Revised: 6/22/2024

Hash Tables

Michael C. Hackett

Assistant Professor, Computer Science

Lecture Topics

- Hash Table Basics
  - A Simple Hash Table
- Hash Functions
- Collision Resolution
  - Open Hashing (Separate Chaining)
  - Closed Hashing (Linear Probing)
- Resizing/Rehashing

Hash Tables

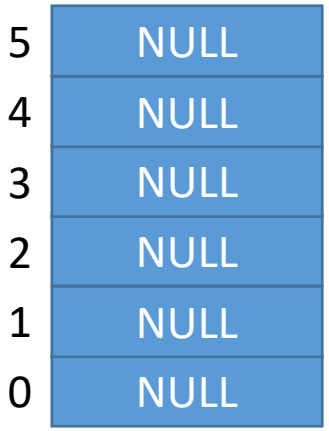
- A **hash table** (sometimes called a “dictionary”, “hash map”, or “map”) is a linear data structure consisting of **Key-Value Pairs** (KVPs).

- Keys and Values can be any data type.
  - Usually, all Keys are the same type and all Values are the same type.

- Implemented using an array.
  - This (in ideal circumstances) gives constant time for putting data into and getting data out of the hash table.

Hash Tables

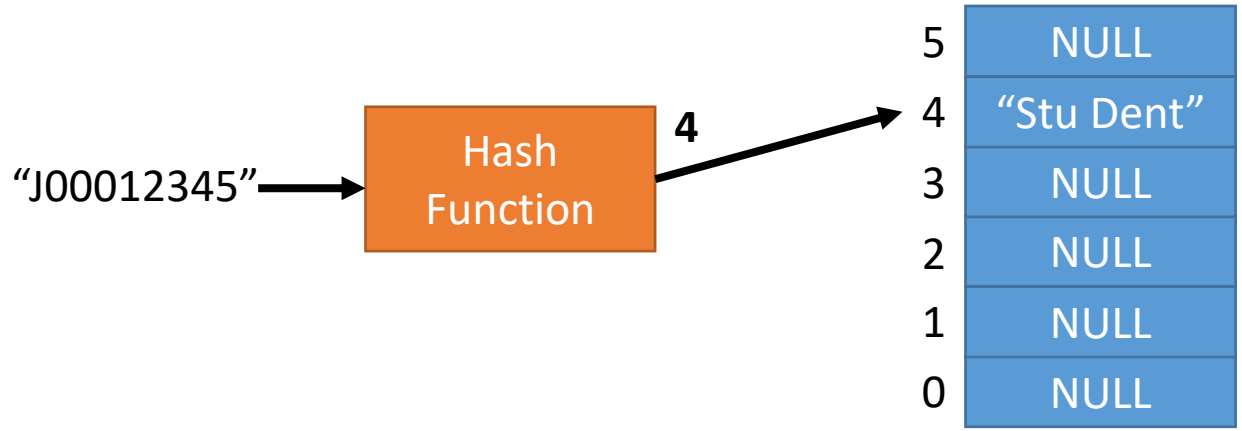
- First, an array is created.



Hash Tables

- Then, a **hash function** converts a KVP's key to an index in the array.

- KVP
  - Key = "J00012345"
  - Value = "Stu Dent"

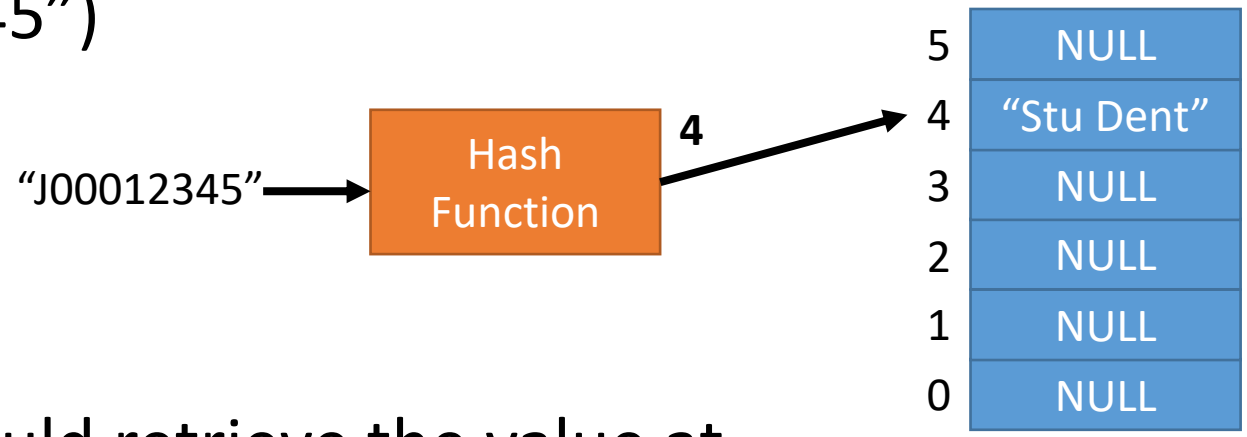


- In this example, the key was hashed to the value 4

Hash Tables

- The same hash function allows us to retrieve the value using the key.

- hashTable.get("J00012345")



- The statement above would retrieve the value at index 4

Hash Tables

- Quite a bit that needs considering.
  - How to convert the key to a valid index in the array?
    - **Hash Functions**
  - How do we handle what happens when two keys hash to the same index?
    - **Collision Resolution**
  - What happens when the hash table runs out of space?
    - **Resizing/Rehashing**

A Simple Hash Table

- We'll start with a very basic example to get the general idea of what is going on.

- Our simple hash table will use ints for keys and strings for values of each KVP.
  - And use a very simple hash function.

- No collision resolution.

A Simple Hash Table

- A KVP object.
  - This is what will be stored in the array.
- No setter for the key.
  - It should never change.
  - The value can be replaced/updated.

```
public class KVP {
    private int key;
    private String value;

    public KVP(int k, String v) {
        key = k;
        value = v;
    }

    public int getKey() {
        return key;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String v) {
        value = v;
    }
}
```

A Simple Hash Table

- map
  - The array of KVPs.
- Constructor:
  - Initializes the size and the array

```
public class HashTable {
    private KVP[] map;
    private final int SIZE;

    public HashTable(int sizeIn) {
        SIZE = sizeIn;
        map = new KVP[SIZE];
    }

}
```

A Simple Hash Table - Put

- Hash Function:
  - key % size.
  - If size is 10, it guarantees a remainder of 0 though 9
- If that index is null:
  - Safe to add the new KVP
- If its not null, but the keys are equal:
  - Update the value of the KVP
- Otherwise, a collision has occurred.

```
public void put(int key, String value) {
    int hashValue = key % SIZE;
    if(map[hashValue] == null) {
        KVP temp = new KVP(key, value);
        map[hashValue] = temp;
    }
    else if(map[hashValue].getKey() == key) {
        map[hashValue].setValue(value);
    }
    else {
        throw new IllegalArgumentException("Hash Collision");
    }
}
```

A Simple Hash Table - Get

- Hash Function:
  - key % size.
  - Same thing we did in “put”.
- If that index is not null:
  - If that KVP has the right key:
    - Return the value
- Otherwise, there is no KVP to return.
  - No KVP at the hashed index
  - The supplied key doesn’t match the KVP’s key

```
public String get(int key) {
    int hashValue = key % SIZE;
    if(map[hashValue] != null) {
        if(map[hashValue].getKey() == key) {
            return map[hashValue]->getValue();
        }
    }
    throw new IllegalArgumentException("Hash Collision");
}
```

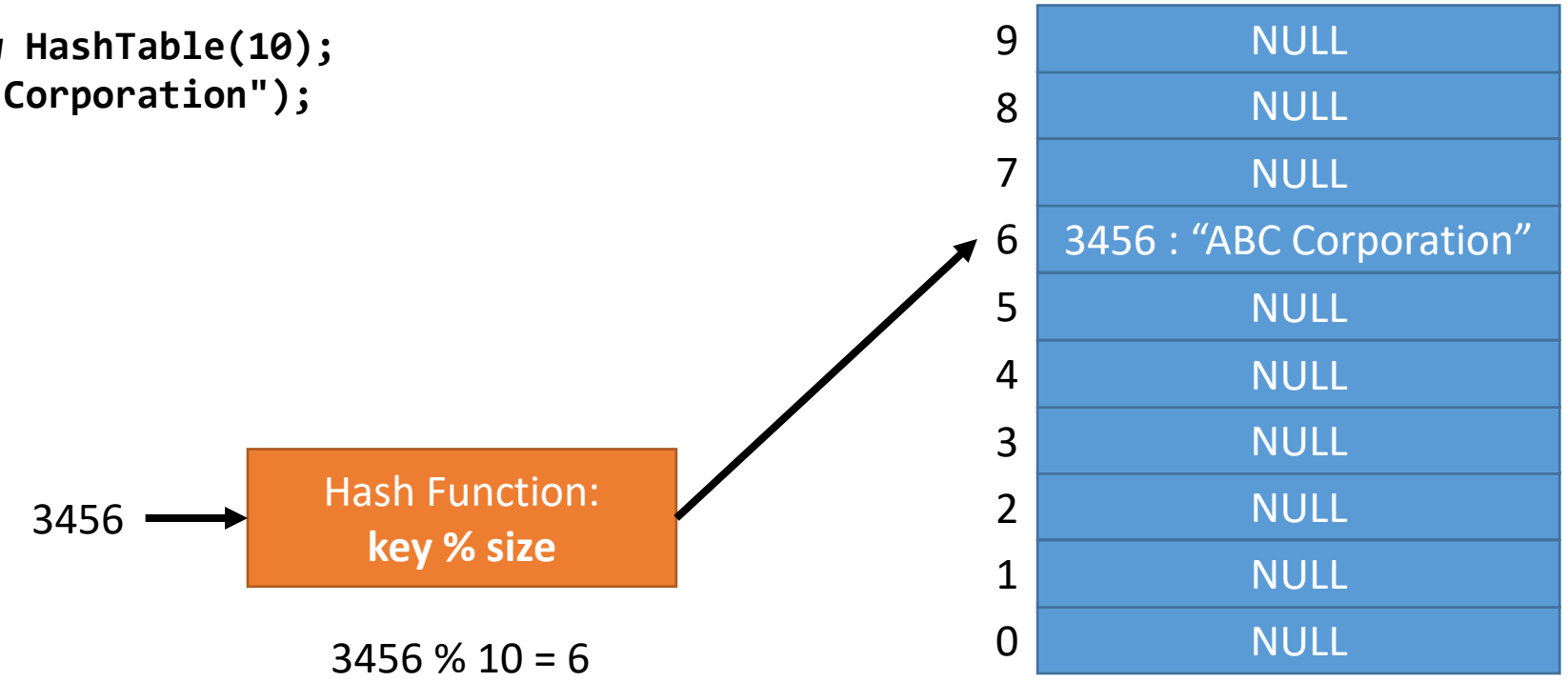
A Simple Hash Table - Remove

- Hash Function:
  - key % size.
  - Same thing we did in “put”.
- If that index is not null:
  - If that KVP has the right key:
    - Set the index to null
- Otherwise, there is no KVP to remove.
  - No KVP at the hashed index
  - The supplied key doesn’t match the KVP’s key

```
public boolean remove(int key) {
    int hashValue = key % SIZE;
    if(map[hashValue] != null) {
        if(map[hashValue].getKey() == key) {
            map[hashValue] = null;
            return true;
        }
    }
    return false;
}
```

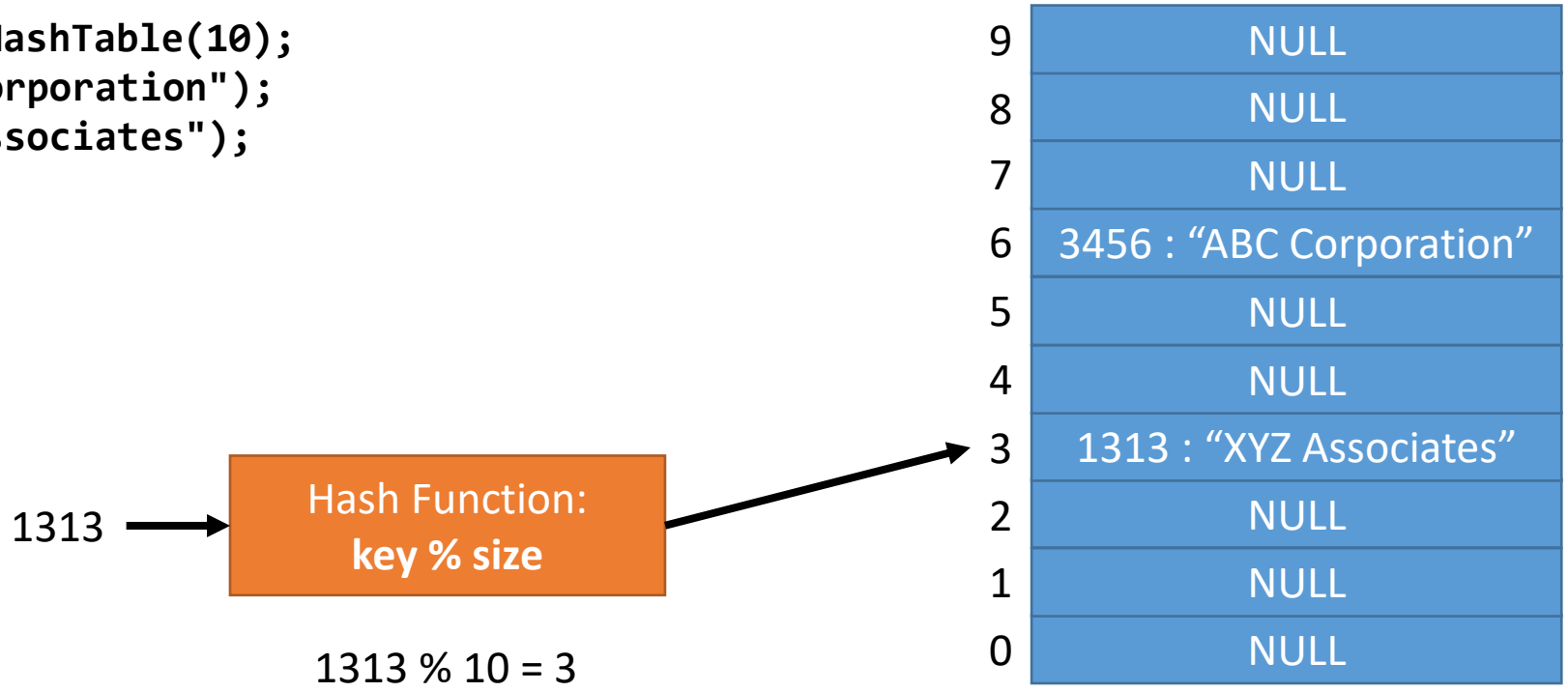
A Simple Hash Table

```
HashTable ht = new HashTable(10);
ht.put(3456, "ABC Corporation");
```



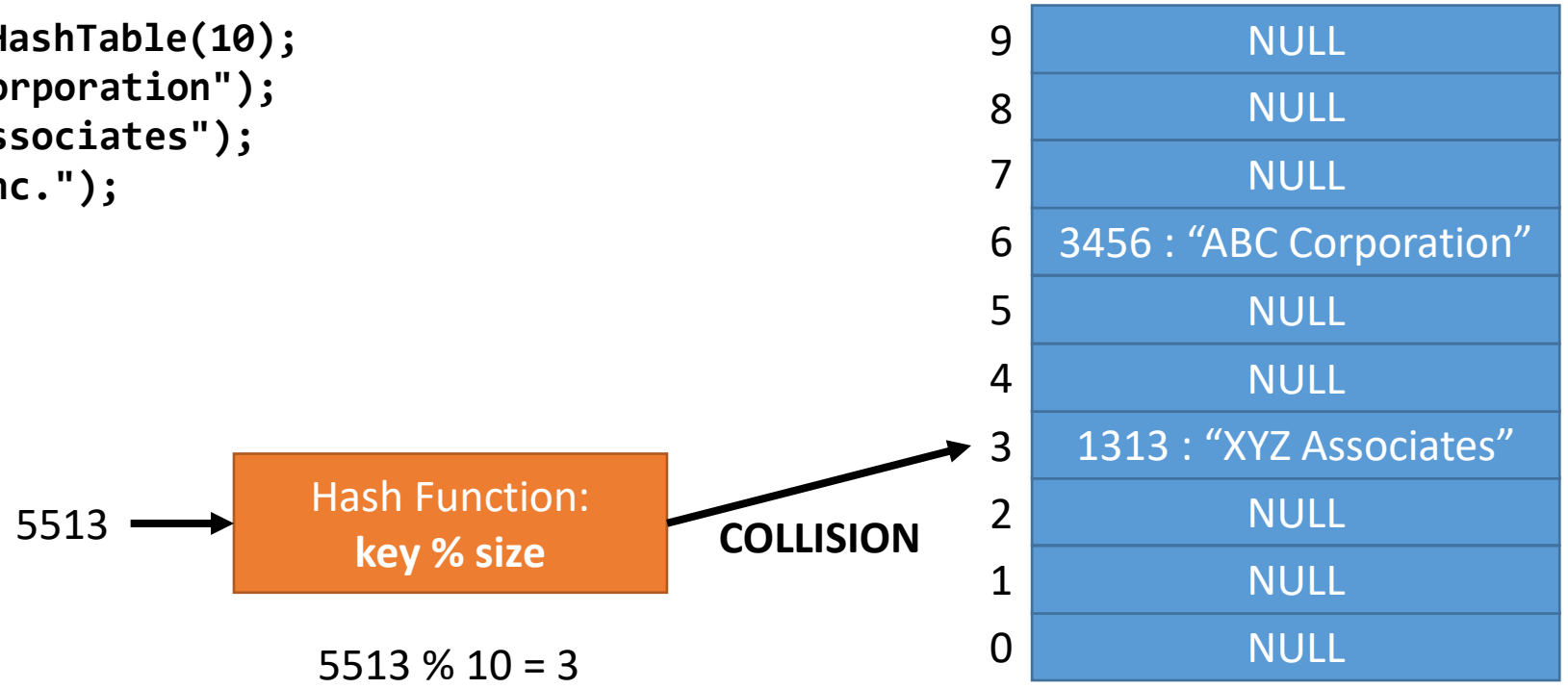
A Simple Hash Table

```
HashTable ht = new HashTable(10);
ht.put(3456, "ABC Corporation");
ht.put(1313, "XYZ Associates");
```



A Simple Hash Table

```
HashTable ht = new HashTable(10);
ht.put(3456, "ABC Corporation");
ht.put(1313, "XYZ Associates");
ht.put(5513, "FGH Inc.");
```



Hash Functions

- There is no single perfect hash function.

- The goals of the hash function are:
  - Distribute keys to indexes the best it can.
  - Minimize collisions.

Hash Functions

- Let’s look again at the hash function shown previously:
  - key % size
  - The size is 10, so key % 10

- The last digit of the key decides the index.
  - 345**6** % 10 = **6**
  - 131**3** % 10 = **3**
  - 551**3** % 10 = **3**

Hash Functions

- If keys are sufficiently different, the performance won't be too bad.
  - Might have a collision here and there that could be resolved without wasting too much time.
  - 345**6** % 10 = **6**
  - 131**3** % 10 = **3**
  - 482**2** % 10 = **2**
  - 9999**9** % 10 = **9**

- If every key will end with a 3….
  - Always a collision.
  - 345**3** % 10 = **3**
  - 131**3** % 10 = **3**
  - 482**3** % 10 = **3**
  - 9999**3** % 10 = **3**

Hash Functions

- One trick is to use array sizes that are prime:
  - key % size
  - If the size is 31, then its key % 31

- It will reduce the number of common factors between the key and the size.
  - 3456 % 31 = 15
  - 1313 % 31 = 11
  - 5513 % 31 = 26

Hash Functions

- Different keys:
  - 3456 % 31 = 15
  - 1313 % 31 = 11
  - 4822 % 31 = 17
  - 99999 % 31 = 24

- Every key ends with a 3….
  - 3453 % 31 = 12
  - 1313 % 31 = 11
  - 4823 % 31 = 18
  - 99993 % 31 = 18
  - 99983 % 31 = 8

- Won’t entirely eliminate collisions.
- Distributes indexes better, leading to fewer collisions.

Hash Functions

- Hashing a string is a little different.
- We wouldn’t want to use the string’s length, because if every key is the same number of characters, we’d always hash to the same index.
- A good way to hash strings is to use each character’s decimal value in the hash function.

Hash Functions

- Add up the decimal value of each character in the key.
- Return:
  - The sum % the table size

```
public int hashFunction(String key) {
    int hash = 0;
    for(int i = 0; i < key.length(); i++) {
        hash = hash + key.charAt(i);
    }
    return hash % SIZE;
}
```

Hash Functions

- This will work well enough for strings with varying characters.
- The characters of some strings may add up to the same total.
  - DDD1, DCE1, DEC1 all add up to the same total.
- Multiplying the hash by a prime number can help reduce the number of collisions.

Hash Functions

- Won't eliminate every collision, but it will distribute indexes a little better in situations where the character decimal values add up to the same sum.

```
public int hashFunction(String key) {
    int hash = 0;
    for(int i = 0; i < key.length(); i++) {
        hash = (31 * hash) + key.charAt(i);
    }
    return hash % SIZE;
}
```

Collision Resolution

- As we've seen, collisions are bound to happen.

- We'll see two techniques to resolve collisions:
  - Open Hashing (using Separate Chaining)
  - Closed Hashing (using Linear Probing)

Open Hashing

- With open hashing, a key's hash value always corresponds to its index in the array.

- Stored at each index is a linked list, where each node is (or contains) a KVP.
  - The list can shrink/grow in size dynamically.
  - Sometimes called a "bucket" in this context.

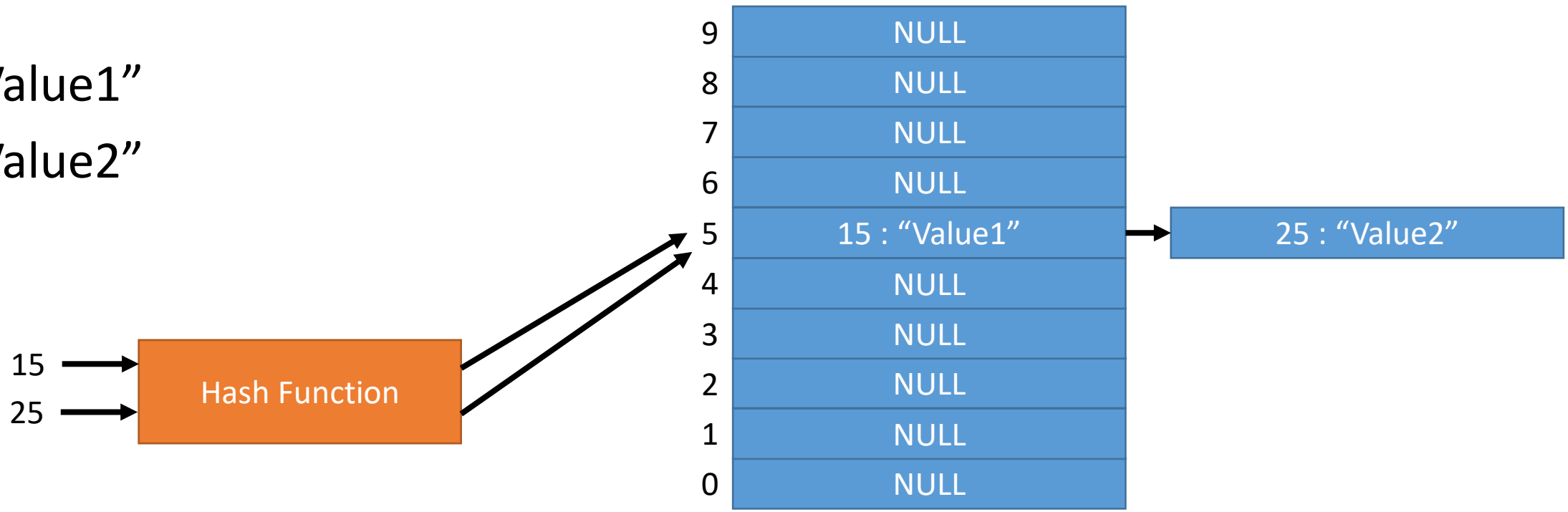
- Basically, we're allowing more than one KVP to be stored at one index.

Open Hashing

KVPs

15:“Value1”

25:“Value2”



Open Hashing - Put

```
public void put(int key, String value) {
    int hashValue = hashFunction(key);
    KVP temp = new KVP(key, value);
    buckets[hashValue].push(temp);
}
```

Open Hashing – Adding to Bucket

```
public void push(KVP newKVP) {
    Node temp = head;
    while(temp != tail && (temp.data.getKey() != newKVP.getKey())) {
        temp = temp.next
    }
    if(temp == tail) {
        Mode newNode = new Node();
        tail.next = newNode;
        tail.data = newKVP;
        tail = tail.next;
    }
        else {
            temp.data = newKVP;
        }
    }
    length++;
}
```

Open Hashing - Get

```
public String get(int key) {
    int hashValue = hashFunction(key);
    buckets[hashValue].remove(key);
}
```

Open Hashing – Getting from Bucket

```
public String get(int key) {
    current = head.next;
    while(current != tail && current.data.getKey() != key) {
        current = current.next;
    }
    if(current == tail) {
        throw new NoSuchElementException("No KVP for the key: " + key);
    }
    return current.data.getValue();
}
```

Closed Hashing

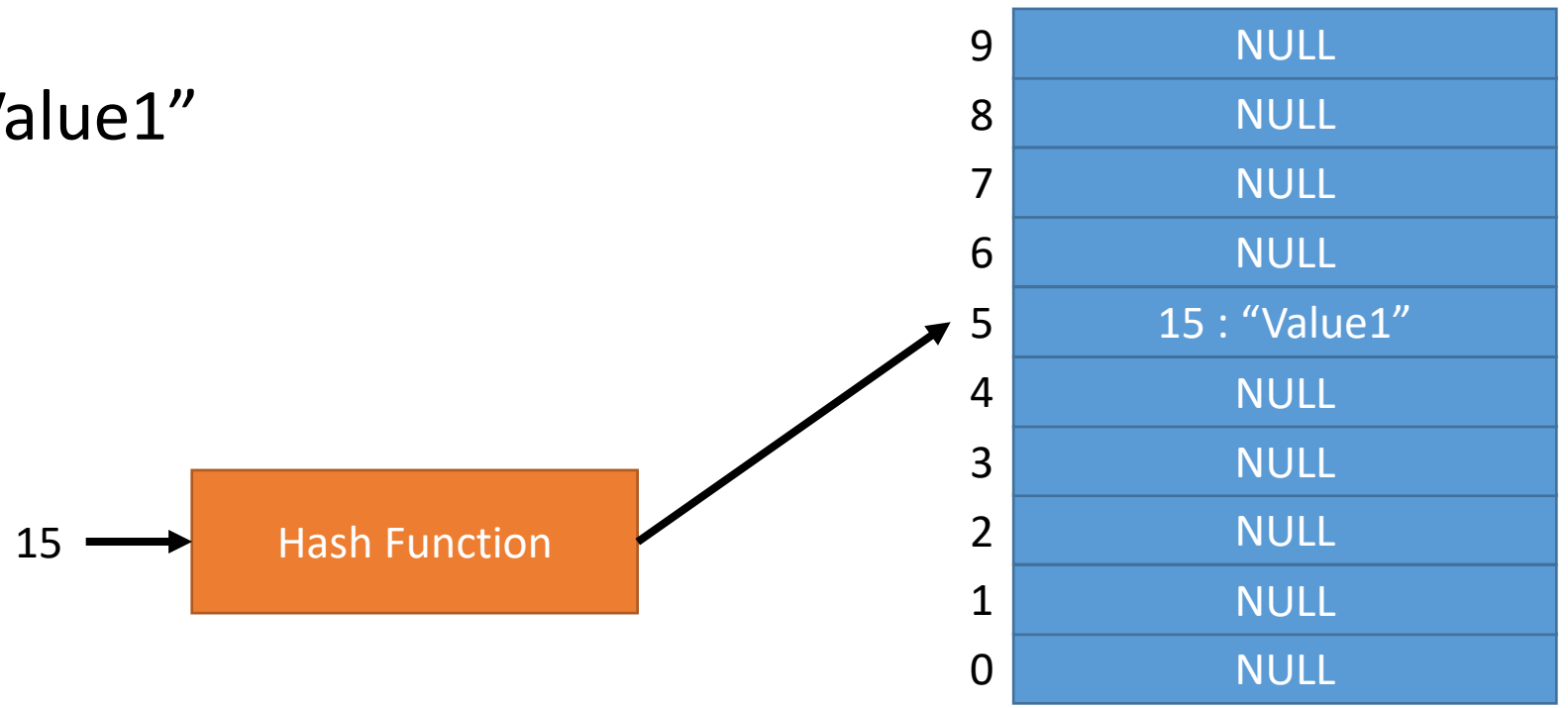
- With closed hashing, a key's hash value *may not* directly correspond to an index in the array.

- If that index is already in use, it checks the next index to see if it is empty, then checks the next index, and so on.
  - **Linear Probing**

Closed Hashing

KVP

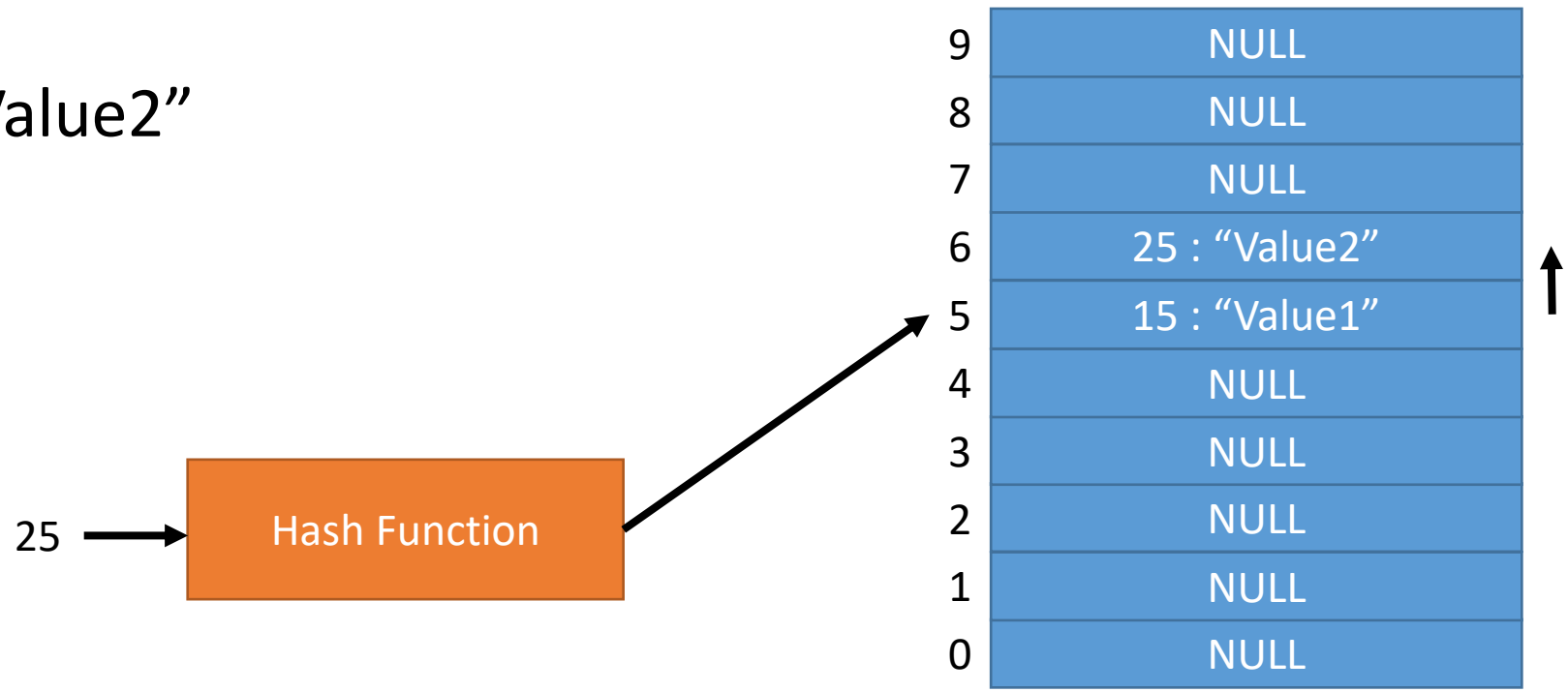
15:“Value1”



Closed Hashing

KVP

25:“Value2”



(Since index 5 was in use, it tries index 6)

Closed Hashing - Put

```
public void put(String key, String value) {
    int hashValue = hashFunction(key);
    int start = hashValue;
    while(map[hashValue] != null && !map[hashValue].getKey().equals(key)) {
        hashValue = (hashValue + 1) % SIZE;
        if(start == hashValue) {
            throw new RuntimeException("Table is full.");
        }
    }
    map[hashValue] = new KVP(key, value);
}
```

Closed Hashing - Get

```
public String get(String key) {
    int hashValue = hashFunction(key);
    int start = hashValue;
    while(map[hashValue] != null && !map[hashValue].getKey().equals(key)) {
        hashValue = (hashValue + 1) % SIZE;
        if(start == hashValue) {
            throw new RuntimeException("Key not found.");
        }
    }
    if(map[hashValue] == null) {
        throw new RuntimeException("Key not found.");
    }
    return map[hashValue].getValue();
}
```

Resizing

- The greater the **load** (utilization) of the hash table, the greater the chance for a collision.

- Making an oversized hash table would waste space, but perhaps reduce the number of collisions.

- Resizing a hash table would be more efficient.
  - Make the table larger when free space starts running low.
  - Make the table smaller when the load has sufficiently decreased.

Resizing

- Create a (temporary) reference to the current map

```
KVP[] temp = map;
```

Resizing

- Decide if shrinking or growing the table

```
if(shrink) {
    size -= (int)(size * .25); //Decrease 25%
    if(size < MINSIZE) {
        size = MINSIZE;
    }
}
else {
    size = size * 2;           //Double the size
}
```

Resizing

- Create a new map and set all values to null
  - Reset the total count of KVPs

```
map = new KVP[size];

count = 0;
```

Resizing

- Rehash each KVP from the old map into the new map.

```
for(int i=0; i < oldSize; i++) {
    if(temp[i] != null) {
        put(temp[i].getKey(), temp[i].getValue());
    }
}
```