Brandon Vo

1. Suppose you are given the following keys: 112, 2542, 9992, 5502 and the following hash function h(x) = x mod 10.

Hash the keys using the hash function. How many keys collide? Choose a random[1] hash function, $h_{a,b}$ from $H_{10007,10}$. Include your random choice of $h_{a,b}$ with your answer to this problem. Before you rehash the numbers with the new hash function, determine the probability that the keys 112 and 2542 collide when hashed with $h_{a,b}$? Hash each of the keys 112, 2542, 9992, 5502 with $h_{a,b}$. How many keys collided?

If you had 1000 keys (all keys were positive integers less than 10000) inserted into a hash table of size 2000 using a hash function, $h_{a,b}$, randomly chosen from $H_{10007,2000}$ (the family of universal hash functions we defined in class). What is the expected number of collisions you would have if you inserted a new key, x, into the hash table?

[1]Use a random number generator.

$$h(x) = 112\ mod\ 10\ \equiv 2$$

$$h(x) = 2542\ mod\ 10\ \equiv 2$$

$H_{a,b}$	H(x)
112	2
2542	2
9992	2
5502	2

All of the keys provided end in 2. All keys will be equivalent to 2 mod 10 and as such, they will all collide.

Choosing a random hash function $h_{a,b}$

$$1 \le a \le p-1, 0 \le b \le p-1, p \ge all\ keys$$

Will be using a random number a = 975 and b = 86 with p = 10007.

$H_{975,86}$ from $H_{10007,10}$: ax+b mod p mod 10

$$h_{1003,8} = \big((975x + 86)\ mod\ 10007\big)\ mod\ 10$$

Probability of keys 112 and 2542 colliding:

Randomly choosing an (r, s) pair where r ≠ s, the probability that they collide by mod m will be ≤ 1/m

$$\big((975 * 112 + 86)\ mod\ 10007\big)\ mod\ 10 = \big((975 * 2542 + 86)\ mod\ 10007\big)\ mod\ 10$$

$$PR[h(k) = h(k')] \le \frac{1}{m}$$

$$With\ mod\ 10, we\ have\ at\ most\ \frac{1}{10}\ probability\ of\ colliding$$

Brandon Vo

Hashing each of the keys

$$h_{a,b} = \left((975k + 86)\ mod\ 10007\right) mod\ 10$$

112	$\left((112 * 975 + 86)\ mod\ 10007\right) mod\ 10$	6
2542	$\left((2542 * 975 + 86)\ \ mod\ 10007\right) mod\ 10$	7
9992	$\left((9992 * 975 + 86)\ mod\ 10007\right) mod\ 10$	5
5502	$\left((5502 * 975 + 86)\ mod\ 10007\right) mod\ 10$	4

None of the keys collided with the a and b values added to the hash family.

Expected number of collisions if you had inserted x into the hash table of size 2000 holding 1000 keys

Probability of Conflict: 1/m*n where m is the size of the table and n is the number of keys we have already inputted.

We have 1000 keys: n = 1000

Table is size 2000: m = 2000

$$Probability\ of\ a\ collision: \frac{1}{m} = \frac{1}{2000}$$

$$E[C_x] = \sum_{y \in S, y \neq x} C_{x,y} \leq \frac{n}{m}$$

$$E[x] = \frac{1}{2000} * 1000 = 0.5$$

Brandon Vo

2. Using the technique for perfect hashing that we discussed in class, store the set of keys: {10, 22, 37, 40, 52, 60, 70, 72, 75}.

For your first hash function ("outer hash" function) use $h_{a,b}(k)$ = ((ak + b) mod p) mod m, where a = 3, b = 42, p = 101, m = 9.

Page 278 CLRS

Hashing the set of keys provides us with

H(k)	((ak + b) mod p) mod m	H(k)
H(10)	$((3*10+42) mod\ 101) mod\ 9$	0
H(22)	$((3*22+42) mod\ 101) mod\ 9$	7
H(37)	$((3*37+42) mod\ 101) mod\ 9$	7
H(40)	$((3*40+42) mod\ 101) mod\ 9$	7
H(52)	$((3*52+42) mod\ 101) mod\ 9$	7
H(60)	$((3*60+42) mod\ 101) mod\ 9$	2
H(70)	$((3*70+42) mod\ 101) mod\ 9$	5
H(72)	$((3*72+42) mod\ 101) mod\ 9$	2
H(75)	$((3*75+42) mod\ 101) mod\ 9$	2

This leaves us with 4 secondary tables of size 1, 9, and 16.

Using the hash family of $H_{3,10}$ for T=2 and $H_{16,23}$ for T=7

$$H_{4,10}(60) = \big((4*60+10)\ mod\ 101\big)\ mod\ 9 = Slot\ 3$$

$$H_{4,10}(72) = \big((4*72+10)\ mod\ 101\big)\ mod\ 9 = Slot\ 6$$

$$H_{4,10}(75) = \big((4*75+10)\ mod\ 101\big)\ mod\ 9 = Slot\ 7$$

$$H_{16,23}(22) = \big((16*22+23)\ mod\ 101\big)\ mod\ 16 = Slot\ 8$$

$$H_{16,23}(37) = \big((16*37+23)\ mod\ 101\big)\ mod\ 16 = Slot\ 9$$

$$H_{16,23}(40) = \big((16*40+23)\ mod\ 101\big)\ mod\ 16 = Slot\ 7$$

$$H_{16,23}(52) = \big((16*52+23)\ mod\ 101\big)\ mod\ 16 = Slot\ 15$$

Brandon Vo

Perfect Hash table

T	Table Size		/	/	/	/
0	1	$H_{0,0}$	H(10)	/	/	/
1			/	/	/	/
2	9	$H_{4,10}$	H(60) [Slot 3]	H(72) [Slot 6]	H(75) [Slot 7]	/
3			/	/	/	/
4			/	/	/	/
5	1	$H_{0,0}$	H(70)	/	/	/
6			/	/	/	/
7	16	$H_{16,23}$	H(40) [Slot 7]	H(22) [Slot 8]	H(37) [Slot 9]	H(52) [Slot 15]

Brandon Vo

3. What is the probability that you use at most 16n space for the secondary hash tables (i.e. $\sum_{j=0}^{m-1}(n_j)^2$)? What is the probability (in terms of k) that you use at most kn space (for some constant k)?

$$\sum_{j=0}^{m-1}(n_j)^2$$

With m as the space being used, if we're using 16n space, then m = 16n

$$Using\ Markov's\ inequality\ for\ high\ probability\ bounds:\ t = 16n$$

$$PR[X \geq \alpha] \leq \frac{E[x]}{\alpha}$$

$$PR\left[\sum_{i=0}^{m-1}(n_i)^2 \geq t\right] \leq 1 - \frac{2n}{t}$$

$$PR\left[\sum_{i=0}^{m-1}(n_i)^2 \leq 16n\right]$$

$$PR\left[\sum_{j=0}^{m-1} m_j \leq 16n\right] \leq E\left[\frac{\sum_{j=0}^{m-1} m_j}{16n}\right]$$

$$E\left[\frac{\sum_{j=0}^{m-1} m_j}{16n}\right] = \left[\frac{\sum_{j=0}^{16n-1} n_j}{16n}\right] = 1 - \frac{2n}{16n}$$

$$1 - \frac{1}{8} = \frac{7}{8}\ chances\ for\ at\ most\ 16n\ spaces.$$

For k spaces:

$$PR\left[\sum_{i=0}^{m-1}(n_i)^2 \leq kn\right]$$

$$PR\left[\sum_{i=0}^{m-1}(n_i)^2 \geq t\right] \leq 1 - \frac{2n}{t}$$

$$PR\left[\sum_{j=0}^{m-1} m_j \leq kn\right] \leq E\left[\frac{\sum_{j=0}^{m-1} m_j}{kn}\right] = 1 - \frac{2n}{kn}$$

$$1 - \frac{2}{k}\ probability\ in\ terms\ of\ k$$

Brandon Vo

4. (10 points) How many times must you attempt to construct one of a perfect hash table's sub-tables so you have one with probability greater than 99.99999%? This is a failure rate of 1 out of one ten million.

In a perfect hash, theorem 11.9 states that storing n keys in a hash table of m = n^2 would have a probability less than ½ for any collision.

Asking for how many times we would be performing this hashing maneuver to reduce the failure rate of collision. Just multiplying ½ constantly to get the probabilities.

$$1-\left(\frac{1}{2}\right)^{n} > 99.99999\%$$

$$\left(\frac{1}{2}\right)^{n} = 0.0000001$$

$$n = 23.25349 \ \approx About\ 24\ subtables\ need\ to\ be\ constructed$$

∗Many of these questions came from outside sources.

Brandon Vo

5. You keep thinking back to the question for homework assignment 1. You are sure you could have solved this faster on average. Here was the question and the new running time you think you can achieve:

> Trying to finance your education, you decide to invest your D dollars in the stock market. The problem is you don't know which stocks to invest in. You decided to ask your friends for stock tips and invest in every stock any of your friends suggests. If you receive n unique stock names you will invest D/n in each of the n stocks. Since some of your friends have given you the same stock name, you need to find a way to remove the duplicates.

Design an efficient algorithm to perform this task. You may call any algorithm as a subroutine that was presented in lectures 1-3. Your algorithm must run in O(n) average case time where n is the number of names given to you by your friends.

Demonstrate that your algorithm runs in O(n) average case time.

State your improved average and worst-case running time is of your algorithm in big-Oh notation. Justify your reasoning.[2]

We can create a hash table and using collision to check for duplicates.

- Create a hash table of size m to hold the set of unique stocks of size n.
- Whenever we insert a stock into the table, we will take name of the stock as the key and hash it. The hash of the stock will be inserted into the hash table.
- If the hash index is empty, that means we have a unique stock option which will be inserted into the table.
    - If our hash ends up occupying a space which already has a value, that means we have a hash collision. This means this stock option is possibly a duplicate stock option provided by another friend. In this case, we can ignore this stock option as it has already been placed into the hash table.
- This function will run in O(n) time because we go over the entire array of stock options only once. The number of iterations depends only on the size of the array *n* holding all stock options.

```
HASH-BUILD-INSERT (A)

    //A is the array of stocks given by friends

    //T will be the list used to hold the hashes of the unique list of stock options

    Create array T          //Will serve as the hash table

    For i = 0 to A.length   //Because we go through all of array A, this will take O(n) time to check

                            //the entire list of stock options and put them into hash table T

        J = h(A[i], i)      //Take the stock option and create a hash from it
```

Brandon Vo

```
                **If** T[j] != NIL or T[j] !=DELETED    //Hash collision detected.  A[i] is a duplicate stock
                        **Continue**          //Skip this entry and continue to the next one
                **Else** T[j]= k        //Otherwise, this slot is empty.  Add the new stock option.
```

HASH-TRAVERSE(T)   //Get the entire list of unique stock options from array T

//To check the entire list of n unique stock options, we must go through the entire hash table T.

//To iterate over the entire list in T, it would take O(n) time.

```
        For i = 0 to T.length
                If T[i] != NIL or DELETED //If the slot already has something
                        PRINT T[i]                      //Print out the stock option
```

Brandon Vo

6. Your strategy worked and you have earned 1/2 of your fall tuition on the stocks you invested in. NYU is full of smart people (or you just got lucky...)

Most of your friends gave you multiple stock tips. You decide to take out to lunch any friend whose stock tips earned you more than 20% return (i.e. if friend 1 suggested AMD, TSLA, and GME - by following your friends advice you made 22%)[3]

Let n be the number of stock tips you received, and f is the number of friends, and $m \leq 5$ is the maximum number of stock suggestions any one of your friends gave you. In $O(n + f)$ average case time, determine who you will be taking to lunch. You have an array A, where A[i] holds the stock name and the people who suggested you buy the stock and how much money you have made from owning this stock.

Design an efficient algorithm to perform this task. You may call any algorithm as a subroutine that was presented in lectures 1-3. Your algorithm must run in $O(n + f)$ average case time where m is the number of names given to you by your friends.

Justify that your algorithm runs in $O(n + f)$ average case time.

if you wrote a clever algorithm, see if you can justify that your algorithm runs in $O(n+f * m)$ average case time where now $m \leq n$. If you can do this, you will receive a small amount of extra credit,

Goal: Find the set of friends whose stock advice adds up to > 20% returns.

We can create a chained hash table which holds the list of friends which hold the list of stock options that are chained by the list of stock values they offered.

- To build the hash table, we will go through the array list A which holds the list of all stock options and friends.
    - We'll create an array called Friend to serve as the hash table.
    - A[i] contains the stock located at index *i*. A[i].list[] will contain the list of friends who suggested that stock.
- To begin, we'll go through the entire list of stocks in A. For each stock option in A, we will go through the list of friends and take the hash of friends.
    - This loop going through the stocks of A will be the outer loop, taking O(n) time to go through the entire list of stocks.
    - For each list of friends, we take the hash and put it into the array RETURNS.
        - If the friend already exists in RETURNS, that means that particular friend suggested an additional stock which we will add to the friend by chaining it in a linked list
        - In addition, we will add up the total amount of returns that friend has provided for us.
        - The process of adding an additional chain for one friend takes O(1) time.
        - Going through the entire list of friends would take O(f) time.

  - Each friend will have up to m <= 5 stock suggestions, meaning the chain for the friend will go up to 5. Going through each friend's linked list of stocks would take O(m). To go through every friend's list of stock suggestions would take O(f*m) time.
- There are *f* friends. Each friend can suggest up to *m* stocks. The inner loop being used to count the total of friends and each of his/her stock returns would take O(f*m) time.
  - Inner Loop: O(f*m)
- The time needed to go through the outer loop of stocks would take O(n) time. The inner loop of O(f*m) depends heavily on the number of stocks being iterated over as each stock counts the list of friends.
  - Outer Loop: O(n)
- To go over the entire list of stock suggestions from f friends, it would take O(n+f*m) time to build the hash table of friends and the total returns from every stock suggestion provided.

Setting up the hash table

```
BUILD-HASH(A):

Create array RETURNS

//Friend will hold the list of friend names while stock will hold the stock name and return

For i = 0 to A.length                //The Outer Loop:  Takes O(n) time to go through the list of stocks

        For j = 0 to A[i].Friend.length //Inner Loop:  Go through the list of f friends and insert

                                        //the new stock.  Calling it will take O(f*m)

//Because the inner loop will depend on the outer loop of stocks, it requires O(n+f*m) to count every

//stock for every friend

                Name = H(A[i].Friend[j])          //Take the friend and hash it

                If RETURNS[Name] != NIL       //This friend isn't in the list

                        RETURNS[Name] = A[i].friend_name  //Add the friend to the friend hash

                        HASH-INSERT-CHAIN(RETURNS, A[i].value) //Insert the stock value

                                                                //to the chain

                        RETURNS[Name].total += A[i].value              //Add to the total return value

                Else    //The friend already made a stock suggestion

                        HASH-INSERT-CHAIN(RETURNS, A[i].friend_name) //Insert new stock value

                        RETURNS[Name].total += A[i].value              //Add to the total return value

        EndFor

EndFor
```

Brandon Vo

Checking which friends to take out

```
CHECK-FRIENDS(RETURNS):
For i = 0 to RETURNS.length        //Travelling through the list of friends takes O(f) time
    If RETURNS[i].total ≥ 20%
        PRINT(RETURNS[i] SHOULD BE TAKEN OUT FOR LUNCH)
    End For
End For
```

Brandon Vo

7. People started to hear about your talent in the stock market (perhaps you shouldn't have posted your gains on r/wallstreetbets) and they have started asking you for advice. You are never one to not spot an opportunity, and you decided to write a weekly newsletter and charge people for access. This took off (perhaps it helped that your cousin posted about your success and your for pay newsletter on reddit...)

[2]at this point, I don't think I need to remind you to not take any stock advice from this class!

[3]If a friend gave you 3 stock tips, you take the average return for those stock tips and test if the average was more than a 20% return.

However your mass email provider has been having problems and sometimes the newsletters are not sent to your subscribers. Not wanting to get a bad reputation, you want to quickly determine if a users complaint is legitimate.

The problem is you haven't gotten a new computer and you don't have that much memory to store all your subscribers, so you need to come up with an alternative method to know (within a reasonable probability) if the request is legitimate. If it is legitimate, you quickly forward them the newsletter (you have their email address from the customer support request.)

Here is your plan:

• You create an array S and initialize all the values to 'n'.

• You then take all your subscribers and hash their email address, and set S[hash(email address)] = 's'.

If a new person subscribes, you hash their email address, and set S[hash(email address)] = 's'.

• If someone emails you saying they didn't get your newsletter, you quickly hash their name. If S(hash(email address)] = 's', you send them your newsletter, otherwise you send them your sign up page on your website.

For this question, assume the simple uniform hashing assumption.

(a) What is the probability a subscriber doesn't get their newsletter and when you checked to see if they were a subscriber you found that S[hash(email address)] = 's'? Assume they used the correct email address.

This is asking for the probability that S[hash(email address)] = 's'

However, all subscribers upon having their email addresses hashed and put in the table will all be set to 's'. When checking if a subscriber didn't get their newsletter, we will check the matching hash and always find it to be 's' because all spots that hold a subscriber will be marked as 's'.

It is guaranteed that a subscriber will be found with S[hash(email address)] = 's'.

(b) What is the probability a non-subscriber doesn't get their newsletter and when you checked to see if they were a subscriber you found that S[hash(email address)] = 's'?

The probability of a non-subscriber also being marked as one would be the probability of the non-subscriber having a hash-collision with someone else that is a subscriber, so we would have to calculate the probability of a non-subscriber having a collision with a subscriber. This means we would have to calculate the probability of a hash collision for each slot.

$$\Pr[hash\ collision\ with\ 1\ object] = \frac{1}{m}\ where\ m = table\ size$$

$$\Pr[no\ collision\ with\ 1\ object] = 1 - \frac{1}{m}$$

$$\Pr\ [X\ is\ not\ a\ subscriber\ but\ S[b] = 'S'$$

This process is repeated for every slot in the hash table because the chances of colliding with one slot is an independent probability.

$$\Pr[at\ least\ one\ subscriber\ collides\ with\ X] = 1 - PR[one\ person\ does\ not\ collide\ with\ X]^a$$

$$1 - \left(1 - \frac{1}{\mathrm{m}}\right)^a = \frac{1}{m}\ where\ a\ is\ the\ number\ of\ indexes\ in\ hash\ table\ S\ marked\ as\ 's'$$

$$Probability\ of\ a\ hash\ collision{:}\ 1 - \left(1 - \frac{1}{m}\right)^a$$

Where *m* is the hash table size and *a* is the number of slots in the hash table marked as ‘s’.

Brandon Vo

8. You started to realize your solution in question 7 was letting a few people have your newsletter without subscribing. You don't want to buy a new laptop (you are making too much money off the market) so instead you use two hash functions as follows. • You create an array S and initialize all the values to 'n'.

• You then take all your subscribers and hash their email address, and set

S[hash1(email address)] = 's', and S[hash2(email address)] = 's'.

If a new person subscribes, you hash their email address, and set S[hash1(email address)] = 's', and S[hash2(email address)] = 's'.

• If someone emails you saying they didn't get your newsletter, you quickly hash their name. If both S[hash1(email address)] = 's' and S[hash2(email address)] = 's', you send them your newsletter, otherwise you send them your sign up page on your website.

For this question, assume the simple uniform hashing assumption.

(a) What is the probability a subscriber doesn't get their newsletter and when you checked to see if they were a subscriber you found that S[hash1(email address)] = 's' and S[hash2(email address)] = 's',? Assume they used the correct email address.

Everyone who is a subscriber will have their names in hash1 and hash2 marked as 's'. As such, when you search for someone who is a subscriber for both hashes, it is a guaranteed collision with an index marked with 's'

It is guaranteed that you will find a subscriber marked as 's' for both hashes.

(b) What is the probability a non-subscriber doesn't get their newsletter and when you checked to see if they were a subscriber S[hash1(email address)] = 's' and S[hash2(email address)] = 's',?

This would now be the probability of a non-subscriber getting two hash-collisions on both hash 1 and hash 2. The probability of getting a hash collision would be independent for both hashing algorithms, making it PR[S[Hash1(email address)]='s'] * PR[S[Hash2(email address) = 's'] for every slot in both hash tables.

$$\text{PR}[\text{S}[\text{Hash1(email address)}] = \text{'s'}] * \text{PR}[\text{S}[\text{Hash2(email address)}] = \text{'s'}]$$

Similar to second level hashing:

Where m represents the table size and a represents the number of indexes in the table marked as 's'.

$$1 - (\Pr[One\ of\ A\ does\ not\ collide\ with\ x]^{a})\hat{}2\ since\ we're\ using\ two\ levels\ of\ hashes.$$

$$1 - \left(1 - \frac{1}{m}\right)^{2a}\ probability\ to\ get\ hashed\ to\ an\ 's'\ position\ for\ hash\ 1\ and\ hash\ 2$$

Brandon Vo

9. (3 bonus points) Think of a good exam question for the material covered in Lecture 3.

Given the following set of hash functions:

$$h(k,i) = \left(\left(c * I + h_{a,b}(k) + h_{c,d}(k) + i * h_{e,f}(k)\right) \bmod m\right) \bmod p$$

How many probe sequences are being used here?