Aufgabe 1

(a)

$T_1(n) = 4n + 23 = O(n)$

$T_2(n) = 23n^2 - 6n + 1024 = O(n^2)$

$T_3(n) = 18n^3 + n^2\log(n) = O(n^3)$

$T_4(n) = (18n^3 + n^2)\log(n) = O(n^3\log(n))$

$T_5(n, m) = nm + 63m\log(n) = O(nm)$

$T_6(n, m) = n\log(m) + m\log(n) = O(n\log(m) + m\log(n))$

$T_7(n, m, o) = 5n^3 + 8m^2 + 10m\log(o) = O(n^3 + m^2 + m\log(o))$

$T_8(n, m) = 3m^3n^2 + 5m^2n^2 + m^3\log(n) = O(m^3n^2)$

$T_9(n, m) = \max(3^m + 5n^2, 16m^3 + 9n + 9867) = O(3^m + n^2)$

$T_{10}(n, m) = 2^{nm} + (nm)^{12} + n^m = O(2^{nm})$

(b)

In *alg*1 wird lediglich eine for-Schleife mit n Schritten durchgeführt, daher ist *alg*1 in $O(n)$. In *alg*2 wird die Zahl bei jedem Schleifendurchlauf gedrittelt. Der Aufwand beträgt somit $O(\log(n))$. Im dritten Algorithmus hat die innere Schleife jeweils eine „Länge“ von 2^i. Der Gesamtaufwand läßt sich also durch

$$\sum_{i=1}^{n} 2^i = O(2^n)$$

beschreiben. In *alg*4 enthält die erste Schleife n Iterationen, die zweite Schleife iteriert im Mittel über $n/2$ Werte. Die innere Schleife hat $\log(n)$ Durchläufe, woraus sich ein Gesamtaufwand von $O(n^2\log(n))$ ergibt.

Aufgabe 2 (Hashing)

(a)

Die Hashfunktion berechnet für die Schlüsselfolge die nachfolgend angegeben Werte:

$h(107) = 2$

$h(20) = 0$

$h(37) = 2$

$h(46) = 1$

$h(1) = 1$

$h(2) = 2, h_1(2) = 3$

$h(4) = 4$

$h(50) = 0$

$h(0) = 0; h_1(0) = 1, h_2(0) = 2, h_3(0) = 3$

Die Hashtabelle sieht dann wie folgt aus:

Feld	Eintrag1	Eintrag2
0	20	50
1	46	1
2	107	37
3	2	0
4	4	

(b)

Beim Doppelhashing ergeben sich folgende Werte:

$h(107) = 8$

$h(20) = 9$

$h(37) = 4$

$h(46) = 2$

$h(1) = 1$

$h(2) = 2, h_1(2) = 6$

$h(4) = 4, h_1(4) = 1, h_2(4) = 3$

$h(50) = 6, h_1(50) = 7$

$h(0) = 0$

Damit ergibt sich die nachfolgend angegebene Hashtabelle:

Feld	Eintrag
0	0
1	1
2	46
3	4
4	37
5	
6	2
7	50
8	107
9	20
10	

(c)

Hashfunktionen sollen drei Eigenschaften haben: Sie sollen surjektiv sein, gleichmäßig auf alle Behälter verteilen und effizient zu berechnen sein.

Beim Doppelhashing ist zusätzlich darauf zu achten, dass beide Hashfunktionen voneinander unabhängig sind, was i.A. aber schwer zu beweisen ist.

Die in Aufgabenteil (b) verwendete Funktion ist nicht surjektiv, verteilt nicht gleichmäßig und ist definitiv nicht unabhängig von $h(x)$. Damit ist $h'(x)$ klar ungeeignet als Hashfunktion allgemein und speziell beim Doppelhashing.

Aufgabe 3

Da die Wörter aus vier Buchstaben bestehen, benötigt Radixsort vier Phasen. Nur das Wort 'vier' besteht tatsächlich aus vier Buchstaben, so daß in der ersten Phase nur zwei Behälter $B_\_$ und B_r gefüllt werden.

$B_\_$	B_r
es sei die mit nur aus und dem das als der sie an	vier

Die Ausgabe der ersten Phase dient als Eingabe für die zweite Phase, deren Ausgabe in der folgenden Tabelle dargestellt ist.

$B_\_$	B_d	B_e	B_i	B_m	B_r	B_s	B_t
es an	und	die sie vier	sei	dem	nur der	aus das als	mit

Nach der dritten Phase zeigt sich folgende Behälterkonfiguration:

B_a	B_e	B_i	B_l	B_n	B_s	B_u
das	sei dem der	die sie vier mit	als	an und	es	nur aus

Der Ergebnis der vierten Phase ist:

B_a	B_d	B_e	B_m	B_n	B_s	B_u	B_v
als an aus	das dem der die	es	mit	nur	sei sie	und	vier

Als Ergebnisfolge wird

als an aus das dem der die es mit nur sei sie und vier

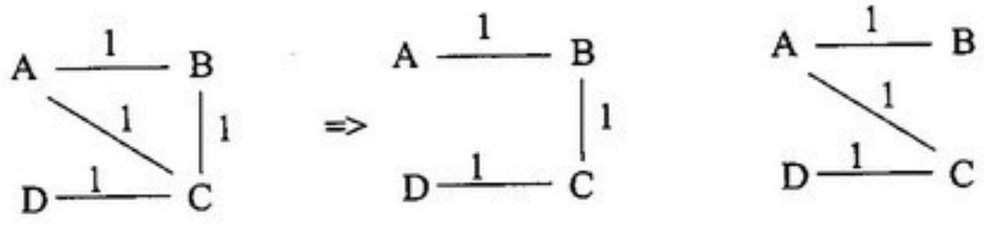
ausgegeben.

Aufgabe 4

(a)

Zu einem verbundenen Graphen $G = (V, E)$, dessen Kanten bewertet sind, gebe $k(e)$ die Kosten für eine Kante e an. Ein Baum $T = (V, E')$ mit $E' \subseteq E$ und $|E'| = |V| - 1$ heißt *minimaler Spannbaum*, falls die Summe aller $k(e')$ unter allen aus G ableitbaren T minimal ist.

Aussage (i) läßt sich folgendermaßen widerlegen:



Zu dem oben angegebenen Graphen lassen sich also leicht zwei Spannbäume mit denselben minimalen Kosten angeben, die aber unterschiedliche Kanten haben.

Aussage (ii) ist offensichtlich falsch, denn jeder Graph, der nur aus zwei Knoten und einer Kante besteht, hat natürlich nur genau einen minimalen Spannbaum.

(b)

Der Test kann folgendermaßen durchgeführt werden:

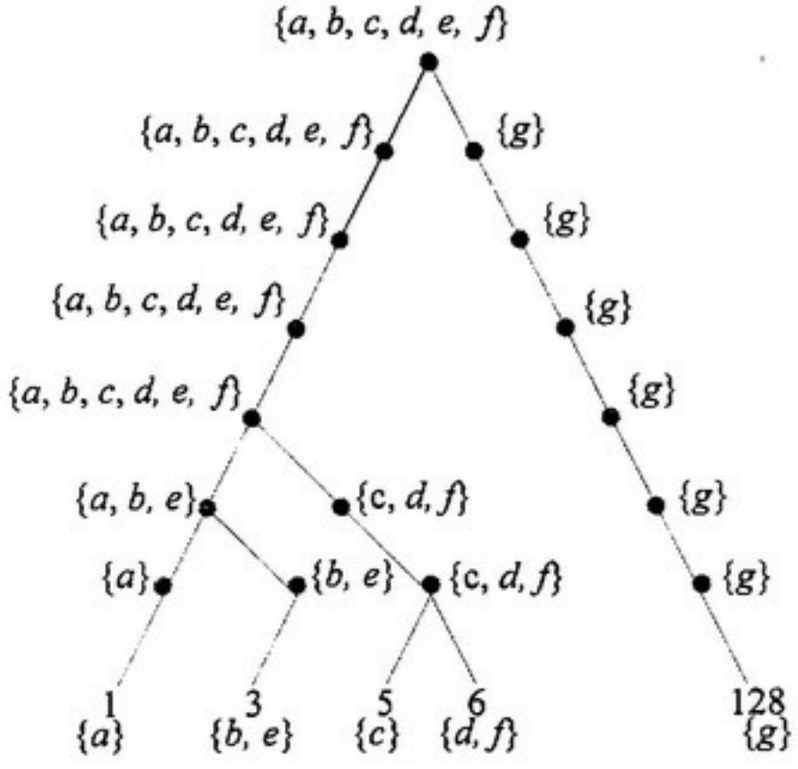
1.) Sei $G = (V, E)$. Zunächst führen wir einen Tiefendurchlauf durch T durch und testen dabei, ob der Graph $T = (V', E')$ ein Baum ist, und $V = V'$, $E' \subseteq E$ und $|E'| = |V| - 1$ gilt. Weiterhin können dabei auch die Kosten von T berechnet werden.

2.) Nun wenden wir den Algorithmus von Kruskal für G an, aber statt den Spannbaum zu berechnen werden lediglich dessen Kosten ermittelt. Sind diese Kosten niedriger als die zuvor berechneten, so ist T kein minimaler Spannbaum von G.

Aufgabe 5

(a)

Um alle Koordinaten im Bereich 1-128 darstellen zu können benötigen wir einen Rangebaum der Höhe 7, die unterste Ebene enthält dann die Koordinaten. Man erhält somit folgenden Baum:



(b)

Zunächst wird die Punktmenge nach x- und y-Koordinaten sortiert, um die benötigten Rasterkoordinaten zu identifizieren. Man berechnet zwei Integer-Arrays *xRaster*, *yRaster*. Danach erzeugt man mittels

Rx = **new** *Rangebaum*(*xRaster*);

einen Rangebaum über dem Raster der benötigten x-Koordinaten. Nun durchläuft man die Punktmenge und für jeden Punkt $P = (x, y)$ fügt man diesen folgendermaßen ein.

```
Object[] aRy = Rx.query(x, x);
Rangebaum Ry;
if ( aRy.length == 0 ) {
    Ry = new Rangebaum(yRaster);
    Rx.insert(x, Ry)
} else {
    Ry = (Rangebaum) aRy[0];
}
Ry.insert(y, P);
```

Man prüft ob die x-Koordinate bereits im Baum enthalten ist, falls nicht, so wird ein neuer Rangebaum Ry über dem y-Raster erzeugt und in Rx eingefügt. Andernfalls erhält man einen Rangebaum Ry. Danach wir P bezüglich der y-Koordinate in Ry eingefügt

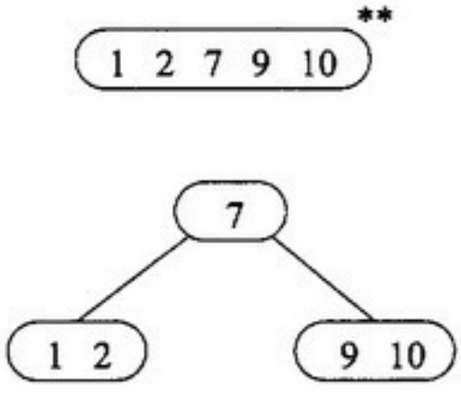
Für ein Query-Rechteck $R(x_1, x_2, y_1, y_2)$ erhält man nun die darin enthaltenen Punkte indem man in einer ersten Query ein Array von Rangebäumen aRy ermittelt und auf diesen eine Suche bezüglich der y-Koordinaten durchführt.

```
aRy = Rx.query(x1, x2);
Object[] aP;
for (int i = 0, i < aRy.length, i++) {
    aP = aRy[i].query(y1, y2);
    /* verarbeite die Punkte in aP */
}
```

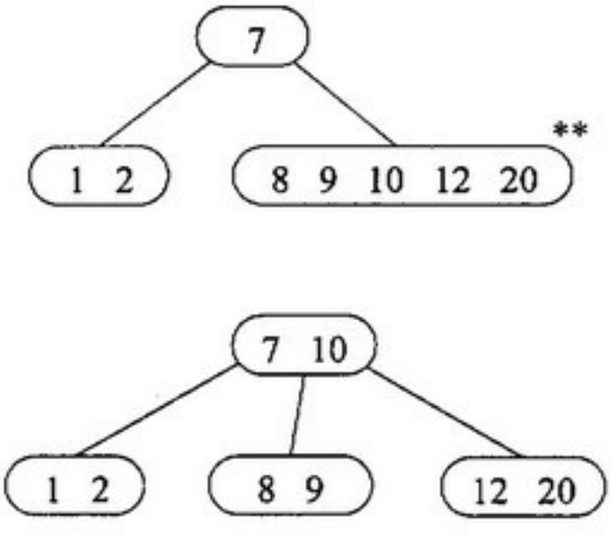
Aufgabe 6

(a)

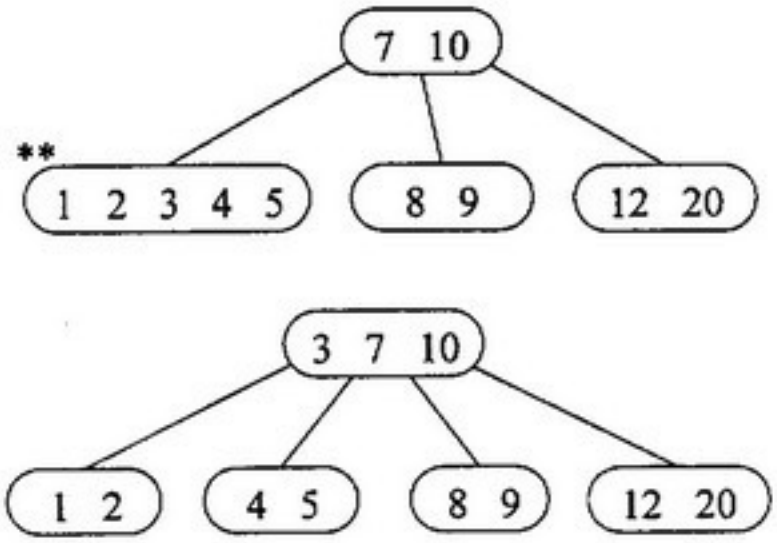
Overflow nach dem Einfügen von 10:



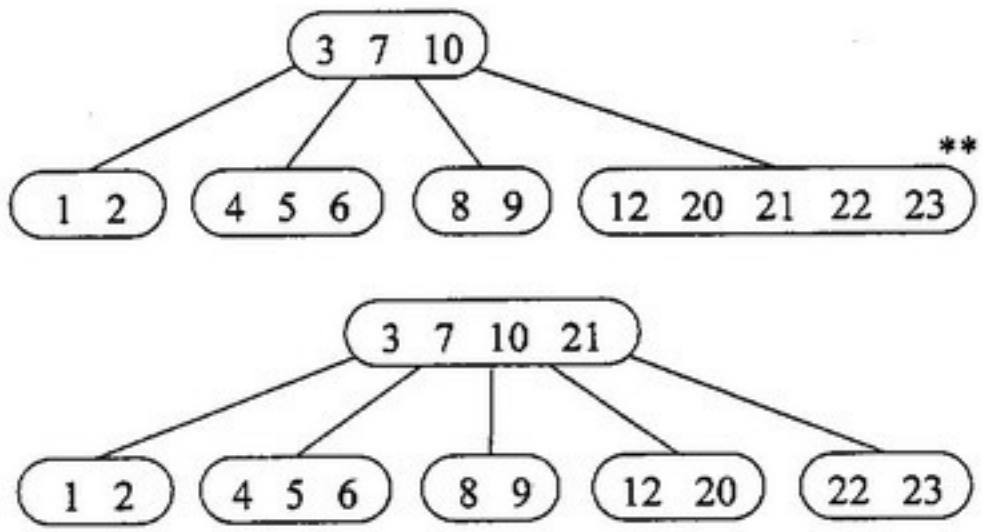
Overflow nach dem Einfügen von 20:



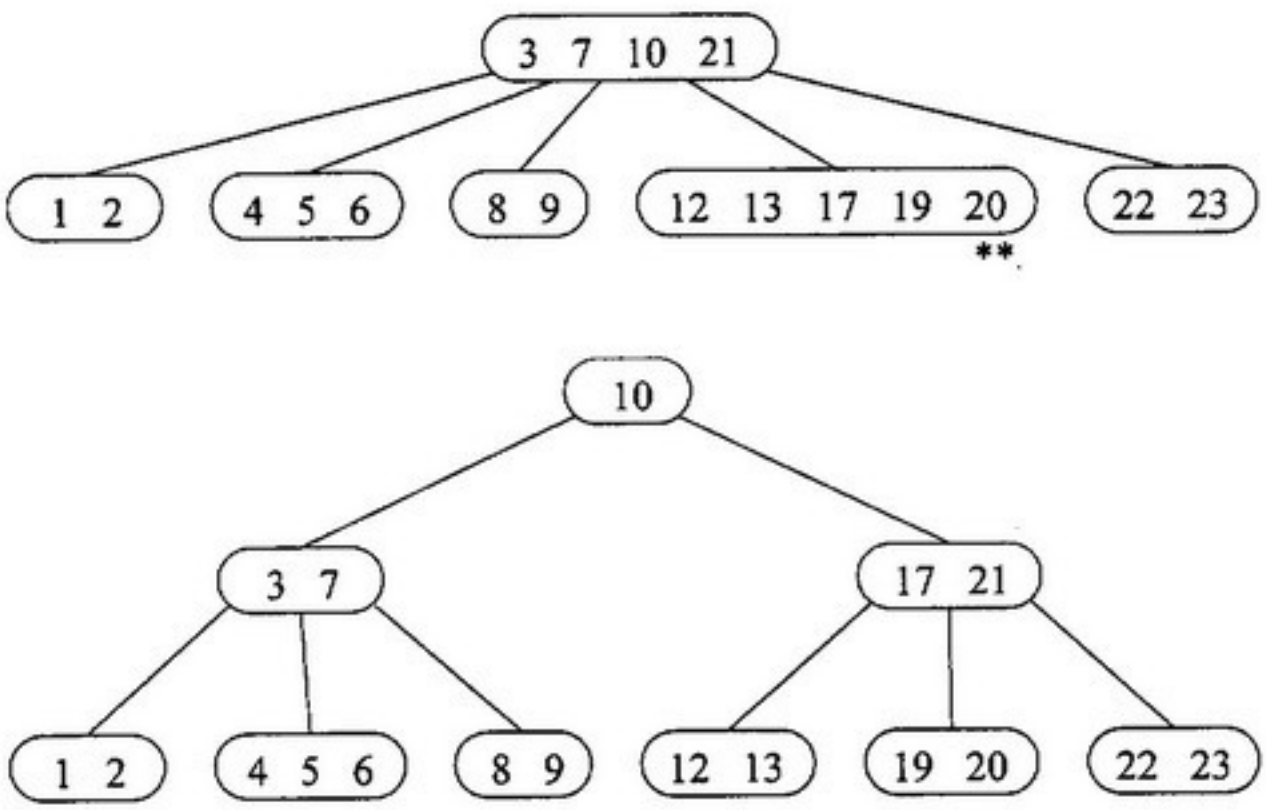
Overflow nach dem Einfügen von 5:



Overflow nach dem Einfügen von 23:



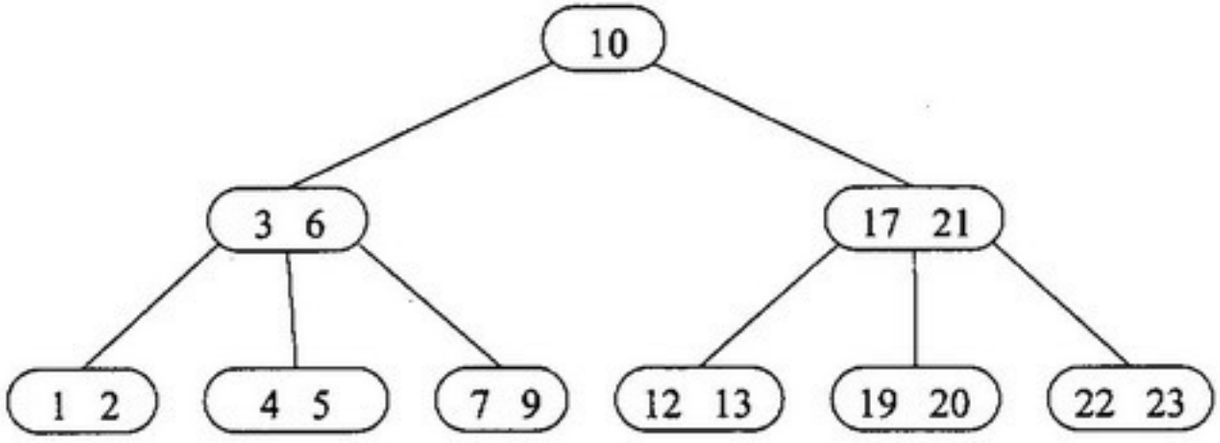
Overflow nach dem Einfügen von 19:



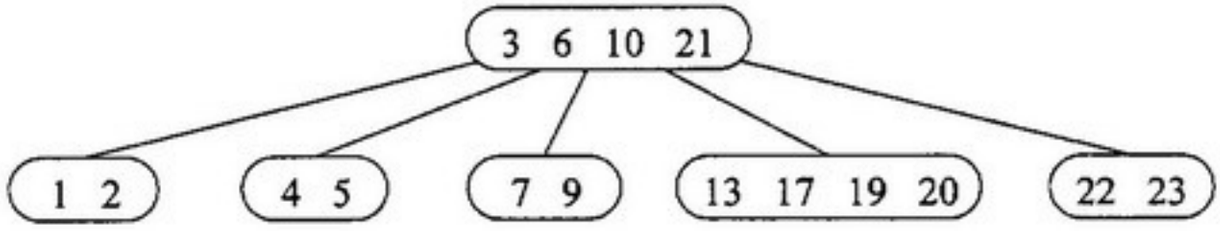
Der letzte Baum ist gleichzeitig der Ergebnisbaum nach allen Einfügeoperationen.

(b)

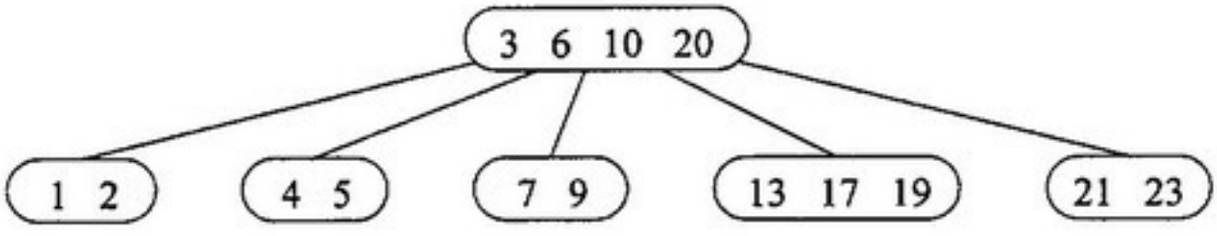
Löschen des Schlüssels 8, balance:



Löschen des Schlüssels 12, merge:



Löschen des Schlüssels 23, balance:



Löschen des Schlüssels 1, merge: