COMP.CS.300 Data structures and algorithms 1, autumn 2021

Programming project 1: Game of Taxes

Last modified on Nov 18, 2021

Table of Contents

Change log

- 11th Oct: Changed the return value of taxer_path() and longest_vassal_path(), if the given id corresponds to no town.
- 18th Oct: Added explanation of allowed town ids to the definition of TownID.
- 18th Oct: Removed space from allowed characters in a town name.
- 2nd Nov: Fixed a couple of typos. Added towns_nearest to the example run.

Topic of the project

In this programming assignment you practice implementing simple algorithms and evaluating their performance. The topic of this programming project is a program which records information about towns (name, coordinates, and collected tax revenue), and towns can be printed out in different orders, and minimum and maximum queries can be made. As a non-compulsory part, towns can also be removed. The program will contain commands for asking about the taxation relationships of the towns (see section "On calculating taxes"), as well as each town's own net tax revenue.

In practice the assignment consists of implementing a given class, which stores the required information in its data structures, and whose methods implement the required functionality. The main program and the Qt-based graphical user interface are provided by the course. (Running the program in text-only mode is also possible.)

One goal in the assignment is to practice how to efficiently use ready-made data structures and algorithms (STL), but it also involves writing one's own efficient algorithms and estimating their performance (of course it's a good idea to favour STL's ready-made algorithms/data structures when their use can be justified by performance). In grading the assignment, both the asymptotic performance and the actual performance of the code will be taken into account. The latter means that the student should implement the algorithms in an efficient and sensible manner.

"Micro optimizations" (like do I write" a = a+b;" or "a += b;", or how do I tweak compiler's optimization flags) do not give extra points.

The goal is to produce code that runs as efficiently as possible, under the assumption that all operations are executed equally often (unless specified otherwise on the command table). In many cases you'll probably have to make compromises about the performance. In those cases it helps grading when you document your choices and their justification in the **document file** that is submitted as part of the assignment. (Remember to write the document in addition to the actual code!)

In particular, note the following:

- In this assignment the emphasis is on the efficient use of ready-made data structures and algorithms (STL), so it's a good idea to prefer STL over self-written algorithms/data structures (when that's reasonable from performance point of view).
- The main program given by the course can be run either with a graphical user interface (when compiled with QtCreator/qmake), or as a textual command line program (when compiled just with g++ or some other C++ compiler). In both cases the basic functionality and students' implementation is exactly the same.
- **Hint** about (not) suitable performance: If the average performance of any of the operations in your code is worse than $\Theta(n \log n)$, this will definitely affect your grade. Most operations can be implemented much faster. *Addition: This doesn't mean that n log n would be a **good** performance for many operations. Especially for often called operations even linear performance is quite bad.*
- **As part of the assignment, file datastructures.hh contains a comment next to each operation. There you should write your estimate of the asymptotic performance of the operation, with a short rationale for your estimate.**
- **As part of the assignment submission, a document should be added to git (in the same directory/folder as the code). This document should contain reasons for choosing the data structures used in the assignment (especially performance reasons). Acceptable formats are plain text (readme.txt), markdown (readme.md), and Pdf (readme.pdf).**

- Operations `remove_town()`, `towns_nearest()`, `longest_vassal_path(), and total_net_tax()` are not compulsory to implement to pass the assignment. However, they are part of grading, so not implementing them affects the grade of the assignment! **If you only implement the compulsory parts, the maximum grade for the assignment is 3.**
- Doing the phase 2 of the programming project is not compulsory to pass the course, but in that case the maximum total grade from the course is 2 (and the sub-grade for phase 2 is recorded as 0). If phase 2 is not going to be done, it's enough to submit a "skeleton" of phase 1 as the phase 1 submiossion. In this skeleton, onlyh operations `town_count()`, `clear_all()`, `add_town()`, `get_town_...(), all_towns()`, `add_vassalship()`, and `town_vassals()` have to be implemented. Then the final phase 1 submission can be done by the phase 2 deadline. *Note that if you only submit a skeleton by phase 1 deadline, you cannot any more continue to phase 2!*
- If the implementation is bad enough, the assignment can be rejected.
- In judging runtime performance, the essential thing is how the execution time changes with more data, not just the measured seconds. 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. The student will not be given extra points for setting compiler optimization flags, using concurrency, hacker optimization of individual C++ lines or similar 'tricks'.
- The performance of an operation includes all work done for the operation, also work possibly done during addition of elements.

Definition of "distance between two points"

Some operations compare the distances of towns, either from origin (0,0) or a given coordinate. The distance between two points is calculated (to minimize rounding errors) in the following way: the "normal" (euclidean) distance between coordinates ($\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$) **is *first* rounded down to an integer**, and this rounded number is used as the distance everywhere.

On sorting

When sorting towns in terms of distance, it's possible that several towns have the same distance (or the same name when sorting alphabetically). The mutual ordering of equal elements doesn't matter in those cases.

The main program only accepts names consisting of letters A-Z, a-z, 0-9, space, and dash -. The alphabetical sorting can be done either with the regular < comparison of C++ string class (in which case upper case letters come first, followed by lower case letters) or the "correct way", where upper and lower case letters are equal.

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

On calculating taxes

In this assignment towns pay taxes to other towns. A town which pays taxes to another town, is called a *vassal town* of the town receiving the tax. Respectively, the town who receives tax from a vassal town is called a *master town*. A town can only be a vassal of one town, and these vassal relationships cannot form cycles (i.e., a town cannot directly or indirectly be it's own vassal).

When a town is added, the amount it collects taxes from its citizens is specified (as an integer). In addition to this tax amount, the town receives 10 % of the total tax of each of its vassal towns (including the taxes the vassals towns possibly receive from their own vassals). From this total tax (taxes from citizens + taxes from vassals) the town pays tax to its own master town (if it has a master). In calculating the taxes, the 10 % is always rounded down to the nearest integer (just like C++ integer division does by default), and this rounded down amount is subtracted from town's total tax amount. The resulting number after adding vassal taxes and subtracting tax to master is called the town's *total net tax*.

Example: Town *x* (tax 20) has two vassals, *v1* (tax 10) and *v2* (tax 5), which do not have vassals of their own. Additionally town *x* has a master town *m.* In this case, town *v1's* total net tax is 9 (because 10 % is paid as tax to *x*), town *v2's* total net tax is 5 (because 10 % of 5 is rounded down to zero). Town *x's* total net tax is 19 (20 + 1 from v*1* (and 0 from v*2*) = 21, from which 10 % (rounded down to 2) is paid to *m*, which makes 19 in total for *x*'s total net tax).

About implementing the program and using C++

The programming language in this assignment is C++17. The purpose of this programming assignment is to learn how to use ready-made data structures and algorithms, so using the C++ STL is highly recommended and is considered in the grading. There are no restrictions in using the C++ standard library. Using libraries outside the C++ standard library is not allowed (for example libraries provided by Windows, etc.). *Please note however, that it's very likely you'll have to implement some algorithms completely by yourself.*

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

Files *mainprogram.hh, mainprogram.cc, mainwindow.hh, mainwindow.cc, mainwindow.ui*:

- You are **NOT ALLOWED TO MAKE ANY CHANGES TO THESE FILES.**
- The main routine/command parser, which takes care of reading input, interpreting commands and printing out results. The main routine contains also commands for testing.
- If you compile the program with QtCreator or qmake, you'll get a graphical user interface, with an embedded command interpreter and buttons for pasting commands, file names, etc

in addition to keyboard input. The graphical user interface also shows a visual representation of the towns, their vassal relationships, results of operations, etc.

File *datastructures.hh:*

- `class Datastructures`: 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, etc.) Of course you are allowed to add new methods and data members to the private side.
- Type definition `TownID`. This is used as a unique identifier for each town (and which is used as a return type for many operations). There can be several towns with the same name (and even the same coordinates), but every town has a different id. (The type is a string, and the main routine allows names to contain characters a-z, A-Z, and 0-9.)
- Type definition `NAME`. This is used for names of towns, for example. (The type is a string, and the main routine allows names to contain characters a-z, A-Z, 0-9, and dash -.)
- Constants `NO_TOWNID`, `NO_NAME`, `NO_COORD`, and `NO_VALUE`. These are used as return values, if information is requested for a town that doesn't exist.
- Type definition `Coord`. This is used in the public interface to represent (x,y) coordinates. (As an example, some operations have been implemented for this type.)
- Exception class `NotImplemented`, which every not-yet-implemented operation intially throws. This way the main program can report on operations, which are still unimplemented (or are non-compulsory operations, which are intentionally left not implemented).

File *datastructures.cc*

- You write the implementation of the class methods here.
- Function `random_in_range` returns a random integer in given range (start and end of the range are included in the range). You can use this function if your implementation requires random numbers.

On using the graphical user interface

When compiled with QtCreator, a graphical user interface is provided. It allows running operations and test scripts, and visualizing the data. In phase 1 the UI has some disabled (greyed out) controls, that are part of phase 2.

The UI has a command line interface, which accepts commands described later in this document. In addition to this, the UI shows graphically created towns and vassal relationships (*if you have implemented necessary operations, see below*). The graphical view can be scrolled and zoomed. Clicking on a town prints out its information, and also inserts the ID on the command line (a handy way to give commands ID parameters). The user interface has selections for what to show graphically.

Note! The graphical representation gets all its information from student's code! **It's not a depiction of what the "right" result is, but what information student's code gives out.** The UI uses the operation `all_towns`() to get a list of towns, and asks their information with the `get_...`() operations. If the option to draw vassal relationships is on, they are obtained with the operation `get_town_vassals`().

Parts of the program to be implemented as the assignment

Files *datastructure.hh* and *datastructure.cc*

- `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.)
- **In the file *datastructures.hh*, for each member function you implement, write an estimation of the asymptotic performance of the operation (with a short rationale for your estimate) as a comment above the member function declaration.**

Additionally the readme document mentioned in the beginning is written as a part of the assignment.

Note! The code implemented by the student 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` (or `qDebug`, if you you use Qt), so that debug output does not interfere with the tests. **Please remember to remove debug outputs from your final submission!**

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

When the program is run, it waits for the user to input one or more of the commands given in the table below. The commands, whose explanation mentions a member function, call the respective member function of the Datastructure class. Some commands are completely implemented by the code provided by the course.

If the program is given a file as a command line parameter, the program executes commands from that file and then quits.

The operations below are listed in the order in which we recommend them to be implemented (of course you should first design the class taking into account all operations).

Command **`Public member function`**	**Explanation**
town_count `unsigned int town_count();`	Returns the number of towns currently in the data structure.
clear_all `void clear_all();`	Clears out the data structure (after this `town_count` returns 0).

Command `Public member function`	Explanation
add_town id nimi (x,y) tax `bool add_town(TownID id, std::string const& name, Coord coord, int tax);`	Adds a town to the data structure with the given unique id, name, coordinates, and tax. New added towns are initially not vassals of any town. If there already is a town with the given id, nothing is done and `false` is returned, otherwise `true` is returned.
(print_town id) `Name get_town_name(TownID id);`	Returns the name of the town with the given ID, or NO_NAME if such a town doesn't exist. (The main program calls this in various places.) *This operation is called more often than others.* (Command print_town calls all three get_ operations.)
(print_town id) `Coord get_town_coordinates(TownID id);`	Returns the coordinates of the town with given ID, or NO_COORD, if such town doesn't exist. (The main program calls this in various places.) *This operation is called more often than others.* (Command print_town calls all three get_ operations.)
(print_town id) `int get_town_tax(TownID id);`	Returns the tax of the town with given ID, or NO_VALUE if such town doesn't exist. (The main program calls this in various places.) *This operation is called more often than others.* (Command print_town calls all three get_ operations.)
all_towns `std::vector<TownID> all_towns();`	Returns a list (vector) of the towns in any (arbitrary) order. *This operation is not included in the default performance tests.*
find towns `std::vector<TownID> find_towns(Name const& name);`	Returns towns with the given name, or an empty vector, if no such towns exist. The order of IDs in the return value can be arbitrary. *The performance of this operation is not critical (it is not expected to be called often), so it's not part of default performance tests.*
change_town_name id newname `bool change_town_name(TownID id, Name const& newname);`	Changes the name of the town with the given ID. If such a town doesn't exist, returns `false`, otherwise `true`.
towns_alphabetically `std::vector<TownID> towns_alphabetically();`	Returns the towns in alphabetical order.
towns_distance_increasing `std::vector<TownID> towns_distance_increasing();`	Returns the towns in increasing order of distance from the origin (0,0). *Note the definition of "distance" earlier in this document.*
mindist `TownID min_distance();`	Returns the town nearest to origin (0,0). If there are several such towns with the same distance, returns one of them. If no towns exist, NO_TOWNID is returned. *Note the definition of "distance" earlier in this document.*

Command `Public member function`	Explanation
maxdist `TownID max_distance();`	Returns the town furthest away from origin (0,0). If there are several such towns with the same distance, returns one of them. If no towns exist, NO_TOWNID is returned. *Note the definition of "distance" earlier in this document.*
add_vassalship vassalid masterid `bool add_vassalship(TownID vassalid, TownID masterid);`	Adds a vassal relationship between towns. A vassal town can only have one master town. You can assume that vassal relationships do not form cycles (i.e., a town cannot directly or indirectly be its own vassal). If either of the towns do not exist, or the vassal to be added already has a master, nothing is done and `false` is returned. Otherwise `true` is returned.
town_vassals id `std::vector<TownID> get_town_vassals(TownID id);`	Returns the IDs of the towns that are immediate vassals of the given town (i.e., vassals of vassals are *not* included), or a vector with *a single element* NO_TOWNID, if no town with the given ID exists. The order of IDs can be arbitrary. (Main program calls this in various places.)
taxer_path id `std::vector<TownID> taxer_path(TownID id);`	Returns a list of towns to which the town pays taxes either directly or indirectly. The return vector first contains the town, then its master, the master of the master, etc. as long as there are masters. If the ID does not correspond to any town, a vector with *a single element* NO_TOWNID is returned.
remove_town id `bool remove_town(TownID id);`	Removes a town with the given id. If a town with the given id does not exist, does nothing and returns `false`, otherwise returns `true`. If the town to be removed has vassal towns and a master town, the vassals of the removed town become vassals of the master of the removed town. If the town to be removed has no master, after removal its vassals are also without a master. *The performance of this operation is not critical (it is not expected to be called often), so it's not part of default performance tests. Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*
towns_nearest (x,y) `std::vector<TownID> towns_nearest(Coord coord);`	Returns a list (vector) of the towns in increasing order of distance from the given coordinate (*x*,*y*) (nearest first). *Note the definition of "distance" earlier in this document. Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

Command `Public member function`	Explanation
longest_vassal_path id `std::vector<TownID> longest_vassal_path(TownID id);`	Returns the longest possible chain of vassals, starting from the given town. The return vector first contains the town, then its vassal, the vassal of the vassal, etc., so that the chain has as many towns as possible (if there are several vassal chains with equally many towns, any such chain may be returned). If the ID does not correspond to any town, a vector with *a single element* NO_TOWNID is returned. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*
total_net_tax id `int total_net_tax(TownID id);`	Returns the town's total net tax (which is explained earlier in this document). If the ID does not correspond to any town, NO_VALUE is returned. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*
random_add n (implemented by main program)	Add *n* new towns with a random id, name, coordinates, and tax (for testing). With 80 % probability the town is also added as a vassal of a random town. Note! The values really are random, so they can be different for each run.
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' [silent] (implemented by main program)	Reads more commands from the given file. If the optional parameter ‘silent’ is given, outputs of the commands are not displayed. (This can be used to read a list of things from a file, run tests, etc.)
stopwatch on / off / next (implemented by main program)	Switch time measurement on or off. When the program starts, measurement is "off". When it is turned "on", the time it takes to execute each command is printed after the command. The "next" option 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 n n1;n2;n3... (implemented by main program)	Run performance tests. Clears out the data structure and adds *n1* random towns (see random_add). Then a random command is performed *n* times. The time for adding elements and running commands is measured and printed out. Then the same is repeated for *n2* elements, etc. If any test round takes more than *timeout* seconds, the test is interrupted (this is not necessarily a failure, just an 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 so that elements are also added during the test loop). If the program is run with a graphical user interface, the "stop test" button can be used to interrupt the performance test (it may take a while for the program to react to the button).
testread 'infilename' 'outfilename' (implemented by main program)	Runs a correctness test and compares results. Reads commands from file infilename and shows the output of the commands next to the expected 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 just add towns. Such data files can then be read in using the "read" command, after which other commands can be tested without having to enter those towns every time by hand.

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

```
# Adding towns
add_town Hki Helsinki (3,0) 1000
add_town Tpe Tampere (2,2) 500
add_town Ol Oulu (3,5) 400
add_town Kuo Kuopio (6,3) 300
add_town Tku Turku (1,1) 30
# Adding vassalships
add_vassalship Ol Kuo
```

```
add_vassalship Kuo Hki
add_vassalship Tpe Hki
```

Example run

Below is an example output from the program. The example's commands can be found in the file *example-in.txt* and the output in file *example-out.txt*. You may use this example for testing by running the running the program and giving the command *testread 'example-in.txt' 'example-out.txt'*.

```
> clear_all
Cleared all towns
> town_count
Number of towns: 0
> read "example-data.txt"
** Commands from 'example-data.txt'
> # Adding towns
> add_town Hki Helsinki (3,0) 1000
Helsinki: tax=1000, pos=(3,0), id=Hki
> add_town Tpe Tampere (2,2) 500
Tampere: tax=500, pos=(2,2), id=Tpe
> add_town Ol Oulu (3,5) 400
Oulu: tax=400, pos=(3,5), id=Ol
> add_town Kuo Kuopio (6,3) 300
Kuopio: tax=300, pos=(6,3), id=Kuo
> add_town Tku Turku (1,1) 30
Turku: tax=30, pos=(1,1), id=Tku
> # Adding vassalships
> add_vassalship Ol Kuo
Added vassalship: Oulu -> Kuopio
> add_vassalship Kuo Hki
Added vassalship: Kuopio -> Helsinki
> add_vassalship Tpe Hki
Added vassalship: Tampere -> Helsinki
>
** End of commands from 'example-data.txt'
> town_count
Number of towns: 5
> towns_alphabetically
1. Helsinki: tax=1000, pos=(3,0), id=Hki
2. Kuopio: tax=300, pos=(6,3), id=Kuo
3. Oulu: tax=400, pos=(3,5), id=Ol
4. Tampere: tax=500, pos=(2,2), id=Tpe
5. Turku: tax=30, pos=(1,1), id=Tku
> mindist
Turku: tax=30, pos=(1,1), id=Tku
> maxdist
Kuopio: tax=300, pos=(6,3), id=Kuo
> towns_distance_increasing
1. Turku: tax=30, pos=(1,1), id=Tku
2. Tampere: tax=500, pos=(2,2), id=Tpe
```

```
3. Helsinki: tax=1000, pos=(3,0), id=Hki
4. Oulu: tax=400, pos=(3,5), id=Ol
5. Kuopio: tax=300, pos=(6,3), id=Kuo
> find_towns Kuopio
Kuopio: tax=300, pos=(6,3), id=Kuo
> change_town_name Tpe Manse
Manse: tax=500, pos=(2,2), id=Tpe
> town_vassals Hki
1. Kuopio: tax=300, pos=(6,3), id=Kuo
2. Manse: tax=500, pos=(2,2), id=Tpe
> taxer_path Tpe
1. Manse
2. Helsinki
> towns_nearest (7,5)
1. Kuopio: tax=300, pos=(6,3), id=Kuo
2. Oulu: tax=400, pos=(3,5), id=Ol
3. Manse: tax=500, pos=(2,2), id=Tpe
4. Helsinki: tax=1000, pos=(3,0), id=Hki
5. Turku: tax=30, pos=(1,1), id=Tku
> longest_vassal_path Hki
1. Helsinki
2. Kuopio
3. Oulu
> total_net_tax Tku
Total net tax of Turku: 30
> total_net_tax Ol
Total net tax of Oulu: 360
> total_net_tax Kuo
Total net tax of Kuopio: 306
> total_net_tax Hki
Total net tax of Helsinki: 1084
> remove_town Kuo
Kuopio removed.
> town_vassals Hki
1. Oulu: tax=400, pos=(3,5), id=Ol
2. Manse: tax=500, pos=(2,2), id=Tpe
```