

# 2024\_t1\_A

## ⚠ Disclaimer

Alles was hier drinnen steht kann Fehler enthalten!, Falls dir etwas auffällt melde dich gerne auf Discord bei mir ([@xmozz](#))

## A1 - Algorithmenanalyse

Stoff: [2. Analyse von Algorithmen](#)

a)

- a) (6 Punkte) Gegeben ist der nachfolgende Algorithmus. Dieser erhält folgende zwei Eingabeparameter:

- Das Array  $A$  welches mit **positiven ganzen Zahlen ohne 0** befüllt ist.
- Die Länge  $n$  des Arrays  $A$ .

**Function**  $f(A, n)$ :

```
for i = 0, ..., n - 1
  while A[i] < n
    A[i] ← A[i] + A[i]
```

Geben Sie die Best-Case und die Worst-Case Laufzeiten des Algorithmus in Abhängigkeit von  $n$  in  $\Theta$ -Notation an. Verwenden Sie möglichst einfache Terme.

Best-Case:

Worst-Case:

**Best-Case:**  $\Omega(n)$ , wenn die `while` -Schleife 0 mal läuft

**Worst-Case:**  $O(n \cdot \log(n))$ , wenn die `while` -Schleife maximal lang laufen muss, um das  $A[i]$  größer zu machen als das  $n$ .

b)

- b) (10 Punkte) Bestimmen Sie die Laufzeiten und Rückgabewerte der beiden unten angegebenen Algorithmen in Abhängigkeit von  $n \in \mathbb{N}$  in  $\Theta$ -Notation. Verwenden Sie möglichst einfache Terme.

$i \leftarrow n^4 \cdot 3^{(n^2)}$ $k \leftarrow 0$ <b>while</b> $i > 0$ $k \leftarrow k + 3i$ $i \leftarrow \lfloor \frac{i}{7} \rfloor$ <b>return</b> $k$	$a \leftarrow 0$ $b \leftarrow 0$ <b>for</b> $i = 1, \dots, n^2$ <b>for</b> $j = 1, \dots, i$ <b>if</b> $i \bmod 2 = 0$ <b>then</b> $a \leftarrow a + b$ $b \leftarrow b + 12 \frac{\log(i)}{i}$ <b>return</b> $\min(a, b)$
--	--

(i)

**Laufzeit:**  $\theta(\log_7(n^4 \cdot 3^{n^2})) = \theta(4 \cdot \log_7(n) + n^2 \cdot \log_7(3)) = \theta(n^2)$

**Rückgabewert:**

Die Variable  $k$  wird in jedem Durchlauf der `while`-Schleife um  $3i$  erhöht. Der initiale Wert von  $i$  ist  $i_0 = n^4 \cdot 3^{n^2}$ , und in jeder Iteration wird  $i$  durch 7 dividiert und abgerundet. Die Summe der Werte, die zu  $k$  addiert werden, ist:

$$k = 3i_0 + 3\lfloor i_0/7 \rfloor + 3\lfloor i_0/7^2 \rfloor + \dots + 3\lfloor i_0/7^m \rfloor$$

wobei  $m$  die Anzahl der Iterationen ist.

Wir können eine obere Schranke für  $k$  finden, indem wir die Abrundungsfunktion ignorieren und die geometrische Reihe summieren:

$$k < 3i_0 \left( 1 + \frac{1}{7} + \frac{1}{7^2} + \dots \right) = 3i_0 \cdot \frac{1}{1 - \frac{1}{7}} = 3i_0 \cdot \frac{7}{6} = \frac{7}{2}i_0$$

Die erste Addition zur Variablen  $k$  ist  $3i_0$ . Da alle Werte von  $i$  während der Schleife positiv sind, gilt  $k > 3i_0$ .

Da  $k$  sowohl nach oben als auch nach unten durch konstante Vielfache von  $i_0$  begrenzt ist, gilt für den Rückgabewert:

**Rückgabewert:**  $\Theta(n^4 \cdot 3^{n^2})$

(ii)

**Laufzeit:**

Die äußere Schleife läuft  $n^2$  Mal, und die innere Schleife läuft im schlimmsten Fall (wenn  $i = n^2$ ) ebenfalls  $n^2$  Mal. Die Gesamtzahl der Operationen ist proportional zur Summe der ersten  $n^2$  natürlichen Zahlen, was  $\frac{n^2(n^2+1)}{2}$  ist. Der dominante Term für große  $n$  ist  $n^4$ .

**Laufzeit:**  $\Theta(n^4)$

## Rückgabewert:

Die Variable  $b$  wird in jeder Iteration der äußeren Schleife einmal aktualisiert. Der Endwert von  $b$  ist die Summe  $\sum_{i=1}^{n^2} 12 \cdot \frac{\log(i)}{i}$ , welche durch das Integral  $\int_1^{n^2} \frac{\log(x)}{x} dx = 2(\log(n))^2$  approximiert werden kann. Die Variable  $a$  wird durch die Summe der  $b$ -Werte für gerade  $i$  beeinflusst und wächst wahrscheinlich schneller oder gleich schnell wie  $b$ . Der Rückgabewert ist  $\min(a, b)$ , und die langsamere Wachstumsrate dominiert.

**Rückgabewert:**  $\Theta((\log(n))^2)$

c)

c) (4 Punkte) Seien  $f$  und  $g$  Funktionen. Beweisen Sie die folgende Aussage:

$$\text{Wenn } f(n) = O(g(n)), \text{ dann } f(\sqrt{n}) = O(g(\sqrt{n})).$$

Der Anfang des Beweises ist bereits gegeben.

*Beweis.* Wir nehmen an, es gilt  $f(n) = O(g(n))$ . Daher gibt es  $c > 0$  und  $n_0 > 0$ , sodass ...

## Beweis:

Wir nehmen an, es gilt  $f(n) = O(g(n))$ . Nach Definition existieren positive Konstanten  $c > 0$  und  $n_0 > 0$ , sodass für alle  $n \geq n_0$  gilt:

$$f(n) \leq c \cdot g(n)$$

Wir wollen zeigen, dass  $f(\sqrt{n}) = O(g(\sqrt{n}))$ . Das bedeutet, wir suchen positive Konstanten  $c' > 0$  und  $n'_0 > 0$ , sodass für alle  $n \geq n'_0$  gilt:

$$f(\sqrt{n}) \leq c' \cdot g(\sqrt{n})$$

Betrachten wir  $f(\sqrt{n})$ . Die Ungleichung  $f(m) \leq c \cdot g(m)$  gilt für alle  $m \geq n_0$ . Setzen wir  $m = \sqrt{n}$ . Damit diese Substitution gültig ist, muss  $\sqrt{n} \geq n_0$  gelten, was äquivalent zu  $n \geq n_0^2$  ist.

Wählen wir nun  $n'_0 = n_0^2$  und  $c' = c$ . Dann gilt für alle  $n \geq n'_0$ :

$$f(\sqrt{n}) \leq c \cdot g(\sqrt{n}) = c' \cdot g(\sqrt{n})$$

Nach der Definition der Groß-O-Notation folgt daraus, dass  $f(\sqrt{n}) = O(g(\sqrt{n}))$ .

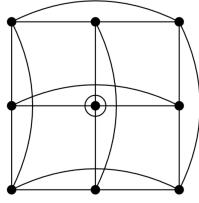
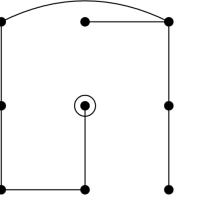
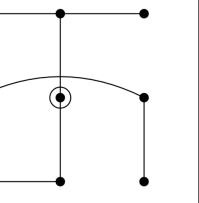
## A2 - Graphen

Stoff: [3. Graphen](#)

a)

- a) (4 Punkte) Betrachten Sie den linken Graphen und kreuzen Sie für jeden der beiden rechten Bäume an, ob es ein Breitensuchbaum (BFS) oder ein Tiefensuchbaum (DFS) des linken Graphen ist. Hierbei nehmen Sie an, dass der Startknoten der hervorgehobene Knoten in der Mitte ist.

(+1 Punkt für jede richtige, -1 Punkt für jede falsche und 0 Punkte für keine Antwort, keine negativen Gesamtpunkte auf diese Unteraufgabe)

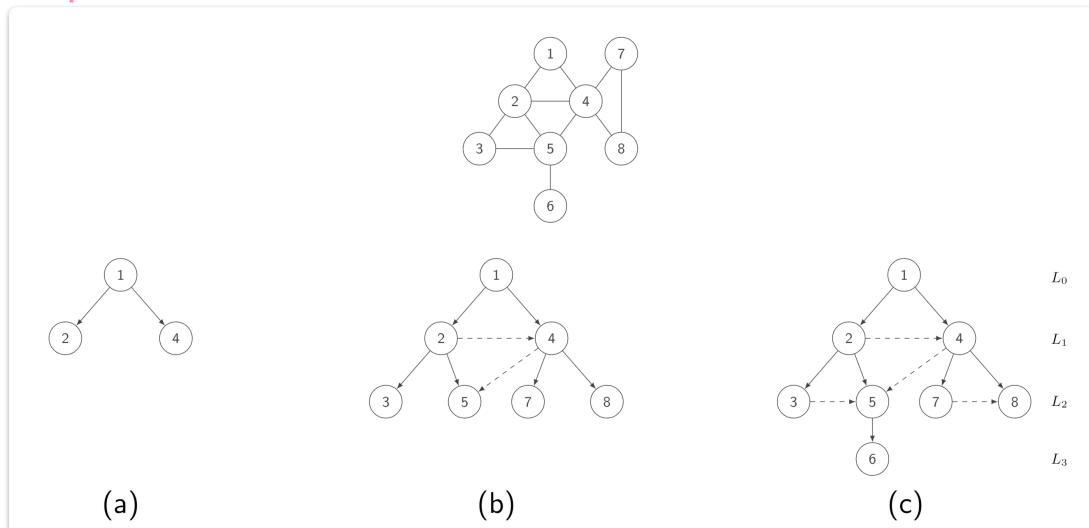
		
DFS	<input type="checkbox"/> Ja <input type="checkbox"/> Nein	<input type="checkbox"/> Ja <input type="checkbox"/> Nein
BFS	<input type="checkbox"/> Ja <input checked="" type="checkbox"/> Nein	<input type="checkbox"/> Ja <input type="checkbox"/> Nein

## BFS-Baum

- **BFS-Baum:** Breitensuche erzeugt einen Baum (BFS-Baum), dessen Wurzel ein Startknoten  $s$  ist und der alle von  $s$  erreichbaren Knoten beinhaltet.
- **Aufbau:** Man startet bei  $s$ . Wird nun ein Knoten  $v$  in der Ebene  $L_i$  gefunden, ist er zu mindestens einem Knoten  $u$  der Ebene  $L_{i-1}$  benachbart. Der Knoten  $u$  von dem aus  $v$  gefunden wird, wird als Elternknoten von  $v$  im BFS-Baum gemacht.

## Eigenschaft

- **Eigenschaft:** Sei  $T$  ein BFS-Baum von  $G = (V, E)$  und sei  $\{x, y\}$  eine Kante von  $G$ . Dann können die Ebenen der Knoten  $x$  und  $y$  höchstens um 1 unterschieden sein.
- **Beispiele:**

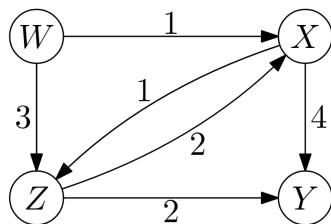


## Durchmusterung

- **Durchmusterung:** Durchmusterung bei DFS unterscheidet sich von der bei BFS.
  - Es wird zunächst versucht, möglichst weit vom Startknoten weg zu kommen.
  - Gibt es in der Nachbarschaft keine möglichen Knoten, dann wird durch rekursive Aufstiege bis zu einer möglichen Verzweigung zurückgegangen (Backtracking).

b)

- b) (6 Punkte) Führen Sie den Algorithmus von Dijkstra auf dem nachfolgenden Graphen mit  $W$  als Startknoten aus. Ergänzen Sie dabei für jeden Durchlauf der äußeren Schleife das Distanzarray  $d$  und den Knoten, der neu als **Discovered** markiert wurde in der unten angeführten Tabelle.



Discovered	$d(W)$	$d(X)$	$d(Y)$	$d(Z)$
W	0	1	$\infty$	3
X	0	1	5	2
Z	0	1	4	2
Y	0	1	4	2

c)

- c) (4 Punkte) Betrachten Sie folgende Operationen auf einem Min-Heap und auf einer einfach verketteten Liste. Kreuzen Sie in jeder Zeile an, auf welcher Datenstruktur die Operation die asymptotisch kleinere Worst-Case Laufzeit hat.

	Liste schneller	Heap schneller	gleich schnell
Einfügen	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Maximum löschen	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Suchen	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Minimum löschen	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Einfügen von Heap:  $O(\log n)$

Einfügen von Liste: am Ende:  $O(1)$  aber in der Mitte oder Anfang:  $O(n)$

Maximum:  $O(n)$  bei Liste

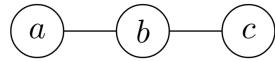
Maximum:  $O(n)$  bei Heap

Suchen: Beide  $O(n)$

Minimum löschen: Heap schneller mit  $O(\log n)$  wobei Liste wieder erstmal das Minimum in  $O(n)$  suchen muss.

d)

- d) (6 Punkte) Sei  $G$  ein ungerichteter **zusammenhängender** Graph. Wir sagen ein Knoten  $v$  ist ein *Engpass* von  $G$ , wenn es zwei verschiedene Knoten  $u, w$  mit  $u \neq v \neq w$  gibt, sodass jeder Pfad von  $u$  nach  $w$  durch  $v$  geht. Zum Beispiel ist im folgenden Graph  $b$  ein Engpass, aber  $a$  nicht.



Vervollständigen Sie folgenden Pseudocode eines Algorithmus, sodass dieser alle Engpässe von  $G$  berechnet. Nehmen Sie dafür an, dass die Methode *Komponenten*( $H$ ) eine Liste aller Zusammenhangskomponenten eines Graphen  $H$  liefert.

```

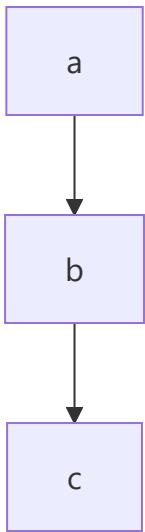
Engpässe( $G = (V, E)$ ):
     $S \leftarrow \emptyset$ 
    for each  $v \in V$ 
         $H \leftarrow$  
         $L \leftarrow \text{Komponenten}(H)$ 
        if 
             $S \leftarrow S \cup \{v\}$ 
    return  $S$ 

```

$H \leftarrow G$  ohne Knoten  $v$  und alle inzidenten Kanten

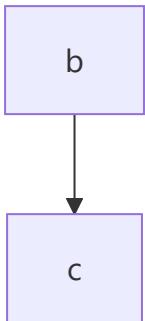
if Anzahl der Komponenten in  $L >$  Anzahl der Komponenten in  $G$  ohne  $v$ :

Wir schauen, ob der Graph nach aufteilen aus mehreren nicht zusammenhängenden Teilen besteht wie davor:



Hier gibt es eine Zusammenhangskomponente.

Wenn wir hier jetzt Knoten *a* entfernen haben wir immer noch eine Zusammenhangskomponente:



Bei der 2. Iteration entfernen wir dann *b*



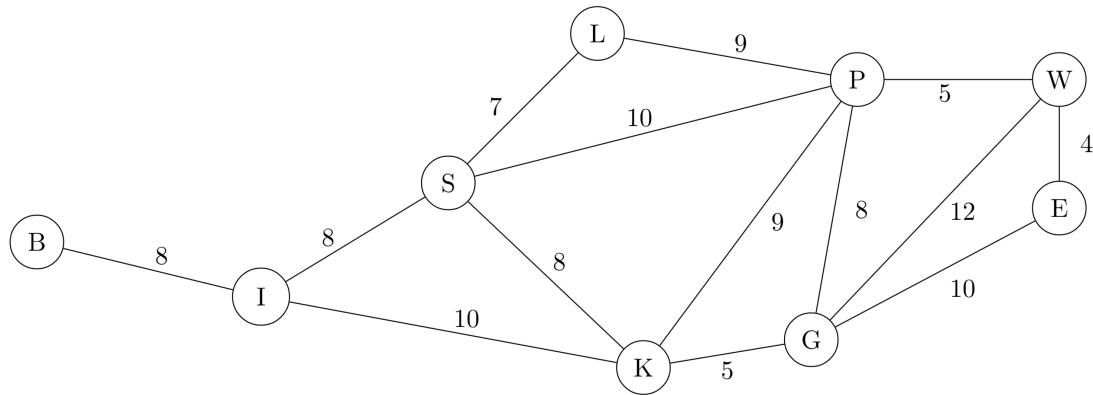
Hier haben wir jetzt 2 Zusammenhangskomponenten, daher fügen wir den Knoten *b* in unsere `return` Menge hinzu.

## A3 - Greedy

Stoff: [4. Greedy-Algorithmen](#)

**a)**

- a) (6 Punkte) Berechnen und markieren Sie einen minimalen Spannbaum des untenstehenden Graphen. Geben Sie eine mögliche Reihenfolge an, in der der Algorithmus von **Prim** die Kanten des Graphen für einen Spannbaum auswählt. Nehmen Sie an, dass der Algorithmus **mit dem Knoten W startet**. Geben Sie die Reihenfolge als Liste von Knotenpaaren an (z.B.: WG, ...). Bei mehreren gleichwertigen Möglichkeiten, nehmen Sie die lexikographisch alphabetisch kleinste Kante, wobei der schon im Spannbaum enthaltene Knoten als erster betrachtet wird. Wenn also z.B. die Knoten V und Z schon im Spannbaum sind und zwischen den Kanten VY und ZX entschieden wird, nehmen Sie VY).



**Reihenfolge:**

WE, WP, PG, GK, KS, SL, SI, IB

b)

- b) (8 Punkte) Kreuzen Sie an, ob die folgenden Aussagen wahr oder falsch sind.

(+2 Punkte für jede richtige, -2 Punkte für jede falsche und 0 Punkte für keine Antwort, keine negativen Gesamtpunkte auf diese Unteraufgabe)

- (Q1) Seien  $G_1 = (V, T_1)$  und  $G_2 = (V, T_2)$  zwei MSTs eines gewichteten Graphen  $G = (V, E)$ . Dann ist  $T_1 \cap T_2 \neq \emptyset$ .  
 **Wahr**  Falsch
- (Q2) Sei  $G$  ein gewichteter schlichter Graph, in dem alle Kantengewichte unterschiedlich sind, außer zwei, welche gleich sind. Dann enthält jeder MST beide jene Kanten, die gleiche Gewichte haben.  
 Wahr  Falsch
- (Q3) Sei  $G$  ein gewichteter Graph, in dem Kantengewichte möglicherweise gleich sind. Der Algorithmus von Prim liefert nicht immer den gleichen MST als Kruskal, aber beide liefern immer einen korrekten MST.  
 **Wahr**  Falsch
- (Q4) Sei  $G$  ein gewichteter Graph, in dem die Kante  $e$  das größte Gewicht hat und alle anderen Kanten strikt kleinere Gewichte haben. Dann ist  $e$  in keinem MST enthalten.  
 Wahr  Falsch

c)

- c) (6 Punkte) Ein Graph  $H = (V_H, E_H)$  ist ein induzierter Teilgraph eines Graphen  $G = (V_G, E_G)$ , wenn  $V_H \subseteq V_G$  und  $E_H = \{e \in E_G \mid e \subseteq V_H\}$ . D.h.,  $H$  ist eindeutig durch eine Teilmenge der Knoten von  $G$  spezifiziert.

Sei  $H = (V_H, E_H)$  ein zusammenhängender induzierter Teilgraph eines zusammenhängenden gewichteten Graphen  $G = (V, E)$  und sei  $G_T = (V, T)$  ein MST von  $G$ .

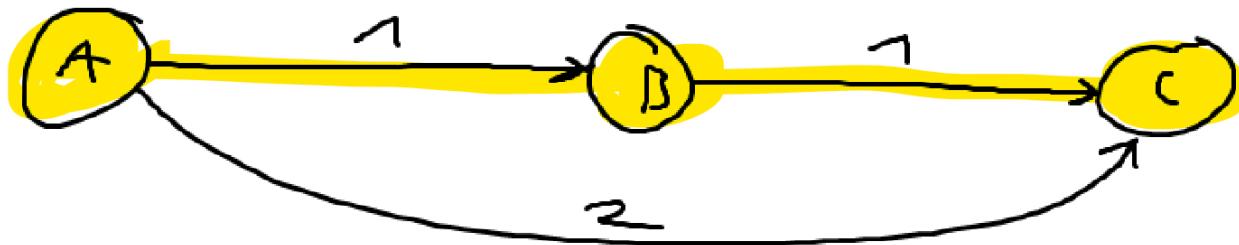
Ist  $H_T = (V_H, T \cap E_H)$  ein MST von  $H$ ? Begründen Sie Ihre Antwort mit einem Beweis/Gegenbeispiel.

Ja     Nein

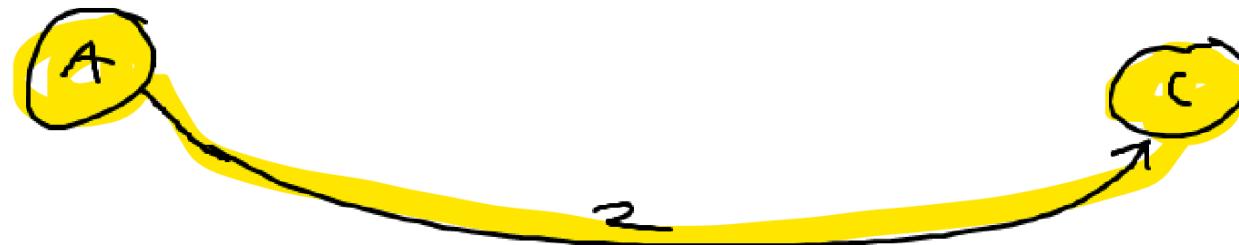
Begründung/Gegenbeispiel:

Begründung / Gegenbeispiel:

Graph mit MST



Teilgraph mit MST:



## A4 - Divide and Conquer

Stoff: 5. Divide and Conquer

In der folgenden Aufgabe ist  $A = [a_1, \dots, a_n]$  stets ein **aufsteigend sortiertes** Array von  $n$  ganzen Zahlen und  $x$  eine ganze Zahl. Nehmen Sie an, wir haben einen **Algorithmus**  $T(A, x)$  zur Verfügung, der den kleinstmöglichen Index  $i \in \{1, \dots, n\}$  zurückgibt, für den  $a_i = x$  gilt bzw.  $-1$  wenn  $x \notin A$ .

- a) Man bestimme, ob das Element  $x$  in  $A$  **mehr als**  $(n/2)$ -mal vorkommt. Nehmen Sie an, dass  $n$  **ungerade** ist. Nutzen Sie den Algorithmus  $T(A, x)$  für Ihre Lösung.
- (i) (8 Punkte) Vervollständigen Sie den folgenden Pseudocode so, dass er das obige Problem löst. Verwenden Sie **keine rekursiven Aufrufe**. Es soll **true** zurückgegeben werden, falls  $A$  und  $x$  die Eigenschaft des Problems erfüllen, andernfalls **false**.

**procedure MajorityAppearance( $A, x$ )**

//  $A$  ist ein aufsteigend sortiertes Array ungerader Länge

$n \leftarrow \text{length}(A)$

$i \leftarrow$   $T(A, x) + 1$

**if**  $i >$   $x$  **then**

**return false**

**if**  $i + (\frac{n}{2}) \geq n$  **then**

**return false**

**if**  $A[i + (\frac{n}{2})] > n$  **then**

**return**  $\text{false}$

**else**

**return**  $\text{true}$

- (ii) (2 Punkte) Kreuzen Sie an, ob die folgende Aussage wahr oder falsch ist.

Ihr Algorithmus **MajorityAppearance** hat eine Worst-Case Laufzeit, die asymptotisch mit jener von  $T$  übereinstimmt.

Wahr  Falsch

b)

- b) (4 Punkte) Wir geben nun den Algorithmus T explizit an (der also den kleinstmöglichen Index  $i \in \{1, \dots, n\}$  zurückgibt, für den  $a_i = x$  gilt; oder  $-1$  zurückgibt, wenn  $x \notin A$ ):

```

procedure T( $A, x$ )
    //  $A$  ist ein Array aus aufsteigend sortierten Zahlen
     $n \leftarrow \text{length}(A)$  // (diese Zuweisung erfolgt in konstanter Laufzeit)
     $low \leftarrow 1$ 
     $high \leftarrow n$ 
     $result \leftarrow -1$ 

    while  $low \leq high$ 
         $mid \leftarrow \lceil (low + high)/2 \rceil$ 
        if  $A[mid] == x$  then
             $result \leftarrow mid$ 
             $high \leftarrow mid - 1$ 
        else if  $x < A[mid]$  then
             $high \leftarrow mid - 1$ 
        else
             $low \leftarrow mid + 1$ 
    return  $result$ 
```

Vervollständigen Sie die folgenden Beobachtungen durch Angabe von **möglichst engen** oberen bzw. unteren Schranken:

In O-Notation liegt die Worst-Case Laufzeit von T in:  $O(\log n)$

In  $\Omega$ -Notation liegt die Worst-Case Laufzeit von T in:  $\Omega(\log n)$ , da selbst wenn das zu findende Element genau in der Mitte liegt, der Algorithmus noch weiter macht bis er wegen der `while` Abbruchbedingung terminiert, da wir ja den kleinsten Index haben wollen.

c)

- c) (6 Punkte) Kreuzen Sie an, ob die folgenden Aussagen wahr oder falsch sind.

(+2 Punkte für jede richtige, -2 Punkte für jede falsche und 0 Punkte für keine Antwort, keine negativen Gesamtpunkte auf diese Unteraufgabe)

(Q1) Gegeben sei ein allgemeines (vergleichsbasiertes) Sortierverfahren. Müssen mit diesem  $n$  verschiedene Schlüssel sortiert werden, dann kann der Worst-Case Aufwand dafür bestenfalls in  $\Omega(n \log(n))$  liegen.

Wahr  Falsch

(Q2) Für den Algorithmus *Insertionsort* ist  $\Theta(n^2)$  seine Worst-Case Laufzeit und  $\Theta(n \log(n))$  seine Average-Case Laufzeit.

Wahr  Falsch

(Q3) Sei  $T(2^k)$ ,  $k \in \mathbb{N}$ , die Worst-Case Laufzeit von *Mergesort* bei Ausführung auf einer Liste der Länge  $2^k$ . Dann gilt folgender Zusammenhang:

$$\sqrt{T(2^{k+1})} \leq T(2^k) + T(2^k) + O(2^{k+1}).$$

Wahr  Falsch

Q3 hab ich nicht verstanden aber das wäre eine Erklärung dafür von Gemini:

Analyse von Aussage (Q3):

Die Aussage lautet:  $\sqrt{T(2^{k+1})} \leq T(2^k) + O(2^{k+1})$ , wobei  $T(2^k)$  die Worst-Case Laufzeit von Mergesort für eine Eingabe der Größe  $2^k$  ist.

Wir wissen, dass  $T(n) = \Theta(n \log n)$  für Mergesort. Daher gilt:

$$T(2^k) = c \cdot 2^k \cdot k$$

$$T(2^{k+1}) = c \cdot 2^{k+1} \cdot (k + 1) = 2c(k + 1)2^k$$

$O(2^{k+1}) \leq d \cdot 2^{k+1} = 2d \cdot 2^k$  für Konstanten  $c, d > 0$  und ausreichend großes  $k$ .

Setzen wir dies in die Ungleichung ein:

$$\sqrt{2c(k + 1)2^k} \leq c \cdot k \cdot 2^k + 2d \cdot 2^k$$

Vereinfachen:

$$\sqrt{2c(k + 1)} \cdot \sqrt{2^k} \leq (ck + 2d) \cdot 2^k$$

Dividiert durch  $\sqrt{2^k}$ :

$$\sqrt{2c(k + 1)} \leq (ck + 2d) \cdot \sqrt{2^k}$$

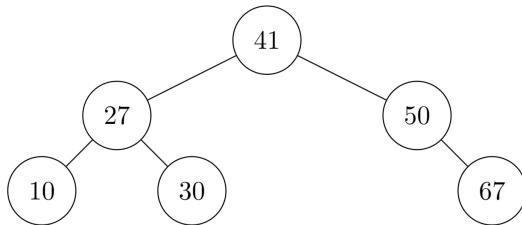
Da  $\sqrt{2^k}$  exponentiell wächst und  $\sqrt{2c(k + 1)}$  nur polynomiell, wird die rechte Seite für große  $k$  dominant. Der Term  $O(2^{k+1})$  stellt sicher, dass die rechte Seite auch für kleinere  $k$  nicht zu klein wird, um die Ungleichung zu verletzen.

**Fazit:** Die Aussage ist **wahr**.

## A5 - Suchbäume und Hashing

Stoff: [6. Suchbäume](#)

- a) (4 Punkte) Geben Sie die *Preorder*- und *Postorder*-Durchmusterungsreihenfolge des folgenden binären Suchbaumes an.



- **Inorder (Links - Wurzel - Rechts):**
  1. Durchlaufe den linken Teilbaum.
  2. Besuche die Wurzel.
  3. Durchlaufe den rechten Teilbaum. (*Ergebnis bei einem binären Suchbaum: die Knoten in sortierter Reihenfolge*)
- **Preorder (Wurzel - Links - Rechts):**
  1. Besuche die Wurzel.
  2. Durchlaufe den linken Teilbaum.

3. Durchlufe den rechten Teilbaum. (*Ergebnis: oft nützlich, um eine präfixartige Darstellung des Baumes zu erhalten*)

- **Postorder (Links - Rechts - Wurzel):**

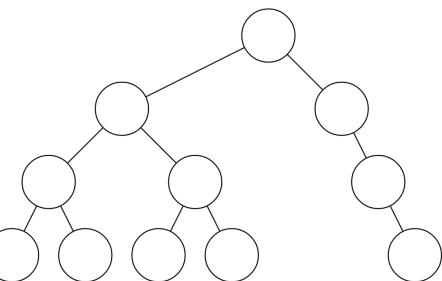
1. Durchlufe den linken Teilbaum.
2. Durchlufe den rechten Teilbaum.
3. Besuche die Wurzel. (*Ergebnis: oft nützlich bei der Auswertung von Ausdrücken oder beim Löschen von Bäumen*)

**Preorder:** 41, 27, 10, 30, 50, 67

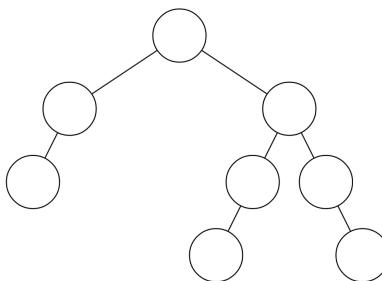
**Postorder:** 10, 30, 27, 67, 50, 41

b)

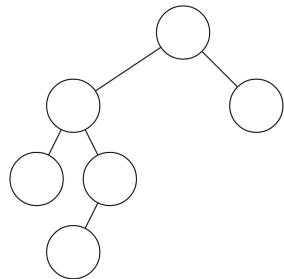
- b) (4 Punkte) Nehmen Sie an, die folgenden Bäume sind durch Einfügen eines Elementes in einen AVL-Baum entstanden. Kreuzen Sie bei jedem Baum an, ob keine, eine einfache, oder eine doppelte Rotation notwendig ist, um die AVL-Eigenschaft wiederherzustellen. Die Richtung der Rotation muss nicht angegeben werden.



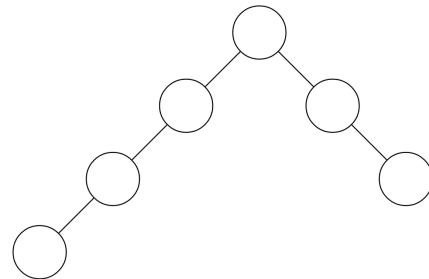
- (i)  keine  einfach  doppelt



- (ii)  keine  einfach  doppelt



- (iii)  keine  einfach  doppelt



- (iv)  keine  einfach  doppelt

c)

- c) (6 Punkte) Bestimmen Sie die Worst-Case Laufzeit der Operationen Suchen und Einfügen in natürlichen binären Suchbäumen, AVL-Bäumen und B-Bäumen abhängig von der Knotenzahl  $n$  in  $\Theta$ -Notation. Füllen Sie dazu die folgende Tabelle aus.

	Suchen	Einfügen
natürliche Suchbäume	$\Theta(h)$ , wobei im schlimmsten Fall $h = \Theta(n)$ ist	$\Theta(h)$ , wobei im schlimmsten Fall $h = \Theta(n)$ ist
AVL-Bäume	$\Theta(\log n)$	$\Theta(\log n)$
B-Bäume	$\Theta(\log_m n)$	$\Theta(\log_m n)$

**d)****Stoff: 7. Hashing**

- d) (6 Punkte) Gegeben ist eine Hashtabelle sowie die Hashfunktion  $h_1(k) = k \bmod 11$ .

Position	0	1	2	3	4	5	6	7	8	9	10
Schlüssel	22		101	14	15		61	40			87

Fügen Sie den Schlüssel 3 als neues Element mittels der jeweiligen Sondierungsvariante in die obige Hashtabelle ein. Geben Sie die Position an, an der der Schlüssel eingefügt wird. Geben Sie außerdem alle Positionen an, bei denen es zu einer Kollision kommt.

- (i) Double Hashing mit  $h(k, i) = (h_1(k) + ih_2(k)) \bmod 11$ , wobei  $h_2$  definiert ist als  $h_2(k) = 1 + (k \bmod 5)$ .

**Kollisionen:** 3, 7, 0, 4**Eingefügt an Position:** 8

- (ii) Lineares Sondieren mit  $h(k, i) = (h_1(k) + i) \bmod 11$ .

**Kollisionen:** 3, 4**Eingefügt an Position:** 5

- (iii) Quadratisches Sondieren mit  $h(k, i) = (h_1(k) + \frac{1}{2}i + \frac{1}{2}i^2) \bmod 11$ .

**Kollisionen:** 3, 4, 6**Eingefügt an Position:** 9