Data Structures
CS 2014

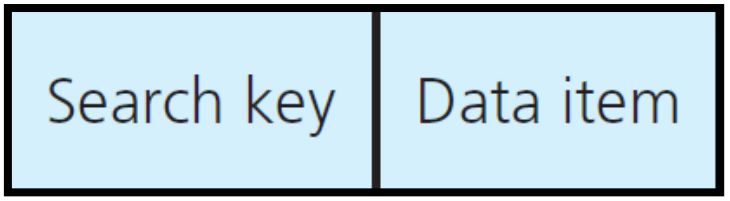
Dictionaries (Maps)

By

Marwa M. A. Elfattah

Dictionary (Map)

- A dictionary is composed of a collection of elements each of which is:
  - Set of (key, value) pairs
  - keys are mapped to values
  - keys must be comparable and unique



A dictionary entry

- Examples:
  - Membership in a club
  - Set of loans made in a library
  - Language dictionary

Key →

ID	student name	hw1	
123	Stan Smith	49	...
124	Sue Margolin	56	...
125	Billie King	34	...
⋮			
167	Roy Miller	39	...

Possible Implementations of Dictionary?

- The most important question in dictionary ADT is how to search – to reach to an element with specific key.
- So Dictionaries can be implemented as:
  - Sorted Lists
  - Trees (BST, AVL, Splay,….)
  - Hash tables

AVL Tree

Balanced BST

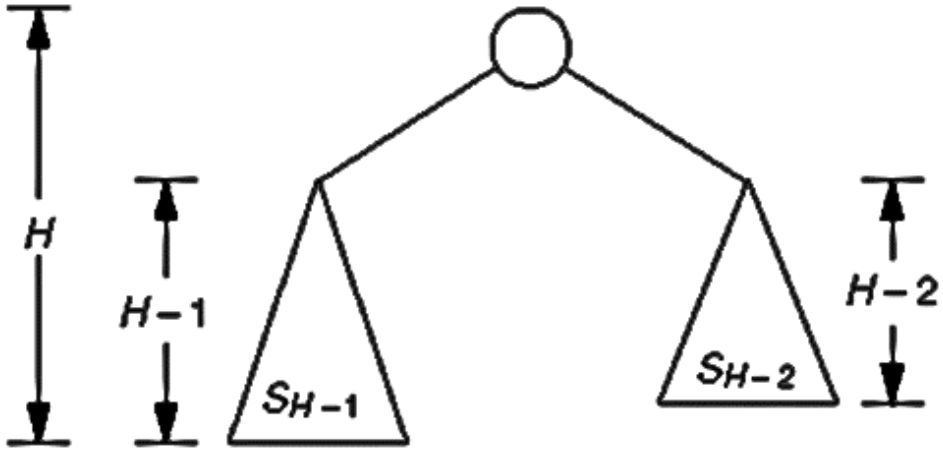
- The disadvantage of a binary search tree is that its height can be as large as N-1 – where N is the number of nodes in the tree.
- Thus, our goal is to keep the height of a binary search tree to be as small as we can.
- Such trees are called **balanced** binary search trees. Examples are ***AVL tree*** and red-black tree.

AVL Trees

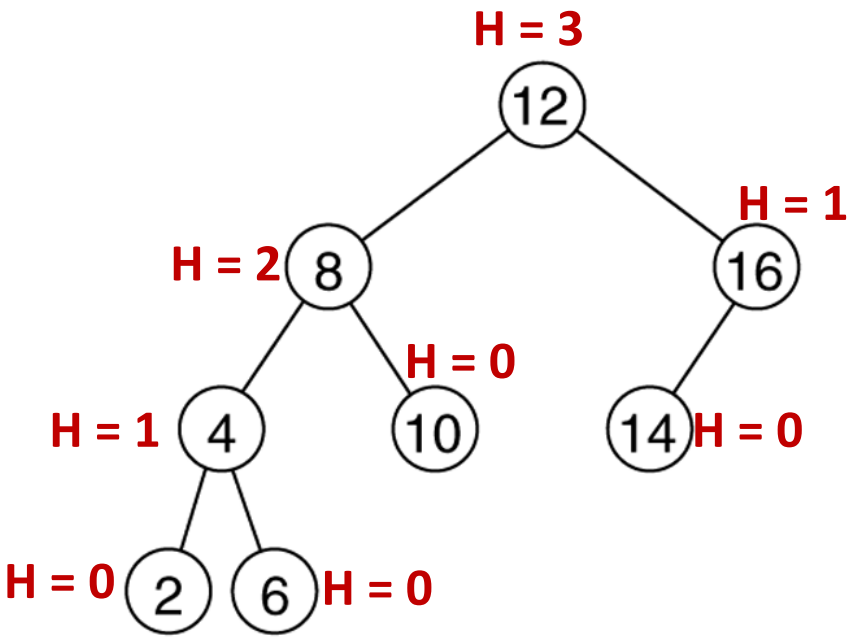
- An AVL tree is a binary search tree with a ***balance* condition – balance factor**:

  ➔for any node in the tree, the height of the left and right subtrees can ***differ by at most 1***.

- It maintains a ***height close to the minimum,*** which *approximates* the ideal tree (completely balanced tree).



AVL Trees



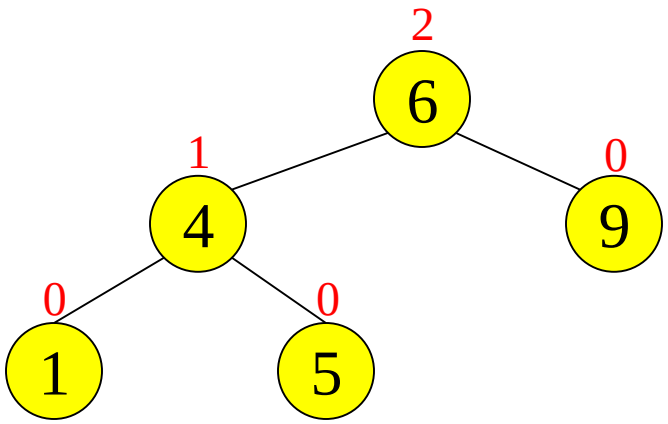
Try to Insert 1

*The height of a leaf is **0**. The height of a NULL pointer is **-1**.*

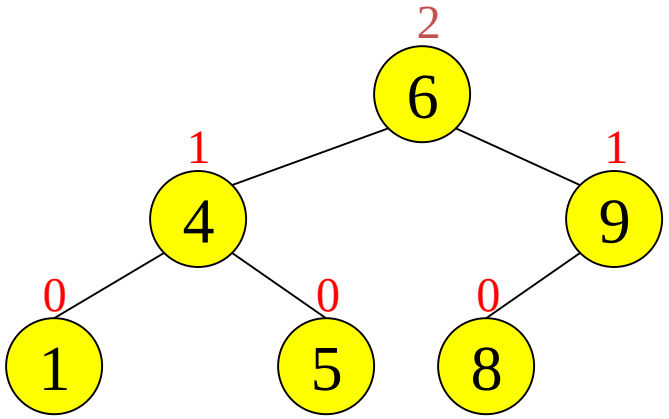
*The height of an internal node is the **maximum height of its children plus 1***

AVL Trees

Tree A (AVL)



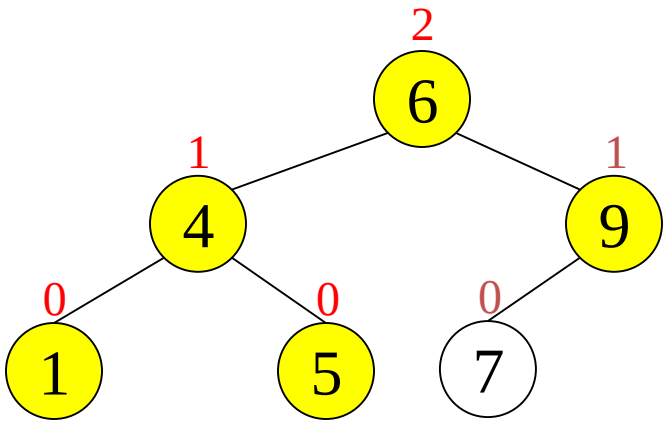
Tree B (AVL)



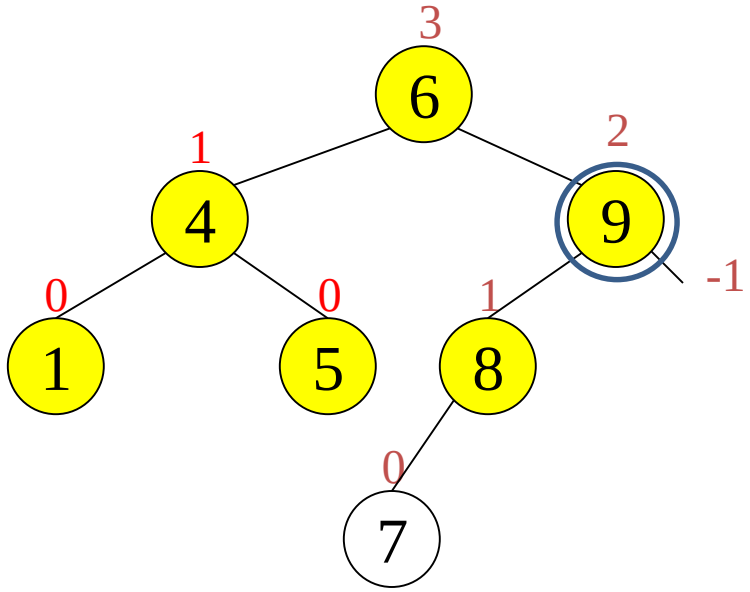
Now: Insert 7

AVL Trees

Tree A (AVL)



Tree B (not AVL)



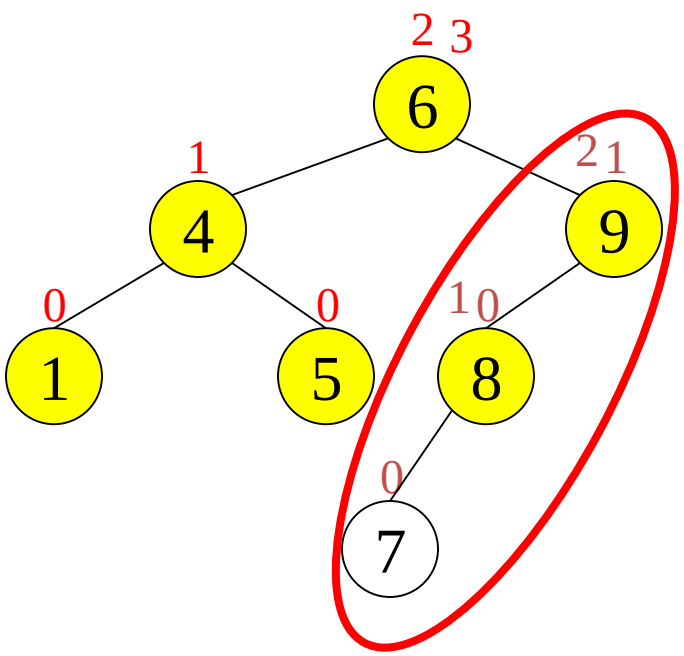
AVL Tree Implementation

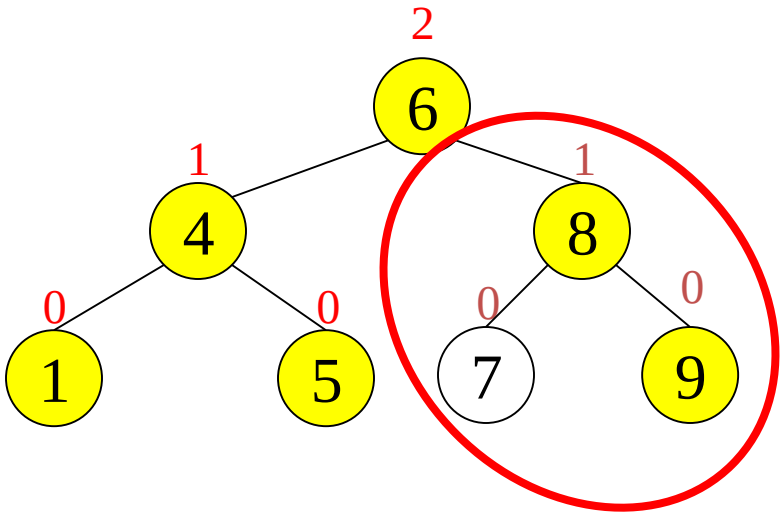
```
typedef struct {
    EntryType       info;
    NodeType        *right;
    NodeType        *left;
    int             height;
} AVLNodeType;

typedef  NodeType * TreeType
```

Insert and Deletion in AVL Trees

- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) and right(x) differ by exactly 2.
- If a balance factor (the difference h_{left}-h_{right}) become 2 or –2, adjust tree by *rotation* around the node

Insert in AVL Trees





Hash Tables

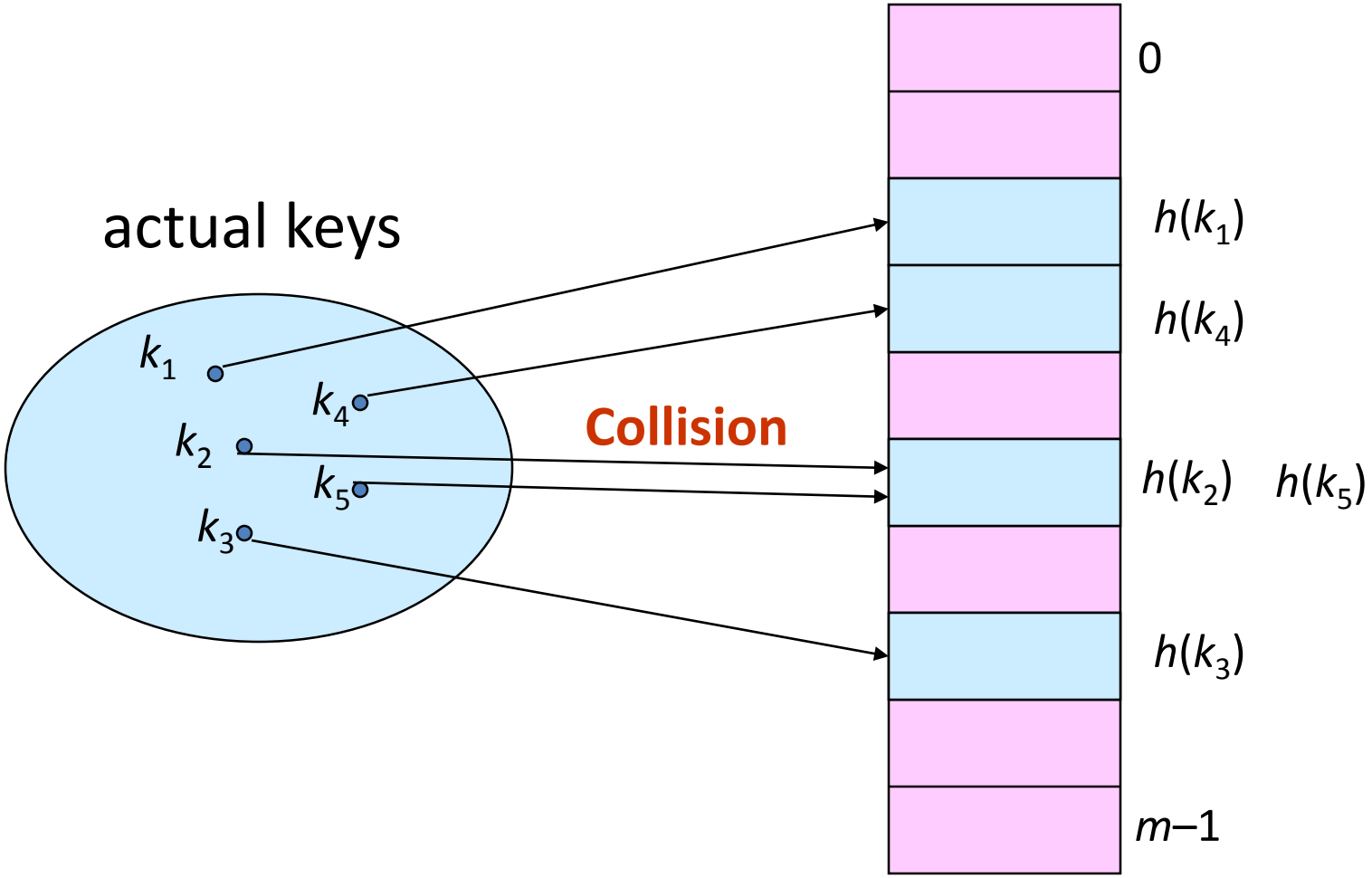
Hashing

- Use *hash function* to map keys into positions in a *hash table*

Ideally:

- If element *e* has key *k* and *h* is hash function, then *e* is stored in position *h(k)* of table
- To search for *e*, compute *h(k)* to locate position. If no element, dictionary does not contain *e*.

Hashing



Where, h(k) is a hash function that maps the key k to specific position in the hash table

Hash Table

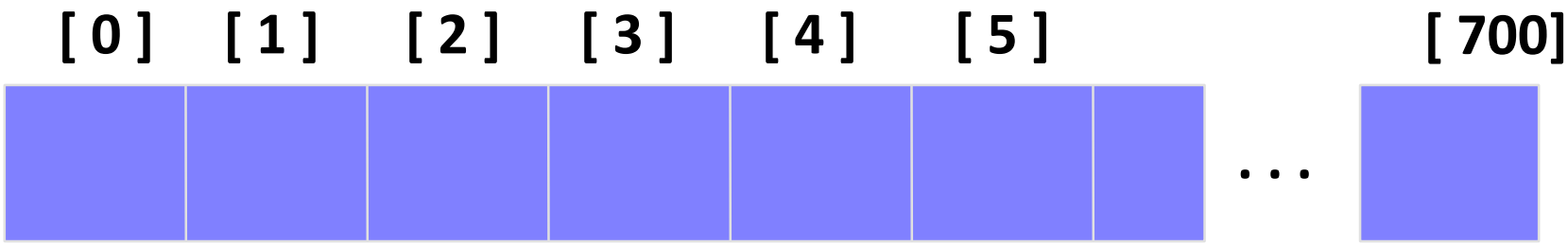
- Effective way of implementing dictionaries.

- The ADT **Hash Table** is an array of elements (associated with a search key unique for each element), together with an **hash function** and **access procedures** (insert, delete, search...).

- The **hash function** takes a key and maps it into an integer array index.

What is a Hash Table ?

- The simplest kind of hash table is an array of records.
  - *Element whose key is k is obtained by indexing into the k^{th} position of the array.*
- Applicable when we can afford to allocate an array with one position for every possible key.



What if can not??

Properties of Good Hash Functions

- Must return number 0, …, tablesize
- Should be efficiently computable.
- Should minimize collisions

  ➔ different keys hashing to same index

Hashing Function Examples

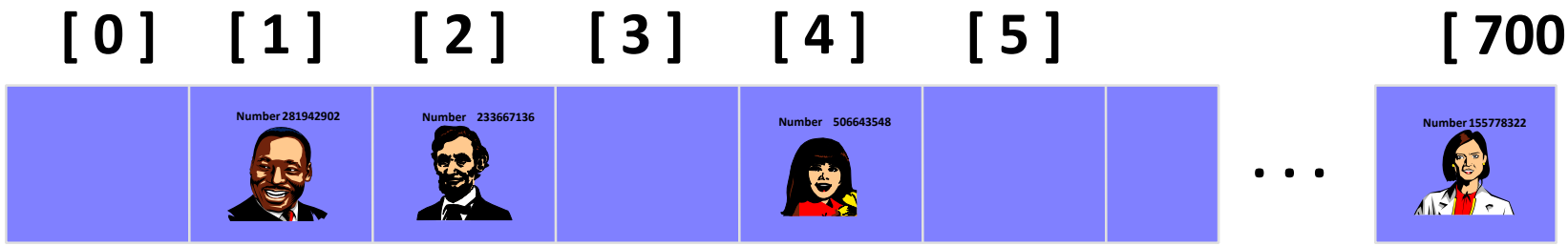
- For Integers could be:
  - **Modulo arithmetic**: given a search key number, the function defines the index to be the modulo arithmetic of the search value with some fix number. *i.e: h(key) = Key% TableSize.*
  - **Selecting digits**: given a search key number composed of a certain number of digits
    the hash function picks digits at specific places in the search key number.

    Ex: h(001364825) = 825 (select the first 3 digits)
- If keys are strings, can get an integer by adding up ASCII values of characters in *key.*
  - What if keys often contain the same characters ("abc", "bca", etc.)? ---Think

Let’s Try…

Inserting a New Record

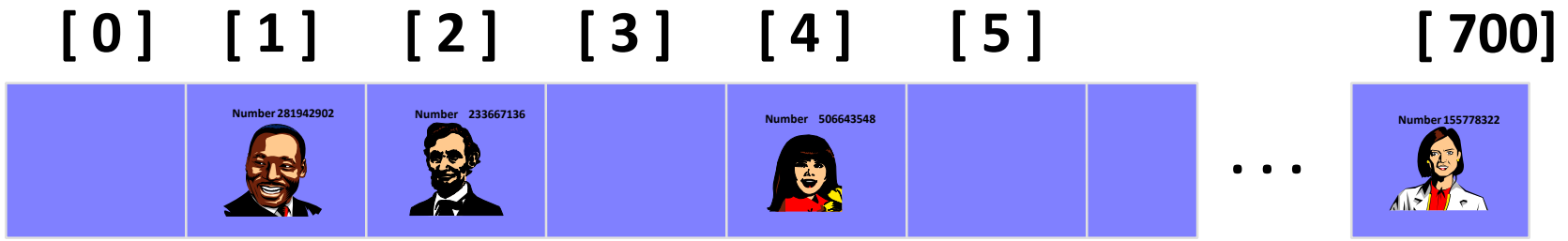
- In order to insert a new record, the key must somehow be converted to an array index.
- The index is called the hash value of the key.



Inserting a New Record

- Typical way create a hash value: (**Key mod TableSize**)

What is (580625685 mod 701) ?

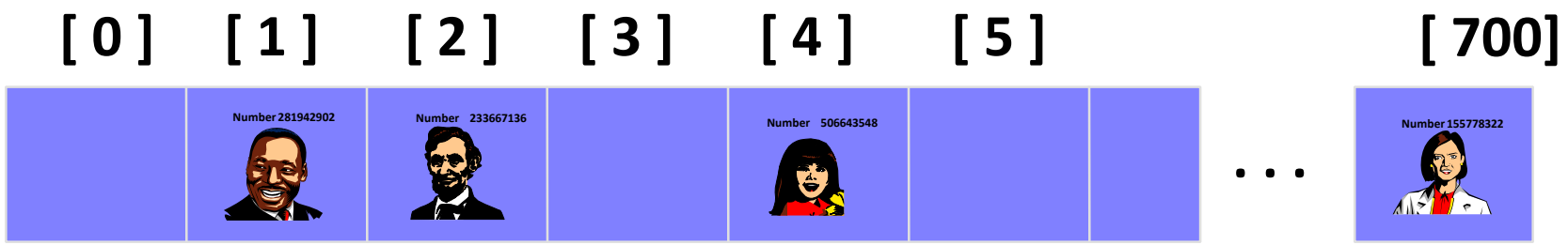


Inserting a New Record

- Typical way to create a hash value:

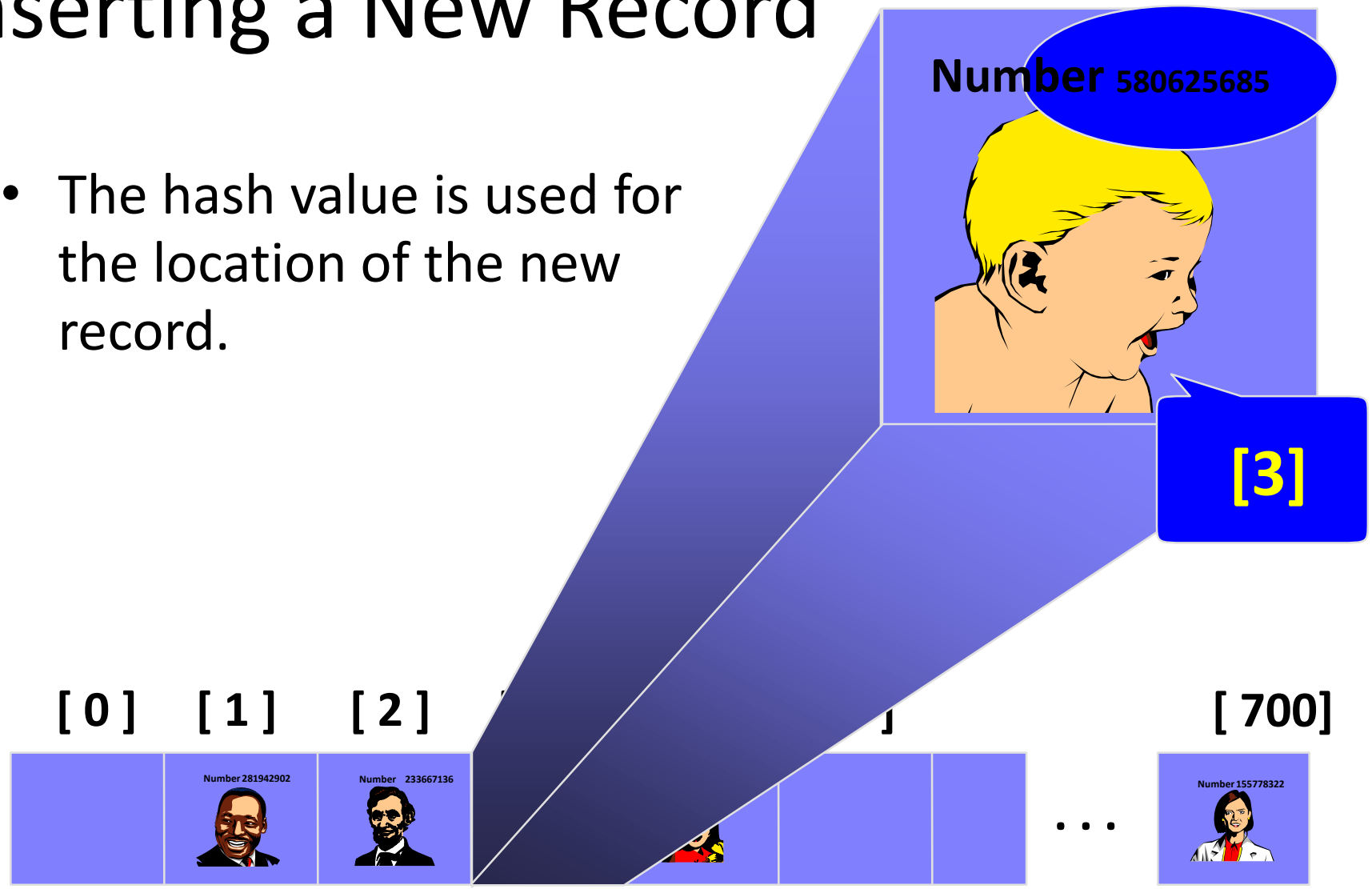
  (Number mod 701)

What is (580625685 mod 701)?



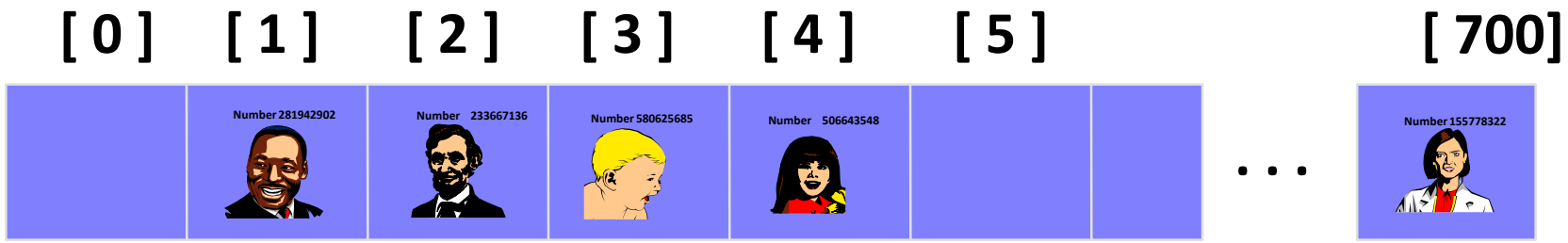
Inserting a New Record

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



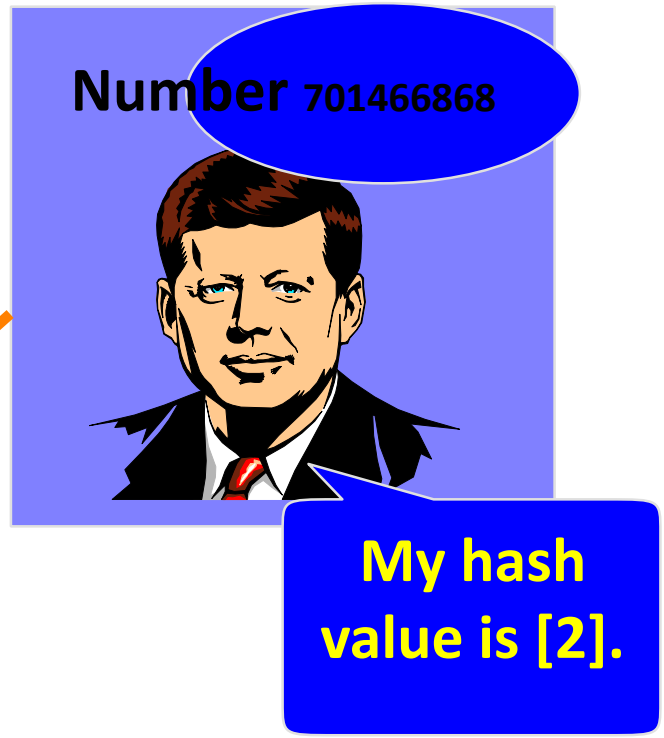
Inserting a New Record

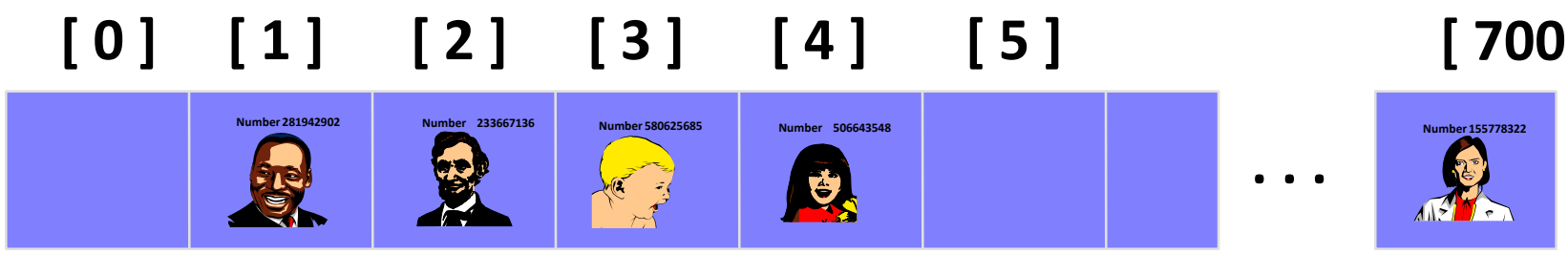
- The hash value is used for the location of the new record.



Collisions

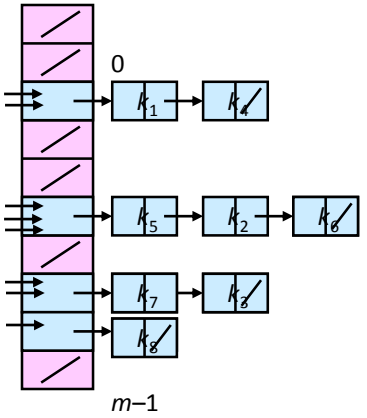
- Here is another new record to insert, with a hash value of 2.



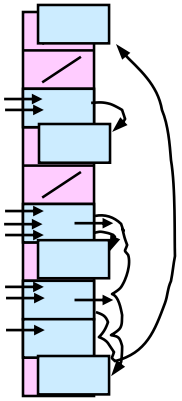


Methods of Collision Resolution

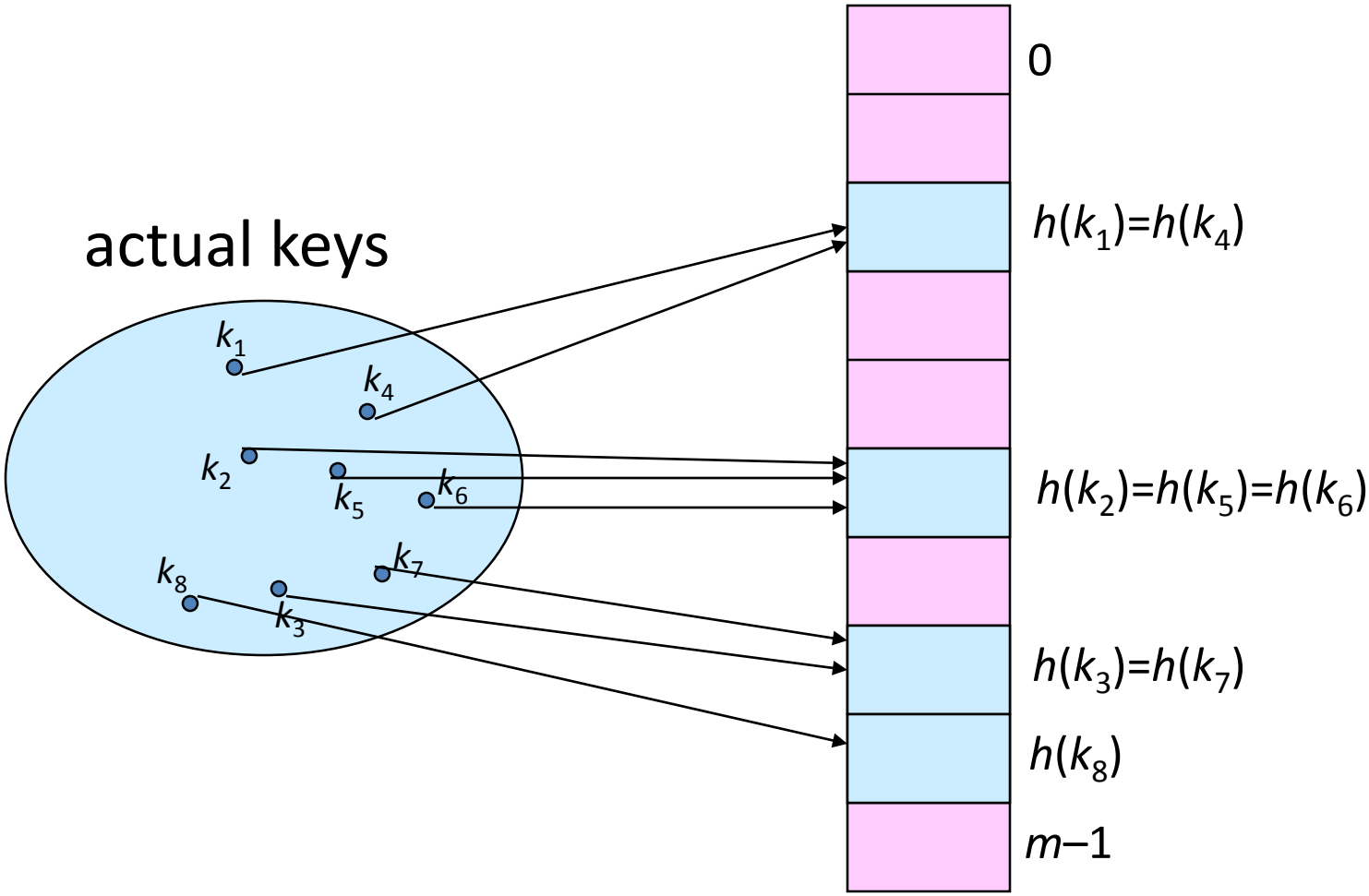
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



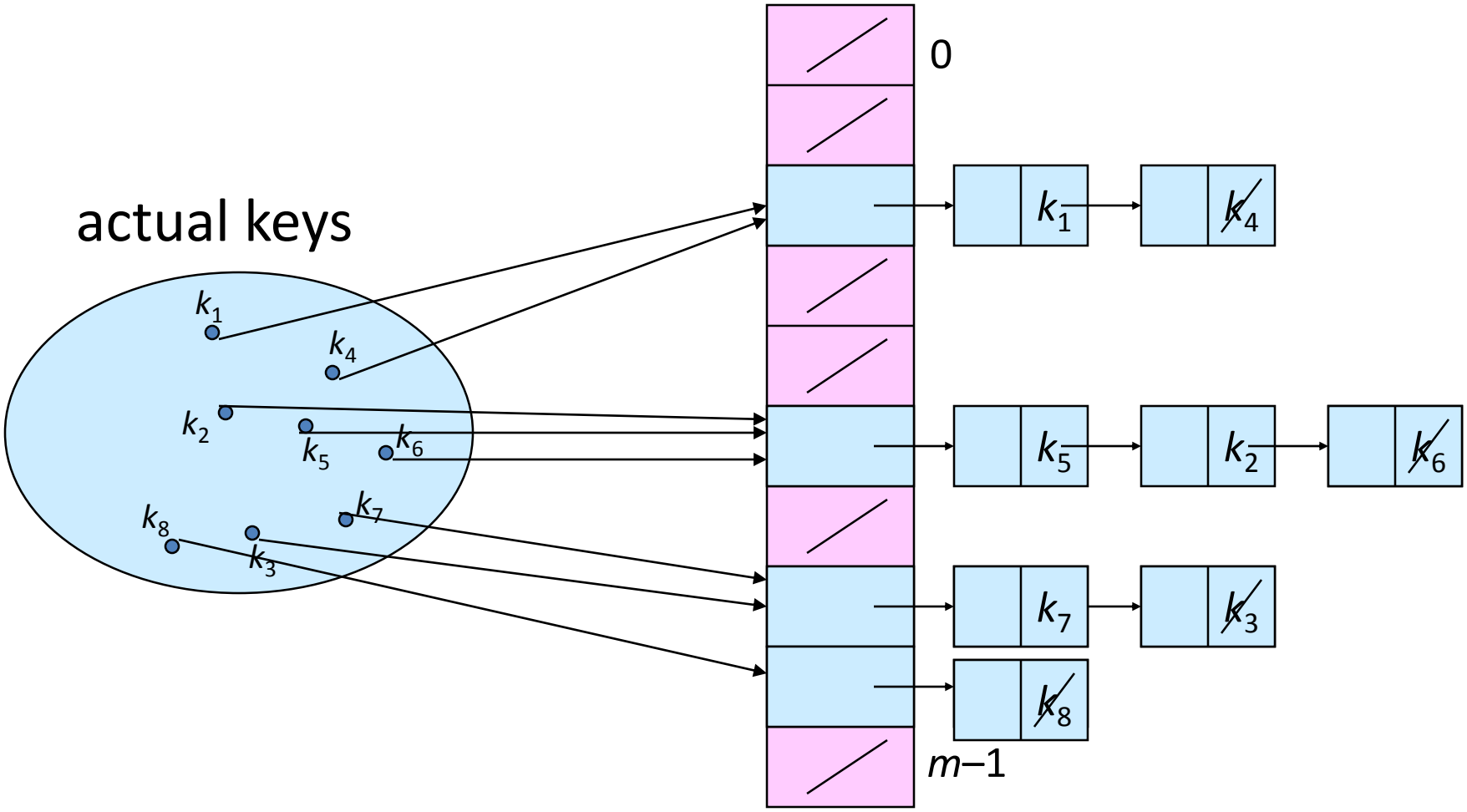
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining

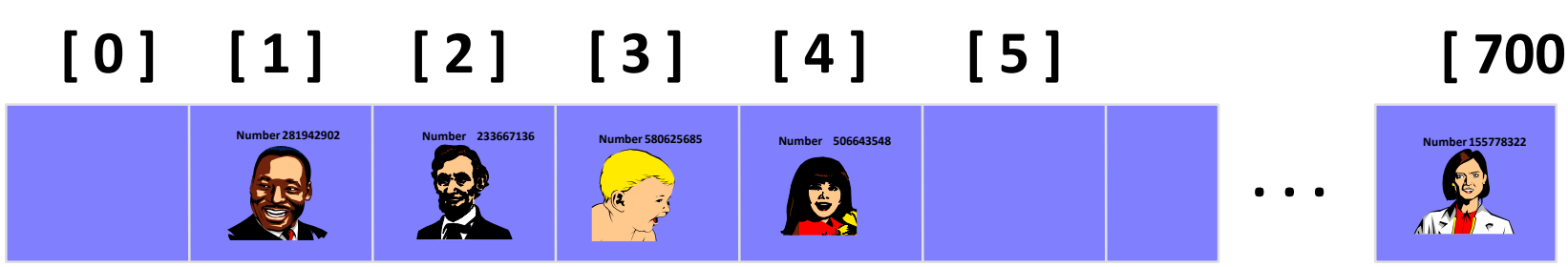


Collision Resolution by Chaining



Collisions Resolution by open addressing

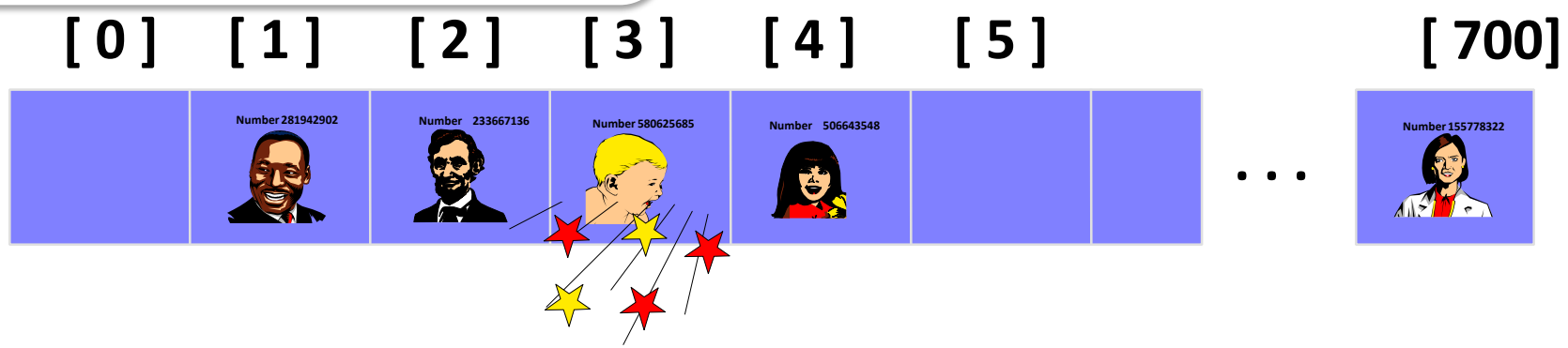
- Here is another new record to insert, with a hash value of 2.



Collisions Resolution by open addressing

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

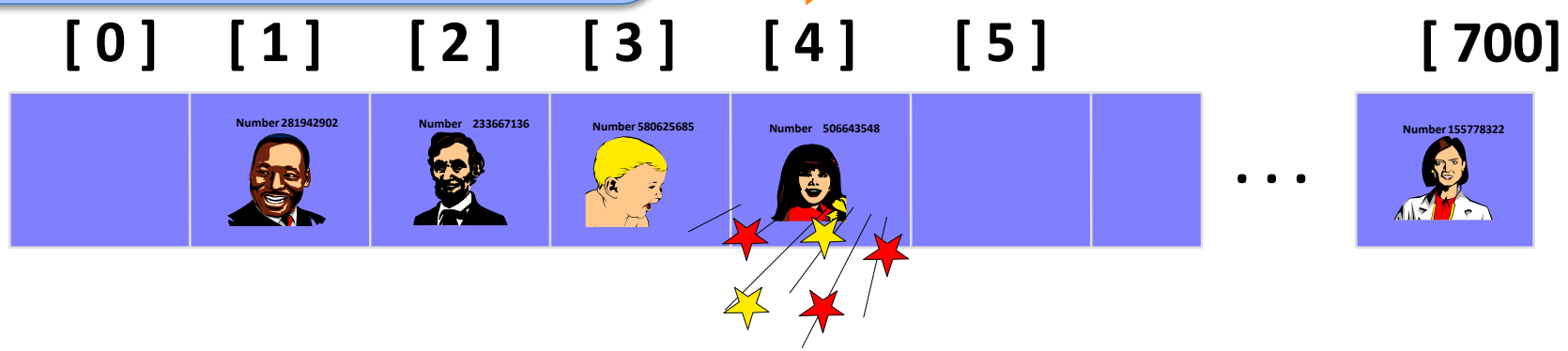
When a collision occurs, move forward until you find an empty spot.



Collisions Resolution by open addressing

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

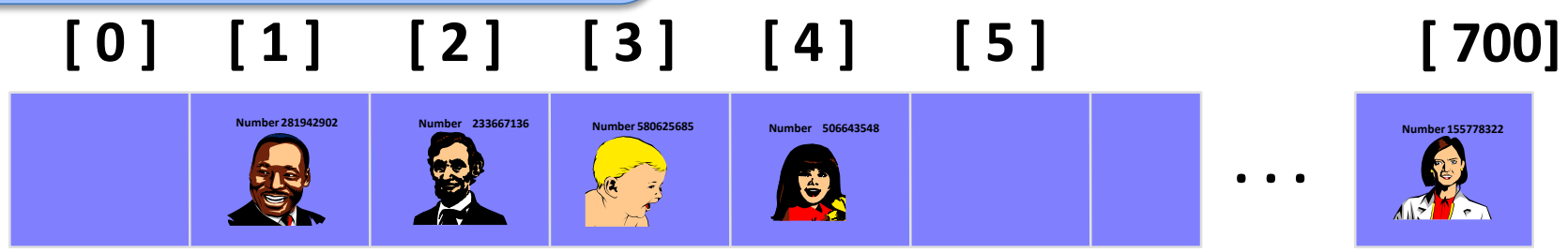
When a collision occurs, move forward until you find an empty spot.



Collisions Resolution by open addressing

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

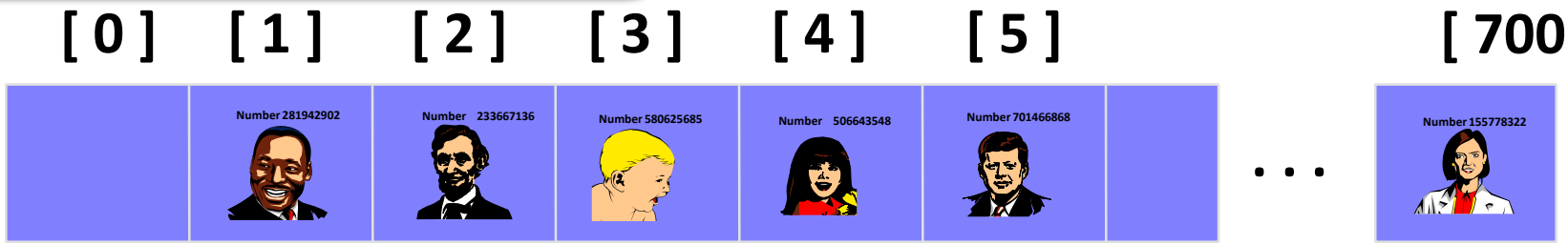
When a collision occurs, move forward until you find an empty spot.



Collisions Resolution by open addressing

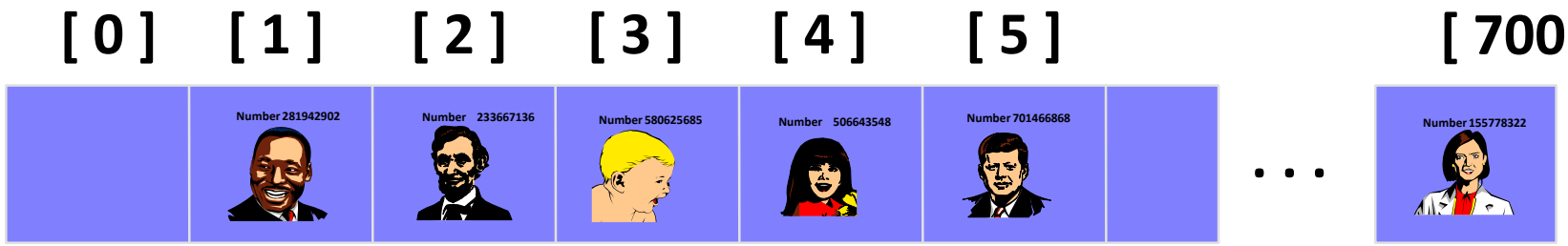
- This is called a **collision**, because there is already another valid record at [2].

The new record goes in the empty spot.



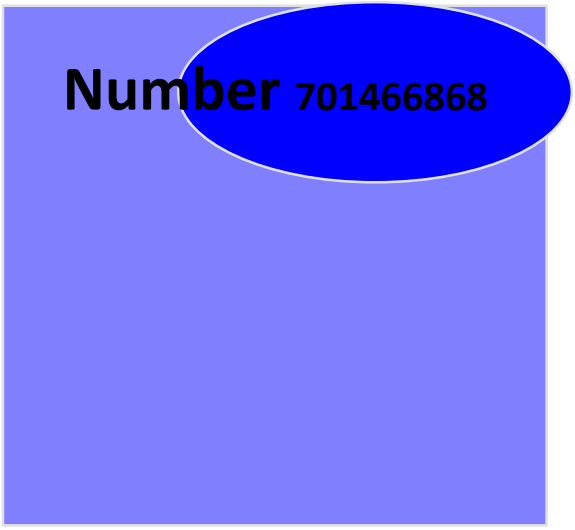
Collisions Resolution by open addressing

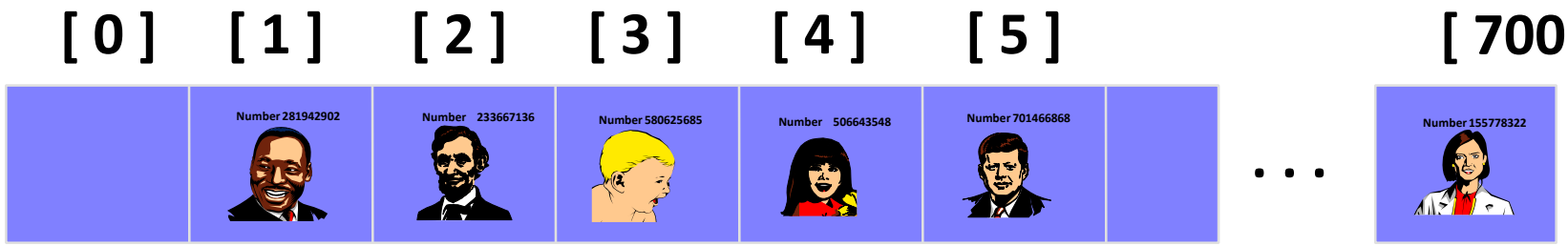
- How can we add another person to location 700?? Think



Searching for a Key

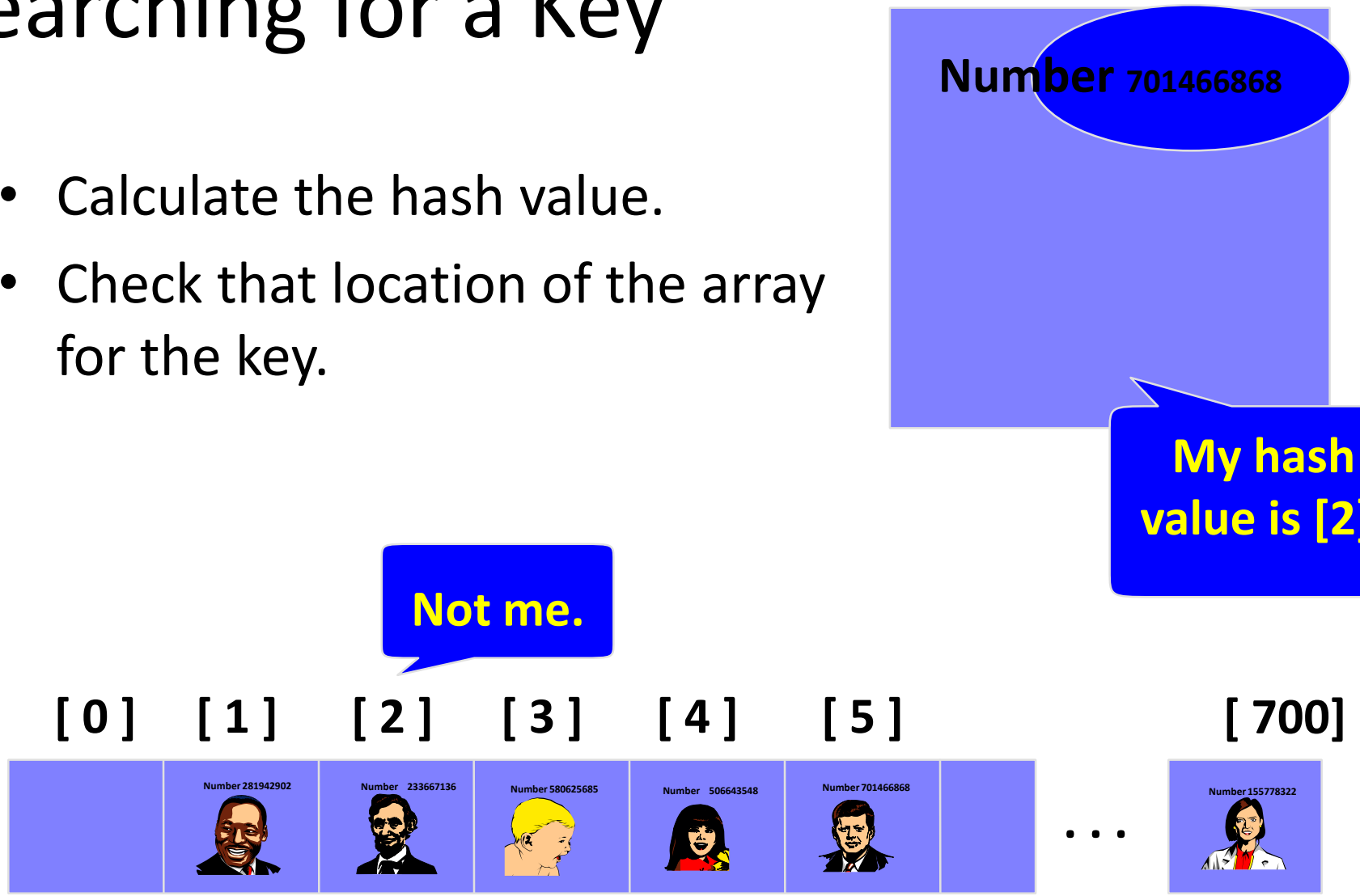
- The data that's attached to a key can be found fairly quickly.





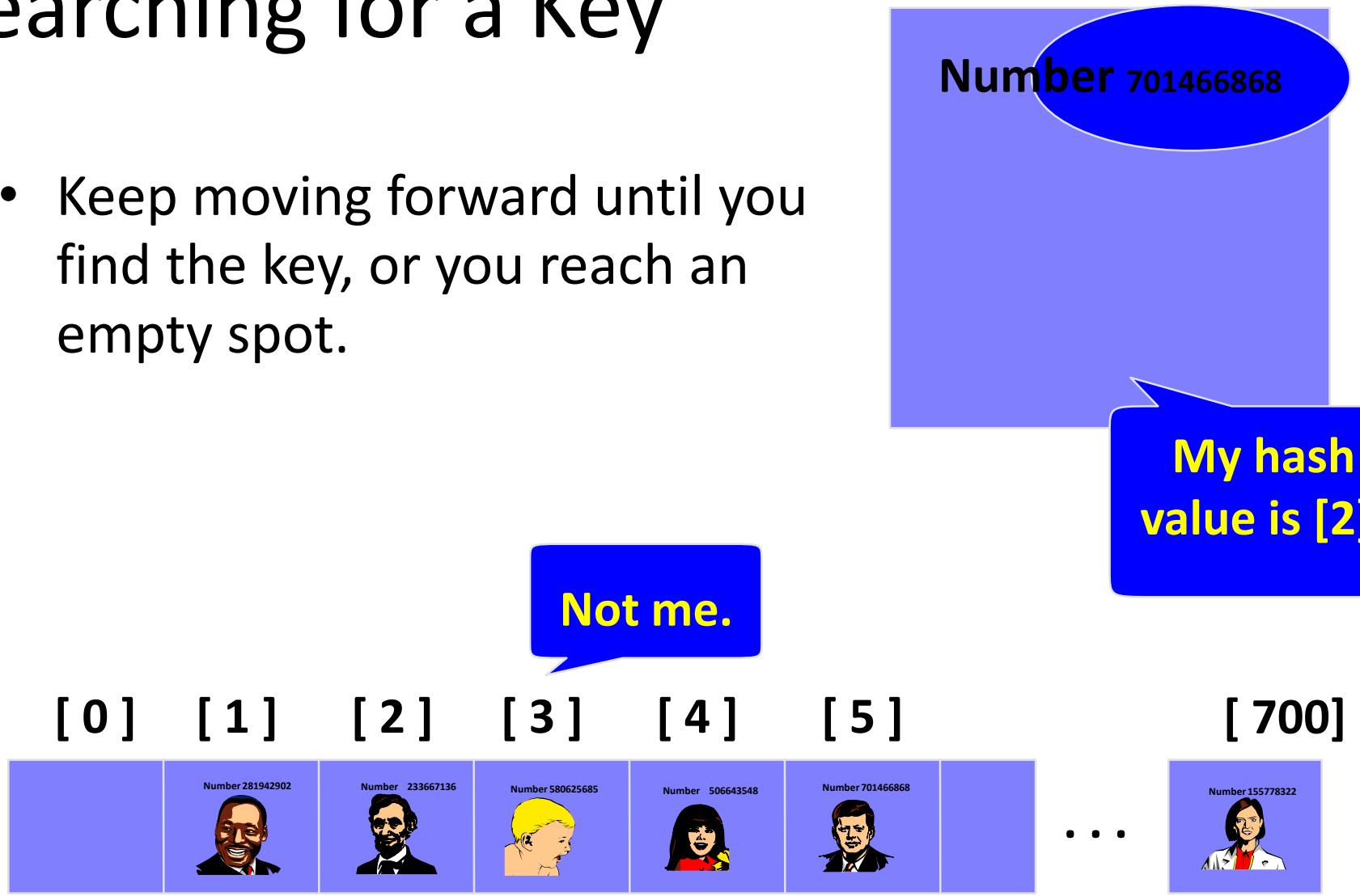
Searching for a Key

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



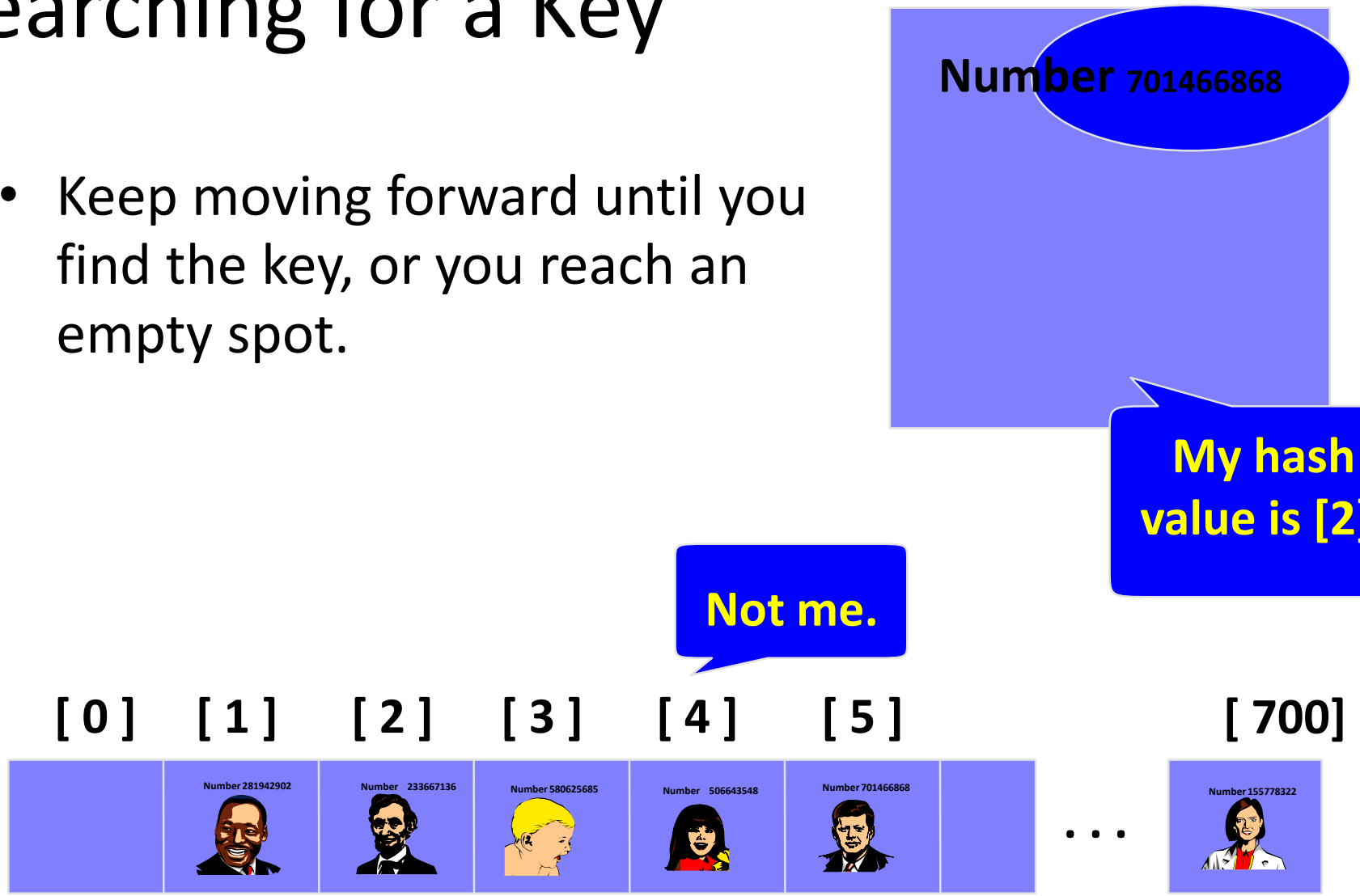
Searching for a Key

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



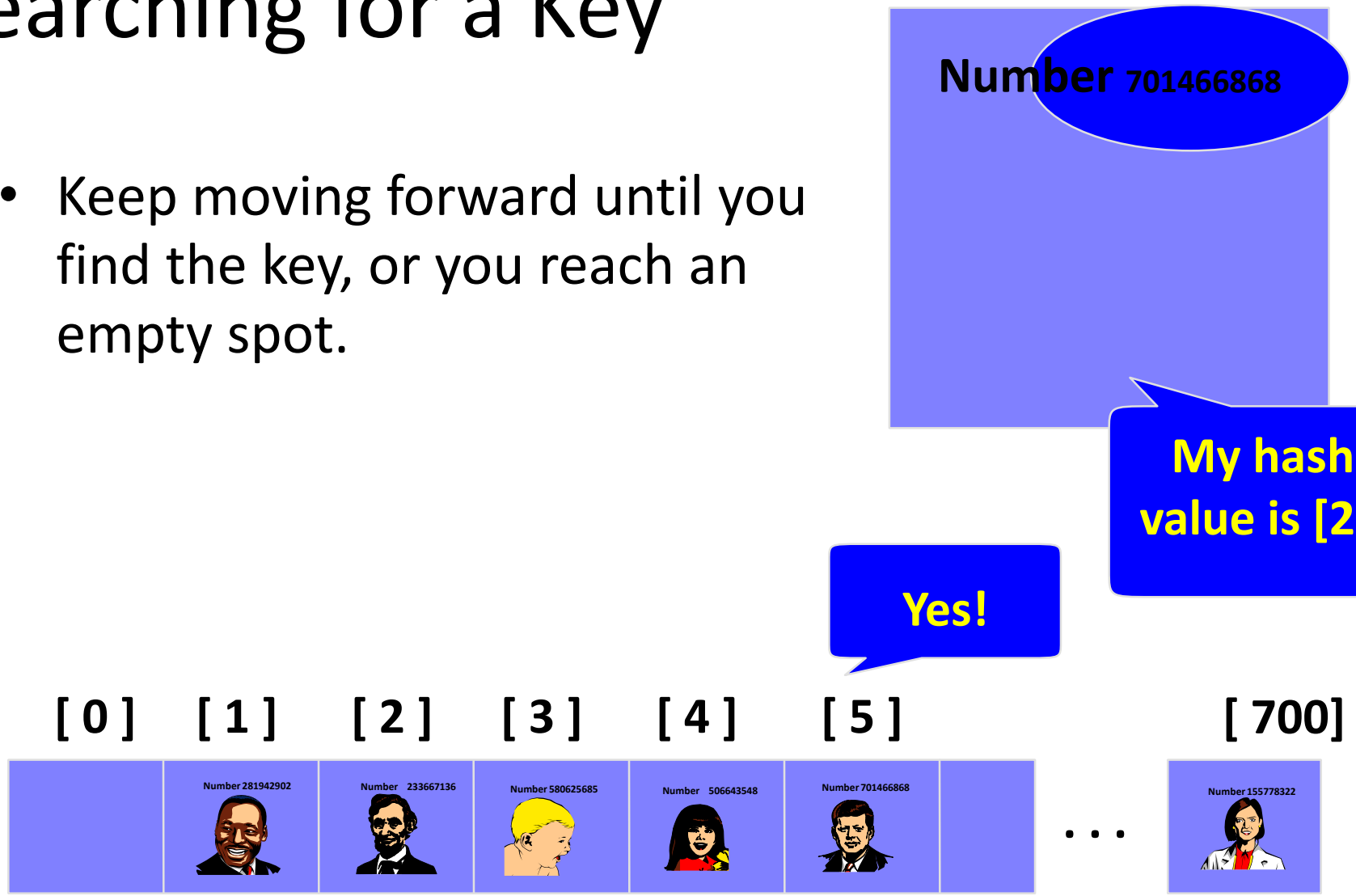
Searching for a Key

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



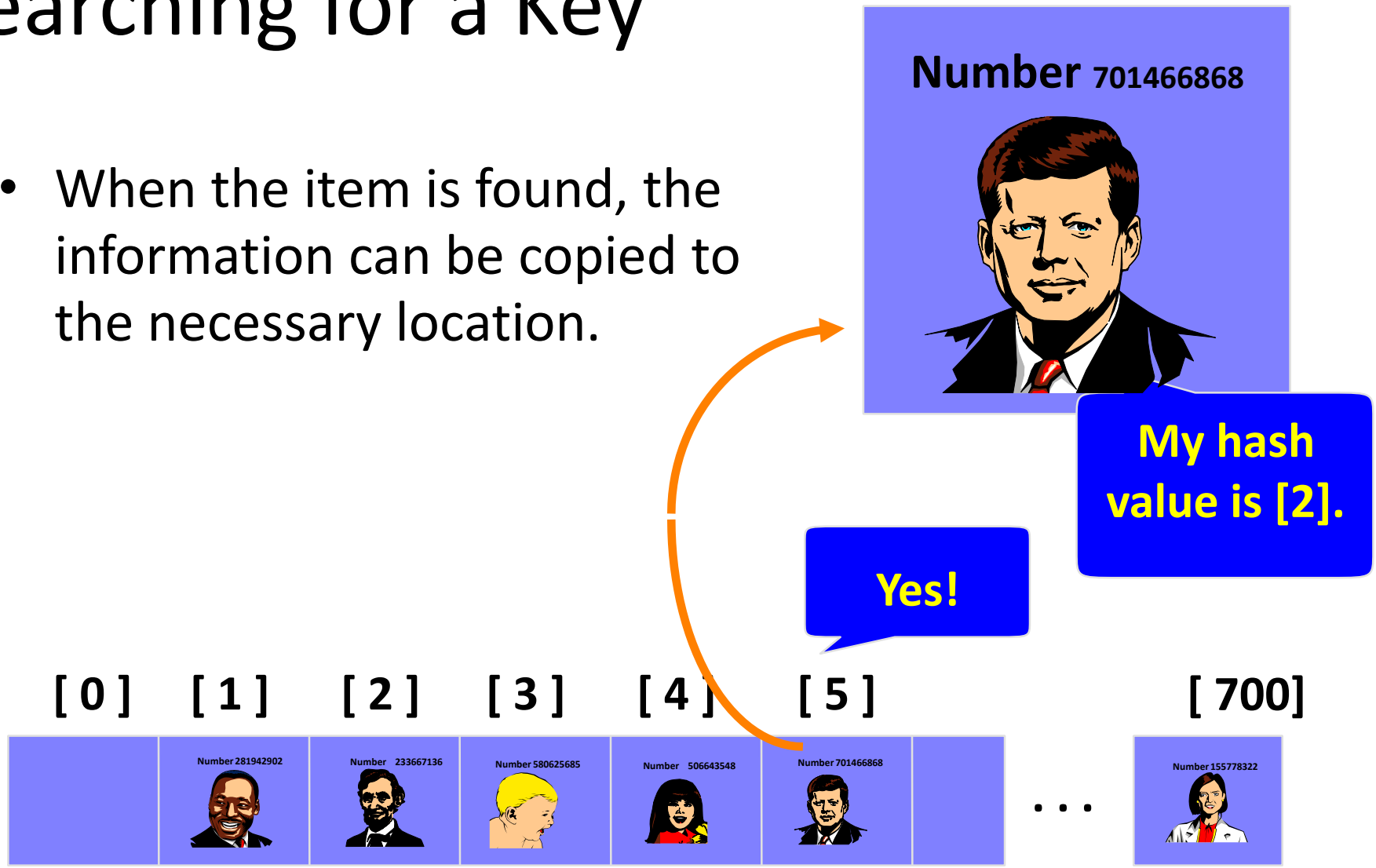
Searching for a Key

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



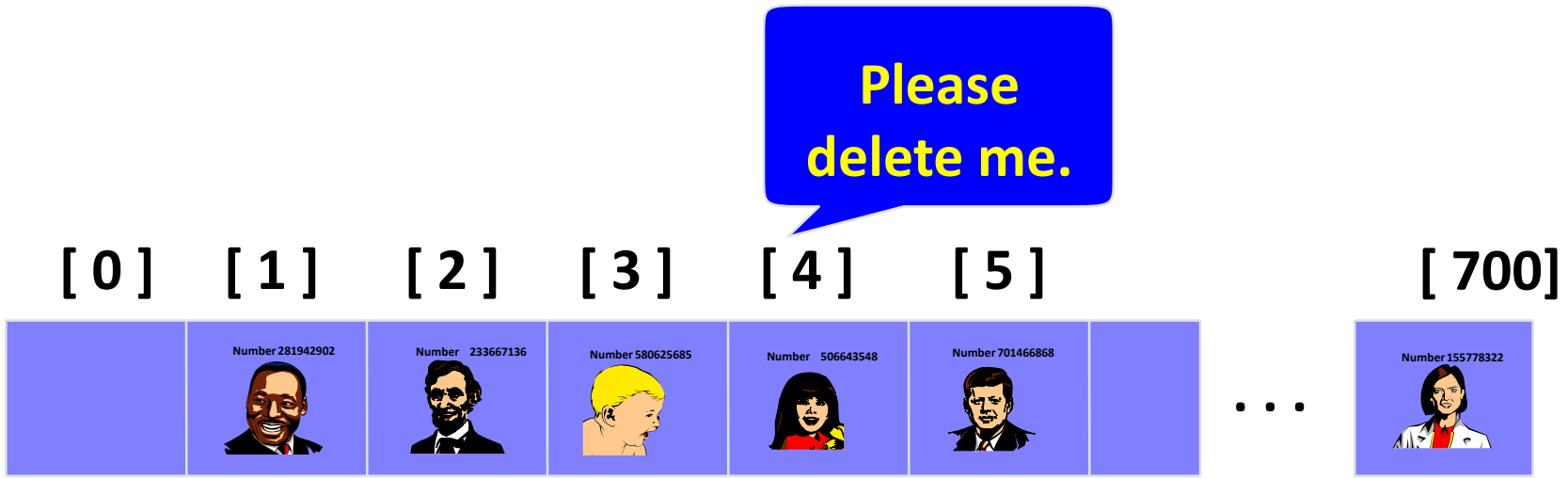
Searching for a Key

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



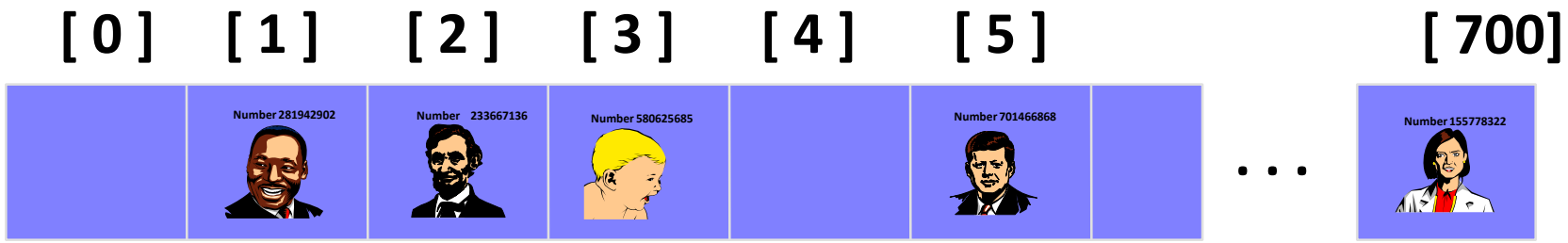
Deleting a Record

- Records may also be deleted from a hash table.



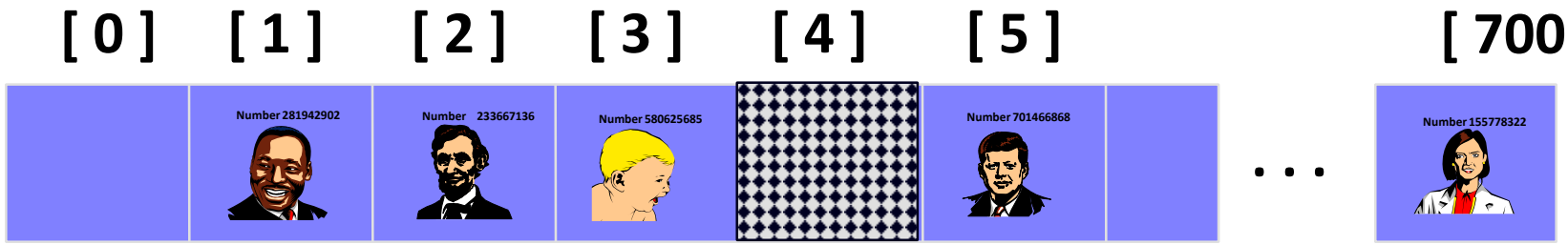
Deleting a Record

- 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. (HOW?? Think)



Thank you