CC-213L

Data Structures and Algorithms

Laboratory 05

Queue ADT

Version: 1.0.0

Release Date: 16-10-2024

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 following.

Teachers:		
Course/Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistants	Tahir Mustafvi	bcsf20m018@pucit.edu.pk
	Maryam Rasool	bcsf21m055@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.

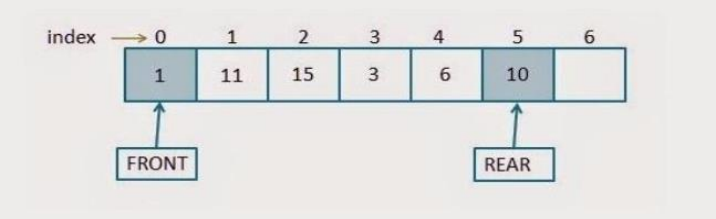

Figure 1(Queue)

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.

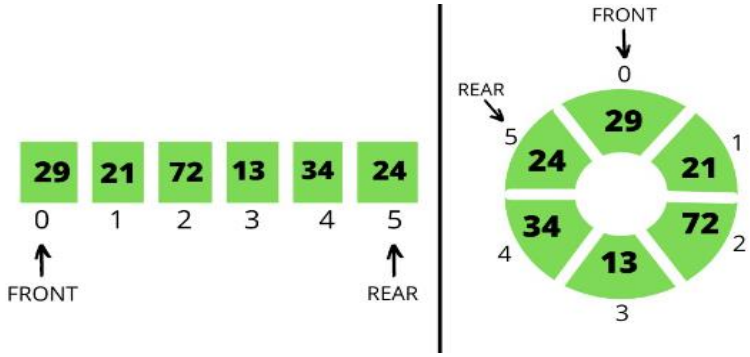

Figure 2(Circular Queue)

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 Linear 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. You are supposed to add time and space complexity of each function in comments.

```
template<typename T>
class myLinearQueue {
   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:
   myLinearQueue() {
      rearIndex = frontIndex = numberOfElements = queueCapacity = 0;
      queueData = nullptr;
   }

   myLinearQueue(const MyQueue<T> &);                          // Copy constructor
   myLinearQueue<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 Circular Queue

A Circular Queue ADT is a data structure that works as a regular queue but with a circular wrap-around concept. It efficiently uses space by treating the queue as circular, meaning after the last element, it loops back to the front if space is available. You are supposed to add time and space complexity of each function in comments.
In this task, we will implement it using **templates** in C++ to allow flexibility with the data types.

```
template <class T>
class CircularQueue {
/***** define data members required to implement CircularQueue *****/
private:
   int front;          // Points to the front element of the queue
   int rear;           // Points to the last element in the queue
   int capacity;       // Maximum size of the queue
   int count;          // Current size of the queue
   T* queueArray;      // Array to hold queue elements

public:
   CircularQueue(int size);   // Constructor to initialize the queue with a given size
   ~CircularQueue();          // Destructor to free memory
   void enqueue(const T& data);  // Function to add an element to the queue
   T dequeue();               // Function to remove and return the front element
   bool isEmpty() const;      // Check if the queue is empty
   bool isFull() const;       // Check if the queue is full
   int size() const;          // Return the current size of the queue
   T peek() const;            // Peek at the front element without removing it
};
```

Explanation:

- The CircularQueue class uses C++ templates to handle elements of different data types.
- The constructor initializes a circular queue with a specified size.
- The destructor releases any dynamically allocated memory when the queue is destroyed.
- The enqueue function adds an element to the queue at the rear, and the position is updated circularly.
- The dequeue function removes and returns the front element, and the front pointer is updated circularly.
- The isEmpty function checks if the queue is empty.
- The isFull function checks if the queue is full.
- The size function returns the current number of elements in the queue.
- The peek function allows you to view the front element without dequeuing it.

Example Usage:

```
CircularQueue<int> intQueue(3);
intQueue.enqueue(10);
intQueue.enqueue(20);
intQueue.enqueue(30);

intQueue.dequeue();  // Removes 10
intQueue.enqueue(40);  // 40 gets added after 30 (circular)


CircularQueue<std::string> stringQueue(2);
stringQueue.enqueue("hello");
stringQueue.enqueue("world");
```

```
stringQueue.dequeue();  // Removes "hello"
stringQueue.enqueue("again");
```

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:** **[30 marks]**
  - Task 01: Implementation of Linear Queue [10 marks]
  - Task 01: Implementation of Circular Queue [20 marks]

References and Additional Material:

Queue Data Structure

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