Heap Algorithms

*We assume that we are storing a heap of n items in an array A[], that is indexed from 1 to n. (We ignore index 0). We assume we are building a **max-heap**, where the root node of the tree is the largest item in the heap (and therefore higher-priority items have larger numbers).*

Heap-up (percolate up, sift up, swim): Used to move a specific item at the bottom of a heap into a valid location if it's out of place. Usually called after adding a new item to the heap. *Idea*: If the item is larger than its parent, swap. Continue at parent. Stop when no swap takes place, or when we reach the root.

```
heapup(A[], int k)                       // k is the index of the out-of-place item
  while (k > 1 and A[k] > A[k/2])        // A[k/2] is A[k]'s parent
    swap A[k/2] and A[k]
    k = k/2
```

Heap-down (percolate down, sift down, sink): Used to move the root element (if it's out of place) into a valid location. Usually called after removing the root node from a heap (deleting the highest-priority item). *Idea:* If an item has at least one child that larger, swap that item with the larger of its two children. Continue at the updated child. Stop when no swap takes place, or when the item we're moving doesn't have any children.

```
heapdown(A[], int k)
  while (2*k < A.length)                 // keep going as long as A[k] has at least one child.
    largerChild = 2*k                    // assume left child is larger
    if (largerChild+1 < A.length and A[largerChild] < A[largerChild+1]))
      largerChild++                      // check if right child (2k + 1) is larger
    if A[k] >= A[largerChild]            // if A[k] is not out of place,
      break                              // ...then end the whole loop.
    swap A[k] and A[largerChild]         // otherwise, swap.
    k = largerChild                      // repeat with k as the larger child.
```

Heapsort

- Start with array A[] of unsorted elements in positions 1 through n (ignore position 0).
- Create a heap in place with those elements:
  - Interpret A[1..n] as a heap structure with many violations of the heap property.
  - Repeatedly call heapdown() on each element, starting from position n/2 and progressing backwards to position 1.
  - This creates a heap.
- Repeatedly swap A[1] (max element in heap) with A[n] (last element in heap). This moves the largest element in the heap to its correct spot at the end of the array.
- Decrease the size of the heap by one (to effectively ignore the new item we just added at the end).
- Call heapdown() to repair the heap from the root node, ignoring all elements past the end of the heap. This moves the next-largest item to the root.

```
heapsort(A[])
  n = size of array A                # this first step is sometimes
  for (int k = n/2; k >= 1; k--)     # known as the "heapify"
    heapdown(A, k)                   # algorithm

  while (n > 1)
    swap A[1] and A[n]
    n--  // decrease size of heap
    heapdown(A, 1)   // Sink takes into account new heap size here.
```