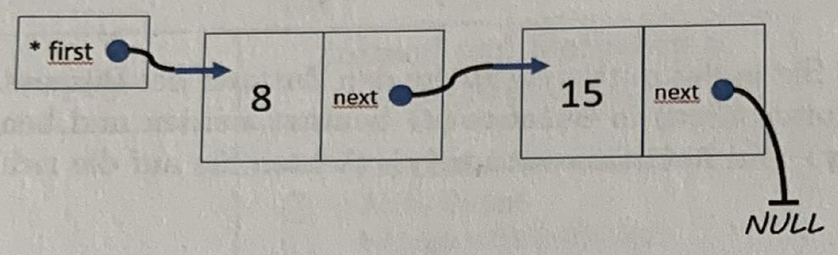
Aufgabe 1: Datenstrukturen Part 1 (/3 Punkte)

In der Vorlesung haben Sie verkettete Listen als Datenstruktur kennengelernt.

Es folgt eine Darstellungsweise für eine **einfach verkettete** Liste ohne zusätzliche Verwaltungselemente. Die Variable `*first` ist der Zeiger auf das erste Element `8`. Dieses Listenelement enthält einen Zeiger auf das nächste Element `15`, welches zugleich das letzte Element ist und demnach einen NULL Pointer beinhaltet.



Gegeben sei die Implementierung eines Stacks als einfach verkettete Liste mit Kopf (engl.: head) als zusätzlichem Verwaltungselement. Die Operation `push(new)` fügt das Element `new` *am Anfang der Liste* ein und aktualisiert die entsprechenden Zeiger. Die Operation `pop()` verhält sich gegensätzlich und entfernt das Element *am Anfang der Liste*.
Zeichnen Sie die Liste der Elemente, die nach der Ausführung der folgenden Operationen noch auf dem Stack liegen. Die Liste ist am Anfang leer.

- `push(new)` für Elemente in der Reihenfolge `20, 10, 42, 68, -30, 5`
- drei mal `pop()`
- `push(new)` für Elemente in der Reihenfolge `-20, 15, 9648, 0`
- zwei mal `pop()`

Kennzeichnen Sie im Verwaltungselement den Zeiger auf das erste Element der Liste (Top of Stack). Zeichnen Sie die Zeiger auf die weiteren Elemente ein. Ist ein Pointer NULL, zeichnen sie einen Pfeil zu NULL.

Aufgabe 2: Datenstrukturen Part 2 (/14 Punkte)

(a) (7 Punkte) In dieser Aufgabe sollen Sie in einen AVL-Baum Knoten hinzufügen und löschen. Dabei sollen die Operationen, wie in der Vorlesung vorgestellt, angewendet werden.

Die Operation soll, wie in dem unten gezeigten Lösungsmuster, auf dem Baum ausgeführt werden, der in der linken Spalte gegeben ist.

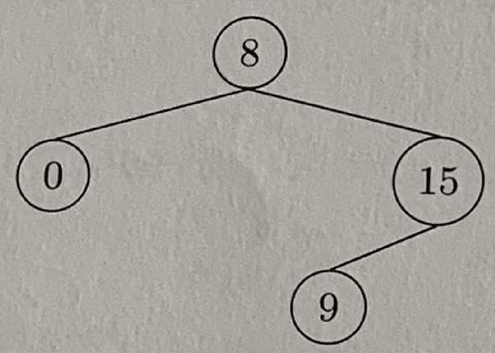
Linke Spalte: Zeichnen Sie zuerst in der linken Spalte die Änderung ein, unmittelbar bevor `Balance(x)` aufgerufen wurde.

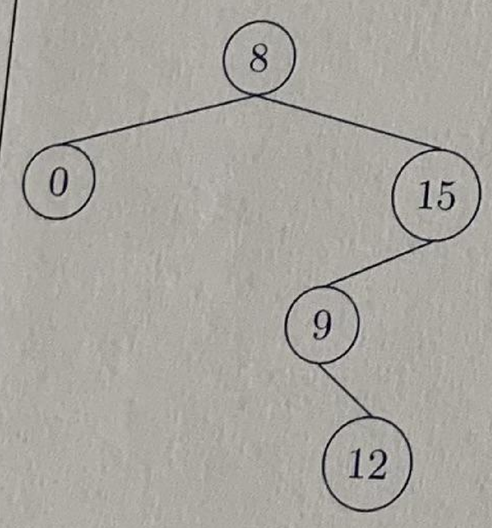
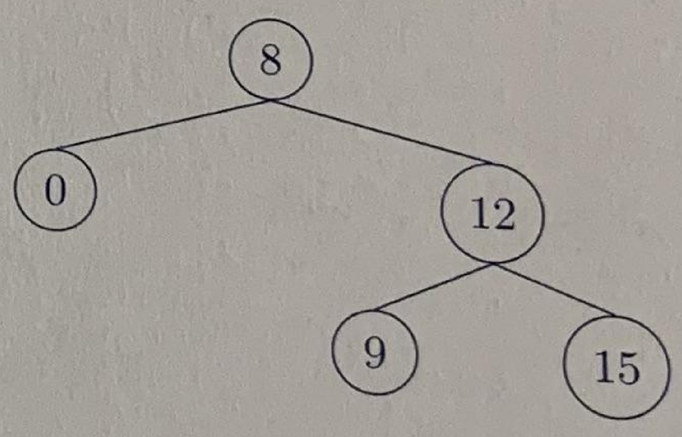
Mittlere Spalte: Geben Sie in der mittleren Spalte den Zustand des Baumes in der linken Spalte an. Ergänzen Sie welche Rotation(en) in `Balance(x)` benutzt werden und benennen Sie den Knoten y für `Linksrotation(y)` und `Rechtsrotation(y)`. Achten Sie auf die richtige Reihenfolge der Rotationen.

Rechte Spalte: Zeichnen Sie in die rechte Spalte den neuen Baum nach dem Aufruf `Balance(x)`, wenn dieser von der Zeichnung in der linken Spalte abweicht.

Lösungsmuster am Beispiel für die Operation `Einfügen(12)`:

gegebener Baum

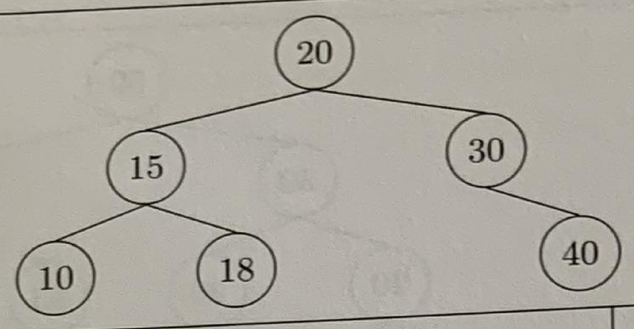


Baum vor Balance(x)	Zustand und Maßnahmen (vor Balance(x))	resultierender Baum (nach Balance(x))
	Der Baum ist ein ○ AVL-Baum. ⊗ Beinahe-AVL-Baum. ○ keine Rotation ○ 1. Rechtsrotation um ⊗ 1. Linksrotation um 9 ⊗ 2. Rechtsrotation um 15 ○ 2. Linksrotation um	

Die einzelnen Aufgaben finden Sie auf den Folgeseiten.

Führen Sie die Operation `Einfügen(35)` aus.

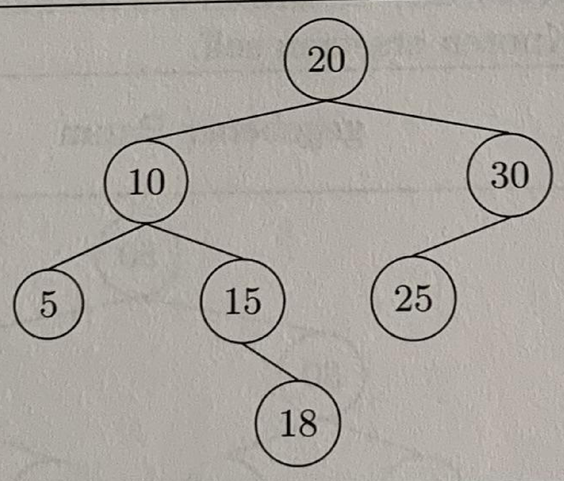
gegebener Baum



Baum vor Balance(x)	Zustand und Maßnahmen (vor Balance(x))	resultierender Baum (nach Balance(x))
	Der Baum ist ein ○ AVL-Baum. ○ Beinahe-AVL-Baum. ○ keine Rotation ○ 1. Rechtsrotation um ☐ ○ 1. Linksrotation um ☐ ○ 2. Rechtsrotation um ☐ ○ 2. Linksrotation um ☐	

Führen Sie die Operation `Einfügen(4)` aus.

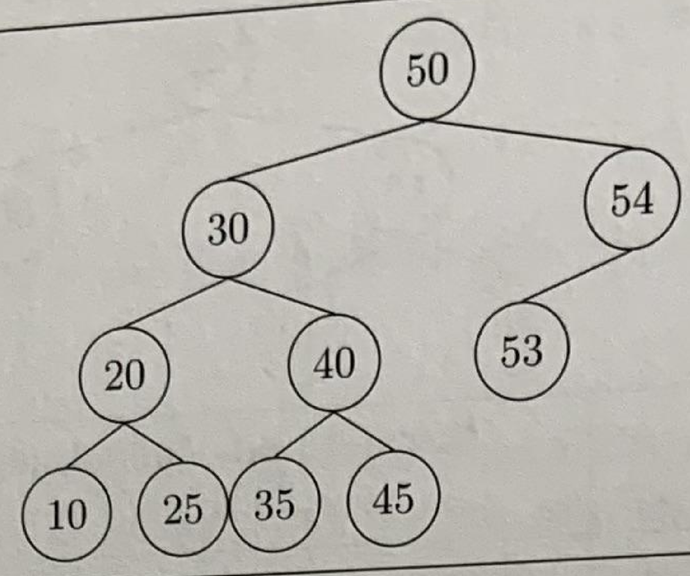
gegebener Baum



Baum vor Balance(x)	Zustand und Maßnahmen (vor Balance(x))	resultierender Baum (nach Balance(x))
	Der Baum ist ein ○ AVL-Baum. ○ Beinahe-AVL-Baum. ○ keine Rotation ○ 1. Rechtsrotation um ☐ ○ 1. Linksrotation um ☐ ○ 2. Rechtsrotation um ☐ ○ 2. Linksrotation um ☐	

Führen Sie die Operation `Löschen(50)` aus. Benutzen Sie die `Vorgängersuche`, um den Knoten zu ermitteln, der den zu löschenden Knoten ersetzen soll.

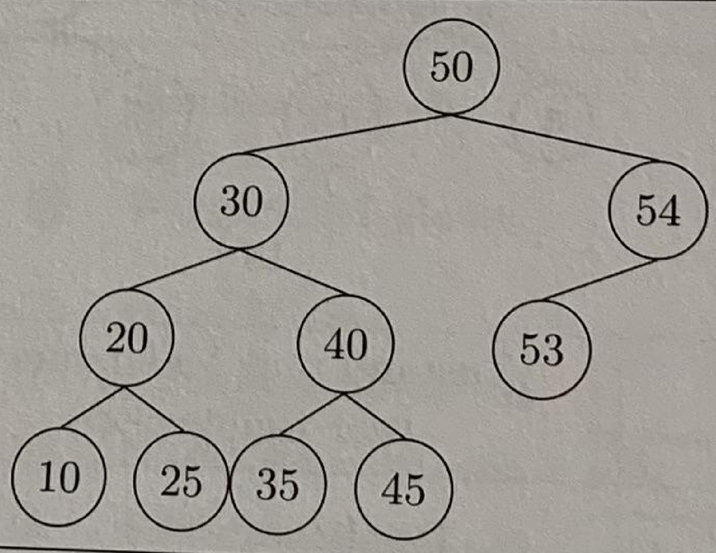
gegebener Baum



Baum vor Balance(x)	Zustand und Maßnahmen (vor Balance(x))	resultierender Baum (nach Balance(x))
	Der Baum ist ein ○ AVL-Baum. ○ Beinahe-AVL-Baum. ○ keine Rotation ○ 1. Rechtsrotation um ☐ ○ 1. Linksrotation um ☐ ○ 2. Rechtsrotation um ☐ ○ 2. Linksrotation um ☐	

Führen Sie die Operation `Löschen(50)` aus. Benutzen Sie die `Nachfolgersuche`, um den Knoten zu ermitteln, der den zu löschenden Knoten ersetzen soll.

gegebener Baum



Baum vor Balance(x)	Zustand und Maßnahmen (vor Balance(x))	resultierender Baum (nach Balance(x))
	Der Baum ist ein ○ AVL-Baum. ○ Beinahe-AVL-Baum. ○ keine Rotation ○ 1. Rechtsrotation um ☐ ○ 1. Linksrotation um ☐ ○ 2. Rechtsrotation um ☐ ○ 2. Linksrotation um ☐	

(b) (7 Punkte) Die in der Vorlesung vorgestellten Operationen auf AVL-Bäumen verändern den Baum immer um einen Knoten. Diese Bäume und deren Knoten verfügen über die in der Vorlesung vorgestellten Eigenschaften:

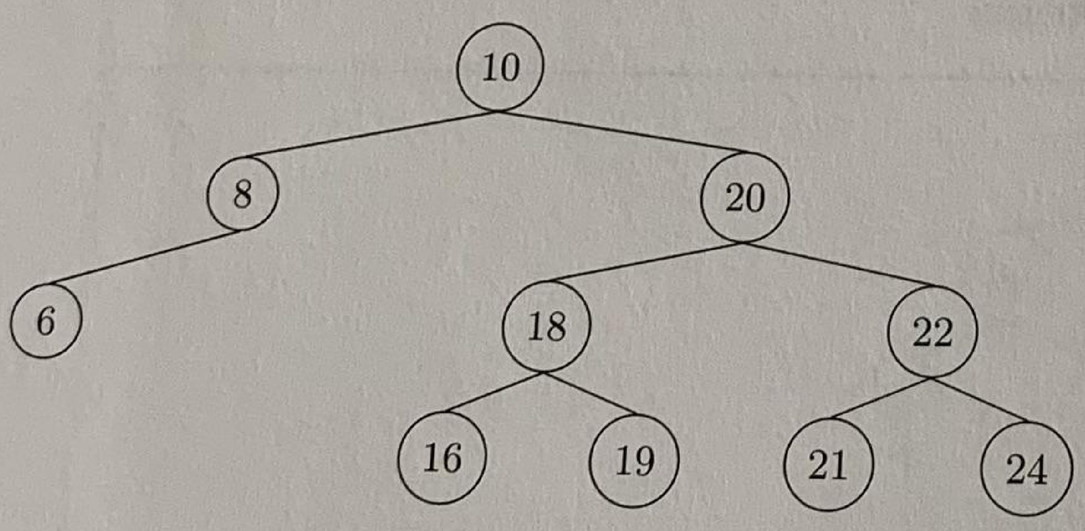
- $root[T]$ ist der Wurzelknoten eines (Teil)baumes T
- $h[x]$ ist die Höhe des (Teil)baumes mit der Wurzel x
- $rc[x]$ ist das rechte Kind von dem Knoten x, $lc[x]$ das linke
- $p[c]$ ist das Elter vom Kindknoten c

Gegeben seien weiterhin zwei AVL-Bäume T_1, T_2 mit den Eigenschaften:

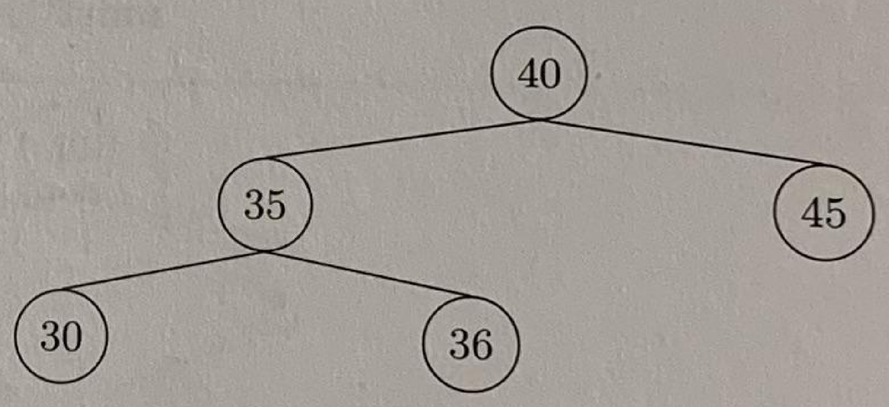
- $h[root[T_1]] > h[root[T_2]]$
- alle Elemente in T_2 sind streng größer als alle Elemente in T_1

Wir führen nun die Operation `Verketten` ein, die mit diesen Notationen definiert ist.

```
1  Verketten(s, t)
2      if h[s] > h[t]
3          if rc[s] = nil
4              rc[s] ← t
5              return
6          Verketten(rc[s], t)
7          return
8
9      b ← MinimumSuche(t)
10     Löschen(t, key[b])
11     rc[p[s]] ← b
12     lc[b] ← s
13     rc[b] ← t
14     h[b] ← 1 + max{h[lc[b]], h[rc[b]]}
15     t ← nil
16     return
```



Baum T_1



Baum T_2

Führen Sie nun die Operation `Verketten(`$root[T_1]$`,` $root[T_2]$`)` aus.

Geben Sie Ihre Antworten auf der nächsten Seite an.

Fortsetzung:

Zeichnen Sie den resultierenden Baum **bevor** die Balancierungsoperation ausgeführt wird.

Geben Sie hier den Zustand des Baumes vor `Balance(x)` an. Geben Sie ggf. den Knoten x die AVL-Baum Eigenschaft nicht mehr erfüllt ist, und durch die Operation `Balance(x)` wiederherstellt werden würde. Ergänzen Sie ggf. welche Rotation(en) bei Aufruf von `Balance(x)` benutzt werden. Benennen Sie den oder die Knoten y für `Linksrotation(y)` und `Rechtsrotation(y)` und achten Sie auf die richtige Reihenfolge der Rotationen.

Der Baum ist ein

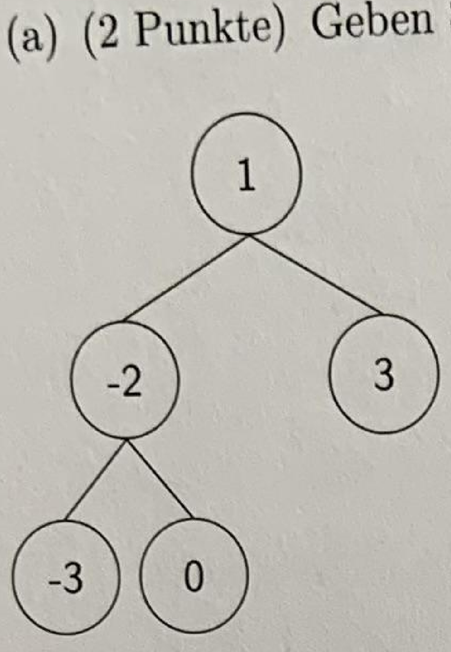
- ○ AVL-Baum.
- ○ Beinahe-AVL-Baum.

- ○ Balance mit ☐

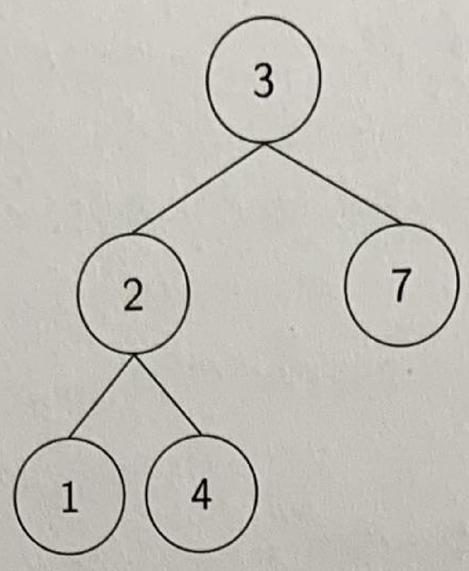
- ○ keine Rotation
- ○ 1. Rechtsrotation um ☐
- ○ 1. Linksrotation um ☐
- ○ 2. Rechtsrotation um ☐
- ○ 2. Linksrotation um ☐

Aufgabe 3: Binäre Suchbäume (/4 Punkte)

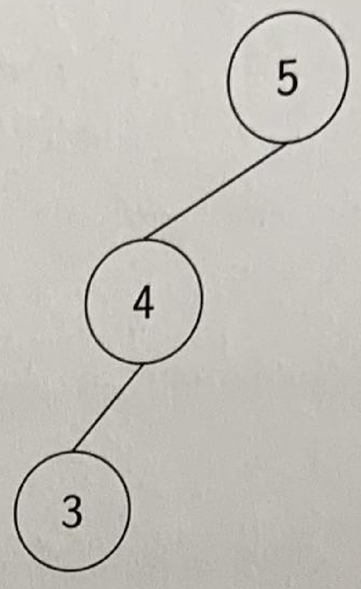
(a) (2 Punkte) Geben Sie an, ob es sich bei den folgenden Bäumen um binäre Suchbäume handelt.



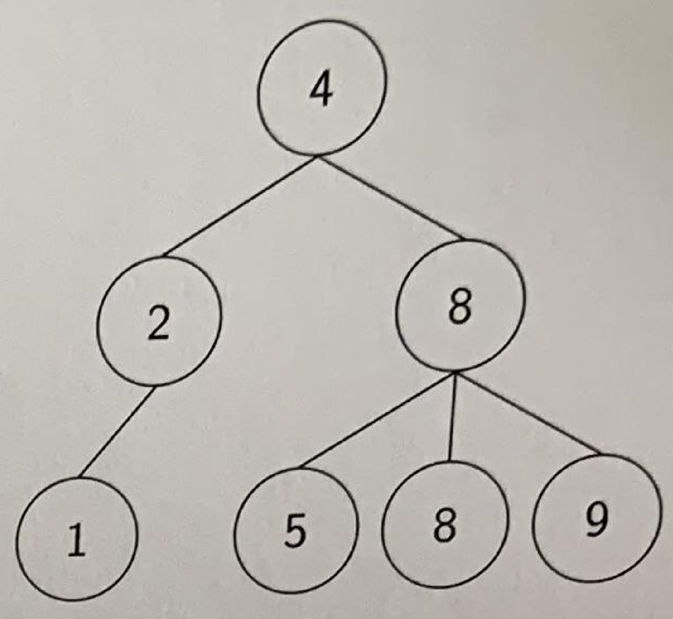
Baum A



Baum B



Baum C



Baum D

Baum	Baum ist ein binärer Suchbaum	
A	◯ Wahr	◯ Falsch
B	◯ Wahr	◯ Falsch
C	◯ Wahr	◯ Falsch
D	◯ Wahr	◯ Falsch

(b) (2 Punkte) Wir betrachten nun Baum A. Kreuzen Sie die Traversierung an, die zur jeweiligen Ausgabe führt.

Ausgabe	Traversierung
-3, -2, 0, 1, 3	◯ In-Order ◯ Pre-Order ◯ Post-Order ◯ Level-Order
1, -2, -3, 0, 3	◯ In-Order ◯ Pre-Order ◯ Post-Order ◯ Level-Order

Ausgabe	Traversierung
1, -2, 3, -3, 0	◯ In-Order ◯ Pre-Order ◯ Post-Order ◯ Level-Order
-3, 0, -2, 3, 1	◯ In-Order ◯ Pre-Order ◯ Post-Order ◯ Level-Order

Aufgabe 4: Binäre Suche (/6 Punkte)

Die binäre Suche findet den Index einer Zahl `target` in einem sortierten Array `A[low..high]`, sofern `target` in `A[low..high]` vorhanden ist. Im Folgenden finden Sie den Algorithmus für die binäre Suche in Pseudocode:

Hinweis: Die Operation $\lfloor$ `x` $\rfloor$ gibt die größte ganze Zahl an, die kleiner oder gleich `x` ist. Beispiel: $\lfloor 5.7 \rfloor = 5$.

```
1  BinäreSuche(A, target, low, high)
2
3      if low = high and A[low] = target
4          return low
5      else if low = high and A[low] != target
6          return -1
7      else
8          mid ← ⌊(low + high) / 2⌋
9          if target <= A[mid]
10             return BinäreSuche(A, target, low, mid)
11         else
               return BinäreSuche(A, target, mid + 1, high)
```

(a) (1 Punkt) Gegeben ist folgendes Array. Die Zahlen unter den Feldern stehen für die jeweiligen Arrayindizes. Welche Operation muss zuerst auf dem Array durchgeführt werden, damit die binäre Suche angewandt werden kann?

5	7	3	4	10	6	7	8	2
1	2	3	4	5	6	7	8	9

(b) (3 Punkte) Führen Sie schrittweise `BinäreSuche(A, 8, 1, len(A))` durch, um den Index der Zahl `8` im Array `A` zu finden. Die **1. Zeile** gibt das Array `A` an. Markieren Sie `low`, `mid` und `high` in jedem Schritt, indem Sie diese in die Felder der **2. Zeile** hineinschreiben. Der Anfangszustand ist vorgefüllt.

Hinweis: Auf der nächsten Seite finden Sie die weiteren Schritte.

2	3	4	5	6	7	7	8	10
low				mid				high
1	2	3	4	5	6	7	8	9

2	3	4	5	6	7	7	8	10
1	2	3	4	5	6	7	8	9

2	3	4	5	6	7	7	8	10
1	2	3	4	5	6	7	8	9

2	3	4	5	6	7	7	8	10
1	2	3	4	5	6	7	8	9

(c) *(1 Punkt)* Geben Sie in $\mathcal{O}$-Notation die asymptotische **Worst-Case**-Laufzeit der binären Suche auf einem sortierten Array der Länge `n` an.

$\mathcal{O}(\quad)$

(d) *(1 Punkt)* Sie möchten nun die binäre Suche zu einer **ternären** Suche ändern. Das heißt, Sie teilen bei jedem Schritt das Array in drei Teile, anstatt in zwei. Geben Sie in $\mathcal{O}$-Notation die asymptotische **Worst-Case**-Laufzeit der ternären Suche auf einem sortierten Array der Länge `n` an.

$\mathcal{O}(\quad)$

Aufgabe 5: Programmierung I

(/5 Punkte)

Wichtig: *Auch wenn Sie eine Programmieraufgabe nicht vollständig lösen können, schreiben Sie trotzdem das Grundgerüst vom Code. Funktionssignaturen, Teillösungen, etc. geben auch schon Punkte.*

Gegeben sei eine Funktion mit Signatur `bool is_fib(int x)`, welche `true` zurückgibt, wenn eine Eingabe-Ganzahl eine *Fibonaccizahl* ist, sonst `false`. Sie müssen diese Funktion **nicht implementieren**, aber Sie dürfen sie aufrufen.
Schreiben Sie eine Funktion in C, welche **die Anzahl** aller Fibonaccizahlen kleiner oder gleich dem `int n` zurückgibt. Alle Fibonaccizahlen sind größer oder gleich `0`. Sie können Integer-Überläufe ignorieren. `<stdbool.h>` wird importiert, d.h., Booleans stehen zur Verfügung. Außer der `is_fib` Funktion dürfen Sie nur Funktionen aufrufen, für die Sie auch eine Implementierung aufschreiben.

```
#include <stdbool.h> /* defines bool, true, false */
```

Aufgabe 6: Programmierung II (/7 Punkte)

Gegeben sei folgender Typ für Bäume:

```
1 /* Ein Baum wird als Pointer auf eine Node dargestellt.
2 Der NULL Pointer stellt den leeren Baum dar. */
3 typedef struct Node_ {
4     int item;
5     struct Node_ *left;
6     struct Node_ *right;
7 } Node;
```

Schreiben Sie eine Funktion in C, welche die Höhe eines Eingabebaums zurückgibt. Zur Erinnerung: die Höhe des leeren Baums ist `-1`, die Höhe eines ein-elementigen Baums ist `0`, etc.
Sie dürfen nur Funktionen aufrufen, für die Sie auch eine Implementierung aufschreiben.

Aufgabe 7: Programmierung III

(/6 Punkte)

Vervollständigen Sie folgende C-Funktion so, dass sie `true` genau dann zurückgibt, wenn im Eingabearray der Länge `len` zwei Zahlen vorkommen, deren Summe genau 40 ist.
Sie dürfen nur Funktionen aufrufen, für die Sie auch eine Implementierung aufschreiben.

```
#include <stdbool.h> /* defines bool, true, false */
#include <stddef.h> /* defines size_t*/

int two_sum_forty(int *arr, size_t len) {

}
```

Aufgabe 8: Mastertheorem **(/6 Punkte)**

Erinnerung – Mastertheorem Wenn ein Algorithmus sein Inputarray A der Länge n in a Teilarrays aufteilt, jeweils mit der Größe n/b, entsteht ein Rekursionsbaum der Höhe $\log_b(n)$. Sei $f(n)$ die Laufzeit vom Teilen und Zusammenfügen der Lösungen auf den Teilarrays. Das Mastertheorem (aus der Vorlesung) gibt in einer Fallunterscheidung eine obere Grenze für die Worst-Case asymptotische Komplexität der ganzen Rekursion, $T(n)$, an:

- Wenn die Laufzeit vom *Teilen und Zusammenfügen des ganzen Arrays* ($f(n)$) (oben im Rekursionsbaum) *dominiert*, $T(n) = O(f(n))$
- Wenn die Laufzeit der *Verarbeitung aller einelementigen Teilarrays* (unten im Rekursionsbaum) *dominiert*, $T(n) = O\left(n^{\log_b(a)}\right)$
- Wenn die Laufzeit der Verarbeitung aller *einelementigen Teilarrays* und vom Teilen und Zusammenfügen des *ganzen Arrays vergleichbar sind*, $T(n) = O\left(n^{\log_b(a)} \cdot \log(n)\right)$

Aufgabenstellung Es folgen Konfigurationen von $f(n)$, a und b. Geben Sie jeweils die asymptotische Worst-Case Komplexität des Teilens und Zusammenfügens des ganzen Arrays ($f(n)$), der Verarbeitung aller einelementigen Teilarrays B_1 bis B_m, und der ganzen Rekursion ($T(n)$).

(a) (2 Punkte) $f(n) = O(n^3)$, $a = 2$, $b = 2$.

Operation \ Aufwand	$O(n)$	$O(n \cdot \log(n))$	$O(n^2)$	$O(n^2 \cdot \log(n))$	$O(n^3)$	$O(n^3 \cdot \log(n))$
Teilen/Zusammenfügen von A	○	○	○	○	○	○
Verarbeitung von $B_1 \dots B_m$	○	○	○	○	○	○
ganze Rekursion	○	○	○	○	○	○

(b) (2 Punkte) $f(n) = O(n)$, $a = 4$, $b = 2$.

Operation \ Aufwand	$O(n)$	$O(n \cdot \log(n))$	$O(n^2)$	$O(n^2 \cdot \log(n))$	$O(n^3)$	$O(n^3 \cdot \log(n))$
Teilen/Zusammenfügen von A	○	○	○	○	○	○
Verarbeitung von $B_1 \dots B_m$	○	○	○	○	○	○
ganze Rekursion	○	○	○	○	○	○

(c) (2 Punkte) $f(n) = O(n^2)$, $a = 9$, $b = 3$.

Operation \ Aufwand	$O(n)$	$O(n \cdot \log(n))$	$O(n^2)$	$O(n^2 \cdot \log(n))$	$O(n^3)$	$O(n^3 \cdot \log(n))$
Teilen/Zusammenfügen von A	○	○	○	○	○	○
Verarbeitung von $B_1 \dots B_m$	○	○	○	○	○	○
ganze Rekursion	○	○	○	○	○	○

(/9 Punkte)

Aufgabe 9: Sortierung über Bins

Ein Inputarray A der Länge n wird in Arrays $B_1 \dots B_r$ — genannt Binarrays — aufgeteilt, indem jeweils der Wert $A[i]$ ins Binarray B_k für $k = A[i] \,\%\, r$ (Modulo-Operation) geschrieben wird. Daraufhin werden nacheinander $B_1 \dots B_r$ (in-place) mit MergeSort aufsteigend sortiert und zuletzt mit einer Merge-Operation zusammengefügt.

Part 1: Binarrays befüllen

(a) (1.5 Punkte) Was ist die asymptotische *Laufzeit* vom initialen *Aufteilen in Binarrays*, d.h., insgesamt vom Schreiben der $A[i]$ ins Binarray mit Index $A[i] \,\%\, r$ für alle $A[i]$? Kreuzen Sie die richtige Antwort an.

- ○ $O(r)$
- ○ $O(n \cdot \log_r(n))$
- ○ $O(n)$
- ○ $O(n^2)$

(b) (1.5 Punkte) Sei i ein Index im Input-Array A. Welche *Invariante* trifft *nach dem Schreiben* aller Werten bis $A[i]$ zu, wobei $A[i]$ selbst ausgeschlossen ist?

- ○ Für alle Indexe i' mit $0 \leq i' \leq i$ gilt, dass $A[i']$ im Binarray mit Index $A[i'] \,\%\, (r-1)$ gespeichert wurde.
- ○ Für alle Indexe i' mit $i \leq i' \leq n$ gilt, dass $A[i']$ im Binarray mit Index $A[i']$ gespeichert wurde.
- ○ $A[i]$ ist im Binarray mit Index $A[i]$ gespeichert.
- ○ Für alle Indexe i' mit $1 \leq i' \leq i-1$ gilt, dass $A[i']$ im Binarray mit Index $A[i'] \,\%\, r$ gespeichert wurde.

Part 2: Binarrays sortieren

(a) (3 Punkte) **Laufzeit pro Binarray.**
Gegeben sei ein Binarray B_j. Was ist die asymptotische *Worst-case-Laufzeit* davon, B_j zu sortieren?

- ○ $O(n \cdot \log(n))$
- ○ $O(r \cdot \log(r))$
- ○ $O(n^2)$
- ○ $O(1)$

Was ist die asymptotische *Best-case-Laufzeit* davon, B_j zu sortieren?

- ○ $\Omega(r \cdot \log(r))$
- ○ $\Omega(n/r \cdot \log(n/r))$
- ○ $\Omega(n)$
- ○ $\Omega(1)$

Angenommen, die Ergebnisse der Modulo-Operationen sind *gleichmäßig verteilt*, d.h., die Binarrays beinhalten alle die gleiche Anzahl an Elementen. Was ist dann die asymptotische *Worst-case-Laufzeit* davon, B_j zu sortieren?

- ○ $O(n/r \cdot \log(n/r))$
- ○ $O(r/n \cdot \log(r/n))$
- ○ $O(n^2 \cdot \log(n))$
- ○ $O(1)$

(b) (2 Punkte) **Laufzeit summiert über Binarrays.**
Die Anahme von gleichmäßiger Verteilung über Binarrays gilt nicht mehr. Was ist die *Worst-case gesamte Laufzeit*, wenn man nacheinander alle einzelnen Binarrays sortiert?

- ○ $O(n \cdot \log(n))$
- ○ $O(r \cdot \log(r))$
- ○ $O(n^2)$
- ○ $O(1)$

Was ist die *Best-case gesamte Laufzeit*, wenn man nacheinander alle einzelnen Binarrays sortiert?

- ○ $\Omega(r/n \cdot \log(r/n))$
- ○ $\Omega(n \cdot \log(n/r))$
- ○ $\Omega(n)$
- ○ $\Omega(n^2)$

Part 3: Binarrays zusammenfügen

(a) (1 Punkt) **Invariante beim Zusammenfügen.**
Gegeben sei ein Binarray B_j für $1 \leq j \leq r$. Seien $A_1 \dots A_r$ initial leere Arrays der Länge n (Länge von A), welche nacheinander wie folgt konstruiert werden: *1)* $A_1 = B_1$, *2)* A_1 und B_2 werden zusammengefügt (Merge-Operation) und das Ergebnis wird in A_2 geschrieben, *3)* A_2 und B_3 werden zusammengefügt und das Ergebnis wird in A_3 geschrieben, ... *r)* A_{r-1} und B_r zusammengefügt werden und das Ergebnis in A_r geschrieben wird. Welche *Invariante* trifft zu, nachdem A_1 bis A_j konstruiert wurden — A_j eingeschlossen?

- ○ Für jedes Index i' von 1 bis n gilt $A_r[i'] = A[i']$.
- ○ Alle Werte im Binarray B_r sind in A'.
- ○ Die Werte in Binarrays B_1 bis B_r sind alle in A_r und absteigend sortiert.
- ○ Die Werte in Binarrays B_1 bis B_j sind alle in A_j und aufsteigend sortiert.