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

Lecture 30
Collision Resolution

FIT 1008
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

Collisions: two main approaches

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

Open Addressing

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.

Insert the following keys into the Hash Table, in the order they appear, using linear probing. Is the following table correct?

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

hash table

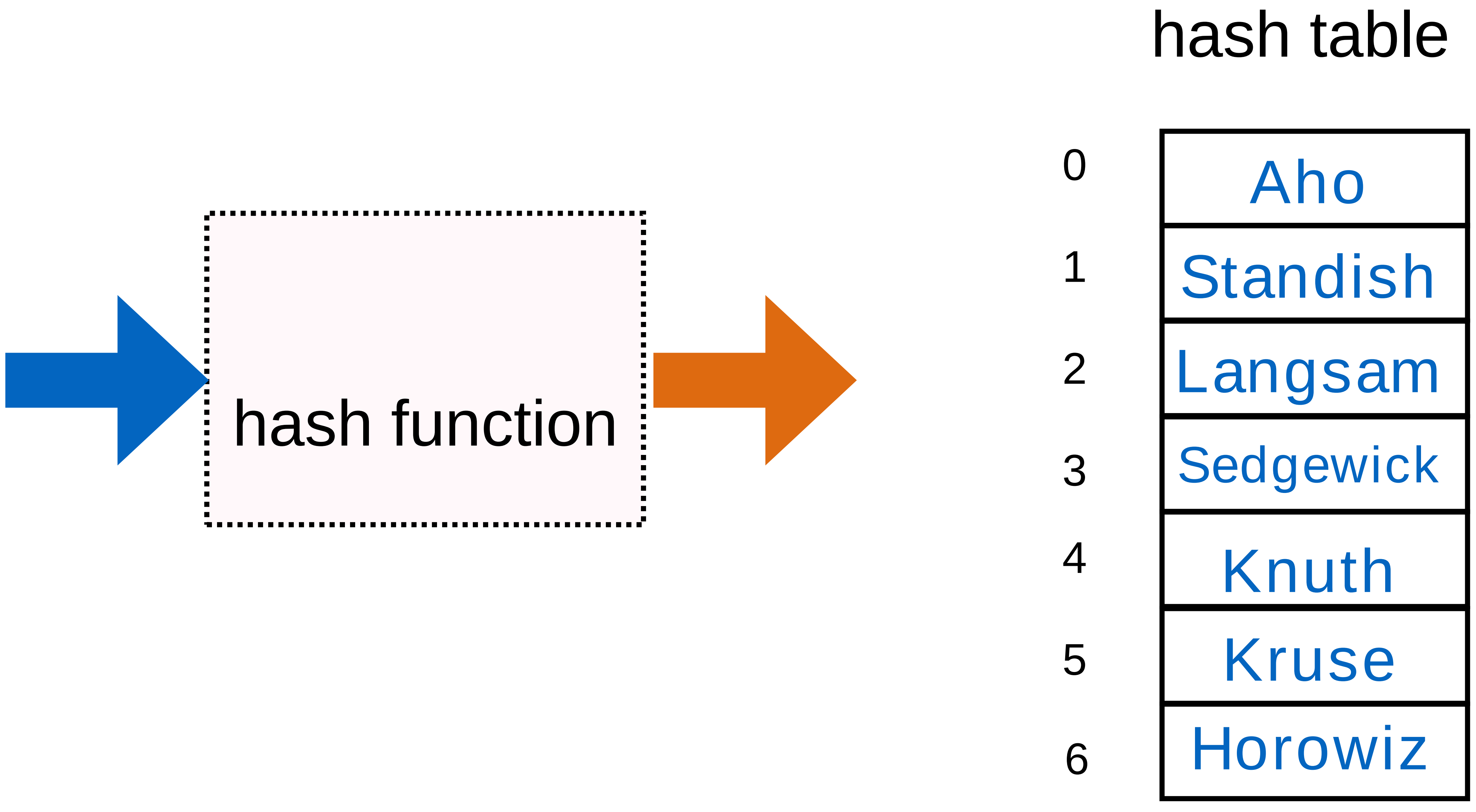
0	Aho
1	Standish
2	Langsam
3	Knuth
4	Sedgewick
5	Kruse
6	Horowitz

A) True

B) False

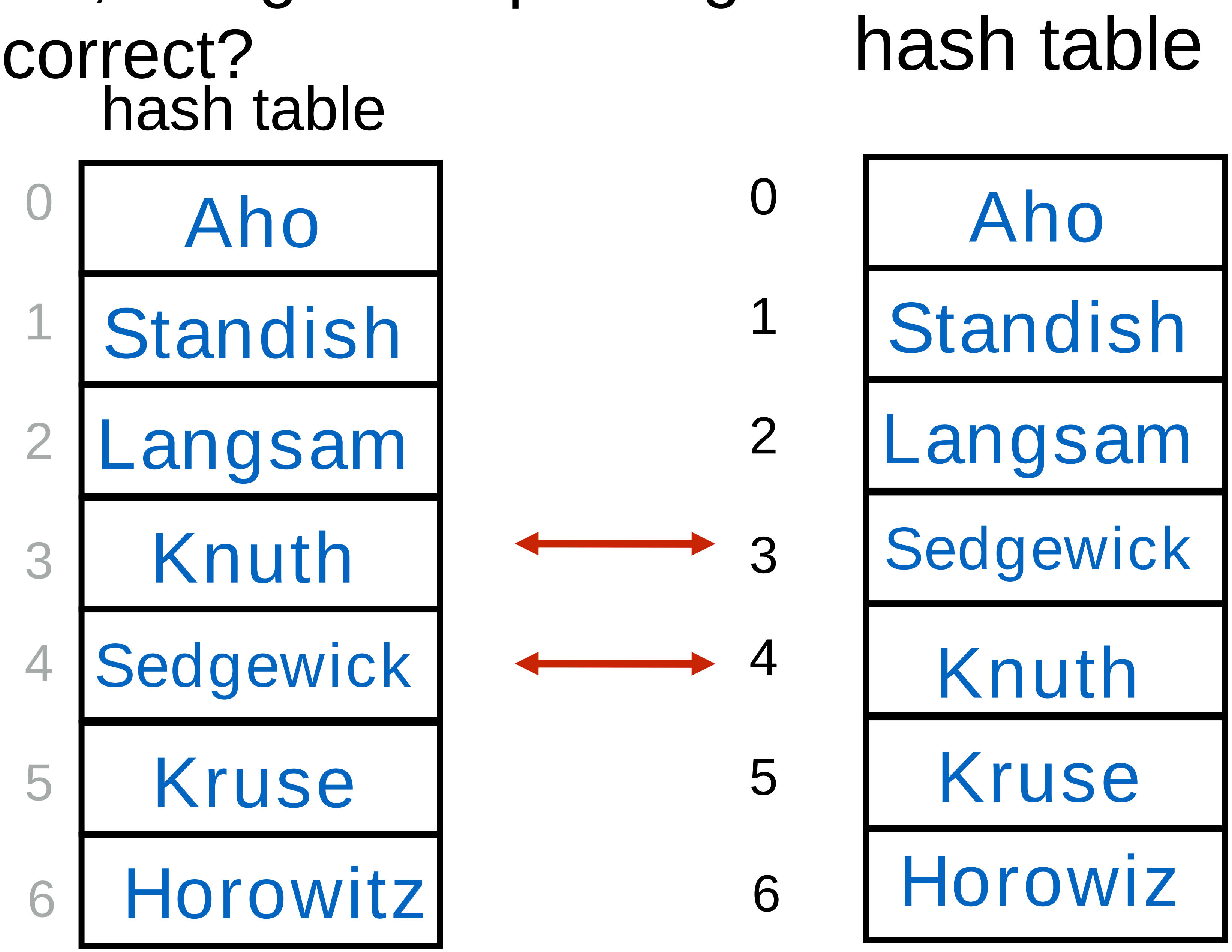
Example

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



Insert the following keys into the Hash Table, in the order they appear, using linear probing. Is the following table correct?

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



A) True

B) False

Let’s implement a Table with Linear Probing….

default size, a prime number

```python
class LinearProbeTable:
    def __init__(self, size=7919):
        self.count = 0
        self.array = [None] * size
        self.table_size = size
```

count: how many items I have stored

array: where I will store things

table_size: size of the underlying array, a prime…

```python
class LinearProbeTable:
    def __init__(self, size=7919):
        self.count = 0
        self.array = [None] * size
        self.table_size = size

    def __len__(self):
        return self.count
```

count: how many items I have stored

overloading operator **`len`** by implementing **`__len__`**

```python
class LinearProbeTable:
    def __init__(self, size=7919):
        self.count = 0
        self.array = [None] * size
        self.table_size = size

    def __len__(self):
        return self.count

    def hash(self, key):
        value = 0
        a = 31415
        b = 27183
        for i in range(len(key)):
            value = (ord(key[i]) + a * value) % self.table_size
            a = a * b % (self.table_size - 1)
        return value
```

Universal hashing

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

base changes for each position pseudorandomly

```python
class LinearProbeTable:
    def __init__(self, size=7919):
        self.count = 0
        self.array = [None] * size
        self.table_size = size

    def __len__(self):
        return self.count

    def hash(self, key):
        value = 0
        a = 31415
        b = 27183
        for i in range(len(key)):
            value = (ord(key[i]) + a * value) % self.table_size
            a = a * b % (self.table_size - 1)
        return value
```

Open Addressing: Linear Probing

- Insert item with hash value N:
  - If array[N] is empty just put item there.
  - If there is already an item there:
    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.

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

hash table

We are storing the key only.

In practice you want to store also some data that you associate with each key.

Key	Data	Hash value
Aho	Data structures and algorithms	0
Kruse	Data structures and program design in C++	5
Standish	Data structures in Java	1
Horowitz	Fundamentals of Data Structures	5
Langsam	Data structures using C and C++	5
Sedgewick	Algorithms in C++	2
Knuth	The art of computer programming	1

hash table

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

We are storing the key only.

In practice you want to store also some data that you associate with each key.

hash table

	key	data
0	Aho	Data structures and algorithms
1	Standish	Data structures in Java
2	Langsam	Data structures using C and C++
3	Sedgewick	Algorithms in C++
4	Knuth	The art of computer programming
5	Kruse	Data structures and program design
6	Horowiz	Fundamentals of Data Structures

hash table

	key	data
0	(Aho ,	Data structures and algorithms)
1	(Standish ,	Data structures in Java)
2	(Langsam ,	Data structures using C and C++)
3	(Sedgewick ,	Algorithms in C++)
4	(Knuth ,	The art of computer programming)
5	(Kruse ,	Data structures and program design)
6	(Horowiz ,	Fundamentals of Data Structures)

my_tuple = (key , data)

Python tuple

my_tuple[0] = key

my_tuple[1] = data

Open Addressing: Linear Probing

(key , data)

- Insert item with hash value N:
  - → If array[N] is empty just put item there.
  - → If there is already an item there:
    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.

Implementation of Serve for an Array-based Circular Queue

We'll use this trick again

```python
def serve(self):
    assert not self.is_empty(), "Queue is empty"
    item = self.the_array[self.front]
    self.front = (self.front+1) % len(self.the_array)
    self.count -=1
    return item
```

```python
self.front +=1
if self.front == len(self.the_array):
    self.front = 0
```

Restarting from position 0 if the end of table is reached

insert(key, data)

- Get the position N using the hash function, N = hash(key)
- If array[N] is empty just put the item (key, data) there.
- If there is <u>already an item there</u>:
  - If there is already something there, with the same key the user is updating the data
  - If there is already something there with a different key, you need to find an empty spot

What if the Table is full?

```python
def insert(self, key, data):
    position = self.hash(key)

        if self.array[position] is None:  # found empty slot
            self.array[position] = (key, data)
            self.count += 1
            return
```

tuple

Position is available.

```python
def insert(self, key, data):
    position = self.hash(key)



        elif self.array[position][0] == key:  # found key
            self.array[position] = (key, data)
            return
```

the key of the tuple currently living at array[position]

Position contains same key (update).

limit iterations to
size of the table

```
def insert(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
```

What if the table is full?

```python
def insert(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
```

Ok, it will not loop forever… but this doesn’t fix the full table situation!

```python
def insert(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.insert(key, data)
```

move everything to a
new larger table and
try again

__setitem__

object.__setitem__(*self*, *key*, *value*)

Called to implement assignment to `self[key]`. Same note as for `__getitem__()`. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the `__getitem__()` method.

https://docs.python.org/3/reference/datamodel.html#object.__setitem__

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

```python
def insert(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.insert(key, data)
```

```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)
```

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