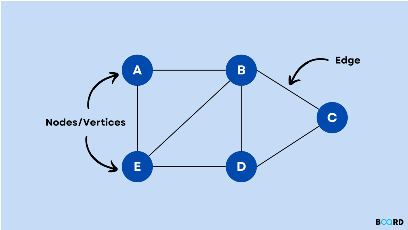
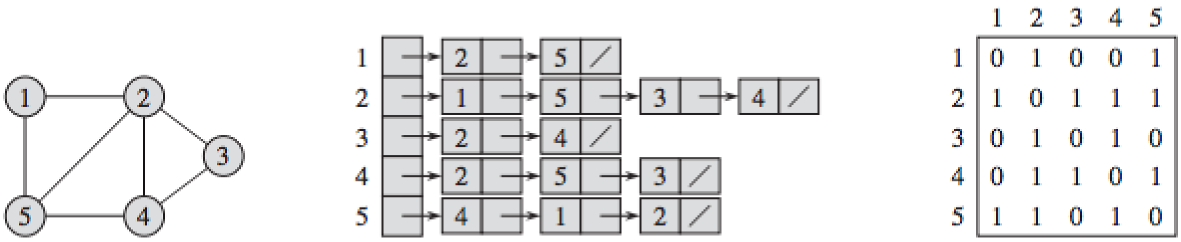
Graphs

A graph is a collection of nodes (vertices) with edges between (some of) them.

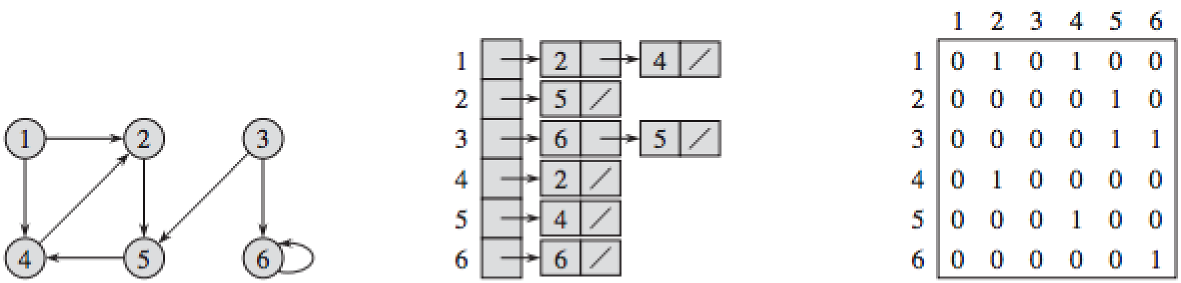


Representation

- Adjacency Matrix (1 if nodes are adjacent, otherwise 0)
- Adjacency List (vector of lists containing node neighbours)



(a) (b) (c)



(a) (b) (c)

Space Complexity	Adj Matrix	Adj List
avg. case	O(V^2)	O(V + E)
worst case	O(V^2)	O(V + E)

Time Complexity	Adj Matrix	Adj List
Adding a vertex	O(V^2)	O(1)
Removing a vertex	O(V^2)	O(V + E)
Adding an edge	O(1)	O(1)
Removing an edge	O(1)	O(E)

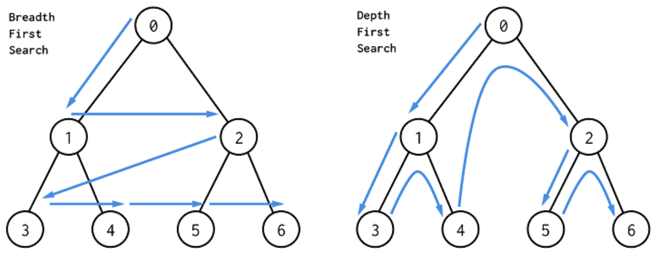
Time Complexity	Adj Matrix	Adj List
Edge query	O(1)	O(V)
Finding neighbours	O(V)	O(V)
Traversal	slow	faster

obs 1: Undirected graph <--> symmetric adjacency matrix

obs 2: Adjacency lists are usually prefered, especially for sparse graphs.

obs 3: There are some situations when the adjacency matrix is prefered (Floyd-Warshall, Complement, Transpose)

Searching



BFS - Breadth-First Search

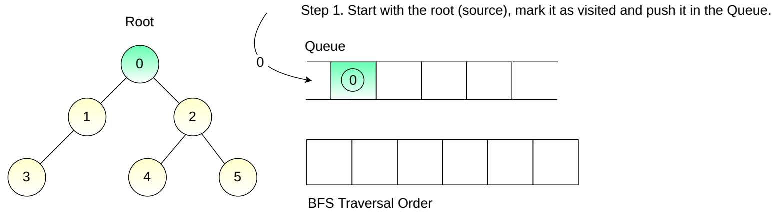
Complexity: **O(V + E)**

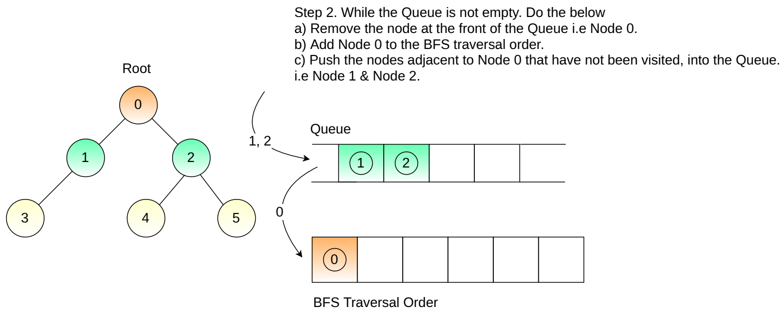
Main idea: start at the given node (lvl 0); visit neighbours of given node (lvl 1); visit unvisited neighbours of nodes on previous level (lvl 2) etc.

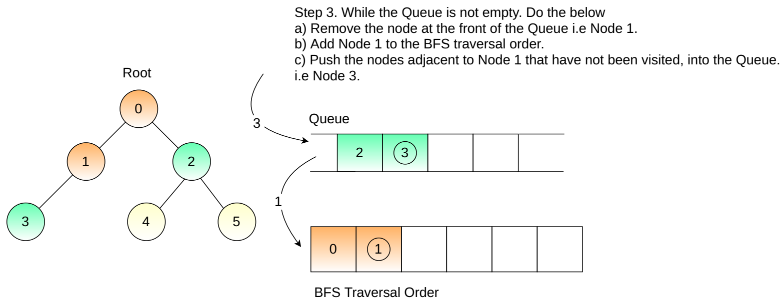
- We go **WIDE**, level by level, visiting the siblings before the children
- Uses a queue (FIFO)

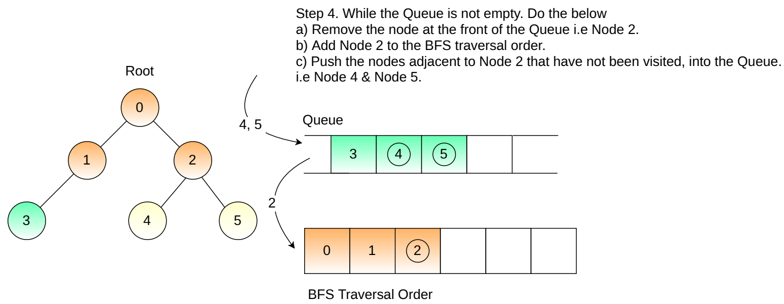
Pseudocode:

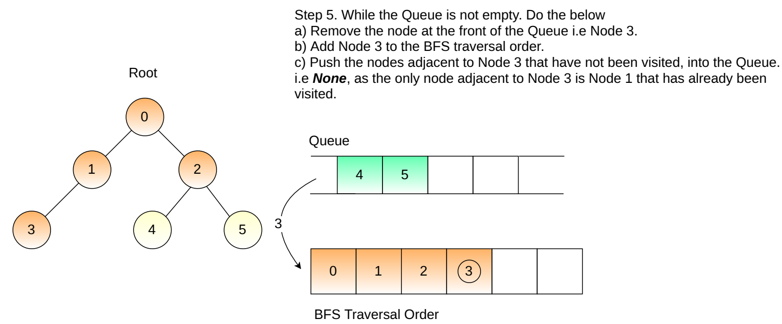
```
create a queue Q
mark v as visited
Q.enqueue(v)

while Q is not empty:
    x = Q.dequeue()
    for y in Neighbors(x):
        if (y has not been marked as visited) then
            mark y as visited
            Q.enqueue(y)
```

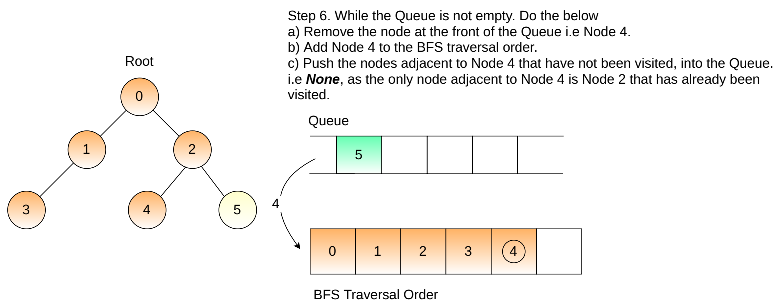




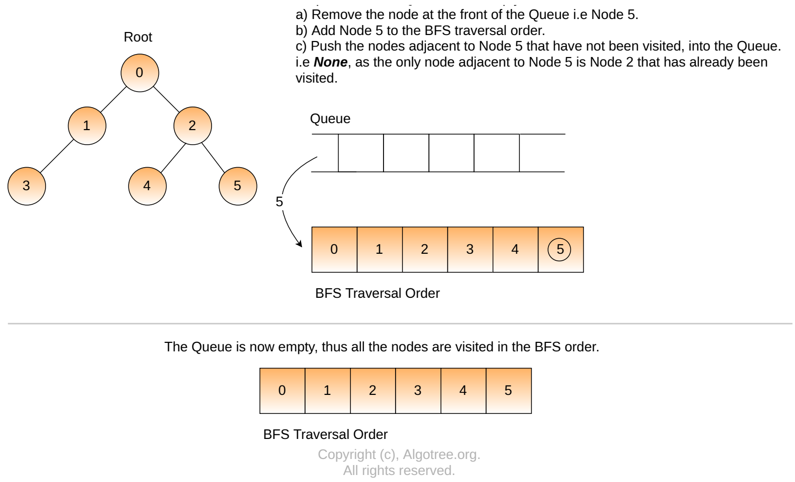








Step 7. While the Queue is not empty. Do the below



obs: BFS is best used for searching vertices close to the source (ex: determining the shortest path, checking if a graph is bipartite etc.)

obs: BFS is a bit slower than DFS and requires more memory.

DFS - Depth-First Search

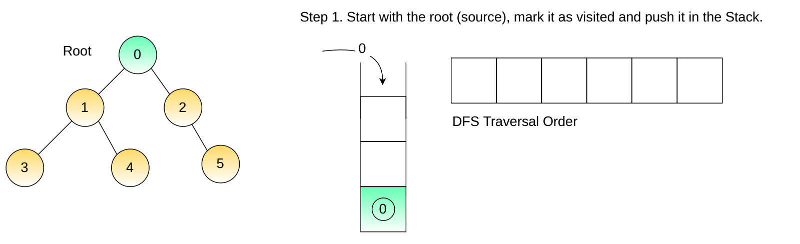
Complexity: **O(V + E)**

Main idea: start at the given node; explore each branch completely before moving on to the next branch

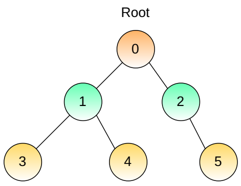
- We go **DEEP**, subtree by subtree, visiting all children before the siblings
- Uses a stack (LIFO) / recursion

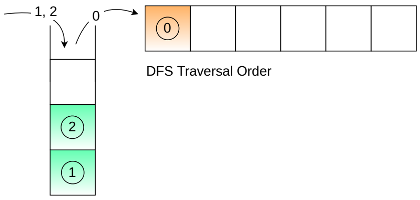
Pseudocode (recursive DFS):

```
DFS(G, u)
    u.visited = true
    for each neighbour v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    for each u ∈ G
        u.visited = false
    for each u ∈ G
       DFS(G, u)
}
```

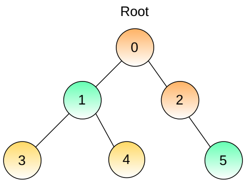


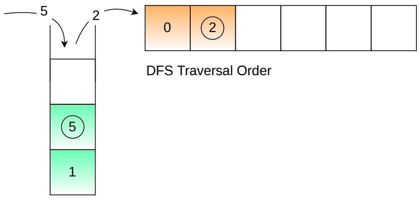
Step 2. While the Stack is not empty. Do the below
a) Remove the node at the top of the Stack i.e Node 0.
b) Add Node 0 to DFS traversal order.
c) Push the nodes adjacent to Node 0 that have not been visited, into the Stack. i.e Node 1 & Node 2.



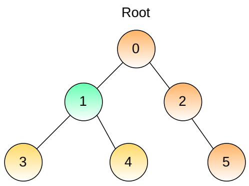


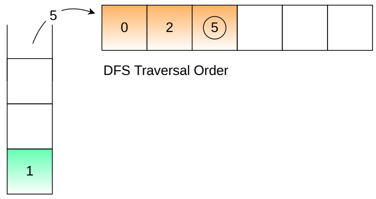
Step 3. While the Stack is not empty. Do the below
a) Remove the node at the top of the Stack i.e Node 2.
b) Add Node 2 to the DFS traversal order.
c) Push the nodes adjacent to Node 2 that have not been visited, into the Stack. i.e Node 5.



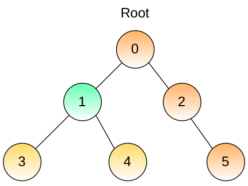


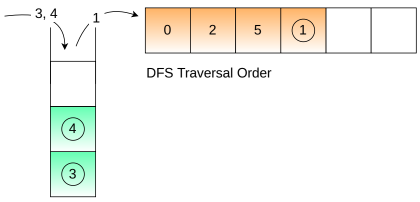
Step 4. While the Stack is not empty. Do the below
a) Remove the node at the top of the Stack i.e Node 5.
b) Add Node 5 to the DFS traversal order.
c) Push the nodes adjacent to Node 5 that have not been visited, into the Stack. i.e **None**. Since the only node adjacent to Node 5 is Node 2 which has already been visited.



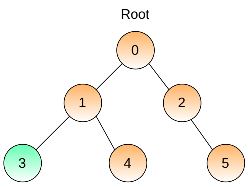


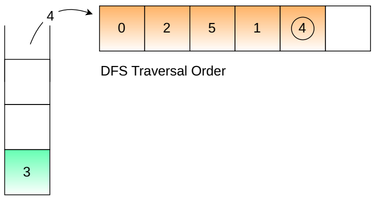
Step 5. While the Stack is not empty. Do the below
a) Remove the node at the top of the Stack i.e Node 1.
b) Add Node 1 to the DFS traversal order.
c) Push the nodes adjacent to Node 1 that have not been visited, into the Stack. i.e Node 3 and Node 4.



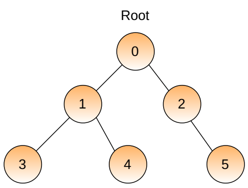


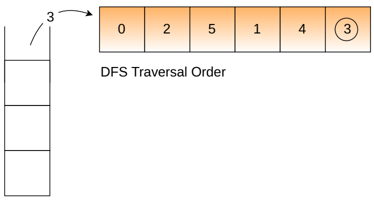
Step 6. While the Stack is not empty. Do the below
a) Remove the node at the top of the Stack i.e Node 4.
b) Add Node 4 to the DFS traversal order.
c) Push the nodes adjacent to Node 4 that have not been visited, into the Stack. i.e ***None***, as the only node adjacent to Node 4 is Node 1 that has already been visited.

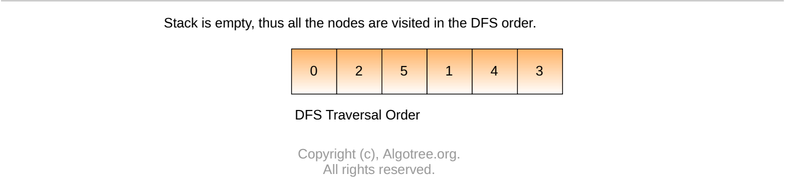




Step 7. While the Stack is not empty. Do the below
a) Remove the node at the top of the Stack i.e Node 3.
b) Add Node 3 to the DFS traversal order.
c) Push the nodes adjacent to Node 3 that have not been visited, into the Stack. i.e ***None***, as the only node adjacent to Node 3 is Node 1 that has already been visited.







obs: **Timed DFS** - find the arrival and departure time of its vertices in DFS

```
TIME = 0
dfs (v):
    arrival_time[v] = TIME ++
    visited[v] = true
    for u in adj[v]:
            if visited[u] == false
                    then dfs(u)

    departure_time[v] = TIME ++
```

Applications of finding Arrival and Departure Time:

- Topological sorting in a DAG (Directed Acyclic Graph).
- Finding 2/3–(edge or vertex)–connected components.
- Finding bridges in graphs.
- Finding biconnectivity in graphs.
- Detecting cycle in directed graphs.
- Tarjan's algorithm to find strongly connected components, and many more…

Obs: CC - DFS + BFS; Cycle - DFS + BFS; Shortest path - BFS; Bipartite - BFS; Topological Sort - DFS

Terminology

Directed graph

Undirected graph

Adjacent nodes

Incident edges

Conex graph: undirected graph; for each pair of vertices, there exists at least one single path which joins them

Complete graph: directed graph: every pair of distinct vertices is connected by a pair of unique edges (one in each direction)

Bipartite graph: its nodes can be divided into two different sets U and V such that every edge connects a node from U with a node in V

Partial graph: same nodes, fewer edges

Subgraph: fewer nodes, same edges connecting the remaining nodes

Clique: subset of nodes of an undirected graph such that the described subgraph is complete

Connected Components: subgraph of an undirected graph, in which every vertex is reachable from every other vertex

Strongly Connected Components: subgraph of a directed graph, in which every vertex is reachable from every other vertex

Cycle: the path described ends in the same point it began

Eulerian Cycle: goes through all edges exactly once (ex: the 7 bridges problem)

Hamiltonian Cycle: goes through all nodes exactly once (!simple cycle: no repetitions)

Graph Complement/Inverse: fill missing edges required to form a complete graph and remove all edges that were previously there

Graph Transpose/Reverse: change orientation of the edges

Topological Sort: (!only for DAGs) placing the nodes along a horizontal line so that all edges are directed from left to right (there are no eges pointing to the parents, quite like a genealogical tree)