



Algorithmen und Datenstrukturen

Zusammenfassung

Matthias Teschner
Graphische Datenverarbeitung
Institut für Informatik
Universität Freiburg

SS 09

Überblick



- Eigenschaften von Algorithmen
- Entwurfstechniken
- Datenstrukturen
- Sortieren
- Suchen
- Hashverfahren
- Sonstiges

Algorithmus



- wohldefinierte **Rechenvorschrift**, die eine Menge von Elementen als **Eingabe** verwendet und eine Menge von Elementen als **Ausgabe** erzeugt
- beschreibt eine Rechenvorschrift zum Erhalt einer durch die Formulierung eines Problems gegebenen **Eingabe-Ausgabe-Beziehung**

Algorithmus f : Eingabe \rightarrow Ausgabe

Eigenschaften von Algorithmen



- **Korrektheit**
 - Ein korrekter Algorithmus stoppt (terminiert) für jede Eingabeinstanz mit der durch die Eingabe-Ausgabe-Relation definierten Ausgabe.
 - Ein inkorrekt Algorithmus stoppt nicht oder stoppt mit einer nicht durch die Eingabe-Ausgabe-Relation vorgegebenen Ausgabe.
- **Effizienz**
 - Bedarf an Speicherplatz und Rechenzeit
 - Wachstum (Wachstumsgrad, Wachstumsrate) der Rechenzeit bei steigender Anzahl der Eingabe-Elemente (Laufzeitkomplexität)

Korrektheit



- Ein Programm S ist bezüglich einer **Vorbedingung** P und einer **Nachbedingung** Q **partiell korrekt**, wenn für jede Eingabe, die P erfüllt, das Ergebnis auch Q erfüllt, falls das Programm terminiert.
- Ein Programm ist **total korrekt**, wenn es partiell korrekt ist und für jede Eingabe, die P erfüllt, terminiert.
- Beispiel:
 - Vorbedingung: gültige Werte für Divisor und Dividend
 - Nachbedingung: Programm liefert den ganzzahligen Rest
- Wir beschränken uns auf partielle Korrektheit.

Hoare-Kalkül



- formales System zum Beweisen der Korrektheit imperativer Programme
- von Sir Charles Antony Richard Hoare (1969) entwickelt, mit Beiträgen von Lauer (1971) und Dijkstra (1982)
- besteht aus Axiomen und Ableitungsregeln für alle Konstrukte einer einfachen imperativen Programmiersprache
 - elementare Operationen
 - sequenzielle Ausführung
 - bedingte Ausführung
 - Schleife

Hoare-Tripel



- zentrales Element des Hoare-Kalküls
- beschreibt die Zustandsveränderung durch eine Berechnung

$$\{P\} S \{Q\}$$

- P und Q
 - sind Zusicherungen (Vor- und Nachbedingung).
 - sind Formeln der Prädikatenlogik.
- S ist ein Programmsegment.
- Wenn Programmzustand P vor der Ausführung von S gilt, dann gilt Q nach der Ausführung von S.
- Beispiel: $\{x + 1 = 43\} y := x + 1 \{y = 43\}$

Prinzip der Verifikation



$$\{P\} S_1 \{A_1\} S_2 \{A_{n-1}\} S_n \{Q\}$$

- $\{P\}$ und $\{Q\}$
 - sind Vor- und Nachbedingung des Programms S_1, S_2, \dots, S_n
 - stellen die Spezifikation der Ein- / Ausgabe-Relation dar
 - sind definierte, bekannte Zusicherungen
- A_1, A_2, \dots, A_{n-1}
 - sind unbekannte Zusicherungen
 - werden durch das Hoare-Kalkül bestimmt
 - beispielsweise wird A_{n-1} über eine Hoare-Regel für das Hoare-Tripel $\{A_{n-1}\} S_n \{Q\}$ mit bekanntem S_n und bekanntem $\{Q\}$ bestimmt
- Tritt dabei kein Widerspruch auf, ist S_1, S_2, \dots, S_n partiell korrekt bezüglich $\{P\}$ und $\{Q\}$.

Asymptotische Analyse



- Methode zur Einschätzung des Grenzverhaltens einer Funktion (bei uns Laufzeitverhalten für wachsende Eingabegrößen)
- Fokus auf den wesentlichen Trend des Grenzverhaltens
- ermöglicht die Einordnung von Funktionen in **Funktionsklassen**
- Beispielsweise können die **Landau-Symbole** Θ , O , Ω verwendet werden, um Klassen von Funktionen zu beschreiben.
- Funktionsklassen werden zur Beschreibung von Laufzeit bzw. Komplexität von Algorithmen verwendet (**asymptotische Effizienz eines Algorithmus**).

O-Notation



$$f \in \Theta(g) : \exists c_1 > 0 \quad \exists c_2 > 0 \quad \exists n_0 \quad \forall n > n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$$f \in O(g) : \exists c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \leq c \cdot g(n)$$

$$f \in \Omega(g) : \exists c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \geq c \cdot g(n)$$

$$f \in o(g) : \forall c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \leq c \cdot g(n)$$

$$f \in \omega(g) : \forall c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \geq c \cdot g(n)$$

O-Kalkül



- Einfache Regeln

$$f = O(f)$$

$$O(O(f)) = O(f)$$

$$kO(f) = O(f)$$

$$O(f + k) = O(f)$$

- Addition

$$O(f) + O(g) = O(\max\{f, g\})$$

- Multiplikation

$$O(f) \cdot O(g) = O(f \cdot g)$$

Praktische Relevanz der Funktionenklassen



- Rechner B 10fach schneller als Rechner A

Maximale Problemgröße auf Rechner B bei gegebener Problemgröße p auf Rechner A

Laufzeit / Komplexität	Problemgröße auf Rechner B
n	$10 p$
n^2	$3.16 p$
n^3	$2.15 p$
2^n	$p + 3.32 = p + \log_2 10$

$$10 \cdot 2^p = 2^{\log_2 10} \cdot 2^p = 2^{p+\log_2 10}$$

- Polynomielle Laufzeiten erlauben die Bearbeitung einer um einen Faktor größeren Problemgröße bei schnellerer Hard- / Software. Exponentielle Laufzeiten nicht.

Laufzeitkomplexität



for i:=0 to n-1 do	O(n)		O(n) · O(i) = O(n ²)
begin		O(n)	
s:=0;	O(1)		
for j:=0 to i do	O(i+1)	O(i)	
s:=s+x[j];	O(3)		
a[i]:=s/(i+1);	O(4)	O(1)	
end;			

i ist von n abhängig, kann nicht als konstant angesehen werden.

- Wie oft werden Anweisungen in der inneren Schleife in Abhängigkeit von der Problemgröße n ausgeführt?

$$1 + 2 + \dots + n = \frac{n(n+1)}{2} = O(n^2)$$

Gaußsche Summenformel

Verschiedene Laufzeiten



- Laufzeit hängt nicht immer ausschließlich von der Größe des Problems ab, sondern auch von der Beschaffenheit der Eingabemenge
- Daraus ergeben sich
 - **beste Laufzeit**
beste Laufzeitkomplexität für eine Eingabeinstanz der Größe n
 - **schlechteste Laufzeit**
schlechteste Laufzeitkomplexität für eine Instanz der Größe n
 - **mittlere oder erwartete Laufzeit**
gemittelte Laufzeitkomplexität für alle Eingabeinstanzen der Größe n

Rekursionsgleichungen



- beschreiben die Laufzeit bei Rekursionen

$$T(n) = \begin{cases} \overset{\text{Trivialfall für } n_0}{f_0(n)} & n = n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}$$

Lösung von a
Teilproblemen
mit reduziertem
Aufwand n/b

Verbinden der
Teillösungen

- n_0 ist üblicherweise klein, oft ist $f_0(n_0) \in \Theta(1)$
- üblicherweise $a > 1$ und $b > 1$
- Je nach Lösungstechnik wird f_0 vernachlässigt.
- T ist nur für ganzzahlige n/b definiert, was auch gern bei der Lösung vernachlässigt wird.

Substitutionsmethode



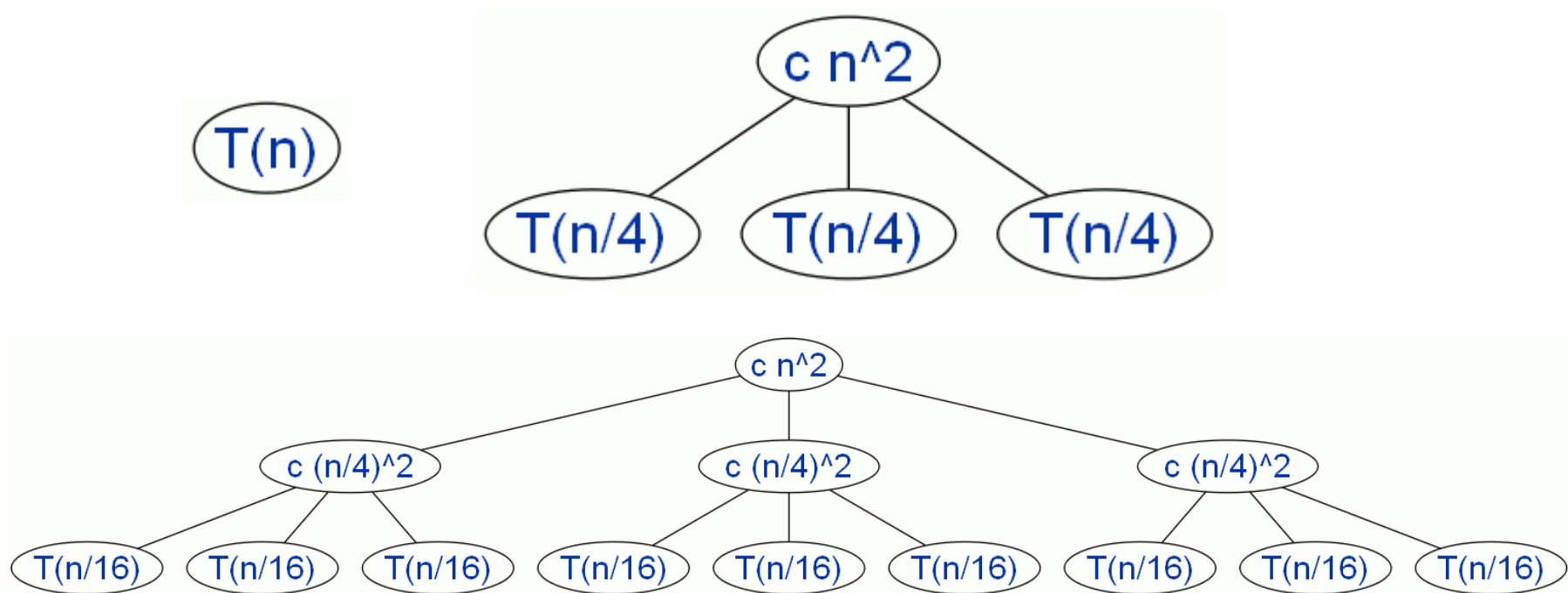
- Lösung raten und mit Induktion beweisen
- Beispiel: $T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$
- Vermutung: $T(n) = n + n \log n$
- Induktionsanfang für $n=1$: $T(1) = 1 + 1 \log 1 = 1$
- Induktionsschritt von $n/2$ nach n :

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\ &= 2 \cdot \left(\frac{n}{2} + \frac{n}{2} \log \frac{n}{2}\right) + n && \text{Induktionsvoraussetzung} \\ &= 2 \cdot \left(\frac{n}{2} + \frac{n}{2} (\log n - 1)\right) + n \\ &= n + n \log n - n + n \\ &= n + n \log n \end{aligned}$$

Rekursionsbaum-Methode



- kann zum Aufstellen von Vermutungen verwendet werden
- Beispiel: $T(n) = 3 T(n/4) + \Theta(n^2) \leq 3 T(n/4) + cn^2$



Mastertheorem



$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- Fall 1: $T(n) \in \Theta\left(n^{\log_b a}\right)$ falls $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$ $\epsilon > 0$
- Fall 2: $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$ falls $f(n) \in \Theta\left(n^{\log_b a}\right)$
- Fall 3: $T(n) \in \Theta\left(f(n)\right)$ falls $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ $\epsilon > 0$
$$af\left(\frac{n}{b}\right) \leq cf(n) \quad 0 < c < 1$$

$$n > n_0$$

Überblick



- Eigenschaften von Algorithmen
- Entwurfstechniken
- Datenstrukturen
- Sortieren
- Suchen
- Hashverfahren
- Sonstiges

Teile und Herrsche



- **Teile** das Gesamtproblem in kleinere Teilprobleme auf.
- **Herrsche** über die Teilprobleme durch rekursives Lösen.
Wenn die Teilprobleme klein sind, löse sie direkt.
- **Verbinde** die Lösungen der Teilprobleme zur Lösung des Gesamtproblems.

- **rekursive** Anwendung des Algorithmus auf immer kleiner werdende Teilprobleme
- **direkte** Lösung eines hinreichend kleinen Teilproblems

Backtracking



- Versuch-und-Irrtum-Prinzip (trial and error)
- Erreichte **Teillösung wird** schrittweise zur Gesamtlösung **ausgebaut**.
- Bewertungsfunktion prüft Gültigkeit eines Schrittes.
- Wenn klar ist, daß Teillösung nicht zur Gesamtlösung führt, **können Schritte rückgängig gemacht werden** und Alternativen getestet werden.
- Tiefensuche
- oft rekursiv implementiert

Greedy



- Wähle **schrittweise** den Folgezustand, der aktuell (lokal) den **größten Zugewinn** verspricht.
- Eine **Bewertungsfunktion** muss existieren, um aus möglichen Folgezuständen den optimalen zu ermitteln.
- **Schritte werden nicht rückgängig gemacht.**
- einfach zu realisieren
- löst viele Probleme nicht optimal (lokales Optimum statt globales Optimum)
- löst eventuell Probleme nicht in optimaler Zahl von Schritten, z. B. Gradientenabstiegsverfahren im Vergleich zu konjugierten Gradienten

Dynamische Programmierung



- berechne optimale Lösungen kleiner Teilprobleme direkt
- setze Lösungen für nächstgrößere Teilprobleme aus bekannten Lösungen für kleinere Probleme zusammen
- **berechnete Teilergebnisse werden gespeichert (tabelliert) und bei Bedarf wiederverwendet**
- wird bei Problemen angewendet, die aus **vielen gleichartigen Teilproblemen** bestehen
- **Rekursion wird** durch Wiederverwendung von bereits berechneten Teilproblemen **vermieden** (Rekursion berechnet eventuell gleiche Teilprobleme mehrmals.)

Vollständige Aufzählung



- Systematische Erzeugung aller Lösungskandidaten
- Auswahl des optimalen Lösungskandidaten anhand einer Zielfunktion (Kostenfunktion)
- meist schlechte Effizienz

Sweep-Verfahren



- Raum mit allen Elementen des Problems wird ausgefegt bzw. abgetastet (sweep).
- Eine Status-Struktur wird während des Abtastens aller Elemente mitgeführt und aktualisiert (Sweep-Status-Struktur).
- In der Regel wird das Problem mit einer (n-1)-dimensionalen Struktur abgetastet.
 - 1D-Feld wird durch Punkt abgetastet (ausgefegt).
 - 2D-Ebene wird durch 1D-Gerade abgetastet.
 - 3D-Raum wird durch 2D-Ebene abgetastet.
- Wenn alle Elemente des Problems besucht (abgetastet) wurden, kann aus der Sweep-Status-Struktur die Lösung ermittelt werden.

Überblick



- Eigenschaften von Algorithmen
- Entwurfstechniken
- **Datenstrukturen**
- Sortieren
- Suchen
- Hashverfahren
- Sonstiges

Implementierung von Mengen



- Implementierungen von Datenstrukturen sind durch unterschiedliche Laufzeiten für verschiedene Operationen charakterisiert.
- **statische Datenstrukturen**
 - Feld

Die Größe eines Feldes kann während der Laufzeit eines Programms nicht verändert werden.
- **dynamische Datenstrukturen**
 - Liste, verkettet oder doppelt verkettet
 - Baum
 - Graph

Beispiele



- **Feld**
 - Zugriff auf ein Element über einen Index
- **Liste**
 - Element besitzt Verweis auf das folgende Element
- **Stapel**
 - Elemente können nur in umgekehrter Reihenfolge des Einfügens gelesen oder gelöscht werden
- **Warteschlange**
 - Elemente können nur in gleicher Reihenfolge des Einfügens gelesen oder gelöscht werden
- **Graphen, Bäume**
 - Elemente besitzen variable Anzahl von Verweisen auf weitere Elemente

Max-Heap / Min-Heap



- Folge von Daten ist ein Max-Heap, wenn

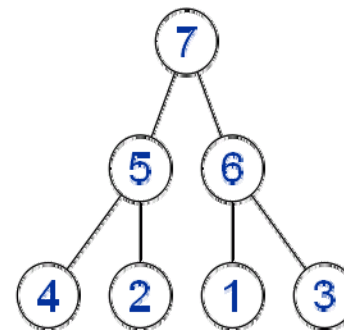
äquivalente
Aussagen

- die Schlüssel beider Nachfolgeknoten eines jeden Knotens k kleiner sind als der Schlüssel von Knoten k
- der Schlüssel des Feldes an Stelle k kleiner ist als die Schlüssel an den Stellen $2k$ und $2k+1$
- über die Relationen anderer Knoten ist nichts bekannt
- aus den Anforderungen ergibt sich, dass der Wurzelknoten den größten Schlüssel enthält

Beispiel: Folge ist ein Max-Heap

7	5	6	4	2	1	3
---	---	---	---	---	---	---

Feldrepräsentation



Baumrepräsentation

Baum



- verallgemeinerte Listen
 - Elemente können mehr als nur einen Nachfolger haben
- spezieller Graph
 - Graph G besteht aus **Knoten** V , die durch **Kanten** E verbunden sind $G = (V, E)$
 - Kanten sind **gerichtet** oder **ungerichtet**
 - Zahl der Knoten: $|V| = n$, Zahl der Kanten: $|E| = m$
 - G ist ein Baum
 - gdw. zwischen je zwei Knoten genau ein Weg existiert
 - gdw. G zusammenhängend ist und $m=n-1$
 - gdw. G keinen Zyklus enthält und $m=n-1$

Baum

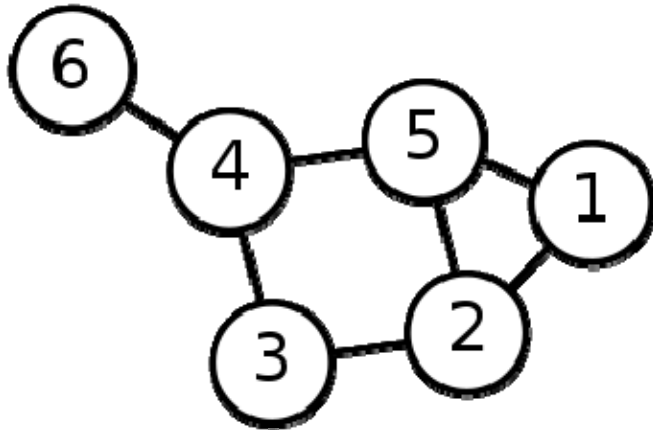


- Wurzel, innerer Knoten, Blatt, ...
- Suchbaum / Blattsuchbaum
- Suchbaum-Eigenschaft
 - sortierte Ausgabe
 - min / max
 - Einfügen, Löschen
- AVL-Baum
 - Definition
 - Einfügen / Rotation

Graph



- Graph G ist ein Paar zweier Mengen $G = (V, E)$ mit
 - $V = \{v_0, \dots, v_{n-1}\}$ Menge von n unterschiedlichen Knoten (Vertices / Vertex)
 - $E = \{e_0, \dots, e_{m-1}\}$ Menge von m Kanten mit $e_i = (v_j, v_k)$ (Edge, Arc)



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{ (1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5), (4, 6) \}$$

Wikipedia: Graph (Graphentheorie)

Graph



- **Eigenschaften**
 - Nachbarschaft, Grad
- **Repräsentation**
 - Adjazenzmatrix
 - Adjazenzliste
- **Traversierung**
 - Tiefensuche mit Stack
 - Breitensuche mit Schlange
- **Kürzester Weg**
 - Breitensuche in ungewichteten Graphen
 - Dijkstra in gewichteten Graphen

Überblick



- Eigenschaften von Algorithmen
- Entwurfstechniken
- Datenstrukturen
- **Sortieren**
- Suchen
- Hashverfahren
- Sonstiges

Verfahren



Verfahren	bester Fall $O(x)$	mittlerer Fall $O(x)$	schlechtester Fall $O(x)$	stabil	rekursiv	Speicher $O(x)$
Bubble	n	n^2	n^2	ja	nein	1
Selection	n^2	n^2	n^2	ja	nein	1
Insertion	n	n^2	n^2	ja	nein	1
Heap	$n \log n$	$n \log n$	$n \log n$	nein	nein	1
Quick	$n \log n$	$n \log n$	n^2	nein	ja	1
Merge	$n \log n$	$n \log n$	$n \log n$	ja	ja	n
Counting	n	n	n	ja	nein	n
Radix	n	n	n	ja	nein	n
Bucket	n	n	n	ja	nein	n

Abschätzung der Tiefe des Entscheidungsbaums



- in Entscheidungsbaum mit $n!$ Blättern ist die mittlere und die maximale Tiefe eines Blattes bestenfalls $\log n!$

$$\begin{aligned}\log n! &= \log (1 \cdot 2 \cdot \dots \cdot n - 1 \cdot n) \\ &= \log (1 \cdot 2 \cdot \dots \cdot \frac{n-1}{2} \cdot \frac{n}{2} \cdot \dots \cdot n - 1 \cdot n) \\ &\geq \log \left(\frac{n}{2} \cdot \dots \cdot n - 1 \cdot n \right) \\ &\geq \log \left(\frac{n}{2} \right)^{\frac{n}{2}} \\ &= \frac{n}{2} \log (n) - 1 \\ &\in \Omega(n \log n)\end{aligned}$$

- untere Schranke für vergleichsbasiertes Sortieren
 $\Omega(n \log n)$

Überblick



- Eigenschaften von Algorithmen
- Entwurfstechniken
- Datenstrukturen
- Sortieren
- **Suchen**
- Hashverfahren
- Sonstiges

Verfahren



- **Sequentielle / lineare Suche**
 - Feld muss nicht sortiert sein, Laufzeit $O(n)$
- **Binäre Suche**
 - Feld muss sortiert sein, Laufzeit $O(\log n)$
- **Exponentielle Suche**
 - sinnvoll, wenn Schlüssel k klein gegenüber Anzahl der Elemente n
 - Feld ist sortiert, Laufzeit $O(\log k)$
- **Interpolationssuche**
 - Feld ist sortiert, vermutet lineares Verhalten der Schlüsselwerte
 - Laufzeit $O(1)$, wenn die Vermutung stimmt, sonst $O(\log(\log(n)))$
- **i -kleinstes Element**
 - naiv $O(n^2)$, min-Heap $O(n \log n)$,
 - basierend auf Quick-Sort bei guter Pivotisierung $O(n)$

Überblick

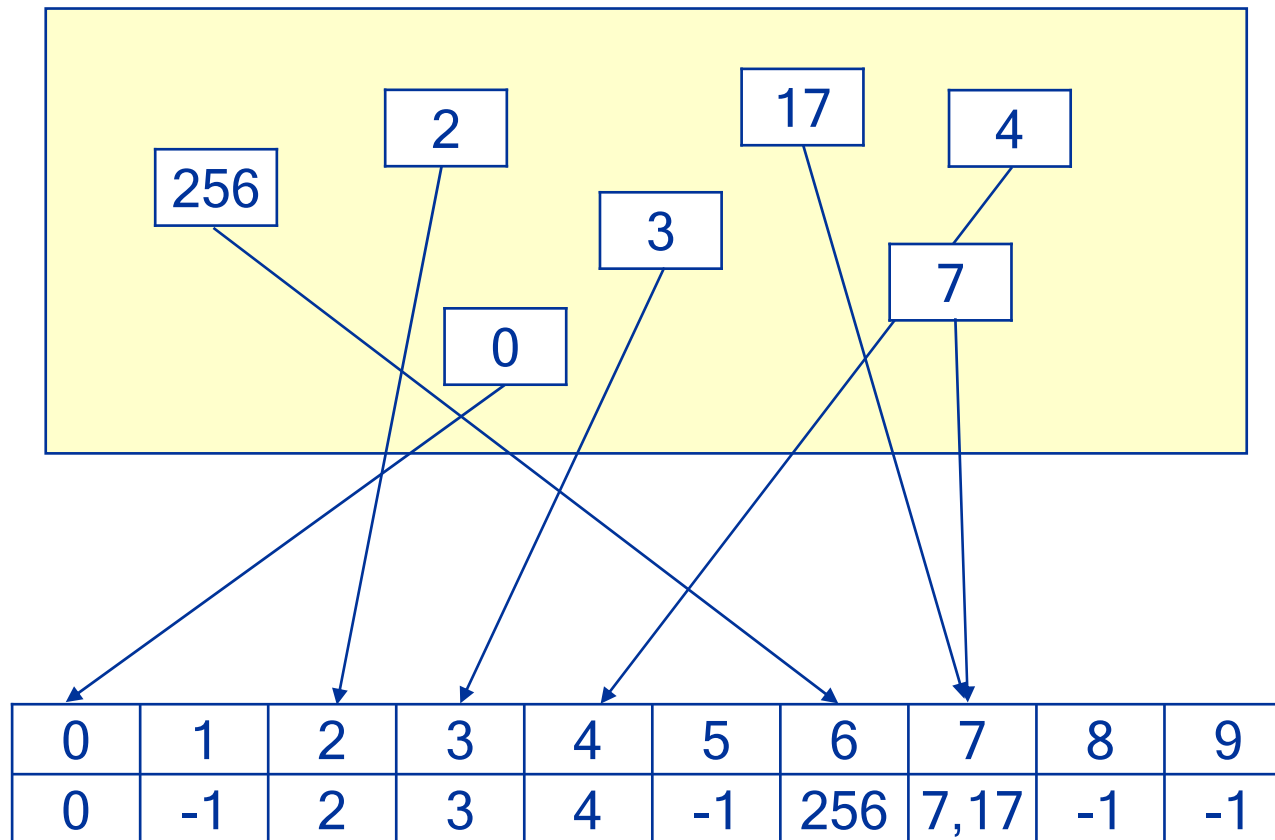


- Eigenschaften von Algorithmen
- Entwurfstechniken
- Datenstrukturen
- Sortieren
- Suchen
- Hashverfahren
- Sonstiges

Hashverfahren / Hashing



Menge mit $n=7$ Elementen



Hashtabelle t der Größe $m=10$

Hashfunktion:

$$h(s) = s \bmod 10$$

Hashwerte / Hashindices:

$$h(256) = 6$$

$$h(2) = 2$$

$$h(0) = 0$$

$$h(3) = 3$$

$$h(17) = 7$$

$$h(7) = 7$$

$$h(4) = 4$$

Hashkollision:

$$h(7) = h(17)$$

Kollisionsbehandlung



- Verkettung (dynamisch, Zahl der Elemente variabel)
 - kollidierende Schlüssel werden in Liste gespeichert
- Offene Hashverfahren (statisch, Zahl der Elemente fest)
 - Bestimmung einer Ausweichsequenz (Sondierungsreihenfolge), Permutation aller Hashwerte (Indizes der Hashtabelle)
 - lineares, quadratisches Sondieren: einfach, führt zu Häufungen, da Sondierungsreihenfolge vom Schlüssel unabhängig
 - uniformes Sondieren, double hashing: unterschiedliche Sondierungsreihenfolgen für unterschiedliche Schlüssel, vermeidet Häufungen von Elementen
- Effizienzsteigerung der Suche durch Umsortieren von Elementen beim Einfügen (Brent, Ordered Hashing)

Überblick



- Eigenschaften von Algorithmen
- Entwurfstechniken
- Datenstrukturen
- Sortieren
- Suchen
- Hashverfahren
- **Sonstiges**

Sonstiges



- Fibonacci-Zahlen, spezielle Summen, Rechenregeln
- Polynomprodukt / Matrixmultiplikation
- Topologische Sortierung
- Auswertung von Klammersausdrücken