FernUniversität in Hagen

Lösungsvorschläge zur Hauptklausur „1661 Datenstrukturen I"

6.8.2011

Aufgabe 1

(a)

algebra *literature*

sorts *card, literature, book, story*

ops

createStory:	*string* × *string* × *card*	→	*story*
createBook:	*card* × *string* × *story*	→	*book*
createSet:		→	*literature*
insertStory:	*book* × *story*	→	*book*
insertBook:	*literature* × *book*	→	*literature*
deleteBook	*literature* × *book*	→	*literature*
deleteStory:	*book* × *story*	→	*book*
equalBooks:	*book* × *book*	→	*bool*
equalStories:	*story* × *story*	→	*bool*
containsStory:	*book* × *story*	→	*bool*
containsBook:	*literature* × *book*	→	*bool*
findStory:	*literature* × *story*	→	*literature*

(b)

sets $card = IN$

$story = \{(titel, autor, seiten) \mid titel, autor \in string, seiten \in card\}$
$= string \times string \times card$

$book = \{(isbn, titel, l) \mid isbn \in card, titel \in string, l \subseteq story \wedge |l| > 0)\}$
$= card \times string \times (F(story) / \{\varnothing\})$

$literature = \{b \mid b \subseteq book\}$

(c)

functions:

$createStory(t, a, s) = (t, a, s)$

$createBook(i, t, s) = (i, t, \{s\})$

$createSet( ) = \{ \}$

$insertStory((t, a, l), s) = (t, a, l \cup \{s\})$

$$insertBook(a, b) = \begin{cases} a & \text{, falls } containsBook(a, b) \\ a \cup \{b\} & \text{, sonst} \end{cases}$$

$$deleteBook(a, b) = \begin{cases} a/\{b\} & \text{, falls } containsBook(a, b) \\ a & \text{, sonst} \end{cases}$$

$$deleteStory((t, a, l), s) = \begin{cases} (t, a, l/\{s\}) & \text{, falls } |l| > 1 \\ (t, a, l) & \text{, sonst} \end{cases}$$

$$equalBooks(b_1, b_2) = \begin{cases} true & \text{, falls } \pi_1(b_1) = \pi_1(b_2) \\ false & \text{, sonst} \end{cases}$$

$$equalStories(s_1, s_2) = \begin{cases} true & \text{, falls } \forall i \in \{1, 2, 3\}: \pi_i(s_1) = \pi_i(s_2) \\ false & \text{, sonst} \end{cases}$$

$$containsStory(b, s) = \begin{cases} true & \text{, falls } \exists i \in \pi_3(b): equalStories(i, s) \\ false & \text{, sonst} \end{cases}$$

$$containsBook(a, b) = \begin{cases} true & \text{, falls } \exists c \in a: equalBooks(c, b) \\ false & \text{, sonst} \end{cases}$$

$findStory(a, s) = \{ b \in a \mid containsStory(b, s)\}$

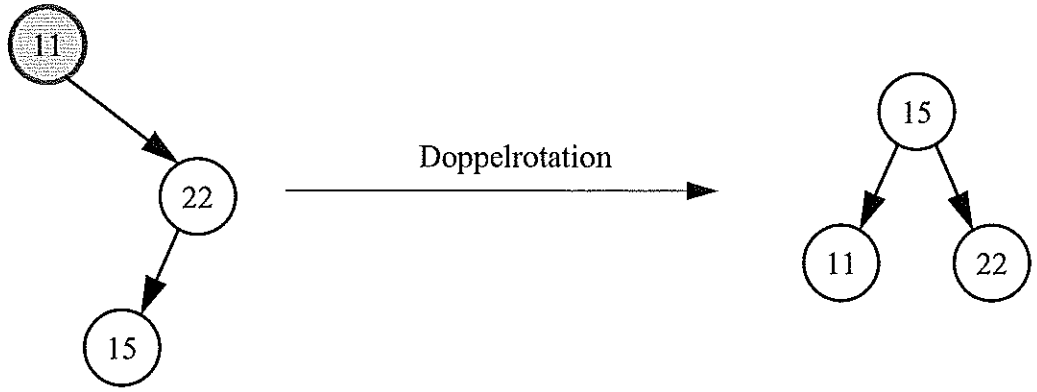
***end** literature*

Anmerkung: Alternative Lösungen sind natürlich zulässig.

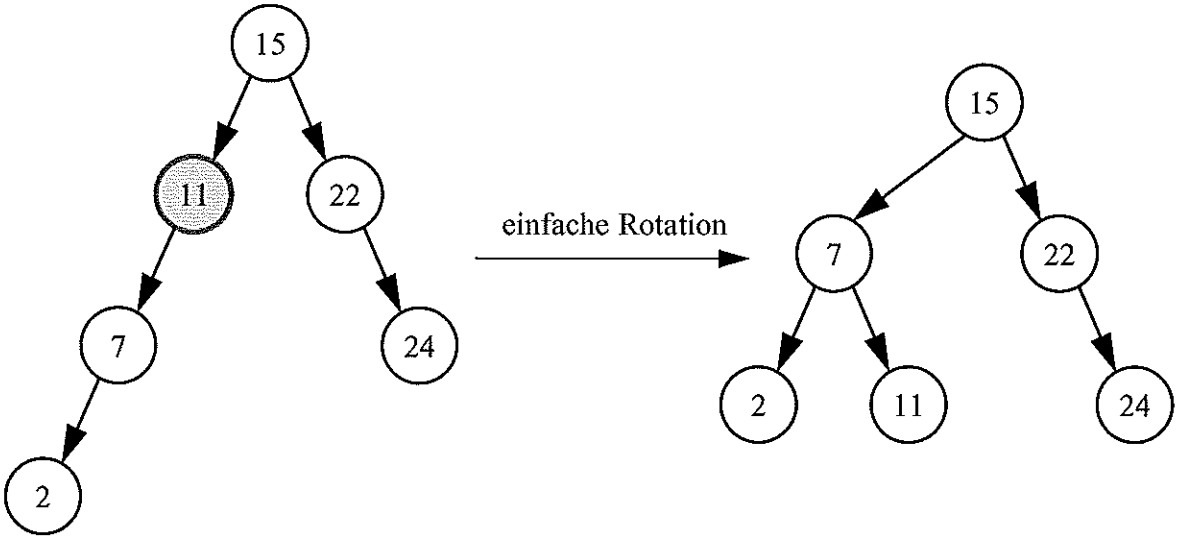
Aufgabe 2

(a)

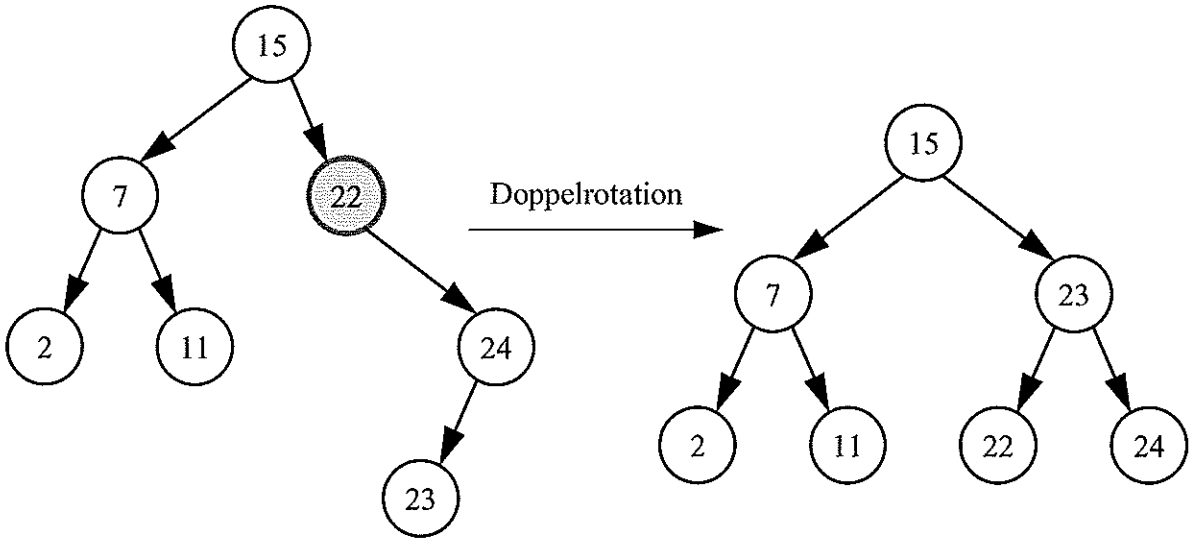
Das erste Balancieren erfolgt nach Einfügen der 15:



Die nächste Rebalancierung erfolgt nach Einfügen der 2:



Eine letzte Ausgleichsmassname wird nach Einfügen des letzten Schlüssels, 23, erforderlich:



(b)

AVL-Knoten stellen wir wie folgt dar: (T_1, k, T_2, h), wobei T_1, T_2 die Teilbäume sind, k der Knotenschüssel und h die Höhe des Baumes, der diesen Knoten zur Wurzel hat.

Ein minimal-gefüllter AVL-Baum lässt sich rekursiv definieren:

Ein minimal gefüllter AVL-Baum der Höhe h=0 ist ein Blatt ((), k, (), 0) mit einem Schlüssel k.
Ein minimal gefüllter AVL-Baum der Höhe h=1 ist entweder ((), k_1, ()), k, (), 1), $k_1<k$,
oder ((), k, (), k_2, ()), 1), $k_2>k$.

Einen minimal gefüllten AVL-Baum der Höhe $h>1$ erhält man, indem man in der Wurzel einen Schlüssel k, einen minimal gefüllten Teilbaum T_1 der Höhe h-1, wobei alle Schlüssel in $T_1<k$ sind, und einen minimal gefüllten Teilbaum T_2 der Höhe h-2, in dem alle Schlüssel $>k$ sind, (oder umgekehrte Verteilung der Höhen für T_1, T_2) kombiniert: (T_1, k, T_2, h).

(c)

Korrekturhinweis: Der Algorithmus muss eine Laufzeitkomplexität von O(n) erreichen!

algorithm *arrayToAVLtree*(*A*, *lower*, *upper*)
{ *A* ist ein Array mit aufsteigend sortierten, disjunkten Schlüsselwerten,
lower ist der Index des ersten zu berücksichtigenden Schlüsselwertes in *A*,
upper ist der Index nach dem letzten zu berücksichtigenden Schlüsselwert in *A*,
es gelte stets *lower* ≤ *upper*.
Der Algorithmus gibt einen AVL-Baum über den indizierten Bereich zurück.
}
if (*upper* = *lower*) **then**
return *empty*;
end if;
if (*upper* - *lower* = 1) **then**
return (*empty*, *A*[*lower*], *empty*, 0)
end if;
if (*upper* - *lower* = 2) **then**
return ((*empty*, *A*[*lower*], *empty*, 0), *A*[*upper* - 1], *empty*, 1);
else
var *mid*;
mid := $lower + \lceil (upper - 1 - lower)/2 \rceil$;
T1 = *arrayToAVLtree*(A, *lower*, *mid*);
T2 = *arrayToAVLtree*(*A*, *mid* + 1, *upper*);
return (*T1*, *A*[*mid*], *T2*, max{*height*(*T1*)+1, *height*(*T2*)+1});
end if;
end *arrayToAVLtree*.

Um aus einem Array A der Länge n einen binären AVL-Baum T zu erzeugen, lautet der Aufruf dann T := *arrayToAVLtree*(A, 1, n + 1).

(d)

Der Algorithmus erzeugt offensichtlich einen binären Suchbaum minimaler Höhe $h=\lfloor \log_2(n+1) \rfloor$ und somit auch einen AVL-Baum: Falls n=0, so erzeugt der Algorithmus einen (balancierten) leeren Binärbaum, falls n=1, ein (ebenfalls balanciertes) binäres Suchbaum-Blatt

(Höhe 0), bei n=2 einen (balancierten) binären Suchbaum der Höhe 1. Falls n>2, entnimmt der Algorithmus der Eingabe den Schlüssel k, der den Rest der Schlüsselmenge in in zwei sortierte Teilmengen A_1, A_2 partitioniert, deren Größen sich um höchstens ein Element unterscheiden (wobei dann A_1 diesen zusätzlichen Schlüssel enthält). Die Schlüssel in A_1 sind dabei allesamt kleiner, die in A_2 allesamt größer als k. Daher können die aus A_1, A_2 rekursiv erzeugten Teilbäume T_1, T_2 sich in der Höhe um höchstens 1 unterscheiden (wobei T_1 der ggf. höhere Teilbaum ist):

$$h(T_1) = \lfloor \log_2(\lceil (n\text{-}1)/2 \rceil) \rfloor \leq \log_2(\lceil (n\text{-}1)/2 \rceil) \leq \log_2(n/2) = \log_2(n)\text{-}1,$$
$$h(T_2) = \lfloor \log_2(\lfloor (n\text{-}1)/2 \rfloor) \rfloor \geq \log_2(\lfloor (n\text{-}1)/2 \rfloor)\text{-}1 \geq \log_2((n/2)\text{-}1)\text{-}1 \geq \log_2(n)\text{-}2,$$

also $h(T_1)$ - $h(T_2) \leq (\log_2(n)\text{-}1)$ - $(\log_2(n)\text{-}2) = 1$. Der Baum (T_1, k, T_2) ist also ein ideal balancierter binärer Suchbaum und somit auch ein AVL-Baum.

Aufgabe 3

(a)
Das Verfahren sortiert die Kisten korrekt aufsteigend nach Größe. Das Verfahren arbeitet ähnlich wie BubbleSort. Alle Elemente werden beim Durchlauf von links nach rechts mindestens einmal betrachtet. Wird dabei ein falsch einsortiertes Element gefunden, wird es solange nach links verschoben, bis es unter den bisher betrachteten Elementen seinen korrekten Platz erreicht hat. Von diesem Platz aus wird dann wieder kontrolliert, ob die „Restreihe" korrekt sortiert ist.

(b)

Am einfachsten ist eine Implementierung mit einem Array. Aber auch doppelt verkettete Listen ermöglichen das „Hin- und Herlaufen" in der Datenstruktur.

(c)

Wir benutzen im Folgenden ein Array *Kisten* als Datenstruktur. *Swap*(*Kisten*, *i*, *j*) sei wie im Kurstext definiert und tausche die Werte an den beiden Positionen des übergebenen Arrays. Der Vergleich zweier Kisten mit < liefere true, wenn die kleinere der beiden Kisten als erster Parameter übergeben wird.

```
algorithm KistenSort (Array Kisten[])
begin
  int pos = 0;
  int endpos = Kisten.Length() - 1;
  while (pos < endpos) do
    if (Kisten[pos+1] < Kisten[pos]) then
      Swap(Kisten, pos, pos + 1);
      if (pos > 0) then
        pos = pos - 1;
      else
        pos = pos + 1;
      end if
```

```
    else
      pos = pos + 1;
    endif
  end while
end algorithm
```

(d)

Im besten Fall ist die Reihe schon sortiert und das Verfahren hat eine Laufzeit von $O(n)$. Im schlimmsten Fall ist die Reihe genau umgekehrt sortiert und das Verfahren hat eine Laufzeit von $O(n^2)$.

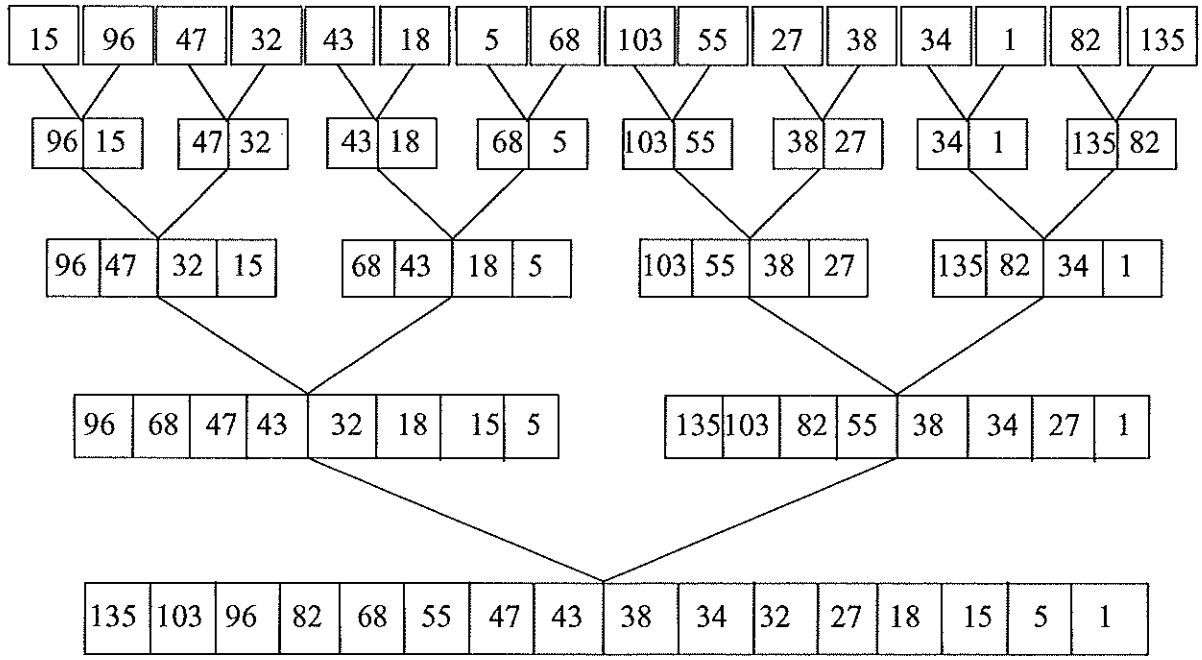
(e)

Das Verfahren ist nicht optimal. Es gibt allgemeine Sortierverfahren, die auch im worst case eine Laufzeit von $O(n \log n)$ erreichen.

(f)

Ob das Verfahren intern oder extern arbeitet, hängt von der Größe der zu verarbeitenden Menge ab. Möglich ist beides. Es handelt sich um ein allgemeines, einfaches, nicht effizientes Sortierverfahren, das in einem einzelnen Array arbeitet.

(g)



Aufgabe 4

(a)

Wird der Zielknoten zum ersten Mal erreicht oder wird ein kürzerer Pfad zum Zielknoten gefunden, so muss geprüft werden, ob die nun aktuellen Kosten unter den gegebenen Kosten liegen.

Wenn dies der Fall ist, gibt der Algorithmus true zurück. Wird der Zielknoten grün gefärbt, so sind die Kosten zu diesem Knoten höher als die vorgegebenen Kosten (sonst wäre dies vorher erkannt worden) und der Algorithmus terminiert mit der Rückgabe false. Wird erstmalig ein Knoten grün gefärbt, dessen Kosten die maximal erlaubten Kosten übersteigen, bricht der Algorithmus mit false ab.

```
algorithm reachable(G, s, t, k)
{prüft ob Knoten t von Knoten s aus in G mit höchstens Kosten in Höhe von k erreicht werden kann}
if k < 0 then return false; end if
if s=t then return true; end if
GRÜN := ∅; GELB := {s}; dist(s) := 0;
while GELB ≠ ∅ do
  wähle w ∈ GELB, so dass ∀ w' ∈ GELB: dist(w) ≤ dist(w');
  färbe w grün;
  if w = t then return false; end if
  if dist(w) > k then return false; end if
  for each w_i ∈ succ(w) do
    if w_i ∉ (GELB ∪ GRÜN) then
      färbe die Kante (w,w_i) rot;
      färbe w_i gelb;
      dist(w_i) := dist(w) + cost(w,w_i);
      if w_i = t ∧ dist(w_i) ≤ k return true; end if
    elsif w_i ∈ GELB then
      if dist(w_i) < dist(w) + cost(w,w_i) then
        färbe die Kante (w, w_i) rot;
        färbe die bisher rote Kante zu w_i gelb;
        dist(w_i) = dist(w) + cost(w,w_i) ;
        if w_i = t ∧ dist(w_i) ≤ k return true; end if
      end if
    else
      färbe (w,w_i) gelb
    end if
  end for
end while
```

(b)

Hier nutzen wir die Tatsache aus, dass die Knoten mit aufsteigendem Abstand grün gefärbt werden. Beim Grünfärben eines Knotens w prüfen wir also, ob $dist(w) > k$ ist.Wenn nicht, so wird w mit in die Ergebnismenge aufgenommen. Ist das Ergebnis dieser Überprüfung positiv, so können wir abbrechen, da die verbleibenden Knoten keine geringeren Distanzen als w aufweisen können.

Aufgabe 5 Deckblatt

Hier erhalten Sie den Punkt, wenn Sie beide Klausurdeckblätter korrekt und vollständig ausgefüllt haben, also Namen, Matrikelnummer und Adresse korrekt eingetragen und genau diejenigen Aufgaben markiert haben, die Sie auch bearbeitet haben.