Project Report:

Critical Thinking 6 – Custom Deque ADT

Alejandro Ricciardi

Colorado State University Global

CSC400: Data Structures and Algorithms

Professor: Hubert Pensado

September 22, 2024

Project Report:

Critical Thinking 6 – Custom Deque ADT

This documentation is part of the Critical Thinking 6 Assignment from CSC400: Data Structures and Algorithms at Colorado State University Global. This Project Report is an overview of the program's functionality and testing scenarios including console output screenshots. The program is coded in Java JDK-22 and named Critical Thinking 6 (Custom Deque).

The Assignment Direction:

Basic implementation of a custom Deque ADT
Implement a custom Deque ADT with an iterator in Java. The Deque should support the basic operations:

insertion
deletion
traversal using an iterator
Requirements
Implement a class named CustomDeque with the following methods:

enqueueFront(int data): Inserts a new element at the front of the deque.
enqueueRear(int data): Inserts a new element at the rear of the deque.
dequeueFront(): Removes and returns the element from the front of the deque.
dequeueRear(): Removes and returns the element from the rear of the deque.
iterator(): Returns an iterator for traversing the deque.
Implement an inner class named DequeIterator within CustomDeque to serve as the iterator. The iterator should have the following methods:

hasNext(): Returns true if there is a next element, false otherwise.
next(): Returns the next element and moves the iterator to the next position.
Demonstrate the functionality of the CustomDeque by iterating through its elements using the custom iterator. To get things started, used the following starter code:

```
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.NoSuchElementException;

public class CustomDeque {
    private Deque<Integer> deque;

    public CustomDeque() {
        this.deque = new LinkedList<>();
    }

    public void enqueueFront(int data) {
        deque.addFirst(data);
```

```
    }

    public void enqueueRear(int data) {
        deque.addLast(data);
    }

    public int dequeueFront() {
        if (isEmpty()) {
            throw new NoSuchElementException("Deque is empty");
        }
        return deque.removeFirst();
    }

    public int dequeueRear() {
        if (isEmpty()) {
            throw new NoSuchElementException("Deque is empty");
        }
        return deque.removeLast();
    }

    public Iterator<Integer> iterator() {
        return new DequeIterator();
    }

    public boolean isEmpty() {
        return deque.isEmpty();
    }

    private class DequeIterator implements Iterator<Integer> {
        private Iterator<Integer> iterator = deque.iterator();

        @Override
        public boolean hasNext() {
            return iterator.hasNext();
        }

        @Override
        public Integer next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return iterator.next();
        }
    }

    public static void main(String[] args) {
        CustomDeque customDeque = new CustomDeque();
```

```
        // Enqueue elements
        customDeque.enqueueFront(1);
        customDeque.enqueueRear(2);
        customDeque.enqueueFront(3);

        // Iterate and display elements
        Iterator<Integer> iterator = customDeque.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

Testing:
There is no need for file I/O. Test your program using an array of ten random integers.
Submit your completed assignment as a .java source code file.

⚠️ **My notes:**

- I changed the Deque data type parameter from the Integer data type to the generic data type; public class CustomDeque<T>.
- In the TestCustomDeque class the integers are directly enqueued into the instantiated Custom Deque object instead of getting them from an array and then enqueuing them.

My Program Description:

The program is a custom double-ended queue (deque) implementation in Java using Java's LinkedList.
It tests the deque insertion and iteration functionalities by enqueuing random integers from both the front and rear; and by iterating starting from the front and then starting from the rear.

Git Repository

I use GitHub as my Distributed Version Control System (DVCS), the following is a link to my GitHub, Omegapy.
My GitHub repository that is used to store this assignment is named My-Academics-Portfolio and the link to this specific assignment is: https://github.com/Omegapy/My-Academics-Portfolio/tree/main/Data-Structures-and-Algorithms-CSC400/Critical-Thinking-6

Classes Description:

- **CustomDeque<T> Class**
  It is A custom implementation of a double-ended queue, a deque
  A deque that allows the insertion and removal of elements from both the front and rear ends
  This class uses Java's LinkedList to store elements

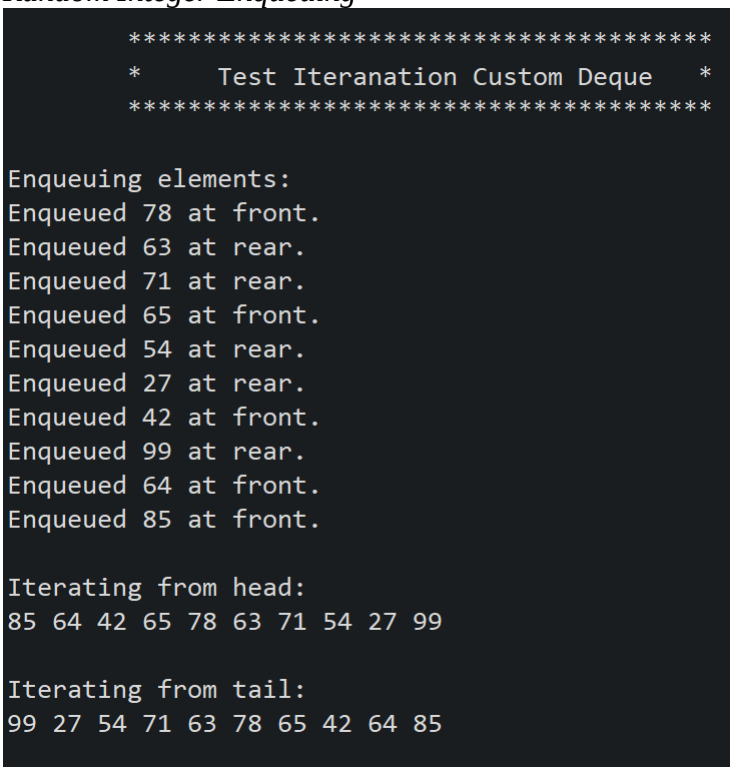
- **TestCustomDeque Class**
  Tests the iterating functionality of the CostumeDeque class. It adds random integers to the deque and then iterates through them from both the front (head) and the rear (tail).

Deque

In Java, the Deque or “double-ended queue” in Java is a data structure in which elements can be inserted or deleted elements from both ends. The deque is an interface in Java belonging to java.util package and it implements java.queue interface. (Sruthy, 2024)
The deque as a stack (Last In, First Out) structure or as a queue (first-in-first-out).

Screenshot

Figure 1
Random Integer Enqueuing



Continue next page

Figure 2
Deleting Elements

```
------------------------------------------------------------
Test 1: Deleting elements from front and rear

Initial elements (iterating from head) before deleting:
69 43 4 66 29 29 47 75 82 12
Elements after deleting from front (from head):
43 4 66 29 29 47 75 82 12

Initial elements (iterating from tail) before deleting:
12 82 75 47 29 29 66 4 43
Elements after deleting from rear (from tail):
82 75 47 29 29 66 4 43
```

Figure 3
Adding Elements

```
------------------------------------------------------------
Test 2: Adding elements to front and rear

Initial elements (iterating from head) before adding 50 to head:
43 4 66 29 29 47 75 82
Elements after adding 50 to front (from head):
50 43 4 66 29 29 47 75 82

Initial elements (iterating from tail) before adding 99 to tail:
82 75 47 29 29 66 4 43 50
Elements after adding 99 to rear (from tail):
99 82 75 47 29 29 66 4 43 50
```

Figure 4
Testing hasNext() and next() From Front

```
------------------------------------------------------------
Test 3: Testing hasNext() and next() functionality (from head)

Iterating and displaying using next():
50 43 4 66 29 29 47 75 82 99
Testing hasNext() trying after iterating from head: false
```

Figure 4
Testing hasNext() and next() From Rear

```
------------------------------------------------------------
Test 4: Testing hasNext() and next() functionality (from tail)

Iterating and displaying using next():
99 82 75 47 29 29 66 4 43 50
Testing hasNext() trying after iterating from tail: false
```

As shown in Figures 1 to 5 the program runs without any issues displaying the correct outputs as expected.