Project Statement 1 - IAED 2022/23

Delivery date: March 24, 2023, at 19:59

LOG changes

- 3mar23 - Publication of the statement.

1. Introduction

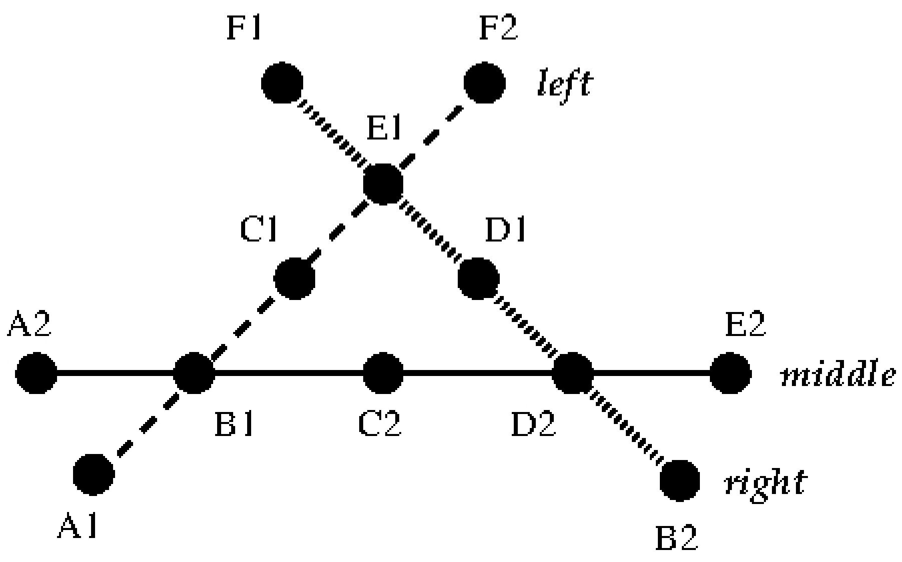
The aim is to build a public transport route management system. To this end, the system should allow stations (stops) and routes to be defined and consulted.

Interaction with the program should take place via a set of lines consisting of a letter (command) and a number of arguments depending on the command to be executed. You can assume that all *input* provided will respect the types indicated, for example where a decimal integer value is expected, a letter will never be entered. The possible commands are listed in the following table and indicate the operations to be performed.

Comano	**Action**
q	ends the program
c	adds and lists careers
p	adds and lists stops
l	adds ~~and lists~~ connections between stops
i	lists the interconnection nodes

2. Problem specification

The aim of the project is to have a public transport route management system. To this end, a set of routes are created whose paths correspond to a sequence of connections between stops.



Each **track** (*left*, *right* or *middle in* the figure) is characterized by a name made up of letters or decimal digits, a source stop and a destination stop. The length of the name cannot exceed **20** *bytes*. Note that an accented character in *utf-8* uses more than one *byte*. For example `square` has 5 letters but takes up 6 *bytes* (`char` in **C**).

A **stop** (*A1*, *A2*, *B1*, ... in the figure) is characterized by a name and a location, latitude and longitude, represented as real numbers in floating point. The name describing the stop may contain white characters (spaces or horizontal tab `\t`). In this case, the name is enclosed in quotation marks. If there are no white characters, the name can be delimited by quotation marks or not. The name never contains a quotation mark in its description. The length of the name cannot exceed **50** *bytes*.

Each **connection** (*A1* to *B1*, *B1* to *C1,* ... in the figure) is characterized by a **route**, two **stops** (origin and destination), a cost and a duration. To be a valid connection, it must represent an extension of the route, i.e. the origin of the connection is the same stop as the terminus of the route or the destination of the connection is the same stop as the departure station of the route. The cost and duration are represented as real numbers in floating point and must be printed to two decimal places (`%.2f`).

Note that there can be circular routes. When you add a connection from the last stop on a route to the first stop on the route, a cycle is formed. In these situations, the new connection is added at the end of the route.

There can be a maximum of **200** routes, **10000** stops and **30000** connections.

3. Input data

The program must read the input data from the terminal command line. No command line exceeds **BUFSIZ** *bytes* (*8192 bytes* on most systems).

During program execution, instructions must be read from the terminal (*standard input*) in the form of a set of lines beginning with a character, which is referred to as a command, followed by a number of pieces of information depending on the command to be executed; the command and each piece of information are separated by at least one white character.

The available commands are described below. The characters `<` and `>` are only used in the description of the commands to indicate the parameters. Optional parameters are indicated between characters `[` and `]`. Repeats are indicated between `{` and `}` characters. Each command always has all the parameters necessary for its correct execution. The characters `...` characters indicate possible repetitions of a parameter.

Each command indicates a particular action, which is characterized in terms of input format, output format and errors to be returned.

If the command generates more than one error, only the first one should be indicated.

- **q** - ends the program:
- Input format: `q`
- Output format: NADA

- **c** - adds and lists careers:
- Input format: `c [ <career-name> [ inverse ] ]`
- Output format without arguments: `<career-name> <stop-origin> <stop-destination> <number-of-stops> <total-cost> <total-duration`>, in the order of creation. Stops are omitted for routes without connections.
- Output format with arguments: `<stop-origin> { , <stop> }`, for the route sequence if no new career is created.
- Note: the `reverse` parameter implies that the stops are printed in reverse order and can be abbreviated up to 3 characters. ~~Otherwise nothing should be printed~~.
- Errors:
  - `incorrect sort option.` if after the career name there is a word other than `inverse` or one of its abbreviations up to 3 characters.

- **p** - adds and lists stops:
- Input format: `p [ <stop-name> [ <latitude> <longitude> ] ]`
- Output format without arguments: `<stop-name>: <latitude> <longitude> <number-of-rails>` for each stop and in the order of creation, one per line where `<number-of-rides>` is the number of rides that stop at the stop.
- Output format with one argument: `<latitude> <longitude`>.
- If the command is invoked with three arguments, a new stop is created without generating data in the output.
- Note: The `<stop-name>` must be enclosed in quotation marks if the name contains white characters (space or horizontal tab); the name cannot contain the quotation mark character.
- Errors:
  - `<stop-name>: stop already exists.` if a stop is created and a stop with the specified name already exists.
  - `<stop-name>: no such stop.` if a stop is listed and there is no stop with the indicated name.
- The coordinates are printed in 16 positions with 12 decimal digits (`%16.12f`).

- **l** - adds links:
- Input format: `l <career-name> <stop-origin> <stop-destination> <cost> <duration>`
- Output format: NADA
- Notes:
  - The `<parent name>` must be enclosed in quotation marks if the name contains white characters (space or horizontal tab); the name cannot contain the quotation mark.
- Errors:

    - `<career-name>: no such line.` if there is no career with the indicated name.
    - `<stop-name>: no such stop.` if there is no stop with the indicated name, origin or destination.
    - `link cannot be associated with bus line.` in case the stops of the link, origin or destination, do not correspond to one of the ends of the route.
    - `negative cost or duration.` in case the cost or duration is negative.

- **i** - list the intersections between careers:
- Input format: `i`
- Output format: `<stop-name> <career-number>: <career-name> ...` for each stop where more than one line stops, one per line in the order in which the stops were created. The route names must be listed alphabetically.

You can only use the library functions defined in `stdio.h`, `stdlib.h`, `ctype.h` and `string.h`

Important note: you are not allowed to use the `goto` statement, the `extern` statement or the C native `qsort` function and none of these *names* should appear in your code.

Examples of how to use the commands

Consider the rows in the picture above.

Command `c`

```
c
```

The `c` command with no arguments allows you to list all the careers on the system.

```
c middle
```

The `c` command followed by an existing route in the system allows you to list all the stops on the route from the origin to the destination.

```
c middle inverse
```

The same as above, but the stops are listed from destination to origin.

```
c down
```

The `c` command followed by a career that doesn't exist in the system allows the new career to be created. In this case there is nothing to show in the output.

Command `p`

```
p
```

The `p` command without arguments allows you to list all system stops.

```
p C1
```

The `p` command with one argument shows the latitude and longitude of the stop.

```
p X1 2.5 45.6
```

The `p` command with three arguments creates a new stop.

Command `l`

```
l left X1 A1 0.5 2
```

The `l` command allows you to add new connections to a route. In this case, if `X1` is the last stop on the route, then the new link is inserted at the end and `A1` is the new last stop. Otherwise, if `A1` is the start of the route, then the link is inserted at the start of the route and `X1` becomes the new start. If both conditions are met (insertion of a cycle in the path), then the link is inserted at the end.

`i` command

```
i
```

The `i` command allows you to list the stops that correspond to route intersections.

4. Compilation and testing

The compiler to use is `gcc` with the following compilation options: `-O3 -Wall -Wextra -Werror -ansi -pedantic`. To compile the program, run the following command:

```
 $ gcc -O3 -Wall -Wextra -Werror -ansi -pedantic -o proj1 *.c
```

The program must write the answers to the commands presented in the *standard input* into the *standard output*. The answers are also lines of text formatted as defined earlier in this statement. Pay attention to the number of spaces between elements of your output, as well as the absence of spaces at the end of each line. Make sure you follow the instructions given.

See the example inputs and outputs in the `public-tests/` folder. The

program should be executed as follows:

```
 $ ./proj1 < test.in > test.myout
```

You can then compare your output (`*.myout`) with the expected output (`*.out`) using the `diff` command,

```
$ diff test.out test.myout
```

To test your program you can perform the steps above or use the `make` command in the `public-tests/` folder.

5. Project Delivery

A `git` repository will be created for each student to develop and submit the project. This repository will be created in RNL's GitLab and will be activated when this statement is published.

In your project submission you should consider the following points:

- Your project development files (`.c` and `.h`) are considered to be in the root of the repository and not in a directory. *Any file outside the root will not be considered to belong to your project*.
- The latest version that is in the RNL repository will be considered the submission for project evaluation. Any previous version or version that is not in the repository will not be considered in the evaluation.
- Before making any submission to the RNL repository, don't forget that you should always do a `pull` to synchronize your local repository.
- When you update the `.c` and `.h` files in the `src` directory `in`your RNL repository, this version will be evaluated and you will be informed whether that version gives the expected response in a set of test cases. As with the lab repository, the result of the automatic evaluation will be placed in the student's repository.
- For the evaluation system to run, you have to wait at least 10 minutes. Every time you make an update to the repository, a new 10-minute waiting period begins. Examples of test cases will be provided in due course.
- Project submission deadline: **March 24, 2023, at 19:59**. You can make as many submissions as you like up until the deadline, and the latest version will be used for evaluation purposes. You should therefore check carefully that the latest version in RNL's GitLab repository corresponds to the version of the project you want to be evaluated. There will be no exceptions to this rule.

6. Project evaluation

The following components will be taken into account when evaluating the project:

1. The first component assesses the performance of the program's functionality. This component is graded between 0 and 16.
2. The second component assesses the quality of the code delivered, namely the following aspects: comments, indentation, structuring, modularity, abstraction, among others. This component can vary between -4 values and +4 values in relation to the rating calculated in the previous item and will be assigned later. Some *guidelines* on this topic can be found at guidelines.
3. The rating for the first component of the project evaluation is obtained by automatically running a set of tests on a computer with the GNU/Linux operating system. It is therefore essential that the code compiles correctly and respects the data input and output format described above. Projects that do not comply with the format indicated in the statement will be penalized in the automatic assessment and may, at the limit, score 0 (zero) if they fail all the tests. The tests considered for assessment may (or may not) include those available on the course website, as well as a

set of additional tests. The execution of each program in each test is limited in the amount of memory it can use, and in the total time available for execution, the time limit being different for each test.

4. It should be noted that the fact that a project successfully passes the set of tests provided on the course page does not imply that the project is completely correct. It only indicates that it has successfully passed some tests, but this set of tests is not exhaustive. It is the students' responsibility to ensure that the code they produce is correct.
5. Under no circumstances will any information be made available about the test cases used by the automatic assessment system. All the test files used to assess the project will be made available on the course page after the submission date.

7. Project Development Tips

Below you can find some simple tips that make it easier to spot common errors in project development. We suggest that you **develop your projects incrementally and that you test your solutions locally before updating them in the remote repository**.

We suggest you follow these steps:

1. Develop and correct the code incrementally, ensuring that it compiles without errors or *warnings*. Don't accumulate a series of errors because *debugging* is more complex the larger the code base to be analyzed.
2. Make sure that you are reading the *input* and writing the *output* correctly, in particular make sure that the *strings* don't have extra spaces, `\` at the end, that the formatting is correct and in line with the statement, *etc.*
3. Try to develop the commands in the order presented.
4. Test each control separately and check that it works correctly.