Eakawat Tantamjarik 5929444

Problem: Werewolf

Knife. Moonlit night. Rotten stump with a short black-handled knife in it. Those who know will understand. Disaster in the village. Werewolf.

There are no so many residents in the village. Many of them are each other's relatives. Only this may help to find the werewolf. The werewolf is merciless, but his descendants never become his victims. The werewolf can drown the village in blood, but he never kills his ancestors.

It is known about all the villagers who is the child of whom. Also, the sad list of the werewolf's victims is known. Your program should help to determine the suspects. It would be a hard task, if a very special condition would not hold. Namely, citizens of the village are not used to leave it. If some ancestor of some citizen lives in the village, then also his immediate ancestor does. It means, that, for example, if the father of the mother of some citizen still lives in the village, than also his mother still lives

Input

The first line contains an integer N, $1 < N \leq 1000$, which is the number of the villagers. The villagers are assigned numbers from 1 to N. Further is the description of the relation "child-parent": a sequence of lines, each of which contains two numbers separated with a space; the first number in each line is the number of a child and the second number is the number of the child's parent. The data is correct: for each of the residents there are no more than two parents, and there are no cycles. The list is followed by the word "BLOOD" written with capital letters in a separate line. After this word there is the list of the werewolf's victims, one number in each line.

Output

The output should contain the numbers of the residents who may be the werewolf. The numbers must be in the ascending order and separated with a space. If there are no suspects, the output should contain the only number 0.

Source: *http://acm.timus.ru/problem.aspx?num=1242*

Problem Description

Basically this werewolf problem can be modelled as one form of graph problem. By simulating the villager to be a node and each relation between villager can be viewed as an edge. From this idea, a village can be depicted as a graph to be more specific a directed acyclic graph (DAG) since for a family it cannot have its descendants become ancestors which means no cycle as shown in figure 1.

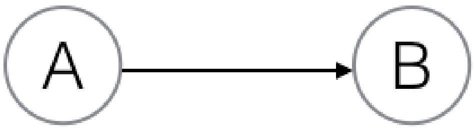


Figure 1. villager graph

In figure 1, it can be interpreted as a village which have 2 people: A and B. While B is descendants of A or A is ancestors of B and can be formally written as

Let G = {V, E} be a directed graph with V is a set of villagers and E is a relationship between each villager

By looking the werewolf condition from the problem, it said that werewolves never kill their descendants and ancestors. This means whenever someone die all their ancestors and descendants cannot be werewolf.

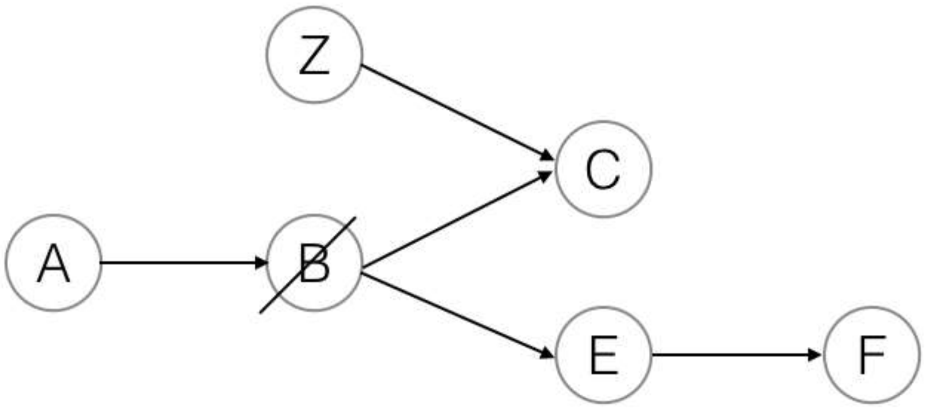


Figure 2. more complex villager graph

Figure 2 shows another more complex village graph. The cross at B node means he/she is death. From this we can know that A is not a werewolf because he is an ancestors of B and C, D, and E are also not a werewolf since they are descendants of B. On the other hand, Z is neither ancestors or descendants of B so this guy is the one suspect to be the werewolf.

Problem Model

This werewolf problem, the best representation is assumed it is a graph problem which already described in previous section. For a graph problem, there are many ways it can represent. Two common ways are Adjacancy Matrix and Adjacancy List.

However, in this problem the Adjacancy Matrix is not a good representation because the ancestors or descendants cannot be known directly once the node is known. It needs to search through all the other nodes to check whether there have an adjacent which will waste a lot of time if the number of node (number of villager) is large.

Adjacancy List is the better solution for this problem but it needs to modify to store 2 lists for each node instead of just one. One for ancestors and the other for descendants.

```
class Villager(object):
  ancestors = list()
  descendants = list()
```

From this model, whenever the villager object is known, the ancestors and descendants of that villager is also directly known.

The Algorithm

By the condition that werewolves never kill their ancestors and descendants, it can be concluded that the villagers who are not part of the ancestors or descendants of death villagers are suspect to be werewolf including his or her brothers and sisters. The easiest way to find out all ancestors is to use BFS (breath-first search) starting with the death villager and walk through the ancestors list until no ancestors left mark all the node (villager) reachable by this way as visited and do the same for descendants. After finished marking all the villagers for both ancestors and descendants from all the death villager find out all the villagers which are not visited. Those villagers are the one suspect to be werewolf.

```
WEREWOLF($N$)
    $Q = \phi$
    for each death $s \in N$
        $s.visited\_ancestor$ = true
        ENQUEUE($Q$, $s$)
    while $Q \neq \phi$
        $u$ = DEQUEUE($Q$)
        for each $v \in u.ancestors$
            if $v.visited\_ancestor$ = false
                $v.visited\_ancestor$ = true
                ENQUEUE($Q$, $v$)

    for each death $s \in N$
        $s.visited\_descendant$ = true
        ENQUEUE($Q$, $s$)
    while $Q \neq \phi$
        $u$ = DEQUEUE($Q$)
        for each $v \in u.descendants$
            if $v.visited\_descendant$ = false
                $v.visited\_descendant$ = true
                ENQUEUE($Q$, $v$)

    for each $s \in N$
        if $v.visited\_ancestor$ = false and $v.visited\_descendant$ = false
            ENQUEUE($Q$, $s$)

    $Q$ is now contains the suspect villagers
```

The algorithm WEREWOLF works as follows. It starts by adding the death villager nodes into the queue and mark those villagers as visited for ancestor. Inside the first while loop, it iterates as long as there still have ancestors. Every node visited this way will be marked to prevent visiting the same node. After the first while loop, all the ancestors of death villagers will be marked. The remaining code does the exact same things but for descendants instead. The suspect list then obtained by appending all the villager nodes which is not both ancestor and descendant (*visited_ancestor* and *visited_descendant* are false)

Algorithm in Action

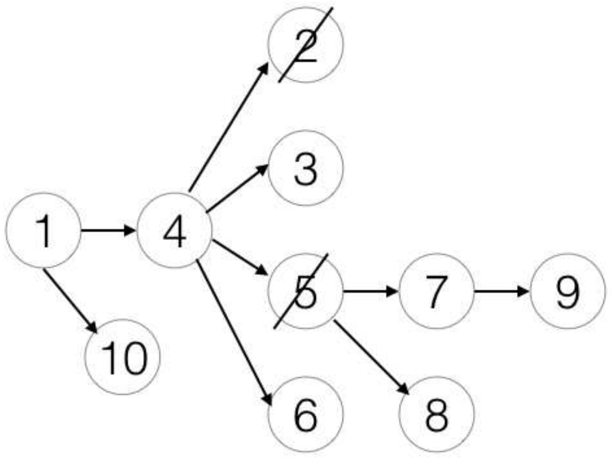
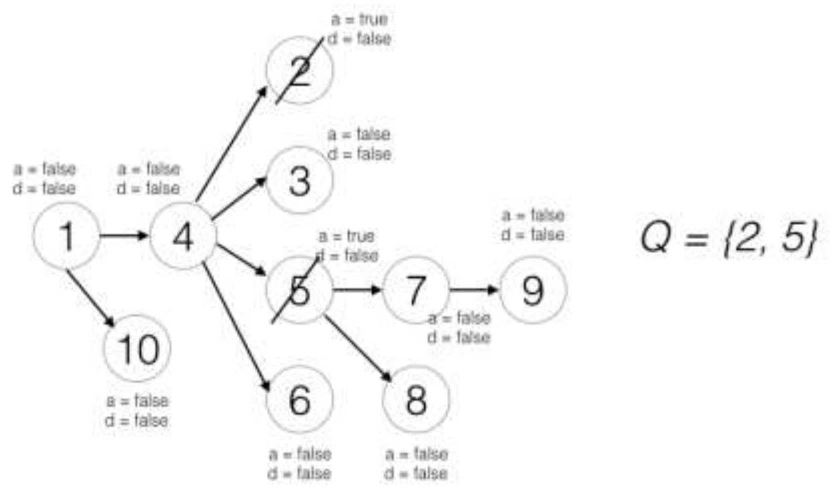


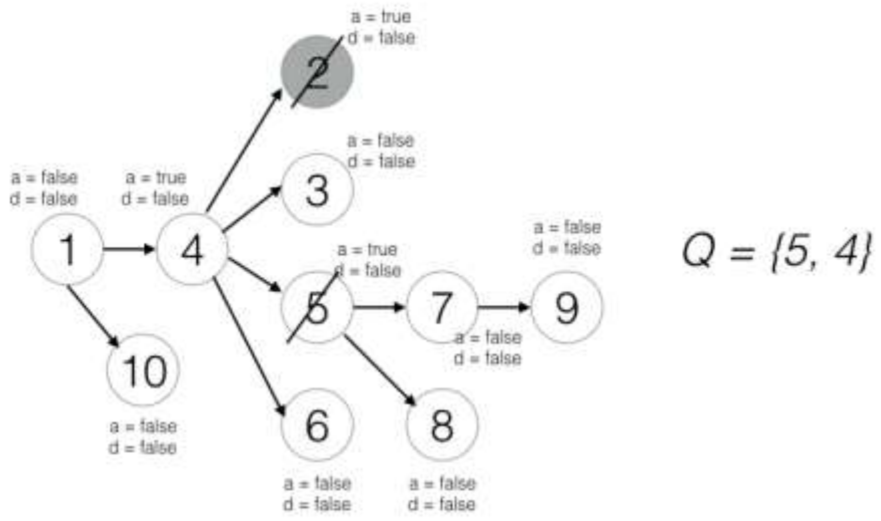
Figure 3. example village graph

The above village graph shows a village which has 10 villagers. The relationship between each villager are described using edges and villager 2 and 5 became the victim to the werewolf. The following (a) – (e) are the first part of the algorithm to find out all ancestors from death villager where *a*, *d* beside each node display the visited flag for ancestor and descendants respectively. *Q* shows the current villagers in the queue. The solid gray color means that node is already dequeue from the queue.

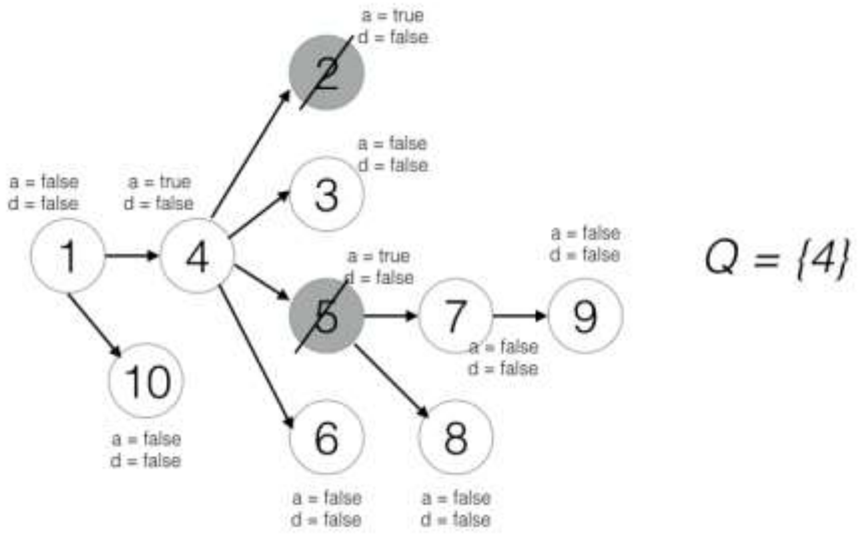
(a)


a = true
d = false
2
a = false
d = false
3
a = false
d = false
1
a = false
d = false
4
a = true
d = false
5
7
a = false
d = false
a = false
d = false
9
10
a = false
d = false
6
a = false
d = false
8
a = false
d = false
Q = {2, 5}

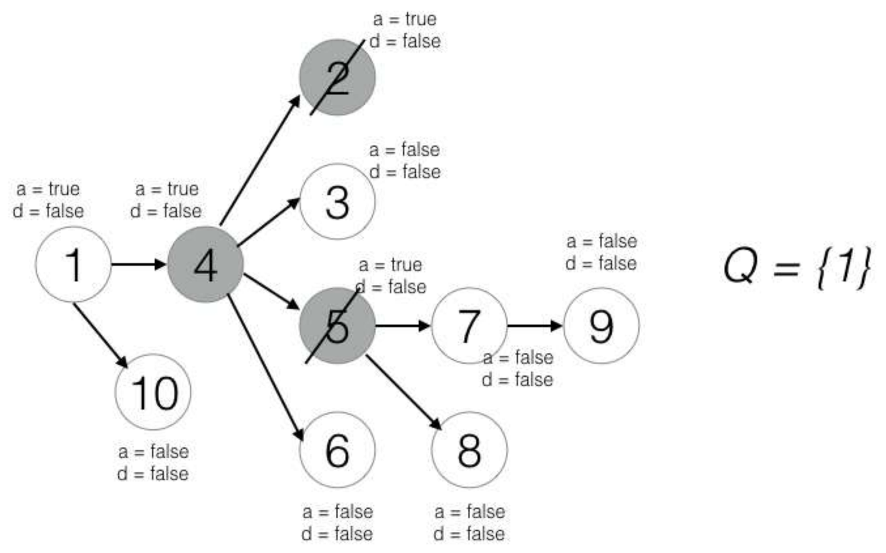
(b)


a = true
d = false
2
a = false
d = false
3
a = false
d = false
1
a = true
d = false
4
a = true
d = false
5
7
a = false
d = false
a = false
d = false
9
10
a = false
d = false
6
a = false
d = false
8
a = false
d = false
Q = {5, 4}

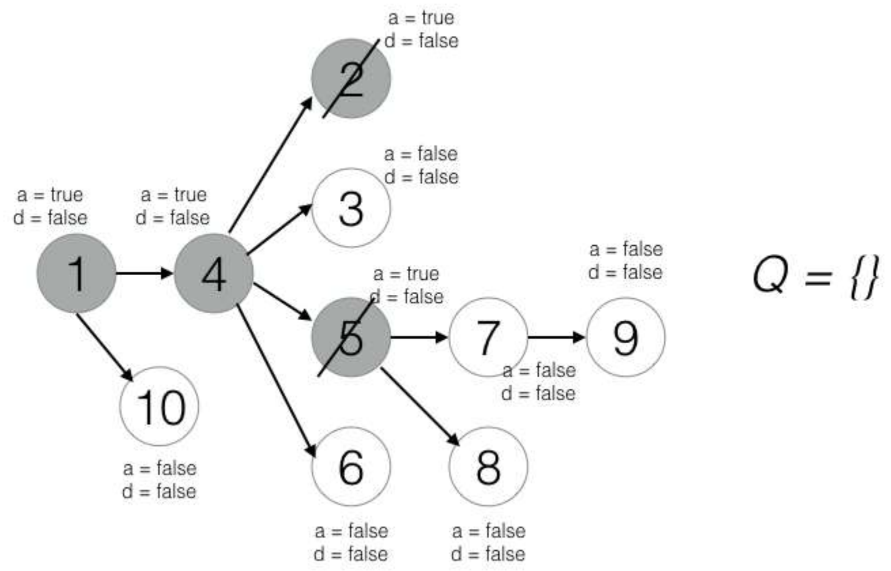
(c)


a = true
d = false
2
a = false
d = false
3
a = false
d = false
1
a = true
d = false
4
a = true
d = false
5
7
a = false
d = false
a = false
d = false
9
10
a = false
d = false
6
a = false
d = false
8
a = false
d = false
Q = {4}

(d)

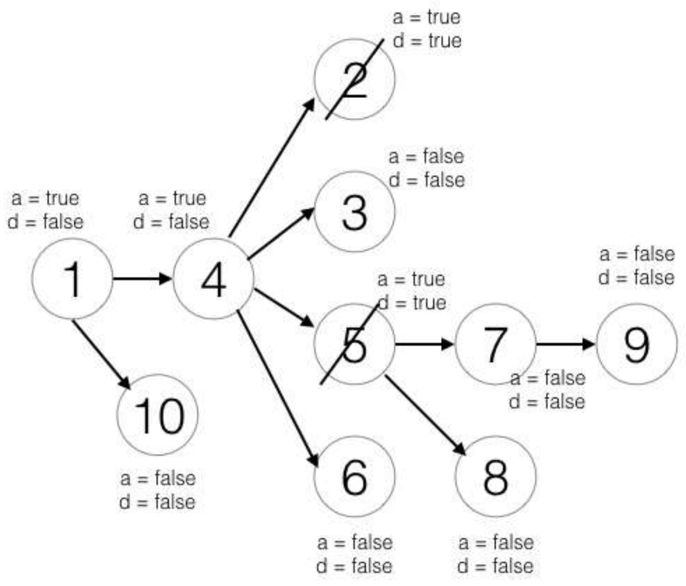


(e)



(a) is the start state for this problem. The algorithm adds the death node into the queue and marks visit ancestor. (b) from the relationship between villager 4 and 2, it knows that 4 is ancestor of 2 so adds node 4 to the queue and mark node 4 as visit ancestor. (c) villager 4 also happens to be ancestor of villager 5 but it is already marked by step (b) thus only dequeue 5. (d) follow villager 4, it knows that villager 1 is ancestor of 4 so adds that to queue and marks as visit ancestor. (e) no more ancestor, hence it finishes the ancestor part.

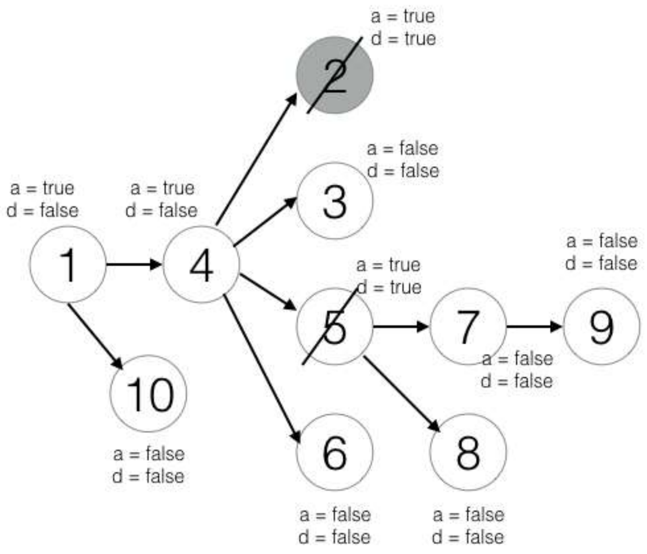
(f)



a = true
d = true
2
a = false
d = false
3
a = true
d = false
1
a = true
d = false
4
a = true
d = true
5
7
a = false
d = false
9
a = false
d = false
10
a = false
d = false
6
a = false
d = false
8
a = false
d = false

$Q = \{2, 5\}$

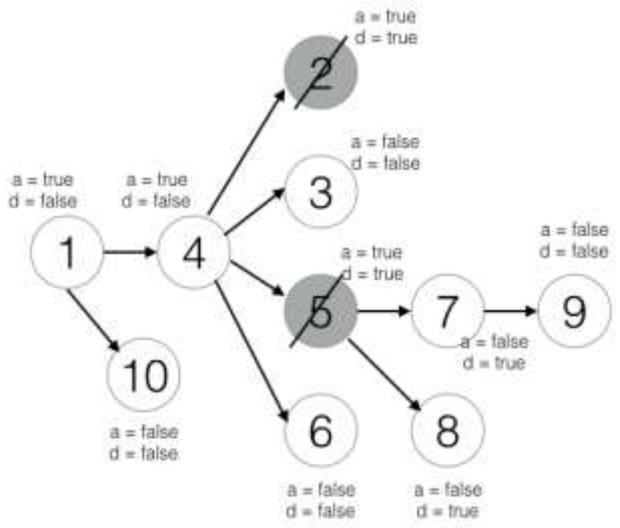
(g)



a = true
d = true
2
a = false
d = false
3
a = true
d = false
1
a = true
d = false
4
a = true
d = true
5
7
a = false
d = false
9
a = false
d = false
10
a = false
d = false
6
a = false
d = false
8
a = false
d = false

$Q = \{5\}$

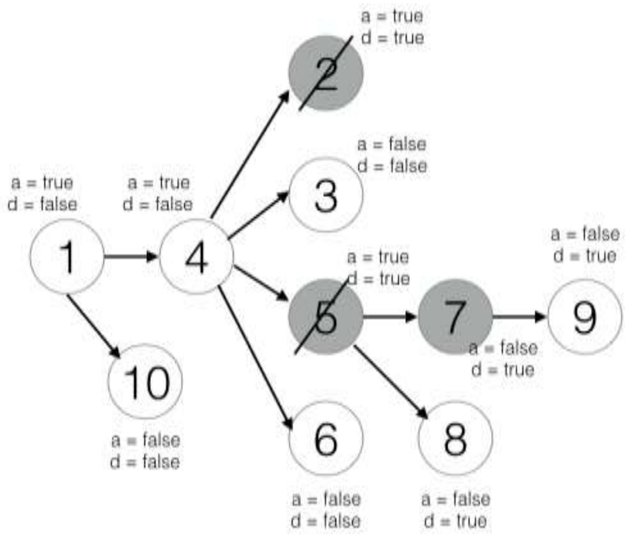
(h)



1
a = true
d = false
4
a = true
d = false
2
a = true
d = true
3
a = false
d = false
5
a = true
d = true
7
a = false
d = true
9
a = false
d = false
10
a = false
d = false
6
a = false
d = false
8
a = false
d = true

$Q = \{7, 8\}$

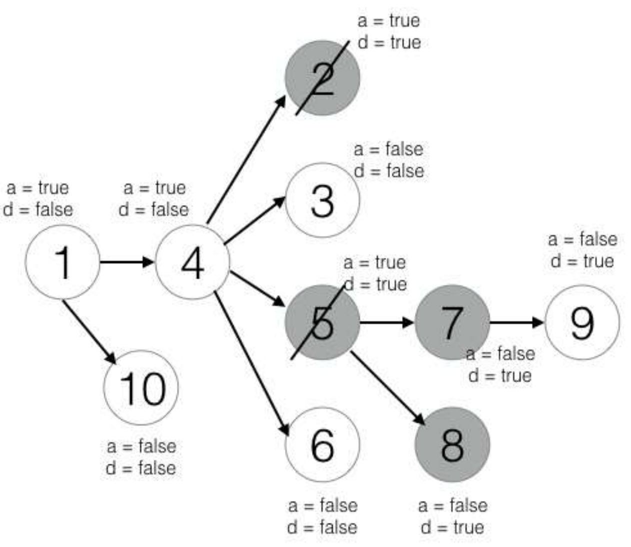
(i)



1
a = true
d = false
4
a = true
d = false
2
a = true
d = true
3
a = false
d = false
5
a = true
d = true
7
a = false
d = true
9
a = false
d = true
10
a = false
d = false
6
a = false
d = false
8
a = false
d = true

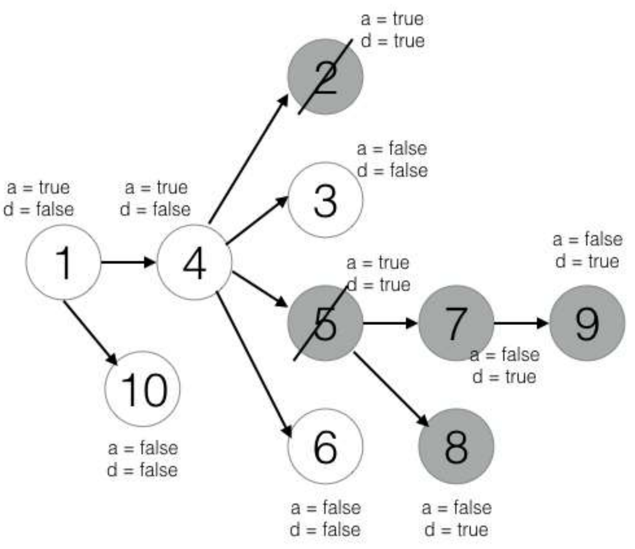
$Q = \{8, 9\}$

(j)



$Q = \{9\}$

(k)



$Q = \{\}$

For descendant part, it starts with the same way as ancestor part by add death nodes into queue to be a start state but marks visit descendant instead of ancestor (f). In (g) the algorithm only dequeue node 2 and does nothing because villager 2 does not have any descendant. From (h) because villager 5 have 2 descendants, it adds both 7, 8 to the queue and marks both as visit descendant. (i) works the same way as when finding ancestor (add to queue and mark visit descendant). (j) does nothing because no descendant. (k) no more descendant thus the algorithm is done marking ancestor and descendant. The last step is finding the node which haven't visited by both ancestor and descendant (node which have both *a* and *d* equal false). Therefore, villager 3, 6, and 10 are suspect to be werewolf.

Analysis

After the algorithm starts, it does almost identical things 2 times one for ancestor and another one for descendant so all the running time should be multiply by 2 but since the notation for Big-O for O(2n) is equivalent with O(n) thus, it can be removed to just analyze the first part for ancestor. First it loops through all the death villager list for resetting ancestor flag since the number of death villager can be as much as the number of villager thus O(V) for loop all death villager list. In the **while** loop the vertex will never be repeated because it can be added to the queue only once while the value of ancestor flag is false so at most the **while** loop will run only up to the number of villager which is O(V). Inside the **while** loop, the algorithm iterates through all the ancestor list for that particular vertex only once because the vertex that got dequeued will never get enqueued again. Thus, this **for** loop will run at most equal to the number of all the edges which is O(E). In the end the werewolf algorithm running time is O(V + V + E) -> O(V + E)

Proof of Correctness for The Algorithm

It is known that for a villager to be suspected as werewolf, the list of normal villagers who are sure to not be werewolf need to be found first and the suspected werewolf will be easily found by concluded that suspect werewolves are not the villager in normal villager list. As state by the rule of the problem, these people can be either ancestor, descendant or both to the death villager. Thus, it can be claimed that normal villagers as a node in the graph need to be visited at least once either ancestor or descendant. The starting point to trace back this clue should have to start from death villager because the problem provides all the relationship, from that death villagers all his/her ancestor and descendant can be traced.

1. **Claim** All normal villagers need to be visited at least once – Since the starting villager in the queue is from death villagers ENQUEUE(Q, v) will called exactly once for either ancestor or descendant only when *v.visited_ancestor* or *v.visited_descendant* is false otherwise the condition will be ignored. This ensure that villager will add to the queue exactly once for either ancestor and descendant.
2. **Claim** All ancestors and descendants of death villager can be reached from death villager – Since the algorithm always adding vertex to the queue and those vertices come from the list of ancestors and descendants which means all those vertices are ancestor or descendant of the starting vertex which is death villager. When the queue is

empty that means the algorithm can no longer find either ancestor or descendant anymore. Hence, all the ancestors or descendants are already found. it can be concluded that way because the number of villager is finite and the process for adding vertex into queue will not repeat the with the same vertex for the same part (ancestor or descendant) (claim 1).