Your Task

**Part 1: Link State Algorithm Basic Form**

1. Reads information about a topology from the standard input.
2. Uses link state updates and Dijkstra's algorithm, as appropriate and:  
   * Output the information in various tables in the required format during intermediate steps.
3. Read updates to the topology from standard input and repeat step 2 above, until no further input is provided.

The Dijkstra's algorithm program you are to provide should be named Dijkstra.

**Part 2:**

**Link State Algorithm Advanced Form**

The second part of the assignment is to write a second version of the program that is the same as the first, except that network is stored in a data structure with ***log(n)*** performance for all relevant operations. This will require small modifications to your  Dijkstra program.

The modified Dijkstra's algorithm program you are to provide should be named DijkstraNlogN.

**Report & Stress Test**

You should produce a report, no longer than ***1 page*** in length as a PDF document named, Comparison.PDF added to your SVN repository to present clear evidence that your two solutions are producing results in the appropriate complexity bound. You should provide a ***graph and textual explanations*** that support this argument. The report, test cases, and script for generating results are worth 50 marks.

The report marks are allocated based on the following:

* Content of the report ***graph and brief textual explanations***, justifying the complexity of the two implementations.
* Script/program/commands described in the report to run the programs Dijkstra and DijkstraNlogN along with the network configurations submitted to SVN and used to generate the data used for the graphs. These scripts should print out the summary run time information in a table format with the following headings:

     For Dijkstra with N \* N complexity

***Number of Nodes     Number of Links     Execution Time (N\*N)***

***...                                ...                               ....***

For Dijkstra with N log N complexity

***Number of Nodes     Number of Links     Execution Time (N log N)***

***...                                ...                               ....***

* Use at least 4 data points for your graph, i.e. four network typologies with increasing numbers of nodes.

**In Your Task**

You will need to craft any internal data structures and design your program in such a way that it will reliably and correctly generates the correct tables. We have *deliberately* not provided you with a code templates and this means that you will have more freedom in your design but that you will have to think about the problem and come up with a design.

Then you will generate four data points to compare the runtime of Dijkstra and DijkstraNlogN.

**You will need to record your progress and development cycle in a logbook as described in the 'Before you Begin' section above**.

Programming Language/Software Requirements

You may complete this assignment using the programming language of your choice, **with the following restrictions**:

* For **compiled** languages (Java, C, C++ etc.) you must provide a Makefile.
  + Your software will be compiled with make (***Please look at this resource on how to use Makefile build tool:***[***https://makefiletutorial.com/ (Links to an external site.)***](https://makefiletutorial.com/))
  + Pre-compiled programs will ***not*** be accepted.
* Your implementation must work with the versions of programming languages installed on the Web Submission system, these are the same as those found in the labs and on the **uss.cs** server and include (but are not limited to):
  + **C/C++:** g++ (GCC) 4.8.5
  + **Java:** java version "1.8.0\_201"
  + **Python:** python 2.7.5 or python 3.6.8
* Your implementation may use any libraries/classes available on Web Submission system, but **no external libraries/classes/modules**.
* Your programs will be executed with the command examples below:
  + For C/C++
  + make
  + ./Dijkstra
  + ./DijkstraNlogN

You can find a ***simple*** example makefile for C++ [HERE](https://myuni.adelaide.edu.au/courses/74994/files/10522912/download?wrap=1)[Download HERE](https://myuni.adelaide.edu.au/courses/74994/files/10522912/download?download_frd=1).[***A good resource is here: https://makefiletutorial.com/****(Links to an external site.)*](https://makefiletutorial.com/)  
This will ***need to be customised*** for your implementation. Make sure you use tabs (***actual tab characters***) on the indented parts

* + For java:
  + make

java Dijkstra  
java DijkstraNlogN

You can find a ***simple*** example makefile for Java [HERE](https://myuni.adelaide.edu.au/courses/74994/files/10522911/download?wrap=1)[Download HERE](https://myuni.adelaide.edu.au/courses/74994/files/10522911/download?download_frd=1). [***A good resource is here: https://makefiletutorial.com/****(Links to an external site.)*](https://makefiletutorial.com/)  
This will ***need to be customised*** for your implementation. Make sure you use tabs (***actual tab characters***) on the indented parts

* + For Python (no make file is needed):

./Dijkstra  
./DijkstraNlogN

Programs written using an interpreted language such as python:

* + - Will need to use UNIX line endings (always test on a uni system such as the **uss**cloud instance).
    - Will not be built with make (as shown above, because they are not compiled)
    - Will require a 'shebang line' at the start of your file to run as above.  
      e.g. #!/usr/bin/env python2 (Python 2) or #!/usr/bin/env python3 (Python 3).

Algorithm

Key Assumptions

In a real routing environment, messages are not synchronised and routers send out their initial messages with their neighbours as needed or configured (i.e. after a certain time interval has elapsed).

In this environment, to simplify your programs:

* Link state routing messages are not part of the simulation and you can assume that all of the routers will have their Link-State Databases synchronized the moment a new link is established/added/removed by reading a link state from the standard input. So the Link-State Databases (LSDBs)  are always synchronised and no intermediate steps need to be modelled.
* All neighbour tables are instantly updated at every router, the moment a new link is established/added/removed by reading a link state from the standard input.
* When DIjksta algorithm is run, and multiple least cost paths exist to select from in an iteration, always select the first one in alphabetical order (in alphabetical order, by router name, so if you have a choice of A, B and C, then select A and add it to the list of nodes for which the shorted path is now known).
* Where multiple best routes exist, always select the first one in an alphabetical ordering of the next hop router name.

Dijkstra's Algorithm

At each node, :

D(v) = min(D(v), D(w) + c(w,v))

The cost to node v is the minimum cost among the current cost to node v and the cost to the via node w plus the cost to node v from the via node w.

At each node x:  
  
INITIALISATION:  
N’ = {u}   
for all nodes v   
 if v is a neighbor of u  
 then D(v) = c(u,v)   
 else D(v) = ∞   
  
LOOP  
find w not in N' such that D(w) is a minimum  
add w to N’   
update D(v) for each neighbor v of w and not in N':  
 D(v) = min(D(v), D(w) + c(w,v))   
 /\* new cost to v is either old cost to v or known least path cost to w plus cost from w to v\*/  
until N' = N

Expected Input Format

Sample Topology

Consider the following network sample topology in our description of the required input format:

Your program will need to read input from the standard input (terminal/command line) to construct such a given topology. Then, the perform  updates, as illustrated below, and also given via the standard input (terminal/command line).

Please refer to this sample topology with the following updates above when looking into the expected input examples below.

The expected input format, taking the sample topology as an example, is shown below:

X

Y

Z

LINKSTATE

X Z 7 X,Y

X Y 2

Y Z 1 X,Z

UPDATE

Y Z -1 X,Y

X Z 5 Y,Z

END

**The input begins with name of each router/node in the topology.**

* + Each **name** is on a new line.
  + Router **names** are case-sensitive.
  + Router **names** may not contain spaces.
  + This section ends with the keyword "LINKSTATE".

**The input continues with the details of each link/edge in the topology followed by a list of chosen router(s).**

* + Written as the names of two routers/nodes followed by the weight of that link/edge, then an ***optional list*** of routers (names separated by comma , and the list of chosen routers can be empty).
  + Your algorithm should run after each line of "LINKSTATE" input and build/update the neighbour table, link-state database (LSDB), and routing tables for all routers with links to other routers, according to the details of the link/edge given in the input. Then, show the **Expected Output**for each router in the list of chosen routers.
  + Weight values should always be integers.
  + A weight value of -1 indicates a link/edge to remove from the topology if present.
  + This section ends with the keyword "UPDATE".

**The input continues with the link state update details of each link/edge in the topology given a link and cost followed by a list of chosen routers.**

* + The values in each line of input in this section should be used to update the current topology.
  + As above, a weight value of -1 indicates a link/edge to remove from the topology if present.
  + Your implementation should update the neighbour tables, link-state databases, and routing tables as a result of receiving the link state update given in the input. Then, show the **Expected Output**for each router accordingto the list of chosen routers (if any).
  + If an unseen new router/node name has been inputted in this section, your program should be able to **add** this new router into the topology.  
    - Y A 10 X,Y
    - From the example input given above, your program should add **A** as a new router into the topology where it has a link with a cost of 10 to **Y**. Then show the **Expected Output**for routers X and then Y. Again, the *list of chosen routers* (X,Y in the example) is optional and none may be given.
  + A user may input 0 or more lines of link-state updates in this section.
  + Continues until the keyword, "END" is inputted, at which point the program exits normally.

Expected Output Format

Each router should maintain a **Neighbour Table**, **Link-state Database** **(LSDB)** and **Routing Table**.  We will ask you to print to standard out (screen/terminal) the

* **Neighbour Table**
* **Link-state Database (LSDB),** and
* **Routing Table**

of the chosen routers in **alphabetical order.**

**Neighbour Table**

The Neighbour Table includes all neighbours of the router and the cost to the neighbours.

X Neighbour Table:

Y,2

Z,7

1. The name of the router/node.
2. The neighbours of the router/node.
3. The cost/distance to the neighbours.
4. The neighbour and the cost/distance should be separated by a comma ,.
5. The neighbours need to be **printed in alphabetical order**.
6. A blank line at the end of the table.

**Link-state Database (LSDB)**

The Link-state Database includes all the unique links for the network topology reachable by the router along with the cost of each link in the topology. 

X LSDB:

X,Y,2

X,Z,7

Y,Z,1

1. The name of the router/node.
2. Each unique link is represented by 2 routers/nodes that are directly connected to each other. The 2 routers/nodes are separated by a comma ,.
3. The cost/distance of the unique link.
4. The unique link and the cost/distance should be separated by a comma ,.
5. The unique links need to be **printed in alphabetical order**(in the example, X, Y first and then X,Z).
6. If a link is registered as down/removed, do not include this in the LSDB.
7. A blank line at the end of the table.

**Routing Table**

The Routing Table includes the destination, next hop and total minimum cost/distance to the destination in the network topology reachable by the router.

X Routing Table:

Y,Y,2

Z,Y,3

where

1. The name of the router/node.
2. The name of the destination router/node.
3. The next hop.
4. The total minimum cost/distance to the destination.
5. The destinations need to be **printed in alphabetical order**.
6. A blank line at the end of the table.

Below is an example of what the output should look like for the *provided topology and link state updates*.

X Neighbour Table:  
Z,7  
  
X LSDB:  
X,Z,7  
  
X Routing Table:  
Z,Z,7  
  
Y Neighbour Table:  
  
Y LSDB:  
  
Y Routing Table:  
  
X Neighbour Table:  
Y,2  
Z,7  
  
X LSDB:  
X,Y,2  
X,Z,7  
Y,Z,1  
  
X Routing Table:  
Y,Y,2  
Z,Y,3  
  
Z Neighbour Table:  
X,7  
Y,1  
  
Z LSDB:  
X,Y,2  
X,Z,7  
Y,Z,1  
  
Z Routing Table:  
X,Y,3  
Y,Y,1  
  
X Neighbour Table:  
Y,2  
Z,7  
  
X LSDB:  
X,Y,2  
X,Z,7  
  
X Routing Table:  
Y,Y,2  
Z,Z,7  
  
Y Neighbour Table:  
X,2  
  
Y LSDB:  
X,Y,2  
X,Z,7  
  
Y Routing Table:  
X,X,2  
Z,X,9  
  
Y Neighbour Table:  
X,2  
  
Y LSDB:  
X,Y,2  
X,Z,5  
  
Y Routing Table:  
X,X,2  
Z,X,7  
  
Z Neighbour Table:  
X,5  
  
Z LSDB:  
X,Y,2  
X,Z,5  
  
Z Routing Table:  
X,X,5  
Y,X,7