \geq 0 \\
\infty & \text{if } d < 0 \\
- \frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise}
\end{cases}
\]

\[
d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v) \left( \Delta r \cdot \Delta r - \sigma^2 \right) \quad \sigma = \sigma_i + \sigma_j
\]

\[
\Delta v = (\Delta v_x, \Delta v_y) = (v_{xi} - v_{xj}, v_{yi} - v_{yj})
\]
\[
\Delta r = (\Delta r_x, \Delta r_y) = (r_{xi} - r_{xj}, r_{yi} - r_{yj})
\]

Important note: This is high-school physics, so we won't be testing you on it!
Collision resolution. When two particles collide, how does velocity change?

\[
\begin{align*}
    v_{x_i}' &= v_{x_i} + \frac{J_x}{m_i} \\
    v_{y_i}' &= v_{y_i} + \frac{J_y}{m_i} \\
    v_{x_j}' &= v_{x_j} - \frac{J_x}{m_j} \\
    v_{y_j}' &= v_{y_j} - \frac{J_y}{m_j}
\end{align*}
\]

Newton's second law (momentum form)

\[
J_x = \frac{J \Delta r_x}{\sigma}, \quad J_y = \frac{J \Delta r_y}{\sigma}, \quad J = \frac{2m_i m_j (\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}
\]

Impulse due to normal force
(conservation of energy, conservation of momentum)

Important note: This is high-school physics, so we won't be testing you on it!
public class Particle {
    private double rx, ry;    // position
    private double vx, vy;    // velocity
    private final double radius;  // radius
    private final double mass;  // mass
    private int count;         // number of collisions

    public Particle(...) { }

    public void move(double dt) { }
    public void draw()       { }
    public double timeToHit(Particle that) { }
    public double timeToHitVerticalWall() { }
    public double timeToHitHorizontalWall() { }

    public void bounceOff(Particle that)    { }
    public void bounceOffVerticalWall()     { }
    public void bounceOffHorizontalWall()   { }
}
public double timeToHit(Particle that)
{
    if (this == that) return INFINITY;
    double dx = that.rx - this.rx, dy = that.ry - this.ry;
    double dvx = that.vx - this.vx; dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    if( dvdr > 0) return INFINITY;
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = this.radius + that.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY;
    return -(dvdr + Math.sqrt(d)) / dvdv;
}

public void bounceOff(Particle that)
{
    double dx = that.rx - this.rx, dy = that.ry - this.ry;
    double dvx = that.vx - this.vx, dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    double dist = this.radius + that.radius;
    double J = 2 * this.mass * that.mass * dvdr / ((this.mass + that.mass) * dist);
    double Jx = J * dx / dist;
    double Jy = J * dy / dist;
    this.vx += Jx / this.mass;
    this.vy += Jy / this.mass;
    that.vx -= Jx / that.mass;
    that.vy -= Jy / that.mass;
    this.count++; this.count++;
    that.count++; that.count++;
}

Important note: This is high-school physics, so we won’t be testing you on it!
Collision system: event-driven simulation main loop

Initialization.

- Fill PQ with all potential particle-wall collisions.
- Fill PQ with all potential particle-particle collisions.

Main loop.

- Delete the impending event from PQ (min priority = \( t \)).
- If the event has been invalidated, ignore it.
- Advance all particles to time \( t \), on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.
Conventions.

• Neither particle null ⇒ particle-particle collision.
• One particle null ⇒ particle-wall collision.
• Both particles null ⇒ redraw event.

```java
private class Event implements Comparable<Event>
{
    private double time;    // time of event
    private Particle a, b;  // particles involved in event
    private int countA, countB; // collision counts for a and b

    public Event(double t, Particle a, Particle b) { }

    public int compareTo(Event that) { return this.time - that.time; }

    public boolean isValid() { }
}
```
public class CollisionSystem
{
    private MinPQ<Event> pq; // the priority queue
    private double t = 0.0;  // simulation clock time
    private Particle[] particles; // the array of particles

    public CollisionSystem(Particle[] particles) { }

    private void predict(Particle a) {
        if (a == null) return;
        for (int i = 0; i < N; i++)
        {
            double dt = a.timeToHit(particles[i]);
            pq.insert(new Event(t + dt, a, particles[i]));
        }
        pq.insert(new Event(t + a.timeToHitVerticalWall(), a, null));
        pq.insert(new Event(t + a.timeToHitHorizontalWall(), null, a));
    }

    private void redraw() { }

    public void simulate() { /* see next slide */ }
}
public void simulate()
{
    pq = new MinPQ<Event>();
    for(int i = 0; i < N; i++) predict(particles[i]);
    pq.insert(new Event(0, null, null));

    while(!pq.isEmpty())
    {
        Event event = pq.delMin();
        if(!event.isValid()) continue;
        Particle a = event.a;
        Particle b = event.b;

        for(int i = 0; i < N; i++)
            particles[i].move(event.time - t);
        t = event.time;

        if      (a != null && b != null) a.bounceOff(b);
        else if (a != null && b == null) a.bounceOffVerticalWall();
        else if (a == null && b != null) b.bounceOffHorizontalWall();
        else if (a == null && b == null) redraw();
        predict(a);
        predict(b);
    }
}
Particle collision simulation example 1

% java CollisionSystem 100
Particle collision simulation example 2

% java CollisionSystem < billiards.txt
Particle collision simulation example 3

% java CollisionSystem < brownian.txt
Particle collision simulation example 4

% java CollisionSystem < diffusion.txt