1. You are encouraged to work in groups of two or three. If you cannot find partners, you may work alone. **2.** Please submit **one** copy of the assignment using `subversion`; if you are working with a partner, both names should appear on the assignment. **3.** Note that you will get a grade of zero if your program does not run.

In this assignment you are going to write a Python program that implements a *dynamic routing policy mechanism*. More precisely, you are going to implement a routing table management daemon, which maintains a link-state database according to the OSPF (Open Shortest Path First) interior routing protocol.

Call your program `routed` (as in *routing daemon*). Once started in command line, it awaits instructions and performs actions:

1. `add rt` ⟨*routers*⟩
   This command adds routers to the routing table, where ⟨*routers*⟩ is a comma separated list of (positive) integers and integer ranges. That is, ⟨*routers*⟩ can be `6,9,10-13,4,8` which would include routers

   `rt4,rt6,rt8,rt9,rt10,rt11,rt12,rt13`

   Your program should be robust enough to accept any such legal sequence (including a single router), and to return an error message if the command attempts to add a router that already exists (but other valid routers in the list ⟨*routers*⟩ should be added regardless).

2. `del rt` ⟨*routers*⟩
   Deletes routers given in ⟨*routers*⟩. If the command attempts to delete a router that does not exist, an error message should be returned; again, we want robustness: routers that exist should be deleted, while attempting to delete non-existent routers should return an error message (specifying the "offending" routers). The program should not stop after displaying an error message.

3. `add nt` ⟨*networks*⟩
   Add networks as specified in ⟨*networks*⟩; same format as for adding routers. So for example "`add nt 89`" would result in the addition of `nt89`. The handling of errors should be done analogously to the case of adding routers.

4. `del nt` ⟨*networks*⟩
   Deletes networks given in ⟨*networks*⟩.

5. `con` x y z
   Connect node x and node y, where x, y are existing routers and networks (for example, $x =$ `rt8` and $y =$ `rt90`, or $x =$ `nt76` and $y =$ `rt1`) and z is the cost of the connection. If x or y does not exist an error message should be returned. Note that the network is directed; that is, the following two commands are **not** equivalent: "`con rt3 rt5 1`" and "`con rt5 rt3 1`."

Important: Two networks cannot be connected directly; an attempt to do so should generate an error message. If a connection between x and y already exists, it is updated with the new cost z.

6. `display`
   This command displays the routing table, i.e., the *link-state database*. For example, the result of adding `rt3`, `rt5`, `nt8`, `nt9` and giving the commands "`con rt5 rt3 1`" and "`con rt3 nt8 6`" would display the following routing table:

```
    rt3  rt5  nt8  nt9
rt3      1
rt5
nt8  6
nt9
```

   Note that (according to the `RFC 2338`, describing OSPF Version 2) we read the table as follows: "column first, then row." Thus, the table says that there is a connection from `rt5` to `rt3`, with cost 1, and another connection from `rt3` to `nt8`, with cost 6.

7. `tree` x
   This commands computes the tree of shortest paths, with x as the root, from the link-state database. Note that x must be a router in this case. The output should be given as follows:

$$\begin{aligned} w_1 &: x, v_1, v_2, \ldots, v_n, y_1 \\ &: \text{no path to } y_2 \\ w_3 &: x, u_1, u_2, \ldots, u_m, y_3 \\ &\vdots \end{aligned}$$

   where w_1 is the cost of the path (the sum of the costs of the edges), from x to y_1, with v_i's the intermediate nodes (i.e., the "hops") to get from x to y_1. Every node y_j in the database should be listed; if there is no path from x to y_j it should say so, as in the above example output.

   Following the example link-state database in the explanation of the `display` command, the output of "`tree rt5`" would be:

```
1 : rt5,rt3
7 : rt5,rt3,nt8
  : no path to nt9
```

   Just as it is done in the OSPF (Open Shortest Path First) standard, the path-tree should be computed with Dijkstra's greedy algorithm.

   Finally, there may be several paths of the same value between two nodes; in that case, explain in the comments in your program how does your scheme select one of them.

8. `quit`
   Kills the daemon.