Algorithmen 1 **1. Klausur** **Sommersemester 2022**

01.09.2022

Vorname: _____

Nachname: _____

Matrikel-Nummer: _____

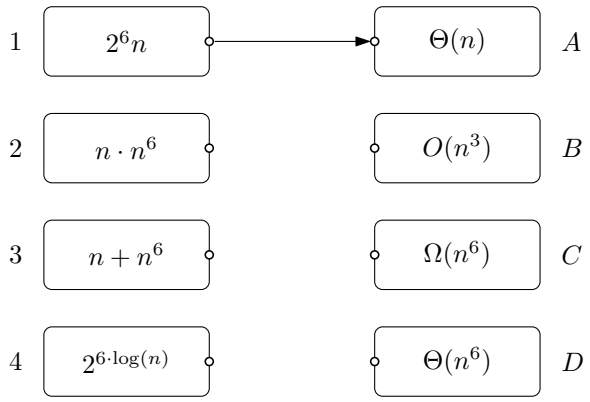
- Bringe den Aufkleber mit deinem Namen und deiner Matrikelnummer oben auf diesem Deckblatt an und beschrifte jedes Aufgabenblatt mit deiner Matrikelnummer.
- Diese Klausur umfasst 11 Seiten (diese Titelseite eingeschlossen) und 6 Aufgaben. Es können maximal 60 Punkte erreicht werden.
- Schreibe die Lösungen auf die Aufgabenblätter und Rückseiten. Am Ende der Klausur sind zusätzliche Leerseiten. Fordere zusätzliches Papier bitte nur an, falls du den gesamten Platz aufgebraucht hast.
- Es werden nur Lösungen gewertet, die mit dokumentenechten Stiften geschrieben sind.
- Als Hilfsmittel ist ein A4-Papier mit beliebigem Inhalt erlaubt.
- Die Tackernadel darf nicht gelöst werden.
- Die Bearbeitungszeit beträgt 2 Stunden.
- Schreibe nicht in die Tabelle auf dieser Seite.

Aufgabe	Mögliche Punkte	Erreichte Punkte
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
Gesamt	60	

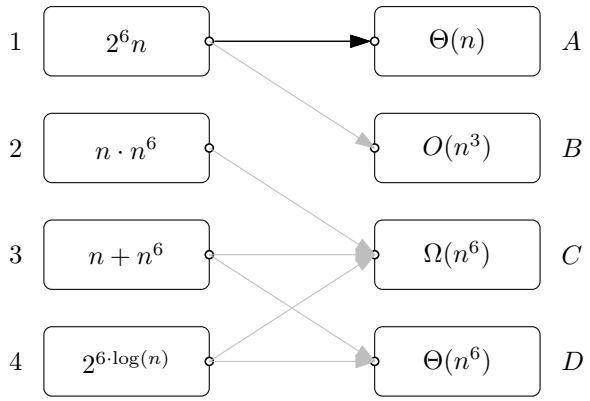
1. **Kleinaufgaben** [10 Punkte]

Hinweis: $\log(n) = \log_2(n)$, $\log^2(n) = \log(n) \cdot \log(n)$ und $\log\log(n) = \log(\log(n))$.

(a) Im folgenden Graphen sollen von jedem Knoten auf der linken Seite $(1, 2, 3, 4)$ genau die gerichteten Kanten zu Knoten auf der rechten Seite (A, B, C, D) eingezeichnet werden, für die die zugehörige Funktion in der jeweiligen Menge enthalten ist. So beschreibt die bereits eingezeichnete Kante $(1,\ A)$, dass $2^6 n \in \Theta(n)$ gilt. Zeichne die fehlenden Kanten ein, oder gib die zugehörigen Tupel aus Zahlen und Buchstaben an.
Hinweis: Es könnte Knoten geben, die keine inzidenten Kanten haben. (4 Punkte)



Lösung:



$(1, B), (2, C), (3, C), (3, D), (4, C), (4, D)$

(b) Im Folgenden sollen Summen asymptotisch abgeschätzt werden. Gib für jede Summe $f_j(n)$ eine Funktion $g_j(n)$ an, sodass $f_j(n) \in \Theta(g_j(n))$. Vereinfache dabei die Funktion $g_j(n)$ so weit wie möglich. (3 Punkte)

$$f_1(n) = \sum_{i=0}^{\log(n)} 4^i \qquad f_2(n) = \sum_{i=1}^{n} \frac{2^4}{2^i} \qquad f_3(n) = \sum_{i=\log(n)}^{2\log(n)} \left(\frac{\log(2^n)}{n}\right)^i$$

Lösung:

$f_1(n) \in \Theta(n^2)$ (exponentielle Summe wird asymptotisch vom größten Summanden dominiert)
$f_2(n) \in \Theta(1)$ (geometrische Summe)
$f_3(n) \in \Theta(\log(n))$ (es wird über 1 summiert)

(c) Bestimme für folgende Situationen, welche möglichst einfache Datenstruktur jeweils geeignet ist und gib an, welche Operationen der von dir gewählten Datenstruktur dabei relevant sind. (3 Punkte)

Hinweis: In der Vorlesung haben wir folgende Datenstrukturen kennengelernt: Listen, (dynamische) Arrays, Hashtabellen, binäre Heaps, $(2,3)$-Bäume, Union-Find, Adjazenzliste.

1. Bei einem Radrennen tragen die Teilnehmer:innen Sensoren mit sich, die regelmäßig ihren prozentualen Fortschritt der Strecke an die Rennleitung mitteilen (Beispielmeldung: „Teilnehmer Marcus hat gerade 45,7 % der Strecke zurückgelegt.“). Die Rennleitung möchte die Positionen der Teilnehmer:innen so verwalten, dass schnell alle Teilnehmer:innen auf einem bestimmten Streckenabschnitt gefunden werden können (Beispiel: finde alle Teilnehmer:innen mit Fortschritt zwischen 14,6 % und 17,3 %).

   Lösung: $(2,3)$-Baum. Aktualisieren (Löschen und Einfügen) und Suchen.

2. Prüflingen werden Noten zugewiesen. Jede:r der n Prüflinge ist durch eine ID aus $\{0,\dots,n-1\}$ eindeutig identifiziert. Die Prüflinge können im System ihre Noten abfragen.

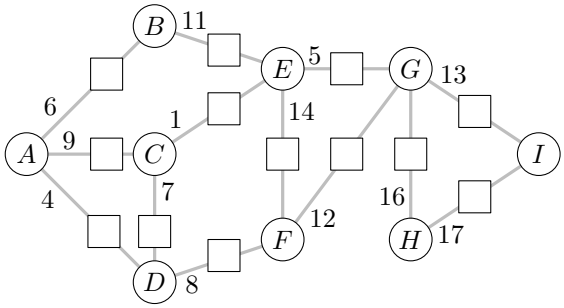
   Lösung: Array. `pushBack` und wahlfreier Zugriff.

3. Prüflingen werden Noten zugewiesen. Jede:r der n Prüflinge ist durch eine Matrikel-Nummer identifiziert, die potenziell wesentlich größer ist als n. Die Prüflinge können im System ihre Noten abfragen.

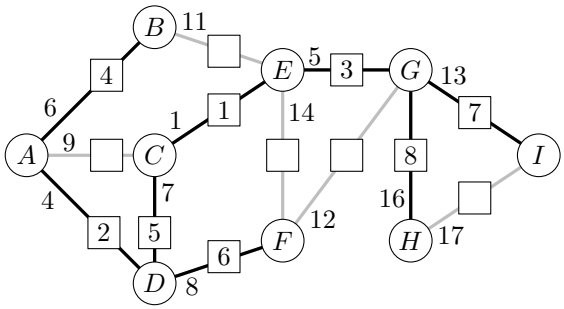
   Lösung: Hashmap. Einfügen, Suchen.

2. **Kruskals Algorithmus** [10 Punkte]

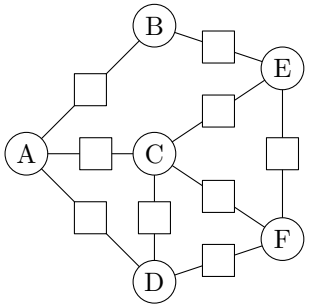
(a) Gegeben sei folgender gewichteter Graph, in dem mittels Kruskals Algorithmus ein minimaler Spannbaum berechnet werden soll. Zeichne die Kanten des minimalen Spannbaums in die Abbildung ein und benutze die zugehörigen Kästchen, um die Reihenfolge anzugeben, in der sie von Kruskals Algorithmus hinzugefügt werden.
Hinweis: Beachte, dass die Kästchen von Kanten, die *nicht* zum minimalen Spannbaum gehören, leer bleiben sollen. (3 Punkte)

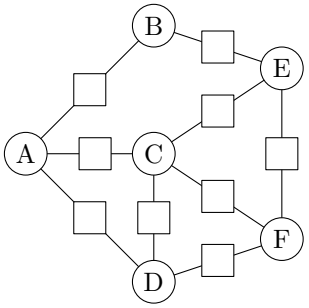


Lösung:



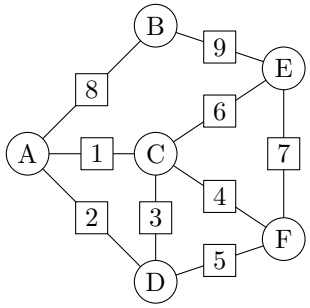
(b) Gegeben sei der folgende Graph. Verteile die Gewichte $1, 2, 3, 4, 5, 6, 7, 8, 9$ auf die Kanten, sodass Kruskals Algorithmus die Kanten mit Gewicht $3, 5, 7$ und 9 *nicht* zum MST hinzufügt. Achte darauf, dass jedes Gewicht genau einmal verwendet wird. Schreibe die Kantengewichte in die Kästchen. (4 Punkte)





(Kopie desselben Graphen, falls du dich beim ersten vertan hast. Markiere deutlich, welche Lösung zu bewerten ist.)

Lösung:

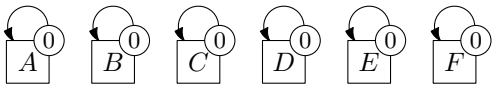


(c) Zeige oder widerlege: Für jeden zusammenhängenden, gewichteten Graphen $G = (V, E)$ mit $|E| > 2|V|$ gilt, dass die schwerste Kante nicht Teil des MST ist. Nimm an, dass die Kantengewichte paarweise verschieden sind. (3 Punkte)

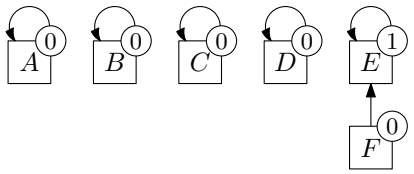
Lösung: Widerlegen. Gegenbeispiel: Betrachte einen Graphen bestehend aus einer Clique mit 6 Knoten und einem weiteren Knoten v der nur zu einem Knoten der Clique verbunden ist. Der Graph hat $(6 \cdot 5)/2 + 1 = 16$ Kanten, also mehr als $2n = 14$. Nun bekommen die 15 Kanten der Clique die Gewichte $1, \ldots, 15$ zugewiesen und die Kante e die inzident zu v ist, das Gewicht 16. Jeder MST muss e enthalten, weil diese den Schnitt darstellt der v vom Rest das Graphen trennt. Zudem ist e die schwerste Kante.

3. **Union-Find** [10 Punkte]

(a) Gegeben seien die Elemente A, B, ..., F, die in einer Union-Find Datenstruktur verwaltet werden. In der folgenden Visualisierung steht der Rang eines Elements oben rechts neben dem zugehörigen Knoten.



Es werden nun `union` und `find` Operationen unter Verwendung von *Union-by-Rank* und *Pfadkompression* ausgeführt. Falls die Wurzeln zweier Bäume den gleichen Rang haben, dann wählt `union` den alphabetisch kleineren Knoten als neue Wurzel. Nach `union`(E, F) sieht die Datenstruktur folgendermaßen aus.

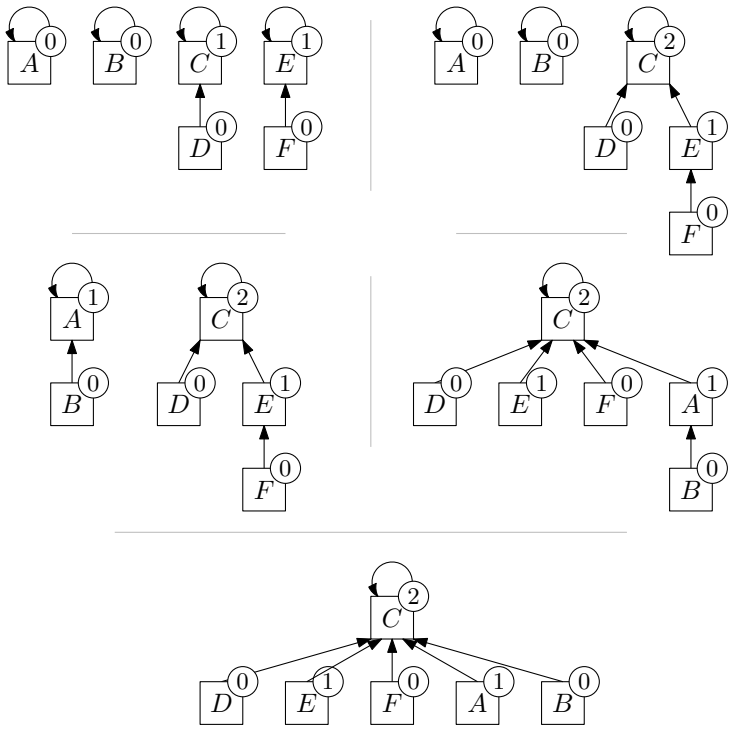


Führe darauf aufbauend folgende Operationen der Reihe nach aus und gib den Zustand der Datenstruktur schrittweise nach jeder dieser Operationen an (4 Punkte)

`union`(C, D), `union`(C, E), `union`(A, B), `union`(A, F), `find`(B).

Hinweis: Zur Erinnerung: `union` führt auch ein `find` auf beiden Knoten aus.

Lösung:



(b) Knoten der Union-Find Datenstruktur speichern wie folgt Daten, wobei `node.parent` = `node` genau dann, wenn Knoten `node` die Wurzel ist:

struct NODE
  `parent`: NODE
  `rank`: $\mathbb{N}$
  `value`: OBJECT

Gib Pseudocode für eine *iterative* Implementierung der `find` Funktion mit Pfadkompression an. Das heißt, die Funktion soll für einen gegebenen Knoten die Wurzel finden und ausgeben und die Elter-Zeiger der

Knoten auf dem Pfad zur Wurzel aktualisieren. (6 Punkte)

`find`(`node`: NODE) : NODE

Lösung:

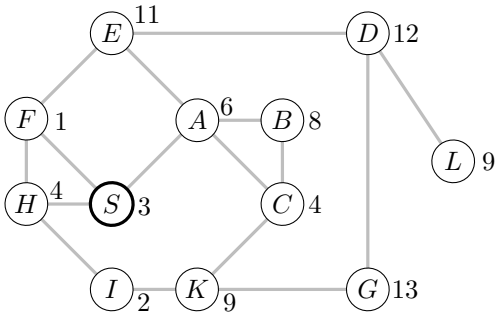
```
root = node
while root.parent ≠ root do
    root := root.parent
while node ≠ root do
    next := node.parent
    node.parent := root
    node := next
return root
```

4. **Zeitreisen** [10 Punkte]

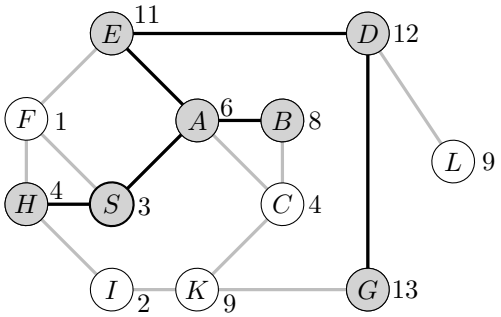
Sei $G = (V, E)$ ein Graph, bei dem jeder Knoten $v \in V$ mit einem Jahr $J(v) \in \mathbb{N}$ versehen ist. Ein Pfad $\pi = \langle v_0, v_1, \ldots, v_k \rangle$ wird nun als *zeitlich plausibel* bezeichnet, wenn für alle $i \in \{1, \ldots, k\}$ gilt, dass $J(v_{i-1}) \leq J(v_i)$. Für einen Knoten $s \in V$ bezeichnen wir einen Knoten $t \in V$ als *von s aus plausibel erreichbar*, wenn es einen zeitlich plausiblen st-Pfad gibt.

(a) Gegeben sei folgender Graph, bei dem das Jahr eines Knotens jeweils rechts vom Knoten steht. Markiere die Knoten, die von S aus plausibel erreichbar sind und zeichne die Kanten der zugehörigen Pfade ein. (4 Punkte)

Hinweis: Stelle sicher, dass aus der Markierung klar hervorgeht, welche Knoten und Kanten Teil der Lösung sind.



Lösung:



S, A, B, D, E, G, H

(b) Beschreibe einen möglichst effizienten Algorithmus in Worten, der als Eingabe einen Graphen $G = (V, E)$, eine Jahresfunktion $J \colon V \to \mathbb{N}$ und einen Knoten $s \in V$ erhält und die von s aus plausibel erreichbaren Knoten ausgibt. Nenne und begründe das asymptotische Laufzeitverhalten deines Algorithmus.(6 Punkte)

Hinweis: Pseudocode wird mit 0 Punkten bewertet.

Lösung: Wir erstellen eine gerichtete Kopie G' von G in der es genau dann eine Kante (u, v) gibt, wenn $\{u, v\} \in E$ und $J(u) \leq J(v)$. Anschließend werden alle Knoten ausgegeben, die bei einer BFS in G' startend bei s gefunden werden.

Folgendes ist nicht Teil der Lösung, sondern soll lediglich helfen zu Verstehen, warum die Lösung korrekt ist:

Da bei jeder Kante in G' das Jahr des Endknotens größer als das des Startknotens ist, sind die zeitlich plausiblen Pfade in G genau die gerichteten Pfade in G'. Demzufolge sind alle von s aus erreichbaren Knoten in G', die von s aus plausibel erreichbaren Knoten in G. Diese werden von der BFS in G' ausgegeben.

Sowohl das Erstellen der Kopie, als auch die BFS in G' benötigen $O(|V| + |E|)$ Zeit. Der Algorithmus hat demzufolge das gleiche asymptotische Laufzeitverhalten.

5. **Vielfaches Programm** [10 Punkte]

Sei $A = \langle a_0, a_1, \ldots, a_n \rangle$ eine Folge natürlicher Zahlen. Jede Folge, die man erhält, indem man Elemente aus A entfernt und die Reihenfolge der übrigen Elemente unverändert lässt, nennen wir *Teilfolge* von A. Zudem bezeichnen wir eine Folge $B = \langle b_0, b_1, \ldots, b_k \rangle$ als *vervielfachend*, wenn es für jedes $i \in \{1, \ldots, k\}$ ein $c \in \mathbb{N}$ gibt, sodass $b_i = c \cdot b_{i-1}$. Beachte, dass eine Teilfolge auch dann vervielfachend ist, wenn sie nur ein Element enthält.

Für die Folge $A = \langle 6, 5, 2, 14, 12, 28, 36 \rangle$ sind beispielsweise $B_1 = \langle 6, 12, 36 \rangle$ (hier ist c jeweils 2 und 3) und $B_2 = \langle 2, 14, 28 \rangle$ (hier ist c jeweils 7 und 2) vervielfachende Teilfolgen.

Beim Problem MaximaleVervielfachung wird die Länge einer längsten, vervielfachenden Teilfolge von A gesucht.

(a) Gib eine längste vervielfachende Teilfolge in folgender MaximaleVervielfachung-Instanz an.(2 Punkte)

$$A = \langle 3, 2, 9, 23, 18, 6, 24, 83, 48 \rangle$$

Lösung:

$$B = \langle 2, 6, 24, 48 \rangle \ (c = 3, 4, 2)$$

Es soll nun ein dynamisches Programm über die Indizes der Folge $A = \langle a_0, a_1, \ldots, a_n \rangle$ entwickelt werden, welches MaximaleVervielfachung für A löst. Dazu wird ein Array T angelegt, um die Teillösungen zu verwalten.

Hinweis: Bedenke, dass nur die *Länge* der längsten vervielfachenden Teilfolge gesucht wird und *nicht* die Teilfolge selbst.

(b) Definiere für jedes i welche Bedeutung $T[i]$ haben soll und stelle darauf aufbauend die Rekurrenz auf.(6 Punkte)

Lösung: $T[i]$ enthält die Länge der längsten vervielfachenden Folge die in a_i endet.
Sei $D_i \subseteq \{1, \ldots, i-1\}$ die Menge der Indizes sodass für jedes $j \in D_i$ das Element a_j ein Teiler von a_i ist. Dann ist die Rekurrenz gegeben durch

$$T[i] = 1 + \max\{\{0\} \cup \{T[j] \mid j \in D_i\}\}$$

(c) Gib an, wie man aus dem Array T schließlich die Länge der längsten vervielfachenden Teilfolge von A bestimmen kann. (2 Punkte)

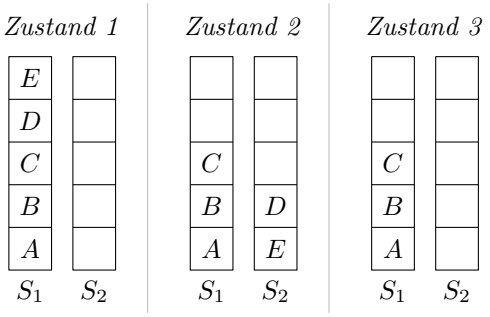
Lösung: Die Lösung ergibt sich aus $\max_{i \in [n]}\{T[i]\}$.

6. **Amortisierter Doppelstapler** [10 Punkte]

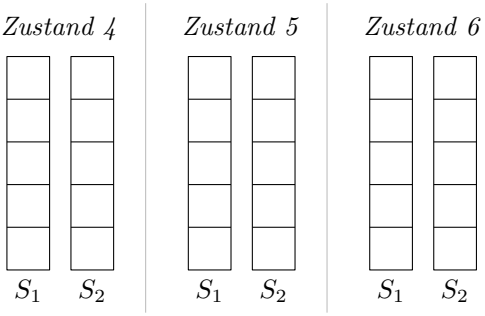
Wir betrachten eine Datenstruktur D, die aus zwei Stacks S_1 und S_2 besteht, deren Anzahl enthaltener Elemente jeweils mit n_1 und n_2 bezeichnet werden. Die Datenstruktur bietet folgende Operationen:

- `add`: Fügt ein Element zu S_1 hinzu und hat Laufzeit $\Theta(1)$.
- `move`: Verschiebt $\lfloor \frac{n_1}{2} \rfloor$ viele Elemente von S_1 nach S_2. Dabei wird schrittweise das oberste Element aus S_1 entfernt und oben in S_2 eingefügt. Die Operation hat insgesamt eine Laufzeit von $\Theta(n_1)$.
- `melt`: Entfernt alle Elemente aus S_2 und hat eine Laufzeit von $\Theta(n_2)$.

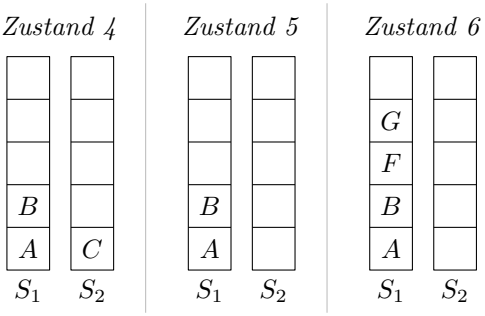
Folgende Abbildung zeigt den Zustand der Datenstruktur nach `add`(A), `add`(B), `add`(C), `add`(D), und `add`(E) (*Zustand 1*), einem darauffolgenden `move` (*Zustand 2*) und schließlich einem `melt` (*Zustand 3*).



(a) Gib die drei weiteren Zustände an, die man erhält, wenn man ausgehend von *Zustand 3* erst `move` (*Zustand 4*), dann `melt` (*Zustand 5*) und schließlich `add`(F) und `add`(G) (*Zustand 6*) ausführt. (2 Punkte)



Lösung:



(b) Ausgehend von einer leeren Datenstruktur sei nun die folgende Sequenz von Operationen gegeben

(n mal `add`), `move`, `melt`.

Nenne und begründe die asymptotische Laufzeit der Sequenz in Abhängigkeit von n. (2 Punkte)

Lösung: Die Ausführung von n `add`s benötigt $\Theta(n)$ Zeit. Dann gilt $n_1 = n$. Ein `move` benötigt nun $\Theta(n_1) = \Theta(n)$ Zeit. Anschließend ist $n_2 = \lfloor \frac{n_1}{2} \rfloor = \lfloor \frac{n}{2} \rfloor$, weswegen `melt` auch eine Laufzeit in $\Theta(n_2) = \Theta(n)$ hat. Insgesamt liegt die Laufzeit der Sequenz damit in $\Theta(n)$

(c) Zeige, dass in jeder beliebigen Abfolge von `add`, `move` und `melt` Operationen jede Operation amortisiert konstante Kosten hat. (6 Punkte)

Lösung: Die Kernidee ist, dass jedes eingefügte Element nur jeweils ein Mal von `move` verschoben und von `melt` entfernt werden kann. Wir verwenden die Charging-Methode um die Kosten von `move` bzw. `melt` auf die jeweils zugehörigen `add` Operationen zu chargen.

Konkret verursachen die Operationen Kosten Token entsprechen ihrer Laufzeit. Das heißt `add` verursacht ein Token, `move` verursacht n_1 Token und `melt` verursacht n_2 Token. Für die Amortisierung chargen wir die Token jetzt wie folgt.

- Ein `move` verschiebt $\lfloor \frac{n_1}{2} \rfloor$ Elemente. Für jedes verschobene Element e werden zwei Token der `add` Operation zugewiesen, die e eingefügt hat. Damit verbleibt höchstens ein Token bei `move`.
- Ein `melt` löscht n_2 Elemente. Für jedes gelöschte Element e wird ein Token der `add` Operation zugewiesen, die e eingefügt hat. Damit verbleiben keine Token bei `melt`.

Da `move` und `melt` nur $O(1)$ Kosten Token behalten, haben sie amortisiert konstante Laufzeit. Außerdem bekommt jedes `add` höchstens zwei Token von `move` und höchstens ein Token von `melt` Operationen zugewiesen, da jedes Element höchstens ein Mal verschoben und höchstens ein Mal gelöscht wird. Damit ist auch `add` amortisiert konstant.