Kurs 1613 “Einführung in die imperative Programmierung”

Musterlösung zur Klausur am 03.03.2001

Lösung 1 (4+4 Punkte)

a)

```
  procedure VerschiebeZyklisch (var ioFeld : tFeld);
  { verschiebt die Werte innerhalb eines Feldes eine Position
    nach rechts; der Wert ioFeld[MAX] wird nach ioFeld[1]
    übertragen }

    var
    sicher: integer;
    ix: tIndex;

  begin
    sicher := ioFeld[MAX];
    (* Der letzte Feldinhalt wird gesichert *)
    for ix := MAX downto 2 do
      ioFeld[ix] := ioFeld[ix-1];
    (* Die Feldinhalte mit Ausnahme des Letzten werden um eine
      Position nach rechts verschoben *)
    ioFeld[1] := sicher
    (* Das ehemalig letzte Element wird neues erstes *)
  end;{VerschiebeZyklisch}
```

b) Eine einfache, wie in der Aufgabenstellung empfohlene Lösung sieht wie folgt aus:

```
procedure VerschiebeBis (inWert : integer;
                         var ioFeld : tFeld);
{ verschiebt den Inhalt von ioFeld solange zyklisch nach
  rechts, bis inWert vorne steht. Falls inWert nicht im Feld
  vorkommt, wird der Feldinhalt nicht verändert }

  var
  vorhanden : boolean;
  ix: tIndex;

begin
  vorhanden := false;
  ix := 0;
  while ((not vorhanden) and (ix < MAX)) do
  (* Die Schleife wird solange durchlaufen, bis inWert
     gefunden wird oder das ganze Feld durchsucht ist *)
  begin
    ix := ix + 1;
    if ioFeld[ix] = inWert then
```

```
      vorhanden := true
  end;
  if vorhanden then
  (* Wenn der Suchwert im Feld vorkommt, wird solange verschoben,
     bis sich dieser an der ersten Position befindet *)
    while ioFeld1[1] <> inWert do
      VerschiebeZyklisch(ioFeld)
end; {VerschiebeBis}
```

Auf die erste Schleife kann man verzichten, wenn man die Schleifenabbruchbedingung der zweiten Schleife erweitert: Der Zusatz `(anz < MAX)` bewirkt, dass die Schleife spätestens dann abbricht, wenn das Feld einmal vollständig verschoben wurde und sich somit wieder im Ursprungszustand befindet.

```
procedure VerschiebeBis (inWert : integer;
                         var ioFeld : tFeld);
{ verschiebt den Inhalt von ioFeld solange zyklisch nach
  rechts, bis inWert vorne steht. Falls inWert nicht im Feld
  vorkommt, wird der Feldinhalt nicht verändert }

    var
    anz : integer;

  begin
    anz := 0;
    while ((ioFeld[1] <> inWert) and (anz < MAX)) do
    begin
      VerschiebeZyklisch(ioFeld);
      anz := anz + 1
    end { while ((ioFeld[1] <> inWert) and (anz < MAX)) }
  end;{VerschiebeBis}
```

Lösung 2 (6+4 Punkte)

a)

```
procedure suchen(inSuchwert: integer; inRefAnf: tRefListe;
                                    var outPos: tRefListe);
{ Gibt mit outPos einen Zeiger auf das letzte Vorkommen von
  inSuchwert in der Liste inRefAnfang zurück }

  var
  lauf,
  pos: tRefListe;

begin
  pos := inRefAnfang;
  lauf := inRefAnfang^.next;
  while lauf <> NIL do
  (* Die Liste wird vollständig durchlaufen und dabei das *)
  (* jeweils letzte Vorkommen von inSuchwert gespeichert *)
  begin
    if lauf^.info = inSuchwert then
      pos := lauf;
    lauf := lauf^.next
  end; { while lauf <> NIL }
  outPos := pos;
end; {suchen}
```

b)

```
procedure loeschen(inSuchwert: integer;
                   var ioRefAnf: tRefListe);
{ löscht das letzte Vorkommen von inSuchwert in der Liste, deren
Anfangszeiger ioRefAnf ist}

  var lauf,
      pos: tRefListe;

begin
  suchen(inSuchwert, ioRefAnf, pos);
  if pos = ioRefAnf then (* Sonderfall: Löschen des Listenanfangs
*)
    ioRefAnf := ioRefAnf^.next
  else
  begin
    lauf := ioRefAnf;
```

```
    while lauf^.next <> pos do
    (* Finden des Vorgängers von pos *)
      lauf := lauf^.next;
    (* Das Element pos wird aus der Liste entfernt: *)
    lauf^.next := pos^.next
  end; { if pos = ioRefAnf }
  dispose(pos)
end; {loeschen}
```

Lösung 3 (10 Punkte)

```
procedure FuegeEin(inWert: integer; var ioRefAnfang: tRefListe);

  var
  lauf, neu, vor: TRefListe;

begin
  new(neu);
  neu^.info := inWert;
  if ioRefAnfang = NIL then                { Leere Liste? }
  begin
    neu^.next := NIL;
    ioRefAnfang := neu
  end
  else
  begin
    vor := ioRefAnfang;
    lauf := ioRefAnfang^.next;
    if vor^.info>inWert then               { Neues Element wird }
    begin                                  { erstes Element }
      neu^.next := vor;                      (*1*)
      ioRefAnfang := neu                     (*2*)
    end
    else
      if lauf = NIL then                   { Die Liste besteht nur aus }
      begin                                { einem Element. }
        vor^.next := neu;                    (*3*)
        neu^.next := NIL                     (*4*)
      end
      else
      begin
        while (lauf^.info < inWert) AND (lauf^.next <> NIL) do
        begin       { Liste besteht aus mindestens zwei Elementen }
          lauf := lauf^.next;                (*5*)
          vor := vor^.next                   (*6*)
        end;
        if lauf^.info >= inWert then       { innerhalb der Liste }
        begin                              { einfügen }
          neu^.next := lauf;                 (*7*)
          vor^.next := neu                   (*8*)
        end
        else
        begin
          lauf^.next := neu;                 (*9*)
          neu^.next := NIL                   (*10*)
        end
      end
  end
end;
```

Lösung 4 (6+6+2 Punkte)

a)

```
function Hoehe(inWurzel: tRefBinBaum): integer;
{ liefert die Höhe des übergebenen Baumes zurück }

  var
  HoeheL,
  HoeheR: integer;

begin
  if inWurzel = NIL then
(* Ein leerer Baum hat die Höhe 0 *)
   Hoehe := 0;
  else
  begin
(* Die Höhe ergibt sich aus dem Maximum der Höhen der linken und
rechten Teilbäume, zu dem 1 addiert wird (die Wurzel) *)
    HoeheL := Hoehe(inWurzel^.links);
    HoeheR := Hoehe(inWurzel^.rechts);
    if HoeheL > HoeheR then
      Hoehe := 1 + HoeheL
    else
      Hoehe := 1 + HoeheR
  end { if inWurzel = NIL }
end; {Hoehe}
```

b)

```
procedure Pruefen(inWurzel: tRefBinBaum;
                  var ioBalanciert: boolean);
{ setzt ioBalanciert auf false, wenn der in inWurzel übergebene
  Baum nicht balanciert ist }

begin
  if inWurzel <> NIL then
  begin
(* Sind alle Teilbaeume balanciert ? *)
    Pruefen(inWurzel^.links, ioBalanciert);
    Pruefen(inWurzel^.rechts, ioBalanciert);
(* Gilt die Balancierteigensschaft zusätzlich auch für die Wur-
zel? *)
    if abs(Hoehe(inWurzel^.links)-Hoehe(inWurzel^.rechts)) >1
then
```

```
      ioBalanciert := false
  end { if inWurzel <> NIL }
end; {Pruefen}
```

c) `ioBalanciert` muss beim ersten Aufruf den Wert `true` besitzen: Ein Baum ist nur dann balanciert, wenn all seine Teilbäume balanciert sind. Dementsprechend kann sich innerhalb eines Teilbaum-Aufrufes immer nur ein bisher balancierter Baum als unbalancierter Baum erweisen, niemals umgekehrt.