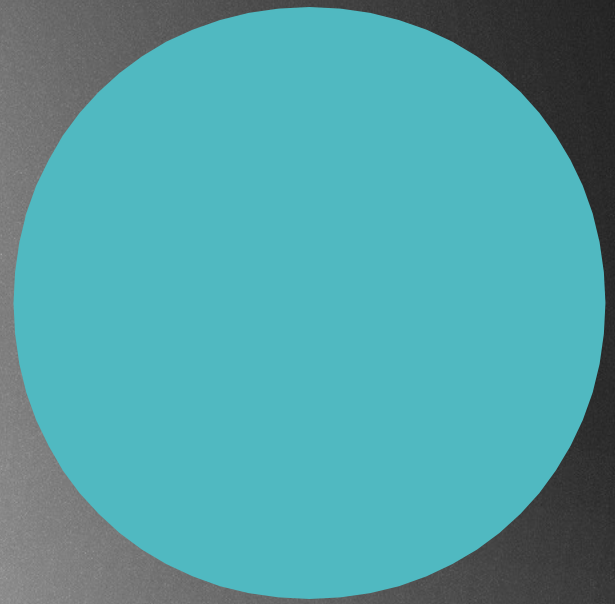
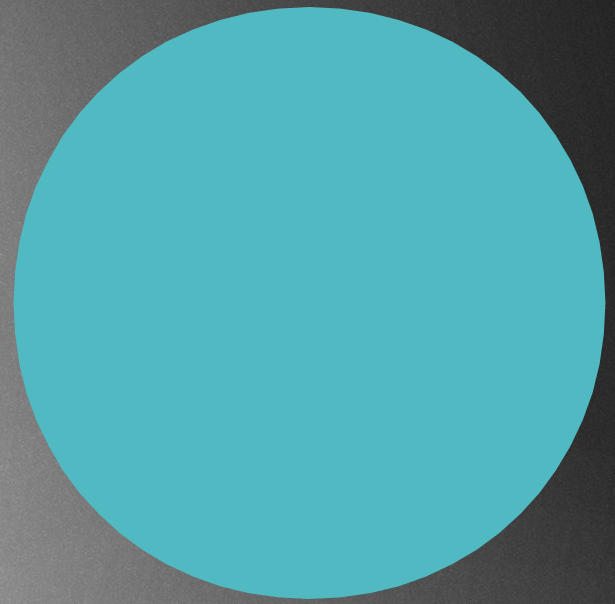
ASSOCIATIVE ARRAYS



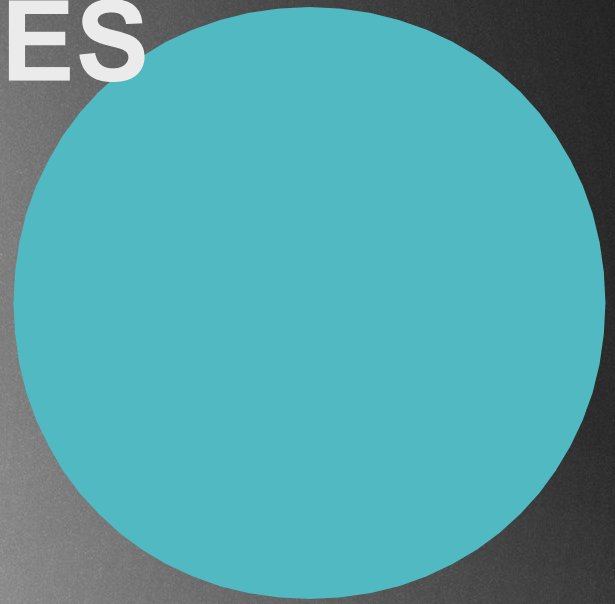
ADT

- Associative arrays / maps / dictionaries are abstract data types !!!
- Composed of a collection of key-value pairs where each key appears only 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

- **Supported operations**:
  - Adding key-value pairs to the collection
  - Removing key-value pairs from the collection
  - Update existing key-value pairs
  - Lookup of value associated with a given key

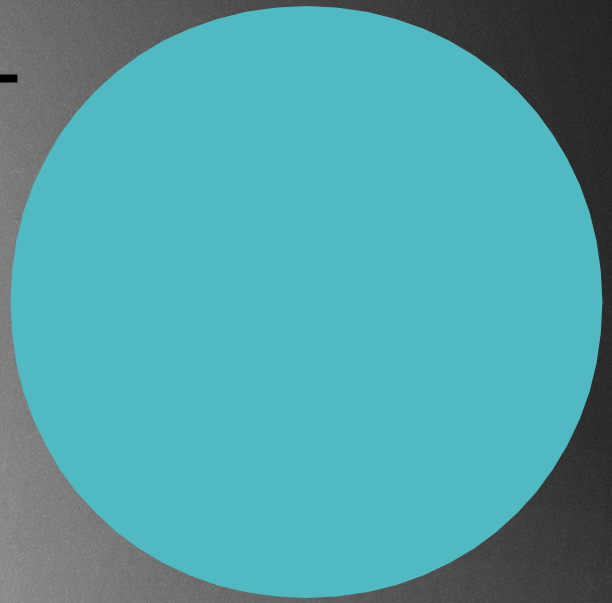


HASH TABLES / DICTIONARIES



Motivation:

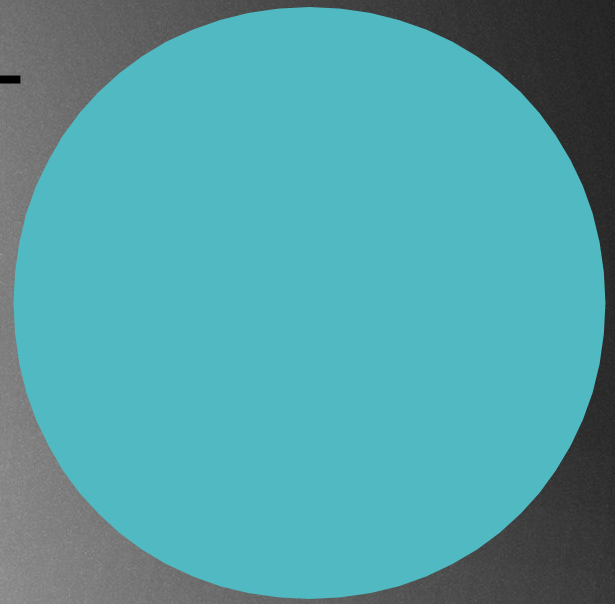
Keys	Values
Goethe	Faust
Schiller	Don Carlos
Heidegger	Being and time



We would like to store authors and the titles of their novels
So we have keys (authors) and values (titles)

Motivation:

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 users and the keys could be their email address. The aim would be to insert / retreive users according to their email address

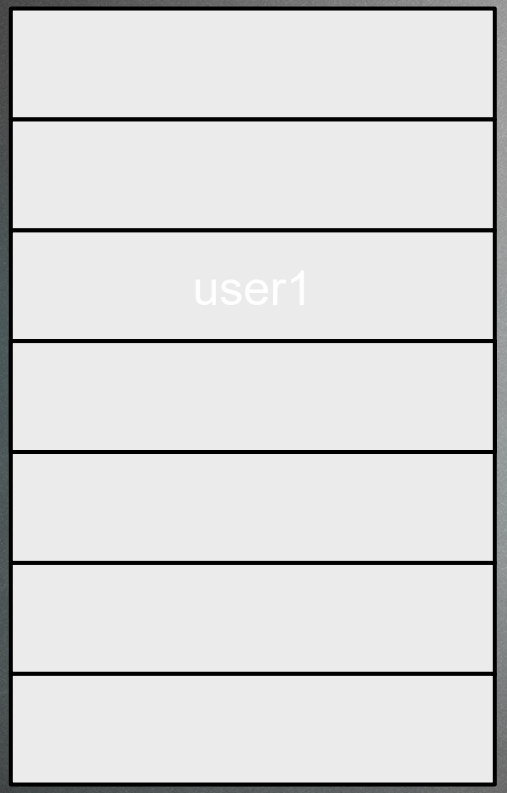
Motivation:

Arrays are just like that: if we know the index, the insert / retreive operations can be done in O(1) time

insert(2,user1)

Motivation:

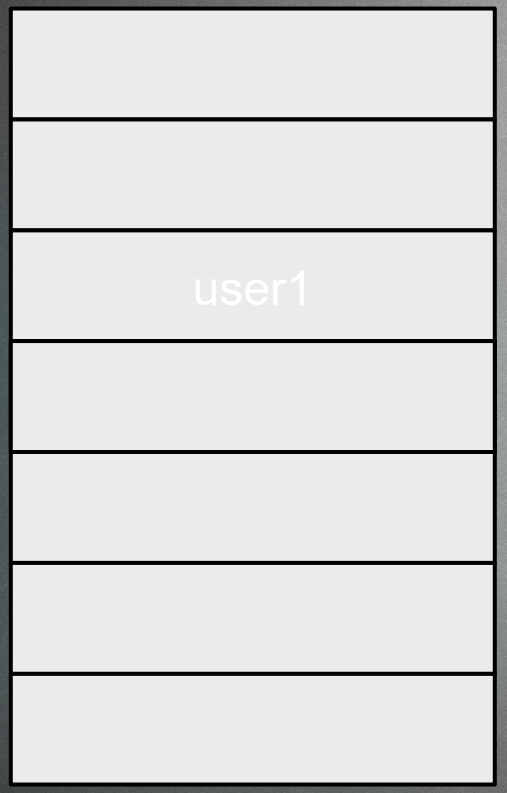
Arrays are just like that: if we know the index, the insert / retreive operations can be done in O(1) time



insert(2,user1)

Motivation:

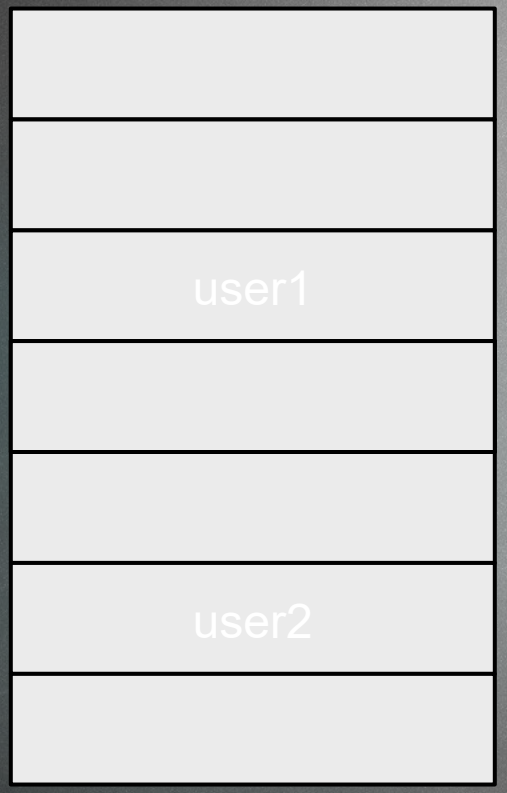
Arrays are just like that: if we know the index, the insert / retreive operations can be done in O(1) time



insert(5,user2)

Motivation:

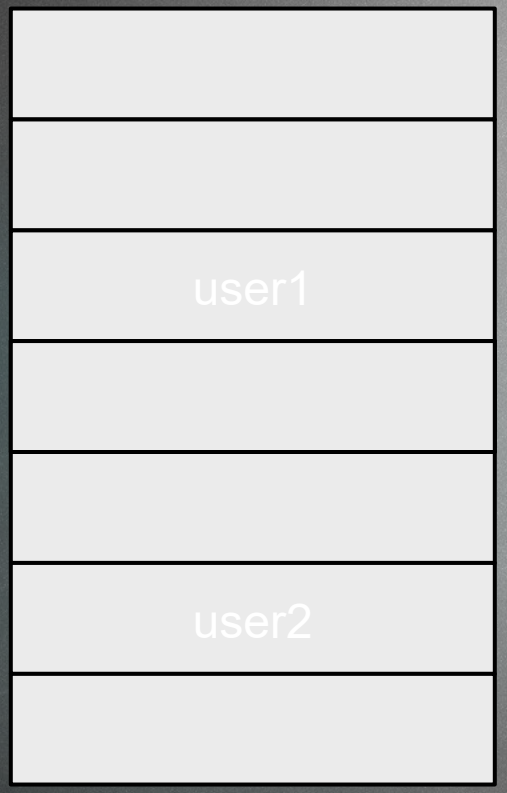
Arrays are just like that: if we know the index, the insert / retreive operations can be done in O(1) time



insert(5,user2)

Motivation:

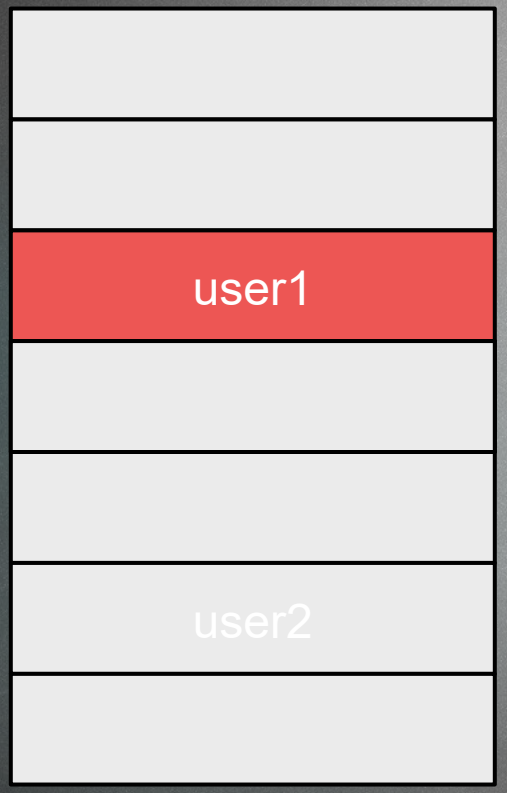
Arrays are just like that: if we know the index, the insert / retreive operations can be done in O(1) time



get(2)

Motivation:

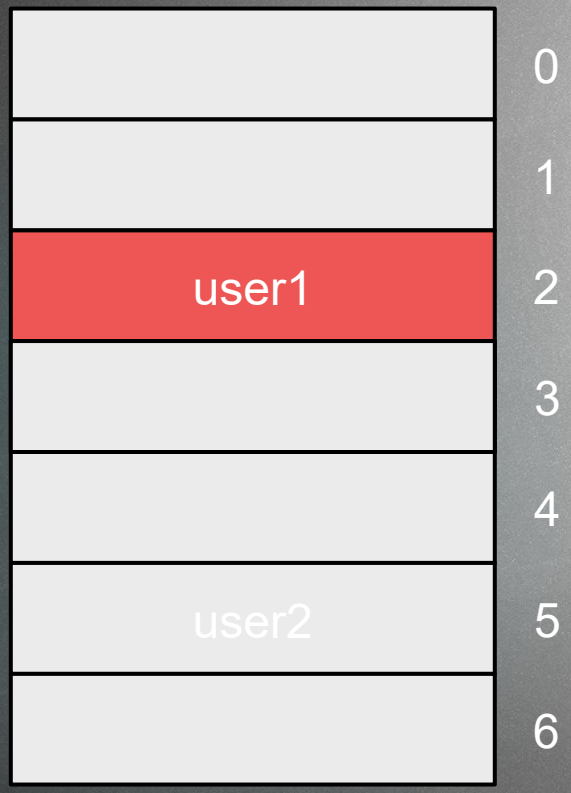
Arrays are just like that: if we know the index, the insert / retreive operations can be done in O(1) time



get(2)

Motivation:

Arrays are just like that: if we know the index, the insert / retreive operations can be done in O(1) time



get(2)

So arrays are going to solve our problem: the operations running time can be reduced to O(1)

PROBLEM: we must transform the keys into array indexes !!!
// this is why hashfunctions came to be

- Balanced BST → we can achieve **O(logN)** time complexity for several operations including search
- Can we do better?
- Yes, maybe we can reach **O(1)** !!!

  This is why hashtables came to be

Array: if we know the index, the insertion and retrieval operations

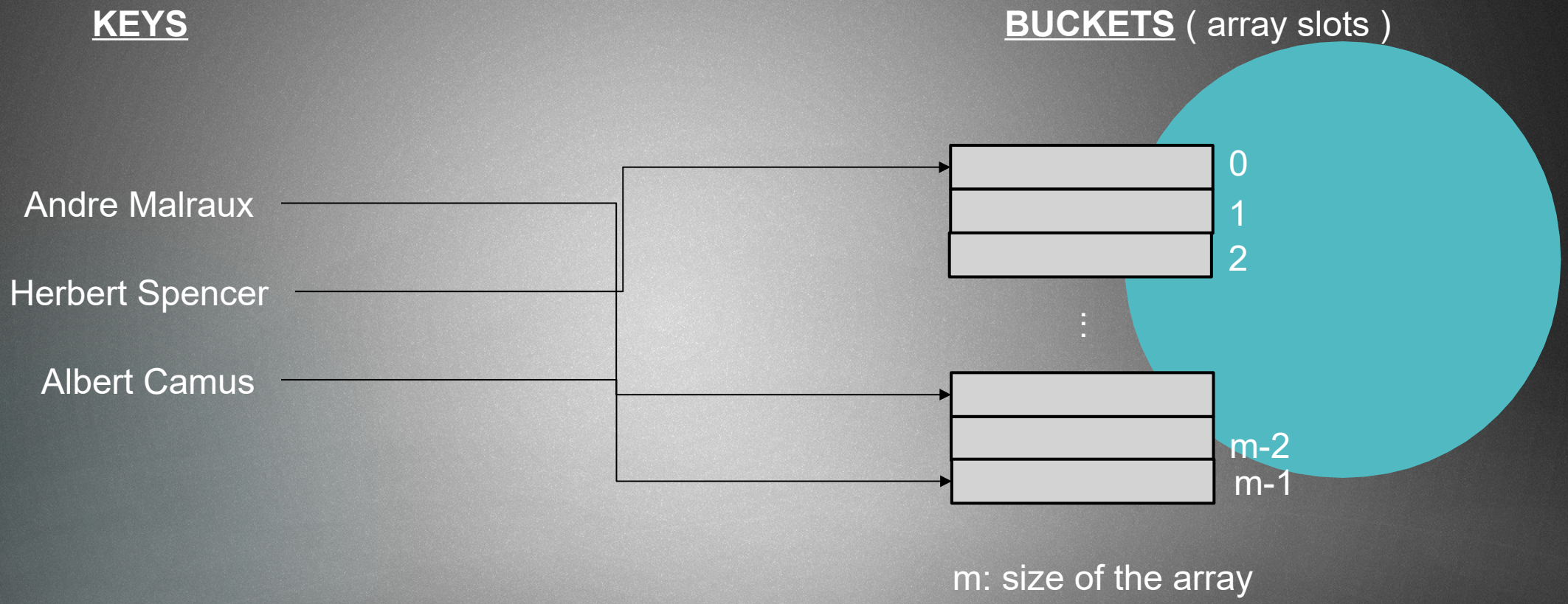
are very fast **O(1)** ... that is what we are after

Here we want to search for a given item with a given key

We have key-value pairs

KEY → slot in a set of buckets

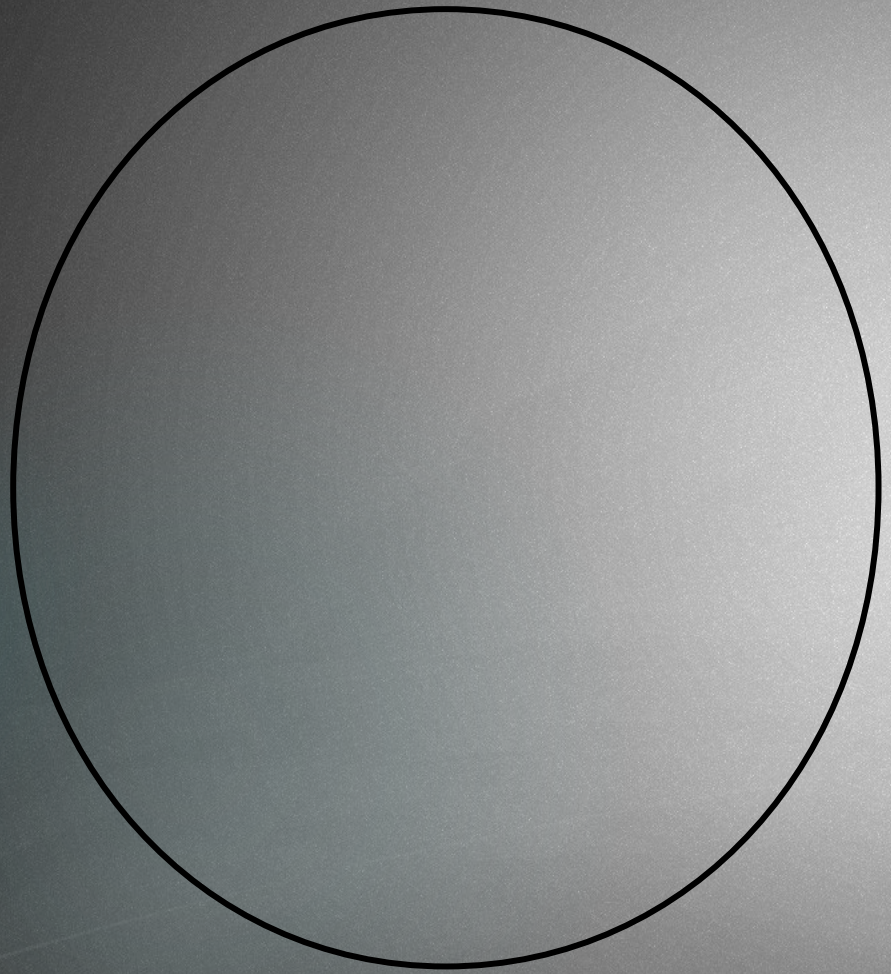
index = h(key) where h() is the hashfunction, it maps keys to indexes in the array !!!



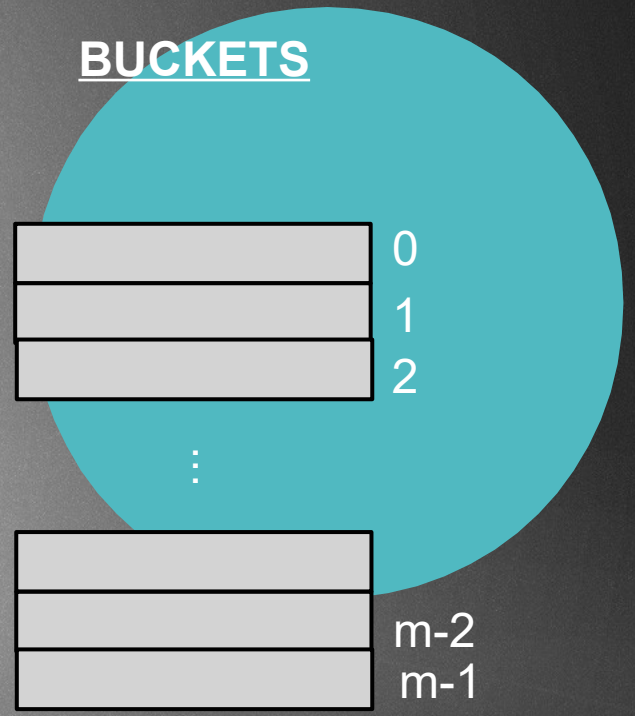
In general: we have ***n*** items to be stored + ***m*** buckets in which we can store items

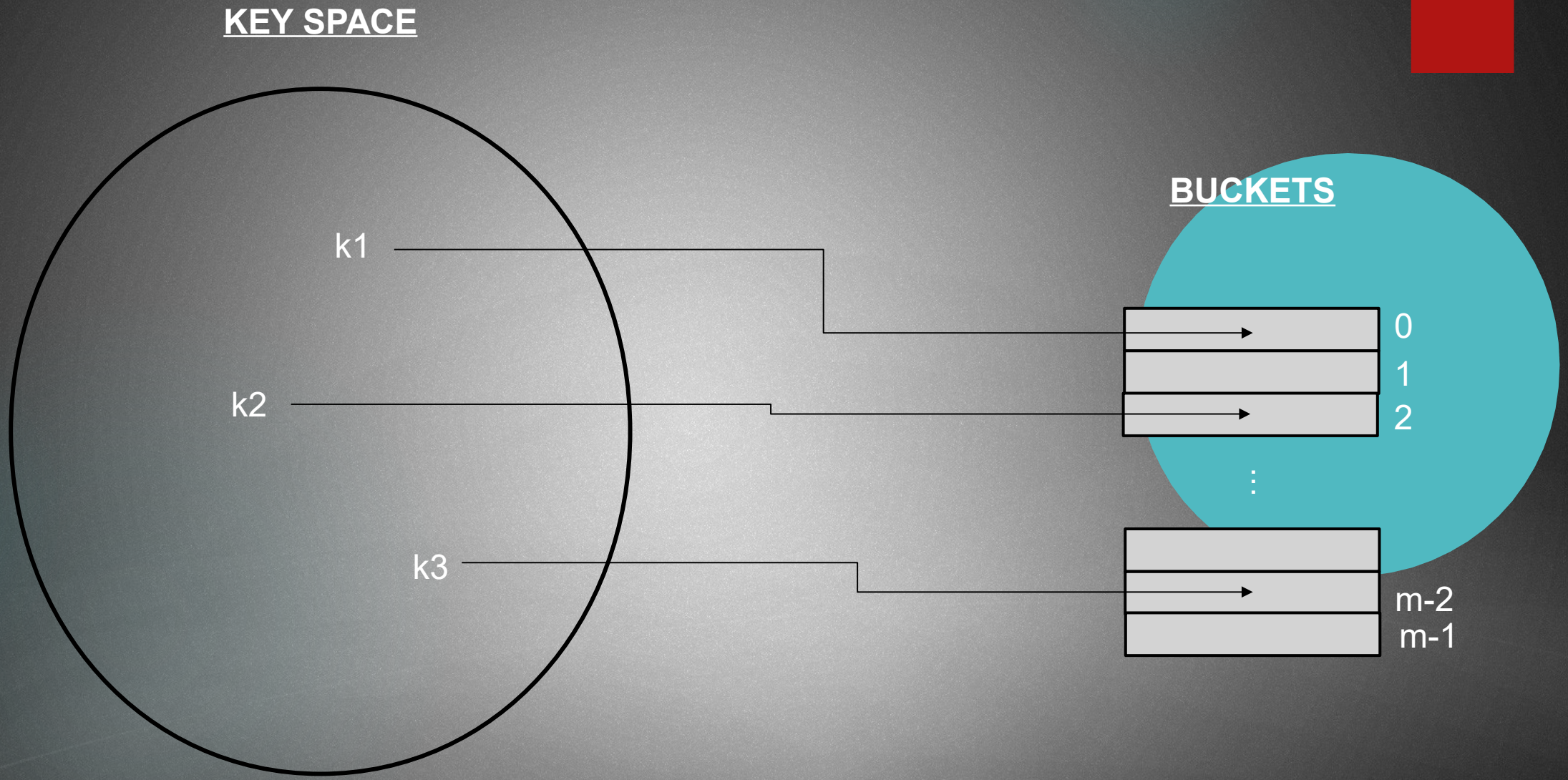
Problem: keys are not always nonnegative integers. We have to do „prehashing” in order to map string keys to indexes of an array !!!

KEY SPACE

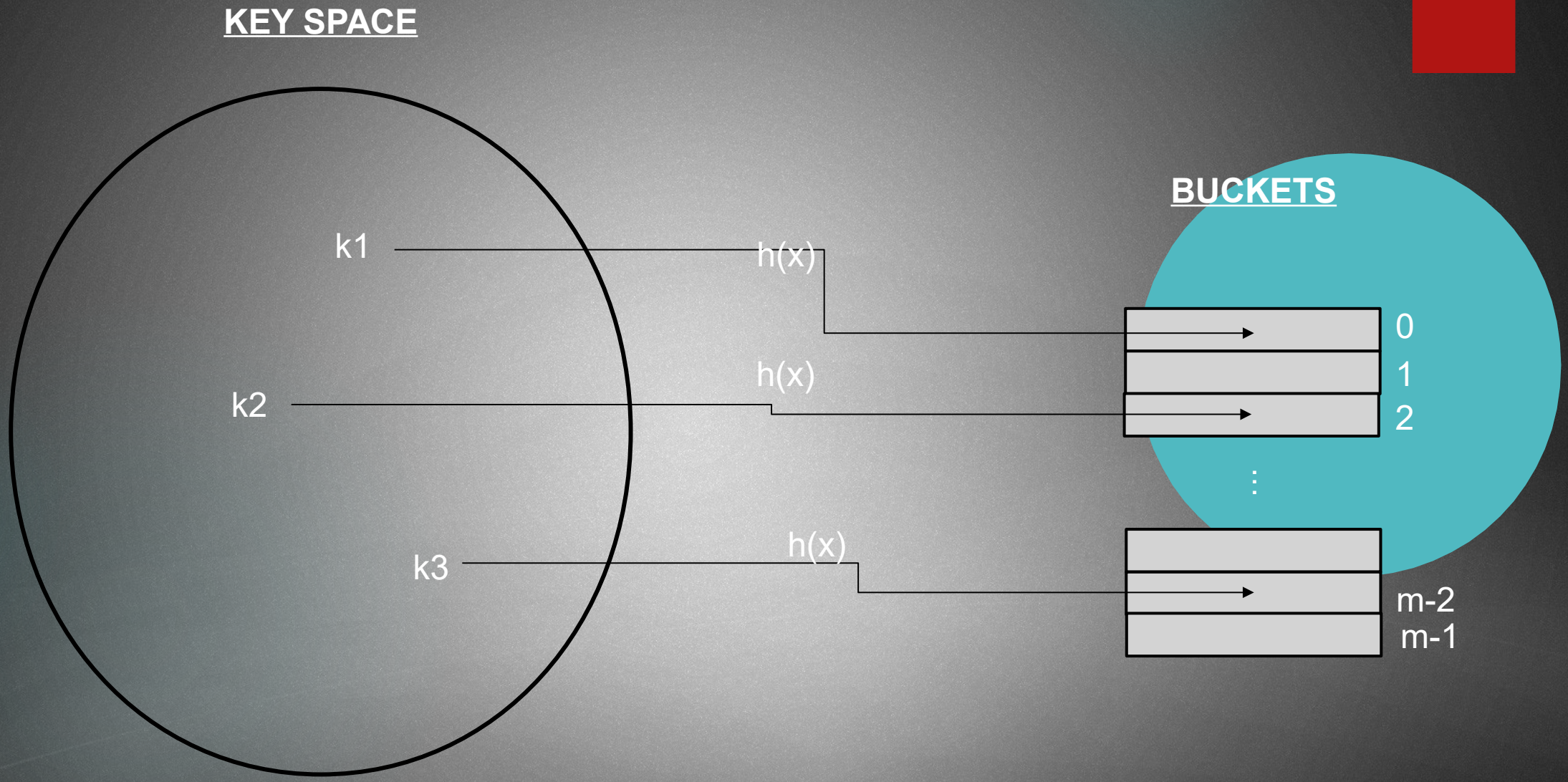


BUCKETS





It is a basically a relationship between the keys and the slots / buckets



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

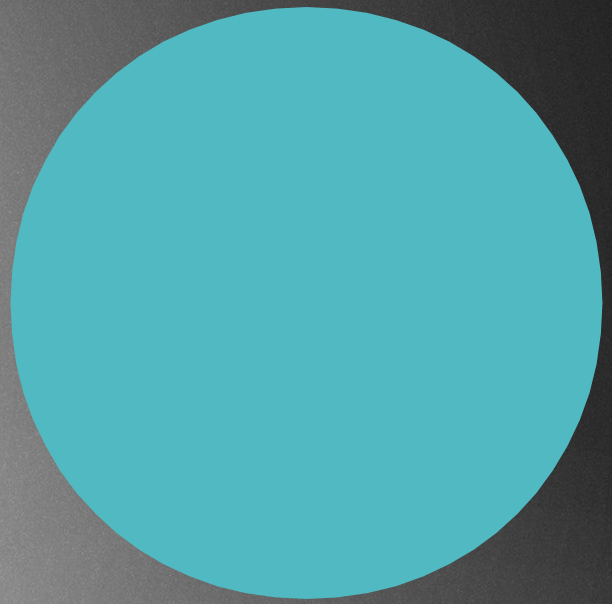
- if we have integer keys we just have to use the modulo operator to transform the number into the range **[0,m-1]** ~ quite easy

- if the keys are strings: we can have the ASCII values of the character and make some transformation in order to end up with an index to the array

Hash function

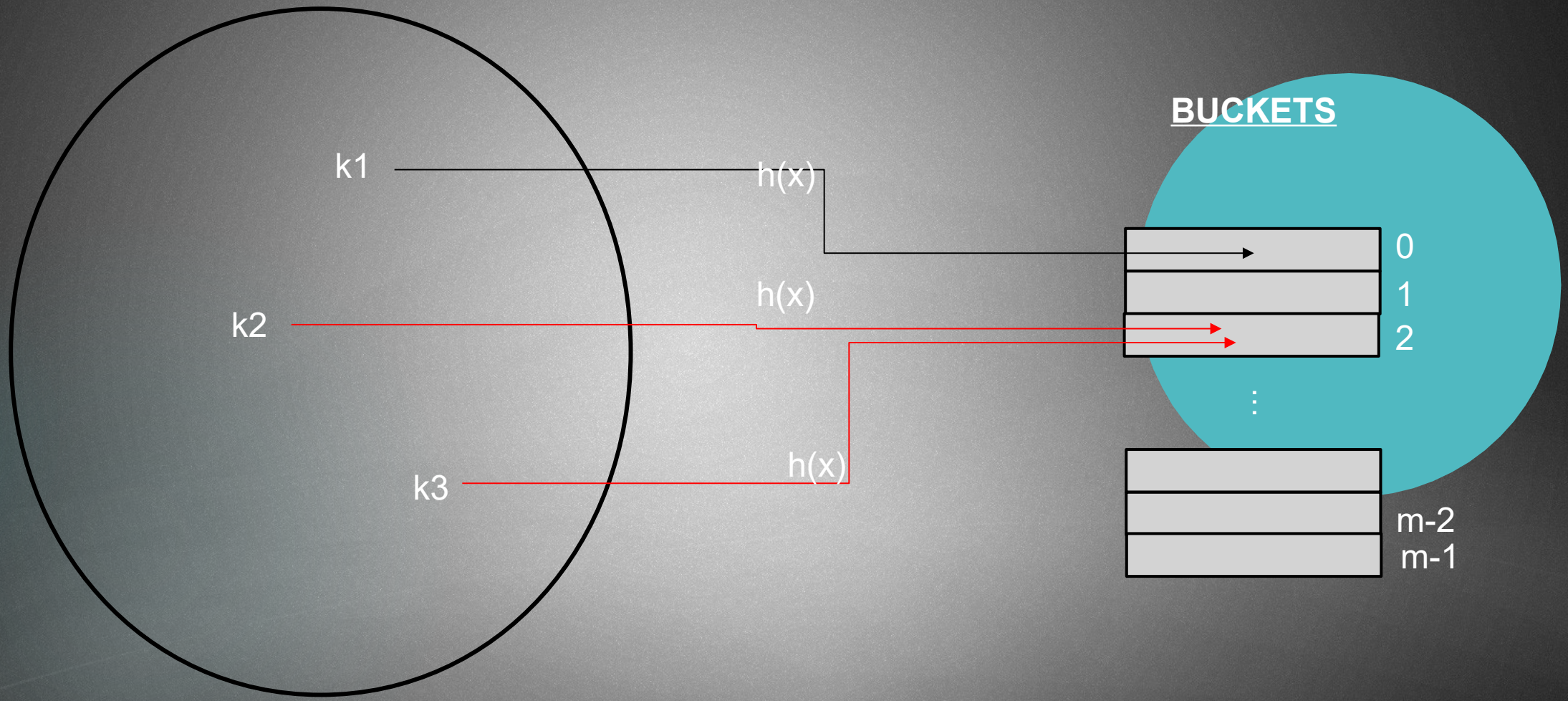
- Distribute the keys uniformly into buckets
- n: number of keys
- m: number of buckets // size of array
- h(x) = n % m (modulo operator)
  - We should use prime numbers both for the size of the array and in our hash function to make sure the distribution of the generated indexes will be uniform !!!
  - String keys: we could calculate the ASCII value for each character, add them up -> make % modulo

Collisions



COLLISION: we map two keys to the same bucket

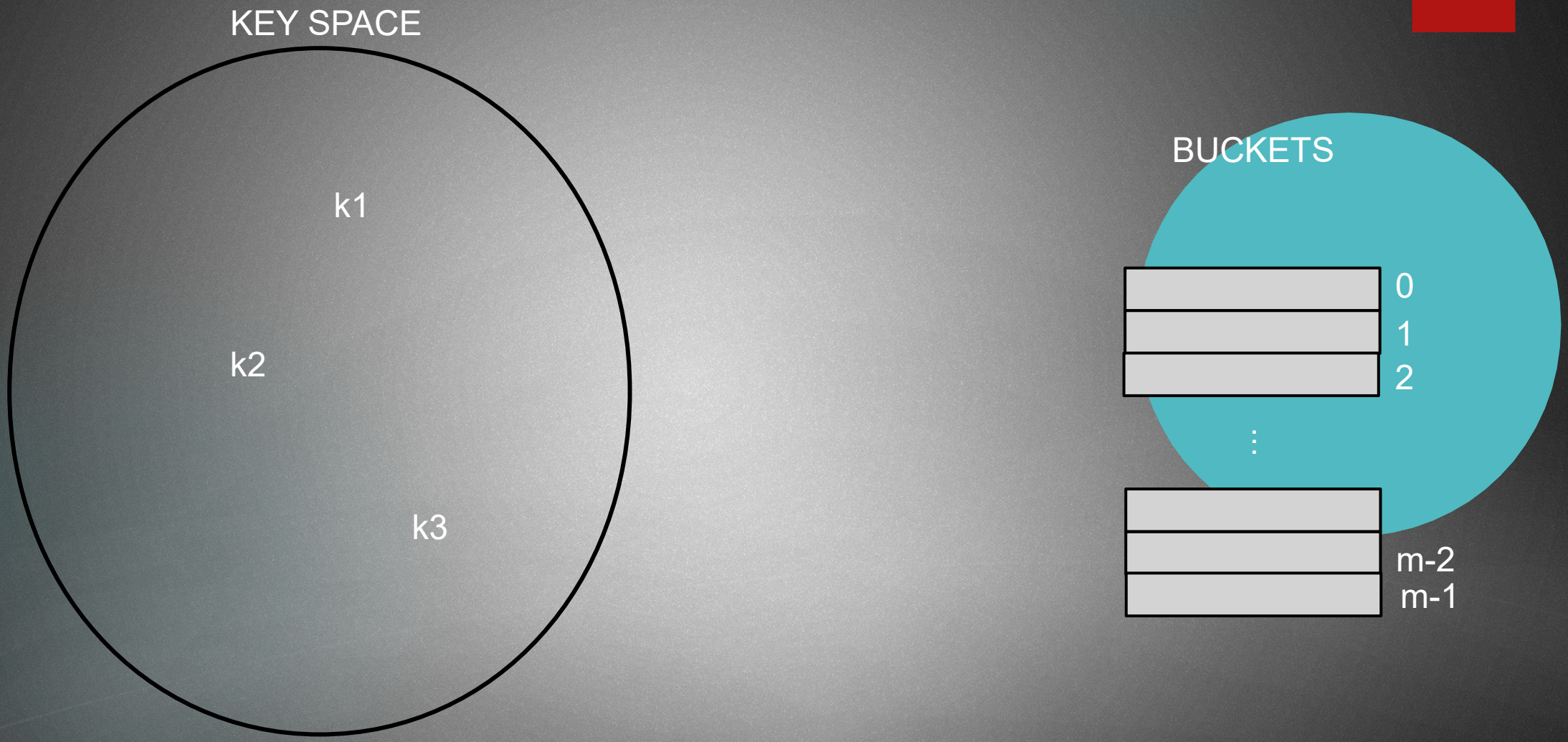
If hash function is perfect: no collisions at all !!!



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

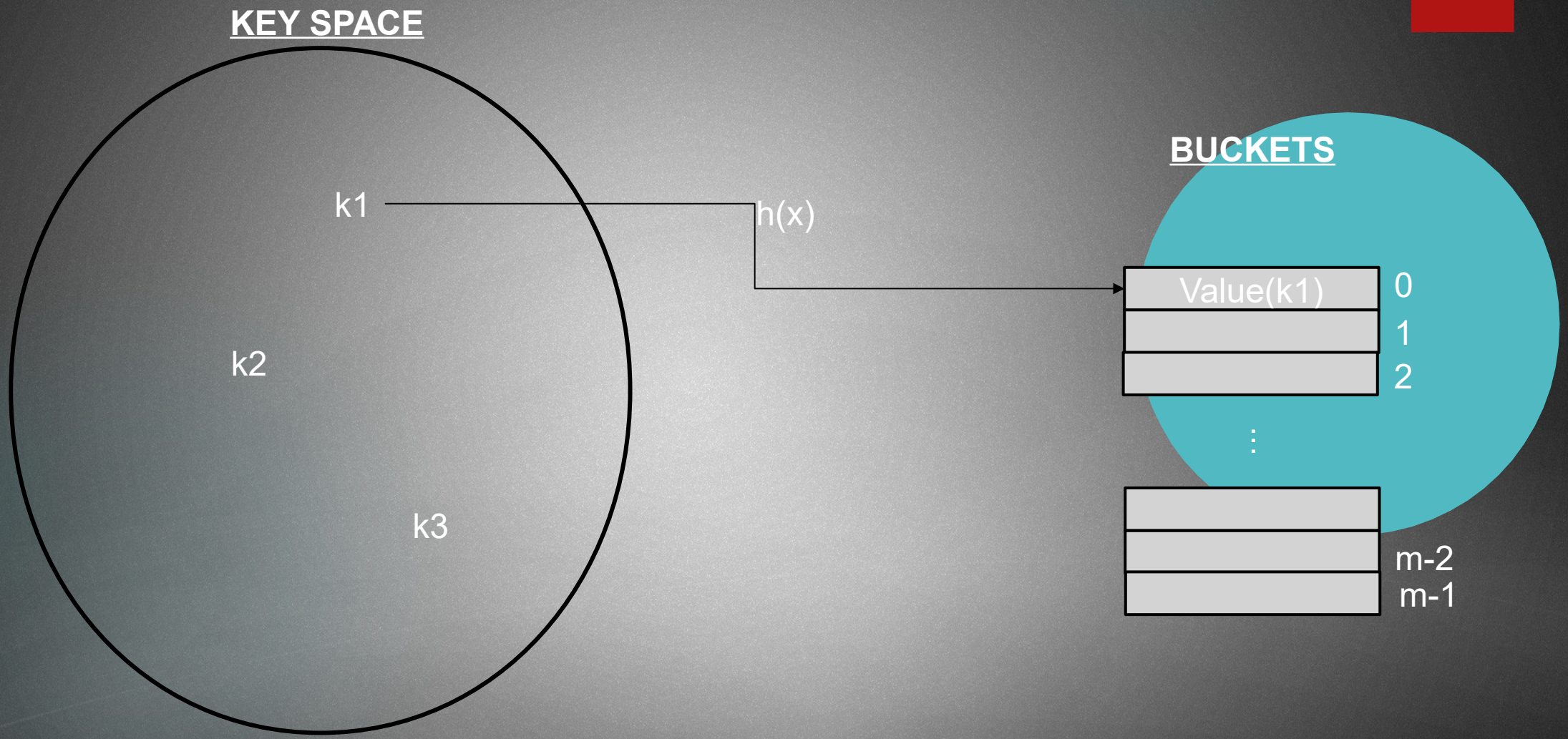
Resolve collision #1 -> **chaining**: we store both values at the same bucket ... we use linked lists to connect them



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

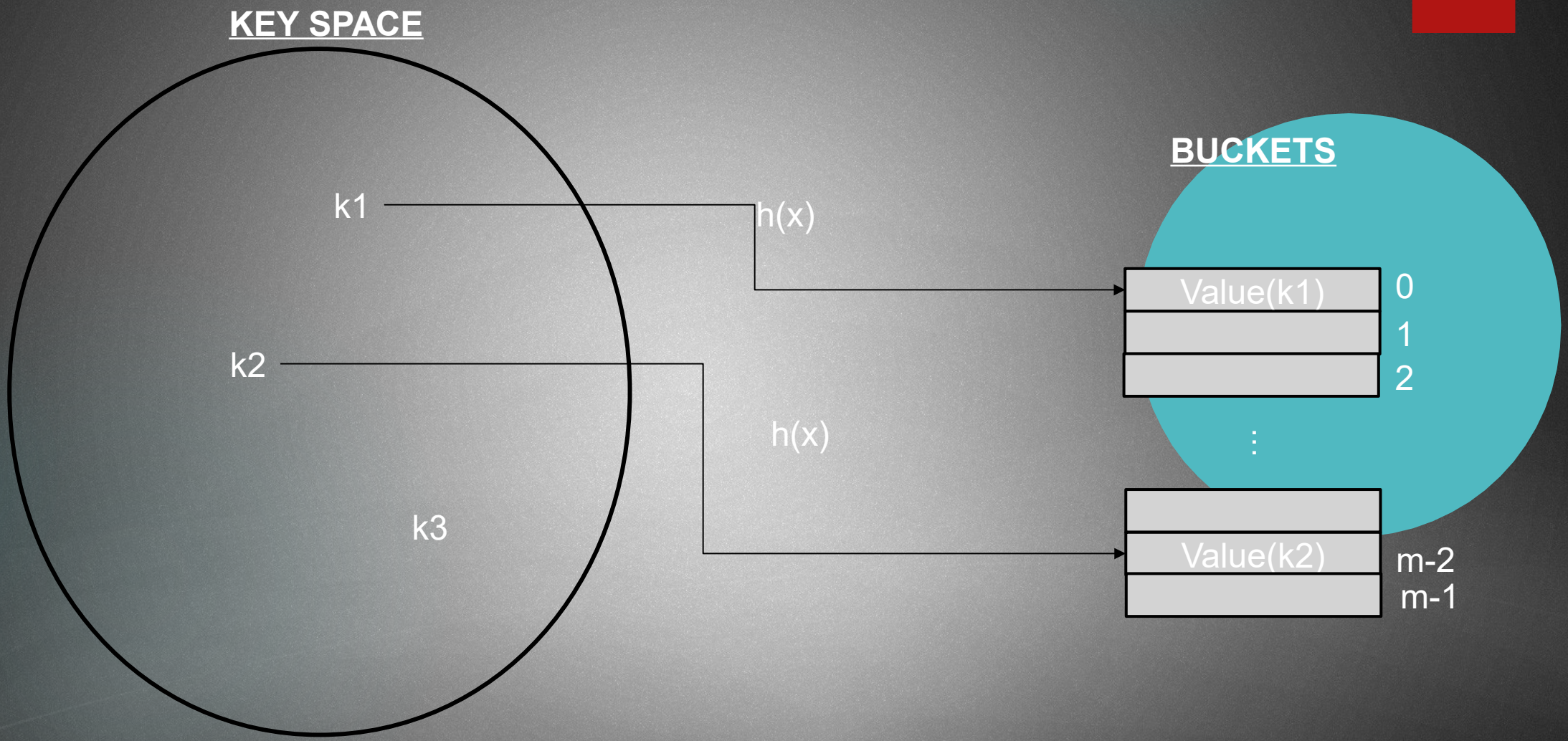
Resolve collision #1 -> **chaining**: we store both values at the same bucket ... we use linked lists



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

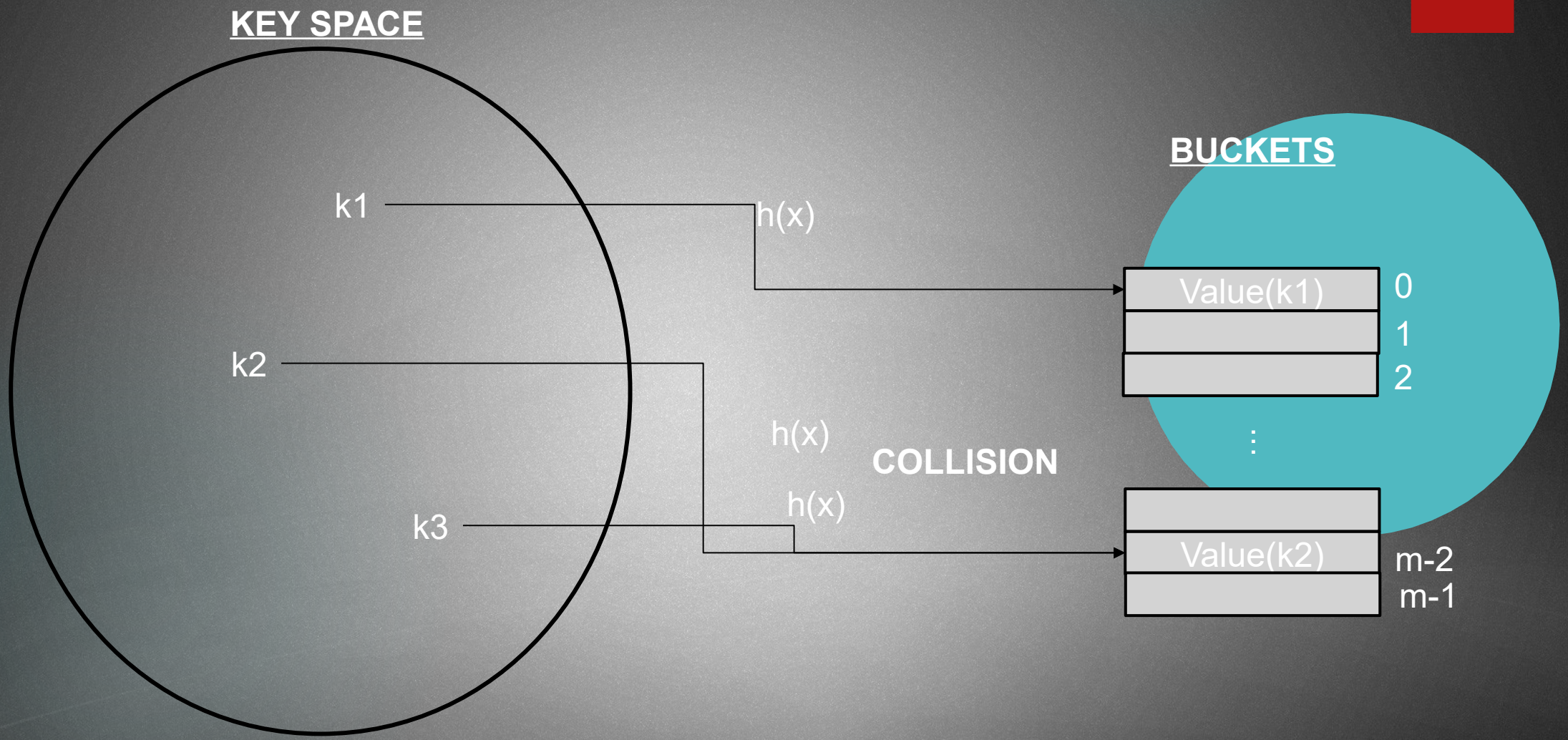
Resolve collision #1 -> **chaining**: we store both values at the same bucket ... we use linked lists



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

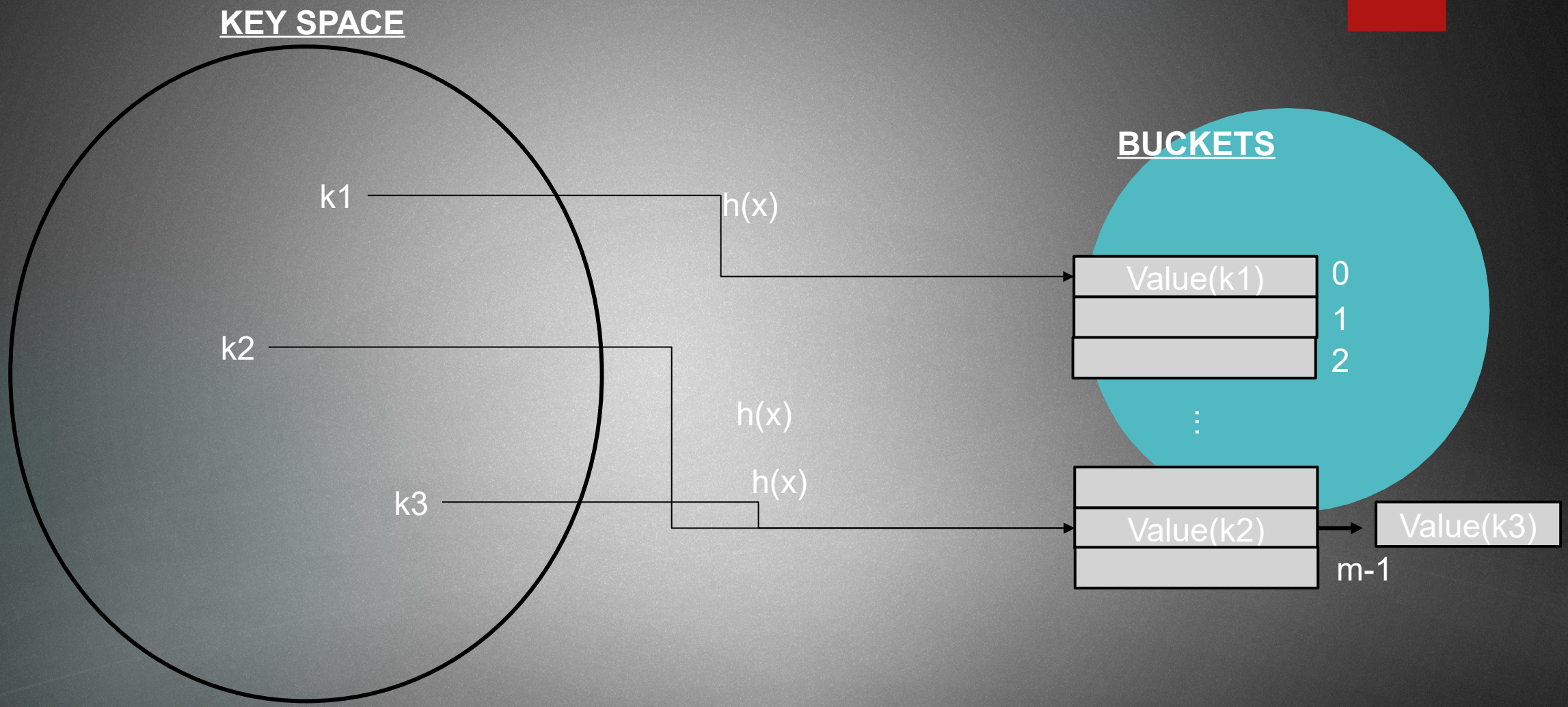
Resolve collision #1 -> **chaining**: we store both values at the same bucket ... we use linked lists



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

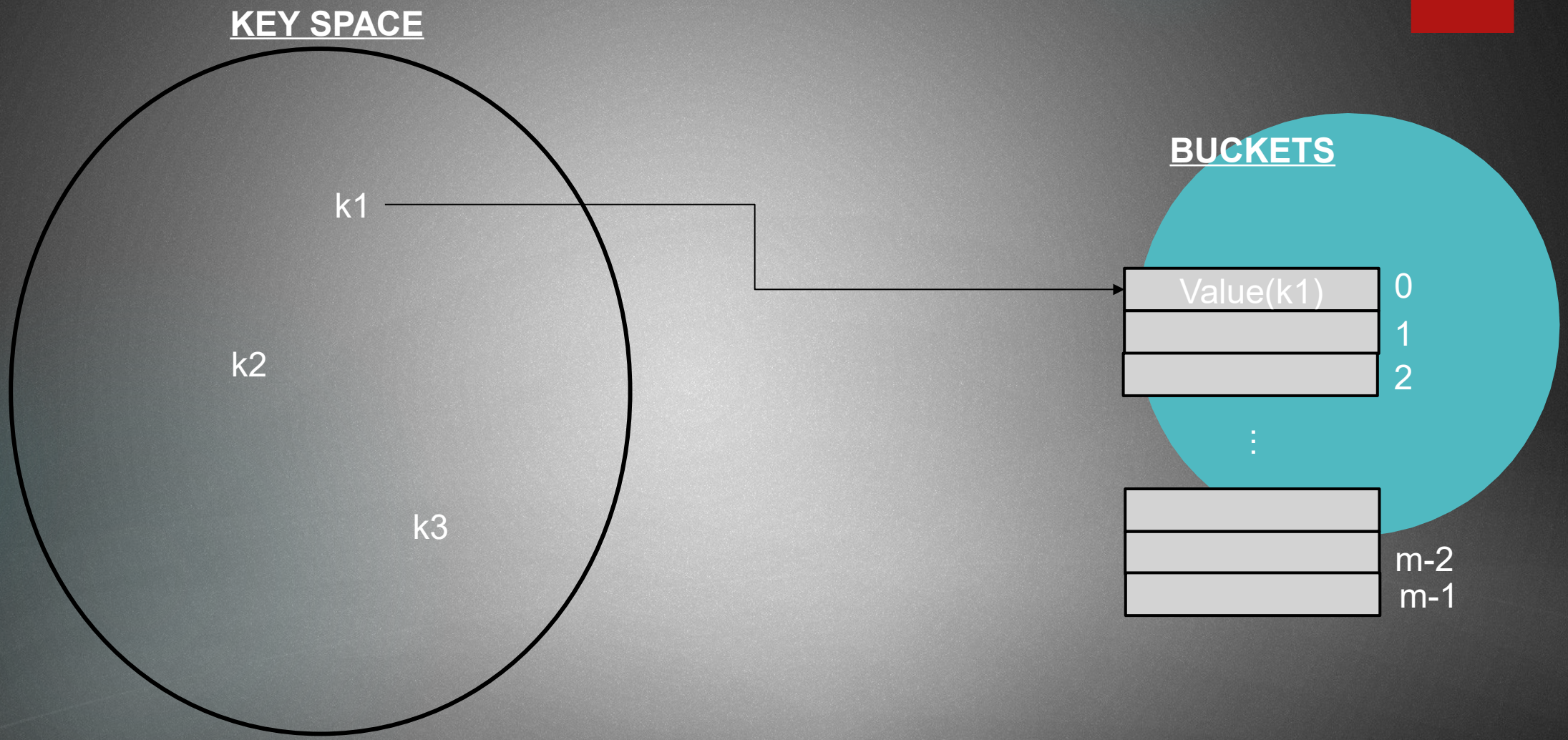
Resolve collision #1 -> **chaining**: we store both values at the same bucket ... we use linked lists



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

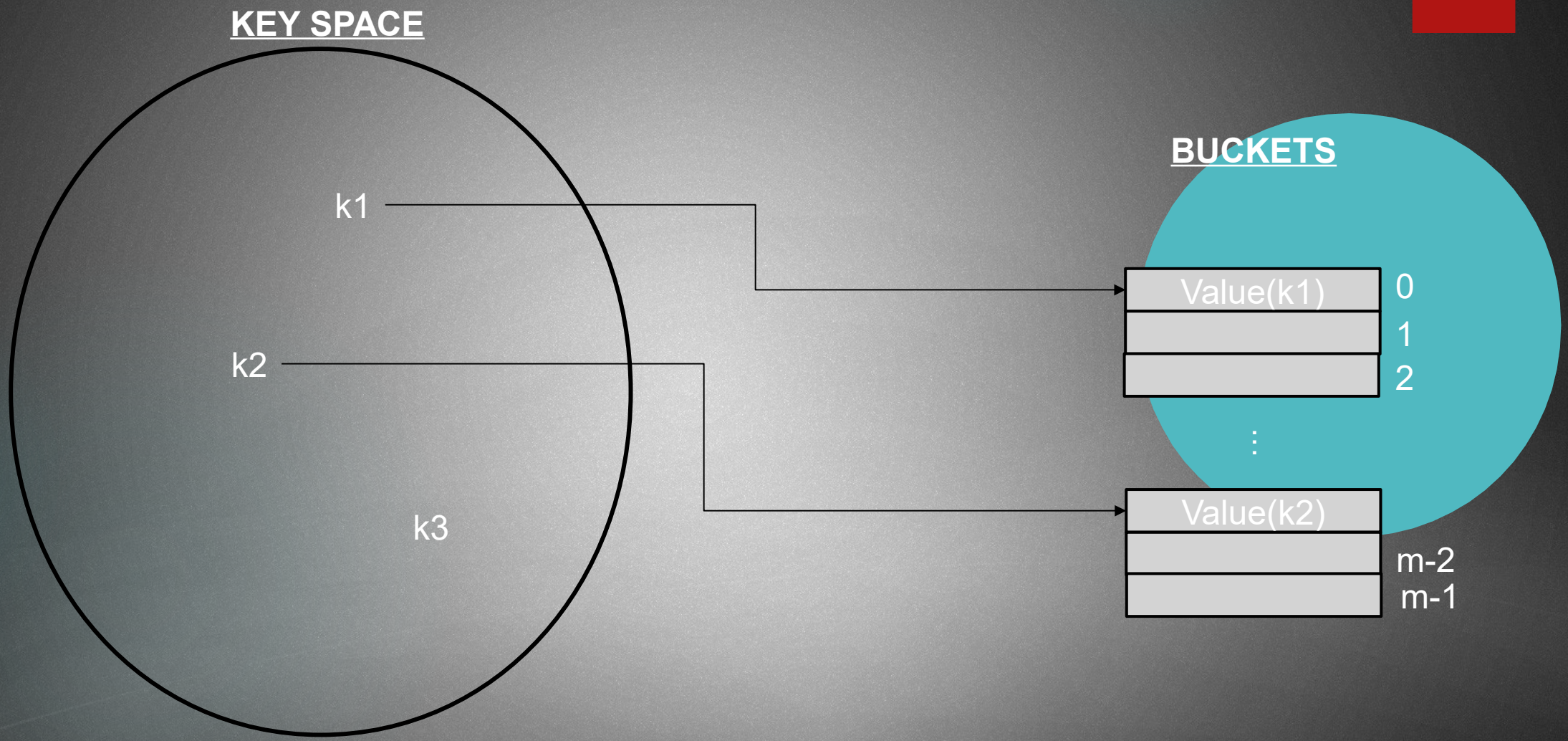
Resolve collision #2 -> **open addressing**: we generate a new index for the item



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

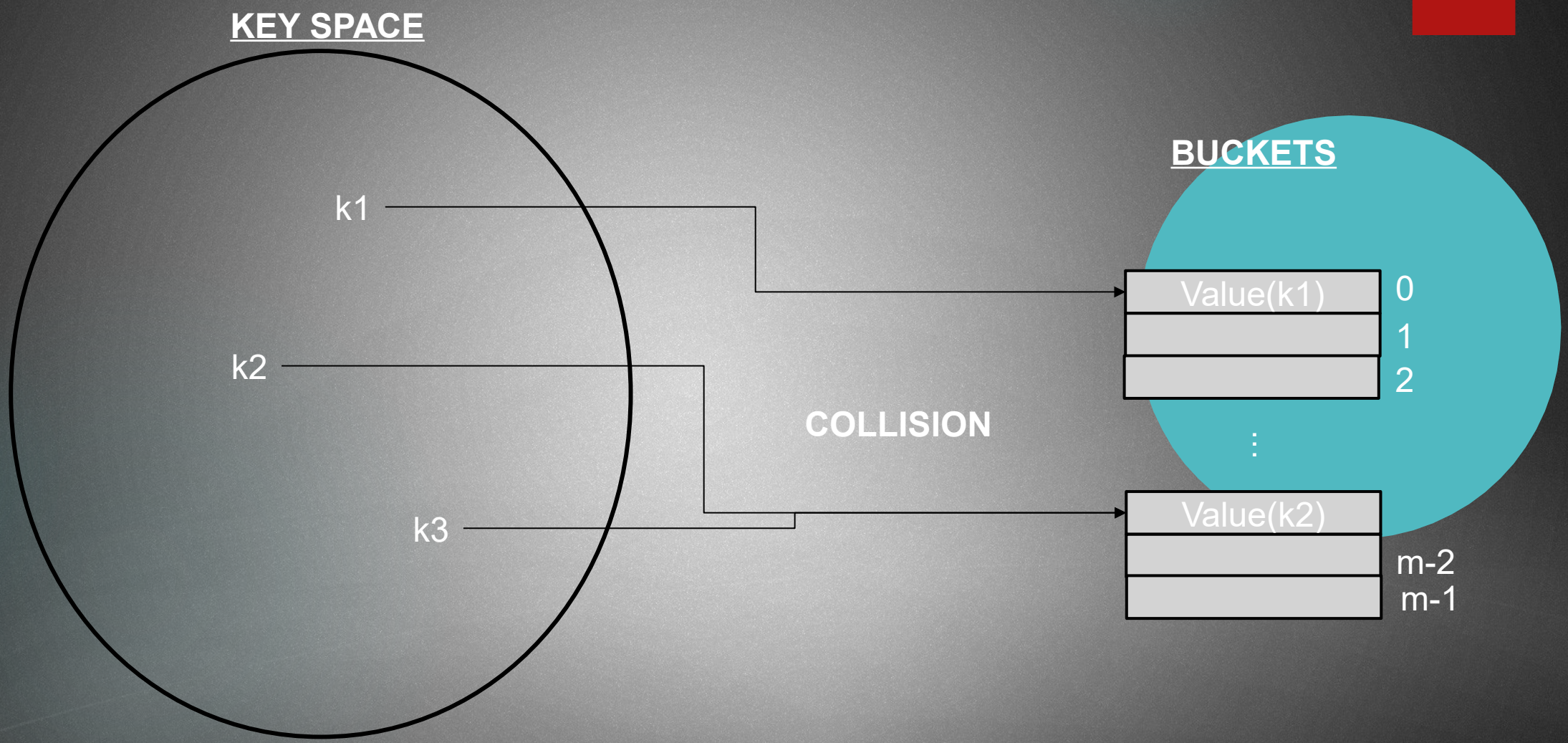
Resolve collision #2 -> **open addressing**: we generate a new index for the item



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

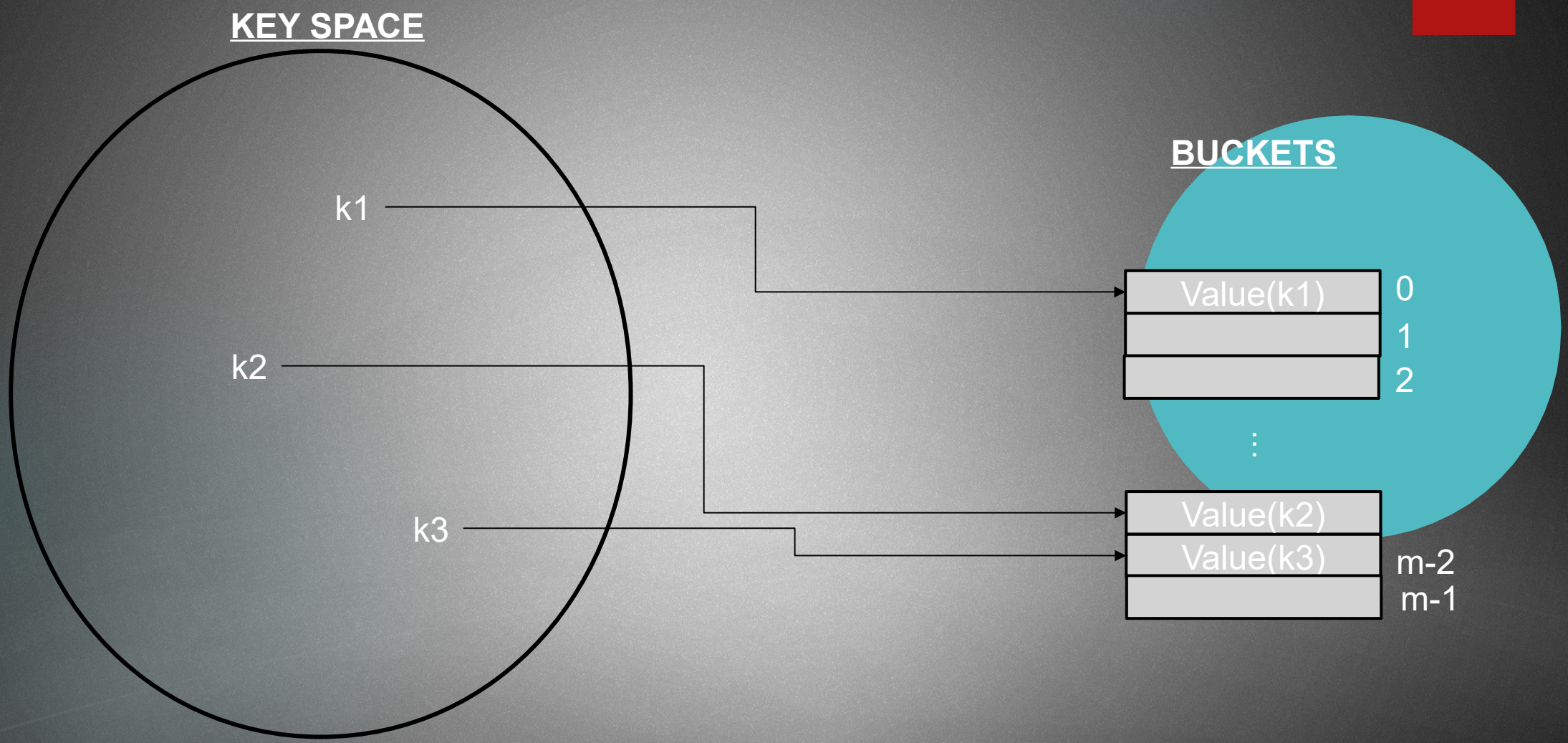
Resolve collision #2 -> **open addressing**: we generate a new index for the item



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

Hashing: we can map a certain key of any type (!!!) to a random array index

Resolve collision #2 -> **open addressing**: we generate a new index for the item



How can we map a certain key to a slot in our array? h(x) hashfunction is needed

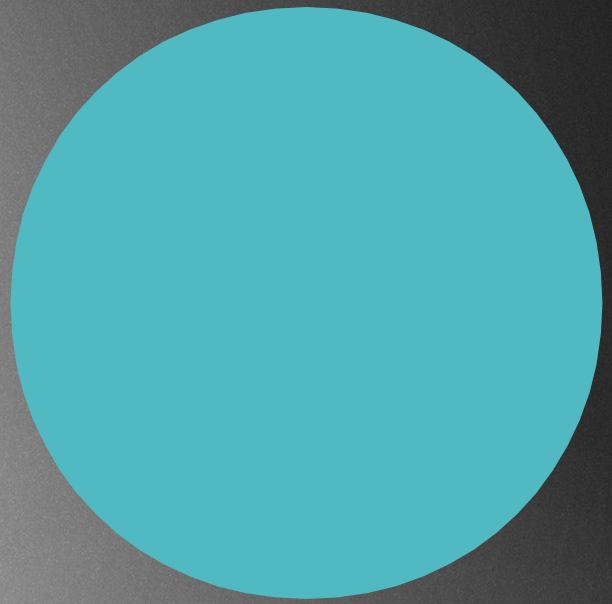
Hashing: we can map a certain key of any type (!!!) to a random array index

Collisions

- Collision resolution with ***chaining***: we put multiple entries into the same slot with the help of a linked list
  - If there are many collisions: O(1) complexity gets worse !!!
  - It has an additional memory cost due to the references
- Collision resolution with **open addressing**: better solution
- If collision occurs, we find an empty slot instead
  - **Linear probing**: if a collision occures, we try the next slot … if there is a collision too we keep trying the next slot until we find an empty slot
  - **Quadratic probing**: we trying slots 1,2,4,8… units far away
  - **Rehashing**: we hash the result again in order to find an empty slot

	Average	Worst case
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Dynamic resizing



Load factor: number of entries divided by the number of slots / buckets

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

This is the load factor. It is 0 if the hashtable is empty, it is 1 if the hashtable is full !!!

- if the load factor is approximately 1 → it means it is nearly full: the performance will decrease, the operations will be slow
- if the load factor is approximately 0 → it means it is nearly empty: there will be a lot of memory wasted

SO: dynamic resizing is needed sometimes !!!

Dynamic resizing:

Performance depends on the load factor: what is the number of entries and number of buckets ratio

Space-time tradeoff is important: the solution is to resize table, when its load factor exceeds given threshold

Java: when the load factor is greater than 0.75, the hashmap will be resized automatically

Python: the threashold is 2/3 ~ 0.66

1.) hash values depend on table's size so hashes of entries are changed when resizing and algorithm can't just copy data from old storage to new one

2.) resizing takes **O(n)** time to complete, where **n** is a number of entries in the table
This fact may make dynamic-sized hash tables inappropriate for real-time applications

Applications

- Databases: sometimes search trees, sometimes hashing is better
- Counting given word occurence in a particular document
- Storing data + lookup tables (password checks...)
- Lookup tables in huge networks (lookup for IP addresses)
- The hashing technique can be used for substring search

  (Rabin-Karp algorithm)