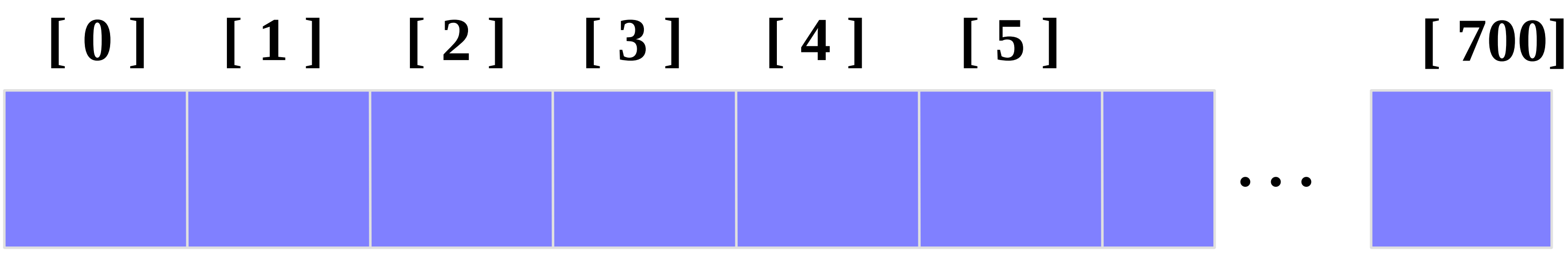
Hash Tables

Introduction

- Hash tables store a collection of records with **keys**.
- The location (index) of a record depends on the **hash value** of the record's key.
- The hash-value (index location) is calculated based on HASH FUNCTIONS

What is a Hash Table ?

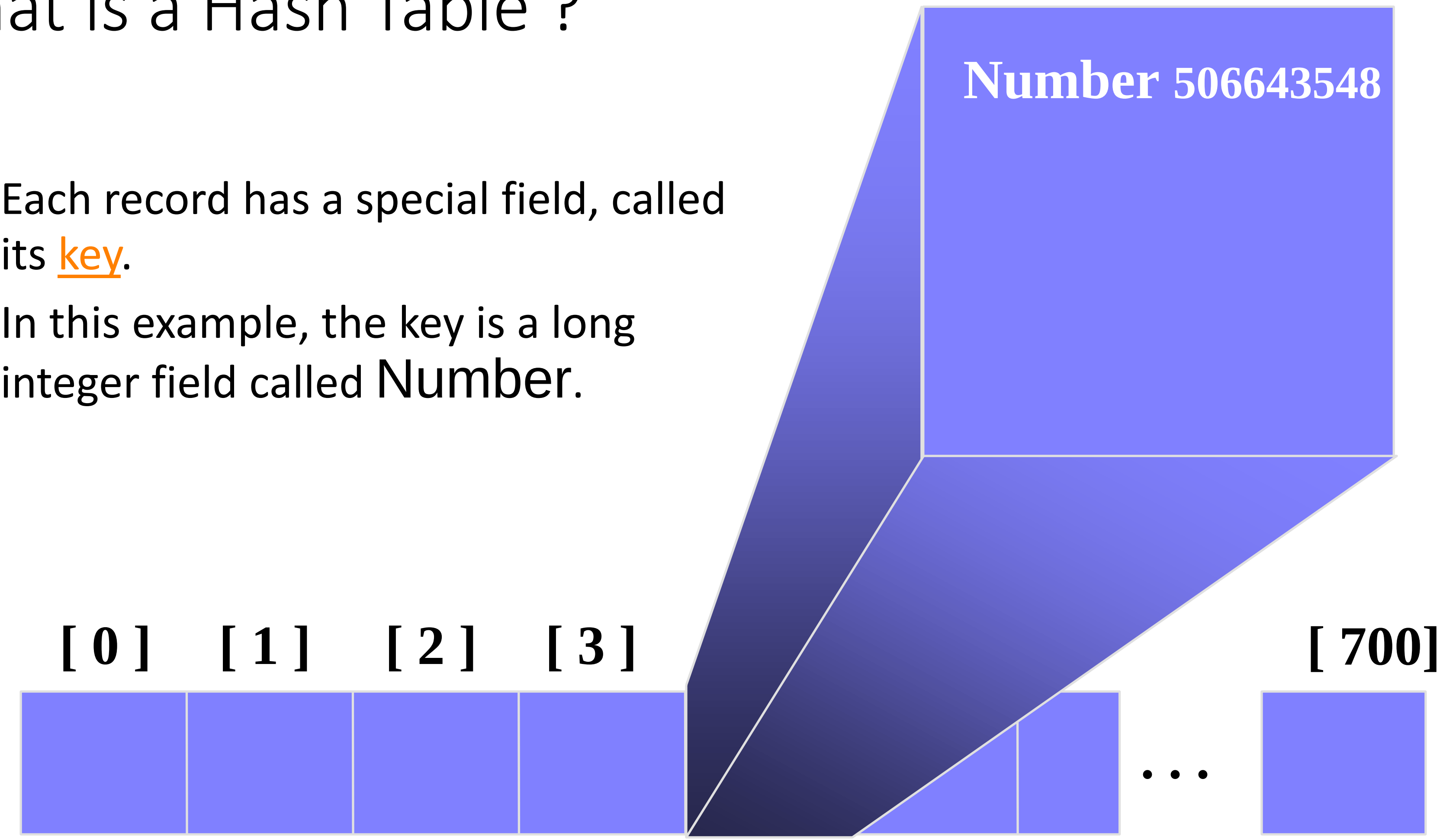
- The simplest kind of hash table is an array of records.
- This example has 701 records.
- Hash function is in our example is:
  - **`MOD size`**



An array of records

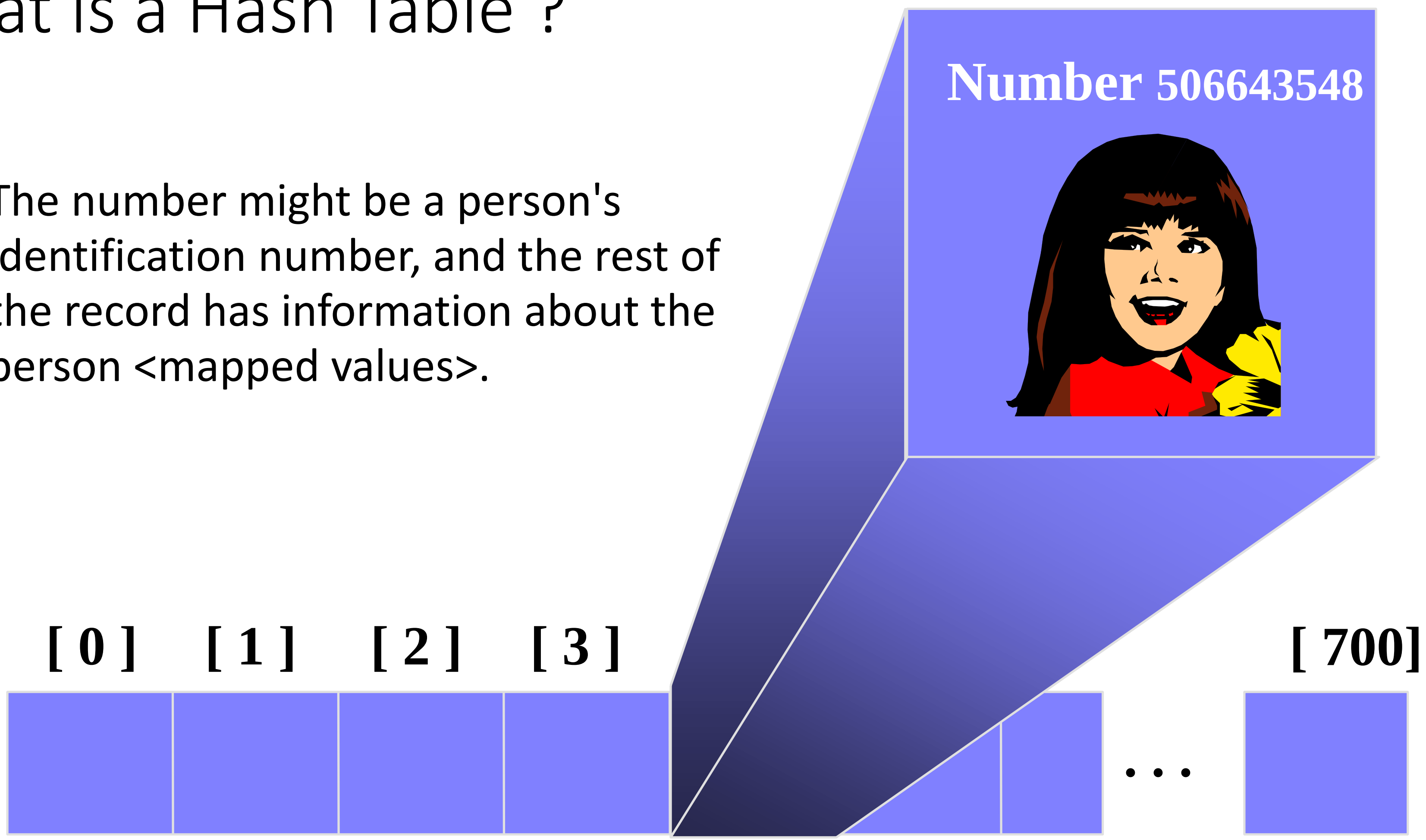
What is a Hash Table ?

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



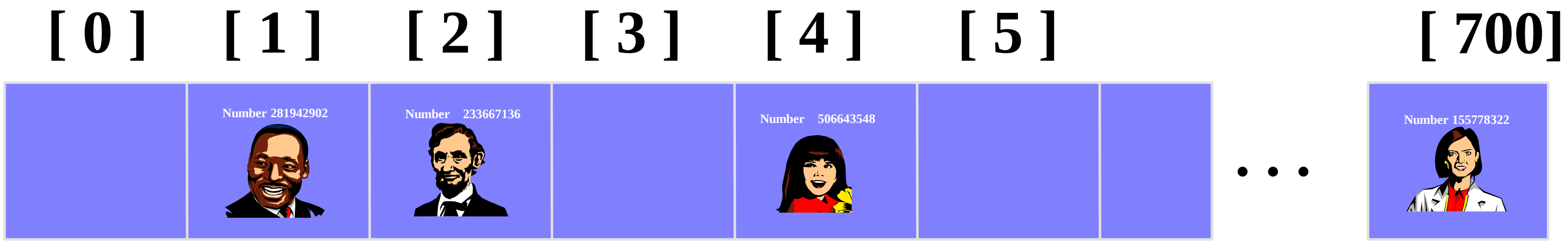
What is a Hash Table ?

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



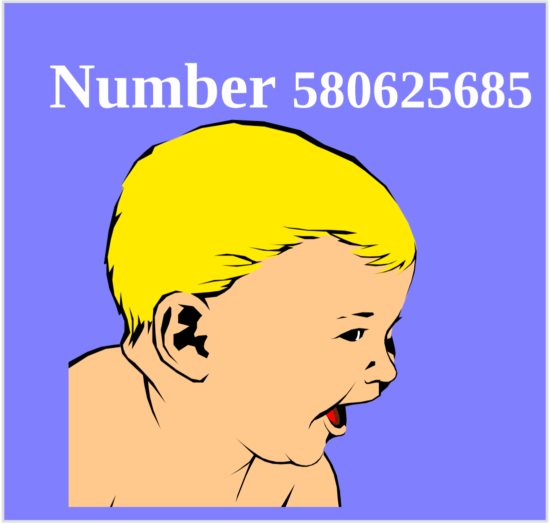
What is a Hash Table ?

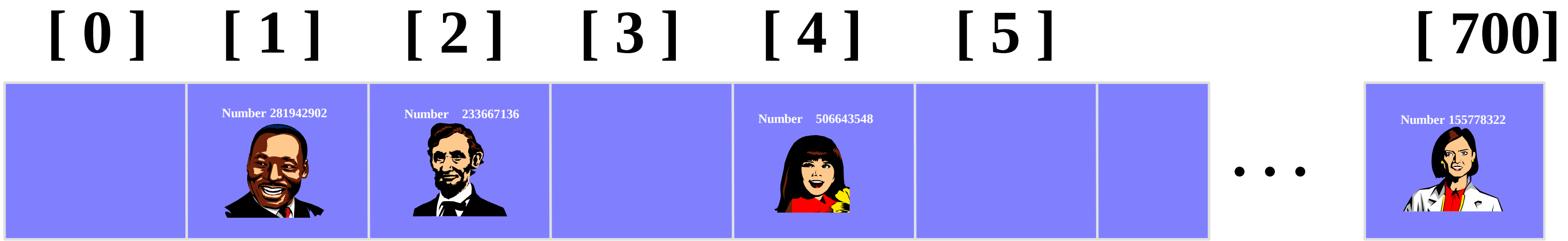
- When a hash table is in use, some spots contain valid records, and other spots are "empty".



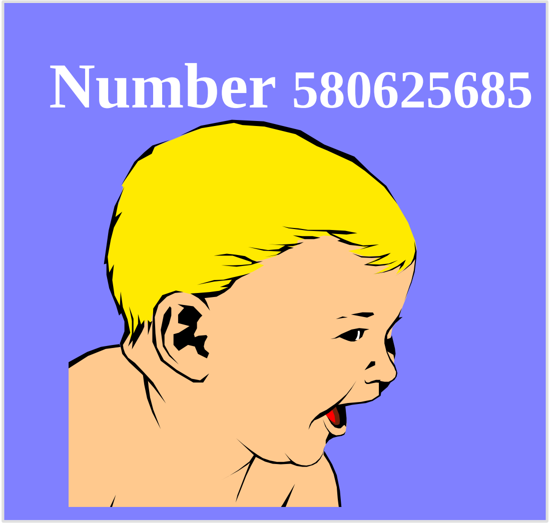
Inserting a New Record

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



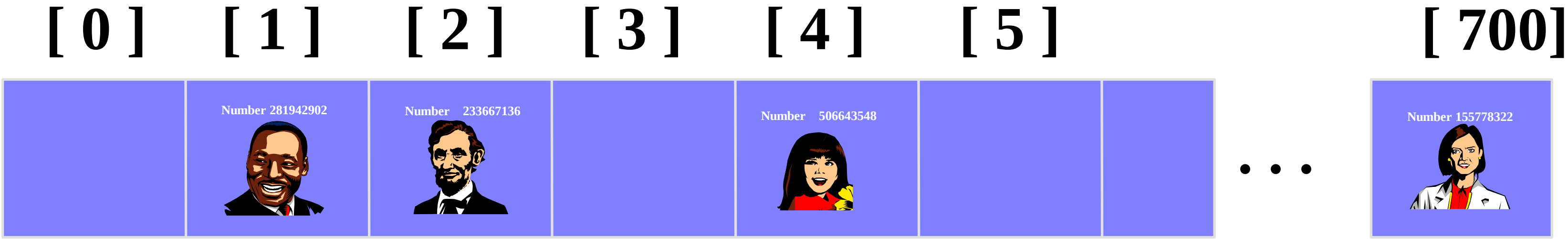


Inserting a New Record



- Typical way to create a hash value:

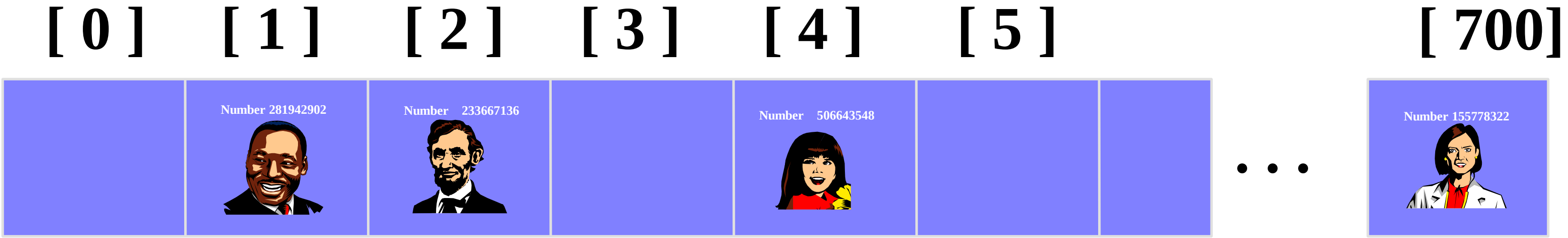
What is (580625685 % 701) ?



Inserting a New Record

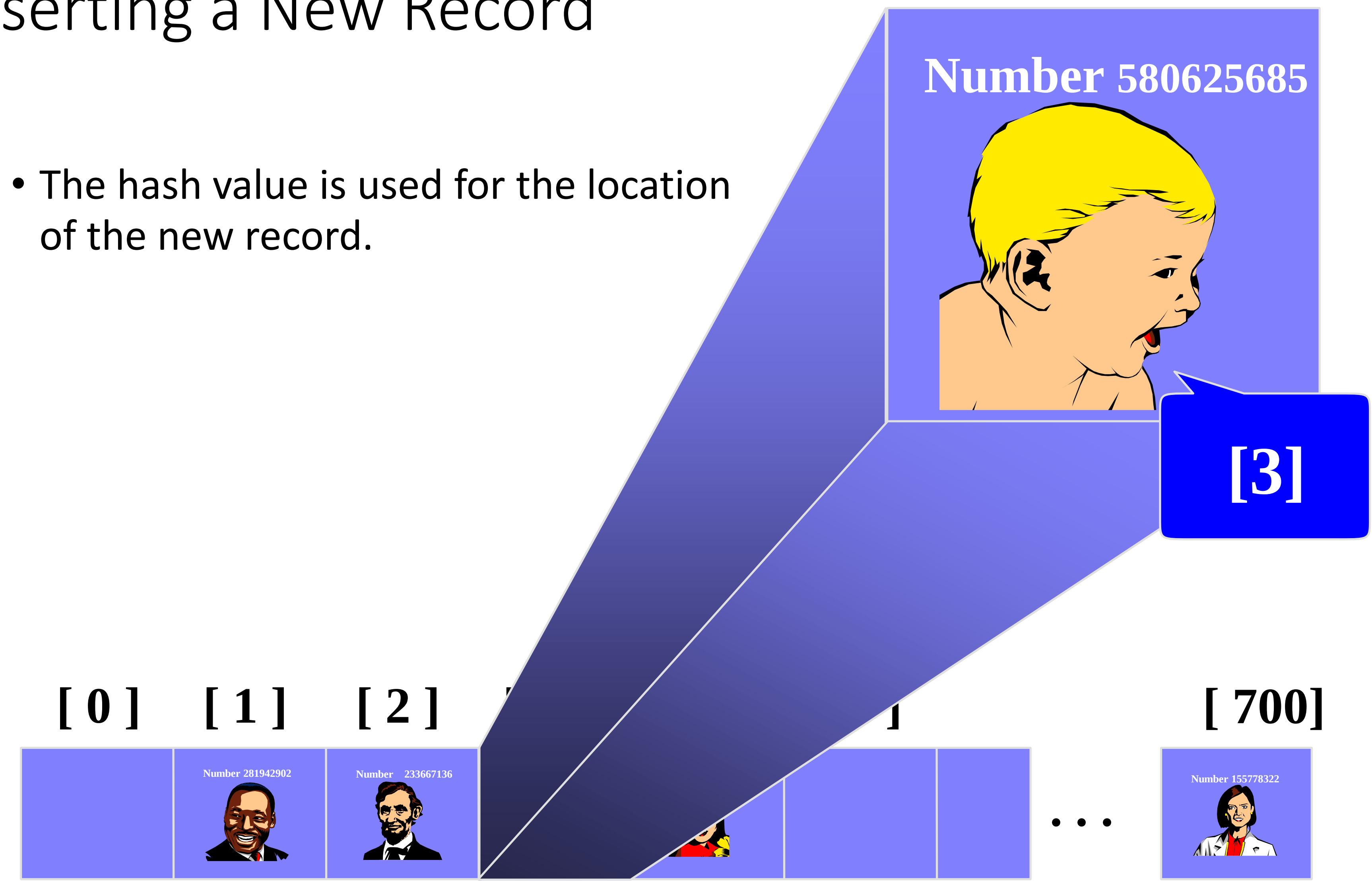
- Typical way to create a hash value:

What is (580625685 % 701) ?



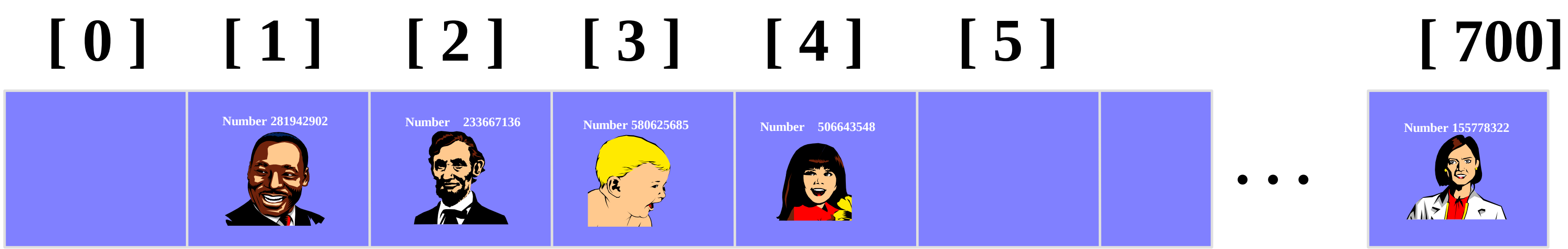
Inserting a New Record

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



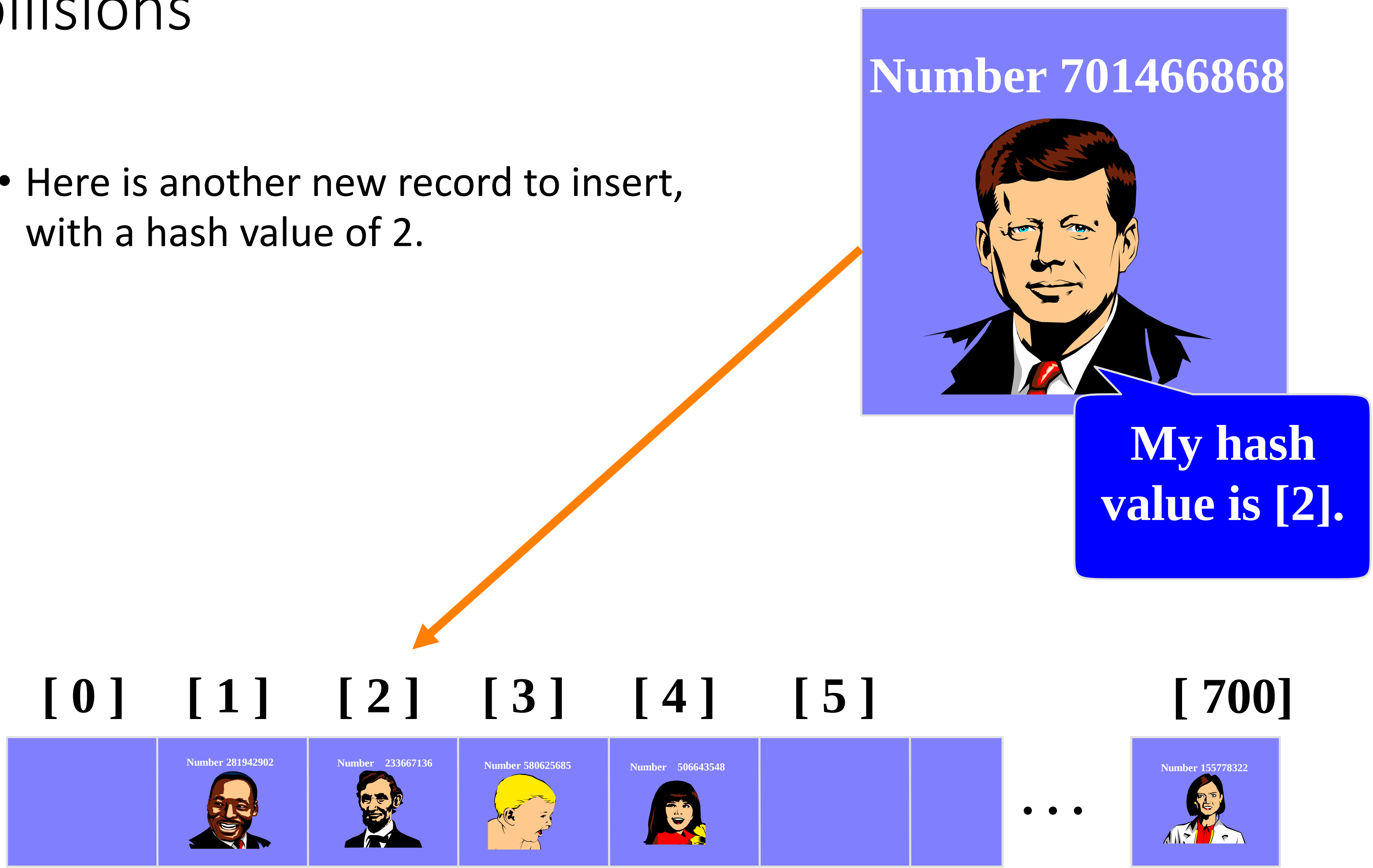
Inserting a New Record

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



Collisions

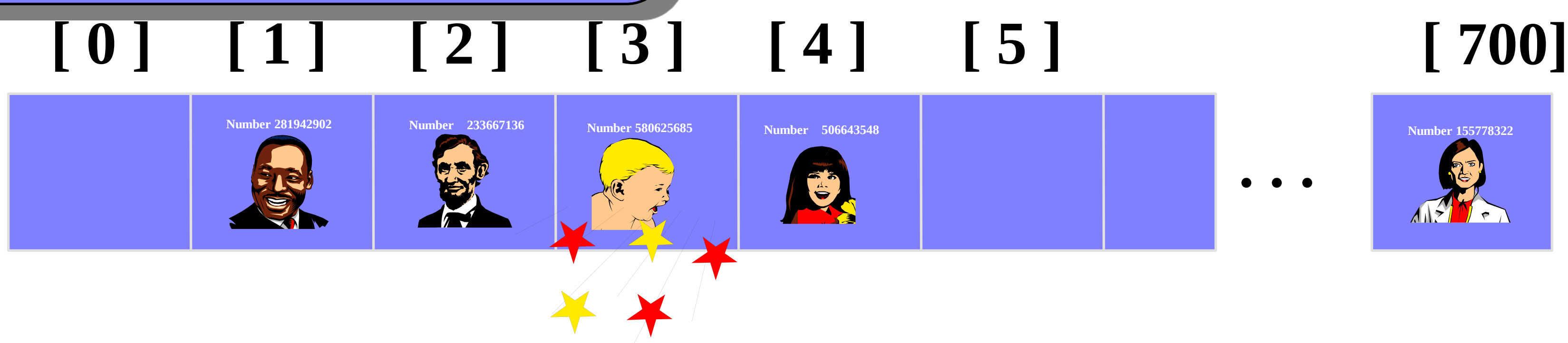
- Here is another new record to insert, with a hash value of 2.



Collisions

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

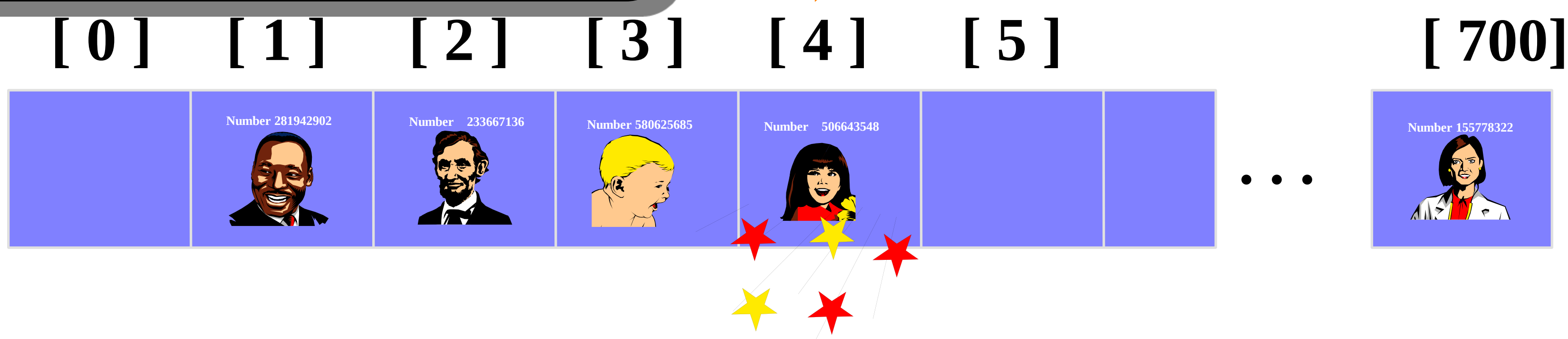
When a collision occurs,
move forward until you find an empty spot.



Collisions

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

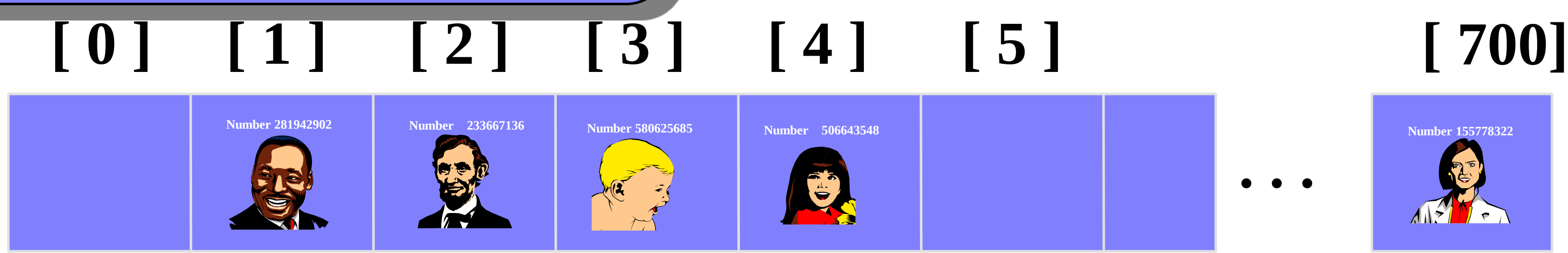
When a collision occurs,
move forward until you find an empty spot.



Collisions

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

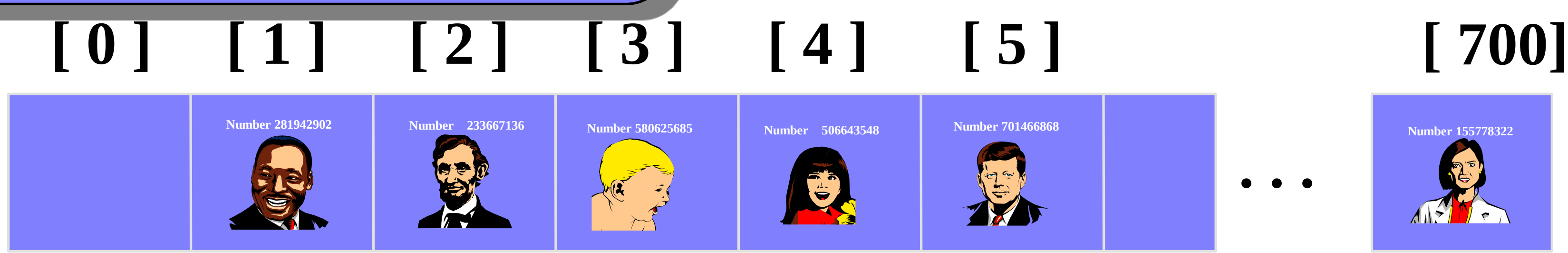
When a collision occurs,
move forward until you find an empty spot.



Collisions

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

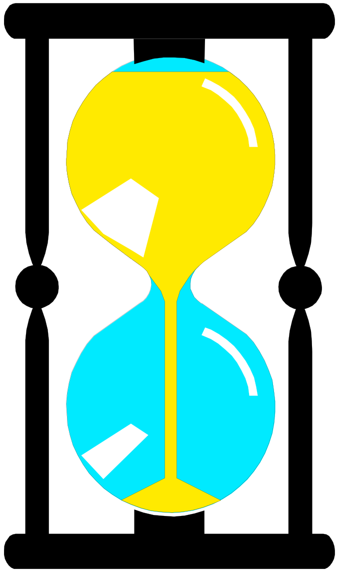
The new record goes in the empty spot.

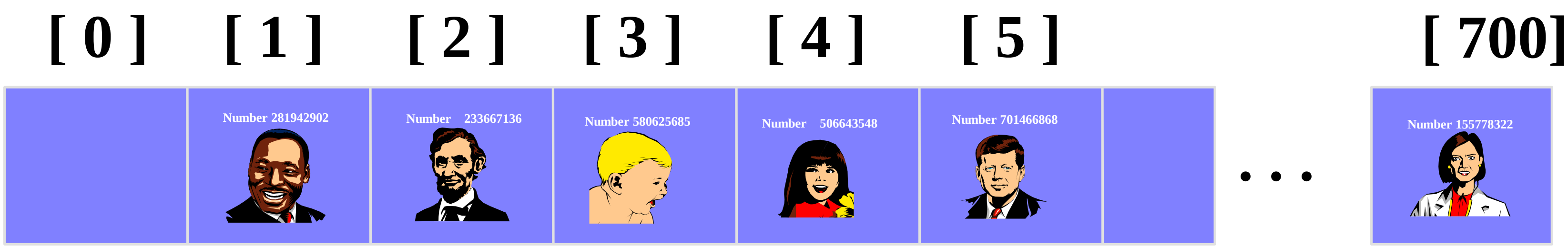


A small Quiz

At what index would you be placed in this table, if your NUMBER is 281942201

all slots from 6 to 699 are empty





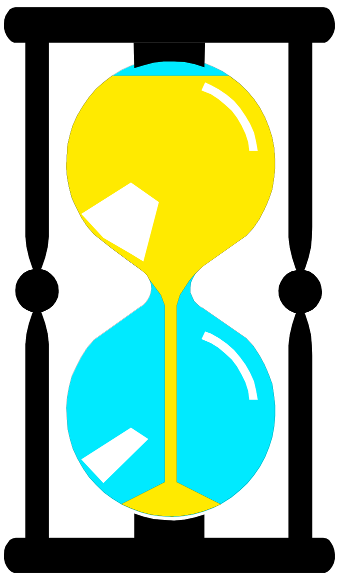
A small Quiz

At what index would you be placed in this table, if your NUMBER is 281942201

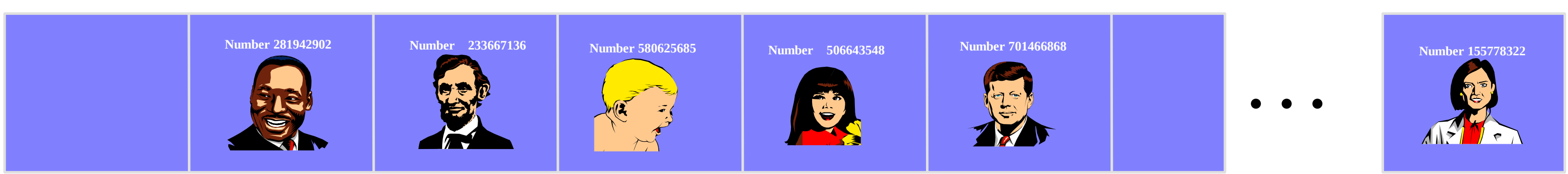
all slots from 6 to 699 are empty

ANSWER = [6]

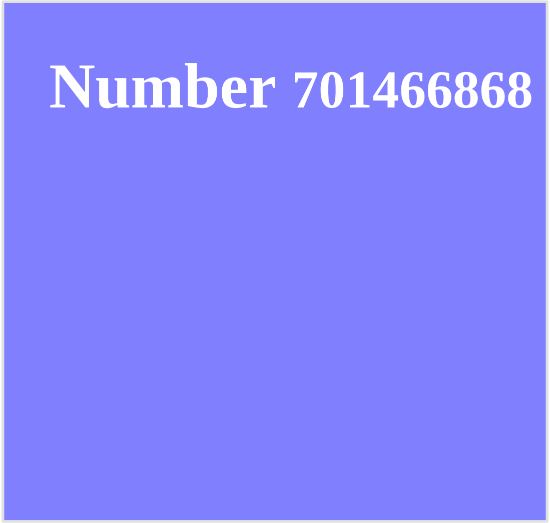
Explanation: 281942201 % 701 is [1], but due to collision, next available space is [6]



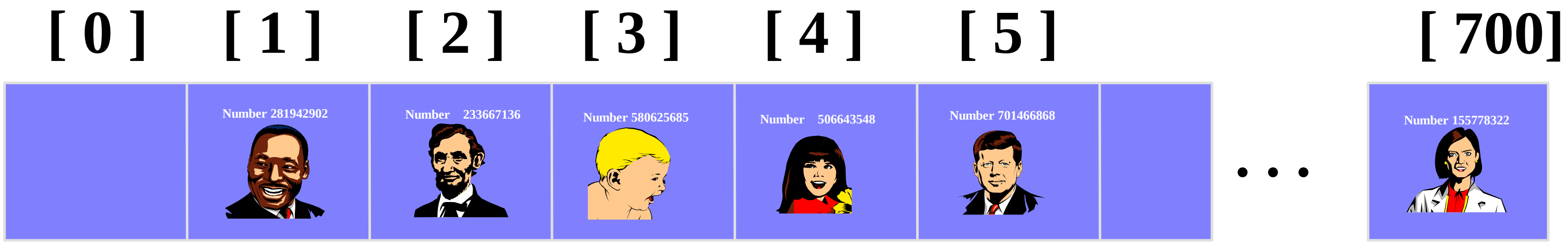
[0] [1] [2] [3] [4] [5] [700]



Searching for a Key

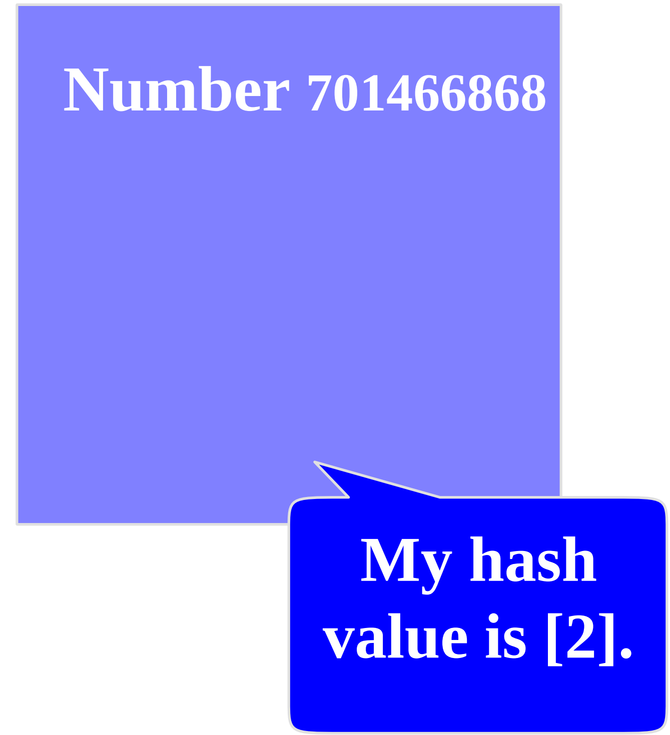


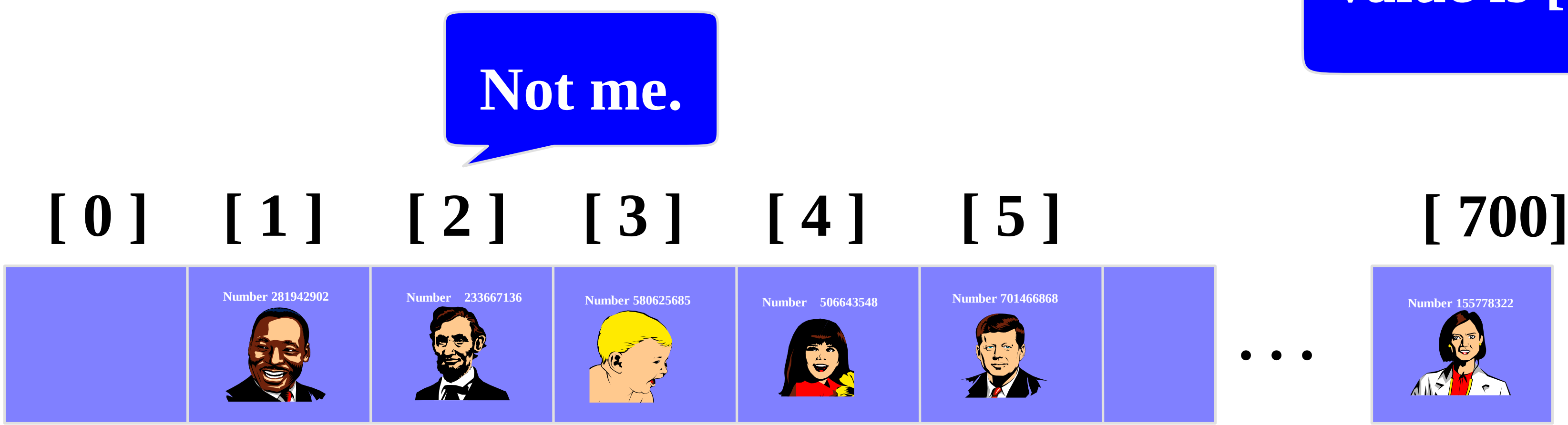
- The data that's attached to a key can be found fairly quickly.



Searching for a Key

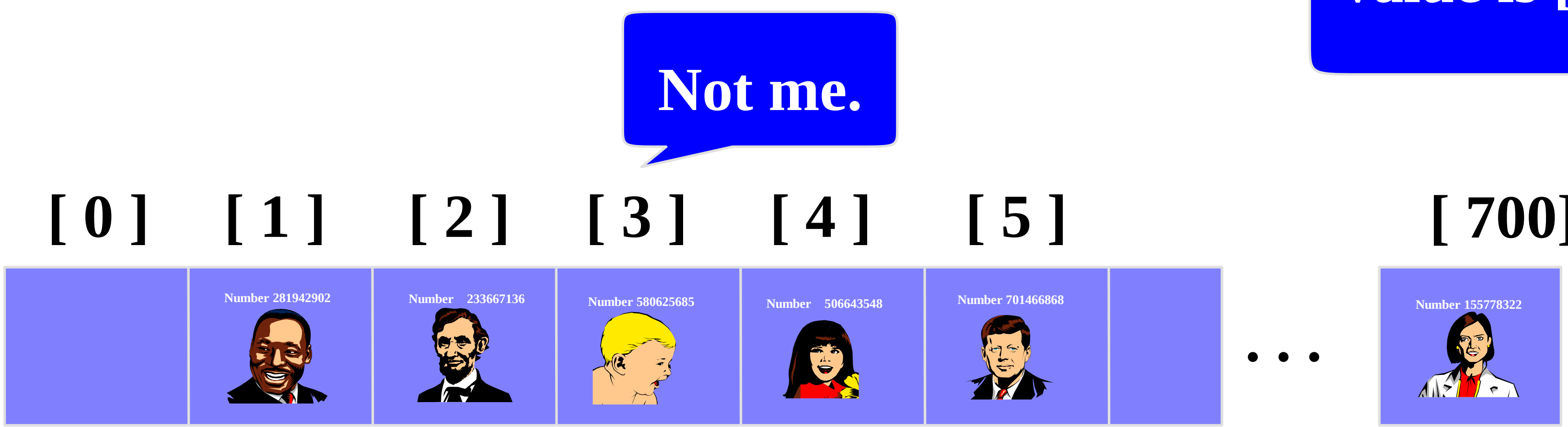
- Calculate the hash value.
- Check that location of the array for the key.





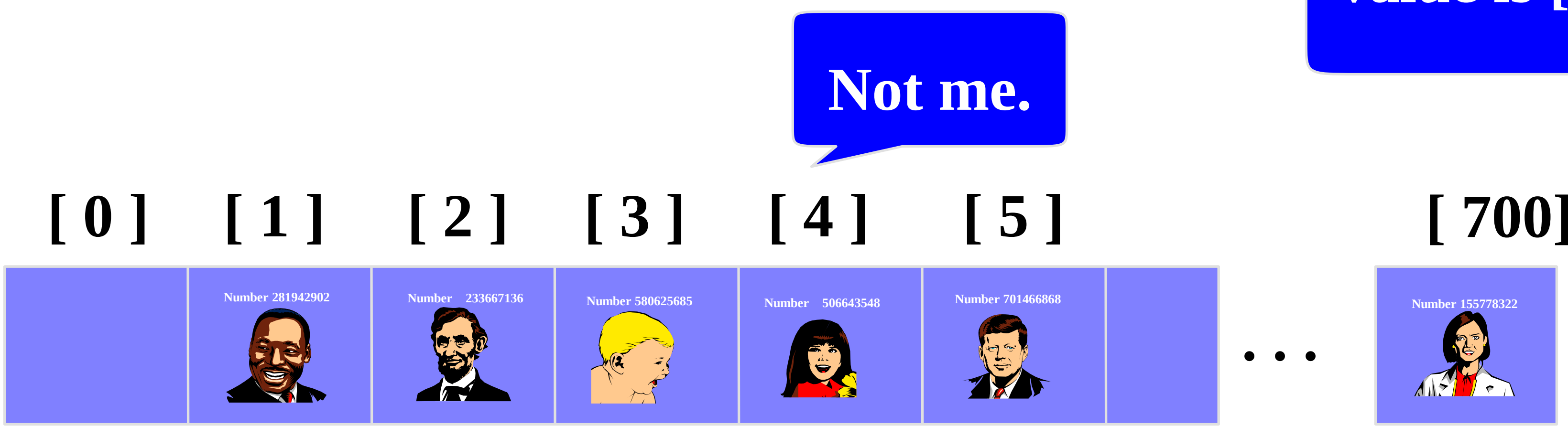
Searching for a Key

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



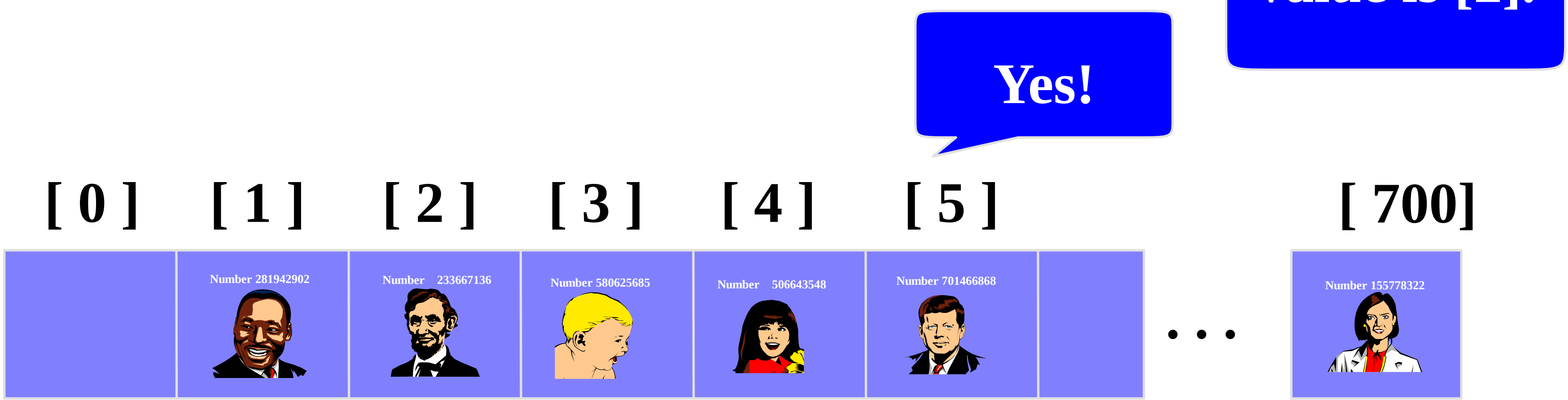
Searching for a Key

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



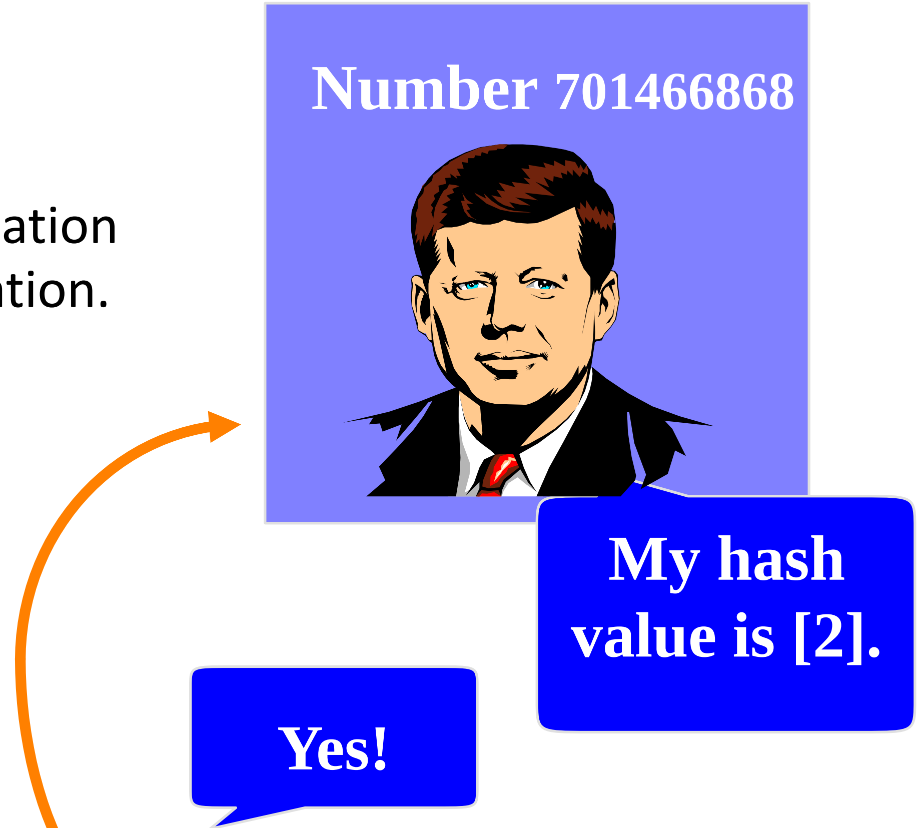
Searching for a Key

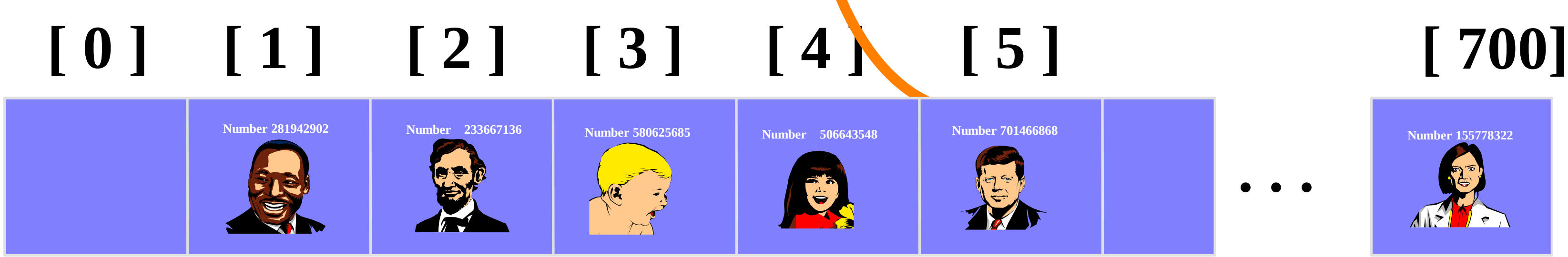
- Keep moving forward until you find the key, or you reach an empty spot.



Searching for a Key

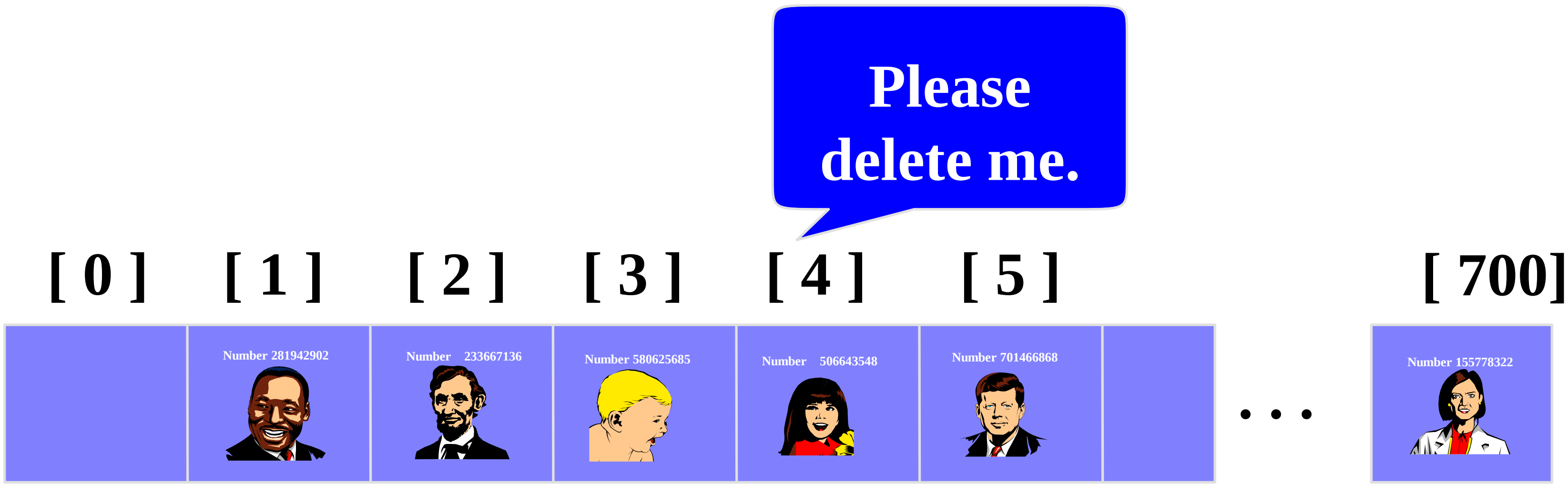
- When the item is found, the information can be copied to the necessary location.





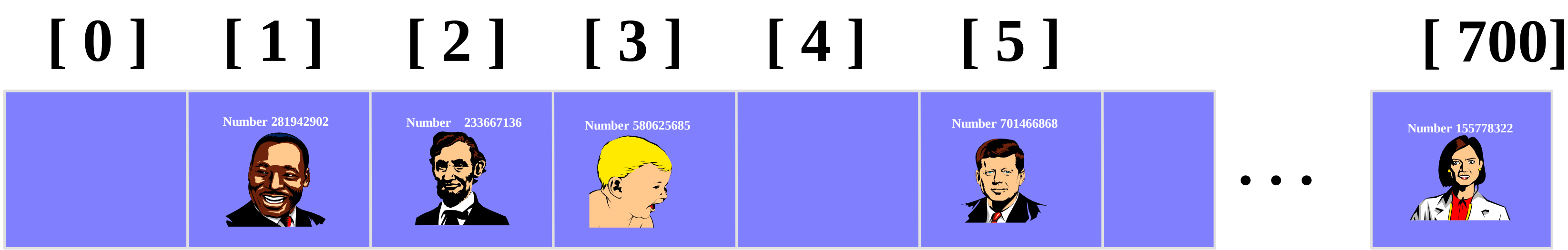
Deleting a Record

- Records may also be deleted from a hash table.



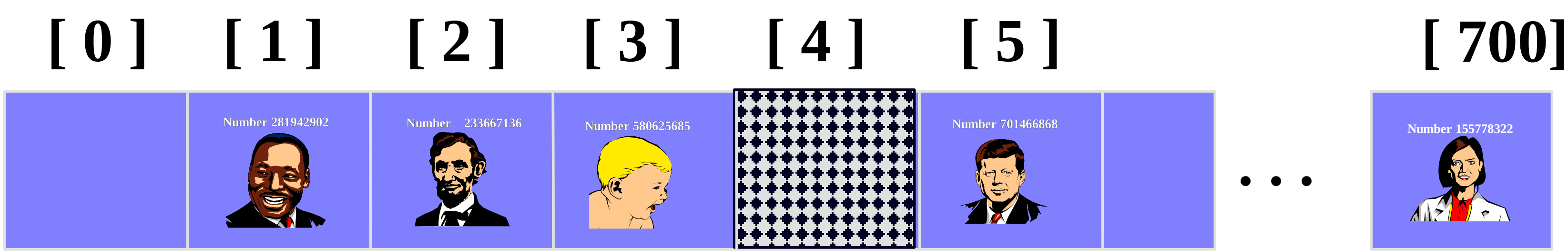
Deleting a Record

- 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.



Hashing: Collision Resolution Schemes

- Collision Resolution Techniques

- Separate Chaining
  - Separate Chaining with String Keys
  - The class hierarchy of Hash Tables
  - Implementation of Separate Chaining

- Introduction to Collision Resolution using Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
  - Rehashing
- Algorithms for insertion, searching, and deletion in Open Addressing
- Separate Chaining versus Open-addressing

Collision Resolution Techniques

- There are two broad ways of collision resolution:

1. Separate Chaining: An array of linked list implementation.

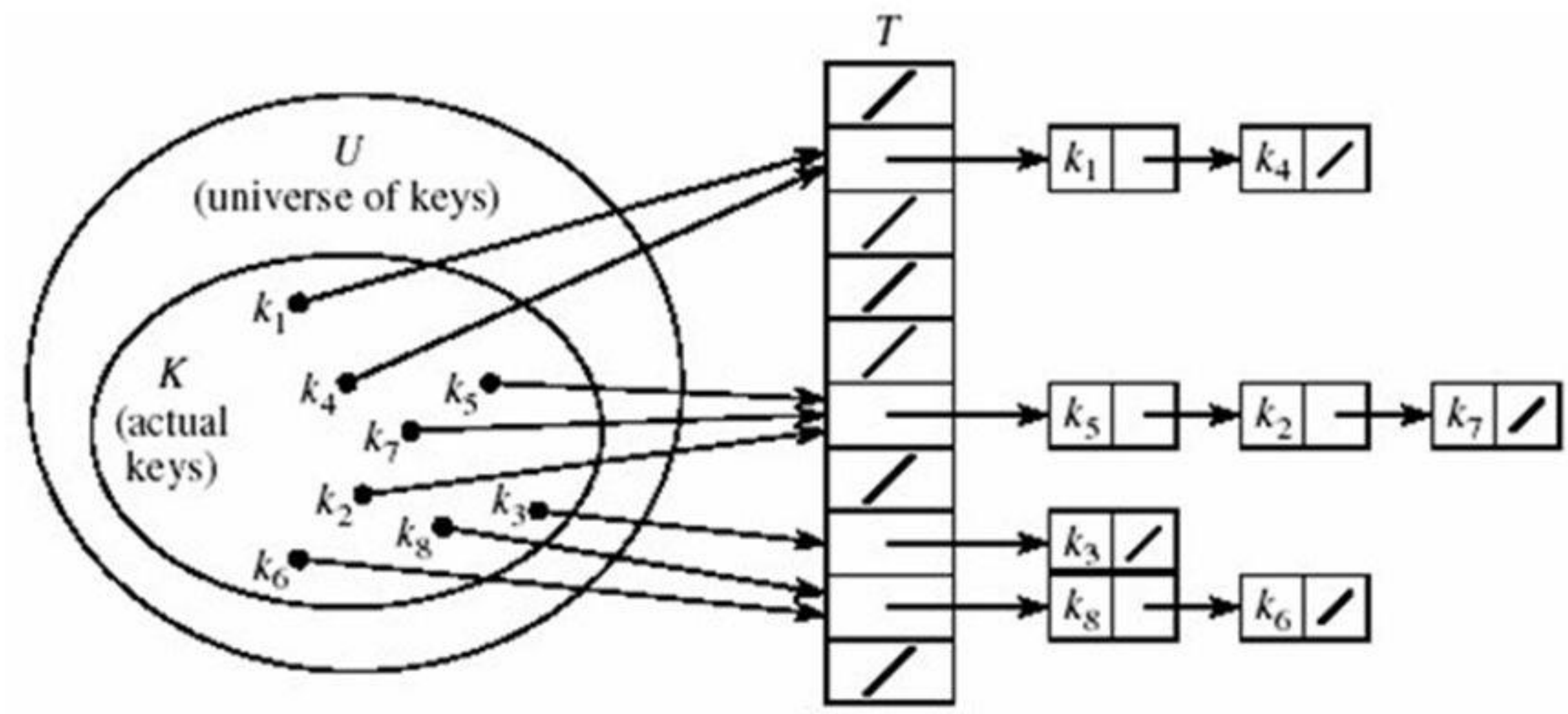
2. Open Addressing: Array-based implementation.

    (i) Linear probing (linear search)
    (ii) Quadratic probing (nonlinear search)
    (iii) Double hashing (uses two hash functions)

Separate Chaining

- The hash table is implemented as an array of linked lists.

- Inserting an item, `r`, that hashes at index `i` is simply insertion into the linked list at position `i`.
- Synonyms are chained in the same linked list.



Separate Chaining (cont'd)

- Retrieval of an item, `r`, with hash address, `i`, is simply retrieval from the linked list at position `i`.
- Deletion of an item, `r`, with hash address, `i`, is simply deleting `r` from the linked list at position `i`.
- **Example:** Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

h(23) = 23 % 7 = 2

h(13) = 13 % 7 = 6

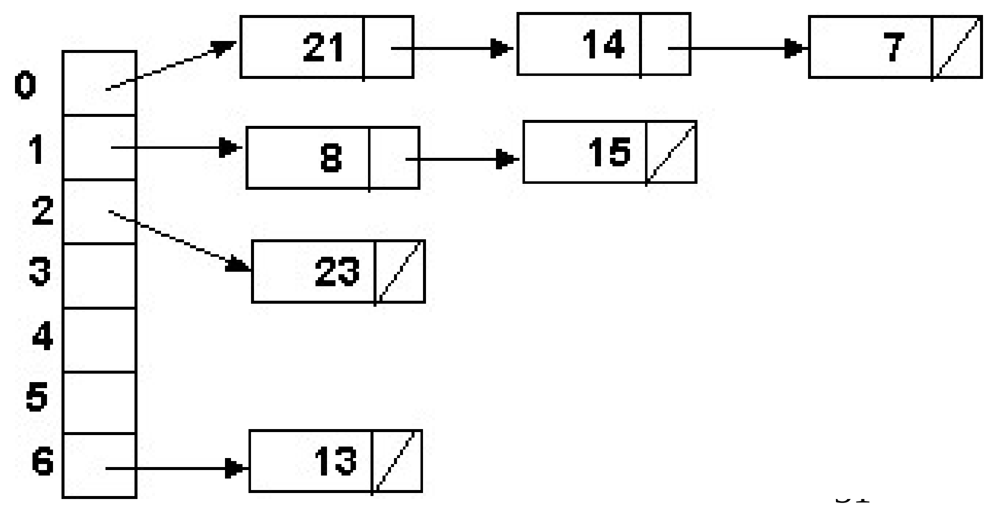
h(21) = 21 % 7 = 0

h(14) = 14 % 7 = 0 collision

h(7) = 7 % 7 = 0 collision

h(8) = 8 % 7 = 1

h(15) = 15 % 7 = 1 collision



Introduction to Open Addressing

- All items are stored in the hash table itself.
- In addition to the cell data (if any), each cell keeps one of the three states: EMPTY, OCCUPIED, DELETED.
- While inserting, if a collision occurs, alternative cells are tried until an empty cell is found.

- **Deletion**: (lazy deletion): When a key is deleted the slot is marked as DELETED rather than EMPTY otherwise subsequent searches that hash at the deleted cell will fail.

- **Probe sequence**: A probe sequence is the sequence of array indexes that is followed in searching for an empty cell during an insertion, or in searching for a key during find or delete operations.

- The most common probe sequences are of the form:

$$h_i(key) = [h(key) + c(i)] \% \ n, \quad \text{for } i = 0, 1, \ldots, n-1.$$

where **h** is a hash function and **n** is the size of the hash table

- The function **c(i)** is required to have the following two properties:

Property 1: $c(0) = 0$

Property 2: The set of values **{c(0) % n, c(1) % n, c(2) % n, . . . , c(n-1) % n}** must be a permutation of **{0, 1, 2,. . ., n – 1}**, that is, it must contain every integer between **0** and **n - 1** inclusive.

Introduction to Open Addressing (cont'd)

- The function **c(i)** is used to resolve collisions.

- To insert item r, we examine array location **$h_0(r) = h(r)$**. If there is a collision, array locations **$h_1(r), h_2(r), ..., h_{n-1}(r)$** are examined until an empty slot is found.

- Similarly, to find item **r**, we examine the same sequence of locations in the same order.

- **Note**: For a given hash function **h(key)**, the only difference in the open addressing collision resolution techniques (linear probing, quadratic probing and double hashing) is in the definition of the function **c(i)**.
- Common definitions of **c(i)** are:

Collision resolution technique	c(i)
Linear probing	**i**
Quadratic probing	**$\pm i^2$**
Double hashing	**$i * h_p(key)$**

where **$h_p(key)$** is another hash function.

Introduction to Open Addressing (cont'd)

- **Advantages of Open addressing:**
  - All items are stored in the hash table itself. There is no need for another data structure.
  - Open addressing is more efficient storage-wise.

- **Disadvantages of Open Addressing:**
  - The keys of the objects to be hashed must be distinct.
  - Dependent on choosing a proper table size.
  - Requires the use of a three-state (Occupied, Empty, or Deleted) flag in each cell.

Open Addressing Facts

- In general, primes give the best table sizes.
- With any open addressing method of collision resolution, as the table fills, there can be a severe degradation in the table performance.
- Load factors between 0.6 and 0.7 are common.
- Load factors > 0.7 are undesirable.
- The search time depends only on the load factor, *not* on the table size.
- We can use the desired load factor to determine appropriate table size:

$$\text{table size} \quad = \quad \text{smallest prime} \geq \frac{\text{number of items in table}}{\text{desired load factor}}$$

Open Addressing: Linear Probing

- **c(i)** is a linear function in **i** of the form **c(i) = a*i**.
- Usually **c(i)** is chosen as:

  c(i) = i **for i = 0, 1, . . . , tableSize – 1**

- The probe sequences are then given by:

  h_i(key) = [h(key) + i] % tableSize for i = 0, 1, . . . , tableSize – 1

- For **c(i) = a*i** to satisfy Property 2, **a** and **n** must be relatively prime.

Linear Probing (cont'd)

Example: Perform the operations given below, in the given order, on an initially empty hash table of size **13** using linear probing with **c(i) = i** and the hash function: **h(key) = key % 13**:

insert(18), insert(26), insert(35), insert(9), find(15), find(48), delete(35), delete(40), find(9), insert(64), insert(47), find(35)

- The required probe sequences are given by:

$$h_i(key) = (h(key) + i) \% 13 \qquad i = 0, 1, 2, \ldots, 12$$

Linear Probing (cont'd)

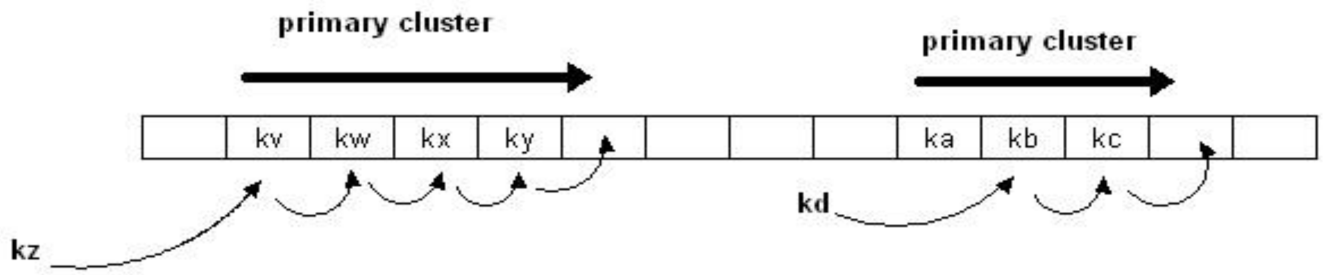
a

OPERATION	PROBE SEQUENCE	COMMENT
insert(18)	$h_0(18) = (18 \% 13) \% 13 = 5$	SUCCESS
insert(26)	$h_0(26) = (26 \% 13) \% 13 = 0$	SUCCESS
insert(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	SUCCESS
insert(9)	$h_0(9) = (9 \% 13) \% 13 = 9$	COLLISION
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
find(15)	$h_0(15) = (15 \% 13) \% 13 = 2$	FAIL because location 2 has **Empty** status
find(48)	$h_0(48) = (48 \% 13) \% 13 = 9$	COLLISION
	$h_1(48) = (9 + 1) \% 13 = 10$	COLLISION
	$h_2(48) = (9 + 2) \% 13 = 11$	FAIL because location 11 has **Empty** status
withdraw(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	SUCCESS because location 9 contains 35 and the status is **Occupied** The status is changed to **Deleted**; but the key 35 is not removed.
find(9)	$h_0(9) = (9 \% 13) \% 13 = 9$	The search continues, location 9 does not contain 9; but its status is **Deleted**
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
insert(64)	$h_0(64) = (64 \% 13) \% 13 = 12$	SUCCESS
insert(47)	$h_0(47) = (47 \% 13) \% 13 = 8$	SUCCESS
find(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	FAIL because location 9 contains 35 but its status is **Deleted**

Index	**Status**	**Value**
0	O	26
1	E	
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	D	35
10	O	9
11	E	
12	O	64

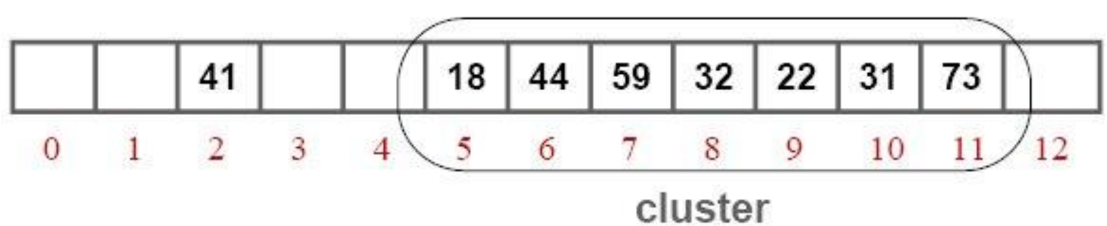
Disadvantage of Linear Probing: Primary Clustering

- Linear probing is subject to a primary clustering phenomenon.
- Elements tend to cluster around table locations that they originally hash to.
- Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency.



Example of a primary cluster: Insert keys: **18, 41, 22, 44, 59, 32, 31, 73**, in this order, in an originally empty hash table of size **13**, using the hash function **h(key) = key % 13** and **c(i) = i**:

h(18) = 5
h(41) = 2
h(22) = 9
h(44) = 5+1
h(59) = 7
h(32) = 6+1+1
h(31) = 5+1+1+1+1+1
h(73) = 8+1+1+1



Open Addressing: Quadratic Probing

- Quadratic probing eliminates primary clusters.
- **c(i)** is a quadratic function in **i** of the form **$c(i) = a*i^2 + b*i$**. Usually **c(i)** is chosen as:

  $c(i) = i^2$ **for $i = 0, 1, \ldots, tableSize - 1$**

  or

  $c(i) = \pm i^2$ **for $i = 0, 1, \ldots, (tableSize - 1) / 2$**

- The probe sequences are then given by:

  $h_i(key) = [h(key) + i^2]$ % tableSize **for $i = 0, 1, \ldots, tableSize - 1$**

  or

  $h_i(key) = [h(key) \pm i^2]$ % tableSize **for $i = 0, 1, \ldots, (tableSize - 1) / 2$**

- Note for Quadratic Probing:
  - Hashtable size should not be an even number; otherwise Property 2 will not be satisfied.
  - Ideally, table size should be a prime of the form 4j+3, where j is an integer. This choice of table size guarantees Property 2.

Quadratic Probing (cont'd)

- Example: Load the keys **23, 13, 21, 14, 7, 8, and 15**, in this order, in a hash table of size **7** using quadratic probing with **$c(i) = \pm i^2$** and the hash function: **h(key) = key % 7**
- The required probe sequences are given by:

$$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

Quadratic Probing (cont'd)

$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

$h_0(23) = (23 \% 7) \% 7 = 2$

$h_0(13) = (13 \% 7) \% 7 = 6$

$h_0(21) = (21 \% 7) \% 7 = 0$

$h_0(14) = (14 \% 7) \% 7 = 0$ collision

$h_1(14) = (0 + 1^2) \% 7 = 1$

$h_0(7) = (7 \% 7) \% 7 = 0$ collision

$h_1(7) = (0 + 1^2) \% 7 = 1$ collision

$h_{-1}(7) = (0 - 1^2) \% 7 = -1$

NORMALIZE: $(-1 + 7) \% 7 = 6$ collision

$h_2(7) = (0 + 2^2) \% 7 = 4$

$h_0(8) = (8 \% 7)\%7 = 1$ collision

$h_1(8) = (1 + 1^2) \% 7 = 2$ collision

$h_{-1}(8) = (1 - 1^2) \% 7 = 0$ collision

$h_2(8) = (1 + 2^2) \% 7 = 5$

$h_0(15) = (15 \% 7)\%7 = 1$ collision

$h_1(15) = (1 + 1^2) \% 7 = 2$ collision

$h_{-1}(15) = (1 - 1^2) \% 7 = 0$ collision

$h_2(15) = (1 + 2^2) \% 7 = 5$ collision

$h_{-2}(15) = (1 - 2^2) \% 7 = -3$

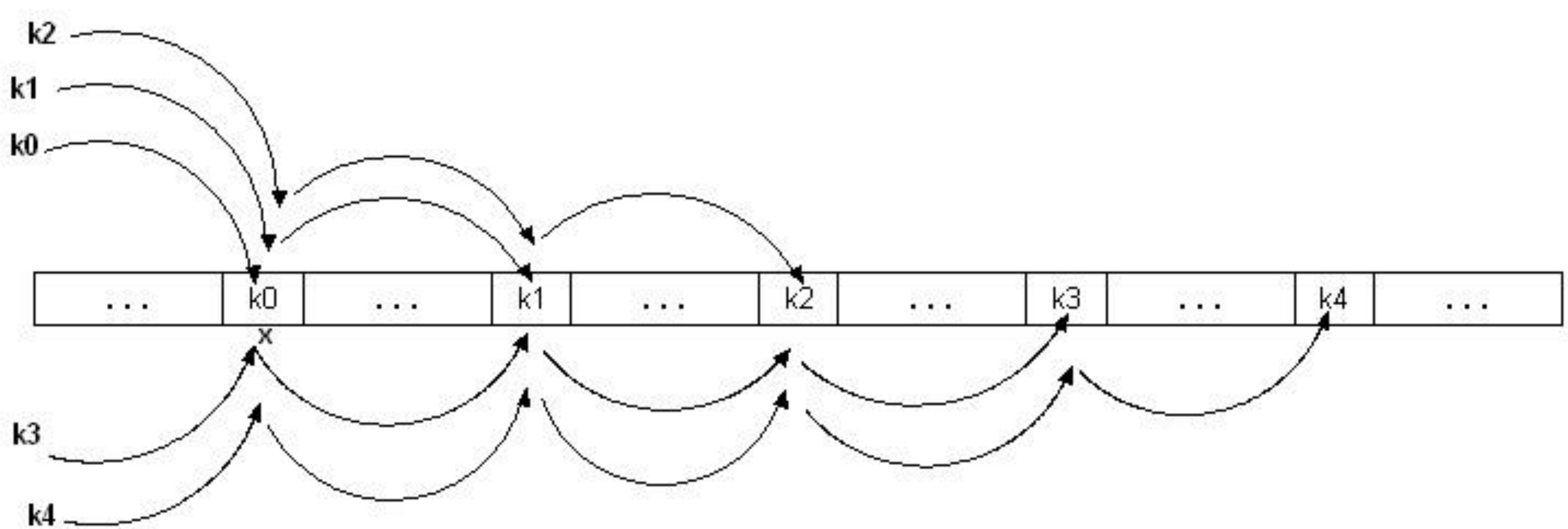
NORMALIZE: $(-3 + 7) \% 7 = 4$ collision

$h_3(15) = (1 + 3^2)\%7 = 3$

0	O	21
1	O	14
2	O	23
3	O	15
4	O	7
5	O	8
6	O	13

Secondary Clusters

- Quadratic probing is better than linear probing because it eliminates primary clustering.
- However, it may result in **secondary clustering**: if **h(k1) = h(k2)** the probing sequences for **k1** and **k2** are exactly the same. This sequence of locations is called a secondary cluster.
- Secondary clustering is less harmful than primary clustering because secondary clusters do not combine to form large clusters.
- **Example of Secondary Clustering:** Suppose keys **k0, k1, k2, k3, and k4** are inserted in the given order in an originally empty hash table using **quadratic probing** with **c(i) = i²**. Assuming that each of the keys hashes to the same array index **x**. A secondary cluster will develop and grow in size:



Double Hashing

- To eliminate secondary clustering, synonyms must have different probe sequences.

- Double hashing achieves this by having two hash functions that both depend on the hash key.

- $c(i) = i * h_p(key)$ **for i = 0, 1, . . . , tableSize – 1**

  where h_p (or h_2) is another hash function.

- The probing sequence is:

  $h_i(key) = [h(key) + i*h_p(key)]\%$ tableSize **for i = 0, 1, . . . , tableSize – 1**

- The function $c(i) = i*h_p(r)$ satisfies Property 2 provided $h_p(r)$ and tableSize are relatively prime.

- To guarantee Property 2, tableSize must be a prime number.

- Common definitions for $\mathbf{h_p}$ are :
  - $h_p(key) = 1 + key \% (tableSize - 1)$
  - $h_p(key) = q - (key \% q)$ where **q** is a prime less than **tableSize**
  - $h_p(key) = q*(key \% q)$ where **q** is a prime less than **tableSize**

Double Hashing (cont'd)

Performance of Double hashing:

- Much better than linear or quadratic probing because it eliminates both primary and secondary clustering.
- BUT requires a computation of a second hash function $\mathbf{h_p}$.

Example: Load the keys **18, 26, 35, 9, 64, 47, 96, 36, and 70** in this order, in an empty hash table of size **13**

(a) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: $\mathbf{h_p(key) = 1 + key \% 12}$

(b) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: $\mathbf{h_p(key) = 7 - key \% 7}$

Show all computations.

Double Hashing (cont'd)

0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

$h_0(18) = (18\%13)\%13 = 5$
$h_0(26) = (26\%13)\%13 = 0$
$h_0(35) = (35\%13)\%13 = 9$
$h_0(9) = (9\%13)\%13 = 9$ collision
$h_p(9) = 1 + 9\%12 = 10$
$h_1(9) = (9 + 1*10)\%13 = 6$
$h_0(64) = (64\%13)\%13 = 12$
$h_0(47) = (47\%13)\%13 = 8$
$h_0(96) = (96\%13)\%13 = 5$ collision
$h_p(96) = 1 + 96\%12 = 1$
$h_1(96) = (5 + 1*1)\%13 = 6$ collision
$h_2(96) = (5 + 2*1)\%13 = 7$
$h_0(36) = (36\%13)\%13 = 10$
$h_0(70) = (70\%13)\%13 = 5$ collision
$h_p(70) = 1 + 70\%12 = 11$
$h_1(70) = (5 + 1*11)\%13 = 3$

$h_i(key) = [h(key) + i*h_p(key)]\% 13$

$h(key) = key \% 13$

$h_p(key) = 1 + key \% 12$

Double Hashing (cont'd)

0	1	2	3	4	5	6	7	8	9	10	11	12
26	9				18	70	96	47	35	36		64

$h_i(key) = [h(key) + i*h_p(key)]\% \ 13$

$h(key) = key \ \% \ 13$

$h_p(key) = 7 - key \ \% \ 7$

$h_0(18) = (18\%13)\%13 = 5$

$h_0(26) = (26\%13)\%13 = 0$

$h_0(35) = (35\%13)\%13 = 9$

$h_0(9) = (9\%13)\%13 = 9$ collision

$h_p(9) = 7 - 9\%7 = 5$

$h_1(9) = (9 + 1*5)\%13 = 1$

$h_0(64) = (64\%13)\%13 = 12$

$h_0(47) = (47\%13)\%13 = 8$

$h_0(96) = (96\%13)\%13 = 5$ collision

$h_p(96) = 7 - 96\%7 = 2$

$h_1(96) = (5 + 1*2)\%13 = 7$

$h_0(36) = (36\%13)\%13 = 10$

$h_0(70) = (70\%13)\%13 = 5$ collision

$h_p(70) = 7 - 70\%7 = 7$

$h_1(70) = (5 + 1*7)\%13 = 12$ collision

$h_2(70) = (5 + 2*7)\%13 = 6$

Separate Chaining versus Open-addressing

Separate Chaining has several advantages over open addressing:

- Collision resolution is simple and efficient.
- The hash table can hold more elements without the large performance deterioration of open addressing (The load factor can be 1 or greater)
- The performance of chaining declines much more slowly than open addressing.
- Deletion is easy - no special flag values are necessary.
- Table size need not be a prime number.
- The keys of the objects to be hashed need not be unique.

Disadvantages of Separate Chaining:

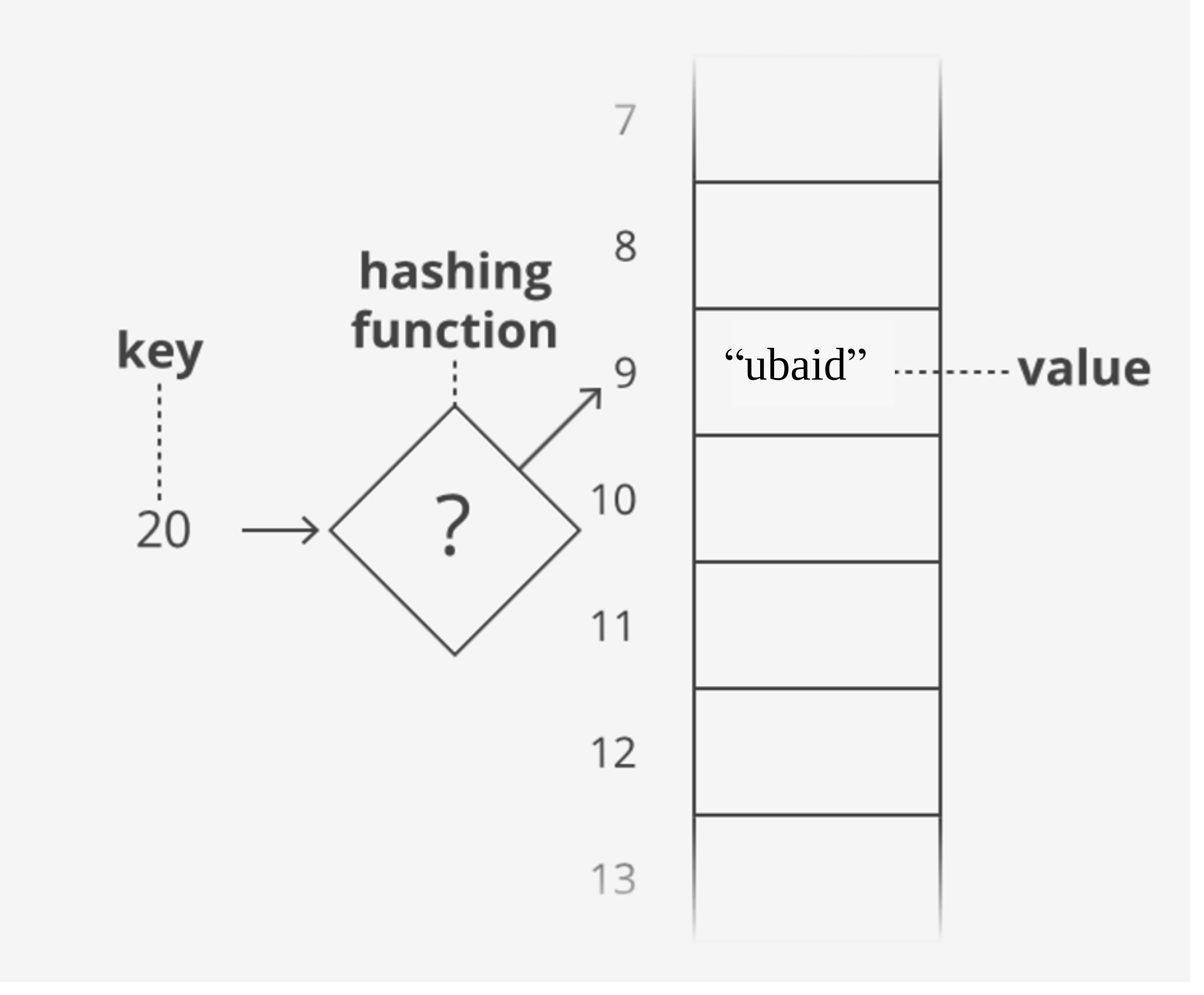
- It requires the implementation of a separate data structure for chains, and code to manage it.
- The main cost of chaining is the extra space required for the linked lists.
- For some languages, creating new nodes (for linked lists) is expensive and slows down the system.

unordered_map<key, mappedValue>

STL

unordered_map<int, string>

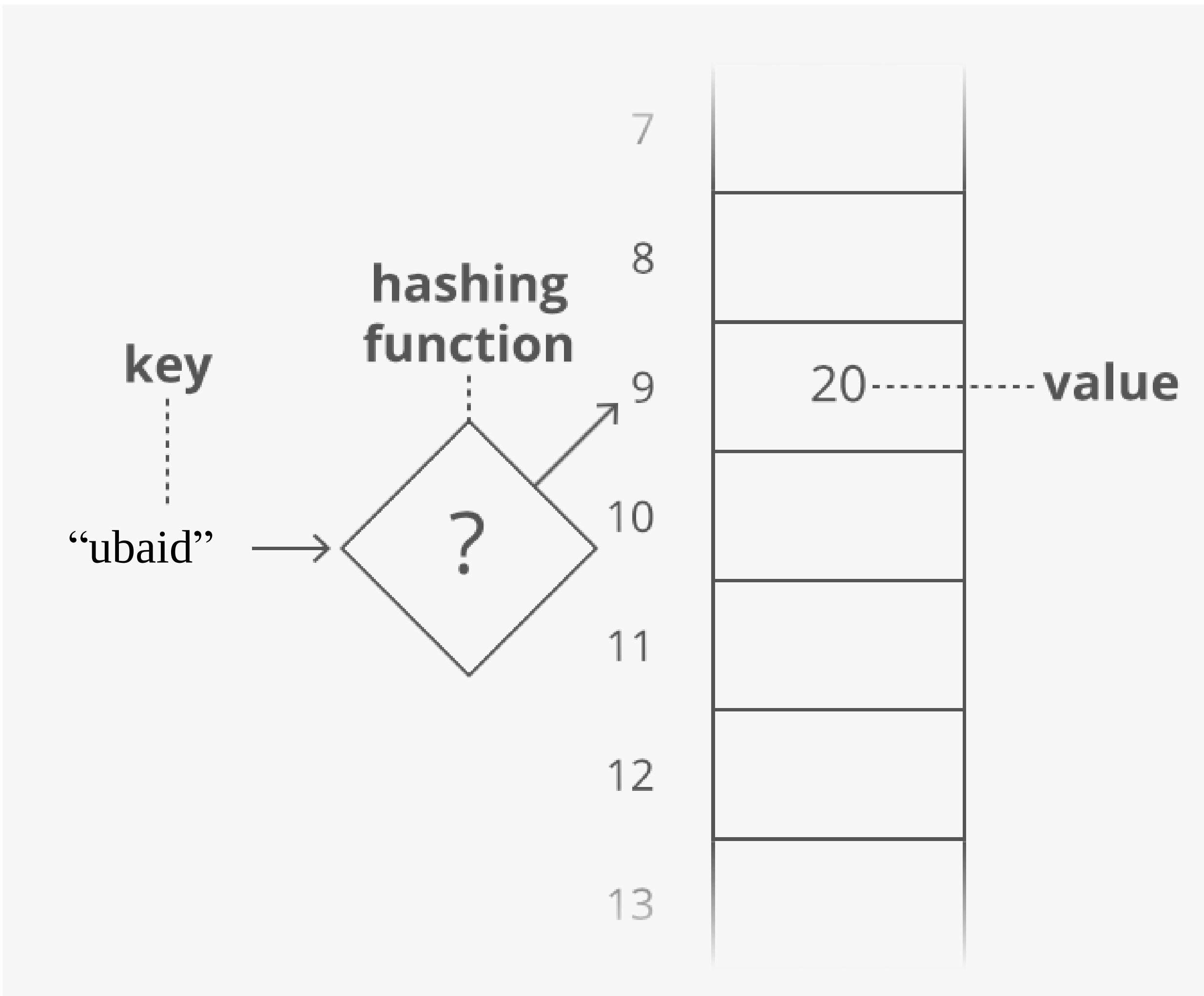
Example:
- key = 20
- mapped_value = “ubaid”



unordered_map<string, int>

Example:

- key = “ubaid”
- mapped_value = 20



Part 2

Hash Functions

Implementations So Far

	unsorted list	sorted array	Trees BST – average R-B – worst case
Search	O(n)	$O(\log_2 n)$	$O(\log_2 n)$

Properties of Good Hash Functions

- Must return an index number:
  0, 1, 2 …, [tablesize-1]
- Should be efficiently computable:
  O(1) time
- Should not waste space unnecessarily
  Load factor lambda λ = (no of keys / TableSize)
- Should minimize collisions

Integer Keys

- Hash(x) = x % TableSize
- Good idea to make TableSize *prime?* Why?

Suppose

data stored in hash table: 7160, 493, 60, 55, 321, 900, 810

tableSize = 10

data hashes to 0, 3, 0, 5, 1, 0, 0

tableSize = 11

data hashes to 10, 9, 5, 0, 2, 9, 7

Integer Keys

- Hash(x) = x % TableSize
- Good idea to make TableSize *prime?* Why?
- There is a high probability that collision will be avoided (it will not be eliminated however)