

Algorithmen 1 SS 2013 – Tutorium 7

3. Tutorium

Sarah Lutteropp

7. Mai 2013

Übersicht

1 2. Übungsblatt

2 Datenstrukturen

3 Amortisierte Analyse

- Aggregatmethode
- Accountmethode

4 Kreativaufgabe

2. Übungsblatt

Allgemeines zur Korrektur

Nachtrag: Halbe Punkte doch erlaubt (bei Teilaufgaben). Nur die Aufgabe gesamt darf keine halben Punkte haben.

2. Übungsblatt

Möchte jemand vorrechnen?

Dynamische Datenstrukturen

- Dynamische Datenstrukturen vs. statische Datenstrukturen
- Typische Vertreter: Stacks, Queues, Listen, Bäume

Stacks

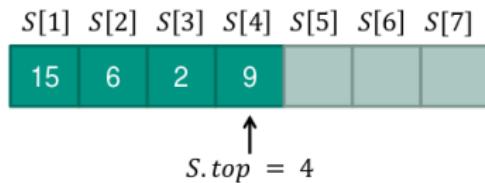
Stack (Stapel)

Last In, First Out Prinzip (LIFO)

Operationen

- *push*: Auf den Stapel legen
- *pop*: Vom Stapel nehmen

Implementierung



Fehlerfälle

- stack underflow
- stack overflow

Queues

Queue (Warteschlange)

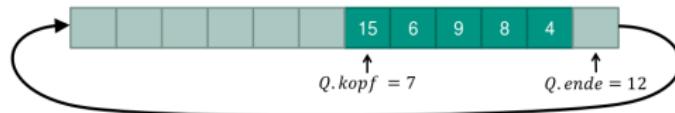
First In, First Out Prinzip (FIFO)

Operationen

- *enqueue*: Als letzter an die Schlange anstellen
- *dequeue*: Ersten in der Schlange abarbeiten

Implementierung

Feld der Größe n für Queue der Größe $n - 1$



- Zur Unterscheidung leer/voll
- Feld wird als Ring aufgefasst

Stacks und Queues

Aufgabe

Implementiere eine Queue mithilfe von zwei Stacks.

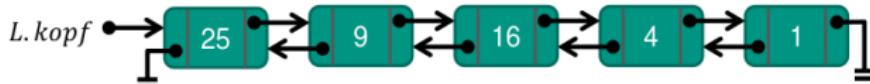
Stacks und Queues

Und andersrum ...

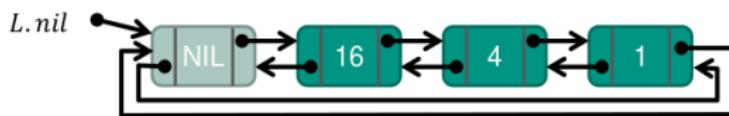
Implementiere einen Stack mithilfe von zwei Queues.

Verkettete Listen

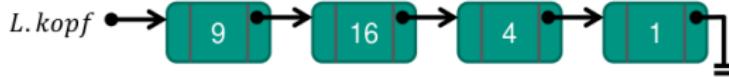
Doppelt verkettete Liste



Zyklische, doppelt verkettete Liste mit Wächterelement (Sentinel)



Einfach verkettete Liste



Unbounded Arrays

Unbounded Array

- Array voll: Kopiere in ein doppelt so großes Array
- Weniger als $\frac{1}{4}$ vom Array belegt: Kopiere in ein halb so großes Array

Kosten beim Kopieren

Amortisiert in $\mathcal{O}(n)$.

Vergleich Listen und Unbounded Arrays

Wann nimmt man was?

Vergleich Listen und Unbounded Arrays

Wann nimmt man was?

Beispielargumente: Unbounded Array bietet Random Access, verkettete Liste ermöglicht einfaches Löschen, verkettete Liste hat Verschnitt, Unbounded Array braucht keine Zeiger, ...

Allgemeine Analyse vs. amortisierte Analyse

Allgemeine Analyse

Betrachtet die maximalen Kosten der einzelnen Operationen

Amortisierte Analyse

Betrachtet die durchschnittlichen Kosten von Folgen von Operationen.

- Garantiert die mittlere Performanz jeder Operation im schlechtesten Fall
- Gibt bessere Laufzeit, wenn teure Schritte selten



Aggregatmethode

Aggregatmethode

Wir zeigen ...

Eine Folge von n (nicht notwendigerweise gleichen) Operationen benötigt im schlimmsten Fall die Zeit $T(n)$.

Dann wissen wir ...

Die amortisierten Kosten pro Operation sind $\frac{T(n)}{n}$.

Beispiel Binärzähler

k -Bit Binärzähler ist Feld $A[0..k - 1]$. Wir betrachten die Laufzeit einer *increment()*-Operation.



Aggregatmethode

Aggregatmethode

Anzahl umgekippte Bits

- $A[0]$ kippt n mal um.
- $A[1]$ kippt $\lfloor \frac{n}{2} \rfloor$ mal um.
- $A[i]$ kippt $\lfloor \frac{n}{2^i} \rfloor$ mal um.

Abschätzung von n *increment()*-Operationen

$$T(n) = n + \lfloor \frac{n}{2} \rfloor + \dots + \lfloor \frac{n}{2^k} \rfloor = n \cdot \sum_{i=0}^k \lfloor \frac{n}{2^i} \rfloor \leq n \cdot \sum_{i=0}^{\infty} \lfloor \frac{n}{2^i} \rfloor = 2 \cdot n$$

⇒ Eine Kipp-Operation liegt in $\frac{T(n)}{n} = \frac{2n}{n} = 2 \in \mathcal{O}(1)$.



Accountmethode

Wie geht die Accountmethode?

Wir weisen jeder Operation neue Kosten zu (höher oder niedriger als die realen, allerdings müssen die Gesamtkosten wieder erreicht werden). Durch das Mehr bei den zugewiesenen Kosten, die höher sind als die realen Kosten, bauen wir Guthaben (account) auf, das wir für die zu niedrig zugewiesenen Kosten wieder verbrauchen.

Beispiel Binärzähler

k -Bit Binärzähler ist Feld $A[0..k - 1]$. Wir betrachten die Laufzeit einer *increment()*-Operation.



Accountmethode

Accountmethode

Beobachtung

- Bei jeder *increment()*-Operation kippt genau eine 0 auf 1
- Es kippen $\leq k$ Bits von 1 auf 0
- Ein Bit kippt nur auf 0, wenn es vorher auf 1 gekippt wurde

Kostenzuweisung

Reale Kosten

- $0 \rightarrow 1 : 1, 1 \rightarrow 0 : 1$

Zugewiesene Kosten

- $0 \rightarrow 1 : 2, 1 \rightarrow 0 : 0$

\Rightarrow *increment()* kostet amortisiert $2 \in \mathcal{O}(1)$

Kreativaufgabe



Entwickeln Sie eine Datenstruktur, die folgendes kann:

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

Kreativaufgabe

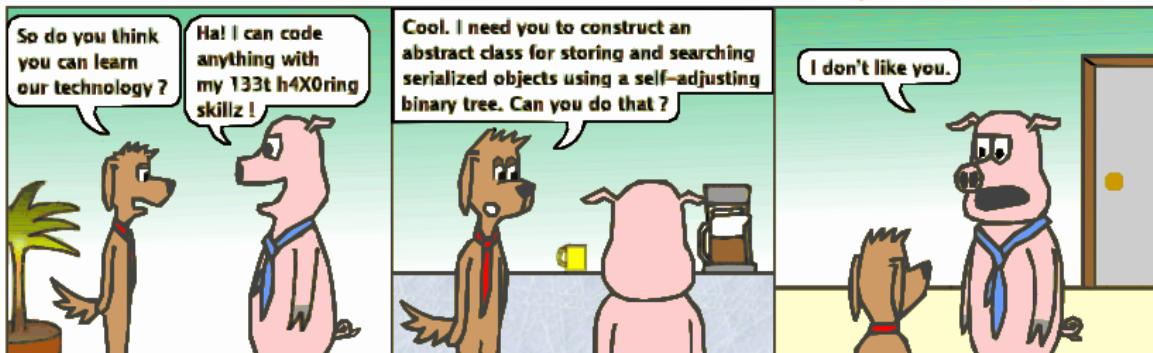


Das geht auch umgekehrt:

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

Bis zum nächsten Mal! 😊

Hackles



<http://hackles.org>

Copyright © 2001 Drake Emko & Jen Brodzik