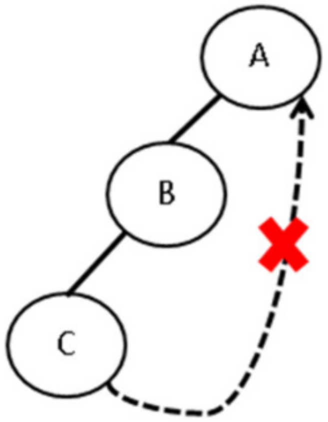
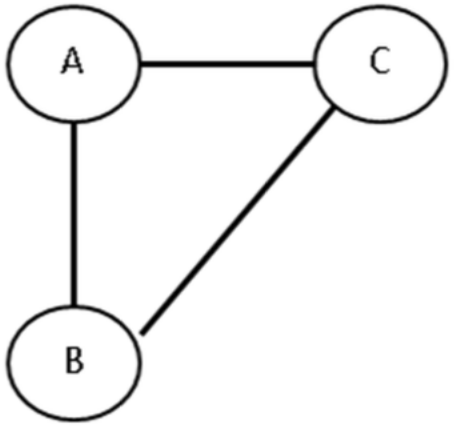
Graph I

Outline

- Definitions
- Representation of Graphs
- Operations
- Traversal

Graph vs Binary Tree





Definitions

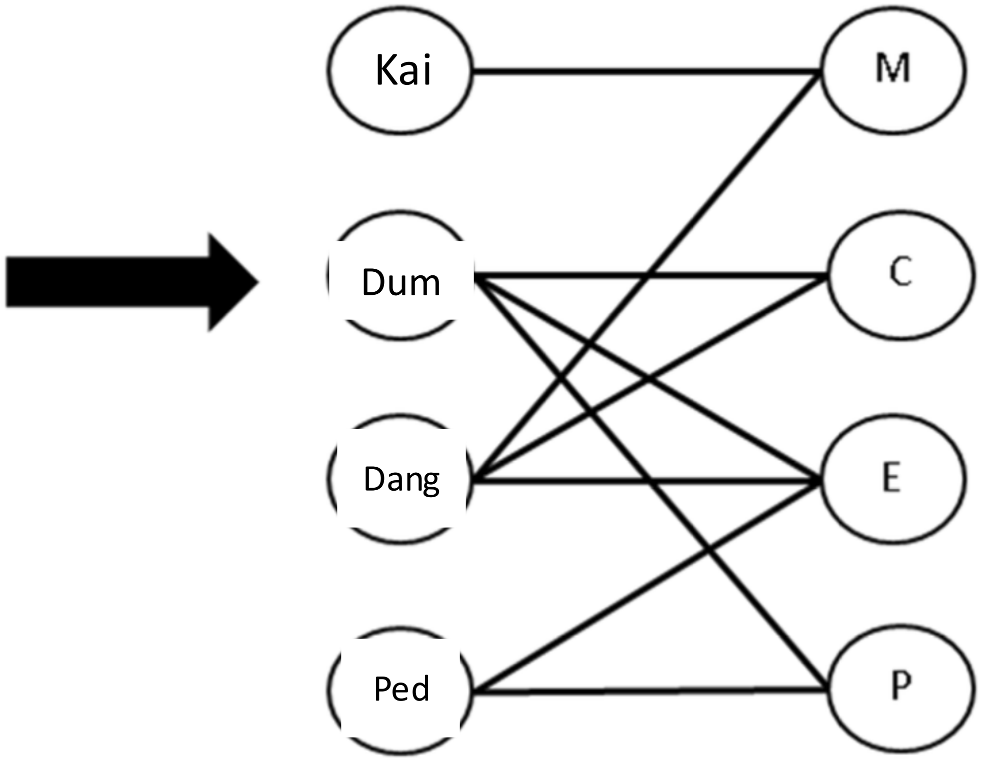
- A graph G = (V, E) consists of a set of vertices, V, and a set of edges, E. Each edge is a pair (v, w), where v, w $\in$ V.
- Edges are sometimes referred to as arcs.
- If the pair is ordered, then the graph is directed. Directed graphs are sometimes referred to as digraphs.
- Vertex w is adjacent to v if and only if (v, w) $\in$ E.
- In an undirected graph with edge (v, w), and hence (w, v), w is adjacent to v and v is adjacent to w.
- Sometimes an edge has a third component, known as either a weight or a cost.

Definitions

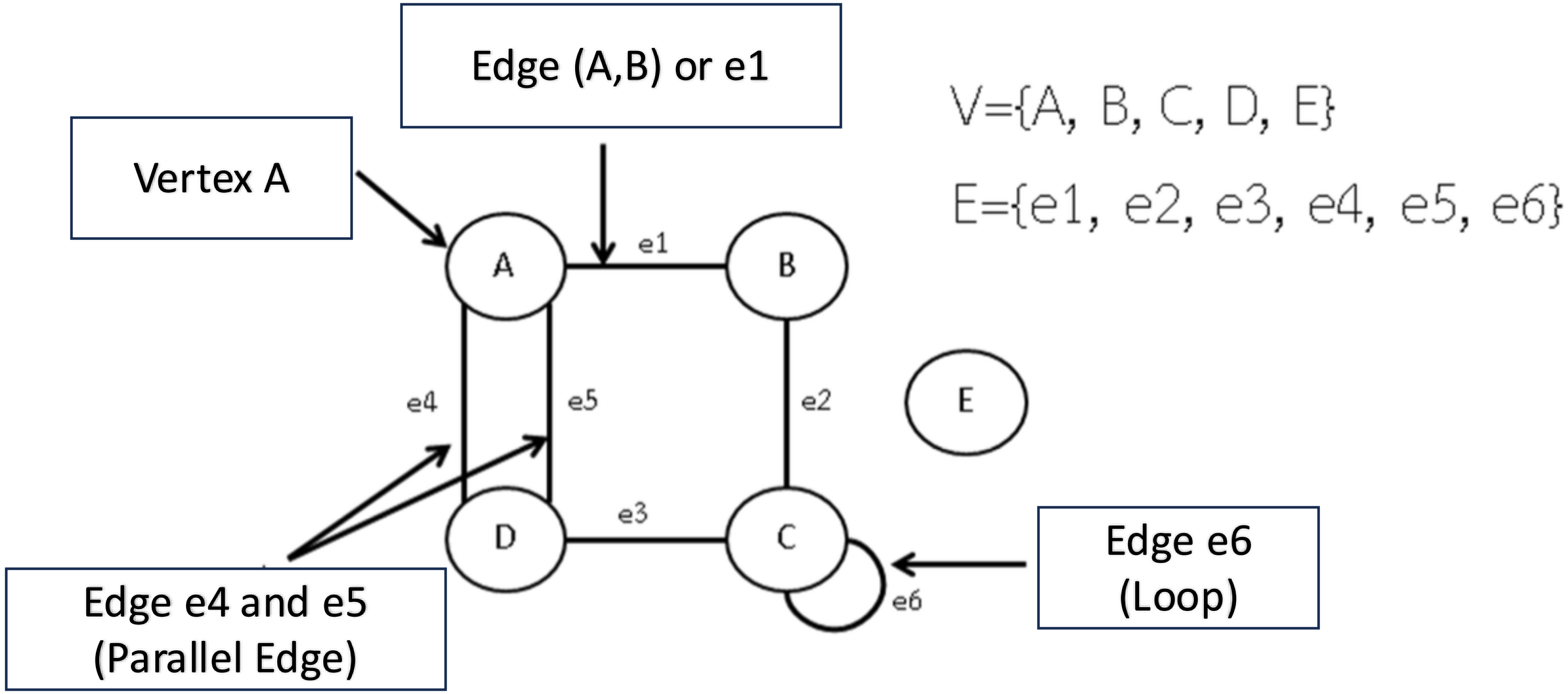
- We can apply graph to show the relationship of large data for example:
  - Transportation
  - Social network
  - Internet and network

Definitions

name	Subject
Kai	Math (M)
Dum	Computer (C)
Dang	Math (M)
Ped	Chemistry (E)
Ped	Physics (P)
Dum	Physics (P)
Dang	Chemistry (E)
Dang	Computer (C)
Dum	Chemistry (E)

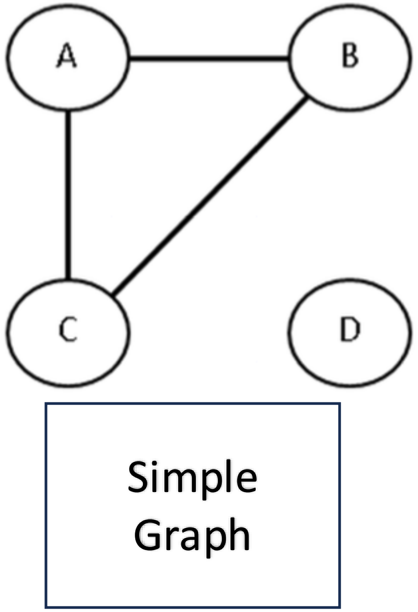


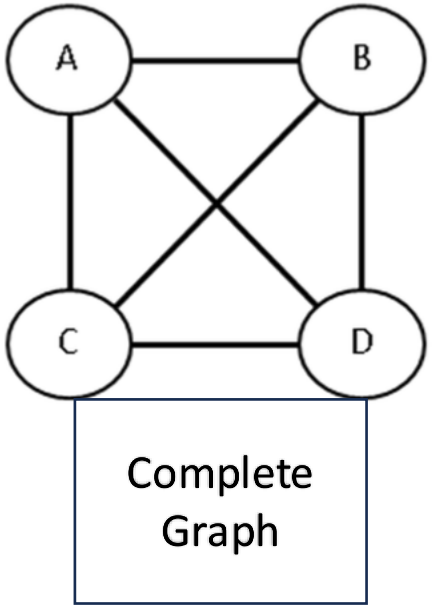
Definitions



Example

Simple Graph vs Complete Graph



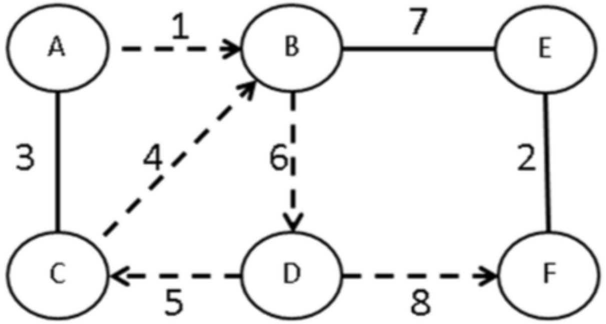


Simple Graph vs Complete Graph

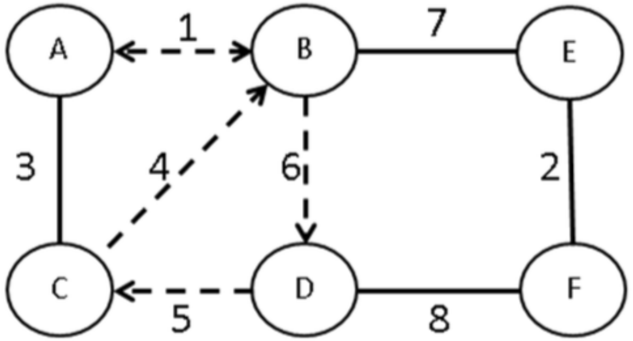
Walks, Trails, Paths, Cycles and Circuits in Graph

- Walk – A walk is a sequence of vertices and edges of a graph i.e. if we traverse a graph then we get a walk.
  - Edge and Vertices both can be repeated.
- Trail – Trail is an open walk in which no edge is repeated.
  - Vertex can be repeated.
- Circuit – Traversing a graph such that not an edge is repeated but vertex can be repeated and it is closed also i.e. it is a closed trail.
  - Vertex can be repeated.
  - Edge can not be repeated.

Example: Walks

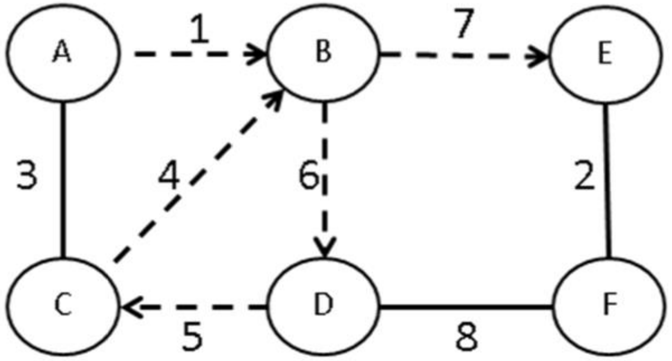


A

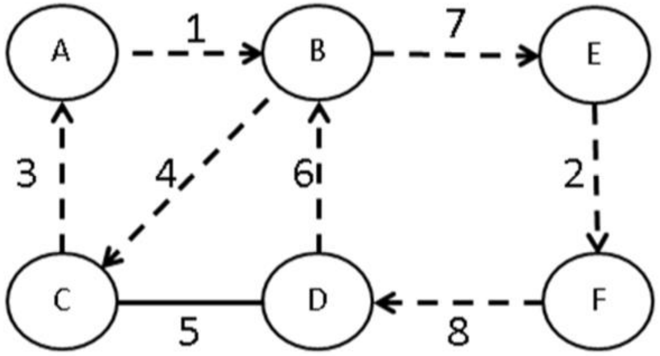


B

Example: Trails and Circuits



A



B

Trails:

A1B6D5C4B7E

Circuits:

A1B7E2F8D6B4C3A

Same

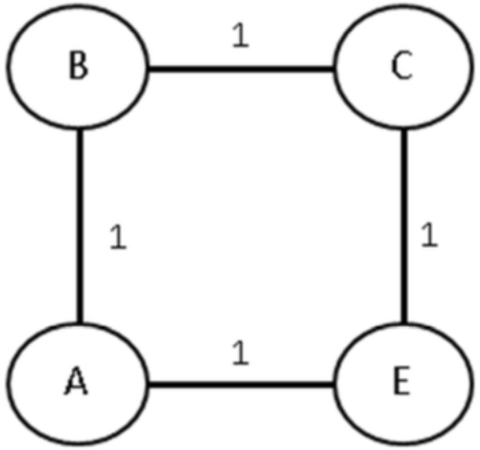
Walks, Trails, Paths, Cycles and Circuits in Graph

- Path – It is a trail in which neither vertices nor edges are repeated i.e. if we traverse a graph such that we do not repeat a vertex and nor we repeat an edge. As path is also a trail, thus it is also an open walk.
  - Another definition for path is a walk with no repeated vertex.
  - This directly implies that no edges will ever be repeated and hence is redundant to write in the definition of path.
  - Vertex not repeated
  - Edge not repeated

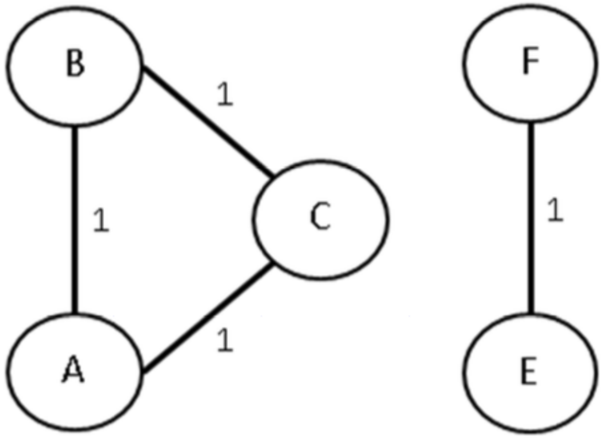
Walks, Trails, Paths, Cycles and Circuits in Graph

- Cycle – Traversing a graph such that we do not repeat a vertex nor we repeat a edge but the starting and ending vertex must be same i.e. we can repeat starting and ending vertex only then we get a cycle.
  - Vertex not repeated
  - Edge not repeated

Connected Graph vs Unconnected Graph

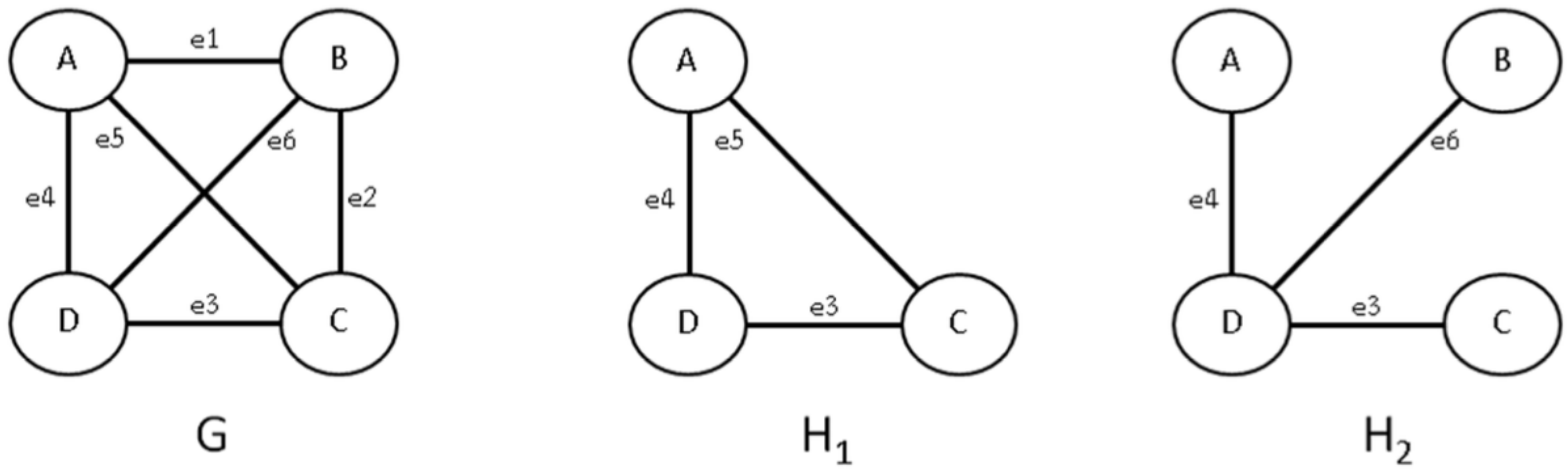


A



B

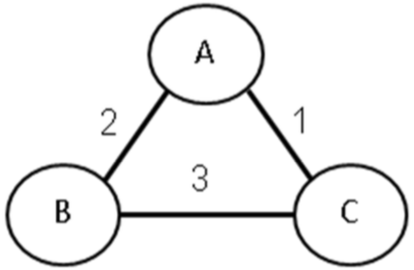
Subgraph



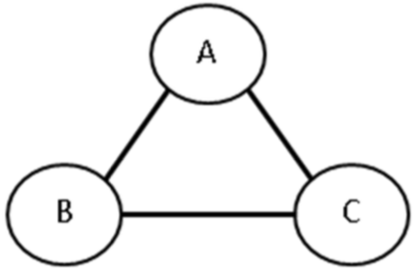
- A subgraph G of a graph is graph G' whose vertex set and edge set subsets of the graph G.
- In simple words a graph is said to be a subgraph if it is a part of another graph.

Types of Graph

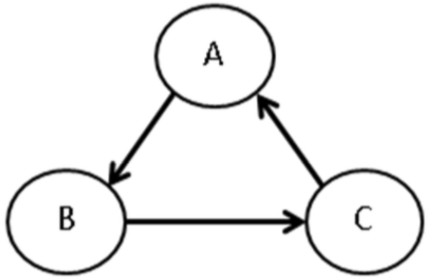
- There are 4 types of graph
  - Weighted Graph
  - Unweighted Graph
  - Direct Graph
  - Undirected Graph



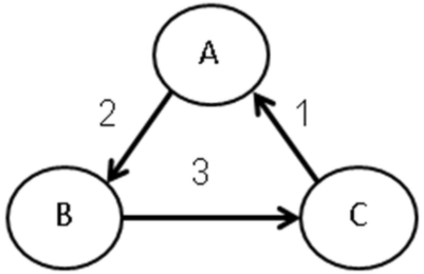
A



B



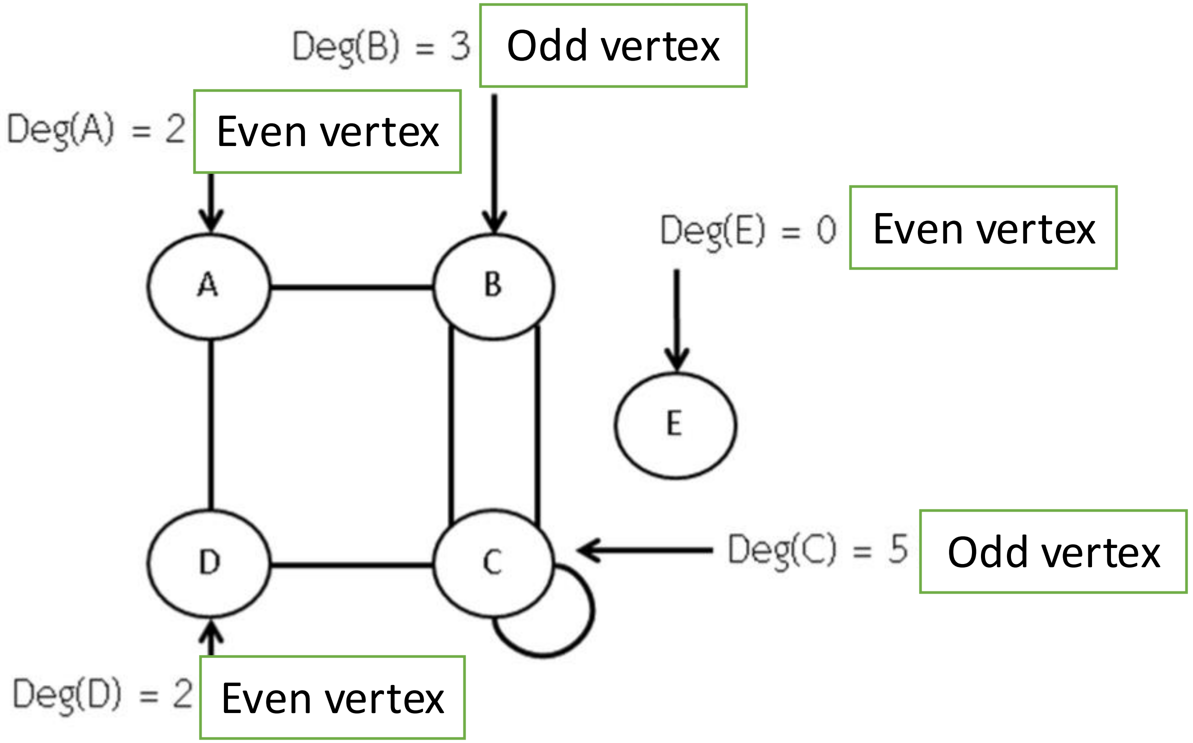
C



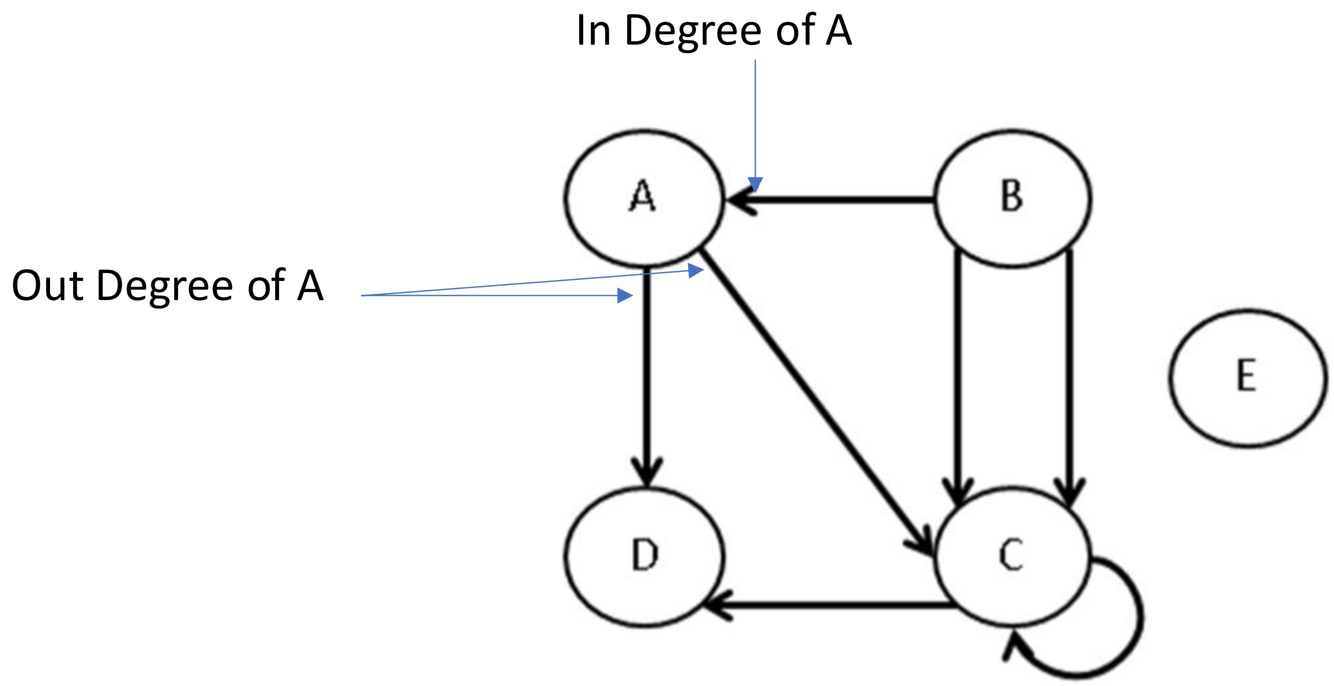
D

Degree

- Degree of Vertex is the number of edges which connects to that vertex.



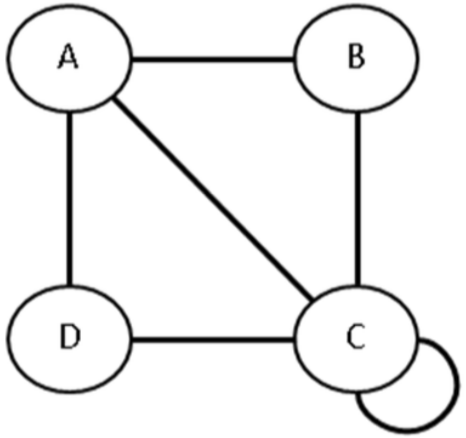
In and Out Degree of vertex



Representation of Graphs

- One simple way to represent a graph is to use a two-dimensional array.
- This is known as an adjacency matrix representation.
- For each edge (u, v), we set A[u][v] to true; otherwise, the entry in the array is false.
- If the edge has a weight associated with it, then we can set A[u][v] equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.
- For instance, if we were looking for the cheapest airplane route, we could represent nonexistent flights with a cost of ∞.
- If we were looking, for some strange reason, for the most expensive airplane route, we could use $-\infty$ (or perhaps 0) to represent nonexistent edges.

Adjacency Matrix



A

Destination

Source

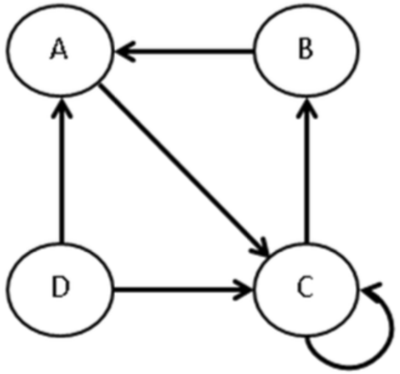
	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	1	1	1
D	1	0	1	0

A to B

B to A

Vertices are not connected.

B



C

Destination

Source

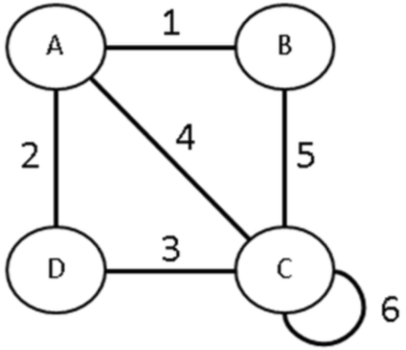
	A	B	C	D
A	0	0	1	0
B	1	0	0	0
C	0	1	1	0
D	1	0	1	0

A to C

Vertices are not connected.

D

Adjacency Matrix



E

Destination

Source

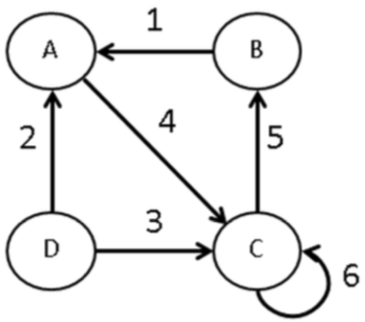
	A	B	C	D
A	0	1	4	2
B	1	0	5	0
C	4	5	6	3
D	2	0	3	0

A to C, weight = 4

C to A, weight = 4

Vertices are not connected.

F



G

Destination

Source

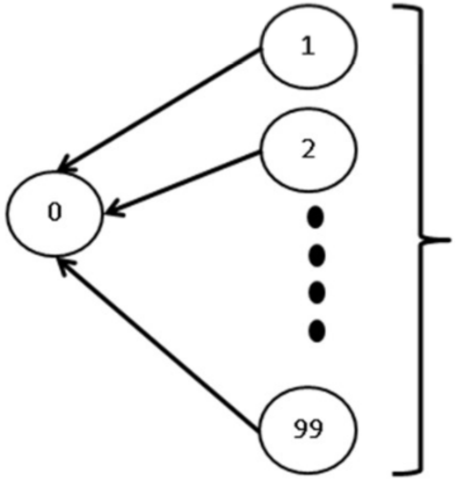
	A	B	C	D
A	0	0	4	0
B	1	0	0	0
C	0	5	6	0
D	2	0	3	0

A to C, weight = 4

Vertices are not connected.

H

Adjacency Matrix



99 vertices are connected to 0

100

100

	0	1	2	...	99
0	0	0	0	...	0
1	1	0	0	...	0
2	1	0			
...	...	...			
99	1	0			

Not used

Adjacency List

- We can use array and Linked List to store the graph.

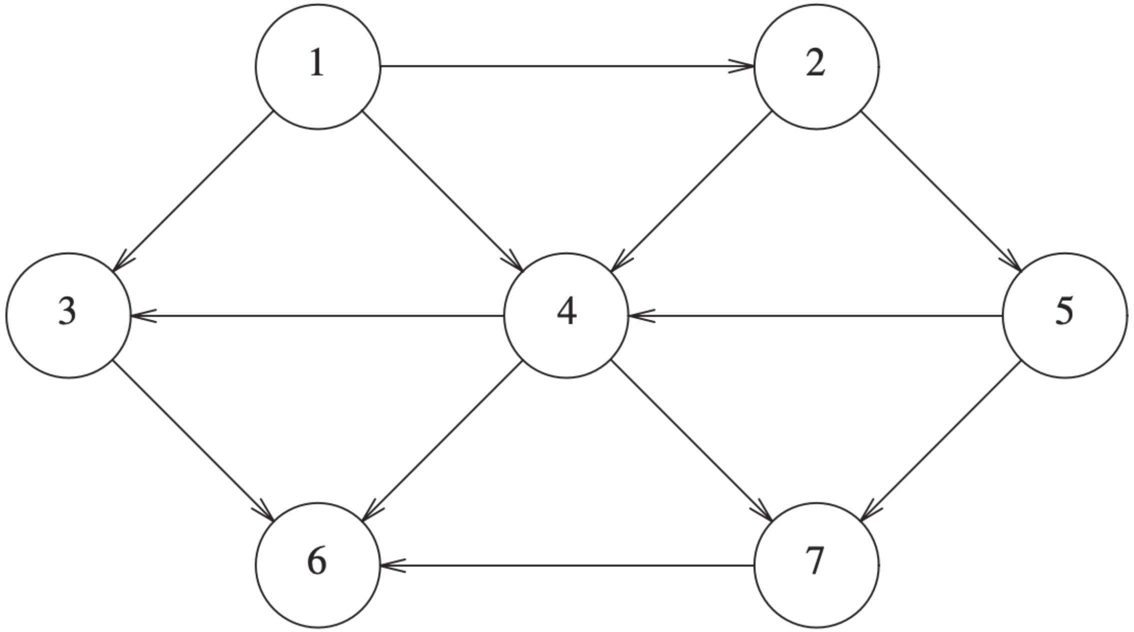


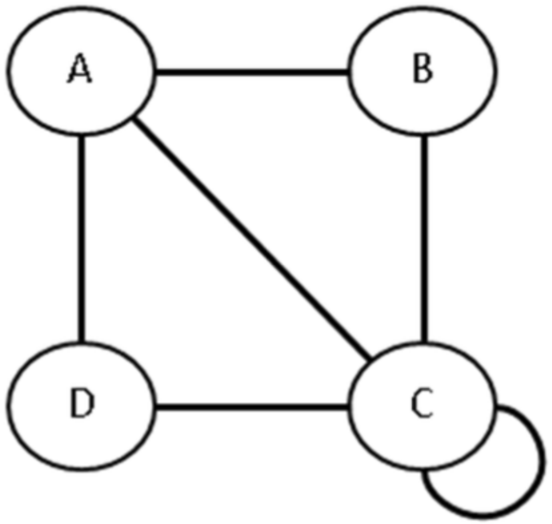
Figure 9.1 A directed graph

Adjacency List

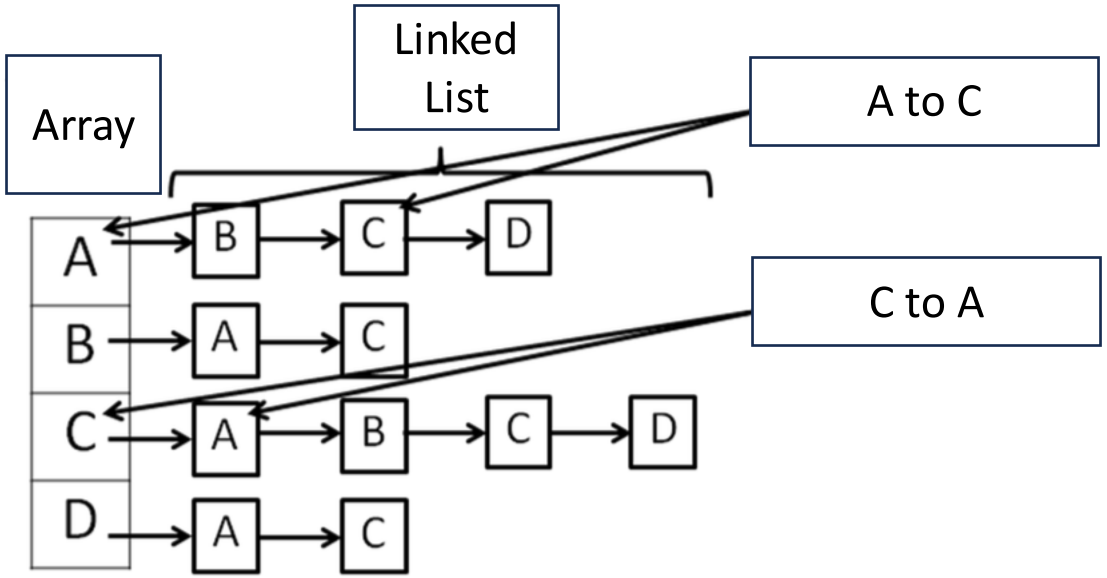
1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

Figure 9.2 An adjacency list representation of a graph

Example: Adjacency List

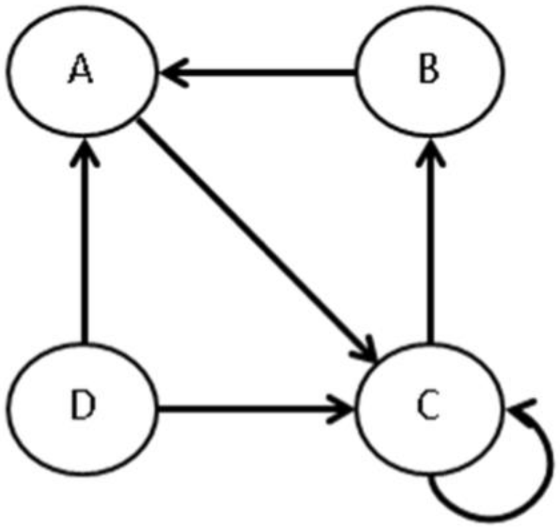


(ก)

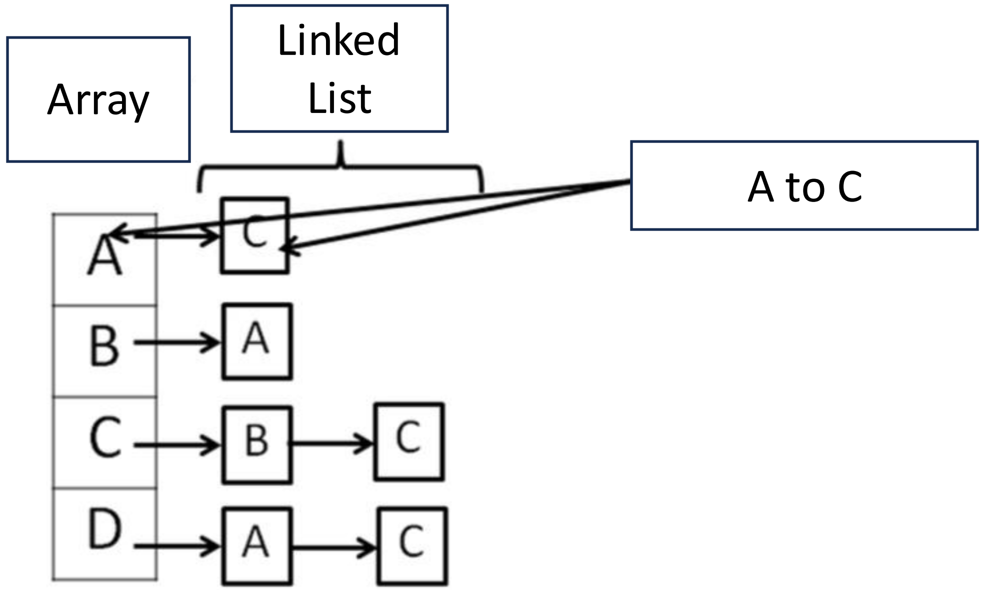


(ข)

Example: Adjacency List

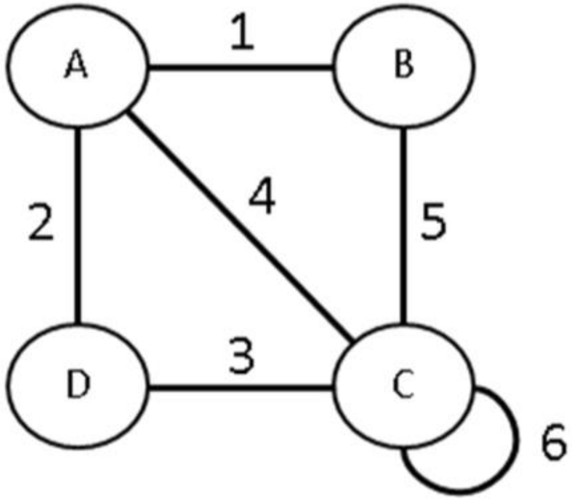


(ค)

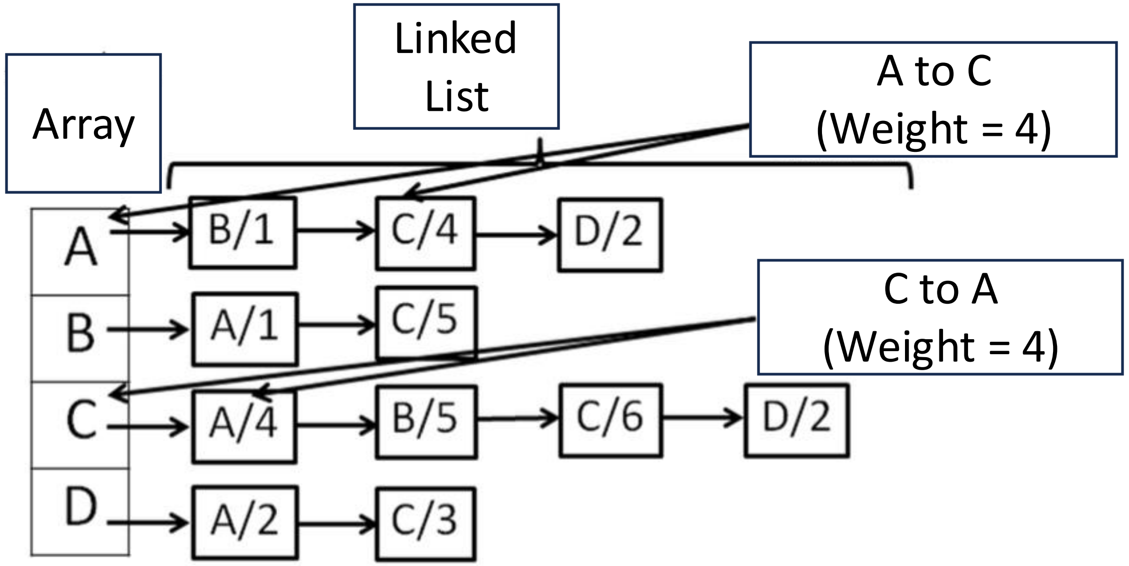


(ง)

Example: Adjacency List

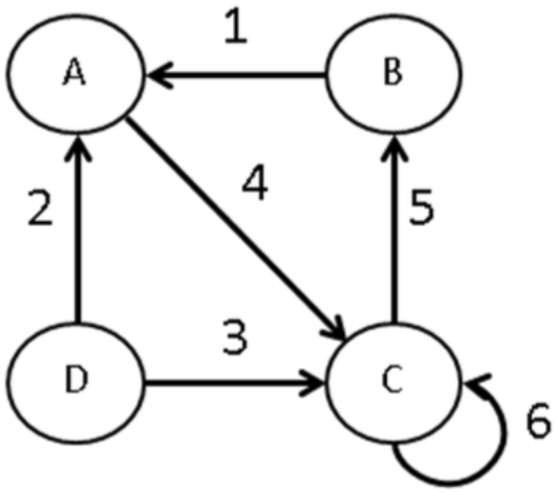


(จ)

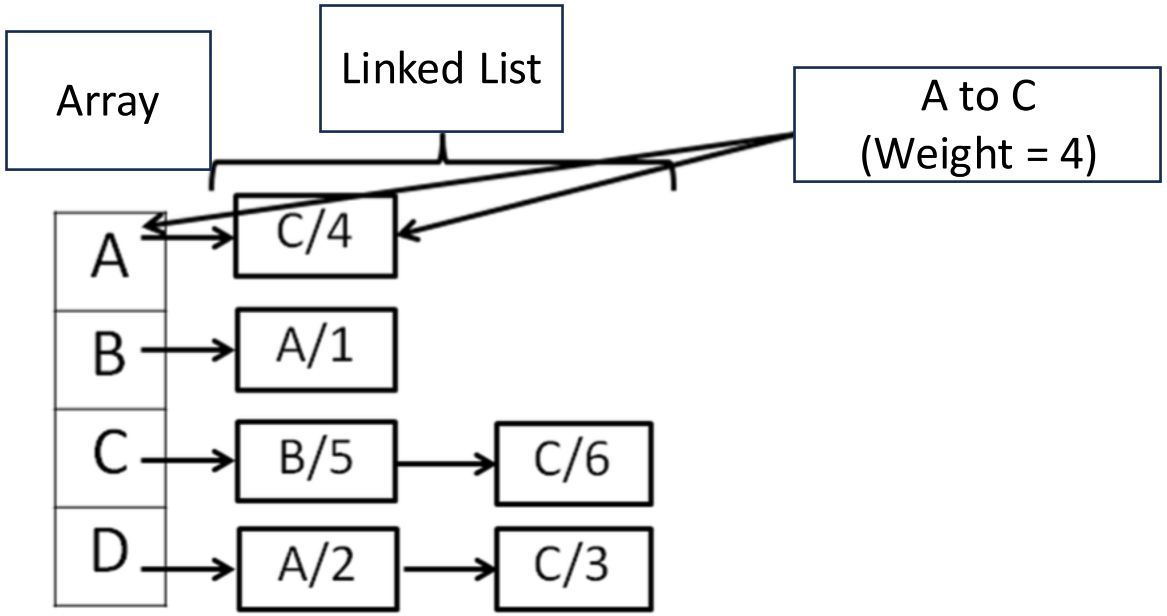


(ฉ)

Example: Adjacency List



(ช)



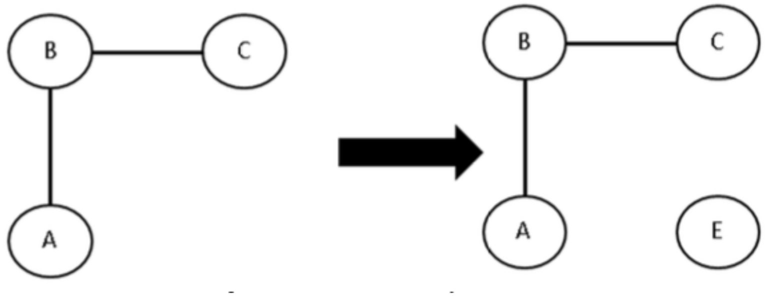
(ซ)

Operations

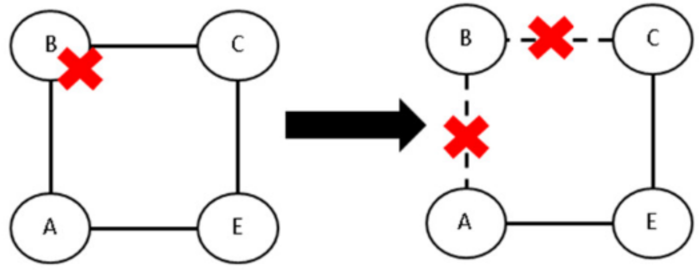
- Add vertex
- Delete vertex
- Add Edge
- Delete Edge
- Graph Traversal
- Find vertex

Add and Delete vertex

- Add vertex: add vertex E

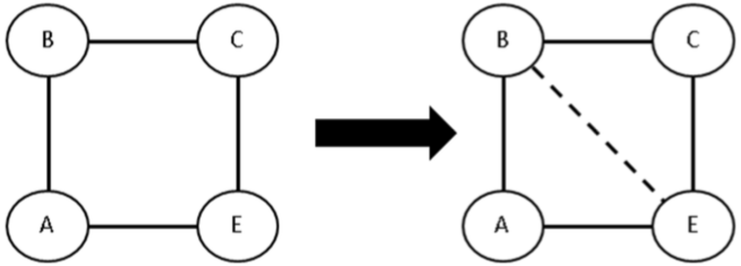


- Delete vertex: delete vertex B (edge A-B and B-C will be deleted)

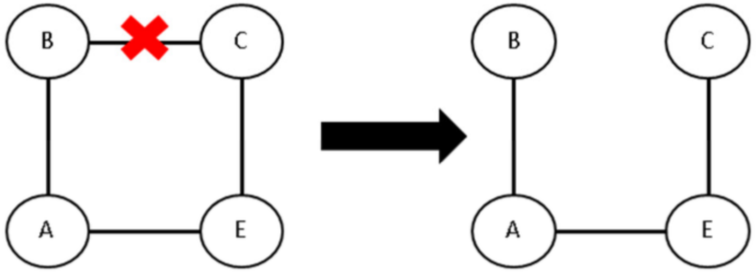


Add and Delete Edge

- Add vertex add edge between B and E



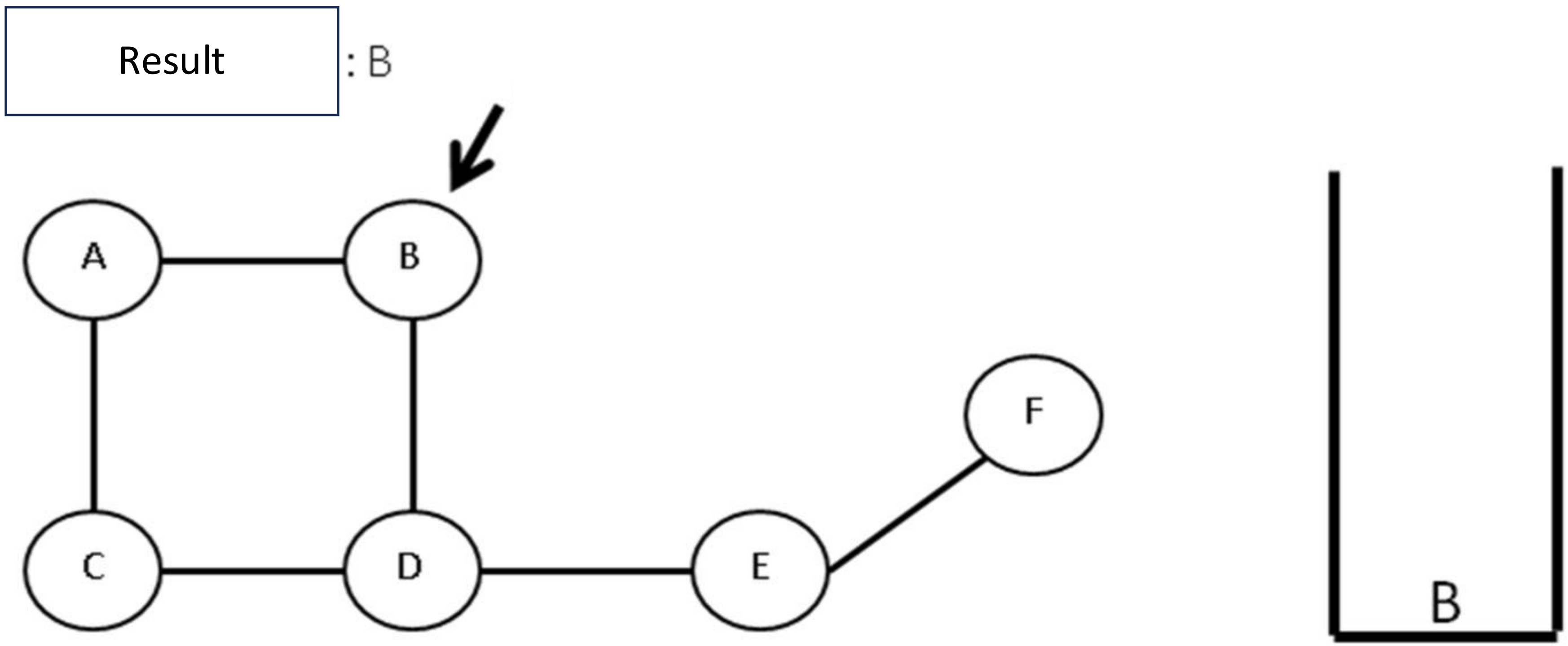
- Delete vertex: Delete edge between B and C



Graph Traversal

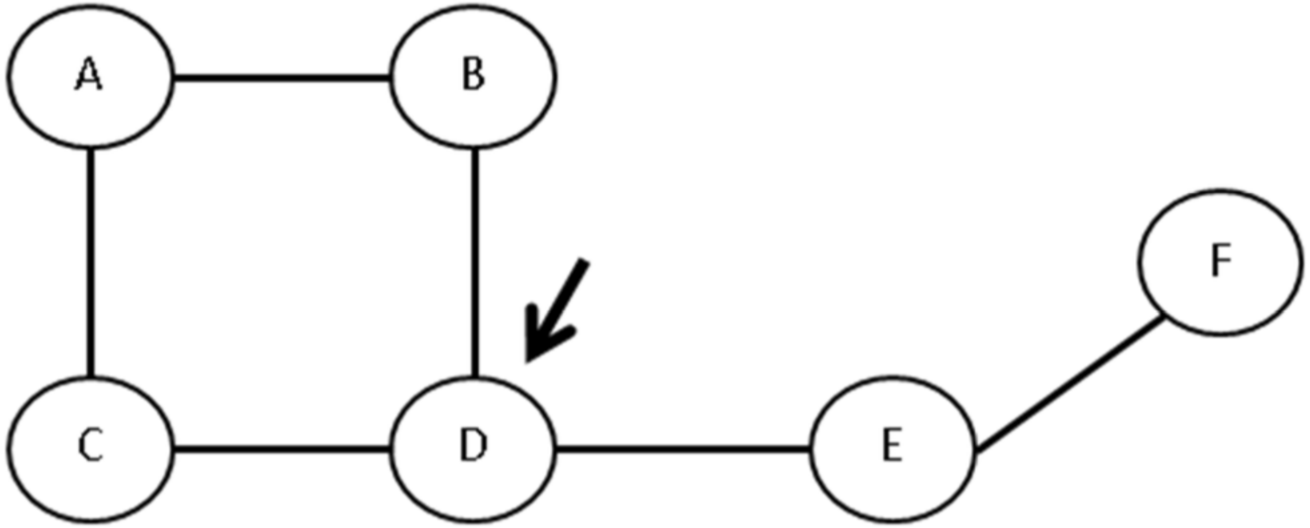
- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a **stack** to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a **queue** to remember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

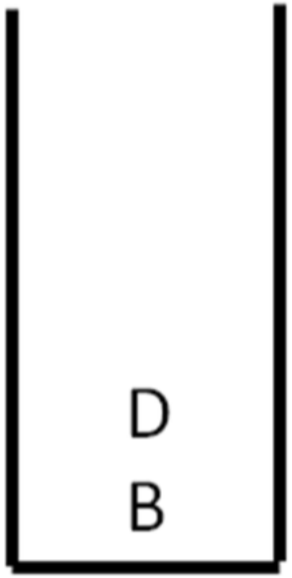
DFS



DFS

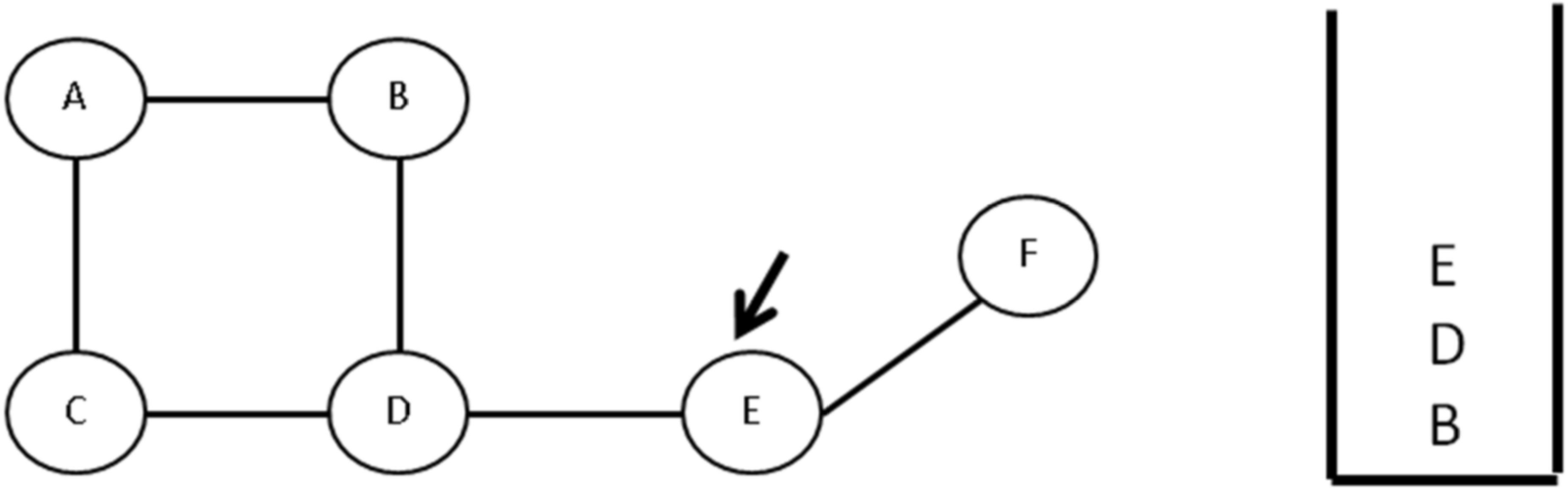
Result : B D





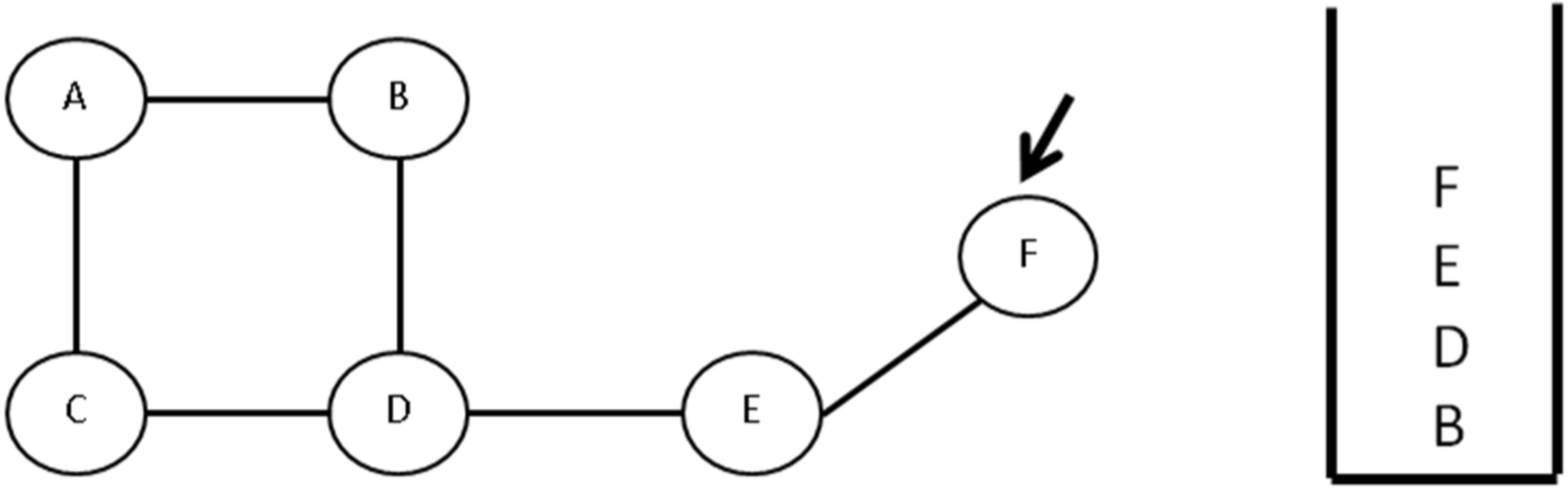
DFS

Result : B D E



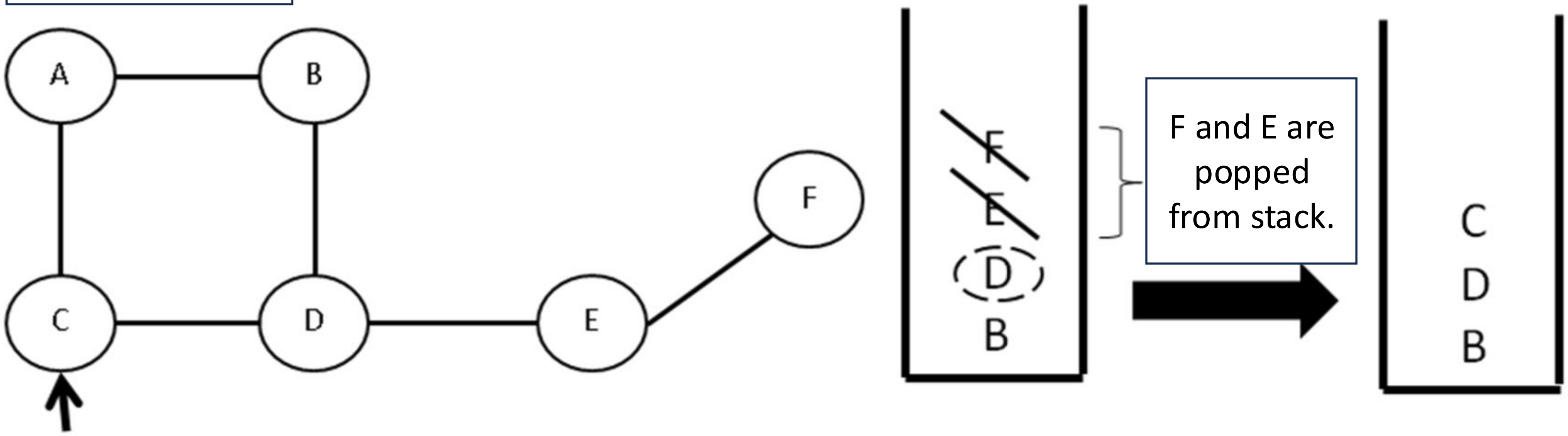
DFS

Result : B D E F



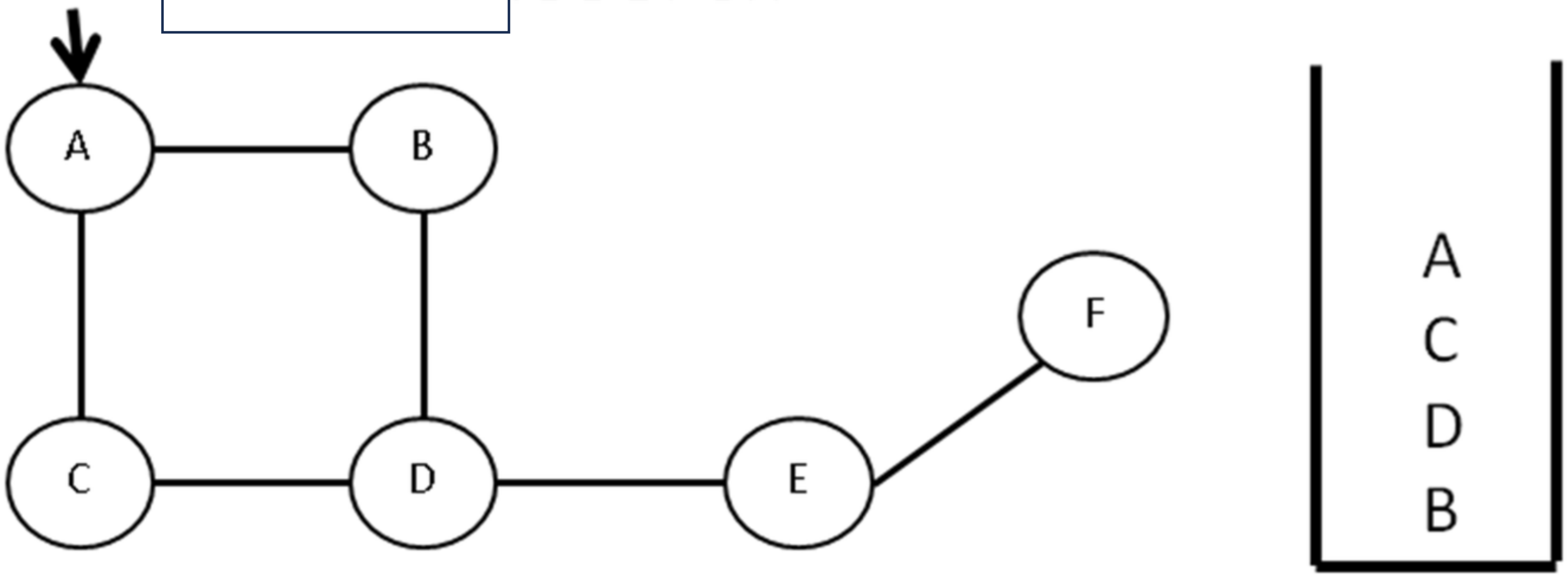
DFS

Result : B D E F C



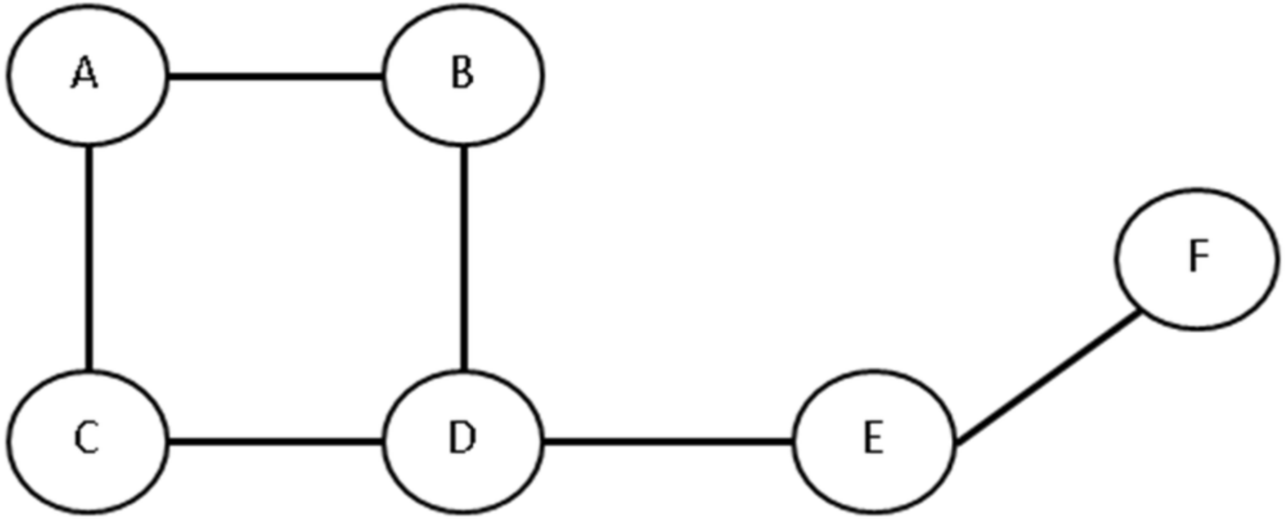
DFS

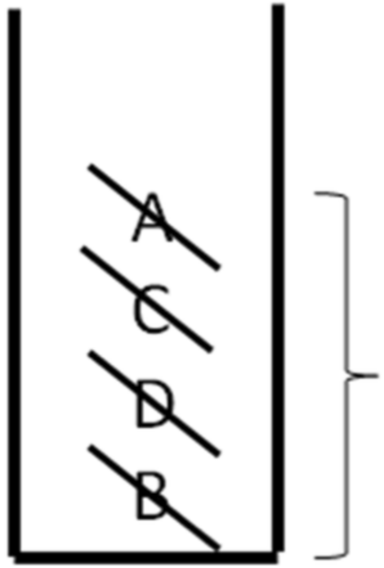
Result : B D E F C A



DFS

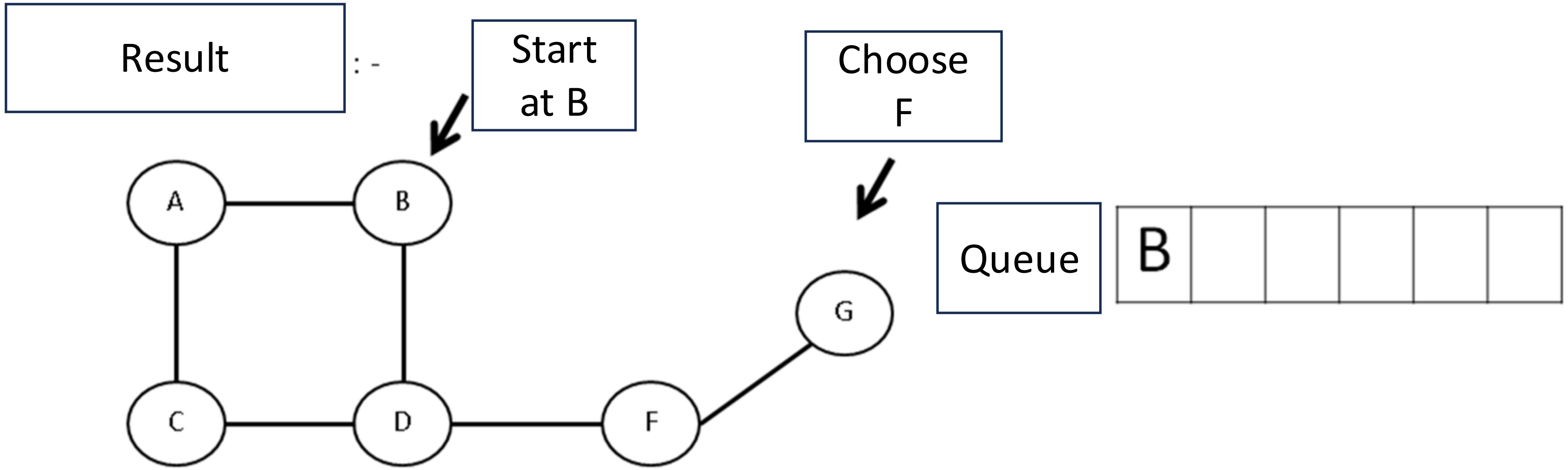
Result : B D E F C A



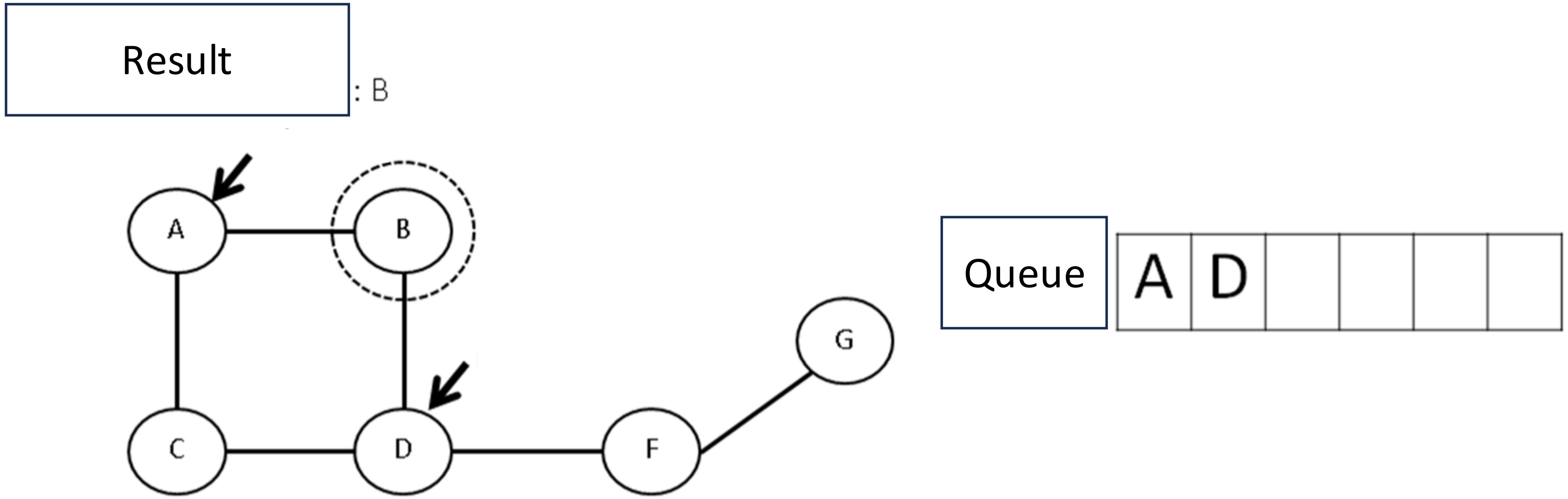


A,C,D and B are popped from stack.

BFS

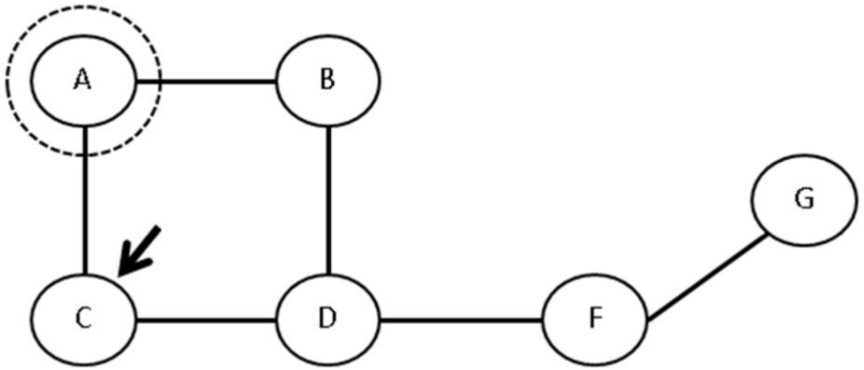


BFS



BFS

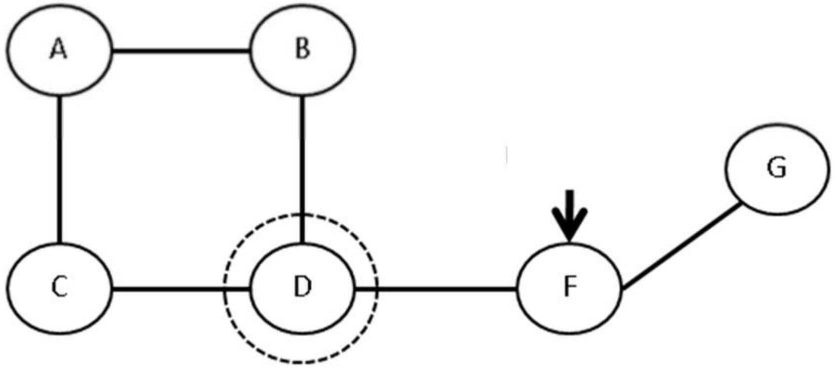
Result : B A



Queue	D	C				

BFS

Result : B A D

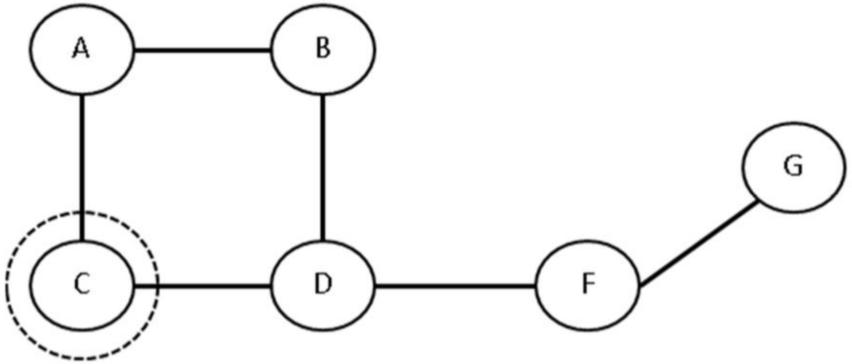


Queue

C	F				

BFS

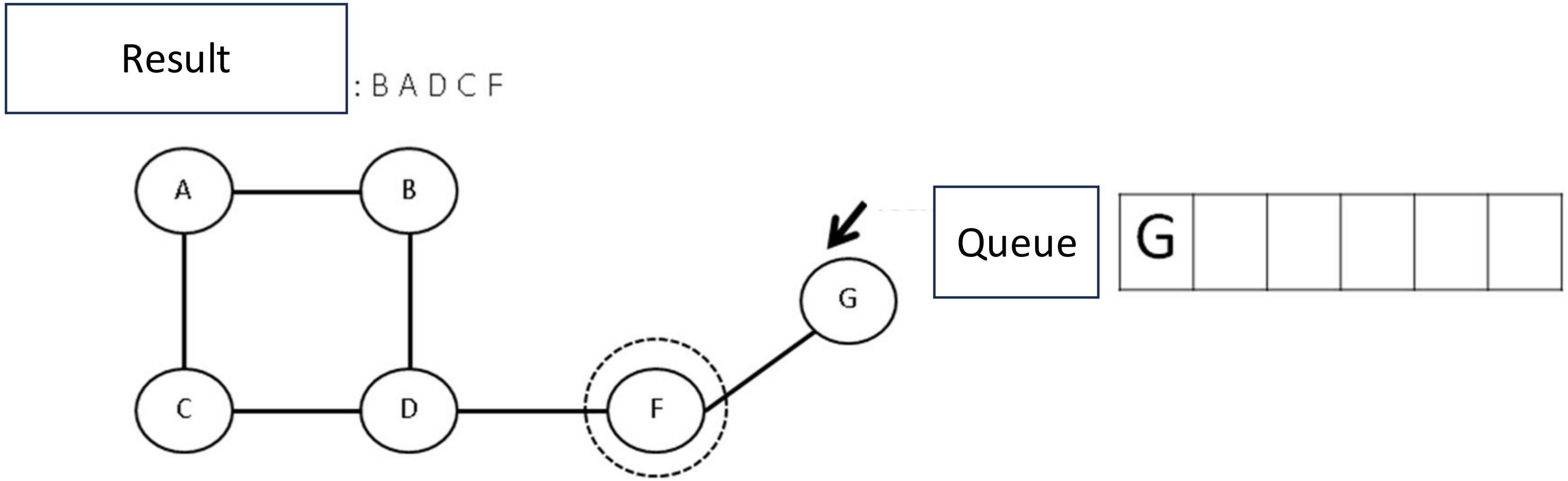
Result : B A D C



Queue

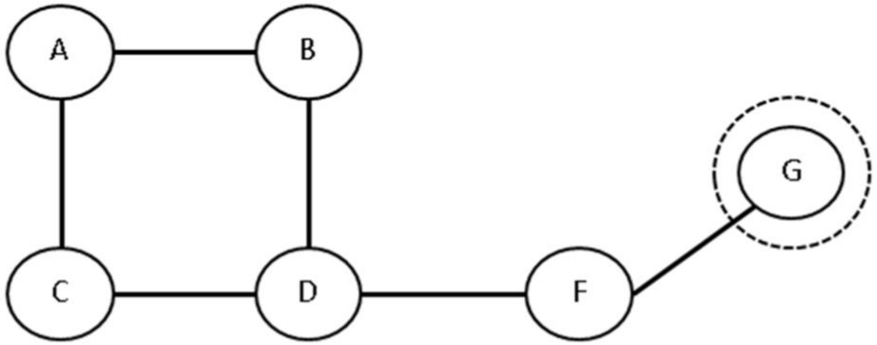
F					

BFS



BFS

Result : B A D C F G



Queue

Example: Adjacency Matrix

```
#include<bits/stdc++.h>
using namespace std;
class graph
{
    public:
    int edges[100][100];
    int s_v;
    graph(int n)
    {
        s_v = n;
        for(int i=0;i<s_v;i++)
        {
            for(int j=0;j<s_v;j++)
            {
                edges[i][j] = 0;
            }
        }
    }
    void add_edge(int x,int y,int w)
    {
        edges[x][y] = w;
    }
```

Example: Adjacency Matrix

```
    void print()
    {
        for(int i=0;i<s_v;i++)
        {
            cout<<i<<" : ";
            for(int j=0 ; j <= s_v ; j++ )
            {
                if(edges[i][j] > 0 )
                {
                    cout<<j<<","<<edges[i][j]<<"  ";
                }
            }
            cout<<endl;
        }
    }
};
```

Example: adjacency list

```
#include<bits/stdc++.h>
using namespace std;
class node
{
    public:
    int     vertex;
    int     weight;
    node    *next;
    node(int v, int w)
    {
        vertex = v;
        weight = w;
        next = NULL;
    }
};
class graph
{
    public:
    int   s_v;
    node*   vertex[1000];
    graph(int n)
    {
        s_v = n;
        for(int i=0;i<s_v;i++)
        {
            vertex[i] = new node(i,0);
        }
    }
```

Example: adjacency list

```
    void add_edge(int x, int y, int w)
    {
        node *n = new node(y,w);
        node *v = vertex[x];
        while(v->next != NULL)
        {
            v = v->next;
        }
        v->next = n;
    }
    void print()
    {
        node *r;
        for( int i=0;i<s_v;i++)
        {
            r = vertex[i];
            cout<<i<<" : ";
            while(r->next != NULL)
            {
                cout<<r->next->vertex<<","<<r->next->weight<<"  ";
                r = r->next;
            }
            cout<<endl;
        }
    }
};
```

Example:

Operations

```
#include<bits/stdc++.h>
using namespace std;
class graph
{
    public:
    int edges[100][100];
    int s_v;
    graph(int n)
    {
        s_v = n;
        for(int i=0;i<s_v;i++)
        {
            for(int j=0;j<s_v;j++)
            {
                edges[i][j] = 0;
            }
        }
    }
    void add_edge(int x,int y,int w)
    {
        edges[x][y] = w;
    }
    void print()
    {
        for(int i=0;i<s_v;i++)
        {
            cout<<i<<" : ";
```

Example: Operations

```
        for(int j=0 ; j <= s_v ; j++ )
        {
            if(edges[i][j] > 0 )
            {
                cout<<j<<","<<edges[i][j]<<"  ";
            }
        }
        cout<<endl;
    }
}
void bft(int start)
{
    bool visited_bft[100];
    for(int i=0;i<100;i++)
    {
        visited_bft[i] = 0;
    }
    visited_bft[start] = 1;
    vector<int> q;
    q.push_back(start);
    while(q.empty() == 0)
    {
        start = q.front();
        cout << start << " ";
        q.erase(q.begin());
        for(int y=0 ; y<s_v ; y++)
        {
```

Example: Operations

```
            if( visited_bft[y] == 0 && edges[start][y] > 0 )
            {
                visited_bft[y] = 1;
                q.push_back(y);
            }
        }
    }
}
bool visited_dft[100];
void sub_dft(int start)
{
    cout<<start<<" ";
    visited_dft[start] = 1;
    for(int y=0;y<s_v;y++)
    {
        if( visited_dft[y] == 0 && edges[start][y] > 0 )
        {
            sub_dft(y);
        }
    }
}
void dft(int start)
{
    for(int i=0;i<100;i++)
    {
        visited_dft[i] = 0;
    }
    sub_dft(start);
}
```

Example: Operations

```
void sub_graph()
{
    int num_subgraph  = 1;
    for(int i=0;i<100;i++)
    {
        visited_dft[i] = 0;
    }
    for(int y=0;y<s_v;y++)
    {
        if( visited_dft[y]==0 )
        {
            cout<<"\nsub graph = "<<num_subgraph<<" : ";
            sub_dft(y);
            num_subgraph = num_subgraph + 1;
        }
    }
}
int n_in_degree[100];
int t_edges[100][100];
void in_degree()
{
    for(int i=0;i<s_v;i++)
    {
        n_in_degree[i] = 0;
        for(int j=0;j<s_v;j++)
        {
```

Example:

Operations

```
                for(int k=0;k<s_v;k++)
                {
                    if( t_edges[j][k] == 1 )
                    {
                        n_in_degree[k]++;
                    }
                }
            }
        }
    }
```

Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ