CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 9

For: March 14–March 18

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 *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. Stack 2 Queue

Do you know that we actually can implement a queue using two stacks? But is it really efficient?

(a) Design an algorithm to push and pop an element from the queue.

(b) Determine the *worst case* and *amortized* runtime for each operation.

Problem 6. 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) 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 a constant amount is "deposited" at every node traversed on an insertion.

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]
```

Note that `key[j]` denotes the j-th bit of the key.

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

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?