Karlsruher Institut für Technologie

Karlsruher Institut für Technologie

Algorithmen I
Tutorium 32

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

Daniel Jungkind (ufesa@kit.edu) | 28. Juli 2017

INSTITUT FÜR THEORETISCHE INFORMATIK

Pingo-Time! ☺

Karlsruher Institut für Technologie

http://pingo.upb.de/685177

WIEDERHOLUNG

„Mehr Schweiß in der Vorbereitung, weniger Blut in der Schlacht." – Alexander Wassiljewitsch Suworow

O-Kalkül

$o(f(n))$	$<$	echt schwächer wachsende Funktionen
$O(f(n))$	$\leqslant$	schwächer oder gleich stark wachsende Funktionen
$\Theta(f(n))$	$=$	genau gleich stark wachsende Funktionen
$\Omega(f(n))$	$\geqslant$	stärker oder gleich stark wachsende Funktionen
$\omega(f(n))$	$>$	echt stärker wachsende Funktionen

Algorithmenanalyse

O-Kalkül: Formeln

$f(n) \in o(g(n))$	$\Longleftrightarrow$	$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in O(g(n))$	$\Longleftrightarrow$	$0 \leqslant \limsup_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$
$f(n) \in \Theta(g(n))$	! $\Longleftarrow$!	$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$
$f(n) \in \Omega(g(n))$	$\Longleftrightarrow$	$0 < \liminf_{n \to \infty} \frac{f(n)}{g(n)} = c \leqslant \infty$
$f(n) \in \omega(g(n))$	$\Longleftrightarrow$	$\limsup_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Algorithmenanalyse

Karlsruher Institut für Technologie

Korrektheitsbeweis

- Korrektheitsbeweis ist **zweiteilig**:
  - 1. Teil – **Funktionalität**: Mit Invariante beweisen, dass der Algorithmus ein **korrektes** Ergebnis erzeugt
  - 2. Teil – **Terminierung**: Beweisen (ggf. anhand einer Invariante), dass der Algorithmus „irgendwann **fertig** wird“.
- **Aufgabenstellung beachten**: Wenn („nur“) eine Invariante angegeben/bewiesen werden soll $\Rightarrow$ Terminierungsbeweis nicht nötig!

Algorithmenanalyse

Karlsruher Institut für Technologie

Invarianten

- Invariante finden: Manchmal offensichtlich, manchmal **Kreativität** gefragt
- Invarianten beweisen – im Prinzip **Induktion**:
- „IA“: Invariante gilt bei **Beginn** des Algorithmus / der Schleife
- „IV“: Die Invariante war beim Ende des **vorherigen** Ausführungsschrittes gültig
- „IS“: Mithilfe der IV zeigen, dass die Invariante auch beim Ende des **aktuellen** Ausführungsschrittes gültig ist
- **Achtung**: Invarianten müssen auch **nach Ende der Schleife** noch gelten!

Algorithmenanalyse

Karlsruher Institut für Technologie

Beispiele für Invarianten

- **Binäre Suche**: Gesuchtes Element kann **nicht** im ignorierten Bereich liegen
- **Quicksort**: Links $\leqslant$ *pivot* $<$ Rechts
- **Mergesort**: Listen, die von rekursiven Aufrufen zurückgegeben werden, sind **sortiert**
- **Dijkstra**: Endgültiger kürzester Pfad zum *min* der *PriorityQueue* ist bekannt
- Doppelt verkettete **Liste**: $\mathit{next} \rightarrow \mathit{prev} = \mathit{prev} \rightarrow \mathit{next} =$ **this**

Algorithmenanalyse

Karlsruher Institut für Technologie

Aufgabe: Korrektheitsbeweis

Beweist die Korrektheit von *ArraySum*:

function ArraySum(A : **array**$[1..n]$ **of** $\mathbb{R}$) : $\mathbb{R}$

$i := 1$

$s := 0$

while $i \leqslant n$ **do**

invariant ???

$s := s + A[i]$

$i{++}$

return s

Algorithmenanalyse

Karlsruher Institut für Technologie

Aufgabe: Korrektheitsbeweis

Beweist die Korrektheit von *ArraySum*:

function ArraySum(A : **array**$[1..n]$ **of** $\mathbb{R}$) : $\mathbb{R}$

$i := 1$

$s := 0$

while $i \leqslant n$ **do**

invariant $1 \leqslant i \leqslant n+1$ **and** $s = \sum_{1}^{i-1} A[i]$

$s := s + A[i]$

$i{+}{+}$

return s

Das Master-Theorem (einfache Form)

a, b, c, d positive Konstanten und für $n \in \mathbb{N}$ sei

$$T(n) = \begin{cases} a, & \text{für } n = 1 \\ d \cdot T(\lceil \frac{n}{b} \rceil) + cn, & \text{für } n > 1 \end{cases}.$$

Dann gilt:

$$T(n) \in \begin{cases} \Theta(n), & d < b \\ \Theta(n \log n), & d = b \\ \Theta(n^{\log_b d}), & d > b \end{cases}.$$

Amortisierte Analyse

How to

- **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.
- **Kontomethode**: Zahle für jede Operation eine **konstante** Menge c an Münzen aufs Konto ein. Zeige: Bei **nicht-konstanten** Operationen mit Kosten k müssen **mindestens** k Münzen aufm Konto sein. (Knifflig: Begründung und geeignetes c finden)
- **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))

Aufgabe: 2016 Blatt 3 A4 „Weltherrschaftskonferenz“

Listen und Co.

Doppelt verkettete Liste

- Invariante: $\mathit{next}{\rightarrow}\mathit{prev} = \mathit{prev}{\rightarrow}\mathit{next} = \textbf{this}$
- **Dummy-Header** h für Bequemlichkeit und als Sentinel (Wächter-Element) beim Suchen
- \+ Flexibel
- − Nicht cachefreundlich

Unbeschränktes Array

- Array voll $\Rightarrow$ Ziehe in **doppelt** so großes Array um
- Array viertel-voll $\Rightarrow$ Ziehe in **halb** so großes Array um
- \+ Cachefreundlich
- − Eher unflexibel

Listen und Co.

Listen vs. Arrays

Operation	(Doppelt verkettet) List	(Einzeln verkettet) SList	(unbounded array) UArray	(cyclic unbounded array) CArray	explanation '*'
$[\cdot]$ (Indexzugriff)	n	n	1	1	
$\lvert\cdot\rvert$ (Länge abrufen)	1^*	1^*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	insertAfter only
remove	1	1^*	n	n	removeAfter only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache-efficient

Listen und Co.

Aufgabe

Entwerft einen Stack, der *push*, *pop* und *min* kann und zwar in $O(1)$ (nicht amortisiert).

Listen und Co.

Lösung

$BasicStack, MinimumStack : Stack$

function min
  return *MinimumStack*.getTop

procedure push(e)
  BasicStack.push(e)
  if $e \leqslant$ min **then** *MinimumStack*.push(e)

function pop
  $r :=$ *BasicStack*.pop()
  if $r =$ min **then** *MinimumStack*.pop()
  return r

Hashing

- **Erwartete** Laufzeit!
- **Hashfunktion** h weist Elemente einem Platz zu
- **Universelle** Hashfunktionen — Vorlesung:
  Wenn $n \in O(m)$ Elemente in die Hashtable eingefügt werden $\Rightarrow$ erwartete $|\text{Kollisionen}| \in O(1)$
- Typische univ. Hashfunktion:
  $h_a(x) := a \cdot x \textbf{ mod } m \quad (0 < a < m) \quad (m\ \textbf{prim}!)$
- Oder generisch (z. B., falls in Klausur nötig): „Sei h eine beliebige Hashfunktion aus der Familie universeller Hashfunktionen“

Hashing

Hashing mit verketteten Listen

$\Rightarrow$ Halte **array of** Lists:
Werfe Element in die Liste, suche es dort

Hashing mit linearer Suche

$\Rightarrow$ Nur array of Element:
Platz besetzt? Gucke rechts davon.
Beim **Löschen**: Ggfs. wieder nach links zurückschieben!

- Ganz rechts im Array Platz dicht?
  $\Rightarrow$ Pufferbereich (der dann hoffentlich langt) oder
  $\Rightarrow$ Zyklisch

Hashing

Hashing mit verketteten Listen

$\Rightarrow$ Halte **array of** Lists:
Werfe Element in die Liste, suche es dort

Hashing mit linearer Suche

$\Rightarrow$ Nur **array of** Element:
Platz besetzt? Gucke rechts davon.
Beim **Löschen**: Ggfs. wieder nach links zurückschieben!

- Ganz rechts im Array Platz dicht?
  $\Rightarrow$ Pufferbereich (der dann hoffentlich langt) oder
  $\Rightarrow$ Zyklisch

Sortieralgorithmen

Vergleichsbasiert

- InsertionSort
- (SelectionSort)
- (BubbleSort)
- Mergesort
- Quicksort
- Heapsort (absteigende Sortierung!)

Ganzzahlig

- BucketSort
- CountingSort
- RadixSort

Sorting Algorithms

Karlsruher Institut für Technologie

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Stabilität**.

InsertionSort SelectionSort Mergesort CountingSort Bucketsort Radixsort	**Ja**
Heapsort Quicksort	**Nein**

Sorting Algorithms

Karlsruher Institut für Technologie

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Stabilität**.

InsertionSort SelectionSort Mergesort CountingSort Bucketsort Radixsort	**Ja**
Heapsort Quicksort	**Nein**

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Cache-Effizienz**.

InsertionSort SelectionSort Heapsort CountingSort Quicksort	Ja
Bucketsort Mergesort Radixsort	Nein

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Cache-Effizienz**.

InsertionSort SelectionSort Heapsort CountingSort Quicksort	**Ja**
Bucketsort Mergesort Radixsort	**Nein**

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Platzverbrauch**.

InsertionSort SelectionSort Mergesort (ohne Rekursionsoverhead) Quicksort (ohne Rekursionsoverhead)	$O(1)$
Heapsort	$O(n)$
CountingSort Bucketsort	$O(n+k)$
Radixsort	$O(n+K)$

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Platzverbrauch**.

InsertionSort SelectionSort Mergesort (ohne Rekursionsoverhead) Quicksort (ohne Rekursionsoverhead)	$O(1)$
Heapsort	$O(n)$
CountingSort Bucketsort	$O(n+k)$
Radixsort	$O(n+K)$

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Worst-Case-Laufzeit**.

Mergesort Heapsort	$O(n \log n)$
Quicksort InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue“)

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Worst-Case-Laufzeit**.

Mergesort Heapsort	$O(n \log n)$
Quicksort InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue“)

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **„Standard-Laufzeit“**.

Mergesort Heapsort Quicksort (erwartet)	$O(n \log n)$
InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue“)

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **„Standard-Laufzeit“**.

Mergesort Heapsort Quicksort (erwartet)	$O(n \log n)$
InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue“)

Binäre Heaps

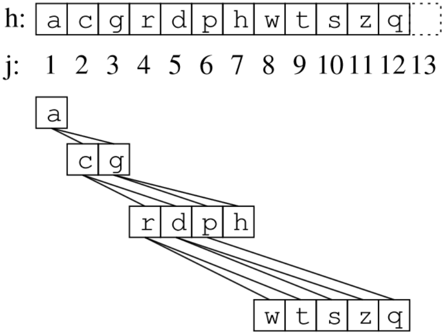
Implementierung

- Repräsentiere binären Baum als **array**$[1 \ldots n]$ mit **Heap-Eigenschaft**
- Die Ebenen des Baumes liegen von **oben** $\rightsquigarrow$ **unten** und von **links** $\rightsquigarrow$ **rechts** nacheinander im Array
- Von Knoten j kriegt man **Eltern** und **Kinder** wie folgt:

$$parent(j) = \left\lfloor \frac{j}{2} \right\rfloor$$

$$leftChild(j) = 2j$$

$$rightChild(j) = 2j + 1$$

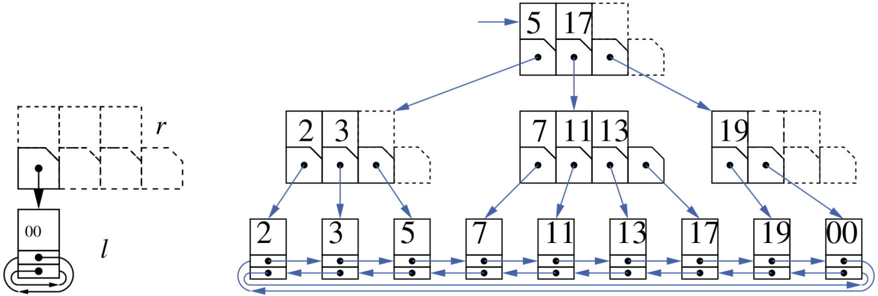


Binäre Heaps

- *insert*: Nach unten und *siftUp*
- *deleteMin*: Element ganz unten rechts $\rightsquigarrow$ ganz oben, *siftDown*
- *buildHeap*: „Ebenen“ von unten $\rightsquigarrow$ oben durchgehen und „down-siften“ in $O(n)$

Sortierte Folgen – (a, b)-Bäume

Beispiel: (2, 4)-Baum („00“ steht in VL für ∞)



Graphen

Repräsentationen

- Kantenfolge
- Adjazenzmatrix
- Adjazenzfeld
- Adjazenzliste

Durchlaufen

- Tiefensuche
- Breitensuche

$\Rightarrow$ Kantenklassifikation

Graphen

Repräsentationen

- Kantenfolge
- Adjazenzmatrix
- Adjazenzfeld
- Adjazenzliste

Durchlaufen

- Tiefensuche
- Breitensuche

$\Rightarrow$ Kantenklassifikation

Graphen

Kürzeste Wege

Dijkstra Kantengewichte $\geqslant 0$, $O((m + n) \log n)$

Bellman-Ford Kantengewichte $\in \mathbb{R}$, erkennt neg. Zyklen, $O(n \cdot m)$

Minimale Spannbäume

- *Schnitteigenschaft:* **Leichteste** Kante in nem Schnitt: Nehmen!
- *Kreiseigenschaft:* **Schwerste** Kante in nem Kreis: Raus!

$\Rightarrow$ *Jarník-Prim:* Dijkstra-ähnlich – Aufspannen

$\Rightarrow$ *Kruskal:* Kanten von leicht $\rightsquigarrow$ schwer hinzufügen, wenn's geht

UnionFind (für Kruskal)

- Kleine Bäumchen repräsentieren **Knoten**mengen
- Pfadkompression
- Union-By-Rank

Graphen

Kürzeste Wege

Dijkstra Kantengewichte $\geqslant 0$, $O((m+n)\log n)$

Bellman-Ford Kantengewichte $\in \mathbb{R}$, erkennt neg. Zyklen, $O(n \cdot m)$

Minimale Spannbäume

- *Schnitteigenschaft*: **Leichteste** Kante in nem Schnitt: Nehmen!
- *Kreiseigenschaft*: **Schwerste** Kante in nem Kreis: Raus!

$\Rightarrow$ *Jarník-Prim*: Dijkstra-ähnlich – Aufspannen

$\Rightarrow$ *Kruskal*: Kanten von leicht $\rightsquigarrow$ schwer hinzufügen, wenn's geht

UnionFind (für Kruskal)

- Kleine Bäumchen repräsentieren **Knoten**mengen
- Pfadkompression
- Union-By-Rank

Graphen

Kürzeste Wege

Dijkstra Kantengewichte $\geqslant 0$, $O((m+n)\log n)$

Bellman-Ford Kantengewichte $\in \mathbb{R}$, erkennt neg. Zyklen, $O(n \cdot m)$

Minimale Spannbäume

- *Schnitteigenschaft*: **Leichteste** Kante in nem Schnitt: Nehmen!
- *Kreiseigenschaft*: **Schwerste** Kante in nem Kreis: Raus!

$\Rightarrow$ *Jarník-Prim*: Dijkstra-ähnlich – Aufspannen

$\Rightarrow$ *Kruskal*: Kanten von leicht $\rightsquigarrow$ schwer hinzufügen, wenn's geht

UnionFind (für Kruskal)

- Kleine Bäumchen repräsentieren **Knoten**mengen
- Pfadkompression
- Union-By-Rank

Optimierungsprobleme

Ansätze:

- Greedy
- DP

... (s. VL)

- ILPs

Aufgaben:
– Altklausur März 2015 A3 „MaxSubArray“;
– Münzproblem
– Altklausur März 2017 A5 „Chemie-ILP“

DANKE FÜRS DASEIN UND VIEL GLÜCK FÜR EURE KLAUSUREN! ☺

Ihr wart ein tolles Tut. ☺

Questions? $\rightsquigarrow$ Mail me!