Your task is to implement class `CPersonalAgenda`, which implements a simple employee database.

We assume the database holds information about company employees. Each employee is registered with his/her name, surname, e-mail, and salary. The database may retrieve and update the stored information, moreover, the database provides an interface to analyze the salaries.

Employees are identified by name and surname, or by their email. The identification is unique, i.e. e-mails are not duplicate in the database. Names and surnames alone are not unique. I.e. there may exist many employees with surname `Svoboda`, and there may exist many employees with name `Peter`. However, the pair `Svoboda Peter` is unique in the database. The database is case sensitive, i.e. names, surnames, and e-mails are considered different even if the only difference is uppercase/lowercase letter(s).

The class shall implement public interface shown below. The interface consists of:

- A constructor (without parameters). This constructor initializes a new empty database.
- Destructor -- it shall free all resources used by the instance.
- Method `Add (name, surname, email, salary )` which adds a new record to the database. The parameters are name, surname, e-mail and the salary of the employee. The method returns `true` if it succeeds, or `false` if it fails (the pair `name, surname` is already present in the database, or the `email` is not unique in the database).
- Methods `Del (name, surname) / Del (email)` removes the corresponding record from the database, the record is identified either by the name and surname, or by e-mail. The methods return `true` if they succeed, or `false` if they fail (the corresponding record was not present).
- Method `ChangeName(email, newName, newSurname)` changes employee name and surname. The employee to update is identified by e-mail, the name of the employee is changed to `newName/newSurname`. The method returns `true` to indicate success or `false` to indicate an error (the employee cannot be found, the newly assigned name/surname pair is not unique).
- Method `ChangeEmail(name, surname, newEmail)` changes employee e-mail. The employee is identified by `name/surname`, the e-mail is changed to `newEmail`. The method returns `true` to indicate success or `false` to indicate an error (the employee cannot be found, the newly assigned e-mail is not unique).
- Methods `SetSalary (name, surname, salary) / SetSalary (email, salary)` update the salary of the given employee. There are two variants of the method - the employee is identified either by the name and surname, or by the e-mail. The methods return `true` for success or `false` for a failure (employee not found).
- Methods `GetSalary (name, surname) / GetSalary (email)` return the actual salary for the given employee. There are two variants of the method - the employee is identified either by the name and surname, or by the e-mail. If the methods succeed, they return the actual employee's salary. The return value is to be 0 to indicate a failure (employee not found).

- Methods `GetRank (name, surname, rankMin, rankMax) / GetRank (email, rankMin, rankMax)` compute the rank of the employee's salary. There are two variants of the method - the employee is identified either by the name and surname, or by the e-mail. Next, there are two output parameters to be set by the function. Suppose the salaries in the database sorted in an ascending order. The goal is to find the position of the employee's salary in that list. Since the same salary may be paid to several employees, the position in the list is not a single number. In general, the position of the employee's salary may be given as an interval of values. That interval is to be filled into the output parameters `rankMin/rankMax`. For example, the searched employee is paid a salary of 20000. The example database contains 37 employees with the salary smaller than 20000 and another 21 employees with salary equal to 20000 (i.e. there is a total of 22 employees with salary 20000). Then the output parameters shall be set to `rankMin=37` and `rankMax=37+22-1=58`. The return value is `true` to indicate success (output parameters are correctly set) or `false` (the employee was not found, the output parameters were not modified).
- Method `GetFirst ( outName, outSurname )` is designed to iterate over the employees in the database. The employees are iterated in an alphabetical order, the order is given by the surname and name (surname is the first key, name is used as the second key if surnames are equal). This method starts the iteration, i.e. it fills in the name and surname of the first employee to the output parameters `outName/outSurname`. Return value is either `true` (the database is not empty, the output parameters were set) or `false` (an empty database, output parameters were not modified).
- Method `GetNext ( name, surname, outName, outSurname )` is designed to iterate over the employees in the database. This method retrieves the next employee that follows employee `name/surname` in the list (see `GetFirst`). The name of the successor is filled into the output parameters `outName/outSurname`. Return value is either `true` (success, `name/surname` was found and there exists a successor), or `false` (employee `name/surname` not found, or `name/surname` is the last employee in the list). The output parameters shall not be modified if the method fails.

Submit a source file with your `CPersonalAgenda` implementation. The class must follow the public interface below. If there is a mismatch in the interface, the compilation will fail. You may extend the interface and add you auxiliary methods and member variables (both public and private, although private are preferred). Moreover, you may add your auxiliary classes to the submitted file. The submitted file must include both declarations as well as implementation of the class (the methods may be implemented inline but do not have to). Do not add unnecessary code to the submitted file. In particular, if the file contains `main`, your tests, or any #include definitions, please, keep them in a conditional compile block. Use the attached source as a basis for your implementation. If the preprocessor definitions are preserved, the file maybe submitted to Progtest.

The interface of the class requires several methods that only differ in the way the employee is identified. Spend some time with the design of the class. Do not just copy everything twice in your implementation (for example use some common private methods).

The class is tested in a limited environment -- both memory and running time is limited. The available memory is big enough to store the records, there is quite a lot of extra memory. The class does not have to implement a copy constructor or an overloaded operator =. This functionality is not tested in this homework.

Although the memory limit is not very strict, there are some requirements on time efficiency. A simple linear-time solution will not succeed (it takes more than 5 minutes for the test data). You may assume `Add` and `Del` calls are rare compared to the other methods. Thus these two methods may run in linear time. Methods `SetSalary` and `GetSalary` are called very often, thus they must be very efficient (e.g. logarithmic time or amortized constant time). The same applies to `GetFirst` and `GetNext`. Finally, `GetRank` is not called often in the mandatory tests. Thus the implementation does not have to be very efficient (linear or `n log n` is fine to pass the mandatory tests), however, much better time complexity is required to pass the bonus test.

There are several ideas and data structures that may be used to complete the bonus test. The following facts may help to choose the implementation:

- the speed of `GetRank` is not very important unless you struggle the bonus test, however, the speed of `SetSalary` is always important,
- the maximum salary is 1000000, higher values are not used,
- duplicate values are quite often in salaries.

Either STL container or dynamic array allocation is required to implement the database. If implemented using the dynamically allocated array, set the initial size of the array to some small value (e.g. one hundred elements). When the array is full, do not increase the size by just one element. The overhead of the resizing would be enormous. Instead, increase the size by e.g. a thousand elements or use a quotient ranging from 1.5 to 2 (better solution).

If STL is used, your implementation does not have to care about the allocation. Caution: only some STL containers are available (see the headers included in the attached source). In particular, `map` / `unordered_map` / `set` / `unordered_set` / ... are disabled in this homework.

The attached source file lists some basic tests. The tests are quite short, you have to add your own tests. Please note that the tests included in the submitted files are recognized as an integral part of your solution. Thus someone else's tests included in your solution are just another form of cribbing.