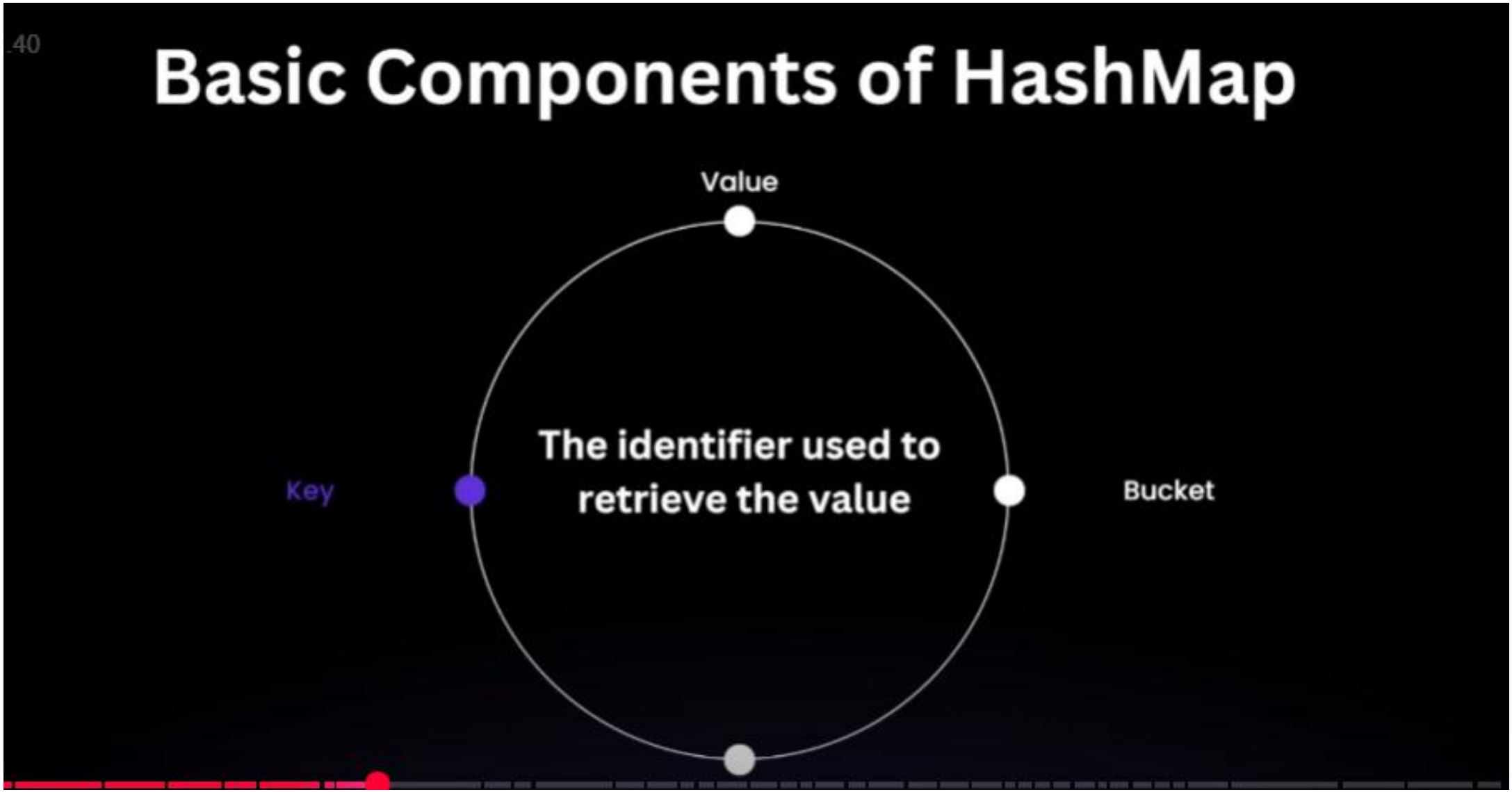
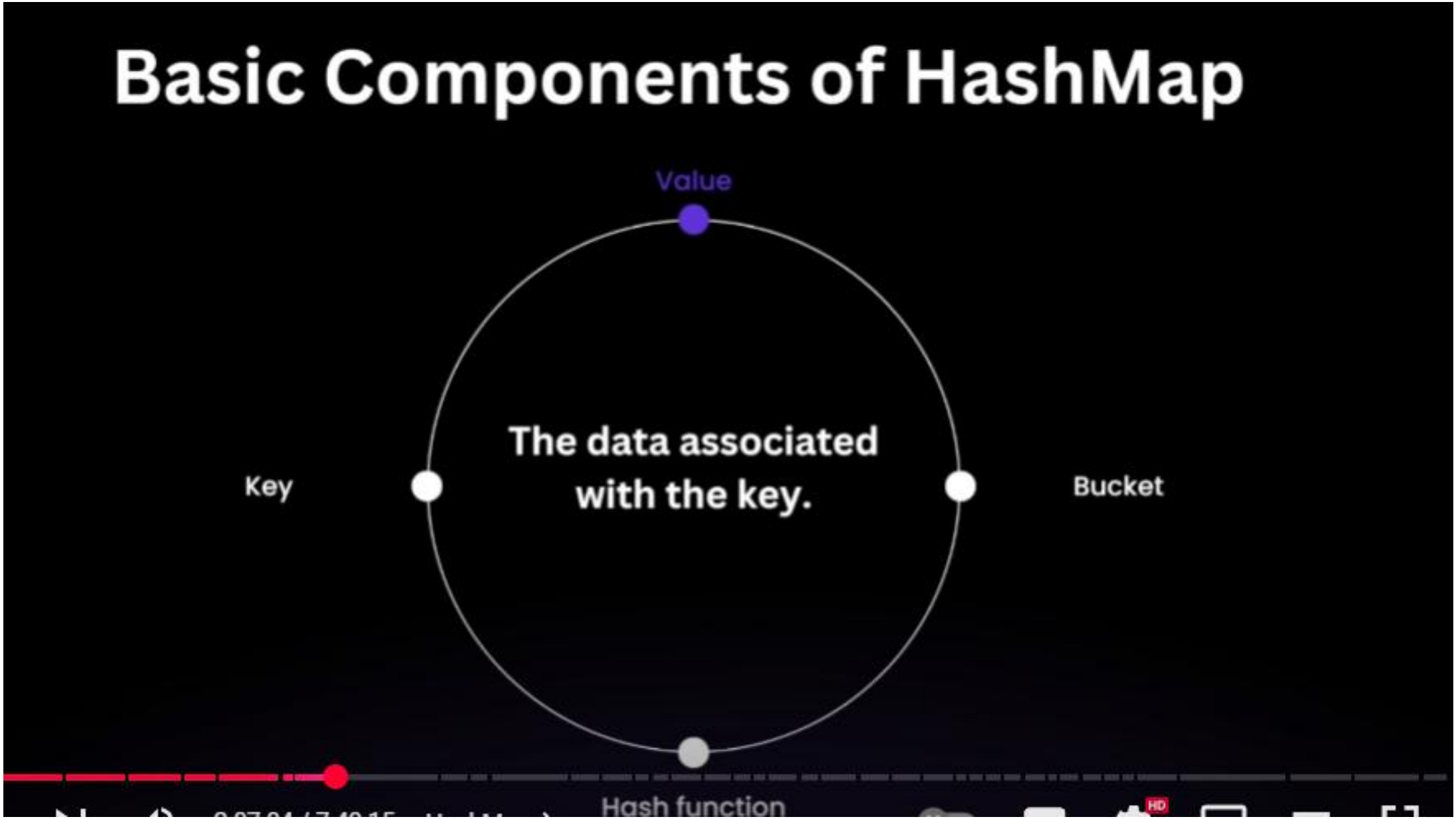
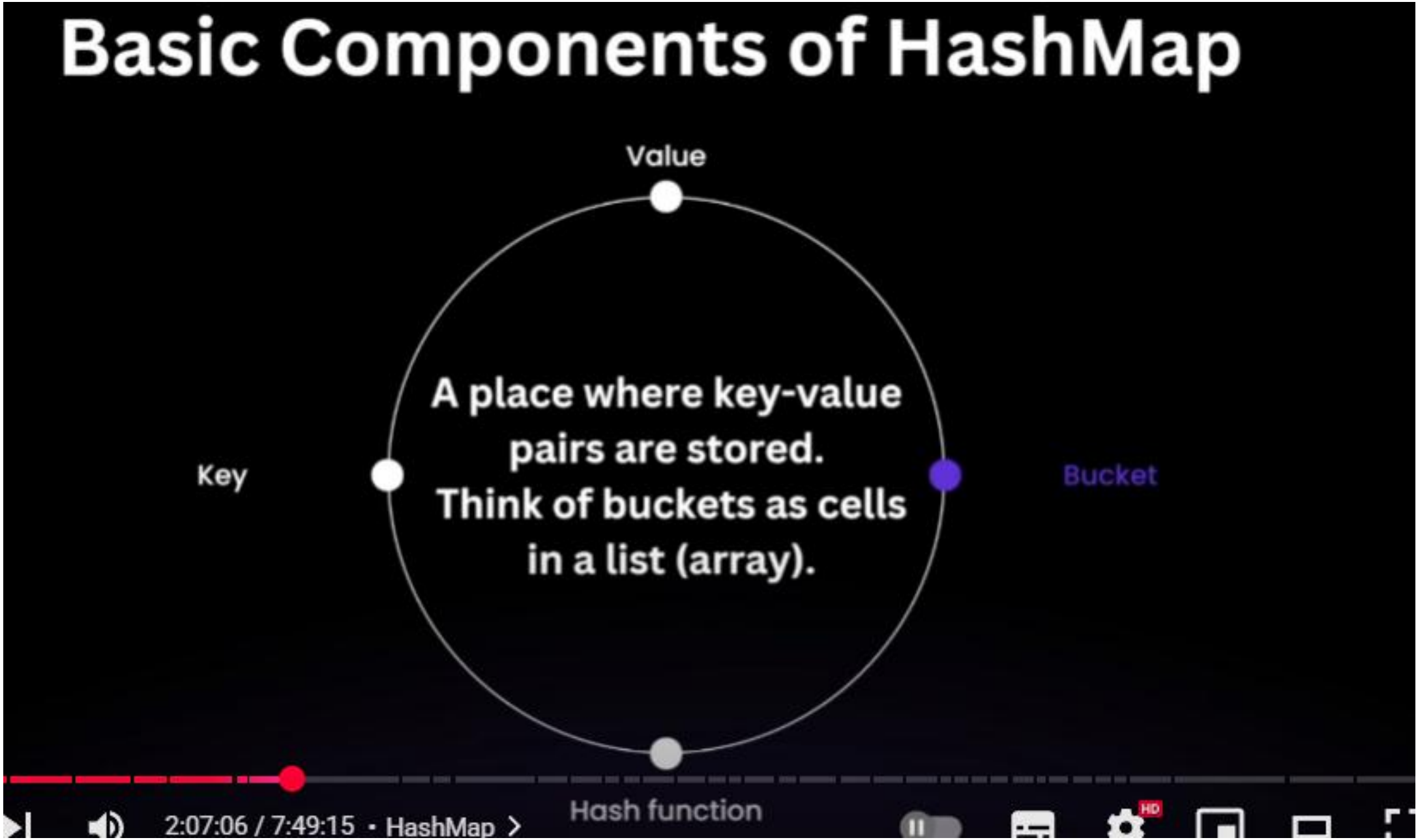
1.30

Key Characteristics

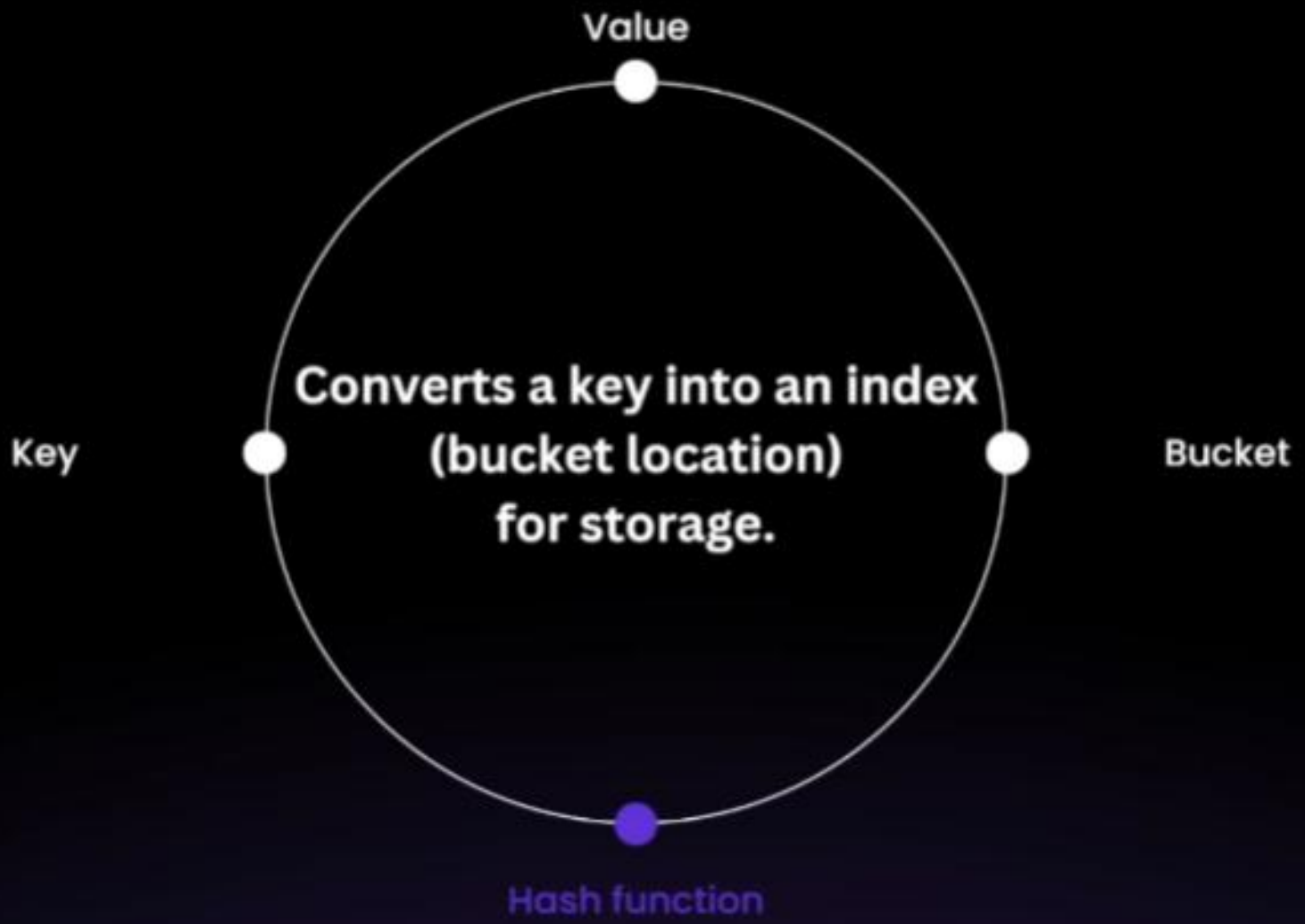
Unordered: Does not maintain any order of its elements.
Allows null Keys and Values: Can have one null key and multiple null values.
Not Synchronized: Not thread-safe; requires external synchronization if used in a multi-threaded context.
Performance: Offers constant-time performance (O(1)) for basic operations like get and put, assuming the hash function disperses elements properly.


40
Basic Components of HashMap
Value
Key
The identifier used to retrieve the value
Bucket


Basic Components of HashMap
Value
The data associated with the key.
Key
Bucket
Hash function


Basic Components of HashMap
Value
A place where key-value pairs are stored. Think of buckets as cells in a list (array).
Key
Bucket
Hash function
2:07:06 / 7:49:15 • HashMap >

Basic Components of HashMap



A hash function is an algorithm that takes an input (or "key") and returns a fixed-size string of bytes, typically a numerical value. The output is known as a hash code, hash value, or simply hash. *The primary purpose of a hash function is to map data of arbitrary size to data of fixed size*

- *Deterministic:* The same input will always produce the same output.
- *Fixed Output Size:* Regardless of the input size, the hash code has a consistent size (e.g., 32-bit, 64-bit).
- *Efficient Computation:* The hash function should compute the hash quickly.

How Data is Stored in HashMap

Step 1: Hashing the Key

First, the key is passed through a hash function to generate a unique hash code (an integer number). This hash code helps determine where the key-value pair will be stored in the array (called a "bucket array").

40

Step 2: Calculating the Index

The hash code is then used to calculate an index in the array (bucket location) using

```
int index = hashCode % arraySize;
```

The index decides which bucket will hold this key-value pair.

For example, if the array size is 16, the key's hash code will be divided by 16, and the remainder will be the index.

Step 3: Storing in the Bucket

The key-value pair is stored in the bucket at the calculated index. Each bucket can hold multiple key-value pairs

(this is called a collision handling mechanism, discussed later).

map.put("apple", 50);

- "apple" is the key.
- 50 is the value.
- The hash code of "apple" is calculated.
- The index is found using the hash code.
- The pair ("apple", 50) is stored in the corresponding bucket.

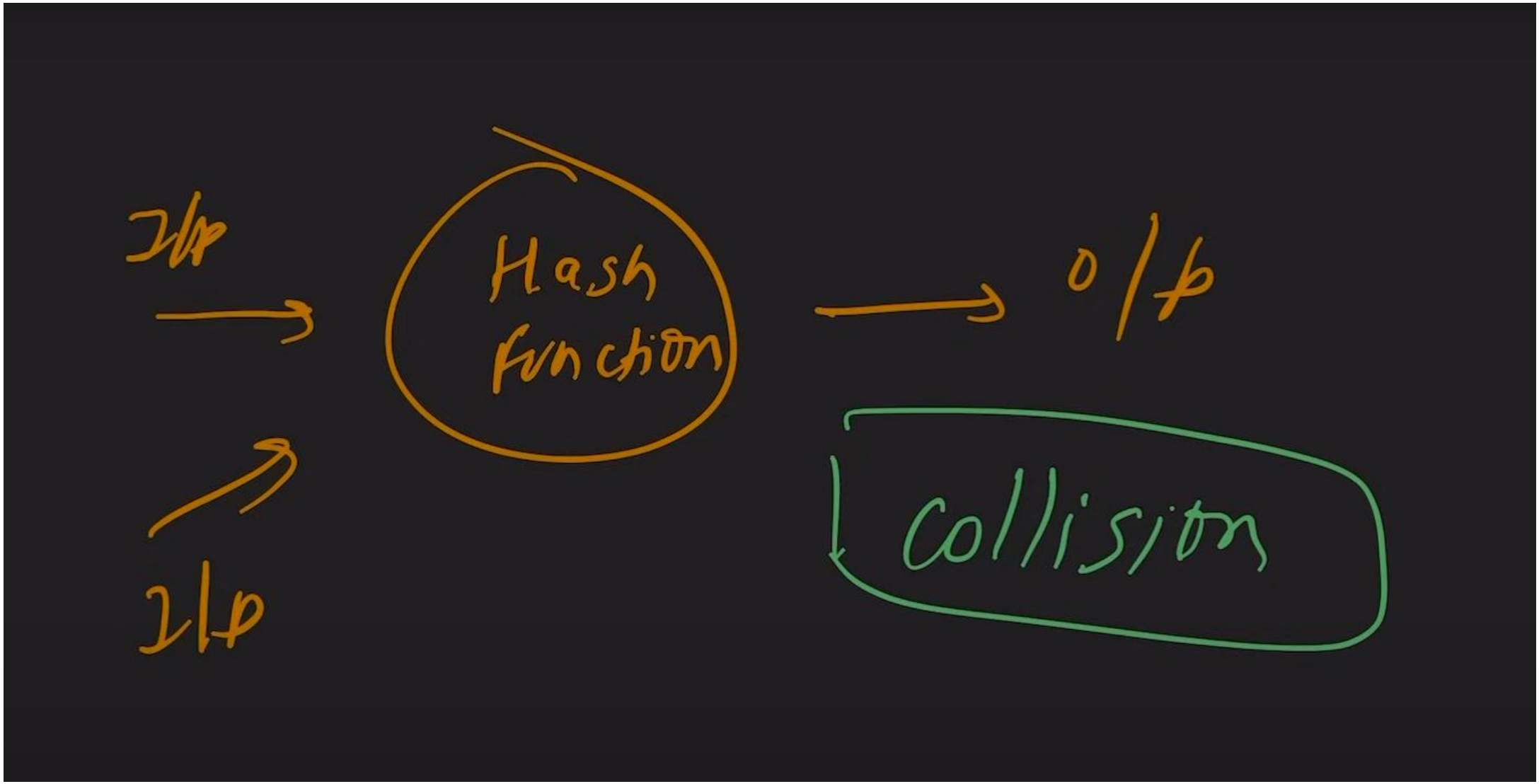
How HashMap Retrieves Data

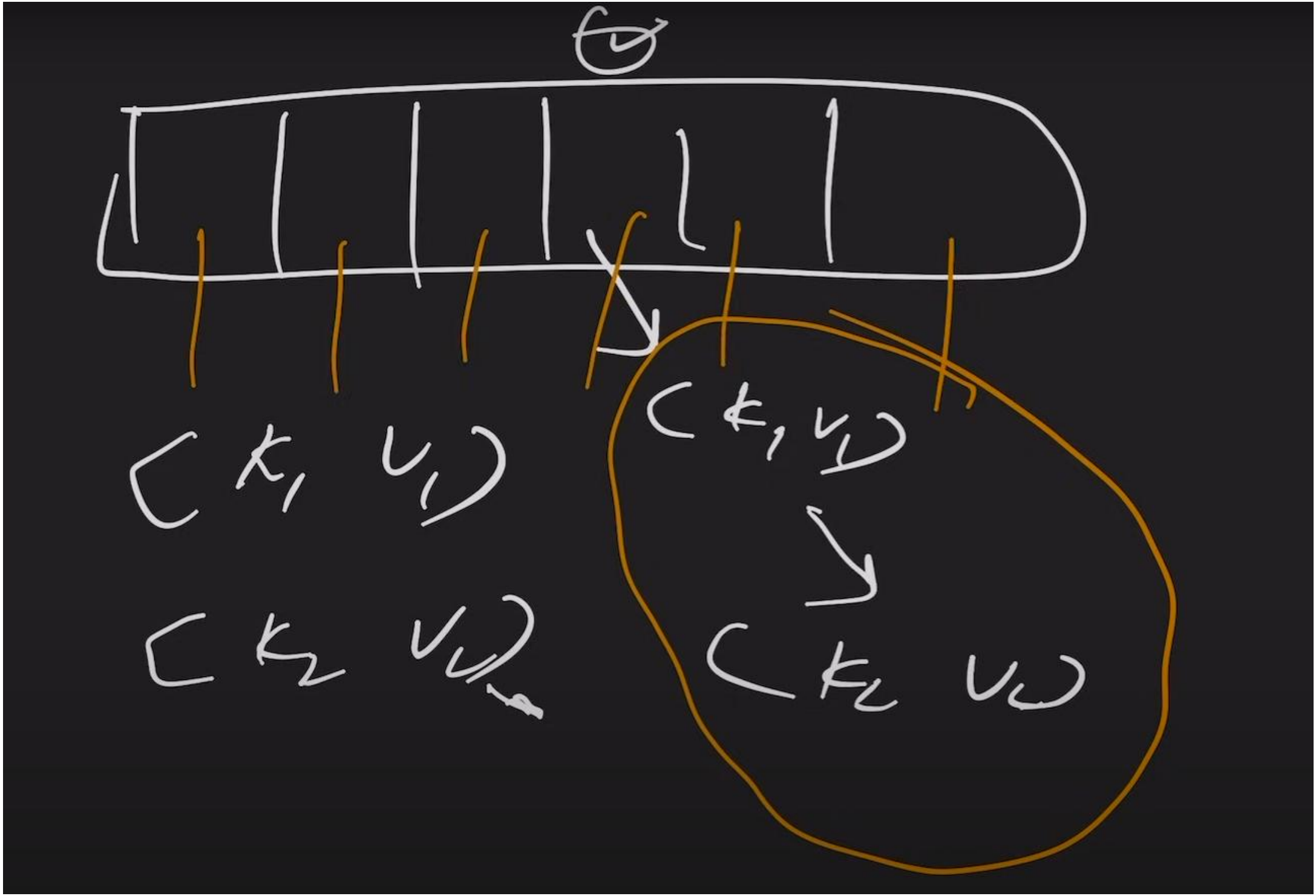
When we call get(key), the HashMap follows these steps:

Hashing the Key: Similar to insertion, the key is hashed using the same hash function to calculate its hash code.

Finding the Index: The hash code is used to find the index of the bucket where the key-value pair is stored.

Searching in the Bucket: Once the correct bucket is found, it checks for the key in that bucket. If it finds the key, it returns the associated value.


I/p
I/p
Hash function
o/p
collision


(k1, v1)
(k2, v2)
(k1, v1)
(k2, v2)

```java
class Node<K, V> {
    final int hash;   // hash code of the key
    final K key;      // the key itself
    V value;          // the value associated with the key
    Node<K, V> next;  // pointer to the next node in case of a collision (linked list)
}
```

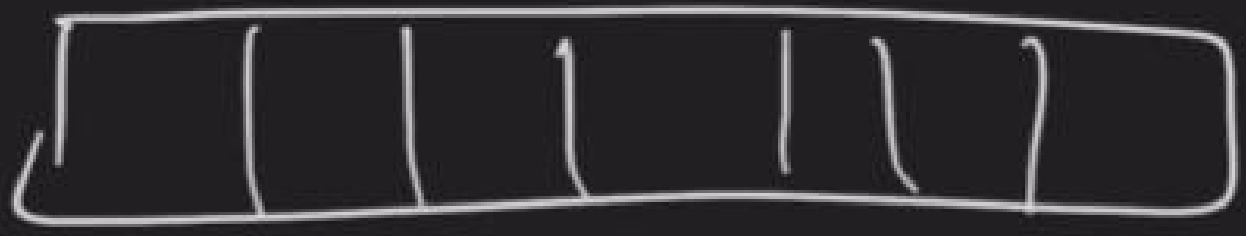
1.40

Handling Collisions

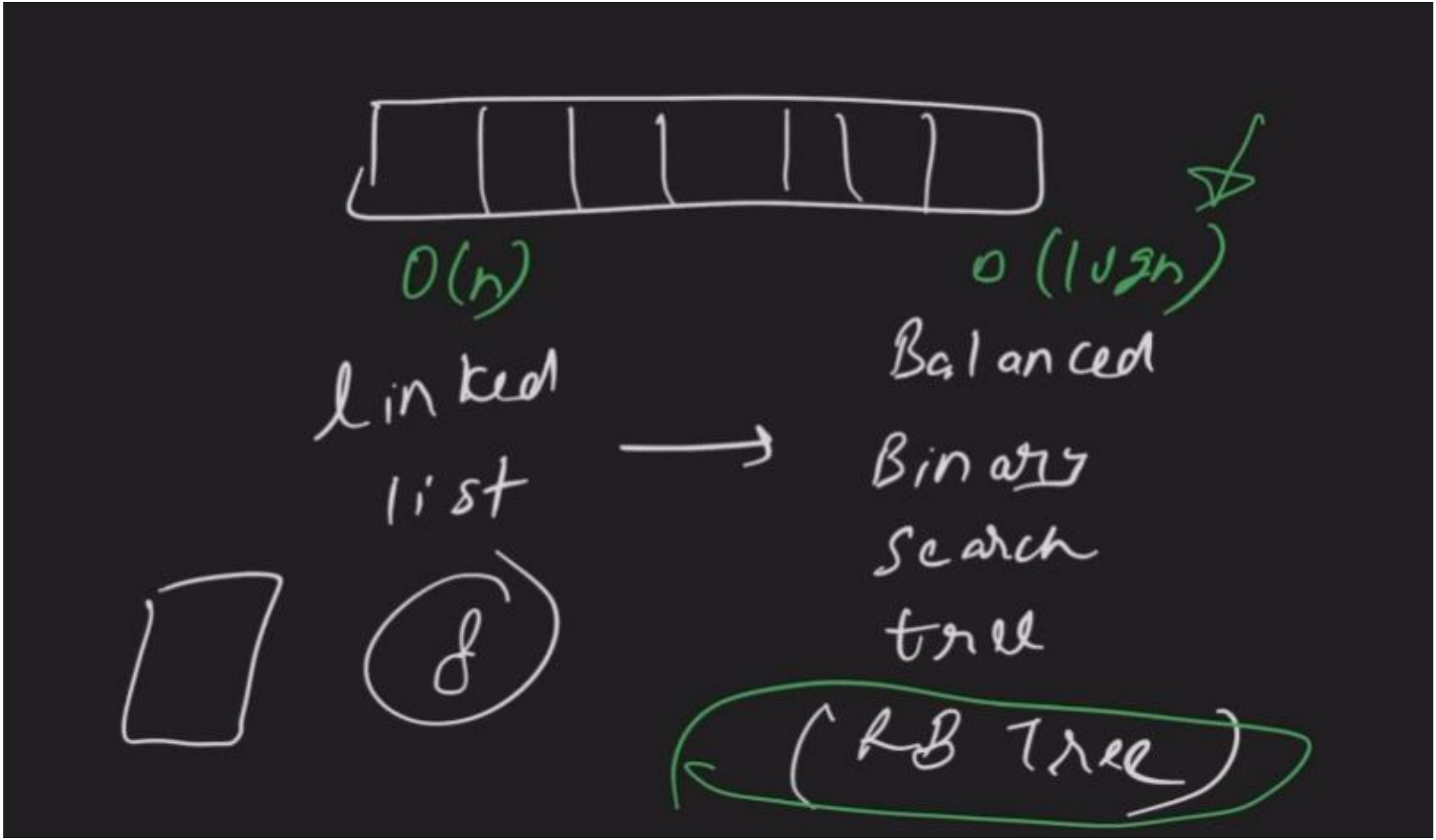
Since different keys can generate the same index (called a collision), HashMap uses a technique to handle this situation. Java's HashMap uses Linked Lists (or balanced trees after Java 8) for this.

If multiple key-value pairs map to the same bucket, they are stored in a linked list inside the bucket.

When a key-value pair is retrieved, the HashMap traverses the linked list, checking each key until it finds a match.



Linked list → Balanced Binary Search tree (RB Tree)


O(n)
O (log n)
Linked list
Balanced Binary Search tree
(RB Tree)

Handling Collisions

```
map.put("apple", 50);
map.put("banana", 30);
map.put("orange", 80);
```

Let's say "apple" and "orange" end up in the same bucket due to a hash collision. They will be stored in a linked list in that bucket:

Bucket 5: ("apple", 50) -> ("orange", 80)

When we do map.get("orange"), HashMap will go to Bucket 5 and then traverse the linked list to find the entry with the key "orange".

HashMap Resizing (Rehashing)

HashMap has an internal array size, which by default is 16.
When the number of elements (key-value pairs) grows and exceeds a certain load factor (default is 0.75), HashMap automatically resizes the array to hold more data. This process is called rehashing.

The default size of the array is 16, so when more than 12 elements (16 * 0.75) are inserted, the HashMap will resize.

During rehashing

The array size is doubled.

1. All existing entries are rehashed (i.e., their positions are recalculated) and placed into the new array.
2. This ensures the HashMap continues to perform efficiently even as more data is added.

1.40

Time Complexity

HashMap provides constant time O(1) performance for basic operations like put() and get() (assuming no collisions). However, if there are many collisions, and many entries are stored in the same bucket, the performance can degrade to O(n), where n is the number of elements in that bucket.

But after Java 8, if there are too many elements in a bucket, HashMap switches to a balanced tree instead of a linked list to ensure better performance O(log n).

Suppose we want to store information about the number of fruits in a store.

Here's what we want to store:

Fruit	Quantity
Apple	50
Banana	30
Orange	80
Grape	20

```
HashMap<String, Integer> fruitMap = new HashMap<>();
```

Let's add the key-value pairs one by one.

```
fruitMap.put("Apple", 50);
```

Internal Process

The key "Apple" is hashed using its hashCode(). Let's assume "Apple" generates a hashCode of 10832233 (this is just an example value).

The hashCode is used to calculate the index in the internal array (bucket array). Let's say the array size is initially 16.

```
index = hashCode % arraySize;
```

```
index = 10832233 % 16 = 9;
```

This means "Apple" will be stored in bucket 9

```
fruitMap.put("Banana", 30);
index = 13942244 % 16 = 4;
```

```
fruitMap.put("Grape", 20);
index = 548734 % 16 = 14;
```

```
fruitMap.put("Orange", 80);
index = 19332414 % 16 = 14;
```

```
fruitMap.put("Grape", 20);
index = 548734 % 16 = 14;
```

```
fruitMap.put("Orange", 80);
index = 19332414 % 16 = 14;
```

```
fruitMap.put("Grape", 20);
index = 548734 % 16 = 14;
```

```
fruitMap.put("Orange", 80);
index = 19332414 % 16 = 14;
```

Since "Orange" is already stored in bucket 14, the HashMap handles the collision by adding "Grape" to the linked list in bucket 14.

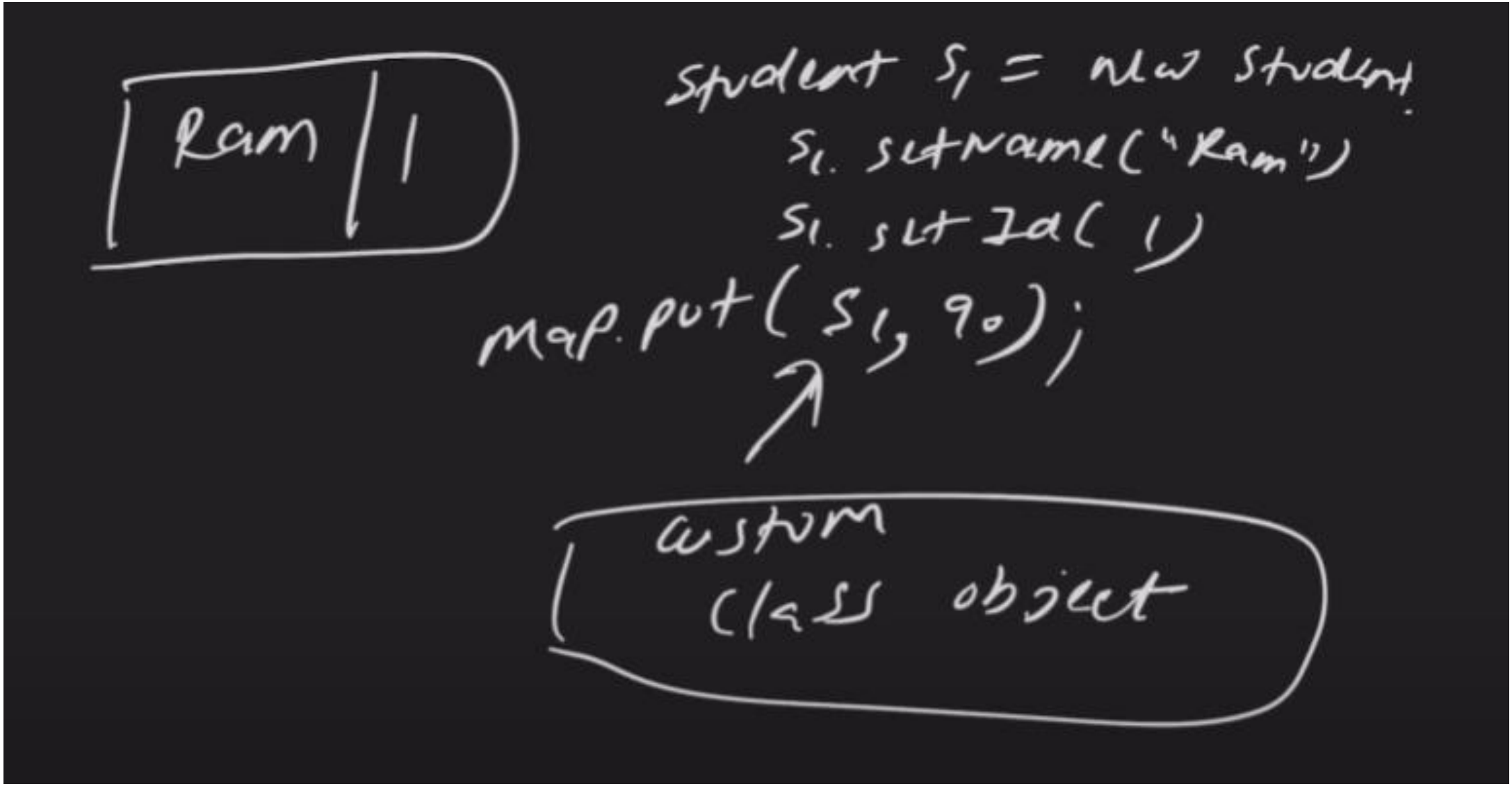
Now, bucket 14 contains two entries: ("Orange", 80) and ("Grape", 20).

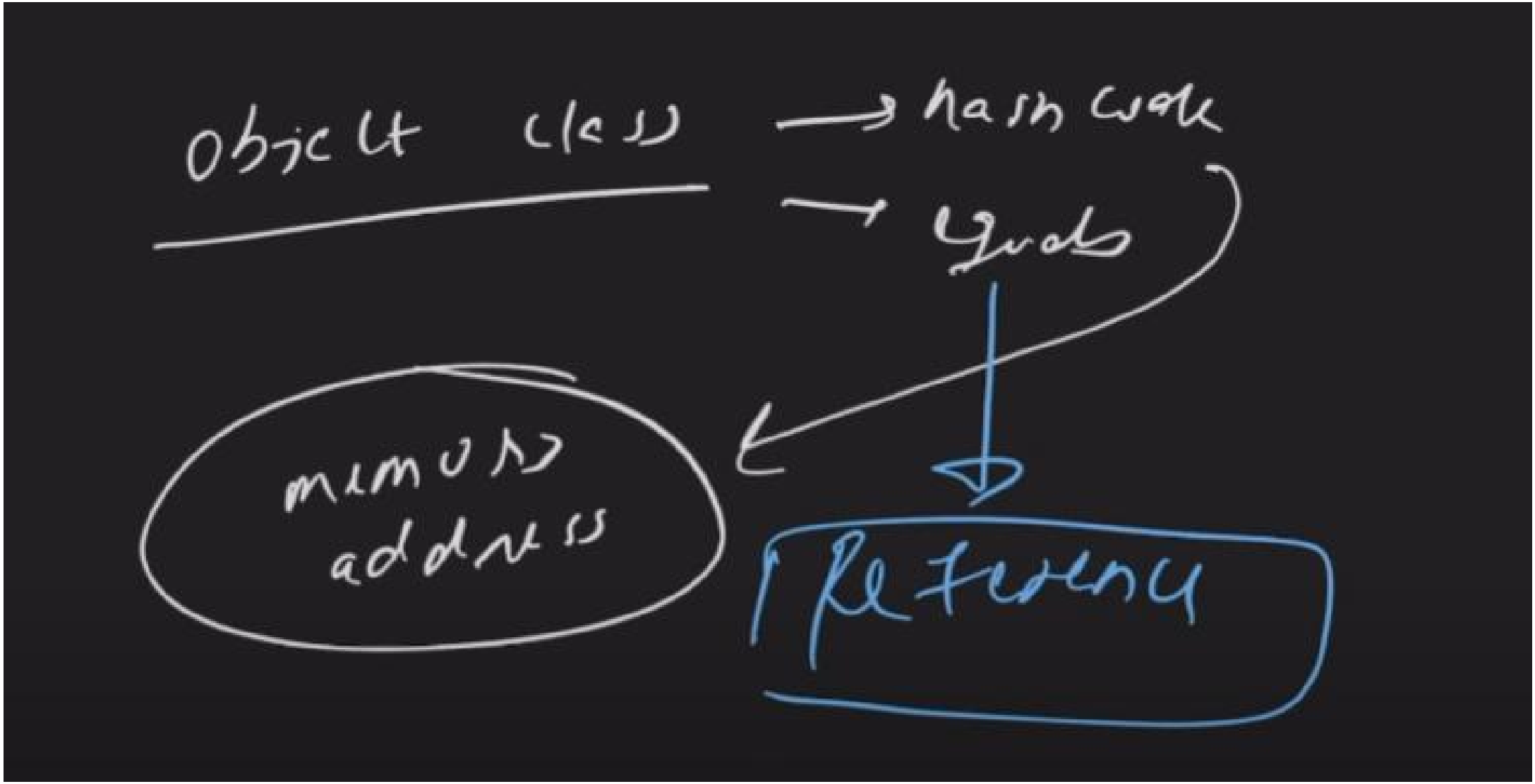
```
HashMap Structure (Array of Buckets, size: 16)

Index | Bucket (Key-Value Pairs)
-------------------------------------------------------------

  0   |
  1   |
  2   |
  3   |
  4   | ("Banana", 30)
  5   |
  6   |
  7   |
  8   |
  9   | ("Apple", 50)
 10   |
 11   |
 12   |
 13   |
 14   | ("Orange", 80) -> ("Grape", 20)   // Collision: stored in a linked list
 15   |
```

```
Student {
    int id;
    String name;
}

HashMap<Student,
```

HashMap<Student, Integer>

(Ram, 1)	90
(Shyam, 2)	89
(Neha, 3)	91


Ram | 1
Student S1 = new Student
S1. setName("Ram")
S1. setId(1)
map.put(S1, 90);
custom
class object


Object class
hash code
equals
memory address
Reference

Operation	Average-Case Time Complexity	Worst-Case Time Complexity	Explanation
put(key, value)	O(1)	O(log n)	Inserts a key-value pair. Average: Constant time due to direct bucket access. Worst-Case: O(log n) when bucket converts to a Red-Black Tree after exceeding collision threshold.
get(key)	O(1)	O(log n)	Retrieves the value associated with a key. Average: Constant time via direct bucket access. Worst-Case: O(log n) when searching within a treeified bucket.
remove(key)	O(1)	O(log n)	Removes the key-value pair associated with a key. Average: Constant time with direct access. Worst-Case: O(log n) when removing from a treeified bucket.

containsKey(key)	O(1)	O(log n)	Checks if a key exists in the map. Average: Constant time via direct bucket access. Worst-Case: O(log n) when searching within a treeified bucket.
containsValue(value)	O(n)	O(n)	Checks if a value exists in the map. Both average and worst-case are linear time since it may need to traverse all entries.
size()	O(1)	O(1)	Returns the number of key-value pairs. Both average and worst-case are constant time as the size is maintained as a separate field.