Problem Set 6

CSCI 3104 Spring 2014
Alex Tsankov
07/22
Partner: Cristobal Salazar

Problem 1

implementation	Add(x)	Find(x)	Remove(x)	Add(x)	Find(x)	Remove(x)
hash table	$O(1+\alpha)$	$O(1+\alpha)$	$O(1+\alpha)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Splay Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))^*$	$O(log(n))^*$	$O(log(n))^*$
Skip List	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(n)$	$O(n)$	$O(n)$
AVL Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$
Red-Black Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$

*the running time is amortized.
When $\alpha = \omega(log(n))$ we know that α will always be greater than $log(n)$ because ω is a non-strict lower bound. When plugged in it is worse than any other algorithm on this table.

Problem 2

a) The minimum is $O(n)$ bits of space. This will allow us to use a total of n possible unique hashes to store before a collision. If we need a false positive rate of 1/100 using $O(n)$ bits of space, then our answer is 100n bits.

b) In this case we want a false positive probability of $1/n$. This is similar to our answer in part a except that in part a we wanted a probability of 1/100. With our desired probability established, we simply multiply this by the function inside of Θ, our k, in this case n, we need $(n * n)$ bits.

c) This is almost exactly the same as Part B. We want our probability to be 1/n. To get this we simply need to multiply the denominator of our desired probability of 1/n by $log(n)$ to receive an answer of $O(n * log(n))$ bits of space as our worst case scenario. This differs from Part B, because our k is different, and because of it, we will require less bits.

d) This is questions is asking for two distinct answers to two different goals. G1 involves making the system unusable. G2 needs us to actually get our spam messages through the filter.

To solve G1, and make the system unusable we need to take advantage of the property of Bloom Filters which says that the larger the set of data, the more false positives are received. While its impossible to make the system **totally** unusable without a simple DDoS attack,because no matter how many false positives we force it to get, there is still a chance it can actually get it right, we can simply send millions of spam messages in such a way that the table behind the Bloom Filter becomes totally inundated with illegitimate spam messages and returning a majority of false-positives .

To solve G2 we first need to understand that, by definition, a bloom filter can't return any false negatives, which means that we can't have our filter identify a non-spam message as spam. To get around this, we need to invert our filter. What this requires is for us to build our blacklist in the beginning with hundreds of non-spam messages, where spam messages should be. With an inverted table, we will start treating non-spam messages like spam, and vice-versa, leading to an ultimately ineffective table.

Problem 3

a) Given that an element A[x] is a local minimum if it is less than or equal to both its neighbours, we know that there must be at least 1 local minimum in an array due to the boundary condition. That is, the elements on the end of the array only have 1 neighbour. This means even if all of the elements are in ascending order (i.e. [1, 2, 3, 4]), then the first element is a local minimum because it only has 1 neighbor. If we change $2 \to 0$ in the example above, we get the array, [1, 0, 3, 4], which also has 1 local minimum at the second element. We can keep going until the n^th element and we will continue to get the pattern of at least 1 local minimum.

b) For this algorithm we will want to use a Divide-and-Conquer method (similar to Binary Search) to find a local minima in $O(log(n))$ time. We will first start out with a variable $i = n/2$ (mid-point). We will use this variable to start, and we will have three cases.
Case 1: if($A[i-1] > A[i]\&\&A[i+1] < A[i]$), return A[i], it must be the local minimum.
Case 2: if($A[i-1] < A[i]$), then the left half of the array must contain a local minimum, so then recurse on the left half. We assume that A[k] is not a local minimum for each $0 <= k < i$. Then A[i-1] is not a local minimum, which implies that $A[i-2] < A[i-1]$. Similarly, $A[i-3] < A[i-2]$. We can continue this pattern until we reach $A[0] < A[1]$.
Case 3: if($A[i+1] > A[i]$). Then the right half of the array must contain a local minimum, so recurse right. This is the same as Case 2 except to the right.

Problem 4

a) The idea behind our solution to the pancake algorithm is to get the largest pancake to the top, something that we can guarantee if we flip the stack above it, then flip the entire stack so that we know for a fact that our largest will be on the true bottom. We then repeat this process with the next largest pancake and so forth until our stack is sorted. Our steps would be as follows:

Step 1: Scan the stack and identify the largest pan cake.
Step 2: Put the spatula under the largest pancake and flip everything above it. This will guarantee our largest pancake is on top.
Step 3: With the largest pancake on top, flip the **entire stack** so that the largest pancake is on the bottom.
Step 4: Repeat step 2 on next largest pancake.
Step 5: Flip the whole stack **one above the bottom**. The most recently flipped stack then becomes our bottom pancake and we can repeat the previous steps up the chain until we have a sorted stack. Stop once the stack is fully sorted.

To start analysing our theorem lets represent our pancakes as integers, with a bigger number representing a bigger pancake. Lets assume our stack of pancakes is three tall with the values [1 3 2] in this case, the first flip (Step 2) would transform the stack into [3 1 2] and with the second flip (Step 3) we would get [2 1 3]. Repeating this step again we would flip 2, to achieve an unchanged stack [2 1 3], because we know that we have our largest pancake on the bottom we would then flip the second lowest pancake to achieve [1 2 3], our final answer.

To prove that this is correct, lets assume that we have a pancake stack of k size. We would identify the largest pancake, i and flip it and everything above it which we know would place i on the top. With this guaranteed position of i at the top, we could then flip the whole stack so that i is at the bottom. The next step would involve identifying $i-1$, we would find it, flip everything above it so that $i-1$ is at the top, and then flip everything above i, our biggest pancake, so that $i-1$ is directly above i. This continues until we reach pancake $i - i + 1$, our smallest pancake, which would be our base case of an already sorted stack so that we can stop and return what we have.

b) In the worst case scenario, we have to do at most $2n - 3$ flips. This stems from the fact that for every large pancake we have to flip the stack twice, with the 3 being subtracted stemming from some flips we can shave off near the end.

c) In the worst case burnt-pancake scenario, we have to do at most $3n/2$ flips because we have to do an extra set of flips for the burnt side of the pancakes.

Sources

- hhttp://www.math.uiuc.edu/ west/openp/pancake.html
- http://mypages.valdosta.edu/dgibson/courses/cs3410/notes/ch08.pdf