Carleton University
Department of Systems and Computer Engineering
SYSC 2100 — Algorithms and Data Structures — Winter 2023

Lab 5 - ADT BoundedPriorityQueue

Getting Started

1. Review *Important Considerations When Submitting Files to Brightspace*, which can be found in the course outline.

2. Launch Wing 101 and configure Wing's code reformatting feature. Instructions can be found in Appendix A of document, *SYSC 2100 - Style Guide for Python Code*.

   All code you submit for grading must be formatted. If you decide to disable automatic reformatting, make sure you manually reformat your code (Appendix A.2). At a minimum, we recommend that you reformat your file as you finish each exercise.

3. Download lab5_boundedpriorityqueue.py and lab5_test_boundedpriorityqueue.py from the *Lab Materials* module in Brightspace and open the files in Wing 101.

4. In lab5_boundedpriorityqueue.py, locate these assignment statements at the top of the file:

   ```
   __author__ = ''
   __student_number__ = ''
   ```

   Replace the empty strings with strings containing your name and student number. (Don't modify the variable names.)

5. Your solutions to the exercises must conform to the coding conventions described in *SYSC 2100 - Style Guide for Python Code*.

6. **Important**: if you decide to write a script in lab5_boundedpriorityqueue.py, it **must** be placed in an `if __name__ == '__main__':` block, like this:

   ```
   class BoundedPriorityQueue:
       # methods not shown

   if __name__ == '__main__':
       pq = BoundedPriorityQueue()
           ...
   ```

   Doing this ensures that the script will only be executed when you run lab5_boundedpriorityqueue.py as the "main" module, and won't be executed when the file is imported into another module; for example, the testing/grading program provided to the TAs.

Overview of the Bounded Priority Queue ADT

A *priority queue* is a variant of the queue ADT in which each item is assigned a priority when it is added to the queue. A *bounded* priority queue restricts the priorities to integers that range from 0 (the highest priority level) to *num_levels* - 1, where *num_levels* is the number of priority levels supported by the queue.

Items with a higher priority are removed before any item with a lower priority. When multiple items have the same priority, they are removed in FIFO order.

Consider this code snippet, which creates an empty bounded priority queue with ten priority levels ranging from 0 to 9. Six items (strings that name colours) are added to the queue:

```
pq = BoundedPriorityQueue(10)
pq.add("purple", 5)
pq.add("black", 0)
pq.add("orange", 3)
pq.add("white", 0)
pq.add("green", 1)
pq.add("yellow", 5)
```

The next snippet is a loop that removes the items and prints them, one-by-one:

```
while len(pq) > 0:
    print(pq.remove())
```

The output will be:

```
black
white
green
orange
purple
yellow
```

The strings `"black"` and `"white"` have the highest priority (0), so those are the first two items that are removed, even though they were added by the second and fourth calls to `add`. `"black"` is removed before `"white"`, because items with equal priority are removed first-in, first-out, and `"black"` was added to the priority queue before `"white"`.

Exercise 1: Software developers often have to read the documentation for modules and libraries that they have not used previously and construct experiments that help them understand the software.

The implementation of the bounded priority queue ADT will use instances of Python's `deque` class as FIFO queues. To learn about this class, read the documentation for this class in the Python *Library Reference*, Section *`deque` objects*:

https://docs.python.org/3/library/collections.html#collections.deque

In the Python shell, try the examples that are presented towards the end of the section.

When an efficient FIFO queue is required in a Python program, a `deque` is often used instead of a Python `list`, because the methods that append and pop items at either end of a `deque` execute in constant time.

Section 5.1.2 of the *The Python Tutorial*, *Using Lists as Queue*s, shows how to use a `deque` as a FIFO queue. Open the tutorial (the URL is https://docs.python.org/3/tutorial/index.html), read this section and try the examples in the shell.

Exercise 2: Class `BoundedPriorityQueue` in lab5_boundedpriorityqueue.py is an incomplete implementation of the bounded priority queue ADT. The data structure is a fixed-capacity array of FIFO queues, with one queue for each priority level.

Four methods in class `BoundedPriorityQueue` have been implemented for you. **Do not modify these methods.**

- `__init__` initializes an empty `BoundedPriorityQueue`. Parameter num_levels is the number of priority levels supported by the priority queue.

  A `BoundedPriorityQueue` object has two attributes (instance variables): `_num_items`, which keeps track of the number of items in the priority queue, and `_queues`, which refers to a fixed-capacity array of FIFO queues (instances of class `deque`). When an item with priority *k* is added to the priority queue, it is enqueued in the deque referred to by `_queues[k]`.

  The array is created by function `_new_array`, which we've used previously in classes `ArrayBag` and `ArrayList`. Note that the number of priority levels supported by a priority queue can be determined by calculating the length of its array; that is, `len(self._queues)`.

- `__str__` returns a string representation of the `BoundedPriorityQueue`. The string contains each item and its priority, arranged from the highest priority to the lowest. When multiple items in the priority queue have the same priority, they are listed left-to-right in the order in which they were added. (See the docstring for method `add` for an example.)

- `__repr__` returns a string representation of the `BoundedPriorityQueue`. This string is a Python expression that could be used to create an empty `BoundedPriorityQueue` object.

- `__len__` returns the number of items in the priority queue.

"Stub" implementations have been provided for methods `add` and `remove`. If you call these methods on a `BoundedPriorityQueue` object, Python will raise a `NotImplementedError` exception. You'll complete the implementation of these methods in Exercises 3 and 4.

File lab5_test_boundedpriorityqueue.py is an incomplete set of unit tests for class `BoundedPriorityQueue`, implemented using the PyUnit (`unittest`) framework. Read the test classes. The script at the bottom of the file calls `unittest.main`, which will run all the test cases (the methods with names that start with `test`). Run the script and review the output in the shell window.

Before you start Exercise 3, read `__init__`. Now consider this statement:

```
pq = BoundedPriorityQueue(4)
```

Draw a memory diagram that depicts the object referred to by `pq`. The diagrams should be similar to one that would be produced by Python Tutor and show attributes `_num_items` and `_queues`, the array referred to by `_queues`, and the empty `deque` objects referred to by the array.

When designing the `add` and `remove` methods in Exercises 3 and 4, we recommend that you draw diagrams of the priority queue, depicting the data structure before and after the methods are called. If you do this, coding these methods should be straightforward.

Exercise 3: Read the docstring for `add`, delete the `raise` statement, then implement the method. The description of `__init__`, above, explains where the item should be stored.

Use the Python shell to try the docstring examples. If the string returned by `str(pq)` differs from the one shown in the docstring, the problem is in `add`, not `__str__`. Fix the problem by modifying `add` - **don't modify `__str__`**!

Class `AddTestCase` (in lab5_test_boundedpriorityqueue.py) contains some test methods. Think about what additional test cases that would be required to thoroughly test `add` and implement them as methods in class `AddTestCase`. Run the test script and review the output in the shell window. If necessary, edit `add` and rerun the test script. At this point, all test cases for `__init__`, `__str__`, `__repr__` and `add` should pass.

Exercise 4: Read the docstring for `remove`, delete the `raise` statement, then implement the method.

Use the Python shell to try the docstring examples.

Class `RemoveTestCase` (in lab5_test_boundedpriorityqueue.py) contains some test methods. Think about what additional test cases that would be required to thoroughly test `remove` and implement them as methods in class `RemoveTestCase`. Run the test script and review the output in the shell window. If necessary, edit `remove` and rerun the test script until all test cases pass.

Wrap-up

- Before submitting lab5_boundedpriorityqueue.py, review your code. Does it conform to the coding conventions, as specified in the *Getting Started* section? Has it been formatted? Did you edit the `__author__` and `__student_number__` variables?
- Submit lab5_boundedpriorityqueue.py to Brightspace. Make sure you submit the file that contains your solutions, not the unmodified file you downloaded from Brightspace! You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the submission due date. Only the most recent submission will be saved by Brightspace.
- Don't submit lab5_test_boundedpriorityqueue.py.
- Solutions that are emailed to your instructor or a TA will not be graded, even if they are emailed before the submission due date.