CC-213L

Data Structures and Algorithms

Laboratory 05

Queue ADT

Version: 1.0.0

Release Date: 14-10-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Queue ADT
- PriorityQueue
- Double Ended Queue
- Applications of Queue ADT

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Madiha Khalid	madiha.khalid @pucit.edu.pk
Teacher Assistants	Muhammad Nabeel	bitf20m009@pucit.edu.pk
	Muhammad Subhan	bcsf20a033@pucit.edu.pk

Background and Overview:

Queue:

A queue is a linear data structure that operates on the principle of First-In-First-Out (FIFO). Think of it as similar to a real-world queue, like people standing in line. In a queue, elements are added at the back, a process known as enqueueing, and removed from the front, referred to as dequeuing. This ensures that the element added first will be the first one to be removed. Let's explore this with an analogy:

Adding an Element to the Queue: Imagine a queue as a line of people waiting for something, like ordering food at a food truck. There are already a few people in the line, and you're joining at the back. When you join, you become the last person in line. Similarly, when you add an element to a queue, it goes to the end of the queue.

For example, if the queue contains numbers like [3, 7, 10], and you want to add the number 15, it will be placed at the end of the queue. So, the updated queue would look like this: Queue: [3, 7, 10, 15]

Removing an Element from the Queue: Continuing with the food truck analogy, let's say it's your turn to order. You step forward, place your order, and then move away from the line. Now, the person who was directly behind you moves up to the front to place their order. In a queue, when you remove an element, the element at the front is taken out, and the element that was behind it becomes the new front.

For instance, if the queue is [3, 7, 10, 15], and you remove the first element, which is 3, the updated queue will look like this: Queue: [7, 10, 15] The element 7, originally second in line, is now at the front.

Implementation Details: To understand how a queue operates, envision a row of boxes and two individuals, Mr. Rear and Mr. Front. Each box represents a spot in the queue where items can be placed.

Mr. Rear stands where the last filled box is located. He knows precisely where the last item was positioned. When you want to add a new item to the line of boxes, you simply ask Mr. Rear. He will instruct you on which box to place the new item in. Once you've added the item, Mr. Rear steps forward, prepared to guide the placement of the next item.

Mr. Front stands where the first filled box is. His role is to inform you which item should be taken out next. When you need to use or remove an item from the line, you ask Mr. Front. He points to the item in the first box and indicates that this is the one you should remove. After you've taken it out, Mr. Front moves to the next box, ready for the next time you need to remove an item.

Working Together: The collaboration between Mr. Rear and Mr. Front ensures that the queue remains in order. As you add new items, Mr. Rear assists you in placing them correctly at the end of the line, effectively extending the queue. When you need to use an item, Mr. Front indicates which item is up next, maintaining the First-In-First-Out (FIFO) principle of the queue.

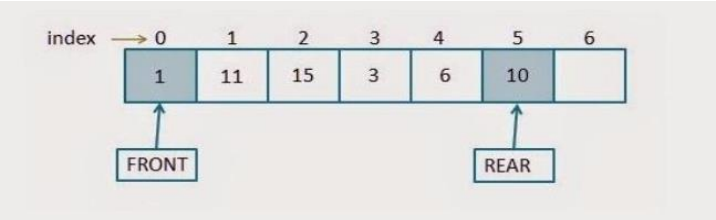

Figure 1(Queue)

When Rear reaches the end of the queue: Let's consider a scenario with 10 boxes. Initially, we placed items in all 10 boxes. Now, we've removed three items, specifically from boxes 1, 2, and 3. Mr. Front is now positioned at box number 4. The filled boxes are 4, 5, 6, 7, 8, 9, and 10, with 1, 2, and 3 being

empty. Now, the question arises: What should we do when we want to add a new item? Should we increase the number of boxes to accommodate new items? But what about the empty boxes 1, 2, and 3?

Instead of expanding the number of boxes, here's a more efficient approach: Place the new item in box number 1 and move Mr. Rear from box number 10 to box number 1. This allows us to utilize the previously empty boxes 1, 2, and 3 for the new items.

Key Takeaways:

- When an item is removed, the front is advanced, and the number of elements is decreased.
- When a new item is inserted, the rear is advanced, and the number of elements is incremented.
- The array is resized when the number of elements matches the capacity.
- The data is stored between the front and rear pointers, creating a continuous array with the front and rear ends connected.

To visualize the underlying data structure of a queue, imagine an array where one end is connected to the other, forming a circular structure.

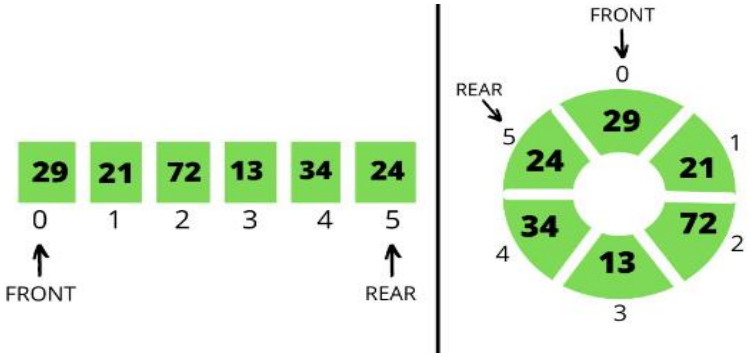

Figure 2(Circular Queue)

Queue Applications:

Queue data structures are widely used in various applications and scenarios where managing data and processes in a first-in, first-out (FIFO) manner is essential. Here are different types of applications where the queue data structure is commonly employed.

Operating Systems and Task Management:

1. Task Scheduling: Operating systems use queues to schedule and prioritize tasks, ensuring that tasks are executed based on their priority and order of arrival.
2. Thread Management: Multithreaded applications use queues to manage and coordinate threads for efficient task execution.
3. Real-Time Systems: Queues play a crucial role in real-time systems, ensuring that events are processed in the order they occur.

Document and Data Management:

1. Print Queues: Print jobs are managed using queues, allowing documents to be printed in the order they are received.
2. Print Spooling: Print spoolers use queues to manage and prioritize print jobs.
3. Buffer Management: Queues are used to manage data buffers, preventing overflow and ensuring controlled data transmission.
4. Data Streaming: In data streaming applications, queues are used to buffer and process data, ensuring a smooth flow of information.

Customer Service and Communication:

1. Call Center Systems: Call centers use queues to manage incoming customer calls and direct them to available agents.
2. Request Handling: Web servers and application servers use queues to manage incoming requests, such as HTTP requests.
3. Message Queues: Message queuing systems enable communication between distributed applications and services in a decoupled manner.

E-commerce and Inventory Management:

1. Order Processing: E-commerce websites use queues to manage incoming orders, ensuring they are processed in sequence.
2. Inventory Management: Retail and supply chain systems use queues to manage inventory, ensuring efficient restocking and shipping.

Data Structures and Algorithms:

1. Data Structures: Queues serve as fundamental data structures and are often used as building blocks for more complex data structures like double-ended queues (deques).
2. Breadth-First Search (BFS): BFS traversal in graph algorithms utilizes a queue to explore nodes level by level.
3. Simulation and Modeling: Queues are used in simulations to model processes, such as traffic flow, customer service, and manufacturing.

Transportation and Traffic Control:

1. Traffic Management: Traffic control systems use queues to manage the flow of vehicles and prevent congestion.

These categories demonstrate the widespread utility of the queue data structure in various domains, emphasizing its role in maintaining the order and priority of tasks and data elements.

Activities

Pre-Lab Activities:

Task 01 Implementation of Queue

Queues are fundamental data structures used in various algorithms and applications. They can be created using arrays or linked lists. However, for our current focus, we will concentrate on implementing a queue using arrays, as we haven't covered linked lists yet.

To gain a deeper understanding of how the queue data structure operates, you will be tasked with implementing the **MyQueue** Abstract Data Type (ADT) in C++. This assignment is part of your grading and will be assessed online in the classroom. It's essential to ensure that your implementation is accurate, handles all possible scenarios and corner cases, as this queue will be utilized in solving problems during in-lab tasks. Any issues in your queue ADT may pose challenges when working on problems that involve queue data structures, as the use of the STL library is not permitted.

```
template<typename T>
class myQueue {
   int rearIndex;                    // Index of the rear element
   int frontIndex;                   // Index of the front element
   int queueCapacity;                // Maximum capacity of the queue
   int numberOfElements;             // Number of elements in the queue
   T * queueData;                    // Array to store queue elements
   void resize(int newSize);         // Private helper method for resizing the queue

public:
   myQueue() {
      rearIndex = frontIndex = numberOfElements = queueCapacity = 0;
      queueData = nullptr;
   }

   myQueue(const MyQueue<T> &);                          // Copy constructor
   myQueue<T> & operator=(const MyQueue<T> &);           // Assignment operator
   ~MyQueue();                                           // Destructor

   void enqueue(const T element);    // Add an element to the back of the queue
   T dequeue();                      // Remove and return the front element
   T getFront() const;               // Get the front element without removing it
   bool isEmpty() const;             // Check if the queue is empty
   bool isFull() const;              // Check if the queue is full
   int size() const;                 // Get the current number of elements
   int getCapacity() const;          // Get the maximum capacity of the queue
   };
```

Task 02 Implementation of Priority Queue

A Priority Queue ADT is a data structure that combines the features of a queue with the added concept of priorities. In a Priority Queue, elements are enqueued based on their priority, and when dequeued, the element with the highest priority is removed first. This ADT is implemented using templates in C++ to allow flexibility with the data types.

```
template <class T>
class PriorityQueue {
        /***** define data members required to implement PriorityQueue *****/
```

```
public:
    PriorityQueue();                        // Constructor to initialize the priority queue
    ~PriorityQueue();                       // Destructor to free memory if necessary
    void enqueue(const T& data, int priority); // Function to enqueue an element with a given priority
    T dequeue();                            // Function to dequeue the element with the highest priority
    bool isEmpty() const;                   // Function to check if the priority queue is empty
    bool isFull() const;                    // Function to check if the priority queue is full
    int size() const;                       // Function to get the size of the priority queue
};
```

Explanation:

- The **PriorityQueue** class is designed using C++ templates to make it generic and capable of handling elements of different data types.
- The constructor initializes an empty priority queue.
- The destructor can be implemented to release any allocated memory when needed.
- The **enqueue** function adds an element with a specified data value and priority. Lower values of priority indicate higher priority.
- The **dequeue** function removes and returns the element with the highest priority from the queue.
- The **isEmpty** function checks whether the priority queue is empty.
- The **isFull** function checks whether the priority queue is full.
- The **size** function returns the current size of the priority queue.

Usage:

Here's how you can use the **PriorityQueue** ADT in C++ with different data types:

```
PriorityQueue<int> intQueue;
intQueue.enqueue(42, 2);
intQueue.enqueue(36, 1);
intQueue.enqueue(17, 3);

PriorityQueue<std::string> stringQueue;
stringQueue.enqueue("apple", 5);
stringQueue.enqueue("banana", 3);
stringQueue.enqueue("cherry", 7);

int highestPriorityInt = intQueue.dequeue();
std::string highestPriorityStr = stringQueue.dequeue();

// Now, highestPriorityInt contains 17 (highest priority)
// highestPriorityStr contains "cherry" (highest priority)
```

In-Lab Activities

Task 01 Double Ended Queue

Double-ended queue is almost similar to the queue that we discussed earlier with one key difference that it allows insertion and removal of elements from both ends. This means you can add elements not only to the back (rear) of the queue but also to the front. Similarly, you can remove elements not only from the front but also from the back. Imagine it as a line where people can join from back or front and can also leave from back or front.

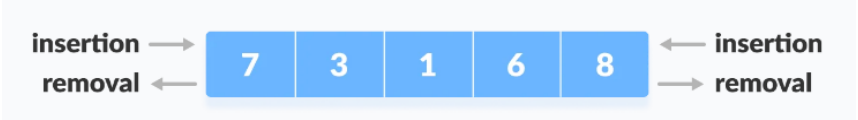

Figure 3(Double Ended Queue)

To insert an element at the front end:

1. Resize the deque if it's full.
2. If the front is equal to 0 or at the initial position:
   2.1. Move the front to point to the last index of the array.
3. Else:
   3.1. Decrement the front by 1.
   3.2. Push the current Value into Arr [front] = Value.
4. The rear remains the same.

To insert an element at the rear end:

1. Check if the deque is full.
2. If the rear is equal to the last index of the array (size - 1):
   2.1. Reinitialize rear to 0.
3. Else:
   3.1. Increment rear by 1.
   3.2. Push the current Value into Arr [rear] = Value.
4. Front remains the same.

To delete element from the rear end:

1. Check if the deque is empty or not.
2. If the deque has only one element:
   2.1. Set front to -1.
   2.2. Set rear to -1.
3. Else, if the rear points to the first index of the array:
   3.1. Move rear to point to the last index (size - 1) of the array.
4. Otherwise:
   4.1. Decrement rear by 1 (rear = rear - 1).

To delete an element from the front end:

1. Check if the deque is empty or not.
2. If the deque has only one element:
   2.1. Set front to -1.
   2.2. Set rear to -1.
3. Else, if front points to the last index of the array:
   3.1. Move front to point to the first index of the array (front = 0).
4. Otherwise:
   4.1. Increment front by 1 (front = front + 1).

Let's demonstrate these operations with a dry run example: Suppose we have an empty deque initially. We perform the following operations:

1. InsertFront(5)
    - Deque: (5)
2. InsertRear(10)
    - Deque: (5, 10)
3. InsertFront(2)
    - Deque: (2, 5, 10)
4. RemoveFront()
    - Deque: (5, 10)
5. RemoveRear()
    - Deque: (5)
6. InsertRear(7)
    - Deque: (5, 7)

This example illustrates the behavior of a deque as we insert and remove elements from both the front and rear ends.

Task 02: FIFO Page replacement Algorithm

In an operating system, "paging" is a method to manage the computer's memory more efficiently. Think of it like organizing a big book into smaller pages. Each page has a specific size, and the computer's memory is divided into these fixed-size pages. Here's how it works: When a program or application runs on your computer, it needs space in memory to store its data and instructions. Instead of finding one large, continuous space in memory (which can be tricky), paging divides memory into these manageable pages. Each page is like a little box that can hold a certain amount of data. When a program needs memory, the operating system assigns it one or more of these pages. If a program needs more space, it gets more pages. Paging makes memory management more flexible and efficient. It's like having lots of sticky notes (pages) that can be moved around in a notebook (memory) to accommodate different things you're working on. This way, the computer can use its memory more effectively and keep everything organized. In simple terms, paging in an operating system is like using sticky notes to organize and manage the computer's memory, making it easier to handle multiple tasks and programs.

Here's how it works: When a new page is requested, and it's not already in the computer's memory, we have what's called a "page fault." To handle this, the computer's operating system replaces one of the pages that's already in memory with the new one.

Different page replacement algorithms suggest different ways to decide which page should be replaced. But they all share a common goal: to reduce the number of page faults, which slow down the computer.

One of the simplest page replacement algorithms is called "First In First Out" or FIFO. In this algorithm, the operating system keeps track of all the pages currently in memory in a queue, which is like a line of pages waiting their turn. The oldest page, the one that came in first, is at the front of the queue.

When a new page needs to be loaded, the operating system selects the page at the front of the queue to be replaced.

Here's an example: Let's say we have a sequence of page references: 1, 3, 0, 3, 5, 6, and we have space for 3 pages in memory. At first, all the memory slots are empty. So, when pages 1, 3, and 0 are requested by a program they come in, they are put into the empty slots, resulting in 3 page faults.

When again page 3 is requested by a program, it's already in memory, so there's no **page fault**, hence it results into a **page hit**.

But then, page 5 is requested by a program, and it's not in memory, so operating system loads it from the hard disk and it replaces the oldest page in memory, which is page 1. This causes another page fault.

Finally, page 6 is requested by the program, and since it's also not in memory, it loaded by the operating system which replaces the oldest page remaining in memory, which is page 3, causing 1 more page fault.

So, in total, there were 5 page faults and one page hit in a sequence of six page requests by a program namely 1, 3, 0, 3, 5, 6 with a queue size (available memory) of three pages.

Your task is to create a program that simulates this FIFO-based page replacement algorithm. In your program, you'll ask the user for the size of the memory queue and then keep asking the user for the page numbers they want to load into the queue. You'll need to determine whether each page is a hit (already in memory) or a fault (needs to be loaded), and you'll display the queue after each operation. The program should continue until the user decides to quit, and then it should display the total number of page faults and page hits.

Task 03: Possible Binary Numbers

Given a number N, your task is to use a queue data structure to print all possible binary numbers with decimal values from 1 to N. For instance, if the input is 4, the expected output would be: 1, 10, 11, 100. Here's an intriguing approach that employs a queue data structure to achieve this:

1. Begin with an empty queue of strings.
2. Add the first binary number, "1," to the queue.
3. Remove and print the front element of the queue.
4. Extend the front item by appending "0" to it and place it back in the queue.
5. Extend the front item by appending "1" to it and place it back in the queue.
6. Repeat steps 3 to 5 until you reach the desired end value.

Example Simulation of Queue:

Let's perform a dry run of the given task for N = 4 to see how it generates binary numbers using a queue:

1. Initialize an empty queue: **Queue = []**
2. Add the first binary number, "1," to the queue: **Queue = ["1"]**
3. Remove and print the front element of the queue:
    - Printed: "1"
    - Queue: **Queue = []**
4. Extend the front item by appending "0" to it and place it back in the queue: **Queue = ["10"]**
5. Extend the front item by appending "1" to it and place it back in the queue: **Queue = ["10", "11"]**
6. Repeat steps 3 to 5 until you reach the desired end value.

Iteration 1:

- Remove and print the front element of the queue:
    - Printed: "10"
    - Queue: **Queue = ["11"]**
- Extend the front item by appending "0" to it and place it back in the queue: **Queue = ["11", "100"]**
- Extend the front item by appending "1" to it and place it back in the queue: **Queue = ["11", "100", "101"]**

Iteration 2:

- Remove and print the front element of the queue:
    - Printed: "11"
    - Queue: **Queue = ["100", "101"]**

- Extend the front item by appending "0" to it and place it back in the queue: **Queue = ["100", "101", "110"]**
- Extend the front item by appending "1" to it and place it back in the queue: **Queue = ["100", "101", "110", "111"]**

Iteration 3:

- Remove and print the front element of the queue:
  - Printed: "100"
  - Queue: **Queue = ["101", "110", "111"]**
- Terminate the loop as 100 which equivalent to 4 is printed.

An example of the queue is given as follows

Front	**Queue**				
1					
10	11				
11	100	101			
100	101	110	111		
101	110	111	1000	1001	

Print
1
10
11
100
101

Post-Lab Activities

Task 01: Least Recently Used (LRU) Cache Replacement Algorithm

Computers utilize cache memory to temporarily store frequently accessed data. This is an efficient method to swiftly retrieve data, as cache memory offers lightning-fast access. However, cache memory has a limited capacity, and it becomes essential to manage the removal of old data to make space for new data.

A Least Recently Used (LRU) Cache provides a solution for this. It organizes data items based on their usage, ensuring that you can quickly identify which data hasn't been used for the longest time. The LRU Cache then efficiently removes the least recently used data to accommodate new information. Think of it as a page replacement algorithm.

Here are the key aspects of an LRU Cache:

- An LRU Cache behaves like a special type of fixed-sized queue where replacements occur at the rear, and the most recently used pages are positioned at the front.
- When the processor requires a page not present in the LRU Cache, it's inserted into the queue at the front.
- If the LRU Cache is already full, we must replace the least recently accessed page with the needed one.
- If the required page is already present somewhere in the Cache, it's moved to the front.

Your task is to implement an LRU Cache with all the essential functions. Additionally, create a **printLRUCache()** function to display the current state of the Cache. In the **main()** function, you'll prompt the user to specify the Cache size and then repeatedly ask for the page number they want to place in the LRU cache. The program will manage these operations and show the state of the LRU cache after each step. The program should continue until the user decides to exit.

Example Simulation of Queue:

Let's walk through a dry run example of the LRU Cache implementation based on the problem statement: Suppose we have an LRU Cache with a size of 3, and we want to interact with it using a series of page requests. Here's how it would work:

1. We initiate the LRU Cache:
   Cache Size: 3
   Current Cache State: []
2. We add a page, say page 5. Since the cache is empty, it's inserted at the front:
   Page Request: 5
   Cache State: [5]
3. We add another page, say page 2. It goes to the front as well:
   Page Request: 2
   Cache State: [2, 5]
4. Now, we request page 7, which isn't in the cache. So, we insert it at the front:
   Page Request: 7
   Cache State: [7, 2, 5]
5. Page 5 is requested again, so we move it to the front since it's now more recently used:
   Page Request: 5
   Cache State: [5, 7, 2]
6. We request page 9, which isn't in the cache. We remove page 2 from the rear and page 9 insert at the front:
   Page Request: 9
   Cache State: [9, 5, 7]

7. If we now request page 5 again, it's already in the cache, so we move it to the front:
   Page Request: 5
   Cache State: [5, 9, 7]
8. Finally, if we request page 3, which isn't in the cache, page 7 is removed from the rear and we insert page 3 at the front:
   Page Request: 3
   Cache State: [3, 5, 9]

This example demonstrates how the LRU Cache operates, ensuring that the most recently used pages are at the front and the least recently used ones are removed when the cache is full.

Task 02: Stack Using Queue

A Stack operates on the Last in First Out (LIFO) principle, meaning that the element added last is the first to be removed. In this task, we aim to create a Stack using Queues. To achieve this, we'll employ two queues and configure them in a manner where the pop operation is equivalent to dequeue, but the push operation is somewhat more resource-intensive.

Considering that Queues follow a First in First Out (FIFO) structure, where the element added first is the first to be removed, our push operation needs to ensure that when a pop operation occurs, the Stack always retrieves the most recently added element. This approach ensures that the Stack adheres to the LIFO principle. You can use the Queue Abstract Data Type (ADT) you've already implemented to accomplish this.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** **[40 marks]**
  - Task 01: Implementation of Queue [20 marks]
  - Task 01: Implementation of PriorityQueue [20 marks]
- **Division of In-Lab marks:** **[60 marks]**
  - Task 01: Double Ended Queue [20 marks]
  - Task 02: FIFO Page Replacement Algorithm [20 marks]
  - Task 03: Possible Binary Numbers [20marks]
- **Division of Post-Lab marks:** **[20 marks]**
  - Task 01: LRU Page Replacement Algorithm [10 marks]
  - Task 02: Stack using Queue [10 marks]

References and Additional Material:

Queue Data Structure

https://www.geeksforgeeks.org/queue-data-structure/

Lab Time Activity Simulation Log:

- Slot – 01 – 02:00 – 00:15: Class Settlement
- Slot – 02 – 02:15 – 02:30: In-Lab Task 01
- Slot – 03 – 02:30 – 02:45: In-Lab Task 01
- Slot – 04 – 02:45 – 03:00: In-Lab Task 02
- Slot – 05 – 03:00 – 03:15: In-Lab Task 02
- Slot – 06 – 03:15 – 03:30: In-Lab Task 02
- Slot – 07 – 03:30 – 03:45: In-Lab Task 03
- Slot – 08 – 03:45 – 04:00: In-Lab Task 03
- Slot – 09 – 04:00 – 04:15: In-Lab Task 03
- Slot – 10 – 04:15 – 04:30: In-Lab Task 03
- Slot – 11 – 4:300 – 04:45: In-Lab Task 03
- Slot – 12 – 04:45 – 05:00: Discussion on Post-Lab Task