Carleton University
Department of Systems and Computer Engineering
SYSC 2100 - Algorithms and Data Structures - Winter 2021

Lab 11 - Heaps and Priority Queues

Submitting Lab Work for Grading

Remember, you don't have to finish the lab by the end of your lab period. The deadlines for submitting solutions to cuLearn for grading are listed in the *Wrap Up* section at the end of this handout. Solutions that are emailed to your instructor or a TA will not be graded, even if they are emailed before the deadline.

Please read *Important Considerations When Submitting Files to cuLearn*, on the last page of the course outline.

References

Problem Solving with Algorithms and Data Structures Using Python, Chapter 7, Sections 7.8 (*Priority Queues with Binary Heaps*), 7.9 (*Binary Heap Operations*) and 7.10 (*Binary Heap Implementation*).

Lecture slides: *Binary Heaps*

Getting Started

Download min_heap.py, max_heap.py and priority_queue.py from the *Lab Materials* section of the main cuLearn course page.

Exercise 1: Sections 7.8 and 7.9 describe class `BinaryHeap`, which implements the *min heap* data structure. The code is in min_heap.py (I've added a docstring to each method, but otherwise, the code is identical to the class presented in the textbook.)

Read Sections 7.8 and 7.9. Experiment with min heaps by creating a `BinaryHeap` object and calling its "public" methods (`insert`, `get_min`, `delete`, `is_empty`, `len` and `heapify`). Make sure you understand these methods, as well as the "private" helper methods `_perc_up`, `_perc_down` and `_get_min_child`. To help you understand the code, you might want to use Python Tutor to run some experiments.

Exercise 2: File max_heap.py is identical to min_heap.py. The lecture slides describe the *max heap* data structure and present the algorithms for inserting and removing items. Using the lecture slides as a reference, change the `BinaryHeap` class in max_heap.py so that it implements a max heap instead of a min heap.

Method `get_min` should be renamed `get_max`, to reflect what a max heap's "get" operation does. Other than that, don't change the names, parameter lists or return types of `insert`, `delete`, `is_empty`, `len` and `heapify`.

Put yourself in the shoes of a software developer who will be asked to maintain your code. Edit your code to reflect the guidelines described in *Programming Style - Writing Readable Code* on page 4. Update the docstrings and variable names to be consistent with the changes you made to the Python code (changing a min heap to a max heap). If additional comments would make the code more readable, you should add them. Assume that anyone who reads your code knows Python, so don't add trival comments, like this:

```
x += 1   # Add 1 to x
```

The short script towards the end of max_heap.py doesn't test class `BinaryHeap` thoroughly. Use the shell or write a short script to test `BinaryHeap`. If you write a script, please put it in a separate module (file). We don't want test scripts in the max_heap.py file you submit to cuLearn. (You don't need to delete the short script that is in the file.)

Exercise 3: Class `PriorityQueue` in file priority_queue.py is a partial implementation of ADT Priority Queue. It uses a max heap as the underlying data structure.

Read the docstrings for methods `insert`, `get_highest`, `remove_highest` and `__str__`. Replace the `raise` statements with correct implementations of these methods.

Use the shell or write a short script to test the function. If you write a script, please put it in a separate module (file). We don't want test scripts in the priority_queue.py file you submit to cuLearn.

Notes:

1. You may not modify the **public** interface of this class; that is, add **public** methods to class `PriorityQueue`, change the number or types of the method parameters, etc.

2. You should reuse your max-heap code from Exercise 2. For example, you can copy/paste the `_perc_up`, `_perc_down` and `_get_max` methods into priority_queue.py. (These are "private" methods, so copying these methods does **not** violate the requirements in Note 1.) Alternately, you can copy the bodies of any of the methods in max_heap.py into methods in priority_queue.py.

3. The items stored in the priority queue are strings. The priority levels are integers: the larger the integer, the higher the priority. For example, if you were writing a simulation of a hospital emergency room, the items could be the names of people. As each person arrives, they would be assigned a priority level based on the urgency of their condition, as determined by the triage nurses. People with the most urgent conditions (highest priorities) would be treated by the ER staff before people with less severe conditions.

4. Each node in the priority queue's heap will store one item and its priority. How you do this is up to you.

5. If a priority queue has two or more items with the same priority, there is no requirement that they be returned/removed in FIFO order. For example, if a priority queue contains one item with priority 10 and three items with priority 9, the priority 10 item must be the

first item that is removed by `remove_highest`. The next call to `remove_hightest` should remove one of the three priority 9 items (the one at the root of the heap). This will not necessarily be the first priority 9 item that was inserted in the queue.

6. The format of the string returned by `__str__` is up to you; however, it should be concise (terse) and contain each of the items in the priority queue, along with their priority levels. In other words, __str__ shouldn't return a wordy string in which each item and priority are on a separate line. (Consider the strings returned by the `__str__` methods from previous labs and assignments.)

7. Your implementation of ADT Priority Queue cannot use any of the "growable" ADTs that were studied earlier in the course; e.g., ADT Bag, ADT Stack, ADT Queue, ADT Deque, ADT UnorderedList, ADT Map. It cannot use a linked list, a hash table, or a binary search tree as the underlying data structure. **You do not need to modify `__init__`. The only instance variable that is required is `self._heap`, which initially refers to an empty Python list.**

Programming Style - Code Formatting

Before submitting your code, please run it through Wing 101's source reformatter, as described in following paragraphs.

Wing 101 can easily be reconfigured to reformat your code so that it adheres to some of the formatting conventions described in the "official" Python coding conventions document, PEP 8 - *Style Guide for Python Code*.[1]

To configure reformatting, follow the instructions in Section 3.2 of *Installing Python 3.9.1 and Wing 101 7.2.7*, which is posted on cuLearn. (The instructions apply to both the Windows 10 and macOS versions of Wing 101.)

If you prefer, you manually reformat your files even if you've disabled automatic reformatting. This is described in Section 3.3. Note that you still have to open the Auto-formatting window to configure which PEP 8 conventions are followed, as described in Section 3.2.

Manually reformatting an open file is easy:

- From the menu bar, select Source > Reformatting
- From the pop-up menu, select Reformat File for PEP 8

Wing 101 7.2.8 fixes a bug in release 7.2.7: the Reformat Selection for PEP 8 command now works.

[1] https://www.python.org/dev/peps/pep-0008/

Programming Style - Writing Readable Code

Focus on writing code that is readable and maintainable, and not just code "that works". Using a code reformatter can improve the layout of your code, but it can't enforce every coding convention. For example, a reformatter won't change your variable names if they aren't descriptive or follow the "official" Python convention (that is, variables names are lowercase, with words separated by underscores).

Here are some things to consider:[2]

"The ease by which other *people* can read and understand a program (often called "readability" in software engineering) is perhaps the most important quality of a program. Readable programs are used and extended by others, sometimes for decades. For this reason, we often say that programs are written to be read by humans, and only incidentally to be interpreted by computers.

A program is composed well if it is concise, well-named, understandable, and easy to follow.

Excellent composition does not mean adhering strictly to prescribed style conventions. There are many ways to program well, just as there are many styles of effective communication. However, the following guiding principles universally lead to better composition of programs:

- **Names**. To a computer, names are arbitrary symbols: "xegyawebpi" and "foo" are just as meaningful as "tally" and "denominator". To humans, comprehensible names aid immensely in comprehending programs. Choose names for your functions and variables that indicate their use, purpose, and meaning.

- **Functions**. Functions are our primary mechanism for abstraction, and so each function should ideally have a single job that can be used throughout a program. When given the choice between calling a function or copying and pasting its body, strive to call the function and maintain abstraction in your program.

- **Purpose**. Each line of code in a program should have a purpose. Statements should be removed if they no longer have any effect (perhaps because they were useful for a previous version of the program, but are no longer needed). Large blocks of unused code, even when turned into comments, are confusing to readers. Feel free to keep your old implementations in a separate file for your own use, but don't turn them in as your finished product.

- **Brevity**. An idea expressed in four lines of code is often clearer than the same idea expressed in forty. You do not need to try to minimize the length of your program, but look for opportunities to reduce the size of your program substantially by reusing functions you have already defined."

[2] Adapted from a style guide used by students in an introductory programming course at the University of California, Berkeley.

Wrap Up

Please read *Important Considerations When Submitting Files to cuLearn*, on the last page of the course outline.

The submission deadline for this lab is Tuesday, April 6, 23:55 (Ottawa time), for all lab sections.

To submit your lab work, go to the cuLearn page **for your lab section** (not the main course page). Submit max_heap.py and priority_queue.py. Ensure you submit the version of the files that contain your solutions, and not the unmodified file you downloaded from cuLearn! You are permitted to make changes to your solutions and resubmit the file as many times as you want, up to the deadline. Only the most recent submission is saved by cuLearn.

Last edited: March 29, 2021 (initial release); April 1, 2021 (Exercises 2 and 3 updated to incorporate answers to students' questions); April 3, 2021 (updated Note 7 on page 3, in response to students' questions).