Mivne Wet 2

234218

Gabriel Uram - 941200156
Ori Razon - 204903397

Our data structure is comprised of the following private members:

1. A hash table for all our customers which is an array of pointers to AVL trees as opposed to the typical array of linked lists. The hash function is (customer_id % table_size).
   Space complexity: O(n)
2. An array of all the records in that month
   Space complexity: O(m)
3. A member_tree which inherits from the typical AVL tree and overloads rotation operations in order to support member-fee calculations. This, in the worst case, holds all the customers.
   Space complexity: O(n)
4. An int which specifies the number of records that month.
   Space complexity: O(1)
5. An instance of our unionFind class, which holds the "stacks" of our records - all private members of unionFind are proportional to the number of records in that month.
   Space complexity: O(m)

Total Space Complexity: O(n+m)

RecordsCompany():

This initializes an empty hash table of shared_ptrs to customers (O(1)), assigns a nullptr as our records array (O(1)), initializes an empty tree (O(1)), initializes an integer as 0 (O(1)), and calls the constructor of our UnionFind class with a size of 1, hence the constructor will do constant time operations proportional to a size of 1: O(1). Overall:

O(1)

~RecordsCompany():

The destructor of our hash table iterates through a table_size proportional to the number of customers, executing constant time operations with each iteration: O(n). The number of nodes throughout all trees in our database is also proportional to the number of customers, and thus deleting these nodes takes O(n). All the shared pointers to records are deleted automatically O(m), and deleting record_copies itself is O(1). The destructor of Union FInd executes a constant number of delete operations: O(1). All in all:

O(n+m)

NewMonth():
First, the function assigns the debt and discounts of every customer to zero: this is a constant time operation which is executed on every customer: O(n). From there we delete our array of records: O(1). We then dynamically allocate an array of size m to our array of records and for each index, store the necessary information. This is a constant time operation which is executed m times: O(m). The UnionFind constructor also receives a size of m, and thus executes constant time operations m times: O(m). In total:
O(n+m)

AddCustomer():
As we have seen in the lectures, the insert operation for a typical hash table which is composed of linked lists is O(1) amortized, even with resizing operations. In this case, we are maintaining the same complexity for our resize operation whilst reducing the complexity of inserting (inserting into a linked list is O(n) whilst inserting into a tree is O(logn)). Thus, it is trivial that inserting a customer to our linked list is **O(1) amortized.**
getPhone():
Finding a node in a hash table is of the same time complexity as inserting a node, and thus, finding a node, or in this case a customer, is O(1) amortized based on the same justification provided in addCustomer(). From here, getting the phone number of the customer is a constant time operation. Overall:
O(1) amortized

makeMember():
Finding the relevant customer tree is a constant time operation as it is simply applying the hash function to the customer's ID. From there, we have to iterate through a tree which, in the worst case, holds all customers in the data structure: n nodes. As we have seen in the lectures, the complexity of reaching any node in an AVL tree with n nodes is O(logn). Once we reach the appropriate customer, making him a member is a constant time operation. Therefore:
O(logn)

isMember():
The justification for the complexity of this function is identical to getPhone(). Just instead of getting their phone number, which is a constant time operation, we are getting their member status, which is also a constant time operation. Overall:
O(1) amortized

buyRecord():

Based on the same justification as makeMember(), finding the relevant customer is O(logn). From there, adding a payment to the customer's monthly debt, if they are a member, is a constant time operation. Accessing the r_id index of our record's array is a constant time operation, and adding a purchase to the record is an incrementation of a private data member which is also a constant time operation. All in all:
O(logn)

addPrize():
There are two calls to addPrize_aux, a function which performs a find operation which iterates through the member tree and finds the relevant node (O(logn)), and at each node, under certain conditions, performs an arithmetic constant time operation (O(logn)). Overall:
O(logn)

getExpenses():
Based on the same justification as makeMember(), finding the relevant customer is O(logn). From there finding their expenses is simply a constant time summation operation. Overall:
O(logn)

putOnTop():
In this function we are executing a "union" operation, similar to the typical union operation which implements both path-compression and rank-optimisation algorithms. As we saw in the lectures, this is of O(log*m) time complexity, where m is the size within our UnionFind class. In addition, we added a constant number of constant time operations throughout the union operation. As such, the total time complexity of putOnTop is unaffected by our additions:
O(log*m)

getPlace():
In this function we are executing a "find" operation, which is identical to the find algorithms we have seen in lectures. As we know, this operation is of O(log*m) time complexity:
O(log*m)