Data Structures and Algorithms

Lesson 7: Use Heaps!

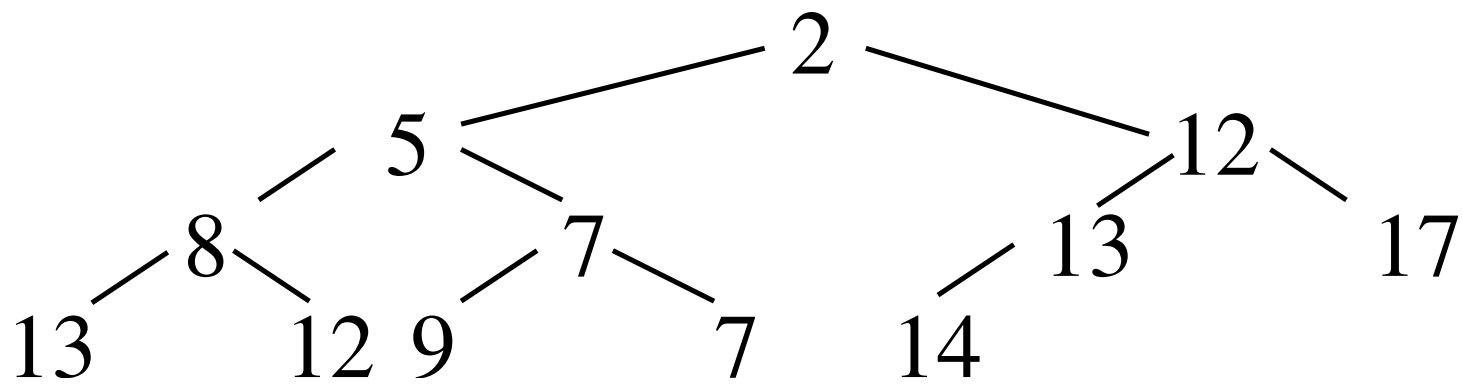
Heaps, Priority queues, Heap Sort

Outline

1. **Heaps**
2. Priority Queues
   - Array Implementation
3. Heap Sort

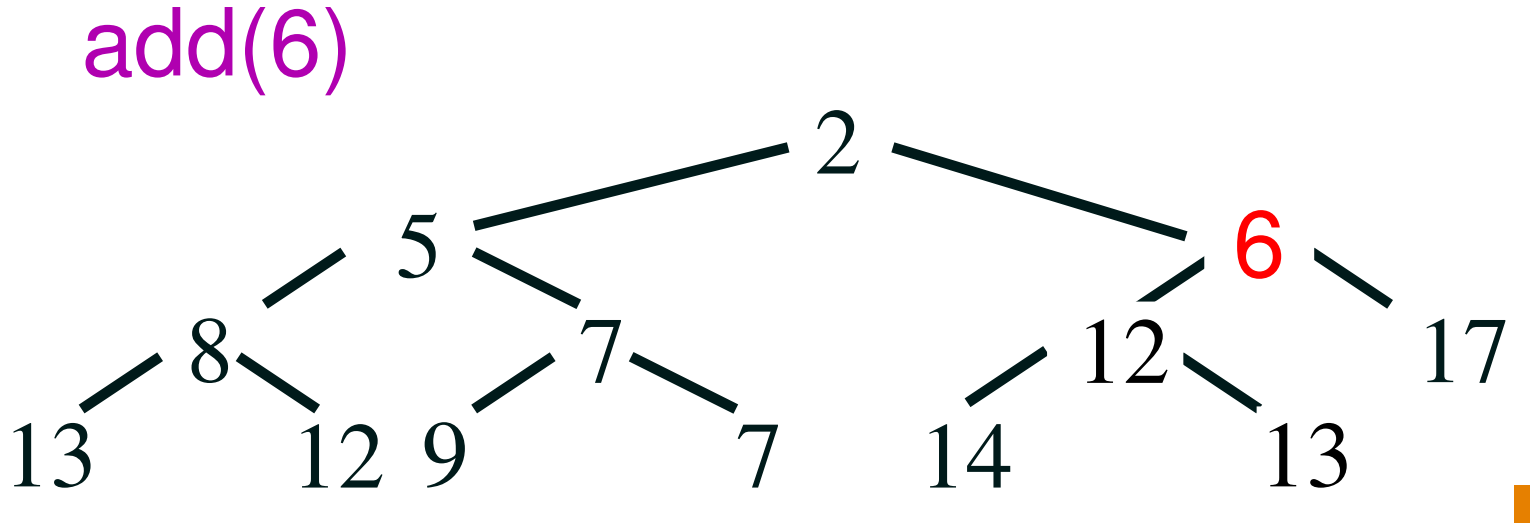
Heaps

- A (min-)heap is a binary tree satisfying two constraints
  - It is a complete tree: every level above the lowest level is fully occupied, and the nodes on the lowest level are all to the left
  - Each child has a value 'greater than or equal to' its parent



Adding to the Heap

- Heaps are easy to maintain
- To add an element to a heap:
  - Add the element to the next available space in the tree
  - Percolate the value up the tree to maintain the correct ordering



Outline

1. Heaps
2. **Priority Queues**
   - Array Implementation
3. Heap Sort

Priority Queues

- One of the prime uses of heaps is to implement a Priority Queue
- A Priority Queue is a queue with priorities
  - we assign a priority to each element we add
  - the element with highest priority (smallest number) is the head of the queue
  - used, for example, to implement "greedy algorithms"

Priority Queue Interface

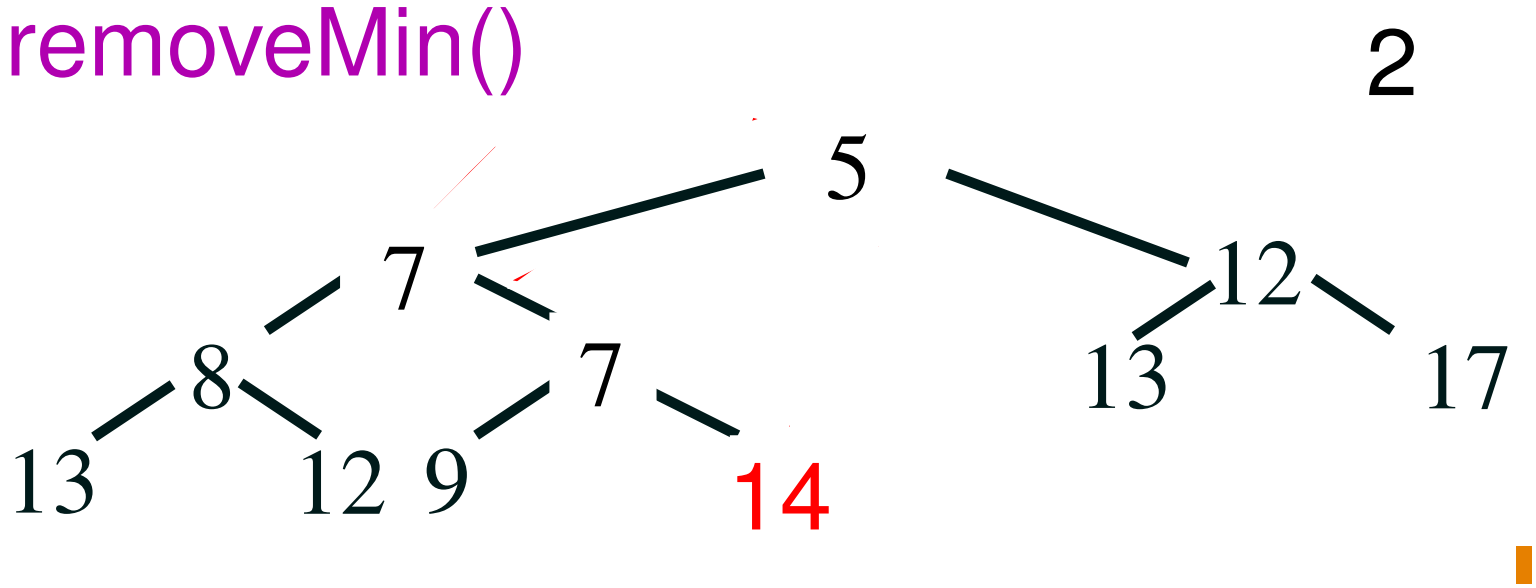
- A simple Priority Queue interface might include

```
interface PQ<T>
{
    int size();                                  // return number of elements
    boolean isEmpty();                           // true if queue is empty,
    void add(T element, int priority);           // add element to queue
    T getMin();                                  // return head of queue
    T removeMin();                               // remove head of queue
}
```

- Java has a PriorityQueue class which extends AbstractQueue and is part of the Java Collection framework

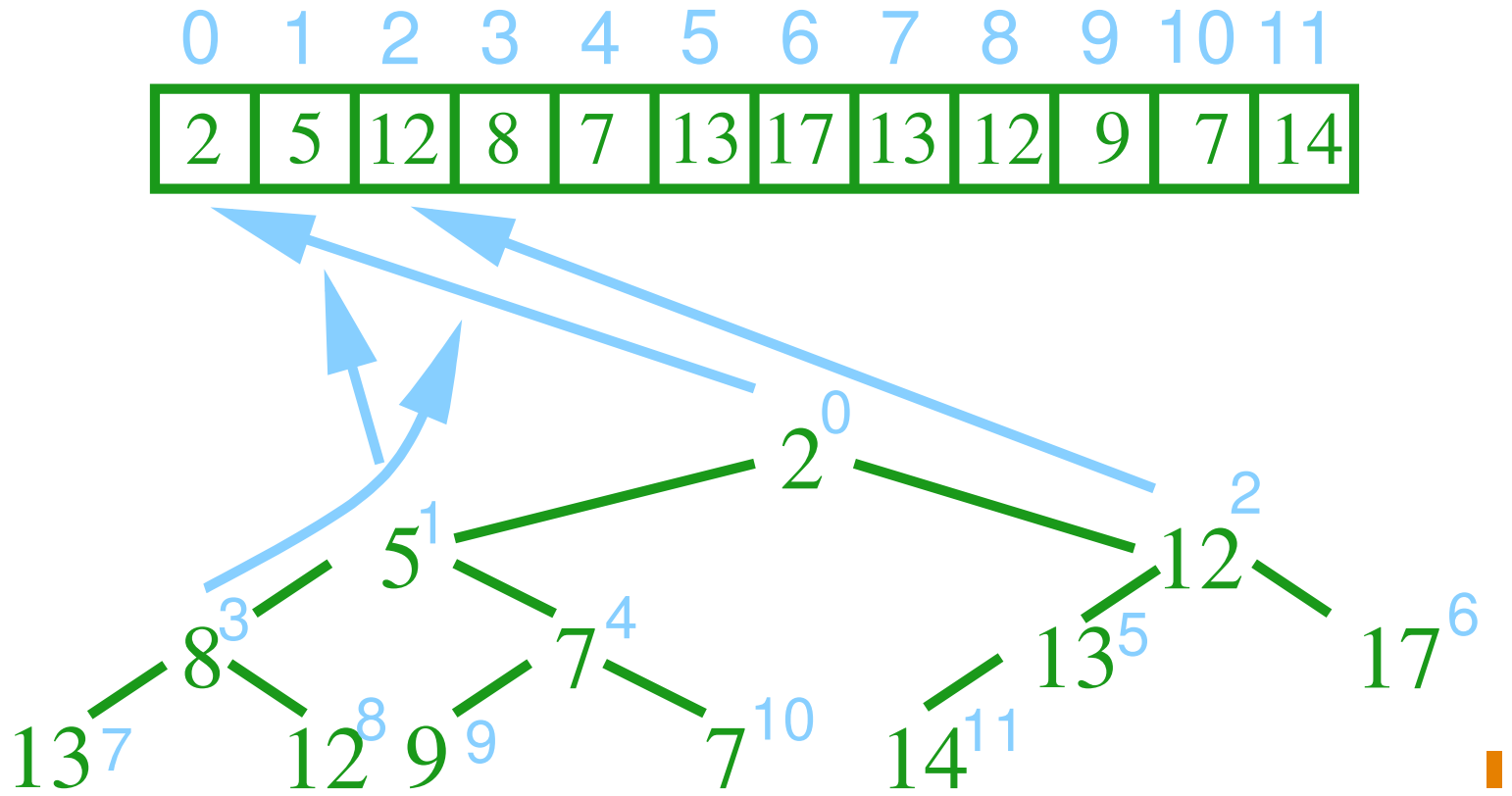
`removeMin`

- The minimum element is the root of the tree
- To remove this element:
  - Pop the root
  - Replace it with the last element in the heap
  - Percolate this element down to the bottom of the heap choosing the minimum child



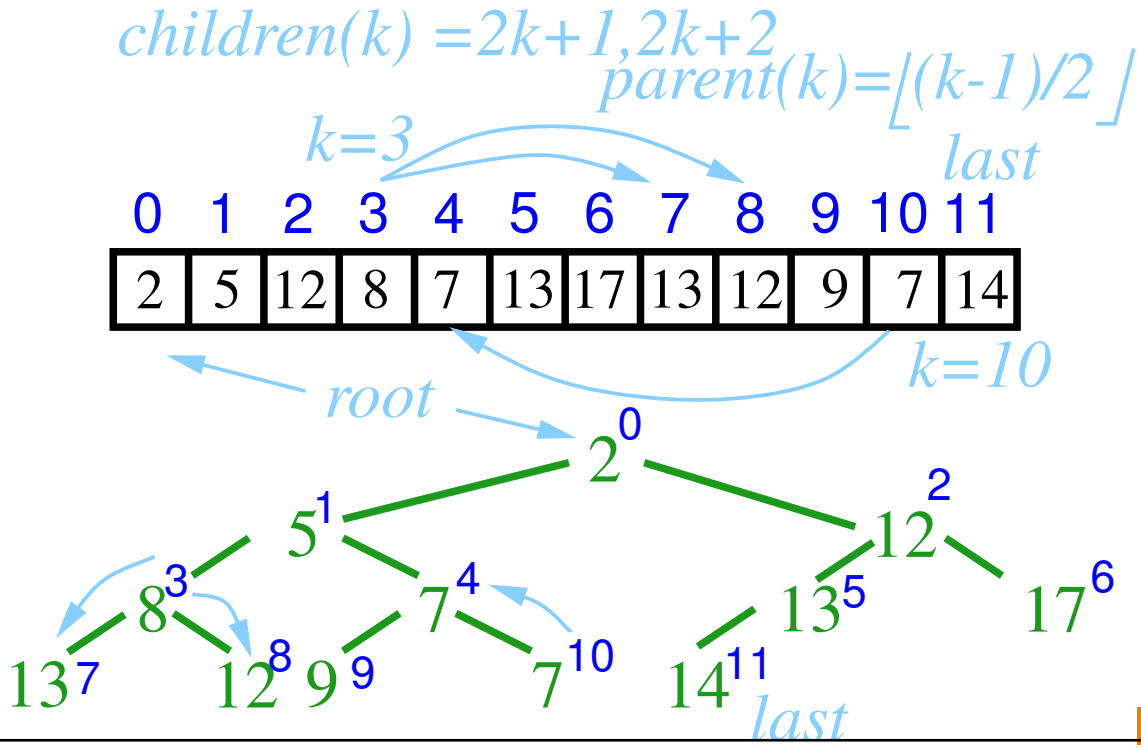
Array Implementation of Heaps

- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!



Navigating a Heap

- To navigate a heap we note that
  - The root of the tree is at array location 0
  - The last element in the heap is at array location `size()-1`
  - The parent of a node k is at array location $\lfloor (k-1)/2 \rfloor$
  - The children of node k are at array locations $2k+1$ and $2k+2$



Code for a Priority Queue

```
import java.util.*;

public class HeapPQ<T> implements PQ<T>
{
    private List<T> list;

    public HeapPQ(int initialCapacity)
    {
        list = new ArrayList<T>(initialCapacity);
    }

    public int size() { return list.size(); }

    public boolean isEmpty() { return list.size()==0; }

    public T getMin() { return list.get(0); }
```

Adding an Element

```
public void add(T element)
{
   list.add(element);
   percolateUp();
}

private void percolateUp()
{
   int child = list.size()-1; //set child to index of new element

   while (child>0) { //not the root
      int parent = (child-1)>>1;  //floor((child-1)/2)
      if (compare(child, parent) >= 0) //child value >= parent value
         break;
      swap(parent,child);
      child = parent; //continue checking for the parent
   }
}
```

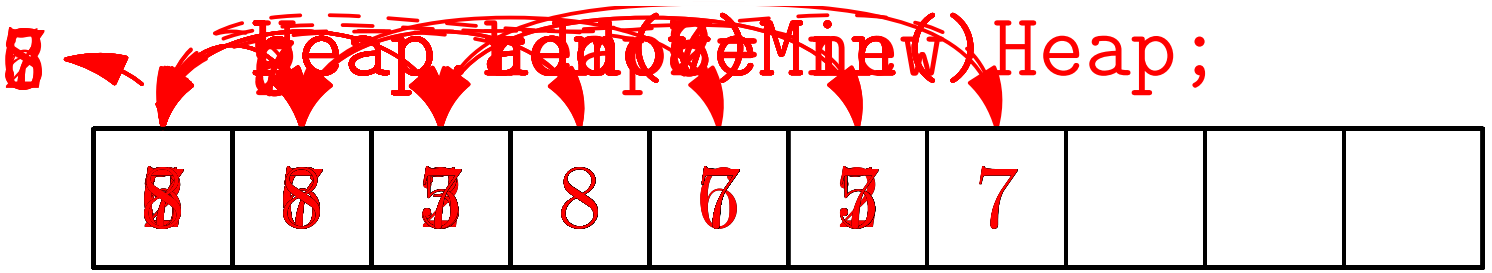
- `compare` and `swap` are trivial helper function

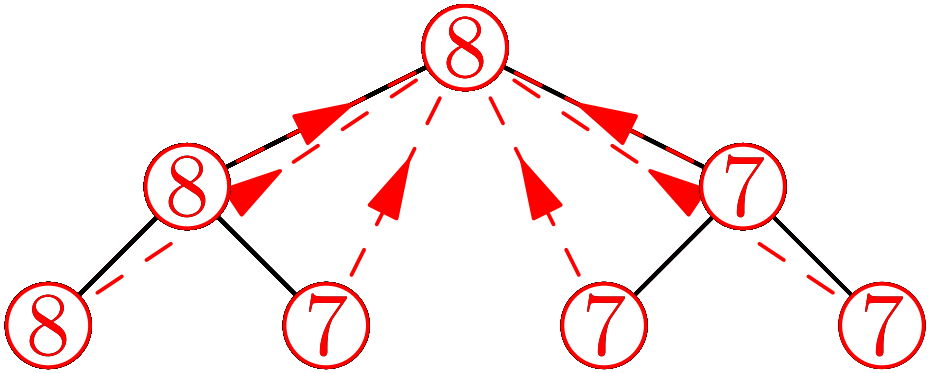
Popping the Top

```
public T removeMin() {
    T minElem = list.get(0); //save minimum value
    list.set(0, list.get(list.size()-1)); //replace root by last element
    list.remove(list.size()-1); //remove last element
    percolateDown(0); //fix heap property
    return minElem;
}

private void percolateDown(int parent) {
    int child = (parent<<1) + 1;  // 2*parent+1
    while (child < list.size()) { //left child exists
        if (child+1 < list.size() && compare(child,child+1) > 0)
            //right child exists and smaller than left
            child++;
        if (compare(child, parent)>=0) //smallest child above parent
            break;
        swap(parent, child);
        parent = child;
        child = (parent<<1) + 1; //continue percolating down
    }
}
```

Heaps in Action





Time Complexity of Heaps

- The two important operations are `add` and `removeMin`
- These work by percolating an element up the tree, respectively by percolating an element down the tree
  - percolating up/down a single level is $\Theta(1)$,
  - the height of the tree is $\Theta(\log(n))$,
  - and so percolating up/down is $\Theta(\log(n))$;
  - the remaining operations in `add`/`removeMin` are $\Theta(1)$
- Thus `add` and `removeMin` are $\Theta(\log(n))$ in the worst case
- Except `add` could also require resizing the array, but the amortised cost of this is low

Back to Priority Queues

- We implemented a priority queue using a heap earlier (`HeapPQ<T>`)
- To make a priority queue we use a `PriorityTask` class for the queue elements:

```
Queue<PriorityTask> pq = new HeapPQ<PriorityTask>();

pq.add(new PriorityTask(stuff, priority));
```

- where

```
class public PriorityTask implements Comparable<PriorityTask> {
  private Stuff stuff;
  private int priority;

  public int compareTo(PriorityTask rhs) {
    return priority-rhs.priority;
  }
⋮
```

Outline

1. Heaps
2. Priority Queues
   - Array Implementation
3. **Heap Sort**

Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

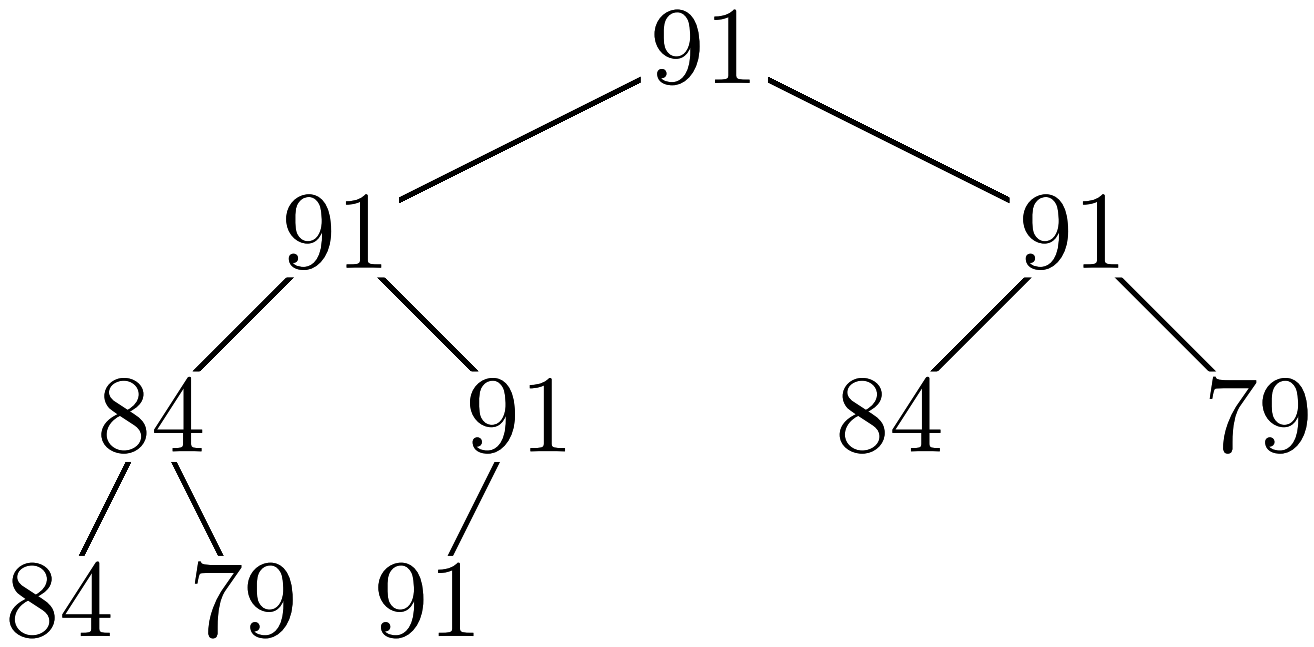
```
public static <T> void sort(List<T> aList)
{
    PQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
        aHeap.add(element);

    aList.clear();
    while(aHeap.size() > 0)
        aList.add(aHeap.removeMin());
}
```

- Note that this is not an in-place sort algorithm – it uses $\Theta(n)$ additional memory!
- The standard Heap Sort algorithm sorts in place.

Example of Heap Sort

19	27	33	39	55	76	78	79	84	91



Worst-Case Complexity of Heap Sort

- the worst-case time complexity is log-linear, i.e. $O(n \log(n))$
  - we have to add n elements and then remove n elements
  - each add/remove is $O(\log(n))$
- This is actually a very efficient algorithm

A Word on the Standard Heap Sort Algorithm (not examinable!)

- Standard Heap Sort (invented by John Williams in 1964) works as follows:
  1. Start with a non-sorted array
  2. Transform this into a max-heap without using any additional storage
     - max-heap grows in size from 1 to the whole array
  3. Order the resulting array by repeatedly removing the maximum from the current heap
     - each time the maximum is removed, it swaps places with the last element in the heap before the heap decreases in size

Standard Heap Sort (not examinable!)

The following implementation of the standard Heap Sort algorithm uses variants of the methods `percolateUp()` and `percolateDown()` that take an additional argument giving the heap size. (This is, in general, different from `list.size()`, both when repeatedly adding to the heap and when repeatedly removing the maximum element from the current heap.)

```
public void Heap Sort(){
  //start with unsorted list and produce a max-heap
  for (i =1; i<list.size(); i++)
     percolateUp(i+1);
  // successively remove maximum element while maintaining heap property
  for (i=list.size()-1; i>0; i--){
    swap(0,i);
    percolateDown(0,i);
  }
}
```

Lessons

- Heaps are a powerful data structure – they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queues have many uses
  - They are used in operating systems
  - They can be used to perform efficient sort
  - They are often used for implementing greedy-type algorithms