Algorithmik

Sebastian Iwanowski
FH Wedel

3. Lösungen für das Wörterbuchproblem
3.1 Hashing und andere für das durchschnittliche Laufzeitverhalten konzipierte Verfahren

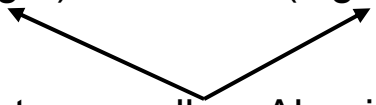
Algorithmik 3

Implementierung von Dictionaries

Ein Dictionary ist eine Datenstruktur für mit einem Schlüssel vergleichbare Elemente,
welche die Funktionen member (key), insert (key, newdata) und delete (key) zur Verfügung stellt

Sortiertes Feld als Dictionary:

member (key) — Laufzeit Θ(log n) w.c. und Θ(log log n) a.c. erreichbar ☺



nicht vom selben Algorithmus

insert (key, newdata) — Laufzeit Θ(n) w.c. und a.c. ☹

delete (key) — Laufzeit Θ(n) w.c. und a.c. ☹

Bessere Methode für insert / delete mit indizierten Feldern: Hashing (Folgefolien)

Referenzen zum Nacharbeiten und Vertiefen:

Skript Alt, S. 30 – 35 (für member, Wiederholung vom letzten Mal)

Welches Problem löst Hashing ?

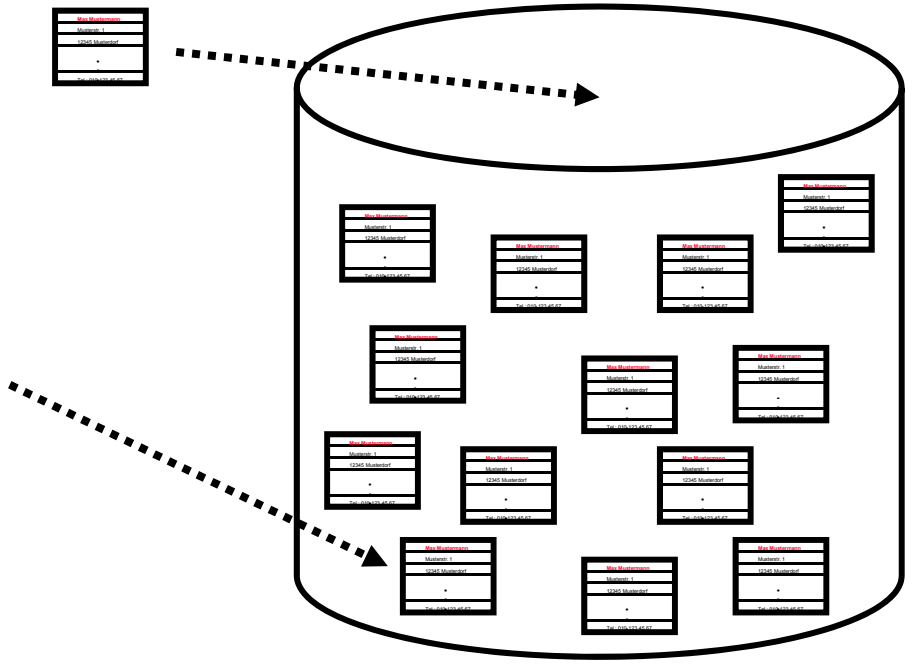
Datensatz:

Suchschlüssel

identifiziert
Datensatz
eindeutig

Max Mustermann
Musterstr. 1
12345 Musterdorf
• • •
Tel.: 010 123 45 67

Datenbank:



`key` `value`

Datenverwaltungs-Operationen: `Map operations`

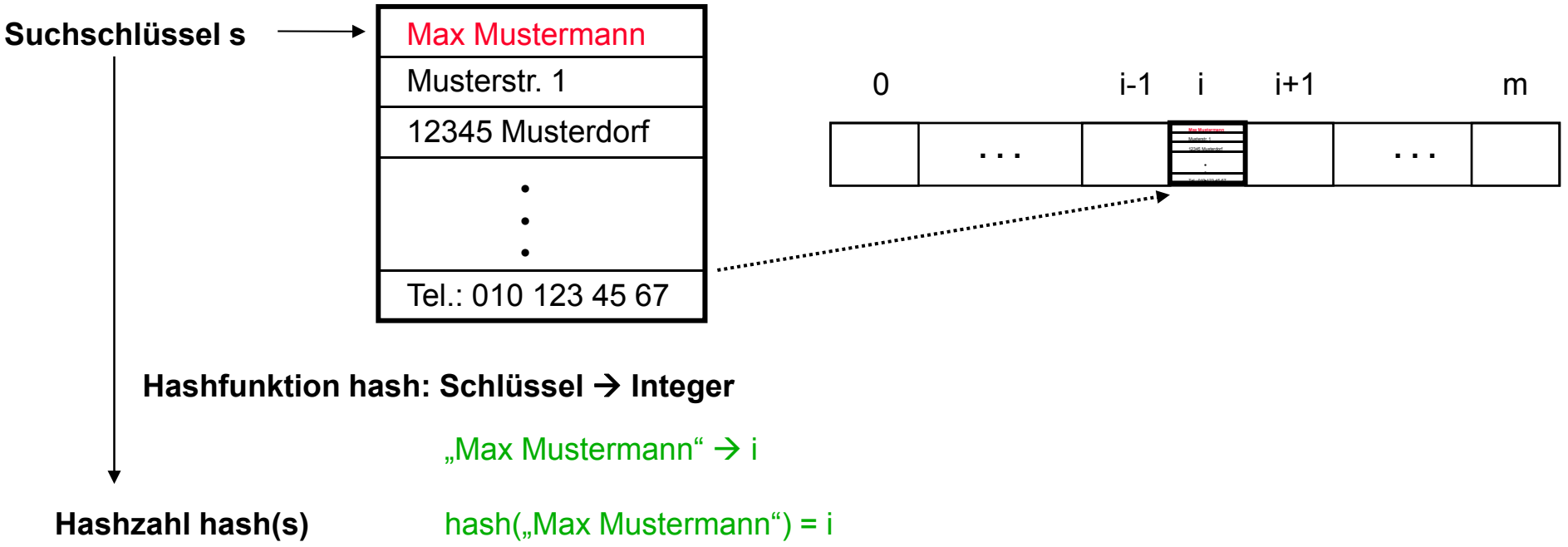
- **Suchen** `get (key)`
- **Einfügen** `put (key, value)`
- **Entfernen** `remove (key)`

→ ***Hashing*** **ist ein Verfahren, das diese Operationen effizient löst.**

Überblick über die Funktionsweise

Datensatz:

Hashtabelle T:



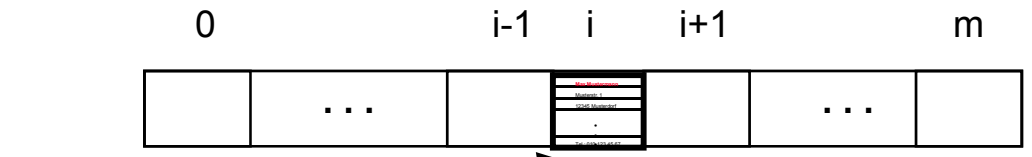
- **Suchen** — Bestimme i=hash(s) → In T[i] steht der gesuchte Datensatz
- **Einfügen** — Bestimme i=hash(s) → Speichere neuen Datensatz in T[i]
- **Entfernen** — Bestimme i=hash(s) → Lösche Datensatz in T[i] (`T[i] = null`)

Aufwerfen von Detailfragen

Datensatz:

Max Mustermann
Musterstr. 1
12345 Musterdorf
• • •
Tel.: 010 123 45 67

Hashtabelle T:



1) **Wie definiert man eine gute Hashfunktion ?**

2) **Wo speichert man den Datensatz in der Hashtabelle ab ?**

1) Wie definiert man eine gute Hashfunktion ?

Fall 1: **Es gibt mindestens so viele Plätze in der Hashtabelle wie verschiedene Schlüssel**

Ziel: **Jeder Schlüssel wird auf eine *andere* Hashzahl abgebildet.**

Perfektes Hashing

Lösung: **Ordne die Schlüssel nach einer Reihenfolge (z.B. lexikographisch) !**

Bilde jeden Schlüssel auf seine Ordnungszahl ab !

Beispiel:
(für Strings als Schlüssel)

„Max Mustermann“ → (13 1 24 0 13 21 19 20 5 18 13 1 14 14)

$$\text{hash}(\text{„Max Mustermann“}) = 13*27^{13} + 1*27^{12} + 24*27^{11} + 0*27^{10} + 13*27^{9} + 21*27^{8} + 19*27^{7} + 20*27^{6} + 5*27^{5} + 18*27^{4} + 13*27^{3} + 1*27^{2} + 14*27^{1} + 14*27^{0}$$

$$\approx 529668343500000000000 \text{ (20-stellige Zahl)}$$

Im Allgemeinen sind sehr viele *verschiedene* Schlüssel möglich !

Fazit: **Fall 1 ist unrealistisch !**

1) Wie definiert man eine gute Hashfunktion ?

Fall 2: **Es gibt weniger Plätze in der Hashtabelle als verschiedene Schlüssel.**

m k

Fall 2: $m < k$

Folgerung: **Verschiedene Schlüssel müssen auf die gleiche Hashzahl abgebildet werden**

Kollision

Ziel:

Auf jede Hashzahl zwischen 0 und m-1 werden ungefähr gleich viele Schlüssel abgebildet. (nämlich ungefähr k/m)

Lösung: **Ordne die Schlüssel nach einer Reihenfolge (z.B. lexikographisch) !**

Bilde jeden Schlüssel auf seine Ordnungszahl modulo m ab !

1) Wie definiert man eine gute Hashfunktion ?

Beispiel:

(für Strings als Schlüssel)

m = 1000 „Antje“ → (1 14 20 10 5)

hash („Antje“) = $(1 * 27^4 + 14 * 27^3 + 20 * 27^2 + 10 * 27^1 + 5) \bmod 1000$

= 821858 mod 1000

= 858

Algorithmus für eine gute Hashfunktion (nach *Horner*schema) :

hash(„Antje“) = ((((1 * 27 + 14) * 27 + 20) * 27 + 10) * 27 + 5) mod 1000

= (((((((1 mod 1000 * 27 + 14) mod 1000) * 27 + 20) mod 1000) * 27 + 10) mod 1000) * 27 + 5) mod 1000

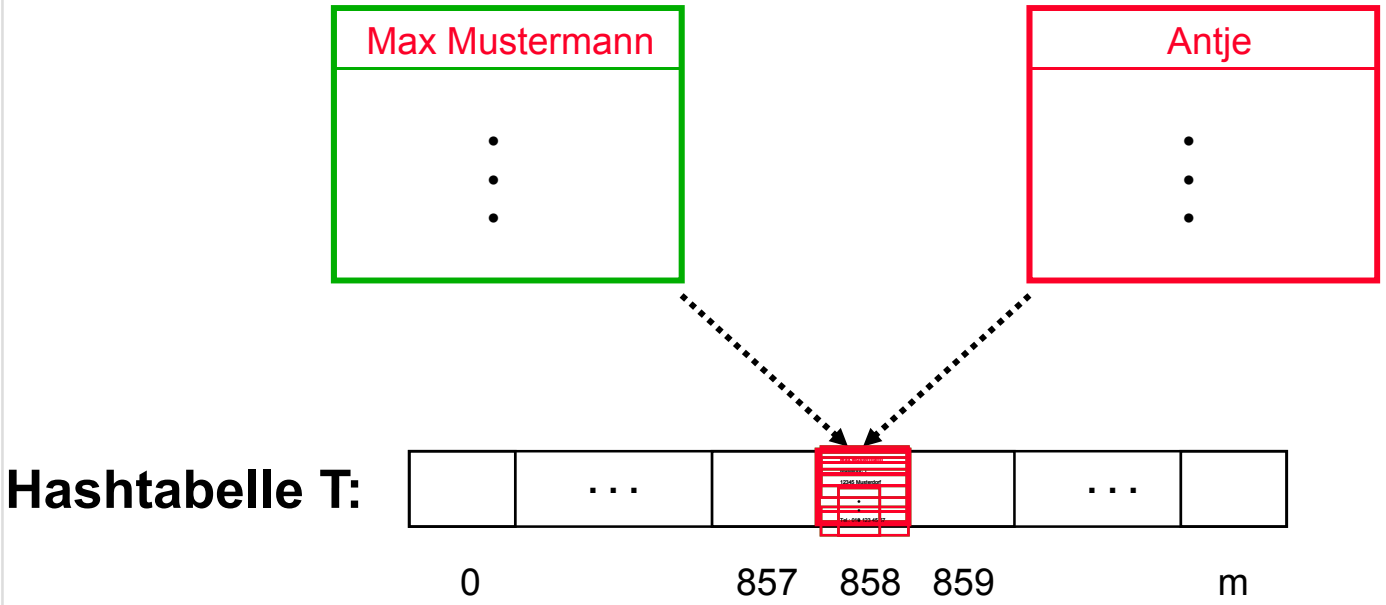
Java-Code:

```
static int hash (String key, int m)
   {
      int result = 0, numberSymbols = 27;
      for (int i = 0; i < key.length(); i++)
         result = (result*numberSymbols + order (key.charAt(i))) % m;
      return result;
   }
```

2) Wo speichert man den Datensatz in der Hashtabelle ab ?

Problem: Wie verfährt man bei Kollisionen ?

Beispiel: hash („Max Mustermann“) = 858
hash („Antje“) = 858

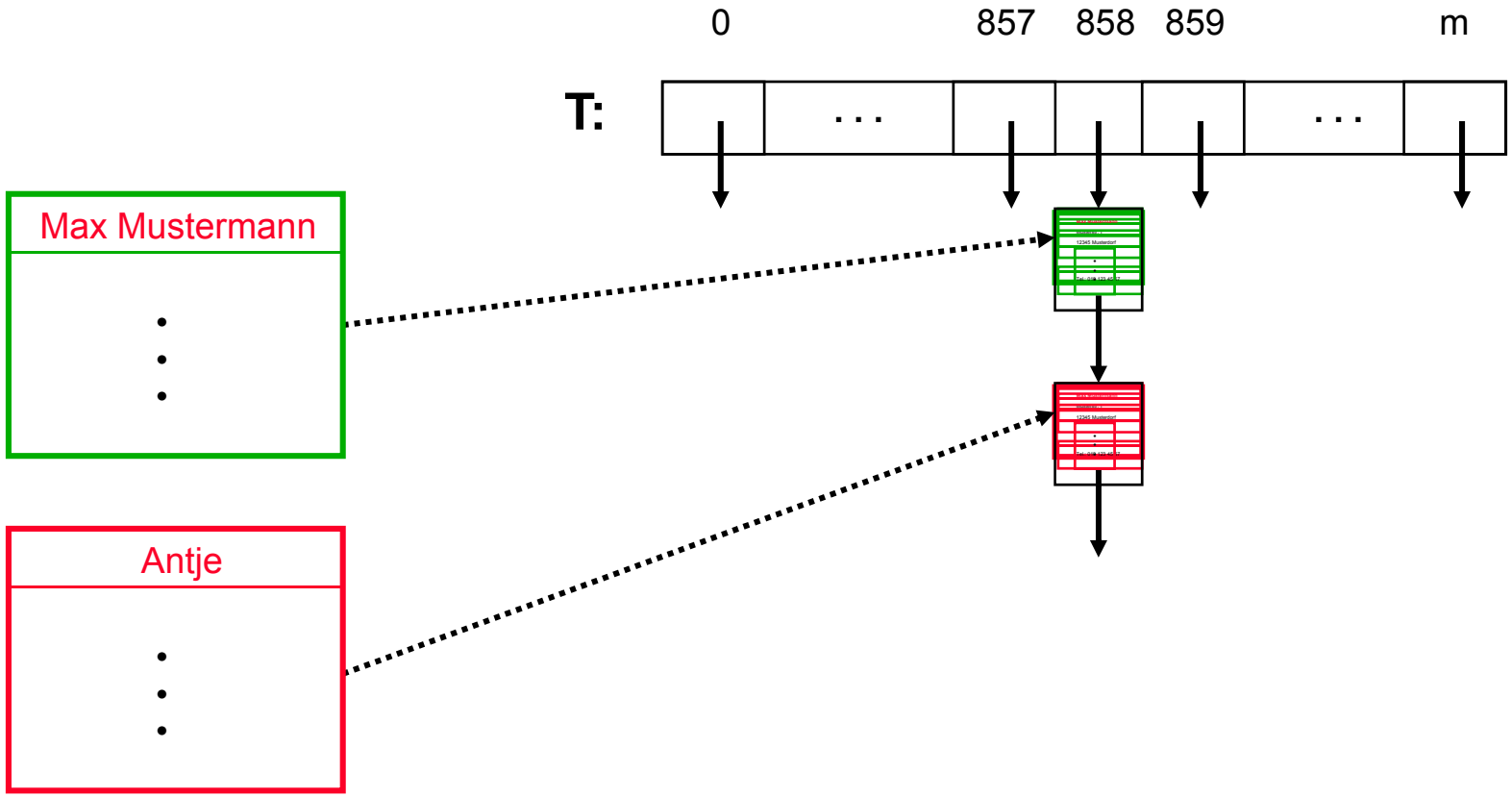


Hat jemand eine bessere Idee ?

2) Wo speichert man den Datensatz in der Hashtabelle ab ?

Problem: Wie verfährt man bei Kollisionen ?

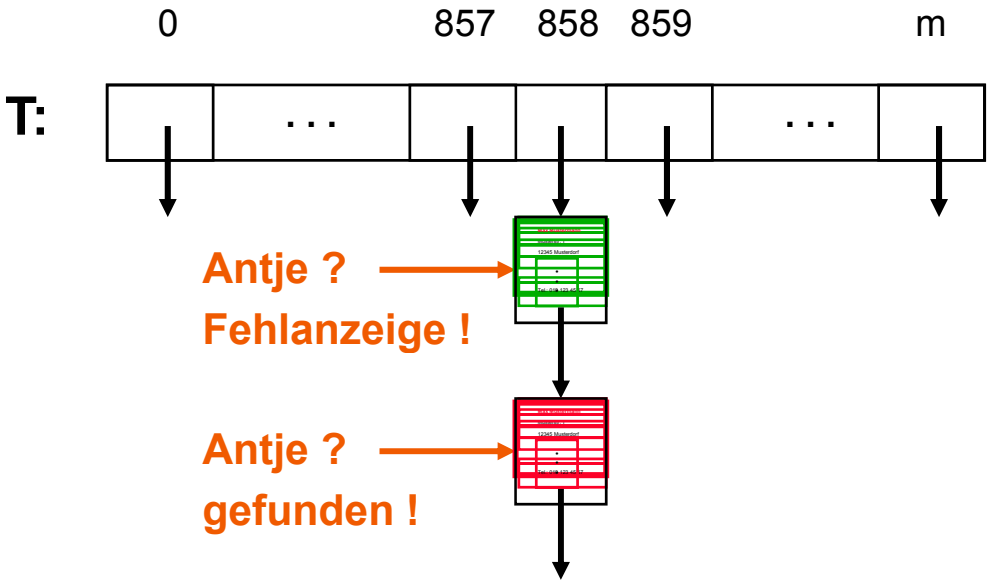
Lösung: **T[i] enthält Zeiger auf verkettete Liste von Datensätzen, deren Schlüssel alle die Hashzahl i haben.**



2) Wo speichert man den Datensatz in der Hashtabelle ab ?

Problem: Wie verfährt man bei Kollisionen ?

Lösung: **T[i] enthält Zeiger auf verkettete Liste von Datensätzen, deren Schlüssel alle die Hashzahl i haben.**



Suchen: **Antje ?**

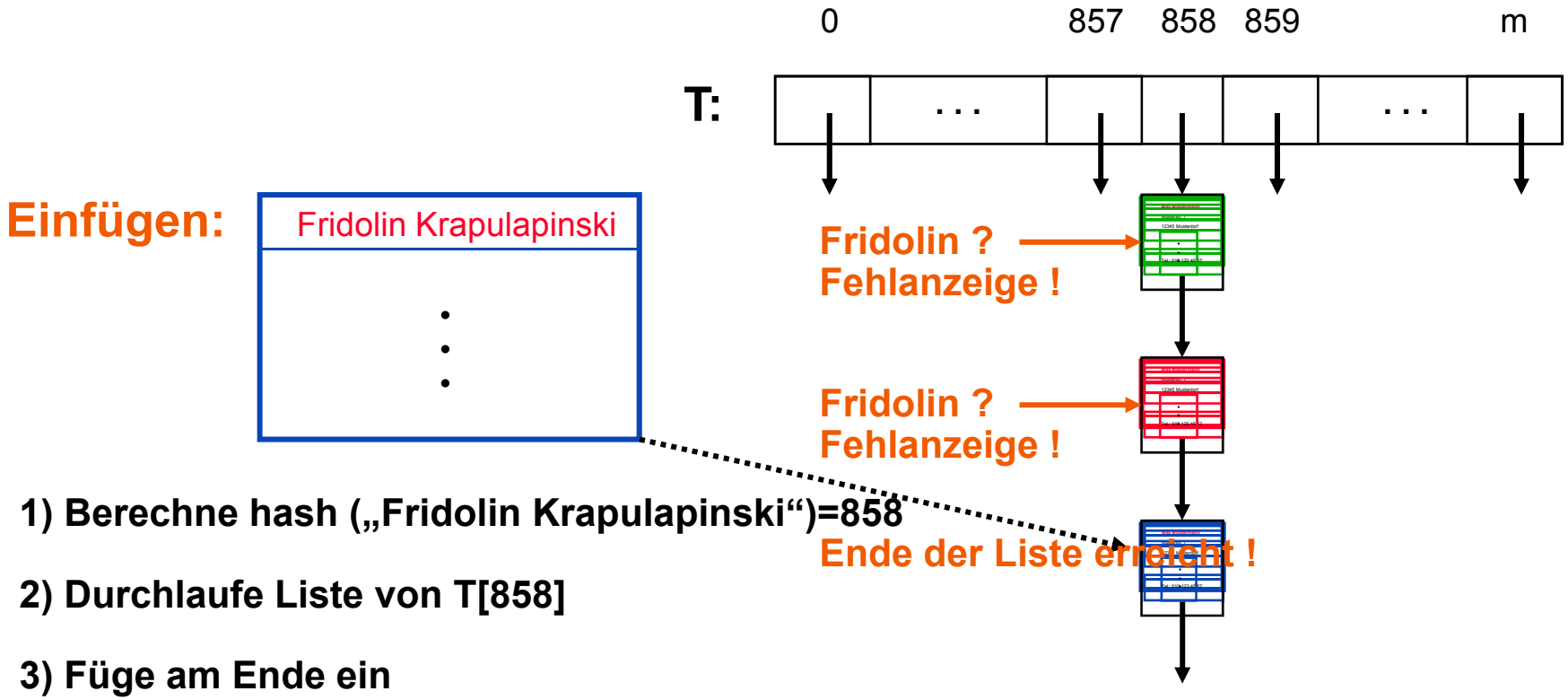
1) Berechne hash („Antje“)=858

2) Durchsuche Liste von T[858]

2) Wo speichert man den Datensatz in der Hashtabelle ab ?

Problem: Wie verfährt man bei Kollisionen ?

Lösung: **T[i] enthält Zeiger auf verkettete Liste von Datensätzen, deren Schlüssel alle die Hashzahl i haben.**



1) Berechne hash („Fridolin Krapulapinski“)=858

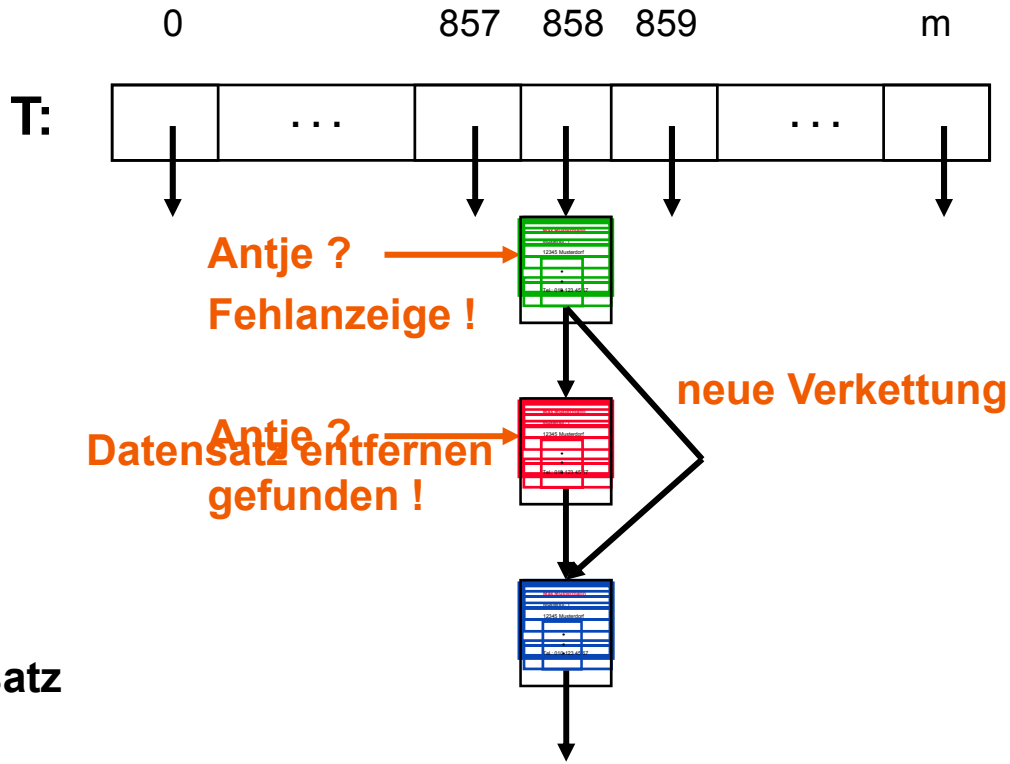
2) Durchlaufe Liste von T[858]

3) Füge am Ende ein

2) Wo speichert man den Datensatz in der Hashtabelle ab ?

Problem: Wie verfährt man bei Kollisionen ?

Lösung: **T[i] enthält Zeiger auf verkettete Liste von Datensätzen, deren Schlüssel alle die Hashzahl i haben.**



Entfernen: **Antje**

1) Berechne hash („Antje“)=858

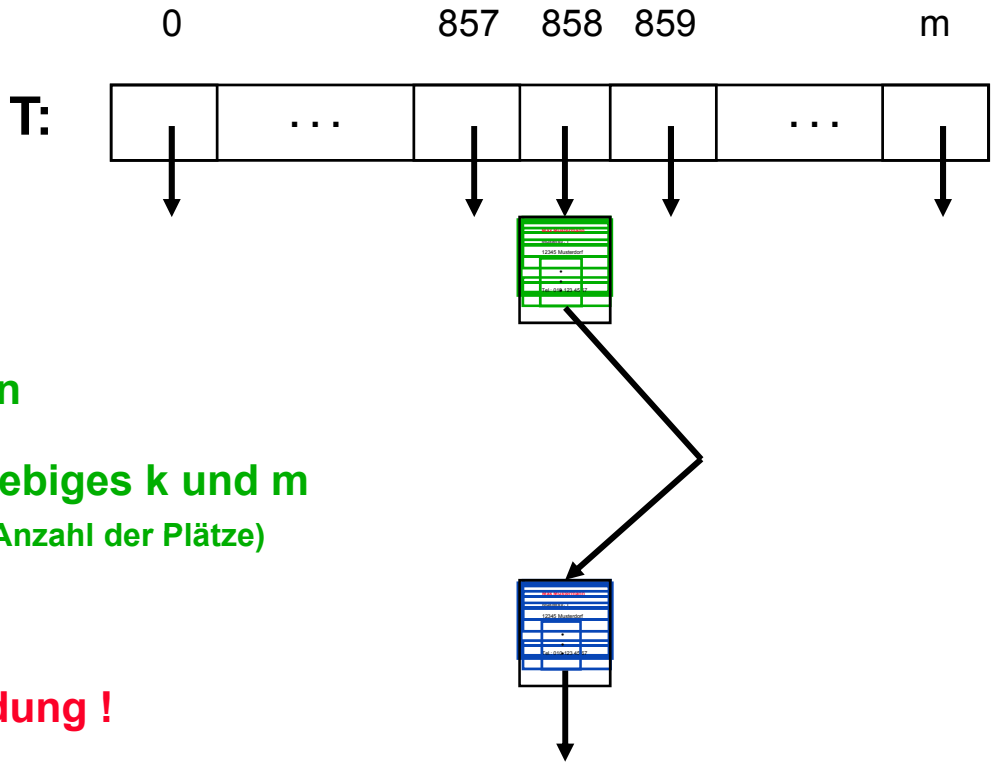
2) Durchlaufe Liste von T[858]

3) Entferne entsprechenden Datensatz

2) Wo speichert man den Datensatz in der Hashtabelle ab ?

Problem: Wie verfährt man bei Kollisionen ?

Lösung: **T[i] enthält Zeiger auf verkettete Liste von Datensätzen, deren Schlüssel alle die Hashzahl i haben.**



Bewertung des bisher vorgestellten Verfahrens zur Abspeicherung:

- einfach zu implementieren
- realisiert Hashing für beliebiges k und m
  (k: Anzahl der Schlüssel, m: Anzahl der Plätze)

Kritik:

- Speicherplatzverschwendung !

2) Wo speichert man den Datensatz in der Hashtabelle ab ?

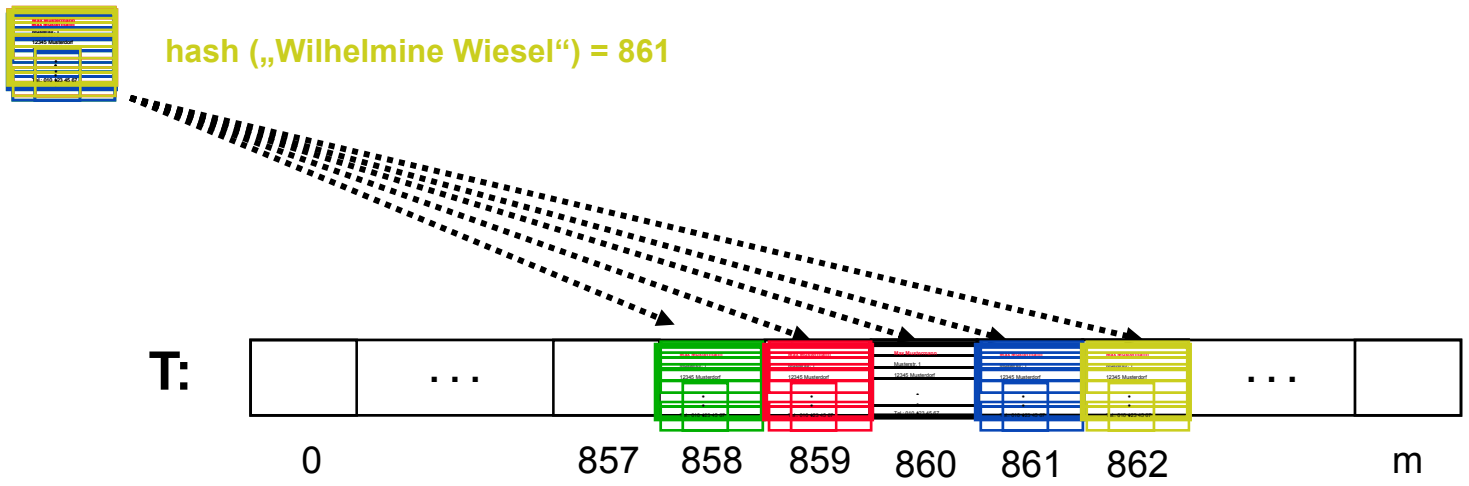
Problem: Wie verfährt man bei Kollisionen ?

Alternative Lösung:
(Offenes Hashing)

Suche anderen leeren Platz in der Hashtabelle:
Rücke von T[i] jeweils nach einer festen Regel weiter,
bis ein freier Platz gefunden wird

Sondierungsregel

Beispiel für Sondierungsregel: **rücke um eins nach rechts**



2) Wo speichert man den Datensatz in der Hashtabelle ab ?

Problem: Wie verfährt man bei Kollisionen ?

Alternative Lösung:
(Offenes Hashing)

Suche anderen leeren Platz in der Hashtabelle:
Rücke von T[i] jeweils nach einer festen Regel weiter,
bis ein freier Platz gefunden wurde

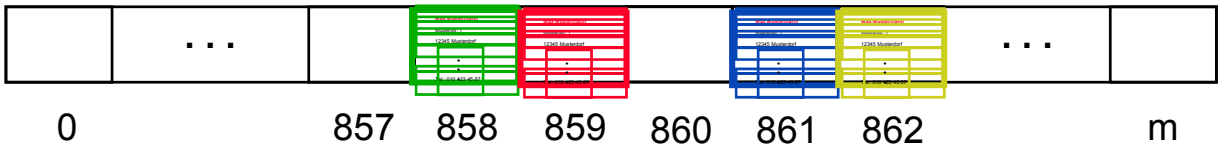
Sondierungsregel

Weitere Möglichkeiten für Sondierungsregeln:

1. **Weiterrücken in quadratischen Abständen**
2. **Weiterrücken gemäß einer zweiten Hashfunktion** ***(double hashing)***
3. **Viele weitere Regeln in der Literatur und Praxis**

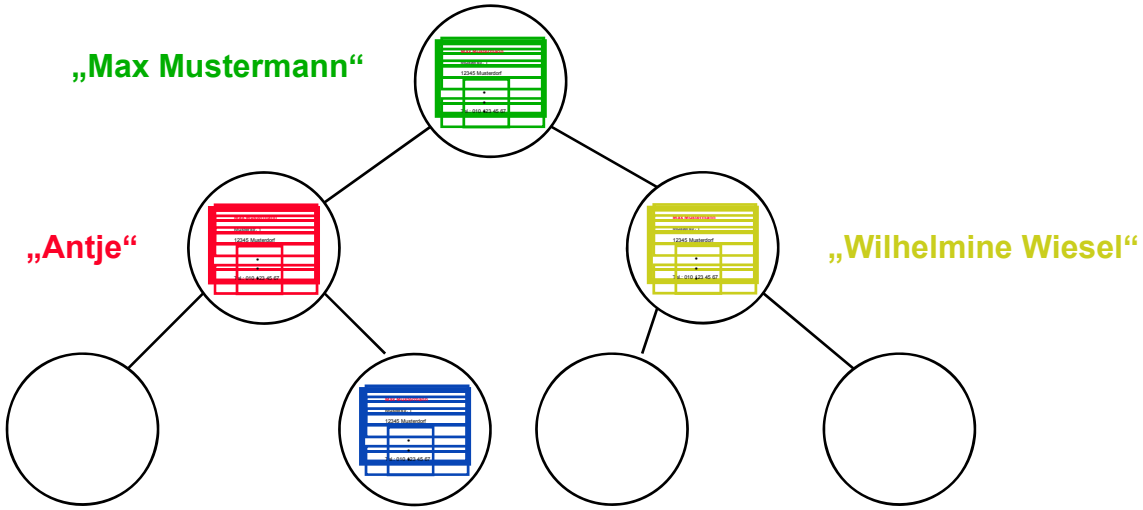
Vergleich mit anderen Verfahren

Hashtabellen



Was ist besser ?

Suchbäume



Vergleich mit anderen Verfahren

m = 1000
n = 500
n = 1000
n = 2000
n = 1 000 000

	Suchbäume (mit n Einträgen)		Hashtabellen (mit n Einträgen und m Hashplätze)		
Speicherplatz	**O(n)**	≈ 500 ≈ 1000 ≈ 2000	**O(m+n)**	≈ 1500 ≈ 2000 ≈ 2500	**Verbesserung durch Offenes Hashing**
durchschnittliche Laufzeit einer Operation (Suchen / Einfügen / Entfernen)	**O(log n)**	≈ 9 ≈ 10 ≈ 11 ≈ 20	**O(n/m)**	≈ 1,2 ≈ 1,3 ≈ 2,1 ≈ 1000	
Anwendbarkeit	**für beliebig viele Daten**		**nur für konstant viele Daten sinnvoll (n ≈ m)**		**Verbesserung durch Dynamisches Hashing**
Einsatz - empfehlung	**bei überwiegendem Einfügen und Entfernen**		**bei überwiegendem Suchen**		

Algorithmik 3

Implementierung von Dictionaries

Ein Dictionary ist eine Datenstruktur für mit einem Schlüssel vergleichbare Elemente,
welche die Funktionen member (key), insert (key, newdata) und delete (key) zur Verfügung stellt

Zusammenfassung Hashing

Datentyp: Indiziertes Array mit m Speicherplätzen

Arbeitsweise:

- Es gibt hash-Funktion h: Keys → {0,…,m-1}
- Jedes Element wird auf Position h(k) abgespeichert,
  wenn diese Position noch frei ist (wobei k der Schlüssel des Elements ist)
- Wenn die Position h(k) besetzt ist,
  muss eine Kollisionsvermeidung gemacht werden (verschiedene Strategien)

Alle 3 Dictionary-Funktionen — Laufzeit $\Theta(n)$ w.c. und $\Theta(n/m)$ a.c.
$\Rightarrow$ für $n \in O(m)$: Laufzeit $\Theta(1)$ a.c.

Referenzen zum Nacharbeiten und Vertiefen:

Cormen, Kap. 11

Algorithmik 3

Implementierung von Dictionaries

Ein Dictionary ist eine Datenstruktur für mit einem Schlüssel vergleichbare Elemente,
welche die Funktionen member (key), insert (key, newdata) und delete (key) zur Verfügung stellt

Zusammenfassung Hashing: Strategien der Kollisionsvermeidung

Datentyp: Indiziertes Array mit m Speicherplätzen

Hashlisten

- An Position h(k) wird anstelle des Datums ein Pointer auf eine lineare Liste vorgehalten.
  Alle Daten, die auf h(k) abgebildet werden, werden sequentiell in die Liste eingefügt.

Offenes Hashing

- Wenn die Position h(k) besetzt ist,
  wird mit spezieller Sondierungsregel eine andere Position bestimmt
- Für die Sondierungsregel gibt es viele Strategien
- Wenn alle Positionen besetzt sind, muss das Array vergrößert werden
  und die Hashfunktion angepasst werden (**Rehashing**)

Referenzen zum Nacharbeiten und Vertiefen:

Cormen, Kap. 11

Algorithmik 3

Implementierung von Dictionaries

Ein Dictionary ist eine Datenstruktur für mit einem Schlüssel vergleichbare Elemente,
welche die Funktionen member (key), insert (key, newdata) und delete (key) zur Verfügung stellt

Zusammenfassung Suchbäume

Datentyp: Paar (Daten, Liste der Kinder-Suchbäume) ← Knoten

Arbeitsweise:

- Jede Operation sieht sich die Daten des Knotens an, auf dem sie aufgerufen wird.
- Wenn die Operation nicht direkt auf diesen Daten ausgeführt werden kann, wird genau mit einem Kind weitergemacht.
- Der Suchbaum hat Invarianten, die immer eingehalten werden müssen (z.B. die Eigenschaft, dass jeder Knoten genau 2 Kinder hat)
- Die Einhaltung der Invarianten erfordert möglicherweise zusätzliche Operationen beim insert und delete.

Alle 3 Dictionary-Funktionen — Laufzeit $\Theta(h)$ ← h ist die Höhe des Suchbaums

h ist zwischen $\Omega(\log n)$ und $O(n)$ w.c., $\Theta(\log n)$ a.c.

↑ verschiedene Betrachtungsweisen

Referenzen zum Nacharbeiten und Vertiefen:

Cormen, Kap. 12, Skript Alt, S. 40-41