Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen Stefan Dollase, Ira Fesefeldt, Tim Quatmann, Jip Spel, Tobias Winkler

Übung 3

Hinweise:

- Die Übungsblätter sollen in 3er Gruppen bearbeitet werden.
- Die Bearbeitung der Übungsblätter ist nicht zwingend notwendig um die Klausurzulassung zu erhalten. Es werden zwar Punkte vergeben, diese dienen jedoch nur zur eignen Einschätzung.
- Zur Vorbereitung auf Präsenzübung und Klausur empfehlen wir trotzdem alle Übungsblätter semesterbegleitend zu bearbeiten. Besonders relevant hierfür sind Aufgaben, die mit einem ★ markiert sind.
- Die Lösung des Übungsblattes wird in RWTHmoodle veröffentlicht. Solange die Tutorien nicht stattfinden können, werden außerdem Videos zu jeder Aufgabe angefertigt.
- Wir haben in RWTHmoodle ein Forum eingerichtet, welches als erste Anlaufstelle für **Fragen** dienen soll.
- Relevante Vorlesungen für dieses Übungsblatt: 2, 3

Aufgabe 1 (Zeit- und Platzkomplexität): 4+14=18 Punkte

Betrachten Sie folgenden Algorithmus:

```
1  Eingabe: Liste ℓ der Länge 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
```

Hinweise:

- `false`^{k} steht für ein Boolean Array der Länge k in dem alle Einträge initial `false` sind.
- Wir nehmen in dieser Aufgabe an, dass Arrays 1-basiert indiziert sind, d.h. `gesehen[1]` ist der erste Eintrag und `gesehen[k]` ist der letzte Eintrag in dem Array.

a) Was ist die Speicherkomplexität im Best-Case? Was ist die Speicherkomplexität im Worst-Case? Nehmen Sie dazu an, dass primitive Datentypen konstant viel Platz benötigen und dass Arrays pro Arrayeintrag konstant viel Platz benötigen. Die Platzkomplexität soll angeben, wie viel Speicher zusätzlich zur Eingabeliste ℓ benötigt wird.

b) Bestimmen Sie die Best-Case, Worst-Case, und Average-Case Laufzeit unter der Annahme, dass

- ℓ genau die Einträge 1 bis k sowie eine zusätzliche 1 enthält (ℓ hat also die Länge $n = k + 1$) und
- jede Reihenfolge gleich wahrscheinlich ist.

Zur Einfachheit nehmen wir an, dass das Prüfen der `if`-Bedingung in Zeile 6 genau c Zeiteinheiten benötigt (für eine Konstante $c > 0$) und alle weiteren Operationen keine Zeit benötigen. Begründen Sie Ihre Antworten kurz.

Hinweise:

- Das Tauschen der beiden 1en ändert die Reihenfolge der Liste nicht.
- Ihre Lösung beim Average-case darf Summenzeichen enthalten.

Aufgabe 2 (Warteschlangen): 2+6+5+3=16 Punkte

In der Vorlesung wurde der ADT Warteschlange (queue) sowie folgende Beispielimplementierung auf beschränkten Arrays vorgestellt:

```
1  class Queue {
2      int head, tail, capacity;
3      int[] A;
4
5      Queue(int N) { // Initialisierung mit Arraygröße N > 0:
6          head = 0; tail = 0; capacity = N;
7          A = new int[N];
8      }
9  };
10
11 bool isEmpty(Queue q) {
12     return (q.head == q.tail);
13 }
14
15 void enqueue(Queue q, Element e) {
16     q.A[q.tail] = e;
17     q.tail = (q.tail + 1) mod q.capacity;
18 }
19
20 Element dequeue(Queue q) {
21     Element e = q.A[q.head];
22     q.head = (q.head + 1) mod q.capacity;
23     return e;
24 }
```

a) Wie in der Vorlesung erwähnt werden Überläufe nicht abgefangen. Geben Sie ein kurzes Beispiel an, das zeigt, dass dadurch die Spezifikation des ADTs (streng genommen) nicht erfüllt sind.

b) Wir möchten nun auch Überläufe behandeln können. Erweitern Sie die Klasse `Queue` um eine Methode `void grow(int N)`. Diese Methode soll die Kapazität der Queue soweit erhöhen, dass sie genau `N` Elemente aufnehmen kann. Sie können annehmen, dass `N` immer größer als die aktuelle Kapazität ist. Ihre Implementierung soll eine Worst-case Laufzeit in $O(\mathtt{N})$ haben. Begründen Sie kurz.

c) Erweitern Sie die obige Implementierung, sodass Überläufe korrekt behandelt werden. Nutzen Sie die Methode `grow` aus **b)**. Überlegen Sie sich eine sinnvolle Strategie (Heuristik) mit der die neue Arraygröße ausgewählt wird.

d) Geben Sie eine möglichst gute obere Schranke für die asymptotische Laufzeit des folgenden Programmes in Abhängigkeit von `N` an. Eine intuitive Begründung genügt.

```
1  void foo(int N) {
2      Queue q(1); // Initialisiere mit Kapazität 1
3      for (int i = 0; i < N; ++i) {
4          enqueue(q, i);
5      }
6  }
```

Hinweis: Wenn Sie in **c** eine gute Strategie gewählt haben, ist eine bessere obere Schranke als $O(\mathtt{N}^2)$ möglich.

Aufgabe 3 (Abstrakte Datentypen): 4·4=16 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(Deque q)` entfernt das Element am weitesten vorne in der Deque und gibt es zurück; benötigt eine nicht-leere Deque `q`.
- `Element dequeueBack(Deque 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 $O(1)$ ausgeführt werden können? Begründen Sie ihre Antwort!

b) Ist es auch möglich, dass alle fünf Operationen des ADT Deque in $O(1)$ 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 $O(n)$ 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 $O(1)$ Zeit benötigt? Begründen Sie ihre Antwort!

Aufgabe 4 (Baumdatenstruktur): 5+10+10=25 Punkte

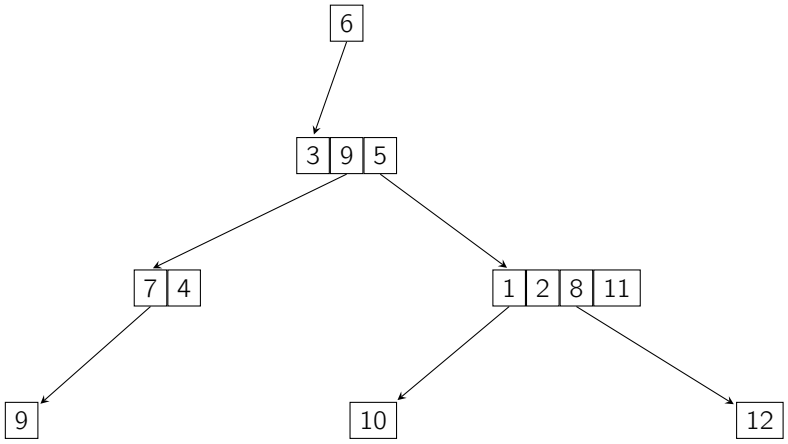
Wir behandeln in dieser Aufgabe Bäume, die nicht zwingend binär sein müssen. Diese Bäume können also beliebig viele Kinder haben. Wir spezifizieren den Baum als Abstrakten Datentyp. Die `Tree` Objekte stellen hierbei die Knoten des Bäumes dar, die `Child` Objekte stellen die Kanten von den Knoten zu ihren Kindern dar. Die Objekte haben folgende Operationen jeweils mit Laufzeit- und Speicherkomplexität von $O(1)$ auf allen Eingaben:

- `Child first(Tree t)` gibt das erste (linkeste) Kind des Knoten `t` zurück, wenn es ein Kind gibt.
- `boolean isLeaf(Tree t)` liefert `true` genau dann, wenn der Knoten `t` keine Kinder hat.
- `int value(Tree t)` gibt den Wert zurück, den der Knoten `t` speichert.
- `int size(Tree t)` gibt die Anzahl der Knoten an, die Nachfahren von `t` (einschließlich `t`) sind.

- `Child next(Child c)` gibt das nächste (rechte) Kind von `c` zurück, wenn `c` nicht das letzte Kind ist.
- `boolean isFirst(Child c)` liefert `true` genau dann, wenn `c` das erste (linkeste) Kind ist.
- `Child previous(Child c)` gibt das vorherige (linke) Kind von `c` zurück, wenn `c` nicht das erste Kind ist.
- `boolean isLast(Child c)` liefert `true` genau dann, wenn `c` das letzte (rechteste) Kind ist.
- `Tree tree(Child c)` liefert den zugehörigen Teilbaum des Kindes `c` zurück.

Bei der Traversierung `levelorder(Tree t)` werden die Werte des Baumes in absteigender Höhe angegeben. Das heißt zuerst werden die Werte aller Knoten auf Ebene 0 angegeben, dann die Werte aller Knoten auf Ebene 1 und so weiter. Die Werte einer Ebene werden nach der Reihenfolge der Kinder sortiert: Zuerst der Wert des ersten (linkesten) Kind, dann der Wert des Kindes rechts davon und so weiter.

a) Geben Sie für den folgenden Baum das Ergebnis einer `levelorder(Tree t)` Traversierung an.



b) Geben Sie ein Algorithmus in Pseudocode an, der die Traversierung `levelorder(Tree t)` für einen Baum `Tree` berechnet. Ihr Algorithmus sollte eine Laufzeit von $O(|V|)$ auf allen Eingaben haben.

c) Begründen Sie, warum Ihr Algorithmus tatsächlich eine Laufzeit von $O(|V|)$ auf allen Eingaben hat und geben Sie weiterhin die Speicherkomplexität ihres Algorithmus in der Groß-O-Notation an.

Hinweise:

- Wir bezeichnen einen Baum als (V, E) wobei V die Menge der Knoten und E die Menge der Kanten ist.
- Sie dürfen in Ihrem Algorithmus auch andere Datenstrukturen aus der Vorlesung nutzen. Geben sie dann aber an, welche Implementierung sie nutzen und welche Laufzeit die Operationen der Datenstrukturen in diesem Fall haben.

Aufgabe 5 (Binärbaum): 10+5+10=25 Punkte

a) Beweisen oder widerlegen Sie die folgende Aussage: Ein Binärbaum der Höhe h enthält höchstens 2^h Blätter.

b) Beweisen oder wiederlegen Sie die folgende Aussage: Die Preorder- und die Postorder-Linearisierung jedes Binärbaumes ist stets unterschiedlich.

c) Begründen Sie: Ist die Preorder- und die Inorder-Linearisierung eines Binärbaumes gegeben, so ist der Baum eindeutig rekonstruierbar. Wir gehen dabei davon aus, dass jeder Wert im Binärbaum nur bei höchstens einem Knoten vorkommt.