In-Class Exercise

Hashing

Used in:

- Encryption – for authentication
  - Hash a digital signature, get the value associated with the digital signature,and both are sent separately to receiver. The receiver then uses the same hash function on the signature, gets the value associated with that signature, compares the messages. If the same - authentication
- Database access
  - Names->phone numbers
  - Author->articles (or books)
  - Topic->articles
  - usernames->passwords
  - Social Security Number->everything about you
  - Zip codes->regions
- Translation
- Anagrams (how?)

Idea:

Hashing is designed solely for finding keys quickly. With hashing, the keys are not put in any order. We cannot find the smallest, the largest, the median, etc without converting from a hash set to a different data structure (or doing a lot of looping). If we hash correctly, we can, however, find whether an item is in a database very quickly.

Usually a key is associated with a value or set of values. In other words, with our student example, the key was the student's id, but the values include the student's first and last name, their address, their year, their major, their gpa, etc. None of the values have to be unique within the database (2 students can even have the same name, let alone the same major), but the key we use for finding this information must be unique.

There are 3 parts to dealing with hashing:

- **Coming up with a good hashing function**
  - Must be quickly computable
  - Must lead to indices that are evenly distributed throughout the array
  - Must be consistent (always compute to the same index)
    - A good hash function can depend on the type of data you're dealing with
- **Coming up with a good array size**
  - Prime numbers are best
  - You want an array that has about 2x the number of keys
    - So a prime number that is greater than 2x the number of keys is a good number
- **Dealing with the inevitable collisions**
  - **Chaining**
    - Make your hash array be a linked list
    - Insert every collision into the linked list
    - Worst case: every key ends up in the same index
      - Linked list is n in length for n keys
      - REALLY bad hash function

- **Linear probing**
  - When you have a second key hashing to the same index as a first key, keep "probing" at the next index(es) in the array until you find an empty spot
  - Insert at that index
    - So if a second key hashes to index 32 and there's already a value there, look at index 33, then 34, then 35, etc. When you get to the end of the array, loop back to index 0
    - Watch out for infinite loops! (when will that happen?)
    - Causes "clustering"
      - When a bunch of keys are clustered together in one area in the array
      - This slows the whole finding of the key quickly process down
      - The more clustering, the more there's a tendency to cluster
      - So we probably want to try to avoid this
- **Quadratic probing**
  - An attempt to avoid clustering
  - After a collision, instead of looking at the next index (h(k) + 1), look at indices quadratically (h(k) + 1^2, then 2^2, then 3^2, etc, so look at h(k)+1, then +4, then +9, etc.
    - Again, loop back to 0 when get to end of array
  - Helps most when a lot of keys cluster to the same area
  - Doesn't help when a lot keys hash to the same index
  - And still end up having areas of clustering
- **Pseudo-random probing**
  - An alternative to quadratic probing
  - Idea: can't randomly choose next index after a collision – could never find the key again
  - But we could generate a list of random numbers and use that list to determine the next index
    - E.g., {0, 8, 3, 4, 7, 2, 9, 6, 1, 5}
    - index = h(k) + randomarr[0]
    - If collision, index = h(k) + randomarr[1],
    - Then h(k)+ randomarr[2], etc.
  - VERY quick to calculate!
  - Maybe a bit better than quadratic probing when it comes to clustering, but you pretty much have the same problems overall
- **Double hashing**
  - Have 2 different hashing functions. (there are MANY hashing functions, and you can always make your own)
  - If the first hash function on a key gives an index that is already taken, try the second hashing function on the key.
  - The second hashing function can be something simple like h(k) + 1 + ct*(k mod (m)) where ct is the number of times you've had a collision with k and m is a prime number less than the size of the array (7 is nice). K is the numeric equivalent of the key (so in a string, it might just be adding up the asci values of the letters, or even a=1, b=2, etc.)
  - We have to add 1 because if k mod m is 0, we could end up in an infinite loop
    - E.g., arraysize = 11, m = 7
      - h2(k) = i+(k mod(m-1))
      - h2(k) = i+(k mod(m)
    - h_0(55) = 55%11 = 0
    - h_0(66) = 66%11 = 0 X

        - H2((66) =(1+k%(M))) = 1 + (66%7) = 4
        - P2(66) = 0 + 1*4 = 4%11 = 4
    - h_0(11) = 11%11 = 0 X
        - H2(11) = 1+k%(M))) = 1 + (11%7) =5
        - P2(11) = 0 + 1*5 = 5%11 = 5
    - h_0(88) = 88%11 = 0X
        - H2(88) = 1+k%(M))) = 1 + (88%7) =5
        - P2(88) = 0 + 1*5 = 5%11 = 5X
        - P3(88) = 0 + 2*5 = 10%11 = 10
  - Double hashing helps with the problem of many keys hashing to the same first value – in theory the 2 hashing functions should be different enough that the chances of both hashing to the same index with the same key should be small
- **Back to array size:**
  - Goal – find things quickly (in 0(1) if possible)
  - REALLY large array sizes
    - few collisions (get close to 0(1))
    - But a lot of extra space
  - Smaller array sizes
    - Many more collisions
    - But less extra space
  - Ideal – about 2x number of keys (up to next prime number)
  - Load factor:
    - What happens if number of keys is changing?
    - Load factor = number of keys/ array size
    - Want this to be less than 70%
    - If the load factor gets over 70%, we want to:
      - Create a new array – double in size of old array, then up to next prime number
      - Rehash all the keys in the old array into the new array
    - Equally, if the load factor gets below 25%,
      - Halve the array size, up to the next prime
      - Re-hash all the keys in the old array
  - Rehashing – 0(n) - not good
    - Don't want to do this too often

Hashing Problems:

1. Currently you've got an array of size 23 and 17 keys. What should you do?

2. With chaining, what would be the worst case for finding a particular key, and when would that occur?

3. With linear probing, why do we want to put as a condition,

   while (ct < arraysize) {

4. What is the best case for finding a particular key in the hash table using double hashing?

5. Why is deleting keys a problem using linear/quadratic/pseudo-random probing?

6. With a perfect hashing function, and a set T consisting of {t1,t2,t3,…tn} that has already been hashed into a hash array, how long would it take to tell whether set S consisting of {s1,s2,s3,…sm} is a subset of set T?

7. Hash the following words into the hash map below using chaining, assuming each character has the corresponding numerical value, and the hash function is:

   word[0]+word[1]+word[2]+…word[wordlength-1]%arraysize

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

Words to hash: cat, dog, bird, ant, bug, bat, fox

Array:

0	1	2	3	4	5	6	7	8	9

8. Hash the following words into the hash map below using linear probing, assuming the same hash function as above (same list of words to hash):

Words to hash: cat, dog, bird, ant, bug, bat, fox

Array:

0	1	2	3	4	5	6	7	8	9

9. Hash the following words into the hash map below using double hashing, assuming the same hash function as above (same list of words to hash) for the first hash, and h(k) + 1 + ct*(h(k)%7) as the second hashing function:

Words to hash: cat, dog, bird, ant, bug, bat, fox

Array:

0	1	2	3	4	5	6	7	8	9

If Finished, Start Lab 5