NOTES

Programming implementation of Adjacency List Representation

```
#create class to represent the edge
class Edge:
	def __init__(self, src, dest):
		self.src = src
		self.dest = dest

#create class to represent the node
class Node:
	def __init__(self, value):
		self.value = value

#create class to graph
class Graph:
	def __init__(self, edges, N):
		# A list of lists to represent adjacency list
		self.adj = [None] * N

		# allocate memory for adjacency list
		for i in range(N):
			self.adj[i] = []

		# add edges to the undirected graph
		for e in edges:
			# allocate node in adjacency List from src to dest
			node = Node(e.dest)
			self.adj[e.src].append(node)


# print adjacency list representation of graph
def printGraph(graph):
	for src in range(len(graph.adj)):
		# print current vertex and all its neighboring vertices
		for edge in graph.adj[src]:
			print(f"({src} -> {edge.value}) ", end='')
		print()


# Input: Edges in a weighted digraph (as per above diagram)
# Edge(x, y) represents an edge from x to y having
edges = [Edge(0, 1), Edge(0, 2), Edge(0, 3), Edge(1, 2), Edge(1, 0), Edge(2, 1), Edge(2, 0), Edge(3, 1)]

# Input: No of vertices
N = 4

# construct graph from given list of edges
graph = Graph(edges, N)

# print adjacency list representation of the graph
printGraph(graph)
```

Output

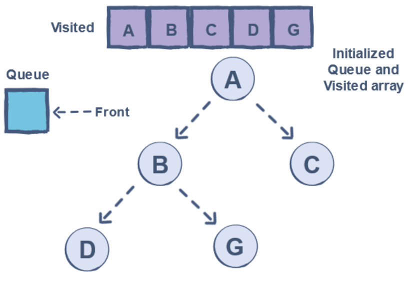
```
(0 -> 1) (0 -> 2) (0 -> 3)
(1 -> 2) (1 -> 0)
(2 -> 1) (2 -> 0)
(3 -> 1)
```

Breadth First Search Algorithm

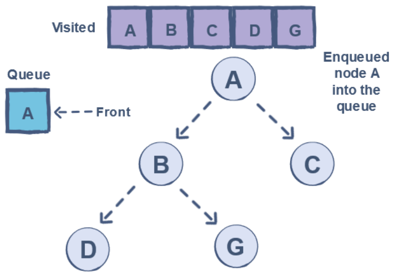
Look at the steps here

1. Pick a node and enqueue all its adjacent nodes into a queue.
2. Dequeue a node from the queue, mark it as visited and enqueue all its adjacent nodes into a queue.
3. Repeat this process until the queue is empty or you meet a goal.

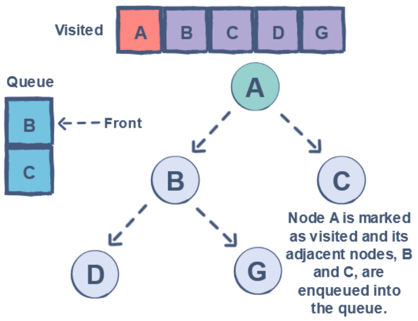
The program can be stuck in an infinite loop if a node is revisited and was not marked as visited before. Hence, prevent exploring nodes that are visited by marking them as visited.



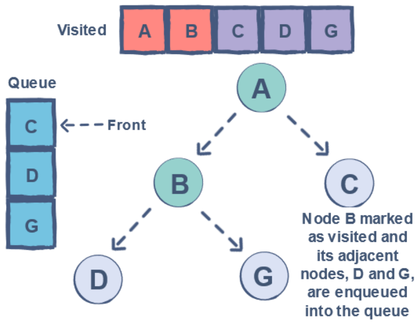
(i)



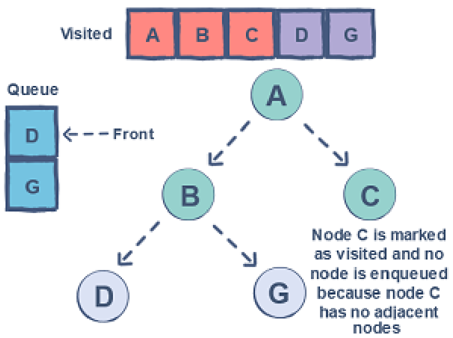
(ii)



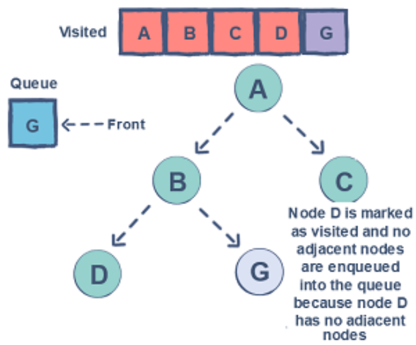
(iii)



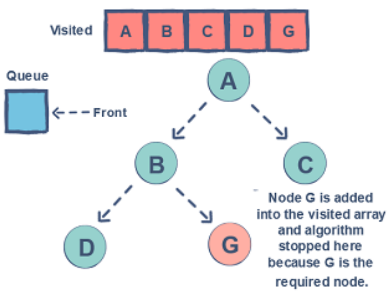
(iv)



(v)



(vi)



(vii)

DFS algorithm

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

[Example from PPT]

Comparison between BFS and DFS

Breadth First Search	Depth First Search

uses queue	uses stack
More suitable for searching vertices which are closer to the given source.	More suitable when there are solutions away from source.
The Time complexity of BFS is O(V + E), where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E), where V stands for vertices and E stands for edges.
Slow	Fast