Hochschule Offenburg
University of Applied Sciences

Algorithmen und Datenstrukturen
9. Hashverfahren

Prof. Dr. Klaus Dorer

Übersicht

- Hashverfahren
  - Hashfunktion
  - Adresskollision und Auflösung
    - Offene Hashverfahren
    - Verkettung der Überläufer

Ziele

- Hashing verstehen und Vor- und Nachteile nennen können
- Hashtabellen einsetzen können
- Hashfunktionen für Datentypen entwickeln können

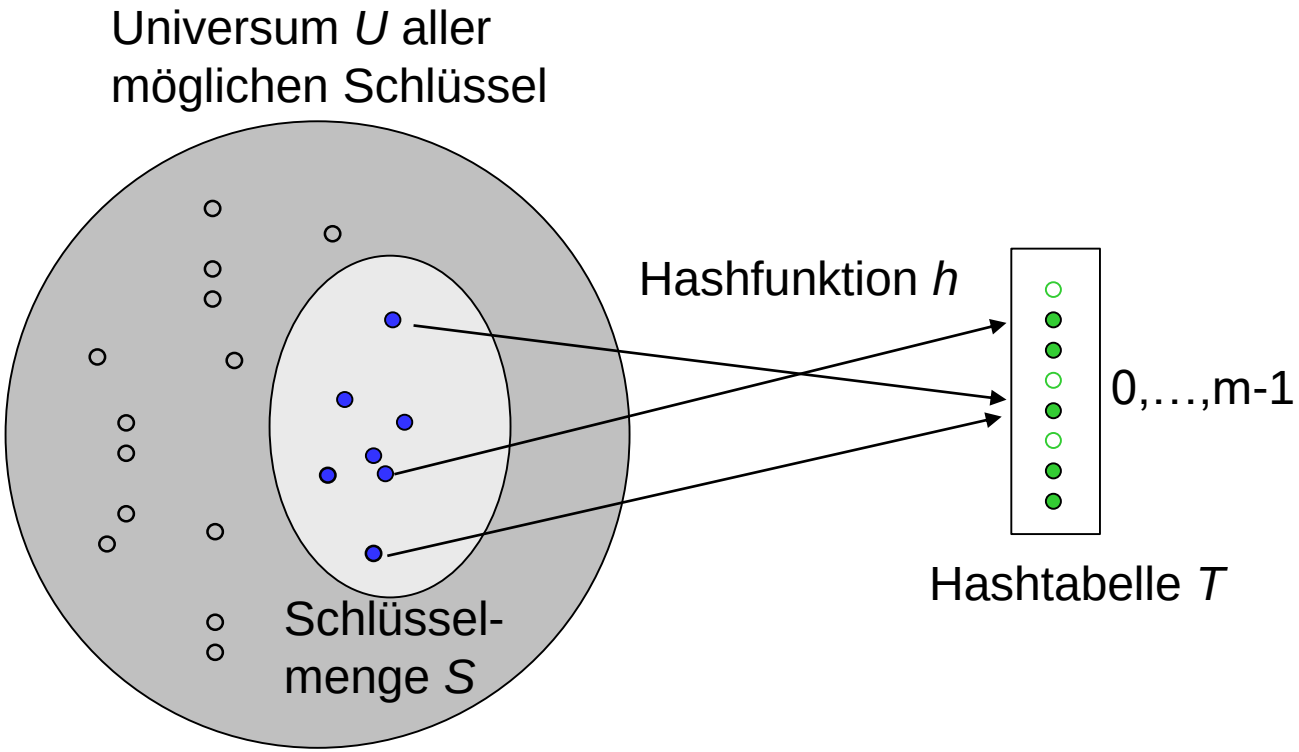
Quellen

- Thomas Ottmann und Peter Widmayer, Algorithmen und Datenstrukturen, 4. Auflage, Spektrum, Berlin, 2002
- S. Trahasch, Vorlesungsfolien Algorithmen und Datenstrukturen, Hochschule Offenburg
- www.wikipedia.de

Hashing

- Bisher kennen wir Datenstrukturen, bei denen das Suchen eines Eintrags mit Schlüssel k in der Größenordnung O(n) (unsortierte Daten) oder O(log n) (sortierte Daten, Bäume) liegt.
- Einzige Operation für Schlüssel war der Schlüsselvergleich
- Hashing ist ein Verfahren, das es erlaubt, die Adresse eines Schlüssels aus dem Schlüssel auszurechnen
- Suchen funktioniert dann im Mittel in konstanter Zeit O(1)

Hashing


Universum *U* aller
möglichen Schlüssel
Hashfunktion *h*
0,…,m-1
Hashtabelle *T*
Schlüssel-
menge *S*

Begriffe

- Datenschlüssel sind in einem linearen Feld mit Indizes 0…m-1, der Hashtabelle
- Eine Hashfunktion h: K->{0…, m-1} ordnet jedem Schlüssel k einen Index h(k) zu, die Hashadresse
- Meist gibt es deutlich mehr als m mögliche Schlüssel. Die Hashfunktion kann also i.A. nicht injektiv sein, verschiedene Schlüssel werden auf dieselbe Hashadresse abgebildet
- Zwei Schlüssel k und k‘ mit h(k) = h(k‘) heißen Synonyme
- Sollen beide in derselben Hashtabelle gespeichert werden ergibt sich eine Adresskollision
- Sind in einer Hashtabelle (der Größe m) n Schlüssel gespeichert ist der Belegungsfaktor α=n/m

Hashverfahren

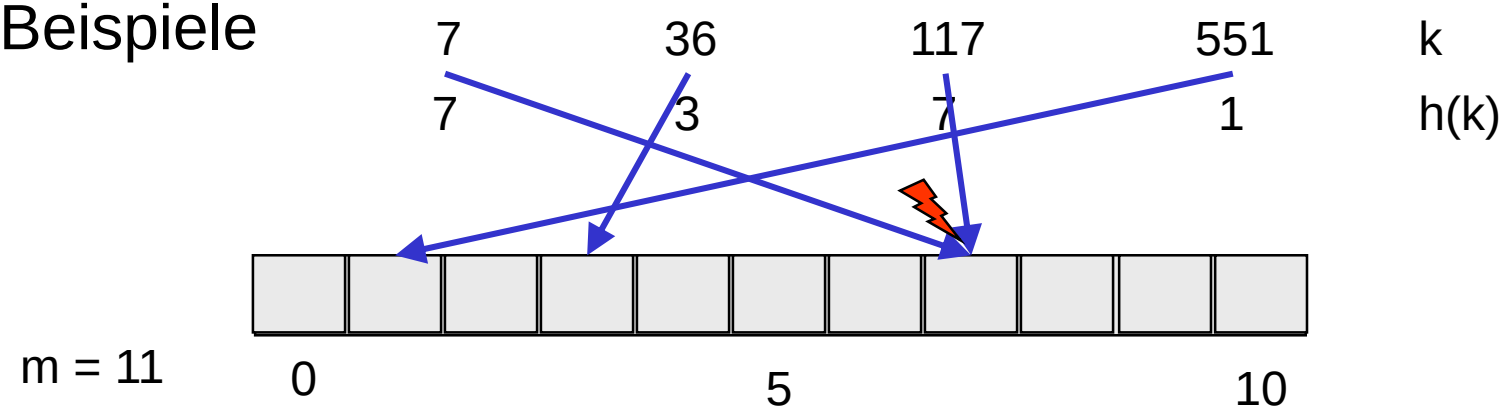
- Effizienz eines Hashverfahrens hängt davon ab, wie gut es Adresskollisionen vermeidet bzw. auflöst
  - Schlüssel sollten möglichst gleichmäßig in der Hashtabelle verteilt sein
  - Bei hohem Belegungsfaktor werden Adresskollisionen immer wahrscheinlicher
  - Die meisten Implementierungen streben einen Belegungsfaktor von unter 75% an ->25 % des Speichers der Hashtabelle liegt brach
- Problem
  - Schlüssel sind meist nicht ganze Zahlen
  - Index hängt von der Größe der Hashtabelle ab
  - Berechnung der Hashadresse muss effizient sein
  - Adresskollisionen sind wahrscheinlicher als man denkt

Divisions-Rest-Methode

- h(k) = k mod m

- Entscheidend ist dann die Wahl von m
  - Ist m gerade -> h(k) gerade falls k gerade ist. Das ist schlecht, wenn das niedrigste Bit gehäuft 0 oder 1 ist
  - Ist $m=2^i$ -> h(k) liefert gerade die letzen i Bit von k, höherwertige Bits haben keinen Einfluss auf das Hashing
  - Man wählt daher üblicherweise bei dieser Methode m als Primzahl

- Beispiele

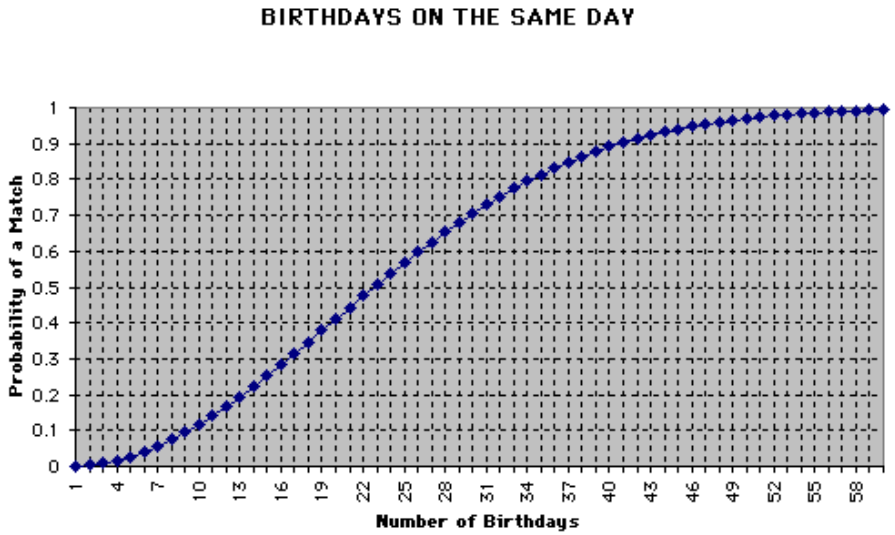


Multiplikative Methode

- $h(k) = \lfloor m(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor) \rfloor$
  - Mit $\phi^{-1} = (\sqrt{5}-1)/2 \approx 0.6180339887$ (goldener Schnitt)
  - $\lfloor\ \rfloor$ - ganzzahliger Anteil
- h(1) … h(10) für m=10 ist dann 6, 2, 8, 4, 0, 7, 3, 9, 5, 1
- Also gleichmäßige Verteilung im Intervall und zwar immer im größten verbliebenen Intervallteil
- Beispiel
  - k=27 -> h(k) = ?

Adresskollision

- Hashfunktion kann in der Regel Adresskollisionen nicht vermeiden



Quelle: http://www.mste.uiuc.edu/reese/birthday/

- Lösung
  - Offene Hashverfahren
    - Überläufer werden an einer anderen Stelle in der Hashtabelle gespeichert
  - Verkettung der Überläufer
    - Überläufer werden in einer verketteten Liste außerhalb der Hashtabelle gespeichert

Offene Hashverfahren

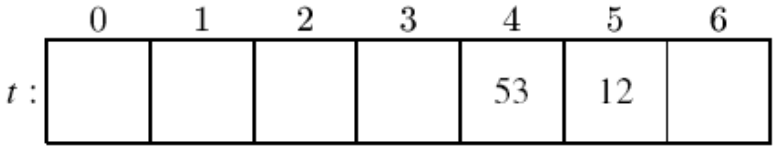
- Überläufer werden in der Hashtabelle untergebracht
- Adresskollision beim Einfügen
  - Nach fester Regel werden andere Tabellenplätze durchsucht
  - Beim ersten freien Tabellenplatz wird eingefügt
- Folge der Tabellenplätze eines Schlüssels heißt Sondierungsfolge
- Entfernen von Schlüsseln ist problematisch, da ‚dahinter' liegende Überläufer dann nicht mehr gefunden werden
  - Daher wird der Schlüssel in der Regel nur als entfernt markiert
  - Ist nur effizient, wenn hauptsächlich eingefügt und gesucht wird

Offene Hashverfahren

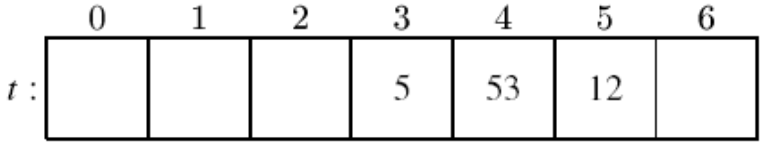
- s(j, k) sei eine Funktion so dass (h(k) – s(j, k)) mod m für j=0, 1, …, m-1 eine Sondierungsfolge bildet
- Suchen
  - Beginne bei i=h(k)
  - Solange k ≠ t[i] und t[i] nicht leer suche bei i= (h(k) – s(j, k)) mod m für aufsteigende Werte von j
  - Wenn t[i] leer ist war Suche erfolglos
- Einfügen
  - Beginne bei i=h(k)
  - Solange t[i] nicht leer suche bei i= (h(k) – s(j, k)) mod m für aufsteigende Werte von j
  - Trage k bei t[i] ein (Annahme: k ist nicht schon in der Tabelle)
- Entfernen
  - Suche nach Schlüssel und markiere ihn als entfernt, falls Suche erfolgreich

Offene Hashverfahren

- Lineares Sondieren
  - s(j, k) = j
- Beispiel
  - Einfügen 12, 53



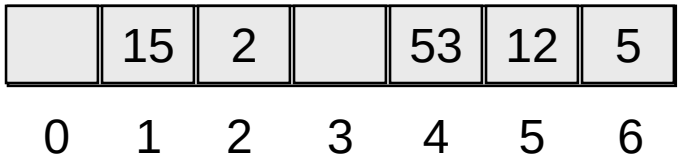
  - Einfügen 5: h(5) = 5->belegt, 4 auch belegt, 3 frei



  - Wahrscheinlichkeit, dass der nächste Schlüssel auf t[2] fällt ist 4/7 für t[1] aber nur 1/7
  - Primäre Häufung (Clustering)
  - Verstärkt sich selbst

Offene Hashverfahren

- Quadratisches Sondieren
  - $s(j,k) = (-1)^j \left\lceil \frac{j}{2} \right\rceil^2$
  - $\lceil\ \rceil$ - nächst höhere ganze Zahl
  - Wenn m eine Primzahl der Form 4i + 3 ist, ist s eine Sondierung
  - Folge: h(k), h(k) – 1, h(k) + 1, h(k) – 4, h(k) + 4, …
- Beispiel
  - Einfügen von 12, 53, 5 (Folge: h(5), h(5) – 1, h(5) + 1), 15, 2, 19?



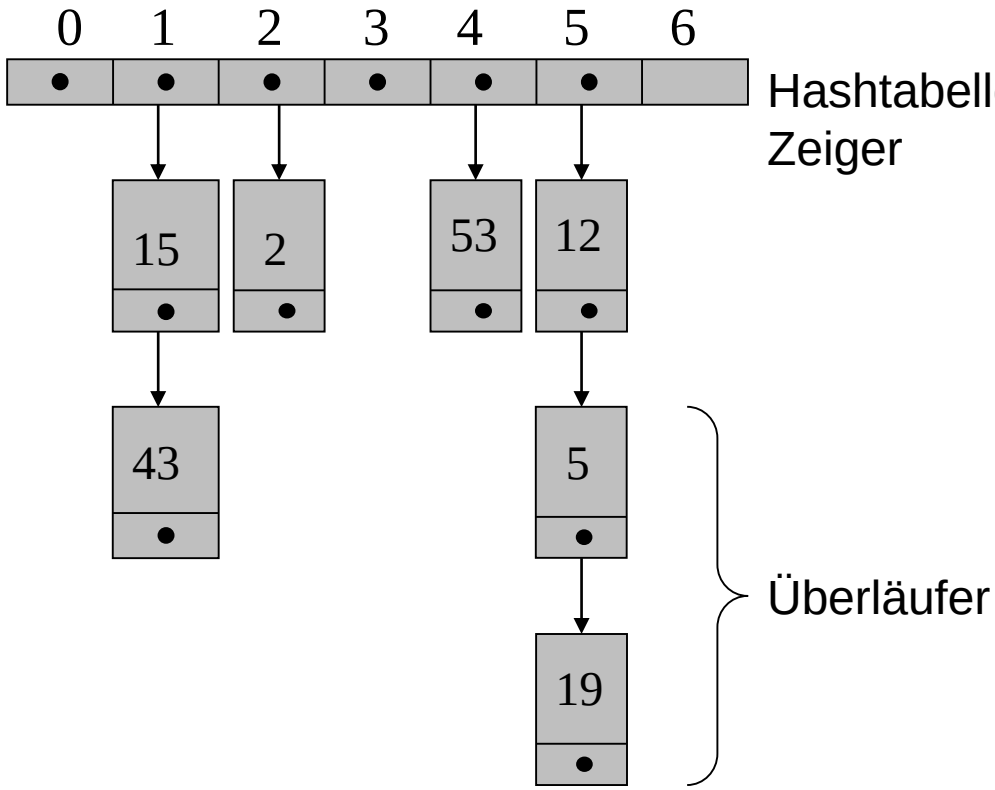
  - Keine primäre Häufung
  - Aber sekundäre Häufung, Synonyme haben immer dieselbe Sondierungsfolge (gilt natürlich auch für lineares Sondieren)
  - Problem: s(j, k) hängt nicht von k ab

Offene Hashverfahren

- Uniformes Sondieren
  - s(j, k) liefert eine Permutation, die nur von k abhängt und jede der m! möglichen Permutationen gleich wahrscheinlich ist
  - Sehr aufwändig praktisch zu realisieren
- Zufälliges Sondieren
  - s(j, k) zufällig in Abhängigkeit von k
  - Es kann dann vorkommen dass s(j‘, k) = s(j, k) für j‘ > j
  - Ist annähernd gleich effizient

Verkettung der Überläufer

- Bei Adresskollision wird das neue Element an die Liste des entsprechendenTabelleneintrags h(k) angefügt
- Beispiel
  - Einfügen von 12, 53, 5, 15, 2, 19, 43 mit h(k) = k mod m und m=7



Verkettung der Überläufer

- Suchen
  - Starte bei der Liste tabelle[h(k)] und durchlaufe die Liste bis das Element gefunden wurde, oder das Ende der Liste erreicht ist
  - O(n) im schlechtesten Fall
  - O(1) im Mittel falls α klein ist
- Einfügen
  - Suche nach k
  - Falls gefunden nicht einfügen, oder altes Element überschreiben
  - Falls nicht gefunden füge Element an Liste an
- Entfernen
  - Suche nach k
  - Falls nicht gefunden fertig
  - Falls gefunden lösche Element aus der Überlaufkette

Verkettung der Überläufer

Vorteile

- Erwartungswert der Anzahl der betrachteten Einträge ist niedrig
- Belegungsfaktor >1 ist möglich
- Löschen von Einträgen ist leicht möglich
- Einfach, leicht zu implementieren

- Nachteile
  - Es wird zusätzlicher Speicher für die Listenzeiger gebraucht, auch wenn gar keine Adresskollision vorliegt

Vergleich der Effizienz

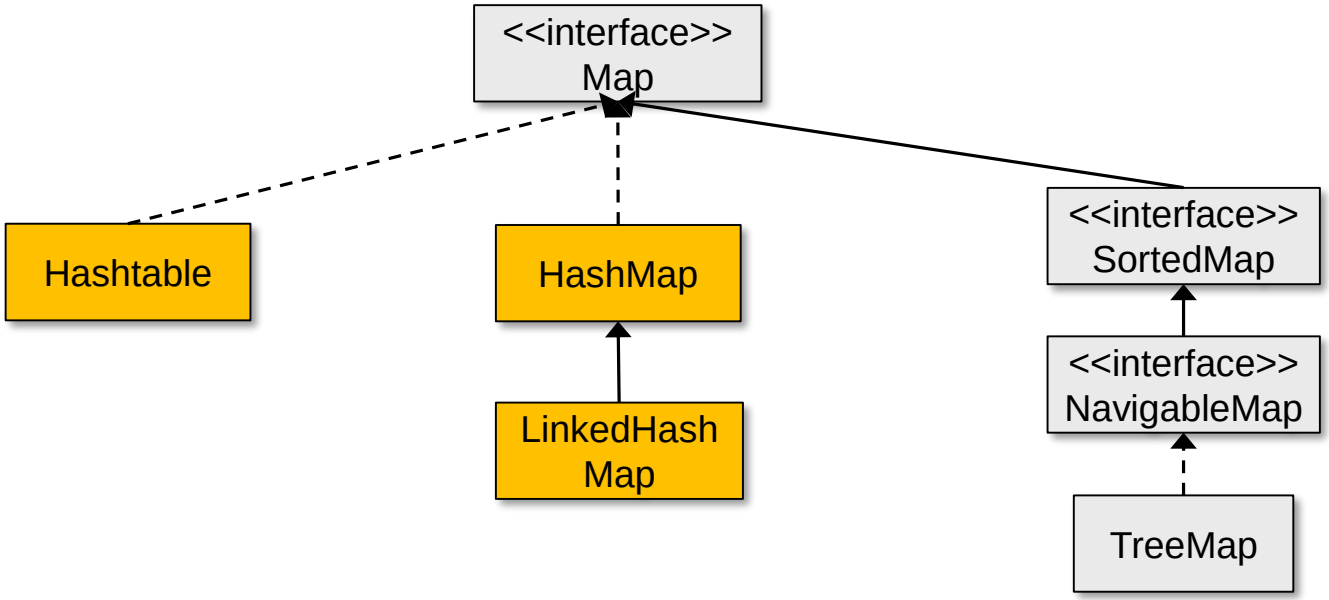
Anzahl betrachteter Einträge (α=)		0.50	0.90	0.95	1.00
Lineares Sondieren	erfolgreich	1.5	5.5	10.5	-
	erfolglos	2.5	50.5	200.5	-
Quadratisches Sondieren	erfolgreich	1.44	2.85	3.52	-
	erfolglos	2.19	11.40	22.05	-
Uniformes Sondieren	erfolgreich	1.39	2.56	3.15	-
	erfolglos	2	10	20	-
Verkettung der Überläufer	erfolgreich	1.110	1.307	1.337	1.368
	erfolglos	1.250	1.450	1.475	1.500

Dynamisches Hashing

- Größe m der Hashtabelle muss vorab festgelegt werden
  - Offene Hashverfahren können nie mehr als m Schlüssel speichern
  - Hashverfahren mit Verkettung werden ineffizient für $\alpha > 1$
- Einfache Lösung: Rehashing
  - Wenn bestimmter Belegungsfaktor α überschritten wird, alle Datensätze in eine neue, größere Hashtabelle speichern
  - Wird so in Java gemacht
  - Problem: bei extern gespeicherten Daten ist das zu teuer
- Komplexere Lösungen
  - Lineares Hashing (2 Hashfunktionen)
  - Virtuelles Hashing (n Hashfunktionen)

Hashdatenstrukturen in Java

- Das Map Interface in Java




```
Map<Integer, String> map = new HashMap<>();
// add with key  in O(1) on average
map.put(1, "Neuer");
// get value from key in O(1) on average
String value = map.get(Integer.valueOf(1));
System.out.println("Value is: " + value);
```

Hashdatenstrukturen in Java

- Hashfunktion
  - h(k) = |k| mod m (Hashtable)

```
int hash = key.hashCode();
int index = (hash & 0x7FFFFFFF) % tab.length;
```

- Jede Klasse stellt 2 Methoden zur Verfügung
  - boolean equals(Object obj): prüft zwei Objekte auf Gleichheit
  - int hashCode(): liefert k (hash) falls Objekte der Klasse als Schlüssel verwendet werden
  - Beispiel der Klasse String (vereinfacht)

```
public int hashCode()
{
    int h = 0;
    int length = value.length >> 1;
    for (int i = 0; i < length; i++) {
        h = 31 * h + getChar(value, i);
    }
    return h;
}
```

Zusammenfassung

- Hashverfahren erlauben einen sehr effizienten Zugriff auf Daten
- Der Preis ist ein erhöhter Speicherverbrauch gegenüber Arrays oder Listen
- Man benötigt
  - Hashtabelle
  - Hashfunktion
  - Vorgehensweise bei Adresskollision