Tutorium 3:
Datenstrukturen, Amortisierte Analyse

Matthias Schimek | 3. Juni 2017

TUTORIUM ZUR VORLESUNG ALGORITHMEN I IM SS17

Gliederung

Letztes Blatt

Beweise:

- Beweise von vorne nach hinten aufschreiben
- **Wenn** $\lim_{n \to \infty} A(n) = 0$ **dann gilt** $\forall c > 0 \, \exists n_0 \in \mathbb{N} \, \forall n \geq n_0 : A(n) < c$

Letztes Blatt

Beweise:

- Beweise von vorne nach hinten aufschreiben
- **Wenn** $\lim_{n \to \infty} A(n) = 0$ **dann gilt** $\forall c > 0 \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 : A(n) < c$
- Gilt auch:
- **Wenn** $\lim_{n \to \infty} A(n) = 1$ **dann gilt** $\exists n_0 \in \mathbb{N} \; \forall n \geq n_0 : A(n) \leq 1$?

Letztes Blatt

Karlsruhe Institute of Technology

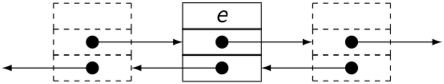
Beweise:

- Beweise von vorne nach hinten aufschreiben
- **Wenn** $\lim_{n\to\infty} A(n) = 0$ **dann gilt** $\forall c > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : A(n) < c$
- **Nein!**
- **Wenn** $\lim_{n\to\infty} A(n) = 1$ **~~dann gilt~~** $\exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : A(n) \leq 1$?

Listen - doppelt verkettet

Class Handle = Pointer to Item

Class Item of Element // one link in a doubly linked list
  e : Element
  next : Handle //
  prev : Handle
  invariant next→prev = prev→next = this



[1]

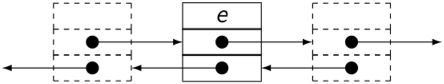
- Problem: Vorgänger erstes Element? Nachfolger von letztem Element?
- Lösung: *dummy header*

[1] Quelle: Vorlesungsfolien, Algo I, KIT

Listen - doppelt verkettet

```
Class Handle = Pointer to Item

Class Item of Element                  // one link in a doubly linked list
    e : Element
    next : Handle         //
    prev : Handle
    invariant next→prev = prev→next = this
```

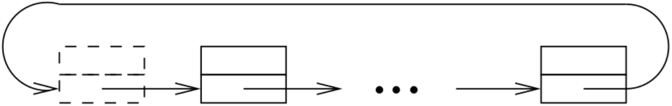


[1]

- Problem: Vorgänger erstes Element? Nachfolger von letztem Element?
- Lösung: *dummy header*

[1]Quelle: Vorlesungsfolien, Algo I, KIT

Listen - einfach verkettet



2

- Nur ein Zeiger
- Dummy Header
- Zeiger auf letztes Element für *pushBack*

[2] Quelle: Vorlesungsfolien, Algo I, KIT

Queue

Queue = Warteschlange

- *FIFO:* First-In-First-Out
- Funktionen: *enqueue*, *dequeue*

Aufgabe: Implementiere *enqueue(a: Item)*, *dequeue()* auf einer doppelt-verketteten Liste

- mittels *splice(a,b,c)*-Funktion
- ohne *splice(a,b,c)*-Funktion

Splice

Procedure splice(a, b, t : Handle) // Cut out $\langle a, \dots, b \rangle$ and insert after t

assert b is not before $a \wedge t \notin \langle a, \dots, b \rangle$

// Cut out $\langle a, \dots, b \rangle$

$a' := a \rightarrow \text{prev}$

$b' := b \rightarrow \text{next}$

$a' \rightarrow \text{next} := b'$ //

$b' \rightarrow \text{prev} := a'$ //

// insert $\langle a, \dots, b \rangle$ after t

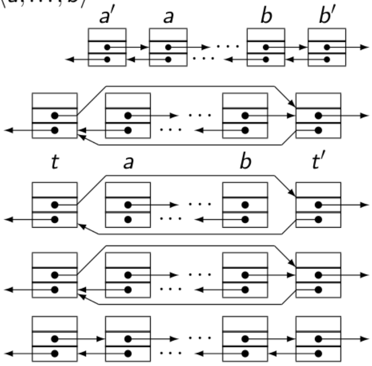
$t' := t \rightarrow \text{next}$ //

$b \rightarrow \text{next} := t'$ //

$a \rightarrow \text{prev} := t$ //

$t \rightarrow \text{next} := a$ //

$t' \rightarrow \text{prev} := b$ //

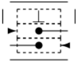


[3]

[3] Quelle: Vorlesungsfolien, Algo I, KIT

Die Klasse 'List'

Class List **of** Element
  // Item h is the predecessor of the first element
  // and the successor of the last element.
  Function head : Handle; **return address of** h
    // Pos. before any proper element
  $h = \begin{pmatrix} \perp \\ \mathit{head} \\ \mathit{head} \end{pmatrix}$: Item   // init to empty sequence



  // Simple access functions
  Function isEmpty : $\{0,1\}$; **return** h.next = head   // $\langle\rangle$?
  Function first : Handle; **assert** $\neg$isEmpty; **return** h.next
  Function last : Handle; **assert** $\neg$isEmpty; **return** h.prev
  $\vdots$
[4]

[4] Quelle: Vorlesungsfolien, Algo I, KIT

Unbounded Array

Karlsruhe Institute of Technology

Unbounded Array = Bounded Array mit *Füllstand*

- w allozierte Größe des Bounded Arrays
- n Anzahl der gespeicherten Elemente im Array
- *Invariante:* $n \leq w < \alpha \cdot n$

$\Rightarrow$ Wenn zu voll $\rightarrow$ umkopieren in größeres Array

$\Rightarrow$ Wenn zu leer $\rightarrow$ umkopieren in kleineres Arraay

- Manchmal kostet *pushBack/popBack* etwas mehr
- *Aber:* Amortisiert konstante Kosten

Unbounded Array

Unbounded Array = Bounded Array mit *Füllstand*

- w allozierte Größe des Bounded Arrays
- n Anzahl der gespeicherten Elemente im Array
- *Invariante:* $n \leq w < \alpha \cdot n$

$\Rightarrow$ Wenn zu voll $\rightarrow$ umkopieren in größeres Array

$\Rightarrow$ Wenn zu leer $\rightarrow$ umkopieren in kleineres Arraay

- Manchmal kostet *pushBack/popBack* etwas mehr
- *Aber:* Amortisiert konstante Kosten

Unbounded Array

Unbounded Array = Bounded Array mit *Füllstand*

- w allozierte Größe des Bounded Arrays
- n Anzahl der gespeicherten Elemente im Array
- *Invariante:* $n \leq w < \alpha \cdot n$

$\Rightarrow$ Wenn zu voll $\rightarrow$ umkopieren in größeres Array

$\Rightarrow$ Wenn zu leer $\rightarrow$ umkopieren in kleineres Arraay

- Manchmal kostet *pushBack/popBack* etwas mehr
- *Aber:* Amortisiert konstante Kosten

Amortisierte Analyse

Warum *amortisierte Analyse*?

- Worst-Case-Abschätzungen zu pessimistisch
- Manchmal: teure Operationen nur in Kombination mit günstigen

Amortisierte Analyse:
Amortisierte Analyse ist eine Methode, um für eine ganze Folge von Operationen eine obere Schranke für die „Kosten“ für ihre Ausführung erhalten. Das Ziel ist dabei eine bessere Schranke zu finden als die Anzahl der Operationen mal schlimmste Kosten einer einzelnen Operation. [5]

[5] Quelle: Thomas Worsch, Skript 'Randomisierte Algorithmen'

Amortisierte Analyse

Warum *amortisierte Analyse*?

- Worst-Case-Abschätzungen zu pessimistisch
- Manchmal: teure Operationen nur in Kombination mit günstigen

Amortisierte Analyse:
Amortisierte Analyse ist eine Methode, um für eine ganze Folge von Operationen eine obere Schranke für die „Kosten" für ihre Ausführung erhalten. Das Ziel ist dabei eine bessere Schranke zu finden als die Anzahl der Operationen mal schlimmste Kosten einer einzelnen Operation. [5]

[5] Quelle: Thomas Worsch, Skript 'Randomisierte Algorithmen'

Amortisierte Anaylse - Kontomethode

- Führe ein *virtuelles* Konto
- Pro Operation kann:
  - auf Konto eingezahlt
  - von Konto abgehohben werden.
- Konto darf nie überzogen werden.
- Unterscheide:
  - Tatsächliche Kosten pro Operation
  - Amortisierte Kosten pro Operation

Kreativaufgabe - Datenstrukturen

Entwickelt eine Datenstruktur, die folgendes kann:

- *pushBack* und *popBack* in $\mathcal{O}(1)$ im Worst-Case (nicht nur amortisiert!).
- Zugriff auf das x-te Element in $\mathcal{O}(\log n)$ im Worst-Case (nicht nur amortisiert!).
    - Achtung: Mit „x-tes Element" ist nicht das Element mit dem Wert x gemeint, sondern das Element an Stelle x innerhalb der Datenstruktur.

Hierbei soll eine Speicherallokation beliebiger Größe nur $\mathcal{O}(1)$ kosten.

Hashtabelle

- Speichere Menge $M \subset$ Universum
- $key(e)$ eindeutig für $e \in M$
- unterstützt folgende Operationen in *mathcalO*(1):
  - $M.insert(e): \; M := M \cup \{e\}$
  - $M.remove(k : key): \; M := M \setminus \{e\}, \; key(e) = k$
  - $M.find(k : key)$: return $e \in M$ with $key(e) = k$; $\perp$ falls $e \notin M$

Hashing - Aufgabe

Gegeben sei ein Array $A = A[1], \dots, A[n]$ mit n Zahlen in beliebiger Reihenfolge.
Für eine gegebenen Zahl x soll ein Paar $(A[i], A[j])$, $1 \leq i, j \leq n$ gefunden werden, für das gilt: $A[i] + A[j] = x$.

1. Gebt eine Lösung für $x = 33$ und $A = (7, 15, 21, 14, 18, 3, 9)$ an.
2. Gebt einen effizienten Algorithmus an, der das Problem in erwarteter Zeit $\mathcal{O}(n)$ löst, und bei Erfolg ein Paar $(A[i], A[j])$ ausgibt, ansonsten NIL.

Hashing - Aufgabe

Gegeben sei ein Array $A = A[1], \ldots, A[n]$ mit n Zahlen in beliebiger Reihenfolge.
Für eine gegebenen Zahl x soll ein Paar $(A[i], A[j])$, $1 \leq i, j \leq n$ gefunden werden, für das gilt: $A[i] + A[j] = x$.

1. Gebt eine Lösung für $x = 33$ und $A = (7, 15, 21, 14, 18, 3, 9)$ an.
2. Gebt einen effizienten Algorithmus an, der das Problem in erwarteter Zeit $\mathcal{O}(n)$ löst, und bei Erfolg ein Paar $(A[i], A[j])$ ausgibt, ansonsten NIL.

Wichtige Grundbegriffe

- Ereignisse
- Wahrscheinlichkeiten
- Gleichverteilung
- Zufallsvariable
- 0/1-Zufallsvariablen
- Erwartungswert
- Linearität des Erwartungswerts
- ...