15-451/651 Algorithms, Spring 2019

Homework #2 **Due: February 5–8, 2019**

(25 pts) 1. **(The Counter Strikes Again!)** We saw in class that if a binary counter begins at 0, and we perform n increments, the amortized cost per increment is $O(1)$ (i.e., the total cost is $O(n)$). Suppose we want to be able to increment *and* decrement the counter.

(a) Show a sequence of n operations allowing both increments and decrements and starting from 0 that, without ever making the counter go negative, costs $\Omega(\log n)$ amortized per operation (i.e., $\Omega(n \log n)$ total cost).

Solution: Let $2^{b+1} \leq n \leq 2^{b+2}$. Use $2^b \leq n/2$ operations to transform the number to a 1 followed by b 0's. This takes at least 2^b work. Then alternately decrement and increment the number for the remaining $\geq 2^b$ steps. Each of these operations takes $b+1$ work. So, the total work over all these operations is at least $2^b + (b+1)2^b = (b+2)2^b$. Thus the amortized cost per operation is more than $b2^b/n \geq b2^b/2^{b+2} = b/4 = \Omega(\log n)$.

To reduce the cost observed in part (a) consider a counter using the following *redundant ternary number system*. A number is represented by a sequence of *trits*, each of which is 0, +1, or −1. The value of the number represented by $t_{k-1}, \ldots, t_0$ (where each $t_i, 0 \leq i \leq k-1$ is a trit) is defined to be

$$\sum_{i=0}^{k-1} t_i 2^i.$$

For example, $\boxed{1}\boxed{0}\boxed{-1}$ is a representation for $2^2 - 2^0 = 3$.

To increment a ternary number: You add 1 to the low order trit. If the result is 2, then it is changed to 0, and a carry is propagated to the next trit. This process is repeated until no carry results. Decrementing a number is similar. You subtract 1 from the low order trit. If it becomes -2 then it is replaced by 0, and a borrow is propagated. Note that the same number may have multiple representations (e.g., $\boxed{1}\boxed{0}\boxed{1} = \boxed{1}\boxed{1}\boxed{-1}$) — it's a *redundant* ternary number system. The cost of an increment or a decrement is the number of trits that change in the process.

(b) Starting from 0, a sequence of n increments and decrements is done. Give a clear, coherent proof that with this representation, the amortized cost per operation is $O(1)$ (i.e., the total cost for the n operations is $O(n)$).

Solution: When the number is incremented or decremented, some 1s may change to 0 and some -1s may change to 0, but a 1 will never directly change to -1 or vice versa. This means that if \$2 is paid each time a 0 is changed (to 1 or -1), there will be enough money in the bank to pay for converting it back to a 0. (Think of having a separate bank account for each digit.) The second fact is that in any increment or decrement, at most one 0 changes (to either 1 or -1). Therefore, the amortized cost per operation is ≤ 2.

(25 pts) 2. **(The Quickest Quack)**

(a) We want to maintain a *max-stack*, a data structure with the following operations.

- `push`(number x), pushes x on the stack
- `pop`, pops the top element off the stack
- `return-max`, returns the maximum number among the elements still on the stack. (Does not push or pop anything.)

How would you implement a max-stack, while maintaining constant time per operation (worst-case). Clarifications: You are allowed to allocate infinitely large arrays, etc. On an empty stack, return `NULL` on a `pop`/`return-max`. (Similar clarifications apply to the following parts.)

Solution: Have an array with a top pointer as in the usual array-based implementation of a stack, but give each element of the array two fields: one for the item and one for the max so far. When pushing a new item x, we set `A[++top].item = x` as usual, and then update the maximum by setting `A[top].max = max(A[top-1].max, x)`. By induction, we maintain the invariant that `A[i].max` is the maximum out of `A[0].item`, ..., `A[i].item` for all values of i from 0 to `top`. (And this invariant is clearly maintained on a pop as well). Thus, the `return-max` operation can just return `A[top].max`.

(b) We want to now maintain a max-queue, which is what you'd expect given the previous definition. It supports the following operations.

- `enqueue`(number x), adds x to the end of the queue
- `dequeue`, removes the element at the front of the queue
- `return-max`, returns the maximum number among the elements still in the queue. (Does not enqueue or dequeue anything.)

Show how to use two max-stacks to implement a max-queue. This max-queue should take $O(1)$ amortized time per operation. That is, a sequence of n operations (consisting of some number of `enqueue`, `dequeue` and `return-max` operations in any order) should take total time $O(n)$.

Solution: we implement the queue using two stacks exactly as in the problem from Quiz #1, using `push` and `pop` operations from part (a) to implement the dump operation (that dumps one stack into the other). This ensures that both stacks maintain the invariant from part (a), namely `A[i].max` is the maximum out of `A[0].item`, ..., `A[i].item`. To implement `return-max` we just call `return-max` on each of the two stacks and return the larger value.

(c) Consider the streaming setting, where you are making one pass over the stream $a_1, a_2, \ldots, a_n$, with each a_i being a number. You are given a number $r \geq 1$. For each time t, when you see a_t, you want to output the maximum value among the past r elements $a_{t-r+1}, a_{t-r+2}, \ldots, a_t$. E.g., if $r = 4$ and the input stream is

$$3, 17, 3, 9, 2, 0, 7, 2, 8, 9, 1, 8, 4, 5, 9$$

then the output stream should be

$$3, 17, 17, 17, 17, 9, 9, 7, 8, 9, 9, 9, 9, 8, 9$$

Give an algorithm that makes one pass over a stream of n numbers to produce the desired output stream, and the total time taken is $O(n)$. You are allowed enough space to store $O(r)$ elements. (Note that taking time $O(rn)$ would be trivial by just storing the most recent r elements and recomputing the max from scratch each time – you want to do better than that.)

Solution: We just use the queue from part (b). For the first r entries, we just do an `enqueue(x)` and `return-max`. For the rest we do `enqueue(x)`, `dequeue()` and then `return-max`.

(25 pts) 3. **(An Interesting Hash Family.)**

We say that H is *ℓ-universal over range m* (or *ℓ-wise independent*) if for every fixed sequence of ℓ distinct keys $\langle x_1, x_2, \ldots, x_\ell \rangle$, if we choose a hash function h at random from H, the sequence $\langle h(x_1), h(x_2), \ldots, h(x_\ell) \rangle$ is equally likely to be any of the m^ℓ sequences of length ℓ with elements drawn from $\{0, 1, \ldots, m-1\}$. It's easy to see that if H is 2-universal then it is universal. (*Check for yourself!*)

Consider a universe U of binary strings $s = s_1, s_2, \ldots, s_n$ of length n from an alphabet of size k. (Each character is an integer in $\{0, 1, \ldots, k-1\}$.) Hence $|U| = k^n$. Assume that $m = 2^b$.

An interesting universal family $\mathcal{G}$ (of functions from U to $\{0, \ldots, m-1\}$) can be obtained as follows. First, generate a 2-dimensional table T of b-bit random numbers; recall that $b = \lg(m)$. The first index of $T_{i,j}$ is in the range $[1, n]$ and the second index is in the range $[0, k-1]$. Now define the hash function $g_T()$ as follows:

$$g_T(s) = \bigoplus_{i=1}^{n} T_{i,s_i}$$

where "$\bigoplus$" represents the bitwise-xor function (recall, each $T_{i,j}$ is a b-bit string). The output of $g_T(s)$ is a b-bit string which is then interpreted as a number in $\{0, \ldots, m-1\}$. Note that since each choice of the table T gives a hash function g_T, and T is specified by $n \cdot k \cdot b$ bits, the family $\mathcal{G}$ consists of 2^{nkb} functions.

(a) (10 pts) Prove that $\mathcal{G}$ is *not* 4-universal.

Solution: For instance, consider the following 4 keys of length 2 (so $n = 2$): 00, 11, 01, 10. Whatever T is, these keys have the property that $g_T(00) \oplus g_T(11) \oplus g_T(01) \oplus g_T(10) = 0$ since each of the four entries in T is xor'ed twice. This means that the hash of any one of the strings can be determined from the hash of the other three, so there is no way that all 4-tuples of hash-codes are equally likely.

Hint: *To show that $\mathcal{G}$ is not 4-universal, you should exhibit 4 distinct keys $\langle x_1, x_2, x_3, x_4 \rangle$ such that if you were told the values of $g_T(x_1)$, $g_T(x_2)$, and $g_T(x_3)$, you could infer the* *value of $g_T(x_4)$ uniquely (without knowing anything else about T). This will mean that not all 4-tuples of hash-values are equally likely, since the first 3 entries in the tuple $\langle g_T(x_1), g_T(x_2), g_T(x_3), g_T(x_4)\rangle$ determined the 4th entry. You can do this using $n = 2$ and $k = 2$.*

(b) (15 pts) Prove that $\mathcal{G}$ *is* 3-universal. (You can get 7 of these points by proving the weaker statement that it is 2-universal.)

Solution: To show why $\mathcal{G}$ is 3-universal, consider three distinct keys x, y, z. We now consider two cases:

Case 1: there is some index i such that all three keys differ in that index. We now argue as follows. Consider choosing all of T except for $T_{i,*}$. Thus, filling in this row of T will determine the hash codes for x, y, and z. We first fill in T_{i,x_i}: since each value for this entry produces a different value for $g_T(x)$, we have that x is equally likely to hash to all possible b-bit values. Now $g_T(x)$ is fixed. We now fill in T_{i,y_i}: again, since each value for this entry produces a different value for $g_T(y)$, we have that y is equally likely to hash to all possible b-bit values, even conditioning on everything done so far. This means that x and y's hash values are independent. Now $g_T(x)$ and $g_T(y)$ are fixed. We now fill in T_{i,z_i}: again, since each value for this entry produces a different value for $g_T(z)$, we have that z is equally likely to hash to all possible b-bit values, even given everything so far. So all triples of hash values are equally likely.

Case 2: there is no index i such that all three keys differ in that index. In this case, choose some index i such that $x_i \neq y_i$ (this must exist since $x \neq y$). We know z_i matches one of the other two, so say without loss of generality that $z_i = y_i$. Choose some other index j such that $z_j \neq y_j$. We know that x_j matches one of those two values so say without loss of generality that $x_j = y_j$. We can now argue as in Case 1. First fill in all of T except for $T_{i,x_i}, T_{i,y_i}, T_{j,z_j}$. Now, filling in T_{i,x_i} determines x's hash (all equally likely), then filling in T_{i,y_i} determines y's hash (all equally likely, no matter what x hashed to), then filling in T_{j,z_j} determines z's hash value (all equally likely, no matter what x, y hashed to)