CYDEO

Data Structures and Algorithms Course

Queues –Hashing Review

Course Content

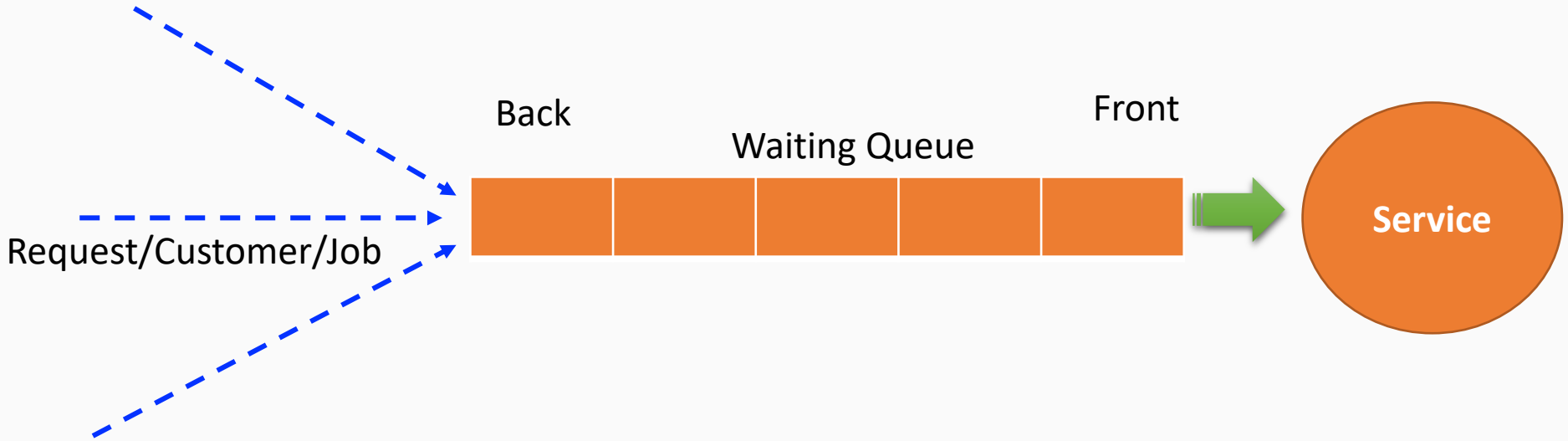
- Big O notation ✓
- Arrays ✓
- Linked Lists ✓
- **Stacks** ✓
- **Recursion** ✓
- Queues
- Hashing
- Sets
- Trees
- AVL Trees
- Heap
- Tries
- Graphs
- Sorting Algorithms
- Searching Algorithms

CYDEO

Queues

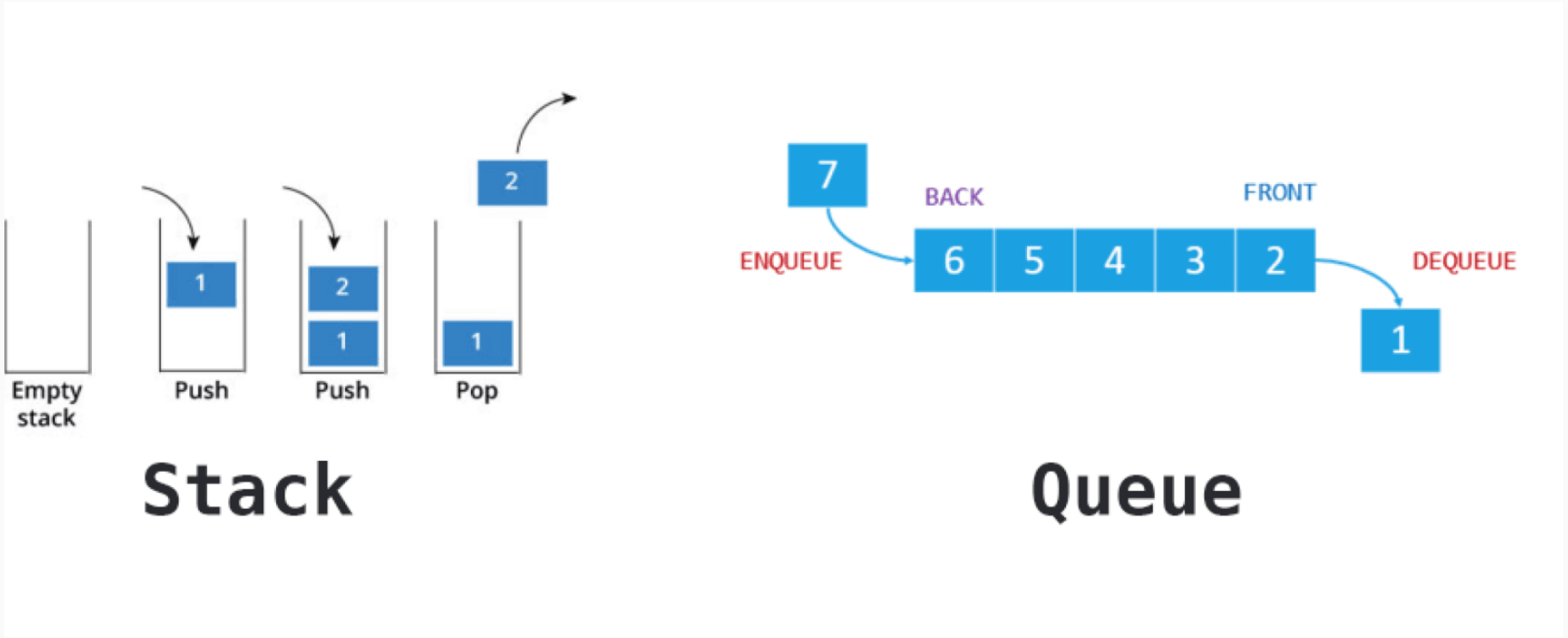
Queues

- Used anywhere you process the jobs in the order you receive them.
  - Web servers (Manage incoming requests)
  - Printers (Use queues to manage printing jobs)
  - Operating systems (Manage processes, processes wait their turn to run)



Queues

- A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.
- Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first.



Queues

Operation	Runtime Complexity
Enqueue (New entry into queue)	O(1)
Dequeue (Removing the front item)	O(1)
Peek (Getting the value of the front without removing)	O(1)
isEmpty	O(1)
isFull	O(1)

Queues – Implementation in Java

- In Java, Queue is an interface, so you can implement this interface or use other classes that implemented Queue.
- Classes that implemented Queue are:

  -AbstractQueue, ArrayBlockingQueue,

  -ArrayDeque, ConcurrentLinkedDeque,

  -ConcurrentLinkedQueue, DelayQueue,

  -LinkedBlockingDeque, LinkedBlockingQueue,

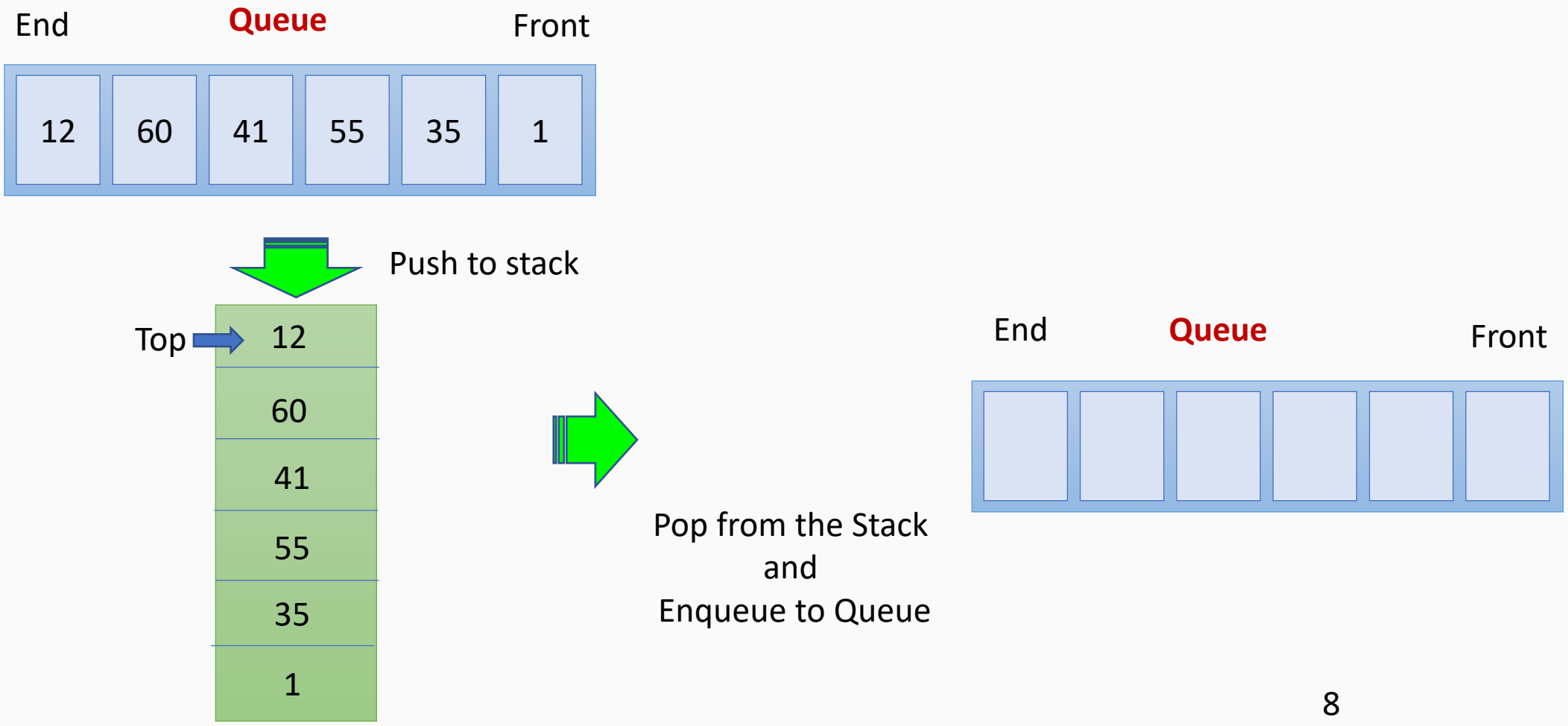
  -PriorityBlockingQueue, **PriorityQueue,**

  -SynchronousQueue

- You can also implement queues using arrays or linked lists (ADT)

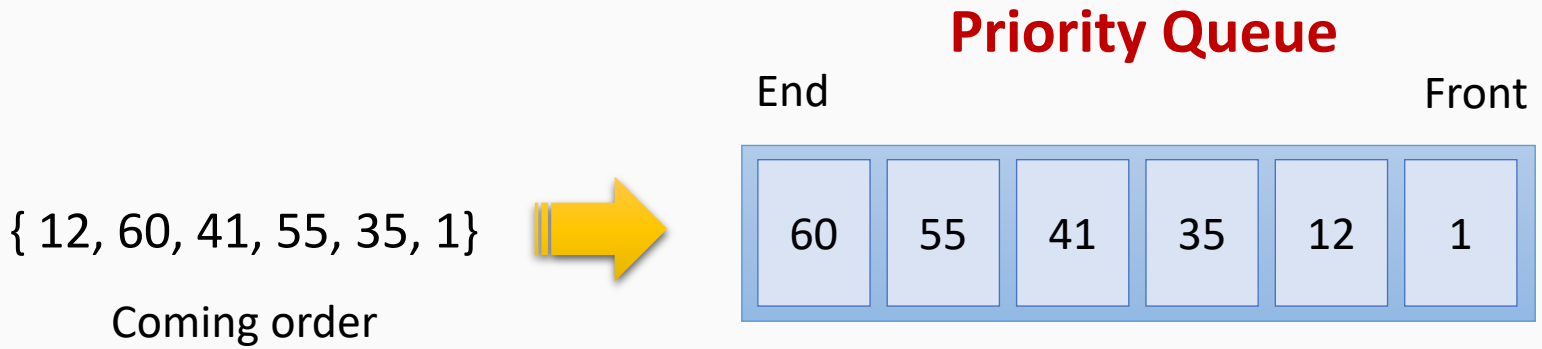
Queues – Interview Question

- How do we reverse the order of a queue?



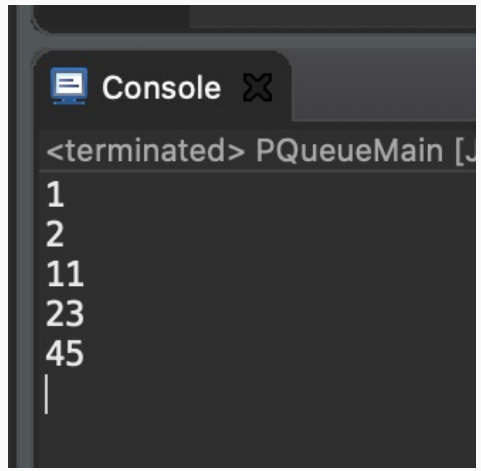
Priority Queues

- In priority queues objects are processed based on their priorities, not the order they come to queue.




```
public static void main(String[] args) {
    PriorityQueue queue=new PriorityQueue();

    queue.add(45);
    queue.add(2);
    queue.add(23);
    queue.add(11);
    queue.add(1);
    while(!queue.isEmpty()) System.out.println(queue.remove());
}
```



CYDEO

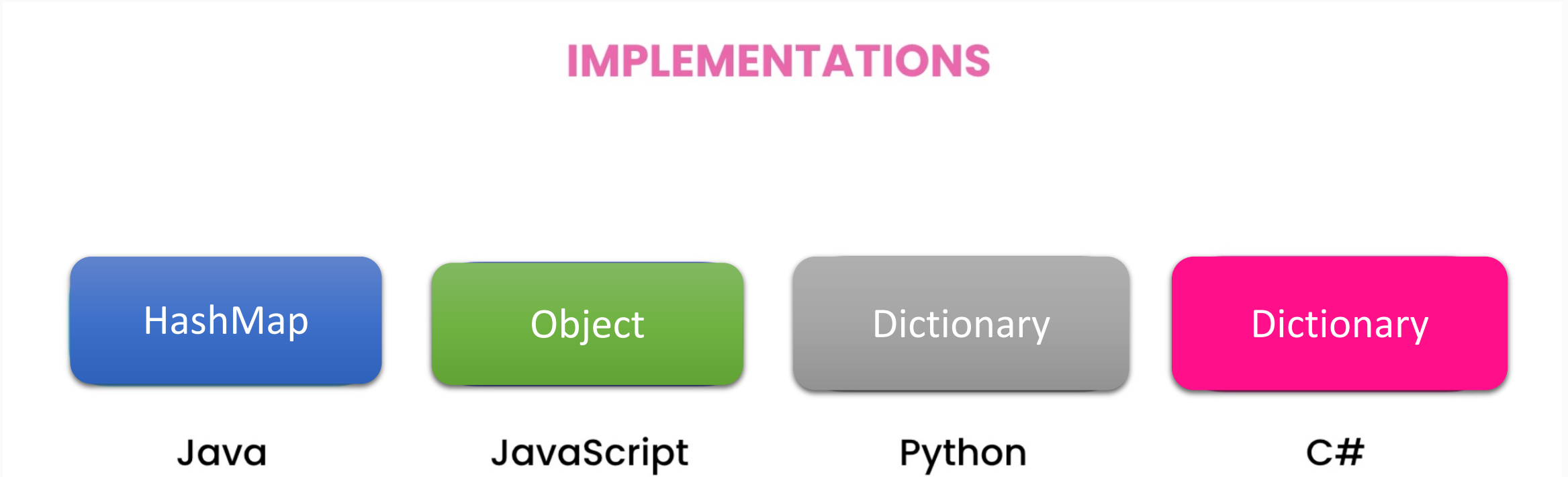
Hashing and Hash Tables

Hash Tables

- Spell checkers (You can quickly lookup a word among thousands of words less than a second.)
- Dictionaries. (You can quickly lookup a word and find its translation.)
- Compilers (They use hash maps to quickly find the address of functions)
- Code editors (Quickly lookup search items)

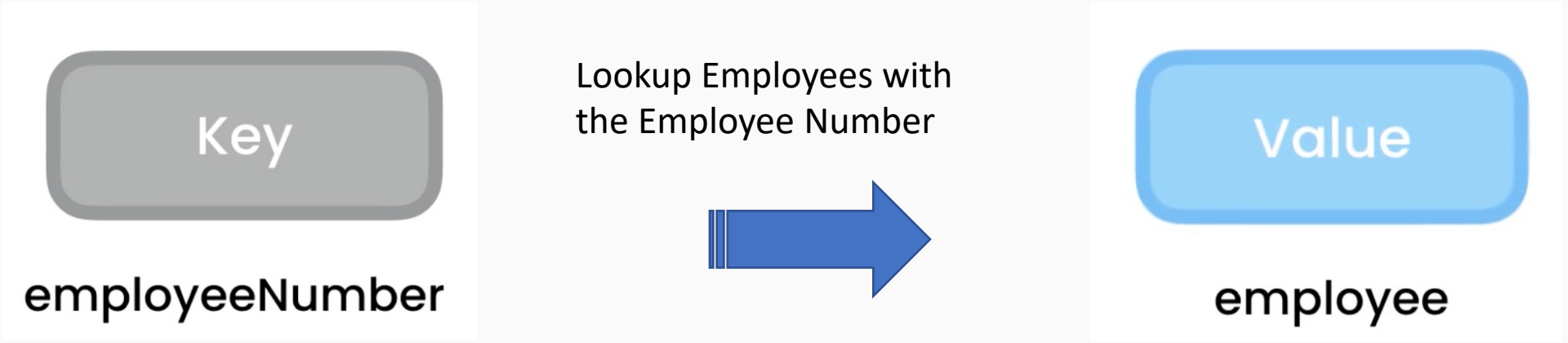
Why Hash ?

Hash Tables – In Programming Languages

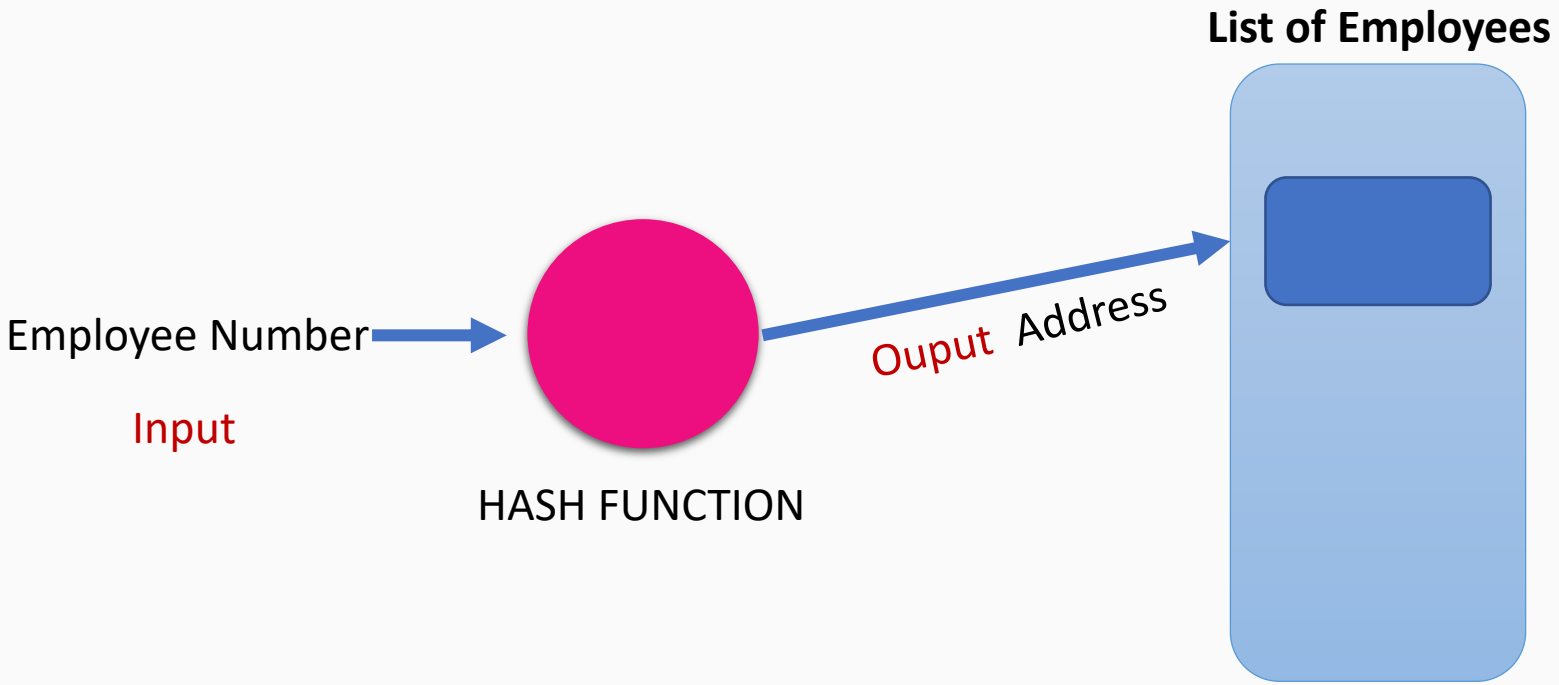


Hash Tables – How Does Hashing Work?

Employee List



Hash Tables – How Does Hashing Work?



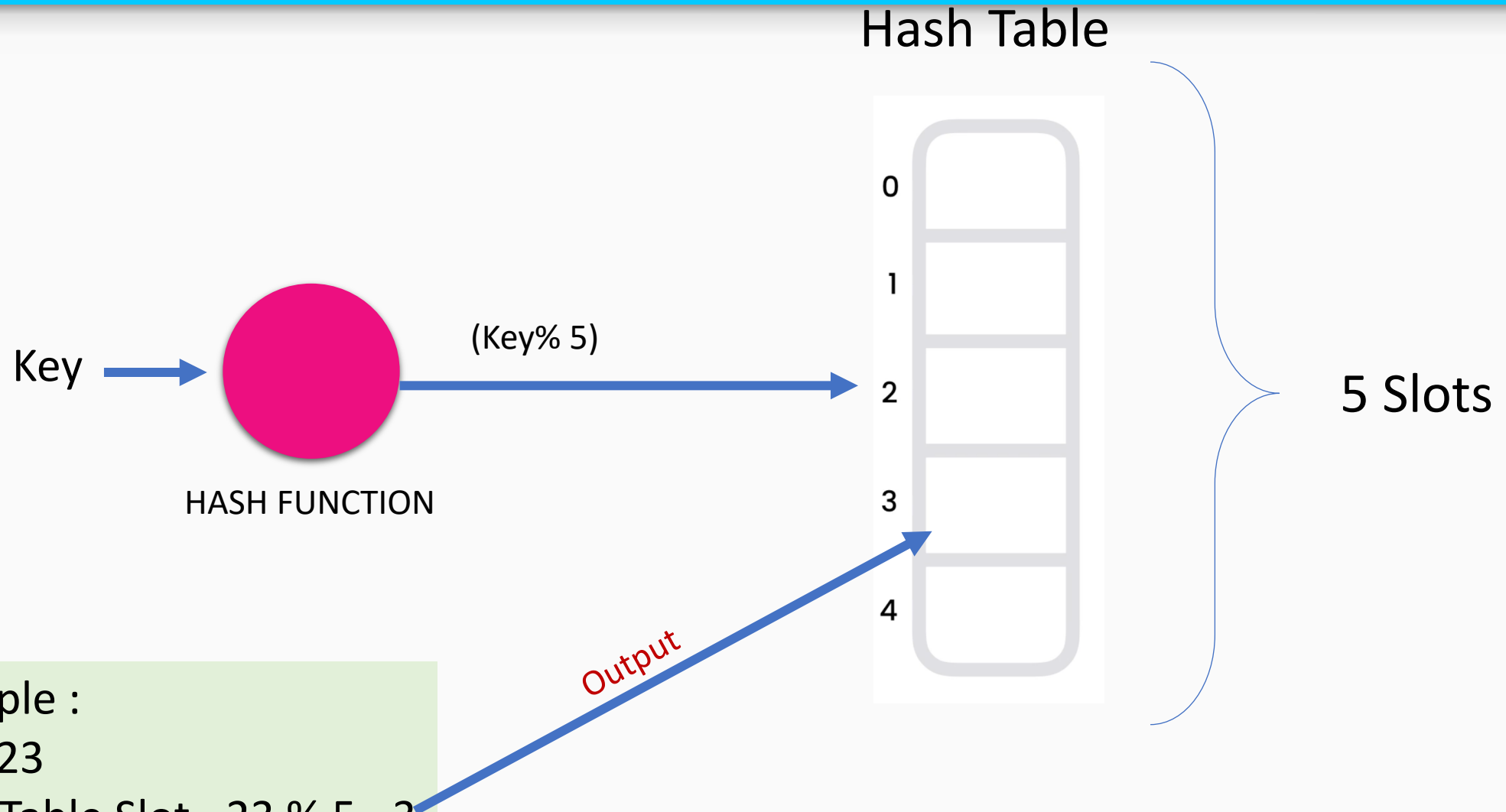
HASH TABLES

Performance

Insert	O(1)
Lookup	O(1)
Delete	O(1)

- Hash Function will calculate the address of the Employee based on the Employee number.
- Hash Function is deterministic, with the same input it will calculate the same address.

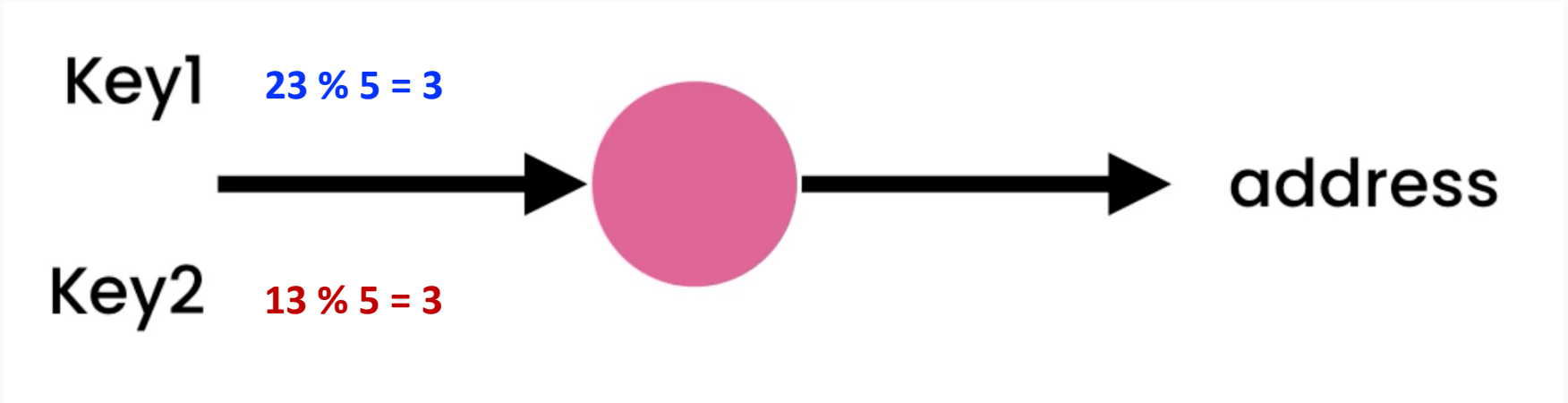
Hash Tables – Hash Function



Example :
Key= 23
Hash Table Slot= 23 % 5 =3

Hash Tables- Collision

- What if Hash Algorithm calculates same address slot for different keys?



We can not store two different values in the same slot, this is called COLLISION.

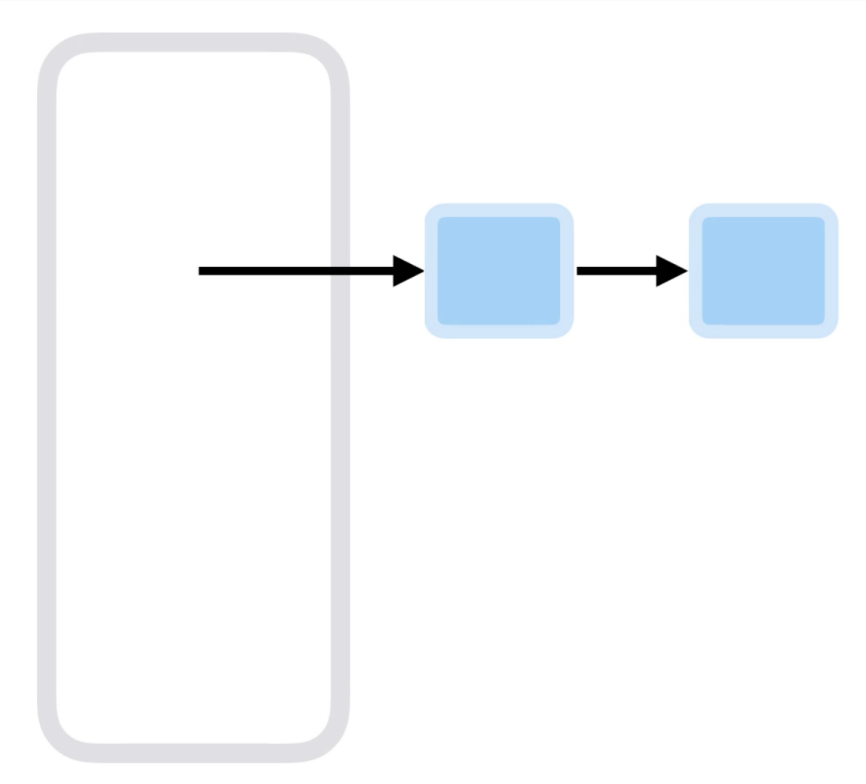
Hash Tables- Collision -Question

- If you do chaining or open addressing when storing/hashing how do you resolve retrieval issues as you might not store the data in the original hash function address

Hash Tables- Collision

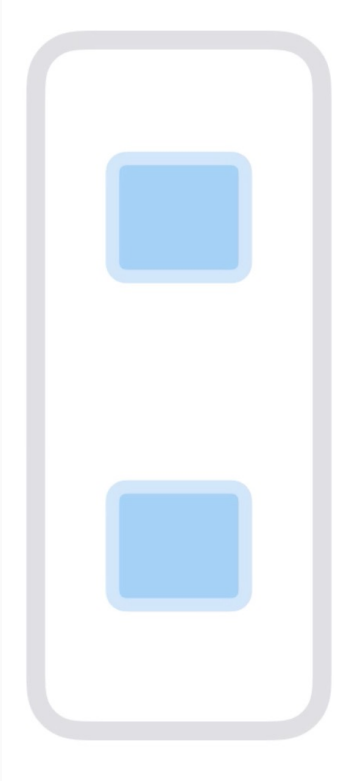
- There are two kinds of solutions to COLLISION problem.

CHAINING



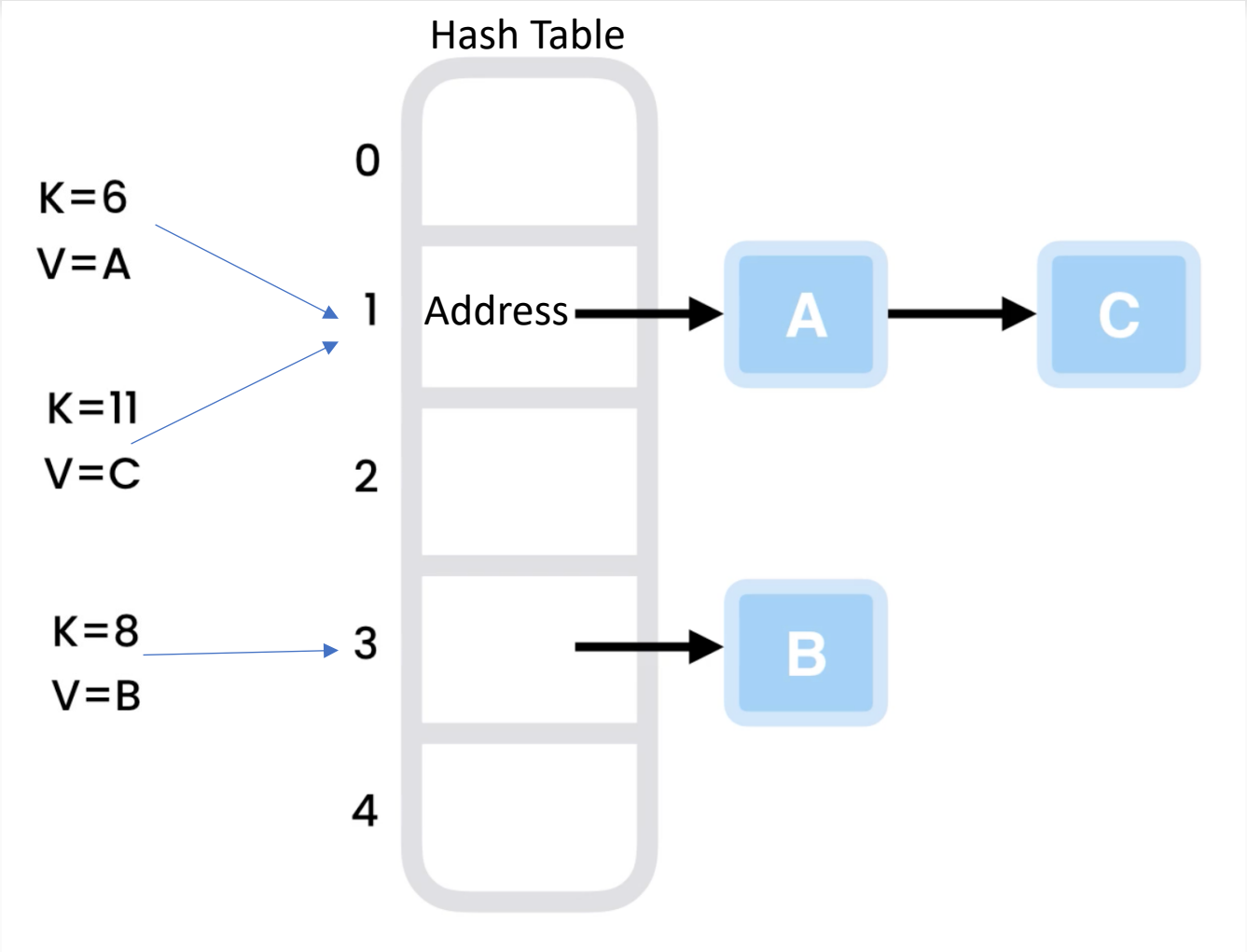
Using a linked lists for additional items addressing same slot

OPEN ADDRESSING



Uses open slots to store the value.

Hash Tables- Chaining

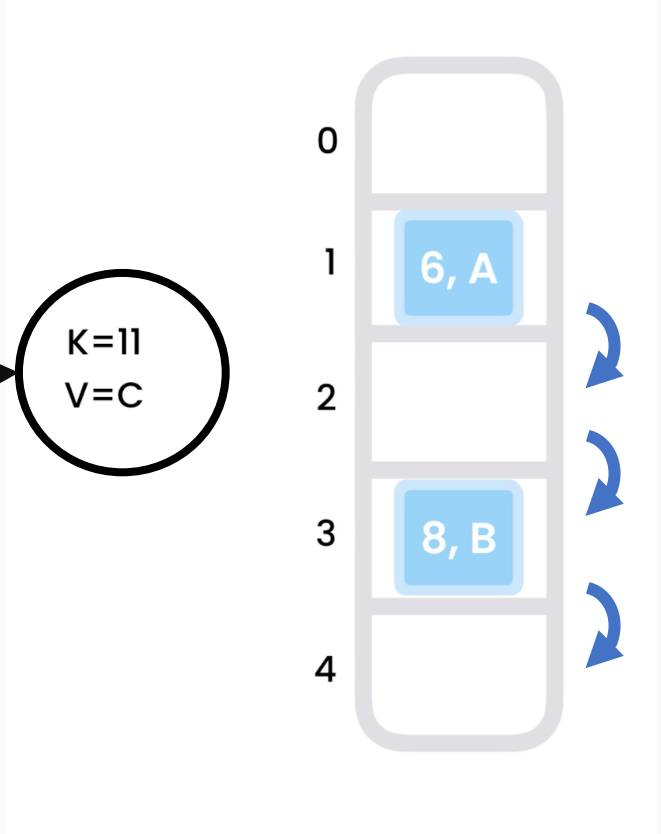


- No longer collisions.
- Linked Lists can grow or shrink without any limitation.

Hash Tables- Open Addressing (Linear Probing)

- We don’t store values in linked lists, instead in the hash table.

Where do we store this newcomer?
Slot 1 is full.



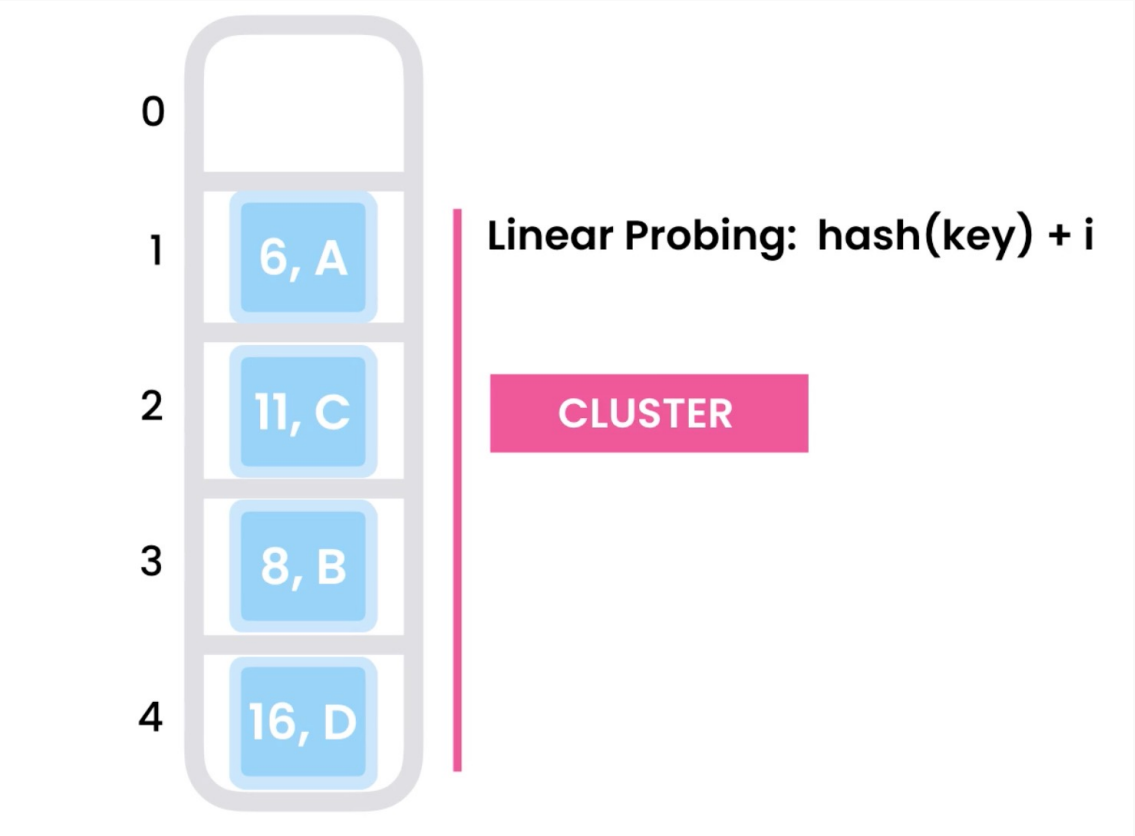
Solution: PROBING
We will search an empty slot for the new value.

LINEAR PROBING
- Start from the next slot and search an empty slot until for the new value.
- What if we can not find an empty slot?
- What if the values form a cluster in sequence? Search time increases.

Linear Probing: hash(key) + i

Hash Tables- Open Addressing

- What if the values form a cluster? Search time increases.



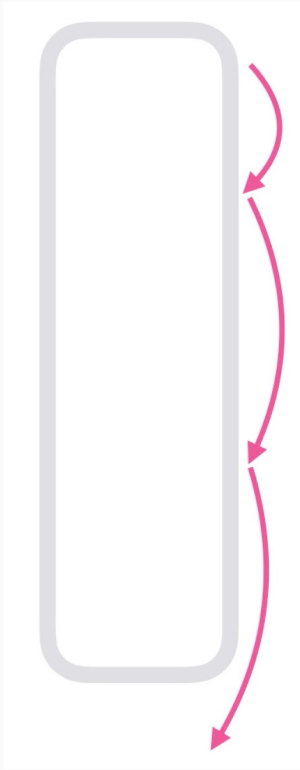
Solution to clustering is **Quadratic Probing**

Hash Tables- Open Addressing (Quadratic Probing)

Linear	Quadratic
hash(key) + i	hash(key) + i^2
1	1
2	4
3	9
4	16
5	25

Problem:

We may end up repeating the same loop.



Hash Tables- Open Addressing (Double Hashing)

Double Hashing

Prime Number < Table size

$$hash2(key) = prime - (key \% prime)$$

$$(hash1(key) + i * hash2(key)) \% table\_size$$

Linear: i

Quadratic: i^2

Double: i * hash2

HashMap Implementations In Java

HashMap:

- *HashMap,* being a hashtable-based implementation, internally uses an **array-based data structure** to organize its elements according to the *hash function*.
- *HashMap* provides expected **constant-time performance *O(1)*** for most operations like *add()*, *remove()* and *contains().* Therefore, it's significantly faster than a *TreeMap*.
- An improper implementation of the *hash function* may lead to a poor distribution of values in buckets which results in:

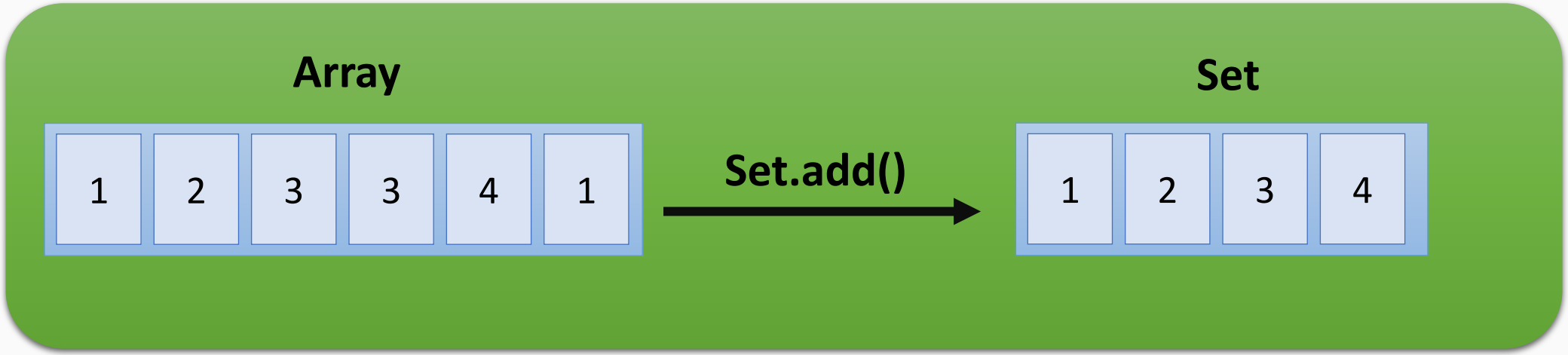
  Memory Overhead – many buckets remain unused

  Performance Degradation – the higher the number of collisions, the lower the performance

- Before Java 8, *Separate Chaining* was the only preferred way to handle collisions. It's usually implemented using linked lists, *i.e.*, if there is any collision or two different elements have same hash value then store both the items in the same linked list.
- Therefore, searching for an element in a *HashMap,* in the worst case could have taken as long as searching for an element in a linked list *i.e. O(n)* time.

Sets

- Data structure similar to HashMaps.
- Don’t allow duplicate keys.
- In Java there is a generic interface for Sets.



Sets

- Java contains three Set implementations:
  - HashSet,
  - TreeSet,
  - LinkedHashSet.

- HashSet, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.
- TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.
- LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).

Practice- LeetCode #706 Design HashMap

- Design a HashMap without using any built-in hash table libraries.
- Implement the MyHashMap class:
  1. MyHashMap() initializes the object with an empty map.
  2. void put(int key, int value) inserts a (key, value) pair into the HashMap. If the key already exists in the map, update the corresponding value.
  3. int get(int key) returns the value to which the specified key is mapped, or **-1** if this map contains no mapping for the key.
  4. void remove(key) removes the key and its corresponding value if the map contains the mapping for the key.

Practice- LeetCode #706 Design HashMap

Example 1:

Input ["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]
[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]

Output [null, null, null, 1, -1, null, 1, null, -1]

Explanation:

MyHashMap myHashMap = new MyHashMap();

myHashMap.put(1, 1); // The map is now [[1,1]]

myHashMap.put(2, 2); // The map is now [[1,1], [2,2]]

myHashMap.get(1); // return 1, The map is now [[1,1], [2,2]]

myHashMap.get(3); // return -1 (i.e., not found), The map is now [[1,1], [2,2]]

myHashMap.put(2, 1); // The map is now [[1,1], [2,1]] (i.e., update the existing value) myHashMap.get(2); // return 1, The map is now [[1,1], [2,1]]

myHashMap.remove(2); // remove the mapping for 2, The map is now [[1,1]]

myHashMap.get(2); // return -1 (i.e., not found), The map is now [[1,1]]

Practice- LeetCode #706 Design HashMap

Example 1:

Input ["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]
[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]

Output [null, null, null, 1, -1, null, 1, null, -1]

Explanation:

MyHashMap myHashMap = new MyHashMap();

myHashMap.put(1, 1); // The map is now [[1,1]]

myHashMap.put(2, 2); // The map is now [[1,1], [2,2]]

myHashMap.get(1); // return 1, The map is now [[1,1], [2,2]]

myHashMap.get(3); // return -1 (i.e., not found), The map is now [[1,1], [2,2]]

myHashMap.put(2, 1); // The map is now [[1,1], [2,1]] (i.e., update the existing value) myHashMap.get(2); // return 1, The map is now [[1,1], [2,1]]

myHashMap.remove(2); // remove the mapping for 2, The map is now [[1,1]]

myHashMap.get(2); // return -1 (i.e., not found), The map is now [[1,1]]

Practice- LeetCode #706 Design HashMap - Template

```
public class MyHashMap {

    public MyHashMap() {}

    public void put (int key, int value) {  }

    public int get(int key) {     }

    public void remove  int key) {    }

}

/**
 * Your MyHashMap object will be instantiated and called as such:
 * MyHashMap obj = new MyHashMap();
 * obj.put(key,value);
 * int param_2 = obj.get(key);
 * obj.remove(key);
 */
```

Practice- LeetCode #706 Design HashMap Solution

- The most distinguish characteristic about hashmap is that it provides a fast access (O(1)) to a **value** that is associated with a given **key**.

- There are two main issues that we should tackle, in order to design an *efficient* hashmap data structure:

  1) *Hash function design*

  2) *Collision handling*.

- Depending on how we deal with each of the above two issues, we could have various implementation of Hashmap data structure.

Practice- LeetCode #706 Design HashMap – Our Approach

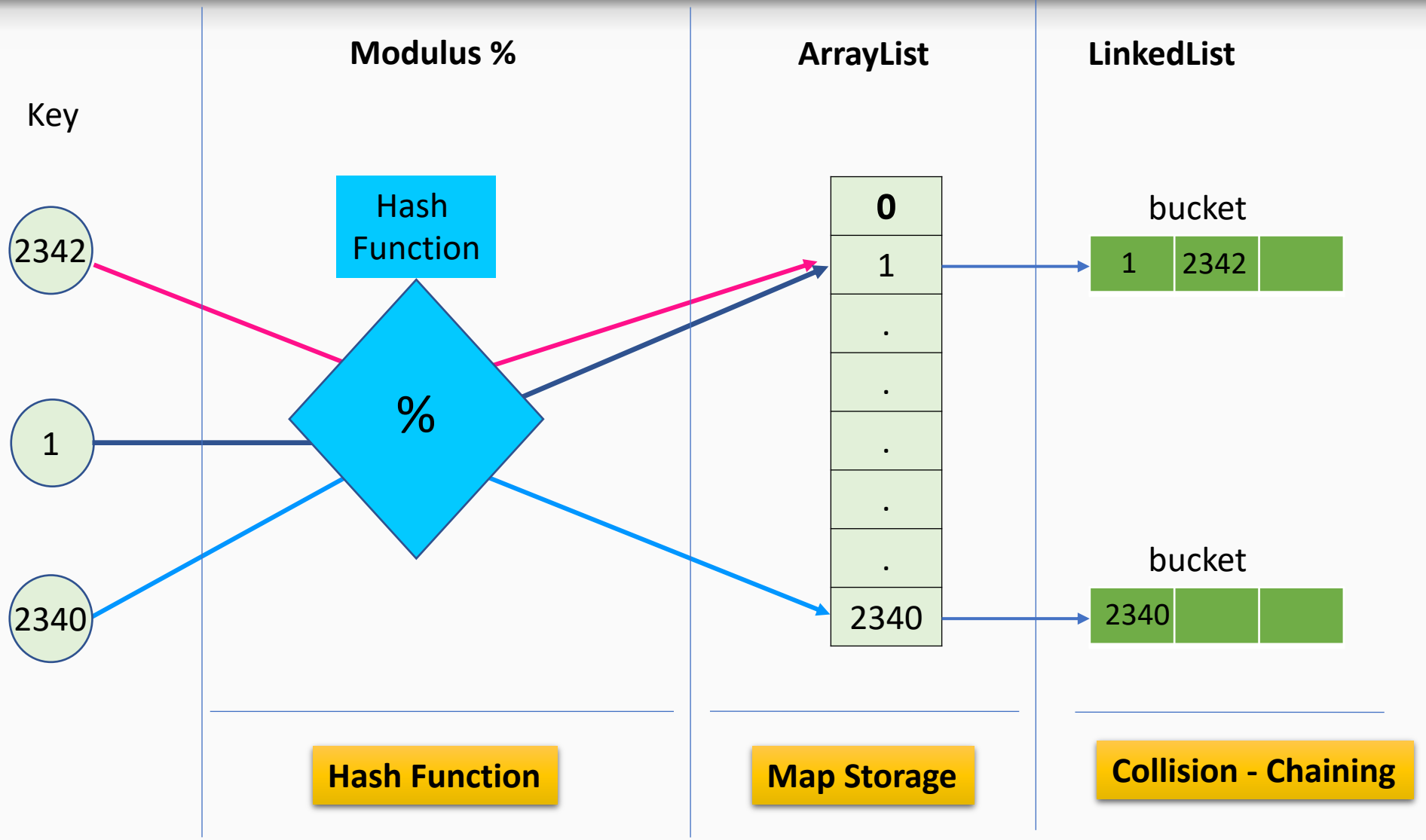
- We could adopt the modulus(%) operation as the hash function, since the key value is of integer type.
- The modulus operator (%)- or more precisely, the modulo operation - is a way to determine the remainder of a division operation. Instead of returning the result of the division, the modulo operation returns the whole number remainder.
- 5 % 1 = 0
  // 5 divided by 1 equals 5, with a remainder of 0

  5 % 2 = 1
  // 5 divided by 2 equals 2, with a remainder of 1

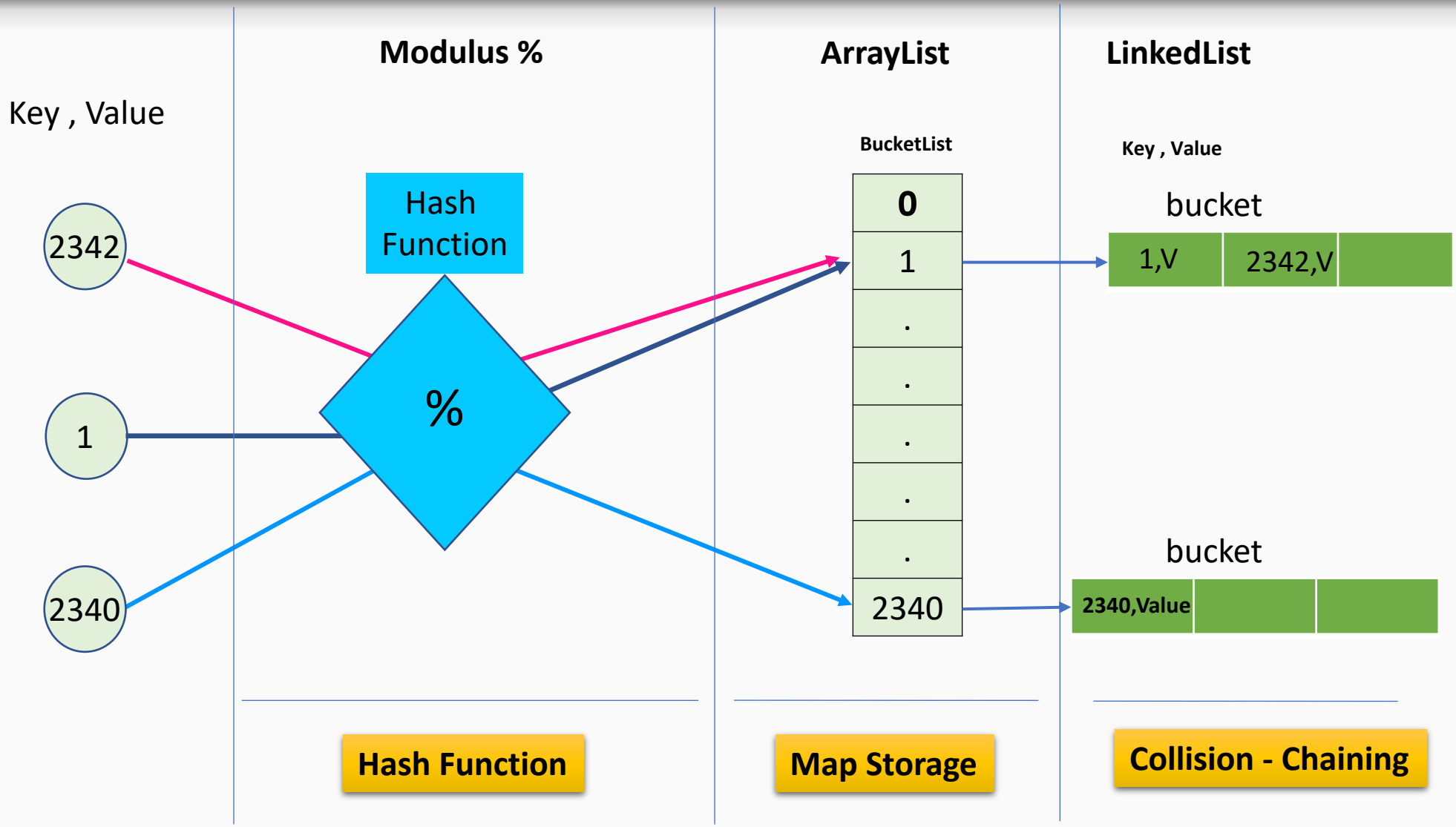
  5 % 3 = 2
  // 5 divided by 3 equals 1, with a remainder of 2

  In addition, in order to minimize the potential collisions, it is advisable to use a prime number as the base of modulus operation. e.g. 2341

Practice- LeetCode #706 Design HashMap – Our Approach



Practice- LeetCode #706 Design HashMap – Our Approach



Questions?