Implement Double-Ended Queue Using Linked List

This program implements a Double-Ended Queue (Deque) data structure using a doubly linked list. It defines a **Deque** class with methods for pushing elements to the front and rear of the deque, popping elements from the front and rear, checking if the deque is empty, getting the front and rear elements, getting the size of the deque, and displaying its elements. The program demonstrates the usage of the **Deque** class by performing various deque operations.

1. **Deque()** - Constructor to initialize an empty deque.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This constructor initializes both **front** and **rear** as **nullptr**, indicating an empty deque. It has constant time and space complexity.
2. **void pushFront(int value)** - Pushes an element to the front of the deque.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This method creates a new node with the given value, updates the links to insert it at the front, and moves the **front** pointer. It has constant time and space complexity.
3. **void pushRear(int value)** - Pushes an element to the rear of the deque.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This method creates a new node with the given value, updates the links to insert it at the rear, and moves the **rear** pointer. It has constant time and space complexity.
4. **int popFront()** - Pops an element from the front of the deque. Returns -1 if the deque is empty.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This method removes the front node, updates the **front** pointer, and returns the deleted element's value. It has constant time and space complexity.

5. **int popRear()** - Pops an element from the rear of the deque. Returns -1 if the deque is empty.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This method removes the rear node, updates the **rear** pointer, and returns the deleted element's value. It has constant time and space complexity.
6. **int getFront()** - Returns the first element of the deque. If the deque is empty, it returns -1.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This method returns the value of the **front** node, which represents the front element of the deque. It has constant time and space complexity.
7. **int getRear()** - Returns the last element of the deque. If the deque is empty, it returns -1.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This method returns the value of the **rear** node, which represents the rear element of the deque. It has constant time and space complexity.
8. **bool isEmpty()** - Checks if the deque is empty.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - **Explanation:** This method checks if **rear** is **nullptr**, indicating an empty deque. It has constant time and space complexity.
9. **int getSize()** - Returns the current number of elements in the deque.
    - **Time Complexity: O(n), where n is the number of elements in the deque.**
    - **Space Complexity: O(1)**
    - **Explanation:** This method iterates through the deque to count the number of nodes and returns the count. It has a time complexity proportional to the number of elements and constant space complexity.
10. **void display()** - Displays the elements in the deque.

- **Time Complexity: O(n), where n is the number of elements in the deque.**
- **Space Complexity: O(1)**
- **Explanation:** This method iterates through the deque to print its elements. It has a time complexity proportional to the number of elements and constant space complexity.

11. **~Deque()** - Destructor to release memory allocated for the deque.
    - **Time Complexity: O(n), where n is the number of elements in the deque.**
    - **Space Complexity: O(1)**
    - **Explanation:** This destructor iterates through the deque and deletes each node to release memory. It has a time complexity proportional to the number of elements and constant space complexity.