Universität Regensburg

PD Dr. J. Reischer 17.2.2016

Wiederholungsklausur "ADP" WS 2015/2016

Nachname, Vorname	
Abschluss (BA, MA, FKN etc.)	
Matrikelnummer, Semester	
Versuch (1/2/3)	

Bitte füllen Sie zuerst den Kopf des Angabenblattes aus!

Die Klausur dauert 90 Minuten.
Es sind maximal 30 Punkte zu erreichen.
Es sind keine Hilfsmittel zugelassen.

Bitte beantworten Sie alle Fragen direkt auf das Angabenblatt.
Nutzen Sie ggf. die Rückseite und kennzeichnen Sie dies entsprechend.
Eigene Schmierblätter sind nicht erlaubt.

Bei mehreren oder mehrdeutigen Lösungen wird die schlechtere Lösung gewertet. Streichen Sie daher ungültige Lösungen eindeutig durch.

Verwenden Sie nur Java, C# oder Pseudokode für Programmieraufgaben.

Viel Erfolg!

Aufgabe 1: *Multiple Choice (diverse Themen)* (4 Punkte)

Korrekte Aussagen sind anzukreuzen. Die Anzahl korrekter Aussagen stimmt nicht notwendigerweise mit der Anzahl vergebener Punkte überein, d. h. eine wahre Aussage kann auch mehr oder weniger als 1 Punkt ergeben. Falsch markierte Aussagen führen zu entsprechendem Punkteabzug; es können jedoch insgesamt nicht weniger als 0 Punkte für Aufgabe 1 erzielt werden. Fehlende Kreuze werden weder positiv noch negativ gewertet. Setzen Sie ein deutliches Kreuz zur Kennzeichnung der Korrektheit einer Aussage; zur Ungültigmachung einer Markierung füllen Sie das Quadrat komplett aus.

☐ Iterative Datenstrukturen wie Listen oder Arrays können nur mit iterativen Algorithmen durchlaufen werden.

☐ Zwei Graphen sind struktur- und inhaltsgleich, wenn ihre Anker-Referenzen identisch sind.

☐ In einer Programmiersprache eines beliebigen Paradigmas muss es immer mindestens einen Referenzdatentyp geben.

☐ Die Elemente eines Arrays können nur Wertetypen enthalten, keine Referenztypen.

☐ In Objekttypen muss mindestens ein Attribut (Datenfeld) zur Datenspeicherung definiert sein.

☐ Zur Vermeidung von Endlosrekursion ist eine Termination der Rekursion notwendig.

☐ Alle Programmiersprachen benötigen Iteration *und* Rekursion als grundlegendes Konzept.

☐ **`for`** ist konzeptionell überflüssig, da es vollständig durch **`while`** ersetzt werden kann.

Aufgabe 2: *Iteration Synthese* (6 Punkte)

Schreiben Sie eine Funktion `int[] CreateDigitIndex(string[] S)`, die aus allen Strings eines beliebig langen Stringarrays `S` einen Ziffernindex erzeugt: Hierzu werden die Ziffernzeichen `'0'` bis `'9'` (und nur diese) ermittelt und deren *Anzahl* in einem Ziffernarray gezählt, das dann zurückgegeben wird.

Beispiel: Stringarray `S = {"ab123xy", "98765", "1p2q3r4s5", "_0815_"}`
Ergebnisarray = `{1, 3, 2, 2, 1, 3, 1, 1, 2, 1}`
(d. h. die `'0'` kommt 1 Mal vor, die `'1'` 3 Mal, die `'2'` 2 Mal usw.)

Hinweis: Der Unicode-Zeichenkode von `'0'` ist kompatibel mit `int` und kann durch eine entsprechende Typkonversion ermittelt werden. Alle höheren Ziffern schließen sich aufsteigend an `'0'` an.

Aufgabe 3: *Iteration Synthese* (10 Punkte)

Zwei gleich große 1D-Arrays beliebiger Länge mit dem Objekttyp `Triple` als Elemente (s. Def. u.) sollen auf Inhaltsgleichheit verglichen werden. Hierzu sind zwei Funktionen zu realisieren: Die erste wird als Methode **`bool`** `CompareTriple(Triple T)` innerhalb des Objekttyps `Triple` angelegt, die zweite **`bool`** `CompareTripleArray(Triple[] T1, Triple[] T2)` außerhalb (wo, sei hier egal).

```
// Definition Triple:
class Triple
{     // Datenfelder für beliebige 3 Ganzzahlen
      public int X; public int Y; public int Z;
      // Konstruktor
      public Triple(int _X, int _Y, int _Z)
      { X = _X; Y = _Y; Z = _Z; }
      // Vergleich einer übergebenen Tripelinstanz T mit dieser Intanz
      public bool CompareTriple(Triple T)
      { // Diese Funktion ist von Ihnen zu implementieren! }
}
```

Aufgabe 3.1: (4 Punkte)

Kode für **`bool`** `CompareTriple(Triple T)` von oben (Fortsetzung auf nä. Seite):

Aufgabe 3.2: Kode für **bool** `CompareTripleArray(Triple[] T1, Triple[] T2)`:
(6 Punkte)

Aufgabe 4: *Rekursion Analyse* (5 Punkte)

Gegeben ist folgende rekursive Funktion F mit positivem Ganzzahl-Parameter N:

```
// Rekursive Funktion F
int F(int N)
{
      switch (N)
      {
            case 0: return N+1;
            case 1: return N+2;
            case 2: return F(N-1);
            case 3: return F(N-1);
            default: return F(N-1)+F(N-2);
      }
}
```

4.1: Welche Ergebnisse liefert die Funktion F für die Aufrufe N = 0 bis 5 zurück? Tragen Sie die Werte für F in unten stehende Tabelle ein (2 Versuche). (3 Punkte)

N	0	1	2	3	4	5
F(N)						
F(N)						

4.2: Welchen Komplexitätsgrad bezüglich des Laufzeitverhaltens O(?) weist die Funktion F im *schlechtesten Fall* auf (Begründung!)? (1 Punkt)

4.3: Welche Formen der Rekursion treten hier auf? Nennen Sie zwei. (1 Punkt)

Aufgabe 5: *Rekursion Synthese* (5 Punkte)

Schreiben Sie eine Funktion **`string`** `MixStrings(`**`string`** `S1, `**`string`** `S2)`, die *rekursiv* die Zeichen zweier *gleich langer* Strings abwechselnd vermischt, beginnend mit `S1`.

Beispiel:

`S1 = "X", S2 = "Y"`: Ergebnis `"XY"`
`S1 = "PQ", S2 = "$%"`: Ergebnis `"P$Q%"`
`S1 = "ABC", S2 = "123"`: Ergebnis `"A1B2C3"`

Hinweis: Sie dürfen folgende Funktionen zur Ermittlung eines Reststrings `R` aus einem String `S` ab einem best. `ZeichenIndex` (bis zum Stringende von `S`) bzw. eines Einzelzeichens `C` an einem best. `ZeichenIndex` aus `S` verwenden:

```
C#:     R = S.Substring(ZeichenIndex);
        C = S[ZeichenIndex];
Java:   R = S.substring(ZeichenIndex);
        C = S.charAt(ZeichenIndex);
```