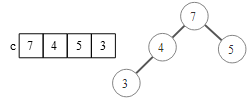
Heap sort

# We assume you understand heaps, both min-heaps and max-heaps. We show you how a max-heap can be used to sort an array of size n in time O(n log n) and space O(1).

# The basic idea

# We illustrate using an array b[0..3] that contains (3, 5, 7, 4). Sorting the array has two steps.

1. **Heapify**: Build a max-heap c[0..3] of the array elements. As you know, this can be done with a loop that inserts b[0], b[1], b[2], and b[3] into the max-heap, bubbling up each value as it is inserted. To the right, we show both the final array c and the tree that it contains.
2. **Poll the heap**: Poll the values from the heap, one by one, and store them in array b starting at the end and working forward. Each poll removes and returns the largest value, which is in c[0], so the values are polled in the order 7, 5, 4, 3. To the right, we show the state after the first value is polled and stored in b[3]. The 7 is in b[3]. The ? in b[0..2] indicates that we don’t care what those values are. The heap is now in c[0..2]. You also see the heap in tree form.

# Analysis of space and time

# Because of the extra array c, this algorithm take space O(n). Below, we show how to eliminate c so that the space is O(1). Step 1 takes time O(n log n). Step 2 does also. So the total time is O(n log n).

# Eliminating array c

# There is no need for array c! Instead, perform all the operations in array b, making this sort an in-place or *insitu* sort. We consider first making the max-heap.

1. **Heapify**: b[0] can already be considered a max-heap of one value. We show this to the right, with heap value 3 boldfaced. Now consider b[1] = 5 to be in the heap and bubble it up. The second diagram to the right shows the state after bubbling b[1] up.

# Next, consider b[2] = 7 to be in the heap and bubble it up, as shown in the first diagram to the right. Finally, do the same for b[3] = 4, the result being the second diagram to the right.

# You see that it is easy to heapify array b: Just bubble up each successive element. The algorithm appears below. It uses method bubbleUp, which appears to the right..

/\*\* Bubble b[k] up so that b[0..k] is a max-heap. <br>

\* Precondition: b[0..k] is a max-heap except that   
 \* b[k] might be > its parent. \*/

**public** **static** **void** bubbleUp(**int**[] b, **int** k) {

**int** p= (k - 1) / 2; // p is the parent of k

// inv: p is the parent of p AND original b[0..k-1]   
 //. is a max-heap except: b[k] might be > b[p].

**while** (k > 0 && b[k] > b[p]) {

Swap b[k] and b[p]

k= p;

p= (k - 1) / 2;

}

}

// Heapify b: Make b into a max-heap

// invariant: b[0..k-1] is a max-heap AND

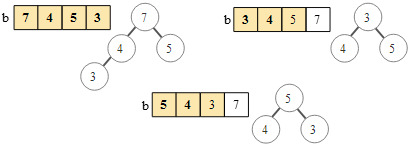
// b is a permutation of its initial value

**for** (**int** k= 1; k < b.length; k= k + 1) {

// Make b[0..k] into a max-heap by   
 // bubbling b[k] up

*bubbleUp*(b, k);

}

1. **Polling the heap**: Suppose b is the max-heap shown in the first diagram to the right, showing both b and its tree form. The first step in polling is to swap b[0] with the last element of the heap, in this case b[3], and to consider that last value to be no longer in the heap. The second diagram to the right shows the state after this step. *Note that the value 7 is now in its final position*.

# The next step in polling is to bubble b[0] down. The result is shown in the third diagram (the lower one). You can see the largest value in the heap, 5, is in the root.

# If the heap is polled again, the 5 will be placed in its final position (see first diagram to the right) and b[0] will be bubbled down (second diagram to the right).

# Polling one more time complete the sorting of b.

# You could probably write the invariant of the polling loop yourself at this point. Here it is.

# Invariant of polling loop: b is a permutation of its initial value AND b[0..k] is a max-heap AND b[k+1..] is sorted AND b[0..k] ≤ b[k+1..]

# With this invariant, we write the second step of heapsort. It uses a method bubbleDown, which appears to the right.

/\*\* Bubble b[0] down to its heap position in b[0..k-1].

\* Precondition: b[0..k-1] is a max-heap except

\* that b[0] may have a larger child \*/

**public** **static** **void** bubbleDown(**int**[] b, **int** k) {

**int** h= 0;

// in: b[0..k-1] is a max-heap except that

// b[h] may be larger than one of its children.

**while** (h < k) {

// c= larger child of h (return if h has no children)

**int** c= 2 \* h + 1; // h's smallest child

**if** (c > k - 1) **return**;

**if** (c < k - 1 && b[c] < b[c + 1]) c= c + 1;

**if** (b[h] >= b[c]) **return**; // b[h] is in the right place

Swap b[n] and b[c];

h= c;

}

}

# // Sort max-heap b[0..]

// invariant: See above

**for** (**int** k= b.length - 1; k > 0; k= k - 1) {

Swap b[0] and b[k];

bubbleDown(b, k);

}