Assignment
Logistics

Change Log

We may make minor changes to the spec to address/clarify some outstanding issues. These may require minimal changes in your design/code, if at all. Students are strongly encouraged to check the change log regularly.

13 November

- Typo in scanf() sample code corrected.

25 October

- Priority queue listed as admissible ADT

Version 1: Released on 20 October 2022

Objectives

The assignment aims to give you more independent, self-directed practice with

- advanced data structures, especially graphs
- graph algorithms
- asymptotic runtime analysis

Admin

Marks	3 marks for stage 1 (correctness) 4 marks for stage 2 (correctness) 3 marks for stage 3 (correctness) 1 mark for complexity analysis 1 mark for style Total: 12 marks
Due	11:00:00am on **Monday** 14 November (week 10)
Late	5% reduction per day late, capped at 5 days (= 120 hours) (e.g. if you are 25 hours late, your mark will be reduced by 1.2 marks)

Aim

Your task is to write a program `logistics.c` to help a company determine the minimum number of distribution centres so that all cities are within a maximum distance from at least one distribution centre.

Input

Cities

Your program should start by prompting the user to input a positive number *n* followed by *n* lines, each containing the name of a city. An example is:

```
prompt$ ./logistics
Enter the number of cities on the distribution network: 3
```

```
Berlin
Frankfurt
Hamburg
```

You may assume that:

- The input is syntactically correct.
- Names require no more than 31 characters and will not use any spaces.
- The names will be input in *alphabetical order*, with no name repeated.

Hint: To read a single line with the name of a city you may use:

```
scanf("%31s", city);   // city is a character array (= string variable)
```

(Originally it said ...%32..., which can be used also provided the string variable is large enugh, i e. 32+1 bytes long.)

Roads

Next, your program should ask the user for the number of roads, *m*, then prompt the user to input *m* roads. Each road requires the name of a city (from), the name of another city (to) and the distance (in km). An example is:

```
Enter the number of roads: 2
Enter the name of a city: Hamburg
Enter the name of a city: Frankfurt
Enter the distance: 492
Enter the name of a city: Berlin
Enter the name of a city: Hamburg
Enter the distance: 289
```

All roads are *directional*, that is, *from* the first city *to* the second city.

You may assume that:

- The input is syntactically correct: Each road will use the names of two different cities that have been input earlier, followed by a positive integer.
- There will be no two roads with the same start and end point.

Maximum distance

Finally, your program should prompt the user to input the required maximum distance to a distribution centre:

```
Enter the required maximum distance: 350
```

Again, you may assume that the input is correct (a non-negative integer).

Stage 1 (3 marks)

For stage 1, you should demonstrate that you can read the input and generate a suitable data structure.

For this stage, all test cases are guaranteed to satisfy the following conditions:

- There is only **one** route that:
    - starts in the (alphabetically) **first** city,
    - goes through all other cities (but not necessarily in alphabetical order).

  Which means that in each scenario, if *n* is the number of cities, then there will be exactly *n* - 1 roads.

- The required maximum distance is large enough so that the unique optimal solution is to have one distribution centre, in the first city.

Hence, all you need to do for this stage is to

1. output the name of the first city
2. find and output, for each city in alphabetical order, the unique route from the first city, along with the distance.

Here is an example to show the desired behaviour of your program for a stage 1 test:

```
prompt$ ./logistics
Enter the number of cities on the distribution network: 3
Berlin
Frankfurt
Hamburg
Enter the number of roads: 2
Enter the name of a city: Hamburg
Enter the name of a city: Frankfurt
Enter the distance: 492
Enter the name of a city: Berlin
Enter the name of a city: Hamburg
Enter the distance: 289

Enter the required maximum distance: 1000

Hubs: Berlin
Routes:
Berlin: Berlin 0
Frankfurt: Berlin - Hamburg - Frankfurt 781
Hamburg: Berlin - Hamburg 289
prompt$
```

Stage 2 (4 marks)

For stage 2, you should extend your program for stage 1 such that it finds an optimal solution, that is, a minimal set of distribution centres (hubs) so that each city is within the maximum required distance from one of the centres.

For this stage, all test cases are guaranteed to satisfy the following condition:

- There is always a unique solution, i.e. a unique smallest set of distribution centres that satisfies the requirements.

Hence, for this stage you need to:

1. find and output an optimal solution
2. output, for each city in alphabetical order, the best way to reach it from the closest distribution centre.

Here is an example to show the desired behaviour and output of your program for a stage 2 test:

```
prompt$ ./logistics
Enter the number of cities on the distribution network: 3
Berlin
Frankfurt
Hamburg
Enter the number of roads: 2
Enter the name of a city: Hamburg
Enter the name of a city: Frankfurt
Enter the distance: 492
Enter the name of a city: Berlin
Enter the name of a city: Hamburg
Enter the distance: 289

Enter the required maximum distance: 350

Hubs: Berlin, Frankfurt
Routes:
Berlin: Berlin 0
Frankfurt: Frankfurt 0
Hamburg: Berlin - Hamburg 289
prompt$
```

Stage 3 (3 marks)

For stage 3, you should extend your program for stage 2 such that:

> *If there are two or more solutions with the same minimum number of distribution centres, you should output **all** minimal solutions, in alphabetical order of the distribution centres.*

For example, if {Berlin,Hamburg} and {Berlin,Frankfurt} are two optimal solutions, then the first solution in your output should be: `Berlin, Frankfurt`.

Here is an example to show the desired behaviour and output of your program for a stage 3 test:

```
prompt$ ./logistics
Enter the number of cities on the distribution network: 3
Berlin
Frankfurt
Hamburg
Enter the number of roads: 2
Enter the name of a city: Hamburg
Enter the name of a city: Frankfurt
Enter the distance: 492
Enter the name of a city: Berlin
Enter the name of a city: Hamburg
Enter the distance: 289

Enter the required maximum distance: 500

Hubs: Berlin, Frankfurt
Routes:
Berlin: Berlin 0
Frankfurt: Frankfurt 0
Hamburg: Berlin - Hamburg 289

Hubs: Berlin, Hamburg
Routes:
Berlin: Berlin 0
Frankfurt: Hamburg - Frankfurt 492
Hamburg: Hamburg 0
prompt$
```

Hint: For each solution, you only need to output *one* shortest route for each city. There will be no test cases in which a city is at equal distance from two distribution centres, or where there are two different shortest routes from a hub to a city.

Complexity Analysis (1 mark)

Your program should include a time complexity analysis for the worst-case asymptotic running time of your program, in Big-Oh notation, depending on the number *n* of cities.

Hints

If you find any of the following ADTs from the lectures useful, then you can, and indeed are encouraged to, use them with your program:

- linked list ADT : `list.h`, `list.c`
- stack ADT : `stack.h`, `stack.c`
- queue ADT : `queue.h`, `queue.c`
- priority queue ADT : `PQueue.h`, `PQueue.c`
- graph ADT : `Graph.h`, `Graph.c`
- weighted graph ADT : `WGraph.h`, `WGraph.c`

You are free to modify any of the six ADTs for the purpose of the assignment (*but without changing the file names*). If your program is using one or more of these ADTs, you should submit both the header and implementation file, even if you have not changed them.

Your main program file `logistics.c` should start with a comment: `/* … */` that contains the time complexity of your program in Big-Oh notation, together with a short explanation.

Testing

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to this assignment. It expects to find, in the current directory, the program* `logistics.c` *and any of the admissible ADTs* (`Graph,WGraph,stack,queue,PQueue,list`) *that your program is using, even if you use them unchanged. You can use dryrun as follows:*

```
prompt$ 9024 dryrun logistics
```

Please note: Passing dryrun does not guarantee that your program is correct. You should thoroughly test your program with your own test cases.

Submit

For this project you will need to submit a file named `logistics.c` and, optionally, any of the ADTs named `Graph,WGraph,stack,queue,PQueue,list` that your program is using, even if you have not changed them. You can either submit through WebCMS3 or use a command line. For example, if your program uses the `Graph` ADT and the `queue` ADT, then you should submit:

```
prompt$ give cs9024 assn logistics.c Graph.h Graph.c queue.h queue.c
```

Do not forget to add the time complexity to your main source code file `logistics.c`.

You can submit as many times as you like — later submissions will overwrite earlier ones. You can check that your submission has been received on WebCMS3 or by using the following command:

```
prompt$ 9024 classrun -check assn
```

Marking

This project will be marked on functionality in the first instance, so it is very important that the output of your program be ***exactly*** correct as shown in the examples above. Submissions which score very low on the automarking will be looked at by a human and may receive a few marks, provided the code is well-structured and commented.

Programs that generate compilation errors will receive a very low mark, no matter what other virtues they may have. In general, a program that attempts a substantial part of the job and does that part correctly will receive more marks than one attempting to do the entire job but with multiple compilation or runtime errors.

Style considerations include:

- Readability
- Structured programming
- Good commenting

Plagiarism

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar projects in previous years, if applicable) and serious penalties will be applied, particularly in the case of repeat offences.

- ***Do not copy ideas or code from others***
- ***Do not use a publicly accessible repository or allow anyone to see your code, not even after the deadline***

Please refer to the on-line sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Academic Integrity and Plagiarism
- UNSW Plagiarism Policy
- UNSW Plagiarism Management Procedure

Help

See FAQ for some additional hints.

Finally …

Have fun! Michael