FHV

Algorithmen & Datenstrukturen

Hashing

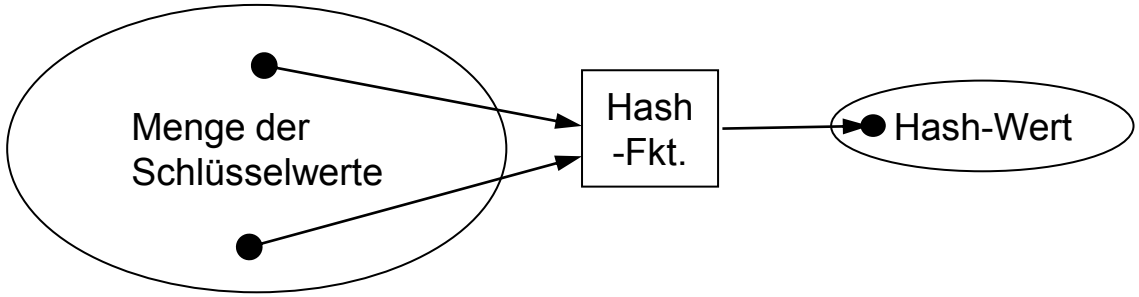
Wolfgang Auer

Motivation

- Suchen ist eine der häufigsten Aufgabenstellungen in der Informatik
- Lösungsansätze
  - Lineares Suchen *O(n)*
    - Die zu durchsuchende Menge besitzt keine festgelegte Ordnung und wird sequentiell durchsucht.
  - Binäres Suchen *O(log n)*
    - Elemente werden sortiert eingefügt. Die Suche erfolgt mittels Devide-and-Conquer.
  - Binärer Suchbaum *O(log n)*
    - Daten werden in einer Baumstruktur gespeichert, wobei auch hier wieder eine Ordnung der Elemente besteht.
  - Hash-Suche *O(1)*
    - Suchkriterium (Schlüssel) wird „direkt“ auf eine Adresse abgebildet.

Hashing

- Idee
  - Abbildung des Suchkriteriums (Schlüssels) auf eine natürliche Zahl im Bereich von 0 .. m-1
  - Diese natürliche Zahl dient als Index in einem Feld (*Hash-Tabelle)*
- Die Abbildung wird als die *Hash-Funktion* bezeichnet
  - Idealer Fall
    - Anzahl der möglichen Schlüssel = Größe der Hash-Tabelle ⇒ **Direkte Adressierung**
  - Allgemeiner Fall



Kardinalität der Menge der Schlüssel >> Karidinalität der Menge der Hash-Werte

Hash-Funktion (1)

FHV

- Hash-Funktion
  - h: K $\rightarrow$ T,
    K.. Menge von Schlüsseln $\{k_0, k_1, ..., k_{n-1}\}$
    T.. Menge der Hash-Werte $\{0, 1, \ldots, m - 1\}$
  - h(K) wird als der Hash-Wert des Schlüssels K bezeichnet.
- Wichtige Eigenschaften
  - Effiziente Berechnung
  - Gleichverteilung der Ergebnisse

- Bsp: Speicherung von Buchstaben in einer Tabelle der Größe 7
  - Als Schlüssel eines Buchstabens wird seine Position im Alphabet verwendet.
    $A_1, B_2, \ldots, Z_{26}$
  - h(key) = key Mod 7
  - Einfügen von B_2, J_{10}, S_{19}

0	1	2	3	4	5	6
		B_2	J_{10}		S_{19}	

Hash-Funktion (2)

FHV

- Die Wahl der Hash-Funktion ist im Prinzip beliebig

- Beispiele für mögliche Hash-Funktionen auf Zeichenketten
  - h(key) = ORD(key[1]) Mod m
  - h(key) = (ORD(key[1]) + Len(key)) Mod m
  - If (len(key) == 1)
    h(key) = ORD(key[1]) * 7 + 1) * 17 Mod m
    else
    h(key) = ORD(key[1]) * 7 + ORD(key[2]) + Len(key)) * 17 Mod m

Hash-Funktionen sind nach zunehmender Güte der Ergebnisse gereiht

Kollision

FHV

- Kardinalität der Schlüsselmenge größer als die Kardinalität der Menge der Hash-Werte
  $\Rightarrow$ Unterschiedliche Schlüssel ergeben den gleichen Hash-Wert.

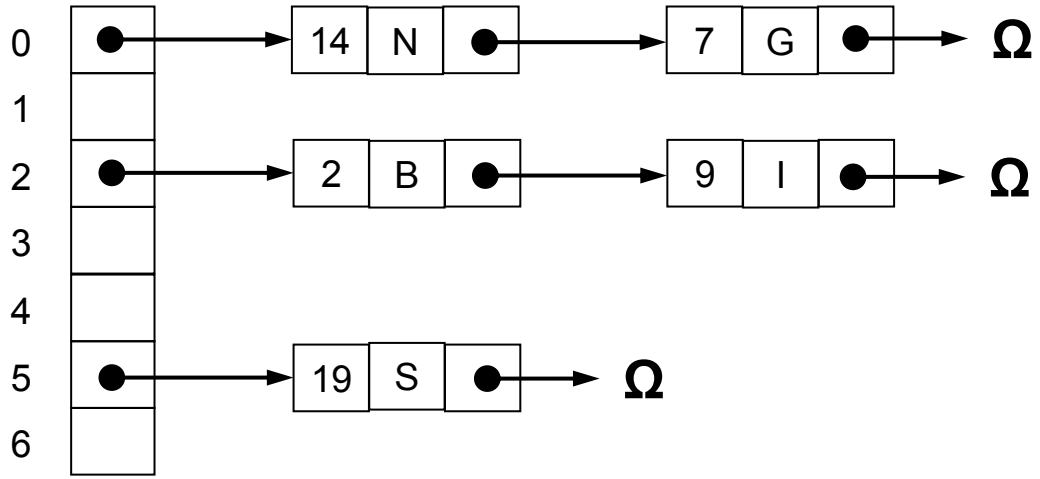
  h(key) ist nicht injektiv, da $h(key_1) = h(key_2)$ nicht bedeutet, dass $key_1 = key_2$ ist.

- Versucht man verschiedene Schlüssel auf eine Tabellenposition abzubilden, tritt eine *Kollision* auf.

- Die Behebung einer Kollision wird als *Kollisionsbehandlung* bezeichnet

„Offenes Hashing“

- Kollisionsbehandlung durch Verkettung. Man spricht auch vom *Separate Chaining*
- Alle Schlüssel, die denselben Hash-Wert *h* liefern, werden in einer linearen Liste gespeichert. Diese Liste wird in der Hash-Tabelle an der Position *h* verankert.



- Einfügen erfolgt mit O(1)
- Suchen mit **O(Länge der Kollisionskette)**

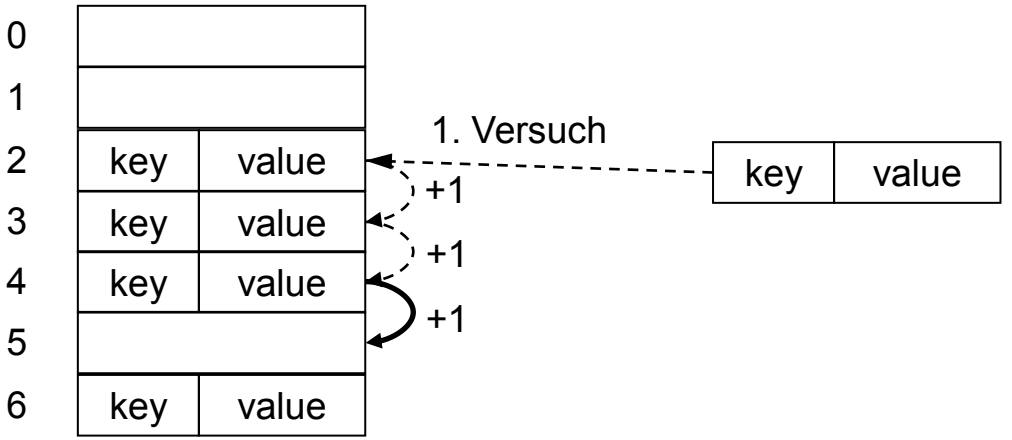
„Geschlossenes Hashing“

FHV

- Im Gegensatz zum Offenen Hashing enthält hier die *Hash-Tabelle* direkt die Werte
- Bei einer Kollision muss eine *alternative Stelle* zum Einfügen des Elements mit dem Schlüssel *K* gesucht werden. Man spricht von der *Sondierung.*
  Die Folge der untersuchten Stellen wird als *Sondierungsfolge* bezeichnet.
- Zur Bestimmung der alternativen Stellen können verschiedene Strategien angewendet werden
  - Lineare Kollisionsstrategie
  - Quadratische Kollisionsstrategie
  - Doppeltes Hashing
  - ...

Lineare Kollisionsstrategie

- Tritt eine *Kollision* auf, wird die Tabelle solange sequentiell durchsucht, bis eine freie Stelle gefunden wird, oder festgestellt wird, dass die Hash-Tabelle voll ist.



- Man wertet die Hash-Funktion
  h(key, i) = (h(key) + i) Mod m für alle i ≥ 0 solange aus, bis eine leere Stelle gefunden wurde und i < m ist.

Probleme der Linearen Kollisionsstrategie

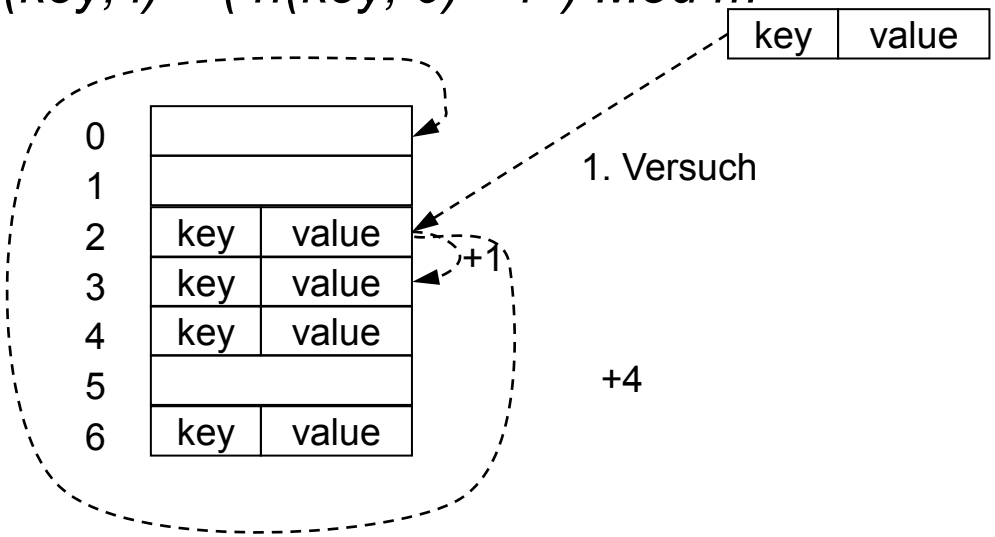
FHV

- *Primäres Clustering:* Durch die sequentielle Abarbeitung entstehen große, zusammenhängende Blöcke von Elementen. Je größer die Cluster werden, desto wahrscheinlicher werden Kollisionen.
  Diese Tendenz wird bei steigendem *Belegungsgrad (load-factor)* der Tabelle noch verstärkt

- *Sekundäres Clustering:* Sondierungsfolge für synonyme Schlüssel ist immer identisch.

Quadratische Kollisionsstrategie

- Anpassung der Hash-Funktion zur Vermeidung des primären Clusterings
- $h(key, 0) = h(key)$
  $h(key, i) = (h(key, 0) + i^2) \; Mod \; m$



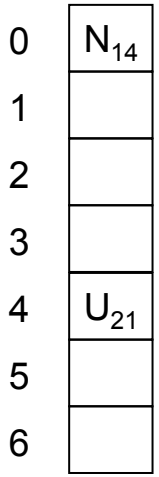
- Problem des sekundären Clusterings bleibt weiterhin bestehen

Doppeltes Hashing

- *Doppeltes Hashing* verwendet bei einer Kollision eine zweite, von *h* unabhängige Hash-Funktion *p* zur Bestimmung einer alternativen Position.
- *h(key, i) = (h(key) + i * p(key)) Mod m*
- Bsp:

  h(k) = k mod m

  p(k) = 1 + k mod (m-1)



Einfügen von N_{14} in die leere Hash-Tabelle
h(14) = 14 mod 7 = 0
p(14) = 1 + 14 mod 6 = 3
⇒h(14,0) = 0

Einfügen von U_{21}
h(21) = 21 mod 7 = 0
p(21) = 1 + 21 mod 6 = 4

h(21,0) = 0 — Kollision mit N_{14}
h(21,1) = (h(21) + 1 * p(21)) Mod 7 =
(0 + 4) Mod 7 = 4

Pitfalls

FHV

- Wahl einer schlechten Hash-Funktion
  - z.B. wird $m = 2^n$ gewählt, bewirkt die Verwendung von $h(k) = k \; Mod \; m$, dass nur die letzten n-bits des Schlüssels berücksichtigt werden.
- Wahl des Sondierungsschritts
  - Bei schlechter Wahl der Schrittweite für die nächste Sondierung, kann es dazu kommen, dass nicht alle Tabelleneinträge angesprochen werden können. D.h. man nützt nicht die gesamte Hash-Tabelle aus.
  - $p(k)$ muss prim zu m sein. D.h. $p(k)$ und m haben keinen gemeinsamen Teiler. Daher wird m oft eine Primzahl sein. Wird $m = 2^n$ gewählt, dann muss $p(k)$ eine ungerade Zahl liefern, damit das Doppelte Hashing funktioniert.

Suchen und Löschen

- Suchen
  - Separate Chaining
    - Mittels der Hash-Funktion wird der Anker der linearen Liste, die das Element hält (enthalten kann) in *O(1)* gefunden. Die Suche in der Liste erfolgt sequentiell in *O(Anzahl der Listeneinträge)*
  - Geschlossenes Hashing
    - Der Sondierungspfad wird solange gefolgt, bis das Element gefunden wurde, oder alle Elemente geprüft worden sind.
- Löschen
  - Separate Chaining
    - Entfernen des entsprechenden Listenelements
  - Geschlossenes Hashing
    - Entfernen eines Elements zerstört den Sondierungspfad. Eine Lücke auf dem Pfad bedeutet, dass das Element nicht enthalten ist, sonst würde es an dieser Position stehen!
    - „Gelöschte“ Elemente werden nur als gelöscht markiert. Beim nächsten Einfügeversuch an dieser Stelle kann das markierte Element überschrieben werden

Hashing Bewertung

FHV

- Allgemein
  - ▲ Aufwand für Zugriff auf ein Element im besten Fall konstant O(1)
  - ▼ Kollisionsbehandlung erfordert zusätzlichen Aufwand
  - ▼ Bei hohem Belegungsgrad steigt die Wahrscheinlichkeit von Kollisionen
  - ▼ Kein Zugriff in sortierter Reihenfolge
- Offenes Hashing
  - ▲ Dynamische Datenstruktur, die beliebig viele Elemente befassen kann
  - ▼ Kann zur Linearen Suche degenerieren
- Geschlossenes Hashing
  - ▲ Schnelles Suchen
  - ▲ Speicherplatz wird gut ausgenützt (abhängig vom Belegungsgrad)
  - ▼ Statische Datenstruktur
  - ▼ Löschen von Elementen umständlich