York University EECS 4101/5101 May 26, 2020

Homework Assignment #2
Due: June 2, 2020 at 11:30 a.m.

1. Let $m \geq 2$ be a fixed positive integer. Consider the following ADT. A state of the ADT is an m-tuple of integers $\langle v_1, v_2, \ldots, v_m \rangle$. The ADT supports two operations:

- INCREMENT(i) which increases the value of v_i by 1.
- MAX returns an index of the maximum value, that is it returns an i such that $v_i \geq v_j$ for all j.

Assume that the initial state of the ADT is $\langle 0, 0, \ldots, 0 \rangle$. We shall use m as the measure of the size of the ADT and state time complexity in terms of m.

Our goal is to design a data structure for this ADT. Each component v_i will be stored in binary in a dynamically-sized table T_i. Each entry of T_i stores a single bit of v_i. This allows each component to grow arbitrarily large, because we can resize the table to get longer and longer as the value grows. Assume $size[i]$ stores the current size of the dynamically-sized table T_i. It gets updated automatically whenever T_i is enlarged. Initially, $size[i] = 1$ for all i.

If this is all we do for the data structure, INCREMENTS will have good amortized time but MAX operations could be very expensive (as you will show in part (b)). In part (c), you will see how to make the MAX operation faster by storing some additional information in the data structure.

(a) Given two numbers represented in binary, give a simple algorithm to compare them. It should output 0 if they are the same, 1 if the first number is bigger, and -1 if the second one is bigger. If accessing a bit takes $O(1)$ time, the time required by your comparison algorithm should be linear in the lengths of the input numbers.

(b) Suppose we compute the MAX as follows.

```
1  MAX
2      $res \leftarrow 1$
3      for $i \leftarrow 2..m$
4          if (number represented in $T_i$) > (number represented in $T_{res}$) then
5              $res \leftarrow i$
6          end if
7      end for
8      return $res$
9  end MAX
```

Line 4 uses your algorithm from part (a).

Show that there is a sequence of s operations (starting from the initial state $\langle 0, 0, \ldots, 0 \rangle$) that takes a total of at least $s \cdot 2^m$ steps. This proves that the amortized cost per operation is $\Omega(2^m)$.

Hint: start by doing lots of INCREMENT operations to make the values big so that the comparisons needed by MAX will be slow.

(c) We now add a little more information to our data structure. We store the index max of a maximum component. Now a MAX operation can simply return max. But we need to do some work to update max when an INCREMENT occurs. To do so efficiently, we also store an array $diff[1..m]$, where $diff[i]$ is the index of the most significant position where component T_i differs from the maximum component. Assume the least significant bit is indexed by 0. If T_i is a maximum component, then $diff[i] = -1$. For example, if $m = 4$ and

$$\begin{aligned} T_1 &= 1001001011 \\ T_2 &= 0010010101 \\ T_3 &= 1001000110 \\ T_4 &= 1001001011 \end{aligned}$$

then, max can be either 1 or 4 and

$$\begin{aligned} \mathit{diff}[1] &= -1 \\ \mathit{diff}[2] &= 9 \\ \mathit{diff}[3] &= 3 \\ \mathit{diff}[4] &= -1 \end{aligned}$$

Give pseudocode for INCREMENT(i), including how it updates max and *diff*, and use the accounting method or the potential method to prove that the amortized time per INCREMENT operation is $O(m)$ using your scheme.

You do *not* have to give a formal proof of correctness for your algorithm but your code should be well commented so that a reader can understand how and why it works.