CS2040 Tutorial 6 Suggested Solution

Week 8, starting 3 Oct 2022

Q1 Simulation

In this question we will simulate the operations add(key) and remove(key) on a hash set, denoted by the shorthand `I(k)` and `D(k)` respectively. Note that a **hash map** works on a **<Key, Value> pair**, while in a **hash set**, the **value is the key** itself

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

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	$\cancel{12}$	**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, **h₂**`(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, **h₂**`(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				

Q2 Hash Functions

A good hash function is essential for good hash table performance. A good hash function is easy/efficient to compute and attempts to 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**

Q3 The Price is Right

A large takeaway food chain has expanded its menu greatly during the pandemic. Their menu contains 4 categories of food: Appetizers, Soups, Mains and Desserts. Each category of food contains **N** items, each having a name and a price in cents. High-end food is sold too, hence the price can be quite large

Given a target amount **k** (in cents), find just one possible selection of an (Appetizer, Soup, Main, Dessert) that costs exactly **k** cents, if exists. What is the time complexity of a brute force algorithm that solves this problem?

Next, design an efficient O(**N**^{2}) algorithm to solve this problem

Answer

A brute force algorithm that does not depend on the target amount **k**, tries all **N**^{4} possible selections (4 nested loops), one from each category. Upon finding one match, return that selection. This runs in O(**N**^{4})

```
select(target, apps, soups, mains, dessts) {
   for app in apps
      for soup in soups
         for mn in mains
            for dess in dessts
               if app.price + soup.price + mn.price + dess.price == target
                  return (app, soup, mn, dess)
}
```

Instead of brute forcing across the 4 categories, we can just brute force each of the 2 categories and find if there is a corresponding pair in O(1) average time having the desired remaining amount using a hash map of pair price to (left item, right item) pair value

For example, if an (appetizer1, soup1) costs 900 cents and the target is 1000 cents, then we just need to find whether there is a (main, dessert) in the hash map that costs a subtotal of 100 cents

```
select(target, apps, soups, mains, dessts) {
   new hash map priceToMDs
   for mn in mains
      for dess in dessts
         priceToMDs.put(target – mn.price – dess.price, (mn, dess))
   for app in apps
      for soup in soups
         pair = priceToMDs.get(app.price + soup.price)
         if pair != null
            return (app, soup, mn, dess)
   return null
}
```

By brute forcing only 2 categories at a time, the time complexity of this algorithm is O(**N**^{2})

Note: Only 1 match is required to be found, so this algorithm just overwrites the existing (main, dessert) value pair when there is a duplicate key

Question 4 (Online Discussion) – Equal Lists

You are given a **N** x **K** 2D-array of 32-bit integers. Each inner array represents a list of **K** elements. The **N** lists contain distinct sequences. You may perform some pre-processing in O(**NK**) time

After that, you are supposed to run queries. Each query is supposed to determine if a given list of another **K** 32-bit integers is equal to any of the **N** initial lists (the sequence of all **K** elements in both lists are equal). If there is a match, output the index of the match

A query should have a very high probability of running in O(**K**) time, while very rarely running in > O(**K**) time

Hashing can be used, if there are very few (O(1) number of) collisions for a key then we can get O(**K**) time for that key. How to hash each list of **K** elements such that there is a very low probability of collisions?