UCIC—COSC002 2019 Semester 1
INTRODUCTION TO COMPUTER SCIENCE

Assignment Part 2

8% of overall grade
Due 11:55pm on Friday 17 May 2019

1 Overview

1.1 Introduction

This assignment presents a number of heap based priority queues applications. The work required is designed to help you understand both the theoretical and practical concerns related to data structures. In addition, it is hoped this assignment will give you an appreciation of the versatility and usefulness of heap based priority queues.

1.2 Due date

11:55pm on Friday 17 May 2019.

1.3 Submission

Submit via the quiz server. The submission page will be open closer to the due date.

1.4 Implementation

All the files you need for this assignment can be found on the quiz server. Do not import any additional standard libraries unless explicitly given permission within the task outline. For each section there is a file with functions or classes for you to fill in. You need to complete these functions and classes, but you can also write other functions that these classes call if you wish, as long as your new functions don't overwrite existing methods or functions (e.g. from a base class). All submitted code needs to pass *pylint* program checking.

1.5 Getting help

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If

you get stuck on a programming issue, you are encouraged to ask your tutor or the lecturer for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work. If you have a general assignment question or need clarification on how something should work, please use the class forums on Learn, as this enables everyone to see responses to common issues, but NEVER post your own program code to Learn. Remember that the main point of this assignment is for you to exercise what you have learnt from lectures and labs, so make sure you do the relevant labs first, so don't cheat yourself of the learning by having someone else write material for you.

2 Priority Queues

This assignment is split into 4 tasks. For the first task of this assignment you will use the `BasicPriorityQueue` class to find the top k values in some list. The second will be related to speeding up heapsort when using priority queues (in particular speeding up the heapify operation). The final two tasks are about augmenting the priority queue with a dictionary for fast removal of non-root items from the priority queue and changing the branch factor of the underlying heap structure (e.g. using a ternary heap instead of a binary heap).

You will be provided with offline tests in the file `tests.py` to check your implemented code before submission. The tests used on the quiz server will be based on these unittests but with a large number of added test cases, so passing the unit test is not a guarantee that full marks will be given for that question (it is however a good indication you're on the right track).

2.1 Provided modules and classes

2.1.1 classes.py

The `classes.py` module contains the definition for the `Key` class and a function that you can use to convert any iterable into a list of `key` objects. The `Key` class is a simple wrapper, around a normal python object, that keeps track of any comparisons (via the `StatCounter` class in the `stats` module. For example:

```
# make a Key from an int
my_key = Key(1)
print(my_key)

# make a key from a str
my_key = Key('a')
print(my_key)

# make a list of keys from a list of ints
my_keys = list_of_keys([1,2,3,4])
my_keys = list_of_keys([1,2,3,4])
print(my_keys)

# make a list of keys from a range
my_keys = list_of_keys(range(0,10))
print(my_keys)
```

2.1.2 The basic priority queue

The main class provided is `BasicPriorityQueue`, which is a max heap based priority queue. The `BasicPriorityQueue` class exposes the the following methods and usage to users:

- `my_pq = BasicPriorityQueue()` creates a new empty priority queue. This creates an empty list, `_items`, to represent a heap.
- `my_pq = BasicPriorityQueue(my_list)` creates a new priority queue with initial items found in the list `my_list`. This creates an internal `_items` list equal to `my_list` and calls the `_heapify()` method to establish the heap property.
- `len(my_pq)` gives the number of items in the priority queue.
- `insert(item)` inserts the given item into the priority queue.
- `peek_max()` returns the item in the priority queue with the highest priority.
- `pop_max()` returns the item in the priority queue with the highest priority and removes the item from the priority queue.
- `get_comparisons()` returns the number of item comparisons made by the priority queue since creation (with the exception of those made in validate). Note: this includes those made in any calls to `_heapify()`.
- `validate()` returns `True` if the heap property holds for the list `_items` and `False` otherwise. This method should only be used for debugging purposes.

In addition, there are a number of methods intended for internal use (by the class itself or its subclasses):

- `_heapify()` establishes the heap property by calling `sift_up` on all items in the list `_items`.
- `_parent_index(cur_index)` returns the index of the parent of the item at index `cur_index` in the list `_items`.
- `_children_indices(cur_index)` returns a list of the indices of the children of the item at index `cur_index` in the list `_items`.
- `_max_child_index(cur_index)` returns the index of a child of the item at `cur_index` with maximal priority.
- `_swap_items(index, swap_index)` swaps the position of the items at `index` and `swap_index`.
- `_sift_up(index)` repeatably swaps the item at `index` with its parent until it has priority smaller than its new parent or is the root item.
- `_sift_down(index)` repeatably swaps the item at index with the item at its `_max_child_index(index)`, until it has priority larger than all its new children or is terminal (a leaf).

Note in the `BasicPriorityQueue`, the priority of an item is simply given by its value. Later in Task 3 we will implement a `ChangePriorityQueue` class which explicitly stores a priority for each item.

The following example code should help you understand how to use `BasicPriorityQueue`:

```
>>> from basic_priority_queue import BasicPriorityQueue
>>> my_pq = BasicPriorityQueue()
>>> my_pq.insert(5)
>>> len(my_pq)
1
>>> my_pq.insert(7)
>>> my_pq.insert(10)
>>> len(my_pq)
3
>>> print(my_pq)
[10, 5, 7]
>>> my_pq.insert(9)
>>> print(my_pq)
[10, 9, 7, 5]
>>> print(peek_max())
10
>>> print(pop_max())
10
>>> print(my_pq)
[9, 5, 7]
>>> print(my_pq.get_comparisons())
6
>>> print(my_pq.validate())
True
>>> my_pq = BasicPriorityQueue([4,7,3,1,2])
>>> print(my_pq)
[7, 4, 3, 1, 2]
>>> print(my_pq.validate())
True
```

2.2 Provided tests

The `tests.py` provides a number of tests to perform on your code. Running the file initially will cause all Task 1 tests to be carried out. Each test has a name indicating what it is testing for and a contained class indicating which task is being tested. In the case of a test case failing, the test case will print which assertion failed or what exception was thrown. The `all_tests_suite()` function has a number of lines commented out indicating that the commented out tests will be skipped; uncomment these lines out as you progress through the assignment tasks to run subsequent tests.

The provided tests are limited and only check basic test cases. As such, you are expected to do your own testing of your code. This is especially true of the Task 2 tests which only check heap property is achieved and not that the number of comparisons made is within the desired bounds. You are encouraged to write your own extra unit tests too!

3 Tasks

3.1 Top k [15 Marks]

You are a software engineer at Custom Data Structures inc. Your company has recently released a product called basic priority queue which uses a binary max heap to implement a priority queue. The aim is to get clients using the basic priority queue and then customize the code based on their needs.

Convince people there is an advantage in using basic priority queue by writing a demo application that compares the priority queue approach to a naive, selection sort based, approach.

3.1.1 Top k technical information

For this task you will need to implement the two functions in `top_k.py`. The first `top_k_select(items, k)` needs to use a modified selection sort to find the top k items from the input list. This can be thought of selecting the biggest item from the list of (remaining unsorted) items, k times. You will need to keep track of the number of item comparisons your function makes. For the second function `top_k_heap(items, k)`, you should use a `BasicPriorityQueue` to find the top k values. The number of comparisons made in this case can be found using the `get_comparisons` method in `BasicPriorityQueue`. You may assume the bound $1 \leq k \leq$ `len(items)` holds.

Note: `top_k_select` and `top_k_heap` should both return the list of the top k values *in descending order* and the number of item comparisons that were carried out when creating the top k list.

3.2 Fast heapify [25 Marks]

A client wants to use our priority queues for fast sorting (an implementation of heap sort) but is unhappy with the performance of constructing the basic priority queue. Write a custom faster priority queue that speeds up the `_heapify()` method average item comparisons from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$.

3.2.1 Fast heapify technical information

This task requires you to implement the `_heapify` method in `fast_priority_queue.py`. This method needs to take less comparisons than the `_heapify` method of `BasicPriorityQueue`. More precisely, for any list of n initial items the number of comparisons needed to heapify the list should be upper bounded by $2n$. A trivial lower bound of $n-1$ should also be observed.

Important note: the unit tests given to you for this task *only* check if `_heapify` produces a valid heap. You need to do testing to confirm that your code achieves the given bounds, ie, you need to test on a variety of different input lists. An alternative is to convince yourself analytically that the method you write is bounded by upper bounded by $2n$ for any initial list of size n.

Hint1: this question is conceptually very difficult, please give yourself enough time to review lecture notes, consult external sources (if necessary and without copying code) and think hard about the problem.

Hint2: think about the number of comparisons the `BasicPriorityQueue._heapify` method takes and look at the code for `sift_up` and `sift_down`.

3.3 Removing from a priority queue [35 Marks]

A client wants to use our priority queue to implement a basic auto trader of watermelons. When the auto trader finds a watermelon being sold online the auto-trader associates a priority with the watermelon deal based on price, weight, shipping cost and a number of other factors. At the start of every hour the auto trader buys the best watermelon deal. If a watermelon is bought by someone else or

is no longer for sale the watermelon must be removed from the priority queue. Since this is happening quite often the client wants to be able to remove items quickly (i.e. $\mathcal{O}(\log n)$) from the priority queue. Your task here is to write a custom `ChangePriorityQueue`, based on `BasicPriorityQueue`, for this client.

3.3.1 Removing from a priority queue technical information

You will need to complete the methods in the file `change_priority_queue.py` for this task. Note: you must not change the `__init__` method that is provided.

The `ChangePriorityQueue` class works significantly differently to its base `BasicPriorityQueue` class. The main difference is the `_items` list is used to house `(priority, item)` tuples instead of just the items themselves. Hence `insert` has been disabled and `insert_with_priority` is used instead as each item needs an explicit associated priority. Item priorities, instead of the items themselves, are used when comparing nodes in the heap.

Another important change is that a dictionary `_item_indices` is maintained to keep track of the index of each item in the `_items` list. This is what allows quick removals from the priority queue as it avoids the need for a linear search through `_items` to find the item that is being removed; instead, we can find the index of an `item` to be removed via `_item_indices[item]`. For this to work correctly you will need to update the `_item_indices` dictionary appropriately in each of the following methods: `_swap_items`, `insert_with_priority`, `pop_max` and `remove`.

A short summary of the methods you need to write is provided:

- `_swap_items(index, swap_index)` swaps the position of the items at `index` and `swap_index`. This is used by `_sift_up(index)` and `_sift_down(index)`. It is crucial to update the dictionary `_item_indices` with the changed indices of both items.
- `insert_with_priority(item, priority)` adds the tuple `(priority, item)` to the `_items` list and adds appropriate value to `_item_indices` dictionary. Then does some sifting to restore the heap property, as per usual.
- `pop_max()` returns the item with max priority in the priority queue and removes it. Remember to update `_item_indices` by removing the item and index entry for the removed item.
- `peek_max()` returns the item with max priority in the priority queue. Note this returns the item not the tuple consisting of priority and item. You will need to use the appropriate index of the tuple at the root of `_items` to complete this method.
- `remove_item(item)` uses the `_item_indices` dictionary to find the `index` to remove from. If the item is not in the priority queue this method returns `None`. If the item is in the priority queue this method swaps the `index` with the last item in `_items`, removes the last item from `_items` and calls either `_sift_up(index)` or `_sift_down(index)` depending which is appropriate for the swapped item. Lastly this method updates `_item_indices` (to remove the index entry for the removed item). This method should return the removed item or `None` (if the item is not in the priority queue).

The following example code should help you understand how `ChangePriorityQueue` and `_item_indices` should work once all methods have been completed:

```
>>> from change_priority_queue import ChangePriorityQueue
>>> my_pq = ChangePriorityQueue()
>>> my_pq.insert_with_priority('a', 5)
>>> my_pq.insert_with_priority('b', 2)
>>> my_pq.insert_with_priority('c', 7)
>>> my_pq.insert_with_priority('d', 3)
>>> print(my_pq)
[(7, 'c'), (3, 'd'), (5, 'a'), (2, 'b')]
>>> print(my_pq._item_indices) #note print order is random for dict.
{'c': 0, 'd': 1, 'b': 3, 'a': 2}
>>> print(my_pq.remove_item('d'))
d
>>> print(my_pq)
[(7, 'c'), (2, 'b'), (5, 'a')]
>>> print(my_pq._item_indices) #note print order is random for dict.
{'a': 2, 'c': 0, 'b': 1}
>>> print(my_pq.remove_item('b'))
b
>>> print(my_pq.remove_item('b'))
None
>>> print(my_pq)
[(7, 'c'), (5, 'a')]
>>> print(my_pq._item_indices) #note print order is random for dict.
{'a': 1, 'c': 0}
>>> print(my_pq.pop_max())
c
>>> print(my_pq._item_indices) #note print order is random for dict.
{'a': 0}
```

3.4 The d-heap [25 Marks]

A coworker has noticed that a rival company's priority queue implementation is faster for many tasks. The rival company uses a ternary heap with three branches at each level instead of the paltry binary heap implementation we offer. We wish to get ahead of our competitors by releasing a new priority queue which uses a branching factor d given by the user (with $d \geq 2$). This is known as a d-heap.

3.4.1 The d-heap technical information

For this task you need to complete the methods in the file `dheap_priority_queue.py`. Specifically you will need to implement the `_parent_index` and `_children_indices` methods. Note: `branch_factor` ≥ 2 and the `_items` is 0-indexed not 1-indexed as you may be used to.

Hint: try use a pen and paper and the `BasicPriorityQueue` class (checking your math) to work out the formula for a 2-heap. Then try work out the formula for a 3-heap, then a 4-heap, etc...until you can see the general formula. Make sure to have a look at the `_parent_index` and `_children_indices` methods in `BasicPriorityQueue` if you get stuck.

Extra: as an unmarked exercise to increase your understanding complete the function `top_k_dheap(items, k, branch_factor)` in the `top_k.py` module and look at the number of comparisons needed under various values of d, k and $items$. For example, what is the best d for finding the top 40 items from a randomly shuffled list of size 1000?