Algorithmen 1 **2. Klausur** **Wintersemester 2023**

24.03.2023

Vorname: _____

Nachname: _____

Matrikel-Nummer: _____

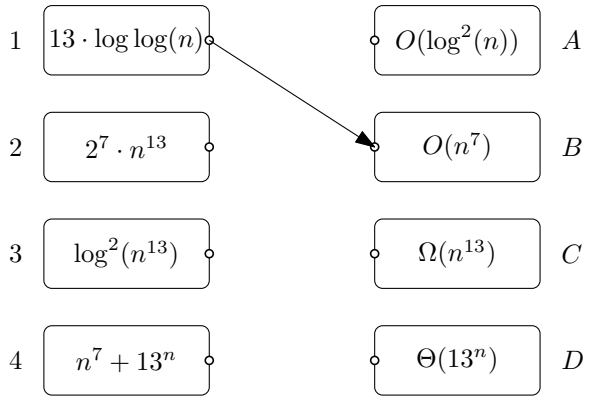
- Bringe den Aufkleber mit deinem Namen und deiner Matrikelnummer oben auf diesem Deckblatt an und beschrifte jedes Aufgabenblatt mit deiner Matrikelnummer.
- Diese Klausur umfasst 13 Seiten (diese Titelseite eingeschlossen) und 6 Aufgaben. Es können maximal 60 Punkte erreicht werden.
- Schreibe die 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	

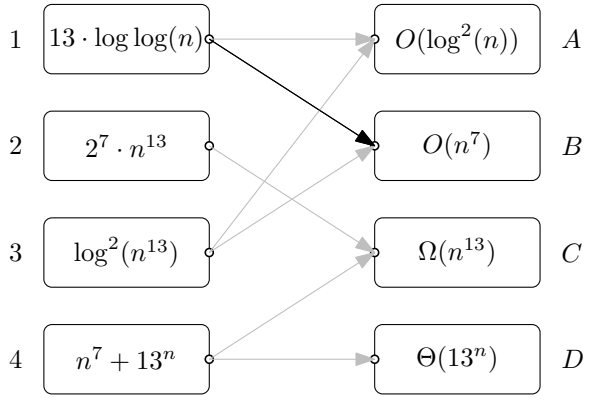
1. **Kleinaufgaben** [10 Punkte]

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 $(1, B)$, dass $13 \cdot \log\log(n) \in O(n^7)$ 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:



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

(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)

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

Lösung:

$$f_1(n) \in \Theta(\log(n))$$
$$f_2(n) \in \Theta(n)$$
$$f_3(n) \in \Theta(1)$$

(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. Für ein E-Book soll ein digitaler Index angelegt werden, der für jedes Wort angibt, auf welchen Seiten es vorkommt.

   Lösung: Hashmap. Einfügen, Suchen.

2. Bei einem Wettessen wird für alle Teilnehmer:innen nachvollzogen, wie viele Bananen sie noch essen müssen, um den Haufen zu leeren. Dabei ist es üblich, dass starke Teilnehmer:innen mehrere Bananen auf einmal verschlingen. Den Zuschauer:innen soll stets die Person angezeigt werden, die die wenigsten Bananen übrig aber noch nicht alles aufgegessen hat. Aufgeben ist keine Option.

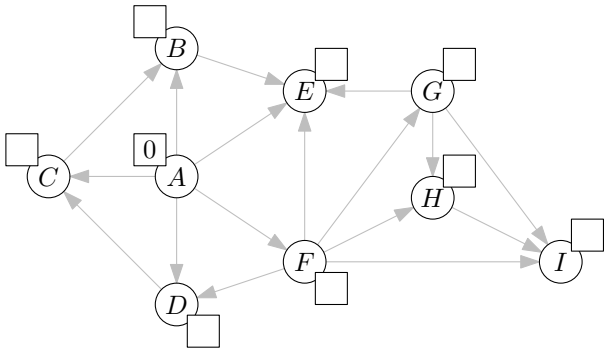
   Lösung: binärer Heap. `push`, `popMin`, `decPrio`.

3. Auf einer Website kann ein Buch vorbestellt werden, dessen verfügbare Kapazität erst zum Verkaufsstart bekannt sein wird. Nutzer:innen können das Buch vor Verkaufsstart vorbestellen. Eine Stornierung ist nicht möglich. Die Exemplare werden dann in der Reihenfolge vergeben, in der sich die Nutzer:innen gemeldet haben, solange der Vorrat reicht.

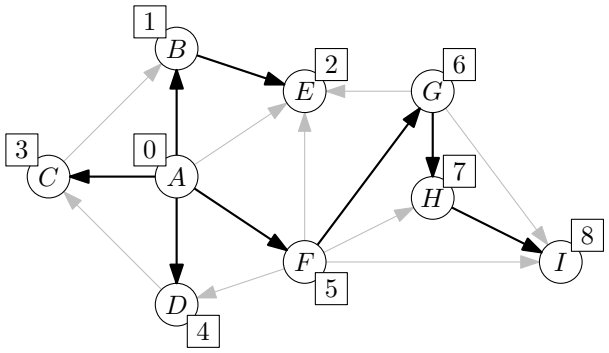
   Lösung: Liste. `popFirst`, `pushBack`

2. **Tiefensuche** [10 Punkte]

(a) Gegeben sei folgender, gerichteter Graph G. Gib den DFS-Baum einer Tiefensuche an die bei A startet, indem du in der folgenden Abbildung die Kanten des Baums markierst. Gib außerdem die Reihenfolge an, in der die Knoten besucht werden (hierfür kannst du die DFS-Nummern in die Kästchen schreiben). Falls bei einem Knoten mehrere unbesuchte Nachbarn zur Auswahl stehen, soll unter denen immer der alphabetisch kleinste Nachbar besucht werden. (3 Punkte)



Lösung:

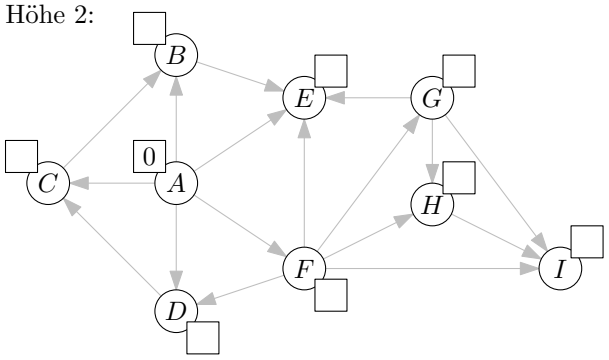


Reihenfolge: $A, B, E, C, D, F, G, H, I$

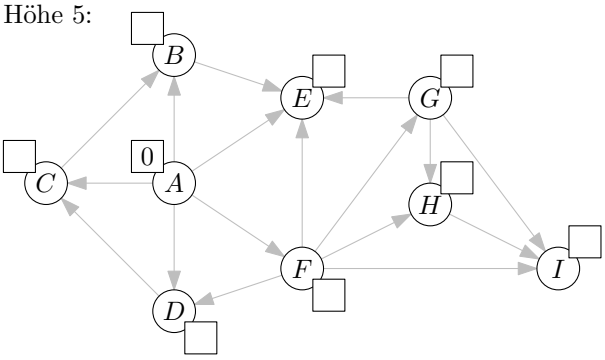
(b) In Teilaufgabe (a) wurden bei der Tiefensuche unbesuchte Nachbarn in alphabetischer Reihenfolge abgearbeitet. Die Wahl einer anderen Reihenfolge kann für den gleichen Graphen mit gleichem Startknoten A einen anderen DFS-Baum ergeben. Gib einen solchen DFS-Baum mit Höhe 2 und einen mit Höhe 5 an, indem du die Kanten der Bäume markierst und jeweils die DFS-Nummern in die Kästchen einträgst. (4 Punkte)

Hinweis: Die Höhe eines Baums ist die größte Anzahl Kanten auf dem Pfad von der Wurzel zu einem Blatt.

Höhe 2:

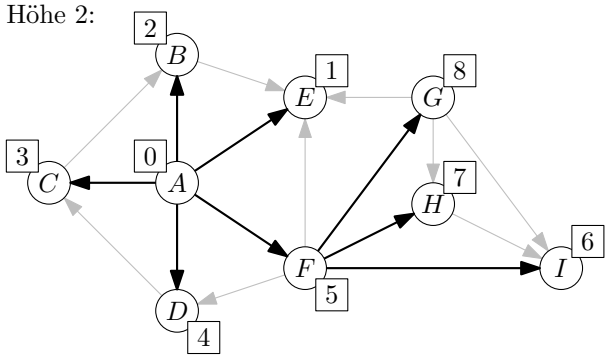


Höhe 5:

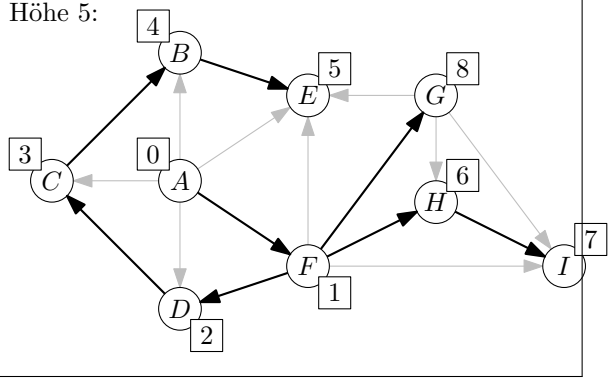


Lösung:

Höhe 2:

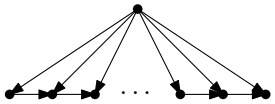


Höhe 5:



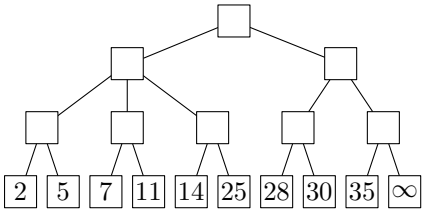
(c) Zeige oder widerlege: Für jedes $n \geq 2$ gibt es einen gerichteten Graphen $G = (V, E)$ mit n Knoten und einem Knoten $w \in V$, sodass es in G sowohl einen DFS-Baum mit Wurzel w und Höhe 1 als auch einen DFS-Baum mit Wurzel w und Höhe $n - 1$ gibt. (3 Punkte)

Lösung: Betrachte einen gerichteten Pfad $v_1, \ldots, v_{n-1}$ und einen weiteren Knoten w der eine Kante zu allen Knoten des Pfads hat. Diese Instanz hat einen DFS-Baum mit Tiefe 1 (wenn zuerst w, dann v_{n-1}, dann v_{n-2} usw. besucht wird) und einen DFS-Bäum mit Tiefe $n - 1$ (wenn zuerst w, dann v_1, dann v_2, usw. besucht wird).



3. **(2, 3)-Bäume** [10 Punkte]

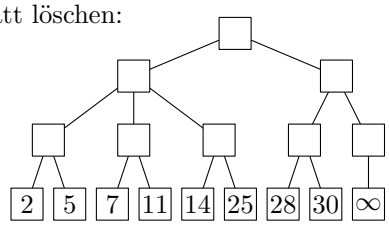
In dieser Aufgabe beschäftigen wir uns mit $(2, 3)$-Bäumen, bei denen der ∞-Trick angewendet wird. Gegeben sei folgender $(2, 3)$-Baum B:



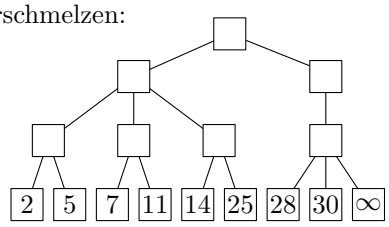
(a) Es soll nun das Element mit Schlüssel 35 aus B gelöscht werden. Gib die Subroutinen an, die durchgeführt werden müssen, um 35 zu Löschen und anschließend die $(2, 3)$-Baum-Eigenschaft wiederherzustellen. Zeichne den Baum nach jeder Anwendung einer Subroutine.
Hinweis: Mögliche Subroutinen sind *Aufspalten*, *Verschmelzen*, *Blatt Löschen*, *Ausbalancieren*. Du brauchst keine Schlüssel in den inneren Knoten einzutragen. (3 Punkte)

Lösung:

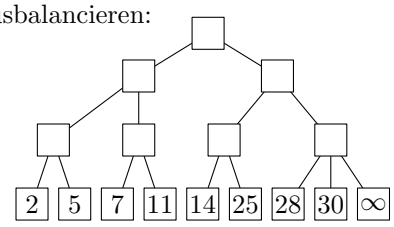
Blatt löschen:



Verschmelzen:



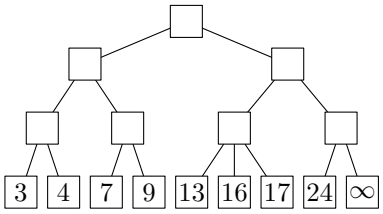
Ausbalancieren:



Im Folgenden beschäftigen wir uns mit einem Algorithmus, der in einem $(2, 3)$-Baum, für einen gegebenen Schlüssel k, alle Elemente löscht, deren Schlüssel echt kleiner als k sind.

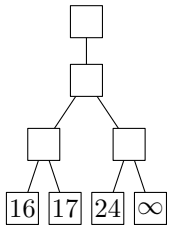
Der Algorithmus geht in drei Schritten vor. In Schritt 1 wird das kleinste Element e mit Schlüssel größer oder gleich k bestimmt. In Schritt 2 werden alle Teilbäume gelöscht, die links des Pfades von e zur Wurzel liegen. Im letzten Schritt wird für den resultieren Baum die $(2, 3)$-Baum-Eigenschaft repariert.

(b) Wir wollen nun mit Hilfe des oben beschriebenen Algorithmus alle Elemente echt kleiner als 16 aus folgendem $(2, 3)$-Baum B entfernen. (3 Punkte)



Zeichne B nach dem *zweiten* Schritt des Algorithmus.

Lösung:



(c) Der oben beschriebene Algorithmus soll nun in der Methode `removeAllSmaller`(`root`: NODE, `key`: KEY) umgesetzt werden. Sie erhält als Eingabe die Wurzel `root` eines $(2, 3)$-Baums und einen Schlüssel `key` und führt die oben genannten drei Schritte aus, um alle Elemente mit Schlüssel echt kleiner `key` zu löschen.

Dazu stehen folgende Subroutinen zur Verfügung:

- `find`(`key`: KEY) – Gibt das kleinste Element mit Schlüssel größer oder gleich `key` zurück.
- `removeSmallerSiblings`(`node`: NODE) – Löscht alle Geschwister links von `node`, inklusive ihrer Teilbäume.
- `repairTree`() – Stellt die $(2, 3)$-Baum-Eigenschaft des Baums wieder her.

Gib den Pseudocode für `removeAllSmaller`(`root`: NODE, `key`: KEY) an. (4 Punkte)

Hinweis: Benutze Kommentare, um zu markieren, zu welchen der drei Schritte die Zeilen gehören.

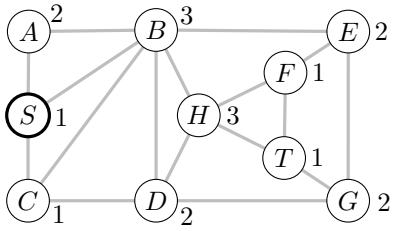
`removeAllSmaller`(`root`: NODE, `key`: KEY):

Lösung:

```
e = find(key)                                   ▷ Schritt 1
while e ≠ root do                               ▷ Schritt 2
    removeSmallerSiblings(e)
    e = e.parent
repairTree()                                    ▷ Schritt 3
```

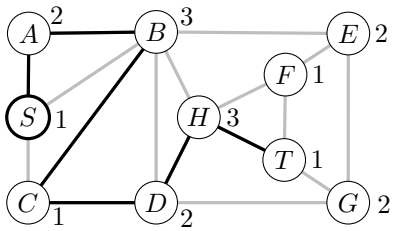
4. **123-Pfade** [10 Punkte]

Sei $G = (V, E)$ ein Graph und $\ell : V \to \{1, 2, 3\}$ eine Funktion, die jedem Knoten ein Label aus der Menge $\{1, 2, 3\}$ zuordnet. Wir nennen einen Pfad $\pi = \langle v_0, v_1, \ldots, v_k \rangle$ einen 123-Pfad, wenn für alle $i \in \{1, \ldots, k\}$ gilt, dass $(\ell(v_{i-1}) + 1) \bmod 3 = \ell(v_i) \bmod 3$. In folgendem Graphen stehen die Label eines Knotens rechts vom Knoten. Hier ist beispielsweise $\langle F, E, B, C \rangle$ ein 123-Pfad, aber $\langle F, E, B, D \rangle$ nicht.



(a) Zeichne in der Abbildung den kürzesten 123-Pfad von S nach T ein. (3 Punkte)

Lösung:



Kürzester Pfad: S, A, B, C, D, H, T

(b) Beschreibe einen möglichst effizienten Algorithmus in Worten, der als Eingabe einen Graphen $G = (V, E)$ und $\ell : V \to \{1, 2, 3\}$, sowie zwei Knoten s und t erhält und den kürzesten 123-Pfad von s nach t ausgibt, oder andernfalls ausgibt, dass ein solcher Pfad nicht existiert.
Begründe, wieso der Algorithmus korrekt ist, und nenne und begründe sein asymptotisches Laufzeitverhalten. (7 Punkte)

Hinweis: Pseudocode wird mit 0 Punkten bewertet.

Lösung: Um einen kürzesten 123-Pfad zu finden, wird eine angepasste Breitensuche verwendet, bei der von einem Knoten v aus nur Kanten zu Knoten w betrachten werden für die $\ell(v)+1 = \ell(w) \bmod 3$ gilt.
Hierdurch wird sichergestellt, dass alle Pfade, die von der Breitensuche gefunden werden 123-Pfade sind. Die Anpassung verändert die asymptotische Laufzeit der Breitensuche nicht. Die Laufzeit liegt somit in $O(|V| + |E|)$.

5. **Dynamisch Zerlegen** [10 Punkte]

Gegeben sei eine Schnur der Länge $S \in \mathbb{N}$. Die Schnur soll in mehrere Stücke zerlegt werden, mit der Absicht, diese möglichst gewinnbringend zu verkaufen. Der Wert eines Stücks Schnur hängt von der Länge ab und es können nur Stücke in ganzzahliger Länge verkauft werden. Der Wert ist durch die Funktion $w\colon \mathbb{N} \to \mathbb{N}$ gegeben. Ein Stück Schnur der Länge k hat also den Wert $w(k)$. Wir nehmen an, dass $w(0) = 0$.

So kann beispielsweise eine Schnur der Länge $S = 3$ mit Werten $w(1) = 2$, $w(2) = 5$ und $w(3) = 4$ in Stücke der Länge 1 und 2 zerlegt werden, welche einen Gewinn von $2 + 5 = 7$ bringen.

(a) Gegeben sei eine Schnur der Länge 4. Außerdem haben Schnüre der Länge 1, 2, 3 und 4 jeweils den Wert 3, 7, 12 und 13. Das heißt $S = 4$ und $w(1) = 3$, $w(2) = 7$, $w(3) = 12$ und $w(4) = 13$. Gib eine Zerlegung an, die den Gesamtwert maximiert. (2 Punkte)

Lösung: $1, 3$ hat den Wert 15.

Es soll nun ein dynamisches Programm über die Länge S der Schnur entwickelt werden, welches den Wert der wertvollsten Zerlegung bestimmt. Dazu wird ein Array T angelegt, um die Teillösungen zu verwalten.

Hinweis: Bedenke, dass nur der *Wert* der wertvollsten Zerlegung gesucht wird und *nicht* die Zerlegung selbst.

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

Lösung: $T[i]$ enthält den maximalen Wert der durch Zerlegung einer Schnur der Länge i erzielt werden kann.

$$T[0] = 0$$
$$T[i] = \max_{j \in [i]} \{w(j) + T[i-j]\} \text{ für } i \in \{1, \ldots, S\}$$

(c) Gib an, wie man aus dem Array T schließlich den Wert der wertvollsten Zerlegung von S bestimmen kann. (2 Punkte)

Lösung: Die Gesamtlösung ist $T[S]$.

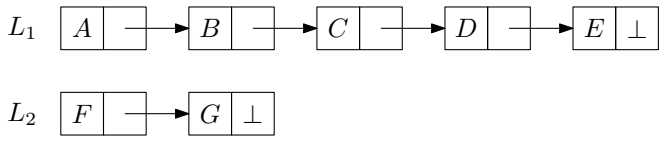
6. **Amortisierende Listen** [10 Punkte]

Wir betrachten eine Datenstruktur, die Elemente in einer Folge von Listen $L_1, L_2, L_3, \ldots$ verwaltet. Eine Liste L_i ist *voll*, wenn sie k Elemente enthält und *leer*, wenn sie keine Elemente enthält. Die Datenstruktur bietet zwei Operationen:

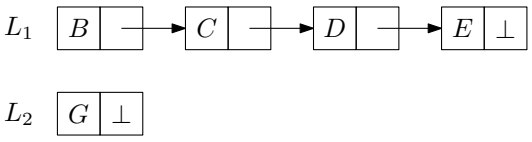
- `add`: Fügt ein Element am Ende der ersten nicht-vollen Liste ein. Falls alle Listen voll sind, wird der Folge eine weitere Liste mit dem neuen Element hinzugefügt. Dafür wird eine Laufzeit von $\Theta(1)$ benötigt.
- `remove`: Löscht das erste Element aus jeder nicht-leeren Liste. Dafür wird eine Laufzeit von $\Theta(\ell)$ benötigt, wobei ℓ die Anzahl nicht-leerer Listen ist.

Folgende Abbildung zeigt für $k = 5$ den Zustand der Datenstruktur nach `add`(A), `add`(B), `add`(C), `add`(D), `add`(E), `add`(F), und `add`(G) (*Zustand 1*) und einem darauffolgenden `remove` (*Zustand 2*).

Zustand 1



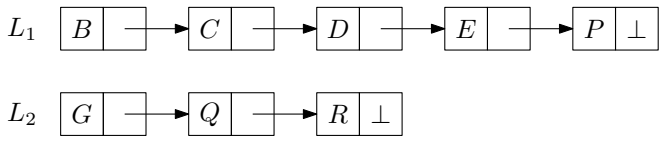
Zustand 2



(a) Gib die beiden weiteren Zustände an, die man erhält, wenn man ausgehend von *Zustand 2* erst `add`(P), `add`(Q), und `add`(R) ausführt (*Zustand 3*) und anschließend ein `remove` ausführt (*Zustand 4*).(2 Punkte)
Hinweis: Bedenke, dass eine Liste hier nur genau dann voll ist, wenn sie 5 Elemente enthält.

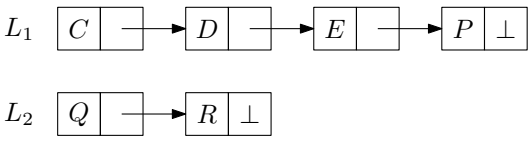
Zustand 3

Lösung:



Zustand 4

Lösung:



(b) Im Folgenden gehen wir von einem Zustand aus, in dem die Datenstruktur keine Elemente enthält. Gib für jedes $n \in \mathbb{N}$ eine Sequenz von $\Theta(n)$ Operationen an, sodass es eine `remove` Operation gibt, die Laufzeit $\Theta(n)$ hat. Deine Konstruktion soll für jede Konstante $k \in \mathbb{N}$ funktionieren. Begründe deine Antwort.(3 Punkte)

Lösung: Nach $n \cdot k$ mal `add` hat eine `remove` Operation eine Laufzeit von $\Theta(n)$, weil es dann n nicht-leere Listen gibt.

(c) Zeige, dass in jeder beliebigen Abfolge von `add` und `remove` Operationen jede Operation amortisiert konstante Kosten hat. (5 Punkte)

Lösung: Bsp. Charging:

Die Operation `add` hat konstante tatsächliche Kosten. Seien $e_1, \ldots, e_\ell$ die ersten Elemente aus allen nicht-vollen Listen. Die Operation `remove` hat dann Kosten ℓ. Charge Kosten ℓ auf `add`(e_j) für jedes $j \in [\ell]$. Damit sind wir ℓ Kosten losgeworden. Folglich hat jedes `add` kosten von $1+1 = 2$ und jedes `remove` Kosten 0. Beide Kosten sind in $O(1)$.