**CS2040S: Data Structures and Algorithms**

# Discussion Group Problems for Week 10

*For: March 20–March 24*

**Problem 1. From Linear to Quadratic Probing**

In the lecture, we learnt that linear probing is one of the open-addressing schemes. In linear probing, the algorithm searches for the next available bucket on a collision sequentially.

(a) Given the following integers: [23, 99, 49, 41, 43, 60, 99, 25, 63] and an empty hash table of size 7 with the following hash function (again!) `h(x) = x % 7`, how does the final hash table look like?

Quadratic probing is another open-addressing scheme very similar to linear probing. Note that we can also express linear probing with the following pseudocode (on insertion of element $x$):

```
for i in 0..m:
  if buckets[(hash(x) + i) % m] is empty:
    insert x into this bucket
    break
```

Quadratic probing follows a very similar idea. We can express it as follows:

```
for i in 0..m:
  // increment by squares instead
  if buckets[(hash(x) + i * i) % m] is empty:
    insert x into this bucket
    break
```

(b) Consider a hash table with size 7 with the same hash function `h(x) = x % 7`. We insert the following elements in the order given: 5, 12, 19, 26, 2. What does the final hash table look like?

(c) Continuing from the above question, we now delete the following elements in the order given: 12, 5. What does the final hash table look like?

(d) Can you construct a case where quadratic probing fails to insert an element despite the table not being full?

**Problem 2. Table Resizing**

Suppose we follow these rules for an implementation of an open-addressing hash table, where $n$ is the number of items in the hash table and $m$ is the size of the hash table.

(a) If $n = m$, then the table is *quadrupled* (resize $m$ to $4m$)

(b) If $n < m/4$, then the table is *shrunk* (resize $m$ to $m/2$)

What is the minimum number of insertions between 2 resize events? What about deletions?

**Problem 3. Implementing Union/Intersection of Sets**

Consider the following implementations of sets. How would intersect and union be implemented for each of them?

(a) Hash table with open addressing

(b) Hash table with chaining

**Problem 4. Binary Counter**

Binary counter ADT is a data structure that counts in base two, i.e. 0s and 1s. Binary Counter ADT supports two operations:

- `increment()` increases the counter by 1
- `read()` reads the current value of the binary counter

To make it clearer, suppose that we have a $k$-bit binary counter. Each bit is stored in an array $A$ of size $k$, where $A[k]$ denotes the $k$-th bit (0-th bit denotes the least significant bit). For example, suppose $A = [1, 1, 0]$, which corresponds to the number `011` in binary. Calling `increment()` will yield $A = [0, 0, 1]$, i.e. `100`. Calling `increment()` again will yield $A = [1, 0, 1]$, the number `101` in binary.

Suppose that the $k$-bit binary counter starts at 0, i.e. all the values in $A$ is 0. A loose bound on the time complexity if `increment()` is called n times is $O(nk)$. What is the amortized time complexity of `increment()` operation if we call `increment()` $n$ times?

**Problem 5. Scapegoat Trees**

Consider the Scapegoat Tree data structure that we have implemented in Problem Sets 4-5. We assume that only insertions are performed on our Scapegoat Tree. In this question, we will use amortized analysis to reason about the performance of inserts on Scapegoat Trees.

(a) Suppose we are about to perform the `rebuild` operation on a node $v$. Show that the amount of entries that *must* have been inserted into node $v$ since it was **last rebuilt** is $\Omega(size(v))$.

(b) Show that the *depth* of any insertion is $O(\log n)$

(c) (Optional) It's correct to claim that the amortized cost of an insertion in a Scapegoat Tree is $O(\log n)$. Use the previous two parts to prove this. *Hint:* Suppose a constant amount is "deposited" at every node traversed on an insertion.

**Problem 6. Locality Sensitive Hashing (Optional)**

So far, we have seen several different uses of hash functions. You can use a hash function to implement a *symbol table abstract data type*, i.e., a data structure for inserting, deleting, and searching for key/value pairs. You can use a hash function to build a fingerprint table or a Bloom filter to maintain a set. You can also use a hash function as a "signature" to identify a large entity as in a Merkle Tree.

Today we will see yet another use: clustering similar items together. In some ways, this is completely the opposite of a hash table, which tries to put every item in a unique bucket. Here, we want to put similar things together. This is known as *Locality Sensitive Hashing.* This turns out to be very useful for a wide number of applications, since often you want to be able to easily find items that are similar to each other.

We will start by looking at 1-dimensional and 2-dimensional data points, and then (optionally, if there is time and interest) look at a neat application for a more general problem where you are trying to cluster users based on their preferences.

**Problem 6.a.** For simplicity, assume the type of data you are storing are *non-negative integers.* If two data points $x$ and $y$ have distance $\leq 1$, then we want them to be stored in the same bucket. Conversely, if $x$ and $y$ have distance $\geq 2$, then we want them to be stored in different buckets. More precisely, we want a hash function $h$ such that the following two properties hold for every pair of elements $x$ and $y$ in our data set:

- If $|x - y| \leq 1$, then $\Pr[h(x) = h(y)] \geq 2/3$.
- If $|x - y| \geq 2$, then $\Pr[h(x) \neq h(y)] \geq 2/3$.

First, we choose buckets of size *size*, and choose a value from the range $[0, size - 1]$ as the starting point for the first bucket. Let this point be $a$. Each bucket then starts immediately when the previous bucket ends. We define this **randomly chosen** hash function $h(x) = floor((a+x)/c)$ as putting a point in the proper bucket.

What is an appropriate value of *size*? What is an appropriate value for $c$ in the hash function above? See if you can show that the strategy has the desired property.

*Note: the hash function is randomly chosen, which is why there is probability involved. However, once it is chosen, it is deterministic and does not change between operations.*

**Problem 6.b.** Now let's extend this to two dimensions. You can imagine that you are implementing a game that takes place on a 2-dimensional world map, and you want to be able to quickly lookup all the players that are near to each other. For example, in order to render the view of player "Bob", you might lookup "Bob" in the hash table. Once you know $h(\text{"}Bob\text{"})$, you can find all the other players in the same "bucket" and draw them on the screen.

Extend the technique from the previous part to this 2-dimensional setting. What sort of guarantees do you want? How do you do this? (There are several possible answers here!)

**Problem 6.c.** What if we don't have points in Euclidean space, but some more complicated things to compare. Imagine that the world consists of a large number of movies $(M_1, M_2, \ldots, M_k)$. A user is defined by their favorite movies. For example, my favorite movies are $(M_{73}, M_{92}, M_{124})$. Bob's favorite movies are $(M_2, M_{92})$.

One interesting question is how do you define distance? How similar or far apart are two users? One common notion looks at what fraction of their favorite movies are the same. This is known as Jaccard distance. Assume $U_1$ is the set of user 1's favorite movies, and $U_2$ is the set of user 2's favorite movies. Then we define the distance as follows:

$$d(U_1, U_2) = 1 - \frac{|U_1 \cap U_2|}{|U_1 \cup U_2|}$$

So taking the example of myself and Bob (above), the intersection of our sets is size 1, i.e., we both like movie $M_{92}$. The union of our sets is size 4. So the distance from me to Bob is $(1 - 1/4) = 3/4$. It turns out that this is a distance metric, and is quite a useful way to quantify how similar or far apart two sets are.

Now we can define a hash function on a set of preferences. The hash function is defined by a permutation $\pi$ on the set of all the movies. In fact, choose $\pi$ to be a random permutation. That is, we are ordering all the movies in a random fashion. Now we can define the hash function:

$$h_\pi(U) = \min_j(\pi[j] \in U)$$

The hash function returns the index of the first movie encountered in permutation $\pi$ that is also in the set of favourite movies $U$. For example, if $\pi$ is $\{M_{43}, M_{92}, ..., M_{124}, ..., M_{73}, ...\}$ and $U = \{M_{73}, M_{92}, M_{124}\}$, $h_\pi(U)$ will give 1 as it maps to the index of $M_{92}$ in $\pi$.

This turns out to be a pretty useful hash function: it is known as a MinHash. One useful property of a MinHash is that it maps two similar users to the same bucket. Prove the following: for any two users $U_1$ and $U_2$, if $\pi$ is chosen as a uniformly random permutation, then:

$$\Pr\left[h_\pi(U_1) = h_\pi(U_2)\right] = 1 - d(U_1, U_2)$$

The closer they are, they more likely they are in the same bucket! The further apart, the more likely they are in different buckets.