Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn

Klausur Informatik 2

Montag den 3. September 2018, 13:00-16:00

Name: .....

Matrikel Nr.: .....

Unterschrift: .....

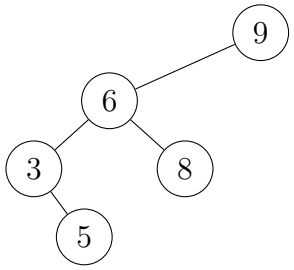
Erst öffnen wenn die Klausuraufsicht die Erlaubnis erteilt!

- Legen Sie Ihren **Studentenausweis** sichtbar vor sich.
- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- Es wird **nur eine Lösung pro Aufgabe** gewertet. Vergewissern Sie sich, dass Sie zusätzliche Lösungen durchstreichen, andernfalls wird die schlechteste Lösung gewertet.
- Detaillierte Schritte können Ihnen zu **Teilpunkten** verhelfen falls das Endergebnis falsch ist.
- Die Schlüsselwörter **Zeigen Sie...**, **Beweisen Sie...**, **Begründen Sie...** oder **Leiten Sie ... her** zeigen an, dass Sie Ihre Antwort sorgfältig und gegebenenfalls formal begründen müssen.
- Die Schlüsselwörter **Geben Sie ... an** zeigen an, dass Sie lediglich die geforderte Antwort und keine Begründung liefern müssen.
- Die folgenden Regeln gelten **überall**, außer sie werden explizit außer Kraft gesetzt.
- Bei Laufzeitfragen ist nur die **asymptotische Laufzeit** notwendig.
- Wenn Sie einen Algorithmus angeben sollen, so können Sie Pseudocode angeben. Eine **ausreichend detaillierte** Beschreibung der Funktionsweise Ihres Algorithmus genügt jedoch.
- Algorithmen aus der Vorlesung **können als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!

Aufgabe	1	2	3	4	5	6	7	8	**Total**
Maximum	13	9	9	11	12	12	13	11	90
Punkte									

Aufgabe 1: Kurze Fragen *(13 Punkte)*

(a) Gegeben sei der folgende binäre Suchbaum mit Wurzel 9:



Geben Sie **alle Folgen** von `insert`(key) Operationen an, welche diesen Baum erzeugen. *(2 Punkte)*

(b) Geben Sie jeweils die Besuchsreihenfolge der Knoten für die **In-Order, Pre-Order, Post-Order und Level-Order** Traversierung für den obigen Baum an. *(2 Punkte)*

(c) Nennen Sie **einen Vorteil** von **einfach** verketteten Listen gegenüber **doppelt** verketteten Listen. Geben Sie für beide Datenstrukturen die Laufzeit der **Operation Löschen** an, falls ein Pointer auf das zu löschende Element gegeben ist. *(2 Punkte)*

(d) Arrangieren Sie die Werte des folgenden Arrays $A = [8, 7, 7, 6, 5, 4, 3, 3, 3, 2, 1]$ um, so dass A einen **gültigen Min-Heap** repräsentiert. Sie können davon ausgehen, dass das Array mit Index 1 beginnt. *(2 Punkte)*

(e) Angenommen wir wissen, dass ein gegebener Algorithmus A für eine Eingabe der Größe n eine Laufzeit von $\Omega(n \log n)$ hat. Wir wissen außerdem, dass ein Algorithmus B für die selbe Eingabe eine Laufzeit von $\Theta(n^2)$ hat. Können wir folgern, dass A eine bessere **asymptotische** Laufzeit hat als B? **Begründen Sie** Ihre Antwort kurz. *(2 Punkte)*

(f) Sei $\Sigma = \{a, b, c, d, e\}$ ein Alphabet. Die Wahrscheinlichkeit (relative Häufigkeit) p_x für ein Zeichen $x \in \Sigma$, sei wie folgt:

$$p_a = \tfrac{3}{10},\ p_b = \tfrac{3}{10},\ p_c = \tfrac{2}{10},\ p_d = \tfrac{1}{10},\ p_e = \tfrac{1}{10}.$$

Konstruieren Sie eine **Huffman-Kodierung** von Σ bzgl. dieser Häufigkeitsverteilung mit dem Greedy-Verfahren aus der Vorlesung. *(2 Punkte)*

(g) Geben Sie die **erwartete (durchschnittliche) Codewortlänge** Ihrer zuvor berechneten Huffman Kodierung an. *(1 Punkt)*

Musterlösung

(a) $9, 6, 3, 8, 5$

$9, 6, 3, 5, 8$

$9, 6, 8, 3, 5$

Ein Punkt für die erste Folge, ein halber für jede weitere.

(b) In-Order: $3, 5, 6, 8, 9$ *($\frac{1}{2}$ Punkte)*

Pre-Order: $9, 6, 3, 5, 8$ *($\frac{1}{2}$ Punkte)*

Post-Order: $5, 3, 8, 6, 9$ *($\frac{1}{2}$ Punkte)*

Level-Order: $9, 6, 3, 8, 5$ *($\frac{1}{2}$ Punkte)*

(c) Eine einfach verkettete Liste benötigt weniger Speicherplatz da weniger Pointer gespeichert werden müssen. *(1 Punkt)*

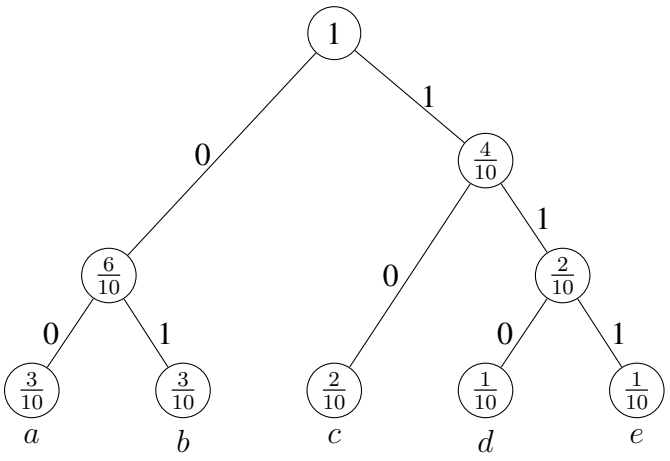
Die Laufzeit von Löschen (bei gegebenem Pointer auf das zu löschende Element) ist bei einfach verkettete Listen $\Theta(n)$ (da der Vorgänger ermittelt werden muss). Bei doppelt verketteten Listen hingegen $\mathcal{O}(1)$. *(1 Punkt)*

(d) Nein. *(1 Punkt)*

Der Grund ist dass für die Laufzeit von A nur eine untere Schranke für die asymptotische Laufzeit gegeben ist. A könnte mit basierend auf unserem Wissen trotzdem beliebig langsam sein. *(1 Punkt)*

(e) $A = (1, 2, 3, 3, 3, 4, 5, 6, 7, 7, 8)$. *Für jeden Knoten der die Min-Heap Eigenschaft verletzt gibt es einen Punkt Abzug.*

(f) Mit dem Verfahren aus der Vorlesung erhalten wir den folgenden Baum:



Wir lesen die folgende Kodierung ab:

$$a : 00,\ b : 01,\ c : 10,\ d : 110,\ e : 111.$$

(g) Die mittlere Codeworlänge ist $2 \cdot \frac{3}{10} + 2 \cdot \frac{3}{10} + 2 \cdot \frac{2}{10} + 3 \cdot \frac{1}{10} + 3 \cdot \frac{1}{10} = \frac{22}{10} = 2{,}2$. *(Rechnung nicht notwendig).*

Aufgabe 2: Landau Notation *(9 Punkte)*

Geben Sie an, ob die folgenden Behauptungen **wahr oder falsch** sind *(jeweils 1 Punkt)*. Beweisen Sie Ihre Aussage **anhand der Definitionen** der Landau Notation *(jeweils 2 Punkte)*.

(a) $10n^{1/3} \in \mathcal{O}(\sqrt{n})$

(b) $n! \in \Omega(n^n)$ *Dabei ist* $n! := n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$.

(c) $2^n + 2^n \in \Theta(4^n)$

Musterlösung

(a) Die Aussage ist wahr. *(1 Punkt)*

Wir zeigen dass es ein $c > 0, n_0 \in \mathbb{N}$ gibt sodass für alle $n \geq n_0$

$$\begin{aligned} & 10n^{1/3} \leq c \cdot \sqrt{n} \\ \iff \quad & 10 \leq c \cdot n^{1/6} \end{aligned}$$

Das ist beispielsweise für $c = 1$ für alle $n \geq 10^6 =: n_0$ der Fall. *(2 Punkte)*

Alternativ darf man zeigen, dass $\frac{10n^{1/3}}{\sqrt{n}}$ gegen einen endlichen Wert konvergiert:

$$\lim_{n\to\infty} \frac{10n^{1/3}}{\sqrt{n}} = 10 \cdot \lim_{n\to\infty} \frac{1}{n^{1/6}} = 10 \cdot 0 = 0.$$

(b) Die Aussage ist falsch. *(1 Punkt)*

Angenommen die Aussage wäre richtig, dann gibt es $c > 0, n_0 \in \mathbb{N}$ sodass für alle $n \geq n_0$

$$\begin{aligned} & n! \geq c \cdot n^n \\ \iff \quad & n \cdot (n-1) \cdots 3 \cdot 2 \geq c \cdot n^n \\ \implies \quad & n^{n-1} \geq c \cdot n^n \\ \iff \quad & \tfrac{1}{c} \geq n \end{aligned}$$

was für $n > \frac{1}{c}$ widersprüchlich ist.

Alternativ zeigen wir dass $n! \in o(n^n)$ was keine Schnittemenge mit $\Omega(n^n)$ hat.

$$0 \leq \lim_{n\to\infty} \frac{n!}{n^n} \leq \lim_{n\to\infty} \frac{n^{n-1}}{n^n} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Damit ist auch $\lim_{n\to\infty} \frac{n!}{n^n} = 0$ (Sandwich Theorem).

(c) Die Aussage ist falsch. *(1 Punkt)*

Angenommen es gäbe ein $c > 0, n_0 \in \mathbb{N}$ sodass für alle $n \geq n_0$, dann ist

$$\begin{aligned} & 2^n + 2^n \geq c \cdot 4^n \\ \iff \quad & \tfrac{1}{c} \geq \tfrac{4^n}{2^n + 2^n} \\ \iff \quad & \tfrac{1}{c} \geq 2^{n-1} \end{aligned}$$

ein Widerspruch für alle $n \geq 1 + \log \frac{1}{c}$. Daher $2^n + 2^n \notin \Omega(4^n)$ und somit $2^n + 2^n \notin \Theta(4^n)$. *(2 Punkte)*

Alternativ zeigen wir dass $2^n + 2^n \in o(4^n)$ was keine Schnittemenge mit $\Theta(4^n)$ hat.

$$\lim_{n \to \infty} \frac{2^n + 2^n}{4^n} = \lim_{n \to \infty} \frac{1}{2^{n-1}} = 0.$$

Aufgabe 3: Sortieralgorithmen *(9 Punkte)*

Gegeben sei der folgende Sortieralgorithmus `BucketSort` als Pseudocode. Der Algorithmus erhält als Argument eine **unsortierte, einfach verkettete Liste** L mit n Listenelementen, denen jeweils ein **positiver, ganzzahliger** Sortierschlüssel zugeordnet ist, sowie den größten Schlüssel m in L.

Algorithm 1 `BucketSort`(List L, max. key m)

```
Create new lists B_0, ..., B_m
while L is not empty do
    e ← L.pop()          ▷ L.pop() removes and returns the first element of L
    k ← e.key
    B_k.append(e)        ▷ Attach e to the tail of B_k
return B_0 ∘ ... ∘ B_m   ▷ The "∘" operator concatenates two lists in constant time
```

(a) Begründen Sie kurz die **Korrektheit** von `BucketSort` und geben Sie die **Laufzeit** in Abhängigkeit von m und n an. *(2 Punkte)*

Wir wollen nun die asymptotische Laufzeit von `BucketSort` der asymptotischen Laufzeit des Sortierverfahrens `QuickSort` gegenüberstellen. Gehen Sie davon aus, dass `QuickSort` die zu sortierende Schlüsselfolge in einem Array erhält und wir stets **das vorderste Element** des aktuellen (Teil-) Arrays als **Pivotelement** wählen.

(b) Geben Sie die Laufzeit dieser Variante von `QuickSort` für den Fall an, dass das Eingabearray $\boldsymbol{n}$ **mal den Schlüssel 1** enthält. **Begründen Sie Ihre Antwort.** *(3 Punkte)*

(c) Geben Sie die Laufzeit von `BucketSort` für den Fall an, dass die Eingabeliste $\boldsymbol{n}$ **mal den Schlüssel 1** enthält. *(1 Punkt)*

(d) Geben Sie für den **Worst-Case** und den **Best-Case** von `QuickSort` obere Schranken m_w, m_b für den größten Schlüssel m an, sodass `BucketSort` die Eingabefolge in dem entsprechenden Fall **asymptotisch** genauso schnell oder schneller sortiert als `QuickSort`. Drücken Sie m_w, m_b mit Hilfe der $\mathcal{O}$-Notation aus. *(3 Punkte)*

Musterlösung

(a) *Korrektheit*: Nach dem Durchlaufen der Schleife enthält jede Liste B_i (Bucket) alle Elemente aus L mit Schlüssel i und jedes Element von L befindet sich in einem Bucket. Da wir die B_i in aufsteigender Reihenfolge in einer Liste zusammenfassen ist die Liste danach sortiert. *(1 Punkt)*

Laufzeit: $\Theta(n+m)$ (ebenfalls ok ist $\mathcal{O}(n+m)$). *(1 Punkt)*

Das liegt daran dass die Buckets wahlfrei adressiert werden können (d.h. in $\mathcal{O}(1)$). Sollte jemand auf die Laufzeit $\Theta(nm)$ kommen, gibt das nur Punkte wenn auch erklärt wird warum das so sein soll. Wenn man beispielsweise argumentiert dass das entsprechende Bucket erst gesucht werden muss, weil die Buckets beispielsweise in einer Liste gespeichert sind (oder einer ähnlich langsamen Datenstruktur). Auch wenn das nicht optimal ist kann man in Kombination mit einer Erklärung den ganzen Punkt geben.

(b) $\Theta(n^2)$. Ok wäre auch $\mathcal{O}(n^2)$ zu schreiben (auch wenn man hier eigentlich die untere asymptotische Schranke betonen möchte). *(1 Punkt)*

Begründung: Quicksort trennt immer nur das Pivot vom Eingabearray ab und partitioniert dann alle restlichen Elemente auf eine Seite des Arrays (je nach Implementierung entweder **alle** nach rechts oder **alle** nach links). *(1 Punkt)*

Quicksort wird also einerseits rekursiv auf ein Array aufgerufen dass nur ein Element weniger hat, während die Rekursion auf das andere (leere) Teilarray sofort terminiert. Die Laufzeit ist also $\sum_{i=0}^{n} ci \in \Omega(n^2)$ (für eine geeignete Konstante $c > 0$). *(1 Punkt)*

(c) $\Theta(n)$. Ok wäre auch $\mathcal{O}(n)$. *(1 Punkt)*

(d) $m_w \in \mathcal{O}(n^2)$. *(1,5 Punkte)*

$m_b \in \mathcal{O}(n \log n)$. *(1,5 Punkte)*

Aufgabe 4: Hashing *(11 Punkte)*

Gegeben seien zwei leere Hashtabellen H_1 der Größe m_1 und H_2 der Größe m_2. Gegeben seien zudem die zugehörigen Hashfunktionen

$$h_1(x) := (x + 5) \mod m_1, \quad h_2(x, i) := (x + 2i) \mod m_2$$

wobei x der einzufügende Schlüssel ist und i die Anzahl der Kollisionen bei dem aktuellen Einfügeprozess. Mittels h_1 wollen wir Elemente in H_1 nach dem Prinzip **Hashing mit Chaining** einfügen. Mittels h_2 wollen wir dasselbe mit H_2 nach dem Prinzip **Hashing mit linearem Sondieren** tun.

(a) Geben Sie für ein allgemeines n eine Folge von n **paarweise verschiedenen** Schlüsseln $x_1, \ldots, x_n$ und einen Suchschlüssel x_j für $1 \leq j \leq n$ an, sodass nach Einfügen von $x_1, \ldots, x_n$ in dieser Reihenfolge das Finden von x_j mindestens $\Omega(n)$ Zeit benötigt. Tun Sie dies **sowohl für H_1 als auch für H_2** unter der Annahme, dass diese initial leer sind und $m_2 > 2n$. *(4 Punkte)*

(b) Sei $m_1 := 7$ und $m_2 := 11$. Fügen Sie die Schlüssel $5, 12, 1, 8, 19$ in der gegebenen Reihenfolge jeweils in **beide** Datenstrukturen H_1, H_2 ein und geben Sie deren Zustand nach dem Einfügen aller Schlüssel in den dafür vorgesehenen Tabellen an. *(3 Punkte)*

(c) Sei H eine weitere Hashtabelle der Größe 7, welche für die Kollisionsauflösung das Prinzip des **Cuckoo-Hashing** benutzt. Seien

$$g_1(x) := (x + 5) \mod 7, \quad g_2(x) := 2x \mod 7$$

die zugehörigen Hashfunktionen. Fügen Sie der Reihe nach die Schlüssel $1, 2, 4, 8, 9$ ein und geben Sie den Zustand von H **nach** Einfügen der 8 und der 9 in den vorgesehenen Tabellen an. *(4 Punkte)*

Musterlösung

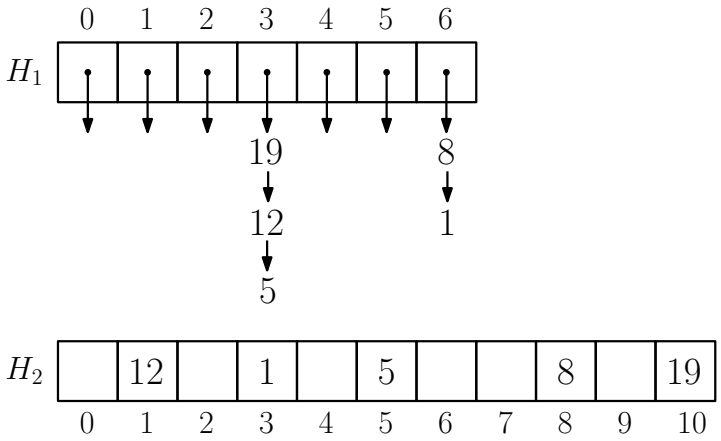
(a) Für H_1: Schlüsselfolge bspw. $x_i := i \cdot m_1$ *(1 Punkt)*

mit Suchschlüssel x_1. *(1 Punkt)*

Für H_2: Schlüsselfolge bspw. $x_i := i \cdot m_2$ *(1 Punkt)*

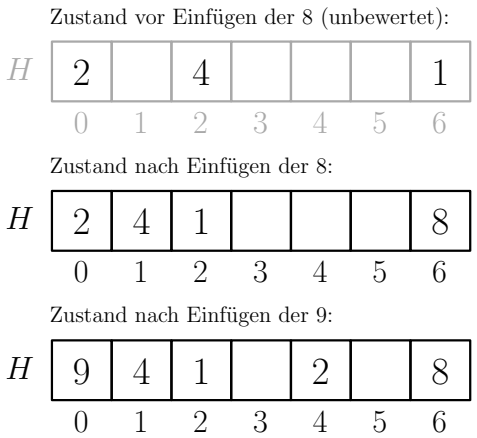
mit Suchschlüssel x_n. *(1 Punkt)*

(b) Tabellen mit Lösung:



1,5 Punkte pro korrekter Tabelle. Daumenregel: Ein Punkt Abzug pro falschem Wert. Es ist beim Hashing mit Chaining auch in Ordnung die Reihenfolge innerhalb einer Liste anders herum zu machen, denn man könnte das so implementieren dass immer am Ende der Liste eingefügt wird (mit einem zusätzlichen Pointer pro Liste). Meine Vermutung ist, dass bei H_1 viele das $+5$ in der Hashfunktion übersehen. Wenn sonst alles richtig ist (also alles nur 5 Zellen nach links verschoben ist) kann man immerhin noch einen halben Punkt geben.

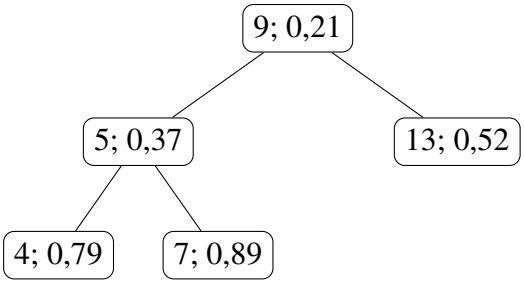
(c) Tabellen mit Lösung:



Zwei Punkte pro korrekter Tabelle. Daumenregel: Ein Punkt Abzug pro falschem Wert. Wenn es dann nicht zu einfach wird (keine Verdrängung), kann man bei der zweiten Tabelle Folgefehler berücksichtigen.

Aufgabe 5: Treap *(12 Punkte)*

Betrachten Sie den folgenden **Treap**, dessen Knoten Paare von Schlüsseln $(key1; key2)$ enthalten.

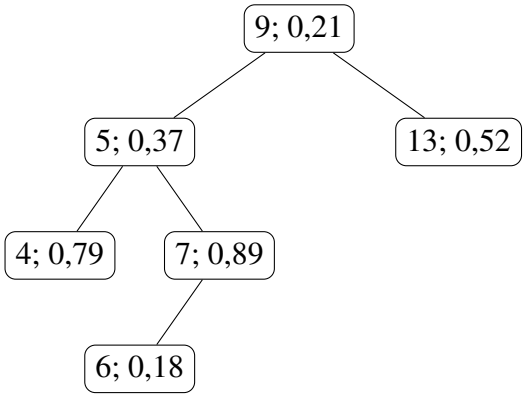


(a) **Fügen Sie das Paar (6; 0,18) in den Treap ein** und führen Sie die notwendigen Operationen durch, um die Treap Eigenschaften wiederherzustellen. Geben Sie nach **jeder** Rotation den entsprechenden Baum an und dokumentieren Sie kurz Ihre Schritte. *(6 Punkte)*

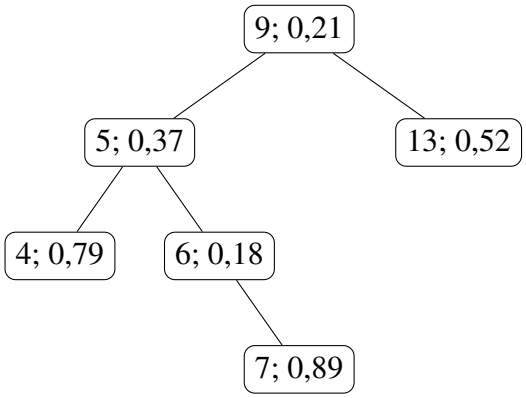
(b) Führen Sie die notwendigen Operationen durch, um **den Knoten mit dem Schlüssel 9** aus dem **ursprünglichen** Treap zu **löschen** (siehe oben). Geben Sie nach **jeder** Rotation den entsprechenden Baum an und dokumentieren Sie kurz Ihre Schritte. *(6 Punkte)*

Musterlösung

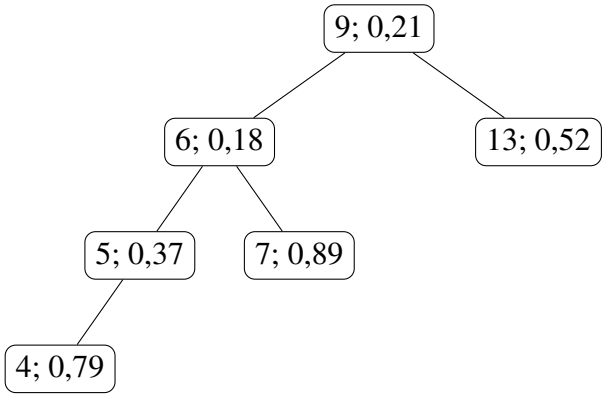
(a) Füge $(6; 0{,}18)$ zunächst entsprechend der binären Suchbaumeigenschaft ein:



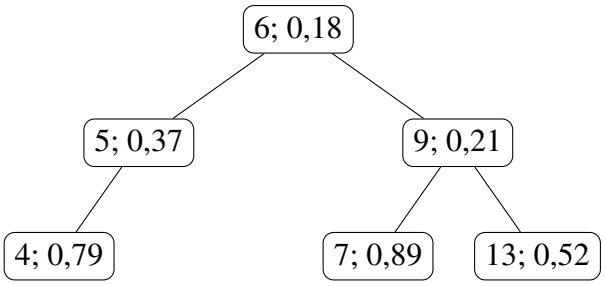
`rotate-right`(7,6): *(2 Punkte)*



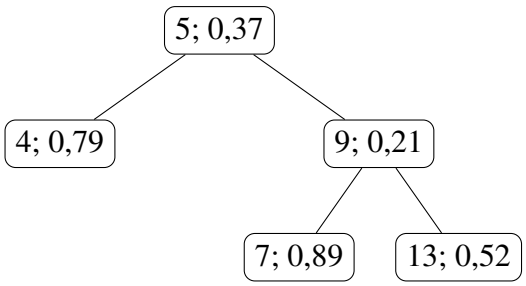
`rotate-left`(5,6): *(2 Punkte)*



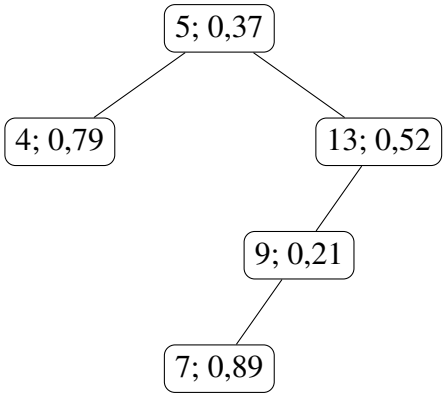
`rotate-right`(9,6): *(2 Punkte)*



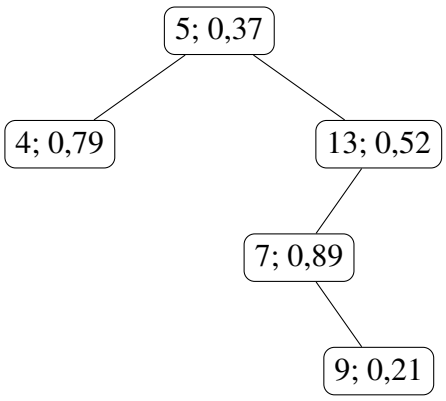
(b) `rotate-right`(9,5): *(2 Punkte)*



`rotate-left`(9,13): *(2 Punkte)*



Da die 9 jetzt nur noch ein Kind hat können wir sie direkt löschen und die 7 direkt an die 13 hängen. *Alternativ* kann man die 9 mittels `rotate-right`(9,7) auch noch weiter bis zu einem Blatt nach unten rotieren und erst dort löschen: *(2 Punkte)*



Aufgabe 6: Algorithmenanalyse *(12 Punkte)*

Betrachten Sie folgenden Algorithmus, der als Eingabe ein Array $A[0, \ldots, n-1]$ der Länge n erhält:

Algorithm 2 `algorithm`(A)

```
for i ← 2 to n − 1 do
    for j ← 1 to i − 1 do
        for k ← 0 to j − 1 do
            if |A[k] − A[j]| = |A[k] − A[i]| then
                return True
return False
```

(a) Geben Sie die **Ausgabe** von `algorithm` für die Eingaben $A_1 = [3, 2, 6, 0]$, $A_2 = [4, 1, 5, 6]$ und $A_3 = [-3, 0, 3, 6]$ an. *(3 Punkte)*

(b) Geben Sie die asymptotische Laufzeit von `algorithm` in Abhängigkeit von n an. *(2 Punkte)*

(c) Beschreiben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(n \log n)$, der für ein Array $A[0, \ldots, n–1]$ überprüft, ob es $i, j \in \{0, \ldots, n-1\}$ mit $i \neq j$ gibt, sodass $A[i] = A[j]$. *(2 Punkte)*

(d) Beschreiben Sie einen Algorithmus, der für jedes Array die gleiche Ausgabe erzeugt wie `algorithm`, aber eine bessere asymptotische Laufzeit hat. Ihr Algorithmus darf einen höheren Speicherbedarf haben als `algorithm`. Erklären Sie die Laufzeit Ihres Algorithmus. *(5 Punkte)*

Hinweis: Sie dürfen den Algorithmus aus (c) *als Blackbox verwenden, auch wenn Sie* (c) *nicht gelöst haben.*

Musterlösung

(a) `algorithm`(A_1) =`True`, `algorithm`(A_2) =`algorithm`(A_3) =`False`.

(b) $\Theta(n^3)$. $\mathcal{O}(n^3)$ ist ebenfalls in Ordnung.

(c) Man sortiert das Array $(\mathcal{O}(n \log n))$ und überprüft es in einem Durchlauf auf gleiche Werte, welche dann, falls vorhanden, nebeneinander stehen $(\mathcal{O}(n))$.

(d) Für jedes $k \in \{0, \ldots, n-3\}$ erstellt man eine Liste mit Einträgen $|A[k] - A[j]|$ für $k < j < n-1$ $(\mathcal{O}(n^2))$. *(2 Punkte)*

Dann überprüft man jede Liste auf gleiche Werte. Dies benötigt nach (c) $\mathcal{O}(n \log n)$ für eine einzelne Liste, also insgesamt $\mathcal{O}(n^2 \log n)$. *(2 Punkte)*

Zusammen benötigt man also $\mathcal{O}(n^2 + n^2 \log n) = \mathcal{O}(n^2 \log n)$. *(1 Punkt)*

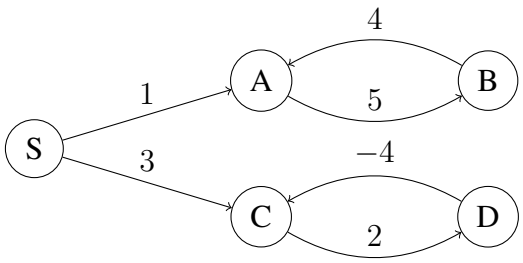
Aufgabe 7: Graphenalgorithmen *(13 Punkte)*

Sei $G = (V, E, w)$ ein ungerichteter, gewichteter Graph. Angenommmen es gebe **für jeden Schnitt** $(S, V \setminus S)$ eine **eindeutige leichte Schnittkante**, d.h. eine Schnittkante, deren Gewicht echt kleiner als das aller anderen Schnittkanten ist.

(a) Zeigen Sie, dass G einen minimalen Spannbaum hat und das dieser **eindeutig** ist. *(6 Punkte)*

(b) Folgt umgekehrt aus der Existenz eines eindeutigen minimalen Spannbaums auch die Existenz einer eindeutigen leichten Schnittkante für jeden Schnitt? **Beweisen Sie** Ihre Antwort. *(3 Punkte)*

Gegeben sei der folgende gewichtete, gerichtete Graph:



(c) Führen Sie auf diesem Graph den **Bellman-Ford** Algorithmus mit Startknoten S aus und tragen Sie **nach jeder** Iteration der äußeren Schleife die bisher berechneten Distanzen von S **in der vorgegeben Tabelle** ein (siehe Lösungsblatt). Iterieren Sie in der inneren Schleife die Kanten **aufsteigend nach Größe der Gewichte**. *(4 Punkte)*

Musterlösung

(a) *Existenz eines MST:* Zeige, dass G zusammenhängend ist. Sei dazu $u \in V$ und S die Zusammenhangskomponente, welche u enthält. Falls $S \neq V$, gibt es nach Voraussetzung eine Kante $\{v, w\} \in E$ mit $v \in S$ und $w \in V \setminus S$. Da w von u aus erreichbar ist, muss $w \in S$ gelten, Widerspruch. *(2 Punkte)*

Eindeutigkeit des MST: Angenommen es gebe zwei verschiedene MST T und T'. Dann gibt es eine Kante $\{u, v\}$ in T, welche nicht in T' ist. Durch Löschen von $\{u, v\}$ aus T zerfällt T in zwei Zusammenhangskomponenten, deren Knotenmengen wir mit T_u und T_v bezeichnen (mit $u \in T_u$ und $v \in T_v$) *(1 Punkt)*.

Sei $\{x, y\}$ die eindeutige leichteste Schnittkante zwischen T_u und T_v. Falls $\{x, y\} \neq \{u, v\}$, so kann man in T $\{u, v\}$ durch $\{x, y\}$ ersetzen und erhält einen Spannbaum mit kleinerem Gewicht als T, Widerspruch. *(1 Punkt)*

Nehme also an, dass $\{u, v\}$ die eindeutige leichteste Kante zwischen T_u und T_v ist. Sei p ein Pfad in T' von u nach v und e eine Kante im Pfad, welche Schnittkante von (T_u, T_v) ist. Fügt

man $\{u, v\}$ zu T' hinzu, erhält man einen Kreis bestehend aus p und $\{u, v\}$. Durch Löschen einer beliebigen Kante aus dem Kreis erhält man wieder einen Spannbaum. Fügt man also $\{u, v\}$ zu T' hinzu und löscht e, erhält man wegen $w(\{u, v\}) < w(e)$ einen Spannbaum mit kleinerem Gewicht, Widerspruch. *(2 Punkte)*

(b) Sei $V = \{x, y, z\}$, $E = \{\{x, y\}, \{x, z\}\}$ und $w(\{x, y\}) = w(\{x, z\}) = 1$. Dann hat $G = (V, E)$ einen eindeutigen MST (G selbst), aber der Schnitt $(\{x\}, \{y, z\})$ hat keine eindeutige leichte Schnittkante. *(3 Punkte)*

(c) Es gibt höchstens einen Punkt pro korrekter Zeile (Folgefehler können ggf. berücksichtigt werden). Innerhalb einer Zeile gibt es einen halben Punkt Abzug pro falschem Wert. Die Reihenfolge der Relaxierung der Kante muss eingehalten werden (siehe Aufgabe)! *(4 Punkte)*

	$\delta(S, S)$	$\delta(S, A)$	$\delta(S, B)$	$\delta(S, C)$	$\delta(S, D)$
Initial.	0	∞	∞	∞	∞
$i = 1$	0	1	6	3	∞
$i = 2$	0	1	6	3	5
$i = 3$	0	1	6	1	3
$i = 4$	0	1	6	-1	1

Aufgabe 8: Dynamische Programmierung *(11 Punkte)*

Betrachten Sie die folgenden Funktionen auf natürlichen Zahlen:

$$f_1(n) = n - 1$$

$$f_2(n) = \begin{cases} \frac{n}{2} & \text{falls } 2 \mid n \\ n & \text{sonst} \end{cases}$$

$$f_3(n) = \begin{cases} \frac{n}{3} & \text{falls } 3 \mid n \\ n & \text{sonst} \end{cases}$$

$m \mid n$ ("m teilt n") bedeutet es gibt ein $k \in \mathbb{N}$ so dass $k \cdot m = n$.

Man betrachtet folgendes Problem: Für ein gegebenes $n \geq 1$, finde die **minimale Anzahl an Anwendungen** von f_1, f_2, f_3, die man benötigt, um 1 zu erhalten. Formal: Finde das minimale k, so dass es $i_1, \ldots, i_k \in \{1, 2, 3\}$ gibt mit $f_{i_1}(f_{i_2}(\ldots(f_{i_k}(n))\ldots) = 1$.

Ein Student schlägt dazu folgenden Algorithmus vor:

Algorithm 3 `steps_to_one`(n)

```
s ← 0
while n > 1 do
    if 3 | n then
        n ← n/3
    else if 2 | n then
        n ← n/2
    else
        n ← n − 1
    s ← s + 1
return s
```

(a) Welches Algorithmendesign-Prinzip benutzt der obige Algorithmus? *(1 Punkt)*

(b) Löst `steps_to_one` für jedes $n \geq 1$ das gegebene Problem? Beweisen Sie Ihre Antwort. *(3 Punkte)*

(c) Geben Sie einen Algorithmus **in Pseudocode** an, der das Prinzip der dynamischen Programmierung anwendet, um das obige Problem zu lösen. Analysieren Sie die (asymptotische) Laufzeit ihres Algorithmus. Diese sollte maximal polynomiell in n sein. *(7 Punkte)*

Musterlösung

(a) Beim gegebenen Algorithmus handelt es sich um einen Greedy-Algorithmus.

(b) Für $n = 10$ führt der Algorithmus folgende Schritte aus:

$$10 \xrightarrow{f_2} 5 \xrightarrow{f_1} 4 \xrightarrow{f_2} 2 \xrightarrow{f_2} 1$$

Optimal hingegen ist

$$10 \xrightarrow{f_1} 9 \xrightarrow{f_3} 3 \xrightarrow{f_3} 1$$

(c) Algorithmus: *(5 Punkte)*

memo $= \{\}$

Algorithm 4 `steps_to_one`(n)

```
1:  if n in memo then
2:      return memo[n]
3:  if n == 1 then
4:      s = 0
5:  else
6:      x = steps_to_one(n − 1)
7:      if n | 2 then
8:          y = steps_to_one(n/2)
9:      else
10:         y = ∞
11:     if n | 3 then
12:         z = steps_to_one(n/3)
13:     else
14:         z = ∞
15:     s = 1 + min{x, y, z}
16: memo[n] = s
17: return s
```

Laufzeit: *(2 Punkte)*

Es gibt n Aufrufe, bei denen nicht der Wert aus *memo* genommen wird. Ein Aufruf benötigt $\mathcal{O}(1)$ (Rekursion ignoriert). Die Laufzeit beträgt also $\mathcal{O}(n)$.