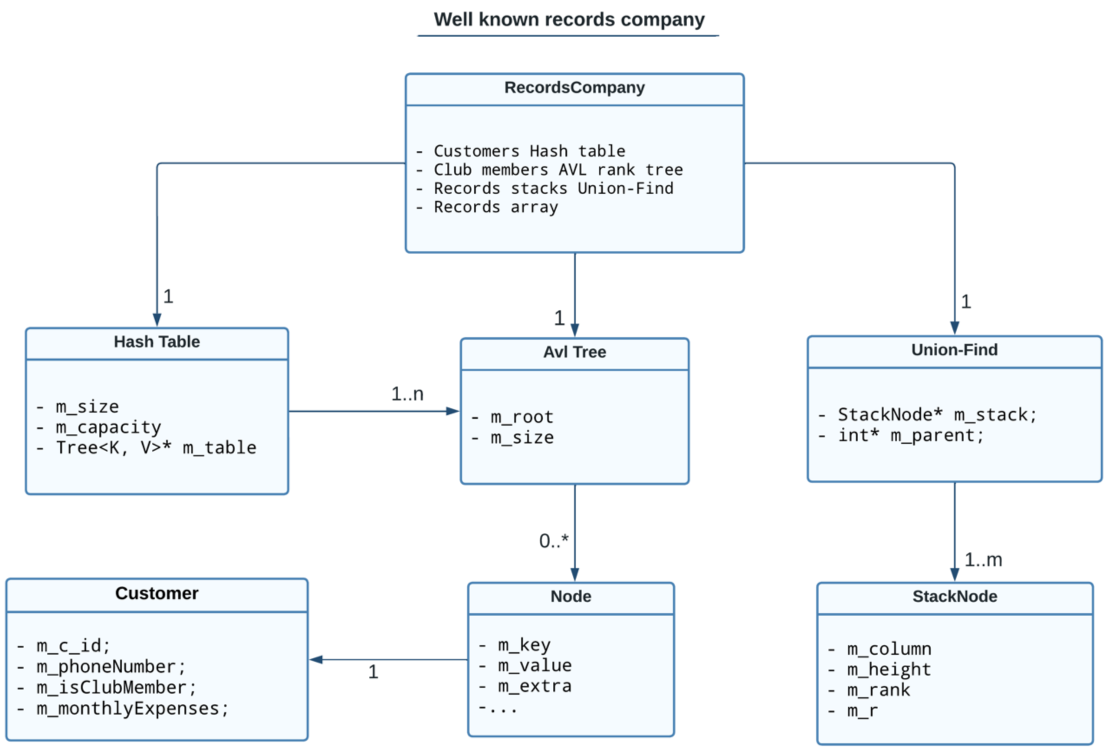
## Data structure description:



The data structure consists of the following classes:

### Customer
This class represents the customers of the record company. Each instance holds various member fields, including the total amount spent by the customer in the current month.

### Node
This class represents a node in an AVL tree. It contains standard fields such as pointers to its left and right children and its height.

### Tree
This class implements an AVL tree with modified methods to meet the needs of the database.

### HashTable

This class implements a hash table with additional methods to support certain database operations efficiently. The collision resolution is implemented using AVL trees in each bucket instead of lists, allowing for worst-case performance of O(log n).

### UnionFind

This class implements a union-find data structure to manage stacks of records. The algorithm used is based on the one presented in the tutorial.

### StackNode

This is internal class in the union find that represents stack or records.

### recordsCompany

This is the main class of the database. An instance of this class holds a hash table of customers, an AVL tree of club members, an array of records, and a union-find data structure for managing stacks of records.

## Database operations:

### RecordsCompany()

- For each data structure used in the *RecordsCompany* class, such as the *m_customers* hash table, the *m_clubMembers* tree, and the *m_recordsUF* union-find data structure, we create it empty and don't allocate or initialize anything at this stage. This takes **O(1)** time complexity.
-

Overall, for this operation we get **O(1)** time complexity.

### ~RecordsCompany()

- The destructor is responsible for releasing any resources that were allocated by the RecordsCompany object. In each data structure used by the RecordsCompany class, we hold **O(n)** customers. Records data are stored in arrays. To free all of these resources, we must go to each member and free it, and free all of the arrays storing data related to records. This won't take more than **O(n)** time complexity for the customers and **O(m)** for the records. Therefore, **O(n + m)** is the overall time complexity for the destructor.

## *StatusType* newMonth(int **records_stocks*, int *number_of_records*)

- Create a new array of size m for the records. This takes **O(1)** time complexity if we assume that array initialization takes constant time.
- Create a new union-find data structure. This involves operations on arrays such as deleting and creating new arrays for m_stack and m_parent, and initializing their values. This takes **O(m)** time complexity.
- Reset all the expenses for club members and the extra field in the AVL rank tree. This involves performing an in-order traversal of the AVL tree. This takes **O(n)** time complexity.

Overall, for this operation we get **O(n + m)** time complexity.

## StatusType addCostumer(**int** c_id, **int** phone)

- Insert the customer into the customers hash table. This takes O(1) average time complexity.

Overall, for this operation we get O(1) average time complexity.

## Output_t<int> getPhone(int *c_id*)

- Find the customer in the customers hash table. This takes O(1) average time complexity.
- Retrieve the customer's phone number. This takes O(1) time complexity.

Overall, for this operation we get O(1) average time complexity.

## *StatusType* makeMember(int *c_id*)

- Find the customer in the customers hash table. This takes **O(log n)** worst-case time complexity due to the use of AVL trees for collision resolution.
- Modify the customer's field to indicate that they are now a club member. This takes **O(1)** time complexity.
- Insert the customer into the club members AVL based rank tree. This takes **O(log n)** time complexity.

Overall, for this operation we get **O(log n)** worst-case time complexity.

## Output_t<bool> isMember(int *c_id*)

- Search for the customer in the customers hash table. This takes O(1) average time complexity.
- Check if the customer is a club member. This takes O(1) time complexity.

Overall, for this operation we get **O(1)** average time complexity.

## *StatusType* buyRecord(int *c_id*, int *r_id*)

- Search for the customer in the customers hash table. This takes **O(log n)** worst-case time complexity due to the use of AVL trees for collision resolution.
- Retrieve the number of purchases for the record from the records array. This takes **O(1)** time complexity.
- If the customer is a club member, increase their monthly expenses. This takes **O(1)** time complexity.
- Increase the record's number of purchases. This takes **O(1)** time complexity.

Overall, for this operation we get **O(log n)** worst-case time complexity.

## *StatusType* addPrize(int *c_id1*, int *c_id2*, double *amount*)

- Perform two traversals in the AVL rank tree of club members essentially performing search for both the range indices. This takes **O(log n)** time complexity.
- While performing the traversals update *extra* fields in nodes determined by the algorithm showed in tutorial. This takes **O(1)** time complexity.

Overall, for this operation we get **O(log n)** time complexity.

## Output_t<double> getExpenses(int *c_id*)

- Perform a search for the customer in the AVL rank tree of club members. During this search, sum up all of the *extra* fields in the nodes on this path. This takes **O(log n)** time complexity.

Overall, for this operation we get **O(log n)** time complexity.

## *StatusType* putOnTop(int *r_id1*, int *r_id2*)

- Perform a modified union operation on the union-find data structure holding stacks of records. During this operation, modify the heights and r fields of the roots participating in the union and modify the column field similar to the algorithm that has been shown in tutorial. As the additional operations take constant time and the union operation takes **O(log * m)** as has been shown in lecture.

Therefore overall time complexity for this operation is O(log * m).

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

- Perform a modified path compression search on the union-find data structure holding stacks of records as shown in the lecture. During this search, sum up the additional r field of each stack on the path to get the relative height of the requested stack similar to how it has been shown in tutorial. This takes **O(log* m)** time complexity.
- Retrieve the column that the stack is in from the root's stack instance. This takes **O(1)** time complexity.

Overall, for this operation we get *O(log * m)* time complexity.