Linked List data structure

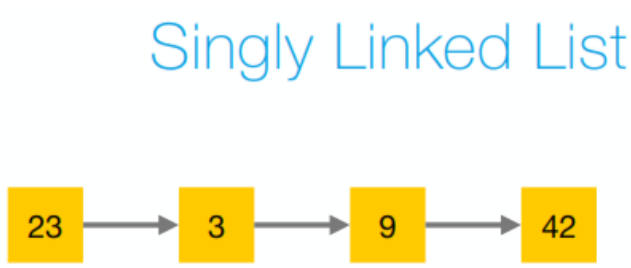
This article is part of Marcus Biel's free Java course focusing on clean code principles. In this one, we will talk about the `Linked List` data structure.

Terminology

First of all, let's have a look at the term "`Linked List`". Why is it called that? For the term **link** – as an analogy – think of a hyperlink on a web page which can bring you from one page to the next. A `List` is a collection of related objects. Think of a shopping list: it contains every item that you plan to buy. A `Linked List` is a collection of related objects where a link brings us from one item to the next.

Singly-Linked Lists

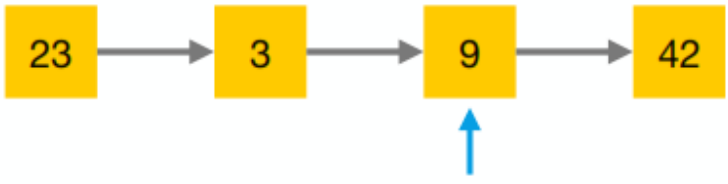
Technically, an item in our `Linked List`, or simply a list entry, is stored in a `Node`. In our simple Figure 1 example, this entry is the number 23. Each node has a *link* or *pointer* to the next element of the list – the next node. For every item that we want to add to the `Linked List`, we create a node and store the item in it. The first item of the list is usually called the *head* while the last item is called the *tail*.



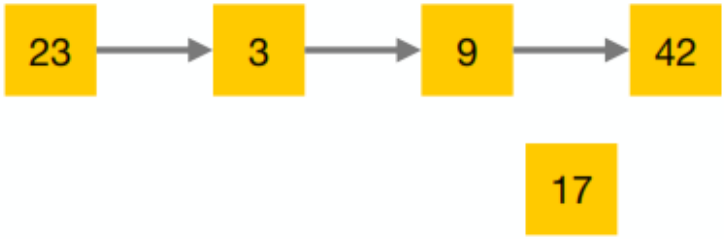
[1]

`Linked List` is a dynamic data structure that can hold any number of elements, as long as enough memory is available. After the list is created, navigating through the list is done by calling a function like `next()` which uses the link to go from one item to the next. This type of Linked List is called a Singly-Linked List because the links only go in one direction, i.e. from the head of the list to the tail. This makes it so that when we navigate to the previous element, e.g. from the element 42 to element 9, as in Figure 1, we would have to go back to the head of the list and call the `next()` function on every single element until we reach 9. If the list contains a lot of elements, this may take some time.

Inserting an element after the current one is relatively easy with a singly-linked list. Say we start with 9 as our current element.

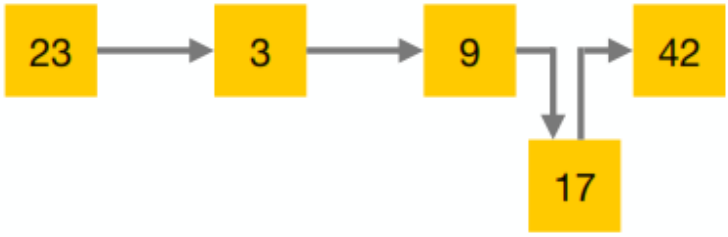


We create a new Node containing the new element.



We link to the new element “17” from the current element “9”.

We add a link pointing from the new “17” element to the existing “42” element.



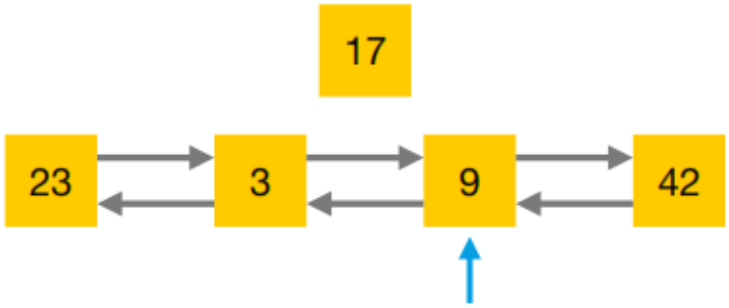
With that, the element is now inserted.

Inserting the same element before the current one is possible in a singly-linked list, but usually not very efficient. It will require us to navigate to the previous element, starting from the beginning of the list as shown before. Removing an element from a singly-linked list has the same issue – it is possible, but generally not efficient.

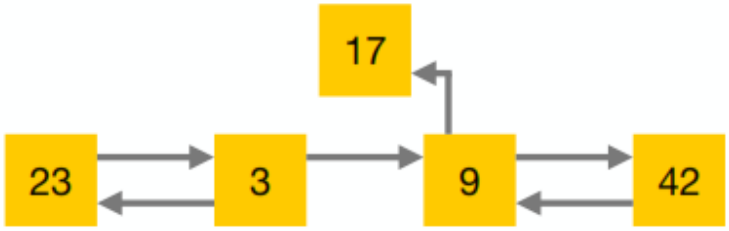
These operations become much easier when we add a second link to each node pointing to the previous one. This allows us to navigate via both directions of the list, forwards and backwards. However, the extra link comes at the cost of some extra memory, as well as some more time to build the more complex structure. Whether the overhead is justified depends a lot on the use case. If performance is an issue, it is advisable to test different options.

Doubly-Linked List

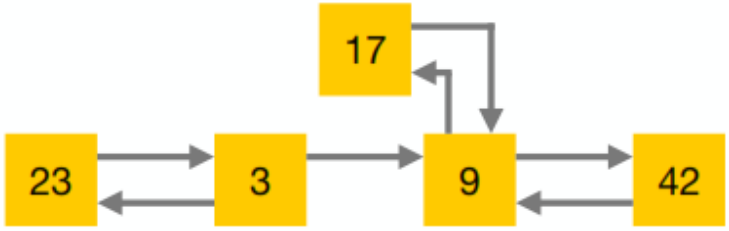
A `Linked List` that contains nodes that provide a link to the next and the previous nodes is called a **doubly-linked list**. For every element added to a doubly-linked list, we need to create two links, making doubly-linked lists somewhat more complicated than their singly-linked counterparts. Navigating both ways in a doubly-linked list is easier, but it's done at the cost of a more complex structure. Thanks to the two-way link structure, adding or removing an element before or after the current element is relatively easy. To demonstrate this, we add the element "17" before the current element "9".



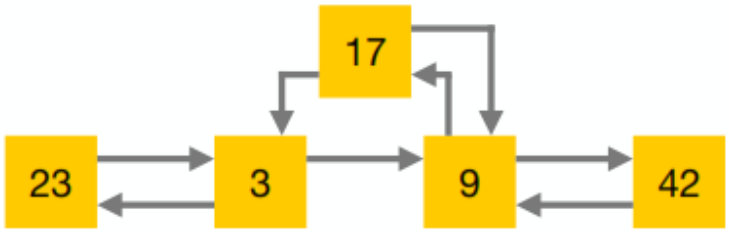
The backlink from "9" to "3" is removed and replaced with a backlink to the new element, "17".



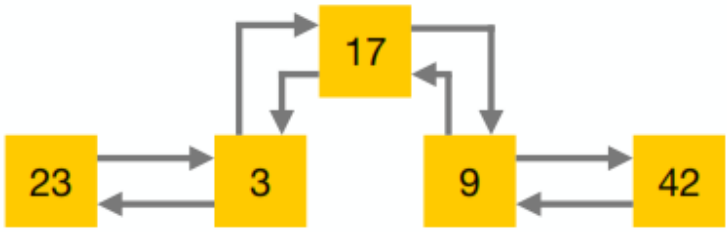
From the node with "17", we make a forward link to "9".



Next, we place a back link from "17" to "3".



The old link from “3” to “9” is replaced by a link from “3” to “17”.



Removing an element from a doubly-linked list follows the same steps as when we insert an element into the list, just in reverse order.

Code Excerpts

Node

```
 1 public class Node<E> {
 2   private E item;
 3   private Node previous;
 4   private Node next;
 5
 6   public Node(Node previous, E element, Node next) {
 7     this.item = element;
 8     this.next = next;
 9     this.previous = previous;
10   }
11
12   public E item() {
13     return item;
14   }
15
16   public Node previous() {
17     return previous;
18   }
19
20   public Node next() {
21     return next;
22   }
23   [...]
24 }
```

Listing 1

LinkedList

```
1 public class LinkedList<E> {
2   private Node currentNode;
3   private Node head;
4
5   public E get (int index) {...}
6   public boolean add (E e) {...}
7   public E remove (int index) {...}
8
9   [...]
10 }
```

Listing 2

Let's look at the code excerpts in Listings 2-3 of a singly-linked list `Node` and its use in the `LinkedList` class. We have a method for retrieving the item contained in a Node in `item()` and a method to go to the next node with `next()`. The node of a doubly-linked list is very similar, but a bit more complex in that it has a reference to the previous node. A `Linked List` usually contains a link to the **head** of the list. The methods of the `Node` class are used to navigate through the list. The concrete implementation of the methods to get, add or remove elements depends on the type of the node, the details of which are hidden from the users of the list. Besides the direct link to the current and head nodes, a `Linked List` may also provide a direct link to its **tail**. This is common for a doubly-linked list, but is also useful to have in a singly-linked one.

Application

The `LinkedList` class implements and forms the foundation of many other data structures, some of which include **`List`, `Queue`, `Stack`, and Double-ended queues, a.k.a. `Deque`**.

When comparing implementations based on singly or doubly-linked lists, there is in fact no overall winner. Both have their pros and cons, making one of them the better choice in some instances, but less than ideal in others. For each data structure mentioned above, we will decide whether a singly or a doubly-linked implementation is ideal. Although it is possible to implement these data structures with arrays for instance, our focus will be on the flexibility of the `Linked List` to implement them.

`Lists` usually require random access to its elements so a doubly-linked list is the better choice. In a "first-in first-out" (FIFO) `Queue`, new elements are inserted at and removed from the tail of the queue, and random access is not required. For `Queues`, singly-linked lists are the better choice. A Stack provides a "last-in last-out" (LIFO) order on operations. Elements are added and removed from the head of the `Stack`, making it even simpler than a Queue. Therefore, singly-linked lists are better for `Stacks`. A double-ended queue or deque is a very dynamic data structure. It allows access, addition and removal from both ends. This makes a doubly-linked list the ideal choice.