Lösungsvorschläge
zur Hauptklausur
„1661 Datenstrukturen I“

12.08.2006

Aufgabe 1

(a)

algebra *sammlung*

sorts	*sammlung*, *marke*, *id*, *zustand*, *real*, *bool*			
ops	*neueSammlung*:		$\rightarrow$	*sammlung*
	neueMarke:	*id* × *real* × *zustand*	$\rightarrow$	*marke*
	MarkeEinfügen:	*sammlung* × *marke*	$\rightarrow$	*sammlung*
	MarkeEntnehmen:	*sammlung* × *marke*	$\rightarrow$	*sammlung*
	MarkeBesser:	*marke* × *marke*	$\rightarrow$	*bool*
	SammlungAuflösen:	*sammlung*	$\rightarrow$	*sammlung*
	MarkeFinden:	*sammlung* × *marke*	$\rightarrow$	*marke* $\cup \perp$
	GesamtwertBerechnen:	*sammlung*	$\rightarrow$	*real*

(b)

sets *marke* = *id* × *zustand* × *real*,
sammlung = $\mathcal{F}$(*marke*) = (*S* $\subset$ *marke* | *S* endlich),
zustand = {1…6}

functions

$$neueSammlung(\,) = \varnothing$$

$$neueMarke(id, zustand, wert) = (id, zustand, wert)$$

$$MarkeEinfügen(S,m) = \begin{cases} S \cup \{m\}, \; falls \; MarkeFinden(S,m) = \perp \\ S \setminus \{n\} \cup \{m\}, \; falls \; MarkeFinden(S,m) = n \\ \quad \wedge \; MarkeBesser(m,n) \\ S \text{ sonst} \end{cases}$$

$$MarkeEntnehmen(S,m) = S \setminus \{m\}$$

$$MarkeBesser((i_1,z_1,w_1),(i_2,z_2,w_2)) = \begin{cases} true, \text{falls } z_1 < z_2 \wedge i_1 = i_2 \\ false \text{ sonst} \end{cases}$$

$$SammlungAuflösen(S) = \varnothing$$

$$MarkeFinden(S,(i,z,w)) = \begin{cases} m, \text{falls } \exists m = (i_m, z_m, w_m) \in S : i_m = i \\ \perp \text{ sonst} \end{cases}$$

$$GesamtwertBerechnen(S) = \sum_{(i_i,z_i,w_i) \in S} w_i$$

(c)

Wir wählen als Datenstruktur beispielhaft eine verkettete Liste *L*, nämlich die, die zu der $list_2$-Algebra aus dem Kurstext gehört (andere geeignete Datenstrukturen sind ebenfalls möglich). Für den Typ *marke* legen wir einen neuen Record-Typen an:

```
type marke=     record      id : string;
                            zustand : int;
                            wert : real;
                end.

algorithm neueSammlung
L := empty.

algorithm neueMarke(id, zustand, wert)
return new marke(id, zustand, wert).

algorithm MarkeEinfügen(m)
found := MarkeFinden(m);
if found = ⊥ or MarkeBesser(m, found) then
        insert(L, last(L), m)
end if.

algorithm MarkeEntnehmen(m)
act := front(L);
while retrieve(L, act).id ≠ m.id ∧ act ≠ last+1 do
        act := next(L, act);
end while;
if retrieve(L, act).id = m then
        delete(L, act)
end if.

algorithm MarkeBesser(m1, m2)
if m1.zustand < m2.zustand then return true
else return false
end if.

algorithm SammlungAuflösen( )
L := empty.

algorithm MarkeFinden(m)
act := front(L);
while retrieve(L, act).id ≠ m.id ∧ act ≠ last+1 do
        act := next(L, act);
end while;
if retrieve(L, act).id = m.id return retrieve(L, act)
else return ⊥
end if.

algorithm GesamtwertBerechnen( )
summe := 0;
act := front(L);
```

```
while act ≠ last+1 do
    summe := summe + m.wert;
    act := next(L, act)
end while;
return summe.
```

Aufgabe 2

(a)

Um die Länge der Liste zu ermitteln und die Ausgangsliste in die beiden Teillisten zu kopieren fällt Zeitaufwand an, der linear zur Länge der übergebenen Liste ist. Somit ist die Funktion $f(n)$, die in der Laufzeitanalyse von Mergesort verwendet wurde, ebenfalls linear und die Laufzeit von Mergesort beträgt weiterhin $O(n \log n)$.

Für eine Liste der Länge 1 werden keine weiteren Listen erzeugt. Somit wird nur Speicherplatz für das ursprüngliche Listenelement benötigt, welcher laut Aufgabenstellung nicht berücksichtigt wird. Bei größeren Listen wird durch das Kopieren der Liste in zwei Teillisten Platz in der Größe der urspünglichen Liste gebraucht. Zusätzlich wird lediglich der Speicherplatzbedarf für einen Mergesort-Aufruf benötigt, da die dort erzeugten Listen ja vor dem zweiten Aufruf bereits wieder freigegeben wurden. Formal ergibt sich:

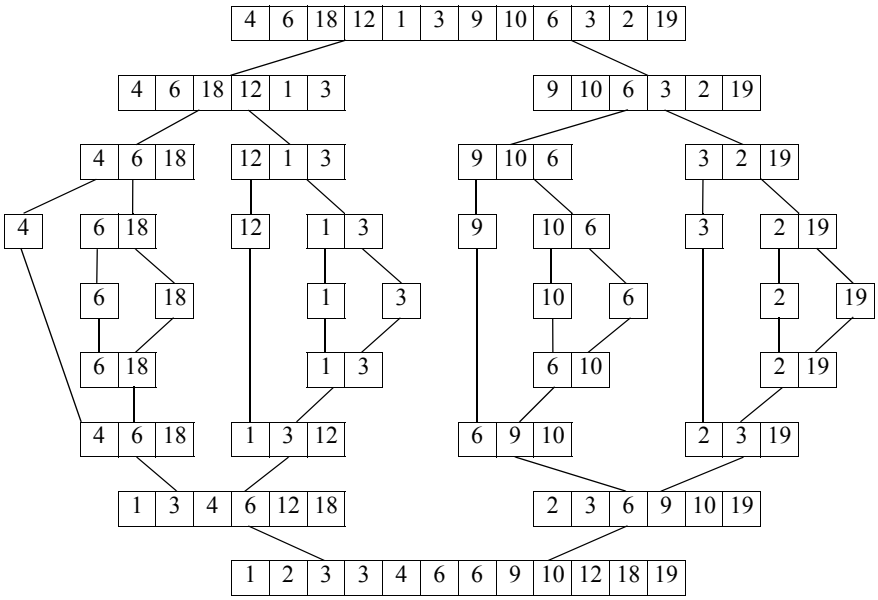
$S(n) = 0$ für $n = 1$

$S(n) = n + S(n/2)$ für $n>1$

Wir vermuten: $S(n) = 2n - 2$

Induktionsschritt : $S(2n) = 2n + S(n) = 2n + 2n - 2 = 2(2n) - 2$

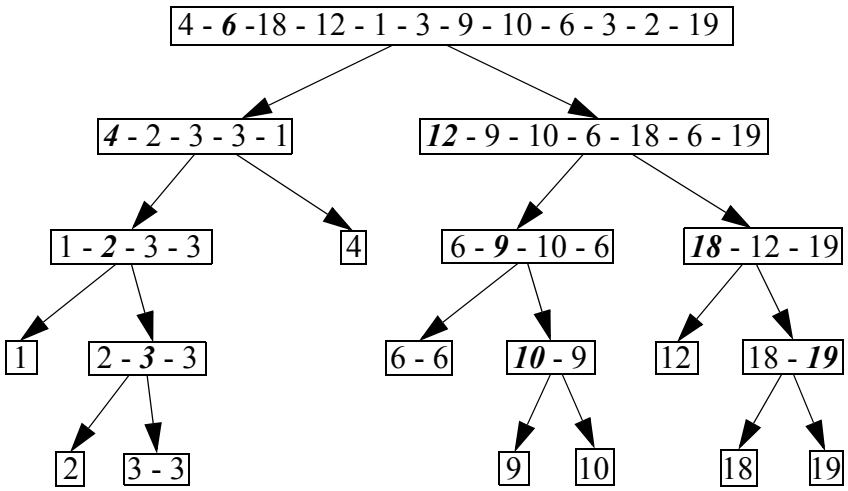
(b)



(c)

Wird die zu sortierende Menge in mehr als zwei Teilmengen aufgespalten, wird der entsprechende Aufrufbaum flacher. Divide- und Merge-Schritt können zusammen trotzdem in linearer Zeit durchgeführt werden. Wird die Menge in x Teile gespalten, so wird die Laufzeit auf $O(n \log_x n)$ reduziert.
Da jedoch gilt: $\log_x n = \log_2 n / \log_2 x$, entspricht dies der gewöhnlichen Laufzeit von Mergesort.

(d) Die gegebene Folge wird wie folgt aufgespalten:



Aufgabe 3

(a)
algorithm *decompose*(*M*, *A*)
{ *M*: Menge, die zerlegt werden soll.
A: Menge von Paaren aus *M*. Jedes Paar repräsentiert eine Äquivalenzanweisung.}

n := 1;
p := *empty*(); /* Erzeuge eine leere Partition. */

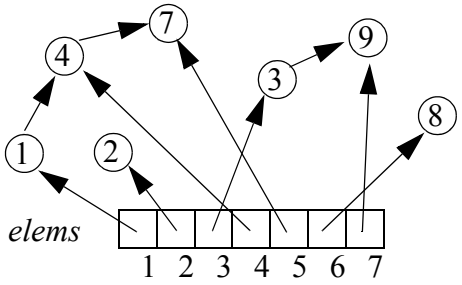
/* Erzeuge zunächst eine Partition, in der jedes Element aus *M* eine Komponente ist. */
for all $m \in M$ **do**
p := *addcomp*(*p*, *n*, *m*);
n := *n* + 1;
end for;

/* Verschmelze alle Komponenten, die äquivalent sind. */
for all $(a, b) \in A$ **do**
n_1 := *find*(*p*, *a*);
n_2 := *find*(*p*, *b*);
p := *merge*(*p*, n_1, n_2);
end for;

return *p*.

(b)

Die Äquivalenzklassen lauten {1, 4, 7}, {2}, {3, 9}, {8}. Damit erhält man die folgende Struktur:



Hierbei ist zu erwähnen, daß es eine bijektive Abbildung f: $M \rightarrow \{1, ..., |M|\}$ geben muß, damit jedes Element aus M durch einen Index im Array *elems* identifiziert werden kann. f entspricht in unserem Fall der Zuordnung

$1\rightarrow1, 2\rightarrow2, 3\rightarrow3, 4\rightarrow4, 5\rightarrow7, 6\rightarrow8, 7\rightarrow9$

Andere Zuordnungen (insgesamt gibt es 7!) sind natürlich ebenfalls möglich.

(c)

Bei der Pfadkompression wird längs des Pfades vom gesuchten Element bis zum Wurzelelement der Komponente der Zeiger auf den Vater so abgeändert, daß er danach direkt auf die Wurzel der Komponente zeigt. Um dies zu implementieren, müssen während des Passieren des Pfades Zeiger auf die besuchten Knoten gespeichert werden.

```
algorithm find(p, e)
{ p: partition einer Menge M.
  e: ein Element aus der in p dargestellten Menge. }

S := ∅;
next := p.elems[f(e)]; /* Setze einen Zeiger auf das gesuchte Element. */

while next↑.father ≠ nil then
  /* next zeigt nicht auf den Wurzelknoten! */
  S := S ∪ next;
  next = next↑.father;
end while;

for all s ∈ S do
  s↑.father = next;
end for.
```