QUEUE

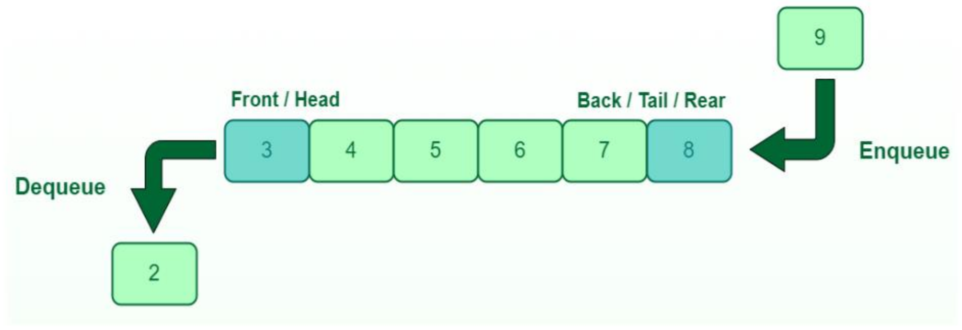
A Queue is a linear data structure that follows the First In, First Out (FIFO) principle. This means that the first element inserted into the queue will be the first one to be removed. Imagine a real-world queue (like a line of people): the person who gets in first is served first.

COMMON IMPLEMENTATIONS:

- Using Arrays
- Using Linked Lists
- Circular Queue
- Priority Queue (Special type of queue where elements are ordered by priority)
-

OPERATIONS ON A QUEUE:

- **Enqueue Operation:**
    - Add an element to the end (rear) of the queue.
    - Time Complexity: O(1)
- **Dequeue Operation:**
    - Remove the front element from the queue.
    - Time Complexity: O(1)
- **Peek (Front):**
    - Returns the element at the front of the queue.
    - Time Complexity: O(1)
- **Check Empty:**
    - Returns whether the queue is empty.
    - Time Complexity: O(1)

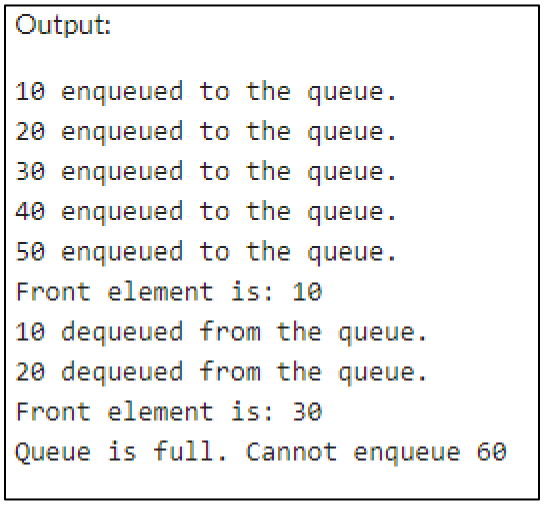


Queue Data Structure

QUEUE USING ARRAY

```
#include <iostream>
using namespace std;


class Queue {
private:
   int front, rear;
   int size;
   int* queue;


public:
   // Constructor to initialize the queue
   Queue(int s) {
      front = -1;
      rear = -1;
      size = s;
      queue = new int[size];
   }


   // Destructor to free up memory
   ~Queue() {
      delete[] queue;
   }


   // Enqueue operation to add an element at the rear
   void enqueue(int value) {
      if (rear == size - 1) { // Queue is full
```

```
        cout << "Queue is full. Cannot enqueue " << value << endl;
    } else {
        if (front == -1) {
            front = 0; // Initialize front on first enqueue
        }
        rear++;
        queue[rear] = value;
        cout << value << " enqueued to the queue." << endl;
    }
}

// Dequeue operation to remove an element from the front
void dequeue() {
    if (front == -1 || front > rear) { // Queue is empty
        cout << "Queue is empty. Cannot dequeue." << endl;
    } else {
        cout << queue[front] << " dequeued from the queue." << endl;
        front++;
    }
}

// Function to get the front element
int peekFront() {
    if (front == -1 || front > rear) {
        cout << "Queue is empty." << endl;
        return -1;
    } else {
          return queue[front];
       }
    }

    // Function to check if the queue is empty
    bool isEmpty() {
       return (front == -1 || front > rear);
    }
};

int main() {
    Queue q(5); // Create a queue with capacity 5

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

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

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

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

    q.enqueue(60); // Try adding after dequeuing
    return 0;

}
```



```
Output:

10 enqueued to the queue.
20 enqueued to the queue.
30 enqueued to the queue.
40 enqueued to the queue.
50 enqueued to the queue.
Front element is: 10
10 dequeued from the queue.
20 dequeued from the queue.
Front element is: 30
Queue is full. Cannot enqueue 60
```

KEY POINTS:

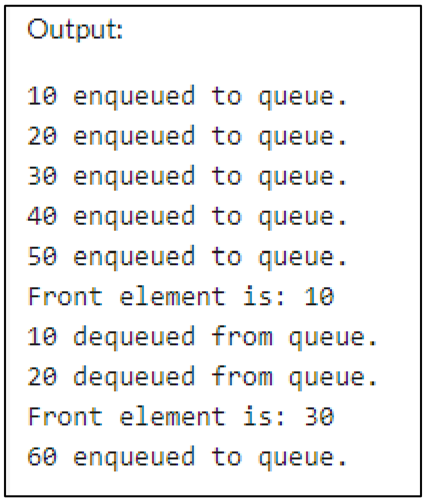
- The front starts at -1 and is initialized to 0 when the first element is enqueued.
- The rear is incremented each time an element is enqueued.
- If the queue is empty (front > rear), operations like dequeue or peek will not work.

QUEUE USING SLL

```
#include <iostream>

using namespace std;



// Node structure for a singly linked list

struct Node {

    int data;

    Node* next;



    // Constructor to initialize a node

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

// Queue class using a singly linked list
class Queue {
private:
    Node* front;
    Node* rear;

public:
    // Constructor to initialize the queue
    Queue() {
        front = rear = nullptr;
    }

    // Enqueue operation to add an element to the rear of the queue
    void enqueue(int value) {
        Node* newNode = new Node(value);

        // If queue is empty, the new node is both the front and rear
        if (rear == nullptr) {
            front = rear = newNode;
            cout << value << " enqueued to queue." << endl;
            return;
        }
        // Add the new node at the end of the queue and update the rear pointer
        rear->next = newNode;
        rear = newNode;
        cout << value << " enqueued to queue." << endl;
    }

    // Dequeue operation to remove an element from the front of the queue
    void dequeue() {
        if (front == nullptr) { // Queue is empty
            cout << "Queue is empty. Cannot dequeue." << endl;
            return;
        }

        // Move the front pointer to the next node
        Node* temp = front;
        front = front->next;

        // If the queue becomes empty, update the rear pointer
        if (front == nullptr) {
            rear = nullptr;
        }

        cout << temp->data << " dequeued from queue." << endl;
        delete temp; // Free the memory of the dequeued node
    }
```

```
    // Function to get the front element
    int peekFront() {
        if (front == nullptr) {
            cout << "Queue is empty." << endl;
            return -1;
        }
        return front->data;
    }

    // Function to check if the queue is empty
    bool isEmpty() {
        return front == nullptr;
    }

    // Destructor to clean up all nodes
    ~Queue() {
        while (front != nullptr) {
            Node* temp = front;
            front = front->next;
            delete temp;
        }
    }
};

int main() {
    Queue q; // Create a queue
```

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

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

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

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

  q.enqueue(60); // Try adding after dequeuing

  return 0;
}
```



Output:

```
10 enqueued to queue.
20 enqueued to queue.
30 enqueued to queue.
40 enqueued to queue.
50 enqueued to queue.
Front element is: 10
10 dequeued from queue.
20 dequeued from queue.
Front element is: 30
60 enqueued to queue.
```

KEY POINTS:

- The queue uses a Singly Linked List, so it can dynamically grow and shrink as needed without the fixed size constraint of an array.
- Enqueue and Dequeue operations work in constant time O(1)O(1)O(1), as they only involve pointer adjustments.
- When a node is dequeued, memory is freed to avoid memory leaks.