# Searching

Read Ch11

# Outline

- Serial search
- Binary search
- Search using open-address hashing

# Binary search

```
private static int binarySearch (int[]array, int idxs, int idxe, int searche){
        if(idxe<idxs) return (-1);
        int idx_middle = (idxe+idxs)/2;

        if(array[idx_middle]==searche)
            return idx_middle;
        else if(searche<array[idx_middle])
            return binarySearch(array,idxs,idx_middle-1,searche);
        else
            return binarySearch(array,idx_middle+1,idxe,searche);
    }

public static int binarySearch (int[] array, int searche){
        int resultPos = binarySearch(array, 0,array.length-1,searche);
        return resultPos;
}
```

# Complexity analysis

- Let T(N) = # of operations to search over N elements
- T(1) = 1
- It takes O(1) time to do the comparisons, then it cuts the search range in half.
  - T(N) = T(N/2) + 1
- Repeat the recurrence…

  T(N) = T(N/4) + 2

  = T(N/8) + 3 = …

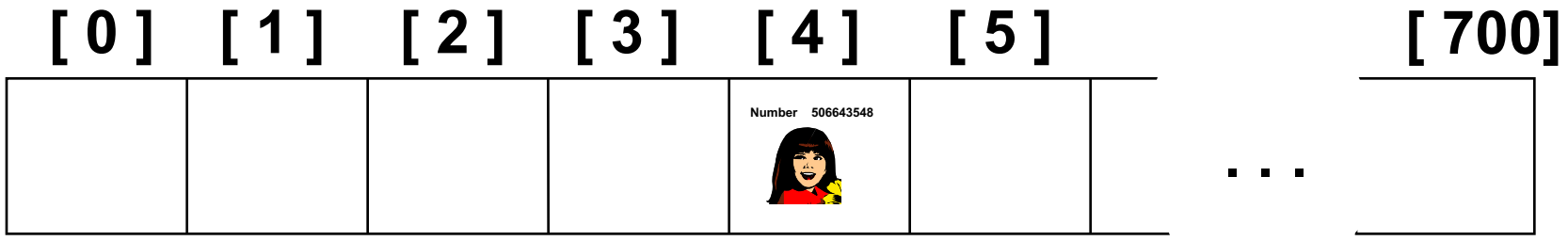
  = $T(N/2^k) + k$
- Round up N to nearest power of 2: $N \leq 2^m$. $T(N) \leq T(2^m/2^k)+k$
- Let k = m. $T(N) \leq T(2^m/2^m)+m = T(1)+m = 1+m = O(m)$
- If $N=2^m$, then m=log N. So T(N) = O(log N)

# Hash Tables

- Hash tables are a common approach to the storing/searching problem.

# What is a Hash Table ?

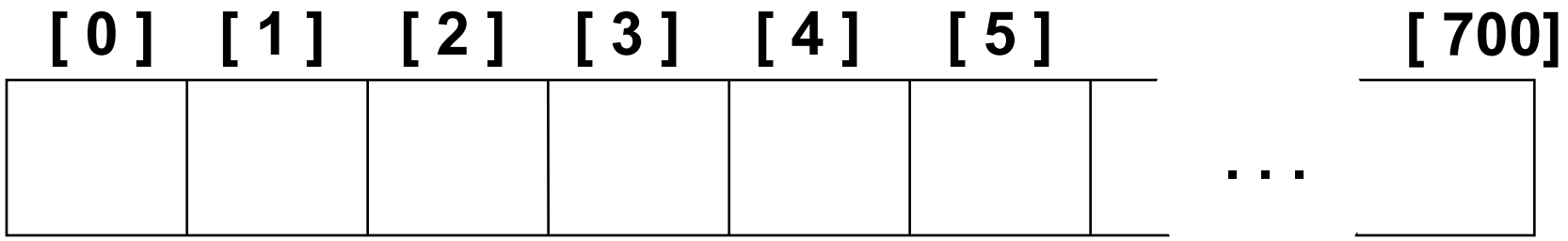
- The simplest kind of hash table is an array of records.
- This example has 701 records.




**An array of records**

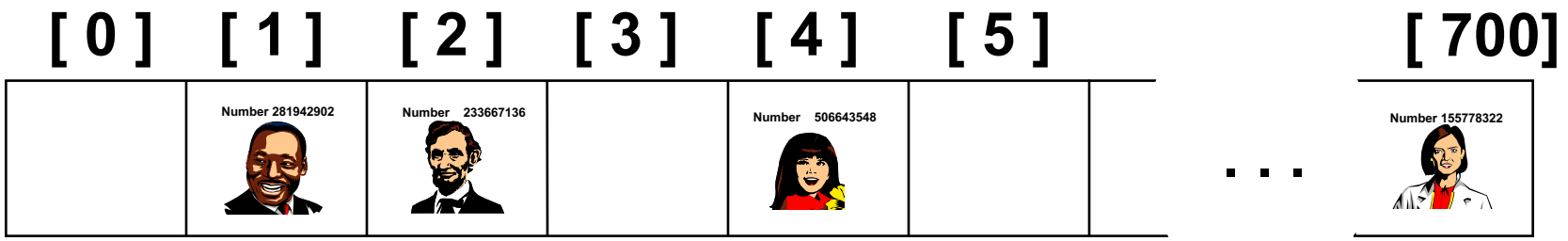
# What is a Hash Table ?

- Each record has a special field, called its key.
- In this example, the key is a long integer field called Number.
- The number might be a person's identification number, and the rest of the record has information about the person.

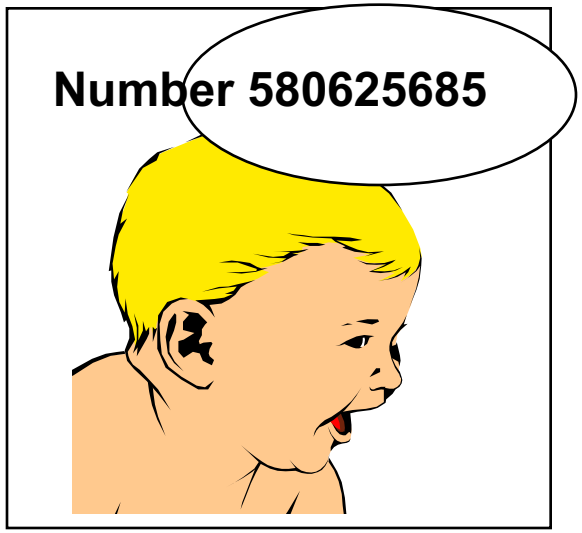


# What is a Hash Table ?

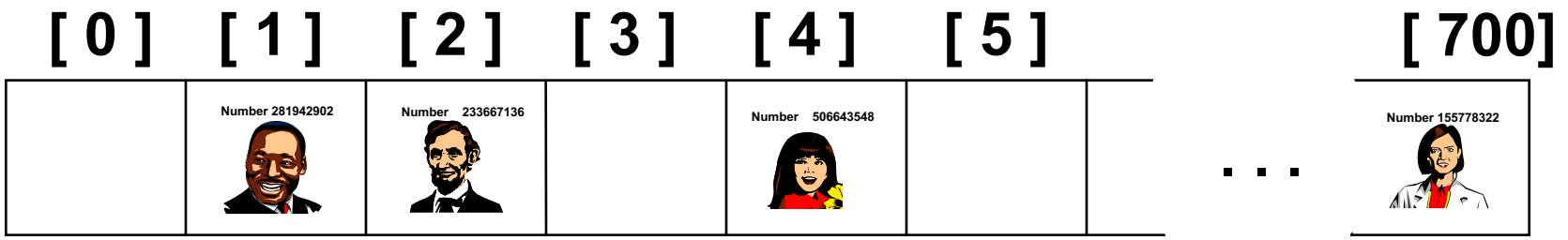
- When a hash table is in use, some spots contain valid records, and other spots are "empty".



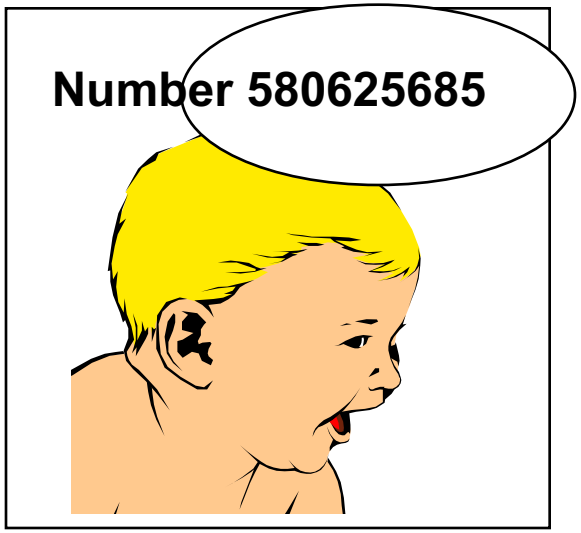
# Inserting a New Record




- 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.



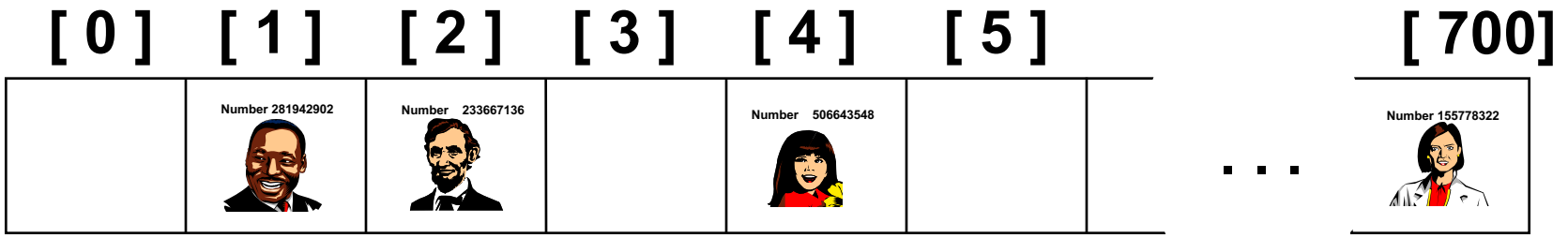
# Inserting a New Record




- Typical way to create a hash value:

  **(Number mod 701)**

*What is (580625685 mod 701)?*

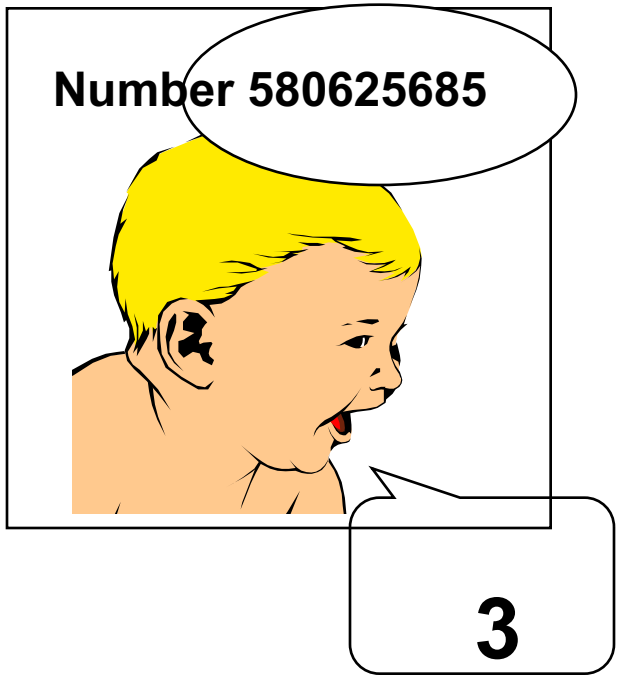


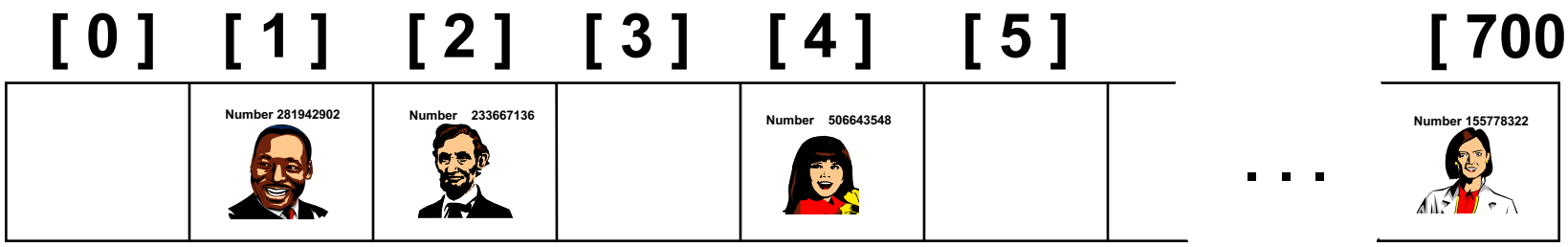
# Inserting a New Record

- Typical way to create a hash value:

  **(Number mod 701)**

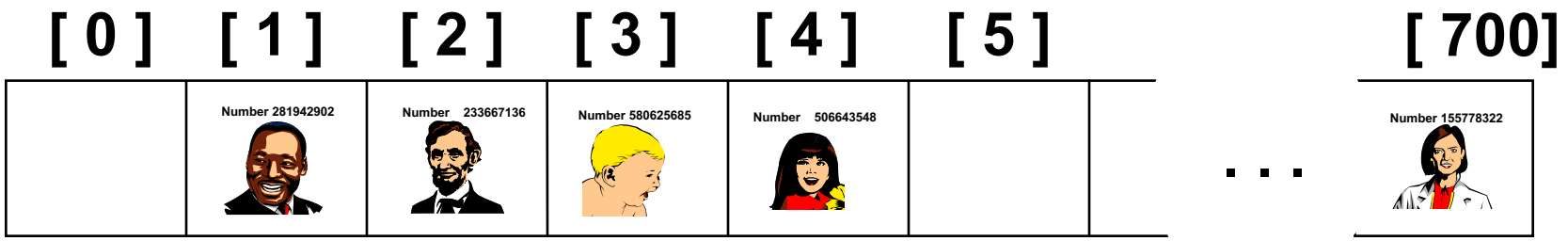
*What is (580625685 mod 701) ?*



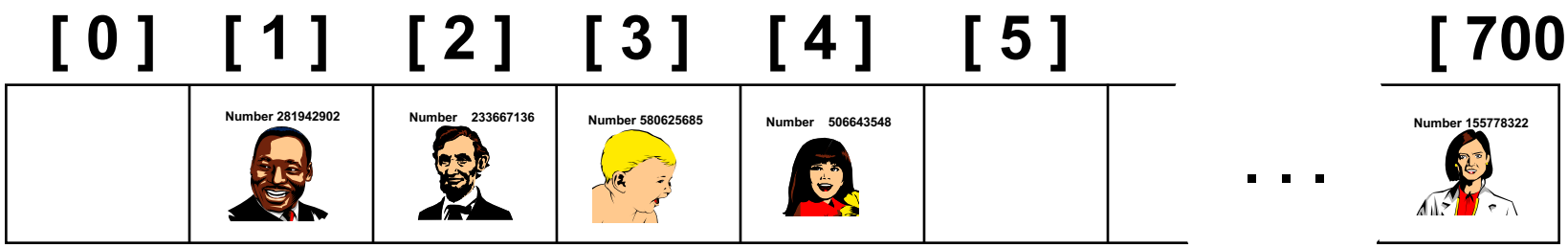
# Inserting a New Record

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



# Collisions

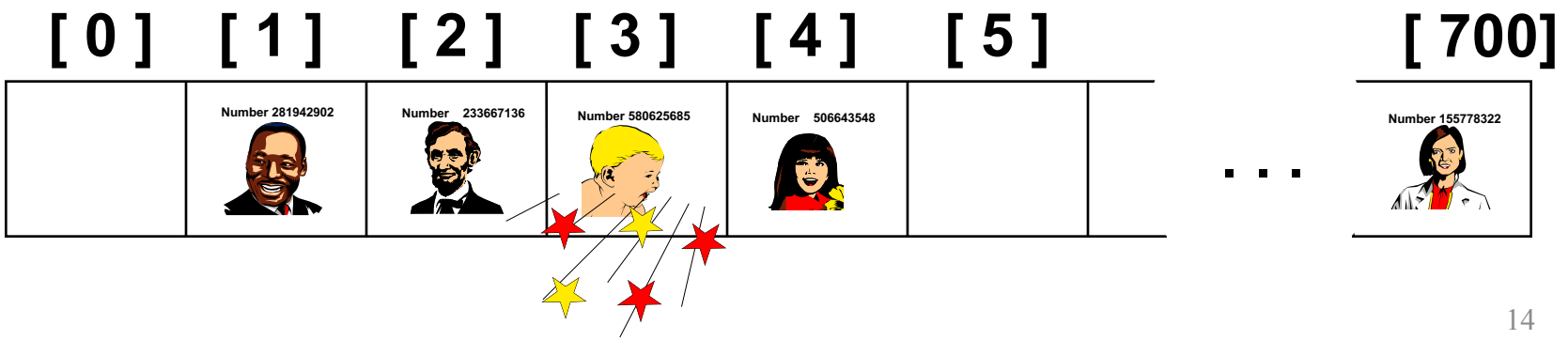
- Here is another new record to insert, with a hash value of 2.

# Collisions

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

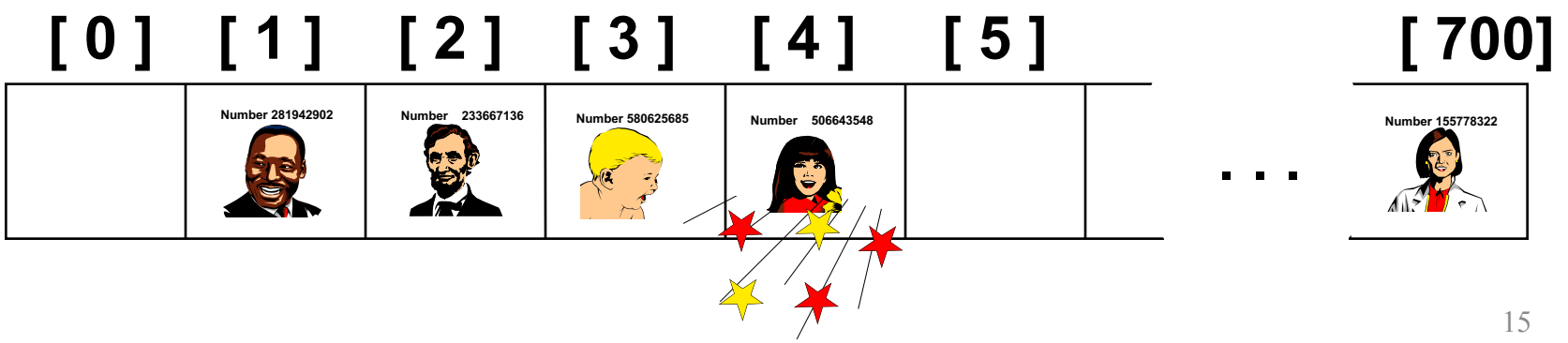
When a collision occurs, move forward until you find an empty spot.

# Collisions

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

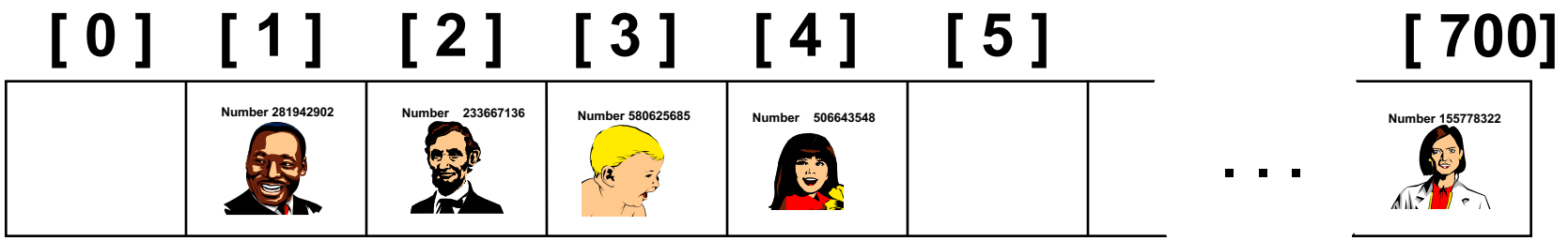

When a collision occurs, move forward until you find an empty spot.



# Collisions


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

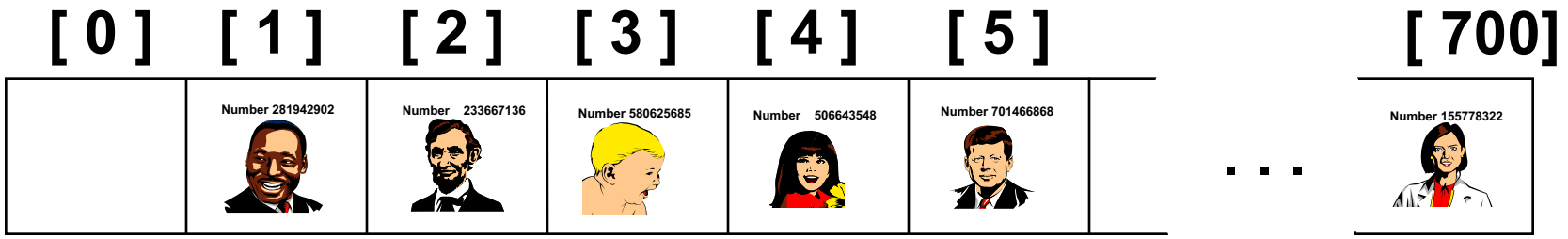
When a collision occurs, move forward until you find an empty spot.



# Collisions

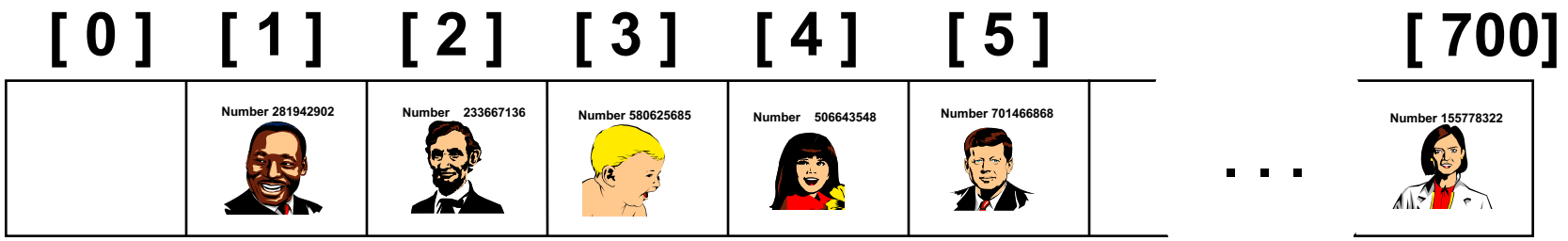
- This is called a **collision**, because there is already another valid record at [2].

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



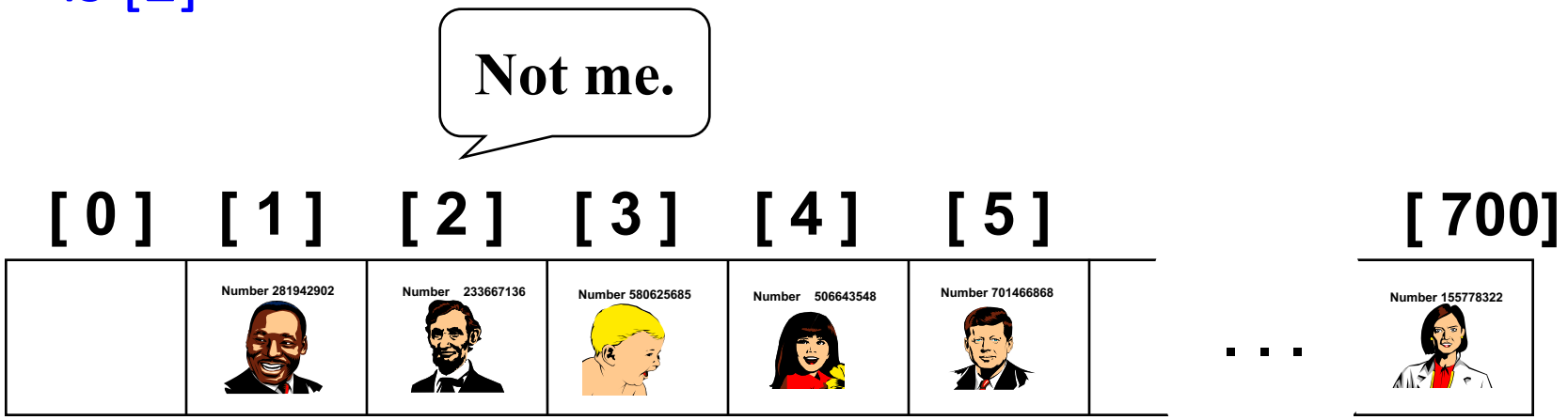
# Searching for a Key

- The data that's attached to a key can be found fairly quickly.
- Search 701466868



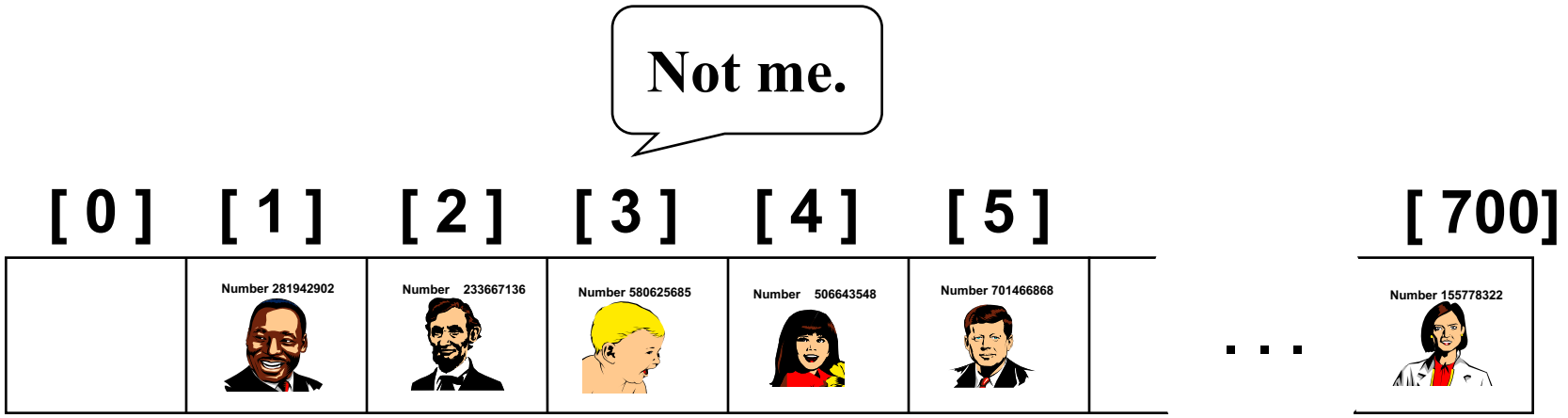
# Searching for a Key

- Calculate the hash value.
- Check that location of the array for the key.
- Search 701466868, hash value is [2]



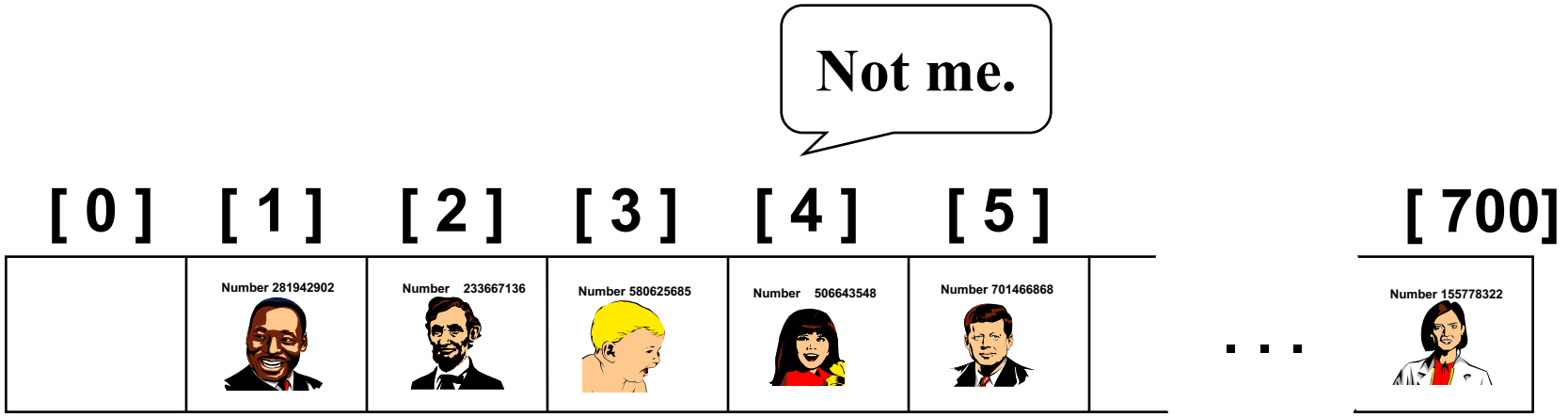
# Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.
- Search 701466868, hash value is [2]



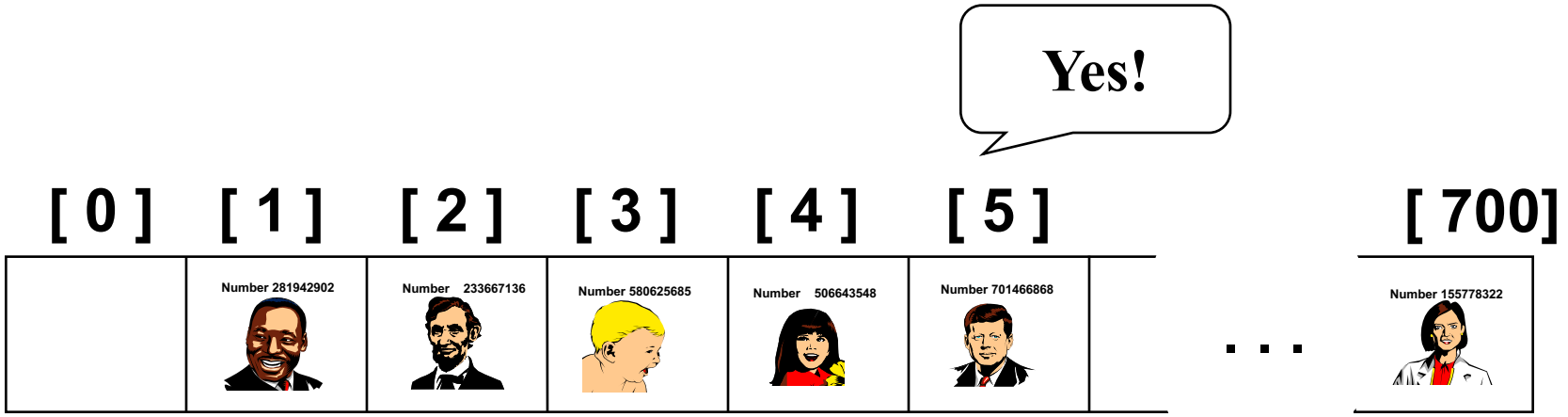
# Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.
- Search 701466868, hash value is [2]



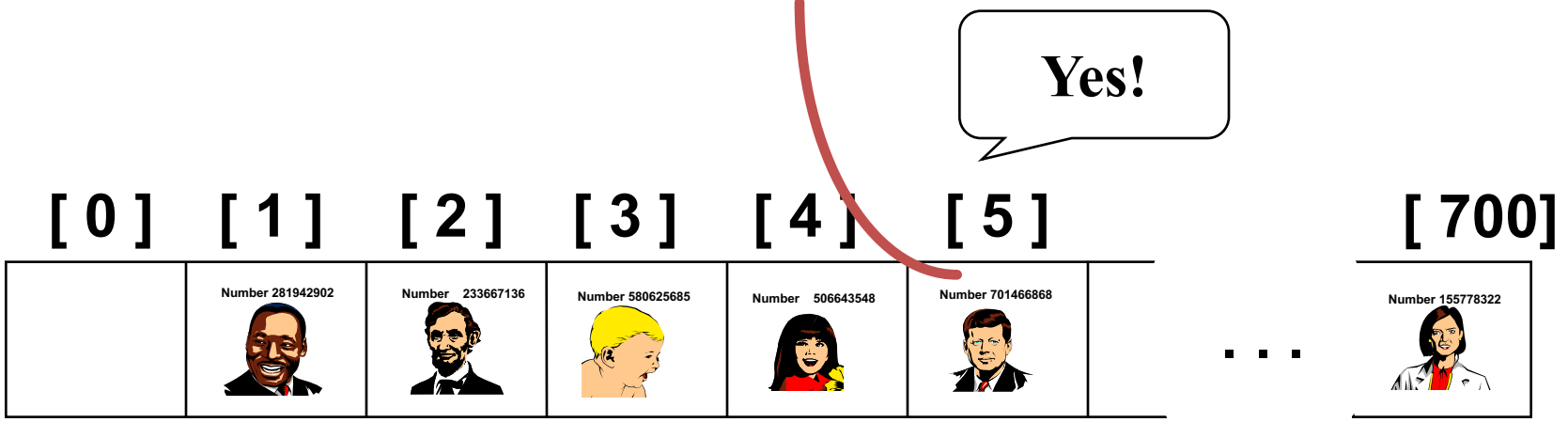
# Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.
- Search 701466868, hash value is [2]



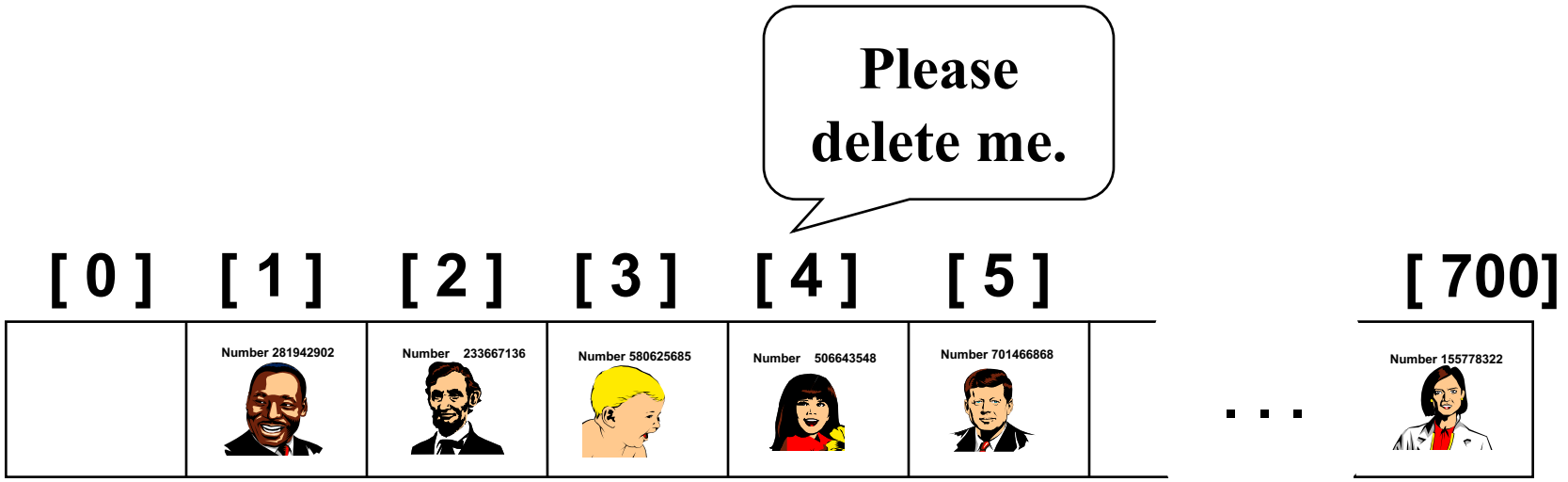
# Searching for a Key

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



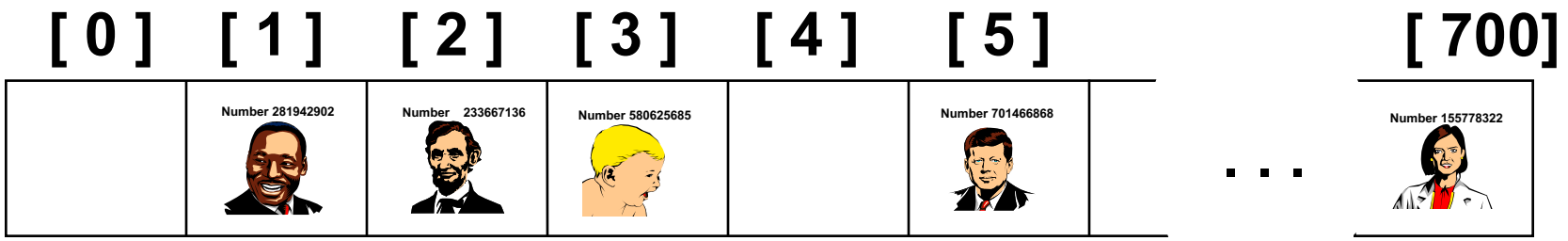
# Deleting a Record

- Records may also be deleted from a hash table.



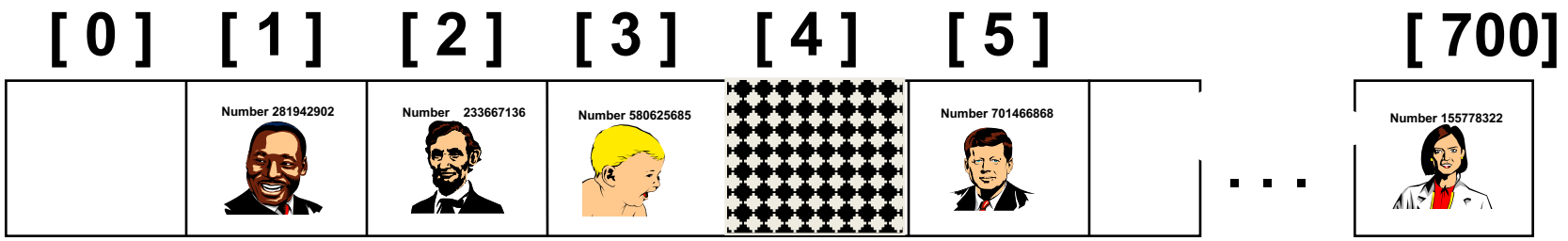
# Deleting a Record

- 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.



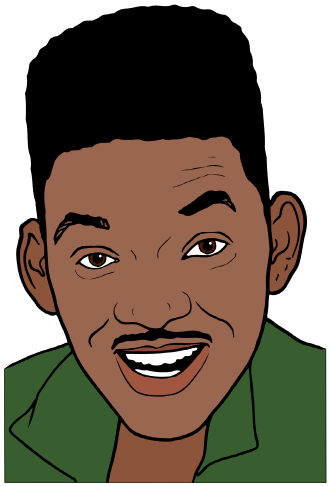
Kathy Martin
817339024

Took Data Structures in Fall 1993.
Grade A.

Hard worker. Always gets things done on time.

Currently working for Hewlett-Packard in Fort Collins.

Will Smith
506643973



Took Data Structures in Fall 1995.
Grade A.

A bit of a goof-off, but he comes through in a pinch.

Currently saving the world from alien invasion.

William “Bill” Clinton
330220393

Took Data Structures in Fall 1998.
Grade B-.

Gets along with most
people well.

Currently working for federal government.

Elizabeth Windsor
092223340

Took Data Structures in Fall 1997.
Grade B-.

Prefers to be called “Elizabeth II” or “Her Majesty.” Has some family problems.

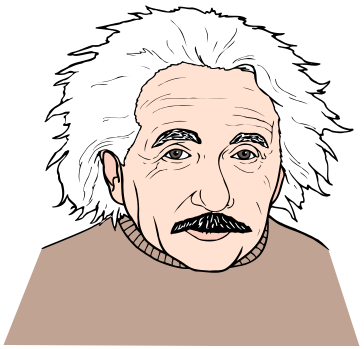
Currently working in public relations near London.

Al Einstein
699200102

Took CSCI 2270 in Fall 1995.
Grade F.

In spite of poor grade, I think there is good academic ability in Al.

Currently a well-known advocate for peace.



# Open-address hashing

- Object (e.g., student, person, computer)
  - Key
- Hash function
  - Example: key value % array length

# Table class

- Member variables
  - num: the number of elements in the table
  - Object[] keys;
  - Object[] data;
  - boolean[] used;
- Methods
  - public Object get(Object key)
  - public Object put(Object  key, Object obj)
  - public Object remove(Object  key)
  - private int findIndex (Object key)
  - private int hash(Object key)

# Hashcode function

```
Integer int1 = new Integer(10);
System.out.println(int1+" hashcode="+int1.hashCode());
Integer int2 = new Integer(1);
System.out.println(int2+" hashcode="+int2.hashCode());

Float flt1 = new Float(1.11);
System.out.println(flt1+" hashcode="+flt1.hashCode());
Float flt2 = new Float(2.11);
System.out.println(flt2+" hashcode="+flt2.hashCode());
```

Output:

```
10 hashcode=10
1 hashcode=1
1.11 hashcode=1066275963
2.11 hashcode=1074203197
```

# Hashcode function

- Keys are strings and other non-integers
  - Function hashcode() converts such keys to integers

```
String str1 = new String("obj1");
System.out.println(str1+" hashcode="+str1.hashCode());

String str2 = new String("obj13");
System.out.println(str2+" hashcode="+str2.hashCode());
```

- Output:

```
obj1 hashcode=3404314
obj13 hashcode=105533785
```

# Function hash()

```
private int hash(Object key)
{
    return Math.abs(key.hashCode())%data.length;
}
```

# Method get

public Object get(Object key)

- Calculate the index idx of key (findIndex(key))
- Case 1: this key does not exist  (i.e., idx==-1)
  - Return -1
- Case 2: this key exists (i.e., idx!=-1)
  - Return the object at index idx

# Function findIndex

private int findIndex (Object key)

- idx = hash value of key
- Initialize a counter=0
- while(counter< data length &used[idx] is true) //data[idx] keeps a search key
  - If key equals to data[idx], return idx;
  - Otherwise, move idx forward
  - Increment counter by 1
- Return -1//cannot find this search key

# Method put

public void put(Object  key, Object obj)

- Special case: the hash table is full
- Calculate the index idx of key (findIndex(key))
- Case 1: key exists (i.e., idx!=-1)
  - Directly set data[idx] to be obj
- Case 2: key does not exist (i.e., idx==-1)
  - idx = hash value of key
  - If (used[idx] is true), loop to find the next available idx
  - Let keys[idx] = key, data[idx]=obj, used[idx]=true, and increment num by 1

# Method remove

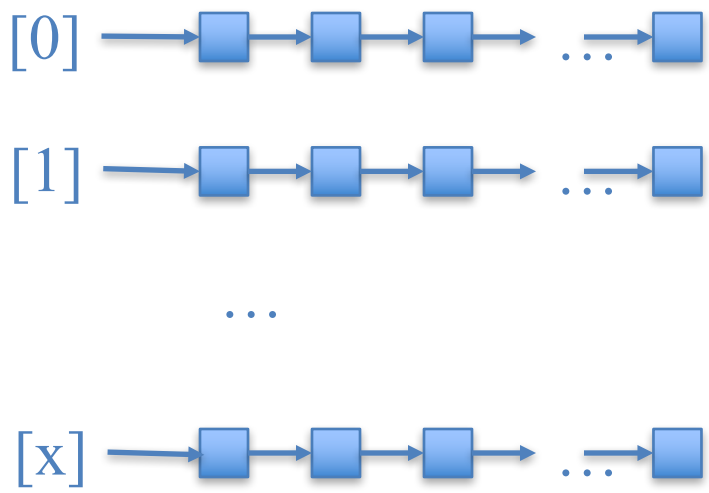
public Object remove(Object  key)

- Calculate the index idx of key (findIndex(key))
- Case 1: key does not exist (i.e., idx==-1)
  - Directly return null
- Case 2: key exists (i.e., idx!=-1)
  - Remember answer = data[idx]
  - Let keys[idx] = null, data[idx]=null, used[idx]=false, and decrement num by 1

# Separate chaining

Instance variables:

- Number of linked lists
- Array of the heads of linked lists (SequentialSearchST<K, V>)



# SequentialSearchST<K, V>

- Instance variables:
  - Node<K, V> first;
- Methods
  - put(K Key, V val);
  - V get (K key)
  - boolean contains(K key)
  - int size()

# Separate chaining

```
public class HashTableSeparateChain<K,V> {
    private int M;
    private SequentialSearchST<K,V>[] st ;

    public HashTableSeparateChain () {
        this (997) ;
    }
    private int hash (K key){…}
    public V get (K key) {…}
    public void put (K key, V val) {…}
}
```

# Hash functions

- Hash function: transforms keys into table addresses.
- If we have a table that can hold M items, then we need a function that transforms keys into integers in the range [0, M – 1].
- An ideal hash function is easy to compute and approximate a random function: for each input, every output should be in some sense equally likely.
- The hash function depends on the key type.
  - (a) Integers or floating-point keys can typically be hashed with just a single machine operation.
  - (b) String keys and other types of compound keys require more attention to efficiency.

# Hash functions

- String: Transform the keys piece by piece.

```
static int hash(String s, int M){
    int h=0, a = 127;
    for(int i=0;i<s.length(); i++)
        h = (a*h + s.charAt(i))%M;
    return h;
}
```

- Compound keys: If a key type has multiple integer fields. Consider a key of type Date
  - int hash = ((( day * R + month) % M ) * R + year) %M

# hashcode() and equals()

- The implementation of hashcode() must be consistent with equals.
  - (i) If a.equals(b) is true, then a.hashcode() must be the same as b.hashcode().
  - (ii) If the hashcode() of two objects are different, the two objects must be different.
  - (iii) If the hashcode() of two objects are the same, they may or may not bee equal. We must use equals() to decide whether they are the same.

# More about hash() function

```
private int hash (Key x){
    return (x.hashCode() & 0x7fffffff) % M;
}
```

0x7fffffff is to mask off the sign bit (to turn the 32-bit number into a 31-bit nonnegative integer).

# 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.
- 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.