What link causes the loop?

Problem Statement

A loop in a linked list occurs when following the linked list chain brings you back where you started. For example, link 1 points to link 2 which points to link 3 which points to link 1, to link 2, to link 3, to link 1...and you go round in an infinite loop. Each test case here involves a single-ended singly-linked list with a potential loop. Output the data of the link whose pointer has caused the linked list to loop back to a previous point. In the above example, it would be whatever data is stored in link 3.

Input Format

There is no input. The linked list is created automatically.

Output Format

Output the String stored in the link which causes the list to loop back to a previous link. If there is no loop output "OK". If the linked list is empty output "empty".

Constraints

0<=#(links)<=100

Building your own test case

You might want to build your own test case and in that case, here's how you do it.

The first line is the number of links n.

The next n lines are the Strings held in each link.

The next lines give the connections between the links. Each of these lines consists of 2 numbers separated by a space. The first is which number link you are talking about, the second is the next link it points to. A -1 is used for null.

Here is an example of a perfect linked list with 4 links:

4

Data1

Data2

Data3

Data4

0 1

1 2

2 3

3 -1

```
import java.util.*;

public class Solution {
    public static void main(String args[] ) throws Exception {
        Scanner myscanner = new Scanner(System.in);
        int num = Integer.parseInt(myscanner.nextLine());
        Link[] array = new Link[num];
        for(int i=0;i<num;i++){
            array[i]=new Link(myscanner.nextLine());
        }
        while(myscanner.hasNext()){
            int select=myscanner.nextInt();
            int next=myscanner.nextInt();
            if(next!=-1){
                array[select].next=array[next];
            }
        }
        LinkedList mylist = new LinkedList();
        if(num>0){
            mylist.first=array[0];
        }
        System.out.println(findloop(mylist));
    }

    public static String findloop(LinkedList mylist){
        if(mylist.isEmpty()){
            return("empty");
        }
        Link[] checklist = new Link[100];
        int counter=0;
        Link forwards = mylist.first;
        while(forwards.next!=null){
            checklist[counter]=forwards;
            Link temp=forwards;
            forwards=forwards.next;
            for(int i=0;i<counter;i++){
                if(forwards==checklist[i]){
                    return(temp.data);
```

```
                }
            }
            counter++;
        }
        return("OK");
    }
}

class Link{
     public String data;
     public Link next;

    public Link(String input){
        data=input;
        next=null;
    }
}

class LinkedList {
     public Link first;

     public LinkedList( ){
          first=null;
     }

     public boolean isEmpty( ){
          return (first==null);
     }

    public void insertHead(Link insert){
        if(isEmpty()){
            first=insert;
        }else{
            insert.next=first;
            first=insert;
        }
    }
}
```

How long is the linked list loop?

Problem Statement

A loop in a linked list occurs when following the linked list chain brings you back where you started. For example, link 1 points to link 2 which points to link 3 which points to link 1, to link 2, to link 3, to link 1...and you go round in an infinite loop. Each test case here involves a single-ended singly-linked list with a potential loop. Output the size of the loop, or 0 if there is no loop. In the above example, the loop is of size 3 (e.g. link 1, link 2, link 3...repeating forever).

Input Format

There is no input. The linked list is created automatically.

Output Format

Output an integer, either 0 if the linked list is empty or has no loop, or else >0 corresponding to the length of the loop in the linked list.

Constraints

0<=#(links)<=100

Building your own test case

You might want to build your own test case and in that case, here's how you do it.

The first line is the number of links n.

The next n lines are the Strings held in each link.

The next lines give the connections between the links. Each of these lines consists of 2 numbers separated by a space. The first is which number link you are talking about, the second is the next link it points to. A -1 is used for null.

Here is an example of a perfect linked list with 4 links:

4

Data1

Data2

Data3

Data4

0 1

1 2

2 3

3 -1

```
import java.util.*;

public class Solution {
    public static void main(String args[] ) throws Exception {
        Scanner myscanner = new Scanner(System.in);
        int num = Integer.parseInt(myscanner.nextLine());
        Link[] array = new Link[num];
        for(int i=0;i<num;i++){
            array[i]=new Link(myscanner.nextLine());
        }
        while(myscanner.hasNext()){
            int select=myscanner.nextInt();
            int next=myscanner.nextInt();
            if(next!=-1){
                array[select].next=array[next];
            }
        }
        LinkedList mylist = new LinkedList();
        if(num>0){
            mylist.first=array[0];
        }
        System.out.println(findLoopLength(mylist));
    }

    public static int findLoopLength(LinkedList mylist){
        if(mylist.isEmpty()){
            return(0);
        }
        Link[] checklist = new Link[100];
        int counter=0;
        Link forwards = mylist.first;
        while(forwards.next!=null){
            checklist[counter]=forwards;
            for(int i=0;i<counter;i++){
                if(forwards==checklist[i]){
                    return(counter-i);
                }
            }
            forwards=forwards.next;
            counter++;
        }
        return(0);
    }
}

class Link{
     public String data;
```

```
      public Link next;

    public Link(String input){
        data=input;
        next=null;
    }
}

class LinkedList {
      public Link first;

      public LinkedList( ){
            first=null;
      }

      public boolean isEmpty( ){
            return (first==null);
      }

    public void insertHead(Link insert){
        if(isEmpty()){
            first=insert;
        }else{
            insert.next=first;
            first=insert;
        }
    }
}
```

O(n) solutions

The above solutions are inefficient because they are $O(n^2)$ – as the number of links in the linked list increases, you have to check over a fraction of an array of size *n* a fraction of *n* times. There's a better way to do it using Floyd's cycle detection algorithm (aka the tortoise and hare algorithm), and Eoin Davey (CT) has kindly coded it up. In brief, the idea is that two pointers move at different speeds through the links until they both point to equal values. For example, the first pointer moves twice as fast as the second pointer. If there's a loop, eventually the faster one will catch up behind the slower one, and it's O(*n*), since the slower one has only moved *n* steps.

Loop causing link (Eoin Davey)

```
public static String findloop(LinkedList)
      // Floyd's cycle finding algorithm O(n)
      if(mylist.isEmpty())return "empty";
      Link t = mylist.first;
      if(t.next==null)return "OK";
      if(t.next.next==null)return "OK";
      t = t.next;
      Link h = t.next;
      // Check if cycle exists
      boolean cycleExists = true;
```

```
      while(t!=h){
            if(t.next==null||h.next==null){
                  cycleExists=false;
                  break;
            }
            t = t.next;
            if(h.next.next==null){
                  cycleExists = false;
                  break;
            }
            h = h.next.next;
      }
      if(!cycleExists)return "OK";
      //find first element of loop
      int mu = 0;
      t = mylist.first;
      while(t!=h){
            t = t.next;
            h = h.next;
            mu++;
      }
      //loop to find last element
      while(t.next!=h){
            t = t.next;
      }
      return t.data;
}
```

Loop length (Eoin Davey)

```
public static int findlooplength(LinkedList mylist){
    // Floyd's cycle finding algorithm O(n)
    if(mylist.isEmpty())return 0;
    Link t = mylist.first;
    if(t.next==null)return 0;
    if(t.next.next==null)return 0;
    t = t.next;
    Link h = t.next;
    // Check if cycle exists
    boolean cycleExists = true;
    while(t!=h){
        if(t.next==null||h.next==null){
            cycleExists=false;
            break;
        }
        t = t.next;
        if(h.next.next==null){
            cycleExists = false;
            break;
        }
        h = h.next.next;
    }
    if(!cycleExists)return 0;
    //find first element of loop
    int mu = 0;
```

```
    t = mylist.first;
    while(t!=h){
        t = t.next;
        h = h.next;
        mu++;
    }
    //loop to find length
    int len = 1;
    t = t.next;
    while(t!=h){
        t = t.next;
        len++;
    }
    return len;
}
```