Lehrstuhl für Informatik 2
Modellierung und Verifikation von Software

Prof. Dr. Ir. Joost-Pieter Katoen
Sebastian Junges, Benjamin Kaminski, David Korzeniewski, Tim Quatmann

Übung 3

Hinweise:

- Die Lösungen müssen bis **Donnerstag, den 03. Mai um 16:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Die Übungsblätter **müssen** in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung abgegeben werden.
- Drucken Sie ggf. digital angefertigte Lösungen aus. Abgaben z.B. per Email sind nicht zulässig.
- Namen und Matrikelnummer sowie die **Nummer der Übungsgruppe** sind auf jedes Blatt der Abgabe zu schreiben. Abgaben, die aus mehreren Blättern bestehen **müssen geheftet bzw. getackert** werden! Die **Gruppennummer muss sich auf der ersten Seite oben links** befinden.
- **Bei Nichtbeachten der obigen Hinweise müssen Sie mit erheblichen Punktabzügen rechnen!**

Aufgabe 1 (Binäre Bäume): (10 + 5 + 10 = 25 Punkte)

a) Beweisen oder widerlegen Sie die folgende Aussage: Ein Baum der Höhe h enthält höchstens $2^h - 1$ innere Knoten.

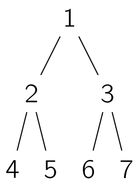
b) Beweisen Sie die folgende Aussage: Wenn nur die Preorder Linearisierung und die Postorder Linearisierung eines Baumes mit eindeutigen Schlüsseln gegeben sind, ist der Baum nicht eindeutig bestimmt.

c) Wir definieren die *Mirror*-Linearisierung eines Baumes als die Ausgabe der folgenden Funktion:

```
Input: die Wurzel node eines (Teil-)Baums
Ausgabe: Die Mirror-Linearisierung des Teilbaums

print(node)
preorder(node.right)
mirror(node.left)
```

Beispielsweise ist 1367254 die Mirror-Linearisierung des folgenden Baumes:



Beweisen oder widerlegen Sie die folgende Aussage: Gegeben die Mirror-Linearisierung und die Inorder-Linearisierung eines Baumes ist der Baum eindeutig bestimmt.

Aufgabe 2 (DAG-Traversierung): (15 Punkte)

Betrachten Sie folgenden Algorithmus zum Traversieren eines DAGs $G = (V, E)$ mit $|V| = n$.

```
Eingabe: DAG G = (V, E) mit eindeutigem Root v_0. \\
Ausgabe: Eine Sequenz mit Knoten v_i ∈ V.

// v.besucht = False fuer alle v ∈ V
traverse(G):
    for each v ∈ V:
        v.besucht = false
    visit(v_0)

visit(v):
    if not v.besucht:
        for each v' ∈ V with (v, v') ∈ E:
            visit(v')
        print(v.wert)
        v.besucht = true
```

Die DAGs die wir hier betrachten haben folgende Eigenschaft (*eindeutiger Root*): Genau der Knoten $v_0 \in V$ hat keine eingehende Kanten. Formal gilt:

- $\forall (v, v') \in E, v' \neq v_0$, und
- $\forall v' \in V \setminus \{v_0\} \; \exists v (v, v') \in E$.

Begründen Sie, dass die Ausgabe von traverse(G) jeden Knoten in G genau einmal ausgibt.

Aufgabe 3 (Abstrakte Datentypen): (5 + 5 + 5 + 5 = 20 Punkte)

a) Wir betrachten den Abstrakten Datentyp (ADT) *double-ended-queue* oder kurz *deque*. Dieser repräsentiert eine Warteschlange (queue), bei der man sich an beiden Enden anstellen kann und auch an beiden Enden ein Element entnehmen kann. Formal hat der ADT folgende Operationen:

- `bool isEmpty(Deque q)` gibt `true` zurück, wenn `q` leer ist, andernfalls `false`.
- `void enqueueFront(Deque q, Element e)` fügt das Element `e` vorne in die Deque `q` ein.
- `void enqueueBack(Deque q, Element e)` fügt das Element `e` hinten in die Deque `q` ein.
- `Element dequeueFront(Dequeu q)` Entfernt das Element am weitesten vorne in der Deque und gibt es zurück; benötigt eine nicht-leere Deque `q`.
- `Element dequeueBack(Dequeu q)` Entfernt das Element am weitesten hinten in der Deque und gibt es zurück; benötigt eine nicht-leere Deque `q`.

Ist es möglich diese Datenstruktur so zu implementieren, dass alle fünf Operationen in Laufzeit $\mathcal{O}(1)$ ausgeführt werden können? Begründen Sie ihre Antwort!

b) Ist es auch möglich, dass alle fünf Operationen des ADT Deque in $\mathcal{O}(1)$ ausgeführt werden können, wenn der ADT nur mit Hilfe *eines* unbeschränkten Arrays und einem Zeiger implementiert werden soll? Ein unbeschränktes Array hat für jedes $i \in \mathbb{N}$ eine Speicherzelle. Begründen Sie ihre Antwort!

c) Wir betrachten den ADT *Set*, der Mengen darstellt. *Set* hat die folgenden Operationen:

- `void add(Set s, Element e)`, fügt das Element `e` zum Set `s` hinzu, falls `e` noch nicht in `s` enthalten ist. Ansonsten bleibt `s` unverändert.
- `bool contains(Set s, Element e)` gibt `true` zurück, falls `e` in `s` enthalten ist, sonst `false`.
- `Set union(Set s1, Set s2)` gibt die Vereinigung von `s1` und `s2` zurück.

Ist die folgende Aussage "Alle drei Operationen des ADT Set benötigen jeweils höchstens $\mathcal{O}(n)$ Zeit, wobei n die Summe der Längen aller Eingabe-Sets ist." korrekt? Begründen Sie ihre Antwort!

d) Gibt es eine Implementierung, sodass die Operation `union` des ADT Set aus dem vorherigen Aufgabenteil nur $\mathcal{O}(1)$ Zeit benötigt? Begründen Sie ihre Antwort!

Aufgabe 4 (Laufzeitanalyse): (4 + 14 + 8 + 14 = 40 Punkte)

Betrachten Sie folgenden Algorithmus:

```
1  Eingabe: Liste l der Laenge n von Zahlen zwischen 1 und k
2  Ausgabe: Gibt es Duplikate von Zahlen in l
3
4  gesehen = false^k
5  for (i in l)
6    if gesehen[i]
7      return true
8    else
9      gesehen[i] = true
10
11 return false
```

a) Was ist die Speicherkomplexität im Best-Case? Was ist die Speicherkomplexität im Worst-Case?

Bitte begründen Sie Ihre Antworten kurz.

b) Bestimmen Sie unter folgenden Annahmen die Best-Case, Worst-Case, und Average-Case Laufzeit.

- `l` ist eine beliebige Liste mit genau den Einträgen 1 bis k, und einer zusätzlichen 1. Die Liste hat also die Länge $n = k + 1$.
- Die Liste ist beliebig geordnet, d.h. jede Reihenfolge ist gleich wahrscheinlich.

Zur Einfachheit nehmen wir an, dass das Prüfen der if-Bedingung in Zeile 6 genau eine Zeiteinheit benötigt und alle weiteren Operationen keine Zeit benötigen. Begründen Sie Ihre Antworten kurz.

Hinweise:

- Das Tauschen der beiden 1en ändert die Reihenfolge der Liste nicht.

c) Geben Sie einen Algorithmus in Pseudo-code an, dessen Eingabe eine Liste mit n Einträgen aus den Zahlen 1 bis k ist und dessen Ausgabe `True` ist, genau dann wenn es ein Duplikat in der Liste gibt. Der Algorithmus soll konstanten Speicherbedarf im Worst-case haben. Analysieren Sie die asymptotische Worst-case Laufzeit von ihrem Algorithmus.

d) Geben Sie jeweils einen Algorithmus für folgendes Problem an.

Eingabe: Liste `l` der Länge n, n gerade, mit beliebigen Einträgen aus 1 bis n (kann Duplikate enthalten).
Ausgabe: Der Modus von `l`, d.h. der Eintrag, der am häufigsten vorkommt.

- Variante 1: Der Algorithmus soll eine lineare asymptotische Laufzeit haben, und Platzbedarf $n \cdot (\log n - 1) + 4 \log n$.
- Variante 2: Der Algorithmus soll (worst-case) Platzbedarf $5 \log n$ haben. Können Sie den Algorithmus auch verbessern, sodass der (worst-case) Platzbedarf $4 \log n$ ist, und wenn ja, wie?

Begründen sie jeden Algorithmus kurz.