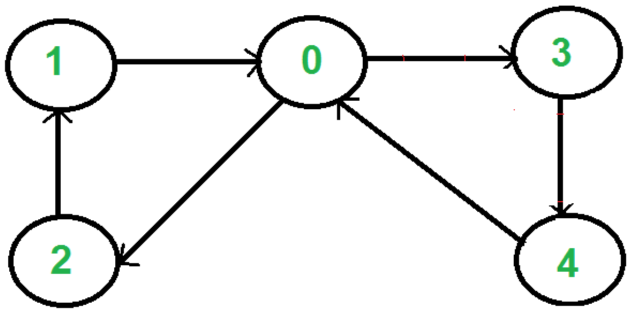
Iterative Depth First Traversal of Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal (DFS) 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, a DFS of below graph is "0 3 4 2 1", other possible DFS is "0 2 1 3 4".



We have discussed recursive implementation of DFS in previous in previous post. In the post, iterative DFS is discussed. The recursive implementation uses function call stack. In iterative implementation, an explicit stack is used to hold visited vertices.

Below is C++ implementation of Iterative DFS. ***The implementation is similar to BFS, the only difference is queue is replaced by stack.***

```
// An Iterative C++ program to do DFS traversal from
// a given source vertex. DFS(int s) traverses vertices
// reachable from s.
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // adjacency lists
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    void DFS(int s);  // prints DFS from a given source s
};

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.
}

// Function to print all vertices reachable from 's'
// using iterative DFS.
void Graph::DFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a stack for DFS
    stack<int> stack;
```

```
    // Mark the current node as visited and push it
    visited[s] = true;
    stack.push(s);

    // 'i' will be used to get all adjacent vertices
    // of a vertex
    list<int>::iterator i;

    while (!stack.empty())
    {
        // Pop a vertex from stack and print it
        s = stack.top();
        cout << s << " ";
        stack.pop();

        // Get all adjacent vertices of the popped vertex s
        // If a adjacent has not been visited, then mark it
        // visited and push it to the stack
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                stack.push(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(5);  // Total 5 vertices in graph
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    cout << "Following is Depth First Traversal "
          "(starting from vertex 0) \n";
    g.DFS(0);

    return 0;
}
```

Output:

```
Following is Depth First Traversal (starting from vertex 0)
0 3 4 2 1
```

Note that the above implementation prints only vertices that are reachable from a given vertex. For example, if we remove edges 0-3 and 0-2, the above program would only print 0. To print all vertices of a graph, we need to call DFS for every vertex. Below is C++ implementation for the same.

```
// An Iterative C++ program to do DFS traversal from
// a given source vertex. DFS(int s) traverses vertices
// reachable from s.
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // adjacency lists
public:
```

```
    Graph(int V);  // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    void DFS();  // prints all vertices in DFS manner

    // prints all not yet visited vertices reachable from s
    void DFSUtil(int s, bool visited[]);
};

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.
}

// prints all not yet visited vertices reachable from s
void Graph::DFSUtil(int s, bool visited[])
{
    // Create a stack for DFS
    stack<int> stack;

    // Mark the current node as visited and push it
    visited[s] = true;
    stack.push(s);

    // 'i' will be used to get all adjacent vertices
    // of a vertex
    list<int>::iterator i;

    while (!stack.empty())
    {
        // Pop a vertex from stack and print it
        s = stack.top();
        cout << s << " ";
        stack.pop();

        // Get all adjacent vertices of the popped vertex s
        // If a adjacent has not been visited, then mark it
        // visited and push it to the stack
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                stack.push(*i);
            }
        }
    }
}

// prints all vertices in DFS manner
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;

    for (int i = 0; i < V; i++)
        if (!visited[i])
           DFSUtil(i, visited);
}

// Driver program to test methods of graph class
int main()
{
    // Let us create a disconnected graph without
    // edges 0-2 and 0-3 in above graph
    Graph g(5);  // Total 5 vertices in graph
```

```
    Graph g(5); // Total 5 vertices in graph
    g.addEdge(1, 0);
    g.addEdge(2, 1);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

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

    return 0;
}
```

Output:

```
Following is Depth First Traversal
0 1 2 3 4
```

Like recursive traversal, time complexity of iterative implementation is O(V + E).