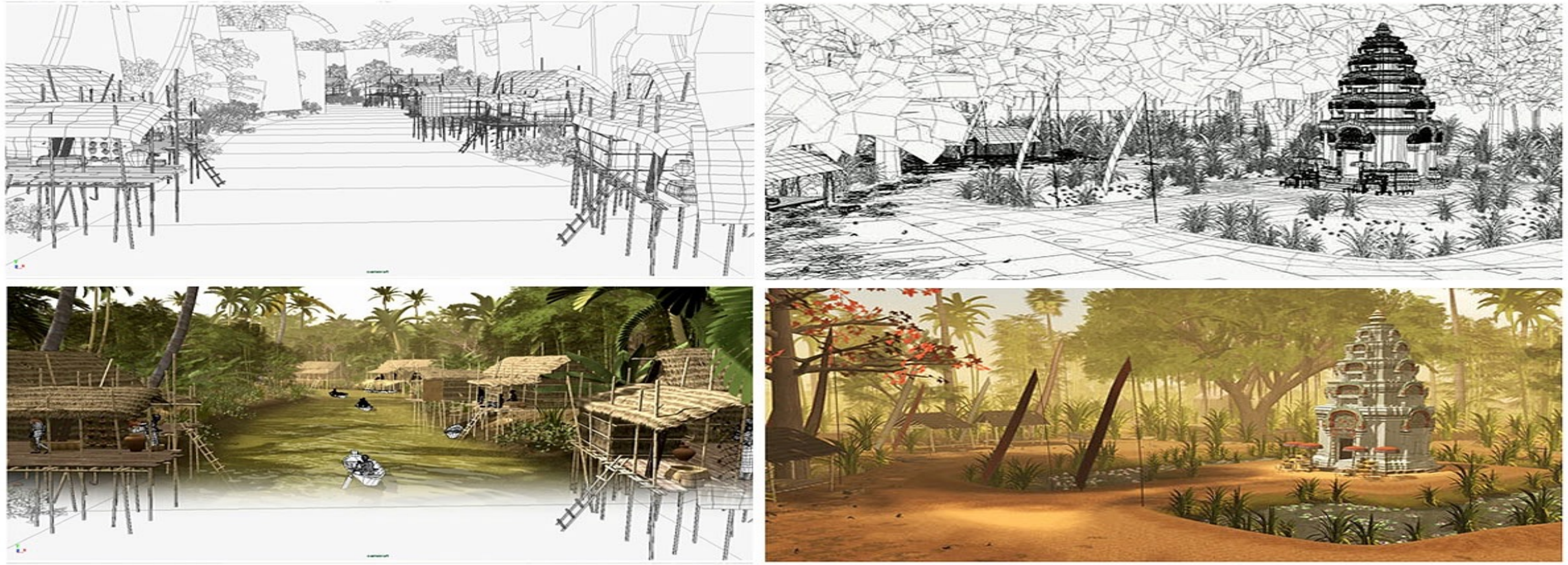
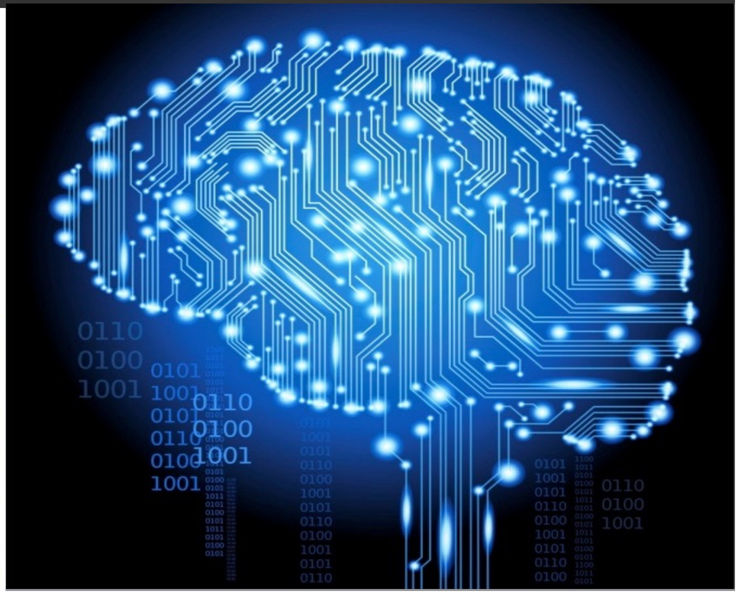
MONASH University

Information Technology

Conflict Resolution with Linear Probing

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor and Alexey Ignatiev





Objectives for this lesson

- **To understand the main method of conflict resolution:**
  - Open addressing:
    - Linear Probing
- **To understand its advantages and disadvantages**
- **To be able to implement it**

Conflict Resolution
Add

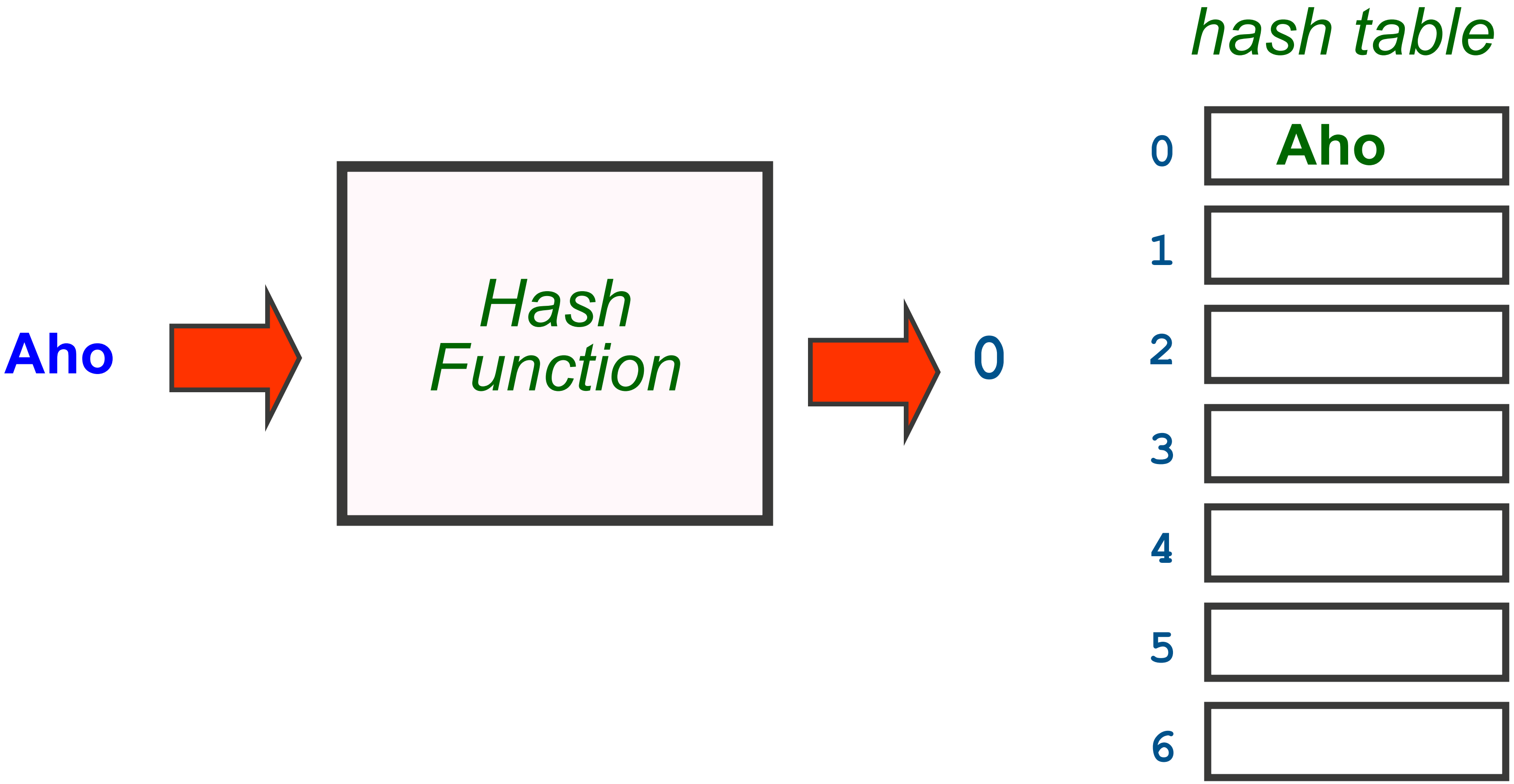
Hash Table operations: Add

- **Apply the hash function to get a hash value (position) N**
- **Try to add key at position N**
- **Deal with collision / conflict if any**

For clarity:
We shall refer to a situation of $hash(key1) = hash(key2)$ as a collision while a conflict is assumed to occur when a position in the hash table *we attempt to use for a given key* is already occupied.

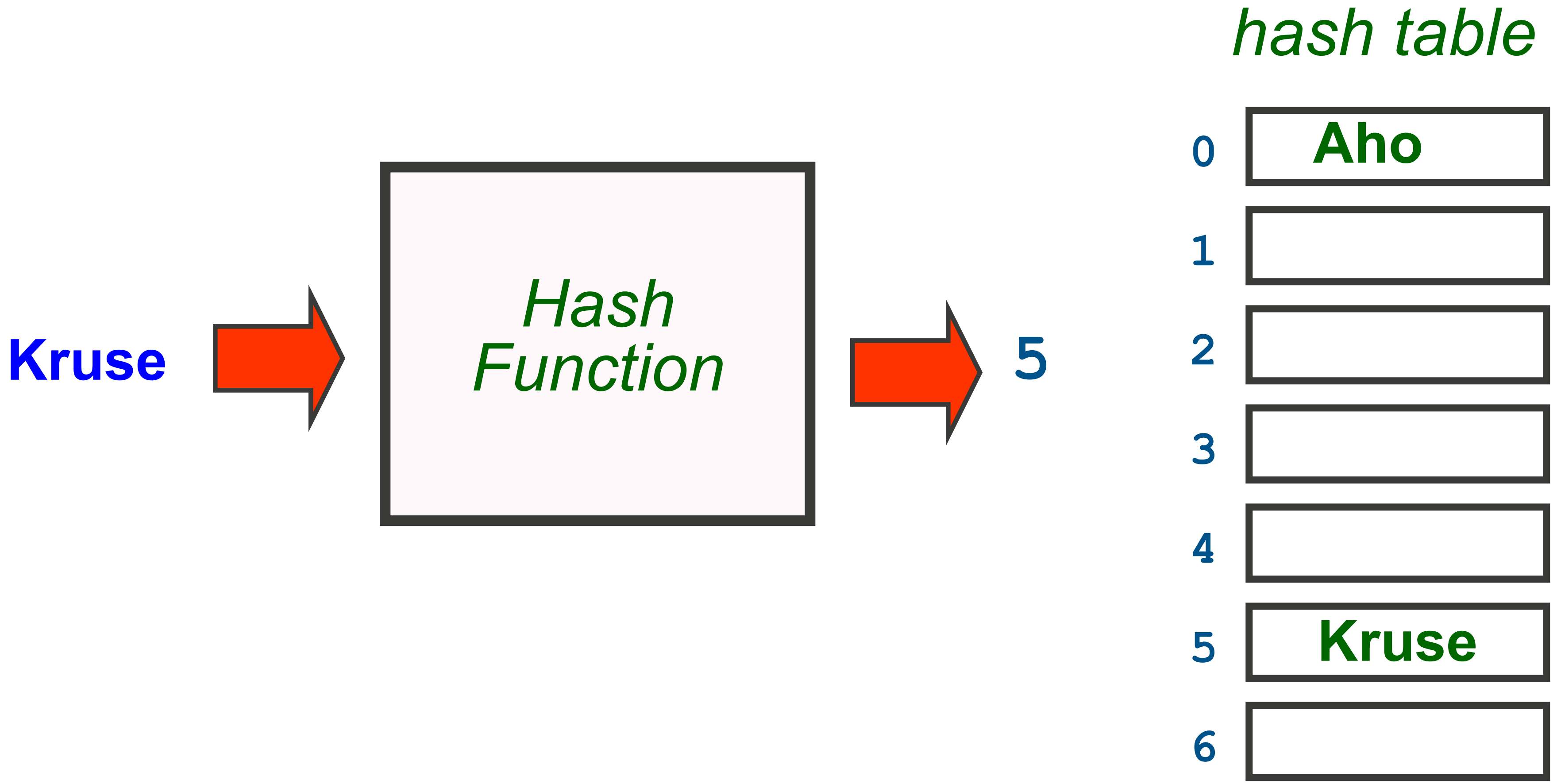
Example: Add

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



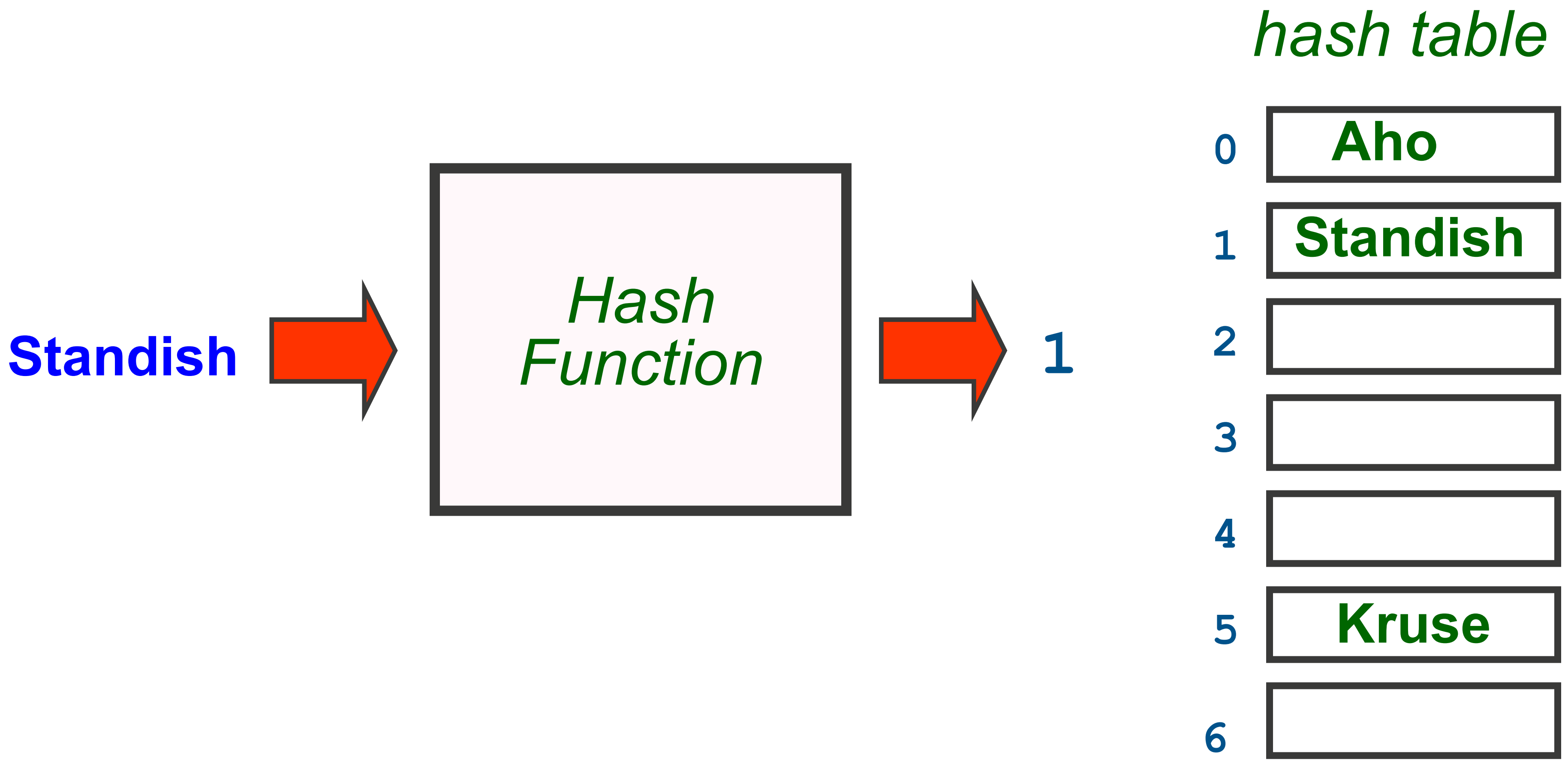
Example: Add

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Example: Add

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Example: Add

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth

$hash(\text{"Horowitz}) = hash(\text{“Kruze”}) = 5$

This is a collision!

Horowitz → *Hash Function* → 5

hash table

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

Collision results in a conflict in the hash table!

What do we do now? How do we resolve this conflict?

Revision: two main approaches to conflict resolution

We've seen this in the previous lesson!

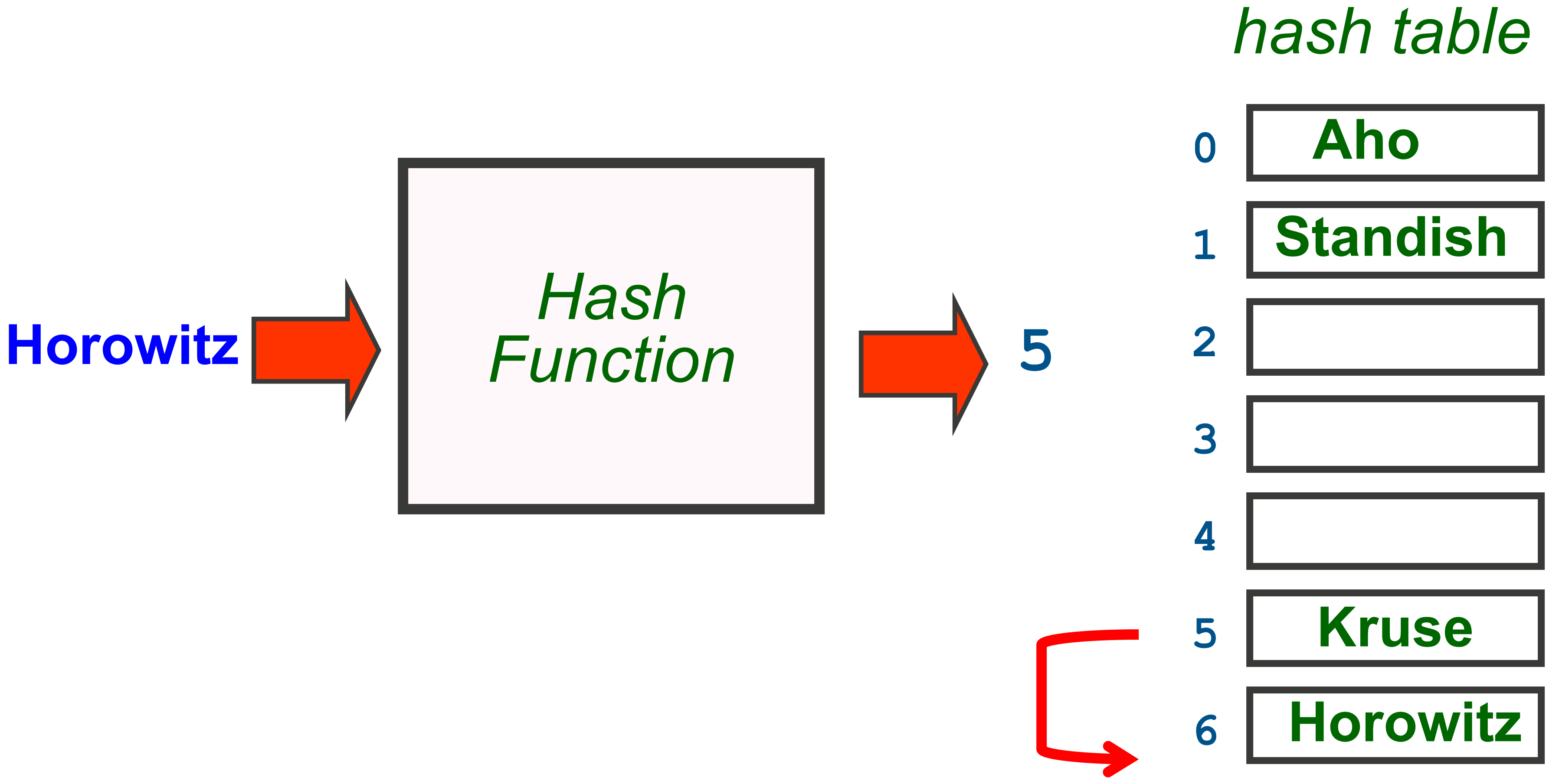
- **Separate chaining:**
  – Each array position contains a linked list of items
  – Upon collision, either update (same key) or add the element to the linked list
- **Open addressing:**
  – Each array position contains a single item
  – Upon collision, either update (same key) or use an empty space to store the new item (which empty space depends on the technique)
  – As we will see:
    - Requires an array of at least double the size of the number of elements
    - Thus, we must be able to estimate in advance the number of elements (or risk a dynamic resize)

Open Addressing: Linear Probing

- **Add item with hash value N:**
  - If array[N] is empty: put item there
  - If there is already an item there with:
    - A different key:
      - search for the first empty space in the array from N+1
      - add the item there (if any)
    - Same key: update the data associated to the key
- **Basically: linear search from N until an empty slot is found**
- **But careful, you must deal with:**
  - Full table (to avoid going into an infinite loop)
  - Restarting from position 0 if the end of table is reached

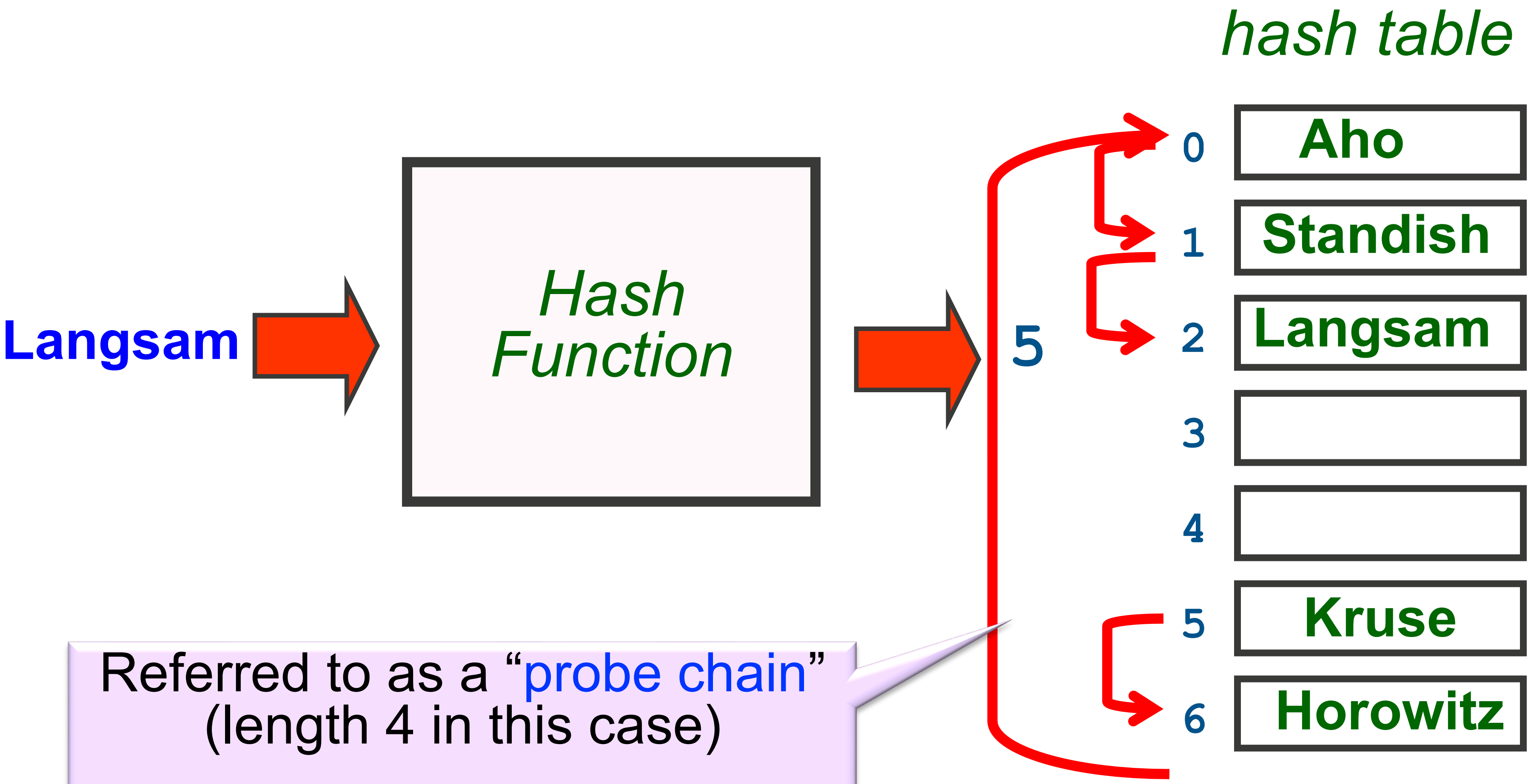
Example: Add

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Example: Add

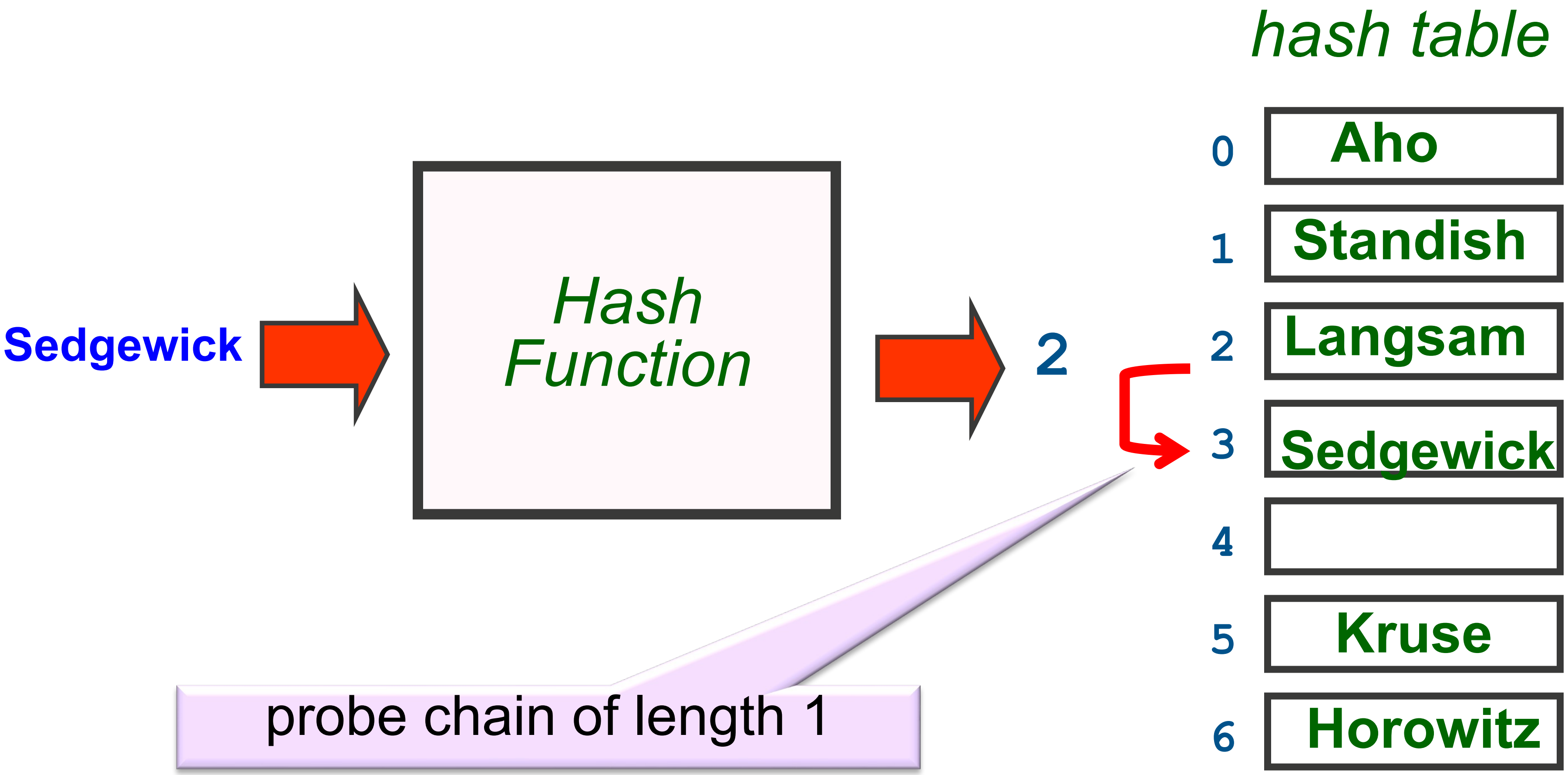
Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Referred to as a “probe chain”
(length 4 in this case)

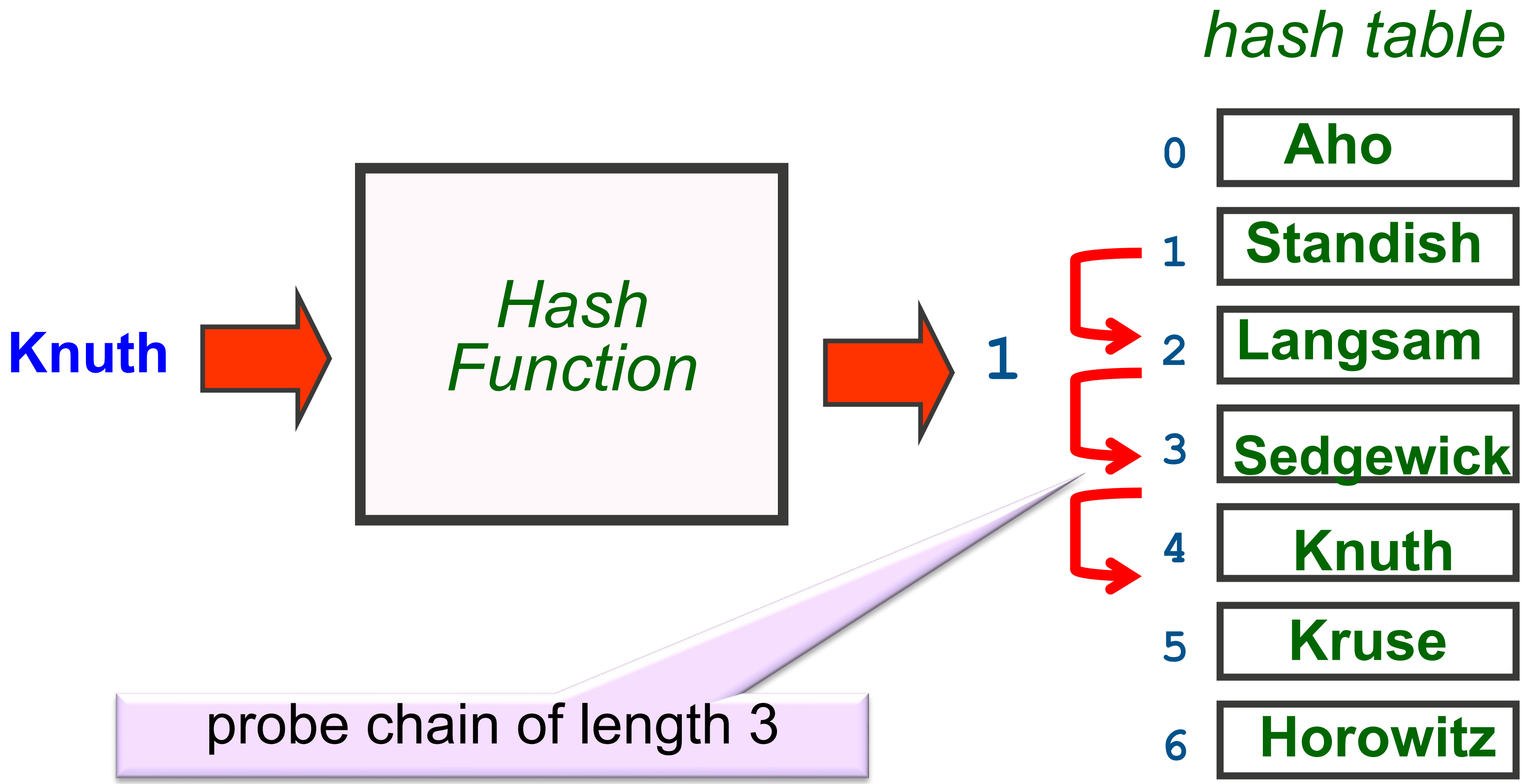
Let’s keep on going

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Let's keep on going

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



```python
from typing import TypeVar, Generic
T = TypeVar('T')


class LinearProbeTable(Generic[T]):

    def __init__(self, size: int = 7919) -> None:
        self.count = 0
        self.table = ArrayR(size)

    def __len__(self) -> int:
        return self.count

    def hash(self, key: str) -> int:
        value = 0
        a = 31415
        b = 27183
        for char in key:
            value = (ord(char) + a*value) % len(self.table)
            a = a * b % (len(self.table)-1)
        return value
```

Default size (a prime)

How many elements?

The array to store them

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 pseudo randomly

Reminder: Adding in Linear Probing

But what is an item?

Up to now, we were storing only the key (the item was the key)

- **Add item with hash value N:**
  - If array[N] is empty: put item there
  - If there is already an item there with:
    - A different key:
      - search for the first empty space in the array from N+1
      - add the item there (if any)
    - Same key: update the data associated to the key
- **Basically: linear search from N until an empty slot is found**
- **But careful, you must deal with:**
  - Full table (to avoid going into an infinite loop)
  - Restarting from position 0 if the end of table is reached

We were storing the key only

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

Hash Table

0	Aho
1	Standish
2	
3	
4	
5	Kruse
6	Horowitz

- **In practice we want to store also data associated to each key**

We also need to store the data

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

Hash Table

0	**Aho**	Data structures and algorithms
1	**Standish**	Data structures in Java
2		
3		
4		
5	**Kruse**	Data structures and program design in C++
6	**Horowitz**	Fundamentals of Data Structures

We also need to store the data (cont)

- **How do we store both key and data in the hash table?**
- **We need an object that stores:**
  - Key
  - Data
- **Have we seen anything already?**
- **Tuples! We could use: (key, data)**
- **We can then access them as usual**

```
>>> my_tuple = ("Aho","Data structures")
>>> my_tuple[0]
'Aho'
>>> my_tuple[1]
'Data structures'
>>>
```

Hash Table

0	(Aho,	Data structures and algorithms)
1	(Standish,	Data structures in Java)
2		
3		
4		
5	(Kruse ,	Data structures and program design in C++)
6	(Horowitz,	Fundamentals of Data Structures)

Reminder: Adding in Linear Probing

- **Add item with hash value N:**
  - If array[N] is empty: put item there
  - If there is already an item there with:
    - A different key:
      - search for the first empty space in the array from N+1
      - add the item there (if any)
    - Same key: update the data associated to the key
- **Basically: linear search from N until an empty slot is found**
- **But careful, you must deal with:**
  - Full table (to avoid going into an infinite loop)
  - Restarting from position 0 if the end of table is reached

But what is an item?

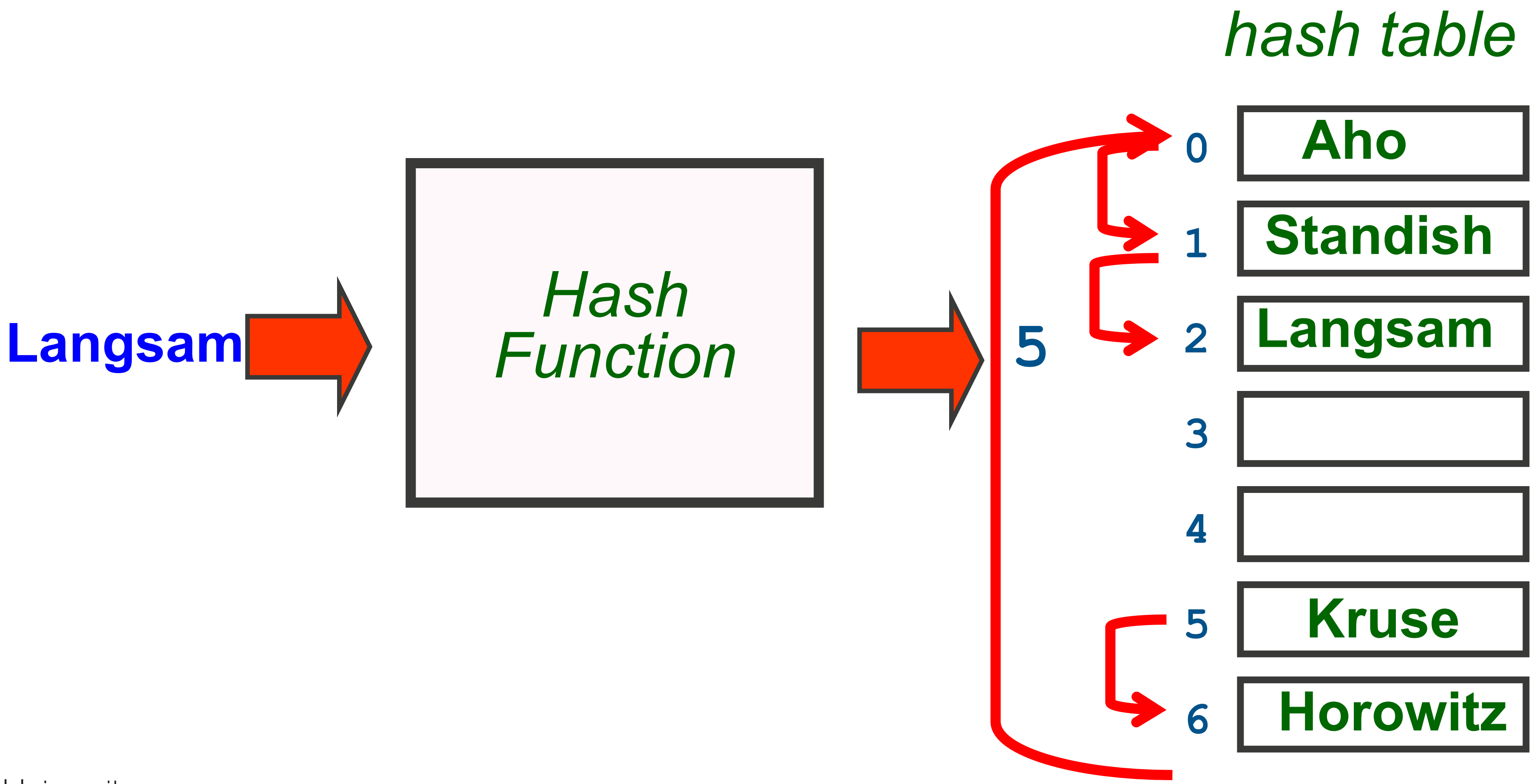
Now we know: an item is a tuple (key,data)

Adding algorithm for Linear Probing

- **The algorithm for `add(key,data)` for linear probing is as follows**
- **Get the position N using the hash function N = hash(key)**
- **If array[N] is empty, just put the item (key, data) there**
- **Else, if there is already an item there:**
  - If the item has the same key: update the data
  - If it has a different key, keep looking in next cell (wrapping around)
    - What if we never find it and there is no empty spot?
    - Then we need to **rehash**: create a bigger array and reinsert all items
- **We must traverse the table before we can rehash:**
  - Even if it is known to be full, in case the key is already in (we are doing an update)
  - Also, as we will see later, rehash in Linear Probing should happen much earlier than when the table is full…

Example: Add

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Allows our container class to provide the [ ] notation

```python
def __setitem__(self, key: str, data: T) -> None:

    position = self.hash(key) # get the position using hash

    for _ in range(len(self.table)): # start traversing
        if self.table[position] is None: # found empty slot
            self.table[position] = (key, data)
            self.count += 1
            return

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

        else: # not found, try next
            position = (position+1) % len(self.table)

    self.rehash() # move everything to a new, larger table
    self.__setitem__(key, data) #try again
```

Traverse each item in our hash table from position

Item already exists, overwrite the data

Linear Probing

Your turn..

- **Write `__str__` for a Linear Probe hash table, e.g.:**

  (Aho,Data structures and algorithms)
  (Standish, Data Structures in Java)

Hash Table

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

```
def __str__(self) -> str:
    result = ""
    for item in self.array:
        if item is not None:
            (key, value) = item
            result += "(" + str(key) + "," + str(value) + ")\n"
    return result
```

But we are traversing the Hash Table! Didn't we say not to do that?

No! We said not to traverse IN A PARTICULAR ORDER

Conflict Resolution Search

Searching in 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 (if so, raise a KeyError(key) exception)
- **But careful, you must deal again with:**
  - Full table (to avoid going into an infinite loop)
  - Restarting from position 0 if the end of table is reached

Searching algorithm for Linear Probing

- **The algorithm for `search(key,data)` is as follows**
- **Get the position N using the hash function N = hash(key)**
- **If array[N] is empty, raise a `KeyError(key)` exception**
- **Else, if there is already an item there:**
  - If the item has the same key: return the associated data
  - If it has a different key, keep looking
    - What if we never find the key and there is no empty spot?
    - Then we raise a `KeyError(key)` exception

- **We used `__setitem__` for adding**
- **We will use `__getitem__` for searching**

```
def __getitem__(self, key: str) -> T:

    position = self.hash(key) # get the position using hash

    for _ in range(len(self.table)): # start traversing
        if self.table[position] is None: # found empty slot
            raise KeyError(key) # so the key is not in
        elif self.table[position][0] == key:# found key
            return self.table[position][1] #return data
        else: # there is something but not the key, try next
            position = (position+1) % len(self.table)

    # At this point, I have gone through the table and not found
    raise KeyError(key)
```

Traverse each item in our hash table from position

Stop if we find an empty slot

Linear Probing

```python
def __linear_probe(self, key: str, is_search: bool) -> int:

    position = self.hash(key)

    for _ in range(len(self.table)):
        if self.table[position] is None: # found empty slot
            if is_search: # if searching
                raise KeyError(key) # key is not in
            else:
                return position  # if adding, return position
        elif self.table[position][0] == key:  # found key
            return position
        else:  # there is something but not the key, try next
            position = (position + 1) % len(self.table)

    raise KeyError(key)
```

Traverse each item in our hash table from position

If we're searching for an item (eg. __getitem__)

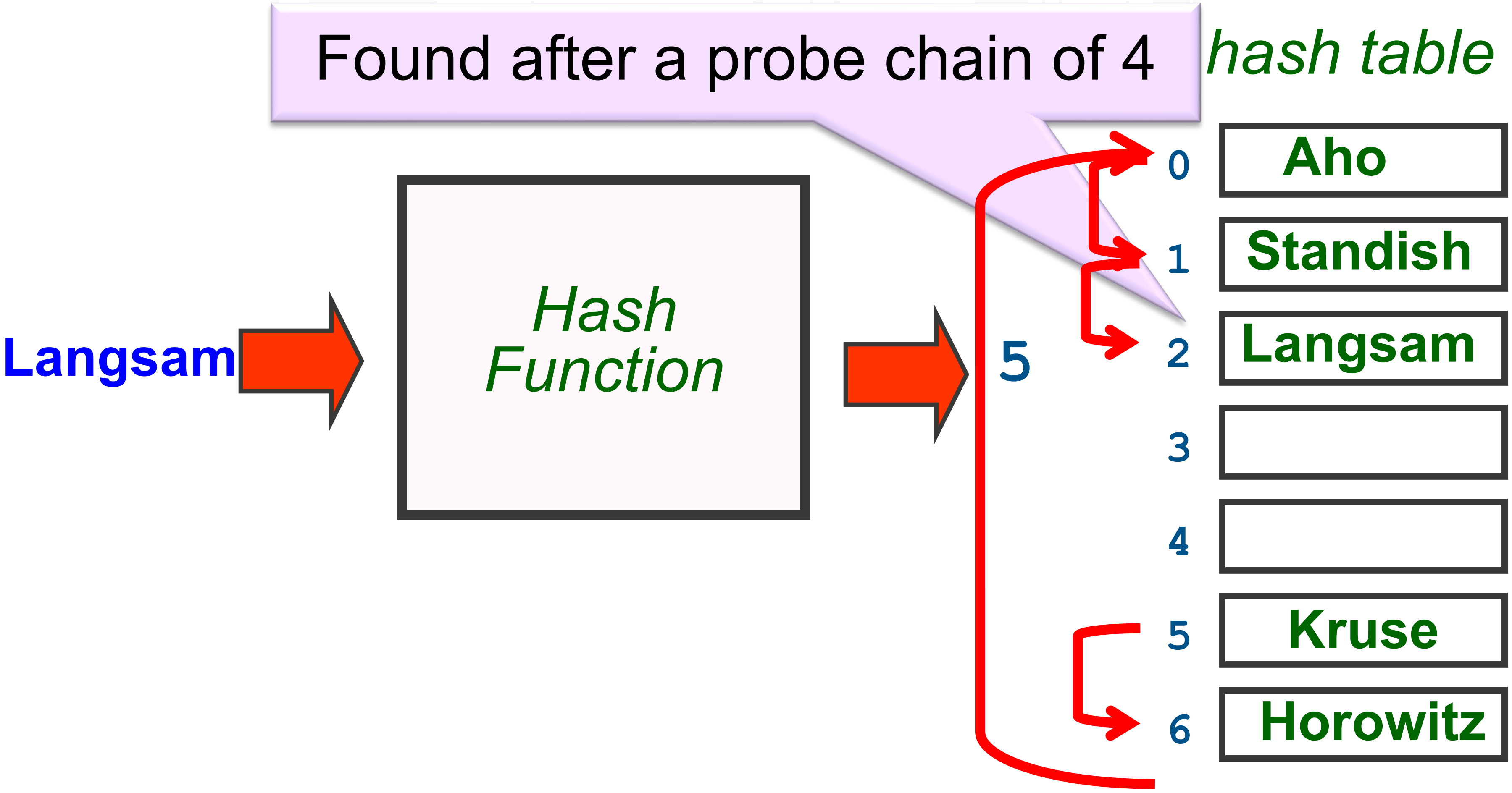
Linear Probing

```python
def __getitem__(self, key: str) -> T:
        position = self.__linear_probe(key, True)
        return self.table[position][1]

def __setitem__(self, key: str, data: T) -> None:
        try:
            position = self.__linear_probe(key, False)
        except KeyError:
            self.__rehash()
            self.__setitem__(key, data)   # try again
        else:
            if self.table[position] is None:    # if it's a new item
                self.count += 1
            self.table[position] = (key, data)
```

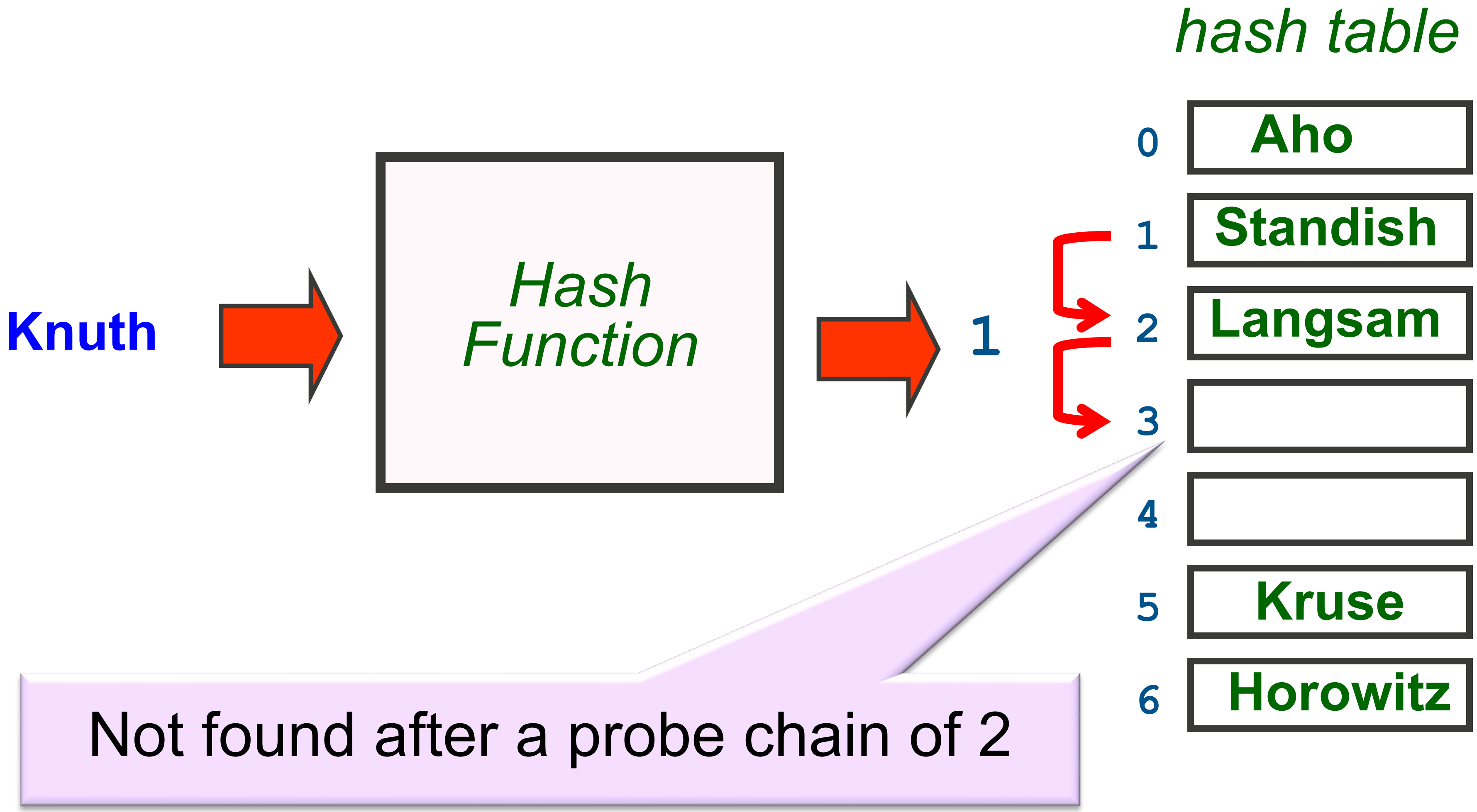
Will raise a KeyError if not found

Full Hash Table, need to resize

Example: search



Example: search



Conflict Resolution
Delete

Deleting in Linear Probing

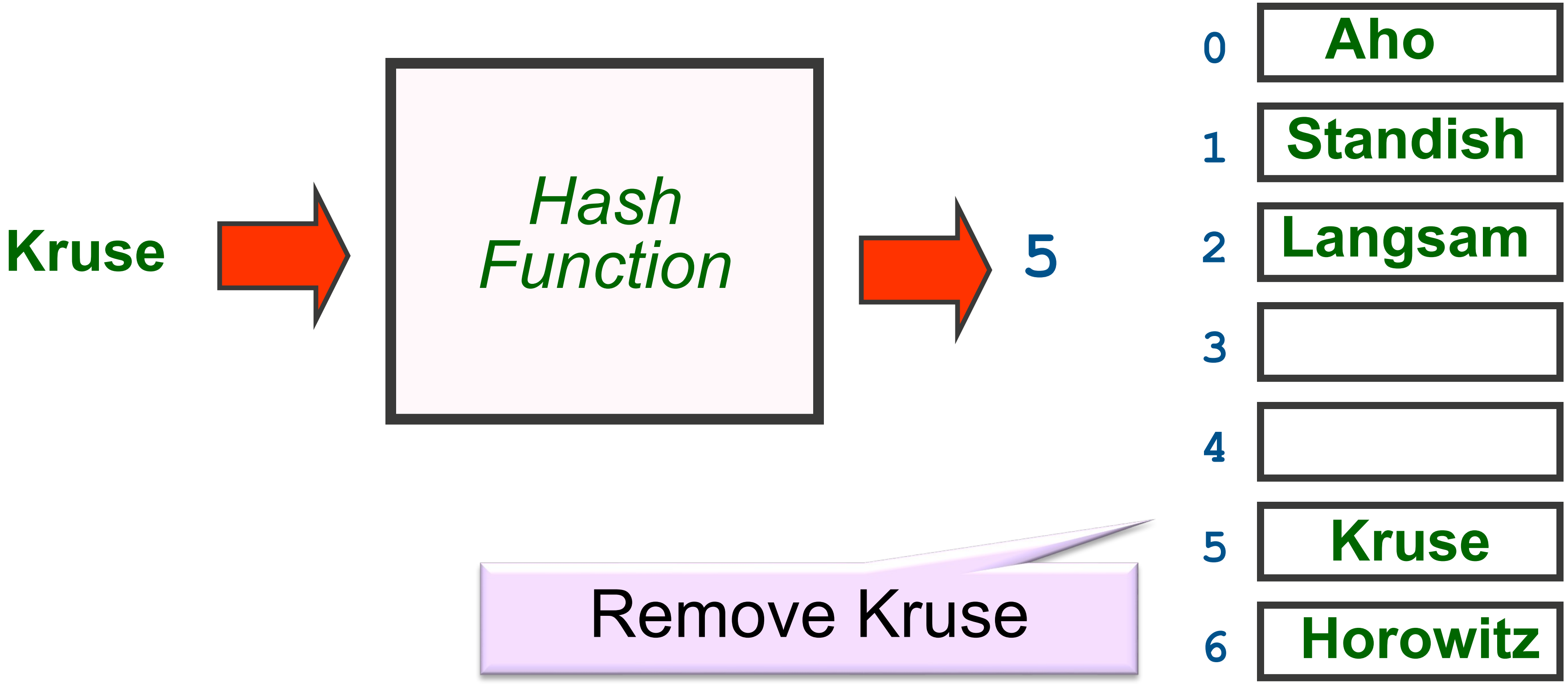
- **What about delete?**
  - Use the search function to find the item
  - If found at N, then what?
- **Should we simply delete it and leave it empty?**
  - No, as empty spots have meaning in linear probing…

Invariant in Linear Probing: if an item with hash(key)=N is in the table, it will always appear between N and the first empty position (wrapping around)

Example: bad delete I

- **Assume we delete Kruse and just leave it empty**

hash table



Example: bad delete I

- **Assume we delete Kruse and just leave it empty**
- **If we now search for Horowitz (key 5)?**
  - It will say it does not find it (raise `KeyError`)

```
def __linear_probe(self, key: str, is_search: bool) -> int:
    position = self.hash(key)

    for _ in range(len(self.table)):
        if self.table[position] is None:
            if is_search:
                raise KeyError(key)
        else:
            return position
        elif self.table[position][0] == key:
            return position
        else:
            position = (position + 1) % len(self.table)
    raise KeyError(key)
```

hash table

0	Aho
1	Standish
2	Langsam
3	
4	
5	
6	Horowitz

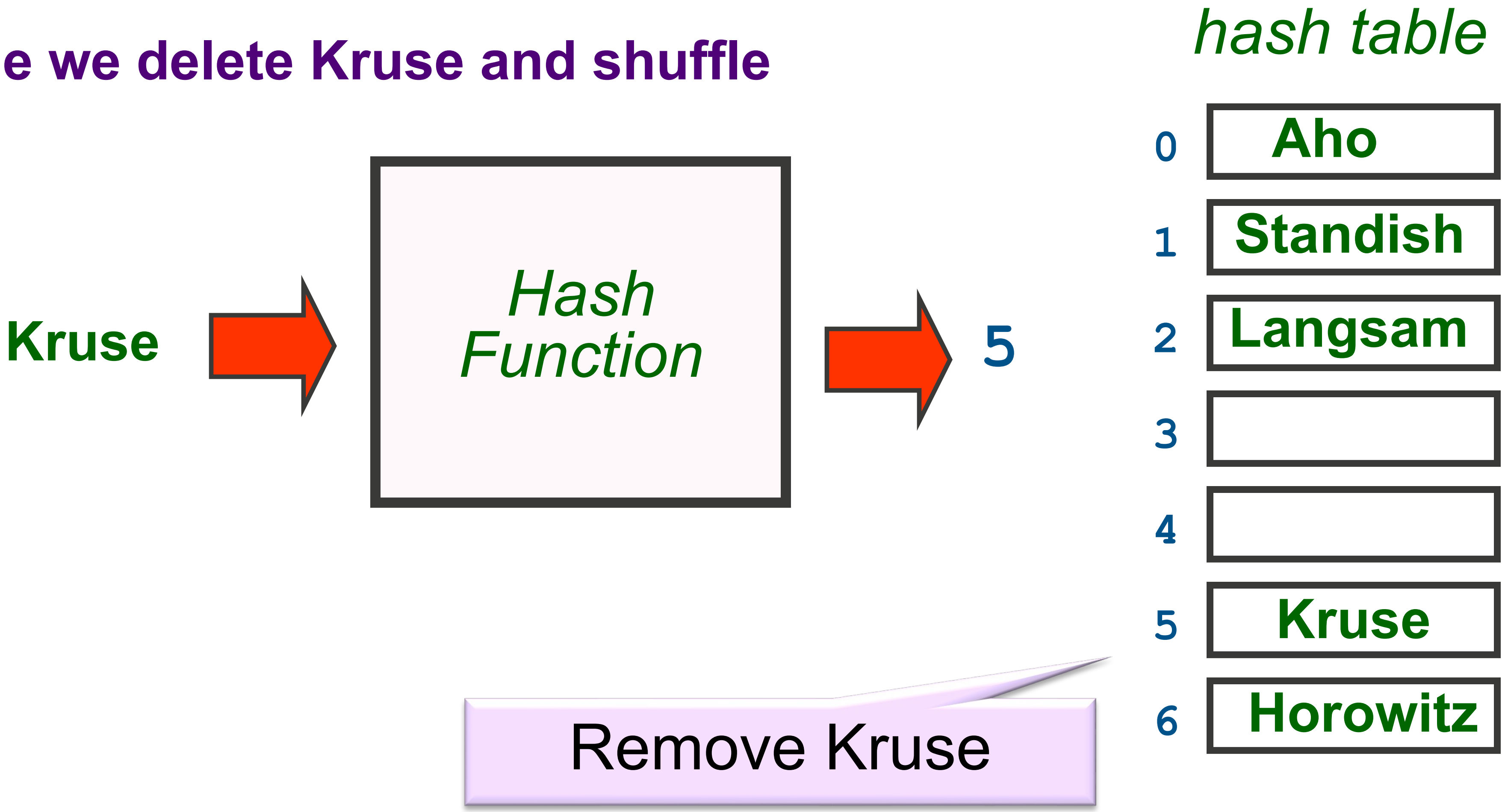
Deleting in Linear Probing

- **What about delete?**
  - Use the search function to find the item
  - If found at N, then what?
- **Should we simply delete it and leave it empty?**
  - No, as empty spots have meaning in linear probing…
- **Should we shuffle everything from N+1 upwards?**
  - From N+1 to what?
    - To the first empty position
  - Is shuffling a good idea though?
    - No, we might move items that were in the correct positions!

Invariant in Linear Probing: if an item with hash(key)=N is in the table, it will always appear between N and the first empty position (wrapping around)

Example: bad delete II

- **Assume we delete Kruse and shuffle**



Example: bad delete II

- **Assume we delete Kruse and shuffle**

hash table

0	**Aho**
1	**Standish**
2	**Langsam**
3	
4	
5	**Horowitz**
6	

Shuffle up until first empty position

Example: bad delete II

- **Assume we delete Kruse and shuffle**

hash table

0	
1	**Standish**
2	**Langsam**
3	
4	
5	**Horowitz**
6	**Aho**

Shuffle up until first empty position

Example: bad delete II

- **Assume we delete Kruse and shuffle**

hash table

0	Standish
1	
2	Langsam
3	
4	
5	Horowitz
6	Aho

Shuffle up until first empty position

Example: bad delete II

- **Assume we delete Kruse and shuffle**
- **If we now search for Horowitz (key 5)?**
  - Will find it without problem
- **And if we search for Aho (key 0)?**
  - Will not find it
- **Shuffling can incorrectly move elements**

hash table

0	Standish
1	Langsam
2	
3	
4	
5	Horowitz
6	Aho

Deleting in Linear Probing

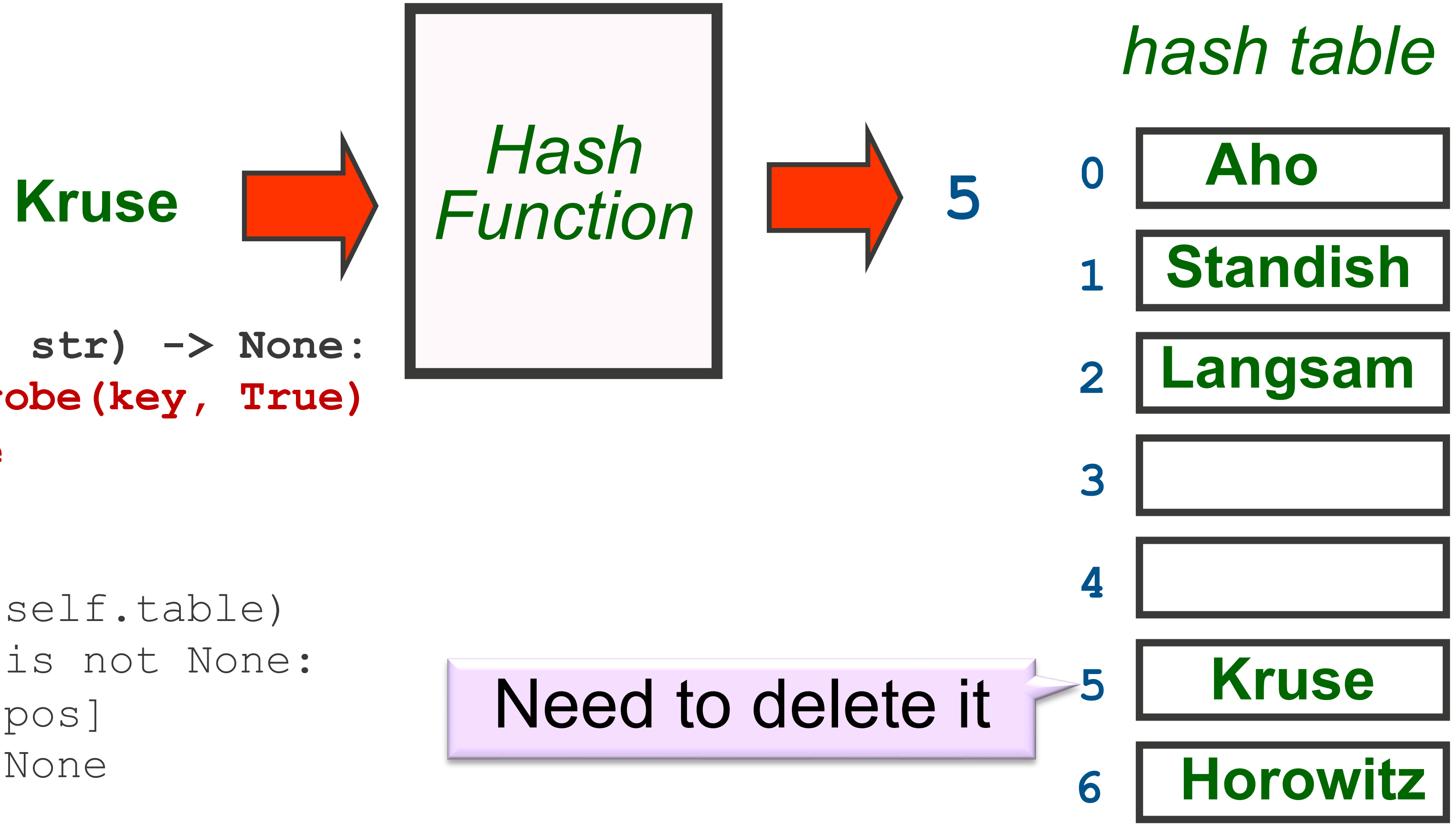
- **What about delete?**
  - Use the search function to find the item
  - If found at N, then what?
- **Should we simply delete it and leave it empty?**
  - No, as empty spots have meaning…
- **Should we shuffle everything from N+1 upwards?**
  - From N+1 to what?
    - To the first empty position
  - Is shuffling a good idea though?
    - No, we might move items that were in the correct positions!

Invariant in Linear Probing: if an item with hash(key)=N is in the table, it will always appear between N and the first empty position (wrapping around)

- **One possibility:**
  - If found at N, reinsert every item from N+1 to the first empty position
  - Time consuming! (though should not be many)

Example: delete

pos 5




```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

Example: delete

pos → 6

```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

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

hash table

0	Aho
1	Standish
2	Langsam
3	
4	
5	
6	Horowitz

Reinsert Horowitz

Example: delete

pos → 6

```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

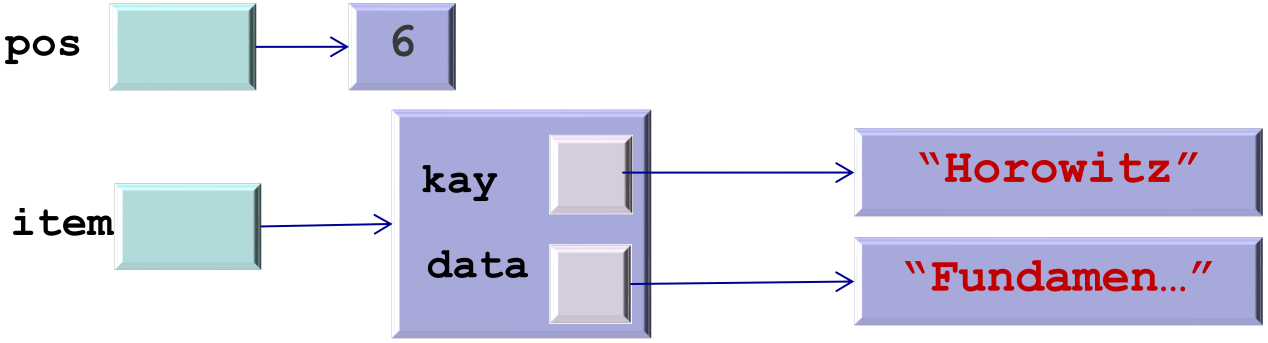
Key	Hash
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

hash table

0	Aho
1	Standish
2	Langsam
3	
4	
5	
6	Horowitz

Reinsert Horowitz

Example: delete

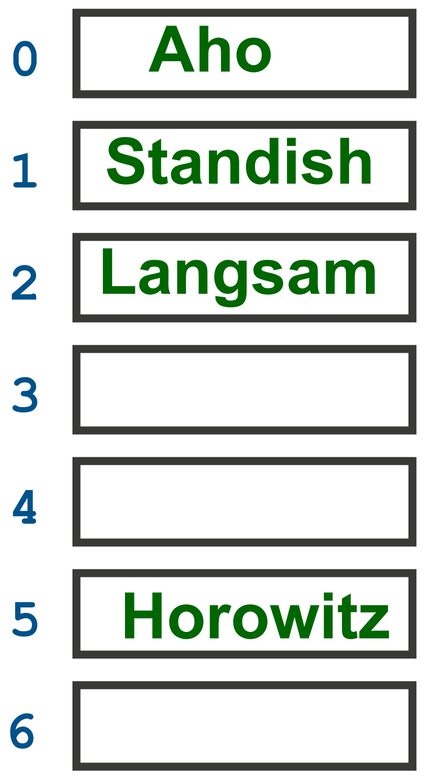



```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

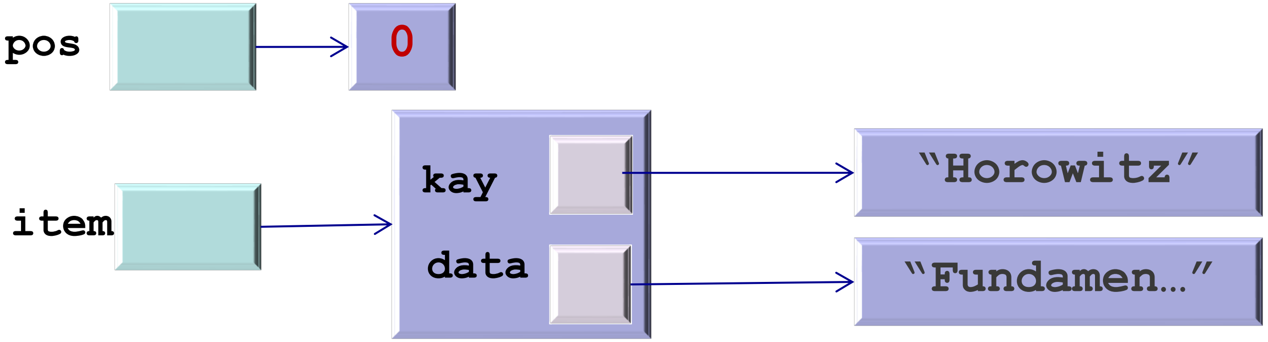
Reuse our __setitem__

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

hash table



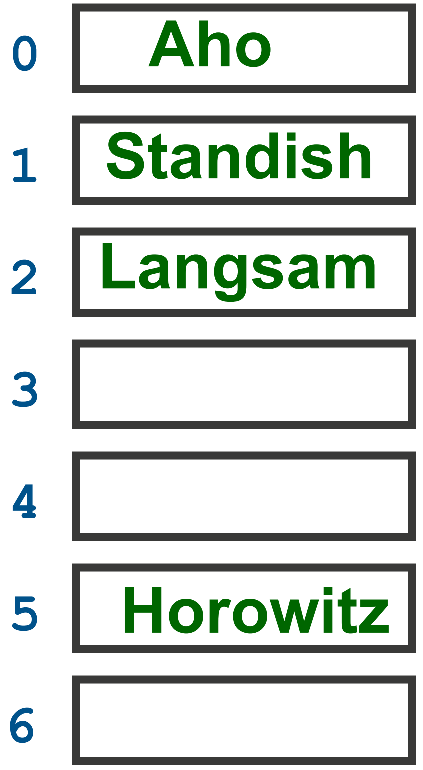
Example: delete




```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

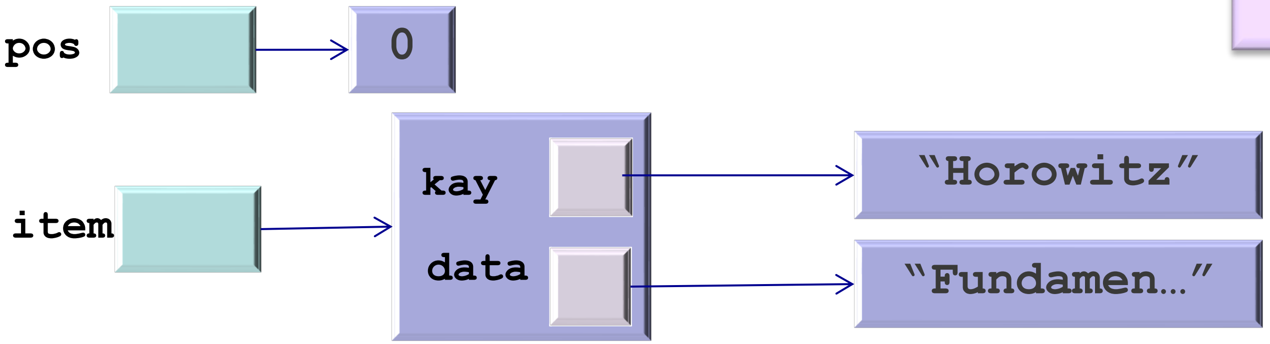
Key	Hash
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

hash table



Example: delete

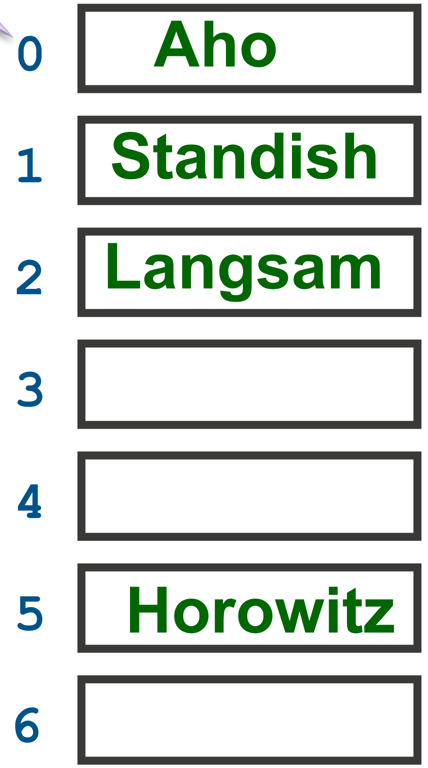
Reinsert Aho




```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

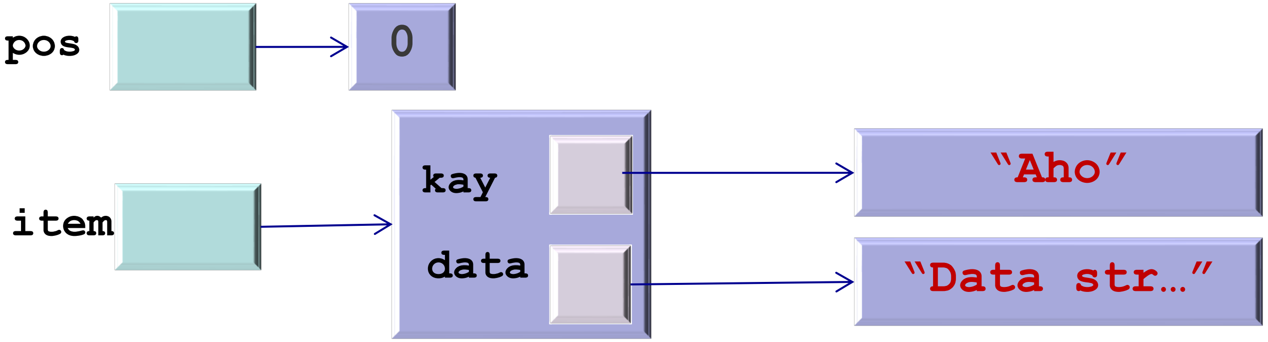
Key	Hash
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

hash table



Example: delete

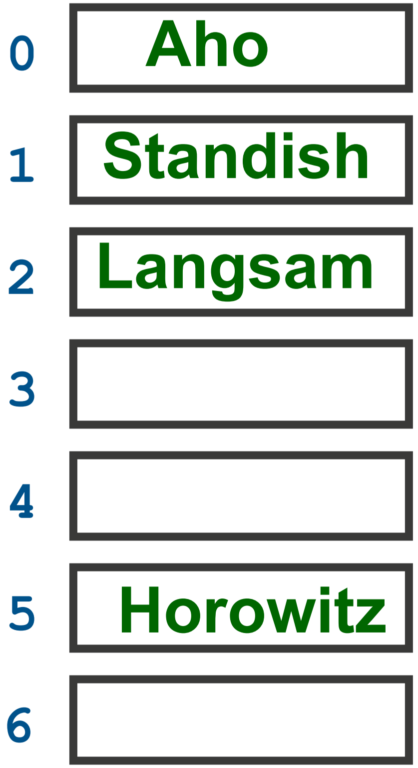
Stays the same



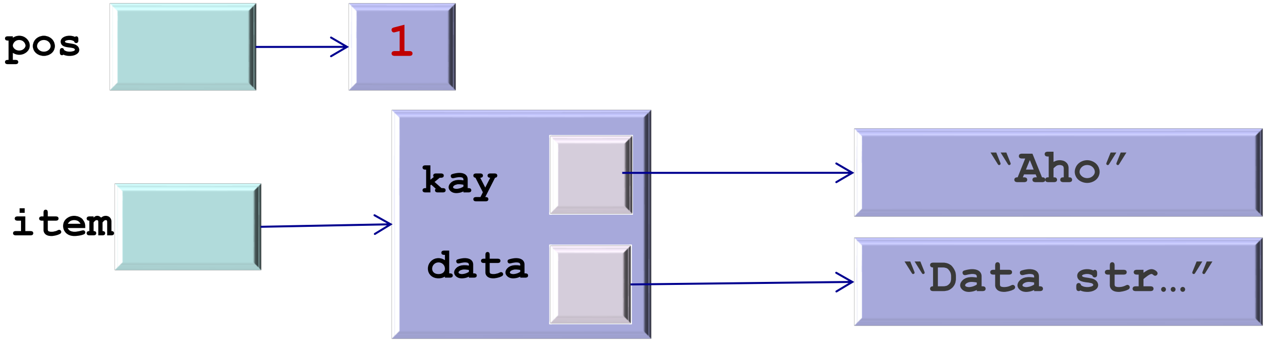

```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

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

hash table



Example: delete




```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

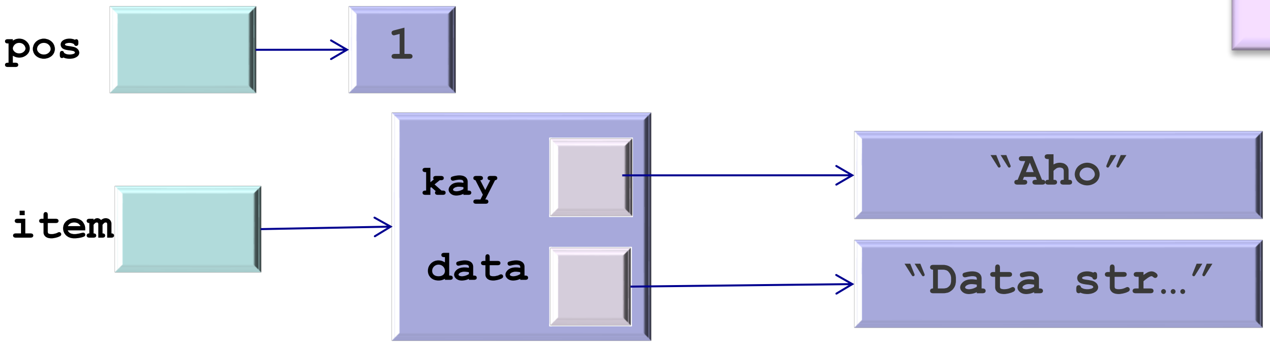
Key	Hash
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

hash table

0	Aho
1	Standish
2	Langsam
3	
4	
5	Horowitz
6	

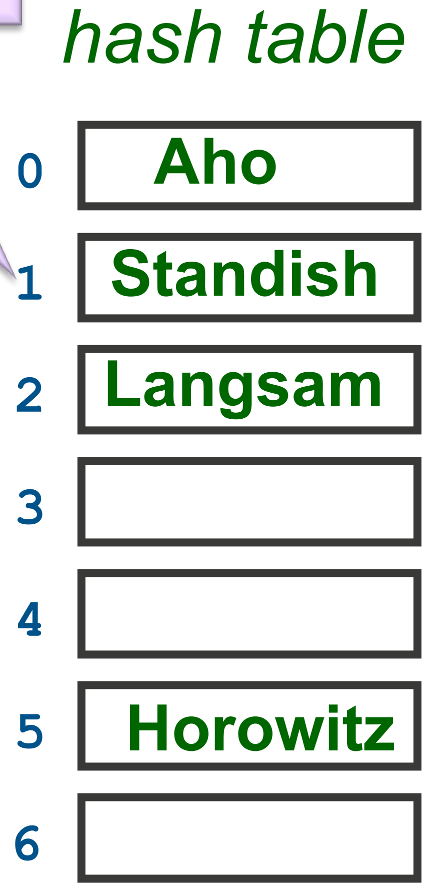
Example: delete

Reinsert Standish

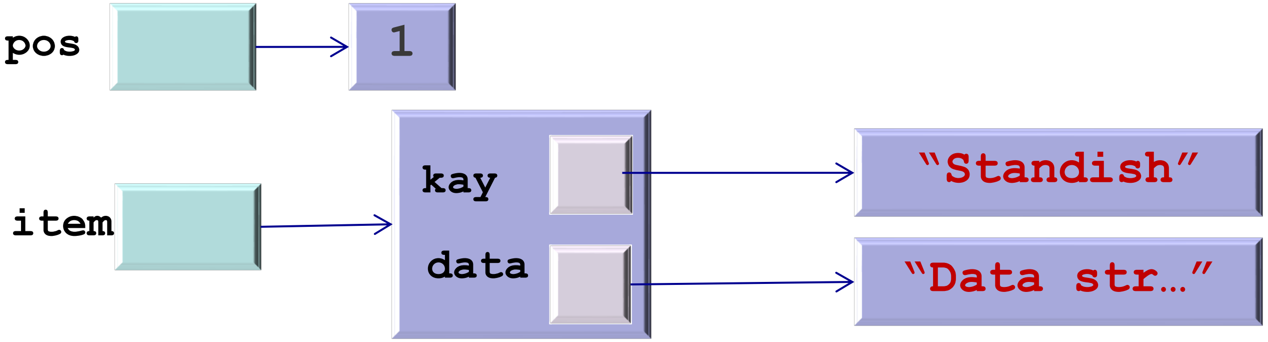



```python
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

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



Example: delete




```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

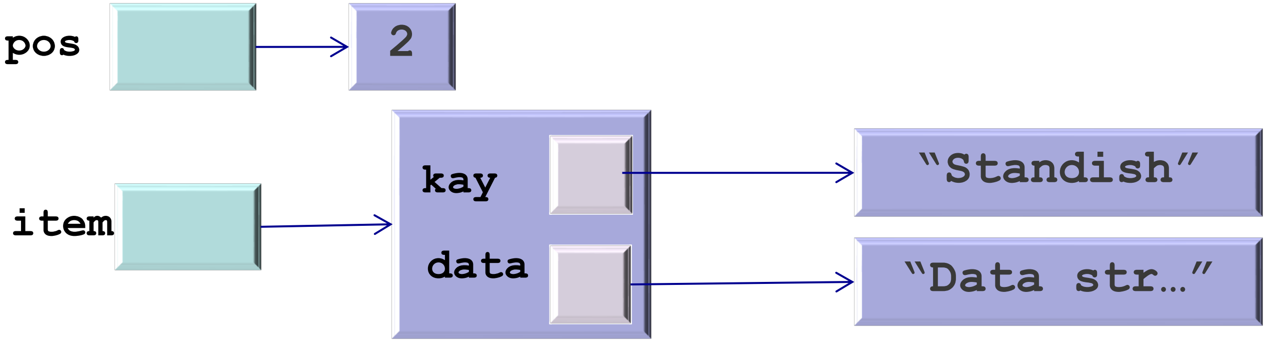
Stays the same

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

hash table

0	Aho
1	Standish
2	Langsam
3	
4	
5	Horowitz
6	

Example: delete



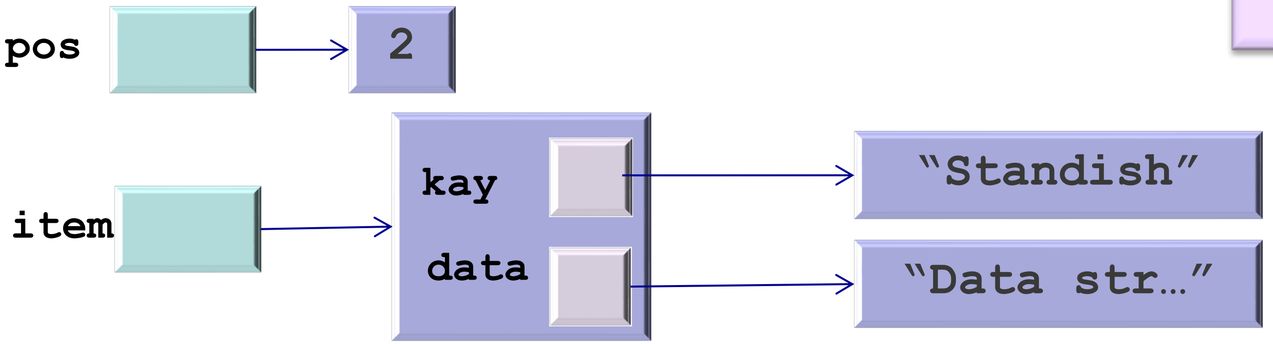

```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

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

hash table

0	Aho
1	Standish
2	Langsam
3	
4	
5	Horowitz
6	

Example: delete

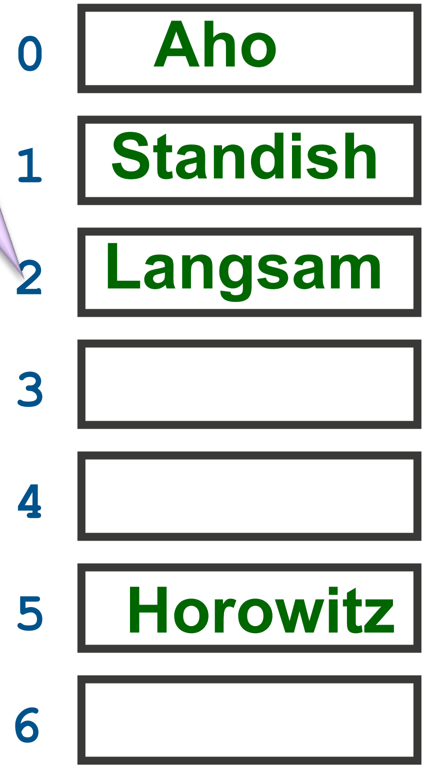



```python
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

Reinsert Langsam

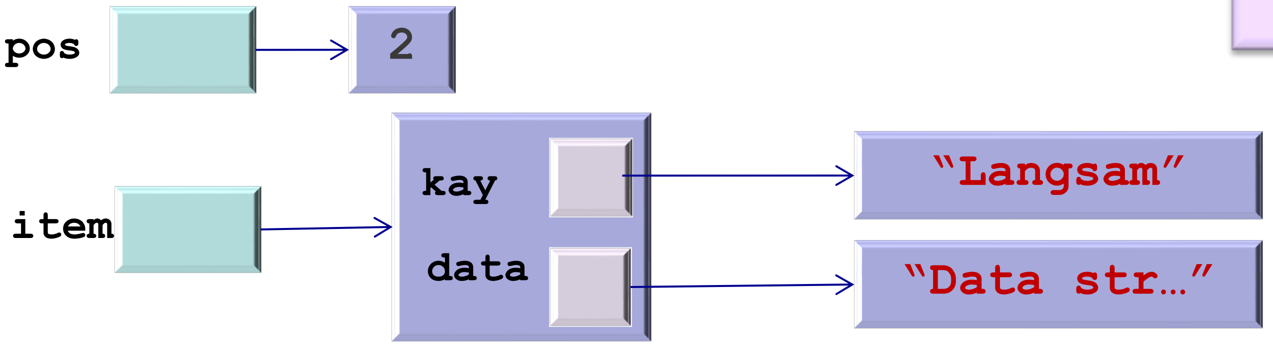
Key	Hash
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

hash table



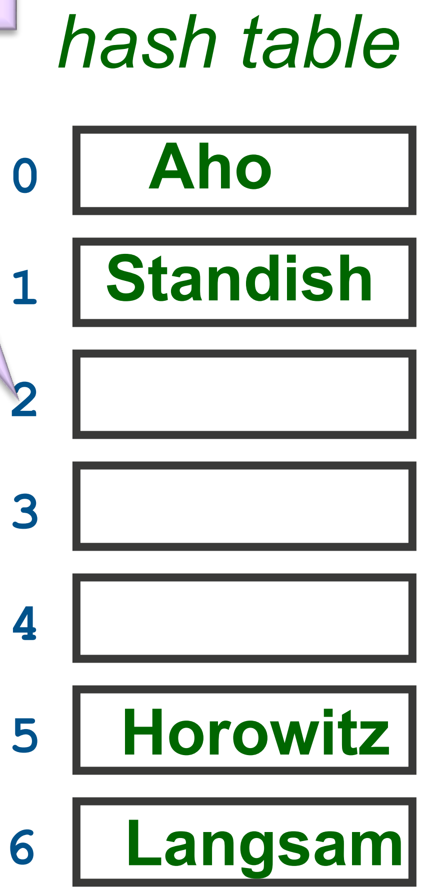
Example: delete

Reinsert Langsam

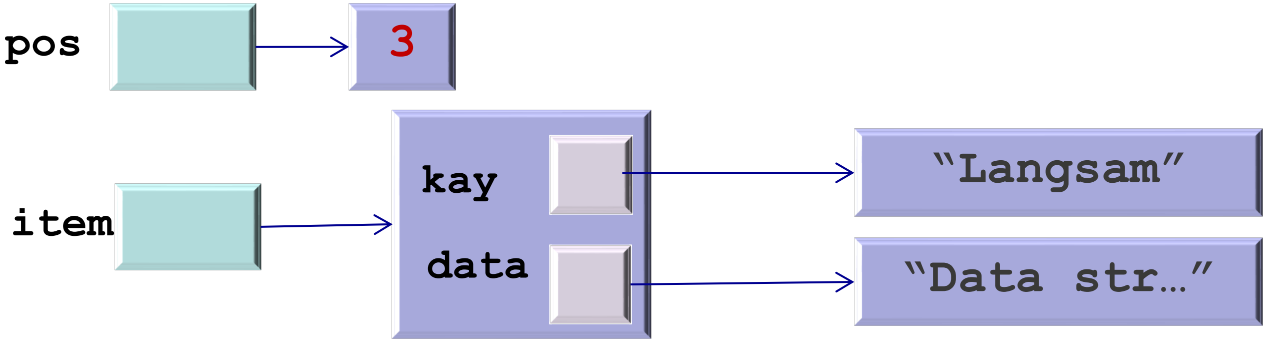



```python
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

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



Example: delete




```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

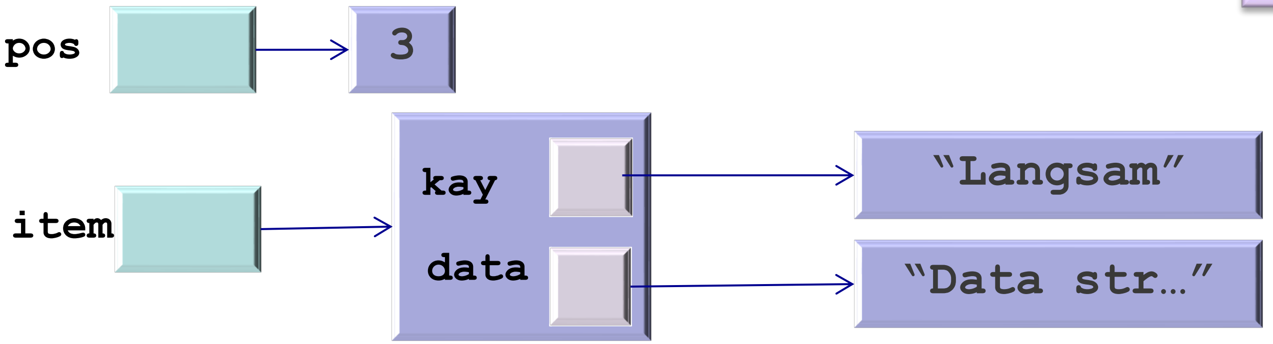
Key	Hash
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

hash table

0	Aho
1	Standish
2	
3	
4	
5	Horowitz
6	Langsam

Example: delete

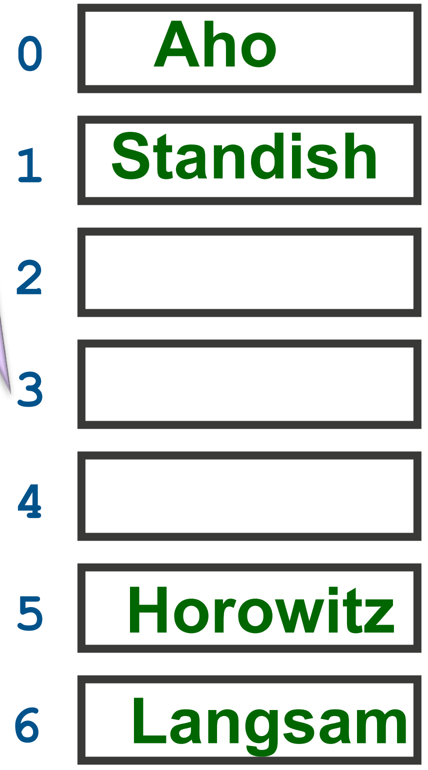
Reached empty, finish



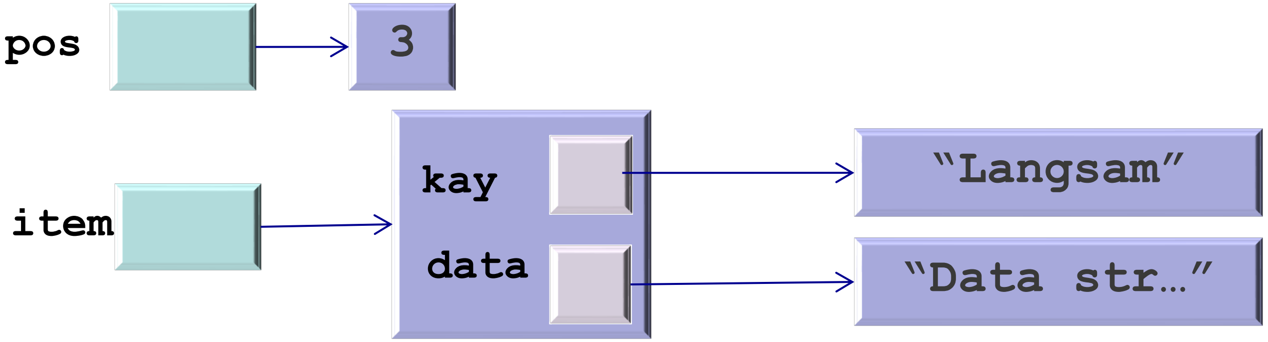

```
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

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

hash table

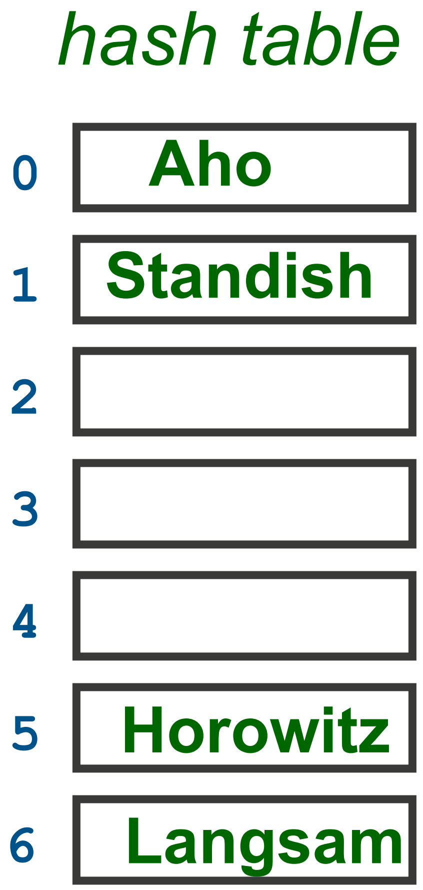


Example: delete




```python
def __delitem__(self, key: str) -> None:
    pos = self.__linear_probe(key, True)
    self.table[pos] = None
    self.count -= 1

    pos = (pos + 1) % len(self.table)
    while self.table[pos] is not None:
        item = self.table[pos]
        self.table[pos] = None
        self.count -= 1
        self[str(item[0])] = item[1]
        pos = (pos + 1) % len(self.table)
```

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



Conflict Resolution
Open Addressing

Open Addressing: Linear Probing

- **Another possibility for delete:**
  - Use a special symbol (a sentinel) to denote delete
  - Modify add and search to take that symbol into account
    - For search: treat the sentinel as you would treat a cell with a key different to the one you are searching for (keep on looking)
      - If found, you could move it to the first deleted cell you found
      - What else would need to be done?
    - For add: treat it as empty and add it in the first deleted cell
      - But this only works for new keys (those not already in the table)
      - If you want updates… think about it!

Open Addressing: Linear Probing

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

Example of cluster

- **All 4 elements are part of a cluster**
- **Two of them have the same hash value:**
  - Kruse and Horowtiz (5)
- **The other two have different hash values (0 and 1)**
- **From then on, any element mapped to 0,1,5 or 6 will be part of the cluster. And adding elements mapped to 0,1,2,4,5, or 6 will make it grow.**

hash table

0	Aho
1	Standish
2	
3	
4	
5	Kruse
6	Horowitz

Linear Probing: Problems

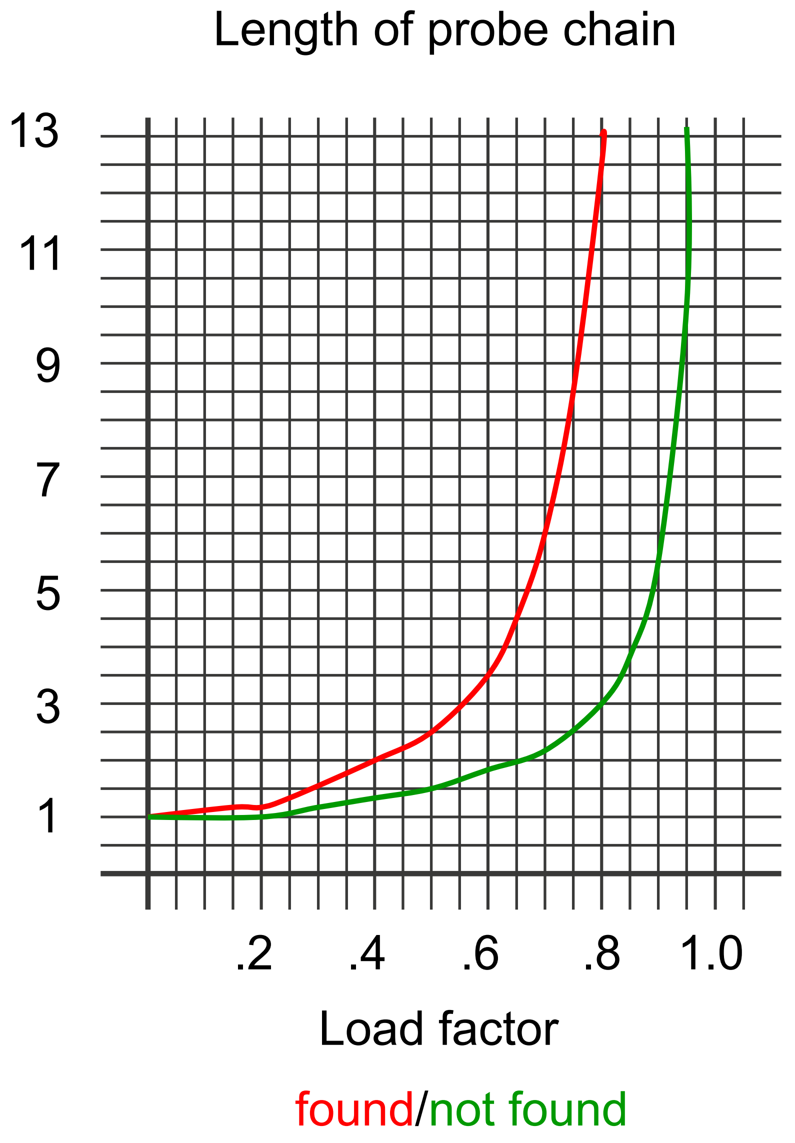
- **Tendency for clustering to occur as the load is > 0.5**
- **Low speed on clustering:**
  - Adding a key with hash value N can drastically increase the search time for keys with values other than N
  - Deletion can also be time consuming, as the entire cluster needs to be rehashed
  - This means we start to under-deliver on the O(1) promise
- **If implemented in arrays – table may become full fairly quickly, resizing is time and resource consuming**

What can we do?

Idea: reduce clustering by taking bigger and bigger steps (rather than always using +S)

Open Addressing

- **You must keep the load under 2/3**
  – Otherwise the probe length (i.e., number of items visited before the element is found/not found) is too high
- **Even better: under 1/2**



Conclusion

- **Hash Tables are one of the most used data types, as they have expected O(1) complexity for adding, deleting and searching, if *built properly***

As you've seen, hard to achieve in practice!

- **You have a very good chance of using them in your professional career**
- **They are very simple conceptually**
- **But they are also very "empirical":**
  - A significant amount of experimental evaluation is usually needed to fine tune the hash function and the TABLESIZE
- **A good choice of hash function, collision handling and load factor are crucial to maintaining an efficient hash table (i.e., keep the O(1) promise)**

Summary

- **Open Addressing**
  - Linear Probing
- **Advantages/Disadvantages**
  - Length of probe
  - Memory usage
  - Resizing