Lösungsvorschläge
zur Hauptklausur
„1662/1663 Datenstrukturen“
12.08.2006

Aufgabe 1

(a)

algebra *sammlung*
sorts *sammlung*, *marke*, *id*, *zustand*, *real*, *bool*
ops

neueSammlung:		→	*sammlung*
neueMarke:	*id* × *real* × *zustand*	→	*marke*
MarkeEinfügen:	*sammlung* × *marke*	→	*sammlung*
MarkeEntnehmen:	*sammlung* × *marke*	→	*sammlung*
MarkeBesser:	*marke* × *marke*	→	*bool*
SammlungAuflösen:	*sammlung*	→	*sammlung*
MarkeFinden:	*sammlung* × *marke*	→	*marke* ∪ ⊥
GesamtwertBerechnen:	*sammlung*	→	*real*

(b)
sets $marke = id \times zustand \times real$,
$sammlung = \mathcal{F}(marke) = (S \subset marke \mid S \text{ endlich})$,
$zustand = \{1 \ldots 6\}$

functions

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

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

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

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

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

$$SammlungAuflösen(S) = \varnothing$$

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

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

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:

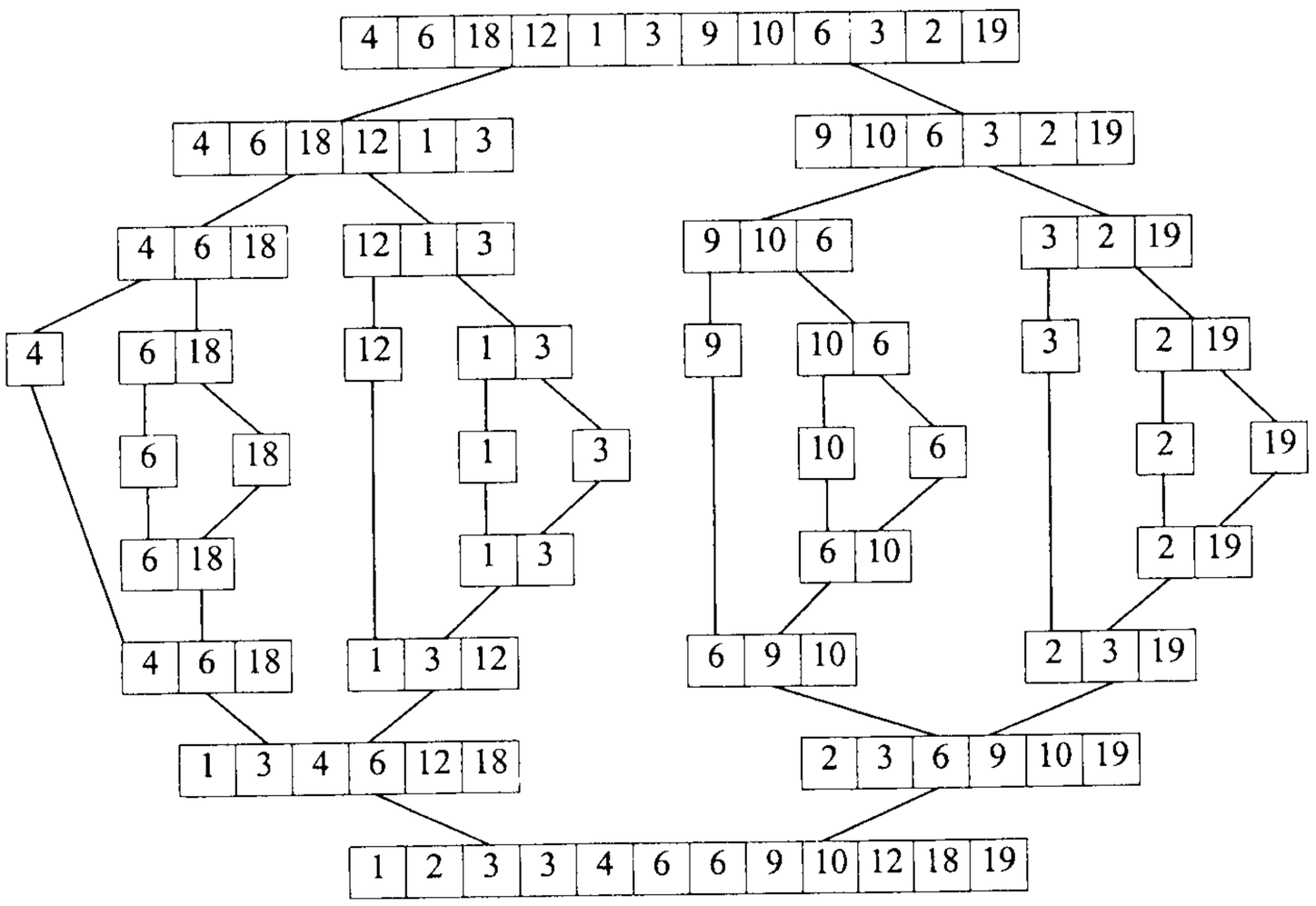
$S(n) = 0$ für $n = 1$

$S(n) = n + S(n/2)$ 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.

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. */

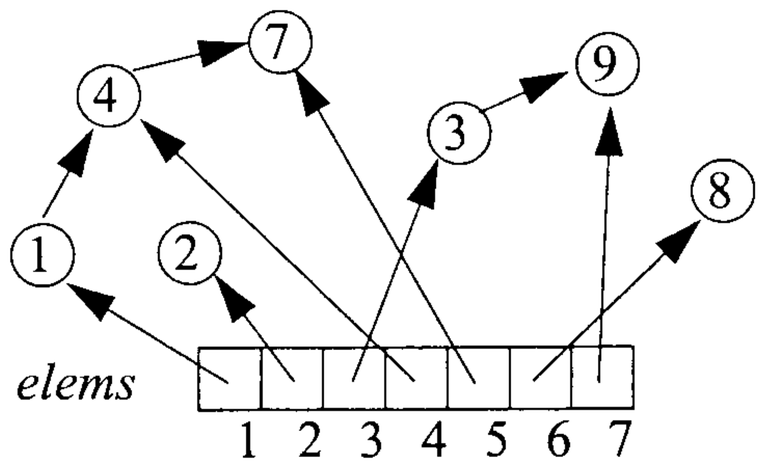
/* Erzeuge zunächst eine Partition, in der jedes Element aus *M* eine Komponente ist. */
for all $m \in M$ **do**
  p := *addcomp*(*p*, *n*, *m*);
  n := *n* + 1;
end for;

/* 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

$1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 7, 6 \rightarrow 8, 7 \rightarrow 9$

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

Aufgabe 4

(a)

Zunächst bauen wir die Sweep-Event-Struktur auf, die die linken und rechten Kanten der Rechtecke nach x-Koordinaten sortiert enthält. Da lediglich die obere y-Koordinate interessant ist, speichern wir die Einträge in der Form (x, *left*, y_{max}, *rid*) bzw. (x, *right*, y_{max}, *rid*), wobei *rid* der entsprechende Rechteckname ist. Liegen mehrere Rechteckkanten auf der gleichen x-Koordinate, werden zunächst die linken Kanten und danach die rechten Kanten eingefügt.
In der Sweepline-Status-Struktur werden Einträge in der Form (y, *rid*) gespeichert.

Es bezeichne y_{stored_max} den maximal in der Sweepline-Status-Struktur gespeicherten y-Wert oder 0, falls diese kein Element enthält. x_{last} bezeichne den x-Wert des Events vor dem gerade bearbeiteten Event oder den minimalen vorhandenen x-Wert, falls das erste Event bearbeitet wird.

Während des Sweeps sind folgende Aktionen durchzuführen:

linke Rechteckkante (x, *left*, y_{max}, r) wird angetroffen:

- falls $x_{last} < x$ gib (x_{last}, y_{stored_max}, x, y_{stored_max}) aus
  gibt horizontales Segment aus, falls Änderung in x-Richtung erfolgt ist
- falls $y_{max} > y_{stored\_max}$ ist, gib (x, y_{stored_max}, x, y_{max}) aus
  gibt vertikales Segment aus, falls neue höchste y-Koordinate gefunden wurde
- füge (y_{max}, r) in die Sweepline-Status-Struktur ein

rechte Rechteckkante (x, *right*, y_{max}, r) wird angetroffen:

- falls $x_{last} < x$ gib (x_{last}, y_{stored_max}, x, y_{stored_max}) aus
  gibt horizontales Segment aus, falls Änderung in x-Richtung erfolgt ist
- lösche (y_{max}, r) aus der Sweepline-Status-Struktur
- falls $y_{max} > y_{stored\_max}$ ist, gb (x, y_{max}, x, y_{stored_max}) aus
  höchste y-Koordinate ist geringer als vor Löschen dieses Segments, daher wird ein vertikales Segment gezeichnet; an dieser Position können keine weiteren „Erhöhungen“ des maximalen y-Wertes auftreten, da alle linken Segmente aufgrund der Einfügereihenfolge (siehe oben) vor den rechten Segmenten bearbeitet werden

(b)

Die Sweepline-Status-Struktur muß das Einfügen und Löschen gegebener Elemente, sowie die Suche nach dem maximal gespeicherten y-Wert effizient unterstützen. Für die Implementierung eignen sich balancierte Suchbäume, z.B. AVL-Bäume oder B-Bäume.

(c)

Sei n die Anzahl der Rechtecke der zugrundeliegenden Szene. Der Aufbau der Sweep-Event-Struktur benötigt $O(n \log n)$ Zeit, da die $2n$ Events sortiert werden müssen. Für jedes Event muß das aktuelle gespeicherte Maximum gefunden werden und jeweils ein Eintrag aus der Sweepline-Status-Struktur entfernt oder in diese eingefügt werden, was beides in $O(\log m)$ Zeit geschehen kann, wobei m die Anzahl der gerade in der Sweepline-Status-Struktur gespeicherten Elemente ist. Da $m \leq n$ gilt, dauert der Sweep insgesamt maximal $O(n \log n)$. Damit liegt der Gesamtzeitbedarf ebenfalls bei $O(n \log n)$.

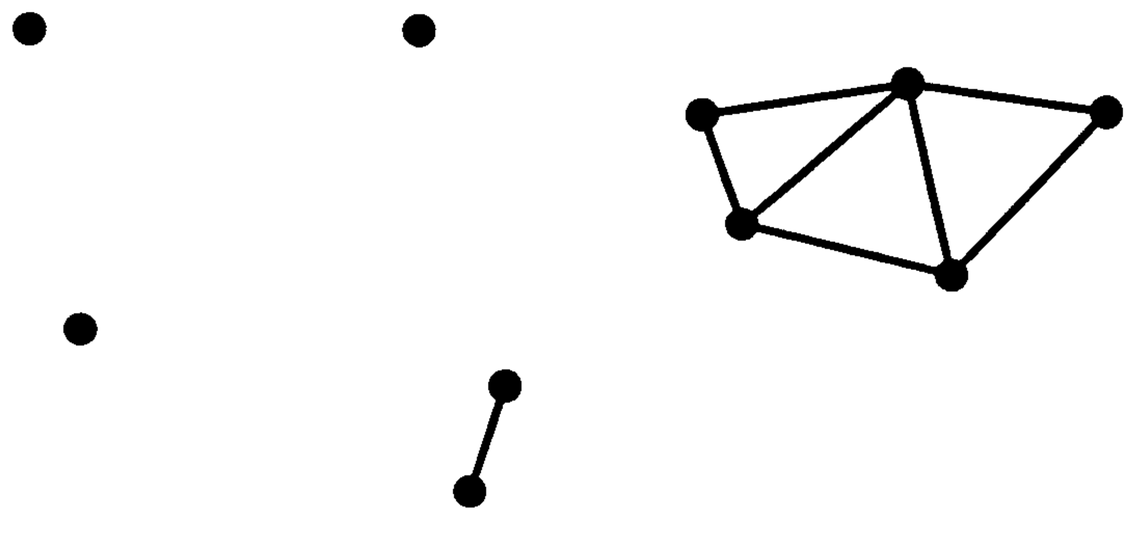
Aufgabe 5

(a)

Die Anzahl der Inseln kann mit dem Algorithmus für die Bestimmung der starken Komponenten in Graphen berechnet werden. Die Anzahl der Komponenten entspricht der Anzahl der Inseln. Die Kosten hierfür sind $O(n + e)$

(b)

Wenn G nicht benutzt werden darf, kann die Lösung ermittelt werden, indem ein Graph $G^* = (V^*, E^*)$ aufgebaut wird, der die Nachbarschaftsbeziehung der Teilgraphen ausnutzt. Die Knotenmenge V^* besteht aus der Menge der Teilgraphen, während die Kantenmenge E^* eine Kante für jedes Paar benachbarter Teilgraphen enthält. Die Nachbarschaftsbeziehung läßt sich über gleiche Kanten in den Teilgraphen prüfen. G^* sieht wie folgt aus (G ist in hellgrau gezeichnet):



Die Anzahl der Inseln ergibt sich gemäß Aufgabe (a) auf G*.

(c)

Um die Länge der Küste berechnen zu können, müssen zunächst die Kanten im Inneren von G, also die, die keine Küsten beschreiben, eliminiert werden. Dies kann über einen paarweisen Vergleich der Kanten der Teilgraphen erfolgen. Wird eine doppelte Kante gefunden, so kann sie aus G gelöscht werden. Am Ende bleiben nur noch Küstenkanten übrig, deren Länge aufsummiert werden kann. Der folgende Algorithmus beschreibt das Verfahren. Sei T die Menge der Teilgraphen.

```
algorithm Küstenberechnung (G, T)
{G=(V,E) ist der Graph und T ist die Menge der Teilgraphen.}
for all t_i ∈ T do
        for all t_j ∈ T, i ≠ j do
                bestimme D = Menge der doppelten Kanten in t_i, t_j;
                G := (V, E \ D)
        end for
end for;
summe := 0;
for all e ∈ E do
        summe := summe + e.weight;
end for;
return summe.
```

(d)

Wenn der Graph G^* aus Aufgabenteil (b) bereits berechnet ist, ergibt sich die Anzahl der Nachbarn direkt aus dem Grad eines jeden Knotens in G^*. Ein Algorithmus ist folglich nicht notwendig.

Aufgabe 6

(a)

Insgesamt sind vier Fälle zu unterscheiden, da in der Wurzel zwischen 1-4 Schlüssel untergebracht werden können. In den Knoten unterhalb der Wurzel sind dann jeweils vier Schlüssel enthalten. Die folgende Tabelle führt die daraus resultierenden Konfigurationen und Schlüsselanzahlen für Bäume der Höhe eins auf:

#Schlüssel in der Wurzel	#Knoten auf Ebene 1	#Schlüssel insgesamt
1	2	1+8 = 9
2	3	2+12 = 14
3	4	3+16 = 19
4	5	4+20 = 24

Auf jeder weiteren Ebene kommen nun für jeden Knoten der vorherigen Ebene 5 weitere hinzu, damit ergeben sich für Bäume mit k Schlüsseln in der Wurzel und n Ebenen folgende Schlüsselanzahlen:

$s(n) = 1 + 8\cdot(5^0 + 5 + ... + 5^{n-1})$, falls $k = 1$,

$s(n) = 2 + 12\cdot(5^0 + 5 + ... + 5^{n-1})$, falls $k = 2$,

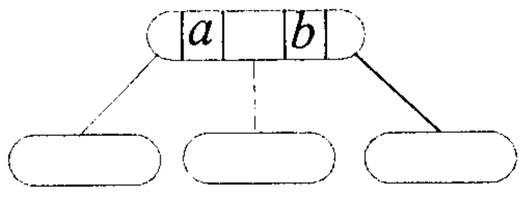
$s(n) = 3 + 16\cdot(5^0 + 5 + ... + 5^{n-1})$, falls $k = 3$,

$s(n) = 3 + 20\cdot(5^0 + 5 + ... + 5^{n-1})$, falls $k = 3$.

Kompakter kann man dies auch schreiben als $s(k, n) = k + 4\cdot(k+1)\cdot(5^0 + 5 + ... + 5^{n-1})$.

(b)

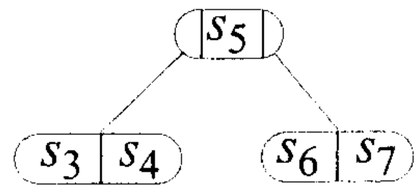
Damit alle Knoten unterhalb der Wurzel zu 100% gefüllt sind kann der B-Baum nur folgende Struktur haben:



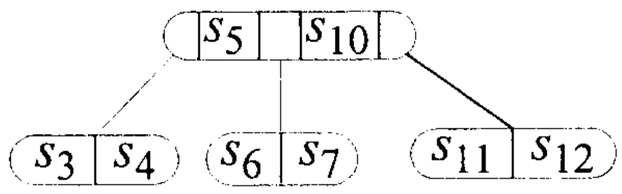
Es gilt also nun, a und b zu bestimmen und dafür zu sorgen, daß bei *split*-Operationen diese in die Wurzel gelangen. Sortiert man die einzufügende Folge, so erhält man eine geordnete Folge s_i mit

$s_1, s_2, s_3, s_4, \mathbf{s_5}, s_6, s_7, s_8, s_9, \mathbf{s_{10}}, s_{11}, s_{12}, s_{13}, s_{14}$

Zwischen bzw. neben den Zahlen s_5 und s_{10} sind jeweils genau 4 andere Zahlen, daher muß der resultierende B-Baum diese beiden Schlüssel in der Wurzel aufnehmen. Wir beginnen also mit 5 Zahlen, deren Mittelwert s_5 ist, z.B, s_3, s_4, s_5, s_6, s_7, nach einer *split*-Operation sieht der Baum dann folgendermaßen aus:



Als nächstes müßen wir dafür sorgen, daß das rechte Blatt aufgespalten wird und s_{10} in die Wurzel gelangt. Dazu fügen wir s_{10}, s_{11}, s_{12} ein und erhalten:



Die restlichen Schüssel passen nun problemlos in die Blätter und können in beliebiger Reihenfolge eingefügt werden.