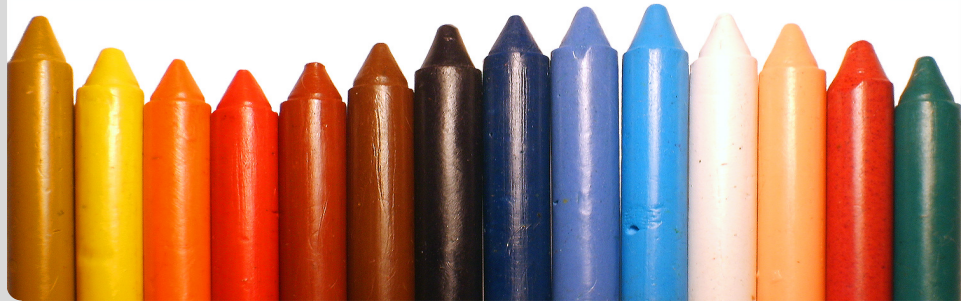


Algorithmen I - Sommersemester 2014

Tutorium Nr. 2

Tobias Hornberger | 8. Mai 2014

INSTITUT FÜR THEORETISCHE INFORMATIK



Rückgabe der Übungsblätter

→ Bei Rückfragen bitte am Ende des Tutoriums vorkommen

Nach dem Email-Desaster:

Kommunikation

- Bitte Email auf dem Blatt eintragen
- Nach dem Tutorium schauen ob meine Email angekommen ist

$$T(n) = \begin{cases} 1, & \text{falls } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n, & \text{falls } n \geq 2 \end{cases}$$

$$n \in 2^m \text{ für } m \in \mathbb{N}_0$$

Vorgehen

- 1 Lösung raten...
- 2 Beweisen, dass die Lösung stimmt

Umformung:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \text{ mit } T(1) = 1$$

$$T(2^m) = 2 * T(2^{m-1}) + 2^m \text{ mit } T(1) = 1$$

$$S(m) = 2 * S(m-1) + 2^m \text{ mit } S(0) = 1$$

Ausrechnen der ersten Werte:

$$S(0) = T(2^0) = T(1) = 1$$

$$S(1) = T(2^1) = T(2) = 2 * 1 + 2 = 4$$

$$S(2) = T(2^2) = T(4) = 2 * (2 * 1 + 2) + 4 = 12$$

$$S(3) = T(2^3) = T(8) = 2 * (2 * (2 * 1 + 2) + 4) + 8 = 32$$

$$S(4) = T(2^4) = T(16) = 2 * (2 * (2 * (2 * 1 + 2) + 4) + 8) + 16 = 70$$

$$S(m) = 2 * S(m - 1) + 2^m \text{ mit } S(0) = 1$$

$$S(0) = 1 = 2^0$$

$$S(1) = 2 * 1 + 2 = 2^1 + 2^1 = 2^2$$

$$S(2) = 2 * (2 * 1 + 2) + 4 = 2^3 + 2^2$$

$$S(3) = 2 * (2 * (2 * 1 + 2) + 4) + 8 = 2^4 + 2 * 2^3$$

$$S(4) = 2 * (2 * (2 * (2 * 1 + 2) + 4) + 8) + 16 = 2^5 + 3 * 2^4$$

Durch scharfes Hinsehen erkennen wir eine mögliche Lösung:

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

Beweis der Lösung durch vollständige Induktion.

IA: $S(0) = T(2^0) = T(1) = 1$

IV: $S(m) = 2^{m+1} + (m-1) * 2^m$

IS: $m \rightarrow m+1$

$$S(m+1) = 2 * S(m) + 2^{m+1} \quad (1)$$

$$= 2 * (2^{m+1} + (m-1) * 2^m) + 2^{m+1} \quad (2)$$

$$= 2^{m+2} + (m-1) * 2^{m+1} + 2^{m+1} \quad (3)$$

$$= 2^{m+2} + m * 2^{m+1} \quad (4)$$

Zurück nach $T(n)$ rechnen für Finale Lösung:

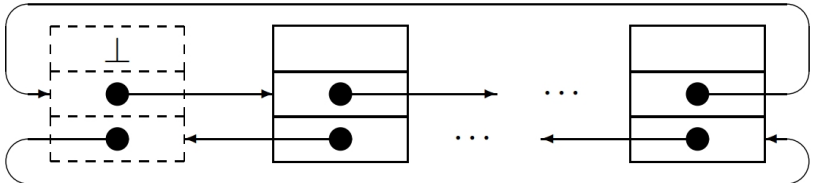
$$T(n) = S(\log_2 n) = 2^{\log_2 n+1} * (\log_2 n - 1) * 2^{\log_2 n}$$

Class Item of Element:

```
e : Element  
next : Pointer to Item  
prev : Pointer to Item
```

Class List of Element:

```
h = Item( $\perp$ ,  
h,  
h)
```



- Einfügen am Anfang
- Einfügen am Ende
- Element finden
- Element entfernen (mit Verweis auf Element)
- Zusammenfügen zweier Listen
- Zugriff auf das k -te Element
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle)
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende
- Element finden
- Element entfernen (mit Verweis auf Element)
- Zusammenfügen zweier Listen
- Zugriff auf das k -te Element
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle)
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende $\mathcal{O}(1)$
- Element finden
- Element entfernen (mit Verweis auf Element)
- Zusammenfügen zweier Listen
- Zugriff auf das k -te Element
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle)
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende $\mathcal{O}(1)$
- Element finden $\mathcal{O}(n)$
- Element entfernen (mit Verweis auf Element)
- Zusammenfügen zweier Listen
- Zugriff auf das k -te Element
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle)
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende $\mathcal{O}(1)$
- Element finden $\mathcal{O}(n)$
- Element entfernen (mit Verweis auf Element) $\mathcal{O}(1)$
- Zusammenfügen zweier Listen
- Zugriff auf das k -te Element
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle)
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende $\mathcal{O}(1)$
- Element finden $\mathcal{O}(n)$
- Element entfernen (mit Verweis auf Element) $\mathcal{O}(1)$
- Zusammenfügen zweier Listen $\mathcal{O}(1)$
- Zugriff auf das k -te Element
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle)
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende $\mathcal{O}(1)$
- Element finden $\mathcal{O}(n)$
- Element entfernen (mit Verweis auf Element) $\mathcal{O}(1)$
- Zusammenfügen zweier Listen $\mathcal{O}(1)$
- Zugriff auf das k -te Element $\mathcal{O}(n)$
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle)
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende $\mathcal{O}(1)$
- Element finden $\mathcal{O}(n)$
- Element entfernen (mit Verweis auf Element) $\mathcal{O}(1)$
- Zusammenfügen zweier Listen $\mathcal{O}(1)$
- Zugriff auf das k -te Element $\mathcal{O}(n)$
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle) $\mathcal{O}(1)$
- Länge der Liste bestimmen

- Einfügen am Anfang $\mathcal{O}(1)$
- Einfügen am Ende $\mathcal{O}(1)$
- Element finden $\mathcal{O}(n)$
- Element entfernen (mit Verweis auf Element) $\mathcal{O}(1)$
- Zusammenfügen zweier Listen $\mathcal{O}(1)$
- Zugriff auf das k -te Element $\mathcal{O}(n)$
- Aufteilen einer Liste in zwei (mit Verweis auf Stelle) $\mathcal{O}(1)$
- Länge der Liste bestimmen $\mathcal{O}(n)$

Implementiert folgende Operationen für eine doppelt verkettete, *sortierte* Liste in Pseudocode:

- ① *insert(e)* (Einfügen des Elements *e* an die richtige Stelle)
- ② *findFirst(e)* (Erstes Vorkommen von *e* in der Liste)
- ③ *remove(e)* (Entfernen des Elements *e* aus der Liste)

- gemittelter Aufwand für eine Sequenz von Operationen im Worst-case
- Kosten einer Operation gering, auch wenn eine einzelne Operation kostspielig ist
- keine Einbeziehung von Wahrscheinlichkeiten
- Beispiel: unbounded Arrays

für jede Operation:

- \hat{c}_i **amortisierte Kosten**
- c_i **tatsächliche Kosten**
- $\hat{c}_i \geq c_i \Rightarrow$ Guthaben einzahlen
- $\hat{c}_i \leq c_i \Rightarrow$ Guthaben abheben
- Konto darf **nie** negativ werden
 - $\forall n \in \mathbb{N} : c + \sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$

- falls ein Feld mit n Elementen durch **reallocate** verkleinert oder vergrößert wird, müssen n Elemente kopiert werden $\Rightarrow n$ Tokens müssen vorhanden sein
- amortisierte Kosten:
 - **push**: 3 Tokens
 - 1 Token: Einfügen des Elements
 - 2 Tokens: Konto
 - **pop**: 2 Tokens
 - 1 Token: Löschen des Elements
 - 1 Token: Konto

- Anfangszustand nach **reallocate**:
 - n Elemente, kein Guthaben vorhanden
- Sequenz von **push**-Operationen:
 - **reallocate** wird nach n **push**-Operationen aufgerufen
 - Guthaben: $2n$ Tokens
 - **reallocate**-Aufruf: $2n$ Elemente werden kopiert \rightarrow genug Tokens
- Sequenz von **pop**-Operationen:
 - **reallocate** wird nach $\frac{n}{2}$ **pop**-Operationen aufgerufen
 - Guthaben: $\frac{n}{2}$ Tokens
 - **reallocate**-Aufruf: $\frac{n}{2}$ Elemente werden kopiert \rightarrow genug Tokens

Entwickelt eine Datenstruktur mit folgenden Eigenschaften:

- *pushBack* und *popBack* in $\mathcal{O}(1)$
- Zugriff auf das k -te Element in $\mathcal{O}(\log n)$

Jeweils für Worst-Case und nicht amortisiert. Speicherallokation ist dabei immer in $\mathcal{O}(1)$.

Jetzt umgekehrt:

- *pushBack* und *popBack* in $\mathcal{O}(\log n)$
- Zugriff auf das k -te Element in $\mathcal{O}(1)$

Jeweils für Worst-Case und nicht amortisiert. Speicherallokation ist dabei immer in $\mathcal{O}(1)$.