Computer Programming 2 — 3th Test - May 4th 2018

- You have 1 hour to complete the assignment.
- If the code does not compile, the exercise won't be accepted for submission.
- Tests are provided to check if your code is correct. **You cannot modify the tests!**
- If a test does not pass, that exercise won't be accepted for submission.
- Code is expected to be readable, clean, and optimal.
- A skeleton of the exercise is provided. Feel free to add more files and/or include more libraries of you need them.
- To check if your code is correct, you can write the code you need in the **main()** of the file *main.cpp*, but leave it **unchanged before submitting the exam**.
- Inside the code, replace **"TYPE YOUR NAME HERE"** with your complete name.
- When you finish, **ZIP the whole folder** with a filename called **"lastname_name.zip"** and upload it to the **"Test3"** folder.

In computer science, *queues* are sequential data structures used to store elements in a first-in-first-out (FIFO) order. This means that the elements are dequeued in the same order they were enqueued, behaving as queues in real life.

In games, a good example of their usefulness is in event systems. Many things (events) can happen at any time during the gameplay (a bomb explodes, a particle collisions with an object, an animation finishes, the current level has been completed...). These events are usually processed in FIFO order, and thus, a *queue* is the preferred data structure. This piece of code makes use of a standard queue:

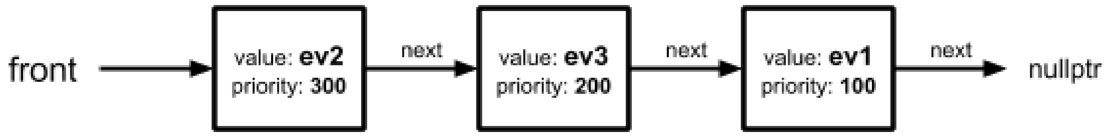
```
// Example of use of a normal queue (seen in class)
queue.enqueue(ev1);
queue.enqueue(ev2);
queue.enqueue(ev3);
queue.dequeue(); // dequeues ev1
queue.dequeue(); // dequeues ev2
queue.dequeue(); // dequeues ev3
```

Sometimes, however, some events need to be more important than others and be processed first (e.g. an *exit-game* event will probably need to be processed before other events that were happening during the game, and even discard them).

For that purpose, *priority queues*, are a special type of queues where the enqueued elements follow a priority-based order. They do not follow a FIFO scheme anymore. Instead, the enqueued elements are positioned in the queue based on a priority value. The higher the priority, the sooner the element will be dequeued. This piece of code makes use of a priority queue (note that the order of dequeuing depends on the priority value given when the elements were enqueued).

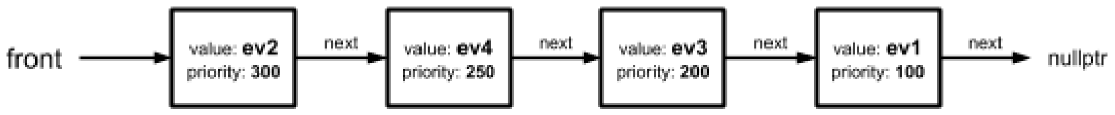
```
// Example of use of a priority queue
queue.enqueue(ev1, 100); // priority 100
queue.enqueue(ev2, 300); // priority 300
queue.enqueue(ev3, 200); // priority 200
queue.dequeue(); // dequeues ev2
queue.dequeue(); // dequeues ev3
queue.dequeue(); // dequeues ev1
```

The file *PriorityQueue* provides the partial code of a *priority queue* class. The implementation is based on singly-linked lists. That means that the collection of elements in the queue are tied together by means of a sequence of nodes that provide a pointer to the next node, as in the following illustration:



As seen in the previous image, the events are organized by priority (events of higher priority are closer to the front of the queue).

If now, for instance, a new event *ev4* with *priority=250* was enqueued, the state of the queue would be the following:



Exercise: Implement the method *enqueue()* of the *PriorityQueue* class.