

Tutorium Grundlagen: Algorithmen und Datenstrukturen

Übungsblatt Woche 1

Organisatorisches

Leistungsnachweis

- Klausur (voraussichtlich) 24.07.2024
 - schriftliche Prüfung
 - Dauer: 90 Minuten
 - Erlaubtes Hilfsmittel: hand-beschriebenes DIN A4 Blatt
- Wiederholungs-Klausur (voraussichtlich) Oktober 2024
- Warnung: für IN0007 in München/Garching anmelden, nicht für Heilbronn (INHN0008)! Auch dort gibt es im SS eine Vorlesung "Grundlagen: Algorithmen und Datenstrukturen".
- Vorbereitung durch aktive Teilnahme an Vorlesung und Übungsbetrieb

Organisatorisches

Übungsaufgaben

- Jedes Übungsblatt enthält Präsenzaufgaben (**P**) für die Tutorien
- Abzugebende **Programmier Hausaufgaben (H)** via Artemis
- Zusätzliche Eigeninitiativ-Aufgaben (**E**): werden nicht abgegeben oder korrigiert
- Alle Aufgaben sind prüfungsrelevant
- Bearbeitung der Übungen ist freiwillig, aber sehr empfehlenswert!
- **Klasurbonus** auf erfolgreich bestandene Klausur / Wiederholungsklausur. Nötig sind 80% der Punkte pro Übungsblatt um 1 Bonuspunkt für die Klausur zu erreichen. Insgesamt sind bis zu 12 Bonuspunkte möglich. Die Klausur hat 90 Punkte.
- Die Hausaufgaben sind **selbstständig** anzufertigen! Nach Plagiaten wird automatisiert und manuell gesucht.

Aufgabe 1.1

Division

Aufgabe 1.1

- Wir betrachten die Schulmethode der Division
(Beschränkung für uns; Dividend: n-stellig, Divisor: 1-stellig, positiv)
- Untersuchen der Anzahl der nötigen „elementaren Operationen“
- Definitionen der Grundoperationen
 - Ganzzahldivision: $g(a, b) = \lfloor a/b \rfloor$
 - Rest: $r(a, b) = a - g(a, b) \cdot b$
 - Konkatenation: $a \oplus b$
 - Zuweisung: $x = y$
 - Vergleich: $x > y$ $x < y$ $x \geq y$ (etc.)
 - Addition: $x + y$

(Indices berechnen zählt nicht als Grundoperation)

Aufgabe 1.1

Input: Ziffer $\llbracket (x_1, x_2, \dots, x_n)$, Ziffer y

```

1 int i := 1    ① → 1 · 1
2 Ziffer  $x_0 := 0$  ① → 1 · 1
3 while  $i \leq n$  do ① →  $(n+1) \cdot 1$ 
4   if  $x_{i-1} > 0$  then ① →  $n \cdot 1$ 
5     Ziffer  $z_i := g(x_{i-1} \oplus x_i, y)$  3 } ⑦ →  $(n-1) \cdot 7$ 
6      $x_i := r(x_{i-1} \oplus x_i, y)$  3
7      $x_{i-1} := 0$ ; 1
8   end
9   else
10    Ziffer  $z_i := g(x_i, y)$  2 } ④ → 1 · 4
11     $x_i := r(x_i, y)$  2
12  end
13   $i := i + 1$  ② →  $n \cdot 2$ 
14 end

```

15// Das Ergebnis der Division ist $z_1 \oplus z_2 \dots$, der Rest is x_n .

- Schleifenbedingung $n+1$ mal ausgeführt
- Schleifenkörper n -mal
- Im 1. Schleifendurchlauf immer else-block
(da $x_0=0$)

- Für Worst-case wird sonst immer der if-block ausgeführt: $(n-1)$ -mal

- Worst-Case-Bsp.: 1111 : 2

$$2 + n+1 + n + 7(n-1) + 4 + 2n$$

$$= \underline{\underline{11n}} \quad \text{Grundoperationen im Worst-case}$$

Wiederholung: Induktionsbeweise

„Die vollständige Induktion ist eine mathematische Beweismethode, nach der eine Aussage für alle natürlichen Zahlen bewiesen wird, die größer oder gleich einem bestimmten Startwert sind.“

https://de.wikipedia.org/wiki/Vollst%C3%A4ndige_Induktion

1. Induktionsanfang / Induktionsbasis:

Zeige: Die Aussage $A(n)$ gilt für $n=n_0$

2. Induktionsannahme / Induktionsvoraussetzung:

Aussage gilt für alle Zahlen kleiner gleich n
(wobei $n \geq n_0$ beliebig gewählt werden kann)

3. Induktionsschritt:

Beweise mithilfe der Annahme \rightarrow Aussage gilt für $n+1$

Aufgabe 1.2

Induktion

Aufgabe 1.2 (a)

Zeigen Sie für alle natürlichen Zahlen $n \geq 1$ mittels Induktion die folgende Behauptung:

Die Summe der ersten n ungeraden Zahlen ist n^2 (als Formel: $\sum_{i=1}^n (2i - 1) = n^2$).

Kennzeichnen bzw. benennen Sie in Ihrem Beweis den Induktionsanfang, die Induktionsvoraussetzung und den Induktionsschritt.

Induktionsanfang: $\sum_{i=1}^1 (2i - 1) = 2 \cdot 1 - 1 = 1 = 1^2$

Induktionsvoraussetzung: Für $n \in \mathbb{N}$ gilt $\sum_{i=1}^n (2i - 1) = n^2$ Bsn.

Induktionsschritt: $\sum_{i=1}^{n+1} (2i - 1) = 2(n+1) - 1 + \underbrace{\sum_{i=1}^n (2i - 1)}_{\text{IV}} \stackrel{\text{IV}}{=} 2n + 1 + n^2 \stackrel{\downarrow}{=} (n+1)^2$

Aufgabe 1.2 (b)

$$-\overline{F_n^2}$$

Zeigen Sie $\overline{F_{n+1} F_{n-1}} = (-1)^n$,

wobei F_n die n-te Fibonaccizahl nach der rekursiven Definition $F_n = F_{n-1} + F_{n-2}$ mit den Anfangswerten $F_0 = 0$ und $F_1 = 1$ ist.

IB: $\overline{F_2 F_0} - \overline{F_1^2} = (-1)^1$ $\overline{F_2} = \overline{F_1} + \overline{F_0} = 1 + 0 = 1$

$$1 \cdot 0 - 1^2 = -1 = (-1)^1$$

IV: $\overline{F_{n+1} F_{n-1}} - \overline{F_n^2} = (-1)^n$ gilt für ein beliebiges aber fixes $n \in \mathbb{N}$

IS: $\overline{F_{n+2} F_n} - \overline{F_{n+1}^2} = (\overline{F_{n+1}} + \overline{F_n}) \overline{F_n} - (\overline{F_n} + \overline{F_{n-1}}) \overline{F_{n+1}}$
 $= \underline{\overline{F_{n+1}} \cdot \overline{F_n}} + \overline{F_n^2} - \underline{\overline{F_{n+1}} \cdot \overline{F_n}} - \overline{F_{n+1}} \cdot \overline{F_{n-1}}$
 $= \overline{F_n^2} - \overline{F_{n+1}} \cdot \overline{F_{n-1}}$
 $= (-1) \underbrace{(\overline{F_{n+1}} \cdot \overline{F_{n-1}} - \overline{F_n^2})}_I^{IV} = (-1) (-1)^n = (-1)^{n+1}$

Aufgabe 1.3

Bubblesort mit Listen

Aufgabe 1.3

```
1  public class List {  
2  
3      private static class Node {  
4          private int data;  
5  
6          public int getData () {  
7              return data;  
8          }  
9  
10         public void setData (int data) {  
11             this.data = data;  
12         }  
13  
14         private Node next;  
15         public Node getNext () {  
16             return next;  
17         }  
18  
19         public Node (int data, Node next) {  
20             this.data = data;  
21             this.next = next;  
22         }  
23     }  
24  
25     private Node head;  
26     private int size;  
27  
28     public int getSize () {  
29         return size;  
30     }  
31  
32     public List () { }  
33  
34     public void prepend (int data) {  
35         head = new Node(data, head);  
36         size++;  
37     }  
38 }
```

```
39     public int get (int index) {  
40         Node it = head;  
41         while (index != 0) {  
42             index--;  
43             it = it.getNext();  
44             if (it == null)  
45                 throw new RuntimeException("Out of bounds");  
46         }  
47         return it.getData();  
48     }  
49  
50     public void swap (int indexFirst, int indexSecond) {  
51         if (head == null) {  
52             throw new RuntimeException("Out of bounds");  
53         }  
54  
55         if (indexFirst > indexSecond) {  
56             swap(indexSecond, indexFirst);  
57             return;  
58         }  
59  
60         int distance = indexSecond - indexFirst;  
61         Node it_first = head;  
62  
63         while (indexFirst != 0) {  
64             indexFirst--;  
65             it_first = it_first.getNext();  
66             if (it_first == null)  
67                 throw new RuntimeException("Out of bounds");  
68         }  
69  
70         Node it_second = it_first;  
71         while (distance != 0) {  
72             distance--;  
73             it_second = it_second.getNext();  
74             if (it_second == null)  
75                 throw new RuntimeException("Out of bounds");  
76         }  
77  
78         int temp = it_second.getData();  
79         it_second.setData(it_first.getData());  
80         it_first.setData(temp);  
81     }  
82 }
```

Aufgabe 1.3 (a) – Implementierung

1. Welche der Methoden sind langsam, welche schnell? Wieso?

- `get(index)` der Liste und `swap` sind langsam → durchlaufen der Liste (im Worst-case bis zum Ende)
- `getter`, `setter`, `prepend` sind schnell → nur schreiben/lesen von einer/einigen Variablen

2. Die swap-Methode ruft sich selbst auf. Wie nennt man Methoden, die sich so verhalten? Welche Motivationen gibt es, derart zu programmieren? Welche Nachteile hat so ein Ansatz?

- Rekursive Methoden
- „Eleganterer“ bzw. lesbarer/intuitiver Code
- Nachteil: Call-Stack für jeden rekursiven Aufruf um 1 höher → potentieller Stack-Overflow bei zu vielen rekursiven Aufrufen

Aufgabe 1.3 (b) – Bubblesort

Die Liste soll nun genutzt werden, um Daten sortiert zu speichern.

Um die Liste zu 4 sortieren, wird folgende Methode verwendet, die den Bubblesort-Algorithmus implementiert:

```
→ void bubblesort(List l) {  
    for(int i = 0; i < l.getSize(); i++)  
        for(int j = 1; j < l.getSize() - i; j++)  
            if(l.get(j - 1) > l.get(j))  
                l.swap(j - 1, j);  
}
```

Aufgabe 1.3 (b) – Bubblesort

1. Die Methode hat keinen Rückgabewert; funktioniert sie dennoch? Wenn ja, wieso?

- Objekte in Java werden durch Referenzen repräsentiert, wobei auch diese Methode eine Referenz auf die Liste erhält
- Somit wird in dieser Methode das Objekt verändert, dessen Referenz übergeben wurde.
- Stichwort: In-Place-Sortierung

2. Wie funktioniert der Algorithmus, warum liefert er stets ein sortiertes Ergebnis?

- Für jeden Durchlauf der inneren Schleife wird das jeweils höchste Element, welches noch nicht an seiner sortierten Position steht, dorthin getauscht (durch eine Kette von swap()-Aufrufen)
- Im ersten Durchlauf das höchste ans Ende der Liste, im zweiten das zweithöchste an die vorletzte Stelle, ...
- Dies wird n-mal ($n \hat{=} \text{Listenlänge}$) wiederholt \Rightarrow Alle Elemente sortiert

Aufgabe 1.3 (b) – Bubblesort

3. Wie oft ungefähr wird eine Liste der Größe n durchlaufen? Berücksichtigen Sie bei Ihrer Antwort nur die Implementierung des Bubblesort-Algorithmus.

- Innere Schleife durchläuft beim ersten Mal die ganze Liste, beim zweiten mal bis zum vorletzten Element, ...
- Im Durchschnitt die halbe Liste pro Durchführung, bei n -Durchläufen

$$\Rightarrow n \cdot \frac{n}{2} = \frac{n^2}{2} \in O(n^2) \text{ bzw. Laufzeit quadratisch in der Länge der Liste}$$

4. Eignet sich unsere Implementierung einer Liste hier besonders gut oder besonders schlecht? Ziehen Sie nun die Implementierung der Listenmethoden in Ihre Laufzeitüberlegung mit ein. Zu welchem Ergebnis kommen Sie?

- Die `get(index)` und `swap(...)` Methoden müssen jedes Mal die Liste von Vorne durchlaufen (!ineffizient!)
- Ein Array mit Indexzugriffen in konstanter Zeit ist deutlich besser dafür geeignet.

Aufgabe 1.3 (c) – Code Qualität

1. Was hat es mit der Schachtelung der Klassen auf sich?

- Node ist eine Hilfsklasse die lediglich für die Implementierung der Liste verwendet wird.
- Der Nutzer soll von dieser Klasse keinen Gebrauch machen → darum private (nicht sichtbar nach außen)

Nützlicher Tipp: Innere Klassen können auf Daten der äußeren Klasse zugreifen.

2. Wozu wurde der Wrapper List implementiert, statt den Benutzer direkt auf Knoten arbeiten zu lassen?
Was hat dies mit abstrakten Datentypen zu tun?

- Die Wrapper-Klasse soll dem Nutzer eine sichere Schnittstelle zur Verwendung der Liste bieten.
Würde man ihn direkt auf den Nodes arbeiten lassen könnte er die Liste zerstören, und später Änderungen
an der Implementierung könnten dann den Code des Nutzers unbrauchbar machen wenn er mit den Nodes selbst arbeitet.

Aufgabe 1.3 (c) – Code Qualität

3. Welchen Zweck erfüllt **throw**? Wieso ist dies besser, als vordefinierte Fehlerwerte zurückzugeben?

- Der Funktionsaufruf wird mit Auslösen einer Exception abgebrochen und das Programm abstürzen wenn diese nicht behoben wird.

(bei Fehlervertrüglichgabe würde das Programm normal weiterlaufen)

- User kann eine gute Fehlermeldung von uns erhalten.

4. Vergleichen Sie die Implementierung der Funktionen **get** und **swap**. Was fällt Ihnen dabei auf? Wann kann das ein Problem sein, und wie lässt sich dieses vermeiden?

- Die while-Schleifen sind jedes Mal identisch → Stichwort: Code-Duplication
- Könnte durch eine private Hilfsfunktion ersetzt werden die die Node an einem bestimmten Index zurückgibt.
 - weniger Code
 - bei Änderungen muss dies nur in der Hilfsmethode gemacht werden, nicht an drei verschiedenen Stellen