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?

(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?

(c) 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 *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?

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.

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

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

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

(b) Hash table with chaining

(c) Fingerprint hash table/Bloom filters

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?

(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?

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 $\Omega(size(v))$.

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

(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.

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 $k_1 \neq k_2$, $\Pr[h(k1) = h(k2)] \leq 1/m$

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

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?

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?