Prof. Dr. Christian Kohls
Alexander Dobrynin, M.Sc.

April 6, 2022
SoSe 2022

Algorithmen und Programmierung 2
Praktikum 3 - Verkettete Liste

Dieses Übungsblatt beschäftigt sich mit den theoretischen Grundlagen und der Implementierung einer einfach verketteten Liste. Sie werden die Basistruktur, grundlegende Methoden und komplexere Methoden einer einfach verketteten Liste implementieren. Am Ende setzen Sie sich mit der Laufzeiteffizienz ihrer Algorithmen auseinander.

Für die Aufgaben sind die folgenden Videos aus der Playlist auf YouTube relevant:

- "38. Verkettete Listen - Erste Schritte" bis "40. Listen-Operationen, Listen durchlaufen".
- "44. Generics / Typparameter" bis "46. Innere Klassen und anonyme Objekte".
- "52. Laufzeitperformanz".
- "59. Funktionen höherer Ordnung". Hinweis: Hier geht es zwar um Bäume anstelle von Listen, allerdings sind die Konzepte von Funktionen höherer Ordnung immer gleich.

Bzw. folgende Kapitel aus dem Buch Programmieren lernen mit Kotlin:

- Kapitel "27 Listen selbst implementieren". Hinweis: Das Kapitel ist freiwillig, hilft aber, die grundsätzlichen Konzepte und Methoden einer Liste zu verstehen.
- Kapitel "28 Verkette Liste".
- Kapitel "12.13 Funktionen höherer Ordnung" bis inklusive Kapitel "12.13.1 Funktionen, die Funktionen als Parameter akzeptieren". Hinweis: Hier werden Funktionen höherer Ordnung als solches beschrieben.
- Kapitel "25 Funktionen höherer Ordnung für Datensammlungen". Hinweis: Hier werden Funktionen höherer Ordnung auf Listen und co. beschrieben.
- Kapitel "30 Optimierung und Laufzeiteffizienz".

Contents

1 Grundlagen zur einfach verketteten Liste

Gegeben sei die Basistruktur einer einfach verketteten Liste, die generisch über den Typparameter `T` ist.:

```
class LinkedList<T> {
  data class Node<T>(val data: T, var next: Node<T>?)

  private var first: Node<T>? = null
}
```

1.1 Theorie

Machen Sie sich mit dem Konzept einer einfach verketteten Liste vertraut, indem Sie folgende Fragen beantworten:

- Wie werden Elemente der Liste hinzugefügt?
- Wodurch wird das Ende einer einfach verketteten Liste gekennzeichnet?
- Wann ist die verkettete Liste leer?
- Wann ist eine einfach verkettete Liste anstelle einer `MutableList` (bezogen auf Kotlin) geeigneter und wann nicht?

1.2 Visualisierung

Visualisieren Sie eine verkettete Liste mit 3 Knoten. Verwenden Sie `Int` Werte als Daten.

2 Grundlegende Methoden implementieren

In dieser Aufgabe werden Sie grundlegende Methoden einer einfach verketteten Liste implementieren. Der Name, die Parameter und die Beschreibung sind vorgegeben. Den Rückgabetyp müssen Sie sich aus der Beschreibung herleiten.

2.1 Basismethoden

Zuerst werden die simpelsten Methoden implementiert:

- `isEmpty()` => Gibt zurück, ob die Liste leer ist oder nicht.
- `addFirst(data: T)` => Fügt `data` als neuen Knoten (`Node`) am Anfang der Liste hinzu.
- `getFirst()` => Gibt das Element (`data`) des ersten Knotens zurück. Falls die Liste leer ist, wird `null` zurückgegeben.
- `clear()` => Leert die Liste.

Legen Sie eine `main` Funktion an und testen Sie Ihre Methoden. Hier ein Beispiel:

```
fun main() {
  val list = LinkedList<String>()
  println(list.isEmpty()) // true

  list.addFirst("Apfel")
  list.addFirst("Banane")
  list.addFirst("Birne")

  println(list.isEmpty()) // false
  println(list.getFirst()) // "Birne"

  list.clear()
  println(list.getFirst()) // null
  println(list.isEmpty()) // true
}
```

In den Kommentaren steht jeweils das, was `println` in der Konsole ausgibt. Sorgen Sie dafür, dass Sie die gleichen Ergebnisse haben. Ansonsten haben Sie einen Fehler in Ihrer Implementierung.

2.2 Methoden mit Iterationen

In dieser Teilaufgabe implementieren Sie Methoden, die über die Liste iterieren. Machen Sie sich damit vertraut, wie eine einfach verkettete Liste iteriert (durchlaufen) wird. Der Code dazu ist:

```
// ...
var runPointer = first
while (runPointer != null) {
  // mache etwas mit runPointer

  runPointer = runPointer.next
}
// ...
```

Implementieren Sie nun die folgenden Methoden:

- `size()` => Gibt zurück, wie viele Elemente die Liste hat.
- `get(index: Int)` => Gibt das Element (`data`) an der Stelle `index` zurück. Der Index beginnt bei 0. Wenn es kein Element für den Index gibt, wird eine `IndexOutOfBoundsException` geworfen.
- `getOrNull(index: Int)` => Macht das gleiche wie `get(index: Int)`, nur dass `null` zurückgegeben wird, anstelle eine Exception zu werfen, wenn es für den Index kein Element gibt.
- `addLast(data: T)` => Fügt `data` als neuen Knoten am Ende der Liste hinzu.

Testen Sie nun die implementierten Methoden. Hier ein paar Beispielaufrufe:

```
fun main() {
  val list = LinkedList<String>()
  println(list.size()) // 0

  list.addFirst("Apfel")
  list.addFirst("Banane")
  list.addFirst("Birne")
  println(list.size()) // 3

  println(list.getOrNull(0)) // Birne
  println(list.getOrNull(1)) // Banane
  println(list.getOrNull(2)) // Apfel
  println(list.getOrNull(3)) // null

  list.addLast("Pfirsich")
  println(list.getOrNull(3)) // Pfirsich
  println(list.getOrNull(4)) // null
}
```

Auch hier steht in den Kommentaren wieder das, was die jeweilige Zeile auf der Konsole ausgibt. Sorgen Sie auch hier dafür, dass Sie die gleichen Ergebnisse haben. Ansonsten haben Sie einen Fehler in Ihrer Implementierung.

3 Funktionen höherer Ordnung implementieren

In dieser Aufgabe implementieren Sie Funktionen höherer Ordnung (*high order functions*) auf der verketteten Liste.

3.1 Begriffsdefinition

Machen Sie sich mit dem Konzept von Funktionen höherer Ordnung vertraut. Was sind Funktionen höherer Ordnung? Geben Sie ein Beispiel.

3.2 ForEach

Das Iterieren über die Liste sieht fast immer gleich aus. In dieser Teilaufgabe extrahieren Sie diese Gemeinsamkeit in eine Funktion höherer Ordnung namens `forEach`. Der Unterschied wird als Lambda übergeben.
Implementieren Sie die `forEach` Funktion, die jedes Element aus der Liste mit der übergebenen Funktion `action` aufruft:

```
class LinkedList<T> {
  // ...

  fun forEach(action: (T) -> Unit) {
    // Durchläuft die Liste und ruft die Funktion action mit jedem
       Element ein Mal auf
  }
}
```

Testen Sie die `forEach` Methoden, indem Sie diese in der `main` Funktion verwenden. Im folgenden Beispiel wird jedes Element mit `println` auf der Konsole ausgegeben:

```
fun main() {
  val list = LinkedList<String>()
  list.addFirst("Apfel")
  list.addFirst("Banane")
  list.addFirst("Birne")

  list.forEach { element ->
    println(element)
  }
}
```

Die Ausgabe in der Konsole ist dann:

```
Birne
Banane
Apfel
```

3.3 Weitere Funktionen höherer Ordnung

a) Implementieren Sie die `size` Methode aus Aufgabe 2.2 mit der `forEach` Methode. Es soll nach wie vor zurückgegeben werden, wie viele Elemente in der Liste sind.

b) Implementieren Sie die Methode `countWhere(condition: (T)-> Boolean): Int`. Die Methode gibt zurück, wie viele Elemente die Bedingung (condition) erfüllen.
Hier ein Beispiel für die Verwendung:

```
fun main() {
  val list = LinkedList<String>()
  list.addFirst("Apfel")
  list.addFirst("Banane")
  list.addFirst("Birne")

  val fruitsStartWithB = list.countWhere { element ->
    element.startsWith("B")
  }

  println(fruitsStartWithB) // 2
}
```

`fruitsStartWithB` ist 2, weil es zwei Elemente in der Liste gibt, die mit `"B"` anfangen.

4 Laufzeiteffizienz (O-Notation)

In dieser Aufgabe bewerten Sie die Laufzeiteffizienz einiger Methoden, die Sie in dieser Praktikumsaufgabe geschrieben haben.

4.1 Begriffsdefinition

Machen Sie sich mit dem Konzept der Laufzeiteffizienz (O-Notation) vertraut. Was bedeutet es, wenn ein Algorithmus eine Laufzeit von O(1), O(n) oder $O(n^2)$ hat?

4.2 Bestimmung

Welche Laufzeit haben die folgenden Methoden einer verketteten Liste? Begründen Sie Ihre Antwort:

- `addFirst(data: T)`
- `addLast(data: T)`
- `size()`
- `getFirst()`
- `get(index: Int)`
- `forEach(action: (T)-> Unit)`