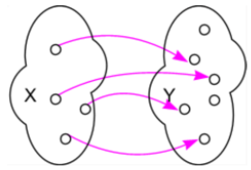
Kapitel 7: Hashing

Grundgedanke:

Verteilung der Datensätze im Adressraum des Speichermediums aufgrund einer Zuordnung (Hash-Funktion), die jedem möglichen Schlüsselwert eine Speicheradresse zuordnet.

Injektivität von Funktionen

- Jeder Punkt der Zielmenge wird höchstens einmal getroffen.



Ideale Konfiguration:

- **Beliebig viel Speicher**, damit jedes Objekt einen Platz erhalten kann.
- Eine **injektive Hash-Funktion** die einen Speicher-Ort für ein Objekt berechnet.

Definitionen:

W = alle möglichen Schlüssel || **S** = zu speichernden Schlüssel ||**m** = verfügbarer Speicher

Hashing unter realen Bedingungen

- Da i.A. S nicht a priori bekannt und |W| sehr gross (z.B. alle Integer-Zahlen...) und m begrenzt, ist die Hashfunktion h i.A. nicht injektiv.
- **Salopp:** Wenn m+1 oder mehr Werte abgespeichert werden können sollen, so ist eine Doppelbelegung einer Speicherposition theoretisch nicht zu verhindern.

Eigenschaften von Hash-Funkt.

- Wenn zwei Elemente k und k' auf dieselbe Speicherposition abgebildet → k und k' **Synonyme**.
- Enthält **S** Synonyme, so treten **Adress-Kollisionen** auf.
- Das Verhältnis **β**=|S|/m heißt **Belegungsfaktor** der Hash- Tabelle.
- Eine **injektive Hashfunktion** heißt auch **perfekt**.

Entwurfsentscheidungen f. Hashing

Wahl einer geeigneten Hash-Funktion

- (fast) injektiv, effizient zu berechnen

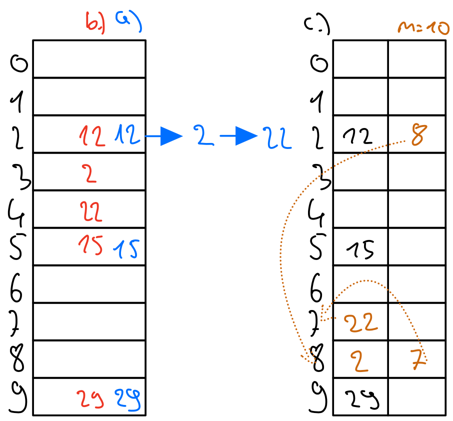
$h(x) = x \bmod 10$ $m=10$

12, 15, 29, 2, 22

Behandlung von Kollisionen

Problem:

- Hash-Tabelle enthält Schlüssel k und soll nun einen Schlüssel k' aufnehmen mit h(k)=h(k'), **k'** heißt **Überläufer**.



Lösungen:

a) Überläufer werden in separatem Überlaufbereich verkettet gespeichert (separate overflow).

a.) -> β > 1 möglich!

b) Es wird ein freier Platz in der Hash-Tabelle gesucht (open adressing).

b.) -> β > 1 nicht möglich!

c) Überläufer werden innerhalb der Hash-Tabelle verkettet gespeichert (coalessed hashing).

c) – Nachteil:
Speicherplatz kann bis zur Hälfte reduziert werden

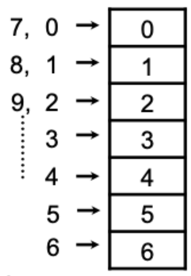
Wahl einer Hash-Funktion **Allgemein**: h : W→ {0,1,...,m-1}

Anforderungen:

- effizient berechenbar.
- möglichst gleichmäßige Verteilung der Daten auf den Speicherbereich zur Kollisionsvermeidung, dabei insgesamt möglichst wenige Kollisionen.
- Kein Hashwert soll unmöglich sein, jedes Ergebnis (jeder Wert im definierten Wertebereich) soll tatsächlich vorkommen können **(surjektiv).**
- h soll ordnungserhaltend sein, falls die Hashfunktion einen sortierten Zugriff in einer Hashtabelle erlauben soll. $a<b \Rightarrow h(a)<h(b)$.

$$p_m(n) = 1 - \frac{m!}{(m-n)! \cdot m^n}$$

Divisions-Rest Methode **h(k) = k mod m**



Forderungen an m:

- m ist eine ungerade Zahl: *(damit gut streut)!*
- $m \neq b^l$, wobei b die Basis der Schlüsseldarstellung

Multiplikative Methode

- $h(k) = \lfloor m\,(k{*}A \bmod 1) \rfloor$
  $= \lfloor m\,(k{*}A - \lfloor k{*}A \rfloor) \rfloor$

Eigenschaften:

- Die Wahl von m ist bei dieser Methode unkritisch
- Der Grad der Verteilung hängt von der Wahl von A ab

$\boldsymbol{\Theta - \lfloor\Theta\rfloor,\ 2\Theta - \lfloor 2\Theta\rfloor,\ \ldots,\ n\Theta - \lfloor n\Theta\rfloor}$ teilt das Intervall [0,1] in n+1 Teilintervalle mit höchstens 3 verschiedenen Längen.

- **Irrationale Zahl**: eine Zahl die nicht als Bruch dargestellt werden kann. Etwa Wurzel n , π, e, ...
- Beste Wahl von A: der **goldene Schnitt (Knuth):** $\bar{\Theta} = \frac{\sqrt{5}-1}{2} \approx 0{,}618033$

Anwendung als Hashfunktion: $\boldsymbol{h(k) = \lfloor m\,(k\Theta - \lfloor k\Theta \rfloor) \rfloor}$

Beispiel:

- S = (1,2,3,4,5,6,7,8,9,10)
- $h(1) = \lfloor 10*(1*0{,}618 - 0) \rfloor = 6$ $h(2) = \lfloor 10*(1{,}23 - 1) \rfloor = 2$ $h(3) = \lfloor 10*(1{,}8 - 1) \rfloor = 8$ ....
  6,2,8,4,0,7,3,9,5,1

Abstände benachbarter Zahlen sehr gleichmäßig (nur 3 verschiedene Werte): 4,6,4,4,7,4,6,4,4

Methode der Quadrierung

Schlüssel wird quadriert, anschließend werden t aufeinanderfolgende Stellen aus der Mitte des Resultats ausgewählt.

- Basis b=2, Länge m=16, Auswahlbereich t=4
- k= 1100100 k2 = 10011 1000 10000
- h(k)=1000

Bewertung der Hash-Funktionen

- Im Mittel arbeitet die Divisions-Rest-Methode am besten
- Genauer: Für den Zweck des Füllens von Hash-Tabellen am besten, für Prüfsummen, Passwörter etc. andere Hash - Funktionen besser.

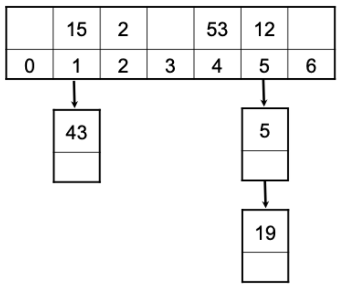
Aufwandsbetrachtung für Hashing - *Hashverfahren mit separatem Überlauf*

Wörterbuchoperationen (Suchen, Einfügen, Löschen):

- **Suche** nach Schlüssel k:
  Beginne bei t[h(k)] und folge den Verweisen der Überlaufkette bis entweder k gefunden (erfolgreiche Suche) oder das Ende der Überlaufkette erreicht ist.
  - worst case: $\Theta(n)$ *(Wenn alles Synonyme)*
  - average case: $O(1)$
- **Einfügen**: Suche erfolglos, dann Element am Ende einfügen
  - Suchen $O(1)$, Einfügen in Liste $O(1)$: $O(1)$ in average
- **Löschen**: Suche erfolgreich, dann lösche das Element.
  - Suchen $O(1)$, Löschen in Liste: $O(1)$ in average:
- **Gesamt**: $O(1)$ in average

Bemerkung: Benutze für Einfügen und Löschen die bekannten Operationen auf verketteten linearen Listen.

Behandlung von Kollisionen



Hashverfahren mit separatem Überlauf

- Beispiel: S(13,53,5,15,2,19,43), m=7

Implementierungsvarianten:

- Jedes Element der Hash-Tabelle ist Anfangselement einer Überlaufliste.
- Jedes Element der Hash-Tabelle ist Zeiger auf eine Überlauf-Kette

Vorteile:

- Anzahl der Schlüsselvergleiche im Mittel niedrig.
- Ein Belegungsfaktor >1 ist möglich
  - wichtig für unerwartet anwachsende Datenbestände
- Einträge werden (echt) gelöscht,
  - d.h. Speicherplatz für später einzufügende Elemente wiederverwendbar.
- Geeignet für den Einsatz mit externen Speichern
  - Hash-Tabelle im Hauptspeicher, Überlauf extern.

Nachteile:

- zusätzlicher Speicherplatz für Zeiger
- Speicherplatz nötig für Überläufer
- selbst bei ausreichend Platz in der Hash-Tabelle.

Offene Hash-Verfahren

Prinzip: • Speicherung der Überläufer an freien Plätzen in der Hash-Tabelle.

Problem:

- Benötigt wird eine Regel, die ausgehend von einer Hash-Adresse weitere Hash-Adressen sondiert (auf Verfügbarkeit überprüft).
- Beispiel: lineares Sondieren: h(k),h(k)-1,...,0,m-1, h(k) +1

Beispiel:

- S = (23,62,17,16) m=7
- h(23)= 23 mod =2
- h(62)= 6, h(17)=3
- h(16)= 2 (besetzt),
    - versuche h(16)-1=2-1=1 (frei).

0	1	2	3	4	5	6	7
	16	23	17			62	

j ist max m

Suche nach Schlüssel k (lineares Sondieren):

- Berechne für aufsteigendes j die Hash-Adresse i=h(k)-j mod m, bis
    - entweder Hash-Tabelle t[i]=k gilt (erfolgreich)
    - oder bis t[i] frei ist (erfolgloses Suchen)

  → **Konsequenz**: gespeicherte Elemente können nicht „echt" entfernt werden, sondern nur markiert uns ggf. überschrieben werden.

Entfernen von Schlüssel k:

• Suche nach k, Falls k gefunden markiere k als gelöscht, aber lasse es in der Hash-Tabelle.

0	1	2	3	4	5	6	7
	16	~~23~~	17			62	

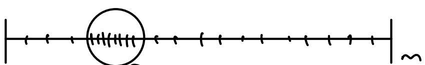
- dahinter kann es linear sondiert sein?

Einfügen eines Schlüssels k:

- Suche nach k. Merke dabei den ersten Index i mit t[i] als entfernt markiert.
- Falls k nicht in Hash-Tabelle:
    - wenn es einen Index i gibt, so füge k bei t[i] ein,
    - wenn es keinen Index i gibt, so füge k an freier Stelle in die Hash-Tabelle ein.

Bemerkung:

- Lineares Sondieren wird ineffizient bei hoher Belegungsdichte, da belegte Adressen zu Clustern zusammenwachsen, daher hohe Anzahl an Schlüsselvergleichen



Knuppelbildung

erhöht die Kollisionswahrscheinlichkeit!

Quadratisches Sondieren:

- Um die Häufung des linearen Sondierens zu vermeiden, wird beim quadratischen Sondieren für Schlüssel k um h(k) mit quadratisch wachsendem Abstand nach einem freien Platz gesucht.
- Sondierungsfolge: h(k) , h(k)+1, h(k)-1, h(k)+4, h(k) -4 , ...
    - allgemein: $h(k) + S(j,k)$ mit $S(j,k) = (\lceil j/2 \rceil)^2(-1)^{j+1}$

Bemerkung:

- ist m ein Primzahl und zusätzlich darstellbar als 4i+3 für ein passendes i, dann ist garantiert, dass die Sondierungsfolge eine Permutation der Hash-Adressen 0, ... , m-1 ist.