Doubly LinkedList

Doubly LinkedList

The elements of a linked list are referred to as *nodes*. In a *doubly linked list*, nodes also have pointers to the previous node:

```
+---------+     +---------+     +---------+     +---------+
|         |---->|         |---->|         |---->|         |
| node 0  |     | node 1  |     | node 2  |     | node 3  |
|         |<----|         |<----|         |<----|         |
+---------+     +---------+     +---------+     +---------+
```

Doubly LinkedList (Cont'd)

You need to keep track of where the list begins. That's usually done with a pointer called the *head*:

```
          +---------+    +---------+    +---------+    +---------+
head --->|         |--->|         |--->|         |--->|         |---> nil
          | node 0 |    | node 1 |    | node 2 |    | node 3 |
 nil <---|         |<---|         |<---|         |<---|         |<--- tail
          +---------+    +---------+    +---------+    +---------+
```

Concerning `wellformed` method

0. comparator is not null
1. If head or tail is null then both are null.
2. If head exists it is first in list.
3. Every Painting with a next is the previous of its next.
4. If tail exists it is last in same list as head.
5. manyItems is the number of paintings in the list
6. Every Painting with a next is lesser than or equal to its next according

Structure

We have to manage `_previous` pointer

```
public class DoublyLinkedList<T> {
    Node _head, _tail;

    private static class Node {
        T _data;
        Node _previous, _next;

        Node(T p, Node previous, Node next) {
            _data = data;
            _previous = previous;
            _next = next;
        }
    }
}
```

Exercise: `add(T value)`

```java
public void add(T value) {
    // TODO:
    // 1) if head or tail are null, i.e. list is empty
    //
    //
    //
    // 2)  add item at the tail of list
    // 2') add item at the head of list
}
```

Solution: `add(T value)`

```java
public void add(T value) {
    if (head == null) {
        tail = head = new Node(value, null, null);
    } else {
        // pay attention to right recursive assignment evaluation
        // a = (b = (c = (e = f)))
        //
        tail = tail.next = new Node(value, tail, tail.next);
        // head = head.previous = new Node(value, head.previous, head);
    }
}
```

Exercise: `remove(T value)`

```java
// Return false if remove failed!
public boolean remove(T value) {
    // case 0) if item is nul or list is empty!
    //       *) fix up new head's previous
    //       *) fix up tail if needed
    // case 1) if head is the target
    //       *) fix up new tail's previous
    // case 2) if tail is the next
    // case 3) item is somewhere in the middle of the list
}
```

Solution: `remove(T value)`

```java
// Return false if remove failed!
public boolean remove(T data) {
    if (value == null || head == null) return false;

    if (head.data.equals(value)) {
        head = head.next;
        if (head != null) head.previous = null;
        else tail = null;
    } else if (tail.data.equals(value)) {
        tail = tail.previous;
        if (tail != null) tail.next = null;
    } else {
        Node n = getNode(value);
        if (n == null) {
            return false;
        } else {
            n.previous.next = n.next;
            n.next.previous = n.previous;
        }
    }

    return true;
}
```

Bubble sort

```
repeat
    if itemCount <= 1
        return
    hasChanged := false
    decrement itemCount
    repeat with index from 1 to itemCount
        if (item at index) > (item at (index + 1))
            swap (item at index) with (item at (index + 1))
            hasChanged := true
until hasChanged = false
```

Lab assignment #5: