תרגיל בית רטוב 2

אביב 2023

ת"ז סטודנט 1 : 212931315

ת"ז סטודנט 2 : 212605356

Structures:

Hash table - clients	Hash table that uses AVL trees instead of linked lists. It stores all the clients. (Excellent distribution 🙂)
Union find - stock	Each group (flipped graph) is a column. Each node in a group is a record that is placed in that column. It has extra value - used to calculate the height of a specific record, when maintaining the regular union-find time complexity -> O(log*(n))
Array of class record - records	Each cell saves info (e.g. purchases count) about the record associated to the cell index (its id)
AVL tree - members	AVL tree that stores only clients that are members. It has an extra value which is used to calculate the sum of all prizes that a member has, using just O(log(n)) time complexity.

Non-trivial functions in structures:

structure	function	description	How it works	Time complexity
AVL tree	updateRange(key1, val):	Adds val to each key that is less or equal to key1.	If key found: In the path from root to node: In the first node of a right-series of steps, we add val to its extra. When turning left, after a right-series, we sub val from its extra. (only then). Else: Do the same steps in the path to the key1 predecessor. Finally: When arriving to the target node: If the last step was left then add val to the node's extra. Sub val from the node's	O(log(n)) Since search is done in O(log(n))

			right son (if exists)	
Union -Find	getValidExtra(key)	In our case: Returns the height of recorde <key> in its column.	Simply calculates the sum of extras (that the UF maintained during its lifetime*) from the node holding the key, to the group root. *maintained in: Union function: Updates (in O(1)) the extras of O(1) amount of nodes so that the "getValidExtra" would return the correct values regarding the union (put column on another, in our case) Find function: When **path compression** is done, it updates extras of the related path (in o(1) for each node) so that "getValidExtra" would return the same values as before the **path compression.**	O(log*(m)) amortized with Find and Union (union used in putOnTop function). Since Its complexity is less than Find and it uses Find in it, we can count each call of this function as a call of Find function, which ends up as before. Maintenance will not affect amortized complexity, since each update is in O(1), and each action (in Find or Union) is done at most with a single update.

m- the number of records.
n- the number of all clients.

Functions:

RecordsCompany():
We initialised recordsInfo to point to nullptr, and recordsNum to zero, because there is no clients yet, and creating an empty avl ranked tree and hashtable by calling the default constructor for members and clients.
Time complexity => o(1)

~RecordsCompany();

- We call the destructor for every structure, for members and stock, and records info.
- We pass on the array of clients, and get the length of it from the its field by o(1), and calling the destructor for every node, we couldn't call the destructor for clients because clients is an array for pointers, therefore we have to pass on the pointers and destroy them.
- Time complexity=> o(n+k)

StatusType newMonth(int *records_stocks, int number_of_records);

- We delete the current stock(union find), and initialises a new one => o(m)
- We delete the current records array and initialises a new one with a new length=>o(m)
- We pass on the new array of records and updating every record in the array according to records_stocks array =>o(m)= number_of_records.
- We call the function resetPrizes thats set the extra fields in the avl node to zero, we pass on all the nodes by inorder traversal =>o(n)
- Time complexity =>o(n+m)

StatusType addCostumer(int c_id, int phone);

- We check if the clients is exist in the hashtable by the function contains, its check if the c_id is exist => o(1) on average
- We initialise a new client with c_id=> o(1).
- We insert the new client to clients(the hash table) =>o(1) on average.
- Time complexity => o(1) on average.

Output_t<int> getPhone(int c_id);

- We search about c_id in clients => o(1) on average
- We return the value of getphone function for the wanted client=> o(1)
- Time complexity => o(1) on average

StatusType makeMember(int c_id);

- We check if the c_id is exist in the clients => o(log n) in the worst case: all the clients in the same element in the array
- We check if the c_id is exist already in members => o(log n)
- We define a new pointer of client points to the client of the c_id => o(1)
- We update the ismembers field in the client => o(1)
- We insert the new pointers to members, insert to avl ranked tree => o(log n)
- Time complexity=> o(log n)

Output_t<bool> isMember(int c_id);

- We check if the c_id is exist in clients => o(1) on average.
- We return the value of the field of ismember => o(1).
- Time complexity => o(1) on average.

StatusType buyRecord(int c_id, int r_id);

- We check if the c_id is exist in the clients => o(log n) in the worst case: all the clients in the same element in the array
- We update the number of purchase for the record, we added 1 to it => o(1)
- We check if the client is member => o(1), after getting the pointer of the client
- We use the function set expense if the client is member => o(1)
- Time complexity => o(log n)

StatusType addPrize(int c_id1, int c_id2, double amount);

- We use updateRange function twice, the first time to add amount for the every c_id<c_id2, the second time we add the value (-amount), for the c_id<c_id1 => o(log n)
  We saw this trick in ido tutorial.
- Time complexity => log(n)

Output_t<double> getExpenses(int c_id);

- We check if the c_id is exist in the clients => o(log n) in the worst case: all the clients in the same element in the array
- We calculate the expense by the following steps:
  1- we get the filed of expense of the client => o(1)
  2- we use getValidExtra to calculate the sum of the prizes for the client=>o(log n)
  3- we sub 2 from 1
  4- return the value after sub.
- Time complexity => o(log n)

StatusType putOnTop(int r_id1, int r_id2):

- Calls union(r_id1, r_id2) for the “stock” structure, union itself chooses root based on which tree has more nodes, and updates the extras for a constant (O(1)) amount of nodes, so that the sum from root to a specific node will return the height of that node (represents a record). Updates also the group info as needed (column height, column index). As described before.
- Ends up with O(log*(m)) amortized with getPlace.

StatusType getPlace(int r_id, int *column, int *hight):

- Calls getValidExtra function which as described before, runs in O(log*(m)) amortized time complexity with putOnTop.
