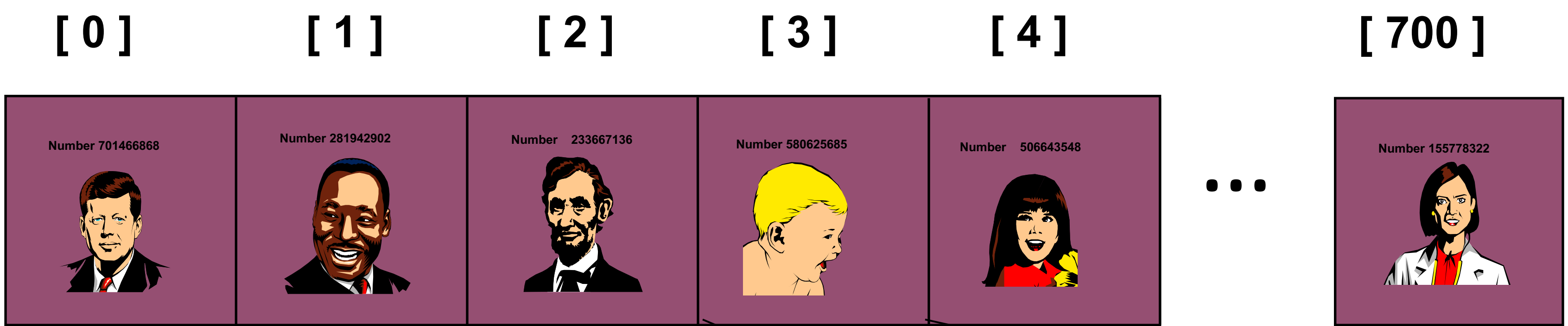
Sequential Search

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?

Sequential 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 Sequential Search

```
// Search for a desired item in the n array elements
// Returns desired record if found.
// Otherwise, return NULL
for(int i = 0; i < n; i++ ) {
        if(a[i] is desired item) {
                return a[i];
        }
}
return NULL;
```

Time Complexity

- 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 Complexity of Sequential 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 Time Complexity of Sequential 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 Complexity of Sequential Search

Generalize for array size *n.*

Expression for average-case running time:

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

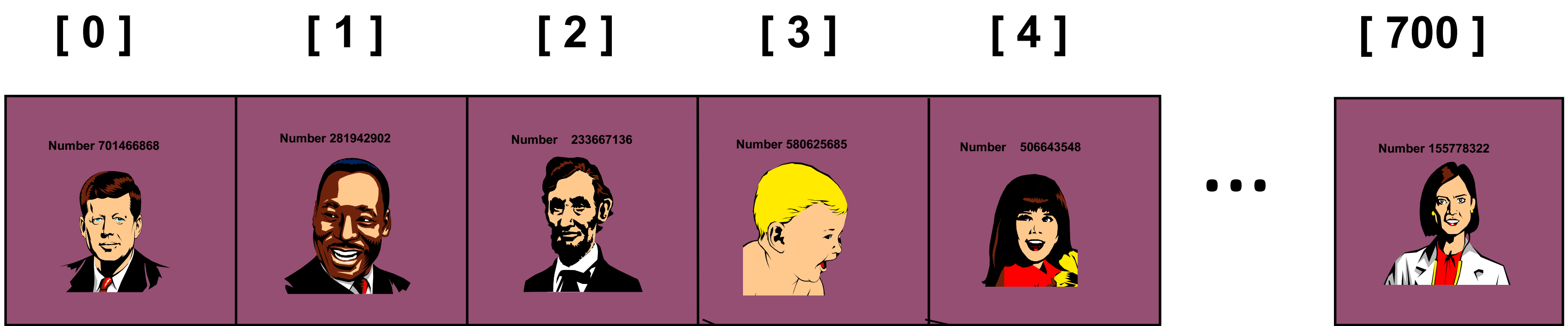
Therefore, average case time complexity for serial search is $O(n)$.

Binary Search

Problem: Search

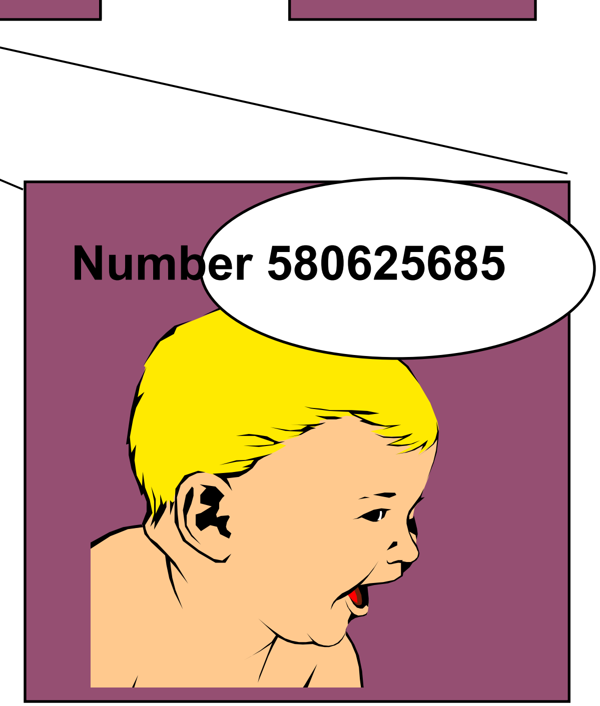
- We are given a list of n records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.

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?



Binary Search

- Idea: Eliminates one half of the elements after each comparison.
  - Locate the middle of the array
  - Compare the value at that location with the search key.
  - If they are equal - done!
  - Otherwise, decide which half of the array contains the search key.
  - Repeat the search on that half of the array and ignore the other half.
  - The search continues until the key is matched or no elements remain to be searched.

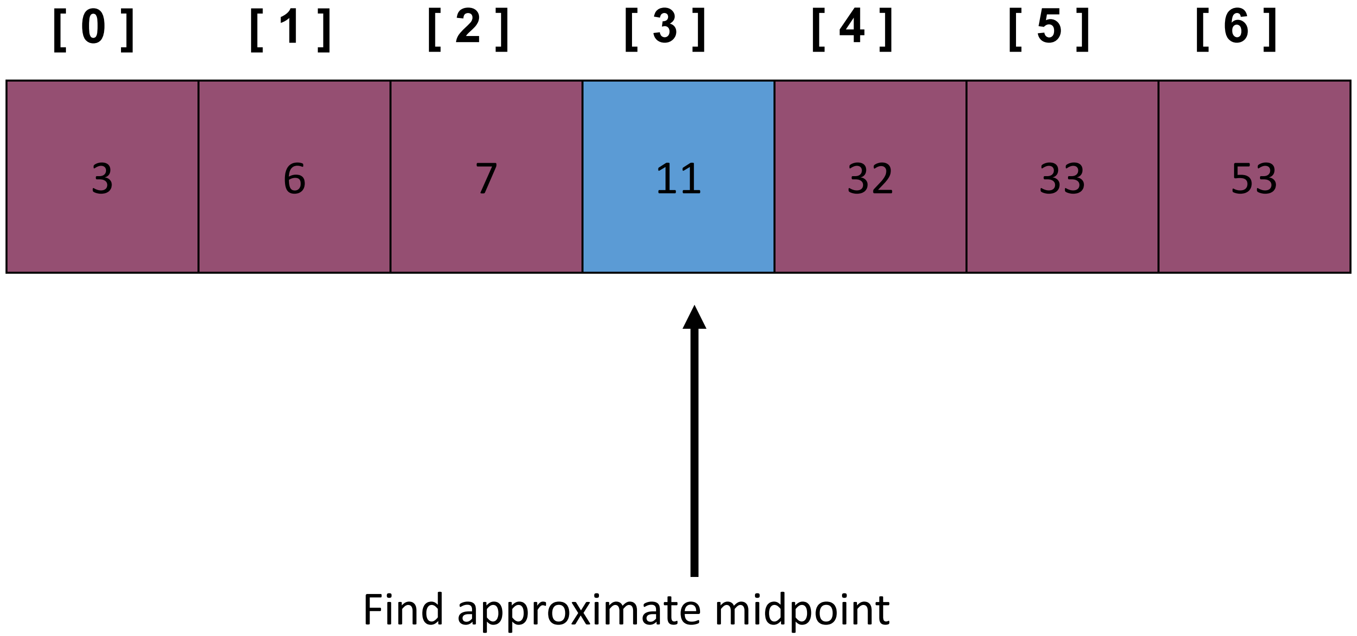
Binary Search

Example: sorted array of integer keys. Search target=7.

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

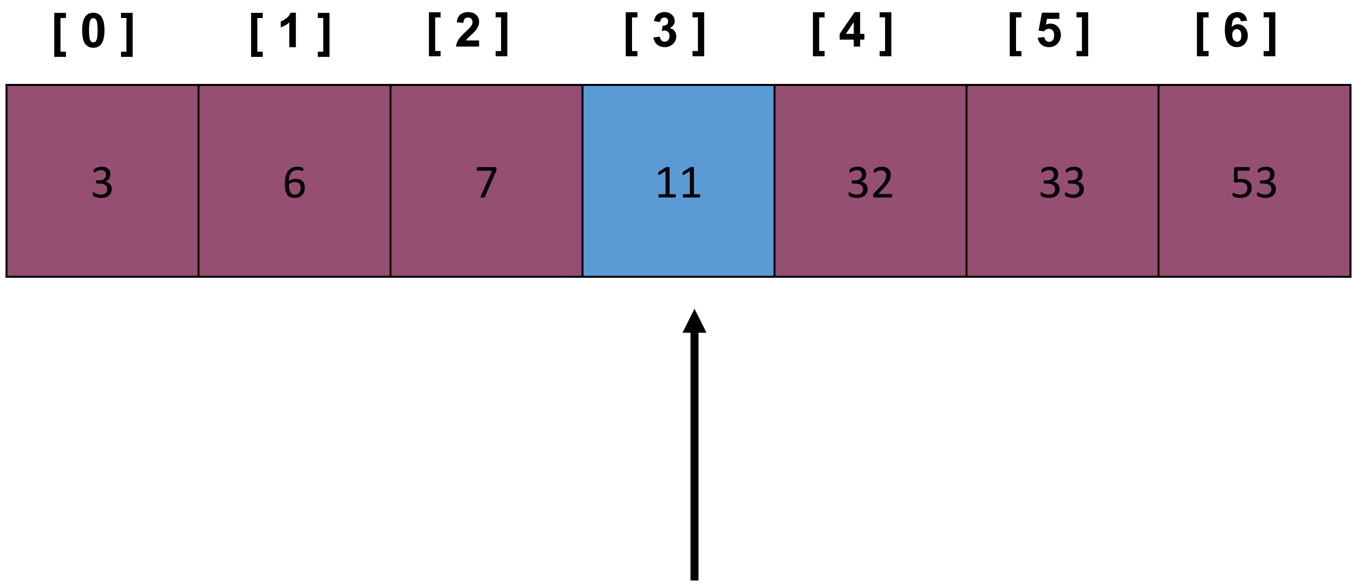
Binary Search

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



Binary Search

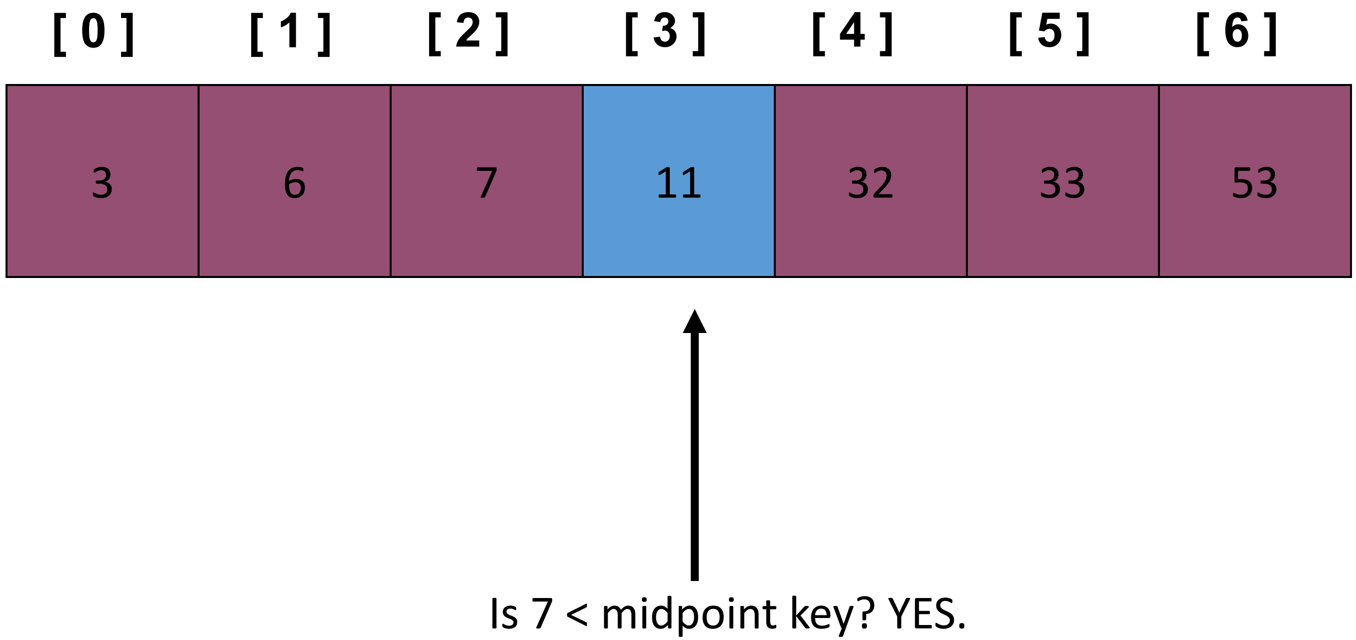
Example: sorted array of integer keys. Target=7.



Is 7 = midpoint key? NO.

Binary Search

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



Binary Search

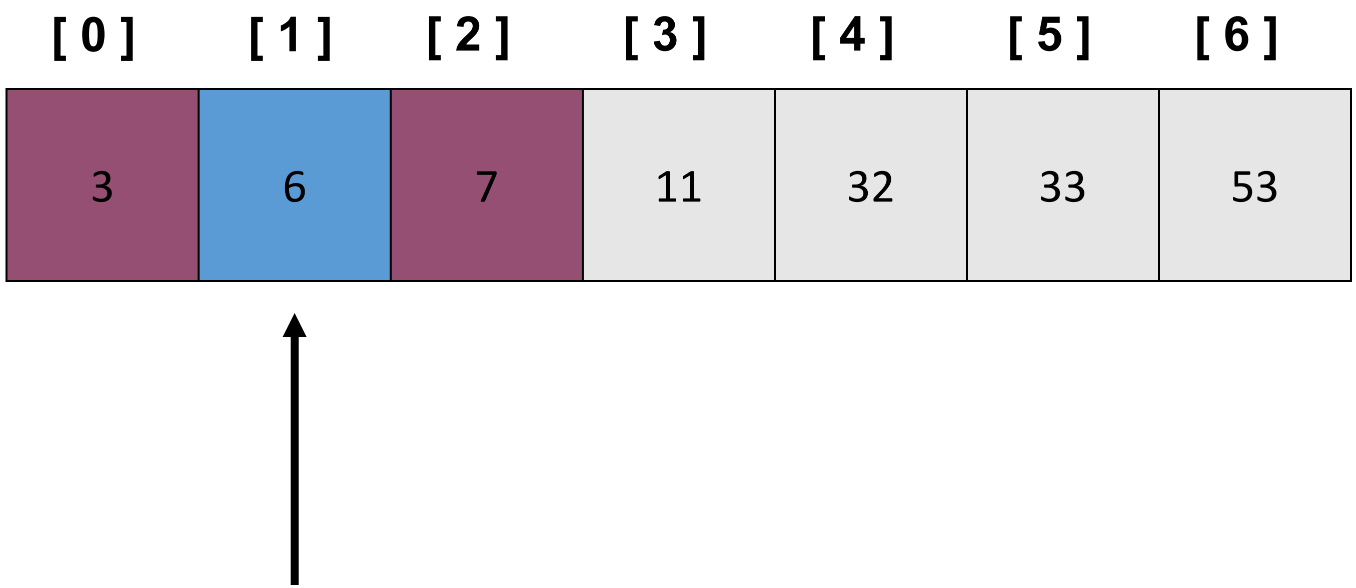
Example: sorted array of integer keys. Target=7.

[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.



Find approximate midpoint

Binary Search

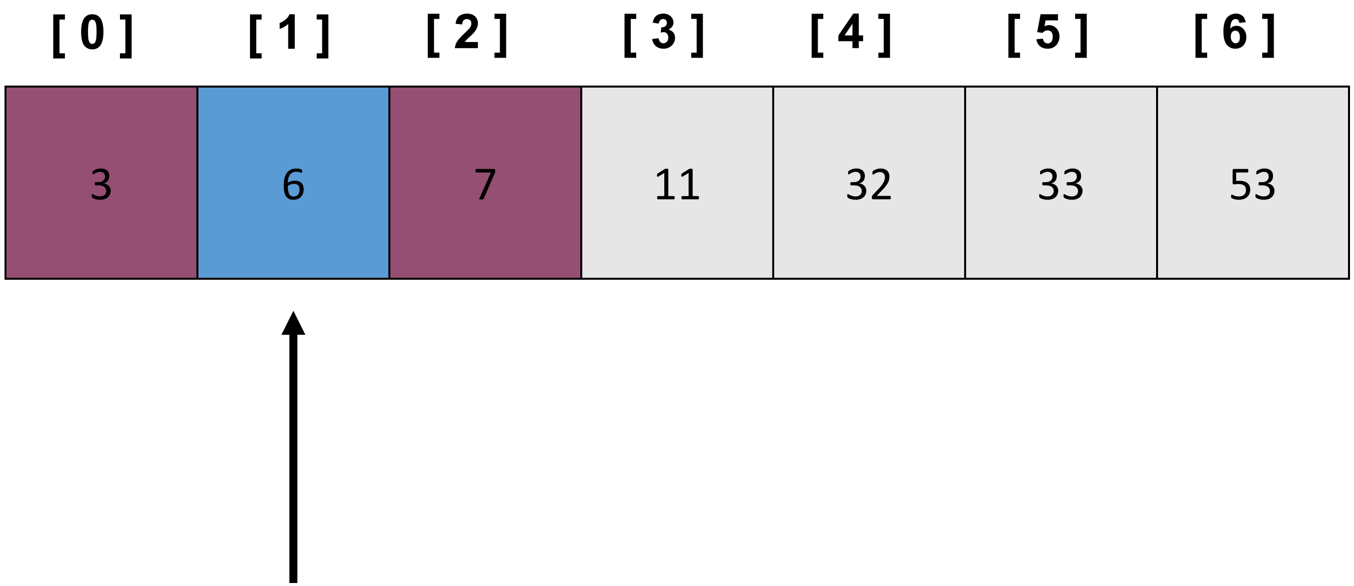
Example: sorted array of integer keys. Target=7.

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

Target = key of midpoint? NO.

Binary Search

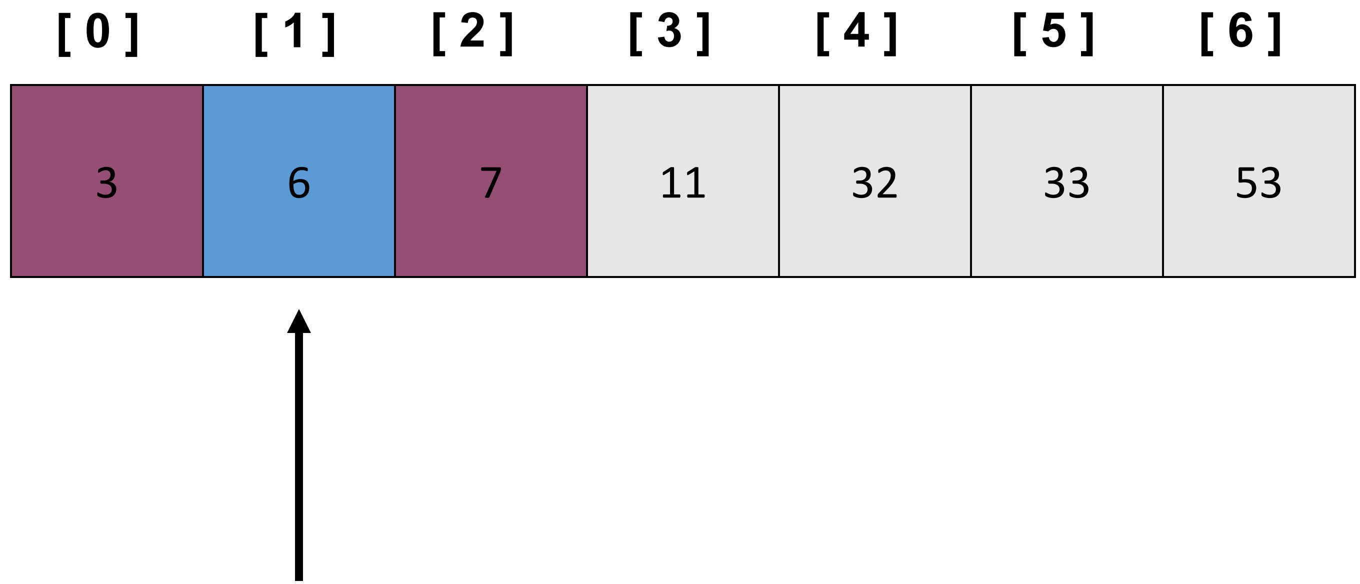
Example: sorted array of integer keys. Target=7.



Target < key of midpoint? NO.

Binary Search

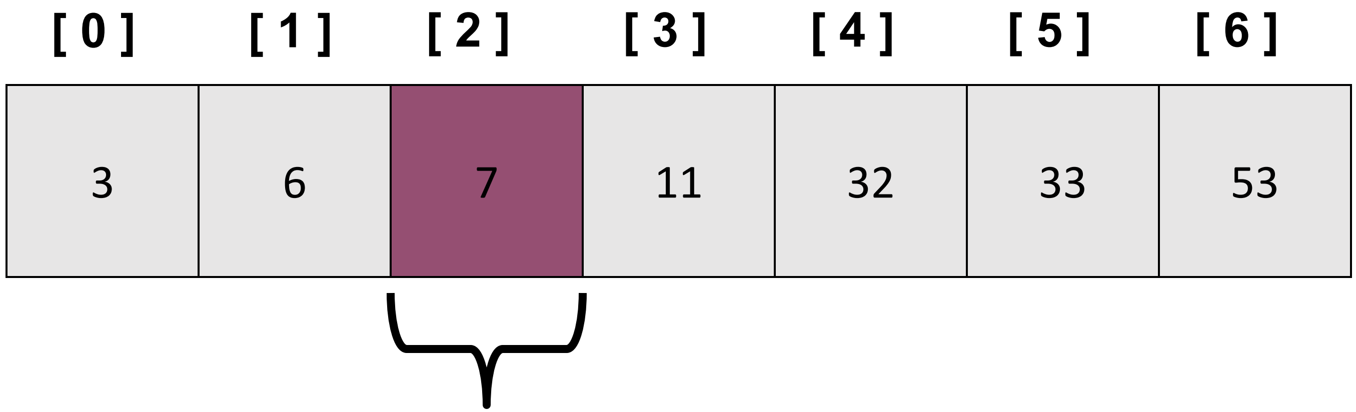
Example: sorted array of integer keys. Target=7.



Target > key of midpoint? YES.

Binary Search

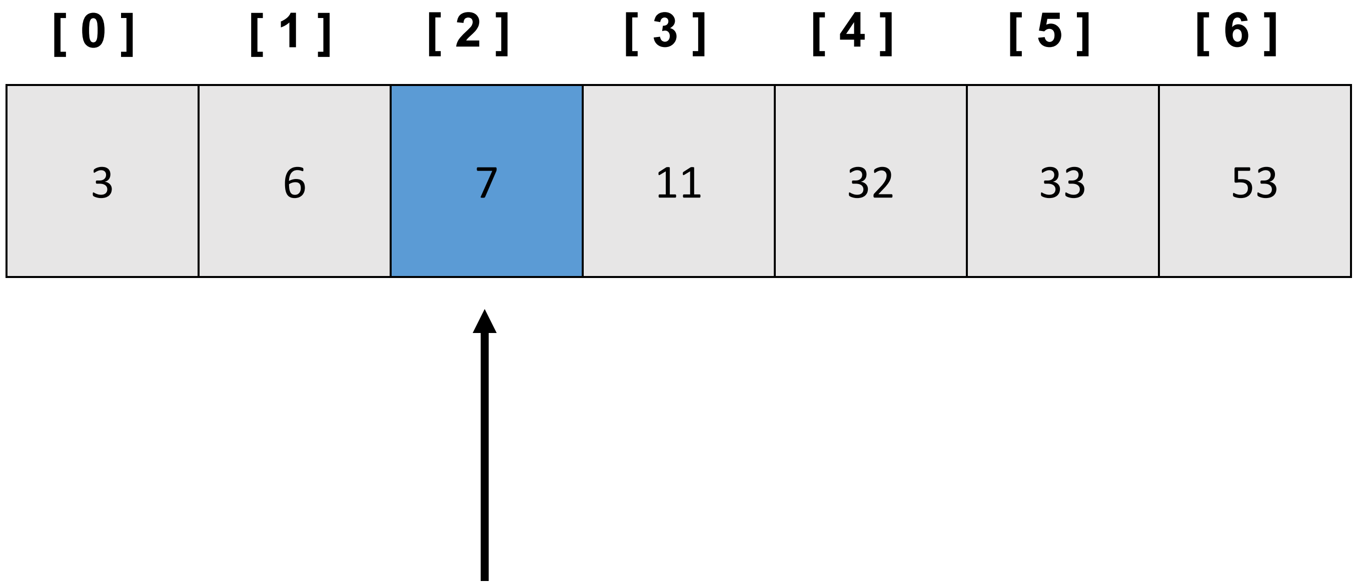
Example: sorted array of integer keys. Target=7.



Search for the target in the area after midpoint.

Binary Search

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



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

Exercise:

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.

- Ex. Binary search for 33.

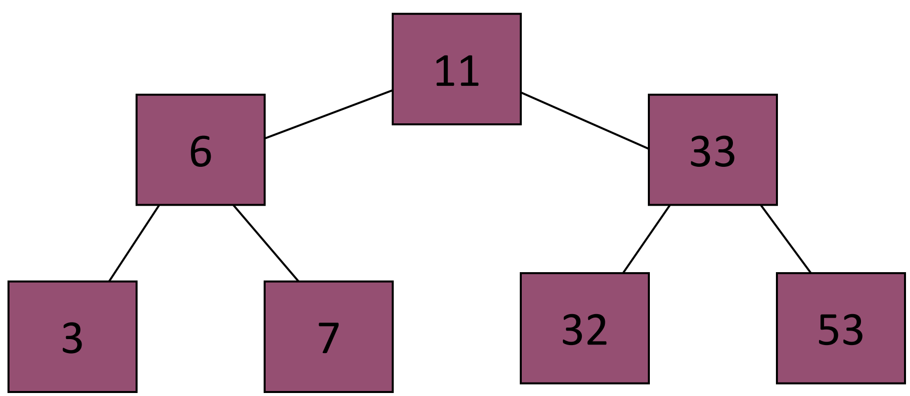
6	**13**	**14**	**25**	**33**	**43**	**51**	**53**	**64**	**72**	**84**	**93**	**95**	**96**	**97**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														**hi**

Relation to Binary Search Tree

Array a:

3	6	7	11	32	33	53

Corresponding complete binary search tree

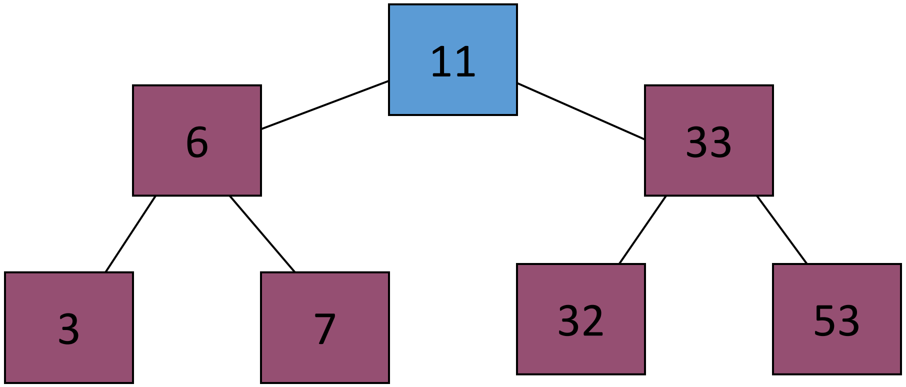


Search for target = 7

Find midpoint:

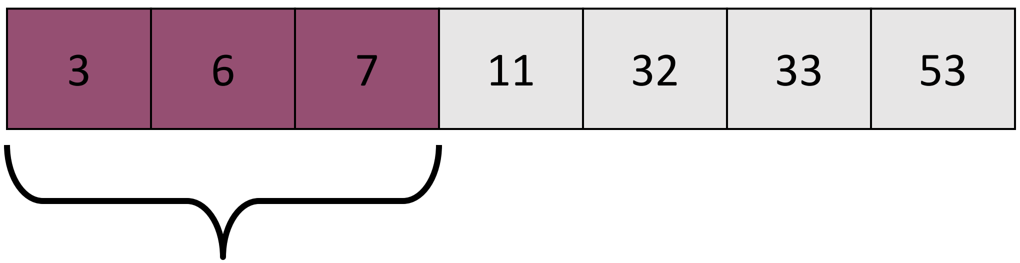
3	6	7	11	32	33	53

Start at root:

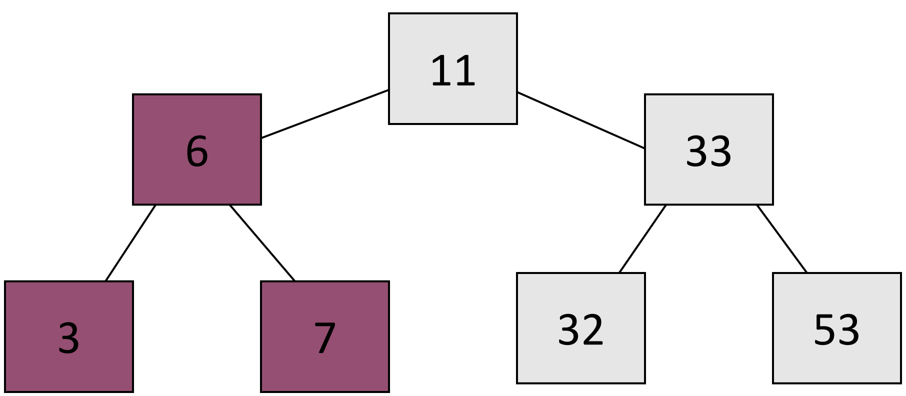


Search for target = 7

Search left subarray:

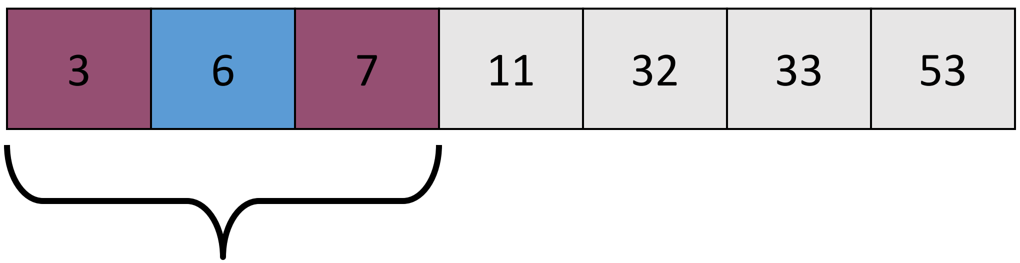


Search left subtree:

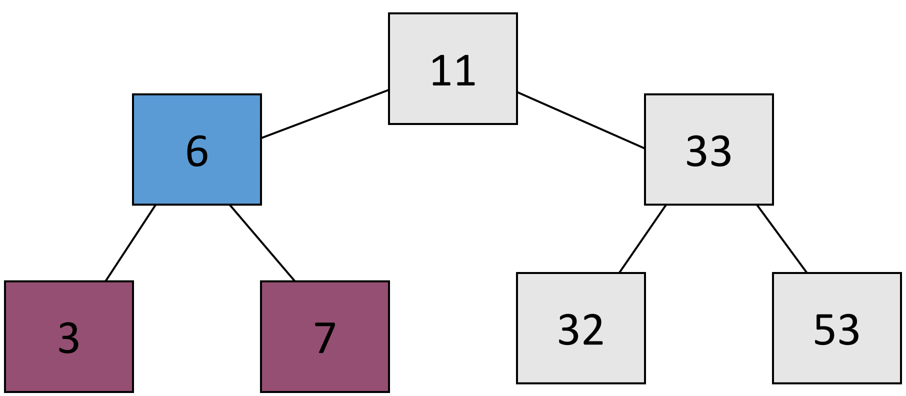


Search for target = 7

Find approximate midpoint of subarray:

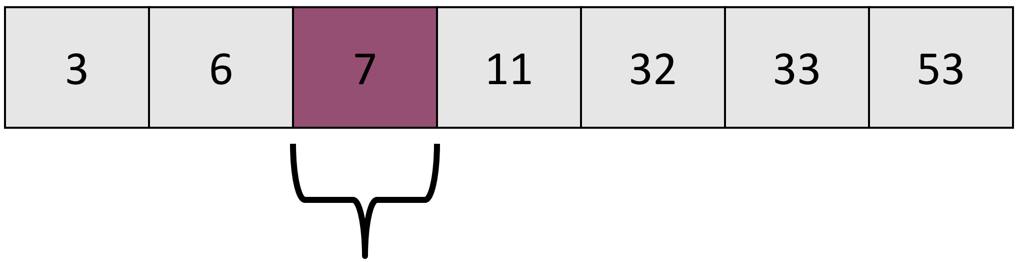


Visit root of subtree:

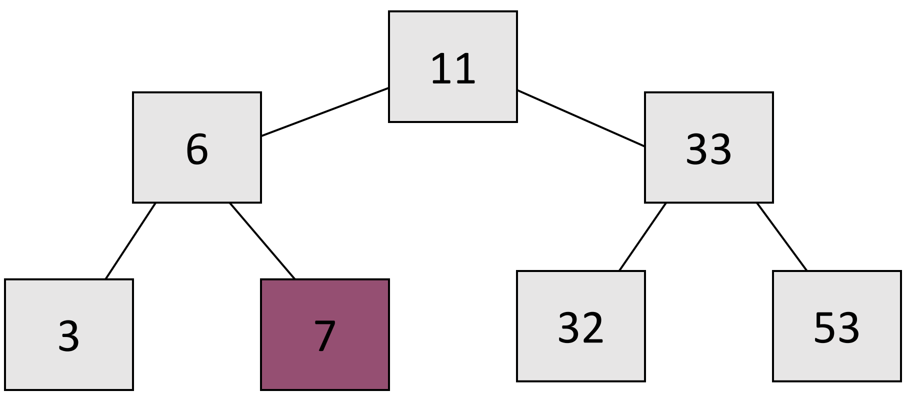


Search for target = 7

Search right subarray:



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 = $O(\log_2 n)$.

Faster than sequential search

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

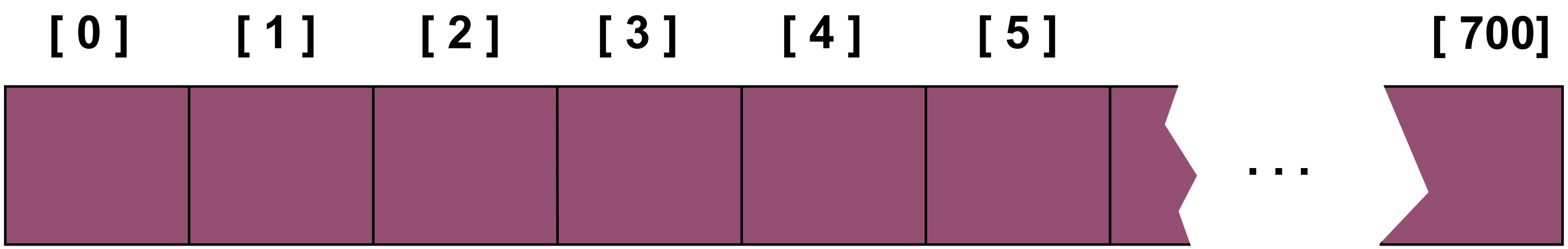
Hash Table

Outline

- Introduction of Hash Table
- Open Address Hashing
- Chained Hashing

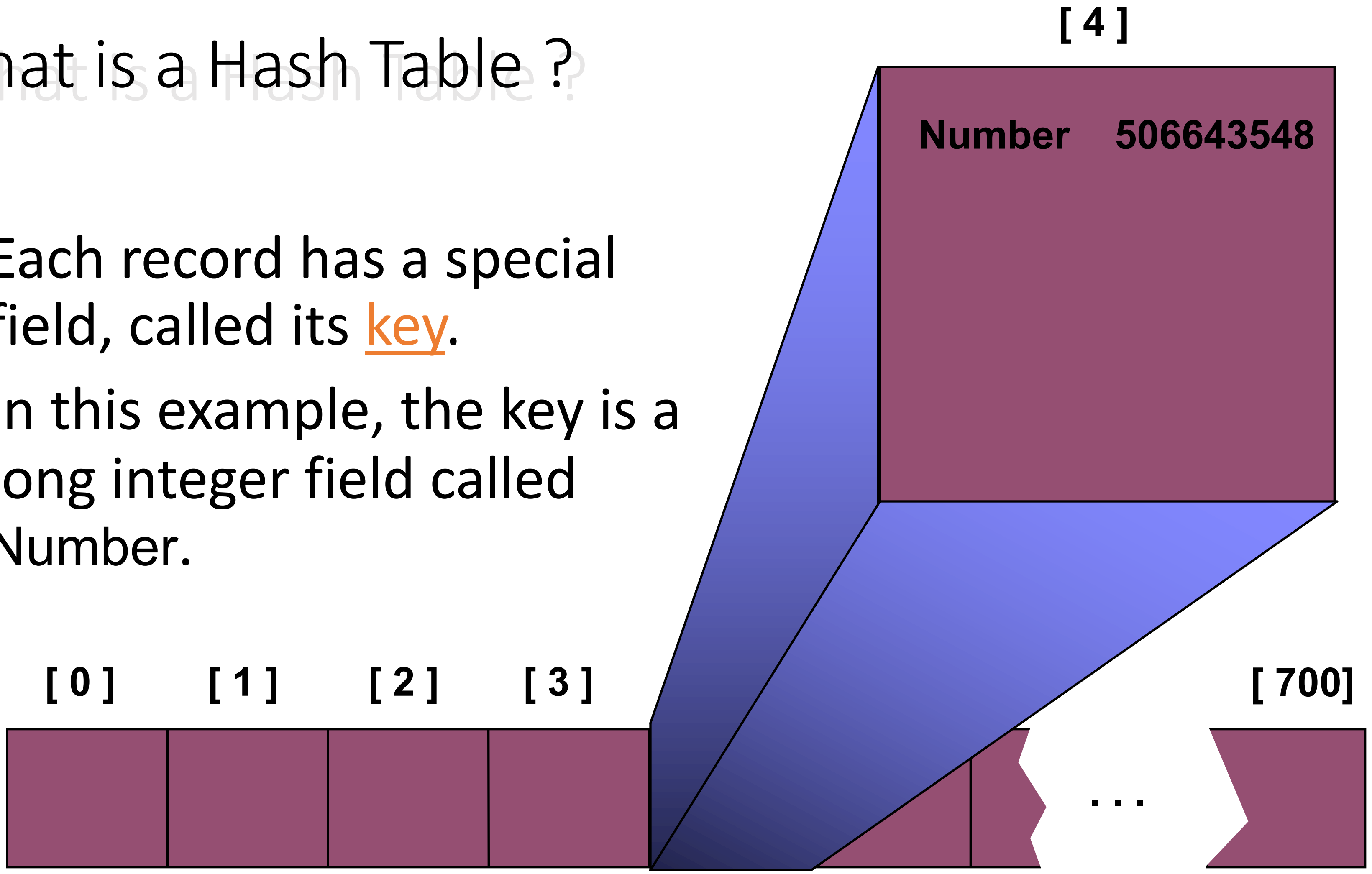
What is a Hash Table ?
What is a Hash Table ?

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



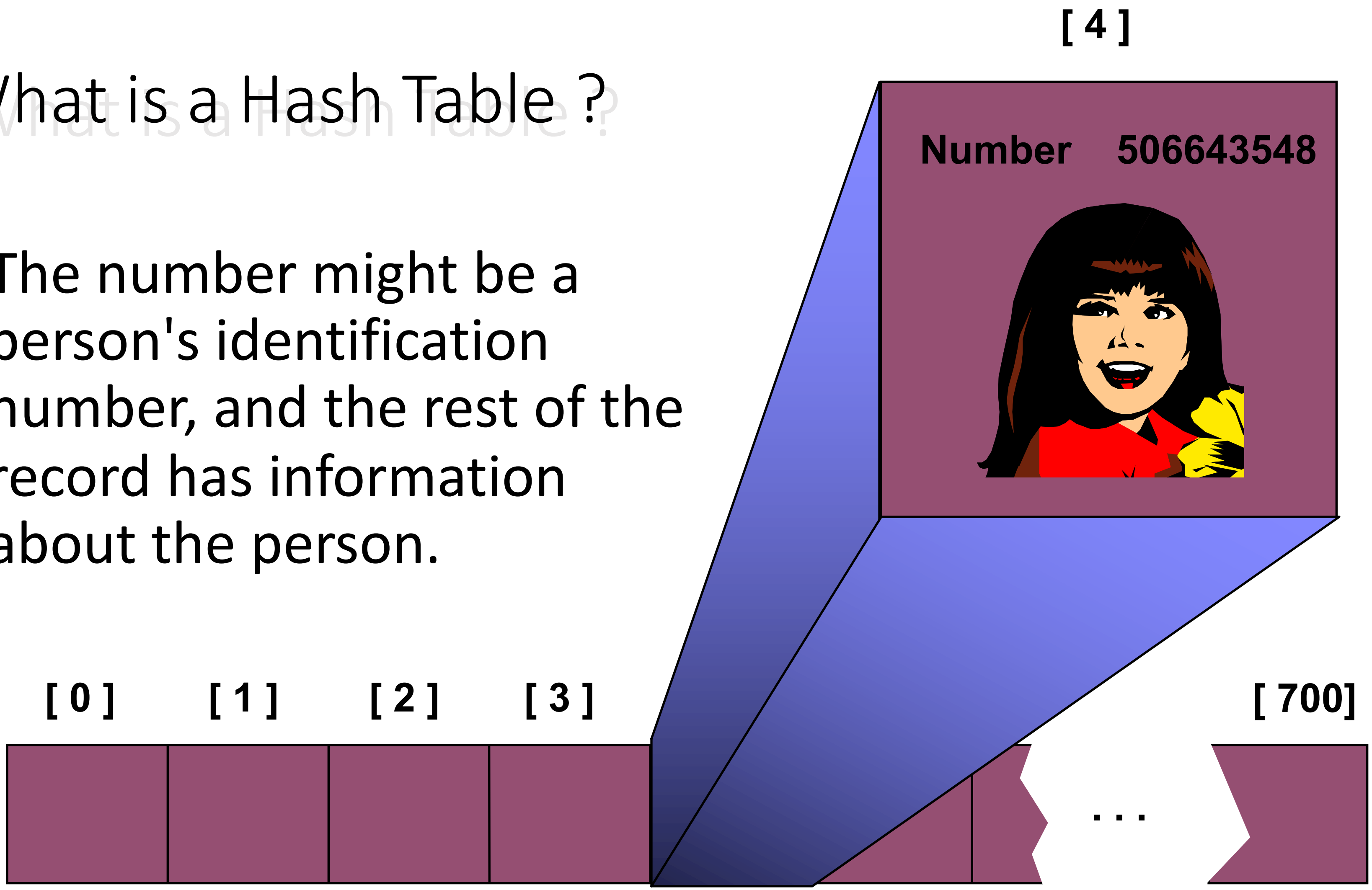
What is a Hash Table ?

- 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 ?

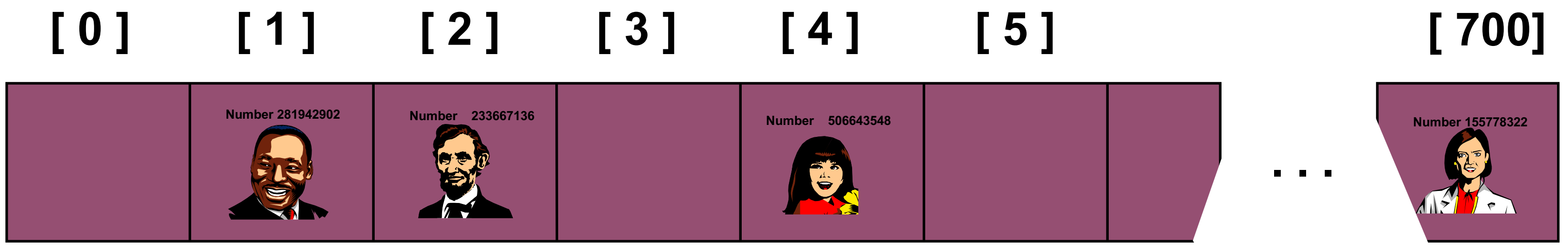
- The number might be a person's identification number, and the rest of the record has information about the person.



What is a Hash Table ?

What is a Hash Table ?

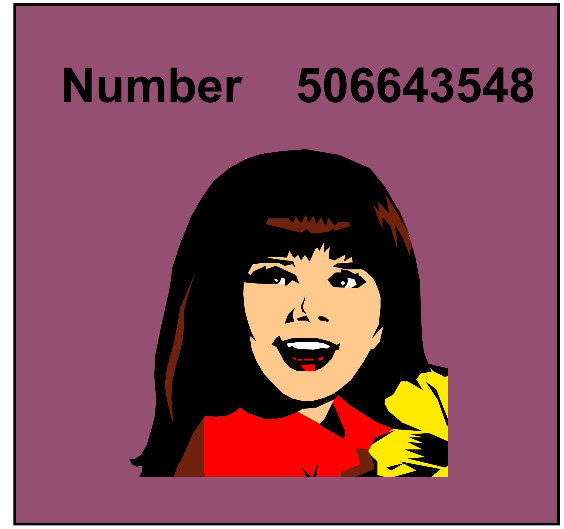
- When a hash table is in use, some spots contain valid records, and other spots are "empty".

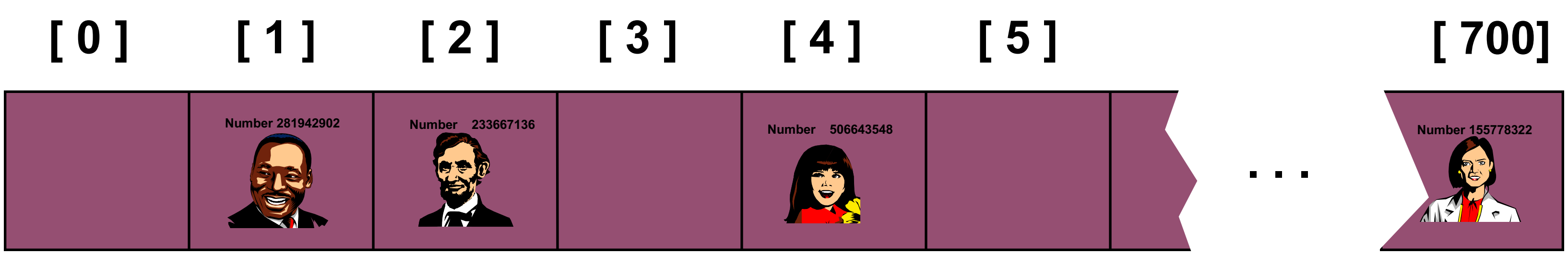


Open Address Hashing

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

[4]



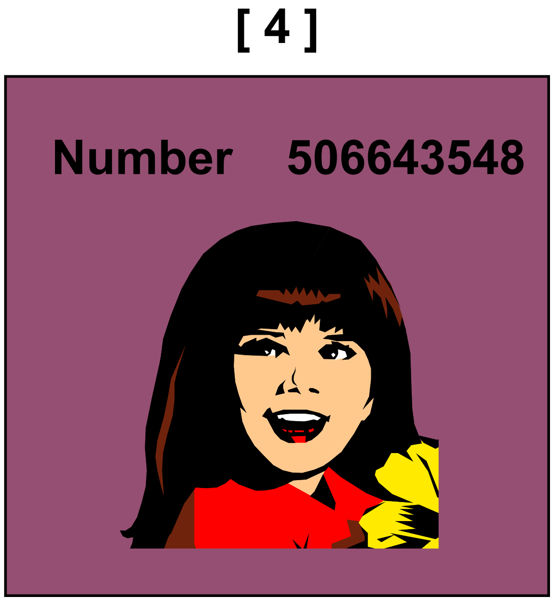


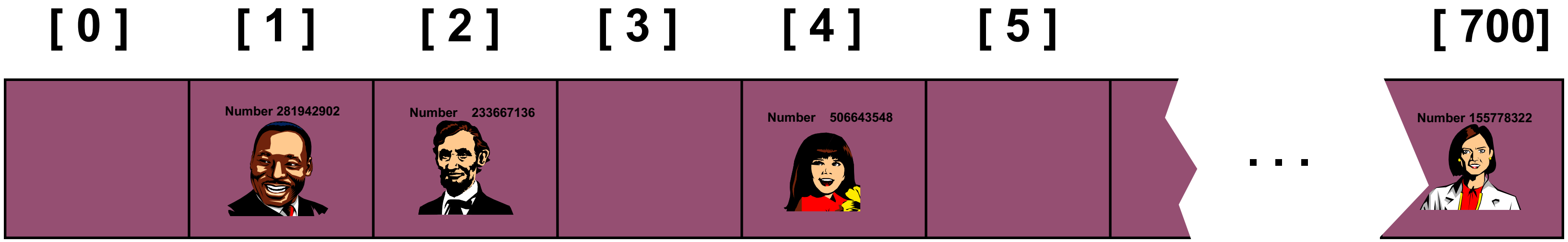
A Typical Hash Function

hash_value = key mod length_of_list

For example,

- 4 = 506643548 % 701

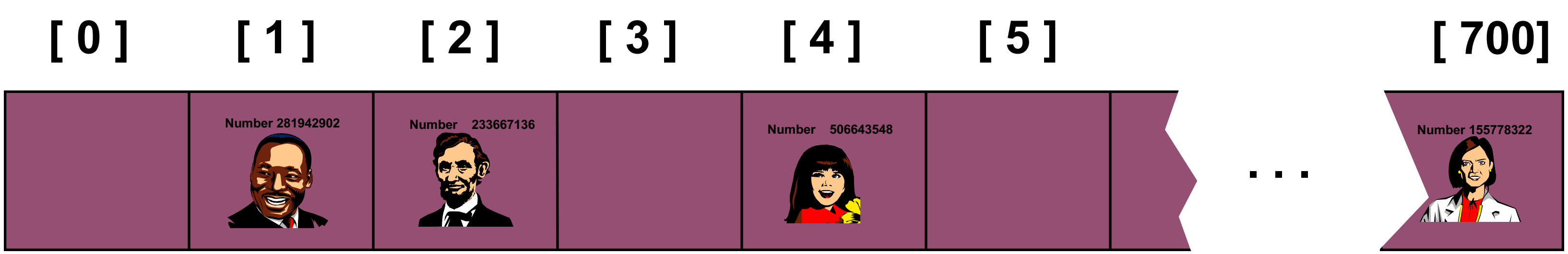




To Insert A Record

- Apply the hash function to the key of the record to be inserted to obtain the hash value

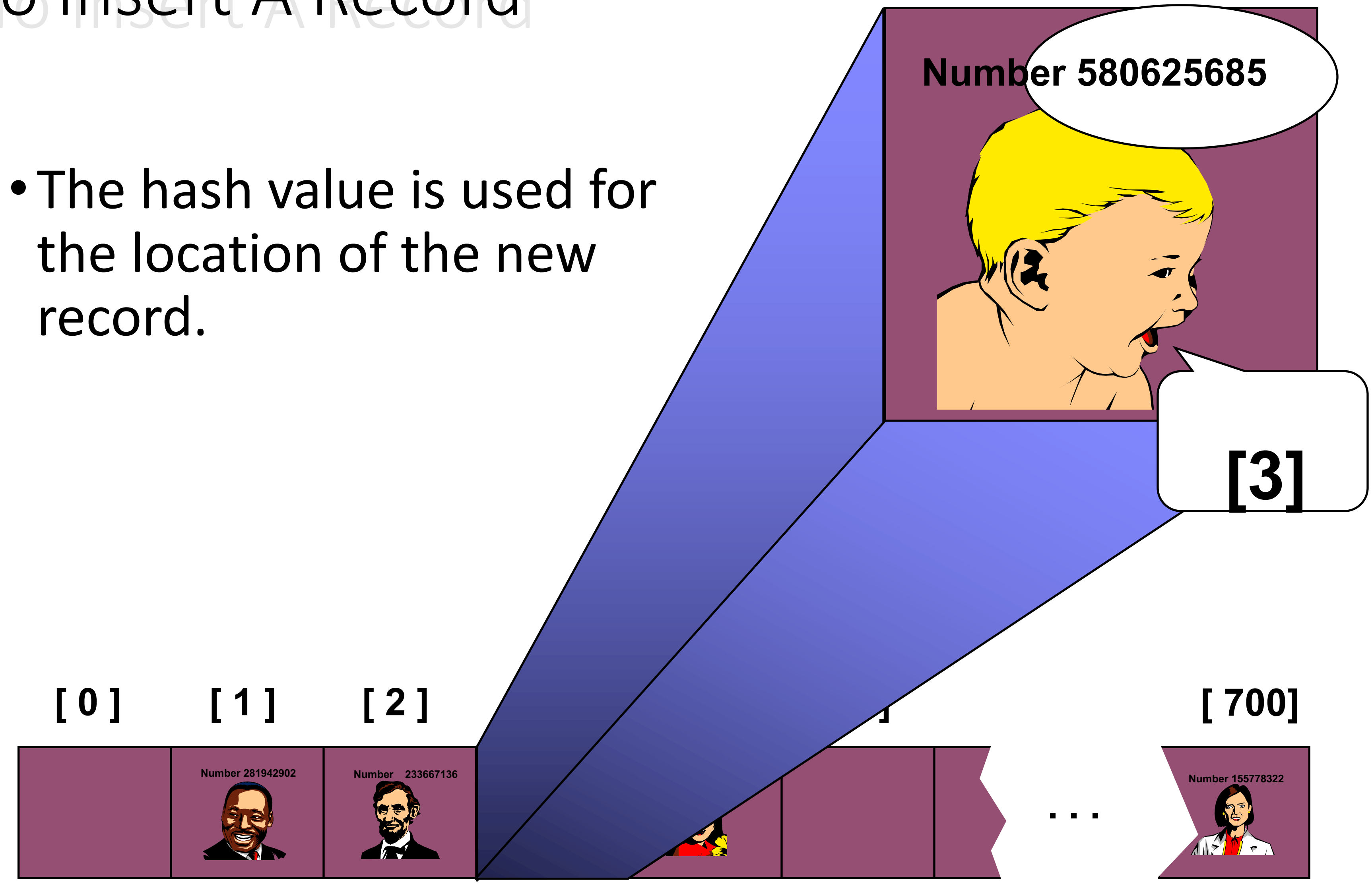
What is (580625685 % 701) ?



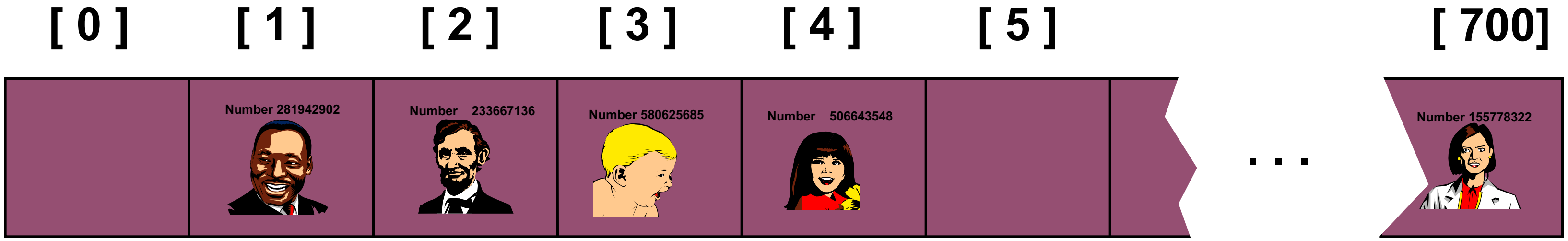
To Insert A Record

To Insert A Record

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

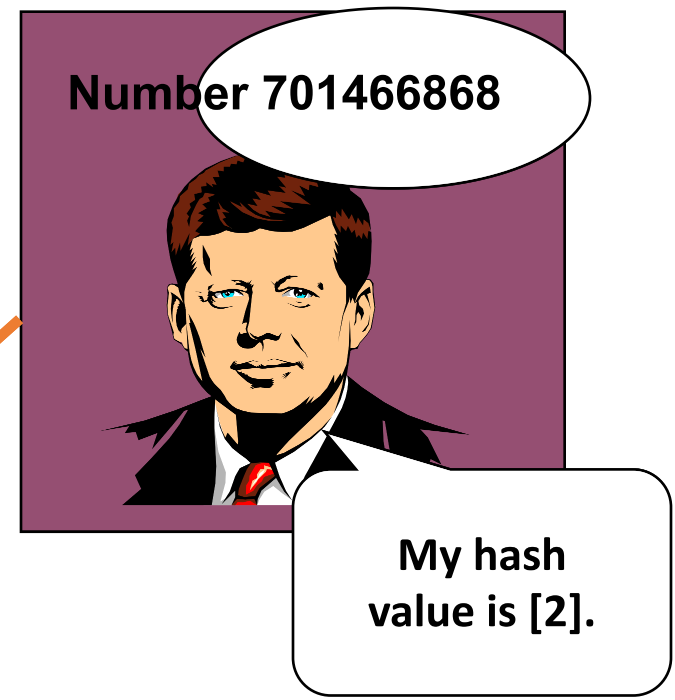


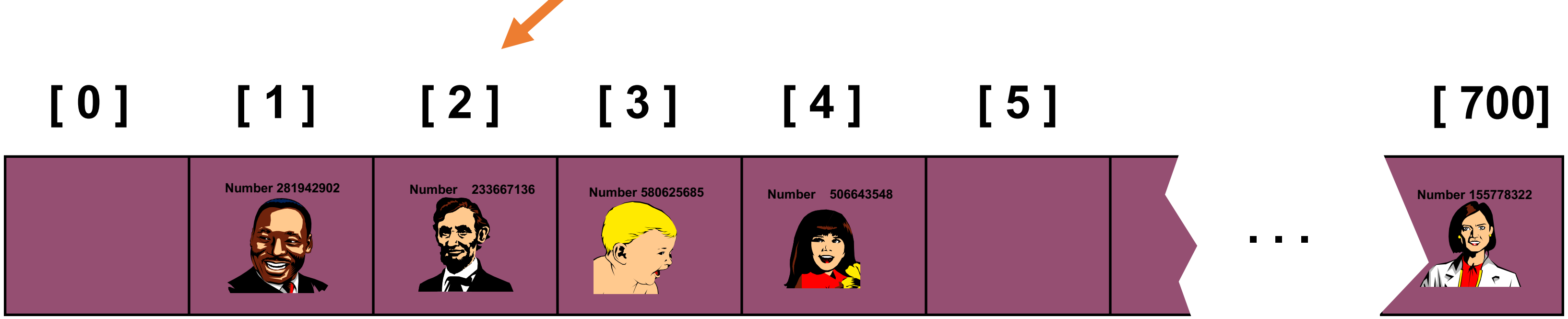
To Insert A Record



Collisions

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

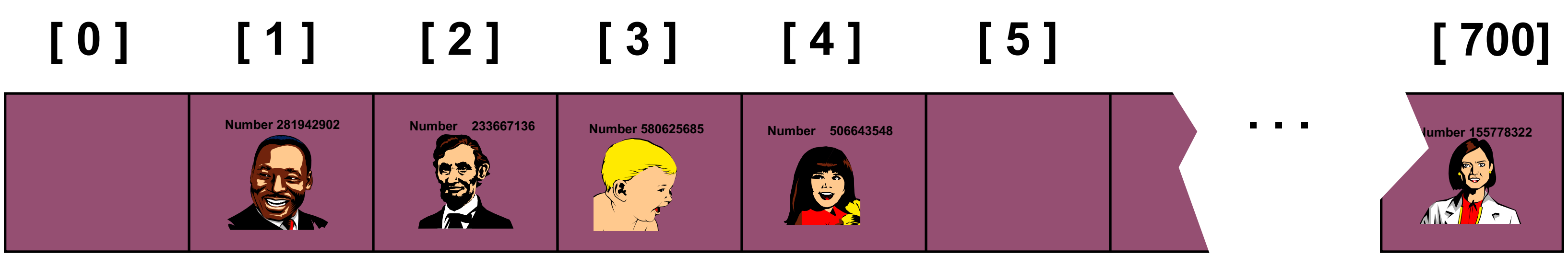




Collisions

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

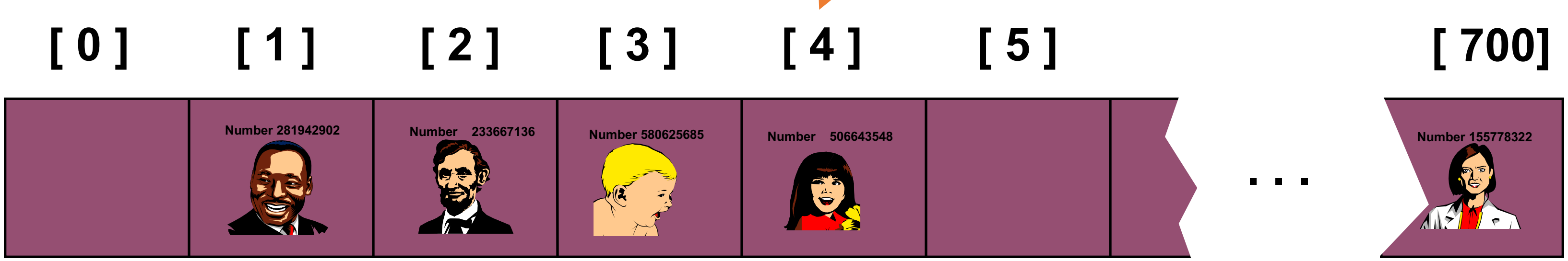
When a collision occurs, move forward until you find an empty spot.



Collisions

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

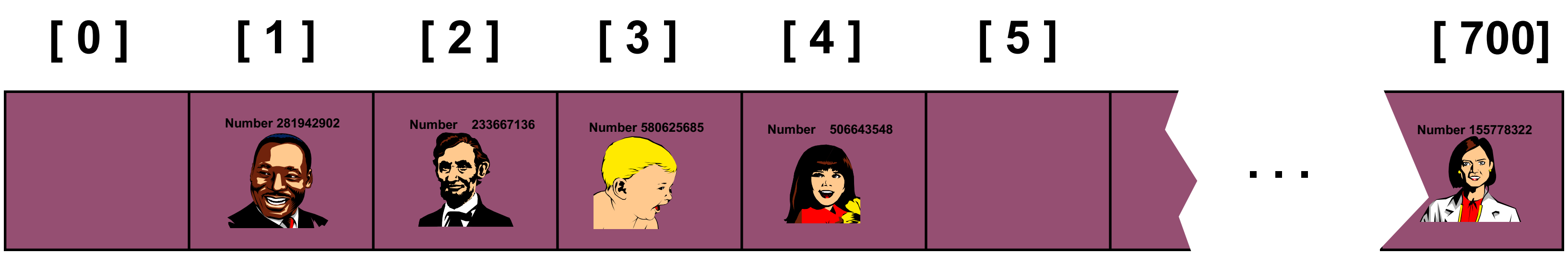
When a collision occurs, move forward until you find an empty spot.



Collisions

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

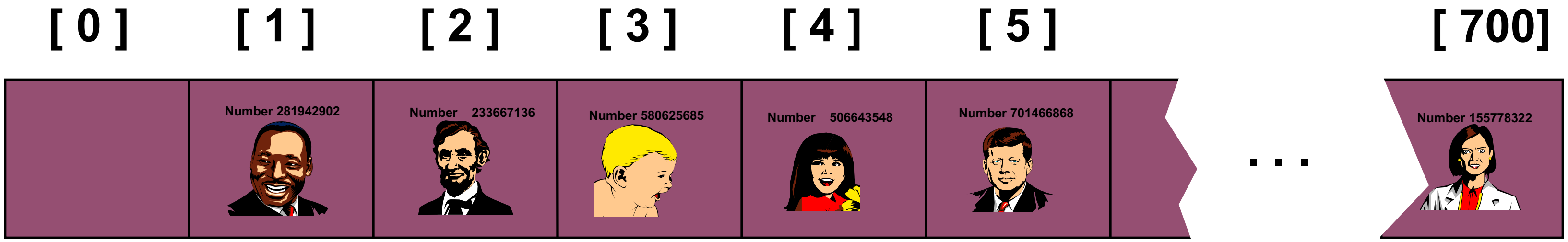
When a collision occurs, move forward until you find an empty spot.



Collisions

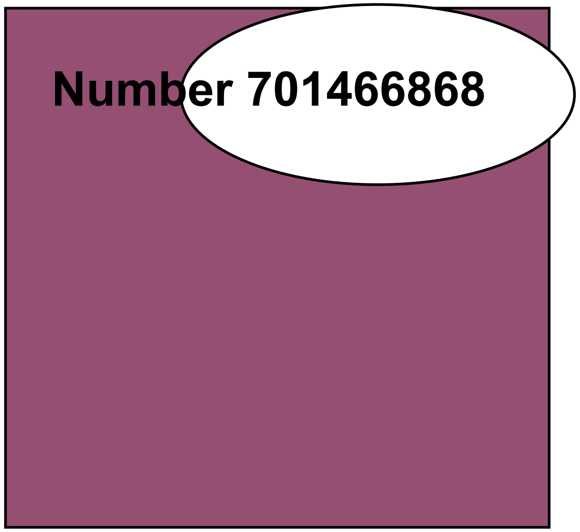
- This is called a **collision**, because there is already another valid record at [2].

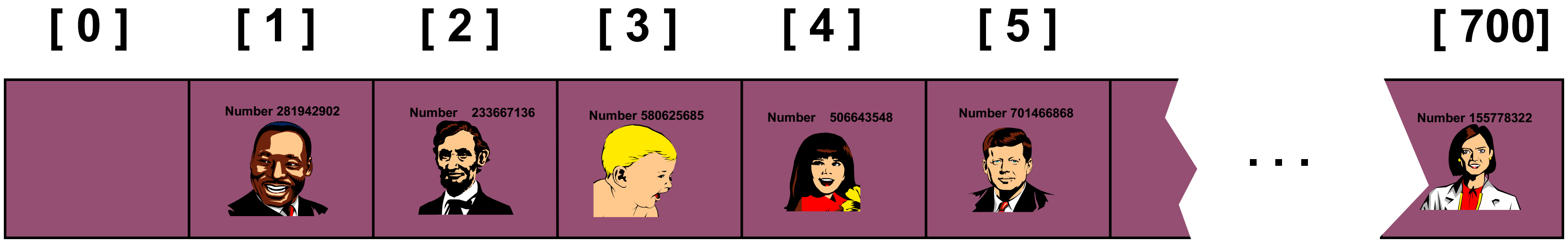
The new record goes in the empty spot.



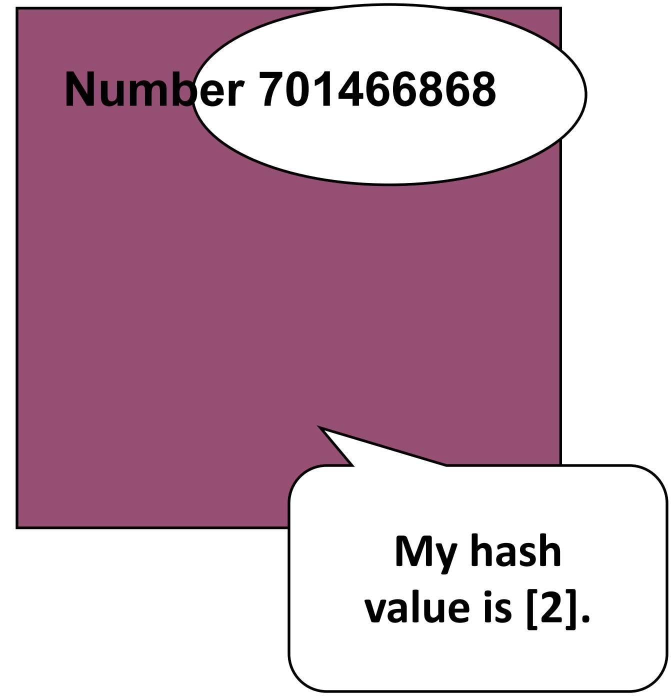
Searching for A Record

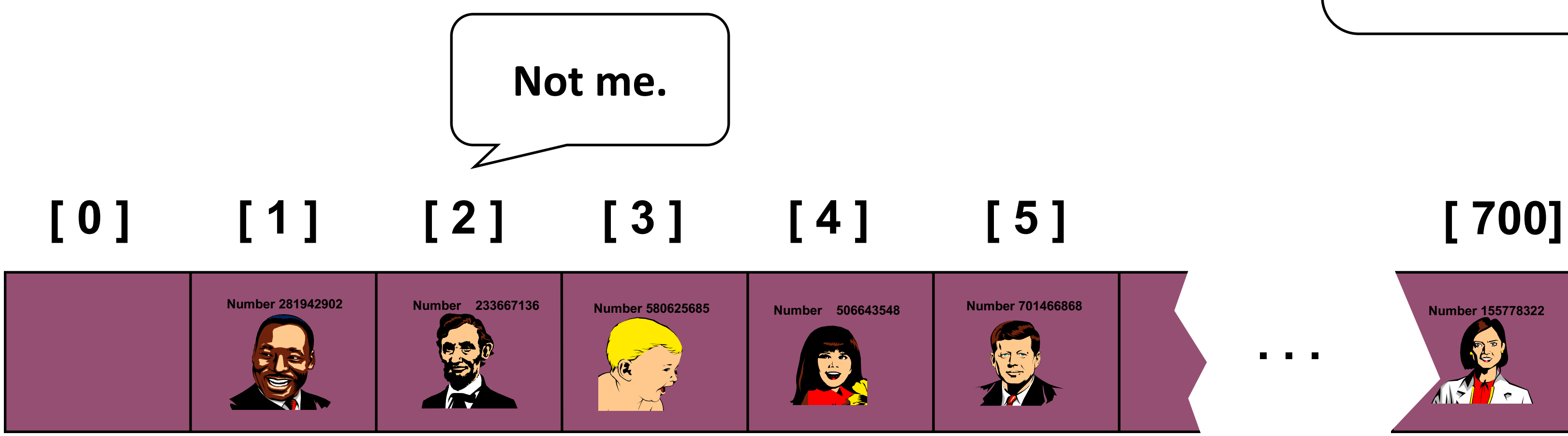
- The data that's attached to a key can be found fairly quickly.



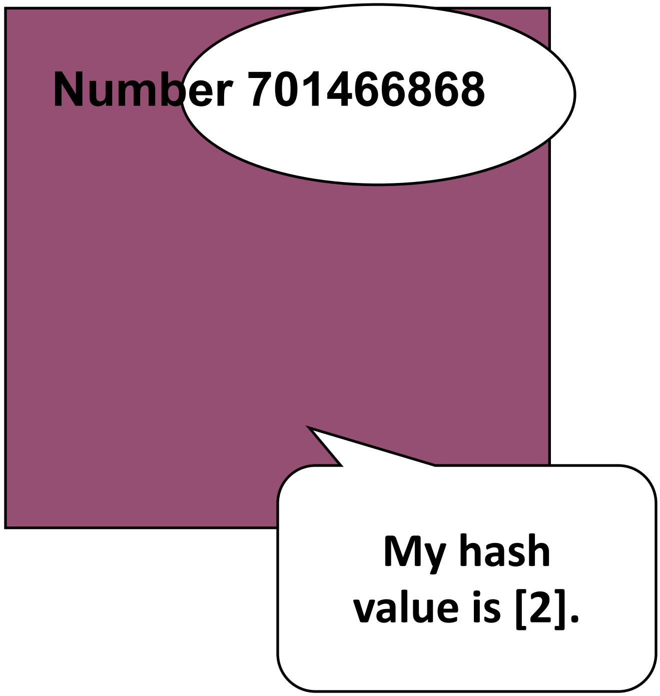


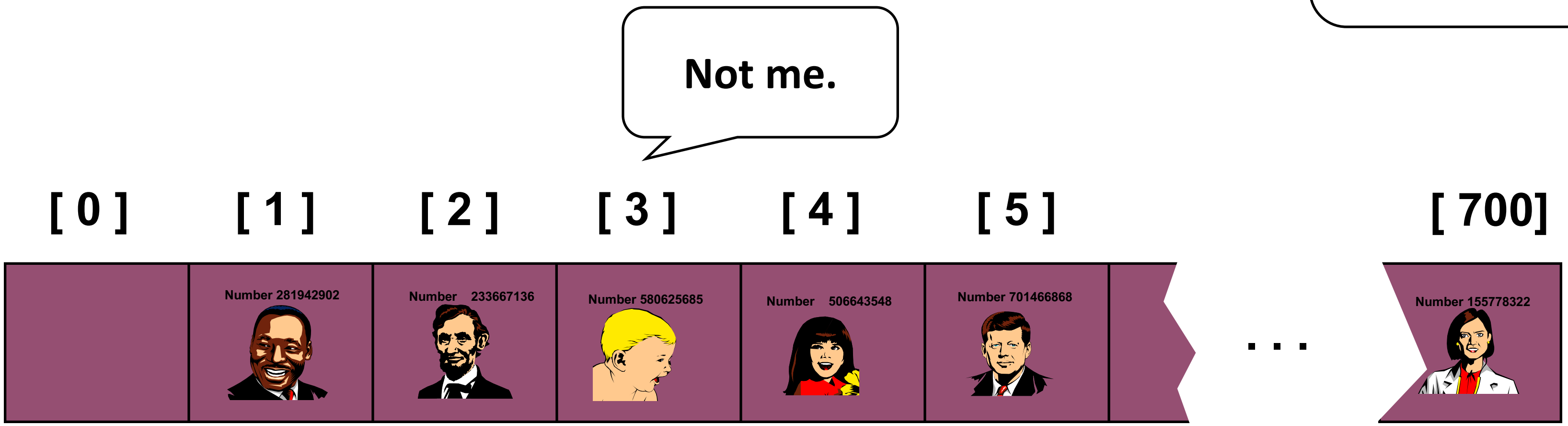
- Calculate the hash value.
- Check that location of the array for the key.



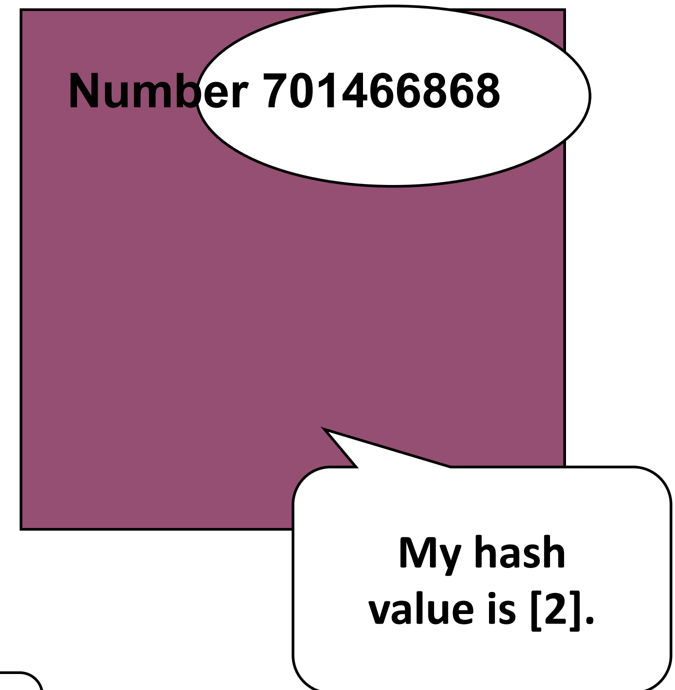


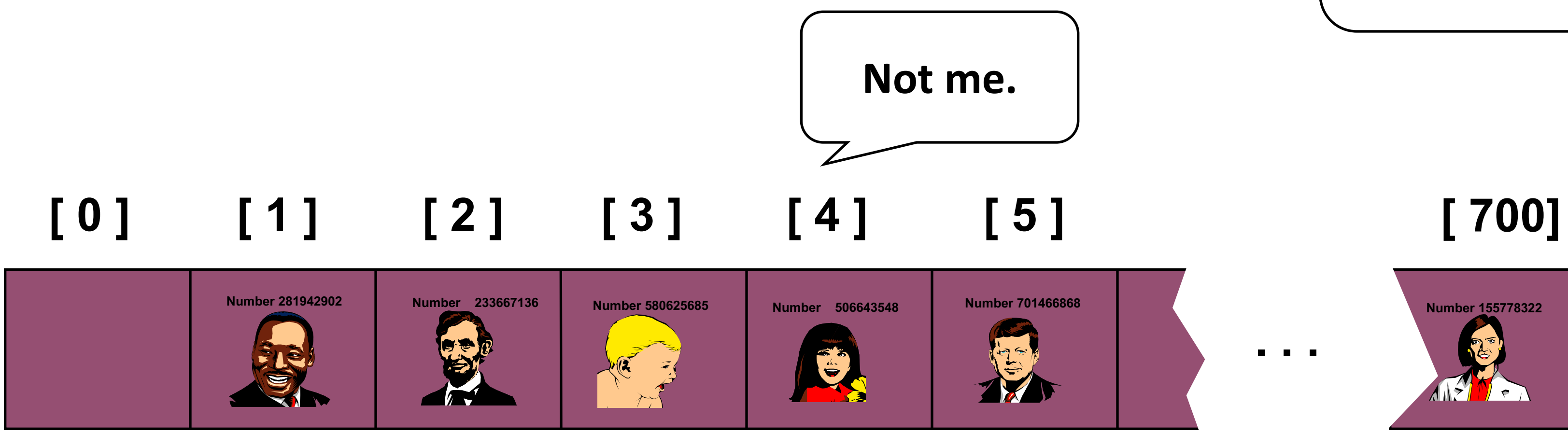
- Keep moving forward until you find the key, or you reach an empty spot.



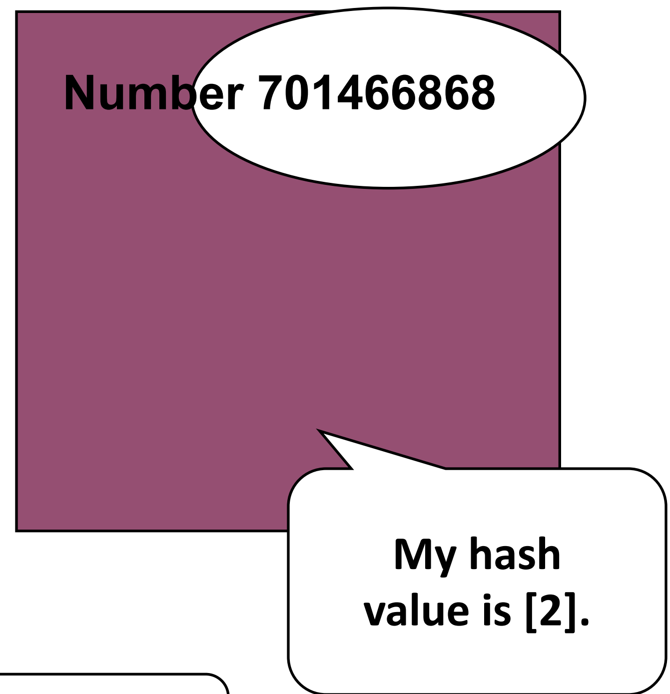


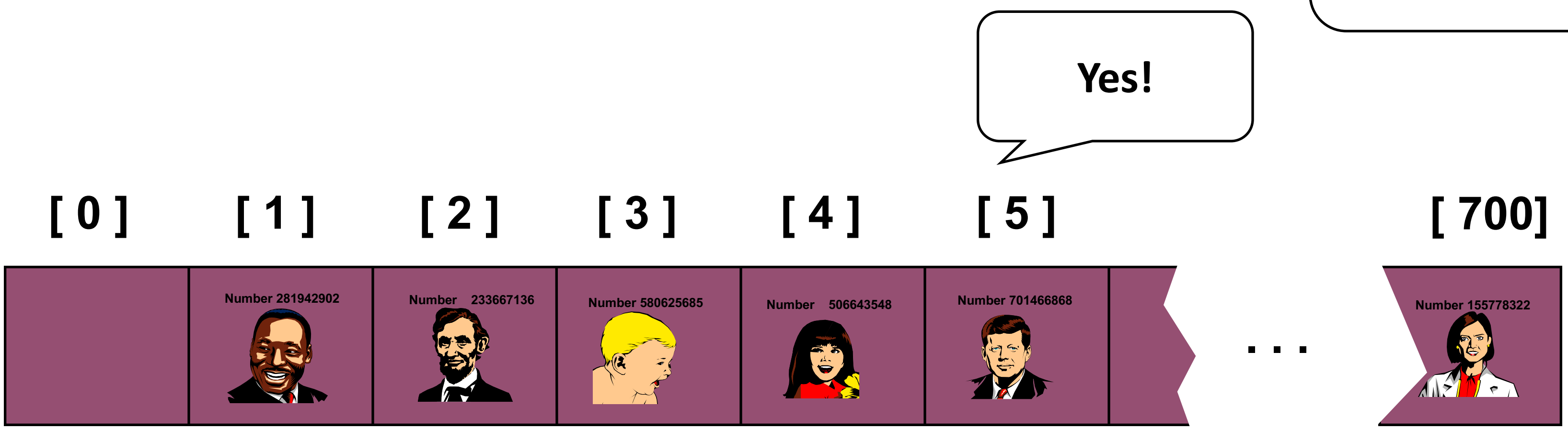
- Keep moving forward until you find the key, or you reach an empty spot.



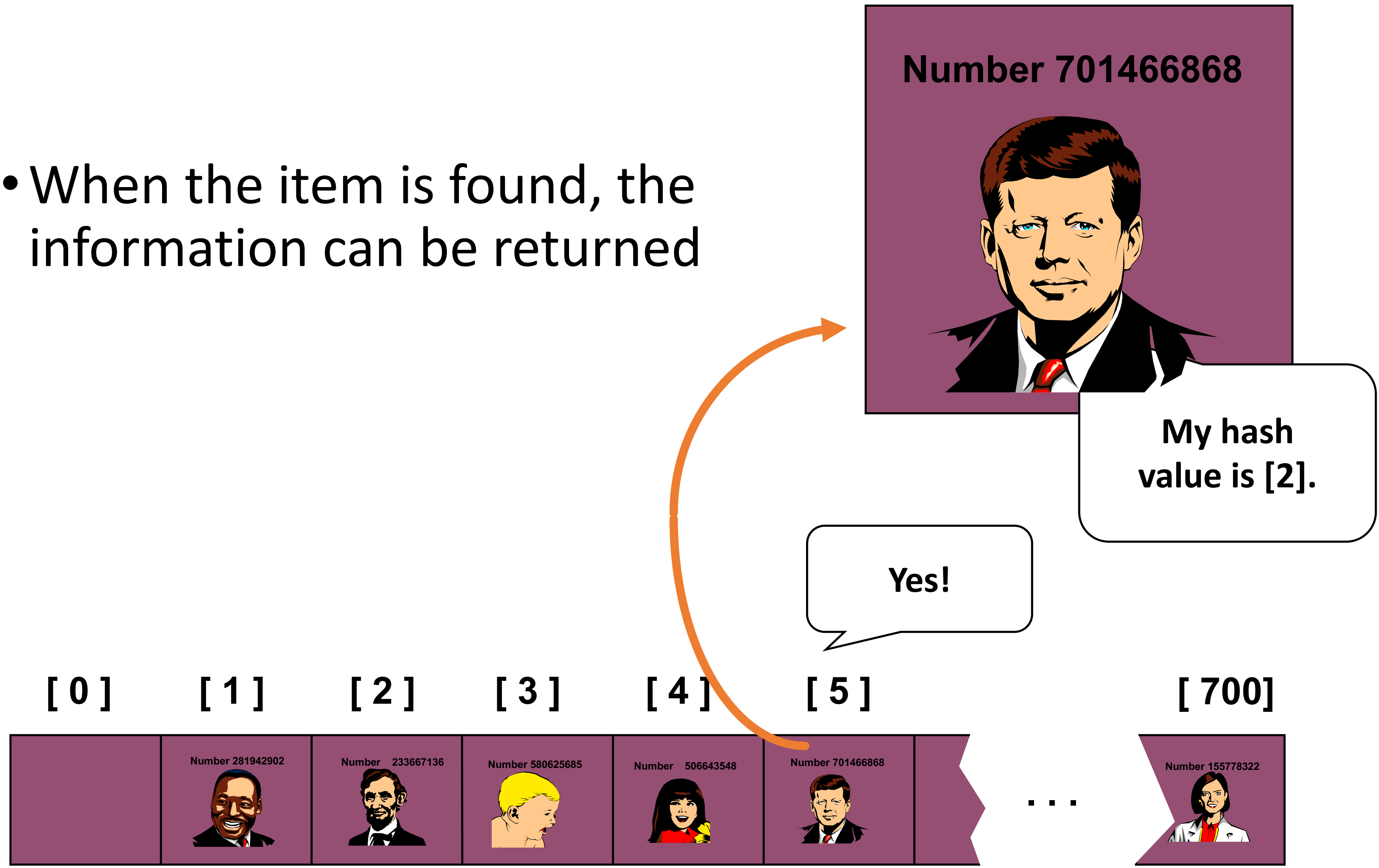


- Keep moving forward until you find the key, or you reach an empty spot.



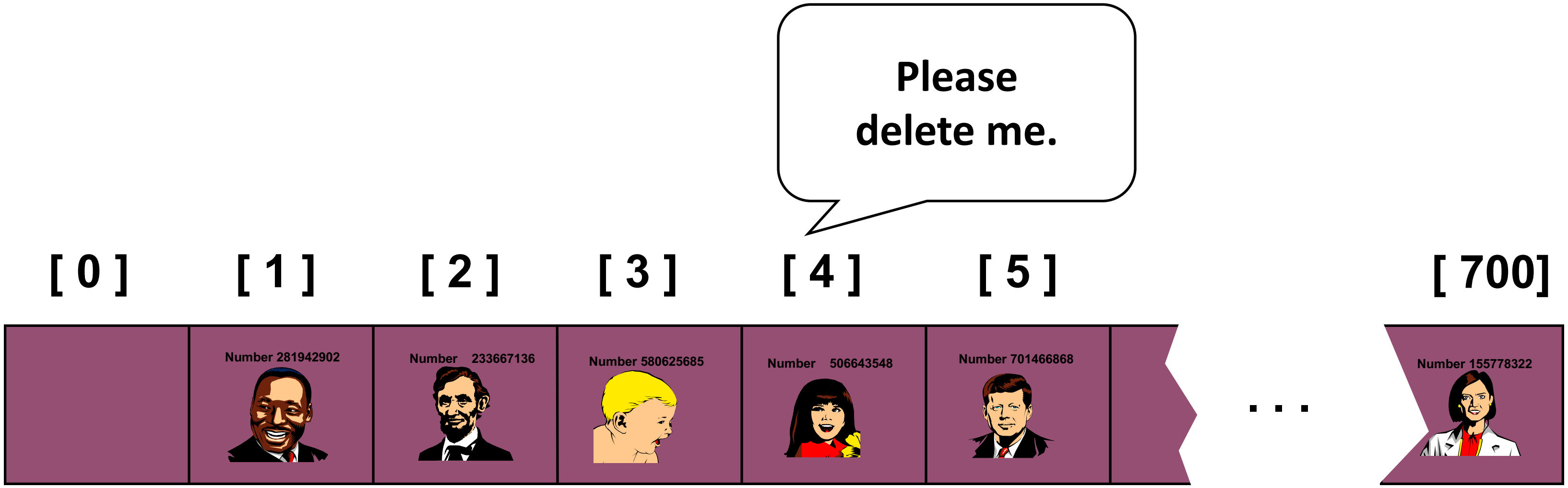


- When the item is found, the information can be returned



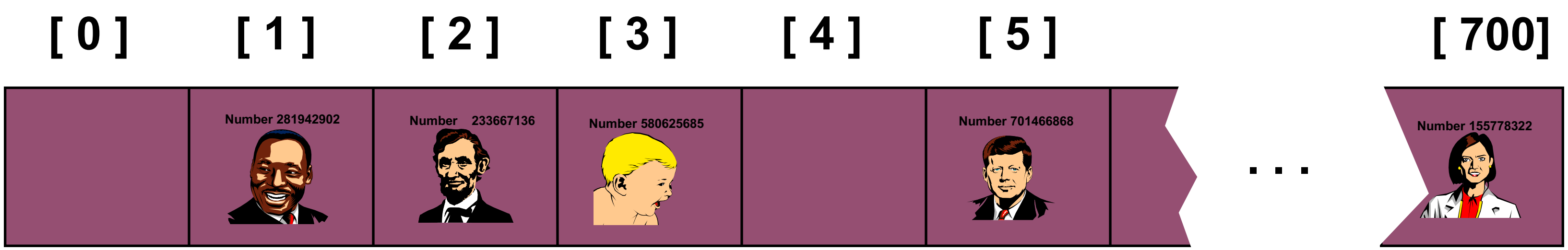
Deleting a Record

- Records may also be deleted from a hash table.



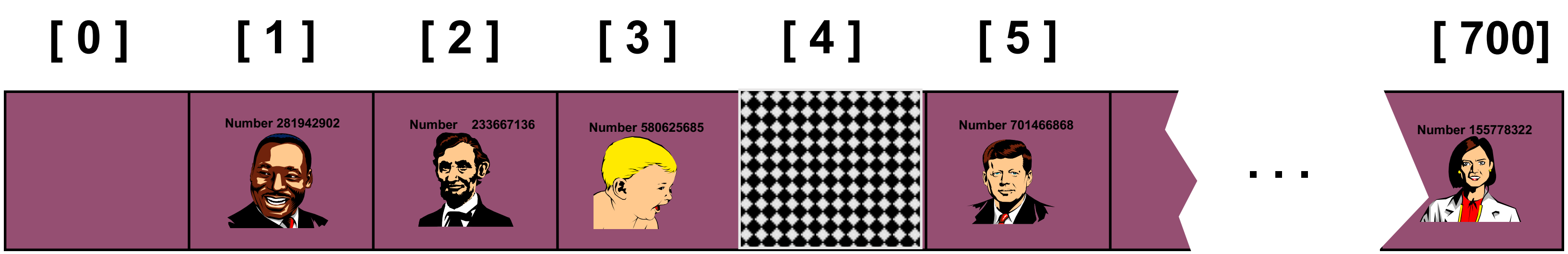
Deleting a Record

- 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.

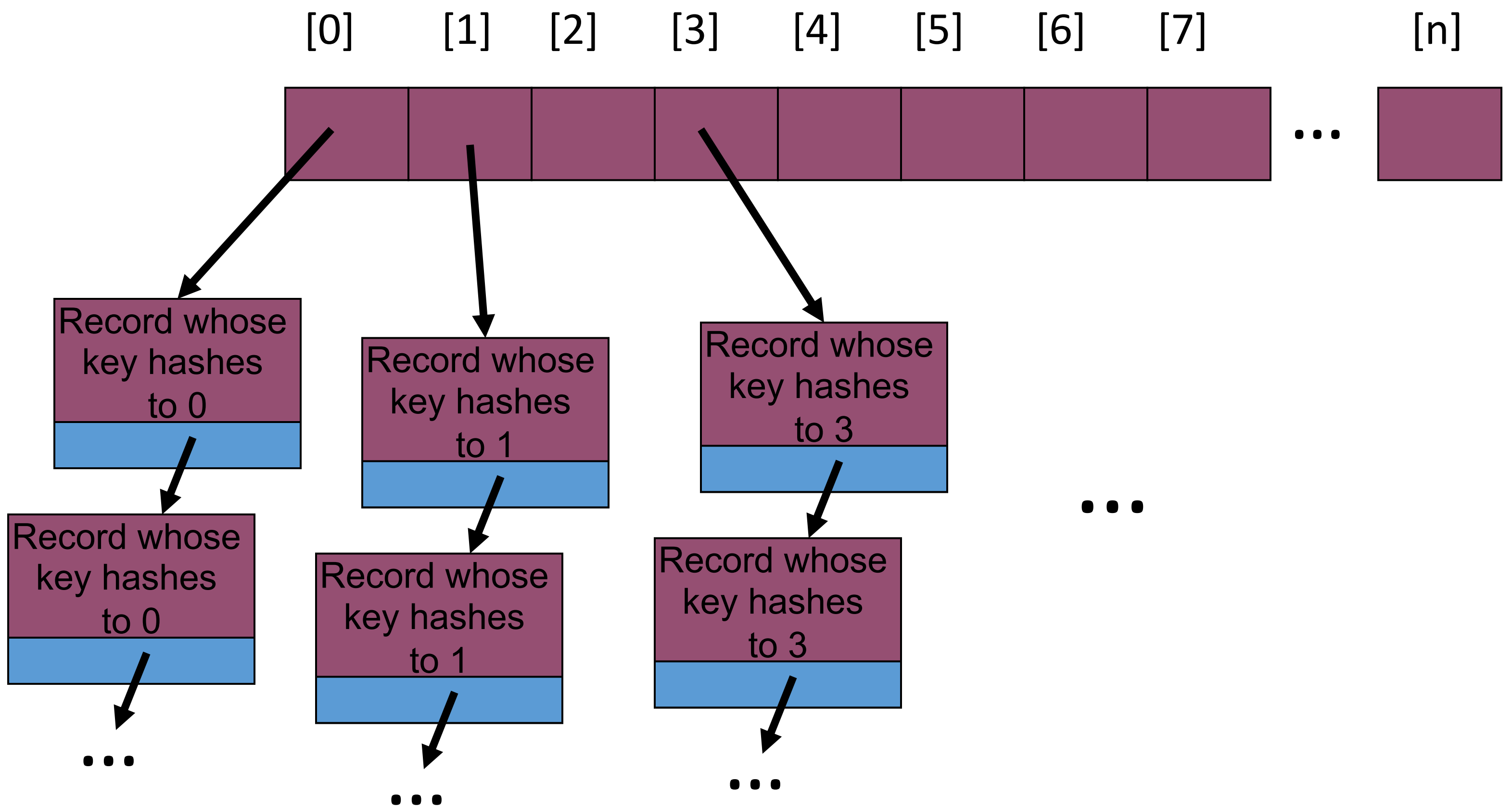


Chained Hashing

- 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:



Summary

- 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
- Chained hashing