Kurs 1613 "Einführung in die imperative Programmierung"

Musterlösung zur Nachklausur am 08.04.2006

Aufgabe 1

a) Idee: Die Zahl wird so lange durch 10 dividiert, bis sie einstellig ist. Die gesuchte Ziffernzahl ist um eins größer als die Anzahl dieser Divisionen.

```
function countZiff (inZahl : tNatZahl): tNatZahl;
{gibt die Stellenanzahl der inZahl zurück}

  var
  i,
  j : tNatZahl;

begin
  i := inZahl;
  j := 1;

  while i >= 10 do
  begin
    j := j + 1;
    i := i div 10
  end;

  countZiff := j
end;
```

b)

```
function potenziere (inZahl,
                     inPotenz: tNatZahl): tNatZahl;
{berechnet die n-te Potenz einer Zahl}

  var
  i,
  ZwischErg : tNatZahl;

begin
  ZwischErg := 1;
  for i := 1 to inPotenz do
    ZwischErg := ZwischErg * inZahl;

  potenziere := ZwischErg
end;
```

c) In der Funktion `isArmstrong` ist die Funktion `potenziere` an der Stelle (2) oder (3) aufzurufen.
Eine geeignete Programmzeile ist:
`ZwischErg := potenziere(LetzteZiffer, AnzDerStellen).`

d)

```
program ArmstrongZahlen;

  var
  i,
  x,
  y : tNatZahl;

begin
  readln(x);
  readln(y);
  for i := x to y do
  begin
    if isArmstrong(i) then
      writeln(i,' ist eine Armstrong-Zahl.');
  end {for}
end;
```

Aufgabe 2

a)

```
function kommtVor(inRefListe:tRefListe; inWert:integer):
                                                  boolean;
{Rückgabe true, falls die übergebene Liste ein Element mit
 info-Komponente inWert enthält, sonst false}
  var lauf: tRefListe;

begin
  if inRefListe<>nil then
  begin
    lauf := inRefListe;
    while (lauf^.next <> nil) and (lauf^.info <> inWert) do
      lauf := lauf^.next;
    { Schleife terminiert, wenn ein Vorkommen gefunden wurde
     oder das letzte Element erreicht ist.
     Genau im ersten Fall ist das Ergebnis true: }
    kommtVor := (lauf^.info=inWert)
```

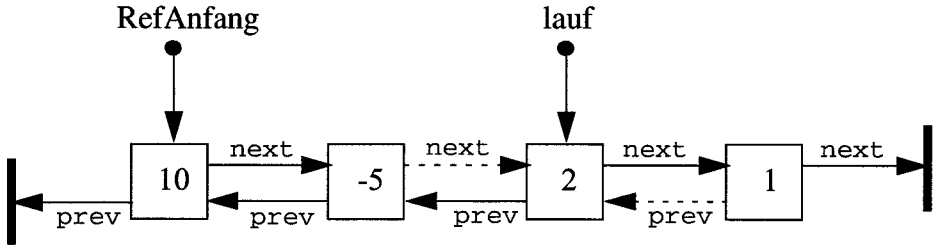
```
    end
    else {inRefListe=nil}
      kommtVor:=false
end;
```

b) Da vorausgesetzt werden durfte, dass genau ein Element mit Wert 0 in der Liste vorkommt, entfällt der Sonderfall der leeren Liste. Wir prüfen daher als erstes, ob die Liste mit 0 beginnt, denn dann ist gar nichts zu tun. Andernfalls suchen wir zunächst das 0-Element, genauer: dessen Vorgänger. Außerdem müssen wir in einer zweiten Schleife noch das letzte Element suchen, da dessen next-Zeiger auch geändert werden muss (siehe Skizze in Aufgabenstellung). Nachdem nun Zeiger auf das erste Element, das letzte Element und den Vorgänger des 0-Elements vorliegen, kann die Verkettungsänderung durch Verändern von drei Zeigern vorgenommen werden.

```
procedure Tausche(var ioRefAnfang: tRefListe);
{ siehe Aufgabenstellung }
var lauf, ende: tRefListe;
begin
  if ioRefAnfang^.info <> 0 then
  { Nur wenn 0 nicht das erste Element ist, muss ueberhaupt
    etwas getan werden }
  begin
    lauf := ioRefAnfang;
    while lauf^.next^.info <> 0 do
      lauf := lauf^.next;
    { lauf zeigt jetzt auf den Vorgaenger des 0-Elements}
    ende := lauf^.next;
    while ende^.next <> nil do
      ende := ende^.next;
    { ende zeigt jetzt auf das letzte Listenelement }
    { Jetzt koennen die Zeiger geaendert werden : }
    ende^.next := ioRefAnfang;
    ioRefAnfang := lauf^.next;
    lauf^.next := nil
  end
end;
```

Aufgabe 3

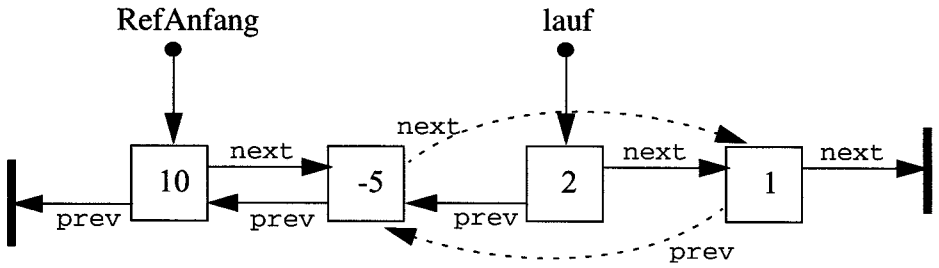
a) Da in der doppelt verketteten Liste jedes Element auch wieder eine Referenz auf seinen Vorgänger hat, ist es zum Entfernen nicht nötig, den Vorgänger des zu entfernenden Elementes zu suchen, sondern es kann direkt das zu löschende Element gesucht werden.
Die folgende Skizze zeigt die Liste aus der Aufgabenstellung und einen Laufzeiger, mit dem das Element mit info-Wert „2" gesucht und gefunden wurde:



Zum Entfernen müssen die beiden Zeiger geändert werden, die auf dieses Element „2" verweisen, in der Skizze oben gepunktet dargestellt, also der `next`-Zeiger des Vorgängers und der `prev`-Zeiger des Nachfolgers:

```
lauf^.prev^.next := lauf^.next; und
lauf^.next^.prev := lauf^.prev;
```

Die Situation nach dieser Verkettungsänderung wird in folgender Skizze dargestellt:



Jetzt muss nur noch das Element, auf das `lauf` verweist, gelöscht werden:
`dispose(lauf).`

Sonderfälle:
Der Sonderfall, dass das Element in der Liste nicht vorkommt, wurde in der Aufgabenstellung ausgeschlossen, ebenso wie der Spezialfall der leeren Liste. Damit verbleiben noch folgende zwei Sonderfälle, die zu behandeln sind:

- Das zu löschende Element kann das letzte sein. In diesem Fall hat es keinen Nachfolger und es entfällt das Ändern von `lauf^.next^.prev`. Mit anderen Worten: `lauf^.next^.prev` muss nur geändert werden, falls `lauf^.next<>nil` ist.
- Das zu löschende Element kann das erste sein. In diesem Fall entfällt das Ändern von `lauf^.prev^.next` und es muss noch der Anfangszeiger `ioRefAnfang` auf das zweite Element `lauf^.next` gesetzt werden.

```
procedure loeschen(var ioRefAnfang: tRefDVListe;
                   inSuch:integer);
{Loescht das erste Listenelement mit Info inSuch. Es muss ein
solches Element existieren!}

var lauf: tRefDVListe;

begin
  {Zu loeschendes Element suchen}
  lauf:=ioRefAnfang;
  while lauf^.info <> inSuch do
    lauf:=lauf^.next;

  {Wenn es Nachfolger gibt, dessen prev aktualisieren.}
  if lauf^.next<>nil then
    lauf^.next^.prev := lauf^.prev;

  {Wenn es Vorgänger gibt, dessen next aktualisieren.}
  if lauf^.prev<>nil then
    lauf^.prev^.next := lauf^.next
  else {Erstes Elem. gelöscht => Anfangszeiger ändern!}
    ioRefAnfang:=lauf^.next;

  {Nun noch das Element selbst löschen}
  dispose(lauf)
end;
```

b) Die Typen `tRefDVListe`/`tDVListe` sowie `tRefBinBaum`/`tBinBaum` beschreiben keine Listen oder Bäume, sondern lediglich *Elemente* solcher Datenstrukturen (genauer: `tDVListe` bzw. `tBinBaum` beschreiben die Elemente, `tRefDVListe` bzw. `tRefBinBaum` beschreiben Zeiger auf solche Elemente). Die Elemente eines Binärbaums und einer doppelt verketteten Liste sind gleich aufgebaut: Sie bestehen hier aus einem Integer und zwei Zeigern. Die eigentliche Struktur, z.B. eine Listen- oder Baumstruktur, wird erst durch geeignete Verkettung der einzelnen Elemente zur Laufzeit gebildet und kommt nicht in den Typen zum Ausdruck. Somit stellen diese Typen auch nicht sicher, dass eine Datenstruktur, die der Prozedur `loeschen` aus Teil a) übergeben wird, immer tatsächlich eine korrekte doppelt verkettete Liste ist.

Aufgabe 4

a) Wir geben im Folgenden eine mögliche Implementierung an, wobei wir einen Postorder-Durchlauf absolvieren, d.h. zuerst jeweils die beiden Teilbäume spiegeln und erst dann die jeweilige Wurzel bearbeiten. Die Spiegelung funktioniert aber auch mit Pre- oder Inorder-Durchläufen.
Da ein Baum auch leer sein kann und wir somit den Sonderfall des leeren Baumes ohnehin berücksichtigen müssen, bietet sich diese Prüfung auch gleich als Rekursionsabbruch an, d.h. wir prüfen nicht extra, ob `links` bzw. `rechts` ungleich **`nil`** ist, bevor wir einen rekursiven Subaufruf tätigen.

```
procedure spiegeln(inRefWurzel: tRefBinBaum);
{Spiegelt einen binären Baum durch Vertauschen der
 Nachfolger eines jeden Knotens.}
var merker:tRefBinBaum;

begin
  {Für leeren Baum ist nichts zu tun.}
  if inRefWurzel<>nil then
  begin
    {Spiegeln beider Teilbaeume:}
    spiegeln(inRefWurzel^.links);
    spiegeln(inRefWurzel^.rechts);

    {Nun Spiegelung durch Vertauschen
     beider Teilbäume komplettieren:}
    merker:=inRefWurzel^.links;
    inRefWurzel^.links:=inRefWurzel^.rechts;
    inRefWurzel^.rechts:=merker;
  end;
end;
```

b) Es handelt sich um eine sinnvolle Anwendung der Rekursion, da jeder Knoten des Baumes bearbeitet werden muss und somit der gesamte Baum einmal durchlaufen werden muss. Um bei einem solchen Durchlauf nach Abarbeitung eines Teilbaums wieder zum Vorgängerknoten zurückspringen zu können, wird ein Stapel benötigt, der bei einer iterativen Lösung explizit zu implementieren wäre, während bei der Rekursion der Laufzeitstapel diese Aufgabe übernimmt.

Aufgabe 5

Ein Testfall ist eine Menge von Testdaten, wobei ein Testdatum wiederum ein Paar ist, welches zum Ersten die Eingabedaten (hier: Eingabe für Parameter inZahl) und zum Zweiten die bei dieser Eingabe laut Spezifikation *erwartete* Ausgabe enthält.

Für Boundary- und Interior-Test geben wird exemplarisch zwei verschiedene Notationsmöglichkeiten (eine formale und eine mit natürlichsprachiger Umschreibung der Quersumme) an.

Abweisender Test:

$$T_0 = \{ (e, e) \mid e \in \{0, ..., 9\} \}$$

Boundary-Test:

$$T_1 = \{ (10z+e, z+e) \mid z \in \{1, ..., 9\} \wedge e \in \{0, ..., 9\} \}$$
$$= \{ (e, a) \mid e \in \{10, ..., 99\} \wedge$$
a ist die Summe der zwei Ziffern von e}

Interior-Test (2 Durchläufe)

$$T_2 = \{ (100h+10z+e, h+z+e) \mid h \in \{1, ..., 9\} \wedge z,e \in \{0, ..., 9\} \}$$
$$= \{ (e, a) \mid e \in \{100, ..., 999\} \wedge$$
a ist die Summe der drei Ziffern von e}