Kurs 1613 "Einführung in die imperative Programmierung"

Musterlösung zur Klausur am 03.03.2007

Aufgabe 1

```
procedure tauschen(inVor1, inVor2:tRefElement);
{Vertauscht zwei Elemente einer Liste durch Änderung der Verket-
tung. In inVor1 und inVor2 werden dabei Referenzen auf die Vor-
gängerelemente der zu vertauschenden Elemente übergeben.
Vorbedingung:
inVor1 und inVor2 zeigen jeweils auf ein existierendes Element
derselben Liste, jedoch nicht auf das letzte.}

var
  Element1, Element2, Nach1, Nach2 : tRefElement;

begin
  {Der Übersichtlichkeit halber setzen wir Hilfszeiger auf die
  zu vertauschenden Elemente sowie ihre Nachfolger:}
  Element1 := inVor1^.next;
  Element2 := inVor2^.next;
  Nach1 := Element1^.next;
  Nach2 := Element2^.next;

  {Damit ist die Verkettungsänderung nun einfach:}
  inVor1^.next := Element2;
  Element2^.next := Nach1;
  inVor2^.next := Element1;
  Element1^.next := Nach2;
end;
```

Aufgabe 2

Das Hauptproblem besteht darin, zu unterscheiden, welche Elemente addiert und welche subtrahiert werden müssen. Wenn man den Algorithmus näher betrachtet, stellt man fest, dass jedes Element mit geradem Array-Index subtrahiert und jedes mit ungeradem Index addiert wird.
Im Folgenden zeigen wir eine besonders kurze Lösung, die diese Beobachtung ausnutzt:

```
function AltSumme(inFeld: tIntFeld): integer;
{ Berechnet alternierende Summe der Feldelemente:
Elemente an geraden Positionen werden subtrahiert, Elemente an
ungeraden Positionen werden addiert.}
var
  summe: integer;
  i: tIndex;

begin
  summe := 0;
  for i := 1 to MAX do
    if i mod 2 = 0 then {Ungerader Index => Subtraktion}
      summe := summe - inFeld[i]
    else  {Gerader Index => Addition}
      summe := summe + inFeld[i];
  AltSumme := summe
end;
```

Aufgabe 3

Wir geben hier der Nachvollziehbarkeit halber eine vollständige Implementierung inklusive der Prozedur `trennen` an. Die Teile, die nicht zur Lösung gehören, sind kleiner gedruckt.

Anmerkung: Die beiden inneren If-Prüfungen in der Prozedur `sortiere` sind in dieser Form nicht notwendig, jedoch muss im ersten Fall mindestens geprüft werden, ob `0 < trennstelle` gilt, da sonst der Wert des Ausdrucks `trennstelle-1` nicht mehr im Wertebereich von `tIndex` läge. Entsprechend muss vor dem zweiten rekursiven Aufruf mindestens `trennstelle < MAX` sichergestellt werden.

```
program qsort;

const MAX=10;

type
  tIndex=1..MAX;
  tIntFeld=array[tIndex] of integer;

var
  i:tIndex;
  a:tIntFeld;
```

```
procedure trenne(inVon, inBis:tIndex; var ioFeld:tIntFeld;
                 var outTE: tIndex);
{Vorbedingung: inVon < inBis
 Ordnet die Elemente von ioFeld im Ausschnitt von inVon bis inBis um und gibt
 einen Feldindex outTE (TE steht für Trenn-Element) aus, so dass folgende
 Nachbedingungen erfüllt sind:
 inVon <= outTE <= inBis
 Für alle i mit inVon<=i<outTE gilt ioFeld[i] <= ioFeld[outTE].
 Für alle j mit outTE <j<= inBis gilt ioFeld[j] > ioFeld[outTE].
}
  var
    lauf,
    einfuegestelle,
    pivot:tIndex;

  procedure tausche(var ioFeld:tIntFeld; inI1, inI2:tIndex);
  {Vertauscht zwei Elemente im Array}
  var merke:integer;
  begin
    merke:=ioFeld[inI1];
    ioFeld[inI1]:=ioFeld[inI2];
    ioFeld[inI2]:=merke
  end;

begin {trenne}
  einfuegestelle:=inVon;
  pivot:=inBis;
  {Pivotelement ist das ganz rechte. Alle anderen Feldelemente werden nun
  durchlaufen. Jedes, das <= Pivotelement ist, wird an die Einfügestelle
  verschoben (durch Vertauschen). Die neue Einfügestelle ist dann 1 Element
  hinter dem eingefügten Element, so dass alle Elemente vor der Einfügestelle
  immer <= Pivotelement sind.}
  for lauf:=inVon to inBis-1 do
  begin
    if ioFeld[lauf]<=ioFeld[pivot] then
    begin
      tausche(ioFeld,einfuegestelle,lauf);
      einfuegestelle := einfuegestelle+1;
    end;
  end;
  {Nun liegen alle Elemente <= Pivotelement links von der Einfügestelle, ab
  der Einfügestelle bis einschließlich ioFeld[inBis-1] folgen nur noch
  größere. Nun nur noch das Pivotelement an die Einfügestelle verschieben und
  diese neue Position als Ergebnis ausgeben:}
  tausche(ioFeld,pivot,einfuegestelle);
  outTE := einfuegestelle;
end; {trenne}
```

```
procedure sortiere(inVon, inBis:tIndex; var ioFeld:tIntFeld);
{Sortiert den Ausschnitt des Feldes ioFeld von inVon bis inBis
 (jeweils inklusive) aufsteigend, sofern inVon <= inBis gilt.
 Um das gesamte Feld zu sortieren, ist inVon=1 und inBis=MAX
 zu übergeben.}
var trennstelle:tIndex;
begin
  if (inVon < inBis) then {es gibt noch etwas zu sortieren.}
  begin
    trenne(inVon, inBis, ioFeld, trennstelle);
    {nun gilt inVon<=trennstelle<=inBis und
    alle Elemente von inVon bis trennstelle-1 sind
    <= ioFeld[trennstelle] und alle von trennstelle +1 bis inBis
    sind > ioFeld[trennstelle]. }
    if inVon < trennstelle then
      sortiere(inVon, trennstelle-1, ioFeld);
    if trennstelle < inBis then
      sortiere(trennstelle+1, inBis, ioFeld);
  end;
end;

begin {Hauptprogramm zum Test}
  for i:=1 to MAX do
    readln(a[i]);
  sortiere(1,MAX,a);
  for i:=1 to MAX do
    write(a[i], ' ');
  writeln;
end.
```

Aufgabe 4

Nachfolgend geben wir die gesamte Funktion an. Die bereits in der Aufgabe vorgegebenen Teile sind dabei kleiner, die Musterlösungen für die Ergänzungen zu a) und b) entsprechend größer gedruckt.

```
function testeUnsortiert(inRefWurzel: tRefBinBaum):boolean;
{Sucht nach einem Beweis, dass der Baum mit Wurzel inWurzel unsortiert ist.
Ausgabe true impliziert, dass der Baum nicht sortiert ist.
Ausgabe false sagt nichts über die Sortierung des Baumes aus.
Für den leeren Baum wird false ausgegeben.}

var gefunden,
    hatlinks,
    hatrechts: boolean;
    lauf: tRefBinBaum;
```

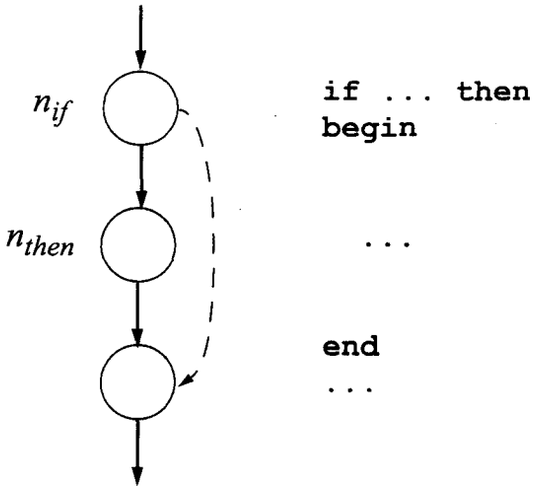
```
begin
  gefunden:=false;
  {gefunden gibt an, ob ein Knoten mit kleinerem rechten oder
   größerem linken Nachfolger gefunden wurde.}
  lauf:=inRefWurzel;
  while (lauf <> nil) and not gefunden do
  begin
    hatlinks := lauf^.links <> nil;
    hatrechts := lauf^.rechts <> nil;

    {Existiert größerer linker oder kleinerer rechter Nachf.?}
    if hatlinks then
      if lauf^.links^.info > lauf^.info then
        gefunden:=true;
    if hatrechts then
      if lauf^.rechts^.info < lauf^.info then
        gefunden:=true;

    {Ansonsten weitersuchen}
    if not gefunden then
      if not hatlinks then
        {hat entweder nur rechten Nachfolger oder gar keinen}
        lauf:=lauf^.rechts
      else {hat mindestens linken Nachf.}
        if not hatrechts then {hat *nur* linken Nachf.}
          lauf:=lauf^.links
        else
          {Hat linken und rechten Nachf., wobei gilt:
          lauf^.links^.info <= lauf^.info <= lauf^.rechts^.info}
          if lauf^.info-lauf^.links^.info >
              lauf^.rechts^.info-lauf^.info then
            lauf:=lauf^.links
          else
            lauf:=lauf^.rechts;
  end;
  testeUnsortiert := gefunden;
end;
```

Zu c)

Das Durchlaufen eines Pfades durch den Baum kann — wie ein Listendurchlauf — durch einfaches Traversieren von Zeigern geschehen. Im Gegensatz zum vollständigen Baumdurchlauf werden keine Rücksprünge benötigt, weshalb die iterative Lösung ohne Stapel auskommt. Eine rekursive Implementierung würde dagegen implizit einen Stapel aufbauen und ist daher nicht sinnvoll.

Aufgabe 5

a) Bei Eingabe von 20 wird zwar eine vollständige Anweisungsüberdeckung, jedoch *keine vollständige Zweigüberdeckung* erreicht:
Die If-Bedingung in Zeile 8 wird bei beiden Schleifendurchläufen zu true ausgewertet, d.h. der abweisende Else-Zweig (ohne Anweisungen) wird nicht durchlaufen.
Zur Verdeutlichung der Problematik geben wir hier einen Ausschnitt aus einem Kontrollflussgraphen an:



Wird eine If-Bedingung immer erfüllt, also der Then-Zweig immer durchlaufen, so bleibt die in der Abbildung gestrichelte Kante, die den Else-Zweig repräsentiert, unüberdeckt.

b) Für den Boundary-Test ist die Schleife genau einmal zu durchlaufen.
Der Testfall dazu ist die Menge genau derjenigen Testdaten, deren Eingabedaten zu exakt einem Schleifendurchlauf führen. Die Eingabedaten müssen daher einstellige Zahlen größer Null sein. Die zweite Angabe eines Testdatums stellt das laut Spezifikation erwartete Ergebnis dar, also die kleinste Ziffer. Bei einstelligen Zahlen stimmt die erwartete Ausgabe mit der Eingabe überein.
Der Testfall lautet somit:
$\{ (z, z) \mid z > 0 \text{ und } z < 10 \} = \{ (1,1), (2,2), (3,3), ..., (9,9) \}$