Lehrstuhl für Informatik 3
Rheinisch-Westfälische Technische Hochschule
Aachen
Univ.-Prof. Dr.-Ing. M. Nagl

Aachen, 22. 06. 2007

Nagl, Ranger, Wörzberger

Übungen zur Vorlesung
„Grundgebiete der Informatik 2:
Algorithmen und Programmiertechniken“

— Lösung zu Blatt 10 —

23. Aufgabe

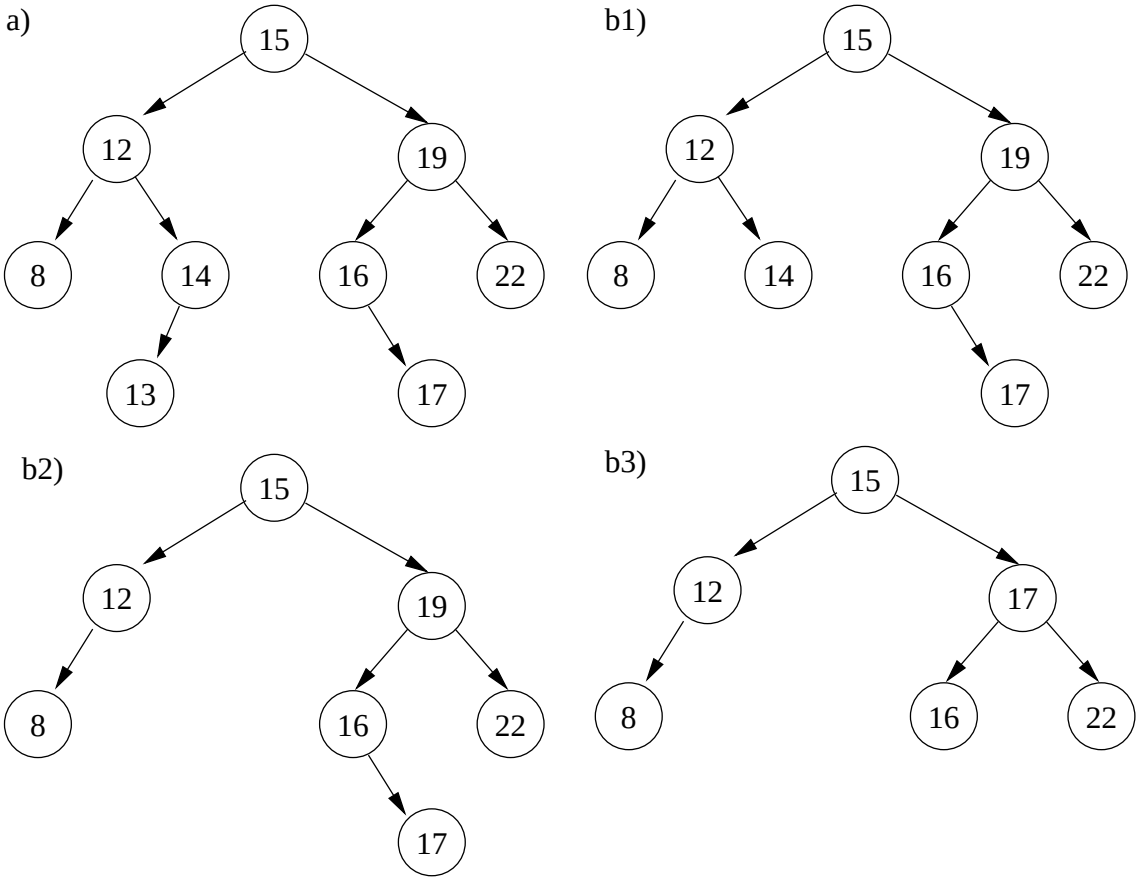
(a) Es gilt: jeder binäre Suchbaum ist ein Binärbaum, jeder Binärbaum ist ein Baum.

	Typ	**max. Auslaufgrad**	**Höhe**
1	binärer Suchbaum	2	3
2	Baum	3	3
3	kein Baum	2	-
4	binärer Baum	2	4
5	binärer Suchbaum	2	5
6	binärer Suchbaum	1	5

(b) Ein binärer Baum der Höhe n hat maximal $2^{n-1} - 1$ innere Knoten (Knoten mit Nachfolger) und 2^{n-1} Blätter, also insgesamt $2^n - 1$ Knoten. Die minimale Höhe eines binären Suchbaums zur Speicherung von m Werten ist $1 + \lfloor \log_2 m \rfloor$.

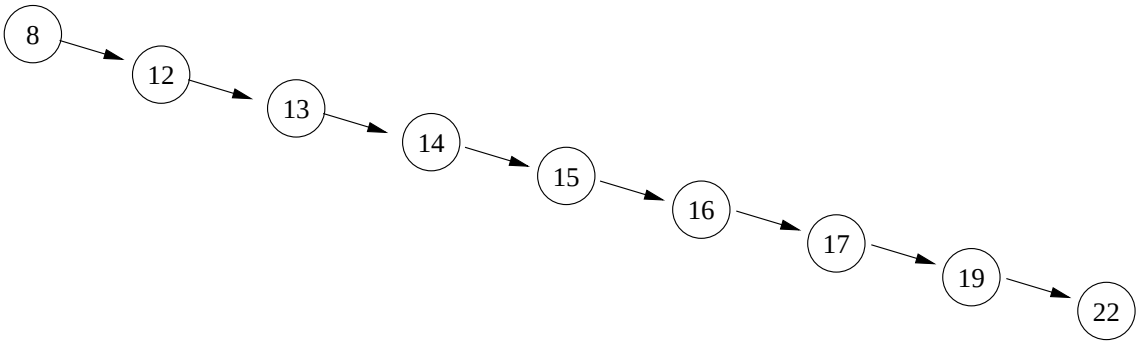
24. Aufgabe

(a) Die Höhe des Baums ist 4.



(b) Beim Löschen der 19 ersetzt in dieser Lösung das maximale Element des linken Teilbaums den gelöschten Knoten.

(c) Hier degeneriert der Suchbaum zu einer linearen Liste. Der Suchbaum aus (a) eignet sich besser zum Suchen, da er flacher ist und deswegen (im Mittel) weniger Vergleiche beim Suchen benötigt werden.



(d) `istBalanciert` verwendet eine Prozedur `istHoeheBalanciert`, die bestimmt, ob ein Teilbaum balanciert ist und außerdem seine Höhe berechnet. Diese arbeitet wiederum rekursiv, indem sie Höhe und Balanciertheit der Teilbäume berechnet. Die Hilfsfunktion `abs` aus `stdlib` berechnet lediglich den Absolutwert einer Zahl.

```
#include <istBalanciert.h>
#include <stdlib.h>             // abs(): Absolutwert einer Zahl
```

```
/* Ermittelt, ob ein Teilbaum balanciert ist, und wenn ja, welche Hoehe er hat. */
/* Vorbedingung : baum ist ein gueltiger (Teil-)baum
   Eingabe      : baum --- der auszumessende Baum
   Ausgabe      : true gdw. baum balanciert ist
                  hoehe --- die Hoehe von baum, falls er balanciert ist.
   Nachbedingung: baum und der darunterliegende (Teil-)Baum unveraendert.*/
static bool istHoeheBalanciert(BinBaumKnotenT const & baum,
                               unsigned int         & hoehe) {
   unsigned int hoeheLinks,      hoeheRechts;
   bool         balanciertLinks, balanciertRechts;

   /* Berechne Balanciertheit und Hoehen der linken und rechten Teilbaeume.
      Falls die Teilbaeume balanciert sind und sich nicht zu sehr in der
      Hoehe unterscheiden, ist der hier untersuchte (Teil-)Baum balanciert
      und die Hoehe ergibt sich aus der des hoeheren Teilbaums. */

   if (linkerAstExistiert(baum)) {
      balanciertLinks = istHoeheBalanciert(*linkerAst(baum), hoeheLinks);
   } else {
      balanciertLinks = true;
      hoeheLinks      = 0;
   }

   if (rechterAstExistiert(baum)) {
      balanciertRechts = istHoeheBalanciert(*rechterAst(baum), hoeheRechts);
   } else {
      balanciertRechts = true;
      hoeheRechts      = 0;
   }

   if (balanciertLinks && balanciertRechts && // Sind die Teilbaeume balanciert?
       abs(hoeheLinks - hoeheRechts) <= 1) { // Beide Unterbaeume balanciert, dann T balancie
                                             // wenn ihre Hoehen sich um maximal eins unterso
      /* Balanciert => noch die Hoehe ermitteln. */
      if (hoeheLinks > hoeheRechts) {
         hoehe = hoeheLinks  + 1;
      } else {
         hoehe = hoeheRechts + 1;
      }
      return true;
   } else {
      /* Nicht balanciert => die Hoehe ist uninteressant */
      return false;
   }
}

/// ======================================================================
/// Implementation der oeffentlichen Methoden
/// ======================================================================
/* Aufrufschnittstelle des Tests */
bool istBalanciert(BinBaumKnotenT const & baum) {
   /* Die interne Balance-Berechnung liefert als Zwischenergebnis die Hoehe
      in einem Ausgabeparameter.  Diese Variable reserviert Speicherplatz
      fuer diesen Parameter, wird aber in dieser Funktion nicht mehr
      benoetigt. */
   unsigned int hoehe;
   return istHoeheBalanciert(baum, hoehe);
}
```