MTE 140: Assignment
Dynamic Circular Queue & Customer Service Desk

Overview

You are provided with a fixed-capacity circular-array `Queue` implementation. In this assignment you will do two things:

1. **Part A (3 pts):** Convert the fixed-capacity queue into a *dynamically resizing* circular queue:
   - Double capacity whenever `enqueue(...)` is called on a full queue.
   - Halve capacity whenever, after a `dequeue()`, the number of items $\leq \frac{1}{4}$ of the current capacity, but *never* below the original (initial) capacity.
2. **Part B (7 pts):** Build a simple "Customer Service Desk" on top of your new dynamic queue by implementing three utility functions:
   - `void addCustomer(int customerID, int arrivalTime)` (2 pts)
   - `void serveNext(int currentTime)` (2 pts)
   - `double averageWaitTime(int currentTime)` (3 pts)

All code should reside in a single file `dynamic_queue_desk.cpp`. You must use the supplied `main()` (below) to demonstrate both resizing behavior and the customer desk functionality.

Part A: Dynamic Resizing (3 pts)

Below is a high-level description of how to modify the provided `Queue` class. You will fill in the **TODO** sections in the code.

Provided Class Skeleton

```
class Queue {
private:
    int init_capacity;      // original capacity
    int current_capacity;   // current array length
    int size;               // number of elements in queue
    int iFront;             // index of front element
    int iRear;              // index of next insertion
    int *items;             // pointer to dynamic array
    int getSize() const { return size; }    // needed for Part B

    // Reallocate the array to exactly newCapacity slots
    void resizeBuffer(int newCapacity) {
        // TODO: Allocate a new array of length newCapacity.
        // TODO: Copy existing elements (in FIFO order) into it.
        // TODO: Reset iFront and iRear accordingly.
        // TODO: Delete old array and update current_capacity.
    }
public:
    Queue(int cap);
    ~Queue();

    void enqueue(int x);
    int dequeue();
    int peek();
    void print();
};
Queue::Queue(int cap) {
    // TODO: If cap <= 0, set cap = 1 or exit with error.
    // TODO: Store init_capacity = cap and current_capacity = cap.
    // TODO: Set size = 0, iFront = 0, iRear = 0.
    // TODO: Allocate items = new int[current_capacity].
}
Queue::~Queue() {
    // TODO: Delete[] items.
}
void Queue::enqueue(int x) {
    // TODO: If size == current_capacity, print doubling message and call resizeBuffer(current_
    // TODO: Insert x at items[iRear], advance iRear circularly, increment size.
}
int Queue::dequeue() {
    // TODO: If size == 0, print error and return sentinel (e.g., -999999).
    // TODO: Otherwise, remove items[iFront], advance iFront circularly, decrement size.
    // TODO: If size <= current_capacity/4 and current_capacity/2 >= init_capacity,
    //       print halving message and call resizeBuffer(max(init_capacity, current_capacity/2)
    // TODO: Return the dequeued value.
    return -999999; // placeholder
}
```

Constructor & Destructor (1 pt)

- In `Queue::Queue(int cap)`:
- In `Queue::`$\sim Queue()$:

  - Call `delete[] items` and set `items = nullptr` (optional).

enqueue(int x) with Growth (1 pt)

dequeue() with Shrink (1 pt)

Part B: Customer Service Desk (7 pts)

Use a global pointer `Queue* desk` (initialized in `main()`) and a 1001-element array `arrivalTimeMap` to store each customer's arrival time (IDs range from 1 to 1000).

```
// Global data for Part B
static Queue* desk = nullptr;
static int arrivalTimeMap[1001];  // arrivalTimeMap[ID] = arrivalTime
// 2 pts: Enqueue customer + record arrival
void addCustomer(int customerID, int arrivalTime) {
    // TODO (English):
    // TODO: Enqueue customerID into desk and record arrivalTimeMap[customerID].
    // TODO: Print confirmation message.
}
// 2 pts: Dequeue next & print service
void serveNext(int currentTime) {
    // TODO: Dequeue next customer from desk (if any), compute wait = currentTime  arrivalTime

}
// 3 pts: Compute average wait among all still-waiting customers
double averageWaitTime(int currentTime) {
    // TODO: If desk is empty, return 0.0.
    // TODO: Otherwise, sum (currentTime  arrivalTime) for each waiting customer and return the
    return 0.0; // placeholder
}
```

Finally, include the following `main()` exactly as shown, to demonstrate both parts.

Grading Rubric (Total 10 pts)

- **Part A: Dynamic Resizing (3 pts)**

- Constructor & Destructor: **1 pt**
  - Correctly set `init_capacity`, `current_capacity`, initialize indices, allocate `items`.
  - Properly `delete[]` in destructor.
- `enqueue(int x)` with Growth: **1 pt**
  - Detect full, print doubling message, call `resizeBuffer(old*2)`.
  - Then insert element correctly.
- `dequeue()` with Shrink: **1 pt**
  - Detect empty, print error + return sentinel.
  - Remove front, adjust indices.
  - If usage $\leq \frac{1}{4}$ and can shrink, print halving message, call `resizeBuffer(...)`.

- **Part B: Customer Service Desk (7 pts)**
  - `addCustomer(...)`, 2 pts
    - Enqueue ID, record arrivalTime, print confirmation.
  - `serveNext(...)`, 2 pts
    - Dequeue next (if any), compute wait time, print service.
  - `averageWaitTime(...)`, 3 pts
    - If empty, return 0.0.
    - Otherwise, sum currentTime - arrivalTime for each waiting customer and divide by count.

Submission Instructions

Submit a single file named `dynamic_queue_desk.cpp` containing:

1. The modified `Queue` class with dynamic resizing implemented.
2. The three functions: `addCustomer()`, `serveNext()`, and `averageWaitTime()`.
3. The supplied `main()` exactly as shown.