ulm university universität
uulm

1. Klausur zur Veranstaltung

Einführung in die Informatik I - Grundlagen

und **Allgemeine Informatik 2**

im Sommersemester 2021

Dr. Jens Kohlmeyer, Stefan Höppner, Dr. Markus Maucher

31.07.2021

Musterlösung

Aufgabe 1 - Verständnisfragen

3 + 4 + 5 = 12 Punkte

a) Erläutern Sie welche **notwendige** Bedingung gelten muss, damit ein gerichteter Graph topologisch sortierbar ist und beschreiben Sie, warum die Nicht-Erfüllung den Graphen topologisch un-sortierbar macht.

Lösung

Der Graph darf keine gerichteten Zyklen enthalten. Im Falle eines Zyklus ist es nicht möglich die Knoten die Teil des Zyklus sind in eine eindeutige Reihenfolge zu bringen.

b) Erläutern Sie warum die Funktionsweise des *Algorithmus von Dijkstra* garantiert, dass die kürzesten Pfade zu allen erreichbaren Knoten gefunden werden.

Lösung

Die Grundidee des Algorithmus ist es, immer derjenigen Kante zu folgen, die den kürzesten Streckenabschnitt vom Startknoten aus verspricht. Andere Kanten werden erst dann verfolgt, wenn alle kürzeren Streckenabschnitte (auch über andere Knoten hinaus) beachtet wurden. Dieses Vorgehen gewährleistet, dass bei Erreichen eines Knotens kein kürzerer Pfad zu ihm existieren kann.

c) Erläutern Sie die Begriffe *Black-Box-Test* und *White-Box-Test*. Gehen Sie dabei auf die Unterschiede zwischen den Tests und die unterschiedlichen Abdeckungskriterien ein.

Lösung

Black-Box-Tests: sind Funktionstests bei denen auf die Ausgabe abhängig von angegebenen Eingaben getestet wird. Das Ziel des Tests ist es ob Eingaben das erwartete/gewünschte Ergebnis produzieren. Hauptaugenmerk legt man bei dieser Art von Tests auf die Funktions- und Eingabe- und Ausgabeüberdeckung da diese alle Merkmale darstellen die nur anhand der Funktionsbeschreibung eindeutig definierbar sind.
Glass-Box-Tests: sind Tests die bei der Erstellung die Ablaufgraphen im Programm berücksichtigen. Dies ermöglicht es das Verhalten in allen potentiell verfolgbaren Pfaden zu prüfen. Hierbei wird darauf geachtet, dass mit den Tests alle Anweisungen mindestens einmal ausgeführt werden (Anweisüberdeckung), dass jeder terminierende Pfad einmal komplett abgelaufen wird (Pfadüberdeckung), dass jeder Term einmal true und einmal false evaluiert (Termüberdeckung) und, dass jede Verzweigung im Ablaufgraph in beide Richtungen einmal abgelaufen wird (Zweigüberdeckung). Diese Art des Testen ermöglicht sicher zu stellen, dass der Prüfling sich in, nach Möglichkeit, allen Ablaufsituationen verlässlich verhält.

Aufgabe 2 - Suchen

3 + 2 + 7 = 12 Punkte

a) Welche der folgenden Aussagen zu den Algorithmen für Mustervergleiche treffen zu?
Hinweis: Mehrere Antworten können richtig sein.

Lösung

- Es gibt Fälle in denen der BF-Algorithmus weniger Vergleiche benötigt als der BM-Algorithmus.
- Es gibt Fälle in denen der BM-Algorithmus weniger Vergleiche benötigt als der KMP-Algorithmus.
- Der KMP-Algorithmus benötigt im *worst-case* O(N+M) Vergleiche.
- Der BM-Algorithmus benötigt im *average-case* O(N/M) Vergleiche.
- Die Randtabelle im KMP-Algorithmus für ein Wort der Länge 6 hat insgesamt 7 Einträge.

b) Erstellen Sie die delta1-Tabelle, welche für den BM-Algorithmus benötigt wird, für das Pattern *banana*.

Lösung

Zeichen	Letztes Vorkommen
b	0
a	5
n	4

c) Suchen Sie mit dem Algorithmus von Boyer-Moore (BM) im unteren Text nach dem **ersten Vorkommen** des Patterns *banana* aus Teilaufgabe **b)**. Füllen Sie für jedes Verschieben des Musters eine Zeile der nachstehenden Tabelle aus. Schreiben Sie in jede Zeile das gesamte Muster und **umkringeln** Sie alle Zeichen des Musters, die mit dem Text verglichen werden. Geben Sie für jeden Shift an, wie Sie diesen bestimmt haben. Notieren Sie dies **hinter** dem Muster auf dem Sie den Shift berechnet haben.
Hinweis: Sie benötigen nicht alle Zeilen der Tabelle.

Lösung

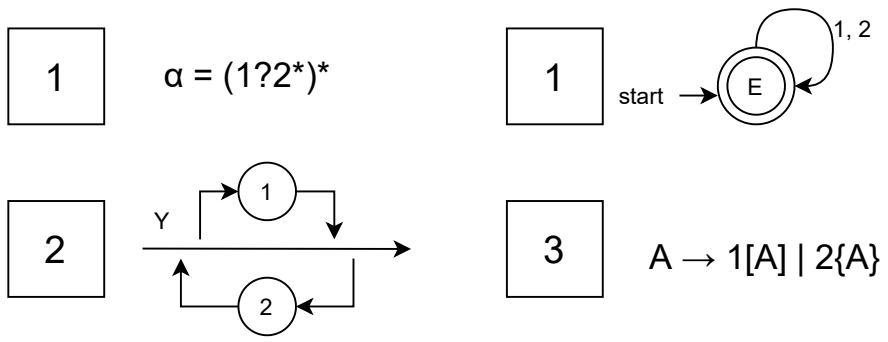
b	a	n	b	a	n	a	n	a	s	e	l
b	a	n	a	n	ⓐ	*Shift* $= 5 - 4 = 1$					
	b	a	ⓝ	ⓐ	ⓝ	ⓐ	*Shift* $= 2 - 0 = 2$				
			ⓑ	ⓐ	ⓝ	ⓐ	ⓝ	ⓐ			

Aufgabe 3 - Formale Sprachen

4 + 4 + 3 + 1 = 12 Punkte

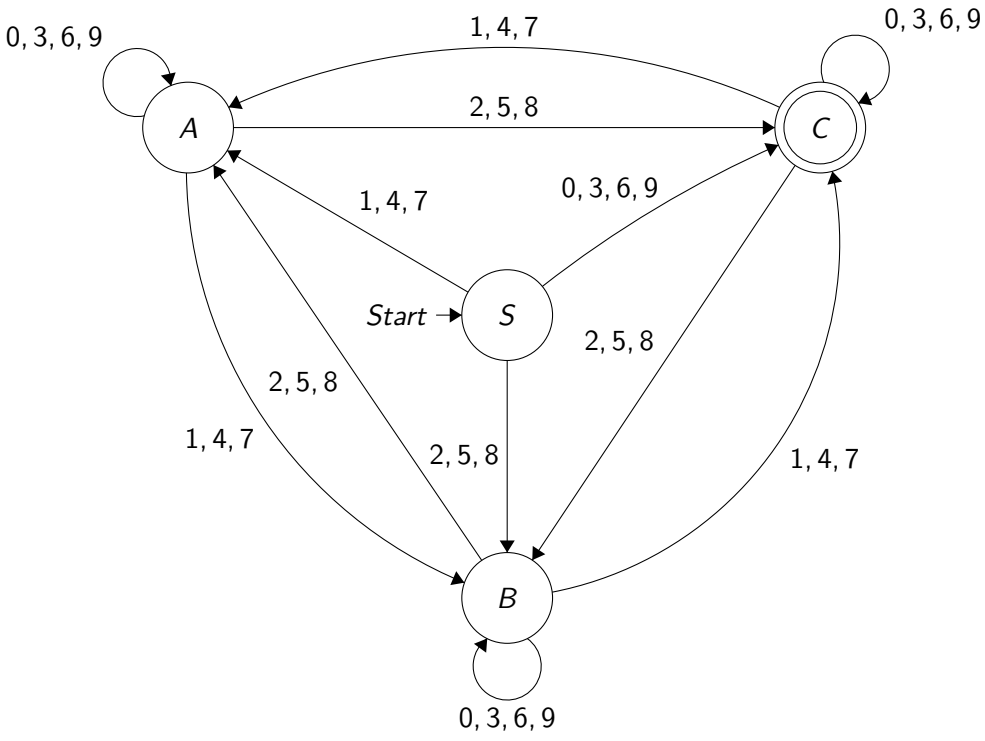
a) Im Folgenden sehen Sie einen regulären Ausdruck, einen endlichen Automaten, ein Syntaxdiagramm und eine EBNF. Markieren Sie, welche davon jeweils die gleiche Sprache erzeugen bzw. beschreiben, indem Sie in die Box eine Zahl eintragen. *Wären Sie zum Beispiel der Meinung, dass alle unterschiedliche Sprachen beschreiben bzw. erzeugen, dann tragen Sie in jede Box eine andere Zahl ein.*

Lösung



b) Definieren Sie einen **deterministischen** endlichen Automaten der alle durch 3 teilbaren Zahlen akzeptiert.
Hinweis: Eine Zahl ist genau dann durch 3 teilbar, wenn ihre Quersumme durch 3 teilbar ist. 0 ist durch 3 teilbar!

Lösung



c) Geben Sie eine Grammatik an, welche eine beliebige Menge von *öffnenden* Klammern, gefolgt von der **doppelten** Menge an *schließenden* Klammern produziert.

Lösung

$$
\begin{aligned}
V &= \{Start\} \\
\Sigma &= \{(,)\} \\
P &= \{Start \rightarrow (Start)), \\
&\quad Start \rightarrow \varepsilon\} \\
S &= Start
\end{aligned}
$$

d) Ist die Sprache die von der Grammatik aus Aufgabe **c)** generiert wird vom Typ 3? Begründen Sie.

Lösung

Nein. Die Anzahl der schließenden Klammern hängt von der Anzahl der öffnenden Klammern abhängig, was mindestens mit einer kontext-sensitiven Grammatik beschrieben werden muss.

Aufgabe 4 - Java - Datenstrukturen

4 + 3 + 5 = 12 Punkte

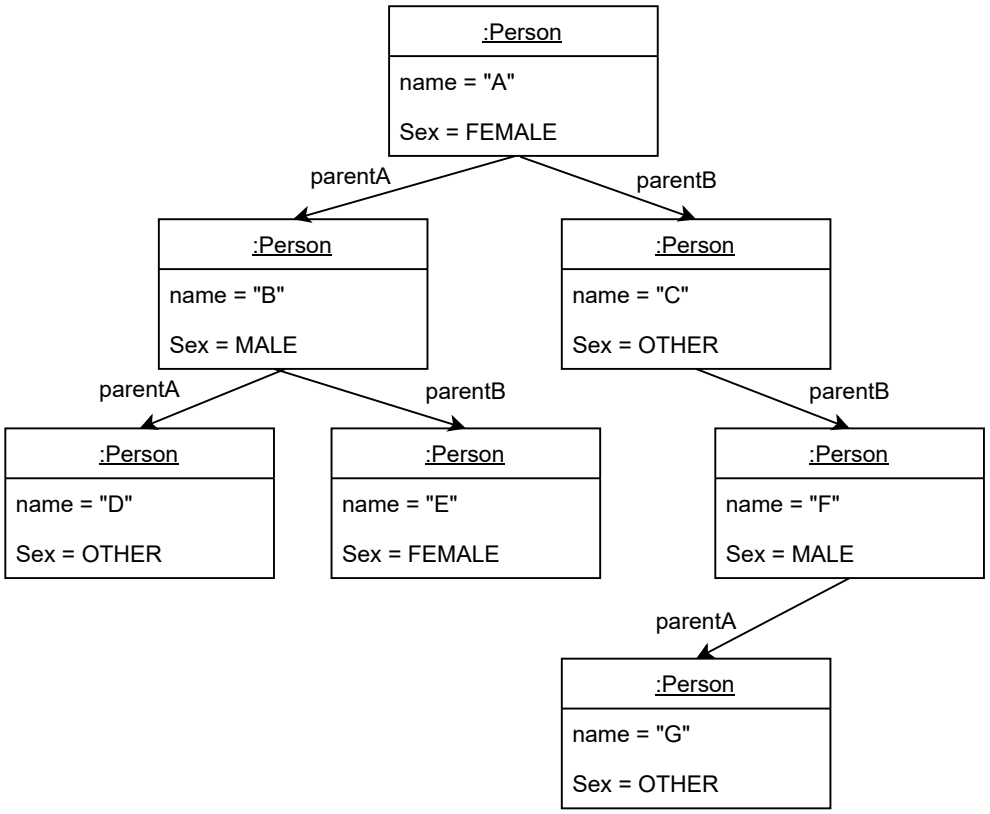
Betrachten Sie folgende Klassen:

```java
public class Person {
    String name;
    Sex sex;
    Person parentA;
    Person parentB;

    public Person(String name, Sex sex) {
        this.name = name;
        this.sex = sex;
    }

    public static Person aFamily() {
        Person a = new Person("A", Sex.FEMALE);
        Person b = new Person("B", Sex.MALE);
        Person c = new Person("C", Sex.OTHER);
        Person d = new Person("D", Sex.OTHER);
        Person e = new Person("E", Sex.FEMALE);
        Person f = new Person("F", Sex.MALE);
        Person g = new Person("G", Sex.OTHER);
        a.parentA = b;
        b.parentA = d;
        f.parentA = g;
        a.parentB = c;
        b.parentB = e;
        c.parentB = f;
        System.out.println(a.noMissingParents());
        return a;
    }
}
```

```java
public enum Sex {
    MALE, FEMALE, OTHER;
}
```

a) Zeichnen Sie das Objekt, welches von der Methode `aFamily()` zurückgeliefert wird, so wie alle Objekte die mit diesem direkt oder indirekt verbunden sind, in UML Objektdiagramm Notation. Schreiben Sie hierbei auch den Attributnamen einer Referenz auf den zugehörigen gerichteten Pfeil zwischen Objekten.

Lösung



b) Implementieren Sie die Methode `noMissingParents()` der Klasse `Person` welche überprüft, ob in der gesamten Familie der Person entweder beide Elternteile existieren oder beide Elternteile unbekannt (`null`) sind und in diesem Fall `true` zurück gibt. Sonst soll `false` zurück gegeben werden.

Lösung

```
1  public boolean noMissingParents() {
2      boolean bothParentsExist =
3        this.parentA != null && this.parentB != null;
4      boolean noParentsExist =
5        this.parentA == null && this.parentB == null;
6      return noParentsExist
7       || bothParentsExist
8          && this.parentA.noMissingParents()
9          && this.parentB.noMissingParents();
10 }
```

c) Implementieren Sie die Methode `getAncestors(int maxGenerations)` der Klasse `Person`. Die Methode soll eine unsortierte Liste aller Vorfahren der Person zurückliefern. Dabei soll maximal `maxGenerations` weit zurückgeblickt werden. *Ein Wert von 1 für `maxGenerations` würde z.B bedeuten, dass nur die Eltern berücksichtigt werden, während ein Wert von 3 alle Vorfahren bis*

einschließlich der Ur-Großeltern zurückliefern würde. Beachten Sie außerdem, dass die Person selbst **nicht** in der Liste ihrer Vorfahren enthalten sein darf!
Hinweis: Sie dürfen die Methode `addAll(List<T> list)` des Interfaces `List` verwenden. Diese fügt alle Elemente der übergebenen Liste an die Liste an, auf der die Methode aufgerufen wurde.
Hinweis: Direkte Vorfahren heißt hier, dass nur die Elternrelationen zwischen Objekten berücksichtigt werden sollen.

Lösung

```
1  public List<Person> getAncestors(int maxGenerations) {
2      return this.ancestors(maxGenerations, 0);
3  }
4  private List<Person> ancestors(int maxGenerations, int current) {
5      List<Person> ancestors = new LinkedList<Person>();

6      if (current <= maxGenerations) {
7          if (current != 0) {
8              ancestors.add(this);
9          }
10         if (this.parentA != null) {
11             ancestors.addAll(
12               this.parentA.ancestors(maxGenerations, current + 1));
13         }
14         if (this.parentB != null) {
15             ancestors.addAll(
16               this.parentB.ancestors(maxGenerations, current + 1));
17         }
18     }

19     return ancestors;
20 }
```

Aufgabe 5 - Java - Queues

1.5 + 2.5 + 3 + 5 = 12 Punkte

Betrachten Sie folgendes Interface und folgende Klassen:

```java
public interface CapacityLimitedQueue<T> {
    boolean enqueue(T element);
    T dequeue();
    int getCapacity();
}
```

```java
public class CapacityLimitedArrayQueue<T>
  implements CapacityLimitedQueue<T> {
    Object[] queue;
    int currentSize = 0;

    public CapacityLimitedArrayQueue(int size) {
        queue = new Object[size];
    }
    //...
    @Override
    public int getCapacity() {
        return queue.length;
    }
}
```

```java
public class QueueUtil {
    //...
}
```

a) Implementieren Sie die Methode `enqueue(T element)` der Klasse `CapacityLimitedArrayQueue` welche das übergebene Element der Queue anfügt insofern diese noch freie Plätze hat und in diesem Fall `true` zurückliefert. Sollte die Kapazität der Queue bereits gefüllt sein, soll das Element verworfen und `false` zurückgeliefert werden.
Beispiele:
Queues haben hier in den Beispielen eine Capacity von 3.
$[1 \leftarrow 2]$, $3 \Rightarrow$ true; $[1 \leftarrow 2 \leftarrow 3]$
$[1 \leftarrow 2 \leftarrow 3]$, $9 \Rightarrow$ false; $[1 \leftarrow 2 \leftarrow 3]$

— **Lösung**

```java
public boolean enqueue(T element) {
    if (currentSize >= queue.length) {
        return false;
    } else {
        queue[currentSize++] = element;
        return true;
    }
}
```

b) Implementieren Sie die Methode `dequeue()` der Klasse `CapacityLimitedArrayQueue` welche den Wert des vordersten Elements zurückgibt und das Element aus der Queue löscht.
Beispiele:
$[1 \leftarrow 2 \leftarrow 3] \Rightarrow 1; [2 \leftarrow 3]$
$[2 \leftarrow 3 \leftarrow 1] \Rightarrow 2; [3 \leftarrow 1]$

Lösung

```java
public T dequeue() {
    T element = (T) queue[0];
    for (int i = 1; i < queue.length; i++) {
        queue[i-1] = queue[i];
        queue[i] = null;
    }
    currentSize = Math.max(currentSize-1, 0);
    return element;
}
```

c) Implementieren Sie die Methode `reverse` der Klasse `QueueUtil` welche die Reihenfolge der Elemente der übergebenen Queue umdreht.
Beispiele:
$[1 \leftarrow 2 \leftarrow 3] \Rightarrow [3 \leftarrow 2 \leftarrow 1]$
$[1 \leftarrow 2 \leftarrow 3 \leftarrow 1] \Rightarrow [1 \leftarrow 3 \leftarrow 2 \leftarrow 1]$

Lösung

```java
public static <T> void reverse(CapacityLimitedQueue<T> queue) {
            Object[] reverser = new Object[queue.getCapacity()];
            for (int i = reverser.length-1; i >= 0; i--) {
                reverser[i] = queue.dequeue();
            }
            for (int i = 0; i < reverser.length; i++) {
                                if (reverser[i] != null) {
                        queue.enqueue((T) reverser[i]);
                                }
            }
        }
```

d) Implementieren Sie die Methode `reprioritize` der Klasse `QueueUtil` welche die Reihenfolge der Elemente in der übergebenen Queue den Prioritäten aus der übergebenen `HashMap` anpasst. Beachten Sie dabei, dass die vergebenen Prioritäten im Intervall [0,127] liegen und niemals doppelt vergeben werden. *Die Methode ist somit nur für Queues mit einer Kapazität von <129 anwendbar.*
Beispiele:
queue: $[1 \leftarrow 2 \leftarrow 3]$, *priority*: $[(1 \rightarrow 10), (2 \rightarrow 127), (3 \rightarrow 100)] \Rightarrow [2 \leftarrow 3 \leftarrow 1]$
queue: $[1 \leftarrow 9]$, *priority*: $[(1 \rightarrow 10), (9 \rightarrow 22)] \Rightarrow [9 \leftarrow 1]$

Lösung

```
1  public static <T> void reprioritize(CapacityLimitedQueue<T> queue,
2           HashMap<T,Integer> priority) {
3              Object[] prioritizer = new Object[128];
4              for (int i = 0; i < queue.getCapacity(); i++) {
5                  T element = queue.dequeue();
6                                  if (element != null) {
7                          prioritizer[priority.get(element)] = element;
8                                  }
9              }
10             for (int i = 127; i >= 0; i--) {
11                 if (prioritizer[i] != null) {
12                     queue.enqueue((T) prioritizer[i]);
13                 }
14             }
15         }
```