Lab 5: Search Through a Database

1. Objectives

In this lab, you will be creating a data structure to help you search through data. You will create binary search trees to store data about employees in a company. The user will enter commands into the database to insert data of employees customers, search through IDs and ages and autocomplete names.

2. Problem Statement

This assignment wants you to search efficiently in a database of employees. Instead of storing data in a list or an array, we create binary search trees to store the data and help us search quickly through it.

2.1. Parser implementation

Initially, you are given starter code that **parses commands from the user**. In main.cpp, you will find

- `searchEmployee()`, `searchIDEmployee()` and `searchAgeRangeEmployee()` functions that are used to parse the commands from the user that search for an ID and age range,
- `addEmployee()` function that parses the commands that adds employees to the database,
- `autocompleteEmployee()` function that parses the command that completes the name of an employee entered,
- `getString()` function that parses strings,
- `getInt()` function that parses integers,
- `getDouble()` function that parses doubles, and
- `foundMoreArgs()` function that is used to detect whether more arguments were passed to the command than expected.

The starter code accepts commands in the following format.

- **Add an employee:** `employee <ID> <firstName> <lastName> <age> <salary>`
- **Search for an employee with a specific ID:** `search ID <ID>`
- **Search for employees within an age range:** `search age <lowAge> <highAge>`
- **Search for employees that have names with a prefix:** `autocomplete <prefix of a name>`

`<ID>` is the ID of the employee, `<firstName>` is the first name of the employee (that has no white spaces), `<lastName>` is the last name of the employee (that has no white spaces), `<age>` is the age of the employee and `<salary>` is the salary of the employee. For a user operating the database, the `<prefix of a name>` can be any part of a first name without the last name or first name followed by a space and a part of the last name of an employee. `<lowAge>` and `<highAge>` can be the minimum age and maximum age (inclusive) of an employee that the user is looking for.

The starter code already checks for invalid inputs, too many or too few arguments and you aren't required to change it.

2.2. Objects in the Database

You will implement binary search tree based data structures to organize the employees in the database for faster search.

To do so, you will implement one class: `BST`.

2.3. System Implementation

Users of the program can use the command line to insert employees into the database.

To organize the employees, we use binary search tree data structures. There are three binary search trees named `nameTree`, `IDTree` and `ageTree`. Each has the same employees, but they are ordered differently. `nameTree`, `IDTree` and `ageTree` should order the employees according to their name, ID and age respectively.

A `employee <ID> <firstName> <lastName> <age> <salary>` inserts an employee into the database. An object of Employee is created and should be inserted into all the tree binary search trees. Before insertion, the program must check if an employee with the same ID exists. If an employee with the same ID in the binary search trees exists, then program should output: `"Error: ID already exists"`. No employees should be inserted in this case. The order with which you insert an employee into `nameTree` is according to their firstName + lastName alphabetical order. None of the names will have any white spaces. You can safely assume no two employees will have the same name and if two employees have the same age, the new employee's age should be considered lower than the existing employee in the database.

A `search ID <ID>` command will search through `IDTree` tree. If the employee with that ID is found, the program will print the information of the Employee as in the print function of the Employee class.

A `search age <lowAge> <highAge>` command will search through `ageTree`. The program should print in order of increasing age the employees information that have an age from `lowAge` to `highAge`.

A `autocomplete <prefix of a name>` command will search through `nameTree`. The program should print in alphabetical order the employees who have `<prefix of a name>` as a prefix in their firstName + lastName.

2.4. Program Classes and Files

2.4.1. The Employee Class

The Employee class holds the information of an employee, including their ID, first name, last name, age, salary and two pointers left and right pointing to the left and right child in the binary search tree. The definition of the class appears in `Employee.h`. Examine the file and read through the comments to understand the variables and methods of the class. This class is implemented in the file `Employee.cpp`.

2.4.2. The BST Class

This class defines a binary search tree of Employee. It contains two data members, `root`, which points to the root `Employee` in the binary search tree, and `order`, which is a string defined to ID, name or age based on how the binary search tree is organized. It is defined in the file `BST.h` and the file contains comments that describe what the methods of the class do. One comment is inaccurate about `searchID` function. It should**n't** print `"<ID> ID does not exist"` if ID doesn't exist in the function. Instead it should return NULL, and the printing should be taken care of in the `main.cpp`. You must implement this class in `BST.cpp`.

2.4.3. The main program

The main function in main.cpp runs the program. The main.cpp has three binary search trees. Each has the same employees, but they are ordered differently. `nameTree`, `IDTree` and `ageTree` should order the employees according to their name, ID and age respectively. Whenever an employee enters the database, their object should be inserted into all the tree binary search trees. Searching through different trees will depend on the information we are searching for in employees. For example, whenever we need to look for an ID, we will use the `IDTree`.

3. Coding Requirements

1. The code you will write shall be contained in only the source files ending in ".cpp": main.cpp and BST.cpp. Skeletons of these files are released with the assignment's zip file. The zip file also contains corresponding ".h" files: Employee.h, Employee.cpp and BST.h. These ".h" files contain comments that describe the classes defined in them and their methods. Please note that the ".h" files are **NOT** to be modified in any way. Modifying these files almost always results in a mark of 0 for the assignment.

   You may make use of helper functions to split up the code for readability and to make it easier to re-use. These functions (and their prototypes) must be in one of the

aforementioned ".cpp" files. That is, you must not add any new ".h" or ".cpp" files.

2. Input and output must be done **only** using the C++ standard library streams `cin` and `cout`.

3. The stream input operator `>>` and associated functions such as `fail()` and `eof()` shall be used for all input. C-style IO such as printf() and scanf() shall not be used.

4. Strings shall be stored using the C++ library type `string`, and operations shall be done using its class members, not C-style strings.

5. The Standard Template Library (STL) shall not be used.

6. Your program shall not leak memory. You will be penalized for leaking memory.

4. Examples

```
Welcome to our employee database!
The database is empty now!
>>> employee 201 Issac Newton 31 62000
>>> employee 202 Albert Einstein 30 80000
>>> employee 202 Albert NotEinstein 23 90000
Error: ID already exists
>>> employee 203 Steve Jobs 34 120000
>>> employee 204 Lionel Messi 32 10982
>>> employee 209 Elon Musk 19 3000
>>> search ID 202
Name: Albert Einstein
ID: 202
Age: 30
Salary: 80000
>>> search ID 200
200 ID does not exist
>>> employee 205 Chris Evan 32 20000
>>> employee 206 Chris Evans 23 22000
>>> employee 207 Chris Williams 35 80000
>>> autocomplete Chr
Name: Chris Evan
ID: 205
Age: 32
Salary: 20000
Name: Chris Evans
ID: 206
Age: 23
Salary: 22000
Name: Chris Williams
ID: 207
Age: 35
Salary: 80000
>>> search age 23 34
Name: Chris Evans
ID: 206
Age: 23
Salary: 22000
Name: Albert Einstein
ID: 202
Age: 30
Salary: 80000
```

```
Name: Issac Newton
ID: 201
Age: 31
Salary: 62000
Name: Chris Evan
ID: 205
Age: 32
Salary: 20000
Name: Lionel Messi
ID: 204
Age: 32
Salary: 10982
Name: Steve Jobs
ID: 203
Age: 34
Salary: 120000
>>> ^D
```

5. Procedure

Create a sub-directory called `lab5` in your ece244 directory, and set its permissions so no one else can read it. Download the `lab5_release.zip` file, un-zip it and place the resulting files in the `lab5` directory.

In the release, there are 2 ".h" files: Employee.h and BST.h. The files define the various classes, as described in Section 2.6. **You may NOT modify these files** to add to or delete from their content. Modifying the files commonly results in a mark of zero for the assignment.

In the release, there are also 3 ".cpp" files in which you will add your code: main.cpp, BST.cpp and Employee.cpp. In the first file, you will complete the code that implements the database search. In the rest you will implement the various classes described in Section 2.6.

There is a reference executable released with the assignment to help you better understand the specifications. You can run it using this command ~ece244i/public/database-ref on ECF. When in doubt, run the executable to observe its behaviour.

To run your own executable, run make then ./database. The public test cases are in files named 1, 2, 3, 4 in the lab5_release.zip file. You may channel all the input from file 1, for example, by running ./database < 1.

The ~ece244i/public/exercise command will also be helpful in testing your program. You should exercise the **executable**, i.e., database, using the command: ~ece244i/public/exercise 5 database

As with previous assignments, some of the exercise test cases will be used by the autotester during marking of your assignment. We will not provide all the autotester test cases in exercise, however, so you should create additional test cases yourself and ensure you fully meet the specification listed above.

6. Deliverables

Submit the `main.cpp` and `BST.cpp` files as lab 5 using the command ~ece244i/public/submit 5