2. Semester: Teilklausur II in Informatik – Kurs TEL 09 GR 2

Bearbeitungszeit:	**90 Minuten**	**Dozent:**	**Dr. S. Lewandowski**
Hilfsmittel:	**keine**	**Datum:**	**16.07.2010**

Aufgabe	max. Punkte	erreichte Punkte
1: Interpolationssuche (Wissens- und Verständnisfrage)	6	
2: Binärbäume (C/C++, durchführen, analysieren, O-Not.)	14	
3: Einfügen in AVL-Bäumen (durchführen)	10	
4: Löschen in AVL-Bäumen (durchführen)	5	
5: Heapsort (durchführen)	12	
6: Dijkstra-Algorithmus (durchführen)	12	
7: Ein Suchbaumalgorithmus (C/C++, O-Notation)	25	

Es sind insgesamt 84 Punkte zu erreichen (80 Punkte werden als 100% gewertet)
(Jeder Punkt entspricht einer Bearbeitungszeit von etwa 1 Minute)

Aufgabe 1 — 6 Punkte

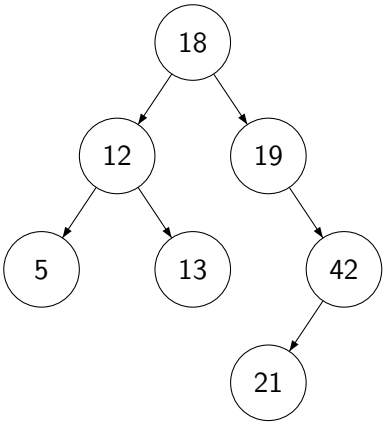
(2+4 = 6 Punkte) **Interpolationssuche:** Gegeben ist ein Array mit den Werten (5,8,x,15,20,23,29) mit einem unbekannten Wert x. Es soll der Wert x gesucht werden.

- (2 Punkte) Geben Sie die Formel an, mit der bei der Interpolationssuche der nächste zu betrachtende Index berechnet wird, verwenden Sie dabei `su` für das zu suchende Element, `a[]` für das Array sowie `kl` und `gr` für den kleinsten bzw. größten Index, der im Array noch zu betrachten ist. (Begründung ist nicht erforderlich)
- (4 Punkte) Geben Sie an, welche Werte das Element x im Array haben kann, damit dieses beim ersten Vergleich bereits gefunden wird (mit kurzer Begründung, 2 Sätze genügen)

Aufgabe 2 — 14 Punkte

(3+3+3+3+2 = 14 Punkte) **Ein Algorithmus auf Binärbäumen:**

Gegeben ist folgender Binärbaum:



und folgende Funktion:

```
void modify(BinBaumPtr root)
{
  if(root!=NULL)
  {
    modify(root->left);
    cout << root->value << ' '; // ein int-Wert
    BinBaumPtr swap=root->left;
    root->left=root->right;
    root->right=swap;
    modify(root->left);
  }
}
```

- (3 Punkte) Geben Sie die fehlende(n) C/C++-Datenstruktur(en) für die Funktion an, sodass diese vom Compiler übersetzt werden kann. Der Datentyp `BinBaumPtr` stehe dabei für einen Zeiger, der auf die Wurzel eines Binärbaums zeigt.
- (3+3 Punkte) Geben Sie an, welche Ausgabe (Zeile `cout`) der Algorithmus bei obigem Binärbaum liefert. Zeichnen Sie den Binärbaum, wie er durch den Ablauf der Funktion verändert wurde.
- (3 Punkte) Beschreiben Sie in 1-2 Sätzen, was die Funktion tut.
- (2 Punkte) Geben Sie den Aufwand der Funktion in O-Notation an.

Aufgabe 3 — 10 Punkte

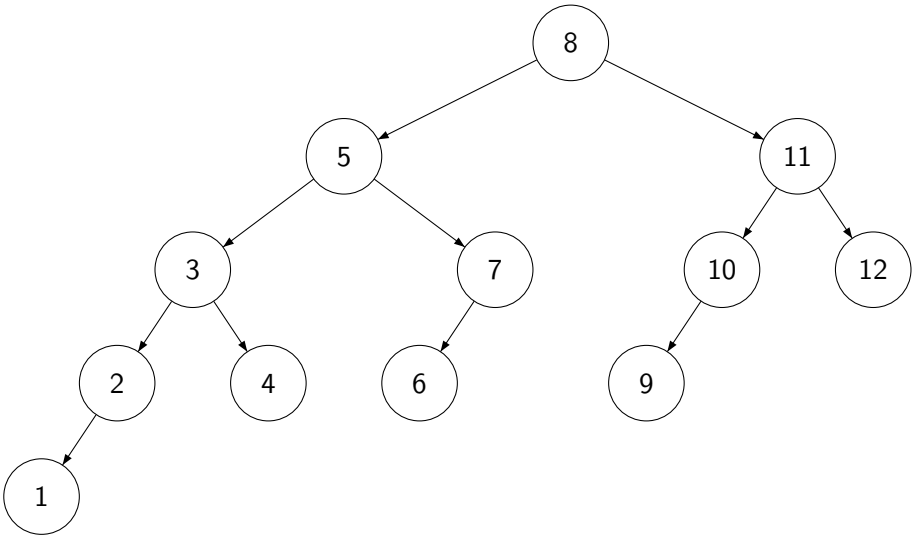
(8+2 Punkte) Fügen Sie in einen anfangs leeren **AVL-Baum** folgende Zahlen in dieser Reihenfolge ein: 1, 3, 5, 9, 7. Wenn eine Rotation notwendig ist, geben Sie den AVL-Baum jeweils vor und nach dieser Rotation an. Geben Sie in jedem Fall den AVL-Baum an, nachdem alle Elemente eingefügt wurden.

Welche der Zahlen 0, 2, 4, 6 und 8 lassen sich in den Baum als sechstes Element einfügen, ohnedass eine Rotation dabei erfolgt?

Aufgabe 4 — 5 Punkte

(5 Punkte) **Löschen in AVL-Bäumen:**

Gegeben ist folgender AVL-Baum:



Löschen Sie in diesem AVL-Baum die 12, geben Sie den Baum nach jeder dabei notwendigen Rotation an. Unterbäume, die sich dabei nicht ändern, brauchen nicht komplett neu gezeichnet zu werden.

Aufgabe 5 — 12 Punkte

(12 Punkte) **Heapsort:** Gegeben ist ein Array mit den Zahlen (9,7,4,6,3,1,2).

Bei Heapsort wird ein Max-Heap verwendet (größtes Element steht „oben“), der in der Phase des Heapaufbaus hergestellt wird.

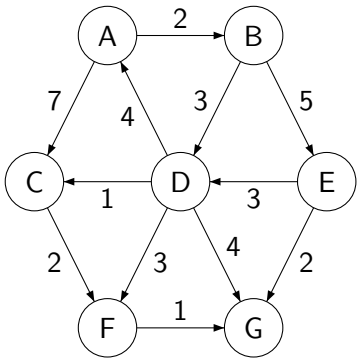
(3 Punkte) Überprüfen Sie, ob das oben gegebene Array die Heapeigenschaft hat. Geben Sie an, welche konkreten Vergleiche dabei durchgeführt werden (z. B. $6 \geq 4$ oder ähnlich, eine allgemeine Beschreibung ist nicht gefordert).

(9 Punkte) Führen Sie nun den Heapsort-Algorithmus durch. Gegen Sie dabei jeweils den Heap an, nachdem das jeweils größte Element mit dem jeweils letzten Element des Heaps vertauscht wurde und die Heapeigenschaft durch Absinkenlassen der neuen Wurzel wiederhergestellt wurde. (Es reicht also die Angabe eines Heaps mit 6 Elementen, dann ein Heap mit 5 Elementen, analog mit 4, 3, 2 und 1 Element. Sie können dabei die Baum-Darstellung wählen.)

Aufgabe 6 — 12 Punkte

(9+3 Punkte) **Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:**

Führen Sie den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten A mit d(A)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. (9 Punkte) Geben Sie in jeder Zeile den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an, den durch delete_min gewählten Knoten und die berechneten Entfernungen d() am Ende des Durchlaufs für alle Knoten.



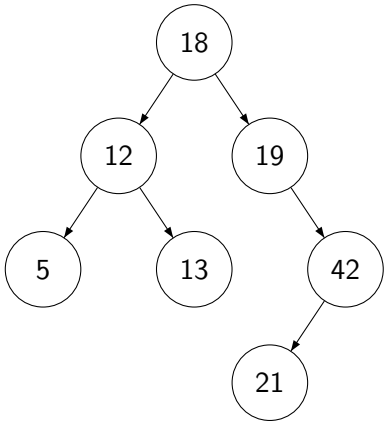
	Menge R	Ergebnis von delete_min(R)	berechnete Entfernungen A	B	C	D	E	F	G
0.	$\emptyset$	„Initialisierung“	0	∞	∞	∞	∞	∞	∞
1.	{A}								
2.									
3.									
4.									
5.									
6.									
7.									
8.									

(3 Punkte) Geben Sie den Kürzeste-Wege-Baum an, also den Baum, der durch die während der Berechnung gemerkten Vorgänger (`pred[]`) bestimmt ist.

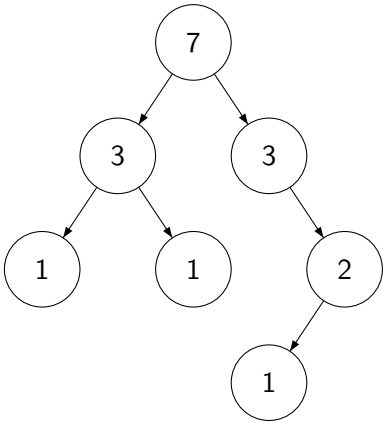
Aufgabe 7 — 25 Punkte

(2+8+10+2+3 = 25 Punkte) **Ein Suchbaumalgorithmus:** In dieser Aufgabe soll zunächst zu einem gegebenen binären Suchbaum ein strukturgleicher Binärbaum erstellt werden, sodass jede Knotenbeschriftung der Anzahl der Knoten im jeweiligen Unterbaum entspricht. Mithilfe dieses so erstellten Baumes kann nun das k-kleinste Element im Suchbaum schnell ermittelt werden.

Ein Beispiel: zu dem Baum aus Aufgabe 2



wird zunächst der folgende Baum erstellt



Soll nun das 3-te Element im linken Suchbaum gefunden werden, so hat der linke Unterbaum bereits 3 Knoten, das gesuchte Element ist somit dort zu finden. Im Unterbaum mit Wurzel 12 sucht man also weiterhin das dritte Element; dieses findet man in dessen rechtem Unterbaum (der linke Unterbaum hat nur 1 Knoten, die Wurzel wäre der zweite Knoten). Es wird folglich im Unterbaum mit der Wurzel 13 nun das erste Element gesucht. Da der linke Unterbaum der 13 leer ist, ist das gesuchte Element die 13.

- (2 Punkte) Geben Sie eine C/C++-Datenstruktur für einen binären Suchbaum oder Binärbaum an. Der Datentyp für eine Variable, die auf die Wurzel eines solchen Baums zeigt, soll dabei `bbPtr` heißen.
- (8 Punkte) Schreiben Sie eine C/C++-Funktion `bbPtr anzahlKopie(const bbPtr root)`, die als Parameter den Zeiger auf die Wurzel des binären Suchbaums bekommt, der kopiert werden soll, und als Ergebnis den Zeiger auf die Wurzel des erstellten Baumes zurückliefert. (Wenn Sie diese Aufgabe nicht komplett lösen können, schreiben Sie alternativ eine C/C++-Funktion, die in einem Binärbaum lediglich die Knotenbeschriftung so ändert, dass diese jeweils der Anzahl der Knoten im Unterbaum entspricht (bis zu 5 Punkte).)
- (10 Punkte) Schreiben Sie eine weitere C/C++-Funktion `int baumselect(const bbPtr root, const bbPtr anzKopie, const int k)`, die als Parameter den Zeiger auf die Wurzel des binären Suchbaums bekommt, in dem gesucht werden soll, einen weiteren Zeiger auf die Wurzel des Binärbaums, der durch die obige Funktion `anzahlKopie` entstanden ist, sowie eine Zahl `k`. Rückgabewert der Funktion soll das nach oben beschriebener Methode gefundene Element sein (falls `k` zu klein oder zu groß ist, so sei das Ergebnis `-1`).
- (2 Punkte) Geben Sie den Aufwand Ihres Algorithmus `anzahlKopie(...)` in O-Notation an.
- (3 Punkte) Mit welchem Aufwand im Mittel würden Sie für einen Aufruf Ihrer Funktion `baumselect(...)` rechnen?

Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe eines Baumes kümmern.

Zusatzaufgabe (schwer): Man hätte die Knoten in der Kopie auch einfach in Inorderreihenfolge nummerieren können und somit einfach das `k` in der Kopie suchen können. Welchen Vorteil könnte das obige Vorgehen über die Anzahl der Knoten in den Unterbäumen haben?

Teilklausur II in Informatik – Lösungshinweise

Aufgabe 1 — Interpolationssuche

Gegeben war ein Array mit den Werten (5,8,x,15,20,23,29). Der gesuchte Wert x sollte dabei beim ersten Versuch gefunden werden.

- Bei **Interpolationssuche** wird jeweils am Index $m = kl + (su - a[kl]) \cdot (gr - kl)/(a[gr] - a[kl])$ gesucht.
- Damit $m = 2$ berechnet wird, muss $x = su \geq 2 * (29 - 5)/(6 - 0) + 5 = 13$ sein und $x = su < 3 * (29 - 5)/(6 - 0) + 5 = 17$. Da das Feld zusätzlich sortiert sein muss, kommen nur die Werte 13 und 14 in Frage.

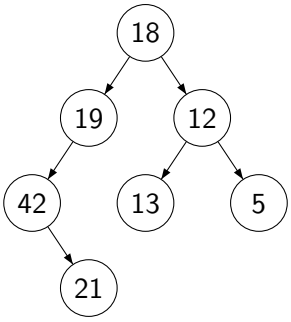
Aufgabe 2 — Ein Algorithmus auf Binärbäumen

Die Datenstruktur lautet z. B.

```
struct bb
{
  int value;
  bb *left, *right;
};
typedef bb *BinBaumPtr;
```

wobei der Name des `struct` beliebig gewählt sein kann. Die Namen der Selektoren und der Datentyp `int` ergeben sich aus dem Programm.

Ergebnis ist folgender Binärbaum:



Die Ausgabe des Programms lautet `5 12 13 18 19 21 42` – dies entspricht dem Inorderdurchlauf. Im Baum werden rekursiv der linke und rechte Unterbaum vertauscht (man beachte, dass nach dem Tausch der Zeiger auf die Unterbäume der zweite rekursive Aufruf mit `root->left` nun den anderen Unterbaum verändert). Als Ergebnis erhält man den gespiegelten Baum (dessen Inorderdurchlauf entspricht dem obigem von rechts nach links gelesen). Jeder Knoten des Baums wird einmal aufgerufen, das Tauschen der Kinder geschieht in konstanter Zeit, Gesamtaufwand damit $O(n)$, wenn n die Anzahl der Knoten im Baum ist.

Aufgabe 3 — Einfügen in AVL-Bäume

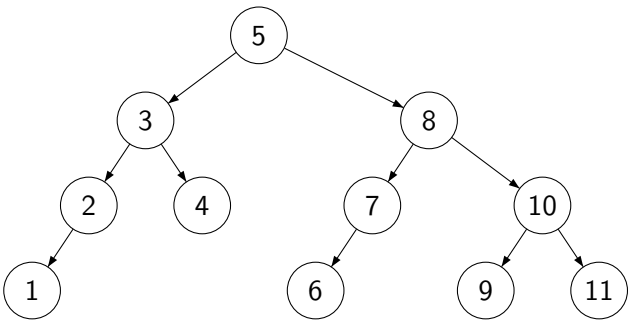
```
Werte: 1, 3, 5                    ( 1)_
Balance an Knoten 1 verletzt           \
Unterbaum vor der Rotation:          ( 3)_
                                          \
                                        ( 5)
```

```
Linksrotation                          _( 3)_
Baum nach der Rotation:               /      \
                                    ( 1)    ( 5)

Nächste Werte: 9, 7                 ( 5)_____
Balance an Knoten 5 verletzt                 \
Unterbaum vor der Rotation:                _( 9)
                                          /
                                       ( 7)

Rechtslinksrotation                 ( 5)_
Unterbaum nach der Rechtsrotation:       \
                                       ( 7)_
                                            \
                                           ( 9)

Baum nach der Linksrotation:           _( 3)_
                                      /      \
                                    ( 1)   _( 7)_
                                          /      \
                                       ( 5)    ( 9)
```

Die Werte 0 und 2 würden jeweils im linken Unterbaum landen und dort keine Höhenbalancen verletzen. Die Werte 4, 6 und 8 würden die Höhe des rechten Unterbaums der 3 auf 3 erhöhen, während der linke Unterbaum bei Höhe 1 verbleibt – hier wäre also eine Rotation notwendig.

Aufgabe 4 — Löschen in AVL-Bäumen

Beim Löschen der 12 wird zunächst eine Rechtsrotation an der 11 nötig (der Unterbaum hat die neue Wurzel 10 mit dem Kind 9 links und 11 rechts). Danach ist noch an der Wurzel die Höhenbalance verletzt und es muss erneut rechts rotiert werden.



Die umgekehrte Reihenfolge der Rotationen oder andere Umordnungen entsprechen nicht dem Algorithmus.

Aufgabe 5 — Heapsort

Das gegebene Array (9,7,4,6,3,1,2) ist bereits ein Max-Heap, innerhalb des Heapsort-Algorithmus testet man beim Aufbau die 3 Unterbaumwurzeln 4 (Vergleich der Kinder 1>2, Vergleich des größeren Kindes mit der Wurzel 4>2), 7 (6>3 und 7>6) sowie 9 (7>4 und 9>7). Ein reines Überprüfen der 3 Unterbaumwurzeln 4 (4>1 und 4>2), 7 (7>6 und 7>3) sowie 9 (9>7 und 9>4) verifiziert zwar ebenfalls die Heapeigenschaft, hilft aber nicht bei notwendiger Korrektur ohne weitere Vergleiche.

Im ersten Schritt wird die 2 zur neuen Wurzel, wird beim Absinken zunächst mit der 7 (7>4), dann mit der 6 (6>3) getauscht. Es entsteht der Heap (7,6,4,2,3,1), das Teilfeld (9) ist bereits sortiert.

Im zweiten Schritt wird die 1 zur neuen Wurzel, wird beim Absinken zunächst mit der 6 (6>4), dann mit der 3 (3>2) getauscht. Es entsteht der Heap (6,3,4,2,1), das Teilfeld (7,9) ist bereits sortiert.

Im dritten Schritt wird die 1 zur neuen Wurzel und wird beim Absinken mit der 4 (4>3) getauscht. Es entsteht der Heap (4,3,1,2), das Teilfeld (6,7,9) ist bereits sortiert.

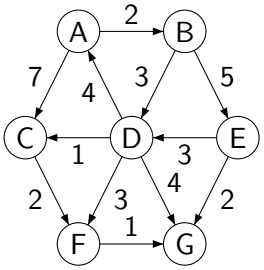
Im vierten Schritt wird die 2 zur neuen Wurzel und wird beim Absinken mit der 3 (3>1) getauscht. Es entsteht der Heap (3,2,1), das Teilfeld (4,6,7,9) ist bereits sortiert.

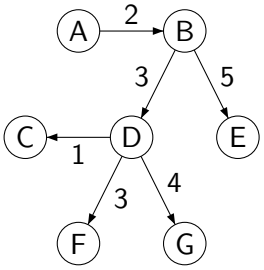
Im fünften Schritt wird die 1 zur neuen Wurzel und wird beim Absinken mit der 2 (einziges Kind) getauscht. Es entsteht der Heap (2,1), das Teilfeld (3,4,6,7,9) ist bereits sortiert.

Im letzten Schritt wird die 1 zur neuen Wurzel des nun einelementigen Heaps. Der Algorithmus ist beendet, das Feld (1,2,3,4,6,7,9) sortiert.

(Die zugehörige Baumdarstellung erhält man, indem die Elemente levelweise jeweils von links nach rechts in die Knoten eines Binärbaums eingetragen werden.)

Aufgabe 6 — Dijkstra-Algorithmus





	Menge R	Ergebnis von delete_min(R)	berechnete Entfernungen						
			A	B	C	D	E	F	G
0.	∅	„Initialisierung“	0	∞	∞	∞	∞	∞	∞
1.	{A}	A	0	2	7	∞	∞	∞	∞
2.	{B,C}	B	0	2	7	5	7	∞	∞
3.	{C,D,E}	D	0	2	6	5	7	8	9
4.	{C,E,F,G}	C	0	2	6	5	7	8	9
5.	{E,F,G}	E	0	2	6	5	7	8	9
6.	{F,G}	F	0	2	6	5	7	8	9
7.	{G}	G	0	2	6	5	7	8	9
8.	∅	„fertig“	0	2	6	5	7	8	9

Bei Gleichheit der Entfernungen wird der Vorgänger nicht(!) verändert!

Aufgabe 7 — Ein Suchbaumalgorithmus

Die Datenstruktur ist (bis auf das `typedef bb *bbPtr;`) identisch zu der in Aufgabe 2!

```
bbPtr anzahlKopie(const bbPtr root)
{
  if(root==NULL) return NULL;
  else
  {
    bbPtr node = new bb;
    node->left = anzahlKopie(root->left);
    node->right = anzahlKopie(root->right);
    int akl=0; if(node->left!=NULL) akl=node->left->value;
    int akr=0; if(node->right!=NULL) akr=node->right->value;
    node->value=akl+akr+1;
    return node;
  }
}
```

Alternativ wäre es auch möglich gewesen, die Anzahlbestimmung durch einen separaten Baumdurchlauf (erspart die zusätzlichen Abfragen auf `!=NULL`) zu bestimmen (identisch mit der Lösung der eingeschränkten Aufgabe):

```
int anz(bbPtr root)
{
  if(root==NULL) return 0;
  else
  {
    root->value=anz(root->left)+anz(root->right)+1;
    return root->value;
  }
}
```

Der Aufwand für `anzahlKopie(...)` ist – wie bei vielen Baumalgorithmen – linear (jeder Knoten wird einmal mit konstantem Aufwand betrachtet), also $O(n)$.

Bei der Selektionsaufgabe muss man sich in beiden Bäumen parallel bewegen. Zu beachten ist dabei, dass beim Auslesen der Anzahl der Knoten im linken Unterbaum dieser leer sein könnte. Im linken Unterbaum bleibt das `k` identisch, im rechten Unterbaum verringert es sich um die Anzahl der Knoten im linken Unterbaum plus 1 (für die Wurzel):

```
int baumselect(const bbPtr root, const bbPtr ak, const int k)
{
  if(ak==NULL || k<0 || k>ak->value) return -1;
  else
  {
    int akl=0; if(ak->left!=NULL) akl=ak->left->value;
    if(k<=akl) return baumselect(root->left,ak->left,k);
    else if(k==akl+1) return root->value;
    else return baumselect(root->right,ak->right,k-akl-1);
  }
}
```

Der Aufwand im Mittel entspricht dem mittleren Aufwand zum Suchen in einem binären Suchbaum, ist also logarithmisch (genauer: es werden im Mittel $1.386 \cdot \log(n)$ Ebenen des Baumes betrachtet).

Hinweis zur Zusatzaufgabe: die Inorderreihenfolge bringt zwar einen konstanten Faktor als Zeitgewinn, wenn jedoch der Suchbaum geändert wird, so lässt sich über den Ansatz der Anzahl der Knoten in den Unterbäumen die Datenstruktur schnell korrigieren (es wird z. B. beim Einfügen eines neuen Knotens auf demgleichen Pfad von der Wurzel zum neuen Blatt die Anzahl der Knoten in den Unterbäumen um jeweils 1 erhöht). Somit lässt sich ein Update der Datenstruktur in logarithmischer Zeit durchführen – beim Ansatz über die Inorderreihenfolge hätte man für jedes Update linearen Aufwand (im Mittel bekommen die Hälfte der Knoten neue Inordernummern)!

Mit Hilfe von z. B. AVL-Bäumen lässt sich dieses erwartete logarithmische Laufzeitverhalten auch garantieren. Das Vorgehen ist dabei identisch, beim Rotieren müssen lediglich die Anzahlen zusätzlich angepasst werden (Details selber überlegen, wie dies ohne komplette Neuberechnung möglich ist).