Chapter 8
Searching and Hashing

Contents

Searching

- A **table** or a **file** is a group of elements, each of which is called a record
- A **key** is used to differentiate among different records
- **Searching** is the process of finding a record (with the target key) among a list of records
- **Searching** is one of the most common and time-consuming operations
- A **search algorithm** may return the entire record or, more commonly, it may return a pointer to that record
- If the record is not found, then it is called an **unsuccessful search**
- A **successful search** is often called a retrieval

Basic searching techniques

The algorithm used to search a list depends to a large extent on the structure of the list.

Two basic searches for arrays are:

1. Sequential search
2. Binary search

Sequential search (aka linear search)

- Is used in an **unordered** list

Steps:

1. Start from the leftmost element of the list and one by one compare the target with each element of the list
2. If the target matches with an element, return the index of the element
3. Otherwise, return -1 indicating that the target is not present in the list

Sequential search

Example: Search for 1 in this unsorted list.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Input:	26	5	37	1	61	11	59	15	48	19

Target: 1

Sequential search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19

input[0] == 1 ?

Index: 0

Sequential search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19

input[1] == 1 ?

Index: 1

Sequential search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19

input[2] == 1 ?

Index: 2

Sequential search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19

input[3] == 1 ? Yes

Index: 3

Sequential search performance

Best case, i.e. when the target is the first element in the list:

O(1)

Worst case, i.e. when the target is not present in the list or is the last element of the list:

O(n)

Average case:

O(n)

Binary search

In sequential search, if there are 1000 elements, 1000 comparisons will be made in the worst case.

If the list is sorted, we can use a more efficient algorithm called the **binary search**.

In general, we should use a binary search whenever the list starts to become large (e.g., when the list has more than 16 elements).

Algorithm: binarySearch(a, target)

Input: A sorted list, a, and the element to be searched, target

Output: Index of the target, if present, otherwise -1

Steps:

```
1.  min = 0
2.  max = n - 1
3.  while max ≥ min
4.      mid = ⌊(min + max ) / 2⌋ # average of max and min
5.      if a[mid] == target
6.          return mid  # target found
7.      else if a[mid] < target
8.          min = mid + 1
9.      else
10.         max = mid - 1
11.     end if
12. end while
13. if max < min, then return -1 # target is not present
14. end if
```

Binary search

Example: Search for 26 in this list.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Input:	1	5	11	15	19	26	37	48	59	61

Target: 26

Binary search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	5	11	15	19	26	37	48	59	61

min	**max**	**mid**
0	9	4

mid $\boxed{4}$

input[4] == 26 ? No
input[4] > 26 ? No
input[4] < 26 ? **Yes**

Binary search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	5	11	15	19	26	37	48	59	61

min	max	mid
5	9	7

mid 7

input[7] == 26 ? No
input[7] > 26 ? **Yes**

Binary search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	5	11	15	19	26	37	48	59	61

min	max	mid
5	6	5

mid $\boxed{5}$ input[5] == 26 ? **Yes**

Target found!

Binary search performance

Best case: $O(1)$

Worst case: $O(\log_2 n)$

Average case: $O(\log_2 n)$

Hashing

Goal of hashing: To find the data with only one test, i.e., expected complexity = O(1). (Also, to insert and delete in O(1) expected time.)

In a **hashed search**, the key determines the location of the data through an algorithmic function called a **hash function**

Main idea:

1. Use a hash function to determine where to insert the record
2. When a record needs to be searched, use the same hash function to locate the record

Hashing

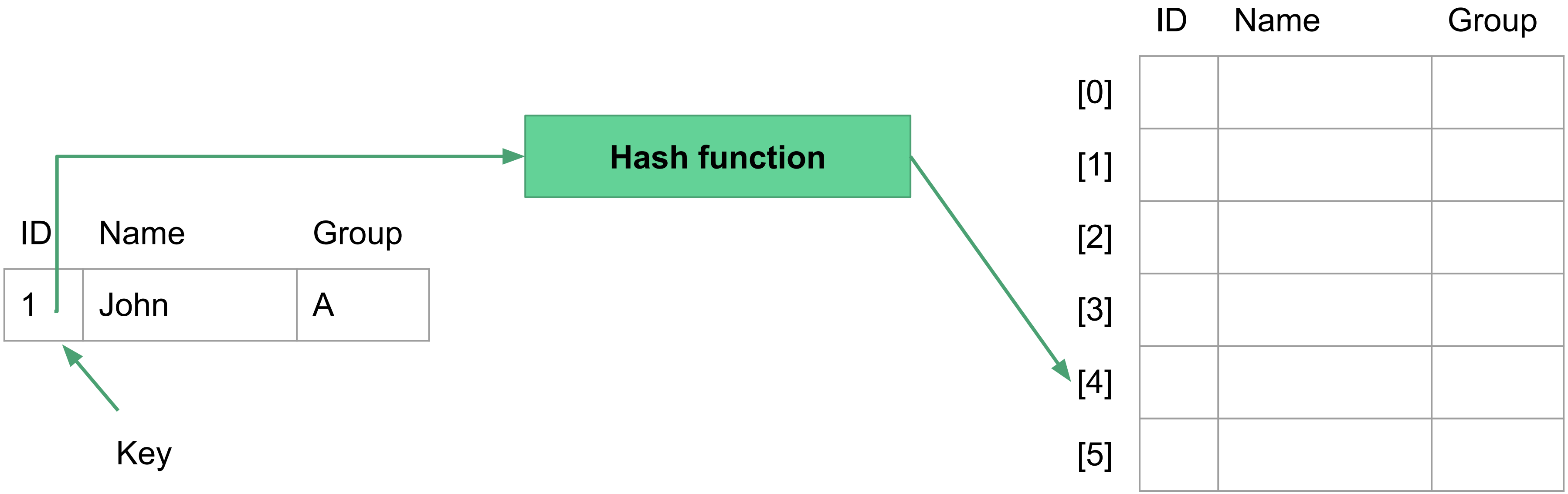
Hashing is a **key-to-address mapping** process

For an array, the address can be the index that contains the data

A **hash function** is a function which when given a key, generates an address in the table

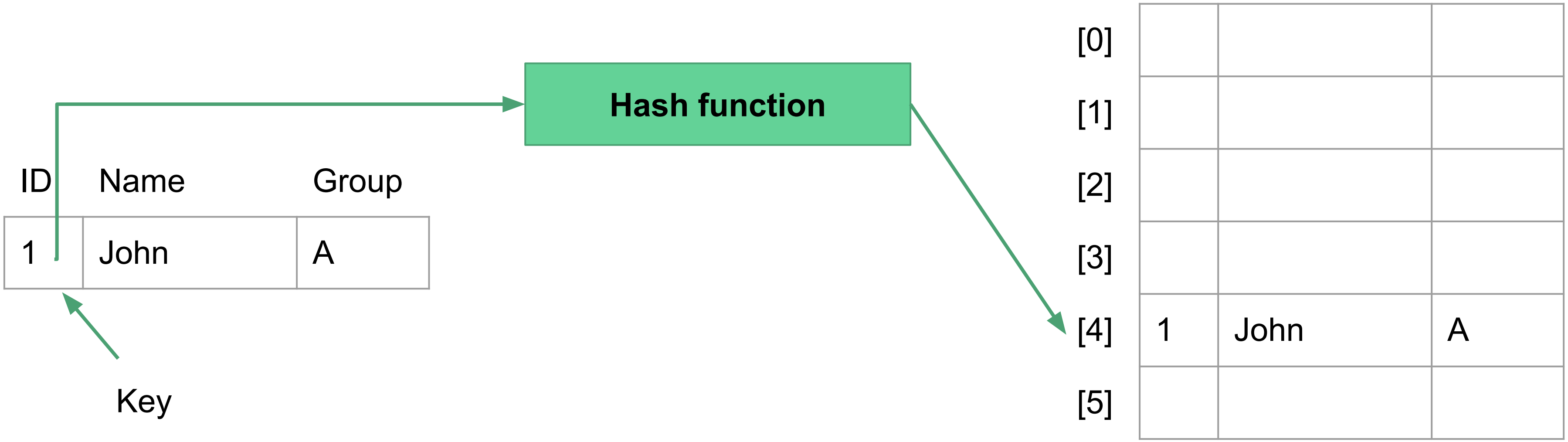
Hashing

Use a hash function to determine where to insert the record

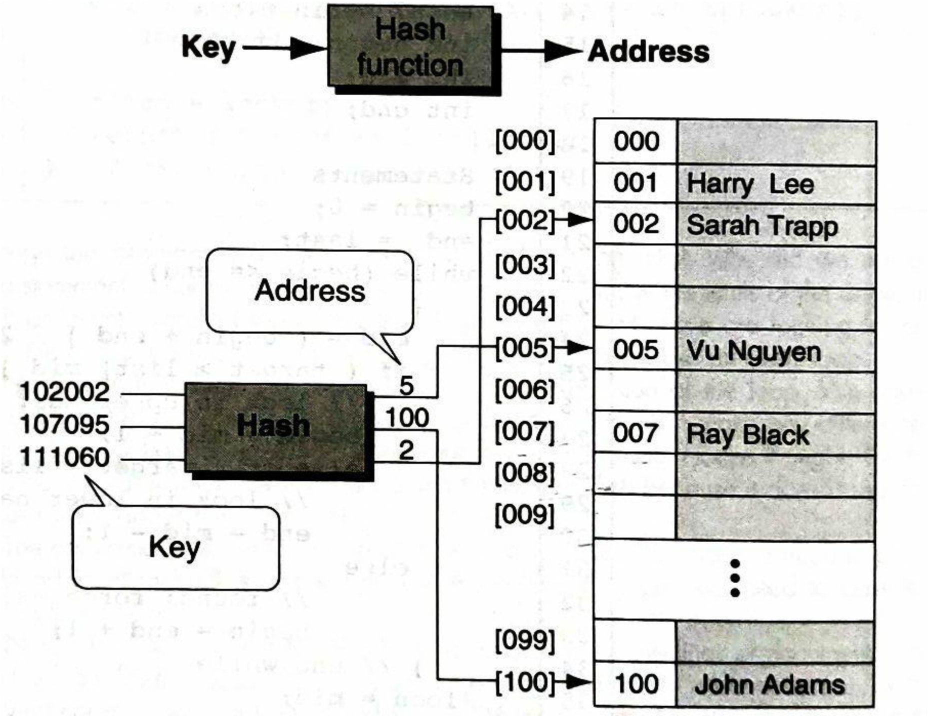


Hashing

When a record needs to be searched, use the same hash function to locate the record



Hashing



Key
Hash function
Address
[000]
000
[001]
001
Harry Lee
[002]
002
Sarah Trapp
[003]
[004]
[005]
005
Vu Nguyen
[006]
[007]
007
Ray Black
[008]
[009]
[099]
[100]
100
John Adams
Address
5
100
2
102002
107095
111060
Hash
Key

Hash function efficiency

Measure of how efficiently the hash function produces hash values for elements within a set of data.

A hash function should be a quick, stable and deterministic operation.

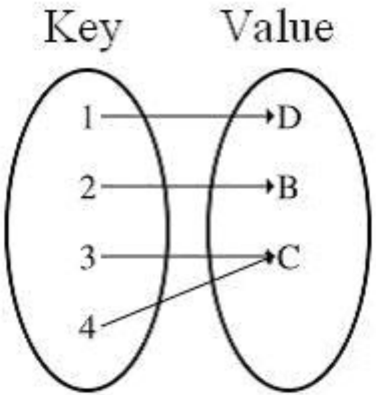
Hash table

A data structure for quickly looking things up

Implements an **associative array** (**map, symbol table**, or **dictionary**) abstract data type

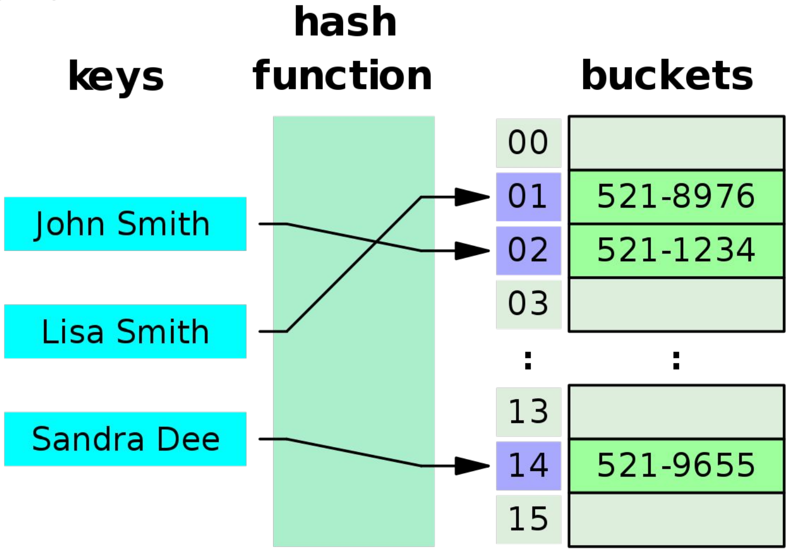
A **dictionary** is a structure that is composed of a collection of **(key, value) pairs** (maps keys to values)

The main operation supported by a dictionary is **searching by key**



Hash table

A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found



Hashing terminologies

Home address: The address produced by the hashing algorithm

Prime area: The memory that contains all of the home addresses

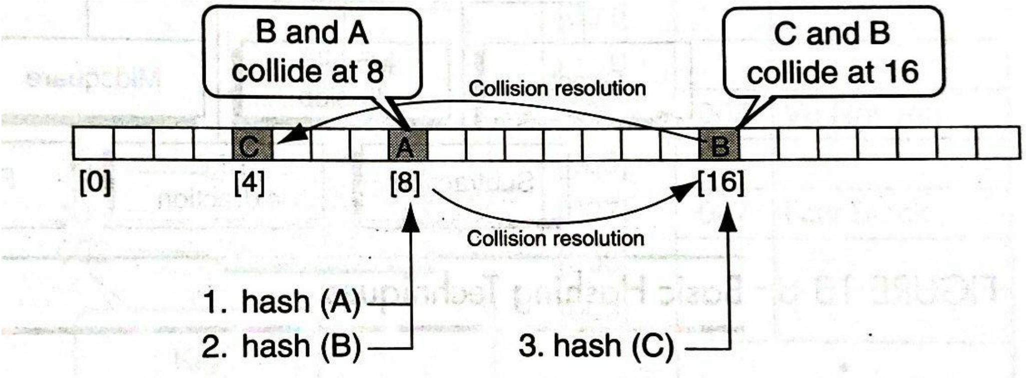
Synonyms: The set of keys that hash to the same location

If the data contain two or more synonyms, we can have **collisions**. A **collision** occurs when a hashing algorithm produces an address for an insertion key and that address is already occupied.

Hashing terminologies

Collision resolution

When two keys collide at a home address, we must resolve the collision by placing one of the keys and its data in another location.



Hashing methods

- Direct hashing
- Subtraction
- Modulo-division / division remainder
- Digit-extraction
- Midsquare hashing
- Folding
- Rotation hashing
- Pseudorandom hashing

Hashing methods

Subtraction method

Subtract a fixed value from the key to determine the address

Address = key - constant

Limitation:

Hashing methods

Modulo-division method

Divide the key by the array size and use the remainder for the address

Address = key % listSize

Example:

Hashing methods

Digit-extraction

Extract selected digits from the key and use them as the address

Example:

37**94**52 → 394

12**12**67 → 112

37**88**45 → 388

Hashing methods

Midsquare hashing

Square the key and select the address from the middle of the squared number

Example:

If key = 9452, the address can be taken as 3403 because $9452^2 = 89\mathbf{3403}04$

Hashing methods

Pseudorandom hashing

The key is used as the seed in a **pseudorandom number generator (PRNG)***, and the resulting random number is then scaled into the possible address range using modulo-division method

* The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's seed. Example PRNG: $y = ax + c$

Collision resolution

A **perfect hash function** will assign each key to a unique bucket (i.e., no collision)

Most hash table designs employ an **imperfect hash function**, which might generate the same index for more than one key, causing collisions

When two keys collide at a home address, we must resolve the collision by placing one of the keys and its data in another location.

Two general approaches to handling collision resolution:

1. Open addressing: resolves collisions in the prime area
2. Chaining: resolves collisions by placing the data in a separate overflow area

Collision resolution

Open addressing

When a collision occurs, the prime area addresses are searched for an open or unoccupied element where the new data can be placed. Each calculation of an address and test for success is called a **probe**.

- Linear probing
- Quadratic probing
- Double hashing (rehashing)
- Random probing

Collision resolution

Linear probing

When inserting a new pair whose key is k, we search the hash table addresses in the order $(\ h(k) + i\ )\ \%\ b,\ 0 \le i \le b - 1$, where h is the hash function, and b is the size of the hash table (or the array).

The search terminates when we find the first unfilled address.

Example: Using modulo-division method and linear probing, store the keys shown below in an array with 19 elements.

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Linear probing

Example: Using modulo-division method and linear probing, store the keys shown below in an array with 19 elements.

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Solution

Here, b = 19 and the hash function is h(k) = k % n

Address for 224562 is h(224562) = 224562 % 19 = 1

[0]	
[1]	224562
[2]	
[3]	
[4]	
[5]	
[6]	
[9]	
[10]	
[11]	
[12]	
[13]	
[14]	
[15]	
[16]	
[17]	
[18]	

Linear probing

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Solution (Contd.)

Address for 137456 is h(137456) = 137456 % 19 = 10

Similarly, h(214562) = 214562 % 19 = 14

[0]	
[1]	224562
[2]	
[3]	
[4]	
[5]	
[6]	

[9]	
[10]	137456
[11]	
[12]	
[13]	
[14]	214562
[15]	
[16]	
[17]	
[18]	

Linear probing

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Solution (Contd.)

h(140145) = 140145 % 19 = 1. Since the index 1 is already occupied, we probe sequentially until we find an unoccupied index.

The next address is (h(140145) + 1) % b = (1 + 1) % 19 = 2, which is unoccupied. So, 140145 will be inserted at the index 2 of the array.

[0]		[9]	
[1]	224562	[10]	137456
[2]	140145	[11]	
[3]		[12]	
[4]		[13]	
[5]		[14]	214562
[6]		[15]	
		[16]	
		[17]	
		[18]	

Linear probing

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Solution (Contd.)

h(214576) = 214576 % 19 = 9

[0]	
[1]	224562
[2]	140145
[3]	
[4]	
[5]	
[6]	
[9]	214576
[10]	137456
[11]	
[12]	
[13]	
[14]	214562
[15]	
[16]	
[17]	
[18]	

Linear probing

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Solution (Contd.)

h(162145) = 162145 % 19 = 18

[0]	
[1]	224562
[2]	140145
[3]	
[4]	
[5]	
[6]	

[9]	214576
[10]	137456
[11]	
[12]	
[13]	
[14]	214562
[15]	
[16]	
[17]	
[18]	162145

Linear probing

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Solution (Contd.)

h(144467) = 144467 % 19 = 10. Collision occurs here.

The next address is (h(144467) + 1) % b = (10 + 1) % 19 = 11, which is unoccupied. So, 144467 will be inserted at the index 11 of the array.

[0]	
[1]	224562
[2]	140145
[3]	
[4]	
[5]	
[6]	

[9]	214576
[10]	137456
[11]	144467
[12]	
[13]	
[14]	214562
[15]	
[16]	
[17]	
[18]	162145

Linear probing

224562, 137456, 214562, 140145, 214576, 162145, 144467, 199645, 234534

Solution (Contd.)

h(199645) = 199645 % 19 = 12

h(234534) = 234534 % 19 = 17

[0]	
[1]	224562
[2]	140145
[3]	
[4]	
[5]	
[6]	

[9]	214576
[10]	137456
[11]	144467
[12]	199645
[13]	
[14]	214562
[15]	
[16]	
[17]	234534
[18]	162145

Linear probing

Summary

Hash. Map key to integer i between 0 and N-1, where N is the array size.

Insert. Put at table index i if free; if not try i+1, i+2, etc.

Search. Search table index i; if occupied but no match, try i+1, i+2, etc.

Note. Array size N must be greater than number of key-value pairs.

Linear probing

Advantages:

- Simple to implement
- Data tend to reamin near their home address

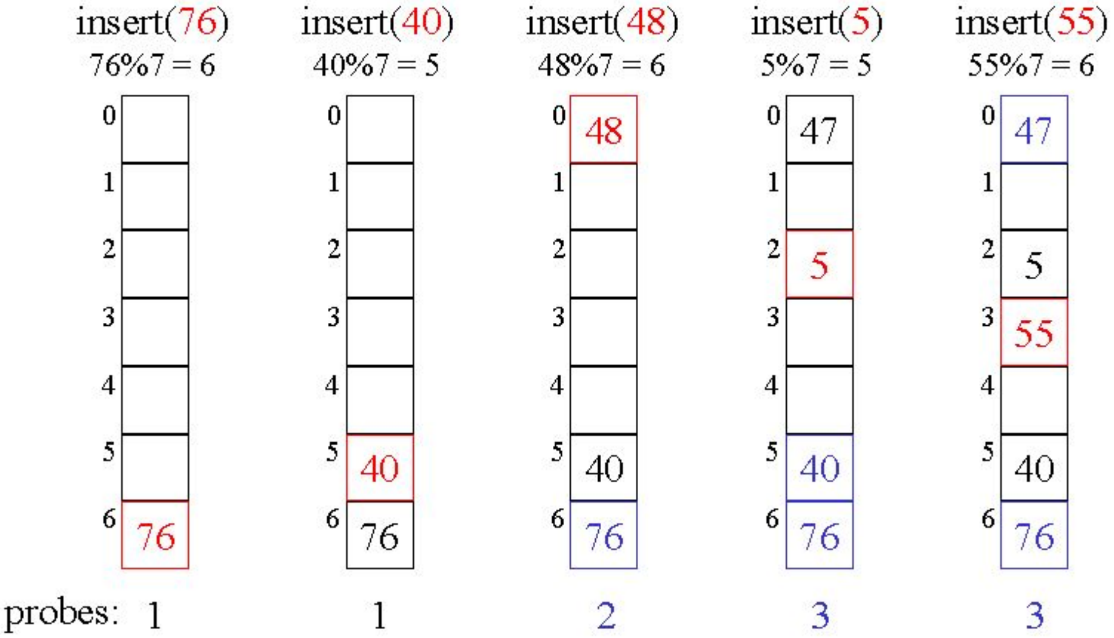
Disadvantages:

- Linear probes tend to produce **primary clustering** (clustering of data aroudn a home address)
- Tend to make the search algorithm more complex, especially after data have been deleted

Quadratic probing

A quadratic function of i is used as the increment, i.e., we examine the addresses $(h(k) + i^2) \% b$

Example:



Quadratic probing

Limitation:

It is not possible to generate a new address for every element in the list.

Solution:

Use a list size that is a prime number. In this case, at least half of the list is reachable.

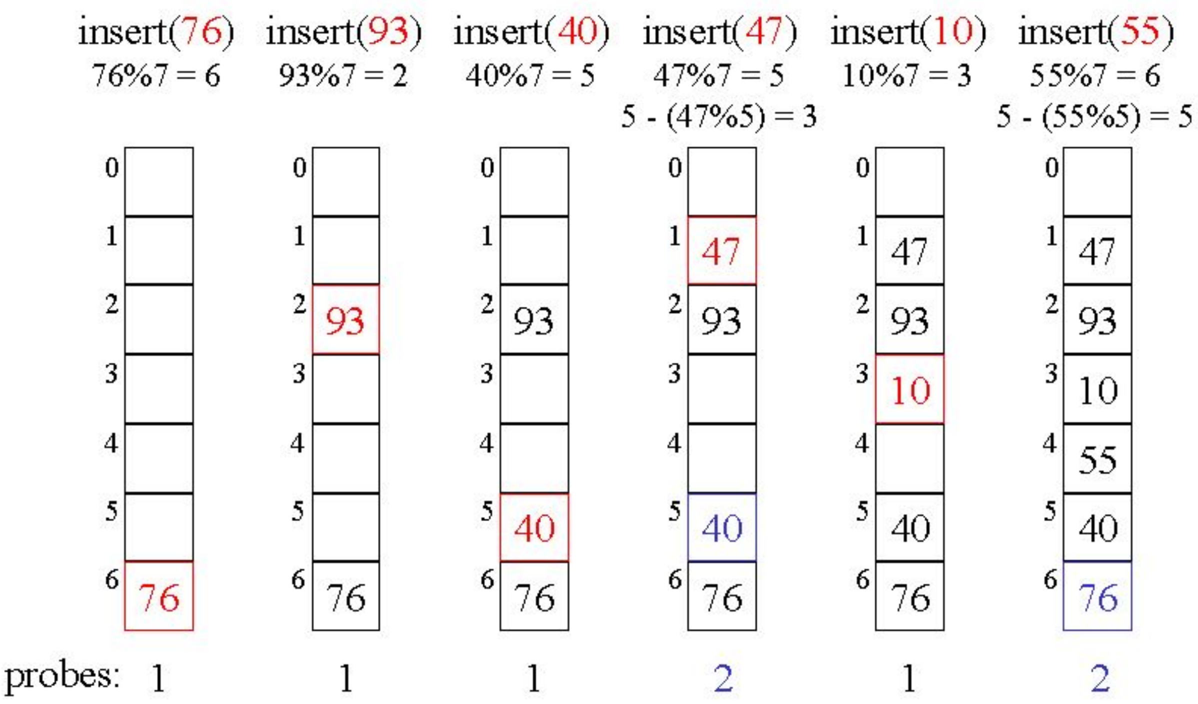
Double hashing

Rehashing: Use a series of hash functions h_1, h_2, ... , h_n.

Double hashing: Use two hash functions, h_1, and h_2.

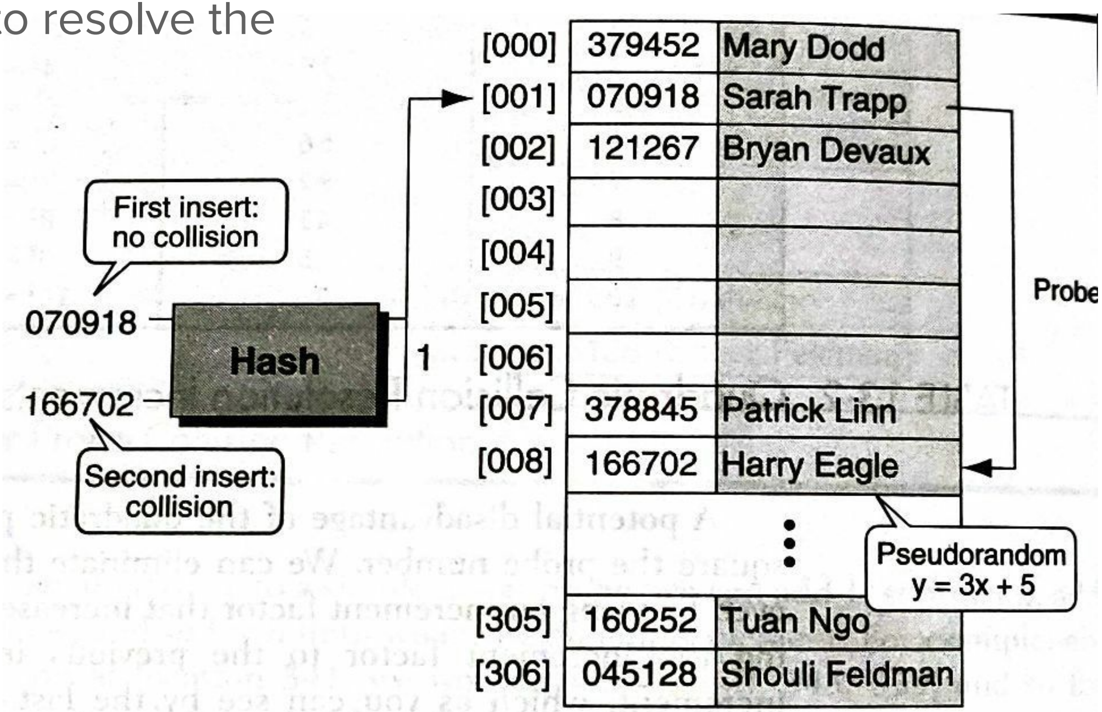
- First probe the location h_1(key) % N, where N is the array size.
- If the location is occupied, we probe the location (h_1(key) + h_2(key)) % N, then (h_1(key) + 2*h_2(key)) % N, and so on.

Double hashing example



(Pseudo)Random probing

Uses a pseudorandom number to resolve the collision



Chaining

A major disadvantage to open addressing is that each collision resolution increases the probability of future collisions. Also, the search for a key involves comparison with keys that have different hash values.

This disadvantage is eliminated in chaining

- Uses a separate area to store collisions and chains all synonyms together in a linked list
- Uses two storage areas: the prime area and the overflow area
- Each element in the prime area contains a link head pointer to a linked list of overflow data in the overflow area

Chaining

When a collision occurs, one element is stored in the prime area and chained to its corresponding linked list in the overflow area