COMP.CS.300 Data structures and algorithms 1, autumn 2022

Programming assignment 1: Railroads

Last modified on 11/24/2022

Change log

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

- 11th November: removed the separate readme-document, it is no longer required.

Contents

Introduction to the assignment

In this programming assignment you will practise implementing simple algorithms and evaluating their performance. This autumn's programming assignment is about railroads, in other words railway stations and trains running between them. The goal of the first programming assignment is to create a program into which you can enter information about railway stations, train departures, and administrative regions surrounding the stations (municipalities, provinces, states, etc.). In the second assignment the program will be extended to also include train connections between stations and doing route searches. Some operations in the assignment are compulsory, others are not. A compulsory operation is required to be implemented to pass the assignment, the non-compulsory parts are not required to pass the assignment, but they are 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).

One goal in the assignment is to practise how to efficiently use ready-made data structures and algorithms (STL). Another goal is to write 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 grading the assignment, both asymptotic and real-life performance (= sensible and efficient implementation aspects) are taken into account. “Micro optimizations” (like ”do I write `a = a+b` or `a += b`”, or ”how do I tweak compiler’s optimization flags”) will not improve the grade.

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 particular, 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:** 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. *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, you should write your estimate of the asymptotic performance and a short rationale for each operation you implement. The file datastructures.hh contains a code comment for this before each operation.**
- Implementing operations `all_subregions_in_region`, `stations_closest_to`, `remove_station`, and `common_parent_of_regions` 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 judging runtime 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.
- More points will also be given for better real performance in seconds (if the required minimum asymptotic performance requirements are 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 the addition of elements.

Terminology

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

- **Station.** Every station has a unique *string id, a name,* and *a location (x,y)*, where x and y are integers (the scale and the origin (0,0) of the coordinate system are arbitrary, x coordinates grow to the right, y grows up). You can assume that two stations cannot have the same coordinate.
- **Region. Regions** are arbitrary (administrative) regions on a map, defined by a polygon. Each region has a unique *integer ID,* a *name,* and a list of coordinates that describe the shape of the region. Each region can contain stations, and also an arbitrary number of (sub)regions, so that every region can belong to at most one "upper" region. An example of this could be a town that contains its stations, and is part of a province, which in turn is part of a state. The relationships between regions and stations are given to the program, i.e. the program doesn't have to deduce them by analyzing regions' coordinates. *The region relationships cannot form cycles, i.e. region 1 cannot be a subregion of region 2, if region 2 is already a direct or indirect subregion of region 1. The assignment does not have to prepare for attempts to add cyclic regions in any way.*

On sorting

Sorting names should by done using the default string class "<" comparison, which works because only characters a-z, A-Z, 0-9, space, and a dash - are allowed in names. Multiple equal names can be in any order with respect to each other.

The operation `stations_distance_increasing`() requires the 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 the origin is the same, the coordinate with the smaller y-value comes first. Coordinates with with equal distances and y-values can be in any order with respect to each other.

In the non-compulsory operation `stations_closest_to()` stations are ordered based on their distance from a given position. For this operation the 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 highly recommended and part of the grading. There are no restrictions in using the C++ standard library, including STL. Using libraries not part of the language is not allowed (for example libraries provided by Windows, Qt libraries, Boost, 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.

Some parts of the program are provided by the course, some parts have to be implemented by the student.

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 files contain the main program, 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 also 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 stations, their regions, 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 `StationID:` Used as a unique identifier for each station (consists of characters A-Z, a-z, 0-9, and dash -). There can be several stations with the same name, but every station has a different id.

- Type definition `Coord:` 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 `TrainID:` Used as a unique identifier for each train (consists of characters A-Z, a-z, 0-9, and dash -).
- Type definition `RegionID:` Used as a unique identifier for each region (it is a non-negative integer). There can be several regions with the same name, but every region has a different id.
- Type definition `NAME:` Used for names of stations and regions (consists of characters A-Z, a-z, 0-9, space, and dash -).
- Type definition `Time:` An integer, which represents a time of day in format HHMM (in 24 hour notation). The main program ensures that all entered times are valid. Note, that the format of the times makes it possible to simply use the < operator to compare two times. In the assignment you don't have to handle the changing of the day after midnight.
- Constants `NO_STATION`, `NO_REGION`, `NO_TRAIN`, `NO_NAME`, `NO_COORD`, `NO_TYPE`, and `NO_DISTANCE`: Used as error codes, if information is requested for a station or region that doesn't exist.

File datastructures.cc

- Here you write the code for the your operations.
- Function `random_in_range`: 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 assignment 1 the UI has some disabled (greyed out) controls needed for assignment 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 stations and regions (*if you have implemented necessary operations, see below*). The graphical view can be scrolled and zoomed. Clicking on a station name (or region 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 the student's code! **It's not a depiction of what the "right" result is, but what information students' code gives out.** The UI uses the operation `all_stations`() to get a list of stations, and asks their information with `get_...`() operations. If the checkbox for drawing regions is on, regions are obtained with the operation `all_regions`(), and the coordinates of each region with `get_region_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.**

Note! The code implemented by the student 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 awaits for commands explained below. The commands, whose explanation mentions a member function, call the respective member function of the Datastructure class (which the student must implement). 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. A program compiled with QtCreator can also be started from the command prompt in a purely textual form with command line argument --console.

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** (Optional parameters of commands are in []-brackets and alternatives separated by |)	**Explanation**

`station_count`
`int station_count()`
	Returns the number of stations currently in the data structure.

Command **Public member function** (Optional parameters of commands are in []-brackets and alternatives separated by |)	**Explanation**

`clear_all`
`void clear_all()`
	Clears out the data structures (after this `all_stations`() and `all_regions()` return empty vectors). *This operation is not included in the default performance tests.*

`all_stations`
`std::vector<StationID> all_stations()`
	Returns all the stations in any (arbitrary) order (the main routine sorts them based on their ID).

`add_station` *`ID "Name" (x,y)`*
`bool add_station(StationID id, Name const& name, Coord xy)`
	Adds a station to the data structure with the given unique id, name, type, and coordinates. If there already is a station with the given id, nothing is done and `false` is returned, otherwise `true` is returned.

`station_info` *`ID`*
`Name get_station_name(StationID id)`
	Returns the name and type of the station with the given ID, or NO_NAME if such a station doesn't exist. (The main program calls this in various stations.) *This operation is called more often than others (it is also called by the perftest command with parameter "all" or "compulsory").*

`station_info` *`ID`*
`Coord get_station_coord(StationID id)`
	Returns the name of the station with the given ID, or value NO_COORD, if such a station doesn't exist. (The main program calls this in various stations.) *This operation is called more often than others (it is also called by the perftest command with parameter "all" or "compulsory").*

(The operations below should probably be implemented only after the ones above have been implemented.)	

`stations_alphabetically`
`std::vector<StationID> stations_alphabetically()`
	Returns station IDs sorted according to alphabetical order of station names. Stations with the same name can be in any order with respect to each other.

`stations_distance_increasing`
`std::vector<StationID> stations_distance_increasing()`
	Returns station IDs sorted according to their coordinates (defined earlier in this document).

`find_station_with_coord` *`(x,y)`*
`StationID find_station_with_coord(Coord xy)`
	Returns a station with the given coordinate, or NO_STATION, if no such station exists.

`change_station_coord` *`ID (x,y)`*
`bool change_station_coord(StationID id, Coord newcoord)`
	Changes the location of the station with given ID. If such station doesn't exist, returns `false`, otherwise `true`.

Command **Public member function** (Optional parameters of commands are in []-brackets and alternatives separated by |)	**Explanation**

`add_departure StationID TrainID Time`
`bool add_departure(StationID stationid, TrainID trainid, Time time)`
	Adds information that the given train leaves from the given station at the given time. If such station doesn't exist or the departure has already been added (train already leaves from given station at the given time), returns `false`, otherwise `true`.

`remove_departure StationID TrainID Time`
`bool remove_departure(StationID stationid, TrainID trainid, Time time)`
	Removes the given train departure from the given station at the given time. If such a station or such departure from the station doesn't exist, returns `false`, otherwise `true`.

`station_departures_after StationID Time`
`std::vector<std::pair<Time, TrainID>> station_departures_after(StationID stationid, Time time)`
	Lists all train departures from the given station at or after the given time. Departures should be sorted based on departure time. If trains have the same departure time, they should be sorted based on their ids. If the given station doesn't exist, pair {NO_TIME, NO_TRAIN} is returned.

(The operations below should probably be implemented only after the ones above have been implemented.)	

`add_region` *`ID Name Coord1 Coord2...`*
`bool add_region(RegionID id, Name const& name, std::vector<Coord> coords)`
	Adds a region to the data structure with given unique id, name and polygon (coordinates). Initially the added region is not a subregion of any region, and it doesn't contain any subregions or stations. If there already is a region with the given id, nothing is done and `false` is returned, otherwise `true` is returned.

`all_regions`
`std::vector<RegionID> all_regions()`
	Returns all the regions in any (arbitrary) order (the main routine sorts them based on their ID). *This operation is not included in the default performance tests.*

`region_info` *`ID`*
`Name get_region_name(RegionID id)`
	Returns the name of the region with the given ID, or NO_NAME if such region doesn't exist. (Main program calls this in various stations.) *This operation is called more often than others (it is also called by the perftest command with parameter "all" or "compulsory").*

`region_info` *`ID`*
`std::vector<Coord> get_region_coords(RegionID id)`
	Returns the coordinate vector of the region with the given ID, or a vector with single item NO_COORD, if such region doesn't exist. (The main program calls this in various stations.) *This operation is not part of performance tests.*

Command **Public member function** (Optional parameters of commands are in []-brackets and alternatives separated by |)	**Explanation**

`add_subregion_to_region` *`RegionID`* *`RegionID`*
 `bool add_subregion_to_region(RegionID id, RegionID parentid)`
	Adds the first given region as a subregion to the second region. If no regions exist with the given IDs, or if the first region is already a subregion of some region, nothing is done and `false` is returned, otherwise `true` is returned.

`add_station_to_region StationID` *`RegionID`*
 `bool add_station_to_region(StationID id, RegionID parentid)`
	Adds the given station to the given region. If no station or region exist with the given IDs, or if the station already belongs to some region, nothing is done and `false` is returned, otherwise `true` is returned.

`station_in_regions` *`StationID`*
 `std::vector<RegionID> station_in_regions(StationID id)`
	Returns a list of regions to which the given station belongs either directly or indirectly. The returned vector first contains the region to which the given station belongs directly, then the region that this region belongs to, etc. If no station with the given ID exists, a vector with a single element NO_REGION is returned.

(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)	

`all_subregions_of_region` *`RegionID`*
 `std::vector<RegionID> all_subregions_of_region(RegionID id)`
	Returns a list of regions which belong either directly or indirectly to the given region. The order of regions in the returned vector can be arbitrary (the main program sorts them in increasing ID order). If no region with the given ID exists, a vector with a single element NO_REGION is returned.

`stations_closest_to` *`Coord`*
 `std::vector<StationID> stations_closest_to(Coord xy)`
	Returns the three stations closest to the given coordinate in order of increasing distance (based on the ordering of coordinates described earlier). If there are less than three stations in total, of course less stations are returned. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

`remove_station` *`ID`*
 `bool remove_station(StationID id)`
	Removes the station with the given id. If a station with the given id does not exist, does nothing and returns `false`, otherwise returns `true`. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

Command **Public member function** (Optional parameters of commands are in []-brackets and alternatives separated by |)	**Explanation**

`common_parent_of_regions RegionID RegionID`
`RegionID common_parent_of_regions(RegionID id1, RegionID id2)`
	Returns the “nearest” region in the subregion hierarchy, to which both given regions belong either directly or indirectly. The returned region must be the "nearest" in the following sense: it does not have a subregion containing both given regions.
	If either of the region ids do not correspond to any region, or if no common region exists, returns NO_REGION.

(The following operations are already implemented by the main program.)	
random_add *n* (implemented by main program)	Add *n* new stations and *n/10* regions (for testing) with random ids, names, and coordinates. The added regions are added to random regions as subregions, and stations to regions with 50 % probability. Note! The values really are random, so they can be different for each run. Also all the coordinates are random, so stations and subregions are probably outside their parent region etc.
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 and executes 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 stations from a file, run tests, etc.)
stopwatch on|off|next (implemented by main program)	Switch time measurement on or off. When the program starts, the stopwatch is "off". When it is turned "on", the time it takes to execute each command is printed after the command. The 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** (Optional parameters of commands are in []-brackets and alternatives separated by |)	**Explanation**
perftest all|compulsory|*cmd1*[;*cmd2*...] *timeout repeat n1*[;*n2*...] (implemented by main program)	Runs performance tests. First the command clears out the data structure and adds *n1* random stations and regions (see random_add). After this 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 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 from the list of 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, not just in the beginning). If the program is run from the 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 "*in-filename*" "*out-filename*" (implemented by main program)	Runs a correctness test and compares the results. This command 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 stations, trains and regions. Those files can then be read in using the "read" command, after which other commands can be tested without having to enter the data every time by hand.

What follows are examples of such data files:

example-stations.txt

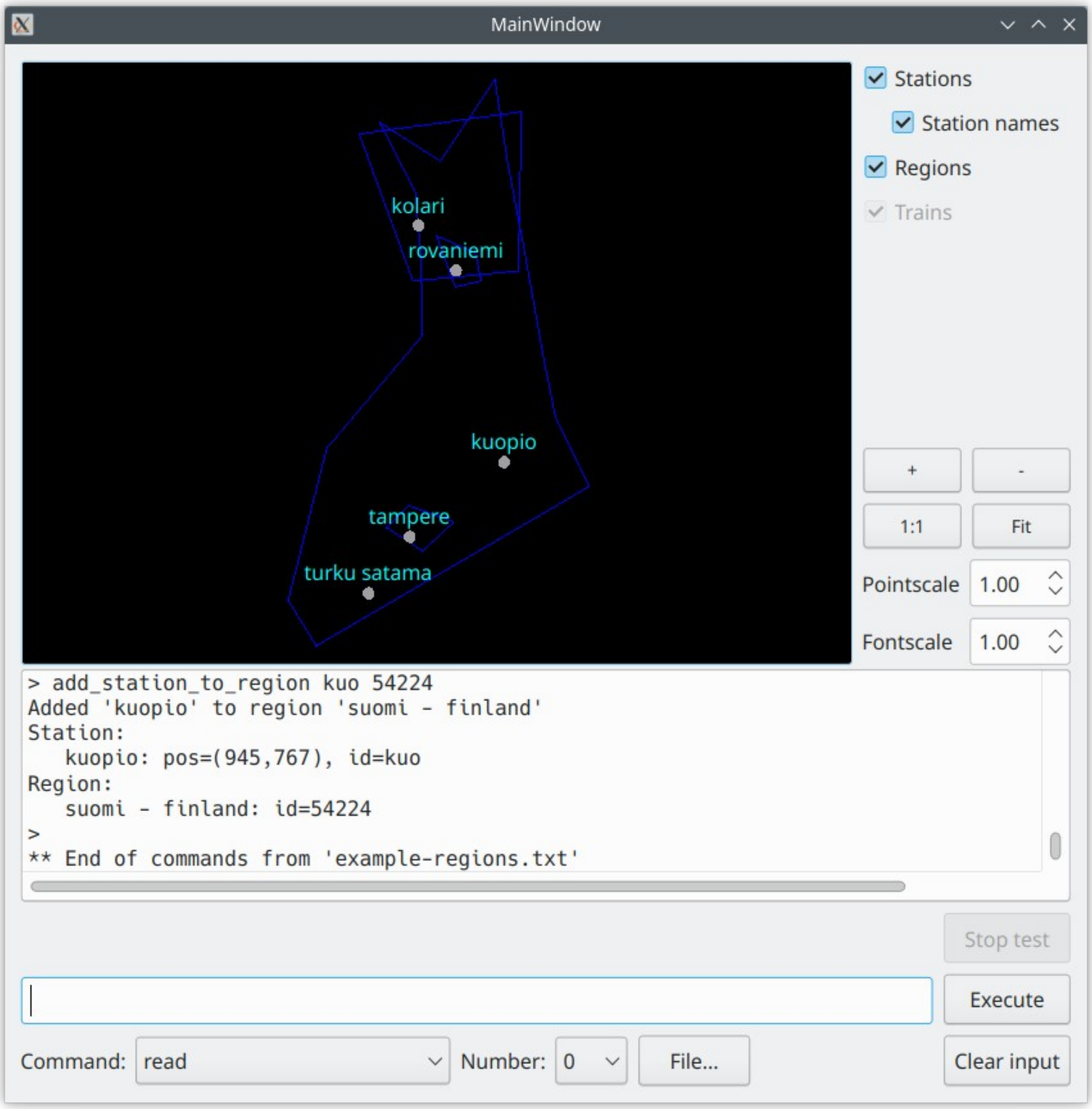
```
add_station kuo "kuopio" (945,767)
add_station tpe "tampere" (542,455)
add_station kli "kolari" (579,1758)
add_station tus "turku satama" (366,219)
add_station roi "rovaniemi" (740,1569)
```

example-regions.txt

```
add_region 6440429 "tampereen seutukunta"  (442,495) (535,586) (729,518)
(597,396) (442,495)
add_station_to_region tpe 6440429
add_region 2528474 "rovaniemi"  (656,1714) (737,1500) (848,1525)
(823,1641) (656,1714)
add_region 1724359 "lappi"  (327,2139) (1020,2232) (1006,1566) (556,1525)
(327,2139)
add_station_to_region roi 2528474
add_subregion_to_region 2528474 1724359
add_station_to_region kli 1724359
add_region 54224 "suomi - finland"  (23,189) (188,827) (598,1300)
(590,1641) (567,1894) (415,2187) (672,2026) (906,2369) (960,2023)
(1030,1664) (1111,1219) (1164,958) (1308,666) (143,0) (23,189)
add_subregion_to_region 6440429 54224
add_subregion_to_region 1724359 54224
add_station_to_region kuo 54224
```

Screenshot of user interface

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



```
> add_station_to_region kuo 54224
Added 'kuopio' to region 'suomi - finland'
Station:
   kuopio: pos=(945,767), id=kuo
Region:
   suomi - finland: id=54224
>
** End of commands from 'example-regions.txt'
```

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.* If you wish to perform a small test on all the compulsory commands, you can do this by running the following:

```
testread "example-compulsory-in.txt" "example-compulsory-out.txt"
```

example-compulsory

```
> clear_all
Cleared all stations
> # Stations
> station_count
Number of stations: 0
> read "example-stations.txt" silent
** Commands from 'example-stations.txt'
...(output discarded in silent mode)...
** End of commands from 'example-stations.txt'
> station_count
Number of stations: 5
> station_info tpe
Station:
   tampere: pos=(542,455), id=tpe
> station_info kli
Station:
   kolari: pos=(579,1758), id=kli
> stations_alphabetically
Stations:
1. kolari: pos=(579,1758), id=kli
2. kuopio: pos=(945,767), id=kuo
3. rovaniemi: pos=(740,1569), id=roi
4. tampere: pos=(542,455), id=tpe
5. turku satama: pos=(366,219), id=tus
> stations_distance_increasing
Stations:
1. turku satama: pos=(366,219), id=tus
2. tampere: pos=(542,455), id=tpe
3. kuopio: pos=(945,767), id=kuo
4. rovaniemi: pos=(740,1569), id=roi
5. kolari: pos=(579,1758), id=kli
> change_station_coord tpe (600,500)
Station:
   tampere: pos=(600,500), id=tpe
> find_station_with_coord (600,500)
Station:
   tampere: pos=(600,500), id=tpe
> add_departure tpe ic20 1000
Train ic20 leaves from station tampere (tpe) at 1000
> add_departure tpe ic22 1200
Train ic22 leaves from station tampere (tpe) at 1200
> add_departure kli pyo276 1942
Train pyo276 leaves from station kolari (kli) at 1942
> station_departures_after tpe 1100
Departures from station tampere (tpe) after 1100:
 ic22 at 1200
> station_departures_after kli 1900
Departures from station kolari (kli) after 1900:
 pyo276 at 1942
> remove_departure kli pyo276 1942
Removed departure of train pyo276 from station kolari (kli) at 1942
> station_departures_after kli 1900
```

```
No departures from station kolari (kli) after 1900
> # Regions
> read "example-regions.txt" silent
** Commands from 'example-regions.txt'
...(output discarded in silent mode)...
** End of commands from 'example-regions.txt'
> all_regions
Regions:
1. suomi - finland: id=54224
2. lappi: id=1724359
3. rovaniemi: id=2528474
4. tampereen seutukunta: id=6440429
> region_info 6440429
Region:
   tampereen seutukunta: id=6440429
> station_in_regions kuo
Station:
   kuopio: pos=(945,767), id=kuo
Region:
   suomi - finland: id=54224
> station_in_regions kli
Station:
   kolari: pos=(579,1758), id=kli
Regions:
1. lappi: id=1724359
2. suomi - finland: id=54224
```

example-all

```
> # First read in compulsory example
> read "example-compulsory-in.txt"
** Commands from 'example-compulsory-in.txt'
...
** End of commands from 'example-compulsory-in.txt'
> all_subregions_of_region 54224
Regions:
1. suomi - finland: id=54224
2. lappi: id=1724359
3. rovaniemi: id=2528474
4. tampereen seutukunta: id=6440429
> stations_closest_to (500,400)
Stations:
1. tampere: pos=(600,500), id=tpe
2. turku satama: pos=(366,219), id=tus
3. kuopio: pos=(945,767), id=kuo
> common_parent_of_regions 2528474 6440429
Regions:
1. rovaniemi: id=2528474
2. tampereen seutukunta: id=6440429
3. suomi - finland: id=54224
> remove_station tpe
tampere removed.
> stations_alphabetically
Stations:
```

```
1. kolari: pos=(579,1758), id=kli
2. kuopio: pos=(945,767), id=kuo
3. rovaniemi: pos=(740,1569), id=roi
4. turku satama: pos=(366,219), id=tus
> stations_distance_increasing
Stations:
1. turku satama: pos=(366,219), id=tus
2. kuopio: pos=(945,767), id=kuo
3. rovaniemi: pos=(740,1569), id=roi
4. kolari: pos=(579,1758), id=kli
> find_station_with_coord (600,600)
Failed (NO_STATION returned)!
> stations_closest_to (500,400)
Stations:
1. turku satama: pos=(366,219), id=tus
2. kuopio: pos=(945,767), id=kuo
3. rovaniemi: pos=(740,1569), id=roi
```