Prof. Dr. Ir. Joost-Pieter Katoen

Sebastian Junges, Benjamin Kaminski, David Korzeniewski, Tim Quatmann

Übung 3
– Musterlösung –

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.

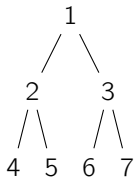
b) Beweisen Sie die folgende Aussage: Wenn nur die Preorder Linearisierung und die Postorder Linearisierung eines Baumes mit eindeutigen Schlüsseln gegeben sind, ist der Baum nicht eindeutig bestimmt.

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.

Lösung:

a) Beweis durch Induktion:

Induktionsanfang: Ein Baum der Höhe 0 enthält $2^0 - 1 = 1 - 1 \geq 0$ innere Knoten.

Induktionsvoraussetzung: Ein Baum der Höhe h enthält höchstens $2^h - 1$ innere Knoten.

Induktionsschritt: Zeige die Aussage für einen Baum der Höhe $h + 1$.

Die Wurzel ist ein innerer Knoten, da $h + 1 > 0$. Die Anzahl der inneren Knoten im linken und rechten Teilbaum ist jeweils nach **IV** durch $2^h - 1$ beschränkt.

Die Anzahl der inneren Knoten ist also beschränkt durch:

$$\begin{aligned} & 2 \cdot (2^h - 1) + 1 \\ =& 2 \cdot 2^h - 2 + 1 \\ =& 2^{h+1} - 1 \end{aligned}$$

Nach dem Prinzip der vollständigen Induktion, folgt die Aussage für alle Bäume.

Alternative Lösung Jeder Baum der Höhe h, lässt sich aus einem vollständigen Baum der Höhe h konstruieren, indem sukzessive Blätter gelöscht werden. Beim Löschen eines Blattes bleibt die Anzahl innerer Knoten gleich, wenn der Elternknoten einen weiteren Nachfolger hat, und sinkt um 1, wenn der Elternknoten keinen weiteren Nachfolger hat. Die Anzahl innerer Knoten in einem Baum der Höhe h ist somit durch die Anzahl innerer Knoten in einem vollständigen Baum der Höhe h beschränkt.

Ein vollständiger Baum der Höhe h hat nach Vorlesung $2^{h+1} - 1$ Knoten. Davon sind 2^h Blätter. Er hat somit $2^{h+1} - 1 - 2^h = 2 \cdot 2^h - 1 - 2^h = 2^h - 1$ innere Knoten.

Somit hat jeder Baum der Höhe h höchstens $2^h - 1$ innere Knoten.

b) Beweis durch Kontraposition. Wir widerlegen die gegenteilige Aussage durch ein Gegenbeispiel.

Betrachte die Preorder Linearisierung $1, 2$ und die Postorder Linearisierung $2, 1$. Diese passen zu beiden folgenden Bäumen:

Somit ist der Baum nicht eindeutig durch die Preorder- und Postorderlinearisierung bestimmt.

c) Beweis durch Induktion:

Induktionsanfang: Ein Baum der Höhe 0 lässt sich eindeutig aus der Mirror-Linearisierung rekonstruieren. Der Knoten, der in beiden Linearisierungen vorhanden ist, stellt die Wurzel dar. Ein Baum der Höhe 0 hat immer nur genau einen Knoten, somit ist der Baum durch die Wurzel eindeutig bestimmt.

Induktionsvoraussetzung: Ein Baum der Höhe n ist eindeutig durch die Mirror-Linearisierung und der Inorder-Linearisierung bestimmt.

Induktionsschritt: Zeige die Aussage für einen Baum der Höhe $n+1$.

Aus der Mirror-Linearisierung lässt sich eindeutig die Wurzel ablesen. Die Wurzel ist der erste Knoten in der Mirror-Linearisierung.

Die Inorder-Linearisierung wird nun in drei Teile l, $node$ und r zerlegt, wobei l der Teil vor der Wurzel ist, $node$ die Wurzel und r der Teil nach der Wurzel. Nach Definition ist l die Inorder-Linearisierung des linken Teilbaums und r die Inorder-Linearisierung des rechten Teilbaums.

Die Mirror-Linearisierung kann in gleicher weise in $node$, r' und l' zerlegt werden. Dabei ist $node$ die Wurzel, r' der Infix, der genau die gleichen Knoten wie r enthält und l' der Postfix, der genau die gleichen Knoten wie l enthält. Nach Definition ist r' die Preorder-Linearisierung des rechten Teilbaums und l' die Mirror-Linearisierung des linken Teilbaums.

Nach Satz aus der Vorlesung ist der rechte Teilbaum durch r und r' eindeutig bestimmt und nach **IV** ist der linke Teilbaum durch l und l' eindeutig bestimmt, da er Höhe n hat.

Durch die Wurzel, den linken und den rechten Teilbaum ist ein Baum eindeutig bestimmt.

Nach dem Prinzip der vollständigen Induktion, folgt die Aussage für alle Bäume.

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

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

```
1  Eingabe: DAG G = (V, E) mit eindeutigem Root v_0. \\
2  Ausgabe: Eine Sequenz mit Knoten v_i ∈ V.
3
4  // v.besucht = False fuer alle v ∈ V
5  traverse(G):
6      for each v ∈ V:
7          v.besucht = false
8      visit(v_0)
9
10 visit(v):
11     if not v.besucht:
12         for each v' ∈ V with (v, v') ∈ E:
13             visit(v')
14         print(v.wert)
15         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.

Lösung:

- *Es existiert ein Pfad von v_0 zu jedem Knoten.*
  - Sei X die Menge von Knoten, zu denen es kein Pfad gibt. Angenommen, X is nicht die Leere Menge.

  - D.h es existiert kein Pfad zu einem Knoten $v_1 \in X$. Dann existiert auch kein Pfad zu den Vorgängern von v_1, d.h, die Vorgänger sind auch in X.
  - Sei Y_i nun die Menge von Zuständen die v_1 in i Schritten erreicht.
  - $\forall i\ Y_i \subseteq X$
  - Da G azyklisch ist, ist $Y_{i+1} \supset Y_i$ oder $v_0 \in Y_i$. v_0 ist aber nicht in X.
  - Also $Y_{i+1} \supset Y_i$. Also ist Y_i eine stetig wachsende Kette. Aber V ist endlich. Widerspruch.

- *Vom Knoten v_0 laufen wir jeden Pfad ab.* Wir besuchen jeweils alle Kinder von einen Knoten, und machen das rekursiv solange wir das von diesem Knoten vorher noch nicht gemacht haben. Entsprechend besuchen wir jeden Knoten mindestens einmal.

- Wenn der Knoten besucht wurde, wird er markiert, vor er (azyklisch!) nochmal besucht wird. Bei einem nächsten Besuch brechen wir ab, ohne den Knoten nochmal auszugeben.

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!

Lösung:

a) Ja. Der ADT kann mithilfe einer doppelt verketteten Liste implementiert werden. Einfügen und löschen am Anfang und Edne dieser Datenstruktur ist in konstanter Zeit möglich, da jeweils nur drei Zeiger geändert werden müssen, die auch in konstanter Zeit gefunden werden können. Prüfen, ob die Datenstruktur leer ist, kann man auch in konstanter Zeit, indem man prüft, ob `head` und `tail` auf `null` zeigen.

b) Nein. Da man nur einen Zeiger zur Verfügung hat, muss entweder der Anfang, oder das Ende der Deque auf einer festen Position im Array liegen, z.B. 0. Der Zeiger verweist dann auf das andere Ende der Deque. Enqueue und Dequeue am Zeigerende kostet konstante Zeit, da nur ein Element geschrieben werden muss und der Zeiger verschoben wird. Enqueue und Dequeue am anderen Ende kostet jedoch lineare Zeit, da alle anderen Elemente verschoben werden müssen.

c) Die Aussage ist falsch. Eine Implementierung kann sich - abgesehen von den Vorgaben - beliebig verhalten. Daher ist es generell nicht möglich eine obere Schranke für alle Implementierungen anzugeben, wie lang eine Operation benötigt.

d) Implementiert man Set mit Hilfe von doppelt verketteten Listen, benötigt `union` nur konstante Zeit. Es genügt die `s1.tail.next` und `s2.head.prev` Zeiger der beiden Listen auf `s2.head` bzw. `s1.tail` zu setzen. Duplikate müssen bei der Operation `union` nicht behandelt werden. Das kann auch an die Implementierung der anderen Operationen delegiert werden.

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.

Lösung:

a) $k + \log k$ Bits für den Best-case und den Worst-case. Der Algorithmus speichert unabhängig von der Eingabe immer k bits in `gesehen`.

b) Beachte: Aufgrund der gegebenen Annahmen ist es für die Einträge nur wichtig, ob der Wert gleich 1 oder ungleich 1 ist. Außerdem terminiert der Algorithmus immer sobald die zweite 1 gefunden wird. Im Best-Case ist der Wert der ersten beiden Einträge 1. Dann überprüfen wir zwei Einträge, d.h.

$$B(n) = 2.$$

Im Worst-Case wird die zweite 1 erst im letzten Schleifendurchlauf gefunden, d.h.

$$W(n) = n.$$

Für den Average-Case nehmen wir zunächst an, dass die zweite 1 an Position $j \in \{2, \ldots, n\}$ steht. In dem Fall beträgt die Laufzeit j. Wir berechnen nun die Wahrscheinlichkeit für diesen Fall. Es gibt $j-1$ mögliche Positionen für die erste 1. Für die restlichen Einträge von 2 bis k gibt es $(n-2)!$ mögliche Anordnungen. Insgesamt gibt es also

$$(j-1) \cdot (n-2)!$$

mögliche Anordnungen für den Fall, dass die zweite 1 an Position j steht. Da es $n!/2$ mögliche Eingaben gibt, beträgt die Wahrscheinlichkeit für diesen Fall

$$(j-1) \cdot (n-2)! \cdot \frac{2}{n!} = \frac{2j-2}{n^2-n}.$$

Insgesamt folgt für die Average-Case Laufzeit

$$A(n) = \sum_{j=2}^{n} j \cdot \frac{2j-2}{n^2-n}.$$

c) Explizites Paarweises vergleichen, Laufzeitkomplexität $\mathcal{O}(n^2)$.

```
for i in 1 .. n:
    for j in i+1 .. n:
        if l[i] = l[j]:
            return true
return false
```

Alternativ: Sortieren (in-place), dann einmal über die Liste laufen und jeweils letztes Element merken. Explizites Paarweises vergleichen, Laufzeitkomplexität $\mathcal{O}(n \log n)$.

d) 1)

```
1  gesehen = 0^n
2  for z in l:
3    if gesehen[z] = n/2 - 1
4      return z
5    else
6      gesehen[z] = gesehen[z] + 1
7
8  max = 0
9  val = 0
10 for i in 1..n:
11     if gesehen[i] > max:
12         max = gesehen[i]
13         val = i
14 return i
```

2) Mit 5 Variablen mit maximal Wert n (also $\log n$ bits)

```
max = 0
val = 0
for i in 1 .. n:
    curr = 0
    for v in l:
```

```
            if v == i:
                curr = curr + 1
        if curr > max:
            max = curr
            val = i
    return val
```

3) Mit 4 Variablen mit maximal Wert n (also $\log n$ bits)

```
max = 0
for i in 1 .. n:
    curr = 0
    for j in 1 .. n:
        if l[j] = i:
            curr = curr + 1
    if curr > max:
        max = curr
for i in 1 .. n:
    curr = 0
    for j in v:
        if v == i:
            curr = curr + 1
    if curr = max:
        return i
```