# A simple min-queue for integers

- Total points: 5 (1 per task)
- Number of submissions allowed: 3
- General rules are as usual: the `"ex.py"` file should be clean; no unnecessary printing; only the behavior matters; refer to the `"runtest.py"` file for specifics and examples. The full specification is at the end of this document.

## Introduction

We want to implement a simple min-queue to store and retrieve integers, in a class called `IntQueue`. Here are the details (examples follow):

1. The queue constructor takes a strictly positive integer argument `m`. This number is stored in the queue and never changes. The queue will then be able to contain integers between `0` (included) and `m` (excluded).
2. We will have methods to `put` integers in the queue and `get` them out, one at a time:
   - The `put` method takes an argument `i`, which is the integer that we will store in the queue.
   - The `get` method takes no arguments and returns the smallest element currently in the queue (also removing it from the queue)
3. Internally, the queue holds a list of size `m`, called `data`. Initially, when the queue is empty, `data` is filled with zeros.
4. When we `put` an integer `i` in the queue, we increment `data[i]` by `1`. Conversely, if `get` gives us an integer `i`, we decrement `data[i]` by `1`. Thus, `data[i]` counts how many `i` are currently stored inside the queue.

So for example:

- Say that `m=5`. Then the `data` is initially `[0,0,0,0,0]` (the queue is empty).
- Say that we `put(4)`. Then `data==[0,0,0,0,1]`. Then we `put(1)`, so `data==[0,1,0,0,1]`. Then we `put(4)` again, so `data==[0,1,0,0,2]`.
- Say that now we `get()`: the output is `1` because that's the smallest element in the queue, and `data==[0,0,0,0,2]`.
- We `get()` again. Now the smallest element is `4`: it's returned and `data==[0,0,0,0,1]`.

More details on the internals:

5. In addition to `m` and `data`, the queue has one more internal attribute, called `nxt` (short for *next*):
   - The attribute `nxt` keeps the current minimum element in the queue. It is called like this because it is the value that the next `get()` will return. It must always correspond to the smallest index where `data` is non-zero. If the queue is empty, this attribute contains `m` (which is an invalid index) and the next `get()` will raise an exception.
   - Therefore, whenever we use `put` or `get` we might need to update `nxt`
6. We will also have a `getdefault` method, described below.

## Tasks

Each task is worth 1 point. The test file `"runtest.py"` contains code that tests all tasks independently. For each task, if it fails you will see a backtrace telling you where the error was produced, otherwise it prints a "tests ok" message (meaning that the code *may* be correct). Note: task 5 depends on task 4; however, if it is written correctly, it still counts as ok even if task 4 is incorrect and the tests fail.

In the initial file the constructor and the `__repr__` method are already implemented.

1. In the constructor, add two checks for the `m` argument: if it is not an `int` raise a `TypeError`, if it is not a valid int raise a `ValueError`.

2. Write a method such that the `len` built-in function can be applied to `IntQueue` objects, returning the number of elements in the queue, which is just the sum of the elements in `data`. One line of code.

3. Write the `put` method as described above: it takes an argument `i`. No argument checks are required. Update the

internal `data` and (if needed) `nxt`. Return `None`.

4. Write the `get` method as described above. If the queue is empty raise a `KeyError`. To check if the queue is empty, you must not look at the `data`, only at `nxt`. If the queue is not empty, you need to update `data` before returning. Depending on the result of the update, you may need to update `nxt` too.

5. Write a `getdefault` method. Its purpose is to behave precisely like `get`, except if the queue is empty. In that case, it returns a default value. The default value is passed as an argument, call it `default`. It must itself have a default value of `-1`. For example:

   - Non-empty case: `get()` would return a number, let's say `4`. Then both `getdefault(2)` and `getdefault()` should return `4` (the `default` value is ignored).
   - Empty case: `get()` would raise a `KeyError`. Then `getdefault(2)` should return `2` and `getdefault()` should return `-1`.

   This method in fact *must use* the `get` method already written, and in particular it must work by calling it (once!) and catching the error in case the queue is empty.

### General rules

In all methods, you can use whatever constructs you want. Only the behavior matters, as long as you follow the indications.

The class methods must not print anything. When the file `"ex.py"` is loaded nothing should be printed. Do not write anything else in the file, just the class(es) and, if needed, `import` statements.

You can do tests by typing `python runtest.py` in the terminal, or pressing the `RUN` button. Edit the file `"runtest.py"` as much as you want when you test. That file is for your own use and it's not used when grading.

Remember that copies of the original files are always present in the `"startercode"` directory, in case you need them.

As for examples of how everything should work, refer to the test script.