# Non Linear Data Structures

- A non-linear data structure is another important type in which data elements are not arranged sequentially; mainly, data elements are arranged in random order without forming a linear structure.
- Data elements are present at the multilevel, for example, tree.
- In trees, the data elements are arranged in the hierarchical form, whereas in graphs, the data elements are arranged in random order, using the edges and vertex.
- Multiple runs are required to traverse through all the elements completely. Traversing in a single run is impossible to traverse the whole data structure.
- Each element can have multiple paths to reach another element.
- The data structure where data items are not organized sequentially is called a non-linear data structure

# Tree

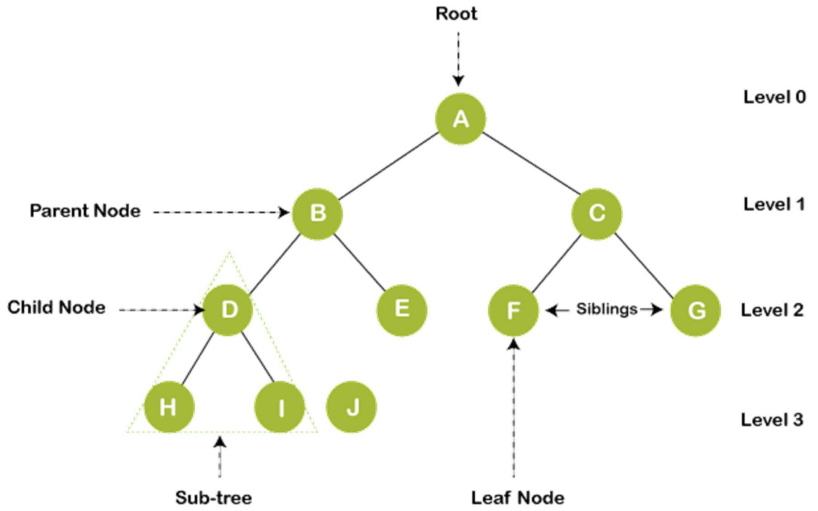
A tree is a Non-Linear Data Structure that is an abstract model of a hierarchical structure consisting of nodes with a parent-child relation. Its applications are Organization charts, File systems, Programming environments. There are four things associated with any tree -Distinction between nodes, Value of nodes, orientation structure and the number of levels.

## Tree Terminology

- **Root**– starting point of the tree is a node called root characterized by a node without parent.
- **External/leaf node** : A node without any children
- **Internal node**: All the nodes other than the root and leaf nodes. It can be said to be a node with at least one child.
- **Ancestors of a node**: parent, grandparent and any other nodes which lie on the path from root to that node.
- **Depth of a node**: number of ancestors for any given node.
- **Height of a tree**: maximum depth among all the leaf nodes.
- **Descendant of a node**: child, grandchild and all other nodes lying on path from a node to a leaf node.
- **Subtree**: tree consisting of a node and its descendants








## Binary Tree

Binary Tree is a tree for which each internal node is allowed to have at most two child nodes. The children of an internal node are called left child and right child depending upon the relationship with the parent. The recursive definition of a binary tree is that it is either a tree consisting of a single node, or a tree whose root has an ordered pair of children, each of which is a binary tree. The applications of Binary Trees are to represent arithmetic expressions, creating decision processes and searching.

## Binary Search Tree

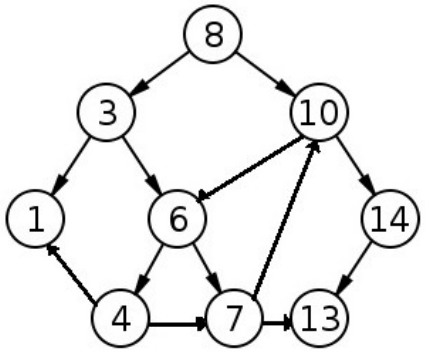
It is binary tree with the property that all the nodes to the left of a given node has key values lesser than that node and all nodes to the right have key values greater than the node. An in-order traversal of a binary search tree returns the elements of the BST in ascending order of key values of nodes.

Search in a BST for a key k, is done by following a downward path starting at the root. The next node visited depends on the value of the key of the current node. If key of current node is greater than k then the left subtree is searched in same recursive fashion. If key of current node is smaller than k then the right subtree is searched in same recursive fashion. If the key of current node matches the value k then the position of current node is returned. If a leaf node is reached, the key is not found and a null position is returned.

# Graph

A graph G(V,E) is a defined with two sets. V , a set of vertices, and E the set of edges between two pair of vertices from the set V.

When the edges of the graph have to be defined with the direction, it is called a *directed graph or digraph*, and the edges are called directed edges or arcs. In a digraph edge (a,b is not the same as edge (b, a). In a directed edge (a,b) a is called the source/starting point and b is called the sink/destination/ end point. In an undirected graph the direction of edge is not specified.




- When the edges of the graph have to be defined with the direction, it is called a *directed graph or digraph*, and the edges are called directed edges or arcs. In a digraph edge (a,b is not the same as edge (b, a). In a directed edge (a,b) a is called the source/starting point and b is called the sink/destination/ end point. In an undirected graph the direction of edge is not specified.Two vertices that are connected t each other with an edge are called neighbors.They are also said to be *adjacent* to each other.





### What is Depth First Search (DFS)?

The algorithm begins at the root node and then it explores each branch before backtracking. It is implemented using stacks. Often while writing the code, we use recursion stacks to backtrack. By using recursion we are able to take advantage of the fact that left and right subtrees are also trees and share the same properties.

For Binary trees, there are three types of DFS traversals.

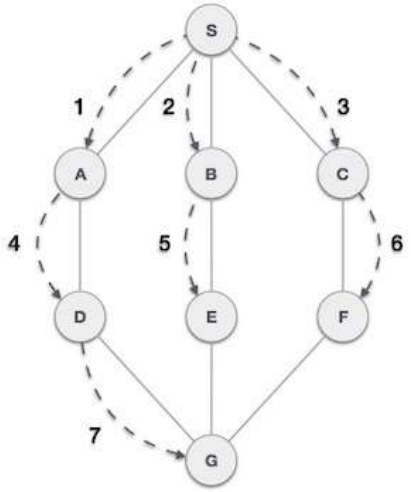
1. **In-Order**
2. **Pre-Order**
3. **Post-Order**

### What is Breadth-First Search (BFS)?

This algorithm also begins at the root node and then visits all nodes level by level. That means after the root, it traverses all the direct children of the root. After all direct children of the root are traversed, it moves to their children and so on. To implement BFS we use a queue.

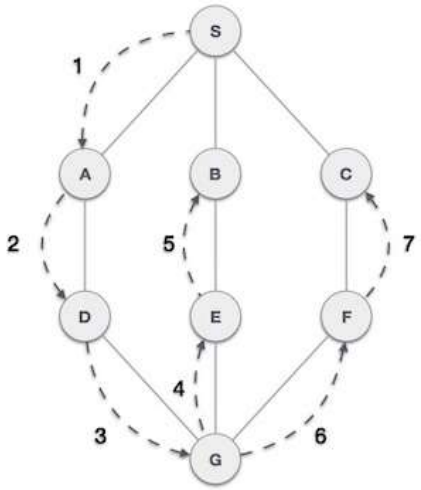
## BFS

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.



## DFS

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.




Following are the important differences between BFS and DFS.

| Sr. No. | Key | BFS | DFS |
|---|---|---|---|
| 1 | Definition | BFS, stands for Breadth First Search. | DFS, stands for Depth First Search. |
| 2 | Data structure | BFS uses Queue to find the shortest path. | DFS uses Stack to find the shortest path. |
| 3 | Source | BFS is better when target is closer to Source. | DFS is better when target is far from source. |
| 4 | Suitablity for | As BFS considers all neighbour so it is not | DFS is more suitable for decision tree. As with one decision, we need to traverse further to |





| Sr. No. | Key | BFS | DFS |
|---|---|---|---|
| | decision tree | suitable for decision tree used in puzzle games. | augment the decision. If we reach the conclusion, we won. |
| 5 | Speed | BFS is slower than DFS. | DFS is faster than BFS. |
| 6 | Time Complexity | Time Complexity of BFS = O(V+E) where V is vertices and E is edges. | Time Complexity of DFS is also O(V+E) where V is vertices and E is edges. |

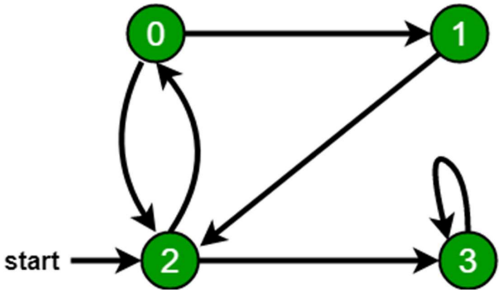
Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

**Example:**
*In the following graph, we start traversal from vertex 2.*




*When we come to **vertex 0**, we look for all adjacent vertices of it.*

- *2 is also an adjacent vertex of 0.*
- *If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.*

*There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :*
*2, 3, 0, 1*
*2, 0, 3, 1*

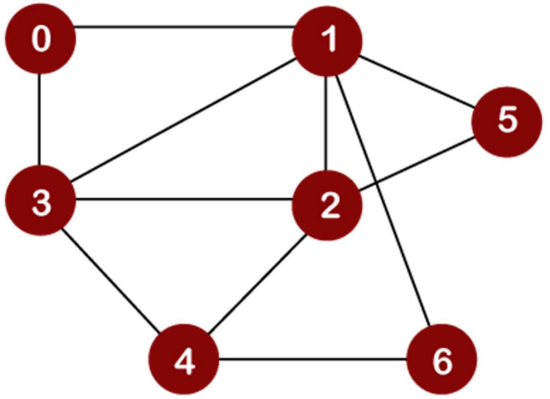




## What is BFS?

BFS stands for ***Breadth First Search***. It is also known as **level order traversal**. The Queue data structure is used for the Breadth First Search traversal. When we use the BFS algorithm for the traversal in a graph, we can consider any node as a root node.

**Let's consider the below graph for the breadth first search traversal.**




Suppose we consider node 0 as a root node. Therefore, the traversing would be started from node 0.

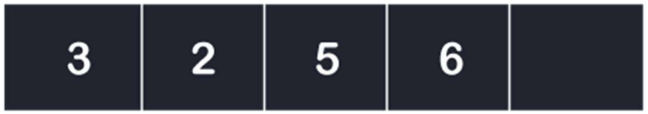

Once node 0 is removed from the Queue, it gets printed and marked as a ***visited node.***

Once node 0 gets removed from the Queue, then the adjacent nodes of node 0 would be inserted in a Queue as shown below:


**Result : 0**

Now the node 1 will be removed from the Queue; it gets printed and marked as a visited node

Once node 1 gets removed from the Queue, then all the adjacent nodes of a node 1 will be added in a Queue. The adjacent nodes of node 1 are 0, 3, 2, 6, and 5. But we have to insert only unvisited nodes in a Queue. Since nodes 3, 2, 6, and 5 are unvisited; therefore, these nodes will be added in a Queue as shown below:




**Result : 0 , 1**

The next node is 3 in a Queue. So, node 3 will be removed from the Queue, it gets printed and marked as visited as shown below:


**Result : 0, 1, 3**

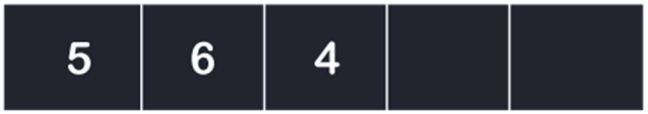
Once node 3 gets removed from the Queue, then all the adjacent nodes of node 3 except the visited nodes will be added in a Queue. The adjacent nodes of node 3 are 0, 1, 2, and 4. Since nodes 0, 1 are already visited, and node 2 is present in a Queue; therefore, we need to insert only node 4 in a Queue.






**Result : 0, 1, 3**

Now, the next node in the Queue is 2. So, 2 would be deleted from the Queue. It gets printed and marked as visited as shown below:




**Result : 0, 1, 3, 2,**

Once node 2 gets removed from the Queue, then all the adjacent nodes of node 2 except the visited nodes will be added in a Queue. The adjacent nodes of node 2 are 1, 3, 5, 6, and 4. Since the nodes 1 and 3 have already been visited, and 4, 5, 6 are already added in the Queue; therefore, we do not need to insert any node in the Queue.

The next element is 5. So, 5 would be deleted from the Queue. It gets printed and marked as visited as shown below:


**Result : 0, 1, 3, 2, 5**

Once node 5 gets removed from the Queue, then all the adjacent nodes of node 5 except the visited nodes will be added in the Queue. The adjacent nodes of node 5 are 1 and 2. Since both the nodes have already been visited; therefore, there is no vertex to be inserted in a Queue.

The next node is 6. So, 6 would be deleted from the Queue. It gets printed and marked as visited as shown below:


Result : 0, 1, 3, 2, 5, 6

Once the node 6 gets removed from the Queue, then all the adjacent nodes of node 6 except the visited nodes will be added in the Queue. The adjacent nodes of node 6 are 1 and 4. Since the node 1 has already been visited and node 4 is already added in the Queue; therefore, there is not vertex to be inserted in the Queue.

The next element in the Queue is 4. So, 4 would be deleted from the Queue. It gets printed and marked as visited.

Once the node 4 gets removed from the Queue, then all the adjacent nodes of node 4 except the visited nodes will be added in the Queue. The adjacent nodes of node 4 are 3, 2, and 6. Since all the adjacent nodes have already been visited; so, there is no vertex to be inserted in the Queue.

## What is DFS?

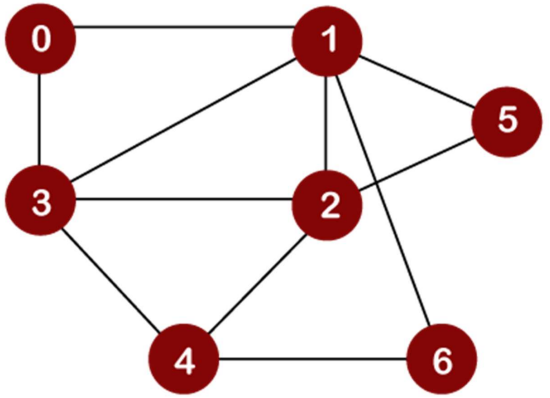
DFS stands for Depth First Search. In DFS traversal, the stack data structure is used, which works on the LIFO (Last In First Out) principle. In DFS, traversing can be started from any node, or we can say that any node can be considered as a root node until the root node is not mentioned in the problem.

In the case of BFS, the element which is deleted from the Queue, the adjacent nodes of the deleted node are added to the Queue. In contrast, in DFS, the element which is removed from the stack, then only one adjacent node of a deleted node is added in the stack.

**Let's consider the below graph for the Depth First Search traversal.**

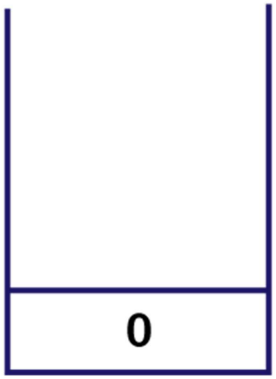




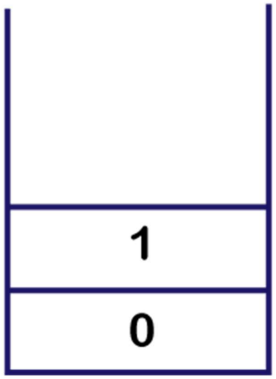



Consider node 0 as a root node.

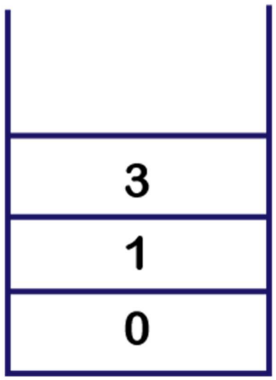
First, we insert the element 0 in the stack as shown below:




The node 0 has two adjacent nodes, i.e., 1 and 3. Now we can take only one adjacent node, either 1 or 3, for traversing. Suppose we consider node 1; therefore, 1 is inserted in a stack and gets printed as shown below:



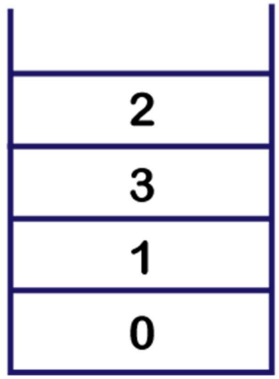

Now we will look at the adjacent vertices of node 1. The unvisited adjacent vertices of node 1 are 3, 2, 5 and 6. We can consider any of these four vertices. Suppose we take node 3 and insert it in the stack as shown below:



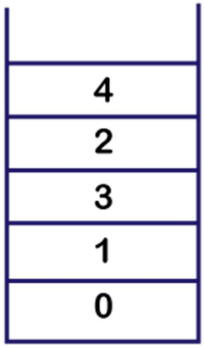

Consider the unvisited adjacent vertices of node 3. The unvisited adjacent vertices of node 3 are 2 and 4. We can take either of the vertices, i.e., 2 or 4. Suppose we take vertex 2 and insert it in the stack as shown below:



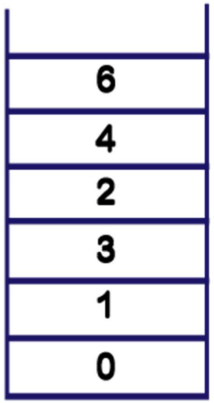





The unvisited adjacent vertices of node 2 are 5 and 4. We can choose either of the vertices, i.e., 5 or 4. Suppose we take vertex 4 and insert in the stack as shown below:



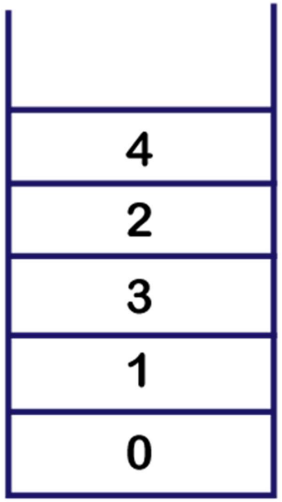

Now we will consider the unvisited adjacent vertices of node 4. The unvisited adjacent vertex of node 4 is node 6. Therefore, element 6 is inserted into the stack as shown below:

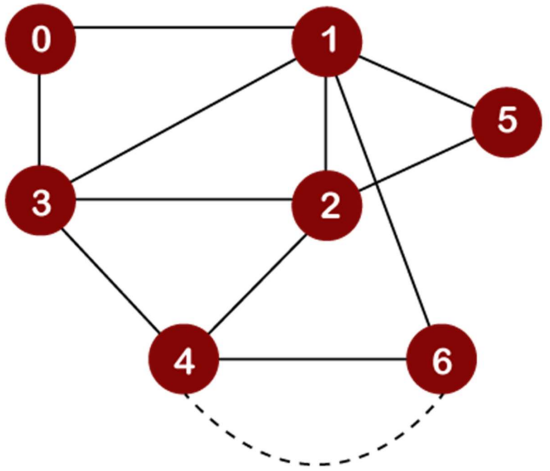



After inserting element 6 in the stack, we will look at the unvisited adjacent vertices of node 6. As there is no unvisited adjacent vertices of node 6, so we cannot move beyond node 6. In this case, we will perform **backtracking**. The topmost element, i.e., 6 would be popped out from the stack as shown below:

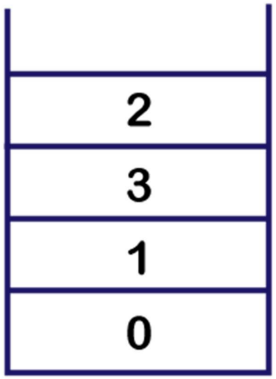


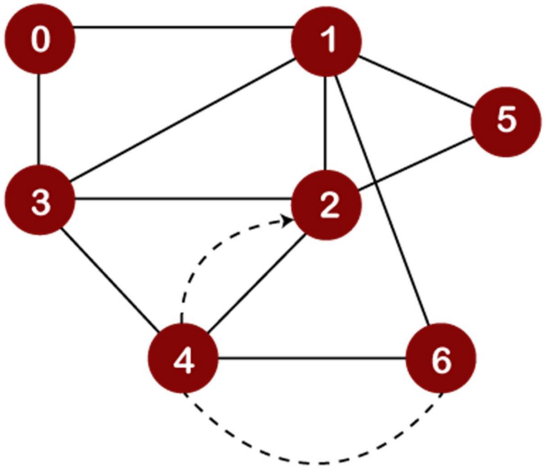





The topmost element in the stack is 4. Since there are no unvisited adjacent vertices left of node 4; therefore, node 4 is popped out from the stack as shown below:

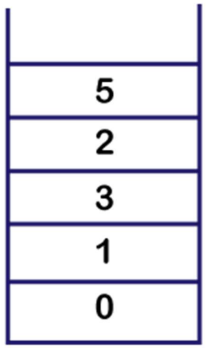


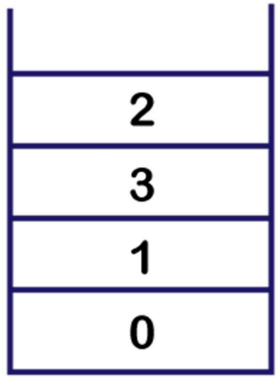

The next topmost element in the stack is 2. Now, we will look at the unvisited adjacent vertices of node 2. Since only one unvisited node, i.e., 5 is left, so node 5 would be pushed into the stack above 2 and gets printed as shown below:



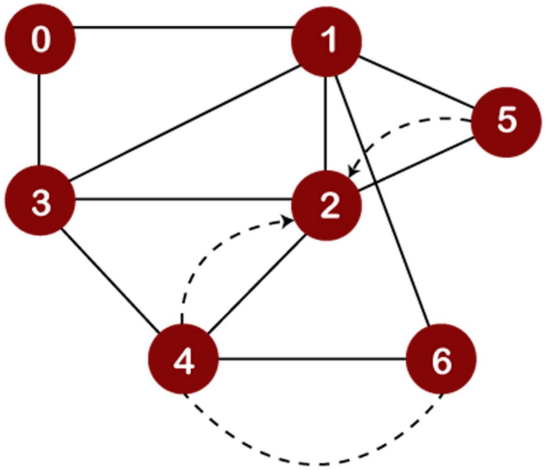





Now we will check the adjacent vertices of node 5, which are still unvisited. Since there is no vertex left to be visited, so we pop the element 5 from the stack as shown below:



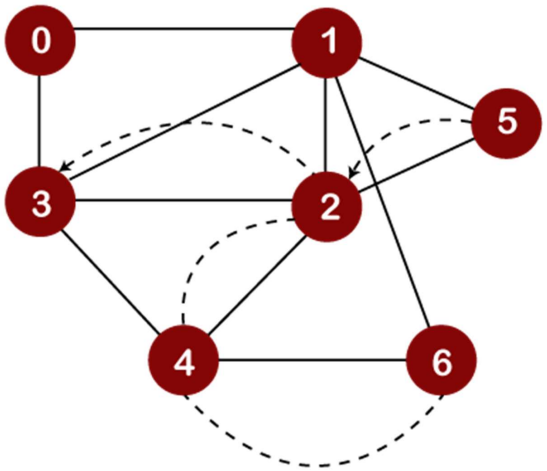

We cannot move further 5, so we need to perform backtracking. In backtracking, the topmost element would be popped out from the stack. The topmost element is 5 that would be popped out from the stack, and we move back to node 2 as shown below:

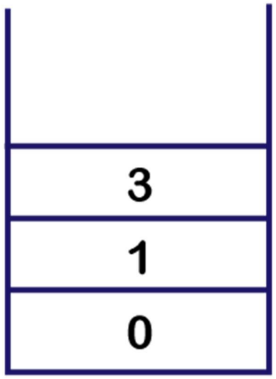



Now we will check the unvisited adjacent vertices of node 2. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 2 would be popped out from the stack, and we move back to the node 3 as shown below:

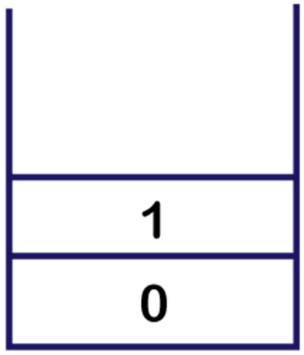


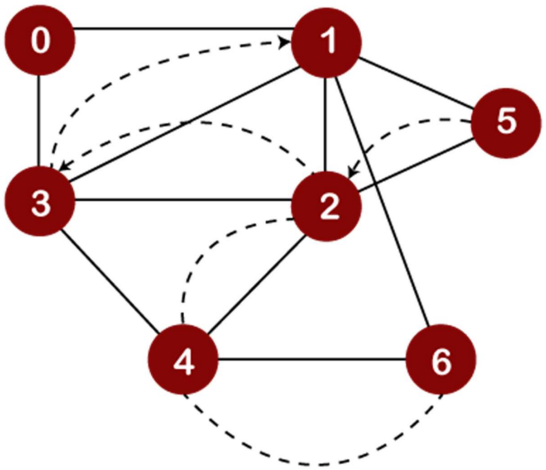





Now we will check the unvisited adjacent vertices of node 3. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 3 would be popped out from the stack and we move back to node 1 as shown below:

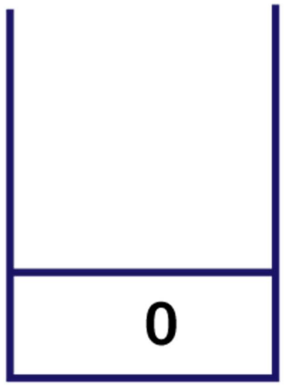


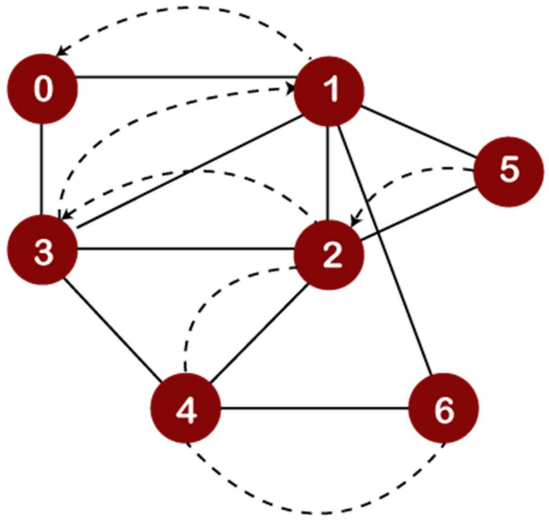




After popping out element 3, we will check the unvisited adjacent vertices of node 1. Since there is no vertex left to be visited; therefore, the backtracking will be performed. In backtracking, the topmost element, i.e., 1 would be popped out from the stack, and we move back to node 0 as shown below:








We will check the adjacent vertices of node 0, which are still unvisited. As there is no adjacent vertex left to be visited, so we perform backtracking. In this, only one element, i.e., 0 left in the stack, would be popped out from the stack as shown below:


As we can observe in the above figure that the stack is empty. So, we have to stop the DFS traversal here, and the elements which are printed is the result of the DFS traversal.