FernUniversität in Hagen

Lösungsvorschläge zur Nachklausur „1661 Datenstrukturen I“

18.9.2010

Aufgabe 1

(a)

algebra *dvd_collection*
sorts *card, movie, dvd, collection*

ops	*createMovie*	: *string* × *set*(*string*) × *card*	→ *movie*
	createDVD	: *int* × *set*(*movie*)	→ *dvd*
	createCollection	:	→ *collection*
	getName	: *movie*	→ *string*
	getDirectors	: *movie*	→ *set*(*string*)
	getRuntime	: *movie*	→ *int*
	insertDVD	: *collection* × *dvd*	→ *collection*
	containsMovie	: *collection* × *string*	→ *bool*
	singleMovies	: *collection*	→ *set*(*movie*)
	longDVDs	: *collection* × *card*	→ *set*(*dvd*)
	moviesOf	: *collection* × *string*	→ *set*(*movie*)
	avgRuntimeOf	: *collection* × *string*	→ *real*

Zusätzlich definieren wir uns noch:

	dvdContainsMovie	: *dvd* × *string*	→ *bool*

(b)

sets $card = \{x \in int \mid x \geq 0\}$
$movie = string \times set(string) \times card$
$dvd = card \times set(movie)$
$collection = \{ C \in set(dvd) \mid \forall x, x' \in C: x \neq x' \Rightarrow x.EAN \neq x'.EAN \}$

(c)

functions

$createMovie(name, directors, runtime) = (name, directors, runtime)$

$createDVD(EAN, movies) = (EAN, movies)$

$createCollection = \emptyset$

$getName((name, directors, runtime)) = name$

$getDirectors((name, directors, runtime)) = directors$

$getRuntime((name, directors, runtime)) = runtime$

$$insertDVD(coll, d) = \begin{cases} coll & \text{falls } \exists x : member(coll, x) \wedge x.EAN = d.EAN \\ insert(coll, d) & \text{sonst} \end{cases}$$

$dvdContainsMovie(\ (EAN, movies), name\ ) = |\ \{m \in movies \mid m.name = name\ \}\ | > 0$

$containsMovie(coll, mov) = |\ \{d \in coll : dvdContainsMovie(d, mov)\}\ | > 0$

$singleMovies(coll) = \{\ m \mid \exists\ d \in coll : size(d.movies) = 1 \wedge contains(d.movies, m)\ \}$

$longDVDs(coll, duration) = \{ d \in coll \mid \sum_{m \in d.movies} m.runtime > duration \}$

$moviesOf(coll, name) = \{ k \mid \exists d \in coll: contains(d.movies, k) \wedge contains(k.directors, name) \}$

$$avgRuntimeOf(coll, name) = \frac{\sum_{m \in moviesOf(coll, name)} m.runtime}{|moviesOf(coll, name)|}$$

Aufgabe 2

Für jede richtige Antwort erhalten Sie 0,25 Punkte. Für jede falsche einen entsprechenden Abzug!

(a)

Der Datentyp *Dictionary* dient der Darstellung von Mengen und stellt die Operationen *Insert* (Einfügen), *Delete* (Löschen) und *Member* (Find, Suchen, Enthält) zur Verfügung.

(b)

Mögliche und akzeptierte Implementierungen für *Dictionaries* aus dem Kurstext und deren worst-case-Laufzeiten sind:

Implementierung	insert	delete	member
Ungeordnete Liste mit Duplikateliminierung	O(1) O(n)	O(n) O(n)	O(n) O(n)
Geordnete Liste	O(n)	O(n)	O(n)
Sequentiell geordnete Liste im Array	O(n)	O(n)	O(log n)
Bitvektor	O(1)	O(1)	O(1)
Hashtabelle	O(n)	O(n)	O(n)
Binärer Suchbaum	O(n)	O(n)	O(n)
AVL-Baum	O(log n)	O(log n)	O(log n)
Vielweg-Suchbaum	O(n)	O(n)	O(n)
B-Baum	O(log n)	O(log n)	O(log n)

Aufgabe 3

Im Folgenden finden Sie die Berechnung der erforderlichen Hashwerte $h_i(i)$:

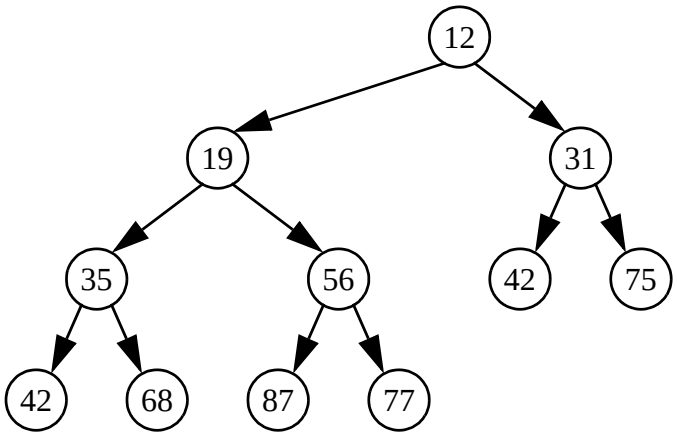
Name	Wert	h(x)	$h_1(x)$	$h_2(x)$	$h_3(x)$	$h_4(x)$
Karl	30	0				
Barbara	21	1				
Boris	35	5				
Anton	35	5	6			
Karla	30	0	1	4		
Katrin	32	2				
Peter	41	1	2	5	0	7
Diana	14	4	5	8		

Daraus ergibt sich folgende finale Hashtabelle:

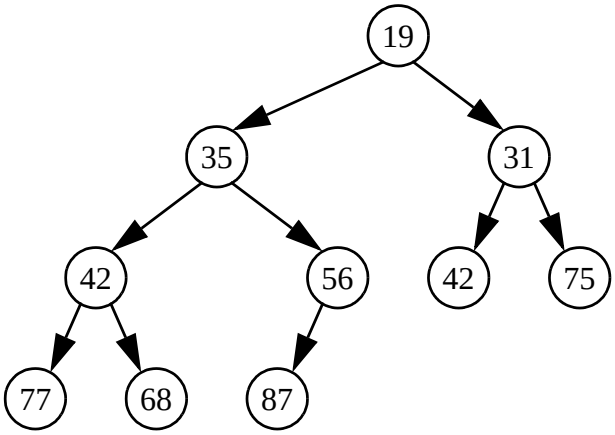
Behälternr.	Name	Behälternr.	Name
0	Karl	5	Boris
1	Barbara	6	Anton
2	Katrin	7	Peter
3		8	Diana
4	Karla	9	

Aufgabe 4

Der initiale Heap lautet:

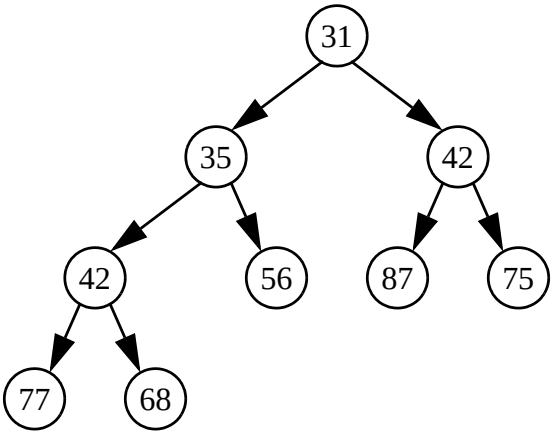


Nach Einfügen der 12 in die sortierte Folge und Einsinkenlassen der 77 ergibt sich folgendes Bild:



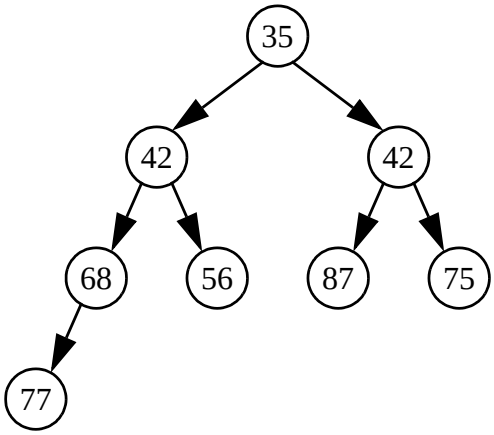
Folge: 12

Im nächsten Schritt erhält man:



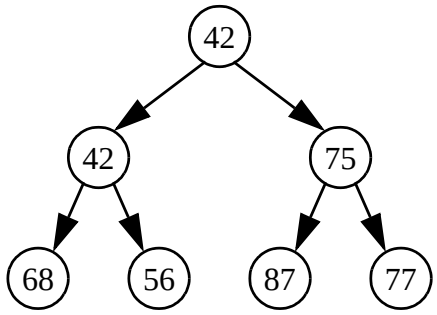
Folge: 19, 12

Nach Übertragen der 31 in die sortierte Folge haben wir:



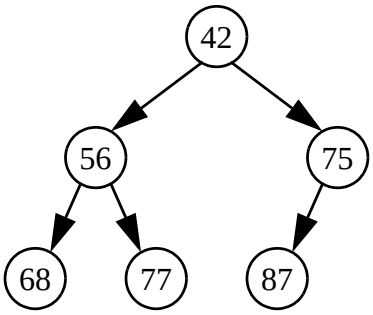
Folge: 31, 19, 12

Dann erhalten wir:



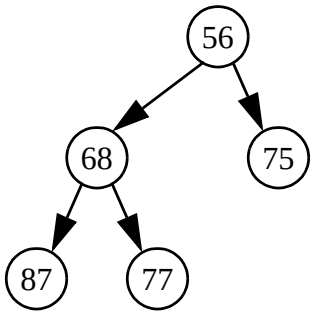
Folge: 35, 31, 19, 12

Nun ergibt sich:



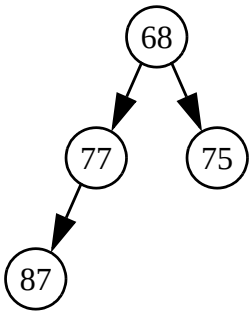
Folge: 42, 35, 31, 19, 12

Nach Verarbeitung der zweiten 42 erhalten wir:



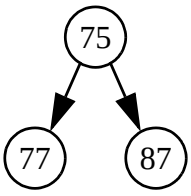
Folge: 42, 42, 35, 31, 19, 12

Danach wird die 56 in die sortierte Folge aufgenommen:



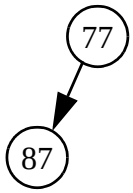
Folge: 56, 42, 42, 35, 31, 19, 12

Nach Einsinkenlassen der 87 ergibt sich folgendes Bild:



Folge: 68, 56, 42, 42, 35, 31, 19, 12

Nach dem nächsten Schritt sehen Heap und Folge wie folgt aus:



Folge: 75, 68, 56, 42, 42, 35, 31, 19, 12

Dann erhalten wir:

Folge : 77, 75, 68, 56, 42, 42, 35, 31, 19, 12

Damit endet der Algorithmus.

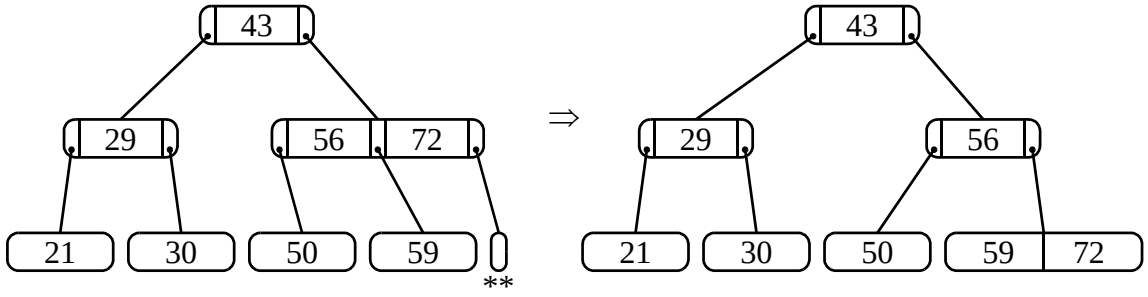
Aufgabe 5

(a)

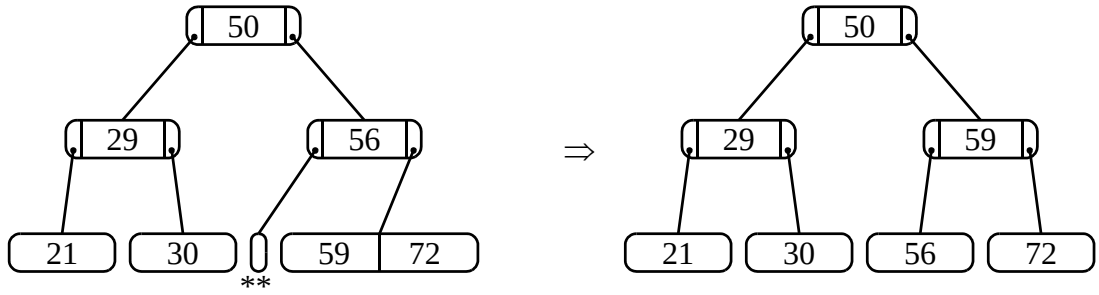
Der Baum ist von der Ordnung 1, denn es gibt in ihm einen Blattknoten, der nicht gleichzeitig Wurzel ist und nur einen Schlüssel enthält.

(b)

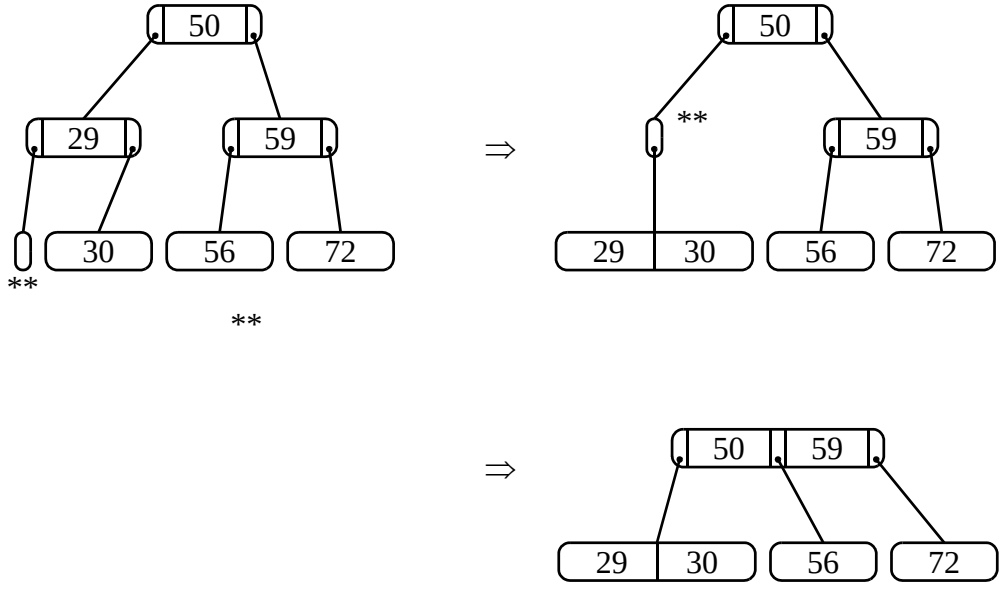
Beim Löschen von 80 tritt ein Unterlauf ein (*merge*):



Zum Löschen von 43 (Wurzel) vertauschen wir zunächst das kleinste Element, das größer ist als 43 aus dem entsprechenden Sohn – die 50 – mit dem zu löschenden Element. Dann löschen wir die 43 rekursiv aus dem entsprechenden Teilbaum. Dabei wird eine Rebalancierung erforderlich (*balance*):

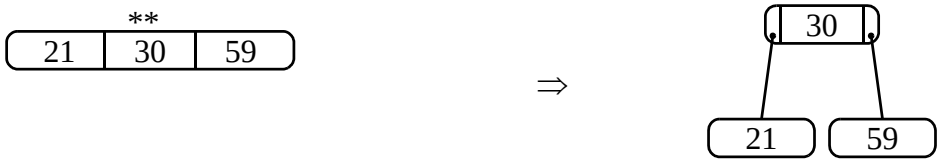


Nun löschen wir die 21. Es entsteht ein Unterlauf, der sich bis in die Wurzel fortsetzt (Behandlung durch zweimaliges *merge*):

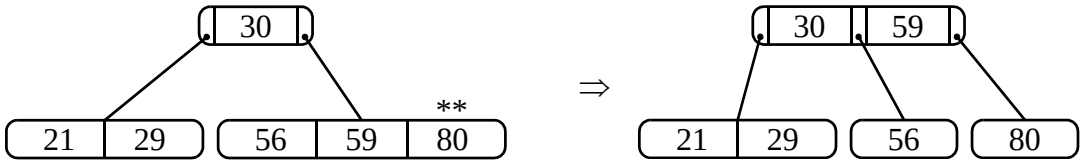


(c)

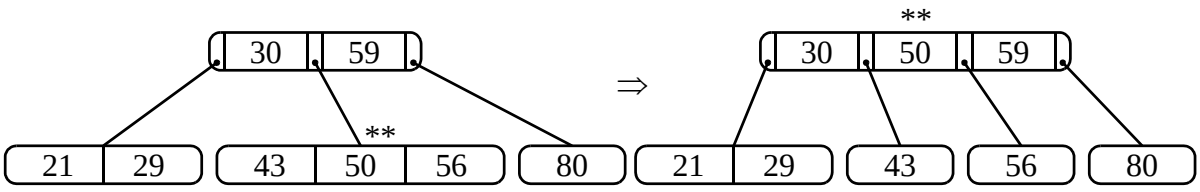
Die Schlüssel 21 und 59 werden direkt in die Wurzel eingefügt. Beim Einfügen von 30 wird eine Overflow-Behandlung erforderlich (Betroffene Knoten werden mit ** markiert):

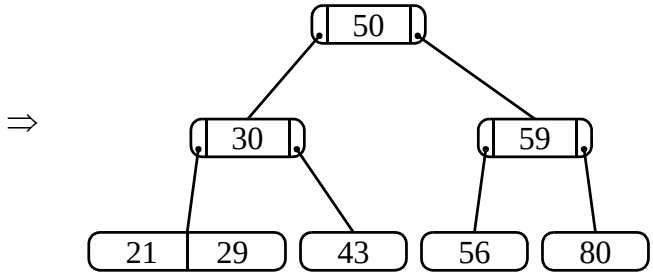


29 und 56 können leicht eingefügt werden. Der nächste Overflow tritt beim Einfügen von 80 auf:

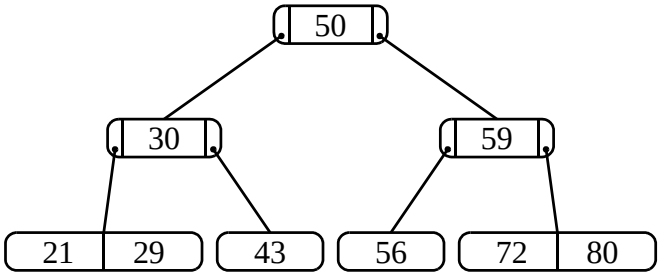


Weiter geht es mit dem Einfügen von 50, beim Einfügen von 43 tritt ein sich nach oben fortsetzender Überlauf auf (erst im mittleren Blatt, dann in der Wurzel):





Das Einfügen von 72 bereitet keine Probleme mehr. Wir erhalten:



Aufgabe 6

(a)

Es wird das Verfahren des parallelen Durchlaufs verwendet. Dabei werden beide Mengendarstellungen etwa in aufsteigender Schlüsselfolge durchlaufen.

Man positioniert je einen Zeiger auf den jeweils kleinsten in jeder Menge enthaltenen Schlüssel. Die Zeiger referenzieren den jeweils „aktuellen“ Schlüssel in beiden Mengen.

Wiederhole, solange beide Mengen noch nicht-behandelte Schlüssel enthalten: Vergleiche beide referenzierten Schlüssel. Sind beide gleich, so gib den Schlüssel einmal aus und rücke beide Zeiger vor zum jeweils nächstgrößeren Schlüssel. Ist ansonsten einer der beiden aktuellen Schlüssel kleiner als der andere, so lasse den entsprechenden Zeiger auf den nächstgrößeren Schlüssel vorrücken (Alternativ, falls dies die Darstellung unterstützt: Rücke zum kleinsten Schlüssel in dieser Menge vor, der größer oder gleich dem aktuellen Schlüssel der anderen Menge wird).

Das Verfahren hat eine worst-case Zeitkomplexität von $O(n + m)$, wobei n und m die Kardinalitäten der beiden Eingabemengen sind. Begründung: Generell muss man bei sequentiellem Durchlaufen einer Menge jedes Element einmal betrachten. Daraus ergibt sich direkt die Laufzeit $O(n + m)$.

Falls die Datenstruktur „Sprünge“ erlaubt, etwa weil sie hierarchisch organisiert ist (etwa beim B-Baum oder anderen Suchbäumen), kann man zumindest näherungsweise im average-case auch $O(e)$ erzielen, wobei e die Kardinalität der Ergebnismenge ist. Dies gilt insbesondere, wenn e im Vergleich zu n und m klein ist. Begründung: „Tote Bereiche“ zwischen zwei Schlüsseln kann man auf einer der oberen Hierarchieebenen schnell überspringen, ohne darunterliegende Schlüssel inspizieren zu müssen. Es ergeben sich jedoch trotz allem mindestens $e = O(\max(n, m)) = O(n + m)$ Halte bei übereinstimmenden Schlüsseln.

(b)

Man verwendet einen Stack (Stapel-/Kellerspeicher) oder eine vergleichbare Datenstruktur, etwa $list_2$ aus dem Kurstext.

Die Platzkomplexität ist sowohl im best-case, worst-case, also auch average-case immer $O(\log_b n) = O(\log n)$, da maximal ein kompletter Pfad von der Wurzel zu einem Blatt enthalten ist.

(c)

Der Iterator verwendet als interne Datenstruktur einen Stack. Dieser verwaltet die als nächstes noch zu besuchenden Schlüssel für jede Ebene oberhalb der gerade betrachteten in Form von Paaren (*Knotenreferenz*, *Schlüsselnummer*).

Bei der Initialisierung des Iterators durch *init*(t) laufen wir von der Wurzel des B-Baums t bis zum Blatt mit dem kleinsten Schlüssel (ganz links außen) und legen Paare (s, 1) für alle dabei erreichten oder passierten Knoten in Besuchsreihenfolge auf den Stack; s ist dabei eine Referenz auf den jeweiligen Knoten im B-Baum t, 1 die Schlüsselnummer des in diesem Knoten als nächstes zu besuchenden Schlüssels (nämlich des ersten). Jeder Knoten enthält $h \in \{1, \ldots, 2b\}$ Schlüssel mit den Indizes 1 bis $2b$ sowie $h + 1$ Teilbäume mit den Indizes 0 bis h.

next() funktioniert wie folgt: Falls der Stack leer ist, wird NULL zurückgegeben. Ansonsten wird das oberste Paar $p = (s, k)$ vom Stack geholt. Nun wird die Ausgabe zwischengespeichert: $out := s.key[k]$. Falls s noch größere Schlüssel als out enthält wird $(s, k + 1)$ auf den Stack gelegt. Dann läuft man im Teilbaum $s.sons[k]$ bis ins äußerste linke Blatt und legt für alle dabei besuchten Knoten s_i Einträge $(s_i, 1)$ in Besuchsreihenfolge auf den Stack. Nun wird noch out ausgegeben. Falls s keine größeren Schlüssel als out enthält, gib einfach out aus.

(d)

Verfahren: Wir verwenden einen parallelen Durchlauf durch beide B-Bäume. Dazu verwenden wir für jeden B-Baum einen in Teilaufgabe (c) beschriebenen Iterator. Für jeden B-Baum wird ein Iterator initialisiert und das in der Lösung zu Teilaufgabe (a) beschriebene Verfahren (ohne Sprünge) angewandt.

Platzaufwand für Hilfsstrukturen: Die Iteratoren verwenden Stacks mit einer Höhe, die der der B-Bäume entspricht, also etwa $O(\log_b n)$ und $O(\log_b m)$, zusammen $O(\log_b n + \log_b m) = O(\log n + \log m)$.

Zeitaufwand: Der Aufruf von *init* für die B-Bäume benötigt offensichtlich $O(\log_b n)$ und $O(\log_b m)$, zusammen $O(\log_b n + \log_b m)$ Zeit. Für jedes Element wird genau einmal *next*() aufgerufen und ein Schlüsselvergleich durchgeführt. Die Gesamtlaufzeit für alle *next*()-Aufrufe ist $O(n) + O(m)$, denn für jeden Schlüssel wird genau zweimal der Stack aktualisiert (in jeweils O(1) Zeit). Der Gesamtzeitaufwand für *intersection* ist also $O(n + m)$.

Aufgabe 7 Deckblatt

Hier erhalten Sie den Punkt, wenn Sie beide Klausurdeckblätter korrekt und vollständig ausgefüllt haben, also Namen, Matrikelnummer und Adresse korrekt eingetragen und genau diejenigen Aufgaben markiert haben, die Sie auch bearbeitet haben.