**Name: Md Tariqul Islam**

**Id: 191-15-12963**

**Sec: 0-14**

**Full Tree Traversal**

Traversing a tree means visiting every node in the tree. Because, all nodes are connected with edges. We always start from root/head node.

Tree Traversal are two types,

1. BFS
2. DFS

BFS has on type traversal which is Level order.

DFS are three types which are

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

**Cycle Finding**

Cycle detection or cycle finding is the algorithmic problem of finding a cycle in a sequence of iterated function values. ... Floyd's tortoise and hare algorithm moves two pointers at different speeds through the sequence of values until they both point to equal values.

## Detect Cycle in a Directed Graph using BFS:

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue

## Time complexity:

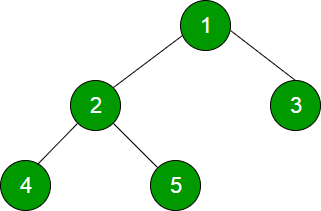
The outer for loop will be executed V number of times and the inner for loop will be executed E number of times, Thus overall time complexity is O(V+E).

**Component Finding**

**Breadth First Search**

Breadth-first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures. The full form of BFS is the Breadth-first search.

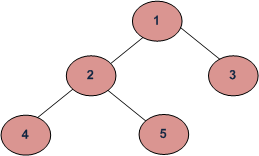
The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Remember, BFS accesses these nodes one by one.



**Output is 1 2 3 4 5**

**Depth First Search**

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.



Depth First Traversals:

1. Inorder (Left, Root, Right): 4 2 5 1 3
2. Preorder (Root, Left, Right): 1 2 4 5 3
3. Postorder (Left, Right, Root): 4 5 2 3 1

**Algorithms Inorder (Tree):**

1. Traversal the left subtree.
2. Visit the root.
3. Traversal the right subtree.

**Algorithm Preorder (Tree):**

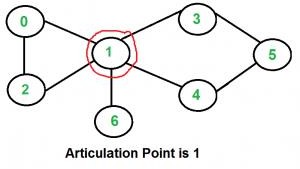
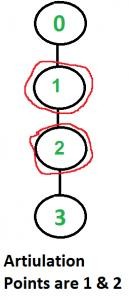
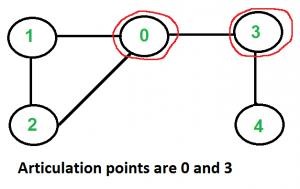
1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right tree.

**Algorithm Postorder (Tree):**

1. Traverse the left tree.
2. Traverse the right tree.
3. Visit the root.

**Articulation Point Finding:**

A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are useful for designing reliable networks. For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components. Following are some example graphs with articulation points encircled with red color.

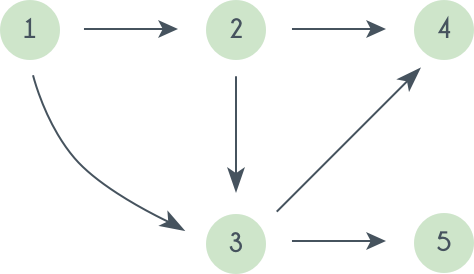


A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

1. For every vertex V;
2. Remove v from graph.
3. See if the graph remains connected (We can either use BFS or DFS).
4. Add v back to the graph.

**Topological sort**

Topological sorting of vertices of a Directed Acyclic Graph is an ordering of the vertices v1, v2,... vn in such a way, that if there is an edge directed towards vertex vj from vertex vi, then vi comes before vj. For example consider the graph given below:



A topological sorting of this graph is: 1 2 3 4 5  
There are multiple topological sorting possible for a graph. For the graph given above one another topological sorting is: 1 2 3 5 4  
In order to have a topological sorting the graph must not contain any cycles. In order to prove it, let's assume there is a cycle made of the vertices v1, v2, v3...vn. That means there is a directed edge between vi and vi+1 (1≤i<n) and between vn and v1. So now, if we do topological sorting then vn must come before v1 because of the directed edge from vn to v1. Clearly, vi+1 will come after vi, because of the directed from vi to vi+1, that means v1 must come before vn. Well, clearly we've reached a contradiction, here. So topological sorting can be achieved for only directed and acyclic graphs.

Let’s see how we can find a topological sorting in a graph. So basically we want to find a permutation of the vertices in which for every vertex vi, all the vertices vj having edges coming out and directed towards vi comes before vi. We'll maintain an array T that will denote our topological sorting. So, let's say for a graph having N vertices, we have an array in\_degree[] of size N whose ith element tells the number of vertices which are not already inserted in T and there is an edge from them incident on vertex numbered i. We'll append vertices vi to the array T, and when we do that we'll decrease the value of in\_degree[vj] by 1 for every edge from vi to vj. Doing this will mean that we have inserted one vertex having edge directed towards vj. So at any point we can insert only those vertices for which the value of in\_degree[] is 0.