Algorithmen und Datenstrukturen 2

Sommersemester 2007
1. Vorlesung

Peter Stadler

Universität Leipzig
Institut für Informatik
studla@bioinf.uni-leipzig.de

Zur Vorlesung allgemein

Vorlesungsumfang: 2 + 1 SWS

Übungen

- Durchführung in zweiwöchentlichem Abstand
- selbständige Lösung der Übungsaufgaben wesentlich für Lernerfolg
- Übungsblätter im Netz unter

http://www.bioinf.uni-leipzig.de/Leere/SS07/ADS2/index.html

Leistungsbewertung

- Klausur am **17.7.07** um 15.15 Uhr über ADS2
- Klausur für alle, außer für diejenigen, die nur einen Übungsschein benötigen
- Voraussetzung: erfolgreiche Übungsbearbeitung ADS2, d.h. 60% der erreichbaren Punkte in den Übungsserien.

Übungsaufgaben und Übungsgruppen

Ausgabe: 1. Serie: in KW 15 unter

http://www.bioinf.uni-leipzig.de/Leere/SS07/ADS2/index.html;

danach 2-wöchentlich

Abgabe:

vor der Vorlesung am Dienstag der übernächsten Woche, d.h. 15:00 Uhr im Gr. Hörsaal, CLI

Beginn Übungsbetrieb: KW 16

Einschreibung für die Übungsgruppen: ab 3.4.2007 unter

http://www.bioinf.uni-leipzig.de/Leere/SS07/ADS2/index.html

Alle bisher erfolgten Einschreibungen sind hinfällig!!!

Übungsgruppen

- **Übungsgruppen für 2. Semester:**

Gruppe 1A:	Mo 9:15 Uhr	A-Woche	Brühl, R508
Gruppe 1B:	Mo 9:15 Uhr	B-Woche	Brühl, R508
Gruppe 2A:	Di 13:15 Uhr	A-Woche	Härtelstr. 16/18, R109
Gruppe 2B:	Di 13:15 Uhr	B-Woche	Härtelstr. 16/18, R109
Gruppe 3A:	Di 11:15 Uhr	A-Woche	Härtelstr. 16/18, R109

- **Übungsgruppen für 4. Semester:**

Gruppe 4A:	Mo 11:15 Uhr	A-Woche	Brühl, R730
Gruppe 4B:	Mo 11:15 Uhr	B-Woche	Brühl, R730
Gruppe 5A:	Di 13:15 Uhr	A-Woche	Brühl, R623
Gruppe 5B:	Di 13:15 Uhr	B-Woche	Brühl, R623
Gruppe 6B:	Di 11:15 Uhr	B-Woche	Härtelstr. 16/18, R109

Ansprechpartner

Übungsleiter:

- Dr. Konstantin Klemm klemm@bioinf.uni-leipzig.de
- Dr. Stephan Steigele steigele@bioinf.uni-leipzig.de
- Christian Heine cheine@informatik.uni-leipzig.de

Studentische Hilfskräfte:

- Stephanie Keller
- Michael Kunze
- Stephanie Kehr
- Claudia Römer
- Michael Becker

Vorläufiges Inhaltsverzeichnis

Literatur

T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, Reihe Informatik, Band 70, BI-Wissenschaftsverlag, 3. Auflage, Spektrum-Verlag, 1996

M.A. Weiss: Data Structures & Algorithm Analysis in Java. Addison-Wesley 1999, 2. Auflage 2002

Weitere Bücher

V. Claus, A. Schwill: Duden Informatik, BI-Dudenverlag, 2. Auflage 1993

D.A. Knuth: The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973

R. Sedgewick: Algorithmen. Addison-Wesley 1992

G. Saake, K. Sattler: Algorithmen und Datenstrukturen - Eine Einführung mit Java. dpunkt-Verlag, 2002

A. Solymosi, U. Gude: Grundkurs Algorithmen und Datenstrukturen. Eine Einführung in die praktische Informatik mit Java. Vieweg, 2000, 2. Auflage 2001

Hashing

Einführung
Hash-Funktionen
- Divisionsrest-Verfahren
- Faltung
- Mid-Square-Methode, . . .

Behandlung von Kollisionen
- Verkettung der Überläufer
- Offene Hash-Verfahren: lineares Sondieren, quadratisches Sondieren, ...

Analyse des Hashing
Hashing auf Externspeichern
- Bucket-Adressierung mit separaten Überlauf-Buckets
- Analyse

Dynamische Hash-Verfahren
- Erweiterbares Hashing
- Lineares Hashing

Hashing: Einführung I

Gibt es bessere Strukturen für direkte Suche für Haupt- und Externspeicher ?

- AVL-Baum: $O(\log_2 n)$ Vergleiche
- B*-Baum: E/A-Kosten $O(\log_k \cdot (n))$, vielfach 3 Zugriffe

Bisher:

- Suche über Schlüsselvergleich
- Allokation des Satzes als physischer Nachbar des "Vorgängers" oder beliebige Allokation und Verknüpfung durch Zeiger

Gestreute Speicherungsstrukturen / Hashing (Schlüsseltransformation, Adreßberechnungsverfahren, scatter-storage technique usw.)

- Berechnung der Satzadresse SA(i) aus Satzschlüssel K_i --> Schlüsseltransformation
- Speicherung des Satzes bei SA(i)
- Ziele: schnelle direkte Suche + Gleichverteilung der Sätze (möglichst wenig Synonyme)

Hashing: Einführung II

Definition:

S sei Menge aller möglichen Schlüsselwerte eines Satztyps (Schlüsselraum)

A = {0, 1, ..., m-1} sei Intervall der ganzen Zahlen von 0 bis m-1 zur Adressierung eines Arrays bzw. einer Hash-Tabelle mit m Einträgen

Eine Hash-Funktion $h : S \rightarrow A$ ordnet jedem Schlüssel $s \in S$ des Satztyps eine Zahl aus A als Adresse in der Hash-Tabelle zu.

Idealfall:

1 Zugriff zur direkten Suche

Problem: Kollisionen

Beispiel: m=10

$h(s) = s \bmod 10$

	Schlüssel	Daten
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Perfektes Hashing: Direkte Adressierung

Idealfall (perfektes Hashing): keine Kollisionen

- h ist eine injektive Funktion.
- Für jeden Schlüssel aus S muss Speicherplatz bereitgehalten werden, d. h., die Menge aller möglichen Schlüssel ist bekannt.

Parameter

l = Schlüssellänge, b = Basis, m = #Speicherplätze

n_p = #S = b^l mögliche Schlüssel

n_a = #K = # vorhandene Schlüssel

Wenn K bekannt ist und K fest bleibt, kann leicht eine injektive Abbildung

$$h: K \rightarrow \{0, \ldots, m-1\}$$

z. B. wie folgt berechnet werden:

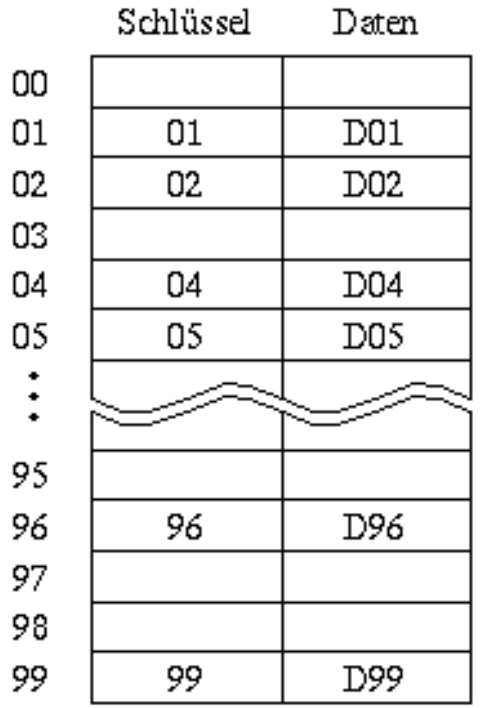
- Die Schlüssel in K werden lexikographisch geordnet und auf ihre Ordnungsnummern abgebildet oder
- Der Wert eines Schlüssels K_i oder eine einfache ordnungserhaltende Transformation dieses Wertes (Division/Multiplikation mit einer Konstanten) ergibt die Adresse: $A_i = h(K_i) = K_i$

Beispiel: Direkte Adressierung

Beispiel: Schlüsselmenge {00, . . . , 99}

Eigenschaften

- Statische Zuordnung des Speicherplatzes
- Kosten für direkte Suche und Wartung ?
- Reihenfolge beim sequentiellen Durchlauf ?



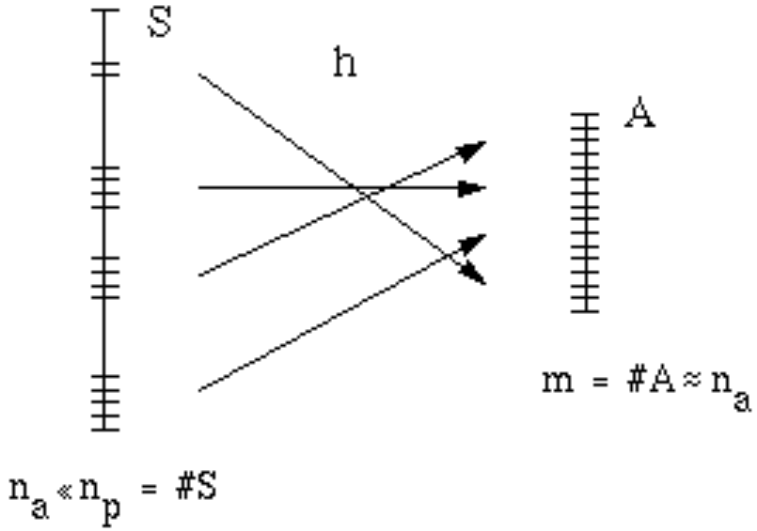
Bestes Verfahren bei geeigneter Schlüsselmenge K, aber aktuelle Schlüsselmenge K ist oft nicht "dicht":

- eine 9-stellige Sozialversicherungsnummer bei 10^5 Beschäftigten
- Namen / Bezeichner als Schlüssel (Schlüssellänge k): Faktor 10^4

Allgemeines Hashing

Annahmen

- Die Menge der möglichen Schlüssel ist meist sehr viel größer als die Menge der verfügbaren Speicheradressen
- h ist nicht injektiv



Definitionen:

- Zwei Schlüssel K_i, K_j kollidieren (bzgl. einer Hash-Funktion h) gdw. $h(K_i) = h(K_j)$.
- Tritt für K_i und K_j eine Kollision auf, so heißen diese Schlüssel Synonyme.
- Die Menge der Synonyme bezüglich einer Speicheradresse A_i heißt Kollisionsklasse.

Geburtstags-Paradoxon I

k Personen auf einer Party haben gleichverteilte und stochastisch unabhängige Geburtstage. Mit welcher Wahrscheinlichkeit p(n, k) haben mindestens 2 von k Personen am gleichen Tag (n = 365) Geburtstag?

Die Wahrscheinlichkeit, dass keine Kollision auftritt, ist

$$q(n,k) = \frac{\textit{Zahl der günstigen Fälle}}{\textit{Zahl der möglichen Fälle}} = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-k+1}{n} = \frac{(n-1)\ldots(n-k+1)}{n^{k-1}}$$

Es ist p (365, k) = 1 - q (365, k) > 0.5 für k>22 (s. nächste Folie)

ALSO: Behandlung von Kollisionen erforderlich !

Geburtstags-Paradoxon II

Wahrscheinlichkeit p, dass keine 2 von k Personen am gleichen Tag Geburtstag haben

k	p	k	p	k	p	k	p
1	1.00	15	0.75	29	0.32	43	0.08
2	1.00	16	0.72	30	0.29	44	0.07
3	0.99	17	0.68	31	0.27	45	0.06
4	0.98	18	0.65	32	0.25	46	0.05
5	0.97	19	0.62	33	0.23	47	0.05
6	0.96	20	0.59	34	0.20	48	0.04
7	0.94	21	0.56	35	0.19	49	0.03
8	0.93	22	0.52	36	0.17	50	0.03
9	0.91	23	0.49	37	0.15	51	0.03
10	0.88	24	0.46	38	0.14	52	0.02
11	0.86	25	0.43	39	0.12	53	0.02
12	0.83	26	0.40	40	0.11	54	0.02
13	0.81	27	0.37	41	0.10	55	0.01
14	0.78	28	0.35	42	0.09	56	0.01

Hash-Verfahren: Einflußfaktoren I

Leistungsfähigkeit eines Hash-Verfahrens: Einflußgrößen und Parameter

- Hash-Funktion
- Datentyp des Schlüsselraumes: Integer, String, ...
- Verteilung der aktuell benutzten Schlüssel
- Belegungsgrad der Hash-Tabelle HT
- Anzahl der Sätze, die sich auf einer Adresse speichern lassen, ohne Kollision auszulösen (Bucket-Kapazität)
- Technik zur Kollisionsauflösung
- ggf. Reihenfolge der Speicherung der Sätze

Hash-Verfahren: Einflußfaktoren II

Belegungsfaktor der Hash-Tabelle

- Verhältnis von aktuell belegten zur gesamten Zahl an Speicherplätzen ß = n_a /m
- für ß>0.85 erzeugen alle Hash-Funktionen viele Kollisionen und damit hohen Zusatzaufwand
- Hash-Tabelle ausreichend groß zu dimensionenieren (m > n_a)

Für die Hash-Funktion h gelten folgende Forderungen:

- Sie soll sich einfach und effizient berechnen lassen (konstante Kosten)
- Sie soll eine möglichst gleichmäßige Belegung der Hash-Tabelle HT erzeugen, auch bei ungleich verteilten Schlüsseln
- Sie soll möglichst wenige Kollisionen verursachen

Hash-Funktionen I: Divisions-Verfahren

1. Divisionsrest-Verfahren: $h(K_i) = K_i \bmod q$, $(q \sim m)$

Der entstehende Rest ergibt die relative Adresse in HT

Beispiel:

Die Funktion nat wandle Namen in natürliche Zahlen um:

nat(Name) = ord (1. Buchstabe von Name) - ord ('A')

h (Name) = nat (Name) mod m

Wichtigste Forderung an Divisor q:

q = Primzahl (größte Primzahl <= m)

- Hash-Funktion muß etwaige Regelmäßigkeiten in Schlüsselverteilung eliminieren, damit nicht ständig die gleichen Plätze in HT getroffen werden

HT: m=10

	Schlüssel	Daten
0		
1	BOHR	D1
2	CURIE	D2
3	DIRAC	D3
4	EINSTEIN	D4
5	PLANCK	D5
6		
7	HEISENBERG	D7
8	SCHRÖDINGER	D8
9		

- Bei äquidistantem Abstand der Schlüssel $K_0 + j \cdot K$, $j = 0, 1, 2, \ldots$ maximiert eine Primzahl die Distanz, nach der eine Kollision auftritt. Eine Kollision ergibt sich, wenn

$K_0 = (K_0 + j \cdot K) \bmod q$ d.h. $j \cdot K = k \cdot q$, $k = 1, 2, 3, \ldots$

- Eine Primzahl q kann keine gemeinsamen Faktoren mit K besitzen, die den Kollisionsabstand verkürzen würden

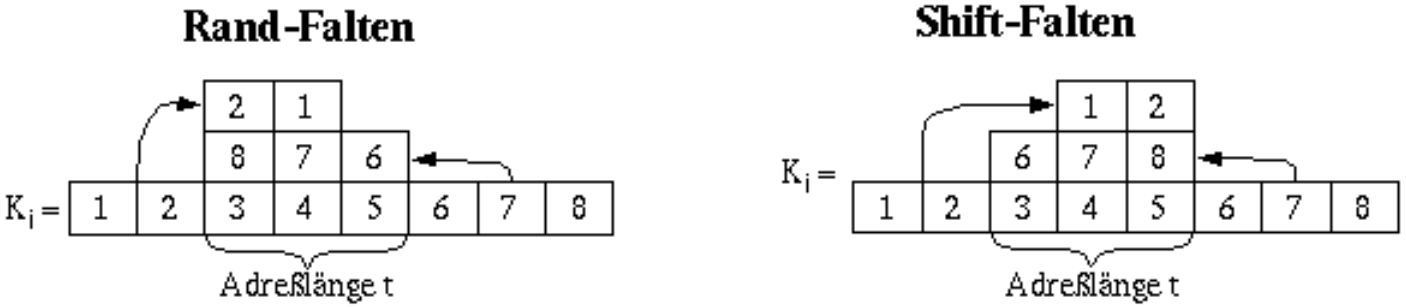
Hash-Funktionen II: Faltung

Schlüssel wird in Teile zerlegt, die bis auf das letzte die Länge einer Adresse für HT besitzen

- Schlüsselteile werden dann übereinandergefaltet und addiert.

Variationen:

- Rand-Falten: wie beim Falten von Papier am Rand
- Shift-Falten: Teile des Schlüssels werden übereinandergeschoben
- Sonstige: z.B. XOR-Verknüpfung bei binärer Zeichendarstellung
- Beispiel: b = 10, t = 3, m = 10^3



Faltung

- verkürzt lange Schlüssel auf "leicht berechenbare" Argumente, wobei alle Schlüsselteile Beitrag zur Adreßberechnung liefern
- diese Argumente können dann zur Verbesserung der Gleichverteilung mit einem weiteren Verfahren "gehasht" werden

Hash-Funktionen III

Mid-Square-Methode

- Schlüssel K_i wird quadriert. t aufeinanderfolgende Stellen werden aus der Mitte des Ergebnisses für die Adressierung ausgewählt.

- Es muss also $b^t = m$ gelten.
- mittlere Stellen lassen beste Gleichverteilung der Werte erwarten
- Beispiel für b = 2, t = 4, m = 16: $K_i = 1100100$

$$K_i^2 = 10011\underbrace{1000}_{t}10000 \rightarrow h(K_i) = 1000$$

Hash-Funktionen IV

Zufallsmethode:

- K_i dient als Saat für Zufallszahlengenerator

Ziffernanalyse:

- setzt Kenntnis der Schlüsselmenge K voraus. Die t Stellen mit der besten Gleichverteilung der Ziffern oder Zeichen in K werden von K_i zur Adressierung ausgewählt

Problemangepaßte Methoden:

Unter Ausnutzung von speziellen Eigenschaften der Schlüssel wird versucht, eine möglichst günstige Hashfunktion zu konstruieren. Im günstigsten Fall erhält man so eine Hashfunktion, die

- injektiv und
- effektiv zu berechnen ist.

Bewertung von Hash-Funktionen

Verhalten / Leistungsfähigkeit einer Hash-Funktion hängt von der gewählten Schlüsselmenge ab

- Deshalb lassen sie sich auch nur unzureichend theoretisch oder mit Hilfe von analytischen Modellen untersuchen
- Wenn eine Hash-Funktion gegeben ist, läßt sich immer eine Schlüsselmenge finden, bei der sie besonders viele Kollisionen erzeugt
- Keine Hash-Funktion ist immer besser als alle anderen

Über die Güte der verschiedenen Hash-Funktionen liegen jedoch eine Reihe von empirischen Untersuchungen vor

- Das Divisionsrest-Verfahren ist im Mittel am leistungsfähigsten; für bestimmte Schlüsselmengen können jedoch andere Techniken besser abschneiden
- Wenn die Schlüsselverteilung nicht bekannt ist, dann ist das Divisionsrest-Verfahren die bevorzugte Hash-Technik
- Wichtig dabei: ausreichend große Hash-Tabelle, Verwendung einer Primzahl als Divisor

Behandlung von Kollisionen

Zwei Ansätze, wenn $h(K_q) = h(K_p)$

- K_p wird in einem separaten Überlaufbereich (außerhalb der Hash-Tabelle) zusammen mit allen anderen Überläufern gespeichert; Verkettung der Überläufer
- Es wird für K_p ein freier Platz innerhalb der Hash-Tabelle gesucht ("Sondieren"); alle Überläufer werden im Primärbereich untergebracht ("offene Hash-Verfahren")

Methode der Kollisionsauflösung entscheidet darüber, welche Folge und wie viele relative Adressen zur Ermittlung eines freien Platzes aufgesucht werden

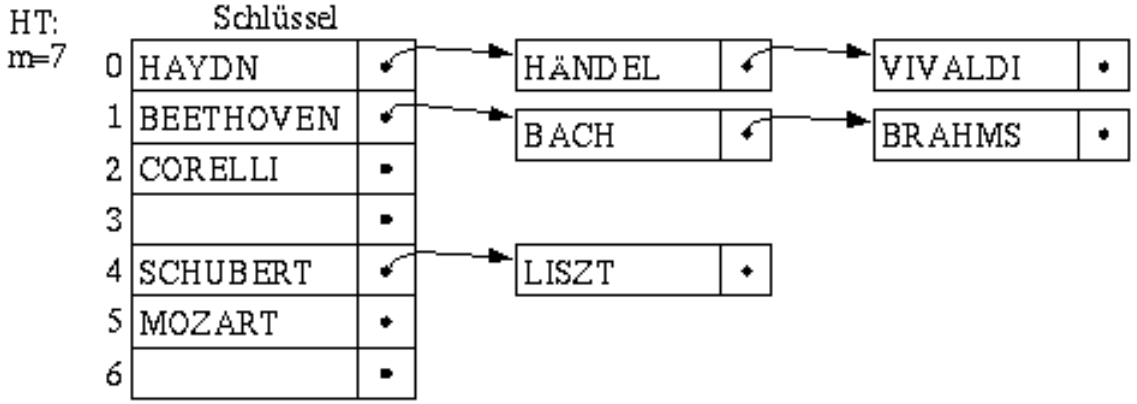
Adressfolge bei Speicherung und Suche für Schlüssel K_p sei $h_0(K_p)$, $h_1(K_p)$, $h_2(K_p)$, ...

- Bei einer Folge der Länge n treten also n-1 Kollisionen auf
- Primärkollision: $h(K_p) = h(K_q)$
- Sekundärkollision: $h_i(K_p) = h_j(K_q)$, $i \neq j$

Hash-Verfahren mit Verkettung der Überläufer (separater Überlaufbereich)

Dynamische Speicherplatzbelegung für Synonyme

- Alle Sätze, die nicht auf ihrer Hausadresse unterkommen, werden in einem separaten Bereich gespeichert (Überlaufbereich)
- Verkettung der Synonyme (Überläufer) pro Hash-Klasse
- Suchen, Einfügen und Löschen sind auf Kollisionsklasse beschränkt
- Unterscheidung nach Primär- und Sekundärbereich: n > m ist möglich !



Entartung zur linearen Liste prinzipiell möglich

Nachteil: Anlegen von Überläufern, auch wenn Hash-Tabelle (Primärbereich) noch wenig belegt ist

Offene Hash-Verfahren: Lineares Sondieren

Eigenschaften

- Speicherung der Synomyme (Überläufer) im Primärbereich
- Hash-Verfahren muß in der Lage sein, eine Sondierungsfolge, d.h. eine Permutation aller Hash-Adressen, zu berechnen

Lineares Sondieren (linear probing)

Von der Hausadresse (Hash-Funktion h) aus wird sequentiell (modulo der Hash-Tabellen-Größe) gesucht. Offensichtlich werden dabei alle Plätze in HT erreicht:

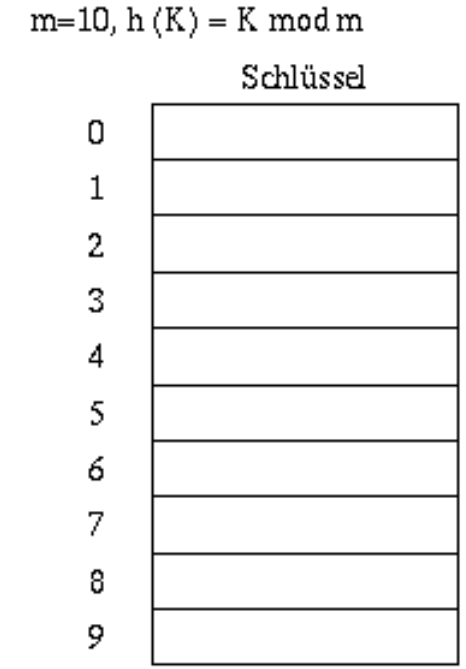
$$h_0(Kp) = h(K_p)$$

$$h_i(Kp) = (h_0(K_p) + i) \bmod m, \quad i = 1, 2, ...$$

Beispiel: Einfügereihenfolge 79, 28, 49, 88, 59

- Häufung von Kollisionen durch "Klumpenbildung"

  => lange Sondierungsfolgen möglich



Lineares Sondieren (2)

Aufwendiges Löschen

- impliziert oft Verschiebungen
- entstehende Lücken in Suchsequenzen sind aufzufüllen, da das Antreffen eines freien Platzes die Suche beendet.