Algo I Blatt 3
Aufgabe A
Musterlösung

Yani Kolev

21. Mai 2018

1 Amortisierte Analyse (3 Punkte)

Auf einer Datenstruktur wird eine Sequenz $\sigma = (\sigma_1, \cdots, \sigma_n)$ von Operationen ausgeführt ($n = 2^m$ für ein $m \in \mathbb{N}$). Die Operation σ_i kostet i, wenn i eine Potenz von zwei ist, sonst 2.

- Berechnen Sie die Kosten für 16 Operationen $T(16)$.
- Geben Sie eine geschlossene Form $T(n)$ für die Kosten von n Operationen an.
- Wie groß sind die amortisierten Kosten pro Operation? Schätzen Sie die amortisierten Kosten auch in $\mathcal{O}$-Notation ab.

2 Lösung

Op	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	1	2	2	4	2	2	2	8	2	2	2	2	2	2	2	16

  Somit ist die gesamte Zeit $T(16) = 53$

- Es ist zunächst $S(m) := T(2^m) = T(n) = \sum_{i=1}^{2^m} cost(i)$ mit $S(0) := 2$, wobei $cost(i)$ die Zeitkosten der i-ten Operation sind.

  Es ist $n = 2^m$. Wir stellen $S(m)$ als Rekurrenz dar, um nachher eine Summenformel zu bekommen. Wir überlegen, dass beim Übergang von $m-1$ auf m alle Operationen zwischen 2^{m-1} und 2^m Kosten $cost(i) = 2$ haben. Das sind genau $2^m - 2^{m-1} - 1$ viele. Die einzige teurere Operation ist die letzte - die hat Konsten $cost(2^m) = 2^m$. Somit ist:

$$S(m) = S(m-1) + 2^m + 2 \cdot (2^m - 2^{m-1} - 1) = S(m-1) + 2^{m+1} - 2$$

Ausrollen der Rekurrenz ergibt:

$$S(m) = \sum_{i=1}^{m} (2^{i+1} - 2) + S(0)$$

Das vereichfacht man zu:

$$S(m) = 2^{m+2} - 2m - 4 + S(0) = 2^{m+2} - 2m - 2$$

Da $m = log_2(n)$, Substituieren liefert:

$$T(n) = 4n - 2log_2(n) - 2$$

- Die amortisierte Kosten pro Operation sind $T(n)/n = \mathcal{O}(1)$.

Karlsruher Institut für Technologie

Algorithmen 1 – Sommersemester 2018
Zuverlässige Softwaresysteme im Kontext der Automobilindustrie
`http://verialg.iti.kit.edu`

Prof. Dr. Carsten Sinz – Markus Iser

Übungsblatt 3

Ausgabe: 09.05.2018 – 15:30
Abgabe: 16.05.2018 – 13:00

A Amortisierte Analyse (3 Punkte)

Auf einer Datenstruktur wird eine Sequenz $\sigma = (\sigma_1, \cdots, \sigma_n)$ von Operationen ausgeführt ($n = 2^m$ für ein $m \in \mathbb{N}$). Die Operation σ_i kostet i, wenn i eine Potenz von zwei ist, sonst 2.

- Berechnen Sie die Kosten für 16 Operationen $T(16)$.
- Geben Sie eine geschlossene Form $T(n)$ für die Kosten von n Operationen an.
- Wie groß sind die amortisierten Kosten pro Operation? Schätzen Sie die amortisierten Kosten auch in $\mathcal{O}$-Notation ab.

B `XOR`-Listen

In der Vorlesung wurden doppelt verkettete Listen L mit zwei Zeigern pro Listenelement `x`, `x.prev` und `x.next`, implementiert. Nehmen Sie an, dass alle Zeigerwerte als k-bit Integer-Zahlen interpretiert werden können. Nehmen Sie weiter für die Aufgabenstellung an, dass jedes Listenelement `x` ein Feld `x.key` hat, nach dem gesucht werden kann.

B.1 Grundlegendes (2 Punkte)

Beweisen Sie für zwei k-Bit-Integer I und J und den bitweisen `XOR`-Operator $\oplus$ die Gültigkeit von:

$$(I \oplus J) \oplus J = I \ .$$

Sie können also zu gegebenem $K = (I \oplus J)$ und I das passende J wiederfinden. Beschreiben Sie, wie eine doppelt verkettete Liste mit nur einem Zeiger `x.np` pro Listenelement `x` damit implementiert werden kann, so dass die Funktionen aus Teilaufgabe B.2 effizient implementiert werden können.

Tipp: Lassen sich die Zeiger auf zwei (wie ausgewählte?) Listenelemente als I und J verwenden? Für Anfang und Ende der `XOR`-Liste bieten sich "Dummy"-Elemente an.

Musterlösung:

Es sei I repräsentiert durch die Bits $[i_0, \ldots, i_{k-1}]$ und J durch $[j_0, \ldots, j_{k-1}]$. Da $\oplus$ jedes Bit einzeln manipuliert, reicht es, die Behauptung für ein beliebiges Bit ℓ zu zeigen. Dazu schauen wir uns die Wertetabelle an:

i_ℓ	j_ℓ	$i_\ell \oplus j_\ell$	$(i_\ell \oplus j_\ell) \oplus j_\ell$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Wie schon in der Aufgabenstellung erklärt, speichert man in jedem Listenelement nur einen Zeiger `x.np`. Diesen wählen wir zu `x.np` = $prev[x] \oplus next[x]$. Dabei bezeichne $prev[x]$ beziehungsweise $next[x]$ jeweils einen Zeiger auf den Vorgänger beziehungsweise Nachfolger von `x`. Weiter speichert man global einen Zeiger auf den Kopf head_L und das Ende der Liste tail_L.

Beachte: das erste Element hat keinen Vorgänger ($\text{prev}[\text{head}_L] = 0$). Damit ist head_L`.np` = $next[\text{head}_L]$. Ebenfalls hat das letzte Element keinen Nachfolger, also tail_L`.np` = $prev[\text{tail}_L]$.

B.2 Pseudocode (2 Punkte)

Implementieren Sie die beiden Funktionen aus der Vorlesung `findNext` in $\mathcal{O}(|L|)$ und `pushFront` in $\mathcal{O}(1)$ für `XOR`-Listen. Implementieren Sie außerdem eine Methode `invert`, welche die `XOR`-Liste in $\mathcal{O}(1)$ spiegelt.

Musterlösung:

```
1 function findNext(L, search_key)
2   prev := 0
3   cur_element := head_L
4   while cur_element ≠ 0 and cur_element→key ≠ search_key do
5     next := cur_element→np ⊕ prev
6     prev := cur_element
7     cur_element := next
8   return cur_element
```

```
1 procedure pushFront(L, element)
2   element→np := head_L
3   if head_L ≠ 0 then
4     head_L→np := head_L→np ⊕ element
5   else tail_L := element
6   head_L := element
```

```
1 procedure invert()
2   swap(head_L, tail_L)
```

C Doppelt-verkettete Listen (3 Punkte)

Sie sollen eine unbeschränkte Queue (Warteschlange) mittels einer doppelt verketteten Liste konstruieren. Geben Sie hierzu den Pseudocode zur Beschreibung eines Listenelements `x` sowie der Operationen `pushBack` und `popFront` an, so dass diese jeweils nur konstante Zeit benötigen. Verwenden Sie nur elementare Operationen, insbesondere *keine* Subroutinen aus der Vorlesung (z.B. `splice`).

Aufgabe C (*Doppelt-verkettete Listen,* 3 Punkte)

Sie sollen eine unbeschränkte Queue (Warteschlange) mittels einer doppelt verketteten Liste konstruieren.
Geben Sie hierzu den Pseudocode zur Beschreibung eines Listenelements *x* sowie von *x.pushBack* und *x.popFront* an, so dass die Operationen jeweils nur konstante Zeit benötigen. Verwenden Sie nur elementare Operationen, insbesondere ***keine*** Subroutinen aus der Vorlesung (z.B. *splice*).

Lösungsvorschlag

Wir verwenden bei folgender Lösung ein *sentinel* Listen-Element, das die doppelt-verkettete Liste zu einem Kreis schließt. Hierdurch werden jegliche Sonderfälle in *pushBack* und *popFront* unnötig.

```
1: class DoublyLinkedListItem of Element
2:     prev, next : Handle
3:     e : Element
4: end class
```

Die Queue selbst:

```
1: class Queue of Element
2:     Invariant Für alle DoublyLinkedListItem e gilt e.next.prev = e = e.prev.next.
3:     sentinel := allocate DoublyLinkedListItem of Element with
4:         sentinel.prev = sentinel.next = sentinel
5:     procedure pushBack (e : Element)
6:         new_back := allocate DoublyLinkedListItem : Handle  // Listen-Item erzeugen
7:         new_back.e := e
8:         old_back := sentinel.prev : Handle              // Item einfügen und Liste wieder
9:         old_back.next := new_back                       // zu einem Kreis schließen
10:        new_back.next := sentinel
11:        sentinel.prev := new_back
12:        new_back.prev := old_back
13:    end procedure
14:    procedure popFront
15:        assert sentinel.next ≠ sentinel              // popFront sinnlos bei leerer Queue
16:        old_front := sentinel.next : Handle
17:        sentinel.next := sentinel.next.next          // Item entfernen und Liste wieder
18:        sentinel.next.prev := sentinel               // zu einem Kreis schließen
19:        dispose old_front
20:    end procedure
21: end class
```