Prepared by Phillip Abramson and Julian García, based on
material by David Albrecht and María García de la Banda

Lecture 29
Collision Resolution II

FIT 1008&2085
Introduction to Computer Science

MONASH University
Information Technology

Objectives for this lecture

- To understand two of the main methods of conflict resolution:
  - Open addressing:
    - Linear Probing ☑
    - Quadratic probing
    - Double Hashing
  - Separate Chaining

- To understand their advantages and disadvantages

- To be able to implement them

Open Addressing: Linear Probing

- **Search** for an item with hash value N:
  - Perform a linear search from array[N] until either the item or an empty space is found

- But careful, you must deal again with:
  - <u>Full table</u> (to avoid going into an infinite loop)
  - Restarting from position 0 if the end of table is reached.

search(key)

- Get the position N using the hash function, **N = hash(key)**
- If **array[N] is empty** return **None**.
- If there is <u>already an item there</u>:
  - If there is already something there, with the **same key** return the associated data.
  - If there is already something there with a **different key**, you need to **find the key** and return **data**

```
__setitem__(self, key, data)
```

insert or update item (key, data)

```
__getitem__(self, key)
```

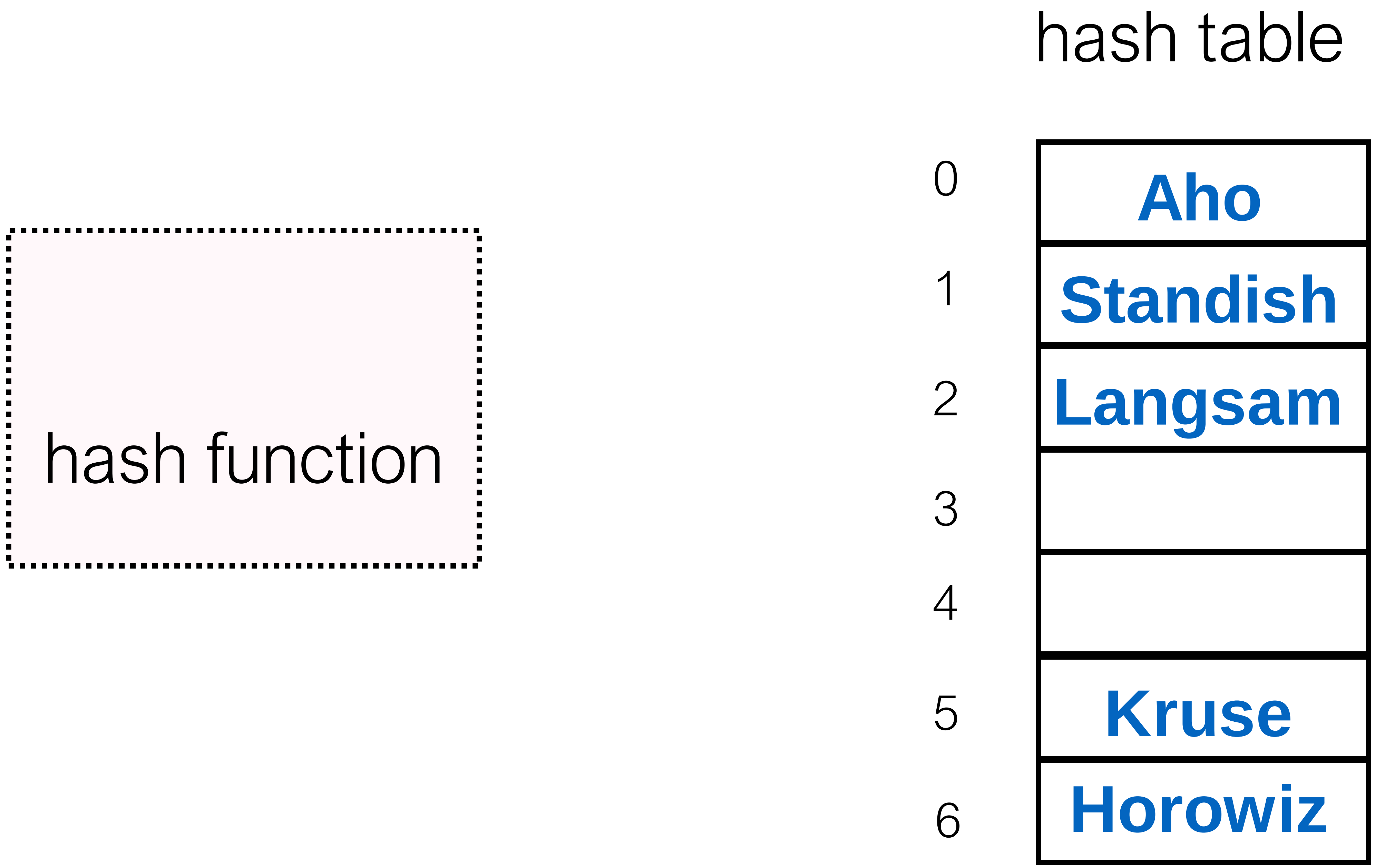
give me the data associated to a key

```
def __getitem__(self, key):
```

```python
def __getitem__(self, key):
    position = self.hash_value(key)
    for _ in range(self.table_size):
        if self.array[position] is None:  # found empty slot
            raise KeyError(key)
        elif self.array[position][0] == key:  # found it
            return self.array[position][1]
        else:
            # there is something there, but different key
            # linear probing, so try next position
            position = (position + 1) % self.table_size
    raise KeyError(key)
```

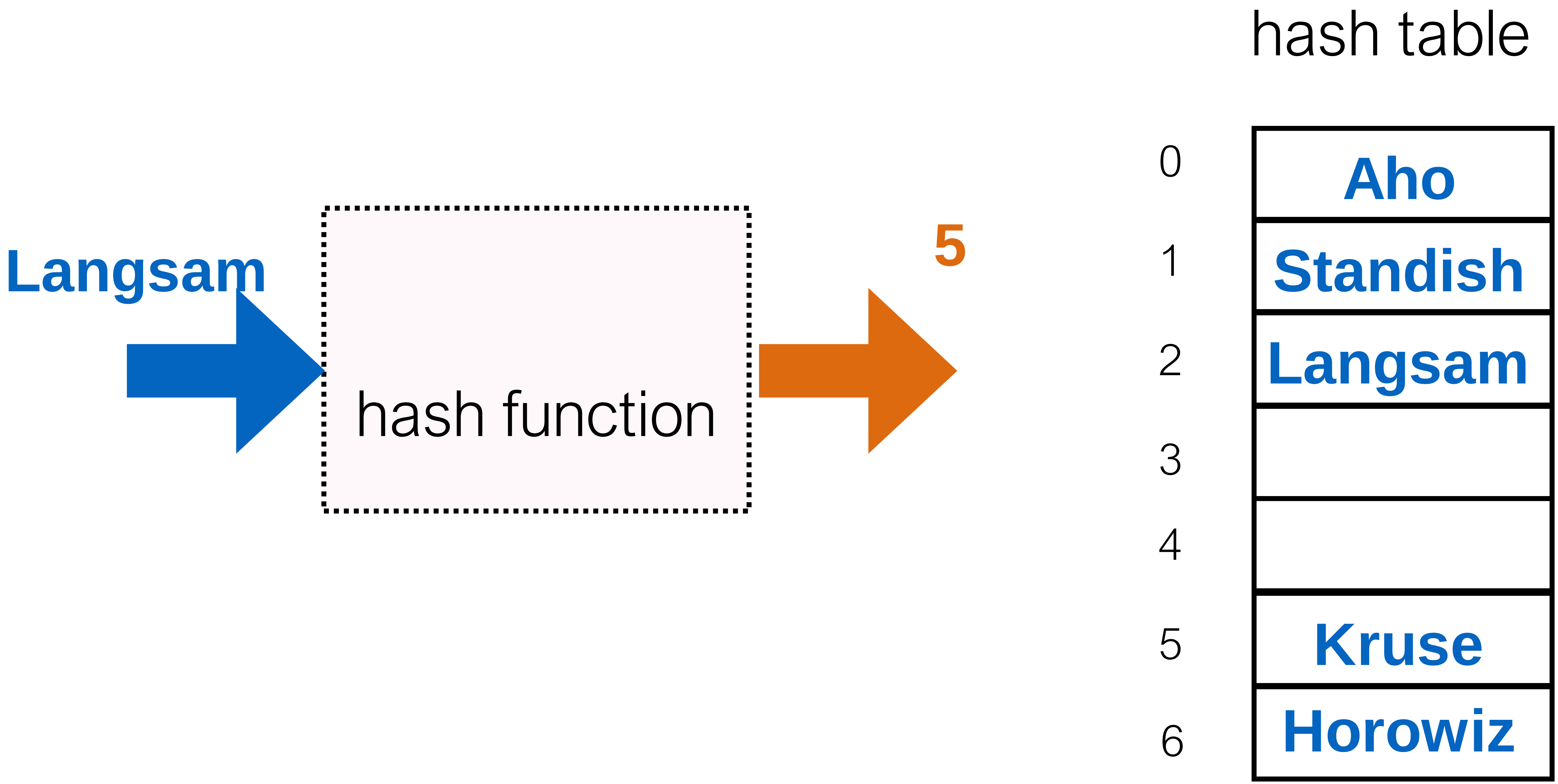
Example: search

key: Langsam



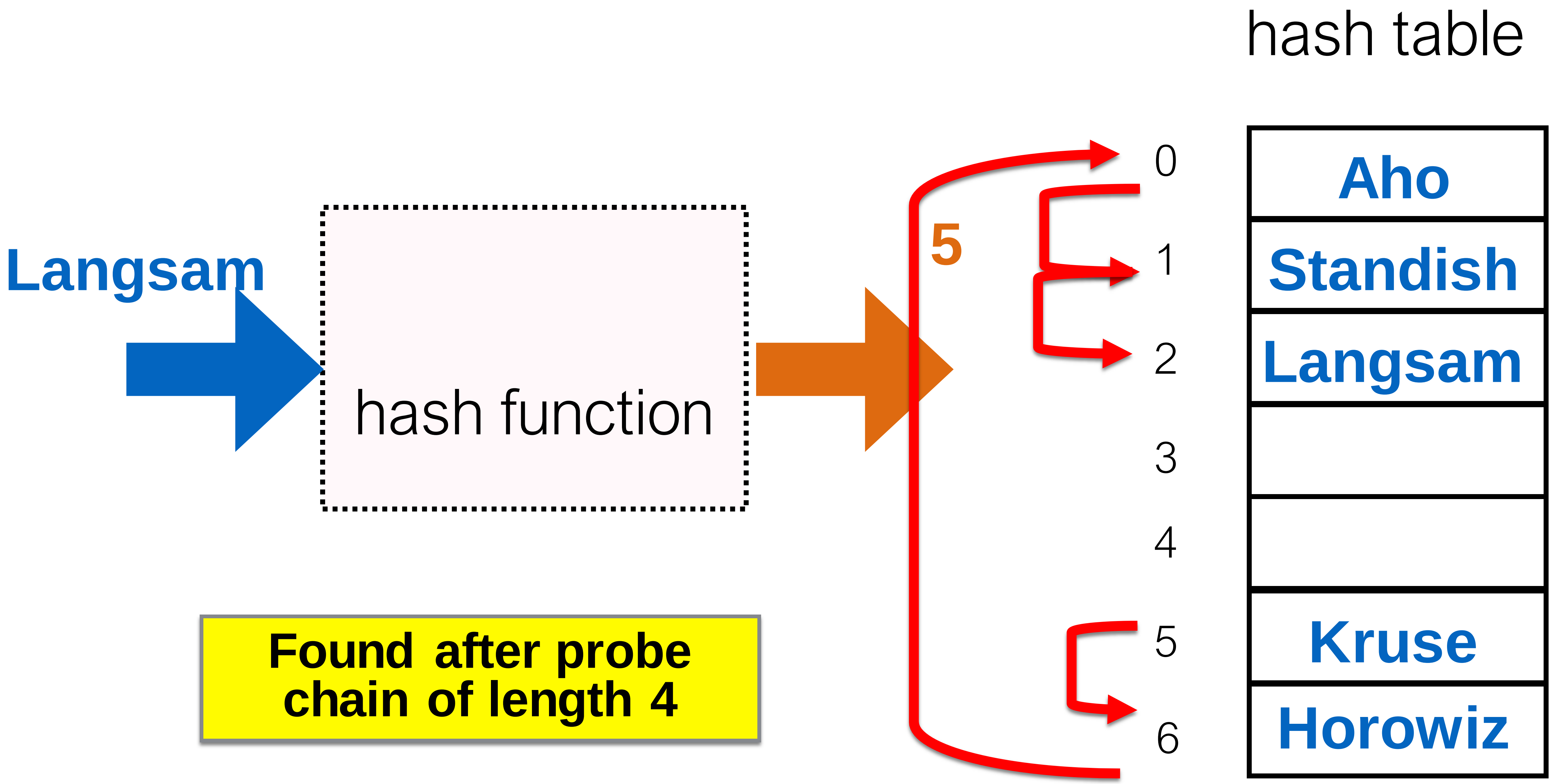
Example: search

key: Langsam



Example: search

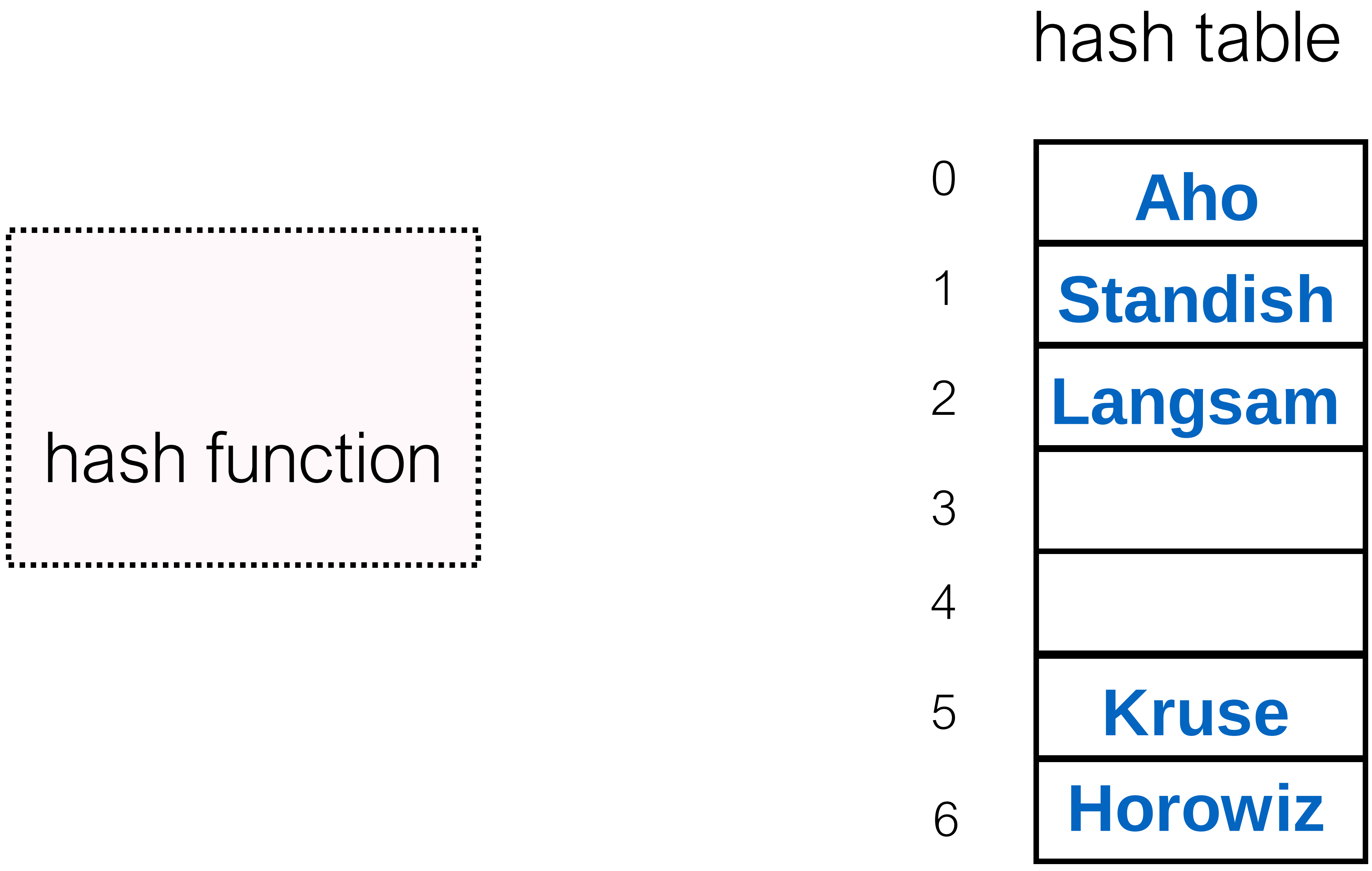
key: Langsam



Found after probe chain of length 4

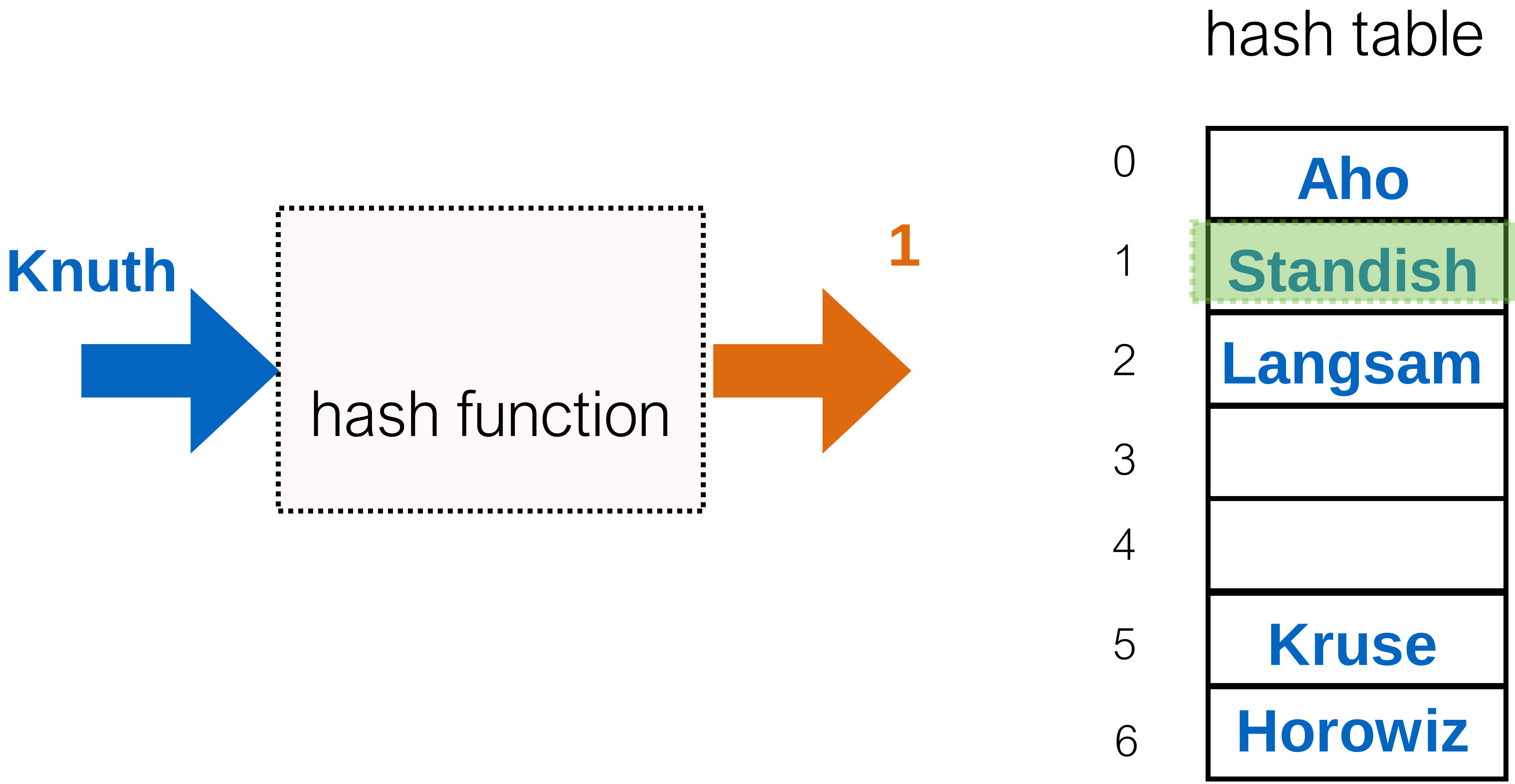
Example: search

key: Knuth



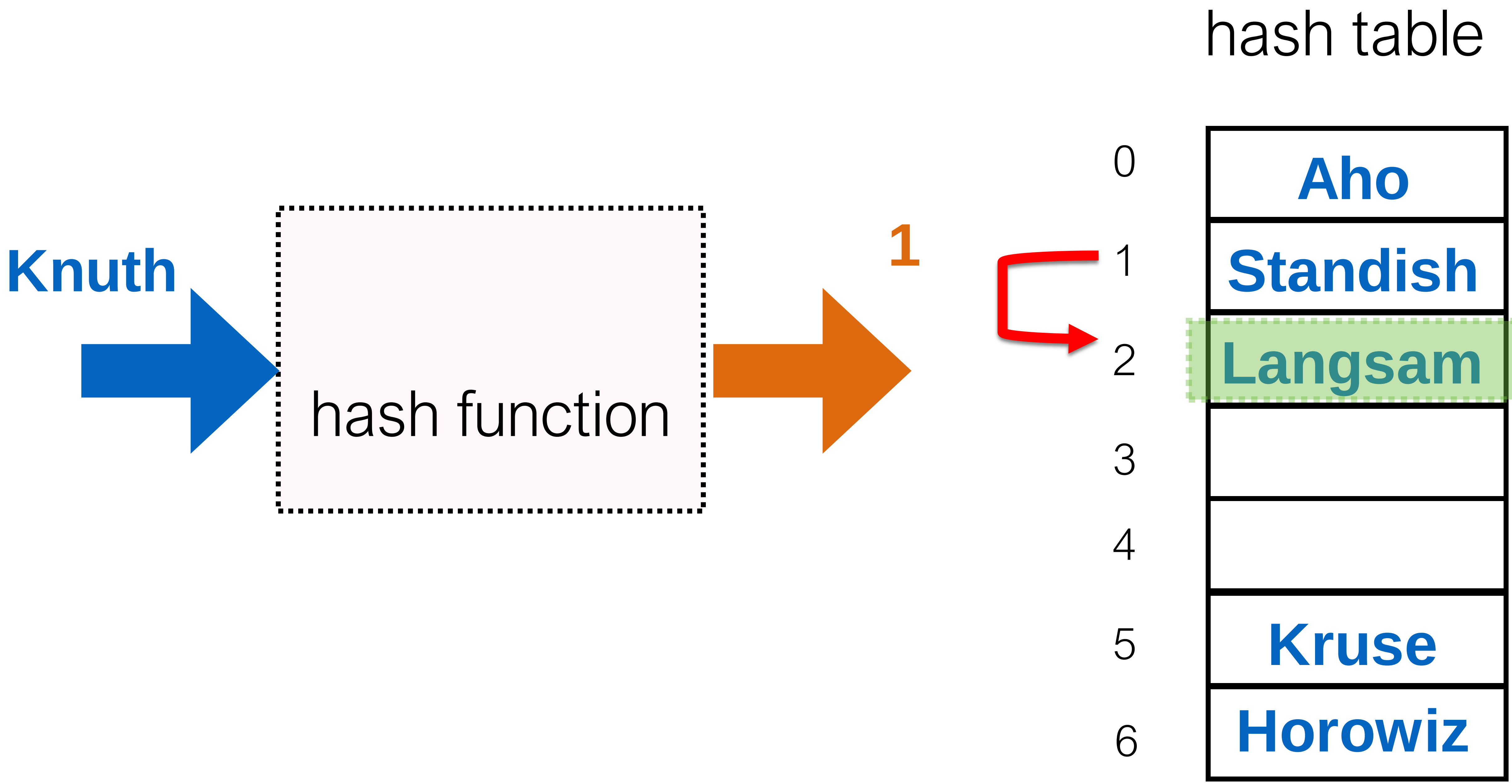
Example: search

key: Knuth



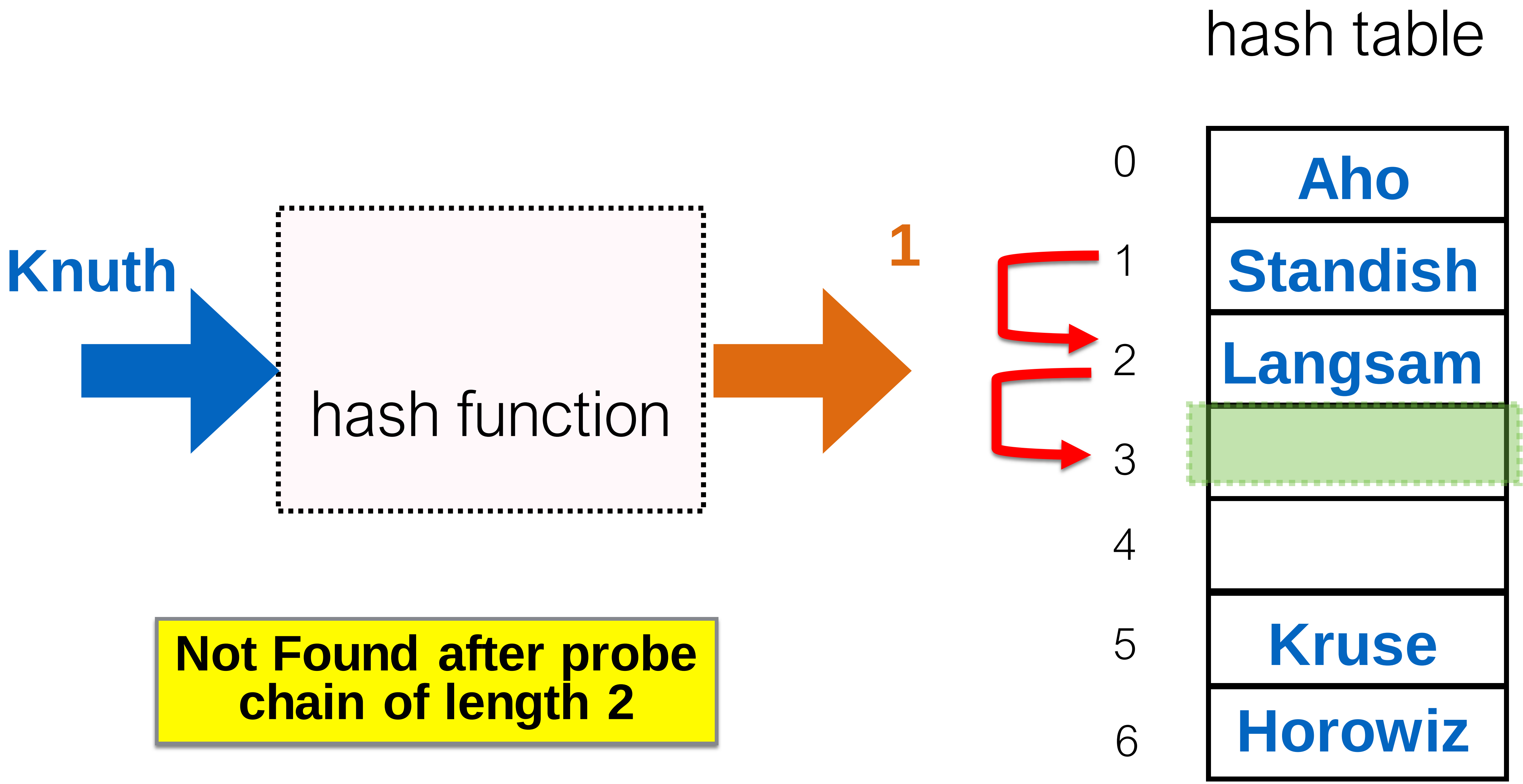
Example: search

key: Knuth



Example: search

key: Knuth



```python
def __setitem__(self, key, data):
    position = self.hash(key)
    for _ in range(self.table_size):
        if self.array[position] is None:  # found empty slot
            self.array[position] = (key, data)
            self.count += 1
            return
        elif self.array[position][0] == key:  # found key
            self.array[position] = (key, data)
            return
        else:  # not found, try next
            position = (position + 1) % self.table_size
    self.rehash()
    self.__setitem__(key, data)
```

```python
def __getitem__(self, key):
    position = self.hash_value(key)
    for _ in range(self.table_size):
        if self.array[position] is None:  # found empty slot
            raise KeyError(key)
        elif self.array[position][0] == key:  # found it
            return self.array[position][1]
        else:
            # there is something there, but different key
            # linear probing, so try next position
            position = (position + 1) % self.table_size
    raise KeyError(key)
```

Idea: use a function for linear probe, make it more compact

Open Addressing: Linear Probing

- What about **delete**?
- One possibility:
  - Use the search function to find the item
  - If found at N delete and reinsert every item from N+1 to the first empty position

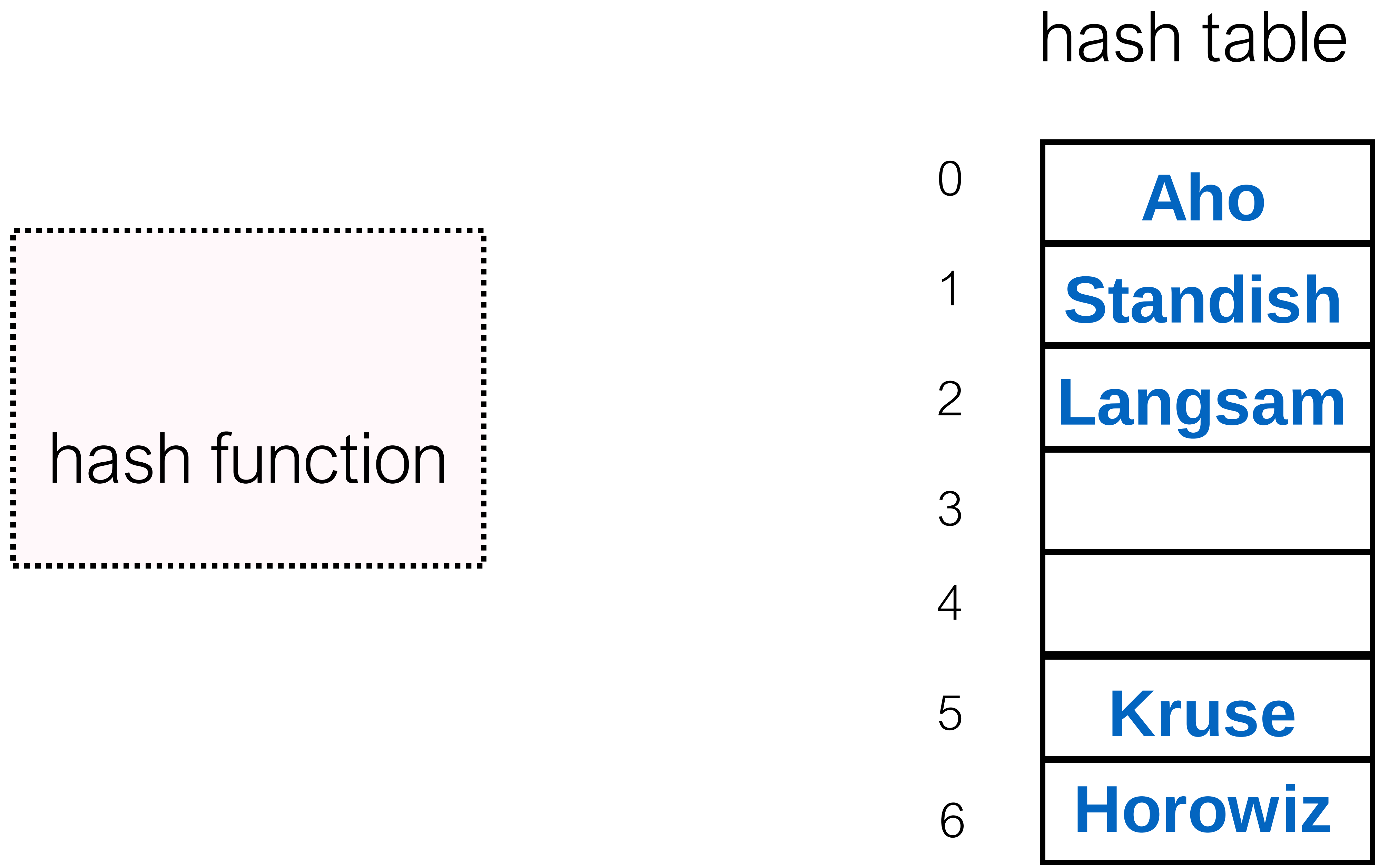
    Time consuming! (though should not be many)

What if I do not reinsert?

search may incorrectly report some items as not found

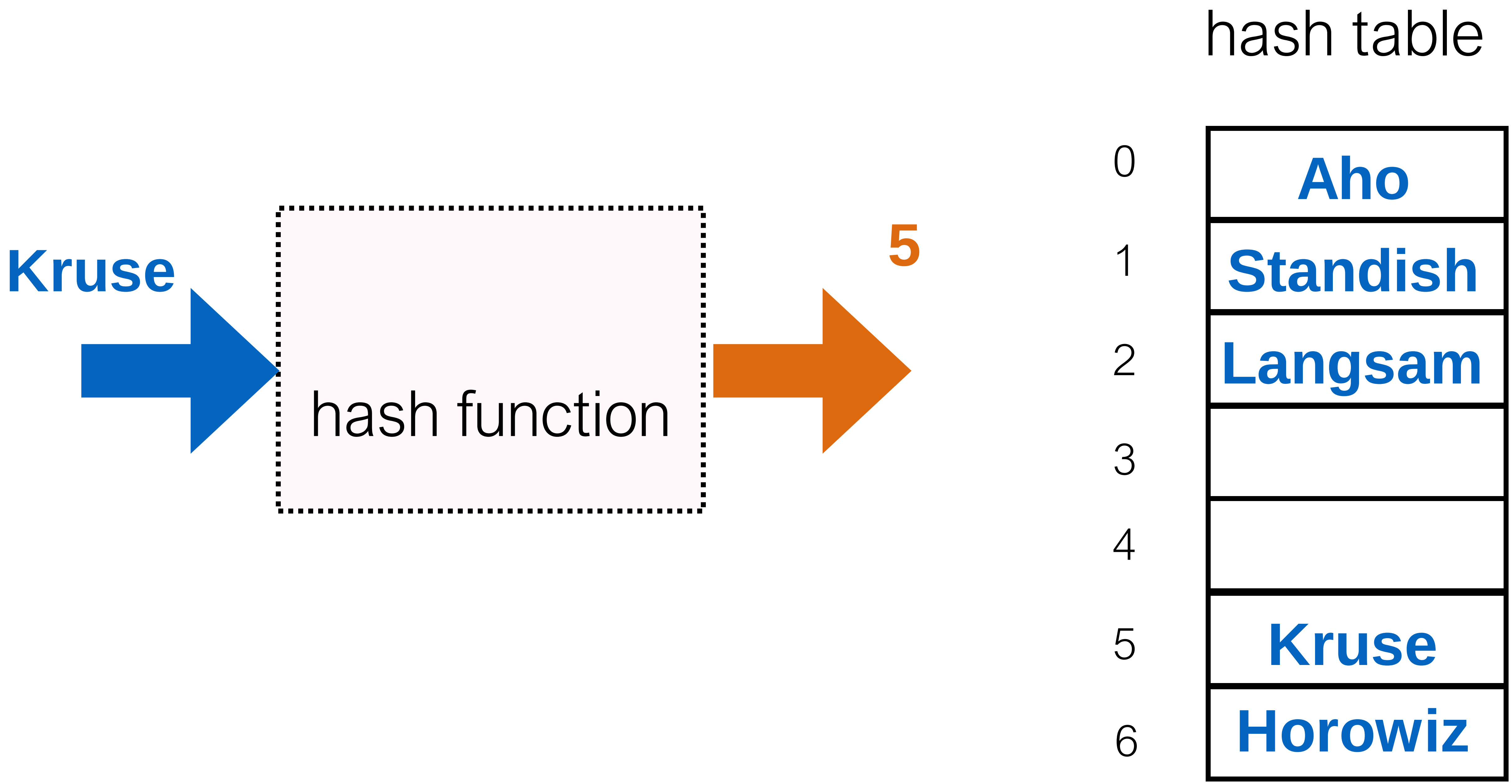
Example: delete

key: Kruse



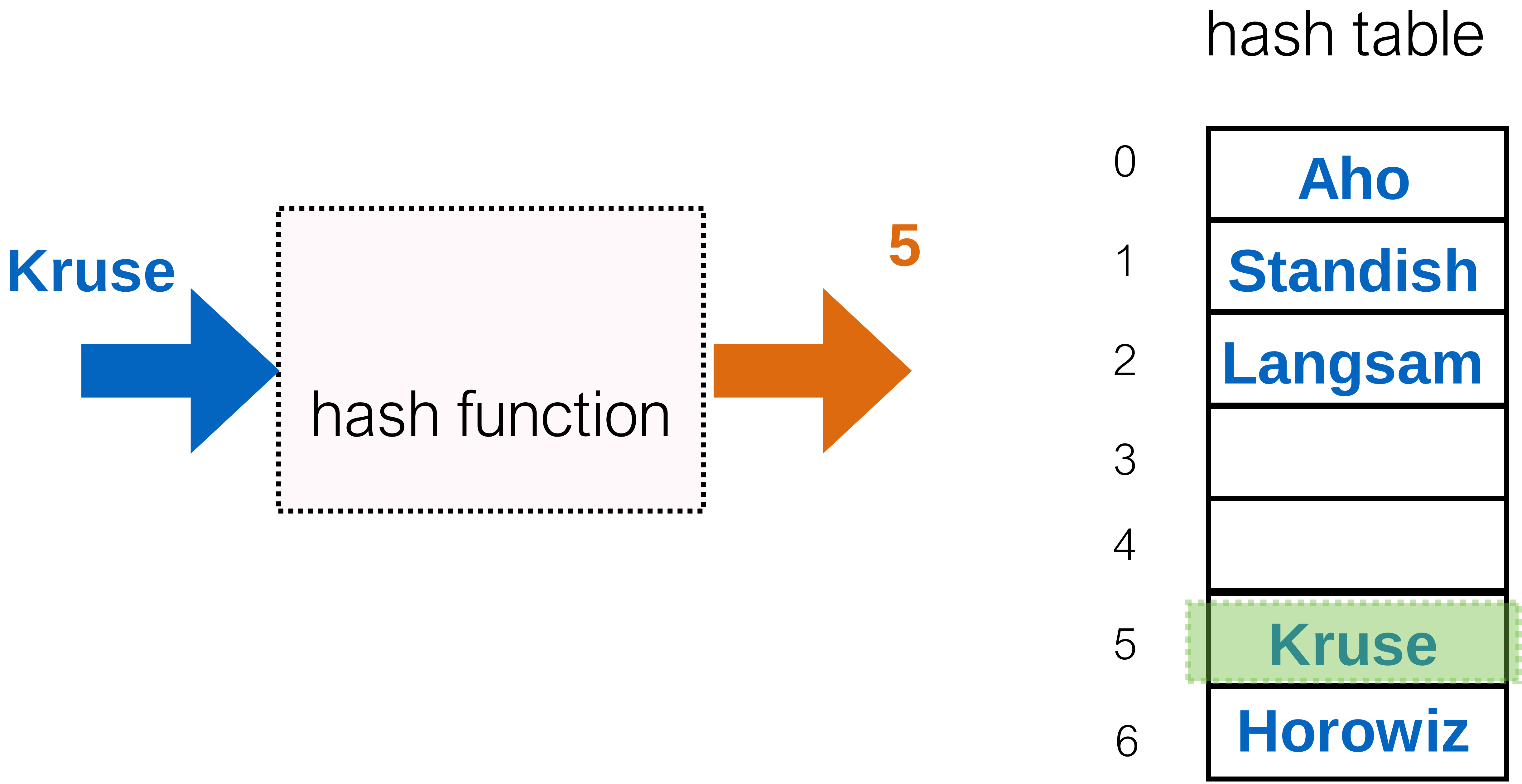
Example: delete

key: Kruse



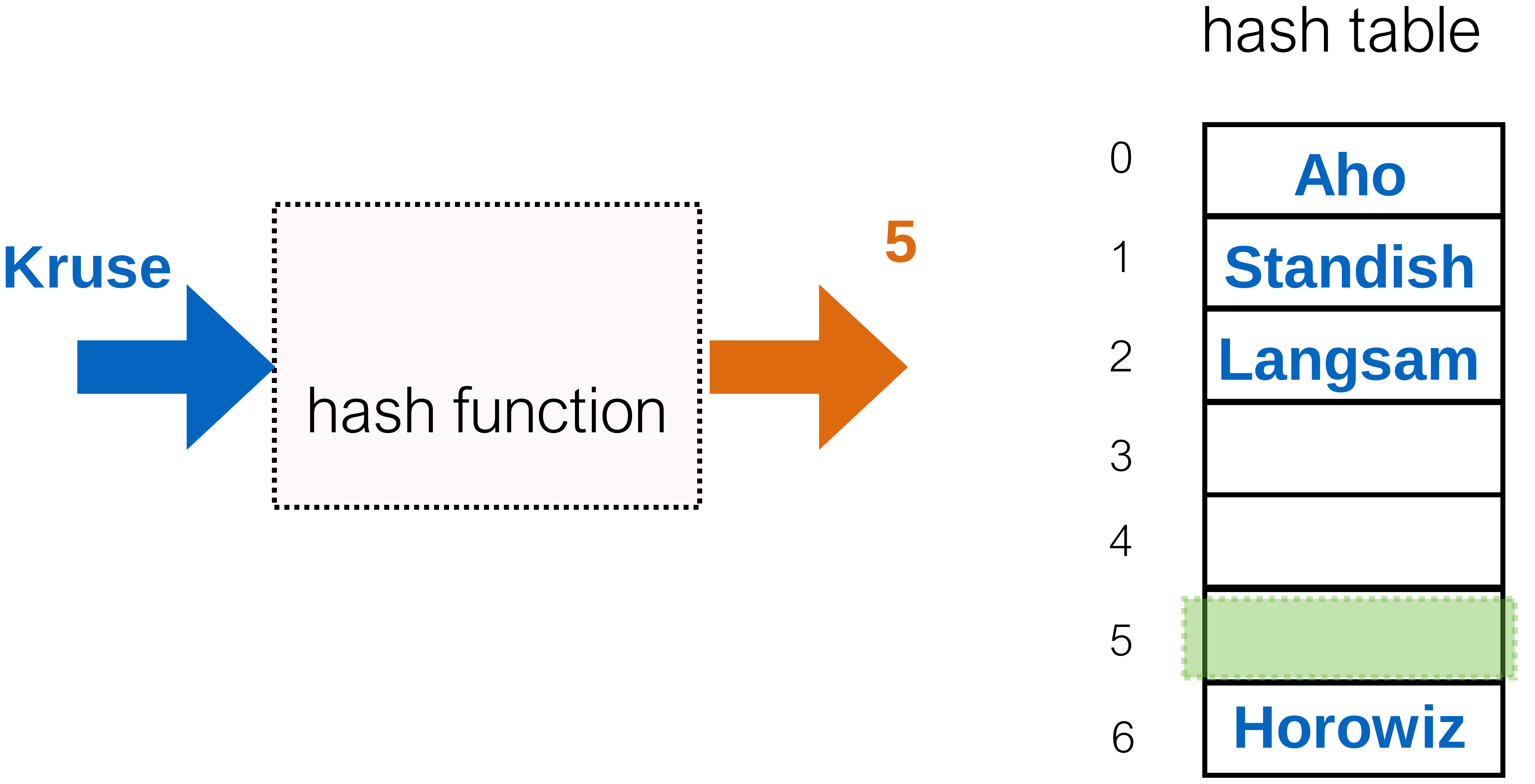
Example: delete

key: Kruse



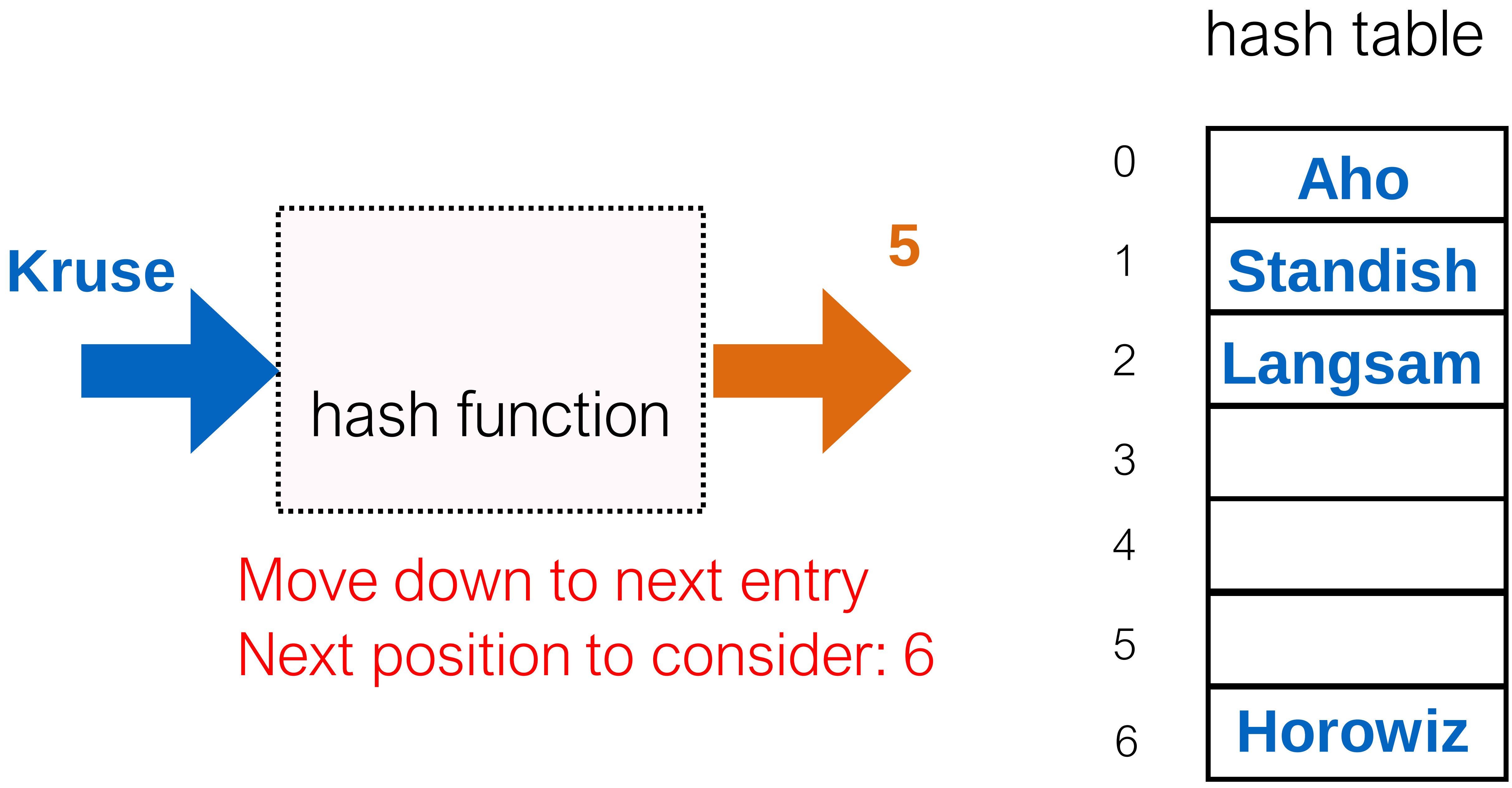
Example: delete

key: Kruse



Example: delete

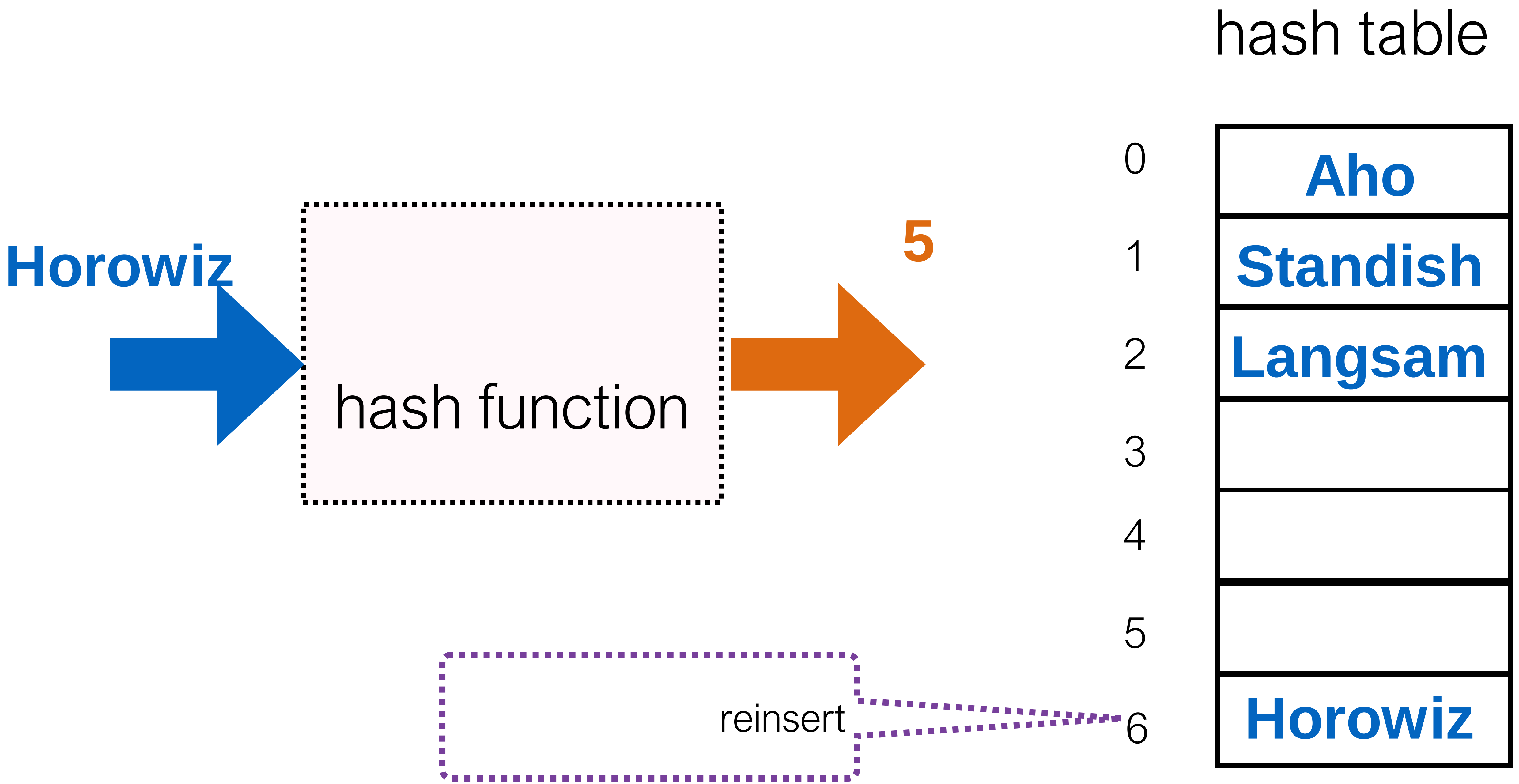
key: Kruse



Move down to next entry
Next position to consider: 6

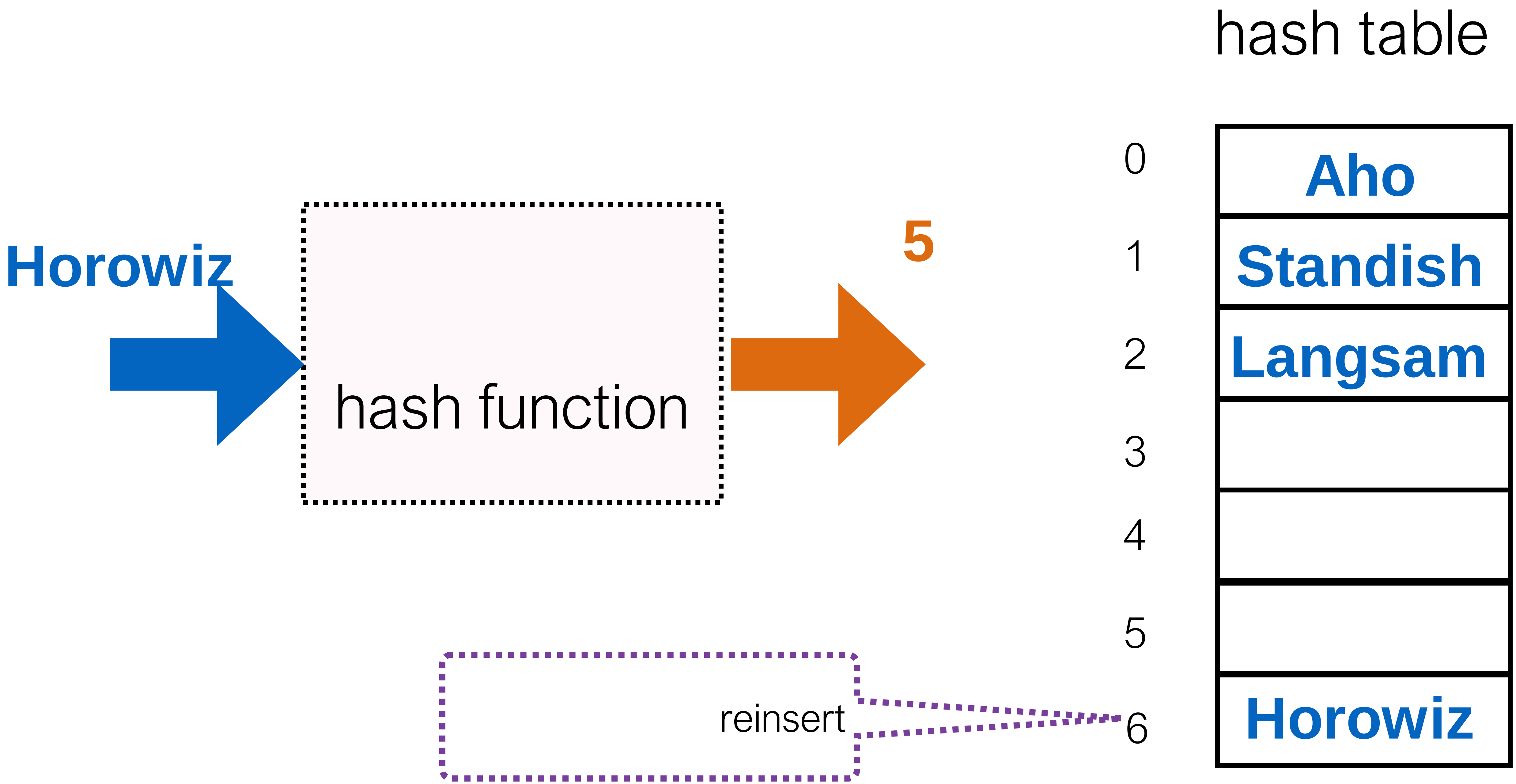
Example: delete

key: Kruse



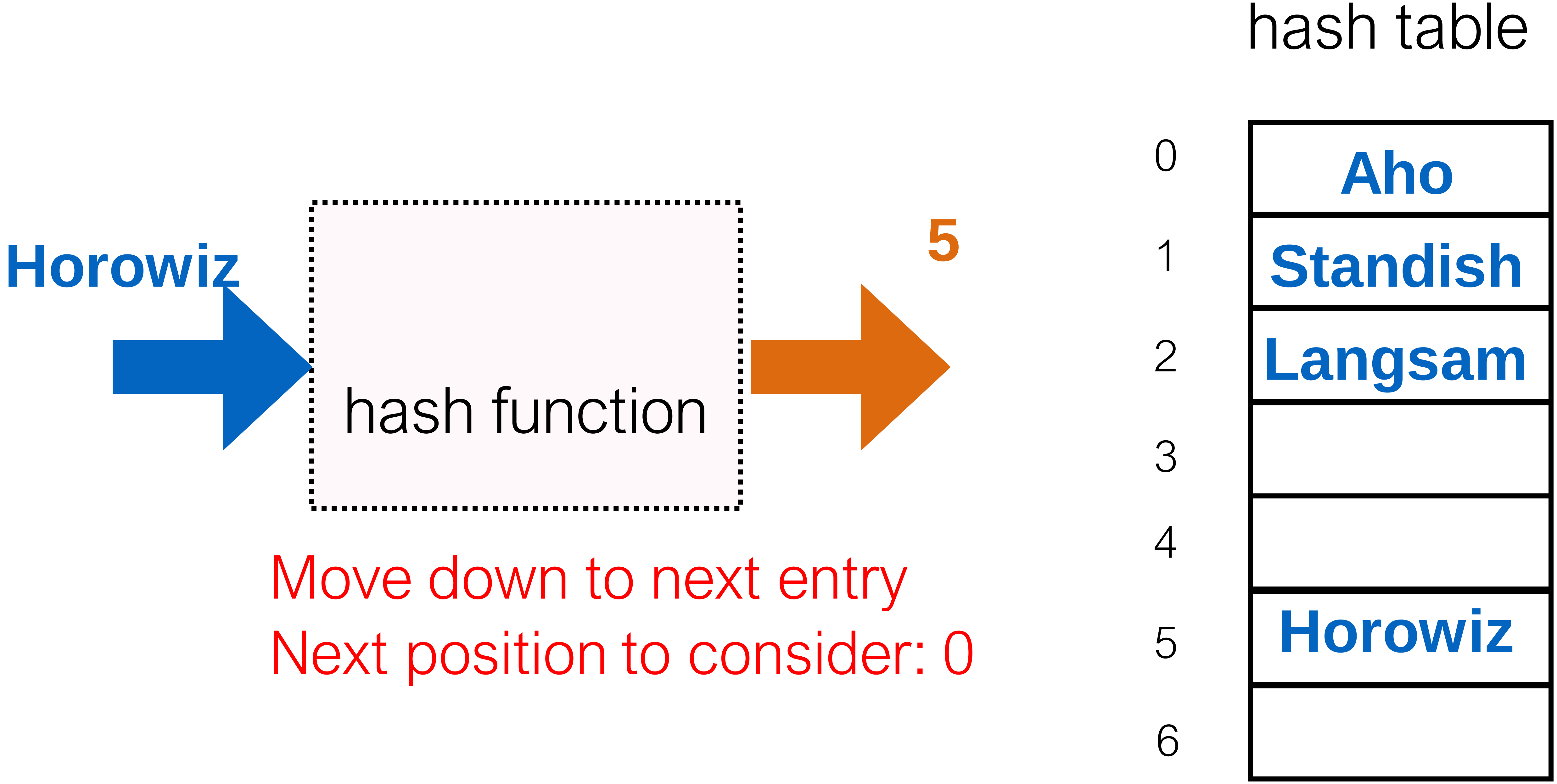
Example: delete

key: Kruse



Example: delete

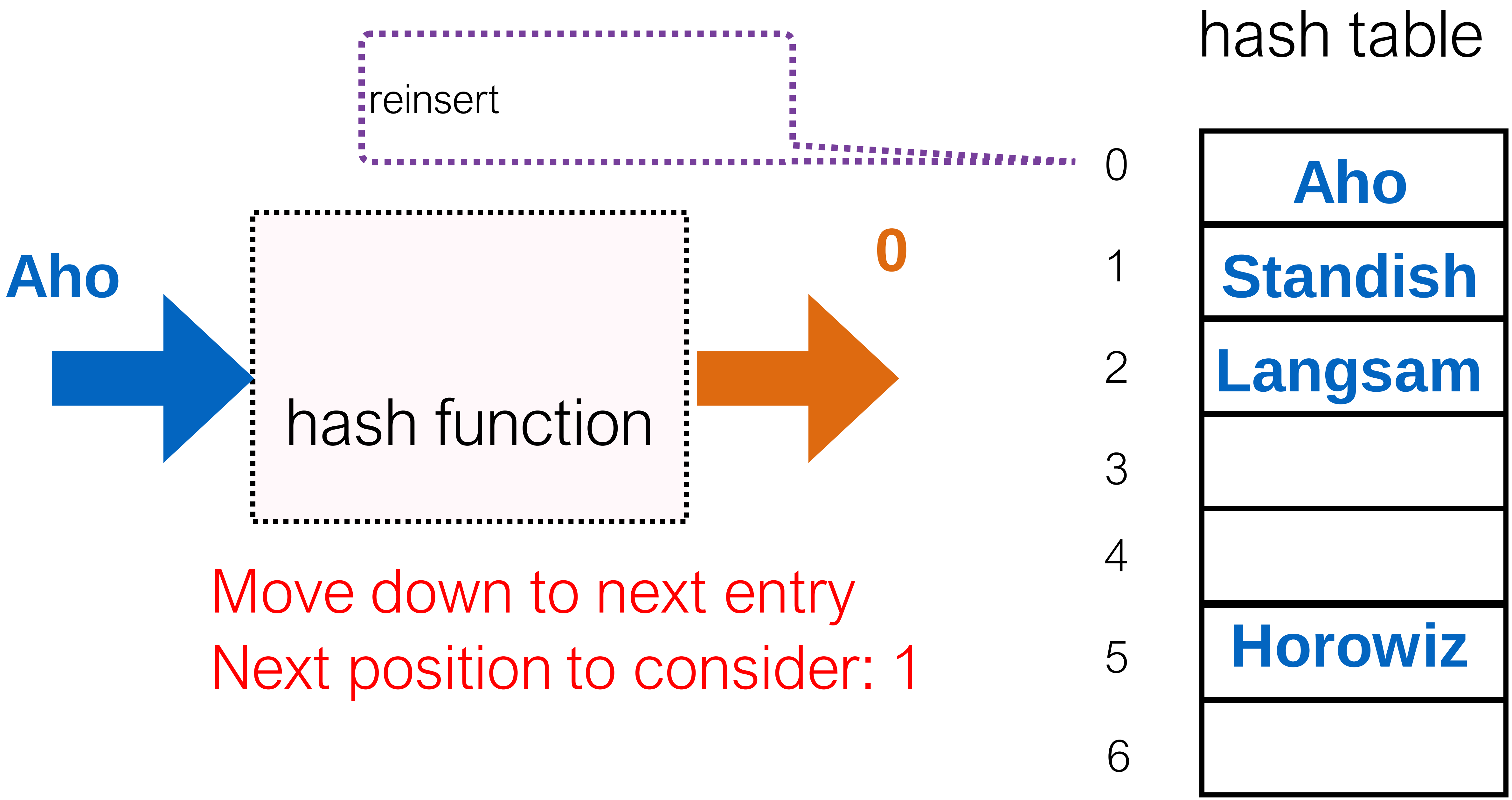
key: Kruse



Move down to next entry
Next position to consider: 0

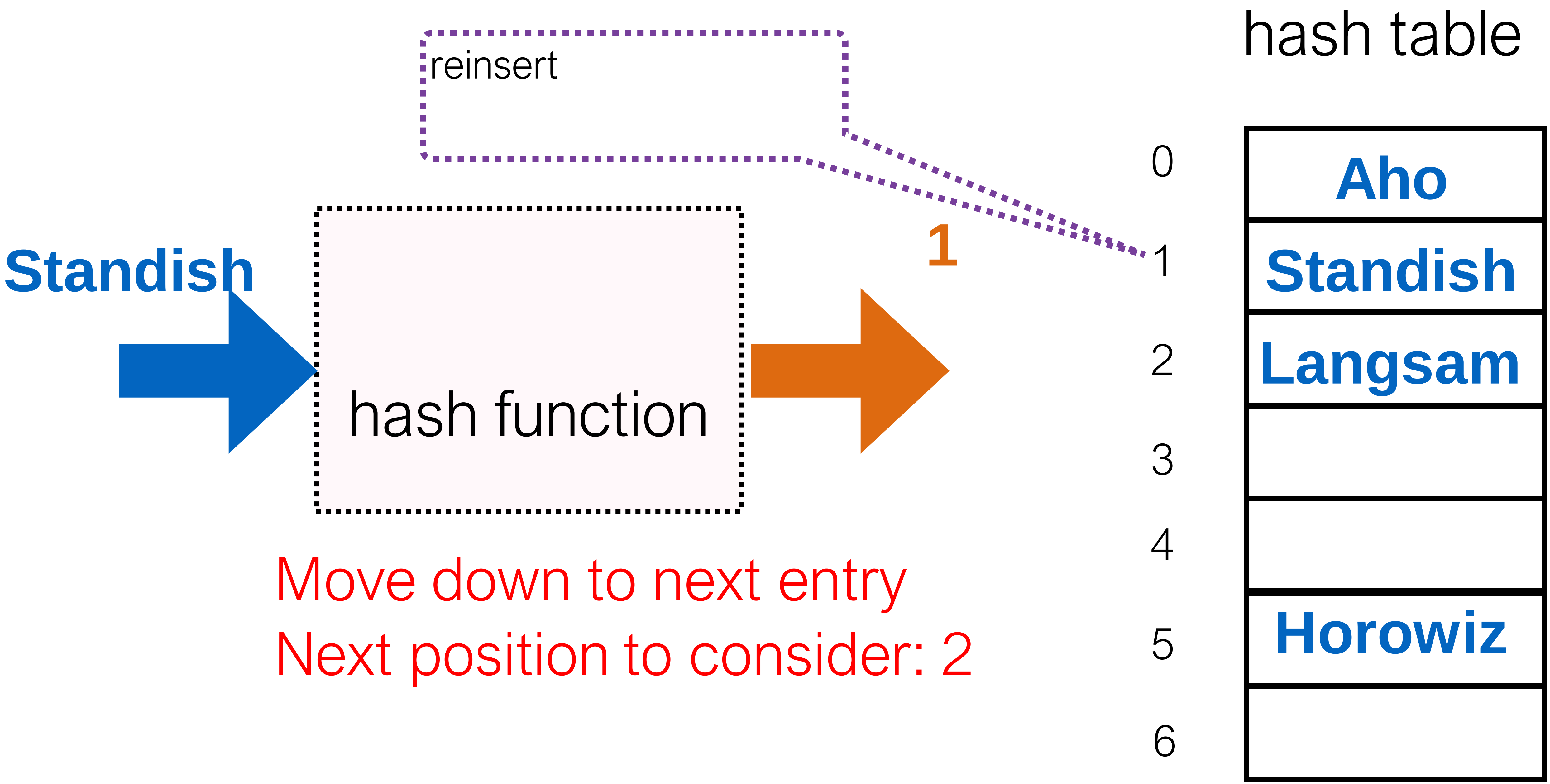
Example: delete

key: Kruse



Example: delete

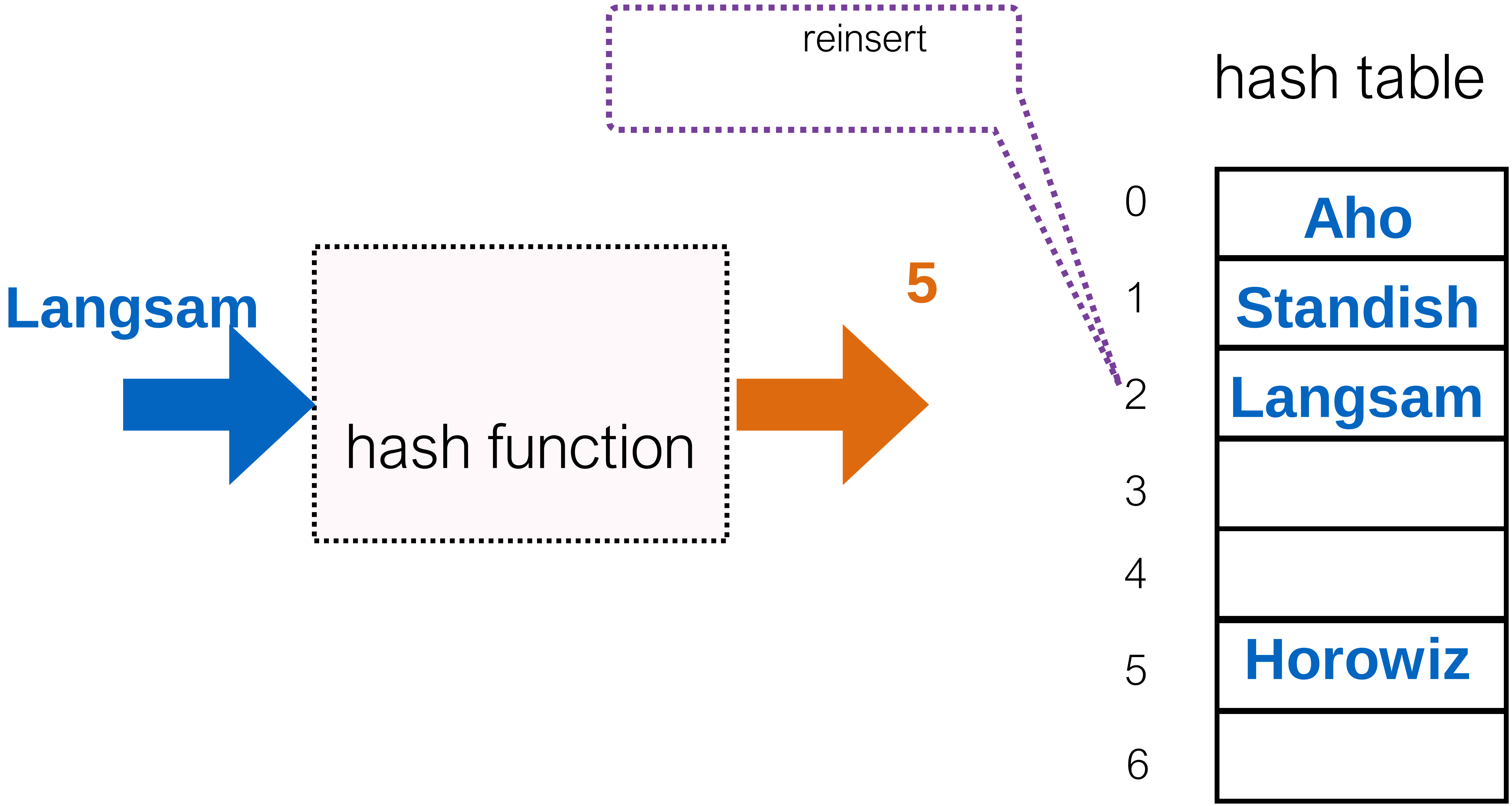
key: Kruse



Move down to next entry
Next position to consider: 2

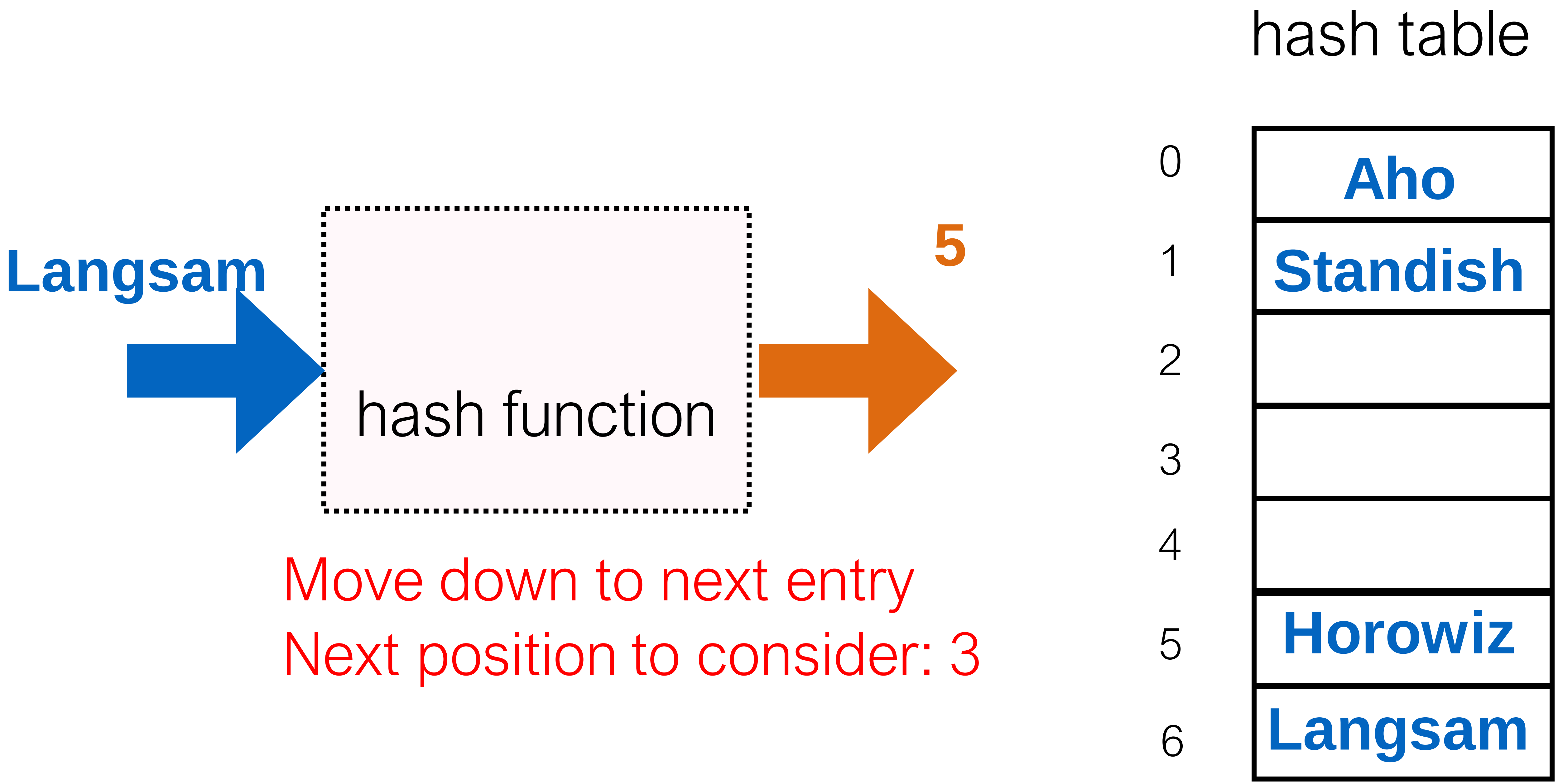
Example: delete

key: Kruse



Example: delete

key: Langsam

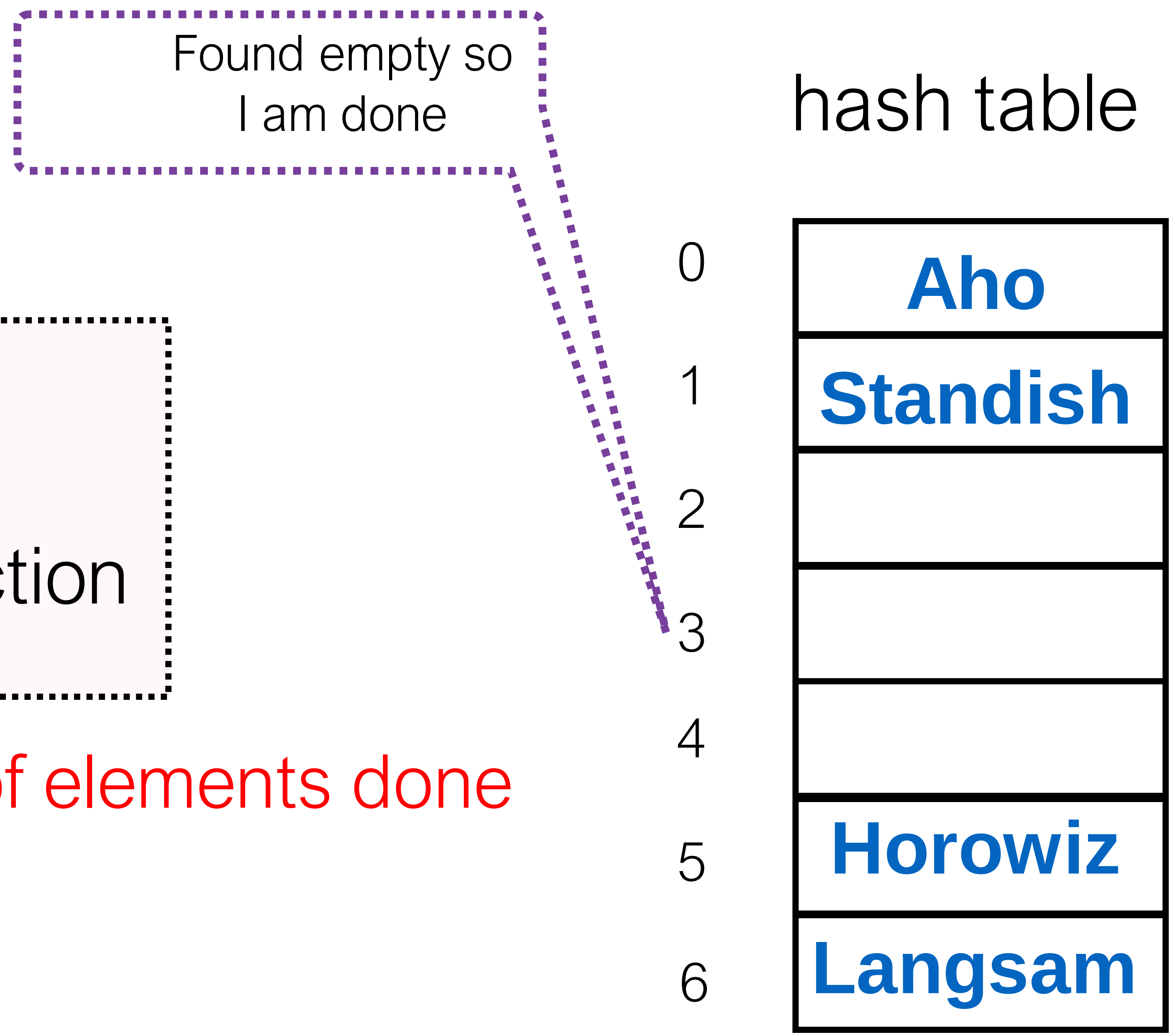


Move down to next entry
Next position to consider: 3

Example: delete

key: Langsam

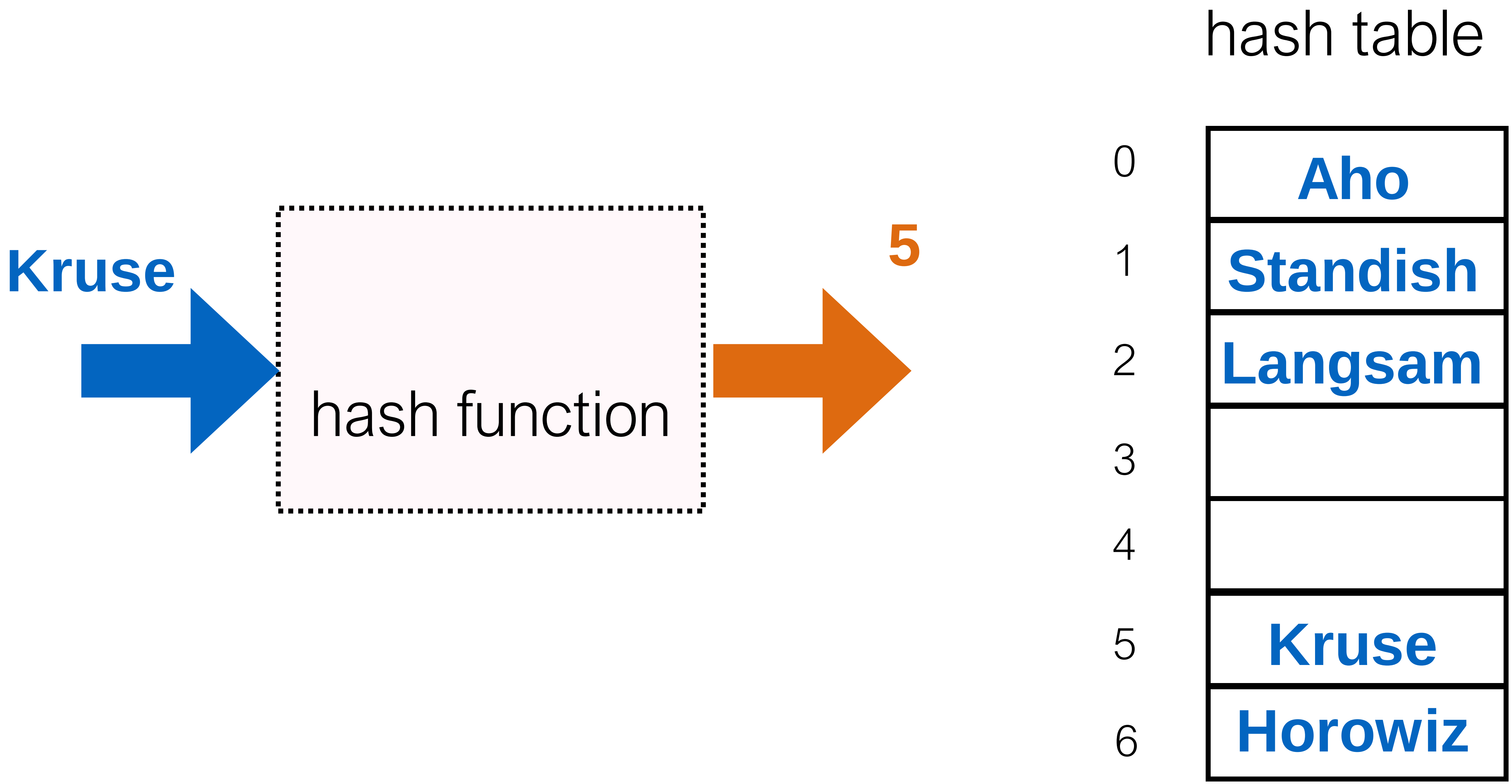
hash function



Reinserting of elements done

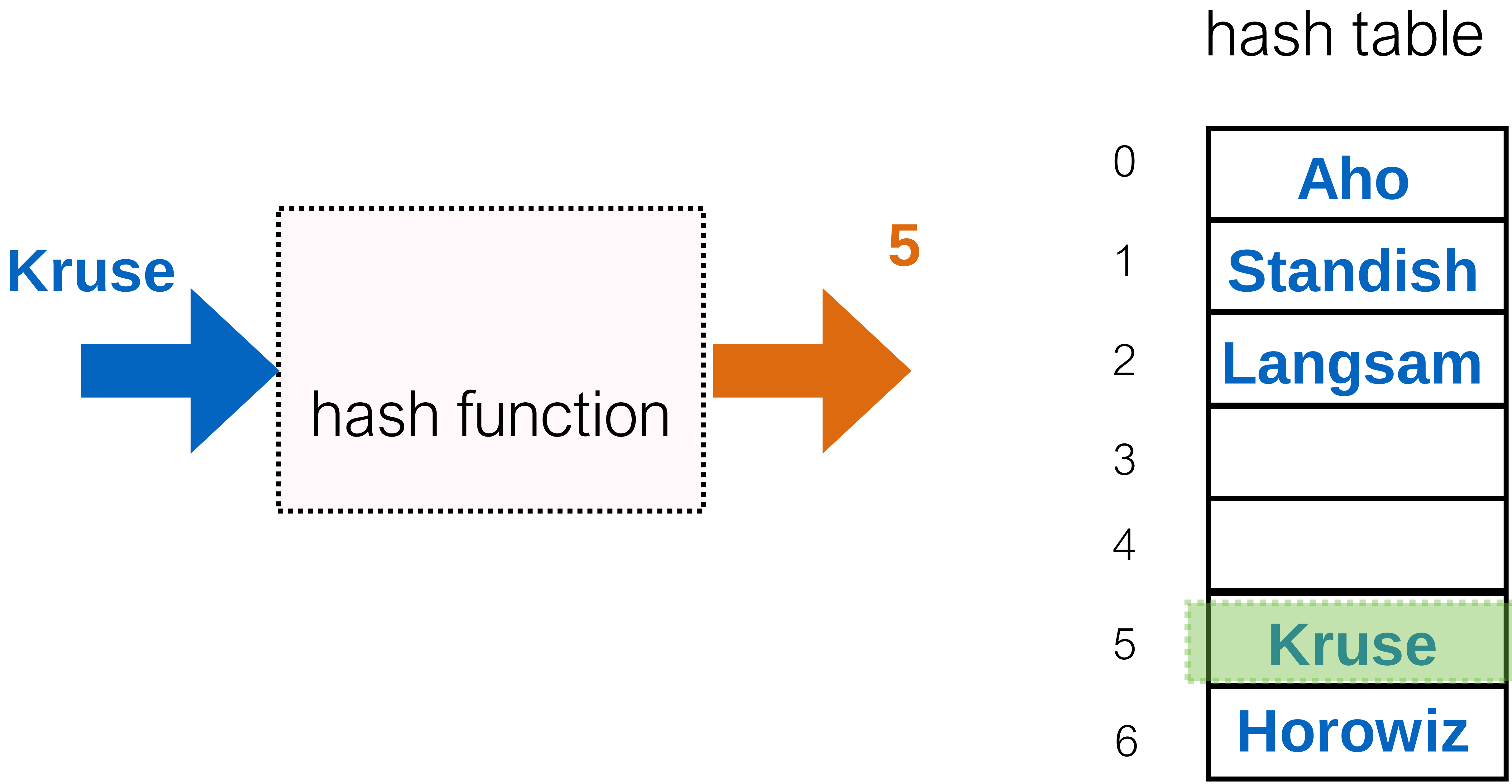
What if we didn’t reinsert?

Delete key: Kruse



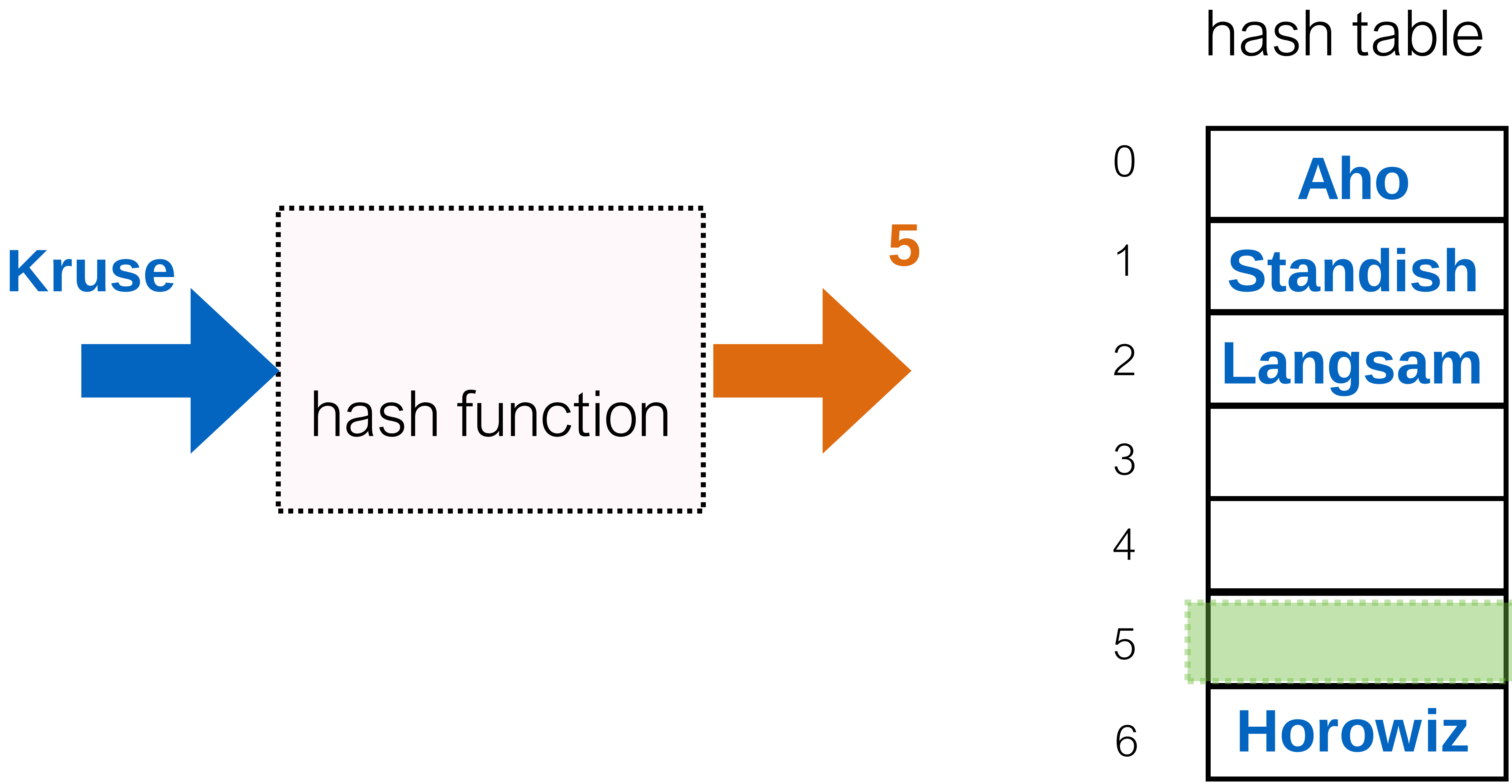
What if we didn't reinsert?

Delete key: Kruse



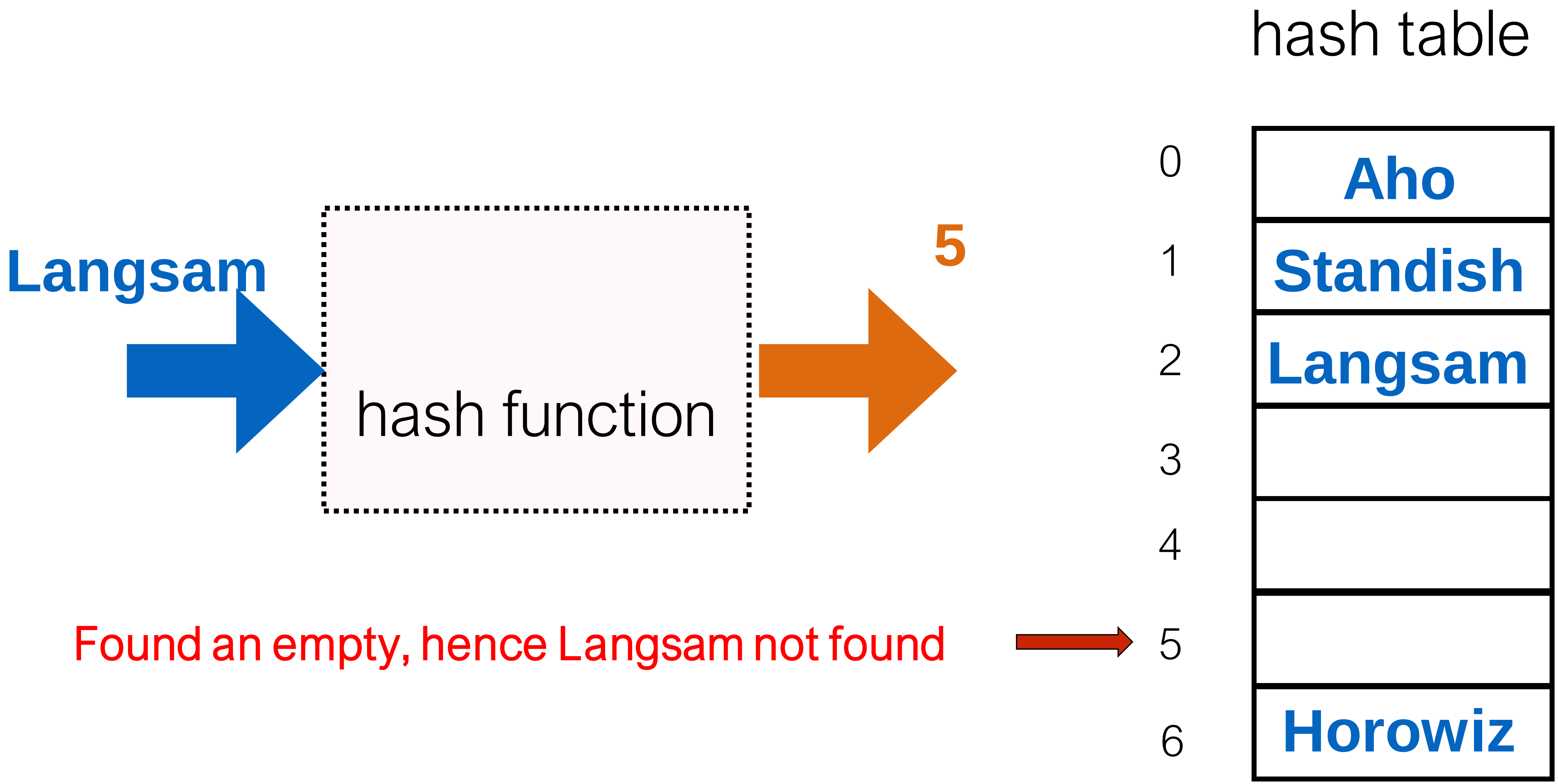
What if we didn't reinsert?

Delete key: Kruse



What if we didn't reinsert?

Search for key: Langsam



What if we didn't reinsert?

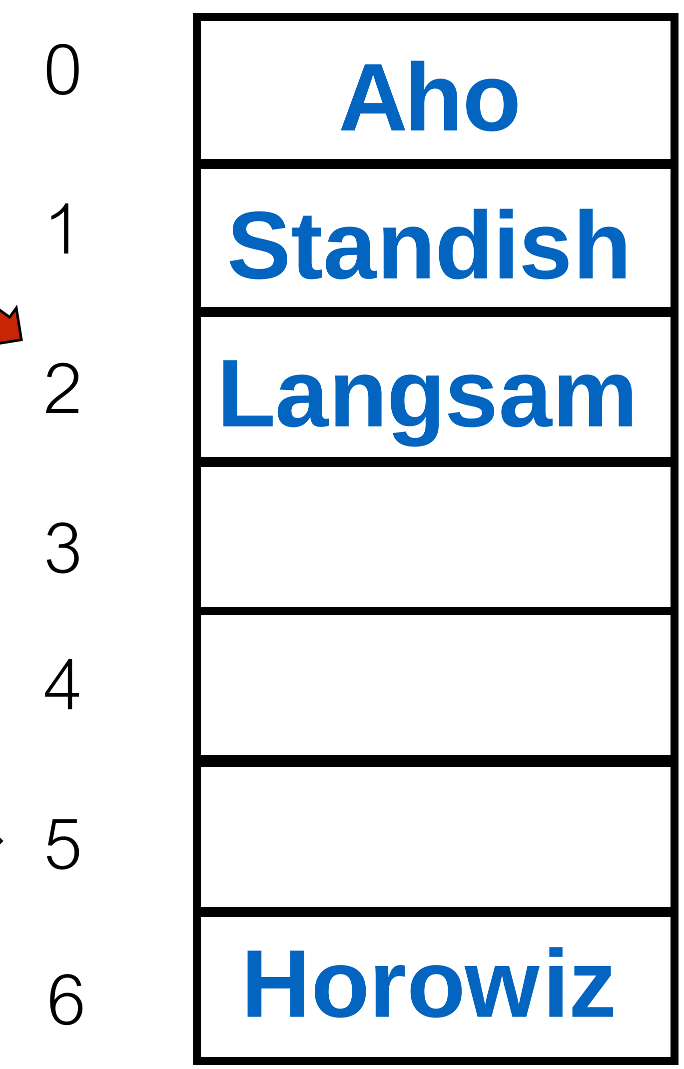
Search for key: Langsam

hash table

Even though Langsam is right here

Without reinserting, some items are
Not findable!

Found an empty, hence Langsam not found



Open Addressing: Linear Probing

- **Load factor**: total number of items/TABLESIZE
- **Cluster**: sequence of full hash table slots (i.e., without an empty slot)
- Clusters once formed, **tend to grow**…
  - Items that hash to a value within the cluster, get inserted at the end making it bigger
  - This might involve more than one hash value
- Cluster can form even when the load is small

Example of **cluster**

- All **5 elements** are part of a cluster
- Langsam, Kruse and Horowiz all have same hash value (5)
- Aho and Standish have values 0 and 1
- From then on, **any element mapped to 0,1,2,5 or 6 will be part of the cluster**

hash table

0	**Aho**
1	**Standish**
2	**Langsam**
3	
4	
5	**Kruse**
6	**Horowiz**

Linear Probing: Problems

- **Tendency for clustering** to occur as the load is > 0.5
- **Low speed on clustering**. We start under-delivering on the promise of constant time search and insert.
- Deletion of records is difficult
- If implemented in arrays – table may become full fairly quickly, resizing is time and resource consuming

Can we reduce clustering by taking bigger and bigger steps?

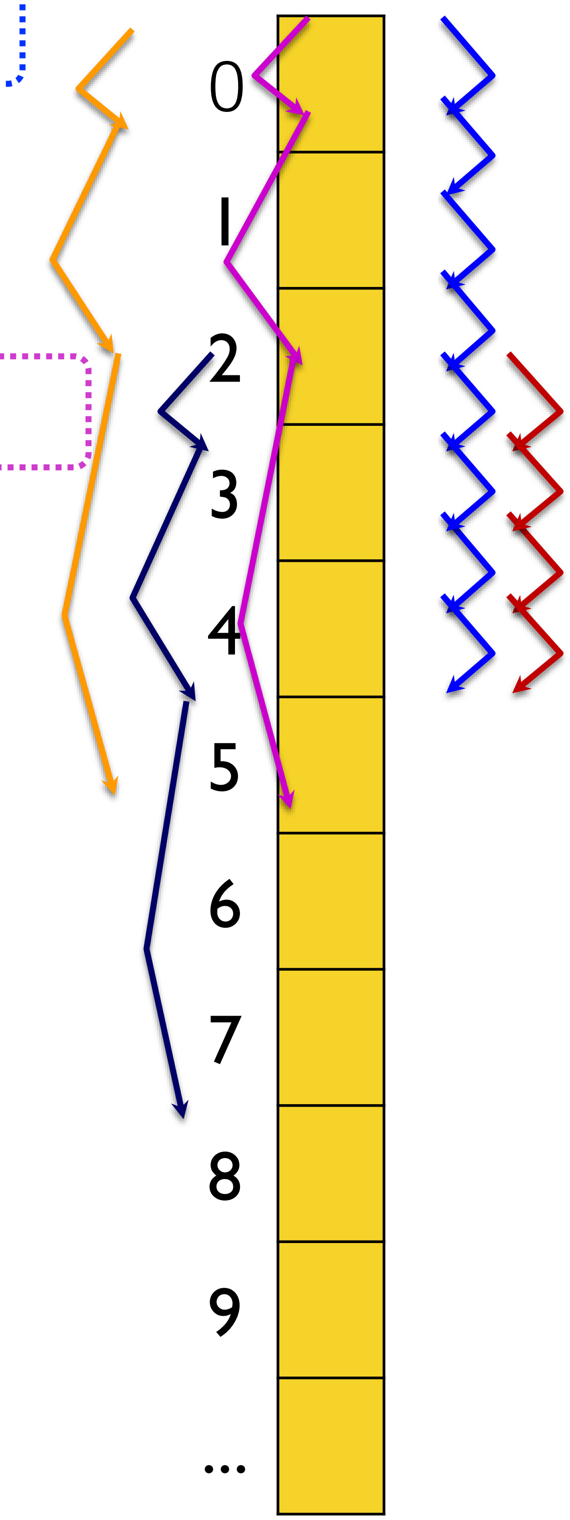
Open Addressing: **Quadratic Probing**

N+S
S is the step

- **Linear probing**: search at N+1, N+2, N+3…
- **Quadratic probing**: search at N+1, N+4, N+9, N+16

$N+S^2$

- primary clustering: keys with different hash values have same probe chains (as in linear probing)
- secondary clustering: keys with same hash values have the same probe chains
- **Advantage**: Quadratic probing eliminates primary clustering, but can suffer from secondary clustering
- There's a better method: **Double Hashing**



Open Addressing: **Double Hashing**

- If a collision occurs, use a **second hash function** to determine the **step**.

```
  position =
(hash1(key) + (collision nr)*hash2(key))%table_size
```

- <u>Second hash function</u>:

  →Cannot hash to 0

  →**Use primes**: table size & step size are co-primes (avoid revisiting the same positions)

- **Eliminates both primary and secondary clustering**

Double hashing example

$h_1(k) = k\%7$

$h_2(k) = 5 - (k\%5)$

Insert: 45, 22, 57, 33

hash table

0	
1	
2	
3	
4	
5	
6	

Double hashing example

$h_1(k) = k\%7$

$h_2(k) = 5 - (k\%5)$

Insert: 45, 22, 57, 33

hash table

0	
1	22
2	
3	45
4	57
5	33
6	

$45\%7 = 3$

$22\%7 = 1$

$57\%7 = 1$ $\quad 5-(57\%5)= 3 \quad 1+3 = 4$

$33\%7 = 5$

Double hashing example

$h_1(k) = k\%7$

$h_2(k) = 5 - (k\%5)$

Insert: 45, 22, 57, 33 ,**1**

hash table

0	
1	22
2	1
3	45
4	57
5	33
6	

$45\%7 = 3$

$22\%7 = 1$

$57\%7 = 1$ $\quad 5-(57\%5)= 3 \quad 1+3 = 4$

$33\%7 = 5$

$1\%7 = 1$ $\quad 5-(1\%5)= 4 \quad 1+4 = 5$

$2*(5-(1\%5))= 8 \quad 1+8 = 9$

$9\%7 =2$

Deletions in Open Addressing

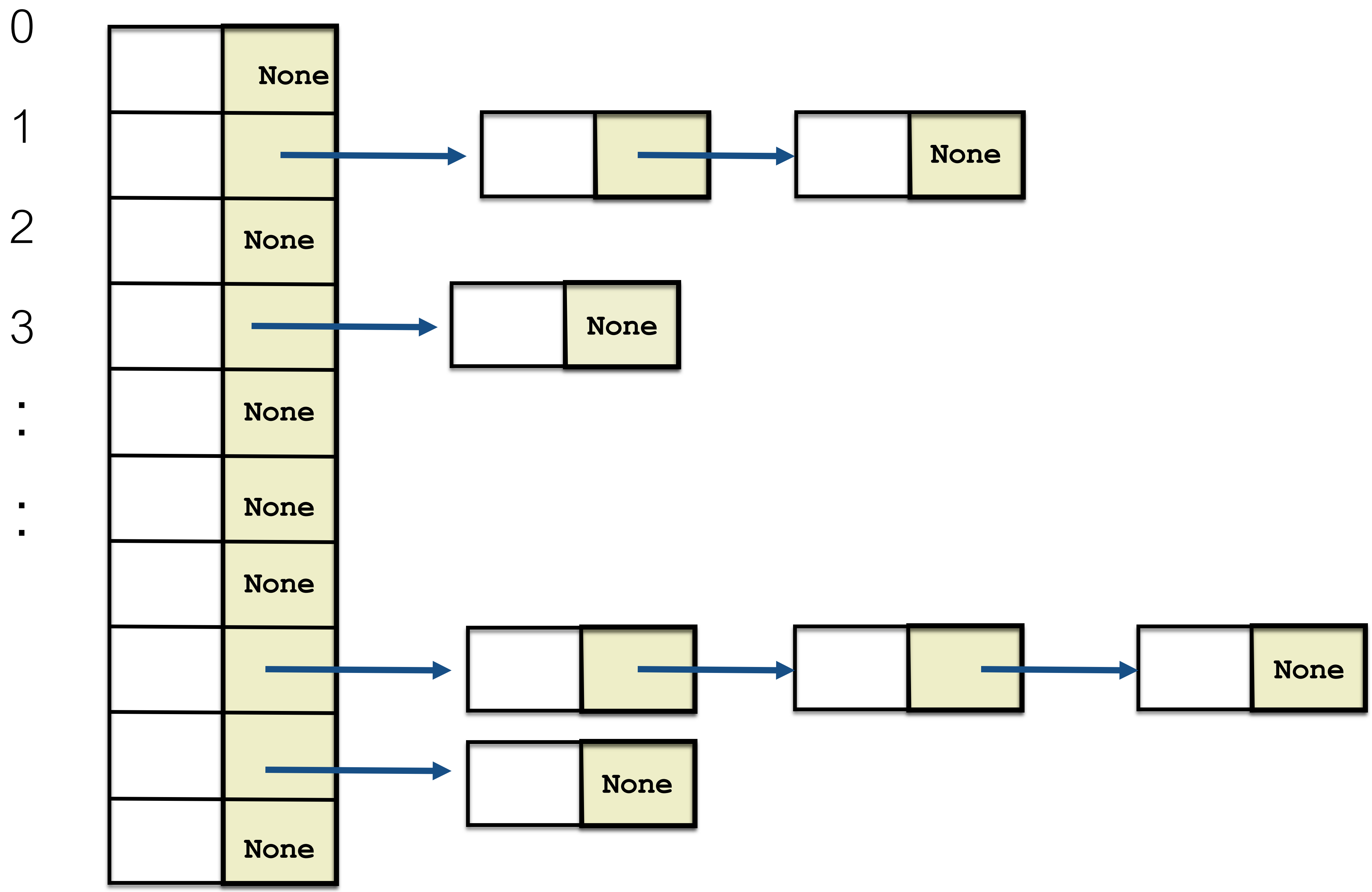
- Deleting is possible for linear probing, but difficult for quadratic or double hashing.
- **Lazy deletions:**
  - Keep a third element in the tuple to “mark” elements as deleted instead of erasing them
  - Marked as deleted elements are handled as empty positions during insertion
  - Marked as deleted elements are handled as occupied positions during search

Separate Chaining

- Uses a **Linked List at each position in the Hash Table**.

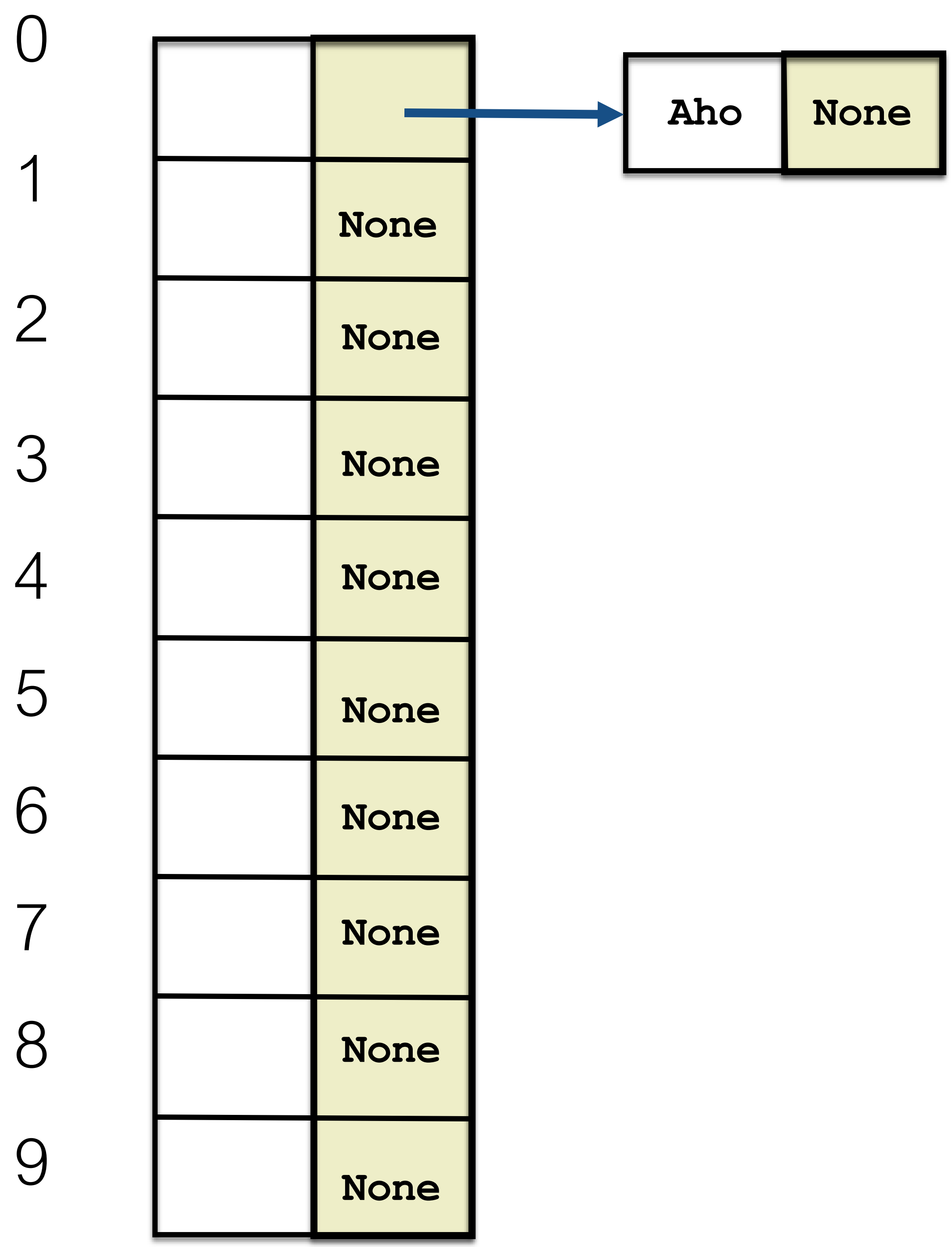
- Linked list at a position contains **all the items that ‘hash’** to that position.

hash table

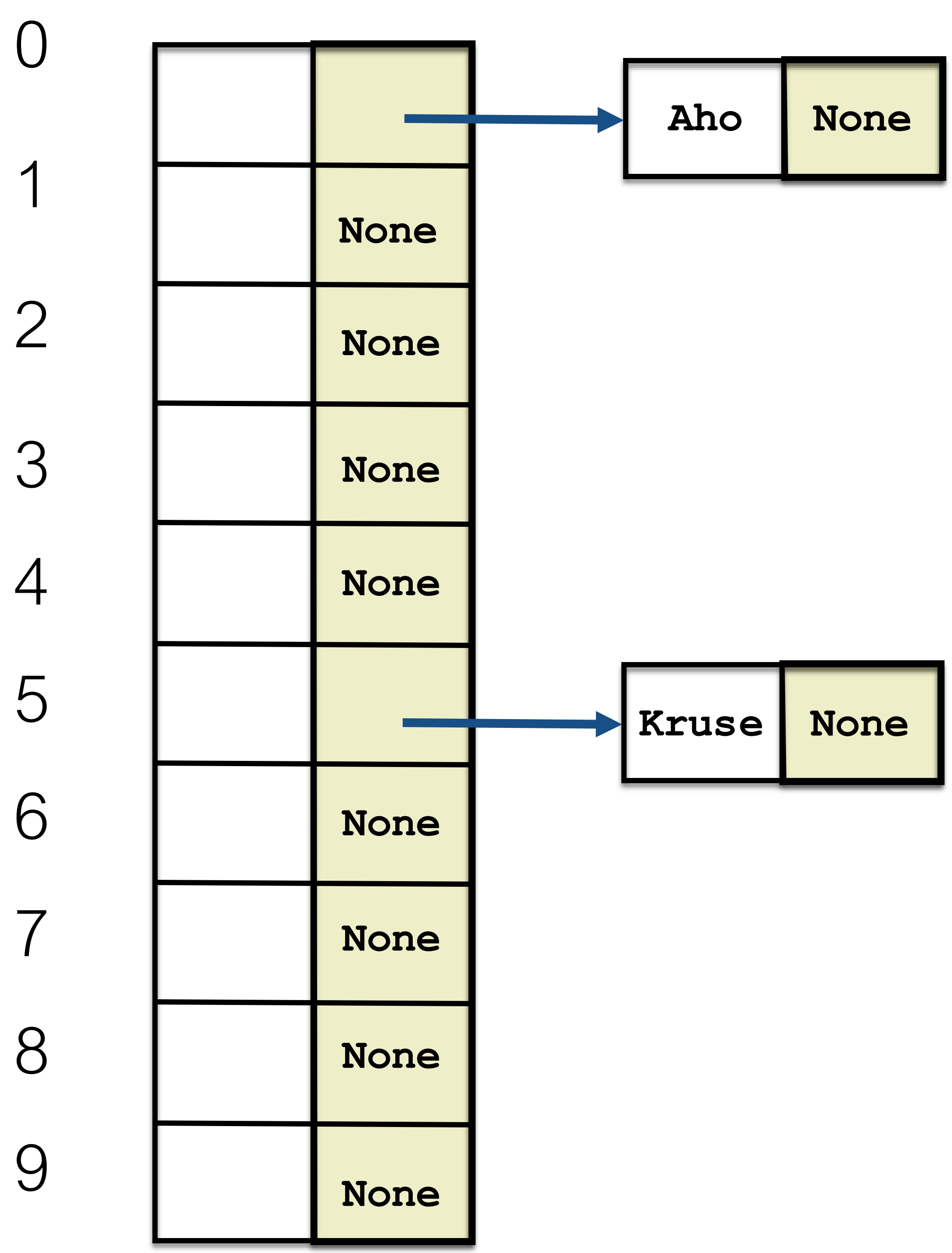


Aho, Kruse, Standish, Horowiz, Langsam, Sedgwick, Knuth

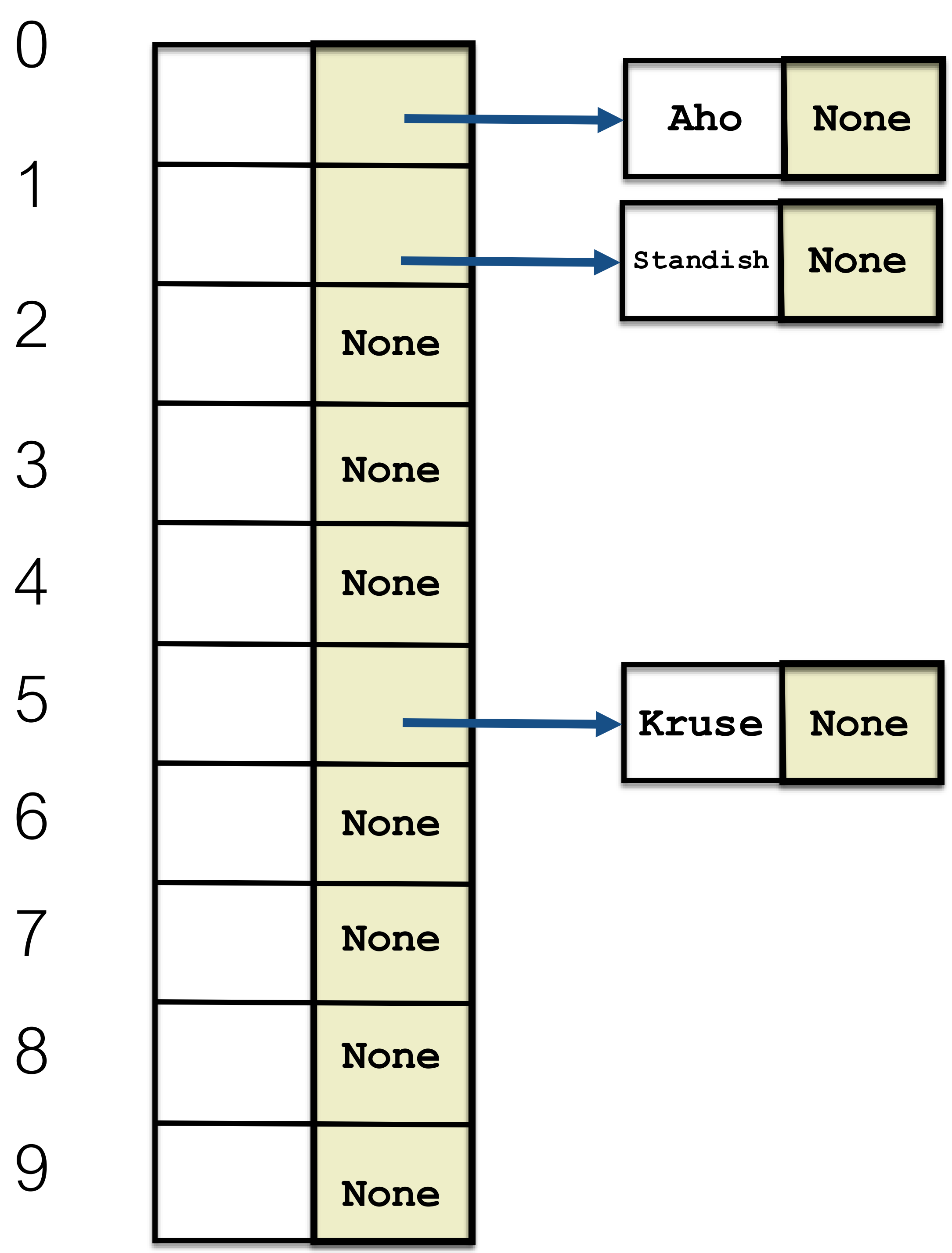
0, 5, 1, 5, 5, 2, 1



Aho, **Kruse**, Standish, Horowiz, Langsam, Sedgwick, Knuth
0, 5, 1, 5, 5, 2, 1

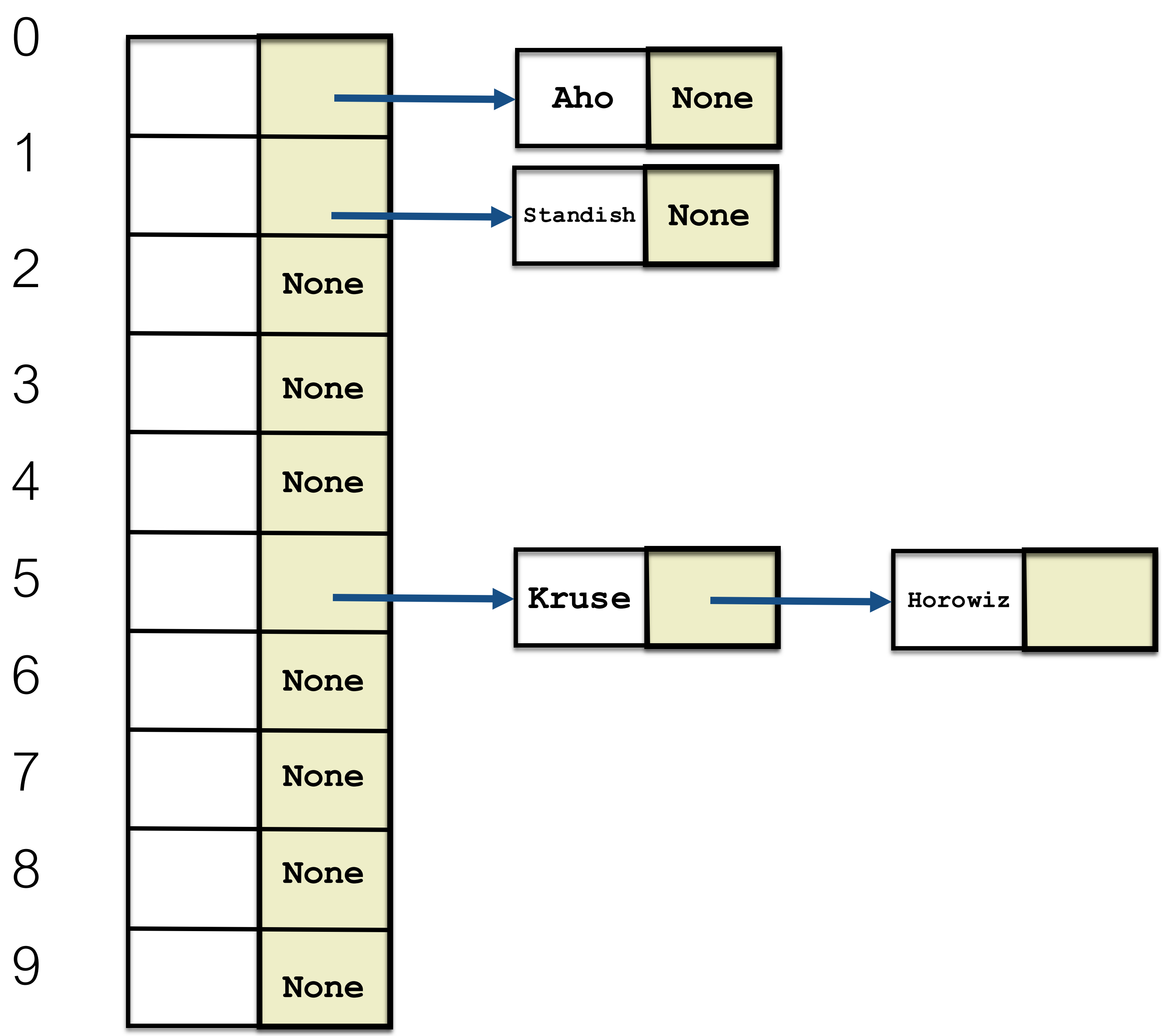


Aho, Kruse, **Standish**, Horowiz, Langsam, Sedgwick, Knuth
0, 5, 1, 5, 5, 2, 1



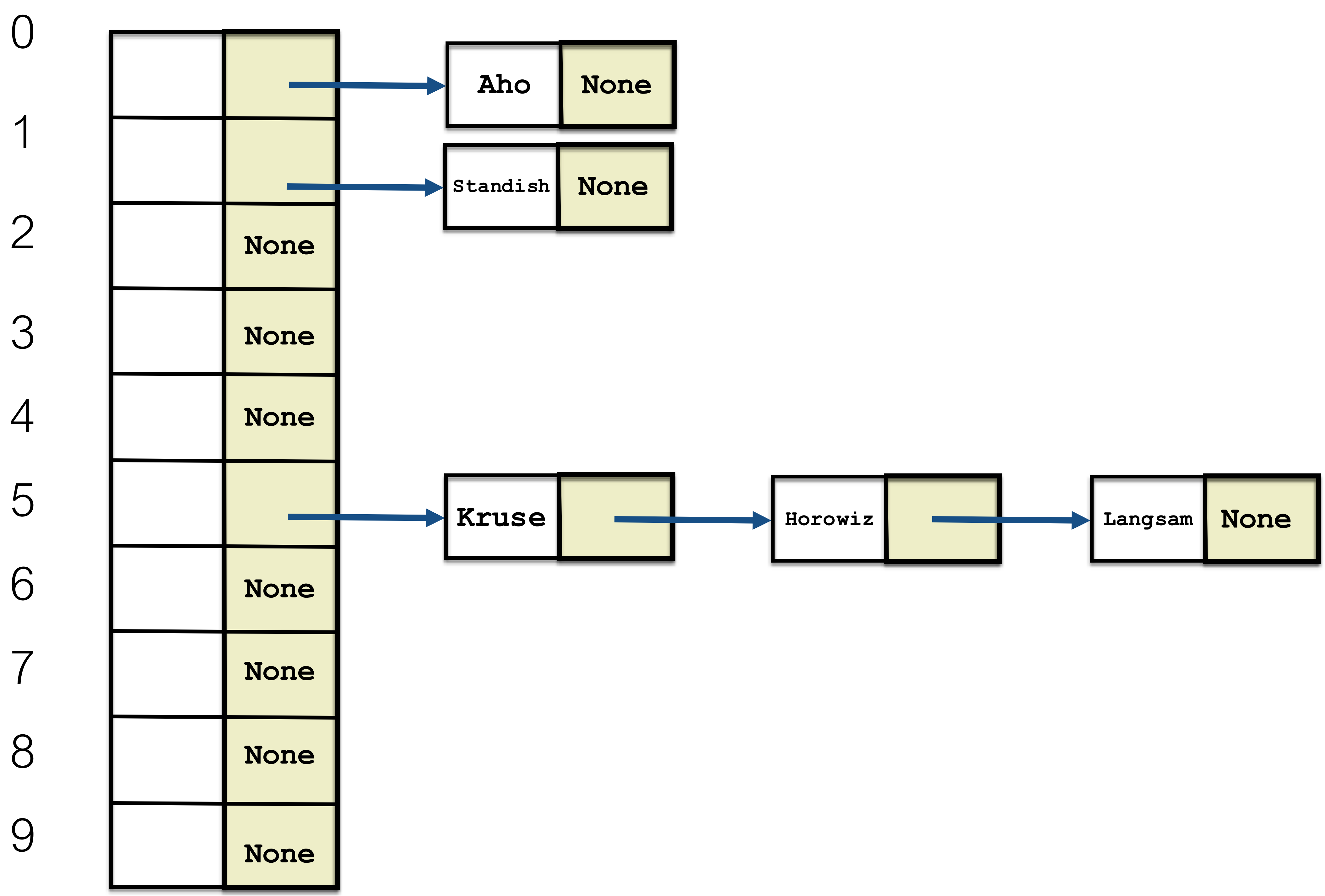
Aho, Kruse, Standish, **Horowiz**, Langsam, Sedgwick, Knuth

0, 5, 1, 5, 5, 2, 1



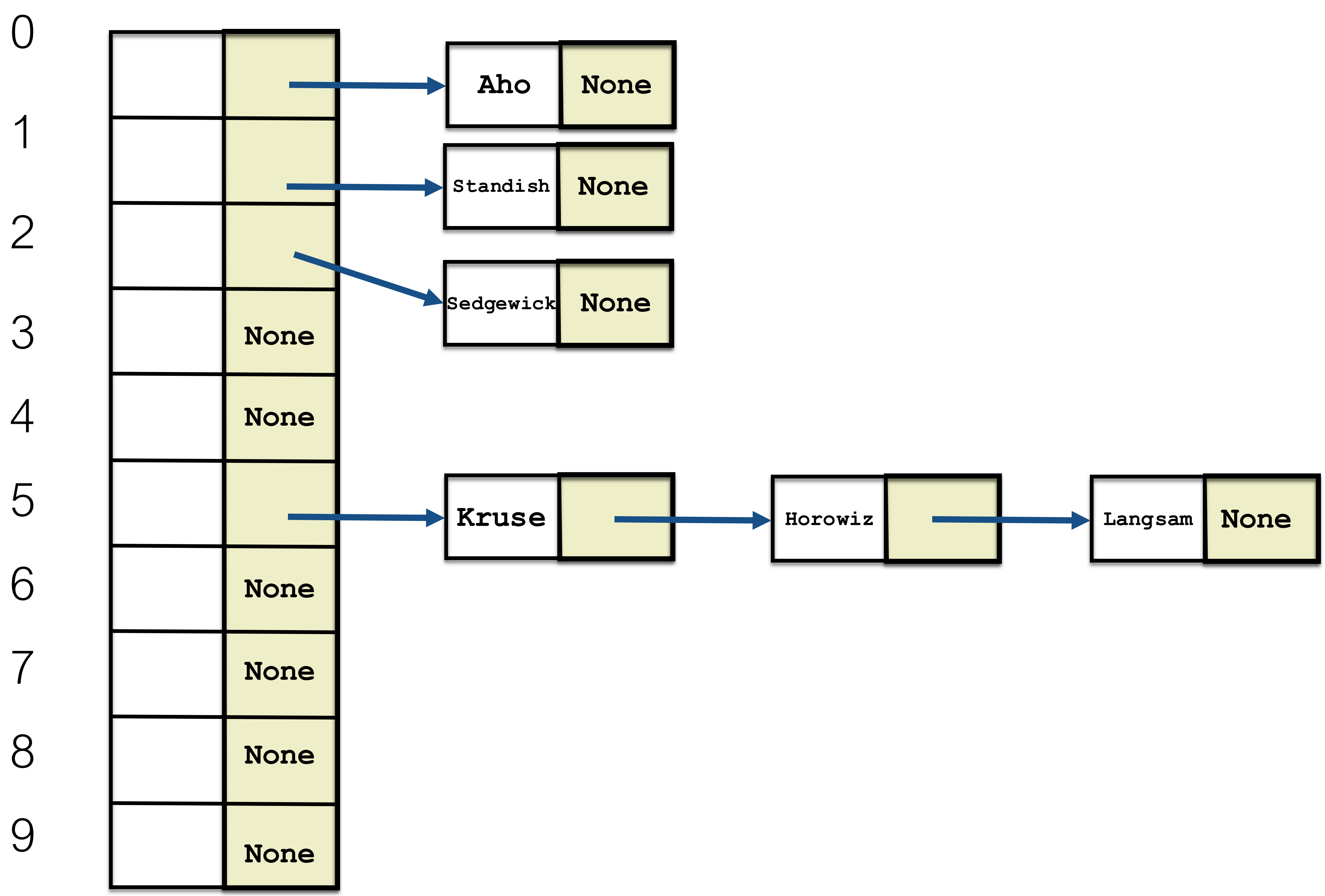
Aho, Kruse, Standish, Horowiz, **Langsam**, Sedgwick, Knuth

0, 5, 1, 5, 5, 2, 1



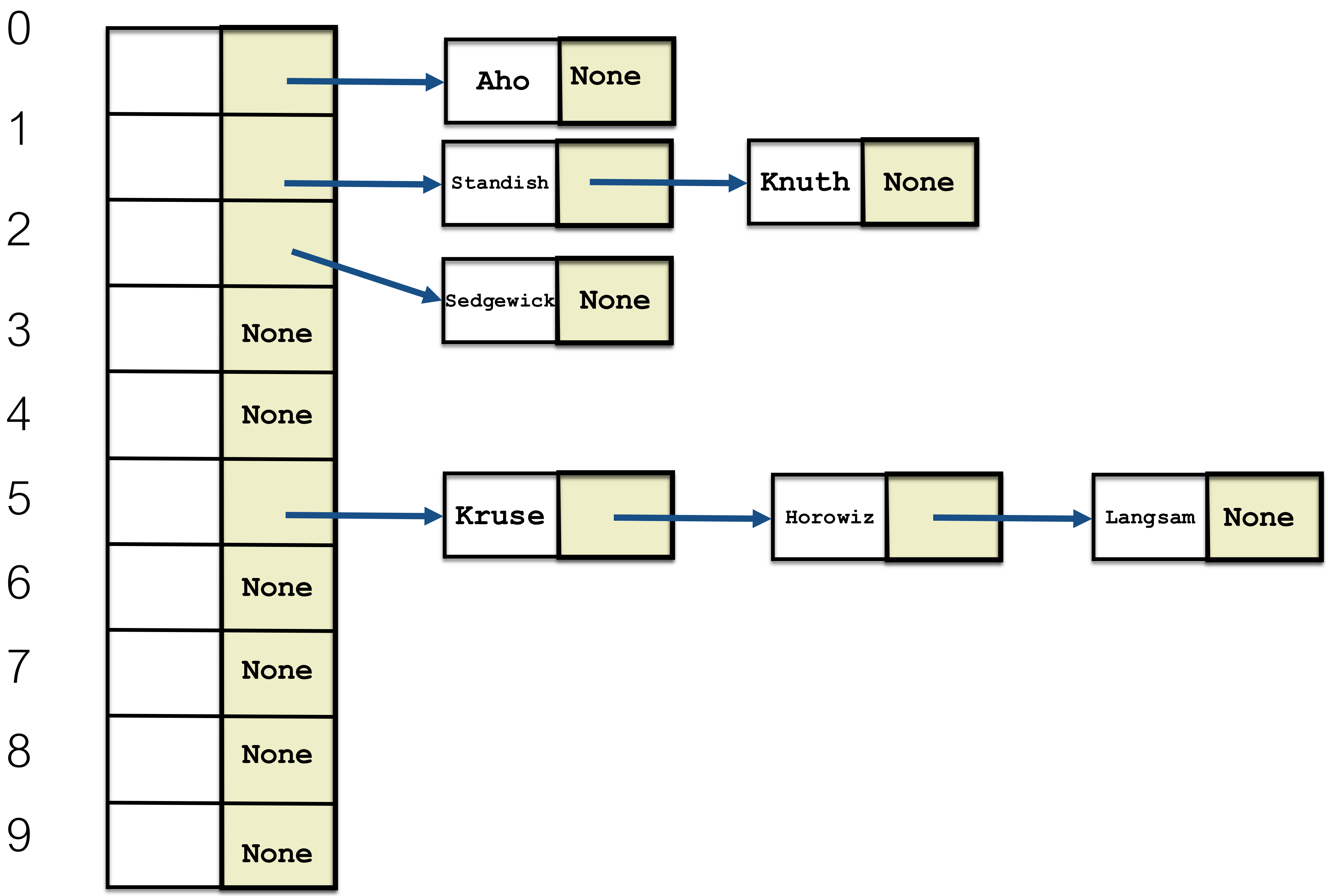
Aho, Kruse, Standish, Horowiz, Langsam, **Sedgwick**, Knuth

0, 5, 1, 5, 5, 2, 1



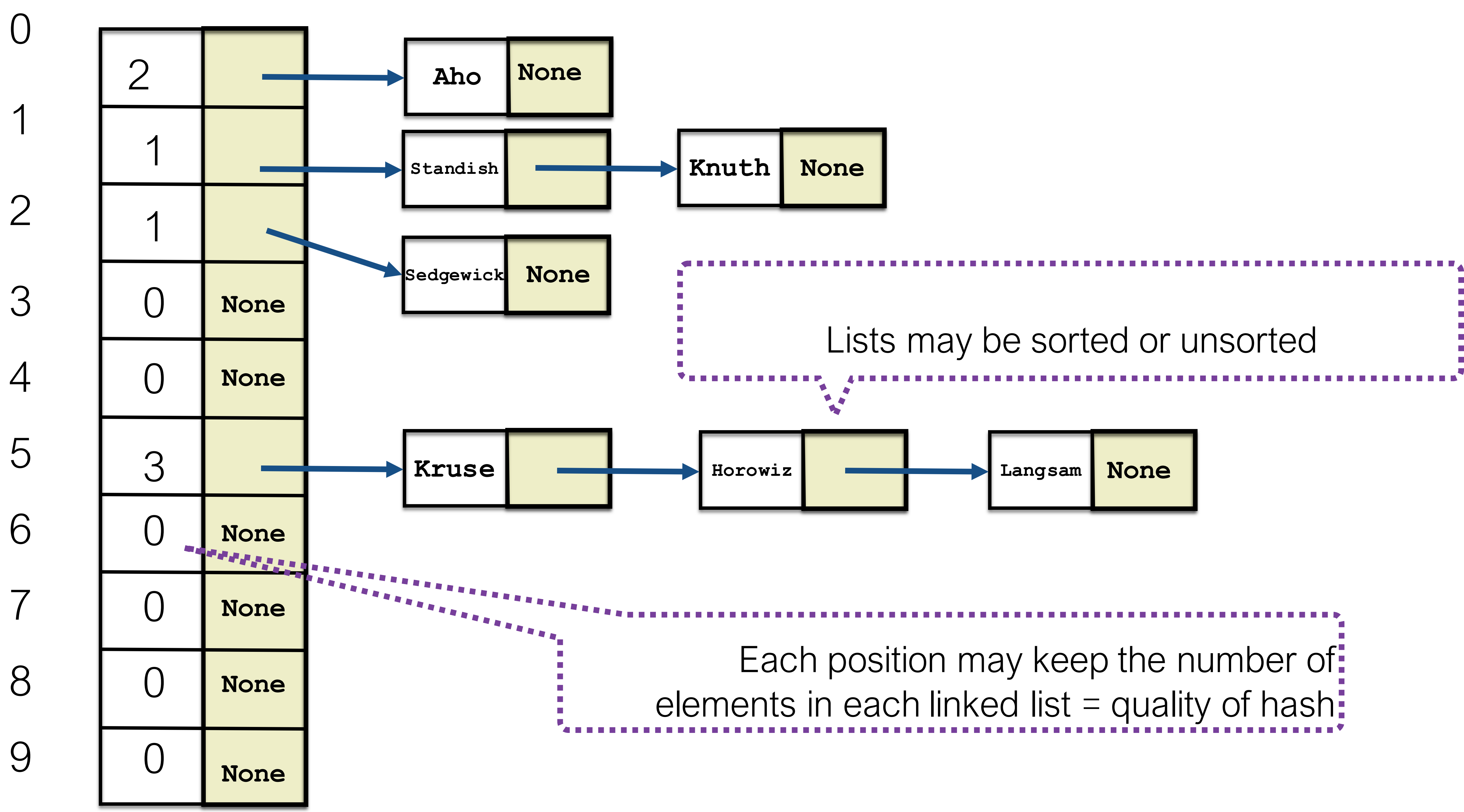
Aho, Kruse, Standish, Horowiz, Langsam, Sedgwick, **Knuth**

0, 5, 1, 5, 5, 2, 1



Aho, Kruse, Standish, Horowiz, Langsam, Sedgwick, **Knuth**

0, 5, 1, 5, 5, 2, 1



Separate Chaining

- Apply hash function to get a position N in the array
- **Insert**: Insert key into the Linked List at position N
- **Search**: Search for key in the Linked List at position N
- **Delete**: Search for key; delete the node in the Linked List at position N

Separate Chaining

Advantages:

- Conceptually simpler
- Insertions and deletions are easy and quick
- Naturally resizable, allows a varying number of records to be stored

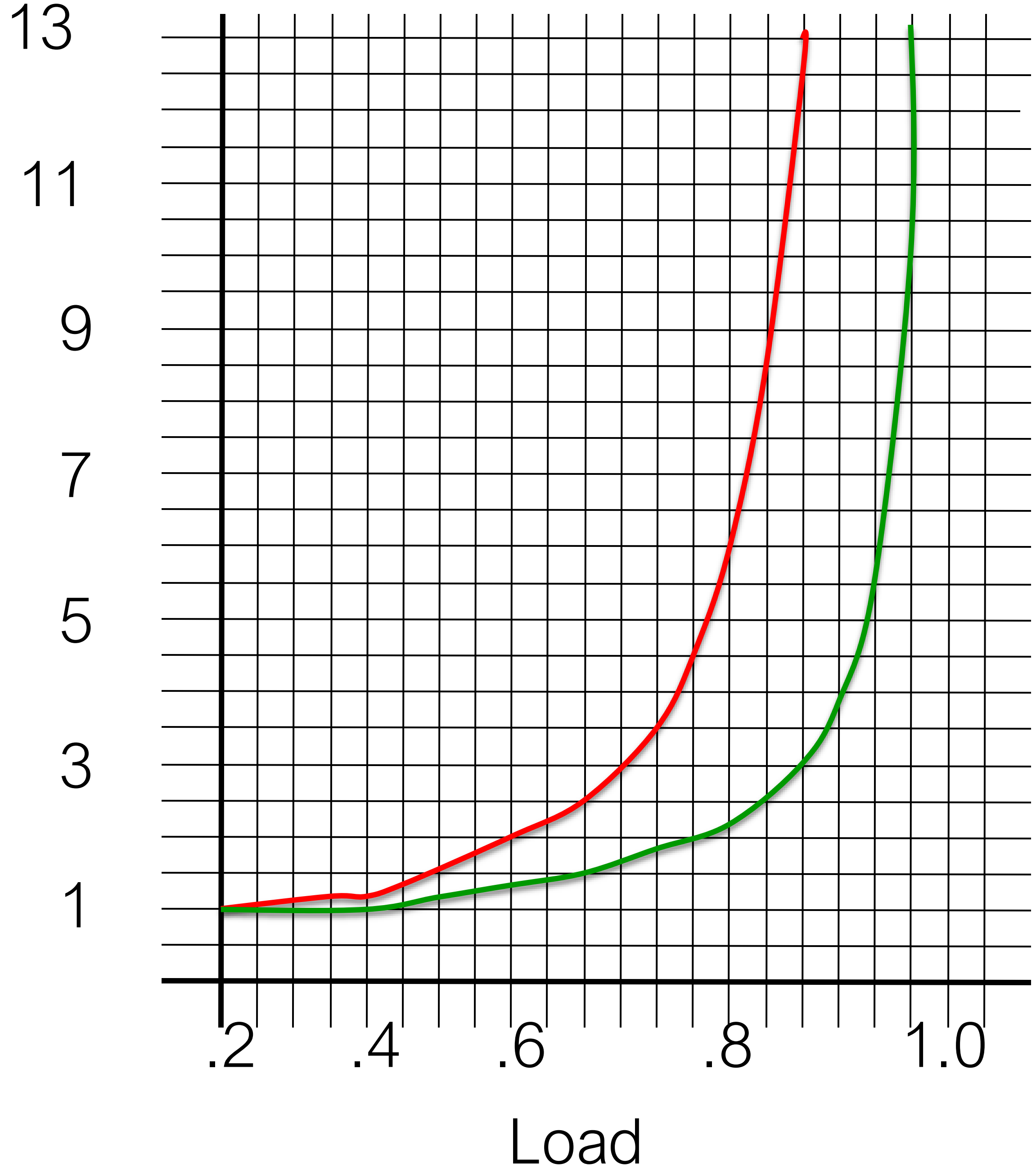
Disadvantages

- Requires extra space for the links
- Requires linear search for elements in a list

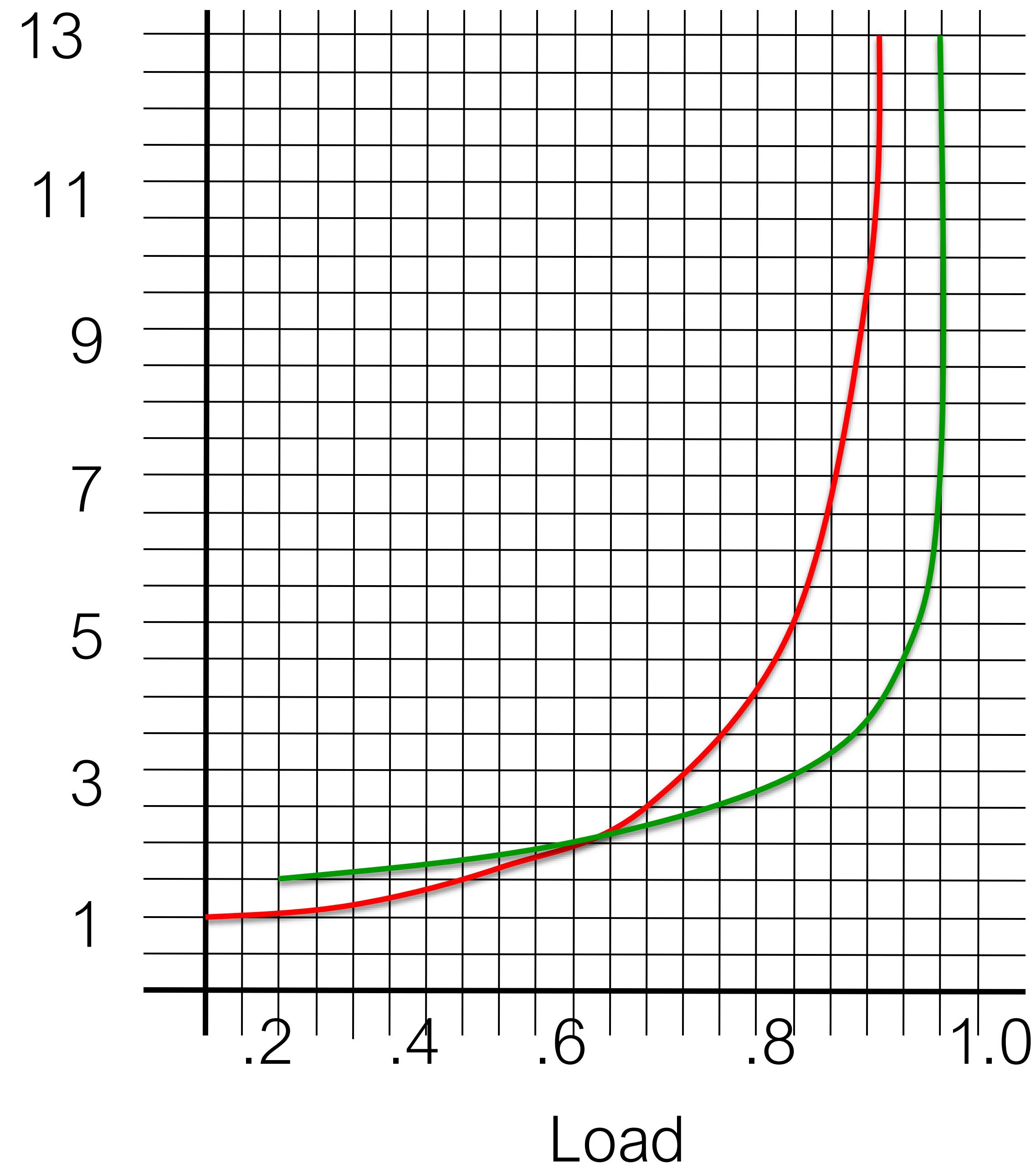
Comparison: general

- Choice depends on the particular application
  - Linear Probing: fast if memory allows for a large table.
  - Double hashing: efficient use of memory but needs to compute a second hash
  - Separate chaining: simple, extra memory, resizable, fast insert and fast delete
- When the load approaches 1: double hashing far outperforms linear probing
- **Open addressing**: keep load under 2/3 …. even better 1/2
- **Separate chaining**: efficiency degrades linearly with load

Linear probe chain length



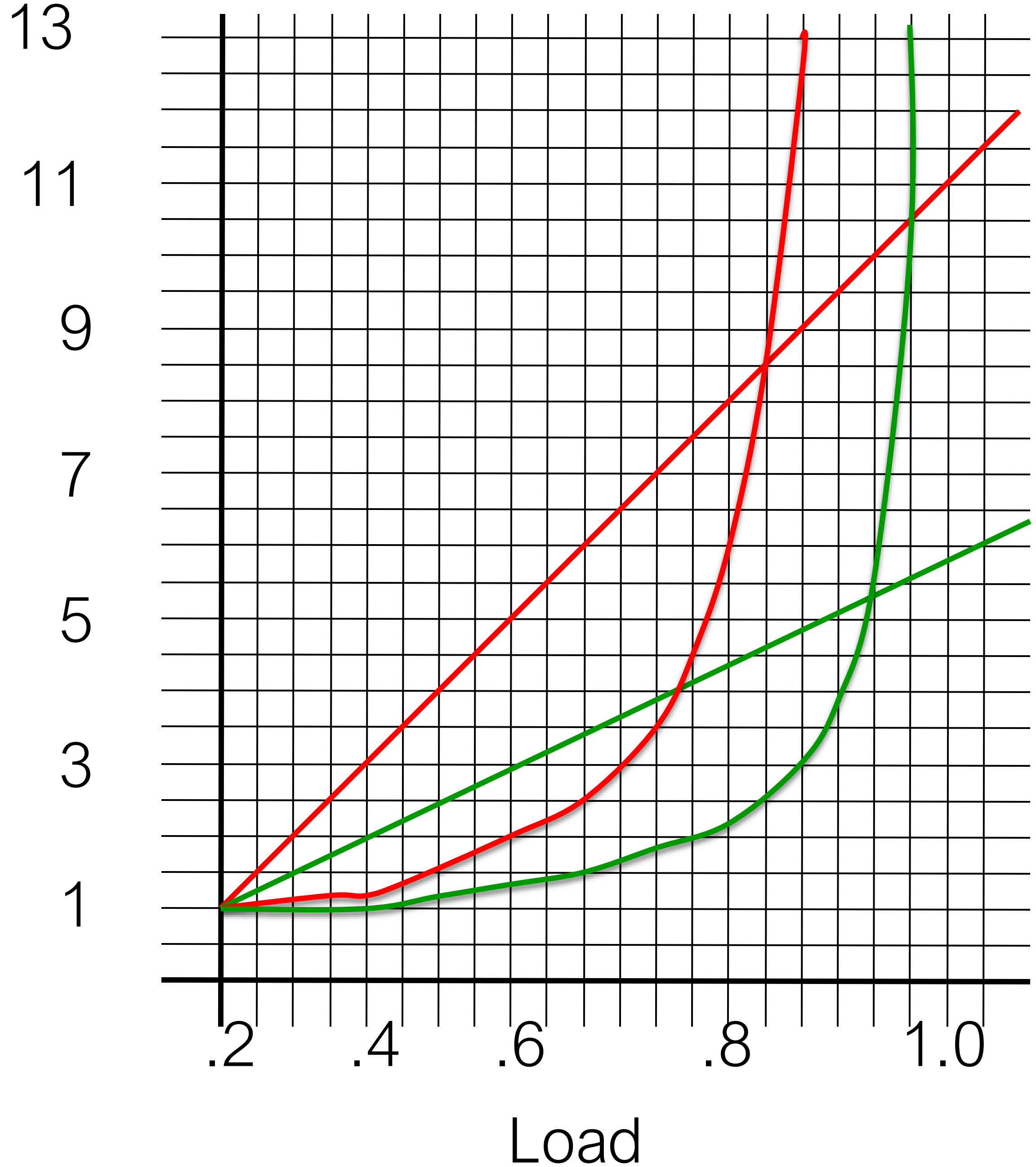
Double hashing probe chain length



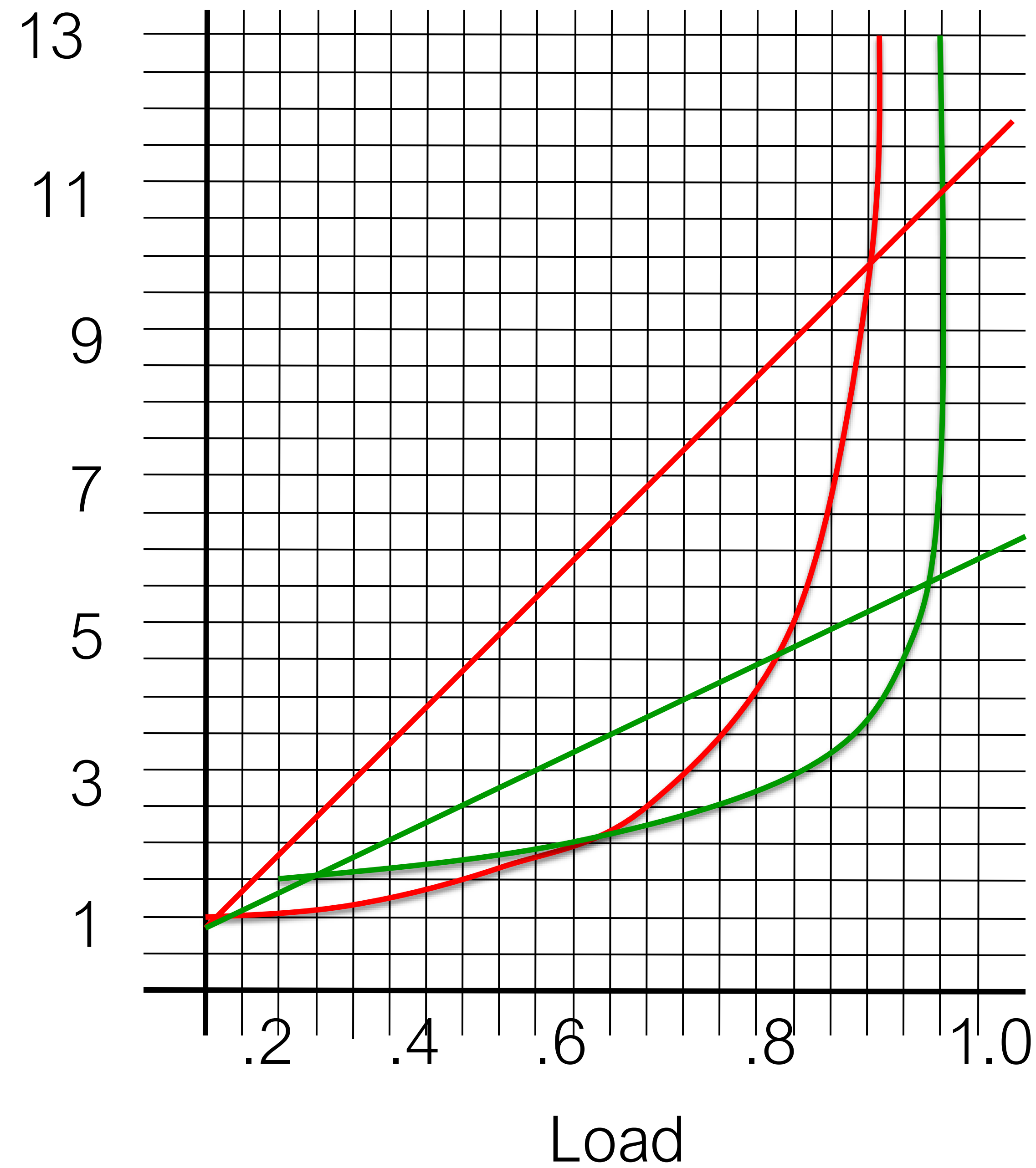
found/not found

Superimposing Separate Chaining

Linear probe chain length



Double hashing probe chain length



found/not found

Dynamic Hashing

- Each time the load in an open address method gets greater than desirable:

  → 1/2 for linear probing
  → 2/3 for double hashing

  we expand the table by doubling its size and adding one
  (so it is still an odd number)

- At increasing the size:

  - Create a new array

  - Rehashing **every item** in the old table into the new one
    (due to the use of TABLESIZE in the hash function)

Conclusion

- Hash Tables are one of the most used data types
- You have a very good chance of using them in your career
- They are very simple conceptually.
- A significant amount of experimental evaluation is usually needed to **fine-tune the hash function** and the TABLESIZE
- Choice of hash function, collision handling and load factor are crucial to maintaining an efficient hash table