Heap Sort

You may have noticed that in a max heap, the largest value can easily be removed from the top of the heap and in the ensuing siftDown() a 'free' slot is made available at what was the end of the heap. The removed value could then be stored in the vacated slot. If this process was repeated until the heap shrank to size 1, we would have a sorted array.

This suggests the basis of a new sorting algorithm, namely *Heap Sort*.

For sorting general purpose arrays, a heap going from array positions 0 to (n-1) is more appropriate than what we used in our Heap class. In this situation:

parent(i) = (i-1)/2, lchild(i) = 2i+1, rchild(i) = 2i+2.

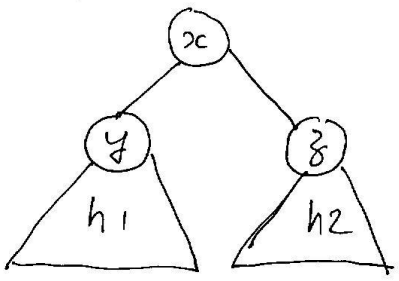
But first of all it is necessary to convert the array into a heap array. We can do this in two ways:

1. A top-down approach whereby we insert array values one at a time into a heap initially consisting of the first array value. This approach can be summarised with:

```
for k = 2 to (n-1)  // array of size n, a[0] to a[n-1]
    siftUp(k, a)
```

The complexity of this loop is $O(n \log_2 n)$

2. A bottom up approach can also be used to do this whereby 2 smaller heaps and a value are combined to give a bigger heap. Consider the diagram below, which shows two small heaps h1 and h2 and a value x. Because y is the biggest value in h1 and z is the biggest in h2, we can simply merge h1, h2 and x into a new heap by a siftDown() starting at x's position.



The smallest heaps are of size 1 and are the leaf nodes of the binary tree. The last tree node with a leaf node for a left child is at position (n-1)/2 where n is the array size. So by iterating the above described procedure starting from the node at (n-1)/2, we gradually construct a heap. It can be shown that constructing a heap in this way from an array takes n steps where n is the array size.

This process can be expressed as:

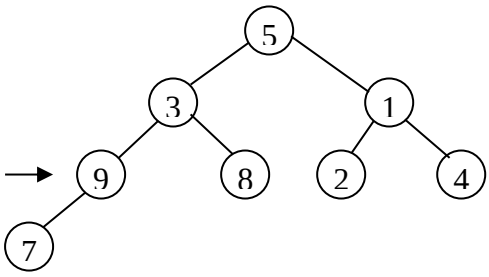
```
for k = (n-1-1)/2 to 0
    siftDown(k, a, n)
```

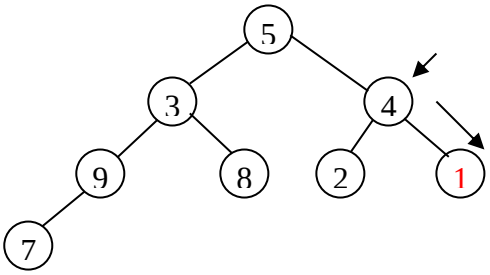
It can be shown that constructing a heap in this way from an array takes n steps where n is the array size. Loop complexity is $O(n)$. For this reason we use this approach.

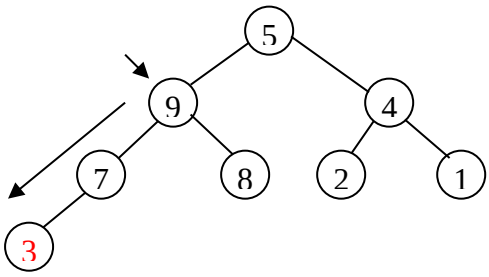
Example

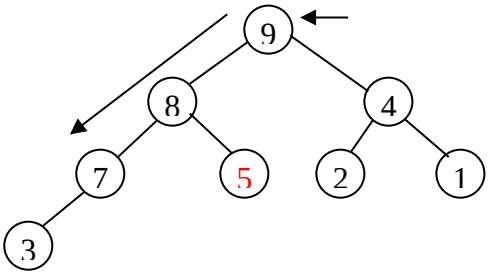
For example, consider the array below which is also shown as a binary tree.
Here n = 8. The last node with a child is the one containing 9 at position (8-1)/2 = 3.
The next node to be sifted down is at position 2. It has the value 1. Then positions 1 and 0 are processed.

5	3	1	9	8	2	4	7









9	8	4	7	5	2	1	3

Using the heap for sorting

Now that a heap has been obtained by rearranging a[ ], we begin to sort it by removing the highest priority value from the heap and placing it at the end of the array.

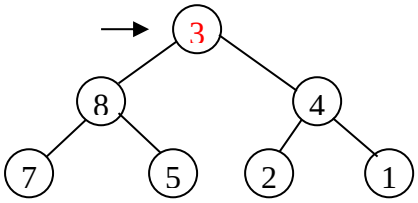
So the largest value is removed from the heap and the heap resized. This requires a siftDown() so that the smaller 'heap' is still a heap and this operation takes at most $\log_2 n$ steps. The removed value is then placed at the array position which has been freed by the removal. This is repeated with the next largest value being placed in the second last position and so on. This process is repeated $n-1$ times.

We express this algorithm with:

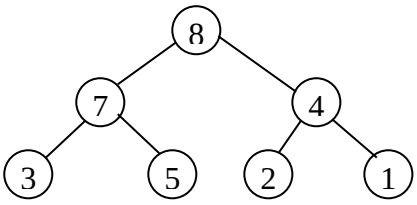
```
for k = n-1 to 1
    v = a[0]            // largest value on heap
    a[0] = a[k]         // a[k] is last value on heap
    siftDown( 0, a, k)  // k is now heap size
    a[k] = v
```

Example continued

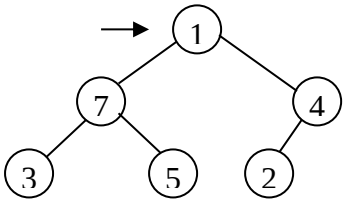
The first 3 iterations of this loop as follows.

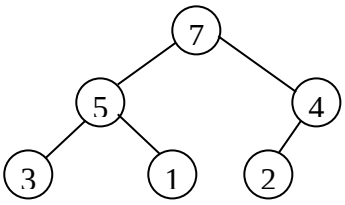


3	8	4	7	5	2	1	9

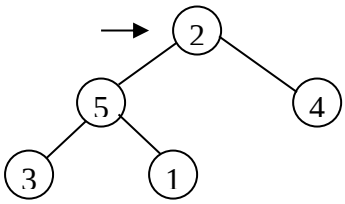


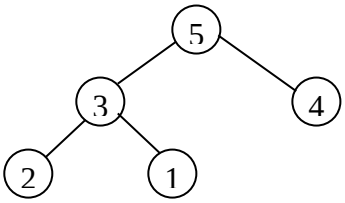
8	7	4	3	5	2	1	9





7	5	4	3	1	2	8	9





5	3	4	2	1	7	8	9

Calculating heap sort complexity

Heap sort requires:

- n steps to form the heap
- and $(n-1)\log_2 n$ steps in the removes/siftDowns.

So the total number of steps is

$$n + (n-1)\log_2 n$$
$$\approx n\log_2 n$$

This is the same as the average cost of quick sort.

Complexity of heap sort = $O(n\log_2 n)$

The average and worst performance of heap sort is $n\log_2 n$ which is also the average performance of quick sort. However, quick sort can be unstable in its performance depending on how well it chooses a pivot. At worst quick sort takes $n^2/2$ steps. So if you are not sure about the nature of the data and want guaranteed performance, heap sort is better.

Algorithm

Finally, the pseudocode fragments are put together to yield

```
// assuming position a[0] does contains data
heapSort( int a[ ], int n )
begin
        for k = n/2 - 1 to 0
                siftDown(k, a, n)

        for k = n-1 to 1
                v = a[0]    // largest value on heap
                a[0] = a[k]
                siftDown( 0, a, k) // heap size is now k
                a[k] = v            //a[k] is no longer in heap
end


// Heap from a[0] to a[N-1], size N
siftDown(int k, int a[ ], int N)
begin
        v = a[k]
        j = 2k + 1  // left child of k since heap begins in a[0]
        while( j ≤ N-1 )  // while left child is within heap
                if( j < N-1 ∧ a[j] < a[j+1])
                    ++j
                if( v ≥ a[j] )
                    break
                a[k] = a[j]
                k = j
                j =2k+1
        a[k] =  v
end
```