Suchen, Maps und Hashing

- Sie wissen, wie in Strings gesucht werden kann
- Sie kennen den Begriff der Invarianten
- Sie wissen, wie binäres Suchen funktioniert
- Sie wissen, wie Hashing funktioniert
- Sie kennen das Java Interface Map und die Implementation HashMap

Zürcher Hochschule
für Angewandte Wissenschaften

zh
aw
School of
Engineering

Suchen

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Beispiele, wo gesucht werden muss:

- Prüfen, ob ein Wort in einem Text vorkommt
- Zählen, wie oft ein Wort in einem Text vorkommt
- Überprüfen, ob alle Worte korrekt geschrieben sind
- Finden einer Telefonnummer in einem Telefonbuch
- Zählen, wie oft die richtige Zahlenkombination im Lotto gewählt wurde
- Prüfen, ob eine Kreditkartennummer gesperrt ist
- Prüfen, ob in zwei Listen die gleichen Elemente vorkommen

Finden eines Teilstrings in einem String

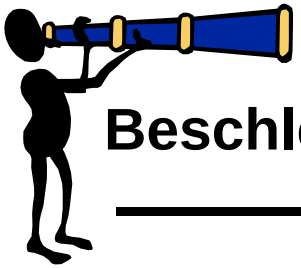
```
D r e i   r e i n e   r e i s e n d e
-------------------------------------
r e i s
  r e i s
            . . .
                    r e i s
-------------------------------------
r e i n
  r e i n
      . . .
          r e i n
```

- int indexOf(String str, String pattern)
  - liefert die Position, an der das Muster beginnt
  - -1 falls das Muster nicht vorkommt

Implementation in Java

```
int indexOf(String str, String pattern) {
  for (int i = 0; i < str.length() - pattern.length() + 1; i++) {
    int k;
    for (k = 0;
         k < pattern.length() && str.charAt(i+k) == pattern.charAt(k);
         k++);
    if (k == pattern.length()) return i;
  }
  return -1;
}
```

- Muster wird an die Position i gesetzt
- Es wird mit dem String verglichen bis
  - Ende des Musters erreicht -> Erfolg
  - Nichtübereinstimmung
- Aufwand ist O(.....)



Beschleunigte Stringsuche: Knuth-Morris-Pratt (KMP)

- Naive Stringsuche ist vor allem aufwändig, wenn grosse Teilübereinstimmungen vorkommen.
- Idee von KMP: bei der Suche wird Information gesammelt, die für die weitere Suche verwendet werden kann.
- Aufwand O(m + n)
  - aber: Muster muss vorher analysiert (compiliert) werden
  - Suche muss als Automat implementiert werden
- Optimierung von <5% der Fälle (wenn Teilübereinstimmung mit Muster)
- ➔ theoretisch interessant aber praktisch irrelevant

Beispiel

```
m: 012345678901234567890 12          - index in string
S: ABC ABCDAB ABCDABCDABDE           - string
P: ABCDABD                           - pattern
i: 0123456                           - index in pattern
```

Mismatch at S[3] → no 'A' found → set m=4

```
m: 012345678901234567890 12
S: ABC ABCD**AB** ABCDABCDABDE
P:     ABCD**AB**D
i:     0123456
```

Mismatch at S[10] → passed 'AB' →
set m=8 → P[0..1] need not to be checked

```
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
P:         ABCDABD
i:         01**23456**
```

Mismatch at S(10) → no 'C' found → set m=11

```
m: 012345678901234567890 12
S: ABC ABCDAB **ABCDAB**CDABDE
P:           **ABCDAB**D
i:           0123456
```

Mismatch at S[17] → passed 'AB' → set m=15 → P[0..1] need not to be checked

```
m: 012345678901234567890 12
S: ABC ABCDAB ABCD**ABCDABD**E
P:               **ABCDABD**
i:               01**23456**
```

match

aus Wikipedia

Suche in zwei Arrays

Griechische Götter
Hermes
Hestia
Hypnos
Kronos
Poseidon
Rhea
Uranos
Zeus

Römische Götter
Amor
Apollo
Jupiter
Mars
Uranos
Venus
Vesta
Vulcanus

- Fragestellung: welchem Gott kann ich opfern, ohne mich mit einer der beiden Götterwelten anzulegen.
- Heutige Fragestellungen:
  - Mitgliedschaft in verschiedenen Institutionen
  - Rasterfahndung

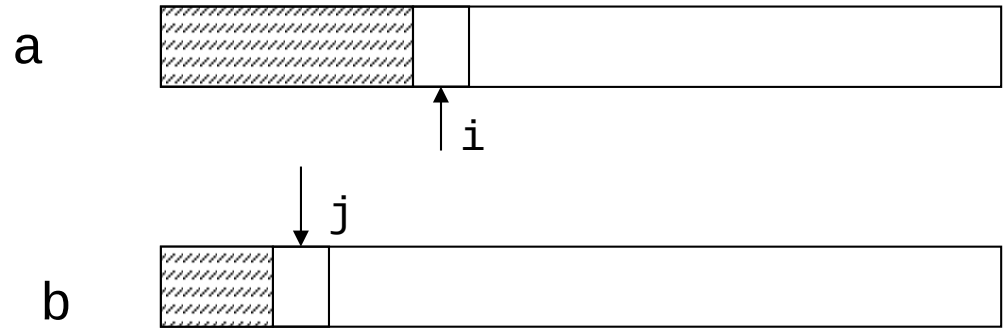
Einfacher Algorithmus

```
int indexOf(String[] a, String[] b) {
  for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < b.length; j++) {
      if(a[i].equals(b[j])) return i;
    }
  }
  return -1;
}
```

- doppelt geschachtelte Schleife
- Aufwand O(n*m)
  - bei 2 * 10000 Elementen ➔ 10^8

Griechische Götter	**Römische Götter**
Hermes	Amor
Hestia	Apollo
Hypnos	Jupiter
Kronos	Mars
Poseidon	Uranos
Rhea	Venus
Uranos	Vesta
Zeus	Vulcanus

Besserer Algorithmus wenn a und b sortiert



- ***Invariante*** $\forall k,n;k<j,n<i;b[k]\neq a[n]$
- Invariante ist eine Bedingung, die erhalten bleibt

- Idee: Bereich, für den die Invariante gilt, sukzessive erweitern
- *b[j] < a[i]* ➔ $\forall k;k\leq j;b[k]<a[i]$ ➔ *j um 1 erhöhen*
- *b[j] > a[i]* ➔ $\forall n;n\leq i;b[j]>a[n]$ ➔ *i um 1 erhöhen*

Schnelle Suche in zwei Arrays

```
int indexOf(String[] a, String[] b) {
  int i = 0, j = 0; boolean found = false;
  //{inv && i == 0 && j == 0}
  while (!found && (i < a.length-1 || j < b.length-1)) {
    int c = a[i].compareTo(b[j]);
    if (c == 0)  found = true;
    else if (c < 0)
       if (i < a.length-1) i++; else break;
    else
       if (j < b.length-1) j++; else break;
  }
  // {inv && (i == a.length-1 && j == b.length-1) ||
  //   (found && a[i] == b[j])}
  if (found) return i else return -1;
}
```

- i und j werden erhöht, so dass die Invariante erhalten bleibt
- Am Schluss gilt: Invariante **and** Abbruchbedingung
- Aufwand O(n+m)

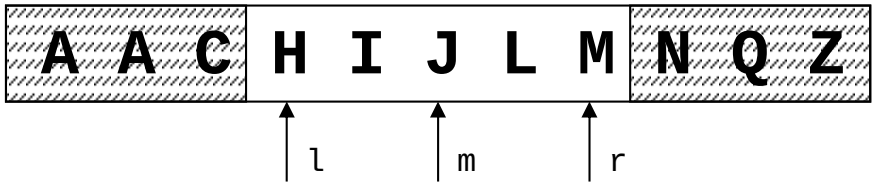
Schnelle Suche in einem Array: Binäres Suchen

- Gegeben sei ein sortierter Array von Werten
- Wie kann in so einem Array effizient gesucht werden?

- Führe zwei Indizes ein: l und r

- Invariante: $\forall k, n; k < l, n > r; (a[k] < s) \wedge (a[n] > s))$

Binäres Suchen



nehme m als Index zwischen l und r

- falls a[m] < s ➔ l = m+1
- falls a[m] > s ➔ r = m-1
- falls a[m] = s ➔ gefunden
- falls l > r ➔ keine Elemente mehr zwischen l und r ➔ nicht gefunden

Binäres Suchen

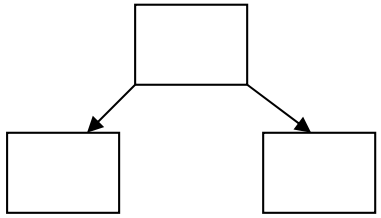
```
int binarySearch(int[] a, int v) {
  int l = -1;
  int r = a.length;
  {inv && l == -1 && r == a.length}
  while (l <= r) {
    int m = (l + r) / 2;
    if (a[m] == v) return m;
    else if (a[m] < v) l = m+1;
    else if (a[m] > v) r = m-1;
  }
  {inv && l > r}
  return -1;
}
```

Demo

- In jedem Durchgang wird r - l halbiert ➔ $\log_2$ Schritte
- Aufwand: O(log n)

Aufwand für Suchen und Einfügen

- **sortierter Array**
  - Einfügen O(n)
  - binäres Suchen O(log(n))
- lineare **sortierte Liste**
  - Einfügen O(n)
  - Suchen O(n)
- **sortierter Binärbaum**
  - Einfügen O(log(n))
  - Suchen O(log(n))



Frage:

- Gibt es ein Verfahren, dessen Aufwand unabhängig von der Anzahl Elemente ist?

Zürcher Hochschule für Angewandte Wissenschaften

zh aw School of Engineering

Maps und Sets

Maps (Abbildungen)

Gegeben sei eine Menge von Datensätzen der Form

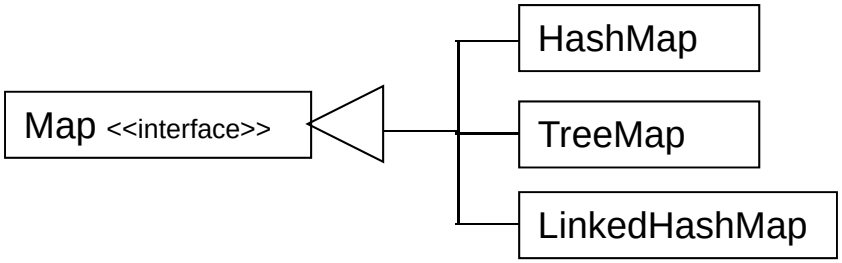
- Der Schlüssel kann ein Teil des Inhaltes sein
- Der Schlüssel besteht im einfachsten Fall aus einem String oder einem numerischen Wert.
- Mittels dem Schlüssel kann der Datensatz wiedergefunden werden.
- Bsp:
  - AHV-Nummer - Personen
  - Matrikel-Nummer- Studenten

- Aufgabe: Es sollen Daten (Objekte) in einen Behälter eingefügt und mittels ihrem Schlüssel wiedergefunden werden können.
- Eine solche Datenstruktur heisst *Map.* Sie bildet eindeutige Schlüssel auf Werte (Inhalte) ab. ➔ Java Interface *Map*.

Maps (Abbildungen)

Eine Map kann auf unterschiedliche Arten implementiert werden.

- Im einfachsten Fall, wenn der Schlüssel ein *int* ist, ist dies ein .....
- Das Java Collection Framework kennt drei Implementationen:
  - HashMap
  - TreeMap
  - LinkedHashMap

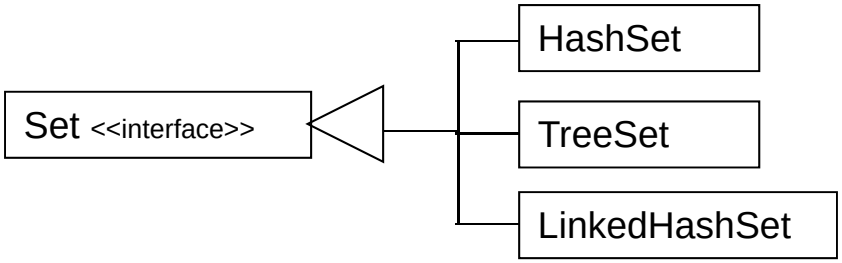


Map<K,V> Interface

void clear()	Löschen aller Elemente
int size()	Anzahl Elemente
V put(K key, V value)	Einfügen eines Elementes
V get (K key)	Finden eines Elementes
V remove(K key)	Löschen eines Elementes
boolean containsKey(K key)	ist Element mit Schlüssel in Table
boolean containsValue(V value)	hat ein Element den Wert
Collection<V> values()	alle Werte als Collection
Set<K> keySet()	alle Schlüssel als Set
Set<Map.Entry<K,V>> entrySet()	alle Schlüssel/Wertepaare als Set

Sets (Mengen)

- Ein *Set* ist eine Collection von Elementen, die keine Duplikate enthält.
- Gleich wie eine Map, aber statt Schlüssel/Wertepaare hat man nur Schlüssel.
- Java Interface *Set*, mit den Implementationen
  - HashSet
  - TreeSet
  - LinkedHashSet



Zürcher Hochschule für Angewandte Wissenschaften

zh aw School of Engineering

Implementationen einer HashMap: Hashtable

Menge der möglichen Schlüsselwerte klein

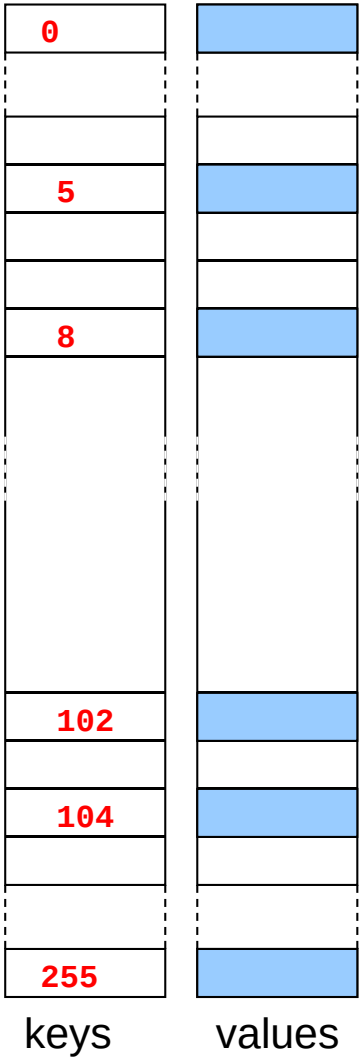
Werte in Array an ihrer Indexposition speichern

- Schlüsselbereich (char): 0..255
- Werte beliebig

- Einfach Object[] h = new Object[256];

- Einfügen h[i] = o;
- Suchen: o = h[i];

- Aufwand
  - Einfügen: O(1)
  - Suchen: O(1)



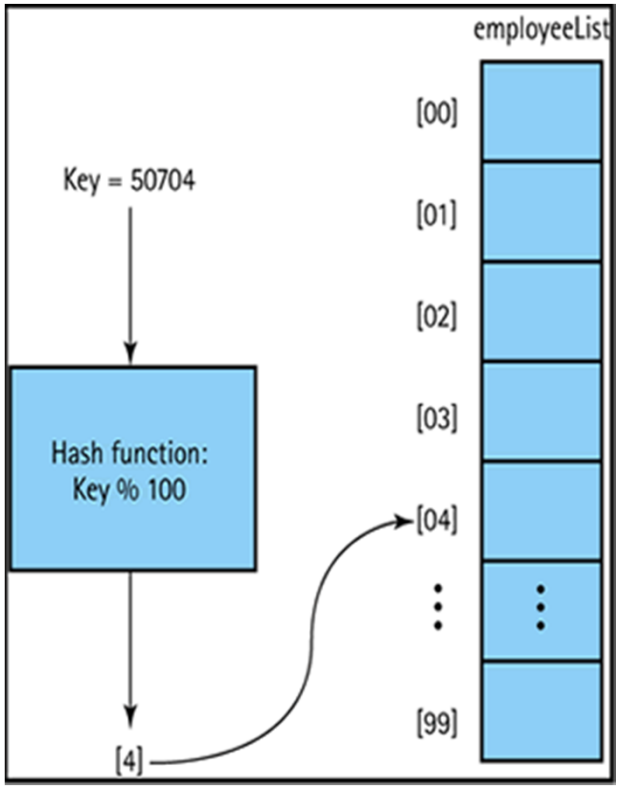
Idee : Hashing

Problem: der Array ist nur schwach belegt
- geht noch für Buchstaben
- was aber bei Zahlen? (>2^{32}) oder Strings

- Lösung:
  - Es wird eine Funktion h verwendet, welche den grossen Wertebereich auf einen kleineren abbildet.

    Einfachste Funktion: `X modulo tableSize`
    → eine Zahl zwischen 0 und tableSize-1

  - Eine solche Funktion nennt man ***Hash Funktion***.



Hash Funktion

Problem 1:

- Hash Wert nicht eindeutig, i.e. $\nexists\, h^{-1}(k)$

Lösung

$\Rightarrow$ **Originalwert** in Tabelle (z.B. Array) speichern

Problem 2:

- Zwei unterschiedliche Objekte können den gleichen Hash Wert haben. d.h. sie müssten an der gleichen Stelle gespeichert werden $\Rightarrow$ **Kollision**

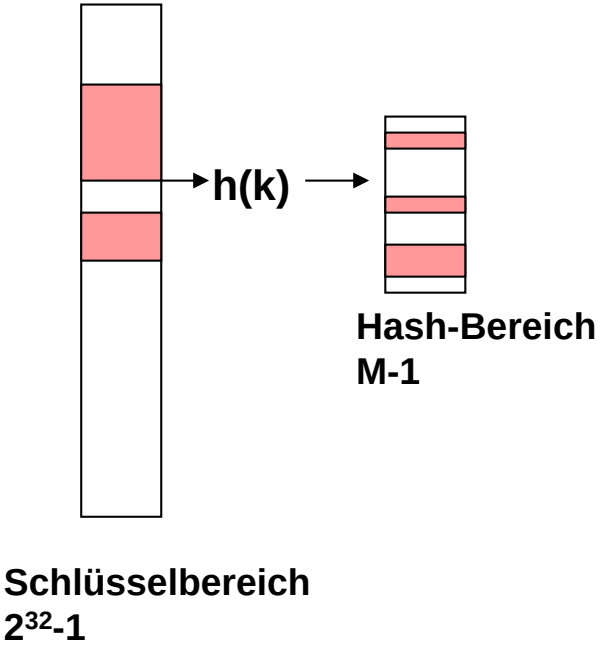
Lösung:

- Kollisionen werden vermieden - oder verringert
- Kollisionen werden aufgelöst
- verschiedene Verfahren zur Auflösung (später):
  - linear/quadratic probing, …

Hash Funktion

- Der grosse Schlüsselbereich wird mittels der Hash-Funktion auf einen kleinen Bereich abgebildet

- Problem
  - **Massierungen** im (Schlüssel-)Wertebereich ⇒ kann zu gleichen Hash-Werten führen ⇒ **Kollisionen**
- Hash = Durcheinander
  - die Hash-Funktion bringt den Schlüssel so "durcheinander", dass er möglichst **gleichmässig** auf den ganzen Hash-Bereich abgebildet wird
- Zwei gute Hashfunktionen
  - **h(k) = k % M | M ∈ Primzahl**
  - **h(k) = (k * N) % M | N,M ∈ Primzahl**



Hashtable ohne Kollisionen

```
public class Hashtable {
  final int MAX = 100;
  final int INVAL = 0;       // nicht das Gelbe vom Ei
  int[] keys = new int[MAX]; // initialisiert mit INVAL
  int[] vals = new int[MAX];

  private int h(int key) {
     return key % 97;
  }

   public void put(int key, int val) {
      int h = h(key);
      if (keys[h] == INVAL) {
         keys[h] = key;
         vals[h] = val;
      }
      else {/* COLLISION */}
   }

   public int get(int key) {
      int h = h(key);
      if (keys[h] == key) {
       return vals[h];
      }
      else /* COLLISION or not found */;
   }
}
```

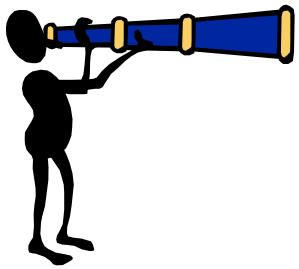
Hash Funktion

Überprüfe ob Feld frei

Speichere Zahl

Überprüfe ob Schlüssel korrekt

Hole Zahl



Hashing von Strings

- Hashfunktion = ASCII Wert $\times$ Position 128^n

*1	*128	*128 ^{2}	*128 ^{3}
B	A	U	M

- es entstehen sehr grosse Zahlen:
  - Ein vier Zeichen langer String führt bereits zu einer Zahl in der Grössenordnung von $128^4 = 2^{28}$, was nur wenig kleiner als der grösste 32-Bit Integer ist.

- In der Praxis werden deshalb Polynome zur Umwandlung von Strings verwendet (Horner Schema): $\mathbf{A_3x^3 + A_2x^2 + A_1x^1 + A_0x^0}$
  - kann als $(((A_3)x + A_2)x + A_1)x + A_0$ gerechnet werden.

- Modulo-Arithmetik: Es gilt:
  - (a+b) mod m = ((a mod m) + (b mod m)) mod m
  - (a*b) mod m = ((a mod m) * (b mod m)) mod m

Hashing von Strings

Beispiel B A U M (ASCII-Werte B=66, A=65, U=85, M=77)

- (((77*128+85)*128+65)*128+66) mod 19
- (((1*14+9)*14+8)*14+9) mod 19
- ((4*14+8))*14+9) mod 19
- ((7*14+9) mod 19
- 107 mod 19
- **12**

Java Methoden hashCode()

String: $s[0]*31^{n-1} + s[1]*31^{n-2} + ... s[n-1]$

Float: Integer Bit Darstellung

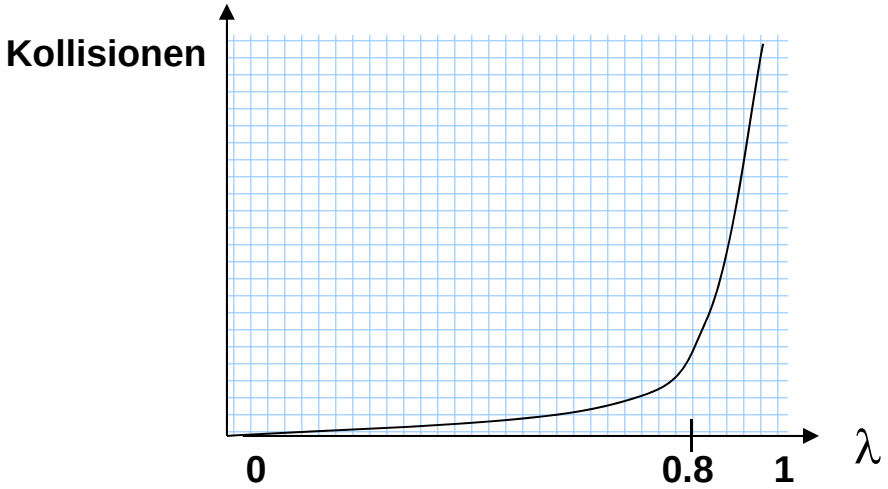
Double: XOR der Integer Bit Darstellung der beiden Teile

Object: Adresse

Kollisionen

- Anzahl Kollisionen hängt von der Güte der Hash-Funktion und der Belegung der Zellen ab

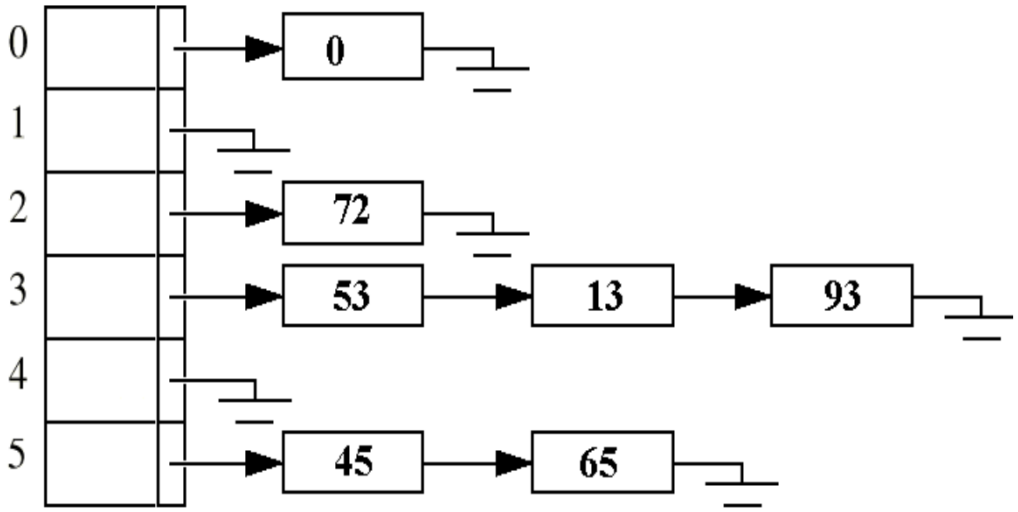
- Der LoadFactor λ
  - sagt wie stark der Hash-Bereich belegt ist
  - bewegt sich zwischen 0 und 1.
  - Anzahl Kollisionen ist abhängig von λ und h: f(h, λ)



Kollisionsauflösung 1 :Separate Chaining

Hashtable lediglich als Ankerpunkt für Listen aller Objekte, die den gleichen HashWert haben : **Überlauflisten** (*Separate Chaining*)

Eine solche Hashtable wird auch als *offene* Hashtable bezeichnet.



- Overhead durch Verwendung einer weiteren Datenstruktur
- Verfahren ist gut bei Load-Faktor nahe (oder grösser) als 1

h(key)=key%10

Übung

Gegeben sind:

- eine Hashtabelle der Grösse 10
- eine Hash-Funktion h(x) = x mod 10
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

Wie sieht die Tabelle aus, nachdem der Input unter Verwendung von Separate Chaining Hashing verarbeitet wurde?

Kollisionsauflösung 2: Open Addressing

Open Addressing: Techniken, wo bei Kollision eine freie Zelle sonstwo in der HashTable gesucht wird.
Wird auch als *geschlossene* Hashtable bezeichnet.

⇒ Setzt einen LoadFactor < ~0.8 voraus.

- **lineares Sondieren (*Linear Probing):***

  sequentiell nach nächster freier Zelle suchen
  (mit *Wrap around*).

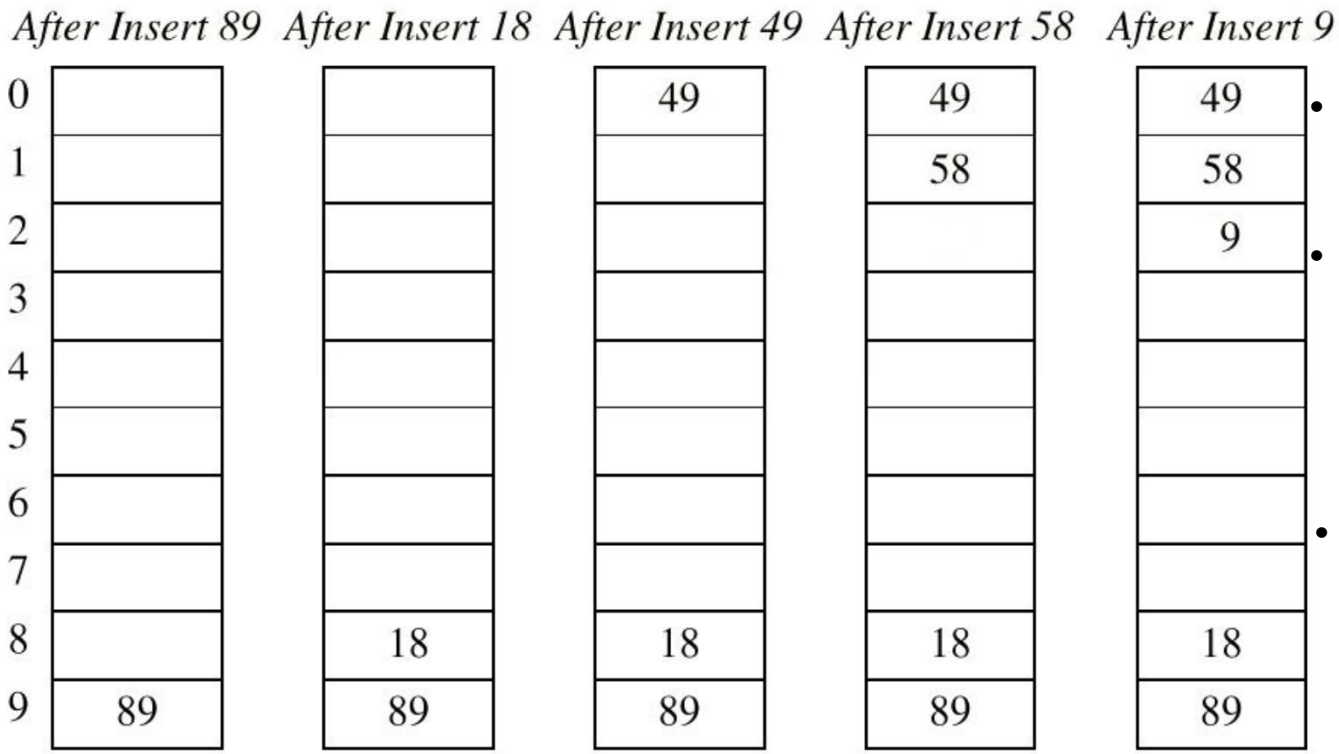
- ***quadratisches Sondieren (Quadratic Probing):***

  in wachsenden Schritten der Reihe nach F+1, F+4, F+9, . . . , F+i^2 prüfen
  (mit *Wrap around*).

Linear Probing 1

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

in eine Hash-Tabelle mit 10 Feldern werden der Reihe nach 89, 18, 49, 58 und 9 eingefügt.

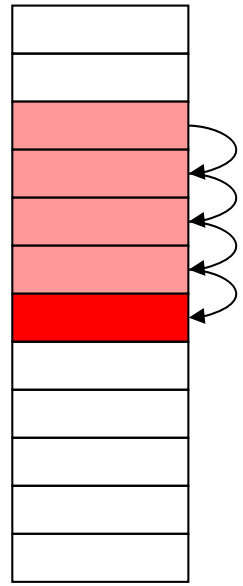


- Hash-Funktion Input modulo Tabellengrösse
- Bei zunehmendem Load Faktor dauert es immer länger bis eine Zelle gefunden wird. (Einfügen und Suchen)
- `find` funktioniert wie `insert`: Element wird in Tabelle ausgehend vom HashWert gesucht bis Wert gefunden oder leere Zelle.

Linear Probing 2

```
int findPos(Object x ) {
    int currentPos = hash(x);

    while( array[ currentPos ] != null &&
       !array[currentPos].equals( x ) ) {
        currentPos = (currentPos  + 1) % array.length ;
    }
    return currentPos;
}
```



zur Bestimmung einer Ausweichzelle wird einfach die nächste genommen →primary Clustering Phänomen

Linear Probing 3

Performance

ziemlich schwierig abzuschätzen, da der Aufwand nicht nur vom Load Faktor, sondern auch von der Verteilung der belegten Zellen abhängt.

Phänomen des Primary Clustering:

mussten einmal freie Zellen neben dem Hash-Wert belegt werden, steigt die Wahrscheinlichkeit, dass weiter gesucht werden muss für:

- alle Ausgangswerte mit gleichem Hash-Wert
- all jene, deren Hash-Wert in eine der nachfolgenden Zellen verweist.

Folge:

- Verlängerung des durchschnittlichen Zeitaufwandes zum Sondieren
- erhöhte Wahrscheinlichkeit, dass weiteres Sondieren nötig wird

⇒ Bei hohem Load Faktor/ungünstigen Daten bricht die Performance ein!

Übung

Gegeben sind:

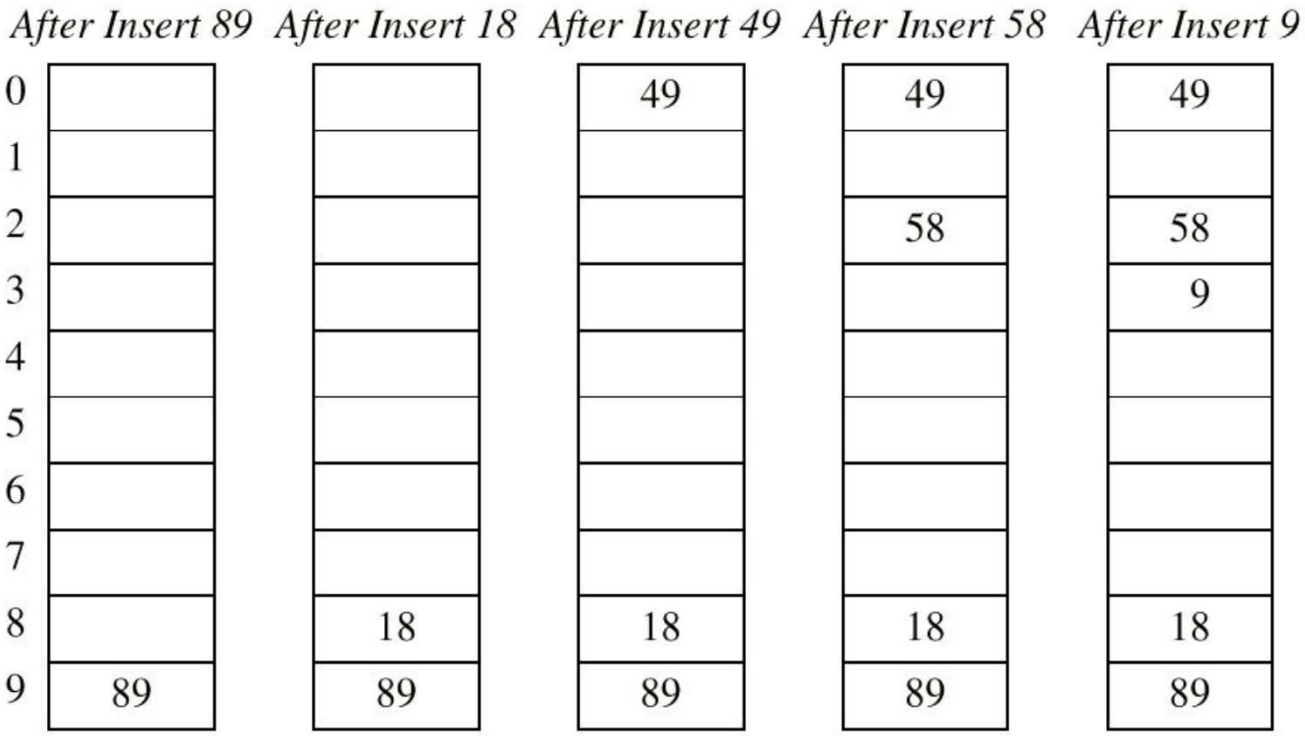
- eine Hashtabelle der Grösse 10
- eine Hash-Funktion H(X) = X mod 10
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

Wie sieht die Tabelle aus, nachdem der Input unter Verwendung einer linearen Sondiermethode verarbeitet wurde?

Quadratic Probing 1

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

in eine Hash-Tabelle mit 10 Feldern werden der Reihe nach 89, 18, 49, 58 und 9 eingefügt.



Hash-Funktion Input modulo Tabellengrösse

jetzt bleiben Lücken in Hash-Tabelle offen

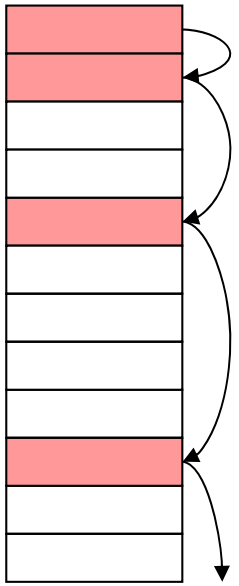
die zuletzt eingefügte 9 findet ihren Platz unbeeinflusst von der zuvor eingefügten 58.

Quadratic Probing 2

```
int findPos( Object x )
{
    int collisionNum = 0;
    int currentPos = hash(x);

    while( array[currentPos] != null &&
       !array[currentPos].equals( x ) ) {
       currentPos += 2 * ++collisionNum - 1;
       currentPos = currentPos % array.length;
    }

    return currentPos;
}
```



bessere Performance als lineares Sondieren weil *primary Clustering* weniger auftritt.

Übung

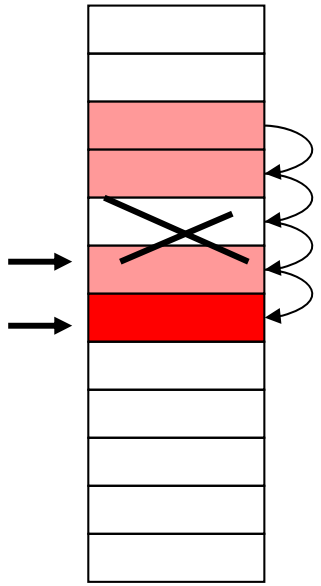
Gegeben sind:

- eine Hashtabelle der Grösse 10
- eine Hash-Funktion H(X) = X mod 10
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

Wie sieht die Tabelle aus, nachdem der Input unter Verwendung einer quadratischen Sondiermethode verarbeitet wurde?

Löschen in Hashtabellen

- Werte können nicht einfach gelöscht werden, da sie die Folge der Ausweichzellen unterbrechen.
- Wenn ein Wert gelöscht wird, müssen alle Werte, die potentielle Ausweichzellen sind, gelöscht und wieder eingefügt werden (rehashing).
- Zweite Möglichkeit: gelöschte Zelle lediglich als "gelöscht" markieren → Tombstones



Vor- und Nachteile von Hashing

Vorzüge

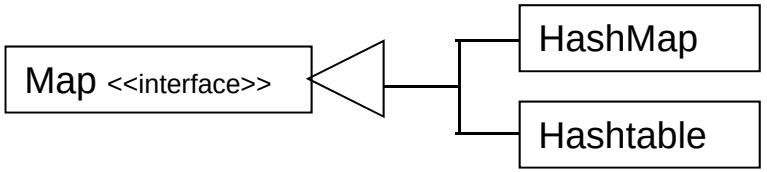
- Suchen und Einfügen in Hash-Tabellen sehr effizient
- binäre Bäume können je nach Inputdaten degenerieren, Hash-Tabellen kaum.
- Der Implementationsaufwand für Hash-Tabellen ist geringer als derjenige für ausgeglichene binäre Bäume.

Einschränkungen

- das kleinste oder grösste Element lässt sich nicht einfach finden
- Geordnete Ausgabe nicht möglich
- die Suche nach Werten in einem bestimmten Bereich oder das Finden z.B. eines Strings, wenn nur der Anfang bekannt ist, ist nicht möglich

Hash-Tabellen sind geeignet wenn:
- die **Reihenfolge** nicht von Bedeutung ist
- nicht nach **Bereichen** gesucht werden muss
- die **ungefähre (maximale) Anzahl** bekannt ist

Hashtable in Java



Hashtable ⇔	**HashMap**
seit java 1.0	seit java 1.2
synchronized	nicht synchronized
	Collections.synchronizedMap() verwenden
keine null-Werte	null-Werte erlaubt
zusätzliche Methoden	

Konstruktoren

`HashMap<K,V>()`	Konstruktor
`HashMap<K,V>(int initialCapacity)`	Konstruktor mit Grösse
`HashMap<K,V>(int initialCapacity, float loadFactor)`	Konstruktor mit Grösse und load Faktor

Methoden

`void clear()`	Löschen aller Elemente
`int size()`	Anzahl Elemente
`V put(K key, V value)`	Einfügen eines Elementes
`V get (Object key)`	Finden eines Elementes
`V remove(Object key)`	Löschen eines Elementes
`boolean containsKey(Object key)`	ist Element mit Schlüssel in Table
`boolean containsValue(Object value)`	hat ein Element den Wert
`Collection<V> values()`	alle Werte als Collection
`Set<K> keySet()`	alle Schlüssel als Set

Zusammenfassung

- Suche
  - Einfache Stringsuche
  - Suche in zwei Sammlungen
  - Binäres Suchen
- Maps und Sets
- Hashing
  - Idee
  - Hashfunktion
    - gute Hashfunktionen
  - Kollisionsauflösung
    - Überlauflisten
    - lineares Sondieren
    - quadratisches Sondieren
  - Vor- und Nachteile
  - Hashtabellen in Java