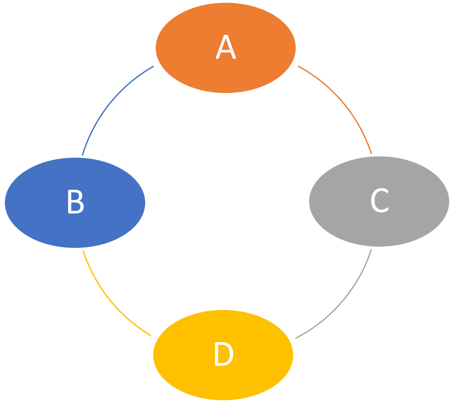
Representation of Graphs in memory
Graph Traversal Methods

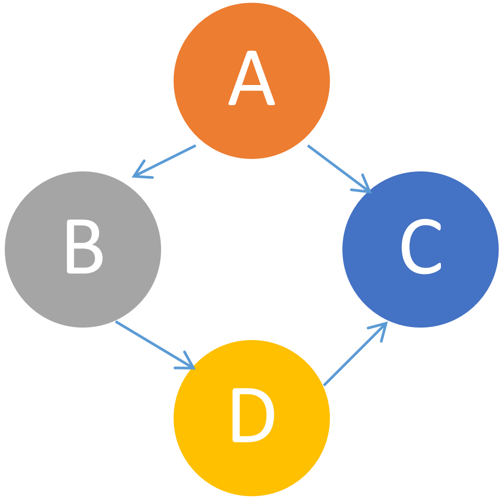
Representation of Graphs

- Adjacency Matrix
- Adjacency List

Adjacency Matrix



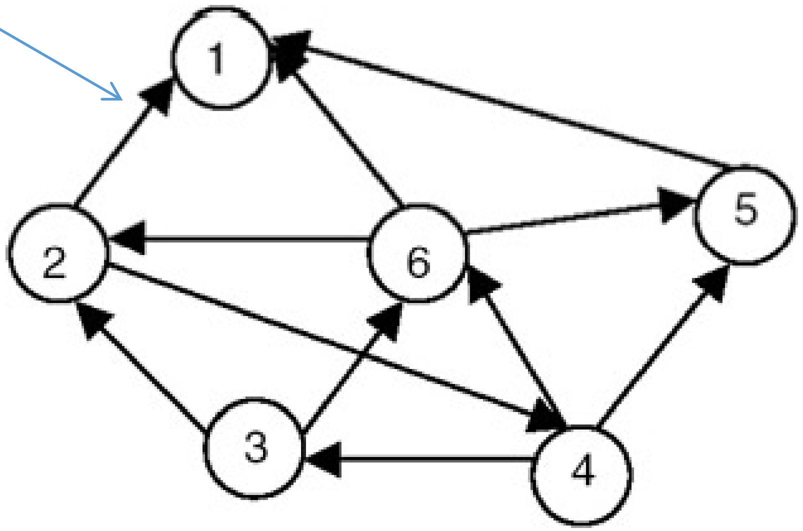
Row/ Column	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0



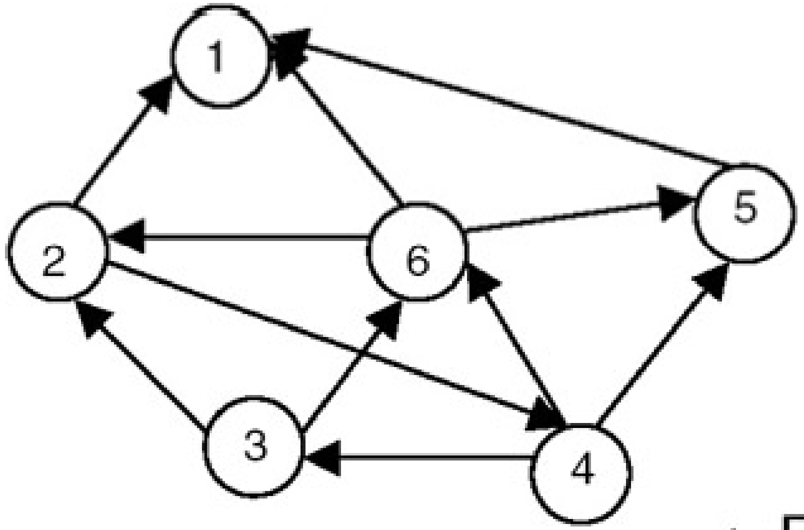
Row/ Column	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	0	0	1	0

1. Draw Graph if Array Representation of the graph is given

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	1	0	0	0	1
4	0	0	1	0	1	1
5	1	0	0	0	0	0
6	1	1	0	0	1	0

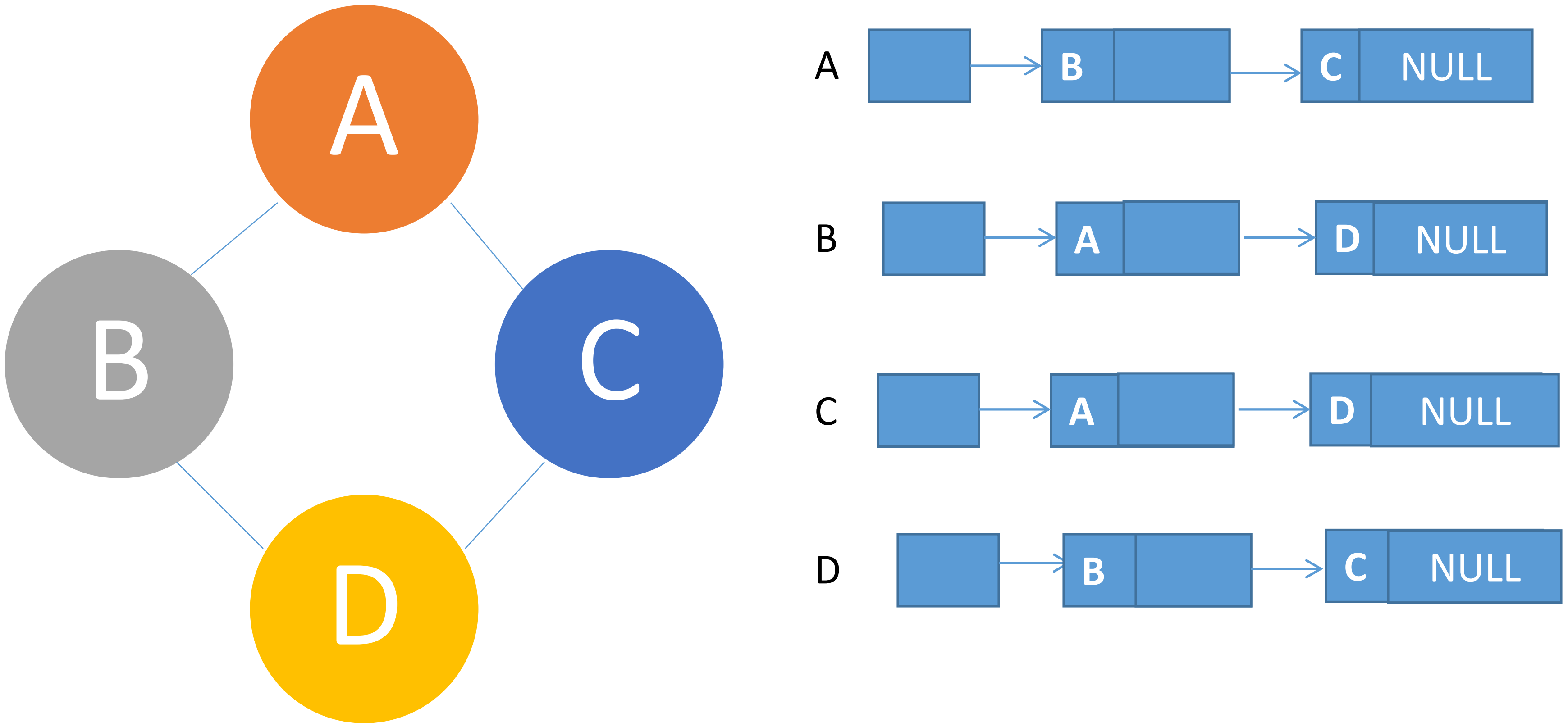


2. Show the Array Representation of the given graph

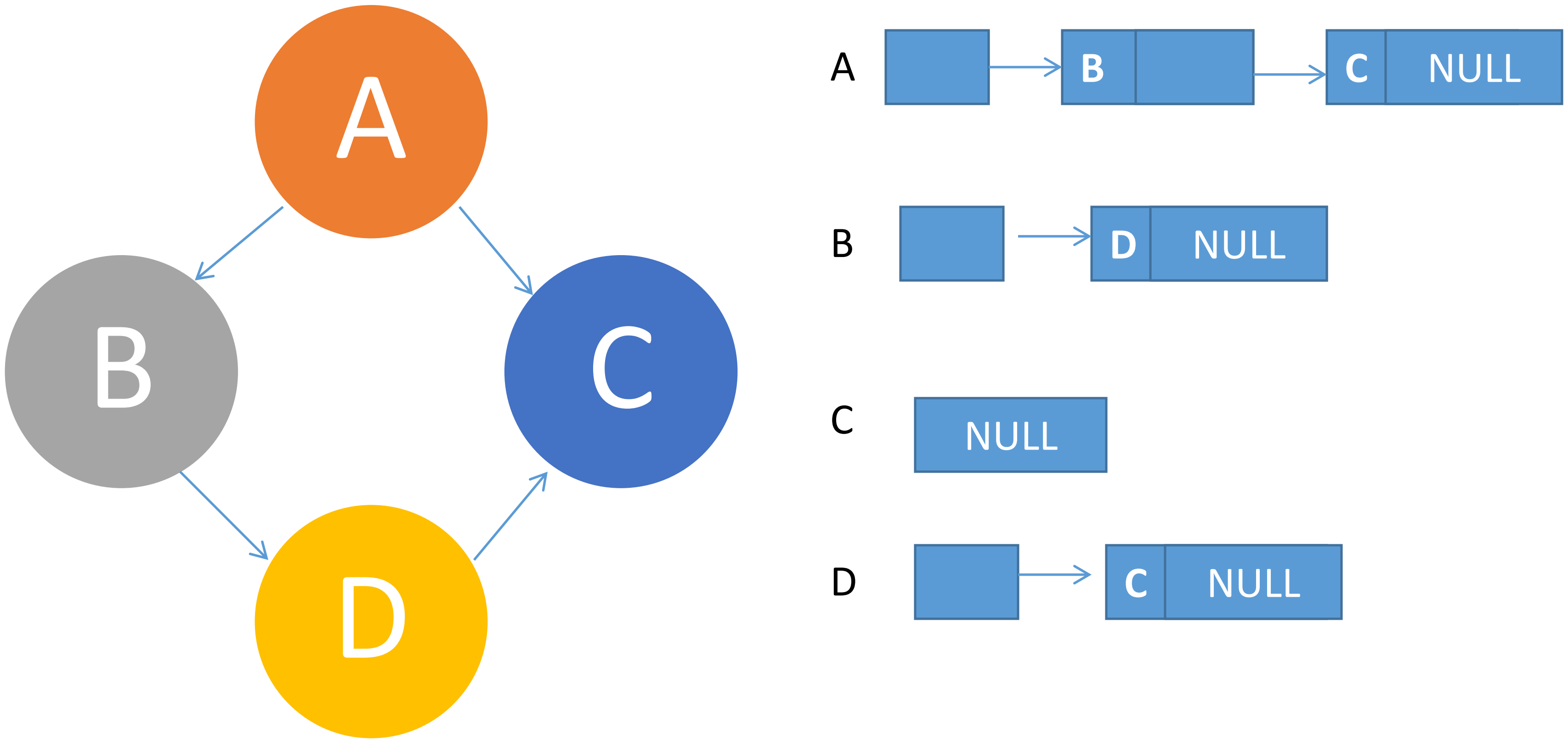


	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	1	0	0	0	1
4	0	0	1	0	1	1
5	1	0	0	0	0	0
6	1	1	0	0	1	0

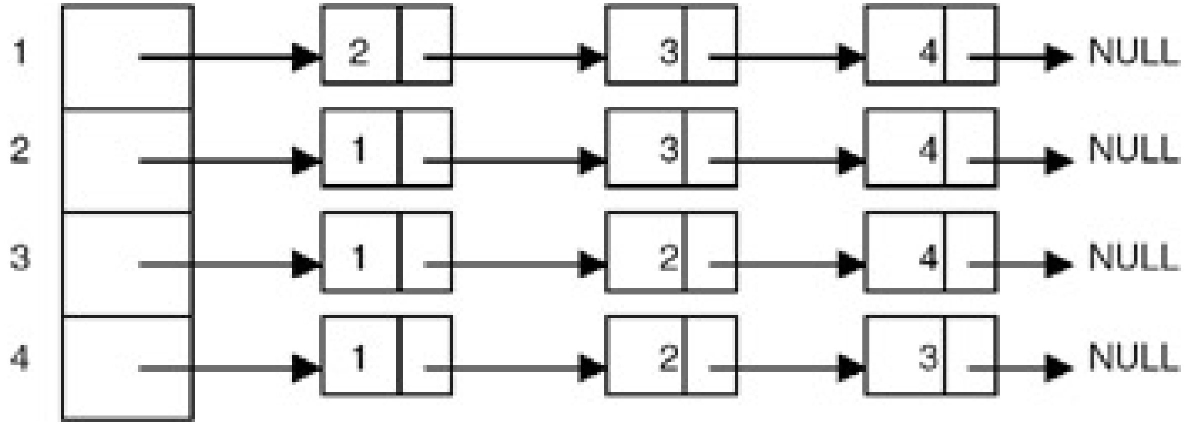
Adjacency List

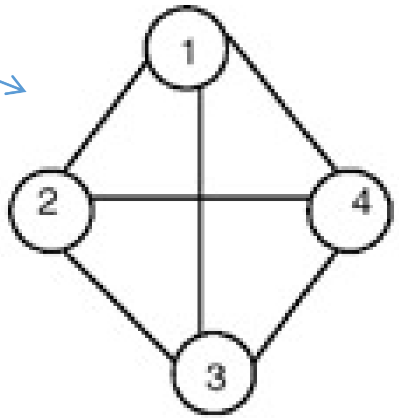


Adjacency List

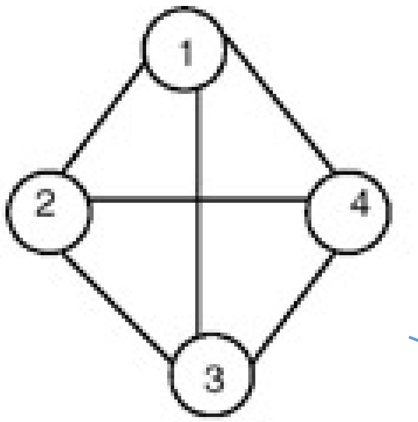


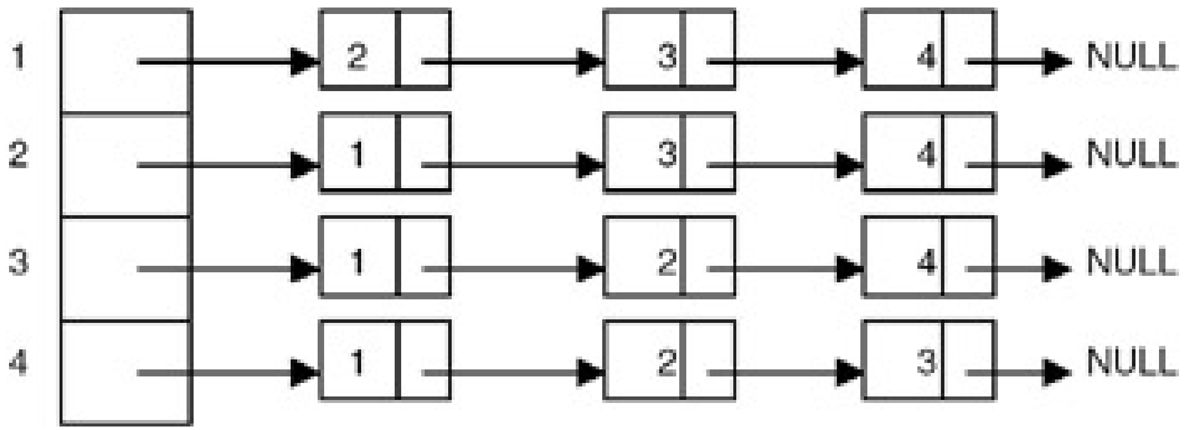
2. Given the Linked List representation create a Graph





2. Show the Linked List representation of the given Graph





TRAVERSAL ALGORITHMS

- Depth First Search (DFS) Algorithm
- Breadth First Search (BFS) Algorithm

Unlike a tree, a graph is not required to have a specific ordering of data

These algorithms start at some node in the graph and then `visit' all those nodes that are reachable from the start node.

Depth First Search (DFS) Algorithm

- Depth first search for graphs is the same as that for trees except that, because of cycles, it must make sure **not to visit a node twice**.
- We assume that all nodes have been initially marked `unvisited' before DFS(u) is first called.
- **Depth first search will visit those nodes that are reachable from the initial node**.
- **This process is repeated recursively and when all new nodes have been visited, the previous node exploration continues**.
- **The search terminates when all reachable nodes have been visited.**

Depth First Search (DFS) Algorithm

dfs(vertices, start)

Input: The list of all vertices, and the start node.

Output: Traverse all nodes in the graph.

Begin

  initially make the state to unvisited for all nodes

  push **start** into the **stack**

  while **stack** is not empty, do

    pop element from stack and set to u

    display the node u

    if u is **not visited**, then

      mark u as visited

      for all nodes i connected to u, do

        if ith vertex is unvisited, then

          push ith vertex into the stack

          mark ith vertex as visited

      done

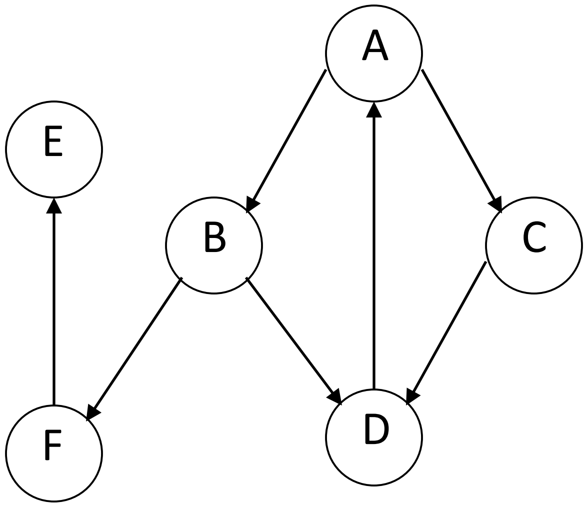
  done

End

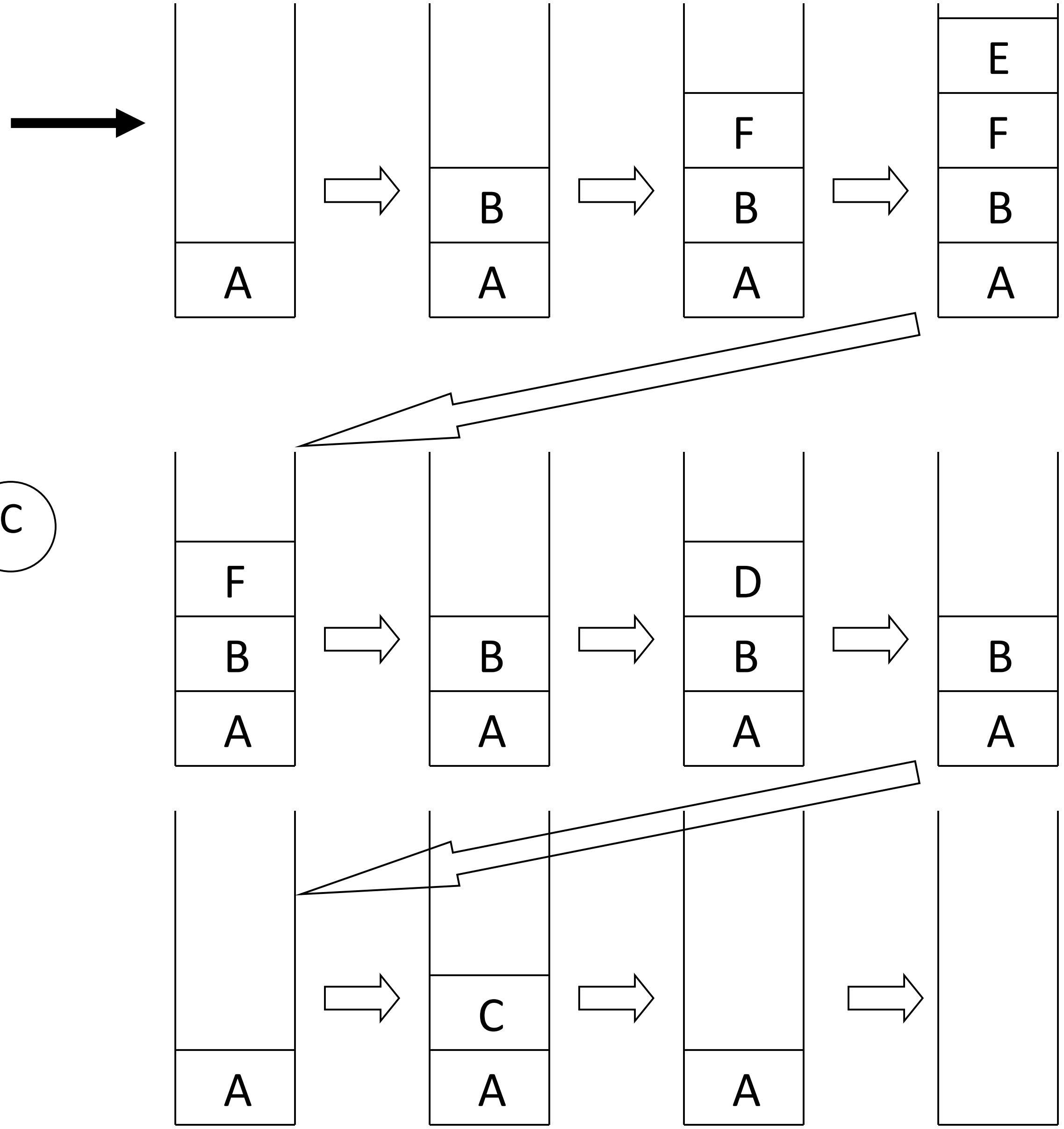
Depth First Search - Recursion

Program Stack

Given: **Start node is A**



depthFirst(A)

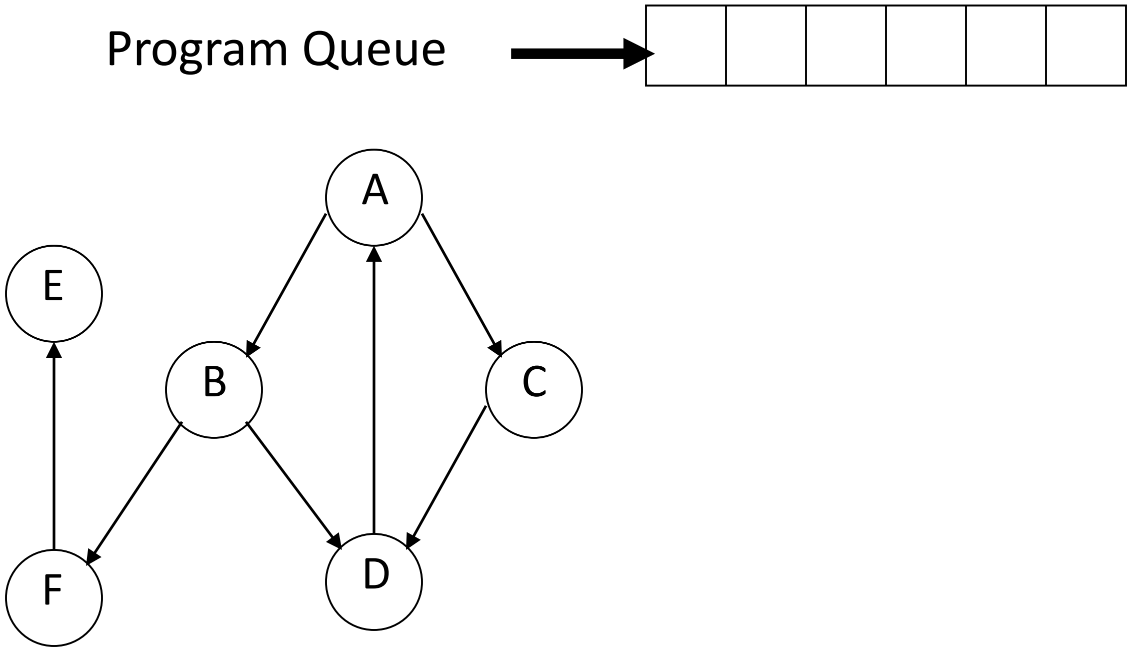


Breadth First Search (BFS) Algorithm

- Breadth first search for graphs is the same as that for trees except that, because of cycles, it must make sure not to visit a node twice.
- In Breadth First Search we start at vertex **v** and mark it as having been reached.
- All unvisited vertices adjacent from **v** are visited next.
  - use a **queue** to store each visited node's adjacent nodes
  - start by Insert/ enqueueing the root
  - Delete/dequeue the first node from the queue
    - **visit the node**
  - Insert/enqueue all of the nodes adjacent nodes
    - be careful not to enqueue an already visited node
  - Delete/dequeue the next node from the queue and repeat the process

Breadth First Search

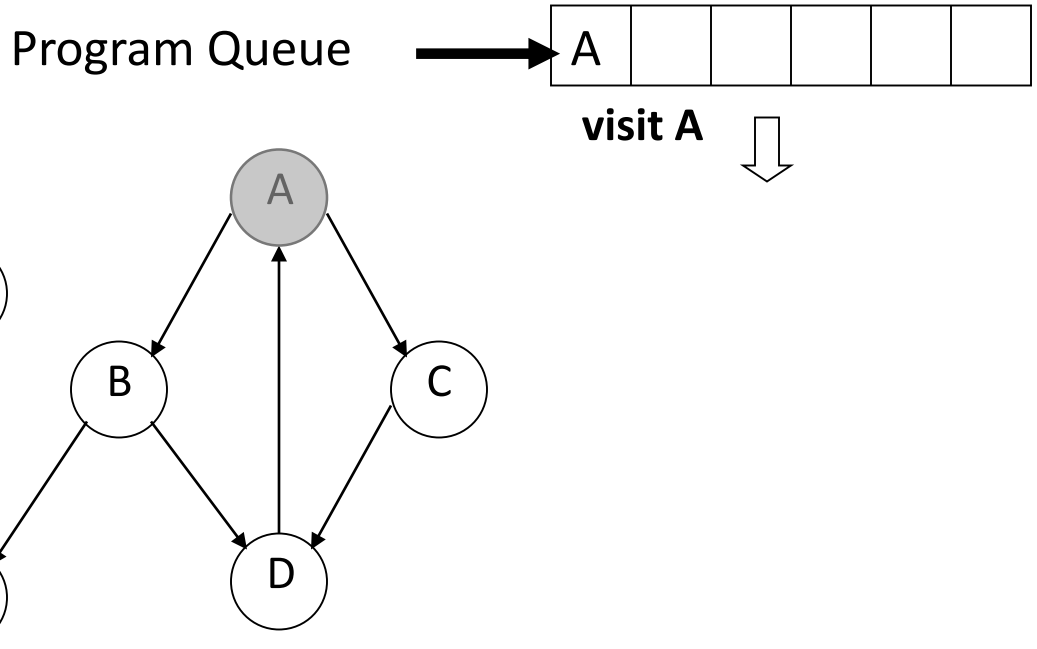
Step1: Initially queue and visited arrays are empty.



breadthFirst(A)

Breadth First Search

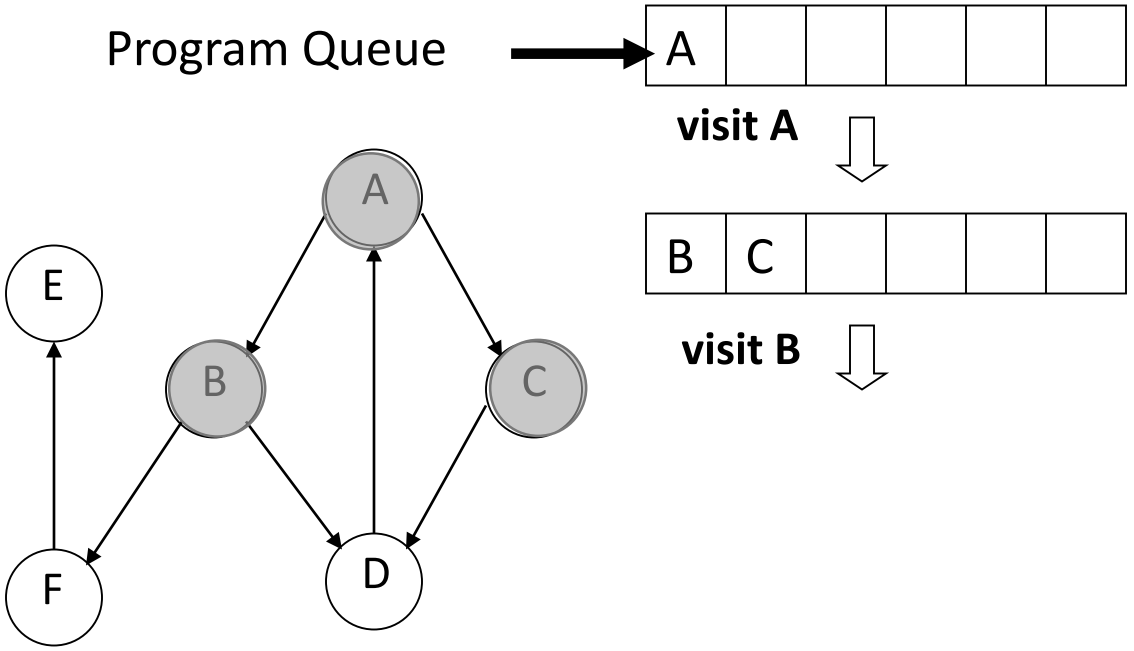
Step2: Push node 0 into queue and mark it visited.



breadthFirst(A)

Breadth First Search

Step 3: Remove node 0 from the front of queue and visit the unvisited neighbors and push them into queue.

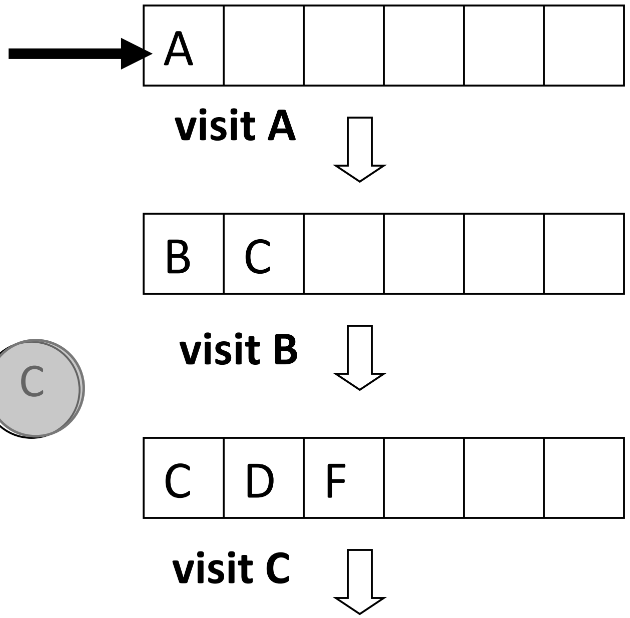


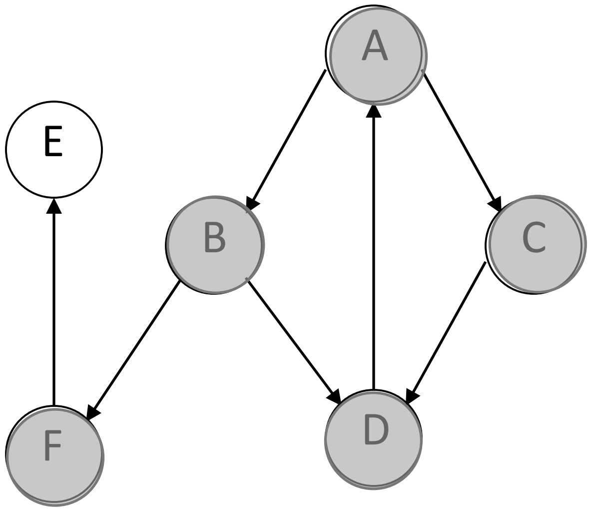
breadthFirst(A)

Breadth First Search

Step 4: Remove node 1 (Node B) from the front of queue and visit the unvisited neighbors and push them into queue.

Program Queue

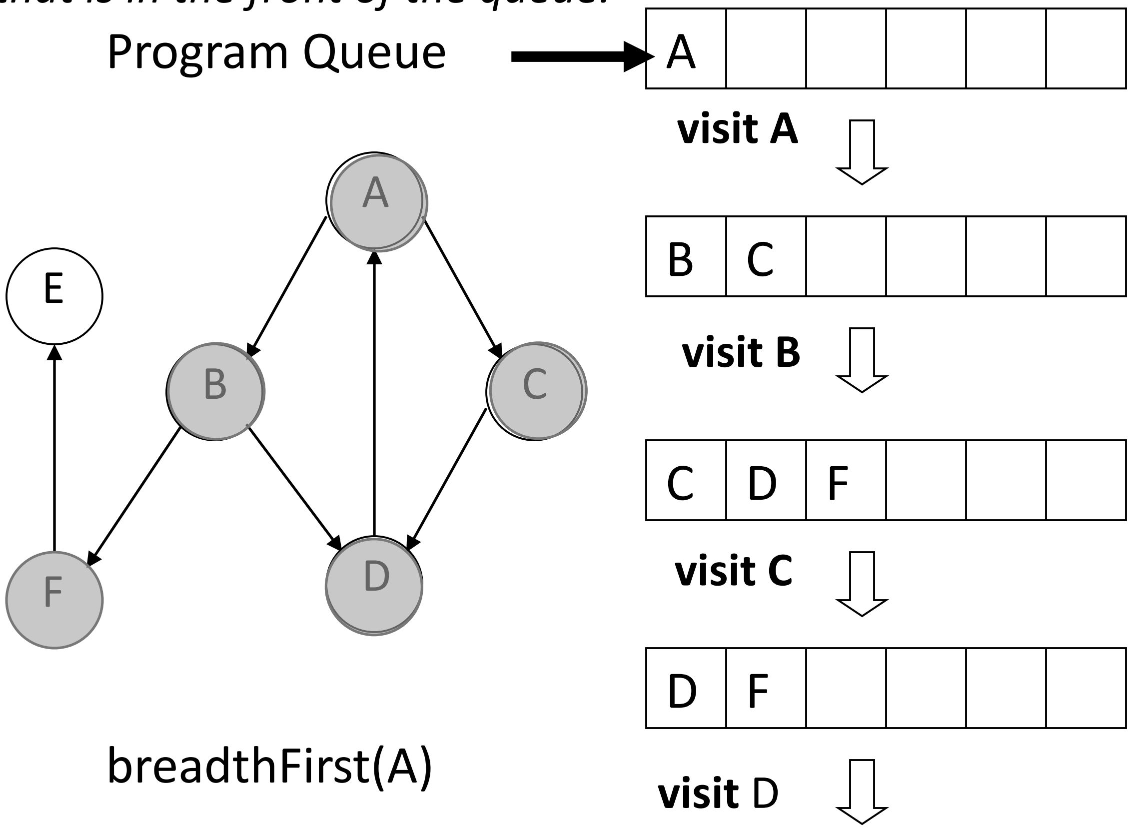




breadthFirst(A)

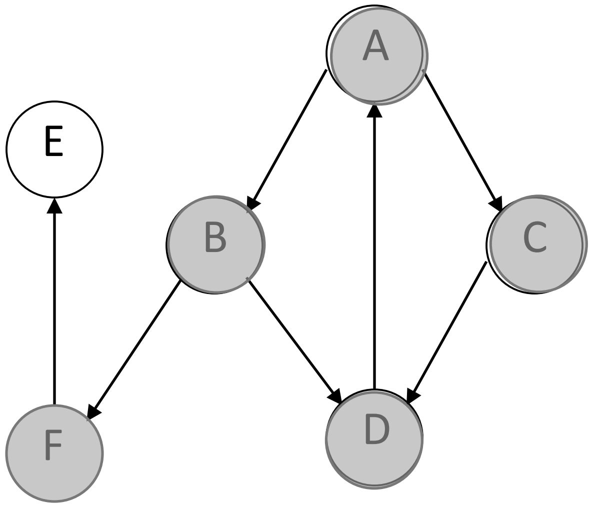
Breadth First Search

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbors (if Any) and push them into queue. In this case Node C (node 2) does not have any unvisited neighbors. *As we can see that every neighbors of node 2 are visited, so move to the next node that is in the front of the queue.*

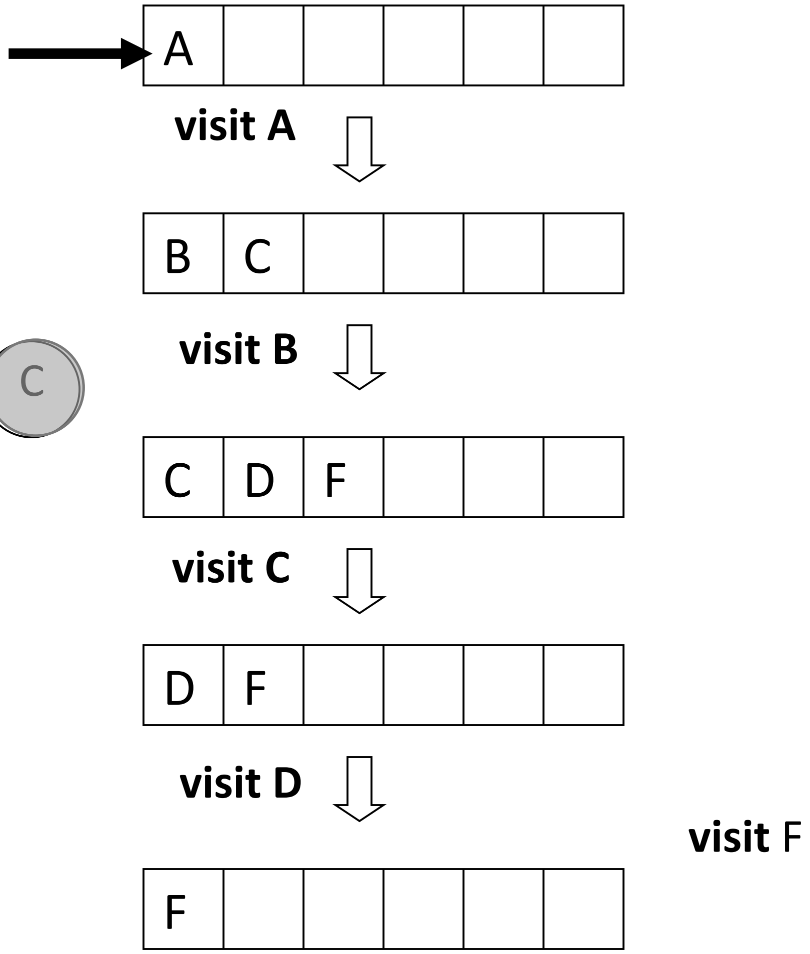


Breadth First Search

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbors (if Any) and push them into queue. In this case Node D (node 3) does not have any unvisited neighbors. *As we can see that every neighbors of node 3 are visited, so move to the next node that is in the front of the queue.*

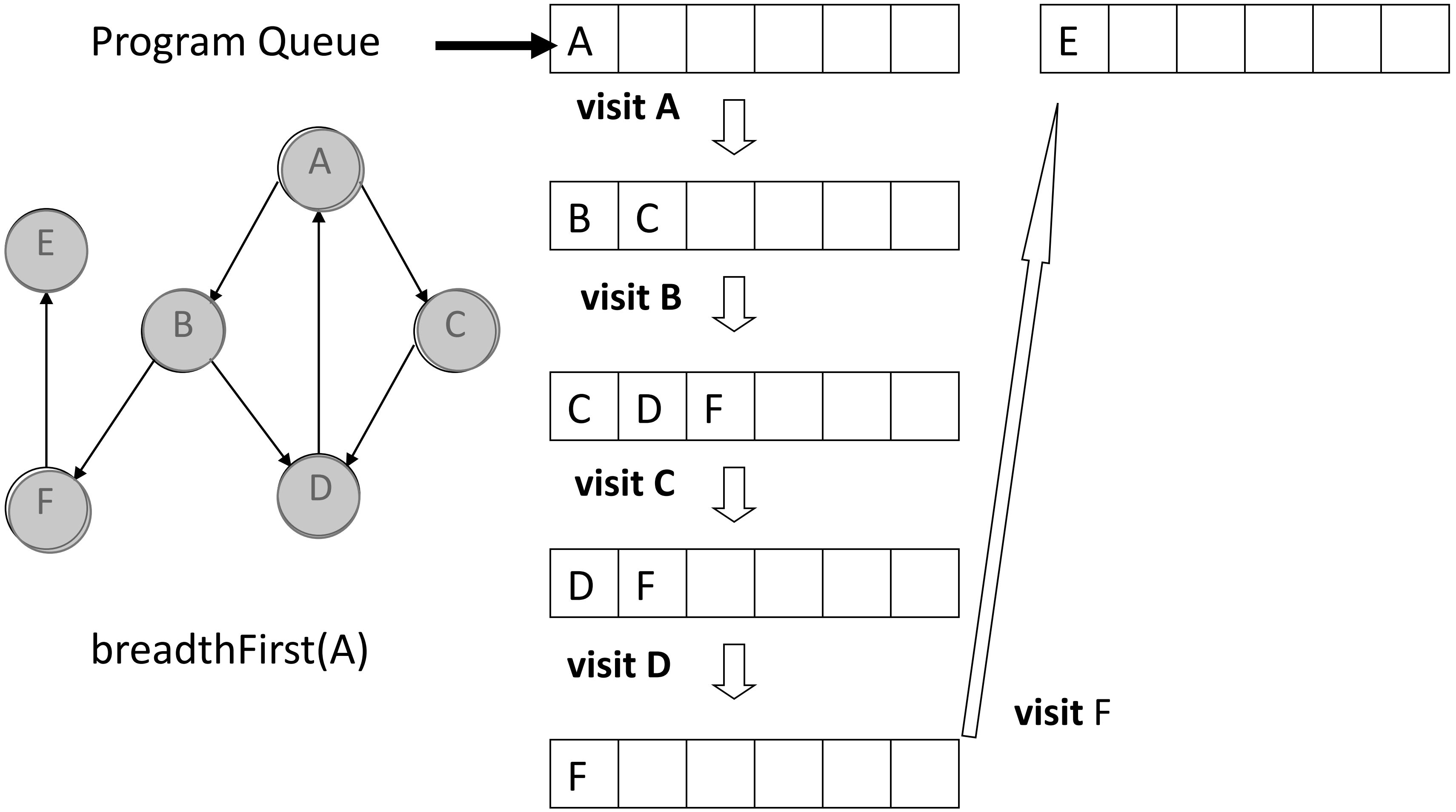


breadthFirst(A)



Breadth First Search

Step 7: Remove node 4 (Node F) from the front of queue and visit the unvisited neighbors (if Any) and push them into queue.



breadthFirst(A)

Breadth First Search

Step 8: Remove node 5 (Node E) from the front of queue and visit the unvisited neighbors (if Any) and push them into queue. *As we can see there are no neighbors of Node 5 (Node E), Queue becomes empty, So, terminate this process of iteration.*