Karlsruher Institut für Technologie

Tutorium Algorithmen 1

02 · Dynamische Arrays, Amortisierte Analyse und Listen · 29.4.2024
Peter Bohner Tutorium 3

Wiederholung

Karlsruher Institut für Technologie

Wahr oder falsch?

$O(f) \cap \Omega(f) = \Theta(f)$

$O(f) \setminus \omega(f) = \Theta(f)$

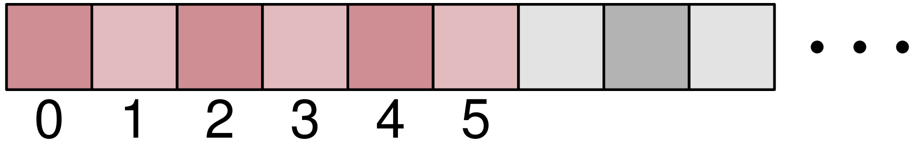
$2^{5n} \in o(2^n)$

$n^4 \in o(n^5)$

$2^n \cdot n^2 \in o(2^n)$

Peter Bohner - peter.bohner@student.kit.edu

Neue Datenstruktur: Dynamische Arrays



Dynamisches Array

Zusammenhängedes Stück Speicher
Durchnummeriert von 0 bis n
Bietet direkten Zugriff auf Elemente

pushBack	amort. $\Theta(1)$
popBack	$\Theta(1)$
get(i: $\mathbb{N}_0$)	$\Theta(1)$
remove(i: $\mathbb{N}_0$)	$\Theta(n)$
find(x)	$\Theta(n)$
remove(x)	$\Theta(n)$
concat	$\Theta(n_1 + n_2)$
splice	$\Theta(n_1 + n_2)$
size	$\Theta(1)$

`https://algo-code.iti.kit.edu/`

Operationen auf Dynamischen Arrays

- **size**()
  - Muss explizit gespeichert werden ($O(1)$)
- **at**(i), **get**(i), **set**(i, x),$[i]$ (Zugriff auf das i-te Element)
  - Adresse lässt sich in $O(1)$ berechnen & zugreifen
- **pushBack**(x)
  - fügt Element am Ende hinzu & Array wächst um ein Element
  - Kapazität muss ggf. vergößert werden
- **popBack**()
  - entfernt & **return**ed letztes Element
  - ggf. Kapazität wieder verringern

Peter Bohner - peter.bohner@student.kit.edu

Amortisierte Analyse

Idee:

- Laufzeiten umverteilen, sodass Operationen *weniger aufwändig* aussehen
- nach **beliebiger** Operationsfolge muss gelten: $\sum$ amortisierte Kosten $\geq \sum$ echte Kosten

Verwendung

- Wenn wir eine Datenstruktur in einem anderen Algorithmus verwenden, dann können wir in der Analyse davon **so tun**, als hätten die Operationen der Datenstruktur tatsächlich ihre amortisierten Kosten

Amortisierte Analyse: Aggregation

Allgemeines Vorgehen

Bilde für jede mögliche Operationsfolge das Mittel der Kosten

Konkret:

Betrachte eine beliebige Operationsfolge und . . .

- Summiere alle Kosten
- Teile durch Anzahl Operationen

Beispiel

Gegeben sei ein binärer Zähler unbegrenzter Länge, der auf 0 initialisiert ist. Er besitzt die Operation **increment**, die den gespeicherten Wert um 1 erhöht. Ein einzelnes Bit zu flippen zählt als konstante Operation. Zu Zeigen: **increment** läuft amortisiert in konstanter Laufzeit.

- Beobachtung: bit i wird alle 2^i Aufrufe geflipt

  $\Rightarrow$im Schnitt Bit Nr. i pro Aufruf $\frac{1}{2^i}$-mal geflipt

  $\Rightarrow$im Schnitt haben Aufrufe von **increment** Kosten:

  $n \cdot \sum_{i=0}^{M} \frac{1}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = n \cdot \frac{1}{1-0{,}5} = 2n$ (M höchstwertiges gesetztes Bit, geometrische Reihe)

- Also amortisiert in $\mathcal{O}(\frac{2n}{n}) = \mathcal{O}(1)$

Peter Bohner - peter.bohner@student.kit.edu

Amortisierte Analyse: Konto

Idee

Günstige Operationen zählen zusätzlich zu ihren Kosten noch auf ein Konto, das teure Operationen benutzen, um ihre Kosten zu decken

Konkret:

- Kosten der einzelnen Operationen ermitteln
- Bestimmen, wie viel jede Operation auf das Konto zahlt/vom Konto abhebt
- amortisierte Kosten = tatsächliche Kosten + Kontoeinzahlung
- Sicherstellen, dass Konto **nie** negativ ist

Intuition (Tokens = Zeit = gut)

- Jeder Operation stehen „amortisierte Kosten viele“ Tokens zur Verfügung, sie benötigt „tatsächliche Kosten viele“
- Auf dem Konto können wir Zeittokens sparen

Beispiel

Beispiel: Dynamisches Array aus der Vorlesung mit **pushBack**

- Array wird immer Verdoppelt wenn es voll ist $\mathcal{O}(n)$
- Einfügen in Array $\mathcal{O}(1)$

$\Rightarrow$Bei einer Verdopplung mit k Elementen wurden $\frac{k}{2}$ Elemente seit der letzten Verdopplung eingefügt

$\Rightarrow$Um k Elemente zu verschieben brauchen wir k Tokens

$\Rightarrow$mit $\frac{k}{2}$ **pushBacks** musss jedes 2 Tokens aufs Konto einzahlen

- Günstige **pushBacks** 2 Tokens (Konto einzahlen) + 1 Token (Einfügen) = 3 Tokens $\in \mathcal{O}(1)$
- Teure **pushBacks** 1 (Einfügen) + k (Verdoppeln) - k (Konto abheben) = 1 $\in \mathcal{O}(1)$

Amortisierte Analyse: Charging

Allgemeines Vorgehen

Verteile Tokens (abstakte Laufzeit) von teueren zu günstigen Operationen um

Konkret:

- Weise jeder Operation eine Anzahl Tokens zu, die (asymptotisch) ihrer Laufzeit entsprechen
- Verteile Tokens von teuren Operationen auf günstiger **frühere** Operationen um, **ohne** die früheren Operationen teurer zu machen
- Die größte Anzahl Tokens bei einem Operationstyp ist die amortisierte Laufzeit pro Operation (des Typs)

Intuition (Tokens = Schulden)

- Jede Operation muss „tatsächliche Kosten“ viele Schulden begleichen, aber nur „amortisierte Kosten“ viel Geld
- Operationen, die mehr Schulden haben, als sie begleichen können, dürfen ihre Schulden auf frühere Operationen abwälzen, die noch Geld übrig haben

Beispiel

Beispiel: Dynamisches Array aus der Vorlesung mit **pushBack**

- Array wird immer Verdoppelt wenn es voll ist $\mathcal{O}(n)$
- Einfügen in Array $\mathcal{O}(1)$

$\Rightarrow$Teure **pushBacks** müssen k Tokens auf günstige Umverteilen

- Seit der letzten Verdopplung wurden $\frac{k}{2}$ günstige **pushBacks** ausgeführt

$\Rightarrow$Charge 2 Tokens auf jede der $\frac{k}{2}$ Eünfügeoperationen

- Günstige **pushBacks** 2 Tokens (Charging) + 1 Token (Einfügen) = 3 Tokens $\in \mathcal{O}(1)$
- Teure **pushBacks** 1 Token (Einfügen) + k (Verdoppeln) - k (Chargen) = 1 $\in \mathcal{O}(1)$

Peter Bohner - peter.bohner@student.kit.edu

Amortisierte Analyse: Potential

Allgemeines Vorgehen

Verwende ein Potential, um die amortisierten Kosten
zu definieren

Konkret:

- Stelle ein Potentialfunktion Φ auf, die einen Zustand auf einen nicht-negativen Wert abbildet
- Bestimme die amortisierten Kosten einer Operation durch

$$\text{amortisierte Kosten} = \text{tatsächliche Kosten} + \Phi(\text{Zustand danach}) - \Phi(\text{Zustand davor})$$

Intuition Potential

- Ist Kontostand der Kontomethode
- Wird jetzt nur noch über Zustand, nicht über vergangene Operationen definiert/bestimmt
- entspricht dem Geld, was wir in Zukunft zusätzlich ausgeben werden (müssen)
  Achtung: müssen wir dafür auch bei jedem Weg in diesen Zustand einsparen?

Peter Bohner - peter.bohner@student.kit.edu

Beispiel

Gegeben sei ein binärer Zähler unbegrenzter Länge, der auf 0 initialisiert ist. Er besitzt die Operation **increment**, die den gespeicherten Wert um 1 erhöht. Ein einzelnes Bit zu flippen zählt als konstante Operation. Zu Zeigen: **increment** läuft amortisiert in konstanter Laufzeit.

Zwei Möglichkeiten für unseren Zustand:

- Anzahl Nullen
- Anzahl Einsen

Wir wollen das unser Potential nach teuren Operationen abnimmt

$\Rightarrow$Wähle Anzahl Einsen als Potential
(Bei vielen flipps werden viele Einsen zu Null aber nur eine Null zu Eins)

Peter Bohner - peter.bohner@student.kit.edu

Beispiel

- $\Phi = m$ (Anzahl Einsen)
- Bei einer Operation in der $n+1$ Bits geflippt werden sind n davon $1 \rightarrow 0$ und 1 davon $0 \rightarrow 1$

$\Rightarrow \Phi(\text{davor}) = m, \qquad \Phi(\text{danach}) = m - n + 1$

amortisierte Kosten = tatsächliche Kosten + Φ(Zustand danach) - Φ(Zustand davor)

$n + 1 + (m - n + 1) - (m) = 2 \Rightarrow \mathcal{O}(1)$

Peter Bohner - peter.bohner@student.kit.edu

Amortisierte Analyse

Achtung

Amortisierte Analyse verändert nichts am Algorithmus!
Es ist nur eine andere Weise, über seine Performance nachzudenken.

Peter Bohner - peter.bohner@student.kit.edu

Aufgabe: Amortisierte Mensa

Eine Linie in der Mensa kann mit einer Ladung Essen genau k Mahlzeiten ausgeben, bevor aus der Küche eine neue Ladung geholt werden muss. Das holen einer neuen Ladung kostet k Zeit. Zeige, dass die Mensa in amortisiert $O(1)$ Zeit eine Mahlzeit ausgibt.

Operation $:=$ Ausgeben einer Mahlzeit // $O(1)$

Nach k Operationen k extra Kosten

Hinweis: Betrachte Operationenfolgen mit $n \cdot k$ Operationen

Aggregation: Nach $n \cdot k$ Operationen $2n \cdot k$ Kosten, also $\frac{2n \cdot k}{n \cdot k} = O(1)$ Kosten pro Operation

Potential: Definiere dazu Φ(M)$:=$ k - m, $0 < m \leq k$ // m = #übrige Mahlzeiten

Für günstige Operationen gilt: $\Phi(M_{nach}) - \Phi(M_{vor})$ = (k - (m - 1)) - (k - m) = 1 // $\in O(1)$
Für teure Operationen gilt: $\Phi(M_{nach}) - \Phi(M_{vor})$ = (k - k) - (k - 1) = 1 - k
Amortisierte Kosten = tatsächliche Kosten + 1 - k = 1 // $\in O(1)$

Peter Bohner - peter.bohner@student.kit.edu

Aufgabe: Amortisierte Mensa

Charging: Bevor k Mahlzeiten Nachgeholt werden müssen, werden k Mahlzeiten Ausgegeben

$\Rightarrow$Charge 1 Token auf jedes Ausgeben

$\Rightarrow$Jedes günstige Ausgeben hat 2 Token, jedes teure 1 Token $\Rightarrow \mathcal{O}(1)$

Konto: Analog zu Charging

Peter Bohner - peter.bohner@student.kit.edu

Aufgabe: Push/Pop Front

Es sollen die Methoden **popFront**(), **pushFront**(x) implementiert werden. Dabei sollen **nicht** wie bei pushBack direkt Operationen auf dem Speicher ausgeführt werden. Beschreibt die Funktionsweise von **popFront**(), **pushFront**(x) in Worten und analysiert jeweils die Laufzeit.
Wie verhält sich die amortisierte Laufzeit?

- Ihr dürft **push** - bzw. **popBack** für eure Methoden benutzen

Peter Bohner - peter.bohner@student.kit.edu

Lösung

popFront()

1. Rotiere alle Elemente nach links (das erste Element kommt dabei an die letzte Stelle)

2. Führe **popBack**() aus und gebe den Rückgabewert zurück

$\Rightarrow$ 1. in $\Theta(n)$, 2. in $O(n)$ $\Rightarrow \Theta(n)$

pushFront(x)

1. Führe **pushBack**(x) aus

2. Rotiere alle Elemente nach rechts (dabei kommt das letzte Element an die erste Stelle)

$\Rightarrow$ 1. in $O(n)$, 2. in $\Theta(n)$ $\Rightarrow \Theta(n)$

amortisierte Laufzeit?
Jede Operation macht asymptotisch gleich viel Arbeit
$\Rightarrow$ Kosten können nicht amortisiert werden

Aufgabe: Duplikate Entfernen

Formuliere einen Algorithmus in Pseudocode, der ein **sortiertes** Array als Eingabe bekommt und alle Duplikate daraus entfernen soll.
Das duplikatfreie Array soll zurückgegeben werden.

Beispiel: removeDup($\langle 1, 2, 2, 3 \rangle$) $\rightsquigarrow \langle 1, 2, 3 \rangle$

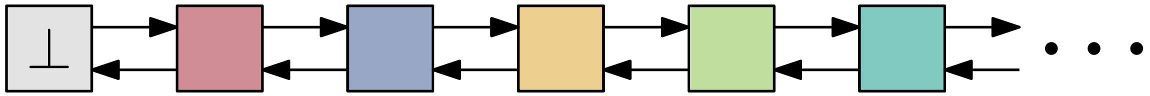
```
removeDup(A):
    D := new empty Array
    if A.empty() then return D
    prev := A[0]
    D.pushBack(prev)
    for i = 1 to A.size() - 1 do
        if A[i] = prev then continue // duplikat
        else
            prev := A[i]
            D.pushBack(prev)
    return D
```

Laufzeit? $O(n)$

Frage: Wie können wir Duplikate in beliebigen Arrays entfernen? Was wäre die Laufzeit eines solchen Algorithmus?

Peter Bohner - peter.bohner@student.kit.edu

Neue Datenstruktur: Listen



pushBack	$\Theta(1)$	
popBack	$\Theta(1)$	
get($i\colon \mathbb{N}_0$)	$\Theta(n)$	
remove($i\colon \mathbb{N}_0$)	$\Theta(n)$	
find(x)	$\Theta(n)$	
remove(x)	$\Theta(n)$	// in $\Theta(1)$, wenn Knoten bekannt
concat	$\Theta(1)$	
splice	$\Theta(1)$	
size	$\Theta(n)$	// $\Theta(1)$, ohne Splice

(Doppelt verkettete) Liste

Die Liste bildet sich aus verketteten Listen-Knoten. Dabei werden Anfangs- und Endknoten mittels Pointer gespeichert.

Variation:

- Einfach verkettete Liste

https://algo-code.iti.kit.edu/

Zusammenfassung

Karlsruher Institut für Technologie

Was haben wir heute gemacht?

- Dynamische Arrays
- Amortisierte Analyse
- Listen

Worauf könnt ihr euch nächste Woche freuen?

- Suchen
- Sortieren

Fragen?

Fragen!

Peter Bohner - peter.bohner@student.kit.edu

Ende

https://xkcd.com/1445/