Universität Regensburg

PD Dr. J. Reischer 20.7.2015

Klausur "ADP" SS 2015

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)

Hinweis: Die Anzahl korrekter Antworten stimmt nicht notwendigerweise mit der Anzahl vergebener Punkte überein, d. h. eine Antwort kann auch mehr oder weniger als 1 Punkt ergeben. Falsche Antworten führen zu entsprechendem Punkteabzug; es können jedoch insgesamt nicht weniger als 0 Punkte für Aufgabe 1 erzielt werden. Fehlende Antworten werden weder positiv noch negativ gewertet. Setzen Sie ein deutliches Kreuz zur Kennzeichnung einer Antwort; zur Ungültigmachung einer Antwort füllen Sie das Quadrat komplett aus.

☐ Rekursive Datenstrukturen wie Bäume können nur mit rekursiven Algorithmen durchlaufen werden.

☐ Zur Erzeugung einer voll- und eigenständigen Kopie eines Graphen muss nur dessen `AnchNode` in einen neuen Ankerknoten kopiert werden.

☐ Objekte müssen programmiersprach-technisch grundsätzlich als Referenztypen realisiert werden.

☐ Jedes Element eines ausgefransten Arrays kann auch ein Graphenknoten sein.

☐ Abgeleitete Typen benötigen mindestens ein zusätzliches Attribut gegenüber dem Basistyp.

☐ Eine Initialrekursion führt niemals zu einer Endlosrekursion.

☐ Alle Programmiersprachen benötigen Iteration *oder* Rekursion.

☐ **`switch`** ist konzeptionell überflüssig, da es vollständig durch **`if`** ersetzt werden kann.

Aufgabe 2: *Iteration Synthese* (10 Punkte)

Es sollen ein-, zwei- und dreidimensionale *ausgefranste* Arrays aus Zeichen erzeugt und initialisiert werden, die in jeder Dimension N Elemente enthalten:

1D: N

2D: N × N

3D: N × N × N

Entsprechend sind drei Funktionen zu implementieren, die jeweils ein 1-, 2- bzw. 3-dimensionales Array als Ergebnis zurückgeben, wobei die einzelnen Array-Werte explizit mit einem übergebenen Zeichen vorzubelegen sind. *Selbstdefinierte Funktionen dürfen in den jeweiligen Aufgaben wiederverwendet werden*!

Hinweis: Die drei Aufgaben können auch unabhängig voneinander bearbeitet werden. Hierzu setzen Sie einfach einen vorgegebenen Funktionskopf ein, wenn Sie ihn benötigen, auch wenn Sie ihn nicht ausprogrammiert haben!

2.1: Schreiben Sie eine Funktion `char[] CreateArray1D(int N, char C)`, die ein 1D-Char-Array der Größe `N` erzeugt und jedes Element explizit mit dem Zeichen `C` vorbelegt. (3 Punkte)

2.2: Schreiben Sie eine Funktion `char[][] CreateArray2D(int N, char C)`, die ein 2D-Char-Array der Größe `N × N` erzeugt und jedes Element explizit mit dem Zeichen `C` vorbelegt. (3 Punkte)

2.3: Schreiben Sie eine Funktion `char[][][] CreateArray3D(int N, char C)`, die ein 3D-Char-Array der Größe `N × N × N` erzeugt und jedes Element explizit mit dem Zeichen `C` vorbelegt. (4 Punkte)

Aufgabe 3: *Iteration Synthese* (6 Punkte)

Schreiben Sie eine Funktion `int Sum(int[] X1, int[] X2)`, die alle Werte zweier *beliebig* langer **int**-Arrays X1 und X2 zusammenaddiert und als Funktionsergebnis zurückgibt (d. h. die Arrays müssen nicht gleich lang sein!). Achten Sie auf bestimmte Sonderfälle von X1 und X2! Gelingt die Addition nicht, soll 0 als Ergebnissumme zurückgegeben werden.

Beispiele:

{1,2,3} (= X1) aufsummiert mit {1,2} (= X2) ergibt 9.

{1,2,3} (= X1) aufsummiert mit {−1,−2,−3,−4} (= X2) ergibt −4.

Aufgabe 4: *Rekursion Analyse* (5 Punkte)

Gegeben sind folgende rekursive Funktionen F1 und F2 mit Ganzzahl-Parametern:

```
// Rekursive Funktionen F1 und F2

int F1(int N1)
{
      if (N1 > 0)
            return F2(N1 – 1);
      else
            return N1 + 1;
}

int F2(int N2)
{
      if (N2 % 2 != 0)
            return F1(N2 – 1);
      else
            return 1;
}
```

4.1: Welche Werte liefert die Funktion F1 für die Aufrufe N1 = 0 bis 5 zurück? Tragen Sie die Ergebnisse für F1 in unten stehende Tabelle ein (die untere Zeile für F2 dient nur für Ihre Zwischenwerte und muss nicht ausgefüllt werden). (3 Punkte)

N1	0	1	2	3	4	5
F1(N1)						
F2(N2)						

4.2: Welchen Komplexitätsgrad bezüglich des Laufzeitverhaltens O(?) weist die Funktion F1 im *schlechtesten Fall* auf (Begründung!)? (1 Punkt)

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

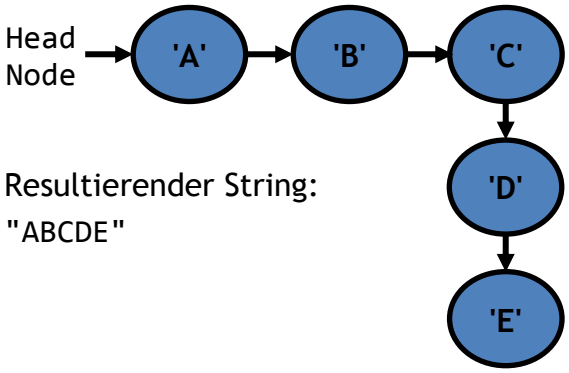
Aufgabe 5: *Rekursion Synthese* (5 Punkte)

Schreiben Sie eine Funktion **`string`** `GetString(ListNode HeadNode)`, die *rekursiv* den in den einzelnen Knoten ab `HeadNode` enthaltenen String einer unidirektionalen Liste zusammenbaut und zurückgibt. In der Lösung darf *keine Schleife* vorkommen. Beachten Sie die Grenzfälle.

Definition von `ListNode`:

```
class ListNode
{
  // Datenfelder
  public char Cont;
  public ListNode Next;
  // Konstruktor
  public ListNode(char InitCont)
  {
    Cont = InitCont;
    Next = null;
  }
}
```

Beispiel:



Resultierender String:

`"ABCDE"`