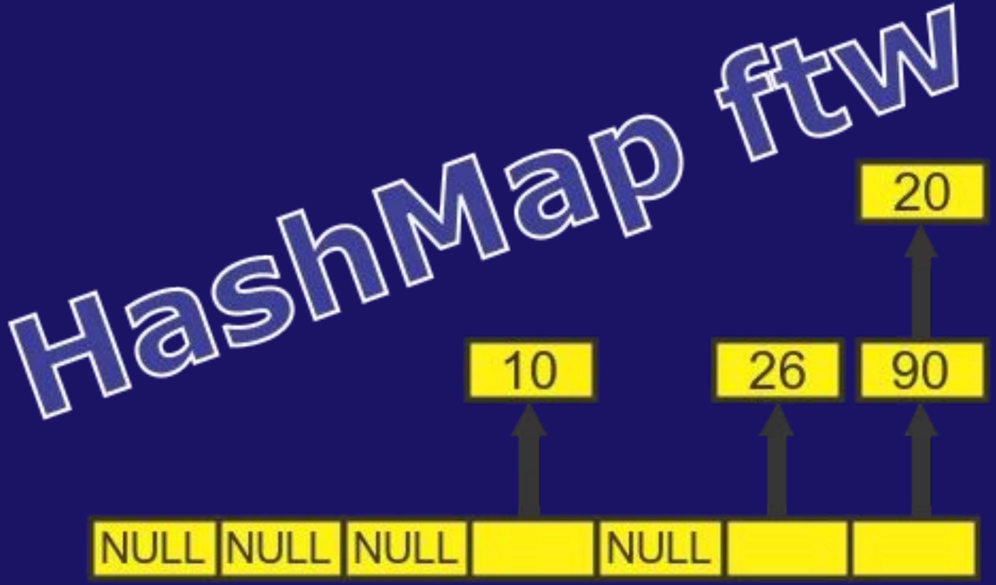
CS2040

Tutorial 4: Hashing

Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)

Let's start with...



:D

HASHING

Why need hashing?

- 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

- 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)
5. Consistent/deterministic

Handling collisions

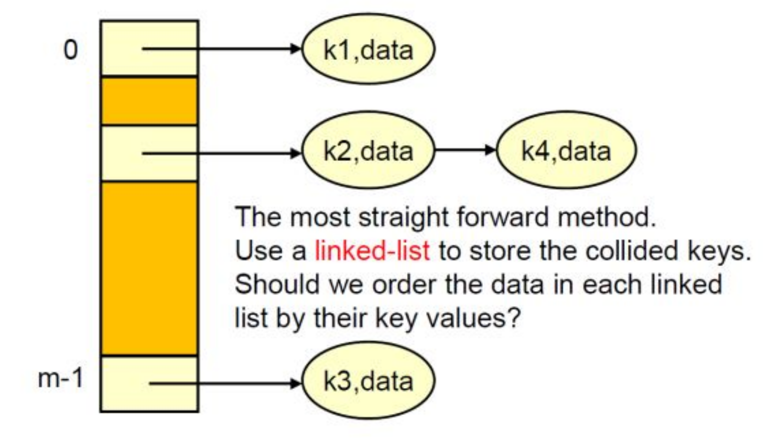
1. Separate chaining

   Use linked list to handle collision

   Insert: O(1)

   Find: O(n), n: length of linked list

   Delete: O(n), n: length of linked list



The most straight forward method.
Use a linked-list to store the collided keys.
Should we order the data in each linked list by their key values?

Handling collisions

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)

Handling collisions

2. Linear probing

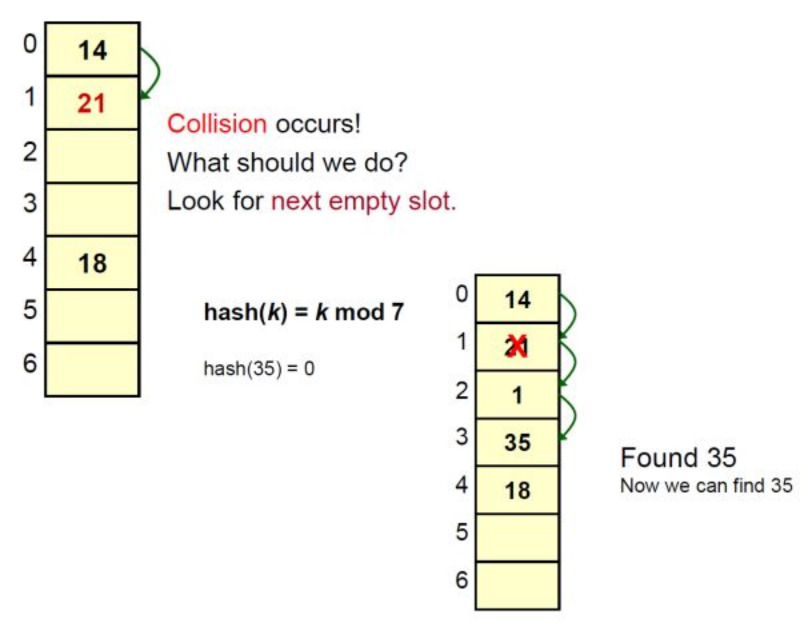
Normal linear probing sequence

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



Collision occurs!
What should we do?
Look for next empty slot.

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

hash(35) = 0

Found 35
Now we can find 35

Handling collisions

3. Quadratic probing

   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)

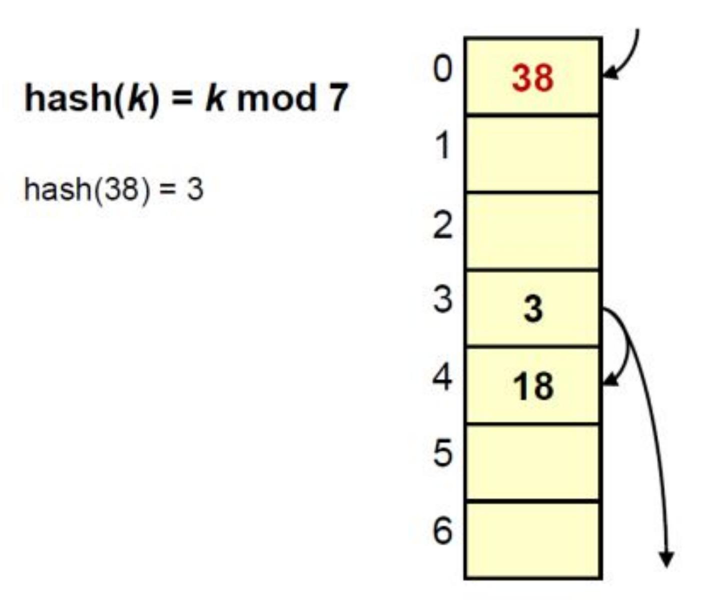
   (hash(key) + 1) % m

   (hash(key) + 4) % m

   (hash(key) + 9) % m

   …

   (hash(key) + k^2) % m



Handling collisions

4. Double hashing

   Double hashing sequence

   hash(key)

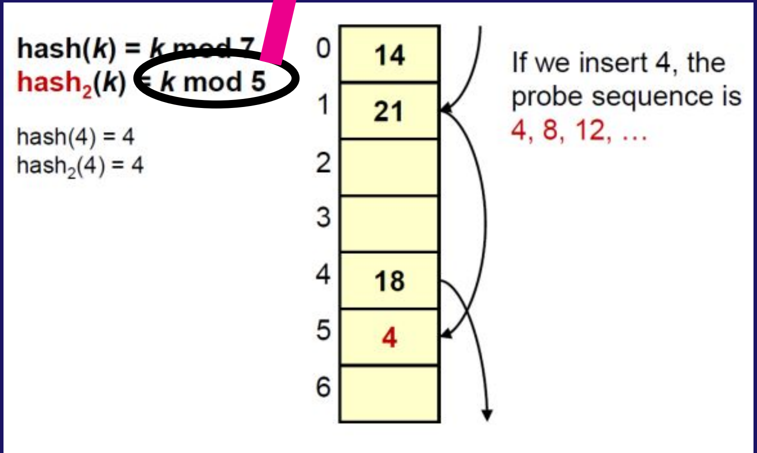
   (hash(key) + 1 * hash_2 (key)) % m

   (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.

5 – (k mod 5) is better. Why?



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

hash_2(*k*) = *k* mod 5

$\text{hash}(4) = 4$

$\text{hash}_2(4) = 4$

0	14
1	21
2	
3	
4	18
5	4
6	

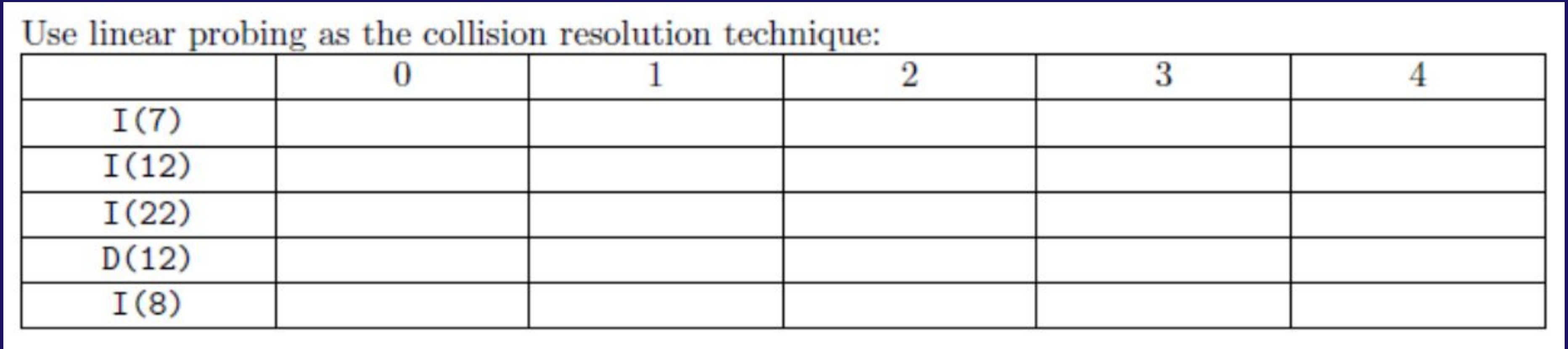
If we insert 4, the probe sequence is 4, 8, 12, …

01

PROBING SIMULATION

Linear Probing

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

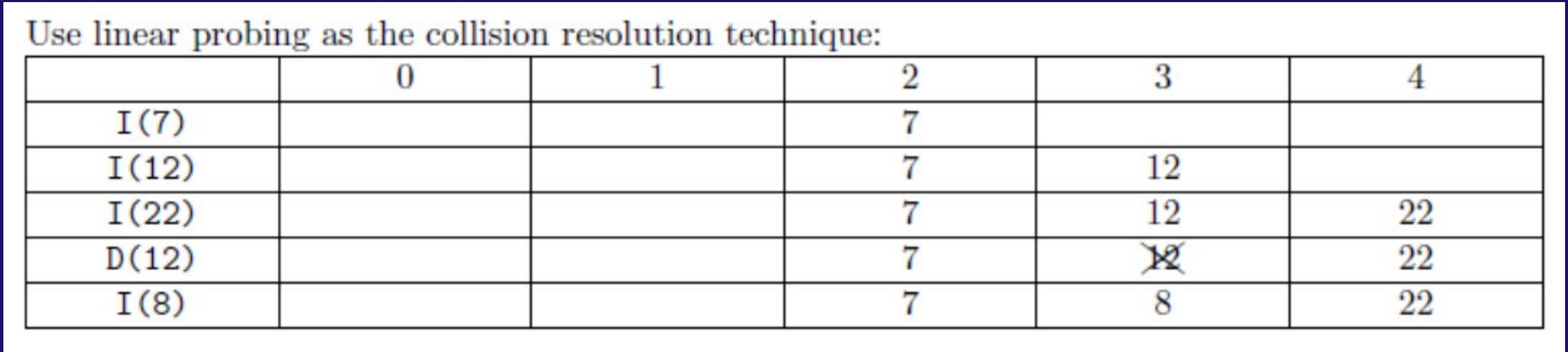


Use linear probing as the collision resolution technique:

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

Linear Probing

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

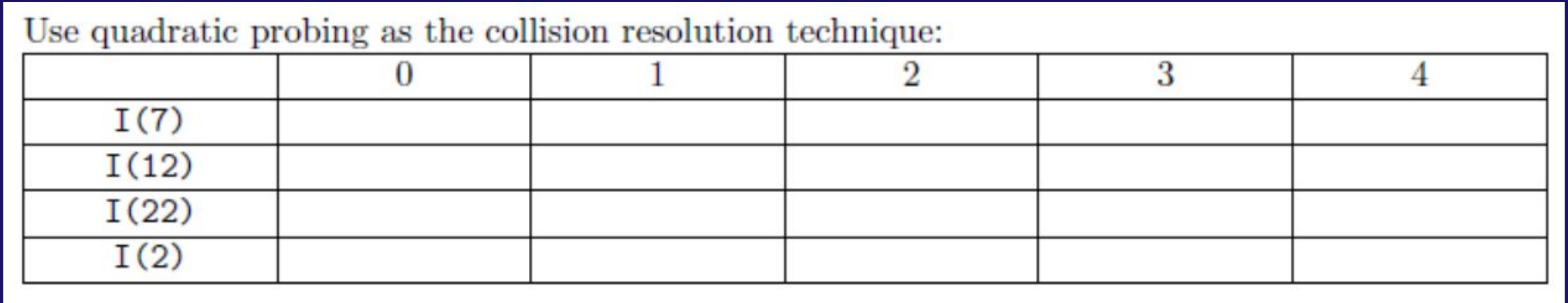


Use linear probing as the collision resolution technique:

	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

Quadratic Probing

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

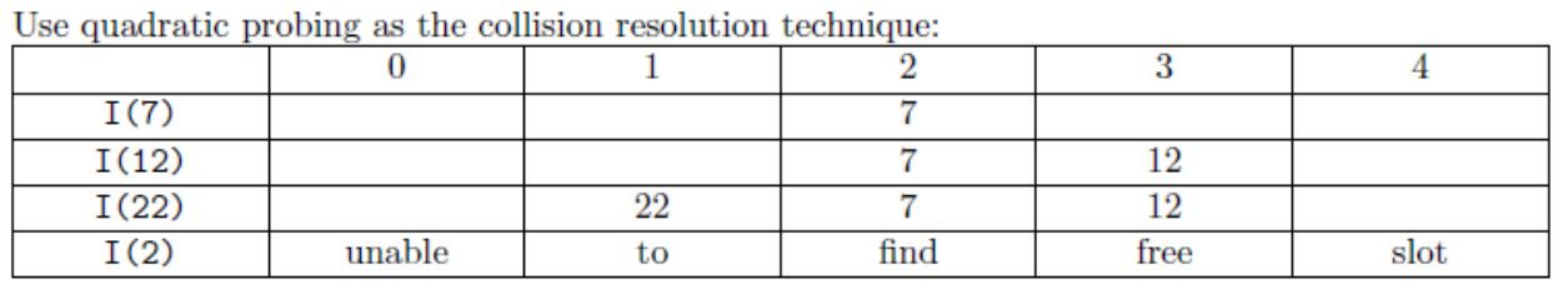


Use quadratic probing as the collision resolution technique:

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

Quadratic Probing

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

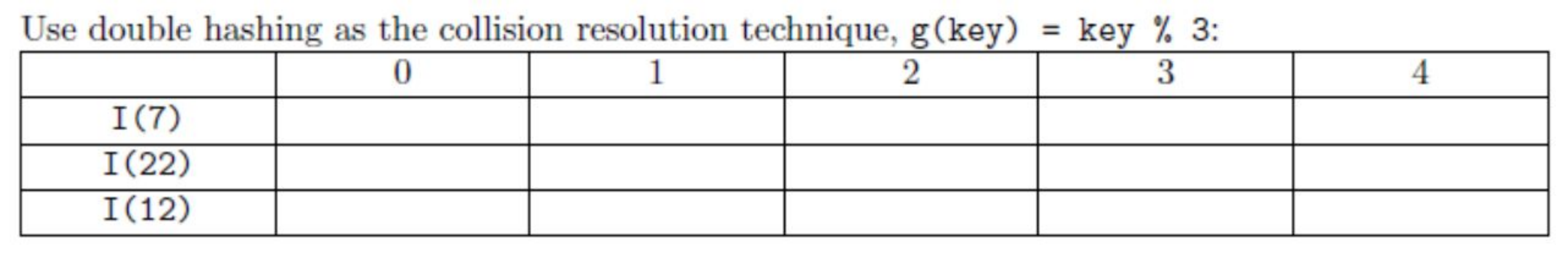


Use quadratic probing as the collision resolution technique:

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

Double Hashing

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

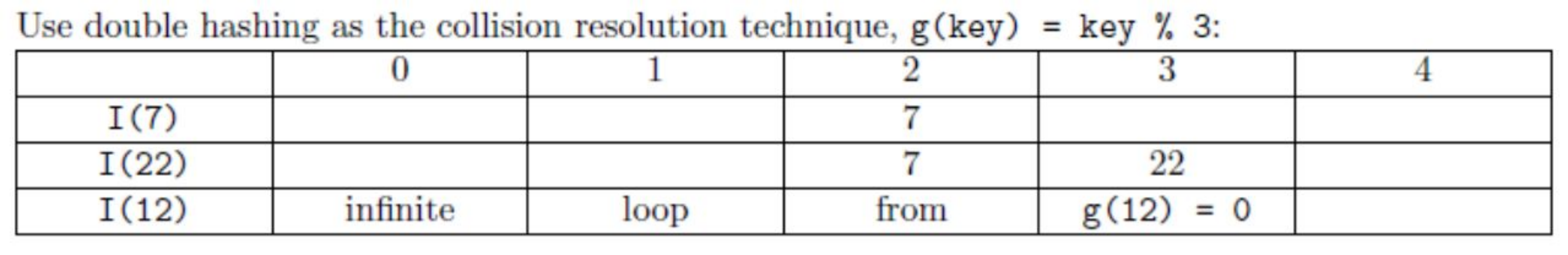


Use double hashing as the collision resolution technique, g(key) = key % 3:

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

Double Hashing

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



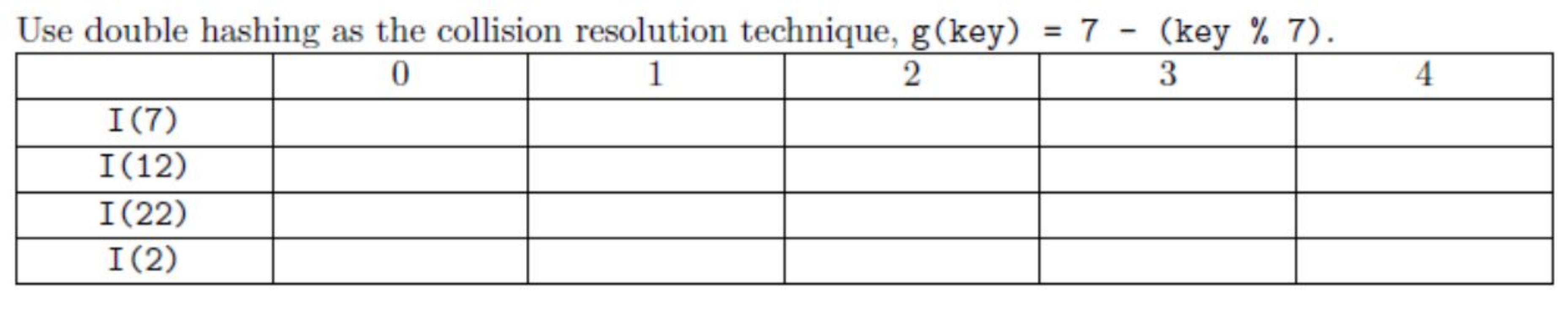
Use double hashing as the collision resolution technique, g(key) = key % 3:

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

Moral of the story:
The secondary hash function must not evaluate to 0.

Double Hashing

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

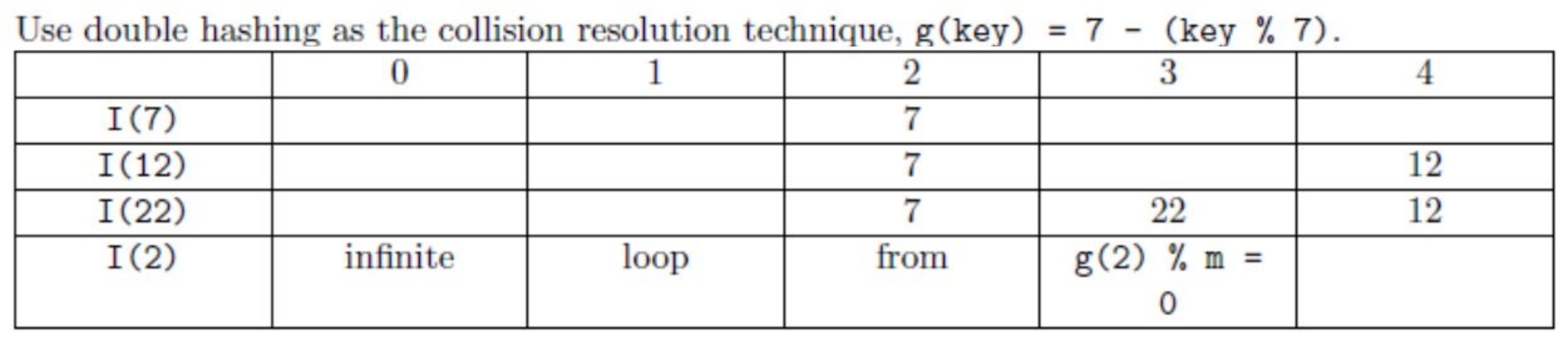


Use double hashing as the collision resolution technique, g(key) = 7 - (key % 7).

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

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	

Moral of the story:
All hash values generated must also be co-prime with m, the size of the hash table.
In order to achieve this use a prime number m’ < m for the secondary hash function.

02

HASH FUNCTIONS

Some Methods

Division Method

$h(key) = key \% m$

where m is the table size.

m should be a prime number close to a power of 2.

Multiplication Method

$h(key) = \lfloor m(key \cdot a - \lfloor key \cdot a \rfloor) \rfloor$

where m is the table size and $0 < a < 1$.

Choice of a: golden ratio.

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.

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.

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

$$\mathrm{h}(\mathrm{key}) = \left\lfloor \sqrt{\mathrm{key}} \right\rfloor \% \, 100.$$

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.

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.

See http://www.asciitable.com/

Answer:
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.

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

$$\mathrm{h}(\mathrm{key}) = \lfloor \mathrm{key} * \mathrm{random} \rfloor \,\%\, 101$$

where 0.0 ≤ random ≤ 1.0.

Answer:

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.

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:

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.

03

STRING MATCHING

String Matching

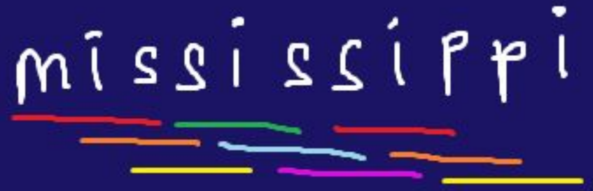
Abridged problem description:

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.

String Matching



mis	1	sip	1
iss	2	ipp	1
ssi	2	ppi	1
sis	1		

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 $O(kn^2)$.

Extra: Preprocessing

The Idea Behind Preprocessing

Sometimes, you can preprocess the input into a structure or format that is easier or faster to work with.

Now, after the preprocessing step, we can then simply answer all the queries by simply reading off for each query.

In the previous question, if we have a string of length n and we want to find the frequency of a k-letter word, and q queries. The time complexity will be $O(nk + qk) = O((n+q)k)$. $O(nk)$ for preprocessing and $O(qk)$ for answering q queries.

Now, let's try with another example preprocessing question.

Extra: Preprocessing

There is an array of n integers. You are given q queries.
The i-th query asks:

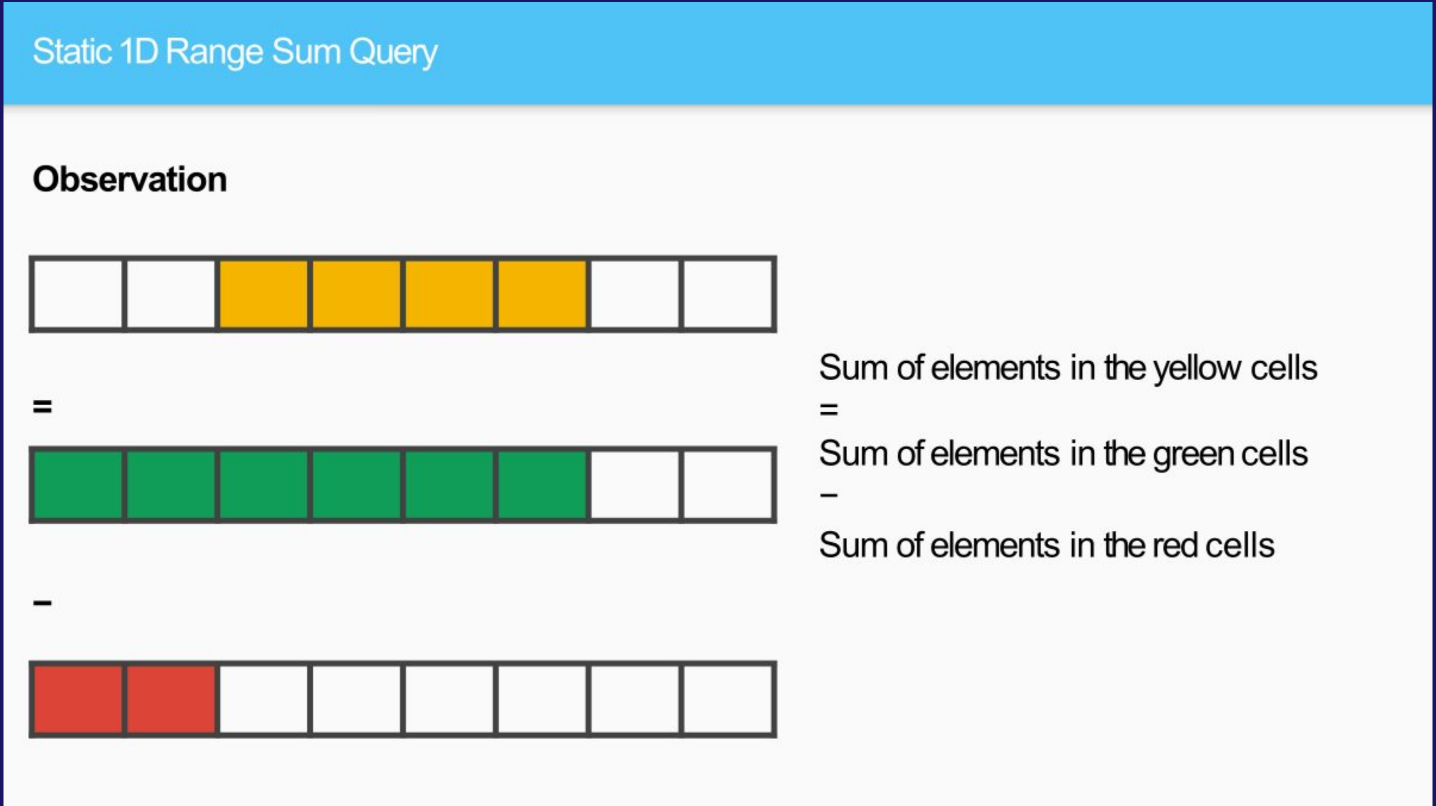
what's the sum of the x_i-th integer to the y_i-th integer, inclusive?

Your algorithm should run in O(n + q) time.

Example:
A = [2, 5, 1, 4, 3, 2, 6, 1]
query(2, 5) = A[2] + A[3] + A[4] + A[5] = 10
query(1, 2) = A[1] + A[2] = 6

You can't use the naive method as it will run in total O(qn) time!

Extra: Preprocessing



Static 1D Range Sum Query

Observation

=

–

Sum of elements in the yellow cells

=

Sum of elements in the green cells

–

Sum of elements in the red cells

Extra: Preprocessing

Preprocess the array A!
Construct another array B, where B[0] = A[0] and B[i] = B[i-1] + A[i]. This takes O(n) time.

Example:
A = [2, 5, 1, 4, 3, 2, 6, 1]
B = [2, 7, 8, 12, 15, 17, 23, 24]

Extra: Preprocessing

Preprocess the array A!
Construct another array B, where B[0] = A[0] and B[i] = B[i-1] + A[i]. This takes O(n) time.

For every query, if x_i = 0 then we can simply output B[y_i]. Otherwise, output B[y_i] - B[x_i - 1]. This takes O(1) time.

For q queries, this takes O(q) time, giving a final time complexity of O(n + q).

Example:
A = [2, 5, 1, 4, 3, 2, 6, 1]
B = [2, 7, 8, 12, 15, 17, 23, 24]

query(2, 5) = A[2] + A[3] + A[4] + A[5] = 10
query(1, 2) = A[1] + A[2] = 6

query(2, 5) = B[5] - B[1] = 17 - 7 = 10
query(1, 2) = B[2] - B[0] = 8 - 2 = 6

04

FINDING SUM

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.

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.

Overall, the time complexity is $O(n^2)$.

Similar/same problem: https://leetcode.com/problems/4sum/
Nerfed version: https://en.wikipedia.org/wiki/3SUM

05

BLOOM FILTER

this is only for CS2040S, please skip otherwise :)

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. However a modification to the standard bloom filter called a counting bloom filter does support the remove operation.

THE END!