DeQue

A deque, short for "double-ended queue," is an abstract data type that represents a collection of elements with two ends: a front and a rear. It allows efficient insertion and deletion of elements from both ends, making it a versatile data structure. The deque can be visualized as a linear structure where elements can be added or removed from either end.

Key features of a deque:

1. Double-Ended: Unlike other linear data structures like stacks and queues, a deque allows insertion and deletion of elements from both ends. This flexibility enables efficient implementations of various algorithms and data manipulation operations.
2. Dynamic Size: Deques can grow or shrink dynamically as elements are added or removed. This means you don't need to specify the maximum number of elements in advance, making them suitable for scenarios where the size is unpredictable or needs to be adjusted dynamically.
3. Random Access: Deques provide random access to elements, allowing you to access or modify any element at a specific position directly. This is different from traditional queues, where access is limited to the front or rear.

Common operations supported by deques:

1. Insertion: Elements can be added at the front or the rear of the deque. This operation is known as "push" or "enqueue." When adding elements, the deque adjusts its size accordingly.
2. Deletion: Elements can be removed from the front or the rear of the deque. This operation is known as "pop" or "dequeue." Similar to insertion, the deque adjusts its size when elements are removed.
3. Access: Elements within the deque can be accessed individually by their position (index). Random access allows you to retrieve or modify any element at a given index in constant time.
4. Size: You can determine the number of elements present in the deque using the "size" operation.
5. Empty Check: You can check whether the deque is empty or not using the "isEmpty" operation.

Deques can be implemented using various data structures such as arrays, linked lists, or dynamic arrays.

The Time and Space Complexity of the functions used in DeQue

1. **printDeQue()**: Prints the elements of the deque.
    - Time Complexity: O(n), where n is the number of elements in the deque.

- Space Complexity: O(1)

2. **printDeQueReverse()**: Prints the elements of the deque in reverse order.
    - Time Complexity: O(n), where n is the number of elements in the deque.
    - Space Complexity: O(1)
3. **main()**: The main function that demonstrates the usage of various deque functions.
    - Time Complexity: Depends on the operations performed.
    - Space Complexity: Depends on the operations performed.
4. **myDeque.empty()**: Checks if the deque is empty.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
5. **myDeque.push_front()**: Inserts an element at the front of the deque.
    - Time Complexity: O(1) amortized.
    - Space Complexity: O(1)
6. **myDeque.push_back()**: Inserts an element at the back of the deque.
    - Time Complexity: O(1) amortized.
    - Space Complexity: O(1)
7. **myDeque.insert()**: Inserts elements at a specific position in the deque.
    - Time Complexity: O(n + k), where n is the number of elements inserted and k is the number of elements shifted.
    - Space Complexity: O(n), where n is the number of elements inserted.
8. **myDeque.begin()**: Returns an iterator pointing to the first element of the deque.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
9. **myDeque.end()**: Returns an iterator pointing one past the last element of the deque.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
10. **myDeque.rbegin()**: Returns a reverse iterator pointing to the last element of the deque (reverse beginning).

- Time Complexity: O(1)
- Space Complexity: O(1)

11. **myDeque.rend()**: Returns a reverse iterator pointing one position before the first element of the deque (reverse end).

- Time Complexity: O(1)
- Space Complexity: O(1)

12. **myDeque.at()**: Accesses an element at a specific position in the deque.

- Time Complexity: O(1)
- Space Complexity: O(1)

13. **myDeque.front()**: Accesses the first element of the deque.

- Time Complexity: O(1)
- Space Complexity: O(1)

14. **myDeque.back()**: Accesses the last element of the deque.

- Time Complexity: O(1)
- Space Complexity: O(1)

15. **std::sort()**: Sorts the elements of the deque.

- Time Complexity: O(n log n), where n is the number of elements in the deque.
- Space Complexity: O(1)

16. **myDeque.erase()**: Erases an element at a specific position in the deque.

- Time Complexity: O(n + k), where n is the number of elements erased and k is the number of elements shifted.
- Space Complexity: O(1)

17. **myDeque.pop_front()**: Removes the first element of the deque.

- Time Complexity: O(1)
- Space Complexity: O(1)

18. **myDeque.pop_back()**: Removes the last element of the deque.

- Time Complexity: O(1)
- Space Complexity: O(1)

19. **myDeque.resize()**: Resizes the deque to a specified size.

- Time Complexity: O(n), where n is the new size.
- Space Complexity: O(n), where n is the new size.

20. **myDeque.size()**: Returns the number of elements in the deque.

- Time Complexity: O(1)
- Space Complexity: O(1)

21. **myDeque.clear()**: Removes all elements from the deque.

- Time Complexity: O(n), where n is the number of elements in the deque.
- Space Complexity: O(1)