Data Structures

Queue Homework 2

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

Problem #1: Priority Queue

- Priority queue is a queue in which each element has a "**priority**" associated with it. Elements with **high priority** are **served first** before low priority.
- Assume, in an OS, we have tasks each with priority 1, 2 or 3
  - Assume we enqueued as following:
  - Enqueue (task_id = 1131, priority = 1)
  - Enqueue (task_id = 3111, priority = 3)
  - Enqueue (task_id = 2211, priority = 2)
  - Enqueue (task_id = 3161, priority = 3)
  - Let's print tasks in order: 3111 3161 2211 1131
    - That is: to dequeue we must first get from priority 3, if nothing from 2, if nothing from 1
- Implement a priority queue class by **black box utilizing** circular queue class

Problem #1: Priority Queue

- Queue of 8 tasks
  - You should not be able to add more than 8 regardless type
- Priority only 1 to 3
- Display: 1 row per priority
- dequeue()
  - If there is a task of priority 1, it should be returned first
  - Otherwise see in 2 or 3
- Time complexity
  - O(1) for all operations

```
PriorityQueue tasks(8);

tasks.enqueue(1131, 1);
tasks.enqueue(3111, 3);
tasks.enqueue(2211, 2);
tasks.enqueue(3161, 3);

tasks.display();
//Priority #3 tasks: 3111 3161
//Priority #2 tasks: 2211
//Priority #1 tasks: 1131

cout << tasks.dequeue() << "\n";     // 3111
cout << tasks.dequeue() << "\n";     // 3161
```

Problem #1: Priority Queue

- In the future, we will learn **heap data structure**, which can be used for priority queue
  - But priority > 1 [not limited]

```
tasks.enqueue(1535, 1);
tasks.enqueue(2815, 2);
tasks.enqueue(3845, 3);
tasks.enqueue(3145, 3);

tasks.display();
//Priority #3 tasks: 3845 3145
//Priority #2 tasks: 2211 2815
//Priority #1 tasks: 1131 1535

while (!tasks.isEmpty())
    cout << tasks.dequeue() << " ";
// 3845 3145 2211 2815 1131 1535
```

Problem #2: Circular Queue

- In the lecture, we found how added_elements variable is making our life easy
  - Easy detect empty/full cases. Recall rear=front can be for both empty and full queue
  - Easy print the queue
- To realize the effect of your design choices, develop the code without added_elements
  - You need to make a critical simple decision to be able to figure out empty/full case
  - Do needed changes for the remaining of the code.
- Testing
  - Use the exact main() body from the lecture code. Don’t change

Problem #3: Sum of last K numbers (stream)

```
class Last_k_numbers_sum_stream {
public:
    Last_k_numbers_sum_stream(int k) {
    }
    int next(int new_num) {
        // Compute and return sum of last
        // K numbers sent so far
        return 0;
    }
};
int main() {
    Last_k_numbers_sum_stream processor(4);

    int num;
    while (cin >> num)
        cout << processor.next(num) << "\n";
```

- This class receives a **infinite stream** of numbers, each time return sum of last k numbers
- E.g. if k = 4
- Stream: 1 2 3 **4** 5 6 7 8 9
- Returns: 1, 1+2, 1+2+3, **1+2+3+4**, 2+3+4+5, 3+4+5+6, ..
  - That is for 6 ⇒ 18

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."