COSC2436: Hash Tables

Direct Hashing

- Overwrites data when a collision is found
- Execution time is very fast since it doesn’t involve a collision resolution technique
- Data is lost when overwritten
- Should not be used when we want to reserve all data

Direct Hashing

Insert 4
index = 4 % 10 = 4

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	-1	-1	-1	-1	-1

Direct Hashing

Insert 24
index = 24 % 10 = 4

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	24	-1	-1	-1	-1	-1

Direct Hashing

Insert 134
index = 134 % 10 = 4

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	134	-1	-1	-1	-1	-1

Direct Hashing

Insert 56
index = 56 % 10 = 6

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	134	-1	56	-1	-1	-1

Direct Hashing

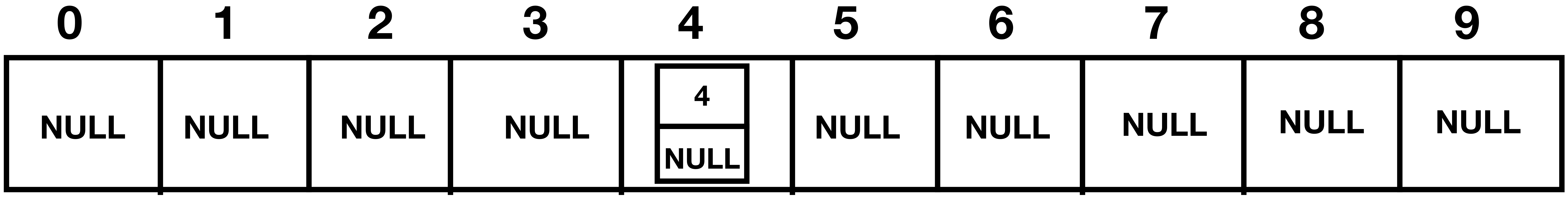
```
void directHashing(int table[], int x, int tableSize){
    int index = x % tableSize;
    table[index] = x;
}
```

Separate Chaining

- Each cell of the hash table points to a linked list of records that have the same hash function value
- Hash table never fills up because more elements can always be added to the “chain”
- If a certain hash value keeps happening it can cause the search time to become **O(n)**

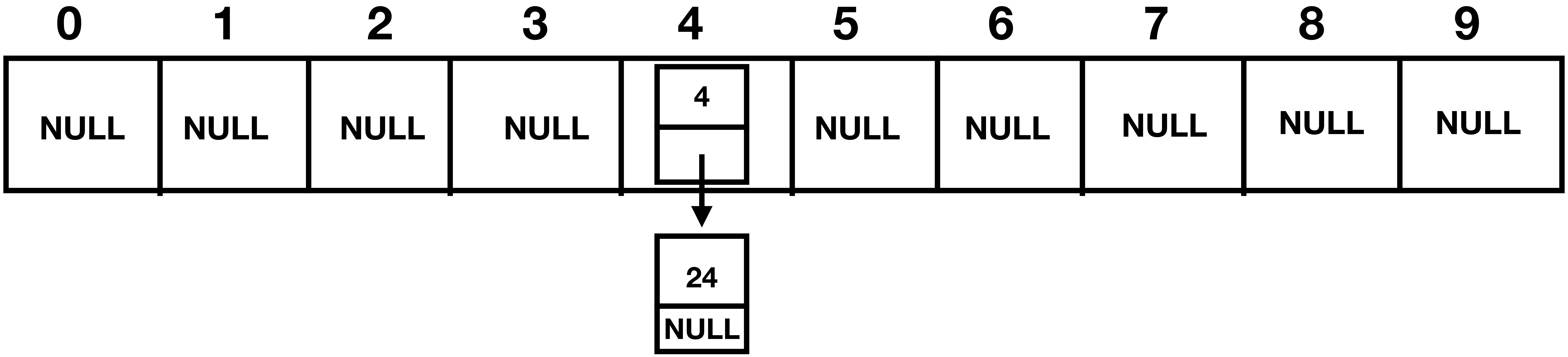
Separate Chaining

Insert 4
index = 4 % 10 = 4
index 4 is NULL so we simply add at the index 4



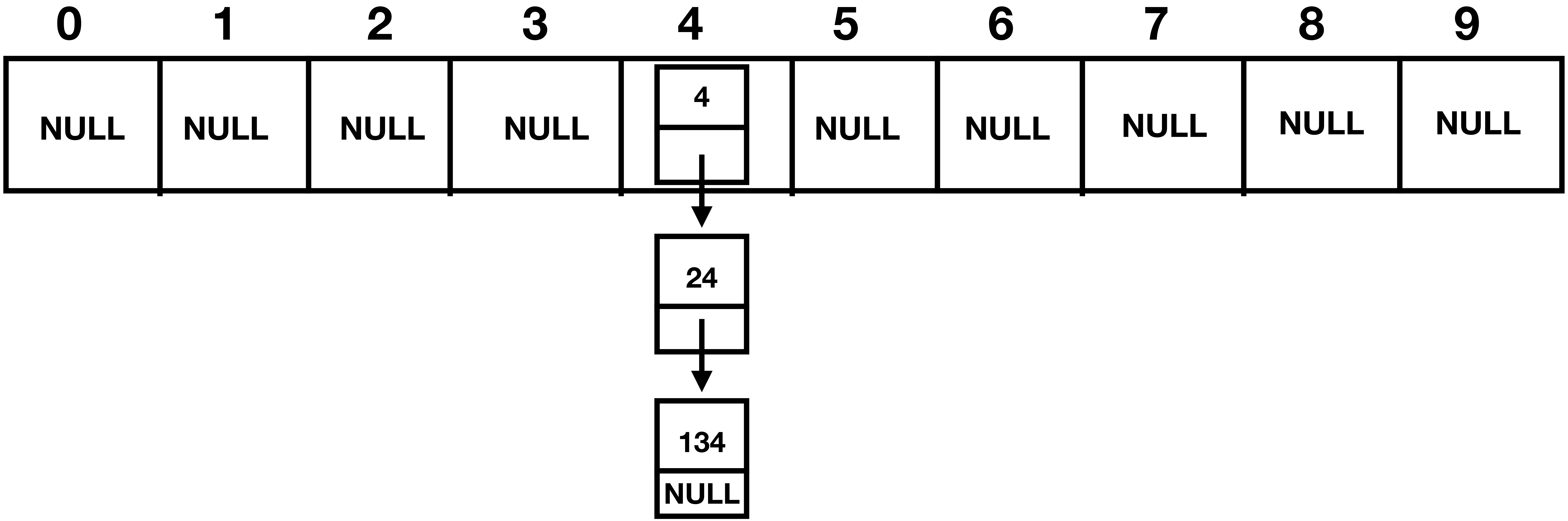
Separate Chaining

Insert 24
index = 24 % 10 = 4
index 4 is not NULL so we go to last element of the linked list



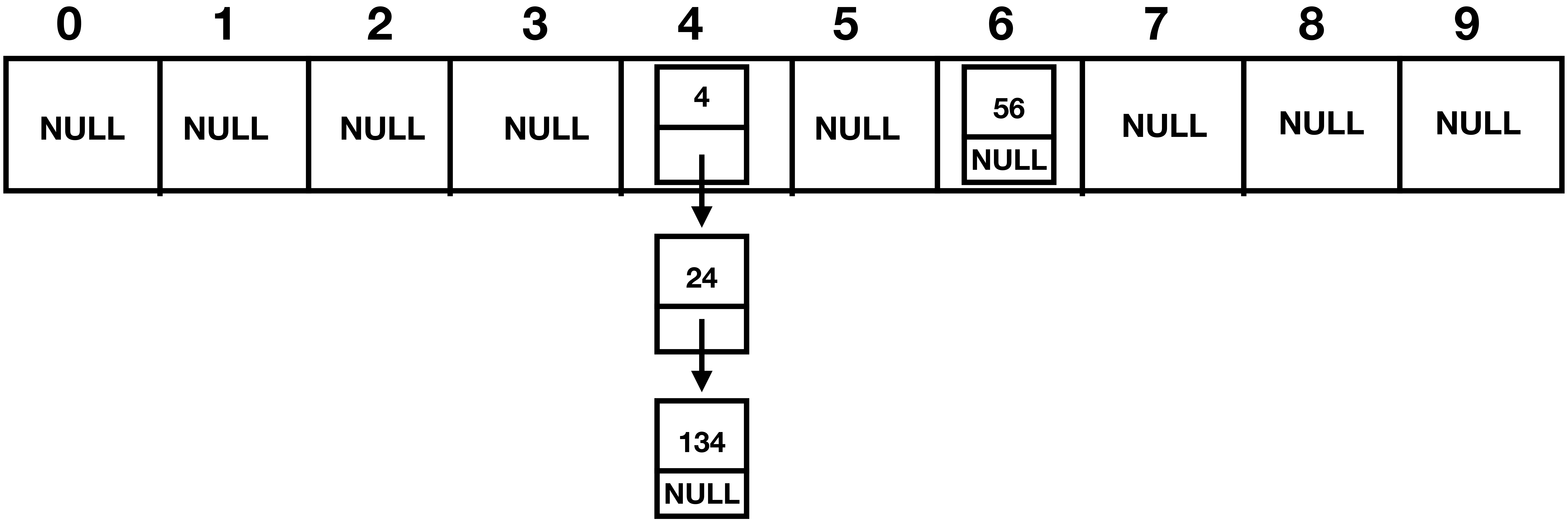
Separate Chaining

Insert 134
index = 134 % 10 = 4
index 4 is not NULL so we go to last element of the linked list



Separate Chaining

Insert 56
index = 56 % 10 = 6
index 6 is NULL so we simply add at index 6



Separate Chaining

```
struct node{
    int value;
    node *next;
    node (int _value){
        value = _value;
        next = nullptr;
    }
};
```

```
void serparateChaining(node *table[], int _value){
    int index = _value % 10;
    node *temp = new node(_value);
    if(table[index] == nullptr){
        table[index] = temp;
    }
    else{
        node *cu = table[index];
        while(cu->next != nullptr){
            cu = cu->next;
        }
        cu->next = temp;
    }
}
```

Linear Probing

- Linearly probe for the next available index
- Formula: **(hash(x) + i) % tableSize** (where **hash()** is **x % tableSize** and **i** is incremented by 1 until a free space is found)
- The problem with linear probing is “clustering.” This happens when many consecutive elements form groups and it can cause the time it takes to find a free slot to increase

Linear Probing

Insert: 4

index = (4 % 10 + 0) % 10 = 4

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	-1	-1	-1	-1	-1

Linear Probing

Insert: 24
index = (24 % 10 + 0) % 10 = 4
index 4 is taken so we increment i
index = (24 % 10 + 1) % 10 = 5

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	24	-1	-1	-1	-1

Linear Probing

Insert: 134

index = (134 % 10 + 0) % 10 = 4

index 4 is taken so we increment i

index = (134 % 10 + 1) % 10 = 5

index 5 is taken so we increment i

index = (134 % 10 + 2) % 10 = 6

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	24	134	-1	-1	-1

Linear Probing

Insert: 56
index = (56 % 10 + 0) % 10 = 6
index 6 is taken so we increment i
index = (56 % 10 + 1) % 10 = 7

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	24	134	56	-1	-1

Linear Probing

```
void linearProbing(int table[], int x, int tableSize){
  int index = 0; //intialize index
  for(int i = 0; i < tableSize; i++){
    index = ((x % tableSize) + i) % tableSize;
    if(table[index] == -1){ //Check to see if table[index] is empty
      table[index] = x;
      break; //Make sure to break so x is only added once
    }
  }
}
```

Quadratic Probing

- When a collision happens, we iterate with **i^2** to look for the next available slot in the table
- Formula: **(hash(x) + i^2) % tableSize** (where **hash()** is **x % tableSize** and **i** is incremented by 1 until a free space is found)
- Since the probe is **i^2**, there will be less clustering in the hash table
- Quadratic probing is faster than linear probing in terms of searching and inserting

Quadratic Probing

Insert: 4

index = ((4 % 10) + 0^2) % 10 = 4

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	-1	-1	-1	-1	-1

Quadratic Probing

Insert: 24

index = ((24 % 10) + 0^2) % 10 = 4

index 4 is taken so we increment i

index = ((24 % 10) + 1^2) % 10 = 5

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	24	-1	-1	-1	-1

Quadratic Probing

Insert: 134

index = ((134 % 10) + 0^2) % 10 = 4

index 4 is taken so we increment i

index = ((134 % 10) + 1^2) % 10 = 5

index 5 is taken so we increment i

index = ((134 % 10) + 2^2) % 10 = 8

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	24	-1	-1	134	-1

Quadratic Probing

Insert: 56
index = ((56 % 10) + 0^2) % 10 = 6

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	24	56	-1	134	-1

Quadratic Probing

```
void quadraticProbing(int table[], int x, int tableSize){
    int index = 0; //initialize index
    for(int i = 0; i < tableSize; i++){
        index = ((x%tableSize) + (i*i)) % tableSize;
        if(table[index] == -1){ //Check to see if table[index] is empty
            table[index] = x;
            break; //Make sure to break so x is only added once
        }
    }
}
```

Double Hashing

- When a collision happens, we use another hash function (hash2(x)) to look for an empty index
- Formula: **(hash1(x) + (i * hash2(x)) % tableSize** (where **i** is incremented by 1)
- Less clustering and faster than linear probing
- **hash1(x) = x % tableSize**
- **hash2(x) = prime - (x % prime)** (where **prime** is a prime number smaller than tableSize)

Double Hashing

Insert: 4

index = ((4 % 10) + (0 * (7 - (4 % 7)))) % 10 = 4

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	-1	-1	-1	-1	-1

Double Hashing

Insert: 24

index = ((4 % 10) + (0 * (7 - (4 % 7)))) % 10 = 4

Index 4 is taken so we increment i

index = ((24 % 10) + (1 * (7 - (24 % 7)))) % 10 = 8

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	4	-1	-1	-1	24	-1

Double Hashing

Insert: 134

index = ((134 % 10) + (0 * (7 - (134 % 7)))) % 10 = 4

index 4 is taken so we increment i

index = ((134 % 10) + (1 * (7 - (134 % 7)))) % 10 = 0

0	1	2	3	4	5	6	7	8	9
134	-1	-1	-1	4	-1	-1	-1	24	-1

Double Hashing

Insert: 56

index = ((56 % 10) + (0 * (7 - (56 % 7)))) % 10 = 6

0	1	2	3	4	5	6	7	8	9
134	-1	-1	-1	4	-1	56	-1	24	-1

Double Hashing

```
50 int hash1(int x, int tableSize){
51     return x % tableSize;
52 }
53 int hash2(int x, int prime){
54     return prime - (x % prime);
55 }
56
57 void doubleHashing(int table[], int x, int tableSize){
58     int index = 0; //initialize index
59     for(int i = 0; i < tableSize; i++){
60         index = (hash1(x, tableSize) + (i * hash2(x, 7))) % tableSize;
61         if(table[index] == -1){ //Check if table[index] is empty
62             table[index] = x;
63             break; //Make sure to break so x is only added once
64         }
65     }
66 }
```

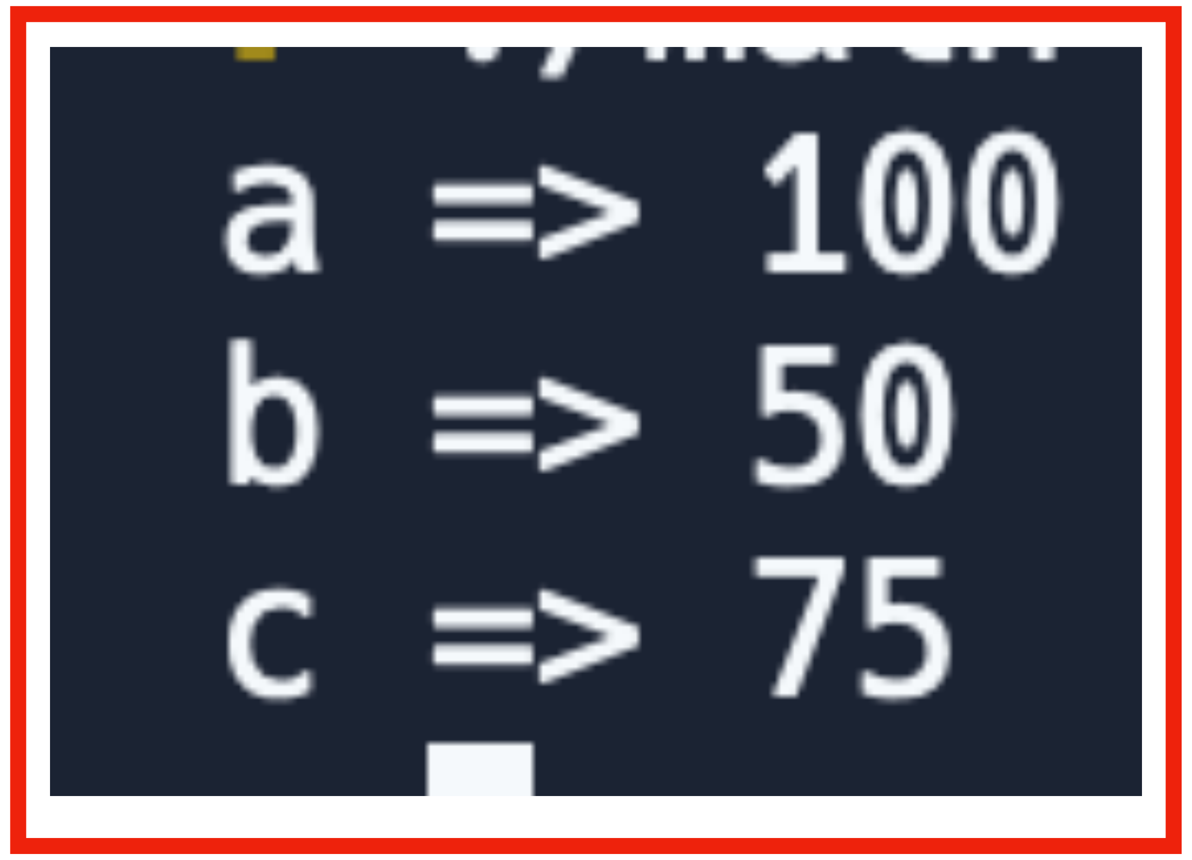
*In this function since the the tableSize was 10, I used 7 as my prime number on line 60

C++ Maps

- Maps are a sort of hash table that is part of the C++ STL
- Maps have two components: a **key value** and a **mapped value**
- The **key value** is used to sort and identify the elements in the map
- The **mapped value** stores the data that is associated with its specific key
- There can be no duplicate keys in a map
- The mapped values for a key can be accessed using the **bracket operator**
- https://cplusplus.com/reference/map/map/

```cpp
1  #include<iostream>
2  #include<map>
3  using namespace std;
4
5  int main(){
6
7      map<char,int> myMap;
8
9      myMap['a'] = 100;
10     myMap['b'] = 50;
11     myMap['c'] = 75;
12
13     for (map<char,int>::iterator it=myMap.begin(); it!=myMap.end(); it++){
14         cout << it->first << " => " << it->second << endl;
15     }
16
17     return 0;
18 }
```

OUTPUT:



- Line 2: include the map from the C++ library
- Line 7: initialize a map with **char** as the key value and **int** as the mapped value
- Line 9-11: Insert different values into the corresponding keys
- Line 13: Create a for loop that goes from the beginning of the map to the end
- Line 14: Print the key value (**first**) and the mapped value (**second**)

containsDuplicates

Given an integer array and its size, return **true** if the array contains any duplicates and **false** otherwise.

containsDuplicates({1, 2, 3, 4, 3}, 5) → **true**

containsDuplicates({1, 1}, 2) → **true**

containsDuplicates({1, 2, 3, 4, 5}, 5) → **false**

```
bool containsDuplicates(int arr[ ], int size){

}
```

containsDuplicates

```
bool containsDuplicate(int arr[], int size){
    map<int,bool> m;
    for(int i = 0; i < size; i++){
        if(m[arr[i]] == true){
            return true;
        }
        else{
            m[arr[i]] = true;
        }
    }
    return false;
}
```

Valid Anagram

An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Given two strings ***s*** and ***t***, return **true** if ***t*** is an anagram of ***s***, and **false** otherwise.

isAnagram(“listen”, “silent”) ⟶ **true**

isAnagram(“hello”, “goodbye”) ⟶ **false**

```
bool isAnagram(string s, string t){

}
```

Valid Anagram

```
bool isAnagram(string s, string t){
    map<char,int> sMap;
    map<char,int> tMap;
    if(s.length() != t.length()){
        return false;
    }
    for(int i = 0; i < s.length(); i++){
        sMap[s[i]]++;
        tMap[t[i]]++;
    }
    for(int i = 0; i < s.length(); i++){
        if(sMap[s[i]] != tMap[s[i]]){
            return false;
        }
    }
    return true;
}
```