Priority Queues

- set S of elements with "keys" (priority) that can be compared

Operations

- `insert(A, x)`
  - add an element `x`
- `max(A)`
  - return the highest priority element
- `extract_max(A)`
  - remove and return the highest priority element

Complete Binary Trees

- every layer before the last is full
- in the last layer, all nodes are as far to the left as possible

Height of a tree: number of *edges* in the longest path (other contexts may use number of nodes)

- height of a complete binary tree is $\log_2(n)$

Max-Heap

- List implementation of priority queue always requires a lot of time for some operation, so we use max-heaps
- complete binary tree
- **max-heap property**: every root node has higher priority than its child nodes
- stored in array that is level-order traversal of tree
  - consider array with indices starting at 1

    - if an element has index i, then its children are stored in indices $2i$ and $2i + 1$
        - (as a result, if an element has index i then its parent has index $\lfloor i/2 \rfloor$)
- `insert(A, x)`
    - algorithm:
        1. add `x` to the leftmost empty place in the bottom layer (i.e. at the end of the list)
        2. if `x` is greater than its parent, swap `x` and its parent (i.e. swap `A[i]` and `A[i/2]`)
        3. repeat step 2 at `x` 's new position, propagating it upwards until `x` is not greater than its parent
    - since the maximum number of iterations of the algorithm is the height of the tree (in the worst case, `x` would need to be propagated to the top of the tree), the algorithm takes $\Theta(\log(n))$ time
- `max(A)`
    - algorithm:
        1. return the first element in the array
    - obviously takes constant time
- `extract_max(A)`
    - algorithm:
        1. store the max
        2. replace the max with the last element (which we will call `x`) in the array, decrement the array size (getting rid of the last element), aka `A[1] = x; A.length -= 1`
        3. if `x` is less than one of its children, swap with that child, if `x` is less than both of its children, swap with the

greater child
    4. repeat step 3 until `x` is not less than either of its children, i.e. the max-heap property has been restored

- similarly to insert, the maximum number of iterations of the algorithm is the height of the tree (since in the worst case, the last element has to be swapped until it is at the bottom of the tree), so the algorithm takes $\Theta(\log(n))$

Heapsort

- algorithm:
    1. make a heap out of the elements of the list ($\Theta(n)$ time)
    2. `extract_max(A)` n times to extract each element - these extractions will be in order of greatest to least element of the list
- each extract_max step takes