# Computer Laboratory 13
# CSci 1913: Introduction to Algorithms, Data Structures, and Program Development
**December 4–5, 2018**

THIS IS THE LAST LAB ASSIGNMENT OF THE COURSE!

## 0. Introduction.

In our culture, we claim to believe that everyone has equal rank. In other cultures, however, some people are thought to have higher rank than others. For example, the following system of ranks is similar to that used in Nineteenth Century Victorian England—and in some modern fantasy and romance novels.

| **RANK** | **DESCRIPTION** |
|---|---|
| 0 | King, Queen |
| 1 | Prince, Princess |
| 2 | Duke, Duchess |
| 3 | Marquess, Marchioness |
| 4 | Count, Countess |
| 5 | Knight, Dame |
| 6 | Gentleman, Gentlewoman |
| 7 | Commoner |

We can denote each rank by a nonnegative integer, with lower integers corresponding to higher ranks. For example, a King has higher rank that a Prince, because 0 < 1. Similarly, a Prince has higher rank than a Count, because 1 < 4. And everyone has higher rank than a Commoner, except another Commoner.

In the lectures, we used a queue to model a line of people waiting to enter a movie theater. As people arrived at the theater, they joined the queue at the rear, and left the queue at the front. This makes sense where people don't have ranks. If there are ranks, however, then things work differently: people arriving at the theater enter in order of rank. For example, if a Commoner and a Prince arrive at the theater, then the Prince enters first. And if a King arrives, he enters before either of them. Commoners enter last of all.

We can model the situation just described by using a different data structure than an ordinary queue, called a *priority queue.* In a priority queue, objects (perhaps representing people) arrive at the rear in the usual way. However, instead of leaving at the front, they leave in order of their ranks. Objects with higher ranks (and lower numbers) leave before objects with lower ranks (and higher numbers).

Priority queues have other uses besides enforcing antiquated dominance hierarchies. For example, a computer operating system might work by sharing the central processor among many different programs. If some programs are more important than others, then the operating system might use a priority queue to decide which program will run first.

## 1. Theory.

We can implement something like a priority queue using a binary search tree (BST) that associates a set of *keys* with a set of *values.* The BST's values are objects. Its keys are nonnegative integers that denote the ranks of those objects. Unlike the BST's discussed in the lectures, the nodes in each left subtree have ranks *less than or equal* to the rank at the root. The nodes in each right subtree have ranks *greater than* the rank at the root. This lets two or more nodes have the same rank, which is reasonable if we assume there are only a small number of nodes in the tree.

When we enqueue a new object, we use a version of the BST addition algorithm discussed in class. If there are $n$ nodes in the tree, then the enqueueing algorithm can work in $O(\log n)$ time, assuming that the objects are added in random order of their ranks.

When we dequeue an object, we first find a node with the minimum key, again using an algorithm discussed in class. We then delete this node from the tree. This is easy to do, because it will always have at least one empty child, so we don't need a complex deletion algorithm that can delete any node. We finally return the object at the removed node. The dequeueing algorithm can also work in $O(\log n)$ time.

The BST used to implement the priority queue must have a head node, to eliminate special cases when enqueueing and dequeueing. It should use `above` and `below` pointers. BST's with head nodes were discussed in the lectures, and code that uses them is available on Moodle—but it is not the same as what you will write here.

By the way, what we've just described isn't really a priority queue! This is because objects of equal rank aren't always deleted in the order they are added. To keep the assignment simple, we'll ignore that problem: we don't care about the order in which objects with equal rank are deleted. All we care about is that no object with low rank (like a Commoner) is deleted before an object of high rank (like a King).

## 2. Implementation.

Write a class called `PriorityQueue` that implements a priority queue using a BST. The priority queue must be able to hold arbitrary ranked objects of type `Base`, so it looks like this.

```
class PriorityQueue<Base>
{
  private class Node
  {
    private Base object;
    private int  rank;
    private Node left;
    private Node right;

    private Node(Base object, int rank)
    {
      this.object = object;
      this.rank = rank;
      left = null;
      right = null;
    }
  }

  private Node root;  //  Root node of the BST.

  ⋮
}
```

The class `PriorityQueue` must have at least the following methods. You may need to write other helper methods.

`public PriorityQueue()`

> Constructor. Make a new, empty priority queue. You must set `root` to a head node, to simplify insertion and deletion by the other methods. You must decide what the `rank` slot of the head node should be. Whatever you choose, it should minimize the number of special cases needed by the other methods. Don't use special cases to add the first node, or to delete the last node!

`public Base dequeue()`

> If the priority queue is empty, then throw an `IllegalStateException`. Otherwise, remove the highest ranked object (with the lowest `rank` number) from the priority queue, using the algorithm described in the previous section. Resolve ties in rank arbitrarily. Return the removed object.

`public void enqueue(Base object, int rank)`

> If `rank` is negative, then throw an `IllegalArgumentException`. Otherwise, add `object` to the priority queue with the given `rank`, using the algorithm described in the previous section.

`public boolean isEmpty()`

Return `true` if the priority queue is empty. Return `false` otherwise.

## 3. Deliverables.

The file `tests.java` on Moodle contains Java source code that performs a series of tests. The tests call methods from the `PriorityQueue` class. Some of them print what those methods return. Each test is followed by a comment that tells how many points it is worth, and what must be printed if it works correctly.

Run the tests, then turn in the Java source code for your `PriorityQueue` class. Your TA will tell you how and where to turn it in. If your lab is on **Tuesday, December 4, 2018**, then your work must be turned in by **11:55 PM** on **Tuesday, December 11, 2018**. If your lab is on **Wednesday, December 5, 2018**, then your work must be turned in by **11:55 PM** on **Wednesday, December 12, 2018**. *To avoid late penalties, do not confuse these dates!*