Dynamic Structures: Linked List

Learning Targets:

- You understand the concept of the linked list and can implement this

1 A further basic data structure: Linked List

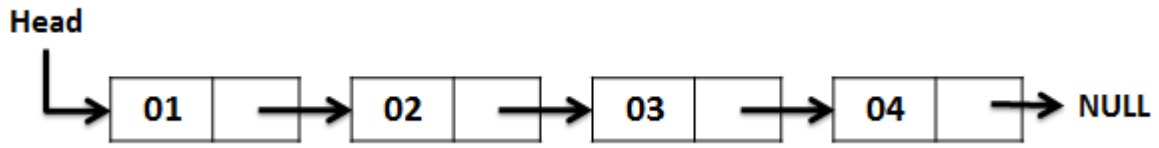
Working with a fixed number of elements quickly has its limits, especially when we want to work with a dynamic (flexible) size of elements.
A classic implementation to solve this problem is the linked list, which we'll implement as our own version (thereby NOT using Java's standard libraries).

We will see that this form of list has its advantages and disadvantages. One big advantage is that you easily add or remove elements from the list. However, when you want to access a particular element quickly, the process is much slower than say in a primitive array.

Principle of a linked list:

Each element points to the next element. This „pointing" works as a link, since an element knows its next element. An element is therefore a node which always also has its next element as a part.

The *node* in this definition is an abstract entity that might hold any kind of data, in addition to the node reference that characterizes its role in building linked lists. As with a recursive program, the concept of a recursive data structure can be a bit mindbending at first, but is of great value because of its simplicity.



Quelle: CrunchifyCode

With object-oriented programming, implementing linked lists is not difficult. We start

with a class that defines the node structure:

```
private class Node{

      Item item;
      Node next;
}
```

A *Node* has two instance variables: an *Item* (a parameterized type) and a *Node*. We define *Node* within the class where we want to use it, and make it private because it is not for use by clients. As with any data type, we create an object of type *Node* by invoking the (no-argument) constructor with `new Node()`. The result is a reference to a *Node* object whose instance variables are both initialized to the value null.

Example of the inner class:

```
private class Node{

            //these are private
            private Object item;
            private Node next;

            //constructor
            public Node (Object value){
                  next = null;
                  item = value;
            }


            //another constructor
            public Node(Object value, Node nextValue){
                  next = nextValue;
                  item = value;
            }

            //add setter and getter:
            …

….
}
```

The basic constructor sets the next element to *null*, because it only implements the current element.

Don’t forget to implement setter and getter methods, since this inner class represents the pure datastructure.

Our linked list class uses the *Node* class to create the list.

Your implementation can look something like this:

```
public class MyLinkedList {

      //reference to the head node
      private Node head;
      private int listCount;  //counter used for looping



      //constructor
      public MyLinkedList(){

      //when initialised it's an empty list, so the reference to the
      //head node is set to a new node with no data:

            head = new Node(null);
            listCount = 0;
      }

…
}
```

Constructor creates the head of the list

When we initialise the list, we simply create a node – and add a value if necessary. The above example begins the list with the value null.

Adding Elements:

There are different ways how you can add elements in a linked list. This example shows how an element is added to the end of the list. You use the "head" element as a starting point, since this should be our first element in the list.

The you proceed through each node, until you reach the end of the list.
Example:

```
public void add (Object value){

      Node newElement = new Node(value);
      Node current = head;  //head is the current node

//just to be sure we are at the end of the list, we loop through the
elements:


      while (current.getNext() != null){
```

```
            current = current.getNext();
      }
      //add the new element to the end of the list:
      current.setNext(newElement);
      //increment list counter:
      listCount++;
}
```

If your node does not point to a next element you have reached the end and can add the new element.

How would you add elements to the beginning of the list?

2 Exercises

2.1 Implement your own linked list

Implement your own linked list by using an *inner class* as datastructure.

a) Implement a method *add* and do some tests to see if your linked list works.
b) Add a further method *showElements* which prints out each value of each element.
c) Extend your linked list: Write a method that shows the values in reversed order. Something like this:

Input: `Hello, Arnold, how are you doing?`
Output: `doing? you are how Arnold, Hello,`

2.2 Removing Elements

Extend your class so you can remove elements.

Depending on your implementation, you must write an algorithm which properly iterates through each element.
In this example the list is iterated from left to right. The current node is set with the next value (*head* is the first element and has no value).

```
Node current = head.getNext(); //head item is always null
```

Now you can iterate. In pseudocode:

```
While (next Element not null && current Element does not have searched
item){
      If (next Element has searched item){
            If(the following Element is not null) then
```

```
                Next element = next of next element
            Else
                Set next element to null
        }
        Current element = next element
}
```

2.3 Adding / removing elements in a specified position

Now extend your list so it can add or remove elements at a specified position. For example, we want to change following sentence:

„To be or not to be, that is the question".

*„To be or not to be, **my dear Hamlet**, that is the question".*

➔ Show your solutions to your teacher.

2.4 Further exercises with linked lists

a) Write another program which can shift elements (as we did with simple arrays). The last entered value is the first value in the list. Use your linked list to do this. (hint: structuring data in this manner is easily done in another dynamic data structure. Which one?)

➔ Compare this implementation with the one you did with primitive arrays. Which solution is better.

b) Write a method insertAfter() that takes two Node objecsts as arguments and inserts the second after the first in the list (and does nothing if either argument is null).

➔ Show your solutions to your teacher. If you found an alternative way to do task a) you might want to show this in class.

3 Additional Exercise with Linked List

3.1 Reverse your list

Write a method which takes the linked list and reverse the order of elements. Obviously you use your linked list.

Upload your solution on BSCW.