Hash Tables

Instructor: Samreen Ishfaq

Hash Tables: Intuition

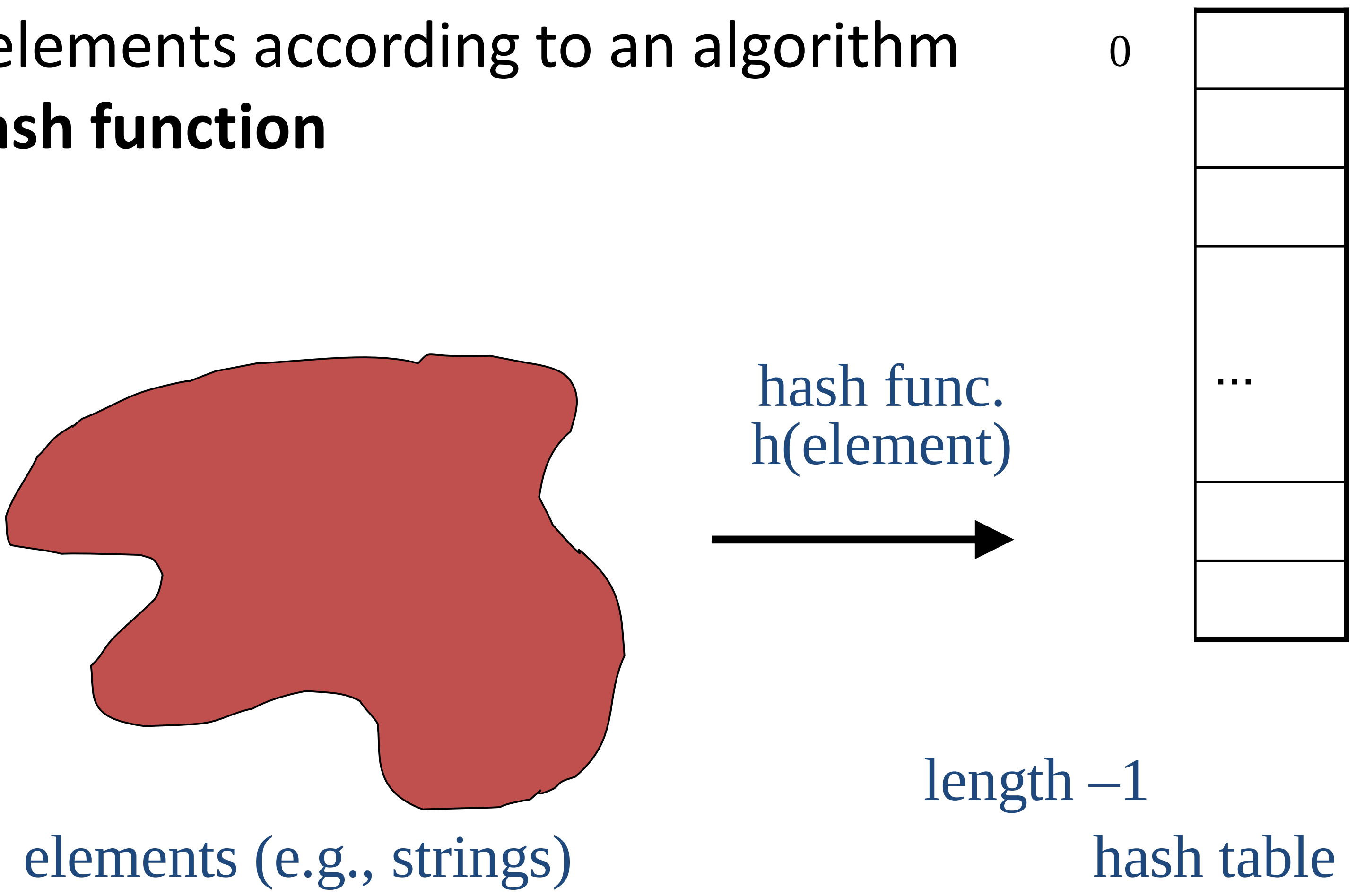
- **Hash table** or **hash map** is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number).
- Thus, a hash table implements an associative array.
- The hash function is used to transform the key into the index (the h*ash*) of an array element (the *slot* or *bucket*) where the corresponding value is to be sought.

- Hashing is function that maps each key to a location in memory.
- A key’s location does not depend on other elements, and does not change after insertion.
    - unlike a sorted list
- A good hash function should be easy to compute.

- With such a hash function, the dictionary operations can be implemented in O(1) time.

Hash tables

- **hash table**: an array of some fixed size, that positions elements according to an algorithm called a **hash function**



Hash function example

- Hash function= element % HtableSize
- *h(i) = i % 10*
- add 41, 34, 7, and 18
- constant-time lookup:
  - just look at *i % 10* again later

18 % 10= 8

34 % 10= 4 FOUND ☺

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Writing a hash function

- If we write a hash table that can store objects, we need a hash function for the objects, so that we know what index to store them

We want a hash function to:

1. be simple/fast to compute
2. map equal elements to the same index
3. map different elements to different indexes
4. have keys distributed evenly among indexes

Hashing : the basic idea

- Map key values to hash table addresses

  keys -> hash table address

  This applies to find, insert, and remove
- Usually: *integers* **->** {0, 1, 2, …, *Hsize*-1}

  Typical example: $f(n) = n \bmod Hsize$

- Non-numeric keys converted to numbers
  - *For example, strings converted to numbers as*
    - Sum of ASCII values
    - First three characters

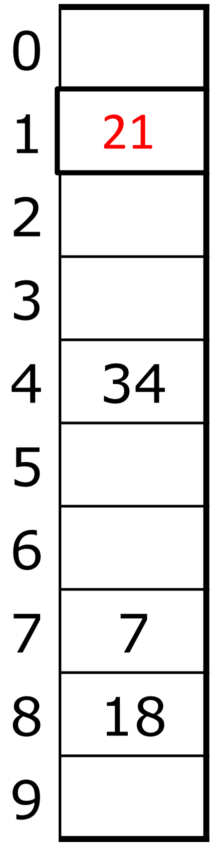
Hashing:

- *Choose a hash function h; it also determines the hash table size.*
- *Store x at location k by finding x%hsize and storing it h(k)*
- *To find if x is in the set, check location h(k).*
- *What to do if more than one keys hash to the same value. This is called collision.*
- *We will discuss two methods to handle collision:*
    - Separate chaining
    - Open addressing

Hash collisions

- **collision**: the event that two hash table elements map into the same slot in the array

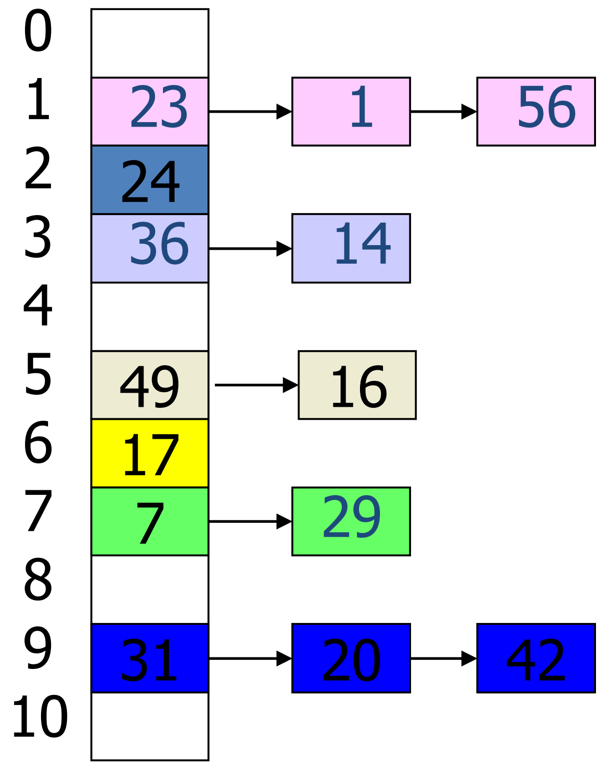
- example: add 41, 34, 7, 18, then 21
  - 21 hashes into the same slot as 41!
  - 21 should not replace 41 in the hash table; they should both be there



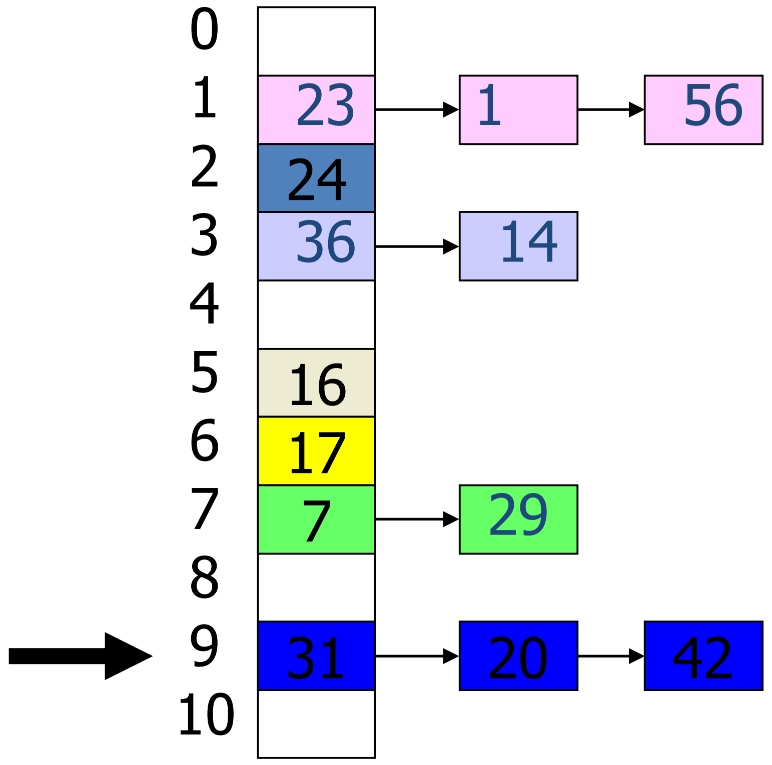
collision resolution: means for fixing collisions in a hash table

Separate chaining

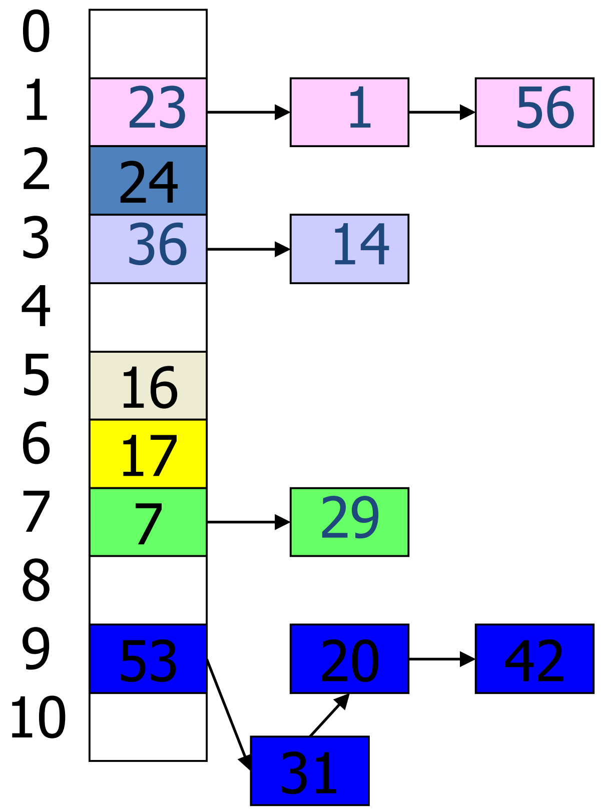
- Maintain a list of all elements that hash to the same value
- Search -- using the hash function to determine which list to traverse
- Insert/deletion–once the “bucket” is found through *Hash*, insert and delete are list operations



Insertion: insert 53



53 = 4 x 11 + 9
53 mod 11 = 9



Analysis of Hashing with Chaining

- Worst case
  - All keys hash into the same bucket
  - a single linked list.
  - insert, delete, find take O(n) time.

Hash Tables

Open Addressing
Part 2

Instructor: Samreen Ishfaq

Open addressing

- If collision happens, alternative cells are tried until an empty cell is found.

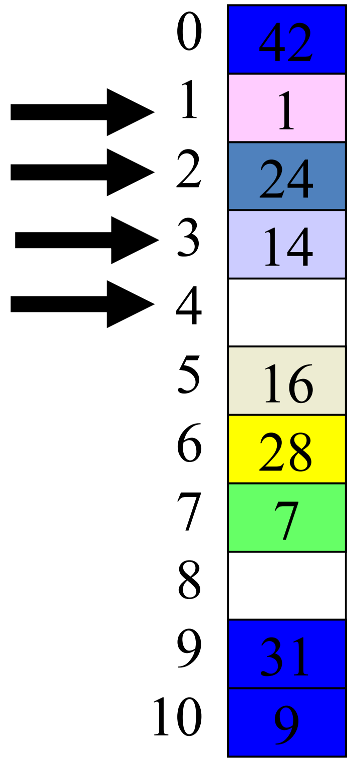
- Linear probing :
  Try next available position

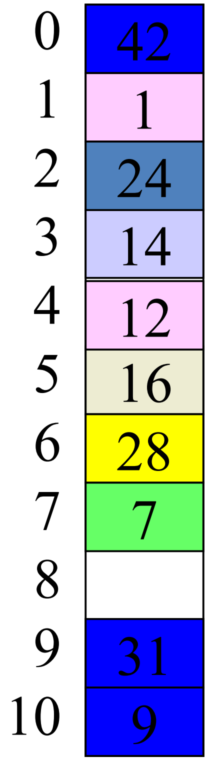
0	42
1	1
2	24
3	14
4	
5	16
6	28
7	7
8	
9	31
10	9

Linear Probing (insert 12)

H(x) -> Hash function
i -> total number of collision

12 = 1 x 11 + 1
12 mod 11 = 1





Linear probing H(x) + i

1 + 1 = 2

1 + 2 = 3

1 + 3 = 4

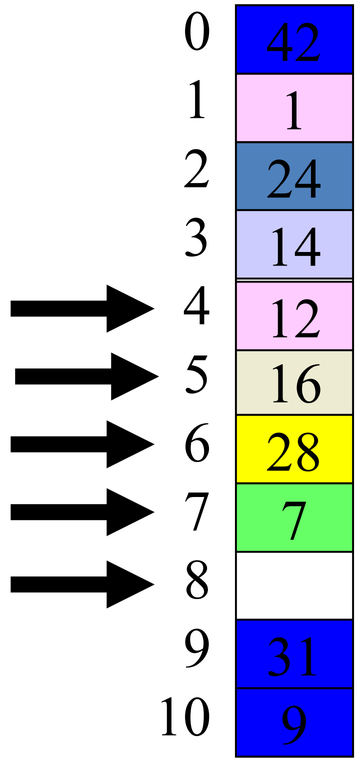
Empty slot data will be inserted at 4th index

Searching in Linear probing

Search an element on the hash function value
If value found you are done
Else
Keep searching it on next location
Terminate search when found an empty location

Search with linear probing (Search 15)

15 = 1 x 11 + 4
15 mod 11 = 4



NOT FOUND !

Linear probing H(x) + i

4 + 1 = 5

4 + 2 = 6

4 + 3 = 7

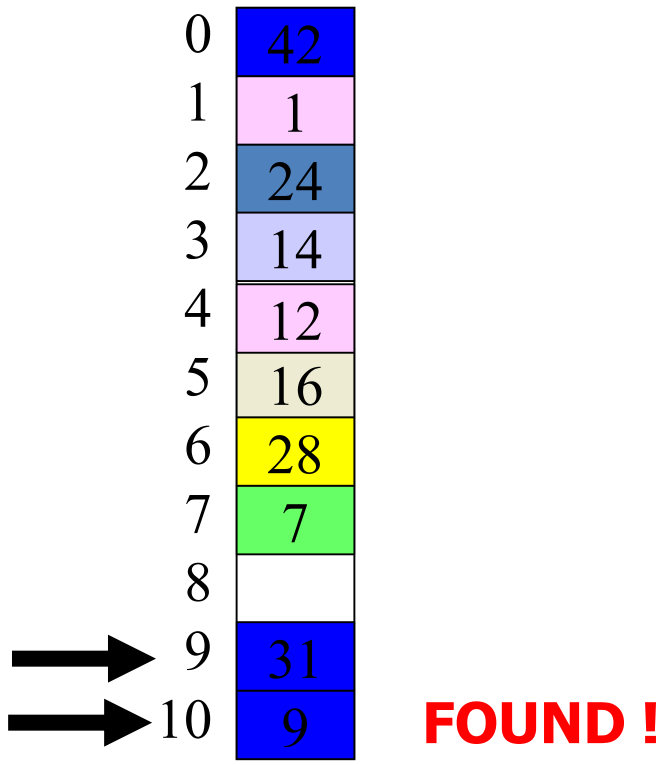
4 + 4 = 8

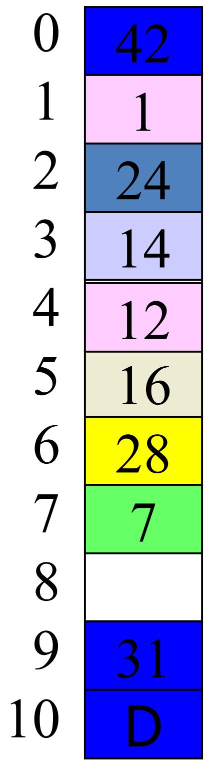
Deletion in Hashing with Linear Probi

- Since empty buckets are used to terminate search, standard deletion does not work.
- One simple idea is to not delete, but mark.
- Insert: put item in first empty or marked bucket.
- Search: Continue past marked buckets.
- Delete: just mark the bucket as deleted.
- Advantage: Easy and correct.
- Disadvantage: table can become full with dead items.

Deletion with linear probing: LAZY (Delete 9)

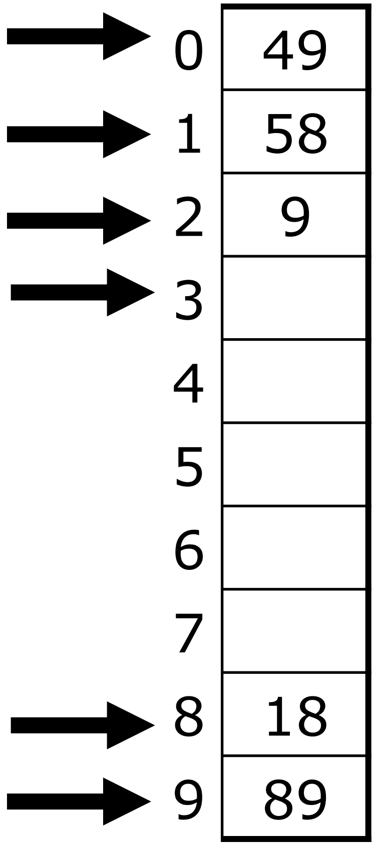
9 = 0 x 11 + 9
9 mod 11 = 9





Clustering problem

- **clustering**: nodes being placed close together by probing, which degrades hash table's performance
  - add 89, 18, 49, 58, 9
  - now searching for the value 28 will have to check half the hash table! no longer constant time...



Linear probing (H(x) + i)% Hsize

8 + 1 = 9

8 + 2 = 10 % 10 =0

8 + 3 = 11 % 10 =1

8 + 4 = 12 % 10 =2

8 + 5 = 13 % 10 =3

Quadratic Probing

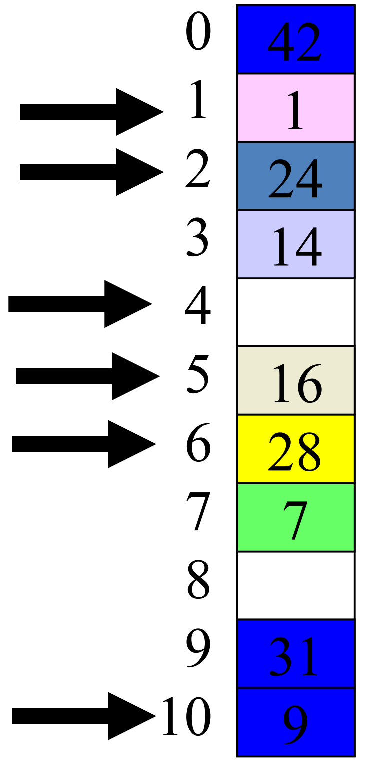
- *Solves the clustering problem in Linear Probing*
  - Check H(x)
  - If collision occurs check **H(x) + 1**
  - If collision occurs check **H(x) + 4**
  - If collision occurs check **H(x) + 9**
  - If collision occurs check **H(x) + 16**
  - ...

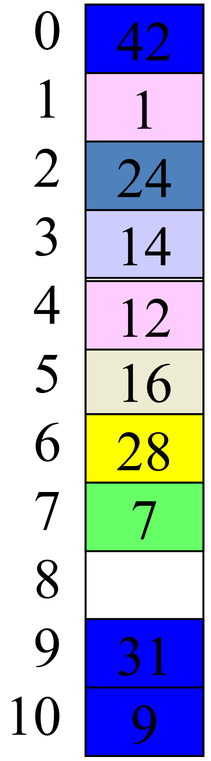
  - **$(H(x) + i^2)$ % HSize**

Quadratic Probing (insert 12)

Quadraic probing (H(x) + i²)% Hsize

12 = 1 x 11 + 1
12 mod 11 = 1





$1+1^2 = 2$

$1+2^2 = 5$

$1+3^2 = 10$

$1+4^2 = 17 \% 11 = 6$

$1+5^2 = 26 \% 11 = 4$ 🙂

Double Hashing

- *When collision occurs use a second hash function*
  - $Hash_2(x) = R – (x \bmod R)$
  - R: greatest prime number smaller than table-size
- *Inserting 12*

  $H_2(x) = 7 – (x \bmod 7) = 7 – (12 \bmod 7) = 2$
  - Check **H(x)**
  - If collision occurs check **H(x) + 2**
  - If collision occurs check **H(x) + 4**
  - If collision occurs check **H(x) + 6**
  - If collision occurs check **H(x) + 8**
  - **$H(x) + i * H_2(x)$**

Double Hashing (insert 12)

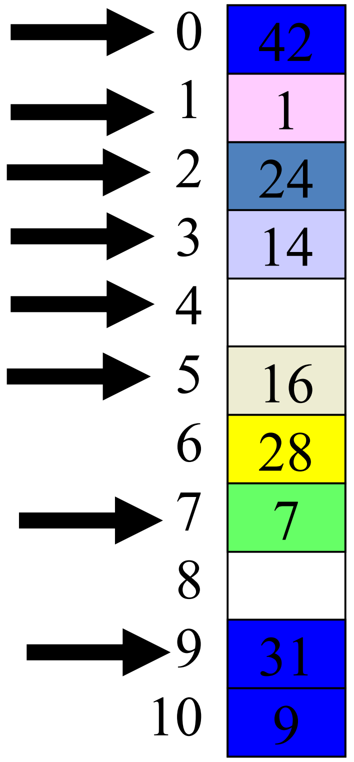
Double Hashing $(H(x) + i * H_2(x)) \% Hsize$

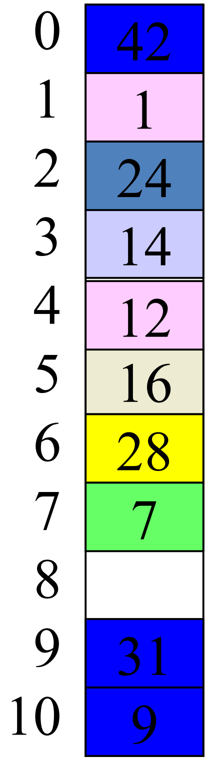
$Hash_2(x) = R - (x \bmod R)$

$12 = 1 \times 11 + 1$

$H(x) = 12 \bmod 11 = 1$

$H_2(x) = 7 - (12 \bmod 7) = 2$





$(H(x) + i * H_2(x)) \% Hsize$

$(1 + (1 * 2)) \% 11 = 3$

$(1 + (2 * 2)) \% 11 = 5$

$(1 + (3 * 2)) \% 11 = 7$

$(1 + (4 * 2)) \% 11 = 9$

$(1 + (5 * 2)) \% 11 = 0$

$(1 + (6 * 2)) \% 11 = 2$

$(1 + (7 * 2)) \% 11 = 4$

Rehashing

- If table gets too full, operations will take too long.
- Build another table, twice as big (and prime).
  - Next prime number after 11 x 2 is 23
- Insert every element again to this table

- Rehash after a percentage of the table becomes full (70% for example)

Rehashing

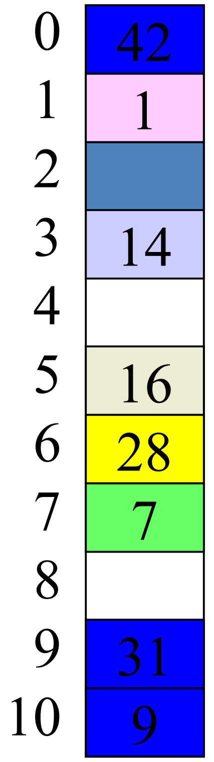
Table size * 2 (prime number)

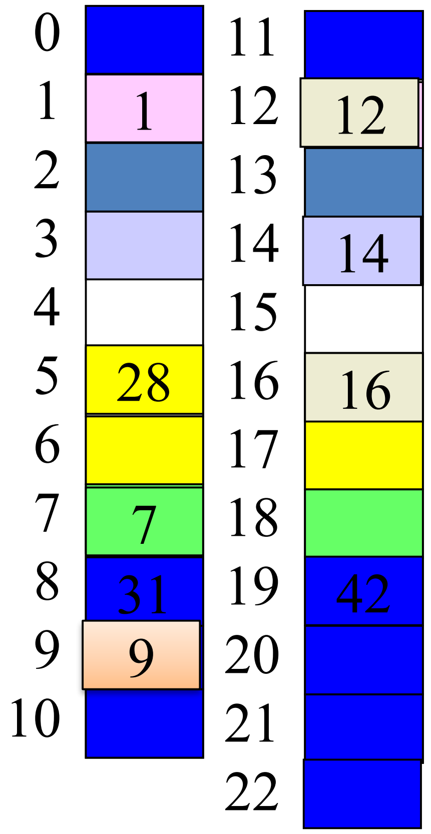
11 * 2 = 22 not a prime number

22 +1 = 23 New table size

Threshold=Total elements/Table Size *100

(8/11)*100=73 >70





42 % 23	= 19
1 % 23	= 1
14 % 23	= 14
16 % 23	= 16
28 % 23	= 5
7 % 23	= 7
31 % 23	= 8
9 % 23	= 9
12 % 23	= 12

Good and Bad Hashing Functio

- Hash using the wrong key
  - Age of a student
- Hash using limited information
  - First letter of last names (a lot of A’s, few Z’s)
- Hash functions choices :
  - keys evenly distributed in the hash table
- Even distribution guaranteed by “randomness”
  - No expectation of outcomes
  - Cannot design input patterns to defeat randomness