Associative Arrays
(Algorithms and Data Structures)

Associative Arrays

- associative arrays (maps or dictionaries) are **abstract data types**
- composed of a **collection of key-value pairs** where each key appears at most once in the collection
- most of the times we implement associative arrays with hashtables but binary search trees can be used as well
- the aim is to reach **O(1)** time complexity for most of the operations

Associative Arrays

- associative arrays (maps or dictionaries) are **abstract data types**
- composed of a **collection of key-value pairs** where each key appears at most once in the collection

EMAIL	USER
k.smith@gmai.com	User(,Kevin Smith', 34)
a.jobs@yahoo.com	User(,Ana Jobs', 26)
daniel@gmail.com	User(,Daniel Musk', 48)

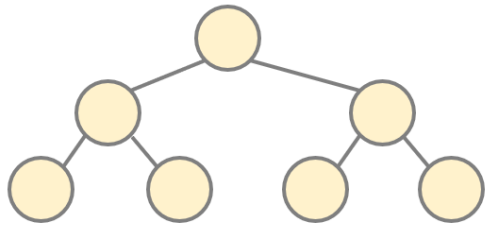
Associative Arrays

0	45
1	34
2	12
3	18
4	9
5	1
6	2
7	11

*finding an arbitrary item in an array takes **O(N)** linear running time*

BUT IT HAS O(1) RANDOM ACCESS

*we can do better with binary search trees with **O(logN)** logarithmic running times*



***AVL trees** and **red-black trees** can guarantee **O(logN)** running times*

*we can combine **random access** with **hash-functions** to end up with **O(1)** running times*

ASSOCIATIVE ARRAYS (!!!)

Associative Arrays

- there are several operations we want to implement – and we want these operations to have **O(1)** running time
- **adding** (key, value) pairs to the collection
- **removing** (key, value) pairs to the collection
- **lookup** a given value associtaed with a given key
- The key and value pairs a– this is why associative arrays do not support **sorting** as an operation

Hashtables
(Algorithms and Data Structures)

Hashtables

The motivation is that we want to store **(key,value)** pairs efficiently – so that the **insert** and **remove** operations takes **O(1)** running time

Hashtables

The motivation is that we want to store **(key,value)** pairs efficiently – so that the **insert** and **remove** operations takes **O(1)** running time

KEYS	VALUES
Goethe	**Faust**
Schiller	**Don Carlos**
Heidegger	**Being and time**

*we would like to store **authors** and the **titles** of their novels and make operations in **O(1)** running time complexity*

Hashtables

The motivation is that we want to store **(key,value)** pairs efficiently – so that the **insert** and **remove** operations takes **O(1)** running time

KEYS	VALUES
daniel@gmail.com	User(„Daniel”,24)
kevin@gmail.com	User(„Kevin”,34)
adam@gmail.com	User(„Adam”,56)

*we would like to store **authors** and the **titles** of their novels and make operations in **O(1)** running time complexity*

Hashtables

The motivation is that we want to store **(key,value)** pairs efficiently – so that the **insert** and **remove** operations takes **O(1)** running time

KEYS	VALUES
daniel@gmail.com	User(„Daniel”,24)
kevin@gmail.com	User(„Kevin”,34)
adam@gmail.com	User(„Adam”,56)

*we would like to store **authors** and the **titles** of their novels and make operations in **O(1)** running time complexity*

Hashtables

Hashtables

INSERT(,Kevin Smith', 34)

Hashtables

INSERT(,Kevin Smith', 34)

Hashtables

INSERT(,Kevin Smith', 34)

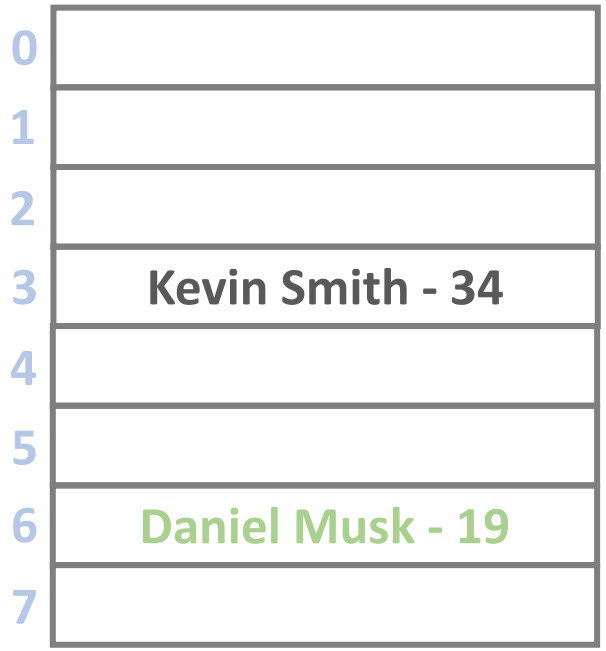
Hashtables

Hashtables

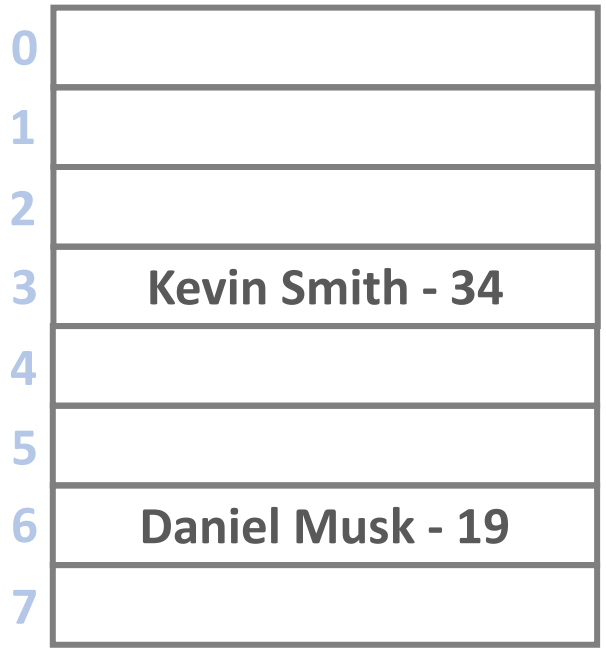
INSERT(,Daniel Musk', 19)

Hashtables

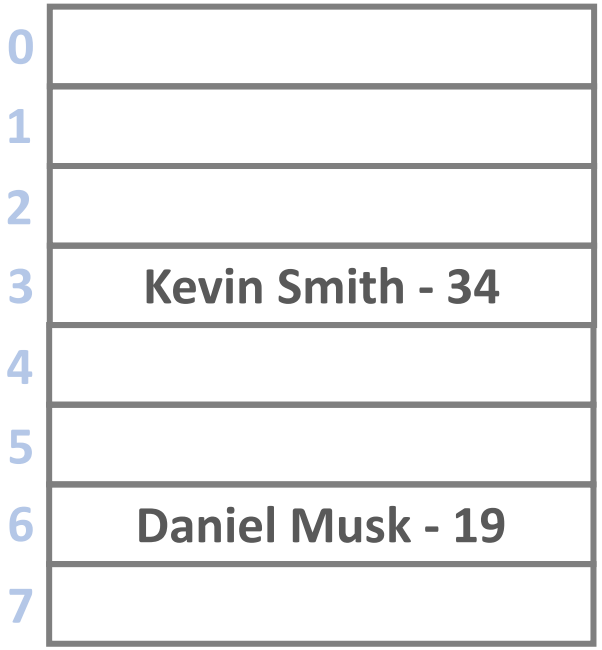
INSERT(,Daniel Musk', 19)



Hashtables



Hashtables

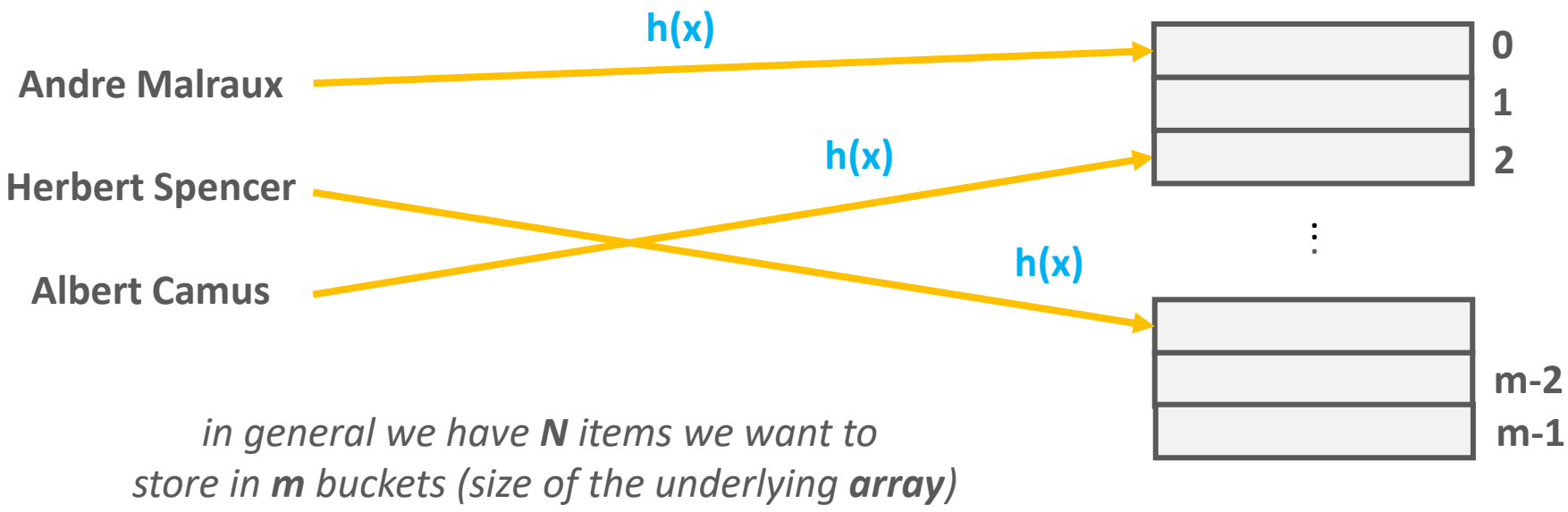


- how to achieve **O(1)** running times for insertion and removal operations?
- we should transform the key into an array index – to achieve **random access**
- this is why **keys must be unique** to avoid using the same indexes
- **h(x)** hash-function transforms the key into an index in the range **[0,m-1]**

Hashtables

*„The **h(x)** hash-function maps keys to array indexes in the array to be able to use **random indexing** and achieve **O(1)** running time”*

Hashtables



*in general we have **N** items we want to store in **m** buckets (size of the underlying **array**)*

THE h(x) HASH-FUNCTION DEFINES THE RELATIONSHIPS BETWEEN THE KEYS AND THE ARRAY INDEXES !!!

Hash-Functions

- the **h(x)** hash-function transforms the keys into array indexes
- it should handle **any types** – strings, floats, integers or even custom object as well
- if we have **integer keys** we just have to use the modulo (%) operator to transform the number into the range **[0,m-1]**
- we can use the **ASCII** values of the letters when dealing with strings

THE h(x) HASH-FUNCTION DISTRIBUTES THE KEYS

UNIFORMLY INTO BUCKETS (ARRAY SLOTS) !!!

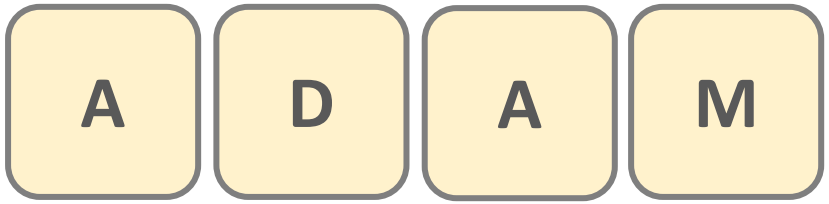
Hash-Functions

Hash-Functions

INSERT(,ADAM', 39)

Hash-Functions

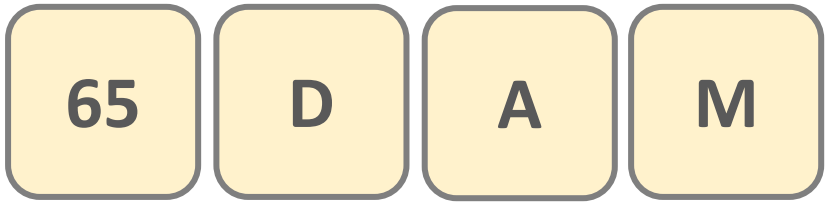
INSERT(‚ADAM’, 39)



*we an use the **ASCII** values*
for the characters to end up with
numerical representation

Hash-Functions

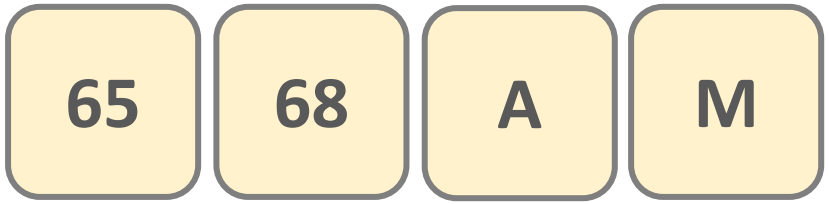
INSERT(‚ADAM’, 39)



*we an use the **ASCII** values*
for the characters to end up with
numerical representation

Hash-Functions

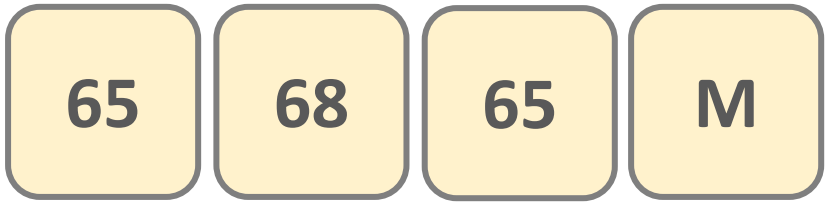
INSERT(‚ADAM’, 39)



*we an use the **ASCII** values*
for the characters to end up with
numerical representation

Hash-Functions

INSERT(‚ADAM’, 39)



*we an use the **ASCII** values*
for the characters to end up with
numerical representation

Hash-Functions

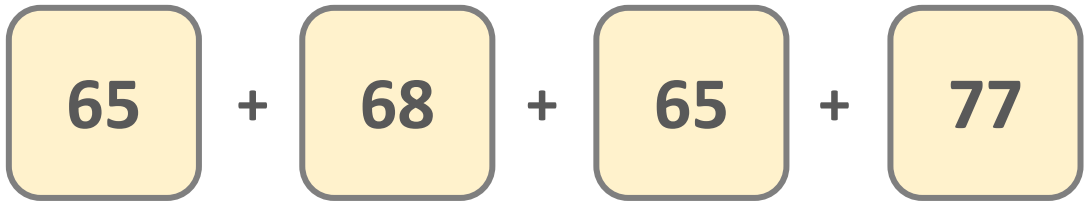
INSERT(‚ADAM’, 39)

65	68	65	77

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

Hash-Functions

INSERT(‚ADAM’, 39)



*we an use the **ASCII** values*
for the characters to end up with
numerical representation

Hash-Functions

INSERT(‚ADAM’, 39)

275

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

+

*we have to make sure the **index** is*
*in the range **[0,m-1]***

Hash-Functions

INSERT(‚ADAM’, 39)

275 **% 8**

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

+

*we have to make sure the **index** is*
*in the range **[0,m-1]***

Hash-Functions

INSERT(‚ADAM’, 39)

3

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

+

*we have to make sure the **index** is*
*in the range **[0,m-1]***

Hash-Functions

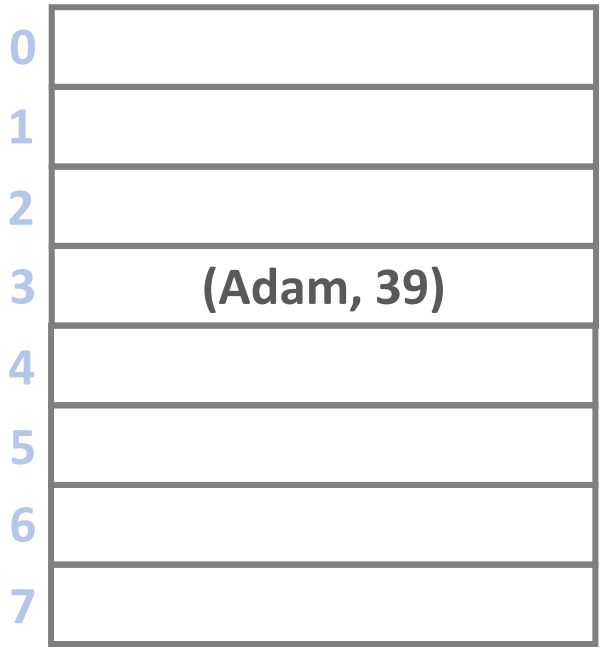
INSERT(‚ADAM’, 39)

3

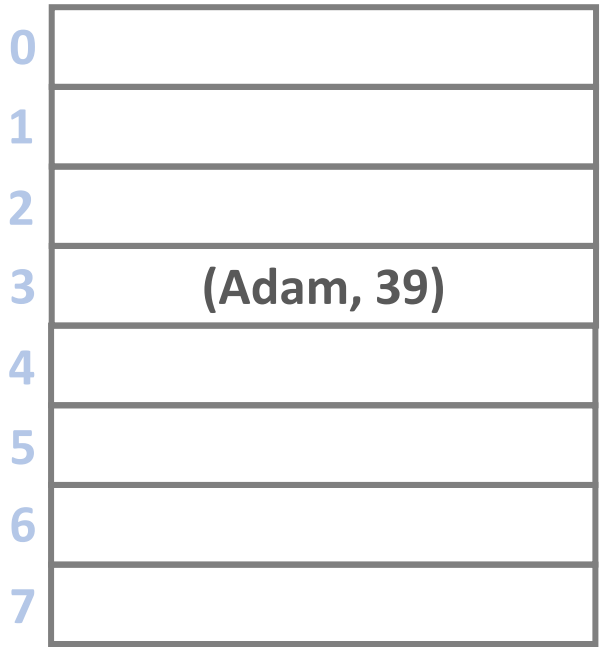
*we an use the **ASCII** values*
for the characters to end up with
numerical representation

+

*we have to make sure the **index** is*
*in the range **[0,m-1]***

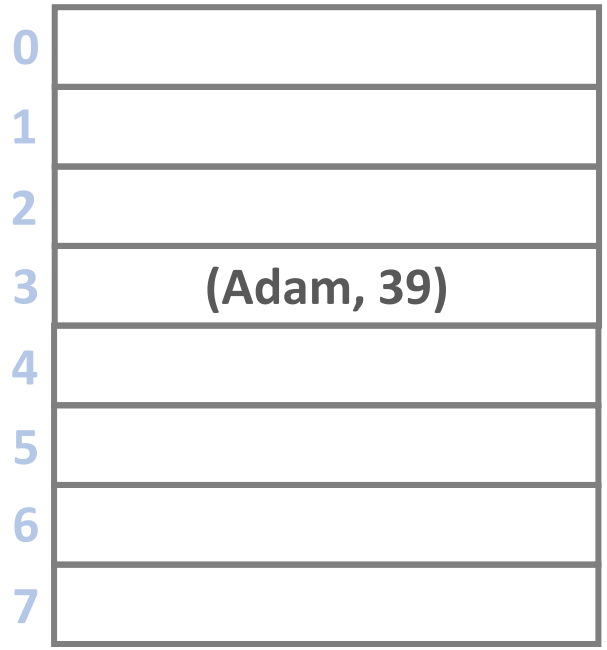


Hash-Functions



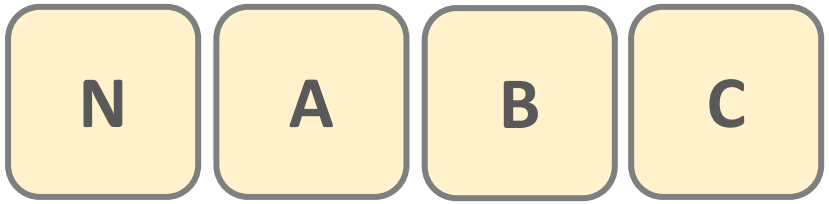
Hash-Functions

INSERT(‚NABC’, 21)

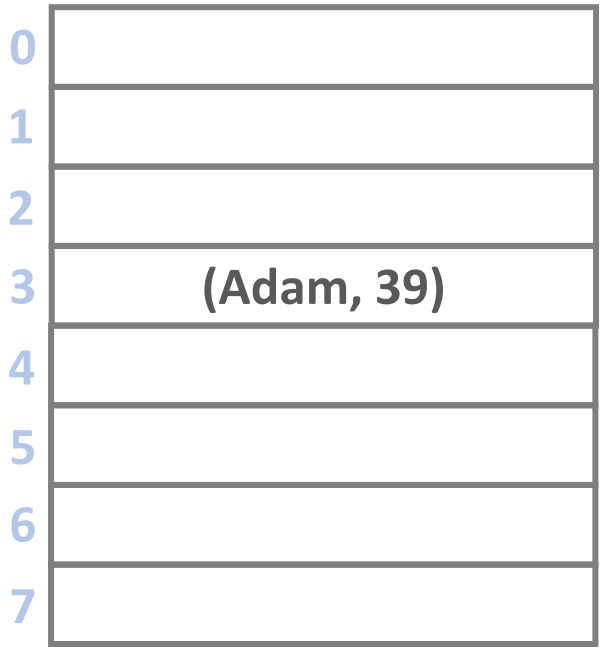


Hash-Functions

INSERT(‚NABC’, 21)

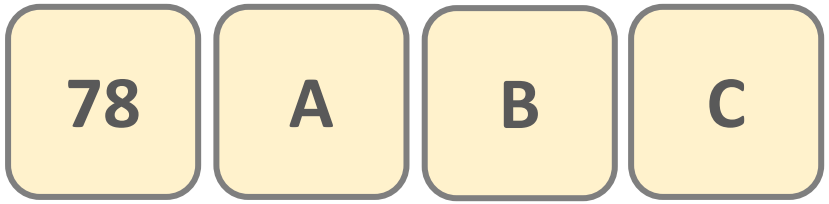


*we an use the **ASCII** values*
for the characters to end up with
numerical representation

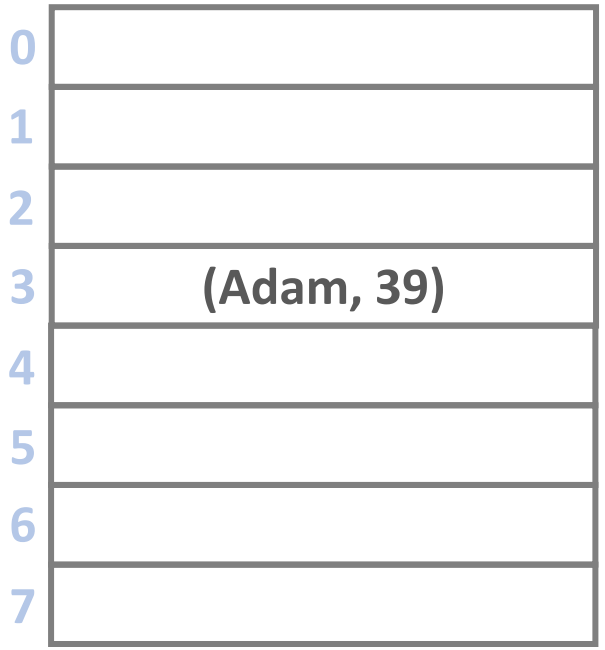


Hash-Functions

INSERT(,NABC', 21)

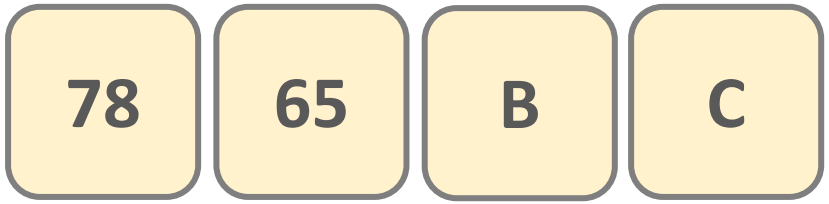


*we an use the **ASCII** values*
for the characters to end up with
numerical representation

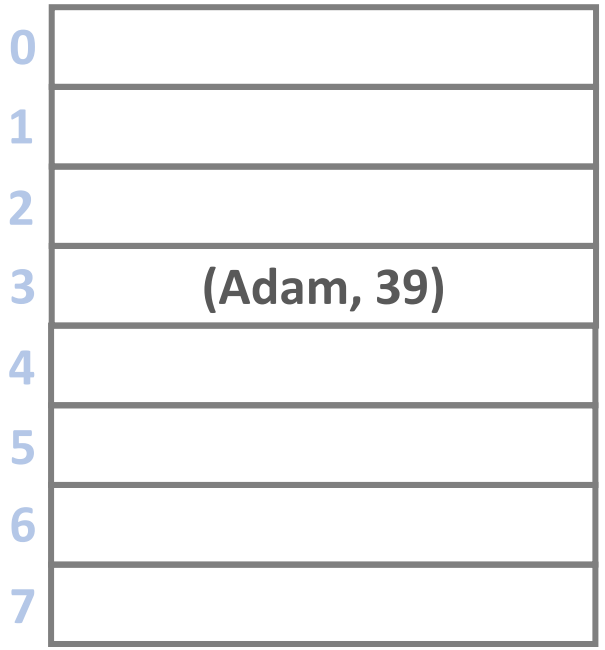


Hash-Functions

INSERT(‚NABC’, 21)

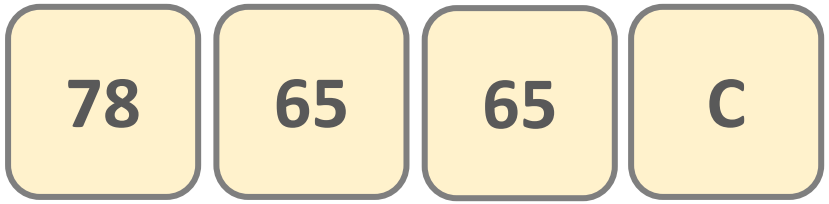


*we an use the **ASCII** values*
for the characters to end up with
numerical representation

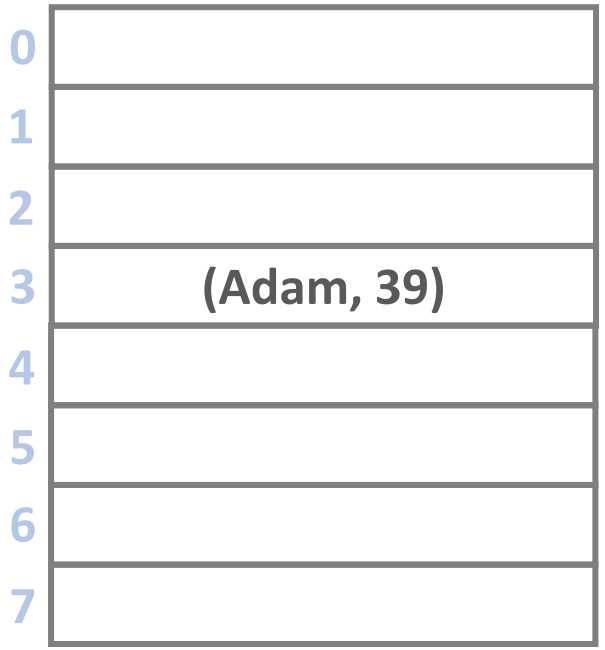


Hash-Functions

INSERT(‚NABC’, 21)



*we an use the **ASCII** values*
for the characters to end up with
numerical representation

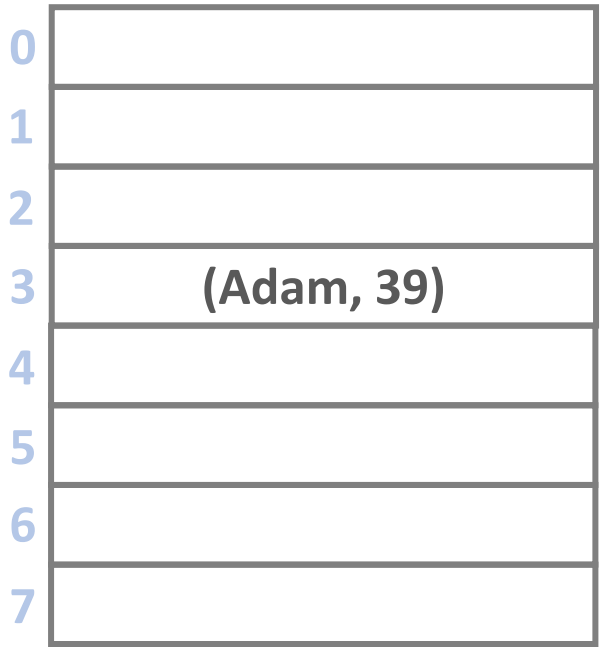


Hash-Functions

INSERT(‚NABC’, 21)

78	65	65	67

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

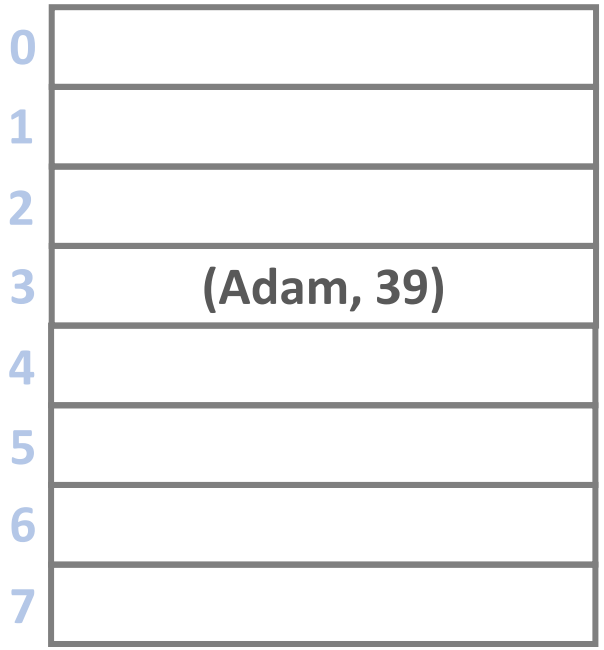


Hash-Functions

INSERT(‚NABC’, 21)

78 + 65 + 65 + 67

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

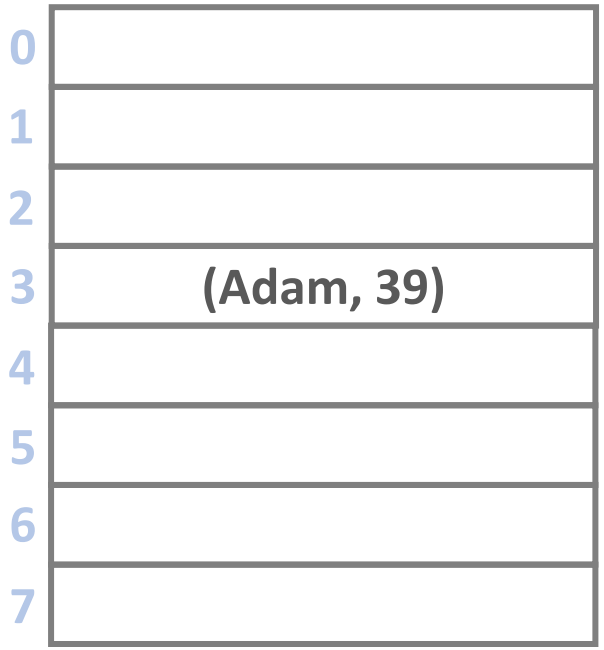


Hash-Functions

INSERT(,NABC', 21)

275

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

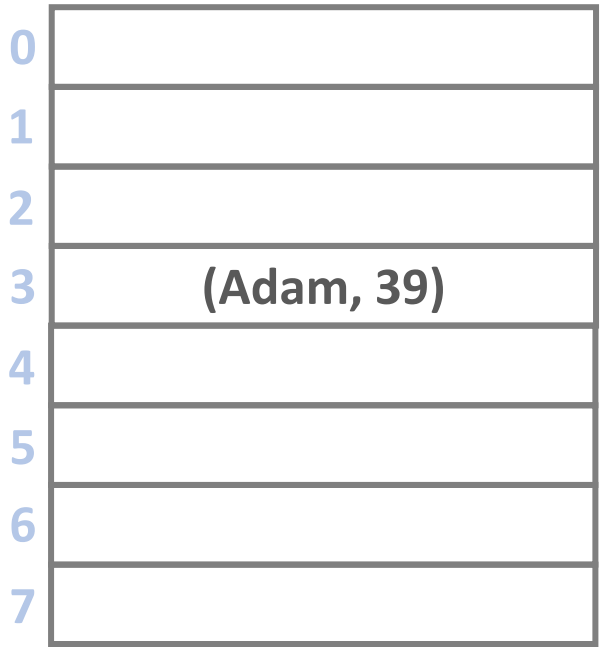


Hash-Functions

INSERT(‚NABC’, 21)

275 **% 8**

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

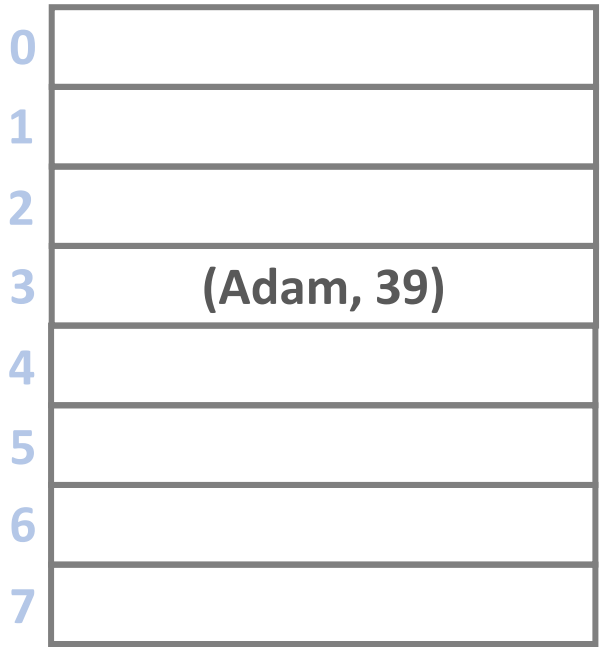


Hash-Functions

INSERT(‚NABC’, 21)

3

*we an use the **ASCII** values*
for the characters to end up with
numerical representation



Hash-Functions

INSERT(‚NABC’, 21)

*we an use the **ASCII** values*
for the characters to end up with
numerical representation

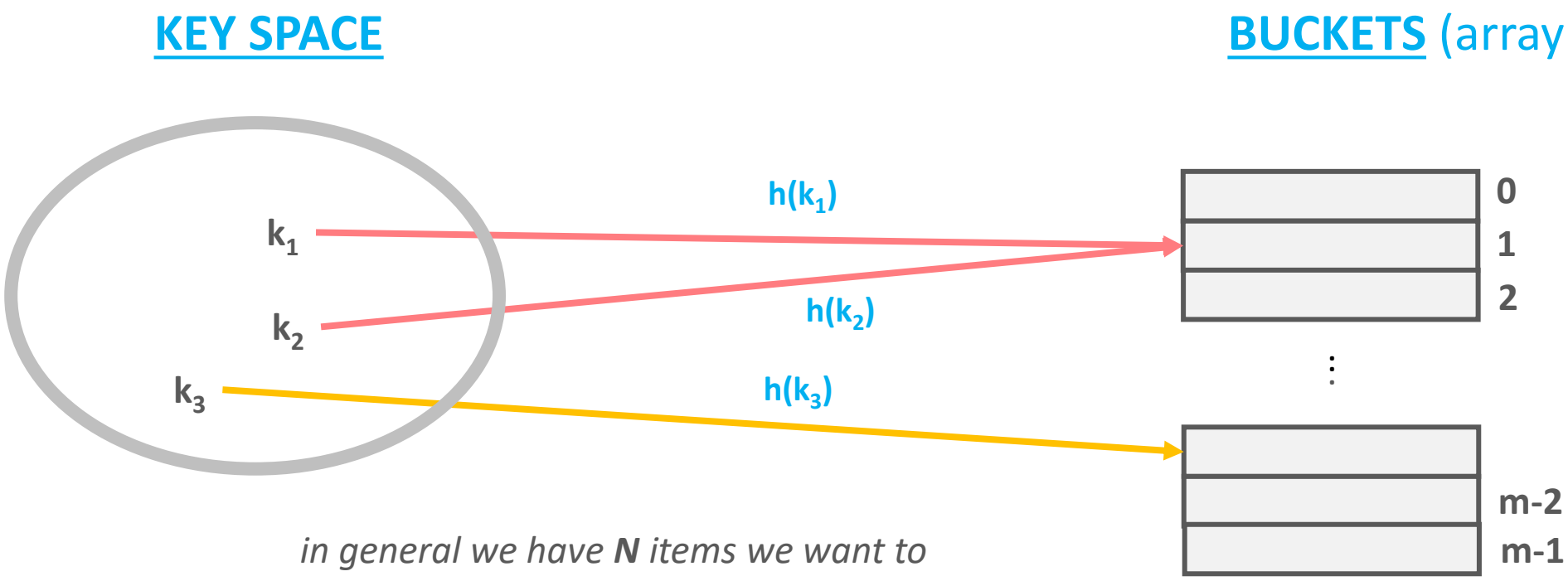
0	
1	
2	
3	(Adam, 39)
4	
5	
6	
7	

Collisions
(Algorithms and Data Structures)

Hashtable Collisions

*„**Collisions** occur when the **h(x)** hash-function maps two keys to the same array slot (bucket)”*

Hashtable Collisions



*in general we have **N** items we want to store in **m** buckets (size of the underlying **array**)*

IF THE h(x) HASH-FUNCTION IS PERFECT THEN THERE ARE NO COLLISIONS FOR SURE !!!

Hashtable Collisions

- the **h(x)** hash-function defines the reltionships between the keys and the array indexes (buckets)
- if the hash-function is perfect then there **are no collisions**
- in real-world **there will be collisions** becase there are no perfect hash-functions

Hashtable Collisions

There are several approaches to deal with collisions:

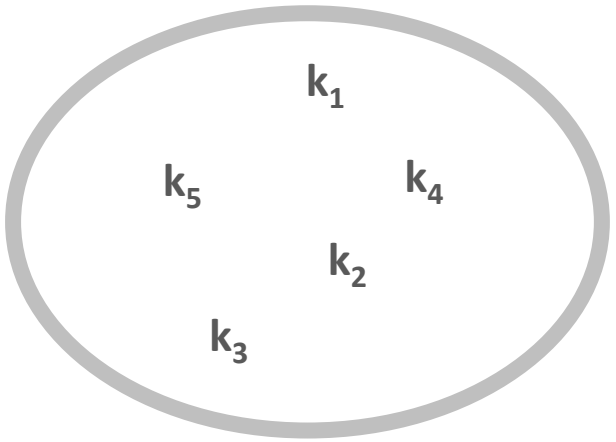
1.) CHAINING

2.) OPEN ADDRESSING

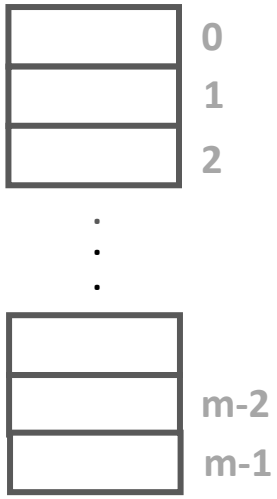
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure

KEY SPACE

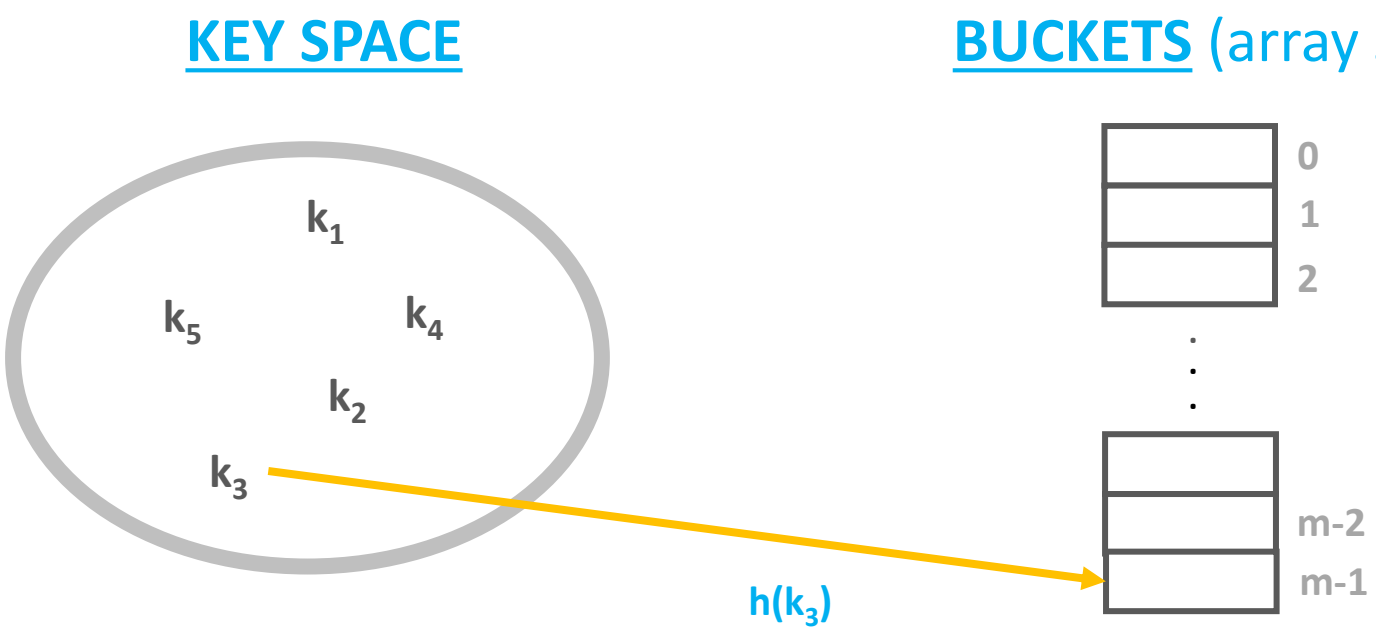


BUCKETS (array slots)



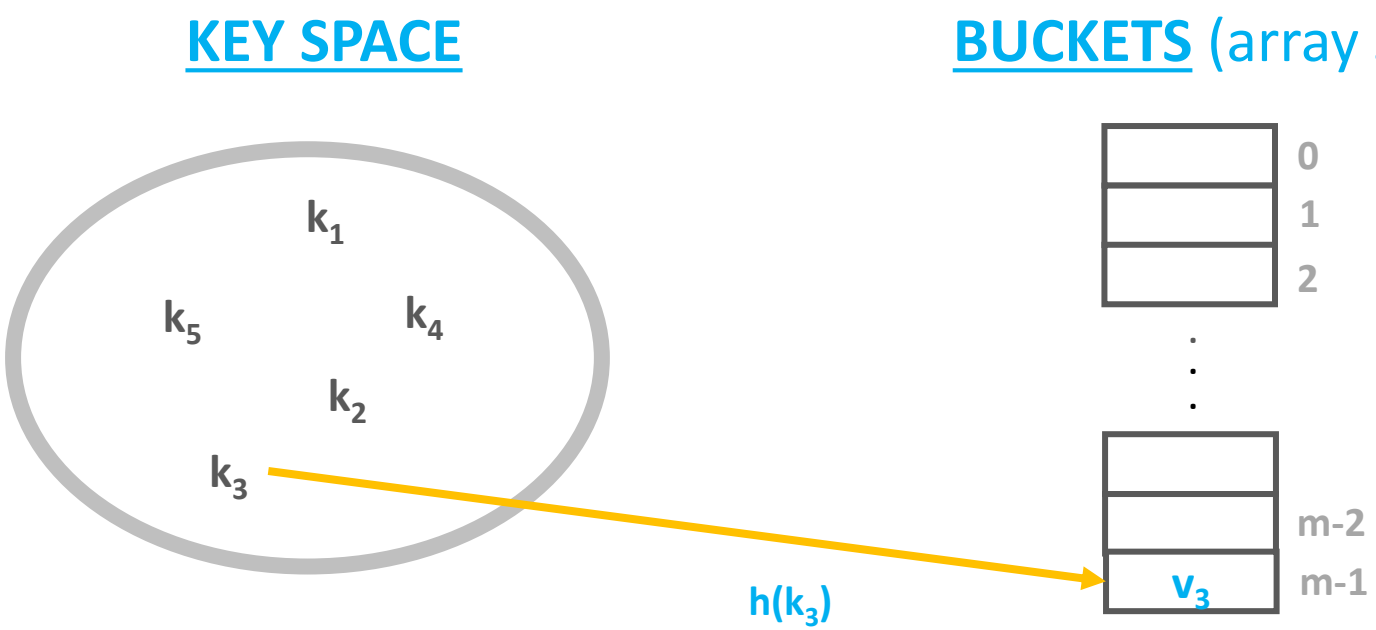
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



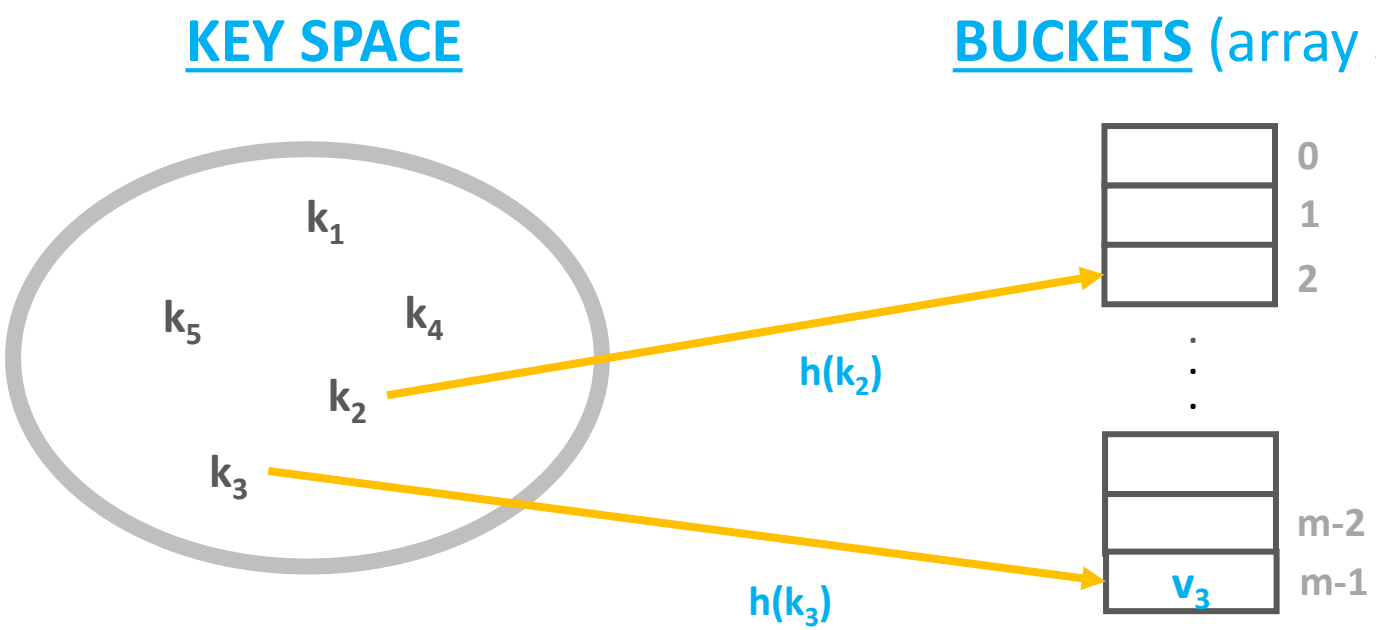
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



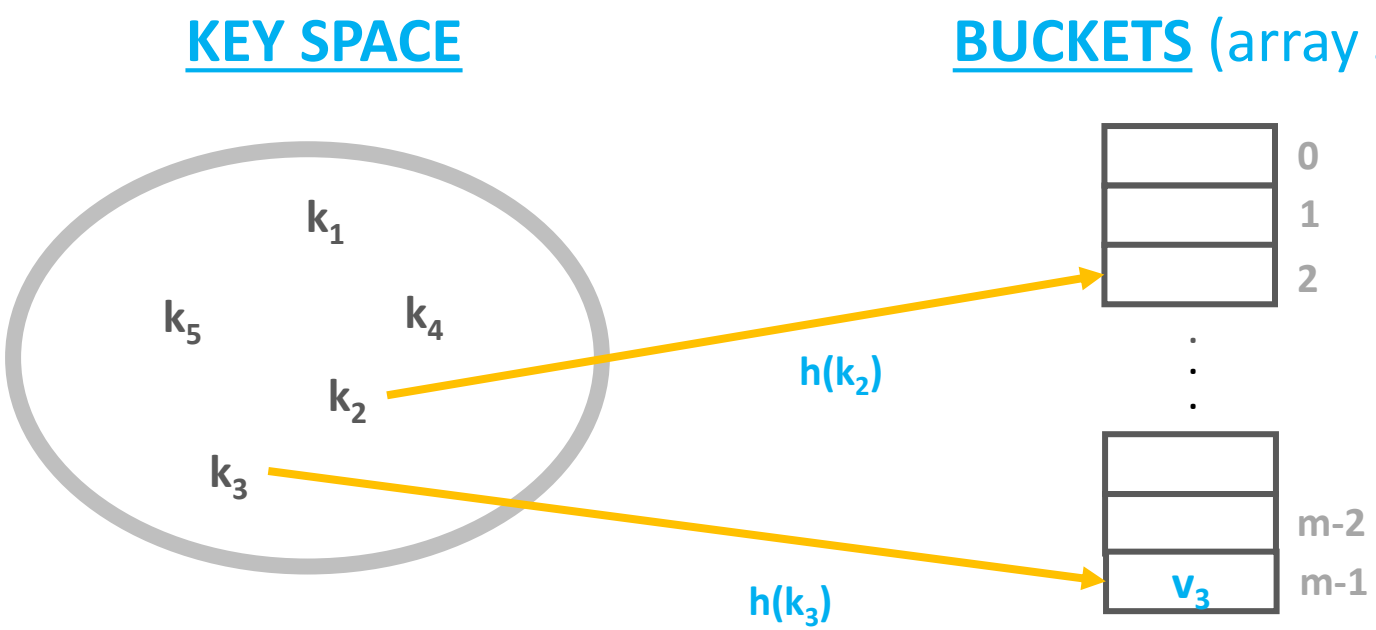
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



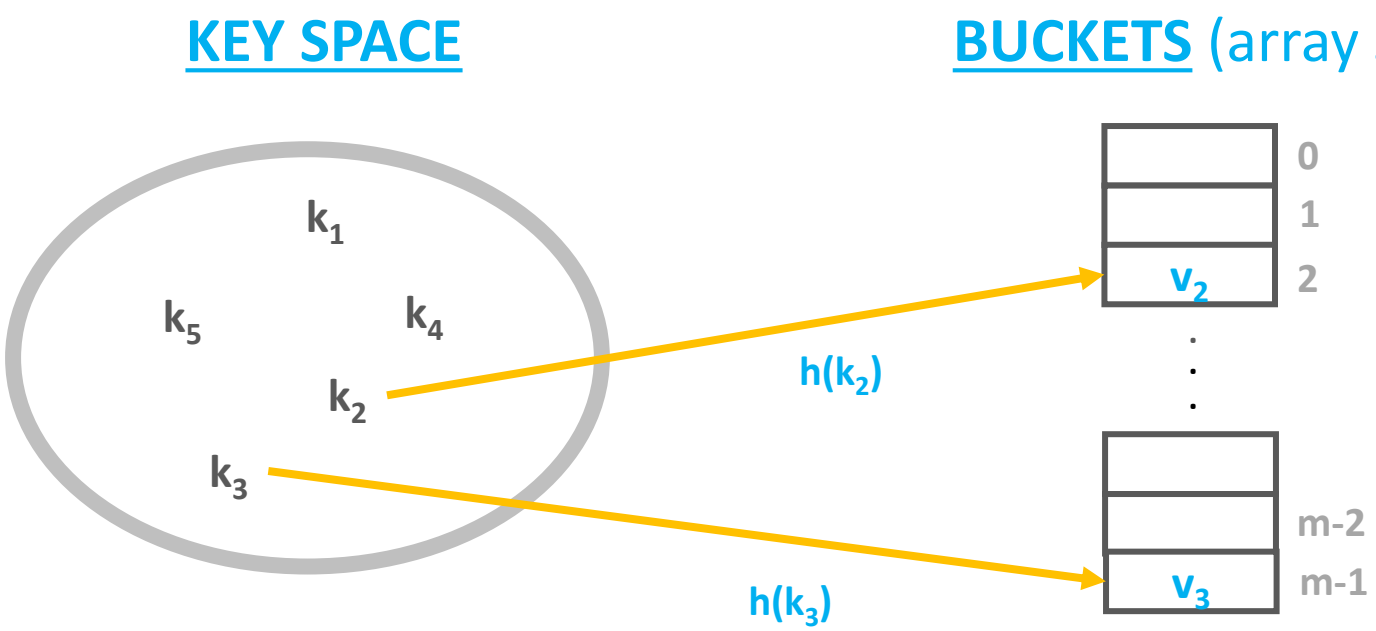
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



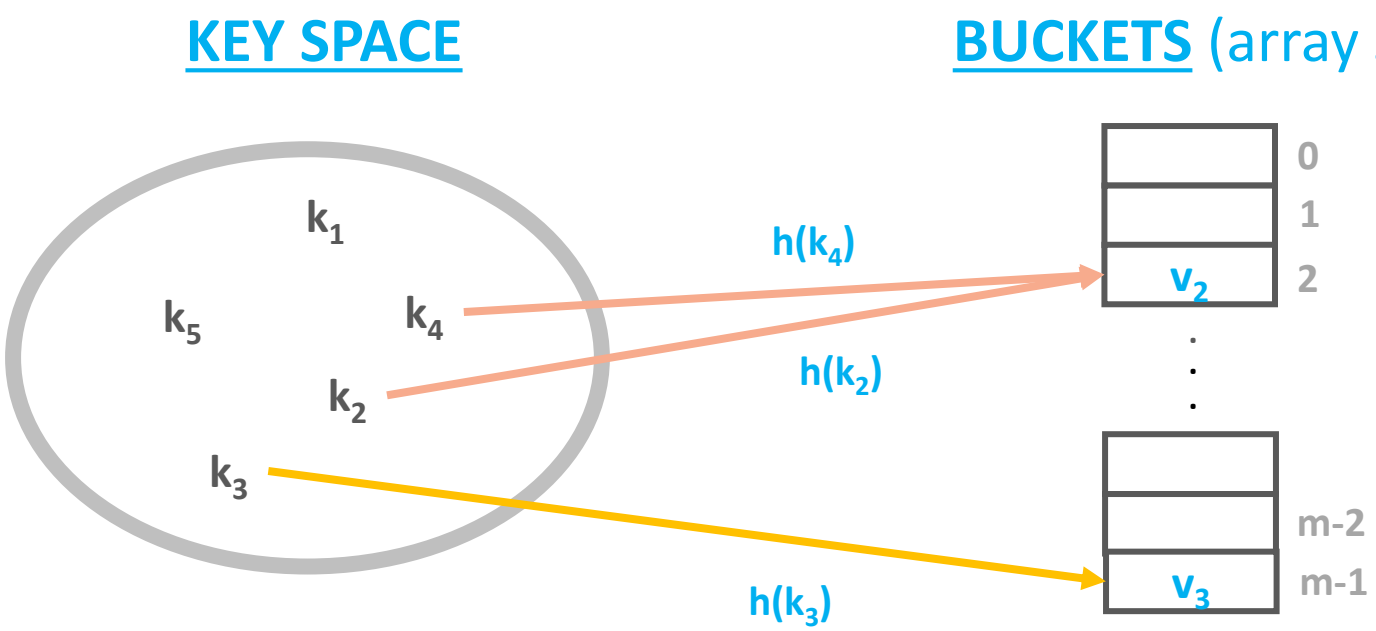
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



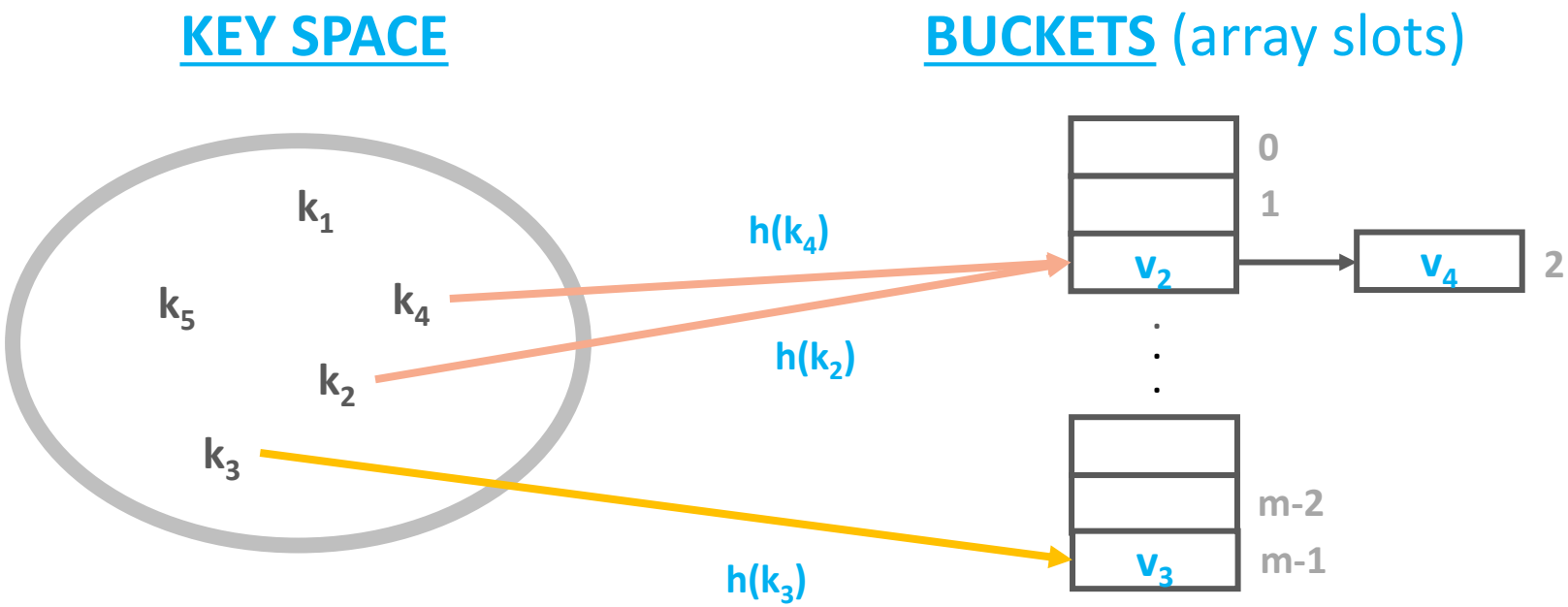
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



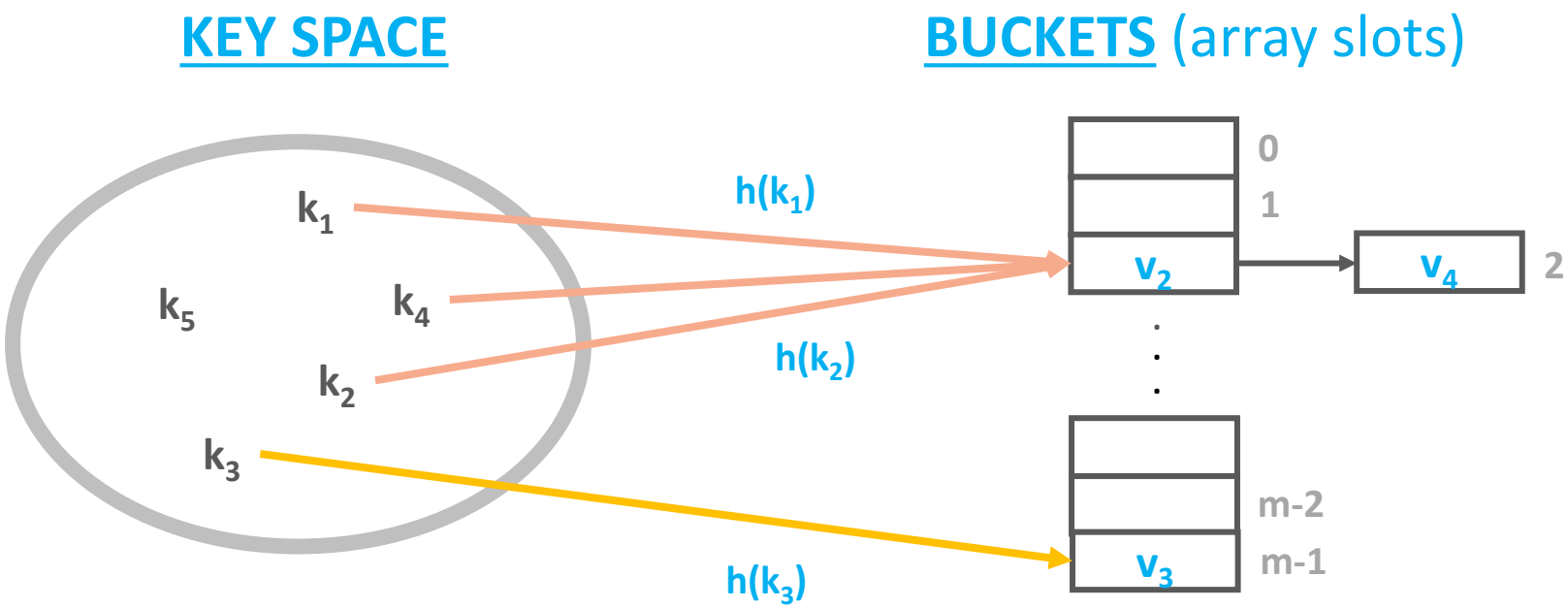
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



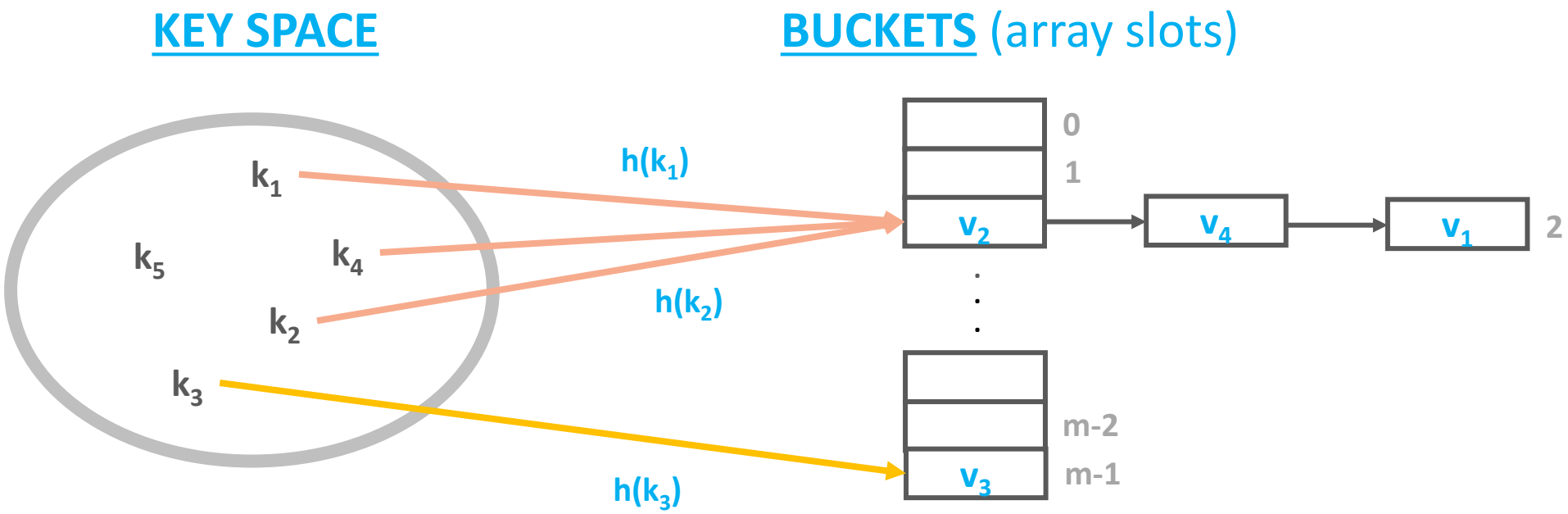
Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure



Hashtable Collisions

1.) **CHAINING:** we store the items in the same bucket (with same indexes) in a **linked list** data structure

→ in worst-case scenario the **h(x)** hash-function puts all the items into the same bucket (array slot)

→ we end up with a linked list with **O(N)** linear runnin time for most of the operations

Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)

Linear probing: if collision happened at array index **k** then

we try index **k+1**, **k+2, k+3** ... until we find an empty bucket

→ not always the best option possible because there will be **clusters** in the underlying array

→ but it has better **cache performance** than other approaches

Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)

Quadratic probing: if collision happened at array index **k** then

we try adding successive values of an arbitrary **quadratic polynomial**

(array slots **1**, **4**, **9**, **16** ... steps aways from the collision)

→ ther will be no clusters (unlike linear probing)

→ but no cache advantage (items are far away in memory)

Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)

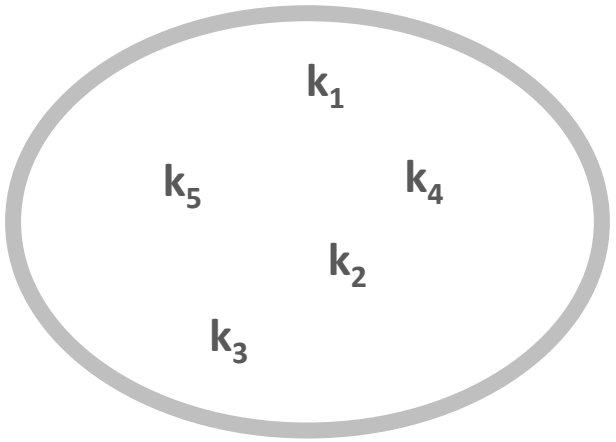
Rehasing: if collision happened at array index **k** then

we use the **h(x)** hash-function again to generate a new index

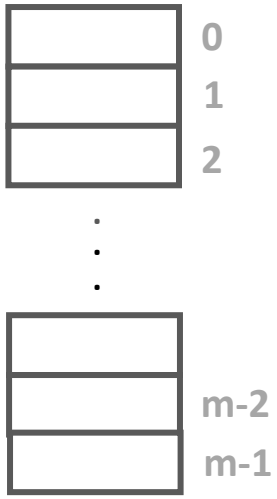
Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)

KEY SPACE

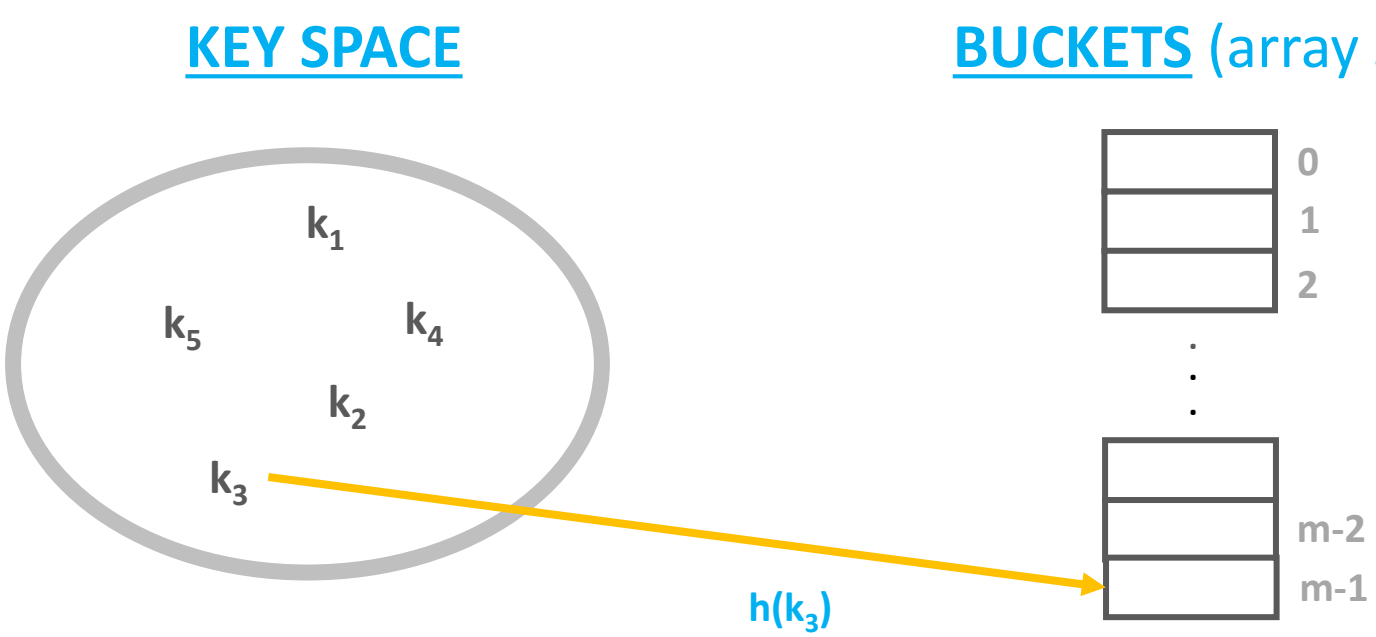


BUCKETS (array slots)



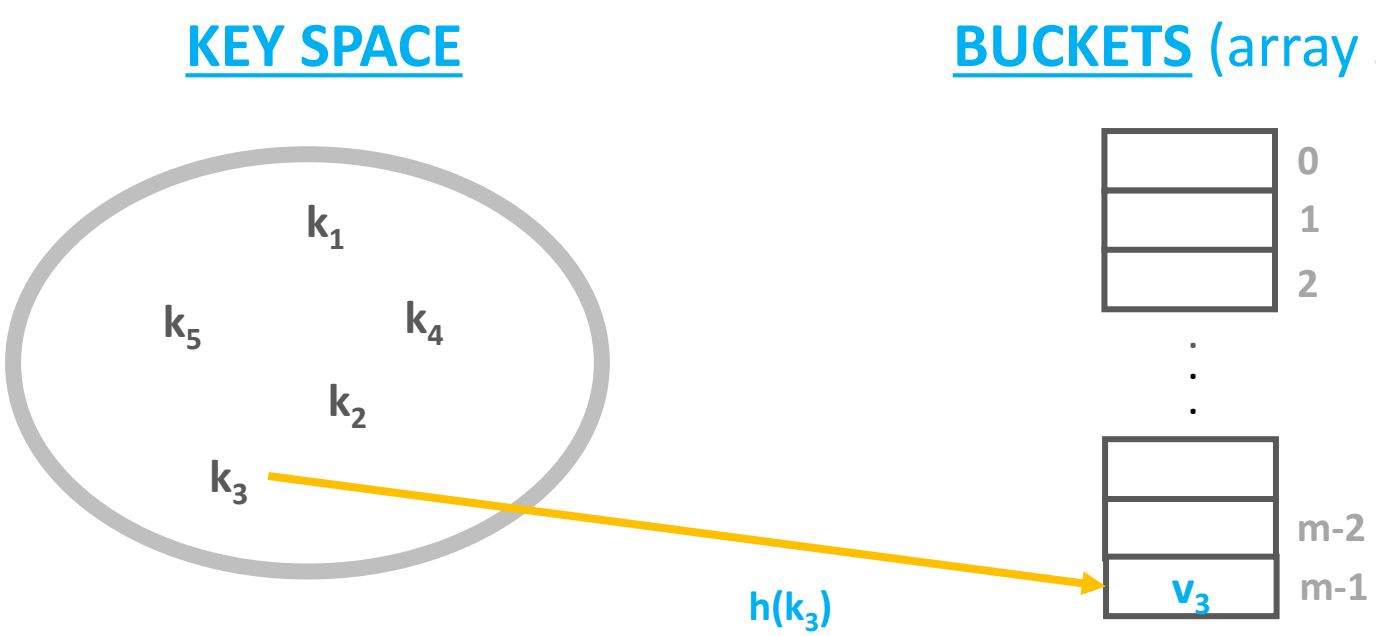
Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)



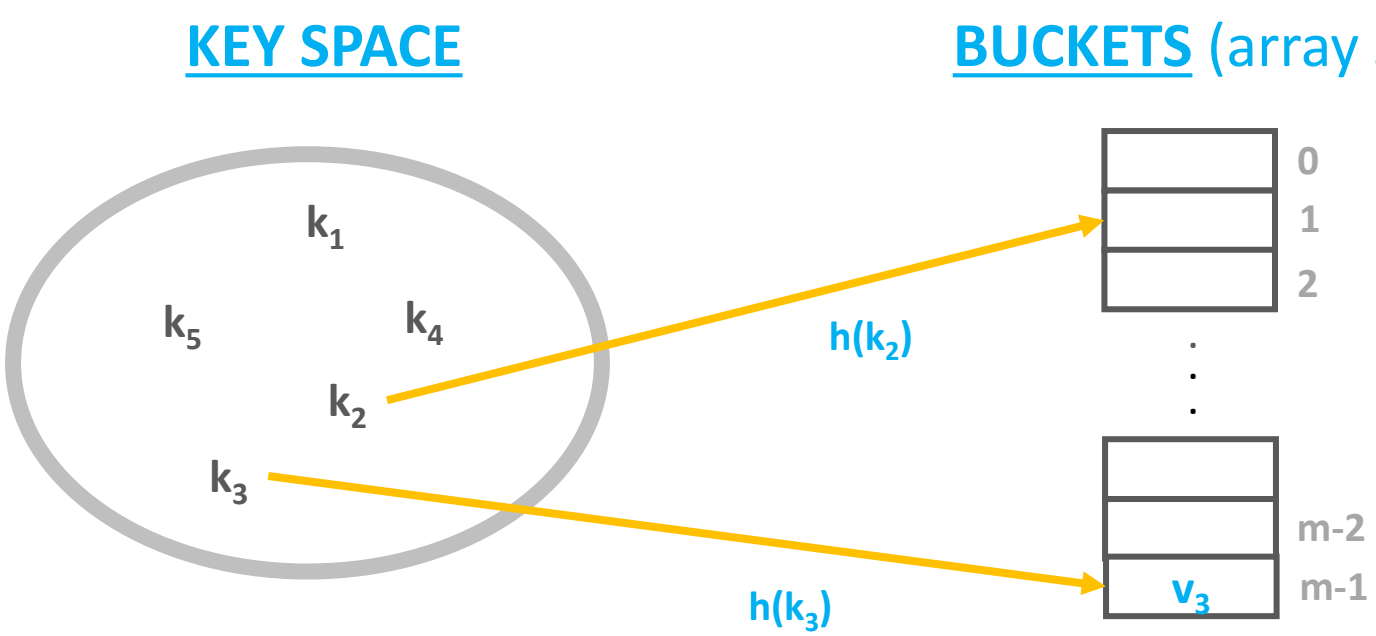
Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)



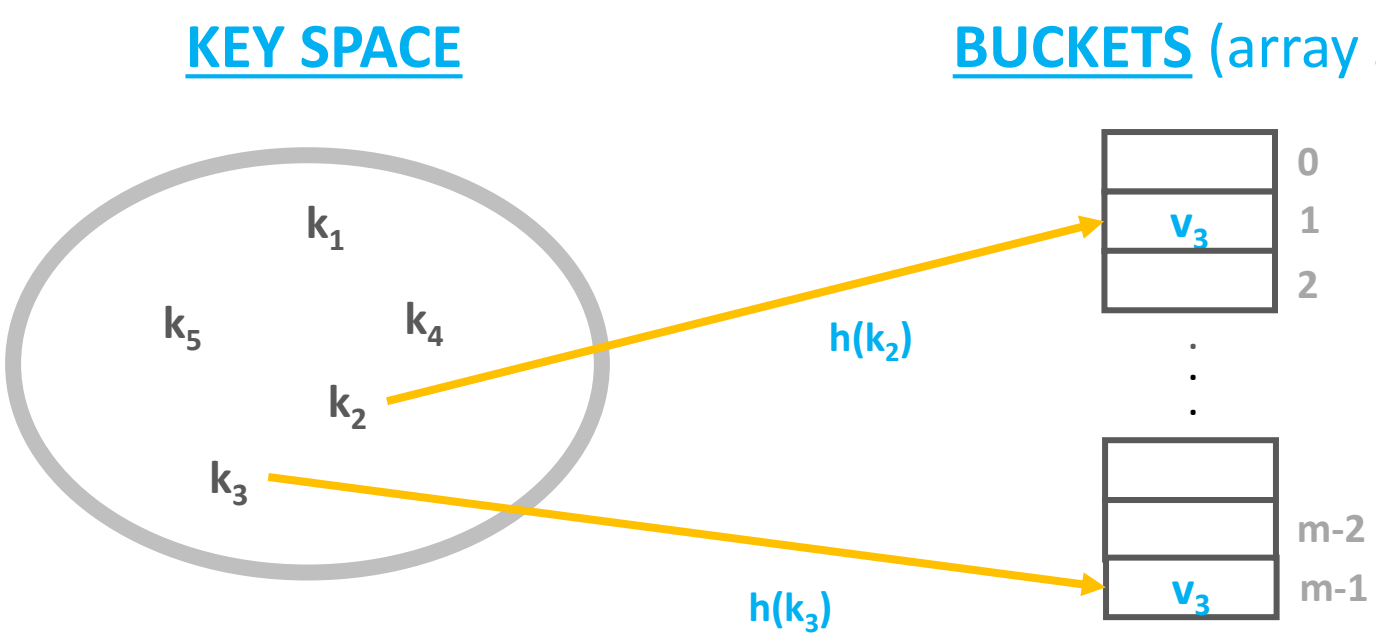
Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)



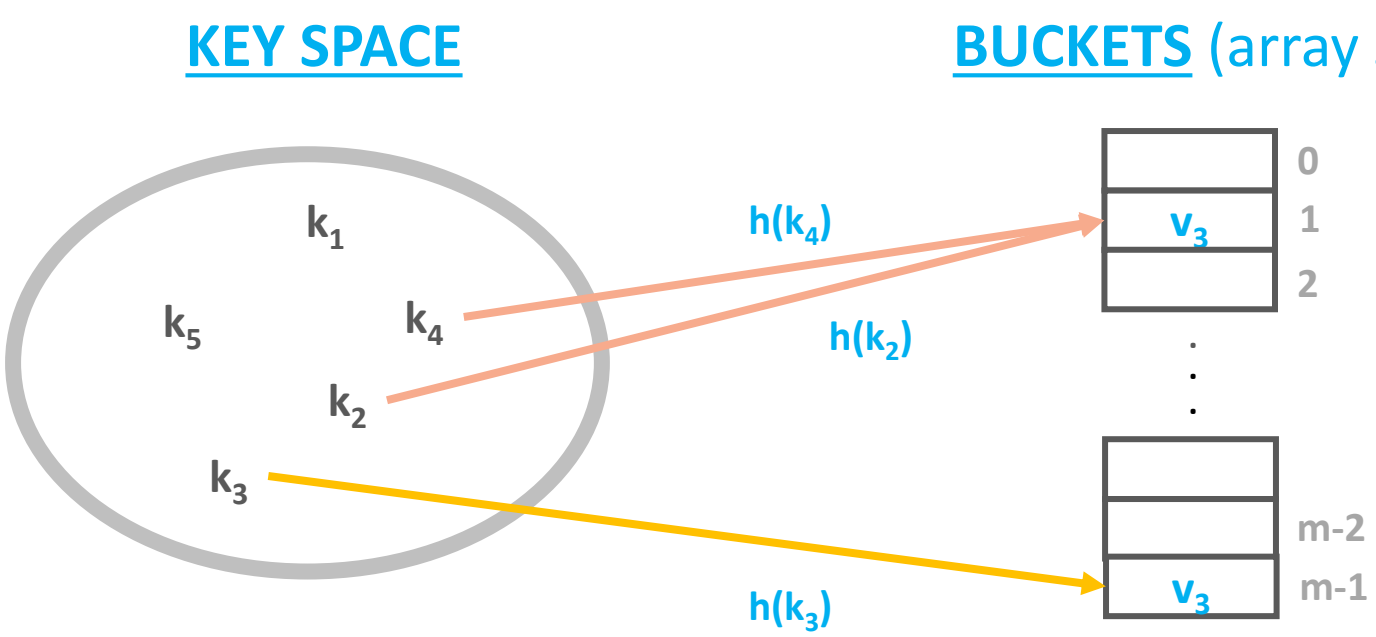
Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)



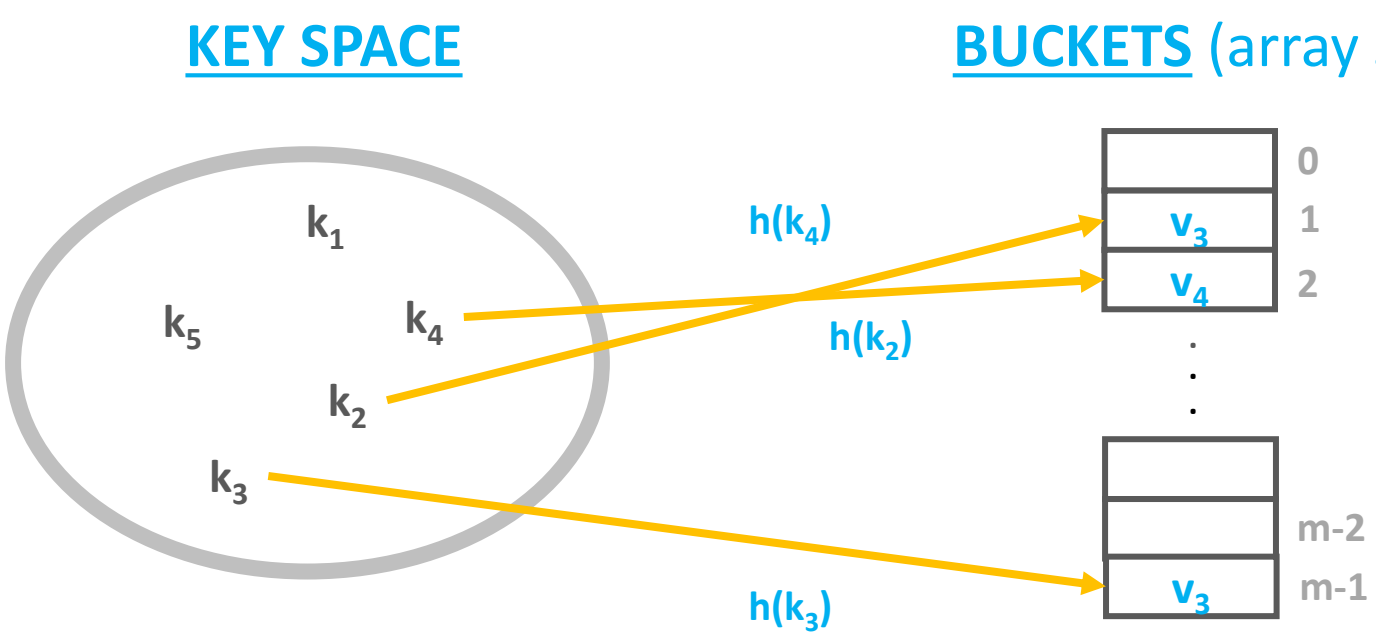
Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)



Hashtable Collisions

2.) **OPEN ADDRESSING:** if we come to the conclusion that there is a collision then we generate a new index for the item (try to find another bucket)



Hashtable Collisions

	AVERAGE-CASE	WORST-CASE
memory complexity	O(N)	O(N)
search	O(1)	O(N)
insertion	O(1)	O(N)
deletion	O(1)	O(N)

Dynamic Resizing
(Algorithms and Data Structures)

Load Factor

- the **p(x)** probability of collision is not constant
- the more items are there in the hashtable the higher the p(x) **probability of collision**
- this is why we have to define a new parameter of the hashtable – the so-called **load factor**

Load Factor

$$\frac{n}{m}$$

***n** is the number of actual items in the array data structure and **m** is the size of the array*

DEFINES A TYPICAL MEMORY AND RUNNING TIME TRADE-OFF

SMALL LOAD FACTOR (around 0)

- the hashtable is **nearly empty** which means low **p(x)** probabiliy of collisions
- but of course a lot of memory is wasted

HIGH LOAD FACTOR (around 1)

- the hashtable is **nearly full** which means high **p(x)** probability of collisions
- no memory is wasted but the running time may be reduced to **O(N)** linear running time

Load Factor and Dynamic Resizing

- the **p(x)** probability of collision is not constant
- the more items are there in the hashtable the higher the **p(x) probability of collision**
- this is why we have to define a new parameter of the hashtable – the so-called **load factor**
- **SOMETIMES WE HAVE TO RESIZE THE HASHTABLE**

Load Factor and Dynamic Resizing

Performance relies heavily on the **load factor**. Sometimes it is better to use memory to achieve faster running times.

→ when the load factor is **> 0.75** then **Java** resize the hashtable automatically to avoid too many collisions

→ **Python** does the same when the load factor **> 0.66**

Dynamic Resizing

- so sometimes it is better to resize and change the size of the underlying **array data structure**
- but the problem is that the **hash values are depending on the size** of the underlying array data structure
- so we have to consider all the items in the old hashtable and insert them into the new one with the **h(x)** hash-function
- it takes **O(N)** linear running time - this fact may make dynamic-sized hash tables inappropriate for real-time applications