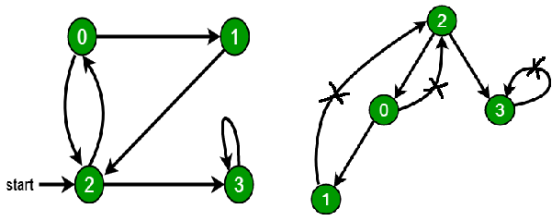
# Depth First Search or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For 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. A Depth First Traversal of the following graph is 2, 0, 1, 3.




See this post for all applications of Depth First Traversal.
Following are implementations of simple Depth First Traversal. The C++ implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes.

**Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.**

## C++

```
// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};
```

```
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

// Driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

### Java

```
// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices

    // Array  of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
```

```
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        DFSUtil(v, visited);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal "+
                           "(starting from vertex 2)");

        g.DFS(2);
    }
}
// This code is contributed by Aakash Hasija
```

## Python

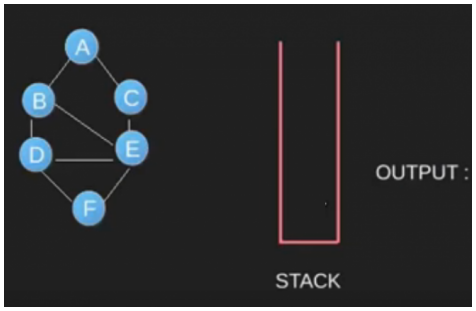
```
# Python program to print DFS traversal from a
# given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
```

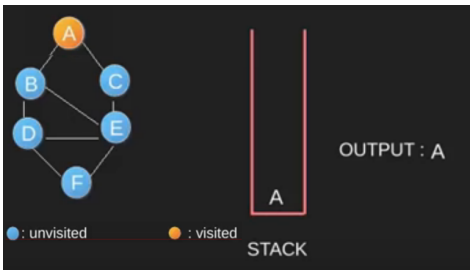
```
    def DFSUtil(self,v,visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print v,

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)


    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self,v):

        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

        # Call the recursive helper function to print
        # DFS traversal
        self.DFSUtil(v,visited)


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is DFS from (starting from vertex 2)"
g.DFS(2)

# This code is contributed by Neelam Yadav
```

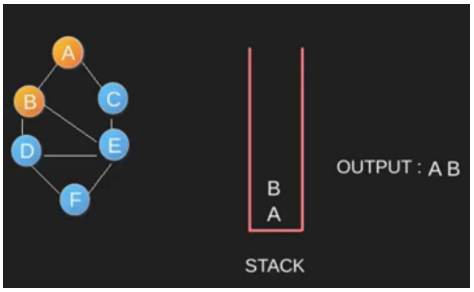
**Output:**

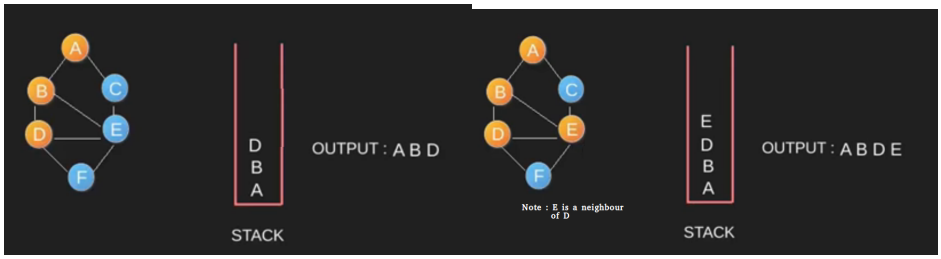
```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

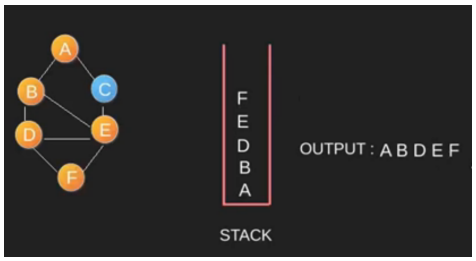
**Illustration for an Undirected Graph :**

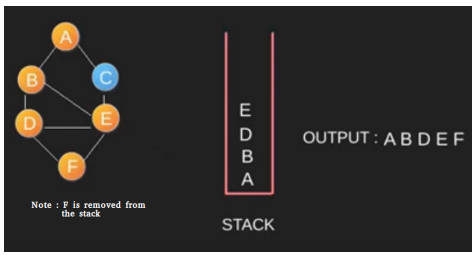


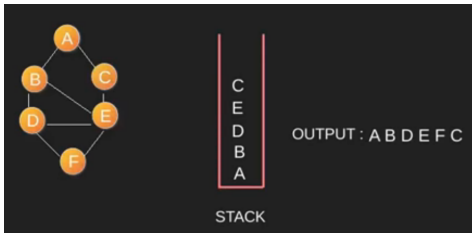



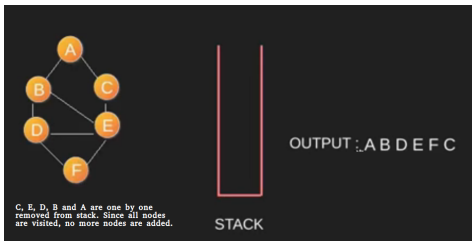






**How to handle disconnected graph?**

The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call DFSUtil() for every vertex. Also, before calling DFSUtil(), we should check if it is already printed by some other call of DFSUtil(). Following implementation does the complete graph traversal even if the nodes are unreachable. The differences from the above code are highlighted in the below code.

### C++

```
// C++ program to print DFS traversal for a given given graph
#include<iostream>
#include         <list>
using namespace std;

class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]);  // A function used by DFS
public:
```

```
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // function to add an edge to graph
    void DFS();    // prints DFS traversal of the complete graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            DFSUtil(*i, visited);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFSUtil(i, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversaln";
    g.DFS();

    return 0;
}
```

### Java

```
// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices

    // Array  of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
```

```
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);    // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS()
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        // starting from all vertices one by one
        for (int i=0; i<V; ++i)
            if (visited[i] == false)
                DFSUtil(i, visited);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal");

        g.DFS();
    }
}
// This code is contributed by Aakash Hasija
```

## Python

```
# Python program to print DFS traversal for complete graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
```

```
    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print v,

        # Recur for all the vertices adjacent to
        # this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)


    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self):
        V = len(self.graph)  #total vertices

        # Mark all the vertices as not visited
        visited =[False]*(V)

        # Call the recursive helper function to print
        # DFS traversal starting from all vertices one
        # by one
        for i in range(V):
            if visited[i] == False:
                self.DFSUtil(i, visited)


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Depth First Traversal"
g.DFS()

# This code is contributed by Neelam Yadav
```

Output:

```
Following is Depth First Traversal
0 1 2 3
```

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.


- **Applications of DFS.**
- **Breadth First Traversal for a Graph**

- **Recent Articles on DFS**

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.



## Recommended Posts:

Iterative Depth First Traversal of Graph
Applications of Depth First Search
Top 10 Interview Questions on Depth First Search (DFS)
Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)
Breadth First Search or BFS for a Graph
Finding minimum vertex cover size of a graph using binary search
Convert the undirected graph into directed graph such that there is no path of length greater than 1
Detect cycle in the graph using degrees of nodes of graph
Graph implementation using STL for competitive programming | Set 2 (Weighted graph)
Flatten a multi-level linked list | Set 2 (Depth wise)
Best First Search (Informed Search)
Dominant Set of a Graph
Hypercube Graph
Graph and its representations
Transpose graph


**Article Tags :** Graph DFS graph-basics

**Practice Tags :** DFS Graph

46

To-do Done

2.3

Based on **319** vote(s)

Feedback/ Suggest Improvement Notes Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

A computer science portal for geeks

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org