```
fct balinsert = (store a, data d) store                    (balanciertes Einsortieren)
    if a = emptytree then cons (emptytree, d, emptytree)
  elif key(d) = key(root(a)) then cons (left(a), d, right(a))
  elif key(d) < key(root(a)) then store b = balinsert (left(a), d);
                                  if hi(b) > hi(right(a))
                                  then leftbalance (b, root(a), right(a))
                                  else cons (b, root(a), right(a))
                                  fi
                                  (Einsortieren in linken Teilbaum, vgl. Abb. S. 343)
       else store b = balinsert
            (wie linker Teilbaum)                          (rechter Teilbaum)
    fi

fct leftbalance = (store b, store aw, store ar) store:
    if hi(left(b)) ≥ hi(right(b))
    then cons (left(b), root(b), cons (right(b), aw, ar))   → vgl. Abb!
    else (d.h. rechter Teilbaum ist mehr gewachsen)
         cons (cons (left(b), root(b), left (right(b)), root(right(b))),
                                       cons (right, right(b)), aw, ar)
    fi

fct rightbalance = ... entsprechend (vgl. Buch)
```

Löschen: Aus AVL-Baum a Element mit Schlüssel k Löschen.

```
fct baldelete = (store a, key k) store
    if a = emptytree then emptytree
  elif key(root(a)) = k then delete_root(a)
  elif k < key(root(a)) then store b = baldelete (left(a), k);
                              if 1+hi(b) < hi(right(a))   Balancier-Kont.
                              then rightbalance (b, root(a), right(a))
                              else cons (b, root(a), right(a))
                              fi
       else store b = baldelete (right(a), k);
            if 1+hi(b) <
```

Das Beispiel der AVL-Bäume ist typisch für Datenstrukturen, die einer größeren Grundsorte entstammen (im Beispiel Binärbäume), aber noch zusätzliche Eigenschaften (im Beispiel: Balanciertheit + Sortiertheit) aufweisen.
Diese Eigenschaften heißen auch Datenstrukturinvarianten.
Prinzip einer Datenstrukturinvarianten:
- Alle zur Verfügung gestellten Operationen haben die Eigenschaft, dass – falls die Invariante für die Eingabeparameter gilt – diese auch für das Ergebnis gilt.

```
fct delete_root = (store a) store:
    if left(a) = emptytree then right(a)
  elif right(a) = emptytree then left(a) else data e = greatest (left(a));
                                  store b = baldelete (left(a), key(e));
                                  if 1+hi(b) < hi(right(a))
                                  then rightbalance (b, e, right(a))
                                  else cons (b, e, right(a))
                                  fi
fi
```

Komplexität des Algorithmus

[h hat O(n) → Algorithmus in der Ordnung der Anzahl Elemente →
Gesamter Algorithmus O(n) (obwohl wir O(log n) wollten)]

Diese Version von (Algorithmen auf) AVL-Bäumen hat linearen Aufwand (O(n) wobei n Anzahl der Elemente im Baum)

Idee: wir speichern im Baum zusätzlich zur jeweiligen Wurzel die Höhe Damit hi statt Funktion Selektor-Fkt.

Ergebnis dieser Prozedur:
O(log n) Algorithmen für Einfügen und Löschen!!

Sortendeklaration: sort htree = cons (htree left, data root, htree right, nat hi)

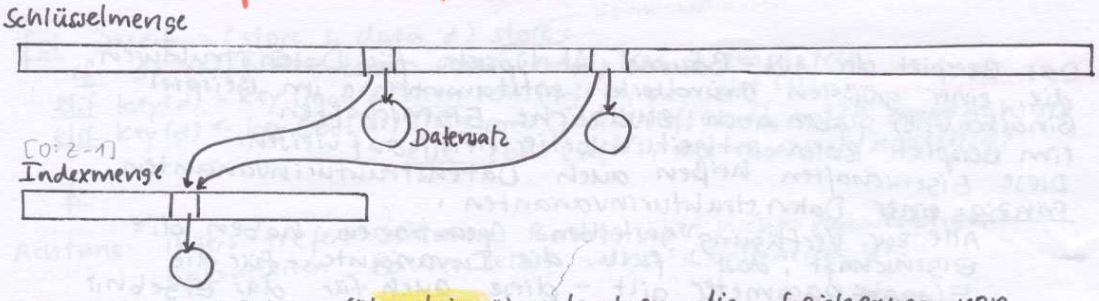
Weitere Optimierung: Statt der Höhe speichern wir eine Zahl aus {-1, 0, 1} für linker Teilbaum höher, ausgewogen, bzw. rechter Teilbaum höher.

Beim Einfügen gilt:
- Wenn einmal balanciert wird, hat der entspr. Baum die gleiche Höhe wie der ursprüngliche Baum vor dem Einfügen. Dies zeigt, dass höchstens ein Balancierschritt pro Einfügevorgang erforderlich ist.
- Dieses Argument gilt nicht für das Löschen. Hier kann im ungünstigen Fall in jedem rekursiven Aufruf ein Balancierschritt auftreten.

Analoges gilt für B-Bäume (Algorithmen im Hintergrundsp.)
AVL-Bäume Hauptspeicher
↳ log. Komplexität, geeignet f. große Datenmengen schnell!!

4.3. Streuspeicherverfahren

vgl. DS-Schubfachprinzip



Streuspeicherverfahren ("hashing") erlauben die Speicherung von Daten unter ihrem Schlüssel in einem linearen Feld der Länge t (wobei z deutlich kleiner ist als die Anzahl der Schlüssel)
Wir benötigen eine Streufunktion h: key → [0:z-1]
Die benötigten Operationen, emptystore, get, insert, delete werden nun für ein Feld realisiert:

sort store = [0:z-1] array data

Dabei nehmen wir an, dass data ein Element "empty" umfasst, das als Platzhalter dient für Feldelemente die nicht belegt sind.
empty store entspricht dem Feld, in dem alle Einträge den Wert "empty" haben. Um ein neues Element in die Streuspeichertabelle ("hash table") einzutragen, berechnen wir zum Schlüssel key(d) den Index h(key(d)). Falls das entsprechende Feldelement leer ist, erfolgt der Eintrag. Falls nicht, sprechen wir von einer Kollision und nutzen ein Verfahren zur Kollisionsauflösung.
(Kollision wie in Abb.: zwei Schlüssel durch Streufunktion auf gleichen Schlüssel der Indexmenge abgebildet → Kollisionsauflösungsverfahren).
~ Sitzplatzvergabe im Flugzeug.

Aufgaben bei Streuspeichertechnik:
- Größe des Feldes festlegen
- Wahl der Streufunktion
- Bestimmung eines Verfahrens zur Kollisionsauflösung.

Erfahrungswerte (mehr im Buch von N. Wirth)
- Tabellengröße so wählen, dass Tabelle höchstens zu 90% gefüllt ist.
- Streufunktion so wählen, dass sie gut streut (wenig Kollisionen) (hängt stark von Daten und Häufigkeit von Daten ab)
- Einfache Wahl (Seien Schlüssel Zahlen)

  $h(i) = i \bmod z$

  Dabei sollte z als Primzahl gewählt werden.
- Folgende Verfahren zur Kollisionsauflösung:
  - Offene Adressierung: Bei Kollisionen wird das kollidierende Element (Datenelement) ebenfalls im Feld gespeichert
  - Geschlossene Adressierung: Die Elemente werden in einem gesonderten Bereich gespeichert.

Das Suchen eines freien Platzes bei offener Adressierung nennen wir „Sondieren": Möglichkeiten:
- Lineares Sondieren besteht darin, dass wir in gleich langen Schritten nach einem freien Platz suchen. Betrachte
  $h(k), h(k)+j, h(k)+2j, \ldots$
- Quadratisches Sondieren:
  $h(k)+1 \quad h(k)+4 \quad h(k)+9 \quad \ldots$
  Natürlich jeweils mod z.

Erfahrungswert: Bei 90% Füllung sind im Mittel 2,56 Sondierungsschritte erforderlich. (also konstanter Zugriff)

|| SEMESTER-ENDE ||

Prof. Broy wünscht viel Erfolg bei den Prüfungen, erholsame Ferien und alles Gute fürs Hauptstudium!