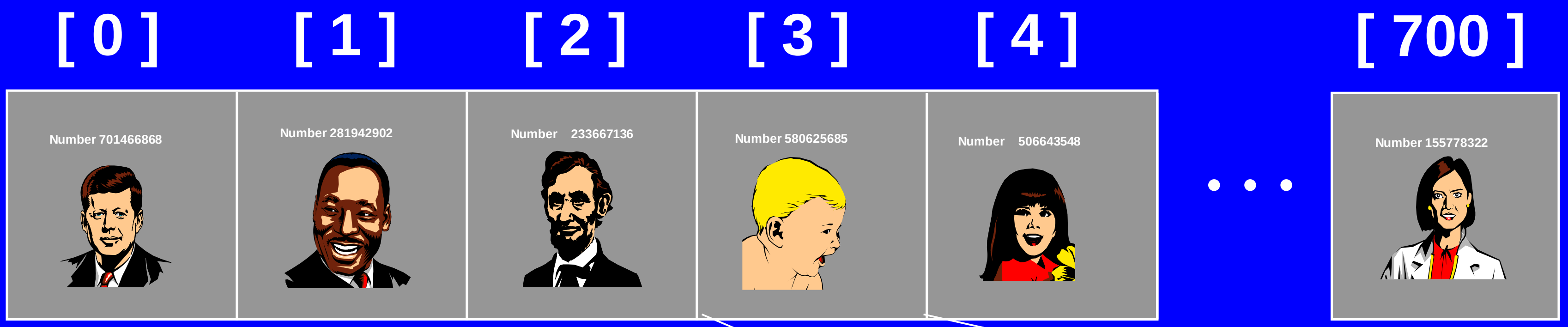
Searching

Problem: Search

- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.

Search



Each record in list has an associated key.
In this example, the keys are ID numbers.

Given a particular key, how can we efficiently retrieve the record from the list?

Serial Search

- Step through array of records, one at a time.
- Look for record with matching key.
- Search stops when
  - record with matching key is found
  - or when search has examined all records without success.

Pseudocode for Serial Search

```
// Search for a desired item in the n array elements
// starting at a[first].
// Returns pointer to desired record if found.
// Otherwise, return NULL
…
for(i = first; i < n; ++i )
        if(a[first+i] is desired item)
                return &a[first+i];

// if we drop through loop, then desired item was not found
return NULL;
```

Serial Search Analysis

- What are the worst and average case running times for serial search?
- We must determine the O-notation for the number of operations required in search.
- Number of operations depends on *n*, the number of entries in the list.

Worst Case Time for Serial Search

- For an array of n elements, the worst case time for serial search requires n array accesses: $O(n)$.
- Consider cases where we must loop over all n records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all

Average Case for Serial Search

Assumptions:

1. All keys are equally likely in a search
2. We always search for a key that is in the array

Example:

- We have an array of 10 records.
- If search for the first record, then it requires 1 array access; if the second, then 2 array accesses. *etc.*

The average of all these searches is:

$$(1+2+3+4+5+6+7+8+9+10)/10 = 5.5$$

Average Case Time for Serial Search

Generalize for array size *n*.

Expression for average-case running time:

$(1+2+\ldots+n)/n = n(n+1)/2n = (n+1)/2$

Therefore, average case time complexity for serial search is O(n).

Binary Search

- Perhaps we can do better than O(n) in the average case?
- Assume that we are give an array of records that is sorted. For instance:
  - an array of records with integer keys sorted from smallest to largest (e.g., ID numbers), or
  - an array of records with string keys sorted in alphabetical order (e.g., names).

Binary Search Pseudocode

```
…
if(size == 0)
    found = false;
else {
    middle = index of approximate midpoint of array segment;
    if(target == a[middle])
        target has been found!
    else if(target < a[middle])
        search for target in area before midpoint;
    else
        search for target in area after midpoint;
}
…
```

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is 7 = midpoint key? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is 7 < midpoint key? YES.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search for the target in the area before midpoint.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Target = key of midpoint? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Target < key of midpoint? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Target > key of midpoint? YES.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search for the target in the area after midpoint.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint.
Is target = midpoint key? YES.

Binary Search Implementation

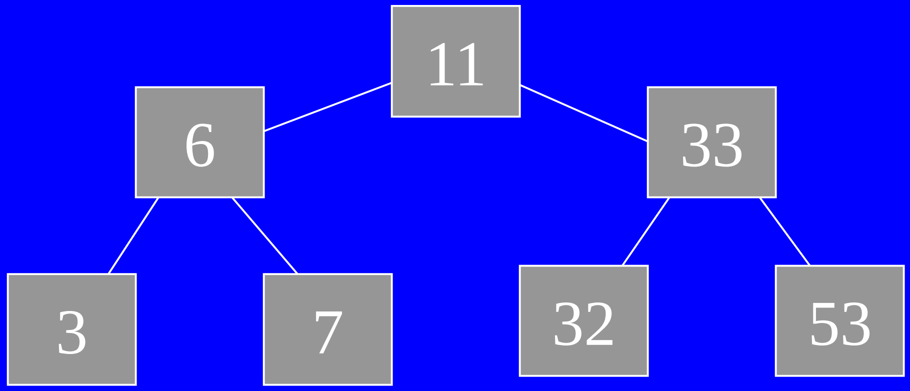
```
void search(const int a[ ], size_t first, size_t size, int target, bool& found, size_t& location)
{
    size_t middle;
    if(size == 0) found = false;
    else {
        middle = first + size/2;
        if(target == a[middle]){
            location = middle;
            found = true;
        }
        else if (target < a[middle])
            // target is less than middle, so search subarray before middle
            search(a, first, size/2, target, found, location);
        else
            // target is greater than middle, so search subarray after middle
            search(a, middle+1, (size-1)/2, target, found, location);
    }
}
```

Relation to Binary Search Tree

Array of previous example:

3	6	7	11	32	33	53

Corresponding complete binary search tree

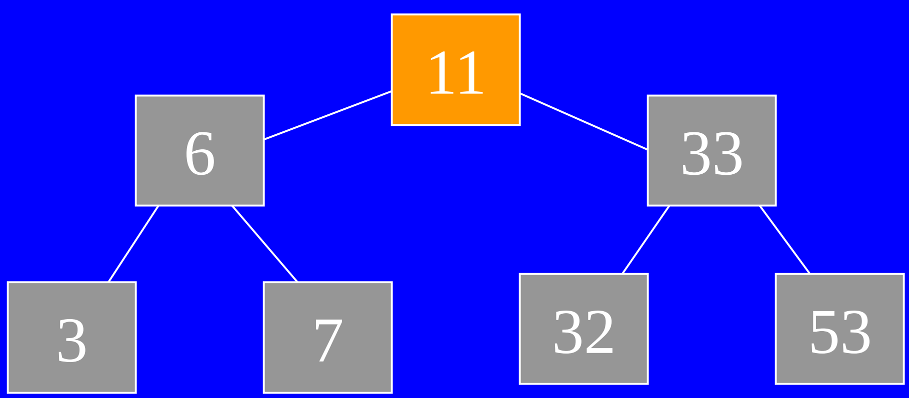


Search for target = 7

Find midpoint:

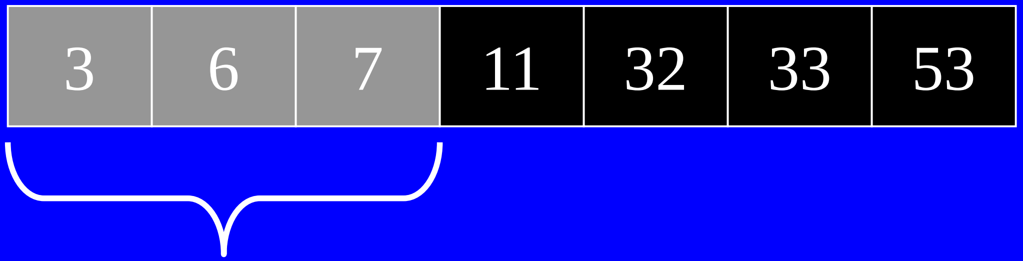
3	6	7	11	32	33	53

Start at root:

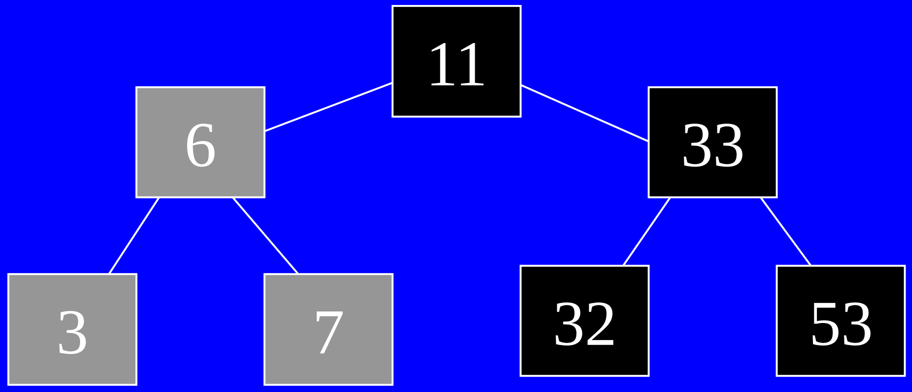


Search for target = 7

Search left subarray:

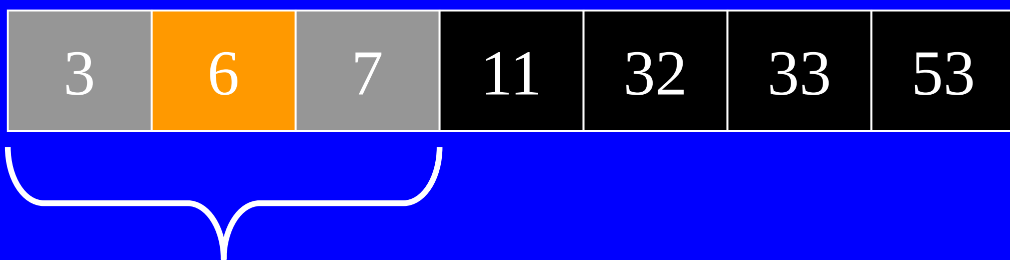


Search left subtree:

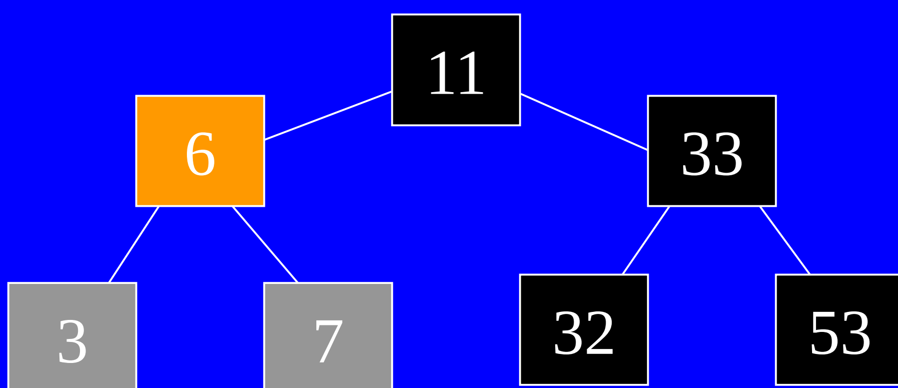


Search for target = 7

Find approximate midpoint of subarray:



Visit root of subtree:

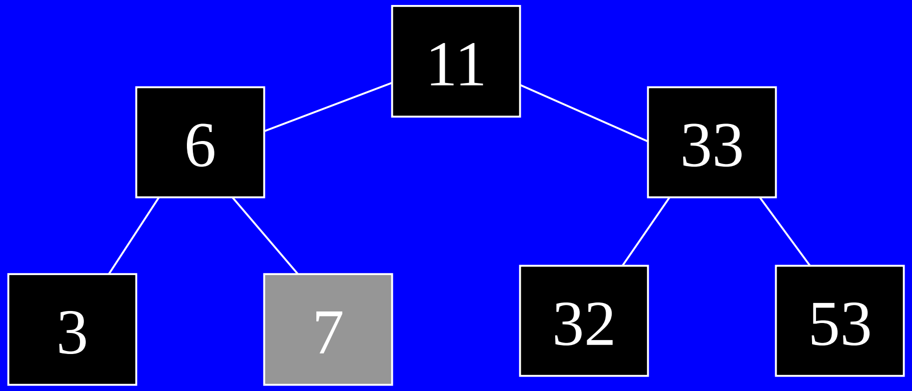


Search for target = 7

Search right subarray:

3	6	7	11	32	33	53

Search right subtree:



Binary Search: Analysis

- Worst case complexity?
- What is the maximum depth of recursive calls in binary search as function of n?
- Each level in the recursion, we split the array in half (divide by two).
- Therefore maximum recursion depth is floor($\log_2 n$) and worst case = $O(\log_2 n)$.
- Average case is also = $O(\log_2 n)$.

Can we do better than $O(\log_2 n)$?

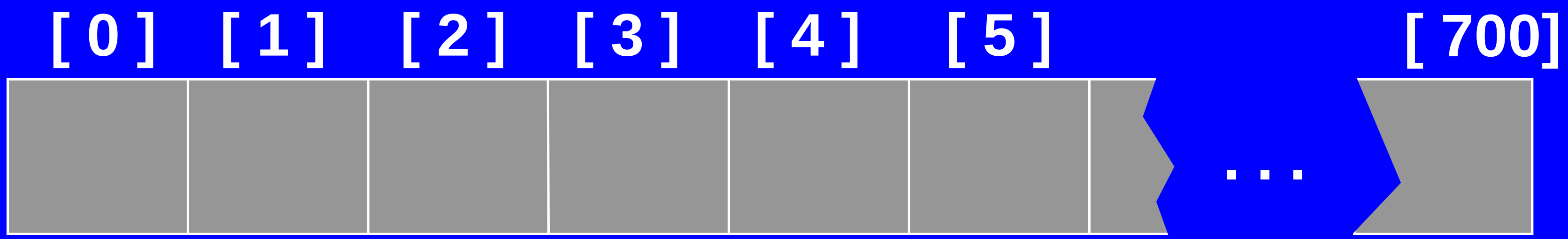
- Average and worst case of serial search = $O(n)$
- Average and worst case of binary search = $O(\log_2 n)$

- Can we do better than this?

YES. Use a hash table!

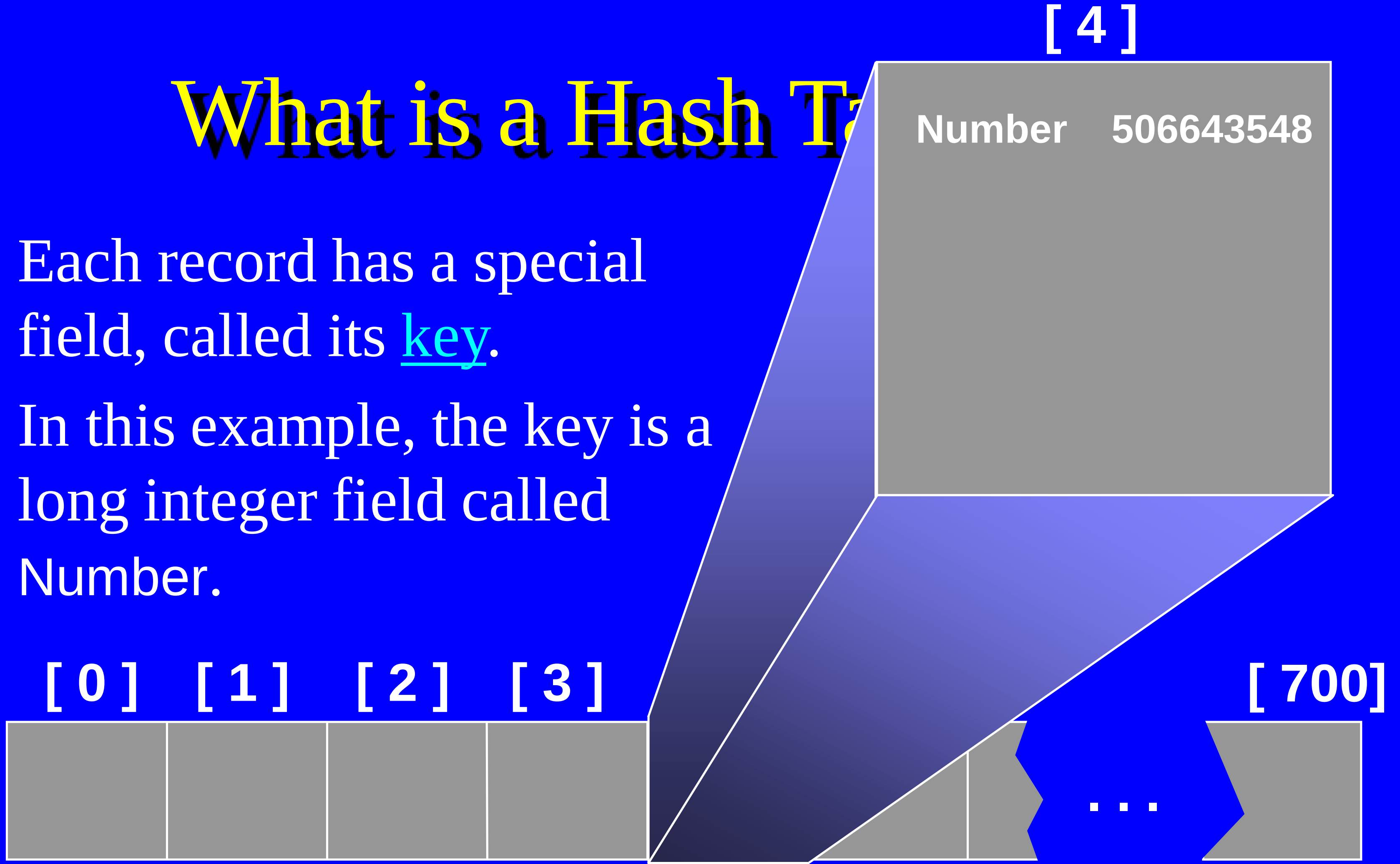
What is a Hash Table ?

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



What is a Hash Ta

- 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 Ta

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

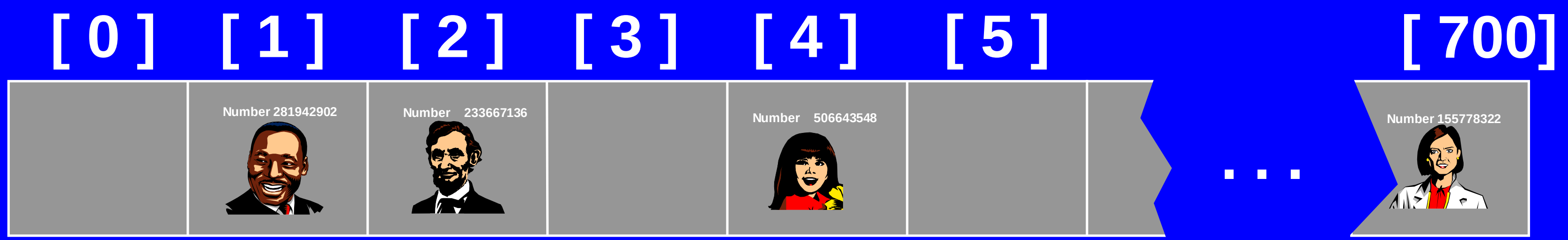
[4]

Number 506643548

[0] [1] [2] [3] . . . [700]

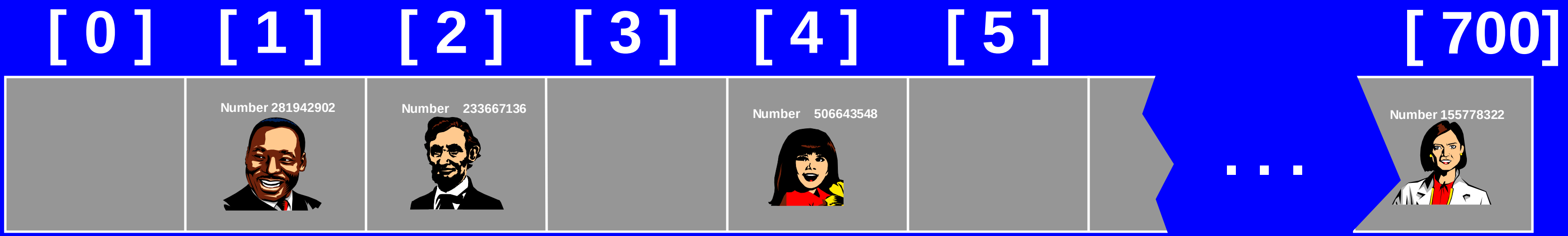
What is a Hash Table ?

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



Open Address Hashing

- 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.

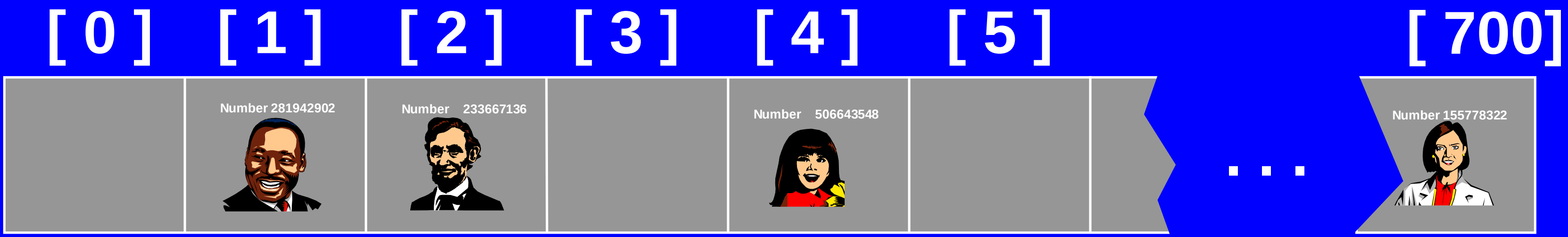


Inserting a New Record

- Typical way create a hash value:

  (Number mod 701)

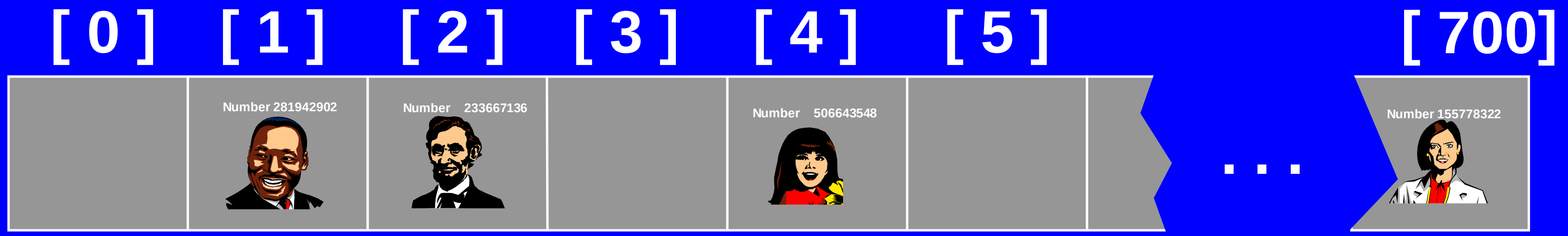
What is (580625685 % 701) ?



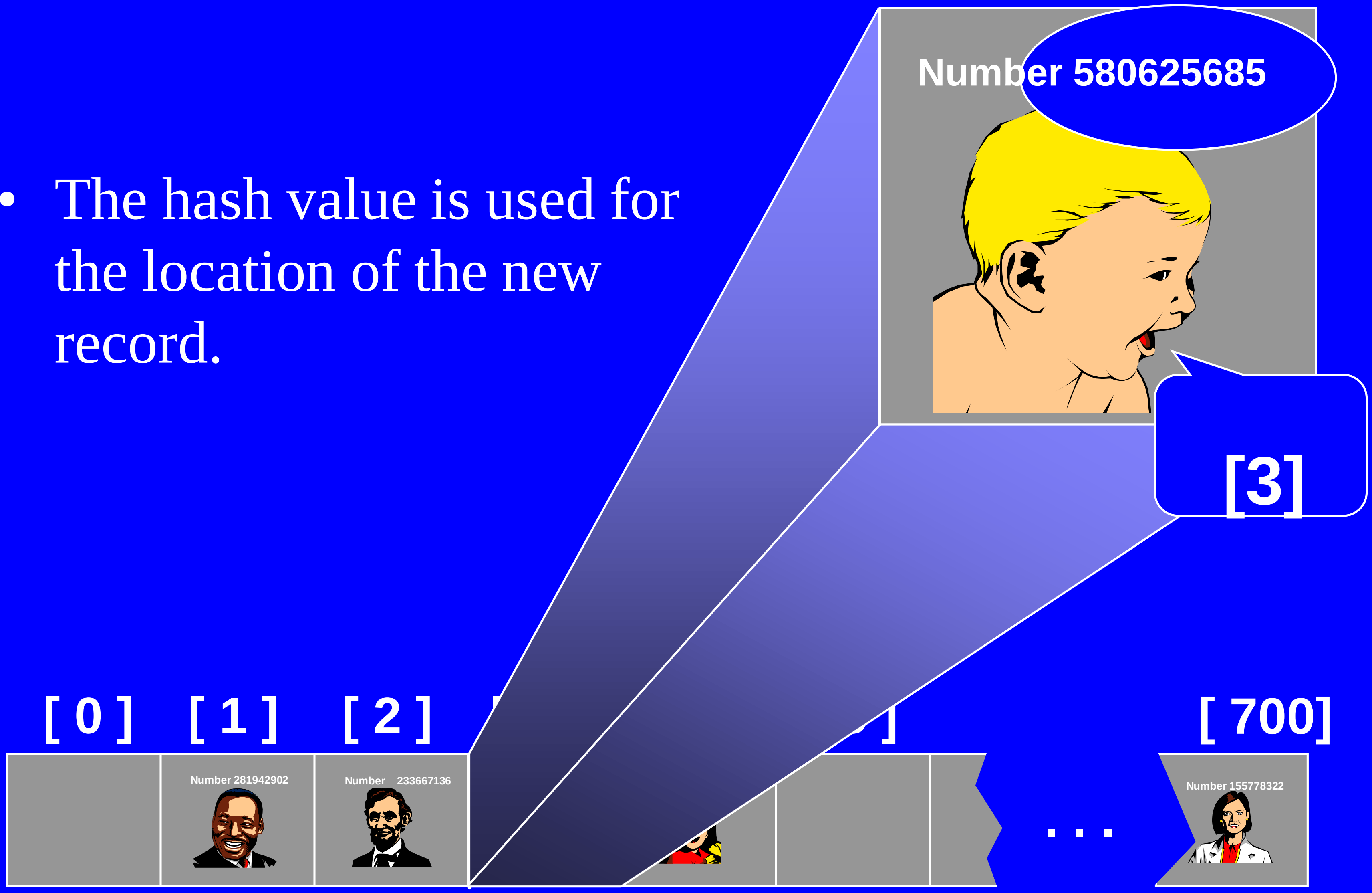
- Typical way to create a hash value:

  (Number mod 701)

What is (580625685 % 701) ?

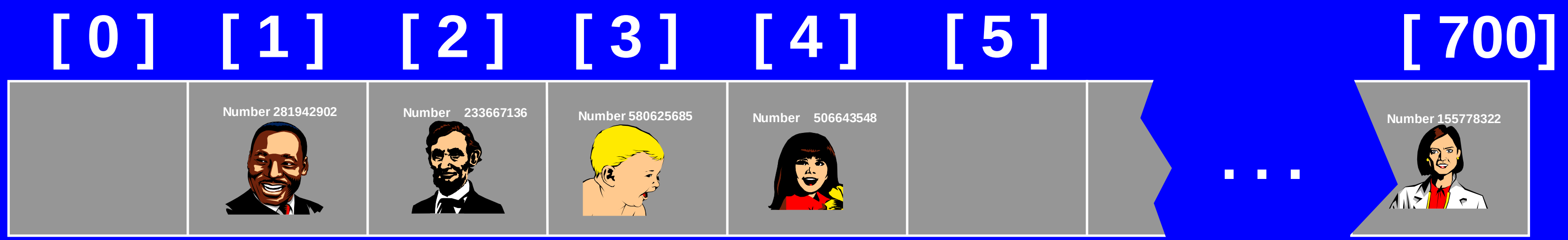


- The hash value is used for the location of the new record.



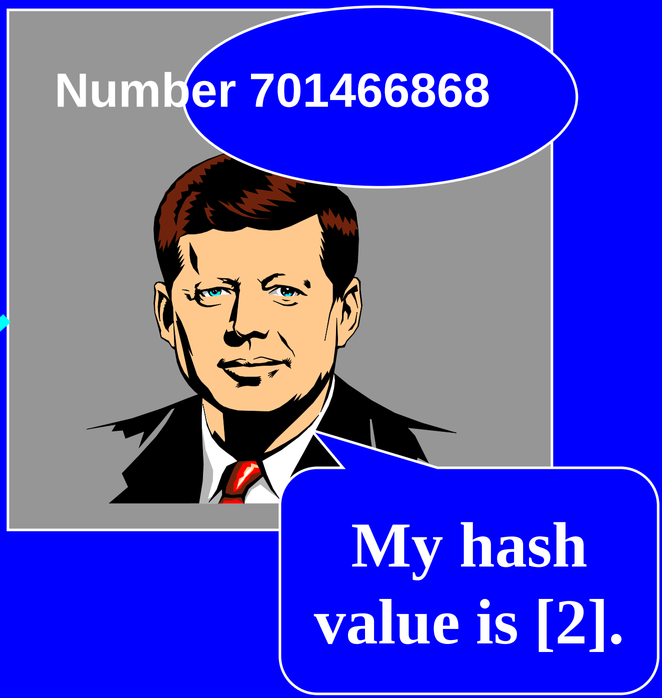
Inserting a New Record

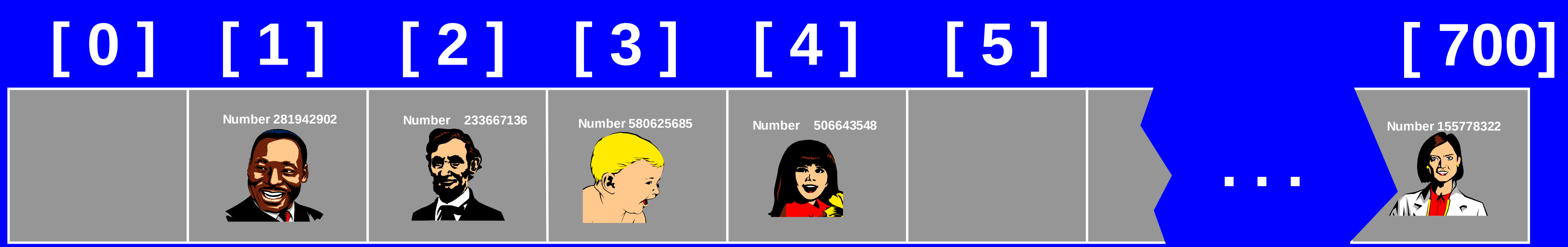
- The hash value is used for the location of the new record.



Collisions

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

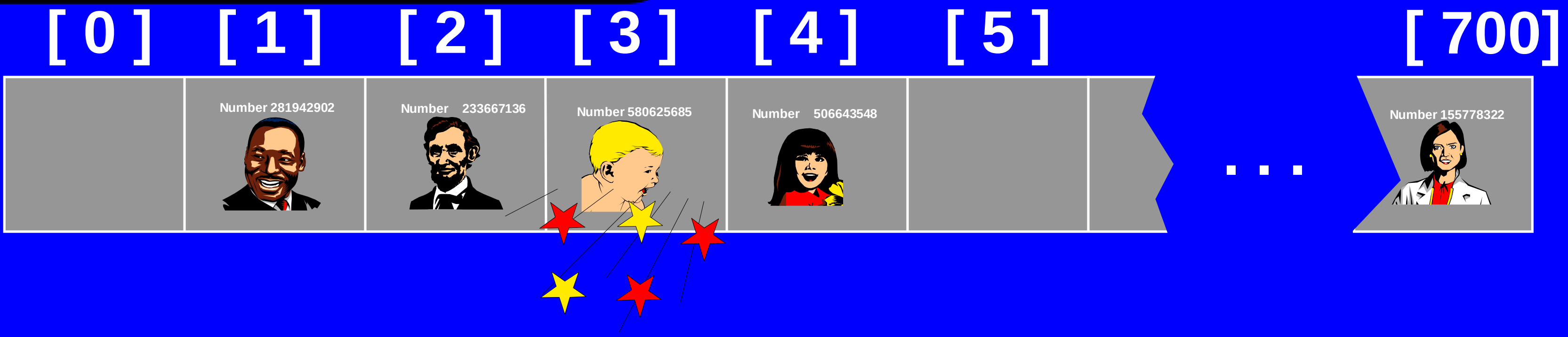




Collisions

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

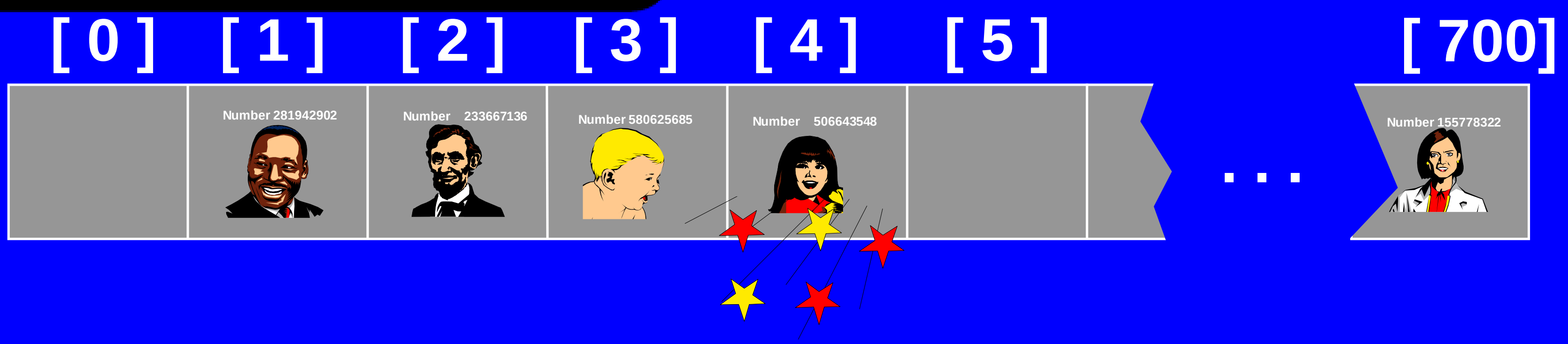
When a collision occurs, move forward until you find an empty spot.



Collisions

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

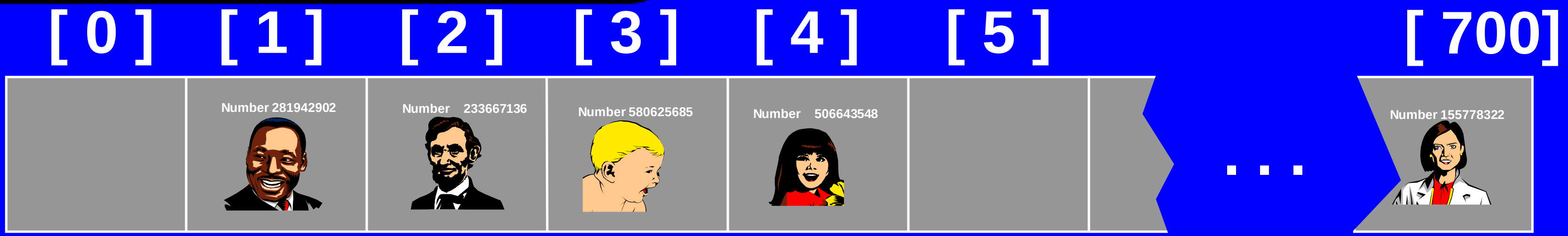
When a collision occurs, move forward until you find an empty spot.



Collisions

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

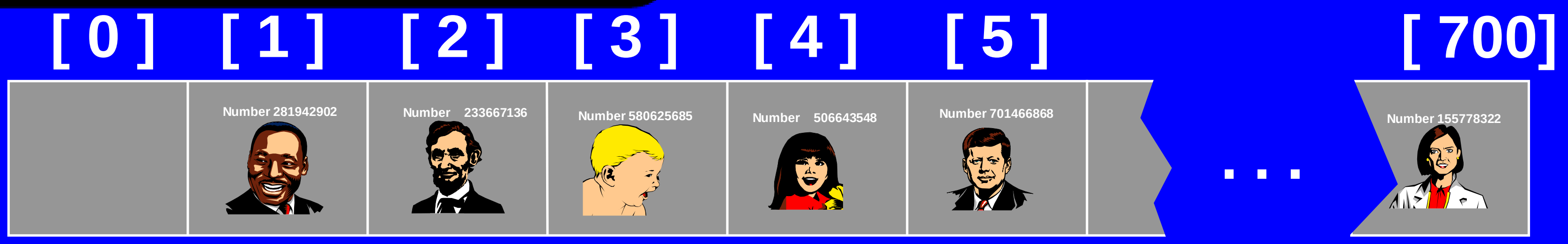
When a collision occurs, move forward until you find an empty spot.



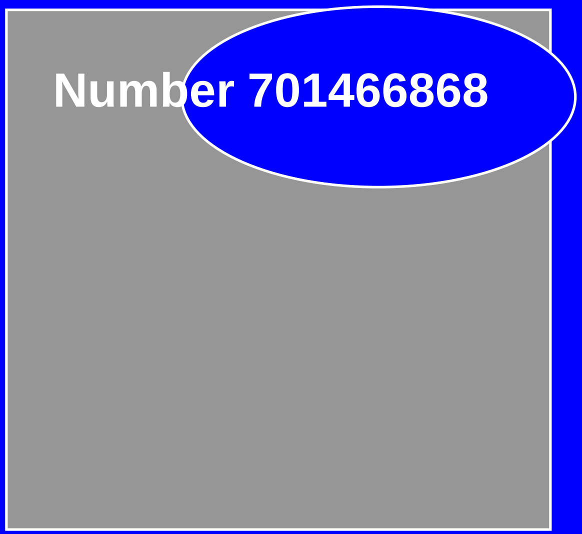
Collisions

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

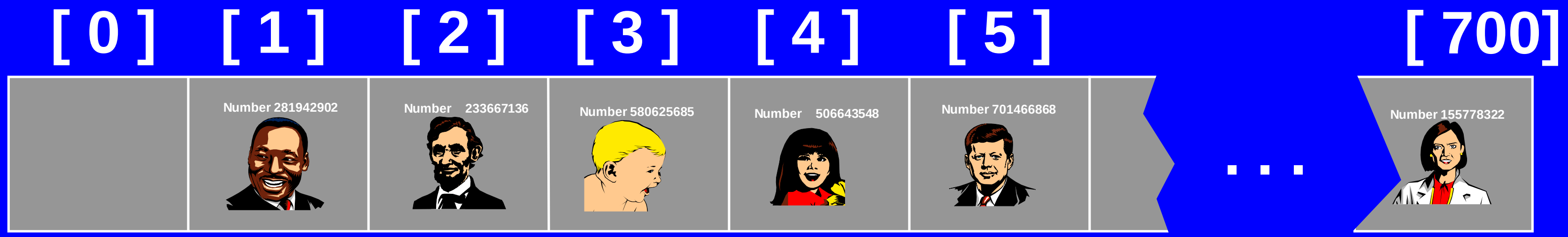
The new record goes in the empty spot.



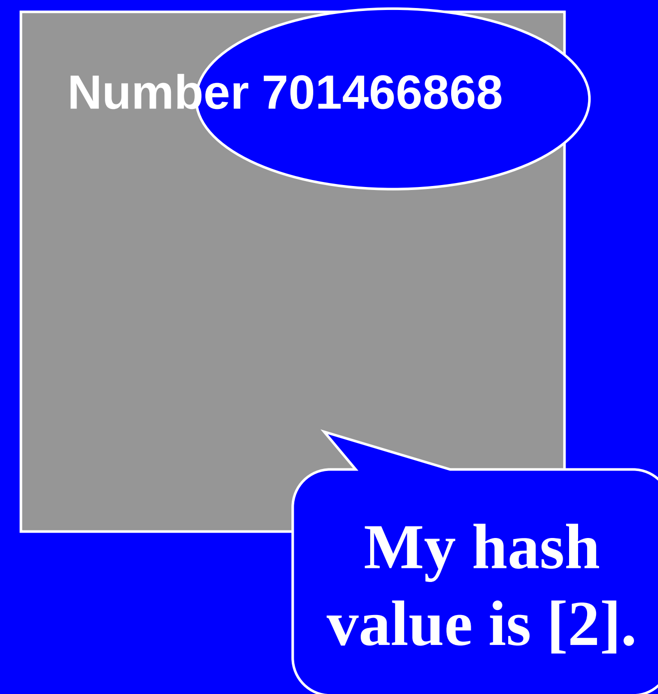
Searching for a Key

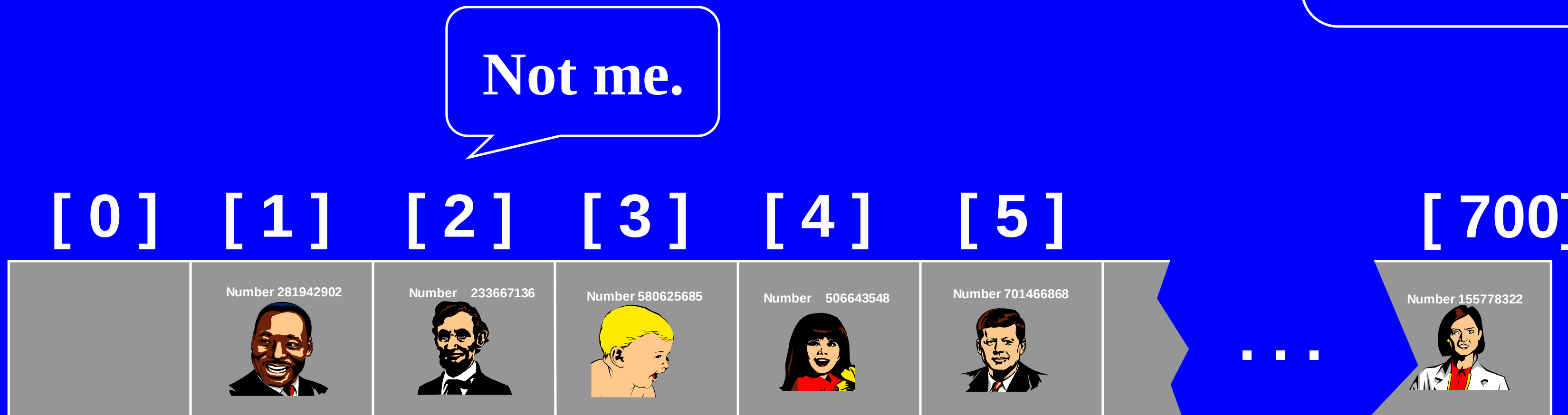


- The data that's attached to a key can be found fairly quickly.

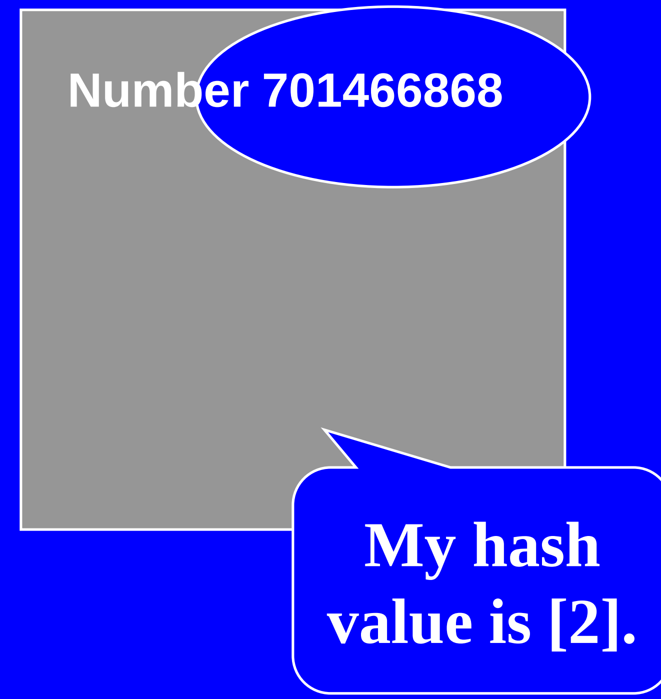


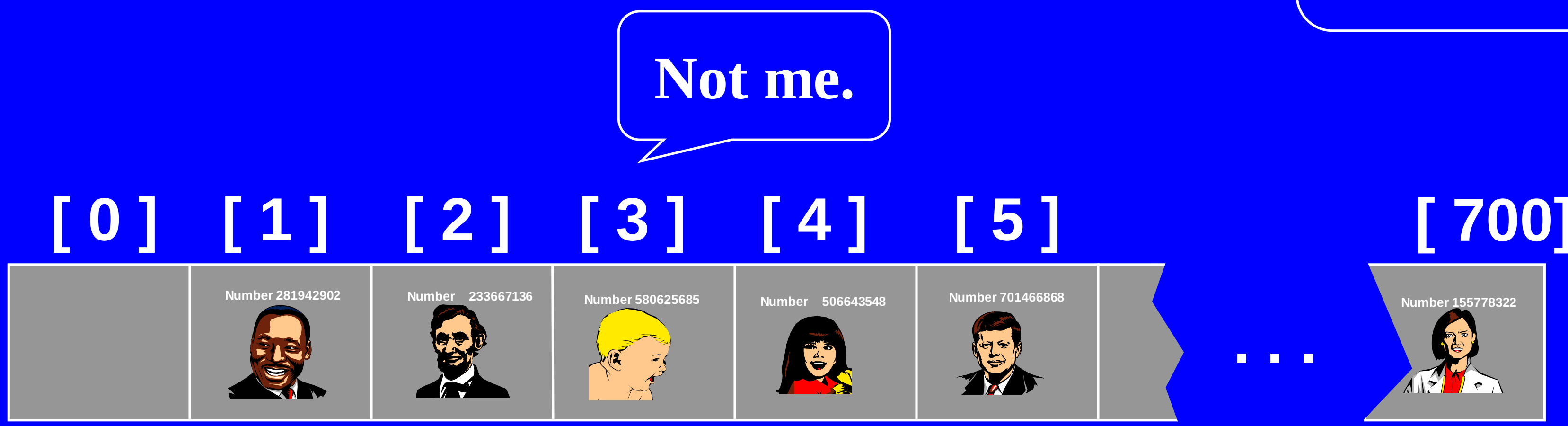
- Calculate the hash value.
- Check that location of the array for the key.



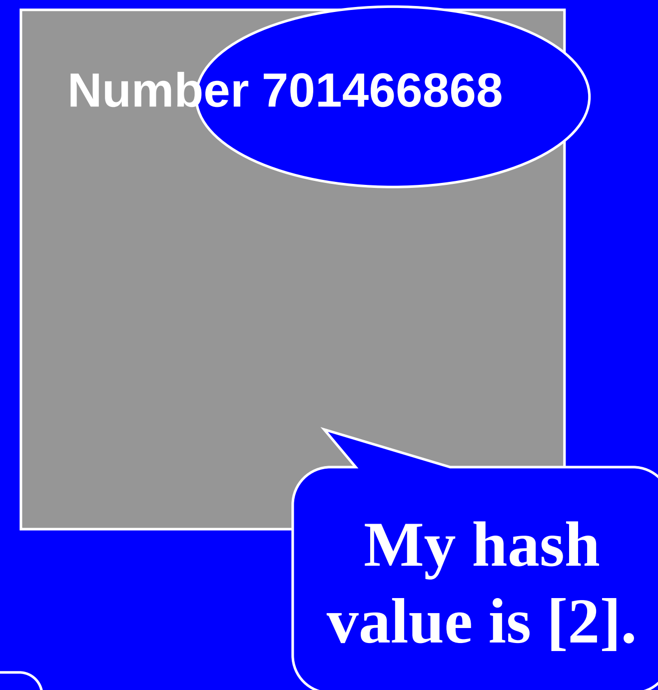


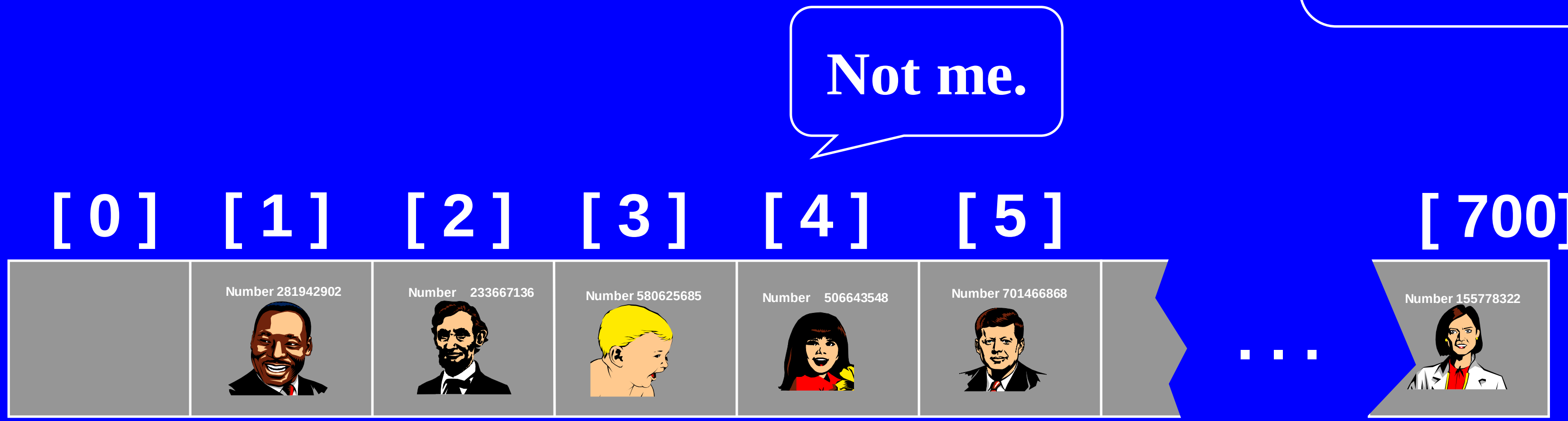
- Keep moving forward until you find the key, or you reach an empty spot.

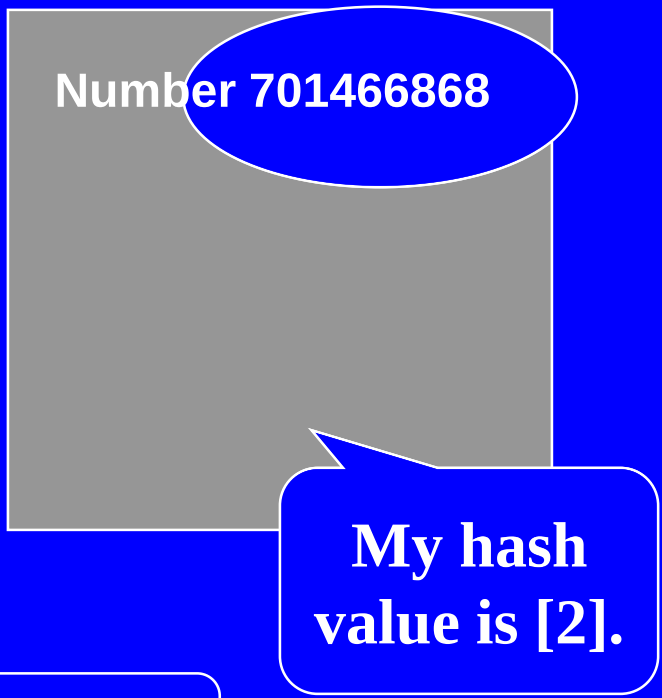




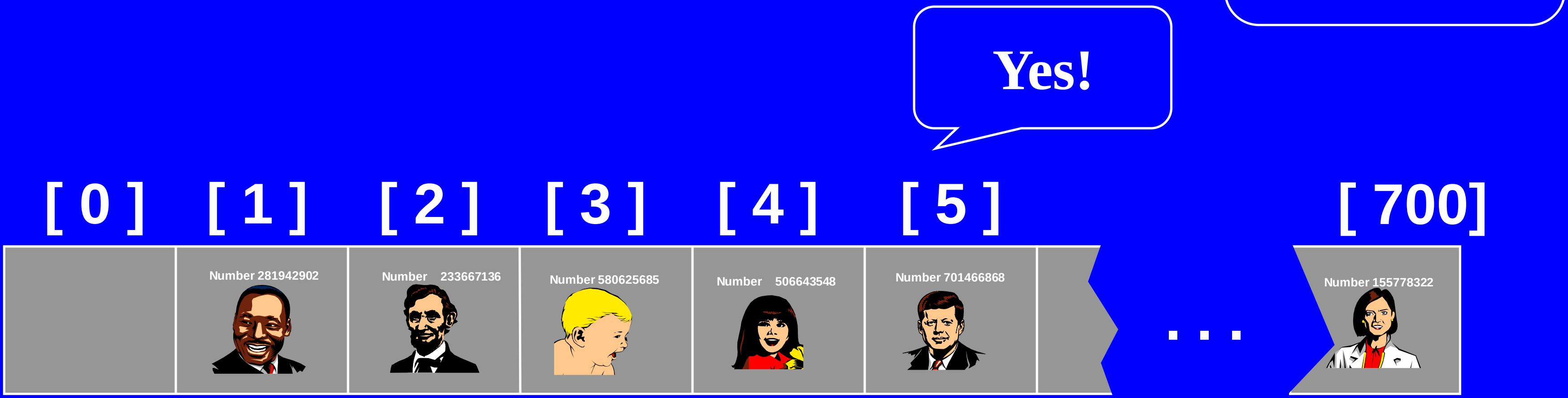
- Keep moving forward until you find the key, or you reach an empty spot.



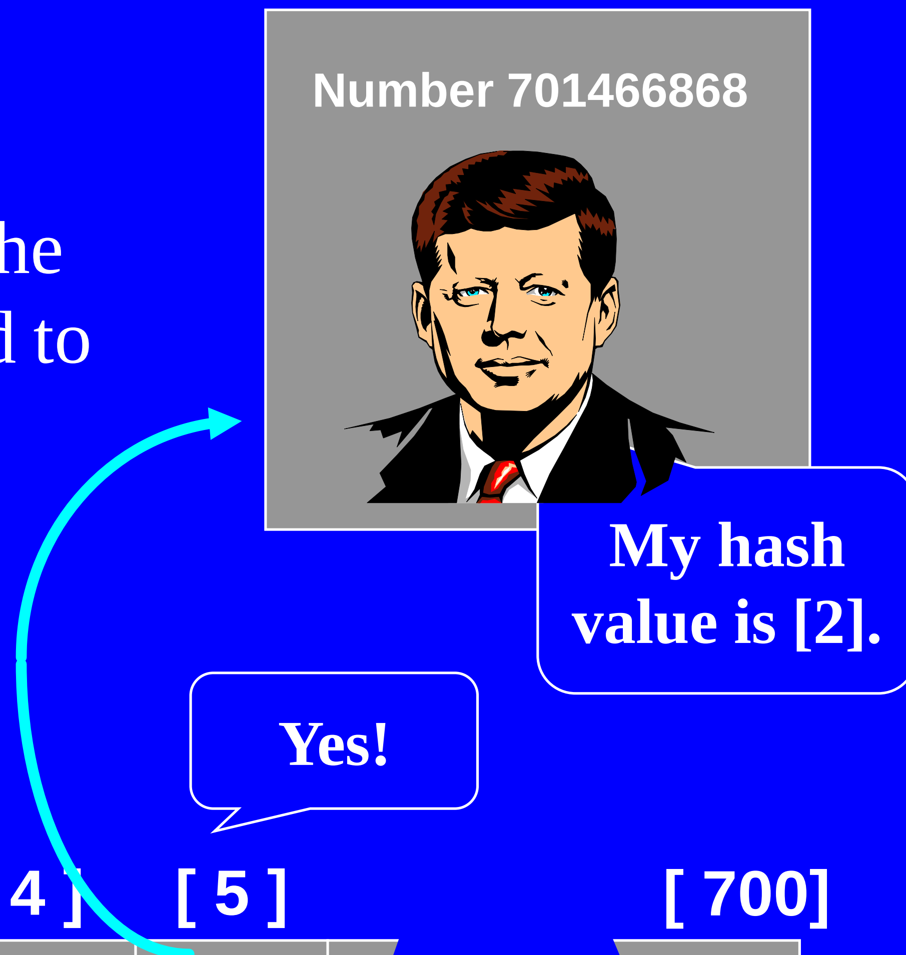


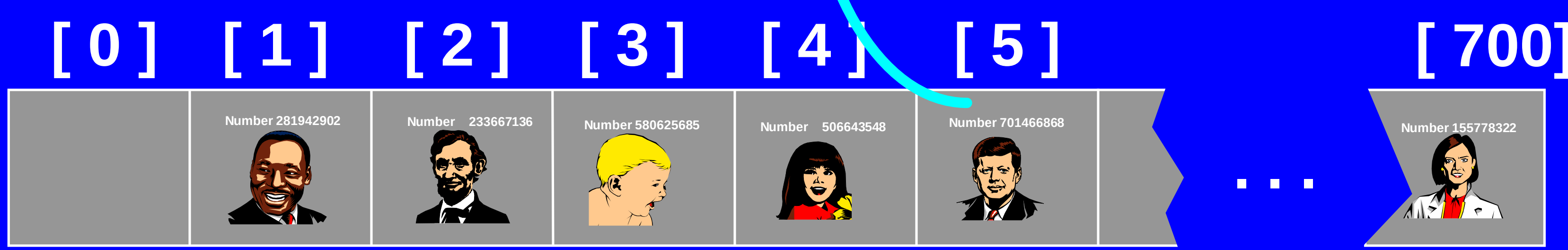


- Keep moving forward until you find the key, or you reach an empty spot.



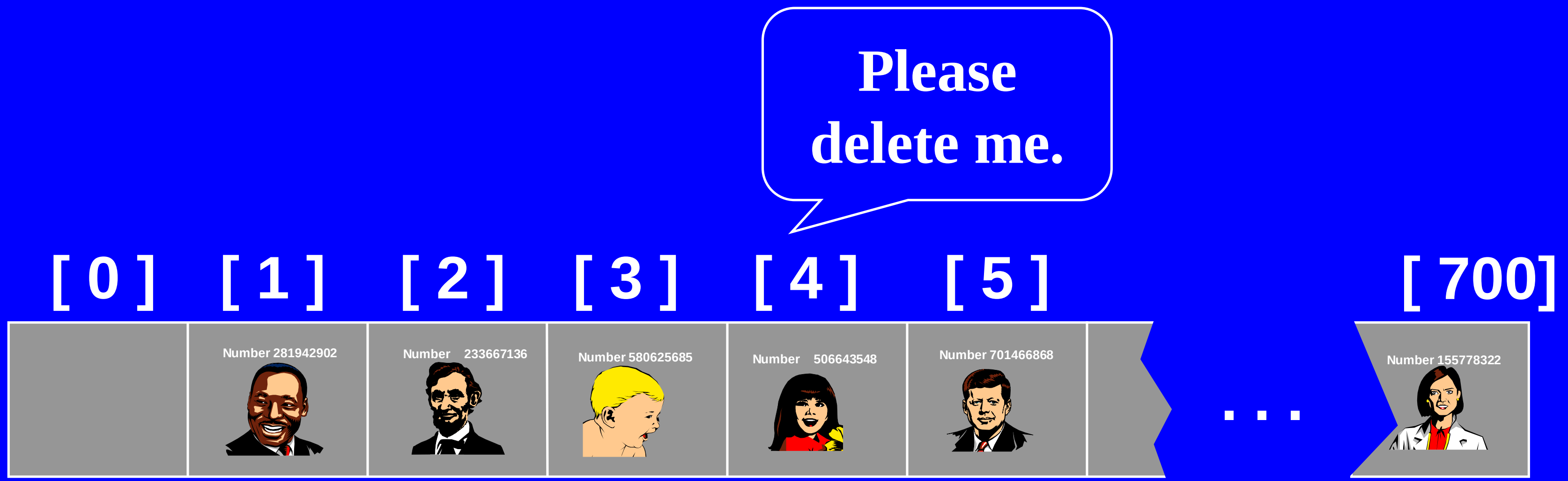
- When the item is found, the information can be copied to the necessary location.





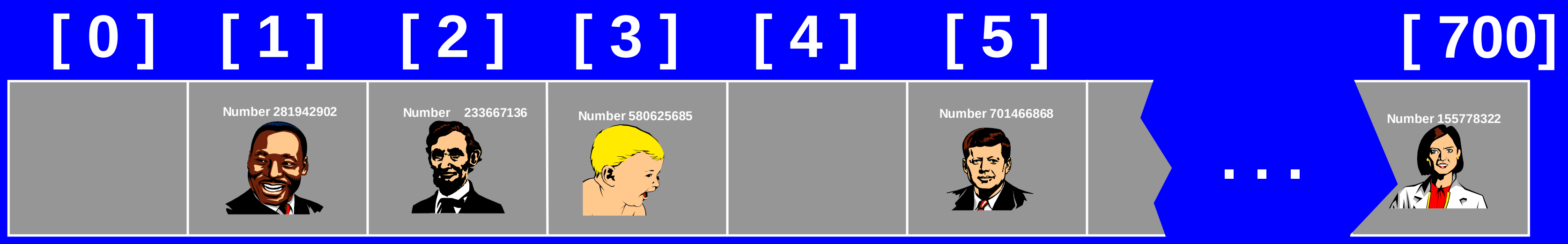
Deleting a Record

- Records may also be deleted from a hash table.



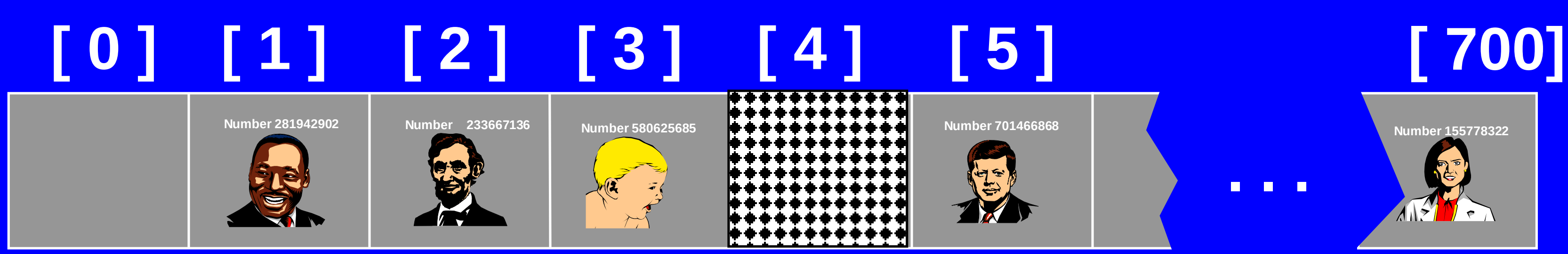
Deleting a Record

- 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

- Hash tables store a collection of records with keys.
- The location of a record depends on the hash value of the record's key.
- Open address hashing:
  - When a collision occurs, the next available location is used.
  - Searching for a particular key is generally quick.
  - When an item is deleted, the location must be marked in a special way, so that the searches know that the spot used to be used.
- See text for implementation.

Open Address Hashing

- To reduce collisions…
  - Use table CAPACITY = prime number of form 4k+3
  - Hashing functions:
    - Division hash function: key % CAPACITY
    - Mid-square function: (key*key) % CAPACITY
    - Multiplicative hash function: key is multiplied by positive constant less than one. Hash function returns first few digits of fractional result.

Clustering

- In the hash method described, when the insertion encounters a collision, we move forward in the table until a vacant spot is found. This is called *linear probing*.
- *Problem:* when several different keys are hashed to the same location, adjacent spots in the table will be filled. This leads to the problem of *clustering*.
- As the table approaches its capacity, these clusters tend to merge. This causes insertion to take a long time (due to linear probing to find vacant spot).

Double Hashing

- One common technique to avoid cluster is called *double hashing*.
- Let's call the original hash function *hash1*
- Define a second hash function *hash2*

Double hashing algorithm:

1. *When an item is inserted, use hash1(key) to determine insertion location i in array as before.*
2. *If collision occurs, use hash2(key) to determine how far to move forward in the array looking for a vacant spot:*

next location = (i + hash2(key)) % CAPACITY

Double Hashing

- Clustering tends to be reduced, because hash2() has different values for keys that initially map to the same initial location via hash1().
- This is in contrast to hashing with *linear probing*.
- Both methods are *open address hashing*, because the methods take the next open spot in the array.
- In linear probing

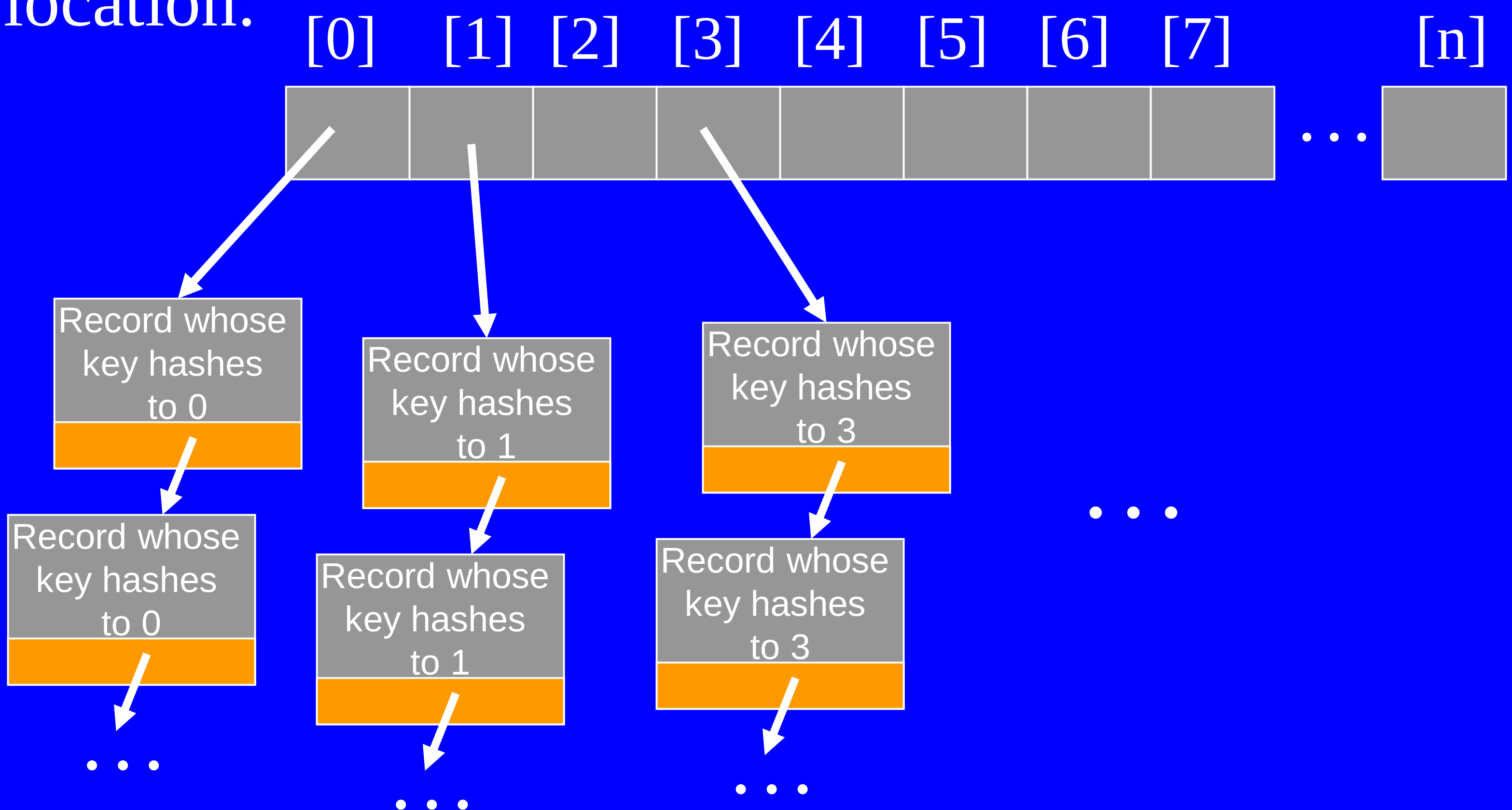
  hash2(key) = (i+1)%CAPACITY
- In double hashing hash2() can be a general function of the form
  - hash2(key) = (I+f(key))%CAPACITY

Chained Hashing

- In open address hashing, a collision is handled by probing the array for the next vacant spot.
- When the array is full, no new items can be added.
- We can solve this by resizing the table.
- Alternative: chained hashing.

Chained Hashing

- In chained hashing, each location in the hash table contains a list of records whose keys map to that location:



Time Analysis of Hashing

- Worst case: every key gets hashed to same array index! O(n) search!!
- Luckily, average case is more promising.
- First we define a fraction called the hash table *load factor:*

$$\alpha = \frac{\textit{number of occupied table locations}}{\textit{size of table's array}}$$

Average Search Times

For open addressing with linear probing, average number of table elements examined in a successful search is approximately:

$$\tfrac{1}{2}\left(1+ 1/(1-\alpha)\right)$$

Double hashing: $-\ln(1-\alpha)/\alpha$

Chained hashing: $1+\alpha/2$

Average number of table elements examined during successful search

Load factor(α)	Open addressing, linear probing $\frac{1}{2}(1+1/(1-\alpha))$	Open addressing double hashing $-\ln(1-\alpha)/\alpha$	Chained hashing $1+\alpha/2$
0.5	1.50	1.39	1.25
0.6	1.75	1.53	1.30
0.7	2.17	1.72	1.35
0.8	3.00	2.01	1.40
0.9	5.50	2.56	1.45
1.0	Not applicable	Not applicable	1.50
2.0	Not applicable	Not applicable	2.00
3.0	Not applicable	Not applicable	2.50

Summary

- Serial search: average case O(n)
- Binary search: average case $O(\log_2 n)$
- Hashing
  - Open address hashing
    - Linear probing
    - Double hashing
  - Chained hashing
  - Average number of elements examined is function of load factor α.