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

Programming assignment 1: Karttaretki (mapping hike)

Last modified on 03/10/2021

Changelog

Below is a list of substantial changes to this document after its initial publication:

- 22.2. Fixed the area ID as an integer, mentioned that an area also contains a coordinate list
- 23.2. Improved the presentation of command parameters in the table
- 24.2. Updated the coordinates in the example run so that there are no equal distances even if distances are rounded to integers
- 25.2. Updated the text so that in all_subareas_in_area command the main program sorts the results in increasing area ID order
- 1.3. Fixed the output of subarea_in_areas in the example run
- 10.3. Added optional parameter ‘silent’ to main program’s ‘read’ command, it discards any printouts from reading the file (handy when reading in large data files)

Contents

Topic of the assignment

Last year has shown a big increase in the popularity of hiking, so new apps are needed to help in that. The first phase of the programming assignment is to create a program into which you can enter information about hiking places (shelters, firepits, etc.) and hiking-related areas (nature reserves, national parks, water areas, etc.). In the second phase the program will be extended to also include hiking ways (paths, tracks, etc.) and hiking route searches. Some operations in the assignment are compulsory, others are not (compulsory = required to pass the assignment, not compulsory = can pass without it, but it is still part of grading).

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).

Terminology

Below is explanation for the most important terms in the assignment:

- **Place.** Every (hiking) place has a unique *integer id, a name* (which consists of characters A-Z, a-z, 0-9, space, and dash -), *a type* (firepit, shelter, parking, peak, bay, area, or other), and *a location (x,y)*, where x and y are integers (the scale and the origin (0,0) of the coordinate system is arbitrary, x coordinates grow to the right, y grows up).
- **Area.** Areas are arbitrary areas on a map, defined by a polygon. Each area has a unique *integer ID,* a *name* (which consists of characters A-Z, a-z, 0-9, space, and dash -), and a list of coordinates that describe the shape of the area. Each area can also contain an arbitrary number of (sub)subareas, and every area can belong to at most one “upper” area. An example of this could be an island that is in a lake that belongs to a national park. *The area relationships cannot form cycles, i.e. area 1 cannot be a subarea of area 2, if area 2 is already a direct or indirect subarea of area 1. The assignment does not have to prepare for attempts to add cyclic areas in any way.*

In the assignment one goal 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 can be justified by performance). In other words, in grading the assignment, asymptotic performance is taken into account, and also the real-life performance (= sensible and efficient implementation aspects). “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 code an as efficient as possible implementation, 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!)

Especially note the following (some of these are new, some are repeated because of their importance):

- 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 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. *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. Into that comment you should write your estimate of the asymptotic performance of the operation, and a short rationale for you 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).**
- Implementing operations `all_subareas_in_area`, `places_closest_to`, `remove_place`, and `common_area_of_subareas` are not compulsory to pass the assignment. **If you only implement the compulsory parts, the maximum grade for the assignment is 3.**
- If the implementation is bad enough, the assignment can be rejected.
- In performance the essential thing is how the execution time changes with more data, not just the measured seconds. 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 doesn't give extra points).

- The performance of an operation includes all work done for the operation, also work possibly done during addition of elements.

On sorting

Sorting names should by done using the default string class “<” comparison, which is ok because names only allow characters a-z, A-Z, 0-9, space, and a dash -. Names with equal names can be in any order with respect to each other.

Operation `places_coord_order`() requires comparison of coordinates. The comparison is based on the “normal” euclidean distance from the origin $\sqrt{x^2+y^2}$ (the coordinate closer to origin comes first). If the distance to origin is the same, the coordinate with the smaller y-coordinate comes first. Coordinates with with equal distances and y-coordinates can be in any order with respect to each other.

In the non-compulsory operation `places_closest_to()` places are ordered based on their distance from a given position. In that case distance is the “normal” euclidean distance $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$, and again if the distances are equal, the coordinate with smaller y comes first.

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 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.). *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)**

- Main routine, 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 places, their areas, 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** (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 `PlaceID`, which used as a unique identifier for each place (and which is used as a return type for many operations). There can be several places with the same name (and even same coordinates), but every place has a different id.
- Type definition `PlaceType`, which lists possible place types.
- Type definition `Coord`, which is used in the public interface to represent (x,y) coordinates. As an example, some comparison operations (==, !=, <) and a hash function have been implemented for this type.
- Type definition `AreaID`, which used as a unique identifier for each area. There can be several areas with the same name, but every area has a different id.
- Type definition `NAME`, which is used for names of places and areas, for example. (The type is a string, and the main routine allows names to contain characters a-z, A-Z, 0-9, space, and dash -).
- Constants `NO_PLACE`, `NO_AREA`, `NO_NAME`, `NO_COORD`, `NO_TYPE`, and `NO_DISTANCE`, which are used as return values, if information is requested for a place or area that doesn't exist.

File *datastructures.cc*

- Here you write the code for the your operations.
- 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). 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) phase 2 controls.

The UI has a command line interface, which accepts commands described later in this document. In addition to this, the UI shows graphically created places and areas (*if you have implemented necessary operations, see below*). The graphical view can be scrolled and zoomed. Clicking on a place name (or area border, which requires precision) 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 code! **It's not a depiction of what the "right" result is, but what information students' code gives out.** The UI uses operation `all_places`() to get a list of places, and asks their information with `get_...`() operations. If drawing areas is on, they are obtained with operation `all_areas`(), and the coordinates of each area with `get_area_coords`().

Parts of the program to be implemented as the assignment

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

- `class Datastructures`: 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 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.pdf mentioned before is written as a part of the assignment.

Note! The code implemented by students does not print out 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.

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

When the program is run, it waits for commands explained below. The commands, whose explanation mentions a member function, call the respective member function of the Datastructure class (implemented by students). 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** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
place_count `int place_count()`	Returns the number of places currently in the data structure.
clear_all `void clear_all()`	Clears out the data structures (after this `all_places()` and `all_areas()` return empty vectors). *This operation is not included in the default performance tests.*
all_places `std::vector<PlaceID> all_places()`	Returns a list (vector) of the places in any (arbitrary) order (the main routine sorts them based on their ID). *This operation is not included in the default performance tests.*
add_place *ID 'Name' type (x,y)* `bool add_place(PlaceID id, Name const& name, PlaceType type, Coord xy)`	Adds a place to the data structure with given unique id, name, type, and coordinates. If there already is a place with the given id, nothing is done and `false` is returned, otherwise `true` is returned.
place_name_type *ID* `std::pair<Name, PlaceType> get_place_name_type(PlaceID id)`	Returns the name and type of the place with given ID, or {NO_NAME, NO_TYPE} if such a place doesn't exist. (Main program calls this in various places.) *This operation is called more often than others, if the parameter of perftest command is "all" or "compulsory".*
place_coord *ID* `Coord get_place_coord(PlaceID id)`	Returns the name of the place with given ID, or value NO_COORD, if such a place doesn't exist. (Main program calls this in various places.) *This operation is called more often than others, if the parameter of perftest command is "all" or "compulsory".*
(The operations below should probably be implemented only after the ones above have been implemented.)	
places_alphabetically `std::vector<PlaceID> places_alphabetically()`	Returns place IDs sorted according to alphabetical order of place names. Places with the same name can be in any order with respect to each other.
places_coord_order `std::vector<PlaceID> places_coord_order()`	Returns place IDs sorted according to their coordinates (defined earlier in this document). Place in the same coordinate can be in any order with respect to each other.

Command **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
`find_places_name name` `std::vector<PlaceID> find_places_name(Name const& name)`	Returns all places with the given name, or an empty vector, if no such places exist. The order of the returned places can be arbitrary (the main routine sorts them based on their ID). *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.*
`find_places_type type` `std::vector<PlaceID> find_places_type(PlaceType type)`	Returns all places with the given type, or an empty vector, if no such places exist. The order of the returned places can be arbitrary (the main routine sorts them based on their ID). *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_place_name ID newname` `bool change_place_name(PlaceID id, Name const& newname)`	Changes the name of the place with given ID. If such place doesn't exist, returns `false`, otherwise `true`.
`change_place_coord ID (x,y)` `bool change_place_coord(PlaceID id, Coord newcoord)`	Changes the location of the place with given ID. If such place doesn't exist, returns `false`, otherwise `true`.
(The operations below should probably be implemented only after the ones above have been implemented.)	
`add_area ID Name Coord1 Coord2...` `bool add_area(AreaID id, Name const& name, std::vector<Coord> coords)`	Adds an area to the data structure with given unique id, name and polygon (coordinates). Initially the added area is not a subarea of any area, and it doesn't contain any subareas. If there already is an area with the given id, nothing is done and `false` is returned, otherwise `true` is returned.
`area_name ID` `Name get_area_name(AreaID id)`	Returns the name of the area with given ID, or NO_NAME if such area doesn't exist. (Main program calls this in various places.) *This operation is called more often than others, if the parameter of perftest command is "all" or "compulsory".*
`area_coords ID` `std::vector<Coord> get_area_coords(AreaID id)`	Returns the coordinate vector of the area with given ID, or a vector with single item NO_COORD, if such area doesn't exist. (Main program calls this in various places.) *This operation is not part of performance tests.*

Command **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
all_areas `std::vector<AreaID> all_areas()`	Returns a list (vector) of the areas in any (arbitrary) order (the main routine sorts them based on their ID). *This operation is not included in the default performance tests.*
add_subarea_to_area ***AreaID AreaID*** `bool add_subarea_to_area(AreaID id, AreaID parentid)`	Adds the first given area as a subarea to the second area. If no areas exist with the given IDs, or if the first area is already a subarea of some area, nothing is done and `false` is returned, otherwise `true` is returned.
subarea_in_areas ***AreaID*** `std::vector<AreaID> subarea_in_areas(AreaID id)`	Returns a list of areas to which the given area belongs either directly or indirectly. The returned vector first contains the area to which the given area belongs directly, then the area that this area belongs to, etc. If no area with given ID exists, a vector with a single element NO_AREA is returned.
(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)	
creation_finished `void creation_finished()`	Performance tests call this operation after all places and areas have been created (after calling new places and areas won't be added). This operation doesn't have to do anything, but you can use it for algorithmic optimizations, if you want. *Note that all operations should work also before calling this operation.*
all_subareas_in_area ***AreaID*** `std::vector<AreaID> all_subareas_in_area(AreaID id)`	Returns a list of areas which belong either directly or indirectly to the given area. The order of areas in the returned vector can be arbitrary (the main program sorts them in increasing ID order). If no area with given ID exists, a vector with a single element NO_AREA is returned.
places_closest_to ***Coord*** **[*type*]** `std::vector<PlaceID> places_closest_to(Coord xy, PlaceType type)`	Returns three places of given type closest to the given coordinate in order of increasing distance (based on the ordering of coordinates described earlier). If no type is given to the command, the main program calls the method with parameter NO_TYPE, in which case three closest places of any type are returned. If there are less than three places in total, of course less places are returned. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

Command **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
`remove_place ID` `bool remove_place(PlaceID id)`	Removes a place with the given id. If a place with given id does not exist, does nothing and returns `false`, otherwise returns `true`. *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).*
`common_area_of_subareas AreaID AreaID` `AreaID common_area_of_subareas(AreaID id1, AreaID id2)`	Returns the "nearest" common area in the subarea hierarchy for the areas. That is, an area to which both areas belong either directly or indirectly (but both areas together don't belong to any of the returned area's subareas). If either of the area ids do not correspond to any area, or if no common area exists, returns NO_AREA.
(The following operations are already implemented by the main program.)	
random_add *n* (implemented by main program)	Add *n* new places and areas with random id, name, type, and coordinates (for testing). The added areas are added to random areas and subareas. 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 optional parameter 'silent' is given, outputs of the commands are not displayed. (This can be used to read a list of places 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** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
perftest all|compulsory|*cmd1*[;*cmd2*...] *timeout repeat n1*[;*n2*...] (implemented by main program)	Run performance tests. Clears out the data structure and add *n1* random places and areas (see random_add). Then a random command is performed *repeat* 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 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, "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 "*in-filename*" "*out-filename*" (implemented by main program)	Runs a correctness test and compares results. Reads command from file in-filename and shows the output of the commands next to the expected output in file out-filename. 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 can add a bunch of places and areas. Those files can then be read in using the "read" command, after which other commands can be tested without having to enter those places every time by hand.

Below are examples of a data files, one of which adds places, the other areas:

- *example-places.txt*

```
# Places
```

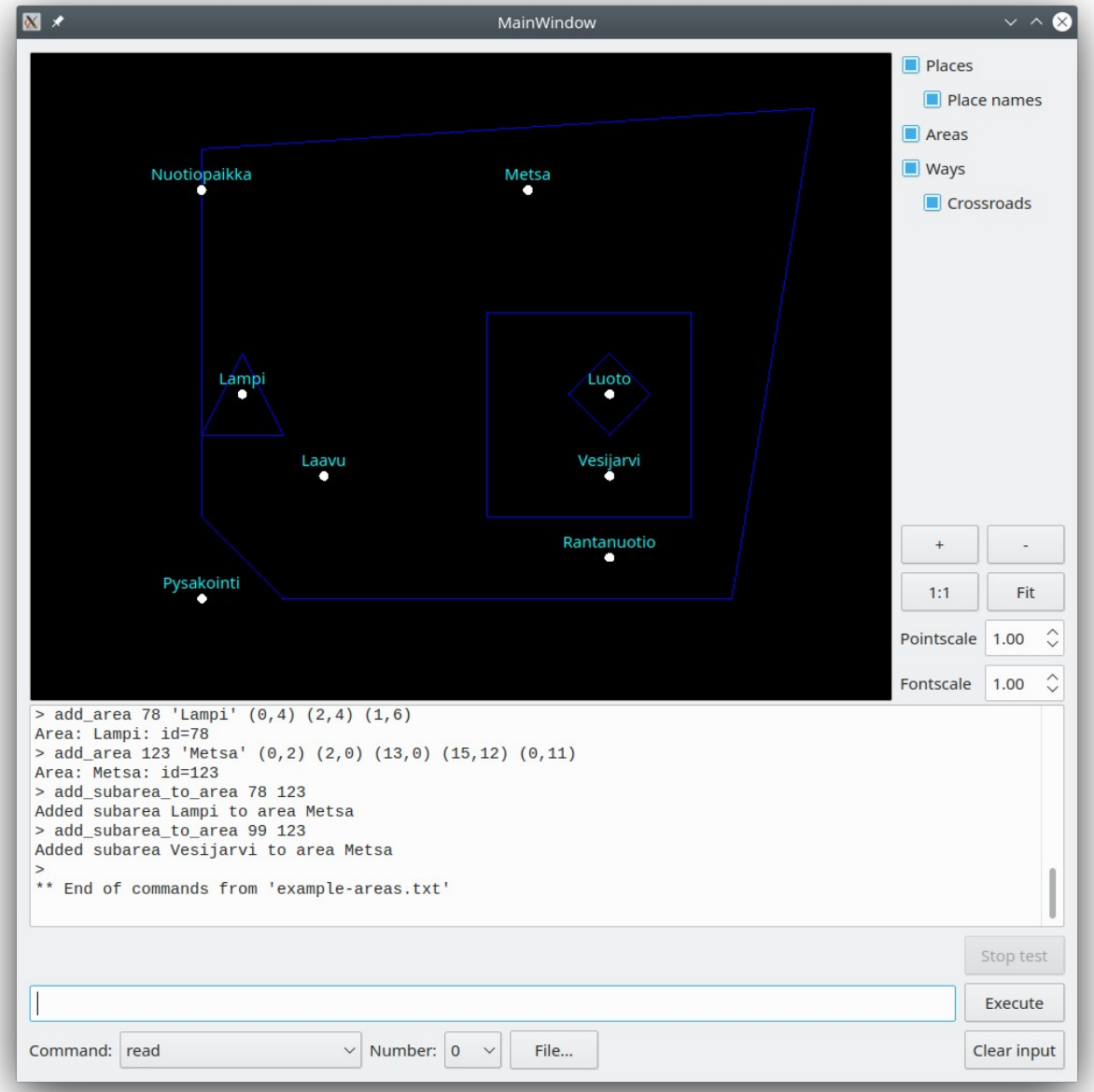
```
add_place 10 'Laavu' shelter (3,3)
add_place 15 'Pysakointi' parking (0,0)
add_place 4 'Nuotiopaikka' firepit (0,7)
add_place 20 'Rantanuotio' firepit (11,1)
add_place 99 'Vesijarvi' area (10,3)
add_place 98 'Luoto' area (10,5)
add_place 78 'Lampi' area (1,5)
add_place 123 'Metsa' area (7,10)
```

- *example-areas.txt*

```
# Areas
add_area 99 'Vesijarvi' (7,2) (12,2) (12,7) (7,7)
add_area 98 'Luoto' (10,4) (11,5) (10,6) (9,5)
add_subarea_to_area 98 99
add_area 78 'Lampi' (0,4) (2,4) (1,6)
add_area 123 'Metsa' (0,2) (2,0) (13,0) (15,12) (0,11)
add_subarea_to_area 78 123
add_subarea_to_area 99 123
```

Screenshot of user interface

Below is a screenshot of the graphical user interface after *example-places.txt* and *example-areas.txt* have been read in.



Example run

Below are example outputs from the program. The example's commands can be found in files *example-compulsory-in.txt* and *example-all-in.txt*, and the outputs in files *example-compulsory-out.txt* and *example-all-out.txt*. I.e., you can use the example as a small test of compulsory behaviour by running command
`testread "example-compulsory-in.txt" "example-compulsory-out.txt"`

```
> clear_all
Cleared everything.
> place_count
Number of places: 0
> read "example-places.txt"
** Commands from 'example-places.txt'
> # Places
> add_place 10 'Laavu' shelter (3,3)
Laavu (shelter): pos=(3,3), id=10
> add_place 15 'Pysakointi' parking (0,0)
Pysakointi (parking): pos=(0,0), id=15
> add_place 4 'Nuotiopaikka' firepit (0,7)
Nuotiopaikka (firepit): pos=(0,7), id=4
> add_place 20 'Rantanuotio' firepit (11,1)
Rantanuotio (firepit): pos=(11,1), id=20
> add_place 99 'Vesijarvi' area (10,3)
Vesijarvi (area): pos=(10,3), id=99
> add_place 98 'Luoto' area (10,5)
Luoto (area): pos=(10,5), id=98
> add_place 78 'Lampi' area (1,5)
Lampi (area): pos=(1,5), id=78
> add_place 123 'Metsa' area (7,10)
Metsa (area): pos=(7,10), id=123
>
** End of commands from 'example-places.txt'
> place_count
Number of places: 8
> place_name_type 10
Place ID 10 has name 'Laavu' and type 'shelter'
Laavu (shelter): pos=(3,3), id=10
> place_coord 4
Place ID 4 is in position (0,7)
Nuotiopaikka (firepit): pos=(0,7), id=4
> places_alphabetically
1. Laavu (shelter): pos=(3,3), id=10
2. Lampi (area): pos=(1,5), id=78
3. Luoto (area): pos=(10,5), id=98
4. Metsa (area): pos=(7,10), id=123
5. Nuotiopaikka (firepit): pos=(0,7), id=4
6. Pysakointi (parking): pos=(0,0), id=15
7. Rantanuotio (firepit): pos=(11,1), id=20
8. Vesijarvi (area): pos=(10,3), id=99
> places_coord_order
1. Pysakointi (parking): pos=(0,0), id=15
2. Laavu (shelter): pos=(3,3), id=10
3. Lampi (area): pos=(1,5), id=78
4. Nuotiopaikka (firepit): pos=(0,7), id=4
5. Vesijarvi (area): pos=(10,3), id=99
6. Rantanuotio (firepit): pos=(11,1), id=20
7. Luoto (area): pos=(10,5), id=98
8. Metsa (area): pos=(7,10), id=123
> find_places_type firepit
1. Nuotiopaikka (firepit): pos=(0,7), id=4
2. Rantanuotio (firepit): pos=(11,1), id=20
```

```
> change_place_name 20 'Nuotiopaikka'
Nuotiopaikka (firepit): pos=(11,1), id=20
> find_places_name 'Nuotiopaikka'
1. Nuotiopaikka (firepit): pos=(0,7), id=4
2. Nuotiopaikka (firepit): pos=(11,1), id=20
> read "example-areas.txt"
** Commands from 'example-areas.txt'
> # Areas
> add_area 99 'Vesijarvi' (7,2) (12,2) (12,7) (7,7)
Area: Vesijarvi: id=99
> add_area 98 'Luoto' (10,4) (11,5) (10,6) (9,5)
Area: Luoto: id=98
> add_subarea_to_area 98 99
Added subarea Luoto to area Vesijarvi
> add_area 78 'Lampi' (0,4) (2,4) (1,6)
Area: Lampi: id=78
> add_area 123 'Metsa' (0,2) (2,0) (13,0) (15,12) (0,11)
Area: Metsa: id=123
> add_subarea_to_area 78 123
Added subarea Lampi to area Metsa
> add_subarea_to_area 99 123
Added subarea Vesijarvi to area Metsa
>
** End of commands from 'example-areas.txt'
> all_areas
1. Lampi: id=78
2. Luoto: id=98
3. Vesijarvi: id=99
4. Metsa: id=123
> area_name 99
Area ID 99 has name 'Vesijarvi'
Vesijarvi: id=99
> subarea_in_areas 98
Area hierarchy for area Luoto: id=98
1. Vesijarvi: id=99
2. Metsa: id=123

… (the rest of the lines are from example-all-out.txt)
> all_subareas_in_area 123
All subareas of Metsa: id=123
1. Lampi: id=78
2. Luoto: id=98
3. Vesijarvi: id=99
> places_closest_to (0,0) firepit
1. Nuotiopaikka (firepit): pos=(0,7), id=4
2. Nuotiopaikka (firepit): pos=(11,1), id=20
> places_closest_to (10,0)
1. Nuotiopaikka (firepit): pos=(11,1), id=20
2. Vesijarvi (area): pos=(10,3), id=99
3. Luoto (area): pos=(10,5), id=98
> add_area 999 'x' (8,3) (9,3) (8,4)
Area: x: id=999
> add_subarea_to_area 999 99
Added subarea x to area Vesijarvi
```

```
> common_area_of_subareas 999 98
Common area of areas x: id=999 and Luoto: id=98 is:
Vesijarvi: id=99
> common_area_of_subareas 98 78
Common area of areas Luoto: id=98 and Lampi: id=78 is:
Metsa: id=123
> remove_place 20
Place Nuotiopaikka(firepit) removed.
> find_places_name 'Nuotiopaikka'
Nuotiopaikka (firepit): pos=(0,7), id=4
> find_places_type firepit
Nuotiopaikka (firepit): pos=(0,7), id=4
> places_alphabetically
1. Laavu (shelter): pos=(3,3), id=10
2. Lampi (area): pos=(1,5), id=78
3. Luoto (area): pos=(10,5), id=98
4. Metsa (area): pos=(7,10), id=123
5. Nuotiopaikka (firepit): pos=(0,7), id=4
6. Pysakointi (parking): pos=(0,0), id=15
7. Vesijarvi (area): pos=(10,3), id=99
> places_coord_order
1. Pysakointi (parking): pos=(0,0), id=15
2. Laavu (shelter): pos=(3,3), id=10
3. Lampi (area): pos=(1,5), id=78
4. Nuotiopaikka (firepit): pos=(0,7), id=4
5. Vesijarvi (area): pos=(10,3), id=99
6. Luoto (area): pos=(10,5), id=98
7. Metsa (area): pos=(7,10), id=123
> places_closest_to (10,0)
1. Vesijarvi (area): pos=(10,3), id=99
2. Luoto (area): pos=(10,5), id=98
3. Laavu (shelter): pos=(3,3), id=10
```