Revised: 10/22/2019

Hash Tables

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Hash Table Basics
  - A Simple Hash Table
- Hash Functions
- Collision Resolution
  - Closed Addressing (Separate Chaining)
  - Open Addressing (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

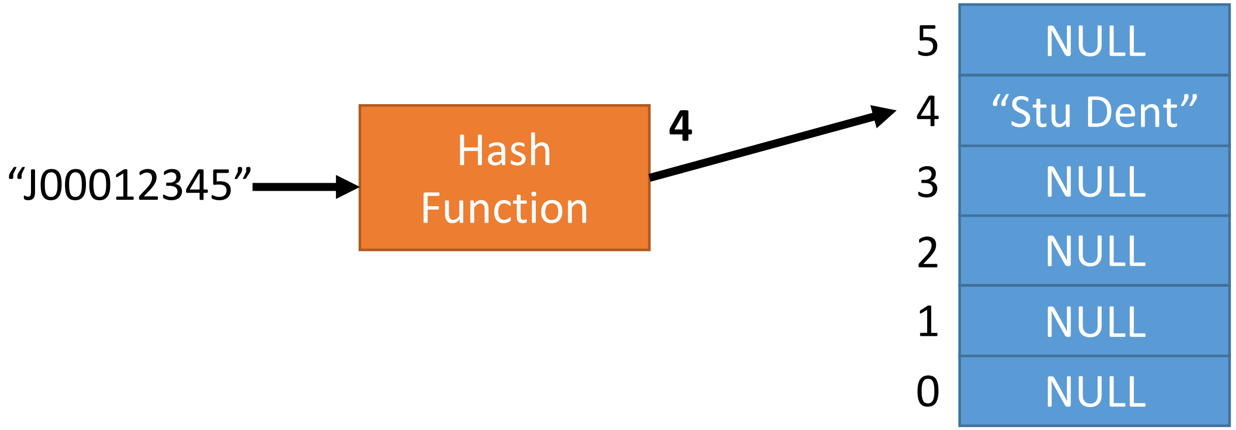
- First, an array is created.

5	NULL
4	NULL
3	NULL
2	NULL
1	NULL
0	NULL

Hash Tables

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

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

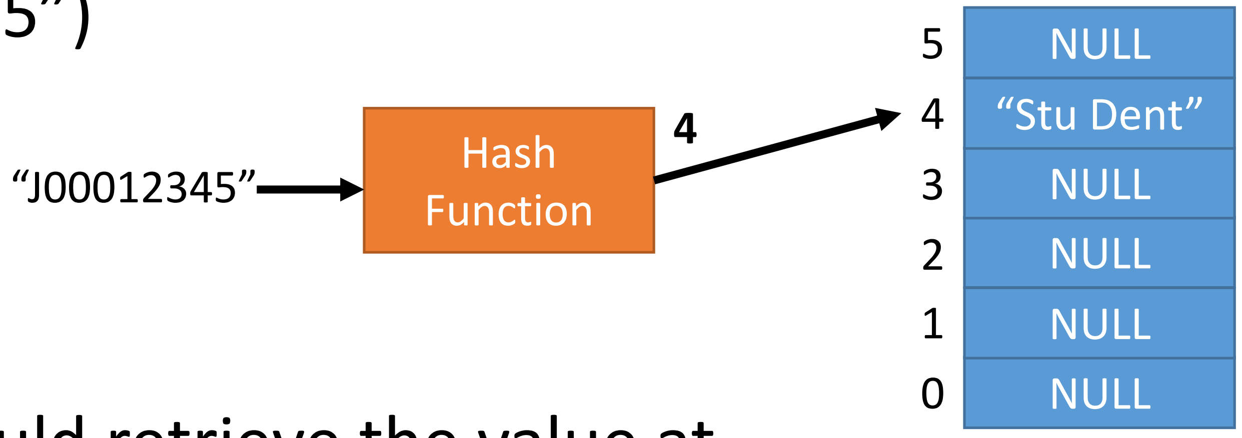


- In this example, the key was hashed to the value 4

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.

```
class KVP {
    private:
        int key;
        string value;
    public:
        KVP(int k, string v) {
            key = k;
            value = v;
        }

        int getKey() {
            return key;
        }

        string getValue() {
            return value;
        }

        void setValue(string v) {
            value = v;
        }
};
```

A Simple Hash Table

- KVP **map
  - A pointer (array) of KVP pointers.
- Constructor:
  - Sets every index to NULL
- Destructor:
  - Deletes each KVP in the array.
  - Deletes the array.

```
class HashTable {
    private:
        KVP **map;
        const int SIZE = 10;
    public:
        HashTable() {
            map = new KVP*[SIZE];
            for(int i=0; i<SIZE; i++) {
                map[i] = NULL;
            }
        }

        ~HashTable() {
            for(int i = 0; i < SIZE; i++) {
                delete map[i];
            }
            delete[] map;
        }
};
```

delete vs free()

- The delete operator releases the object’s memory AND calls the object’s destructor.
  - **delete myObject;**

- The free function releases the object’s memory but does NOT call the object’s destructor.
  - **free(myObject);**

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.

```
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_invalid_argument("Hash Collision");
    }
}
```

A Simple Hash Table - Get

- Hash Function:
  - key % size.
  - Same thing we did in “put”.
- If that index is not null:
  - Return the value
- Otherwise (it was null) there is no value to return.

```
string get(int key) {
    int hashValue = key % SIZE;
    if(map[hashValue] != NULL) {
        return map[hashValue]->getValue();
    }
    else {
        __throw_invalid_argument("KVP not found");
    }
}
```

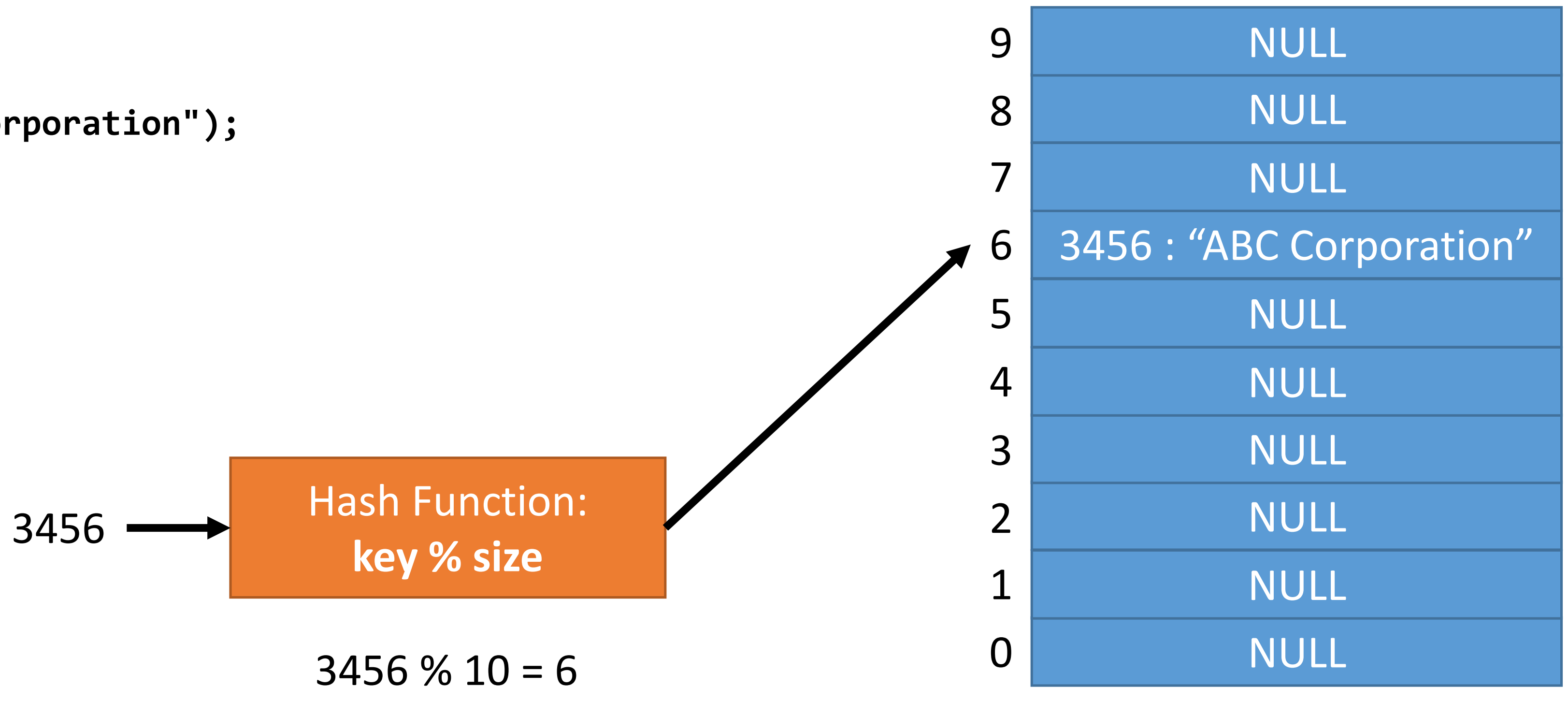
A Simple Hash Table - Remove

- Hash Function:
  - key % size.
  - Same thing we did in “put”.
- If that index is not null:
  - Delete the KVP
  - Set that index to null

```
bool remove(int key) {
    int hashValue = key % SIZE;
    if(map[hashValue] != NULL) {
        delete map[hashValue];
        map[hashValue] = NULL;
        return true;
    }
    else {
        return false;
    }
}
```

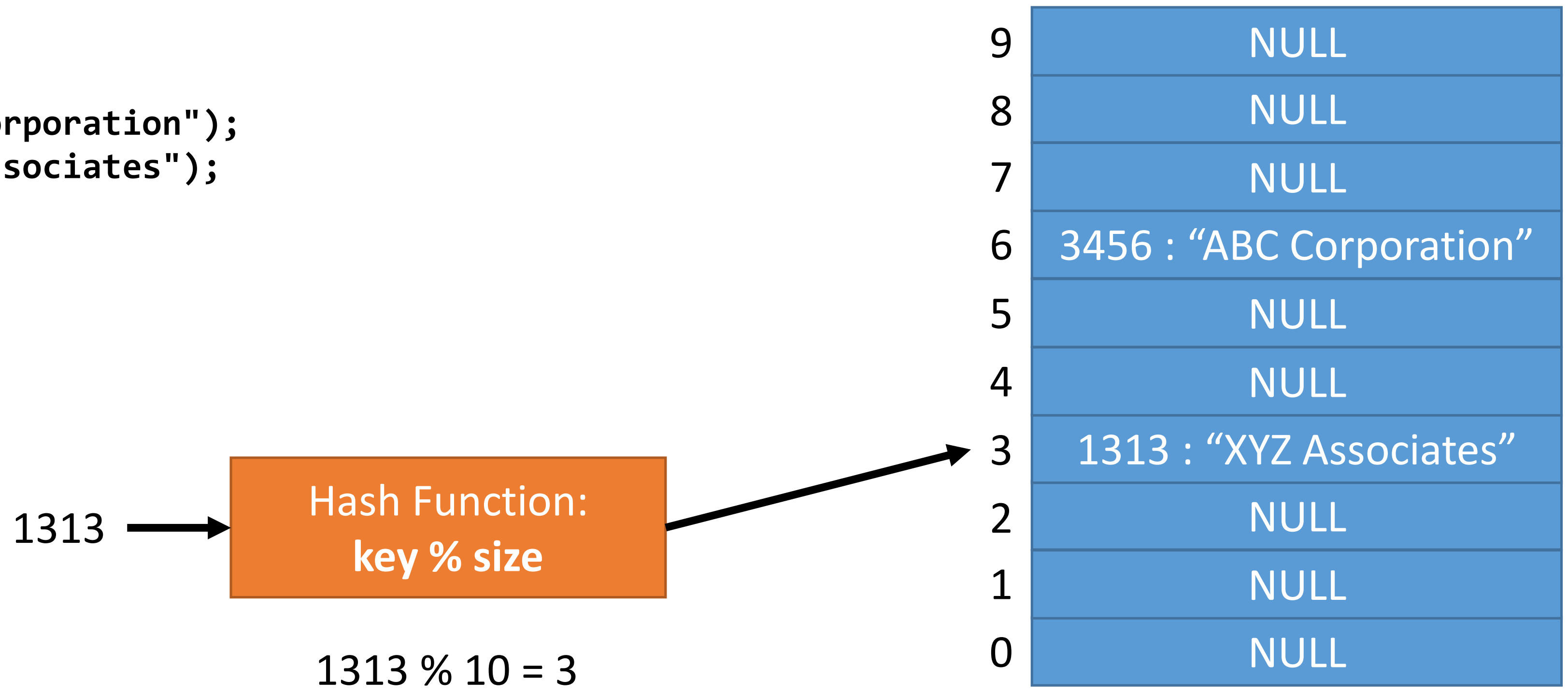
A Simple Hash Table

```
int main() {
    HashTable ht;
    ht.put(3456, "ABC Corporation");
}
```



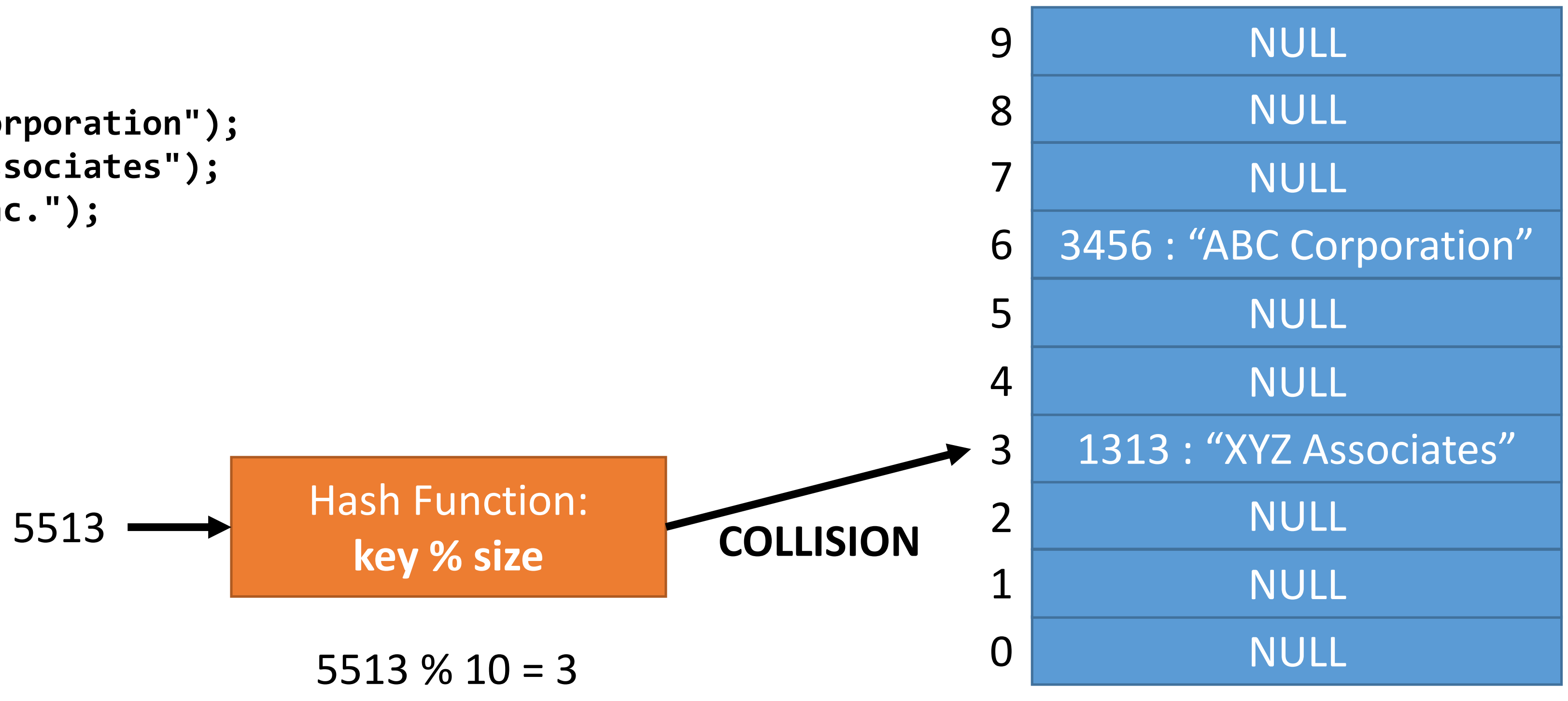
A Simple Hash Table

```
int main() {
    HashTable ht;
    ht.put(3456, "ABC Corporation");
    ht.put(1313, "XYZ Associates");
}
```



A Simple Hash Table

```
int main() {
    HashTable ht;
    ht.put(3456, "ABC Corporation");
    ht.put(1313, "XYZ Associates");
    ht.put(5513, "FGH Inc.");
}
```



Hash Functions

- There is no 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 <u>reduce the number of common factors</u> 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

```
int hashFunction(string key, int size) {
    int hash = 0;
    for(int i = 0; i < key.length(); i++) {
        hash = hash + key[i];
    }
    return hash % size;
}
```

Hash Functions

- This will work well enough for strings with varying characters.

- But, 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.

```
int hashFunction(string key, int size) {
    int hash = 0;
    for(int i = 0; i < key.length(); i++) {
        hash = (19 * hash) + key[i];
    }
    return hash % size;
}
```

Collision Resolution

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

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

Closed Addressing

- Closed Addressing means that the key’s hash value always corresponds to its index in the array.

- Stored at each index is a linked list, where each node is a KVP.
  - The list can shrink/grow in size dynamically.
  - Sometimes called a “bucket” in this context.

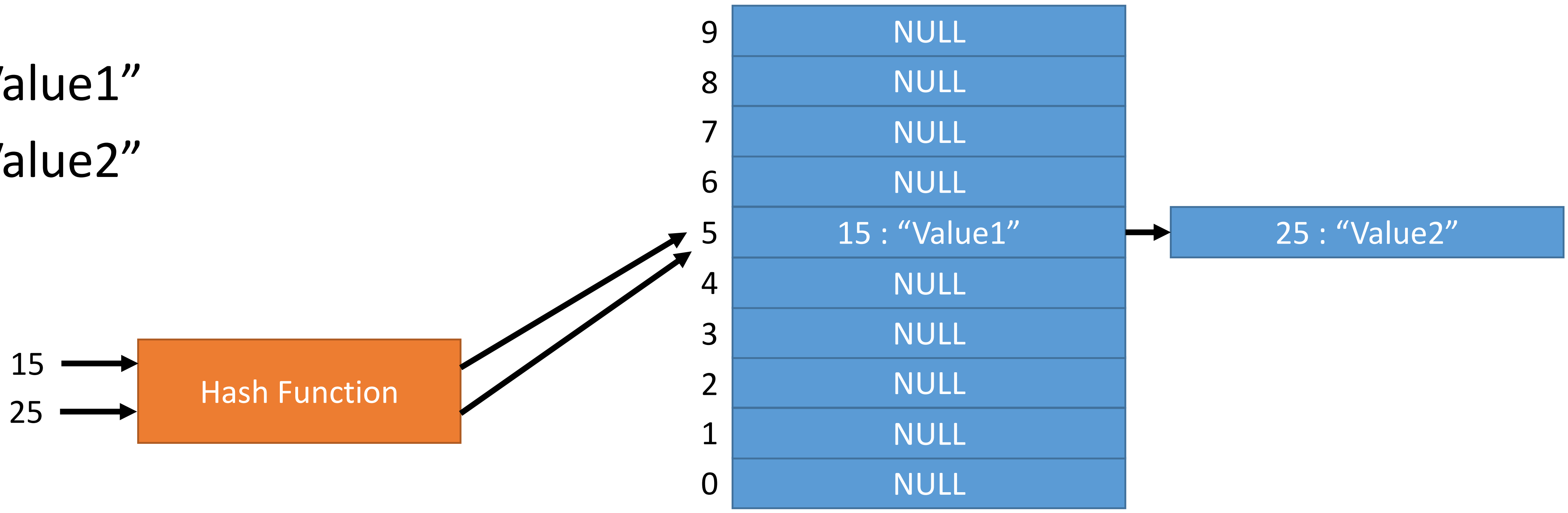
- Basically, we’re allowing more than one KVP to be stored at one index.

Closed Addressing

KVPs

15:“Value1”

25:“Value2”



Closed Addressing - Put

```
void put(int key, string value) {
    int hashValue = key % SIZE;
    KVP *newEntry = new KVP(key, value);
    if(map[hashValue] == NULL) {
        Bucket *newList = new Bucket;
        map[hashValue] = newList;
    }
    map[hashValue]->add(newEntry);
}
```

Closed Addressing – Adding to Bucket

```
void add(KVP* newKVP) {
    if(head == NULL) {
        head = newKVP;
        tail = newKVP;
    }
    else {
        KVP *temp = head;
        while(temp != NULL && (temp->getKey() != newKVP->getKey())) {
            temp = temp->getNext();
        }
        if(temp == NULL) {
            tail->setNext(newKVP);
            tail = tail->getNext();
        }
        else {
            temp->setValue(newKVP->getValue());
        }
    }
}
```

Closed Addressing - Get

```
string get(int key) {
    int hashValue = key % SIZE;
    if(map[hashValue] == NULL) {
        __throw_invalid_argument("Key not found");
    }
    KVP *e = map[hashValue]->get(key);
    if(e == NULL) {
        __throw_invalid_argument("Key not found");
    }
    else {
        return e->getValue();
    }
}
```

Closed Addressing – Getting from Bucket

```
KVP* get(int key) {
    KVP *current = head;
    while(current != NULL && current->getKey() != key) {
        current = current->getNext();
    }
    return current;
}
```

Open Addressing

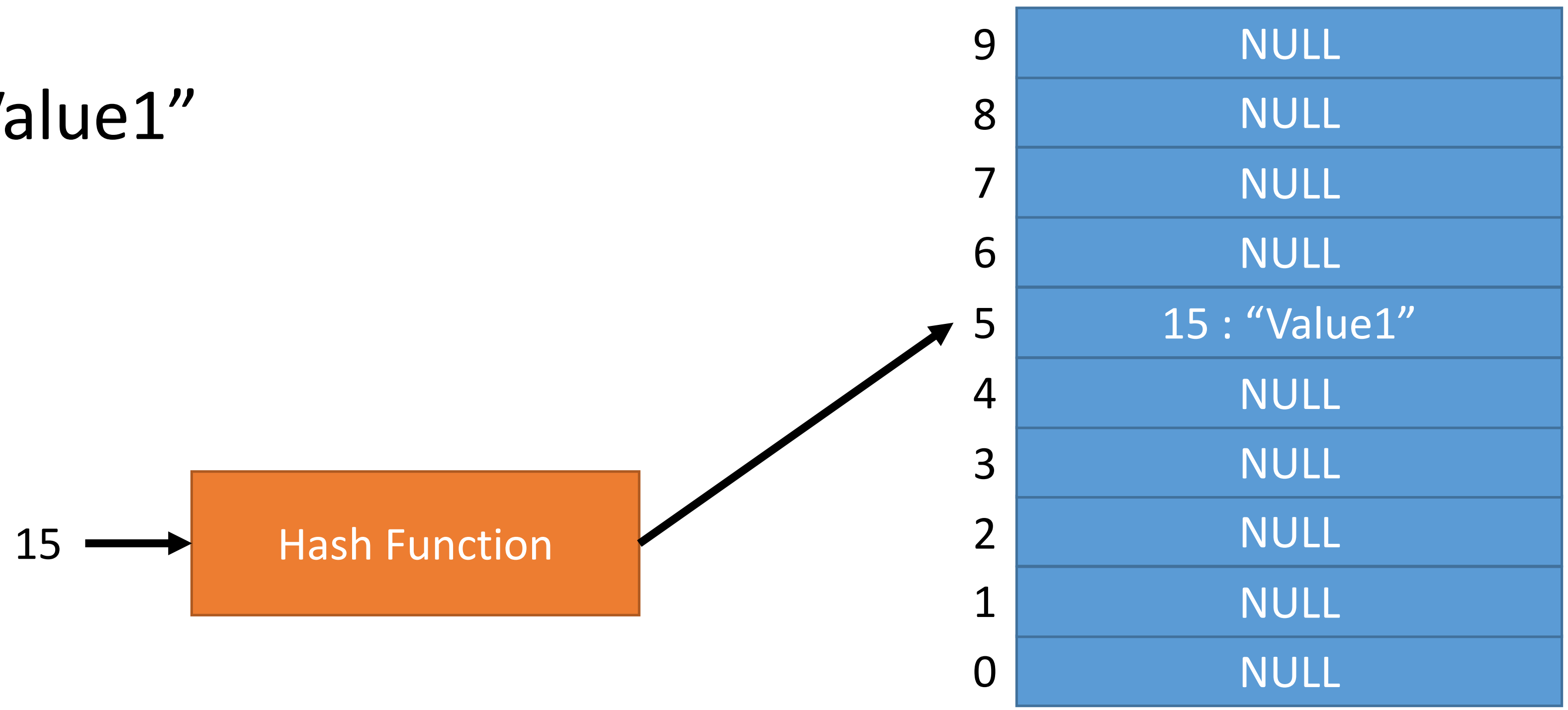
- Open Addressing means that the key's hash value *may not* directly correspond to an index in the array.

- 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**

Open Addressing

<u>KVP</u>

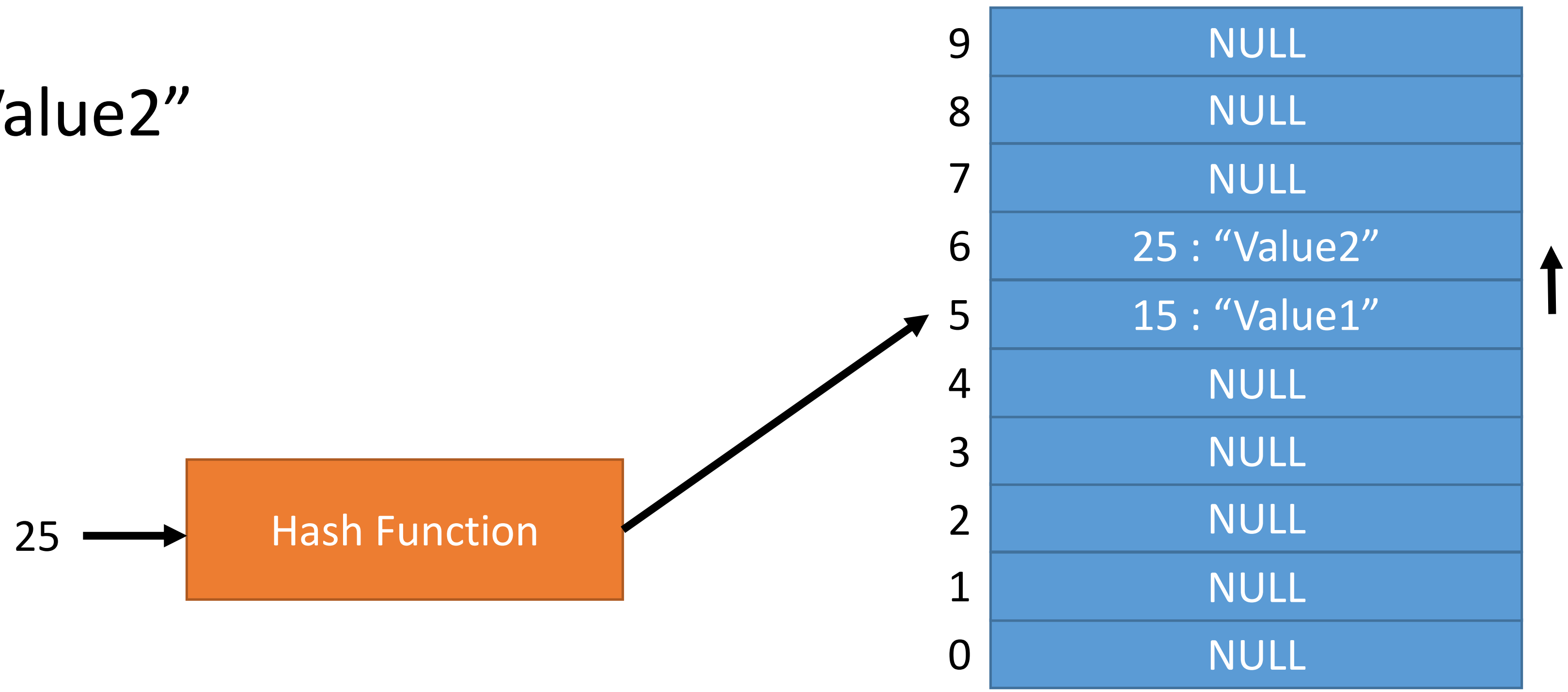
15:“Value1”



Open Addressing

<u>KVP</u>

25:“Value2”



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

Open Addressing - Put

```
void put(string key, string value) {
    int hashValue = hashFunction(key);
    int start = hashValue;
    while(map[hashValue] != NULL && map[hashValue]->getKey().compare(key) != 0) {
        hashValue = (hashValue + 1) % SIZE;
        if(start == hashValue) {
            __throw_overflow_error("Table is full.");
        }
    }
    if(map[hashValue] != NULL) {
        delete map[hashValue];
    }
    map[hashValue] = new KVP(key, value);
}
```

Open Addressing - Get

```
string get(string key) {
    int hashValue = hashFunction(key);
    int start = hashValue;
    while(map[hashValue] != NULL && map[hashValue]->getKey().compare(key) != 0) {
        hashValue = (hashValue + 1) % SIZE;
        if(start == hashValue) {
            __throw_overflow_error("Key not found.");
        }
    }
    if(map[hashValue] == NULL) {
        __throw_invalid_argument("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) pointer 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];

for(int i=0; i<SIZE; i++) {
    map[i] = NULL;
}

count = 0;
```

Resizing

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

```
for(int i=0; i < oldSize; i++) {
    if(temp[i] != NULL) {
        put(temp[i]->getKey(), temp[i]->getValue());
        delete temp[i];
    }
}
delete[] temp;
```

Resizing

```
void rehash(bool shrink) {
    int oldSize = SIZE;
    KVP **temp = map;
    if(shrink) {
        SIZE -= (int)(SIZE * .25); //Decrease 25%
        if(SIZE < MINSIZE) {
            SIZE = MINSIZE;
        }
    }
    else {
        SIZE = SIZE * 2; //Double the size
    }

    map = new KVP*[SIZE];
    for(int i=0; i<SIZE; i++) {
        map[i] = NULL;
    }

    count = 0;

    for(int i=0; i < oldSize; i++) {
        if(temp[i] != NULL) {
            put(temp[i]->getKey(), temp[i]->getValue());
            delete temp[i];
        }
    }
    delete[] temp;
}
```