CS1102: Data Structure and Algorithms

Tutorial 10 - Hashing (solutions)

Week of 5 April 2010

1. A good hash function is essential for good hash table performance. A good hash function is easy to compute and will evenly distribute the possible keys. Comment on the performance of the following hash functions. Assume the load factor α = number of keys / table size = 0.7 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 positive integers in the range of [0,10000]. The hash function is:
      h (key) = floor (sqrt (key)) % 100

   d) The hash table has size 2003. The keys are English words. The hash function is:
      h (key) = (sum of positions in alphabet of key's letters) % 2003
      E.g.: letter 'a' has position 1 and letter 'z' has position 26.

   e) The hash table has size 1009. The keys are *valid* email addresses. The hash function is: (see: http://www.asciitable.com/ for ASCII values)
      h (key) = (sum of ASCII values of last 10 characters) % 1009

   f) The hash table has size 101. The keys are integers in the range of [0,1000]. The hash function is:
      *h (key) = floor (key * random)% 101 , where* $0.0 \leqslant random \leqslant 1.0$

   Answer:

   a) Keys will not be evenly distributed. No key will be hashed directly to odd-numbered slots in the table. The hash table size is also not good and we should use a prime number as the size.

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

   c) Keys are not evenly distributed. Note that h ($1 \leqslant$ key $\leqslant 3$) = 1 i.e. 3 keys are hashed to slot 1, h ($4 \leqslant$ key $\leqslant 8$) = 2 i.e. 5 keys are hashed to slot 2, h ($9 \leqslant$ key $\leqslant 15$) = 3 i.e. 7 keys are hashed to slot 3, and so on. The difference between successive square numbers increases as the numbers get larger, leading to non-uniform distribution of keys into each slot. Also, the hash table size is not a prime number.

d) Most English words are short (20 letters or less), so most of hash values of the keys will be less than 20 * 26 = 520, which would cause the remaining ~ 1400 slots to be empty (non uniform distribution of key). Furthermore, words with the same letters will be hashed to the same value, e.g. h(“post”) = h(“stop”) = h(“spot”).

e) Keys are not evenly distributed because many email addresses have the same domain names, and they will all be hashed to the same value, e.g. “hotmail.com”.

f) This hash function does not work because we usually cannot reproduce the random value to retrieve the element once it is inserted into the hash table.

CS1102: Data Structure and Algorithms

2. Suppose that a hash table with size m = 13 and the following keys are to be mapped into the table in the sequence given:

```
10  100  32  45  58  126  3  29  200  400  0
```

a) Show the final hash table after all the values are hashed using simple hash function key % m. Use linear probing to resolve collisions.

Answer:

```
Multiples of 13: 13, 26, 39, 52, 65, 78, 91, 104, 117
Keys | Keys % 13 (quick calculation without using calculator)
----------------
10   | 10
100  | 100 – 91 = 9 → 91 is the nearest multiples of 13 below 100
32   | 32 – 26 = 6
45   | 45 – 39 = 6 (collide with 32, inserted at index 7)
58   | 58 – 52 = 6 (collide with 32 and 45, inserted at index 8)
126  | 126 – 117 = 9 (collide with 100 and 10, inserted at index 11)
3    | 3
29   | 29 – 26 = 3 (collide with 3, inserted at index 4)
200  | 200 – 117 = 83 – 78 = 5
400  | 400 – 117 – 117 – 117 = 49 – 39 = 10
       (collide with 10 and 126, inserted at index 12)
0    | 0

index |  0|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10| 11| 12|
-----------------------------------------------------------
key   |  0|   |   |  3| 29|200| 32| 45| 58|100| 10|126|400|
```

b) Show the final hash table after all the values are hashed using the following hash function and use quadratic probing to resolve collisions.

```
int hashFunction(int key, int m) {
  int sum = digitSum(key); // digitSum add all digits of key
  return sum % m;          // e.g. key = 126, sum = 9
}
```

Answer:

```
Multiples of 13: 13, 26, 39, 52, 65, 78, 91, 104, 117
Keys | digitSum(key) | hashFunction(key, 13)
-------------------------------------------
10   | 1             | 1
100  | 1             | 1 (collide with 10, inserted at index 2)
32   | 5             | 5
45   | 9             | 9
58   | 13            | 0
126  | 9             | 9 (collide with 45, inserted at index 10)
3    | 3             | 3
29   | 11            | 11
200  | 2             | 2 (collide with 100 and 3,
                       inserted at index 2+4=6)
```

```
400   | 4               | 4
0     | 0               | 0 (collide with 58, 10, 400, 45, and 3,
                          inserted at index (0+25)%13=12

Answer:
index |  0|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10| 11| 12|
-----------------------------------------------------------
key   | 58| 10|100|  3|400| 32|200|   |   | 45|126| 29|  0|
```

You may want to think about the below question:

What are the average numbers of probes for successful key search for both (a) and (b)?

3. Binary Search versus Hash Table

Suppose that 511 distinct integers are arranged in ascending order in an array named **values**. Suppose also that the following code is used in a method to locate an integer named **key** in the array.

```
int leftIndex = 0;
int rightIndex = values.length() - 1;
while (leftIndex <= rightIndex) {
        int middleIndex = (leftIndex+rightIndex)/2;
        if (values[middleIndex] == key) {
                return true;
        } else if (values[middleIndex] < key) {
                leftIndex = middleIndex+1;
        } else {
                rightIndex = middleIndex-1;
        }
}
return false;
```

a. Compute the total number of comparisons necessary to locate <u>all</u> 511 values in the array using the above loop. Keep in mind that a comparison occurs in one of the two lines

```
if (values[middleIndex] == key) { ...
} else if (values[middleIndex] < key) { ...
```

You are NOT allowed to reuse the result from other searches; you can only perform each search separately.

b. Now suppose that the 511 distinct integers are already stored in a hash table with size p, where p is a prime. The hash function is *h(key) = key % p*. We use separate chaining to resolve conflicts. We assume all the 511 integers are uniformly distributed, and there are either ceiling(511/p) or floor(511/p) nodes in each chain. Find out the minimal p such that the number of comparisons necessary to access <u>all</u> 511 values in this hash table is smaller than or equal to the result of 3.a. Keep in mind that even though the first node in the chain is the value we are searching for, we still need a comparison.

You are NOT allowed to reuse the result from other searches; you can only perform each search separately.

c. One of the problems with separate chaining is that when there might be some very long chains, which will slow down the search, because we can only compare the value sequentially. One suggestion is instead of using linked list as a chain, we use a sorted array, i.e. each entry in the hash table contains a reference to an sorted array. Arrays may have different sizes. Each array is sorted such that we can now apply binary search in 3.a. on it to speed up the search. We can use the same hash function as above and the same p as the result of 3.b., but adopting this sorted array instead of linked list to perform separate chaining. Suppose the 511 integers are already stored in the new hash table. Is this a good solution in this case? Is this a good solution for general purpose? Explain the reasons.

ANSWER

a.
Search for values[255], the middle element, requires one comparison; search for values[127] or values[383], the middle elements of the first 255 and last 255 elements respectively, requires three comparisons; four keys require five comparisons; and so on. Total comparisons to find all 511 keys is
1*1 + 2*3 + 4*5 + 8*7 + 16*9 + 32*11 + 64*13 + 128*15 + 256*17 = **7683**.

b.
First of all, the big trend is clear: the larger p is, the smaller number of comparisons needed. Thus, we can find the result p by starting from 2, and do the calculation for each prime in increasing order. However, this is not efficient.

So let's approximate p first. If we can locate the p that produces very close number of comparisons to 7683, then the result p should not be far away. We use C(p) to denote the number of comparisons we need to search for all 511 integers when a certain p is chosen.

Notice that the lengths of all p chain should be quite even, either floor(511/p) or ceiling(511/p). In each chains, we need 1 comparison for the first integer, 2 for the second, …, so we have approximately

$$1+2+\ldots+c = (c+1)*c/2 \approx (511/p+1)*511/p/2$$

number of comparisons for each chain, where c is the length of this chain. Thus, altogether we have about

$$C(p) \approx p * (511/p+1)*511/p/2 = (511/p+1)*511/2$$

comparisons.

We want (511/p+1)*511/2 <= 7683, thus p>=17.58. Good, we choose **19**.

For the sake of completeness, some calculations are shown below (You may omit it).

Let's try **p=19**. We have 19 chains. Notice that 511=19*26+17, which means 17 chains contain 27 entries, while other 2 chains contain 26 entries. Thus, the exact number of comparison is

C(19)=17*(27+1)*27/2+2*(26+1)*26/2=7128<7683.

Good. Now we hope that for 17, the largest prime below 19, we can show C(17)>7683, because this will ensure that 19 is the smallest prime such that C(p)<=7683. Let's try

Let's try **p=17.** We have 17 chains. Notice that 511=30*17+1, which means 1 chain contains 31 entries, while other 16 chains contain 30 entries. Thus, the exact number of comparison is

C(17)= 1*(31+1)*31/2+16*(30+1)*30/2=7936>7683.

Thus, the answer is 19.

c.
It is a good solution in this particular case, because we only care about search, and fewer comparisons are needed for all integers. We know the size of each chain beforehand, so we can allocate the arrays to fit the length of the chain exactly. No space wasted. However, it's not a solid solution in general. The main drawbacks include, but not limited to:

- Overhead maintaining the sorted order
  In this particular case, we assume the data is already in the hash table. But in real life, maintaining a sorted order for every newly added or modified entry is time-consuming. Using normal separate chaining, it costs only O(1) time to insert a new entry or modify an existing entry. But now, we need to perform an insertion either way, which comes with O(c) complexity, where c is the length of the chain it belongs to.

- Static array is not a wise choice when the input size is unknown
  In this particular case, we know exactly how long each chain, as an array, will be. So we can allocate the size beforehand without a problem. But in real life, we don't know the size of the array we will need. So either we allocate very big arrays and waste some space, or, we allocate normal-sized arrays and waste some time when some array is full, because we have to create a double-sized array and copy everything to that array.