Workshop Custom Linked List

Overview

In this workshop, we are going to create another custom data structure, which has similar functionalities as the **Java `LinkedList`**. Just like the structures from the previous workshop, our custom **`LinkedList`** will work only with integers. It will have the following functionalities:

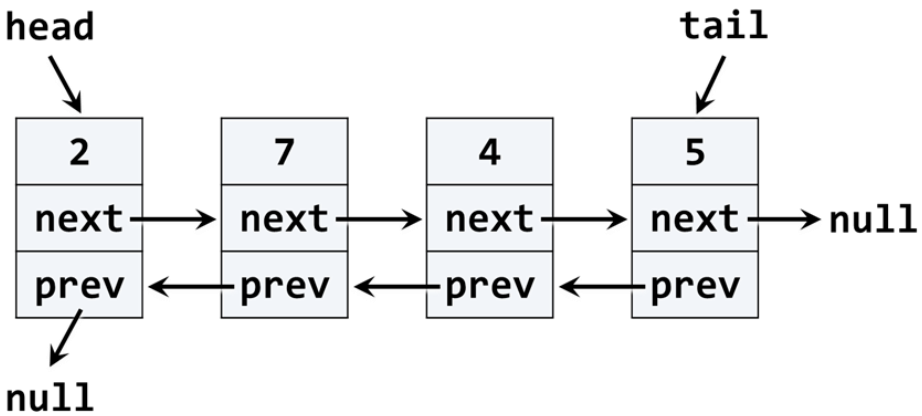
- **`void addFirst(int element)`** – adds an element at the beginning of the collection
- **`void addLast(int element)`** – adds an element at the end of the collection
- **`int get(int index)`** - Returns the element at the specified position in this list
- **`int removeFirst()`** – removes the element at the beginning of the collection
- **`int removeLast()`** – removes the element at the end of the collection
- **`void forEach()`** – goes through the collection and executes a given action
- **`int[] toArray()`** – returns the collection as an array

Feel free to implement your own functionality or to write the methods by yourself.

1. Implement the Custom DoublyLinkedList Class

Details About the Structure

The **Java `LinkedList`** use a doubly linked list which provides a linear data structure that resembles a list, but has different functionalities. Each element in it "knows" about the previous one, if there is such, and the next one, again, if there is such. This is possible, because the **doubly linked list** has **nodes** and each node has **two reference fields** pointing to other nodes and **a value field,** which contains some kind of data. By definition, the **doubly linked list** has a **head** (list start) and a **`tail`** (list end). The typical operations over a linked list are **add** / **remove** an element at **both of the endings** and **traverse**. If you are interested, you can find more detailed information here: https://en.wikipedia.org/wiki/Doubly_linked_list. This figure shows how the structure looks:



Now, that we are somewhat familiar with the doubly linked list, we can proceed to the implementation of our own custom doubly linked list. We will try to implement the main functionalities, but you are free to add other ones if you are interested.

Implementation

The first step when implementing a linked / doubly linked list is to understand that we need **two classes**:

- **`ListNode`** – a class to hold a single list node (its value + next node + previous node)

- **`DoublyLinkedList`** – a class that holds the entire list (its head + tail + operations)

Now, let's create the **`ListNode`** class. It should hold an **`item`** and a reference to its previous and next **node**. We can do that inside the **`DoublyLinkedList`** class, because we will use it only internally inside it. Here is how the class should look:

```
public class DoublyLinkedList {
    private class ListNode {
        private int item;
        private ListNode next;
        private ListNode previous;

        private ListNode(int item) {
            this.item = item;
        }
    }
}
```

The class **`ListNode`** is called a **recursive data structure**, because it references itself recursively. In this case our nodes **`item`** field will be of type **`int`**. After the **Generics** lecture from this module, you can try to change that and make the structure generic, which means it will be able to work with any type.

Implement Head, Tail and Count

Now, let's define the **`head`** and **`tail`** of the doubly linked list. They will be of type **`ListNode`**:

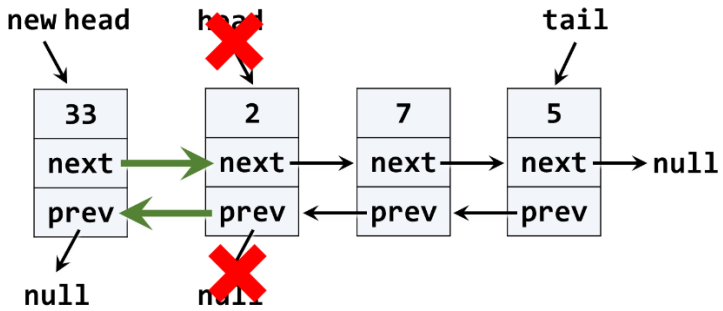
```
public class DoublyLinkedList {
    private class ListNode {...}

    private ListNode head;
    private ListNode tail;
    private int size;
}
```

Implement addFirst(int element) Method

Adding an element at the beginning of the list (before its head) has **two scenarios** (considered in the above code):

- **Empty list** → add the new element as **`head`** and **`tail`** in the same time.
- **Non-empty list** → add the new element as **new head** and redirect the **old head** as second element, just after the new head.



The above graphic visualizes the process of inserting a new node at the start (**head**) of the list. The **red** arrows denote the removed pointers from the old head. The **green** arrows denote the new pointers to the new head.

Next, implement the **`addFirst(int element)`** method:

```
public void addFirst(int element) {
    ListNode newHead = new ListNode(element);
    if (this.size == 0) {
        this.head = this.tail = newHead;
    } else {
        newHead.next = this.head;
        this.head.previous = newHead;
        this.head = newHead;
    }

    this.size++;
}
```

Implement addLast(int element) Method

Next, implement the **`addLast(int element)`** method for appending a new element as the list **`tail`**. It should be very similar to the **`addFirst(int element)`** method. The logic inside it is exactly the same, but we append the new element at the **`tail`** instead of at the **`head`**:

```
public void addLast(int element) {
    ListNode newTail = new ListNode(element);

    if (this.size == 0) {
        this.head = this.tail = newTail;
    } else {
        newTail.previous = this.tail;
        this.tail.next = newTail;
        this.tail = newTail;
    }

    this.size++;
}
```

Implement get(int index) Method

Next, implement the **`get(int index)`** method for returning the element at specified position in this list.

Here is how the method should work step by step.

- Check whether the given index is valid:

```
public int get(int index) {
    // throw if index is out of the array
    checkIndex(index);

}
```

```
private void checkIndex(int index) {
    if (index < 0 || index >= this.size) {
        String message = String.format("Index: %d, Size: %d", index, this.size);
        throw new IndexOutOfBoundsException(message);
    }
}
```

- We can optimize our iterating with condition **if index is less than `size / 2`** iterate from **0** to **index** and return the **last iterated node item**, else iterate from size - 1 to index and again return the **last iterated node item**:

```
if (index <= this.size / 2) {
    ListNode firstNode = this.head;

    for (int i = 0; i < index; i++) {
        firstNode = firstNode.next;
    }

    return firstNode.item;
} else {
    ListNode lastNode = this.tail;
    for (int i = this.size - 1; i > index; i--) {
        lastNode = lastNode.previous;
    }

    return lastNode.item;
}
```

Implement removeFirst() Method

Next, let's implement the method **`removeFirst()` → `int`**. It should **remove the first element** from the list and move its **head** to point to the second element. The removed element should be returned as a result from the method. In case of an empty list, the method should throw an exception. We have to consider the following three cases:

- **Empty list** → throw and exception.
- **Single element in the list** → make the list empty (**head** == **`tail`** == **`null`**).
- **Multiple elements in the list** → remove the first element and redirect the head to point to the second element (**head = `head.NextNode`**).

A sample implementation of **`removeFirst()`** method is given below:

```
public int removeFirst() {
    // throw NoSuchElementException if the size is zero
    checkSize();

    int firstItem = this.head.item;
    this.head = this.head.next;

    if (this.head == null) {
        // Single element in the list
        this.tail = null;
    } else {
        // Multiple elements in the list
        this.head.previous = null;
    }

    this.size--;
    return firstItem;
}
```

```
private void checkSize() {
    if (this.size == 0) {
        throw new NoSuchElementException("The list is empty");
    }
}
```

Implement removeLast() Method

Next, let's implement the method **`removeLast()`** → **`int`**. It should **remove the last element** from the list and move its **`tail`** to point to the element before the last. It is very similar to the method **`removeFirst()`**:

```
public int removeLast() {
    checkSize();

    int lastItem = this.tail.item;
    this.tail = this.tail.previous;

    if (this.tail == null) {
        this.head = null;
    } else {
        this.tail.next = null;
    }

    this.size--;
    return lastItem;
}
```

Implement forEach(Consumer<Integer>) Method

We have a doubly linked list. We can add elements to it. But we cannot see what is inside, because the list still does not have a method to traverse its elements (pass through each one of them, one by one). Now let's define **`forEach(Consumer<Integer>)`** method. In programming, such method is known as a "**visitor**" pattern. It takes as argument a function (consumer) to be invoked for each of the elements in the list. The algorithm behind this method

is simple: start from **head** and pass to the next element until the last element is reached (its next element is **null**). A sample implementation is given below:

```
public void forEach(Consumer<Integer> consumer) {
        ListNode currentNode = this.head;

        while (currentNode != null) {
            consumer.accept(currentNode.item);
            currentNode = currentNode.next;
        }
}
```

For example, if you want to print all of the items you can use the following code:

```
doublyLinkedList.forEach(e -> System.out.println(e));
```

Where **list** is **DoublyLinkedList** type object.

Implement toArray() Method

Now, implement the next method: **`toArray()` → `int[]`**. It should copy all elements of the linked list to an array of the same size. You could use the following steps to implement this method:

- Allocate an array **`int[]`** of size **`this.size`**
- Pass through all elements of the list and fill them to **`int[0]`**, **`int[1]`**, …, **`int[size - 1]`**
- Return the array as result

```
public int[] toArray() {
    int[] array = new int[this.size];
    int counter = 0;
    ListNode currentNode = this.head;

    while (currentNode != null) {
        array[counter++] = currentNode.item;
        currentNode = currentNode.next;
    }

    return array;
}
```

Congratulations! You have implemented your doubly linked list.

"Somewhere, something incredible is waiting to be known." – **Carl Sagan**