ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2019 Prof. R. Wattenhofer

Computer Engineering II

Solution to Exercise Sheet Chapter 8

Quiz

1 Quiz

a) The correct answer is iii): The number of collisions goes up.
Some buckets will have fewer than $\frac{1}{m}$ keys in them on average, and some significantly more. Since the number of collisions is quadratic in the number of keys in a bucket, this means that we get more collisions than if the hash function wasn't biased.

For a specific example, consider the case of 2 buckets, one with x keys in it and the other with y keys. If we increase x and decrease y, what happens to the number of collisions, which is $\binom{x}{2} + \binom{y}{2}$? We show what happens to the proportional term $x^2 + y^2$.

$$(x+d)^2 + (y-d)^2 = x^2 + 2dx + d^2 + y^2 - 2dy + d^2 = x^2 + y^2 + 2d(x-y) + d^2$$

If initially, $x = y$ — which is the case in expectation if we distribute keys evenly among the buckets — then we see that moving d keys from one bucket to the other results in

$$(x+d)^2 + (y-d)^2 = x^2 + y^2 + 2d(x-y) + d^2 = x^2 + y^2 + d^2 > x^2 + y^2$$

b) We only need to consider:

i) Number of keys
iii) Size of hash table
v) Method for resolving collisions

If we insert many keys into a fixed size hash table, then we get more collisions and thus need to do more work to resolve those collisions than if we only insert few keys. Analogously, if we insert a fixed number of keys into a small hash table, then we get more collisions than if we insert them into a large hash table. Finally, the method of resolving collisions makes a difference, as can be seen for example in Table 8.18 in the script.

Altogether, we need to consider the number of keys, the size of the hash table, and the method we use for resolving collisions. More succinctly, it is the ratio between number of keys and size of the table that is relevant, and this is the load factor.

The genius of universal hashing is precisely that we do not need to consider the distribution of keys; we know that in expectation, we get a good hash function in few tries no matter what the distribution of keys looks like.

As for similarities between keys, some applications require "similar" keys to be close to each other in the hash table, and there are techniques to handle this. In general, this is not a requirement we need to consider.

c) If every single operation has to be fast, hashing is a bad choice; in the worst case, a single search operation can take linear time if we have to look at every bucket for hashing with probing, or if all keys are in one bucket for hashing with chaining. The guarantees we get from hashing are in expectation — at least one of insert/delete/search can only be fast in expectation and will cost more than constant time in the worst case.

Basic

2 Trying out hashing

Solution to our exercise:

	0	1	2	3	4	5	6	7	8	9	10
Linear	22	88			4	15	28	17	59	31	10
Quadratic	22		88	17	4		28	59	15	31	10
Double hashing	22		59	17	4	15	28	88		31	10

To give an example of how to arrive at this, we show how the insert operations go with linear probing:

$h_0(10) = 10 + 1 \cdot 0 \mod 11 = 10$

$h_0(22) = 22 + 1 \cdot 0 \mod 11 = 0$

$h_0(31) = 31 + 1 \cdot 0 \mod 11 = 9$

$h_0(4) = 4 + 1 \cdot 0 \mod 11 = 4$

$h_0(15) = 15 + 1 \cdot 0 \mod 11 = 4$
$h_1(15) = 15 + 1 \cdot 1 \mod 11 = 5$

$h_0(28) = 28 + 1 \cdot 0 \mod 11 = 6$

$h_0(17) = 17 + 1 \cdot 0 \mod 11 = 6$
$h_1(17) = 17 + 1 \cdot 1 \mod 11 = 7$

$h_0(88) = 88 + 1 \cdot 0 \mod 11 = 0$
$h_1(88) = 88 + 1 \cdot 1 \mod 11 = 1$

$h_0(59) = 59 + 1 \cdot 0 \mod 11 = 4$
$h_1(59) = 59 + 1 \cdot 1 \mod 11 = 5$
$h_2(59) = 59 + 1 \cdot 2 \mod 11 = 6$
$h_3(59) = 59 + 1 \cdot 3 \mod 11 = 7$
$h_4(59) = 59 + 1 \cdot 4 \mod 11 = 8$

3 Using hash tables

a) Build a hash table M from T. For each key $k \in S$, search whether the key is in the hash table M. If every search says "yes, that's here", answer "yes". Else, answer "no" after the first search that came back "not in the set".

b) Our cost is the time for building plus the time for searching.
Since the input is static, we use perfect static hashing, and we get expected cost in the order of $\mathcal{O}(q + r)$: building the table costs expected time linear in $|T| = r$, and searching all keys in S in the table costs (worst case for perfect static hashing) $|S| = q$ time.

The time complexity of a simple algorithm that sorts and compares the elements in the two sets is $\mathcal{O}(q \log q + r \log r)$, which asymptotically worse than the time complexity of the algorithm that uses hash tables ($\mathcal{O}(q + r)$).

4 r-independent hashing

The difference between universal hashing and r-independent hashing is this: with universal hashing, if we fix any two keys and sample a hash function from a universal family, then the chance of the two keys colliding under that hash function is at most $\frac{1}{m}$. r-independent hashing is not defined via collisions, but via the possible combinations of buckets into which a random hash function will put any r fixed keys, and the statement here is: they are equally likely to be put into any bucket combination from $\langle 0, \ldots, 0\rangle$ to $\langle m-1, \ldots, m-1\rangle$, i.e. for each of those combinations, the chance of getting those hashes is $\frac{1}{m^r}$. The purpose of this exercise is to show that r-independence is a strictly stronger property than universality.

a) Let $\mathcal{H}$ be 2-independent. By the definition of 2-independence, for any two distinct keys $k \neq l$ we have $\Pr[h(k) = a_1 \text{ and } h(l) = a_2] = \frac{1}{m^2}$ for any $a_1, a_2 \in M$ if we sample $h \in \mathcal{H}$ uniformly. Therefore:

$$\begin{aligned}\Pr[h(k) = h(l)] &= \sum_{c=0}^{m-1} \Pr[h(k) = h(l) \text{ and } h(k) = c] \\ &= \sum_{c=0}^{m-1} \Pr[h(k) = c \text{ and } h(l) = c] \\ &= \sum_{c=0}^{m-1} \frac{1}{m^2} = m \cdot \frac{1}{m^2} = \frac{1}{m}\end{aligned}$$

Therefore, if h is 2-independent, then h is universal.

An alternative proof: we know that $\Pr[h(k) = a_1 \text{ and } h(l) = a_2] = \frac{1}{m^2}$ for any $a_1, a_2 \in M$. There are exactly m possible vectors $\langle a, a\rangle \in M^2$ that constitute all possible collisions, and since each of them has probability $\frac{1}{m^2}$, we get a total collision probability of $\frac{m}{m^2} = \frac{1}{m}$.

b) Let $k = (0, \ldots, 0)$ and $l \in M^s, k \neq l$ arbitrary. Since $h_a(k) = 0$ for all choices of a, for any pair of hashes $\langle r, q\rangle \in M^2$ with $r \neq 0$, we have $\Pr[\langle h_a(k), h_a(l)\rangle = \langle r, q\rangle] = 0 \neq \frac{1}{m^2}$. Thus, the family defined in the script is not 2-independent.

5 Not quite universal hashing

The trick is similar to 4b). If we pick keys $k = (0, \ldots, 0)$ and $l = (1, 1, 0, \ldots, 0)$, then $h_a(k) = 0$ for all a, and $h_a(l) = 0 \Leftrightarrow a_1 + a_2 \equiv 0 \mod m$. Since $a_1, a_2 > 0$, this means $a_2 = m - a_1$. There are $m-1$ choices for a_1, each of which uniquely determines a_2, and so $\Pr[h_a(k) = h_a(l)] = \frac{1}{m-1} > \frac{1}{m}$.

6 Obfuscated quadratic probing

a) What the algorithm does is this: it iterates j from 0 to $m-1$, and in every iteration, it increases i by the current j. Thus, if i_j denotes the value of i in the jth iteration, then

$$\begin{aligned}i_0 &= h(k) \\ i_1 &= h(k) + 1 \\ i_2 &= h(k) + 1 + 2 \\ &\vdots \\ i_j &= h(k) + \sum_{n=0}^{j} n\end{aligned}$$

Thus if we denote our paramterized hash function as $h_j(k) = i_j \mod m$, we only have to express the partial sum in i_j as a quadratic function to prove that this is an instance of quadratic probing. This particular partial sum is well known:

$$\sum_{n=0}^{j} n = \frac{j(j+1)}{2} = \frac{1}{2}j + \frac{1}{2}j^2$$

Therefore, $h_j(k) = h(k) + \frac{1}{2}j + \frac{1}{2}j^2 \mod m$.

b) To prove that the probing sequence of every key covers the whole table, we show that any two steps of the sequence are distinct. Thus, let k be some key and let $r, s \in [m]$ with $r < s$. Now we have

$$\begin{aligned}
& h_r(k) \equiv h_s(k) && \mod m \\
\Leftrightarrow \quad & h(k) + \frac{1}{2}r + \frac{1}{2}r^2 \equiv h(k) + \frac{1}{2}s + \frac{1}{2}s^2 && \mod m \\
\Leftrightarrow \quad & \frac{1}{2}r^2 + \frac{1}{2}r \equiv \frac{1}{2}s^2 + \frac{1}{2}s && \mod m \\
\Leftrightarrow \quad & \frac{1}{2}s^2 + \frac{1}{2}s - \frac{1}{2}r^2 - \frac{1}{2}r \equiv 0 && \mod m
\end{aligned}$$

This is the case if and only if there exists an integer t such that

$$\begin{aligned}
\frac{1}{2}s^2 + \frac{1}{2}s - \frac{1}{2}r^2 - \frac{1}{2}r &= tm \\
\frac{1}{2}(s^2 - r^2 + s - r) &= tm \\
(s-r)(s+r+1) &= t2^{p+1}
\end{aligned}$$

The last step used that $m = 2^p$. We now show that this equation has no solution. Notice that $t > 0$ since the left hand side of the equation is positive.

Exactly one of $(s-r)$ and $(s+r+1)$ can be even: if $(s-r)$ is even, then $(s-r)+2r+1 = (s+r+1)$ is odd, and vice versa. Thus, 2^{p+1} can divide at most one of $(s-r)$ and $(s+r+1)$ since only even numbers have 2 as a factor.

Since $r < s \leq m-1$, we know that $(s-r) < m = 2^p < 2^{p+1}$, so 2^{p+1} cannot divide $(s-r)$. We also know that $(s+r+1) \leq (m-1) + (m-2) + 1 < 2m = 2^{p+1}$, and so 2^{p+1} cannot divide $(s+r+1)$ either. Therefore, 2^{p+1} divides neither.

Since only one of $(s-r)$ and $(s+r+1)$ is even, i.e. only one of them has 2 as a factor, 2^{p+1} would have to divide one of these two terms if it were to divide their product. Since 2^{p+1} divides neither of them, we conclude that $(s-r)(s+r+1) = t2^{p+1}$ wit $t > 0$ has no solutions, therefore, $h_r(k) \not\equiv h_s(k) \mod m$.

7 Robin Hood hashing

a) Suppose a collision happens and the two objects have probe sequence lengths (psl) i and j. No matter which object we choose to keep in the bucket, the other object will travel the same length in its probing sequence from that point on, denoted k, since the next empty bucket is the same for both objects. Therefore, the sum of the probe sequence length will remain the same and equal to $i+j+k$. Thus, the expected value of the psl does not change with the Robin Hood modification.

b) Every time we have two colliding objects, we move the one with the shorter psl. Since linear probing suffers from primary clustering, both object from this point on will need the same number of probes to find an empty bucket. Thus, the longest psl cannot be longer since we started with the shorter psl.

c) Similarly to the previous question, it is easy to see that the shortest psl will be larger compared to the hashing with linear probing. Thus, the variance decreases. More formally, suppose we have two colliding objects k_1 and k_2 and their corresponding probing sequence length (in collision) are i and j, respectively, where $i < j$. Then we have three possibilities:

- $i < j < \mathbb{E}(psl)$: If we move k_1 then the variance will decrease more than in the case we moved k_2.
- $i < \mathbb{E}(psl) < j$: If we move k_1 we decrease the variance, while if we move k_2 we increase it.
- $\mathbb{E}(psl) < i < j$: If we move k_1 we increase the variance less than if we move k_2.

Thus, in any case, moving the object with the smallest probing sequence position is beneficial towards decreasing the variance of the psl.

d) If we delete an object and do not mark the bucket then every time we search for an object we need to search the entire hash table to be sure it does not exist. Instead of leaving the bucket empty we place a marker (tombstone). This way, we can stop the search if we find an empty bucket (but not a marked one).

e) Although the table might be almost empty, we still have to go through the entire probing sequence we used to insert the objects. Thus, the search operation can be very inefficient, since at the worst case we have to go through the entire table.
To improve the performance we suggest a similar modification to the delete operation as we originally did to the insert operation (backward shift deletion). Specifically, every time we want to delete an object we do the following procedure: After we locate the object to be deleted, we do another linear probing until we find either an empty bucket or a bucket where the object was placed with the first probe; we will refer to this bucket as the stop bucket. Then, we delete the object and we shift backward all the buckets between the deleted bucket and the stop bucket. Then, we mark the last bucket (previous to the stop bucket) as empty.
The idea behind this modification is to keep the average psl as well as the variance of the psl low even after a large number of deletions, and hence improve the efficiency of the search operation.