Review of Priority queues

- **Priority Queues**
  - A priority queue is a data structure which supports insertion, and removal of elements in order of priority; i.e., the element with the lowest (first) priority is the one to be removed next
  - Some operations in the Java API:
    - `offer` inserts a new item into the queue
    - `poll` removes the item with the first priority
    - `peek` returns, but does not remove, the entry with first priority

- **Example**

  ```java
  public class PQ {
    private static class Entry<K implements Comparable<K>, V>
      implements Comparable<Entry<K,V>> {
      private K key;  // determines priority
      private T val;

      public Entry(K key, V val) {
        this.key = key;
        this.val = val;
      }

      public int compareTo(Entry<K,V> e) {
        return key.compareTo(e.key);
      }
    }

    public static void main(String[ ] args) {
      PriorityQueue<Entry<Integer,String>> pq =
        new PriorityQueue<Entry<Integer,String>>() {
        pq.offer(new Entry<String>(1, "a"));
        pq.offer(new Entry<String>(2, "b"));
        pq.offer(new Entry<String>(0, "c"));
        System.out.println(pq.poll().val);
        System.out.println(pq.poll().val);
        System.out.println(pq.poll().val);
      }
    }
  }
  
  Keys and Values are separate because they aren't necessarily inherently related (though they might be)

- **Implementation using a Heap**
  - A complete binary tree is one in which the depth of all leaves differs by at most 1, and the deeper leaves are as far to the left in the tree as possible (terminology is
ambiguous; sometimes "complete" means that the tree has the most possible nodes for its height
  o A **heap** is a complete binary tree such that:
    - for every node n other than the root, the key stored at n is less than or equal to the key stored at n's parent
  o Alternatively, for every node n other than the root, the key stored at n is greater than or equal to the key stored at n's parent

- **Efficient implementation of a heap**
  o A heap can be implemented as an array in which the $i^{th}$ item in the array (root = 0) has children at positions $2i+1$ and $2i+2$
  o Array implementation can only work if the tree is complete

- **Heaps and complete binary trees**
  o It's easy to determine where to insert a new item into a heap if the heap is a complete binary tree
  o Insert at the next available leaf position; then percolate item up the tree until it's at the right level
  o This is called **swimming**

- **Removal from a heap implemented as a complete binary tree**
  o The entry with the smallest key is always at the root
  o Replace the root with the "last" element in the tree (the rightmost deepest-level leaf); then if need be swap the new root with its smallest child, and so on
  o This is called **sinking**

- **Example**

  (In this example, lower priorities come first)

  offer:

  (10, A)
  (6, B)
  (7, C)
  (7, D)
  (9, E)
  (5, F)
  (8, G)
  (2, H)
  (1, I)
  (3, J)

  offer (10, A)
| (10,A) |     |     |      |      |      |     |     |     |     |
|------------------|

\textbf{offer (6, B)}

| (10,A) | (6,B) |     |      |      |      |     |     |     |     |
|------------------|

Swim

| (6,B) | (10,A) |     |     |     |     |     |     |     |     |
|------------------|

\textbf{offer (7, C)}

| (6,B) | (10,A) | (7,C) |     |     |     |     |     |     |     |
|------------------|

\textbf{offer (7,D)}

| (6,B) | (10,A) | (7,C) | (7,D) |     |     |     |     |     |     |
|------------------|

\textbf{swim}

| (6,B) | (7,D) | (7,C) | (7,C) | (10,A) |     |     |     |     |     |
|------------------|

\textbf{offer (9, E)}

| (6,B) | (7,D) | (7,C) | (7,C) | (10,A) | (9,E) |     |     |     |     |
|------------------|

\textbf{offer (5, F)}

| (6,B) | (7,D) | (7,C) | (10,A) | (9,E) | (9,E) | (5,F) |     |     |     |
|------------------|

\textbf{swim}

| (5,F) | (7,D) | (6,B) | (10,A) | (9,E) | (7,C) | (7,C) |     |     |     |
|------------------|

\textbf{offer (8, G)}

| (5,F) | (7,D) | (6,B) | (10,A) | (9,E) | (7,C) | (8,G) |     |     |     |
|------------------|
offer (2, H)

\[
\begin{array}{llllllllll}
(5, F) & (7, D) & (6, B) & (10, A) & (9, E) & (7, C) & (8, G) & (2, H) & \\
\end{array}
\]

swim

\[
\begin{array}{llllllllll}
(2, H) & (5, F) & (6, B) & (7, D) & (9, E) & (7, C) & (8, G) & (10, A) & \\
\end{array}
\]

offer (1, I)

\[
\begin{array}{llllllllllllll}
(2, H) & (5, F) & (6, B) & (7, D) & (9, E) & (7, C) & (8, G) & (10, A) & (1, I) & \\
\end{array}
\]

swim

\[
\begin{array}{llllllllllllll}
(1, I) & (2, H) & (6, B) & (5, F) & (9, E) & (7, C) & (8, G) & (10, A) & (7, D) & \\
\end{array}
\]

offer (3, J)

\[
\begin{array}{llllllllllllll}
(1, I) & (2, H) & (6, B) & (5, F) & (9, E) & (7, C) & (8, G) & (10, A) & (7, D) & (3, J) & \\
\end{array}
\]

swim

\[
\begin{array}{llllllllllllll}
(1, I) & (2, H) & (6, B) & (5, F) & (3, J) & (7, C) & (8, G) & (10, A) & (7, D) & (9, E) & \\
\end{array}
\]

10 calls to poll:

poll -> I

\[
\begin{array}{llllllllllllll}
(9, E) & (2, H) & (6, B) & (5, F) & (3, J) & (7, C) & (8, G) & (10, A) & (7, D) & \\
\end{array}
\]

sink

\[
\begin{array}{llllllllllllll}
(2, H) & (3, J) & (6, B) & (5, F) & (9, E) & (7, C) & (8, G) & (10, A) & (7, D) & \\
\end{array}
\]

poll -> H

\[
\begin{array}{llllllllllllll}
(7, D) & (3, J) & (6, B) & (5, F) & (9, E) & (7, C) & (8, G) & (10, A) & \\
\end{array}
\]
sink

| (3, J) | (5, F) | (6, B) | (7, D) | (9, E) | (7, C) | (8, G) | (10, A) |

poll -> J

| (10, A) | (5, F) | (6, B) | (7, D) | (9, E) | (7, C) | (8, G) |

sink

| (5, F) | (7, D) | (6, B) | (10, A) | (9, E) | (7, C) | (8, G) |

poll -> F

| (8, G) | (7, D) | (6, B) | (10, A) | (9, E) | (7, C) |

sink

| (6, B) | (7, D) | (7, C) | (10, A) | (9, E) | (8, G) |

e etc.

- Another example

(5, a) (8, b) (3, c) (2, d) (4, e) (9, f) (6, g) (7, h) (1, i) (0, j)

- Complexity of operations using a complete binary tree
  - offer: \( \log(n) \)
  - poll: \( \log(n) \)

- Why not just use a balanced search tree?
  - Same complexity, but heap implementation is more efficient
  - Contiguous storage of arrays
  - Heap formation is only \( \Theta(n) \)
    - What is it for a balanced search tree? – \( n \log n \)

- Bottom-up heap construction
o If all n entries in a heap are given at once, we can construct the heap in $\Theta(n)$ time
o Algorithm
o Example