



CS-2001 DATA STRUCTURE

Dr. Hashim Yasin

**National University of Computer
and Emerging Sciences,
Faisalabad, Pakistan.**

HASHING

What is a database?

3

- Structured collection of data.

Student ID	First Name	Last Name	Email	Major	Faculty
200120	Kate	West	kwest@email.com	Music	Arts
200121	Julie	McLain	jmclain@email.com	Finance	Business
200122	Tom	Erlich	terlich@email.com	Sculpture	Arts
200123	Mark	Smith	msmith@email.com	Biology	Science
200124	Jen	Foster	jfoster@email.com	Physics	Science
200125	Matt	Knight	mknight@email.com	Finance	Business
200126	Karen	Weaver	kweaver@email.com	Music	Arts
200127	John	Smith	jsmith@email.com	Sculpture	Arts
200128	Allison	Page	apage@email.com	History	Humanities
200129	Craig	Cambell	ccambell@email.com	Music	Arts
200130	Steve	Edwards	sedwards@email.com	Biology	Science
200131	Mike	Williams	mwilliams@email.com	Linguistics	Humanities
200132	Jane	Reid	jreid@email.com	Music	Arts

The Dictionary ADT

4

- A dictionary (table) is an **abstract model** of a database
- A dictionary stores **key-element pairs**
- The main operation supported by a dictionary is searching by key

Dictionary

5

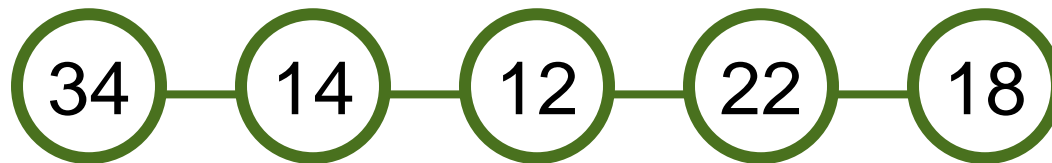
- Collection of pairs.
 - ▣ (key, value)
 - ▣ Each pair has a unique key.
- Operations.
 - ▣ `Get(theKey)`
 - ▣ `Delete(theKey)`
 - ▣ `Insert(theKey, theValue)`

Implementing a Dictionary

6

□ Unordered sequence

- searching and removing takes $O(n)$ linear time
- inserting takes $O(1)$ constant time
- applications to log files (*frequent insertions, rare searches, and removals*)
- For Example: 34 14 12 22 18



Implementing a Dictionary

7

□ Array-based ordered sequence

(assume keys can be ordered)

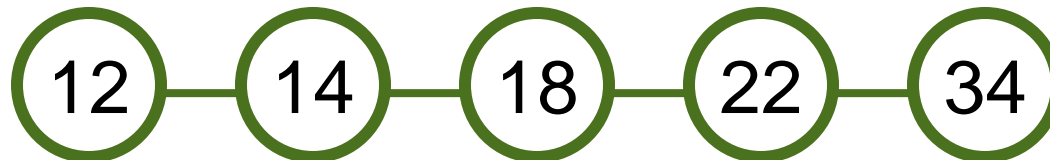
- searching takes $O(\log n)$ time (**binary search**)

- inserting and removing takes $O(n)$ time

- application to lookup tables

(**frequent searches**, **rare insertions**, and **removals**)

□ For Example: 34 14 12 22 18



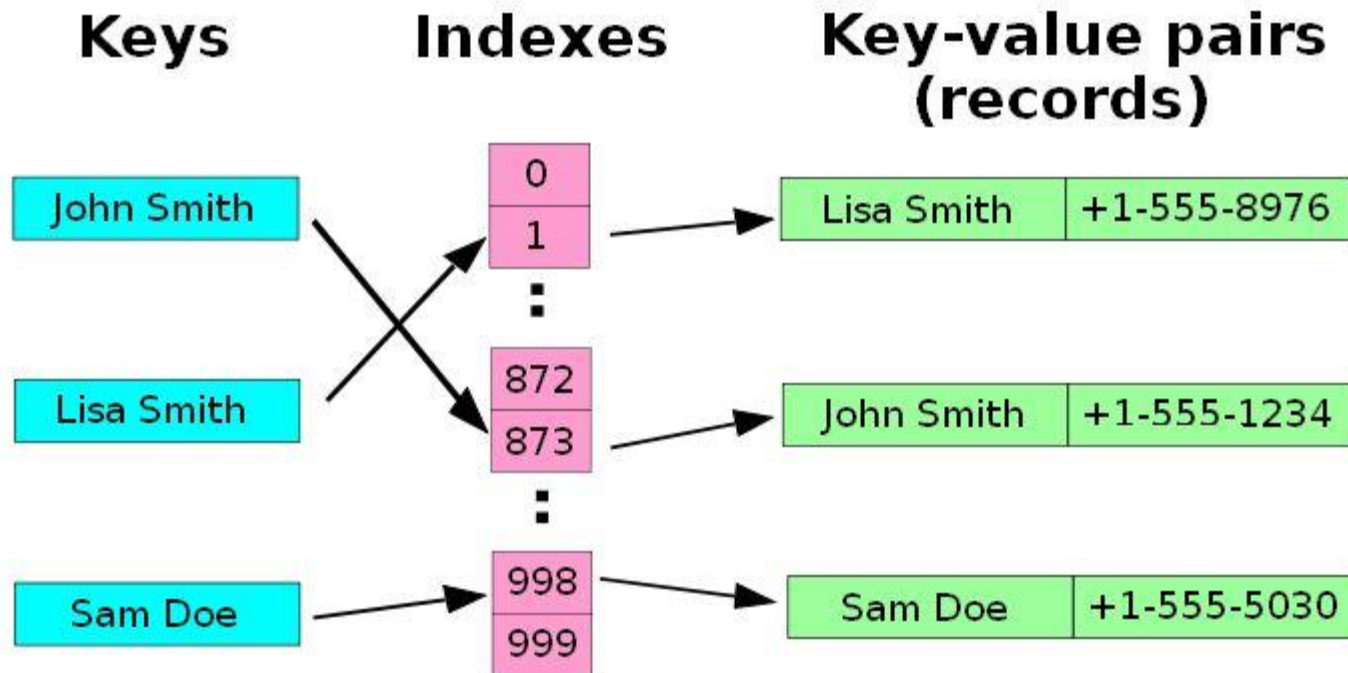
Concept of Hashing

8

- A **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).
 - ▣ Look-Up Table
 - ▣ Dictionary
 - ▣ Cache
 - ▣ Extended Array

Example

9



A small phone book as a hash table.

(Figure is from Wikipedia)

Hash Table

10

- Hash table:
 - ▣ Collection of pairs,
 - ▣ Lookup function (Hash function)
- Hash tables are often used to implement associative arrays,
 - ▣ Worst-case time for **Get**, **Insert**, and **Delete** is $O(\text{size})$.
 - ▣ Expected time is $O(1)$.

Origins of the Term

11

- The term "hash" comes by way of analogy with its standard meaning in the physical world, to "chop and mix." D. Knuth notes that Hans Peter Luhn of IBM appears to have been the first to use the concept, in a memo dated January 1953; the term hash came into use some ten years later.

Applications

12

- Keeping track of **customer account information** at a bank
 - ▣ Search through records to check balances and perform transactions
- Keep track of **reservations on flights**
 - ▣ Search to find empty seats, cancel/modify reservations
- **Search engine**
 - ▣ Looks for all documents containing a given word

Search vs. Hashing

13

- **Search tree methods:** key comparisons
 - ▣ Time complexity: $O(\text{size})$ or $O(\log n)$
- **Hashing methods:** hash functions
 - ▣ Expected time: $O(1)$
- **Types**
 - ▣ Static hashing
 - ▣ Dynamic hashing

Static Hashing

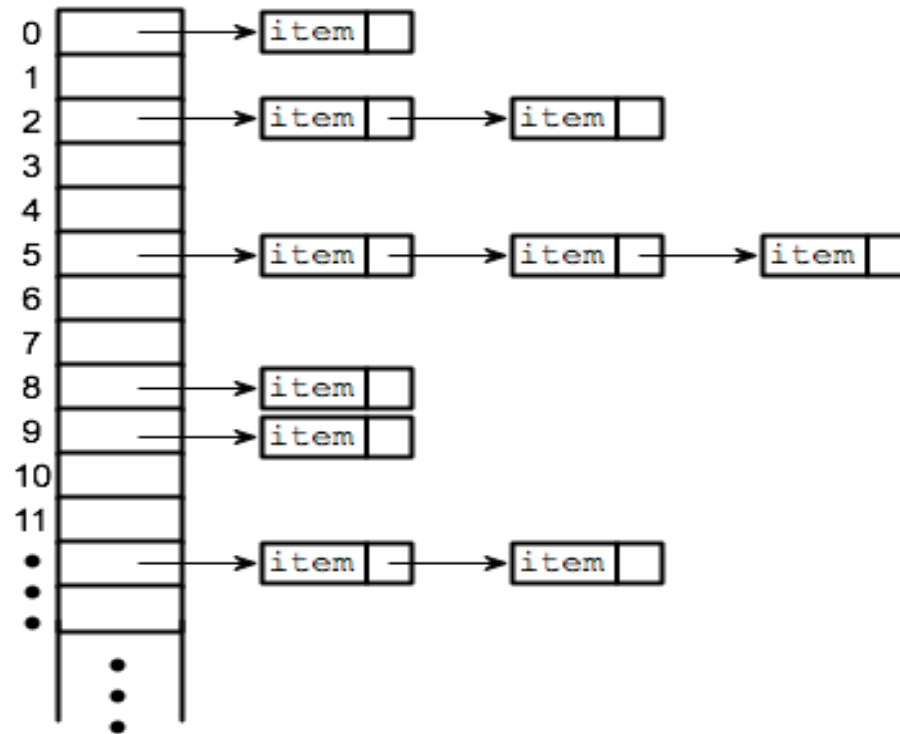
14

- Key-value pairs are stored in a **fixed size table** called a **hash table**.
 - ▣ A hash table is partitioned into many **buckets**.
 - ▣ Each bucket has many **slots**.
 - ▣ Each slot holds one record.
- A **hash function $f(x)$** transforms the identifier (key) into an address in the hash table.

Hash Table

15

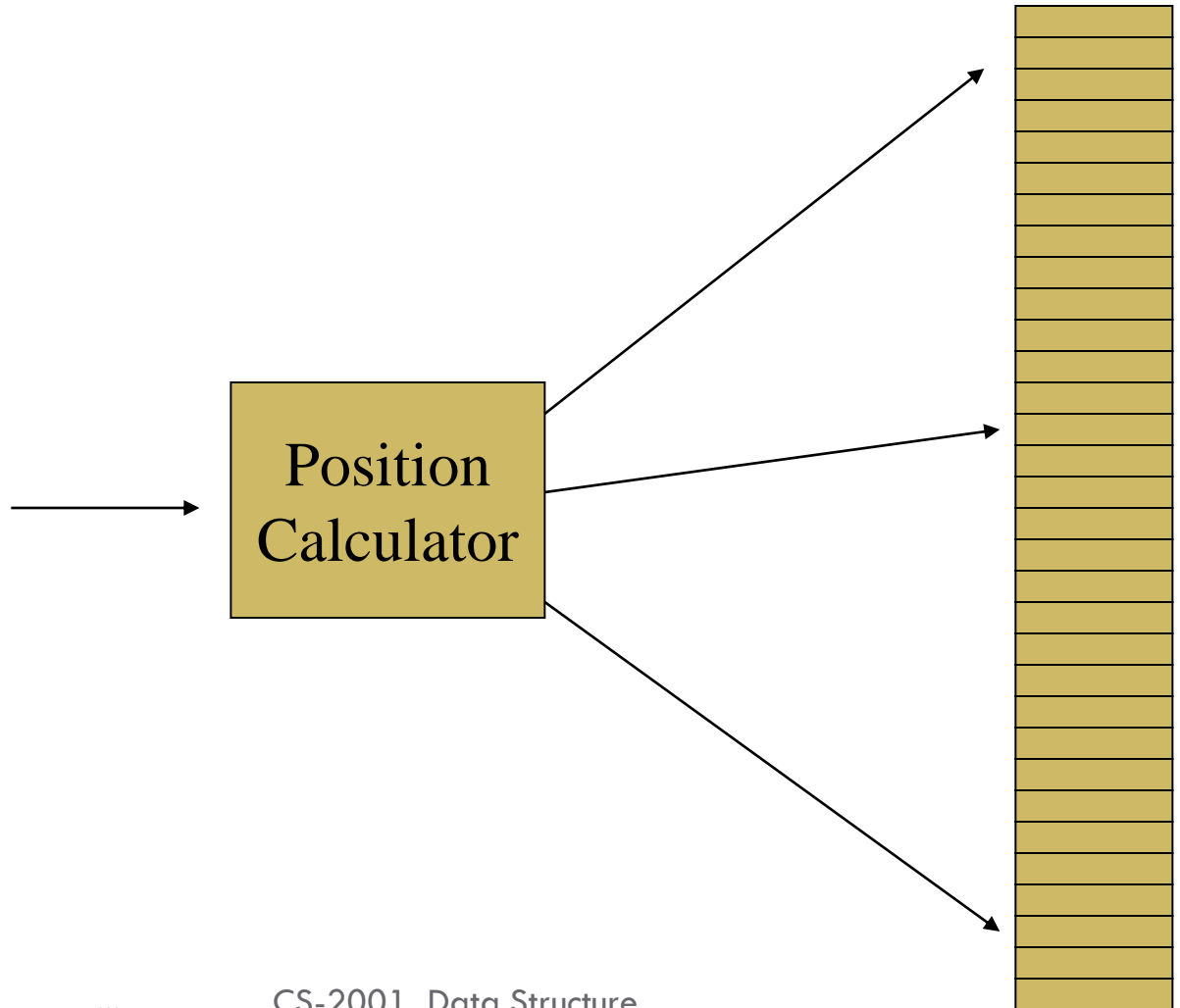
- Tables that can be searched for an item in **$O(1)$** time using a hash function to form an address from the key.



Hashing Engine

16/51

□ **itemKey**



Hashing – A Simple Scenario

17

- A list of employees of a fairly small company.
- Each of the 100 employees has an ID number in the range 0 to 99,
- We have to access the employee records using the key *idNum*.

Hashing- A complicated Scenario

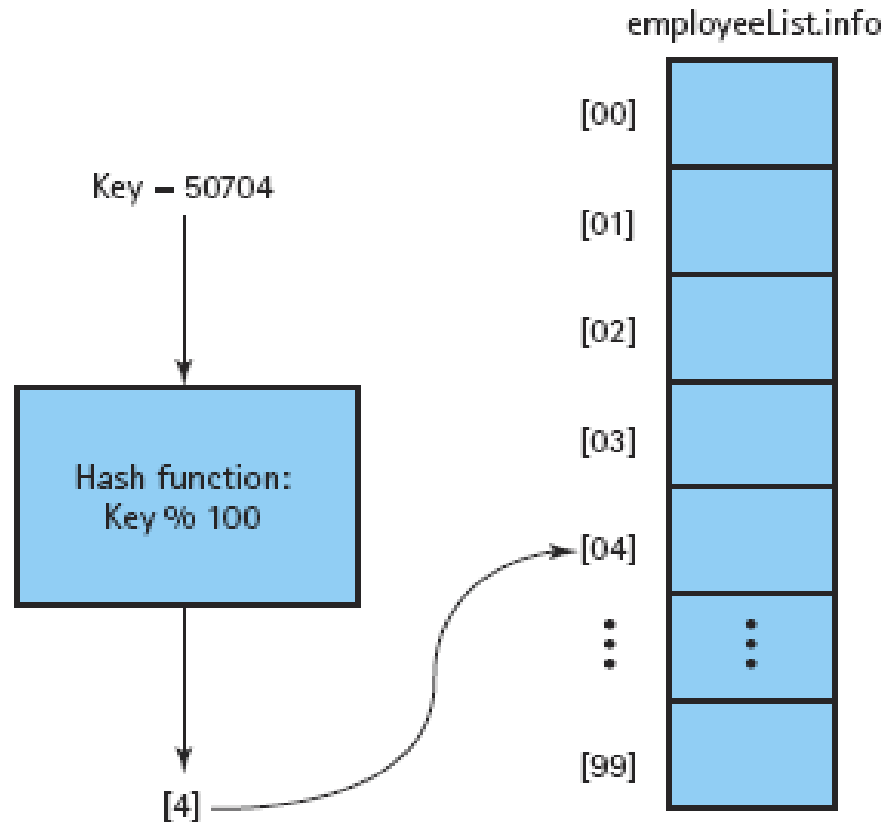
18

- A similar small company that uses its employees **five-digit ID number** as the primary key,
- Number of employees are still 100.
- Use a **hash function** to determine the exact location, e.g.,

```
Hash(){  
    return (idNum % MAX_ITEMS);  
}
```

Hashing- A complicated Scenario

19

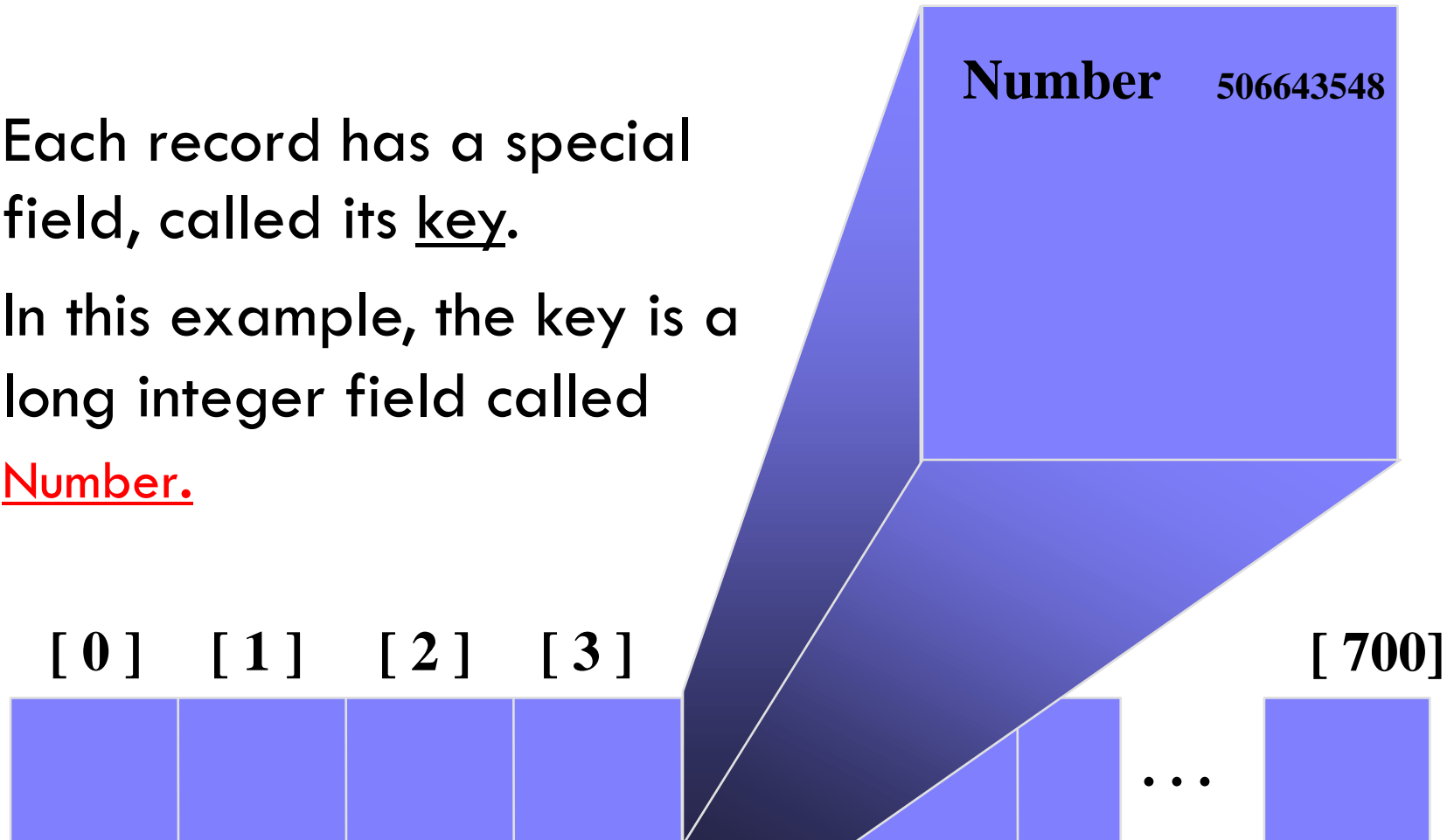


What is a Hash Table ?

20

[4]

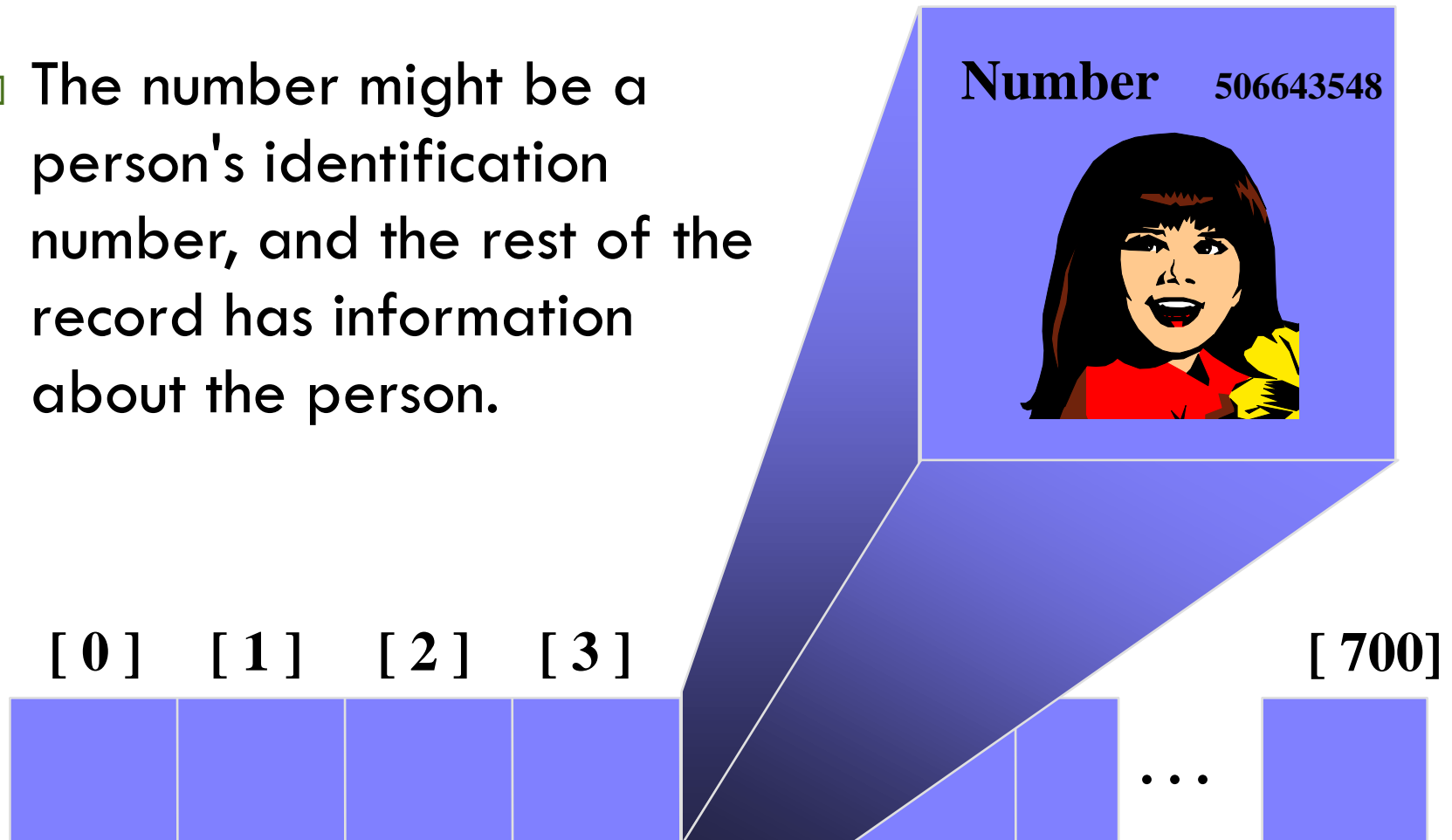
- Each record has a special field, called its key.
- In this example, the key is a long integer field called Number.



What is a Hash Table ?

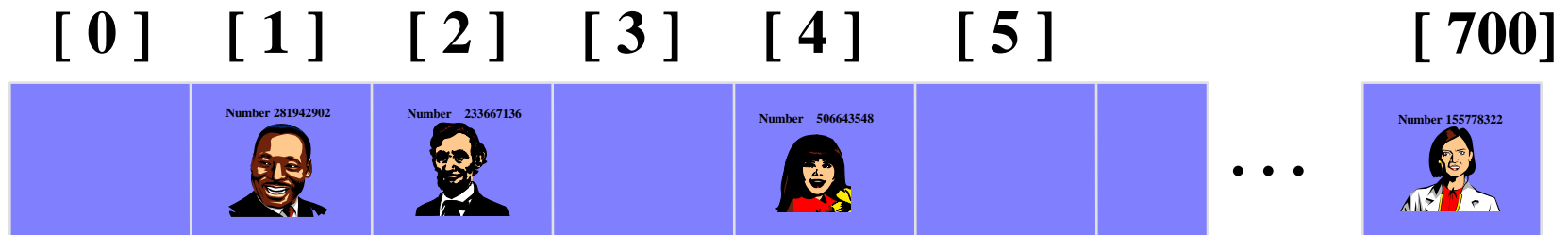
21

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



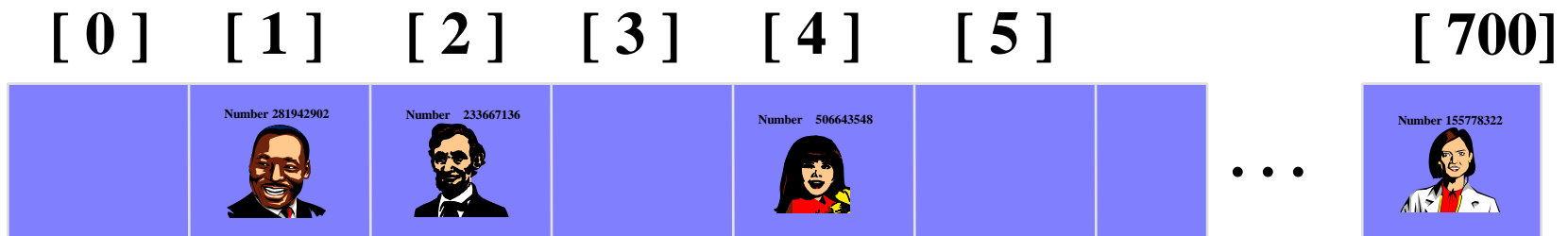
What is a Hash Table ?

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



Inserting a New Record

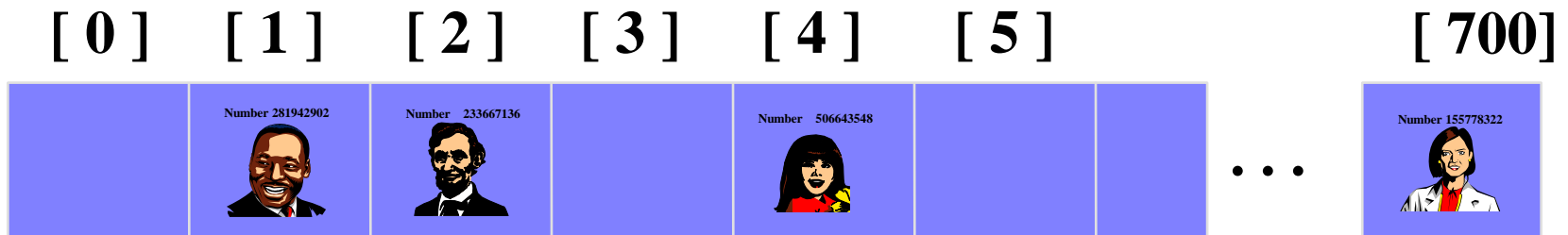
- In order to insert a new record, the **key** must somehow be **converted to** an array **index**.
- The index is called the **hash value** of the key.



Inserting a New Record

- **Typical way to create a hash value:**

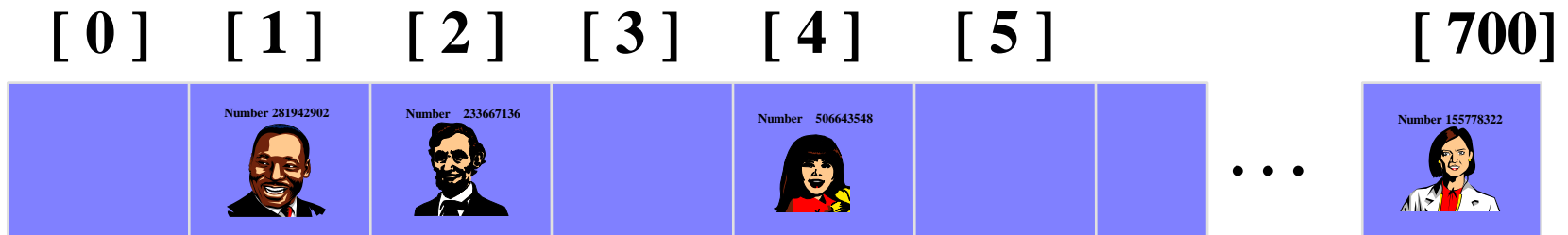
What is $(580625685 \bmod 701)$?



Inserting a New Record

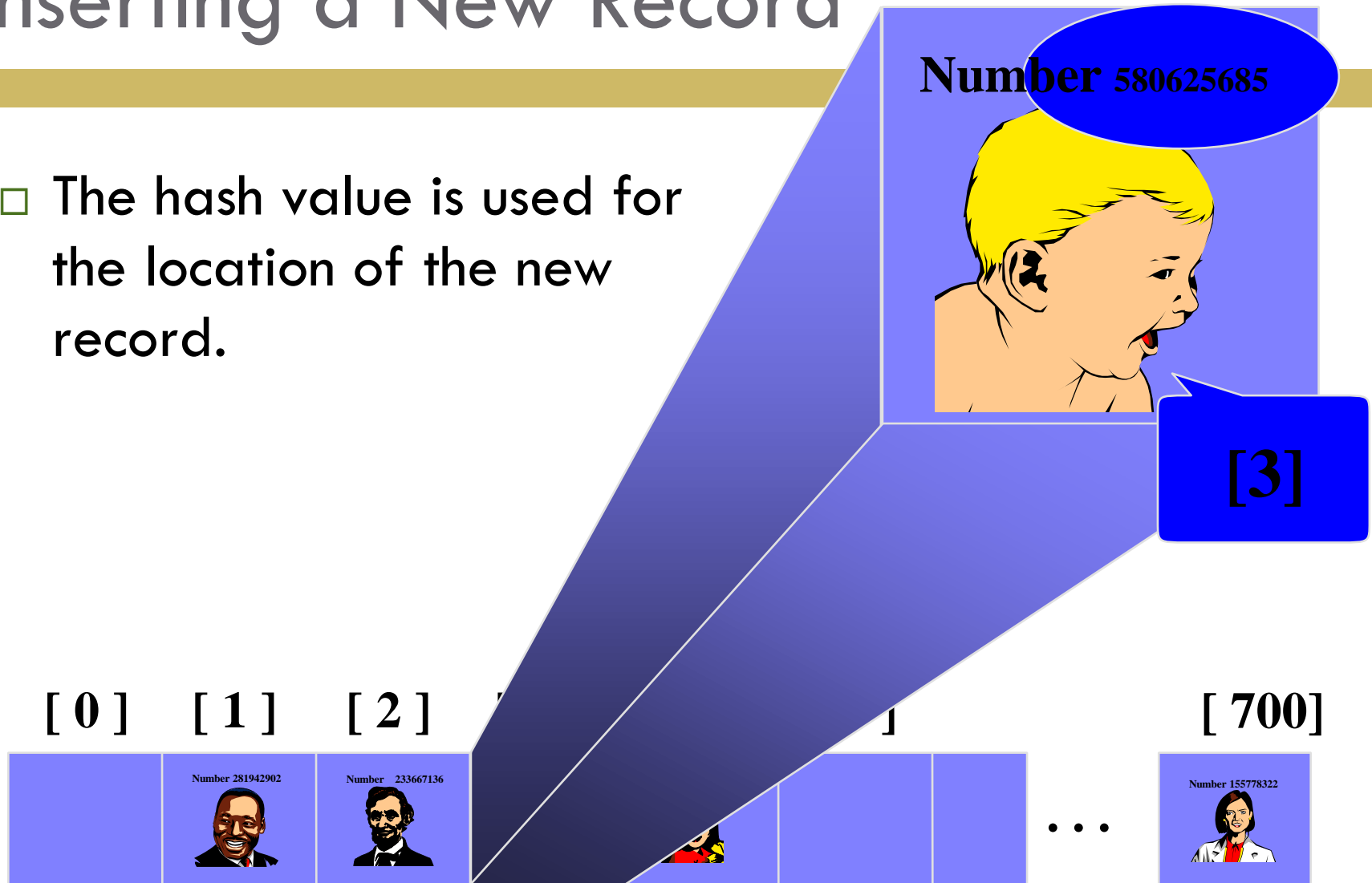
- **Typical way to create a hash value:**

What is $(580625685 \bmod 701)$?



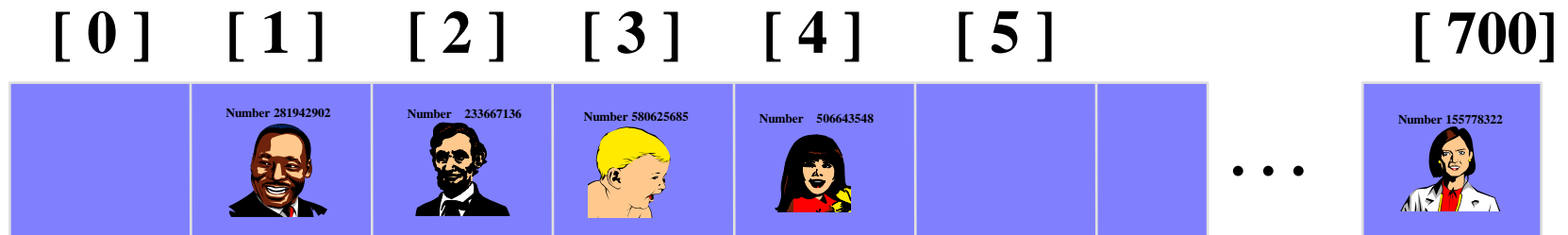
Inserting a New Record

- 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.

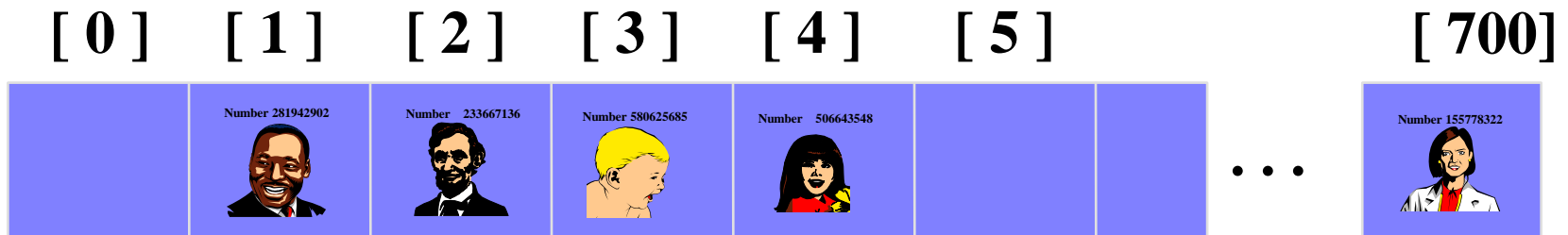


Collisions

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



My hash value is [2].

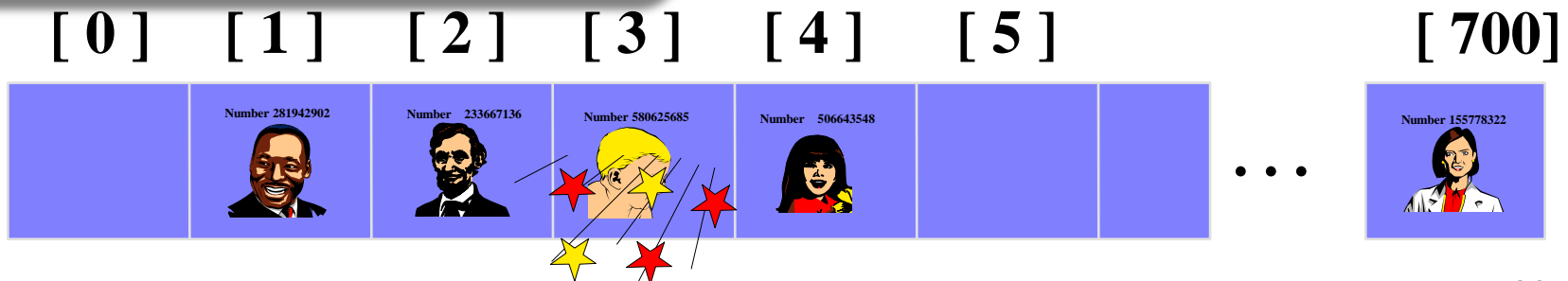


Collisions

- 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.

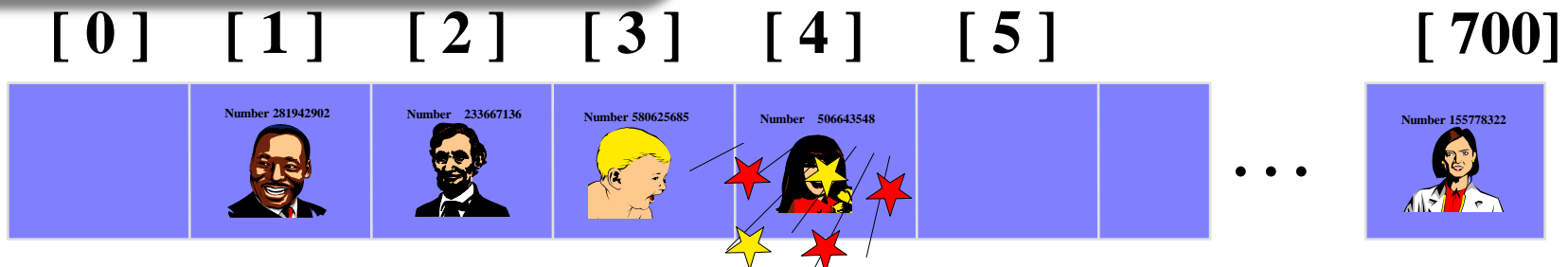


Collisions

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



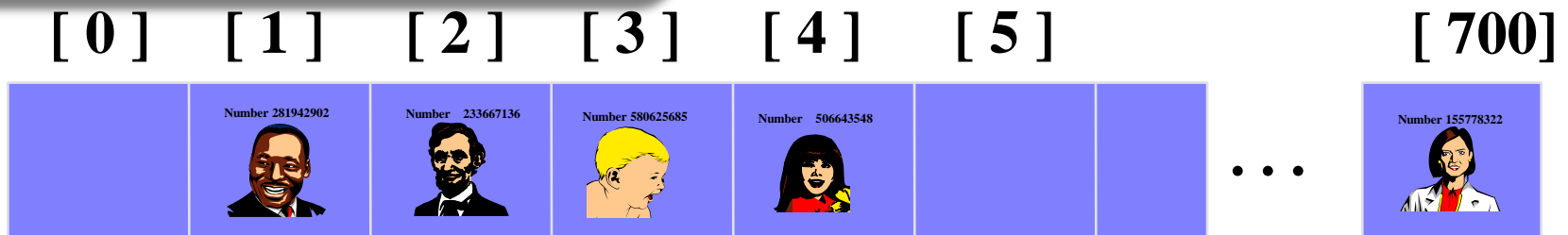
When a collision occurs, move forward until you find an empty spot.



Collisions

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

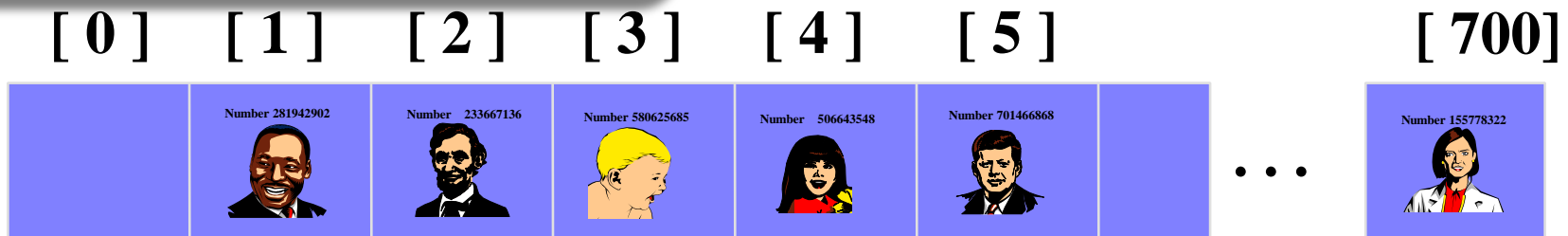
When a collision occurs, move forward until you find an empty spot.



Collisions

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

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



Some Issues

33

- **Choice of hash function.**

- *Really tricky!*

- To avoid **collision** (two different pairs are in the same bucket.)

- Size (number of buckets) of hash table.

- **Overflow handling method.**

- **Overflow**: there is no space in the bucket for the new pair.

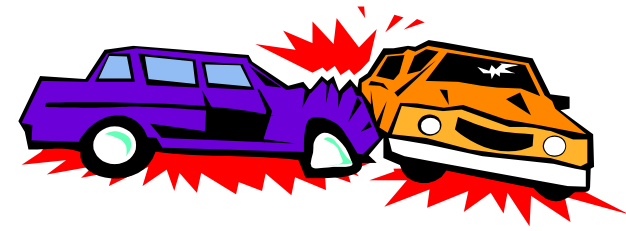
Choice of Hash Function

34

□ Requirements

- easy to compute
- minimal number of collisions
- If a hashing function groups key values together, this is called **clustering** of the keys.
- A good hashing function distributes the key values uniformly throughout the range.

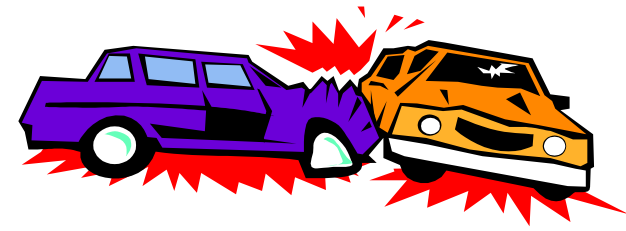
Collision



35

- The condition resulting when two or more keys produce the same hash location.
- A good hash function minimizes collisions by spreading the elements uniformly throughout the array.

Collision



36

- Collision handling techniques
 - ▣ Linear Probing
 - ▣ Rehashing
 - ▣ Double Hashing
 - ▣ Quadratic Probing
 - ▣ Random Probing
 - ▣ Buckets
 - ▣ Chaining

Collision Resolution Techniques

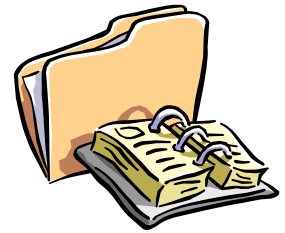
37

- There are two broad ways of collision resolution:

1. Open Addressing: Array-based implementation.

- (i) Linear probing (linear search)
- (ii) Quadratic probing (nonlinear search)
- (iii) Double hashing (uses two hash functions)

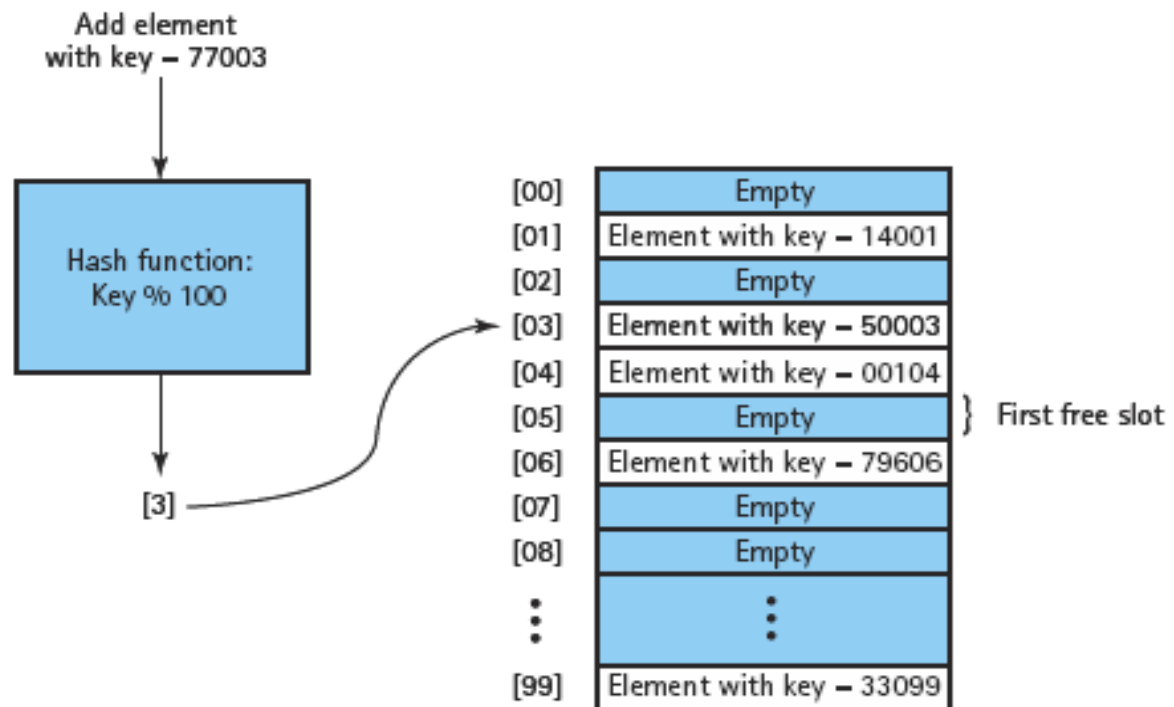
2. Separate Chaining: A linked list implementation



Linear Probing

38

- *Resolving a hash collision by sequentially searching* a hash table *beginning at the location return by the hash function.*



Linear Probing - Problem

39

- What happens if the key hashes to the last index in the array and that space is in use?
- **Solution**
 - ▣ We can consider the array to be a circular structure and continue looking for an empty slot at the beginning of the array

Linear Probing - Searching

40

To search for an element using Linear probing:

- Perform the hash function on the key
- Compare the desired key to the actual key in the element at the designated location
- If the keys do not match use linear probing beginning at the next slot in the array
- If the key is found return true
- If not found return false

Linear Probing - Example

41

- divisor = b (number of buckets) = 17.
- Home bucket = $\text{key} \% 17$.
- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

0	4				8				12				16				
34	0	45				6	23	7				28	12	29	11	30	33

Linear Probing - Example

42

- $h(k) = k \bmod 13$
- Insert keys:
 - ▣ 18 41 22 44 59 32 31 73

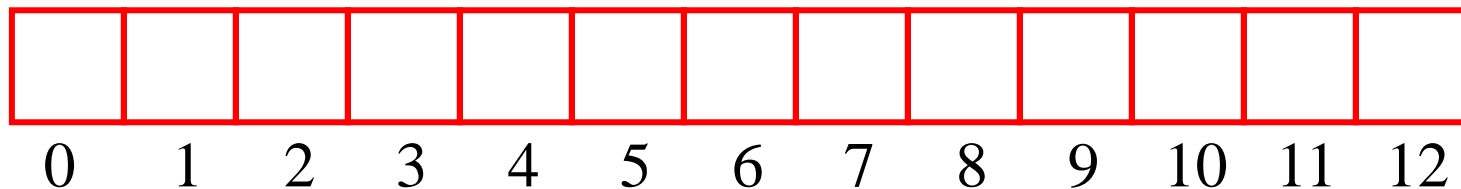
Linear Probing - Example

43

□ $h(k) = k \bmod 13$

□ Insert keys:

▣ 18 41 22 44 59 32 31 73



Linear Probing - Example

44

□ $h(k) = k \bmod 13$

□ Insert keys:

▣ 18 41 22 44 59 32 31 73

--	--	--	--	--	--	--	--	--	--	--	--	--

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

		41			18	44	59	32	22	31	72	
--	--	----	--	--	----	----	----	----	----	----	----	--

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

Linear Probing - Deletion

45

To delete an element using Linear probing

- Find the element with the same search approach
- **Replace the element with a constant** to identify the place that was previously occupied
- It will help pre-mature termination of the loop for searching.

Linear Probing - Deletion

46

- What if there are many deletions?
 - ▣ Reduced searching efficiency

- Conclusion
 - ▣ Hash Tables are not efficient where there are many deletions

Linear Probing - Deletion

47

0	4				8				12				16			
34	0	45				6	23	7			28	12	29	11	30	33

□ **Delete(0)**

0	4				8				12				16			
34		45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill the vacated bucket.

0	4				8				12				16			
34	45					6	23	7			28	12	29	11	30	33

Linear Probing - Deletion

48

0	4				8				12				16				
34	0	45				6	23	7				28	12	29	11	30	33

□ **Delete(34)**

0	4				8				12				16			
	0	45				6	23	7			28	12	29	11	30	33

□ Search cluster for pair (if any) to fill the vacated bucket.

0	4				8				12				16			
0		45				6	23	7			28	12	29	11	30	33

0	4				8				12				16			
0	45					6	23	7			28	12	29	11	30	33

Linear Probing - Deletion

49

0	4				8				12				16				
34	0	45				6	23	7				28	12	29	11	30	33

□ **Delete(29)**

0	4				8				12				16			
34	0	45				6	23	7			28	12		11	30	33

□ Search cluster for pair (if any) to fill vacated bucket.

0	4				8				12				16			
34	0	45				6	23	7			28	12	11		30	33

0	4				8				12				16			
34	0	45				6	23	7			28	12	11	30		33

0	4				8				12				16				
34	0					6	23	7				28	12	11	30	45	33

Implementation of Linear Probing

50

- **Search operation** for locating index:

```
private int findIndex(long key) {  
    // return -1 if the item with key 'key' was not found  
    int index = h(key);  
    int probe = index;  
    int k = 1; // probe number  
    do {  
        if (table[probe]==null) {  
            // probe sequence has ended  
            break;  
        }  
        if (table[probe].getKey()==key)  
            return probe;  
        probe = (index + step(k)) % table.length; // check next slot  
        k++;  
    } while (probe!=index);  
    return -1; // not found  
}
```

Implementation of Linear Probing

51

□ Find and Deleting the item:

```
public T find(long key) {  
    int index = findIndex(key);  
    if (index >= 0)  
        return (T) table[index];  
    else  
        return null; // not found  
}
```

```
public T delete(long key) {  
    int index = findIndex(key);  
    if (index >= 0) {  
        T item = (T) table[index];  
        table[index] = AVAILABLE; // mark available  
        return item;  
    } else  
        return null; // not found  
}
```

References

52

- Nell Dale – Chapter 10.
- <http://www.cplusplus.com/doc/tutorial/templates/>
- Robert Lafore, Chapter 14, Page 681

