Kurs 1613 „Einführung in die imperative Programmierung“

Musterlösung zur Klausur am 21.02.2009

Aufgabe 1

a)

```
function istSortiert(var inFeld: tFeld): Boolean;
{Ergebnis ist true, genau dann wenn inFeld aufsteigend sortiert
ist, also keines der Elemente kleiner als sein Vorgänger ist.}

var i: tIndex;
    sort: boolean;

begin
  {Annahme "Feld ist sortiert"}
  sort := true;

  {Suche nach Gegenbeispiel, dazu jedes Arrayelement
  (ab dem zweiten) mit Vorgänger vergleichen}
  for i := 2 to GRENZE do
    if inFeld[i] < inFeld[i-1] then
      sort := false;

  istSortiert := sort
end;
```

b) Es ist effizienter, die Schleife abzubrechen, sobald ein erstes Gegenbeispiel, d.h. ein Element mit größerem Vorgänger, gefunden wurde. Das Feld muss dann nicht mehr bis zum Ende durchlaufen werden. (Realisiert werden kann das z.B. mit Hilfe einer While-Schleife.)

Aufgabe 2

a) Es gibt folgende drei Sonderfälle:

- `inVor1 = inVor2`: inVor1 und inVor2 zeigen auf dasselbe Listenelement.
- `(inVor1^.next = nil) or (inVor2^.next = nil)`:
  inVor1 oder inVor2 zeigt auf das letzte Listenelement, d.h. es gibt keinen zu vertauschenden Nachfolger.
- `(inVor1 = inVor2^.next) or (inVor2 = inVor1^.next)`:
  inVor1 und inVor2 zeigen auf zwei direkt aufeinander folgende Elemente, d.h. ein Element ist mit seinem Nachfolger zu vertauschen. (In diesem Fall müssten nur drei statt vier Zeigern verbogen werden.)

Anmerkung: Der Fall „Leere Liste“ kann unter der gegebenen Vorbedingung nicht eintreten, muss hier also auch nicht genannt werden. Der Fall „einelementige Liste“ ist in Fall „inVor1 und inVor2 zeigen auf dasselbe Listenelement“ enthalten und muss daher ebenfalls nicht genannt werden. Ebenso werden die Fälle der zwei- und dreielementigen Liste von obigen Sonderfällen mit abgedeckt.

b) Im Folgenden verwenden wir eine Lösung mit vier Hilfszeigern. Diese hat den Vorteil, dass die darauf folgenden Verkettungen sehr einfach und vor allem in beliebiger Reihenfolge geändert werden können.

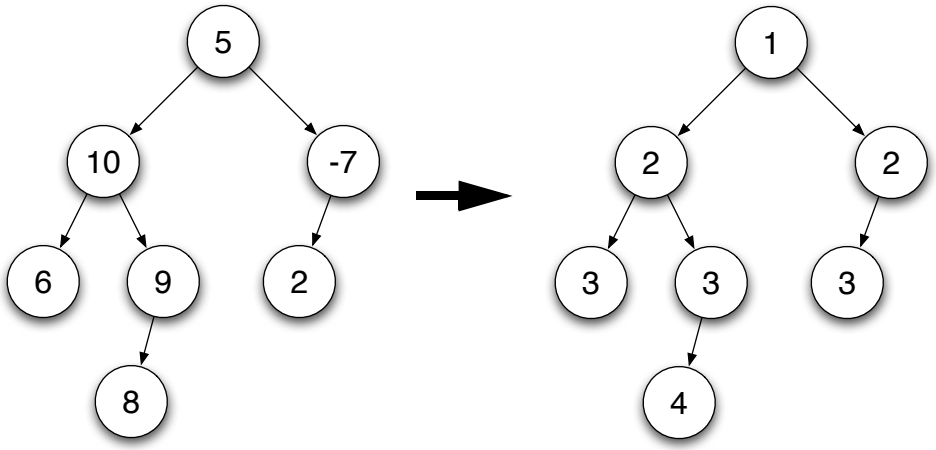
```
procedure tauschen(inVor1, inVor2:tRefElement);
{Vertauscht zwei Elemente einer Liste durch Änderung der
Verkettung. inVor1 und inVor2 sind dabei Zeiger auf die
Vorgängerelemente der zu vertauschenden Elemente.
Vorbedingung:
inVor1 und inVor2 zeigen jeweils auf ein existierendes
Element derselben Liste, jedoch nicht auf das selbe, nicht
auf das letzte und nicht auf zwei direkt aufeinander folgende
Elemente.}

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;
```

Man kann auch mit weniger Hilfszeigern auskommen, jedoch typischerweise zu Lasten der Übersichtlichkeit.

Ein Ansatz, der mit einem einzigen Hilfszeiger (dafür aber mit mehr Verkettungsänderungen teils temporärer Natur und mit gleichbleibend 8 Programmanweisungen) auskommt, wäre z.B., den Nachfolger von Vor1^ erst aus der Liste herauszutrennen und hinter dem Nachfolger von Vor2^ einzufügen, anschließend den Nachfolger von Vor2^ zu entfernen und ihn hinter Vor1^ wieder einzufügen.

Aufgabe 3

a) Die folgende Abbildung zeigt die Veränderung am Beispiel-Baum.



Allgemein wird bei Aufruf mit `inT=1` für die Wurzel in jeden Knoten (genauer in seine Komponente `t`) die Tiefe[1] des Knotens eingetragen. Existierende Werte im Baum werden dabei überschrieben.

Die Wurzel erhält also den Wert 1, ihre Nachfolgerknoten den Wert 2, deren Nachfolger den Wert 3 etc. Allgemein ist der Wert (der Komponente `t`) jedes inneren Knotens immer um eins größer als der seines Vorgängerknotens.

b) Symmetrische Reihenfolge oder inorder-Reihenfolge

c) Die Durchlaufreihenfolge ist irrelevant. Die Prozedur bearbeitet jeden Knoten isoliert, d.h. unabhängig von seinen Teilbäumen und somit insbesondere unabhängig davon, ob bzw. wann seine Teilbäume verändert werden.

d) Zur Erledigung der Aufgabe muss der gesamte Baum durchlaufen werden. Für einen solchen Baumdurchlauf wird ein Stapel benötigt, da man zu bereits besuchten Knoten zurückkehren muss, um deren zweiten Teilbaum ebenfalls durchlaufen zu können. Die rekursive Lösung ist sinvoll, weil sie den implizit vorhandenen Laufzeitstapel dafür ausnutzt, während bei einer iterativen Lösung ein Hilfsstapel explizit implementiert werden müsste.

1. Tiefe des Knotens K = Anzahl der Knoten auf dem Pfad von der Wurzel zu K.

Aufgabe 4

Wir durchlaufen den Baum in Hauptreihenfolge, d.h. wir bearbeiten zunächst den Wurzelknoten und durchlaufen danach rekursiv jeden seiner Teilbäume. Dazu wird mit einer While-Schleife die Nachfolgerliste des Wurzelknotens durchlaufen und für jedes Listenelement, also jeden Teilbaum, ein rekursiver Aufruf gestartet.

```
procedure baumNegation(inRefWurzel: tRefMultibaumKn);
{ Durchläuft den Baum und ersetzt den Info-Wert jedes Knotens
durch seinen negativen Wert.}
var l: tRefNachfListenElem;
begin
  if inRefWurzel <> nil then { Baum ist nicht leer }
  begin
    { Wurzel bearbeiten }
    inRefWurzel^.info := - inRefWurzel^.info;
    { Nachfolgerliste durchlaufen (sofern nicht leer) }
    l := inRefWurzel^.nachfListe;
    while l <> nil do
    begin
      { jeden Teilbaum ebenfalls durchlaufen }
      baumNegation(l^.teilbaumWurzel);
      l := l^.next
    end
  end
end;
```

Aufgabe 5

a) $E_3 = \{0\}$

Erläuterung: Eingabeäquivalenzklassen müssen eine Zerlegung der Menge der zulässigen Eingaben (hier $\mathbb{Z}$) bilden, d.h. sie müssen disjunkt sein und ihre Vereinigung muss wieder $\mathbb{Z}$ ergeben. Wir suchen hier demnach die Menge $E_3 = \mathbb{Z} \setminus (E_1 \cup E_2)$, also $E_3 = \{0\}$.

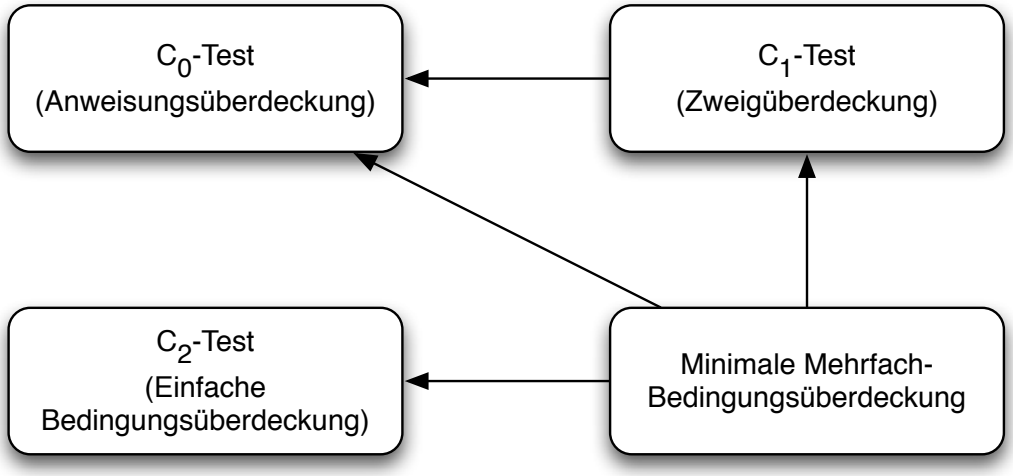
Oder anschaulich gesprochen: Die einzige ganze Zahl, die weder positiv noch negativ ist, also weder in E_1 noch E_2 enthalten ist, ist die Zahl 0, die somit allein die fehlende Äquivalenzklasse bildet.

b) $T_2 = \{ (-n, n) \mid n \in \mathbb{N}^+ \}$

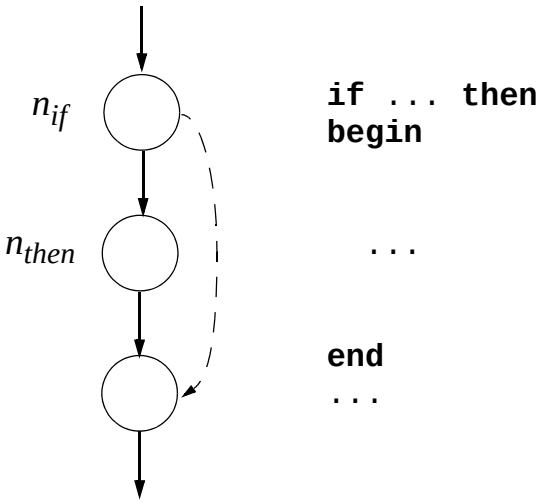
Erläuterung: Um die Eingabeäquivalenzklasse zu einem Testfall zu erweitern, ist jede einzelne Eingabe aus der Äquivalenzklasse zu einem vollständigen Testdatum zu erweitern, d.h. um die laut Spezifikation erwartete Ausgabe zu ergänzen.

Aufgabe 6

a)



b) Bei Eingabe von 15 wird eine *vollständige Anweisungsüberdeckung* erreicht, da sowohl der **`while`**- als auch der **`then`**-Block betreten und damit alle Anweisungen ausgeführt werden.
Es wird 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 (der „leere Else-Zweig“) unüberdeckt.