CS 232 – Final project

Important dates

Assigned: April 13, 2023
Weekly check-ins due**: April 13, April 20, and April 27 by 11:59pm**
Final project due: **May 5, 2023 by 11:59pm**
Grace period for final project ends May 8, 2023 by 11:59pm

Overview

This semester, we've talked a lot about Big O and the concepts behind it. We know that in theory, enqueue-ing an object can take O(1) time (depending on how you implement it) and that locating an item in an unordered list takes O(n) time. But we don't necessarily have a sense of what that means on a practical, day-to-day level.

For this project, you are going to implement different versions of priority queues and develop a testing framework to examine how these Big O differences play out in "real life". This will also allow you to look at whether there are real differences between node-based and list-based implementations of data structures.

When you turn in the project, you will **turn in all the code you used to test it and a write-up of the answers to the questions at the end of each level ON MOODLE**. Unit tests will still be available on Gradescope for your convenience.

D-level project

For full credit:

- Turn in all the weekly-checkins. These should be short write-ups of your progress so far that answer the following questions:
    - What did you accomplish in the last week? Did you meet your goals from the previous week?
    - How far have you gotten in the project?
    - What do you plan to have done before the next deadline? What parts do you think will be the most difficult, and which parts will be easier? How do you plan to allocate your time?
- Also, when you do turn in the final version of the project, your code has to run. If I can't run your code, I can't verify your results, so you won't get any credit for this assignment.

C-level project

Complete the D-level project. In addition:

Remember how a binary heap is a tree-based implementation of a Priority Queue (PQ)? Well, we can have sequential Priority Queues too. For this part of the final project, create a file called **PriorityQueueList.py** containing the **class PriorityQueueList**. In it, you will implement **one** of these list-based priority queues:

- A PQ where the enqueue method always appends an item to the end of the list, and the dequeue method searches the list for the highest-priority item and removes it from wherever it is.
- A PQ where the enqueue method searches the list to find where the new item should be inserted so the list is in order, and the dequeue method removes the first item in the list.

I have included the beginning of a plain old list-based queue in the files for this project (**ListQueue.py**) in case you don't want to use your own code the lab on queues we already did. But you can definitely use your own code if you want to!

For full credit for this part, you need to:

- Implement one of the PQs described above, including enqueue() and dequeue() methods. Your PQ should also have size(), is_empty(), and __str__() methods
- Your PQ should pass all tests in the C-level code on Gradescope regardless of what version you implement – the tests work for both versions
  - Some unit tests on Gradescope will be hidden, so make sure to test all the functions in a driver!
- Answer the following questions:
  - Which version of the list-based Priority Queue did you implement?
  - What are the Big O times for the enqueue(), dequeue(), size(), str(), and is_empty() functions for the PQ you implemented? Explain your answers.
- **Submit your code on Gradescope and your code and the answers to the questions on Moodle**

B-level project

Implement the D- and C-level projects. In addition:

Create a file called **PriorityQueueNode.py**. In it, you will implement one of these **node-based (aka linked list)** priority queues:

- A PQ where the enqueue method always adds an item to the end of the list, and the dequeue method searches the list for the highest-priority item and removes it from wherever it is.
  - If you remove a node from the middle or end of the list you will need to make sure to adjust the pointers to the nodes around it so the list doesn't break
- A PQ where the enqueue method searches the list to find where the new item should be inserted so the list is in order, and the dequeue method removes the first item in the list.
  - If you add a node to the middle or beginning of the list you will need to make sure to adjust the pointers to the nodes around it so the list doesn't break

I have included the beginnings of a "regular" node-based queue in the files for this project (**LinkedQueue.py, Node.py**), in case you don't want to use your own code from the midterm project as

a starting place for this one. But you can definitely use your own code if you want to! You can also make this a doubly-linked list instead of a singly-linked list if you want to.

For full credit for this part, you need to:

- Implement one of the PQs described above, including enqueue() and dequeue() methods. Your PQ should also have size(), is_empty(), and __str__ methods
- Your PQ should pass all tests in the B-level code on Gradescope regardless of what version you implement – the tests work for both versions
  - Some unit tests on Gradescope will be hidden, so make sure to test all the functions in a driver!
- Answer the following questions:
  - Which version of the Node-based Priority Queue did you implement? Why did you pick that one?
  - What are the Big O times for the enqueue, dequeue, size, is_empty and __str__ functions for the PQ you implemented? Explain your answers
- **Submit your code on Gradescope and your code and the answers to the questions on Moodle**

A-level project

We've spent all semester talking about Big O, worst-case scenarios, and how much "work" each function has to do depending on the size of the input. But we've mostly talked about it in the abstract, so now is the time to get some data and see how our theories hold up to the "real world"

Complete the D-, C-, and B-level projects. In addition:

- Examine the provided code for **BinaryHeap.py** and **BinaryHeapDataTests.py**. BinaryHeap.py is a modified version of the regular BinaryHeap that keeps track of how much "work" the insert() and perc_up() functions do and returns that number. BinaryHeapDataTests.py has code that writes the amount of "work" the function did to a file called **BinaryHeapData.csv**.
  - Notice that the file repeats the tests multiples times on random sizes of data – this is so that you can collect more data about different sizes of n.
- **Make copies** of your C and D level code (I suggest naming them something like PQList_test.py and PQNode_test.py) and modify the functions so that they also count how much "work" is being done and returning that value.
- Write your own tests to collect data on your modified Priority Queues and output the data to files.
- Answer the following questions:
  - How did you decide how to count the amount of "work" each function is doing? What did you change in the code?
  - Are the functions running in the expected Big O, more or less? Explain your answer and back it up with data for each function.
    - Note that you may not be getting "work" numbers that are exactly the same as the Big O that we talk about for that function, but you should be getting something close.
      - For example, in the code that I gave you for BinaryHeap, the "work" being done isn't exactly O(log n), but it's pretty close to (5*log n). But in terms of growth, 5*log n is still closer to log n then to any of the other options (e.g., n, n^2, etc).

        - If you don’t understand what I mean, calculate the values for log n, n, and n^2 for all the sizes of n that you have and graph them along with the “work” numbers
    - How does the “work” required to run the different operations change as the size of n changes? If you were to chart the times and work compared to the number of n, what would the graphs looks like? Were you surprised by any of the output? If so, what?
        - Hint: If you write your code output correctly, the BinaryHeapsData.csv file can be opened in Excel/Google Docs and you can make graphs with it. So the graphs thing doesn’t have to be theoretical.
- **Submit your code on Gradescope and your code, data, and the answers to the questions on Moodle**

Extra Credit (up to 5 points)

Modify the BinaryHeap.py file so that it counts the “work” being done for each function, not just insert and perc_up. Add tests to **BinaryHeapDataTests.py** to collect data on each of the other functions. Analyze the data by answering the questions for the A-level code.

You can do the extra credit in addition to any level of the final project.

Submit your code, data, and the answers to the questions on Moodle.