CS/IT 200

Lab 6: Priority Queues

Recommended Due Date: Thursday, October 22, 11:59pm

Submission

- Submit all code in a single zip file to the Lab 6 assignment folder on Kodiak.

Goal: Add functionality to `HeapPriorityQueue`. Simulate job scheduling on a computer with a variable number of CPUs.

Part I – Additional Functions

Add the following two functions to `HeapPriorityQueue`.

- `pushpop(k, v)` – Adds `(k, v)` to the priority queue and then removes and returns the minimum (key, value) pair. Calling this function should be equivalent to calling `add(k, v)` followed immediately by `remove_min()`. If `k` is the new smallest key, then `(k, v)` would be returned. This function should run in $O(\log n)$ time; however, it should be faster than if we actually called `add(k, v)` and `remove_min()`.

- `replace(k, v)` – Removes the minimum (key, value) pair and adds `(k, v)` to the priority queue, then returns the pair that was removed. Calling this function should be equivalent to calling `remove_min()` followed by `add(k, v)`. Unlike the function above, it would be impossible for `(k, v)` to be returned since the removal happens before the addition. If the priority queue is empty, then raise an `IndexError`, since the remove must happen before the addition. Again, the running time is $O(\log n)$, but it should be faster than calling `remove_min()` and `add(k, v)`.

Demonstrate these methods in `lab6.py`.

Part II – Simulation

Note: The two parts are independent. You should not need your functions from Part I to complete Part II.

One of the main applications of priority queues is in operating systems—for scheduling jobs on a CPU. In this project you are to build a program that schedules simulated CPU jobs on a computer with a number of CPUs specified at the start of the simulation.

Your program should run in a loop, each iteration of which corresponds to a cycle for the CPU. Each job is assigned a priority, an integer between -20 (highest priority) and +19 (lowest priority), inclusive. From among the jobs waiting to be processed, the scheduler must assign highest priority jobs first to a CPU that is available.

In this simulation, each job will also come with a length value, which is an integer between 1 and 100, inclusive, representing the number of cycles required to process the job. For

simplicity, assume that it is not possible to interrupt a job—once it is assigned to a CPU, the job must be completed before another job can be assigned to the same CPU.

For this simulation, the jobs will come from a CSV file. The user must be asked to input a filename at the start of the program. Each cycle, the simulation should read one row of the file. If there is no job to add that cycle, the row will read "No job". Otherwise, the format of the row is: `job_name, length, priority`.

Each cycle, the program should output which jobs are running on which CPUs. For example, in a simulation with 4 CPUs where a job is running on 3 of the 4 CPUs at cycle 22, the program should output:

```
22: A, B, -, C
```

where A, B, and C are job names, and the – indicates an empty CPU. Assume the first cycle is #1.

The simulation ends when there are no more rows to read from the file. At the end of the simulation, print the number of jobs that were completed.

Sample input/output: (user input in red)	File cpu_sim.csv:
Enter number of CPUs: 3 Enter filename: cpu_sim.csv 1: A, -, - 2: A, -, - 3: A, B, - 4: A, B, - 5: -, B, - ... Total number of jobs completed: 17	A, 4, 5 No job B, 7, -1 No job No job ...

Actual sample input/output are provided with this lab. You will be graded on different test data.

What to Submit:

- Your code (`lab6.py`, `cpu_simulator.py` and any other necessary modules, including modules given as class materials)

Rubric

Grade	Overall	Part I	Part II
4	A 4 in one part and at least a 3 in the other part.	Well commented. Optimal time and behavior.	Asks for file name. Well commented. Output matches on nearly all test cases

3	At least a 3 in one part, and at least a 2 in the other part.	Minor bugs in one or both functions.	Perfect output, but never requests filename or no comments. -OR- Incorrect output on 3-5 test cases (count jobs completed and final CPU state separately)
2	At least a 2 in both parts.	Major bug (including running time) in one function.	Never crashes or hangs. Some totally correct output, some partially correct, 1-2 completely wrong.
1	At least a 1 in both parts.	Major bugs in both functions.	Consistently matching # of jobs completed or final state of CPUs in most test cases. Never crashes or hangs.
0		Code is the same as calling add() and remove_min().	Test cases cause crashes or hangs (infinite loops). Most output doesn't match.