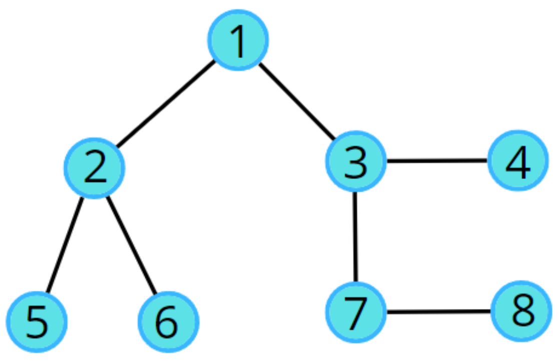
August 29, 2022 ▪ Arrays / Data Structure / Graph

Detect Cycle in an Undirected Graph (using DFS)

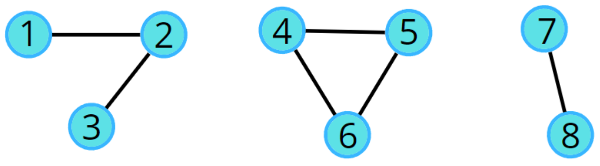
Problem Statement: Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

Examples:

```
Example 1:
Input:
V = 8, E = 7
```



```
Output: No

Explanation: No cycle in the given graph.
```



```
Output: Yes

Explanation:
4->5->6->4 is a cycle.
```

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Intuition:

The cycle in a graph starts from a node and ends at the same node. DFS is a traversal technique that involves the idea of recursion and backtracking. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes. The intuition is that we start from a source and go in-depth, and reach any node that has been previously visited in the past; it means there's a cycle.

Approach:

The algorithm steps are as follows:

- In the DFS function call make sure to store the parent data along with the source node, create a visited array, and initialize to 0. The parent is stored so that while checking for re-visited nodes, we don't check for parents.

Recent Posts

Print Nodes at Distance K in a Binary Tree

LCA in Binary Search Tree

Check if a tree is a Binary Search Tree or Binary Tree

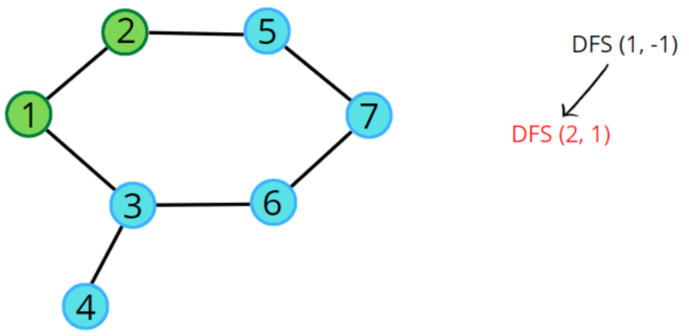
Delete a Node in Binary Search Tree

Insert a Given Node in Binary Search Tree

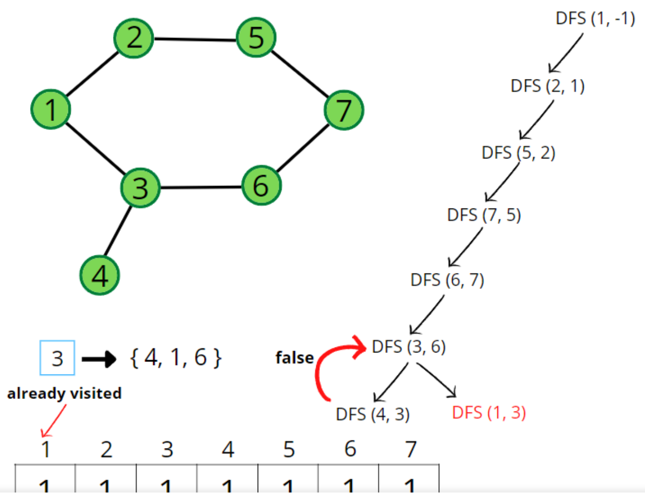
- If we come across a node that is already marked as visited and is ***not a parent node***, then keep on returning true indicating the presence of a cycle; otherwise return false after all the adjacent nodes have been checked and we did not find a cycle.

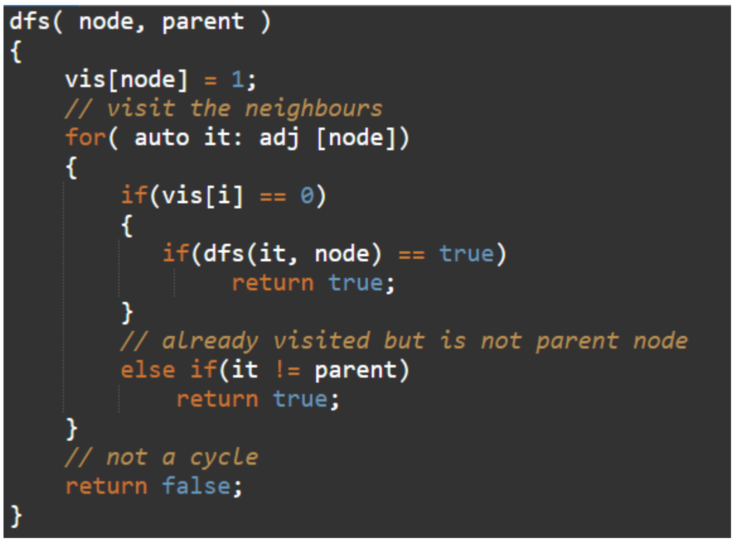
NOTE: We can call it a cycle only if the already visited node is a non-parent node because we cannot say we came to a node that was previously the parent node.

For example, node 2 has two adjacent nodes 1 and 5. 1 is already visited but it is the parent node (DFS(2, **1**)), So this cannot be called a cycle.



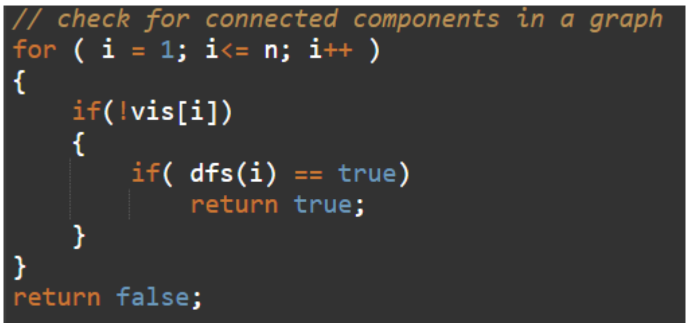
Node 3 has three adjacent nodes, where 4 and 6 are already visited but node 1 is not visited by node 3, but it's already marked as visited and is a non-parent node (DFS(3, **6**)), indicating the presence of cycle.





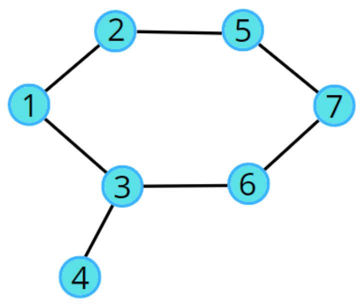
```
dfs( node, parent )
{
    vis[node] = 1;
    // visit the neighbours
    for( auto it: adj [node])
    {
        if(vis[i] == 0)
        {
           if(dfs(it, node) == true)
                return true;
        }
        // already visited but is not parent node
        else if(it != parent)
            return true;
    }
    // not a cycle
    return false;
}
```

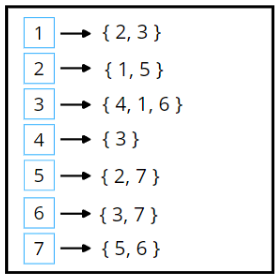
A graph can have connected components as well. In such cases, if any component forms a cycle then the graph is said to have a cycle. We can follow the algorithm for the same:



```
// check for connected components in a graph
for ( i = 1; i<= n; i++ )
{
    if(!vis[i])
    {
        if( dfs(i) == true)
            return true;
    }
}
return false;
```

Consider the following graph and its adjacency list.

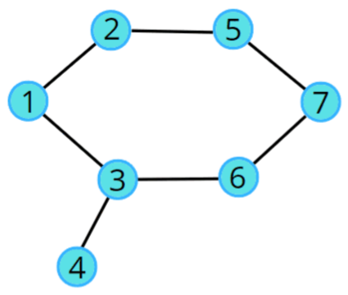




1	{ 2, 3 }
2	{ 1, 5 }
3	{ 4, 1, 6 }
4	{ 3 }
5	{ 2, 7 }
6	{ 3, 7 }
7	{ 5, 6 }

Adjacency List

Consider the following illustration to understand the process of detecting a cycle using DFS traversal.



1	2	3	4	5	6	7
0	0	0	0	0	0	0

Visited Array

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
  private:
    bool dfs(int node, int parent, int vis[],
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node]) {
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
                if(dfs(adjacentNode, node, vi
                    return true;
            }
            // visited node but not a parent
            else if(adjacentNode != parent) r
        }
        return false;
    }
  public:
    // Function to detect cycle in an undirec
    bool isCycle(int V, vector<int> adj[]) {
       int vis[V] = {0};
       // for graph with connected components
       for(int i = 0;i<V;i++) {
           if(!vis[i]) {
               if(dfs(i, -1, vis, adj) == tru
           }
```

```
int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}
```

Output: 0

Time Complexity: O(N + 2E) + O(N), Where N = Nodes, 2E is for total degrees as we traverse all adjacent nodes. In the case of connected components of a graph, it will take another O(N) time.

Space Complexity: O(N) + O(N) ~ O(N), Space for recursive stack space and visited array.

Java Code

```
import java.util.*;

class Solution {
    private boolean dfs(int node, int parent,
    adj) {
        vis[node] = 1;
        // go to all adjacent nodes
        for(int adjacentNode: adj.get(node))
            if(vis[adjacentNode]==0) {
                if(dfs(adjacentNode, node, vi
                    return true;
            }
            // if adjacent node is visited ar
            else if(adjacentNode != parent) r
        }
        return false;
    }
    // Function to detect cycle in an undirec
    public boolean isCycle(int V, ArrayList<A
        int vis[] = new int[V];
```

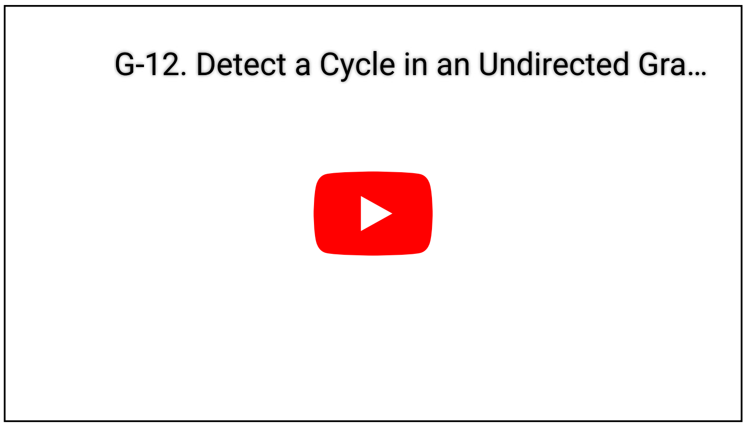
```
            return false;
        }
        public static void main(String[] args)
        {
            ArrayList<ArrayList<Integer>> adj = r
            for (int i = 0; i < 4; i++) {
                adj.add(new ArrayList < > ());
            }
            adj.get(1).add(2);
            adj.get(2).add(1);
            adj.get(2).add(3);
            adj.get(3).add(2);

            Solution obj = new Solution();
            boolean ans = obj.isCycle(4, adj);
            if (ans)
                System.out.println("1");
            else
                System.out.println("0");
        }

    }
```

Output: 0

Time Complexity: O(N + 2E) + O(N), Where N = Nodes, 2E is for total degrees as we traverse all adjacent nodes. In the case of connected components of a graph, it will take another O(N) time.

Space Complexity: O(N) + O(N) ~ O(N), Space for recursive stack space and visited array.

Special thanks to **Vanshika Singh Gour** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



« Previous Post
Detect Cycle in an Undirected Graph (using BFS)

Next Post »
Palindromic substrings

Load Comments