Hogeschool van Amsterdam

Datastructures

Assignment 1C Deque

Tijmen Stor, 500752989, IS205

Index

Introduction 2

Assignment 1C 3

1.1 Implementing Deque 3

1.2 Deque Change Methods 6

1.3 Implementing Iterable<Item> 7

1.4 Testing the Deque 7

Introduction

In this assignment, I will be answering the questions that are part of assignment 1C. We are going to create a Double-Ended-Queue, which is a LinkedList that works from the left and the right side. We will be implementing the methods which are given to us from the start, such as inserting, deleting and changing elements in the deque. At the end of the assignment, the deque will be tested by running two separate testcases on it.

Assignment 1C

1.1 Implementing Deque

The first assignment was about implementing the necessary methods, such as isEmpty, pushing (inserting) elements on both sides and popping (deleting) elements on both sides. These are the methods that had to be implemented and written for this assignment:

- isEmpty()
- size()
- pushLeft(Item item)
- pushRight(Item item)
- popLeft()
- popRight()
- changeLeft()
- changeRight()

changeLeft and changeRight are also in this row of methods, but this will be created further into the assignment. In this part, we will be focusing us on creating the other methods. Down here is the code I have written for each method.

```
Node<Item> left, right;
int length = 0;

@Override
public boolean isEmpty() {
    return length == 0;
}

@Override
public int size() {
    return length;
}

@Override
public void pushLeft(Item item) {
    Node<Item> node = new Node<>(null, null, item);
    if (isEmpty()) {
        left = right = node;
    } else {
        node.setNext(left);
        left.setPrevious(node);
        left = node;
    }
    length++;
}

@Override
public void pushRight(Item item) {
    Node<Item> node = new Node<>(null, null, item);
    if (isEmpty()) {
        right = left = node;
    } else {
        node.setPrevious(right);
        right.setNext(node);
        right = node;
    }
```

```
        length++;
    }

    @Override
    public Item popLeft() {
        if (isEmpty()) {
            throw new IndexOutOfBoundsException("Deque is empty");
        }
        Item deleteLeft = left.getItem();
        if (left.getNext() != null) {
            left = left.getNext();
            left.setPrevious(null);
        } else {
            left = null;
        }
        length--;

        return deleteLeft;
    }

    @Override
    public Item popRight() {
        if (isEmpty()) {
            throw new IndexOutOfBoundsException("Deque is empty");
        }
        Item deleteRight = right.getItem();
        if (right.getPrevious() != null) {
            right = right.getPrevious();
            right.setNext(null);
        } else {
            right = null;
        }
        length--;
        return deleteRight;
    }
```

The isEmpty() method returns a boolean, which is based on the size of the deque. If it's 0, it will return true. Any other size will make it return false.

The size() method returns the size of the deque.

The pushLeft() will receive an item and pushes that item to the left-side of the deque. This is done, by first creating a new node containing that item. If the deque was already empty, the node becomes the middle of the deque, thus becoming left and right. If this is not the case, the most left node will be pushed to the right side of the new node. The new node will be pushed to the left side of the old left node. To finish it off, the new node is made to be the new left node in the class. To finish the method completely, the length of the deque is increased.

The above explanation also applies to the pushRight(), but instead the most right node is moved to the left side of the new node and vice versa, after which the new right node is made the right node in the class.

The popLeft() method is in charge of deleting the most left item in the deque. This is done by first checking if the deque is not already empty. After that, a new item is created to store the item from the node that is getting deleted (because this needs to be returned later on). Then the node next to the left node is being checked. If there is a node next to the left, this node becomes the new left nodes. The left node is then being changed to have no nodes in front of him, because he is the most left node. If however, there is no node next to the left one at the start, the deque becomes empty after removing the left one. Therefore, the left node becomes null. After that, the length of the deque is diminished by 1 and the method returns the deleted item.

For popRight() apply the same rules again, with the changes to checking the previous number to the most right node and changing the previous node to the right node.

1.2 Deque Change Methods

For this part of the assignment, we had to implement the changeLeft() and the changeRight() methods, which would be used to change an element within the deque based on the input that is given to the method. Down below is the code that I have written to make the methods work.

```
@Override
public Item changeLeft(int n, Item newItem) {
    if (n >= length || n < 0) {
        throw new IndexOutOfBoundsException(n + "is out of bounds");
    }
    Node<Item> node = left;
    for (int i = 0; i < n; i++) {
        node = node.getNext();
    }
    Item old = node.getItem();
    node.setItem(newItem);
    return old;
}

@Override
public Item changeRight(int n, Item newItem) {
    if (n >= length || n < 0) {
        throw new IndexOutOfBoundsException(n + "is out of bounds");
    }
    Node<Item> node = right;
    for (int i = 0; i < n; i++) {
        node = node.getPrevious();
    }
    Item old = node.getItem();
    node.setItem(newItem);
    return old;
}
```

The changeLeft() receives two parameters, an int n and an Item newItem. The int n is used to iterate to the nth element in the deque. The nth element in the deque is then replaced by the newItem parameter. At the start of the method, the int n is checked whether it is not a number under 0 and if it is not bigger than the size of the deque itself. Then a new node is created to store the left node in, which will soon be the node of the nth element. We then loop through the deque until we arrive at the nth element. For changeLeft() we move to the right (getNext()) and for changeRight() we move to the left (getPrevious()). The item is then gotten from the node and stored into a new Item variable called old. The old item is then replaced by the new item newItem and the method then returns the old item, which was stored in the item old.

1.3 Implementing Iterable<>

To actually be able to iterate through the deque, the Iterable interface had to be implemented. With this, the method iterator() had to be implemented into the code and be written. The code for this is down below here.

```
@Override
public Iterator<Item> iterator() {
    return new Iterator<Item>() {

        private Node<Item> node = left;

        @Override
        public boolean hasNext() {
            return node != null;
        }

        @Override
        public Item next() {
            Item item = node.getItem();
            node = node.getNext();
            return item;
        }
    };
}
```

We start it off by making it return a new Iterator<Item>. After that a new node is created with left, because we want to iterate from left to right through the deque. We then check if there is a node after the current node. If there is no node after the current one, the iterator stops iterating after the current node. The next method will get the item from the current node and then change the current node to the next node. After that, the old item is returned and the iterator starts over again with the new current node.

1.4 Testing the Deque

The final assignment is about testing the output of the deque. This small block of code had to be implemented and a small enhanced for-loop had to be created. The code looks like this:

```
public static void main(String[] args) {
    Deque<String> words = new Deque<>();
    words.pushLeft("Datastructuren");
    words.pushLeft("is");
    words.pushRight("heel");
    words.pushRight("leuk");
    words.pushLeft("of");
    words.pushRight("niet?");
    words.changeLeft(3, "test");
    words.changeRight(3, "een");
    words.changeLeft(5, "Dit");
    words.changeLeft(0, words.popRight());
    words.popRight();

    for(String word : words){
        System.out.println(word);
    }
```

```
        Deque<Integer> numbers = new Deque<>();
        numbers.pushLeft(1);
        numbers.pushRight(2);
        numbers.pushLeft(3);
        numbers.pushLeft(numbers.popLeft() - numbers.popRight());
        numbers.pushRight(2);
        numbers.pushLeft(5);
        numbers.pushLeft(5);
        numbers.pushRight(8);
        numbers.pushRight(numbers.popRight() - numbers.popLeft());
        numbers.pushRight(3);
        numbers.pushRight(9);
        numbers.pushRight(13);
        numbers.pushRight(7 * numbers.changeLeft(4, numbers.popLeft()));
        numbers.changeRight(2, 8);
        numbers.changeLeft(6, numbers.changeRight(1, 0));
        for (Integer number : numbers) {
            System.out.print(number);
        }
    }
```

The following output is produced after executing.

```
------------------------------------------------------------------------
Building dequeue 0.0.1-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ dequeue ---
Dit
is
een
test
1123581321
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.038s
Finished at: Mon Nov 27 18:27:45 CET 2017
Final Memory: 5M/155M
------------------------------------------------------------------------
```