Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 08.04.2006

Aufgabe 1 (2+2+1+2 Punkte)

In der Mathematik gibt es die so genanten Armstrong-Zahlen, die wie folgt definiert werden:

> Eine Armstrong-Zahl ist eine Zahl, bei der die Summe der einzelnen Ziffern, die jeweils mit der Anzahl der Ziffern der Zahl potenziert sind, gleich der Zahl ist.
>
> z.B.: $153 = 1^3 + 5^3 + 3^3$ (3 Ziffern je hoch 3)

a) Schreiben Sie eine Funktion `countZiff`, welche die Anzahl der Ziffern einer positiven Zahl zurückliefert (in unserem Beispiel die Zahl 3). Benutzen Sie den im Programmrahmen `ArmstrongZahlen` (siehe unten) vorgegebenen Funktionskopf.

b) Schreiben Sie eine Funktion `potenziere`, welche die n-te Potenz einer Zahl berechnet. Der Sonderfall des undefinierten Wertes 0^0 muss nicht berücksichtigt werden. Benutzen und ergänzen Sie den vorgegebenen Funktionskopf (siehe unten).

c) Die Funktion `isArmstrong` (siehe rechte Seite) gibt *true* zurück wenn der übergebene Parameter eine Armstrong-Zahl ist, sonst *false*. In dieser Funktion werden die Funktionen `countZiff` und `potenziere` benutzt.
Bestimmen Sie, an welcher der Stellen (1) - (4) in der Funktion `isArmstrong` die Funktion `potenziere` einzusetzen ist und geben Sie die komplette Programmzeile an.

d) Schreiben Sie ein Programm, das zwei positive ganze Zahlen x und y ($x > 0$, $y <$ MAXINT, $x < y$, diese Bedingungen müssen Sie nicht überprüfen) einliest und dann alle Armstrong-Zahlen mit $x \leq a \leq y$ ausgibt. Auch wenn Sie die Teilaufgaben a) - c) nicht gelöst haben, können Sie die Funktionen voraussetzen. Nutzen Sie dazu den vorgegebenen Programmrahmen.

```
program ArmstrongZahlen (input, output);

  type
    tNatZahl = 0..maxint;
  ...
  function countZiff (inZahl : tNatZahl): tNatZahl;
  {gibt die Stellenanzahl der inZahl zurück}
      ...

  function potenziere ( ??? ): tNatZahl;
  {berechnet die n-te Potenz einer Zahl}
      ...

  function isArmstrong (inZahl : tNatZahl): boolean;
  {prüft ob inZahl eine Armstrong-Zahl ist}
      ...
```

Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 08.04.2006

```
function isArmstrong (inZahl : tNatZahl): boolean;
{prüft ob inZahl eine Armstrong-Zahl ist}
  var
  i,
  j,
  ggfArmstrong,
  AnzDerStellen
  ZwischErg,
  EndErg,
  LetzteZiffer : tNatZahl;

begin
  ggfArmstrong := inZahl;
  AnzDerStellen := countZiff(inZahl);
  EndErg := 0;

  for i := 1 to AnzDerStellen do
  begin
    (1)
    LetzteZiffer := ggfArmstrong mod 10;
    (2)
    ggfArmstrong := ggfArmstrong div 10;
    (3)
    EndErg := EndErg + ZwischErg
    (4)
  end;

  if EndErg = inZahl then
    isArmstrong := true
  else
    isArmstrong := false;
end;
```

Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 08.04.2006

Aufgabe 2 (3+5 Punkte)

Gegeben ist folgende Definition einer linearen Liste:

```
type
tRefListe = ^tListe;
tListe = record
           info: integer;
           next: tRefListe
         end;
```

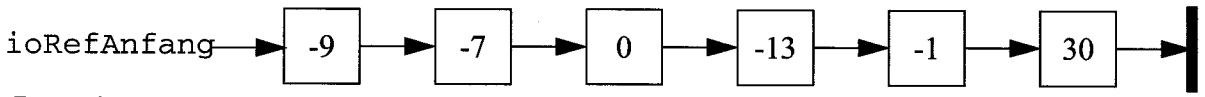
a) Implementieren Sie eine Funktion `kommtVor`, die prüft, ob in einer übergebenen linearen Liste ein Element mit einem ebenfalls zu übergebenden info-Wert vorkommt.
Verwenden Sie folgenden Funktionskopf:

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

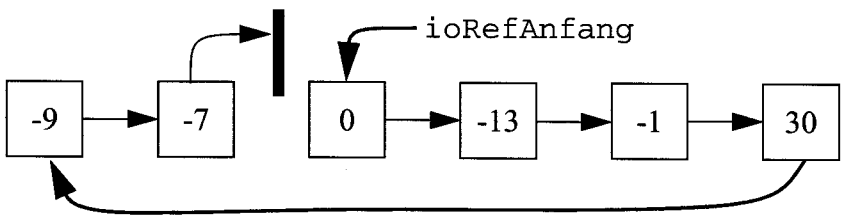
b) Implementieren Sie eine Prozedur `Tausche`, die in einer übergebenen Liste das Element mit der info-Komponente 0 sucht und die mit diesem Element beginnende Restliste nur durch Ändern der Verkettung aushängt und an den Beginn der Liste einfügt. Sie können davon ausgehen, dass die 0 in der Liste genau einmal vorkommt. Ist das Element mit info-Komponente 0 das erste, so ist nichts zu tun.
Verwenden Sie folgenden Prozedurkopf:

```
procedure Tausche(var ioRefAnfang: tRefListe);
```

Ein Beispiel:
Vor dem Aufruf:



Danach:



Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 08.04.2006

Aufgabe 3 (5+2 Punkte)

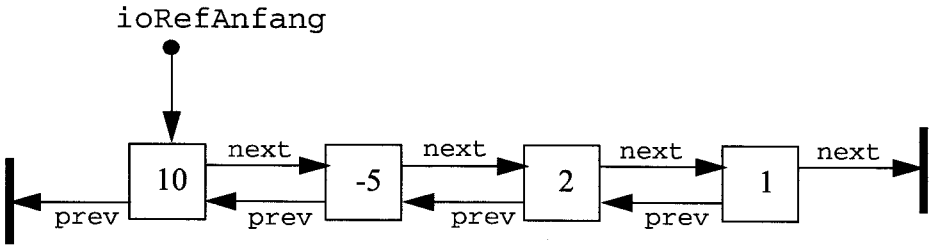
Gegeben sei folgende Typdeklaration:

```
type
  tRefDVListe = ^tDVListe;
  tDVListe =  record
                info: integer;
                prev,
                next:tRefDVListe
              end;
```

Mit Hilfe dieses Typs wird eine so genannte *doppelt verkettete Liste* gebildet. Im Gegensatz zu den im Kurs betrachteten einfach verketteten linearen Listen besitzt hier ein Listenelement neben einem Zeiger `next` auf seinen Nachfolger auch einen Zeiger `prev` auf seinen Vorgänger. So wie der `next`-Zeiger des letzten Elementes zeigt der `prev`-Zeiger des ersten Elementes auf **`nil`**.

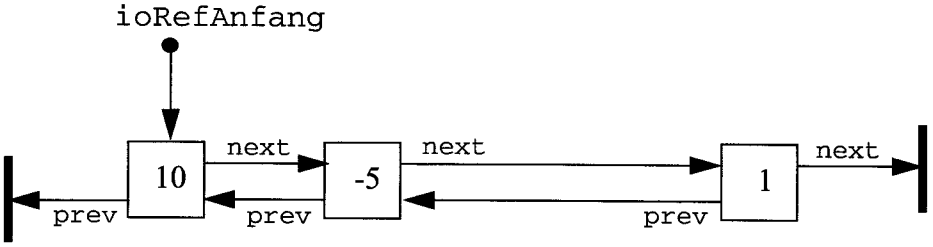
Eine solche Liste werde unverändert über einen Anfangszeiger angesprochen, der das erste Element referenziert.

Die folgende Skizze zeigt ein Beispiel einer solchen doppelt verketteten Liste aus vier Elementen.



a) Implementieren Sie eine Prozedur `loeschen`, die das erste Element mit einem zu übergebenden `info`-Wert in der Liste sucht und es aus der Liste löscht. Sie dürfen dabei davon ausgehen, dass das zu löschende Element in der Liste vorkommt.

Die folgende Skizze zeigt z.B. das Ergebnis, wenn aus oben abgebildeter Liste das Element mit Info „2" gelöscht wurde.



Benutzen Sie folgenden Prozedurkopf:

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

b) Vergleicht man obige Typdeklaration zur Bildung doppelt verketteter Listen mit der z.B. in Aufgabe 4 gegebenen Typdeklaration zur Bildung binärer Bäume, so stellt man fest, dass beide strukturell identisch sind und sich nur in Bezeichnern unterscheiden.

Erklären Sie, warum derselbe Typ sowohl zur Bildung von Bäumen als auch zur Bildung von doppelt verketteten Listen verwendet werden kann, obwohl doch beides unterschiedliche Datenstrukturen sind!

Tipp: Überlegen Sie sich, was genau die Typen `tDVListe` bzw. `tBinBaum` sowie `tRefDVListe` bzw. `tRefBinBaum` beschreiben.

Aufgabe 4 (3+1 Punkte)

Gegeben sei folgende Typdeklaration zur Bildung eines binären Baumes:

```
type
  tRefBinBaum = ^tBinBaum;
  tBinBaum =  record
                info: integer;
                links,
                rechts:tRefBinBaum
              end;
```

a) Implementieren Sie *rekursiv* eine Prozedur `spiegeln`, die einen beliebigen übergebenen Baum aus Knoten des Typs `tRefBinBaum` spiegelt, indem für jeden Knoten des Baumes die beiden Nachfolger vertauscht werden.
Sollte der eingegebene Baum sortiert sein, so wird durch diese Spiegelung übrigens die Sortierung umgekehrt, d.h. aus einem aufsteigend sortierten Baum würde ein absteigend sortierter und umgekehrt.

Verwenden Sie den folgenden Prozedurkopf, wobei an der mit **??** gekennzeichneten Stelle die Übergabeart geeignet zu ergänzen ist:

```
procedure spiegeln(??RefWurzel: tRefBinBaum);
```

b) Handelt es sich hier um eine sinnvolle Anwendung der Rekursion? Begründen Sie Ihre Antwort. (Eine unbegründete Antwort wie „Ja" oder „Nein" wird nicht bewertet!)

Aufgabe 5 (3 Punkte)

Die Funktion `quersumme` soll die Quersumme einer eingegebenen natürlichen Zahl berechnen, also die Summe all ihrer Ziffern.

Typdefinition und Funktion seien wie folgt gegeben.

```
type
tNatZahl = 0..maxint;

function quersumme (inZahl : tNatZahl): tNatZahl;
{berechnet die Quersumme von inZahl}

    var
    qs,
    rest : tNatZahl;

begin
  qs:=0;
  rest:=inZahl div 10;
  while rest > 0 do
  begin
    qs := qs + rest mod 10;
    rest := rest div 10
  end; {while}
  quersumme:=qs;
end; {quersumme}
```

Ermitteln Sie die Testfälle für den *boundary-interior Pfadtest* (interior-Test für genau zwei Schleifendurchläufe) der Funktion `quersumme`. Sie brauchen nicht die zu den Testfällen gehörenden Pfade im Kontrollflussgraphen anzugeben.