TIE-20106 Data structures and algorithms, spring 2017

Programming assignment 3: It's a social netowork, baby!

Last modified on 03.04.2017

Note: Since this programming assignment is based on the previous one, in the text below all new / changed things are marked with a grey background.

Topic of the assignment

The third assignment expands the first two assignments by adding support for friendships between people. *Backstory: The company has noticed, that sometimes it is necessary to manipulate employees by spreading rumours. In Finland the rumours spread best among friends, especially if malt beverages are involved. So the company has begun to keep track of friendships between its employees, including how many pints are needed in each friendship (after this simply "cost") before rumours start to spread. For simplicity it's assumed that the cost is the same in a friendship no matter in which direction the rumour spreads.*

In addition to adding and removing friendships, the following queries are added to the system:

- Who are the immediate friends of an employee, and what is the cost for each friendship.
- A list of all friendships between employees.
- What is the *smallest amount* of friendships (and pub nights) in order to spread a rumour from one employee to another, and what is the total cost of this operation (in case the employer wants to verify spreading the rumour by paying for beverages).
- What is the *smallest total cost* and route to spread a rumour from one employee to another. This route may include more friendships than the previous one (and thus be slower), but it's the most cost-effective.
- If the rumour should be spread to the whole personnel, what friendships should be used to do this with minimal total cost. Because the friendships do not necessarily form a single connected network, it's possible that the result is not connected either (in which case the employer has to start the rumour with several employees). In this assignment this operation is implemented by removing all friendships that are not part of this "cost-optimized" friendship network.

In the second assignment the first programming assignment is expanded to boss relationships and a couple of other new things. Employees are now identified by a unique ID. In addition to this the program can produce other salary-related statistics, which are used in real life as well: minimum and maximum salaries, median salary and salary quartiles (definitions of these can be found later in this document).

(Side note: medians and quartiles tell more about salaries than average and standard deviation, because salaries are usually really unevenly distributed. This means that a small number of billionaires can make the average unrealistically high compared to "normal" salaries).

Since this is a Data structures and algorithms assignment, performance of the program is an important grading criteria. The goal is to code an as efficient as possible implementation, under the assumption that all operations are executed equally often (unless specified otherwise on the command table). Getting better performance helps to increase the grade. Especially note the following:

- In this assignment *only the new operations and updated parts of the old operations* are graded.
- The compulsory parts of the third assignment are add_friendship(), all_friendships(), remove_friendship(), get_friends(), and shortest_friendpath(). Operations check_boss_hierarchy(), cheapest_friendpath() ja leave_cheapest_friendforest() are not compulsory to pass the assignment. However, they are part of grading, so not implementing them affects the grade!
- A hint in implementing the non-compulsory operations: The educated guess of the lecturer is that check_boss_hierarchy() is the easiest to implement, followed by cheapest_friendpath(), and then leave_cheapest_friendforest() as the most difficult. For the last operation you might have to think and search for a suitable algorithm yourself. Such an algorithm is met during the course, but not necessarily on lectures.
- It's quite possible that there's not much you can do to affect the asymptotic performance in the third assignment, because the performance is dictated by the algorithms. That's why the implementation, quality and correctness of the algorithms is the main focus in grading. However, to help you there will be performance tests in the third assignment as well, a separate test for each query.
- **Hint** about (not) suitable performance: If the performance of any of your operations is worse than $\Theta(n \log n)$ on average, the performance is definitely *not* ok. Most operations can be implemented much faster. In the third assignment some operations can necessarily be slower.
- **As part of the programming submission in git there has to be a file "readme.pdf". The document should explain the reasons for selecting the data structures used in the implementation, especially in regard to performance. It also includes your estimate of the performance of each operation you've implemented using the big-O notation. In the third assignment this document has to be updated only for the new and updated operations.**
- Implementing operations `remove()`, `nearest_common_boss()`, and `higher_lower_ranks()` is not compulsory to pass the assignment. However, they are part of grading, so not implementing them affects the grade of the second assignment!

- In performance the essential thing is how the execution time changes with more data, not just the measured seconds.
- The performance of an operation includes all work done for the operation, also work possibly done during addition of elements.
- More points are given if operations are implemented with better performance.
- Likewise more points will be given for better real performance in seconds (if the required minimum asymptotic performance is met). **But** points are only given for performance that comes from algorithmic choices and design of the program. (For example setting compiler optimization flags, using concurrency or hacker optimizing individual C++ lines don't give extra points).
- If the implementation is bad enough, the assignment might not pass.
- Examples of questions, which may help in improving the performance: Is somewhere the same thing re-done several times? Can you sometimes avoid doing something completely? Does some part of the code do more work than is absolutely necessary? Can some things be done "almost free" while doing something else?

Definitions of the statistics used in the assignment

In this assignment the following statistics are used to get information about salaries. To make the assignment easier, these definitions are a little simplified compared to the "real" mathematical definitions. Below "salary order" means salaries ordered from smallest to largest.

- *Minimum / maksimum:* Person with the smallest/largest salary. If there are more than one such person, any one of them can be chosen.
- *Median:* The middle person in salary order. In this assignment the person with index $\lfloor \frac{n}{2} \rfloor$. ($\lfloor x \rfloor$ is rounding down to nearest integer, i.e. the normal C++ rounding in integer division).
- *First quartile:* The person in the 1/4 position in salary order. In this assigment the person with index $\lfloor \frac{n}{4} \rfloor$.
- *Third quartile:* The person in the 3/4 position in salary order. In this assigment the person with index $\lfloor \frac{(3n)}{4} \rfloor$.

On sorting

While sorting employees in salary order it's possible that several persons have the same salary (or the same name when sorting alphabetically). The mutual ordering of such persons doesn't matter.

The main program only accepts names consisting of letters A-Z, a-z and spaces. The alphabetical sorting can be done either with the regular < comparison of C++ string class (in which case space comes first, then upper case letters followed by lower case letters) or the "correct way", where upper and lower case letters are equal, but space still comes before letters.

When sorting employee IDs, C++ string comparison "<" should be used.

About implementing the program and using C++

The purpose of this programming assignment is to learn how to use ready-made data structures and algorithms, so using C++ STL is very recommended and part of the grading. There are no restrictions in using C++ standard library. Naturally using libraries not part of the language is not allowed (for example libraries provided by Windows, etc.).

NOTE that since the purpose of this programming exercise is to practise using STL, it's *very likely* that the data structures you used in the first programming assignment are NOT good solutions in this one. Similarly consider where you could replace your own algorithm implementations of the first programming assignment with ready-made STL algorithms!

Structure and functionality of the program.

Part of the program code is provided by the course, part has to be implemented by students.

Parts provided by the course

File *main.cpp* (you are **NOT ALLOWED TO MAKE ANY CHANGES TO THIS FILE)**

- Main routine, which takes care of reading input, interpreting commands and printing out results. The main routine contains also commands for testing.

File *datastructure.hpp*

- `class Datastructure`: The implementation of this class is where students write their code. The public interface of the class is provided by the course. You are **NOT ALLOWED TO CHANGE THE PUBLIC INTERFACE** (i.e. change names, return type or parameters of the given public member functions).
- Type definition `Salary`: integer. Constant `NO_SALARY` is defined for situations, where for example the salary is not found or known.
- Type definition `PersonID`: string consisting of numbers 0-9 and letters a-z and A-Z. merkkijono, joka koostaa numeroista 0-9 ja kirjaimista välillä a-z tai A-Z. Constant `NO_ID` is defined for situations, where person with a suitable ID cannot be found.

File *datastructure.cpp*

- Function `random_in_range`: Like in the first assignment, returns a random value in given range (start and end of the range are included in the range).

Parts of the program to be implemented as the assignment

Files *datastructure.hpp* and *datastructure.cpp*

- class Datastructure: The given public member functions of the class have to be implemented. You can add your own stuff into the class (new data members, new member functions, etc.)

Note! The code implemented by students should not do any output related to the expected functionality, the main program does that. If you want to do debug-output while testing, use the cerr stream instead of cout, so that debug output does not interfere with the tests.

Commands recognized by the program and the public interface of the Datastructure class

When the program is run, it shows prompt ">" and waits for commands explained below. The commands where the explanation mentions a member function, call that member function of the Datastructure class (implemented by students). Other commands are completely implemented by the code provided by the course.

If the program is given a file as a command line parameter when it is run, the program reads commands from that file and then quits.

Command **`Public member function`**	**Explanation**
add 'name' id 'title' salary `void add_person(string name, PersonID id, string title, Salary salary);`	Adds an employee to the data structure with given name, unique id, title and salary. At first an employee does not have a boss. This operations is also tested in hw3.
remove id `void remove_person(PersonID id);`	Removes a person with the given ID. If such a person has not been added, does nothing. If the person to be removed has underlings, they become the underlings of the boss of the person to be removed. If there's no boss, the underlings will not have a boss after removal. If a person with given ID does not exist, nothing happens. When an employee is removed, all friendships related to him/her are also removed. *Implementing this command is not compulsory (but is taken into account in grading of 2nd assignment).*
add_boss id bossid `void add_boss(PersonID id, PersonID bossid);`	Gives an employee a boss. An employee can have at most one boss. You can assume that boss relationships do not form cycles (i.e., a person cannot directly or indirectly be his/her own boss).

`(no command)`
`string get_name(PersonID id);`
	Returns the name of an employee with given ID. (Main program calls this in various places.) *This operation is called more often than others. This operations is also tested in hw3.*

`(no command)`
`string get_title(PersonID id);`
	Returns the title of an employee with given ID. (Main program calls this in various places.) *This operation is called more often than others. This operations is also tested in hw3.*

Command **`Public member function`**	**Explanation**

`(no command)`
`Salary get_salary(PersonID id);`
	Returns the salary of an employee with given ID. (Main program calls this in various places.) *This operation is called more often than others. This operations is also tested in hw3.*

size unsigned int size();	Returns the number of employees currently in the data structure. This operations is also tested in hw3.
clear `void clear();`	Clears out the data structure (after this size returns 0). This operations is also tested in hw3.

`find 'name'`
`vector<PersonID> find_persons(string name);`
	Returns a list (vector) of persons with the given name. The list has to be sorted to increasing ID order.

`titlelist 'title'`
`vector<PersonID> personnel_with_title(string title);`
	Returns a list (vector) of persons with the given title. The list has to be sorted to increasing ID order. *This operation is called seldom, and it is not part of the performance tests.*

`change_name id 'new name'`
`void change_name(PersonID id, string new_name);`
	Changes the name of the person with given ID.

`change_salary id newsalary`
`void change_salary(PersonID id, Salary new_salary);`
	Changes the salary of the person with given ID.

alphalist `vector<PersonID> personnel_alphabetically();`	Returns a list (vector) of the employees in alphabetical order.
salarylist `vector<PersonID> personnel_salary_order();`	Returns a list (vector) of the employees in salary order (smallest first).
min `PersonID min_salary();`	Returns the employee with the smallest salary (definition of minimum is elsewhere in this document).
max `PersonID max_salary();`	Returns the employee with the largest salary (definition of maximum is elsewhere in this document).
median `PersonID median_salary();`	Returns the employee with the median salary (definition of median is elsewhere in this document).
1stquartile `PersonID first_quartile_salary();`	Returns the employee with the 1st quartile salary (definition of 1st quartile is elsewhere in this document).
3rdquartile `PersonID third_quartile_salary();`	Returns the employee with the 3rd quartile salary (definition of 3rd quartile is elsewhere in this document).
underlings id `vector<PersonID> underlings(PersonID id);`	The *command* prints out an indented list of person's underlings and their underlings. **NOTE!** The *member function* only returns a list of **direct** underlings. The list has to be sorted to increasing ID order. The main program calls the member function recursively to print the hierarchy.

Command **`Public member function`**	**Explanation**
ceo `PersonID find_ceo();`	Returns the CEO of the organization (person whose direct or undirect underlings *all* employees are), and make sure that there is exactly one such person. If not, returns NO_ID.
nearest_common_boss id1 id2 `PersonID nearest_common_boss(PersonID id1, PersonID id2);`	Returns the nearest common boss of two employees, or NO_ID if no such boss exists. *Implementing this command is not compulsory (but is taken into account in grading of 2nd assignment).*
higher_lower_ranks id `pair<unsigned int, unsigned int> higher_lower_ranks(PersonID id);`	Returns a pair, whose first number tells how many employees with higher rank the organization has, and the second number tells how many employess with lower rank. The "rank" of an employee tells, how many bosses away CEO is from the person. The rank of the CEO is highest, the rank of his/her underlings is one less, etc. If the organization does not form a single hierarchy, the return value can be anything. *Implementing this command is not compulsory (but is taken into account in grading of 2nd assignment).*
add_friend id friendid cost `void add_friendship(PersonID id, PersonID friendid, Cost cost);`	Adds a friendship between employees. The friendship has the given cost. The friendship is mutual (and the cost in both directions is the same). If either id does not exist, nothing is done.
remove_friend id friendid `void remove_friendship(PersonID id, PersonID friendid);`	Removes a friendship between given employees. If the friendship or either employee does not exist, nothing is done.
friends_of id `vector<pair<PersonID, Cost>> get_friends(PersonID id);`	Returns a list of the friends of an employee, and the costs associated with the friendships. The list is ordered according to ascending id.
all_friendships `vector<pair<PersonID, PersonID>> all_friendships();`	Returns a list of all friendships in the organization. Each friendship is in the list only once and expressed so that the smaller valued id in a friendship is first in the pair. The list is sorted primarily by the first id, then the second (this is how comparison of std::pair works by default). *The performance of this operation is not measured or graded, it exists to make debugging and testing easier.*
shortest_friendpath `vector<pair<PersonID, Cost>> shortest_friendpath(PersonID fromid, PersonID toid);`	Returns the *shortest* "friendship route" from employee fromid to employee toid. The route tells through which employees (and with which cost) a rumour can be spread with the *smallest amount of friendships*. The return value includes a list of steps (an employee and the cost to get to that employee from previous one) in correct order. If no path is found, and empty list is returned. The main program calculates the total cost.

Command **`Public member function`**	**Explanation**
check_boss_hierarchy `bool check_boss_hierarchy();`	Checks that the *boss hierarchy* forms a single hierarchy with no cycles (no-one is directly or indirectly his/her own boss), and that the hierarchy covers all employees. If this is the case, **true** is returned, otherwise **false**. (This differs from find_ceo in that find_ceo assumes that there are no cycles in the hierarchy, this operations *verifies* that there are no cycles and all employees are in a single hierarchy).
cheapest_friendpath `vector<pair<PersonID, Cost>> cheapest_friendpath(PersonID fromid, PersonID toid);`	Returns the *cheapest* "friendship route" from employee fromid to employee toid. The route tells through which employees (and with which cost) a rumour can be spread with the *smallest total cost*. The return value includes a list of steps (an employee and the cost to get to that employee from previous one) in correct order. If no path is found, and empty list is returned. The main program calculates the total cost.
leave_cheapest_friendforest `pair<unsigned int, Cost> leave_cheapest_friendforest();`	Leaves friendships that allow reaching every employee with the smallest possible total cost. If friendships do not form a single (connected) network, this is done for all parts of the network. Friendships not belonging to the result are removed. The return values tells, how many separate (interconnected) components there were in the network, and what is the total cost of the friendships that were not removed.
random_friends n	Adds a given number of friendships with random costs between random employees.
random_add n (implemented by main program)	Add n new employees with random id, name, title, and salary, and a random boss (for testing). Note! The values really are random, so they can be different for each run. This command does not add friendships.
random_seed n (implemented by main program)	Sets a new seed to the main program's random number generator. By default the generator is initialized to a different value each time the program is run, i.e. random data is different from one run to another. By setting the seed you can get the random data to stay same between runs (can be useful in debugging).
read 'filename' (implemented by main program)	Reads more commands from the given file (This can be used to read a list of employees from a file, run tests, etc.)
stopwatch on / off / next (implemented by main program)	Switch time measurement on or off. When program starts, measurement is "off". When it is turned "on", the time it takes to execute each command is printed after the command. Option "next" switches the measurement on only for the next command (handy with command "read" to measure the total time of a command file).

Command **`Public member function`**	**Explanation**
perftest all/compulsory/cmd1;cmd2... timeout friendcount n n1;n2;n3... (implemented by main program)	Run performance tests. Clears out the data structure and add n1 random employees, and 0-*friendcount* friends for each employee with a random cost. Then a random command is performed n times. The time for adding elements and runnin commands is measured and printed out. Then the same is repeated for n2 employees, etc. If any test round takes more than timeout seconds, the test is interrupted (this is not necessarily a failure, just arbitrary time limit). If the first parameter of the command is **all**, commands are selected from all commands. If it is **compulsory**, random commands are selected only from operations that have to be implemented. If the parameter is a list of commands, commands are selected from that list (in this case it's a good idea to include also random_add and random_friends so that elements are also added during the test loop).
testread 'infilename' 'outfilename' (implemented by main program)	Runs a correctness test and compares results. Reads command from file infilename and shows the output of the commands next to the exptected output in file outfilename. Each line with differences is marked with a question mark. Finally the last line tells whether there are any differences.
help (implemented by main program)	Prints out a list of known commands.
quit (implemented by main program)	Quit the program. (If this is read from a file, stops processing that file.)

"Data files"

The easiest way to test the program is to create "data files", which using "add "commands enter a list of employees into the data structure. Those files can them be read in using the "read" command, after which other commands can be tested without having to enter those employees every time by hand.

Below is an example of a data file, which can be found as *example-data.txt*:

```
# Adding people
add 'Meikkis Matti' mm 'basic worker' 2000
add 'Teikkis Terttu' tt 'technical evangelist' 4000
add 'Miljoona Miikka' richbastard 'commander' 1000000
add 'Sorrettu Sami' doesall 'general utility' 1
add 'Keskiverto Keijo' kk1 'basic worker' 3000
add 'Kukalie Kirsi' kk2 'basic worker' 2500
add 'Olematon Oskari' nobody 'useless' 6000
# Adding boss relationships
add_boss mm richbastard
add_boss doesall mm
add_boss nobody mm
add_boss tt richbastard
add_boss kk1 tt
add_boss kk2 tt
# Add friendships
add_friend doesall nobody 1
add_friend mm doesall 3
add_friend tt mm 5
add_friend tt kk2 2
add_friend kk1 nobody 1
add_friend kk2 kk1 2
add_friend kk2 mm 4
add_friend richbastard mm 8
add_friend richbastard tt 2
```

Example run

Below is an example output from the program. The example's commands can be found in file *example-in.txt* and the output in file *example-out.txt*. This means you can use this example also as a test by running the program and giving command *testread 'example-in.txt' 'example-out.txt'*.

```
> clear
Cleared all persons
> size
Number of employees: 0
> read 'example-data.txt'
** Commands from 'example-data.txt'
> # Adding people
> add 'Meikkis Matti' mm 'basic worker' 2000
> add 'Teikkis Terttu' tt 'technical evangelist' 4000
> add 'Miljoona Miikka' richbastard 'commander' 1000000
> add 'Sorrettu Sami' doesall 'general utility' 1
> add 'Keskiverto Keijo' kk1 'basic worker' 3000
> add 'Kukalie Kirsi' kk2 'basic worker' 2500
> add 'Olematon Oskari' nobody 'useless' 6000
> # Adding boss relationships
> add_boss mm richbastard
> add_boss doesall mm
> add_boss nobody mm
> add_boss tt richbastard
> add_boss kk1 tt
> add_boss kk2 tt
> # Add friendships
> add_friend doesall nobody 1
> add_friend mm doesall 3
> add_friend tt mm 5
> add_friend tt kk2 2
> add_friend kk1 nobody 1
> add_friend kk2 kk1 2
> add_friend kk2 mm 4
> add_friend richbastard mm 8
> add_friend richbastard tt 2
>
```

```
** End of commands from 'example-data.txt'
> size
Number of employees: 7
> all_friendships
doesall - mm
doesall - nobody
kk1 - kk2
kk1 - nobody
kk2 - mm
kk2 - tt
mm - richbastard
mm - tt
richbastard - tt
> friends_of tt
id kk2 : basic worker Kukalie Kirsi, salary 2500 (cost 2)
id mm : basic worker Meikkis Matti, salary 2000 (cost 5)
id richbastard : commander Miljoona Miikka, salary 1000000 (cost 
2)
> friends_of mm
id doesall : general utility Sorrettu Sami, salary 1 (cost 3)
id kk2 : basic worker Kukalie Kirsi, salary 2500 (cost 4)
id richbastard : commander Miljoona Miikka, salary 1000000 (cost 
8)
id tt : technical evangelist Teikkis Terttu, salary 4000 (cost 5)
> remove_friend tt mm
> friends_of tt
id kk2 : basic worker Kukalie Kirsi, salary 2500 (cost 2)
id richbastard : commander Miljoona Miikka, salary 1000000 (cost 
2)
> friends_of mm
id doesall : general utility Sorrettu Sami, salary 1 (cost 3)
id kk2 : basic worker Kukalie Kirsi, salary 2500 (cost 4)
id richbastard : commander Miljoona Miikka, salary 1000000 (cost 
8)
> shortest_friendpath richbastard nobody
Shortest path of friends is:
id mm : basic worker Meikkis Matti, salary 2000 (cost 8)
id doesall : general utility Sorrettu Sami, salary 1 (cost 3)
id nobody : useless Olematon Oskari, salary 6000 (cost 1)
Total cost is 12
> cheapest_friendpath richbastard nobody
Cheapest path of friends is:
id tt : technical evangelist Teikkis Terttu, salary 4000 ( cost2)
id kk2 : basic worker Kukalie Kirsi, salary 2500 ( cost2)
id kk1 : basic worker Keskiverto Keijo, salary 3000 ( cost2)
id nobody : useless Olematon Oskari, salary 6000 ( cost1)
Total cost is 7
> leave_cheapest_friendforest
Remaining friend forest has 1 trees with the total cost of 11
> all_friendships
doesall - mm
doesall - nobody
kk1 - kk2
```

```
kk1 - nobody
kk2 - tt
richbastard - tt
> check_boss_hierarchy
Boss hierarchy is ok.
> add 'Me' me 'great' 1000
> add 'You' you 'sucker' 1
> add_boss me you
> add_boss you me
> check_boss_hierarchy
Boss hierarchy is broken.
> quit
```