Hash Maps

A basic problem

- We have to store some records and perform the following:
  - add new record
  - delete record
  - search a record by key
- Find a way to do these efficiently!

Unsorted array

- Use an array to store the records, in unsorted order
  - add - add the records as the last entry fast *O(1)*
  - delete a target - slow at finding the target, fast at filling the hole (just take the last entry) *O(n)*
  - search - sequential search slow *O(n)*

Sorted array

- Use an array to store the records, keeping them in sorted order
  - add - insert the record in proper position. much record movement slow *O(n)*
  - delete a target - how to handle the hole after deletion? Much record movement slow *O(n)*
  - search - binary search fast *O(log n)*

Linked list

- Store the records in a linked list (unsorted)
  - add - fast if one can insert node anywhere *O(1)*
  - delete a target - fast at disposing the node, but slow at finding the target *O(n)*
  - search - sequential search slow *O(n)*
    (if we only use linked list, we cannot use binary search even if the list is sorted.)

More approaches

- have better performance but are more complex
  - Hash table
  - Tree (BST, Heap, …)

Array as table

studid	name	score
0012345	andy	81.5
0033333	betty	90
0056789	david	56.8
...		
9801010	peter	20
9802020	mary	100
...		
9903030	tom	73
9908080	bill	49

Consider this problem. We want to store 1,000 student records and search them by student id.

Array as table

	name	score
0		
:	:	:
12345	andy	81.5
:	:	:
33333	betty	90
:	:	:
56789	david	56.8
:	:	:
:	:	:
9908080	bill	49
:	:	:
9999999		

One ‘stupid’ way is to store the records in a huge array (index 0..9999999). The index is used as the student id, i.e. the record of the student with studid 0012345 is stored at A[12345]

Array as table

- Store the records in a huge array where the index corresponds to the key
  - add - very fast *O(1)*
  - delete - very fast *O(1)*
  - search - very fast *O(1)*
- But it wastes a lot of memory! Not feasible.

Hash function

```
function Hash(key: KeyType): integer;
```

Imagine that we have such a magic function Hash. It maps the key (studid) of the 1000 records into the integers 0..999, one to one. No two different keys map to the same number.

H('0012345') = 134
H('0033333') = 67
H('0056789') = 764
…
H('9908080') = 3

Hash Table

To store a record, we compute Hash(stud_id) for the record and store it at the location Hash(stud_id) of the array. To search for a student, we only need to peek at the location Hash(target stud_id).

0			
	:	:	:
3	9908080	bill	49
	:	:	:
67	0033333	betty	90
	:	:	:
134	0012345	andy	81.5
	:	:	:
764	0056789	david	56.8
	:	:	:
999	:	:	:

Hash Table with Perfect Hash

- Such magic function is called a perfect hash
  - add - very fast *O(1)*
  - delete - very fast *O(1)*
  - search - very fast *O(1)*
- But it is generally difficult to design a perfect hash. (e.g. when the potential key space is large)

Hash function

- A hash function maps a key to an index within in a range
- Desirable properties:
  - simple and quick to calculate
  - even distribution, avoid collision as much as possible

```
function Hash(key: KeyType);
```


Division Method

$$\boxed{h(k) = k \bmod m}$$

- Certain values of m may not be good:
  - When $m = 2^p$ then $h(k)$ is the p lower-order bits of the key
  - Good values for m are prime numbers which are not close to exact powers of 2. For example, if you want to store 2000 elements then m=701 (m = *hash table length*) yields a hash function:

$$h(key) = k \bmod 701$$

Collision

- For most cases, we cannot avoid collision
- Collision resolution - how to handle when two different keys map to the same index

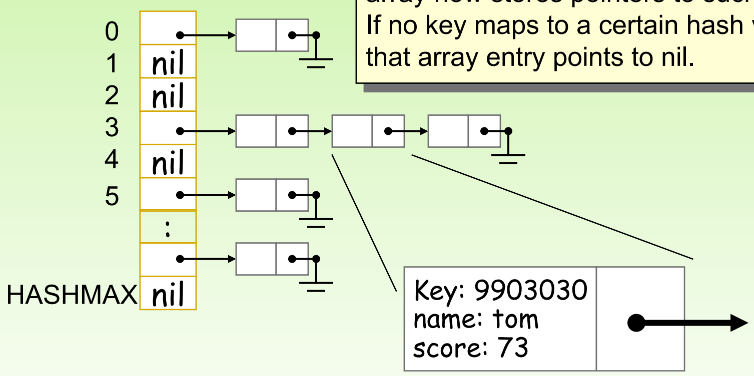
H('0012345') = 134
H('0033333') = 67
H('0056789') = 764
…
H('9903030') = **3**
H('9908080') = **3**

Solutions to Collision

- The problem arises because we have two keys that hash in the same array entry, a **collision**. There are two ways to resolve collision:
  - **Hashing with Chaining:** every hash table entry contains a pointer to a linked list of keys that hash in the same entry
  - **Hashing with Open Addressing:** every hash table entry contains only one key. If a new key hashes to a table entry which is filled, systematically examine other table entries until you find one empty entry to place the new key

Chained Hash Table

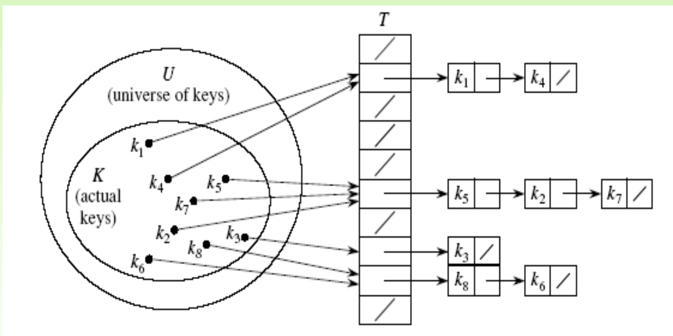
One way to handle collision is to store the collided records in a linked list. The array now stores pointers to such lists. If no key maps to a certain hash value, that array entry points to nil.



Chained Hash Table

Put all elements that hash to the same slot into a linked list.

- Slot *j* contains a pointer to the head of the list of all stored elements that hash to *j*
  - If there are no such elements, slot *j* contains NIL.



Chained Hash table

- Hash table, where collided records are stored in linked list
  - good hash function, appropriate hash size
    - Few collisions. Add, delete, search very fast *O(1)*
  - otherwise…
    - some hash value has a long list of collided records..
    - add - just insert at the head fast *O(1)*
    - delete a target - delete from unsorted linked list slow
    - search - sequential search slow *O(n)*

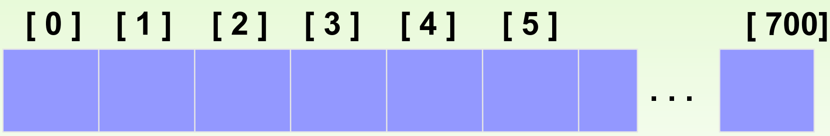
Open Addressing

An alternative to chaining for handling collisions.

- • Store all keys in the hash table itself.
- • Each slot contains either a key or NIL.
- • To search for key *k*:
  - Compute *h(k)* and examine slot *h(k)*.
    Examining a slot is known as a ***probe***.
  - If slot *h(k)* contains key *k*, the search is successful.
    If this slot contains NIL, the search is unsuccessful.
  - There's a third possibility: slot *h(k)* contains a key that is not *k*.
    We compute the index of some other slot, based on *k* and on which probe (count from 0: 0th, 1st, 2nd, etc.) we're on. Keep probing until we either find key *k* (successful search) or we find a slot holding NIL (unsuccessful search).

What is a Hash Table ?

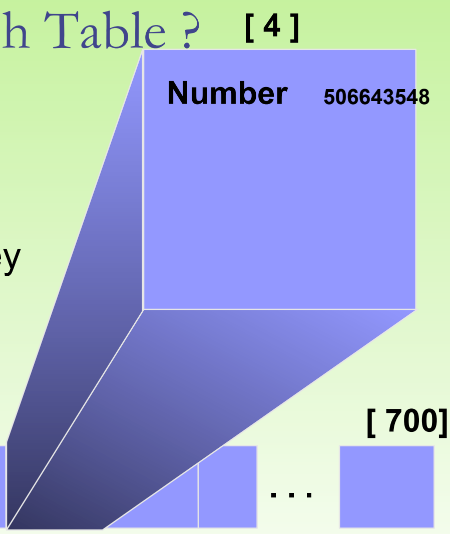
- The simplest kind of hash table is an array of records.
- This example has 701 records.



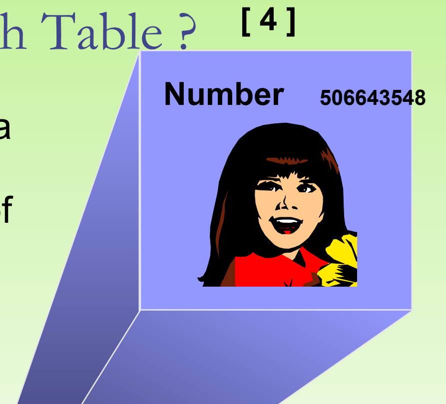
An array of records

What is a Hash Table ?

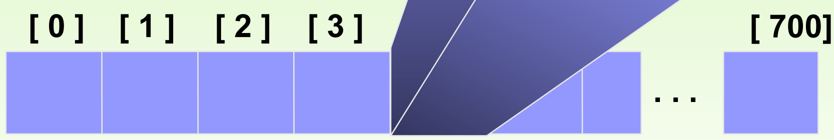
- Each record has a special field, called its key.
- In this example, the key is a long integer field called Number.



What is a Hash Table ?

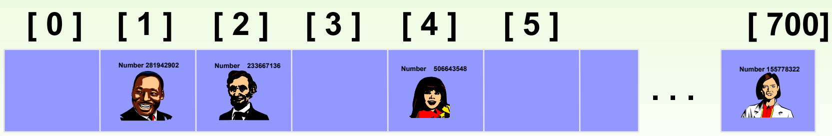


- The number might be a person's identification number, and the rest of the record has information about the person.



What is a Hash Table ?

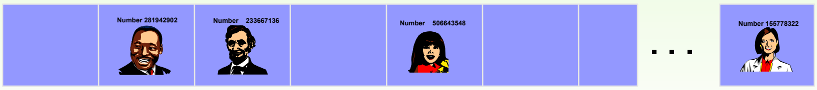
- When a hash table is in use, some spots contain valid records, and other spots are "empty".



Inserting a New Record

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

[0] [1] [2] [3] [4] [5] [700]



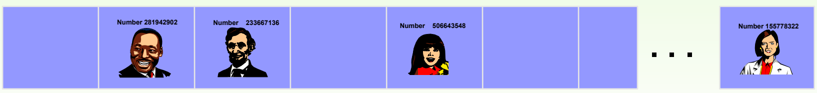
Inserting a New Record

- Typical way create a hash value:

  (Number mod 701)

What is (580625685 mod 701) ?

[0] [1] [2] [3] [4] [5] [700]

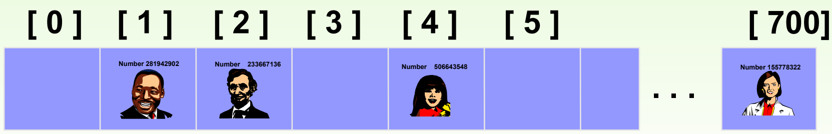


Inserting a New Record

- Typical way to create a hash value:

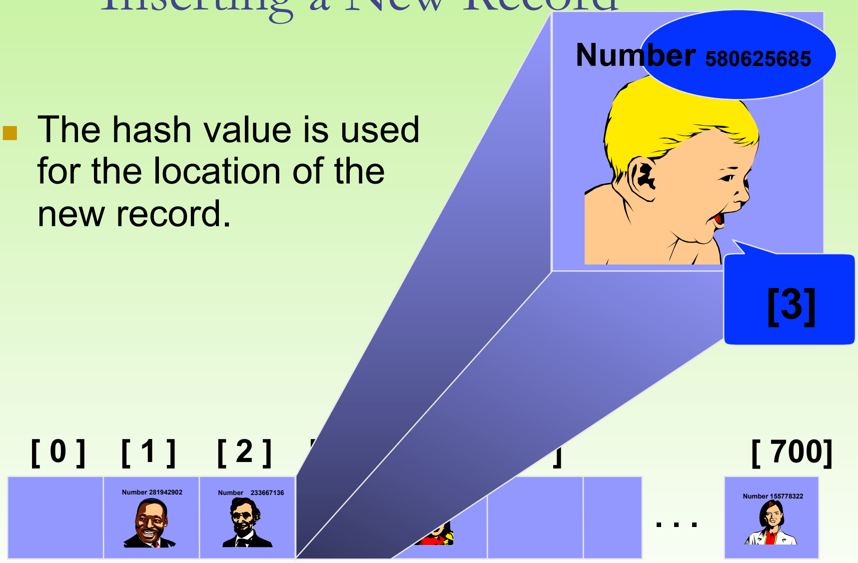
  (Number mod 701)

What is (580625685 mod 701) ?



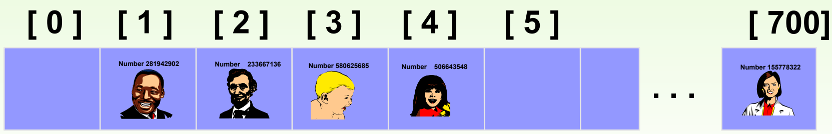
Inserting a New Record

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



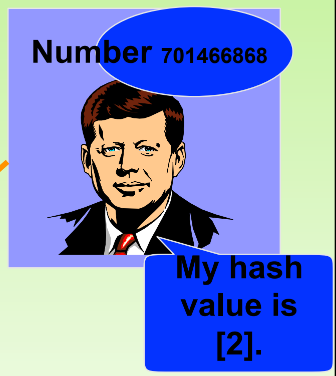
Inserting a New Record

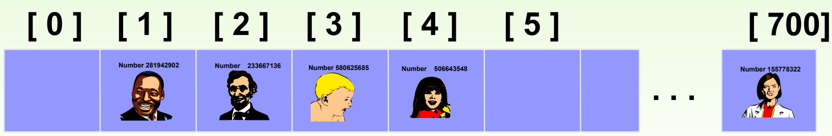
- The hash value is used for the location of the new record.



Collisions

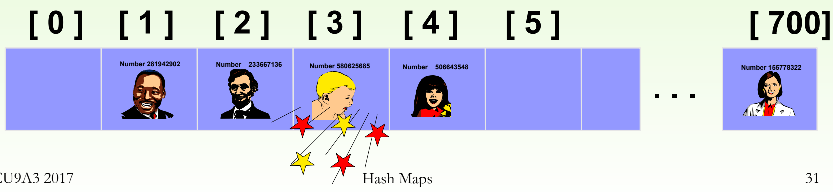
- Here is another new record to insert, with a hash value of 2.





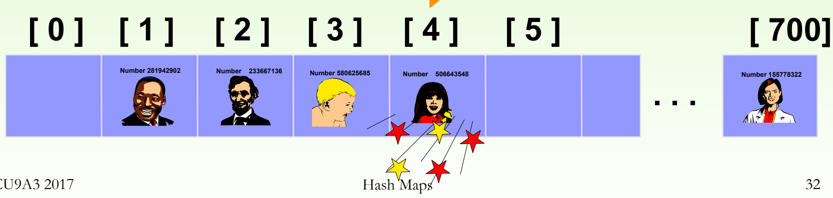
Collisions

- 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.



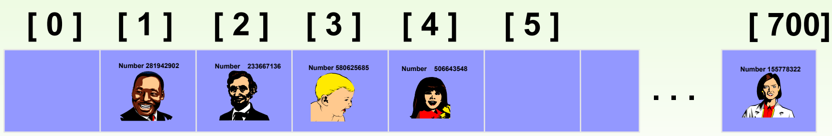
Collisions

- 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.



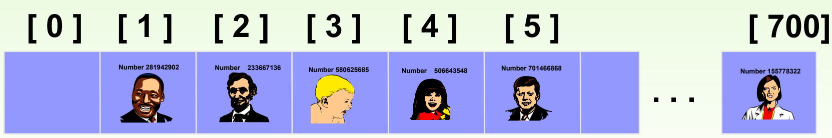
Collisions

- 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.

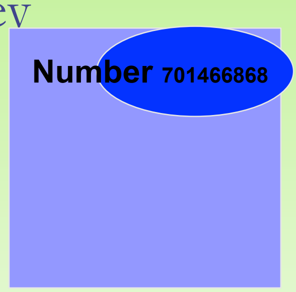


Collisions

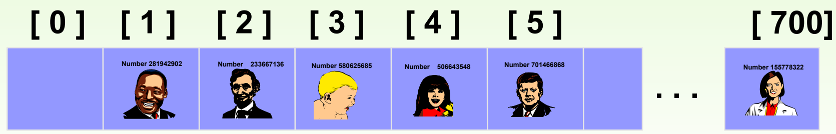
- The new record goes in the empty spot!



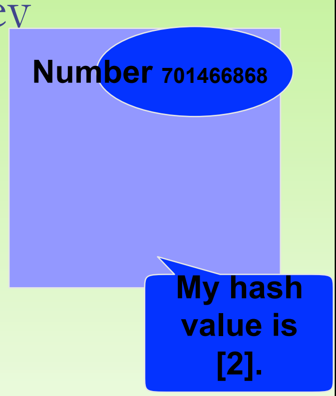
Searching for a Key



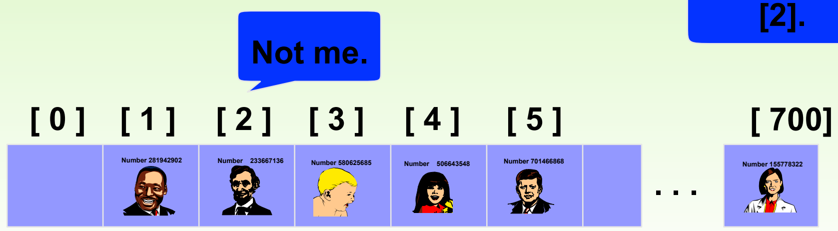
- The data that's attached to a key can be found fairly quickly.



Searching for a Key

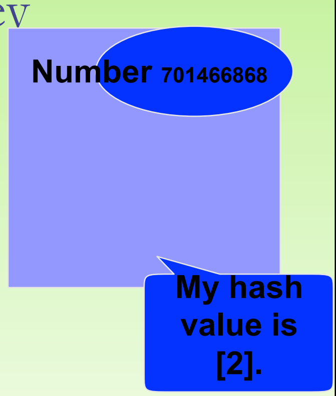


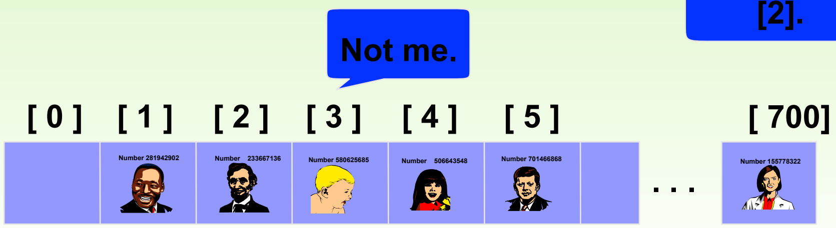
- Calculate the hash value.
- Check that location of the array for the key.



Searching for a Key

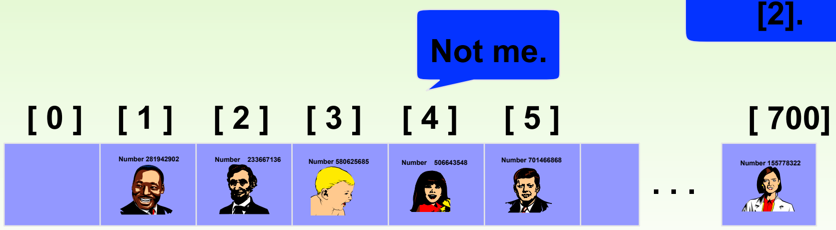
- Keep moving forward until you find the key, or you reach an empty spot.





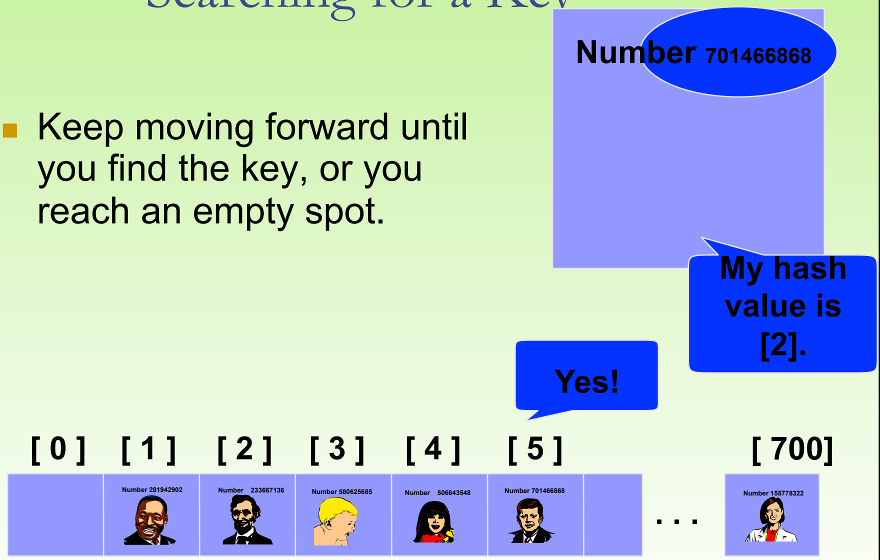
Searching for a Key

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



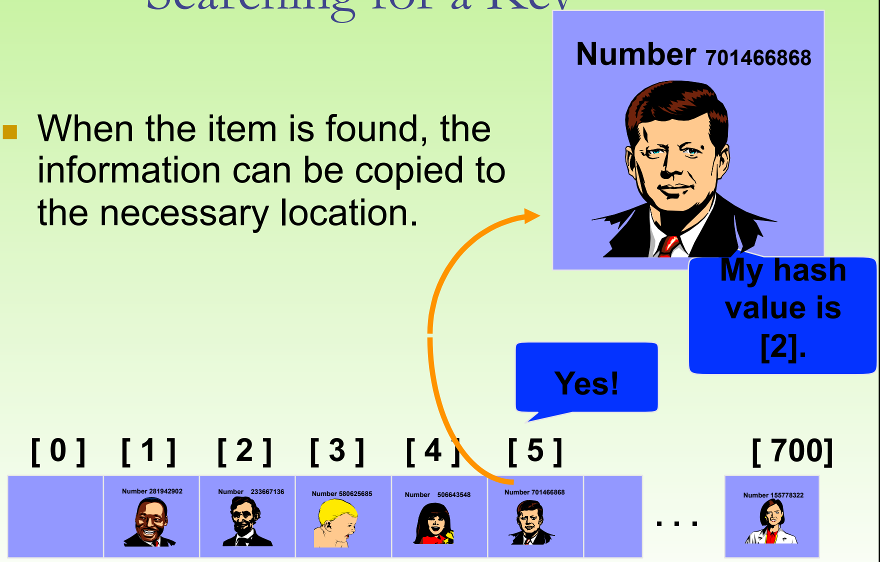
Searching for a Key

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



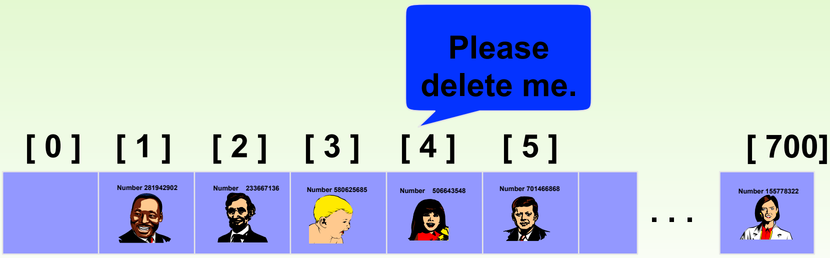
Searching for a Key

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



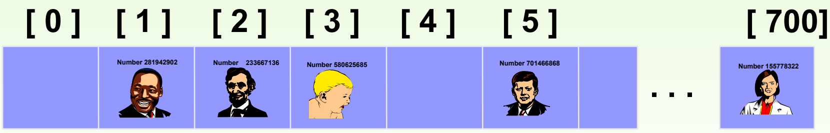
Deleting a Record

- Records may also be deleted from a hash table.



Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.



Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.
- The location must be marked in some special way so that a search can tell that the spot used to have something in it.

End of Lecture