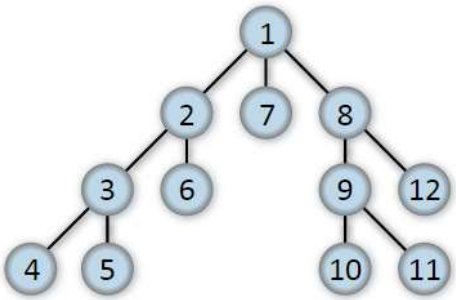
Depth First Search (DFS) – Iterative and Recursive Implementation

Depth–first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root for a graph) and explore as far as possible along each branch before backtracking.

The following graph shows the order in which the nodes are discovered in DFS:



Depth–first search in trees

A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any acyclic connected graph is a tree. For a tree, we have the following traversal methods:

- Preorder: visit each node before its children.
- Postorder: visit each node after its children.
- Inorder (for binary trees only): visit left subtree, node, right subtree.

These are already covered in detail in separate posts.

Depth–first search in Graph

A Depth–first search (DFS) is a way of traversing graphs closely related to the preorder traversal of a tree. Following is the recursive implementation of preorder traversal:

```
procedure preorder(treeNode v)
{
   visit(v);
   for each child u of v
      preorder(u);
}
```

To turn this into a graph traversal algorithm, replace “child” with “neighbor”. But to prevent infinite loops, keep track of the vertices that are already discovered and not revisit them.

```
procedure dfs(vertex v)
{
   visit(v);
   for each neighbor u of v
      if u is undiscovered
         call dfs(u);
```

}

The recursive algorithm can be implemented as follows in C++, Java, and Python:

- C++
- Java
- Python

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

// A class to store a graph edge
class Edge
{
    int source, dest;

    public Edge(int source, int dest)
    {
        this.source = source;
        this.dest = dest;
    }
}

// A class to represent a graph object
class Graph
{
    // A list of lists to represent an adjacency list
    List<List<Integer>> adjList = null;

    // Constructor
    Graph(List<Edge> edges, int n)
    {
        adjList = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<>());
        }

        // add edges to the undirected graph
        for (Edge edge: edges)
        {
            int src = edge.source;
            int dest = edge.dest;

            adjList.get(src).add(dest);
            adjList.get(dest).add(src);
        }
    }
}

class Main
{
    // Function to perform DFS traversal on the graph on a graph
    public static void DFS(Graph graph, int v, boolean[] discovered)
    {
        // mark the current node as discovered
        discovered[v] = true;
```

```
50
51          // print the current node
52          System.out.print(v + " ");
53
54          // do for every edge (v, u)
55          for (int u: graph.adjList.get(v))
56          {
57              // if `u` is not yet discovered
58              if (!discovered[u]) {
59                  DFS(graph, u, discovered);
60              }
61          }
62      }
63
64      public static void main(String[] args)
65      {
66          // List of graph edges as per the above diagram
67          List<Edge> edges = Arrays.asList(
68                  // Notice that node 0 is unconnected
69                  new Edge(1, 2), new Edge(1, 7), new Edge(1, 8), new Edge(2, 3),
70                  new Edge(2, 6), new Edge(3, 4), new Edge(3, 5), new Edge(8, 9),
71                  new Edge(8, 12), new Edge(9, 10), new Edge(9, 11)
72              );
73
74          // total number of nodes in the graph (labelled from 0 to 12)
75          int n = 13;
76
77          // build a graph from the given edges
78          Graph graph = new Graph(edges, n);
79
80          // to keep track of whether a vertex is discovered or not
81          boolean[] discovered = new boolean[n];
82
83          // Perform DFS traversal from all undiscovered nodes to
84          // cover all connected components of a graph
85          for (int i = 0; i < n; i++)
86          {
87              if (!discovered[i]) {
88                  DFS(graph, i, discovered);
89              }
90          }
91      }
92  }
```

Download Run Code

Output:

0 1 2 3 4 5 6 7 8 9 10 11 12

The time complexity of DFS traversal is O(V + E), where V and E are the total number of vertices and edges in the graph, respectively. Please note that O(E) may vary between O(1) and O(V2), depending on how dense the graph is.

Iterative Implementation of DFS

The non-recursive implementation of DFS is similar to the non-recursive implementation of BFS but differs from it in two ways:

- It uses a stack instead of a queue.
- The DFS should mark discovered only after popping the vertex, not before pushing it.
- It uses a reverse iterator instead of an iterator to produce the same results as recursive DFS.
  Following is the C++, Java, and Python program that demonstrates it:

- C++
- Java
- Python

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Stack;

// A class to store a graph edge
class Edge
{
    int source, dest;

    public Edge(int source, int dest)
    {
        this.source = source;
        this.dest = dest;
    }
}

// A class to represent a graph object
class Graph
{
    // A list of lists to represent an adjacency list
    List<List<Integer>> adjList = null;

    // Constructor
    Graph(List<Edge> edges, int n)
    {
        adjList = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<>());
        }

        // add edges to the undirected graph
        for (Edge edge: edges)
        {
            int src = edge.source;
            int dest = edge.dest;

            adjList.get(src).add(dest);
            adjList.get(dest).add(src);
        }
    }
}
```

```java

class Main
{
    // Perform iterative DFS on graph starting from vertex `v`
    public static void iterativeDFS(Graph graph, int v, boolean[] discovered)
    {
        // create a stack used to do iterative DFS
        Stack<Integer> stack = new Stack<>();

        // push the source node into the stack
        stack.push(v);

        // loop till stack is empty
        while (!stack.empty())
        {
            // Pop a vertex from the stack
            v = stack.pop();

            // if the vertex is already discovered yet, ignore it
            if (discovered[v]) {
                continue;
            }

            // we will reach here if the popped vertex `v` is not discovered yet;
            // print `v` and process its undiscovered adjacent nodes into the stack
            discovered[v] = true;
            System.out.print(v + " ");

            // do for every edge (v, u)
            List<Integer> adjList = graph.adjList.get(v);
            for (int i = adjList.size() - 1; i >= 0; i--)
            {
                int u = adjList.get(i);
                if (!discovered[u]) {
                    stack.push(u);
                }
            }
        }
    }

    public static void main(String[] args)
    {
        // List of graph edges as per the above diagram
        List<Edge> edges = Arrays.asList(
                // Notice that node 0 is unconnected
                new Edge(1, 2), new Edge(1, 7), new Edge(1, 8), new Edge(2, 3),
                new Edge(2, 6), new Edge(3, 4), new Edge(3, 5), new Edge(8, 9),
                new Edge(8, 12), new Edge(9, 10), new Edge(9, 11)
                // (6, 9) introduces a cycle
        );

        // total number of nodes in the graph (labelled from 0 to 12)
        int n = 13;

        // build a graph from the given edges
        Graph graph = new Graph(edges, n);

```

```
100         // to keep track of whether a vertex is discovered or not
101         boolean[] discovered = new boolean[n];
102
103         // Do iterative DFS traversal from all undiscovered nodes to
104         // cover all connected components of a graph
105         for (int i = 0; i < n; i++)
106         {
107             if (!discovered[i]) {
108                 iterativeDFS(graph, i, discovered);
109             }
110         }
111     }
112 }
```

Download Run Code

Output:

0 1 2 3 4 5 6 7 8 9 10 11 12

Applications of DFS

- Finding connected components in a graph.
- Topological sorting in a DAG(Directed Acyclic Graph).
- Finding 2/3–(edge or vertex)–connected components.
- Finding the bridges of a graph.
- Finding strongly connected components.
- Solving puzzles with only one solution, such as mazes.
- Finding biconnectivity in graphs and many more…

From <https://www.techiedelight.com/depth-first-search/>