Algorithmen I
Tutorium 32

Eine Lehrveranstaltung im SS 2017 (mit Folien von Christopher Hommel)

Daniel Jungkind (ufesa@kit.edu) | 19. Mai 2017

INSTITUT FÜR THEORETISCHE INFORMATIK

Datenstrukturen – dynamisch viele Elemente

Arrays = toll, aber ...

- nur begrenzt groß
- *von Anfang an* volle Größe
- Einfügen zwischendrin ist scheiße

(Doppelt) Verkettete Listen

- Mehrere **Segmente** (*Items*): durch Referenzen (*Handles*) jeweils miteinander „verbunden“
- Ein **Segment** besteht jeweils aus
  - dem **Element**: Eigentliches Datum an der Stelle
  - **next**: Referenz auf das nächste Segment
  - **prev**: Referenz auf das vorherige Segment

$\Rightarrow$ Ganze Liste von *einem* Segment aus erreichbar! (Dank Verlinkung)

- Invariante: $\mathit{next}{\rightarrow}\mathit{prev} = \mathit{prev}{\rightarrow}\mathit{next} =$ **this**

(Doppelt) Verkettete Listen

- Was folgt auf das **letzte** Segment? Was kommt vor dem **ersten** Segment?
- Möglichkeit: Nullpointer
  – Viele Sonderfälle und die Invariante geht **kaputt** ☹
- Praktischer: Zyklische Verknüpfung
- Anfang, Ende erkennen? Leere Liste!?

⇒ Verwende einen **Dummy-Header** (*h*) – hält kein Element und markiert „breaking point“:
  - *h.next*: **erstes** Segment der eigentlichen Liste
  - *h.prev*: **letztes** Segment der eigentlichen Liste

+ Bequemer Code, denn: Viel **weniger Sonderfälle** und eine happy Invariante! ☺

(Doppelt) Verkettete Listen

- Jetzt möglich:
  - **Einfügen** und **Entfernen** in $O(1)$ (wir brauchen eine konkrete Stelle)
  - **Inter-List-Splice** (Abschnitte zwischen Listen umhängen) in $O(1)$
  - Platzbedarf linear
- Auch möglich: **Einfach** verketten statt doppelt $\Rightarrow$ Weniger Speicher
  $\Rightarrow$ schränkt manche Funktionen deutlich ein

Einschub

Inter-List-Splice

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

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

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

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

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

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

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

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

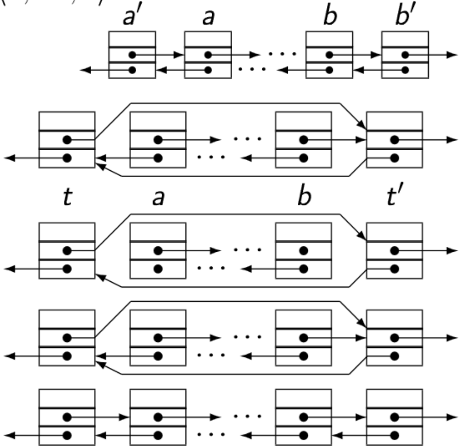
  $t' := t \rightarrow \text{next}$ //

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

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

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

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



Verkettete Listen

Anwendung: Konstruierung einer *Unbounded Queue*

// Erinnerung: splice$(a, b, t : Handle)$ schneidet $\langle a, ..., b\rangle$ aus
// seiner zugehörigen Liste aus und fügt es nach t ein

$head = createHandle(\bot) : Handle$

procedure pushBack$(e : Element)$
  $new := createHandle(e)$
  $new{\rightarrow}prev := new{\rightarrow}next := new$
  $splice(new, new, head{\rightarrow}prev)$

function popBack: *Element*
  assert $\neg isEmpty$
  $old := head{\rightarrow}prev$
  $e := old{\rightarrow}e$
  $d := createHandle(\bot)$
  $d{\rightarrow}prev := d{\rightarrow}next := d$

  $splice(old, old, d)$

  dispose old, d
  return e

Verkettete Listen

Karlsruher Institut für Technologie

Anwendung: Konstruierung einer *Unbounded Queue*

// Erinnerung: splice$(a, b, t : \mathit{Handle})$ schneidet $\langle a, \ldots, b\rangle$ aus
// seiner zugehörigen Liste aus und fügt es nach t ein

$head = createHandle(\bot) : Handle$

procedure pushFront(e : *Element*)
- $new := createHandle(e)$
- $new \rightarrow prev := new \rightarrow next := new$
- $splice(new, new, head)$

function popFront: *Element*
- **assert** $\neg isEmpty$
- $old := head \rightarrow next$
- $e := old \rightarrow e$
- $d := createHandle(\bot)$
- $d \rightarrow prev := d \rightarrow next := d$
- $splice(old, old, d)$
- **dispose** old, d
- **return** e

Verkettete Listen – Zweifel?

Arrays: nur begrenzt toll
verkettete Listen: **unbegrenzt toll**? (#WorstPunEver)

- List[i] nicht in $O(1)$, sondern linear
- List.*size* in $O(1)$ $\Leftrightarrow$ *Inter-List-Splice*: Autsch
- Cache-Freundlichkeit sieht anders aus

Einschub: Cache-Freundlichkeit

- **Cache**: Prozessor-nahes Datenzwischenlager. Schneller als RAM.
- **Lokalitätsprinzip**: Irgendwo Zugriff im RAM ⇒ demnächst wieder Zugriff in der Nähe

⇒ *Idee*: Nicht nur **einen** Wert im Cache puffern, sondern dessen **Nachbarschaft** gleich mit!
Kostet auch nicht mehr.

- Verkettete Listen: Segmente **nach Bedarf** angelegt:
  Landen da, wo's passt ⇒ **kreuz und quer** im RAM verteilt

⇒ Vorgänger/Nachfolger sicher **nicht nebeneinander** ⇒ selten gemeinsam im Cache

- Wie kriegen wir bloß Daten **zusammenhängend** in den Speicher?

Arrays to the rescue!

- Array läuft voll $\Rightarrow$ Größeres Array anlegen und Daten umkopieren
- Naiv: Einfügen von n Elementen in $\Theta(n^2)$
- **Trick**: Jedes Mal ein doppelt so großes Array anlegen
- Ist das nicht trotzdem **teuer**!?

Amortisierte Analyse – a taste

Einfügen in ein **volles** Array (Größe $2n$):

$\Rightarrow$ Array muss vergrößert (also umkopiert werden)

- Genau **dieses** Einfügen kostet jetzt $2n$ (wegen Kopiererei)
- **Vorher** mind. n Einfüge-Operationen in $O(1)$

$\Rightarrow$ **Amortisierte** Kopierkosten pro Einfügen: $\frac{2n}{n} = 2 \in O(1)$

Genauso: **Entfernen** aus einem **viertel-vollen** Array (n von $4n$ gefüllt)

$\Rightarrow$ Array muss verkleinert (= umkopiert) werden

- Genau **dieses** Entfernen kostet n (wegen Kopiererei)
- **Vorher** mind. n Entfern-Operationen in $O(1)$

$\Rightarrow$ **Amortisierte** Kopierkosten pro Entfernen: $\frac{n}{n} = 1 \in O(1)$

Amortisierte Analyse – How to

Entweder

Aggregatmethode: Schätze nach oben ab:

$$\text{Gesamtkosten von } n \text{ beliebigen Ops} = \text{„}T_{\text{Gesamt}}\text{“} \leqslant c \cdot n$$

(c irgendeine Konstante).

Knifflig: Diese Abschätzung finden und zeigen.

Amortisierte Analyse – How to

... oder

Kontomethode: Kosten von teuren Operationen auf die billigen **umlegen**

Für jede Op der Art i: c_i „Münzen" auf ein „Konto" einzahlen (c_i **konstant**!)

Bsp.: Arten von Ops {Einfügen, Löschen} $\Rightarrow c_{\text{Einfügen}}, c_{\text{Löschen}}$ festlegen)

Begründen:

Wenn mal eine Op **mehr als konstante Zeit** kostet (sagen wir x) $\Rightarrow$ auf dem Konto mind. x Münzen da, um das zu bezahlen.

Knifflig: Begründen und die jeweiligen c_i finden.

Amortisierte Analyse – How to

- **Generell**: Genau überlegen, unter welchen Vorbedingungen die teuren Operationen auftreten
- **Aufgabenstellung** beachten, ob spezifische Methode gefordert ist! (Falls nein $\Rightarrow$ klare logische Begründung des Sachverhaltes reicht (im Prinzip))

Amortisierte Analyse

Aufgabe 1: Hochgestackte Ziele

Gegeben seien zwei Stacks mit *size*$()$: $\mathbb{N}_0$, *pushBack*$($*e* : *Element*$)$ und *popBack*$()$: *Element*, alle jeweils in konstanter Zeit.
Entwerft daraus eine Queue, die die Operationen *pushBack*$($*e* : *Element*$)$ und *popFront*$()$: *Element* jeweils in amortisiert konstanter Zeit beherrscht.

Amortisierte Analyse

Lösung zu Aufgabe 1

- ein „**Input**-Stack“, ein „**Output**-Stack“
- Queue.*pushBack*: auf den Input-Stack legen
- Queue.*popFront*: vom Output-Stack oben wegnehmen
  Output $= \varnothing$?
  $\Rightarrow$ Gesamten **Input**-Stack nach und nach auf Output **umschaufeln** (in $O(n)$). Dabei wird automatisch Reihenfolge richtigrum-gedreht.
- Queue.*pushBack* sowieso in $O(1)$
- Queue.*popFront* hat **einmal** Kopieraufwand n, aber **danach** geht *popFront* n-mal in $O(1)$
  $\Rightarrow$ amortisiert in $O(1)$

Amortisierte Analyse

Aufgabe 2: A little bit more?

Gegeben sei ein binärer Zähler mit einer unbegrenzten Anzahl an Bits, die alle auf 0 initialisiert sind. Der Zähler besitzt die Operation *increment* (erhöht den im Zähler gespeicherten Wert um 1). Ein einzelnes Bit zu flippen zählt jeweils als eine konstante Operation. Zeigt anhand der **Aggregatmethode**, dass die Operation *increment* stets in amortisiert konstanter Zeit läuft.

Amortisierte Analyse

Lösung zu Aufgabe 2

Beobachtung:

Bit Nr. i $(i \geqslant 0)$ wird genau alle 2^i Aufrufe geflippt

(d.h. Bit Nr. 0 bei jedem Aufruf, Bit Nr. 1 alle zwei Aufrufe usw.)

$\Rightarrow$ im Schnitt wird Bit Nr. i pro Aufruf „$\frac{1}{2^i}$-mal" geflippt

$\Rightarrow$ im Schnitt haben n Aufrufe von *increment* die Kosten

$$n \cdot \sum_{i=0}^{M} \frac{1}{2^i} < n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 \cdot n$$

(M: Index des höchsten gesetzten Bits)

$\Rightarrow$ läuft amortisiert in $O(1)$.

Amortisierte Analyse

Aufgabe 3: Bitte null Bit!

Gegeben sei ein binärer Zähler mit einer unbegrenzten Anzahl an Bits, die alle auf 0 initialisiert sind. Der Zähler besitzt die Operationen *increment* (erhöht den im Zähler gespeicherten Wert um 1) **und** *reset* (setzt den im Zähler gespeicherten Wert auf 0 zurück). Ein einzelnes Bit zu flippen zählt jeweils als eine konstante Operation. Zeigt anhand der **Kontomethode**, dass beide Methoden stets in amortisiert konstanter Zeit laufen.

Amortisierte Analyse

Lösung zu Aufgabe 3

Beobachtung:

- Bei jedem *increment*-Aufruf:
  genau ein Bit von $0 \rightsquigarrow 1 \implies O(1)$
  u. U. **viele** Bits von $1 \rightsquigarrow 0 \implies$ nicht-konstante Zeit
- Bei jedem *reset*-Aufruf:
  u. U. **viele** Bits von $1 \rightsquigarrow 0 \implies$ nicht-konstante Zeit

Idee: Jedes Mal wenn ein Bit von $0 \rightsquigarrow 1$ (pro Methodenaufruf max. 1-mal!): $+1$ Münze einzahlen. Wenn das entspr. Bit wieder von $1 \rightsquigarrow 0$: Bezahle mit dieser Münze.

$\Rightarrow$ Anzahl 1er im Zähler $=$ Anzahl Münzen auf dem Konto

$\Rightarrow$ Bei beliebiger Op kann das Setzen von n Bits von $1 \rightsquigarrow 0$ garantiert mit n Münzen bezahlt werden.

$\Rightarrow$ Beide Ops laufen amortisiert in $O(1)$.

Datenstrukturen

Aufgabe 4

Nehmt an, dass eine Speicherallokation beliebiger Größe in $O(1)$ läuft.
Entwickelt eine Datenstruktur mit

- $\mathit{pushBack}(e : \mathit{Element})$ und $\mathit{popBack}() : \mathit{Element}$ in $O(1)$ im Worst-Case (**nicht** nur amortisiert)
- Zugriff auf das k-te Element in $O(\log n)$ im Worst-Case (**nicht** nur amortisiert)

Datenstrukturen

Lösung zu Aufgabe 4

Lege eine verkettete Liste von Arrays an, deren Größe sich jeweils verdoppelt, und habe immer eine Referenz auf das letzte Array inklusive Index des letzten belegten Slots

- Element anfügen: Falls Slot frei $\Rightarrow$ offensichtlich konstant. Falls kein Slot frei $\Rightarrow$ lege neues Array doppelter Größe an, füge es an die verkettete Liste hinzu, lege Element rein (alles in konstanter Zeit möglich)
- Zugriff auf das k-te Element: Laufe die verkettete Liste ab und verringere den Index um die Länge des aktuell betrachteten Arrays, bis der Index für das aktuell betrachtete Array gültig ist. Es sind insgesamt logarithmisch viele Arrays, also auch logarithmische Laufzeit

Datenstrukturen

Aufgabe 5: Aufgabe 4 remastered

Nehmt an, dass eine Speicherallokation beliebiger Größe in $O(1)$ läuft.
Entwickelt eine Datenstruktur mit

- $pushBack(e : Element)$ und $popBack() : Element$ in $O(\log n)$ im Worst-Case (**nicht** nur amortisiert)
- Zugriff auf das k-te Element in $O(1)$ im Worst-Case (**nicht** nur amortisiert)

Datenstrukturen

Lösung zu Aufgabe 5

Analog zur Lösung von Aufgabe 4: Lege ein Array A mit Referenzen auf Arrays an, deren Größe sich jeweils verdoppelt (d.h. das Array an der Stelle $A[i]$ hat die Größe 2^i)

- Element anfügen: Falls Slot frei $\Rightarrow$ offensichtlich konstant. Falls kein Slot frei $\Rightarrow$ lege neues Array B der Größe $2^{|A|}$ an, kopiere A mit logarithmisch vielen Array-Referenzen in ein neues Array der Größe $|A| + 1$ (an dessen Ende B landet) und füge das Element in B ein $\Rightarrow \; O(\log n)$
- Zugriff auf das k-te Element: Berechne mit dem Logarithmus den Index von B in A und den Index des k-ten Elements in B, der Zugriff ist in konstanter Zeit.