# Lösungsvorschläge zur Hauptklausur „1662/1663 Datenstrukturen“

**9. August 2003**

Aufgabe 1

(a)

Bei jedem rekursiven Aufruf werden drei neue Felder erzeugt. (eins im Prozedurkopf, und zwei durch die merge-Aufrufe). Jedes Feld benötigt MAX Speicherstellen. Vor Verlassen werden diese wieder gelöscht. Die Anzahl der erzeugten Felder ist daher abhängig von der Rekursionstiefe der Funktion, d.h. von der Tiefe des Aufrufbaumes. Da es sich um einen ausgeglichenen Binärbaum handelt ist beträgt die Höhe log(MAX) und der
Speicherbedarf ist O(MAX log(MAX)).

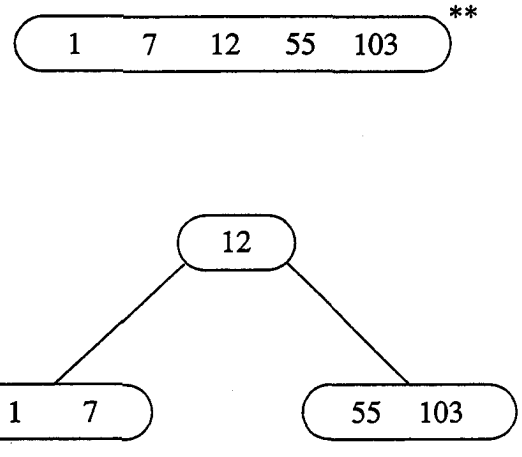
(b)

Wird der temporäre Speicher nicht wieder freigegeben, so bleiben von jedem rekursiven Aufruf die beiden durch new erzeugten Felder mit jeweils MAX Speicherstellen übrig. Es gibt so viele Aufrufe, wie der Aufrufbaum Knoten besitzt. Somit beträgt die Speicherverschwendung $O(MAX^2)$.

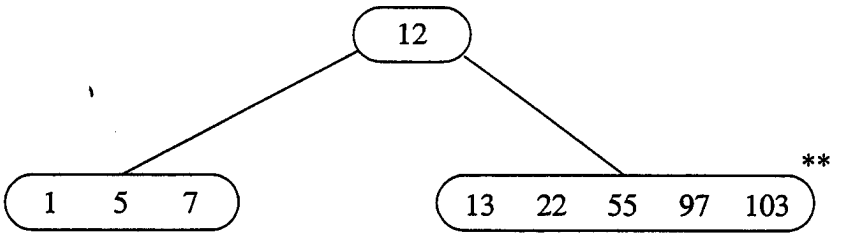
## Aufgabe 2

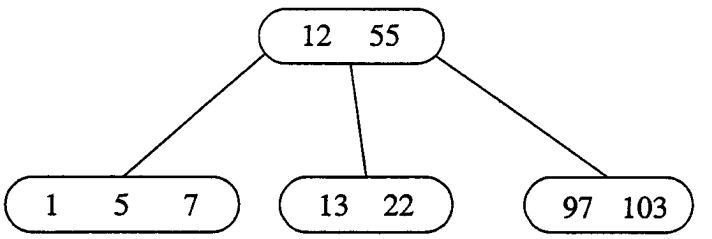
(a)

Nach dem Einfügen der Schlüssel 7, 12, 55, 103, 1 ist die erste Overflow-Behandlung notwendig. Die betroffenen Knoten werden mit ** markiert.

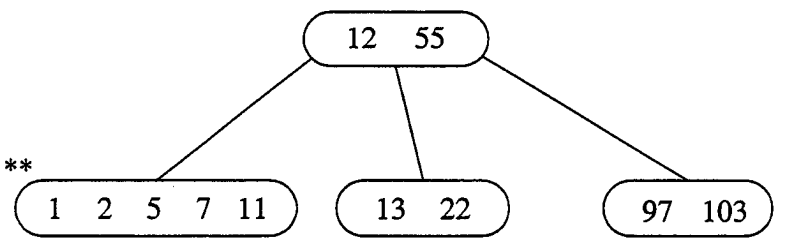


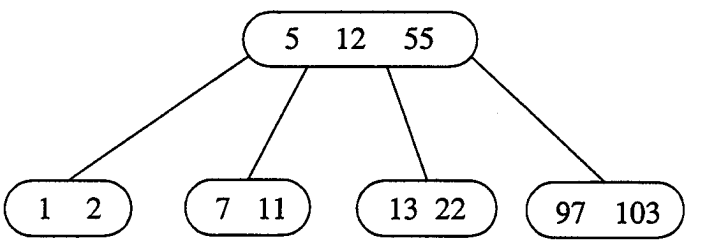
Der nächste Overflow tritt beim Einfügen des Schlüssels 13 auf.



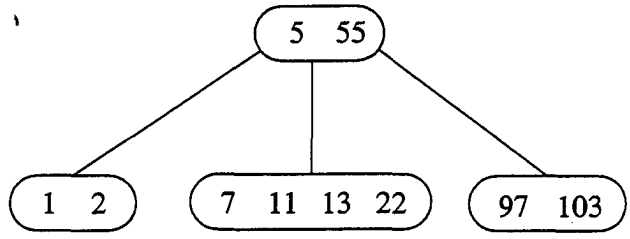

Die letzte Overflow-Behandlung wird mit dem Einfügen des Schlüssels 2 nötig.



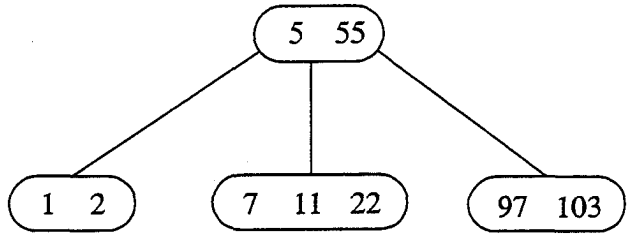



(b)

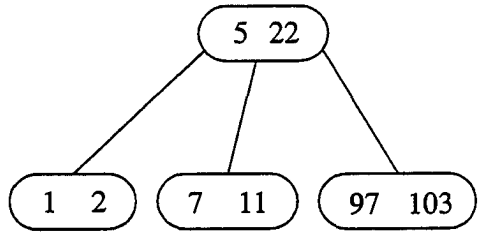
Der Baum nach dem Löschen des Schlüssels 12:




Nach dem Löschen des Schlüssels 13:




Der Ergebnisbaum nach dem Löschen des Schlüssels 55:



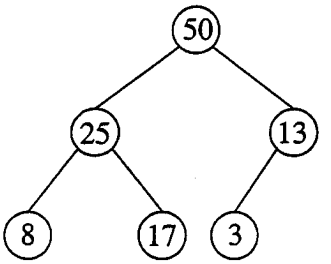

## Aufgabe 3

(a)

Im Gegensatz zum „normalen“ Heapsort werden die Entscheidungen beim Reheap getrennt. Zunächst wird der Heap von oben nach unten durchlaufen, um den Pfad von der Wurzel bis zu einem Blatt *b* zu bestimmen, auf dem das einsinkende Element *e* liegen bleiben muß. Bei jedem Knoten ist, von der Wurzel ausgehend, der kleinere Sohn zu wählen. Anschließend wird genau dieser Pfad von unten nach oben so lange zurückverfolgt, bis ein Element *q* gefunden wurde, das kleiner (beim Maxheap größer) als *e* ist. Nun wird *e* aus der Wurzel entnommen, alle Elemente auf dem Pfad von oben bis zu *q* werden eine Position nach oben geschoben und *e* wird an der alten Position von *q* eingefügt.

(b)

Der Heap sieht zu beginn folgendermaßen aus:

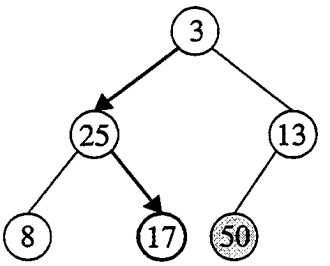


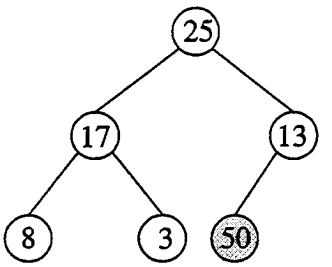

Dies ist die Array-Einbettung:

| 50 | 25 | 13 | 8 | 17 | 3 |
|---|---|---|---|---|---|

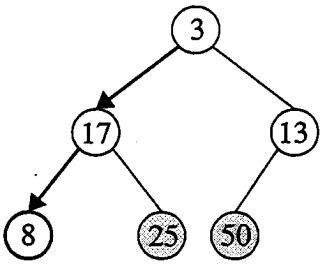
Im Folgenden wird immer das oberste Element entnommen und mit dem letzten Element vertauscht. Die sortierte Folge wird grau hinterlegt.

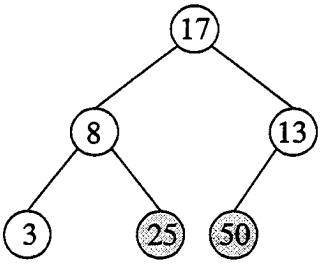
Entnehmen von Schlüssel 50:



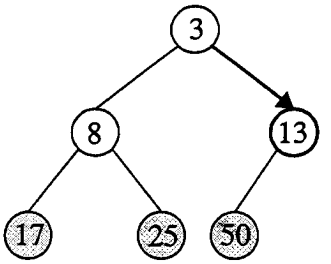



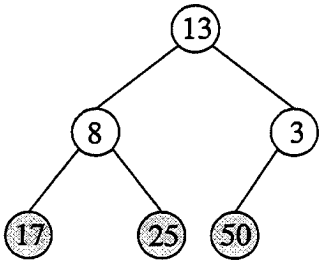
Entnehmen von Schlüssel 25:



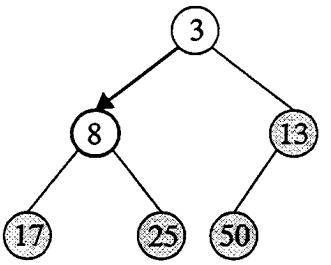



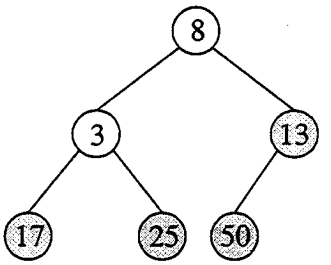

Entnehmen von Schlüssel 17:



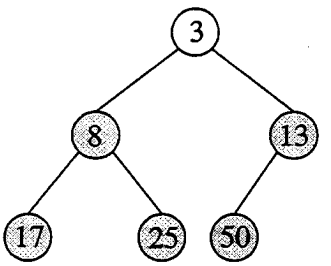



Entnehmen von Schlüssel 13:




Entnehmen von Schlüssel 8:




Nun ist die Folge sortiert und kann ausgegeben werden.

## Aufgabe 4

(a)

**algorithm** *TransHuelle*(
  **in:** $A$, Adjazenzmatrix ($|V|\times|V|$) des ungerichteten Graphen $G=(V,E)$,
  **out:** $T$, Adjazenzmatrix der Transitiven Hülle von $G$ )
**begin**
  initialisiere $T$ mit den Werten von $A$ und speichere alle
  Koordinaten $(i,j)$ mit $j \geq i$ in denen $A_{ij} = 0$ ist in einer Liste $L$;
  **for** $k := 1$ **to** $|V|$ **do**
    **for all** $(i,j)$ **in** $L$ **do**
      $T_{ij} := T_{ik}$ **and** $T_{kj}$; $T_{ji} := T_{ji}$;
      remove $(i,j)$ from $L$;
    **end**
  **end**
**end** *TransHuelle*

Statt Kostenmatrizen werden nun nur noch Adjazenzmatrizen benötigt. Im Gegensatz zu Floyd wird hier nur noch geprüft, ob die Knoten $i$ und $j$ über den „Umweg“ $k$ verbindbar sind. Da die Adjazenzmatrix $A$ eines ungerichteten Graphen symmetrisch ist, folgt daß die Adjazenzmatrix $T$ der Transitiven Hülle symmetrisch sein muss, denn aus $T_{ik} = T_{ki}$ und $T_{kj} = T_{jk}$ impliziert $T_{ij} = T_{ji}$. Weiterhin werden in jedem Iterationsschritt $k$ nur neue Werte für $T_{ij} \neq 0$ berechnet. Um die Iteration zu verkürzen, werden alle Paare, die keine direkte Verbindung haben, in einer Liste gespeichert.

(b)

Die Adjazenzmatrix $A$ für den Graphen aus der Aufgabenstellung sieht folgendermaßen aus:

$$A_{ij} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Damit sind lediglich folgende Berechnungen durchzuführen:

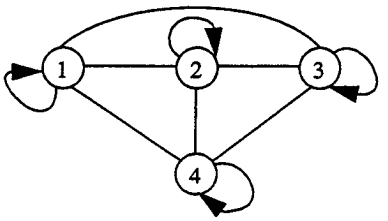
$k$=1:
$A_{11} = A_{11} \wedge A_{11} = 0$
$A_{13} = A_{11} \wedge A_{13} = 0$
$A_{24} = A_{21} \wedge A_{14} = 0$
$A_{22} = A_{21} \wedge A_{12} = 1$
$A_{33} = A_{31} \wedge A_{13} = 0$
$A_{44} = A_{41} \wedge A_{13} = 1$

$k$=2:
$A_{11} = A_{12} \wedge A_{21} = 1$
$A_{13} = A_{12} \wedge A_{23} = 1$
$A_{24} = A_{22} \wedge A_{24} = 0$
$A_{33} = A_{32} \wedge A_{23} = 1$

$k$=3:

$A_{24} = A_{23} \wedge A_{34} = 1$

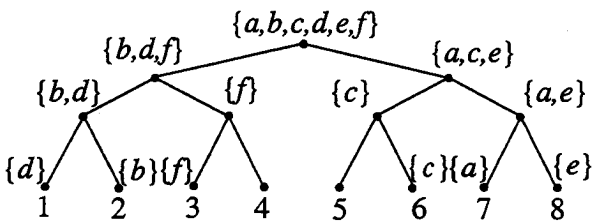
Damit gilt stets $T_{ij} = 1$, weitere Berechnungen sind also nicht notwendig, der Graph der Transitiven Hülle sieht folgendermaßen aus:




## Aufgabe 5

(a)

Man erzeugt einen Baum über 8 Rasterpunkten und trägt die Koordinaten in die Knotenlisten längs des Weges von der Wurzel zu einem Rasterelement ein. Die Buchstaben stellen Referenzen zu den Werten der einzufügenden Zahlenfolge dar:




(b)

Ein vollständiger Baum, der auf der letzten Ebene $N$ Elemente besitzt, besteht insgesamt aus $2N - 1$ Elementen und hat somit einen Platzbedarf von $O(N)$. Die Höhe eines solchen Baumes beträgt $O(\log(2N\text{-}1)) = O(\log(N))$. Beim Einfügen von $n$ Elementen wird also Speicherplatz in der Größenordnung $O(n\log(N))$ benötigt. Insgesamt benötigt der Range-Baum dann $O(N+n\log(N))$ Speicherplatz.

## Aufgabe 6

(a)

Wir verwenden eine Variante von Quicksort. Die Idee ist, daß das gesuchte Element nur in einer der beiden entstandenen Partitionen gesucht werden muß. Der Algorithmus erhält die zu durchsuchende Folge $S=s_1..s_n$ sowie $k$ als Eingabe.

```
algorithm findk(S,k)
if |S|=1 then
  return s_1.key
else
  wähle x=s_j.key aus S, so daß x nicht minimal in S ist
  berechne eine Teilfolge S_1=s_1...s_j und
  eine Teilfolge S_2=s_{j+1}..s_n wie im Quicksort-Algorithmus
  if k ≤ j then
    return findk(S_1,k)
  else
    return findk(S_2,k-j)
  fi
fi
```

(b)

Durchschnittsanalyse:

Bei der Partitionierung wird ein zufälliges $j$ im Bereich von $1..n$ berechnet. In Abhängigkeit von $j$ und $k$ wird dann die Teilfolge von $1..j$ oder von $j+1..n$ betrachtet. Dies kann entweder die größere oder kleinere (oder gleich große) Teilfolge sein. Bei angenommener Gleichverteilung ist im Durchschnitt die halbe Eingabefolge zu untersuchen. Für die Auswahl von $x$ und die anschließende Partitionierung wird $O(n)$ Zeit benötigt (vgl. Quicksort). Der Unterschied zu Quicksort ist, daß jeweils nur eine Teilfolge durchsucht werden muß.Wir erhalten daher als Rekursionsgleichung:

$$T(1) = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + n$$

Aufgelöst ergibt sich : $T(n) = 2n - 1$

IA: 2-1=1

IS: $n \rightarrow 2n$:

$$\begin{aligned} T(2n) &= T(n) + 2n \\ &= 2n - 1 + 2n \\ &= 2 \cdot 2n - 1 \end{aligned}$$