CS1020E: DATA STRUCTURES AND ALGORITHMS I

Tutorial 10 – Hashing

(Week 12, starting 31 October 2016)

1. Simulation

In this question we will simulate the operations insert(key) and erase(key) on a `std::unordered_set`, denoted by the shorthand `I(k)` and `D(k)` respectively. Note that a **Hash Table**, e.g. `unordered_map`, works on a **<Key, Value> pair**, while in a **Hash Set**, or `unordered_set`, the **value is the key** itself.

Fill the contents of the hash table after each insert / delete operation.
The Hash Table has "table size" of 5, i.e. 5 buckets. The hash function is **h**`(key) = key % 5`

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

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				

Use double hashing as the collision resolution technique, $\mathbf{h_2}$`(key) = key % 3`:

	0	**1**	**2**	**3**	**4**
I(7)			**7**		
I(22)			7	**22**	
I(12)	Infinite loop on collision resolution as $h_2(12) == 0$				

Use double hashing as the collision resolution technique, $\mathbf{h_2}$`(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 on collision resolution as $h_2(2)$ % m == 0				

2. *Hashing or No Hashing*

Hash Table is an ADT that allows for find, insert, and delete operations in O(1) average-case time, if properly designed. However, it is not without its limitations. For each of the cases described below, state if Hash Table can be used.

In each case, if it is possible to use Hash Table, describe its design, including:

1) The <Key, Value> pair
2) Hashing function/algorithm
3) Collision resolution (including second hash function/algorithm for double hashing)

Otherwise, why is Hash Table not suitable for that particular case?

(a) A bank has many bank accounts. Each account has name, account number, and current balance. The operations to perform are: retrieve balance, deposit money, and withdraw money. For each operation, either the name or account number will be given. The amount will also be given for the last two operations.

(b) A mini-population census is to be conducted on every person in your neighbourhood. We are only interested in storing every person's name and age. The operations to perform are: retrieve age by name, and retrieve names by age.

(c) A larger population census is also conducted across the country, similarly containing only every person's name and age. The operation to perform is: retrieve names of people eligible for voting. Only people above a certain age are eligible for voting. However, we still need to store the rest of the data.

(d) A different population census similarly contains only the name and age of every person. The operation to perform is: retrieve people with a given last name.

(e) In a scientific experiment, data on energy converted for different runs is collected. Each run has a different velocity and distance, both floating point numbers. The operation to perform is: retrieve the energy value for a given velocity and distance. [Hint: What would happen when you hash a floating-point number?]

(f) A grades management program stores a student's index number and his/her final marks in one GCE 'O' Level subject. There are ~1,000,000 students, each scoring final marks in [0.0, 100.0]. The operation to perform is: retrieve a list of students who passed. A student passes if the final marks are more than 65.5. Whether a student passes or not, we still need to store all students' performance.

Possible Answers

(a) Yes. 2 Hash Tables for efficient lookup using either key

<Key, Value> Pair: <name, BankAccount*> and <account number, BankAccount*>

Hashing Algorithm: string hashing algorithm `f(key, charWeight, tableSize)`, that takes into account every character weighted by its position, modulo the size of the hash table

Collision Resolution: Double hashing using h_2`(name)`$=1+f_2$`(name, weight, m)`, where f_2 has a different character weighting from f, and the modulo `m` used is smaller than table size

We are here assuming that the table size is a prime number, so that every h_2(name) is co-prime to it.

(b) Yes, using 2 Hash Tables for efficient lookup both ways.

<Key, Value> Pair: <name, age> and <age, name**s**>, to find age and names respectively
Hashing Algorithm: h(name) = standard string hashing, and, h(age) = age (direct addressing)

Collision Resolution:
<name, age> Hash Table – Double hashing using h_2`(name)` as in part (a)
<age, name> Hash Table – Separate chaining for names with the same age

(c) Yes, **direct addressing** table as in part (b)'s <age, name**s**> Hash Table. The operation can be performed by iterating through age, traversing the chain/list for each age, starting from the minimum eligible age.

(d) Yes, if we can extract the last name from a name.

<Key, Value> Pair: <last name, pair<full name, age>**s**>
Hashing Algorithm: h(name) = standard string hashing
Collision Resolution: Separate chaining for people with the same last name

(e) Not a good choice, as floating-point numbers are an **approximation** of real numbers. For example, 0.1 cannot be represented as a binary floating point number. It would be difficult to design a good hash function that evenly distributes all keys in general.

However, if we allow precision of up to a number of binary digits, then this may be possible. We can then map a key to a particular address, and find a value that is approximately equal to that key. We will probably have to write our own hash function, as a function of both numbers.

(f) Depends on **precision**. If the precision is fixed, say 2 decimal places, the number can be converted to an integer, and stored as in (c).

3. Hash Functions

A good hash function is essential for good hash table performance. A good hash function is easy/efficient to compute and will evenly distribute the possible keys. Comment on the flaw (if any) of the following hash functions. Assume the load factor α = number of keys / table size = 0.3 for all the following cases:

(a) The hash table has size 100. The keys are positive even integers. The hash function is

h(*key*) = *key* % 100

(b) The hash table has size 49. The keys are positive integers. The hash function is

h(*key*) = (*key* * 7) % 49

(c) The hash table has size 100. The keys are non-negative integers in the range of [0, 10000]. The hash function is

h(*key*) = $\lfloor \sqrt{key} \rfloor$ **% 100**

(d) The hash table has size 1009. The 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 for ASCII values

(e) The hash table has size 101. The keys are integers in the range of [0, 1000]. The hash function is

h(*key*) = $\lfloor$ ***key* * *random***$\rfloor$ **% 101**, where **0.0 ≤ *random* ≤ 1.0**

Answer

(a) No key will be hashed directly to **odd-numbered slots** in the table, resulting in wasted space, and a higher number of collisions in the remaining slots than necessary.

Aside from the hash function, the 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.

(b) All keys will be hashed only into slots 0, 7, 14, 21, 28, 35 and 42. Also, as in (a), the hash table size is not a prime number.

(c) Keys are **not uniformly** distributed. Many more keys are mapped to the larger-indexed slots. Also, as in (a), the hash table size is not a prime number.

(d) 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.

(e) 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**.

4. Searching for a Key

You have a group of non-empty strings, with each string having an integer ID associated with it. The strings are stored in `unordered_map<int, string> map`. Which of these 3 ways successfully finds whether an element with key 3 exists in `map`?

Read the documentation and try it out. Remember to compile using the C++11 standard.

```
bool found = (map.find(3) != map.end());
```

```
bool found = (map.at(3) != "");
```

```
bool found = (map[3] != "");
```

How about doing the same in an `unordered_set<int>`?

Answer

If we do not know whether an element with a given key exists, we should only use `find()`. `find()` returns an iterator 'pointing' to the key-value pair if found, or an iterator equivalent to `end()` if no such key is found.

When no such key exists:

- operator[] creates a new element, "" in this case, adding an additional element to the map
- at() throws an Exception, a runtime error that may stop the program

Both `unordered_map<K, V>` and `unordered_set<K>` have `insert()`, `erase()`, and `find()`. `unordered_set<K>`, however, does not have `at()` or `operator[]`.

Image from Brandon Duncombe: https://twitter.com/BrandonDuncombe