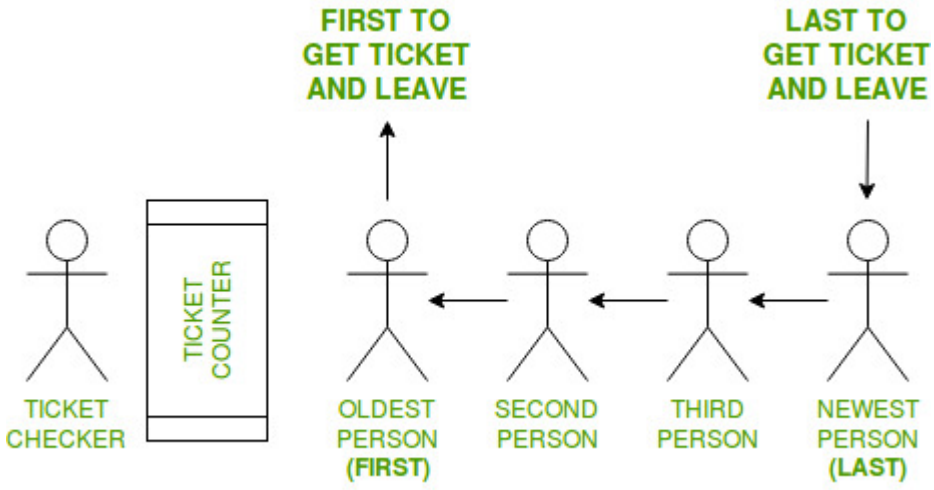
Queue Data Structure and C++ STL Queue Cheatsheet

1. Introduction to Queue Data Structure

A Queue is a linear data structure that follows a particular order in which operations are performed. The order is **First In, First Out (FIFO)**. This means the element that is inserted first will be the first one to be removed. It is analogous to a real-world queue, like people waiting in a line for a ticket counter, where the person who comes first gets the ticket first.

1.1 FIFO Principle

The FIFO (First In, First Out) principle is the core concept behind the Queue data structure. It dictates that the element that has been in the queue the longest is the next one to be removed. Conversely, the most recently added element is at the end of the queue and will be removed last. This behavior is in contrast to a Stack, which operates on a Last In, First Out (LIFO) principle.

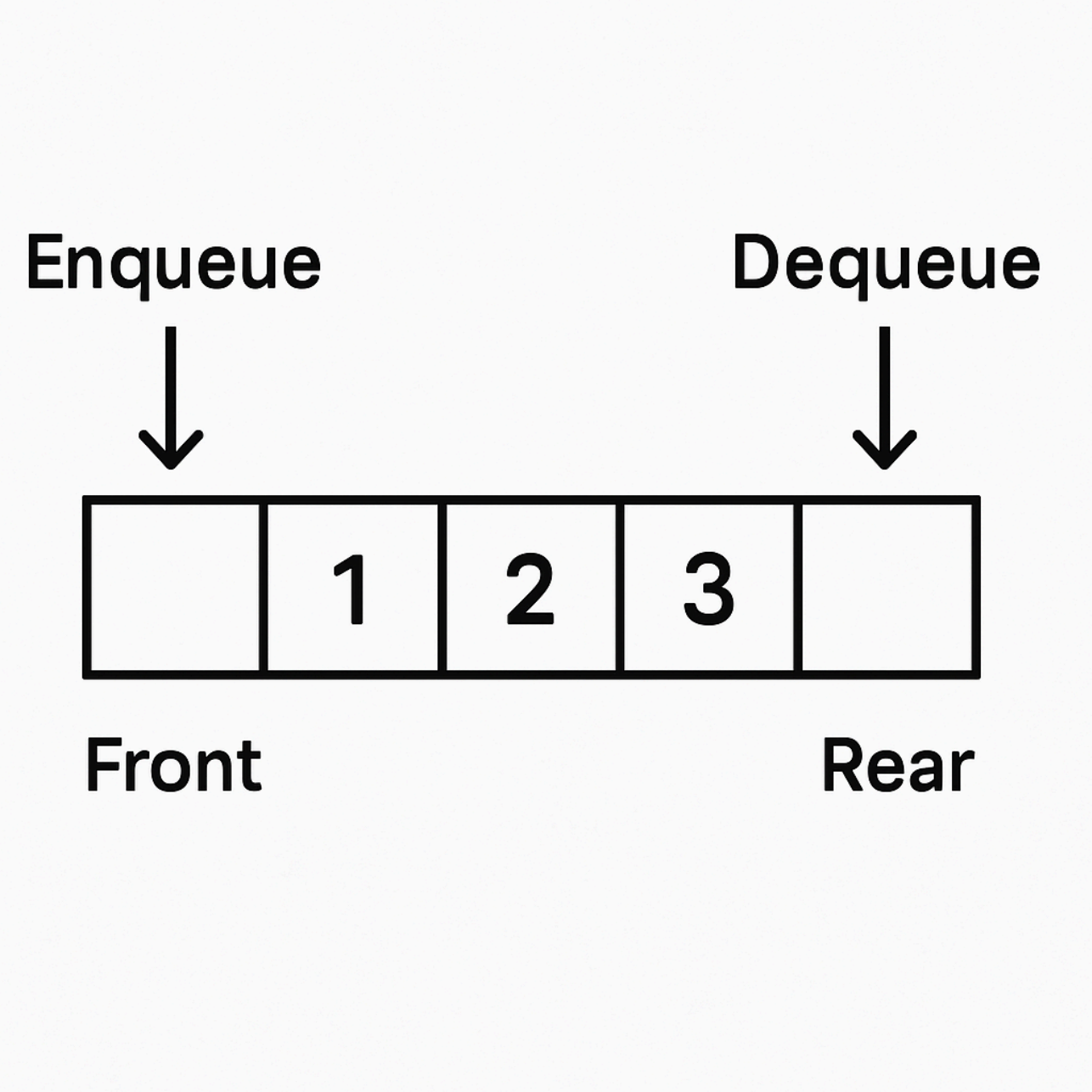


2. Basic Queue Operations

Queues support several fundamental operations:

- **Enqueue (or Push):** Adds an element to the rear (or back) of the queue.
- **Dequeue (or Pop):** Removes an element from the front of the queue.
- **Front (or Peek):** Returns the element at the front of the queue without removing it.
- **Rear (or Back):** Returns the element at the rear of the queue without removing it.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full (primarily for array-based implementations).

2.1 Enqueue and Dequeue Operations Diagram



2.2 Time Complexity of Basic Operations

The time complexity of queue operations depends on the underlying implementation. Here's a general overview:

Operation	Array Implementation (Circular Queue)	Linked List Implementation
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Front	O(1)	O(1)
Rear	O(1)	O(1)
isEmpty	O(1)	O(1)
isFull	O(1)	O(1)

In both array (circular) and linked list implementations, the basic operations typically achieve a constant time complexity (O(1)) because they involve a fixed number of steps regardless of the number of elements in the queue. For array-based queues, this assumes a circular buffer to avoid costly element shifts.

3. Queue Implementations

Queues can be implemented using various underlying data structures, most commonly arrays and linked lists.

3.1 Array Implementation

In an array-based implementation, a fixed-size array is used to store the queue elements. Two pointers, `front` and `rear`, are maintained to keep track of the front and rear of the queue, respectively. When an element is enqueued, the `rear` pointer is incremented. When an element is dequeued, the `front` pointer is incremented.

One common issue with a simple array implementation is that after several enqueue and dequeue operations, the `front` pointer might move far ahead, leaving empty space at the beginning of the array. This can lead to a situation where the queue is not actually full, but there's no space to enqueue new elements. This problem can be solved using a **circular array**.

Advantages: * Simple to implement. * Good cache performance due to contiguous memory allocation.

Disadvantages: * Fixed size: The size of the queue must be determined at compile time. * Inefficient space utilization in a linear array if not implemented as a circular queue.

C++ Code Example (Array Implementation - Circular Queue)

```cpp
#include <iostream>

class CircularQueue {
private:
    int* arr;
    int front;
    int rear;
    int capacity;
    int count;

public:
    CircularQueue(int size) {
        capacity = size;
        arr = new int[capacity];
        front = -1;
        rear = -1;
        count = 0;
    }

    ~CircularQueue() {
        delete[] arr;
    }

    void enqueue(int item) {
        if (isFull()) {
            std::cout << "Queue is full. Cannot enqueue " << item << std::endl;
            return;
        }
        if (isEmpty()) {
            front = 0;
        }
        rear = (rear + 1) % capacity;
        arr[rear] = item;
        count++;
        std::cout << "Enqueued: " << item << std::endl;
    }

    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty. Cannot dequeue." << std::endl;
            return;
        }
        std::cout << "Dequeued: " << arr[front] << std::endl;
        front = (front + 1) % capacity;
        count--;
        if (isEmpty()) {
            front = -1;
            rear = -1;
        }
    }

    int peekFront() {
        if (isEmpty()) {
            std::cout << "Queue is empty." << std::endl;
            return -1; // Or throw an exception
        }
        return arr[front];
    }
```

```
    int peekRear() {
        if (isEmpty()) {
            std::cout << "Queue is empty." << std::endl;
            return -1; // Or throw an exception
        }
        return arr[rear];
    }

    bool isEmpty() {
        return count == 0;
    }

    bool isFull() {
        return count == capacity;
    }

    int size() {
        return count;
    }
};

int main() {
    CircularQueue q(5);

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);

    q.dequeue();
    q.dequeue();

    q.enqueue(60);
    q.enqueue(70);

    std::cout << "Front element is: " << q.peekFront() << std::endl;
    std::cout << "Rear element is: " << q.peekRear() << std::endl;

    while (!q.isEmpty()) {
        q.dequeue();
    }

    q.dequeue(); // Trying to dequeue from an empty queue

    return 0;
}
```

3.2 Linked List Implementation

A linked list-based implementation provides a dynamic-size queue. Each element (node) in the queue contains the data and a pointer to the next element. Two pointers, `front` and `rear`, are maintained to point to the first and last nodes of the linked list, respectively.

Advantages: * Dynamic size: The queue can grow or shrink as needed, limited only by available memory. * Efficient insertion and deletion (O(1) time complexity).

Disadvantages: * Higher memory overhead due to pointers. * Slightly more complex to implement compared to a simple array.

C++ Code Example (Linked List Implementation)

```cpp
#include <iostream>

// Node structure for the linked list
struct Node {
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

class LinkedListQueue {
private:
    Node* front;
    Node* rear;
    int count;

public:
    LinkedListQueue() : front(nullptr), rear(nullptr), count(0) {}

    ~LinkedListQueue() {
        while (!isEmpty()) {
            dequeue();
        }
    }

    void enqueue(int item) {
        Node* newNode = new Node(item);
        if (isEmpty()) {
            front = newNode;
            rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        count++;
        std::cout << "Enqueued: " << item << std::endl;
    }

    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty. Cannot dequeue." << std::endl;
            return;
        }
        Node* temp = front;
        std::cout << "Dequeued: " << temp->data << std::endl;
        front = front->next;
        if (front == nullptr) { // If queue becomes empty after dequeue
            rear = nullptr;
        }
        delete temp;
        count--;
    }

    int peekFront() {
        if (isEmpty()) {
            std::cout << "Queue is empty." << std::endl;
            return -1; // Or throw an exception
        }
        return front->data;
```

```
    }

    int peekRear() {
        if (isEmpty()) {
            std::cout << "Queue is empty." << std::endl;
            return -1; // Or throw an exception
        }
        return rear->data;
    }

    bool isEmpty() {
        return front == nullptr;
    }

    int size() {
        return count;
    }
};

int main() {
    LinkedListQueue q;

    q.enqueue(100);
    q.enqueue(200);
    q.enqueue(300);

    std::cout << "Front element is: " << q.peekFront() << std::endl;
    std::cout << "Rear element is: " << q.peekRear() << std::endl;

    q.dequeue();
    q.dequeue();

    std::cout << "Front element is: " << q.peekFront() << std::endl;

    q.enqueue(400);

    while (!q.isEmpty()) {
        q.dequeue();
    }

    q.dequeue(); // Trying to dequeue from an empty queue

    return 0;
}
```

4. C++ STL `std::queue`

The C++ Standard Template Library (STL) provides a `queue` container adaptor that offers a convenient way to implement a queue data structure. It is not a container itself but an adaptor that provides a queue interface for other underlying container types, such as `std::deque` (default) or `std::list`.

4.1 Features of `std::queue`

- **FIFO Principle:** Strictly adheres to the First In, First Out principle.
- **Container Adaptor:** By default, it uses `std::deque` as its underlying container, but `std::list` can also be specified.
- **Restricted Access:** Elements can only be added to the back and removed from the front.

4.2 Member Functions

Here are the most commonly used member functions of `std::queue`:

Function	Description
`push(element)`	Adds `element` to the back of the queue.
`pop()`	Removes the element from the front of the queue.
`front()`	Returns a reference to the front element.
`back()`	Returns a reference to the back element.
`empty()`	Returns `true` if the queue is empty, `false` otherwise.
`size()`	Returns the number of elements in the queue.

4.3 C++ Code Example (`std::queue`)

```cpp
 #include <iostream>
#include <queue> // Required for std::queue
#include <string>

int main() {
    // Create a queue of integers
    std::queue<int> myQueue;

    // Enqueue elements
    myQueue.push(10);
    myQueue.push(20);
    myQueue.push(30);

    std::cout << "Queue size: " << myQueue.size() << std::endl;
    std::cout << "Front element: " << myQueue.front() << std::endl;
    std::cout << "Back element: " << myQueue.back() << std::endl;

    // Dequeue elements
    myQueue.pop();
    std::cout << "After one pop, front element: " << myQueue.front() <<
std::endl;

    myQueue.pop();
    std::cout << "After second pop, front element: " << myQueue.front() <<
std::endl;

    // Check if queue is empty
    if (myQueue.empty()) {
        std::cout << "Queue is empty." << std::endl;
    } else {
        std::cout << "Queue is not empty. Current front: " << myQueue.front()
<< std::endl;
    }

    myQueue.pop();

    if (myQueue.empty()) {
        std::cout << "Queue is now empty." << std::endl;
    }

    // Example with strings
    std::queue<std::string> stringQueue;
    stringQueue.push("Apple");
    stringQueue.push("Banana");
    stringQueue.push("Cherry");

    std::cout << "\nString Queue front: " << stringQueue.front() << std::endl;
    stringQueue.pop();
    std::cout << "String Queue front after pop: " << stringQueue.front() <<
std::endl;

    return 0;
}
```

5. Resources

- GeeksforGeeks: Queue Data Structure
- GeeksforGeeks: Queue in C++ STL
- cplusplus.com: std::queue
- Programiz: C++ Queue (With Examples)
- Wikipedia: Queue (abstract data type)

3.1.1 Complexity Analysis of Array Implementation (Circular Queue)

- **Time Complexity:**
  - **Enqueue:** O(1) - Adding an element involves updating the `rear` pointer and placing the element in the array, which takes constant time.
  - **Dequeue:** O(1) - Removing an element involves updating the `front` pointer, which also takes constant time.
  - **peekFront/peekRear:** O(1) - Accessing the front or rear element is a direct array lookup.
  - **isEmpty/isFull/size:** O(1) - These operations involve simple checks or returning a stored count.
- **Space Complexity:**
  - O(Capacity) - The space required is proportional to the maximum capacity of the queue, as a fixed-size array is allocated.

3.2.1 Complexity Analysis of Linked List Implementation

- **Time Complexity:**
  - **Enqueue:** O(1) - Creating a new node and updating pointers (`rear->next` and `rear`) takes constant time.
  - **Dequeue:** O(1) - Updating the `front` pointer and deallocating the old front node takes constant time.
  - **peekFront/peekRear:** O(1) - Accessing the front or rear element is a direct pointer dereference.

    - **isEmpty/size:** O(1) - These operations involve simple checks or returning a stored count.

- **Space Complexity:**
    - O(N) - The space required is proportional to the number of elements (N) in the queue, as each element requires a node with data and a pointer.

4.4 Time Complexity of `std::queue` Operations

The time complexity of `std::queue` operations depends on the underlying container. However, for the default `std::deque` and `std::list` containers, the operations are generally constant time.

Function	Time Complexity (using `std::deque` or `std::list`)
`push()`	O(1)
`pop()`	O(1)
`front()`	O(1)
`back()`	O(1)
`empty()`	O(1)
`size()`	O(1)

This consistent O(1) performance makes `std::queue` a highly efficient choice for implementing queue-like behavior in C++ applications.