

2022_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)

(4 Punkte) Ordnen Sie folgende Funktionen nach Dominanz (\ll), beginnend mit der asymptotisch am schwächsten wachsenden. Es genügt die Funktionen zu reihen, ein Beweis der Gültigkeit der Relationen ist nicht erforderlich.

$$\frac{4n^5 + 5n^4}{7n^3}, \quad 3^{-n}, \quad \log(n), \quad 2 \cdot \sqrt{5n^6}, \quad (3n)!, \quad 1.1^n, \quad \frac{n}{50}$$

$$3^{-n} \ll \log(n) \ll \frac{n}{50} \ll \frac{4n^5 + 5n^4}{7n^3} \ll 2 \cdot \sqrt{5n^6} \ll 1.1^n \ll (3n)!$$

b)

(4 Punkte) Bestimmen Sie die Laufzeit des angegebenen Algorithmus in Abhängigkeit des Eingabeparameters $n \in \mathbb{N}$ in Θ -Notation. Verwenden Sie hierfür möglichst einfache Terme.

```

 $k \leftarrow 0$ 
for  $i = 1, \dots, n$ 
    for  $j = 1, \dots, \lceil \log(i) \rceil$ 
         $k \leftarrow k + 1$ 
    for  $j = i + n, \dots, 3n$ 
         $k \leftarrow k - 1$ 
return  $k$ 

```

Laufzeit (in θ -Notation): $O(n^2)$

c)

(4 Punkte) Bestimmen Sie die Best-Case und Worst-Case Laufzeit des angegebenen Algorithmus in Abhängigkeit des Eingabeparameters $n \in \mathbb{N}$ in Θ -Notation. Das Integer-Array A der Länge n ist Teil der Eingabe. Verwenden Sie hierfür möglichst einfache Terme.

```

if  $n < 1000$  then
     $a \leftarrow 1$ 
    for  $j = 1, \dots, 2^n$ 
         $a \leftarrow 2a$ 
    return  $a$ 
else
    for  $i = 1, \dots, n$ 
        if  $A[i] \bmod 2 = 0$  then
            return  $i$ 
    return 0

```

Bestcase: $\Omega(1)$

Worstcase: $O(2^n)$ oder $O(n)$ für große n

d)

(6 Punkte) Bestimmen Sie die Laufzeit und den Rückgabewert des angegebenen Algorithmus in Abhängigkeit vom Eingabeparameter $n \in \mathbb{N}$ in Θ -Notation. Verwenden Sie hierfür möglichst einfache Terme.

```

 $a \leftarrow n$ 
 $b \leftarrow 0$ 
for  $i = 1, \dots, n$ 
    for  $j = 1, \dots, n$ 
         $a \leftarrow a/2$ 
        if  $a < 1$  then
            return  $b$ 
         $b \leftarrow b + 1$ 
    return 0

```

Laufzeit: $\theta(\log n)$

Rückgabewert: $\theta(\log n)$

wir kommen immer in der ersten iteration von der äußeren schleife zum return statement. egal wie groß n ist, wenn man n, n mal halbiert, hat man im Endeffekt eine zahl die kleiner als 1 ist. daher kommt es nie zu einer 2. iteration der äußeren schleife und dadurch auch nie zu einem $n \cdot \log n$ sondern nur zu einem $\log n$

e)

e) (2 Punkte) Sei $f : \mathbb{N}^+ \rightarrow \mathbb{N}^+$ eine Funktion. Nehmen Sie an, es gilt:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{n^2} = 0$$

Was folgt daraus? Kreuzen Sie alle zutreffenden Aussagen an:

- (MC1) $f(n)$ ist in $O(n^2)$
- (MC2) $f(n)$ ist in $\Theta(n^2)$
- (MC3) $f(n) \ll n^2$
- (MC4) keine der zuvor genannten Aussagen

A2 - Graphen

Stoff: [3. Graphen](#)

a)

- (Q1) Sei G ein gerichteter Graph für den genau eine topologische Sortierung existiert, dann ist G schwach zusammenhängend.
 Wahr Falsch
- (Q2) Sei G ein gerichteter Graph für den genau eine topologische Sortierung existiert, dann ist G stark zusammenhängend.
 Wahr Falsch
- (Q3) Jeder gerichtete Graph mit n Knoten, der einen gerichteten Kreis der Länge n enthält, ist stark zusammenhängend.
 Wahr Falsch
- (Q4) Jeder azyklische gerichtete Graph mit n Knoten besitzt genau n starke Zusammenhangskomponenten.
 Wahr Falsch
- (Q5) Jeder gerichtete Graph mit k schwachen Zusammenhangskomponenten besitzt höchstens k starke Zusammenhangskomponenten.
 Wahr Falsch

Q1: Ja weil sonst gibts generell keine topologische Sortierung

Q2: Falsch, weil dann gibts keine topologische Sortierung

Q3: Ja weil dann alle im Kreis sind.

Q4: Ja weil jeder Knoten für sich selbst eine ist, und da es keine Kreise gibt gibts keine weiteren.

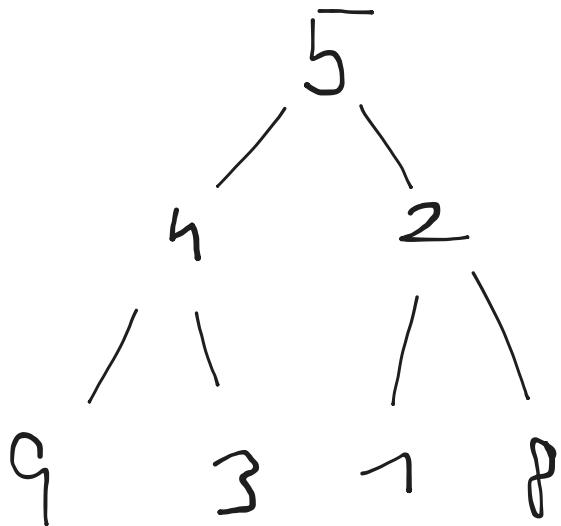
Q5: Falsch, da auch jeder Knoten für sich selbst eine starke Zusammenhangskomponente ist.

b)

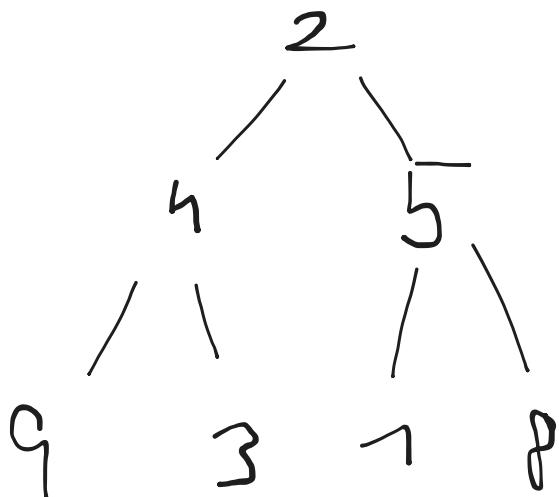
(6 Punkte) Gegeben ist folgendes Array, das zur Repräsentation eines Heaps verwendet wird (Kodierung wie aus der Vorlesung bekannt).

	5	4	2	9	3	1	8
---	---	---	---	---	---	---	---

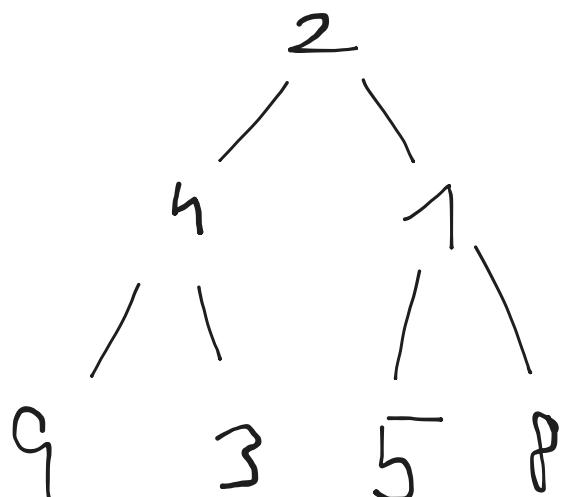
Führen Sie den Algorithmus zur Erstellung eines Min-Heaps aus der Vorlesung aus. Geben Sie den initialen Baum und den Baum nach jeder Iteration an (in Baumdarstellung, nicht als Array).

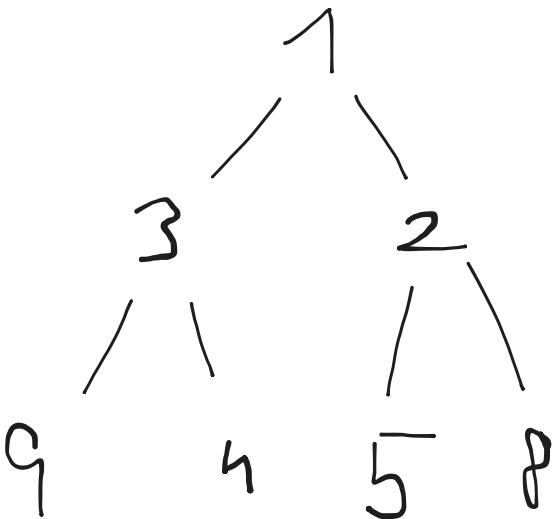
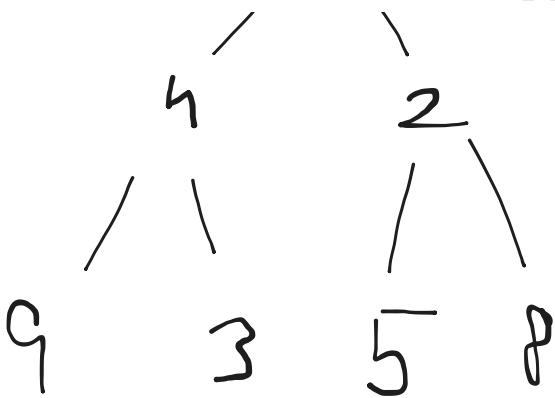


Heap up: 1. Iteration, tausche Wurzel
wenn größer als Kind mit kleinerem Kind



Und so weiter:





c)

(9 Punkte) Ein ungerichteter Graph $G = (V, E)$ wird **bipartit** genannt genau dann wenn die Knoten in zwei disjunkte Mengen A, B partitioniert werden können, sodass für jede Kante $(u, v) \in E$ gilt: $u \in A, v \in B$ oder $v \in A, u \in B$. Es existiert also keine Kante, die beide Endknoten in der gleichen Menge hat.

Geben Sie an, wie die Breitensuche verwendet werden kann, um für einen gegebenen Graphen mit n Knoten und m Kanten in Zeit $O(n + m)$ zu testen, ob er bipartit ist oder nicht.

Eine präzise Beschreibung reicht als Antwort. Ein detaillierter Pseudocode oder Korrektheitsbeweis ist nicht erforderlich.

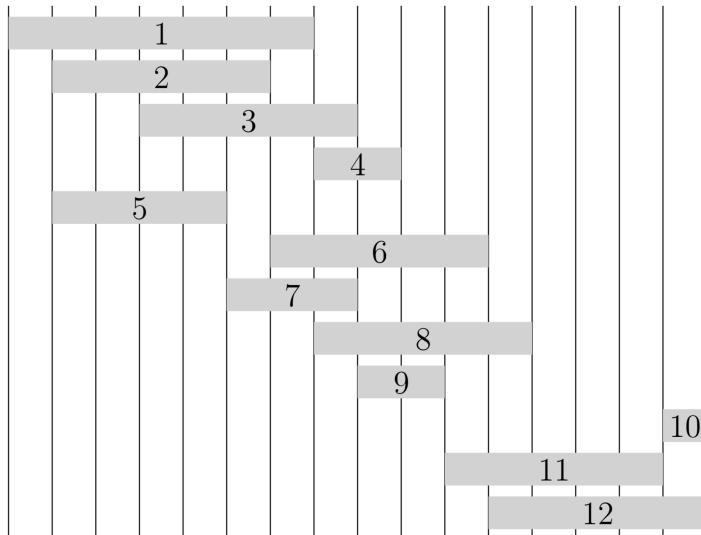
In jeder Iteration der Breitensuche, weisen wir alle Adjazenten Knoten zum Anfangsknoten der entgegengesetzten Menge, als der Startknoten zu. Wenn man dann nach einer weiteren Iteration irgendwie zurückkommt und einen Knoten findet, der schon in einer Menge ist, kann es entweder die Menge sein, die man jetzt dem Knoten auch zuweisen würde, dann wäre dieser Graph immer noch potenziell bipartit, oder auch die Menge in der man gerade ist also die entgegengesetzte von der die man jetzt dem Knoten zuweisen würde. Ist zweiteres der Fall, so ist die Biparität ausgeschlossen.

A3 - Greedy

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

a)

(4 Punkte) Geben Sie eine optimale Lösung für folgende Interval Scheduling Instanz an:



Wir sortieren nach Beendigungszeit:

- 5
- 2
- 1
- 3
- 7
- 4
- 9
- 6
- 8
- 11
- 10
- 12

Dann nehmen wir den ersten von den Sortierten in unser Ergebnis auf:

- 5

Dann vergleicht man den nächsten aus der Liste ob die Endzeit von **5** kleiner ist als die Startzeit des nächstens:

- 5 endet nachdem 2 begonnen hat --> 2 wird gestrichen
- 5 endet nachdem 1 begonnen hat --> 1 wird gestrichen
- 5 endet nachdem 3 begonnen hat --> 3 wird gestrichen
- 5 endet und danach beginnt 7 --> 7 wird dem Ergebnis hinzugefügt.

Das machen wir jetzt bis zum Ende.

Dann haben wir als Ergebnis:

[5, 7, 9, 11, 10]

b)

(3 Punkte) Vervollständigen Sie durch Ankreuzen den folgenden Text, sodass er einen Algorithmus beschreibt, der das Interval Scheduling Problem in Polynomialzeit löst:

Sortiere die Intervalle aufsteigend nach

Länge Startpunkt Anzahl ihrer Konflikte.

Solange noch Intervalle da sind, füge

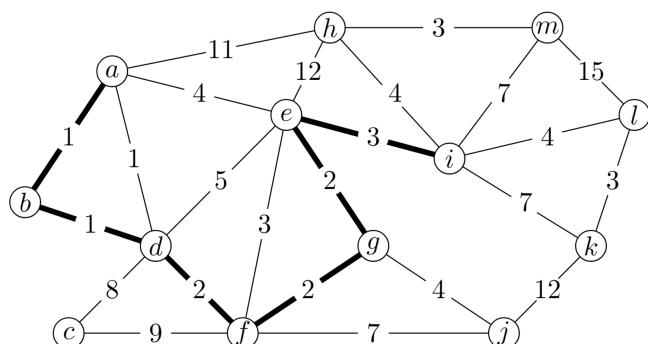
das erste Intervall das letzte Intervall ein beliebiges Intervall

zur Lösung hinzu und lösche alle Intervalle die mit diesem Intervall in Konflikt stehen.

`Startpunkt` und `das letzte Intervall` wenn man argumentiert, dass man dann den Algorithmus rückwärts verwendet.

c)

(6 Punkte) Betrachten Sie folgenden durch fette Kanten markierten partiellen Spannbaum, wie der Algorithmus von Kruskal oder Prim ihn im Laufe seiner Ausführung bisher berechnet haben könnte. Geben Sie für jeden der beiden Algorithmen an, welche Kante als nächstes zum partiellen Spannbaum hinzugefügt wird. Wenn es mehrere Möglichkeiten gibt, geben Sie alle an.

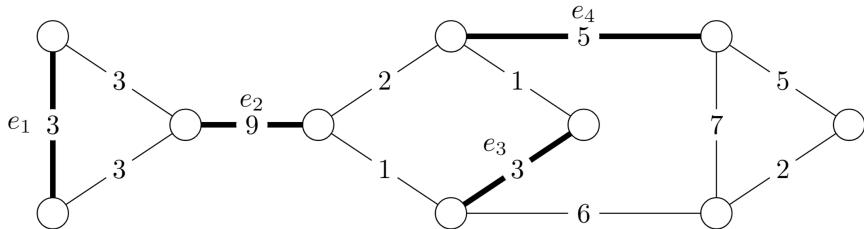


Kruskal: hm oder 1k

Prim: ih , il oder gj

d)

(4 Punkte) Geben Sie für die hervorgehobenen vier Kanten an, ob sie in keinem minimalen Spannbaum, in mindestens einem aber nicht allen minimalen Spannbäumen oder in allen minimalen Spannbäumen enthalten sind.



- | | | | |
|--------------|---|--|--|
| (Q1) e_1 : | <input type="checkbox"/> in keinem | <input checked="" type="checkbox"/> in mind. einem, aber nicht allen | <input type="checkbox"/> in allen |
| (Q2) e_2 : | <input type="checkbox"/> in keinem | <input type="checkbox"/> in mind. einem, aber nicht allen | <input checked="" type="checkbox"/> in allen |
| (Q3) e_3 : | <input checked="" type="checkbox"/> in keinem | <input type="checkbox"/> in mind. einem, aber nicht allen | <input type="checkbox"/> in allen |
| (Q4) e_4 : | <input type="checkbox"/> in keinem | <input type="checkbox"/> in mind. einem, aber nicht allen | <input checked="" type="checkbox"/> in allen |

e) Begründung für e_4 :

Da wir einen Kreis gegeben haben, welcher Kante e_4 mit Gewicht 5 und eine andere Kante mit Gewicht 6 haben, kann man davon ausgehen, dass die Kante mit niedrigerem Gewicht, da es sonst keine Verbindung in den Kreis geben würde genommen werden muss.

Da das in dem Fall die Kante e_4 ist, wird diese in jeder Möglichkeit eines minimalen Spannbaums genommen.

Formal argumentiert mit Kantenschnittlemma:

Wenn man sich einen Schnitt vorstellt, welcher von Vertikal durch den Graphen geht und das Dreieck auf der rechten Seite vom Rest trennt, do ist e_4 die kleinste Kante die durch den Schnitt geht und daher immer bei jedem minimalen Spannbaum.

A4 - Sortierverfahren und Divide-and-Conquer

Stoff: 5. Divide and Conquer

a)

- (Q1) Der Stack von Runs mit den Längen (Länge des obersten Elements zuerst) $(7, 11, 19, 26)$ erfüllt die Invarianten von TimSort.
- Wahr Falsch
- (Q2) SelectionSort wie er in der Vorlesung besprochen wurde ist ein stabiles Sortierverfahren.
- Wahr Falsch
- (Q3) Die Anzahl der Vergleiche, die ein allgemeines (d.h. vergleichsbasiertes) Sortierverfahren im **Best-Case** durchführt, liegt immer in $\Omega(n/\log n)$.
- Wahr Falsch
- (Q4) Zum Verschmelzen (Mergen) zweier sortierter Arrays A und B mit jeweils n Elementen und $\forall i, j \in \{1, \dots, n\} : A[i] < B[j]$ werden in TimSort $\Theta(\log n)$ Vergleiche durchgeführt.
- Wahr Falsch

Q2: Falsch, siehe [AD_08_PraktischeDatenstrukturen](#), p.35

Q3: Falsch, weil $\Omega(n)$

Q4: Falsch, weil $\Theta(n)$

b)

(12 Punkte) Gegeben sind zwei natürliche Zahlen x und y mit jeweils $n = 2^k$ Bit. Man betrachte nun die Bitstrings der binären Repräsentation von x und y und teile sie in die Hälfte der höherwertigen und die Hälfte der niederwertigen Bits. Ist beispielsweise $x = 01101010_2$, dann ergibt sich die höherwertige Hälfte zu 0110_2 und die niederwertige Hälfte zu 1010_2 .

Sei $\text{Split}(x)$ die Funktion, die diese Aufteilung vornimmt. D.h. $x_1, x_0 \leftarrow \text{Split}(01101010_2)$ resultiert in $x_1 = 0110_2$ und $x_0 = 1010_2$. Mit $x_1, x_0 \leftarrow \text{Split}(x)$ und $y_1, y_0 \leftarrow \text{Split}(y)$ ergibt sich das Produkt $x \cdot y$ dann zu

$$z = x \cdot y = \underbrace{x_1 \cdot y_1}_{z_2} \cdot 2^n + \underbrace{(x_1 \cdot y_0 + x_0 \cdot y_1)}_{z_1} \cdot 2^{n/2} + \underbrace{x_0 \cdot y_0}_{z_0}.$$

Beachten Sie, dass folgende Gleichung gilt:

$$z_1 = z_2 + z_0 - (x_1 - x_0) \cdot (y_1 - y_0)$$

Der Divide-and-Conquer-Algorithmus auf der nächsten Seite soll die beiden Übergabeparameter x und y miteinander multiplizieren. Vervollständigen Sie die fehlenden Teile. Verwenden Sie dazu nur Additionen, Subtraktionen, die Betragsfunktion $|\cdot|$ und rekursive Aufrufe von `Multiply`. Achten Sie darauf, dass sich die Methode `Multiply` nur **drei** mal selbst aufruft.

```

Multiply( $x, y, k$ ):
//  $x$  und  $y$  sind ganze, nichtnegative Zahlen mit jeweils  $n = 2^k$  Bit
if  $k = 0$  then
    return  $x \cdot y$ 
 $x_1, x_0 \leftarrow \text{Split}(x)$ 
 $y_1, y_0 \leftarrow \text{Split}(y)$ 
 $A \leftarrow$  
 $C \leftarrow$  
if  $(x_1 > x_0 \wedge y_1 > y_0) \vee (x_1 < x_0 \wedge y_1 < y_0)$  then
     $B \leftarrow$  
else
     $B \leftarrow$  
return  $A \cdot 2^{2^k} + B \cdot 2^{2^{k-1}} + C$ 

```

A <- Multiply(x_1, y_1, k-1)

Das berechnet $z_2 = x_1 \cdot y_1$

C <- Multiply(x_0, y_0, k-1)

Das berechnet $z_0 = x_0 \cdot y_0$

B <- Multiply(|x_1 - x_0|, |y_1 - y_0|, k-1)

B <- Multiply(x_1 - x_0, y_1 - y_0, k-1)

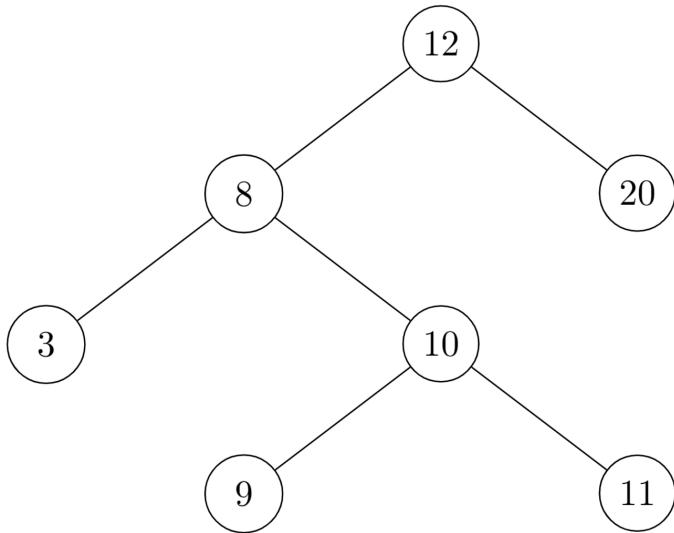
Die letzten beiden berechnen $(x_1 - x_0) \cdot (y_1 - y_0)$

A5 - Suchbäume und Hashing

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

a)

(12 Punkte) Gegeben ist folgender binärer Suchbaum T :



(i)

Geben Sie die *Inorder*- und *Postorder*-Durchmusterungsreihenfolge von T an.

Inorder: 3, 8, 9, 10, 11, 12, 20

Postorder: 3, 9, 11, 10, 8, 20, 12

(ii)

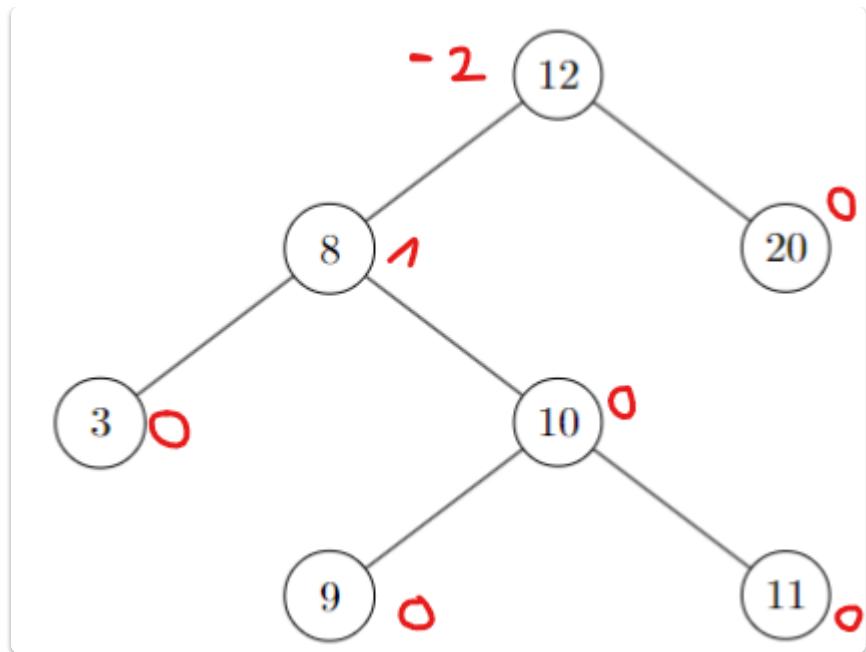
Wie viele Schlüsselvergleiche benötigt eine erfolgreiche Suche in T durchschnittlich?

Hinweis: Bei der Suche nach dem Wurzelknoten wird genau ein Schlüsselvergleich benötigt.

$\frac{19}{7} = 2.71$ Vergleiche im Durchschnitt

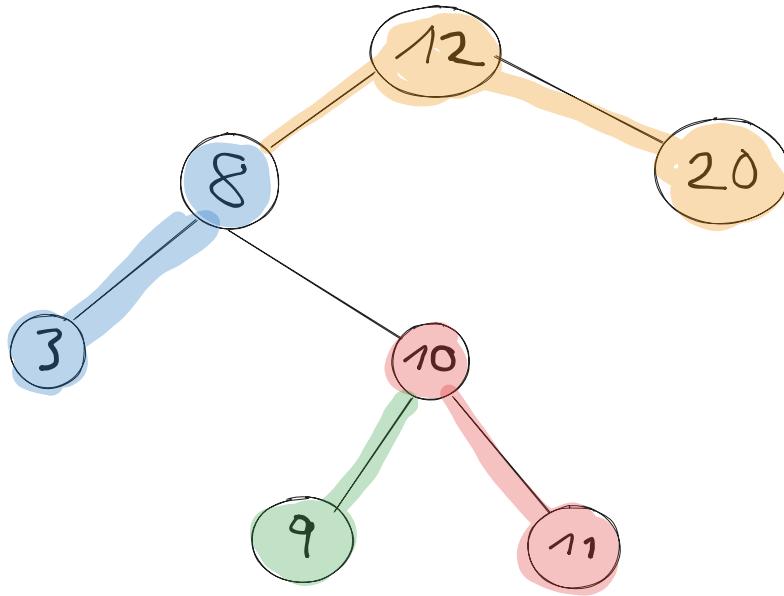
(iii)

Bestimmen Sie die Balance jedes Knoten in T , wie aus der Vorlesung zu AVL-Bäumen bekannt.

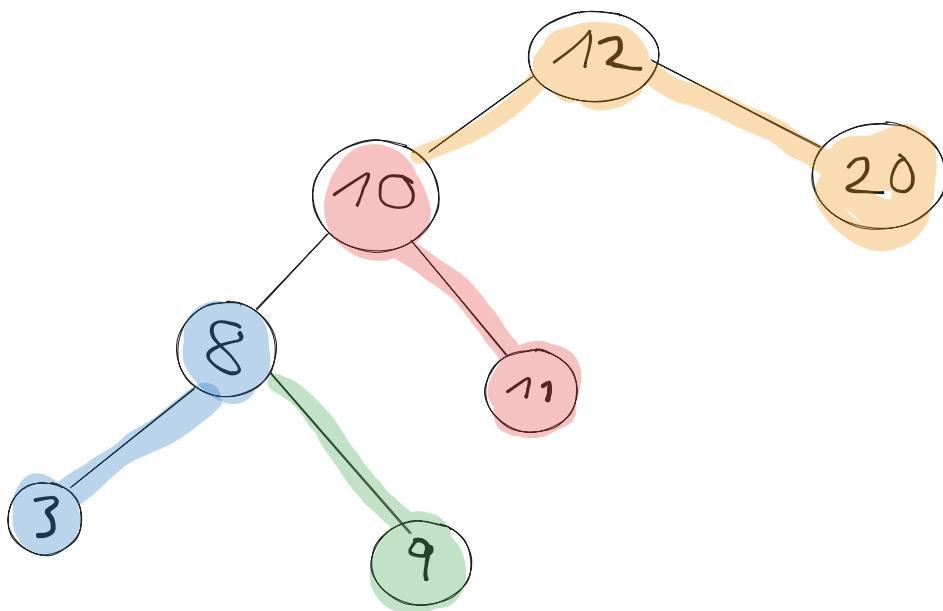


(iv)

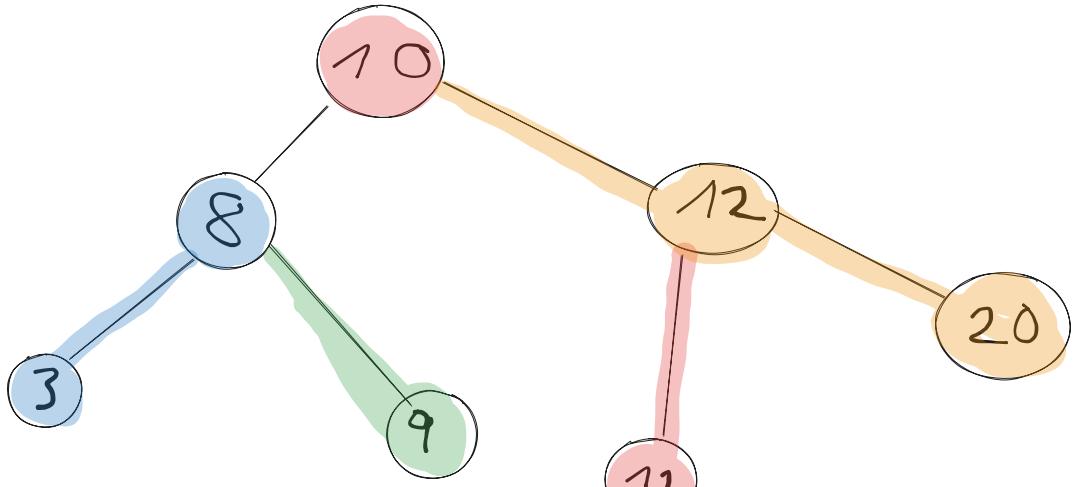
Rebalancieren Sie T indem Sie die benötigte(n) Rotation(en) durchführen, wie aus der Vorlesung zu AVL-Bäumen bekannt. Geben Sie den resultierenden Baum nach jedem Zwischenschritt an, also nach jeder Rotation.



1. Linksrotation



2. Rechtsrotation an der Wurzel



b)

(4 Punkte) Gegeben ist folgende Hashtabelle sowie die Hashfunktion $h'(k) = k \bmod 7$.

Position	0	1	2	3	4	5	6
Schlüssel		8		17	4		20

Fügen Sie den Schlüssel 10 als neues Element mittels der jeweiligen Sondierungsvariante in die Hashtabelle ein. Geben Sie dabei alle Zwischenschritte an, also alle Kollisionen zu denen es beim Einfügen kommt.

(i)

Lineares Sondieren mit $h(k, i) = (h'(k) + i) \bmod 7$.

$$h(10) = 10 \bmod 7 = 3$$

- 3 besetzt:
 - $+1 \rightarrow 4$ besetzt
 - $+1 \rightarrow 5$ frei \rightarrow 5 gewählt

(ii)

Quadratisches Sondieren mit $h(k, i) = (h'(k) + \frac{1}{2}i + \frac{1}{2}i^2) \bmod 7$.

$$h(10) = 10 \bmod 7 = 3$$

- 3 besetzt:
 - $+0.5 + 0.5 \rightarrow 4$ besetzt
 - $+0.5 \cdot 2 + 0.5 \cdot 4 = +3 \rightarrow 6$ besetzt
 - $+0.5 \cdot 3 + 0.5 \cdot 9 = +6 \rightarrow$ 2 gewählt

c)

(Q1) Beim Einfügen eines Schlüssels k in eine Hashtabelle mittels *Verbesserung nach Brent* kann es vorkommen, dass ein sich bereits in der Tabelle befindliches Element k' verschoben wird, um für k Platz zu machen.

Wahr Falsch

(Q2) Beim Uniform Hashing in einer Tabelle mit Größe m erhält jeder Schlüssel mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $0, 1, \dots, m - 1$ als Sondierungsreihenfolge zugeordnet.

Wahr Falsch

(Q3) Beim Hashing mit Verkettung der Überläufer gilt für den Belegungsfaktor $\alpha = \frac{n}{m}$ immer $\alpha \leq 1$, wobei m die Größe der Hashtabelle ist und n die Anzahl der in der Tabelle gespeicherten Schlüssel.

Wahr Falsch

(Q4) Bei offenen Hashverfahren wird das Flag „*wieder frei*“ gesetzt, um sicherzustellen, dass weiterhin alle Elemente in der Hashtabelle gefunden werden können.

Wahr Falsch

Q3: Falsch, bei Verkettung bei Überlauf, werden Überlaufende Elemente in Listenform dran gehängt.