Problem Set Six

Zoe Farmer
Jeremy Granger
Ryan Roden

February 26, 2024

1. Complete the following table for the listed implementations of a dictionary ADT. You will need to look up each of these implementations yourself and determine the running times for each operation, in each case. Then, explain why hash tables become poor choices relative to these alternatives when $\alpha = \omega(\log n)$.

   (a)

	Average Case			Worst Case		
Implementation	`Add(x)`	`Find(x)`	`Remove(x)`	`Add(x)`	`Find(x)`	`Remove(x)`
Hash Table	$O(1+\alpha)$	$O(1+\alpha)$	$O(1+\alpha)$	$O(n)$	$O(n)$	$O(n)$
Splay Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	Amortized $O(\log n)$	Amortized $O(\log n)$	Amortized $O(\log n)$
Skip Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

   $\Omega(\log n)$ is a non-strict lower bound. When alpha equals $\Omega(\log n)$, this is saying that alpha running time will always be greater (in some cases, it could be *much* greater) than $\log n$. Therefore, when $\Omega(\log n)$ is inserted in the hash table's average running time, the time becomes $O(1 + \Omega(\log n)) \Rightarrow O(\Omega(\log n))$, which is (potentially) a far worse running time than any of the other choices in our table.

2. Acme Corp. now wants Professor Flitwick to help them develop a new email application, with a user-specific "blacklist" of n messages that are spam.

   Here's what Acme wants. When a new message arrives in a user's mail queue, the app needs to test whether the message is in the user's blacklist. (If so, it will then mark it as spam.) However, space is at a premium and Acme doesn't want to literally store all n spam messages in memory (assume each message is many bits in length).

   Flitwick's proposal is to use a Bloom filter, a probabilistic data structure with k hash functions that can be used to test whether an element is in a set. Assume that each hash function uses an its own binary array of length l, can map a string of arbitrary length (an email) to some bit within this array, and is a uniform hash. Under this scheme, a false positive occurs when a non-spam message is incorrectly labeled as spam, i.e., is hashed to the same location as one of the n spam messages.

   (a) For a false positive rate of no more than $f = 1/100$, what is the minimum number of bits of space needed to guarantee that Flitwick can test in $O(1)$ time whether a new message is in the blacklist?

i. We know that the formula for the rate of false positives is

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

$$\begin{aligned}\text{Assuming } k = 1 \\ 1 - e^{-n/m} &= \frac{1}{100} \\ m &= \frac{-n}{\ln\left(1 - \frac{1}{100}\right)} \approx \boxed{100n}\end{aligned}$$

If we assume that $k = O(1)$, in other words our k is constant and we have a constant number of hash functions, we can solve for m and obtain $100n$. Assuming that our k hash functions have equal probability of hashing to a random position in our array, m, we know that the chance of any index i getting chosen is $\frac{1}{m}$. In order to guarantee testing any given email in $O(1)$ time with error rate less than $\frac{1}{100}$ you must have have the length of the array as $100n$. This is apparent because the chance now for any index to be chosen is $\frac{1}{100n}$, which keeps the error rate below the given.

(b) Assume that Flitwick uses $k = \Theta(1)$ hash functions. Using asymptotic notation, how many bits will Flitwick need if he wants the probability of false positives to be no more than $f = 1/n$?

i. This is merely a modification of our above computation. We declared the rate of false positives, and then when assuming $k = 1$ we determined our length of m. We can now generalize for any given case.

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

$$\begin{aligned}\text{Assuming } k = 1 \\ 1 - e^{-n/m} &= \frac{1}{n} \\ m &= O\left(\frac{-n}{\ln\left(1 - \frac{1}{n}\right)}\right)\end{aligned}$$

(c) Assume that Flitwick uses $k = \Theta(\lg n)$ hash functions. Using asymptotic notation, how many bits will he need if he wants the probability of false positives to be no more than $f = 1/n$? Explain why this answer differs from that of 2b.

i. This is a further generalization of our previous equations, as we're no longer assuming that $k = 1$, and instead it is replaced by a fuction of n.

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

$$\text{Assuming } k = \lg n$$

$$\left(1 - e^{\frac{-n \lg n}{m}}\right)^{\lg n} = \frac{1}{n}$$

$$\therefore m = \frac{-n \lg n}{\ln\left(1 - \left(\frac{1}{n}\right)^{\frac{1}{\lg n}}\right)}$$

(d) Suppose the adversary knows that Acme is using Flitwick's scheme, but does not have access to the hash functions being used. Describe a sequence of emails that would defeat the Bloom filter approach either by rendering the system unusable or by getting spam messages through the filter.

i. There are two main strategies to beating a Bloom filter with unknown hash functions. The first is *Saturation*, in which we defeat the filter by creating a sequence of emails that max out the filter itself. By default, a filter is useless if it filters everything, so if we flooded it with unique emails that were all "marked as spam" the filter would add all of them, at which point it would start to automatically filter everything, and the filter would only find false positives. This strategy relies on each successive email being radically different than the prior one, as that leads to the highest chance of the new email not hashing to just filled entries in m.
The other strategy is a little trickier and relies on some information being known about the filtering techniques. Even if we don't know the hash functions themselves, if we can reproduce the filter's conditions "black box" style then we can take a more direct approach and determine which emails get past the filter. This strategy relies on much more information being known, and as a result is less likely to be used.

3. Professor Dumbledore gives you an array $A[1 \ldots n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. He instructs you that an element $A[x]$ is a local minimum if it is less than or equal to both its neighbors, or more formally, if $A[x1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are five local minima (each boldfaced) in the following array:

9	7	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9

Even a neophyte Algorithms wizard can find a local minimum in $O(n)$ time by scanning through the array. Dumbledore asks you to instead (a) prove that A must contain at least one local minimum, and (b) describe, analyze and prove the correctness for an algorithm that always finds a single such local minimum in $O(\log n)$ time.

(a) To answer this we can look at the Extreme Value Theorem[1]

[1]

Theorem 1 (Extreme Value Theorem)**.** *If a real-valued function f is continuous in the closed and bounded*

Even though this applies solely to functions, we can define our own theorem that states that for any array of numerical values there will be at least one minimum and maximum. In essence, this is provable in that any sorted array of numbers has to start at a certain value, and has to end at a certain value. There is no such case in which a numerical array does not have a minimum and a maximum.

(b) We first declare x to be a global variable starting with null value. The first line always tests if x is non-null and then returns ceasing all recursion while sending x back up the tree. If a local min has not been found, test the middle index of the array argument for function loc_min. If this is a local min, set it to x and return x. If the middle index is not a local min and we have traversed down to the left most leaf, then we know we will not find a local min in this section of the array, so return x still set as null. If all of these base cases have still not returned anything, then recurse left and right.
Our algorithm is listed below.

```
x = NULL
def loc_min(A[1:n]):
    if x != NULL or len(A) <= 2:
        return x
    # Helper function of O(1) time testing if
    # element is a local minimum
    if test_loc_min(A[len(A) / 2]):
        return x = A[len(A) / 2]
    # If recursion traverses down to left most
    # leaf and doesn't find local min, return
    # x as null and start checking right branches
    loc_min(A[1:n/2])
    loc_min(A[n/2:n])
```

4. Recall the pancake problem from the midterm, in which we have a stack of n pancakes of different sizes that we want to sort so that smaller pancakes are on top of larger pancakes. The only operation we can perform to change the ordering is a flip: insert a spatula under the top k pancakes, for some k between 1 and n, and flip them all over.

(a) Describe an algorithm based on Insertion Sort that sorts an arbitrary stack of n pancakes, and prove that your algorithm is correct.

i.

```
i = 0
A[] //pancake stack

for (i; i < len(A) - 2; i++) {
    if (A[i+1] > A[i]) {
        key_value = A[i+1]
        key_index = i+1
        flip(i+1) //reverses order of first i+1 pancakes
```

interval $[a, b]$, then f must attain a maximum and a minimum, each at least once.
http://en.wikipedia.org/wiki/Extreme_value_theorem

```
            for (j=1; j < key_index AND A[j] > key_value; j++) {
                tmp_flip = j
            }

            flip(tmp_flip)
            flip(tmp_flip - 1)
            flip(key_index)
        }
    }
```

(b) In the worst case, exactly how many flips does your algorithm perform?

i. Even though worst case runtime is $O(n^2)$, the pancake flips are only dependent on the outer loop, $O(n)$ so if there are 4 flips/iteration, then $4(len(A) - 1)$ flips take place for the worst case time.

(c) Suppose one side of each pancake is burned. In the worst case, exactly how many flips do you need to sort the pancakes and have the burned side of every pancake on the bottom?

i. When you move the key pancake to the top of the stack, you want to have the burned side up (so that when it is flipped into the sorted position the burned side is now facing down). This can only add upto 1 extra flip from part (b), so the worst case is now $5(len(A) - 1)$ flips.