Lösungsvorschläge
zur Klausur
„1661 Datenstrukturen I“

23.09.2006

Aufgabe 1

(a)

algebra *pointset*
sorts *pointset, point, real, int, bool*
ops

empty:		$\rightarrow$	*pointset*
newPoint:	*real* × *real*	$\rightarrow$	*point*
insert:	*pointset* × *point*	$\rightarrow$	*pointset*
delete:	*pointset* × *point*	$\rightarrow$	*pointset*
member:	*pointset* × *point*	$\rightarrow$	*bool*
isEmpty:	*pointset*	$\rightarrow$	*bool*
horizontalSize:	*pointset*	$\rightarrow$	*real*
verticalSize:	*pointset*	$\rightarrow$	*real*
quadrant:	*pointset* × *point* × *int*	$\rightarrow$	*pointset*
windowQuery:	*pointset* × *point* × *point*	$\rightarrow$	*pointset*
circleQuery:	*pointset* × *point* × *real*	$\rightarrow$	*pointset*

(b)

sets $point = real \times real$,
$points = \mathcal{F}(point) = \{P \subset point \mid P \text{ endlich}\}$

functions

$empty() = \emptyset$

$newPoint(x, y) = (x, y)$

$insert(P, p) = P \cup \{p\}$

$delete(P, p) = P \setminus \{p\}$

$$find(P, p) = \begin{cases} true, \text{falls } p \in P \\ false, \text{sonst} \end{cases}$$

$isEmpty(P) = (P = \emptyset)$

$horizontalSize(P) = | p_r.x - p_l.x |$, *mit*

$p_r \in P : \forall p_i \in P : p_r.x \geq p_i.x$

$p_l \in P : \forall p_i \in P : p_l.x \leq p_i.x$

$verticalSize(P) = | p_t.y - p_b.y |$, *mit*

$p_t \in P : \forall p_i \in P : p_t.y \geq p_i.y$

$p_b \in P : \forall p_i \in P : p_b.y \leq p_i.y$

$$quadrant(P, p, q) = \begin{cases} \{p_i \in P \mid p_i.x > p.x \wedge p_i.y > p.y\}, \text{falls } q = 1 \\ \{p_i \in P \mid p_i.x < p.x \wedge p_i.y > p.y\}, \text{falls } q = 2 \\ \{p_i \in P \mid p_i.x < p.x \wedge p_i.y < p.y\}, \text{falls } q = 3 \\ \{p_i \in P \mid p_i.x > p.x \wedge p_i.y < p.y\}, \text{falls } q = 4 \\ \emptyset \; sonst \end{cases}$$

$windowQuery(P, p_{tl}, p_{br}) = \{p_i \in P \mid p_{tl}.x < p_i.x < p_{br}.x \wedge p_{tl}.y < p_i.y < p_{br}.y\}$

$circleQuery(P, p, r) = \{p_i \in P \mid dist(p_i, p) < r\}$

(c)

Die geänderten Komponenten der Algebra sehen nun wie folgt aus:

sorts *pointset, point, id, real, int, bool*

ops				
	newPoint:	*real* × *real* × *id*	→	*point*
	delete:	*pointset* × *id*	→	*pointset*
	member:	*pointset* × *id*	→	*bool*
	quadrant:	*pointset* × *id* × *int*	→	*pointset*
	windowQuery:	*pointset* × *id* × *id*	→	*pointset*
	circleQuery:	*pointset* × *id* × *real*	→	*pointset*

sets $point = real \times real \times ID$

functions

$$newPoint(x, y, id) = (x, y, id)$$

$$insert(P, p) = \begin{cases} P \cup \{p\}, \text{ falls } \forall p_i \in P : p_i.id \neq p.id \\ P \text{ sonst} \end{cases}$$

$$delete(P, id) = \begin{cases} P \setminus \{p\}, \text{ falls } \exists p \in P : p.id = id \\ P \text{ sonst} \end{cases}$$

$$find(P, id) = \begin{cases} true, \text{ falls } \exists p \in P : p.id = id \\ false \text{ sonst} \end{cases}$$

$$quadrant(P, id, q) = \begin{cases} \{p_i \in P \mid p_i.x > p.x \wedge p_i.y > p.y\}, \text{ falls } q = 1 \wedge \exists p \in P : p.id = id \\ \{p_i \in P \mid p_i.x < p.x \wedge p_i.y > p.y\}, \text{ falls } q = 2 \wedge \exists p \in P : p.id = id \\ \{p_i \in P \mid p_i.x < p.x \wedge p_i.y < p.y\}, \text{ falls } q = 3 \wedge \exists p \in P : p.id = id \\ \{p_i \in P \mid p_i.x > p.x \wedge p_i.y < p.y\}, \text{ falls } q = 4 \wedge \exists p \in P : p.id = id \\ \varnothing \; sonst \end{cases}$$

$$windowQuery(P, id_1, id_2) = \begin{cases} \{p_i \in P \mid p_{tl}.x < p_i.x < p_{br}.x \wedge p_{tl}.y < p_i.y < p_{br}.y\}, \\ \qquad falls \; \exists p_{tl} \in P : p_{tl}.id = id_1 \wedge \exists p_{br} \in P : p_{br}.id = id_2 \\ \varnothing \text{ sonst} \end{cases}$$

$$circleQuery(P, id, r) = \begin{cases} \{p_i \in P \mid dist(p_i, p) < r\}, \text{ falls } \exists p \in P : p.id = id \\ \varnothing \text{ sonst} \end{cases}$$

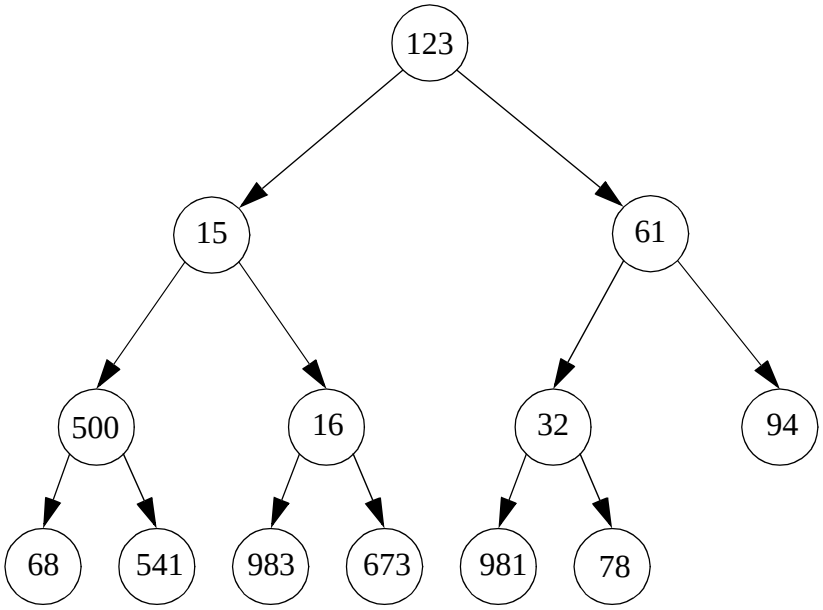
Aufgabe 2

(a)

Heapsort muß zunächst die Länge der übergebenen Liste *n* bestimmen, was durch einen linearen Durchlauf erfolgt. Beim Sortieren unterscheidet sich lediglich der Zugriff auf die Elemente der zu sortierenden Menge im Gegensatz zum normalen Arrayzugriff. Um auf ein Element (lesend oder schreibend) zuzugreifen, durchläuft man die Liste vom Beginn oder der aktuellen Position aus, bis die gewünschte Position erreicht ist.
Aufgrund des teilweisen Durchlaufs durch die Liste, um ein gewünschtes Element anzusprechen, benötigt ein *reheap* bei dieser Darstellung lineare Zeit. Damit erhöht sich der Gesamtzeitaufwand auf $O(n^2)$. Also eignet sich Heapsort nicht, um solche Listen direkt zu sortieren.

(b)

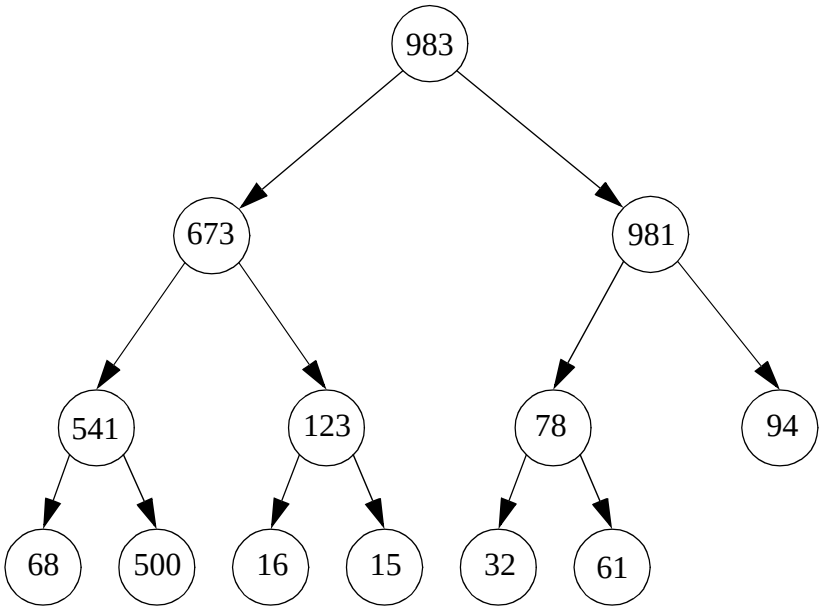
Der Ausgangsbaum sieht wie folgt aus:



Da die Folge in aufsteigender Reihenfolge sortiert werden soll, muß zunächst ein Maximum-Heap aufgebaut werden. Die Pfade des Einsinkens sind:

$32 \rightarrow 981$
$16 \rightarrow 983$
$500 \rightarrow 541$
$61 \rightarrow 981 \rightarrow 78$
$15 \rightarrow 983 \rightarrow 673$
$123 \rightarrow 983 \rightarrow 673$

Der resultierende Heap wird wie folgt als Baum dargestellt:



Wir tauschen die 983 mit der 61 und lassen die 61 einsinken:

983 ↔ 61; 61 → 981 → 94

Die verbleibenden Aktionen sind:

981 ↔ 32; 32 → 673 → 541 → 500
673 ↔ 15; 15 → 541 → 500 → 68
541 ↔ 16; 16 → 500 → 123
500 ↔ 32; 32 → 123 → 68
123 ↔ 15; 15 → 94 → 78
94 ↔ 61; 61 → 78
78 ↔ 15; 15 → 68 → 32
68 ↔ 16; 16 → 61
61 ↔ 15; 15 → 32
32 ↔ 16
16 ↔ 15

Damit endet das Verfahren.

(c)

```
algorithm radixsort(s : list)
{s: Liste der zu sortierenden Strings}
/* bestimme die Länge des längsten Strings */
s.first();
len := 0;
while not s.onEnd() do
  if length(s.get())>len then len := length(s.get()) end if;
  s.next();
end while
B : array[0..1][0..127] of list;
B[1][0] := s;
for i := len downto 0 do
  for b_1 := 0 to 127 do // initialisiere die Zielbehälter
    B[(len-i)%2][b_1] := emptyList()
  end for;
  for b := 0 to 127 do // durchlaufe alle Behälter
    // durchlaufe die enthaltene Liste und teile diese auf neue Behälter auf
    B[(len-i+1)%2][b].first();
    while not B[(len-i+1)%2)][b].onEnd() do
      str := B[(len-i+1)%2)][b].get();
      B[(len-i+1)%2)][b].next();
      B[(len-i)%2)][str.get(i)].append(str)
    end while
  end for
end for
// füge sortierte Folge zusammen
res := emptyList();
```

```
for b := 0 to 127 do
    B[(len+1)%2][b].first();
    while not B[(len+1)%2][b].onEnd() do
        res.append(B[(len+1)%2][b].get());
        B[(len+1)%2][b].next()
    end while
end for;
return res.
```

Um das Zerstören der gerade zu verarbeitenden Listen zu vermeiden, verwenden wir zwei Behältermengen (dargestellt als zweidimensionales Array), die abwechselnd mit Hilfe der *modulo*-Operation (%) angesprochen werden.

Da die Anzahl der Behälter konstant ist, fließt diese nicht in die Laufzeitbetrachtungen ein. Damit ist der zeitkritische Teil die verschachtelte Schleife, die jeden String *len* mal bearbeitet. Insgesamt ergibt sich eine Laufzeit von O(*len***n*), wobei *len* die Länge des längsten Strings und *n* die Anzahl der zu sortierenden Strings darstellt. Dieses Verfahren ist (abgesehen von konstanten Faktoren) anderen Verfahren vorzuziehen, falls für die zu verarbeitenden Daten gilt: *len* < log(*n*) also die Länge der Strings im Verhältnis zur Anzahl sehr gering ist.

(d)

Im ersten Durchlauf werden die Zahlen wie folgt in die Behälter eingefügt (leere Behälter sind nicht dargestellt):

B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_8
500	61	32	123	94	15	16	68
	541		983				78
	981		673				

Nach dem zweiten Durchlauf ergibt sich folgendes Bild::

B_0	B_1	B_2	B_3	B_4	B_6	B_7	B_8	B_9
500	15	123	32	541	61	673	981	94
	16				68	78	983	

Der nächste Durchlauf ergibt::

B_0	B_1	B_5	B_6	B_9
15	123	500	673	981
16		541		983
32				

61				
68				
78				
94				

Hieraus ergibt sich die Folge:
15 - 16 - 32 - 61 - 68 - 78 - 94 - 123 - 500 - 541 - 673 - 981 - 983

Aufgabe 3

(a)

Im folgenden bezeichne $h(e, m)$ bezeichne eine Hashfunktion, die ein Element e auf einen Behälter zwischen 0 und $m-1$ abbildet.

algorithm *insert*(*s*: *set*, *e*: *elem*)
{ *s*: Menge, die in einer Hashtabelle organisiert ist.
e: Element, das hinzugefügt werden soll.}

/* Prüfe, ob reorganisiert werden muß. */
if $n / m > c$ **then**
reorganize(*s*)
else
$i := h(e, m)$
$ptr := s\uparrow[i]$;
erzeuge ein neues Element von Typ *listelem* und definiere einen Zeiger *l*, der darauf zeigt;
$l\uparrow.value := e$;
$l\uparrow.next := ptr$;
$s\uparrow[i] := l$;
$n := n + 1$
end if.

algorithm *member*(*s set*, *e elem*)
{ siehe oben }

$ptr := s[h(e, m)]$; /* Zeiger auf den Behälter, der *e* enthalten könnte. */
while $ptr \neq nil$ **do**
if $ptr\uparrow.value = e$ **then**
return true
else
$ptr := ptr\uparrow.next$
end if
end while;
return false.

```
algorithm delete(s set, e elem)
{ siehe oben }

ptr := s↑[h(e, m)];
/* Um ein Element aus der Liste zu löschen, benötigen wir den Vorgänger. */
prev := ptr;
while ptr ≠ nil do
  if ptr↑.value = e then
     next := ptr↑.next;
     prev↑.next = next;
  else
     prev := ptr;
     ptr := ptr↑.next
  end if
end while.

algorithm reorganize(s: set)
{ Die Anzahl der Behälter wird geändert. }

k := m; /* alte Behälterzahl */
m := n; /* neue Behälterzahl */
erzeuge ein neues Array der größe m mit Elementen vom Typ ↑listelem;
setzte einen Zeiger a auf das Array;
for i = 0 .. k–1 do
  ptr := s↑[i];
  while ptr ≠ nil do
     insert(a, ptr↑.value);
     ptr := ptr↑.next
  end while
end for.
```

Laufzeitfunktion für Operation *member*

Um ein gesuchtes Element, welches in der Hashtabelle gespeichert ist, aufzufinden, müssen im Durchschnitt die Hälfte aller Elemente, die in einem Behälter gespeichert sind, verglichen werden. Die durchschnittliche Anzahl von Elementen in einem Behälter ist n / m. Daher ist die durchschnittliche Laufzeit gegeben durch

$$t_1(n) = 1 + n / 2m.$$

Ist das gesuchte Element nicht in der Hashtabelle gespeichert, werden alle Elemente eines Behälters mit diesem verglichen, es entstehen die Kosten

$$t_2(n) = 1 + n / m.$$

Die aus beiden Werten gemittelten durchschnittlichen Kosten hängen also vom Verhältnis zwischen Anfragen nach gespeicherten und nicht gespeicherten Elementen ab. Man erhält

$$t(n) = p \cdot t_1(n) + (1 - p) \cdot t_2(n) = 1 + n \cdot (2 - p) / 2m.$$

(b)

Aus $P_k < x$ folgt, daß für die Wahrscheinlichkeit für das Gegenereignis P_{nk} (es findet keine Kollision statt) gilt: $P_{nk} \geq 1 - x$. Die Wahrscheinlichkeit für P_{nk} berechnet sich als das Produkt $P(1) \cdot P(2) \cdots P(n)$, man erhält:

$$P_{nk} = \frac{m(m+1)\ldots(m+n+1)}{m^n}$$

Ersetzt man im Zähler jeden Faktor durch $m–n$, so folgt:

$$P_{nk} = \frac{m(m+1)\ldots(m+n+1)}{m^n} > \left(\frac{m+n}{m}\right)^n \geq 1 - x$$

Durch Anwendung des Logarithmus kann man dies folgendermaßen nach m auflösen:

$$\log_n(1-x) \leq 1 - n/m \Leftrightarrow 1 - \log_n(1-x) \geq n/m \Leftrightarrow m \geq n \,/\, (1 - \log_n(1-x))$$

Damit läßt sich zu gegebenem n und c ein m ermitteln, so daß $P_k < x$ gilt.