Our Data Structure contains:

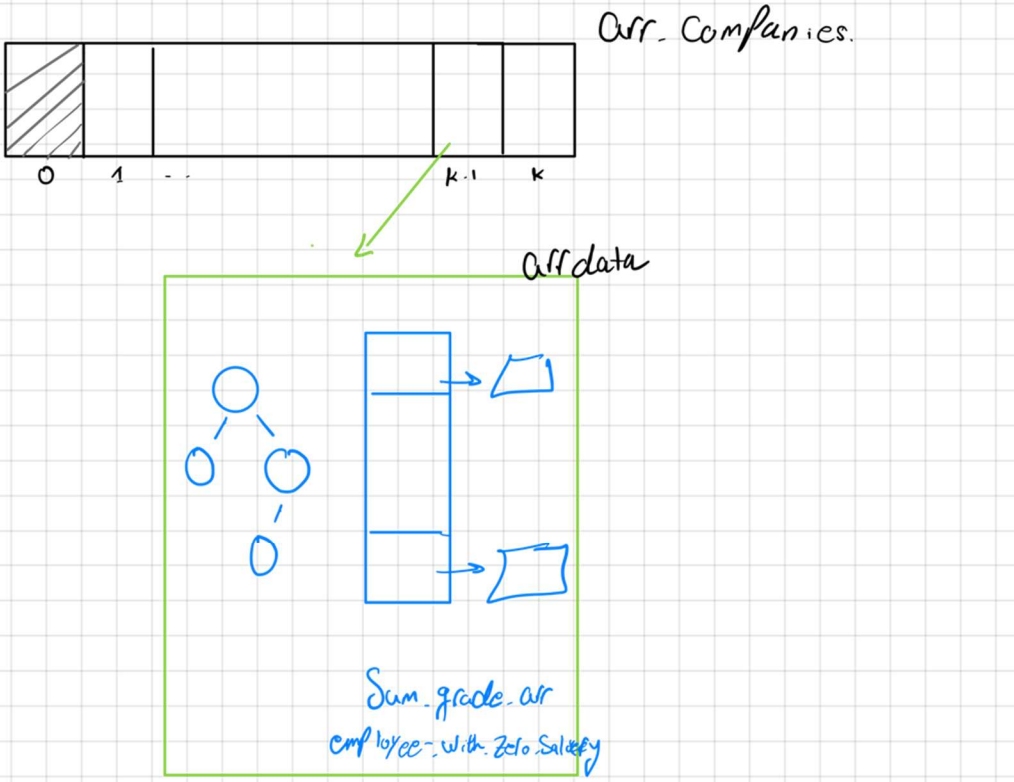
1. Employee_hash: hash table for all employees - EmployeeData, with a regular hash function which maintains a uniform distribution for the employees , the data is EmployeeData.
2. Employees_tree : rank tree , EkeyBySalary (will be explained later), the data is intObject which refer to employee ID , the tree includes only employees which the salary is not zero .
3. Companies : A Union Find Structure , the elements of the UF is companies and the groups are group pf companies , 2 companies are in the same group if one of them by another or both of them bought by the the same group of companies.
4. arr : Array of k companies , arr[i] contains information about the I company . arr[i] points to ArrData class which will be explained shortly .

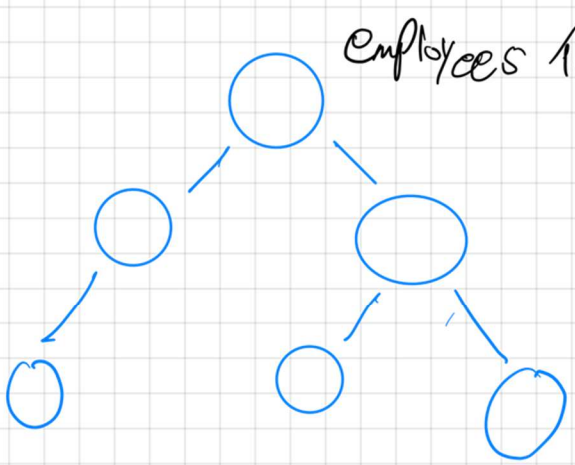
Details for each class:

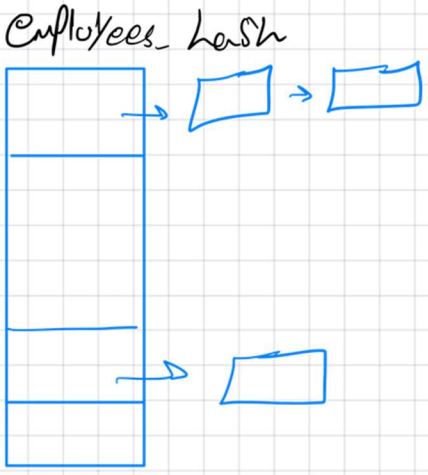
1. ArrData : class which describes the data stored in each index in the array .
   - Each ArrData classcontains :
     - HashTable<EmployeeData>* employees_hash_arr - hash table for the employees in company i.
     - RankTree<EKeyBySalary,intObject>* employees_tree_arr Rank tree which contains the emplyees whom there salary in not zero .
     - int sum_grades_arr
     - int employees_with_zero_Salary_arr

2. EmployeeData : Describes the data of an employee .
   - Each EmployeeData node contains :
     - Employee Id
     - Salary
     - Grade

3. EkeyBySalary : we use this class in order sort the trees sorted by salary as the main classification and id as secondary classification .
   - Each EkeyBySalary contains :
     - Salary
     - Employee Id

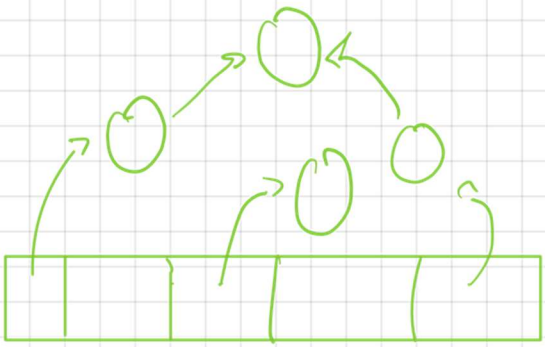






Sum_grade

employes_with_salary_zero



Comparison

Space Complexity: O(n + k)

1. EmployeeHash: contains k nodes , a node for each company .
   due to the fact that we build this AVL such that it contains only companies with at least one employee , in addition to … every employee associated with only one one company , this AVL tree contains n+k nodes .
   every node contains constant number of fields Regardless of n or k .
2. allCompanies : contains k nodes , a node for each company .
   every node contains constant number of fields Regardless of n or k .
3. employeeById : contains n nodes , a node for each employee .
   every node contains constant number of fields Regardless of n or k .
4. employeeBySalary : contains n nodes , a node for each employee .
   every node contains constant number of fields Regardless of n or k .

   space complexity = O(n+k)

functions :

1. void* Init() :initializing empty rank tree , hash table , empty array – size m , empty Union Find .
   ➔ time complexity : O(1) .

2. StatusType AddEmployee : adding a new employee to a given company , salary 0 and grade zero .
   First of all we check that there are no suck employee before adding , else we return failure .
   We search if the company exists in the Union Find structure which we named Companies previously
   2.1: insert the employee in the global hash table of employees .
   2.2: we increase the counter of employees with zero salary .
   2.3:insert the employee to the given company hash table .
   2.4: increase the counter of employees with zero salary in the given company .
   2.5:we update the grade in the company sum of grades and the global sum of grades variables .

   ➔ Time complexity : O(log*k) . – find(companyid) costs O(log*k).

3. StatusType RemoveEmployee : looking for the employee in the global hash table of employees .
   From the data in the hash table we can extract the salary , grade and the company id which the employee works for .
   If salary equals zero :
   3.1.1: from the total sumGrade we decrease the grade of the employee (in order to support function averageBumpGradeBetweenSalaryByGroup .
   3.1.2 : from the total sumGrade of the given company we decrease the grade of the employee (in order to support function averageBumpGradeBetweenSalaryByGroup .
   3.1.3:remove the employee from the global hash table and the employee company hash table.
   3.1.4: decrease the counter of employees with salary equal to zero from the global and the given company counter of employees with zero salary.
   Otherwise :
   3.2.1 : remove the employee from the global employees tree
   3.2.2 Remove the employee from the employees tree of the company which the employee works for.
   3.2.3 : remove the employee from the global hash table and the employee company hash table.

   ➔ Time Complexity : O(log(n) במוצע על הקלט משוערך
   n is the number of employees in the system .
   O(1):
   - בממוצע על הקלט משוערך משום שאנחנו מסירים מ טבלת ערבול
   O(log(n): remove from rank tree which includes n employees .

4. StatusType aquireCompany: first of all we check that all the parameters valid , otherwise we return INVALIDINPUT .
   In this function we unite the groups of the given companies by size ,which means we add the smaller group of companies to the bigger one and we do contraction of the routes (כיווץ מסלולים) whish allow us to maintain the time complexity of O(log*k).
   4.1:finding the groups of the given companies and extracting there sizes from the array of k companies which we showed earlier.
   4.2:after checking all the cases (size) we extract the employees from the tree of employees for each company using inorder over each tree .
   4.3:merge the arrays of employees .
   4.4: from 4.3 we receive sorted array of employees , we can build a new rank tree from the new array in o(nA+nB) time complexity as we saw in the lecutres/tutorials.
   4.5:assign the new rank tree to the bigger size tree ArrData which we point to from the array of K companies .
   4.6:merge the two hash tables and updates the relecant data in each company ArrData
   - ➔ Total time complexity : O(log*k +nA + nB)
     nA – number of employees in group A
     nB - number of employees in group A

5. StatusType employeeSalaryIncrease : first of all we check that we received valid input ,otherwise we return INVALIDINPUT .
   5.1: we extract the grade and the salary and the company id which the employee works for , from the global employee hash table .
   5.2: update the salary and the of the given employee in both the global hash table and the hash table of the company which the employee works for .
   5.3:if the salary before increasing is 0 we decrease the grade from the total sum of grades of employees which there salary is 0 .

Also, we decrease the grade from the company sum of grades of employees which there salary is 0.
We update the global counter and the companies counter of employees with salary 0.
5.4:update the employee key in the global employees tree and the company which the employee works for tree .

6. StatusType promoteEmployee: first of all, we check that all the parameters valid , otherwise we return INVALIDINPUT .
   6.1: looking for the employee in employee_hash hash table , in case we didn't find the employee we return FAILURE.
   6.2: if the bumpGrade parameter is non-positive we return success.
   6.3: we find the grade, salary and company id of this employee and we add the bumpGrade to his grade.
   6.4: if his salary is zero, we add the bumpGrade to the sum_grades_arr in the for his company.
   6.5: else we create a new node with the updated grade, and we insert it in the employee tree for his company, also in the general employee tree, and lastly, we remove the old node.
   Time Complexity : O(log(n)) in average
   O(log(n)): inserting and deleting from a rank tree.
   O(1) in average: searching in the hash table.

7. StatusType sumOfBumpGradeBetweenTopWorkersByGroup
   check if we received valid input , otherwise we return INVALIDINPUT .
   7.1: if the company Id 0 we work with the global employees tree , otherwise we work with the tree of the given company id .
   7.2: if the size of the tree les than m we return Failure .
   7.3: we search of the employee which rank is size-m , find the common father of this employee and the maximum employee .
   We extract the sumGrade (additional data which stores the sum of grades in the tree which the employee is the root) . find the employee which ranks size-m (start from the common father) every time we take right in the path

we decrease the current employee sum grade from the total sum of grades .

7.4: return sum .

➔ Time complexity : O(log*k + logn).
Select and common father -O(logn)
Searching for the employee - O(logn)

8. StatusType averageBumpGradeBetweenSalaryByGroup : first of all, we check that all the parameters valid , otherwise we return INVALIDINPUT.
   8.1: we check if the company id is zero then we need to go through the employee tree for all the employees in our data structure, but also we check if the salary was zero then we need to include the grades of the employees with the salary zero.
   8.2: else we need to go through the union find to find the right company with company id.
   8.3: we insert two nodes, one with a minimal key and lower salary, and the other with maximal key with higher salary, and we insert them into the right employee tree that has been found earlier, in order to mark the nodes, we need to go through.
   8.4: we check the number of nodes we have to go through and save it in a variable named range, then we find the father of the two nodes we added, and add the total sum of grades to a variable named num.
   8.5: from here we search the lower salary node again and every time we go to the right we decrease the grades of the left sub-tree, then we go and search for the higher salary node and every time we go to the left, we decrease the grades of the right sub-tree.
   8.6: lastly, we return num/range.
   Time Complexity : $O(\log*(k)+\log(n))$ משוערך.
   Log*(k): due to going through the UF.
   Log(b): due to the search in the rank tree.

9. StatusType companyValue: check if we received valid input , otherwise we return INVALIDINPUT , looking for the company using find function (UF) ,return the value of the company .

➔ Time complexity : O(log*k)
Find function over UF data structure which contains K companies at most .

10. void Quit: we delete the UF, the employee main tree, and for each company in the companies array we delete the employee hash table and the employee tree.
Time Complexity: $O(n+k)$.
n – the total number of the employees in all of the data structure.
K – the total number of the companies in all of the data structure.

11. StatusType bumpGradeToEmployees: check if we received valid input , otherwise we return INVALIDINPUT.
11.1: first of all, we added to the employee tree nodes a new field called extra, which is meant to contain the addition we need to make to the grade of an employee when searching for him, in order to get the updated grade
11.2: we go through the companies' array, and for each employee tree in the companies we add a node with a minimal id and lower salary, and a node with maximal id with higher salary, then we search for the father of these two nodes(אב קדמון).
11.3: for the fathers' extra field we add grade.
11.4: then we search for the new node we added with the lower salary and every time we go to the right, we updated the left sub-tree extra field to (-grade), and we search for the higher salary new node, and every time we go to the left, we update for the right sub-tree.
11.5: since we know the extra field for the first father of every two nodes, we can get the updated sum_grade which helps us in the function sumOfBumpGradeBetweenTopWorkersByGroup, and this in it's turn will help us with the averageBumpGradeBetweenSalaryByGroup function because we always have the updated sum_grade.
Time Complexity: $O(k \cdot \log(n))$
Log(n): the search for two nodes in the rank tree.
K: we do this process for the k companies.

Space complicity :

companies_arr : an arraywith size k+1, each company contain its employees so in total we hae O(k+n) space complicity wich k represents the number of companies and n represents all the employees

additional tree and hash for all the employee

so we have : $k + 2n + 1 = O(k+n)$

ahmad khader 209113018

mohammed bisher 207723040