

2023_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)

(i)

```

 $k \leftarrow n$ 
 $a \leftarrow 1$ 
while  $k > 1$ 
    (i)      for  $i = 1, \dots, n$ 
             $a \leftarrow a + 3$ 
             $a \leftarrow a - 1$ 
             $k \leftarrow \frac{k}{4}$ 
return  $a$ 

```

Laufzeit: $\theta(n \log(n))$

Rückgabewert: $1 + 3n \cdot \log_4(n) - \log_4(n)$

(ii)

```

 $d \leftarrow n$ 
 $r \leftarrow 0$ 
while  $d > 1$ 
     $d \leftarrow \frac{d}{2}$ 
     $r \leftarrow r + 1$ 
 $d \leftarrow 0$ 
while  $r > 1$ 
     $r \leftarrow \frac{r}{2}$ 
     $d \leftarrow d + 1$ 
return  $d$ 

```

Laufzeit: $\theta(\log(n))$

Rückgabewert: $\theta(\log(\log n))$

b)

(8 Punkte) Gegeben ist der folgende Algorithmus, der ein Array A und dessen Länge $n \geq 1$ als Eingabe erhält:

```

Function  $f(A, n)$ :
    while  $A[0] = 0$ 
         $i \leftarrow n - 1$ 
        while  $i > 0$  and  $A[i] = 0$ 
             $i \leftarrow i - 1$ 
        if  $i = 0$  then
            return
         $A[i] \leftarrow 0$ 
        do
             $i \leftarrow i - 1$ 
        while  $A[i] \neq 0$ 
         $A[i] \leftarrow 1$ 

```

(i)

Angenommen, der Algorithmus erhält das Array $A = [0, 0, 1, 1]$ der Länge $n = 4$ als Eingabe. Wie sieht das Array nach dem Durchlauf aus?

Array A: [0, 0, 0, 0]

(ii)

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

Best Case: $\Omega(n)$

Worst Case: $O(n^2)$

c)

(2 Punkte) Beweisen oder widerlegen Sie die folgende Aussage. Geben Sie für einen Beweis passende Konstanten c und n_0 an.

$$2^{2n+3} \in O(5^n)$$

Ziel: Zeigen, dass es positive Konstanten c und n_0 gibt, sodass für alle $n \geq n_0$ gilt:

$$2^{2n+3} \leq c \cdot 5^n$$

Schritte:

1. **Umformen der Ungleichung:**

$$2^{2n} \cdot 2^3 \leq c \cdot 5^n$$

$$8 \cdot (2^2)^n \leq c \cdot 5^n$$

$$8 \cdot 4^n \leq c \cdot 5^n$$

2. **Dividieren durch 5^n :**

$$8 \cdot \frac{4^n}{5^n} \leq c$$

$$8 \cdot \left(\frac{4}{5}\right)^n \leq c$$

3. **Betrachten des Grenzwerts:**

Da $0 < \frac{4}{5} < 1$, gilt:

$$\lim_{n \rightarrow \infty} \left(\frac{4}{5}\right)^n = 0$$

4. **Finden einer Konstante c :**

Der Ausdruck $8 \cdot \left(\frac{4}{5}\right)^n$ hat für $n \geq 0$ einen maximalen Wert von $8 \cdot \left(\frac{4}{5}\right)^0 = 8 \cdot 1 = 8$.

Wählen wir $c = 8$ und $n_0 = 0$. Dann gilt für alle $n \geq 0$:

$$8 \cdot \left(\frac{4}{5}\right)^n \leq 8$$

Somit ist die Bedingung $2^{2n+3} \leq c \cdot 5^n$ erfüllt.

Fazit:

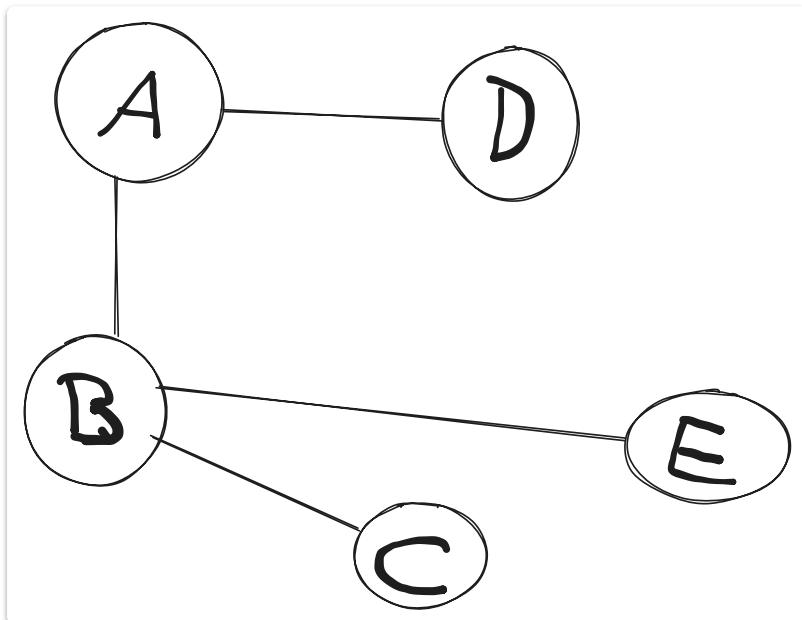
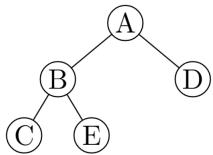
Die Aussage $2^{2n+3} \in O(5^n)$ ist **wahr**.

A2 - Graphen

Stoff: 3. Graphen

a)

(3 Punkte) Zeichnen Sie einen Graphen, auf dem sowohl die Tiefensuche als auch die Breitensuche den unten stehenden Baum erzeugen. Wir nehmen an, dass beide Durchmusterungsverfahren bei A anfangen und bei mehreren Auswahlmöglichkeiten zuerst den lexikographisch kleinsten Knoten besuchen.



b)

(2 Punkte) Ist der geforderte Graph aus der vorherigen Unteraufgabe eindeutig? Begründen Sie Ihre Antwort kurz.

Nein ist er nicht

Es dürfen zusätzliche Knoten und Kanten enthalten sein, **solange sie nicht mit den Knoten A, B, C, D, E verbunden sind**.

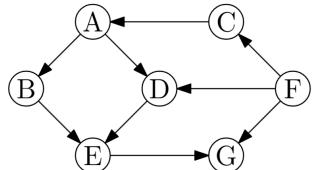
Nur der **Teilgraph**, der die Knoten A–E enthält, muss **genau so aufgebaut sein**, damit

Tiefensuche und Breitensuche denselben Baum erzeugen.

Außerhalb dieses Teilgraphen sind beliebige Ergänzungen möglich, ohne das Ergebnis zu beeinflussen.

c)

(3 Punkte) Geben Sie **alle** topologischen Sortierungen des folgenden Graphen an.



Glaube es sollte nur die beiden hier geben:

- F>C>A>D>B>E>G
- F>C>A>B>D>E>G

d)

(3 Punkte) Füllen Sie die Lücken in folgendem Text aus.

Max-Heaps können benutzt werden, um aus einer Menge von Elementen das größte Element in Zeit $O(\boxed{\quad})$ zu extrahieren. Die Heapeigenschaft fordert für jeden Knoten w des Max-Heaps und jedes Kind u von w , dass $\boxed{\quad}$.

1. $O(1)$
2. $w \geq u$ oder $w.key \geq u.key$

e)

(i)

(2 Punkte) Nennen Sie eine Datenstruktur aus der Vorlesung, um Graphen $G = (V, E)$ zu repräsentieren, mit der Sie für beliebige Knoten $u, v \in V$ in Zeit $O(1)$ entscheiden können, ob $(u, v) \in E$.

Adjazenzmatrix (Man braucht zwar $O(n^2)$ um eine Kante zu überprüfen, aber um zu überprüfen ob ein Knoten darin existiert muss man nur schauen ob der eintrag für u, v existiert und das geht in $O(1)$)

(ii)

(4 Punkte) Angenommen Sie bekommen als Eingabe einen schlichten Graphen G , repräsentiert durch diese Datenstruktur. Schreiben Sie einen Pseudocode, der in Zeit $O(n^3)$ entscheidet, ob G ein Dreieck (einen Kreis der Länge drei) enthält.

```
for Knoten in G:
    for Knoten2 in G:
        if Adjazenzmatrix[Knoten][Knoten2] == 1:
            for Knoten3 in G:
                if Adjazenzmatrix[Knoten2][Knoten3] == 1:
                    if Dajazenzmatrix[Knoten][Knoten3] == 1:
                        return true
return false
```

(iii)

(3 Punkte) Können Sie das Problem aus der vorigen Unteraufgabe in der gleichen asymptotischen Laufzeit $O(n^3)$ lösen, wenn Sie anstelle dieser Datenstruktur, den Graphen G als Adjazenzliste gegeben bekommen? Begründen Sie Ihre Antwort kurz.

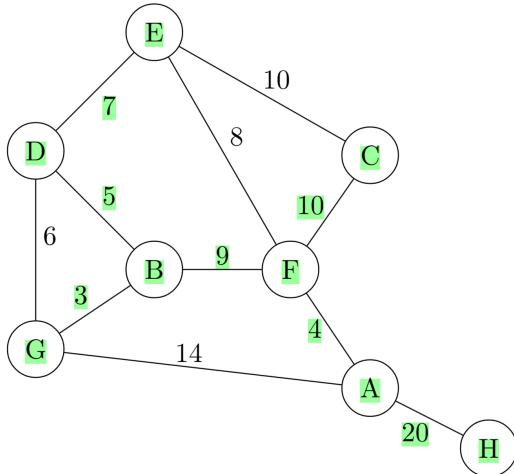
Nein, da wir für jeden Knoten dann genau ein mal durchgehen müssen, was wir auch bei der Matrix machen, aber mit dem feinen Unterschied, dass wir in der letzten Abfrage wo wir `Knoten` und `Knoten3` vergleichen, auch im Worstcase $O(n)$ brauchen. Daher wäre es nicht möglich das in $O(n^3)$ zu machen und es würde auf ein $O(n^4)$ hinauslaufen.

A3 - Greedy

4. Greedy-Algorithmen

a)

(6 Punkte) Berechnen Sie einen minimalen Spannbaum des untenstehenden Graphen. Geben Sie eine mögliche Reihenfolge an, in der der Algorithmus von **Kruskal** die Kanten des Graphen für einen Spannbaum auswählt. Geben Sie die Reihenfolge als Liste von Knotenpaaren an (z.B.: AB, BC, CD, ...).



Reihenfolge: BG, AF, BD, DE, BF, CF, AH

b)

(2 Punkte) Hat der Graph aus Unteraufgabe a) einen eindeutigen minimalen Spannbaum? Begründen Sie Ihre Antwort.

Nein, man könnte beispielsweise auch EC statt FC nehmen

c)

(Q1) Der Greedy-Algorithmus aus der Vorlesung, der das Interval Scheduling Problem in polynomieller Laufzeit löst, sortiert die Intervalle aufsteigend nach Startzeit.

Wahr Falsch

(Q2) Sei K ein Kreis in einem gewichteten Graphen G , sodass dieser Kreis zwei Kanten e_1 und e_2 enthält, die das minimale Gewicht aller Kanten in K haben. Dann existiert ein minimaler Spannbaum von G der e_1 und e_2 enthält.

Wahr Falsch

(Q3) Sei K ein Kreis in einem gewichteten Graphen G , sodass dieser Kreis zwei Kanten e_1 und e_2 enthält, die das minimale Gewicht aller Kanten in G haben. Dann existiert ein minimaler Spannbaum von G der e_1 und e_2 enthält.

Wahr Falsch

(Q4) Sei S eine Teilmenge von Knoten eines zusammenhängenden Graphen und e eine Kante maximalen Gewichts, die genau einen Endknoten in S hat. Dann gibt es keinen minimalen Spannbaum, der e enthält.

Wahr Falsch

- Q1, nein es war die Endzeit sortiert
- Q2, Falsch,

- Nur weil zwei Kanten im Kreis das minimale Gewicht innerhalb dieses *speziellen Kreises* haben, bedeutet das nicht, dass sie im globalen Kontext des Graphen leicht genug sind, um in jedem minimalen Spannbaum enthalten zu sein. Es könnten Kanten außerhalb des Kreises mit noch geringerem Gewicht existieren, die bevorzugt in einem MST verwendet würden.
- Q3, Stimmt
- Q4, kann ja trotzdem darin vorkommen, wenn das die einzige Verbindung zum Rest ist

d)

(4 Punkte) Gegeben ist eine Währung, deren Münzen eine Stückelung von 1,4,10, und 15 haben. Liefert der Greedy-Algorithmus zum Geldwechseln aus der Vorlesung bei dieser Stückelung immer ein optimales Ergebnis? Ein optimales Ergebnis ist ein Ergebnis mit minimaler Anzahl an Münzen.

Ja Nein

Nein, da man wenn man beispielsweise 20€ wechseln will, nur 2 Münzen ($2 \cdot 10$) braucht, während man als Greedy-Lösung allerdings 3 Münzen ($1 \cdot 15 + 1 \cdot 4 + 1 \cdot 1$) bräuchte.

A4 - Sortierverfahren und Divide-and-Conquer

Stoff: [5. Divide and Conquer](#)

a)

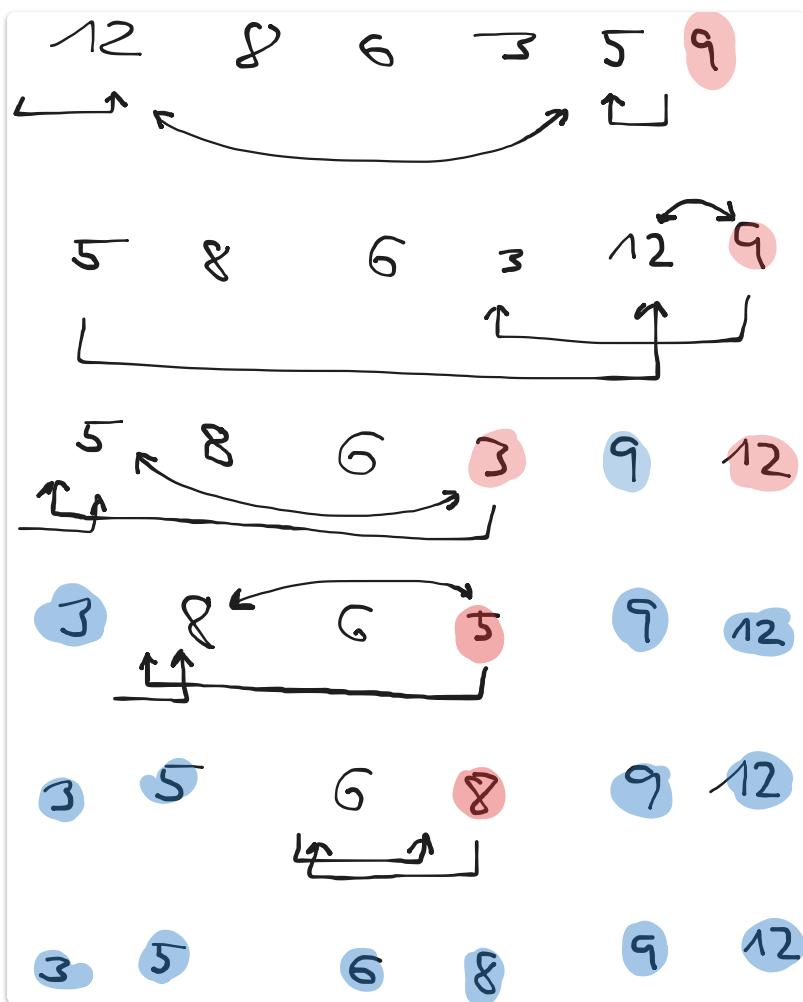
- (Q1) Countsort lässt sich auf Arrays mit Einträgen aus den natürlichen Zahlen \mathbb{N} als Eingabe anwenden und hat dann eine Laufzeit in $O(n \cdot \log n)$.
- Wahr Falsch
- (Q2) Die Best-Case Laufzeit von Mergesort liegt in $O(n \cdot \log n)$.
- Wahr Falsch
- (Q3) Bei einer Ausführung von Mergesort gilt, dass jeder Schlüssel mit $O(\log n)$ anderen verglichen wird.
- Wahr Falsch
- (Q4) Die Average-Case Laufzeit von Quicksort liegt in $O(n \cdot \log n)$.
- Wahr Falsch
- (Q5) Im Best-Case liegt die Höhe des Aufrufbaumes von Mergesort in $\Omega(\log n)$.
- Wahr Falsch

- Q1, Falsch, hat eine Laufzeit von $O(n)$
- Q2, Stimmt

- Q3, Stimmt
- Q4, Stimmt
- Q5, Stimmt

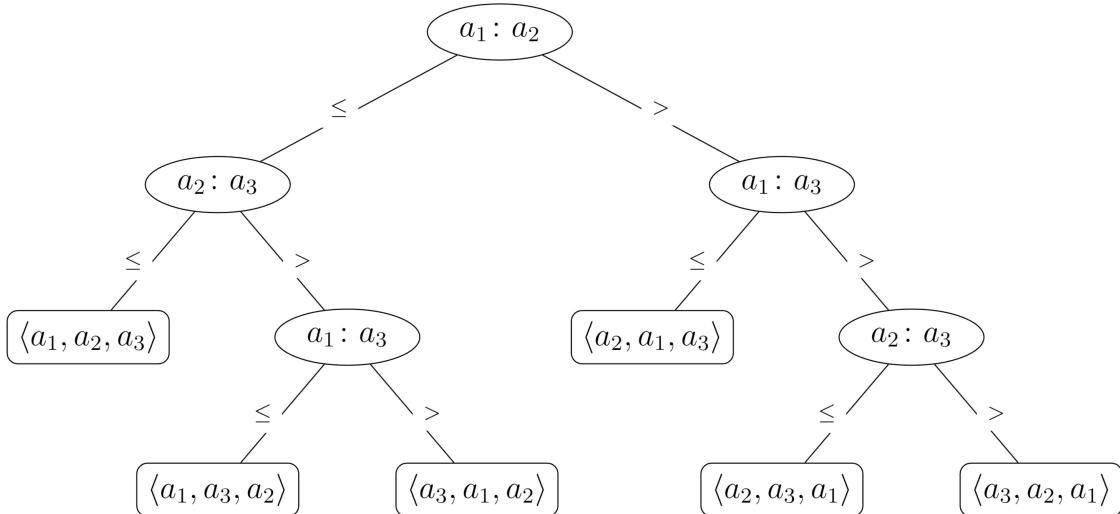
b)

(4 Punkte) Führen Sie **Quicksort** auf dem unten gegebenen Array aus, um es aufsteigend zu sortieren. Wählen Sie stets das **letzte** Element als Pivotelement aus und implementieren Sie den Schritt des Aufteilens so, dass die Elemente einer Subfolge immer in derselben Reihenfolge angeordnet werden wie in der Originalfolge. Geben Sie die Zwischenschritte an, wie in der Vorlesung und Übung kennen gelernt. Markieren Sie in jedem Zwischenschritt das ausgewählte Pivotelement indem Sie es einkreisen. Gegeben ist das Array [12, 8, 6, 3, 5, 9].

**c)**

(6 Punkte) In der Vorlesung haben Sie einige vergleichsbasierte Sortieralgorithmen kennengelernt, wie zum Beispiel Quicksort und Mergesort. Ein vergleichsbasierter Algorithmus geht nach dem folgenden Prinzip vor: Er bekommt als Eingabe eine Liste A von paarweise verschiedenen Zahlen und führt eine Serie von Vergleichen von Paaren dieser Zahlen aus. Basierend auf den Ergebnissen dieser Vergleiche, gibt er die Zahlen in A in aufsteigender Reihenfolge aus.

Untenstehend ein Beispiel dafür, wie ein vergleichsbasierter Algorithmus bei Eingabe der Liste $A = \langle a_1, a_2, a_3 \rangle$ vorgehen könnte. In diesem Beispiel vergleicht er zuerst a_1 mit a_2 . Wenn $a_1 \leq a_2$, dann vergleicht er a_2 mit a_3 , wenn $a_1 > a_2$, dann vergleicht er a_1 mit a_3 und so weiter.



Zeigen Sie, dass jeder vergleichsbasierte Algorithmus im Worst-Case eine Laufzeit von $\Omega(n \cdot \log n)$ hat, wobei n die Anzahl der Elemente in der Eingabeliste A ist. Sie dürfen ohne Beweis annehmen, dass $\log_2(n!) \in \Omega(n \cdot \log n)$.

Um zu zeigen, dass jeder vergleichsbasierte Sortieralgorithmus im Worst-Case eine Laufzeit von $\Omega(n \cdot \log n)$ hat, argumentieren wir wie folgt:

1. **Anzahl der möglichen Ausgaben:** Für n verschiedene Eingabeelemente gibt es $n!$ mögliche Permutationen, die der Algorithmus korrekt sortieren muss.
2. **Entscheidungsbaum:** Ein vergleichsbasierter Sortieralgorithmus kann durch einen Entscheidungsbaum modelliert werden. Jeder innere Knoten repräsentiert einen Vergleich, und jedes Blatt repräsentiert eine der $n!$ möglichen sortierten Ausgaben.
3. **Höhe des Entscheidungsbaumes:** Ein binärer Baum mit mindestens $n!$ Blättern hat eine Höhe von mindestens $\log_2(n!)$. Die Worst-Case-Laufzeit des Algorithmus entspricht der Höhe dieses Entscheidungsbaumes.
4. **Untere Schranke für $\log_2(n!)$:** Es ist gegeben, dass $\log_2(n!) \in \Omega(n \cdot \log n)$.
5. **Schlussfolgerung:** Da die Worst-Case-Laufzeit durch die Höhe des Entscheidungsbaumes beschränkt ist und die Höhe mindestens $\log_2(n!) \in \Omega(n \cdot \log n)$ beträgt, hat jeder vergleichsbasierte Sortieralgorithmus im Worst-Case eine Laufzeit von $\Omega(n \cdot \log n)$.

A5 - Suchbäume und Hashing

Stoff: [6. Suchbäume](#) und [7. Hashing](#)

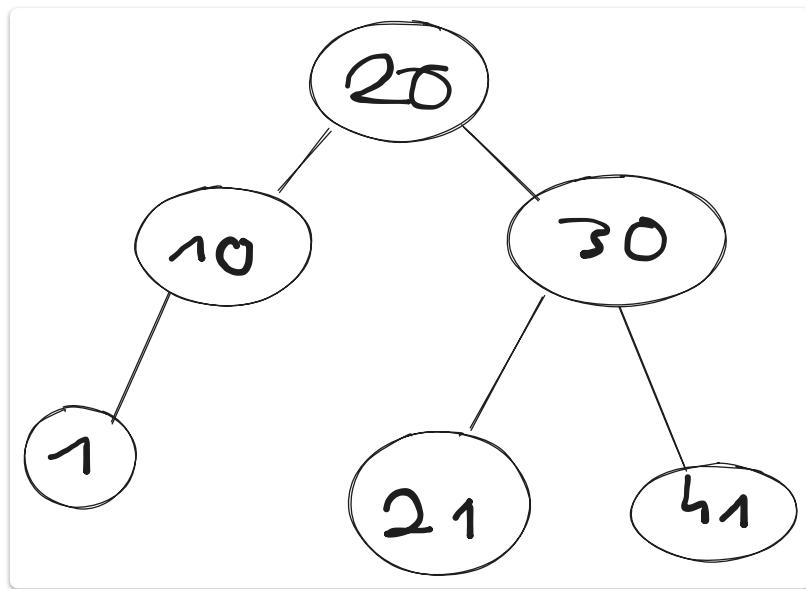
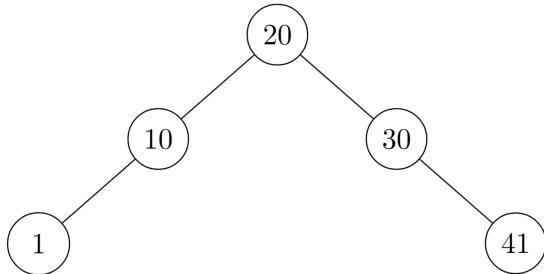
a)

(9 Punkte) Gegeben ist ein Algorithmus Foo, welcher die Wurzel *root* eines natürlichen binären Suchbaumes als Eingabe erhält und die aus der Vorlesung bekannten Funktionen **Minimum**, **Maximum**, **Search** und **Insert** für Suchbäume verwendet.

```
Foo(root):
    a ← Minimum(root)
    b ← Maximum(root)
    c ← ⌊(a + b)/2⌋
    if Search(root, c) = null then
        Insert(root, c)
```

(i)

Führen Sie Foo mit folgendem Suchbaum als Eingabe aus und geben Sie den Baum nach Ausführung der Funktion an. Sie dürfen dazu die untenstehende Abbildung ergänzen.



(ii)

Bestimmen Sie die Worst- und Best-Case Laufzeit von Foo in O -Notation abhängig von der Knotenanzahl n des gegebenen Suchbaumes. Wählen Sie eine kleinstmögliche obere Schranke.

Bestcase:

`Minimum(root)` in $O(\log n)$

`Maximum(root)` in $O(\log n)$

`Insert(root, c)` schon vorhanden also in $O(1)$

$$\implies O(\log n)$$

Worstcase:

`Minimum(root)` in $O(n)$

`Maximum(root)` in $O(n)$

`Insert(root, c)` in $O(n)$

$$\implies O(n)$$

(iii)

Nehmen Sie jetzt an, die Eingabe für Foo ist auf AVL-Bäume beschränkt und die Funktion `Insert`, welche in Foo verwendet wird, ist die Einfügeoperation für AVL-Bäume. Bestimmen Sie die Worst-Case und Best-Case Laufzeit von Foo in O -Notation unter dieser Einschränkung. Wählen Sie eine kleinstmögliche obere Schranke.

Bestcase bleibt gleich also $O(\log n)$

Worstcase:

`Minimum(root)` in $O(\log n)$

`Maximum(root)` in $O(\log n)$

`Insert(root, c)` in $O(\log n)$

$$\implies O(\log n)$$

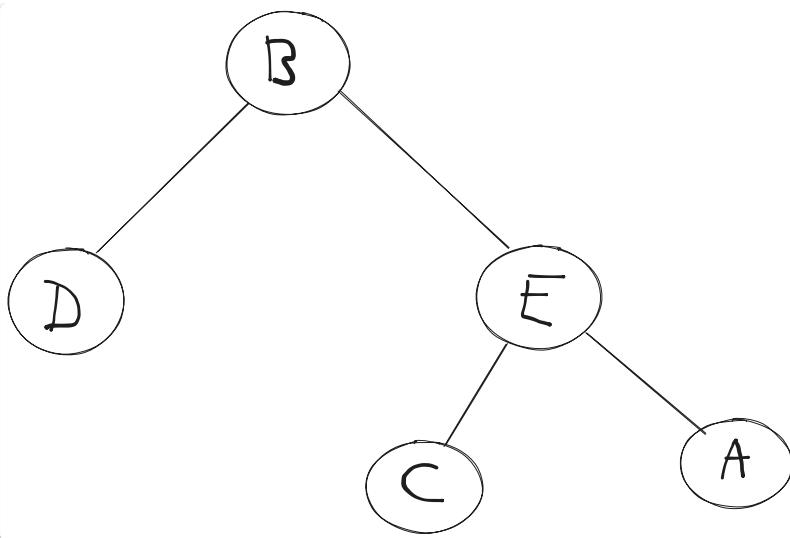
b)

(4 Punkte) Bestimmen Sie den binären Wurzelbaum, der durch die folgende Inorder- und Postorder-Durchmusterungsreihenfolge gegeben ist.

Inorder: d, b, c, e, a

Postorder: d, c, a, e, b

- **Inorder:** Linker Teilbaum -> Wurzel -> Rechter Teilbaum
 - D -> B -> C -> E -> A
- **Postorder:** Linker Teilbaum -> Rechter Teilbaum -> Wurzel
 - D -> C -> A -> E -> B



c)

(7 Punkte) Gegeben ist folgende Hashtabelle T der Länge $m = 7$.

Position	0	1	2	3	4	5	6
Schlüssel			9		2	12	
Flag	frei	frei	besetzt	wieder frei	besetzt	besetzt	frei

Zur Kollisionsbehandlung wird **lineares Sondieren** mit $h(k, i) = (h'(k) + i) \bmod 7$, wobei $h'(k) = k \bmod 7$, verwendet. Für das Sondieren gilt $i = 0, 1, \dots, m - 1$.

(i)

Führen Sie eine Suche nach Schlüssel 2 in T aus. Geben Sie alle dabei sondierte Positionen in entsprechender Sondierungsreihenfolge an. Ist die Suche erfolgreich?

$2 \rightarrow h(2) = 2 \bmod 7 = 2$, 2 ist besetzt also Sondieren:

- +1, wir sind bei 3, 3 ist wieder frei
- +1, wir sein bei 4, 4 beinhaltet 2 \Rightarrow gefunden, **Suche erfolgreich**.

(ii)

Führen Sie jetzt eine Suche nach Schlüssel 16 in T aus. Geben Sie erneut alle dabei sondierte Positionen in entsprechender Sondierungsreihenfolge an. Ist die Suche erfolgreich?

$16 \rightarrow h(16) = 16 \bmod 7 = 2$, 2 ist besetzt also Sondieren:

- +1 \rightarrow 3 (wieder frei)
- +1 \rightarrow 4 (besetzt und nicht richtig)
- +1 \rightarrow 5 (besetzt und nicht richtig)
- +1 \rightarrow 6 (frei) \Rightarrow 16 ist nicht in der Hashmap, **Suche erfolglos**

(iii)

Fügen Sie den Schlüssel 23 in T ein. An welcher Position wird der Schlüssel eingefügt? Welcher Flag wird nach dem Einfügen an dieser Position gesetzt?

23 --> $h(23) = 23 \bmod 7 = 2$, 2 ist besetzt also Sondieren:

- +1, besetzt
- +1, wieder frei --> Wir setzen unsere 23 in **Feld 3** und ändern den Flag auf **besetzt**

(iv)

Angenommen wir löschen den Schlüssel 9 aus T . Welcher Flag wird nach dem Löschen an der betroffenen Position gesetzt?

- **Wieder frei**