Übungsblatt 3

Abgabe via Moodle.
Deadline Fr. 26ter Mai

Aufgabe 1 (*Amortisierte Analyse*, $2+4+2$)

Auf einer Datenstruktur wird eine Sequenz $\sigma = (\sigma_1, \cdots, \sigma_n)$ von Operationen ausgeführt. Die Operation σ_i kostet i, wenn i eine Potenz von vier ist, sonst 1.

1. Berechnen Sie die Kosten für 256 Operationen $T(256)$.
2. Geben Sie eine geschlossene Form $T(n)$ für die Kosten von n Operationen an. Nehmen Sie dabei vereinfachend an, dass $n = 4^m$ für ein $m \in \mathbb{N}$ ist. **Tipp:** Die Formel für endliche Partialsummen einer geometrischen Reihe könnten hilfreich sein.
3. Wie groß sind dann die amortisierten Kosten pro Operation? Schätzen Sie die amortisierten Kosten auch in $\mathcal{O}$-Notation ab.

Lösung:

1. Auflisten der Operationen und deren Kosten ergibt folgende Tabelle:

Op. No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...
cost(i)	1	1	1	4	1	1	1	1	1	1	1	1	1	1	1	16	1	...

   Aufsummieren der Kosten liefert $T(256) = 256 - 4 + 4 + 16 + 64 + 256 = 592$.

2. Wir zeigen $T(n) = \frac{7}{3}n - \log_4 n - \frac{4}{3}$ auf zwei verschiedenen Wegen, die beide darlegen wie man auf diese Formel kommt. Es ist zunächst $S(m) := T(4^m) = T(n) = \sum_{i=1}^{4^m} \text{cost}(i)$

   - Es ist $n = 4^m$. Nun stellt man $S(m)$ als Rekurrenz dar, um nachher eine schöne Summenformel zu bekommen. Das überlegt man sich folgendermaßen: Beim Übergang von $m-1$ nach m haben alle Zahlen zwischen 4^{m-1} und 4^m Kosten $cost(i) = 1$. Das sind genau $4^m - 4^{m-1} - 1$ viele. Die einzige Operation die höhere Kosten hat ist die Letzte. Sie hat Kosten $\text{cost}(4^m) = 4^m$. Also ist $S(m) = S(m-1) + 4^m + (4^m - 4^{m-1} - 1) = S(m-1) + 7 \cdot 4^{m-1} - 1$. Ausrollen der Rekurrenz (für $m > 1$) ergibt: $S(m) = S(1) + \sum_{i=1}^{m-1}(7 \cdot 4^i - 1) = 7 + 7 \cdot \sum_{i=0}^{m-1} 4^i - (m-1) - 7 = 7 \cdot \frac{4^m - 1}{3} - m + 1$. Und da $m = \log_4 n$ liefert uns Rücksubstituieren eine Formel für T: $T(n) = \frac{7}{3}n - \log_4 n - \frac{4}{3}$.
   - Die zweite Möglichkeit diese Formel zu zeigen ist die Folgende: Im Wesentlichen teilt man die Kosten für $S(m) = S_1(m) + S_2(m)$ in zwei Hälften. $S_1(m)$ seien dabei gerade die Kosten für die Operationen die jeweils Kosten 1 haben und $S_2(m)$ für die Operationen mit Kosten > 1. Unter den 4^m Operationen gibt es genau m, die Kosten > 1 haben (nämlich genau die vierer Potenzen, 1 ausgenommen). Also ergibt sich $S_1(m) = 4^m - m$. Die Operationen mit Kosten >1 sind genau die vierer Potenzen. Also $S_2(m) = \sum_{i=1}^{m} 4^i = -1 + \sum_{i=0}^{m} 4^i = \frac{4^{m+1}-1}{3} - 1$. Also ist $S(m) = 4^m - m + \frac{1}{3} \cdot 4^{m+1} - \frac{4}{3} = (1 + \frac{4}{3})4^m - m - \frac{4}{3} = \frac{7}{3} \cdot 4^m - m - \frac{4}{3}$ und damit $T(n) = \frac{7}{3}n - \log_4 n - \frac{4}{3}$.

3. Als amortisierte Kosten pro Operation ergibt sich $\frac{T(n)}{n} = \frac{7}{3}n - \log_4 n/n \leq \frac{7}{3}$. Das liefert $T(n)/n = \text{O}(1)$, also konstante amortisierte Kosten pro Operation.

Aufgabe 2 (*Stack, Queue und Deque mittels einfach verketteter Liste,* $3+3+2$ *Punkte*)

1. Wie baut man einen unbeschränkten Stack mittels einer *einfach* verketteten Liste? Geben Sie hierzu "Implementierungen" von *pushFront* und *popFront* in Pseudocode an, so dass die Operationen jeweils nur konstante Zeit benötigen. Dabei darf *keine* bereits fertig "'implementierte"' Datenstruktur verwendet werden, d.h. Sie müssen alles selbst bauen.
2. Wie baut man eine unbeschränkte Queue mittels einer *einfach* verketteten Liste? Erweitern Sie hierzu Ihre "'Implementierung"' aus Teilaufgabe a) um eine Operation *pushBack* und geben Sie diese in Pseudocode an, so dass die Operation nur konstante Zeit benötigt.
3. Eine *Double-Ended Queue* (Deque) hat die Operationen *pushFront*, *pushBack*, *popFront* und *popBack*. Lässt sich auch eine Deque mit einer einfach verketteten Liste so konstruieren, dass die Operationen *pushFront*, *pushBack*, *popFront* und *popBack* alle nur konstante Zeit benötigen? Oder braucht man z.B. eine doppelt verkettete Liste? Begründen Sie kurz!

Lösung:

1. Bitte beachten Sie, dass bei dieser Lösung die einfach verkettete Liste stets einen Kreis bildet. Das wird schon beim erschaffen eines neuen Listen Items sichergestellt, indem *next* immer auf das Item selbst zeigt. Außerdem gibt es ein Kopf-Item. D.h. die leere Liste wird durch ein Item dargestellt, bei dem *next* auf das Item selbst verweist.

   So werden die Einzelteile der einfach verketteten Liste dargestellt:

```
1: class SinglyLinkedListItem of Element
2:     next := this : Handle      // WICHTIG: bei neuen Items zeigt next auf das Item selbst
3:     e : Element
4: end class
```

   Der Stack selbst:

```
1: class Stack of Element
2:     invariant Die Liste mit Kopf-Item data bildet einen Kreis oder eine Schleife.
3:     data := allocate SinglyLinkedListItem of Element : Handle
4:     procedure pushFront(e : Element)
5:         old_first := data.next : Handle
6:         data.next := allocate SinglyLinkedListItem
7:         data.next.e := e
8:         data.next.next := old_first              // Liste wieder ein Kreis oder eine Schleife
9:     end procedure
10:    procedure popFront
11:        assert data.next ≠ data                  // popFront sinnlos bei leerer Queue
12:        old_first := data.next : Handle
13:        data.next := data.next.next              // Liste wieder ein Kreis oder eine Schleife
14:        dispose old_first
15:    end procedure
16: end class
```

2. Hier muss die Klasse aus Teilaufgabe a) lediglich *erweitert* werden. Wir schreiben das mit Hilfe einer Art Vererbung auf, wie vom objektorientierten Programmieren bekannt:

```
1: class Queue of Element is a Stack of Element
2:     procedure pushBack(e : Element)
3:         pushFront(⊥)
4:         data.e = e
5:         data := data.next
6:     end procedure
```

```
7: end class
```

3. In a) und b) wurde schon die Operationen *pushFront*, *pushBack* und *popFront* in konstanter Zeit realisiert. Die Operation *popBack* aber kann man mit einer einfach verketteten Liste nicht ohne weiteres in konstanter Zeit ausführen. Das liegt daran, dass man vom "'Einstiegspunkt"' in die Liste aus gesehen keinen Handle zum anderen Ende hat. Somit kann auf das andere Ende nicht in konstanter Zeit zugegriffen werden. Dort müssen beim entfernen des letzten Elementes aber die Handles richtig gesetzt werden. Natürlich kann man sich einen bzw. eine feste Anzahl Handles auf das letzte Element bzw. die letzten Elemente merken und diese Handles dann in *popBack* verwenden. Man kann aber immer ein *popBack* mehr ausführen als man Handles hat, was diesen "'Trick"' aushebelt. Mit einer doppelt verketteten Liste wäre das kein Problem.

Aufgabe 3 (*Queue mittels Arrays,* $3+3$ *Punkte*)

Wie baut man eine unbeschränkte Queue mit Hilfe von (unbeschränkten) Arrays?

1. Geben Sie hierzu "'Implementierungen"' von *pushBack* und *popFront* in Pseudocode an, so dass die Operationen jeweils nur *amortisiert* konstante Zeit benötigen.
2. Zeigen Sie, dass *pushBack* und *popFront*, wie von Ihnen in a) angegeben, tatsächlich das gewünschte Zeitverhalten aufweisen.

Lösung:

1.
```
1: class Queue of Element
2:     capacity := 1 : ℕ>0
3:     data := allocate BoundedFifo(1) of Element
4:     procedure reallocate(capacity' : ℕ>0)
5:         data' := allocate BoundedFifo(capacity') of Element
6:         while ¬data.isEmpty do                    // loop copies all elements of data to data'
7:             data'.pushBack(data.first)
8:             data.popFront
9:         end while
10:        dispose data
11:        data := data'                                          // pointer assignment
12:        capacity := capacity'
13:    end procedure
14:    procedure pushBack(e : Element)
15:        if data.size = capacity then reallocate(2 · capacity)
16:        data.pushBack(e)
17:    end procedure
18:    procedure popFront
19:        assert ¬data.isEmpty
20:        data.popFront
21:        if 4 · data.size ≤ capacity ∧ data.size > 0 then reallocate(2 · data.size)
22:    end procedure
23: end class
```

2. Die amortiesierten Laufzeiten ermitteln wir mit der Bankkontomethode. Die einzige Operation, die mehr als konstante Zeit benötigt, ist *reallocate*: Sie kopiert jeweils *data.size* Elemente. Das kopieren eines Elementes koste jeweils **einen** Token. Um das auszugleichen zahlen wir für jedes *pushBack* **zwei** Token und für jedes *popFront* **einen** Token auf das Konto ein.

Behauptung. Der Kontostand reicht immer aus, die durch *reallocate* anfallenden Kosten zu decken.
Beweis. Beim ersten *reallocate* haben wir mindestens zwei Tokens auf dem Konto, die von dem einen Element in der Queue stammen. Nun muss nur genau ein Element kopiert werden, was genau einen Token kostet. Nach einem Aufruf von *reallocate* hat man *capacity*/2 unbenutzte Plätze in *data*. Der nächste aufruf von *reallocate* findet statt, wenn entweder $data.size = capacitiy$ oder $4 \cdot data.size \leq capacitiy$ gilt. Im ersten Fall wurden seit dem vorherigen Aufruf mindestens *capacity*/2 Elemente eingefügt, die jeweils zwei Tokens mitbringen. Der Kontostand reicht also aus um *capacity* Elemente zu kopieren. Im zweiten Fall wurden seit dem vorherigen Aufruf mindestens *capacity*/4 Elemente entfernt, was jeweils einen Token erbrachte. Das ist genug um das Kopieren von *capacity*/4 Elementen zu Bezahlen.

Aufgabe 4 (*Unbounded Arrays,* 2 *Punkte*)

Gegeben sei ein unbounded Array, dessen Größe halbiert wird, sobald das Array nur noch zur Hälfte gefüllt ist. Sei $n \in \mathbb{N}, n \geq 8$ eine Viererpotenz. Geben Sie eine Folge von n Operationen an, so dass ein derart schlecht implementierter unbounded Array mit diesen Operationen einen Aufwand von $\Theta(n^2)$ verursacht. Nehmen Sie an, dass das Array zu Beginn leer ist.

Lösung:
Man führt zunächst $\frac{n}{2}$ `pushBack` Operationen aus. Der Array enthält dann $\frac{n}{2}$ Elemente. Dann führt man $\frac{n}{4}$ Mal die Operationsfolge `pushBack popBack` aus. Bei jedem `pushBack` wird ein Array der Größe n alloziiert, und $\frac{n}{2}$ Elemente umkopiert. Danach beim `popBack` wird wieder ein kleineres Array der Größe $\frac{n}{2}$ alloziiert und alle verbliebenen $\frac{n}{2}$ Elemente dorthin kopiert.

Für ein Operationspaar `pushBack popBack` müssen also n Kopieroperationen durchgeführt werden. Es werden nur die Kopieroperationen betrachtet, da der Aufwand für alle anderen Operationen insgesamt in $\mathrm{O}(n)$ liegt. Insgesamt werden für n Operationen damit mindestens $\frac{n}{4} \cdot n = \Theta(n^2)$ Kopieroperationen durchgeführt.

Aufgabe P3 (*Merging Arrays, optional*)

Für die praktischen Übungen verwenden wir die Plattform `www.hackerrank.com`. Hier müssen Sie sich registrieren um an den Übungen teilzunehmen. Unter dem Link

`https://www.hackerrank.com/adsi-2023`

finden die praktischen Übungen in der Form eines Programmierwettbewerbs statt.

Die dritte Challenge heißt "Merging Arrays". Ihre Aufgabe ist es zwei gegebene, bereits sortierte Arrays zu einem zusammen zu führen. Das Resultat sollte ebenfalls ein sortiertes Array sein. Dieser Vorgang entspricht dem letzten Schritt des Algorithmus **Merge Sort**, den Sie in der Vorlesung kennen lernen.

Um die Aufgabe zu lösen müssen Sie die Funktion **merge**, sowie die Funktion **array_size** vervollständigen.

Da Sortierung ein wichtiger Bestandteil vieler Problemlösungen ist sollte dieses Subproblem möglichst effizient gelöst werden.

Um die Effizienz ihres Algorithmus zu testen gibt es einen großen Test (Case 3) auf Hackerrank. Wenn Ihr Algorithmus schnell genug ist, wird der Test in der vorgegebenen Zeit von 2 Sekunden meistens durchlaufen. Leider laufen diese Tests nicht immer gleich schnell, weshalb eine mehrmalige Abgabe Ihres Algorithmus hilfreich sein könnte.

Falls dieser Test trotzdem immer aufgrund von Zeitmangel fehlschlägt, gibt es eventuell die Möglichkeit Ihren Algorithmus noch zu verbessern.

Eine genauere Beschreibung, sowie ein Beispiel finden Sie auf HackerRank.