תרגיל רטוב 2 – חלק יבש

שרה גריפית – 341312304
שלמה אבנון – 322677063

Description of the Data Structure:

For the record company's system, we will build a data structure made up of three main structures:

- *m_members* – an AVL tree that holds the record company customers who are club members
- *m_customersHashTable* – a hash table of pointers to AVL trees of all the company's customers
- m*_records* – an array of pointers to the records the record company owns

The tree *m_members* is an AVL rank tree, with its extra data being the "prize" each customer was given.

The hash table *m_customers* contains all of the company's customers in the system, where AVL trees are used for conflict resolution. This is instead of using a structure such as a linked list for cases in which two customers are given the same cell in the hash table.

Additionally, the records in the system, saved in the *m_records* array, are organized into up-trees. As shown in class, this means that each tree contains a root, where every other node in the tree points to that root. Each tree represents a stack of records, and internal fields of height and column save each individual record's location within the stack. Actions done on the stacks are implemented through the up-trees using Union-Find.

The system also contains three integer values:

- m_numRecords – the total number of records in the system
- m_currentHashSize – the current size of the hash table
- m_numCustomers - the total number of customers in the system

To implement the structure in this way, three classes were added: Customer, Record, and Stack, which contain the details of the customers and records in the system.

- The **Customer** class contains the basic information of each customer in the system: the customer's ID, phone number, whether the customer is a member of the company's club, and the customer's debt.
- The **Record** class contains the basic details of each record in the system: the record's ID, the number of copies existing of that record, the number of times the record was bought, the record's location (column and height in the stacks). Additionally, the class contains a pointer to each record's parent in the up-trees for the Union-Find structure. Finally, each record has a pointer to its stack.
- The **Stack** class is used to contain general information about each up-tree of records in the system. Each stack contains the stack's total height, column, and the total number of records in that stack. This information is useful during functions such as Union, where the total size of the stack determines the implementation. Additionally, each record's column is determined by its stack. This class does not represent the data structure "stack," rather a "stack of records."

Let us note that there is only one copy of each object:

- The tree *m_members* is an AVL tree of pointers to members.

- The hash table *m_customersHashTable* contains AVL trees that point to members.
- The array *m_records* contains pointers to the records in the system.

Each of the above structures point to the original copy of each object. We will also note that at most, each record is in its own stack, and therefore there are at most m stacks in the system at any given time, making the space complexity used by the stacks O(m).

Space Complexity: In each function, there is no additional space complexity used, beyond what is required by the assignment. As such, the space complexity remains O(n+m), where n is the number of customers and m is the number of records in the system.

The functions required for the system and an explanation of their implementations and complexities:

RecordsCompany() –

The number of records and customers in the system are each initialized to zero. The AVL tree *m_members* is initialized as empty through a call to their constructors. The array of records, *m_records*, is initialized as *nullptr*. The initial size of the hash table is initialized as seven. Thereafter, an empty AVL tree is initialized in each of the hash table's cells. This is done through a call to the AVL tree's constructor.

Time Complexity:

- An empty tree contains a single, empty node, and the time it takes to create it is O(1). Similarly, it takes O(1) to implement each int value and set the array of records equal to *nullptr*.
- Initializing the hash table of length seven takes O(7) = O(1), as the initialization of each cell takes O(1).

In total, the time complexity is **O(1).**

~RecordsCompany() –

Firstly, the data of all the customers is destroyed. This is done by going over the hash table containing the system's customers in a loop, deleting the contents of each cell individually. For each cell, a function is called that frees the memory held in the data held by that AVL tree. Afterwards, the memory held by the hash table itself is freed. Once the customers have been deleted, the records are deleted. This is done in a loop that goes over each record in the records array. For each cell, the records' stack is deleted as well. At the end of the loop, the array of records itself is deleted.

Time Complexity:

To destroy the customers' data, each customer is gone over individually in the hash table. This requires time O(n). Similarly, destroying the records requires going over each record individually. Throughout, each record's stack is destroyed. Each record can only belong to one stack, and so at most, there are m stacks. Therefore, the time it takes to destroy the records is O(m). In total, the time complexity is **O(m+n).**

newMonth –

In this function, the records in the system are reset, along with each member customer's debt. First, the validity of the input is checked. If the number of records given is less than zero, INVALID_INPUT is returned. After, the previous records in the system are deleted. This is done as in the destructor: the array containing the records is gone over, and each record and its stack is deleted, one by one. Afterwards, a new array of records is created, according to the given size. The information about each record given is used to fill the array with new objects of the type Record. Additionally, each record is initialized in its own stack (as in the function MakeSet() shown in class). Thereafter, the number of records in the system is set equal to the given number of records. Finally, each member customer's debt and prize fields are set equal to zero. This is done in an inorder walk on the m_members tree. At any point, if there is an allocation error, ALLOCATION_ERROR is returned. Otherwise, SUCCESS is returned.

<u>Time Complexity:</u>

- The time it takes to delete the previous records - O(m) as explained in the destructor
- The time it takes to create and fill a new array according to the given – Each stack and Record is initialized in time O(1) through a call to their respective constructors. Therefore, this takes a total time of O(m).
- The time it takes to set each member customer's debt and prize fields equal to zero – This requires going over each member customer, where resetting each field takes O(1) each. Therefore, this takes a time of O(n) in the worst case, where all customers are members.

In total, the time complexity is **O(n+m)**.

addCostumer –

In this function, a new customer is added to the system. First, the validity of the input is checked. If the given ID or phone number are less than zero, INVALID_INPUT. Thereafter, memory is allocated for the new customer. The customer is initialized with the given ID and phone numbers. If there is an error during the memory allocation, ALLOCATION_ERROR is returned. Because each customer is initialized as not a member, there is no need to add a new customer to the member tree. From here, the hash table is checked. If the hash table's size is equal to the number of customers in the system (including the new one), the hash table must be rehashed. In this case, the table is enlarged, and all the data in the old table is copied into the new table using the new hash function that is calculated according to the table's new size. If there is an error during memory allocation in this stage, ALLOCATION_ERROR is returned. Thereafter, the new customer's index in the array is calculated according to the hash function. The customer is inserted into the AVL tree at that index. If there is already a customer in the hash table with the given ID number, the insertion will fail and ALREADY_EXISTS will be returned. If an allocation error occurs during the insertion, ALLOCATION_ERROR will be returned. Otherwise, the number of customers in the system is increased by one, and SUCCESS is returned.

<u>Time Complexity:</u>

Creating a new customer that is initialized with the given ID and phone number takes a time of O(1).

If needed, the hash table's size will be increased. The hash table's new size will be 2n, where n is the number of customers in the system. New trees will be created in each cell of the hash table through the AVL tree constructor. This takes a time of O(2n) = O(n). From here, a temporary array will be created, whose size is equal to the number of customers in the system. Each customer from the old hash table will be inserted into the temporary array. In total, this requires going over the n cells and AVL trees of the previous hash table, along with the n cells of the temporary array. This will therefore take a time of O(2n) = O(n). After inserting each customer into the temporary array, each customer's location in the new hash table is calculated using the new hash function, and each customer is then inserted into the new hash table. Finally, the temporary array and previous hash table are destroyed in time O(2n) = O(n). According to the uniform distribution assumption, there is an average of one customer in each cell of the hash table (i.e., the average size of each AVL tree is 1). Therefore, the amortized insertion and deletion time is O(1) on average on the input. In total, the time it takes to increase the hash table's size is O(n). Increasing the hash table's size only happens when the load factor is equal to 1. This happens once in every n insertion actions. As such, the amortized time, on average on the input, for increasing the hash table's size is O(1) as seen in class.

The new customer is then inserted into the hash table. As explained above, this takes an amortized time of O(1).

Therefore, the total amortized time complexity is **O(1)** on average on the input.

getPhone –

In this function, a customer's phone number is returned. First, the validity of the input is checked. If the given ID number is less than zero, INVALID_INPUT is returned. If the input is valid, the customer's index in the hash table is calculated according to the hash function and the given ID number. The customer is then searched for in the tree contained in the hash table at the calculated index. If there is no customer with the given ID, DOESN'T_EXISTS is returned. Otherwise, the customer's phone number, which is saved as an internal field in the Customer class, is returned.

<u>Time Complexity:</u>

Calculation of the customer's index in the hash table requires a simple mathematical evaluation, and therefore takes time O(1). According to the uniform distribution assumption, there is an average of one customer in each cell of the hash table. This means that the average size of each AVL tree is one. Therefore, the time it takes to search each cell on average is O(1) on average on the input (because searching an AVL tree that contains only a root takes time O(1)). Once found, accessing the customer's phone number is done with a simple getter function, taking time O(1).

In total, the time complexity is **O(1)** on average on the input.

makeMember –

In this function, a customer becomes a club member. First, the validity of the input is checked. If the given customer ID is less than zero, INVALID_INPUT is returned. Otherwise, the customer's array index in the hash table is calculated using the hash function. The customer is then searched for in the AVL tree held at that index in the hash table. If no such customer is found, DOESN'T_EXISTS is returned. Otherwise, the customer is made a member through a call to a helper function implemented in the Customer class. If the customer is already a member, ALREADY_EXISTS is returned. Otherwise, the customer is inserted into the members tree. During the insertion, the customer's initial prize must remain zero. To do this, before rotations, the customer's current prize, the sum of the prizes of its parents, is calculated. This is done by going up the search path from the new node. The customer is then given an initial prize equal to the negative value of the calculated prize. By doing this before rotations, the system can ensure that the prizes of all rotating nodes remain the same. This is done through two simple mathematical calculations in each rotation. If an allocation error occurs during insertion, ALLOCATION_ERROR is returned. Otherwise, SUCCESS is returned.

Time Complexity:

- The time it takes to calculate the customer's index in the hash table - O(1) as explained before
- The time it takes to search for the customer in that cell of the hash table - O(logn), because in the worst case, all the customers are in the same cell of the hash table (i.e., in the same AVL tree)
- The time it takes to insert a new customer into the members tree – Because calculating the new node's initial prize is done along the search path, this takes time O(logn). Additionally, keeping the prizes of rotating nodes requires two mathematical calculations in each rotation. In the worst case, this is a time of O(4) = O(1). Due to this, the time it takes to insert a customer into the members tree remains unchanged at O(logn).

In total, the time complexity is **O(logn)** in the worst case.

isMember –

In this function, the system checks whether the given customer is a club member. First, the validity of the input is checked. If the given ID number is less than zero, INVALID_INPUT is returned. Otherwise, the customer's array index in the hash table is calculated using the hash function. The customer is then searched for in the AVL tree held at that index. If the customer does not exist, DOESNT_EXISTS is returned. Otherwise, whether the customer is a club member is returned. This is done by returning the customer's internal field that represents this.

Time Complexity:

Calculation of the customer's index in the hash table requires a simple mathematical evaluation, and therefore takes time O(1). According to the uniform distribution assumption, there is an average of one customer in each cell of the hash table. This means that the average size of each AVL tree is one. Therefore, the time it takes to search each cell on average is O(1) on average on the input (because searching an AVL tree that contains only a root takes time O(1)). Once found, accessing whether the customer is a member is done with a simple getter function, taking time O(1).

In total, the time complexity is **O(1)** on average on the input.

buyRecord –

In this function, a customer buys a record. First, the validity of the input is checked. If the given customer ID or the record ID is less than zero, INVALID_INPUT is returned. If the record ID is greater than or equal to the total number of records in the system, DOESNT_EXISTS is returned. Otherwise, the customer's array index in the hash table is calculated using the hash function. The customer is then searched for in the AVL tree held at that index. If the customer does not exist, DOESNT_EXISTS is returned. Otherwise, if the customer is a club member, the cost of its purchase is added to its monthly debt. This is done by adding the cost of the record (100 + the number of times the record was bought until now) to the customer's debt field in the Customer class. If the customer is not a club member, there is no need to update any of its internal fields. Afterward, the number of times that record was bought is increased by one. This is done by accessing the record directly according to the given ID number in the records array.

Time Complexity:

- The time it takes to calculate the customer's array index in the hash table – O(1) as explained before
- The time it takes to search for the customer in that cell of the hash table - O(logn), because in the worst case, all the customers are in the same cell of the hash table (i.e., in the same AVL tree)
- The time it takes to add the cost of the record to the customer's debt – O(1)
- The time it takes to add one to the number of time the record was bought – O(1)

In total, the time complexity is **O(logn)** in the worst case.

addPrize –

In this function, a prize is given to the club members whose ID numbers are within the given range. This prize is later deducted from their expenses. First, the validity of the input is checked. If the given ID numbers are less than zero, the amount of the prize is less than or equal to zero, or the upper bound of the ID range is less than the lower bound, INVALID_INPUT is returned. Otherwise, a helper function is called. The helper function adds a given prize to all members whose ID numbers are less than or equal to a given minimum. The helper function is called twice, once with the value -amount and on the lower ID bound – 1, and once with the value amount and on the upper ID bound – 1 (because the upper bound of the range is not included in the prize). The helper function is implemented such that the node that is closest to but smaller than the given minimum is searched for in the members tree. This is done similarly to the way a search happens during insertion. However, in this case, there is a possibility that the node chosen will still be larger than the desired. If so, the node's parent will be chosen, until the node chosen is no longer greater than the minimum given. If there is not a node that satisfies this condition, the function ends without making any changes. Otherwise, if such a node is found, the prizes of the nodes along the search path to the minimum node are updated:

- On the first of a series of right turns, add the given amount to the parent turning from.
- On a left turn that follows at least one right turn, add -amount to the parent turning from.

- On a left turn with no right turn directly before it, do nothing.
- When reaching the given node, if it is reached without a series of right turns, add the given amount to the given node.
- If the given node has a right child, add -amount to that child.

Once the helper function is called twice as described above, SUCCESS is returned.

Time Complexity:

Each call of the helper function requires searching along the tree at most twice, as described above. This takes time O(logn). Changing the prize of each child happens in O(1) per change, and takes place throughout the second search in the tree. Therefore, the total time of each call of the helper function is O(logn). The helper function is called twice, and therefore the total time complexity is O(2logn) = **O(logn)**.

getExpenses –

In this function, a member customer's expenses are calculated. First, the validity of the input is checked. If the given customer ID is less than zero, INVALID_INPUT is returned. Otherwise, the customer's array index in the hash table is calculated using the hash function. The customer is then searched for in the AVL tree held at that index. If the customer does not exist, DOESNT_EXISTS is returned. Otherwise, the customer's debt is accessed using a getter from the Customer class. The customer's total prize is then calculated. This is done by finding the customer's node in the member tree, and then going up along the search path, adding the prizes of each of its parents to its own prize. Finally, the value debt – prize is returned.

Time Complexity:

- The time it takes to calculate the customer's array index in the hash table – O(1) as explained before
- The time it takes to search for the customer in that cell of the hash table - O(logn), because in the worst case, all the customers are in the same cell of the hash table (i.e., in the same AVL tree)
- The time it takes to access the customer's debt – O(1)
- The time it takes to find the member's node in the members tree – O(logn) as shown in class (the time it takes to search in an AVL tree)
- The time it takes to calculate the member's prize – O(logn) because the calculation happens along the search path of the AVL tree

In total, the time complexity is **O(logn)** in the worst case.

putOnTop –

In this function, a stack of records is put on top of another. First, the validity of the input is checked. If either of the given record ID numbers is less than zero, INVALID_INPUT is returned. Similarly, if either of the given record ID number is greater than or equal to the total number of records in the system, DOESNT_EXISTS is returned. Otherwise, the root of each of the two given records is searched for. This is done by going up each record's up-tree until reaching the root. On the way, the path between the record and the root is shortened, and the record is connected directly to the root. In other words, this is an implementation of the find() function shown in Union-Find in class. However, when the path between the record and the root is shortened, the record's height must stay the same. A record's height is normally calculated by summing its parents' heights. To ensure that the record's height does not change, the sum of its parents' heights (except for the root itself) are added to its height as the path is shortened. This sum happens in O(1) for each parent, and therefore does not change the overall complexity of the find() function. Once both roots are found, they are compared. If both records have the same root, they are already in the same stack, and FAILURE is returned. Otherwise, the function union() is called. As shown in class, union() takes place according to the size of each stack. The stack's sizes are accessed from the Stack class using a getter function. From there, the smaller of the two stacks is united with the larger. This is done by having the smaller stack's root point to the larger stack's root. The height of each of the roots is updated according to the direction of the union (for example, whether the smaller stack is the stack that is on top). When the stack that is on top is united into, its column must be updated to be equal to the column of the stack that is physically on bottom. Finally, the number of records in the united stack is updated. The previous stack is deleted.

Time Complexity:

The time it takes to find each of the roots is O(log*m) as shown in class. Changing the stack's relevant pointers requires time O(1), as does updating the height of each root and the column of the stack if necessary. Updating the number of records in the united stack also requires time O(1). The class "Stack" contains a finite number of fields, and therefore deleting it takes time O(1). Therefore, the time of the function union() in our case takes time O(1) (because, unlike in the union() shown in class, we receive each root as a parameter).

In total, the time complexity is **O(log*m)**.

getPlace –

In this function, the column and height of a record is returned. First, the validity of the input is checked. If the given record ID is less than zero, or the column or height given are *nullptr*, INVALID_INPUT is returned. If the record ID is greater than or equal to the total number of records in the system, DOESNT_EXISTS is returned. Otherwise, the record's height and column are calculated. The record's height is calculated by summing its parents' heights. The record's column is calculated by calling the find() function, and then accessing the stack's column (the stack's column is the same as each record's column in the stack). Finally, SUCCESS is returned.

Time Complexity:

As explained in the previous function, calculating a record's height is done by summing its parents' heights, similar to the find() function. As explained above, find() takes a time that is unchanged relative to what was seen in class. The find() function returns the stack's root, which contains a pointer to the stack itself. As such, accessing the column after find() takes time O(1). Therefore, in total, the time complexity is **O(log*m)**.