Jingyi Bai
267936

The reasons for selecting the data structures

Firstly, to save the data, I defined a Tree struct named Person, which has four contents, a vector of Person pointer to save all children (lower ranks), a Person pointer to save parent. Four contents are name, id, title, and salary. The Tree structure is more convenience for saving the company relationship. Each person has one direct boss (parent), but each boss may have several employees (children). *I was thinking that maybe the structure of children changes to unordered_map would be better, the performance may be better than this one, but I don't have time to reconstruct my codes.*

```
struct Person {
    string name;
    PersonID id;
    string title;
    Salary salary;
    vector<Person*> children;
    Person* parent = NULL;
};
```

Then, there are two vector<Person*> which saving pointers of Person, named vperson_ and vperson_name_. vperson_ is used to save pointers of Person ordered by salary, vperson_name_ is for pointers of Person ordered by name. There also an unordered_map<PersonID, Person*> called m_, which is for quickly getting information of Person. The reason using unordered_map here is this structure could be very fast in search by key. Vector has to traversal all data but the time complexity of unordered_map to do the same search is O(1). The main function will call get_name() function or other similar functions so many times, and unordered_map can save a lot of searching time.

```
vector<Person*> vperson_;
vector<Person*> vperson_name_;
unordered_map<PersonID, Person*> m_;
```

There're three Person* struct, two of them save the pointer of Person which has maximum and minimum salary, and p_ is for assigning the value when adding person.

```
Person* pmax_salary_;
Person* pmin_salary_;
Person* p_;
```

In the end, four bool type flags.

```
bool sorted_name_;
bool sorted_salary_;
bool max_salary_removed_;
bool min_salary_removed_;
```

Estimation of the performance of each operation

add_person: **O(1)**

remove_person: **O(1)**

get_name, get_title, get_salary: **O(1)**

find_persons, personnel_with_title: **O(nlogn)**

copy_if: O(n)

for loop: O(n)

sort: Θ(nlogn), this one only sort part of vector, not for the whole vector, so the time complexity will smaller than O(nlogn).

change_name, change_salary: **O(1)**

add_boss: **O(1)**

size, clear: **O(1)**

underlings: **O(mlogm)**

for loop: O(m), m<n, m is the number of children of n

sort: O(mlogm)

personnel_alphabetically, personnel_salary_order: **O(n)**

sort: O(nlogn)

while loop: O(n)

find_ceo: **O(n)**

nearest_common_boss, higher_lower_ranks: **O(m)**

while loop: O(m), m<n, m is the number of all parents of n

count_all_children, nodes_of_higher_ranks, nodes_of_lower_ranks: **O(m)**

while loop: O(m), m<n, m is the number of children of n

min_salary, max_salary, median_salary, first_quartile_salary, third_quartile_salary: **O(nlogn)**

sort: O(nlogn)