Data Structures and Algorithms

Lecture 9: Searching, and Hashing

Outline of Today's Lecture

- Searching
  - Unsorted and Sorted Arrays
  - Self-Organizing Lists
  - Bit Vectors for Representing Sets
- Hashing
  - Hash Tables
  - Hash Functions
  - Open and Closed Hashing
  - Operations

Problem definition

- Suppose we have a collection L of n records of the form $(k_1, I_1), (k_2, I_2), \dots, (k_n, I_n)$, where I_j is information associated with key k_j from record $\left(k_j, I_j\right)$ for $1 \leq j \leq n$.
- Given a query K, the **Search Problem** is to locate a record (k_s, I_s) in L such that $k_s = K$ (if one exists).

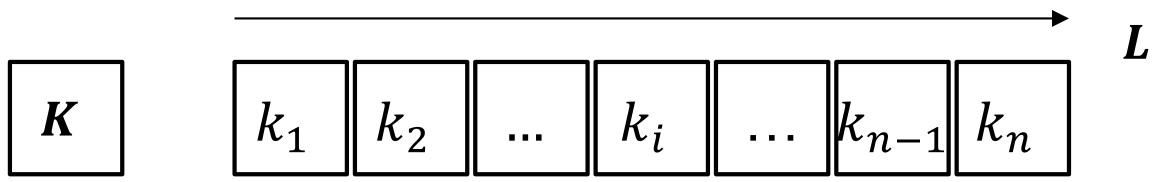
- Two types of **query** problems
  - An exact-match query is a search for the record whose key values matches a specified key value.
  - A range query is a search for all records whose key value falls within a specified range of key values.

Search algorithms

- Three general approaches
  - Sequential and list methods
  - Direct access by key value -- hashing
  - Tree indexing methods (next lecture)

Search in unsorted arrays (1/2)

- The **sequential search** algorithm
  - Basic idea: search from the beginning to the end
  - The simplest form of search



- Best case: $\Theta(1)$
- Worst case: $\Theta(n)$
- Average case: $\Theta(n/2) = \Theta(n)$
- Sometimes called linear search.

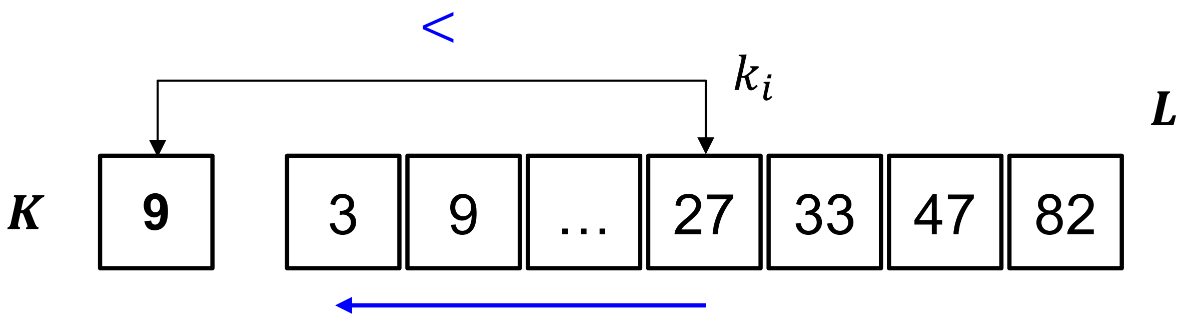
Search in unsorted arrays (2/2)

- A simple implementatin for sequential search

```
/* Find the position in A that holds value K, if any
does */
int sequential(int A[], int size, int K) {
  for (int i=1; i<size; i++) // For each element
    if (A[i] == K) // if we found it
      return i; // return this position
  return size; // Otherwise, return the array length
}
```

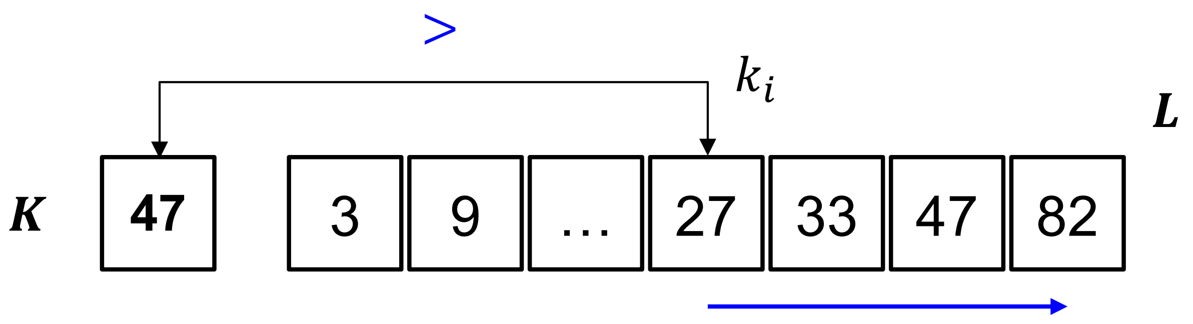
Search in sorted arrays

- Sequential search is somewhat slow. Θ(n)
- One way to reduce search time is to preprocess the records by sorting them.
- Given a sorted array, an obovious improvement over simple linear search is to test if the *current element* in L is greate than K.



Search in sorted arrays - Jump search (1/3)

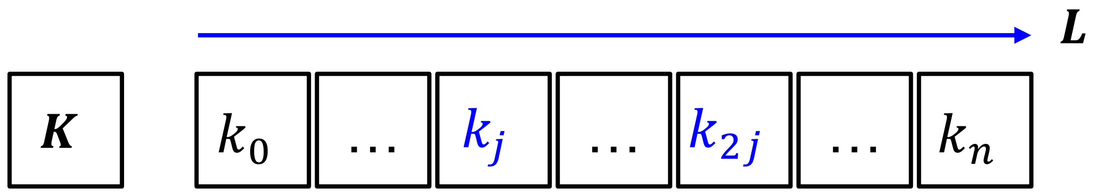
- Jump search
  - Suppose we look first position i and find that K is bigger, then we rule out position i as well as position 0 to $i-1$.



  - What if we carry this to the extreme and look first at the *last position* in *L* and find that *K* is bigger?
    - Then we know in one comparison that *K* is not in *L*.

Search in sorted arrays – Jump search (2/3)

- Basic idea of Jump search algorithm
  - For a jump size j, we check every j-th element in L.



  - So long as K is greater than the checking values, we continue on.
  - Otherwise, we do a linear search on the piece of length *j-1* that we know brackets *K* if it is in the list.
- A typical divide and conquer algorithm.
- What is the right amount to jump?

Search in sorted arrays - Jump search (3/3)

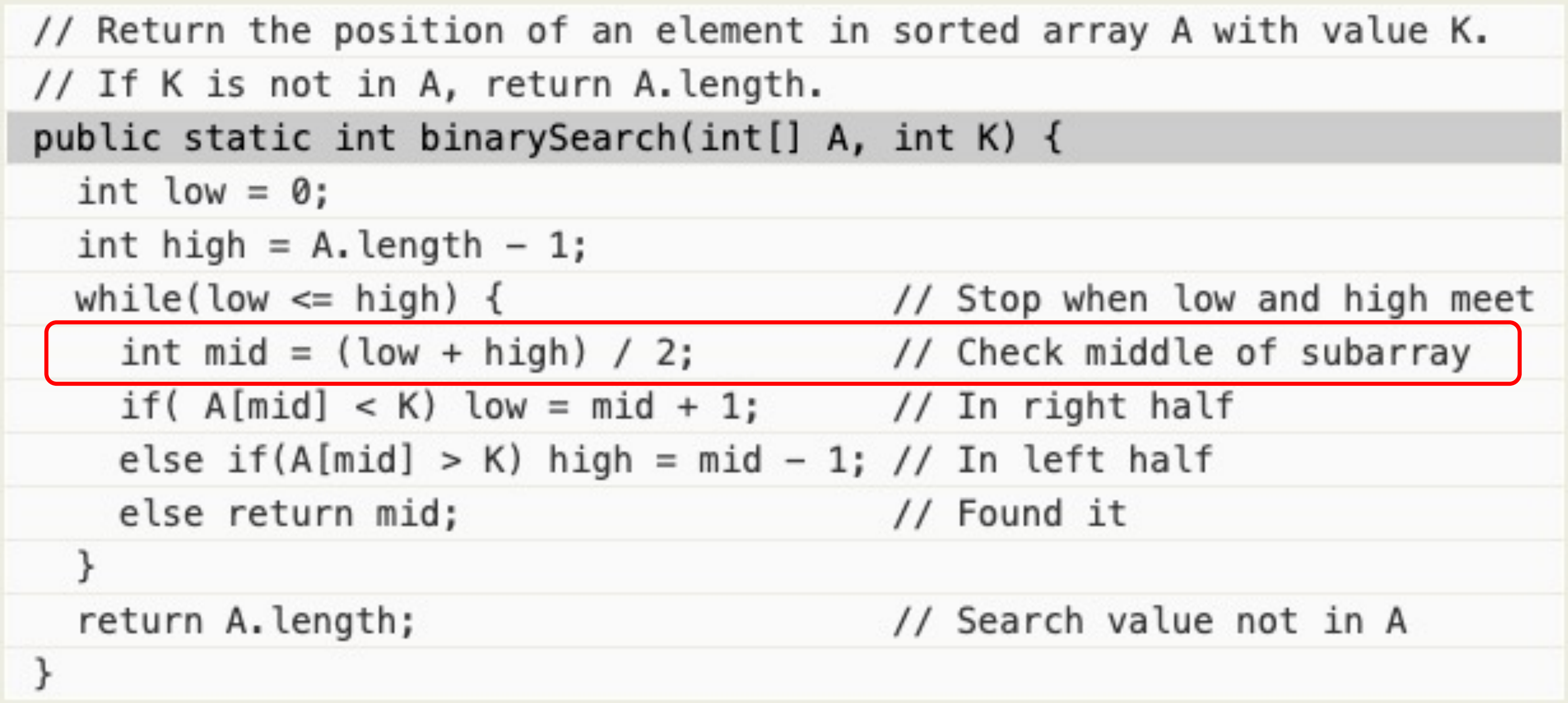
- Define m such that $mj \leq n < (m+1)j$, then the total cost of this algorithm is at most $m + j - 1$ 3-way comparison. (**3-way**: less, equal, or greater)
- Therefore, the cost to run the algorithm on n items with a jump of size j is

$$T(n,j) = m + j - 1 = \left\lfloor \frac{n}{j} \right\rfloor + j - 1$$

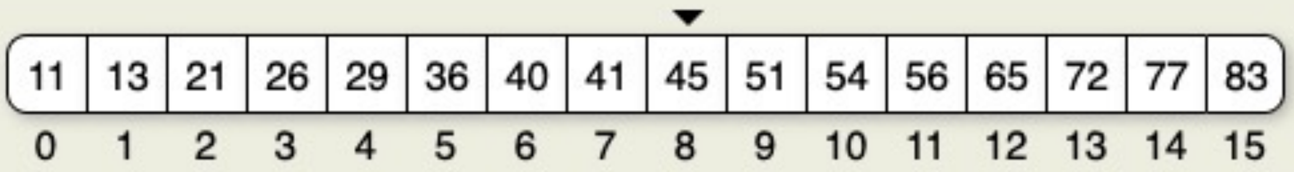
- Minimize the cost:
  - Take the derivative and solve for $T'(n,j) = 0$ to find the minimum, which is $j = \sqrt{n}$.
- In this case, the worst case cost is roughly $2\sqrt{n}$.

Search in sorted arrays - Binary search (1/3)

- Basic idea: recursion



```
// Return the position of an element in sorted array A with value K.
// If K is not in A, return A.length.
public static int binarySearch(int[] A, int K) {
  int low = 0;
  int high = A.length - 1;
  while(low <= high) {                        // Stop when low and high meet
    int mid = (low + high) / 2;               // Check middle of subarray
    if( A[mid] < K) low = mid + 1;            // In right half
    else if(A[mid] > K) high = mid - 1;       // In left half
    else return mid;                          // Found it
  }
  return A.length;                            // Search value not in A
}
```



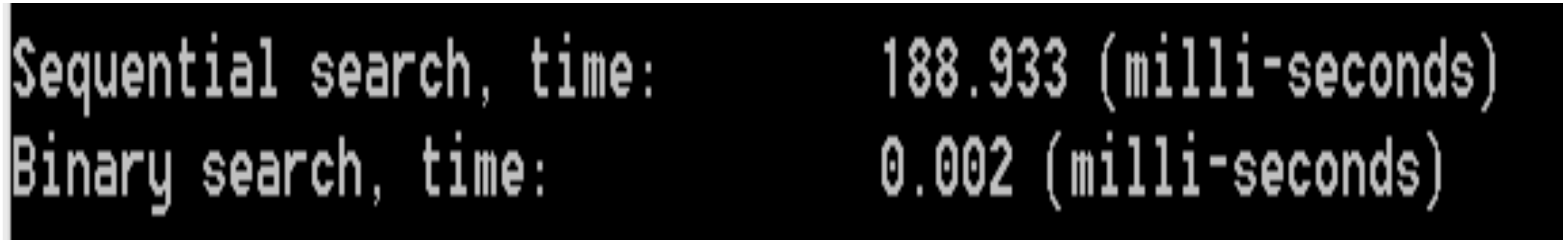
Search in sorted arrays - Binary search (2/3)

- An optimal algorithm for a sorted list.
- Time complexity: $\Theta(\log n)$
- If the data are not sorted, using binary search requires to pay the cost of soring the data, e.g., $\Theta(\log n)$ by a balanced binary search tree (BST).

- Two special forms of binary search (see book):
  - Dictionary or interpolation search
  - Quadratic binary search

Performance Comparison (n=400M)

- Running time of the sequential search is about 200 ms
- Running time of the binary search is only 0.002 ms
- Binary search is about 100,000 times faster for n=400M



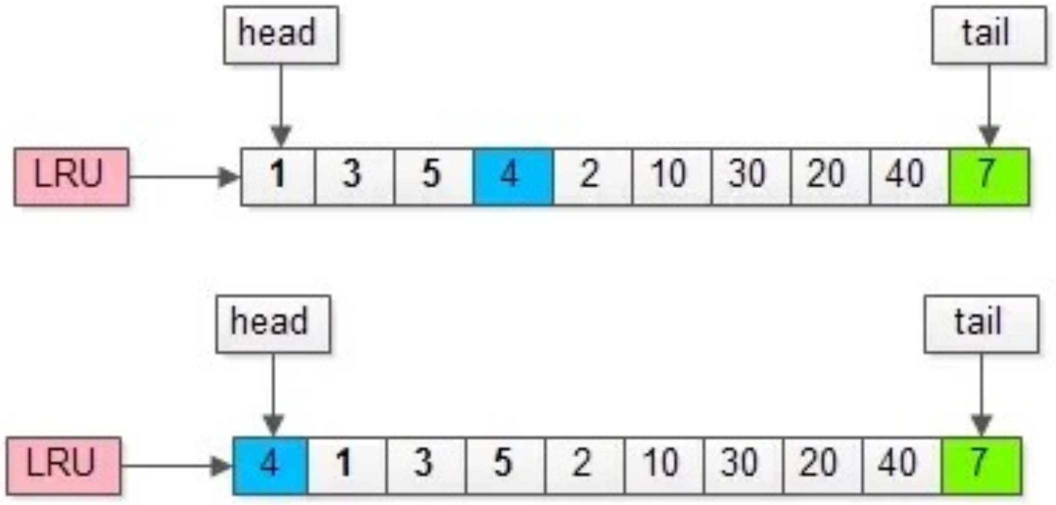
```
Sequential search, time:          188.933 (milli-seconds)
Binary search, time:              0.002 (milli-seconds)
```

Self-organizing lists (1/6)

- Self-organizing lists are simply linked-lists of data
  - It reorganizes the data such that items that have been accessed recently or more frequently, are moved closer to the front of the list.
- Motivation for sorting by access frequency
  - Most searchable data sets contain some items that are accedded frequently, and many items that are accessed rarely.
  - Can speed up sequential search.

Self-organizing lists (2/6)

- Usually, the frequencies are unknown.
- Self-organizing lists use a heuristic for deciding how to redorder the list.
  - Similar to the rules for managing buffer pools. E.g.,

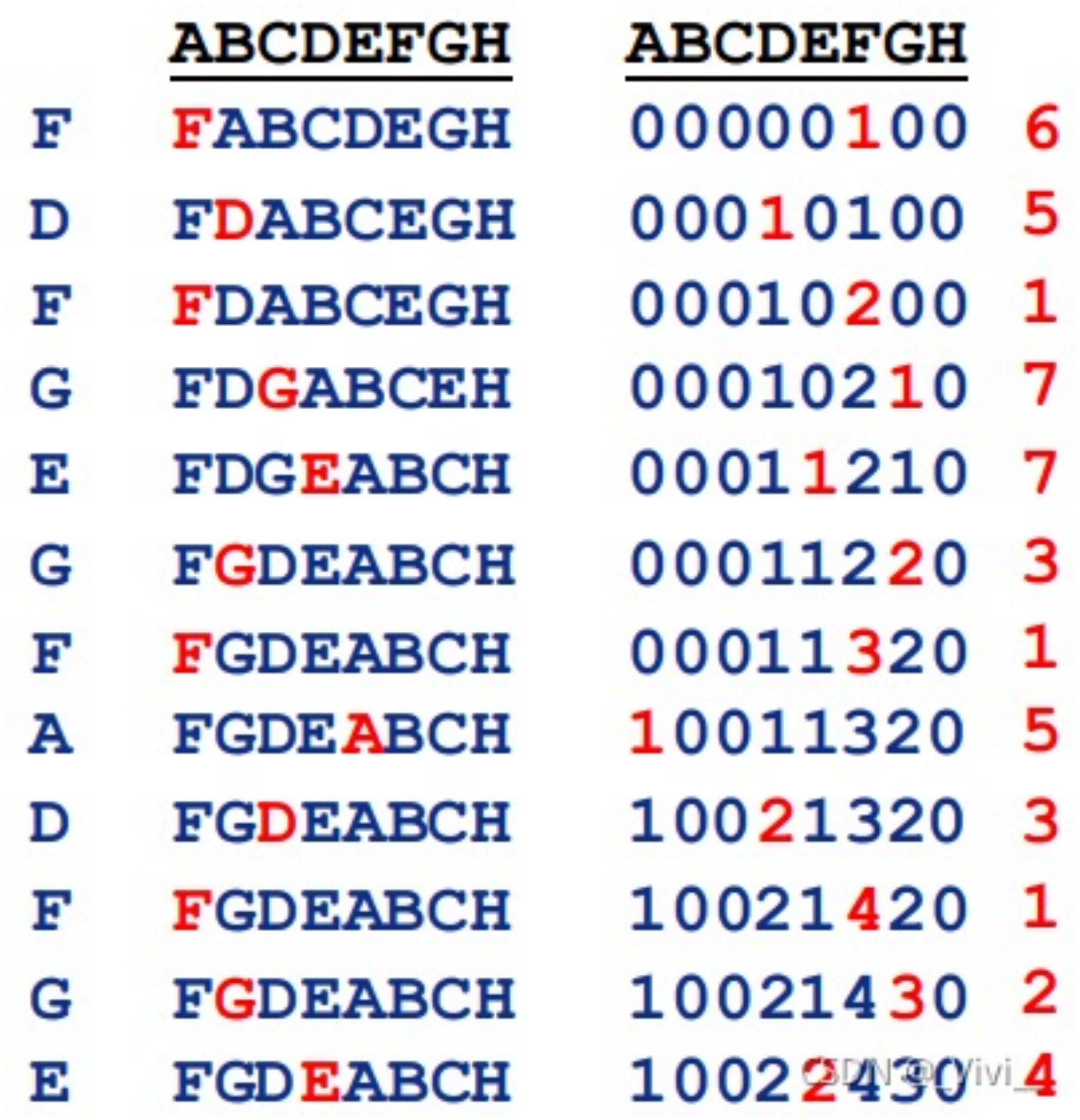


Self-organizing lists (3/6)

- Basic ideas
  - modify the order of records within the list based on the actual pattern of record access, by moving a found key nearer to the front of the list (insert and delete operations can stay the same).
- We consider three heuristics
  - Frequency Count
  - Move-To-Front
  - Transpose

Self-organizing lists (4/6)

- Frequency Count
  - When a record is found, move forward the front of the list if its number of accesses becomes greater than a record preceding it.



	ABCDEFGH	ABCDEFGH	
F	FABCDEGH	00000100	6
D	FDABCEGH	00010100	5
F	FDABCEGH	00010200	1
G	FDGABCEH	00010210	7
E	FDGEABCH	00011210	7
G	FGDEABCH	00011220	3
F	FGDEABCH	00011320	1
A	FGDEABCH	10011320	5
D	FGDEABCH	10021320	3
F	FGDEABCH	10021420	1
G	FGDEABCH	10021430	2
E	FGDEABCH	10022430	4

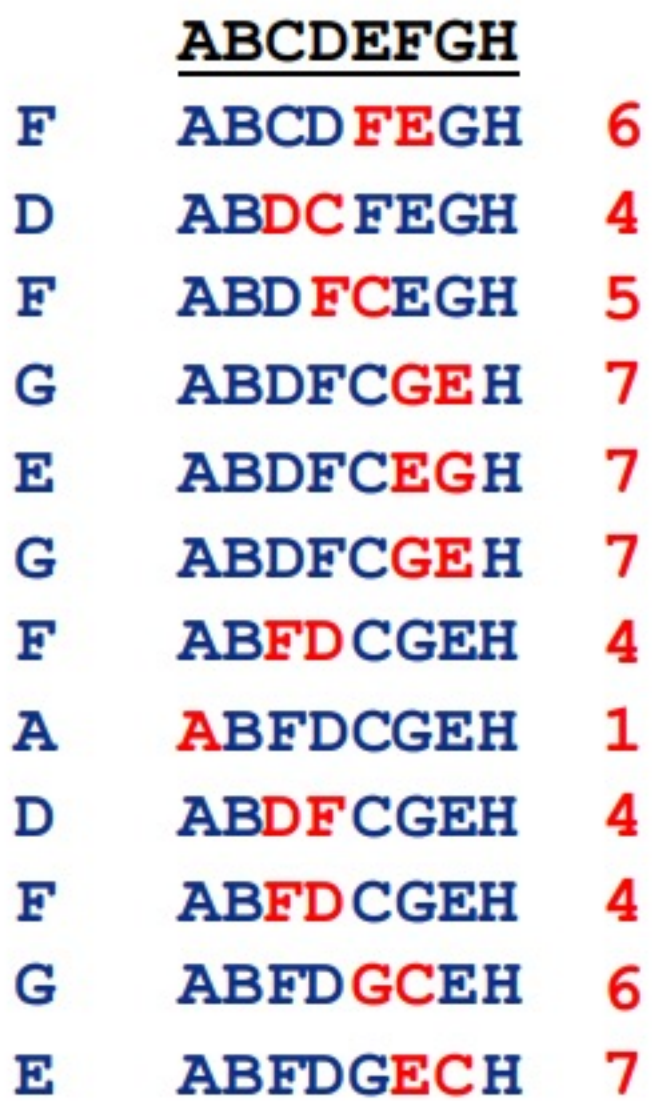
Self-organizing lists (5/6)

- Move-To-Front
  - When a record is found, move it to the front of the list.

	ABCDEFGH	
F	FABCDEGH	6
D	DFABCEGH	5
F	FDABCEGH	2
G	GFDABCEH	7
E	EGFDABCH	7
G	GEFDABCH	2
F	FGEDABCH	3
A	AFGEDBCH	5
D	DAFGEBCH	5
F	FDAGEBCH	3
G	GFDAEBCH	4
E	EGFDABCH	5

Self-organizing lists (6/6)

- Transpose
  - When a record is found, swap it with the record ahead of it.



	ABCDEFGH	
F	ABCDFEGH	6
D	ABDCFEGH	4
F	ABDFCEGH	5
G	ABDFCGEH	7
E	ABDFCEGH	7
G	ABDFCGEH	7
F	ABFDCGEH	4
A	ABFDCGEH	1
D	ABDFCGEH	4
F	ABFDCGEH	4
G	ABFDGCEH	6
E	ABFDGECH	7

Example: self-organizing lists

- Text compression and transmission
  - By the move-to-front rule

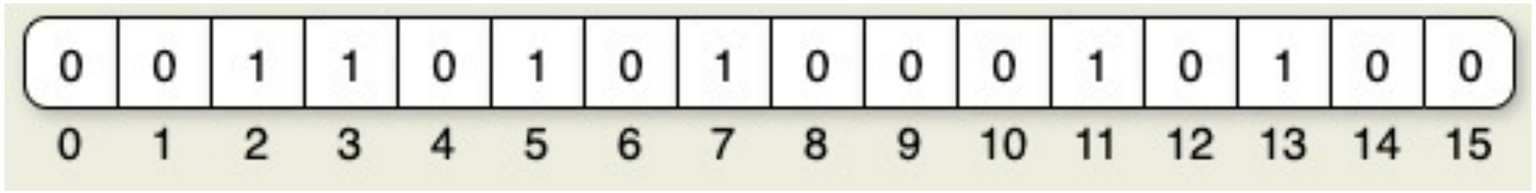
1. If the word has been seen before, transmit the current position of the word in the list. Move the word to the front of the list.
2. If the word is seen for the first time, transmit the word. Place the word at the front of the list.

```
The car on the left hit the car I left

The car on 3 left hit 3 5 I 5
```

Bit vectors for representing sets

- Representing sets using a bit array with a bit position allocated for each potential member.
  - **1** denotes 'in the set'; **0** denote 'not in the set'.
- Example: a set of primes



0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- Benefits by the logical bit-wise operations
  - set union, intersection, and difference

Hashing

Hashing

Given ***n*** records with unique **keys**,

	`insert`	`search`	`delete`
Unsorted list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Balanced BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Magic array	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Sufficient "**magic**":

- Use **key** to map array index for a record in $\Theta(1)$ time
- Search by direct access based on key value

Data collection

- Data collection is a set of records (static or dynamic)
- Each record consists of two parts
  - A key: a unique identifier of the record.
  - Data item: it can be arbitrarily complex.
- The key is usually a number, but can be a string or any other data type.
  - Non-numbers are converted to numbers when applying hashing.

Basic ideas of hashing

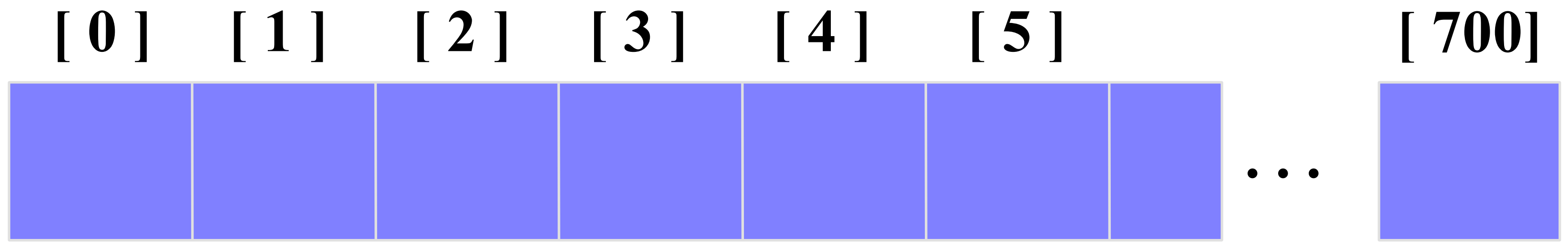
- Use *hash function* to map keys into positions in a *hash table*

Ideally

- If data item or 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/item, dictionary does not contain *e*.

A simple Hash Table

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



An array of cells

Following the example

- We want to store a dictionary of Object Records, no more than 701 objects
- Keys are Object ID numbers, e.g., 506643548
- Hash function: h(k) maps k(=ID) into distinct table positions 0-700
- Operations: insert, delete, and search

Complexity (ideal case)

- Why hashing and hash table?
  - It is very efficient.

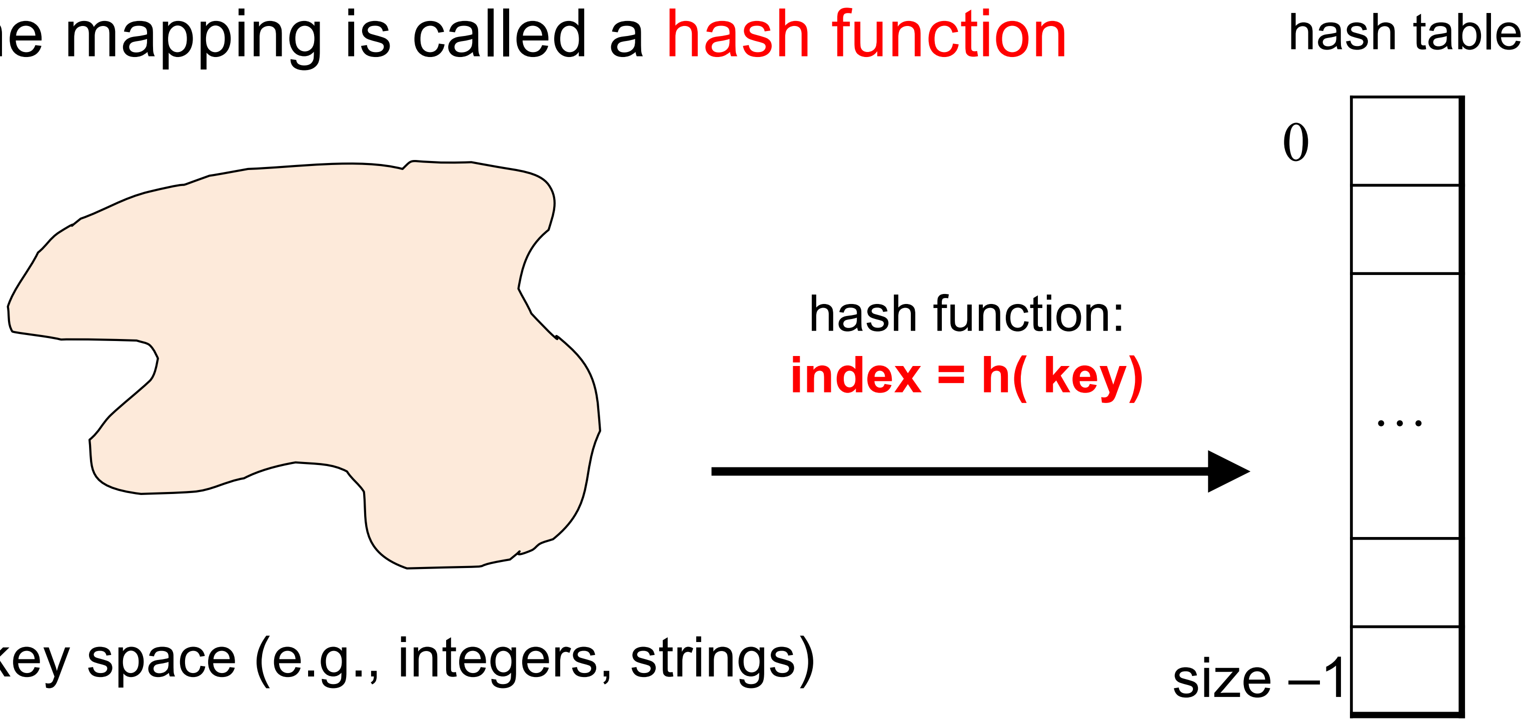
- O(D) time to initialize hash table (D number of positions or cells in hash table)
- O(1) time to perform *insert, remove, search*

Limitations of the Simple Hash Table

1. The maximum number in array A must be ≤ c*n, where c is constant
   - Otherwise, the hash table may be too large
2. Keys must be integers
   - But may be strings, real numbers, etc. in a real application

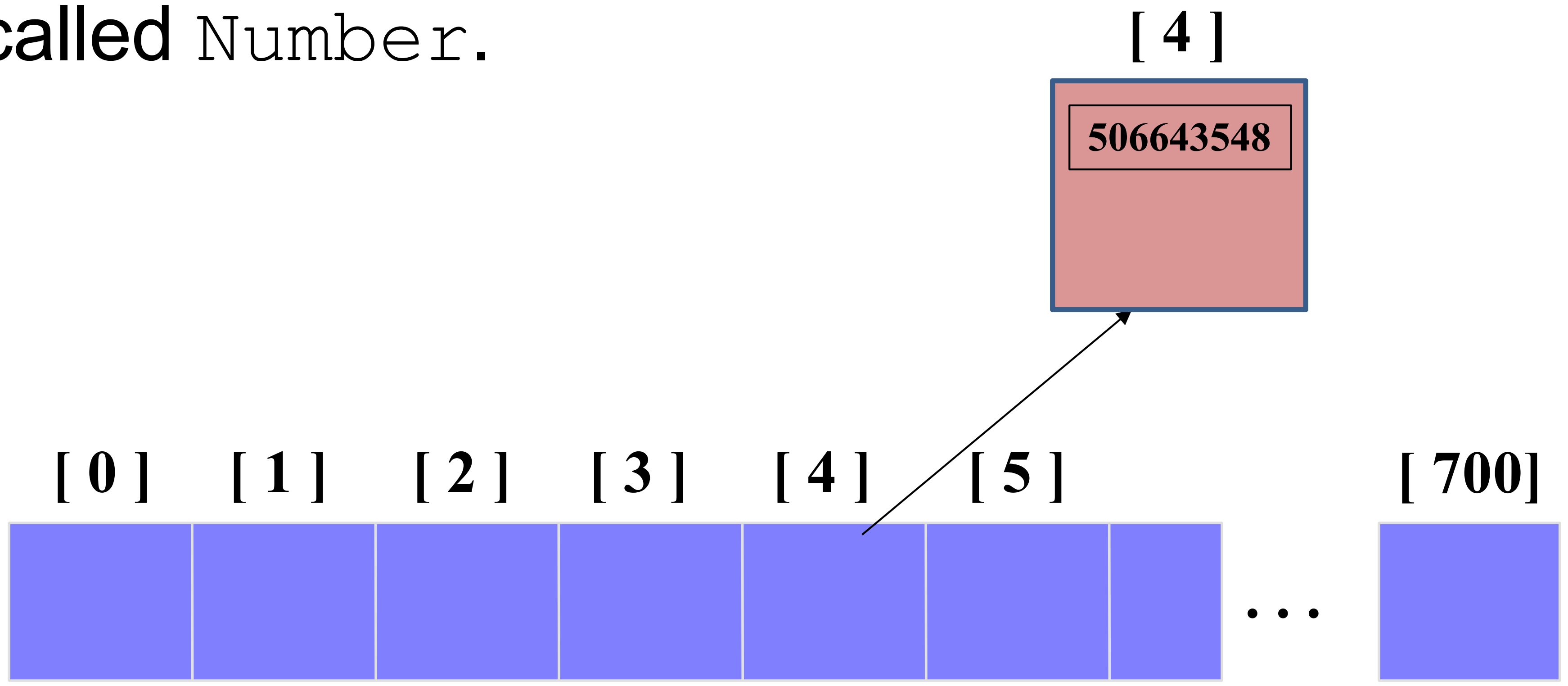
Hash Table and Hash Function

- A hash table is an array of some fixed size, storing the records
- Each key is mapped into some *location* in the range 0 to size-1 in the table
- The mapping is called a hash function



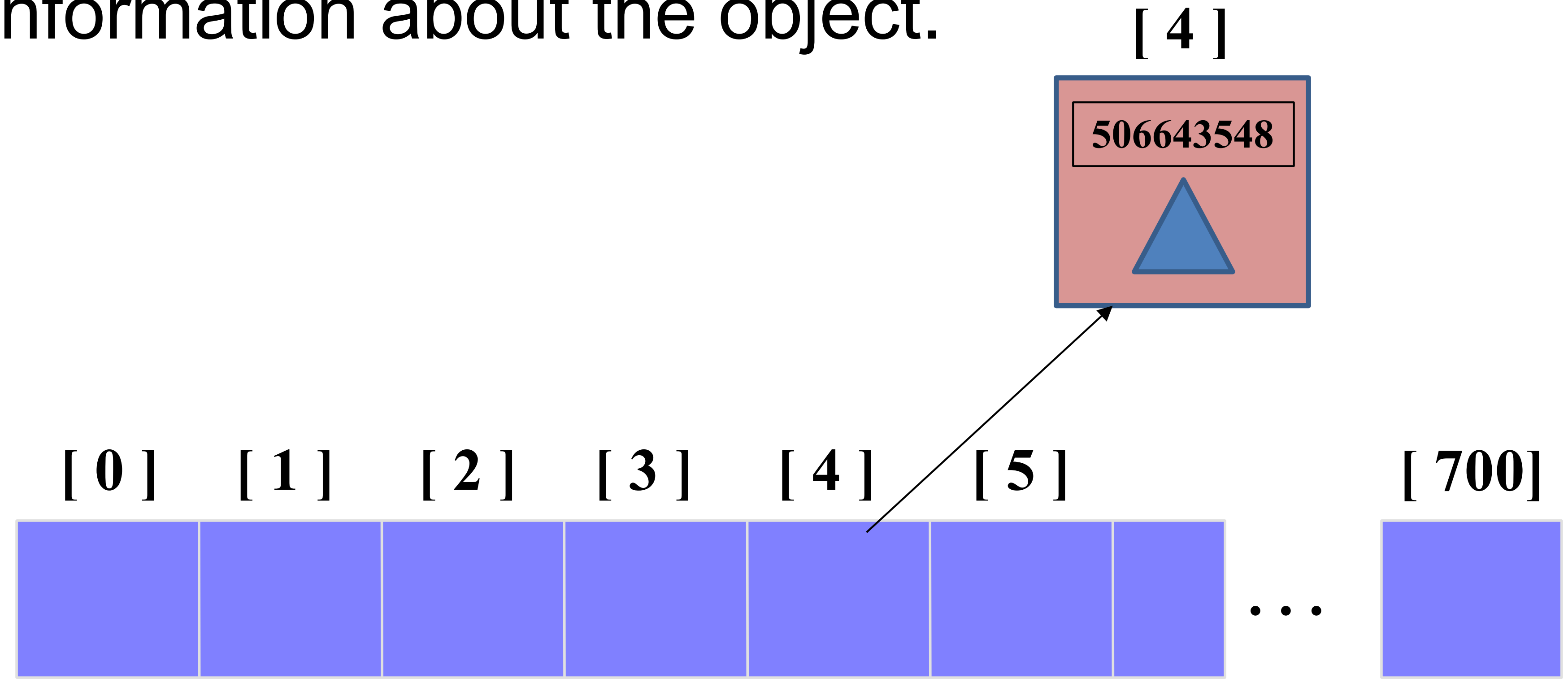
Use the Hash Table

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



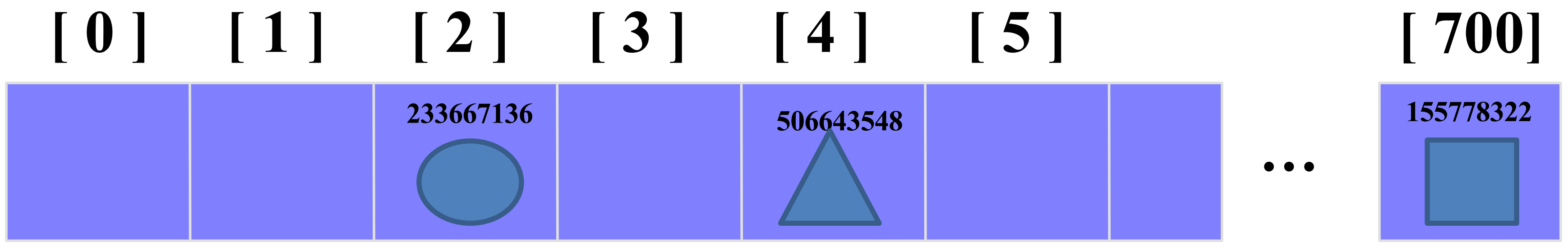
Use the Hash Table

- The number is a object's identification number, and the rest of the record has information about the object.



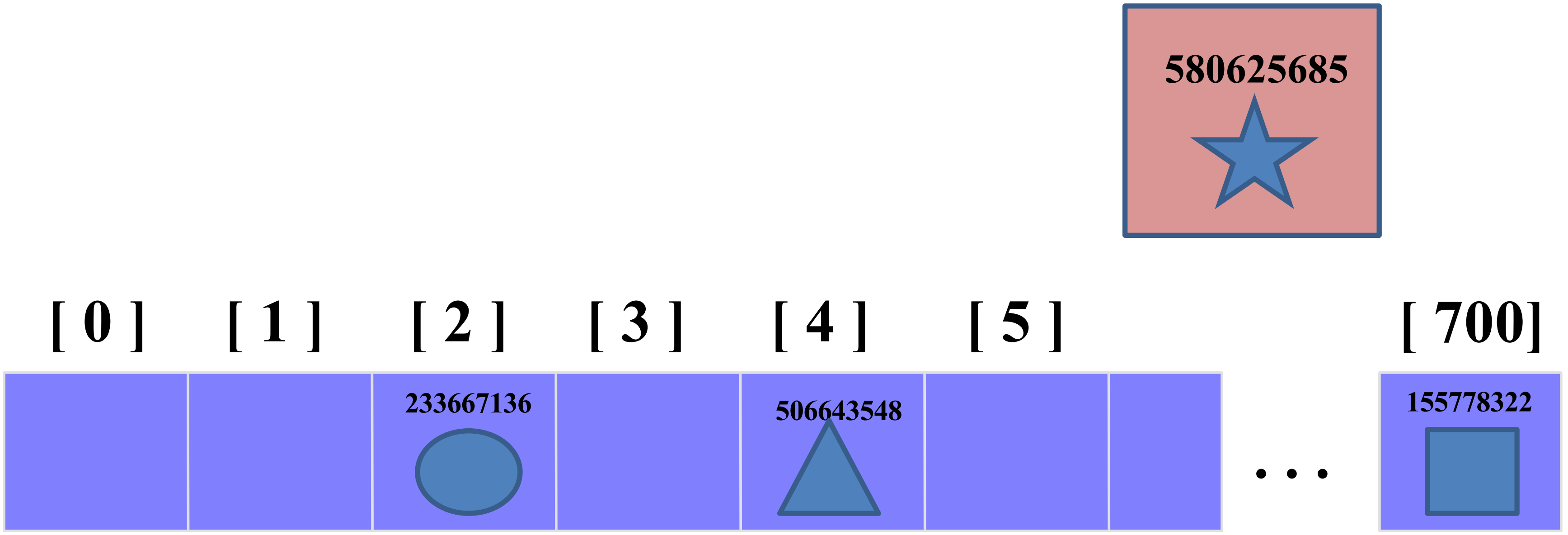
Use the Hash Table

- 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 **mapped to** an array **index** using a **hash function**.
- The index is called the **hash value** of the key.



Three design considerations of hash

1. Design a general hash function h(K) that maps a record with key K to a location in hash table
2. Given any two records with keys K_1 and K_2, the probability that they are mapped to the same location in the hash table should be as small as possible, i.e.,
   - Pr[h(K_1) = h(K_1)] is very small
   - Otherwise, many records are mapped to the same location, which is called a collision
3. Solve the collision problem

Hash functions

- Popular hash functions: hashing by division

  h(k) = k mod D, where D is number of cells in hash table

- Example: hash table with 701 cells

  h(k) = k mod 701

  h(80) = 80 mod 701 = 80

  h(1000) = 1000 mod 701 = 299

Hash Function design – a simple mod function

- Consider n=5 keys
  - A[5]=11, 35, 54, 99, 42
- Allocate an array Table[10] with size M=10
- Hash function h(key) = key % 10
- Place 11 at location 11%10=1 in hash table

		11	**42**		**54**	**35**				**99**
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

- But there may be many collisions
- Consider other 5 keys
  - B[5]=11, 21, 31, 41, 51
  - Each key is mapped to location 1

Hash Function design – a better hash function

- Consider n=5 keys
  - B[5]=11, 21, 31, 41, 51
- Allocate an array Table[10] with size M=10
- Hash function by **mid-square**, given a key ***K*****,**
  - Location is the middle *r* digits of value K^2
  - $11^2=121$, $21^2=441$, $31^2=961$, $41^2=1681$, $51^2=2601$
  - Consider the middle digit, i.e., $r=1$

	51		**11**		**21**		**31**		**41**	
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

- The location is correlated with all digits in the key, not just the lowest digit.

Hash function for a string-A simple way

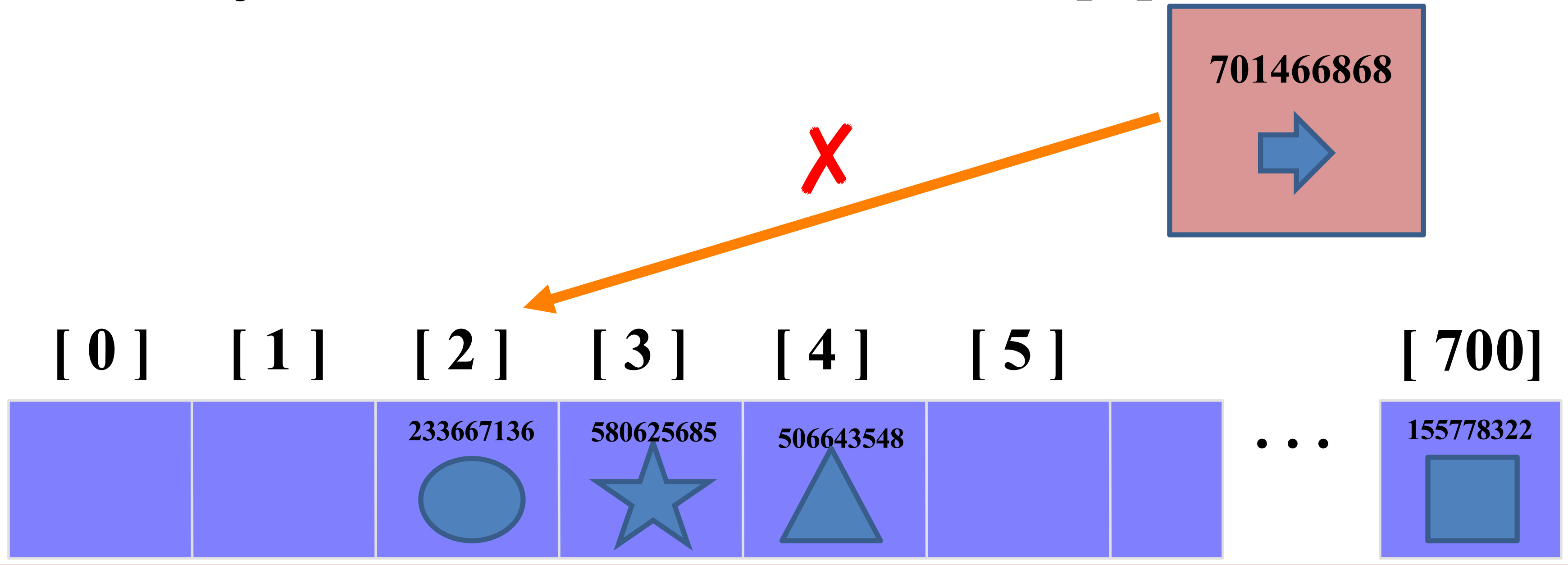
- Given a string of characters, e.g. “AZ”
- First consider the ASCII value of each character
  - E.g., 65 for “A”, 90 for “Z”
- Then, sum up the ASCII values of the characters
  - E.g., 65+90 = 155
- Finally, mod M, where M is the size of the hash table
  - E.g., 155 %10 = 5;
- String "AZ" is mapped to location 5 in the hash table

Collisions

- Problem: ***collision***
  - two ***keys*** may be mapped to the same location
  - Can we ensure that any two distinct keys get different locations?
    - No, if the size of the key space is larger than the size of the hash table

Collisions - example

- Suppose we insert a new record, with a hash value of 2.
- This is called a **collision**, because there is already another valid record at [2].



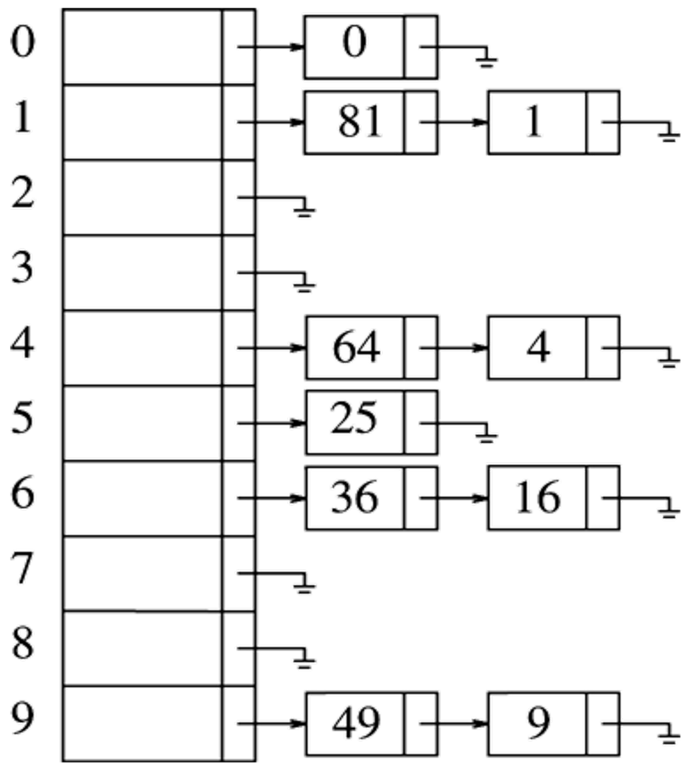
Collision Resolution Techniques

- Two strategies:
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing
- Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing).

Open hashing / Separate Chaining

- Instead of a hash table, use a table of linked list
- keep a linked list of records with keys mapped to the same location

h(K) = K mod 10



Separate Chaining (cont.)

- To search a record with key ***K***
  - Calculate h(*K*), takes Θ (1) time
  - Search the linked list at table[h[*K*]], which takes Θ (*d*) time, *d* is the list size
- Average list size $\alpha = \frac{n}{m}$, n: # of records, m: hash table size
- Searching time is Θ (1+α) on average

Improve performance of separate chaining

- Searching time is $\Theta(1+\alpha)$ on average
- $\alpha = \frac{n}{m}$ usually is called the load factor
- When the α exceeds a **threshold**, e.g. 1.5, double the table size
- **Rehash** each record in the old table into the new table
- Then, the value of α decreases
- Searching time is $\Theta(1+\alpha) = \Theta(1)$ on average

Separate Chaining (cont.)

- **Advantage**: implementation is easy for inserting, searching, and deleting
- **Disadvantage**: memory allocation for a new node will slow down the program

Closed hashing / Probing hash tables

- Basic Idea:
  - To insert a key **K**, compute **h(K)**. If location **h(K)** is empty, insert it there
  - If a collision occurs, probe alternative locations **h_1(K)**, **h_2(K)**, ..., until an empty location is found
- $h_i(K)$ = **(h(K) + f(i)) % *TableSize***,
  - f(.): ***collision resolution strategy***
- All data are stored inside the table, hash table size must be larger than the number of records
  - ***i.e., $m \geq n$***
  - Otherwise, no alternative locations can be found

Probing hash tables

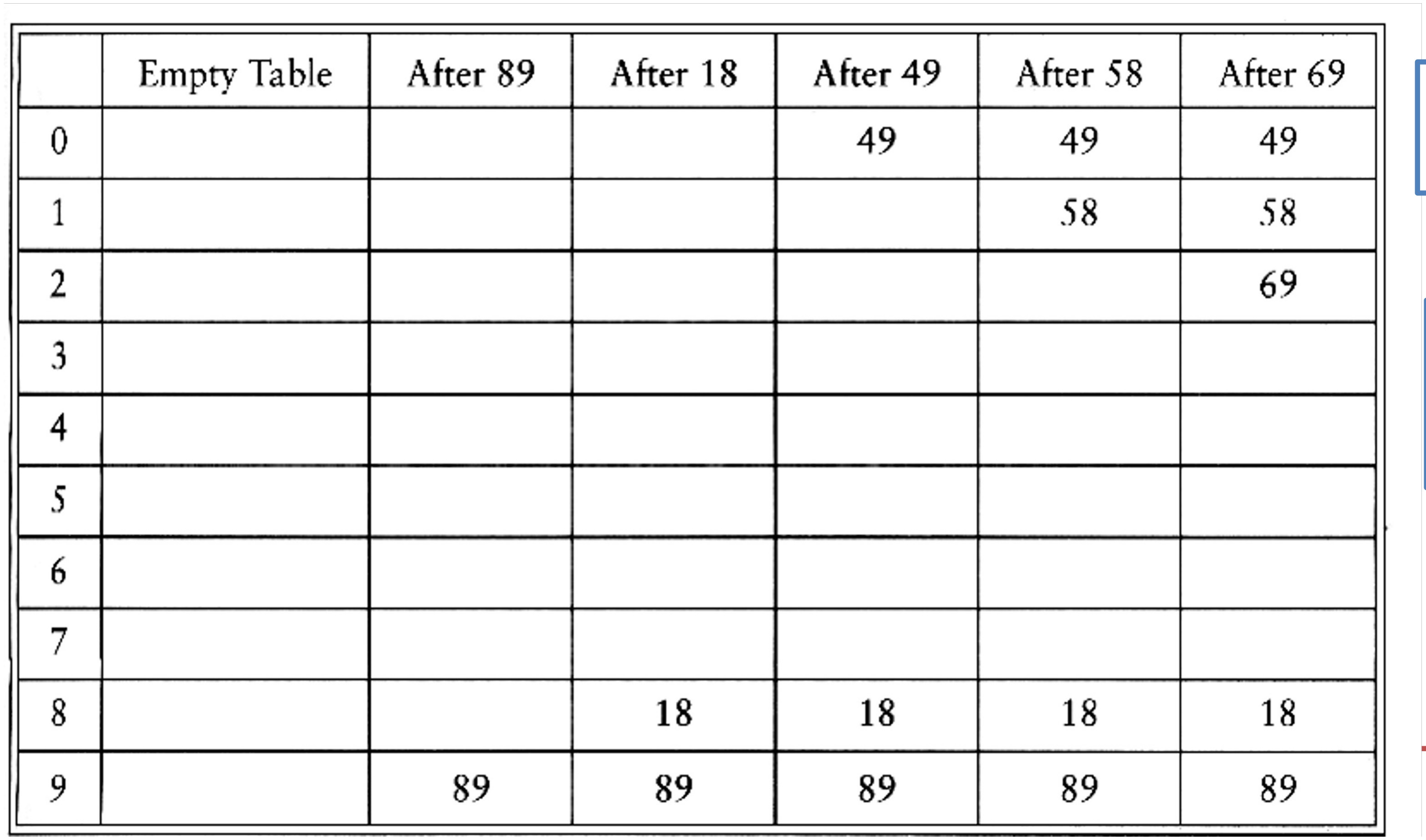
- Three approaches
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Solution 1: Linear Probing

- *f* is a linear function of *i*: i.e., *f(i)=i*
  - Locations are probed sequentially
  - $h_i(K) = (h(K) + i) \% \textit{TableSize}$
- **Insertion**:
  - Let **K** be a new key to be inserted， compute **h(K) first**
  - For i = 0 to *TableSize*-1
    - compute **L** = (h(K) + i) % *TableSize*
    - **Table[L]** is empty, then we put **K** there and stop.

Example of linear probing

- $h_i(K) = (h(K) + i) \% m$
  - E.g, inserting keys 89, 18, 49, 58, 69 with $h(K)=K \% 10$
- A clustering problem: small clusters grow to big clusters



	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

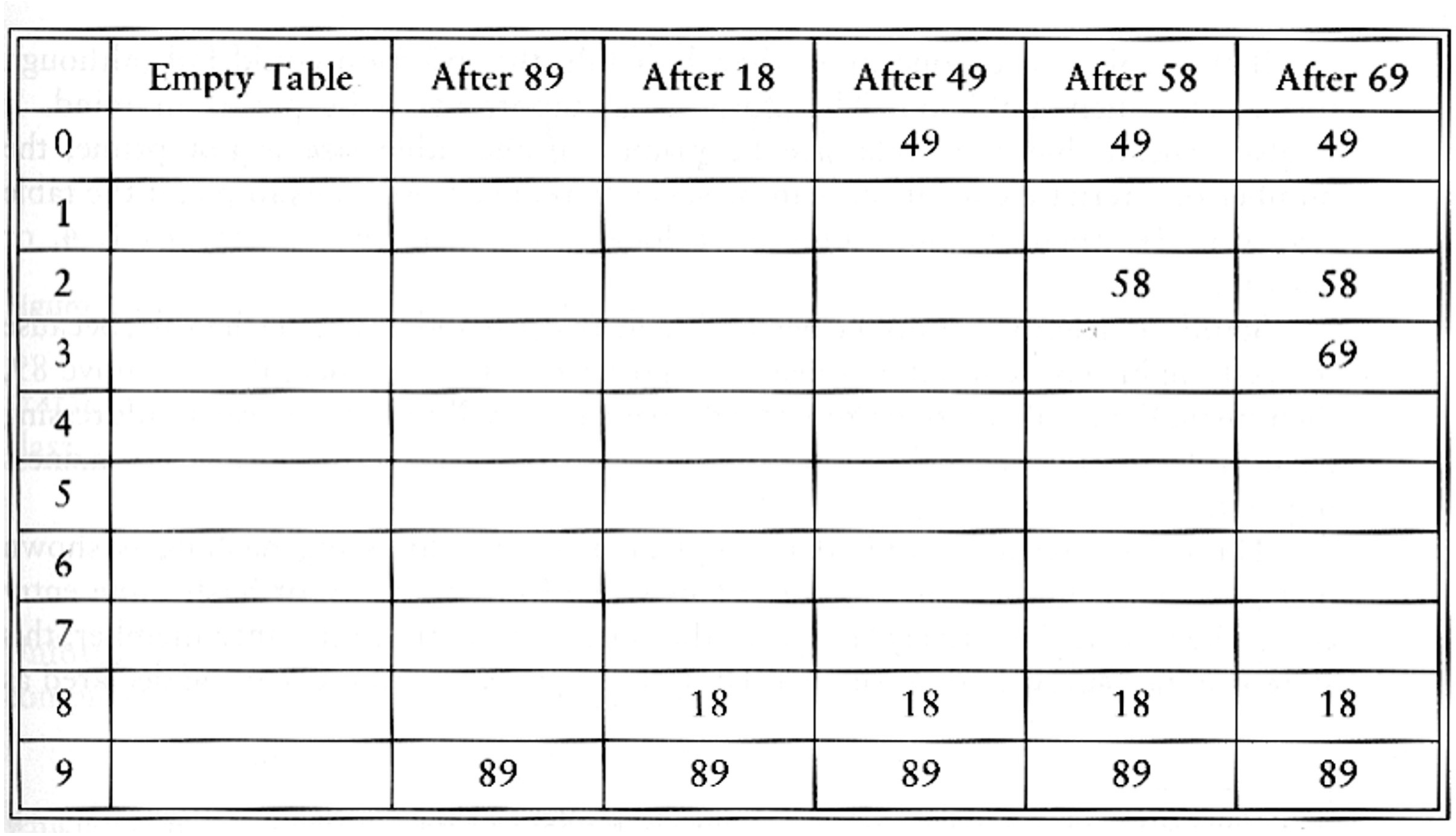
To insert 49, probe T[9], T[0]

To insert 58, probe T[8], T[9], T[0], T[1]

To insert 69, probe T[9], T[0], T[1], T[2]

Solution 2: Quadratic Probing

- $f(i) = i^2$
- $h_i(K) = (h(K) + i^2)$ % *TableSize, e.g.,* $h(K) = K$ % 10
  - E.g., inserting keys 89, 18, 49, 58, 69



	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

To insert 49, probe T[9], T[0]

To insert 58, probe T[8], T[9], T[(8+2^2) mod 10]

To insert 69, probe T[9], T[(9+1) mod 10], T[(9+2^2) mod 10]

Quadratic Probing

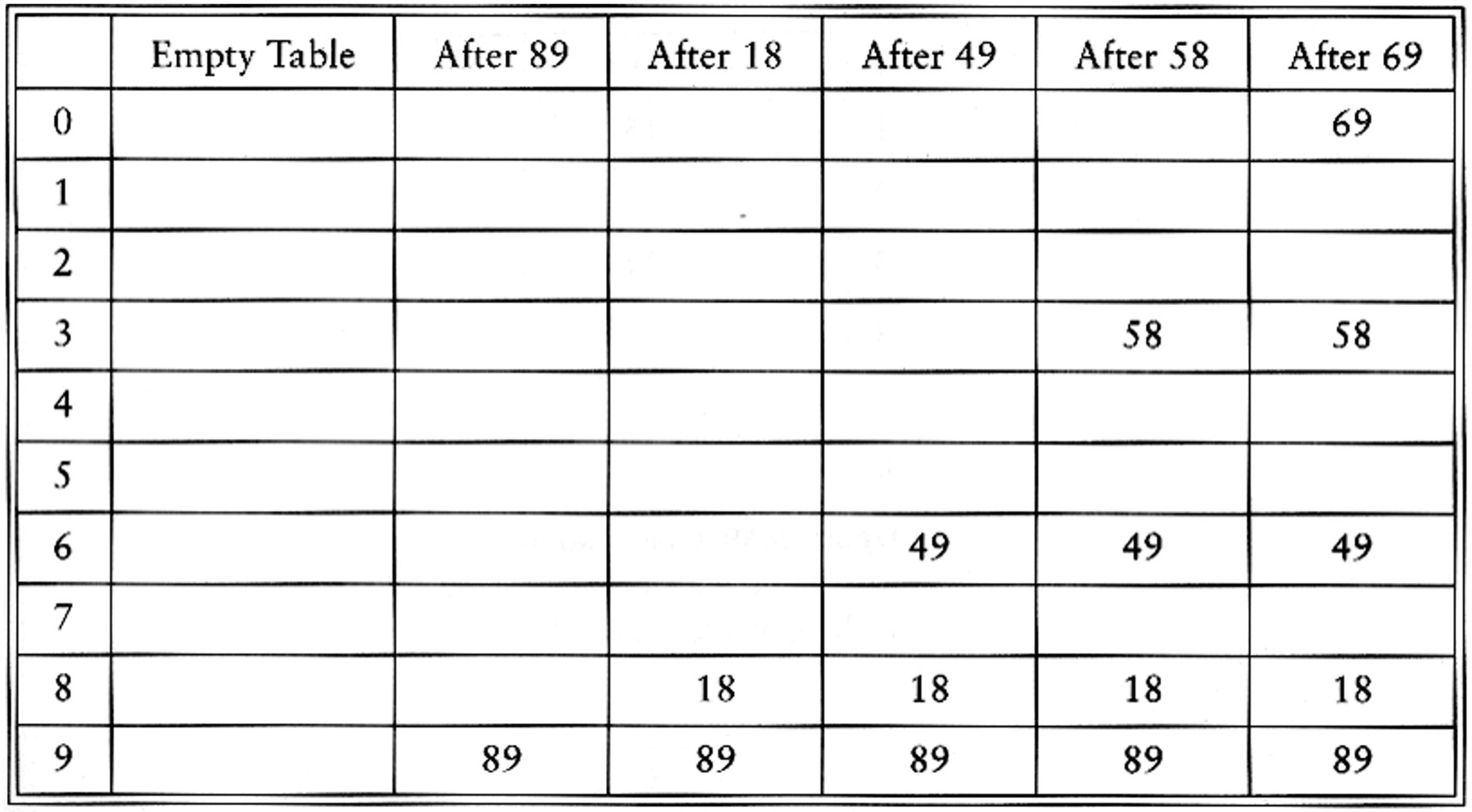
- Two keys with different initial hash locations will have different probe sequences
  - h(k1)=30, h(k2)=29, with difference only one
  - probe sequence for k1: 30, 31, 34, 39, …
  - probe sequence for k2: 29, 30, 33, 38,…
- If the table size *m* is prime, then a new key can always be inserted if the table is at least half empty

Solution 3: Double Hashing

- Use two hash functions: h() and h2()
- f(i) = i * h2(K)
- `h`$_i$`(K)=(h(K)+f(i))%m`
  - E.g. h2(K) = R - (K mod R), with R is a prime smaller than m
- The probe sequence `f(1),f(2),…` is independent of its initial location h(K)

Double Hashing

- `hi(K)=(h(K)+f(i))%m`; `h(K)=K%m`
- f(i) = i * h2(K);   h2(K) = R - (K mod R),
- Example: m=10, R = 7 and insert keys 89, 18, 49, 58, 69



	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

To insert 49, h2(49)=7, 2nd probe is T[(9+7) mod 10]

To insert 58, h2(58)=5, 2nd probe is T[(8+5) mod 10]

To insert 69, h2(69)=1, 2nd probe is T[(9+1) mod 10]

Choice of hash function `h2()`

- `h2(K)` cannot be 0, as i*0=0
- For any key K, `h2(K)` must be relatively prime to the table size m. Otherwise, we may probe only a fraction of the table entries.
  - e.g., if h(*K*)=0 and `h2(K) = m/2`, (m is even), then we will only examine entries Table[0], Table[m/2], and nothing else!
- One solution is to make m prime, and choose R to be a prime smaller than m, and set

  ```
  h2(K) = R - (K mod R)
  ```
- Quadratic probing, however, does not require the use of a second hash function
  - likely to be simpler and faster in practice

The performance of probing hash tables

- Load factor $\alpha = \frac{n}{m} \le 1$ as $n \le m$
- Collision probability is α for each probe
- Insert successfully at 1st probe with probability $1-\alpha$
- Insert successfully at 2nd probe with prob. $\alpha(1-\alpha)$
- Insert successfully at 3rd probe with prob. $\alpha^2(1-\alpha)$
- …
- Insert successfully at kth probe with prob. $\alpha^{k-1}(1-\alpha)$
- Average probe times are $\frac{1}{1-\alpha}$
- Insert and search average time is $\Theta(\frac{1}{1-\alpha}) = \Theta(1)$ if α is small, e.g., $\alpha=0.5$

Performance Comparison (n=400M)

- Sequential search : 200 ms
- Binary search: 0.002 ms
- Hash search: < 0.001 ms

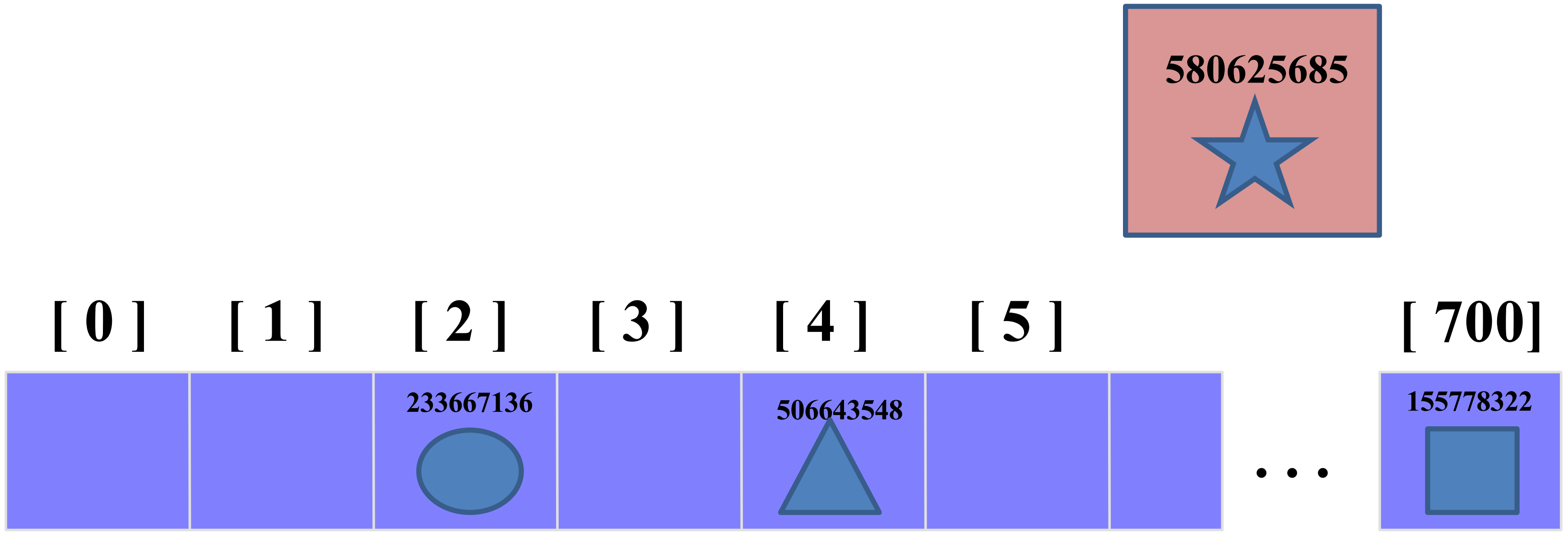
Insert

- Apply hash function to get a location
- Try to insert key at the location
- Deal with **collision**

Inserting a New Record

- Let us find the hash value for 580625685

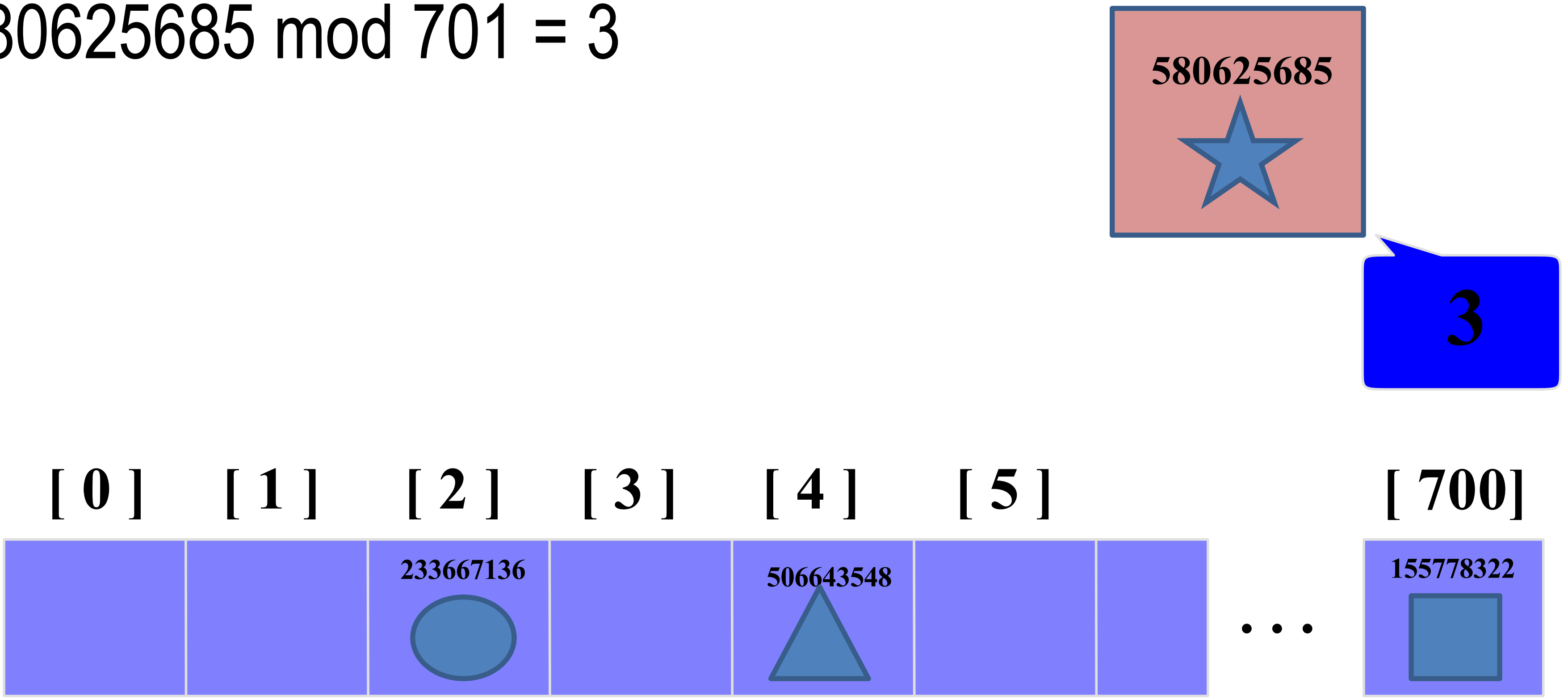
What is (580625685 mod 701) ?



Inserting a New Record

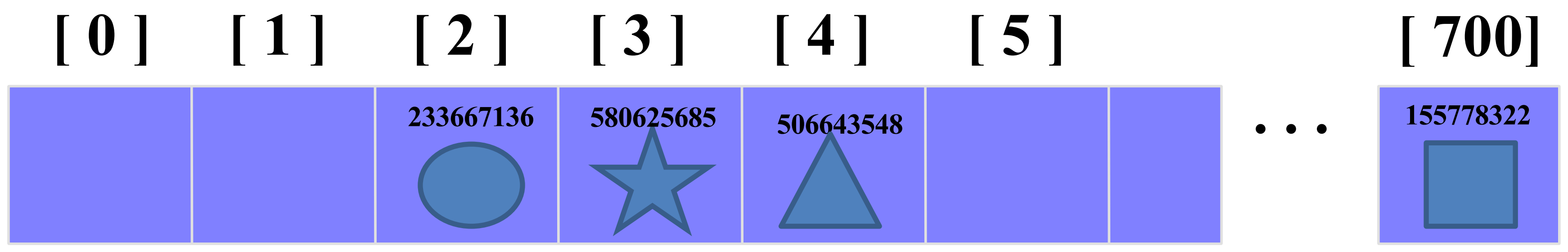
- Let us find the hash value for 580625685

580625685 mod 701 = 3



Inserting a New Record

The hash value is used to find the location of the new record.

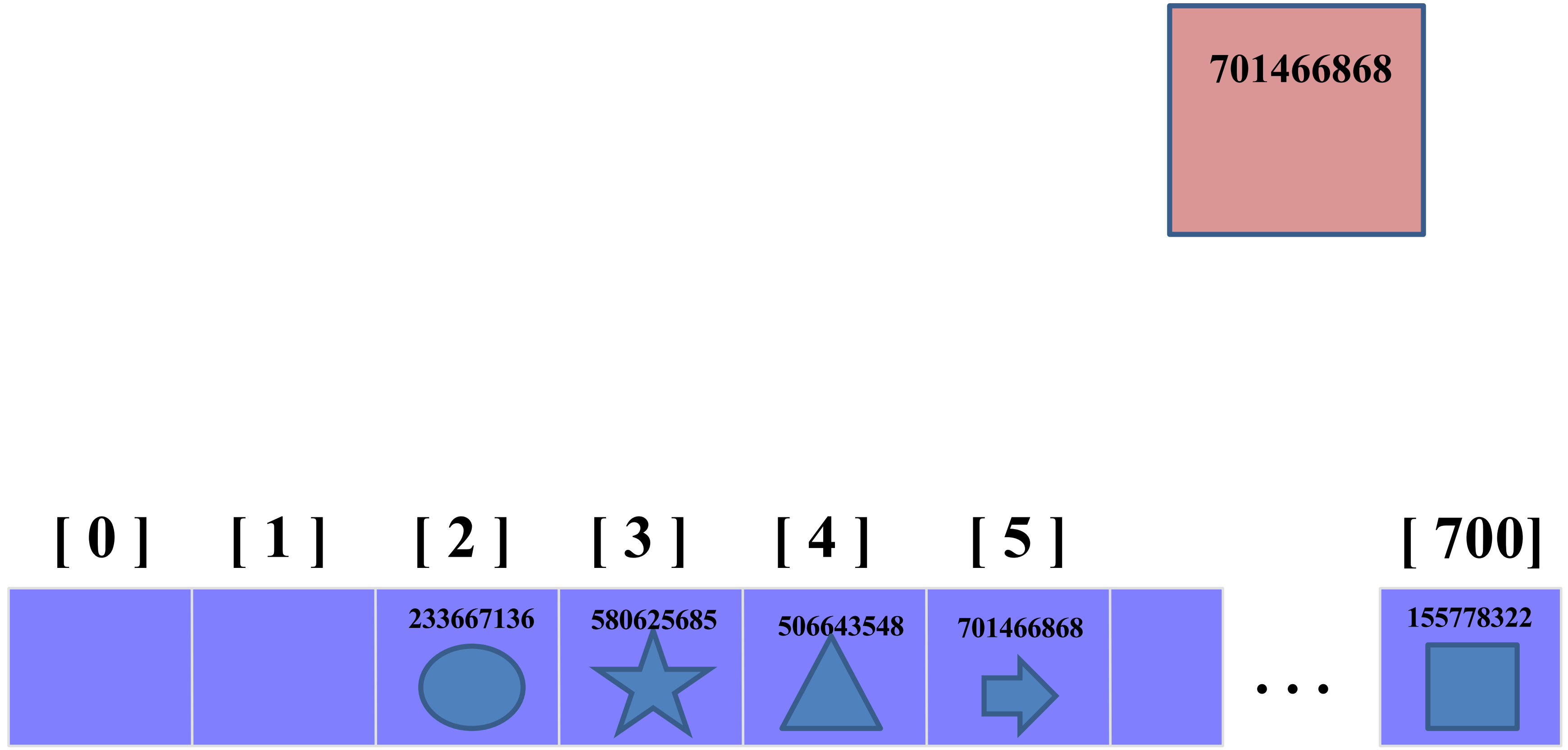


Search

- Apply the hash function to get a location
- Look at that location.
- Deal with collision.

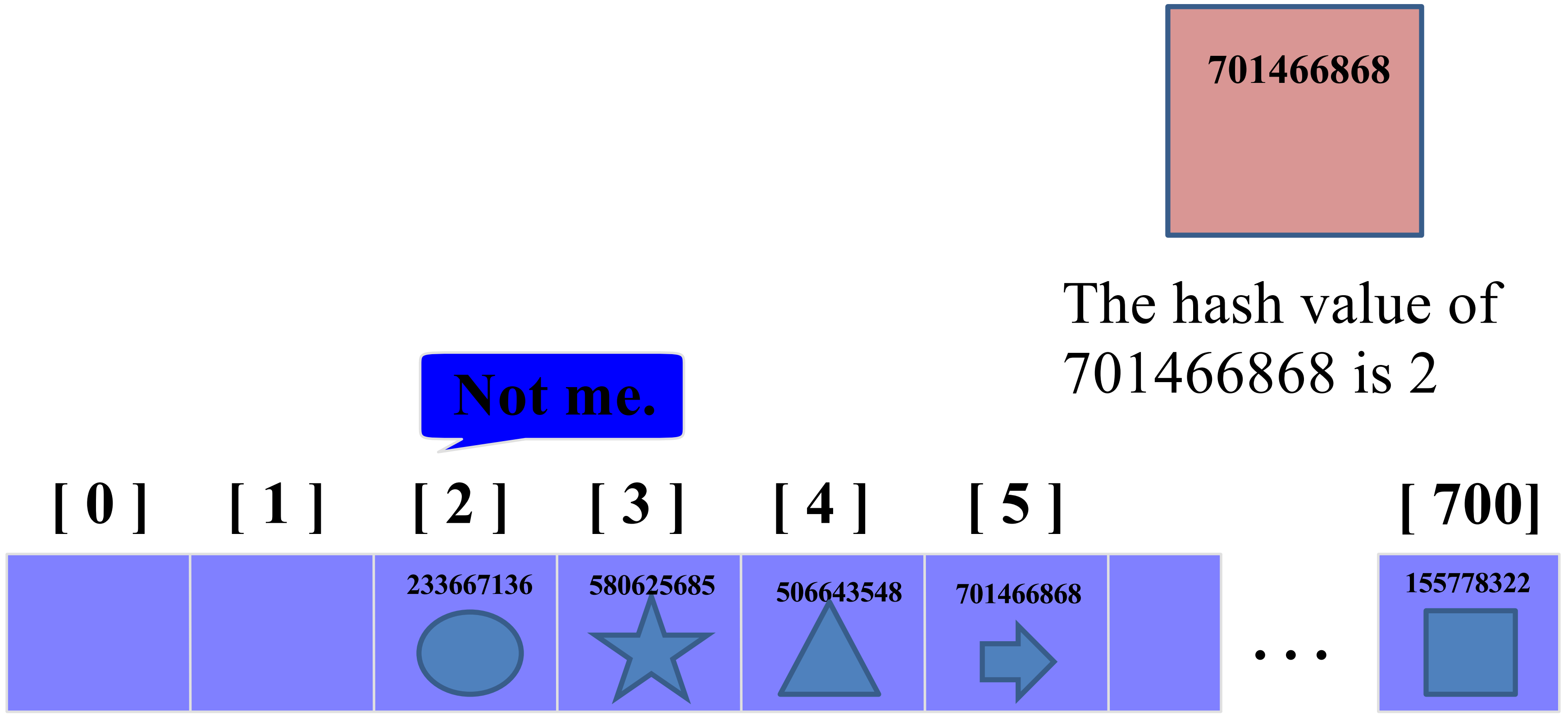
Searching for a Key

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



Searching for a Key

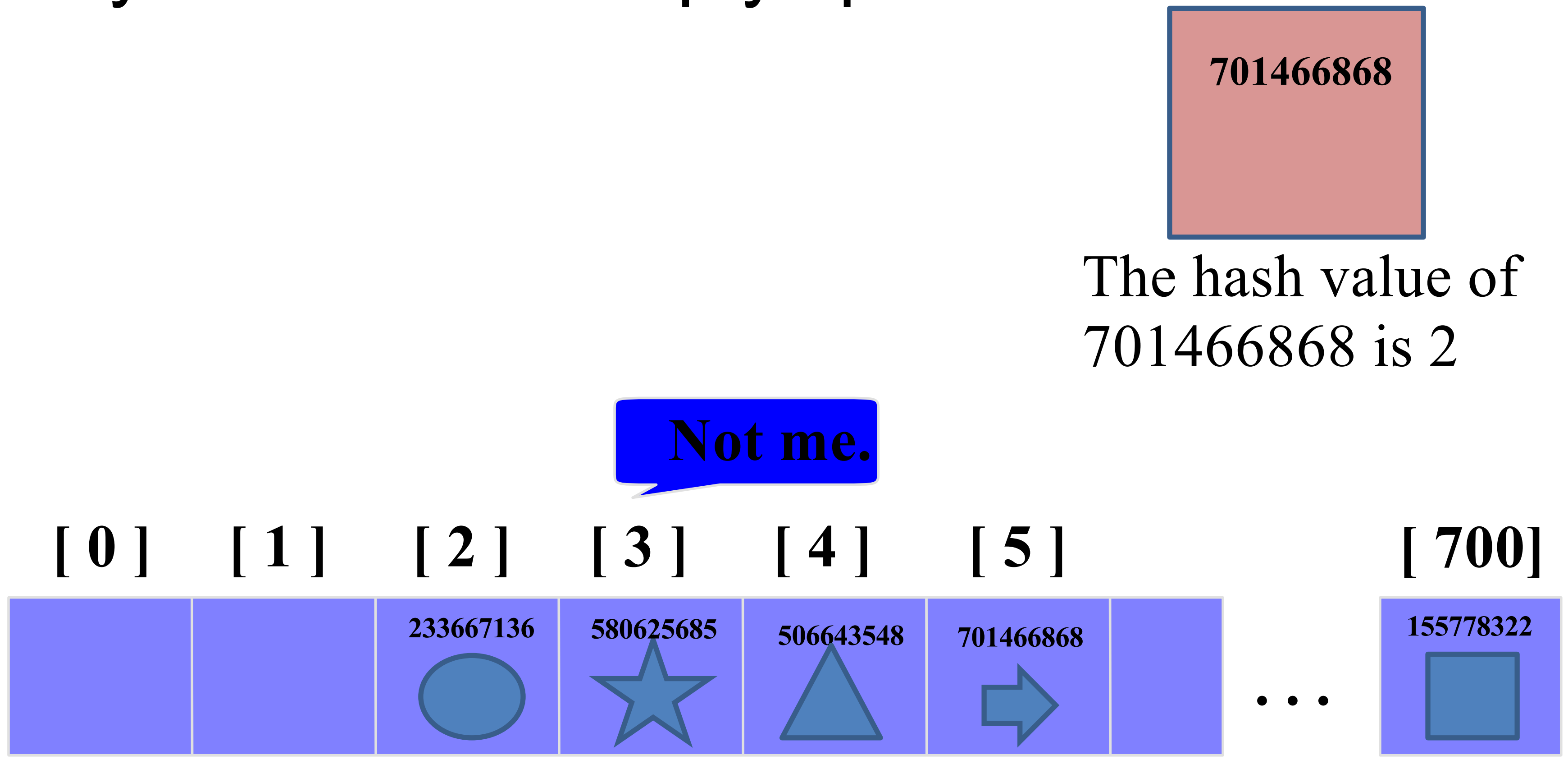
- Calculate the hash value.
- Check that location of the array for the key.



The hash value of 701466868 is 2

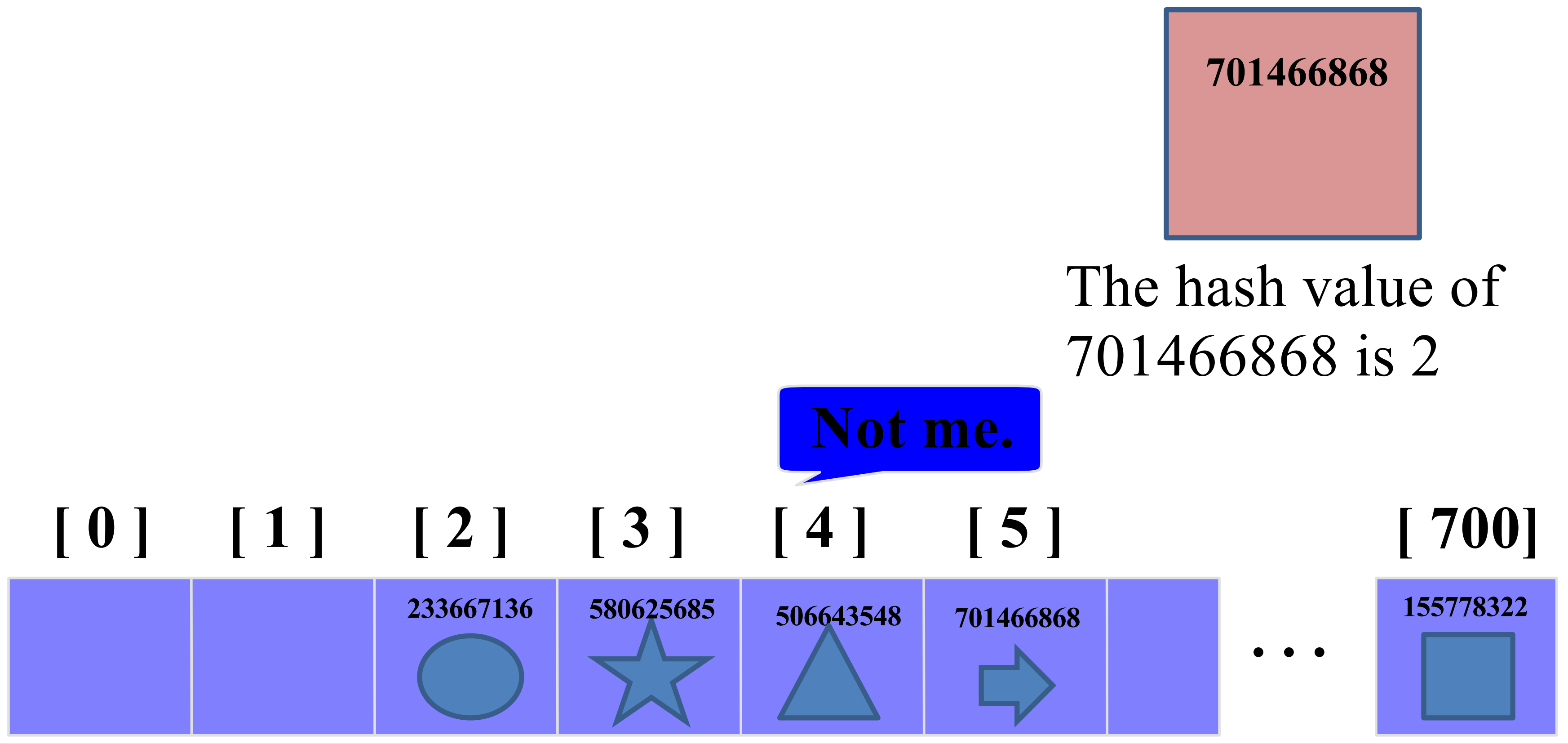
Searching for a Key

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



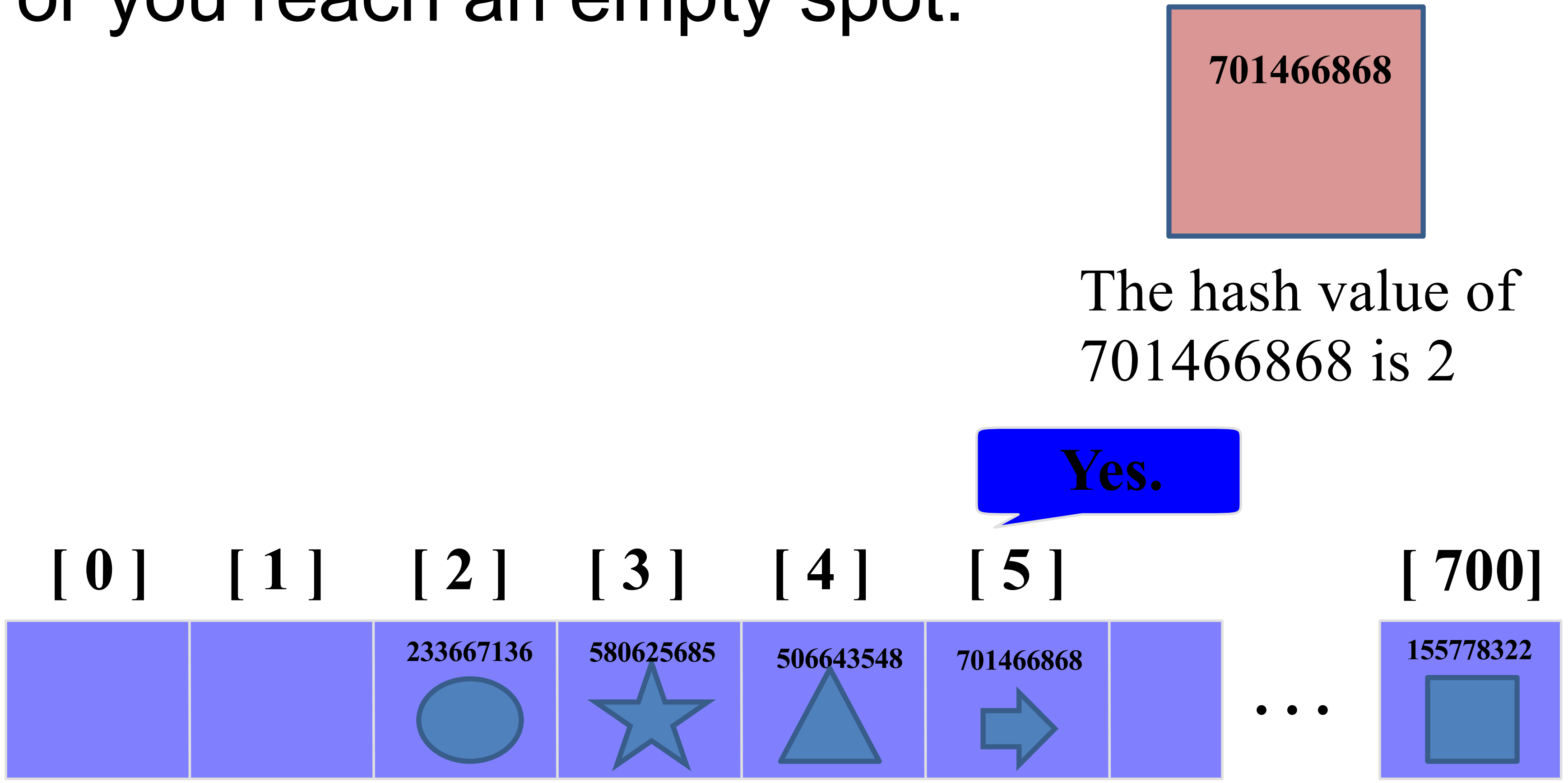
Searching for a Key

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



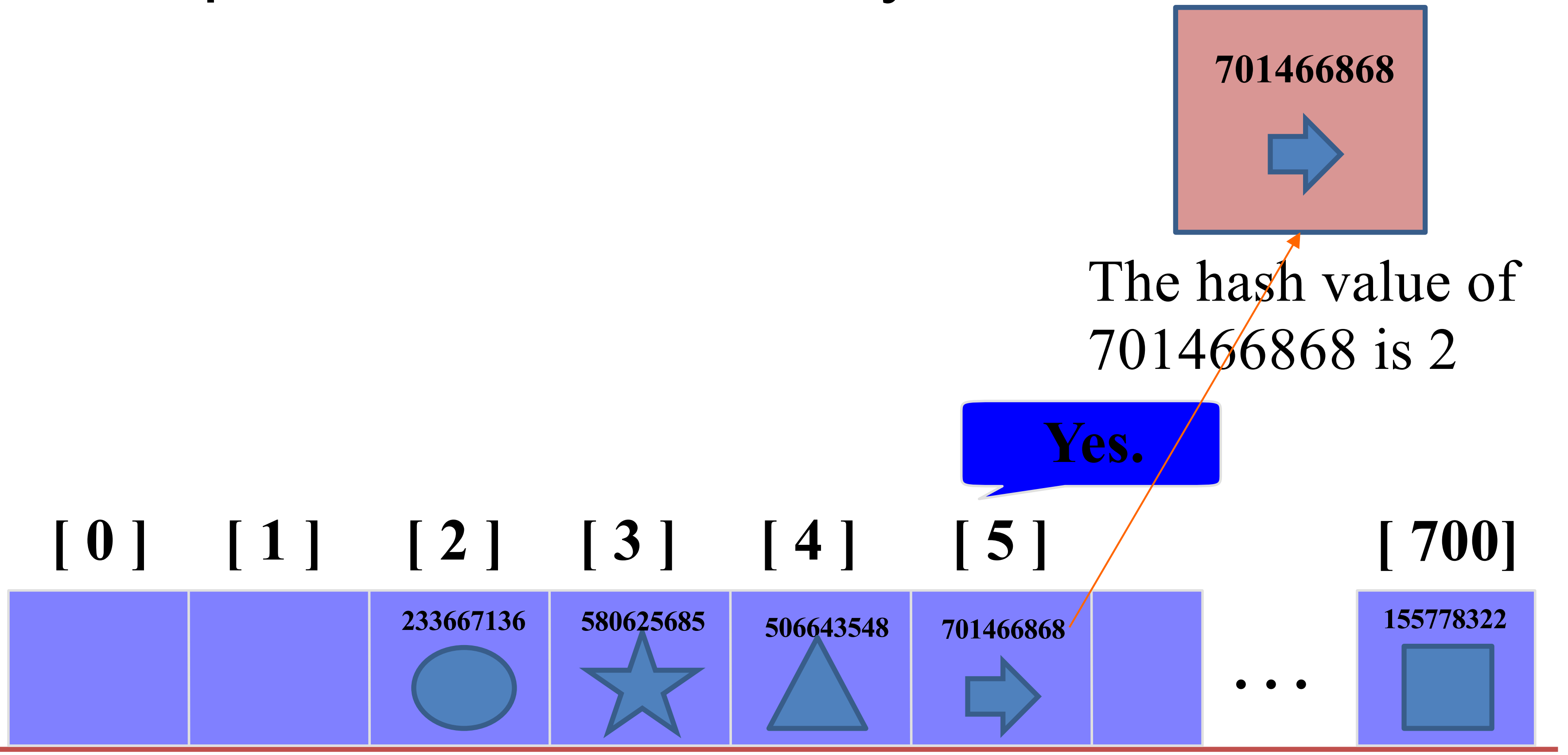
Searching for a Key

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



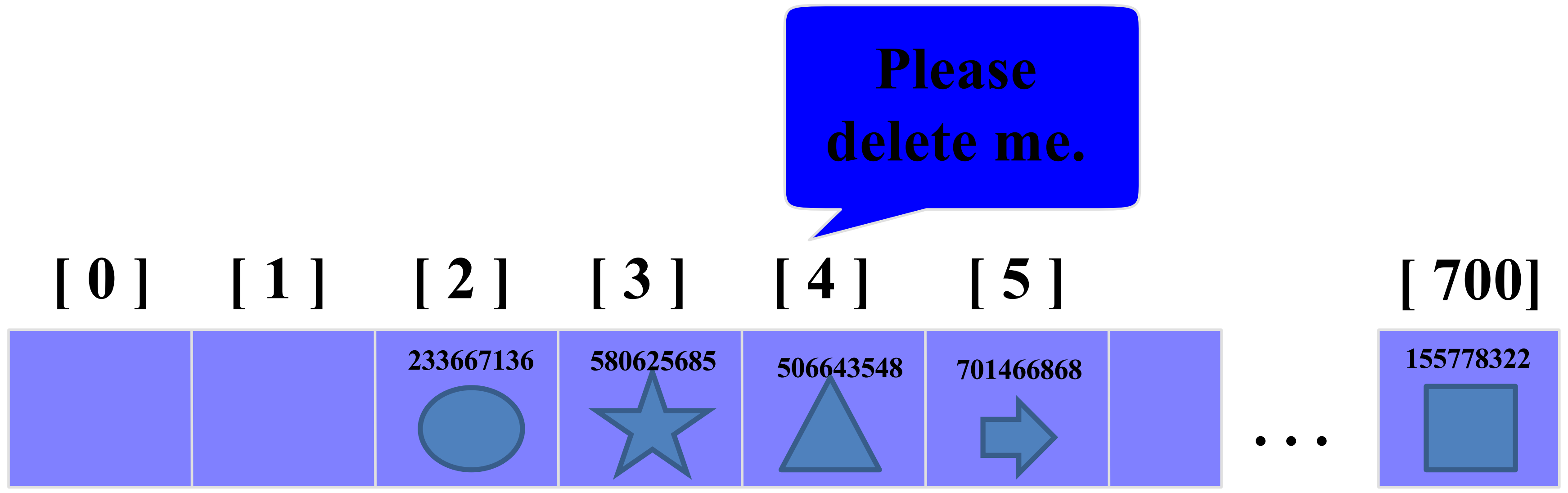
Searching for a Key

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



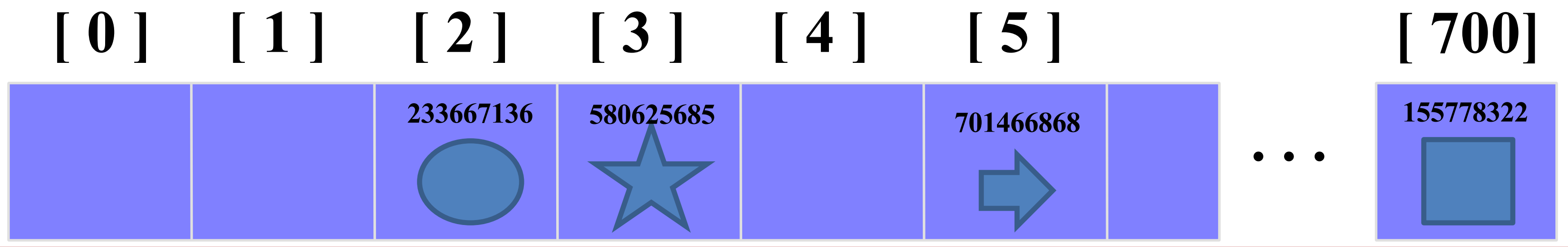
Deleting a Record

- Records may also be deleted from a hash table.



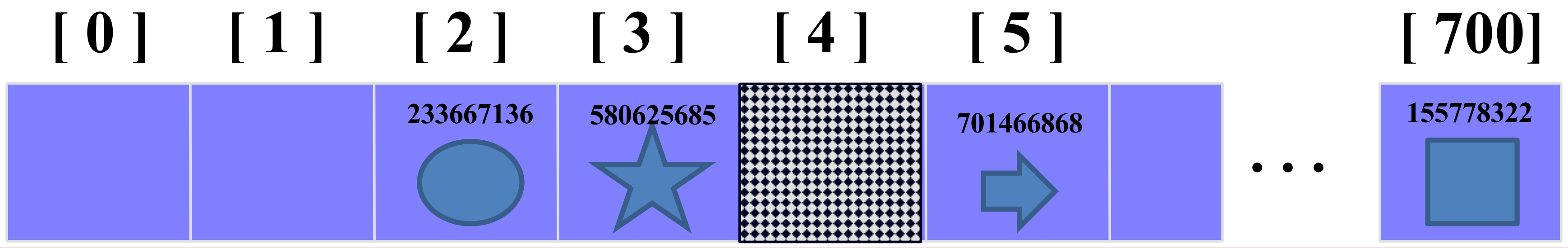
Deleting a Record

- 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.



Conclusions

- Sequential search: $\Theta(n)$ on average
  - improve to $\Theta(\sqrt{n})$ by Jump search with ordered arrays
- Binary search: $\Theta(\log n)$
- Self-organizing lists, and Bit vectors
- Hashing: $\Theta(1)$ on average
  - Hash table size usually is prime.
  - Hash functions
    - mod function, mid-square, sum for strings
  - Collision solutions
    1. Separating chaining
    2. Probing hash tables
       - linear probing, quadratic probing, double hashing