Student Number: 268461

README FILE FOR PROGRAMMING ASSIGNMENT #2 - TIE-20106 - DATA STRUCTURE AND ALGORITHM.

I. Data structure for saving employee's data.

Every person will be structed by "Person" struct

- The flag "leader" in struct Person is used for determining the ones that does not have the boss but have underlings. If there is only one "leader" in datastructure, he is the CEO (without anyone that does not have boss, and underlings either).

- I chose to save the ID of underlings of any Person in a vector since I need to access it a lot and accessing element is O(1). I also need to make changes to this vector everytime we remove a person. In addition, the return value of the function "underlings" is vector<PersonID> so it is a good idea to just return the exact type and sort it later. (since the number of underlings might not be so much, sorting will not take too much time)
- I chose to save friendship as unordered_map in order to access it quickly.

I used 3 associative data structure: map, multimap and unordered_map in the private interface of the class "Datastructure".

The unordered_map is needed because I need to search for a specific person with the ID very often, while searching in unordered_map is O(1). This is the only use of this datastructure.

The multimap is used just to satisfy the fact that we don't need to sort salary anymore. Since more than one people can get the same salary amount, this is a good datastructure.

The map is used with the value of each pair is a set of PersonID. I chose this because I need to search people based on the name (the "find_name" function). The value is in the state of set since it will automatically sort for me.

II. Asymtotic Performance of every command.

1. Function "add_person"

- To create a Person and assign value, it is O(1).

- To insert that Person with the key of ID to the unordered_map is O(1)

- To find the name of this person in the map name_dict is O(log n)

- If the name is found, insert the ID to the set value is O(log n).

- If the name is not found, insert a new pair of ID and a new set is O(log n)

=> The performance of this function is O(log n). (n is the number of names in datastructure)

2. Function "remove_person"

- Find the bossid of the person: O(1) (Finding the name is O(1) and we can access the Person and the bossid property.

- If the boss is found:

+ Create a reference to that boss's underlings is O(1)

+ Delete the person from the boss's underlings is O(n)

+ Copy the person's underlings to boss's underlings is O(n)

+ Change the boss of each underlings is O(n)

- If the boss is not found:

+ Delete the bossid of the underlings is O(n)

- Remove the person's ID from map name_dict is O(log n)

- Remove the person's ID from multimap salary_dict is O(n) (need to scan all the possible person to check his ID)

- If the current to be deleted is a leader (leader=true), we need to convert all the underlings to be leader: O(n)

- Delete the pair of the deleted person from unordered_map id_dict is O(1).

- Remove all friendship associate with the deleted person is O(n)

=> The performance is O(n)

3. The function get_name, get_title, get_salary is O(1) (we need to search the key ID in unordered_map id_dict and then return the value's corresponding property.

4. Function "find_persons"

- Find the name in the name_dict map: O(log n)

- Convert the corresponding sorted set of ID to a vector: O(n)

=> The performance is O(n)

5. Function "person_with_title"

- Search for all element in id_dict (unordered_map) to check if that person has the desired title O(n)

- Put the output in a vector O(n)

- Sort that vector O(n logn)

=> The performance is O(n log n)

6. Function change_name:

- Get the old name: O(1)

- Find the new_name in the name_dict (map) and insert the ID to that key: O(log n)

- Find the old_name in the name_dict (map) and then find the ID in the corresponding set and remove it: O(log n)

- Assign new name: O(1)

=> The performance is O(log n)

7. Function change_salary:

- Get the old_salary: O(1)

- Find the old_salary in salary_dict (multimap) and scan all the possible person with the same salary to find the correct person: O(n).

- Delete that pair found above: O(log n)

- Insert a new pair of new_salary and the ID to the multimap salary_dict: O(log n)

=> The performance is O(n)

8. Function add_boss:

- Check and reassign the property "leader": O(1).

- Assign new boss: O(1):

- Push the ID the the boss's underlings: O(1).

=> The performance is O(1)

9. The function size: O(1).

10. The function clear:

- Delete the unordered_map's data: O(n)

- Delete the multimap's data: O(n)

- Delete the map's data: O(n)

=> The performance is O(n)

11. The function underlings:

- Access the underlings vector in the ID: O(1).

- Sort that vector: O(n log n)

=> The performance is O(n log n) (n this time is only the number of underlings, in most of the time, it is not large).

12. The function all_underlings.

- The performance is O(n) with n is the number of size of all underlings vector of all ID in the parameter.

13. Function personnel_alphabetically:

- Scan through all key in name_dict (map) and put the set to the vector output: O(n)

14. Function personnel_salary_order:

- Scan through all key in salary_dict (multimap) and put them to the vector output O(n)

15. Function find_bosses:

- This is a recursive function and find all the bosses (direct and undirect) of an ID. Worst case is when the data is just a tree same as a list: O(n)

- The function's return value is deque since I want to keep the order of bosses (I need to push front)

16. Function find_ceo:

- Scan through all key in id_dict (unordered_map) and check if the ID is a leader or not, if yes then put to the vector of leaders: O(n).

- If we found someone who does not have boss and not a leader => We don't have a CEO

=> The performance is O(n)

17. Function nearest_common_boss:

- Use find_bosses on two person : O(n)

- Loop through the bosses from two deque using index: O(n)

=> The performance is O(n)

18. Function higher_lower_rank:

- Run find_ceo O(n).

- Find the current rank of the ID: using find_bosses: O(n).

- Recursively call all_underlings to everyone from the ceo to the rank that is just above the current ID: O(n) (because in the worst case, we are just actually take all the employee, which means the current ID is the lowest rank.

- Find all ID that have the same rank as the current ID: using all_underlings to the rank just above: O(n)

- Subtract the size of higher_rank and equal_rank to get lower_rank: O(1).

=> The performance is O(n)

19. Function min_salary, max_salary:

- Take the first and last iterator and return the ID: O(1)

20. Functions median, first_quartile, third_quartile:

- Increment the first iterator to the desired place and return the ID: O(n)

========== NEW IMPLEMENTATION IN HW3 ==========

21. Function add_friendship:

- Find two ids in id_dict O(n) but it would be Θ(1) on average.
- If all ids are found, insert new friendship to unordered_map O(n) and Θ(1) on average.
- => The performance is Θ(1) on average.

22. Function remove_friendship:

- Find two ids in id_dict O(n) but it would be Θ(1) on average.
- In each id, find other in its own friendship unordered_map: O(n) but it would be Θ(1) on average.
- Remove the ids out of friendship unordered_map: O(n) but it would be Θ(1) on average.
- -> The performance is Θ(1) on average.

23. Function get_friends:

- Find the id in id_dict: O(n) but it would be Θ(1) on average.
- Copy data from that id's friendship unordered_map to a vector is O(n).
- Sort the vector: O($nlog(n)$)
- => The performance is O($nlog(n)$).

24. Function all_friendship:

- Copy all data from id_dict to a temporary unordered_map: O(n)
- Add new friendship to a vector and then delete that friendship to prevent it from being added later. So it is basically add all friendship to a vector. Let's say there are m friendships -> The performance is Θ(m)
- Sort the vector: Θ($mlogm$)
- => The performance is O(n + mlog m)

25. Function shortest_friendpath + print_path:

- This is just an implementation of Bread-first search algorithm so the performance will be O(N+V) with n is number of friends and V is number of friendships.

26. Function check_boss_hierarchy:

- Finding freelancers and leaders: O(n)
- Doing bread-first search to find out if the network is a single hierarchy: O(N+V)
- => The performance is O(N+V)

27. Function cheapest_friend_path + relax_dijkstra:

- This is an implementation of Dijkstra algorithm so the performance is O((N+V)log(V))