Dynamische Datenstrukturen: Verkettete Liste

Lernziele:

- Sie kennen das Konzept einer verketteten Liste und können diese mittels Beispiele umsetzen

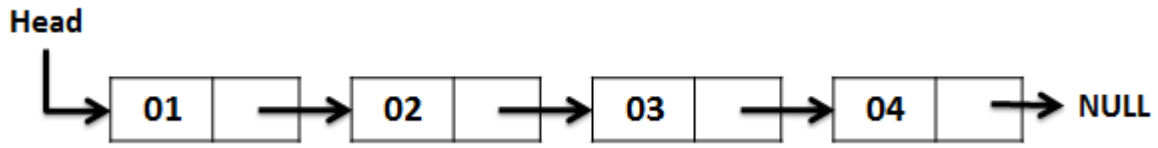
1 Eine weitere elementare Datenstruktur: Verkettete Liste

Das Arbeiten mit einer fixen Grösse von Elementen stösst sehr schnell an seine Grenzen. Vor allem wenn wir es mit dynamischen Datenmengen zu tun haben.
Hier kommt eine weitere elementare Datenstruktur zur Anwendung, welche ihre Grösse verändern kann (was wir bei einem Array nicht können). Die Verkettete Liste („LinkedList") kann mit dynamischen Datenmengen umgehen.
Ein weiterer Vorteil ist, dass sie eine höhere Flexibilität ermöglicht, indem die Elemente in effizienter Weise umgeordnet werden können. Dafür ist ein schneller Zugriff auf ein Element nicht möglich.

Prinzip einer verketteten Liste:

Jedes Element zeigt auf das nächste Element. Das „Zeigen" ist zugleich eine Verknüpfung, weil somit jedes Element das nächste Element kennt.

Ein Element ist somit ein Knoten („Node"), welches durch Verkettung auf den nächsten Knoten verweist („next").



Quelle: CrunchifyCode

In einer OO-Sprache ist die Implementierung von verketteten Listen nicht sehr schwierig. Wir beginnen mit einer inneren Klasse, welche die Knotenabstraktion definiert:

```
private class Node{

      Item item;
      Node next;

}
```

Ein *Node*-Objekt besitzt zwei Instanzvariablen: eine vom Typ *Item* (ein parametrisierter Typ, den Sie selber bestimmen können) und eine vom Typ *Node*. Wir definieren *Node* direkt in der Klasse, in der wir den Typ verwenden wollen, und zwar als *private*, da Clients keinen Zugriff darauf erhalten sollen.

Beispiel für die innere Klasse:

```
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:
            …

….
}
```

Der Basis-Konstruktor setzt das nächste Element auf *null*, weil es nur das aktuelle Element initialisiert.

Vergessen Sie nicht, *set*- und *get*-Methoden zu definieren, damit sie auf die Daten zugreifen können.
Sie benötigt aber sonst keine Methoden (weil es sich um eine reine Datenstruktur handelt).

Ausserhalb der inneren Klasse haben wir unsere VerketteteListe-Klasse. Hier werden die Knoten (Nodes) verkettet.

Ungefähr so:

```
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;
      }

…
}
```

Konstruktor setzt den *Head* der Kette:

Ganz wichtig ist, dass im Konstruktor eine Liste mit einem Element erstellt wird. Wir verwenden dabei das vorderste Element („Head"). Das vorderste Element hat keine Verweis auf ein Nächstes, deshalb setzen wir den Knoten auf `null`.

Elemente hinzufügen:

Prinzip: ein neues Element wird ans Ende der Liste gestellt. Ausgangspunkt ist das Head-Element in der Liste. D.h. wir fangen mit dem Head-Element an, definieren es als aktuelles Element („current") und iterieren dann bis ans Ende der Liste:

Im Code:

```
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++;
}
```

Wenn wir das letzte Element erreicht haben (d.h. es zeigt nicht auf ein weiteres Element), wird das neue Element hinzugefügt. Das geschieht in den letzten vier Zeilen. Der Zähler wird um 1 erhöht.

2 Aufgaben

2.1 Eigene verkettete Liste implementieren

Schreiben Sie eine eigene Klasse für verkettete Listen mit der entsprechenden inneren Klasse für die Datenstruktur.

a) Implementieren Sie die Methode *add* und testen Sie, ob Ihre Liste funktioniert, indem der Benutzer Wörter eingeben kann.
b) Erstellen Sie eine weitere Methode *showElements*, welche diese mittels einer Schleife wieder ausgibt.
c) Erweiterung: Wie könnte man eine Methode schreiben, welche die Elemente in umgekehrter Reihenfolge ausgibt? Es würde so aussehen:

Eingabe: `Hello, Arnold, how are you doing?`
Ausgabe: `doing? you are how Arnold, Hello,`

2.2 Elemente entfernen

Erweitern Sie Ihre Klasse, so dass Elemente entfernt werden können. Und zwar soll einfach das letzte Element in der Liste entfernt werden.

Achten Sie auf folgendes:
Die Liste iteriert von links nach rechts. Dh. Das erste aktuelle Element wird mittels dem *Head* gesetzt:

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

Nun können Sie von links nach rechts iterieren, und zwar so (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 (übernächstes element)
            Else
                  Set next element to null (keine weiteren Elemente)
      }
      Current element = next element
}
```

2.3 Elemente an einer bestimmten Position einfügen / entfernen

Jetzt erweitern wir unsere Liste, damit wir auch Elemente an einer bestimmten Position einfügen oder auch löschen können. Erstellen Sie eine verkettete Liste, welche aus String-Elementen besteht. Wir wollen folgenden Satz anpassen:

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

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

➔ Zeigen Sie Ihre Lösungen der Lehrperson.

2.4 Weitere Aufgaben mit verketteten Listen

a) Schreiben Sie nochmals ein Programm, dass Elemente von einem Benutzer in eine Liste abfüllt und dabei die Elemente nachrutscht. Dh. Das zuletzt eingegebene Element ist das erste Element in der Liste. Verwenden Sie dieses Mal Ihre Verkettete Liste. Welche Struktur ist einfacher? Der simple Array oder ihre Liste?

➔ Dokumentieren Sie beide Varianten (d.h. Sie haben beide Code-Varianten in separaten Klassen und verwenden Sie eine Start-Klasse, welche beide Varianten benützen kann).

b) Schreiben Sie eine Methode *insertAfter(),* das zwei Node-Objekte einer verketteten Liste als Argumente übernimmt und das zweite nach dem ersten in die Liste einfügt.

➔ Zeigen Sie Ihre Lösungen der Lehrperson.

2.5 Suchen in einer verkettete Liste vs. in einem primitiven Array

Ein grosser Vorteil der verketteten Liste ist die dynamische Grösse. Die Anzahl Elemente müssen nicht fix definiert sein. Anderseits ist das Suchen in einer verketteten Liste viel mühsamer als in einem Array. In einem Array können wir jederzeit via Index direkt auf ein Element zugreifen. In der verketteten Liste müssen wir die ganze Liste durchsuchen.

Implementieren Sie eine Suche in der verketteten Liste. Dh. es wird nachgeschaut, ob ein bestimmtes Element in der Liste existiert.

3 Zusätzliche Aufgabe zu verkettete Liste

3.1 Verkettete Liste umkehren

Schreiben Sie eine Methode, die eine bestehende verkettete Liste nimmt und die Elemente umkehrt. Verwenden Sie Ihre verkettete Liste.

Stellen Sie Ihre Lösung aufs BSCW.