HASH TABLES

Data Structures and Algorithms
Waheed Iqbal

Department of Data Science, FCIT
University of the Punjab, Lahore, Pakistan

Dictionary ADT

- Dictionary entry is a” **<key_type, data_type>** pair for example, **<title, mp4_file>**
  - **<Avatar, avatar.mp4>**

- Normally associate a given key with only a single value or a pointer to data or object

- Dictionary is optimized to quickly add **<key, data>** pairs and retrieve data by key

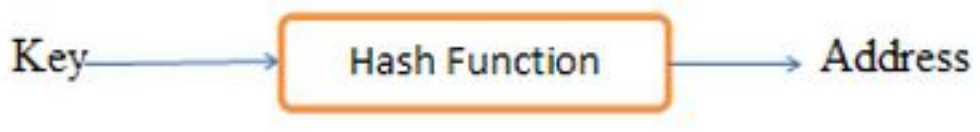
Hashing

Access table items by their keys in relatively constant time regardless of their locations

Main idea: use arithmetic operations (**hash function**) to transform keys into table locations

- the same key is always hashed to the same location
- such that insert and search are both directed to the same location in **O(1)** time!

Hash table: an array of buckets, where each bucket contains items assigned by a hash function!



Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10
- *h*(*i*) = *i* % 10
- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10
- *h*(*i*) = *i* % 10
- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	
6	
7	7
8	
9	

Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10

- *h*(*i*) = *i* % 10

- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	
6	
7	7
8	
9	

Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10

- *h*(*i*) = *i* % 10

- **Insert**: 7, 18, 41, 34

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

Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10

- *h*(*i*) = *i* % 10

- **Insert**: 7, 18, 41, 34

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

Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10

- *h*(*i*) = *i* % 10

- **Insert**: 7, 18, 41, 34

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

Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10
- *h*(*i*) = *i* % 10
- **Insert**: 7, 18, 41, 34

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

Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10

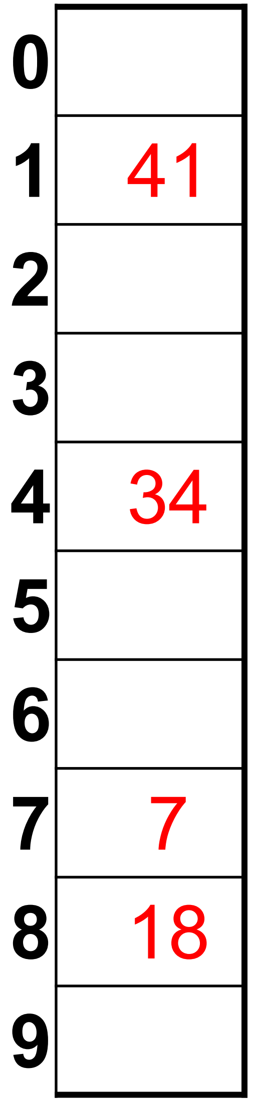
- *h*(*i*) = *i* % 10

- **Insert**: 7, 18, 41, 34

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

Hash function example

- Desirable properties of a hash function
  - efficient computation
  - deterministic/stable result
  - uniformly distributed values over range
- *h(i) = i % 10*
  - Does this function have the properties above?
- Drawbacks?
  - Lose all ordering information:
    - getMin, getMax, removeMin, removeMax
    - Ordered traversals; printing items in sorted order



Hash collisions

- Example: add 7, 18, 41, 34, then 21
  - 21 hashes into the same slot as 41!
  - Should 21 replace 41?
    - No!
- **collision**: the event that two hash table elements map into the same slot in the array
- **collision resolution**: means for fixing collisions in a hash table

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

Hash function for strings

- elements = Strings
- How do we map a string into an integer index? (i.e., how do we "hash" it?)

- Let's view a string by its letters:
  - String s : $s_0, s_1, s_2, \ldots, s_{n-1}$

- One possible hash function:
  - Treat first character as an int, and hash on that
    - $h(s) = s_0 \ \% \ TableSize$
    - Is this a good hash function? When will strings “collide”?
    - What about $h(s) = s.length \ \% \ TableSize$?

Better string hash functions

- Another possible hash function:
  - Treat each character as an int, sum them, and hash on that

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

    - What's wrong with this hash function? When will strings collide?

- Coming up with a great hash function is hard.

Collision Resolution by Linear Probing

- After checking spot h(k), try spot
  - h(k)+1, if that is full,
  - try h(k)+2,
  - then h(k)+3 etc.

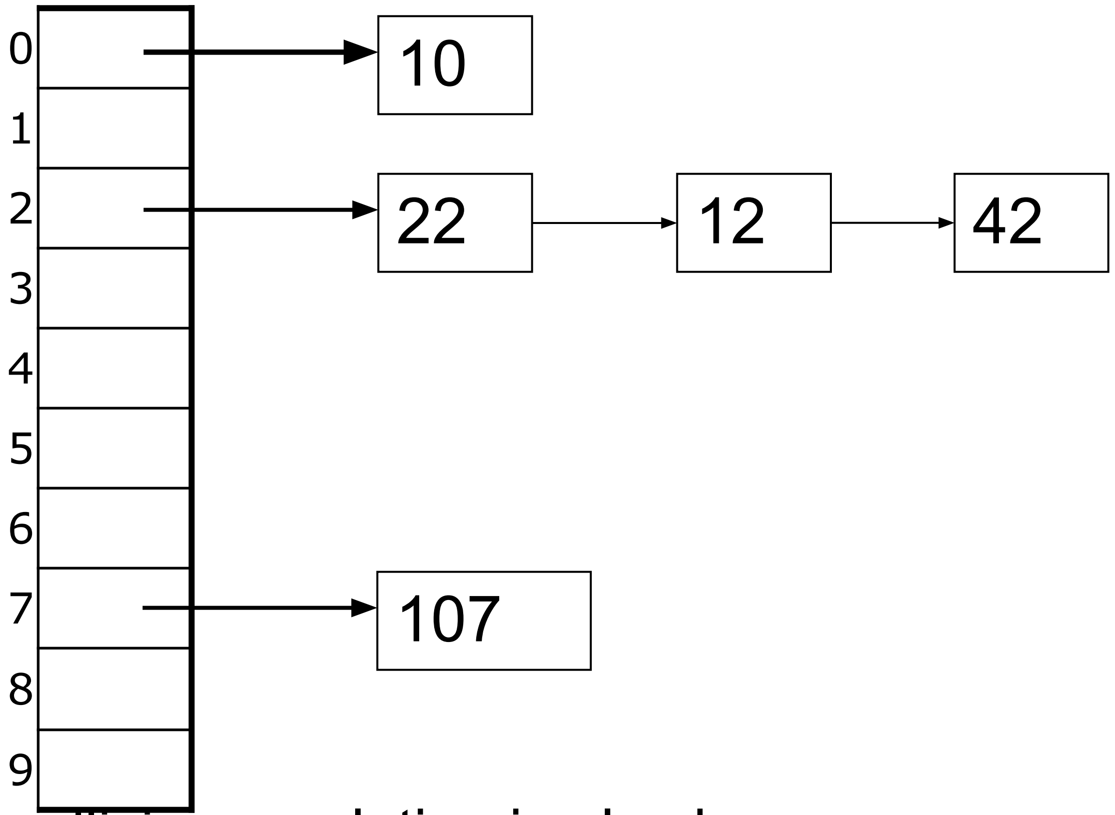
- Load Factor = Number of elements / Table size

- If the load factor is close to 1 the number of probes that may be required to find or insert a given key rises dramatically

- This method of collision resolution is also known as **close hashing**

Collision Resolution by Chaining

- **chaining**: All keys that map to the same hash value are kept in a linked list



This method of collision resolution is also known as **open hashing.**

Designing a Good Hash Functions

- We need to adopt a good hash function to minimize the collisions
- A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the **m** slots.
- Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn.

Designing a Good Hash Functions (Cont.)

The division method

$$h(k) = k \bmod m$$

- we usually avoid certain values of **m**. For example, **m** should not be a power of **2**
- A prime not too close to an exact power of 2 is often a good choice for **m**.

Designing a Good Hash Functions (Cont.)

- **Multiplication Method**

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$$

Where 0 < A <1 and and m is not critical here we usually take m = 2^P for positive integer P.

Knuth *(father of analysis of algorithm)* suggests value of A

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887$$

Universal Hashing

- Enables to select hash function randomly from a family of hash functions
- Fix hash function may give us linear time search instead of constant
- Using Universal Hashing algorithm may behave differently on the same input

Hashtable in Python

- In Python, the average time complexity for inserting and searching in a dictionary (hash table) is O (1). However, it's important to note that this is an average-case complexity.

- In the worst-case scenario, particularly when there are many collisions (i.e., multiple keys hashing to the same index), the time complexity for insertion and search operations can degrade to O (n)

Solve in O(n)?

1. **Find count for each element in an array**

Solve in O(n)?

2. Jim is a kidnapper who wrote a ransom note, but now he is worried it will be traced back to him through his handwriting. He found a magazine and wants to know if he can cut out whole words from it and use them to create an untraceable replica of his ransom note. The words in his note are case-sensitive and he must use only whole words available in the magazine. He cannot use substrings or concatenation to create the words he needs.

Given the words in the magazine and the words in the ransom note, print Yes if he can replicate his ransom note exactly using whole words from the magazine; otherwise, print No.

Credits

This lecture notes contains some of the material from the following resources:

- These notes contain material from Chapter 11 of Cormen, Leiserson, Rivest, and Stein (3rd Edition).
- http://courses.cs.washington.edu/courses/cse373/12sp/lectures/04-30-hashing/16-hashing.ppt