Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering

Assignment 3 | Queue

CSE106 — DSA I

1 Queue ADT

Overview

We define the Queue ADT as a C++ *abstract class*.

We will start with an abstract class, which defines what features our queue should have. In our case we will define the following operations: `enqueue`, `dequeue`, `clear`, `size`, `front`, `back`, `empty` and `toString`. These methods present how any implementation of a queue should behave so that the user of the queue can use it without knowing the details of the implementation.

We already provided you with the abstract class `Queue` in the file `queue.h`. It contains the required methods definitions. You don't need to modify this file. So, for this section you don't need to write any code.

Implmentation Notes

- You have to use C++ for this assignment.
- You have been given template code,
  YOU SHOULD STRICTLY EDIT TEMPLATE FILES ONLY.
- **DO NOT MODIFY UNLESS THERE IS A TODO COMMENT**.
- You must use **only your own queue implementations** for all the tasks.
- No built-in data structures (e.g. STL) allowed.

Method Specifications

The most fundamental operations first!

`void enqueue(int item)`

Description: Adds a new item to the back of the queue.

Parameters: `item` — element to enqueue (`int`) **Returns:** None

```cpp
Queue* q = new ListQueue();
q->enqueue(5);
q->enqueue(15); // Queue: 5, 15 (15 is at back)
cout << q->toString() << endl;      // Output: <5, 15|
delete q;
```

`int dequeue()`

Description: Removes and returns the item at the front of the queue.

Parameters: None **Returns:** Front item (`int`)

```cpp
Queue* q = new ListQueue();
q->enqueue(8);
cout << q->toString() << endl;  // Output: <8|
int val = q->dequeue();         // val = 8
cout << q->toString() << endl;  // Output: <|
delete q;
```

`void clear()`

Description: Empties the queue, removing all elements. After this operation, the queue becomes logically empty.

Parameters: None **Returns:** None

```cpp
Queue* q = new ListQueue();
q->enqueue(10);
q->enqueue(20);
cout << q->toString() << endl;  // Output: <10, 20|
q->clear();                     // Queue is now empty
cout << q->toString() << endl;  // Output: <|
delete q;
```

`int size() const`

Description: Returns the number of elements currently in the queue.

Parameters: None **Returns:** Integer representing queue size

```cpp
Queue* q = new ListQueue();
q->enqueue(3);
q->enqueue(7);
cout << q->toString() << endl; // Output: <3, 7|
int len = q->size();           // len = 2
delete q;
```

`int front() const`

Description: Returns the front item without removing it from the queue.

Parameters: None **Returns:** Front item (`int`)

```cpp
Queue* q = new ListQueue();
q->enqueue(42);
cout << q->toString() << endl;  // Output: <42|
int f = q->front();             // f = 42
cout << q->toString() << endl;  // Output: <42| (unchanged)
delete q;
```

`int back() const`

Description: Returns the back (rear) item without removing it from the queue.

Parameters: None **Returns:** Back item (`int`)

```cpp
Queue* q = new ListQueue();
q->enqueue(42);
q->enqueue(99);
cout << q->toString() << endl; // Output: <42, 99|
int b = q->back();             // b = 99
cout << q->toString() << endl; // Output: <42, 99| (unchanged)
delete q;
```

`bool empty() const`

Description: Checks if the queue is empty.

Parameters: None **Returns:** `true` if the queue is empty, `false` otherwise.

```
Queue* q = new ListQueue();
cout << q->toString() << endl;  // Output: <|
bool isEmpty = q->empty();      // true
q->enqueue(10);
cout << q->toString() << endl;  // Output: <10|
isEmpty = q->empty();           // false
delete q;
```

`string toString() const`

Description: Converts queue to a string representation.

Parameters: None **Returns:** `String`

```
Queue* q = new ListQueue();
q->enqueue(10);
q->enqueue(20);
q->enqueue(30);
// Example of toString() return format:
cout << q->toString() << endl; // Output: <10, 20, 30|
delete q;
```

2 Task 1: ArrayQueue Implementation

Overview

We will implement the Queue ADT using a dynamically allocated array.

You must have used arrays at this point. Whenever we need to store more than one item of the same type, we use an array. Arrays are great, but they have some limitations. They are fixed in size, meaning you have to decide how many items you want to store before you create the array. However, there is a way to overcome this limitation by using a **dynamic array**; dynamically allocating an array of a size that you can change at runtime. And if we still need more space, we can create a new array and copy all the elements from the old array to the new one. This is called **resizing** the array.

In this assignment, when the queue becomes full (i.e., reaches its maximum capacity), we will resize it to double its capacity and copy all elements in order. When it is less than 25% of the current capacity, we will resize it to half its capacity (but not less than 2). This strategy balances memory usage and performance by minimizing resizing overhead while maintaining space efficiency.

Data Member Specifications

Private Data Members

- `int* data` — *Dynamic array pointer* that stores elements in contiguous memory
- `int capacity` — *Maximum size* of the array; doubles when full and shrinks when sparsely populated
- `int front_idx` — *Index* of the current front item in the queue
- `int rear_idx` — *Index* of the current rear item in the queue

Method Specifications

You are required to implement an extra method in ArrayQueue.

`int getCapacity() const`

Description: Returns the number of elements the underlying array can currently hold without resizing.

Returns: Integer representing the current capacity of the dynamic array.

```cpp
ArrayQueue* q = new ArrayQueue(20); // Initial capacity 20
q->enqueue(1);
q->enqueue(2);
cout << q->getCapacity() << endl; // Output: 20
delete q;
```

Constructor and Destructor

`ArrayQueue(int capacity = 10)`

Description: Creates a new ArrayQueue with the specified initial capacity.

Parameters: `capacity` - Initial capacity of the array (default: 2) **Returns:** None

```cpp
ArrayQueue* queue = new ArrayQueue(20); // Creates queue with initial
    capacity of 20
ArrayQueue* defaultQueue = new ArrayQueue(); // Creates queue with
    default capacity (2)
```

`~ArrayQueue()`

Description: Destroys the ArrayQueue and releases all allocated memory.

Parameters: None **Returns:** None

```cpp
ArrayQueue* queue = new ArrayQueue();
delete queue; // Destructor called, memory freed
```

Implementation Requirements

ArrayQueue Requirements

- ✓ Implement the methods in `arrayqueue.cpp` according to the specifications
- ✓ Implement proper memory management:
  - Deallocate the array when the queue is cleared or destructed
  - Ensure no memory leaks occur during any operation
- ✓ Implement dynamic resizing:
  - When the queue becomes full, create a new array with double the current capacity and copy elements in logical order
  - When the queue's size drops to 25% or less of its capacity, shrink the array to half its current capacity (but not less than 10)
- ✓ Implement circular buffer behavior using modulo arithmetic for `front` and `rear` indices
- ✓ Gracefully handle edge cases (e.g., dequeue from empty queue)
- ✓ Time complexity requirements:
  - `enqueue, dequeue`: O(1) amortized time
  - `front, back, size, empty`: O(1) time

3 Task 2: ListQueue Implementation

Overview

We will now implement the Queue ADT using a linked list.

You probably also know about **linked lists**. They are a data structure where each element (node) contains a value and a pointer to the next node. Linked list-based queues offer several advantages: they can grow dynamically without requiring resizing operations, and they efficiently use memory as each element is allocated individually. You don't need to resize the entire structure when adding or removing elements.

The ListQueue maintains pointers to both the head (front of the queue) and tail (rear of the queue) nodes, and keeps track of the current size. When new elements are enqueued, they are added to the back (tail) of the list. When elements are dequeued, they are removed from the front (head). This allows efficient O(1) time for both enqueue and dequeue operations without needing to traverse the list.

Data Member Specifications

Private Data Members and Structures

- `struct Node` — *Internal structure* representing each element:
  - `int data` — *Value* stored in the node
  - `Node* next` — *Pointer* to the next node in the sequence
  - `Node(int value, Node* next)` — *Constructor* to initialize a new node
- `Node* front_node` — *Front pointer* marking the front of the queue
- `Node* rear_node` — *Rear pointer* marking the back of the queue
- `int current_size` — *Element counter* tracking the total items in the queue

Constructor and Destructor

`ListQueue()`

Description: Creates a new empty ListQueue.

Parameters: None **Returns:** None

```
ListQueue* queue = new ListQueue(); // Creates an empty linked list
    queue
```

~ListQueue()

Description: Destroys the ListQueue and releases all allocated memory for nodes.

Parameters: None   **Returns:** None

```
ListQueue* queue = new ListQueue();
// ... use the queue
delete queue; // Destructor called, all nodes are deallocated
```

Implementation Requirements

ListQueue Requirements

- ✓ Implement the methods in `listqueue.cpp` according to the specifications
- ✓ Implement proper memory management:
  - Deallocate all nodes when the queue is cleared or destructed
  - Ensure no memory leaks occur during any operation
- ✓ Gracefully handle edge cases
- ✓ Time complexity requirements:
  - `enqueue`, `dequeue`, `front`, `back`, `size`, `empty`: O(1) time
- ✓ New nodes should be inserted at the tail and removed from the head for efficient queue operations

4 Task 3: Simulating Hospital Vaccination Queues

Overview

To test if your queue implementation works correctly, you need to simulate Hospital Vaccination Queues.

You are managing a vaccination center in a hospital. There are two booths where patients queue up to get vaccinated. However, due to crowd control, emergency handling, and workflow balancing, the queue system can dynamically change.
How People Behave:

- Patients arrive at specific times and enter a queue.
- A patient might leave the queue (to use the restroom, for phone calls, etc.).
- When a patient returns, they are reinserted at the end of a queue. So, they are treated like a new patient.
- Booths may be merged into one centralized queue during rush hours.
- Later, the merged queue might be split again for faster service.

Problem Description

There are initially two queues **Q1** and **Q2**. Later possibly merged into one queue **Q**, or split back into **Q1**, **Q2**. You need to perform **N** operations. After each operation print the content of **Q1**, **Q2** and **Q**.

Operation Types

Arrival

- **Input format:** `1 id timestamp`
- A new patient with ID **id** arrives at timestamp. Where $\mathbf{1} \leq \mathbf{id} \leq \mathbf{10000}$ and unique for each patient.
- The patient is added to either **Q1** or **Q2** based on the following rules:
  - If both queues are empty, select one randomly.
  - If one queue is empty, add the patient to that queue.
  - If both queues are non-empty, compare the timestamp of the last patient (i.e., `back()`) in each queue, and insert into the queue whose back has a smaller timestamp.

Leave

- **Input format:** `2 id timestamp`
- Patient with ID **id** leaves the queue at timestamp.
- The patient is removed from the queue. Note: You need to implement this using **O(1)** extra space.

Merge

- **Input format:** `3`
- If two queues exist (**Q1** and **Q2**), merge them into one queue.
- Merge order is based on timestamp of arrival (earlier comes first). Note: You can use an extra array to track the timestamp of each patient.
- After merging, all future arrivals go to the merged queue only.

Split

- **Input format:** `4`
- If the system currently has only one merged queue, split it into two queues:
  - People at odd positions (From the front of queue) go to **Q1**. For example, 1st person goes to **Q1**.
  - People at even positions go to **Q2**.
- After splitting, future arrivals again go to either **Q1** or **Q2**.

Departure

- **Input format:** `5`
- If there are two non-empty queues, select a queue at random (e.g., **Q1**, **Q2**). Remove the person at the front of that queue.
- If there is only one non-empty queue, remove the person at the front of that queue.
- If all queues are empty, do nothing.

Implementation Requirements

General Requirements

- ✓ Implement this task in `main.cpp`.
- ✓ Use `ArrayQueue` to implement the merged queue **Q**, and `ListQueue` for **Q1** and **Q2**.
- ✓ To ensure deterministic behavior, use the `randomQueue()` function provided in `main.cpp` when selecting a queue at random.
  - This function returns either `1` or `2`. Return value `1` means select **Q1**, and `2` means select **Q2**.
- ✓ **Important:** Avoid unnecessary calls to `randomQueue()` as this may cause mismatches with the expected sample output.
- ✓ Properly deallocate all memory before the program exits to prevent memory leaks.

Sample Input-Output

Following table shows a sample input and output. You need to follow this output format. **Timestamps are strictly in increasing order.**

Sample Input

```
10
1 101 1
1 102 2
1 103 3
2 102 4
3
2 103 5
1 105 8
4
1 106 9
5
```

Sample Output

```
Operation 1 (Arrival 101 1): randomQueue() called
Q1: <|
Q2: <101|
Q : <|
Operation 2 (Arrival 102 2):
Q1: <102|
Q2: <101|
Q : <|
Operation 3 (Arrival 103 3):
Q1: <102|
Q2: <101, 103|
Q : <|
Operation 4 (Leave 102 4):
Q1: <|
Q2: <101, 103|
Q : <|
Operation 5 (Merge):
Q1: <|
Q2: <|
Q : <101, 103|
Operation 6 (Leave 103 5):
Q1: <|
Q2: <|
Q : <101|
Operation 7 (Arrival 105 8):
Q1: <|
Q2: <|
Q : <101, 105|
Operation 8 (Split):
Q1: <101|
Q2: <105|
Q : <|
Operation 9 (Arrival 106 9):
Q1: <101, 106|
Q2: <105|
Q : <|
Operation 10 (Departure): randomQueue() called
Q1: <106|
Q2: <105|
Q : <|
```

5 Skeleton Code

You will receive a zip file containing the skeleton code. The folder structure is as follows:

```
|-- main.cpp
|-- listqueue.cpp
|-- arrayqueue.cpp
|-- queue.h
|-- test_input_1.txt
|-- test_output_1.txt
|-- test_input_2.txt
|-- test_output_2.txt
```

You are required to edit only `main.cpp`, `arrayqueue.cpp` and `listqueue.cpp`.

6 Compiling and Running

Compiling Commands

```
# Compile the queue implementations and main program
g++ -std=c++11 main.cpp arrayqueue.cpp listqueue.cpp -o main
# Run the main program
./main
```

Notes on Compilation

- Be sure to include all source files in the compilation command
- The `-std=c++11` flag ensures compatibility with the C++11 standard
- The `-o main` flag specifies the output executable name

7 Submission

Submission Process

1. Create folder named: `<your_id>` (e.g., `2305xxx`)
2. Include all the files provided in the skeleton code.
3. Compress as `<your_id>.zip` and upload to Moodle

Deadline: June 13, 2025, 10:00 PM

WARNING

- ✗ Do not copy your code from your peers or other sources;
  Will result in **-100% penalty**.
- ✗ Do not use ChatGPT/Copilot.
- ✗ No submissions will be accepted after the deadline. Submit on time.

Evaluation Policy

Task 1	
ArrayQueue Functions	30%
ArrayQueue Resize	5%
Task 2	
ListQueue Functions	30%
Task 3	
Operation implementations	30%
Proper memory deallocation and printing	5%

Good luck with your implementation!