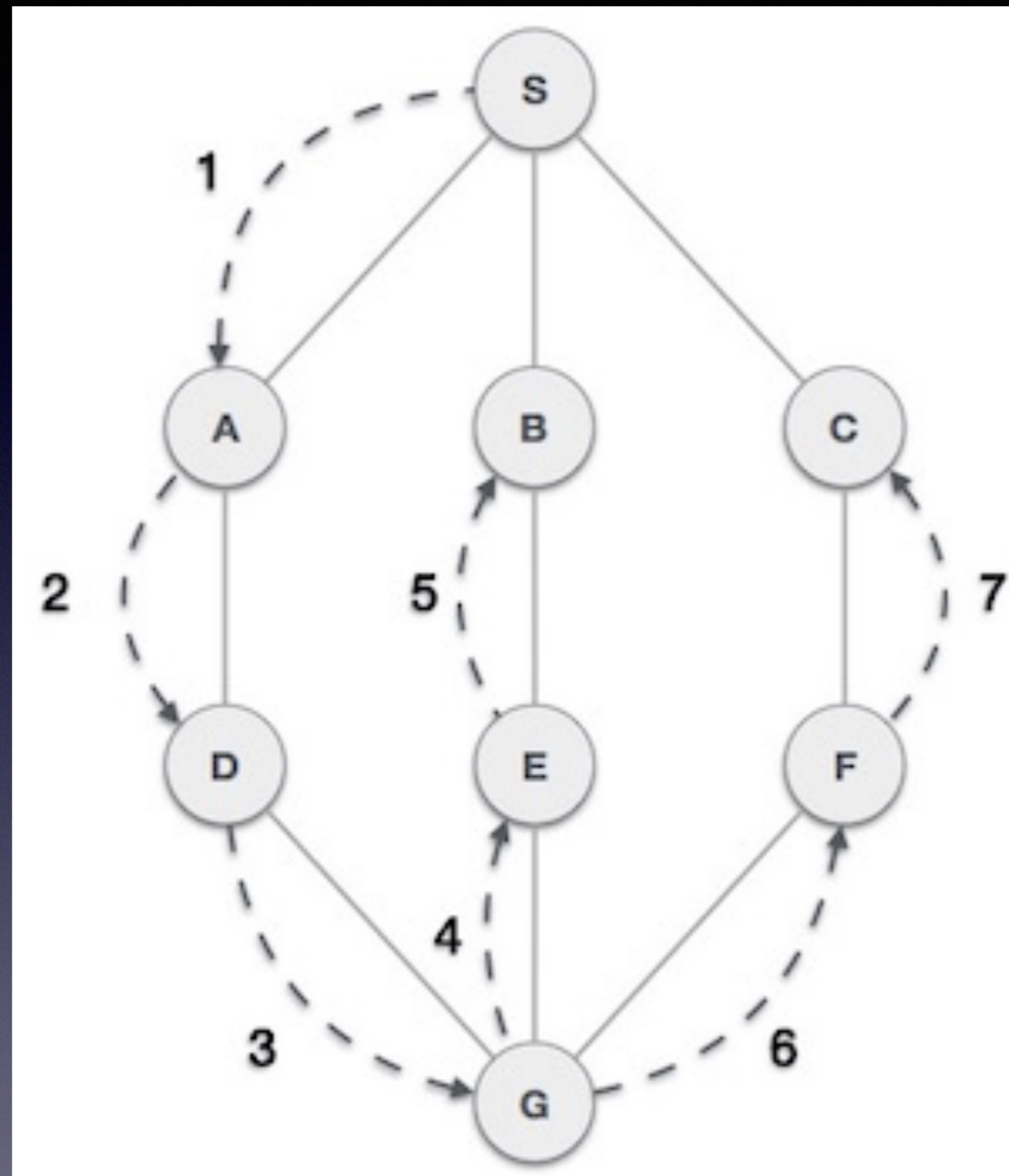


# Data Structures and Algorithms

Depth First Search and Breadth First Search

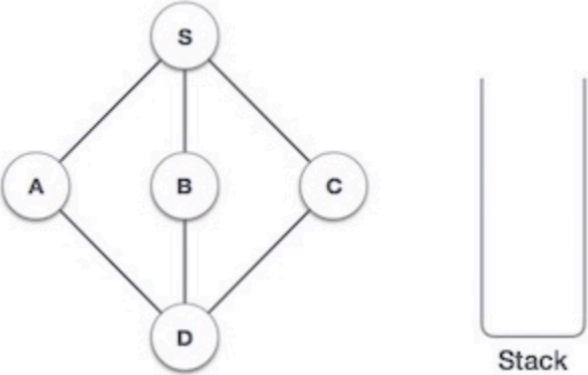
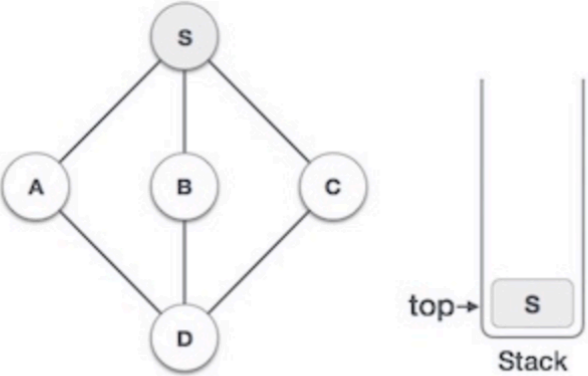
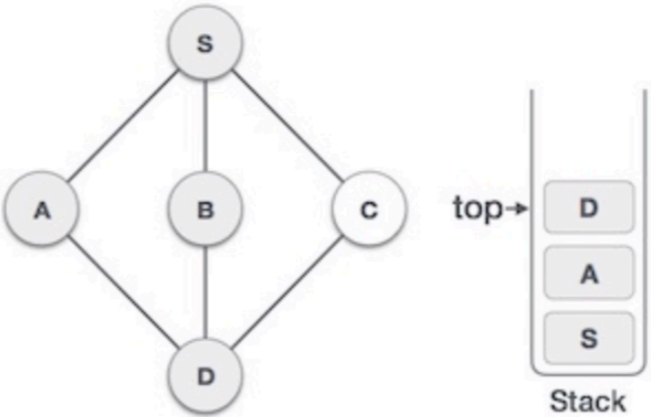
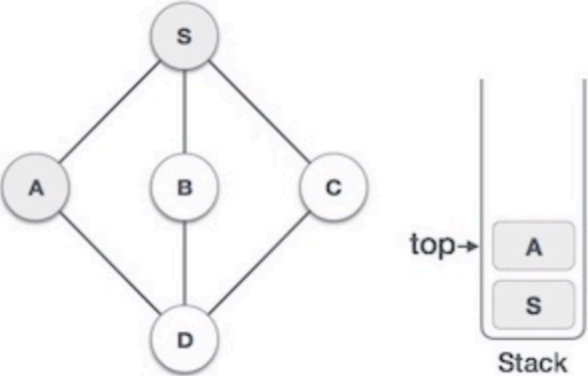
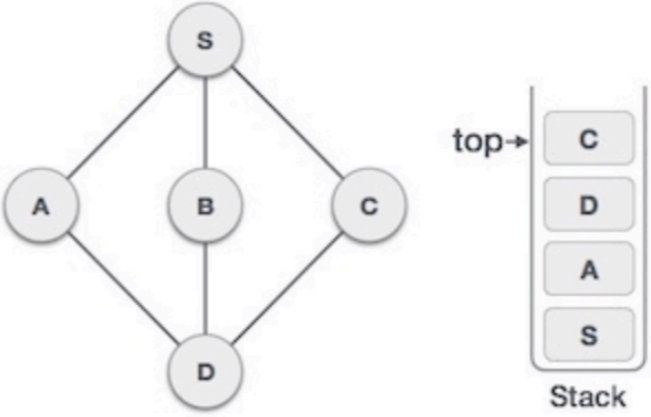
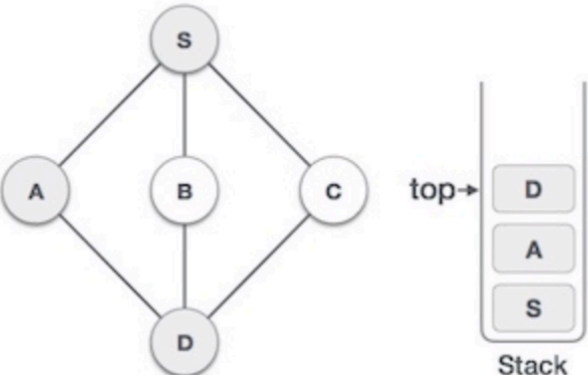


# Depth-First-Search

- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration

# DFS - Necessary Steps

- ▣ **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- ▣ **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- ▣ **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description	5	
1		Initialize the stack.		
2		Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b> . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.	6	
3		Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.	7	
4		Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are unvisited. However, we shall again choose in an alphabetical order.		

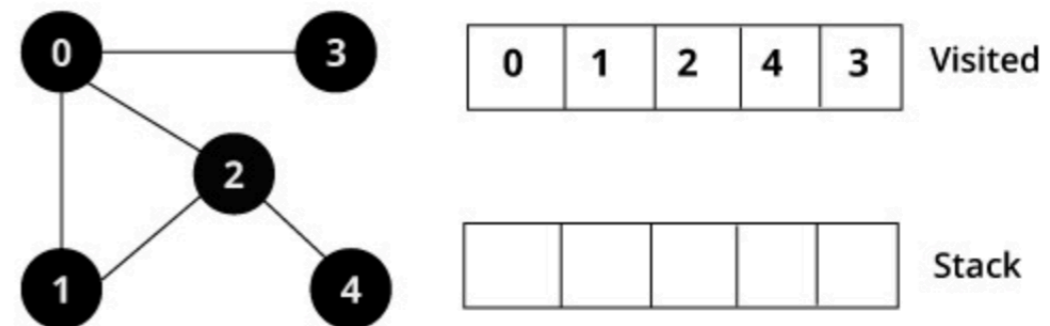
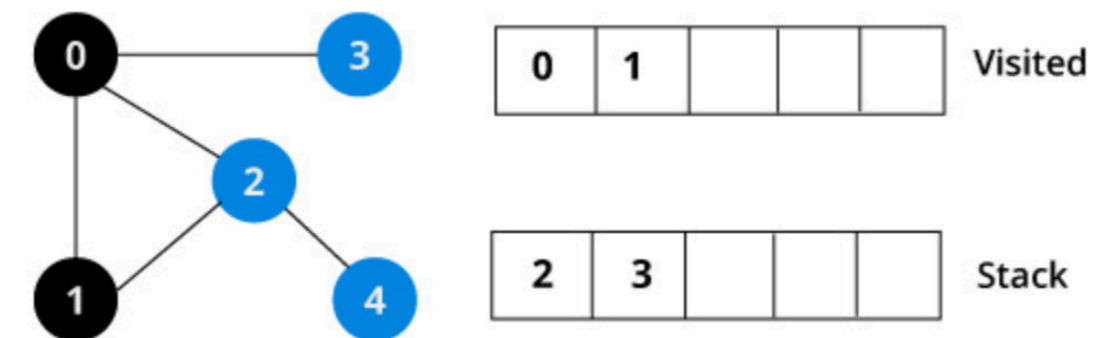
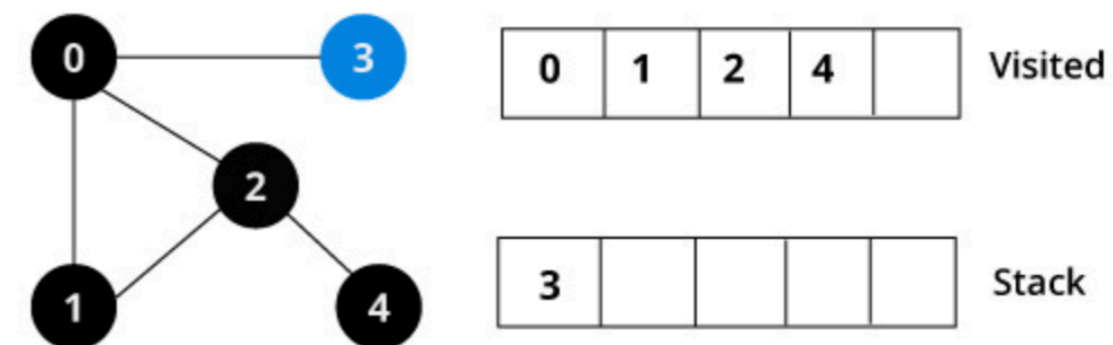
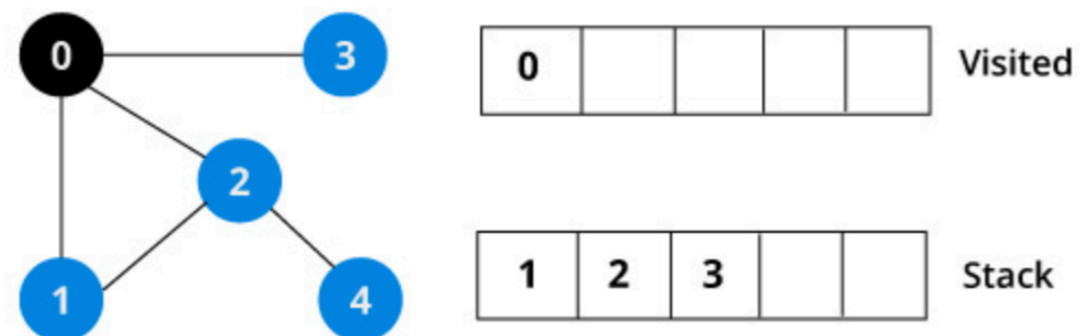
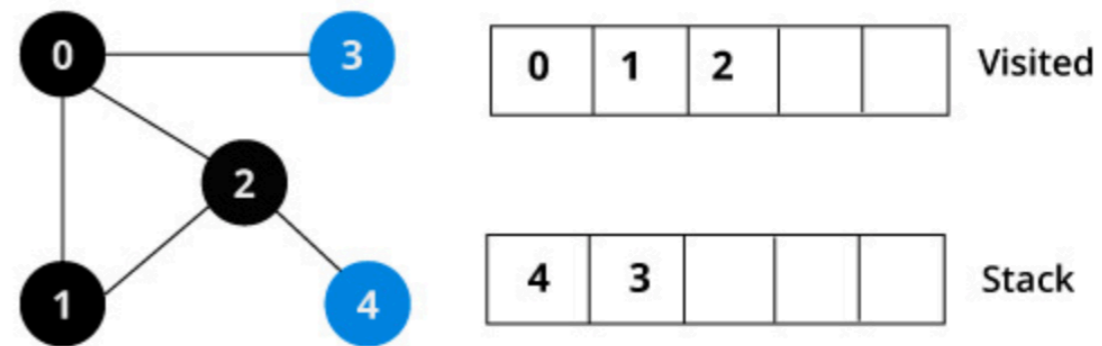
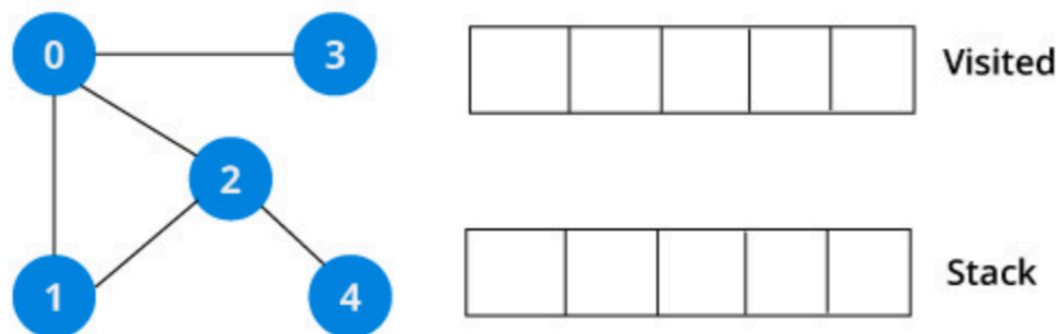
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

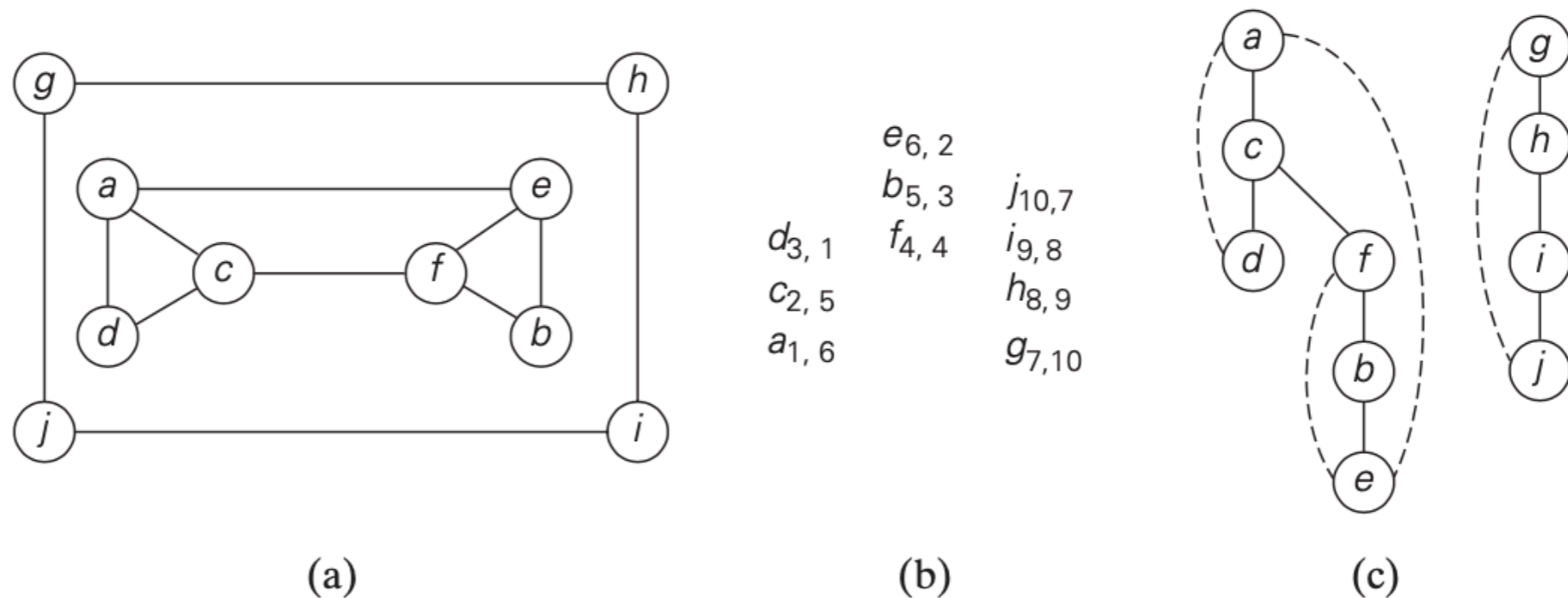
Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

# DFS - A Closer Look

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 stack.
4. Keep repeating steps 2 and 3 until the stack is empty.



# Depth First Search



**FIGURE 3.10** Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

# PseudoCode for DFS

**ALGORITHM**  $DFS(G)$ 

//Implements a depth-first search traversal of a given graph

//Input: Graph  $G = \langle V, E \rangle$

//Output: Graph  $G$  with its vertices marked with consecutive integers

// in the order they are first encountered by the DFS traversal

mark each vertex in  $V$  with 0 as a mark of being “unvisited”

$count \leftarrow 0$

**for** each vertex  $v$  in  $V$  **do**

**if**  $v$  is marked with 0

$dfs(v)$

$dfs(v)$

//visits recursively all the unvisited vertices connected to vertex  $v$

//by a path and numbers them in the order they are encountered

//via global variable  $count$

$count \leftarrow count + 1$ ; mark  $v$  with  $count$

**for** each vertex  $w$  in  $V$  adjacent to  $v$  **do**

**if**  $w$  is marked with 0

$dfs(w)$

# C code for DFS

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int vertex;
    struct node* next;
};

struct node* createNode(int v);

struct Graph
{
    int numVertices;
    int* visited;
    struct node** adjLists; // we need int** to store a two dimensional array. Similar
};

struct Graph* createGraph(int);
void addEdge(struct Graph*, int, int);
void printGraph(struct Graph*);
void DFS(struct Graph*, int);
```

```
int main()
{

    struct Graph* graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);

    printGraph(graph);

    DFS(graph, 2);

    return 0;
}
```

# C code for DFS

```
void DFS(struct Graph* graph, int vertex) {
    struct node* adjList = graph->adjLists[vertex];
    struct node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while(temp!=NULL) {
        int connectedVertex = temp->vertex;

        if(graph->visited[connectedVertex] == 0) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

struct node* createNode(int v)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
```

```
struct Graph* createGraph(int vertices)
{
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));

    graph->visited = malloc(vertices * sizeof(int));

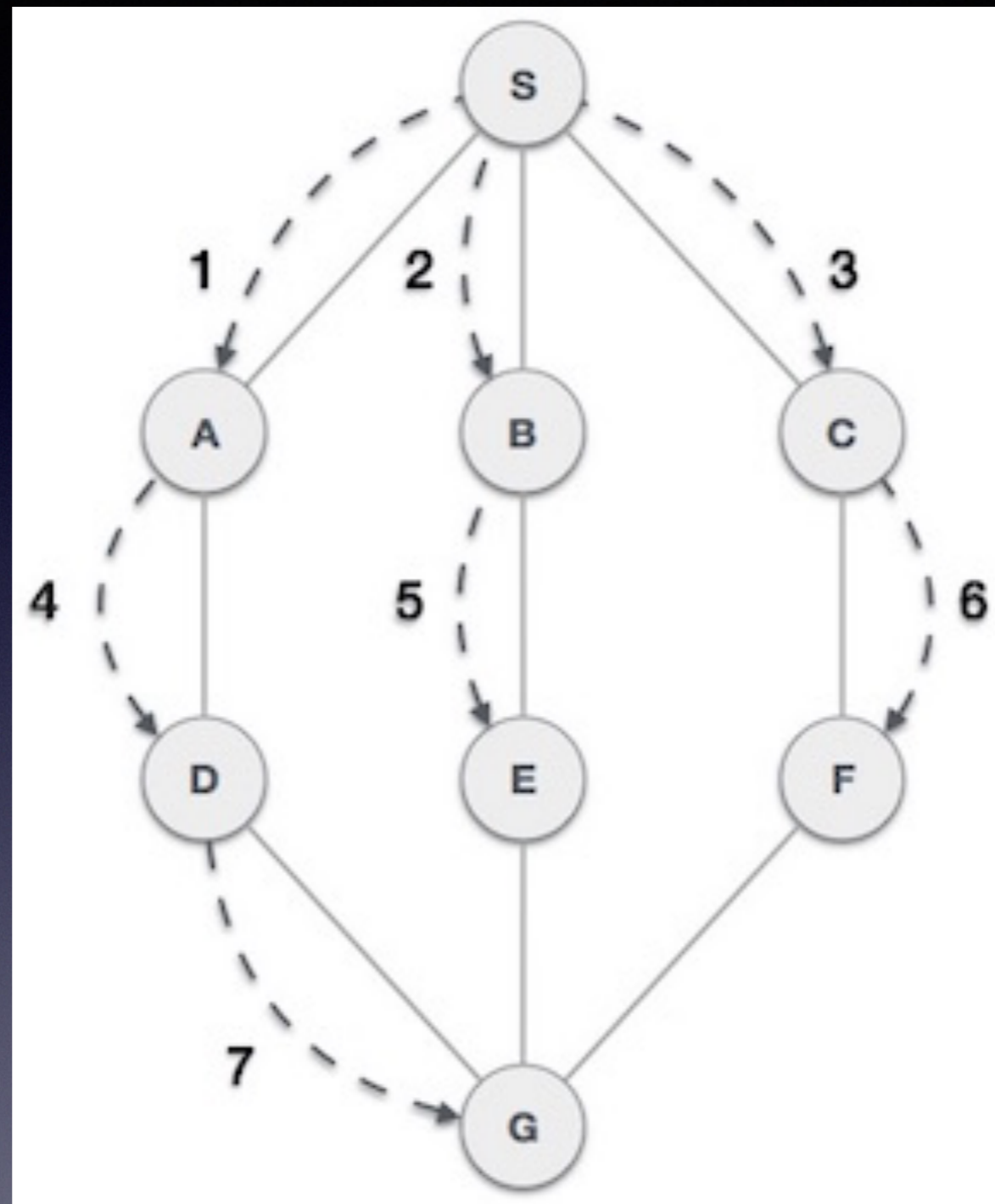
    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}
```

# C code for DFS

```
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
```

```
void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {
        struct node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
```

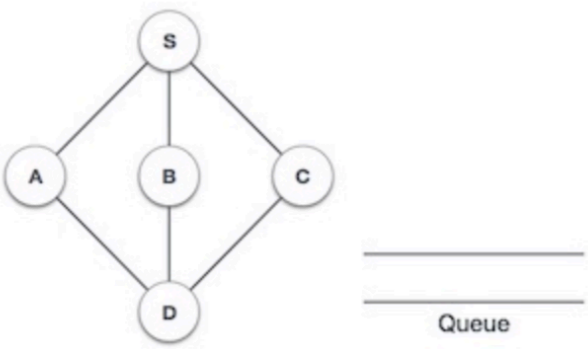
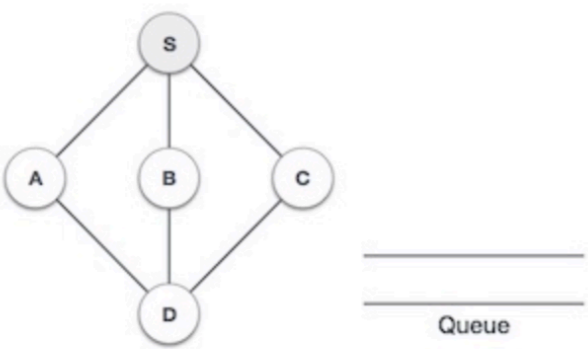
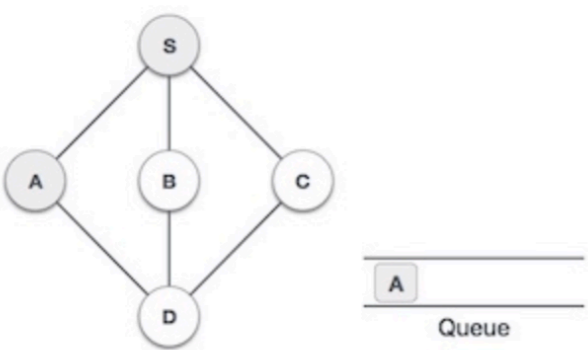
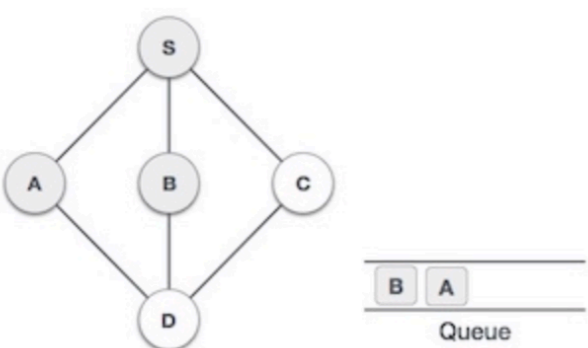
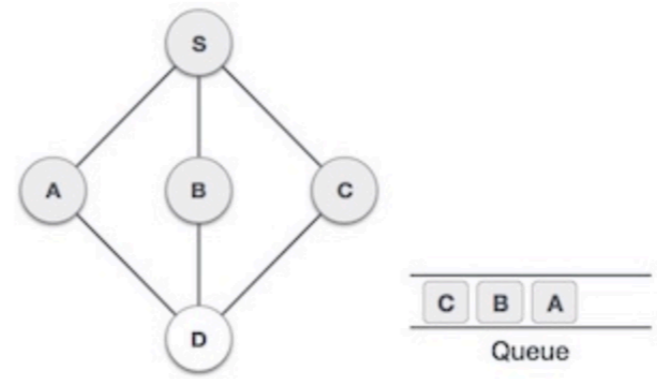
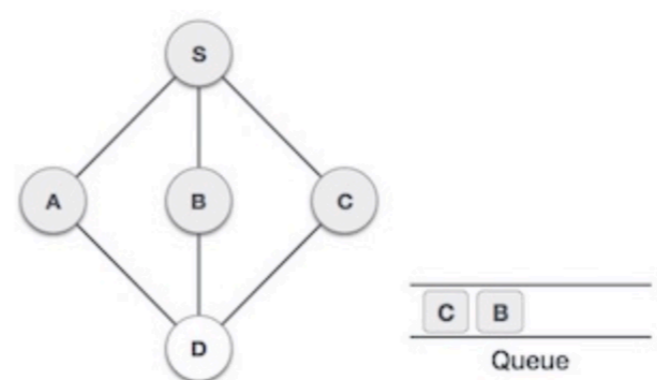
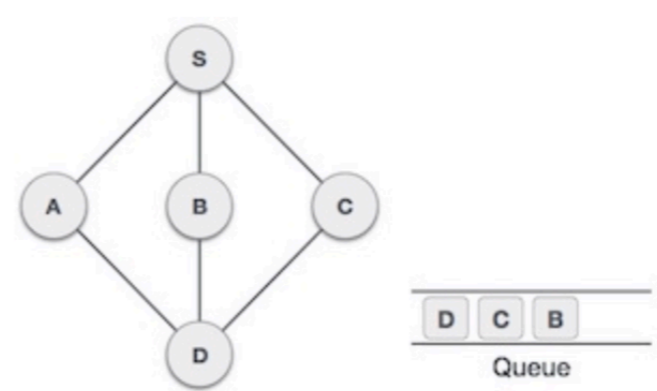


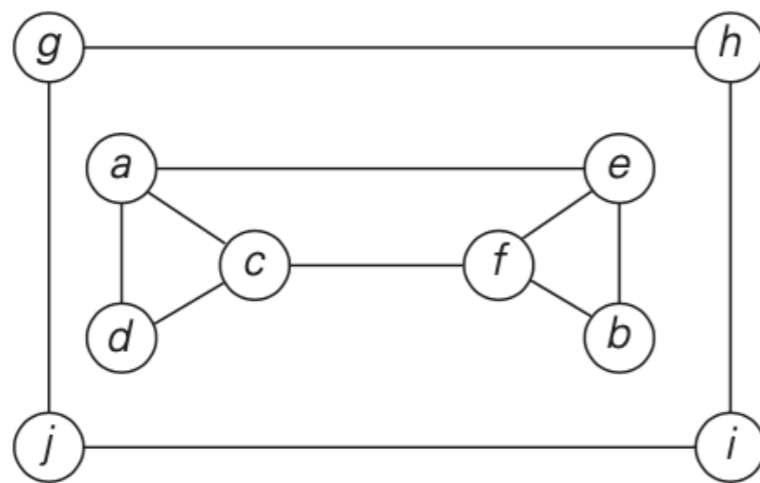
# Breadth-First-Search

- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

# BFS Algorithm

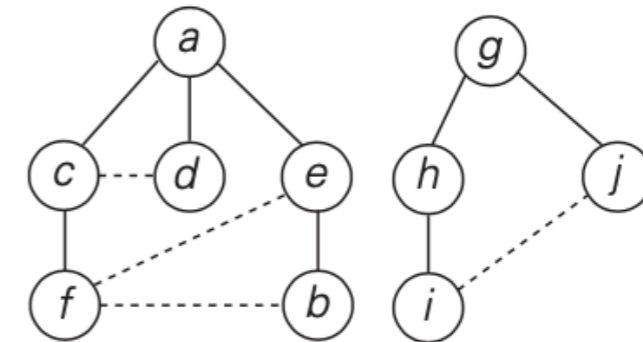
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting <b>S</b> (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> , mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from <b>S</b> is <b>B</b> . We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from <b>S</b> is <b>C</b> . We mark it as visited and enqueue it.
6		Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b> .
7		From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.



(a)

$a_1 c_2 d_3 e_4 f_5 b_6$   
 $g_7 h_8 j_9 i_{10}$



(c)

**FIGURE 3.11** Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

# PseudoCode of BFS Algorithm

**ALGORITHM** *BFS*(*G*)

//Implements a breadth-first search traversal of a given graph

//Input: Graph  $G = \langle V, E \rangle$

//Output: Graph  $G$  with its vertices marked with consecutive integers

// in the order they are visited by the BFS traversal

mark each vertex in  $V$  with 0 as a mark of being “unvisited”

*count*  $\leftarrow$  0

**for** each vertex  $v$  in  $V$  **do**

**if**  $v$  is marked with 0

*bfs*( $v$ )

*bfs*( $v$ )

//visits all the unvisited vertices connected to vertex  $v$

//by a path and numbers them in the order they are visited

//via global variable *count*

*count*  $\leftarrow$  *count* + 1; mark  $v$  with *count* and initialize a queue with  $v$

**while** the queue is not empty **do**

**for** each vertex  $w$  in  $V$  adjacent to the front vertex **do**

**if**  $w$  is marked with 0

*count*  $\leftarrow$  *count* + 1; mark  $w$  with *count*

            add  $w$  to the queue

        remove the front vertex from the queue

```

#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue {
    int items[SIZE];
    int front;
    int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node
{
    int vertex;
    struct node* next;
};

struct node* createNode(int);

```

```

struct Graph
{
    int numVertices;
    struct node** adjLists;
    int* visited;
};

struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void bfs(struct Graph* graph, int startVertex);

int main()
{
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);

    return 0;
}

```

```

void bfs(struct Graph* graph, int startVertex) {

    struct queue* q = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while(!isEmpty(q)){
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

        struct node* temp = graph->adjLists[currentVertex];

        while(temp) {
            int adjVertex = temp->vertex;

            if(graph->visited[adjVertex] == 0){
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

```

```

struct node* createNode(int v)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices)
{
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

```

```

void enqueue(struct queue* q, int value){
    if(q->rear == SIZE-1)
        printf("\nQueue is Full!!");
    else {
        if(q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

int dequeue(struct queue* q){
    int item;
    if(isEmpty(q)){
        printf("Queue is empty");
        item = -1;
    }
    else{
        item = q->items[q->front];
        q->front++;
        if(q->front > q->rear){
            printf("Resetting queue");
            q->front = q->rear = -1;
        }
    }
    return item;
}

```

```

void printQueue(struct queue *q) {
    int i = q->front;

    if(isEmpty(q)) {
        printf("Queue is empty");
    } else {
        printf("\nQueue contains \n");
        for(i = q->front; i < q->rear + 1; i++) {
            printf("%d ", q->items[i]);
        }
    }
}

```

# Comparison of DFS and BFS

**TABLE 3.1** Main facts about depth-first search (DFS) and breadth-first search (BFS)

	<b>DFS</b>	<b>BFS</b>
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta( V ^2)$	$\Theta( V ^2)$
Efficiency for adjacency lists	$\Theta( V  +  E )$	$\Theta( V  +  E )$