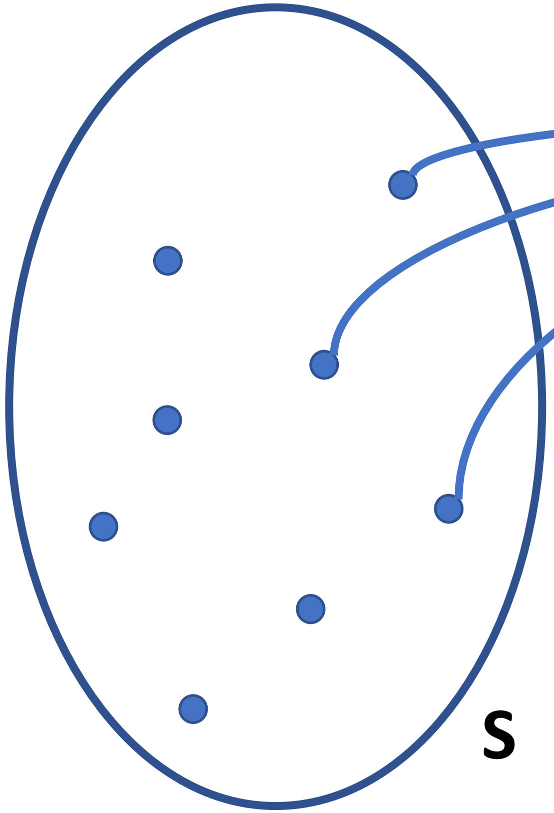
Tutorium
Grundlagen: Algorithmen und Datenstrukturen

Übungsblatt Woche 4

Aufgabe 4.2 – Wiederholung Bankkonto Methode



Menge an Operationen

Folge	σ_1	σ_2	σ_3
Laufzeit	$T(\sigma_1)$	$T(\sigma_2)$	$T(\sigma_3)$
Tokens	$\Delta(\sigma_1)$	$\Delta(\sigma_2)$	$\Delta(\sigma_3)$
am. Laufzeit	$A(\sigma_1)$	$A(\sigma_2)$	$A(\sigma_3)$

$$A(\sigma_i) = T(\sigma_i) + \Delta(\sigma_i)$$

Gesucht ist Δ, wobei:

1) **Summe aller Tokens darf nie negativ sein**
2) **Δ muss möglichst gut gewählt sein**

Aufgabe 4.2

Zu Bedingung 1:

$$A(\sigma_1, \ldots, \sigma_m) = \sum_{i=1}^{m} A(\sigma_i) = \sum_{i=1}^{m} (T(\sigma_i) + \Delta(\sigma_i))$$
$$= T(\sigma_1, \ldots, \sigma_m) + \underbrace{\sum_{i=1}^{m} \Delta(\sigma_i)}_{\geq 0} \geq T(\sigma_1, \ldots, \sigma_m)$$

- Die Amortisierte Laufzeit ist eine obere Schranke für die tatsächliche Laufzeit.
- Die Tokens geben an wieviel die amortisierte Laufzeit die tatsächliche Laufzeit übersteigt (obere Schranke T)

Zu Bedingung 2:

Ziel:
Amortisierte Laufzeit von Operationsfolgen soll möglichst gering sein

Ansatz:
Wähle Δ, sodass die größte am. Laufzeit einer Einzeloperation minimal wird

Erreicht:
Obere Schranke O(m*max(A(σ))) für Worst-Case
(m Anzahl Operationen)

Aufgabe 4.3

Aufgabe 4.3 (P) Amortisation

Wir betrachten eine Folge von $m \geq 1$ Inkrement-Operationen iterativ angewandt auf 0, das heißt, dass die k-te Inkrementoperation σ_k die Zahl $k-1$ zur Zahl k inkrementiert. Wir nehmen an, dass alle Zahlen in *Dezimalschreibweise* geschrieben werden, also mit den Ziffern $0, 1, \ldots, 9$. Außerdem nehmen wir (vereinfachend) an, dass die Änderung einer Stelle (von x zu $x+1$ für alle $x \in \{0, \ldots, 8\}$ bzw. von 9 zu 0) Laufzeit 1 hat und die Laufzeit jeder Inkrement-Operation gerade die Anzahl der Stellen ist, die durch die Operation geändert werden.

(a) Definieren Sie ein zulässiges Amortisationsschema $\Delta : \{\sigma_1, \sigma_2, \ldots\} \to \mathbb{R}$ der Bankkonto-Methode, sodass für jede Inkrementoperation σ_k die amortisierte Laufzeit $A(\sigma_k) = T(\sigma_k) + \Delta(\sigma_k)$ gerade $\frac{10}{9}$ beträgt (hierbei sei $T(\sigma_k)$ die tatsächliche Laufzeit von σ_k).

(b) Zeigen Sie, dass Ihr Amortisationsschema zulässig ist, indem Sie nachweisen, dass das Tokenkonto stets nichtnegativ ist.

Hinweis: Verwenden Sie in Ihrem Amortisationsschema für jede Operation σ_k die Anzahl $S_{x \to x+1}(\sigma_k)$ der Stellen, die durch σ_k von x zu $x+1$ geändert werden, sowie die Anzahl $S_{9 \to 0}(\sigma_k)$ der Stellen, die durch σ_k von 9 zu 0 geändert werden.

Aufgabe 4.3

(a) Definieren Sie ein zulässiges Amortisationsschema $\Delta : \{\sigma_1, \sigma_2, \ldots\} \to \mathbb{R}$ der Bankkonto-Methode, sodass für jede Inkrementoperation σ_k die amortisierte Laufzeit $A(\sigma_k) = T(\sigma_k) + \Delta(\sigma_k)$ gerade $\frac{10}{9}$ beträgt (hierbei sei $T(\sigma_k)$ die tatsächliche Laufzeit von σ_k).

$$\boxed{\begin{matrix} S_{x \to x+1}(\sigma_k) \\ S_{9 \to 0}(\sigma_k) \end{matrix}}$$

$A(\sigma_k) = \frac{10}{9}$

$A(\sigma_k) = T(\sigma_k) + \underline{\Delta(\sigma_k)}$

$T(\sigma_k) = S_{x \to x+1}(\sigma_k) + S_{9 \to 0}(\sigma_k)$

$\Longrightarrow \frac{10}{9} = S_{x \to x+1}(\sigma_k) + S_{9 \to 0}(\sigma_k) + \Delta(\sigma_k)$

$\Delta(\sigma_k) = \frac{10}{9} - \underbrace{S_{x \to x+1}(\sigma_k)}_{} - S_{9 \to 0}(\sigma_k)$

$= \frac{10}{9} - 1 - S_{9 \to 0}(\sigma_k) =$

$= \frac{1}{9} - S_{9 \to 0}(\sigma_k)$

(b) Zeigen Sie, dass Ihr Amortisationsschema zulässig ist, indem Sie nachweisen, dass das Tokenkonto stets nichtnegativ ist.

$1 \to \frac{1}{9}$

$2 \to \frac{2}{9}$

$\vdots$

$9 \to 1$

$10 \to \frac{1}{9}$

$11 \to \frac{2}{9}$

$\vdots$

$20 \to \frac{2}{9}$

Aufgabe 4.1 - Programmieraufgabe

Sie sehen das Grundgerüst für eine sich selbst-organisierende, dynamische Liste, wie sie in der Vorlesung angesprochen wurde. Diese wollen wir im Folgenden erweitern. Beachten Sie die zusätzlichen Laufzeit- und Speicher-Beschränkungen. Sie können vereinfachend davon ausgehen, dass nur ein Thread auf die Liste zugreift.

a) Implementieren Sie Methode `void add(...)`, die einen neuen Knoten erstellen soll und an das Ende der Liste anhängt. Die Laufzeit der Funktion soll in $\mathcal{O}(1)$ sein. Sie dürfen hierfür die Klasse `SelfOrganizingList` erweitern.

b) Implementieren Sie die Methode `Optional<T> findFirst(Predicate<T> p)`. Für den ersten Knoten `n`, für den das Prädikat zu `true` evaluiert (`p.test(n.data)`), soll dessen Wert zurückgeben werden (`Optional.of(n.data)`). Außerdem soll dieser Knoten an den Anfang der Liste verschoben werden. Sollte kein Knoten dem Prädikat genügen, so soll `Optional.empty()` zurückgegeben werden und die Liste nicht verändert werden.

c) Implementieren Sie die Methode `void removeDuplicates()`, die *gleiche* Elemente entfernt und nur das erste dieser Elemente in der Liste belässt. Ihre Funktion soll $\mathcal{O}(1)$ Speicher verbrauchen.

Aufgabe 4.4

Entwerfen Sie im Folgenden zwei einfache Funktionen und erarbeiten Sie korrekten `java`-Code. Bestimmen Sie dann die asymptotische Laufzeit Ihres Algorithmus. Sie dürfen in ihren Algorithmen ausschließlich Schleifen, If-Statements, Additionen, Subtraktion sowie Multiplikation verwenden.

a) Entwerfen Sie eine Funktion, die für zwei `integer` a und b die ganzzahlige Division berechnet. Sie können davon ausgehen, dass $a > 0$ und $b > 0$ gilt.

b) Entwerfen Sie eine Funktion, die für zwei `integer` a und b den Rest berechnet, also a `mod` b. Sie können davon ausgehen, dass $a > 0$ und $b \neq 0$ gilt.