

# Data Structures and Algorithms

## Lecture# 6

Shikha Mehrotra

# Searching Algorithms



## • **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.

A	2	6	13	21	36	47	63	81	97
	0	1	2	3	4	5	6	7	8

x exists ?

36 Yes (4)

45 No

97 Yes (8)

Search(A,n,x)

```

{
  for i=0 to n-1
    if A[i]==x
      return i
  return -1
}

```

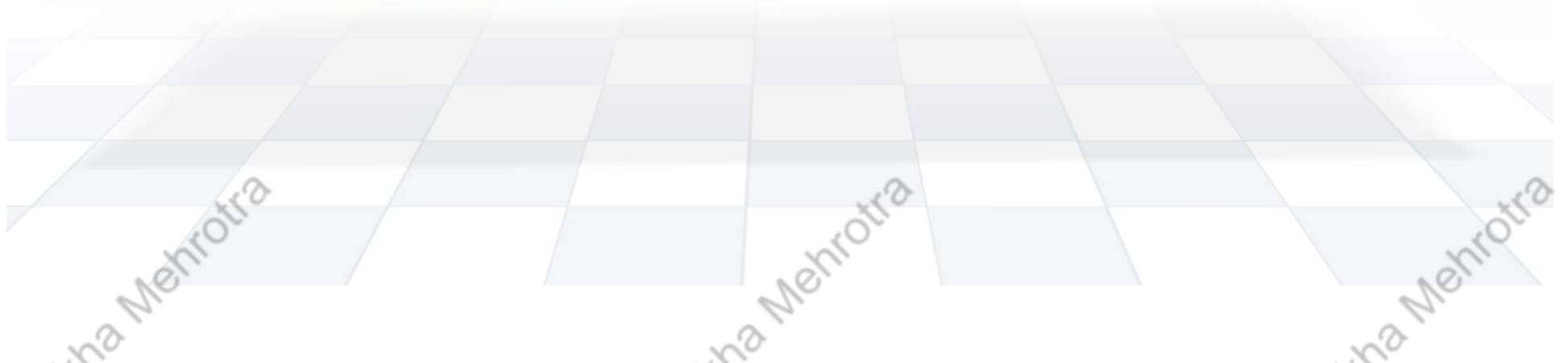
Best Case:

1 Comparison  $O(1)$

Worst Case"

n Comparison  $O(n)$

# Sequential / Linear Search



## • 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



# • Problem with Sequential Search

- Inefficiency

1. On the average, it will take 5 searches to find a name (assuming no bias in choice of name)
2. If name is not found, it will take 10 searches to ascertain that information. A list of 500 names requires 500 searches.

Note: in this age of speed, 500 searches can be done in a microsecond. It is hard to justify a need to generate a more efficient method to save such a small amount of time. It should be mentioned that if there were 10 million names with the process continuing over and over, the need for efficiency is much more pronounced. Hence a more efficient method is needed.

## • 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:

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

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

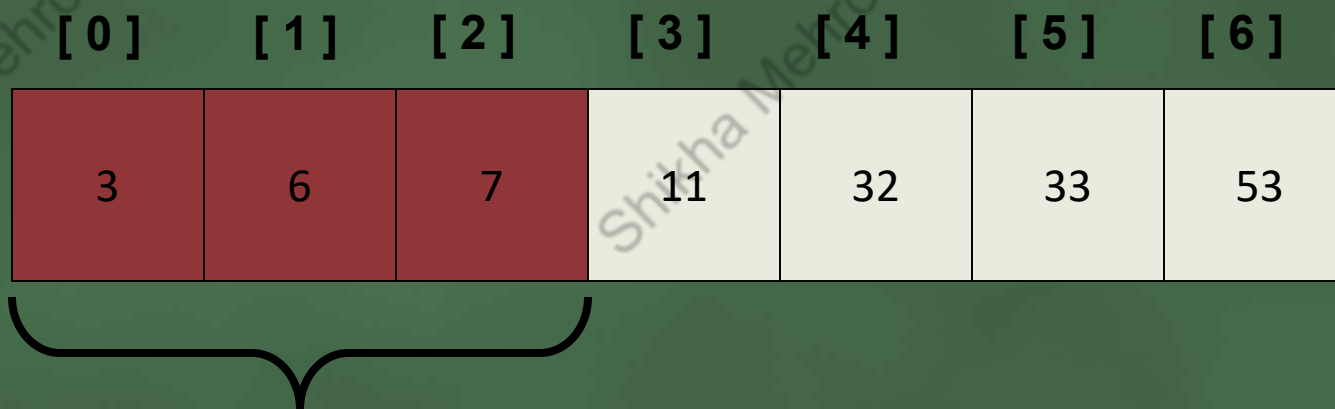
Is  $7 < \text{midpoint key}$ ? YES.



# • 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

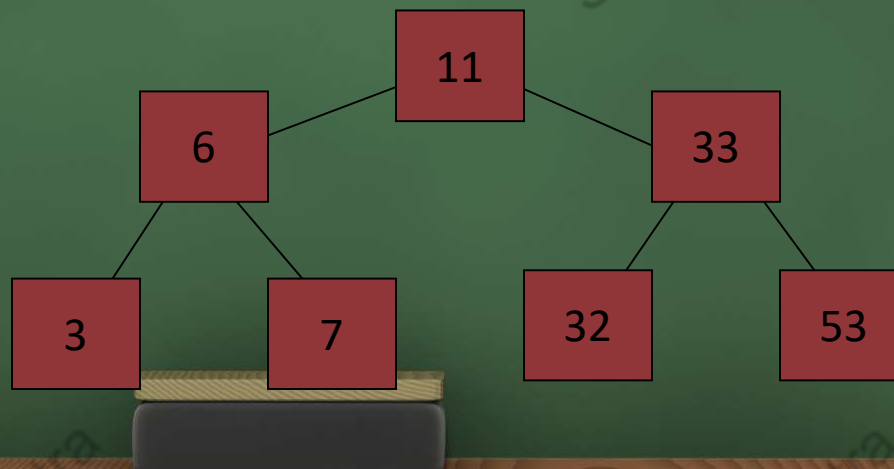
```
void search(const int a[ ], size_t first, size_t size, int target, bool& found, size_t& location)
{
    size_t middle;
    if(size == 0) found = false;
    else {
        middle = first + size/2;
        if(target == a[middle]){
            location = middle;
            found = true;
        }
        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

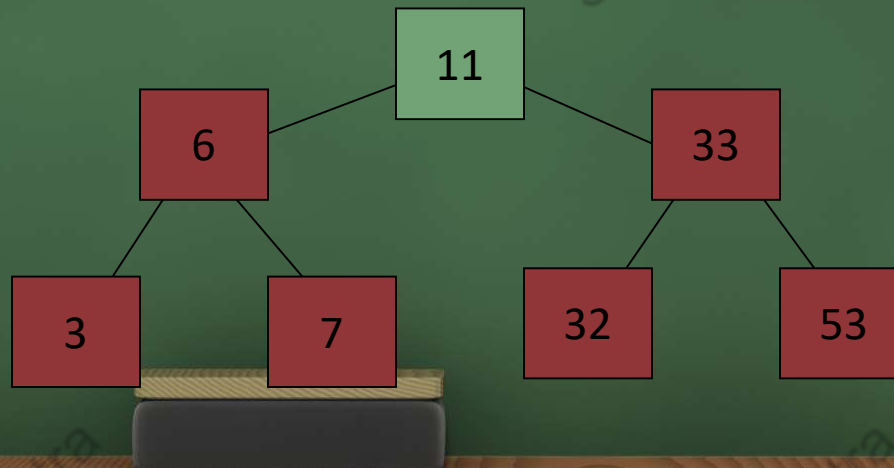


# •Search for target = 7

Find midpoint:

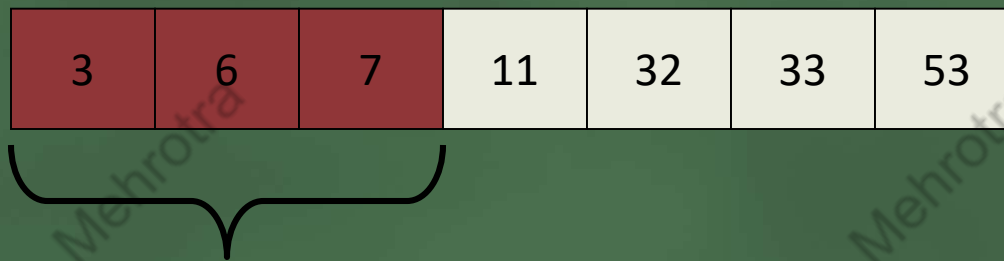
3	6	7	11	32	33	53
---	---	---	----	----	----	----

Start at root:

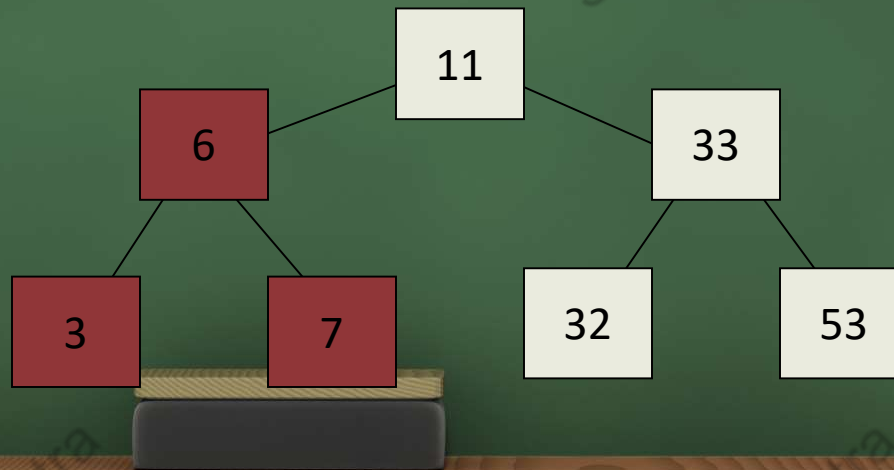


# •Search for target = 7

Search left subarray:

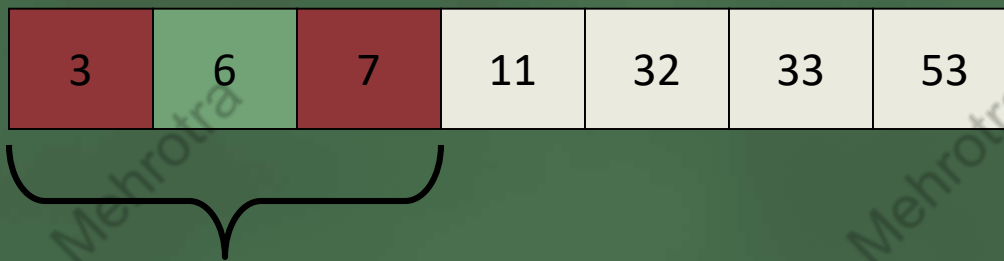


Search left subtree:

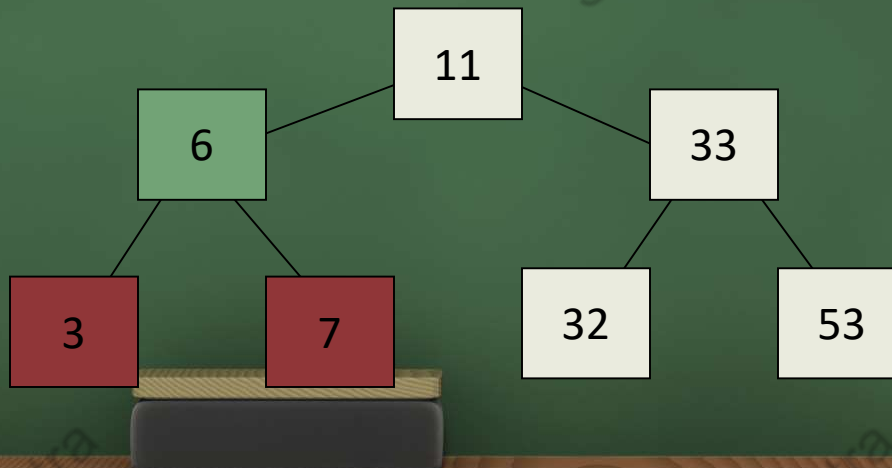


# •Search for target = 7

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:



## • Efficiency of binary search

# of names	Maximum sequential searches necessary	Maximum binary searches necessary
10	10	4
100	100	7
1,000	1,000	10
5,000	5,000	13
10,000	10,000	14
50,000	50,000	16
1,00,000	1,00,000	17
10,00,000	10,00,000	20
1,00,00,000	1,00,00,000	24
1,00,00,00,000	1,00,00,00,000	30

With the incredible speed of today's computers, a binary search becomes necessary only when the number of names is large.



# • 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  $\text{floor}(\log_2 n)$  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 sequential 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!

# Hashing



# Need of Hash Table ?





# Need of Hash Table ?



# • Introduction to Hashing

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.
  - A linked list implementation would take  $O(n)$  time.
  - Using an array of size 100,000 would give  $O(1)$  access time but will lead to a lot of space wastage.
- Is there some way that we could get  $O(1)$  access without wasting a lot of space?
- The answer is **hashing**.

# • **Common Hashing Functions**

## 1. **Division Remainder (using the table size as the divisor)**

- Computes hash value from key using the % operator.
- Table size that is a power of 2 like 32 and 1024 should be avoided, for it leads to more collisions.
- Also, powers of 10 are not good for table sizes when the keys rely on decimal integers.
- Prime numbers not close to powers of 2 are better table size values.



# • **Common Hashing Functions (cont'd)**

## 2. Truncation or Digit/Character Extraction

- Works based on the distribution of digits or characters in the key.
- More evenly distributed digit positions are extracted and used for hashing purposes.
- For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision.
- Very fast but digits/characters distribution in keys may not be very even.

# • **Common Hashing Functions (cont'd)**

## 3. Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.
- To map the key 25936715 to a range between 0 and 9999, we can:
  - split the number into two as 2593 and 6715 and
  - add these two to obtain 9308 as the hash value.
- Very useful if we have keys that are very large.
- Fast and simple especially with bit patterns.
- A great advantage is ability to transform non-integer keys into integer values.

# • Common Hashing Functions (cont'd)

## 4. Radix Conversion

- Transforms a key into another number base to obtain the hash value.
- Typically use number base other than base 10 and base 2 to calculate the hash addresses.
- To map the key 55354 in the range 0 to 9999 using base 11 we have:

$$55354_{10} = 38652_{11}$$

- We may truncate the high-order 3 to yield 8652 as our hash address within 0 to 9999.

# • Common Hashing Functions (cont'd)

## 5. Mid-Square

- The key is squared and the middle part of the result taken as the hash value.
- To map the key **3121** into a hash table of size **1000**, we square it  $3121^2 = 9740641$  and extract **406** as the hash value.
- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys have to be preprocessed to obtain corresponding integer values.

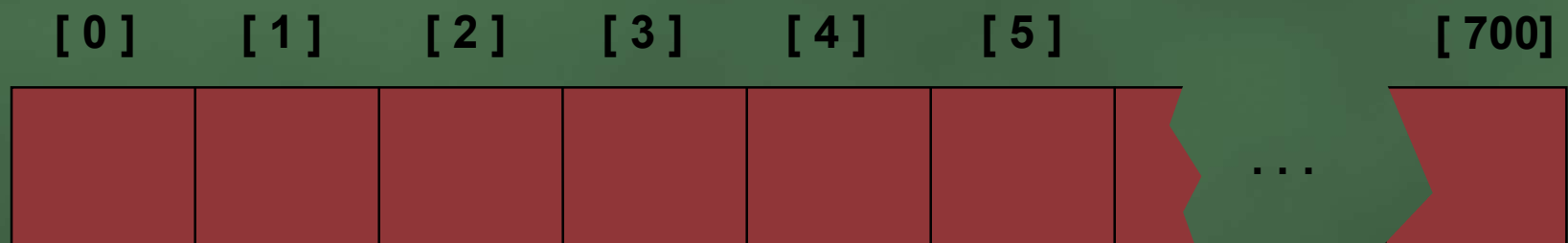
# • **Common Hashing Functions (cont'd)**

## 6. Use of a Random-Number Generator

- Given a seed as parameter, the method generates a random number.
- The algorithm must ensure that:
  - It always generates the same random value for a given key.
  - It is unlikely for two keys to yield the same random value.
- The random number produced can be transformed to produce a valid hash value.

# •What is a Hash Table ?

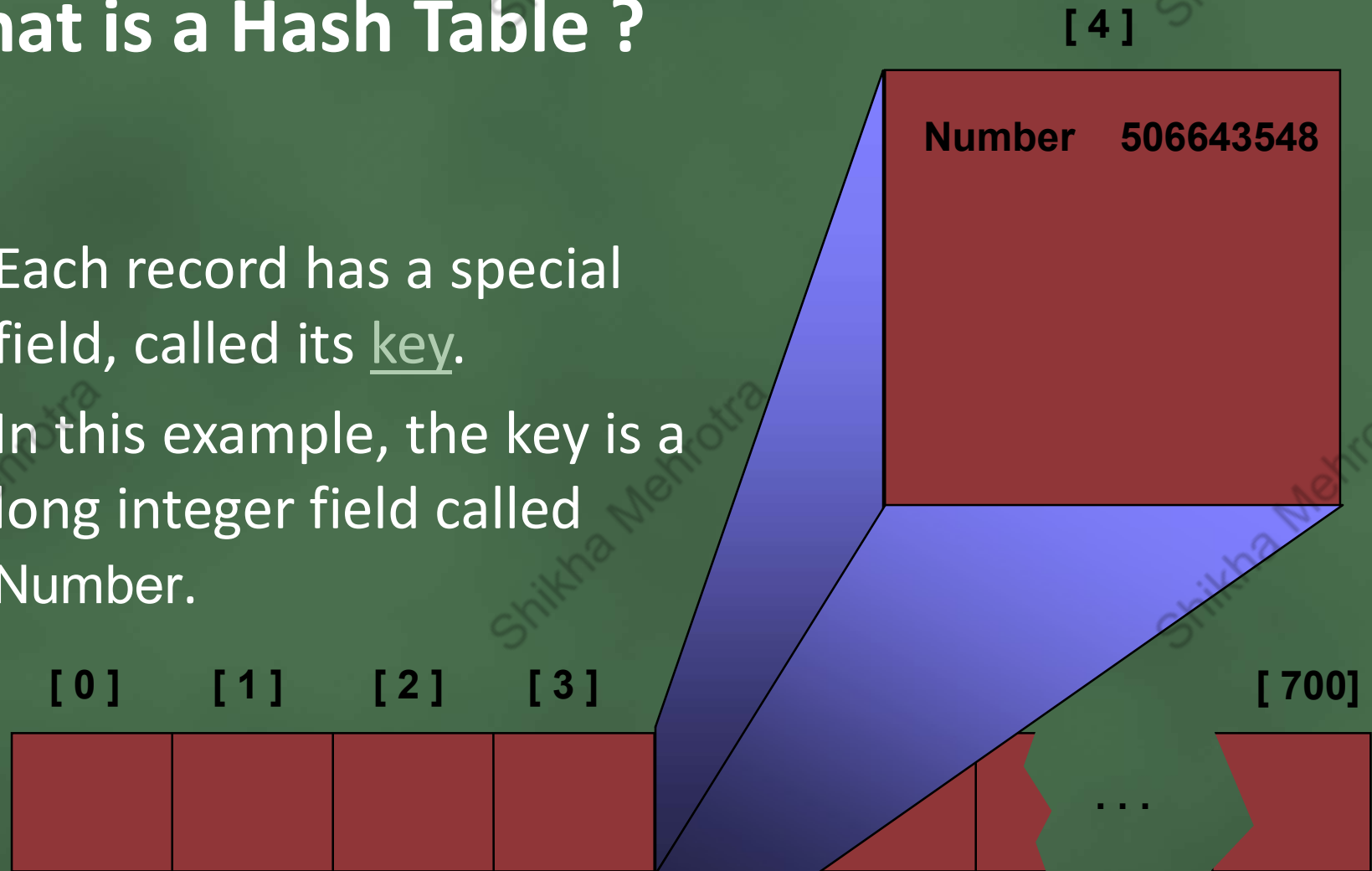
- The simplest kind of hash table is an array of records.
- This example has 701 records.





# •What is a Hash Table ?

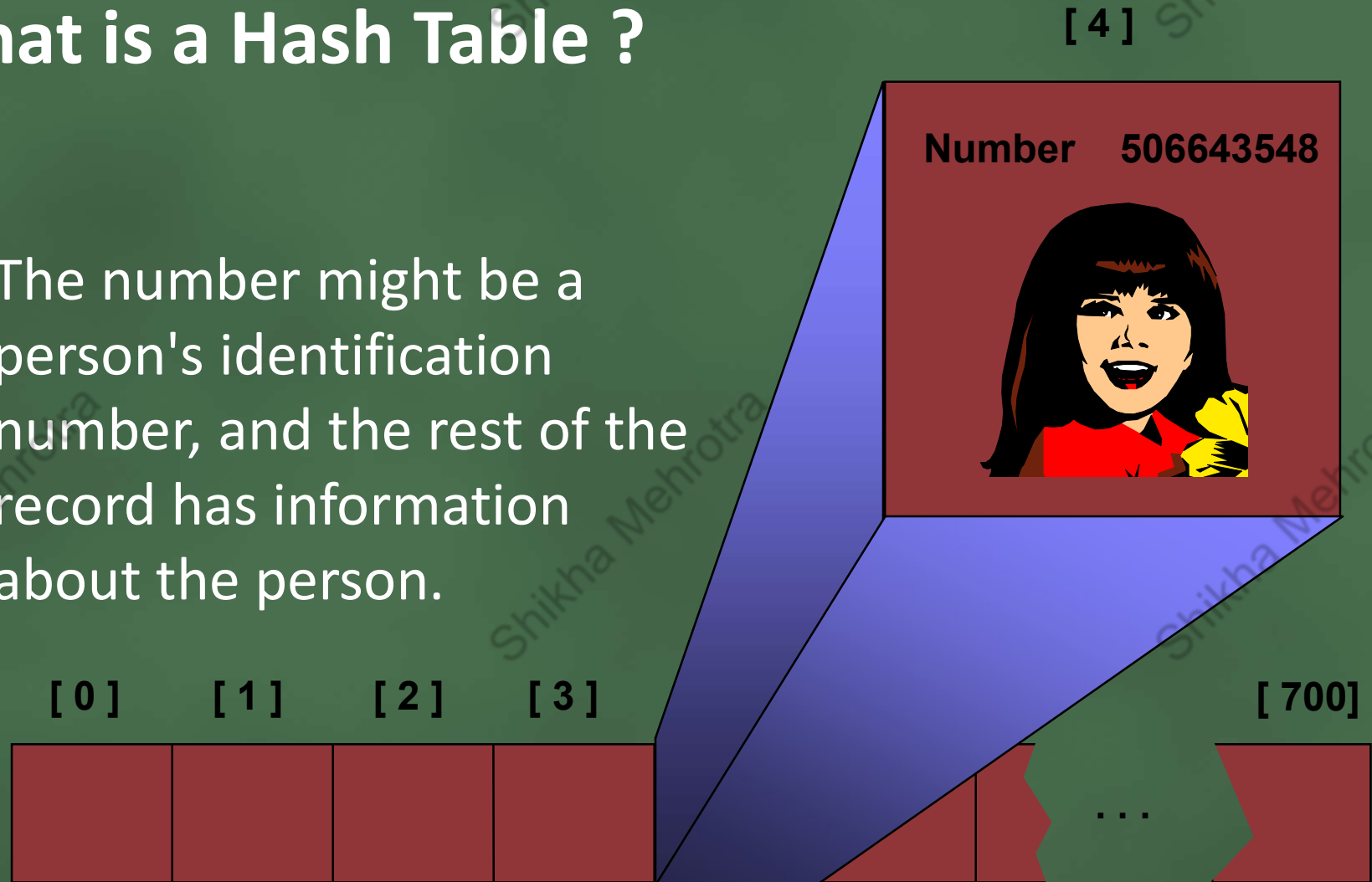
- Each record has a special field, called its key.
- In this example, the key is a long integer field called Number.





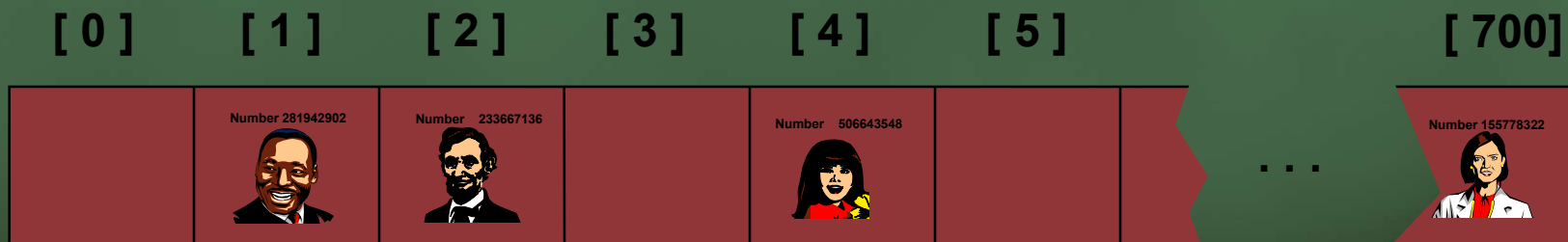
# •What is a Hash Table ?

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



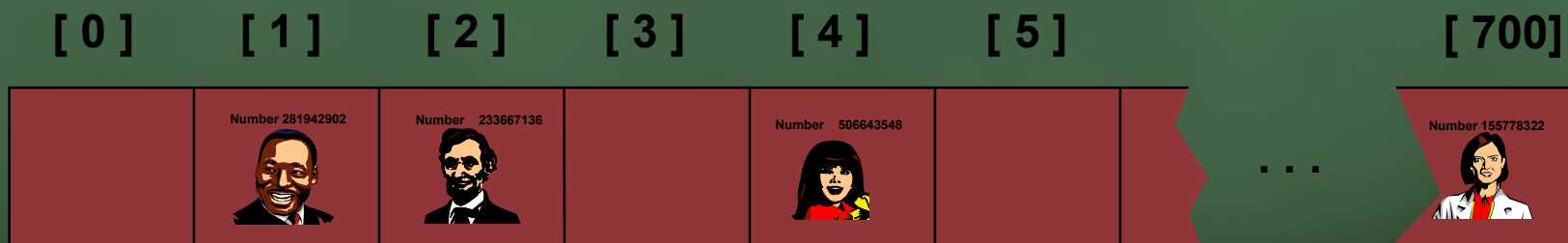
# •What is a Hash Table ?

- 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.

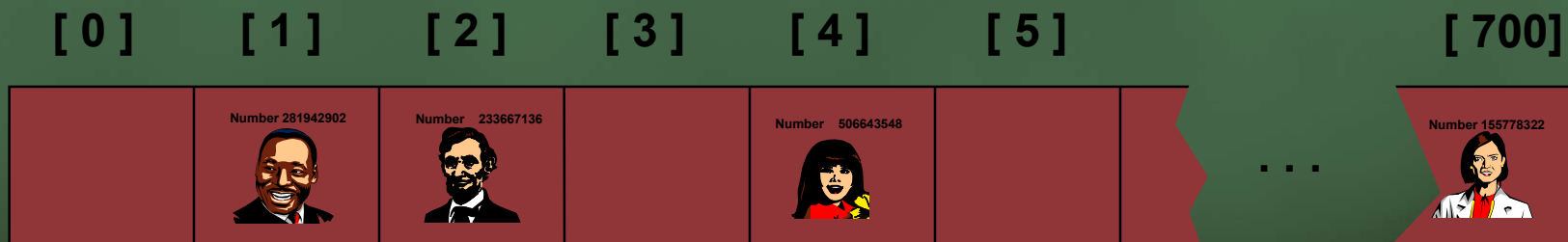


# • Inserting a New Record

- Typical way create a hash value:

(Number mod 701)

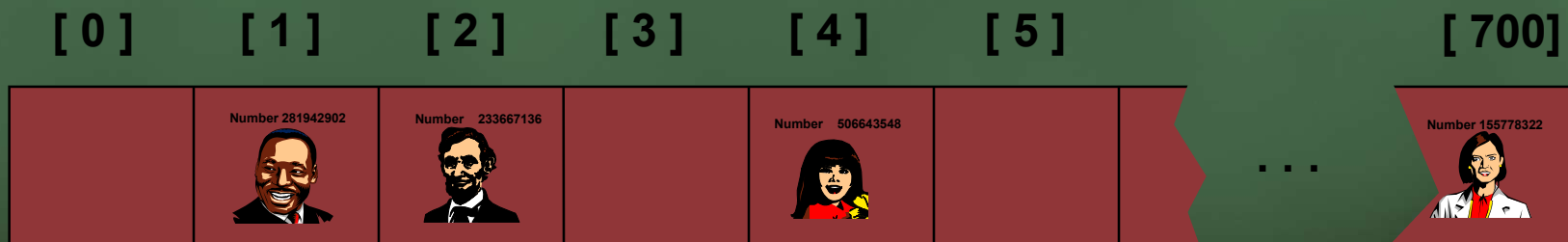
*What is  $(580625685 \% 701)$  ?*



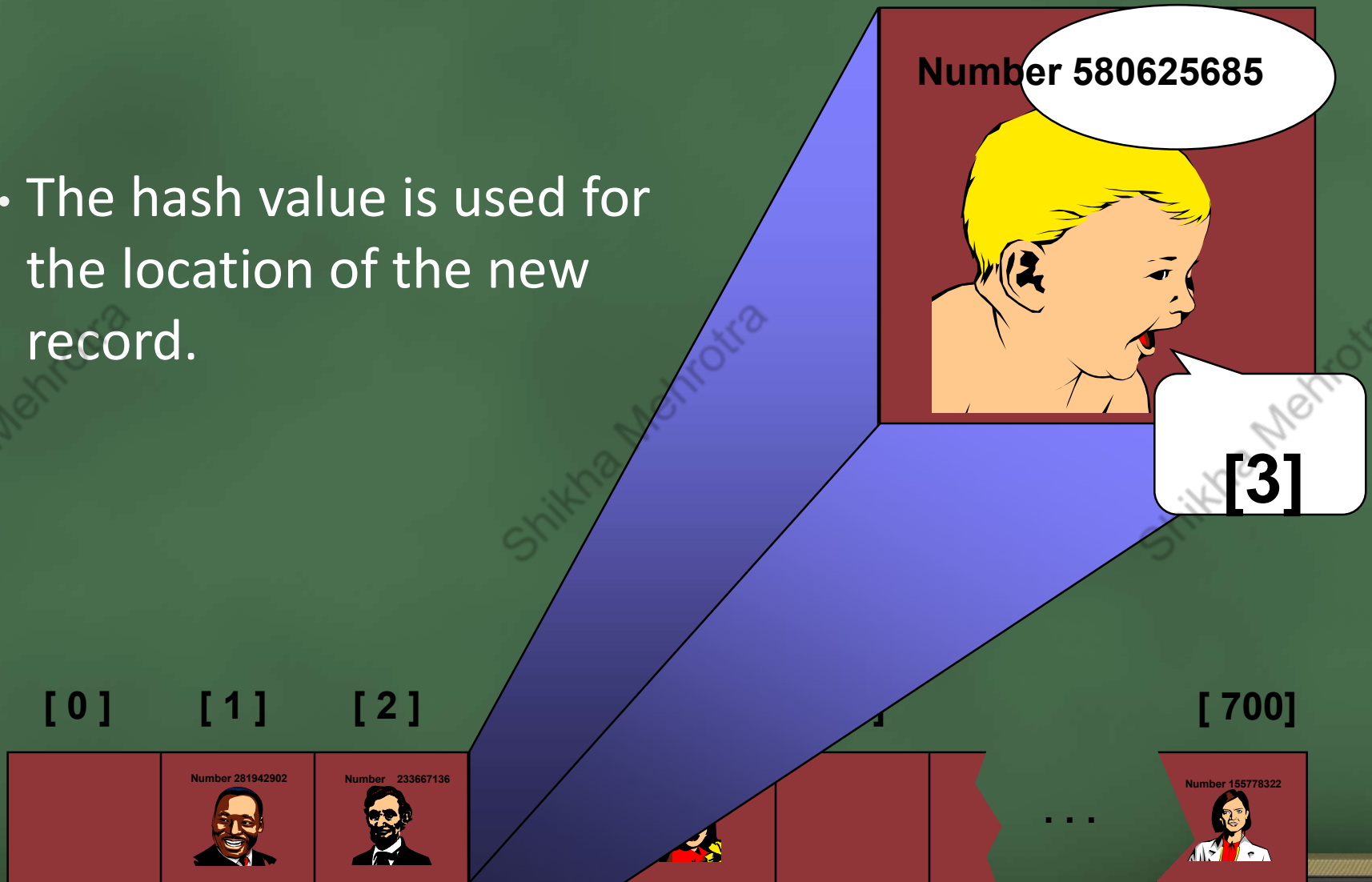
- Typical way to create a hash value:

(Number mod 701)

*What is  $(580625685 \% 701)$ ?*

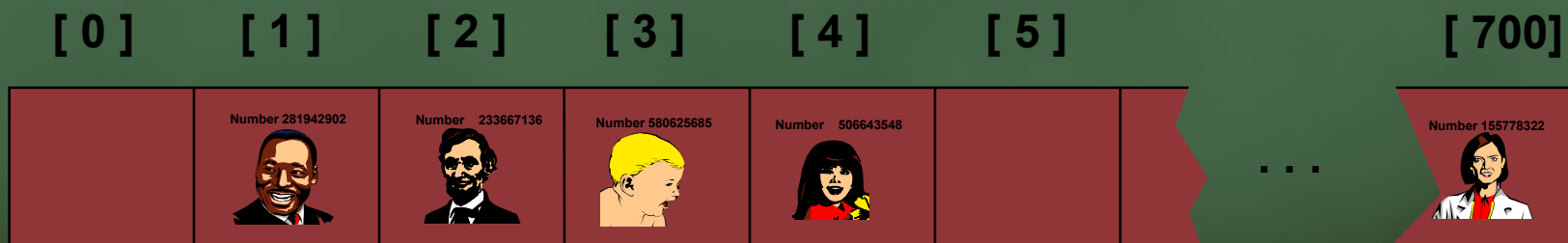


- The hash value is used for the location of the new record.



# • Inserting a New Record

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





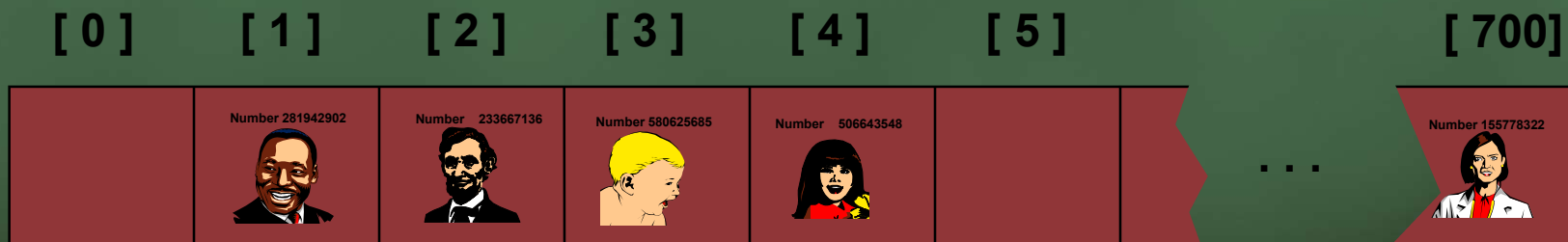
# • Collisions

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

Number 701466868



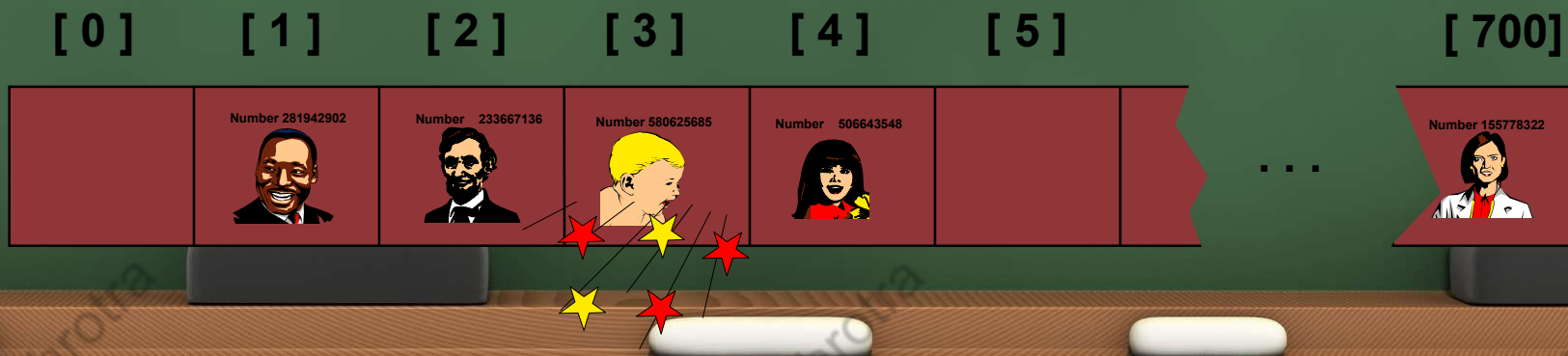
My hash value is [2].



# • 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.**

Number 701466868



[ 0 ]

[ 1 ]

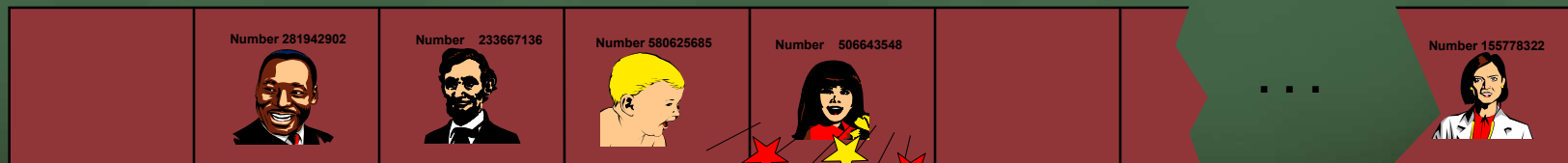
[ 2 ]

[ 3 ]

[ 4 ]

[ 5 ]

[ 700 ]



# • 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.**

Number 701466868



[ 0 ]

[ 1 ]

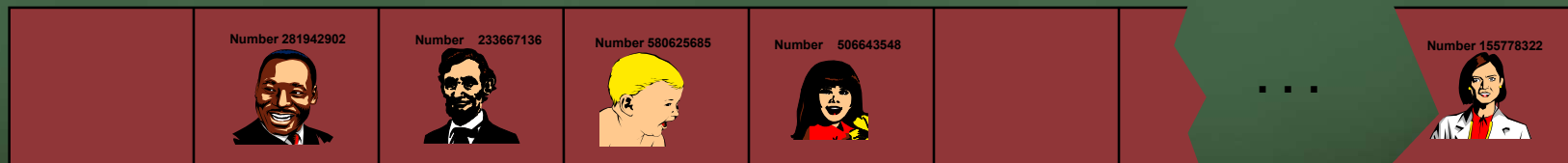
[ 2 ]

[ 3 ]

[ 4 ]

[ 5 ]

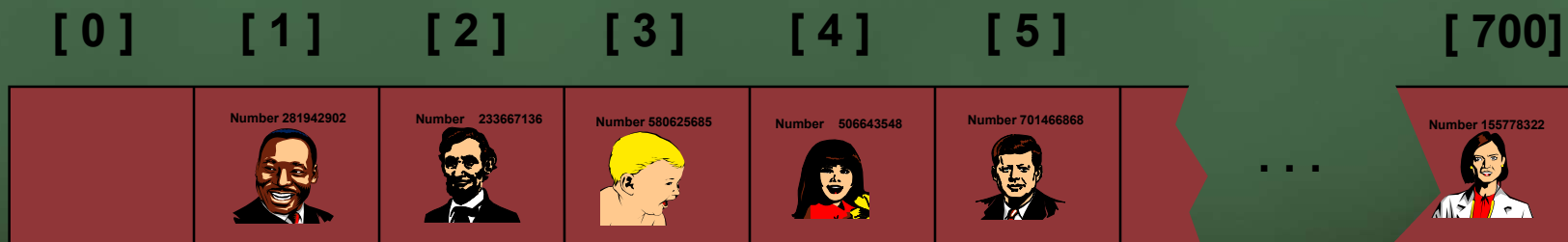
[ 700 ]



# • Collisions

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

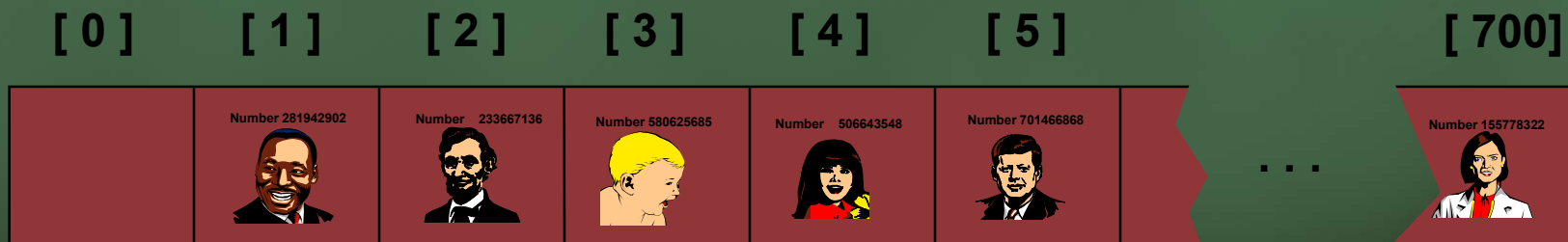
**The new record goes in the empty spot.**



# • Searching for a Key

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

Number 701466868

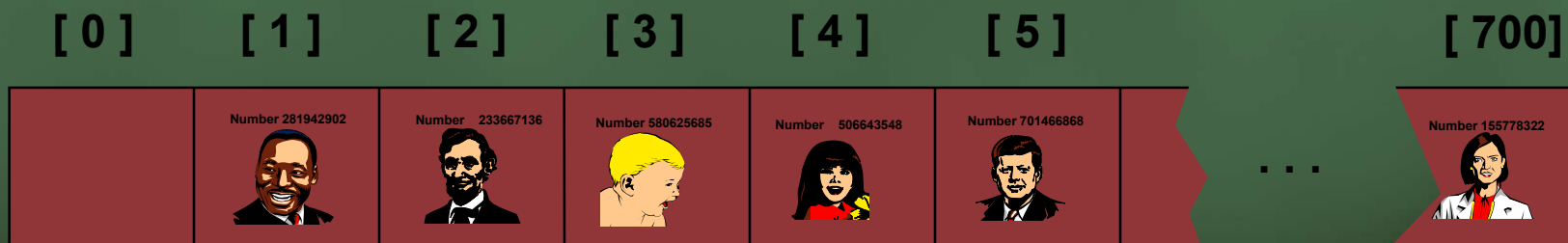


- Calculate the hash value.
- Check that location of the array for the key.

Number 701466868

My hash value is [2].

Not me.



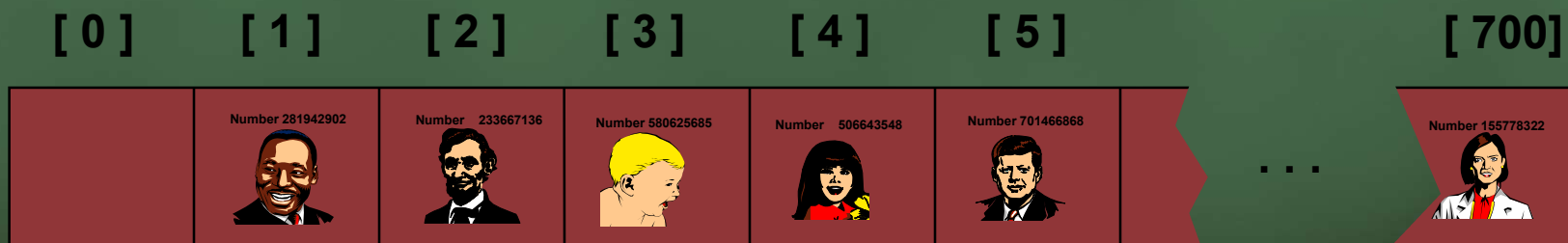


- Keep moving forward until you find the key, or you reach an empty spot.

Number 701466868

My hash value is [2].

Not me.

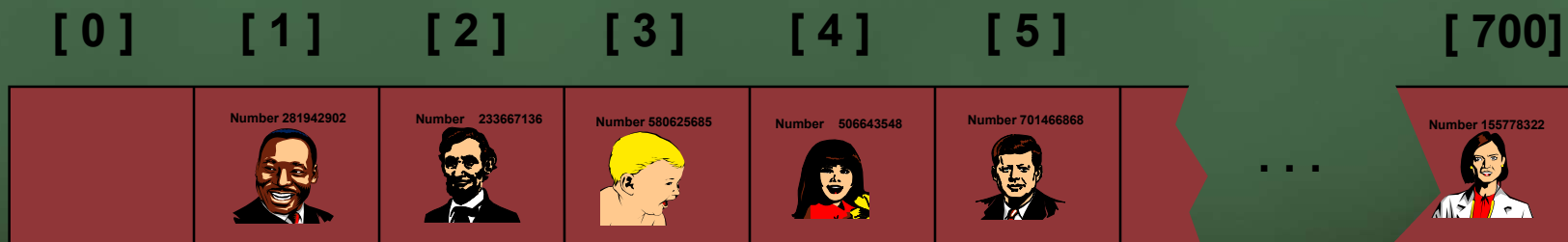


- Keep moving forward until you find the key, or you reach an empty spot.

Number 701466868

My hash value is [2].

Not me.

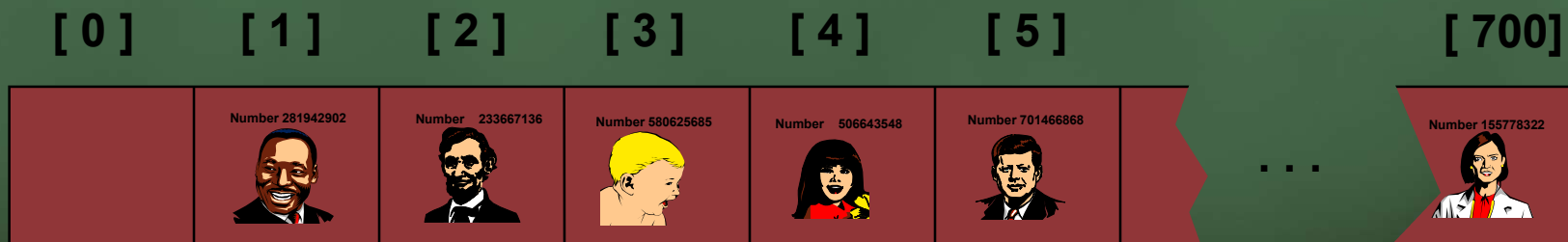


- Keep moving forward until you find the key, or you reach an empty spot.

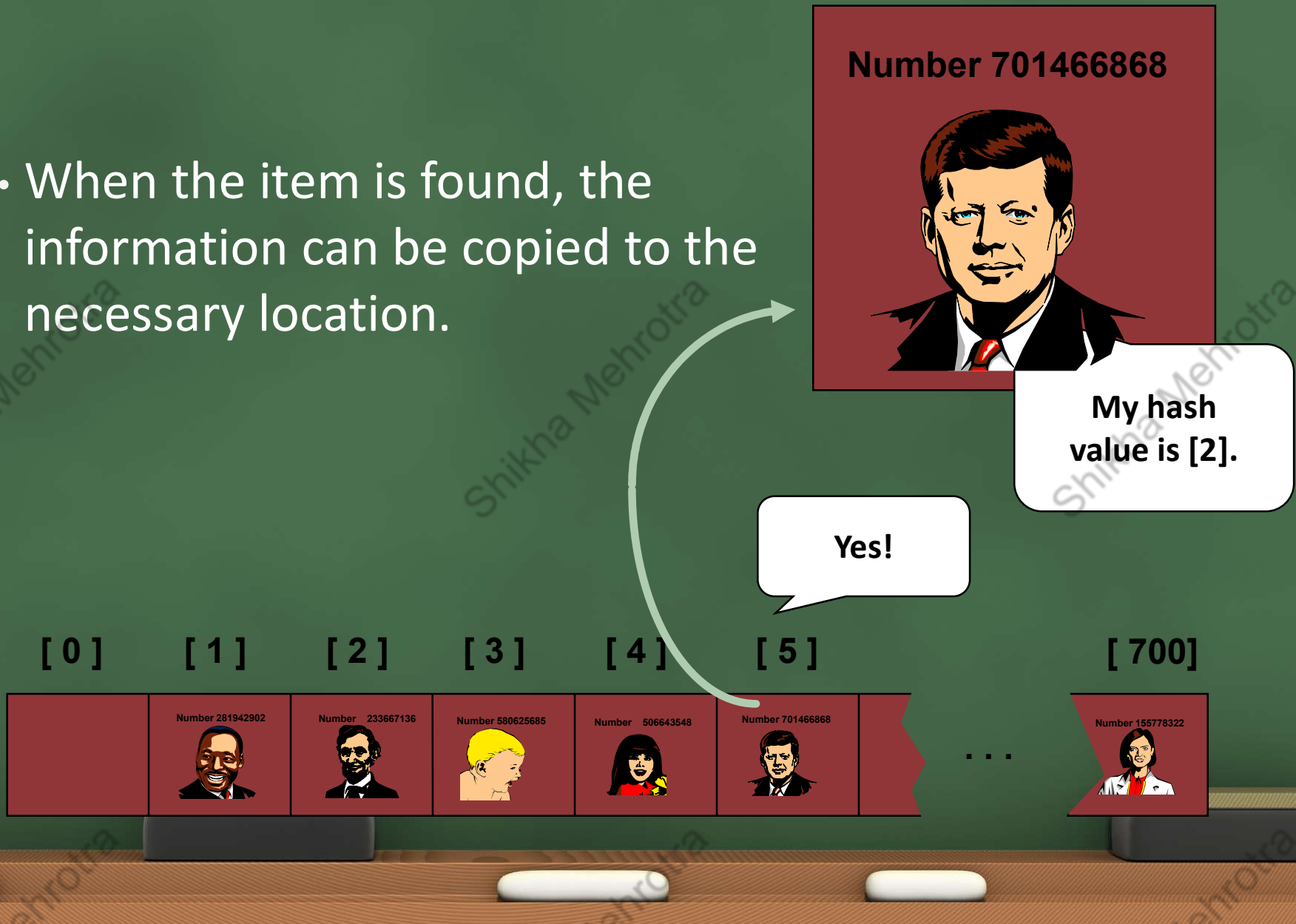
Number 701466868

My hash value is [2].

Yes!









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



# • Deleting a Record

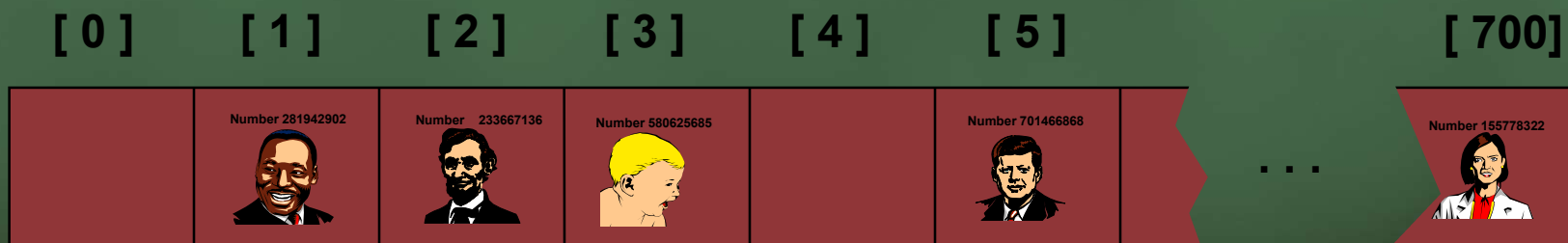
- Records may also be deleted from a hash table.

Please delete me.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	...	[ 700 ]
	Number 281942902 	Number 233667136 	Number 580625685 	Number 506643548 	Number 701466868 	...	Number 155778322 

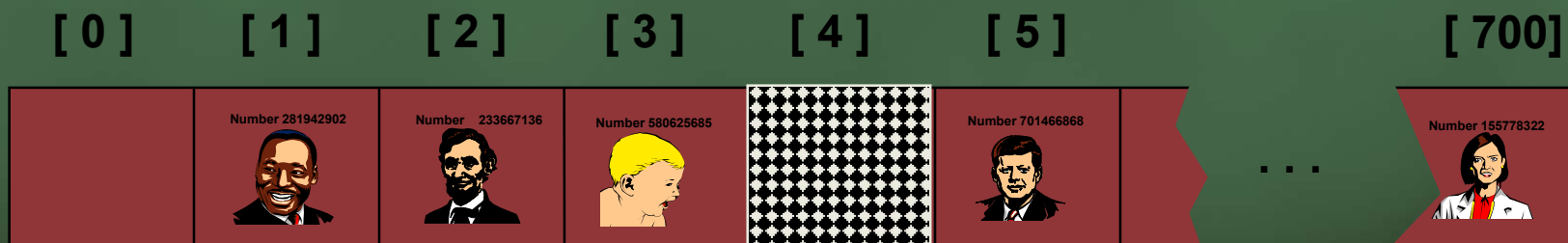
# •Deleting a Record

- 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.



# • Open Address Hashing

- To reduce collisions...
  - Use table CAPACITY = prime number of form  $4k+3$
  - Hashing functions:
    - Division hash function:  $\text{key} \% \text{CAPACITY}$
    - Mid-square function:  $(\text{key} * \text{key}) \% \text{CAPACITY}$
    - 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).

Quadratic probing  
Rehashing

# • 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  $\text{hash1}(\text{key})$  to determine insertion location  $i$  in array as before.*
2. *If collision occurs, use  $\text{hash2}(\text{key})$  to determine how far to move forward in the array looking for a vacant spot:*

$$\text{next location} = (i + \text{hash2}(\text{key})) \% \text{CAPACITY}$$



# • Double Hashing

- Clustering tends to be reduced, because  $\text{hash2}()$  has different values for keys that initially map to the same initial location via  $\text{hash1}()$ .
- 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
$$\text{hash2}(\text{key}) = (i+1)\% \text{CAPACITY}$$
- In double hashing  $\text{hash2}()$  can be a general function of the form
  - $\text{hash2}(\text{key}) = (1+f(\text{key}))\% \text{CAPACITY}$



## • Chained Hashing

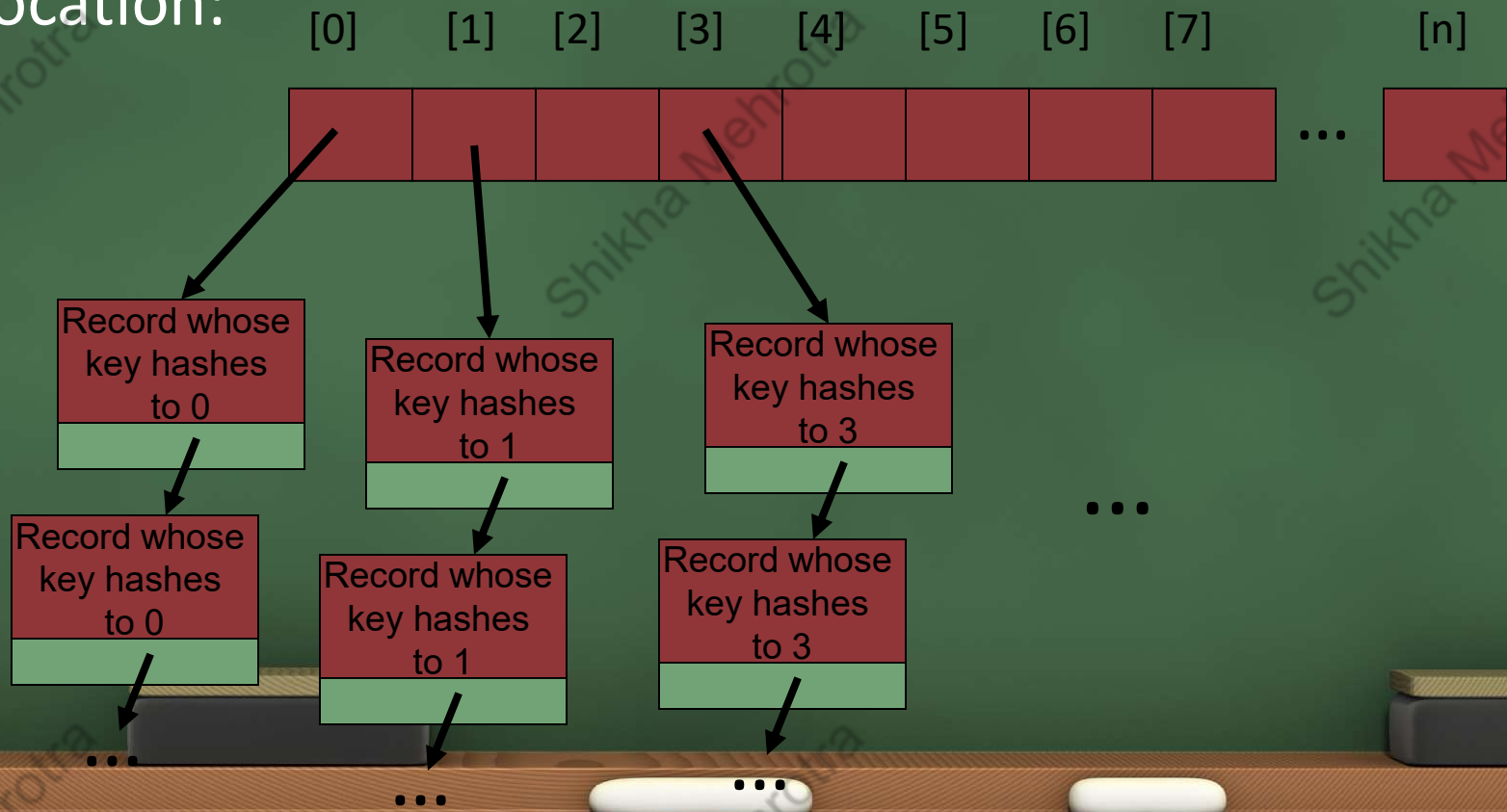
- In open address hashing, a collision is handled by probing the array for the next vacant spot.
- When the array is full, no new items can be added.
- We can solve this by resizing the table.
- Alternative: chained hashing.





# •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*:

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



## • Average Search Times

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

$$\frac{1}{2} (1 + 1/(1-\alpha))$$

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

Chained hashing:  $1 + \alpha/2$

## Average number of table elements examined during successful search

Load factor(a)	Open addressing, linear probing $\frac{1}{2} (1+1/(1-a))$	Open addressing double hashing $-\ln(1-a)/a$	Chained hashing $1+a/2$
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

## •Some Applications of Hash Tables

- **Database systems:** Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.
- **Symbol tables:** The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.
- **Data dictionaries:** Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.
- **Network processing algorithms:** Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.
- **Browser Cashes:** Hash tables are used to implement browser caches.
- **Password Storage:** Insecure passwords are hashed and stored securely.
- **Databases:** Hashing is used in indexing to retrieve data quickly.
- **Caching:** Web browsers use hashing to cache web pages for faster retrieval.
- **Checksum Algorithms:** Hashing helps in detecting data corruption."



## •Problems for Which Hash Tables are not Suitable

### 1. Problems for which data ordering is required.

Because a hash table is an unordered data structure, certain operations are difficult and expensive. Range queries, proximity queries, selection, and sorted traversals are possible only if the keys are copied into a sorted data structure. There are hash table implementations that keep the keys in order, but they are far from efficient.

### 2. Problems having **multidimensional data**.

### 3. **Prefix searching** especially if the keys are long and of variable-lengths.

### 4. **Problems that have dynamic data:**

Open-addressed hash tables are based on 1D-arrays, which are difficult to resize once they have been allocated. Unless you want to implement the table as a dynamic array and rehash all of the keys whenever the size changes. This is an incredibly expensive operation. An alternative is use a separate-chained hash tables or dynamic hashing.

### 5. **Problems in which the data does not have unique keys.**

Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is use separate-chained hash tables.

# • Summary

- Sequential 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  $\alpha$ .