Homework 6: Hash Maps, Proofs, and Trees

Data Structures and Algorithms Spring 2020

Sparsh Bansal

1 Question 1

1.1 Part a

For the updated algorithm, I want to pre-process one of the two pieces of the given data (the data with information about the list of students taking each of the classes at Olin). The post-processed data must be capable of being indexed in a hash map data structure with each of the students as keys, mapping to each of the classes that they are taking. The reason for this is to facilitate the process of acquiring all the classmates of the current suspected student by iterating through all the classes that this student is taking.

The new algorithm is as follows: I initialize an empty list (AllSuspects) in which I store all the suspected students for being infected by Acedimitis. I use the piece of data that is given, the 0^{th} infected student by enqueue-ing it to a newly defined Queue structure (Suspects), and the list of all students at Olin, by creating a hash table (Status) with all the students as keys that map to the string 'negative'. Then, while this queue has a length of greater than 0, I do the following:

(I) dequeue an element

(II) Iterate through all of the classmates of this student using the pre-processed data hast table :- constant runtime (Worst case scenario would be that each class has 25 students, O(125). This is the reason as to why I have assumed this operation will be completed in constant runtime.)

(III) For each of the classmates, check to see if they have been marked as Academitis 'positive' or 'negative' in hash table Status

(IV-A) If the classmate is marked 'negative,' mark them as positive and enqueue them to Suspects and to AllSuspects

(IV-B) If the classmate is marked 'positive,' do nothing

The total runtime for steps I through IV-A take a linear runtime, O(n). The worst case being that we need to queue each of the classmates of a suspected student, iterate through each of their classmates, and maintain Suspects, AllSuspects and Status.

1.2 Part b

Suppose that a student caught acedemitis but was not added to the list. Since the algorithm adds suspected students based on their suspected classmates, consider the first time that not all suspected students are added. Then, this student is not queued up in Suspects, and as the dequeue process goes on, this students' classmates will not be checked as suspected students. But, according to the algorithm, the classmates of each suspected student are being checked for 'positive' or 'negaitve,' which leads to a contradiction.

2 Question 2

2.1 Inductive Hypothesis

Given that the Merge Sort algorithm returns a sorted list when applied on a list of length l, doubling the length of the input list to 2l would return a sorted list.

2.2 Base Case

The base case is when the length of the subarray at the lowest level is 1.

2.3 Inductive Step

If l = 1, the algorithm satisfies the base case (a subarray of length 1 is already sorted). When l is 2, or when we merge two of these subarrays together, they switch or maintain their location based on their magnitudes so as to sort in ascending order. Towards the end of this process, these two subarrays are appended into a bigger array which, in this case, forms a list that is the size of the input list. Because this list contains the same elements as the input list, but now sorted through the merge sort algorithm, the inductive hypothesis holds for longer arrays.

3 Question 3

3.1 Part a

3.1.1 make-set

To create a set with a single node i, we can initialize a double linked list with the input node as the initial node. This node would point to a tail (The next item in the list), which will be None in this case. The runtime for this operation would be constant O(1).

We can make a modification to the orginal node class (which we constructed for a previous homework assignment) by giving it the ability to not just hold the value of the node and pointers to the next and previous nodes, but also the head to which the node is being assigned to. In this case, the head is the node itself.

3.1.2 union(A,B)

The merge/union of the sets A and B can be carried out by iterating through the shorter list, updating the stored head value to the head value of the longer list's elements, and linking each of its elements to the longer list. The runtime for this operation would be O(min(n_A, n_B)) where n_A is the number of elements in A and n_B is the number of elements in B. This is due to the fact that we will have to iterate through each of the elements in the smaller of the two lists and link them to the longer of the two lists. I have assumed that updating the head value and link takes a constant runtime O(1), and the iteration through all the elements take linear runtime O(n).

3.1.3 find(i)

The find operation receives the node as an input, which can be used to access the head of this node. This can be completed in constant runtime, O(1). There is no further operations that take place in the find function.

3.2 Part b

3.2.1 make-set

For the make-set function, I intialize a node with its value, as well as information about the root of the tree (which in this case, is the node itself). This process takes a constant, O(1) runtime.

3.2.2 union(A,B)

For the union function, I update the current root of B to be the current root of A. Assuming that the update process takes O(1) constant runtime, union(A,B) should run in constant runtime.

3.2.3 find(i)

In order to return the head element of element i, in a tree structure, we would have to essentially go from subroot to subroot (here, a subroot is a stored root in a node that is not the root of the tree), until we reach the head value, which is the root of the tree. This takes a logarithmic runtime O(log(n)), since we have to go through the height of the tree, the worst case runtime being when we start at a node in the lowest level of the tree.