**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

1. 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?

**Ans:** [26, X, 19, 2, X, 5, 12]

1. 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?

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

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

**Ans:** Consider the case when table capacity = 3, buckets 0 and 1 filled but 2 unfilled. Insertion of *x* where hash(*x*) = 0 would fail. (caused by a bad hash function, need to use a good hash function)

# 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?

**Ans:**

**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?

**Ans:**

**Naïve implementation:**

Loop through every element *i* once from 1… *n*. In this outermost loop, we initialise the variables sum and count to keep track of the total array values being summed this iteration and the total number of elements that have been iterated respectively.

For each element *i*, we loop through the remaining *i*… *n* elements *j* of the array and add the current element *j* to the sum variable and increment the count variable by 1. We also calculate the current average value by taking average = sum / total, and check if the average value == *k*. If it is, we return the current *i* and *j* values. This would have a time complexity of *O*(*n*2).

**Efficient implementation:**

Firstly, we initialise the variable prefixSum with the value 0 and will be used to keep track of the total sum of the elements in the array from 1… *i*. We will also need an empty hash map to be used later. Firstly, we loop through every element *i* once from 1… *n*. For each iteration, we add the current *ith* element to the prefix sum. Then we calculate the value prefixSum – *k* \* *i* and check if this value exists in the Hashmap. If it does not exist, we hash the current value as key and the current *ith* index as the value. Suppose the value calculated has an existing key in the hashmap, then we return the key’s respective (value + 1)th element, as well as the current *ith* element in the array.

**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.

1. 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.

**Ans:** We can use QuickSelect to find the *kth* largest element in the array in O(*n*) time. The *k* largest elements are located between the *kth* largest and the last element in the array (inclusive).

1. 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.

**Ans:** We can store the largest *k* elements in a min priority queue of no more than *k* elements. After each insertion, if the priority queue contains more than *k* elements, remove the smallest element so that the priority queue size remains to be *k*. Each insertion and removal run in O(log *k*) time since the priority queue contains at most *k* elements. For getKLargest(), we extract and return all items from the priority queue, then insert them back. These can be done in O(*k* log *k*) time. Since , this approach is able to support the operations better than O().

# Problem 5. Stack 2 Queue

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

1. Design an algorithm to push/enqueue and pop/dequeue an element from the queue using two stacks (and nothing else).

**Ans:** We can make use of two stacks, with one stack (Stack 1) being the input and one stack (Stack 2) being the output. Stack 1 will be used for enqueue operations while Stack 2 will be used for dequeue operations.

During an enqueue operation, we simply push the element onto Stack 1.

During a dequeue operation, we check if Stack 2 is empty. If it is not empty, we simply pop the first element from Stack 2 and return it. If it is empty, then we move (pop then push) all elements from Stack 1 to Stack 2, reversing the order of the elements in Stack 1. Then, we pop the top element from Stack 2 to be returned.

1. 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* · *f*(*n*)) + *O*(*s* · *g*(*n*)).

For enqueue, the time complexity is O(1), since it is just a push operation onto Stack 1.

For dequeue, the amortised cost is O(1) but the worst case time complexity would be O(*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*))

**Ans:** We can store pairs (*x*, *m*), where *m* is the minimum value of the stack. This way, for getMin, all we have to do is to peek at the top of the stack and output m.

To insert x into an empty stack, insert (*x*, *x*). Else, run getMin to obtain *m* and push (*x*, min(*m*, *x*)) onto the stack.

To pop, just remove the pair.

How to implement this using a queue? Use the 2 stack solution presented earlier!