Exercises for lecture 3: Hashing [last updated Jan. 22]

Questions

1. With quadratic probing and m = 16, where does the first collision occur?

2. Based on the simple scheme of user names and hashed passwords that I gave in class and your own experiences with forgetting a password, describe what a web server typically does when a user forgots his/her password and clicks on a "forgot password" link on a login page. Take the simple case that the webserver emails a new (randomly generated) password to the user.

3. Suppose you could steal a system file with user names and hashed passwords and suppose you knew the hash function used for the passwords. Would this give you access to user accounts on the system?

4. Suppose you know someone's login name and you know a password that is different from their password, but this other password has the same hash value as their password. Does that allow you to log in to their account?

5. Suppose you use open hashing (chaining) and the following keys are inserted: 5, 28, 19, 15, 20, 33, 12, 17, 10 and m = 9. For simplicity, here we do not distinguish a key from its hashcode, so we assume h(key)=key. In which slots do collisions occur?

6. Now suppose you use closed hashing with linear probing and the same keys as above are inserted. More collisions occur than in the previous question. Where do the collisions occur and where do the keys end up?

7. Java has a class called HashSet<E> which is used to represent a set of objects of type E. HashSet<E> uses the hashCode() method of class E and HashMap<K,V> use the hashCode() method of class K and both use a hash table. What is stored in the hash table buckets in each case? (Assume open hashing, that is, linked lists.)

8. For quadratic probing, note that if m=2500 then the quadratic step sizes will lead to a "self collision" on step i = 50 since i*i = 50*50 = 2500. Yet the result given in class says that the first self collision doesn't occur until i = m/2. This seems to be a contradiction. What is wrong with this argument?

9. [updated Jan. 18] This question assumes you have some background in basic probability theory. Suppose n keys are chosen randomly and the hash function distributes the keys uniformly at random over {0, 1, …, m-1}.

a. What is the probability that n=2 independently chosen keys have the same hash value i.e. what is the probability there is a collision among those two keys?

b. What is the probability that there are no collisions among any of the n keys? Assume n <= m.

10. Why is "(h0 + i*i) % m = (h0 + j*j) % m if and only if (i*i – j*j) % m = 0", as stated in the lecture ? Note that this holds for any m. [This question should be a piece of cake for those of you who've taken MATH 240 or some other course where you learned a bit of number theory.]

11. Double hashing solves one of the problems of quadratic probing (see slides) but it also has a disadvantage. What is the disadvantage?

12. Would a cryptographic hash function like SHA1 be suitable for use as a hashCode method in an implementation of a hash table?

13. In the context of closed hashing: suppose you have inserted so many keys/values into your hash table such that all slots are taken. Then collisions occur everytime. What can you do?

14. In the lecture slides, when we looked at collisions for closed hashing and the linear probing solution, we followed a sequence h(key), h(key) + 1 % m, h(key) +2 % m, etc. Notice that the "%m" was not computed for the first one, namely the initial h(key). Why not?

Answers

1. i*i = 0,1,4,9,16,.. so i*i mod 16 = 0, 1, 4, 9, 0, 9,.. The first collision is shown in red.

2. Before the the web server emails the new password, it hashes this new password and replaces the entry (username, hashed new password) in the password file.

3. It wouldn't give you direct access. Even though you know the user names, you don't know the passwords. You only know the hashed passwords. When you log in, you don't enter the hash of the password. You have to enter the password itself and you don't know it.

4. Yes. The system doesn't know their true password. It only knows the hash value of their password. So if you enter your password and it hashes to the same hash value, then the system cannot tell the difference. The hashed password matches the one in the "password file" and it assumes you have the correct password. It lets you in.

[You will not be surprised to learn that authentication schemes have been devised to avoid this scenario. Check out the Wikipedia entry for "rainbow tables" if you want to read more. But keep in mind this is beyond the scope of this course.]

5. The keys 5, 28, 19, 15, 20, 33, 12, 17, 10 map to slots 5, 1, 1, 6, 2, 6, 3, 8, 1. Collisions are shown in red.

6. The key 19 collides with key 28 at slot 1 and 19 goes into slot 2. Key 20 collides with key 19 at slot 2 and needs to go to slot 3. Key 33 collides with key 15 at slot 6 and needs to go to slot 7. Key 12 collides with key 20 at slot 3 and needs to go to slot 4. Key 17 goes into position 8. Key 10 collides with key 28 at slot 1, and then collides with 19 at slot 2, etc until it finally finds an open slot at position 0. The final positions of the keys are [10, 28, 19, 20, 12, 5, 15, 33, 17]. [updated Jan 22 -- the key positions haven't changed in the solution but more details were given.]

7. For HashMap<K,V>, the hashCode() method is defined by K class – that is, you hash the keys. Each bucket contains a linked list of (key, value) ordered pairs. For HashSet<E>, the hashCode() method is defined by the E class. Each bucket contains a linked list of objects of type E.

8. The array size has to be a prime number. But m=2500 is not prime.

9. Let the randomly chosen keys be k1,.. kn.

    a) The first key can have any hash value, so fix that hash value. The probability that k2 has that same hash value is 1/m.

    b) Continuing from (a)… The probability that k2 has a different hash value than the fixed k1 is (m-1)/m since k2 can have any of m-1 values without colliding with k1. So now we fix k2 (no collision) and ask, what is the probability that k3 has a different hash value than k1 and k2? The answer is (m-2)/m since k3 can hash to any of the m-2 values, i.e. values other than the hash values of k1 and k2. Reasoning similarly, the probability that k_j+1 has a different hash value from k_j is (m-j)/m, for j <= m. So, if n <= m, then the probability that all n keys have a different hash values (no collisions) is (m-1)(m-2)…(m-(n-1)) / m^(n-1).

10. The expression "((h0 + i*i) % m) = ((h0 + j*j) % m)" means that "h0 + i*i" and "h0 + j*j" have the same remainder when you divide these quantities by m, which implies that each of them is of the form "am + r" where a is an integer and the remainder r is in {0, 1, … m-1}. Thus when you take the difference "(h0 + i*i) – (h0 + j*j)", the remainders cancel and you get that the difference is a multiple of m. But since the h0's also cancel, this means that i*i – j*j is a multiple of m, which means that if (i*i – j*j) % m = 0.

11. Computing a hash function takes time – a good hash function requires a lot of computation. Computing two hash functions takes twice as much time (whereas computing i*i and adding it to h(key) and taking the mod is relatively cheap).

12. No. Cryptographic hash functions are expensive to compute and the main design goal is to make it virtually impossible to take a hash value, call it h(x) for some unknown x, and find an x* (possibly x, but not necessarily) such that h(x*) = h(x). In particular, cryptographic hash function produce large values (e.g. a 256 bit hash value can be interpreted as a binary number in the range 0 to 2^256-1. (See my COMP 250 lecture 1 if you don't understand that last statement.) But for a hash table you will compress this huge value into a small value that can index into a small array. In a nutshell, cryptographic hashing is overkill for use as a hashcode in a hash table.

13. You need to build a new hash table. Specifically you need a larger array (much bigger m). Note that the compression function will be "mod" a new value of m and so the hash function changes. (You can use the same hashcode as before though.) With the new hash function, take each of the key/values from the current full hash table and put them (in some arbitrarily chosen sequence) into the new hash table. You will probably have some collisions when you do this, and you'll need to resolve them using your favorite scheme.

14. The initial hash value is in {0, 1, …, m-1} so there is no need for "%m". However, once we start adding values to h(key), we can exceed m-1 and so we need to take the "mod".