Queue Implementation Using Doubly Linked List

Introduction

A queue is a linear data structure that follows the **FIFO (First In, First Out)** principle. It allows insertion at the back and deletion from the front. However, in this implementation, we use a **doubly linked list** to create a queue that supports additional functionalities like insertion and deletion from both ends, making it a **deque (double-ended queue).**

Class Design

1. Node Class (node)

Each node in the doubly linked list contains:

- An integer value val.
- A pointer next pointing to the next node.
- A pointer prev pointing to the previous node.

Code Implementation:

```
class node{
 public:
 int val;
 node* next;
 node* prev;
 node(int val){
  this->val = val;
  next = NULL;
  prev = NULL;
 }
};
```

2. Queue Class (queue)

This class maintains:

- f: Pointer to the front node.
- b: Pointer to the back node.
- size: Keeps track of the number of elements.

Code Implementation:

```
class queue{
```

```
 public:
 node* f;
 node* b;
 int size;

 queue(){
  f = NULL;
  b = NULL;
  size = 0;
 }
```

Queue Operations

1. push_back(int val) – Insert at the Back

This function inserts a new element at the back of the queue:

- If the queue is empty, both f and b point to the new node.
- Otherwise, the new node is added at the back, and pointers are updated.

```
void push_back(int val){
 if(size == 0){
  node* temp = new node(val);
  f = temp;
  b = temp;
 }
 else{
  node* temp = new node(val);
  b->next = temp;
  temp->prev = b;
  b = temp;
 }
 size++;
}
```

2. push_front(int val) – Insert at the Front

Similar to push_back, but inserts an element at the front.

```
void push_front(int val){
 if(size == 0){
  node* temp = new node(val);
  f = temp;
  b = temp;
 }
 else{
  node* temp = new node(val);
  f->prev = temp;
  temp->next = f;
  f = temp;
 }
 size++;
}
```

3. pop_back() – Remove from the Back

- If the queue is empty, prints an error message.
- Otherwise, removes the last element and updates b.

```
void pop_back(){
 if(size == 0){
  cout << "QUEUE IS ALREADY EMPTY!";
 }
 else{
  node* temp = b;
  b = b->prev;
  delete temp;
  if (b) b->next = NULL;
  size--;
 }
}
```

4. pop_front() – Remove from the Front

- If the queue is empty, prints an error message.
- Otherwise, removes the front element and updates f.

```
void pop_front(){
 if(size == 0){
  cout << "QUEUE IS ALREADY EMPTY!";
 }
 else{
  node* temp = f;
  f = f->next;
  delete temp;
  if (f) f->prev = NULL;
  size--;
 }
}
```

5. front() – Get the Front Element

Prints the front element.

```
void front(){
 cout << f->val << endl;
}
```

6. back() – Get the Last Element

Prints the last element.

```
void back(){
 cout << b->val << endl;
}
```

7. display() – Print the Queue Elements

Prints all the elements in the queue.

```
void display(){
 node* temp = f;
 while(temp){
  cout << temp->val << " ";
  temp = temp->next;
```

```
  }
  cout << endl;
}
```

8. length() – Get the Queue Size

Prints the current size of the queue.

```
void length(){
  cout << size << endl;
}
```

Main Function (main())

The main() function demonstrates the working of the queue.

```
int main(){
  queue q;
  q.push_back(1);
  q.push_back(2);
  q.push_back(3);
  q.display();  // Output: 1 2 3

  q.pop_back();
  q.display();  // Output: 1 2

  q.push_front(0);
  q.push_front(-1);
  q.push_front(-2);
  q.display();  // Output: -2 -1 0 1 2

  q.pop_front();
  q.display();  // Output: -1 0 1 2

  q.front();    // Output: -1
  q.back();     // Output: 2
```

```
  q.length();   // Output: 4

  return 0;
}
```

Key Takeaways

1. **Doubly Linked List is used**: Each node has prev and next pointers.
2. **Efficient Insertions and Deletions**: Operations at both ends take **O(1) time**.
3. **Functionality beyond Standard Queue**: This implementation allows both front and back operations (like a deque).
4. **Memory Management**: Dynamic memory is allocated using new and properly freed using delete.

Conclusion

This queue implementation using a doubly linked list provides flexibility with efficient insertion and deletion from both ends. It serves as the foundation for **deque** (double-ended queue), which has multiple applications in scheduling, buffering, and data handling scenarios.