CS2040S TUTORIAL 4

Lucia Tirta Gunawan
luciatirtag@u.nus.edu

WELCOME TO THE FOURTH TUTORIAL

THIS WEEK'S TOPIC IS…

HASHING

WHY NEED HASHING?

1. Map large integers to smaller integers (when keys are sparse/not dense)
   - Instead of a[14527266] = 5 we can hash the key, for example h(14527266) = 1, and we can do a[1] = 5
2. Map non-integer keys to integers
   - h("Book") = 5. Since we cannot use string as index in array, we can make a function h that maps "Book" to 5, and we can do a[5] = 3

Hashing is used to address the limitation of Direct Addressing Table

HASHING COLLISION

Hash is many to one mapping

So we can have collision where the key maps to the same number.

For example h(67774385) = h(66752378)

Perfect hash function is a one-to-one mapping, no collision.

Possible if all keys are known beforehand

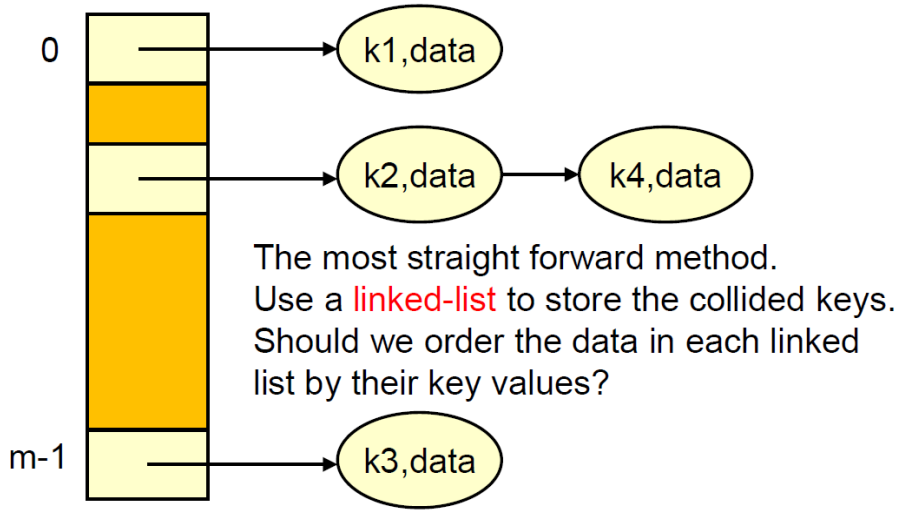
Minimal perfect hash function: The table size is the same as the number of keywords supplied.

GOOD HASH FUNCTIONS

1. Fast to compute
2. Scatter keys evenly throughout the hash table
3. Less collisions
4. Need less slots (space)

HANDLING COLLISION

1. Separate Chaining

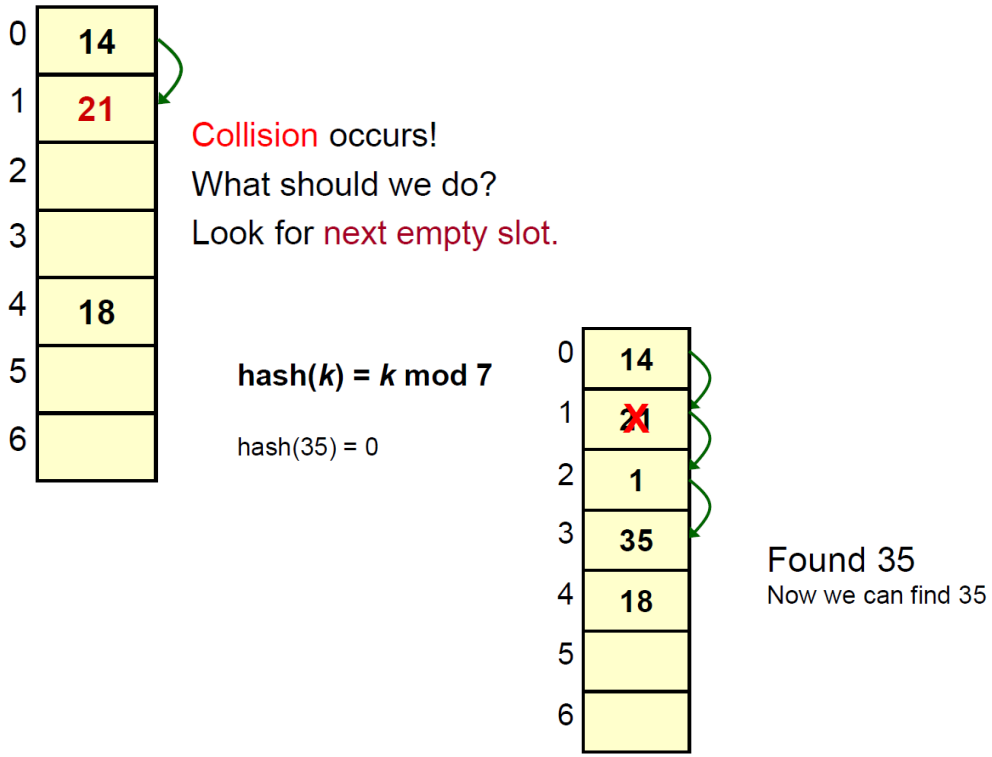


Use linked list to handle collision

With bounded load factor (α), the average case performance is:
Insert: O(1)
Find: O(1)
Delete: O(1)

HANDLING COLLISION

2. Linear Probing



Find next empty slot to handle collision

When using probing, we cannot delete element.
Just mark an element as deleted

Problem: primary clustering (create many consecutive occupied slots)

Normal linear probing sequence

hash(key)
(hash(key) + **1**) % *m*
(hash(key) + **2**) % *m*
(hash(key) + **3**) % *m*

Modified linear probing to handle clustering

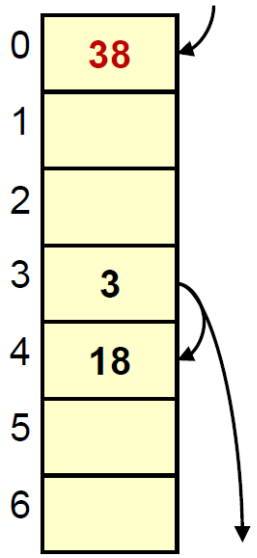
hash(key)
(hash(key) + **1** * ***d***) % *m*
(hash(key) + **2** * ***d***) % *m*
(hash(key) + **3** * ***d***) % *m*
d is a constant positive integer coprime to m

HANDLING COLLISION

3. Quadratic Probing

hash(*k*) = *k* mod 7

hash(38) = 3



Find next empty slot (with quadratic steps) to handle collision

Problem: secondary clustering (two keys have the same initial position, their probe sequences are the same)

Quadratic probing sequence

hash(key)

(hash(key) + **1**) % *m*

(hash(key) + **4**) % *m*

(hash(key) + **9**) % *m*

…

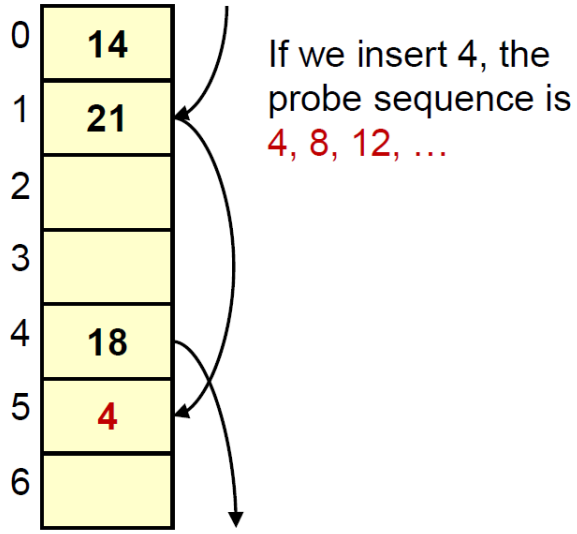
(hash(key) + $\mathbf{k^2}$) % *m*

HANDLING COLLISION

4. Double Hashing

hash(*k*) = *k* mod 7
$hash_2$(*k*) = *k* mod 5

hash(4) = 4
$hash_2$(4) = 4



If we insert 4, the probe sequence is 4, 8, 12, …

Double hashing sequence
hash(key)
(hash(key) + 1*$hash_2$(key)) % *m*
(hash(key) + 2*$hash_2$(key)) % *m*
(hash(key) + 3*$hash_2$(key)) % *m*
$hash_2$: secondary hash function, the number of slots to jump each time a collision occurs.

BLOOM FILTER

To check if a key exists in a hash table. But the value is only 0/1. We don't store the key (less memory). That's why we can't do a removal in BF.

Insertion: Use m different hash functions that maps to m indices and set the bit array value in these indices to 1.

Retrieval: Find the indices from the m hash functions, if at least 1 value at these indices in the bit array is 0, the key is not in the filter.

We could have false positive (the value of the hash table in $H_1(k)$, … $H_m(k)$ are all 1 but it actually comes from other keys that maps to these indices)

From the number of keys inserted and the probability of false positive that we want to achieve, we can calculate the size of the bit array and the number of hash functions used

Need to calculate m hash functions → time space trade off

Usage: check for bad password, check for online status

Q1. SIMULATION (LINEAR PROBING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use linear probing as the collision resolution technique:

	0	1	2	3	4
I(7)					
I(12)					
I(22)					
D(12)					
I(8)					

Q1. SIMULATION (LINEAR PROBING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use linear probing as the collision resolution technique:

	0	1	2	3	4
I(7)			7		
I(12)			7	12	
I(22)			7	12	22
D(12)			7	~~12~~	22
I(8)			7	8	22

Q1. SIMULATION (QUADRATIC PROBING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use quadratic probing as the collision resolution technique:

	0	1	2	3	4
I(7)					
I(12)					
I(22)					
I(2)					

Q1. SIMULATION (QUADRATIC PROBING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use quadratic probing as the collision resolution technique:

	0	1	2	3	4
I(7)			7		
I(12)			7	12	
I(22)		22	7	12	
I(2)	unable	to	find	free	slot

Q1. SIMULATION (DOUBLE HASHING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use double hashing as the collision resolution technique, `g(key) = key % 3`:

	0	1	2	3	4
I(7)					
I(22)					
I(12)					

Q1. SIMULATION (DOUBLE HASHING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use double hashing as the collision resolution technique, `g(key) = key % 3`:

	0	1	2	3	4
I(7)			7		
I(22)			7	22	
I(12)	infinite	loop	from	g(12) = 0	

The secondary hash function must not evaluate to 0,

Q1. SIMULATION (DOUBLE HASHING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use double hashing as the collision resolution technique, g(key) = 7 - (key % 7).

	0	1	2	3	4
I(7)					
I(12)					
I(22)					
I(2)					

Q1. SIMULATION (DOUBLE HASHING)

h(key) = key % 5, I(X): add(X), D(X): remove(X)

Use double hashing as the collision resolution technique, `g(key) = 7 - (key % 7)`.

	0	1	2	3	4
I(7)			7		
I(12)			7		12
I(22)			7	22	12
I(2)	infinite	loop	from	g(2) % m = 0	

All hash values generated must also be co-prime with m, the size of the hash table. To achieve this, we can use a prime number m' < m for the secondary has function.

Q2. HASH FUNCTIONS

Comment the flaw of the following cases

The hash table has size 100 with positive even integer keys. The hash function is h(key) = key % 100.

Answer:

1. No key will be hashed directly to odd-numbered slots in the table, resulting in wasted space
2. Higher chance of collision in the remaining slots
3. Hash table size may not be good as it is not a prime number. If there are many keys that have identical last two digits, then they will all be hashed to the same slot, resulting in many collisions.

Q2. HASH FUNCTIONS

Comment the flaw of the following cases

The hash table has size 49 with positive integer keys. The hash function is h(key) = (key*7) % 49.

Answer:

1. All keys will be hashed only into slots 0, 7, 14, 21, 28, 35 and 42.
2. The hash table size is not a prime number

Q2. HASH FUNCTIONS

Comment the flaw of the following cases

The hash table has size 100 with non-negative integer keys in the range [0, 10000]. The hash function is h(key) = $\lfloor \sqrt{key} \rfloor$ **% 100.**

Answer:

1. Keys are not uniformly distributed. Many more keys are mapped to the larger-indexed
2. The hash table size is not a prime number.

Q2. HASH FUNCTIONS

Comment the flaw of the following cases

The hash table has size 1009, and keys are valid email addresses. The hash function is h(key) = (sum of ASCII values of each of the last 10 characters) % 1009.

http://www.asciitable.com/

Answer:

1. Keys are not evenly distributed because many email addresses have the same domain names e.g. “u.nus.edu”, “gmail.com”. Many email addresses will be hashed to the same slot, resulting in many collisions.

Q2. HASH FUNCTIONS

Comment the flaw of the following cases

The hash table has size 101 with integer keys in the range of [0, 1000]. The hash function is h(key) = $\lfloor key * random \rfloor$ % 101, where 0.0 ≤ random ≤ 1.0.

Answer:

1. This hash function is not deterministic. The hash function does not work because, while using a given key to retrieve an element after inserting it into the hash table, we cannot always reproduce the same address.

Q2. HASH FUNCTIONS

Comment the flaw of the following cases

The hash table has size 54 with String keys, with the hash function

```
int hash(String key) {
    h = 0
    for (int i = 0; i <= key.length()-1; i++)
        h += 9 * (int) key.charAt(i)
    h = (h mod 54)
    return h
}
```

Answer:

1. This is not a good hash function because 9 and 54 share a common divisor of 9, so the hash function only produces hash values that are multiples of 9, or 0 itself, i.e. 0, 9, 18, 27, 36, 45, which means it only uses 6 out of the 54 possible locations in the array, which is not uniform.

Q3. STRING MATCHING

Abridged problem description:

Given a long string, find a list of k-letter words in the text.

Design and implement an algorithm that performs a **preprocessing step** on the text, so that you can subsequently **query the number of occurrences of a k-letter** word within the text of length n in **O(k) average time**. State, with justification, the time complexity of the algorithm

Q3. STRING MATCHING

1. Assume k is known, loop through all possible (n - k+1) k-letter words in the text of length n.
2. Create a Hash Table with k-letter words as keys, and the values being the frequency of appearance in the text.
3. Assuming the evaluation of the hash function is dependent on the length of the string k, each operation on the hash table such as insertion and query will take O(k) average time. Thus, the average time complexity is O(kn).
4. In the worst case, a linear number of probes will be required due to collisions, such as if each k-letter word is unique and hashes to the same value. If separate chaining is used, for every word inserted we need to search through the entire chain before adding it to the end of the chain if it is not inside. The results in a worst case time complexity of $O(kn^2)$.

Q4. FINDING SUM

Abridged problem description:

There are four components: Appetizers, Soups, Mains, Dessert

Each component has n items

Choose 1 item from each component such that their prices add up to SGD 100

Find the most efficient algorithm to solve this, and state their time complexity.

Q4. FINDING SUM

We denote the four lists in the menu as m1, m2, m3, m4.

For each pair of values (x, y) on lists (m1, m2), add x + y to a hash table (with the attached item names as the value). This take $O(n^2)$ time (to generate all the pairs), assuming we can achieve O(1) average time to insert into our hash table.

For every pair of values (w, z) on lists (m3,m4), look up 100 - (w + z) in the hash table until you find a value that is present in the hash table. This also takes $O(n^2)$ time.

Time complexity $O(n^2)$

Q5. BLOOM FILTER

Why is there no delete/remove operation for Bloom Filter?

As given in the example from the lecture notes, the bits set for a key can overlap with the bits set for other keys in the Bloom Filter. So if we remove a key (meaning set the bits representing the key back to 0) then it will also "`remove"' other keys that share the bits which is not what we want. Thus, there is no delete/remove operation for a Bloom Filter.

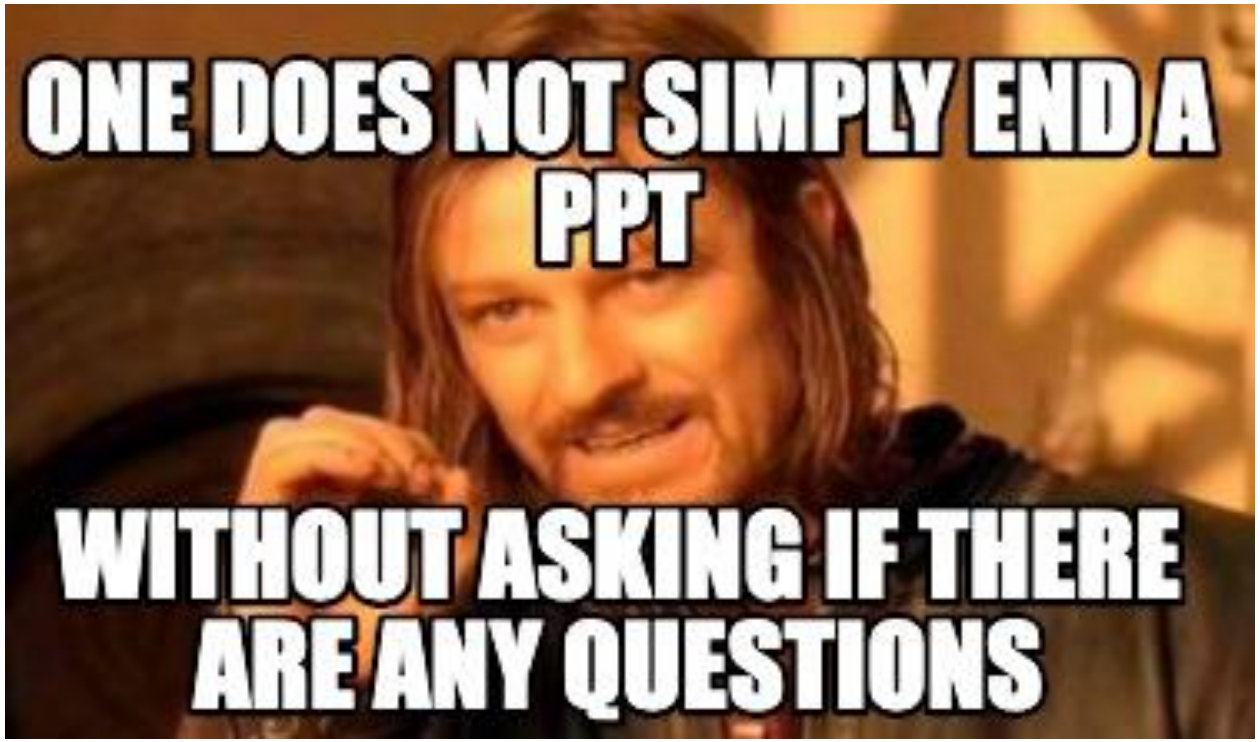
Can you think of a modification of Bloom Filter that allows delete/remove?

Counting Bloom Filter*

COUNTING BLOOM FILTER

- Uses counters instead of bits
- Insertion is done by incrementing the counters
- Retrieval: take a threshold as an argument and check if all counters in all m indices from m hash functions are greater than or equal to the threshold.
- Deletion: decrement the counters
- False positive: Adding “foo” and “bar” three times each, could for example make it look like “baz” has been added three times
- No false negatives: a counting Bloom filter would be able to say with certainty that “baz” has not been added more than three times.
- One caveat here is that you may only delete an element that has previously been inserted

ANY QUESTION?



SEE YOU IN THE NEXT TUTORIAL!