CS2040C Tut 6

Midsem Debrief

Hash Function

Hash Function

Hash Function

Main Idea

A function that converts any key to an integer ID (usually within a range).

F(*key*) → *integer id* (between 0 .. **M**-1?)

F(“hello”) → *39485*

Real life example: Postal Code

Good Hash Function

Hash functions need to balance between two '*conflicting*' properties.

- Reduce **collision**
  - Similar to **scattering** keys **evenly**
- Reduce **range** of possible hash values

Of course, the same key must always hash to the same ID.

Good Hash Function

Reducing Collision (Why?)

More hash collision →

More *time* required to resolve

Good Hash Function

Reducing Range of Hashed Values (Why?)

Large range of hashed values →

More *space/memory* required to store them

Unused values in the middle of the range are ‘wasted’.

(Increases collision in other values as well)

Q1 Part 1

Hash Table Size:	**M** = 100
Keys:	Positive even integers, **k**
Hash Function:	**h(k) = k % M**

Q1 Part 1

Hash Table Size: **M** = 100

Keys: Positive even integers, **k**

Hash Function: **h(k) = k % M**

Odd slots will *never* be utilized. (Wasted space)

Better: h(k) = (**k / 2**) % M

Q1 Part 1

Hash Table Size: **M** = 100

Keys: Positive even integers, **k**

Hash Function: **h(k) = k % M**

M is also not prime.

Better: h(k) = (**k / 2**) % 101

Q1 Part 2

Hash Table Size: **M** = 1009

Keys: Valid email addresses

Hash Function:

ASCII sum of *last* **10** characters % **M**

ASCII is a numerical representation of characters.

Eg: ‘a’ = 97; ‘A’ = 65; ‘#’ = 35;

Q1 Part 2

Hash Table Size: **M** = 1009

Keys: Valid email addresses

Hash Function:

ASCII sum of *last* **10** characters % **M**

Suffix of emails are largely the same!

Eg: @u.nus.edu, @gmail.com

Q1 Part 2

Hash Table Size: **M** = 1009

Keys: Valid email addresses

Hash Function:

ASCII sum of *last* **10** characters % **M**

Many emails will hash to the same value.

Increased collision, hash can be better designed.

Q1 Part 3

Hash Table Size: **M** = 101

Keys: Integer **K** ∈ [0, 1000]

Hash Function: ⌊**K** * *random*⌋ % **M**

(where 0.0 ≤ *random* ≤ 1.0)

Yes, *random* is floating point type.

So I added a *floor* to make it clearer.

Q1 Part 3

Hash Table Size: **M** = 101

Keys: Integer **K** ∈ [0, 1000]

Hash Function: ⌊**K** * *random*⌋ % **M**

(where 0.0 ≤ *random* ≤ 1.0)

random might differ for the same value of **K**.

- Indeterministic.
- Same key will start at different positions.

Hashing in C++ 11

C++11 built-in hash function

C++11 has implementing hashing for

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

So you can do:

```
unordered_map<string, int>
```

(Self-read)

Hashing in C++ 11

C++11 built-in hash function

However, hashing floating point types is ***not recommended.***

Why?

Those of you that took CS2100...? :D

Hash Table Applications

Key-Value Mappings

Q2 Part 1

A *mini* population census is to be conducted on every person in your (*not so large*) neighbourhood.

We are only interested in storing every person's **name and age**.

→ **age** is *Integer*, **name** is *String*

Retrieve **age** by **name**.

Retrieve **name(s)** by **age**.

Q2 Part 1

To retrieve **age** by **name**:

<Key, Value> : <**name**, **age**>

h(**name**): Standard string hashing method

https://visualgo.net/en/hashtable?slide=4-7

Q2 Part 1

To retrieve **age** by **name**:

<Key, Value> : <**name**, **age**>

Collision resolution:

- Double Hashing
  - Unlikely to use up all the ‘slots’

Q2 Part 1

To retrieve **name(s)** by **age**:

<Key, Value> : <**age**, **name**>

h(**age**) = **age**

Direct Addressing Table (aka just-an-array)

Q2 Part 1

To retrieve **name(s)** by **age**:

<Key, Value> : <**age**, **name**>

Collision resolution:

- Separate Chaining
  - Already multiple values for the same *key*, no choice

Q2 Part 2

A *much larger* population census is also conducted across the country.

We are only interested in storing every person's **name and age**. → **age** is *Integer*, **name** is *String*

Retrieve **name(s)** of people who are **X** = 17 years or older.

Q2 Part 2

Retrieve **name(s)** of people who are **X** = 17 years or older:

- Age is likely to be from [0, 120]
- We can still use Direct Addressing Table
  - With separate chaining (like Q2 P1)
- Loop through all possible ages ≥ **X**

Q2 Part 3

A *different* population census is conducted across the country.

We are only interested in storing every person's **name and age**. → **age** is *Integer*, **name** is *String*

However, we now want to retrieve **name and age** of people with a given **last name**.

Q2 Part 3

To retrieve **name(s)** by **last name**:

<Key, Value> : <**last name**, **person**>

Person = *pair* of (**name**, **age**)

Collision resolution:

- Double Hashing (unlikely to have *that many* different last names)
- Separate Chaining

(Self-read)

Q2 Part 4

A grades management program stores a student's **index number** and his/her **final marks** in a module. There are 1,000,000 students, each scoring final marks in [0.0, 100.0].

Store **all** the student's performance.

Print the list of students in **ranking order** that are *more than 65.5.*

(Self-read)

Q2 Part 4

Not really...

If

<Key, Value>: <**index number**, **grades**> ...

Then we cannot get list of students in ranking order, without looping through everything, and then sort.

(Self-read)

Q2 Part 4

Not really...

If <Key, Value>: <**grades, index number**> ...

To deal with issues with *floating points,* we can round grades to a certain precision (3 d.p.) and converting them to integer. Eg: 98.234 → 98234

Iterating through all possible scores > 65.5 is still quite a lot.

Hash Table Discussion

Table ADT

Perfect Hash Function

Best Collision Resolution

List ADT vs Table ADT

- Relative **order** is important to a List ADT.
  - Not so for Table ADT.
- **Mapping** between index to value is important to Table ADT.
  - Not quite for List ADT.
  - Inserting in middle position will change the mapping for the remaining list.

PS3

Scheduling ~~Deliveries~~ Deletes

PS3C

Approach

Can you solve the question if ...

1. Instead of woman names, we label them with an integer ID
2. The integer ID is their order of arrival (1, 2, 3... etc)

PS3C

Approach

If you can solve with the constraints above...

Can you create these *constraints* by yourself?

>> How about I use a **Hash Table**?

PS3C

Approach

A hash table can map woman's name to our assigned integer ID.

We can keep a counter to assign the IDs in the order of their arrival.

- Then update the hash table

PS3C

Implementation

As much as possible, use the *integer ID* in other parts of your code instead of the *woman's name*.

Passing integers is *much* faster than passing strings.

PS3C

Question

How do I remove *any* element from a binary heap?

Be lazy. Do it later.

Or... use a **Balanced Binary Search Tree (STL Set)** :P

PS3C

Question

I can insert and delete values from a priority queue.

How do I update a value in a priority queue?

1. Delete the old value
2. Insert the updated value

:O

Questions?