8. Hashing

Lernziele:

- Hashverfahren verstehen und einsetzen können,
- Vor- und Nachteile von Hashing gegenüber Suchbäumen benennen können,
- verschiedene Verfahren zur Auflösung von Kollisionen kennen, deren Funktionsweise nachvollziehen und erläutern können und
- Speicherverfahren, die auf Hashing beruhen, in Java implementieren können.

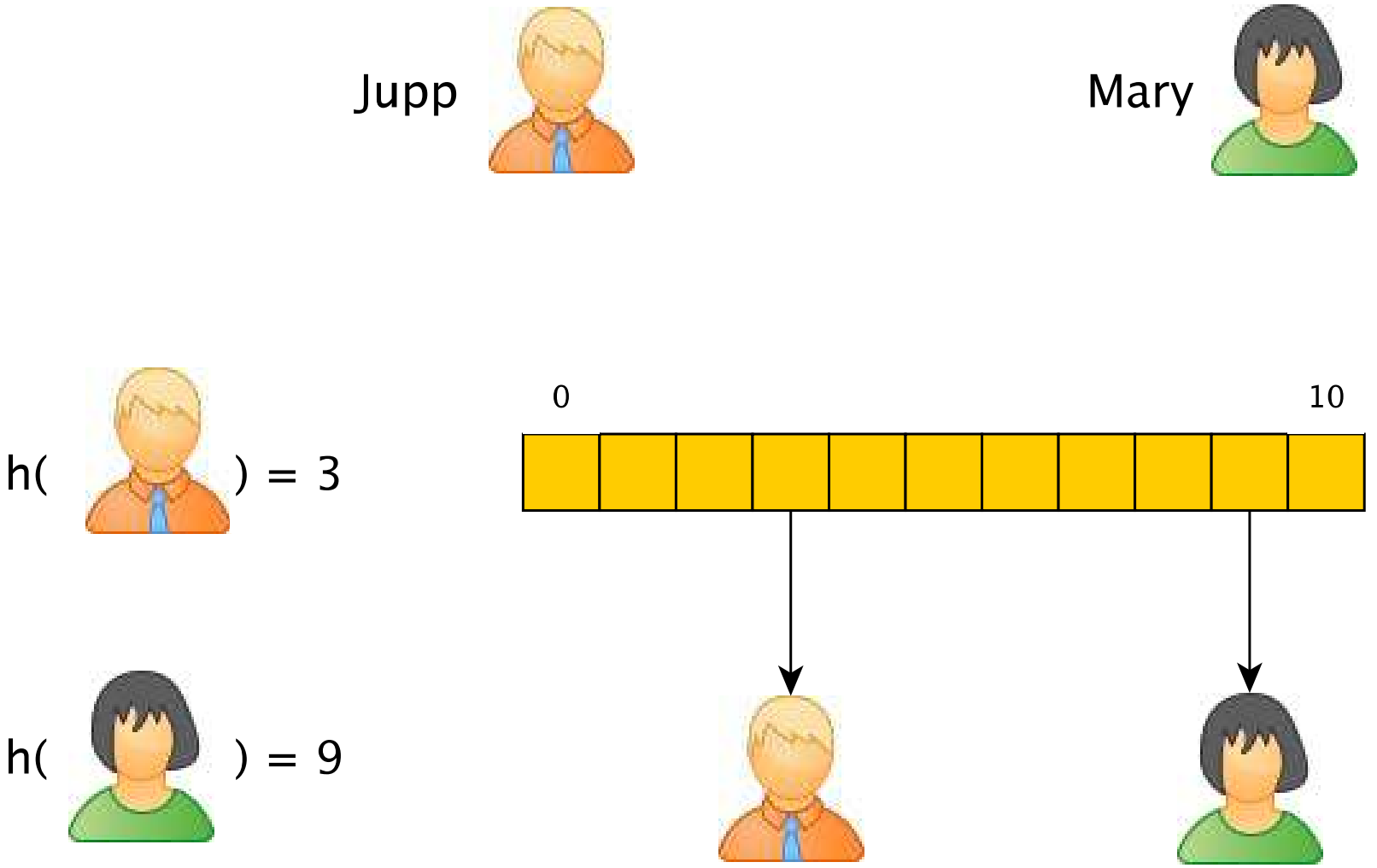
Idee des Hashing

- Hashing dient der Verwaltung einer Menge von Objekten vom Typ T (vgl. Folie 231).
- Objekte werden in einem "normalen" Feld mit direktem Zugriff gespeichert.
- Eine sogenannte Hashfunktion ermöglicht den direkten Zugriff auf ein Objekt.

☞ Statt einer ausgereiften Datenstruktur wie bei Bäumen benötigen wir hier eine ausgereifte Funktion zur Adressberechnung.

Grundprinzip

- Die Speicherung der Objekte erfolgt in einem Feld der Länge N, der sogenannten *Hashtabelle*.
- Eine Hashfunktion $h : T \to \{0, \ldots, N-1\}$ bestimmt für ein Objekt vom Typ T die Position des Objekts im Feld.
- Die Hashfunktion muss für eine "gute" Verteilung der zu speichernden Objekte in dem Feld sorgen.


Jupp
Mary
h( ) = 3
h( ) = 9
0
10

Kollisionen

Weil

- der Datentyp T (Menge der möglichen Objekte) in der Regel deutlich mehr als N Elemente enthält und
- die zu speichernden Objekte vorher unbekannt sind,

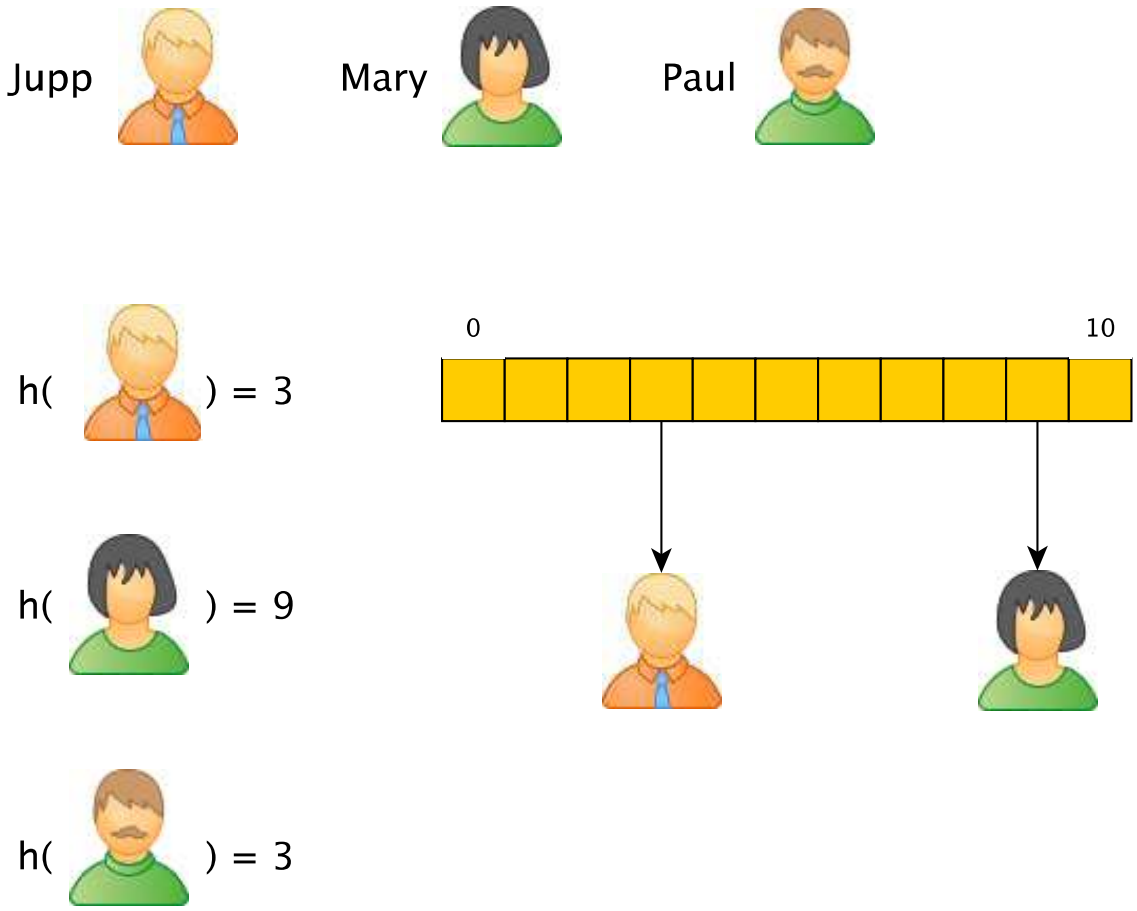
kann es zwangsläufig passieren, dass

- für zwei Objekte o_1 und o_2 mit $o_1 \neq o_2$
- $h(o_1) = h(o_2)$

gilt.

Solche eine Situation nennen wir *Kollision*.

Wohin mit Paul?



Anforderungen

- Die Hashfunktion sollte die zu speichernden Objekte möglichst gleichmäßig auf die Hashtabelle verteilen. So wird auch die Anzahl der Kollisionen gering gehalten.
- Wenn keine gleichmäßige Verteilung erfolgt, entartet das Hashverfahren, d.h. es kommt zu einer hohen Anzahl an Kollisionen.
- Da typischerweise $|T| > N$ gilt, lassen sich Kollisionen in der Regel nicht vermeiden.

Zentrale Fragen beim Hashing:

1. Was ist eine geeignete Hashfunktion?
2. Wie gehen wir mit Kollisionen um?

Anforderungen an eine Hashfunktion

- Effizienz der Berechnung
  Idealerweise in Zeit $O(1)$, insbesondere unabhängig von der Anzahl der Werte in der Hashtabelle
- Surjektivität
  $h(T) = \{0, \ldots, N-1\}$, so dass keine Plätze in der Hashtabelle leer bleiben.
- Kollisionsvermeidung bzw. Gleichverteilung
  Für zwei Objekte o_1 und o_2 sollte die Wahrscheinlichkeit, dass $h(o_1) = h(o_2)$ gilt, möglichst klein sein.
- Unstetigkeit
  Ähnliche Objekte sollten nicht zu ähnlichen Hashwerten führen.
- Effizienz des Speicherung
  Der Hashwert $h(o)$ eines Objektes $o \in T$ sollte möglichst kompakt gespeichert werden können.

Hashfunktionen für numerische Werte

Für nichtnegative ganze Zahlen wird oft der direkte Integer-Wert i in Verbindung mit dem Modulo-Operator als Hashfunktion genutzt.

$$h(i) = i \bmod N$$

- N (und damit die Größe der Hashtabelle) sollte eine große Primzahl sein.
- Für $2|N$ (d.h. 2 ist Teiler von N) wäre bspw. die Parität eine invariante Eigenschaft:
  - Wenn o_1 und o_2 die gleiche Parität hätten, dann auch $h(o_1)$ und $h(o_2)$.
  - Wenn es sich bei den Werten i z.B. um künstlich generierte Schlüssel handelt, die stets die gleiche Parität aufweisen, würde die Hälfte der Hashtabelle nicht belegt.
- Für $N = 2^k$ wäre $h(i)$ gleich dem Wert der letzten k Bits von i.
  - Problematisch z.B. bei der Verwendung von ungleich verteilten Prüfziffern.

- Vorsicht bei negativen Zahlen: Der `mod`-Operator entspricht in Java (und auch anderen Sprachen) nicht der mathematischen Definition.

  Mathematik: $-2 \bmod 3 = 1$, denn es gilt $-2 = (-1) \cdot 3 + 1$.
  Java: $-2 \% 3 = -2$, denn $-(2 \bmod 3) = -2$
- Für das Hashing von Fließpunktzahlen kann man z.B. Mantisse und Exponent addieren (oder auf andere Weise kombinieren).
- Da für die zu hashenden Integer-Werte meistens keine Gleichverteilung vorliegt, ist der `div`-Operator zur Konstruktion von Hashfunktionen ungeeignet.

  Beispiel: Matrikelnummern der Hochschule Bonn-Rhein-Sieg.

Hashfunktionen für Strings (1)

- ASCII-Werte der einzelnen Zeichen eines Strings nutzen. Für ein Zeichen $c \in \texttt{char}$ bezeichne $\texttt{ascii}(c)$ den ASCII-Wert des Zeichens c.
- Es sei $s = s[0]s[1] \cdots s[n-1]$ ein String der Länge n.
- Eine (meistens zu) einfache Hashfunktion:

$$h(s) = \left(\sum_{i=0}^{n-1} \texttt{ascii}(s[i]) \right) \mod N$$

Probleme:

- $h(s)$ liefert für alle Strings, die aus den gleichen Zeichen bestehen (sogenannte Anagramme), gleiche Hashwerte:
  - algorithmus und logarithmus

  - lager und regal
  - _regierung und genug_irre
- Bei kurzen Strings und großer Hashtabelle werden die hinteren Bereiche der Hashtabelle erst gar nicht erreicht. Es sei $N = 10000$:

$$\begin{aligned} h(\text{jupp}) &= 106 + 117 + 112 + 112 = 447 \\ h(\text{mary}) &= 109 + 97 + 114 + 121 = 441 \\ h(\text{paul}) &= 112 + 97 + 117 + 108 = 434 \end{aligned}$$

  Für Namen zwischen 3 und 10 Zeichen würde sich alles zwischen $3 \cdot \texttt{ascii}(\text{a}) = 291$ und $10 \cdot \texttt{ascii}(\text{z}) = 1220$ ballen.

Hashfunktionen für Strings (2)

Häufig nutzt man für ein festes $x \in \mathbb{N}$ die Hashfunktion:

$$h(s) = \left(\sum_{i=0}^{n-1} \texttt{ascii}(s[i]) \cdot x^{n-i-1} \right) \mod N$$

- Polynom vom Grad $n-1$ mit den ASCII-Werten der Zeichen $s[i]$ als Koeffizienten, ausgewertet an der Stelle x.
- Für $x = 1$ ist dies die Hashfunktion von Folie 340.
- Für $x = 2$ ergibt sich

$$\begin{aligned} h(\text{jupp}) &= 106 \cdot 8 + 117 \cdot 4 + 112 \cdot 2 + 112 = 1652 \\ h(\text{mary}) &= 109 \cdot 8 + 97 \cdot 4 + 114 \cdot 2 + 121 = 1609 \\ h(\text{paul}) &= 112 \cdot 8 + 97 \cdot 4 + 117 \cdot 2 + 108 = 1626 \end{aligned}$$

was schon etwas stärker streut.

- Beliebt ist $x = 2^8$, auch wegen effizienter Berechnung durch Bitoperationen (s.u.):

$$\begin{aligned} h(\text{jupp}) &= (106 \cdot 2^{24} + 117 \cdot 2^{16} + 112 \cdot 2^8 + 112) \bmod 10000 = 1392 \\ h(\text{mary}) &= (109 \cdot 2^{24} + 97 \cdot 2^{16} + 114 \cdot 2^8 + 121) \bmod 10000 = 2841 \\ h(\text{paul}) &= (112 \cdot 2^{24} + 97 \cdot 2^{16} + 117 \cdot 2^8 + 108) \bmod 10000 = 5244 \end{aligned}$$

  Aber wegen der Zweierpotenz ist dies auch nicht ganz unproblematisch.

- Java nutzt zur eingebauten Berechnung von Hashwerten (Methode `hashCode()`) für Strings $x = 31 = 2^5 - 1$. Dies lässt sich ebenfalls effizient berechnen (s.u.).

Effiziente Berechnungen beim String-Hashing

Der Term

$$\sum_{i=0}^{n-1} \texttt{ascii}(s[i]) \cdot x^{n-i-1}$$

der Hashfunktion wird nicht via Definition berechnet, sondern über das sogenannte *Horner-Schema*.

Auswertung nach dem Horner-Schema:

- Gegeben sei das Polynom

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Dies ist identisch zu der folgenden Form

$$f(x) = (\cdots (a_n x + a_{n-1})x + \cdots)x + a_0$$

- Beispiel:

$$\begin{aligned} f(x) &= 4x^3 + 5x^2 + x + 2 \\ &= ((4x + 5)x + 1)x + 2 \end{aligned}$$

- Vorteil: Es müssen keine Potenzen gebildet werden, nur $n - 1$ statt $2n - 1$ Multiplikationen
- Java-Methode zur Polynomauswertung:

```
public static double f(double[] a, double x) {
    double fx = 0.0;                // Funktionswert f(x) an der Stelle x

    for (int i=a.length-1 ; i>=0 ; i--) {
        fx = fx * x + a[i];
    }
    return fx;
}
```

Weitere Optimierungen

- Für z.B. $x = 256 = 2^8$ oder $x = 31 = 2^5 - 1$ kann die Berechnung durch Bitoperationen (siehe EidP, Folien 119–123) noch effizienter erfolgen.
- Die Multiplikation mit 2^8 entspricht einer Verschiebung aller Bits um acht Positionen nach links. Operator <<
- Nach der Verschiebung haben die acht niedrigstwertigen Bits alle den Wert 0.
- Eine Addition mit einer Zahl $0 \leq a \leq 255$ entspricht dann einer bitweisen Oder-Verknüpfung. Operator |

```
public static int f(char[] s) {
    int hash = 0;

    for (int i=0 ; i<s.length ; i++) {
        hash = (hash << 8) | s[i];
    }
    return hash;
}
```

- Die Multiplikation eines Wertes i mit 31 lässt sich durch eine Linksverschiebung um 5 Bits und eine Subtraktion von i ausdrücken.

```
i = (i << 5) - i;
```

- Manche Prozessoren benötigen für die gesamte Berechnung nur eine Instruktion (z.B. ARM).
- Insgesamt:

```
public static int f(char[] s) {
    int hash = 0;

    for (int i=0 ; i<s.length ; i++) {
        hash = (hash << 5) - hash + s[i];
    }
    return hash;
}
```

Hashfunktionen für sonstige Objekte

Für Objekt o vom Typ T bestehend aus Teilobjekten o_1 bis o_k mit Typen $T_1, \ldots, T_k$:

- Nutze existierende Hashfunktionen h_i für die Typen T_i und
- aggregiere die einzelnen Hashwerte auf angemessene Weise, z.B. durch Summenbildung.

$$h(o) = \left(\sum_{i=1}^{k} h_i(o_i) \right) \mod N$$

Ansätze zur Behandlung von Kollisionen

Verkettung: Objekte mit gleichem Hashwert werden in einer verketteten Liste gespeichert.

- Die Hashtabelle enthält also nicht die eigentlichen Objekte sondern Verweise auf Listen, die die Objekte enthalten.

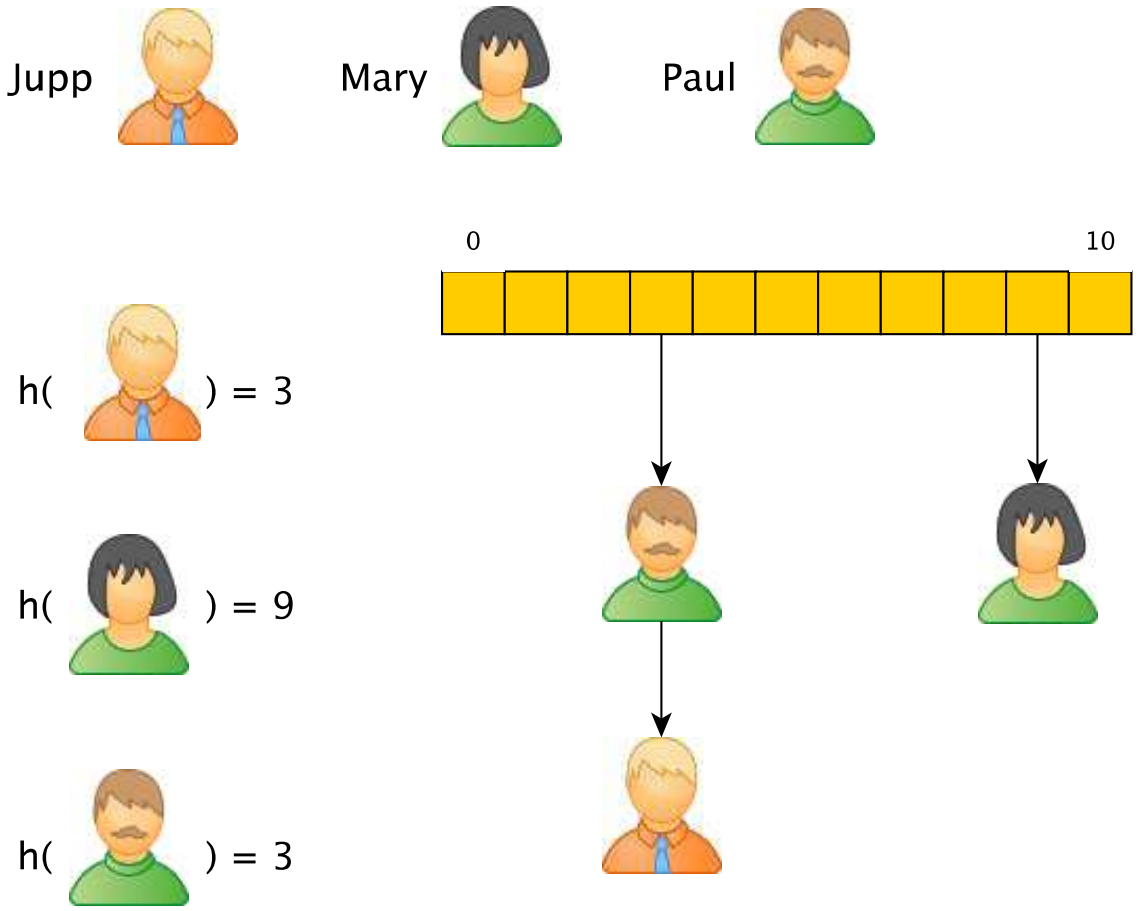
Offene Adressierung: Es wird nach einer alternativen Position in der Hashtabelle gesucht.

- Beim Sondieren geschieht dies durch eine vordefinierte Folge von Offsets relativ zur eigentlichen Position.
- Beim mehrfachen Hashen werden weitere Hashfunktionen für eine alternative Positionsbestimmung genutzt.

Verkettung

- Unter einem Überläufer versteht man ein Objekt, das zu einer Kollision führt.
- Bei der Verkettung wird das Objekt an einer Position der Hashtabelle zusammen mit allen Überläufern in einer verketteten Liste gespeichert.
- Die Verkettung von Überläufern kann zu einer linearen Liste "entarten", wenn die Mehrzahl der Elemente durch die Hashfunktion auf dieselbe Position abgebildet wird.
- Zur Verhinderung der Entartung kann statt einer Liste daher auch ein ausgeglichener Suchbaum verwendet werden.

Dahin mit Paul!



Hashing mit Verkettung in Java

```
public class HashSet<T> {

    private static final int N = ...;

    private class Item {       // Element der verketteten Liste besteht aus
        T    value;            // Verweis auf gespeichertes Objekt und
        Item next;             // Verweis auf naechstes Element der Liste
    }

    private Item[] hashTable = new Item[N];     // die Hashtabelle

    public void insert(T elem) {
        int  i = elem.hashCode() % N;     // setzt hashCode() >= 0 voraus
        Item item = new Item();

        item.value = elem;                // neues Objekt wird das erste
        item.next = hashTable[i];         // in der verketteten Liste
        hashTable[i] = item;
```

```
    }

    public boolean contains(T elem) {
        Item item = hashTable[elem.hashCode() % N];  // setzt hashCode() >= 0 voraus
                                                     // nach Element suchen
        while (item != null && !item.value.equals(elem)) {
            item = item.next;
        }
        return item != null;          // wenn gefunden, dann ist item != null
    }

    public void remove(T elem) {
        int  i = elem.hashCode() % N;
        Item item = hashTable[i];          // setzt hashCode() >= 0 voraus
        Item pre = null;                   // Vorgaenger in Liste
                                                     // nach Element suchen
        while (item != null && !item.value.equals(elem)) {
            pre = item;
            item = item.next;
        }
        if (item != null) {                // nur wenn gefunden
            if (pre == null) {             // wenn erstes Element in Liste
```

```
                hashTable[i] = item.next; // dann Eintrag in Hashtabelle mit Nachfolger belegen
            }
            else {                        // ansonsten
                pre.next = item.next;     // Vorgaenger auf Nachfolger zeigen lassen
            }
        }
    }
}
```

Sondieren

- Kommt es zu einer Kollision, dann suchen wir für den Überläufer nach einer anderen, noch unbesetzten Position in der Hashtabelle gesucht.
- Das Suchen nach solch einer Position bezeichnen wir als *Sondieren*.
- Für das Sondieren nutzt man eine Offset-Folge $\text{offset}(i)$ für $i = 1, 2, \ldots$
- Mit der Offset-Folge wird für die Positionen

$$(h(o) + \text{offset}(i)) \bmod N$$

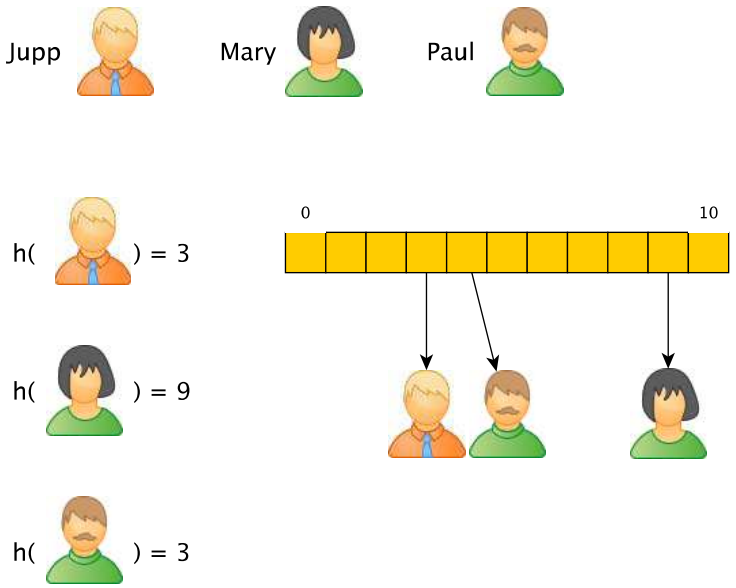
  geprüft, ob sie noch frei sind.
- Beim kleinsten i mit einem freien Platz wird das Objekt o in die Hashtabelle eingefügt.

Lineares Sondieren

- Beim Linearen Sondieren gilt

$$\text{offset}(i) = i$$

  Es wird also linear nach dem nächsten freien Platz gesucht.
- Problem: Dadurch kann es leicht zu Ballungen um belegte Plätze herum kommen.



Beispiel: Lineares Sondieren

	leer	insert 89	insert 18	insert 49	insert 58	insert 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

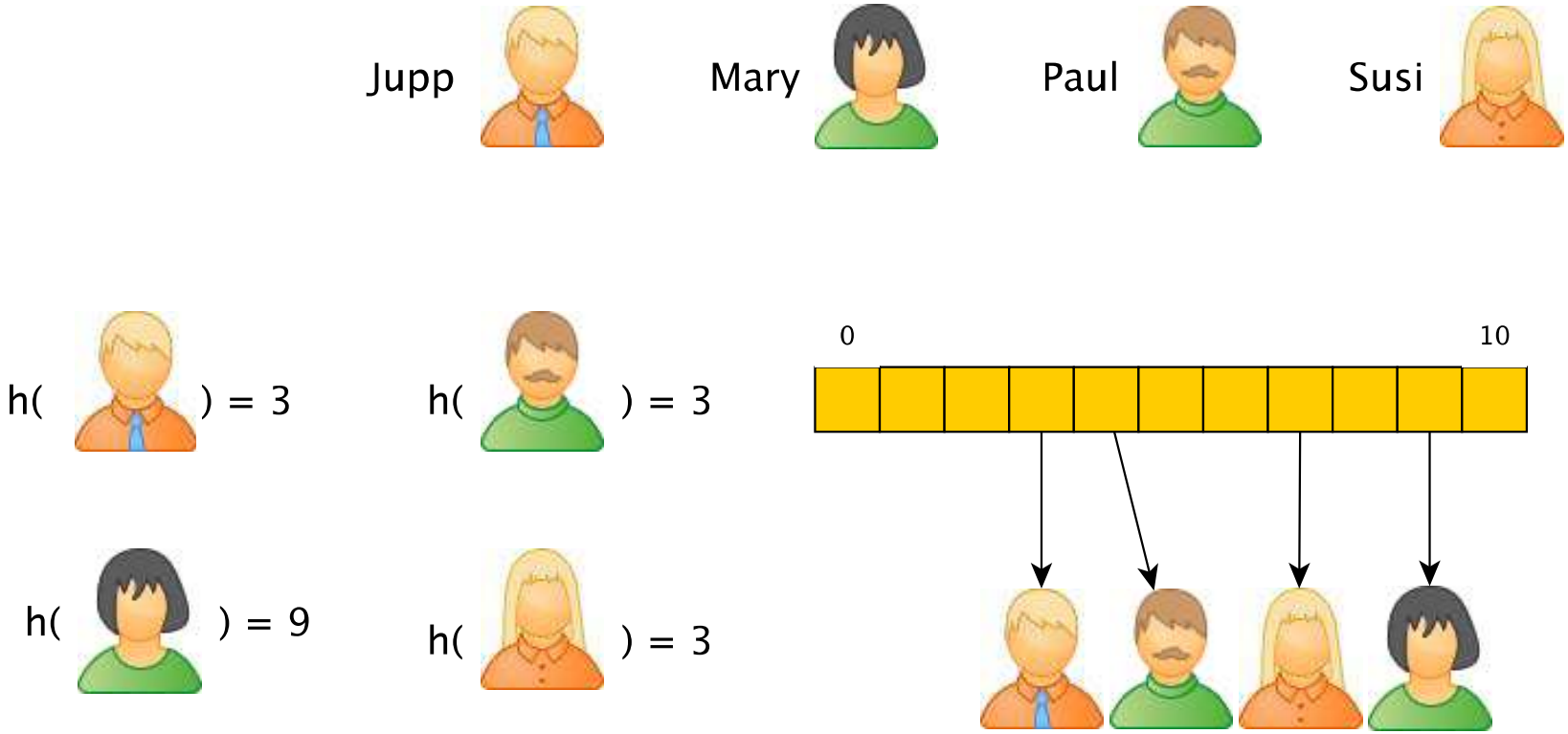
Quadratisches Sondieren

- Beim Quadratischen Sondieren gilt

$$\text{offset}(i) = i^2$$

- Vorteil: Ballungen um belegte Plätze sind weniger wahrscheinlich.
- Nachteil: Beim Sondieren werden u.U. nicht alle Plätze der Hashtabelle berücksichtigt.
  - Ist bspw. N eine Quadratzahl, so werden beim quadratischen Sondieren u.U. nur $\sqrt{N} - 1$ alternative Positionen der Hashtabelle untersucht. Beispiel: 16
  - Wenn N eine Primzahl ist, kann aber garantiert werden, dass mindestens $N/2$ Positionen untersucht werden (s.u.).

Dahin mit Paul und Susi!



Beispiel: Quadratisches Sondieren

	leer	insert 89	insert 18	insert 49	insert 58	insert 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Satz zum quadratischen Sondieren

Satz 8.1. *Es sein* N *die Größe einer Hashtabelle.*

Wenn N *eine Primzahl ist, dann werden beim quadratischen Sondieren mindesten* $N/2$ *alternative Positionen geprüft.*

Beweis: Es sei j die kleinste natürliche Zahl, für die beim quadratischen Sondieren eine schon untersuchte Position wieder untersucht wird. Es sei i diejenige natürliche Zahl, die zur ersten Untersuchung dieser Position gehört.

Dann gilt:

$$\begin{aligned} \text{offset}(i) \bmod N = \text{offset}(j) \bmod N \quad &\Leftrightarrow \quad i^2 \bmod N = j^2 \bmod N \\ &\Leftrightarrow \quad j^2 - i^2 \bmod N = 0 \\ &\Leftrightarrow \quad (j-i) \cdot (j+i) \bmod N = 0 \end{aligned}$$

- Daraus folgt, dass $(j - i) \cdot (j + i)$ gleich N oder ein Vielfaches von N sein muss, d.h.

$$\exists c \in \mathbb{N} : (j - i) \cdot (j + i) = c \cdot N$$

- Es gilt $j - i < N$, denn es gibt nur $N - 1$ alternative Positionen.
- Somit muss, da N eine Primzahl ist, $(j + i)$ gleich N oder ein Vielfaches von N sein.
- Wegen $j > i$ folgt hieraus $j > N/2$.

Diskussion Sondieren

Das Sondieren verkompliziert Such-, Einfüge- und Löschoperationen:

- Beim Suchen nach Objekt o: Befindet sich an Position $i = h(o)$ ein Objekt $o' \neq o$, dann wurde o mittels Sondieren u.U. an einer anderen Position platziert. Wir müssen bei der Suche also ebenfalls Sondieren.

  Bemerkung: Wenn der Eintrag an der Stelle i der Hashtabelle leer ist, dann kann o nicht enthalten sein.
- Beim Löschen von Objekt o: Ein einfaches Löschen könnte die Suchkette für andere Elemente zerstören.

  Deshalb: nicht Löschen, sondern nur als gelöscht markieren und Platz beim Einfügen wiederverwenden.

☞ Sondierung bzw. offene Adressierung nur dann sinnvoll, wenn Löschen selten ist.

Hashing in Java: Die Methode `hashCode()`

- In Java verfügt jedes Objekt über eine Methode `hashCode()` zur Berechnung eines Hash-Wertes.
- Durch Bitverschiebung oder einen Überlauf bei den Integer-Operationen können auch negative Hashwerte als Resultat der Methode `hashCode()` entstehen (vgl. Hashfunktionen für Strings).
- Auch die Modulo-Operation zur Abbildung auf die Hashtabelle wird nicht durch `hashCode()` durchgeführt.
- In der Klasse `Object` wird der Hashwert auf Basis der Speicheradresse ermittelt.
- Für die Definition eigener Hashfunktionen überschreibt man in den betreffenden Klassen die Methode `hashCode()`.

Hashing in Java: Collection Classes

Alles im Paket `java.util`:

- Klasse `Hashtable<K,V>`
  - Bildet mit Hilfe einer Hashtabelle Schlüssel vom Typ `K` auf Objekte vom Typ `V` ab. Interface für solch eine Art der Abbildung: `Map<K,V>`
  - Der Typ `K` muss die Methoden `hashCode()` und `equals()` geeignet implementieren.
  - `null` ist nicht erlaubt, sowohl als Schlüssel als auch als Objekt.
  - Die Hashtabelle wird automatisch vergrößert, wenn sie zu voll wird. Größe und Vergrößerung können durch Parameter gesteuert werden.
  - Ist `synchronized`, d.h. Parallelverarbeitung wird unterstützt.
- Klasse `HashMap<K,V>`
  - Leistet vom Prinzip das gleiche wie `Hashtable`.
  - Erlaubt im Gegensatz zu `Hashtable` aber `null` als Schlüssel bzw. als Objekt.
  - Unterstützt keine Parallelverarbeitung.

- Klasse `HashSet<E>`
  - Realisierung einer Menge von Objekten des Typs `E` mit Hilfe einer Hashtabelle.
  - Implementiert die Schnittstelle `Set<E>`.
  - Basiert auf `HashMap`.