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

Lecture 23
Hash Tables

FIT 1008
Introduction to Computer Science

COMMONWEALTH OF AUSTRALIA
Copyright Regulations 1969
WARNING
This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act).The material in this communication may be subject to copyright under the Act.Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.
Do not remove this notice.

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**

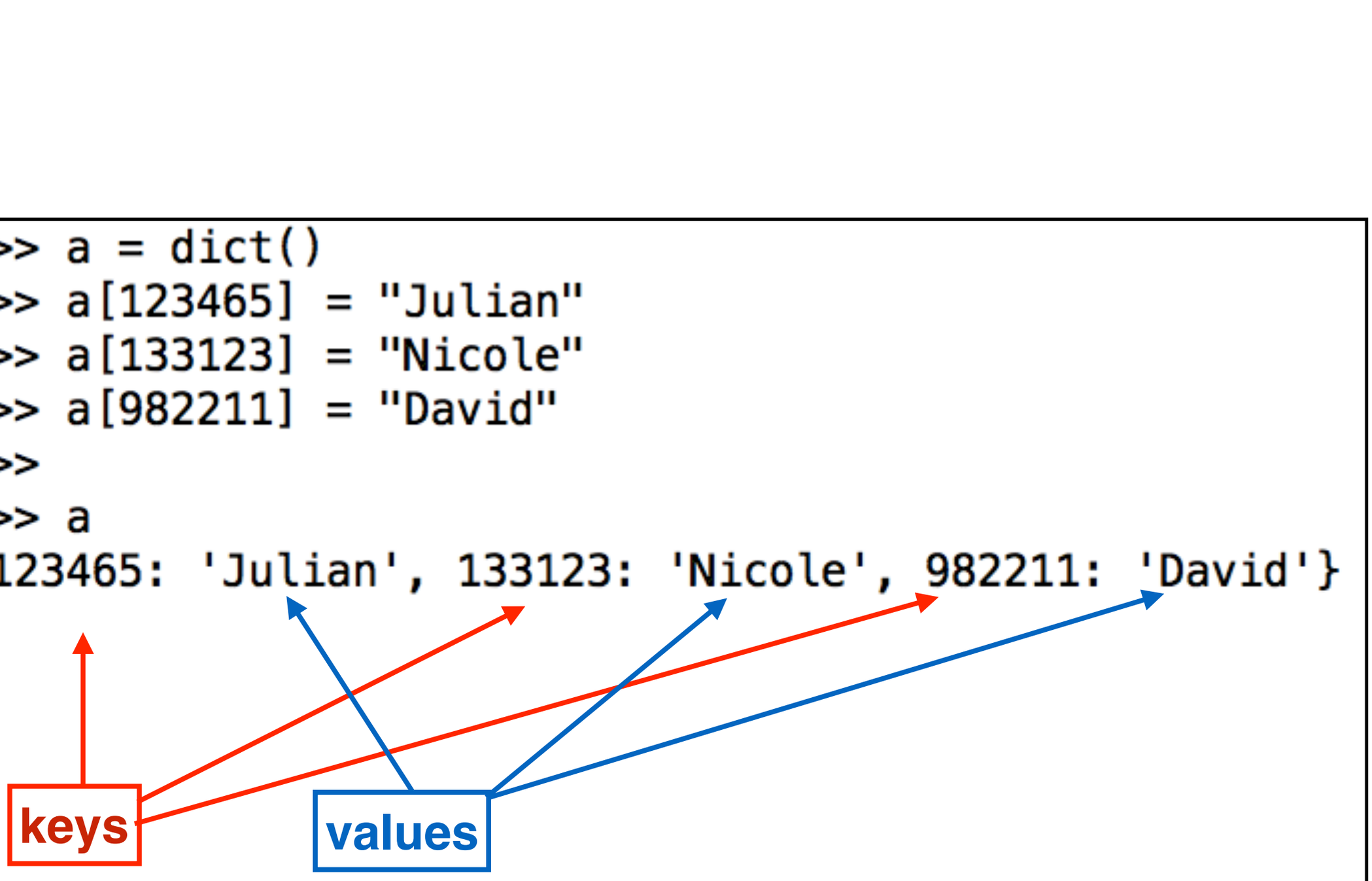
- To understand the challenges posed by collisions and start looking at solutions

Dictionary ADT

- Permits access to data items by content, e.g., a key.

- Operations:
  - ➡ Search
  - ➡ Insert
  - ➡ Deleta

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



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

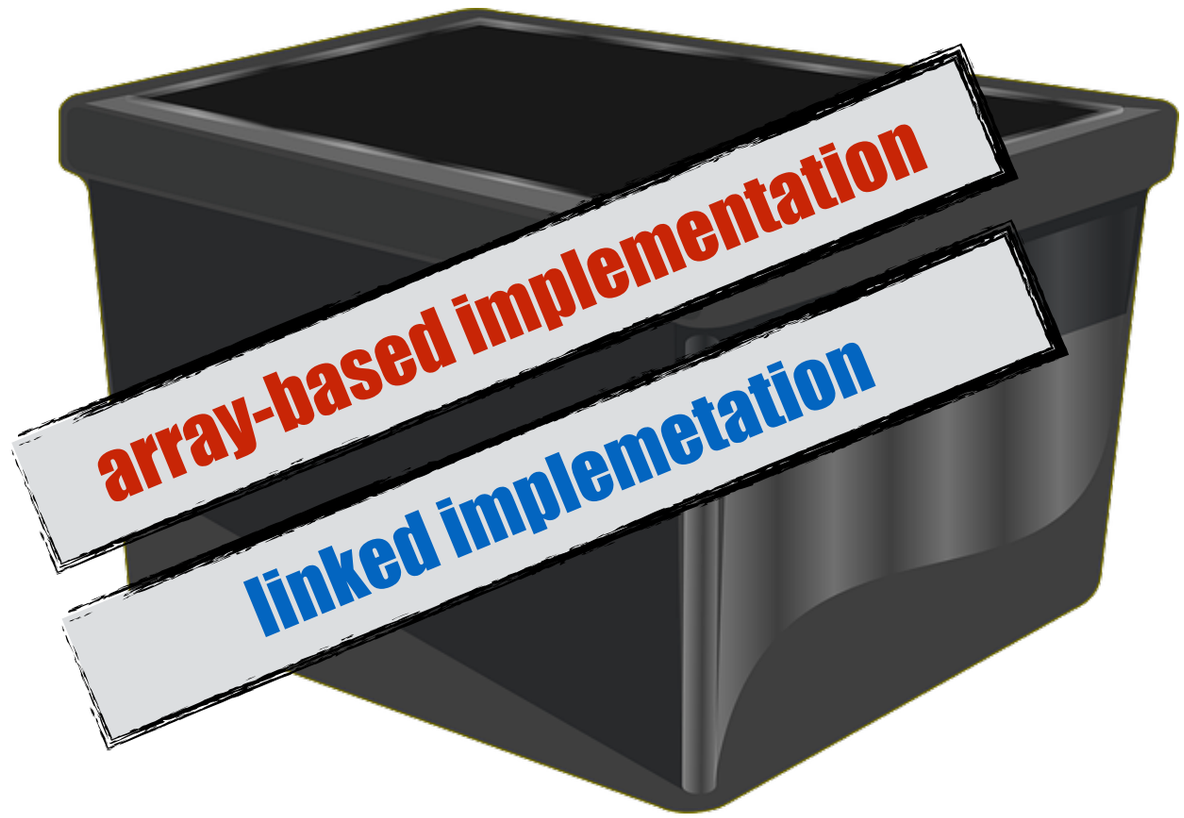
- Assume we are interested in **storing** a very significant amount of data (a big N)
- Assume we are going to need to perform the following operations relatively often:
  - **Search** for an item
  - **Insert** a new item
  - You might also want to delete an item (optional)
- But we do **not** need to traverse them in a **particular order** or sort them (at least not often)

Container ADTs

- **Stores** and removes items **independent of contents.**

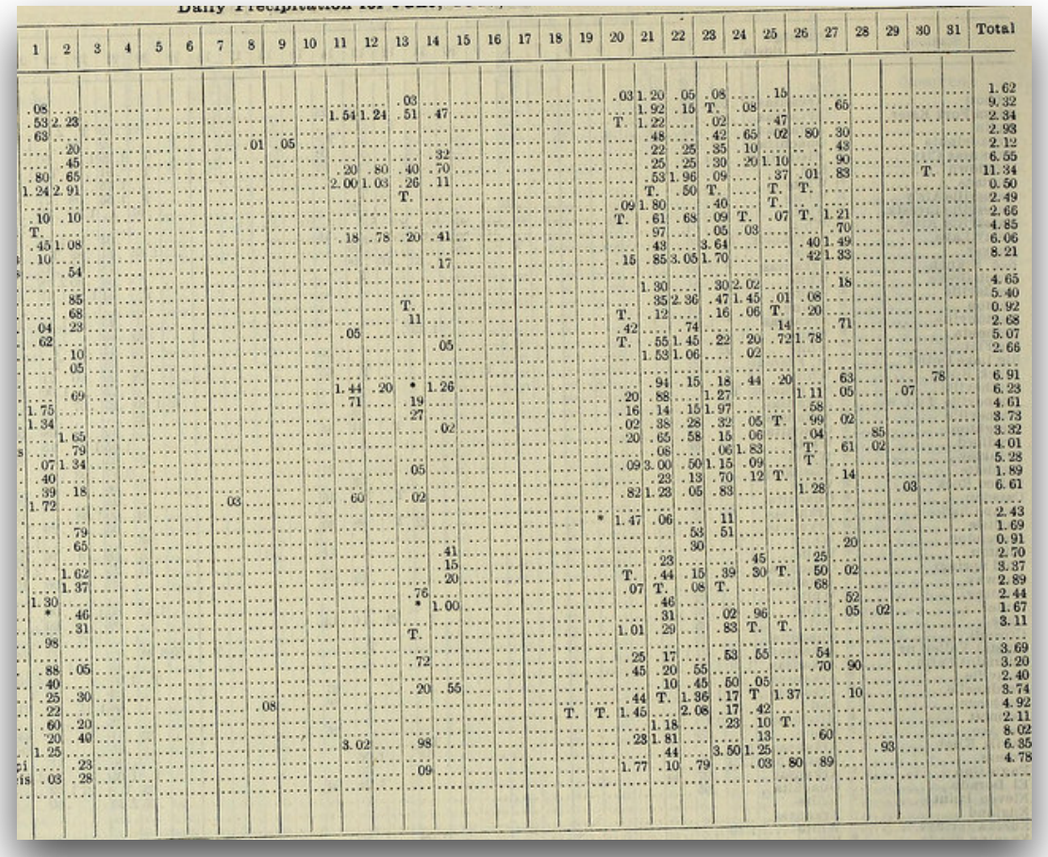
- **Examples** include:
  - List ADT ☑
  - Stack ADT ☑
  - Queue ADT. ☑

- Core **operations**:
  - ➡ add item
  - ➡ delete item
  - ➡ search



- **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)

Hash Tables: aim



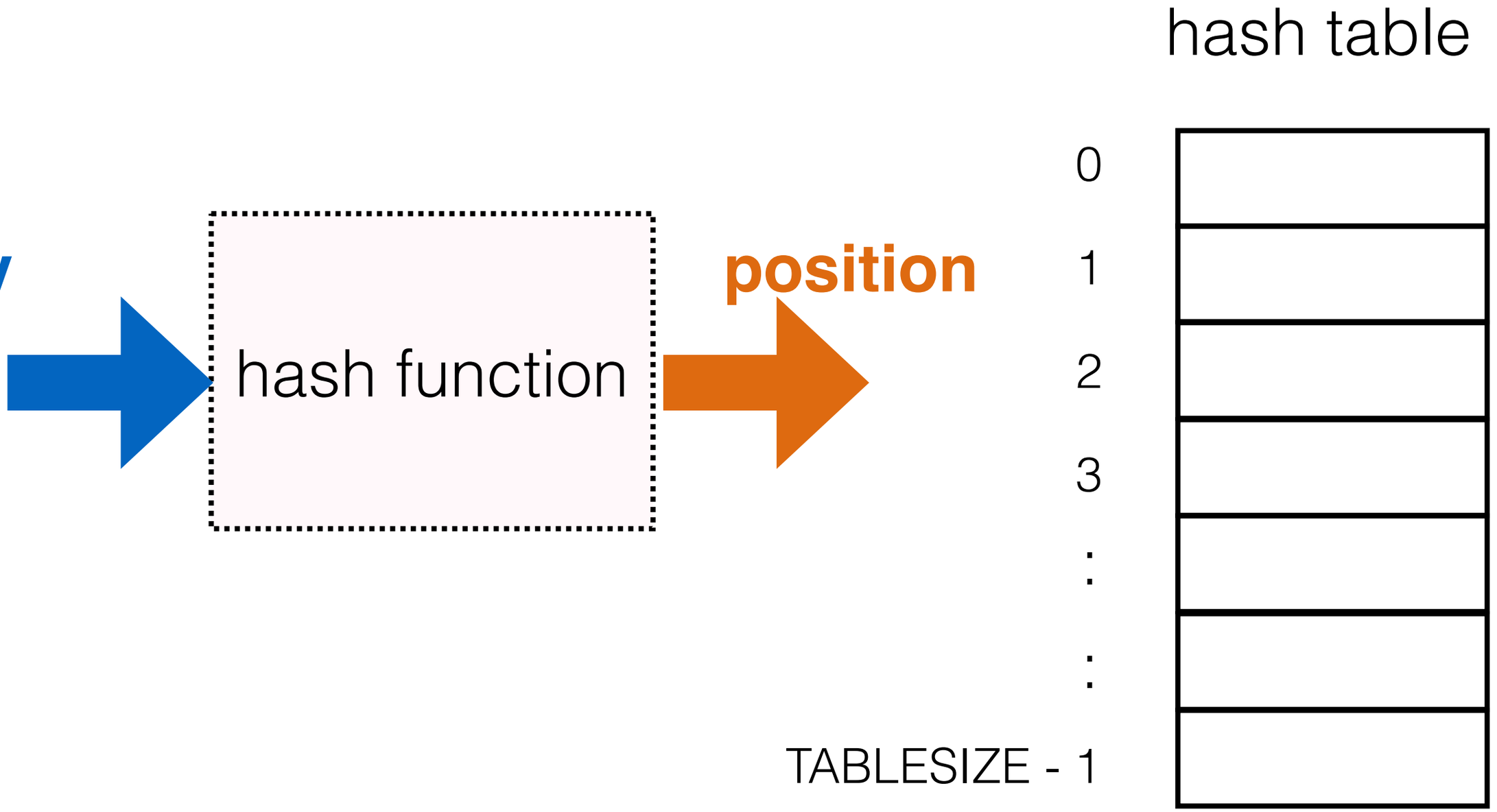
- Hash Tables promise:
  - **Constant time** operations (in most cases)
  - Worst case: still *O(N)*
- How?
  - Using **arrays**: constant time access to a given position
  - But this means, each item must have an **assigned position**

Hash Table Data Type

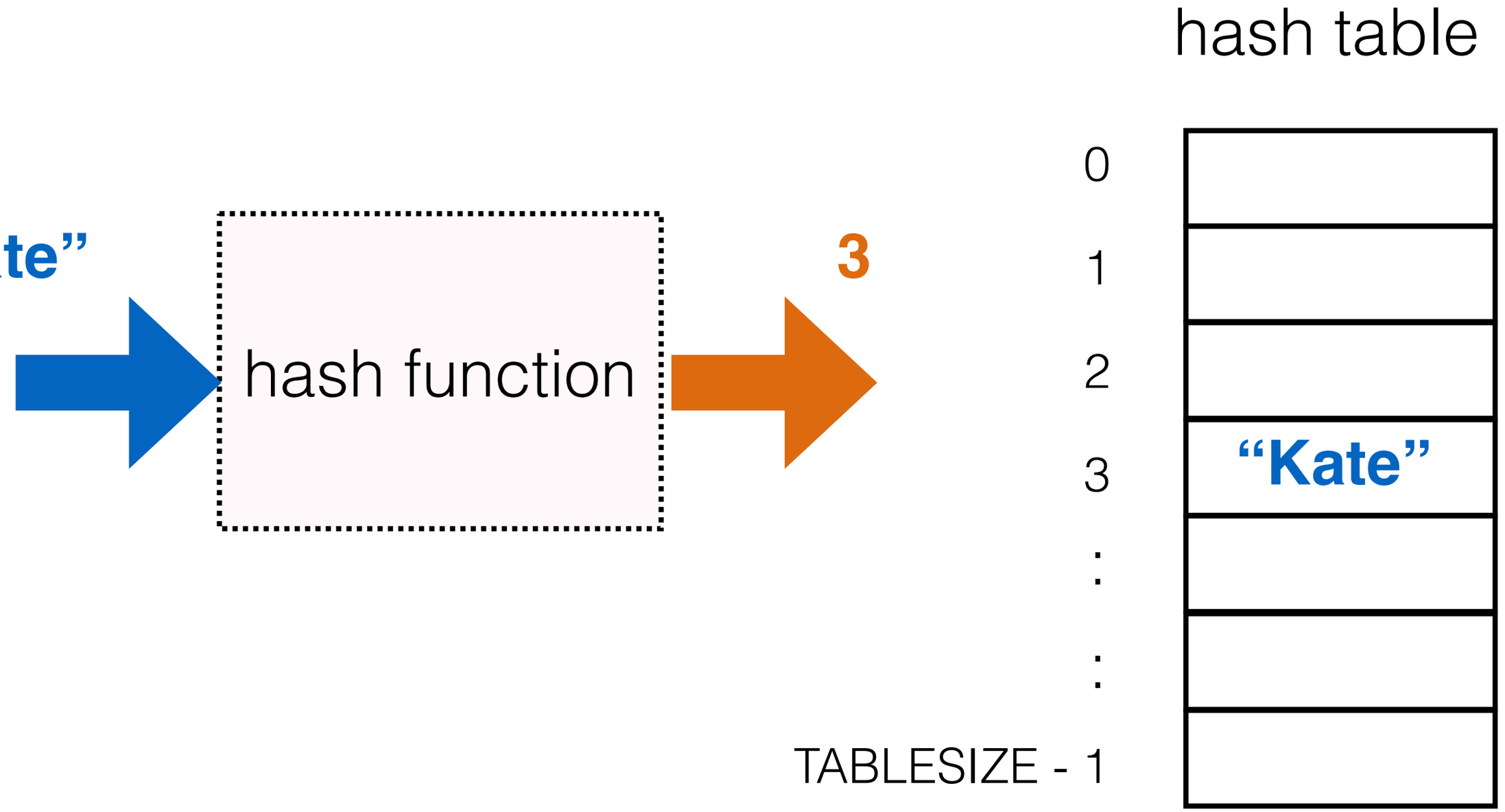
- **Data** :
  - Items to be stored
  - Each item must have a **unique key**
  - Underlying Data Structure: Large Array (also referred to as the Hash Table)

- **Operations**:
  - Insert
  - Search
  - Delete
  - **Hash Function**:
    maps a unique key to an array position

Overview



Example



Example

Hash Function's properties

- **Basic properties**:
  - Type dependent: depends on the type of the item's key
  - 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**)

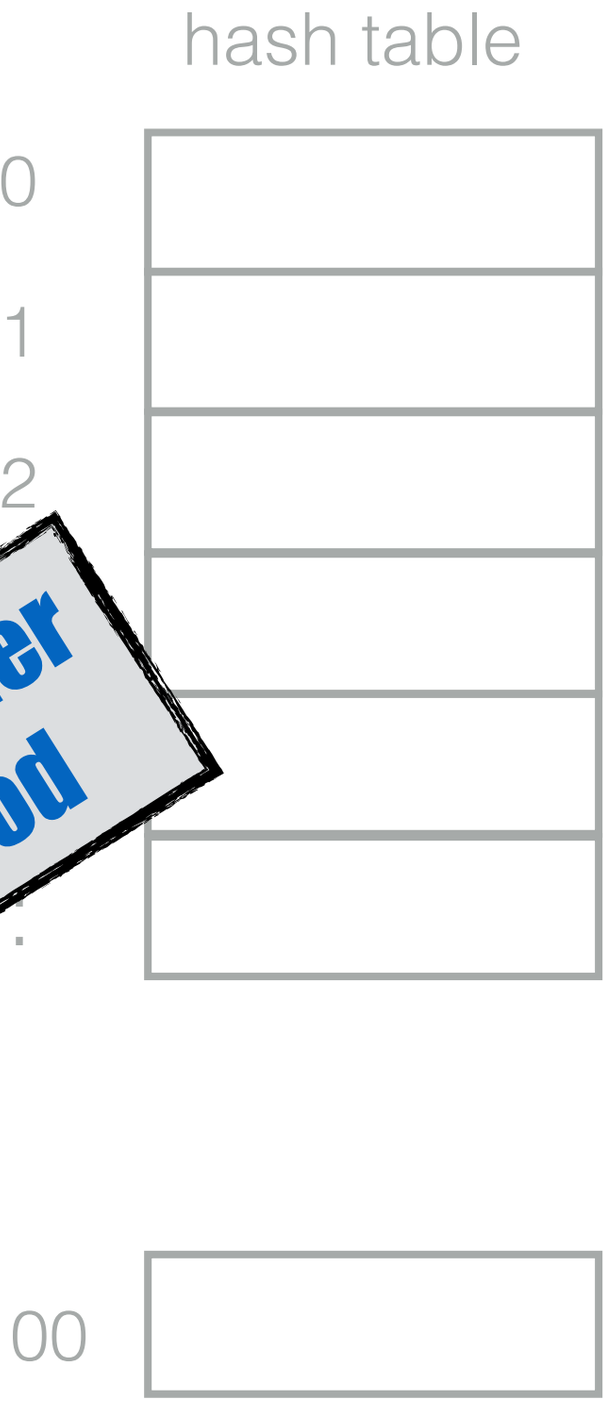
How to define Hash Functions?

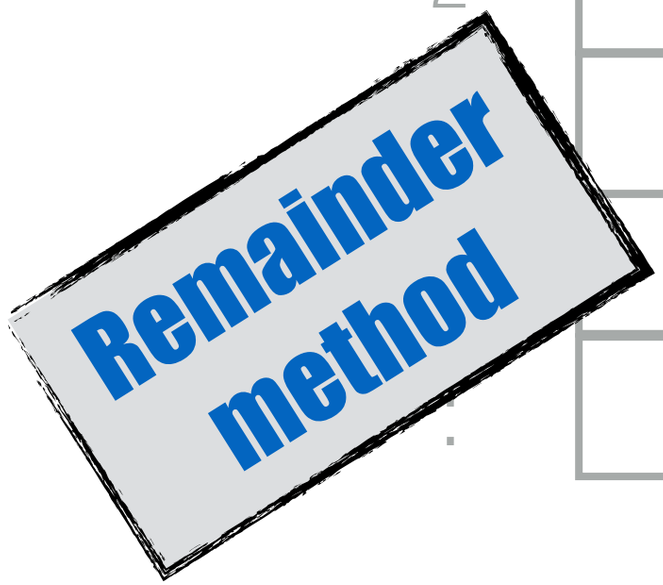
If the **key is an integer** and random.

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 practices for hashing

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

What if keys are strings?

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**:
  - maria → 12 + 0 = 12
  - bernd → 1 + 4 = 5
  - malena → 12 + 0 = 12

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**:
  - maria → 12 + 0 + 17 + 8 + 0 = 37
  - bernd → 1 + 4 + 17 + 13 + 3 = 38
  - malena → 12 + 0 + 11 + 4 + 13 + 0 = 40

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.

Idea: Use all characters and take into account the position.

(Have we done something like this before?)

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**:
  - maria → $12*26^4 + 0*26^3 + 17*26^2 + 8*26^1 + 0*26^0 = 5'495.412$
  - zzzzzzzzzz is greater than $26^9 > 5{,}000{,}000{,}000$

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**

array position

character code

character position

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

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

$$h = ((\ldots(a_0x + a_1)x + \cdots + a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$$

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

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.

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.

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

$$h = ((\ldots(a_0x + a_1)x + \cdots + a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$$

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

The choice of ***a, h*** and ***table_size*** affects the performance of the hash.

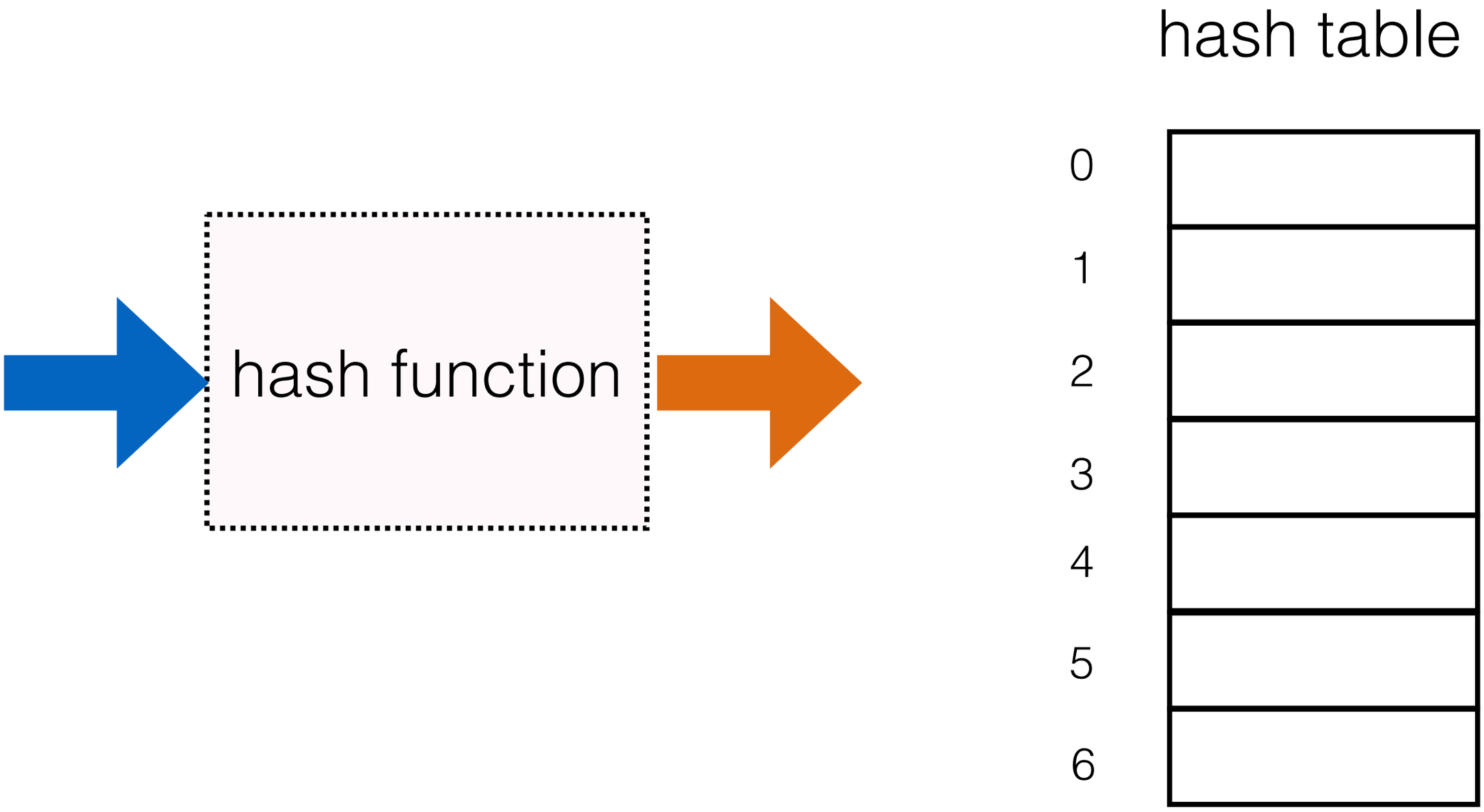
(Often empirically chosen based on data)

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

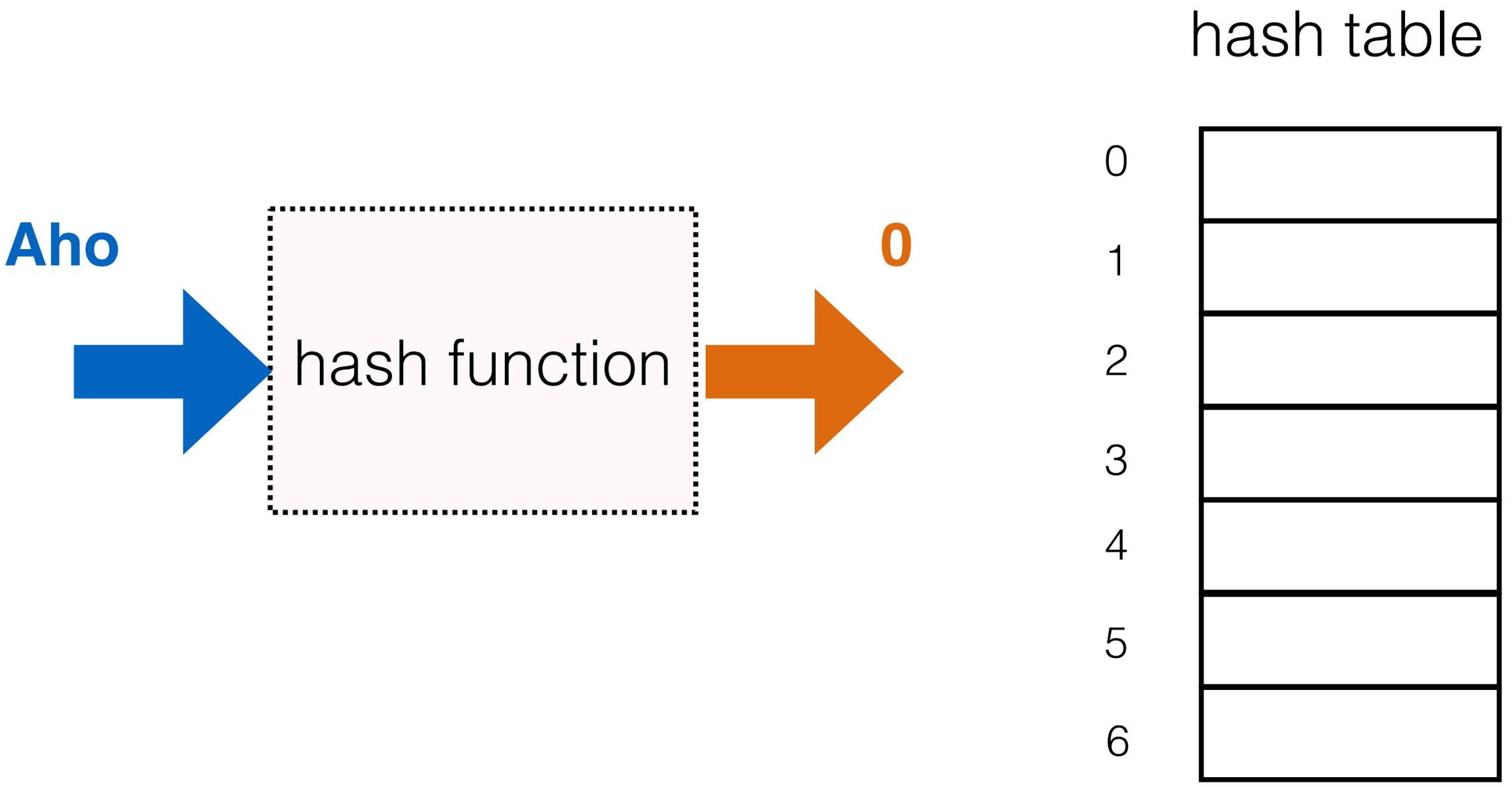
Example

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



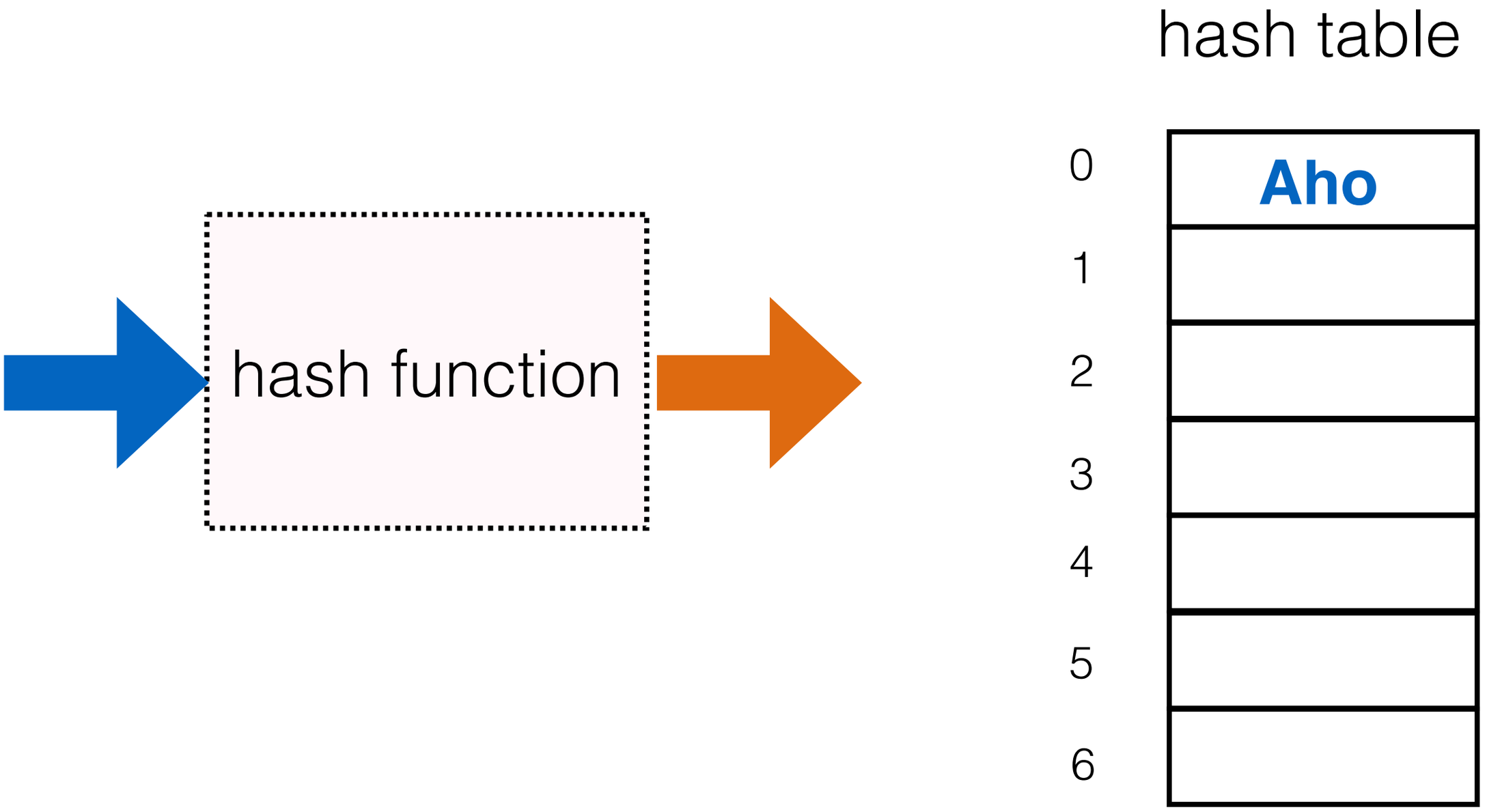
Example

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



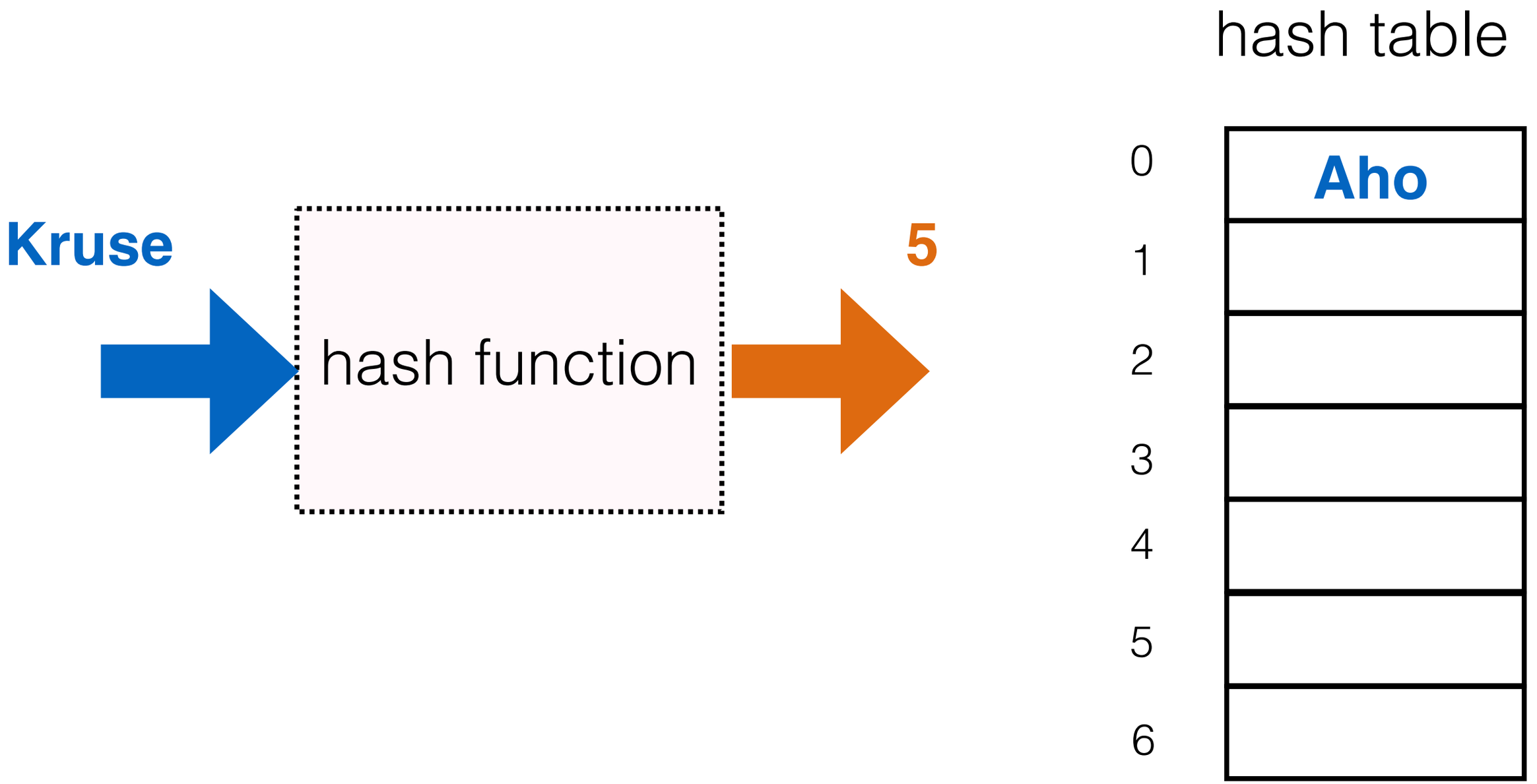
Example

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



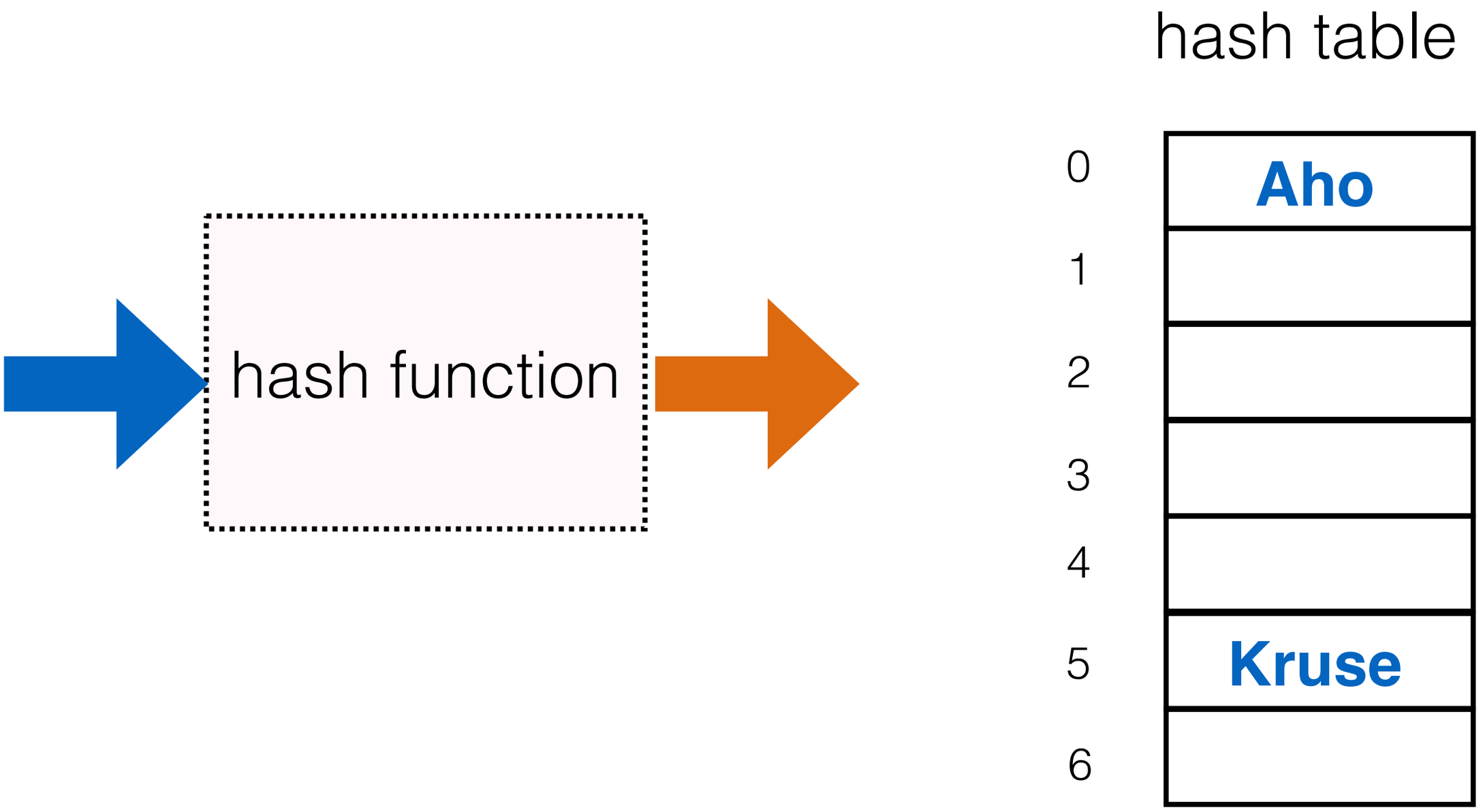
Example

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



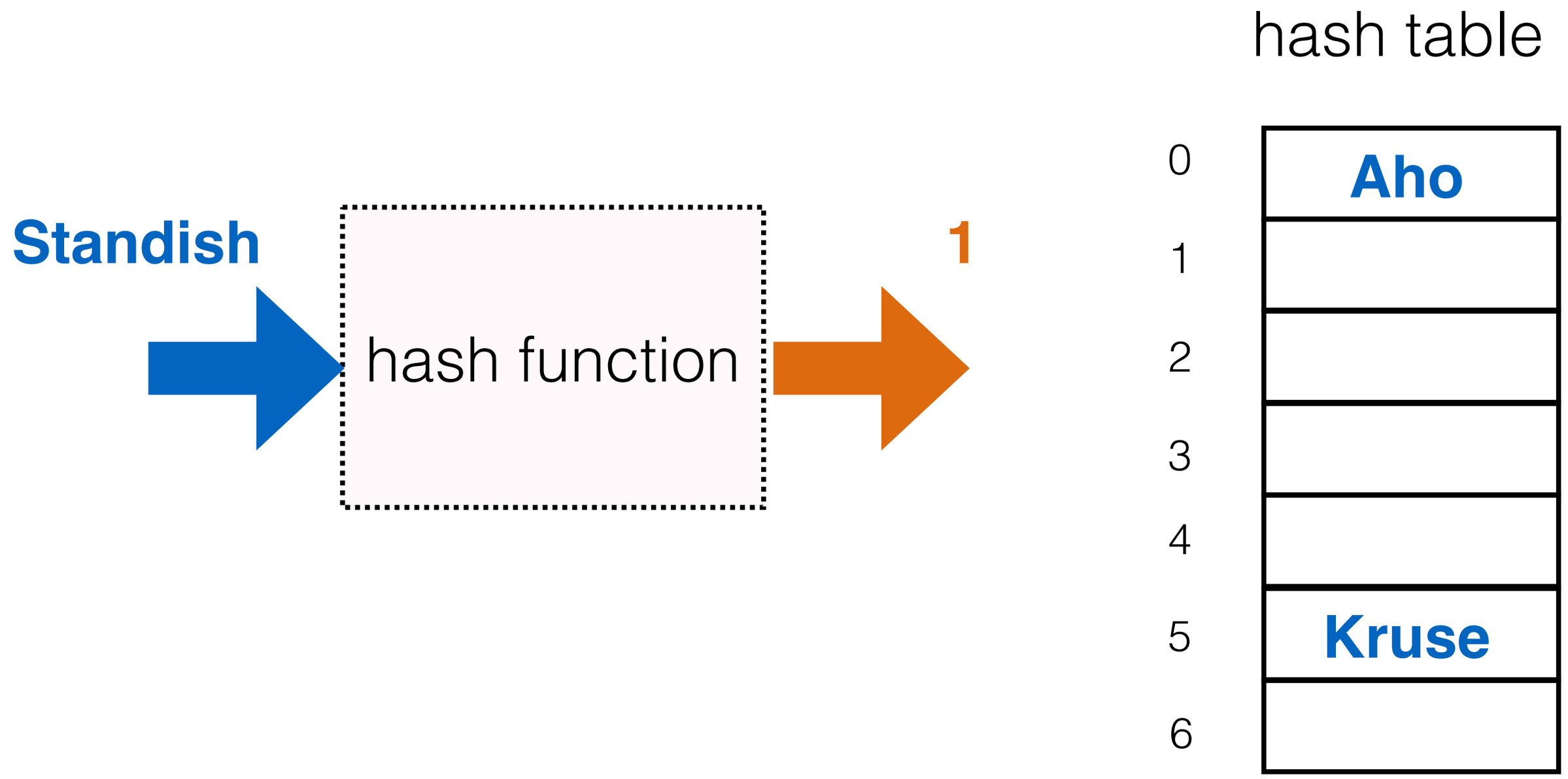
Example

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



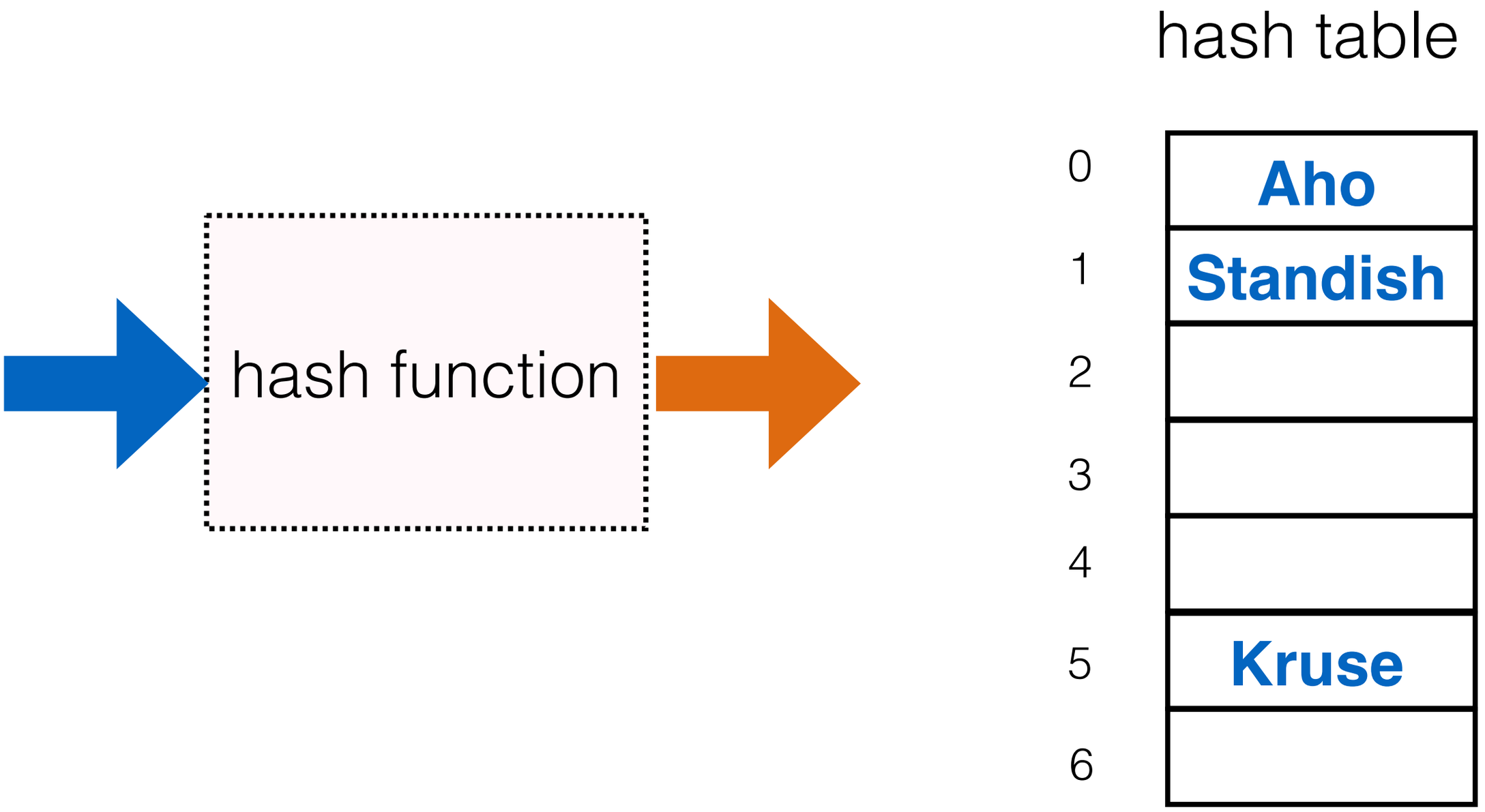
Example

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



Example

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

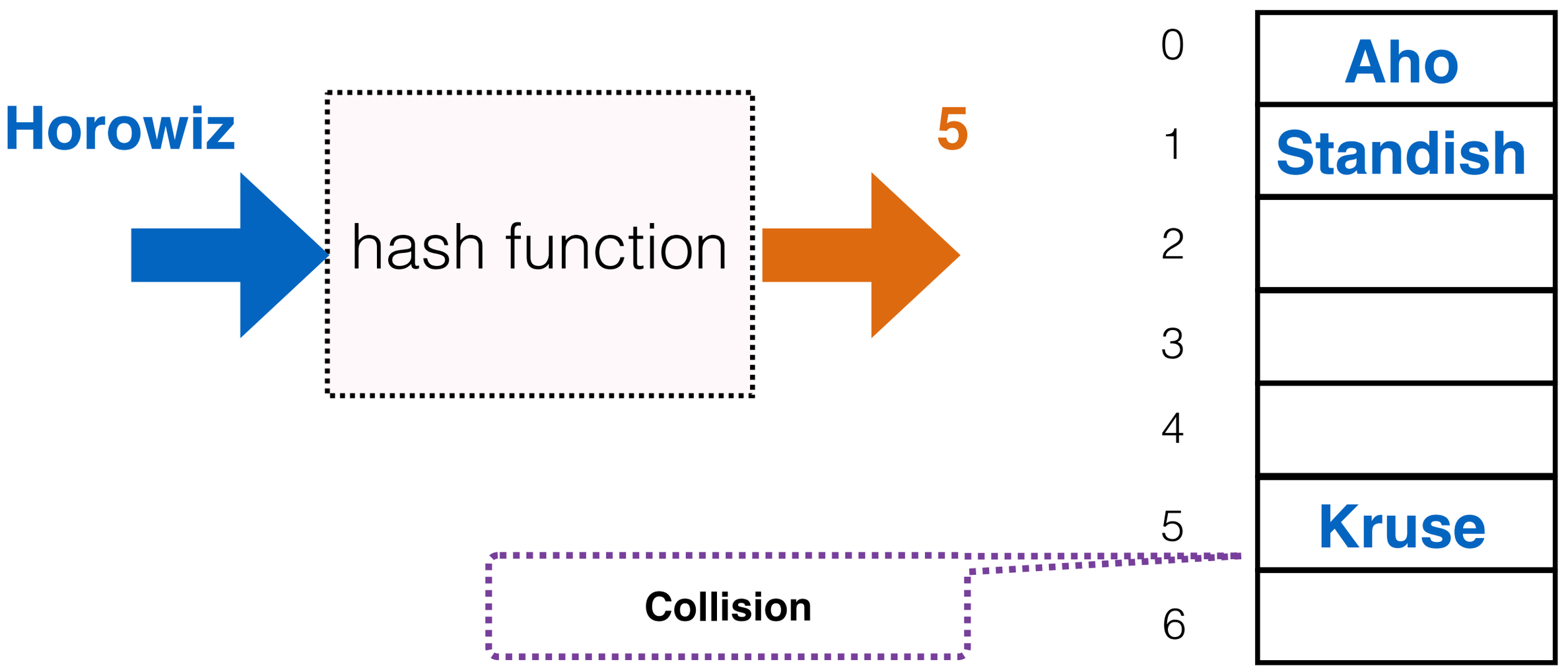


Example

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

What to do?

hash table



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)

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