CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 9

For: March 15–March 20

1 Review Questions

Problem 1. Quadratic Probing

Quadratic probing is another open-addressing scheme very similar to linear probing. Recall that a linear probing implementation searches the next bucket on a collision.

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
```

(a) Consider a hash table with size 7 with 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]

(b) 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)]

(c) 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 *doubled* (resize m to $2m$)

(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: $m/2$, deletions: $m/4$

Problem 3. Necessary Evils

We discussed that when using a Bloom filter, we can choose which elements to place in order to tolerate false positives or false negative. For example, we would rather tolerate false negatives for displaying online friends (i.e., having an online person appearing as offline rather than an offline person appearing as online) - and as such we would put the set of offline friends into our bloom filter. How would you use a Bloom filter in the following situations?

(a) New article filter - you want to use a Bloom filter so only unread articles are shown to users.

Solution: The Bloom filter should maintain the set of articles that have been read by the user. In this case, it's okay for our system to wrongly think that an article has been read by a user instead of otherwise

(b) Search engines - you want to use Bloom filters to help users look for relevant documents based on search terms

Solution: For each document, maintain a Bloom filter containing the search terms they contain, and use this Bloom filter to filter documents based on search terms. This means potentially irrelevant documents may be returned, but no relevant documents would be omitted.

(c) IP filter - you want to use a Bloom filter to prevent attackers on your web server (e.g. to prevent DDoS)

Solution: This could be both - the Bloom filter can either be used to keep the set of trusted IP addresses, or the set of malicious IP addresses. Typically the latter strategy is more defensive and you can wrongly turn away trusted IP addresses. The former strategy would require an additional layer of trust (e.g. CAPTCHA) and acts more like a fast rough filter.

Problem 4. 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 $O(m_a + n_a(\frac{1}{1-\alpha_b} + \frac{1}{1-\alpha_r}))$.
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 $O(m_b + n_b(\frac{1}{1-\alpha_a}))$

(b) Hash table with chaining

Solution: We can use the same strategies for intersects and unions as in 4a). The runtime of both these solutions are $O(m_a + \sum_{k \in a} len(h_b(k)))$, and $O(m_b + \sum_{k \in b} len(h_a(k))$ respectively. 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 $O(size(a) + size(b) + m)$

(c) Fingerprint hash table/Bloom filters

Solution: These are the most straightforward to implement - simply use logical AND and OR respectively for intersect and union. It should take $O(m)$ time total where m is the table size.

Problem 5. Removing Fingerprints

In this question, we will implement a delete function for our Bloom Filter/Fingerprint Hash Table (FHT).

(a) One way is by keeping count of items which hash to each fingerprint. How would you use this to implement the delete function? What are the tradeoffs and potential issues with this implementation?

Solution: Store a counter for each slot in the table, maintain on insert/delete. When checking for an entry, if its fingerprint (or any of its fingerprints in a Bloom Filter) is 0, then the entry is not present. The probability of false positives is equal to the probability of collisions.
Assuming that there can be up to m collisions, we would need $\log m$ bits to store this counter at each table row.
It is also possible in this case that we have false negatives that arise due to deletion of items which were not previously added into the set.

(b) Another way is by using another FHT to keep track of deleted entries. How would you implement this? What are some issues with this implementation?

Solution: Just add items to "tombstone" FHT on delete as you would add an item to a normal FHT. When checking for an entry, we also check if its fingerprint(s) is deleted in the "tombstone" FHT. We would get the problem of *false negatives* instead - when collisions occur, we might wrongly think that an entry has been deleted.
Compared with the counter solution, this only takes twice the amount of space.

2 Scapegoat Trees

Problem 6. 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.
Then we know that $w_0 + l > 2(w_0 + r + 1) > 2w_0 + 2$ 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, $size(v.left), size(v.right) \leq 2/3size(v)$. We can then use this to bound the height of the tree with the relation $T(size(v)) = \max(T(size(v.left)), T(size(v.right))) + 1 \leq T(2/3size(v)) + 1$, thus any insertion is $O(\log n)$ deep.

(c) Now use the previous two parts to show that the amortized cost of an insertion in a Scapegoat Tree is $O(\log n)$. *Hint:* Suppose an 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.

3 Hashing

Problem 7.
(Tabulation Hashing.)
Suppose we are creating a hash function keys $N = \log n$ bits long, mapped to buckets indexed by $M = \log m$. So, the hash function maps an N bit key to an M bit identifier for a bucket. To do this, we construct a 2D-array $T[2, N]$ and fill each entry in the table with a random M bit value.

Now to hash a key, we just XOR the key and the table:

```
hash = 0
for (j = 1 to N)
    hash = hash XOR T[key[j], j]
```

Problem 7.a. Show that for a given key k and bucket b, $\Pr[h(k) = b] = 1/m$, and that for two keys $k_1 \neq k_2$, $\Pr[h(k1) = h(k2)] \leq 1/m$

Solution: First, we will show that for each bit in the result of the hash function $hash$, $\Pr[hash[i] = 1] = \Pr[hash[i] = 0] = 1/2$. Let $hash_j$ be the value of $hash$ at the end of iteration j in the algorithm described.

1. From how the table is generated, for all $i \in [0, 1], j \in [1..N], k \in [1..M]$, $\Pr[T[i, j][k] = 0] = \Pr[T[i, j][k] = 1] = 1/2$.
2. When $j = 1$, $\Pr[hash_j[k] = 0] = \Pr[T[key[i], 1][k] = 0] = 1/2$.
3. If $\Pr[hash_{j-1}[k] = 0] = 1/2$, then $\Pr[hash_j[k] = 0] = \Pr[hash_{j-1}[k] = 1 \wedge T[i, j][k] = 1] + \Pr[hash_{j-1}[k] = 0 \wedge T[i, j][k] = 0] = 1/2$.
4. Therefore by induction $\Pr[hash_N[i] = 0] = 1/2$.

Since every bit in $hash$ is 0 with probability half, $\Pr[h(k) = b] = 1/2^M = 1/m$

To show that $\Pr[h(k_1) = h(k_2)] \leq 1/m$, there must be at least 1 bit where $k_1[i] \neq k_2[i]$. Then for this bit $\Pr[T[1, i] = T[0, i]] = 1/2^M = 1/m$. Thus, the inequality would hold.

It can also be said that since the keys differ in this bit, the m-bit hash in each bucket of the bits are independent of each other, and from here on the resulting hash would be independent. That means for each possible hash results a, b, $\Pr[h(k_1) = a, h(k_2) = b] = \Pr[h(k_1) = a]\Pr[h(k_2) = b] = 1/m^2$. So $\Pr[h(k_1) = h(k_2)] = \sum_b \Pr[h(k_1) = h(k_2) = b] = m \cdot 1/m^2 = 1/m$

Problem 7.b. How much space does this table require?

Solution: $2NM$ - 2 rows of N columns of M bits

Now let's generalize the function. Choose some integer R that divides N. (In the previous example, $R = 1$.) Now generate a 2d table of size $[2^R, N/R]$. As before, fill each entry with a

random M bit value. As before, take the hash by breaking the key into R bit chunks, look up each chunk in the table, and perform XOR:

```
hash = 0
for (j = 1 to N/R)
   hash = hash XOR T[key[(j-1)R .. jR]][j]
```

Notice now the table needs one entry for each of the 2^R possible chunks of the key.

Problem 7.c. What is the size of the table?

Solution: $MN2^R/R$

Problem 7.d. *Bonus* Here's another simple hashing scheme:

Construct a 2D binary array $A[M, N]$, where each entry is a random 0 or 1. For key k, let $h = Ak$ (think of it as matrix multiplication, where k is a column vector, A is a matrix, and the result is an m-bit column vector).

Is this the same as tabulation hashing?

Solution: It's kind of the opposite, as we construct the output bucket id one bit at a time.