Problem:

In this exam, you are given a maze consisting of various rooms connected to each other via a direct door. In one of those rooms, there is a secret treasure and your purpose is to find that treasure. You do not know in which room the treasure is placed. Therefore starting from the entrance, you search for the treasure walking through room-by-room. During the search, you print the path that you follow until you reach the treasure.

Here are the details of the problem structure:

- The maze is actually a connected undirected graph. Each room is a node of the graph. If a room is connected to an other room, there is an edge between those two rooms.
- Each room is defined in the type of ***Struct Room***. This structure has 3 components:
  - *int id:* Each room has a unique id.
  - *char content*: Shows the content of the room. All rooms have the content of '-' character except the room containing the treasure. That room has the content of '*' character representing the treasure.
  - *vector<Room*> neighbors*: Holds a pointer for the rooms which are connected to the current room via a door.
- If a Room Y is defined as a neighbor to Room X, then you can be sure that Room X is also defined as a neighbor to Room Y in its neighborhood vector.
- The rooms of the maze will be given to the function as in the type of *vector<Room*>*.
- You are expected to return the path as vector of ids of rooms which are visited.

Here are the details of how to search/traverse the maze:

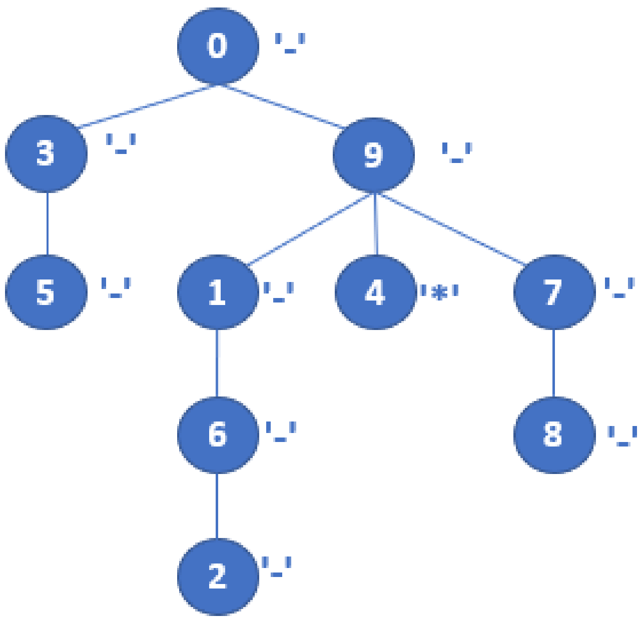
- You will actually do a kind of DFS. You will start from the first room (first means the firstly defined room, not the room with the first id) to traverse. You will pass to one of its neighbor rooms, and then to one of the neighbors of it, and to one of the neighbors of it, and so on. As you pass through a new room each time, you will add the id of that room to the output path. Upto here, it is exactly DFS.
- When you come to an end, that is a room with no unvisited neighbor, then you should turn back. While going back, you should also add the ids of the rooms that you need to visit one more time into the output path. For instance, assume that Room 5 is neighbor to Room 12 and assume that you come to Room 5 at some point and have not visited Room 12, yet. Also assume Room 12 is not neighbor to any other nonvisited room. Then, in your output path a pattern like the following have to exist: 5, 12, 5 . That means "you pass through Room 5, then Room 12, then you turn back to Room 5 again since there is not left any nonvisited room neighbor to Room 12. In short, in addition to usual DFS output, you are expected to print the nodes at each time you visit.
- When you find the treasure (The Room whose content is '*'), you should turn back totally. That is, you need to go back over the route that you follow. You should not go into any new room. During the going back, you again add the ids of the rooms that you visit.
- For the neighbor selection, you need to follow the order in which the rooms are defined as a neighbor for that Room. For instance, if the neighbors of Room 5 are ordered as <Room 12, Room 7, Room 9> inside the neighbor vector, then you should select Room 12 first. After completing Room 12, you should continue from Room 7 and next from Room 9. Assume that Room 7 was visited before. Then you should follow Room 9 after completing the Room 12 and its neighbors. In other words, you should skip Room 7.

- There will always be exactly one room including the treasure.

Example IO:

Please pay attention to the ordering of the neighbors for each node. It affects the resulting path!

EXAMPLE-1



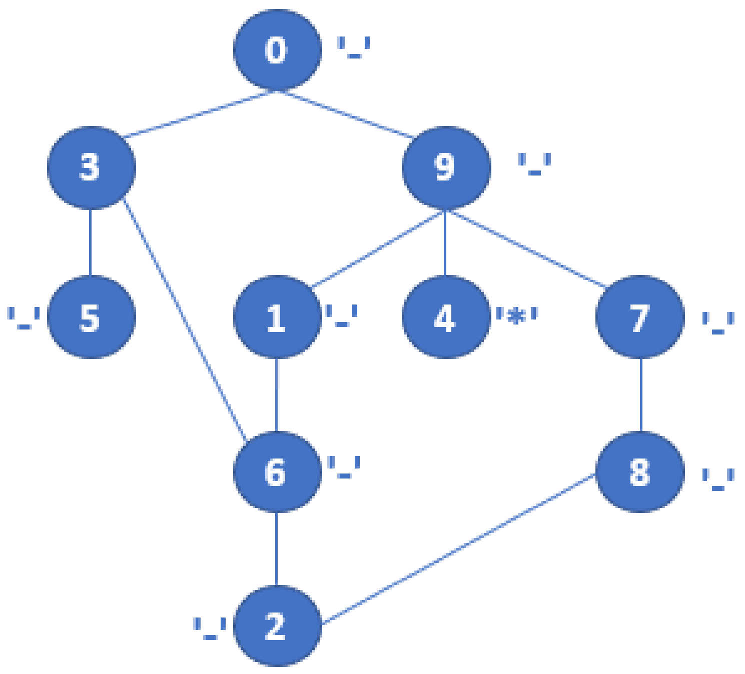
Rooms:

{id: 0, content: '-', neighbors: {3, 9}}
{id: 1, content: '-', neighbors: {6, 9}}
{id: 2, content: '-', neighbors: {6}}
{id: 3, content: '-', neighbors: {0, 5}}
{id: 4, content: '*', neighbors: {9}}
{id: 5, content: '-', neighbors: {3}}
{id: 6, content: '-', neighbors: {1, 2}}
{id: 7, content: '-', neighbors: {8, 9}}
{id: 8, content: '-', neighbors: {7}}
{id: 9, content: '-', neighbors: {0, 1, 4, 7}}

Path:

{0, 3, 5, 3, 0, 9, 1, 6, 2, 6, 1, 9, 4, 9, 0}

EXAMPLE-2



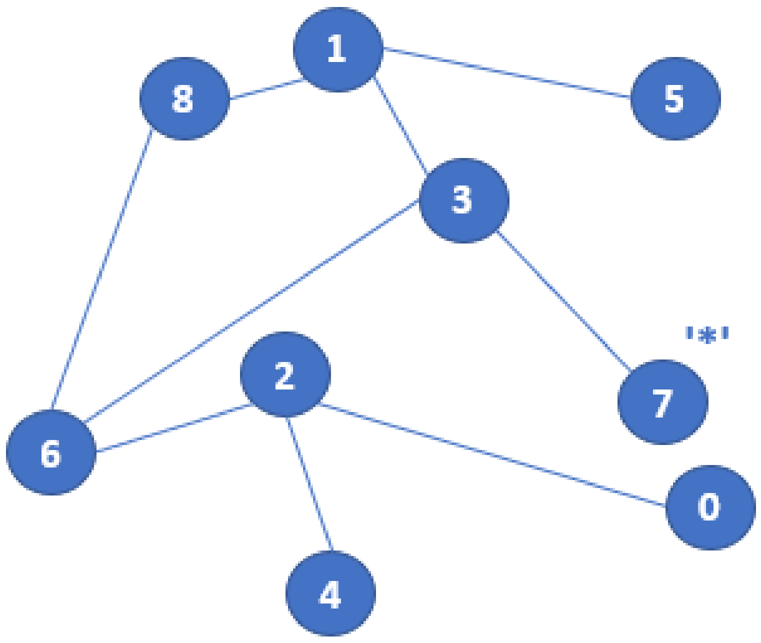
Rooms:

{id: 0, content: '-', neighbors: {3, 9}}
{id: 1, content: '-', neighbors: {6, 9}}
{id: 2, content: '-', neighbors: {6, 8}}
{id: 3, content: '-', neighbors: {0, 5, 6}}
{id: 4, content: '*', neighbors: {9}}
{id: 5, content: '-', neighbors: {3}}
{id: 6, content: '-', neighbors: {1, 2, 3}}
{id: 7, content: '-', neighbors: {8, 9}}
{id: 8, content: '-', neighbors: {2, 7}}
{id: 9, content: '-', neighbors: {0, 1, 4, 7}}

Path:

{0, 3, 5, 3, 6, 1, 9, 4, 9, 1, 6, 3, 0}

EXAMPLE-3



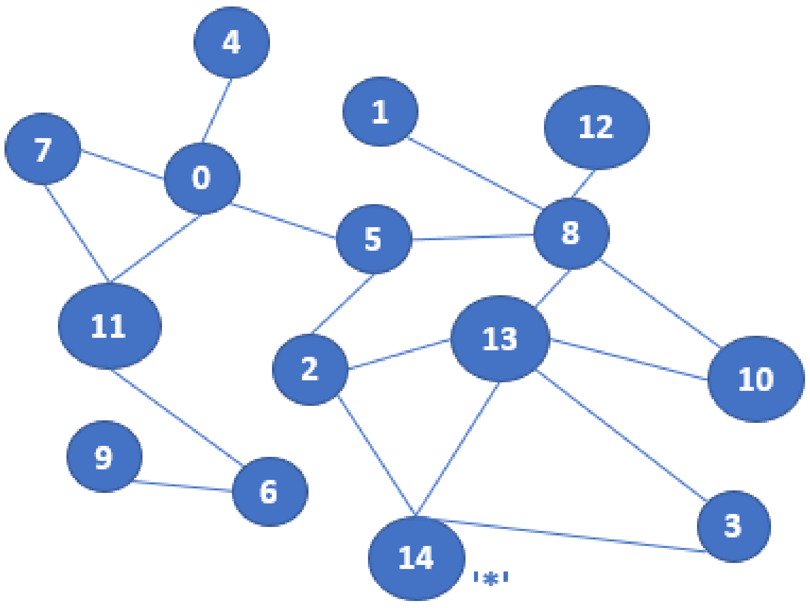
Rooms:

{id: 0, content: '-', neighbors: {2}}
{id: 1, content: '-', neighbors: {8, 5, 3}}
{id: 2, content: '-', neighbors: {6, 4, 0}}
{id: 3, content: '-', neighbors: {1, 7, 6}}
{id: 4, content: '-', neighbors: {2}}
{id: 5, content: '-', neighbors: {1}}
{id: 6, content: '-', neighbors: {8, 3, 2}}
{id: 7, content: '*', neighbors: {3}}
{id: 8, content: '-', neighbors: {1, 6}}

Path:

{**0, 2, 6, 8, 1, 5, 1, 3,** 7**, 3, 1, 8, 6, 2, 0**}

EXAMPLE-4



Rooms:

{id: 0, content: '-', neighbors: {7, 4, 11, 5}}
{id: 1, content: '-', neighbors: {8}}
{id: 2, content: '-', neighbors: {13, 5, 14}}
{id: 3, content: '-', neighbors: {14, 13}}
{id: 4, content: '-', neighbors: {0}}
{id: 5, content: '-', neighbors: {0, 8, 2}}
{id: 6, content: '-', neighbors: {9, 11}}
{id: 7, content: '-', neighbors: {11, 0}}
{id: 8, content: '-', neighbors: {10, 5, 1, 12, 13}}
{id: 9, content: '-', neighbors: {6}}
{id: 10, content: '-', neighbors: {8, 13}}
{id: 11, content: '-', neighbors: {0, 6, 7}}
{id: 12, content: '-', neighbors: {8}}
{id: 13, content: '-', neighbors: {8, 2, 3, 14, 10}}
{id: 14, content: '*', neighbors: {3, 2, 13}}

Path:

{**0, 7, 11, 6, 9, 6, 11, 7, 0, 4, 0, 5, 8, 10, 13, 2, 14, 2, 13, 10, 8, 5, 0**}