Kurs 1613 „Einführung in die imperative Programmierung“

Musterlösung zur Klausur am 28.03.2009

Aufgabe 1

a)

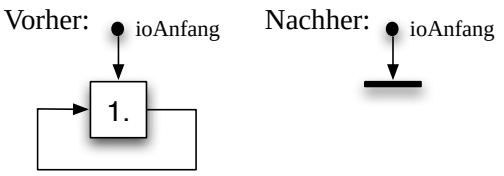
```
procedure FeldMinMax(inFeld: tFeld;
                     var outMin, outMax: integer);
{ Ermittelt den kleinsten sowie den größten aller Werte in
  inFeld und gibt diese in den Ausgabeparametern outMin bzw.
  outMax zurück.}

var min, max:integer;
    i: tIndex;

begin
  {Betrachte zunächst das erste Feldelement. Dieses ist das
   Minimum und das Maximum der bisher betrachteten Elemente.}
  min := inFeld[1];
  max := inFeld[1];
  {Betrachte anschließend alle weiteren Feldelemente und
   aktualisiere ggf. das bisherige Minimum/Maximum}
  for i:=2 to MAXINDEX do
  begin
    if inFeld[i] < min then
      min := inFeld[i];
    if inFeld[i] > max then
      max := inFeld[i];
  end;
  {Wurden alle Feldelemente betrachtet, enthalten min/max
   das Minimum bzw. Maximum aller Feldelemente. -> Ausgeben!}
  outMin := min;
  outMax := max
end;
```

b) Die Verwendung einer For-Schleife ist hier sinnvoll. Zum Ersten steht die Anzahl der Schleifendurchläufe von vornherein fest, da jedes Feldelement genau einmal betrachtet werden muss, und auch die Laufvariable `i` wird ohnehin zum Zugriff auf die einzelnen Array-Positionen benötigt. Die Verwendung der For-Schleife hat also keinen Nachteil. Zum Zweiten müsste bei Verwendung einer While- oder Repeat-Schleife die Laufvariable `i` manuell initialisiert und in jedem Schleifendurchlauf um 1 erhöht werden. Der Programmcode würde damit geringfügig länger und etwas weniger gut lesbar. Außerdem ist es ein häufiger Programmierfehler, das Inkrementieren in der Schleife zu vergessen, wenn man sich ganz auf die eigentliche Aufgabe des Schleifenrumpfes konzentriert.

Aufgabe 2

Im Normalfall arbeitet unsere Prozedur wie folgt: Um das erste Element aus der Liste zu löschen, ist der Next-Zeiger des letzten Elementes zu verändern, wozu wiederum zunächst das letzte Element mit einer Schleife zu suchen ist. Nachdem das erste Element aus der Liste herausgelöst wurde, wird es „disposed“, und der Anfangszeiger wird auf das ehemals zweite Listenelement gesetzt (vgl. Skizze in Aufgabenstellung).

Das setzt jedoch voraus, dass die Liste mindestens zwei Elemente hat. Neben der leeren ist also auch die einelementige Liste ein Sonderfall (vgl. Abbildung rechts). Anders als im Normalfall gibt es hier kein Vorgängerelement, dessen Next-Zeiger zu verändern ist. Auch kann der Anfangszeiger nicht auf das nächste Element weitergerückt werden. Vielmehr ist das einzige Element einfach zu löschen sowie dem Anfangszeiger **`nil`** zuzuweisen. Man erkennt die einelementige zyklische Liste daran, dass das erste Element Nachfolger von sich selbst ist (vgl. Skizze).




```
procedure LoescheErstesElement(var ioAnfang: tRefElement);
{ Löscht das erste Element aus der nicht leeren zyklischen Liste
  mit Anfangszeiger ioAnfang. }
var lauf: tRefElement;

begin
  { Sonderfall: Einelementige Liste }
  if ioAnfang^.next = ioAnfang then
  begin
    dispose(ioAnfang);
    ioAnfang := nil
  end
  else { Normalfall }
  begin
    lauf := ioAnfang;
    { Vorgängerelement suchen }
    while lauf^.next <> ioAnfang do
      lauf := lauf^.next;
    { ioAnfang^ aus Liste ausketten und zerstoeren }
    lauf^.next := ioAnfang^.next;
    dispose(ioAnfang);
    { Anfangszeiger neu setzen }
    ioAnfang := lauf^.next
  end
end;
```

Aufgabe 3

- Alle drei Funktionen bestimmen das Maximum korrekt.

  Erläuterungen (nicht in der Aufgabe verlangt):

  - Funktionsweise von `suchbaumMax1`: Da der linke Teilbaum eines Suchbaums keine größeren Werte als der Wurzelknoten enthalten kann, enthält in einem Suchbaum ohne rechten Teilbaum der Wurzelknoten das Maximum. suchbaumMax1 prüft daher, ob die Wurzel des betrachteten (Teil-)Baums einen rechten Nachfolger hat. Falls nicht, ist der Wert der Wurzel das Maximum, andernfalls ist das Maximum im rechten Teilbaum zu suchen, was hier durch einen rekursiven Aufruf geschieht.
  - Funktionsweise von `suchbaumMax2`: Es werden die folgenden Werte ermittelt:
    — Wert der Wurzel (immer),
    — Maximum des linken Teilbaums (nur, wenn der linke Teilbaum nicht leer ist),
    — Maximum des rechten Teilbaums (nur, wenn der rechte Teilbaum nicht leer ist).
    Der größte dieser ein bis drei Werte wird ausgegeben, er ist das Maximum des Baums.
  - Funktionsweise von `suchbaumMax3`: Es wird iterativ der am weitesten rechts liegende Knoten des Baumes ermittelt und dessen Wert ausgegeben. Bei einem binären Suchbaum enthält dieser Knoten das Maximum, denn alle vorher passierten Knoten ebenso wie deren linke Teilbäume enthalten definitionsgemäß kleinere oder gleiche Werte, und jeder Knoten im linken Teilbaum (sofern vorhanden) des ganz rechten Knotens enthält ebenfalls einen kleineren oder gleichen Wert.

- Worin sich die Ansätze unterscheiden:

  - `suchbaumMax1` und `suchbaumMax2` arbeiten rekursiv, `suchbaumMax3` dagegen iterativ.
  - `suchbaumMax2` durchsucht jeden Knoten des Baums (und funktioniert dabei für beliebige nicht leere binäre Bäume), während `suchbaumMax1` und `suchbaumMax3` die Sortierung eines Suchbaums ausnutzen und den am weitesten rechts liegenden Knoten bestimmen (der in Suchbäumen das Maximum enthält.)

- Welcher Ansatz zu bevorzugen ist:

  - Vorzuziehen ist hier zunächst ein Ansatz, der die Sortierung dazu ausnutzt, nicht den gesamten Baum durchsuchen zu müssen. Von `suchbaumMax1` und `suchbaumMax3` ist dabei der iterative Ansatz (also `suchbaumMax3`) zu bevorzugen, da er ohne Stapel mit konstantem Speicherbedarf auskommt, während ein rekursiver Ansatz hier unnötig den Laufzeitstack belastet.

Aufgabe 4

a) Der Parameter repräsentiert den Anfangszeiger auf das erste Listenelement. Er wird nicht verändert, ist also ein Eingabeparameter.

Der Rückwärtsdurchlauf erfolgt, indem erst die hinter dem ersten Element der eingegebenen (Teil-)Liste hängende Restliste rekursiv bearbeitet und dann der Wert des Nachfolgerelements zum ersten Element addiert wird. Der Rekursionsabbruch erfolgt bei einelementiger Liste, wenn also kein Nachfolger vorhanden ist. Die Musterlösung fängt zusätzlich auch noch den Fall der leeren Liste ab, das war aber nicht verlangt.

```
procedure rueckwaertsAddieren(inAnfang: tRefElement);
{Addiert den Wert des letzten Listenelements zum Wert des
 vorletzten, den neuen Wert des vorletzten zum Wert dritt-
 letzten etc.}
begin
  { Bei leerer Liste ist nichts zu tun }
  if inAnfang <> nil then
    { Bei letztem Listenelement ist auch nichts zu tun }
    if inAnfang^.next <> nil then
    begin
      { Das aktuelle Element ist nicht das letzte.
        1. Restliste rekursiv bearbeiten: }
      rueckwaertsAddieren(inAnfang^.next);
      { 2. Wert des Nachfolgers zum akt. Elem. addieren }
      inAnfang^.info:=inAnfang^.info+inAnfang^.next^.info;
    end
end;
```

b) Iterativ lässt sich eine solche einfach verkettete lineare Liste (ohne zusätzlichen Aufwand) nur vorwärts durchlaufen. Um diese Aufgabe zu lösen, ist aber zuerst das vorletzte Listenelement zu bearbeiten, danach sein Vorgänger etc., die Liste ist also rückwärts zu durchlaufen. Um das zu bewerkstelligen, ist zunächst ein Vorwärtsdurchlauf durch die Liste nötig, in welchem jedes passierte Listenelement auf einen Stapel gelegt wird. Anschließend können die Elemente vom Stapel genommen und wie beschrieben bearbeitet werden (da sie vom Stapel gerade in umgekehrter Reihenfolge abgenommen werden).

Bei der Rekursion wird hierfür implizit der Laufzeitstapel verwendet, während man bei einer iterativen Lösung den Stapel explizit selbst implementieren müsste.

Streng genommen gibt es auch einen alternativen iterativen Ansatz, der ohne Stapel auskommt. Dazu ist zunächst das vorletzte Listenelement mit einer Schleife zu ermitteln. Von dort aus wird in einer weiteren Schleife die Liste rückwärts durchlaufen, indem zum jeweils zuletzt betrachteten Element das jeweilige Vorgängerelement gesucht wird. Letzteres geschieht mit einer inneren Schleife, die vom Listenanfang eben bis zum Vorgänger des zuletzt betrachteten Elements läuft.

Eine solche stapellose Lösung kommt zwar mit konstantem Speicherbedarf aus und kann damit sogar beliebig lange Listen verarbeiten, erkauft sich diesen Vorteil aber mit einem sehr ineffizienten Laufzeitverhalten, da die Liste immer wieder neu von vorne durchlaufen wird. Im Vergleich zur Laufzeit bei Eingabe einer (n-1)-elementigen Liste verlängert sich die Laufzeit bei Eingabe einer n-elementigen Liste bei dieser Fassung um ca. n Schleifendurchläufe, bei den stapelbasierten Varianten dagegen nur um einen Schleifendurchlauf bzw. um einen rekursiven Aufruf. Auch ist diese iterative Lösung mit 3 Schleifen komplizierter als die rekursive Lösung.

Laut Faustregel ist somit die rekursive Lösung vorzuziehen, da es keine einfache iterative Lösung gibt (es wird entweder ein Stapel benötigt oder zeitlich komplexe „Mehrfachdurchläufe“). Die Empfehlung ist aufgrund der linearen Rekursionstiefe insofern zu relativieren, als dass die rekursive Lösung nur für nicht allzu lange Listen funktioniert. Sollten die Listen so lang werden können, dass der begrenzte Laufzeitstack zum Problem wird, wäre ggf. eine iterative Lösung mit dynamischem Hilfsstapel vorzuziehen, für den mehr Speicherplatz zur Verfügung steht. Die stapellose Lösung könnte höchstens dann als sinnvoll angesehen werden, wenn man Wert auf die Möglichkeit legt, beliebig lange Listen verarbeiten zu können, ganz egal wie lange es dauert.

Aufgabe 5

a) Die Äquivalenzklassen müssen paarweise disjunkt sein und die Vereinigung aller Äquivalenzklassen muss genau die Menge der zulässigen Ausgaben ergeben. Dazu fehlen jedoch noch zwei Klassen:

Für $min = max$ sind mit A_3 und A_4 die Fälle abgedeckt, dass beide Werte kleiner als Null sind und dass beide Werte gleich 0 sind. Es fehlt noch der Fall, dass beide Werte > 0 sind:

$A_7 := \{ (min, max) \mid 0 < min, min = max \}$

Für $min \neq max$ sind mit A_1, A_2, A_5 und A_6 die Fälle abgedeckt, dass beide Werte < 0 sind, beide Werte > 0 sind oder einer beider Werte = 0 ist. Es fehlt noch der Fall, dass das Mininum kleiner und das Maximum größer als Null ist:

$A_8 := \{ (min, max) \mid min < 0, 0 < max \}$

b) Das einzige Testdatum zu Klasse A_4 ist ([0,0,0], 0, 0).

(Der Testfall zu Klasse A_4 enthält genau dieses eine Testdatum, denn nur wenn das Feld ausschließlich Nullen enthält, sind sowohl Minimum als auch Maximum aller Feldwerte = 0.)

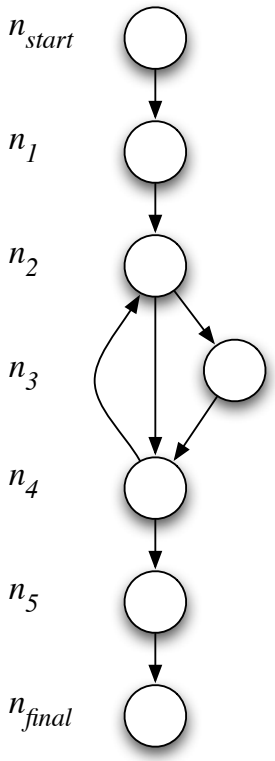
Ein mögliches Testdatum zu Klasse A_5 ist beispielsweise ([2,7,4], 2, 7).

(Allgemein muss für eine Ausgabe der Klasse A_5 ein Array eingegeben werden, dessen Elemente sämtlich größer als Null sind und nicht alle drei wertgleich sind. Der Testfall zu Klasse A_5 lässt sich also formal wie folgt beschreiben:

$\{ ([a,b,c], x, y) \mid a>0, b>0, c>0, (a \neq b \vee a \neq c \vee b \neq c),$
$x = \min(\{a, b, c\}), y = \max(\{a, b, c\}) \})$

Aufgabe 6

a) Wir geben im Folgenden den Kontrollflussgraphen an, wobei wir zu Gunsten der Lesbarkeit nicht nur die Zeilennummern, sondern auch die Programmzeilen selbst neben die Knoten schreiben.
Zeile 4 enthält nur ein Schlüsselwort und darf daher alternativ auch Knoten n_1 zugeordnet werden. Die Zuordnung aller weiteren Programmzeilen ist eindeutig.




```
1 begin
2   gefunden := false;
3   i := 1;

4   repeat
5     i := i + 1;
6     if inFeld[i] = 0 then

7       gefunden := true;

8   until gefunden or (i = FELDMAX);

9   enthaeltNull := gefunden;
10 end;
```

b) Die Boundary-Klasse besteht aus genau allen Pfaden, welche die Schleife nicht wiederholen. Aufgrund der If-Verzweigung im Schleifenrumpf gibt es hier zwei mögliche Pfade:
A) (n_{start}, n_1, n_2, n_3, n_4, n_5, n_{final}) und
B) (n_{start}, n_1, n_2, n_4, n_5, n_{final}) .
Die Interior-Klasse für n=2 besteht aus genau allen Pfaden, welche die Schleife genau einmal wiederholen. Das sind folgende vier Pfade:
C) (n_{start}, n_1, n_2, n_3, n_4, n_2, n_3, n_4, n_5, n_{final}),
D) (n_{start}, n_1, n_2, n_3, n_4, n_2, n_4, n_5, n_{final}),

---

E) (n_{start}, n_1, n_2, n_4, n_2, n_3, n_4, n_5, n_{final}) und
F) (n_{start}, n_1, n_2, n_4, n_2, n_4, n_5, n_{final}).

c) *Es sind genau die Pfade A und E ausführbar:*
Die Schleife bricht ab, sobald `FELDMAX-1` (also 3) Durchläufe absolviert wurden oder sobald die Variable `gefunden` den Wert true annimmt. Für einen Abbruch nach einem oder zwei Durchläufen, muss demnach `gefunden` true werden, was wiederum genau im Then-Fall (Knoten n_3) eintritt. D.h. ein ausführbarer Pfad (mit weniger als 3 Durchläufen) muss Knoten n_3 genau im letzten Schleifendurchlauf passieren. Die Pfade B und F sind nicht ausführbar, da sie n_3 nicht passieren, also einen Schleifenabbruch trotz `gefunden`=false voraussetzen. Die Pfade C und D sind nicht ausführbar, da sie einen zweiten Schleifendurchlauf trotz bereits im ersten Durchlauf wahr gewordener Abbruchbedingung beschreiben.