Algorithmics

Sebastian Iwanowski
FH Wedel

3. Solutions for the dictionary problem
3.1 Hashing and other methods for optimizing the avarage case behaviour

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

Using a sorted array for a dictionary:

member (key) — run time Θ(log n) w.c. and Θ(log log n) a.c. achievable

not by the same algorithm

insert (key, newdata) — run time Θ(n) w.c. and a.c.

delete (key) — run time Θ(n) w.c. and a.c.

Better method for insert / delete with indexed arrays: Hashing (cf. following slides)

References:

Skript Alt, S. 30 – 35 (for member) in German
more information: cf. previous chapter

Which problem does hashing solve?

data record:

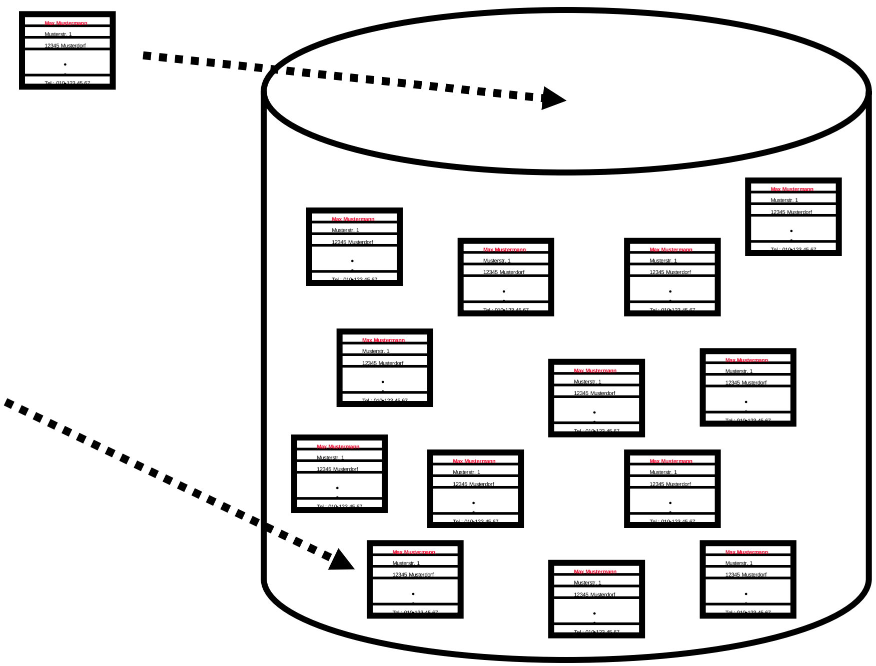
key for searching →

identifies
data record
uniquely

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

`key` `value`

data base:



data administration operations:

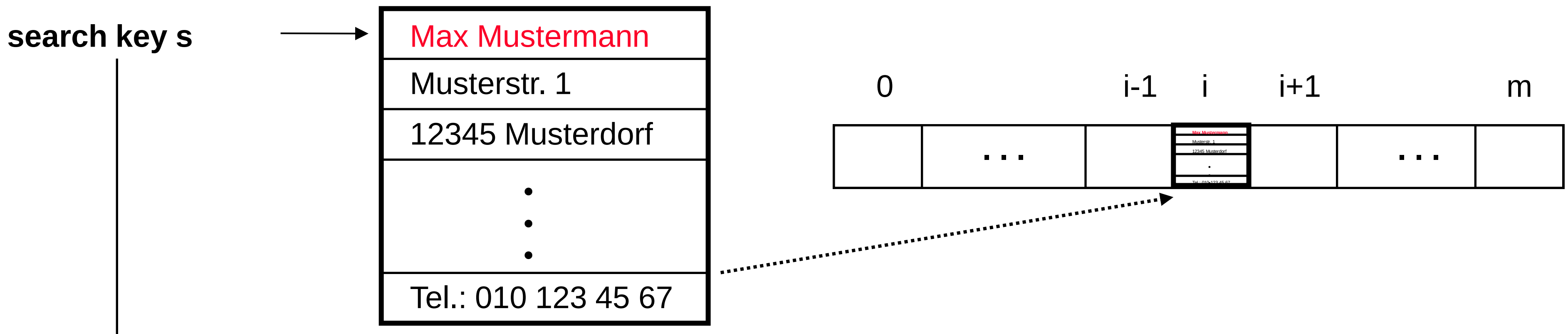
`map operations`

- **search** `get (key)`
- **insert** `put (key, value)`
- **delete** `remove (key)`

→ ***Hashing*** **is a method implementing these operations efficiently.**

Outline of method

data record: **hash table T:**



function hash: key → integer

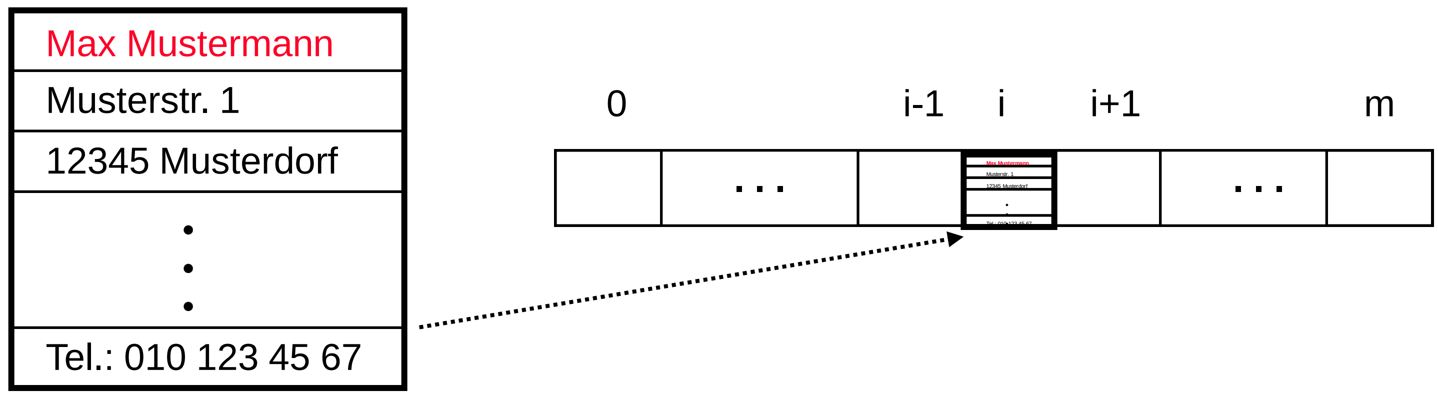
„Max Mustermann“ → i

hash number hash(s) hash(„Max Mustermann“) = i

- **search** — Determine i=hash(s) → Data record searched is in T[i].
- **insert** — Determine i=hash(s) → Store new data record in T[i].
- **delete** — Determine i=hash(s) → Delete data record from T[i]. `(T[i] = null)`

Discussing details

data record: **hash table T:**



1) **How to define a good hash function?**

2) **Where to store the data record in the hash table?**

1) How to define a good hash function ?

Case 1: **Hash table contains at least as many records as different keys are possible.**

Goal: **Each key is mapped to a *different* hash number.**

perfect hashing

Solution: **Sort the keys by order (e.g. lexikographically) !**

Map each key to its order number!

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

(for strings as keys)

hash („Max Mustermann“) = $13 \cdot 27^{13} + 1 \cdot 27^{12} + 24 \cdot 27^{11} + 0 \cdot 27^{10} + 13 \cdot 27^{9} + 21 \cdot 27^{8} + 19 \cdot 27^{7}$

$+ 20 \cdot 27^{6} + 5 \cdot 27^{5} + 18 \cdot 27^{4} + 13 \cdot 27^{3} + 1 \cdot 27^{2} + 14 \cdot 27^{1} + 14 \cdot 27^{0}$

≈ 52966834350000000000 (20-digit number)

In general a lot of *different* keys are possible!

Conclusion: **Case 1 is not realistic !**

1) How to define a good hash function ?

<u>Case 2:</u> **Hash table contains fewer records than different keys that are possible.**

m k

Case 2: $m < k$

Conclusion: **Different keys have to be mapped to the same hash number**

collision

Goal:

Each hash number 0 thru m-1 is the function value of aproximately equally many keys (i.e. approximately k/m).

Solution: **Sort the keys by order (e.g. lexikographically) !**

Map each key to its order number modulo m !

1) How to define a good hash function ?

Example:

(for strings as keys)

m = 1000 „Antje“ → (1 14 20 10 5)

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

$= 821858 \bmod 1000$

$= 858$

Algorithm for a good hash function (according to *Horner*‘s method) :

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

$= (((((((1 \bmod 1000 * 27 + 14) \bmod 1000) * 27 + 20) \bmod 1000) * 27 + 10) \bmod 1000) * 27 + 5) \bmod 1000$

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;
  }
```

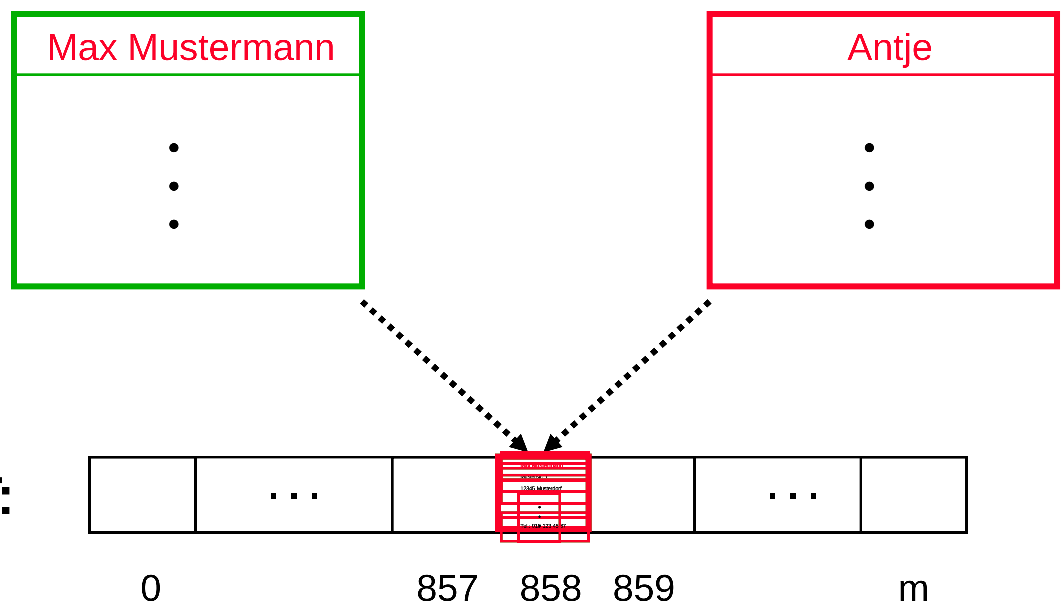
2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Example: hash („Max Mustermann“) = 858

hash („Antje“) = 858

Hash table T:

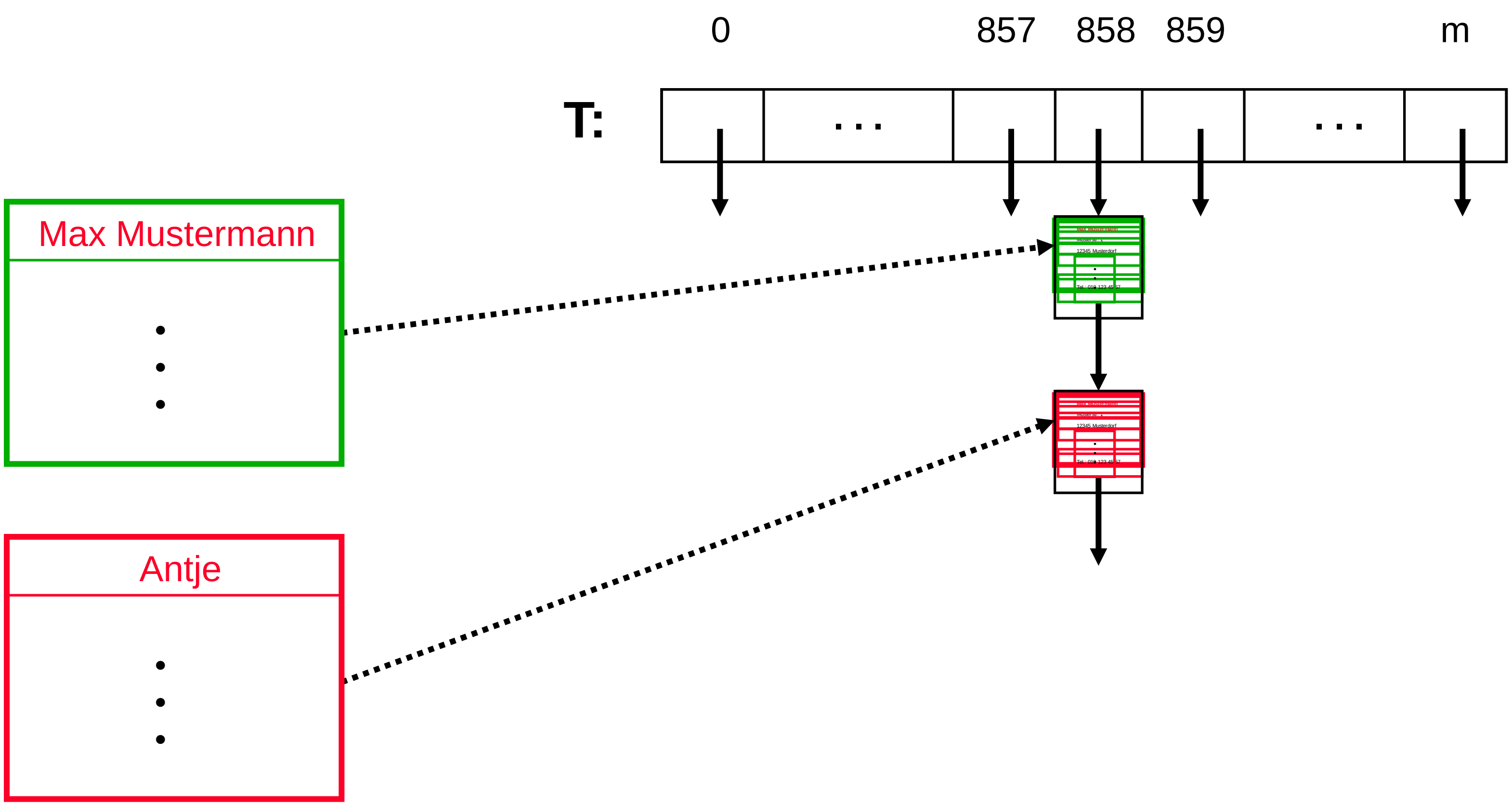


Does anybody have a better idea?

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

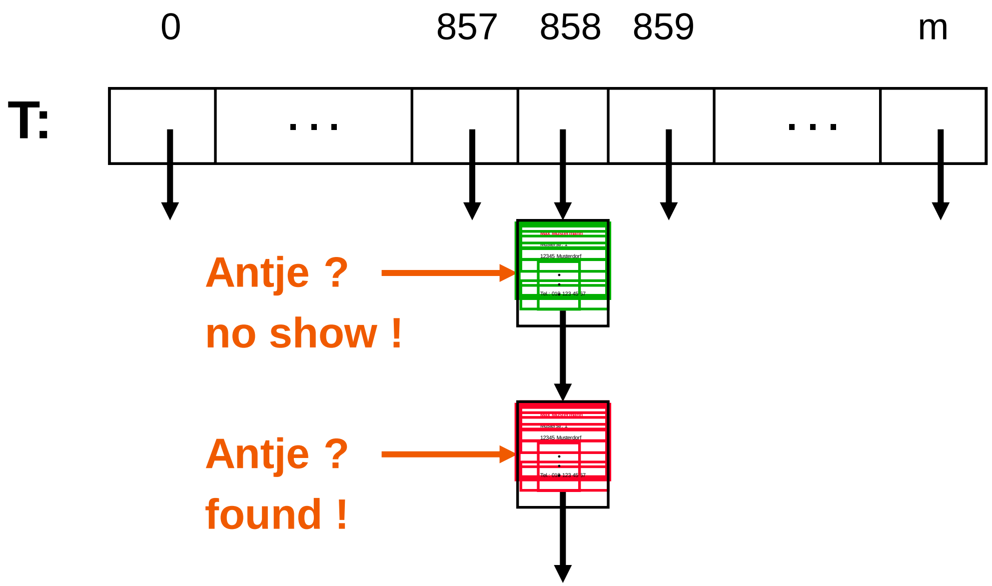
Solution: **T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.**



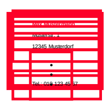
2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: **T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.**



Search: **Antje ?**



1) Determine hash („Antje“) = 858

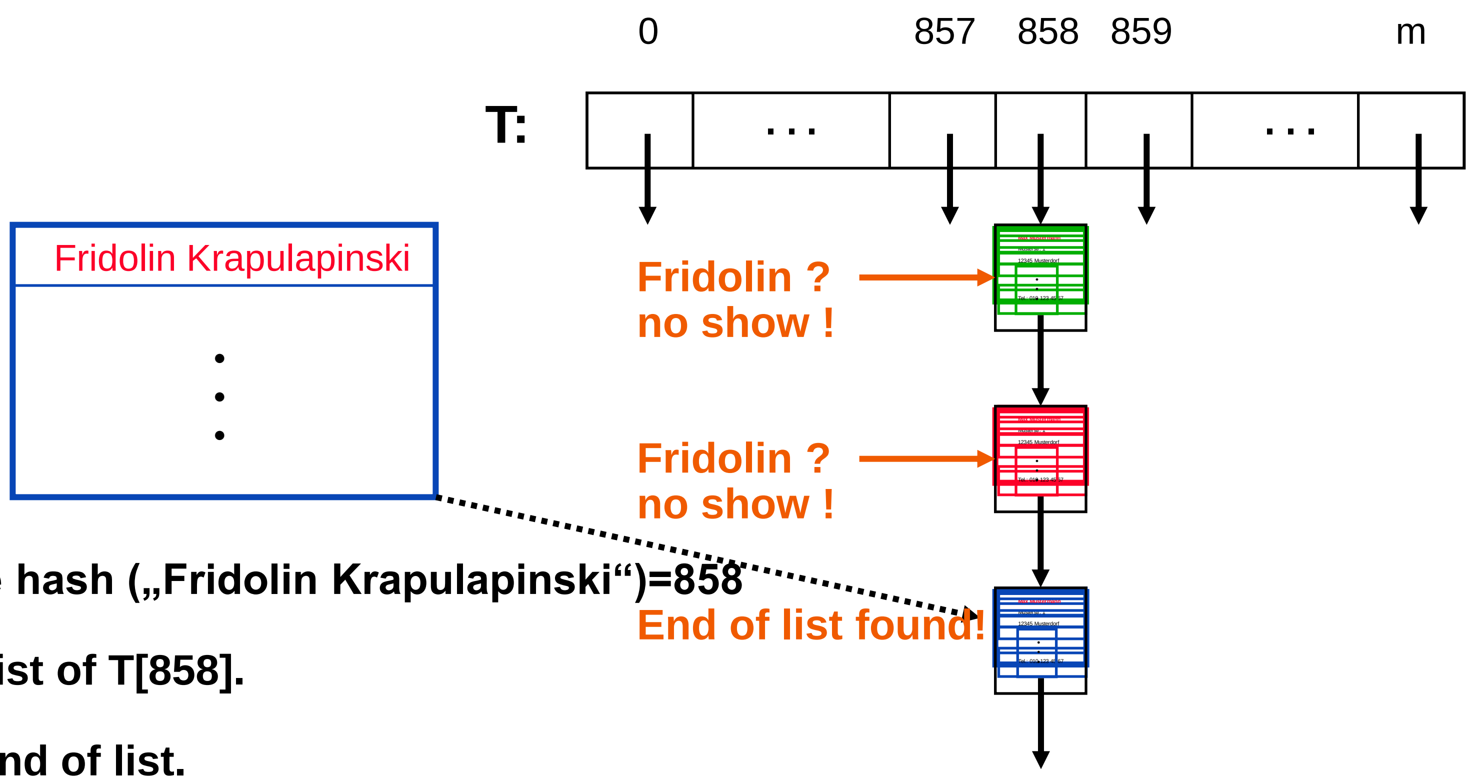
2) Traverse list of T[858]

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: **T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.**

Insert:



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

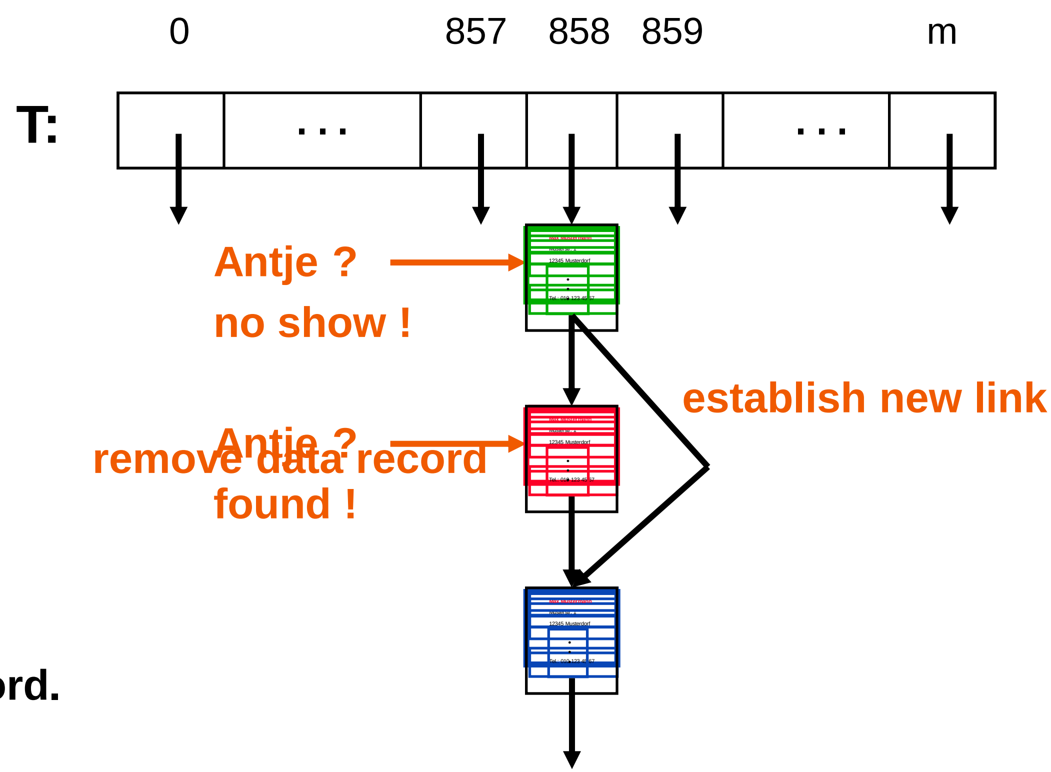
2) Traverse list of T[858].

3) Insert at end of list.

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: **T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.**



Delete: Antje

1) Determine hash („Antje“)=858

2) Traverse list of T[858].

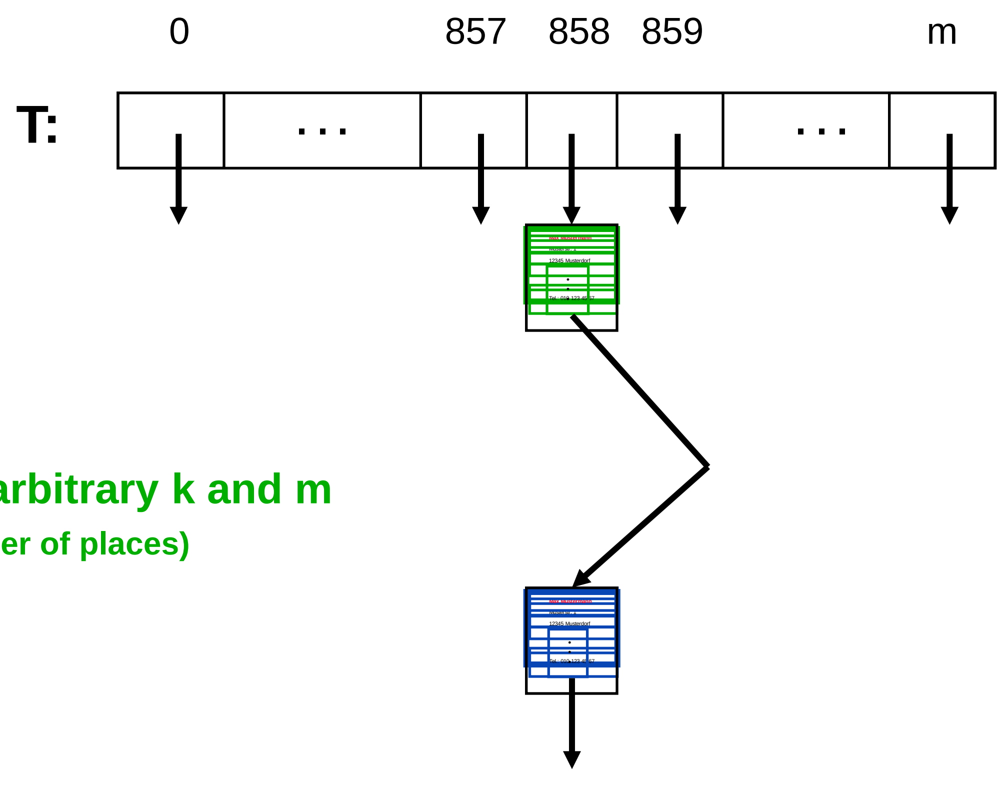
3) Remove the respective data record.

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: **T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.**

(closed hashing)



Evaluating the presented method:

- **easy to implement**
- **implements hashing for arbitrary k and m**
  (k: number of keys, m: number of places)

objection:

- **waste of storage space !**

2) Where to store the data record in the hash table?

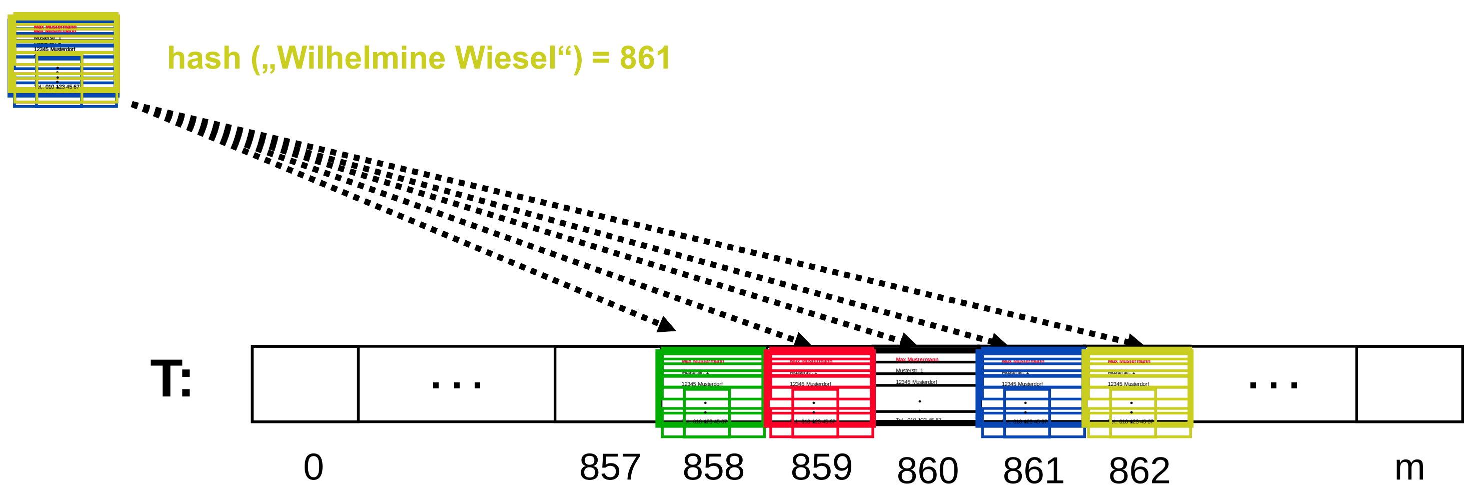
Problem: How to handle collisions?

Alternative solution: ***(open hashing)***

**Search for other free space in hash table:
Proceed from T[i] according to a certain rule
until free space is found**

probing rule

Example for probing rule: **move right by one**



2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Alternative solution:
(open hashing)

Search for other free space in hash table:
Proceed from T[i] according to a certain rule
until free space is found

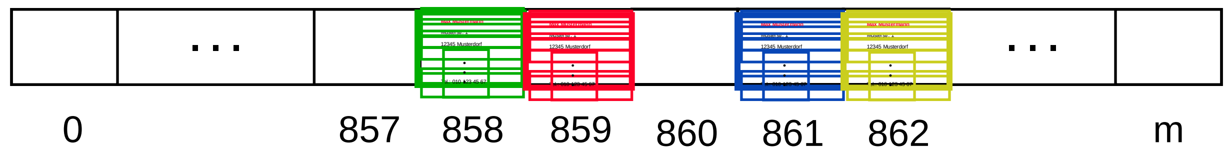
↓ ***probing rule***

Other methods for probing rules:

1. **move by quadratically increasing distances**
2. **move according to a second hash function** ***(double hashing)***
3. **lots more of rules in literature and practice**

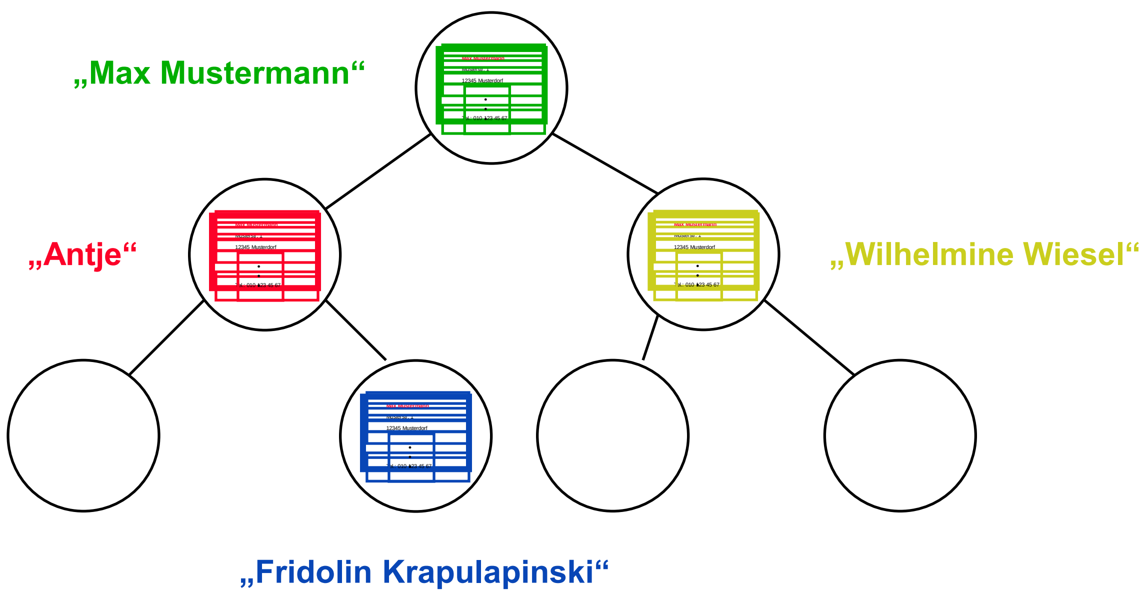
Compare with other techniques

Hash tables



What is better?

Search trees



Compare with other techniques

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

	Search trees (containing n entries)		Hash tables (containing n entries and m hash places)		
Storage	**O(n)**	≈ 500 ≈ 1000 ≈ 2000	**O(m+n)**	≈ 1500 ≈ 2000 ≈ 3000	**improvement by open hashing**
avarage run time of one operation (search / insert / delete)	**O(log n)**	≈ 9 ≈ 10 ≈ 11 ≈ 20	**O(n/m)**	≈ 1,2 ≈ 1,3 ≈ 2,1 ≈ 1000	
Applicability	**for arbitrarily many data**		**only for constant number of data (n ≈ m)**		**improvement by dynamic hashing**
Recommendation of use	**for frequent insert and delete**		**for frequent search**		

Algorithmics 3

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

Summarizing hashing

data type: Indexed array with m positions

Principle of operation:

- There is a hash function h: Keys $\rightarrow \{0,\ldots,m-1\}$
- Each element is stored at h(k),
  as long as this position is still free (where k is the element's key)
- If position h(k) is occupied,
  a collision handling must be performed (different strategies available)

All 3 dictionary functions: run time $\Theta(n)$ w.c. and $\Theta(n/m)$ a.c.
$\Rightarrow$ for $n \in O(m)$: run time $\Theta(1)$ a.c.

References:

Cormen, ch. 11

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

Summarizing hashing: Strategies for collision handling

data type: Indexed array with m positions

Hash lists

- At position h(k), there is a pointer to a linked lists instead of the data record .
- All data to be mapped to h(k) will be inserted sequentially into the linked list.

Open hashing

- If position h(k) is occupied,
  a special probing rule determine a different position.
- There are different strategies for probing rules.
- If all positions are occupied, the array must be enhanced
  and the hash function must be adapted (**rehashing**)

References:

Cormen, ch. 11

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

Summarizing search trees:

data type: pair (data, list of children trees) ← nodes

Principle of operation:

- Each operation inspects the data of the node where it is invoked.
- If the operation may not be executed directly at node, it will be passed to one of the children. The choice to which child will be decided locally in the node.
- The search tree bares invariants that must be maintained (e.g. property that each node has got exactly 2 children)
- The maintenance of the invariants may require additional operations for insert and delete.

Each operation must be performed in constant time per node.

All 3 dictionary functions run time $\Theta(h)$ ← h is height of search tree

h is between $\Omega(\log n)$ and $O(n)$ w.c., $\Theta(\log n)$ a.c.

↑ different ways of considering a.c.

References:

Cormen, ch. 12, Skript Alt, S. 40-41 (in German)