Übungsblatt 12 – Lösungen

Aufgabe 1 - Mustererkennung

- Gebe die Zustandsüberführungsfunktion für einen *Moore-Automaten* an, der in einer Folge von Zeichen aus Σ mit $\{a,b,c\} \subseteq \Sigma$ das Muster `ababaca` erkennt. Benutze die aus der Vorlesung bekannte tabellarische Notation.

	0	1	2	3	4	5	6	7
a	1	1	3	1	5	1	7	1
b	0	2	0	4	0	4	0	2
c	0	0	0	0	0	6	0	0
$s \in \Sigma \setminus \{a,b,c\}$	0	0	0	0	0	0	0	0

- Gebe die Zustandsüberführungsfunktion für einen *Moore-Automaten* an, der in einer Folge von Zeichen aus Σ mit $\{a,b,c\} \subseteq \Sigma$ das Muster `aabcaac` erkennt. Benutze die aus der Vorlesung bekannte tabellarische Notation.

	0	1	2	3	4	5	6	7
a	1	2	2	1	5	6	2	1
b	0	0	3	0	0	0	3	0
c	0	0	0	4	0	0	7	0
$s \in \Sigma \setminus \{a,b,c\}$	0	0	0	0	0	0	0	0

Aufgabe 2 - Hashing

In der Vorlesung ist Hashing als eine Form zum schnellen Ablegen und Wiederfinden von Informationen präsentiert worden.
Die dort vorgestellte Inplementierung benutzt Listen, um Objekte mit kollidierendem Hashcode abzulegen.
Die Implementierung kann aber auch ohne Listen erfolgen:
Tritt eine Kollision auf, wird für das neu abzulegende Objekt das "nächste" freie Element des Feldes verwendet – also der kleinste größere Index, dessen zugehöriges Element auf `null` verweist. Wird bei der Suche nach so einem Element das Ende des Feldes erreicht, so wird mit dem Index `0` fortgefahren.
Überlege Dir, wie die Methode `contains` arbeiten muss.
Die Methode `contains` muss das Verhalten der Methode `put` nachbilden, also ebenfalls mit ab der durch den HashCode bestimmten Position in der Tabelle weitersuchen. Ist das Löschen von Werten nicht vorgesehen, kann die Suche in `contains` beim ersten freien Element abgebrochen werden, da ja der zu suchende Wert dort durch `put` eingetragen sein müsste.

Welches besondere Problem wird bei dieser Implementierung durch das Löschen eines Wertes hervorgerufen?
Beim Löschen können beliebig freie Elemente entstehen, so dass das Nichtenthaltensein nur durch einen kompletten Durchlauf durch das Feld festgestellt werden kann.
Überlege Dir eine Lösung!

Lösungen könnten sein:
– Diese Implementierung nur dann einsetzen, wenn Löschen nicht benötigt wird – und dann auch keine Methode `remove` anbieten.
– In einem zweiten Feld gleicher Größe wird markiert, ob ein Element jemals benutzt wurde oder nicht. Dann kann `contains` enden, wenn ein nie benutztes Element gefunden wird.

Implementiere die Klasse `SimpleHashTable` mit den bekannten Methoden:

```
public void put( T o )
public boolean contains( T o )
public void remove( T o )
public int size()
private void rehash()
```

Aufgabe 3 - Heap

- Wie viele Ebenen hat ein Heap mit
  – 32 000 Elementen
  – 1 000 000 Elementen?

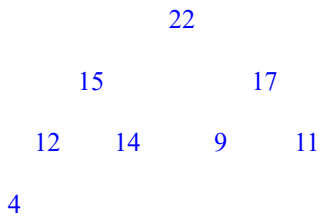
  Die k-te Ebene des Heaps hat 2^{k-1} Elemente,
  ein Baum mit k Ebenen also $2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = 2^k - 1$.
  Daraus folgt für n Elemente: (aufgerundet) $\log_2(n+1)$ Ebenen.
  Also benötigt ein Baum mit 32.000 Elementen $\log_2(32000) = 15$ Ebenen,
  ein Baum mit 1 000 000 Elementen $\log_2(1000000) = 20$ Ebenen.
  Für grobe Abschätzungen: $2^5 = 32$, $2^{10} = 1024$, *also* $32 \times 1024 = 2^{15}$, $1024 \times 1024 = 2^{20}$

  An welchen Stellen der Algorithmen zum Einfügen und Löschen beeinflusst die Zahl der Ebenen die Laufzeit?

  Die Zahl der Ebenen bestimmt bei beiden Algorithmen die maximale Zahl der Vergleiche.
  Da immer nur entlang eines Pfads von der Wurzel zu einem Blatt (Löschen) oder von einem Blatt zur Wurzel (Einfügen) verglichen wird, stellen $2 \cdot \log_2(n)$ für das Löschen und $\log_2(n)$ für das Einfügen Obergrenzen für die Zahl der Vergleiche dar.

- Das unten stehende Feld bildet einen Heap. Gib die zugehörige Baumdarstellung an!

	0	1	2	3	4	5	6	7	8
pq		22	15	17	12	14	9	11	4



- Erzeuge aus dem Heap `pq` mit Hilfe der folgenden Anweisungen (siehe Folie 1185) eine aufsteigende Sortierung:

```
for ( int i = elements.length-1; i >= 0; i-- ) {
    elements[i] = pq.poll();
}
```

Gib an, wie die beiden Felder am Ende jedes Schleifendurchlaufs aussehen

	0	1	2	3	4	5	6	7	8
pq		22	15	17	12	14	9	11	4

	0	1	2	3	4	5	6	7
elements	0	0	0	0	0	0	0	0

pq		17	15	11	12	14	9	4	0

elements	0	0	0	0	0	0	0	22

pq		15	14	11	12	4	9	0	0

elements	0	0	0	0	0	0	17	22

pq		14	12	11	9	4	0	0	0

elements	0	0	0	0	0	15	17	22

pq		12	9	11	4	0	0	0	0

elements	0	0	0	0	14	15	17	22

pq		11	9	4	0	0	0	0	0

elements	0	0	0	12	14	15	17	22

pq		9	4	0	0	0	0	0	0

elements	0	0	11	12	14	15	17	22

pq		4	0	0	0	0	0	0	0

elements	0	9	11	12	14	15	17	22

pq		0	0	0	0	0	0	0	0

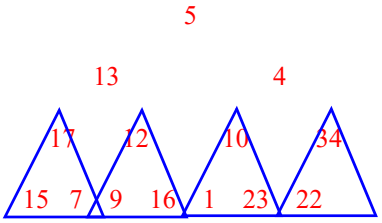
elements	4	9	11	12	14	15	17	22

- Die in der Vorlesung vorgestellte Methode `poll` entfernt den Wert in der Wurzel des Heaps, ersetzt ihn durch einen neuen Wert und stellt den Heap mit der Methode `heapify` aus den beiden Teilbäumen, die Heaps sind, wieder her. Dieses Vorgehen kann auch benutzt werden, um aus einer Folge von unstrukturierten Werten einen Heap herzustellen.

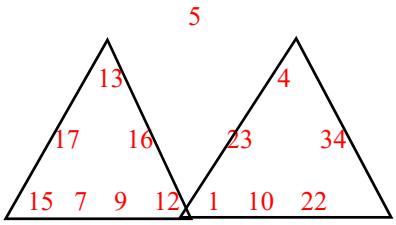
  Stelle das folgende Feld als Baum gemäß der für Heaps gültigen Abbildung der Werte dar: Es bildet dann **keinen** Heap! Füge nun ausgehend von den Vorgängerknoten der Blätter – ein einzelner Knoten ist ein Heap – analog zu `heapify` – ebenenweise von unten immer größere Heaps zusammen.

.

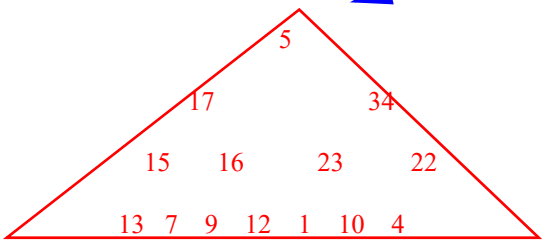
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	5	13	4	17	12	10	34	15	7	9	16	1	23	22



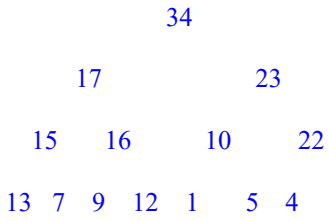
zunächst heapify für die Knoten
34, 10, 12 und 17 und es entsteht folgender Aufbau:



dann für die Knoten 13 und 4
und es entsteht folgender Aufbau:



und für die Wurzel 5:



Beschreibe das so entwickelte Vorgehen durch eine allgemeine Handlungsanweisung!

Rufe für die vorderen `n/2` Werte des Feldes vom größten (`n/2`) zum kleinsten Index (`0`) jeweils eine Methode `heapify( int i )` in einer Schleife auf, die am Index `i` mit ihren Vergleichen beginnt.

```
for ( int i = n/2; i>0; i-- ) {
    heapify( i );
}
```