# STL Containers

- Sequence Containers – store sequences of values
  - vector, deque, list
- Container Adapters – specialized interfaces to general containers
  - stack, queue, priority_queue
- Associative Containers – use "keys" to access data rather than position (Account #, ID, SSN, …)
  - map
  - Multimap
  - set
  - multiset

# Associative Containers

- Similar to vector & list – other storage structures with operations to access & modify elements.
- Stores elements based on a key.
- Key can consist of one or more attributes to uniquely identify each element (we will assume only one attribute).
- Example: Department of Motor Vehicles (DMV) uses license-plate # to identify a vehicle.
- Main difference is that an associative container uses the key rather than an index (vector) or linear search (list) to retrieve an element.

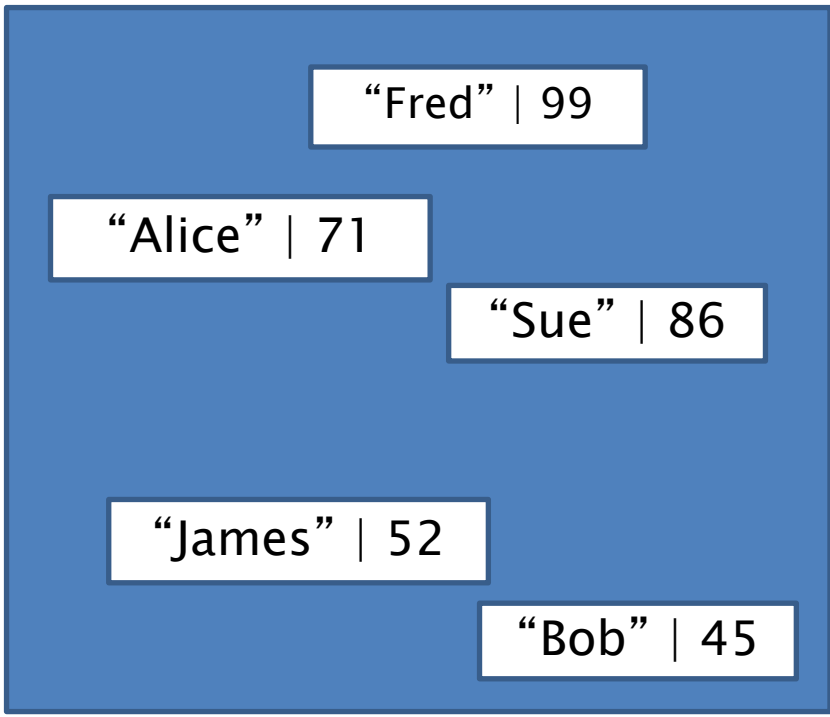
# Associative Containers: unordered_map

- Stores a set of (key, value) pairs
- Each key has one value
- Implemented as a hash table

```
#include <unordered_map>
//define it with
//keys of type string
//and values of int
Unordered_map<string, int> um;
```

- Fast insert and delete

```
um["Fred"] = 99;
insert, erase
```




http://www.cplusplus.com/reference/unordered_map/unordered_map/

# STL Unordered_map: Data Storage

- An STL map is implemented as a tree-structure, where each node holds a "pair".

- Most important to know when retrieving data from the table
  - Some functions return the pair, not just the value

- A pair has two fields, *first* (holding the key) and *second* (holding the value)

# STL Unordered_map: Data Storage

- If you have a *pair object*, you can use the following code to print the key and value:

```
cout << myPairObject.first << " " <<
   myPairObject.second;
```

- If you have a *pointer to the pair object*, use the arrow operator instead

```
cout << myPairObject->first << " " <<
   myPairObject->second;
```

# STL Unordered_map: Data Storage

- Access element `at`
  - Returns a reference to the mapped value of the element identified with key k.
  - If k does not match the key of any element in the container, the function throws an out_of_range exception.

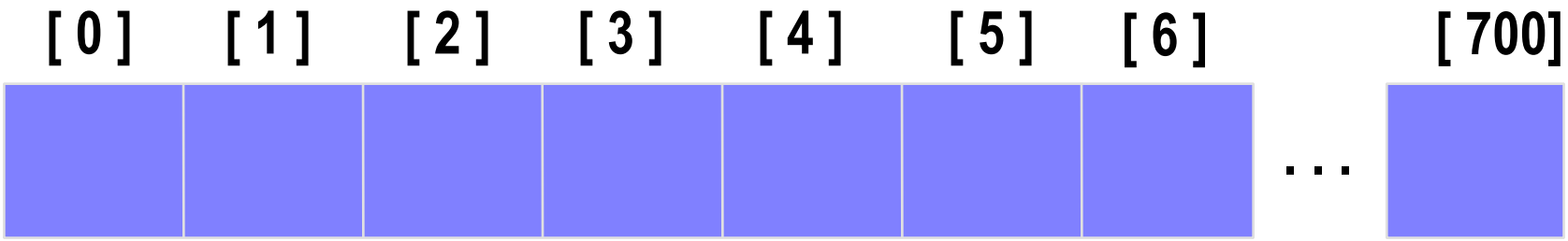
- Access element `[]`
  - If k matches the key of an element in the container, the function returns a reference to its mapped value.
  - If k does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value.

# unordered_map.cpp

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
    // Declaring umap to be of <string, int> type
    // key will be of string type and mapped value will
    // be of double type
    unordered_map<string, int> umap;

    // inserting values by using [] operator
    umap[“Fred”] = 99;
    umap[“Sue”] = 86;
    umap[“Bob”] = 45;

    // Traversing an unordered map
    unordered_map<string, int>:: iterator itr;
    for (itr = umap.begin(); itr != umap.end(); itr++) {
      cout << itr->first << " " << itr->second << endl;

}
```

# What is a Hash Table ?

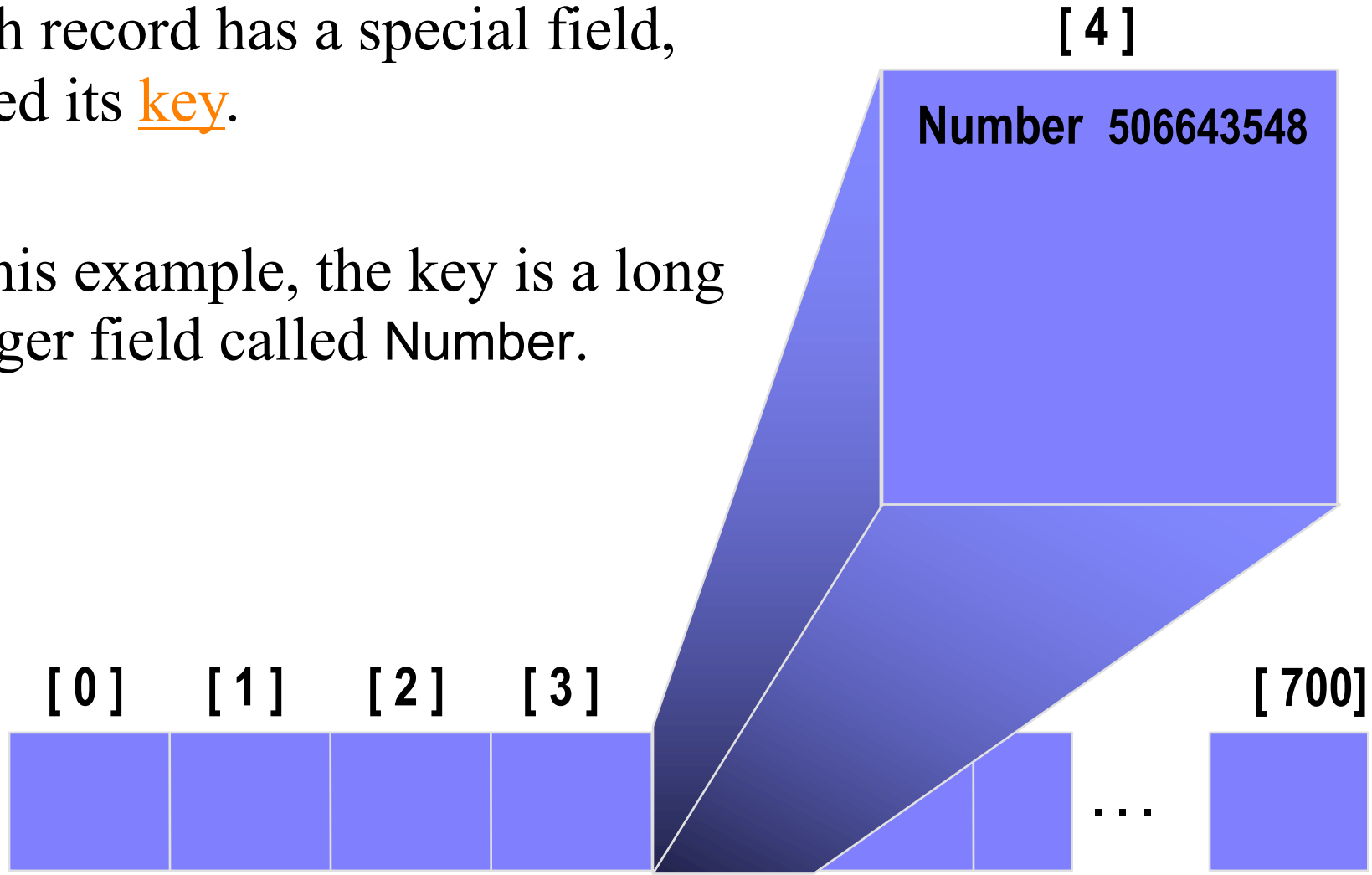
- Think about it as a Dictionary with <key,value> pairs.
- The simplest kind of hash table is an array of records.
- This example has 701 records.




**An array of records**

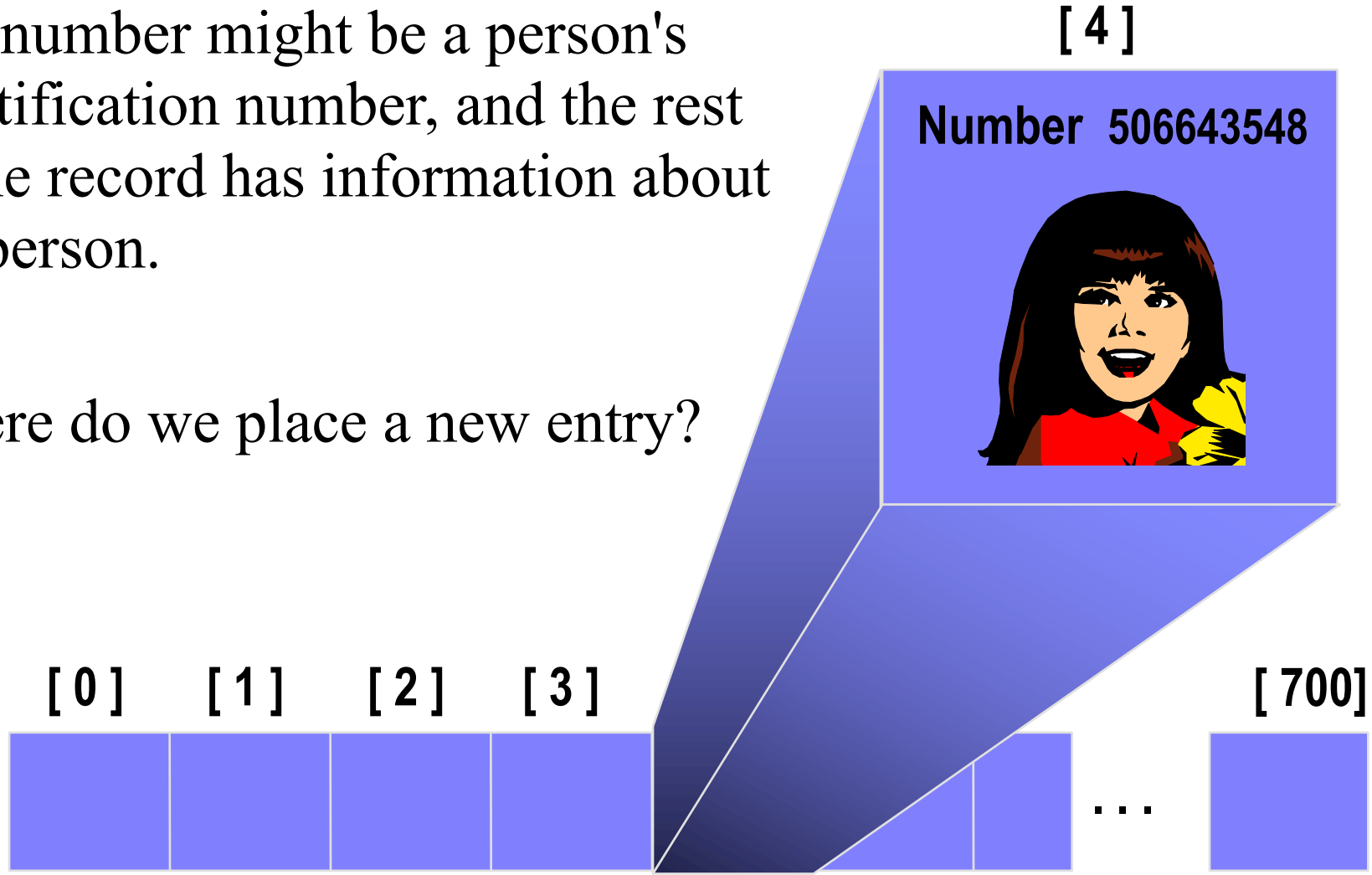
# What is a Hash Table ?

- Each record has a special field, called its key.
- In this example, the key is a long integer field called Number.



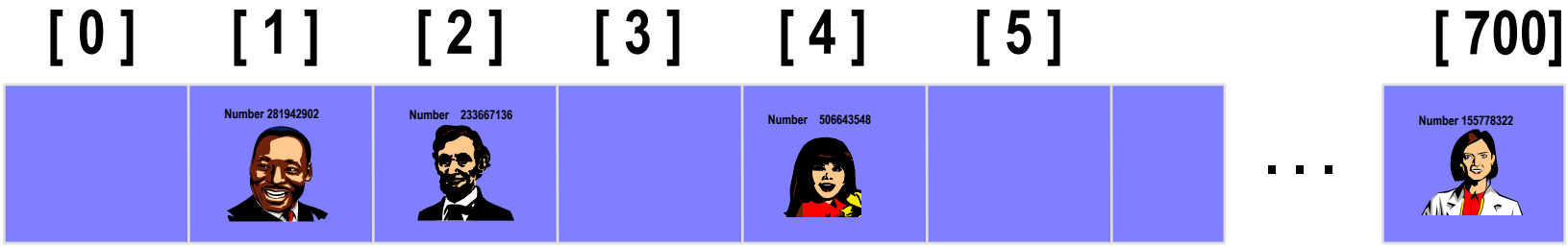
# What is a Hash Table ?

- The number might be a person's identification number, and the rest of the record has information about the person.
- Where do we place a new entry?



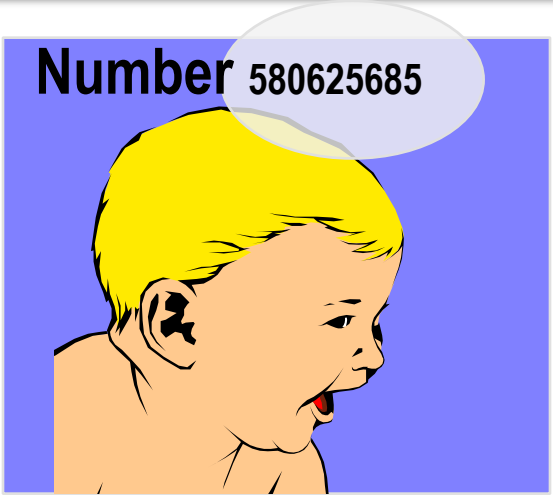
# What is a Hash Table ?

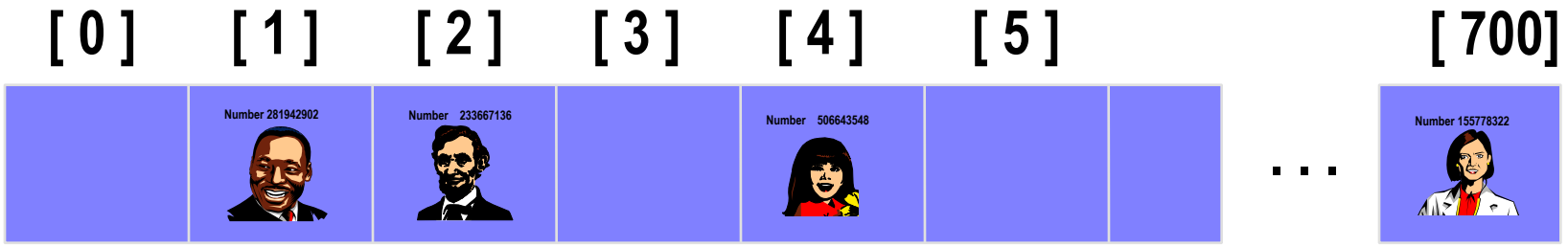
- When a hash table is in use, some spots contain valid records, and other spots are "empty".



# Inserting a New Record

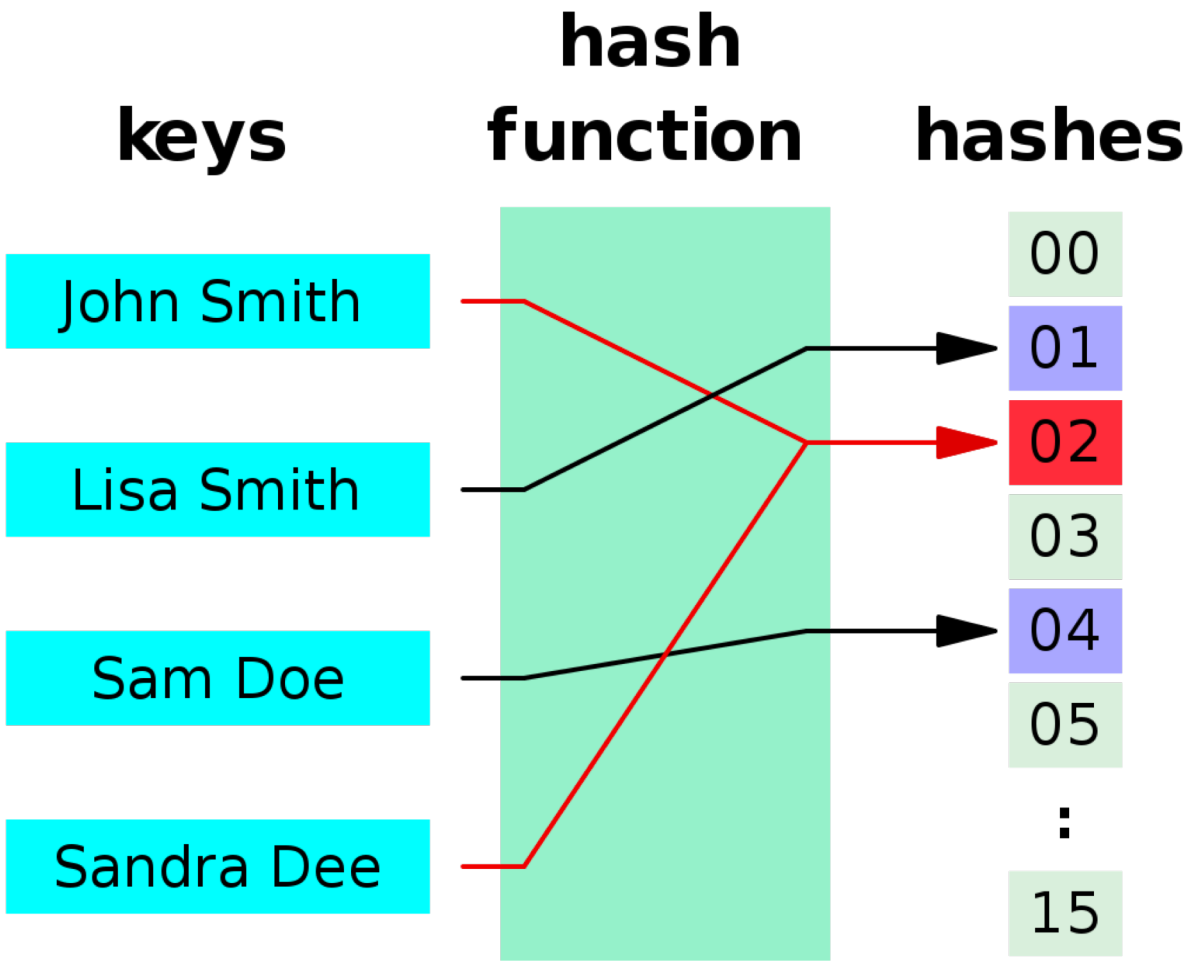
- In order to insert a new record, the **key** must somehow be converted to an array **index**.
- The index is called the **hash value** of the key.



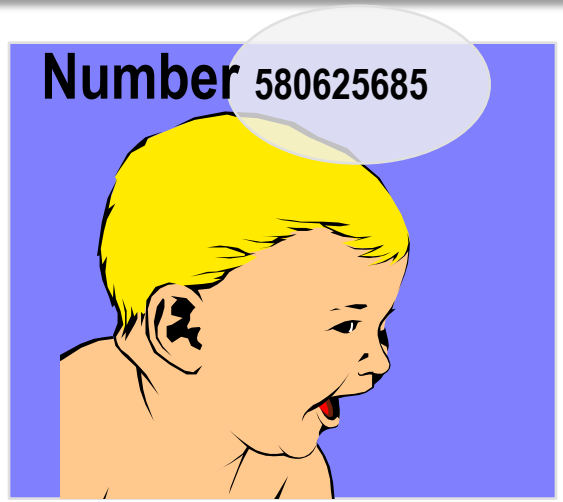

How can this conversion happen?

# Hash Function



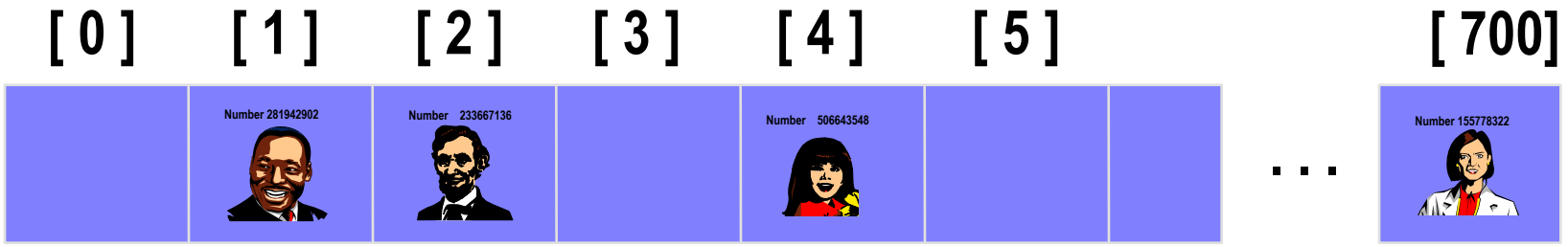
# Inserting a New Record

- Typical way create a hash value:




**(Number mod 701)**

*What is (580625685 mod 701) ?*

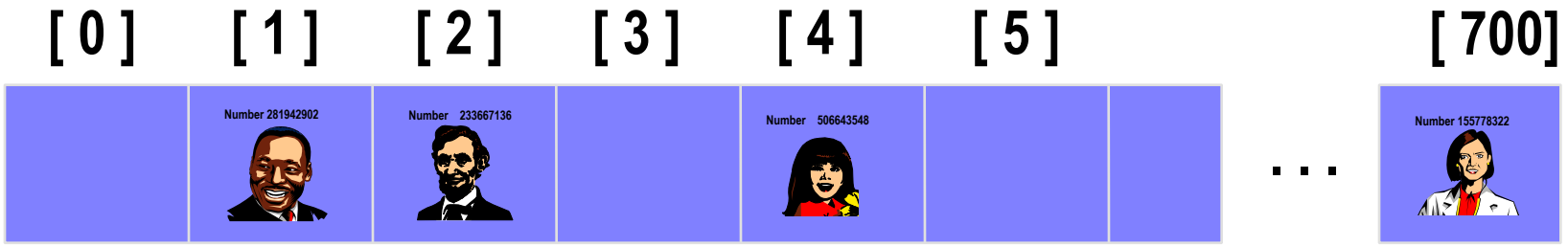


# Inserting a New Record

- Typical way create a hash value:

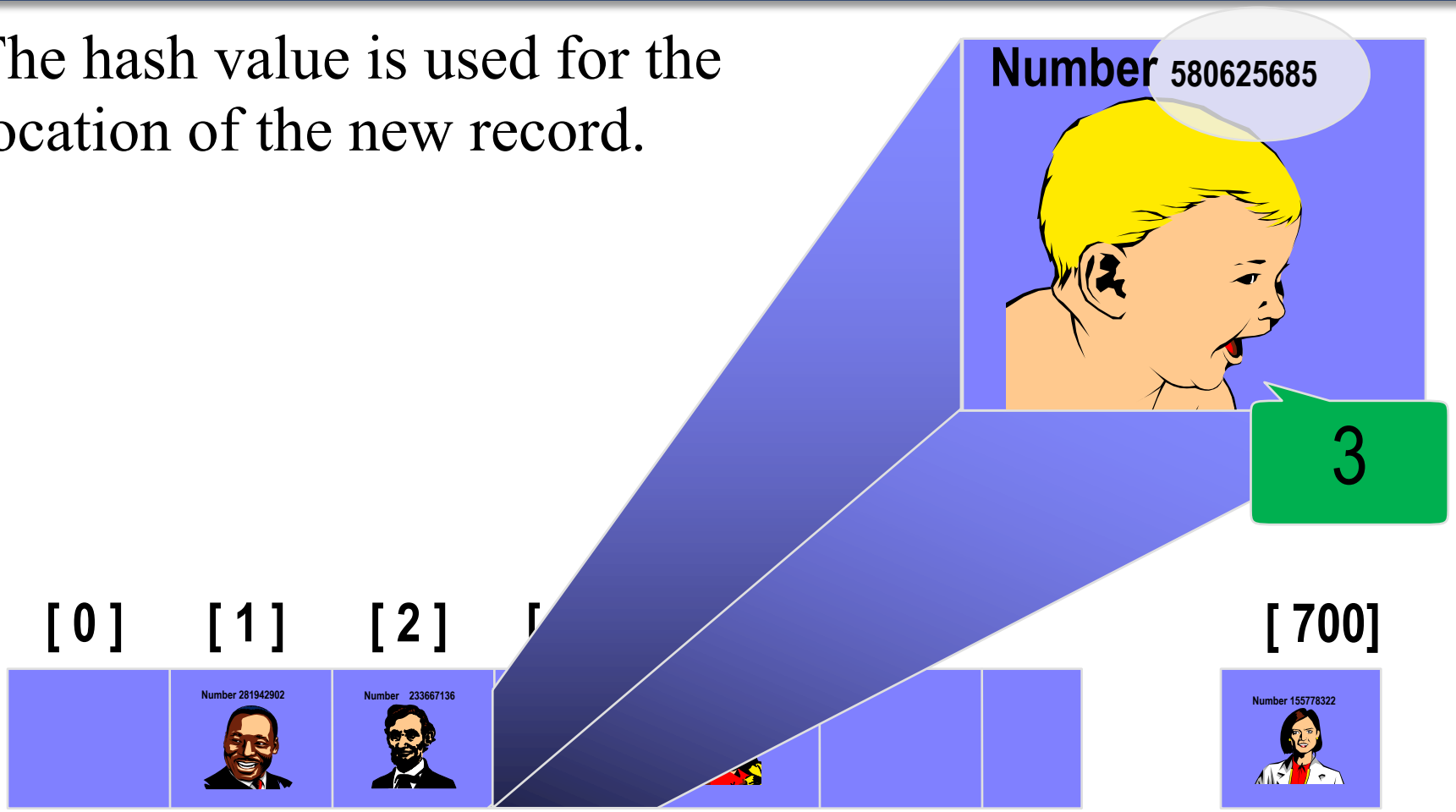
**(Number mod 701)**

*What is (580625685 mod 701) ?*

# Inserting a New Record

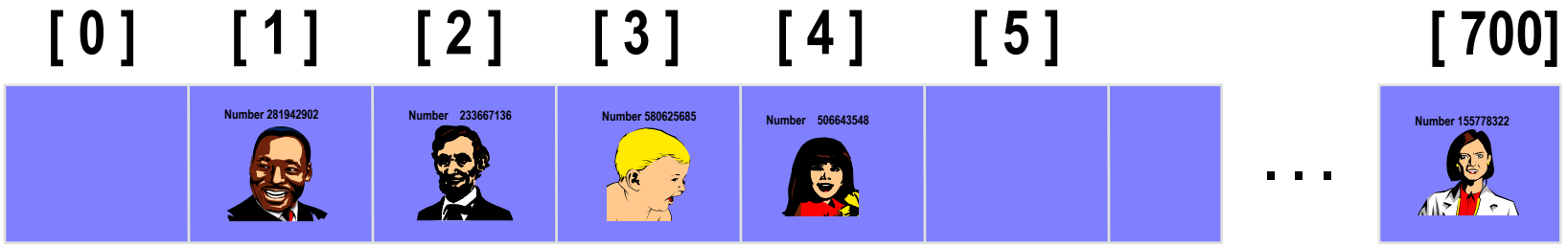
- The hash value is used for the location of the new record.



# Inserting a New Record

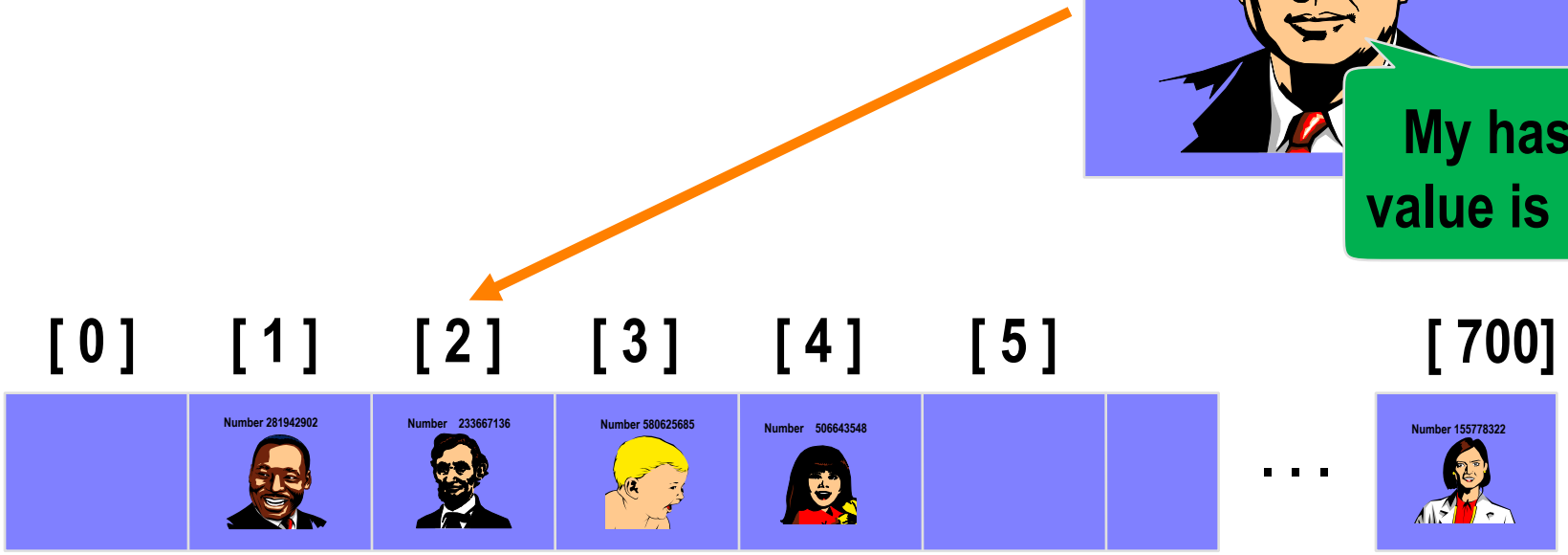
- The hash value is used for the location of the new record.

- Do you see any problem?



# Another Insertion

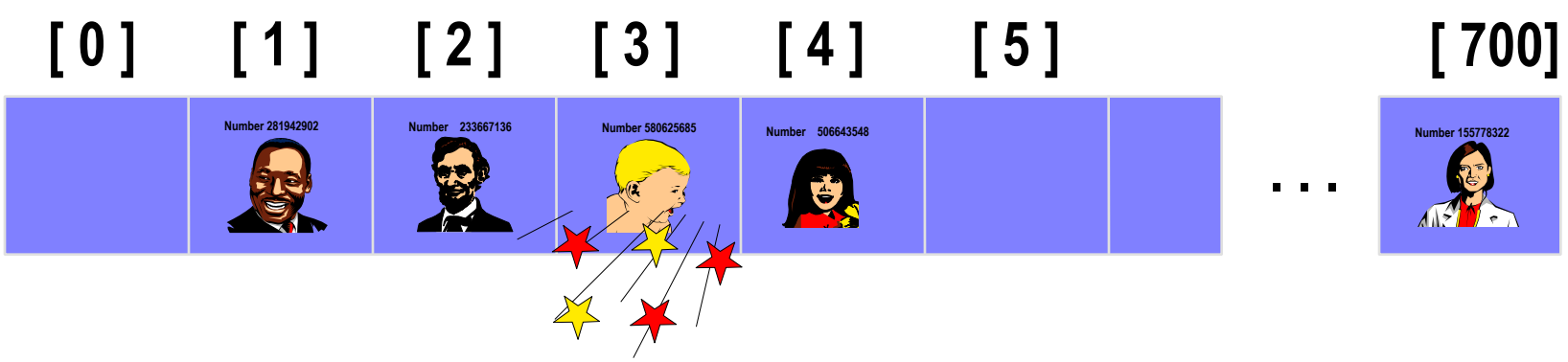
- Here is another new record to insert, with a hash value of 2.

# Collisions

- This is called a **collision**, because there is already another valid record at [2].

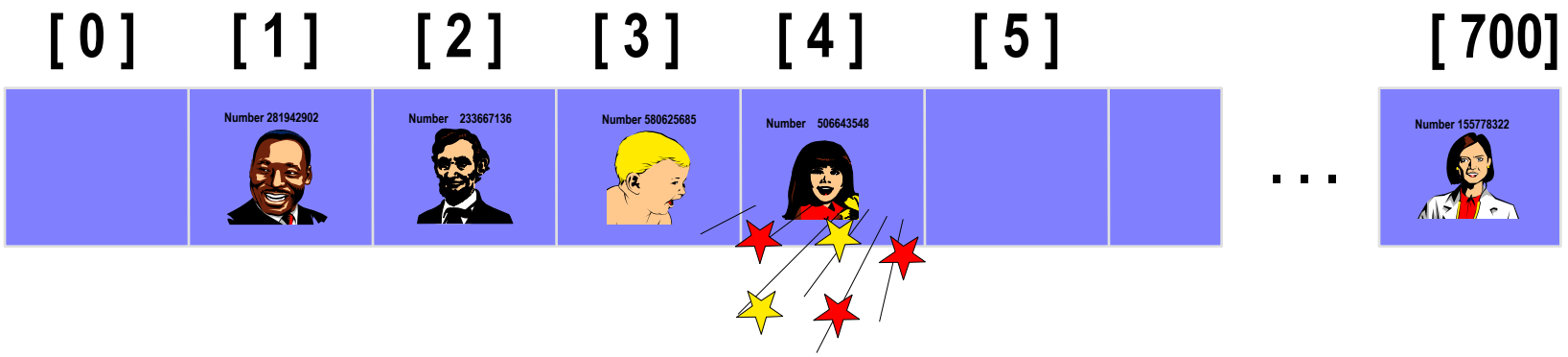
**When a collision occurs, move forward until you find an empty spot.**

# Collisions

- This is called a **collision**, because there is already another valid record at [2].

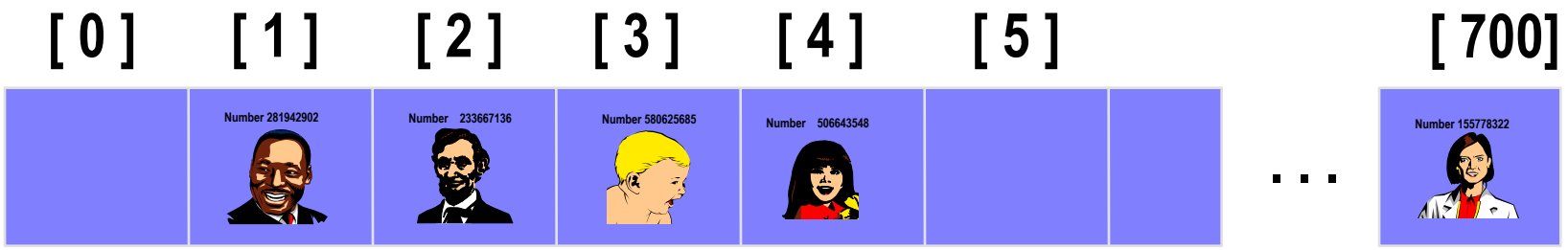
**When a collision occurs, move forward until you find an empty spot.**

# Collisions

- This is called a **collision**, because there is already another valid record at [2].

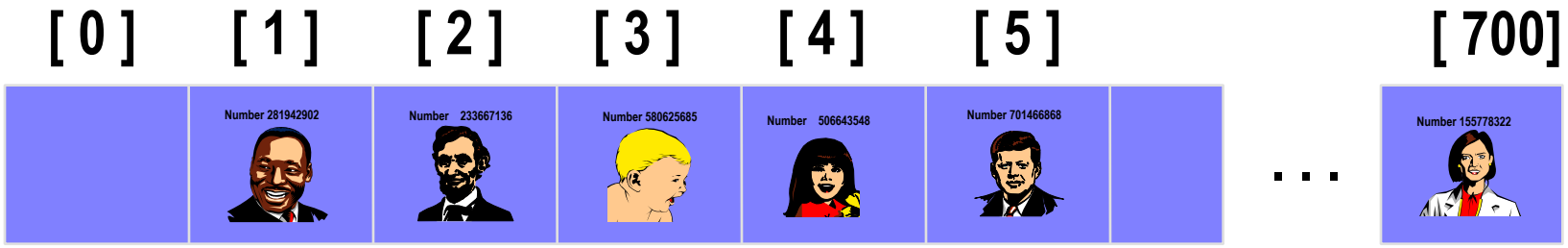
**When a collision occurs, move forward until you find an empty spot.**

# Collisions

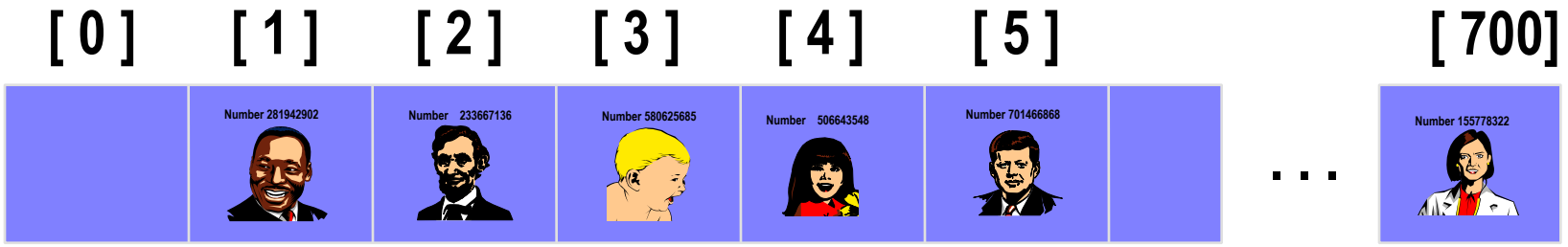
- This is called a **collision**, because there is already another valid record at [2].

The new record goes in the empty spot.



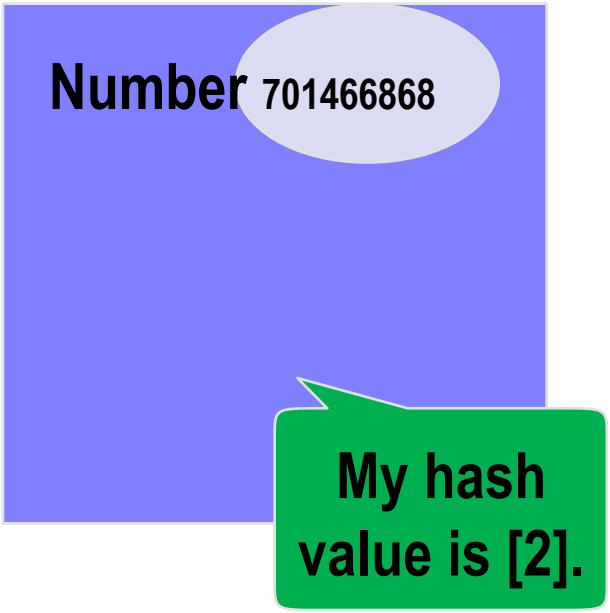
# Searching for a Key

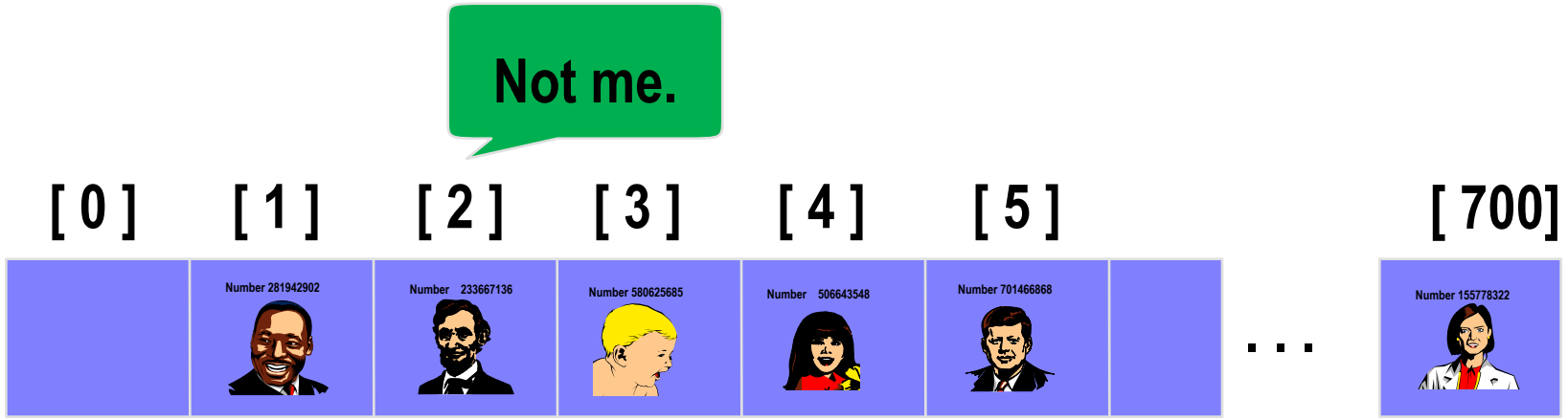
- The data that's attached to a key can be found fairly quickly.

# Searching for a Key

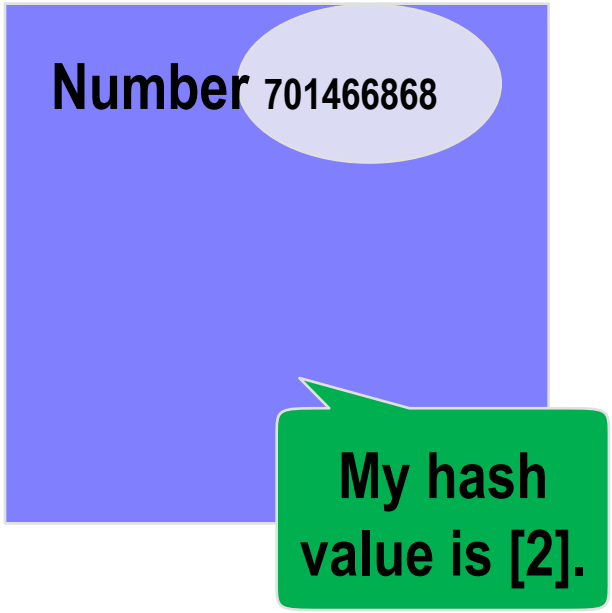
- Calculate the hash value.
- Check that location of the array for the key.



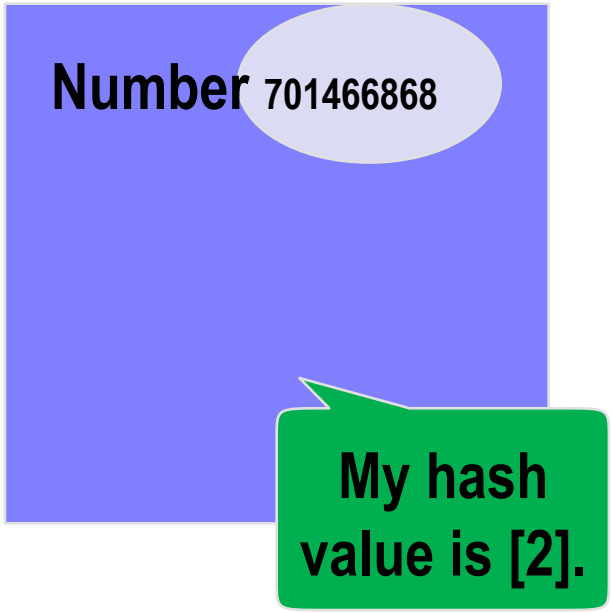
# Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.



# Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.



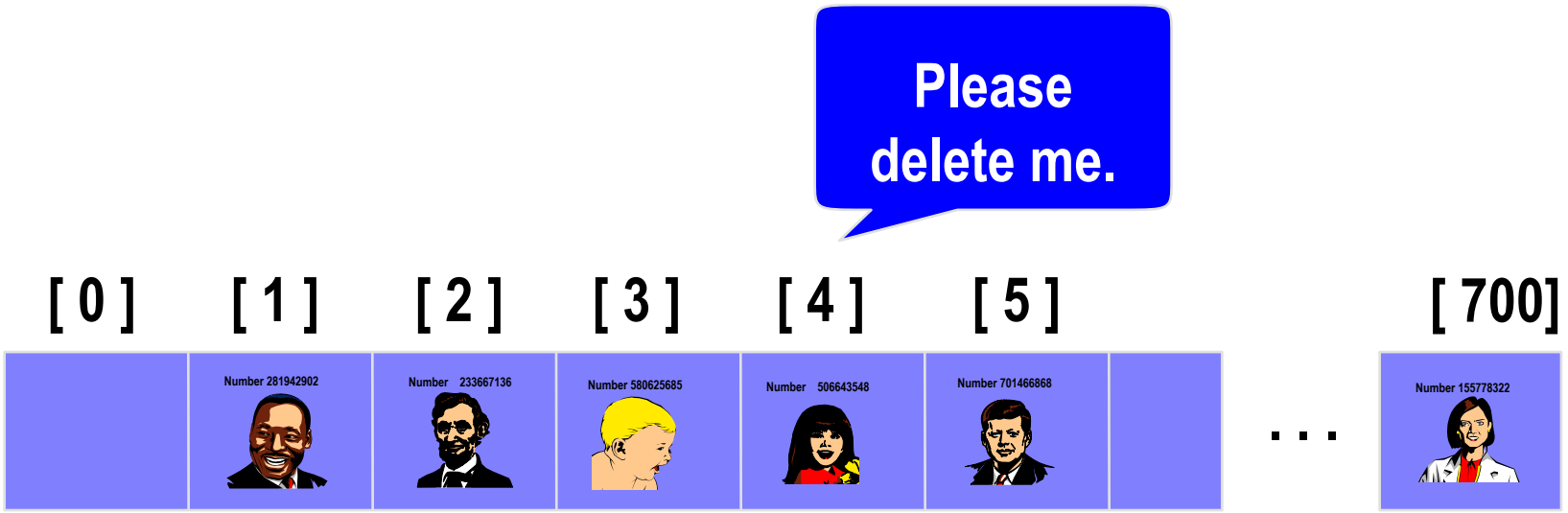
# Searching for a Key

- When the item is found, the information can be copied to the necessary location.

- What if you reach an empty spot or the end of array?

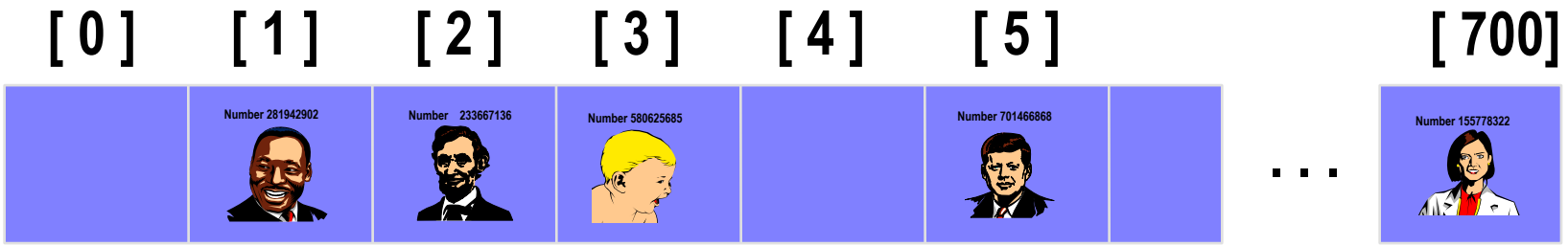
# Deleting a Record

- Records may also be deleted from a hash table.



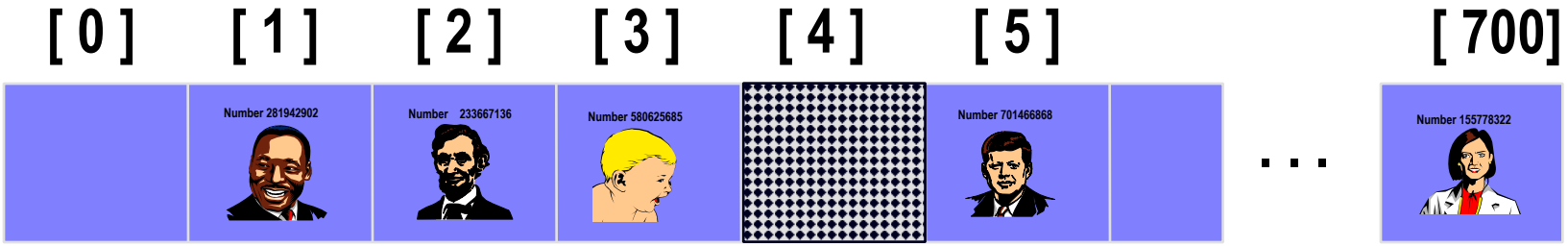
# Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.



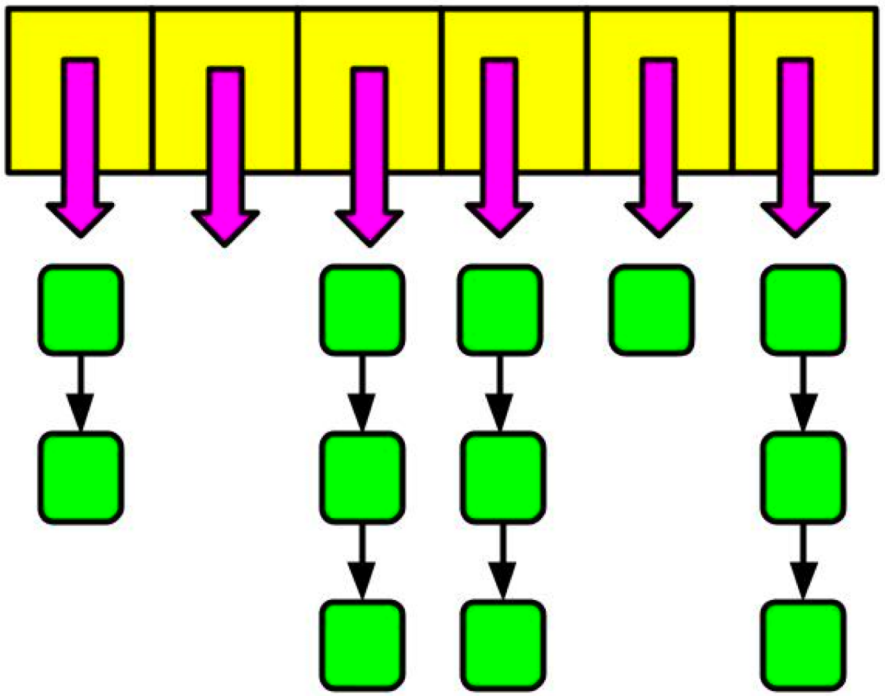
# Deleting a Record

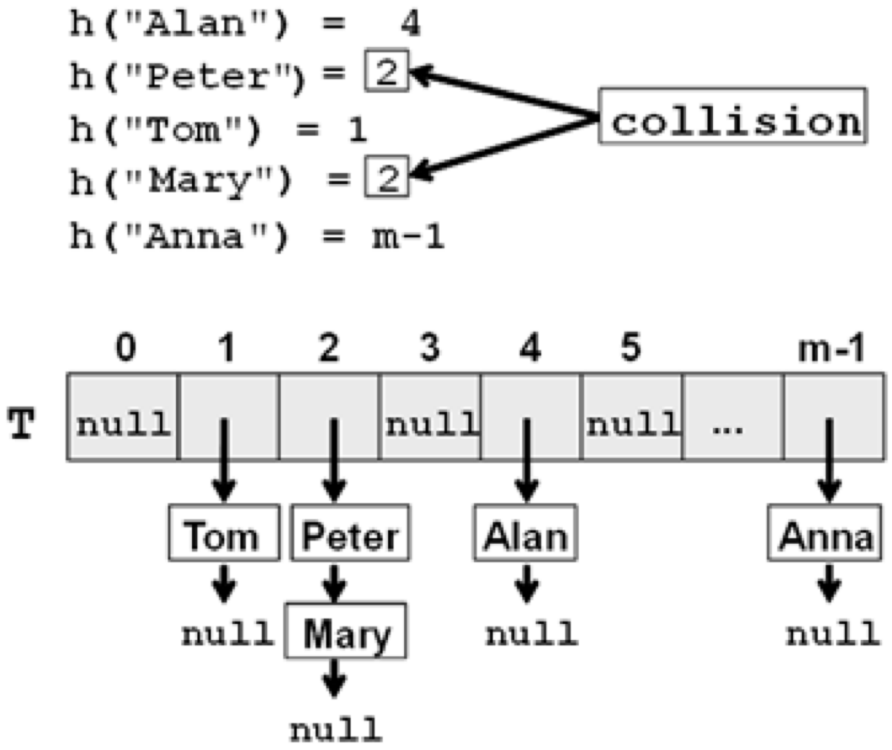
- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.
- The location must be marked in some special way so that a search can tell that the spot used to have something in it.



# List based Hash Table

Can we do better?





# Hash Map

- Implement Hash Map
  - Hash function just returns the remainder when the key is divided by the hash table size.
  - Hash entry (node) has key and value structure.
  - In addition, the class contains *search(key)* function to access mapped value by key, *insert(key,value)* function to put key-value pair in table and *remove(key)* function to remove hash node by key.
  - For collision resolution, separate chaining strategy could be used.

# Map Methods

Delegate operations to a list-based map at each cell:

**Algorithm search(*k*):**
***Output:*** The value of the key *k* or null
bucket = h(k)
return *A*[*bucket*].search(*k*) {delegate the search to the list-based map at *A*[*h*(*k*)]}

**Algorithm insert(*k*,*v*):**
bucket = h(k)
*t* = *A*[*bucket*].insert(*k*,*v*)
*n* = *n* + 1
return *t* {delegate the put to the list-based map at *A*[*h*(*k*)]}

**Algorithm remove(*k*):**
bucket = h(k)
*A*[*bucket*].remove(*k*)
*n* = *n* – 1 {delegate the remove to the list-based map at *A*[*h*(*k*)]}

# Implement Simple Hash Map: testHashMap.cpp

```cpp
#include<iostream>

using namespace std;

const int TABLE_SIZE = 128;

// HashEntry Class Declaration
class HashEntry
{
    public:
        int key;
        int value;
        HashEntry(int key, int value)
        {
            this->key = key;
            this->value = value;
        }
};

//HashMap Class Declaration
class HashMap
{
    private:
        HashEntry **table;
    public:
        HashMap()
        {
            table = new HashEntry * [TABLE_SIZE];
            for (int i = 0; i< TABLE_SIZE; i++)
            {
                table[i] = NULL;
            }
        }

        // Hash Function
        int HashFunc(int key)
        {
            return key % TABLE_SIZE;
        }

        void Insert(int key, int value)
        {
            int hash = HashFunc(key);
            while (table[hash] != NULL && table[hash]->key != key)
            {
                hash = HashFunc(hash + 1);
            }
            if (table[hash] != NULL)
                delete table[hash];
            table[hash] = new HashEntry(key, value);
        }

        // Search Element at a key
        int Search(int key)
        {
            int  hash = HashFunc(key);
            while (table[hash] != NULL && table[hash]->key != key)
            {
                hash = HashFunc(hash + 1);
            }
            if (table[hash] == NULL)
                return -1;
            else
                return table[hash]->value;
        }

        // Remove Element at a key
        void Remove(int key)
        {
            int hash = HashFunc(key);
            while (table[hash] != NULL)
            {
                if (table[hash]->key == key)
                    break;
                hash = HashFunc(hash + 1);
            }
            if (table[hash] == NULL)
            {
                cout<<"No Element found at key "<<key<<endl;
                return;
            }
            else
            {
                delete table[hash];
            }
            cout<<"Element Deleted"<<endl;
        }
```

```
89          ~HashMap()
90          {
91              for (int i = 0; i < TABLE_SIZE; i++)
92              {
93                  if (table[i] != NULL)
94                      delete table[i];
95                  delete[] table;
96              }
97          }
98  };
99
100 int main()
101 {
102     HashMap hash;
103     int key, value;
104     int choice;
105     while (1)
106     {
107         cout<<"\n---------------------"<<endl;
108         cout<<"Operations on Hash Table"<<endl;
109         cout<<"\n---------------------"<<endl;
110         cout<<"1.Insert element into the table"<<endl;
111         cout<<"2.Search element from the key"<<endl;
112         cout<<"3.Delete element at a key"<<endl;
113         cout<<"4.Exit"<<endl;
114         cout<<"Enter your choice: ";
115         cin>>choice;
```

```
117         switch(choice)
118         {
119         case 1:
120             cout<<"Enter element to be inserted: ";
121             cin>>value;
122             cout<<"Enter key at which element to be inserted: ";
123             cin>>key;
124             hash.Insert(key, value);
125             break;
126         case 2:
127             cout<<"Enter key of the element to be searched: ";
128             cin>>key;
129             if (hash.Search(key) == -1)
130             {
131             cout<<"No element found at key "<<key<<endl;
132             continue;
133         }
134         else
135         {
136             cout<<"Element at key "<<key<<" : ";
137             cout<<hash.Search(key)<<endl;
138         }
139             break;
140         case 3:
141             cout<<"Enter key of the element to be deleted: ";
142             cin>>key;
143             hash.Remove(key);
144             break;
145         case 4:
146             exit(1);
147         default:
148            cout<<"\nEnter correct option\n";
149        }
150     }
151     return 0;
152 }
```

# Double Hashing

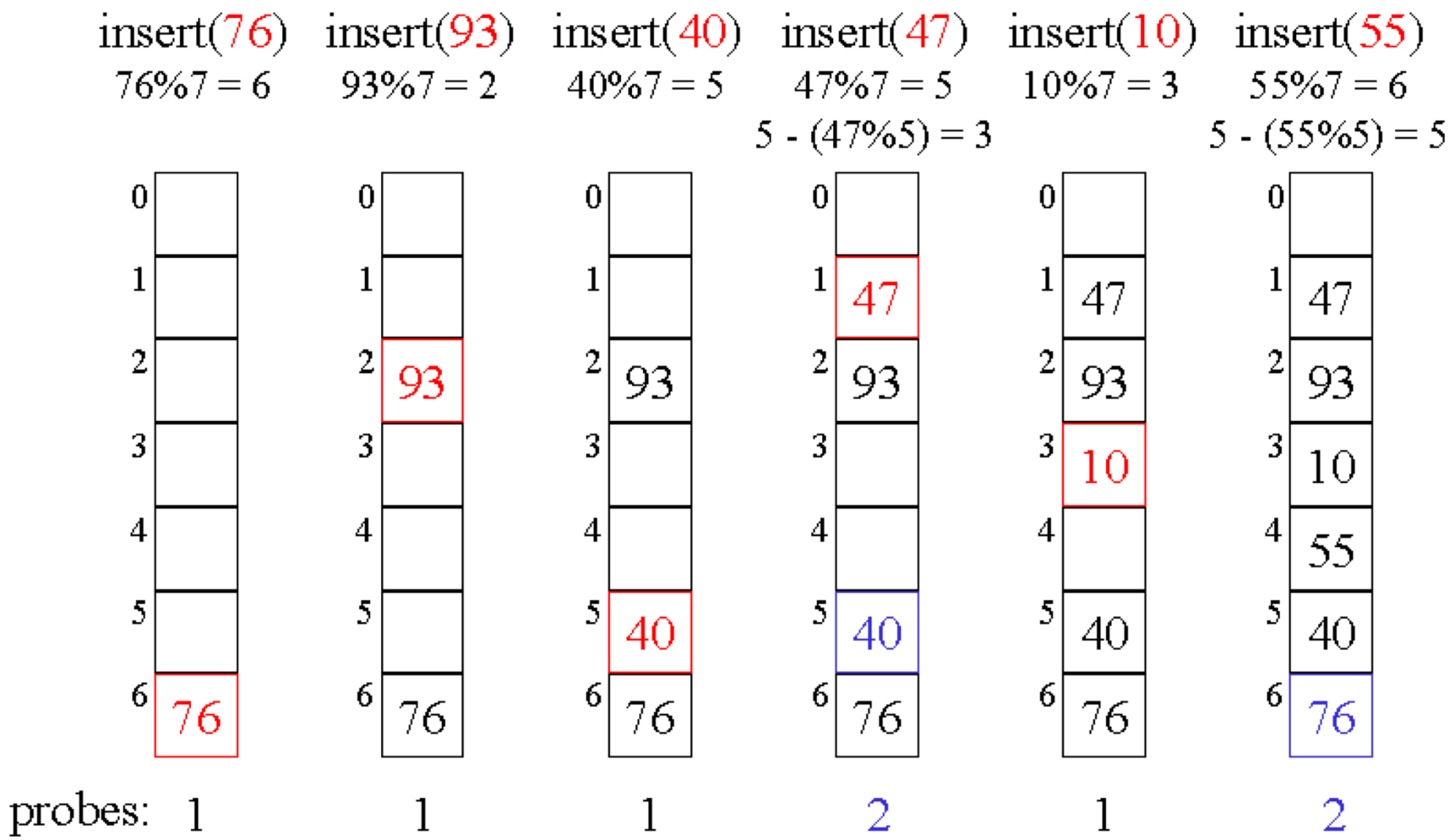
**Double hashing** is a collision resolving technique in open addressed hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

*Double hashing can be done using:*

***(hash1(key) + i * hash2(key)) % TABLE_SIZE***

First hash function is typically: **hash1(key) = key % TABLE_SIZE**
Second hash function can be: **hash2(key) = PRIME – (key %PRIME)** where PRIME is a prime smaller than the TABLE_SIZE.

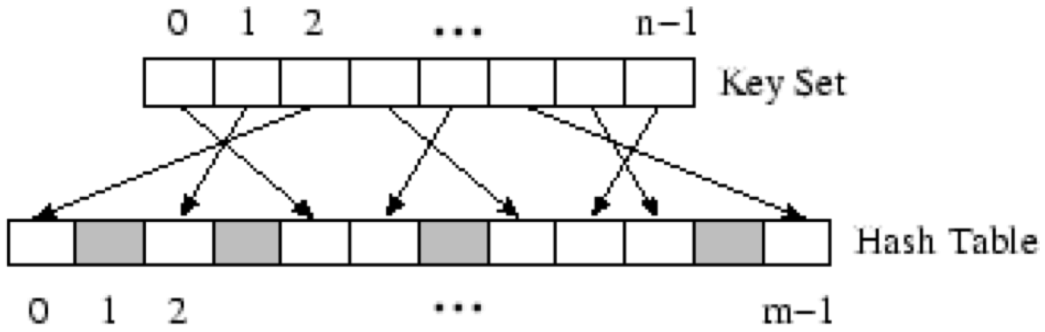
# Double Hashing



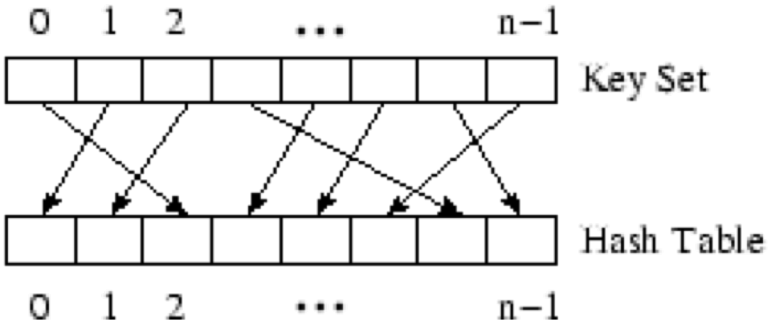
# Good Hash Function

- A hash function is ***perfect*** if it maps items to buckets with no collisions.
- The number of items and all keys should be known to design a perfect hashing that results in constant complexity for insert, search, and remove operations.

Perfect hashing function




Minimal perfect hashing function




https://emn178.github.io/online-tools/sha256.html

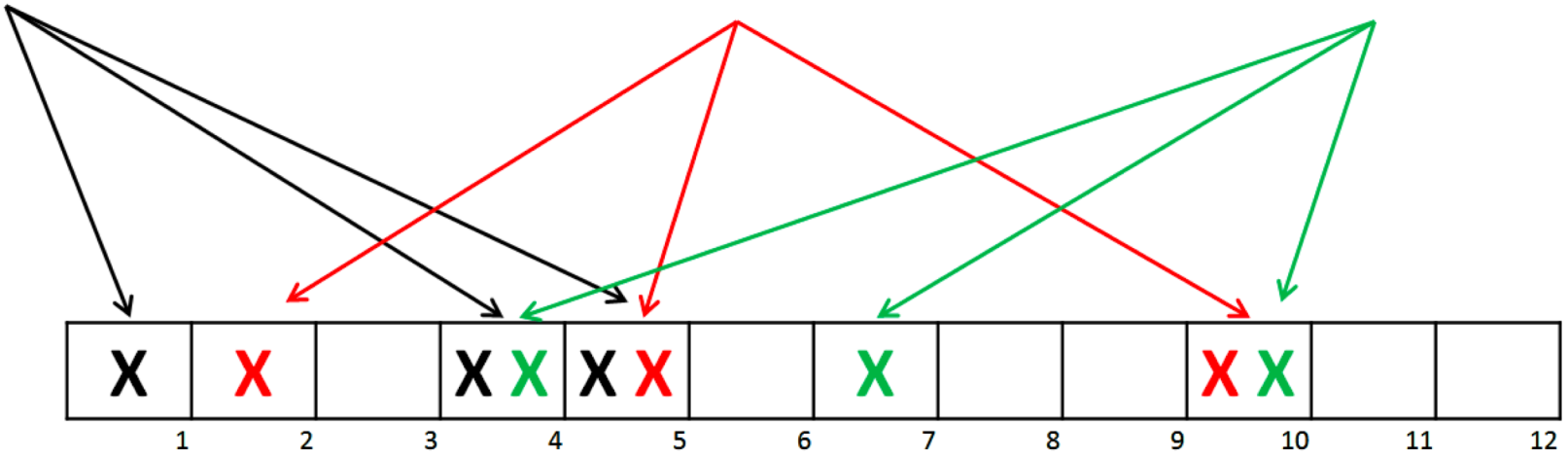
# Bloom filter

A space-efficient data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a set or not with high probability.

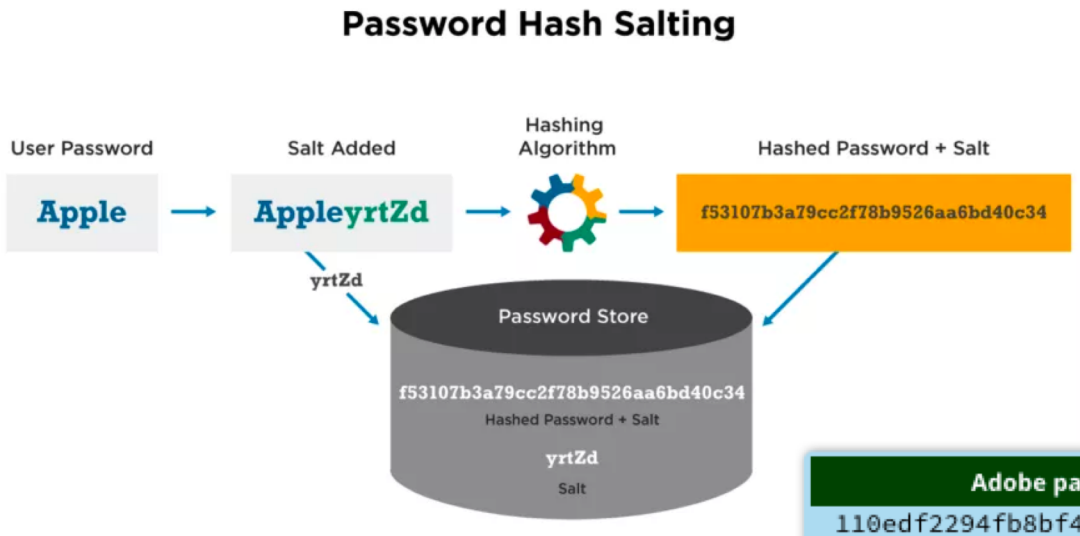
h1("oracle") = 1
h2("oracle") = 4
h3("oracle") = 5

h1("database") = 2
h2("database") = 5
h3("database") = 10

h1("filter") = 4
h2("filter") = 7
h3("filter") = 10



# Storing user passwords




Hashing is used extensively in computer security:

- Authentication
- Signature
- Storing password
- Blockchain

| Adobe password data | | Password hint | |
|---|---|---|---|
| 110edf2294fb8bf4 | -> | numbers 123456 | ❶ 123456 |
| 110edf2294fb8bf4 | -> | ==123456 | |
| 110edf2294fb8bf4 | -> | c'est "123456" | |
| 8fda7e1f0b56593f e2a311ba09ab4707 | -> | numbers | ❷ 12345678 |
| 8fda7e1f0b56593f e2a311ba09ab4707 | -> | 1-8 | |
| 8fda7e1f0b56593f e2a311ba09ab4707 | -> | 8digit | |
| 2fca9b003de39778 e2a311ba09ab4707 | -> | the password is password | ❸ password |
| 2fca9b003de39778 e2a311ba09ab4707 | -> | password | |
| 2fca9b003de39778 e2a311ba09ab4707 | -> | rhymes with assword | |
| e5d8efed9088db0b | -> | q w e r t y | ❹ qwerty |
| e5d8efed9088db0b | -> | ytrewq tagurpidi | |
| e5d8efed9088db0b | -> | 6 long qwert | |
| ecba98cca55eabc2 | -> | sixxone | ❺ 111111 |
| ecba98cca55eabc2 | -> | 1*6 | |
| ecba98cca55eabc2 | -> | sixones | |

Adobe's password database format made many users' passwords easy to recover