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

Lecture 29
Hash Tables

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

Objectives for this lecture

- To understand what is expected from a Hash Table

- To understand
  - What is a hash function
  - The properties of a good hash function

- To be able to implement simple hash functions

- To understand the challenges posed by collisions and start looking at solutions

Dictionary ADT

- Permits access to <u>data items</u> by content, e.g., a key.
- Operations:
  - ➞ Search
  - ➞ Insert
  - ➞ Deleta

__dict__

```python
class Coffee:
    def __init__(self, coffee_type, price):
        self.coffee_type = coffee_type
        self.price = price
```

```
>>> from lecture_24 import Coffee
>>> new_coffee = Coffee("latte", 4.8)
>>> new_coffee.__dict__
{'coffee_type': 'latte', 'price': 4.8}
>>> 
```

keys

values

```
>>> a = dict()
>>> a[123465] = "Julian"
>>> a[133123] = "Nicole"
>>> a[982211] = "David"
>>>
>>> a
{123465: 'Julian', 133123: 'Nicole', 982211: 'David'}
```

keys

values

insert

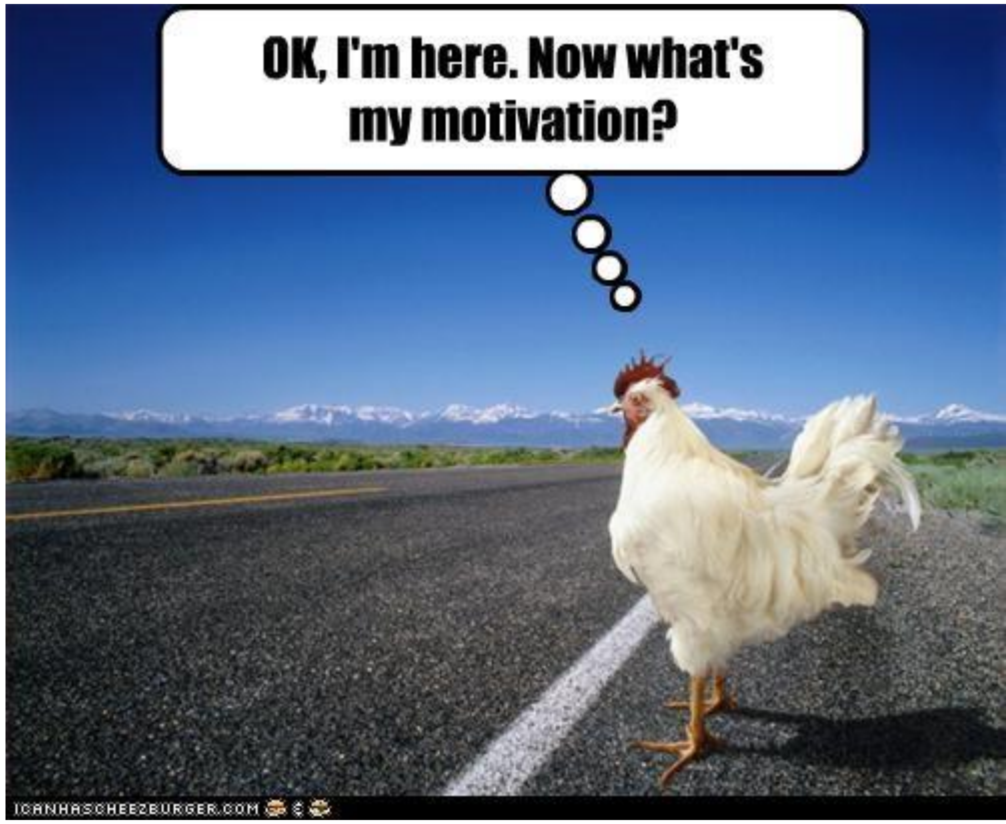
```
>>> a = dict()
>>> a[123465] = "Julian"
>>> a[133123] = "Nicole"
>>> a[982211] = "David"
>>>
>>> a
{123465: 'Julian', 133123: 'Nicole', 982211: 'David'}
>>>
>>>
>>> a[133123]
'Nicole'
```

search

Python dictionaries are implemented using Hash Tables

Hash Tables: Motivation

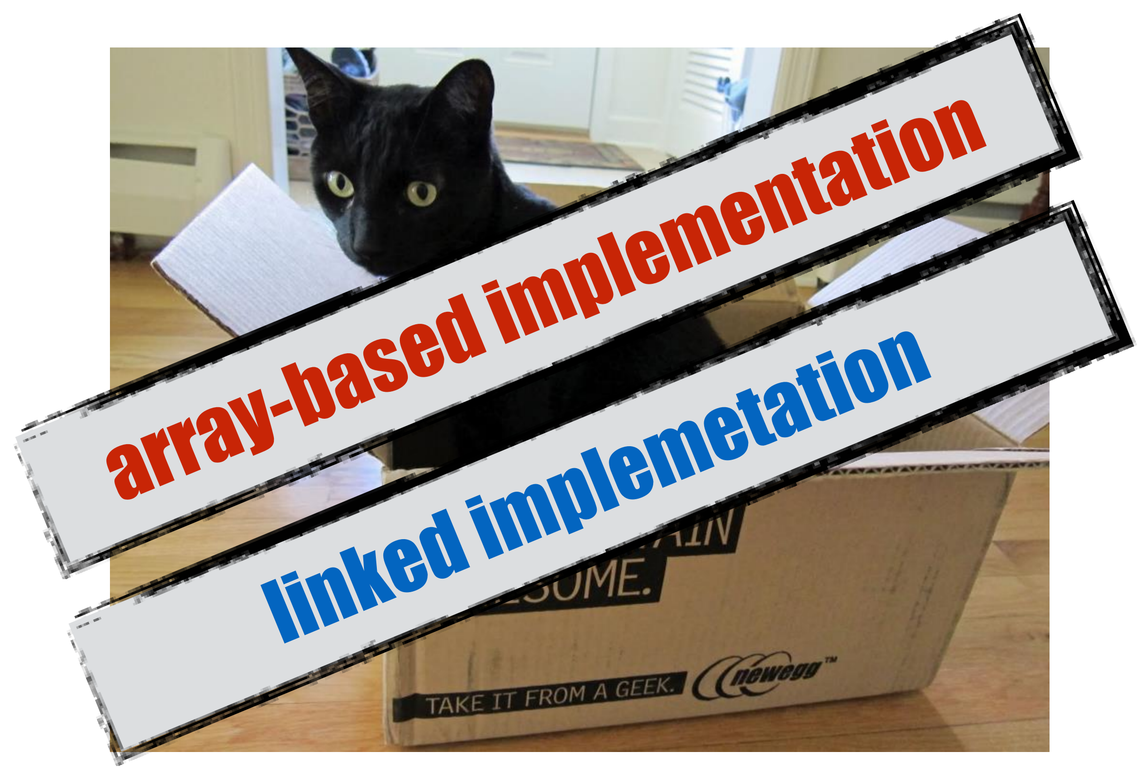
- Assume we are interested in storing a very significant amount of data (a big N)
- Assume we are going to need to perform the following <u>operations</u> relatively often:
  - Search for an item
  - Insert a new item
  - You might also want to delete an item (optional)



- But we do not need to traverse them in a particular order or sort them (at least not often)

Container ADTs

- Stores and removes items independent of contents.
- Examples include:
  - List ADT
  - Stack ADT
  - Queue ADT.
- Core operations:
  - add item
  - delete item
  - search



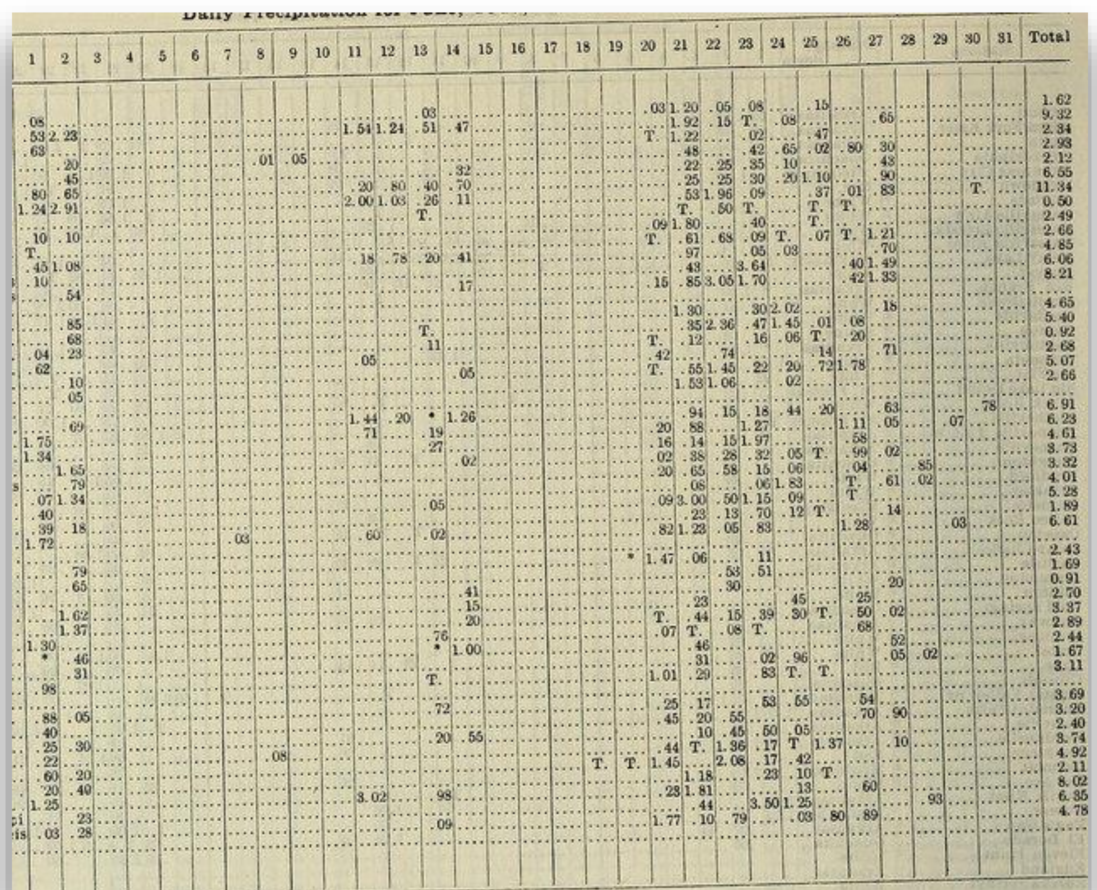
https://3.bp.blogspot.com/-cH5oY27UbAA/T4mvyNynaSI/AAAAAAAACYg/K1n6s3yg7B8/s1600/02_Felix_cat_MayContainAwesome_box.jpg

Can’t we do that already?

- Stacks:
  - Follow LIFO
  - Therefore, not suitable for searching/deleting
- Queues:
  - Follow FIFO
  - Therefore, not suitable for searching/deleting
- Unsorted Lists:
  - Searching: O(1) best and O(N) worst (*Comparison)
  - Adding: O(1) best and worst
  - Deleting: O(1) best and O(N) worst (*Comparison)
- Sorted Lists (worst case and *Compare):
  - Searching: O(N) if linked lists O(log N) if array (*Comparison)
  - Adding: O(N) in linked lists and arrays
  - Deleting: O(N) in linked lists and arrays (*Comparison)

Wouldn’t it be great if we could search in constant time?

Hash Tables: aim



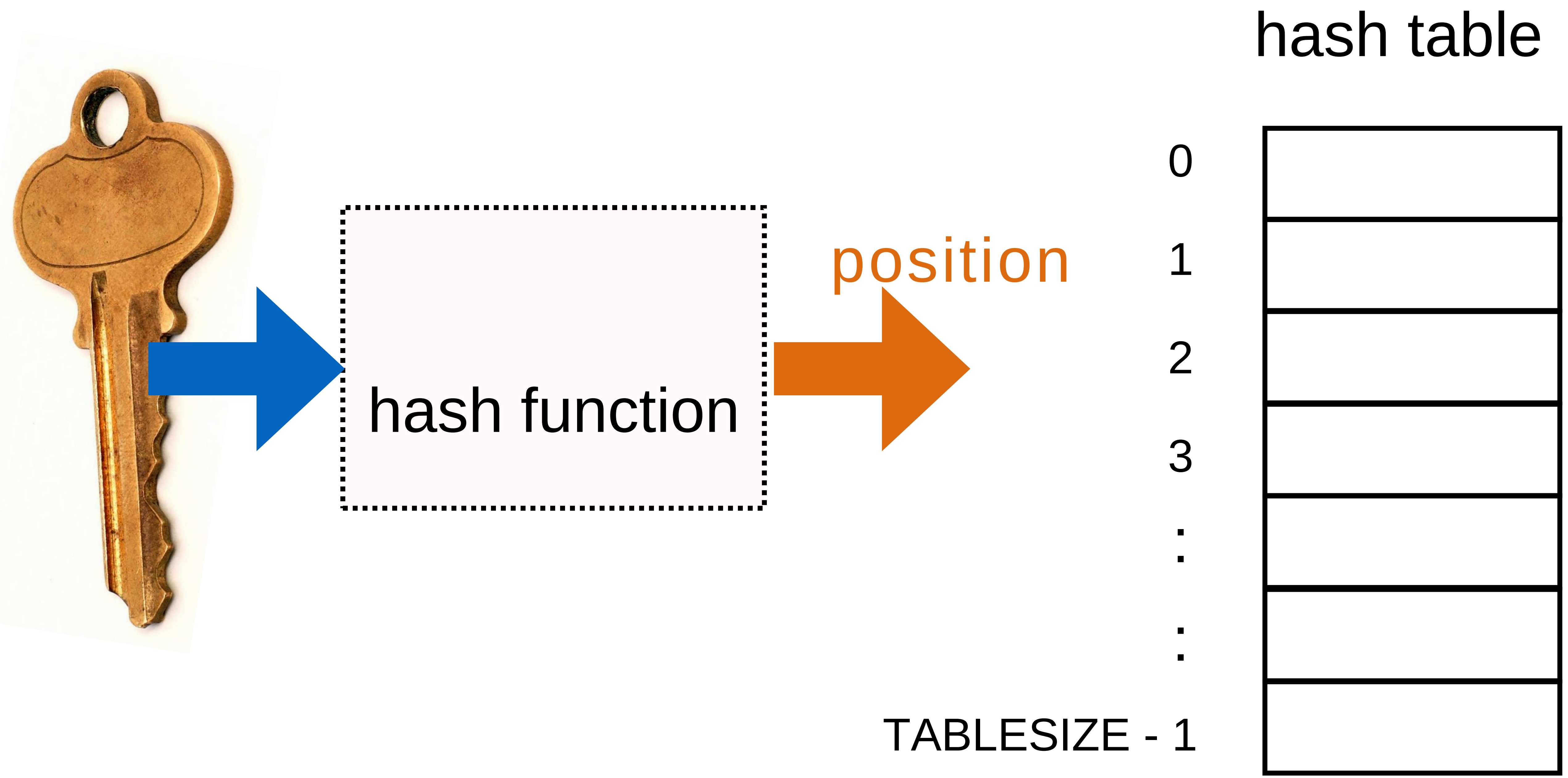
- Hash Tables promise:
  - Constant time operations (in most cases)
  - Worst case: still *O(N)*
- How?
  - <u>Using arrays</u>: constant time access to a given position
  - But this means, each item must have an assigned position

Hash Table Data Type

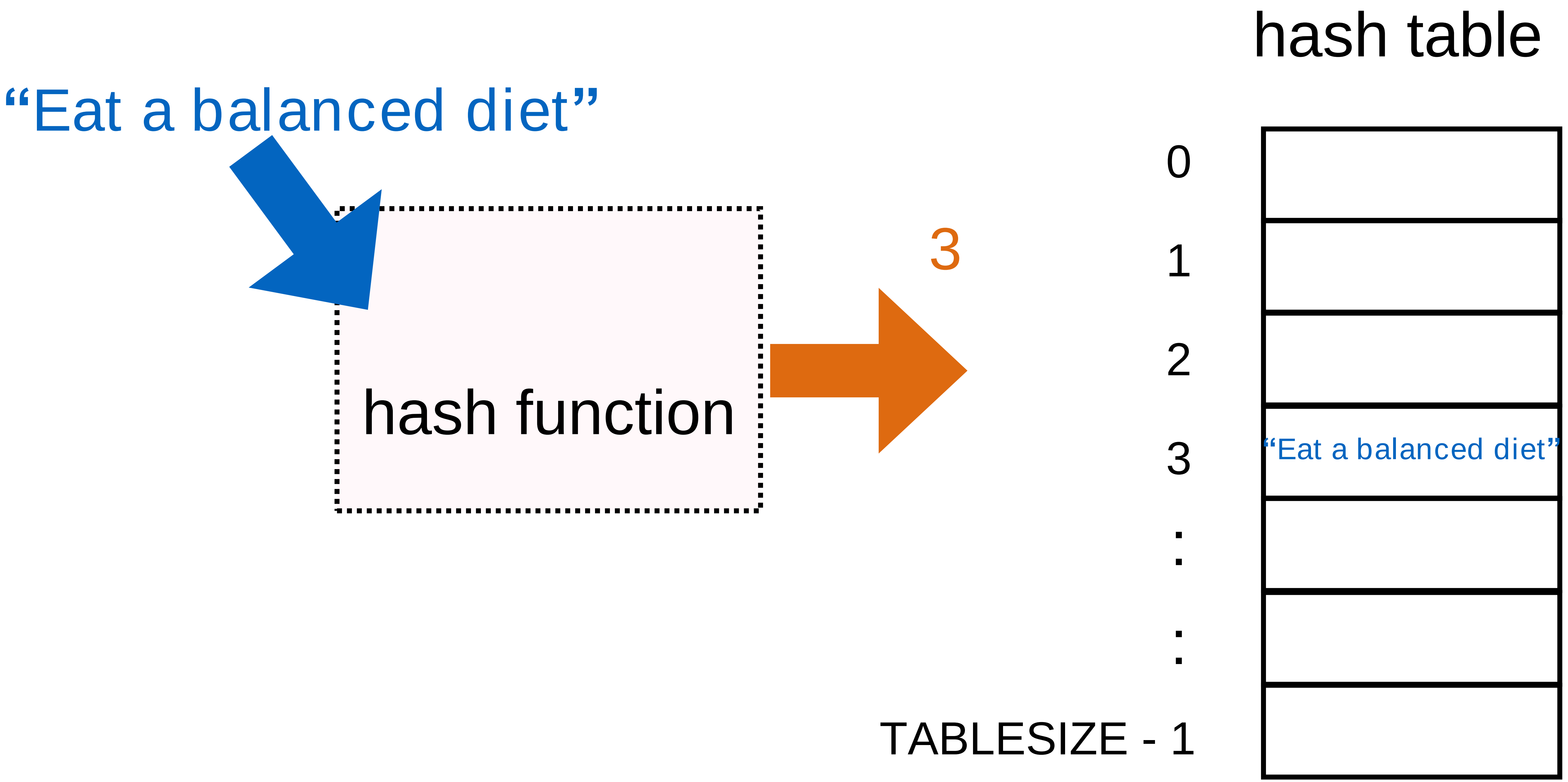
- Data :
  - Items to be stored
  - Each item must have a unique key
  - Underlying Data Structure: Large Array (also referred to as the Hash Table)

- Operations:
  - Insert
  - Search
  - Delete
  - Hash Function:
    maps a <u>unique key to an array position</u>

Overview



Example


hash table
“Eat a balanced diet”
hash function
3
0
1
2
3
“Eat a balanced diet”
:
:
TABLESIZE - 1

Hash Function’s properties

- Basic properties:
  - Type dependent: depends on the type of the item’s <u>key</u>
  - Return value within array’s range [0 .. TABLESIZE-1]
- Desirable:
  - Fast, a slow hash function will degrade performance
  - Minimises collisions (two keys mapped to same position)
- Perfect Hash maps every key into a different array position
  - Perfect hash functions are rare
  - Rely on very particular properties of the keys
- Good functions approximate random functions
- Chance of a collision is 1/TABLESIZE (Universal hash)

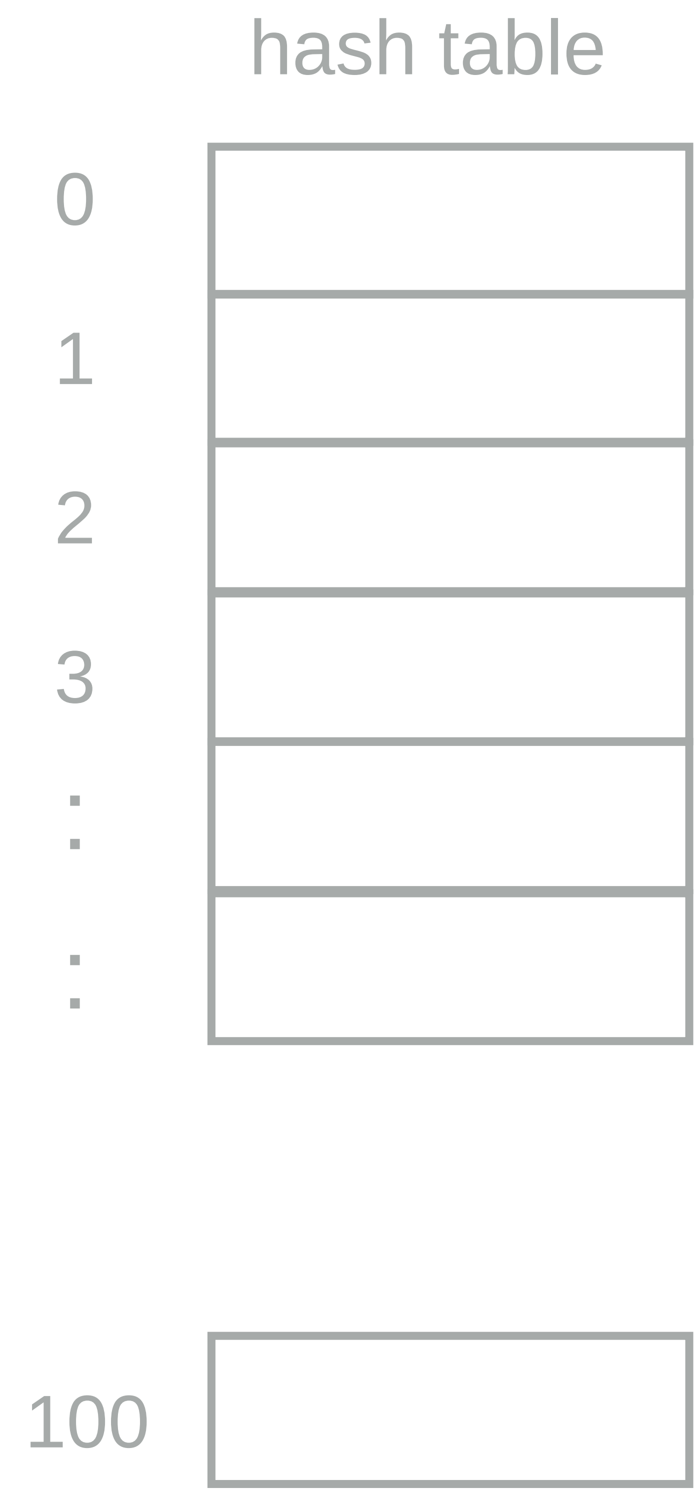
How to define Hash Functions?

If the key is an integer and random.

position = key % TABLESIZE

is random and fast

key	position
92258	45
2561	36
18243	63
55525	76
17271	0



How to define Hash Functions?

033-400-03-94-530

- 033: Supplier number (1..999, currently up to 70)
- 400: Category code (100,150,200, 250, up to 850)
- 03: Month of introduction (1..12)
- 94: Year of introduction (00 to 99)
- 530: Checksum (sum of all other fields mod 100)

Good hashing

- Don’t use non-data (no checksum)
- Modify the key until all bits count
  (category codes should be changed to 0..15)

How to define Hash Functions?

- Keys are words of up to ten letters
- Hash function:
  - Convert each character into a number (0..25)
  - Add the first two characters to obtain the array position
- Example:
  - shower$\rightarrow$ 18 + 7 = 25
  - often$\rightarrow$ 14 + 5 = 19
  - sleep $\rightarrow$ 18 + 11 = 29

Observations

- All words starting with the same two characters go to the same array position (collision)
- The more elements (characters, digits, etc) in the key you use, the better the hash function (in terms of collisions)
- Careful though: considering all might be too slow

How to define Hash Functions?

- Keys are words of up to ten letters
- Hash function:
  - Convert each character into a number (0..25)
  - Add all of them obtain the array position
- Example:
  - shower → 18 + 7 + 14 + 22 + 4 + 17 = 82
  - often → 14 + 5 + 19 + 4 + 13 = 55
  - sleep → 18 + 11 + 4 + 4 + 15 = 52

Observations

- Smallest positon: word a → 0 = 0
- Biggest: word zzzzzzzzzz → 10*25= 250
- But we have about 50,000 words in our dictionary!
- Many collisions: each array position would be the hash key for 200 words! Anagrams since position is disregarded
- A better hash function needs to take into account the position.

	2^3	2^2	2^1	2^0
	8	4	2	1
9	1	0	0	1
8	1	0	0	0
6	0	1	1	0
5	0	1	0	1

Words made of two characters… 1 and 0

How to define Hash Functions?

- Keys are words of up to ten letters
- Hash function:
  - Convert each character into a number (0..25)
  - Multiply each character by 26^i where i is the character position
  - Add them to obtain the position
- Example:
  - well → $22*26^3 + 4*26^2 + 11*26^1 + 11*26^0 = 389{,}673$
  - zzzzzzzzzz is greater than $26^9 > 5{,}000{,}000{,}000{,}000$

Observations

- Good discrimination: unique position per word
- Might exceed the capability of our table (or overflow our index)
- Too big for our 50,000 words: lots of empty positions
- We want something in the range of our TABLESIZE
- If the resulting number is too big: use % TABLESIZE (beware overflow in certain languages)

array position

Ex. Where will it live…

character code

Ex. which letter

character position

Ex. where in the string

$$h = a_0 x^n + \cdots + a_{n-3}x^3 + a_{n-2}x^2 + a_{n-1}x^1 + a_n$$

base (e.g., 26)

$$h = ((\ldots(a_0 x + a_1)x + \cdots + a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$$

At each step we take mod

$$h = ((\ldots(a_0 x + a_1)x + \cdots + a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$$

table_size = 101

```
def hash_function(word):
    value = 0
    for i in range(len(word)):
        value = (value*31 + ord(word[i])) % 101
    return value
```

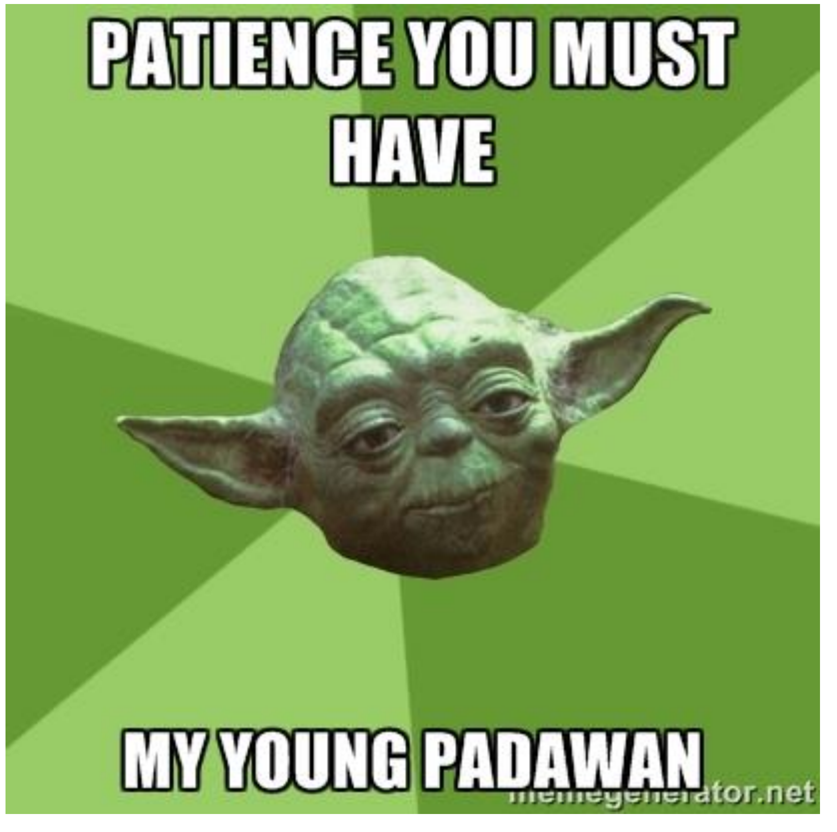
base = 31

```
ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

```
value = (value*31 + ord(word[i])) % 101
```

31? But I thought I had 26 letters?!



How to define Hash Functions?

Consider the word “Aho”

```
              value = 0
'A' = 65      value = (31 * 0 + 65) % 101 = 65
'h' = 104     value = (31 * 65 + 104) % 101 = 99
'o' = 111     value = (31 * 99 + 111) % 101 = 49
```

49

$65*(31^2) + 104*(31^1) + 111 = 65800$

$65800 \bmod 101 = 49$

How to define Hash Functions?

- If the key an integer and is randomly distributed then position = key % TABLESIZE is random and fast.

- Use a prime table size: If many values and TABLESIZE share common factors they will hash to the same position.

- Example: TABLESIZE=10 and all our keys finish in 0.
  Then all keys are hashed to 0.

- If you are multiplying by a constant and taking modulo, it helps if the value and the constant have no common factors.

  - Observation: 26 is not prime, but 31 is.

Prime table size and prime base will lead to spread out values…

Example

```
value = (value*31 + ord(word[i])) % 101
```

Key	Hash value
Aho	49
Kruse	95
Standish	60
Horowitz	28
Langsam	21
Sedgewick	24
Knuth	44

This results in a sparse Table because 31 and 101 are primes

Example

```
value = (value*1024 + ord(word[i])) % 128
```

Key	Hash value
Aho	111
Kruse	101
Standish	104
Horowitz	122
Langsam	109
Sedgewick	107
Knuth	104

Things end up close to each other… and we also get collisions…

"clustering"

Example

```
value = (value*3 + ord(word[i])) % 7
```

Key	Hash value
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

Reasonable size… too small a table.

How to define Hash Functions?

- Even more effective than selecting a single coefficient, like 31…
  - ➞Choose your coefficients (pseudo) randomly
  - ➞Use a different coefficient for each position (in a predictable manner).

```python
def hash_function(word, TABLE_SIZE):
    value = 0
    a = 31415
    b = 27183
    for i in range(len(word)):
        value = (a*value + ord(word[i])) % TABLE_SIZE
        a = a * b % (TABLE_SIZE-1)
    return value
```

Universal

Hash Functions properties (recap)

- Type dependent
- Must return value within array's range
- Fast: not too many arithmetic operations. Still linear in the size of key.
- Minimise collisions (each position equally likely)
  - Don't use non-data
  - Use all elements (or a reasonable subset – odd/even positions)
  - Use the position of each element
  - Avoid common factors
- And of course, it must be a function! Same value, same input

Hash Table operations: Insert

- Apply the hash function to get a position N
- Try to insert key at position N
- Deal with collision if any

Example

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth

What to do?

Horowiz → hash function → 5

hash table

0	Aho
1	Standish
2	
3	
4	
5	Kruse
6	

http://cdn.moneycrashers.com/wp-content/uploads/2010/08/car-collision-men.jpg

Collisions: two main approaches

- Separate chaining:
  - Each array position contains a linked list of items
  - Upon collision, the element is added to the linked list

- Open addressing:
  - Each array position contains a single item
  - Upon collision, use an empty space to store the item (which empty space depends on which technique)

Open Addressing: Linear Probing

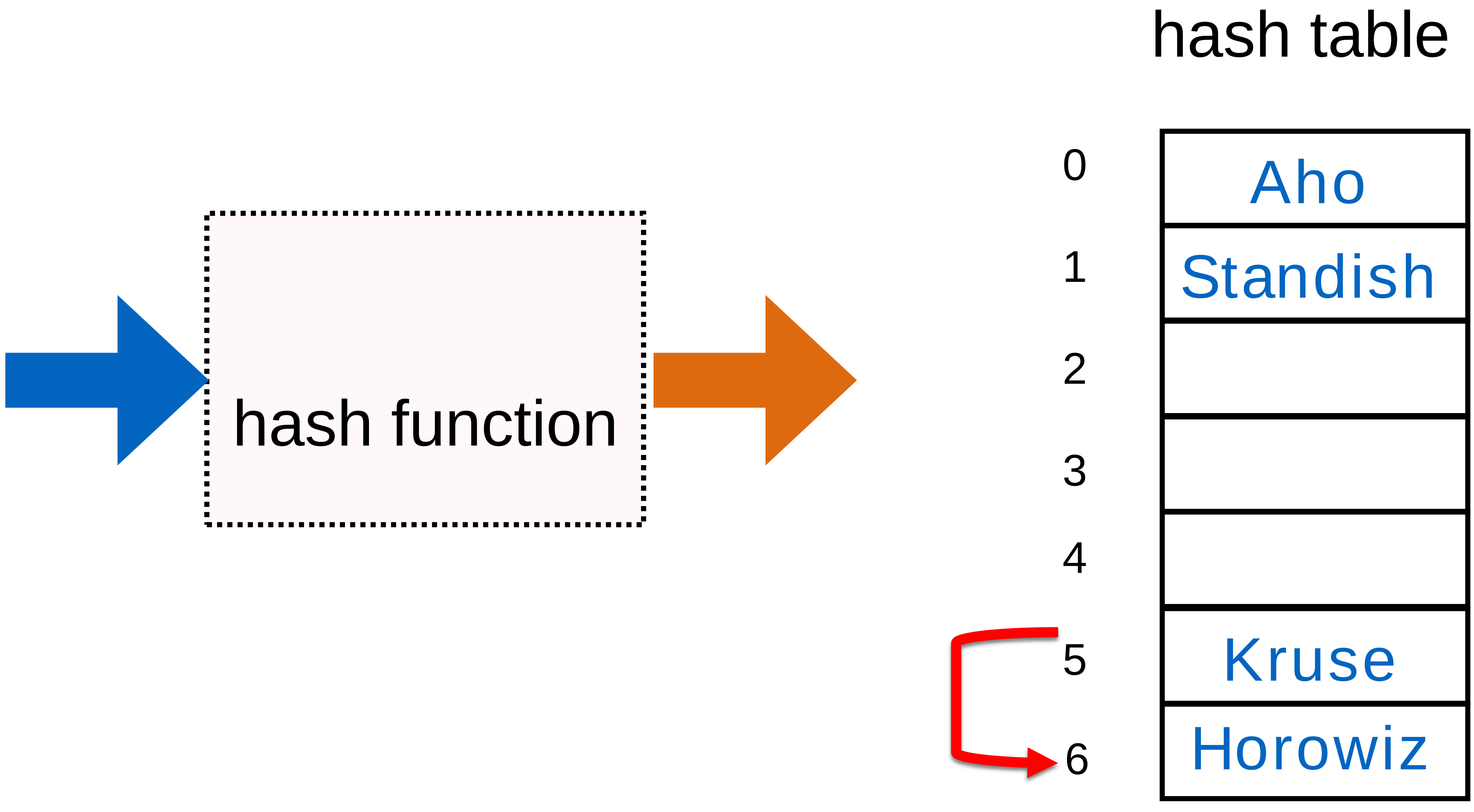
- Insert item with hash value N:
  - If array[N] is empty just put item there.
  - If there is <u>already an item there</u>:
    look for the first empty space in the array from N+1 (if any) and add it there

- Linear search from N until an empty slot is found

- Things to think about:
  - Full table (to avoid going into an infinite loop)
  - Restarting from position 0 if the end of table is reached
  - Finding an item with the same key.

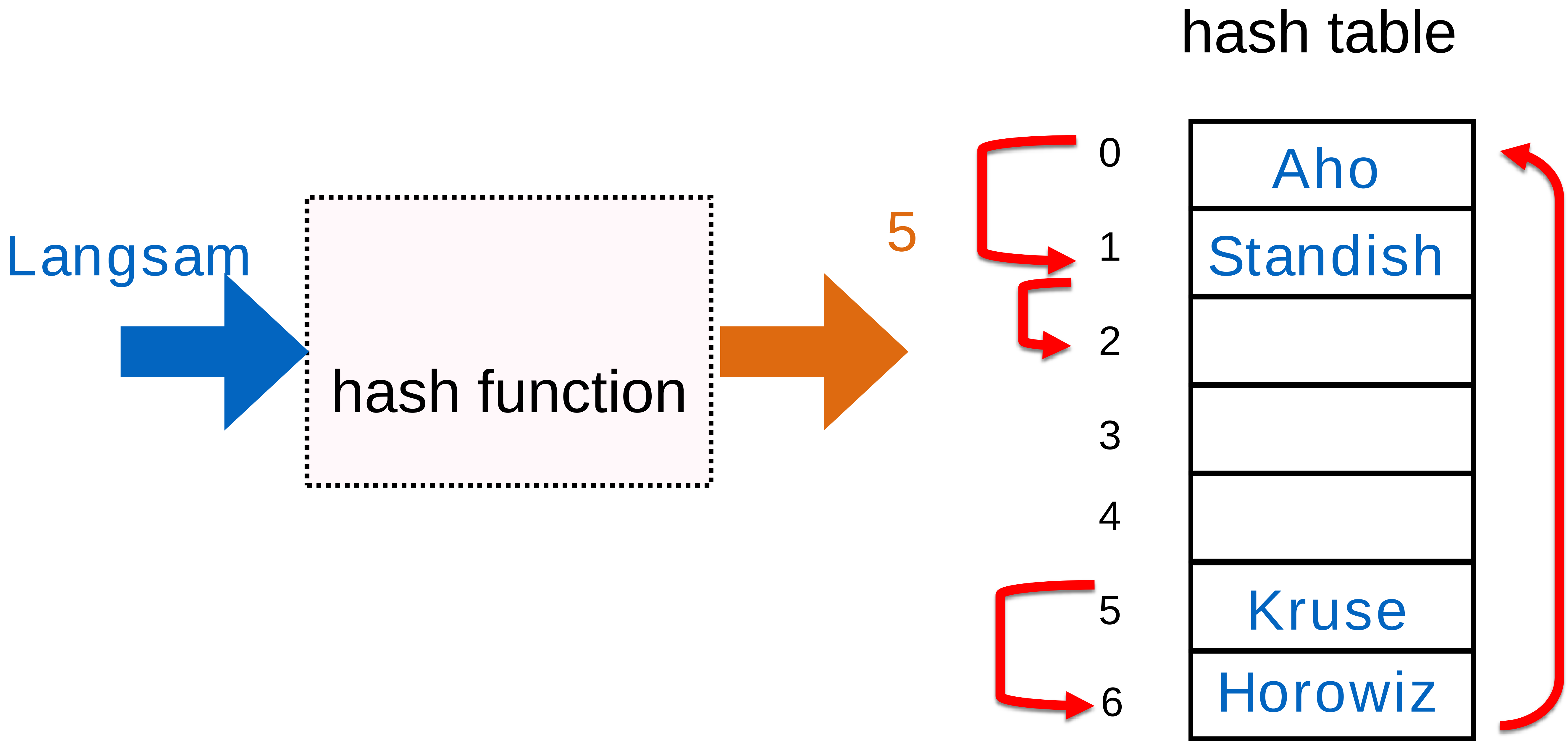
Example

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



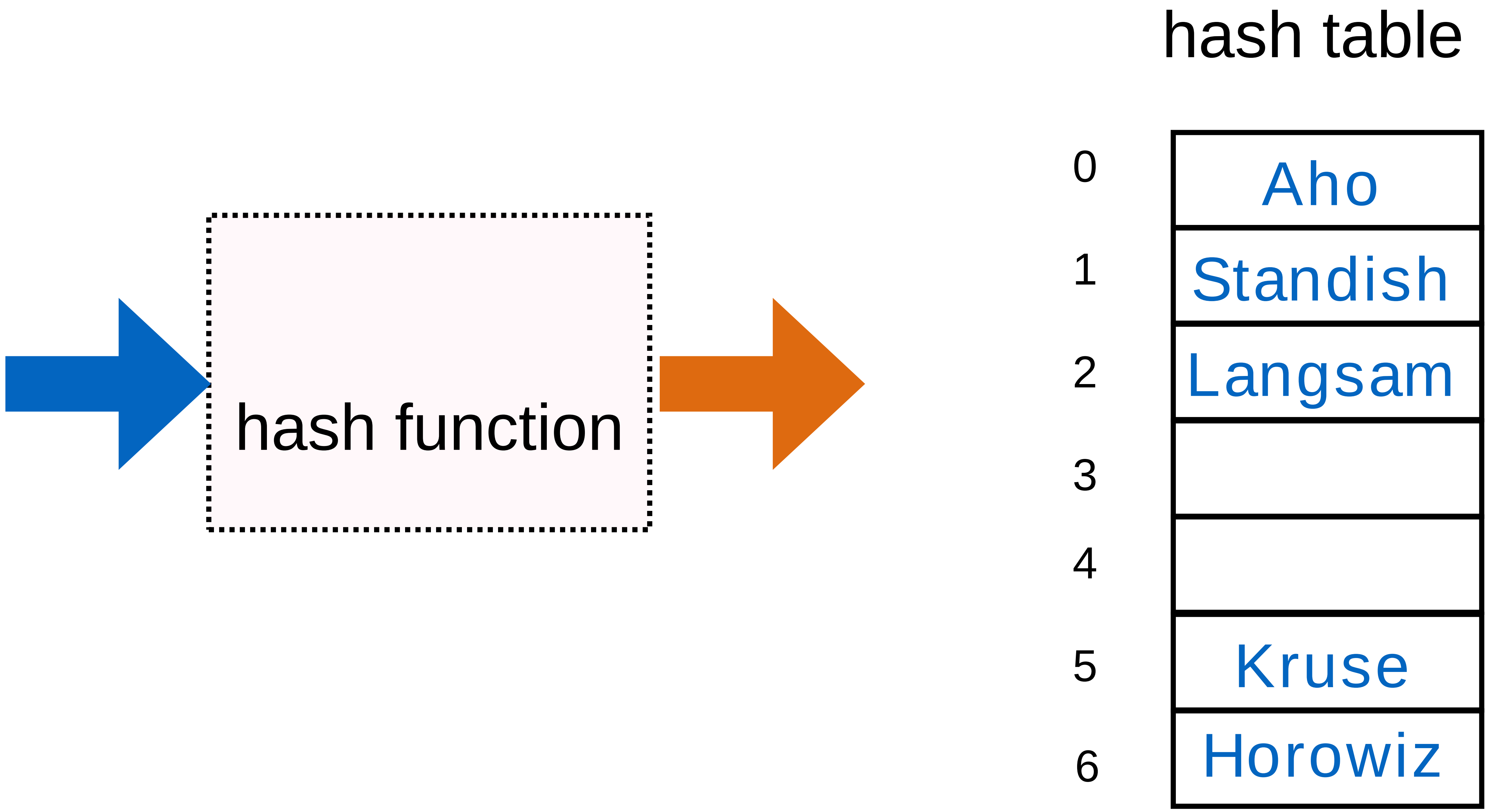
Example

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



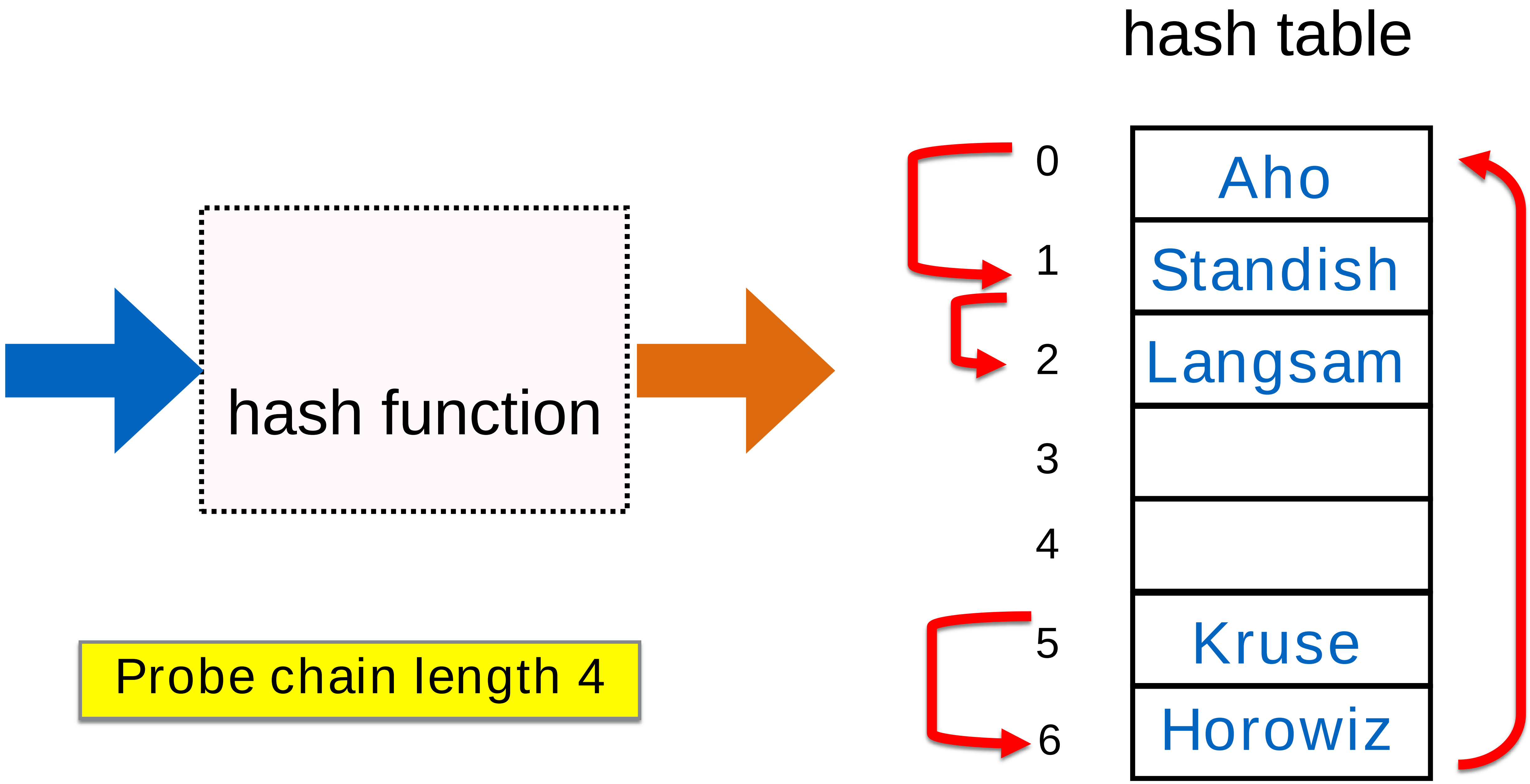
Example

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



Example

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



Probe chain length 4

Example

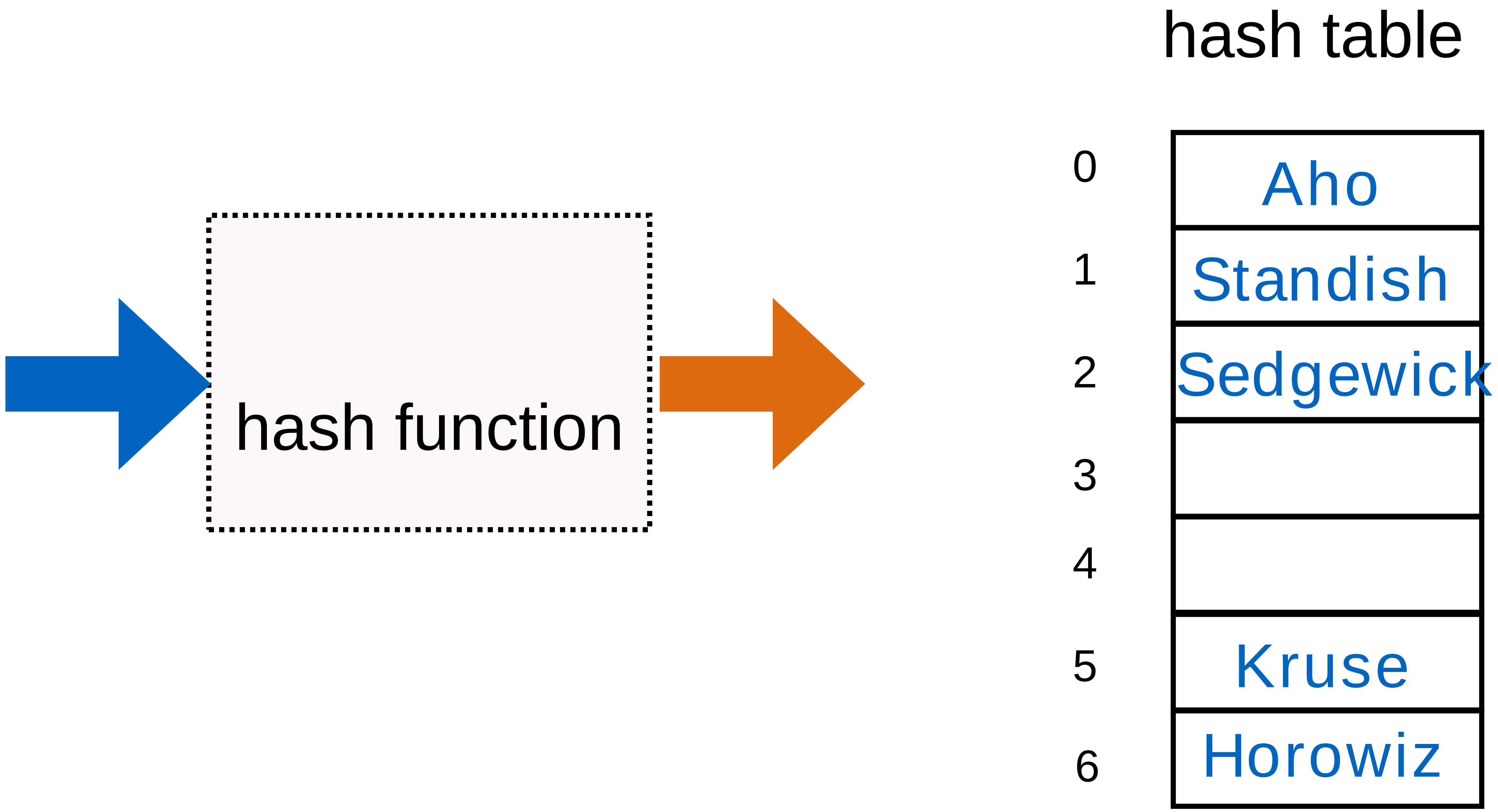
Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth

hash table

0	Aho
1	Standish
2	
3	
4	
5	Kruse
6	Horowiz

Example

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



Summary

- What is a hash table data type and why is it needed
- Hash Functions
  - Definition
  - Properties
  - How to define them
- Perfect hash functions
- Universal hash functions
- Resolving Collisions