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?

Solution: [49, 99, 23, 43, 60, 25, 41]. Note that the last item insertion (i.e. 63) will generate `TableFullException`.

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?

Solution: [26,X,19,2,X,5,12]

(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?

Solution: [26,X,19,2,X,5 (deleted),12 (deleted)]

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

Solution: Consider the case when table capacity = 3, buckets 0 and 1 filled but 2 unfilled. Insertion of x where hash(x) = 0 would fail.

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?

Solution: Insertions: $m/2 + 1$; Deletions: 1

Suppose that the new inserted entry will cause $n = m$. Then the table will be resized into $m' = 4m$. When we delete an entry, the number of entry will be $n - 1 = m - 1 = m'/4 - 1 < m'/4$. The table will resize again. Hence, the minimum number of deletion before resizing is 1.

At this time, the new table size is $m'' = m'/2 = 2m$. The number of insertions before it expands is $m'' - (n - 1) = m'' - (m''/2 - 1) = m''/2 + 1$ elements. Note that the answer is calculated relative to the *current* table size.

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

Solution: For intersects, we would need to iterate through the bins in one set a and check if the element is also present in the other set b. Elements that are present in both are then placed in the result set r, which can be initialised to an appropriate capacity given that we know the sizes of a and b. Under the uniform hashing assumption, the expected complexity is $O(m_a + n_a(\frac{1}{1-\alpha_b} + \frac{1}{1-\alpha_r}))$, where m_a denotes the size of hash table a and n_a as the number of entries in hash table a, and α_a, α_r denotes the load factor of set a, r respectively.

For unions, we can iterate through the elements in set b and insert them into set a. With similar analysis as in intersects, this is in $O(m_b + n_b(\frac{1}{1-\alpha_a}))$, where m_b and n_b is defined similarly.

(b) Hash table with chaining

Solution: We can use the same strategies for intersects and unions as in 3a). The runtime of both these solutions are $O(m_a + \sum_{k \in a}(len(h_b(k)) + len(h_r(k))))$, and $O(m_b + \sum_{k \in b} len(h_a(k))$ respectively. $len(h_t(x))$ denotes the number of entries in the bucket $h_t(x)$ in table t.

In practice, this solution would be good enough, as the length of chains should be $O(1)$ (under SUHA, and an appropriate load factor). But to mitigate bad hash functions, Java implements an interesting strategy where buckets exceeding a threshold size are turned into a tree. While this increases insertion times to $O(\log size(h(x)))$, this improves searches in each *bucket* to the same complexity. Using an ordered structure like a tree also means that in the very specific scenario where the two sets' capacities and hash functions are the same, we unions and intersects can be done in exactly $O(size(a) + size(b) + m)$

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?

Solution: We may notice that most of the `increment()` invert only a few bits, while few of them invert many. With this observation, we can solve this problem using the idea of having a "bank account" for our cost. We can store little extra "money"(cost) into the account during cheap operations, and pay for expensive operations that may come later with the "balance" in the account. We also need to ensure that the bank's balance ≥ 0 at all time.

The loose upper bound for each increment is $O(k)$ which is achieved when we have $k - 1$ 1s as the LSBs (i.e. `111...110`). Since there are n increments, we have $O(nk)$ upper bound. Using amortized analysis, we are looking for a tighter bound than $O(nk)$ where n is the number of times increment is called and k is the total number of digits.

We can use accounting method to calculate the amortized cost of n increments. Note that in order for a digit to be flipped to 0, it must have been flipped to 1 first. Moreover, we want to "save" up for the more expensive operations, which is when many 1s have to be flipped to 0s.

By assigning the cost to flip of a bit from 0 to 1 to be 2 units - The intuition is that 1 unit is used to flip from 0 to 1, and another 1 unit to be saved in the bank for when the bit must be flipped from 1 to 0 in the future. Note that at each increment, there will be only 1 bit that is flipped from 0 to 1 (by construction, if the LSB bit is 0, then it will be flipped to 1 and leave the remaining bits untouched, otherwise it will carry over to the 2nd LSB bit and we can repeat the process).

This has the invariant that the amount in the bank never dips below 0. To be exact, the "bank" balance is equal to the number of 1s bit as when you flip the bit from 0 to 1, you pay in advance for the future flip back to 0. Hence the amortised cost of each increment is 2 and therefore `increment()` is $O(1)$ in amortized time and n increments is bounded by $O(n)$.

The solution above is formally called *Accounting Method*, which is very useful for amortized analysis. You will learn it in detail in CS3230.

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 $\Omega(size(v))$.

Solution: As the `rebuild` operation is performed on node v, there is an imbalance in its children. WLOG, let's assume that $size(v.left) > 2/3 \cdot size(v)$. Let w_0 be the size of each child tree after the last rebuild operation, and l, r be the number of insertions on $v.left, v.right$ respectively since.
The intuition of the proof is to imagine the "state" of the tree after **the last rebuilt** operation. How many number of nodes that we have to insert to the left of node v so that v is unbalanced? l should be at least two times the weight of the right child. Using that intuition, we can construct the following inequalities:
Since v is unbalanced, $size(v.left) > 2/3 \cdot size(v)$. From that inequality, we have $w_0 + l > 2(w_0 + r + 1) > 2w_0 + 2$ and $l > w_0 + 1$. So, $l + r > w_0 + 1$. From here, we can find a lower bound on the ratio of $l + r$ to $size(v) = 2w_0 + l + r + 1$:

$$\frac{l+r}{2w_0 + l + r + 1} \geq \frac{l+r}{2(l+r) + l + r} = \frac{1}{3}$$

As such, $l + r > \frac{1}{3} size(v)$ and hence the number of insertions on v since its last rebuild is $\Omega(size(v))$

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

Solution: In order to prove this, we need to use the fact that after every insertion, every node is balanced due to our `rebuild` procedure.
Now, we know that for every node v, $size(v.left), size(v.right) \leq 2/3 size(v)$. We can then use this to bound the height of the tree with the relation $T(size(v)) = \max\{T(size(v.left)), T(size(v.right))\} + 1 \leq T(2/3 size(v)) + 1$, thus any insertion is $O(\log n)$ deep.

(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.

Solution: At every insertion, the new node should deposit \$3 at each node it visits when it is added. The total amount deposited per insertion would then be 3 times the height of the tree, which we have shown previously to be $O(\log n)$, and the immediate cost of the traversal would also be $O(\log n)$

Suppose a rebuild is performed on the subtree rooted at node v. Then, since the last rebuild called on v, $size(v)/3$ nodes would have been inserted into the subtree, depositing $\$size(v)$ total cumulatively. This is enough to pay for the $O(n)$ cost of rebuild being performed.

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 ≤ 1, then we want them to be stored in the same bucket. Conversely, if x and y have distance ≥ 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.

Solution:

- Choose $size = 3$ and $c = 3$, then we have buckets of size 3 and we will choose a random integer in the range $[0, 2]$ as the starting point a for the first bucket.
- Then, $h(x) = floor((a + x)/3)$.
- Now look at two points x and y, and assume that $y \geq x$. Notice that the bucket containing x will begin somewhere in the range $[x - 2, x]$, and that its start point was chosen uniformly at random. These are the only 3 possible scenarios that x's bucket can be, and they occur with equal probability.
  - Scenario 1: $\{x - 2, x - 1, x\}$
  - Scenario 2: $\{x - 1, x, x + 1\}$
  - Scenario 3: $\{x, x + 1, x + 2\}$
- Consider $y - x$:
  - If $y - x = 0$, then y and x will always be in the same bucket in all 3 scenarios.
  - If $y - x = 1$, then y is in the same bucket as x if the bucket for x starts anywhere in the range $[x - 1, x]$ (scenarios 2 and 3). Since, this occurs for 2 out of the 3 total scenarios, the probability that x and y are in the same bucket is at least $2/3$.
  - If $y - x = 2$, then y is in the same bucket as x only for scenario 3. Since, this occurs for only 1 out of the 3 total scenarios, we conclude that the probability of x and y being in the same bucket is at most $1/3$, i.e., with probability at least $2/3$, they are in different buckets.
  - If $y - x \geq 3$, then y and x will always be in different buckets in all 3 scenarios.

Suppose x is in the bucket B_k.
This is how the neighbouring buckets would look like for scenario 1.

	Bucket B_{k-1}	Bucket B_k	Bucket B_{k+1}	
...	$x - 5, x - 4, x - 3$	$x - 2, x - 1, x$	$x + 1, x + 2, x + 3$	...

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!)

Solution: Solution 1: The simplest possible idea is to divide the world up into grid squares, and require that two players be hashed to the same bucket only if they are in the same grid square. (Many old games used this type of strategy!) This is simple, but you might have two players that are very close together, but are in adjacent grid squares and so are in separate buckets.

Solution 2: You might try to extend the strategy from part a. Overlay the map with 3-by-3 grid squares, but choose a random start position for the $(0, 0)$ point of the grid. The challenge, now, is that it is a bit harder to compute the probabilities because there are two different dimensions.

Consider the case where $||y - x||^2 \leq 1$ (where we want to get all points within distance 1):

- In each dimension, the distance between x and y is at most 1. From part a, we can show that the probability that x and y are in the same bucket is at least $(2/3)(2/3) = 4/9$.
- However, this is not the tightest lower bound. We can do slightly better. Let the distances in the two dimensions be d_1 and d_2 respectively. Using Pythagoras' Theorem, we know that $d_1^2 + d_2^2 = ||y - x||^2$. By extension, $d_1^2 + d_2^2 \leq 1$.
- Let the event whereby x and y are in different "buckets" in dimension i be D_i. Notice that as long as the points are in different "buckets" in at least one dimension, they will be in different buckets in the 2-dimensional grid. Thus, the probability that x and y are in different buckets is (by a union bound):

$$P(D_1 \cup D_2) = P(D_1) + P(D_2) - P(D_1 \cap D_2) \leq P(D_1) + P(D_2) = d_1/3 + d_2/3$$

- To maximize the probability that two points are in different buckets (i.e. worse case), this happens when $d_1 = d_2 = \sqrt{2}/2$. Thus, the maximum probability that they are in different buckets is $\sqrt{2}/3 \leq 1/2$. This implies that the probability of two points (with distance 1) being in the same bucket, would be at least $1/2$. If you do the calculation more carefully (without the upper bound), you get a probability of them being in the same bucket of approximately 0.53.

Solution 3: Another solution is to just draw a one dimensional line at a random angle (going through $(0, 0)$). Just like in the previous part, you can divide the line into buckets. For each point, find the closest point on the line (i.e., a projection, using trigonometry) and map the point to the associated bucket. It turns out that this works pretty well too (but we won't do the math today).

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 π on the set of all the movies. In fact, choose π 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 π that is also in the set of favourite movies U. For example, if π 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 π.

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 π is chosen as a uniformly random permutation, then:

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

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

Solution: The users U_1 and U_2 are mapped to the same bucket if $\min_j(\pi[j] \in U_1) = \min_k(\pi[k] \in U_2)$. Imagine you are iterating through the movies in order of π. Eventually, you hit a movie that is in U_1 or U_2. If that first movie you hit is in *both* sets U_1 and U_2, then they map to the same bucket. Otherwise, if that movie is only in one of the two sets, then they map to different buckets. So we have to decide how likely that first movie is to be in both sets.

In total, there are $|U_1 \cup U_2|$ items in the two sets together. And those items are ordered in a uniformly random order by π (since we chose π at random). So we are equally likely to hit any of those $|U_1 \cup U_2|$ items first.

How many of the above items are in both sets? There are exactly $|U_1 \cap U_2|$ movies in both sets. So the probability that the first item found in the enumeration is one of these $|U_1 \cap U_2|$ shared movies is $|U_1 \cap U_2|/|U_1 \cup U_2|$. That is, the probability is exactly $1 - d(U_1, U_2)$.

So we have proved that the probability that U_1 and U_2 are hashed to the same bucket is $1 - d(U_1, U_2)$.