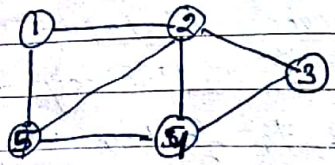
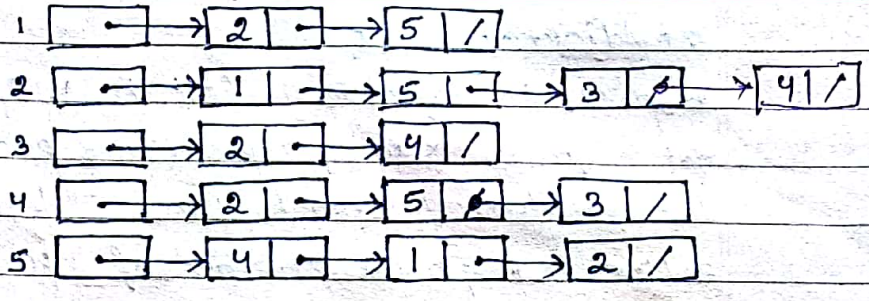
Sem III

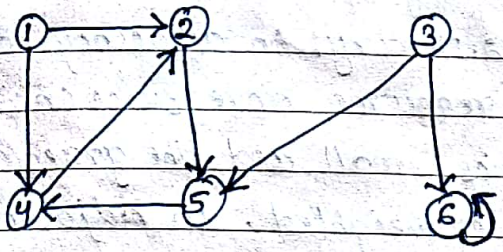
GRAPH ALGORITHM

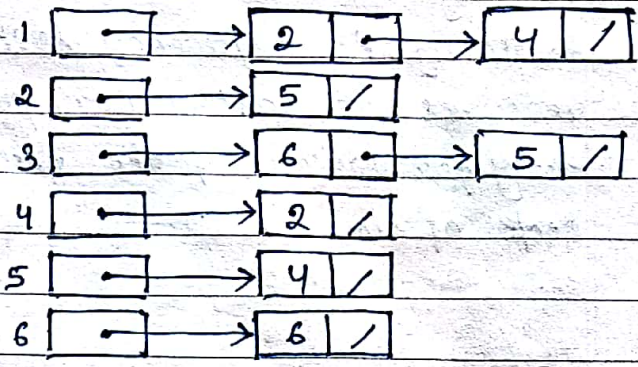
→ The pictorial representation of any object in a 2D plane is called a graph.
→ Graph is a collection of vertices and edges where vertices are connected by edges.
→ A graph G=(V,E) can be represented by two ways.
(1) Adjacency lists.
(2) Adjacency Matrix.
→ A simple path is a path with no vertex repeated.
→ A simple cycle is a simple path except the first and last vertex is repeated. For an undirected graph, number of vertices >=3.
→ A tree is a graph with no cycles.
→ A complete graph is a graph in which all edges are present.
→ A sparce graph is a graph with relatively few edges.
→ A dense graph is a graph with many edges.
→ In sparce graph |E| is much less than $|V|^2$
→ In dense graph |E| is close to $|V|^2$.
→ An undirected graph has no specific direction between the vertices.
→ A directed graph has edges that are "one way". We can go from one vertex to another, but not vice versa.
→ A weighted graph has weights associated with each edge. The weight can represent distances, costs etc.
→ The advantage of dense graph is that it will take less memory space.

(1) Adjacency list: An adjacency list is an array, which contains a list for each vertex.
→ Two vertices are said to be adjacent if ~~they are~~ their end points are of same edge.
→ The list for a particular vertex contains all other vertices that are connected to that vertex.
→ If G is a graph then the sum of the lengths of all the adjacency list is 2|E|





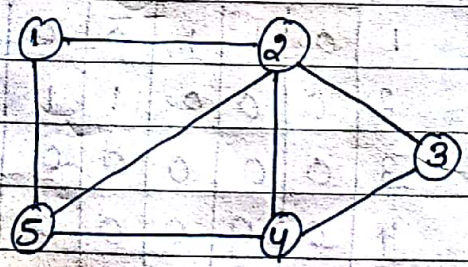




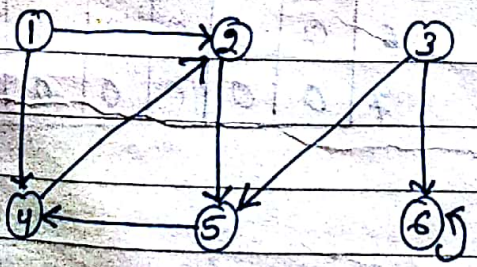
Adjacency Matrix

→ An adjacency matrix is a matrix with a row and column for each vertex.

→ The matrix entry is 1 if there is an edge between the row vertex and the column vertex. Each edge is represented twice.



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



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

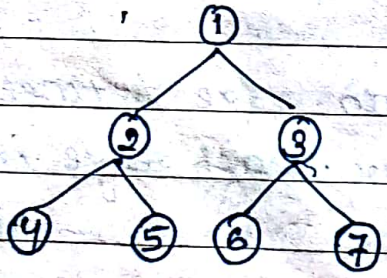
Adjacency list

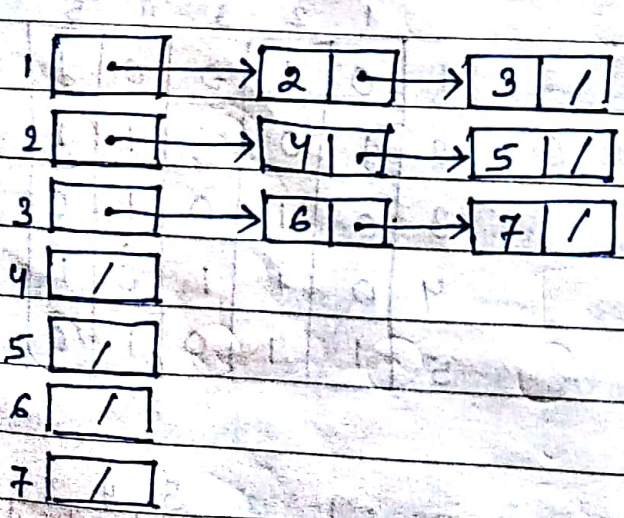
1. Speed: An adjacency list is not a faster way of representation. It takes time to search the entire list for the presence of the edge.
2. Memory: An adjacency list requires at least 1 word per entry for address of the next node of the list.

Adjacency Matrix

1. Speed: An adjacency matrix provides the faster way to determine whether or not a particular edge is present in the graph.
2. Memory: An adjacency matrix requires more space (n^2). If n is small and the graph is unweighted, an adjacency matrix only needs 1 bit per entry.

Q Give an adjacency list and adjacency matrix representation of a complete binary tree on 7 vertices.





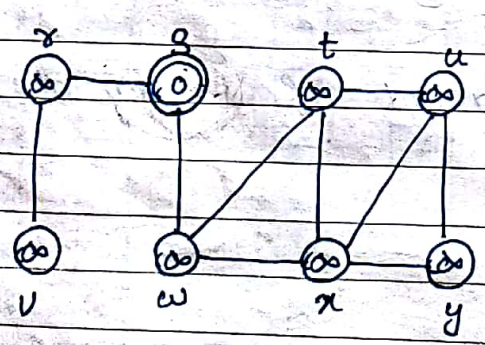
	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

BREADTH-FIRST SEARCH

→ BFS is used to search a graph.
→ A graph $G = (V, E)$ and having a source vertex s. The BFS systematically explores the edges of G to discover every vertex that is reachable from s.
→ It computes the distance from s to each reachable vertex.
→ It produces a breadth-first tree with root s that contains all reachable vertices.
→ BFS algorithm works for both directed and undirected graphs.
→ The algorithm discovers all vertices at distance k from s before discovering any vertices at distance k+1.
→ To keep track of progress. BFS colors each vertex white, gray or black.
→ All vertices start out white and may later become gray and then black.
→ Breadth first search constructs a breadth-first tree, initially containing only its root which is the source vertex s.
→ whenever a white vertex v is discovered while scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree.
→ u is the predecessor or parent of v in the BFT.

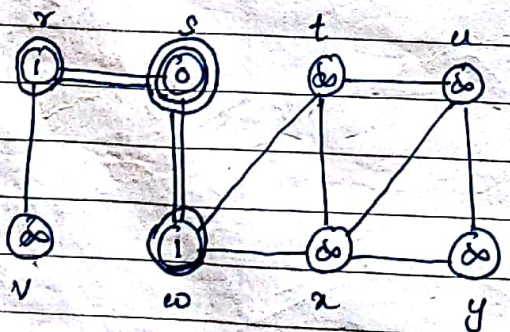
```
BFS(G,S)
1. for each vertex u ∈ V[G] - {s}
2:    do color[u] ← WHITE
3.       d[u] ← ∞
4.       π[u] ← NIL
5. color[s] ← GRAY
6. d[s] ← 0
7. π[s] ← NIL
8. Q ← ∅
9. ENQUEUE(Q,s)
10. while Q ≠ ∅
11.      do u ← DEQUEUE(Q)
12.         for each v ∈ Adj[u]
13.             do if color[v] = WHITE
14.                then color[v] ← GRAY
15.                     d[v] ← d[u] + 1
16.                     π[v] ← u
17.                     ENQUEUE(Q,v)
18.         color[u] ← BLACK
```

step-1



Q | s |
0

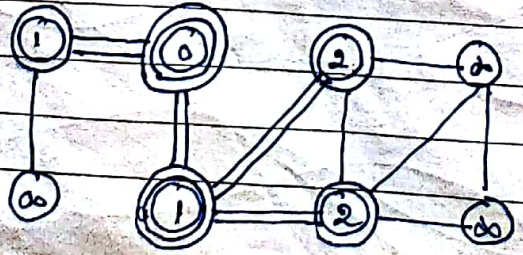
step-2



Q | w | r |
1 1

Visited node [s]

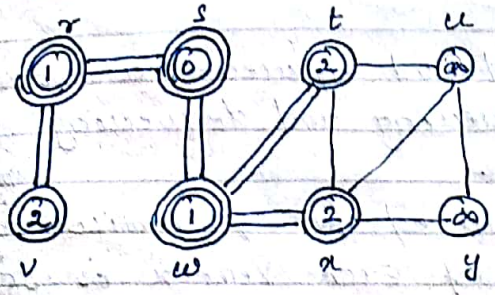
step-3



Q | r | t | x |
1 2 2

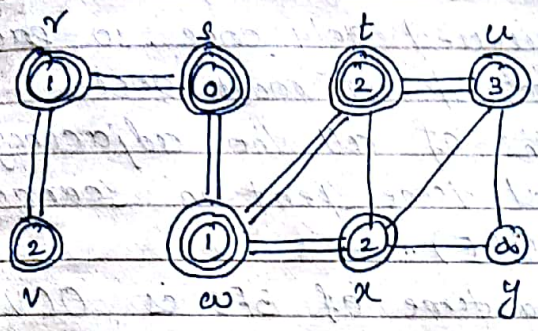
[s, w]

Step-4



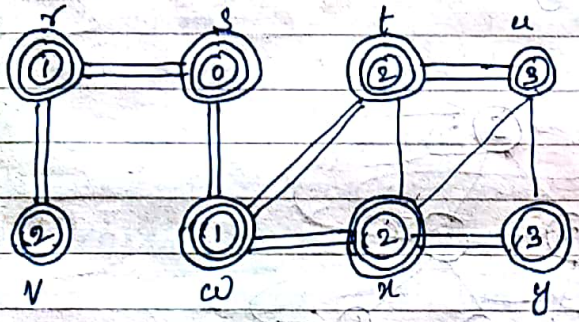
Q | t | x | v | [s, w, r]
2 2 2

Step-5



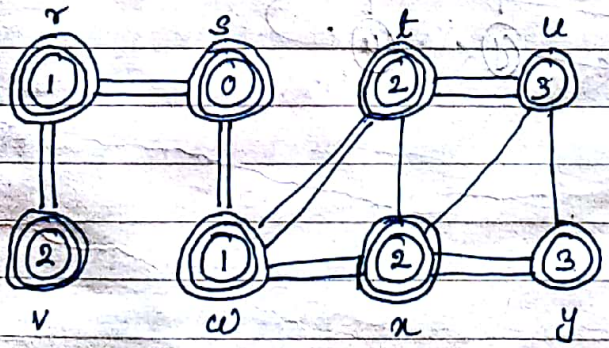
Q | x | v | u | [s, w, r, t]
2 2 3

Step-6



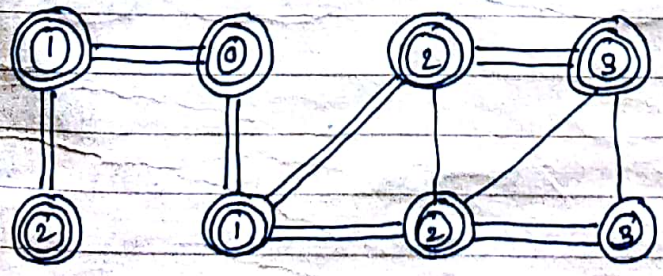
Q | v | u | y | [s, w, r, t, x]
2 3 3

Step-7



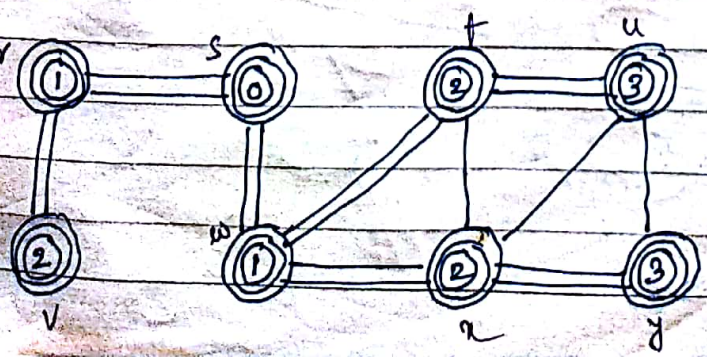
Q | u | y | [s, w, r, t, x, v]
3 3

Step-8



Q | y | [s, w, r, t, x, v, u]
3

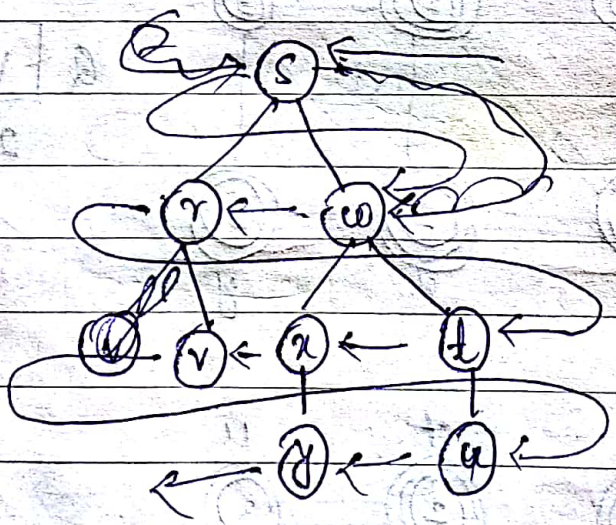
Step-9



Q | ∅ | [s, w, r, t, x, v, u, y]

Analysis Of BFS

→ Each vertex is enqueued and dequeued at most once.
→ The operations of enqueuing and dequeuing of single node takes O(1) time.
→ So, the total time required for V vertices are O(V)
→ The adjacency list of each vertex is scanned only when the vertex is dequeued.
→ Every vertex is dequeued only once, so each adjacency list is scanned at most once.
→ The sum of the lengths of all the adjacency list is Q(E), so the total time spent in scanning adjacency lists is O(E).
→ The total ~~time~~ running time of BFS is O(V+E).



DEPTH FIRST SEARCH

→ In DFS, edges are explored out of the most-recently discovered vertex v that still has unexplored edges leaving it.
→ When all the edges of node v have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered.
→ This process continues until we have discovered all the vertices that are reachable from the original source vertex.
→ If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source.
→ This entire process is repeated until all vertices are discovered.
→ In DFS, each vertex v has two timestamps:
(1) Discovery time d[v] records when v is first discovered (which is in grey colored)
(2) finish time f[v] records when the search finishes examining v's adjacency list i.e. black in color
→ The timestamps are integers between 1 and 2|V|
→ For every vertex d[u] < f[u]
→ vertex u is WHITE before time d[u], GRAY between time d[u] and time f[u] and BLACK thereafter.

```
DFS(G)
1. for each vertex u ∈ V[G]                 } O(V)
2.     do color[u] ← WHITE
3.        π[u] ← NIL
4. time ← 0
5. for each vertex u ∈ V[G]                 } O(V)
6.     do if color[u] = WHITE
7.        then DFS-VISIT(u)
```

DFS-VISIT(u)

1. color[u] ← GRAY ▷ white vertex u has just been discovered.
2. time ← time + 1
3. d[u] ← time
4. for each v ∈ Adj[u] ▷ Explore edge (u, v)
5. do if color[v] = WHITE
6. then π[v] ← u
7. DFS-VISIT(v)
8. color[u] ← BLACK ▷ Blacken u; it is finished.
9. f[u] ← time ← time + 1

$\Theta(V)$ (lines 1–3 and 8–9)

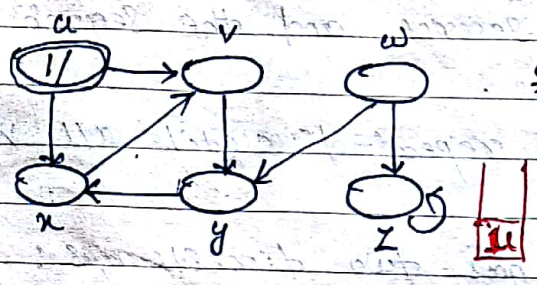
$\Theta(E)$ (line 4)

$\Theta(V)$ (lines 5–7)

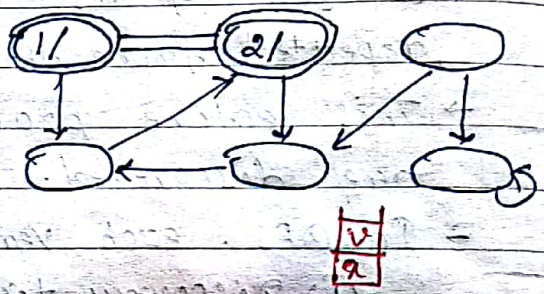
$\Theta(E)$ (lines 4–7)

$\Theta(V+E)$

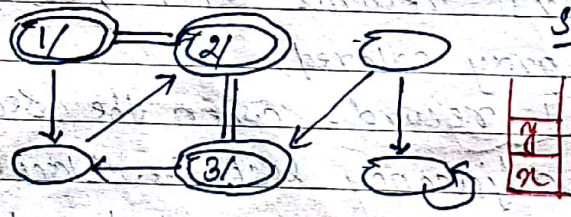
step-1



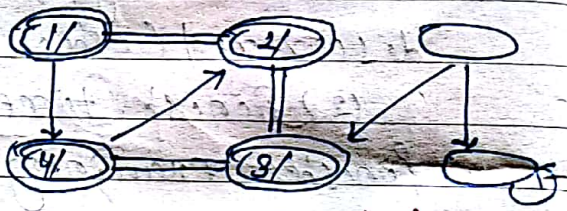
step-2



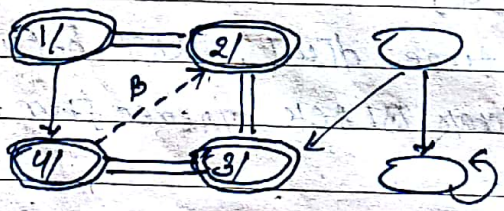
step-3



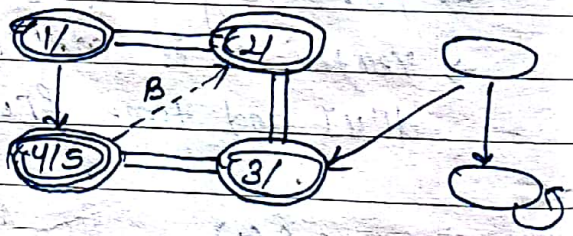
step-4



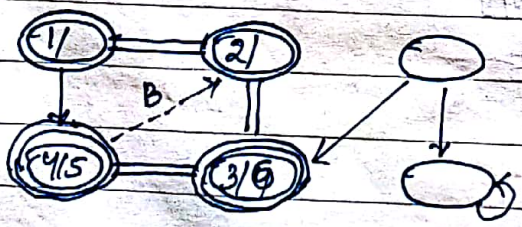
step-5



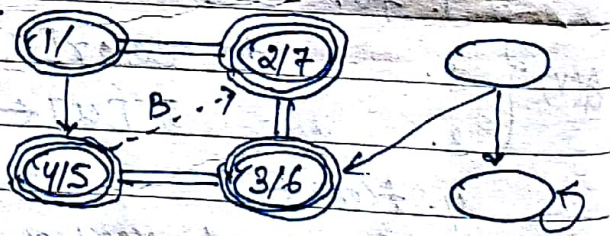
step-6



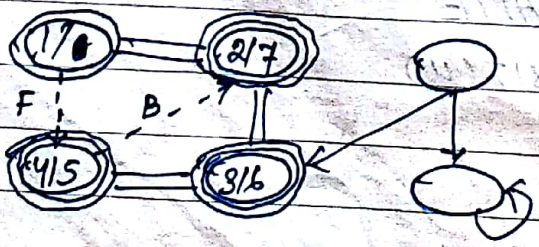
step-7



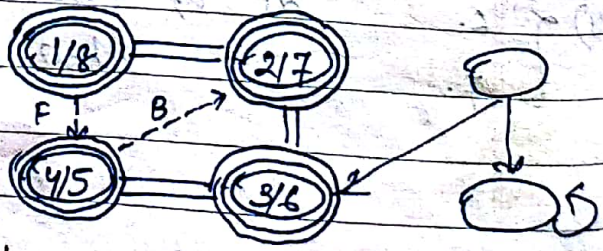
step-8

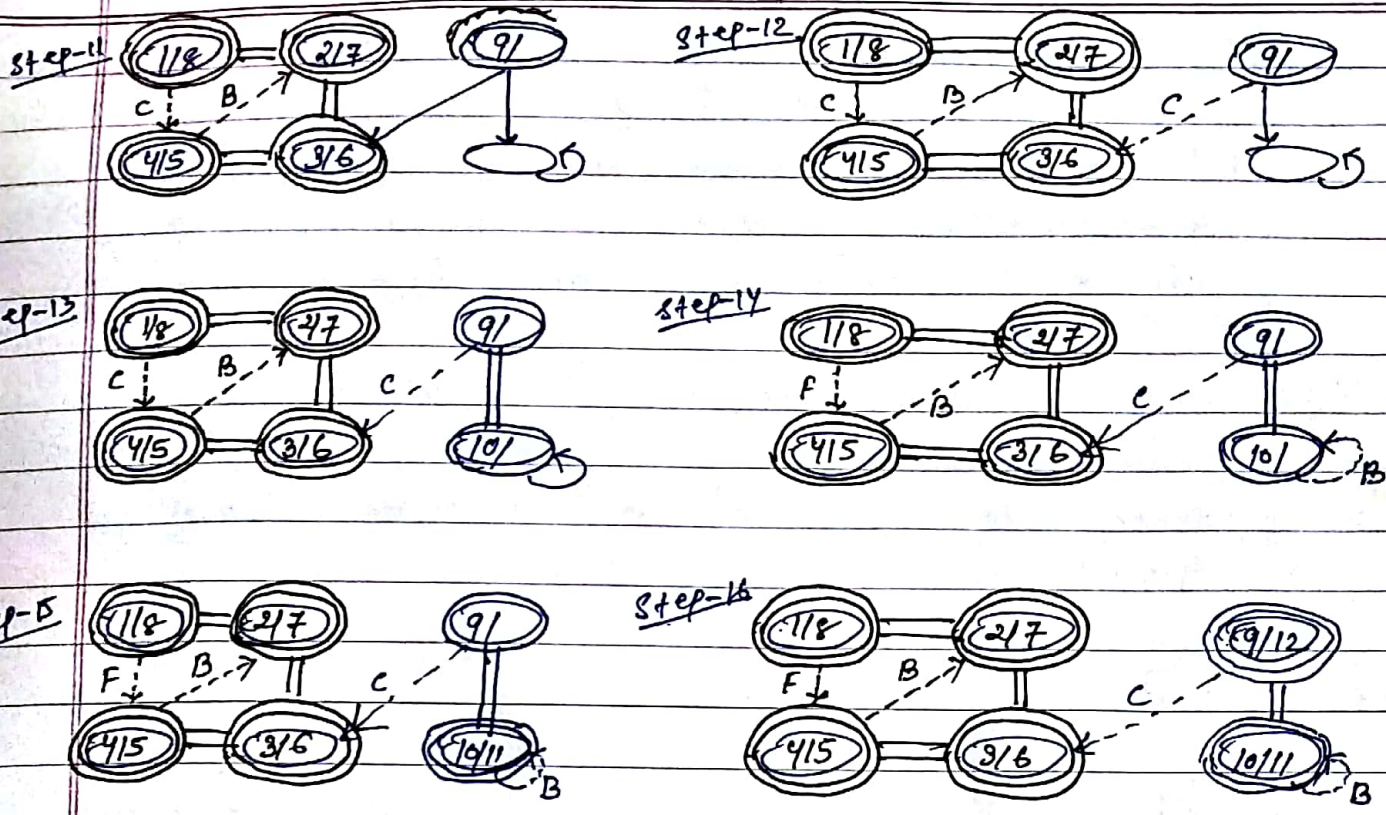


step-9



step-10





Complexity:

→ The loop on line 1-3 and line 5-7 of DFS takes time $\Theta(V)$.

→ The procedure DFS-VISIT is called exactly once for each vertex $v \in V$. Since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray.

→ During an execution of DFS-VISIT(v), the loop on lines 4-7 is executed $|Adj[v]|$ times

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

So, total cost of executing lines 4-7 of DFS-VISIT is $\Theta(E)$.

The running time of DFS is $\Theta(V+E)$.