**Priority Queues and Heapsort**

Mar. 2, 2017

**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

**TODAY**

› Heapsort
› API
› Elementary implementations
› Binary heaps
› Heapsort

**Acknowledgement:** The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
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 \( O(1) \) 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 \( O(1) \) 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 \( O(1) \) time (why?)

• As with array representations, adding and removing an element takes \( O(n) \) 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.

Priority queue API

Requirement. Generic items are Comparable.
Key must be Comparable (bounded type parameter)

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

Priority queue applications

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

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

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

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

Priority queue client example

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

```
% java TopM 5 < 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
Hoare 5/10/1993  3229.27
```

Priority queue client example

```
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();
}
pq contains largest $M$ items
```

Priority queue: unordered and ordered array implementation

```
operation           arguments          return                   size          contents (unordered)          contents (ordered)
insert P             1 P               P                         P                   ```
insert Q             2 P Q             P Q                       ```
insert E             3 P Q E           E P Q                     ```
remove max Q          2 P E             E P                       ```
insert X             3 P E X           E P X                     ```
insert A             4 P E X A         A E P X                   ```
insert M             5 P E X A M       A E M P X                 ```
remove max X          4 P E M A         A E M P                   ```
insert P             5 P E M A P       A E M P                   ```
insert L             6 P E M A P L     A E L M P P               ```
insert E             7 P E M A P E     A E E L M P P             ```
remove max P          6 E M A P E       A E E L M P                ```
```

A sequence of operations on a priority queue
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];
    }
}
```

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>

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

Binary tree

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

Complete tree. Perfectly balanced, except for bottom level.

Property. Height of complete tree with N nodes is ⌊lg N⌋.
Pf. Height only increases when N is a power of 2.
A complete binary tree in nature

Heap

**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.

Demotion in a heap

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

To eliminate the violation:
- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```java
private void sink(int k)
{
    while (2*k <= N)
    {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

Power struggle. Better subordinate promoted.

Insertion in a heap

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

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

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

Delete the maximum in a heap

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

**Cost.** At most $2 \lg N$ 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

T P R N H O A E I G

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

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

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.

add to heap

S

insert S

T P R N H O A E I G

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.

violates heap order (swim up)

S

T P R N H O A E I G 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.

violates heap order (swim up)

S

T P R N S 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.

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

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

Insert $S$

![Tree diagram showing insertion and heap order violations](image1)

Remove the maximum

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

Remove the maximum

![Tree diagram showing removal and exchange with root](image2)
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.

heap ordered

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

**Binary heap operations**

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

remove the maximum

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

**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 operations**

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

remove the maximum

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

```
1
```

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

```
1
```

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

exchange with root

```
1
```

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

```
1
```

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

exchange with root

```
1
```

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

```
1
```

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

```
1
```

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

```
1
```

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

```
1
```

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

```
1
```

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

```
1
```

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

```
1
```

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

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

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

remove the maximum

---

**Binary heap 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.

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.

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**

![Diagram showing heap operations]

- **insert S**

![Diagram showing 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.

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

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

---

**Binary heap operations**

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

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

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

---

**Binary heap: Java implementation**

```java
public class MaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;
    private int N;

    public MaxPQ(int capacity) {
        pq = new Comparable[capacity+1];
    }

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

    public void insert(Key key) {
        pq[++N] = key; // see previous code
        swim(N);
    }

    public Key delMax() {
        Key root = pq[1];
        pq[1] = pq[N--];
        sink(1);
        return root;
    }

    private void swim(int k) {
        while (less(k, parent(k))) {
            exch(k, parent(k));
            k = parent(k);
        }
    }

    private void sink(int k) {
        while (less(k, child(k, 1))) {
            exch(k, child(k, 1));
            k = child(k, 1);
        }
    }

    private boolean less(int i, int j) {
        return pq[i].compareTo(pq[j]) < 0;
    }

    private int parent(int i) {
        return i / 2;
    }

    private int left(int i) {
        return i * 2 + 1;
    }

    private int right(int i) {
        return i * 2 + 2;
    }
}
```

---

**Priority queues implementation cost summary**

<table>
<thead>
<tr>
<th></th>
<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>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>binary heap</td>
<td>log N</td>
<td>log N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>log_d N</td>
<td>log_d N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>log N</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† Why impossible?

† Amortized
Binary heap considerations

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

Underflow and overflow.
- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

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

Other operations.
- Remove an arbitrary item.
- Change the priority of an item. can implement with sink() and swim() [stay tuned]

Immutability: implementing in Java

Data type. Set of values and operations on those values.

Immutable data type. Can’t change the data type value once created.

```java
public final class Vector {
    private final int N;
    private final double[] data;

    public Vector(double[] data) {
        this.N = data.length;
        this.data = new double[N];
        for (int i = 0; i < N; i++)
            this.data[i] = data[i];
    }
    ...
}
```

Immutable. String, Integer, Double, Color, Vector, Transaction, Point2D.

Mutable. StringBuilder, Stack, Counter, Java array.

Immutability: properties

Data type. Set of values and operations on those values.

Immutable data type. Can’t change the data type value once created.

Advantages.
- Simplifies debugging.
- Safer in presence of hostile code.
- Simplifies concurrent programming.
- Safe to use as key in priority queue or symbol table.

Disadvantage. Must create new object for each data type value.

“Classes should be immutable unless there’s a very good reason to make them mutable... If a class cannot be made immutable, you should still limit its mutability as much as possible.”

— Joshua Bloch (Java architect)
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.

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

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

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

sink 5

```
S O R T L X A M P E E
  5  10
```

sink 5

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

sink 4

```
S O R T L X A M P E E
  4  10
```

sink 4

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

3-node heap
Heapsort

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

sink 3

sink 2

sink 3

sink 2
Heapsort

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

sink 2

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

sink 2

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

sink 2

7-node heap

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

sink 1

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

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

sink 1

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

1 3

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

exchange 1 and 11

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

1 11

end of construction phase 11-node heap

**Heapsort**

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

end of construction phase 11-node heap

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

exchange 1 and 11

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

1 11
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

1. exchange 1 and 10

2. sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

Exchange 1 and 9

sink 1
**Heapsort**

*Sortdown.* Repeatedly delete the largest remaining item.

exchange 1 and 8

sink 1

exchange 1 and 8

sink 1
**Heapsort**

Sortdown. Repeatedly delete the largest remaining item.

sink 1

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

exchange 1 and 7

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

```
1 7
POEMLEARSSTX
```

```
exchange 1 and 7
```

```
R S T X
AOEMLEPRSTX
```

```
1 7
AOEMLEPRSTX
```
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

1 1

A O E M L E P R S T X

sink 1

2 3

O A E M L E P R S T X

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

1 2

O A E M L E P R S T X

sink 1

2 4

O M E A L E P R S T X

sink 1

2 3

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

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

<table>
<thead>
<tr>
<th>O</th>
<th>M</th>
<th>E</th>
<th>A</th>
<th>L</th>
<th>E</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

<table>
<thead>
<tr>
<th>O</th>
<th>M</th>
<th>E</th>
<th>A</th>
<th>L</th>
<th>E</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>E</th>
<th>A</th>
<th>L</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></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>E</th>
<th>A</th>
<th>L</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></td>
<td></td>
<td>2</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

exchange 1 and 5

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

Heapsort

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

Sortdown. Repeatedly delete the largest remaining item.

Exchange 1 and 4

```
E
1
A
4
L
M
O
P
R
S
T
X
```

AEELMOPRSXT

1

4

Sink 1

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

AEELMOPRSXT

1

Exchange 1 and 2

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

EAELMOPRSXT

1

2

Sink 1

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

EAELMOPRSXT

1

2
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2

end of sortdown phase
Heapsort

Ending point. Array in sorted order.

Heapsort: sortdown

Second pass.
• Remove the maximum, one at a time.
• Leave in array, instead of nulling out.

while (N > 1)
  { exch(a, 1, N--);
    sink(a, 1, N);
  }

Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

for (int k = N/2; k >= 1; k--)
  { /* as before */ }

Heapsort: Java implementation

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

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

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

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

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

<table>
<thead>
<tr>
<th>N</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>k</td>
<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>
<td></td>
</tr>
<tr>
<td>a[i]</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Initial values: SORT EXAMPLE

Heapsort (array contents just after each sink)

Heapsort animation

50 random items

Heapsort trace (array contents just after each sink)

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 $2N\log N$ 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>N</td>
<td></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>x</td>
<td></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></td>
<td>$N\lg N$</td>
<td>$N\lg N$</td>
<td>$N\lg N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>