Karlsruher Institut für Technologie

Algorithmen I
Tutorium 33

Woche 13 | 20. Juli 2018

Daniel Jungkind (daniel.jungkind@student.kit.edu)

INSTITUT FÜR THEORETISCHE INFORMATIK

Inhalt

Algorithmenanalyse

Master-Theorem / Amortisierte Analyse

Listen und Co.

Hashing

Sortieren

Binäre Heaps / Sortierte Folgen

Graphen

Optimierungsprobleme

Hint: Materialien von SS 2016/17

SS 16: http://crypto.iti.kit.edu/algo-sose16

SS 17: https://crypto.iti.kit.edu/index.php?id=799

Hier findet ihr auch die hier erwähnten Ü-Blätter, die alte Probeklausur etc.

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

Algorithmenanalyse

O-Kalkül

$o(f(n))$	$\prec$	echt schwächer wachsende Funktionen
$O(f(n))$	$\preccurlyeq$	schwächer oder gleich stark wachsende Funktionen
$\Theta(f(n))$	$\asymp$	genau gleich stark wachsende Funktionen
$\Omega(f(n))$	$\succcurlyeq$	stärker oder gleich stark wachsende Funktionen
$\omega(f(n))$	$\succ$	echt stärker wachsende Funktionen

Algorithmenanalyse

O-Kalkül: Formeln

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

Aufgaben: Z. B.: Altklausuren 2015 A1a, 2010 A1g/h, ...

Algorithmenanalyse

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

Invarianten

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

Algorithmenanalyse

Invarianten

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

Algorithmenanalyse

Invarianten

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

Algorithmenanalyse

Invarianten

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

Algorithmenanalyse

Invarianten

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

Algorithmenanalyse

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**: *next*$\rightarrow$*prev* $=$ *prev*$\rightarrow$*next* $=$ **this**

Aufgaben: Z. B.: Altklausur 2016_2 A5a,c, ...

Algorithmenanalyse

Aufgabe 1: Korrektheitsbeweis

Beweist die Korrektheit von *ArraySum*:

```
function ArraySum(A : array[1..n] of ℝ) : ℝ
    i := 1
    s := 0
    while i ≤ n do
        invariant ???

        s := s + A[i]
        i++
    return s
```

Algorithmenanalyse

Aufgabe 1: 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** $s = \sum_{j=1}^{i-1} A[j]$
  - $s := s + A[i]$
  - $i{+}{+}$
- **return** s

Algorithmenanalyse

Lösung zu Aufgabe 1

$\Longrightarrow$ Invariante:

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

Bezeichne s_i den Wert von s zu Beginn von Schleifendurchlauf i.

IA. $(i = 1)$: $\quad 0 = s_1 = \sum_{j=1}^{i-1} A[j] = \sum_{j=1}^{0} A[j] = 0. \quad \checkmark$

IV.: Die Invariante galt zu Beginn von Schleifendurchlauf i für festes i.

IS. $(i \rightsquigarrow i+1)$: Es gilt zu Beginn von Schleifendurchlauf $i+1$:

$$s_{i+1} = s_i + A[i] \overset{\text{IV}}{=} \sum_{j=1}^{i-1} A[j] + A[i] = \sum_{j=1}^{i} A[j]. \qquad \square$$

Nach dem n-ten Schleifendurchlauf gilt also $s = \sum_{j=1}^{n} A[j]$.

Algorithmenanalyse

Karlsruher Institut für Technologie

Lösung zu Aufgabe 1

$\Longrightarrow$ Invariante:

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

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

  $s := s + A[i]$

  $i{+}{+}$

Bezeichne s_i den Wert von s zu Beginn von Schleifendurchlauf i.

IA. ($i = 1$): $\quad 0 = s_1 = \sum_{j=1}^{i-1} A[j] = \sum_{j=1}^{0} A[j] = 0. \quad \checkmark$

IV.: Die Invariante galt zu Beginn von Schleifendurchlauf i für festes i.

IS. ($i \rightsquigarrow i+1$): Es gilt zu Beginn von Schleifendurchlauf $i+1$:

$$s_{i+1} = s_i + A[i] \overset{\text{IV}}{=} \sum_{j=1}^{i-1} A[j] + A[i] = \sum_{j=1}^{i} A[j]. \qquad \square$$

Nach dem n-ten Schleifendurchlauf gilt also $s = \sum_{j=1}^{n} A[j]$.

Algorithmenanalyse

Lösung zu Aufgabe 1

$\Longrightarrow$ Invariante:

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

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

  $s := s + A[i]$

  $i{+}{+}$

Bezeichne s_i den Wert von s zu Beginn von Schleifendurchlauf i.

IA. $(i = 1)$: $\quad 0 = s_1 = \sum_{j=1}^{i-1} A[j] = \sum_{j=1}^{0} A[j] = 0. \quad \checkmark$

IV.: Die Invariante galt zu Beginn von Schleifendurchlauf i für festes i.

IS. $(i \rightsquigarrow i+1)$: Es gilt zu Beginn von Schleifendurchlauf $i+1$:

$$s_{i+1} = s_i + A[i] \overset{\text{IV}}{=} \sum_{j=1}^{i-1} A[j] + A[i] = \sum_{j=1}^{i} A[j]. \qquad \square$$

Nach dem n-ten Schleifendurchlauf gilt also $s = \sum_{j=1}^{n} A[j]$.

Algorithmenanalyse

Lösung zu Aufgabe 1

$\Longrightarrow$ Terminierung:

- Zu Beginn ist $i = 1$
- Die Schleife läuft nur, solange $i \leqslant n$
- In jedem Durchlauf wird i um eins erhöht

$\Rightarrow$ Nach n Durchläufen terminiert die Schleife.

Algorithmenanalyse

SS 17 Blatt 1 Aufgabe 2 c)

function $\mathrm{f}(n, m : \mathbb{N}) : (\mathbb{N}_0, \mathbb{N}_0)$

$a = 0 : \mathbb{N}_0$
$b = m : \mathbb{N}_0$
$c = 1 : \mathbb{N}_0$
while $m - c \cdot n \geqslant 0$ **do**
    invariant ???
    $a := c$
    $c := c + 1$
    $b := m - a \cdot n$
return (a, b)

Invariante?

Algorithmenanalyse

Karlsruher Institut für Technologie

SS 17 Blatt 1 Aufgabe 2 c)

function $\mathrm{f}(n, m : \mathbb{N}) : (\mathbb{N}_0, \mathbb{N}_0)$

- $a = 0 : \mathbb{N}_0$
- $b = m : \mathbb{N}_0$
- $c = 1 : \mathbb{N}_0$
- **while** $m - c \cdot n \geqslant 0$ **do**
  - **invariant** $m = a \cdot n + b$
  - $a := c$
  - $c := c + 1$
  - $b := m - a \cdot n$
- **return** (a, b)

Invariante? Beweis?

Algorithmenanalyse

Lösung

while $m - c \cdot n \geqslant 0$ **do**

  invariant $m = a \cdot n + b$
  $a := c$
  $c := c + 1$
  $b := m - a \cdot n$

Bezeichne i die Nummer des aktuellen Schleifendurchlaufs und a_i, b_i, c_i den Wert von a, b, c zu Beginn von Schleifendurchlauf i.

IA. ($i = 1$): $a_1 \cdot n + b_1 = 0 \cdot n + m = m$. ✓

IV.: Die Invariante galt zu Beginn von Schleifendurchlauf i für festes i.

IS. ($i \rightsquigarrow i + 1$): Es gilt zu Beginn von Schleifendurchlauf $i + 1$:

$$a_{i+1} = c_i,$$
$$b_{i+1} = m - a_{i+1} \cdot n.$$
$$\implies a_{i+1} \cdot n + b_{i+1} = c_i \cdot n + (m - c_i \cdot n) = m. \quad \square$$

Algorithmenanalyse

Lösung

while $m - c \cdot n \geqslant 0$ **do**

 invariant $m = a \cdot n + b$

 $a := c$

 $c := c + 1$

 $b := m - a \cdot n$

Bezeichne i die Nummer des aktuellen Schleifendurchlaufs und a_i, b_i, c_i den Wert von a, b, c zu Beginn von Schleifendurchlauf i.

IA. ($i = 1$): $a_1 \cdot n + b_1 = 0 \cdot n + m = m.$ ✓

IV.: Die Invariante galt zu Beginn von Schleifendurchlauf i für festes i.

IS. ($i \rightsquigarrow i+1$): Es gilt zu Beginn von Schleifendurchlauf $i+1$:

$$a_{i+1} = c_i,$$
$$b_{i+1} = m - a_{i+1} \cdot n.$$
$$\implies a_{i+1} \cdot n + b_{i+1} = c_i \cdot n + (m - c_i \cdot n) = m. \quad \square$$

Algorithmenanalyse

Lösung

while $m - c \cdot n \geqslant 0$ **do**

  invariant $m = a \cdot n + b$

  $a := c$

  $c := c + 1$

  $b := m - a \cdot n$

Bezeichne i die Nummer des aktuellen Schleifendurchlaufs und a_i, b_i, c_i den Wert von a, b, c zu Beginn von Schleifendurchlauf i.

IA. ($i = 1$): $a_1 \cdot n + b_1 = 0 \cdot n + m = m.$ ✓

IV.: Die Invariante galt zu Beginn von Schleifendurchlauf i für festes i.

IS. ($i \rightsquigarrow i + 1$): Es gilt zu Beginn von Schleifendurchlauf $i + 1$:

$$a_{i+1} = c_i,$$
$$b_{i+1} = m - a_{i+1} \cdot n.$$
$$\Longrightarrow a_{i+1} \cdot n + b_{i+1} = c_i \cdot n + (m - c_i \cdot n) = m. \quad \square$$

Algorithmenanalyse

Lösung

```
while m − c · n ⩾ 0 do
    invariant m = a · n + b
    a := c
    c := c + 1
    b := m − a · n
```

Bezeichne i die Nummer des aktuellen Schleifendurchlaufs und a_i, b_i, c_i den Wert von a, b, c zu Beginn von Schleifendurchlauf i.

IA. ($i = 1$): $\quad a_1 \cdot n + b_1 = 0 \cdot n + m = m. \quad \checkmark$

IV.: Die Invariante galt zu Beginn von Schleifendurchlauf i für festes i.

IS. ($i \rightsquigarrow i + 1$): Es gilt zu Beginn von Schleifendurchlauf $i + 1$:

$$a_{i+1} = c_i,$$
$$b_{i+1} = m - a_{i+1} \cdot n.$$
$$\implies a_{i+1} \cdot n + b_{i+1} = c_i \cdot n + \left(m - c_i \cdot n\right) = m. \quad \square$$

Das Master-Theorem (einfache Form)

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

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

gegeben.

Dann gilt:

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

Amortisierte Analyse

How to

- **Aggregatmethode**: Schätze nach oben ab:
  $$\text{Gesamtkosten von } n \text{ beliebigen Ops} = „T_{\text{Gesamt}}“ \leqslant c \cdot n$$
  (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: SS 2016 Blatt 3 A4 „Weltherrschaftskonferenz“

Listen und Co.

Doppelt verkettete Liste

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

+ Flexibel, meiste Veränderungen in $\mathcal{O}(1)$

− Nicht cachefreundlich, Indexzugriff in $\mathcal{O}(n)$

Unbeschränktes Array

- Array voll $\Rightarrow$ Ziehe in **doppelt** so großes Array um
- Array viertel-voll $\Rightarrow$ Ziehe in **halb** so großes Array um

+ Cachefreundlich, Indexzugriff in $\mathcal{O}(1)$

− Eher unflexibel, viele Veränderungen in $\mathcal{O}(n)$

Z. B.: Altklausur 2016_2 A6.

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.

Karlsruher Institut für Technologie

Aufgabe 2
Entwerft einen Stack, der *push*, *pop* und *min* kann und zwar in $O(1)$ (nicht amortisiert).

Listen und Co.

Karlsruher Institut für Technologie

Lösung zu Aufgabe 2

$\mathit{BasicStack}, \mathit{MinimumStack} : \mathit{Stack}$

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

Aufgaben:
Konstruktion: Probeklausur ’17 A4, SS 2017 ÜB4 A1, Altklausuren 2015 A6b, 2010 A3, ...
Op-Folge angeben: Probeklausur ’17 A2b, Altklausur 2010 A6a, ...

Hashing

Hashing mit verketteten Listen

⇒ Halte **array of** Lists:
Werfe Element in die Liste, suche es dort

Hashing mit linearer Suche

⇒ Nur array of Element:
Platz besetzt? Gucke rechts davon.
Beim **Löschen**: Ggfs. wieder nach links zurückschieben, damit Lücken wieder zu!

- Ganz rechts im Array Platz dicht?
  ⇒ Pufferbereich (der dann hoffentlich langt) oder
  ⇒ 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, damit Lücken wieder zu!

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

Sortieralgorithmen

Vergleichsbasiert

- InsertionSort (Elemente blubbern einzeln nach unten, bis sie passen)
- (SelectionSort) (Wähle nächstes passendes Element aus)
- (BubbleSort) (Elemente blubbern einzeln nach oben, bis sie passen)
- Mergesort (Listen mehrmals halbieren, dann zurückmergen)
- Quicksort („Pivot-Vorsortierung“, Teile rekursiv weitersortieren)
- Heapsort (**ab**steigende Sortierung!) (baue Heap auf, n-mal *deleteMin*)

Ganzzahlig

- BucketSort (Elemente in passenden „Eimer“ werfen)
- CountingSort (Anzahl zählen statt reinzuwerfen)
- LSD-RadixSort (nach den einzelnen Ziffern sortieren)

Sorting Algorithms

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 **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

Karlsruher Institut für Technologie

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

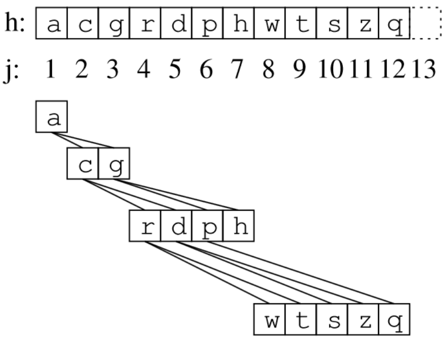
Implementierung

- Repräsentiere binären Baum als **array**$[1 \dots n]$ 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:

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

$$leftChild(j) = 2j$$

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

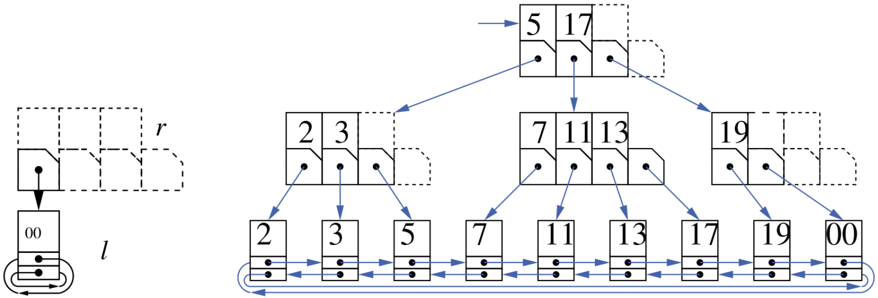


Binäre Heaps

- *insert*: Nach unten rechts und *siftUp*
- *deleteMin*: Element ganz unten rechts $\rightsquigarrow$ ganz oben, *siftDown*
- *buildHeap*: „Ebenen“ von unten $\rightsquigarrow$ oben durchgehen und „down-siften“, dass es passt 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

Aufgaben:
Konstruktion: Altklausur 2010 A1a, ...
BFS anwenden, sodass: Altklausur 2010 A1f, ...

Graphen

Repräsentationen

- Kantenfolge
- Adjazenzmatrix
- Adjazenzfeld
- Adjazenzliste

Durchlaufen

- Tiefensuche
- Breitensuche

$\Rightarrow$ Kantenklassifikation

Aufgaben:
Konstruktion: Altklausur 2010 A1a, ...
BFS anwenden, sodass: Altklausur 2010 A1f, ...

Graphen

Kürzeste Wege

Dijkstra	Kantengewichte $\geqslant 0$,	$O((m+n)\log n)$
Bellman-Ford	Kantengew. $\in \mathbb{R}$, erkennt neg. Zyklen,	$O(n \cdot m)$

Aufgaben: *Anwenden* (Altk. 2010 A2, 2013 A2a), *Theorie* (Altk. 2013 A2c), ...

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

Aufgaben: *Anwenden* (Altk. 2010 A5a, 2015 A2b, 2014 A6b), ...

Union-Find (für Kruskal)

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

Graphen

Kürzeste Wege

Dijkstra	Kantengewichte $\geqslant 0$,	$O((m+n)\log n)$
Bellman-Ford	Kantengew. $\in \mathbb{R}$, erkennt neg. Zyklen,	$O(n \cdot m)$

Aufgaben: *Anwenden* (Altk. 2010 A2, 2013 A2a), *Theorie* (Altk. 2013 A2c), ...

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

Aufgaben: *Anwenden* (Altk. 2010 A5a, 2015 A2b, 2014 A6b), ...

Union-Find (für Kruskal)

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

Graphen

Kürzeste Wege

Dijkstra	Kantengewichte $\geqslant 0$,	$O((m+n)\log n)$
Bellman-Ford	Kantengew. $\in \mathbb{R}$, erkennt neg. Zyklen,	$O(n \cdot m)$

Aufgaben: *Anwenden* (Altk. 2010 A2, 2013 A2a), *Theorie* (Altk. 2013 A2c), ...

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

Aufgaben: *Anwenden* (Altk. 2010 A5a, 2015 A2b, 2014 A6b), ...

Union-Find (für Kruskal)

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

Optimierungsprobleme

Karlsruher Institut für Technologie

Ansätze:

- Greedy
- DP

... (s. VL)

- ILPs

Aufgaben:

- Altklausur 2014_2 A3 „MaxSubArray";
- Münzproblem (SS 2016 Übung 12 Folie 5 http://crypto.iti.kit.edu/fileadmin/User/Lectures/Algorithmen_SS16/ue12-slides.pdf)
- Altklausur 2016_2 A5 „Chemie-ILP"

DANKE FÜRS DASEIN UND VIEL GLÜCK FÜR EURE KLAUSUREN! ☺

Ihr wart ein tolles Tut. ☺

Questions? $\rightsquigarrow$ E-Mail / ILIAS...