CS 2124: DATA STRUCTURES
Spring 2024

Lecture 13

Topics: Hash Tables

Quiz - 2

30th April – 6:00 AM till End of day (11: 58 PM)

- Quiz -2 :
  - Points: 5
  - Date: 30th April
  - Quiz availability Time: 6:00 AM till End of day (11: 58 PM)
  - Number of MCQ: 7 (Each MCQ Points vary based on difficulty)
  - Once the Quiz starts students will have 20 Min to complete it.
  - The quiz cannot be paused or stopped. It must be attempted in one sitting
  - Kindly do not refresh the page.
  - One question will be visible at one time.
  - Once you answer the question (i.e. submitted) it cannot be changed
  - Note: Do keep a pencil, paper, and a calculator with you while attempting the quiz

Final Exam

7th May, 8:00 AM - 3:00 PM

- Final Exam: 7th May 2024 (From 8:00 AM to 3:00 PM)
- On Canvas -> Quizzes
- Points: 20
- Number of MCQ: Based on Points (most probably 20 – 22)
- Once the exam starts students will have **45 Min to complete it.**
- The exam cannot be paused or stopped. It must be attempted in one sitting
- Kindly do not refresh or go back to the previous question (press back on the browser) as that is not allowed.
- Students with Accommodation approval from the UTSA S.D Office should email me before and after completing their Exam so that they can be informed about the points and questions they need to attempt.
- **Attempt the exam on time as the exam cannot be reopened once it is completed**
- **Do keep a pencil, paper, and a calculator with you while attempting the exam**

Topics

1. Hashing
   - Hash Table
   - Hash Function
2. Insertion
3. Collusion Due To Insertion
4. Hash Function (Example)
5. Hash Function (Implementation)

6. Collusion Avoidance
7. Searching
8. Deletion
9. Brent’s Method
10. Hashing Case/Example

Hashing

Definition/basic Idea

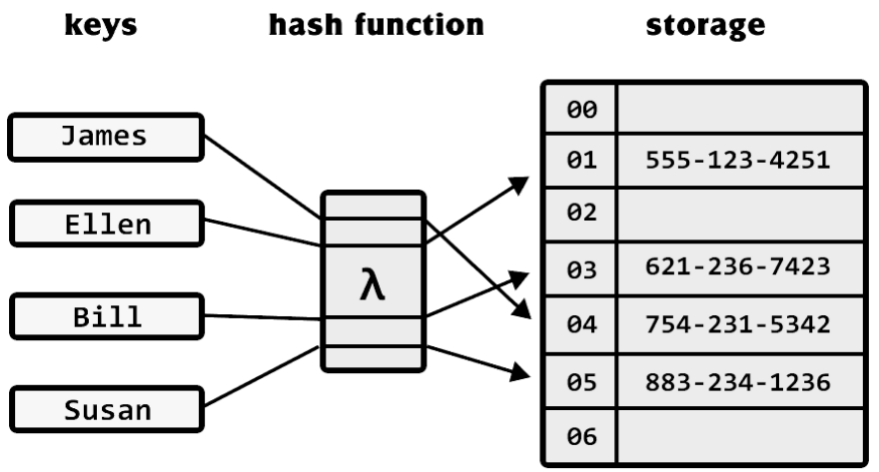
Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number (ABC123) that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

A **hash table**, or a **hash map**, is a data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number).

Hashing
Working



- Assume that you have an object and you want to assign a key to it to make searching easy.
- To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values.
- However, in cases where the **keys are large** and **cannot be used directly** as an index, you should use hashing.

- In hashing, large keys are converted into small keys by using **hash functions**.
- The values are then stored in a data structure called **hash table**.
- The idea of hashing is to distribute entries (key/value pairs) uniformly across an array.
- Each element is assigned a key (converted key).
- By using that key you can access the element in **O(1)** time.
- Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing
Applications

1. Hash tables are often used to implement associative arrays, sets and caches.
2. Like arrays, hash tables provide constant-time O(1) lookup on average, regardless of the number of items in the table.
3. The (rare) worst-case lookup time in most hash table schemes is O(n).
4. Compared to other associative array data structures, hash tables are most useful when we need to store a large numbers of data records.
5. Hash tables may be used as in-memory data structures. Hash tables may also be adopted for use with persistent data structures; database indexes commonly use disk-based data structures based on hash tables.
6. Hash tables are also used to speed-up string searching in many implementations of data compression.
7. In computer chess, a hash table can be used to implement the transposition table.

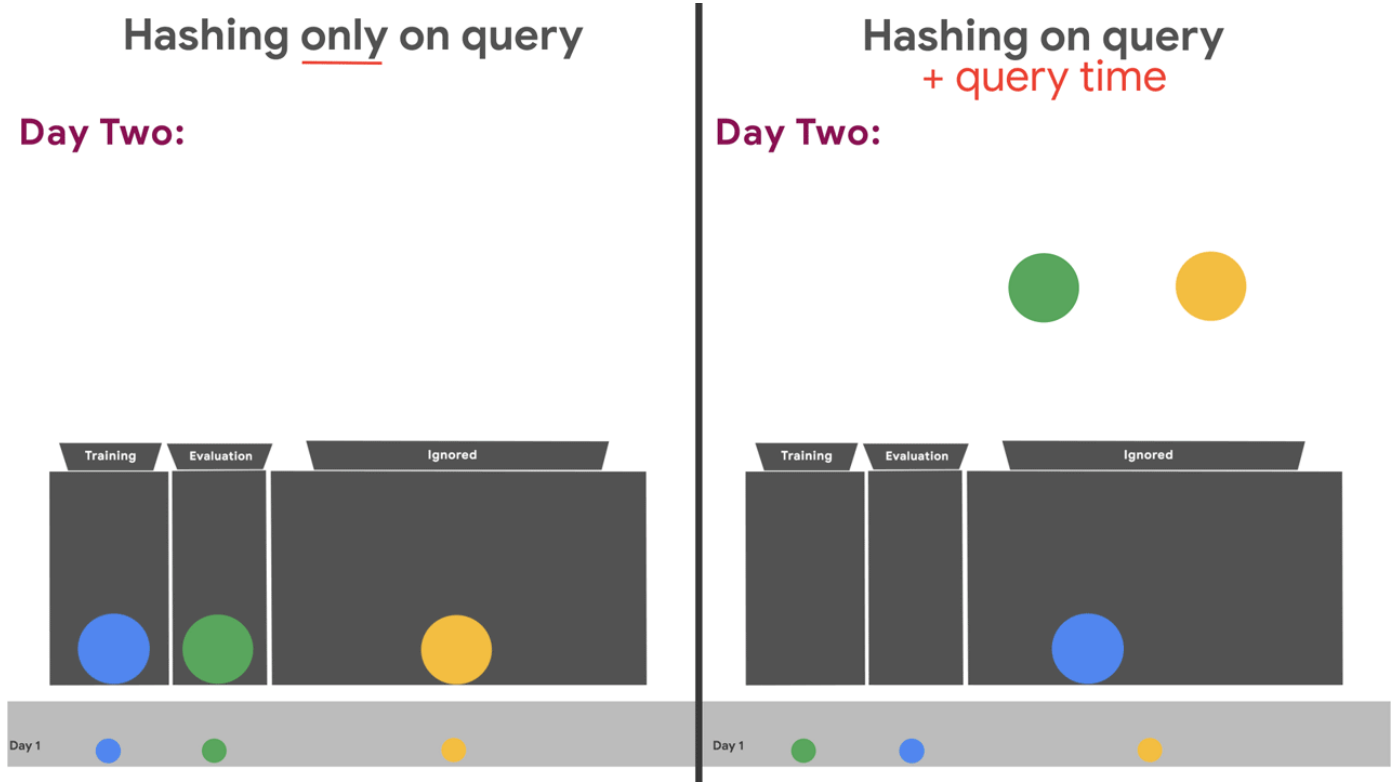
Hashing

Applications (Randomization – Machine Learning) - Link

- **Practical Considerations**
- Make your data generation pipeline reproducible. Say you want to add a feature to see how it affects model quality. For a fair experiment, your datasets should be identical except for this new feature. If your data generation runs are not reproducible, you can't make these datasets.

- In that spirit, make sure any randomization in data generation can be made deterministic:

1. Seed your random number generators (RNGs).
2. Use invariant hash keys.



Hashing

- Hashing is implemented in two steps:
  1. An element is converted into an integer by using a **hash function**. This element can be used as an index to store the original element, which falls into the **hash table**.
  2. The element is stored in the hash table where it can be quickly retrieved using hashed key.
     - hash = hashfunc(key)
     - index = hash % array_size

- In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

Hash Function

- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

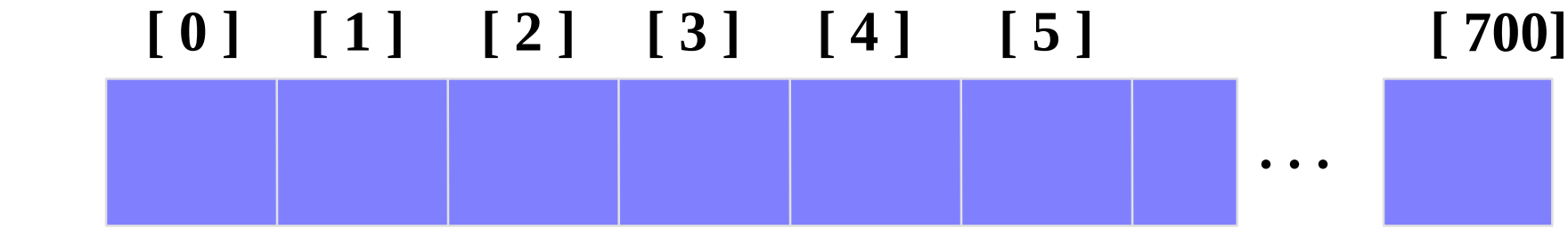
To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. **Easy to compute:** It should be easy to compute and must not become an algorithm in itself.
2. **Uniform distribution:** It should provide a uniform distribution across the hash table and should not result in clustering.
3. **Less collisions:** Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Hash Table

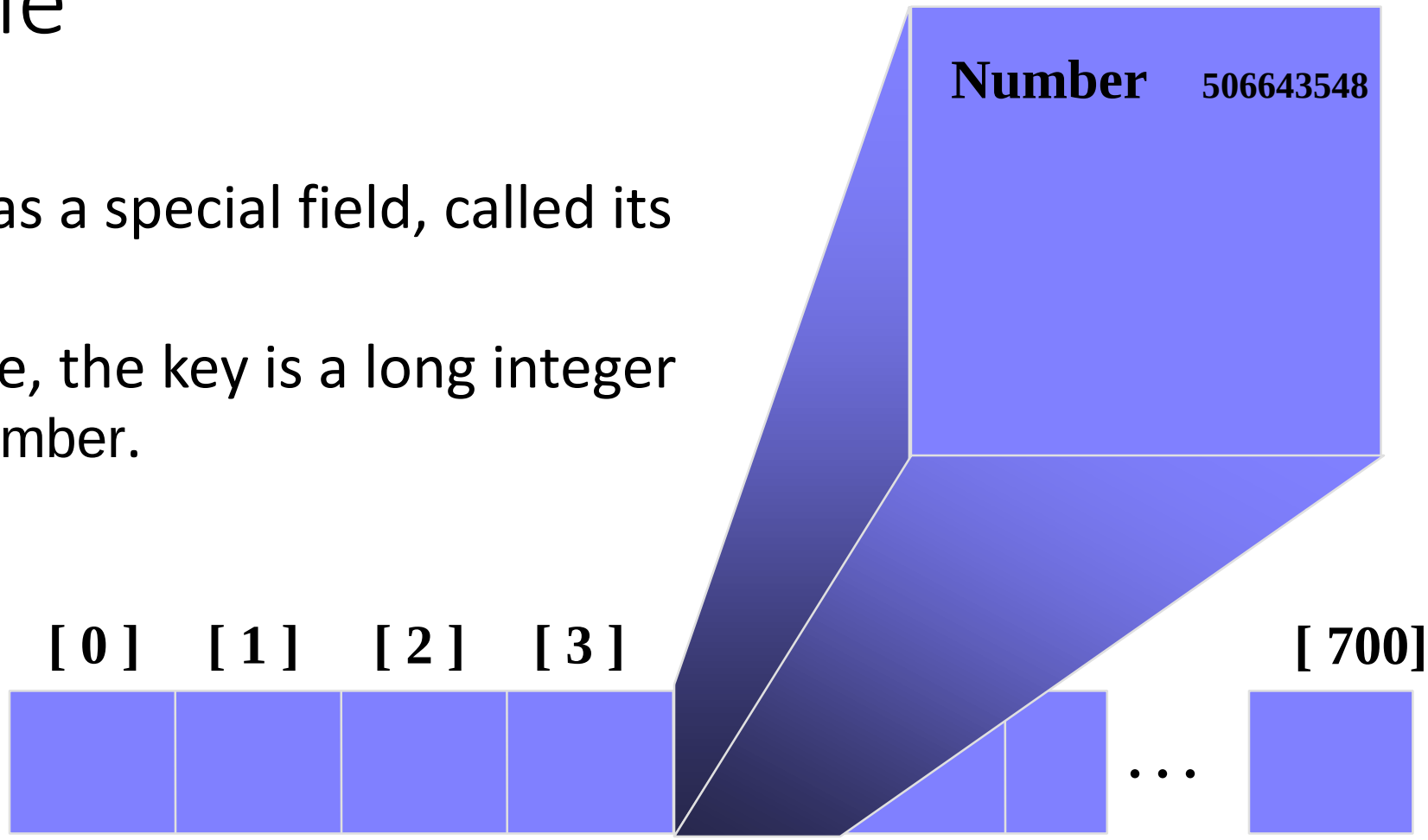
- The simplest kind of hash table is an array of records.
- This example has 701 records.



An array of records

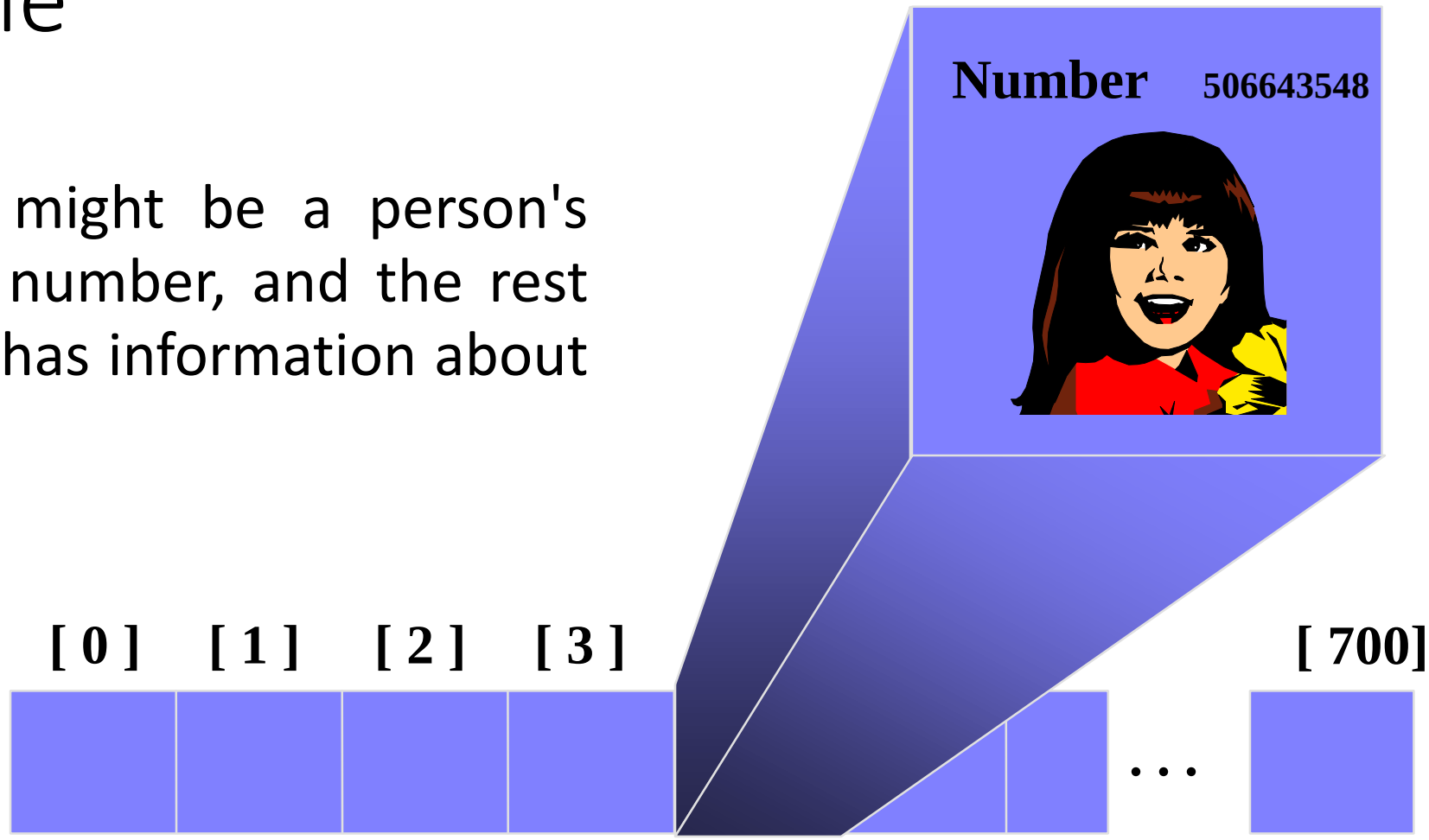
Hash Table

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



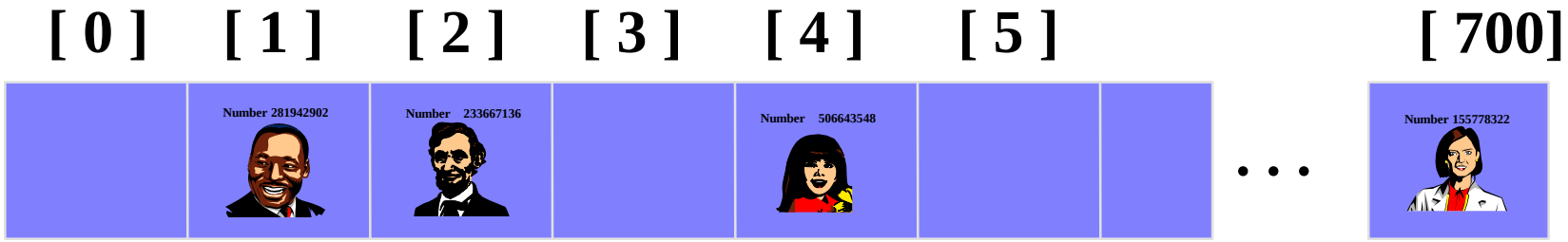
Hash Table

- The number might be a person's identification number, and the rest of the record has information about the person.



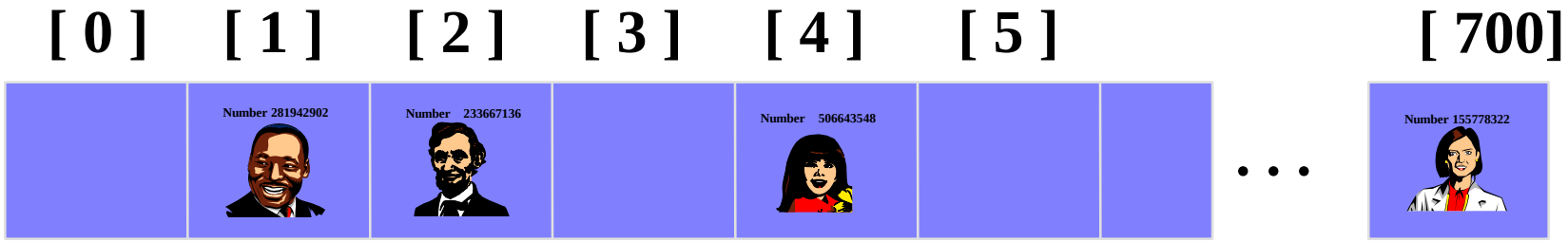
Hash Table

- When a hash table is being used as a dictionary, some of the array locations are in use, and other spots are "empty", waiting for a new entry to come along.
- Often times, the empty spots are identified by a special key.
- For **example**, if all our identification numbers are positive, then we could use 0 as the Number that indicates an empty spot.
- With this drawing, locations [0], [3], [6], and maybe some others would all have Number=0.



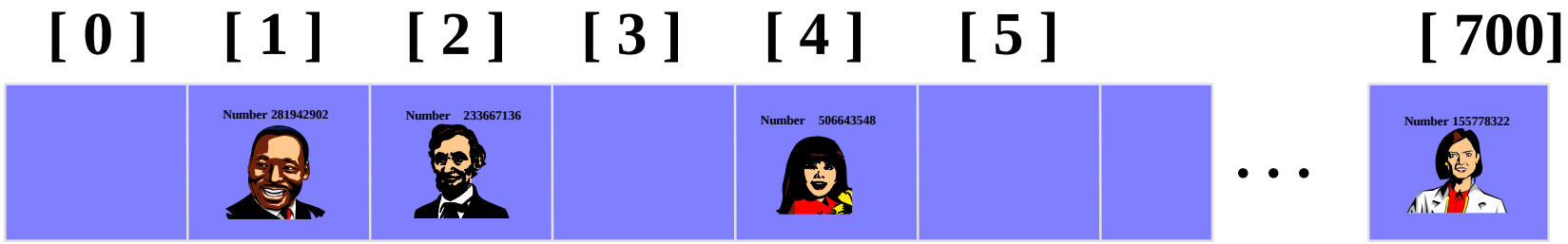
Inserting a New Record

- In order to insert a new entry, the key of the entry must somehow be converted to an index in the array.
- For our example, we must convert the key number into an index between 0 and 700.
- The conversion process is called **hashing** and the index is called the **hash value** of the key.



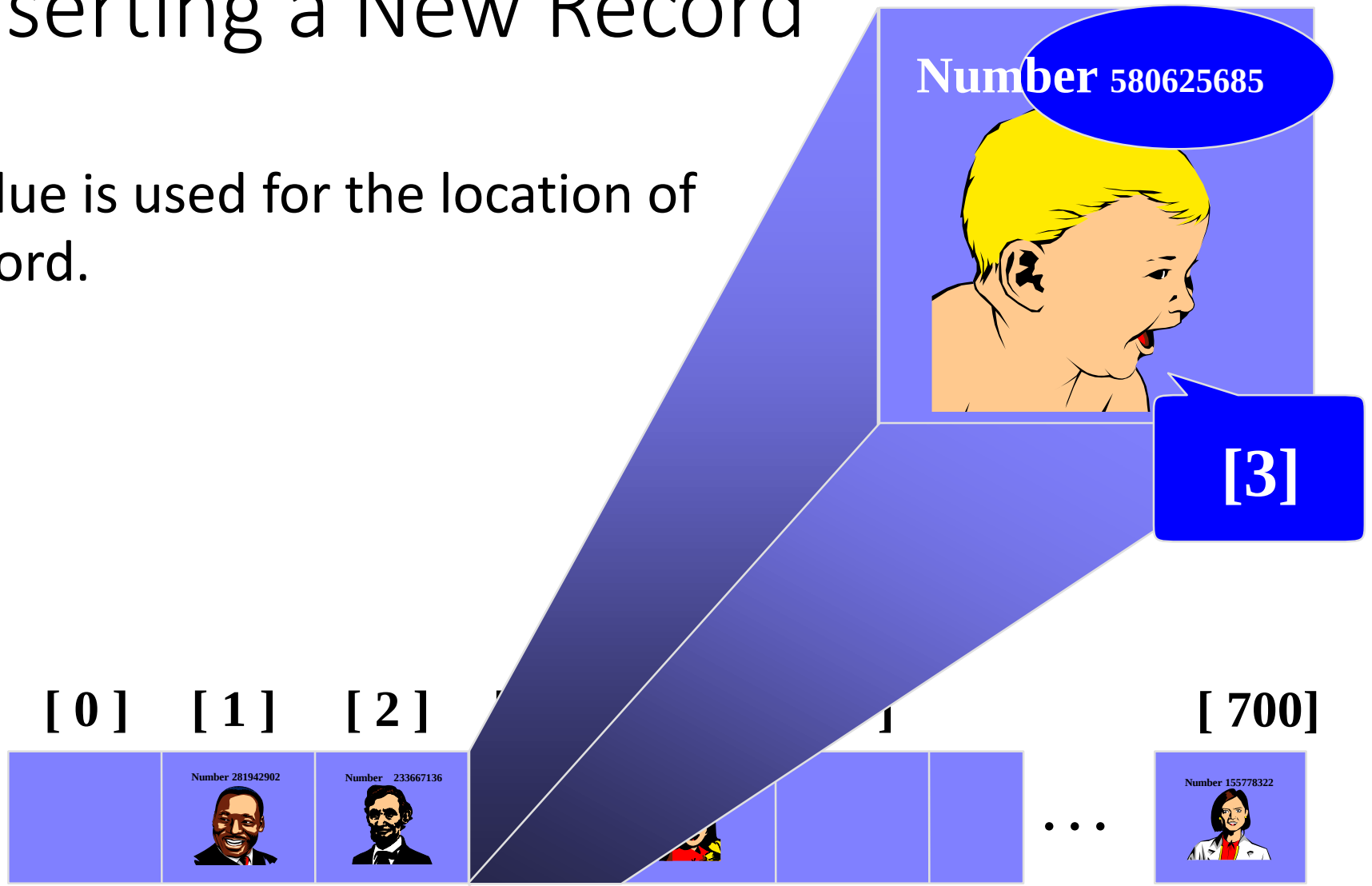
Inserting a New Record

- There are many ways to create hash values. Here is a typical approach.
- Take the key mod 701 (which could be anywhere from 0 to 700).
- (Number mod 701) = (580,625,685 mod 701) ?



Inserting a New Record

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



Inserting a New Record (Allocate the values)

Total 8 Values:

1. 36
2. 18
3. 72
4. 43
5. 6
6. 10
7. 5
8. 12

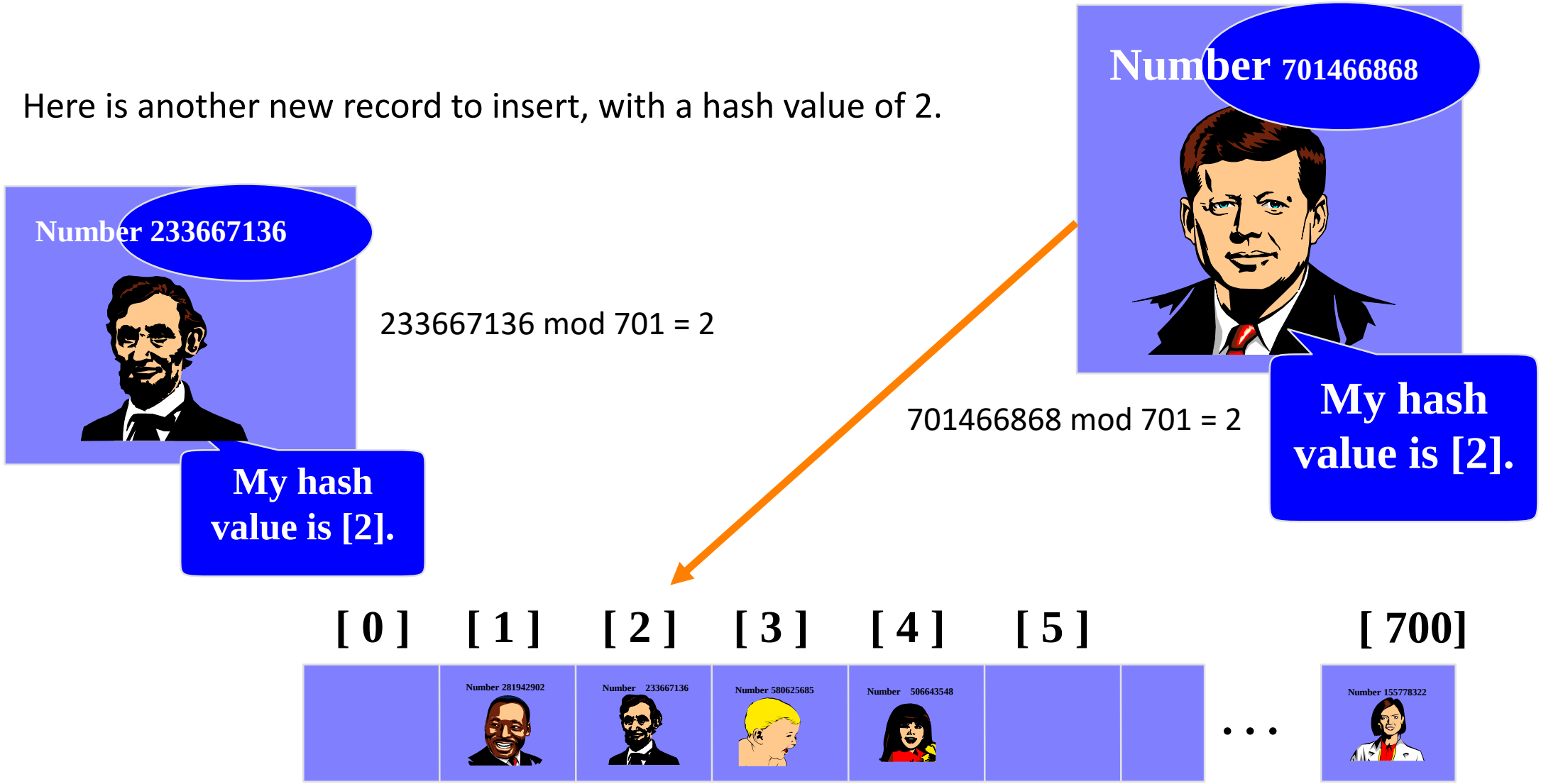
[0]	
[1]	
[2]	
[3]	
[4]	
[5]	
[6]	
[7]	

Array

Hash key = key % table size

Collisions

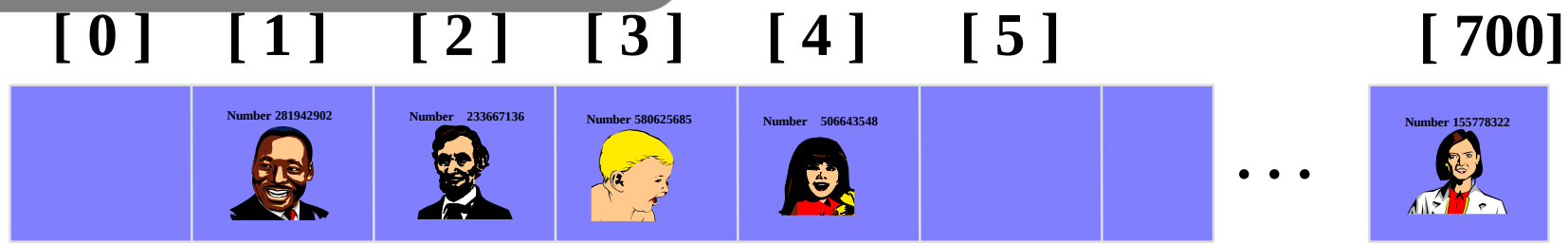
- Here is another new record to insert, with a hash value of 2.



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.

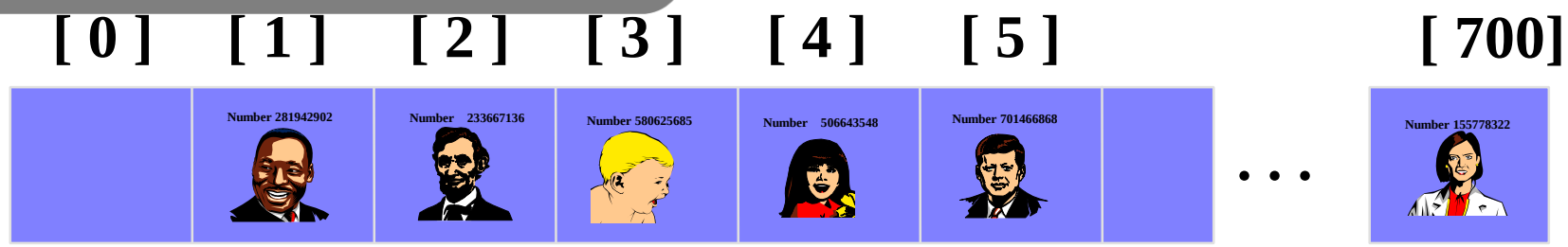


Collisions

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

0	Jeffery
1	Helen
2	
3	
4	victor
5	Anna
6	

The new record goes in the empty spot.



Need For A Good Hash Function

- Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.
- To compute the index for storing the strings, use a hash function that states the following:
    1. The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

Character	ASCII
a	97
b	98
c	99
d	100
e	101
f	102

2. As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions).
    I. It is recommended that you use *prime numbers in case of modulo.
    II. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively.
    III. Since all the strings contain the same characters with different permutations, the sum will 599.

Prime numbers are famously only divisible by 1 and themselves. Thus, choosing to set your hash table length to a large prime number will greatly reduce the occurrence of collisions.

Need For A Good Hash Function

- Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.
- To compute the index for storing the strings, use a hash function that states the following:
  1. The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.
  2. As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.
  3. The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.

599 % 597 = 2

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc

Here, it will take O(n) time (where n is the number of strings) to access a specific string. This shows that the hash function is not a good hash function.

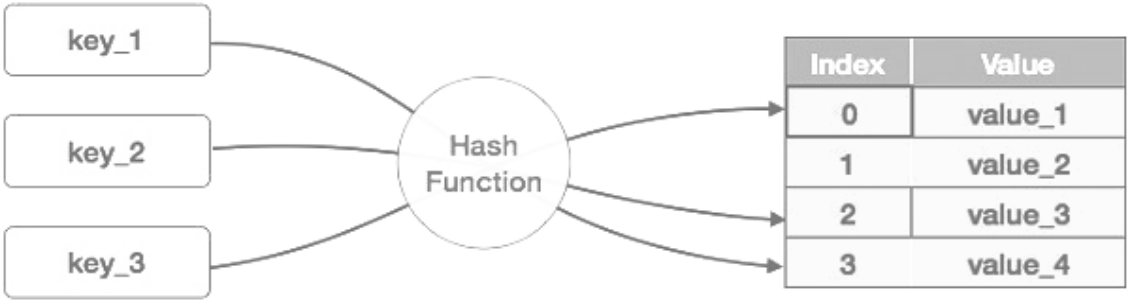
Need For A Good Hash Function

- Let’s try a different hash function.
- The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

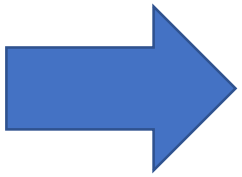
String	Hash function	Index
abcdef	(97*1 + 98*2 + 99*3 + 100*4 + 101*5 + 102*6)%2069	38
	97 + 196 + 297 + 400 + 505 + 612 = 2107 % 2069	
bcdefa	(98*1 + 99*2 + 100*3 + 101*4 + 102*5 + 97*6)%2069	23
cdefab	(99*1 + 100*2 + 101*3 + 102*4 + 97*5 + 98*6)%2069	14
defabc	(100*1 + 101*2 + 102*3 + 97*4 + 98*5 + 99*6)%2069	11

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	

Hashing Function (Example)

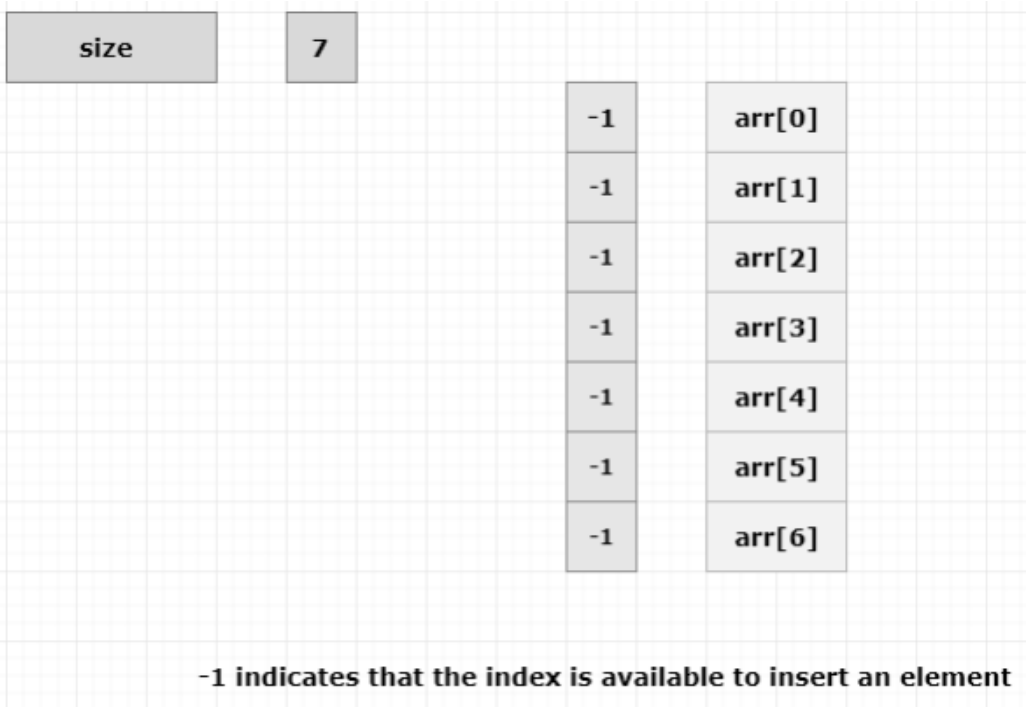


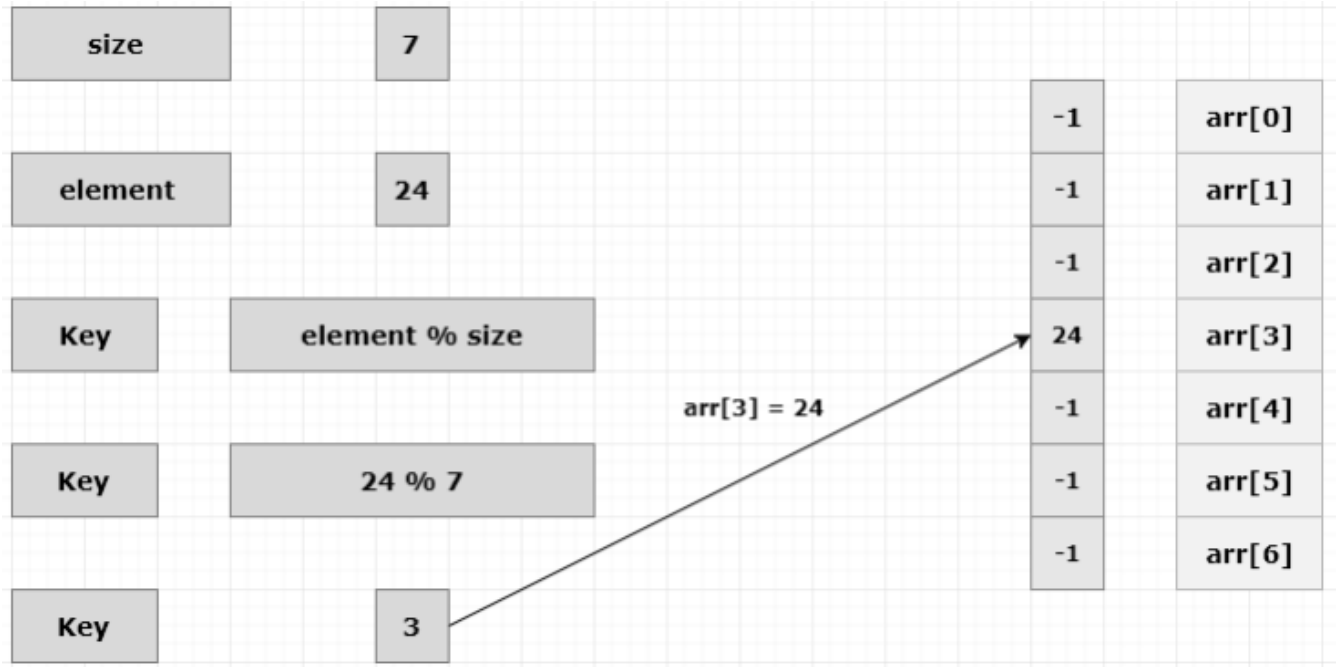
Key & Value
• (1,20) • (2,70) • (42,80) • (4,25) • (12,44) • (14,32) • (17,11) • (13,78) • (37,98)



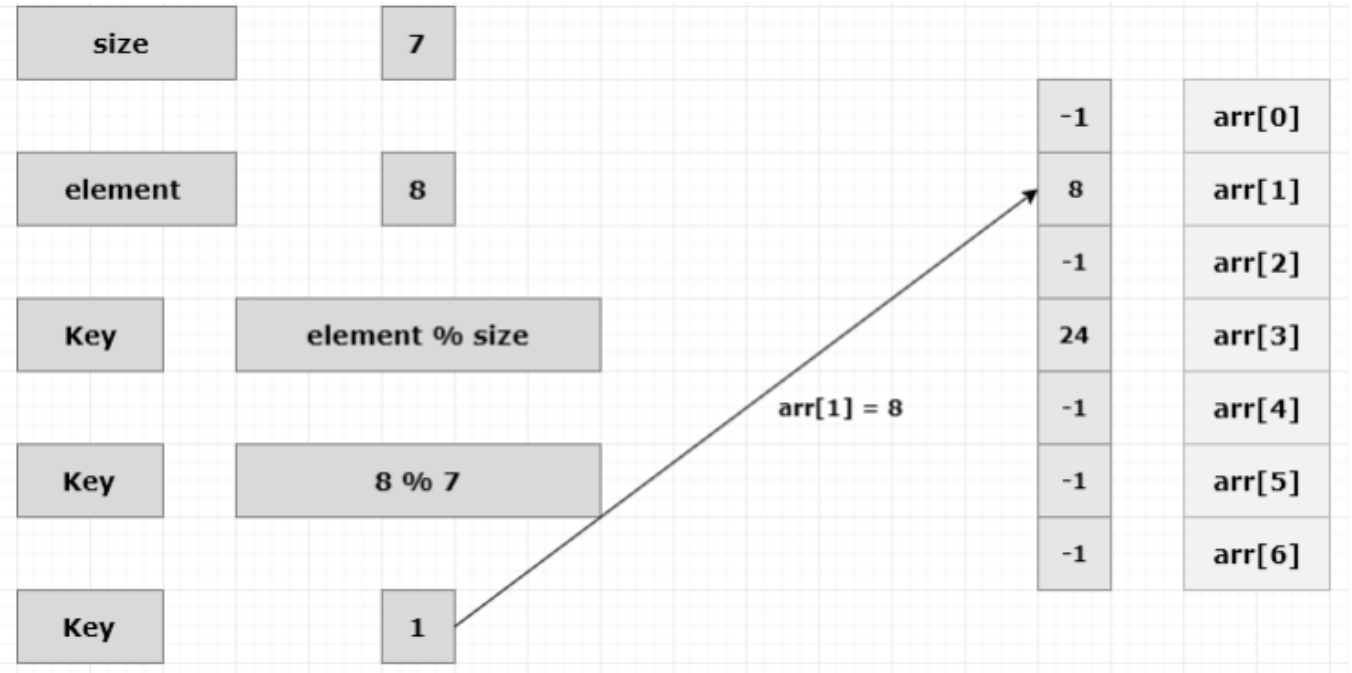
Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

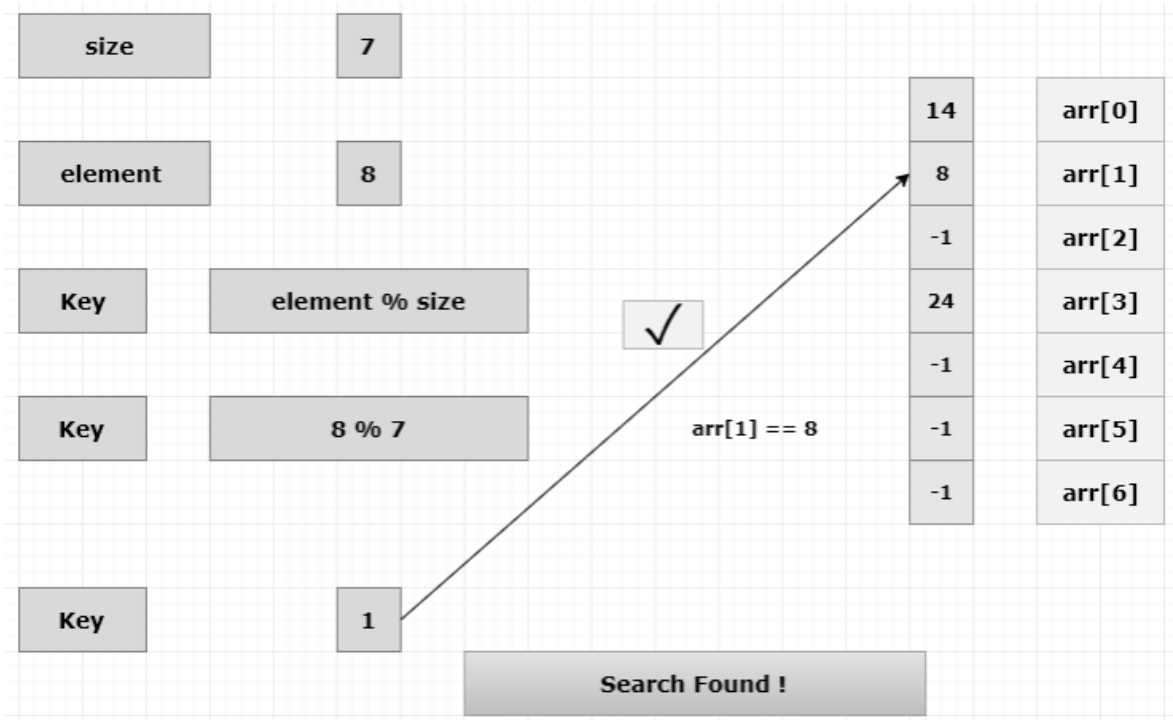
Hashing Function (Example)





Hashing Function (Example)
Searching





Hashing Function (Implementation)

```c
#include<stdio.h>
#define size 7
int arr[size];
void init()
{
    int i;
    for(i = 0; i < size; i++)
        arr[i] = -1;
}
void insert(int value)
{
    int key = value % size;
    if(arr[key] == -1)
    {
        arr[key] = value;
        printf("%d inserted at arr[%d]\n", value,key);
    }
    else
    {
        printf("Collision : arr[%d] has element %d already!\n",key,arr[key]);
        printf("Unable to insert %d\n",value);
    } }
void del(int value)
{
    int key = value % size;
    if(arr[key] == value)
        arr[key] = -1;
    else
        printf("%d not present in the hash table\n",value);
}
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Hashing Function (Implementation)

```
31 void search(int value)
32 {
33     int key = value % size;
34     if(arr[key] == value)
35         printf("Search Found\n");
36     else
37         printf("Search Not Found\n");
38 }
39 void print()
40 {
41     int i;
42     for(i = 0; i < size; i++)
43         printf("arr[%d] = %d\n",i,arr[i]);
44 }
45 int main()
46 {
47     init();
48     insert(10); //key = 10 % 7 ==> 3
49     insert(4);  //key = 4 % 7  ==> 4
50     insert(2);  //key = 2 % 7  ==> 2
51     insert(3);  //key = 3 % 7  ==> 3 (collision)
52     printf("Hash table\n");
53     print();
54     printf("\n");
55     printf("Deleting value 5..\n");
56     del(5);
57     printf("After the deletion hash table\n");
58     print();
59     printf("\n");
60     printf("Searching value 4..\n");
61     search(4);
62     printf("Searching value 10..\n");
63     search(10);
64     return 0;
65 }
```

```
10 inserted at arr[3]
4 inserted at arr[4]
2 inserted at arr[2]
Collision : arr[3] has element 10 already!
Unable to insert 3
Hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = 10
arr[4] = 4
arr[5] = -1
arr[6] = -1
```

Del function: line 23

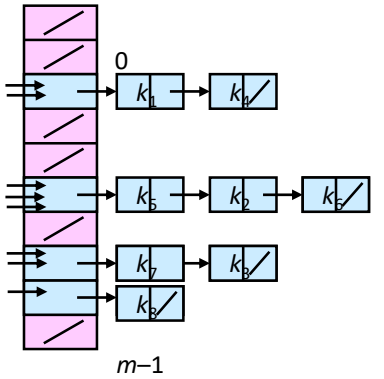
```
Deleting value 5..
5 not present in the hash table
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = 10
arr[4] = 4
arr[5] = -1
arr[6] = -1

Searching value 4..
Search Found
Searching value 10..
Search Found
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

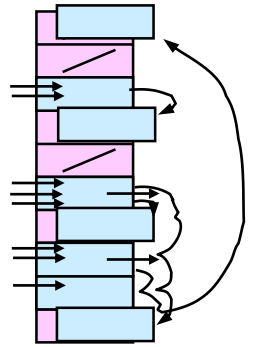
Coalesced Hashing (Avoid Collisions)

- Coalesced hashing is a collision avoidance technique when there is a fixed sized data. It is a combination of both **Separate chaining** and **Open addressing**.
- It uses the concept of Open Addressing(linear probing) to find first empty place for colliding element from the bottom of the hash table and the concept of Separate Chaining to link the colliding elements to each other through pointers.

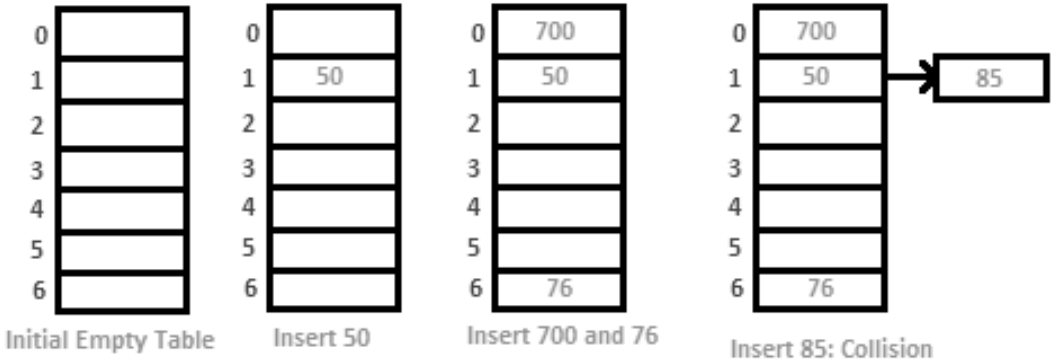
- Chaining (concept):
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.

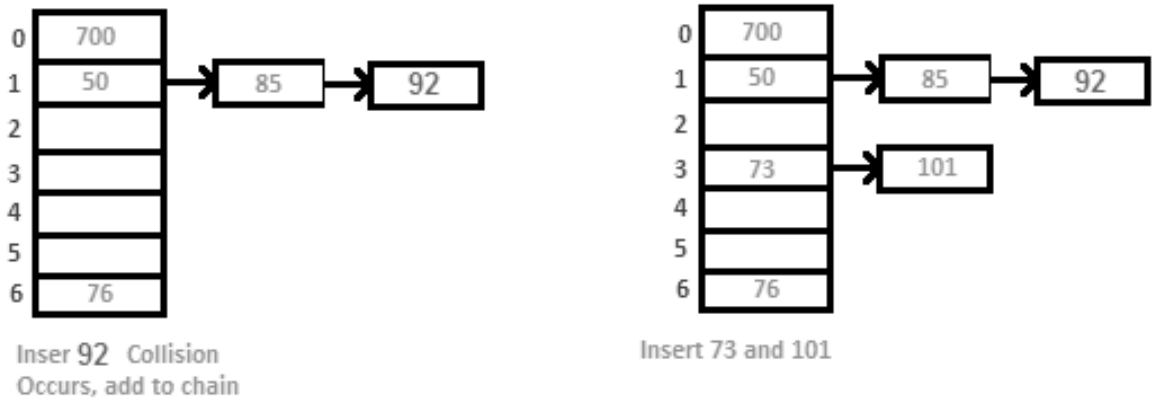


- Open Addressing (concept):
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Avoid Collisions



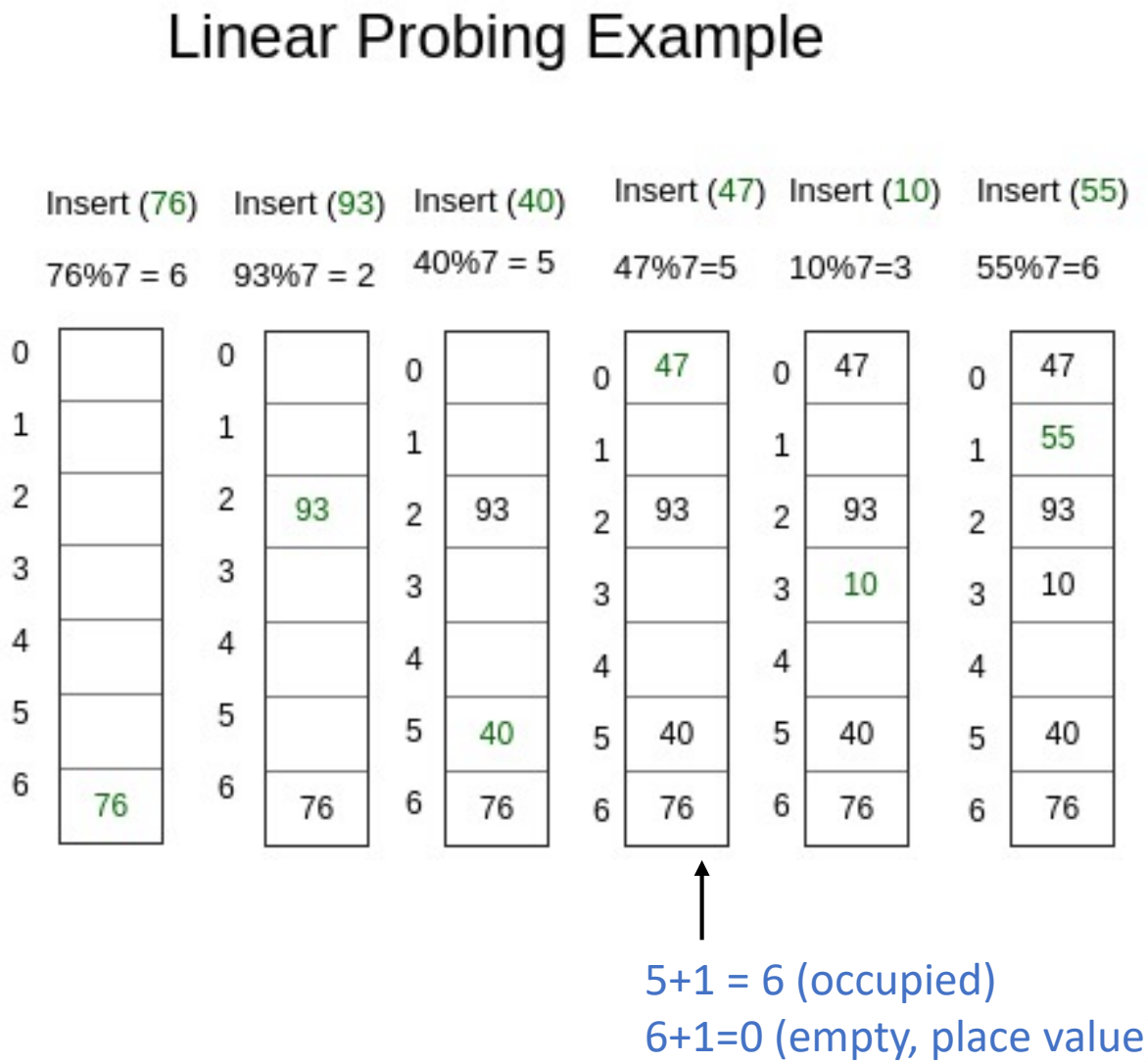


- **Separate chaining** (open hashing)

- Separate chaining is one of the most commonly used collision resolution techniques.
- It is usually implemented using linked lists.
- In separate chaining, each element of the hash table is a linked list.
- To store an element in the hash table you must insert it into a specific linked list.
- If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

Avoid Collisions

- The simplest approach to resolve a collision is linear probing.
- In this technique, if a value is already stored at a location generated by hash(key), it means collision occurred then we do a sequential search to find the empty location.
- Here the idea is to place a value in the next available position.



Avoid Collisions

- **Double hashing** is similar to linear probing and the only difference is the interval between successive probes.
- Here, the interval between probes is computed by using two hash functions.
- Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied.
- You must start traversing in a specific probing sequence to look for an unoccupied slot.
- The probing sequence will be:

  - index = hash_function_1(key) % Table_Size;
  - index = hash_function_2(key) % Table_Size;

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]