Queue Implementation Using Linked List: Detailed Breakdown

Introduction

A queue is a **FIFO (First In, First Out)** data structure. This means that the element added first is removed first. The queue implementation in this code uses a **linked list**, making it dynamically resizable without any fixed size limitation.

In this implementation, two pointers are used:

1. **f (front)** - Points to the first element of the queue.
2. **b (back)** - Points to the last element of the queue.

Additionally, a **size variable** keeps track of the number of elements in the queue.

**Node Structure (class node)

Each node represents an element of the queue and consists of:

- An integer value (val) to store data.
- A pointer (next) that points to the next node in the queue.
- A constructor that initializes these attributes.

This allows us to dynamically create new nodes as needed.

**Queue Structure (class queue)

The queue maintains:

- f (front) and b (back) pointers.
- size, initialized to zero.

The queue supports the following operations:

1. Push Operation (push(int val))

This function inserts an element at the end of the queue.

- If the queue is empty (size == 0), the new node is both the front and the back.
- Otherwise, the new node is linked to the previous back, and b is updated.
- The size of the queue increases by 1.

Time Complexity: O(1) (constant time insertion)

2. Pop Operation (pop())

This function removes the front element of the queue.

- If the queue is empty, it prints an error message.
- Otherwise:

    - The front node is temporarily stored.
    - f is moved to the next node.
    - The old front node is deleted to free memory.
    - The size of the queue decreases by 1.

Edge Case: If the last element is removed, b should be set to NULL.

Time Complexity: O(1) (constant time deletion)

3. Front Operation (front())

Displays the value of the front node.

- If the queue is empty, an appropriate message is displayed.
- Otherwise, the front value is printed.

Time Complexity: O(1)

4. Back Operation (back())

Displays the value of the back node.

- If the queue is empty, an appropriate message is displayed.
- Otherwise, the back value is printed.

Time Complexity: O(1)

5. Display Operation (display())

Prints all elements of the queue from front to back.

- A temporary pointer (temp) starts at f and iterates through the queue, printing each value.
- The loop stops when temp == NULL.

Time Complexity: O(n) (traverses the entire queue)

Working of the Queue

Example Execution:

1. push(1) → Queue: [1]
2. push(2) → Queue: [1 → 2]
3. push(3) → Queue: [1 → 2 → 3]
4. display() → Output: 1 2 3
5. pop() → Queue: [2 → 3]
6. display() → Output: 2 3

Key Takeaways

- The queue follows the **FIFO principle**.
- A **linked list is used** for dynamic memory allocation.
- The push() function inserts elements at the back in O(1) time.
- The pop() function removes elements from the front in O(1) time.
- front(), back(), and display() functions help access queue elements efficiently.

Improvements and Edge Cases

1. **Handling Empty Queue for front() and back()**
   - Before accessing f->val or b->val, check if size == 0.
2. **Ensuring Proper Memory Deallocation**
   - The pop() function should use delete to prevent memory leaks.
3. **Handling the Last Element Removal**
   - If the last element is deleted, set b = NULL to prevent dangling pointers.

Conclusion

This queue implementation using a linked list is an efficient and flexible approach for dynamic data storage. Since it avoids the limitations of a static array-based queue, it is well-suited for applications where the queue size is unknown beforehand. The operations maintain optimal time complexity, ensuring fast insertions and deletions.