CS2040: Data Structures and Algorithms

Tutorial Problems for Week 6: Hashing

For: 19 September 2024, Tutorial 4

Solution: Secret! Shhhh... This is the solutions sheet.

Problem 1. Simulation?

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

Fill the contents of the hash table after each `add` or `remove` 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, `g(key) = key % 3`:

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

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`	

Thus for double hashing not only must the secondary hash function not evaluate to 0, all hash values generated must also be co-prime with m the size of the hash table (used in the primary hash function). In order to achieve this use a prime number m' $<$ m for the secondary hash function.

Problem 2. 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 $\alpha = \frac{\text{number of keys}}{\text{table size}} = 0.3$ for all the following cases.

a) The hash table has size 100 with positive even integer keys. The hash function is `h(key) = key % 100`.

b) The hash table has size 49 with positive integer keys. The hash function is `h(key) = (key*7) % 49`.

c) The hash table has size 100 with non-negative integer keys in the range [0, 10000]. The hash function is `h(key) =` $\lfloor\sqrt{\texttt{key}}\rfloor$ `% 100`.

d) 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` for ASCII values.

e) The hash table has size 101 with integer keys in the range of [0, 1000]. The hash function is `h(key) =` $\lfloor$ `key` $*$ `random` $\rfloor$ % 101, where `0.0` $\leq$ `random` $\leq$ `1.0`.

f) 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
}
```

Solution:

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**.

f) This is not a good hash function. This is 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.

Problem 3. String Matching

Text search is a problem where given a long string, the text, you are to find a list of k-letter words hidden in the text, where k is given. For example, in the text "thequickbrownfoxjumpsoverthelazydog", it contains the 5-letter words ("quick", "overt") within it.

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.

Solution: Since k is given, by going through all possible $(n-k+1)$ k-letter words in the text of length n, we can build a Hash Table containing all the possible k-letter words as keys, with their corresponding values being the frequency of appearance in the text. 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 $O(kn)$.

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 $O(kn^2)$. See `Matching.java` for a sample solution.

Note that while working with immutable values in the Hash Table, such as Integer and String, we usually read from the Hash Table, compute the updated value, then put it back into the Hash Table. For mutable elements, we may change the object used as value, but we should NOT change the object used as the key, as this will mess up the Hash Table.

Problem 4. Finding Sum

You are dining at the hottest new restaurant in town: Algorithms Cafe. As you walk in the door, a bell goes off, and the owner of the restaurant comes out to announce that you are the 2147483647^{th} customer and have won a special deal.

Their menu consists of four components: Appetizers, Soups, Mains, and Desserts. Each component of the menu contains a long and daunting list of n items. If you can choose one item from each section that add up to 100 dollars, then you can eat for free! Come up with the most efficient algorithm to solve this problem, and state the time complexity.

Solution: We denote the four lists in the menu as m_1, m_2, m_3, m_4. For each pair of values (x, y) on lists (m_1, m_2), 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 (m_3, m_4), 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.

This is a slight modification to the famous 3SUM problem, and it is conjectured (but as of yet unproved, in general models of computation) that you cannot do much better than $\Theta(n^2)$. (There exists recent work showing that you can do just a little bit better.) Take a look at `https://en.wikipedia.org/wiki/3SUM`.