CS301 Algorithms - Homework 2

Ege Demirci - 28287

Sabanci University, Spring 2022-2023, Algorithms

1. Question - 1

1.1. Part - A

Firstly, my algorithm would find the m-th grade in the grades list using the worst-case linear selection algorithm. (Normally WCL would find the m-th element from the end of the list, but let's say this WCL variation can find the m-th element from the beginning.) It then iterates over the list of students and add all students with grade greater than m-th grade to selected ones. Then it would iterate over the list of students again and add all the students with grade equal to m-th grade to tied ones. Then I would shuffle the tied students list to select randomly. Then I would iterate over to tied students and add students from this list to selected ones until I reach the number of missing students. Since there is shuffling, ties are broken randomly. The algorithm has a time complexity of O(n), which is linear.

Here's the full algorithm:
1 - Create an empty list for selected students
2 - Use the WCL selection algorithm to find the m-th grade in grades list.
3 - Iterate over all students and add those with a grade greater than the m-th grade to selected students
4 - Create a list tied students containing the students with the grades that equal to m-th grade
5 - Shuffle tied students
6 - Select the first (m - length of selected students) students from tied students and add them to selected students
7 - Return selected students

1.2. Part - B

For the time complexity part, let's analyze the algorithm step by step:
1 - Create an empty list for selected students: O(1)
2 - Use the WCL selection algorithm to find the m-th grade in grades list: O(n)
3 - Iterate over all students and add those with a grade greater than the m-th grade to selected students: O(n)
4 - Create a list tied students containing the students with the grades that equal to m-th grade: O(n)
5 - Shuffle tied students: O(k) where k is the number of students with the m-th grade
6 - Select the first (m - length of selected students) students from tied students and add them to selected students: O(m - len(selected students))
7 - Returning selected students: O(1)

The overall time complexity of the algorithm would be O(n + m − numselected + k), which is equivalent to O(n) in the worst case when all students have the same grade, since k and numselected are both bounded by n. So upper bound for the run time complexity of my algorithm is O(n) which is linear.

2. Question - 2

The decision problem for the given problem:

Given your friends $F = f_1, f_2, \ldots, f_n$ and for each pair of your friends $f_i, f_j \in F$, whether f_i and f_j know each other or not, and an integer $k \leq n$, is it possible to write k or less than k different messages, so that no two of your friends who know each other will get the same message?

3. Question - 3

3.1. Part - A

This statement should be **true**, we can explain this question using the cases explained in the lecture.

At the start of our hypothetical cases; x is the added node so it's the child, p is the parent of x, u is the uncle of x, and g is the parent of p and u.

Case 1 - Case 1 Symmetric: In this case, the uncle (these nodes are male) of the child in the red-red pair is also red, and x is the right or left child of p. In this case, we have a red-red pair problem, and in this case, we're just pushing the red-red pair to the root of the tree, using recoloring operation. In the worst case, case 1 may keep applying. We will be pushing the red-red pair problem closer to the root in every application of case 1. In this case, we will not need any rotation since we're solving the issue by recoloring. So for this case since we can simply say there is no rotations, the number of rotations is constant and this is equivalent to saying that this case requires O(1) rotations.

Case 3 - Case 3 Symmetric: In this case, x is the left child of p, p is the left child of g, u is black (Case 3) or x is the right child of p, p is the right child of g, u is black (Case 3 Symmetric). In the first case, we're solving the issue by right rotation and re-coloring the p and g, or in the second case, we're solving the issue by left rotation and re-coloring p and g. In both cases, we only need single rotation so we can simply say the number of rotations is constant and this is equivalent to saying that this case requires O(1) rotations.

Case 2 - Case 2 Symmetric: In this case, x is right child of p, p is left child of g, uncle is black (Case 2) or x is left child of p, p is right child of g, uncle is black (Case 2 - Symmetric). This is actually quite similar to Case 3 by its logic. For the first case, we're doing left rotation operation but it's not enough to solve the problem. In this case our problem simply become Case 3 and we're solving it by right rotation. For the symmetric case, we're doing right rotation operation but it's not enough to solve the problem. In this case our problem simply become Case 3 Symmetric and we're solving it by left rotation operation. Again there will be coloring operations at the end but not necessary to mention for now. In both cases, we only needed double rotation so we can simply say the number of rotations is 2 and since he number of rotations is constant and this is equivalent to saying that this case requires O(1) rotations.

So it's **true** that a red-black tree insertion requires O(1) rotations in the worst case.

3.2. Part - B

This statement should be **false**, we can explain this question using the cases explained in the lecture.

At the start of our hypothetical cases; x is the added node so it's the child, p is the parent of x, u is the uncle of x, and g is the parent of p and u.

Case 1 - Case 1 Symmetric : In this case, the uncle (these nodes are male) of the child in the red-red pair is also red, and x is the right or left child of p. In this case, we have a red-red pair problem, and in this case, we're just pushing the red-red pair to the root of the tree, using recoloring operation. In the worst case, case 1 may keep applying. We will be pushing red-red pair problem closer to the root in every application of case 1. Since in every application we're doing recoloring and we're pushing to the root we will need O(h) (h is the height of the tree) node recoloring to resolve the problem. h is dependent on the size of the tree so it's not constant.

Therefore, in the worst-case, a red-black tree insertion requires O(h) node recolorings, where h is the height of the RBT.

It's not necessary to explain Case 2 and Case 3 for this case since Case 1 already proved that this statement is false.

So it's **false** that a red-black tree insertion requires O(1) node recoloring in the worst case.

3.3. Part - C

This statement should be **false**.

The reason is pre-order traversal for a red-black tree with n nodes would take O(n) time, not (n lg n). The reason for this lies on the logic of the pre-order travelsal. The pre-order traversal means visiting each node in the tree exactly once, and performs a constant amount of work (printing etc.) at each node. So it will be always proportional to the number of nodes in the tree, which is n. Therefore, the time complexity of pre-order traversal of a red-black tree is O(n).

So it's **false** that walking a red-black tree with n nodes in pre-order takes $\Theta(nlgn)$.

4. Question - 4

4.1. Part - A

NP stands for nondeterministically polynomial.

4.2. Part - B

The decision problem is said to be in NP if there exists a polynomial-time algorithm that can verify the correctness of a proposed solution to the problem. More formally, given a guess g as the solution of the problem P, if the solution g can be checked for being a solution to P in time $O(n^k)$, then P is said to be an NP problem.