HOW HASH MAP WORKS IN JAVA

Index

- Intro

- *Map* is an array that lets you store *key-value* associations. This means that it's possible to store a key against value and then to look up the value we can use the key.

- ***hashing***: Is the transformation of a string of characters (text) to a shorted fixed-length value that represents the origin string. A shorter value helps in indexing and faster searches.

- It's impossible to produce the same hash value entering different inputs and therefore the same message always produces the same hash value.

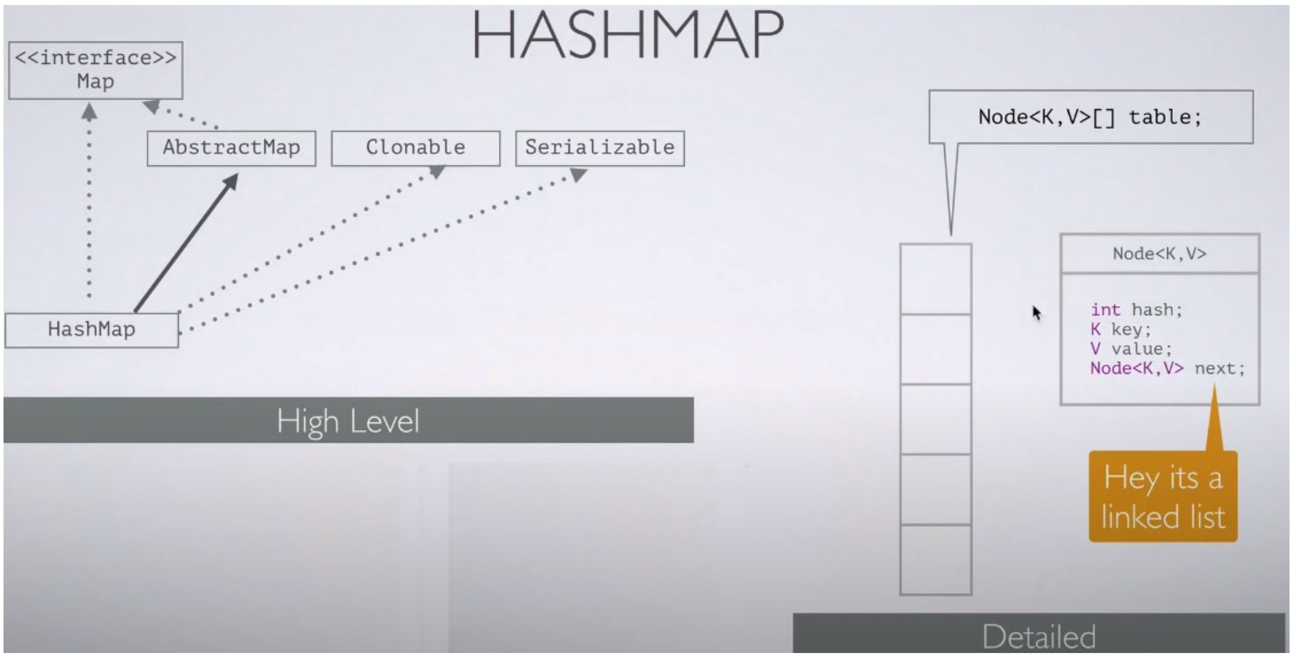
- In Java every object has a method *hashCode* that will return a hash value for a given object.

- In Java there is an *equal* and *hashCode* contract that says that if two objects are equal, then they should have the same hash code as well. That means that is very important to have a robust hash code implementation.

- HashMap structure

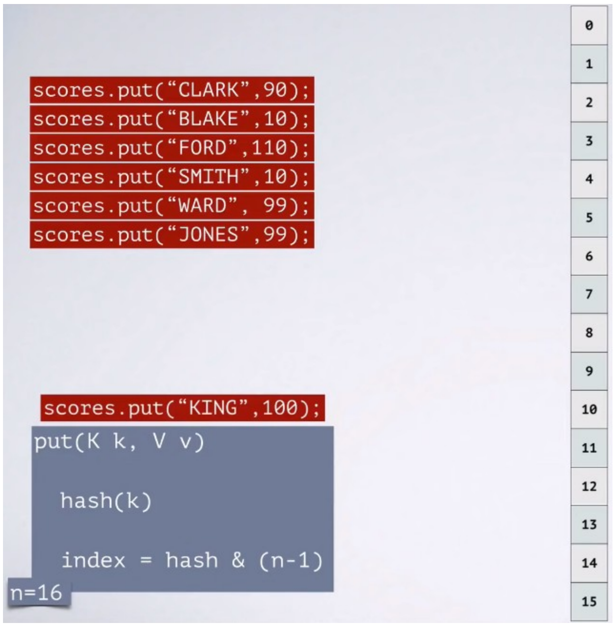
- Hashmap is an array in which each element is a LinkedList of type *Node<K,V>* class composed for:
  - int hash
  - K key (passed to the hashmap)
  - V value
  - Node<K,V> next : pointer to the next element.

- Each index in the array is known as a *bucket* and each *bucket* is a *Node* which in turn can be a *LinkedList* of nodes.



- How *put* operation works

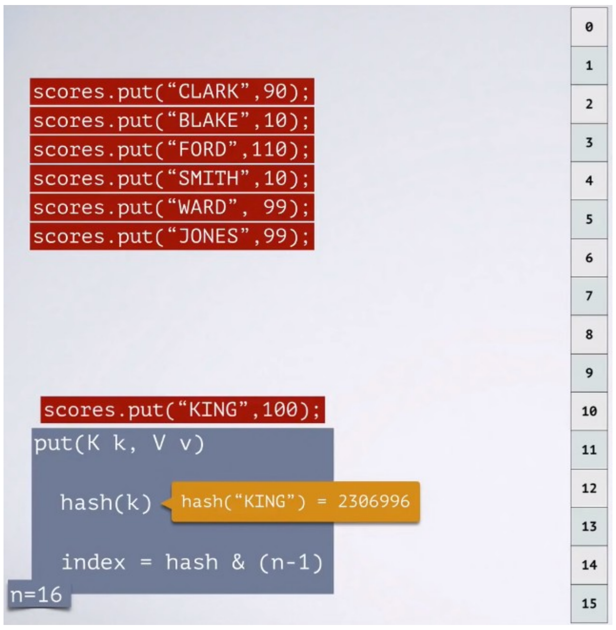
- The array is sized to 2^m and by default, a HashMap comes with an array of size 16, so the index of the table ranges from 0 to 15.
- Java HashMap allows *null* key, which always goes to index 0 as hash of null is 0.
- In this example we are trying to put a person name and his score into a HashMap.
- First we are trying to put the key *KING* and value 100 into the hash map.



```
scores.put("CLARK",90);
scores.put("BLAKE",10);
scores.put("FORD",110);
scores.put("SMITH",10);
scores.put("WARD", 99);
scores.put("JONES",99);

scores.put("KING",100);
put(K k, V v)

  hash(k)

  index = hash & (n-1)
n=16
```

- The *put* map API is called and it basically computes a hash of the key which is 2306996 for *KING*.
- There can not be an array with the size corresponding to the hashed key because if each and every HashMap is going to have such a huge array within it, there will be an out of memory error.
- There is an index computation to find out where exactly we can fit the hash code in the array of range 0..15.

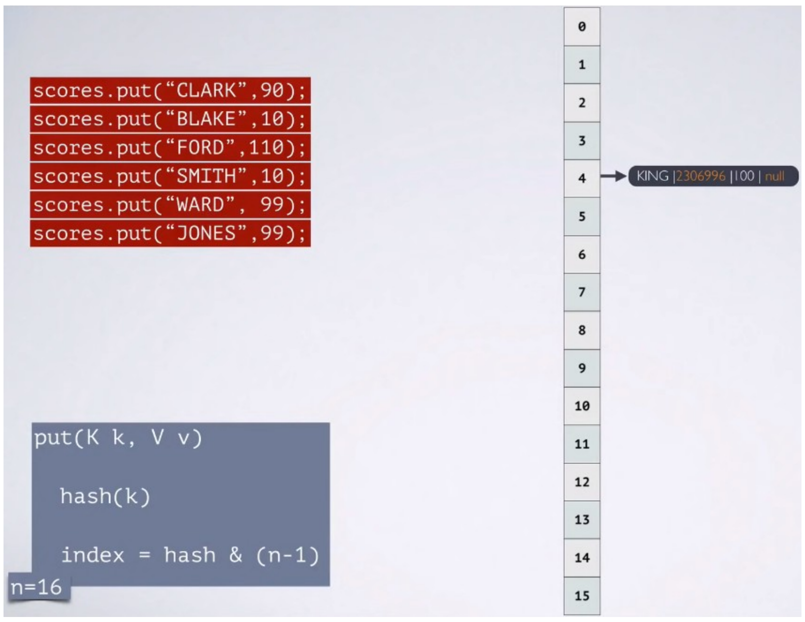
- The index is computed using a **modular operation** that’s the equivalent of dividing the hash code and the maximum value of the range and getting the reminder that will be the index of the array to place the element.



```
scores.put(“CLARK”,90);
scores.put(“BLAKE”,10);
scores.put(“FORD”,110);
scores.put(“SMITH”,10);
scores.put(“WARD”, 99);
scores.put(“JONES”,99);

scores.put(“KING”,100);
put(K k, V v)

  hash(k)   hash(“KING”) = 2306996

  index = hash & (n-1)
n=16
```

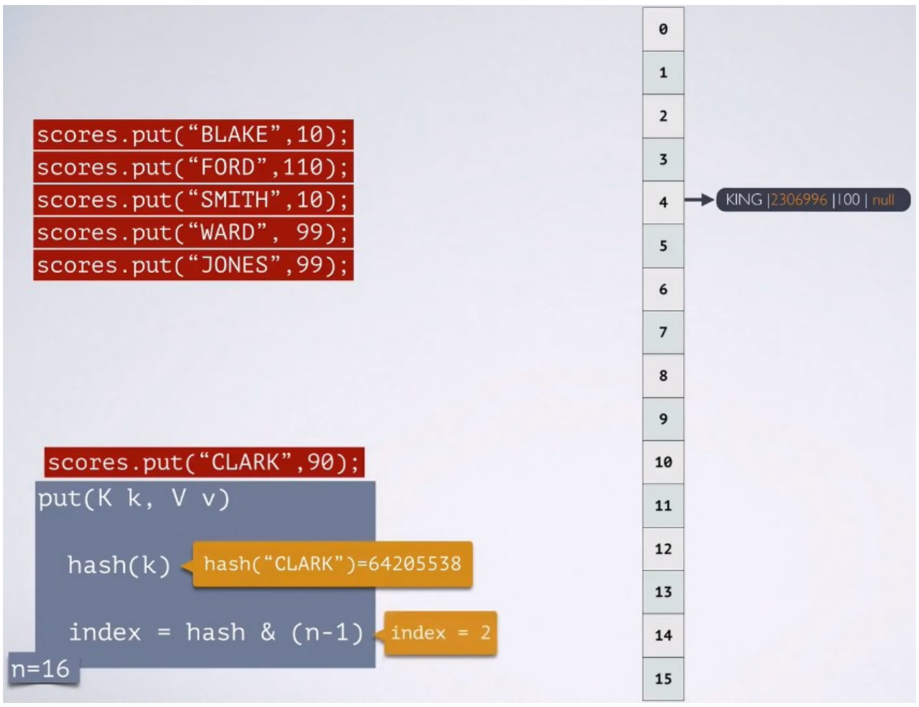
index = 2306996 mod 16 = 4

```
1000110011001110110100 → 2306996
                  1111 →  15 (16-1)
                  0100 →   4
```



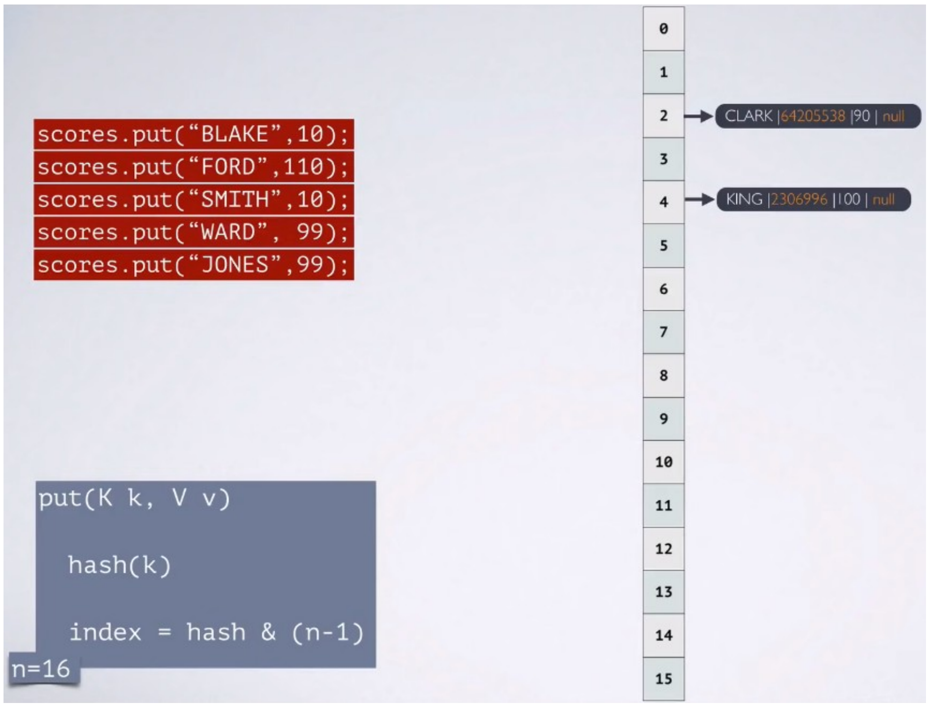
```
scores.put(“CLARK”,90);
scores.put(“BLAKE”,10);
scores.put(“FORD”,110);
scores.put(“SMITH”,10);
scores.put(“WARD”, 99);
scores.put(“JONES”,99);

put(K k, V v)

  hash(k)

  index = hash & (n-1)
n=16
```

- Now we proceed with *CLARK* which computed hash is 64205538 and the corresponding index should be 2 (64205538 mod 16).



```
scores.put("BLAKE",10);
scores.put("FORD",110);
scores.put("SMITH",10);
scores.put("WARD", 99);
scores.put("JONES",99);
```

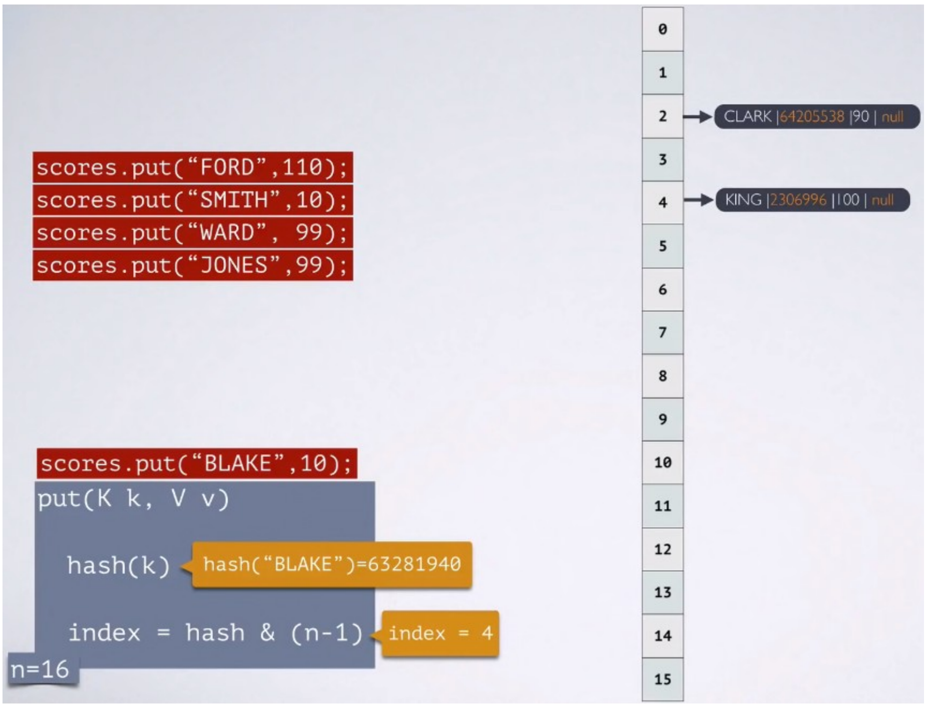
```
scores.put("CLARK",90);
put(K k, V v)

  hash(k)      hash("CLARK")=64205538

  index = hash & (n-1)    index = 2
n=16
```



```
scores.put("BLAKE",10);
scores.put("FORD",110);
scores.put("SMITH",10);
scores.put("WARD", 99);
scores.put("JONES",99);
```

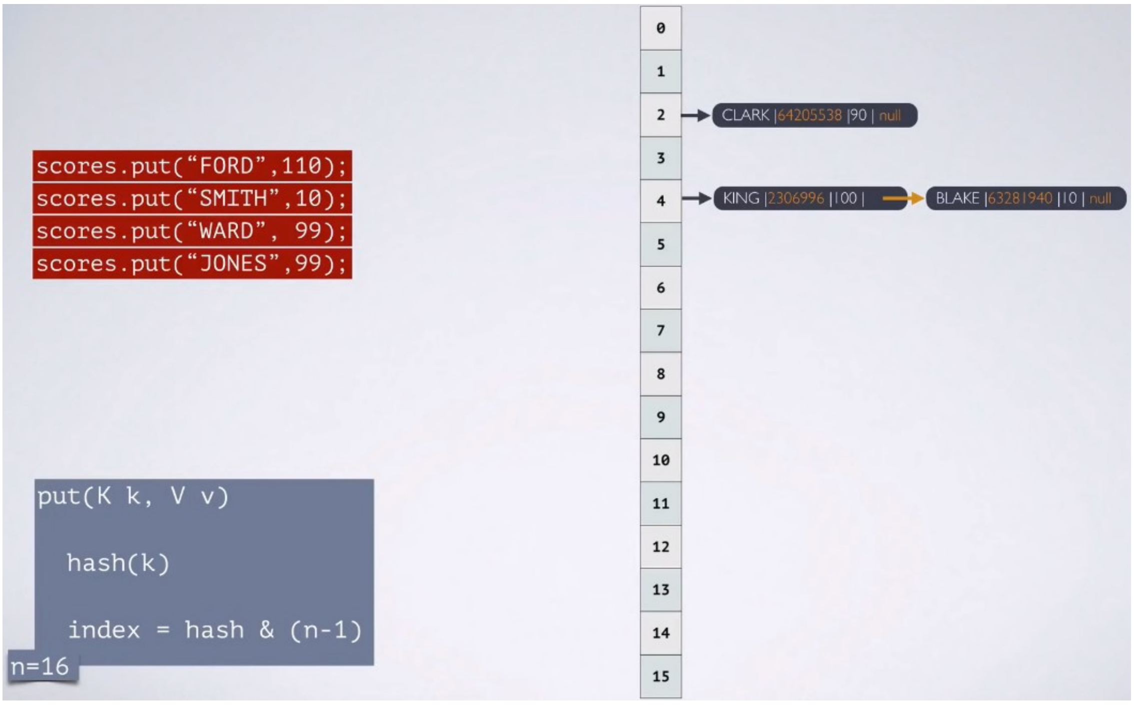
```
put(K k, V v)

  hash(k)

  index = hash & (n-1)
n=16
```

- Continue with BLAKE which has is 63281940 and the corresponding index should be 4 (63281940 mod 16).



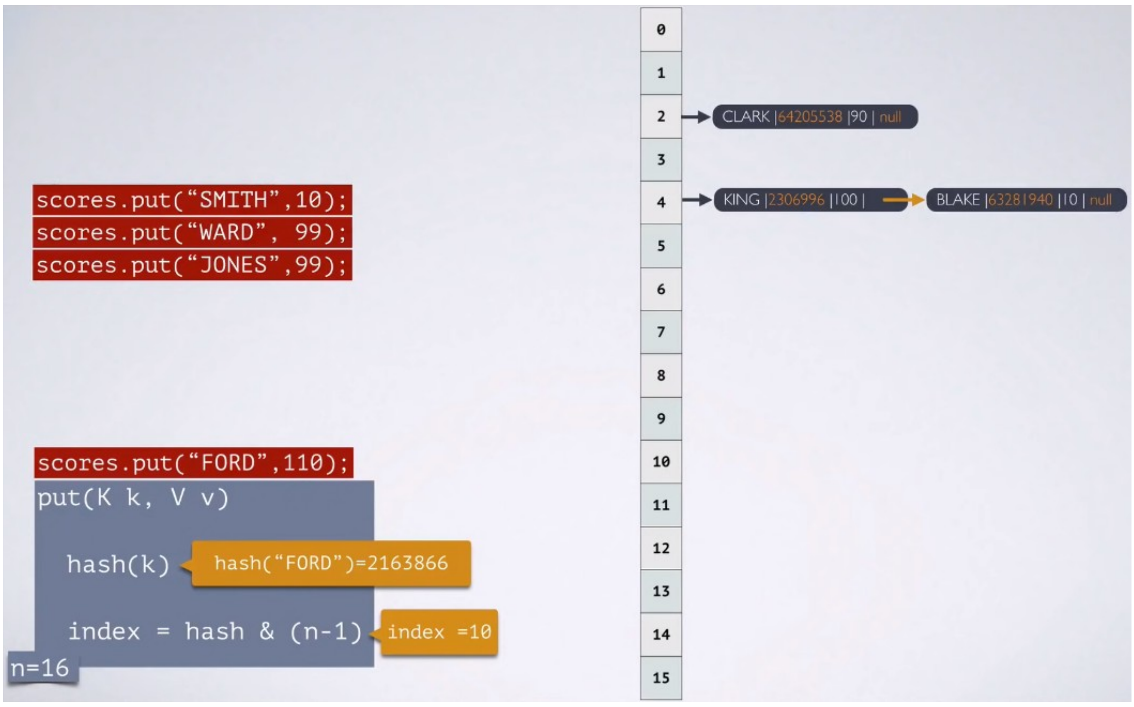
```
scores.put("FORD",110);
scores.put("SMITH",10);
scores.put("WARD", 99);
scores.put("JONES",99);

scores.put("BLAKE",10);
put(K k, V v)

  hash(k)                 hash("BLAKE")=63281940

  index = hash & (n-1)    index = 4
n=16
```

- Here we have a collision because we already have an entry at index 4 so what we would do is that this entry will be added as the next node of the already existing node at index 4.

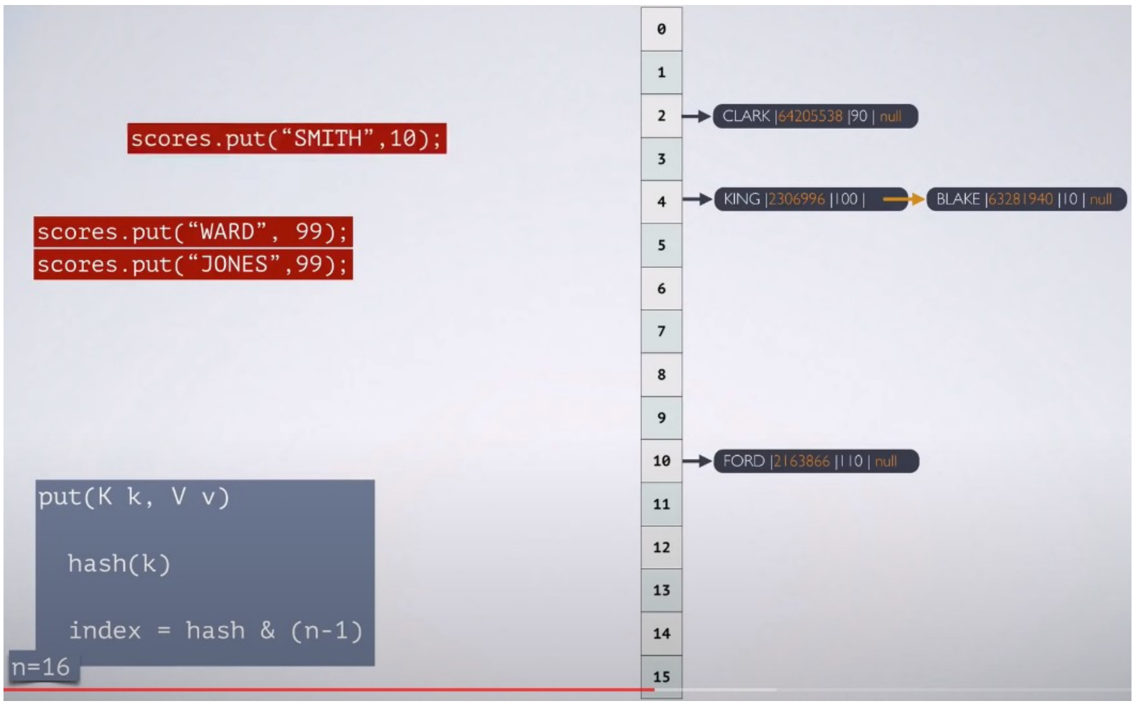


```
scores.put("FORD",110);
scores.put("SMITH",10);
scores.put("WARD", 99);
scores.put("JONES",99);

put(K k, V v)

  hash(k)

  index = hash & (n-1)
n=16
```

- For adding FORD:



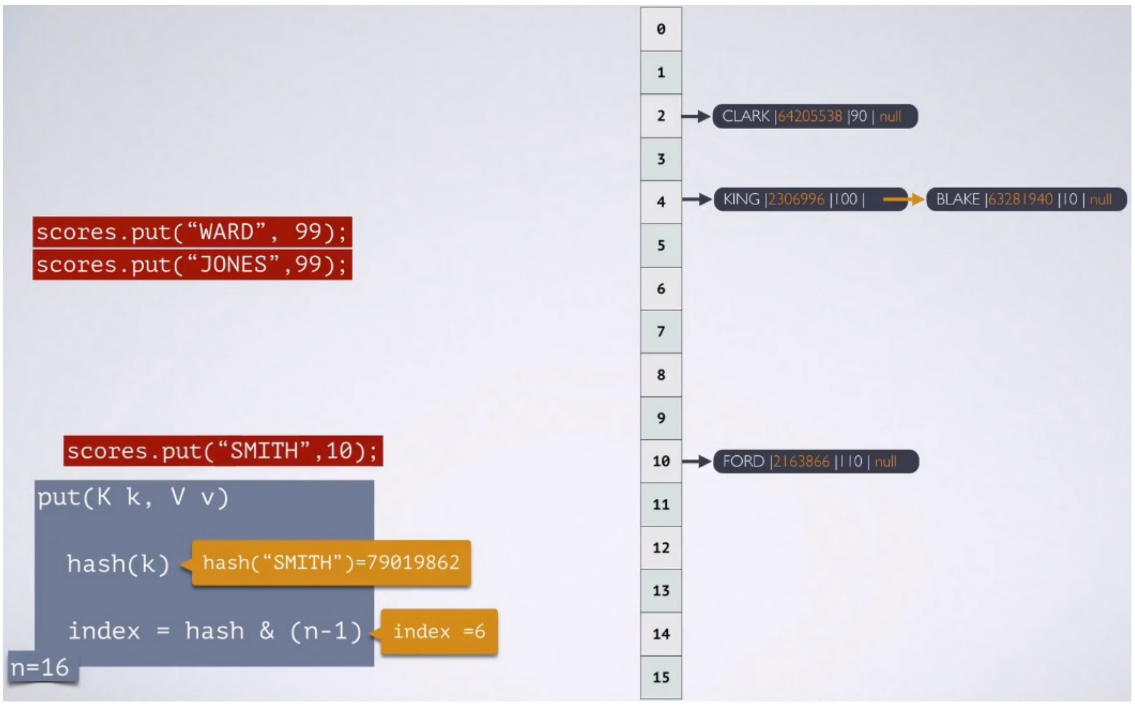
```
scores.put("SMITH",10);
scores.put("WARD", 99);
scores.put("JONES",99);

scores.put("FORD",110);
put(K k, V v)

  hash(k)            hash("FORD")=2163866

  index = hash & (n-1)   index =10
n=16
```



```
scores.put("SMITH",10);

scores.put("WARD", 99);
scores.put("JONES",99);

put(K k, V v)

  hash(k)

  index = hash & (n-1)
n=16
```

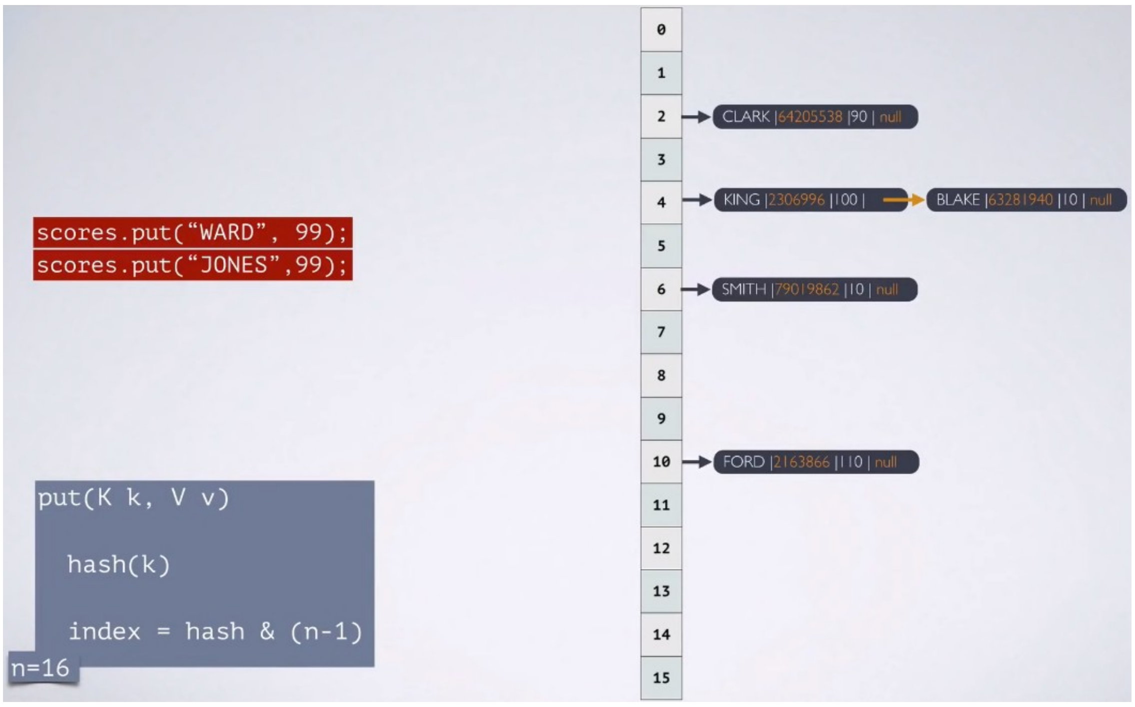
0
1
2
CLARK |64205538 |90 | null
3
4
KING |2306996 |100 |
BLAKE |63281940 |10 | null
5
6
7
8
9
10
FORD |2163866 |110 | null
11
12
13
14
15

- For SMITH:



```
scores.put("WARD", 99);
scores.put("JONES",99);

scores.put("SMITH",10);
put(K k, V v)
  hash(k)      hash("SMITH")=79019862
  index = hash & (n-1)     index =6
n=16
```

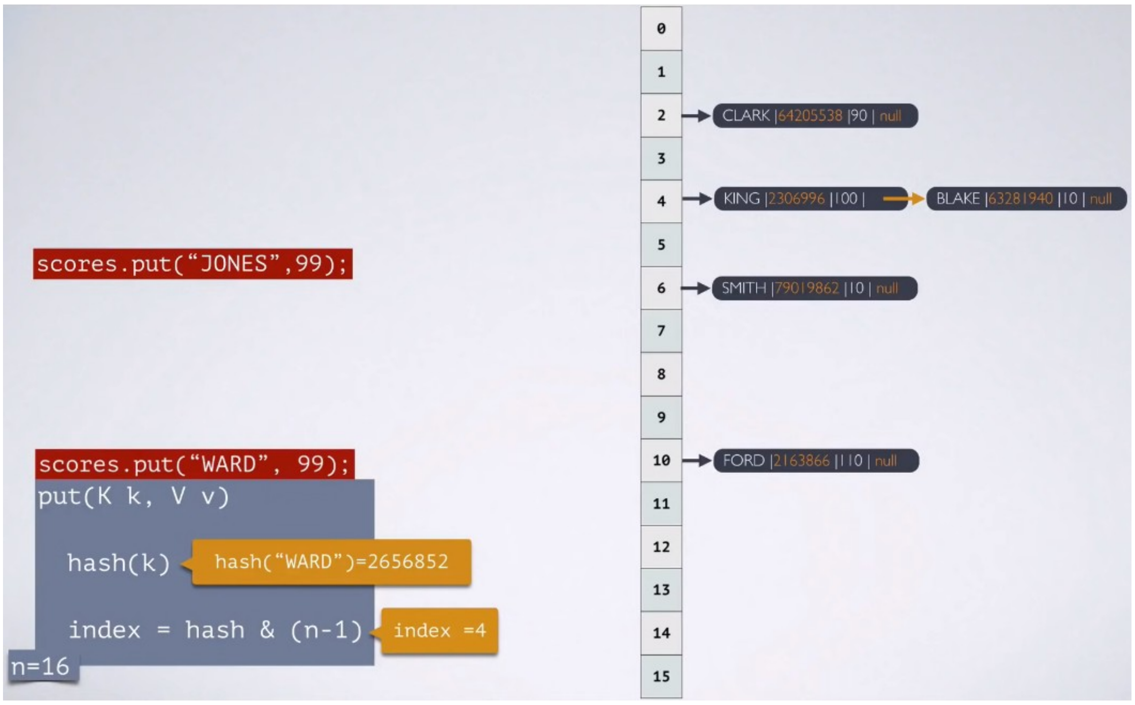
0
1
2
CLARK |64205538 |90 | null
3
4
KING |2306996 |100 |
BLAKE |63281940 |10 | null
5
6
7
8
9
10
FORD |2163866 |110 | null
11
12
13
14
15



```
scores.put("WARD", 99);
scores.put("JONES",99);

put(K k, V v)
  hash(k)
  index = hash & (n-1)
n=16
```

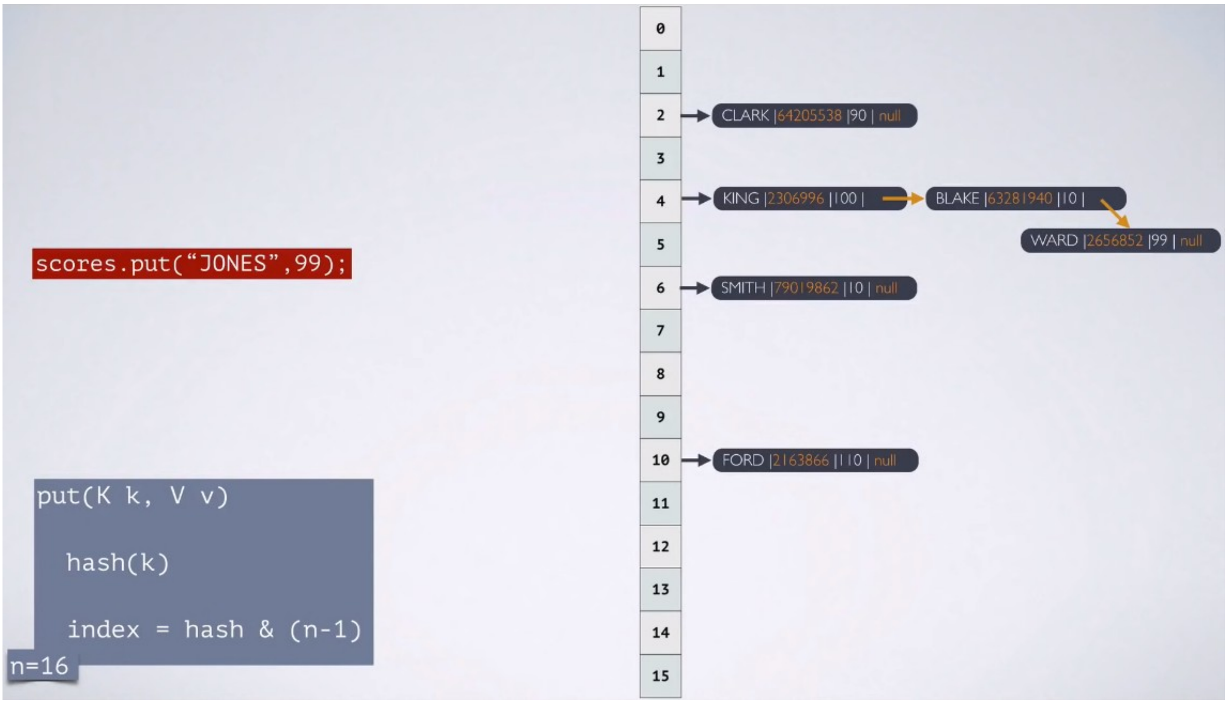
0
1
2
CLARK |64205538 |90 | null
3
4
KING |2306996 |100 |
BLAKE |63281940 |10 | null
5
6
SMITH |79019862 |10 | null
7
8
9
10
FORD |2163866 |110 | null
11
12
13
14
15

- For WARD:



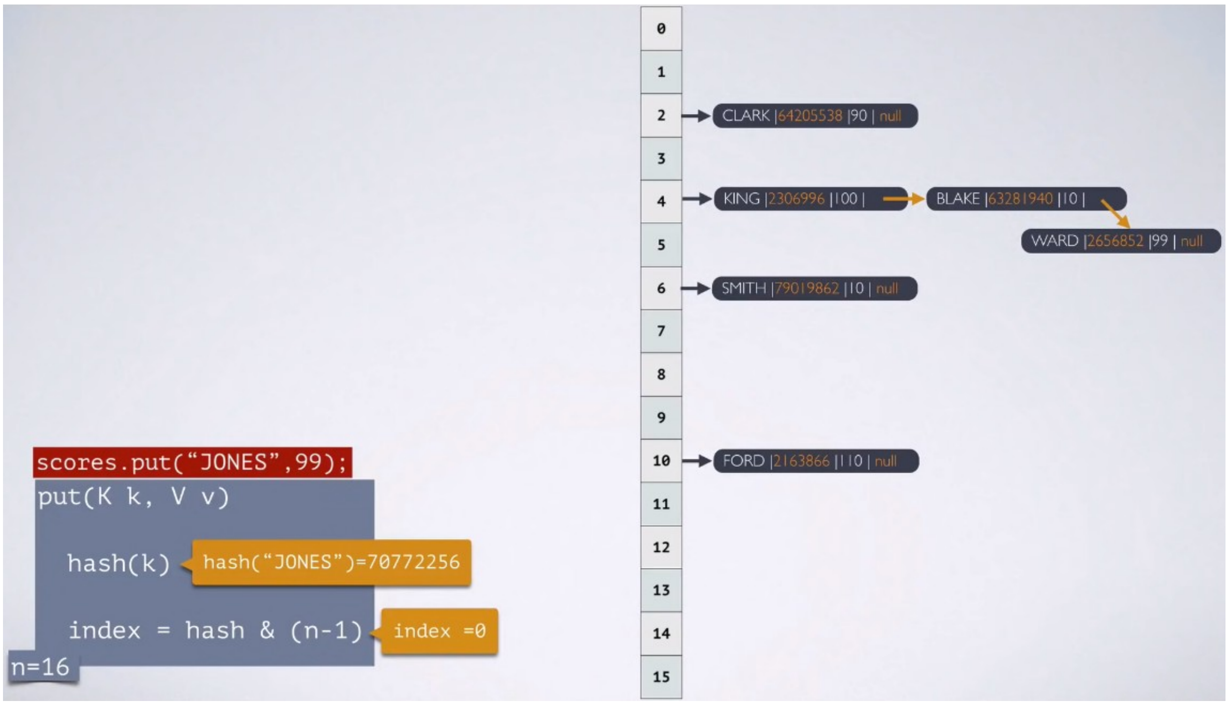
```
scores.put("JONES",99);
scores.put("WARD", 99);
put(K k, V v)
  hash(k)
  index = hash & (n-1)
n=16
```

hash("WARD")=2656852
index =4
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
CLARK |64205538 |90 | null
KING |2306996 |100 |
BLAKE |63281940 |10 | null
SMITH |79019862 |10 | null
FORD |2163866 |110 | null

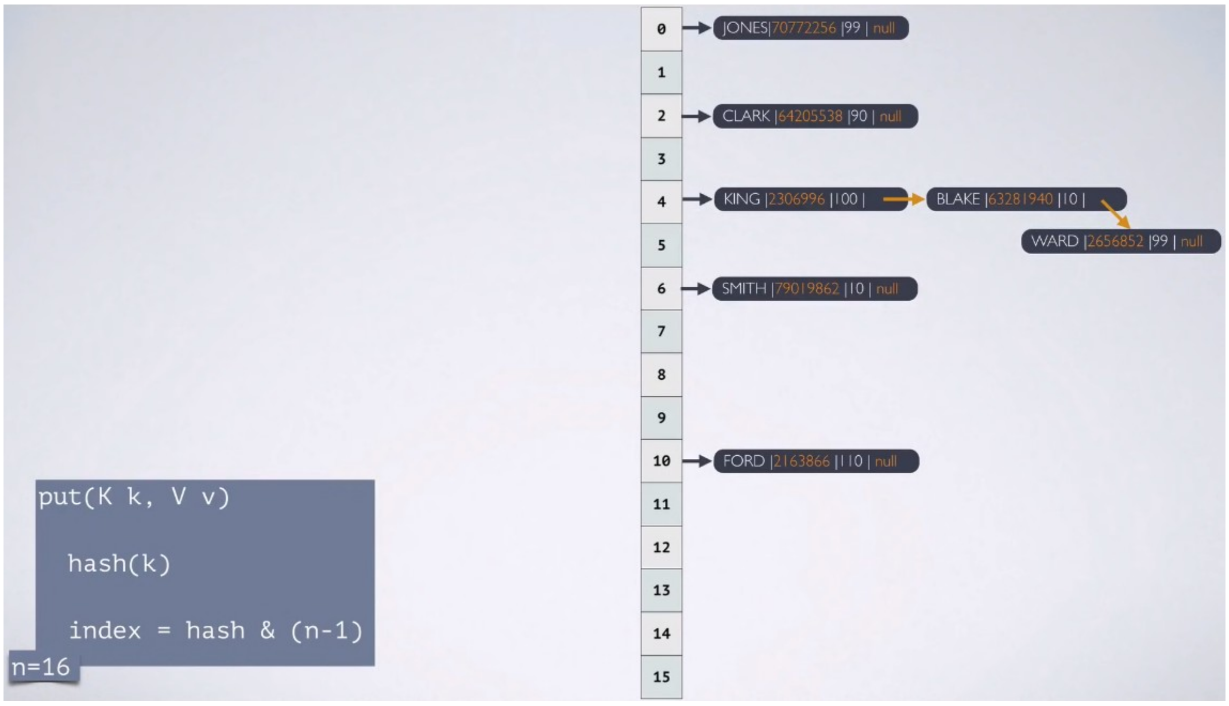


```
scores.put("JONES",99);
put(K k, V v)
  hash(k)
  index = hash & (n-1)
n=16
```

- For JONES:



```
scores.put("JONES",99);
put(K k, V v)

  hash(k)

  index = hash & (n-1)
n=16
```

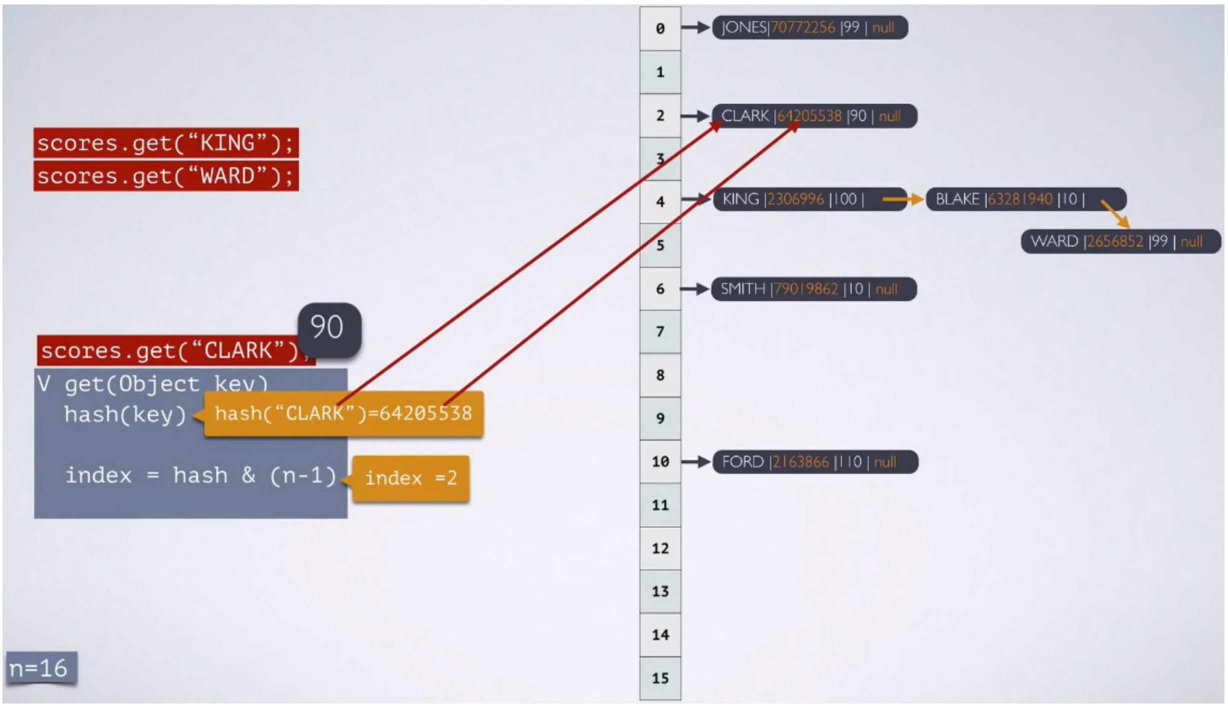


```
put(K k, V v)

  hash(k)

  index = hash & (n-1)
n=16
```

0
JONES|70772256 |99 | null
1
2
CLARK |64205538 |90 | null
3
4
KING |2306996 |100 |
BLAKE |63281940 |10 |
WARD |2656852 |99 | null
5
6
SMITH |79019862 |10 | null
7
8
9
10
FORD |2163866 |110 | null
11
12
13
14
15

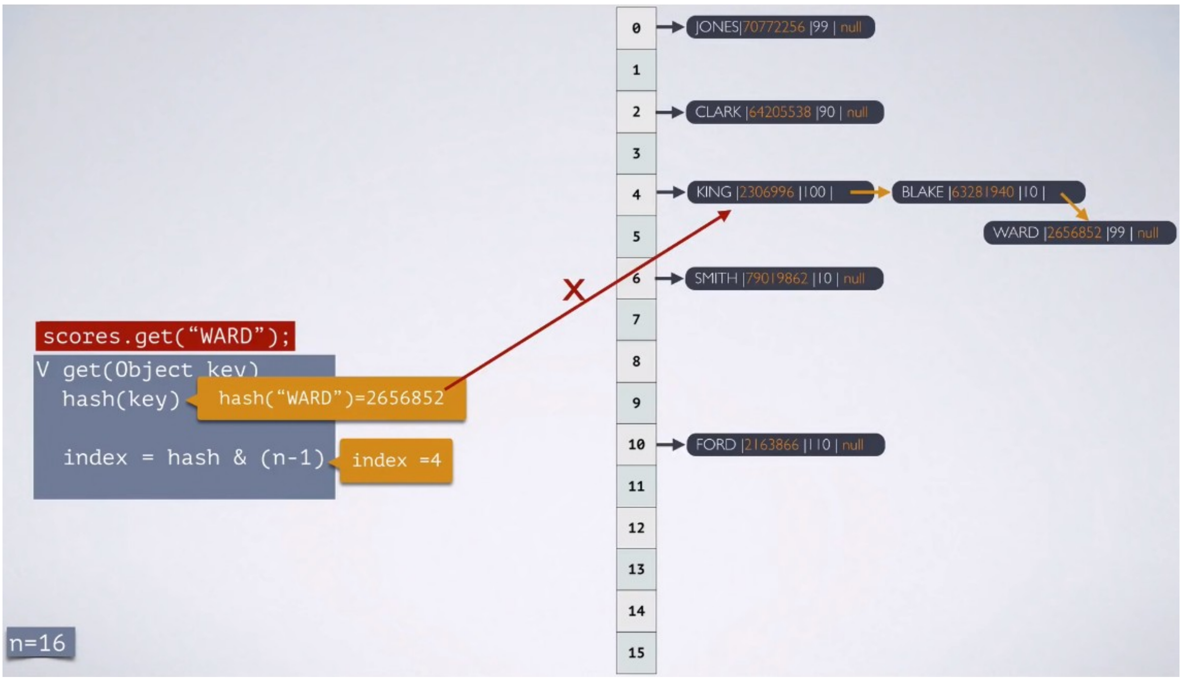
- How *get* operation works

- For the *get* operation, it also finds the hash of the key and then computes the index were the key could have fitted.
- Once the entry has been located, both hash keys are compared. If they match, both keys are then compared using the equals method. If match, the entry's value is returned to the caller, who in turn returns it.



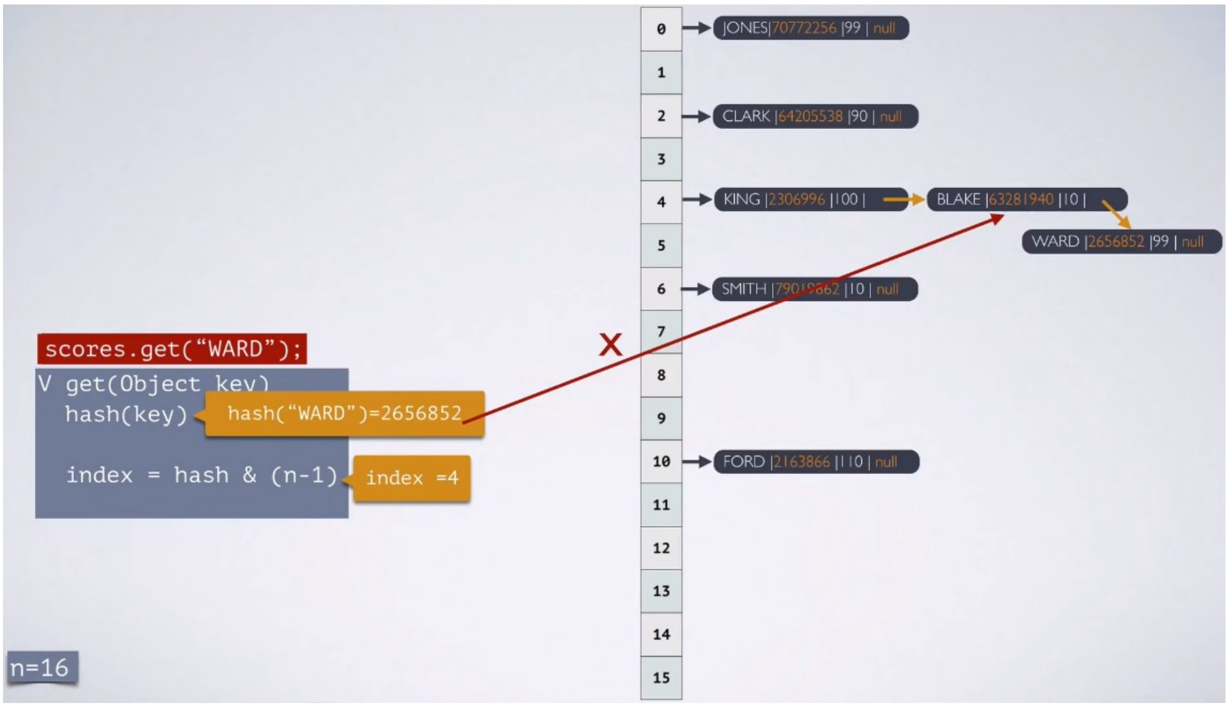
scores.get("KING");
scores.get("WARD");
90
scores.get("CLARK");
V get(Object key)
hash(key)
hash("CLARK")=64205538
index = hash & (n-1)
index =2
n=16
0 JONES|70772256 |99 | null
1
2 CLARK |64205538 |90 | null
3
4 KING |2306996 |100 | BLAKE |63281940 |10 | WARD |2656852 |99 | null
5
6 SMITH |79019862 |10 | null
7
8
9
10 FORD |2163866 |110 | null
11
12
13
14
15

- In case of WARD, once the hash code is computed, it computes the index.
- In this case we see that both hashes don't match, so the entry is skipped and it tries with the following.

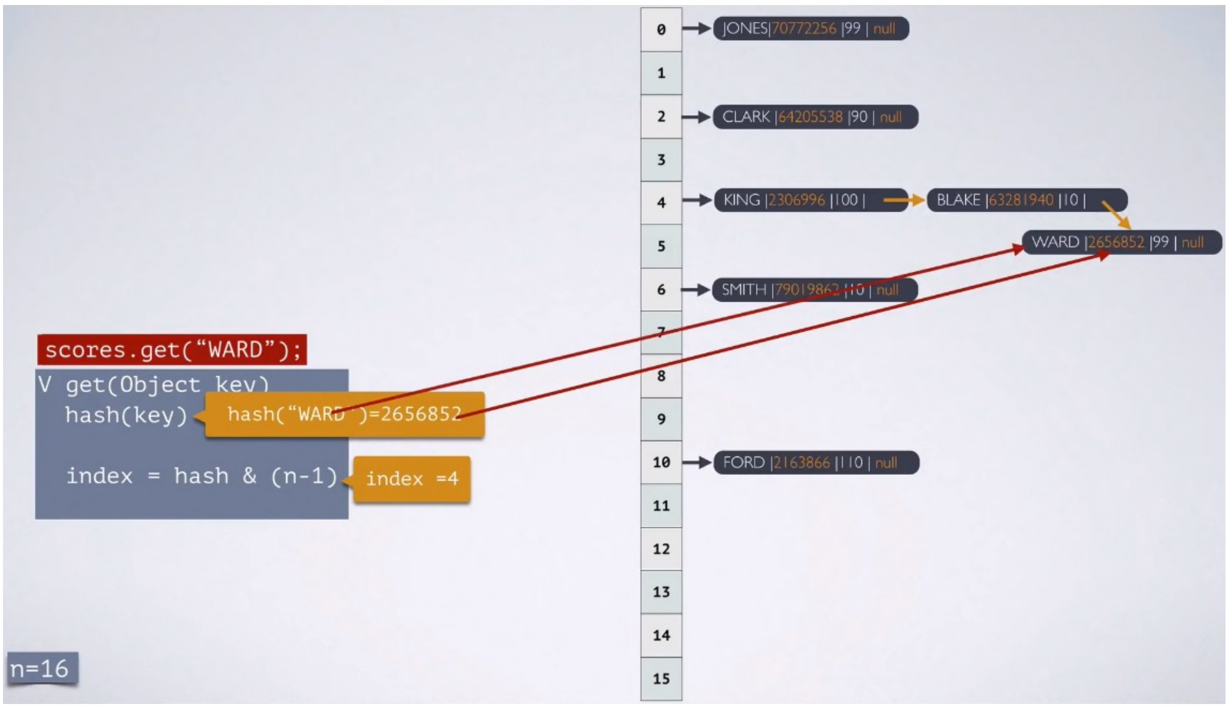


X
scores.get("WARD");
V get(Object key)
hash(key)
hash("WARD")=2656852
index = hash & (n-1)
index =4
n=16
0 JONES|70772256 |99 | null
1
2 CLARK |64205538 |90 | null
3
4 KING |2306996 |100 | BLAKE |63281940 |10 | WARD |2656852 |99 | null
5
6 SMITH |79019862 |10 | null
7
8
9
10 FORD |2163866 |110 | null
11
12
13
14
15

- The same happens with the next entry, not matching both hashes.



- But with the last entry there is a match between both keys, so the entry's value is returned to the caller, who in turn returns it.



- Change in Java 8

- In Java 8, when we have too many unequal keys which gives same hashcode (index), when the number of items in a shash bucket grows beyond certain threshhold (TREEIFY_THRESHOLD = 8), content of that bucket switches from using a linked list of entry objects to a balanced tree. This theoretically improves the worst-case performance from $O_{(n)}$ to $O_{(\text{long } n)}$.