Priority Queues

- In a priority queue, things get inserted into the queue in order of priority
- Priority queues contain entries = \{keys, values\}

```java
/** Interface for a key-value pair entry **/
public interface Entry {
    public Object key();
    public Object value();
}
```

- No longer FIFO – you can still remove from front but insertion is now priority-based.
  - Elements can have same priority.
- We also need same way to compare keys. Another class Comparator.
- Comparator objects are external to the keys that are to be compared and compare two objects: compare (a, b)
- When the priority queue needs to compare two keys, it uses the comparator it was given to do the comparison.
- A comparator object is any object that has a method that compares any two objects and returns 1 if a has higher priority, 0 if they have the same priority, and 1 if a has lower priority.

In Java, we can use Comparable and Comparator to compare objects.

  - **Comparable**
  
  A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.

```java
public interface Comparable {
    public int compareTo(Object a);
    //returns 1 if object is greater than a,
    //returns 0 if object is equal to a
    //returns -1 if is less than a.
}
```

  - **Comparator**

A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class’s instances. This comparator class must implement the java.util.Comparator interface.

```java
interface Comparator
```
```java
{  
    int compare(Object a, Object b);
}

//returns 1 if a has higher priority,
//returns 0 if both have the same priority
//returns -1 if b has higher priority.
}

• Operations
  o  add(x): add item x
  o  remove: remove the highest priority item (usually the smallest element)
  o  peek: return the highest priority item (without removing it)
  o  size: return the number of items in the priority queue
  o  isEmpty: return whether the priority queue has no items

public interface PriorityQueue {
    public int size();
    public boolean isEmpty();
    public Object peek();
    public void add(Object x);
    public Object remove();
}

• Priority Queues Implementation
  o  Unsorted Array - elements are inserted according to their arrival order
     •  add is O(1) — just add the item at the end (except that every once in a while we have
to copy the whole array.)
     •  remove is O(n) — must find the element.
  o  Sorted Array
     •  add is O(n) — because we will likely need to insert into the middle of the array, and
will have to shift the rest of the elements over, to make room
     •  remove is O(1)
  o  Unsorted Linked List - elements are inserted according to their arrival order
     •  add is O(1) — just add the item at the end.
     •  remove is O(n) — because we need to examine every element to determine which
one has the highest priority
  o  Sorted Linked List
     •  add is O(n) — requires a list traversal
```
- remove is $O(1)$

- Code: Implementation of PQ using ArrayList (PriorityQueue1.java)

**Heap**

Priority Queues can be implemented as heap.

A heap is a complete binary tree in which:
- the elements are comparable, and
- no child is greater than its parent (it may be equal).

The ordering can be one of two types:

A **min-heap** is a binary tree such that
- the data contained in each node is less than (or equal to) the data in that node’s children.
- the binary tree is complete

A **max-heap** is a binary tree such that
- the data contained in each node is greater than (or equal to) the data in that node’s children.
- the binary tree is complete

- By convention, the smallest element is the one with the highest priority.
- And when talking about heap, normally referring to min-heap (unless max-heap is clearly specified)
- Since a heap is a complete binary tree, it has a smallest possible height - a heap with \( N \) nodes always has \( O(\log N) \) height.

Using a heap to implement a priority queue, we will always have the element of highest priority in the root node of the heap.

- **PQ implementation using Heap**
  - add is \( O(\log n) \)
  - remove is \( O(\log n) \)

We will implement a Priority Queue using a heap. To do so efficiently, we need to be able to quickly find the last node, add new or remove nodes in the next available location in the bottom level of the tree, and swap data between parent and children. Therefore, we will implement the heap using an array (we implement a heap with an array)!

Root is in position 1 (not zero)

![Heap Diagram]

Given a node at index \( i \):
- Index of left child: \( 2i+1 \)
- Index of right child: \( 2i+2 \)
- Index of parent: \( (i-1)/2 \)

Because the heap is complete, there will be no unused slots in the array mixed in with the used slots. We can easily find the last item and add or remove items at the end and we can easily move between parents and children. As we add and delete nodes from a heap, we will always “reheapify” in such a way that the graph is made complete as well as a heap.

**Insert into a heap**
Elements are inserted into a heap at the farthest left location at the bottom level. Then the heap property is restored by percolating the new element up the tree until its key is greater than its parent (or less or equal to the data in its child nodes). On each iteration, the child is swapped with its parent.

**ReheapUp operation:**
1. Compare the new item with its parent
2. If smaller, swap and repeat step 1

**Remove from a heap**
The heap removal algorithm always removes the root element from the tree. This is done by moving the last leaf element into the root element and then restoring the heap property by percolating the new root element down the tree until its key is less than its children. On each iteration, the parent is swapped with the smaller of its two children.
ReheapDown operation:
   1. Compare the new root with its children
   2. If bigger, swap with the smallest child, and repeat step 1