Full Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too.Because all nodes are connected via edges.We always start from the root node.

BFS DFS

Level order

Traversal Inorder

Traversal Preorder

Traversal Postorder

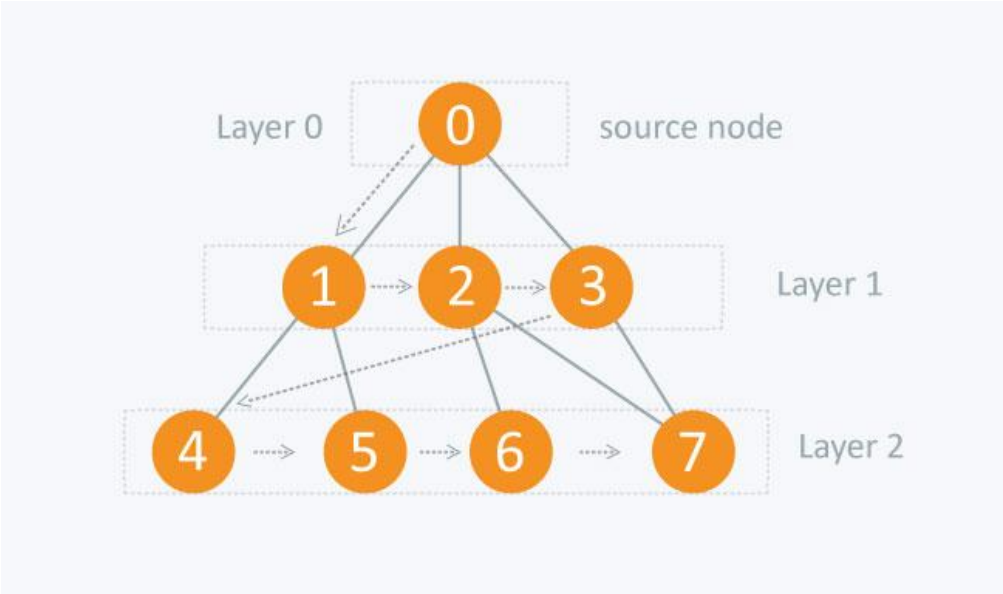
Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer



Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

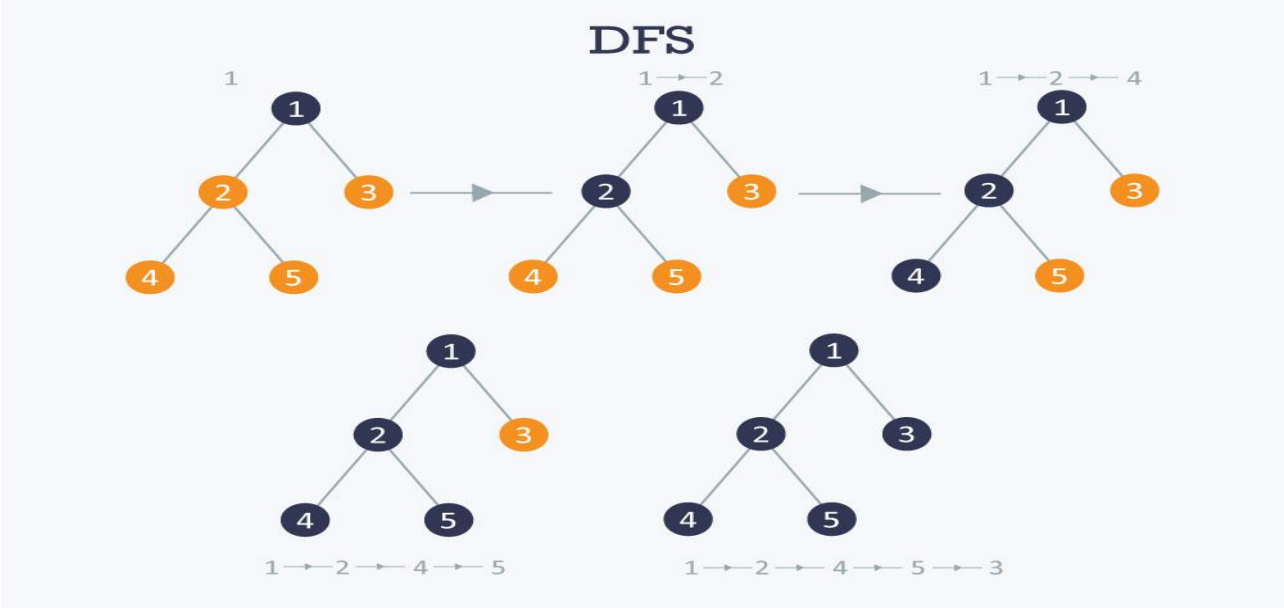
Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

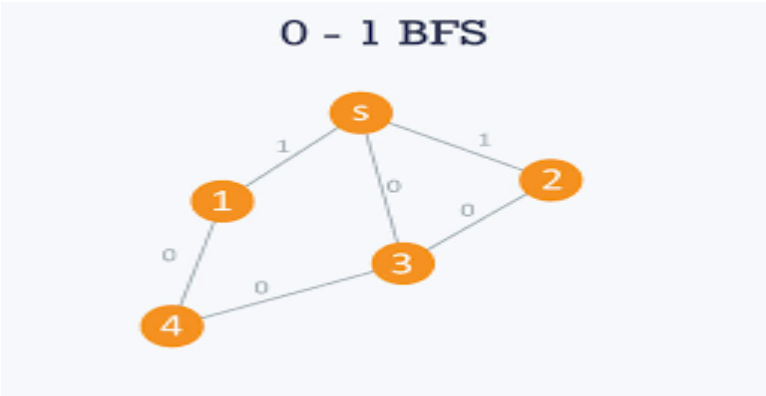


Depth First Traversals:
(a) Inorder (Left, Root, Right): 4 2 5 1 3
(b) Preorder (Root, Left, Right): 1 2 4 5 3
(c) Postorder (Left, Right, Root): 4 5 2 3 1

Algorithm Inorder(tree):
1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.
Algorithm Preorder(tree):
1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.
Algorithm Postorder(tree):
1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.
#Cycle Finding:
Like directed graphs, we can use DFS to detect cycle in an undirected graph in
O(V+E) time. ... We do BFS traversal of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph.

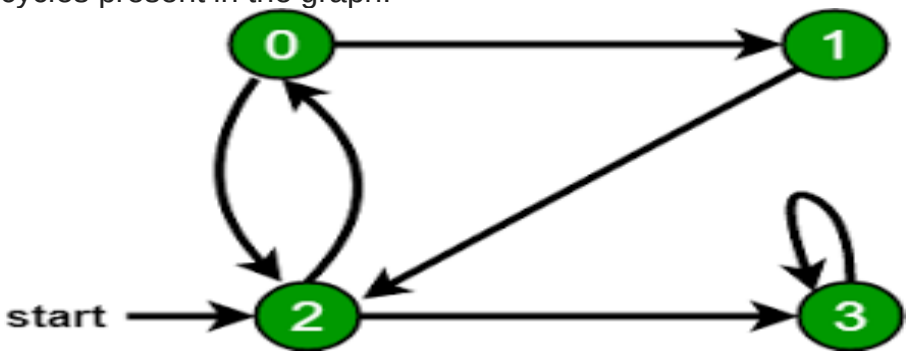
Detect Cycle in a Directed Graph using BFS:

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.



we do a BFS traversal of the given graph. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle.
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).
Detect Cycle in a Directed Graph using DFS:
Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected

graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.

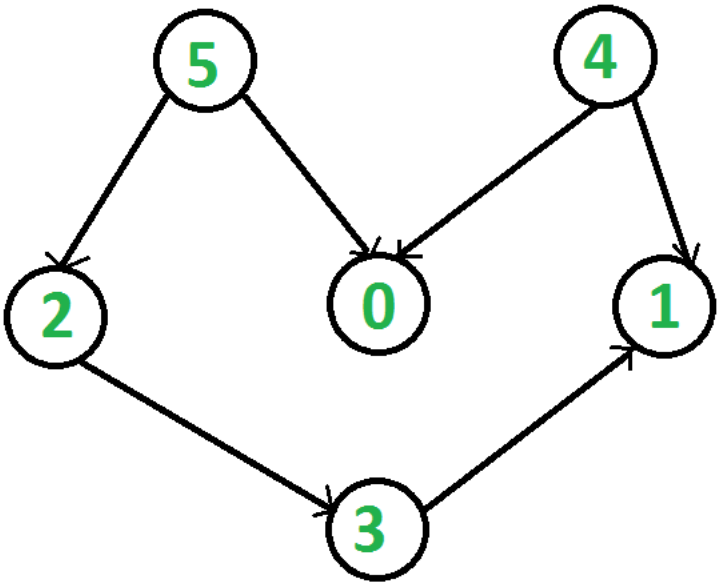


Time complexity:
A graph without cycles has at most |V| - 1 edges (it's a forest). Therefore, if the DFS discovers |V| edges or more then it already found a cycle and terminates. The runtime is accordingly bounded by O(|V|)

Topological sort

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS here. We can modify DFS to find Topological Sorting of a graph. IN DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.
Time Complexity: O(V+E).
The above algorithm is simply DFS with an extra stack. So time complexity is the same as DFS which is.
Note:
Here, we can also use vector instead of stack. If the vector is used then print the elements in reverse order to get the topological sorting.