Tutorial 05
Midterm Post-Mortem
Hash Function

CS2040C Semester 1 2020/2021

By Matthew and Jin Zhe, AY1920 S1, adapted from slides by Ranald+Si Jie, AY1819 S2

Midterm post-mortem

Solutions + Review

Q1 Code-Tracing

- Just trace the code.
- 1a: 1024
  - 8*128
  - 7*128 is incorrect but gets partial 1/3 marks
- 1b: heretoo
  - Testing fallthrough (shown in /thegrandadventure)

Q1 Code-Tracing

- 1c: 3.4000000
  - setprecision sets dp (shown in /pervasiveheartmonitor)
- 1d: 0024CCShiisst
  - Just (bubble) sort
- 1e: 7
  - there are 1 abcd each and 3 es, so f will be the 8th character (index 7)
  - Find difference between iterator positions.

Q1 Code-Tracing

- Median: 10/15
- Mean: 9.7/15

- Just code tracing of stuff that’s covered before.

Q2 Complexity Analysis

- 2a: O(nlogn)
    - n iterations of log n
- 2b: O(1)
- 2c: O(n)
- 2d: O(m^2)
- 2e: O(l log l)

- Median: 5/5
- Mean: 4.4/5
- Pretty straightforward, mostly full scores.

Q3 Sorting

- 3a: 1
  - Only sorted array goes through a single pass of bubble sort
- 3b: 720
  - 6! = 720
  - Work backwards from 6^{th} pass: {2, 1, 3, 4, 5, 6, 7}
  - Realise that 1 must then be right-most (otherwise above array cannot happen)
  - Any permutation that satisfies this works

Q3 Sorting

- 3c: Yes correct, but no point – merge sort O(nlogn), overall O(dnlogn)
    - Might as well just merge sort from the start
- 3d: Merge sort. Not frequently used due to not in-place, high constant factor.
    - With most common implementation
- (not marked yet)

Q4 LL, Queue, Stack

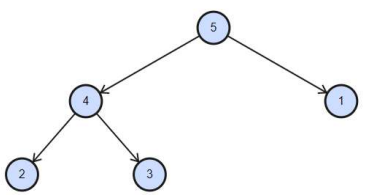
- 4a: Cannot. To do O(n) duplicate checks in-place, need sorted list which requires O(nlogn).
- 4b: Correct, but severe memory leaks will occur as deleted vertices never released from memory.
- 4c: std::deque D
  - push_back/pop_back
  - fixed-sized has fixed size issue
  - Binary heap H can be turned into stack, but O(log n)

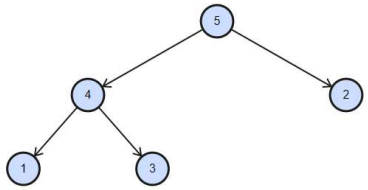
Q4 Data Structures

- 4d: fixed size array A (of size C)
  - Circular queue: https://visualgo.net/en/list?slide=5-3
  - 1-meter is distraction
  - Fixed size array is lightweight, avoids need for resizing like in std::deque vector of vectors
- 4e: Use 2 normal queues, but with pair, 2nd field for arrival counter
  - O(1), check 2nd field whenever front() is called
  - PS3B? doctorkattis

Q5 PQ

- 5a: {-1, 5, 4, 1, 2, 3} or {-1, 5, 4, 2, 1, 3}





  - Easy, just make sure bubble down to left side

- 5b: Remove(v). If Remove(v) is not NULL, Insert(v+deltav).
  - Both O(logn), overall O(logn)
  - PS3B? doctorkattis

- Mean: 5.3/10 (below expected)

Q6 Application - Sorting

- O(N^2), just run any "slow" comparison-based sorting algorithm, e.g., bubble sort (2 marks)
    - Why would you do this with O(N log N) std::sort?
- O(N log N), just run any "fast" comparison-based sorting algorithm, e.g., merge sort (3 marks)
- O(NK) brute force, find min in each of subarray of size K and print (5 marks)
    - Min is O(K), do O(N) times, overall O(NK)

Q6 Application - PQ

- Best Algorithm?
- Idea: Each K window, do something
- Do something = find minimum
- Find minimum in K elements faster than O(K)?
- Min PQ
- Slide the min PQ forward
- Extract and Insert in O(log K)

Q6 Application - PQ

- O(N log K), O(N) pass, use O(log K) min PQ to extract min and then insert the next value (very few will be able to reach this stage) (10 marks)
  - Build heap once in O(K), Extract min O(log K), Insert O(log K), insert O(N) times
  - Overall O(K) + (O(log K) + O(log K))*O(N) = O(N log K)
  - Big brain algorithm

Q6 Application - Sorting

- PS: O(d(N+k)) Radix/ O(N) Counting sort are wrong. (1 mark for effort)
    - Don’t blindly copy.

- Mean: 2.6/10
    - Lower than 3 marks just for std::sort

Q7 Application - PQ

- O(NK) brute force, try all possible K (5 marks)
  - Just check every set of K

- O(N log K), O(N) pass, use O(log K) min PQ with delete any (to delete item that is K indices old from min PQ) (10 marks)
  - Similar to Q6, with delete O(log K) as well
  - Hard to get if Q6 wasn't 10/10.

Q7 Application - Deque

- O(N), sliding window with deque, CP4 Section 9.2, variant no 4 (very few will be able to reach this stage) (15 marks)
  - Not expected to know this unless you're interested in CS3233 Competitive Programming
  - Idea: Keep track of all possible minimums, with their index in a deque
  - Humongous brain algorithm

Q7 Application – Deque (self-read)

Intuition - Credit to Matthew

1. the rightmost guy should be a candidate at all times
2. the leftmost guy may or may not be a candidate at all times
3. if there are any candidates before the rightmost guy, they better be smaller than the rightmost guy
4. (generalize from 3) all candidates should be smaller than every candidates on the right, otherwise they should not be candidates

Hash Tables

Motivation

A motivating example

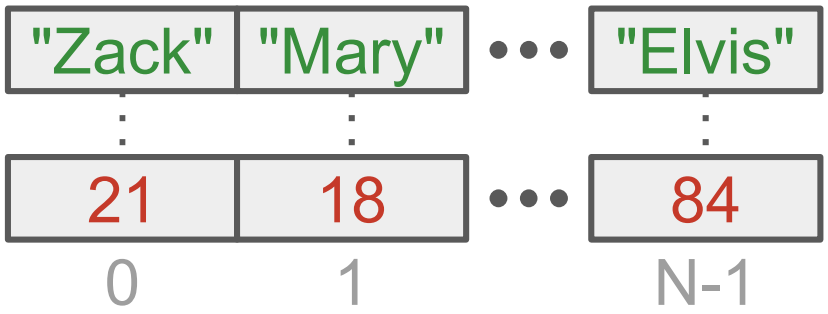
Suppose we have the following table (e.g. a spreadsheet) containing entries of people's name and age and we would like to capture it in a data structure which would support `search(name)`, `insert(name, age)` and `remove(name)` operations.

Name	Age
Zack	21
Mary	18
⋮	⋮
Elvis	84

Solution 1

Use two arrays

- One for storing `string` name
- One for storing `int` age
- The data entry for a person is represented by corresponding indexes in both arrays

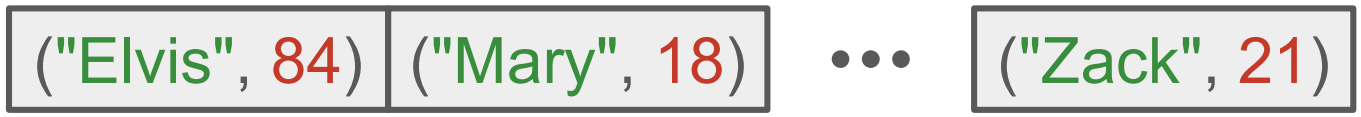


Solution 1 analysis

- `search(name)`: Linear search the entire array for names. *O(N)*
- `insert(name, age)`: Just append to the back of arrays. *O(1)*
- `remove(name)`: Find the name, then remove elements at that index from both arrays. *O(N)*

Solution 2

- We know that we can do better if the array is sorted!
- Sorting two arrays while maintaining index correspondences is hairy, so we shall just use an array of (name, age) pairs and sort it based on first element of pair, i.e. the name



Solution 2 analysis

- `search(name)`: Binary search! *O(log N)*
- `insert(name, age)`: Binary search lower bound then insert at rightful rank in the array. *O(N)*
- `remove(name)`: Binary search, then remove it from array. *O(N)*
- If given an array in the beginning, we pay an initial penalty of *O(N log N)* for sorting it

Can we do better?

Observations:

- It incurs *O(log N)* time to search — required by all 3 operations
- It incurs *O(N)* time to shift array elements — incurred by `insert` and `remove`

Key idea

Can we somehow find a magical function which **maps each possible name to a fixed and unique array index** in *O(1)*? i.e. allows us to achieve random access in the array!

Realize with this we also no longer need to shift elements in the array during `insert` and `remove` since other names must always be fixed to their respective indexes.

Key idea

We call such a magical function a *hash function*!

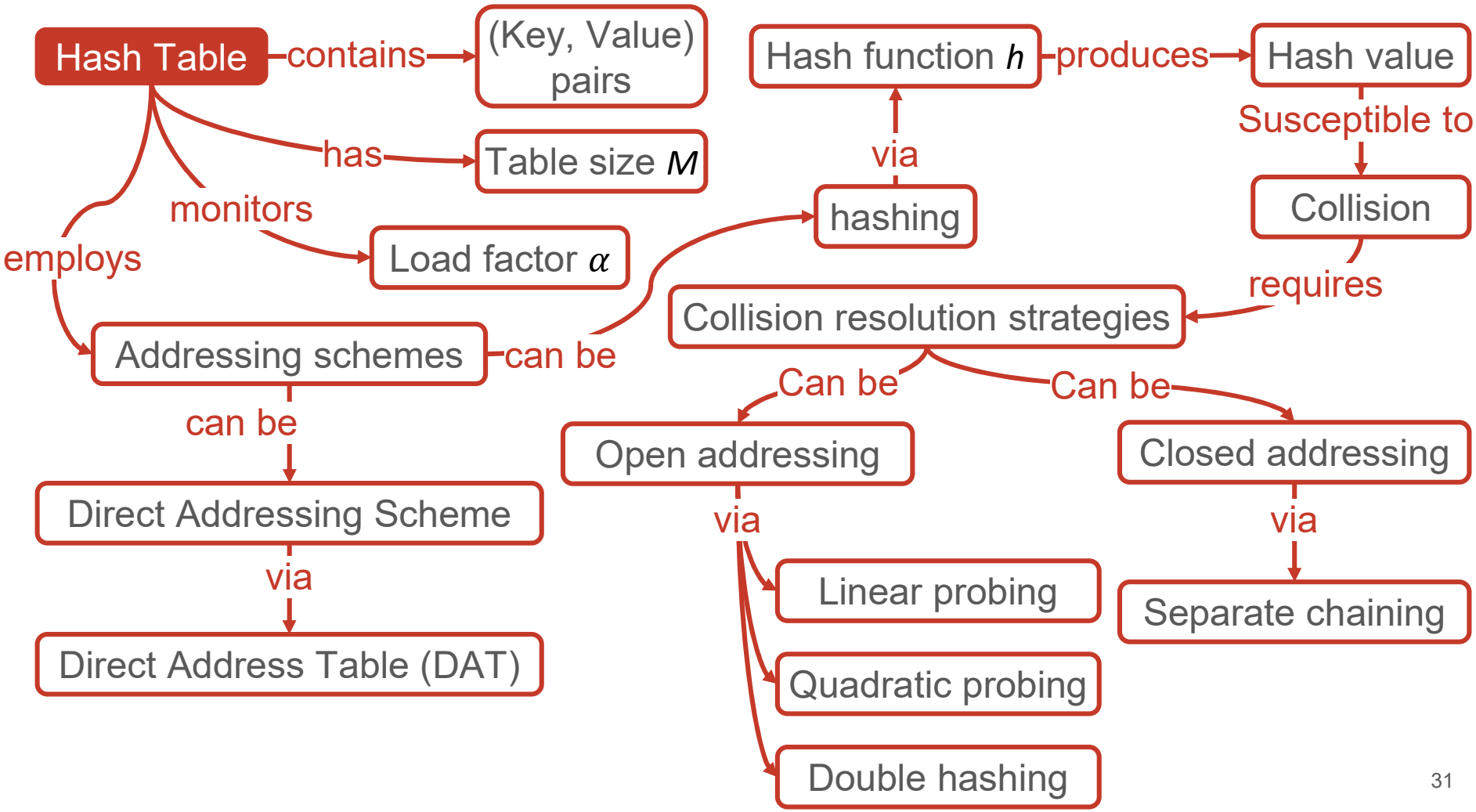
The data-structure it supports is called a *Hash Table*.

Did you know?

The term "hash" offers a natural analogy with its non-technical meaning (to "chop" or "make a mess" out of something), given how hash functions scramble their input data to derive their output. In his research for the precise origin of the term, Donald Knuth notes that, while Hans Peter Luhn of IBM appears to have been the first to use the concept of a hash function in a memo dated January 1953, the term itself would only appear in published literature in the late 1960s, on Herbert Hellerman's Digital Computer System Principles, even though it was already widespread jargon by then.

Source: Wikipedia

Conceptual re*hash*


Hash Table
contains
(Key, Value) pairs
has
Table size M
monitors
Load factor α
employs
Addressing schemes
can be
can be
Direct Addressing Scheme
via
Direct Address Table (DAT)
Hash function h
produces
Hash value
via
hashing
Susceptible to
Collision
requires
Collision resolution strategies
Can be
Can be
Open addressing
Closed addressing
via
via
Linear probing
Quadratic probing
Double hashing
Separate chaining

Other terminologies

- Hash Table is often also known as *Hashmap*, *Table*, *Map ADT* and *Dictionary*
- Hash value is also commonly referred to as *Hashcode*
- The act of calling hash function on a key is known as *hashing*
- Load factor is also referred to as *density*
- An index position in the Hash Table is commonly referred to as *table address*
- In closed addressing, a table address is sometimes also referred to as a *bucket* since it can contain multiple (key, value)

Table ADT

Common Table ADT operations

`search(v)`	Determine if `v` exists in the ADT or not. If it does return `true`, else return `false`.
`insert(v)`	Insert `v` into the ADT.
`remove(v)`	Remove `v` from the ADT.

We want a data structure implementation such that all three major Table ADT operations are done in *O(1)* on average

Hash function

A hash function *h(k)* is one that maps an input key *k* (of any type) to a table address within a range $[0 .. M-1]$ where *M* is the *table size*.

Example:

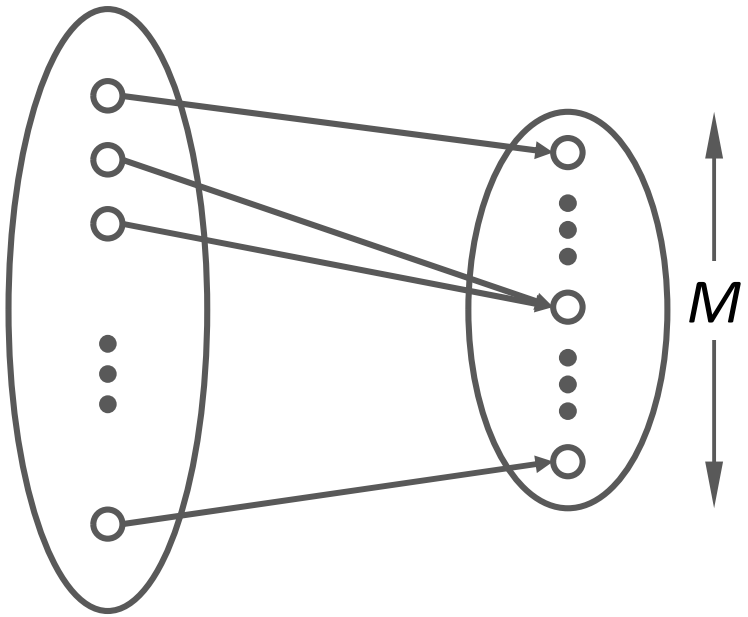
`h(2040)` → `6208`

`h("Hello")` → `39485`

Real life example: Postal Code

Hash function

$h: K \rightarrow \{0, \ldots, M-1\}$

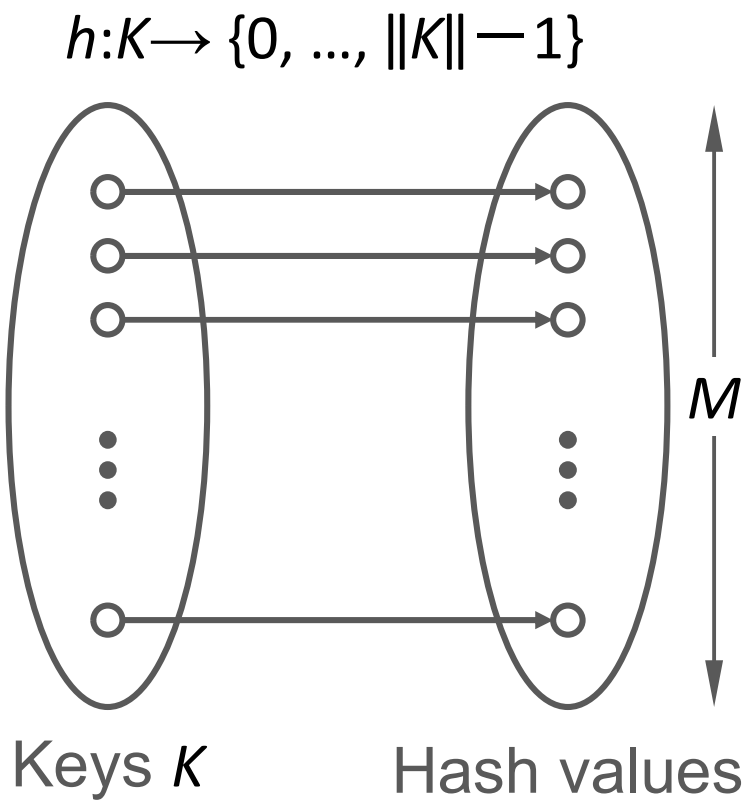


When we choose $M < \|K\|$, the hash function produces a many-to-one mapping.

Therefore collisions are inevitable **by design**.

Recall Pigeonhole Principle.

Hash function



A *perfect hash function* is one which achieves a one-to-one mapping (i.e. $M = \|K\|$), thereby preventing collisions. (See VisuAlgo Hashtable slide 4-4)

Realize this property is analogous to a Direct Address Table (DAT) which conceptually uses the identity function $f(k) = k$ in place of a hash function.

Hash function

If perfect hash function and DAT prevents collision from occurring, why don’t we always use them?

- Perfect hash functions can only be derived if all possible keys are known beforehand. This is rarely the case!
- DATs can be extremely wasteful in terms of space. E.g. For keys that are positive 32-bit integers, a DAT would require ~8GB of memory, most of which are probably never used! :O
- DATs are only achievable for keys that have finite ranges! In practice, many keys have almost infinite ranges! E.g. for keys of type `string`, `float`, `double` etc. Therefore it’s often inevitable that $M < \|K\|$

Good Hash Function

In general, hash functions need to optimize for two **competing objectives**:

1. **Minimising table size** M: The range of possible hash values
2. **Minimising collision**: When 2 different keys are hashed to same table address

Good Hash Function

Minimize table size *M* because

- Large range of hashed values means more *space/memory* required
- If *M* > range of keys, then many addresses in the table are 'wasted' because no key maps to them

Good Hash Function

Minimize hash value collisions because

- More hash collisions means more time required to determine if a key is in the table (main subroutine)
- Assuming constant table size, the most optimal distribution of keys (after hashing) over addresses is attained when collisions are minimized

Question 2 — Hash Table Basics

Finding a good hash function

A good hash function is essential for good Hash Table performance. It should be easy/efficient to compute and evenly distribute the possible keys.

For all the proposed hash functions on the following slides, comment on their flaw(s) (if any).

Question 2 Part 1

```
int index = (rand() * (key[0]-’A’)) % N;
```

No! How to match an input to its hash?

Problem: Non-deterministic hash. It’s not even a (math) function.

Question 2 Part 2

```
int index = (key[0]-’A’) % N;
```

Recall: ASCII is a numerical representation of characters.

E.g. `'a' = 97; 'A' = 65; '#' = 35;`

Only the first letter of the key is used. Any problems?

Matric number?

Problem: Non-uniform distribution of hash values.

Question 2 Part 3

```
int index = hash_function(key) % N;
```

`hash_function()` is standard string hashing formula (See VisuAlgo Hashtable 4-7)!

Base 26 computation:

$$key[0] \times 26^{N-1} + key[1] \times 26^{N-2} + \cdots + key[N-1]$$

Problem: Only limited to ['A', ... , 'Z']

Solution: Modify a bit.

Question 3 — Hash Table Basics II

Finding a good hash function

A good hash function is essential for good Hash Table performance. It should be easy/efficient to compute and evenly distribute the possible keys.

For all the proposed hash functions on the following slides, comment on their flaw(s) (if any).

Assume load factor α = number of items N / Hash Table size $M = 0.3$ (i.e. low enough)

Question 3 Part 1

Hash Table Size M: 100

Keys `k`: Positive even integer

Hash Function `h(k)`: `k % 100`

Question 3 Part 1

Problems:

- Odd slots will *never* be utilized! Wasted half of the capacity!
  - Better `h(k)`: `(k / 2) % M`
- *M* is also not a prime! Why do we want primes?
  - Better `h(k)`: `(k / 2) % 97`

Self-read

Why should table size *M* preferably be a prime?

Informal proof adapted from this blog post:

- Suppose your hash function is perfect and produces raw hashed values (before `% M`) which are uniformly distributed. i.e. If a key is supplied at random, then every hashed value in possible output range is equally likely
- The entire space of possible raw hash values will contain many groups of numbers $\{x, 2x, 3x, 4x, 5x, 6x, \ldots\}$ which are multiples of a number x. i.e. x is the greatest common factor (*GCF*) between all the numbers in the group
- After `% M`, all these raw hash values will collide over k buckets, where $k = M/GCF(M, x)$.* E.g. For $M = 22$, $x = 4$, after `% M`, multiples of x will fall into $22/2 = 11$ buckets: $\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$

*Proof in CLRS chapter 31,theroem 31.20

Self-read

Why should table size *M* be a prime? (cont'd)

- One way to prevent that is to choose a hashing scheme which does not generate raw hash codes that are multiples of one another but that would mean it's also a bad hash function which doesn't have a good distribution over its raw output range
- We observe what we really want is to force k to be equal to *M* so as to make each index in table a bucket! This is achieved by making *GCF(M, x)* equal to *1*! In other words, we force *M* to be *coprime* to x. Since x can be just about any number, we simply make sure that *M* is a large prime number!

Question 3 Part 2

Hash Table Size M: 100

Keys `k`: [0, 10000]

Hash Function `h(k)`: `floor(sqrt(key)) % 100`

Question 3 Part 2

Problem:

`sqrt` function is not a uniform distribution ⇒ hash function is not uniform either

Also, size is not a prime number.

Question 3 Part 3

Hash Table Size M: 101

Keys `k`: [0, 1000]

Hash Function `h(k)`: `floor(k * random) % 101`,

where 0.0 ≤ `random` ≤ 1.0

Note: `random` is floating point type so we have to `floor` the expression to ensure return value is an integer

Question 3 Part 3

Pros: Likely lead to optimal distribution of keys over table addresses

Cons:

- `random` will most likely be different every time for the same value of `k`. i.e. Non-deterministic hash values!
- We can insert a (key, value) quickly, but we cannot find it afterwards unless we linear search the table! :O

Hash Table in C++11

C++11 provides Hash Table data structure via `std::unordered_map`.

Built-in hash functions available for keys of primitive types

- Integer types (`int`/`long long`/`char`/`short`)
- Floating point types (`float`/`double`/`long double`)
- `string`

There is no need to custom define hash functions when our keys are of primitive types!

E.g. a Hash Table for `string` key, `int` value pairs is declared via:
`unordered_map<string, int>`

Self-read

A caveat

However, keys that are hashing floating point types is **not recommended.**

Why?

Floating point numbers are prone to **precision errors**.

A small difference will lead to a completely different hash value!

Eg: 2.0000000000000000 vs 2.0000000000000001

Questions?

PS2

- A: Just use std::list
- B: Just use 2x std::deque

/shiritori