

Applications of Sorting

One reason why sorting is so important is that once a set of items is sorted, many other problems become easy.

Searching

Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.

Speeding up searching is perhaps the most important application of sorting.

Closest pair

Given n numbers, find the pair which are closest to each other.

Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.

Element uniqueness

Given a set of n items, are they all unique or are there any duplicates?

Sort them and do a linear scan to check all adjacent pairs.

This is a special case of closest pair above.

Frequency distribution – Mode

Given a set of n items, which element occurs the largest number of times?

Sort them and do a linear scan to measure the length of all adjacent runs.

Median and Selection

What is the k th largest item in the set?

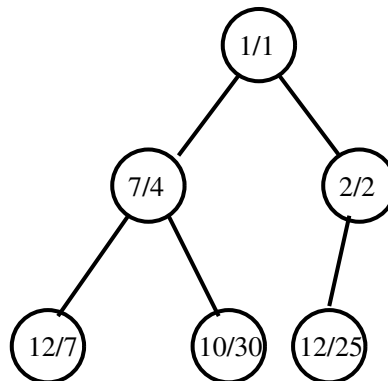
Once the keys are placed in sorted order in an array, the k th largest can be found in constant time by simply looking in the k th position of the array.

Binary Heaps

A *binary heap* is defined to be a binary tree with a key in each node such that:

1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The key in root is \geq all its children, and the left and right subtrees are again binary heaps.

Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.



Constructing Heaps

Heaps can be constructed incrementally, by inserting new elements into the left-most open spot in the array.

If the new element is greater than its parent, swap their positions and recur.

Since at each step, we replace the root of a subtree by a larger one, we preserve the heap order.

Since all but the last level is always filled, the height h of an n element heap is bounded because:

$$\sum_{i=1}^h 2^i = 2^{h+1} - 1 \geq n$$

so $h = \lfloor \lg n \rfloor$.

Doing n such insertions takes $\Theta(n \log n)$, since the last $n/2$ insertions require $O(\log n)$ time each.

Algorithm 1 Max-Heapify

```
1: procedure MAX-HEAPIFY( $A, i$ )  
2:    $l \leftarrow 2i + 1$  ▷ Index of left child  
3:    $r \leftarrow 2i + 2$  ▷ Index of right child  
4:    $n \leftarrow \text{length of } A$  ▷ Size of the array/heap  
5:    $largest \leftarrow i$   
6:   if  $l < n$  and  $A[l] > A[largest]$  then  
7:      $largest \leftarrow l$   
8:   if  $r < n$  and  $A[r] > A[largest]$  then  
9:      $largest \leftarrow r$   
10:  if  $largest \neq i$  then  
11:    swap( $A[i], A[largest]$ )  
12:    MAX-HEAPIFY( $A, largest$ )
```

Heapsort

Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

```
Heapsort(A)
    Build-heap(A)
    for  $i = n$  to 1 do
        swap(A[1], A[i])
         $n = n - 1$ 
        Heapify(A, 1)
```

If we construct our heap from bottom to top using Heapify, we do not have to do anything with the last $n/2$ elements.

With the implicit tree defined by array positions, (i.e. the i th position is the parent of the $2i$ th and $(2i + 1)$ st positions) the leaves start out as heaps.

Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm, named *Heapsort*.