Tramway

Goal: I will practice using containers and iterators of STL as well as file management. I will also consider how to divide program into parts by using functions.

I will especially practice choosing the correct data structure among the different data structures of STL.

Instructions: Retrieve the code template: templates/07/tramway2/ -> student/07/tramway2/. However, the code template only gives you the picture:

```
=====//==================//===

  __<<_____________<<__

 | ____ ____ ____ ____ ____ |

 ||  ||  ||  ||  ||  ||

 |_|__|_|__|_|__|_|__|_|__|_|

.|               RASSE   |.

:|_____________________________|:

___(o)(o)___(o)(o)___(o)(o)____

------------------------------
```

Implement a program that first reads information concerning tramways from the input file, stores them into a suitable data structure, and then permits the user to make searches, insertions, and removals in the data structure in question.

Important

Before starting to write code, read carefully the whole assignment. Especially, note the required commits.

Note

This project can be done either in pairs or independently. If you do the project in pairs,

you need to form a group. If you have not created a group yet or you wish to change your working partner, choose ”Form a group” from the menu on the left to register a new group. In addition, the submission box at the very end of the current page shows two choices: “Submit alone” / “Submit with…”. You should be careful when selecting the choice, because you cannot change it afterwards. Enter only the address of the Git repository owned by either of you (not both) in the submit box. The code files must contain (in comments) the personal data of both of you.

You can look for a group via Kooditorio’s Discord link (see Timetable & links).

Header comment and feedback language

In the same way as in the first project, the header comment is again required. This time we provide no ready-made header comment, but you should write it by yourself. You can use the first project’s assignment as an example (see 4.6 (P) Pairs).

The feedback language will be chosen in the submission box (at the very end of this page). **By default, the feedback language is Finnish**, but you can change it, if you want to have the feedback, given by an assistant, in English. We will use the language you have given in the submission box of the final submission.

If you are working in pairs, add personal data of both of you.

Data processed by the program

The input file given to the program contains lines, each of which has (at most) three data fields: a line, a stop, and the distance of the stop from the departure stop. These fields are separated by semicolons from each other. The structure of an input file accepted by the program can, for example, be as follows:

```
West;Pyynikintori;0
West;Tuulensuu;0.5
West;Keskustori;1.5
West;Koskipuisto;1.8
West;Rautatieasema;2.2
West;Tulli;2.5
```

```
West;Sammonaukio;2.8

East;Sammonaukio;

East;Kaleva;0.2

East;Uintikeskus;0.3

East;Kalevanrinne;0.6

East;Hakametsa;1

East;Turtola;3

East;Hallila;5

East;Hervanta;7

East;Hervannan kampus;7.1

South;Hervannan kampus

South;Etela-Hervanta;0.4

South;Hervantajarvi;0.7

Hospital;Sammonaukio

Hospital;Kalevan kirkko;0.1

Hospital;Hippos;0.4

Hospital;TAYS;0.6

Hospital;Kaupin kampus;0.7
```

As can be seen above, the departure stop can be given in three ways:

```
West;Pyynikintori;0

East;Sammonaukio;

South;Hervannan kampus
```

Of course, the distance from the departure stop from itself is zero. This can be given as a number, as an empty field, or as a missing field. (In program code, you had better store zero in stop data in every case.)

Here, all the stops of a line have been listed one after the other, but the lines could have been listed mixed with other. However, the input file contains the stops of each line are in

the same order than they really exist in the line. Therefore, you can assume that the distances of the stops of a line are listed in ascending order.

The names of both lines and stops may consist of several words. If this is the case, queries require quote marks, as will be explained in later examples.

Error messages related to file management

When the program starts up, it prints tramway and reads the name of the input file from the user as follows:

```
=====//====================//===
   __<<_____________<<__
  | ____ ____ ____ ____ ____ |
  | |  | |  | |  | |  | |  | |
  |_|__|_|__|_|__|_|__|_|__|_|
.|                RASSE   |.
:|________________________|:
___(o)(o)___(o)(o)___(o)(o)____
--------------------------------
Give a name for input file: data.txt
```

If the program cannot open the input file, it will print the text:

```
Error: File could not be read.
```

After that, the program terminates with no other prints with the return value EXIT_FAILURE.

If the input file can be opened but the file does not correspond to the specification above, the program prints the error message:

```
Error: Invalid format in file.
```

and terminates with the return value EXIT_FAILURE. This can be the case, if, for example, a line of the input file contains something else than two or three text pieces separated by

a semicolon. Empty line or stop names are not accepted either.

The program checks that the input file does include duplicates, i.e. a stop and a distance can be in a line only once. For example, if the input file had lines:

```
West;Koskipuisto;1.8

West;Koskipuisto;1.9
```

or:

```
West;Koskipuisto;1.8

West;Rosso;1.8
```

the program would print in either case the error message:

```
Error: Stop/line already exists.
```

and terminates with the return value EXIT_FAILURE.

Even if the input file had a line with both the same stop and the same distance several times, the above error message is given only once.

However, a stop can belong to several lines. For example, in the input file shown above, the stop Sammonaukio belongs to lines West, East, and Hospital.

There are more error messages related to the commands. They will be explained in the context of the command in question.

Commands and error messages in general

The next subsection describes the commands to be implemented. Each command has 0-3 parameters. If too few parameters are given for a command, the program gives the error message:

```
Error: Invalid input.
```

If there are too many parameters, the extra ones are ignored. No error message is given, the extra ones are just useless.

Parameters are typically lines and/or stops. If a command is used to print something about them, the program ensures that the line/stop in question can be found in the data structure. On the other hand, if a command concerns adding a line or stop, the program ensures that the line/stop in question does not already exist.

The commands available in the program with their error messages

The program parses the lines of the input file and stores the information about lines and stops in an adequate internal data structure.

After reading the input file, the program starts the actual execution. Each time the user is expected to give input, the program prints the prompt:

```
tramway>
```

To this prompt, the user can give the following commands:

- QUIT - The program terminates with the return value EXIT_SUCCESS without printing anything.

- LINES - The command prints the lines one after the other in alphabetical order. For example:

  ```
  tramway> LINES
  All tramlines in alphabetical order:
  East
  Hospital
  South
  West
  tramway>
  ```

  Here alphabetical order means the order according to ASCII codes. It almost like alphabetical order but all upper-case letters precede all lower-case letters. For example, if we had above written East as east, it would have been written as the last one in the list.

- LINE <line> - The command prints the stops belonging to the line <line>. Each

stop will be printed in the line of its own with a preceeding dash. The stops will be printed in the order of the line, i.e. in the same order as they were given in the input file. For example:

```
tramway> LINE East
Line East goes through these stops in the order they are listed:
 - Sammonaukio : 0
 - Kaleva : 0.2
 - Uintikeskus : 0.3
 - Kalevanrinne : 0.6
 - Hakametsa : 1
 - Turtola : 3
 - Hallila : 5
 - Hervanta : 7
 - Hervannan kampus : 7.1
tramway>
```

If a line has no stops, only the first line of the above output would be printed. Such a situation can be seen more precisely at the command ADDLINE a bit later.

If the decimal part of a floating point number is zero, it is not printed. For example, the output of the distance of Hervanta is 7, instead of 7.0. The output does not depend on which form the distance was given in the input file. (This is the default way of printing floating point numbers, and thus, you are not required to implement any special actions for this.)

The command must have at least one parameter. If this is not the case, the program gives the error message:

```
tramway> LINE
Error: Invalid input.
tramway>
```

If the given line is unknown, the program prints the error message:

```
tramway> LINE North
Error: Line could not be found.
tramway>
```

- STOPS - The command prints the stops one after the other in alphabetical order. (Again the order is based on ASCII code as with the command LINES.) For example:

```
tramway> STOPS
All stops in alphabetical order:
Etela-Hervanta
Hakametsa
Hallila
Hervannan kampus
Hervanta
Hervantajarvi
Hippos
Kaleva
Kalevan kirkko
Kalevanrinne
Kaupin kampus
Keskustori
Koskipuisto
Pyynikintori
Rautatieasema
Sammonaukio
TAYS
Tulli
Turtola
Tuulensuu
```

```
Uintikeskus

tramway>
```

- STOP <stop> - The command prints in alphabetical order (based on ASCII codes) those lines that have the stop <stop> in their route. Each line will be printed in the line of its own with a preceeding dash. For example:

```
tramway> STOP Sammonaukio

Stop Sammonaukio can be found on the following lines:

 - East

 - Hospital

 - West

tramway>
```

The command requires at least one parameter. If this is not the case, the given error message is the same as with the command LINE in the corresponding situation.

If the given stop is unknown, the program prints the error message:

```
tramway> STOP Iidesjarvi

Error: Stop could not be found.

tramway>
```

The name of a stop can consist of several words. In such cases, the name must be enclosed with quotes, since otherwise the name cannot be considered as a single parameter. For example:

```
tramway> STOP "Hervannan kampus"

Stop Hervannan kampus can be found on the following lines:

 - East

 - South

tramway>
```

Even if the name of a stop consists only of a single word, the user is allowed to use quotes. For example the commands STOP Sammonaukio and STOP "Sammonaukio" have the same effect.

The abovementioned things hold also in the name of a line, if it consists of several words.

- DISTANCE <line> <stop1> <stop2> - The command counts the distance between two stops (<stop1> and <stop2>) in line <line>. The stops can be given in both orders, and the result is always a non-negative number. For example:

```
tramway> DISTANCE East Sammonaukio Uintikeskus
Distance between Sammonaukio and Uintikeskus is 0.3
tramway> DISTANCE West Tulli Tuulensuu
Distance between Tulli and Tuulensuu is 2
tramway> DISTANCE West Tuulensuu Tulli
Distance between Tuulensuu and Tulli is 2
tramway>
```

As told with the command LINE the decimal part zero of a floating point number is not printed.

The given two stops can be the same, whereupon the program counts the distance from a stop to itself, resulting zero.

The command requires at least three parameters. If this is not the case, the given error message is the same as with the command LINE in the corresponding situation.

If the given line is unknown, the program prints the same error message as shown with the LINE command, i.e.:

```
Error: Line could not be found.
```

If the given stop is unknown, the program prints the same error message as shown with the STOP command, i.e.:

```
Error: Stop could not be found.
```

If both the line and a stop (one or both of them) are unknown, the program informs only about the unknown line. If the line exists but both the stops are unknown, an error message concerning an unknown stop is given only once.

- ADDLINE <line> - The command adds the line <line> in the data structure (without stops). For example:

```
tramway> ADDLINE Pirkkala

Line was added.

tramway> LINES

All tramlines in alphabetical order:

East

Hospital

Pirkkala

South

West

tramway> LINE Pirkkala

Line Pirkkala goes through these stops in the order they are listed:

tramway>
```

From the execution above, you can also see how to print a line that has no stops (yet).

The command requires at least one parameter. If this is not the case, the given error message is the same as with the command LINE in the corresponding situation.

If the line to be added already exists, the program prints the following error message:

```
tramway> ADDLINE South

Error: Stop/line already exists.
```

```
tramway>
```

- ADDSTOP <line> <new stop> <distance> - The command adds in the data structure the stop <new stop> for the line <line>. The stop <new stop> will be added in such a point in the line that the ascending order of the stops is preserved. For example:

```
tramway> ADDSTOP Hospital Teiskontie 0.2

Stop was added.

tramway> ADDSTOP Hospital Teisko 30

Stop was added.

tramway> LINE Hospital

Line Hospital goes through these stops in the order they are listed:

 - Sammonaukio : 0

 - Kalevan kirkko : 0.1

 - Teiskontie : 0.2

 - Hippos : 0.4

 - TAYS : 0.6

 - Kaupin kampus : 0.7

 - Teisko 30

tramway>
```

The command requires at least three parameters. If this is not the case, the given error message is the same as with the command LINE in the corresponding situation.

If the line given as the first parameter is unknown, the given error message is the same as with the command LINE in the corresponding situation.

If the stop to be added already exists in the line (for example, added just previously), or the line has already a stop with the same distance, the program prints the error message:

```
tramway> ADDSTOP Hospital Teiskontie 0.3
```

```
Error: Stop/line already exists.
tramway> ADDSTOP Hospital Kaleva 0.2
Error: Stop/line already exists.
tramway> ADDSTOP Hospital Teiskontie 0.2
Error: Stop/line already exists.
tramway>
```

Even if both stop and distance were given earlier, only one error message is printed.

- REMOVE <stop> - The command removes the stop <stop> from all the lines. For example:

```
tramway> REMOVE "Hervannan kampus"
Stop was removed from all lines.
tramway>
```

The command requires at least one parameter. If this is not the case, the given error message is the same as with the command LINE in the corresponding situation.

If the stop to be removed is unknown, the program prints the following error message:

```
tramway> REMOVE Pohjois-Hervanta
Error: Stop could not be found.
tramway>
```

The addition and remove commands described above do not require modifying the file. It is enough to just change the data structure.

If the user gives another command, not listed above, the program prints the error message:

```
tramway> EXIT
```

```
Error: Invalid input.

tramway>
```

After completing the execution of a command given by the user, the program prints the prompt again, until the user gives the command QUIT.

Both lower-case and upper-case letters are allowed in the commands. (Here you can use toupper function provided for char type. Before comparing user input e.g. to the word LINES, change all the letters in the user-given command into upper-case ones.)

Tips

- For splitting the file input and the user input, you can use the function split which you implemented in the weekly exercises. The implementation was given in the templates of the exercises Sum of vector elements and Network marketing, both from Round 6.
- Recall that an iterator becomes invalid if the container changes, i.e. elements are added or removed.
- Recall that as a default, the program will be compiled into a build- directory. So, for testing purposes, you can move the input data file to the same directory with the binary code.
- Before implementing the program, think carefully how to structure the program to be able fulfill all the requirements. Note that a stop can belong to several lines, and sometimes printing is done in alphabetical order (based on ASCII codes) and sometimes in the order specified in the input file.
- Consider how to divide the program into parts. If you wish, you can use classes, but you will not lose points for not using them. It more important that you can divide the code into functions that are not too long.

Implementing the program in parts

There are basically three parts in the implementation of the program:

- choice of the data structure and reading the file into the data structure
- user interface

- search algorithms.

Consider carefully in which order you want to implement them so that it is possible for you to create and test them one by one before moving on to the next step. **The minimum requirement for using the version control is these three parts having been executed as separate commits**, but it would, of course, be best if you had more commits.

Special requirements

If you want your assignment to pass the evaluation, you must meet these requirements:

- You can only read the input file once during the program execution.
    - This means that you must read the data from the input file to STL data structures. After that all searches, additions, and removals must be done in the chosen data structures (not in the input file).
- You must use STL library, not e.g. Boost.

Evaluation

To end up to assistents' evaluation, your work must first pass the automatic tests. If the automated tests give 0 points for your work, also your final points will be 0, and your work will not be evaluated by an assistant.

The assistant evaluates the submissions that have passed the automated testing (= 1 p) **and fulfilled the special requirements of the assignment** based on the last commit before the deadline, according to the following criteria:

- The overall principle of the solution: 0-30 points:
    - The learning goals of the exercise have been achieved.
    - The program code has been split into logical, suitably long segments using functions, classes, and/or methods.
    - If the program uses classes and objects, these have been implemented according to the basics of object-oriented programming (see from round 4 the section About programming style at Object-oriented programming).

        - The data structure does not include repetitive data nor unnecessary parts. The chosen data structures have been used in a reasonable way.
        - The program code does not include unnecessary repetitions nor other unnecessary parts.
        - The program code does not include unnecessary limitations or assumptions or other forced solutions.
        - The implementations of the program’s structures are easy to understand.
        - Global variables have not been used in the program code (global constants are OK).
        - The program does not terminate with the exit function.
- Programming style: 0-10 points:
    - Variables and functions are named clearly and appropriately.
    - Use named constants instead of magic numbers.
    - The program code is neatly formatted.
    - The length of program code lines do not exceed 80 characters.
    - At the beginning of each file, there is a comment explaining the purpose of the file, the creator(s) of the project and other necessary information (see the assignment fo the first project on round 4).
    - At the beginning of each function/method (in the header file if possible), there is a comment describing its working, return value and parameters.
    - There are comments in the code where necessary.
    - All the variables have been initialized.
    - The compiler does not give warnings while compiling.
- Using the version control: 0-10 points:
    - There are enough commits.
    - The content of commit messages is clear and relevant.