Theory: Doubly linked list in Java

39 minutes 0 / 5 problems solved

Doubly linked list is an extension of a linked list. Aside from pointers to the next elements, a doubly linked list also has pointers to previous elements. As you may recall, while going through a linked list we can't turn back since we don't have a pointer to the previous element. In this structure there's no such problem as we can go either direction, **forward** or **backward**.

In this topic, we will see how doubly linked list works.

§1. Implementation

We will consider the following methods for doubly linked lists:

- `addFirst()` – add an element to the beginning of the list;
- `addLast()` – add an element to the end of the list;
- `add(elem)` – add an element before the element 'elem';
- `removeFirst()` – remove an element at the beginning of the list;
- `removeLast()` – remove an element at the end of the list;
- `remove(elem)` – remove an element 'elem'.

Apart from the main methods, we will also discuss some less important ones: `size()`, `isEmpty()`, `toString()`.

The main class `DoublyLinkedList<E>` has a nested class `Node<E>`:

```
static class Node<E> {

    private E value;
    private Node<E> next;
    private Node<E> prev;

    Node(E element, Node<E> next, Node<E> prev) {
        this.value = element;
        this.next = next;

        this.prev = prev;

    }



    Node<E> getNext() {

        return next;

    }



    Node<E> getPrev() {

        return prev;

    }

}
```

Class `Node<E>` shapes the node of the list. It contains the value of generic type and two pointers: to the next and the previous elements. Getters for pointers are set for the manual search through the list, which is required sometimes.

As you probably noticed, the variable `value` has generic type. It means that the value of elements in the list can be anything based on which type is passed to the list's constructor.

The main class that we are going to create:

Current topic:

Doubly linked list in Java

Topic depends on:

- ✕ Doubly linked list
- ✕ Throwing exceptions
- ✕ Generics and Object

Table of contents:

```
class DoublyLinkedList<E>
```

Class `DoublyLinkedList<>` has the following fields:

```
private Node<E> head;
private Node<E> tail;
private int size;

public DoublyLinkedList() {
    size = 0;
}
```

Pointers' head and tail point to the beginning and the end of the list respectively. The variable size keeps the number of elements in the list.

Let's start with the extra methods since they are simpler compared to addition and removal.

```
public Node<E> getHead() {
    return head;
}

public Node<E> getTail() {
    return tail;
}

public int size() {

    return size;

}



public boolean isEmpty() {

    return size == 0;

}



public String toString() {



    Node<E> tmp = head;

    StringBuilder result = new StringBuilder();



    while (tmp != null) {

        result.append(tmp.value).append(" ");

        tmp = tmp.next;

    }



    return result.toString();

}
```

Getters for receiving pointers to the start and the end of the list, as well as getters for the next and previous elements are required for manual management of the list. Methods `size()` and `isEmpty()` are obvious. Method `toString()` goes through the list from left to right and prints values to console. It's not difficult to modify it to go through the list in the opposite direction.

§2. Addition

```
void addFirst(E elem) {

    Node<E> tmp = new Node<>(elem, head, null);

    if (head != null) {
        head.prev = tmp;
    }

    head = tmp;



    if (tail == null) {

        tail = tmp;

    }

    size++;

}



void addLast(E elem) {



    Node<E> tmp = new Node<>(elem, null, tail);



    if (tail != null) {

        tail.next = tmp;

    }



    tail = tmp;



    if (head == null) {

        head = tmp;

    }

    size++;

}



void add(E elem, Node<E> curr) {



    if (curr == null) {

        throw new NoSuchElementException();

    }



    Node<E> tmp = new Node<>(elem, curr, null);



    if (curr.prev != null) {

        curr.prev.next = tmp;

        tmp.prev = curr.prev;

        curr.prev = tmp;
```

```
4
5       } else {
4
6           curr.prev = tmp;
4
7           tmp.next = curr;
4
8           head = tmp;
4
9       }
5
0       size++;
5
1   }
```

The first two operations are very similar. In the first case, we create a new node, then change the head pointer to it and the double link to the former head element. The second case is basically the same, we just apply the steps to the tail instead.

Generic addition is slightly more complex. First is if the added element doesn't have a previous one, that is, if we add at the start (else branch). In other cases we insert the element between two others. Evidently, we need to change next pointer of previous element to point to the element we add, while doing the same to prev pointer of the next element. We also need to make sure to properly initialize pointers for the added element. The pointer from the current element to the next one is initialized in node constructor, and three other pointers are changed in the first `if` branch. Since all methods have only conditional operators and assignment operations, it has $O(1)$ asymptotic, which is very good.

§3. Removal

```
public void removeFirst() {
    if (size == 0) {
        throw new NoSuchElementException();
    }
    head = head.next;
    head.prev = null;
    size--;
}


public void removeLast() {

    if (size == 0) {

        throw new NoSuchElementException();

    }

    tail = tail.prev;

    tail.next = null;

    size--;

}



public void remove(Node<E> curr) {

    if (curr == null)

        throw new NoSuchElementException();

    if (curr.prev == null) {

        removeFirst();

        return;

    }

    if (curr.next == null) {

        removeLast();

        return;

    }

    curr.prev.next = curr.next;

    curr.next.prev = curr.prev;

    size--;

}
```

For removing from the start or the end of the list, we change the head/tail pointer appropriately and put `null` for pointer in next/previous element. In `remove()` method we call these methods in order to avoid duplicating the code. In general, when the current element has both neighbors, you should change their pointers. The methods have the same asymptotic as addition: $O(1)$.

§4. Usage

Let's see how we could use the class we got. Make an instance of the class and apply various operations to it:

```
DoublyLinkedList<Integer> dll = new DoublyLinkedList<>();
dll.addFirst(10);
dll.addFirst(34);
dll.addLast(56);
dll.addLast(364);
dll.add(1, dll.getHead().getNext());
dll.add(2, dll.getHead());
System.out.println(dll.toString());
dll.remove(dll.getHead().getNext().getNext());

System.out.println(dll.toString());

dll.remove(dll.getHead());

System.out.println(dll.toString());

dll.remove(dll.getTail());

System.out.println(dll.toString());
```

The result of executing that code should be as follows:

```
2 34 1 10 56 364
2 34 10 56 364
34 10 56 364
34 10 56
```

§5. Conclusion

A big advantage of double linked lists in comparison to arrays is that all addition and removal operations have $O(1)$ asymptotic and you don't need any relocation or data copying. But with this advantage comes a disadvantage: it is impossible to access elements via index in such lists, unlike in arrays. Therefore, to find any element you'll need to go through the list, which can take $O(n)$ time in the worst case. Considering all pros and cons, we can conclude that each data structure is appropriate for different tasks. That is why, when you think about ways to solve a particular task, you'll need to understand which operations suit the situation better.

Report a typo

13 users liked this theory. **2** didn't like it. **What about you?**

😍 🙂 😐 😕 😡

Start practicing

Comments (5) Hints (2) Useful links (0) Show discussion