CSC263 Winter 2020 – Problem Set 2

Remember to write your **full name** and **student number** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Vivekanand Gadhia — 1004165253, Piotr Szaran — 1004171424 **Thursday, February 27th, 2020**

---

1. [**total: 6**] You are given an array `A` that contains n **distinct** numbers. The task is to compute another array `B` such that for all `j`, we want `B[j]` to be the number of elements in `A` that appear after `A[j]` and are strictly smaller than `A[j]`.

   As an example, if you are given the input array `A = [8, 10, 4, 7, 3, 5, 9, 1]`, then you want to output the array `B = [5, 6, 2, 3, 1, 1, 1, 0]`. This is because, in `A`, there are 5 elements that appear after `A[0] = 8` and are smaller than `8` (i.e. `4, 7, 3, 5, 1`), there are 6 elements that appear after `A[1] = 10` and are smaller than `10` (i.e. `4, 7, 3, 5, 9, 1`), etc.

   Design an algorithm to compute `B` given `A` as input. The **worst case** complexity of your algorithm **must be** $\mathcal{O}(n \log n)$.

   This algorithm can be described in brief given what we have learned in class.

   Write a clear description of your algorithm (what data structure is used and how the algorithm utilizes the data structure) and justify its correctness and runtime. As usual, please use learned algorithms and analysis results from lectures and tutorials without repeating them.

The data structure we decided to use was a Balanced BST, but for every node in the tree we include the size of the subtree of that node.

The way our algorithm work is by initially traversing the inputted array from back to front inserting every key into the tree while maintaining the Balanced BST properties. As we insert every key into the tree, we compare the size of the key being inserted with the root of the tree.

- If the new node is larger, then we know that the minimum size of the new node will be the size of the roots left subtree, but we still have to traverse down the right subtree adding one for every node that is smaller than the one being inserted.

- If the new node is smaller, the new nodes size starts at zero, and we traverse down the left subtree adding one to the size for every node that is smaller.

This algorithm will yield a runtime of **O(nlogn)**, as required.

2. **[total: 6]** Suppose that we want to generate a BST by inserting the keys 1 through n into an initially empty BST (assume $n > 1$). Assume that the insert sequence is a random permutation of $[1, 2, 3, \ldots, n]$ and each permutation is equally likely to occur. Answer the following questions.

   (a) What is the maximum possible height of the generated BST? Describe **four** different insert sequences that would result in a BST with the maximum height.

   The maximum possible height of a generated BST is 'n'.

   - $[n, (n-1), (n-2), ..., 3, 2, 1]$
   - $[1, (n), (n-1), (n-2), ..., 3, 2]$
   - $[1, 2, 3, ..., (n-2), (n-1), n]$
   - $[n, 1, 2, 3, ..., (n-2), (n-1)]$

   (b) By picking a random permutation of $[1, 2, 3, \ldots, n]$ as the insert sequence, what is the probability that the resulting BST has the maximum height? Show detailed steps of your calculation with clear justification.

   In order to generate a BST with a maximum possible height, the root node must either be the smallest number or the largest number. By doing this one child of the root will be empty, as required to achieve a maximum height. This implies that that there are 2C1 ways to pic the first number in the sequence.

   After choosing either 1 or n to be the first number in the sequence, we are left with either [2,3, ... , (n-1), n] or [1, 2, 3, ... ,(n-1)]. However, the idea from above can be generalized to subtrees. So we pick one out of the next largest and smallest numbers. This also has 2C1 ways.

   Applying this idea until we are out of keys to insert give us: $(2C1)^{n-1}$ ways, but 2C1 = 2! = 2 $\implies 2^{n-1}$.

   Further, there are n! ways to permute n objects, **therefore the probability** is $\frac{2^{n-1}}{n!}$ for n > 1

3. [**total: 12**] In this problem, you will design the data structure for implementing an ADT called `RESTAURANT-SET`. Below is the description of the ADT.

Objects: A collection of restaurants in the same city. Each `restaurant` has the following attributes:

- `restaurant.name`: a string which is the name of the restaurant. Each restaurant has a unique name, i.e., no two restaurants have the same name.
- `restaurant.cost`: a decimal value such as 22.99 that denotes the cost of a 1 person meal at the restaurant. For simplicity, assume that each restaurant's cost is unique, i.e., no two restaurants have the same cost.
- `restaurant.rating`: a decimal value between 0.0 and 5.0, e.g., 0.012, 3.1415926. For simplicity, assume that each restaurant's rating value is unique, i.e., no two restaurants have the same rating value.

Operations:

- `GET-RESTAURANT(R, name)`: returns the restaurant with name `name` if it exists in the `RESTAURANT-SET R`; returns `NIL` if the restaurant does not exist in `R`.
- `ADD-RESTAURANT(R, restaurant)`: Add a restaurant `restaurant` to the `RESTAURANT-SET R`. You may assume that `restaurant` has a unique name, a unique cost and a unique rating.
- `AVG-COST(R, r1, r2)`, `r1` $\leq$ `r2`: considers all restaurants that have a rating `r` such that `r1` $\leq$ `r` $<$ `r2`, and returns the average of the meal costs over all these restaurants.

Requirements: Let n be the size of R,

- `GET-RESTAURANT(R, name)` must have average-case runtime $\mathcal{O}(1)$.
- `ADD-RESTAURANT(R, restaurant)` must have **average-case** runtime $\mathcal{O}(\log n)$.
- `AVG-COST(R, r1, r2)` must have worst-case runtime $\mathcal{O}(\log n)$.

Give a *detailed* description of the design of your data structure by answering the following questions.

(a) [**2**] Which data structure(s) do you use in your design? What is the key of each data structure? What attributes does each node in your data structure(s) keep?

We use two data structures in our code are **Hash tables and AVL's**

- Hash tables are used so that we can get an average case O(1) run time for GET-RESTAURANT by using a hash search which was shown in class to have average csae O(1).
- Balanced BST are built based on the restaurant ratings. Our balanced BST's are augmented so each node keeps track of the name, cost, and subtree's total cost. We use balanced BST so that the ADD-RESTAURANT function can have a average-case runtime of O(logn) because we showed in class that inserting into a balanced BST has this average-case runtime.

(b) [**2**] Explain concisely in English how your `GET-RESTAURANT` operation works, and justify its runtime.

Reiterating what was said above, our GET-RESTAURANT operation takes the input given by the user and by using a hash function will instantly return the name if the restaurant exists, or NIL if it does not. We can guarantee this has an average-case runtime of O(1) because of what we learned in lecture about simple uniform hashing.

(c) [**2**] Explain concisely in English how your `ADD-RESTAURANT` operation works, and justify its runtime. In particular, explain why all the attributes kept in each node can be maintained efficiently upon the addition of the new node.

The ADD-RESTAURANT operation works like regular Balanced BST, with the node values being the rating of each restaurant (since no two rating can be the same by assumption). Therefore this function has the runtime of adding into a balanced BST, which was shown in class to have average-case runtime of O(logn).

(d) [**6**] Explain how your `AVG-COST` operation works and **write the pseudocode** of this operation, then justify its runtime.

```
 1 #Main Function
 2 def AVG-COST(R, r1, r2):
 3     if r1 < R.root.rating and r2<R.root.rating:
 4         return AVG-COST(R.root.left, r1, r2)
 5     else if r1 > R.root.rating and r2 > R.root.rating:
 6         return AVG-COST(R.root.right, r1, r2)
 7     else:
 8         min = search_min(R.root.left, r1)
 9         max = search_max(R.root.right, r2)
10         return (min[1] + max[1] + R.root.price)/(min[0]+max[0] + 1)
11 
12 #Helper Functions
13 def search_min(R, r1):
14     '''
15   search_min returns a tuple (number of elements in subtree,
16   total price of subtree) greater than/equal r1
17   '''
18   if R.root.rating > r1:
19     tuple = search_min(R.root.left, r1)
20     return (tuple[0]+(R.root.right.size)+1, tuple[1]+R.root.right.total+R.root.price)
21   else if R.root.left.rating < r1:
22     return search_min(R.root.right, r1)
23   else:
24     return ((R.root.right.size)+1, R.root.right.total+R.root.price)
25 
26 def search_max(R, r2):
27     '''
28   search_max returns a tuple (number of elements in subtree,
29   total price of subtree) less than r2
30   '''
31   if R.root.rating > r2:
32     return search_max(R.root.left, r2)
33   else if R.root.left.rating < r2:
34     tuple = search_max(R.root.right, r2)
35     return (tuple[0]+(R.root.left.size)+1, tuple[1]+R.root.left.total+R.root.price)
36   else:
37     return ((R.root.left.size)+1, R.root.left.total+R.root.price)
```

- The function AVG-COST is a modified version of search on an AVL tree and has been proven to have a runtime of O(log n).
- The functions search_ min and search_ max are proven in class to have a worst-case runtime of O(log n) respectively.
- Therefore the total runtime of the function AVG-COST can be determined by adding the runtimes of the separate calls. The calls made on line 4, 6, 8, 9 all have a runtime of O(log n).
- $O(logn) + O(logn) + O(logn) + O(logn) \implies 4O(logn)$ a worst case runtime of O(log n) as required.

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In each problem set, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

4. [**total: 12**] In this problem, we will deal with the notion of **pseudo-similar** triangles.

   Each triangle is represented by a 3-tuple of positive numbers, specifying the sides of the triangle.

   We say that two triangles `t1` and `t2` are **pseudo-similar** if triangle `t2` can be obtained by rotating and reflecting triangle `t1`.

   For example, `t1 = (6, 9, 12)` is **pseudo-similar** to `t2 = (6, 12, 9)` since `t2` can be obtained by rotating and reflecting `t1`.

   $$\texttt{(6, 9, 12)} \xrightarrow{\text{rotate}} \texttt{(9, 12, 6)} \xrightarrow{\text{reflect}} \texttt{(6, 12, 9)}$$

   However, `t1 = (6, 9, 6)` is not **pseudo-similar** to `t2 = (6, 9, 9)`.

   We say that two triangles are the same **kind** if they are **pseudo-similar**.

   In Python, a triangle will be represented as a 3-tuple of positive integers.

   Your task is to write the function `num_triangle_kinds`, which determines the **number** of different kinds of triangles in the list. **(8 points)**

   Requirements:

   - Your code must be written in Python 3, and the filename must be `triangle.py`.
   - We will grade only the `num_triangle_kinds` function; please do not change its signature in the starter code. include as many helper functions as you wish.
   - You are **not** allowed to use the built-in Python dictionary or set.
   - To get full marks, your algorithm must have average-case runtime $\mathcal{O}(n)$. You can assume Simple Uniform Random Hashing.

   Write-up (4 points): in your `ps3.pdf/ps3.tex` files, include the following: an explanation of how your code works, justification of correctness, and justification of desired $\mathcal{O}(n)$ average-case runtime.

For our code we used simple uniform hashing, the multiplication method to be specific.

For our constants, we let our A = $\frac{\sqrt{5}-1}{2}$ (as suggested by Donald Knuth), and $m = 2^{32}$. However, since $2^{32} \approx$ 4.3 billion, and the multiplication method hashes a number between 1 and (m-1), the possibility of two numbers being hashed to the same number is so small that we can assume our average case operations are all O(1).

Ultimately our hash equation looked like: h(k) = $m * (kA(\text{mod } 1))$, and we let k be each triangle side, and then add and return h(side 1) + h(side 2) + h(side 3). This is so triangles (1,2,3) and (2,2,2) do not have the same hash, but triangles (3,6,12) and (12,3,6) do have the same hash.

How our code works, is we begin by looping through every triangle and hashing each triangle. If the hashed triangle doesn't already exist in our list, we add it, otherwise we are just looking at a pseudo-similar triangle and there is no need to add it to the list. The loop runs until every triangle is read and once the loop is done we return the length of the list.

Since the operations in the loop take O(1) average-case time, and our loop runs 'n' times **we get a runtime of O(n)**.