Aufgabenblatt zur 7. Übung

Zeitraum: 29.11. bis 03.12.2010

1. Aufgabe: (AGS 4.11*)

Gegeben sei das folgende *C*-Programm:

```
 1  #include <stdio.h>
 2
 3  void f (int *a, int *b);
 4
 5  void g (int *x, int *y) {
 6    int z;
 7    z = *y;             /*label1*/
 8    if (z > 0)
 9      f(&z, y); /* $1 */
10    else
11      *x = z;
12    /*label2*/
13  }
14
15  void f (int *a, int *b) {
16    *b = *a - 1;        /*label3*/
17    while (*a > 1) {
18      g(a, b); /* $2 */ /*label4*/
19    }
20  }
21
22  int main ()
23  {
24    int e, a;
25    scanf("%i", &e);    /*label5*/
26    f(&e, &a); /* $3 */ /*label6*/
27    printf("%d", a);
28    return 0;
29  }
```

(a) Geben Sie den Gültigkeitsbereich jedes Objektes des Programms an.

(b) Stellen Sie eine Rechnung des Programms für die Eingabe `e = 2` als pulsierenden Speicher dar, wobei die aktuelle Situation bei jedem Passieren der Marken `label1` bis `label6` gezeigt werden soll. Dokumentieren Sie hierzu jeweils alle sichtbaren Variablen mit ihren Wertebelegungen. Hat eine Variable zu einem Zeitpunkt noch keinen Wert erhalten, so geben Sie anstelle des Wertes ein ? an (also z. B. x = ?, wenn x zum Zeitpunkt der Protokollierung noch keinen Wert besitzt). Führen Sie des Weiteren auch ein Rücksprungmarkenprotokoll.

Beachten Sie: `$1` bis `$3` seien die bereits festgelegten Rücksprungmarken.

2. Aufgabe: (AGS 3.14)

Definieren Sie einen Datentyp `IntList`, der beliebig lange einfach verkettete Listen ganzer Zahlen darstellen kann. Implementieren Sie eine Funktion `void Insert(IntList *l, int n)`, die in eine bereits sortierte Liste `l` eine ganze Zahl `n` einsortiert. Schreiben Sie anschließend ein kleines Hauptprogramm zum Test der Funktion `Insert`.

3. Aufgabe: (AGS 3.15)

Implementieren Sie entsprechend der 2. Aufgabe eine Funktion `void Insert2(IntList *l, int n)`, die nach einer rekursiven Lösungsstrategie arbeitet. Vergleichen Sie anschließend die rekursive mit der iterativen Lösung.

4. Aufgabe: (AGS 3.19)

Gegeben seien die folgenden Typ- und Variablenvereinbarungen für binäre Bäume:

```
typedef struct ele *node;
typedef struct ele {
  int key;
  node left, right;
} b_ele;

node Tree;
int anz;
```

(a) Schreiben Sie in *C* eine boolesche Funktion `int blatt(node t)`, die für einen beliebigen Baumknoten des gegebenen Typs ermittelt, ob es ein Blatt ist oder nicht. Liefern Sie als Funktionswert eine `1`, wenn es sich um ein Blatt handelt, sonst den Wert `0`.

(b) Gehen Sie davon aus, dass eine boolesche Funktion `int blatt(node t)` gemäß Aufgabe (a) existiert. Schreiben Sie in *C* eine rekursive Funktion `int anzahl(node t)`, die von einem gegebenen Baum des angegebenen Typs unter Verwendung der Funktion `blatt` die Anzahl derjenigen Knoten als Ergebnis liefert, welche keine Blätter als (unmittelbare) Nachfolger haben und selbst auch keine Blätter sind. Geben Sie abschließend ein Beispiel für einen Funktionsaufruf an.

Zusatzaufgabe: (AGS 3.16*)

Gegeben sei die folgende Deklaration für Elemente einer verketteten Liste:

```
typedef struct list_ele *list;
typedef struct list_ele {
  int key;
  list next;
} l_ele_type;
```

(a) Schreiben Sie in *C* eine Funktion `void delete_n(list *l, int n)`, die aus einer beliebigen Liste mit Elementen des oben aufgeführten Typs *alle* Elemente mit dem Schlüsselwert `n` entfernt.

(b) Geben Sie in *C* eine Funktion `int ordered(list l)` an, welche ermittelt, ob eine solche Liste aufsteigend geordnet ist oder nicht, und die Aussage (`0` für nicht geordnet und `1` für geordnet) als Funktionswert zurückgibt.

Beachten Sie: Schlüsselwerte dürfen hier mehrfach vorkommen; in diesem Fall müssen gleiche Schlüsselwerte nebeneinander stehen.