CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 9

For: March 17–March 21

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. Implementing Union/Intersection of Sets

You are given 2 finite sets, A and B. How can you efficiently find the intersection and union of the two sets?

Problem 3. You're given an array of n integers (possibly negative), and an value k. Decide if there is a contiguous sub-array whose average value is k.

E.g. Given array $[1, 3, 2, 5, 7, 20]$, and $k = 6$. Then the answer is yes, because $[5, 7]$ has average value 6.

What is a straightfoward solution that solves this problem in $O(n^2)$ time? What is a solution that solves this in expected $O(n)$ time?

Problem 4. (Priority queue)

There are situations where, given a data set containing n unique elements, we want to know the top k highest-valued elements. A possible solution is to store all n elements first, sort the data set in $O(n \log n)$, then report the right-most k elements. This works, but we can do better.

(a) Design a data structure that supports the following operation better than $O(n \log n)$:

- `getKLargest()`: returns the top k highest-valued elements in the data set.

(b) Instead of having a static data set, you could have the data streaming in. However, your data structure must still be ready to answer queries for the top k elements efficiently. Expand or modify your data structure to support the following two operations better:

- `insertNext(x)`: adds a new item x into the data set in $O(\log k)$ time.
- `getKLargest()`: returns the current top k highest-valued elements in the data set in $O(k)$ time.

For example, if the data set contains $\{1, 13, 7, 9, 8, 4\}$ initially and we want to know the top 3 highest value elements, calling `getKLargest()` should return the values $\{13, 9, 8\}$.

Suppose we then add the number 11 into the data set by calling `insertNext(11)`. The data set now contains $\{1, 13, 7, 9, 8, 4, 11\}$ and calling `getKLargest()` should return $\{13, 11, 9\}$.

Note: we do not need to have to return the elements in sorted order.

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/enqueue` and `pop/dequeue` an element from the queue using two stacks (and nothing else).

(b) Determine the *worst case* and *amortized* runtime for each operation. Recall that if push was amortised to a cost $O(f(n))$, pop was amortised to a cost of $O(g(n))$, then after a series of t pushes and s pops, the sum total cost of the entire series of operations is at most $O(t \cdot f(n)) + O(s \cdot g(n))$.

Problem 6. Min Queue

Implement a queue (FIFO) that supports the following operations:

- `push/enqueue` - pushes/enqueues a value x
- `pop/dequeue` - pops/dequeues a value x

- `getMin` - returns the minimum value currently stored in the queue

Do this so that any sequence of t operations runs in $O(t)$ time. (I.e. the sum total cost of t operations is $O(t)$)