Hash Tables

1

Basics

2

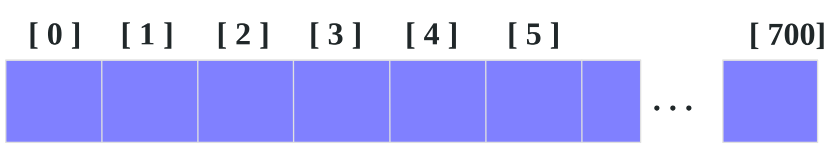
Assume that you have an object and you want to assign a key to it to make searching easy.

How could you store the object?

What happen when the keys are large?

What is a Hash Table ?

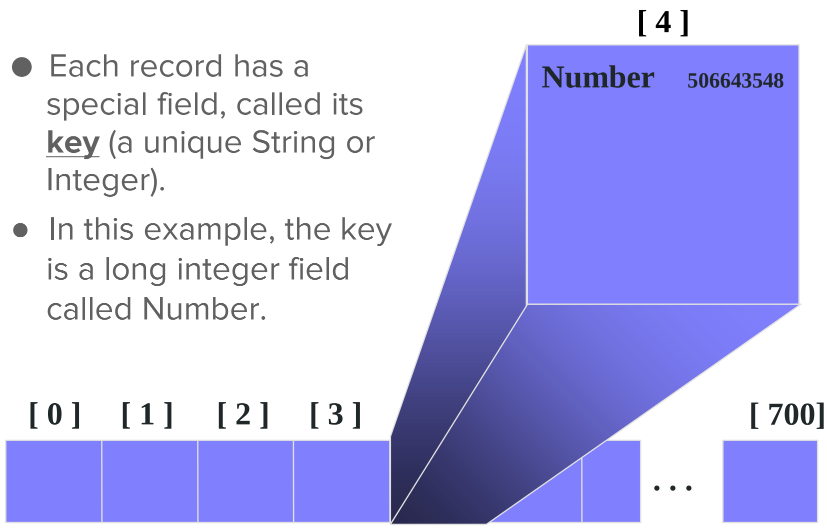
- The simplest kind of hash table is an array of records.
- Provides virtually direct access to objects.



An array of records

What is a Hash Table ?

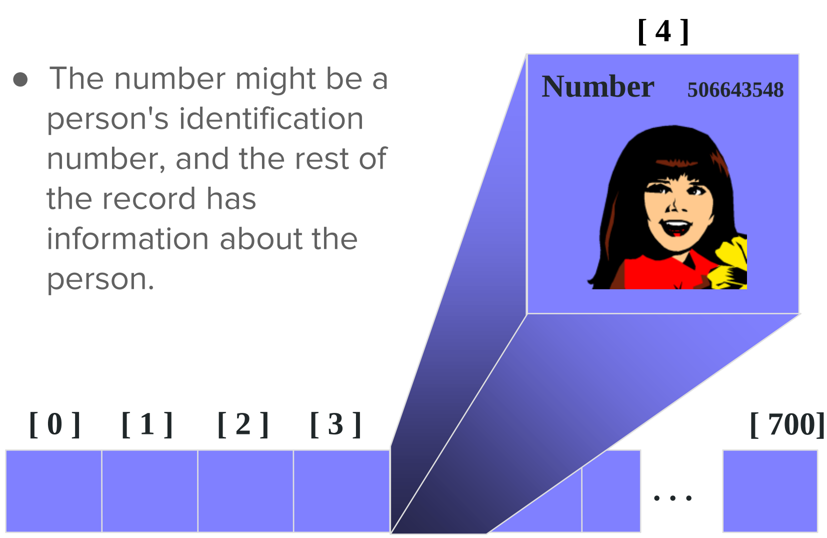
- Each record has a special field, called its **key** (a unique String or Integer).
- In this example, the key is a long integer field called Number.



5

What is a Hash Table ?

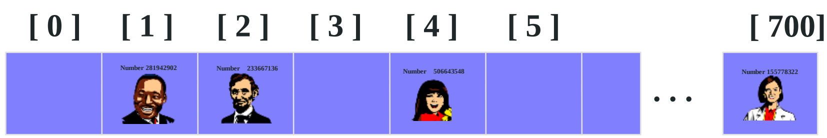
- The number might be a person's identification number, and the rest of the record has information about the person.



6

What is a Hash Table ?

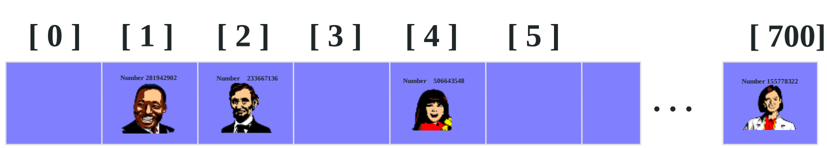
- When a hash table is in use, some spots contain valid records, and other spots are "empty".



7

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.



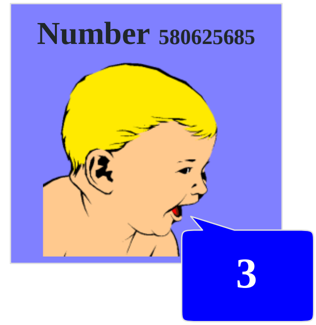
8

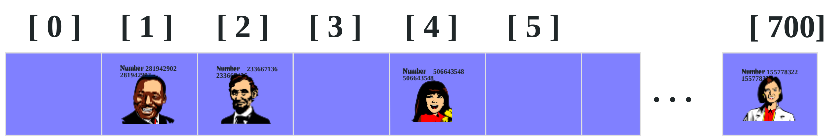
Inserting a New Record

- Typical way to create a hash value:

(Number % 701)

What is (580625685 mod 701) ?

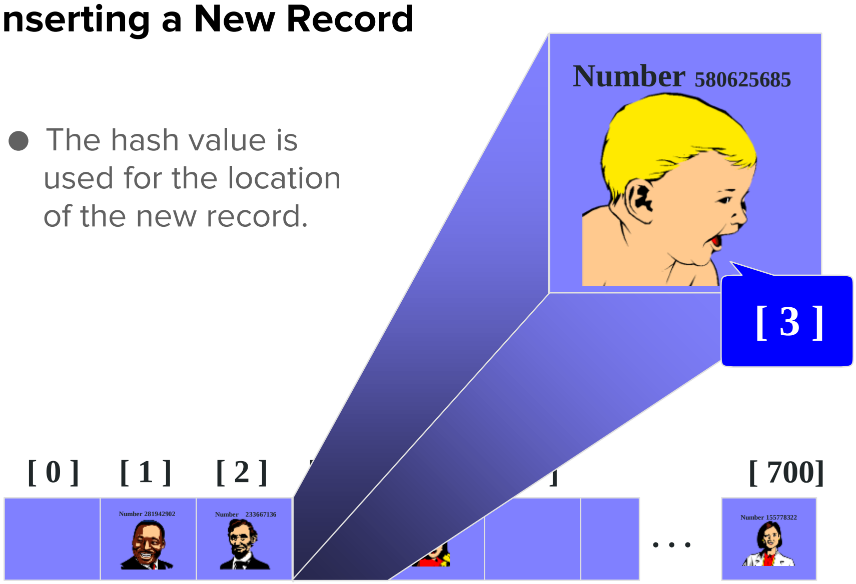




9

Inserting a New Record

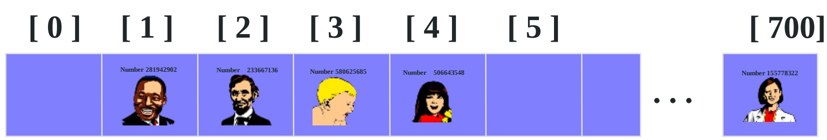
- The hash value is used for the location of the new record.



10

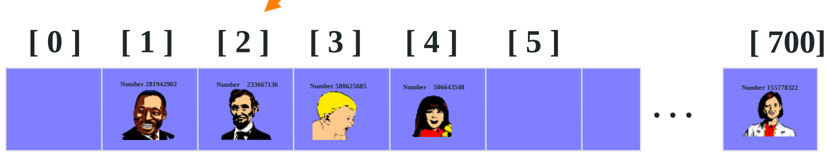
Inserting a New Record

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



Collisions

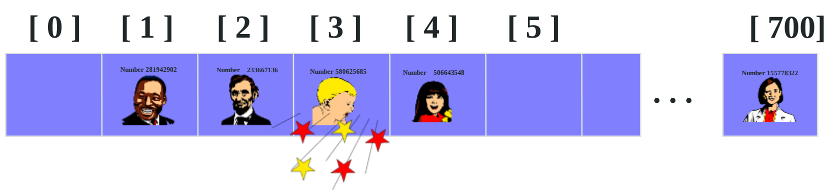
- Here is another new record to insert, with a hash value of 2.



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.

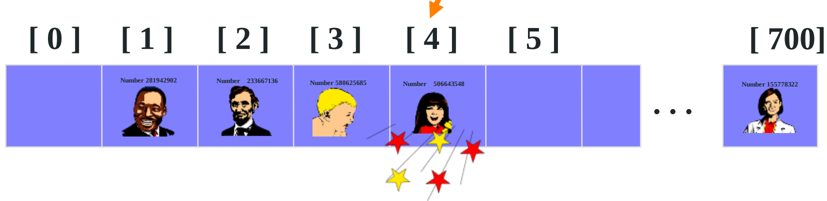


13

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.

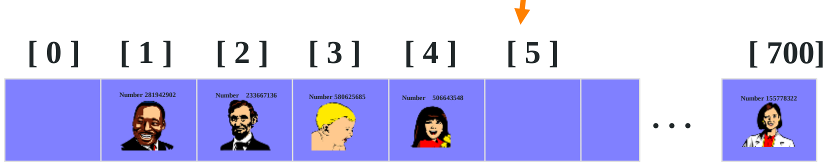


14

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.

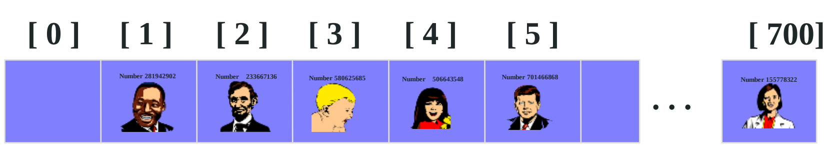


15

Collisions

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

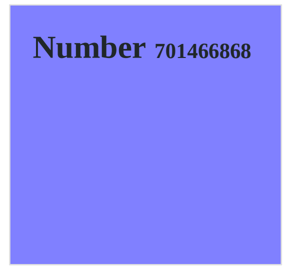
The new record goes in the empty spot.

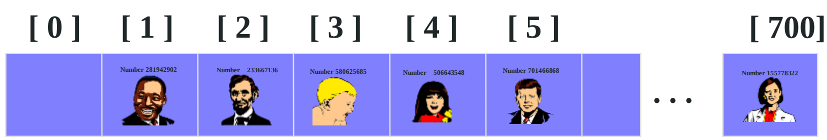


16

Searching for a Key

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

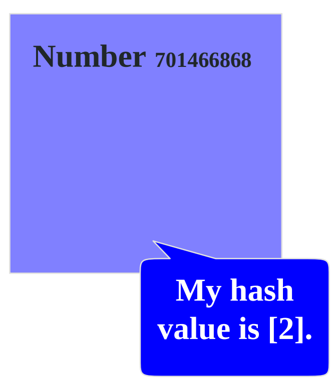


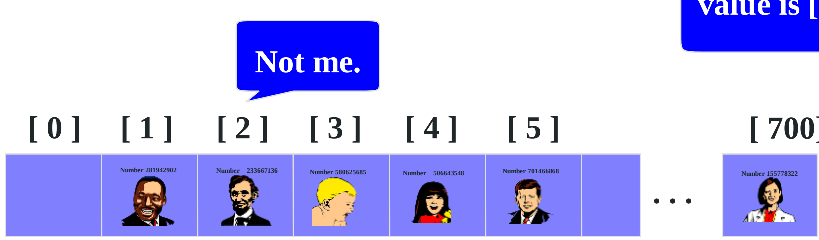


17

Searching for a Key

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

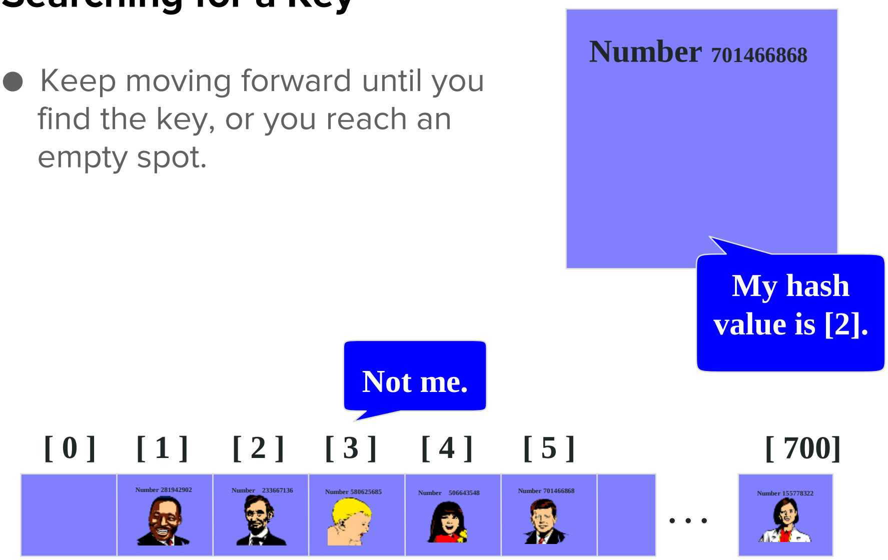




18

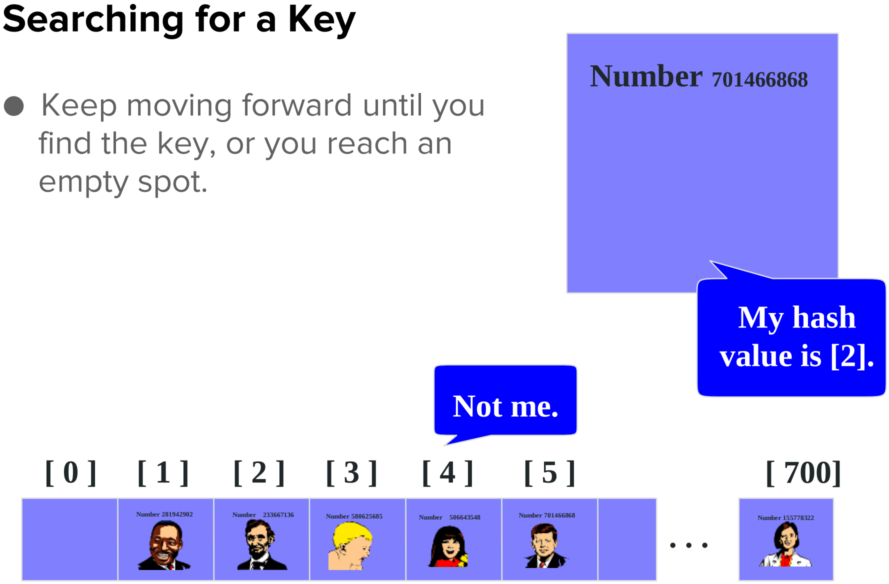
Searching for a Key

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



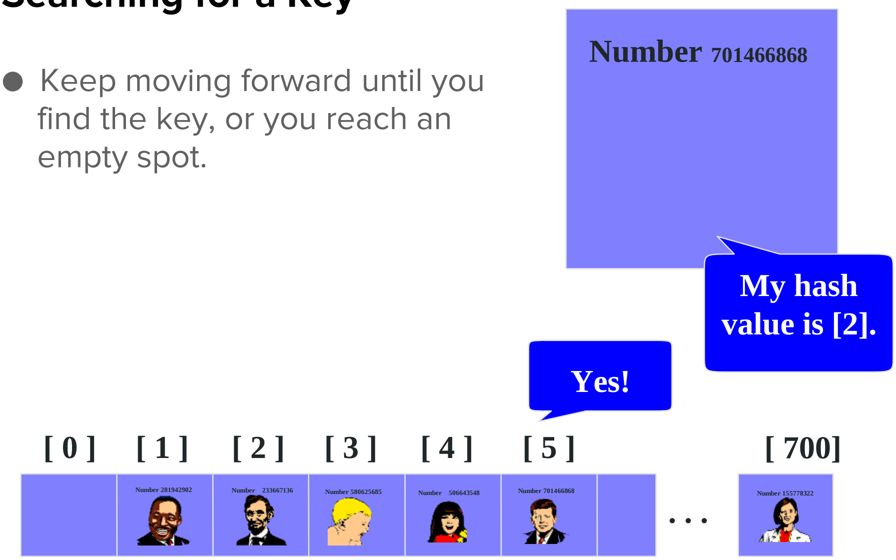
Searching for a Key

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



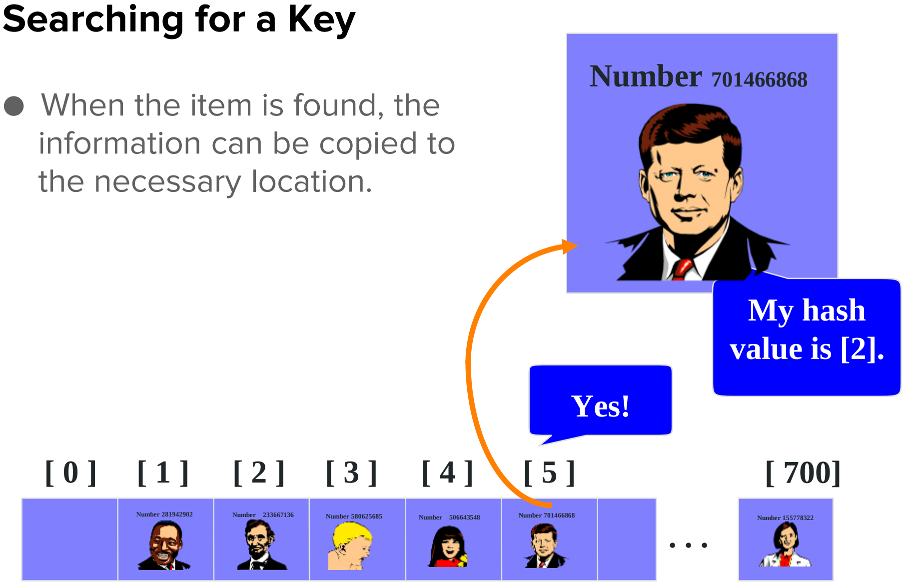
Searching for a Key

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



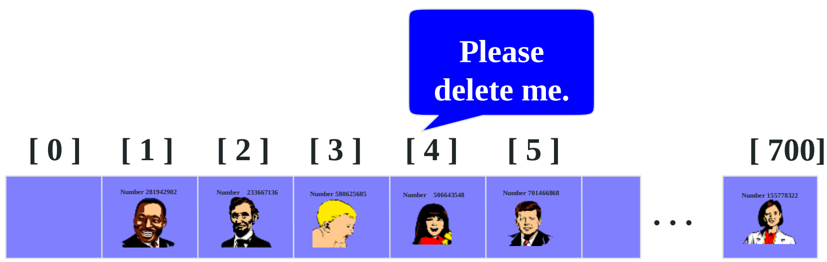
Searching for a Key

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



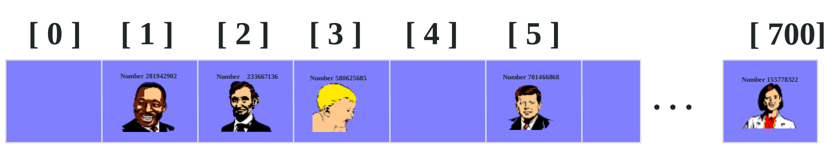
Deleting a Record

- Records may also be deleted from a hash table.



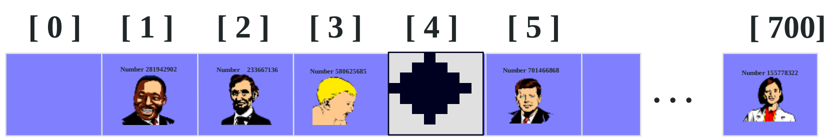
Deleting a Record

- 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.



25

Hashing

26

Hashing

- In cases where the keys are large and cannot be used directly as an index, you should use **hashing**.
- In hashing, large keys are converted into small keys by using **hash functions**.
- The values are then stored in a data structure called **hash table**.

27

Hashing

- The idea of hashing is to distribute entries (key/value pairs) **uniformly** across an array.
- By using that key you can access the element in **O(1) time**.
- Using the key, it is computed an index that suggests where an **entry can be found or inserted**.

28

Hashing

Hashing is implemented in two steps:

```
hash = hashfunc(key)
index = hash % array_size
```

In this method, the `hash` is independent of the array size and it is then reduced to an `index` (a number between 0 and array_size − 1) by using the modulo operator (%).

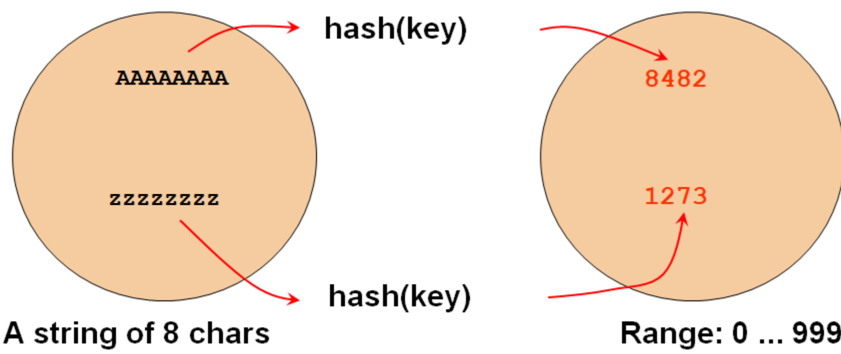
Hash function

A good hash function has the following basic requirements:

- **Easy to compute:** It should be easy to compute and must not become an algorithm in itself.
- **Uniform distribution:** It should provide a uniform distribution across the hash table and should not result in clustering.
- **Less collisions:** These should be avoided.

Hash function works something like ...

Convert a String key into an integer that will be in the range of 0 through the maximum capacity-1 *(Assume the array capacity is 9997)*



31

Integer keys

`h(x) = x % TableSize`

Good idea to make TableSize **prime**. Why?

32

Integer keys

`h(x) = x % TableSize`

Good idea to make TableSize **prime**. Why?

- Because keys are typically not randomly distributed, but usually have some *pattern*
  - mostly even
  - mostly multiples of 10
  - in general: mostly multiples of some *k*
- If k is a factor of TableSize, then only (TableSize/*k*) slots will ever be used!
- Since the only factor of a prime number is itself, this phenomena only hurts in the (rare) case where *k*=TableSize

Integer keys: why prime?

Suppose:

- data stored in hash table: `7160, 493, 60, 55, 321, 900, 810`
- tableSize = 10

  `data hashes to 0, 3, 0, 5, 1, 0, 0`
- tableSize = 11

  `data hashes to 10, 9, 5, 0, 2, 9, 7`

String as keys

If keys are strings, can get an integer by adding up ASCII values of characters in key

```
for (i=0; i<key.length(); i++)
  hashVal += key.charAt(i);
```

Problem 1: What if TableSize is 10,000 and all keys are 8 or less characters long?

Problem 2: What if keys often contain the same characters (“abc”, “bca”, etc.)?

35

More samples of hash functions

Let s be a string:

$$s = s_0\ s_1\ s_2\ \dots\ s_{k-1}$$

- h(s) = s_0 % TableSize
- h(s) = $\left(\sum_{i=0}^{k-1} s_i\right)$ % TableSize
- h(s) = $\left(\sum_{i=0}^{k-1} s_i \cdot 37^i\right)$ % TableSize

36

Collision resolution techniques

Load factor

The **load factor** of a hash table is:

$$\lambda = \mathbf{N / M}$$

where N is the number of elements and M is the table size.

Collision resolution

- A **collision** occurs when two different keys hash to the same value.
- Two different methods for collision resolution:
  - **Separate Chaining:** use a linked list to store multiple items that hash to the same slot.
  - **Closed Hashing (or probing):** search for empty slots using a second function and store item in first empty slot that is found.

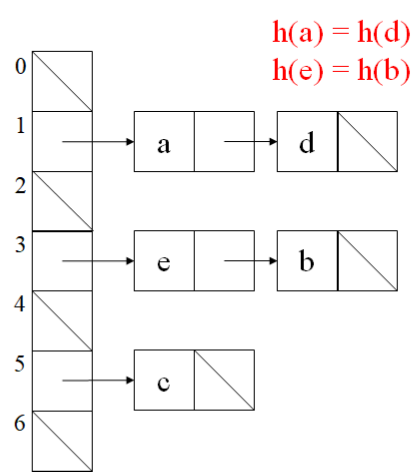
Separate chaining = Open hashing

Closed hashing = Open addressing

39

Separate chaining

- An unordered linked list (chain) is associated with each entry.
- Properties:
  - Performance degrades with length of chains.
  - **λ can be greater than 1**



40

Load factor in Separate Chaining

For separate chaining,

λ = average # of elements in a list

- Unsuccessful search:
  Traverse the whole list, on average λ

- Successful search:
  Traverse the half of list, on average $\lambda/2 + 1$

41

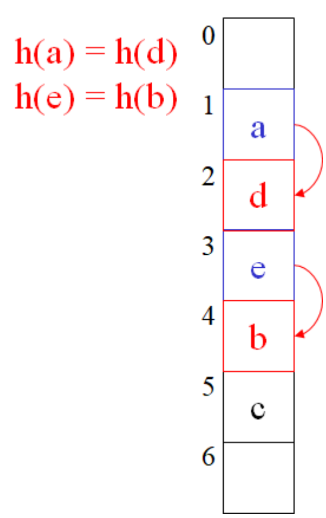
Closed hashing

What if we only allow one key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must go in another spot

Properties

- **λ <= 1**
- performance degrades with difficulty of finding right spot



42

Closed hashing

Given an item X, try cells $h_0(X)$, $h_1(X)$, $h_2(X)$, …, $h_i(X)$

$$h_i(X) = (\mathrm{Hash}(X) + F(i)) \,\%\, \mathrm{TableSize}$$

- Define F(0) = 0
- F is the collision resolution function. Some possibilities:
  - Linear: $F(i) = i$
  - Quadratic: $F(i) = i^2$
  - Double Hashing: $F(i) = i * \mathrm{Hash}_2(X)$

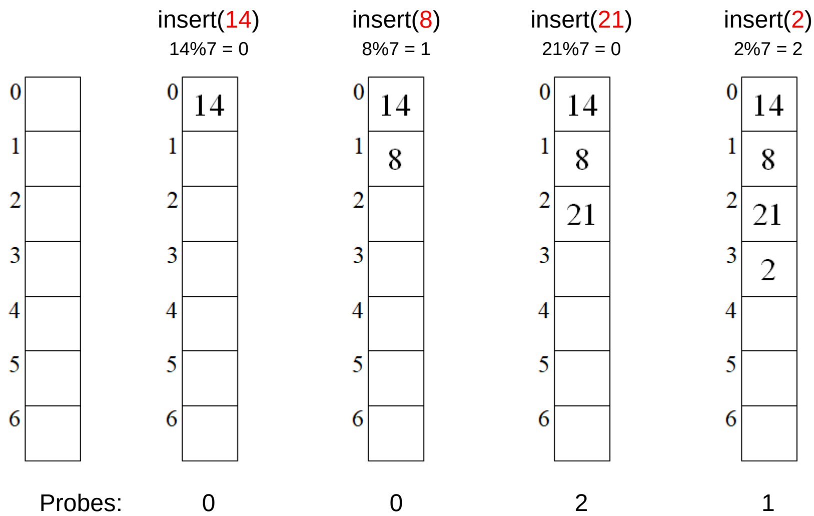
Closed hashing: Linear Probing

Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell

$$h_i(X) = (\mathrm{Hash}(X) + i) \,\%\, \mathrm{TableSize} \qquad (i=0,1,2,\ldots)$$

Compute hash value and increment it until a free cell is found.

Closed hashing: Linear Probing



45

Closed hashing: Linear Probing

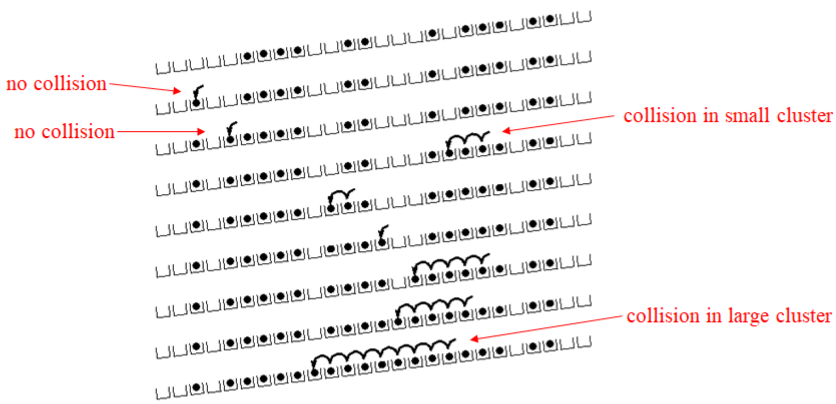
Drawbacks

- Works until array is full, but as number of items N approaches TableSize ($\lambda \approx 1$),
- Very prone to **cluster formation**
  - If a key hashes anywhere into a cluster, finding a free cell involves going through the entire cluster.
  - **Primary clustering** – clusters grow when keys hash to values close to each other
- Does not satisfy good hash function criterion of distributing keys uniformly, because can have cases where table is empty except for a few clusters.

46

Closed hashing: Linear Probing

Clustering



Load factor in Linear Probing

- For any $\lambda < 1$, linear probing will find an empty slot
- Expected # of probes (for large table sizes)
  - Successful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

  - Unsuccessful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- Performance quickly degrades for $\lambda > 1/2$

Closed hashing: Quadratic Probing

Main Idea: Spread out the search for an empty slot – Increment by i^2 instead of i

$$h_i(X) = (Hash(X) + i^2) \ \% \ TableSize \qquad (i=0,1,2,\ldots)$$

$h_0(X) = Hash(X) \ \% \ TableSize$

$h_1(X) = (Hash(X) + 1) \ \% \ TableSize$

$h_2(X) = (Hash(X) + 4) \ \% \ TableSize$

$h_3(X) = (Hash(X) + 9) \ \% \ TableSize$

Closed hashing: Quadratic Probing

		insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 = 0	insert(2) 2%7 = 2
0		14	14	14	14
1			8	8	8
2					2
3					
4				21	21
5					
6					
Probes:		0	0	2	0

Closed hashing: Quadratic Probing

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 = 0	insert(2) 2%7 = 2	insert(7) 7%7 = 0
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
Probes:	0	0	2	0	??

Load factor in Quadratic Probing

- If TableSize is prime and λ <= 1/2, quadratic probing will find an empty slot; for greater λ, might not.
- Quadratic probing does not suffer from primary clustering (keys hashing to the same area).
- Quadratic probing still get clustering from **identical keys** (*secondary clustering*)

Closed hashing: Double Hashing

Main Idea: Spread out the search for an empty slot by using a second hash function.

$$h_i(X) = (Hash_1(X) + i*Hash_2(X)) \% TableSize$$
$$(i=0,1,2,\dots)$$

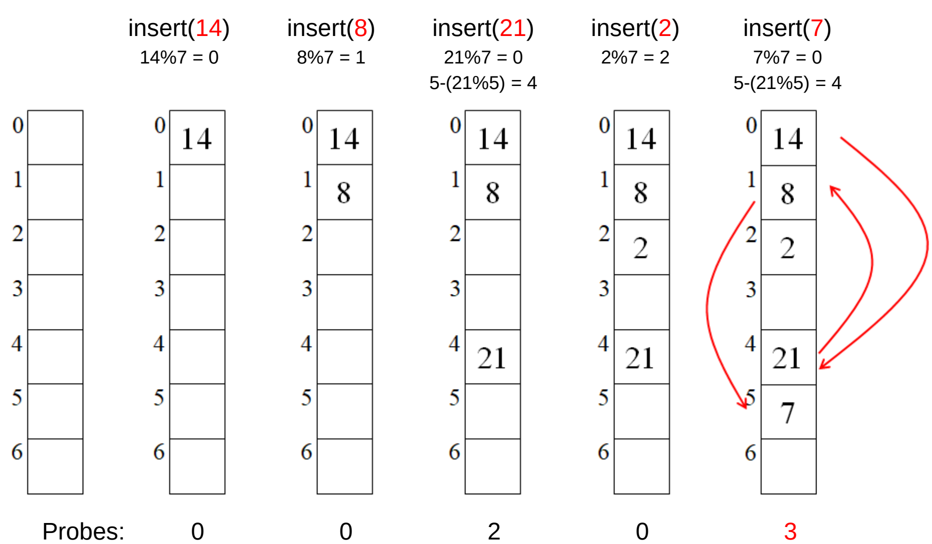
Good choice of $Hash_2(X)$ can guarantee does not get "stuck" as long as $\lambda < 1$

- Integer keys:

  $Hash_2(X) = R - (X \% R)$

  where R is a prime smaller than TableSize

Closed hashing: Double Hashing



Load factor in Double Hashing

- For any $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and $hash_2$).
- Search cost approaches optimal (random re-hash):
  - Successful search:

$$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

  - Unsuccessful search:

$$\frac{1}{1-\lambda}$$

- No primary clustering and no secondary clustering.
- Still becomes costly as λ nears 1.

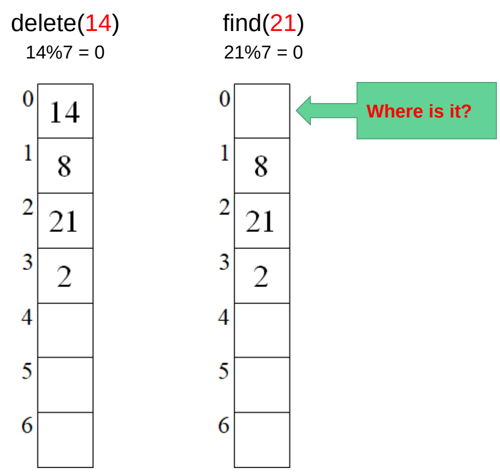
55

Deletion in Separate Chaining

- Use the index to access to the linked list and evaluate the elements to find and remove the corresponding element.

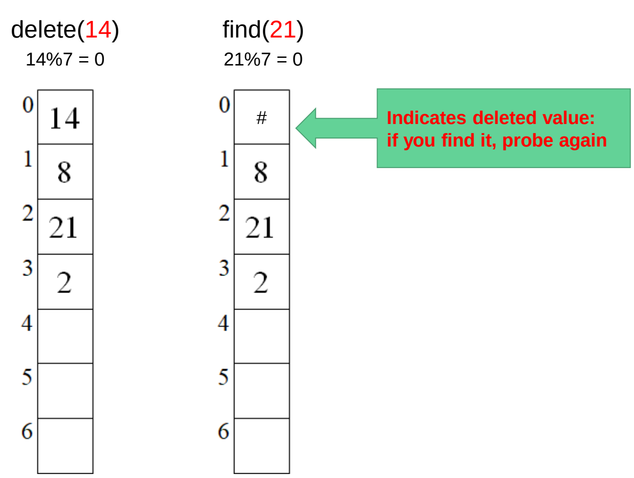
56

Deletion in Closed Hashing



57

Lazy Deletion



58

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.
- 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.