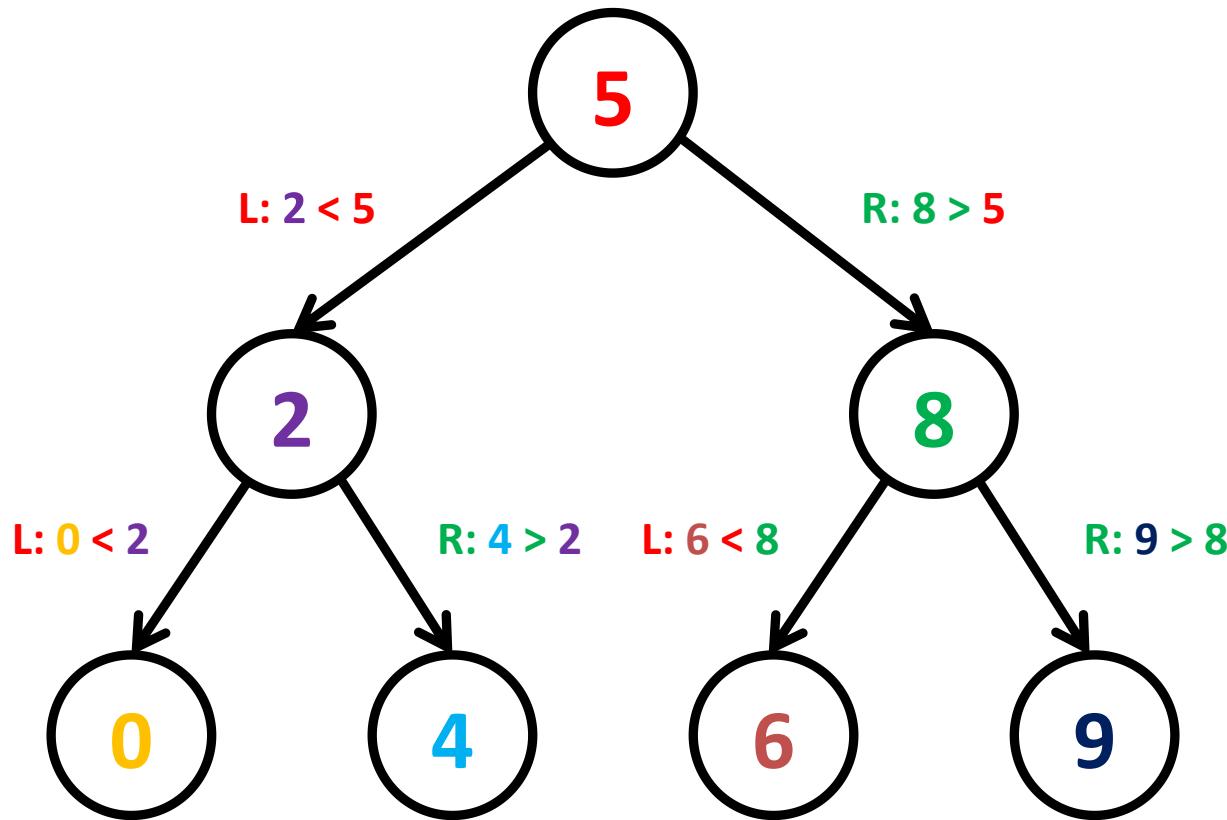


Binary Search Tree



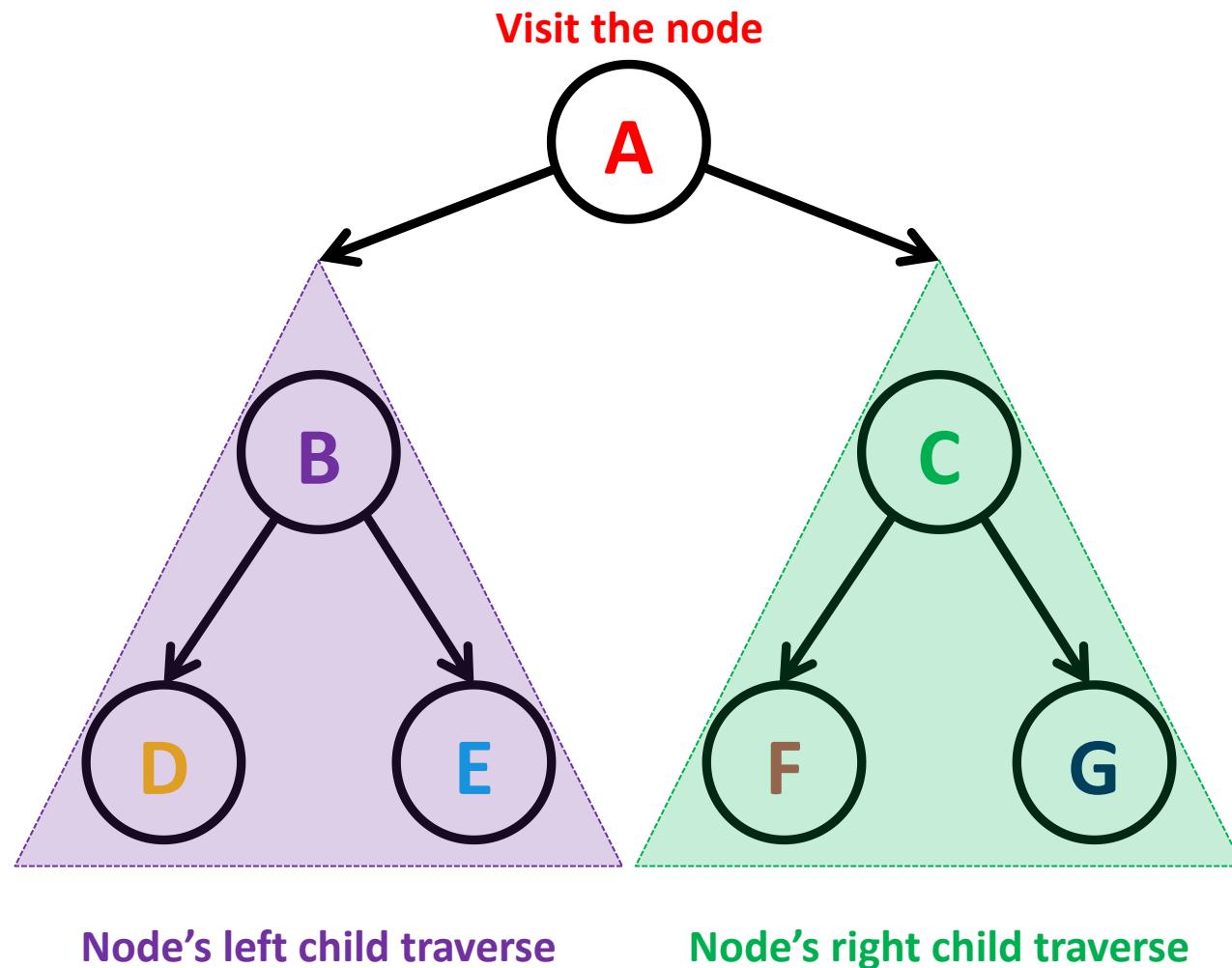
(Binary) Tree Traversals

- Tree Traversal: a process to visit **all nodes** in a tree is called a tree traversal. Typically used to **find a key or process all or some keys.**
- Traversal always **starts at the root of the tree**

Binary Tree Traversal involves three “visits”:

- The **node** is visited (the root)
- The **node's left child** is traversed
- The **node's right child** is traversed

Binary Tree Traversal



Binary Tree Traversals

The sequence in which those three “visits” are executed determines what kind of traversal it is.

Typical traversals:

- Pre-order Traversal (**Node**, **Left Child**, **Right Child**)
- In-order Traversal (**Left Child**, **Node**, **Right Child**)
- Post-order Traversal (**Left Child**, **Right Child**, **Node**)
- Level-order (Level-by-level | left-to-right)

Binary Tree Traversals

- Pre-order Traversal (**Node**, **Left Child**, **Right Child**)
 - See **parent first**, then look at **left** and **right** branches
- In-order Traversal (**Left Child**, **Node**, **Right Child**)
 - See **left branch first**, then look at the **parent**, and finally look at the **right** branch
- Post-order Traversal (**Left Child**, **Right Child**, **Node**)
 - Look at **left** and **right** branches first, and then see **parent**

Pre-order (Node, Left, Right)

Step 1:

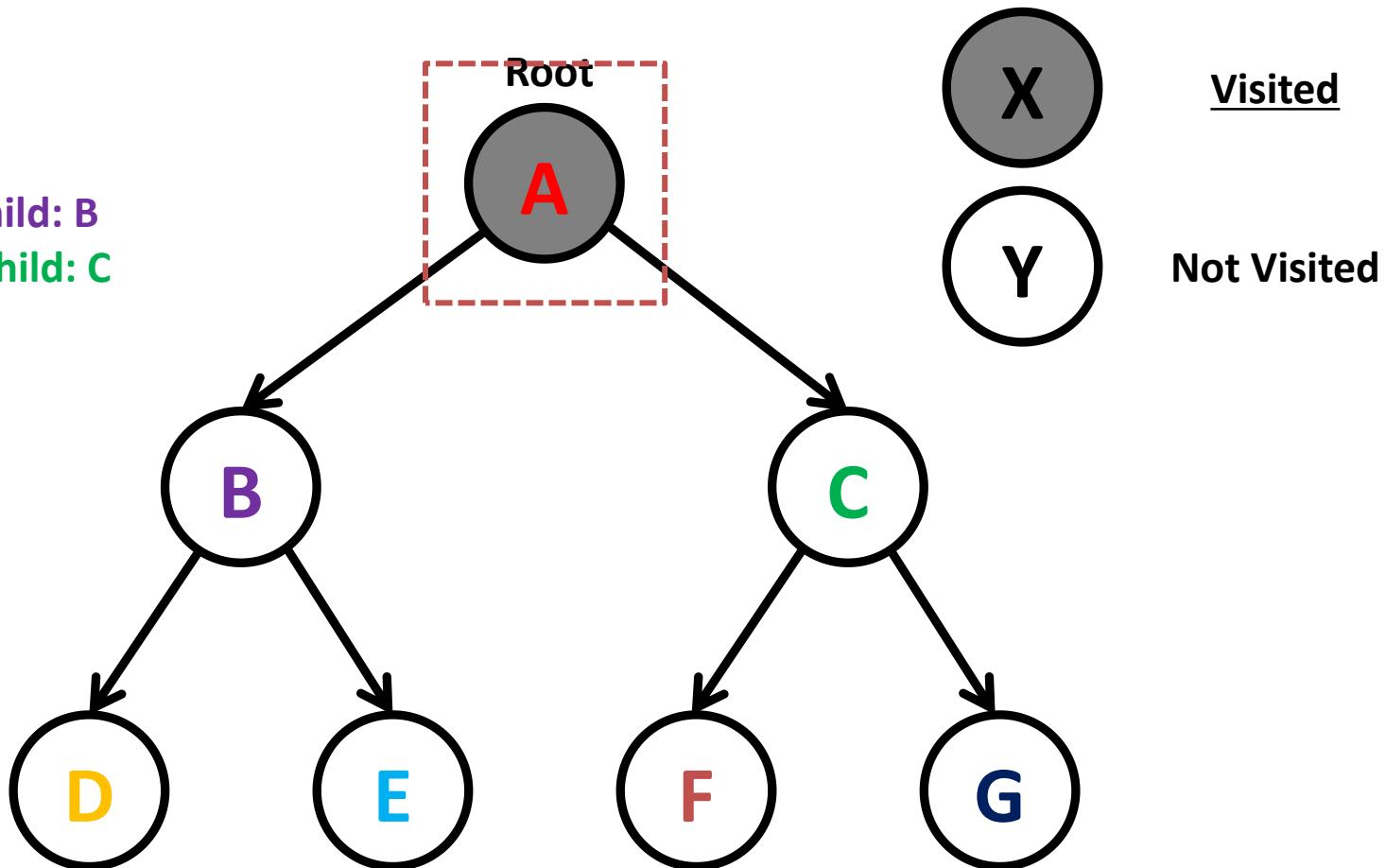
Node: A

Node's Left Child: B

Node's Right Child: C

Visited

Not Visited



Nodes traversed so far: A

Pre-order (Node, Left, Right)

Step 2:

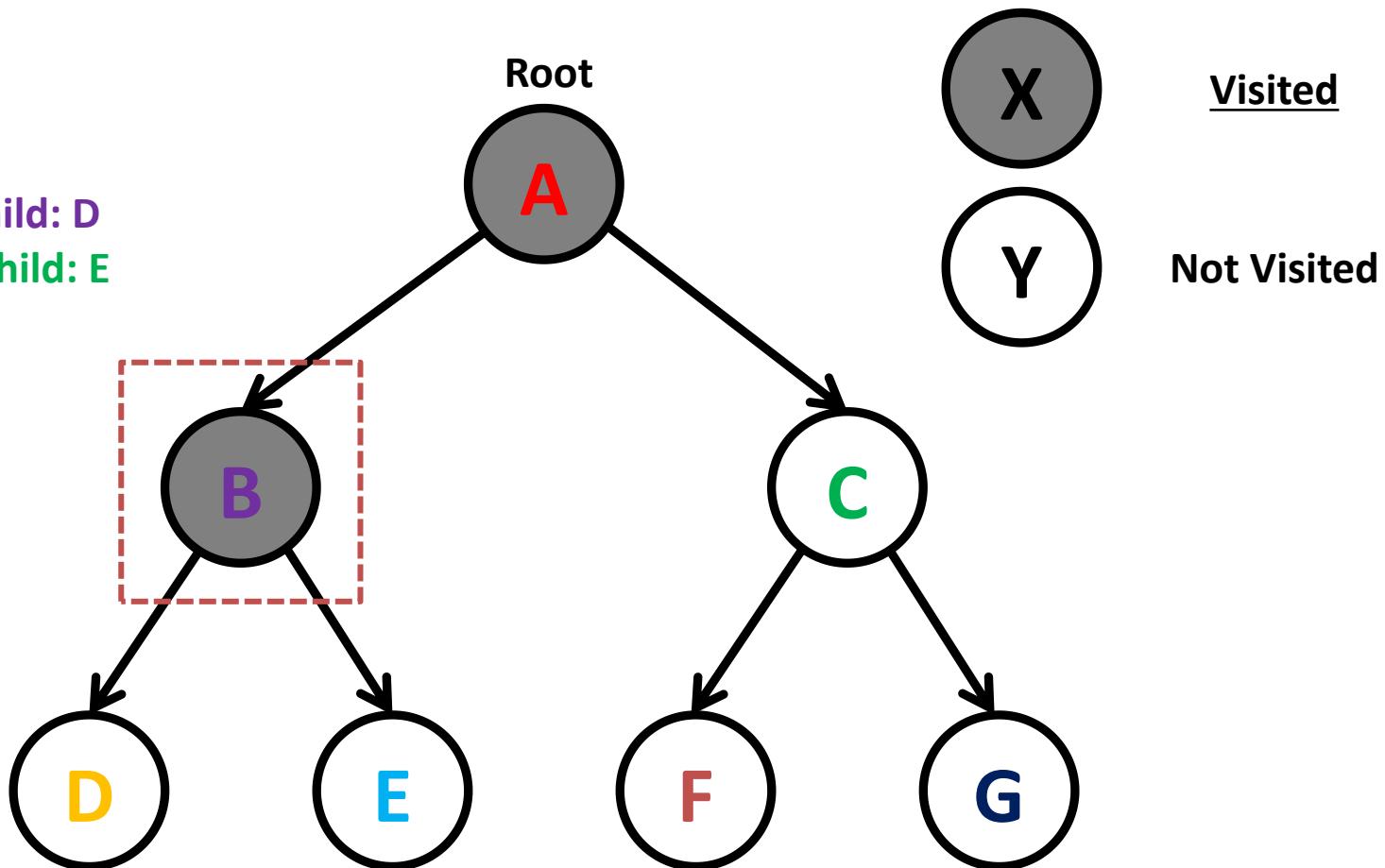
Node: B

Node's Left Child: D

Node's Right Child: E

Visited

Not Visited



Nodes traversed so far: A B

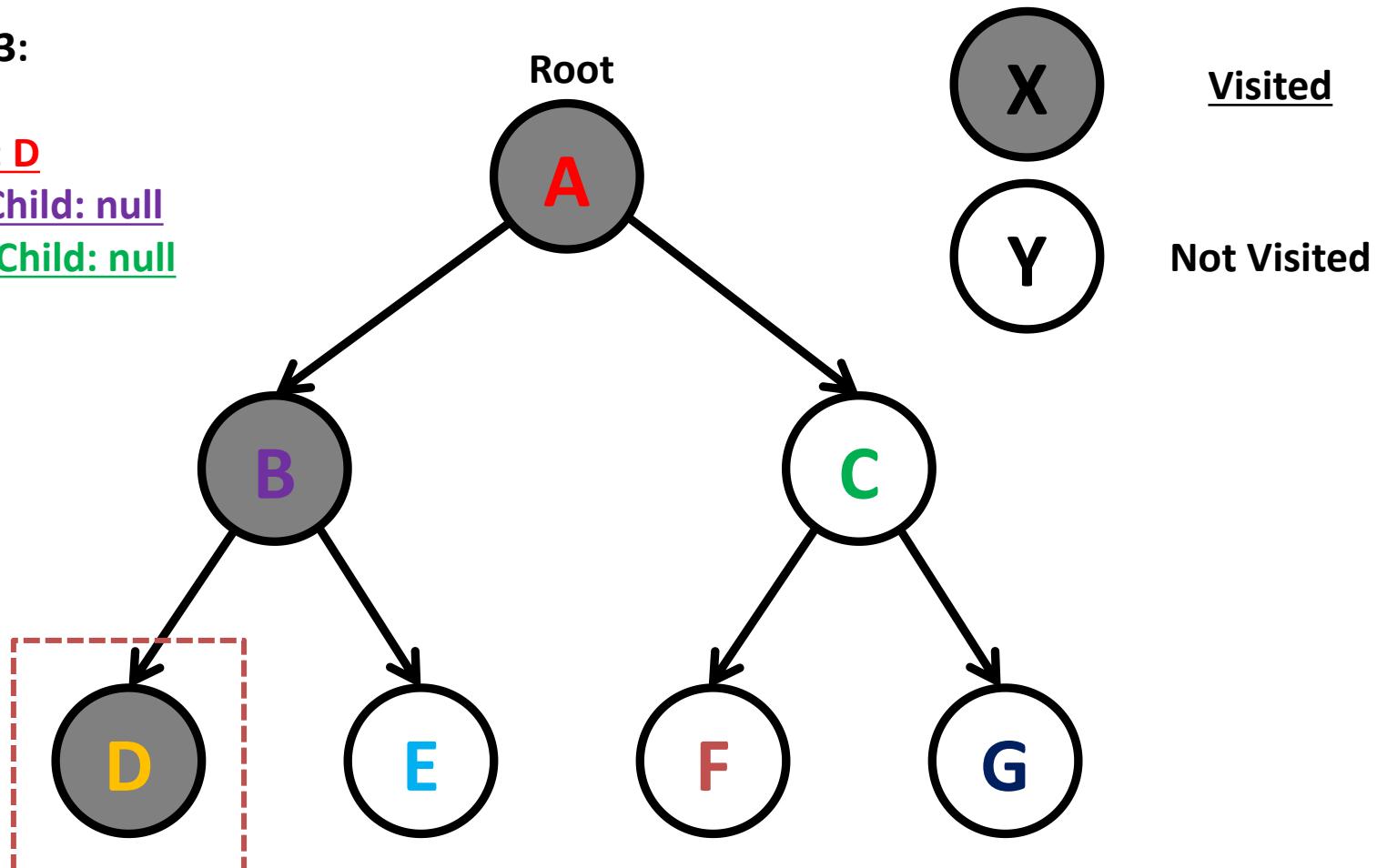
Pre-order (Node, Left, Right)

Step 3:

Node: D

Node's Left Child: null

Node's Right Child: null



Nodes traversed so far: A B D

Pre-order (Node, Left, Right)

Step 4:

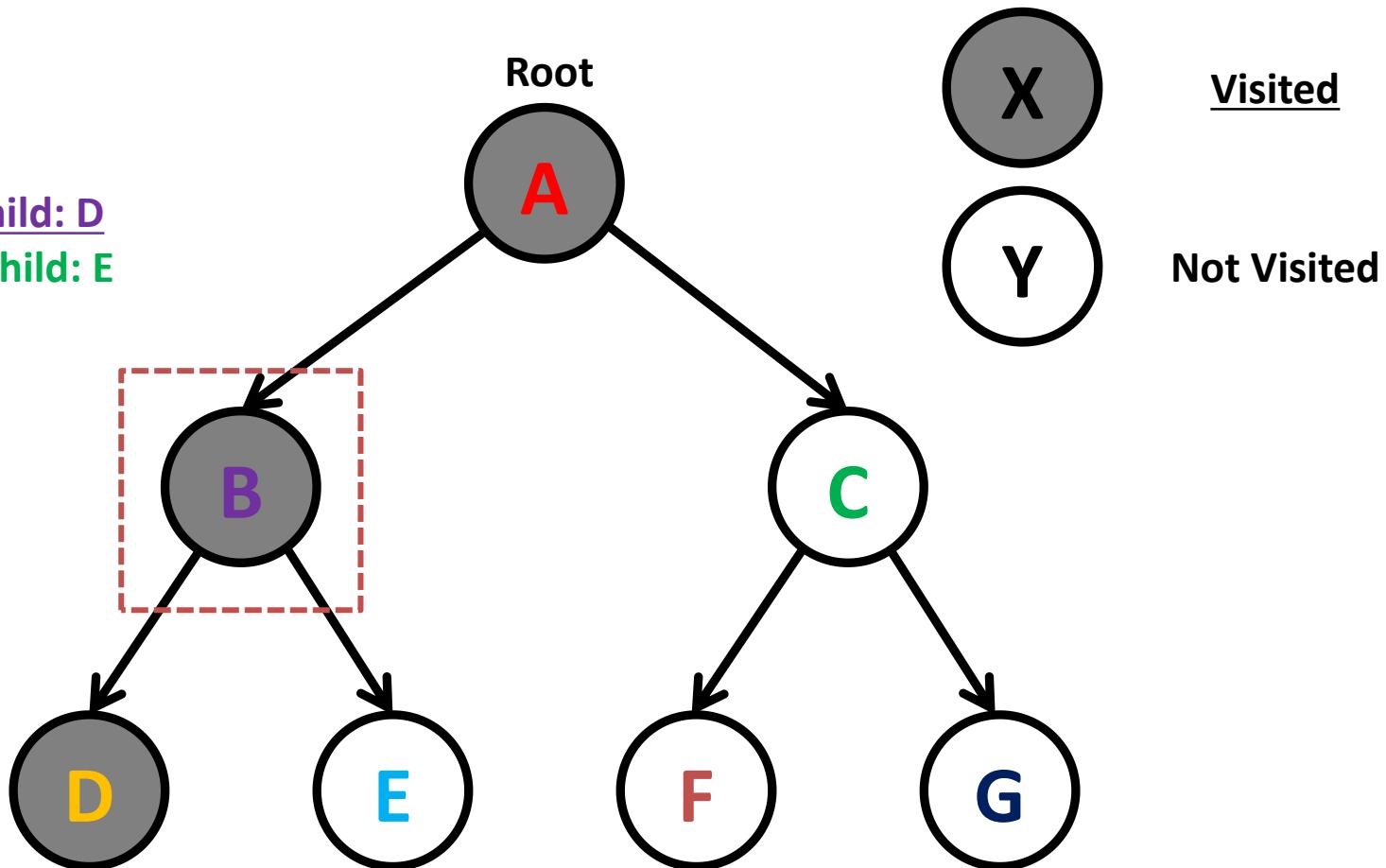
Node: B

Node's Left Child: D

Node's Right Child: E

Visited

Not Visited



Nodes traversed so far: A B D

Pre-order (Node, Left, Right)

Step 5:

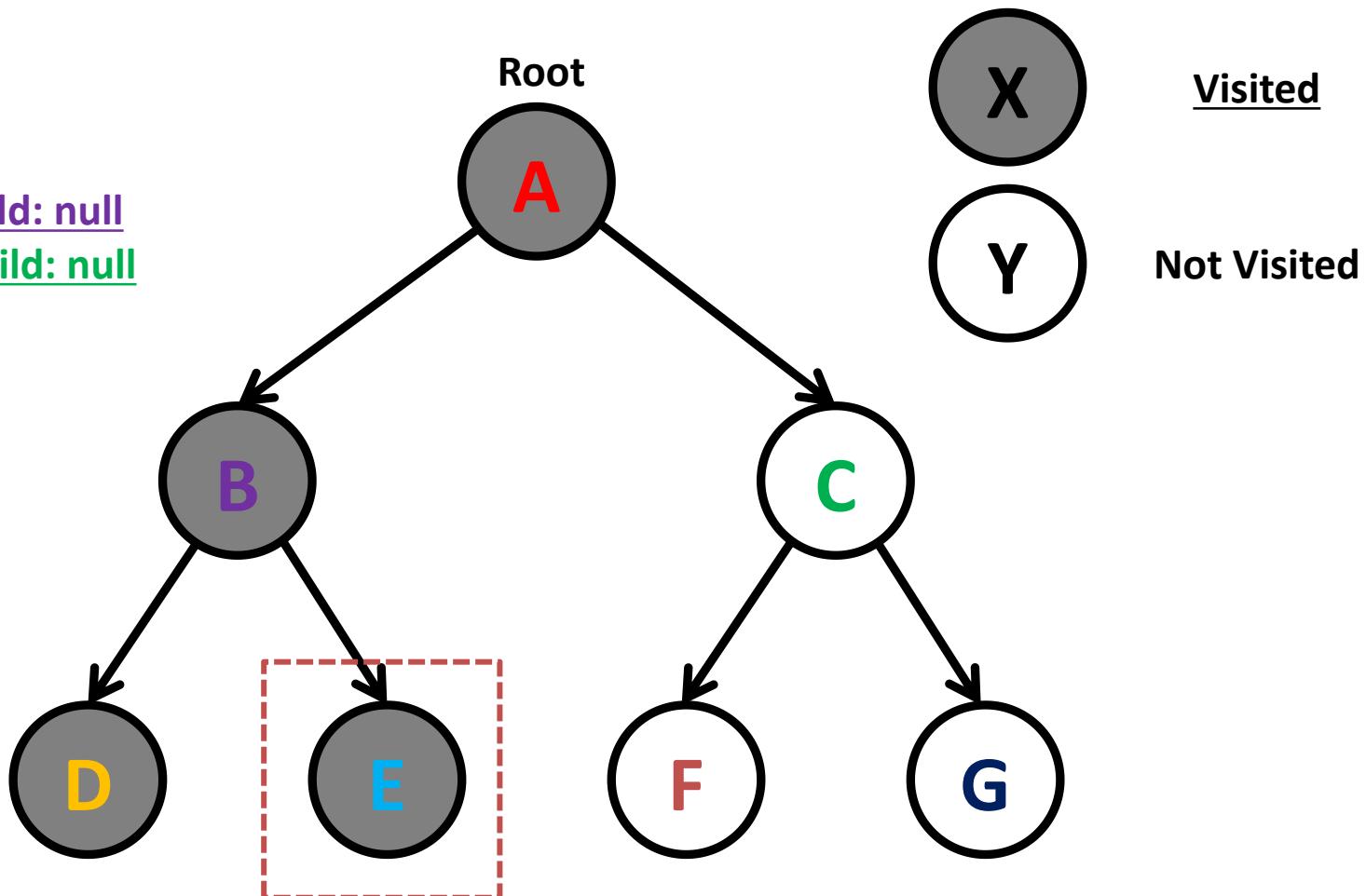
Node: E

Node's Left Child: null

Node's Right Child: null

Visited

Not Visited



Nodes traversed so far: A B D E

Pre-order (Node, Left, Right)

Step 6:

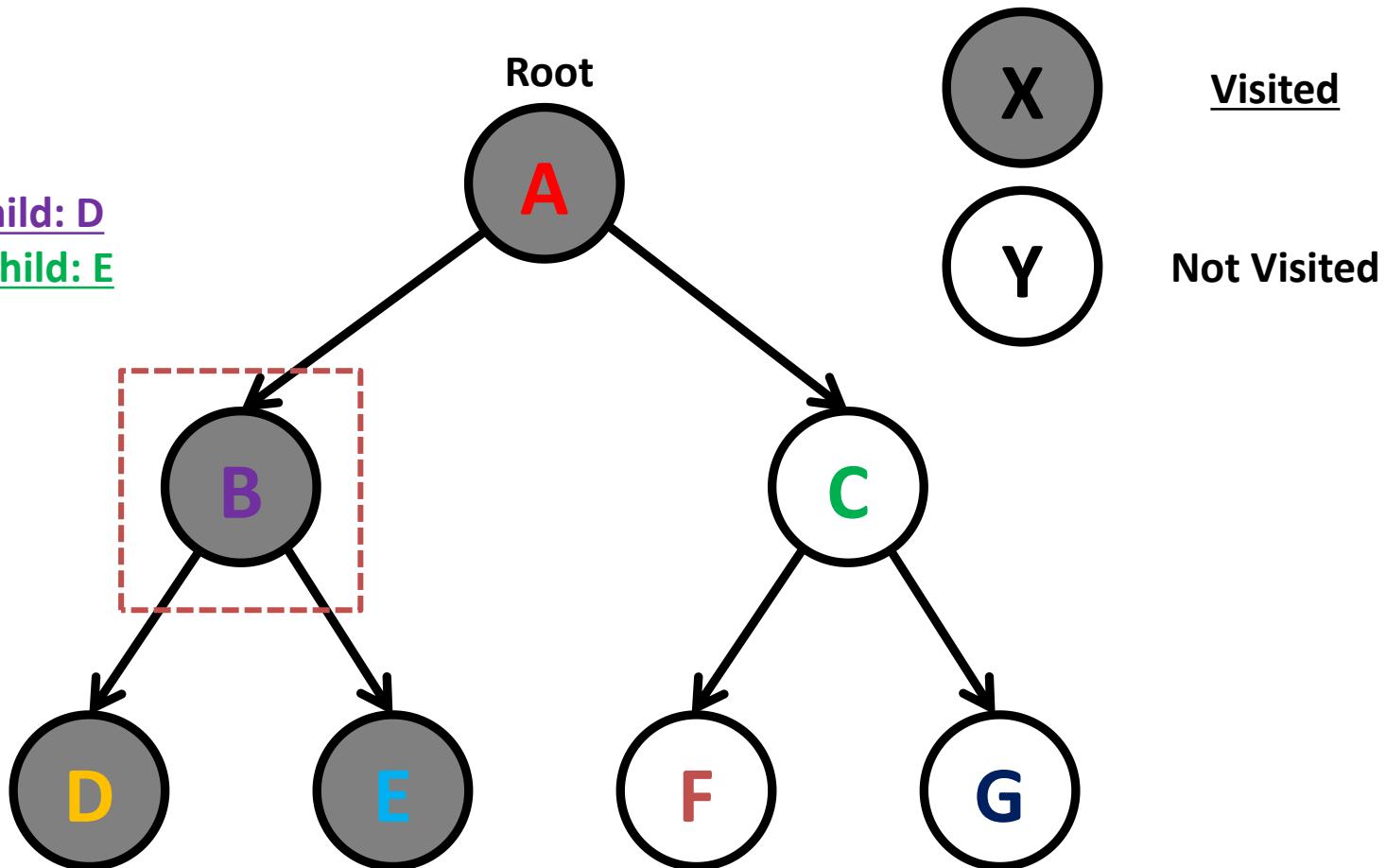
Node: B

Node's Left Child: D

Node's Right Child: E

Visited

Not Visited

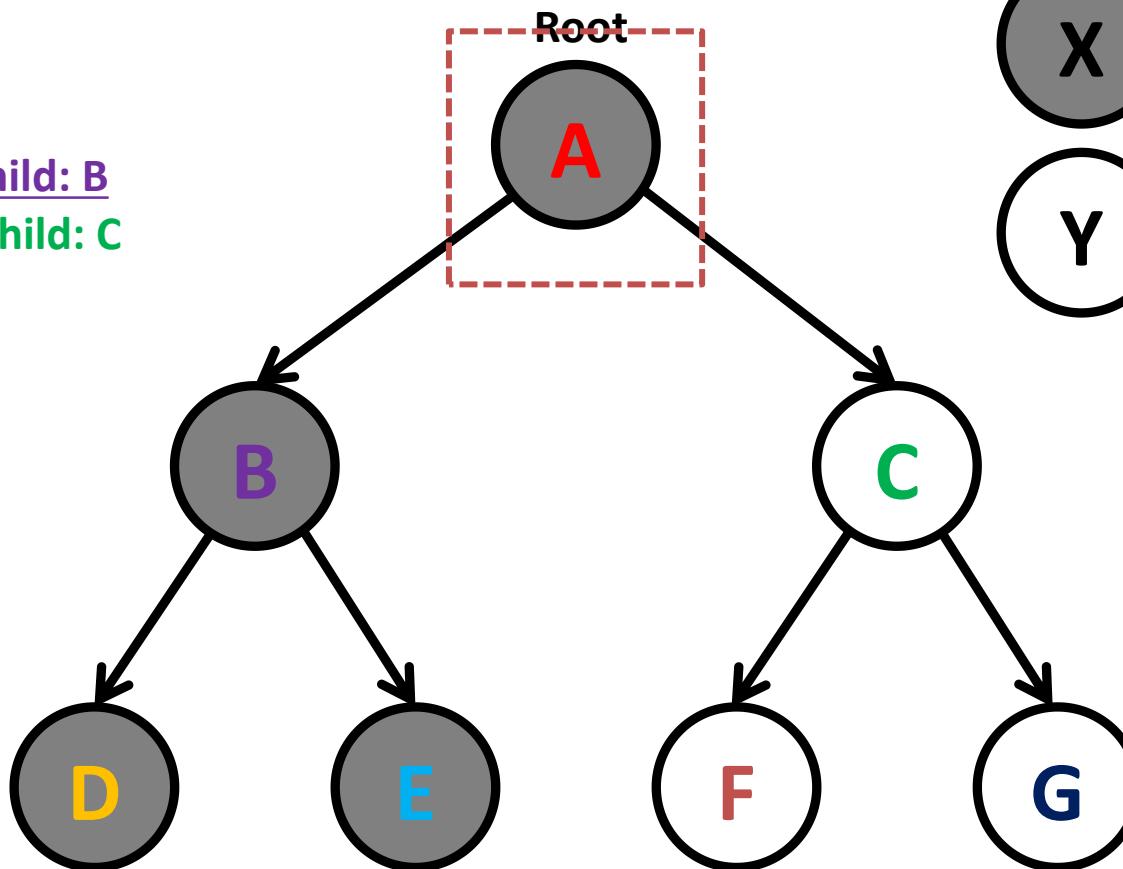
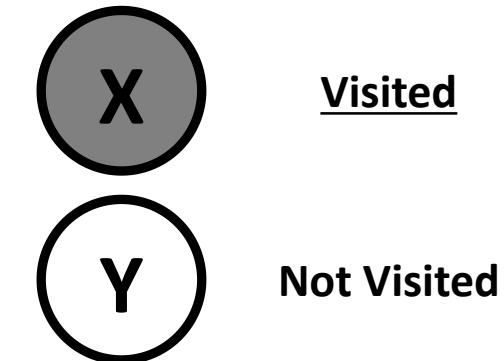


Nodes traversed so far: A B D E

Pre-order (Node, Left, Right)

Step 7:

Node: A
Node's Left Child: B
Node's Right Child: C



Nodes traversed so far: **A** **B** **D** **E**

Pre-order (Node, Left, Right)

Step 8:

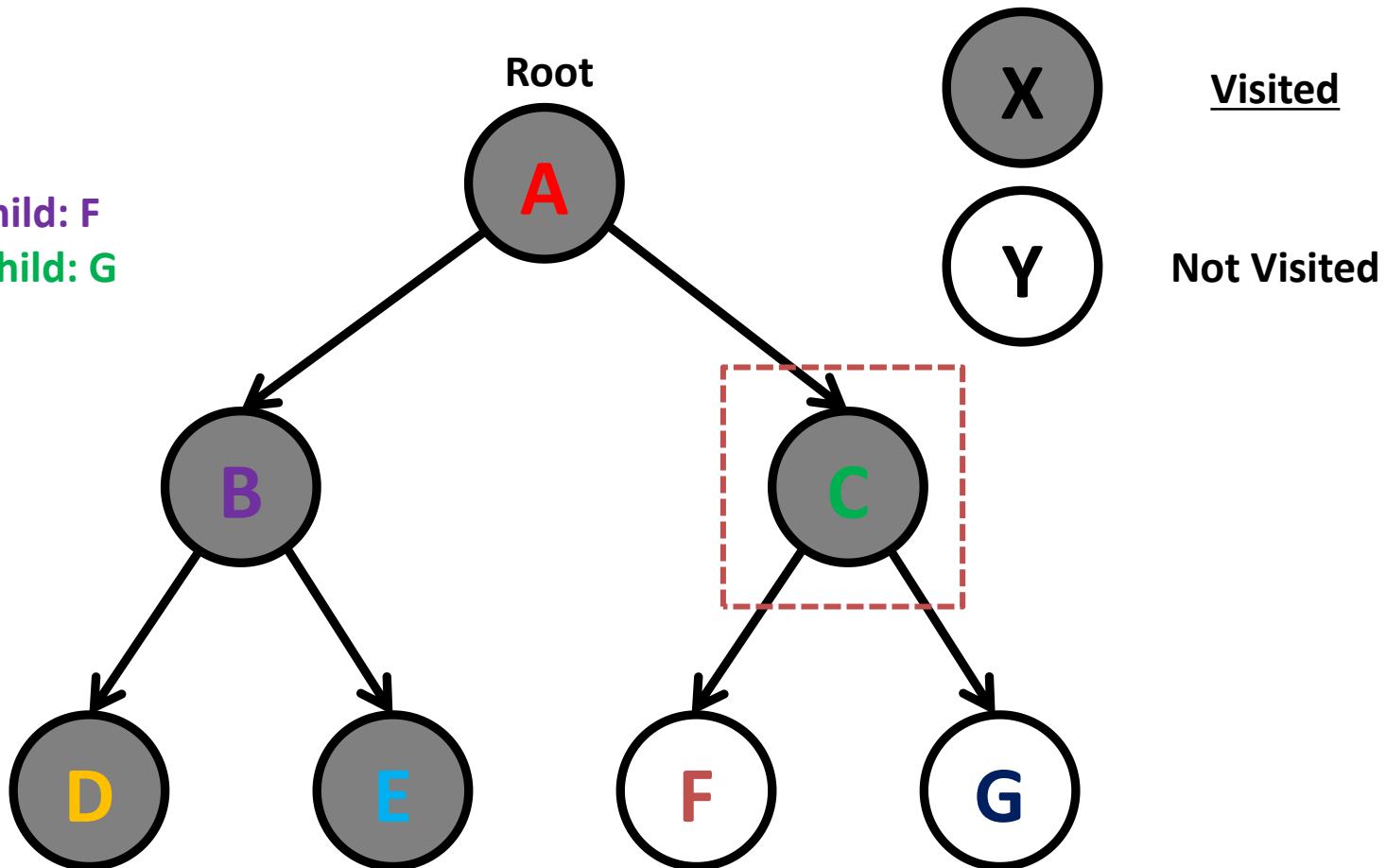
Node: C

Node's Left Child: F

Node's Right Child: G

Visited

Not Visited



Nodes traversed so far: A B D E C

Pre-order (Node, Left, Right)

Step 9:

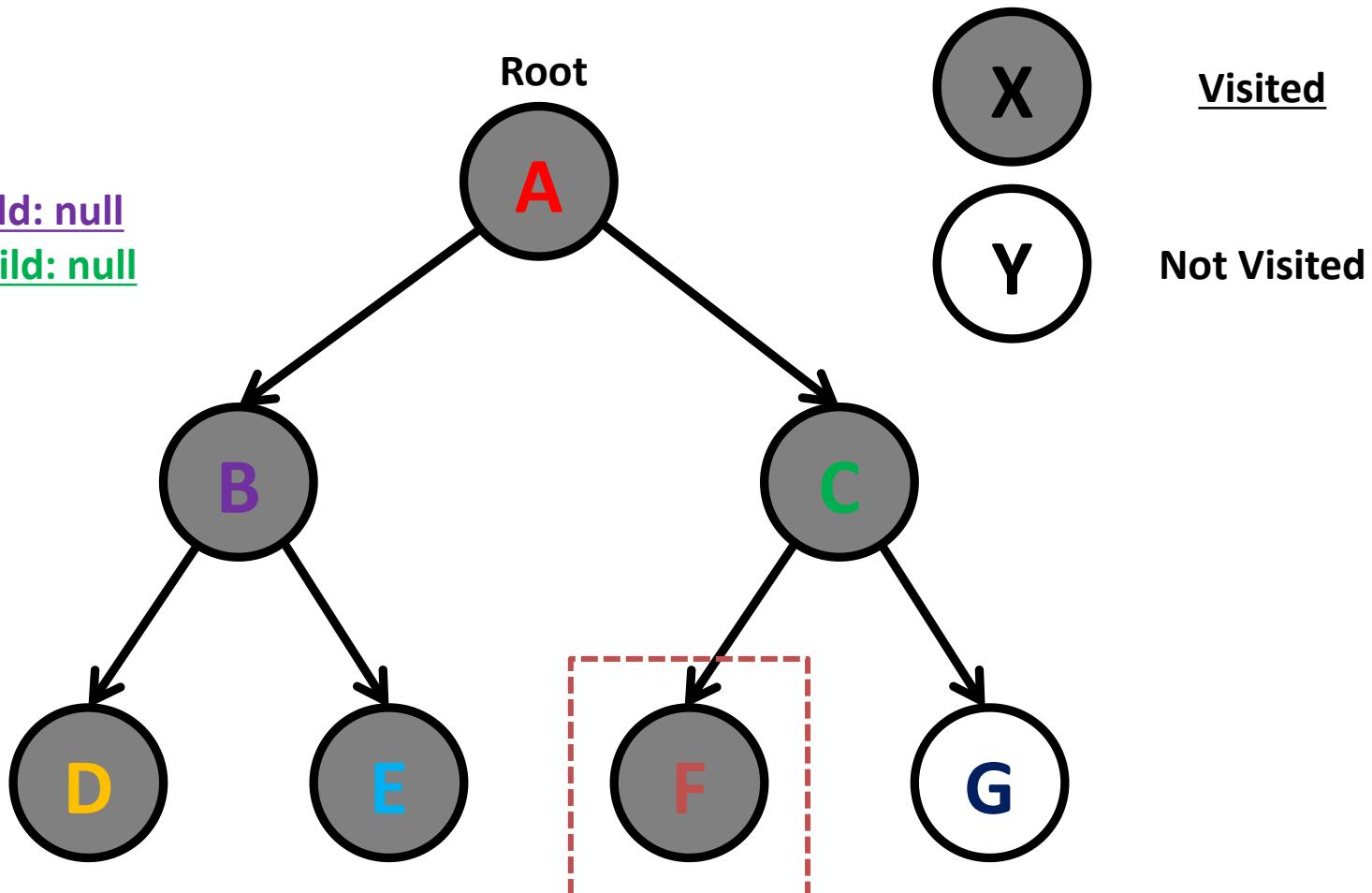
Node: F

Node's Left Child: null

Node's Right Child: null

Visited

Not Visited



Nodes traversed so far: A B D E C F

Pre-order (Node, Left, Right)

Step 10:

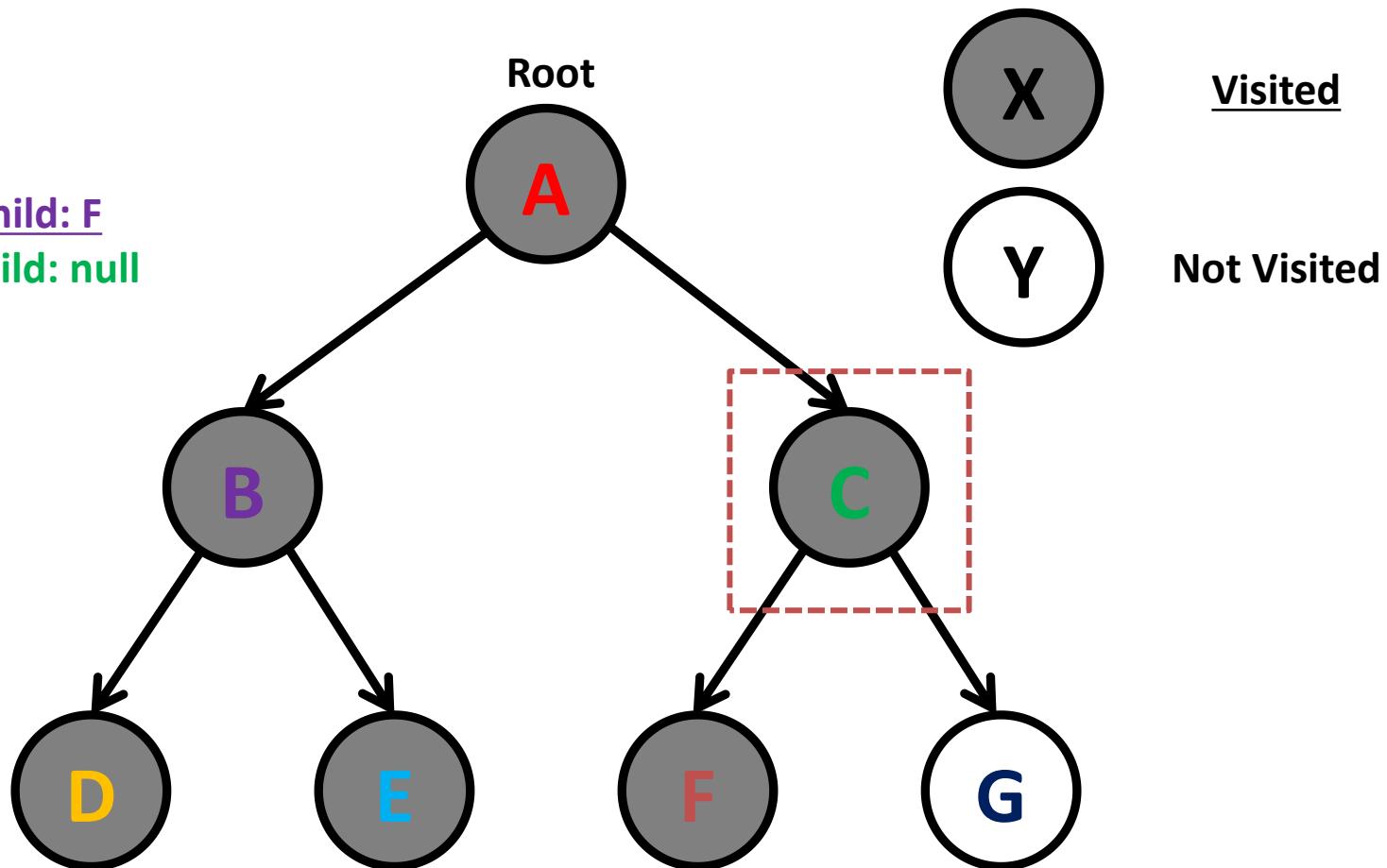
Node: C

Node's Left Child: F

Node's Right Child: null

Visited

Not Visited



Nodes traversed so far: A B D E C F

Pre-order (Node, Left, Right)

Step 11:

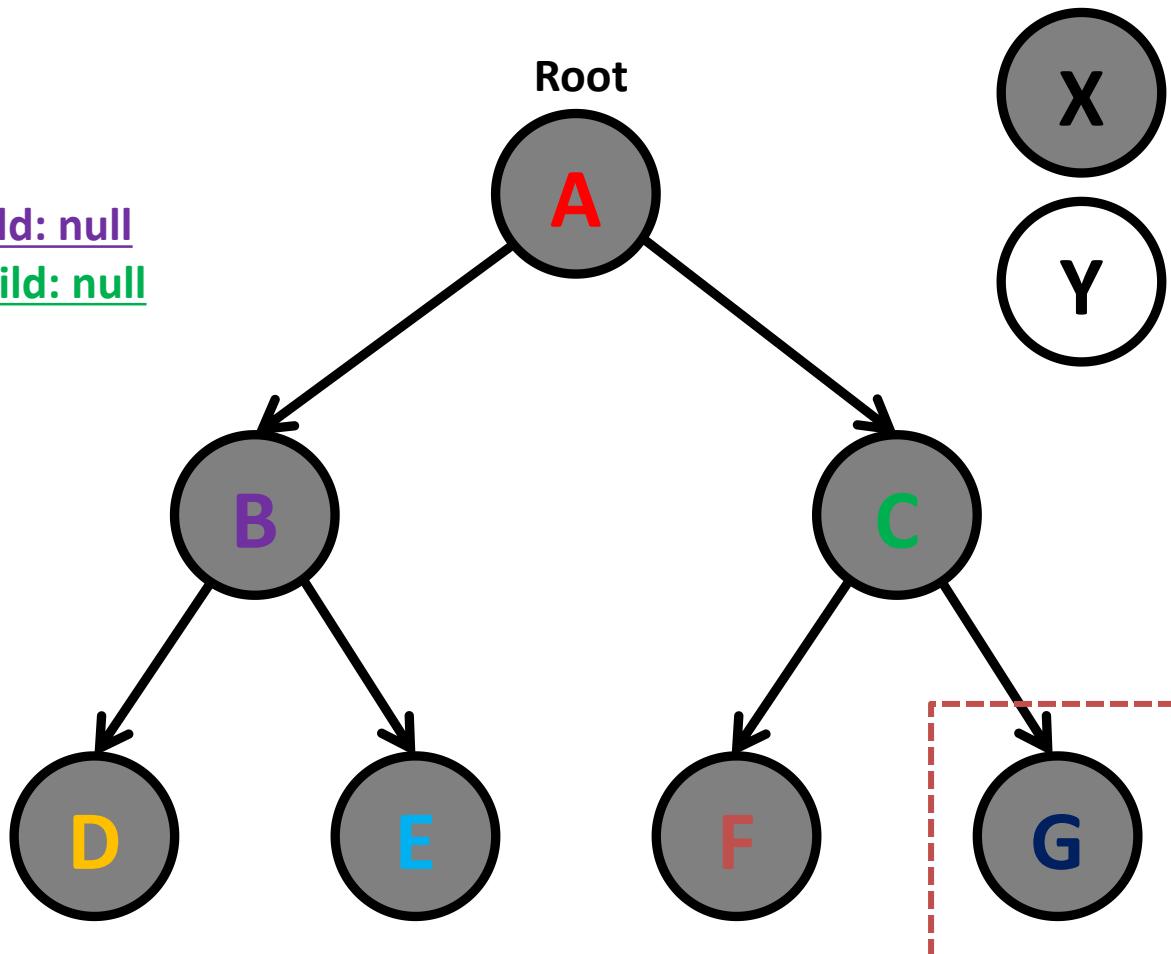
Node: G

Node's Left Child: null

Node's Right Child: null

Visited

Not Visited



Nodes traversed so far: A B D E C F G

Pre-order (Node, Left, Right)

Step 12:

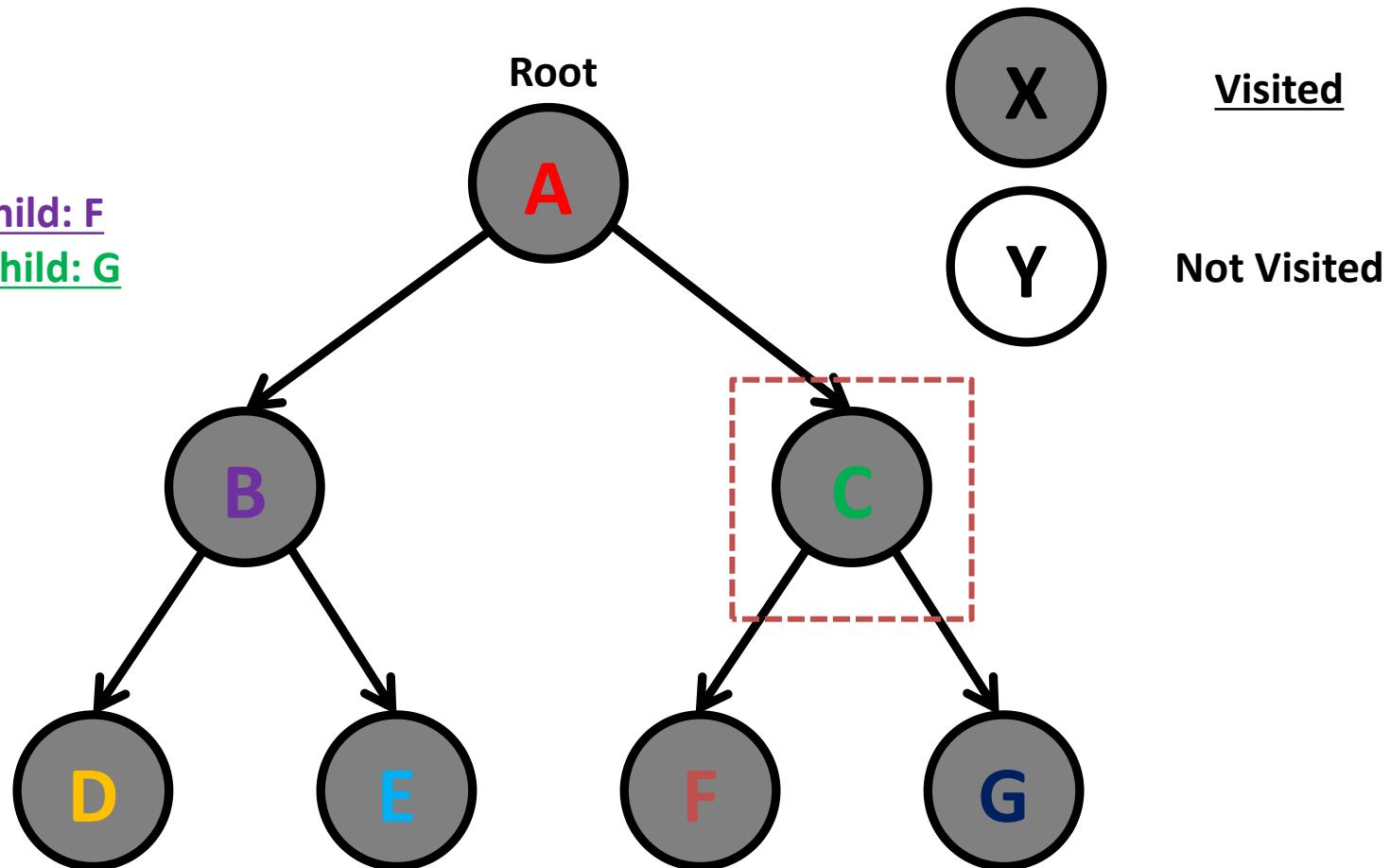
Node: C

Node's Left Child: F

Node's Right Child: G

Visited

Not Visited



Nodes traversed so far: A B D E C F G

Pre-order (Node, Left, Right)

Step 13:

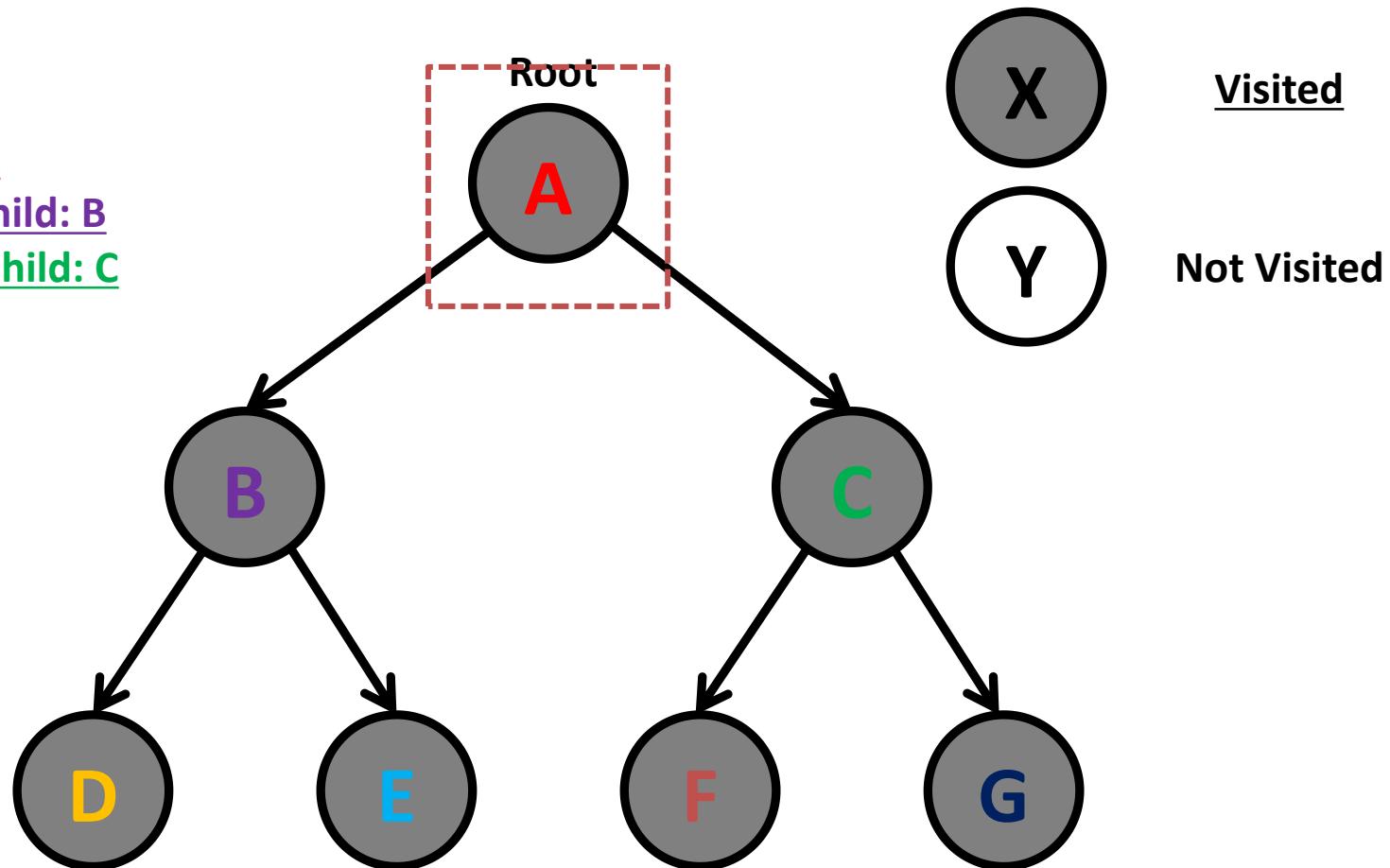
Node: A

Node's Left Child: B

Node's Right Child: C

Visited

Not Visited



Nodes traversed so far: A B D E C F G

In-order (Left, Node, Right)

Step 1:

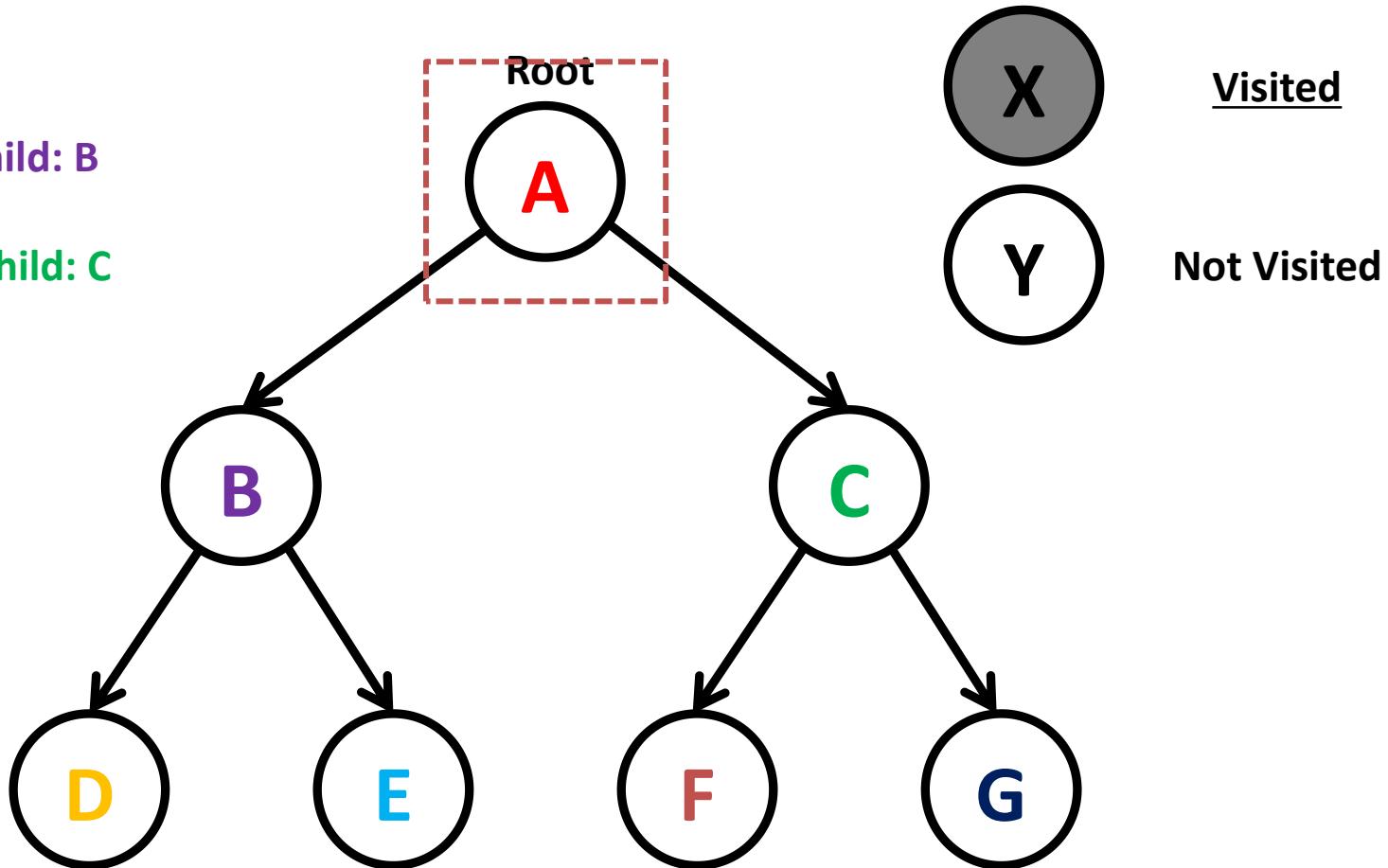
Node's Left Child: B

Node: A

Node's Right Child: C

Visited

Not Visited



Nodes traversed so far:

In-order (Left, Node, Right)

Step 2:

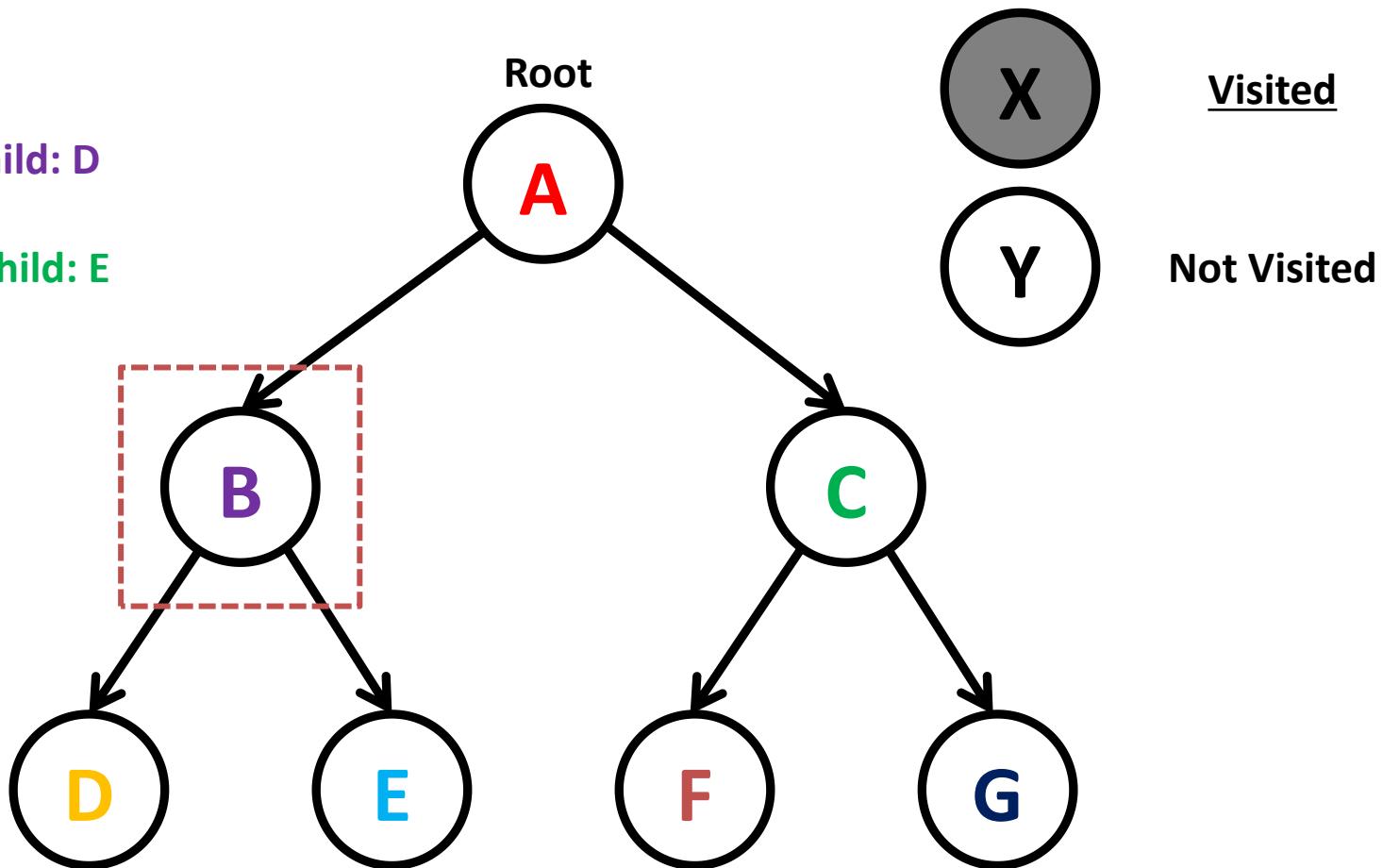
Node's Left Child: D

Node: B

Node's Right Child: E

Visited

Not Visited



Nodes traversed so far:

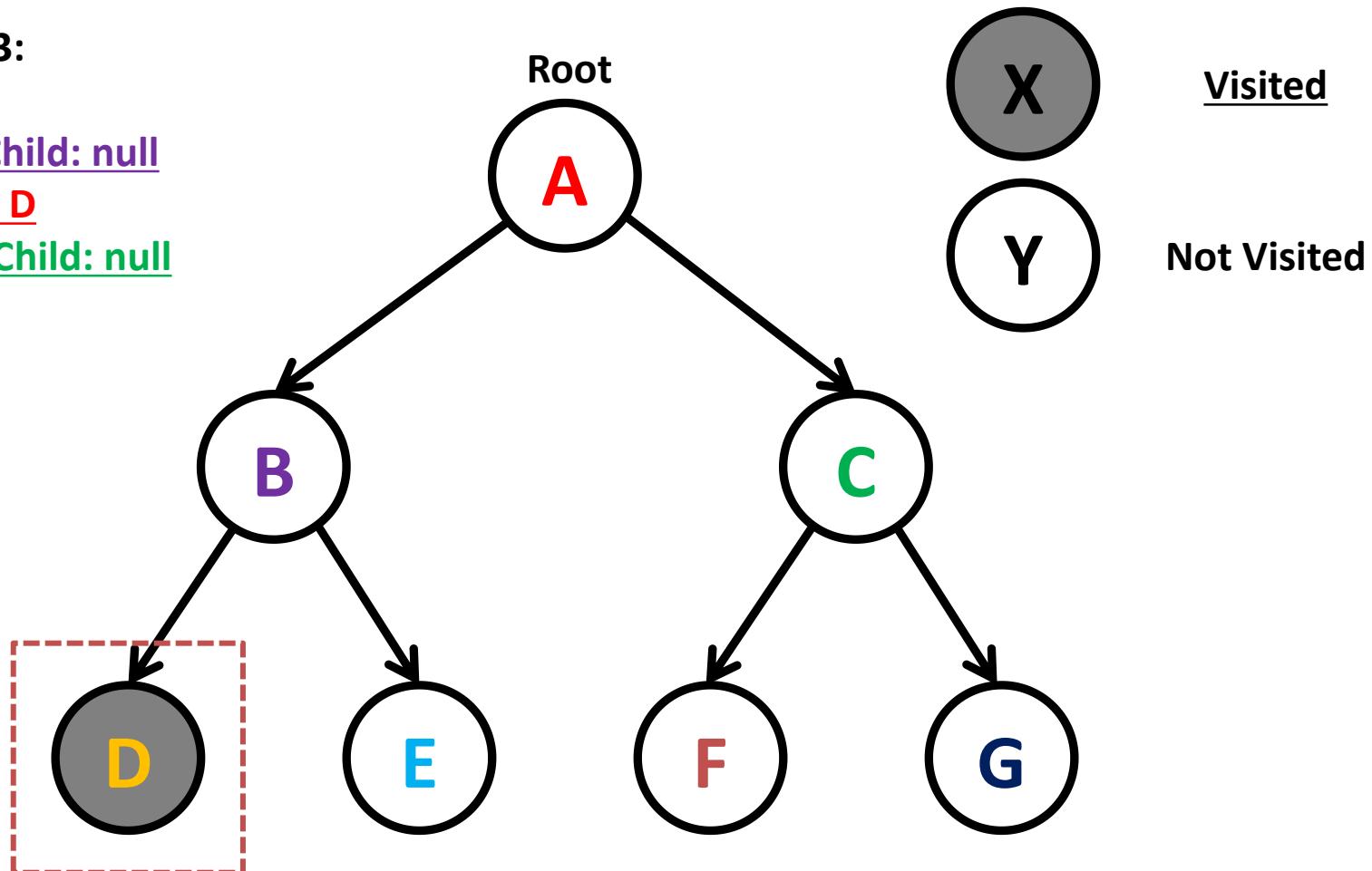
In-order (Left, Node, Right)

Step 3:

Node's Left Child: null

Node: D

Node's Right Child: null



Nodes traversed so far: D

In-order (Left, Node, Right)

Step 4:

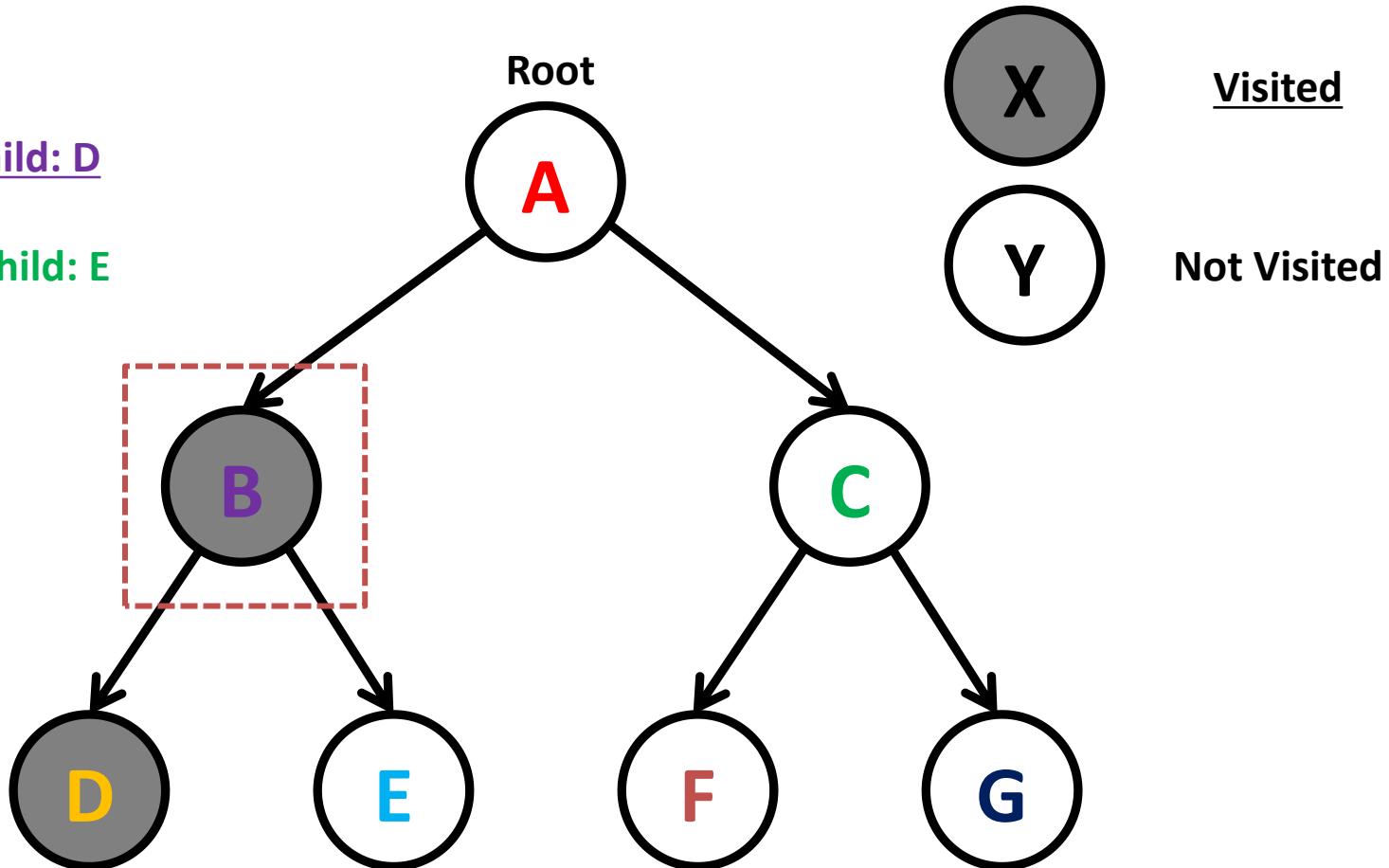
Node's Left Child: D

Node: B

Node's Right Child: E

Visited

Not Visited



Nodes traversed so far: D B

In-order (Left, Node, Right)

Step 5:

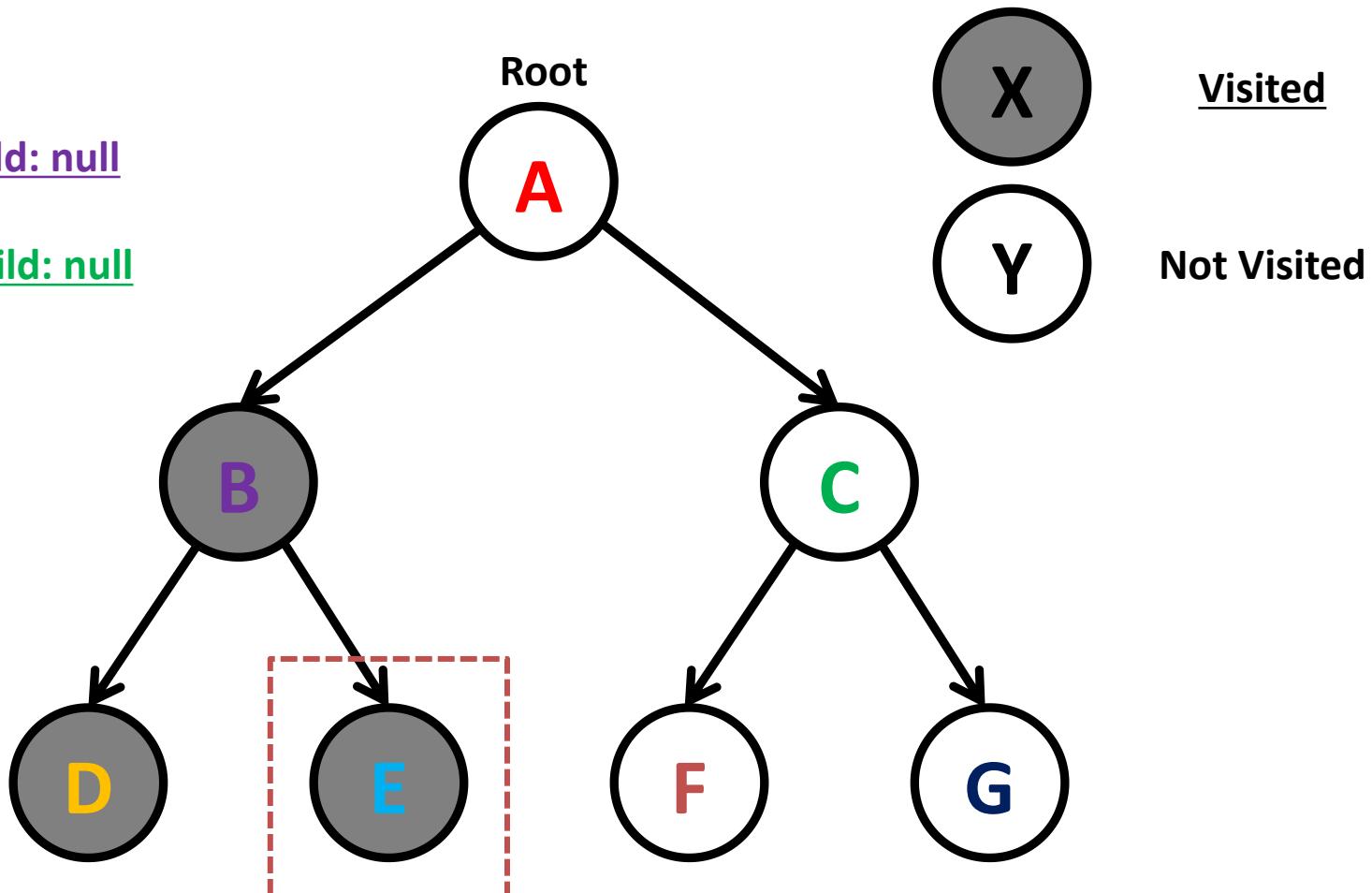
Node's Left Child: null

Node: E

Node's Right Child: null

Visited

Not Visited



Nodes traversed so far: D B E

In-order (Left, Node, Right)

Step 6:

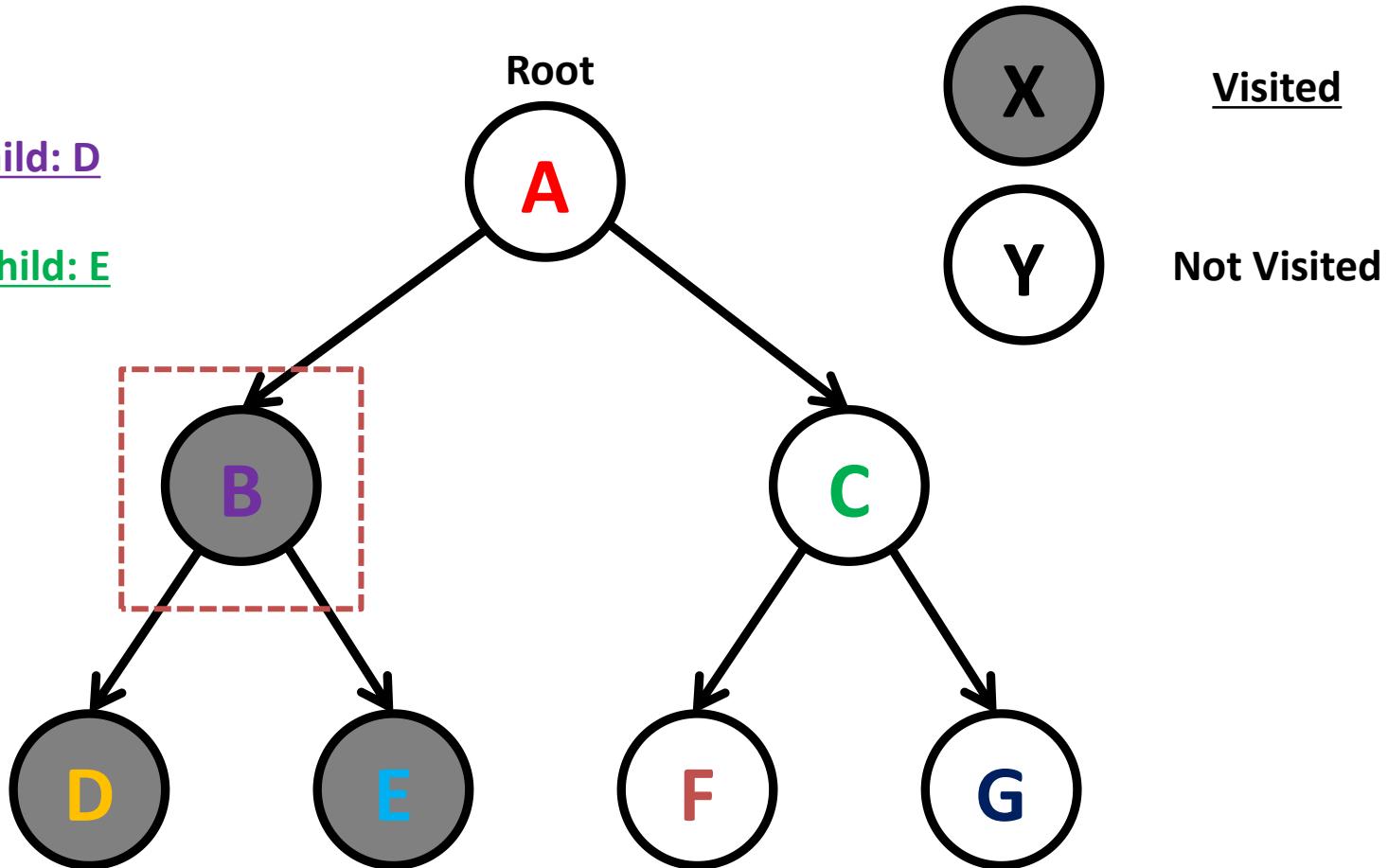
Node's Left Child: D

Node: B

Node's Right Child: E

Visited

Not Visited



Nodes traversed so far: D B E

In-order (Left, Node, Right)

Step 7:

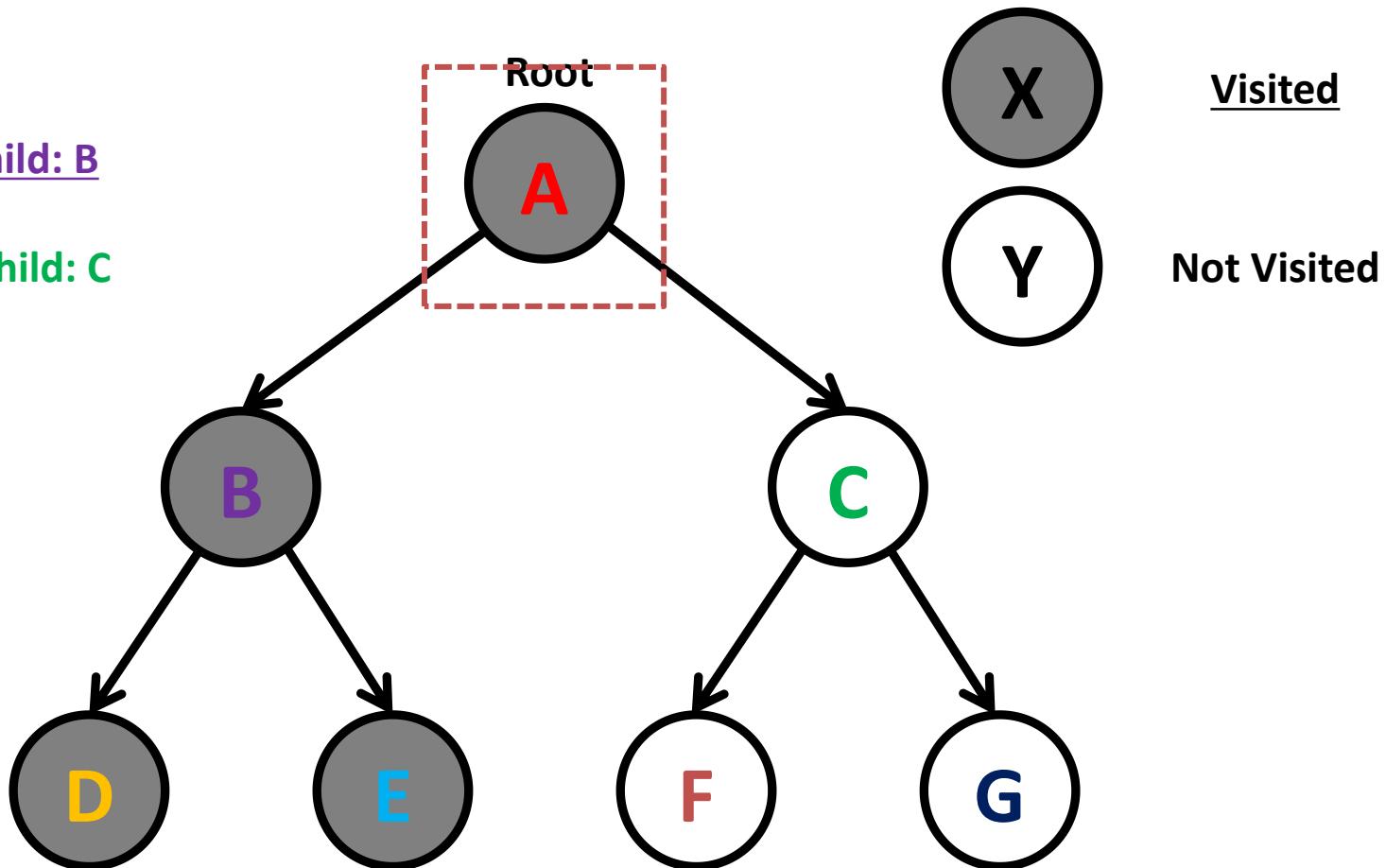
Node's Left Child: B

Node: A

Node's Right Child: C

Visited

Not Visited



Nodes traversed so far: D B E A

In-order (Left, Node, Right)

Step 8:

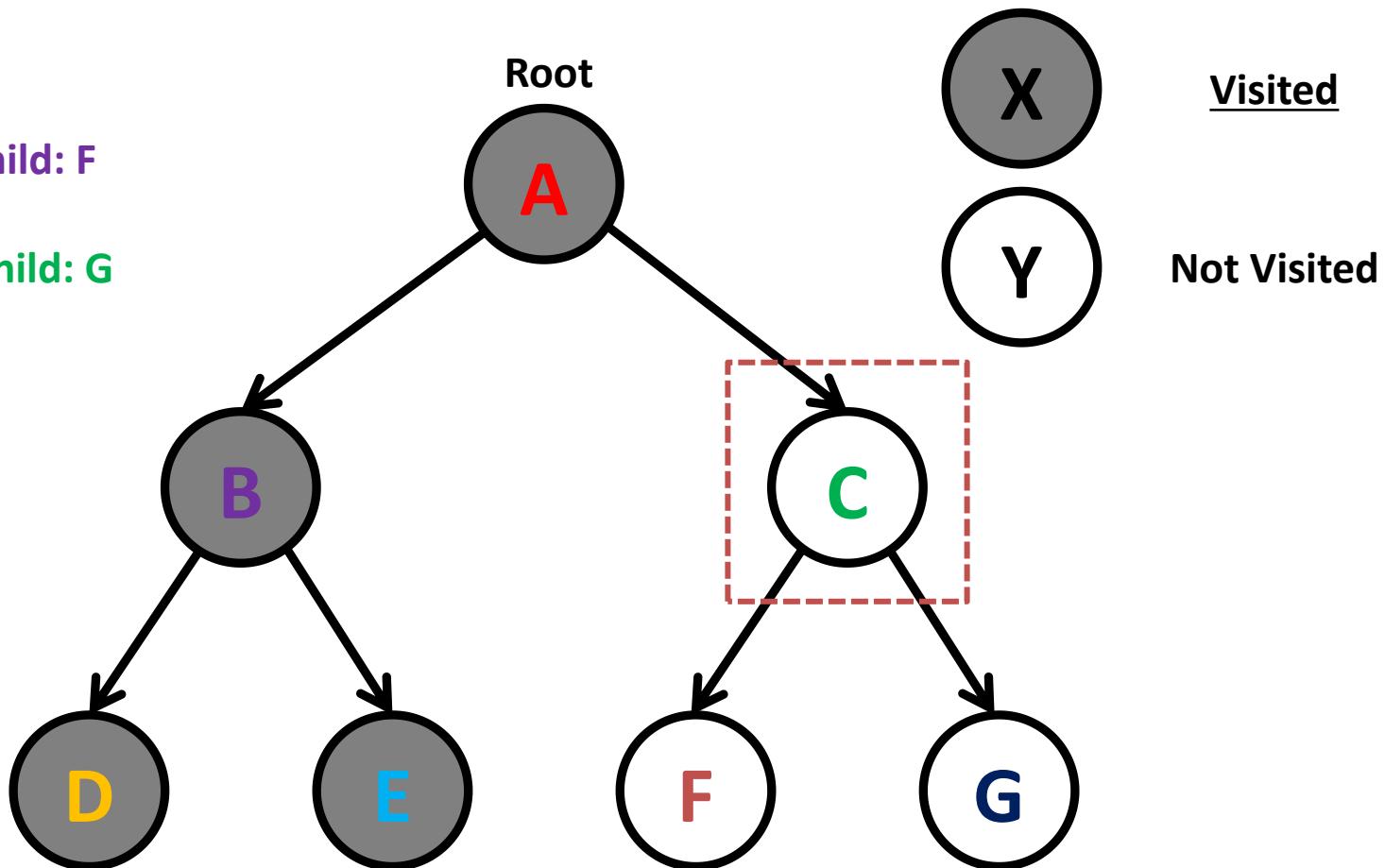
Node's Left Child: F

Node: C

Node's Right Child: G

Visited

Not Visited



Nodes traversed so far: D B E A

In-order (Left, Node, Right)

Step 9:

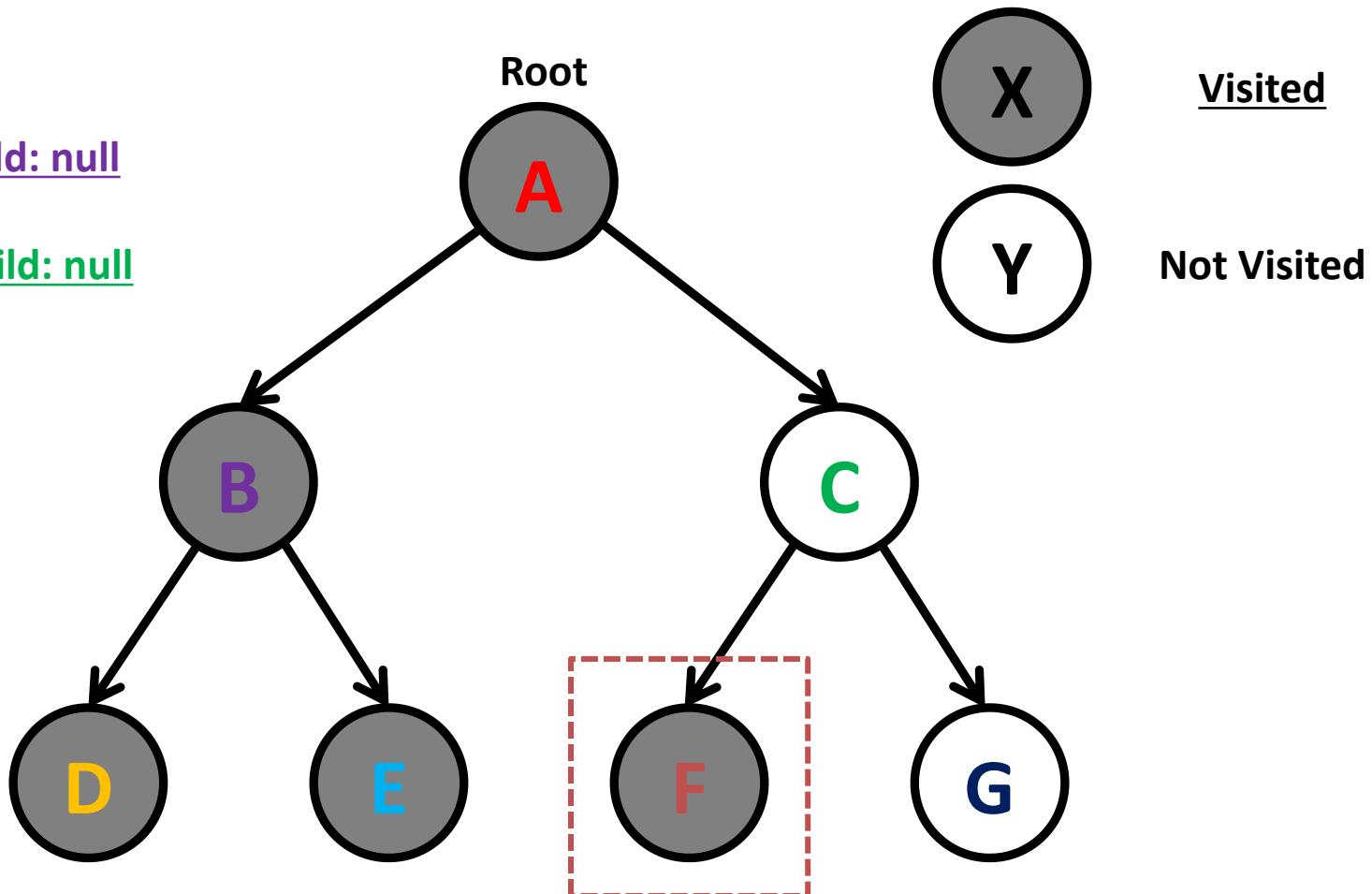
Node's Left Child: null

Node: F

Node's Right Child: null

Visited

Not Visited



Nodes traversed so far: D B E A F

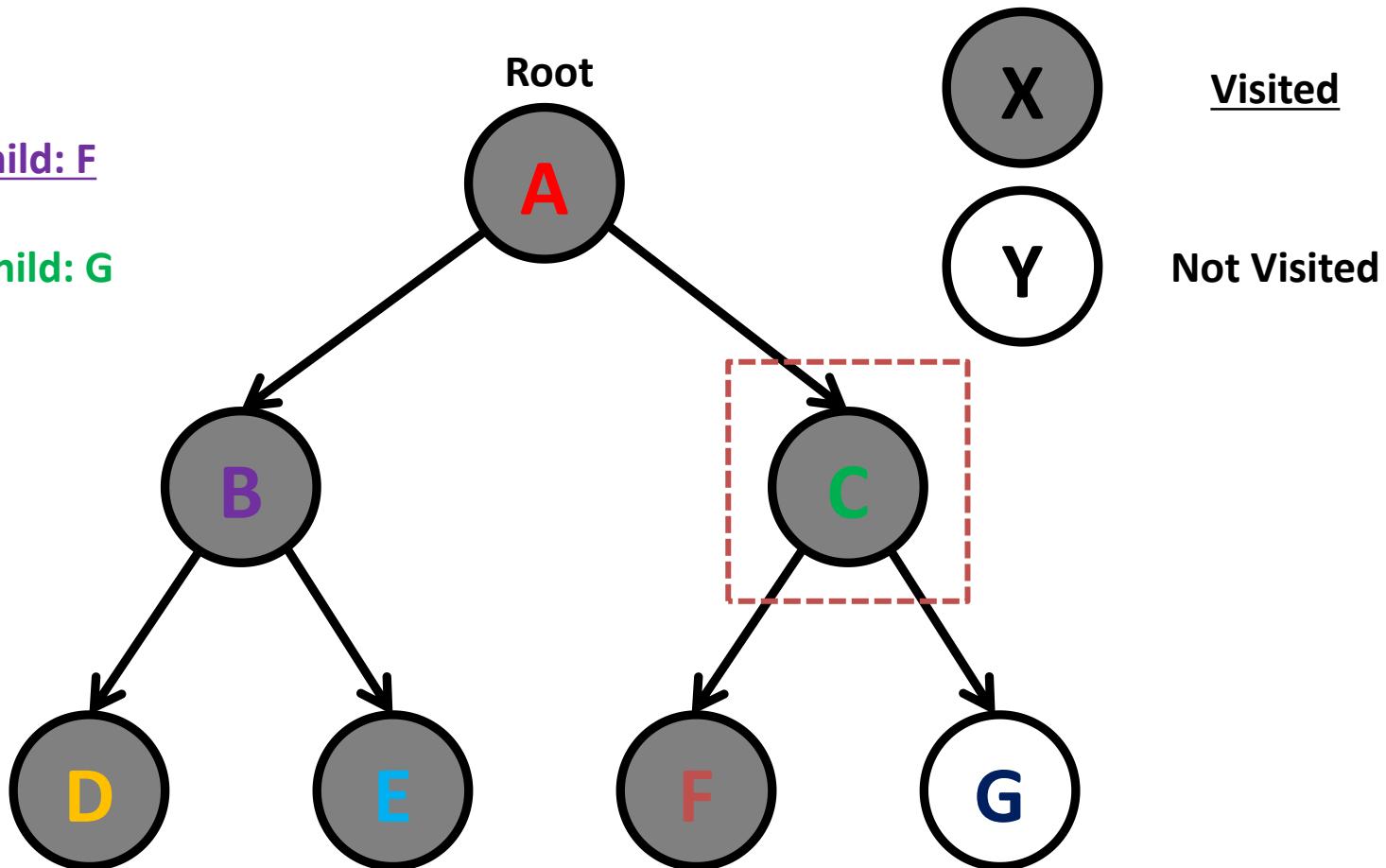
In-order (Left, Node, Right)

Step 10:

Node's Left Child: F

Node: C

Node's Right Child: G



Nodes traversed so far: D B E A F C

In-order (Left, Node, Right)

Step 11:

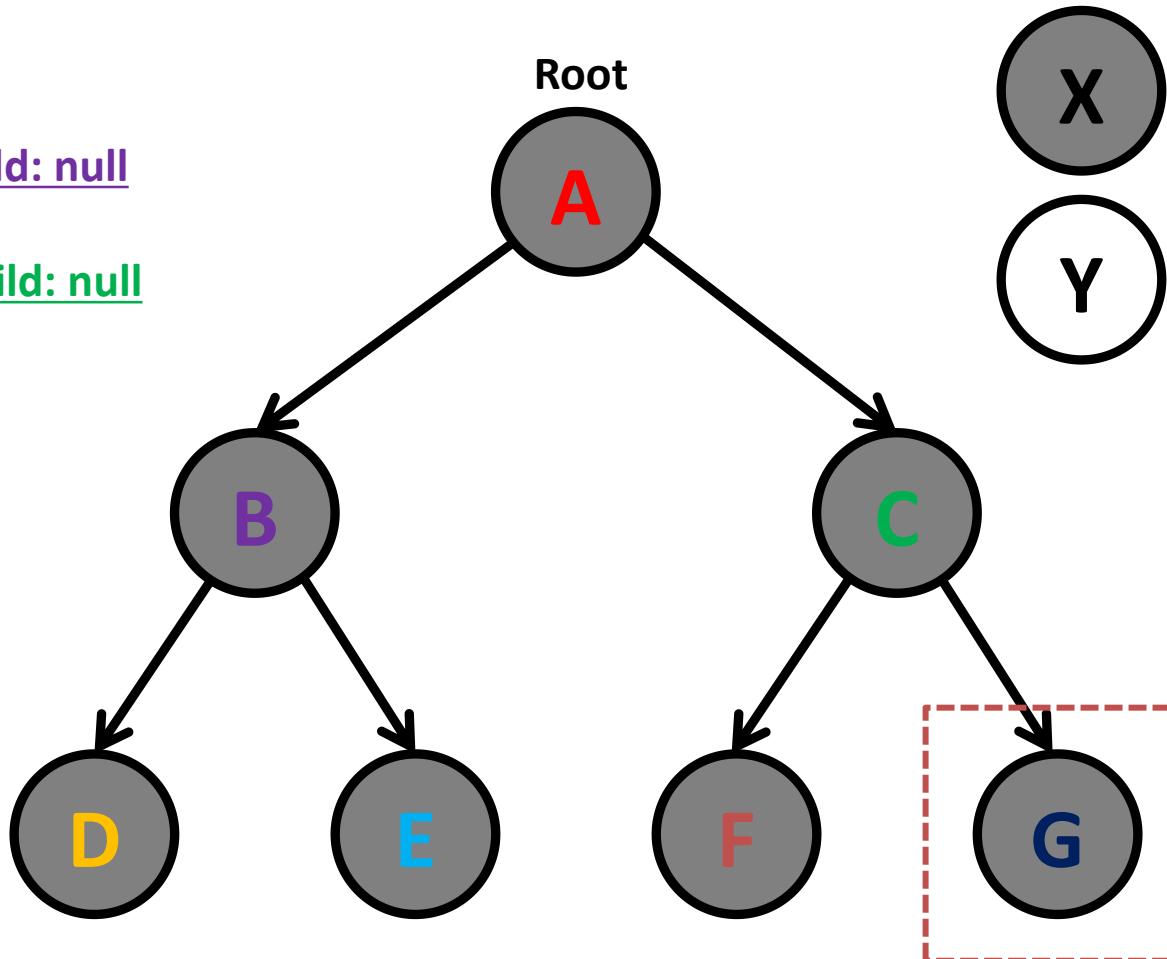
Node's Left Child: null

Node: G

Node's Right Child: null

Visited

Not Visited



Nodes traversed so far: D B E A F C G

In-order (Left, Node, Right)

Step 12:

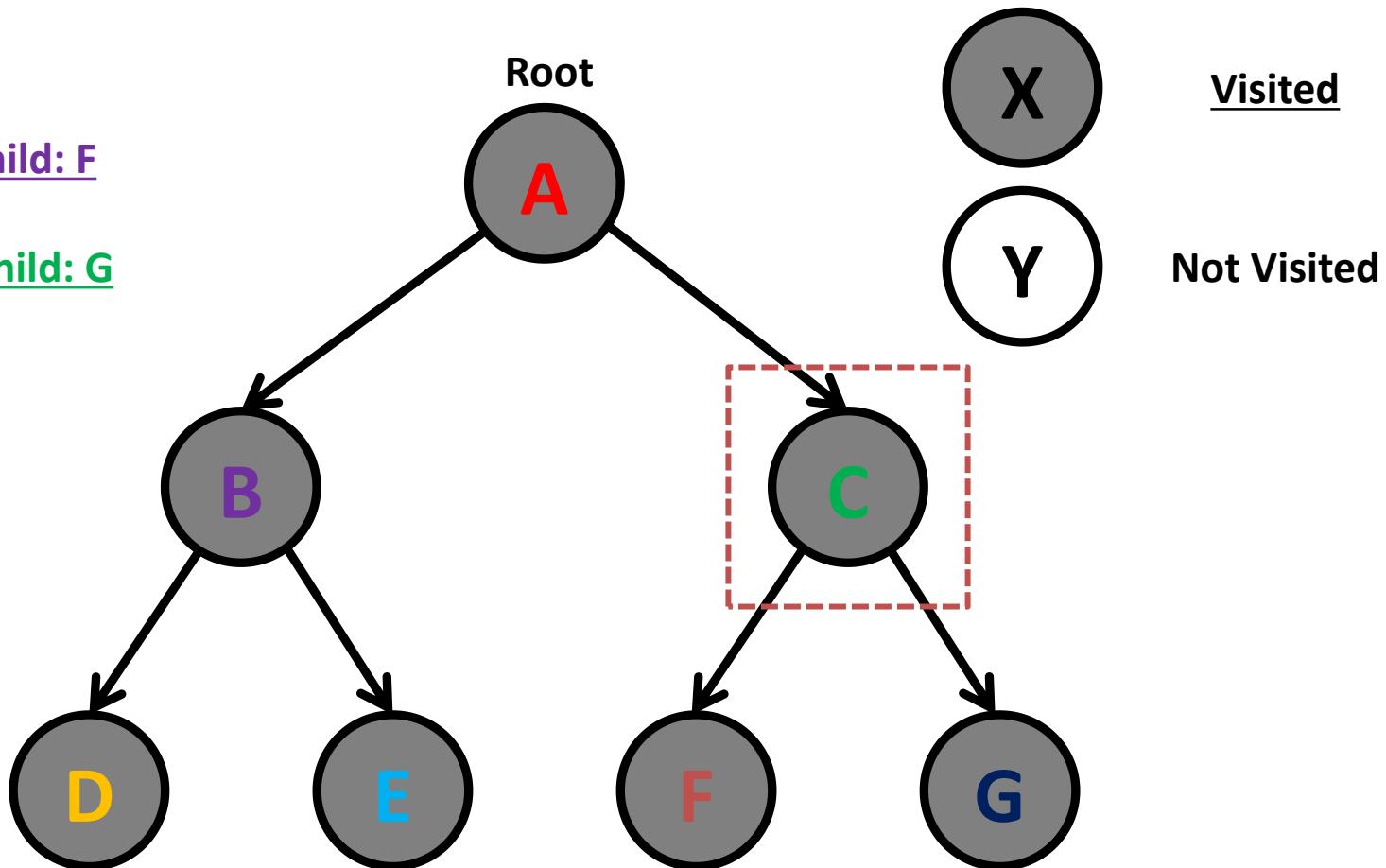
Node's Left Child: F

Node: C

Node's Right Child: G

Visited

Not Visited



Nodes traversed so far: D B E A F C G

In-order (Left, Node, Right)

Step 13:

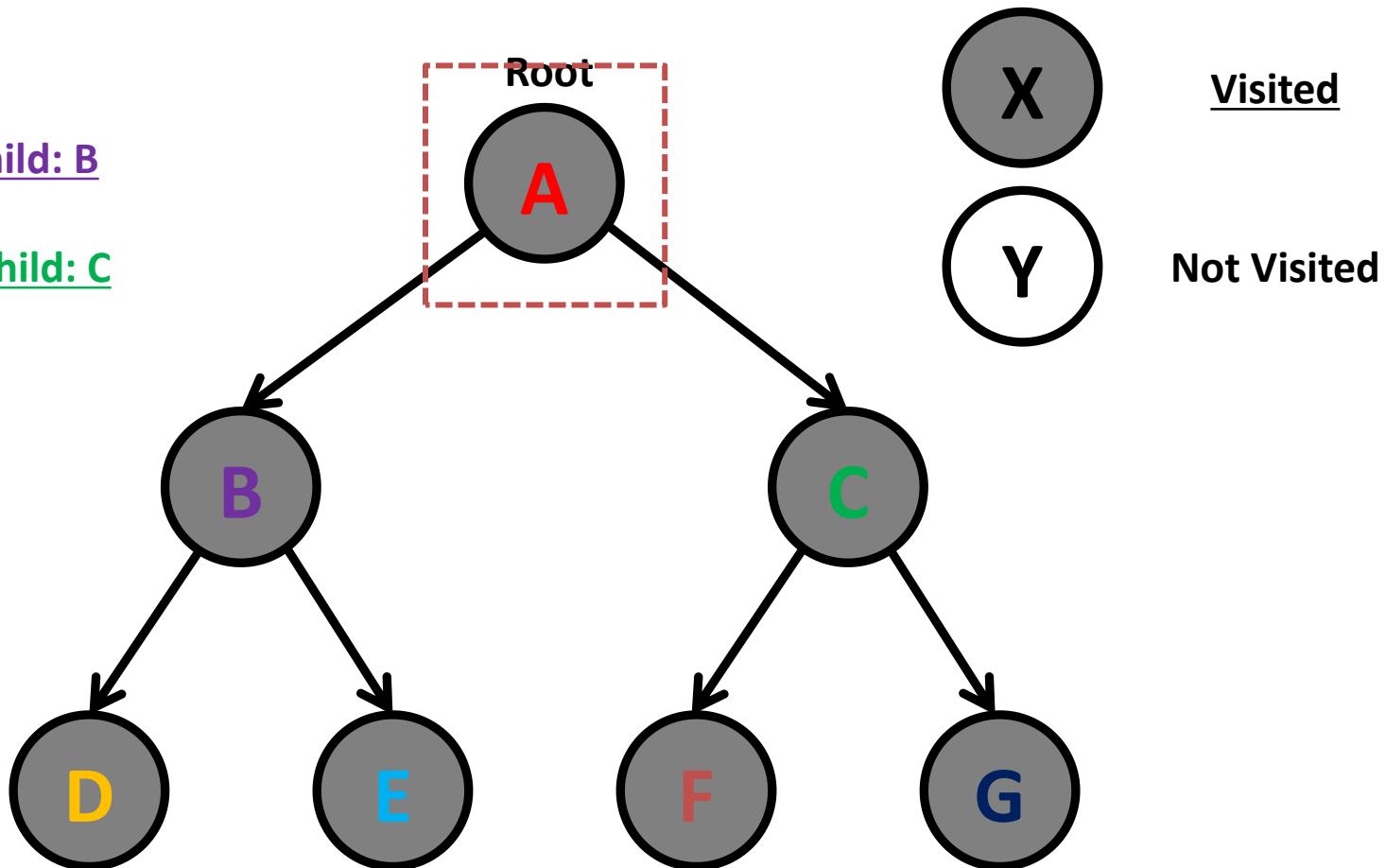
Node's Left Child: B

Node: A

Node's Right Child: C

Visited

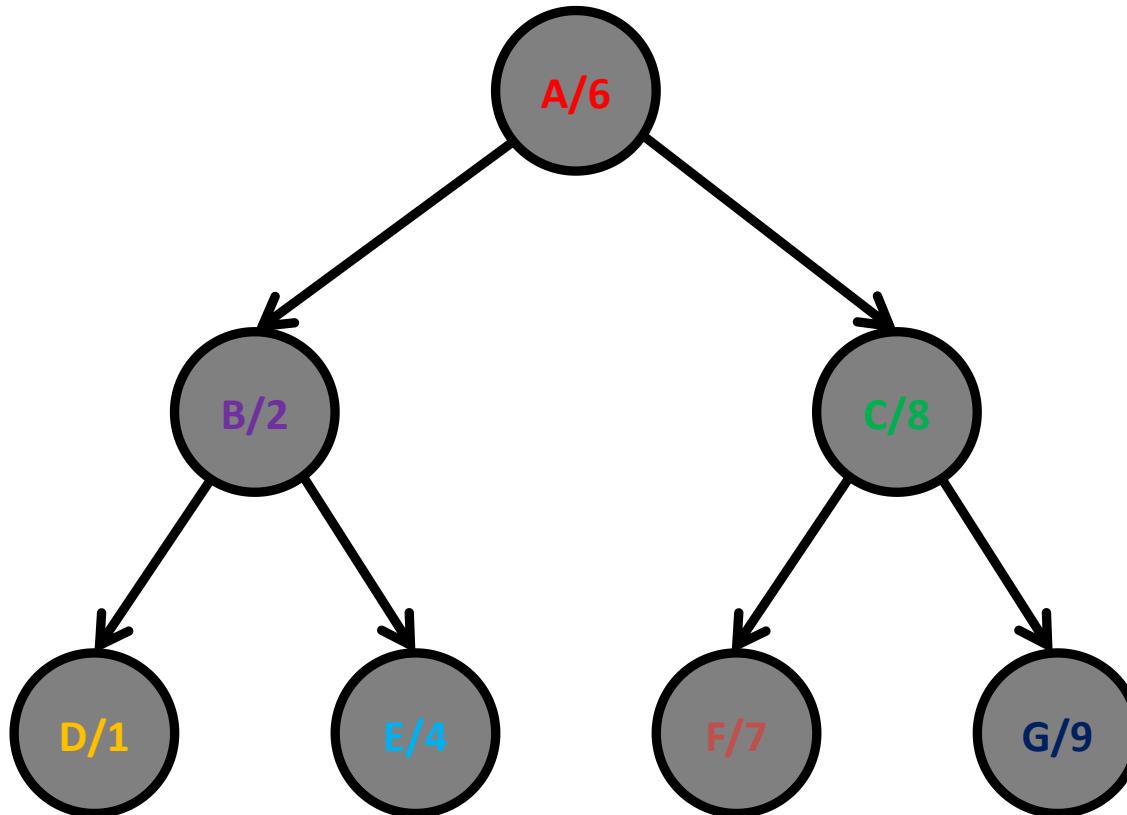
Not Visited



Nodes traversed so far: D B E A F C G

In-order (Left, Node, Right)

If we had BST with numerical keys -> sorted order!



In-order traversal: **D/1 B/2 E/4 A/6 F/7 C/8 G/9**

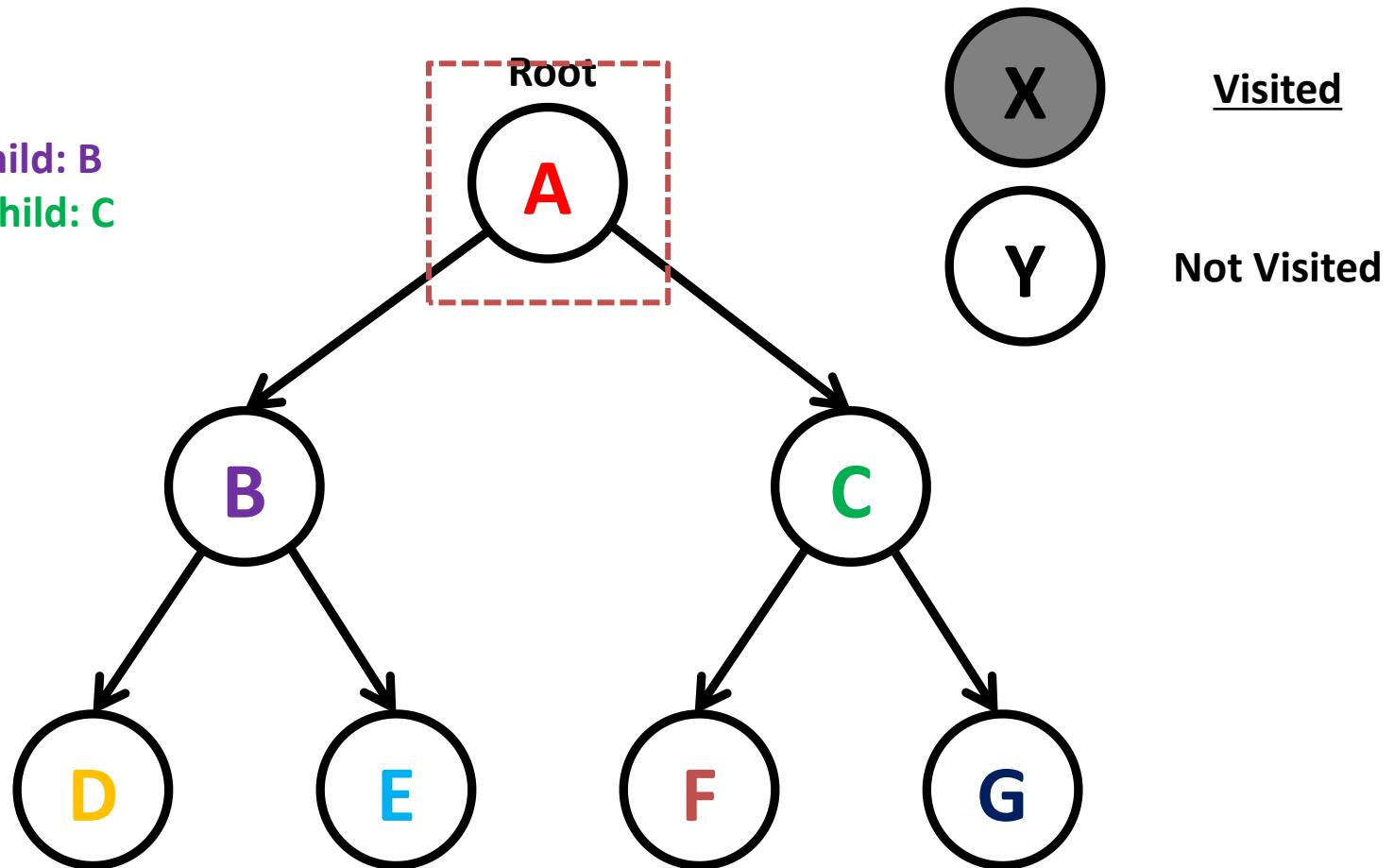
Post-order (Left, Right, Node)

Step 1:

Node's Left Child: B

Node's Right Child: C

Node: A



Nodes traversed so far:

Post-order (Left, Right, Node)

Step 2:

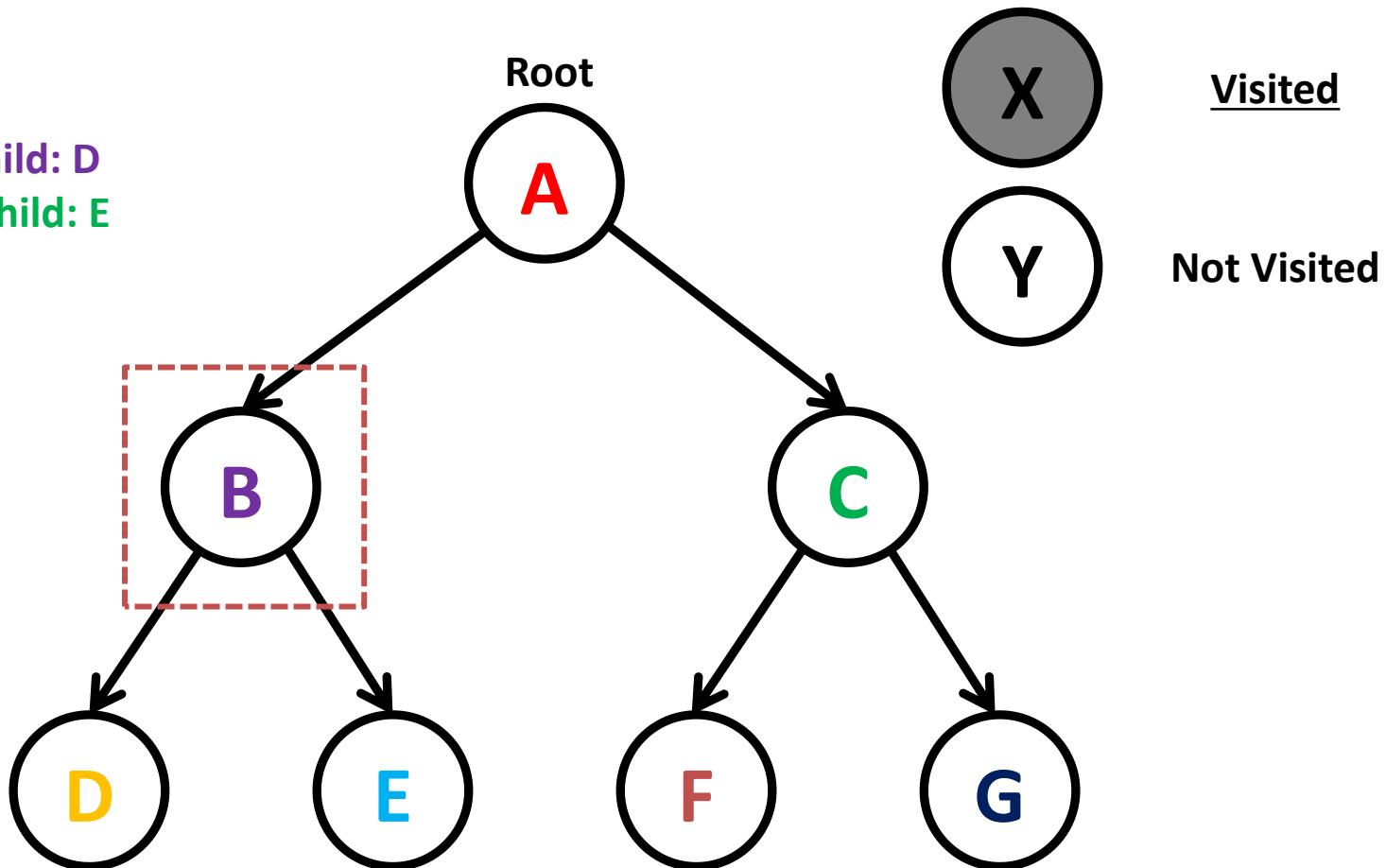
Node's Left Child: D

Node's Right Child: E

Node: B

Visited

Not Visited

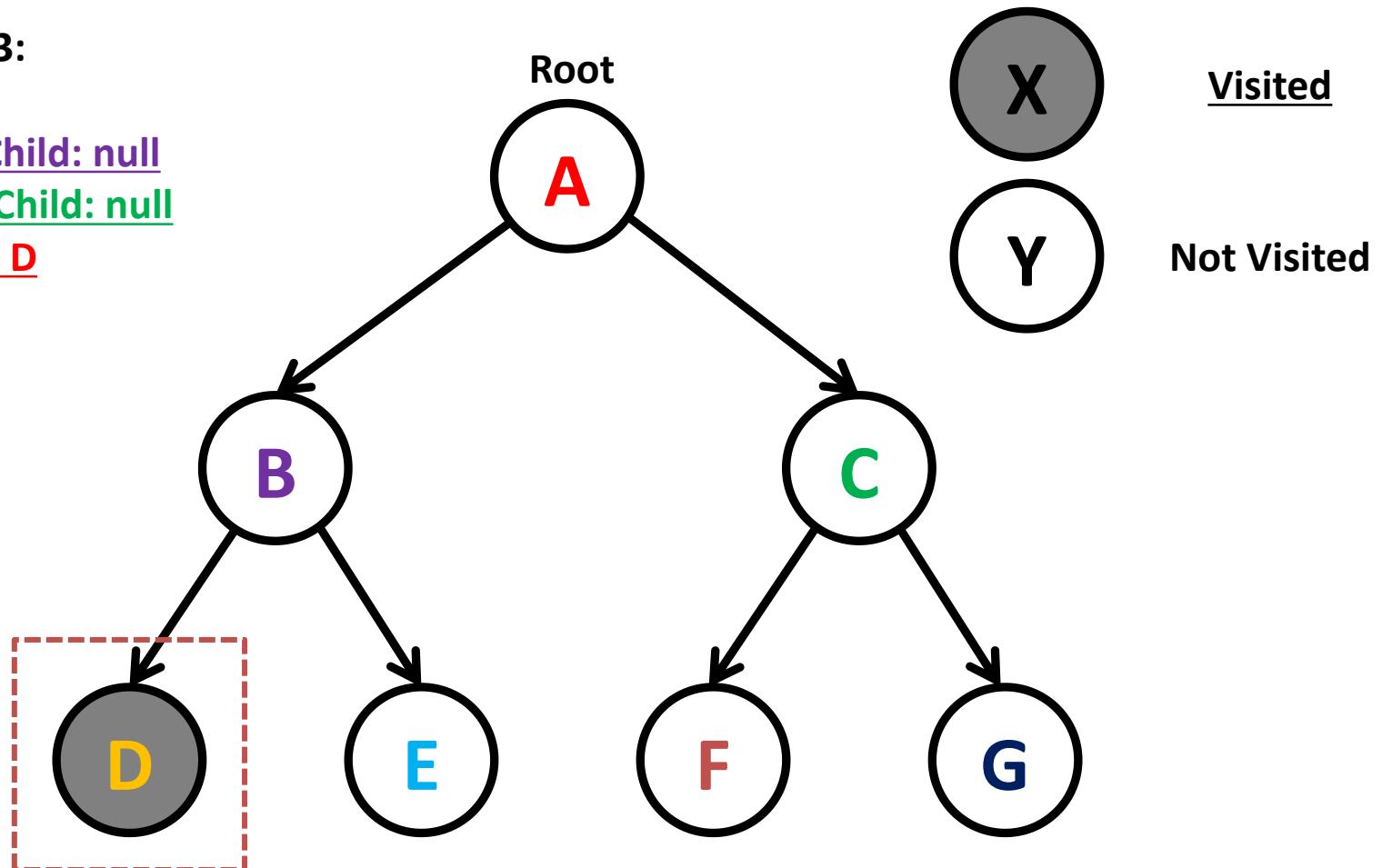


Nodes traversed so far:

Post-order (Left, Right, Node)

Step 3:

Node's Left Child: null
Node's Right Child: null
Node: D



Nodes traversed so far: D

Post-order (Left, Right, Node)

Step 4:

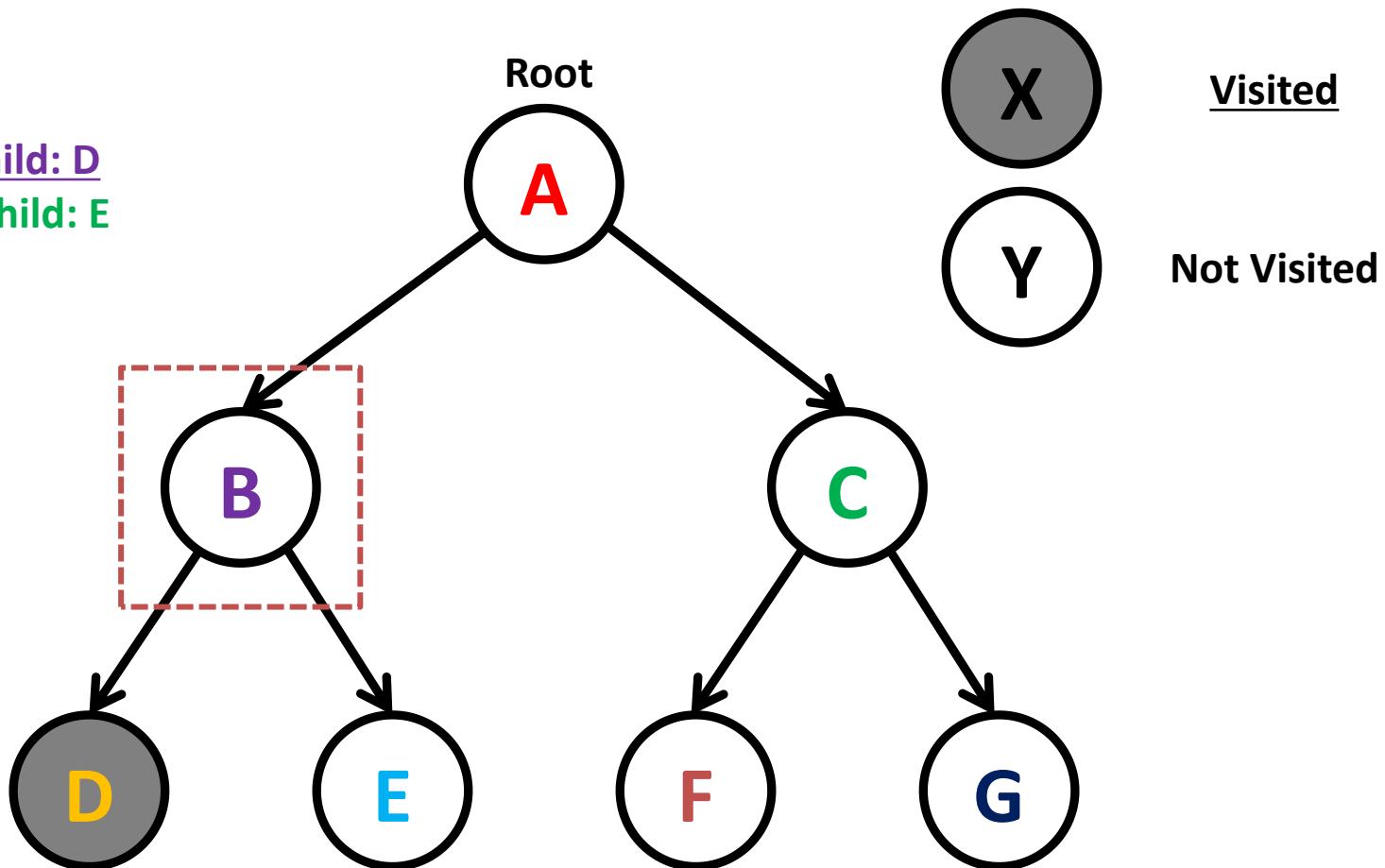
Node's Left Child: D

Node's Right Child: E

Node: B

Visited

Not Visited

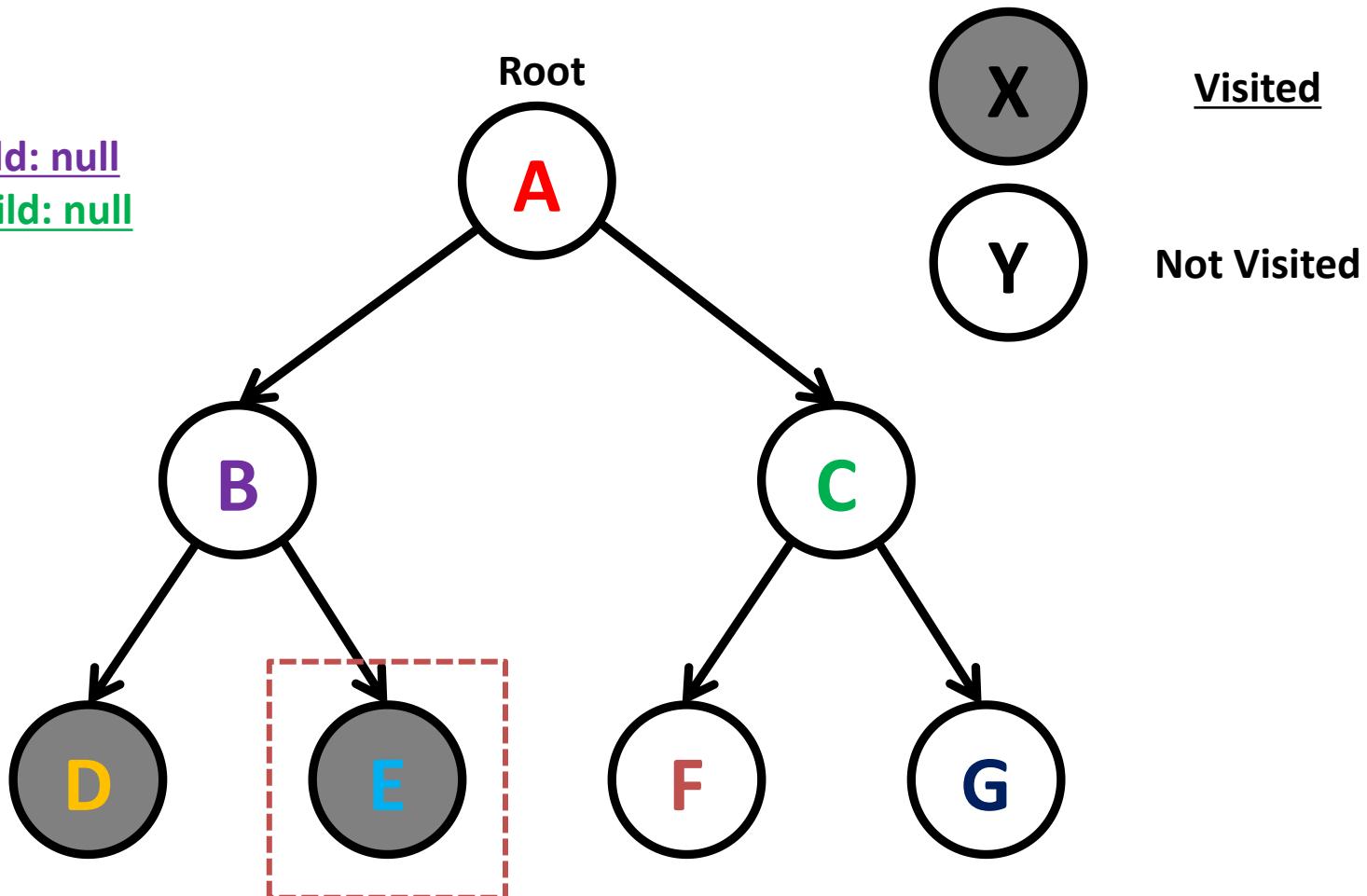


Nodes traversed so far: D

Post-order (Left, Right, Node)

Step 5:

Node's Left Child: null
Node's Right Child: null
Node: E



Nodes traversed so far: D E

Post-order (Left, Right, Node)

Step 6:

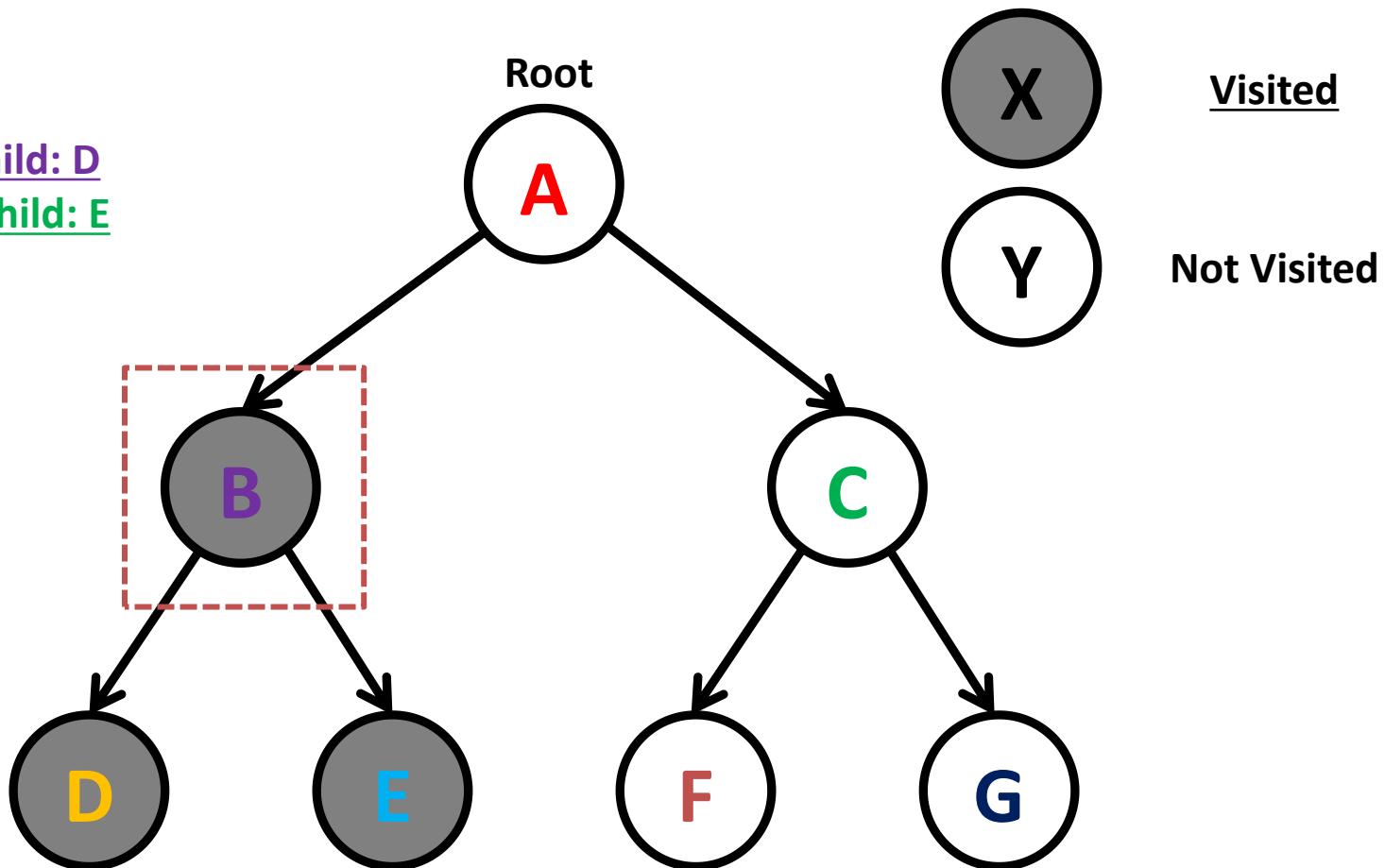
Node's Left Child: D

Node's Right Child: E

Node: B

Visited

Not Visited



Nodes traversed so far: D E B

Post-order (Left, Right, Node)

Step 7:

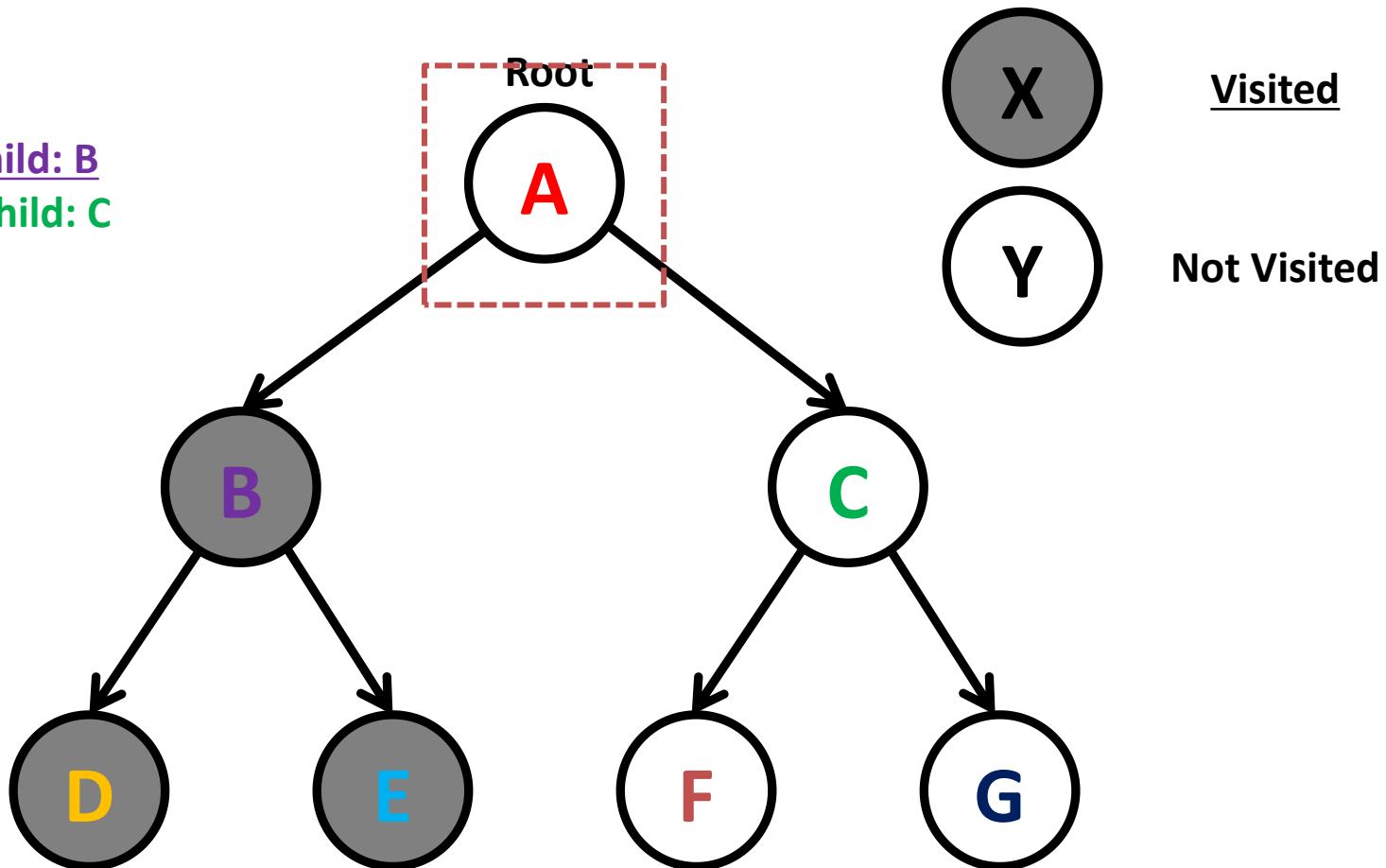
Node's Left Child: B

Node's Right Child: C

Node: A

Visited

Not Visited



Nodes traversed so far: D E B

Post-order (Left, Right, Node)

Step 8:

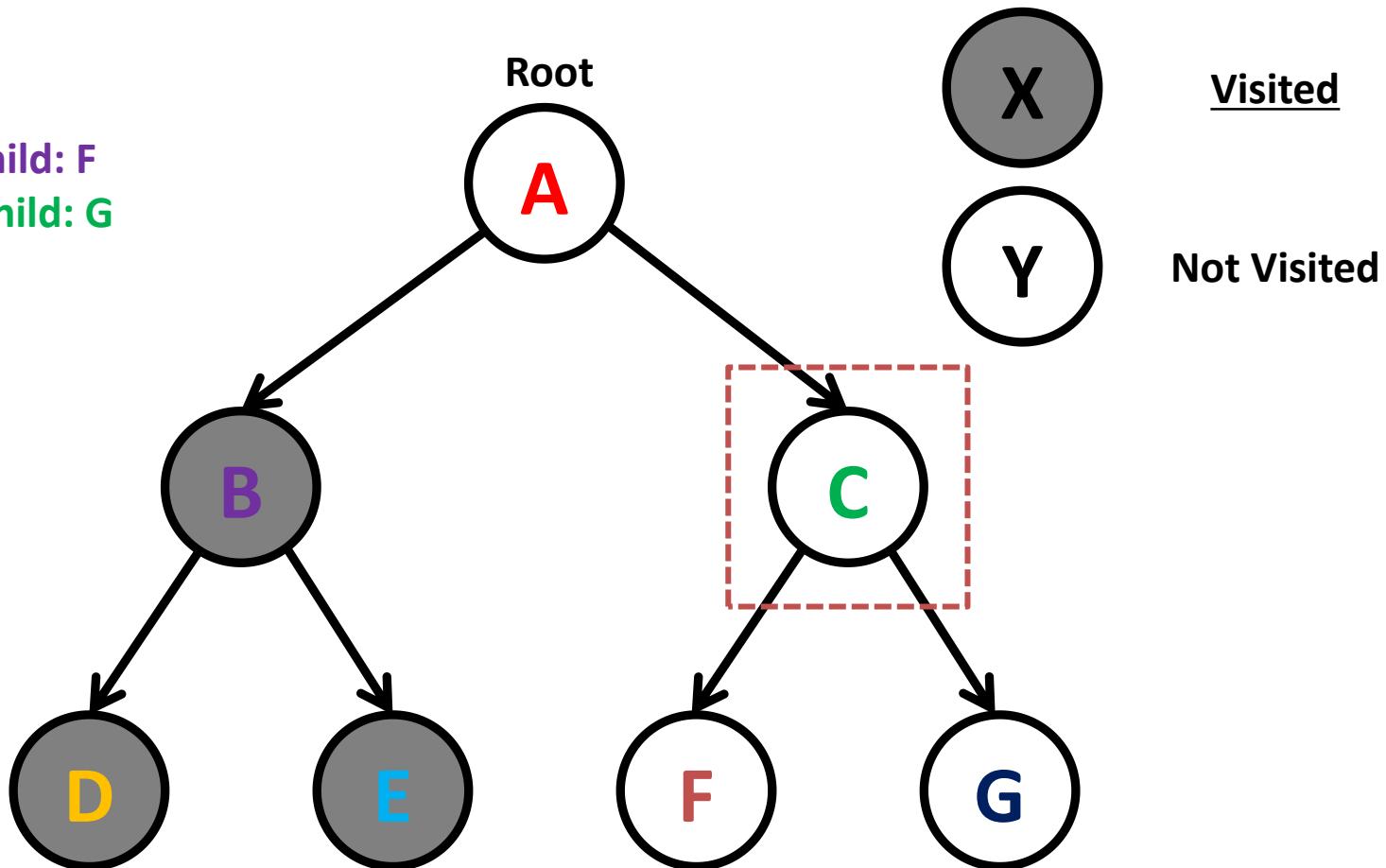
Node's Left Child: F

Node's Right Child: G

Node: C

Visited

Not Visited



Nodes traversed so far: D E B

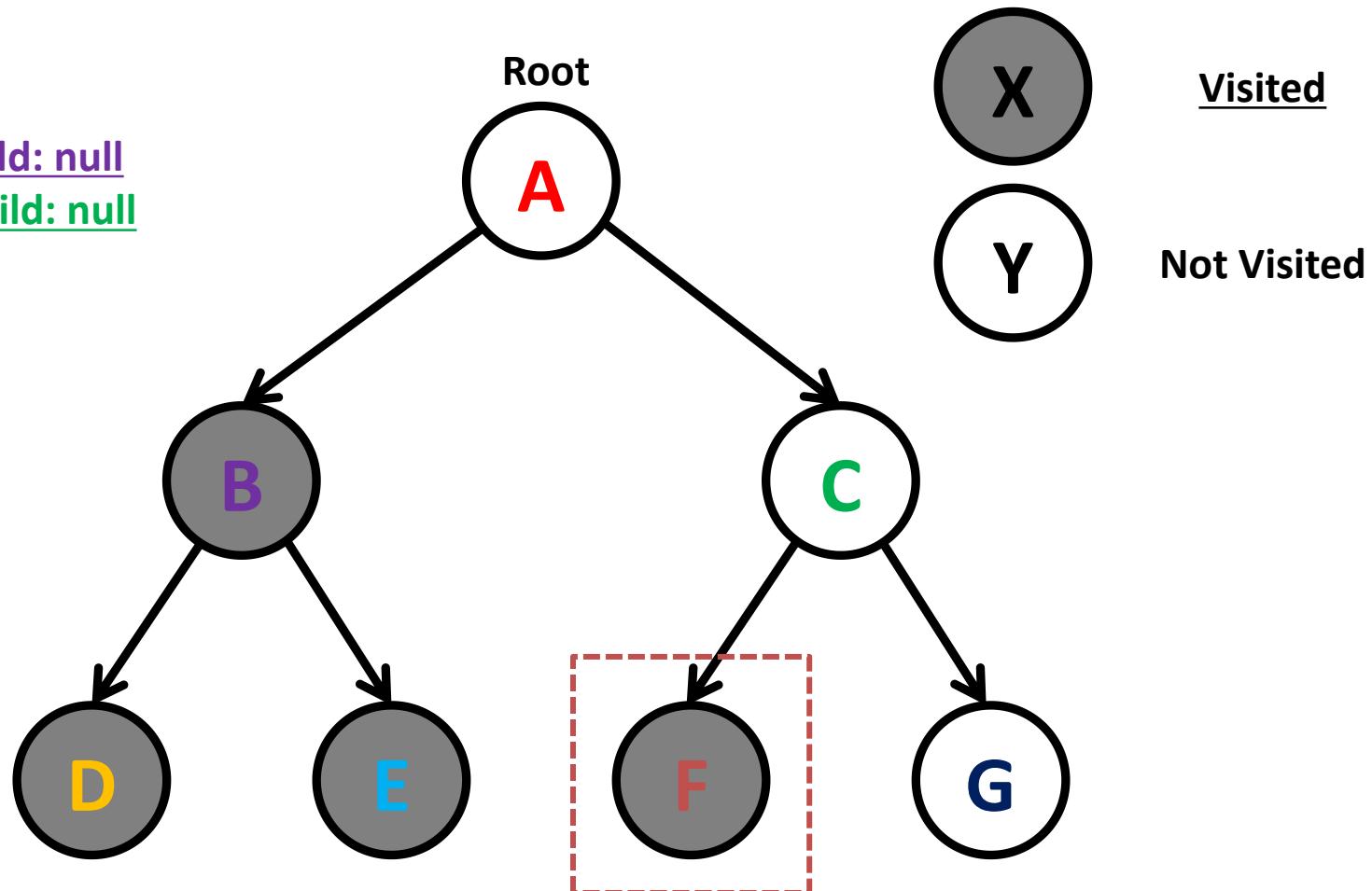
Post-order (Left, Right, Node)

Step 9:

Node's Left Child: null

Node's Right Child: null

Node: F



Nodes traversed so far: D E B F

Post-order (Left, Right, Node)

Step 10:

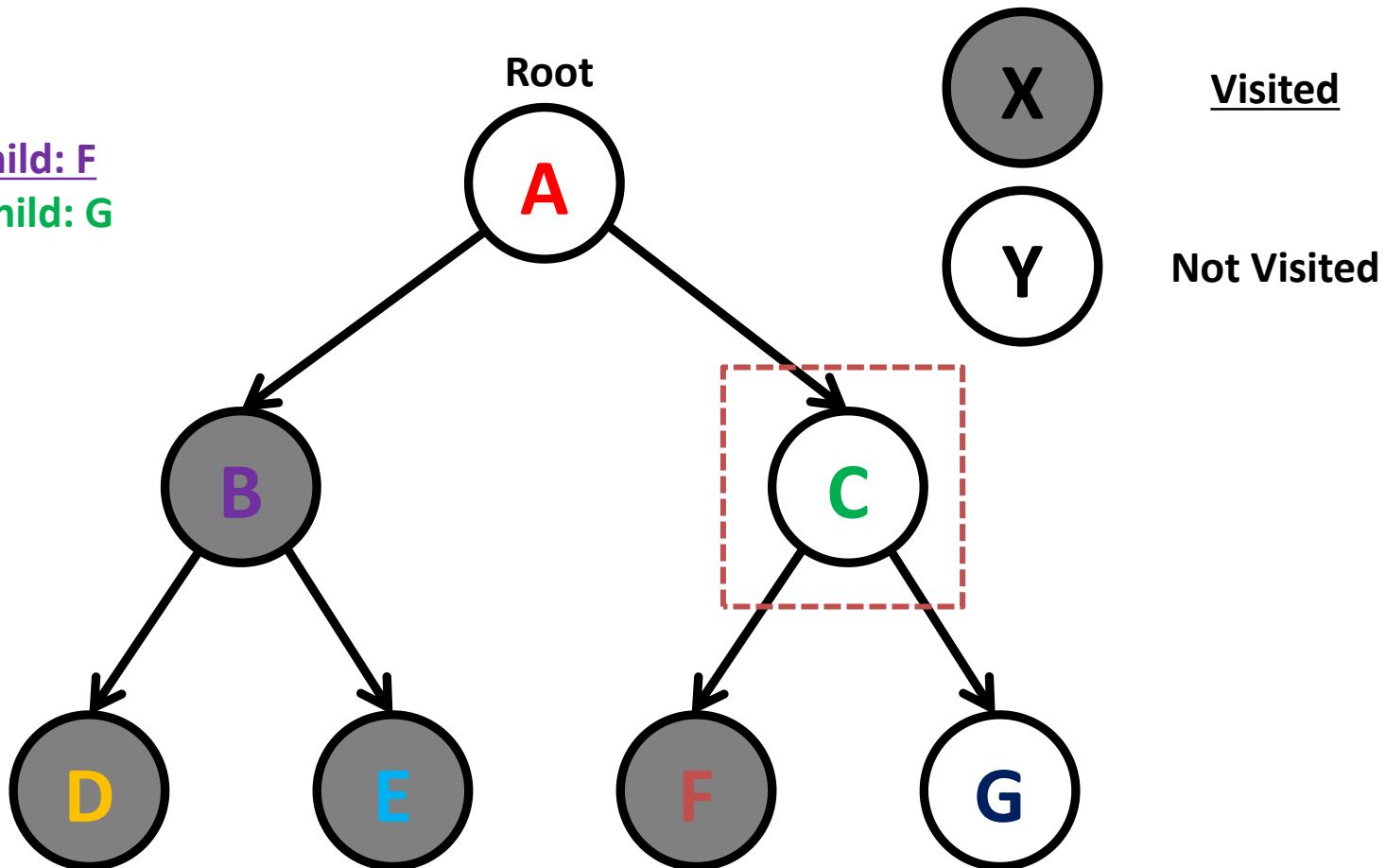
Node's Left Child: F

Node's Right Child: G

Node: C

Visited

Not Visited



Nodes traversed so far: D E B F

Post-order (Left, Right, Node)

Step 11:

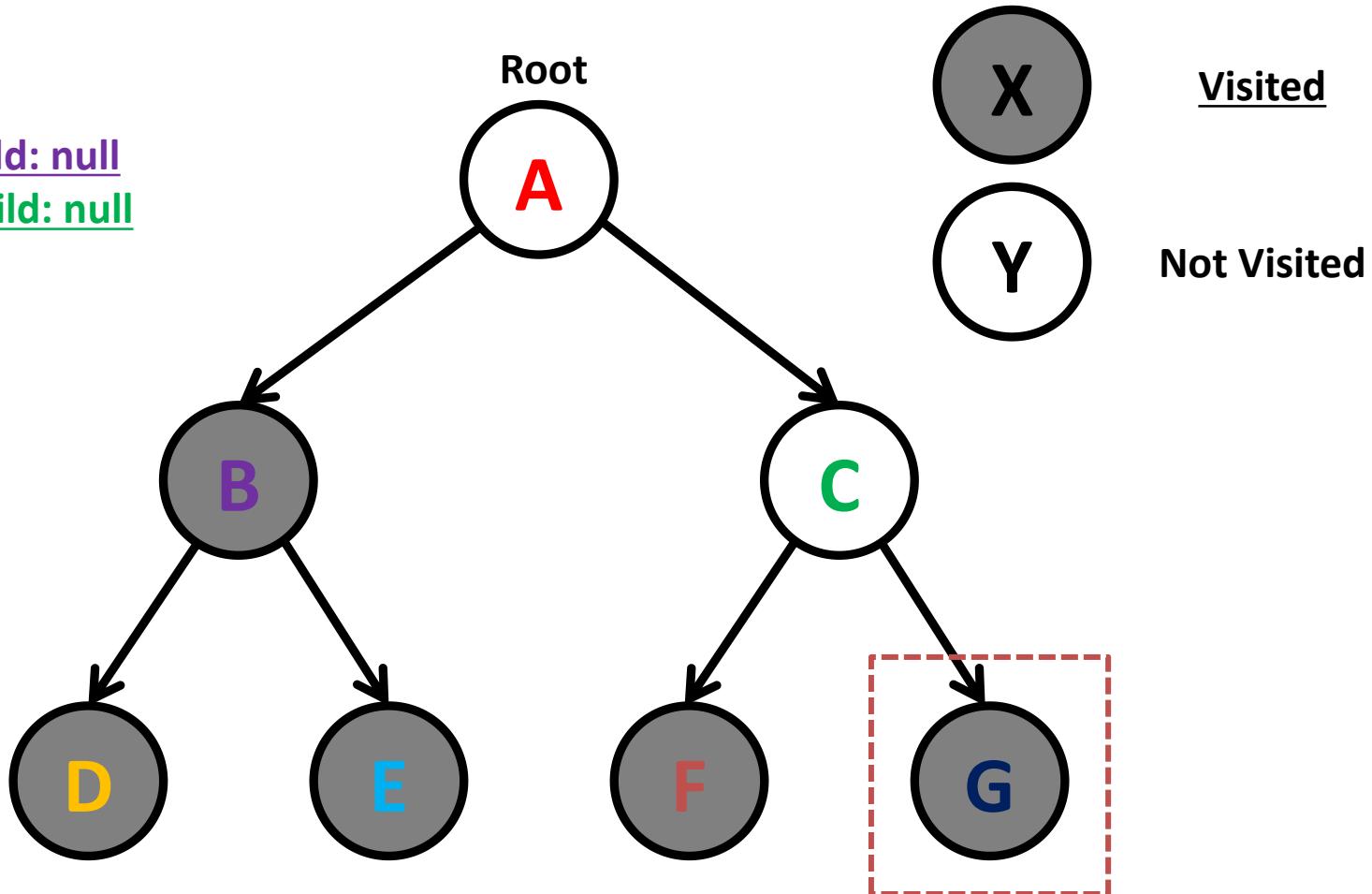
Node's Left Child: null

Node's Right Child: null

Node: G

Visited

Not Visited



Nodes traversed so far: D E B F G

Post-order (Left, Right, Node)

Step 12:

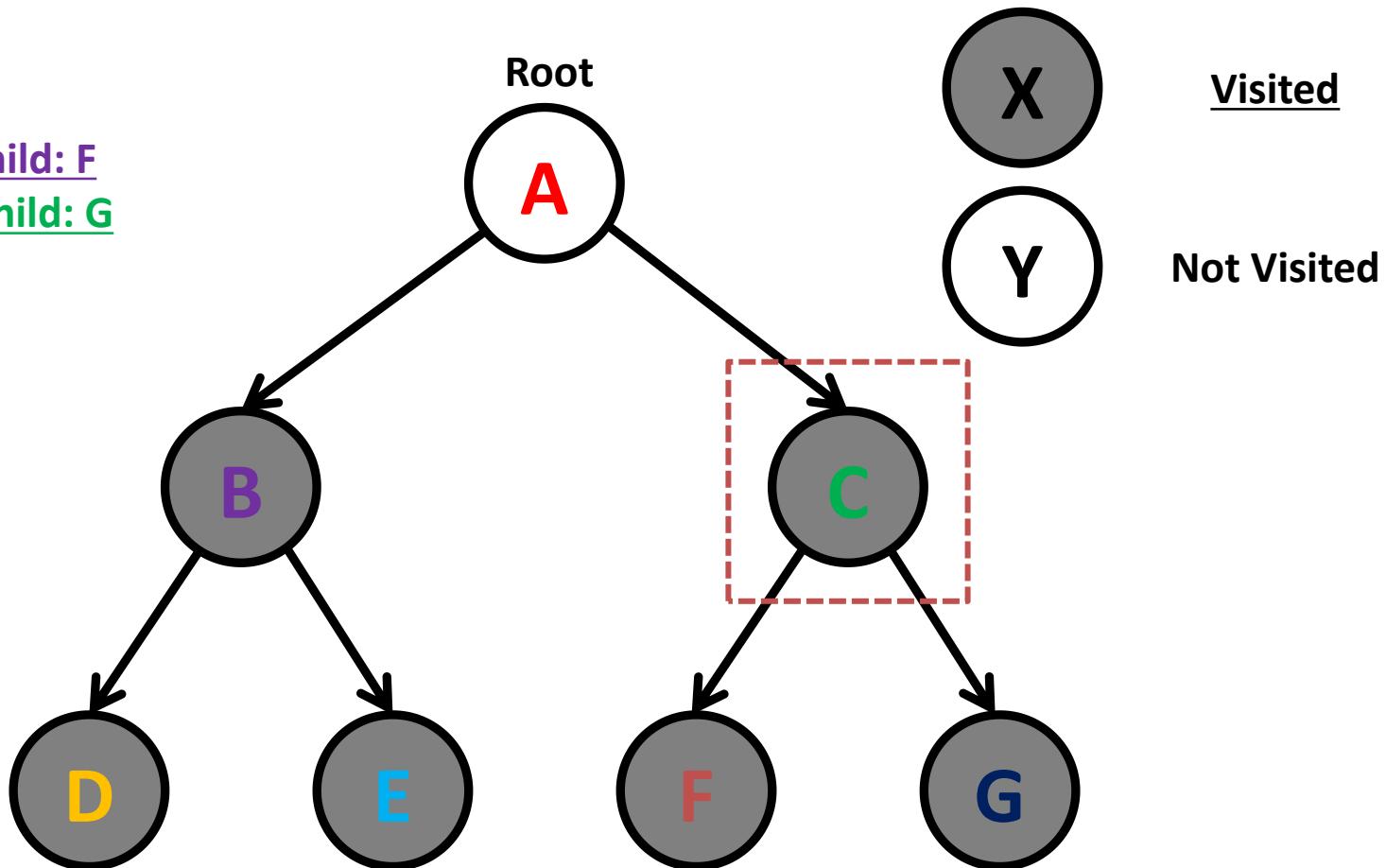
Node's Left Child: F

Node's Right Child: G

Node: C

Visited

Not Visited



Nodes traversed so far: D E B F G C

Post-order (Left, Right, Node)

Step 13:

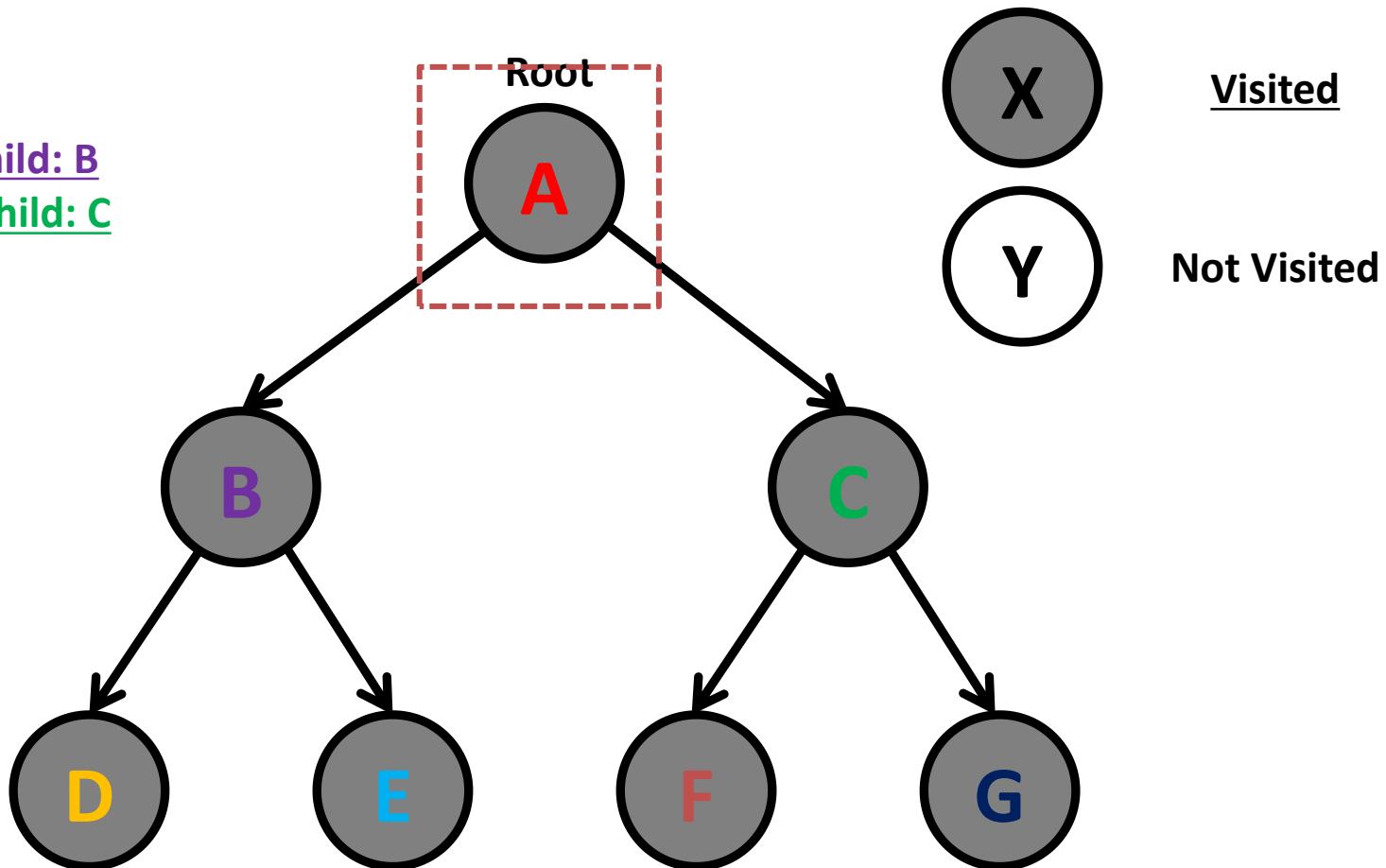
Node's Left Child: B

Node's Right Child: C

Node: A

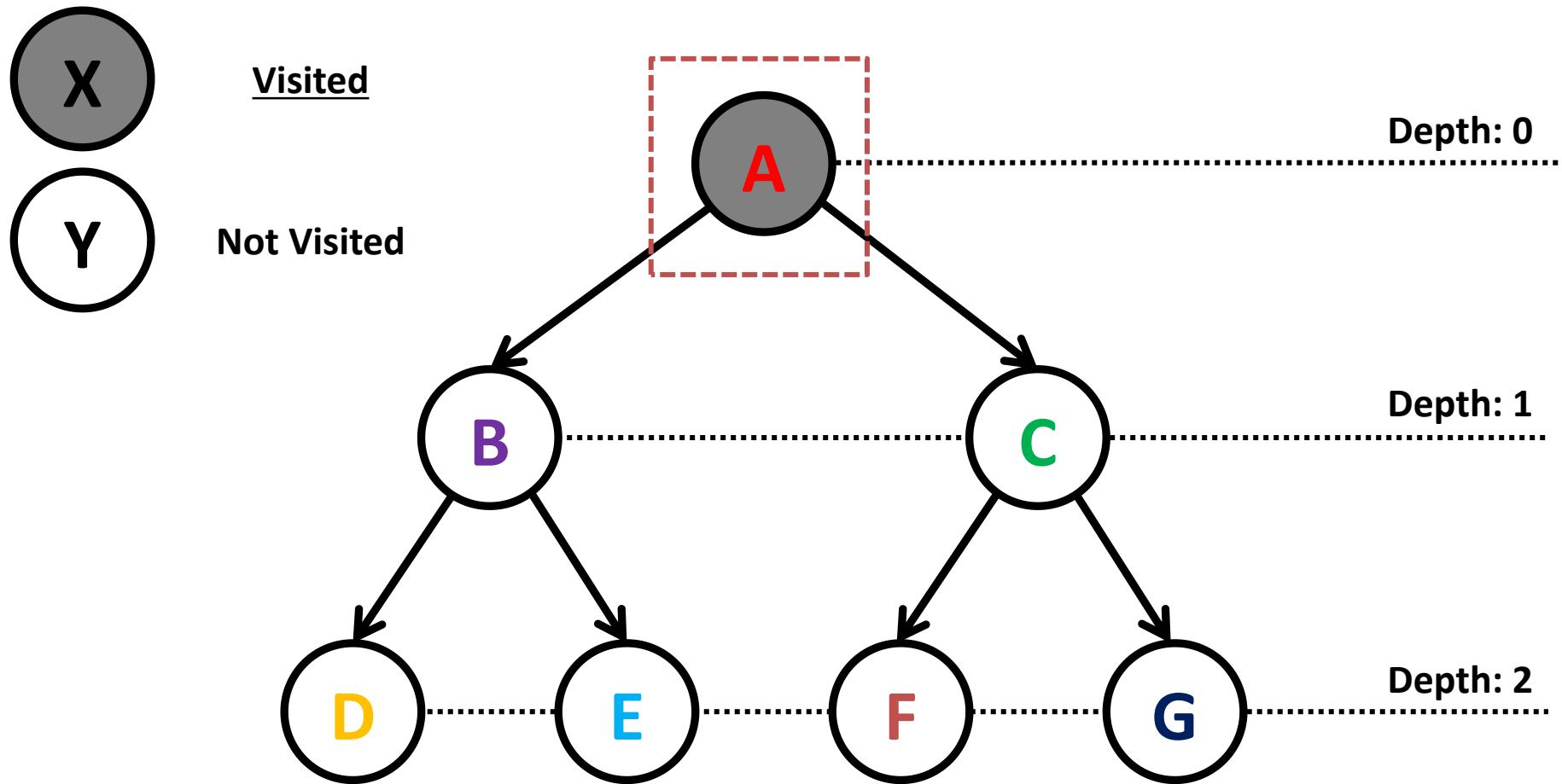
Visited

Not Visited



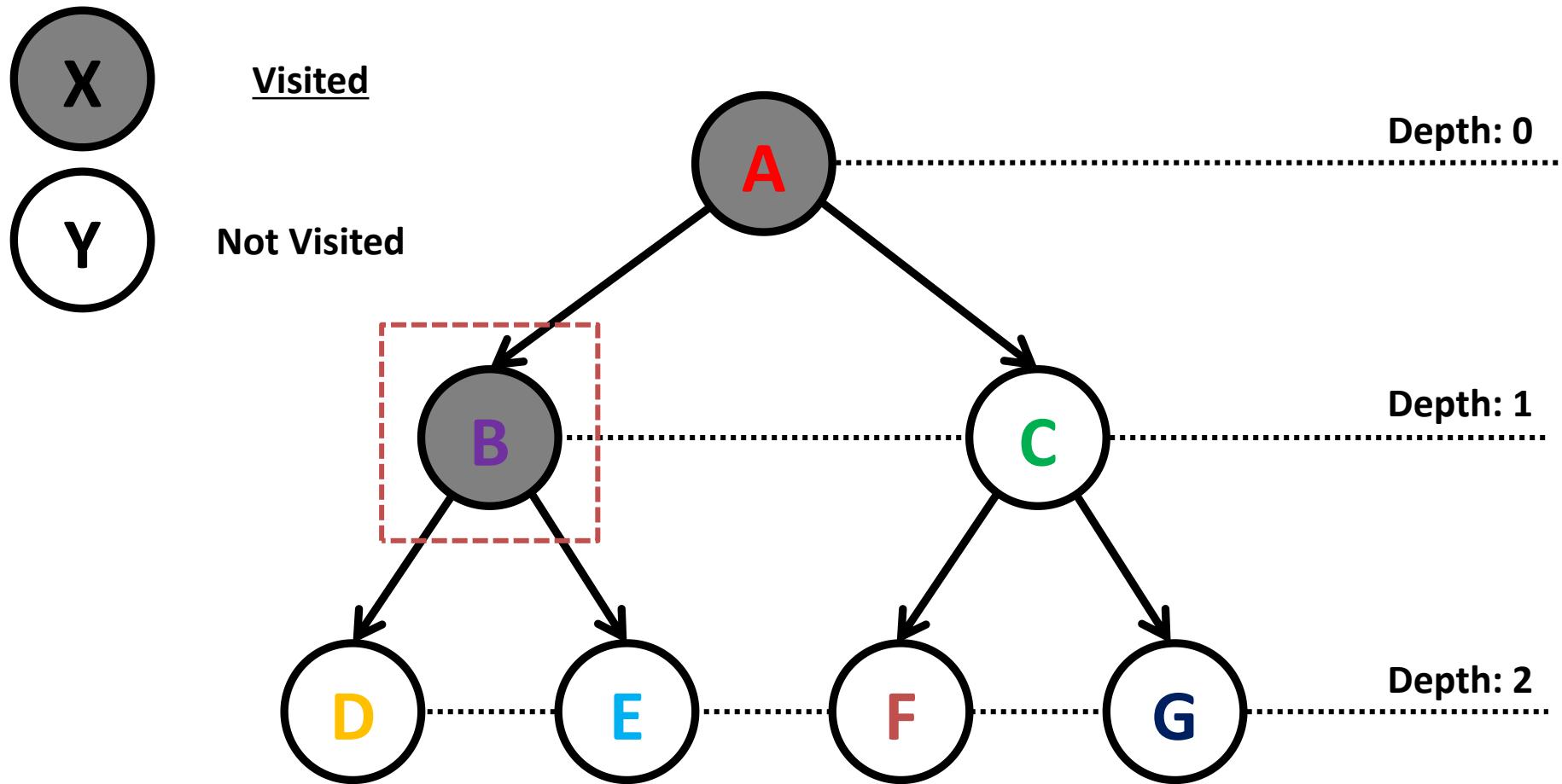
Nodes traversed so far: D E B F G C A

Level-order (Level-first | Left-Right)



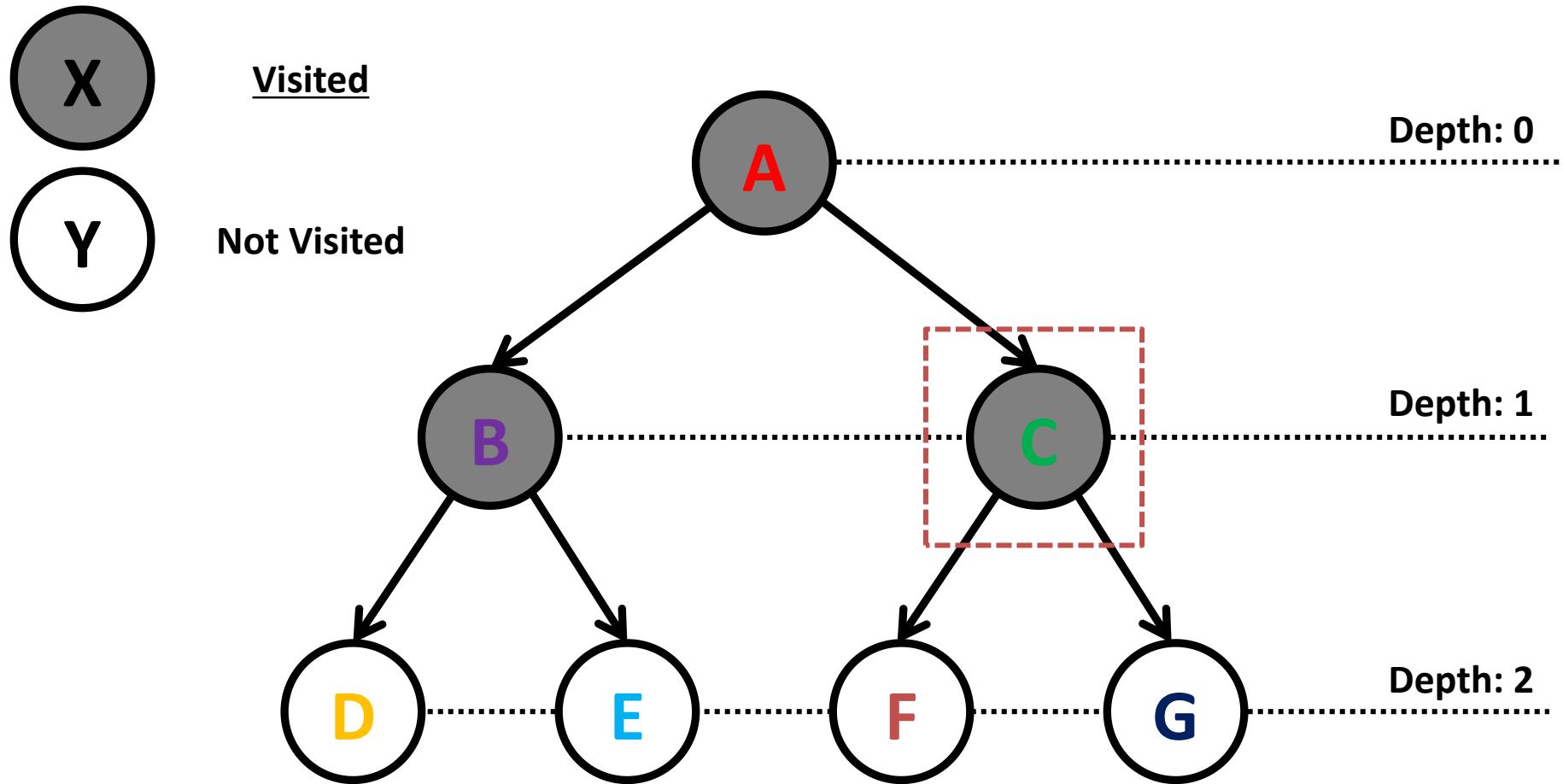
Nodes traversed so far: **A**

Level-order (Level-first | Left-Right)



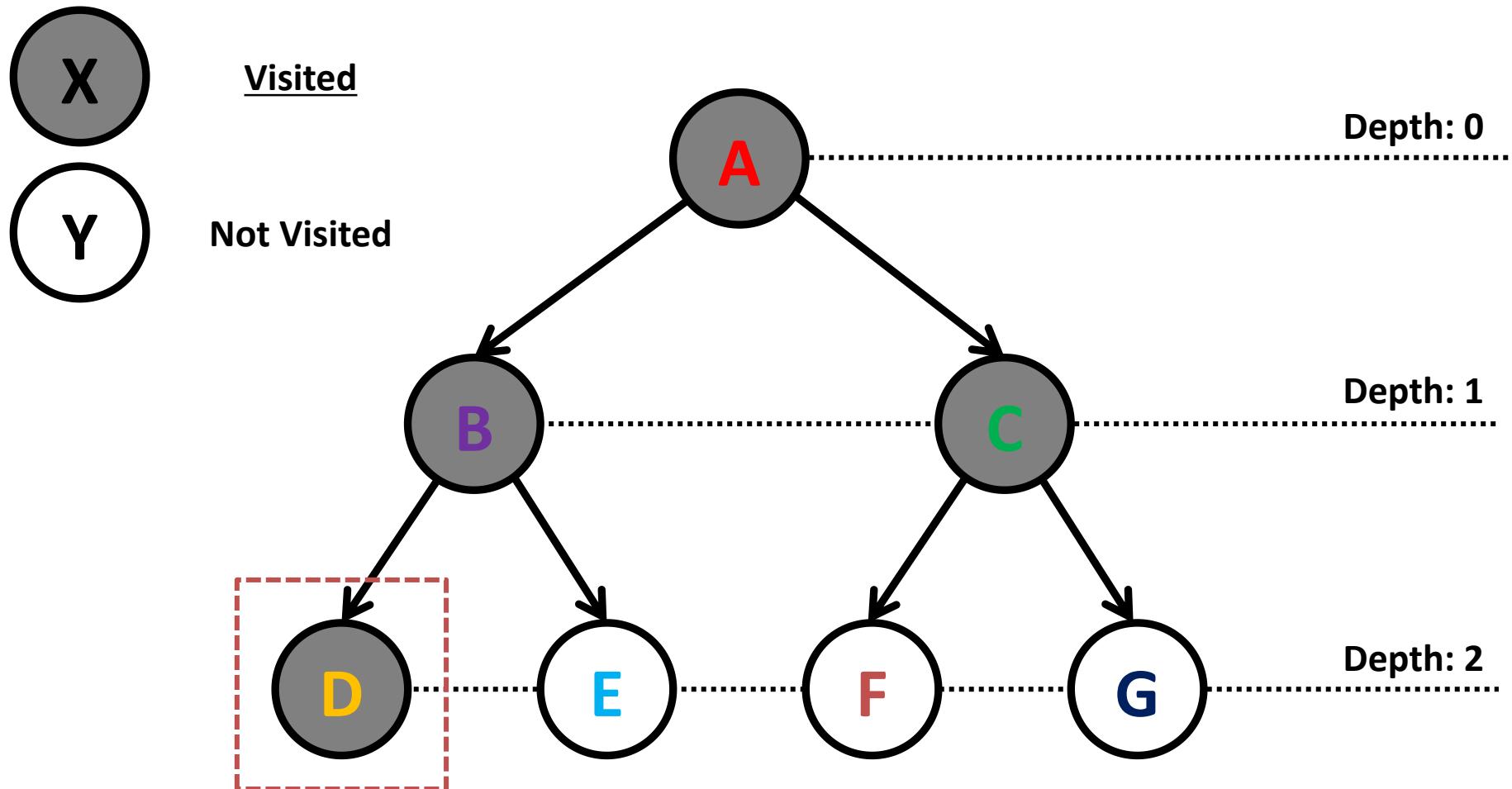
Nodes traversed so far: A B

Level-order (Level-first | Left-Right)



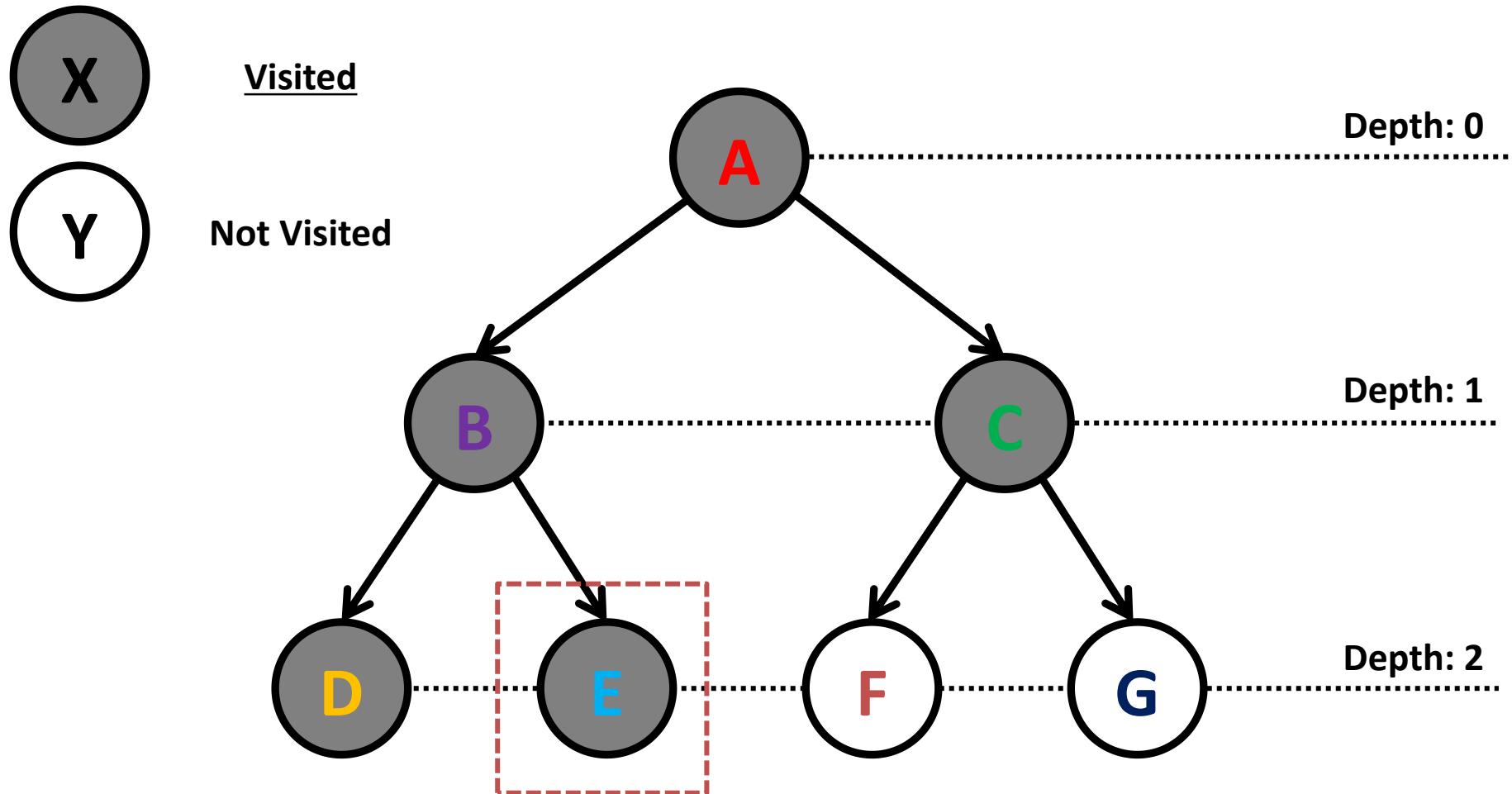
Nodes traversed so far: A B C

Level-order (Level-first | Left-Right)



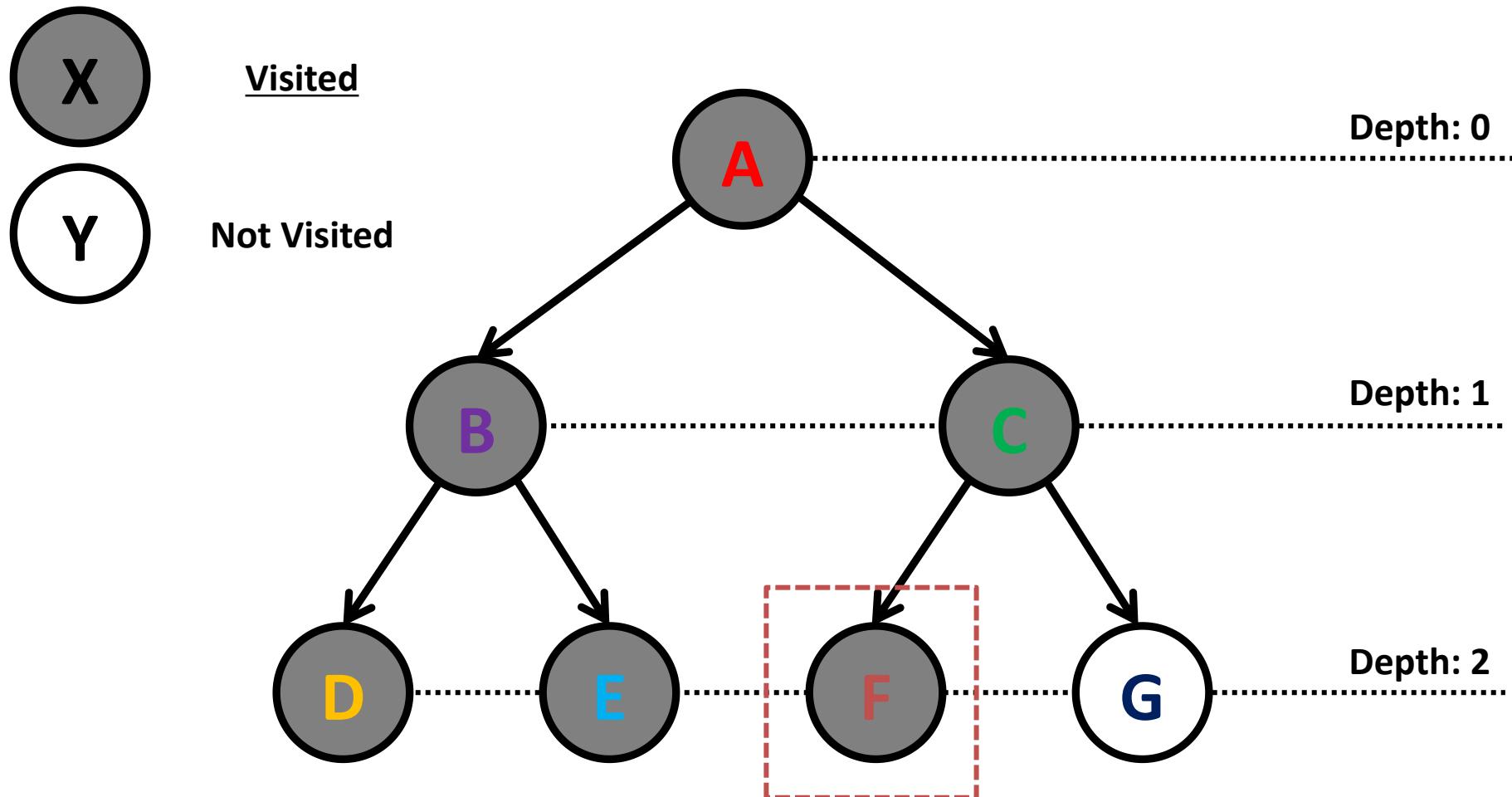
Nodes traversed so far: A B C D

Level-order (Level-first | Left-Right)

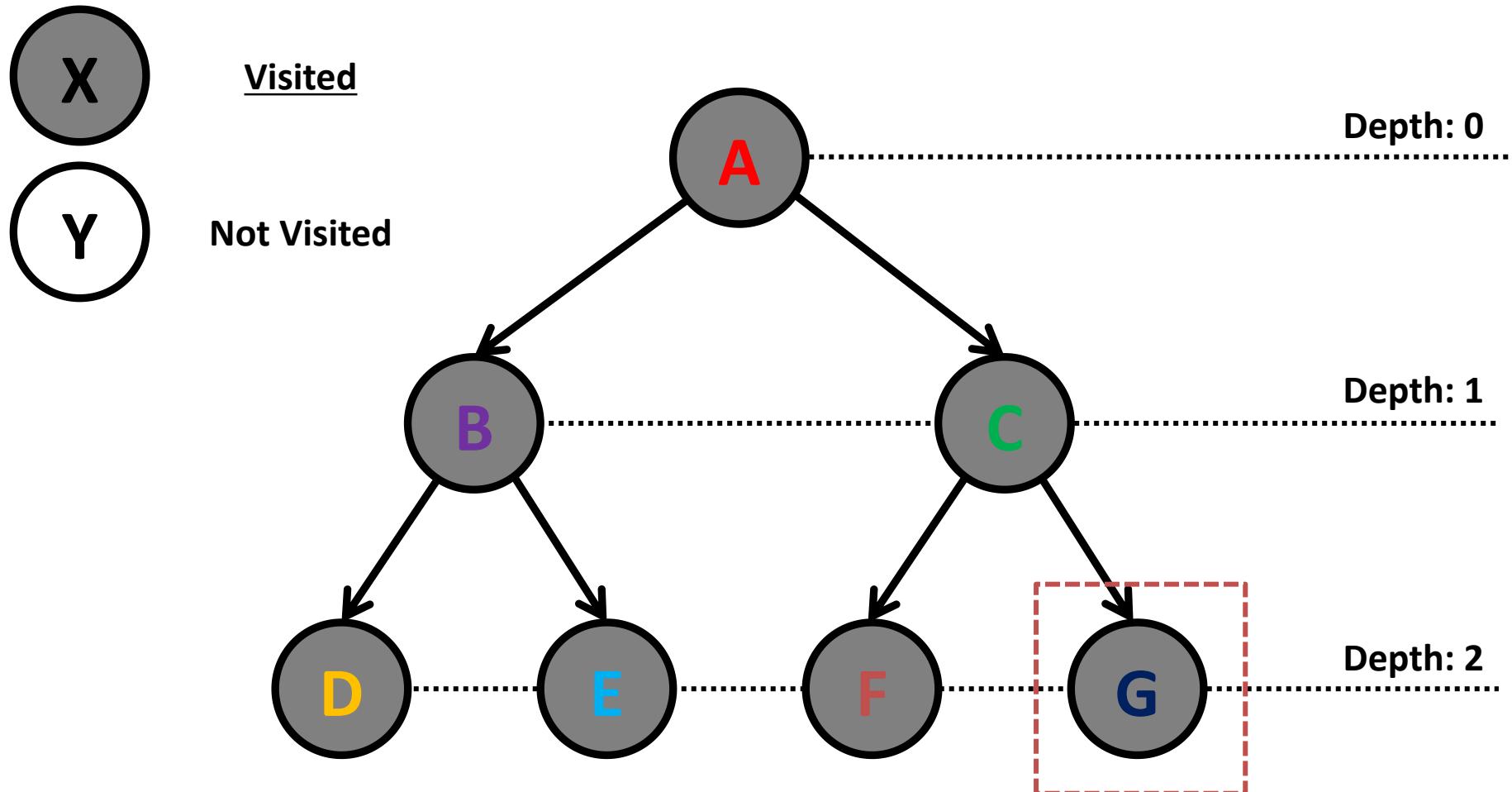


Nodes traversed so far: A B C D E

Level-order (Level-first | Left-Right)



Level-order (Level-first | Left-Right)



Nodes traversed so far: A B C D E F G

Depth First vs Breadth (Width) First

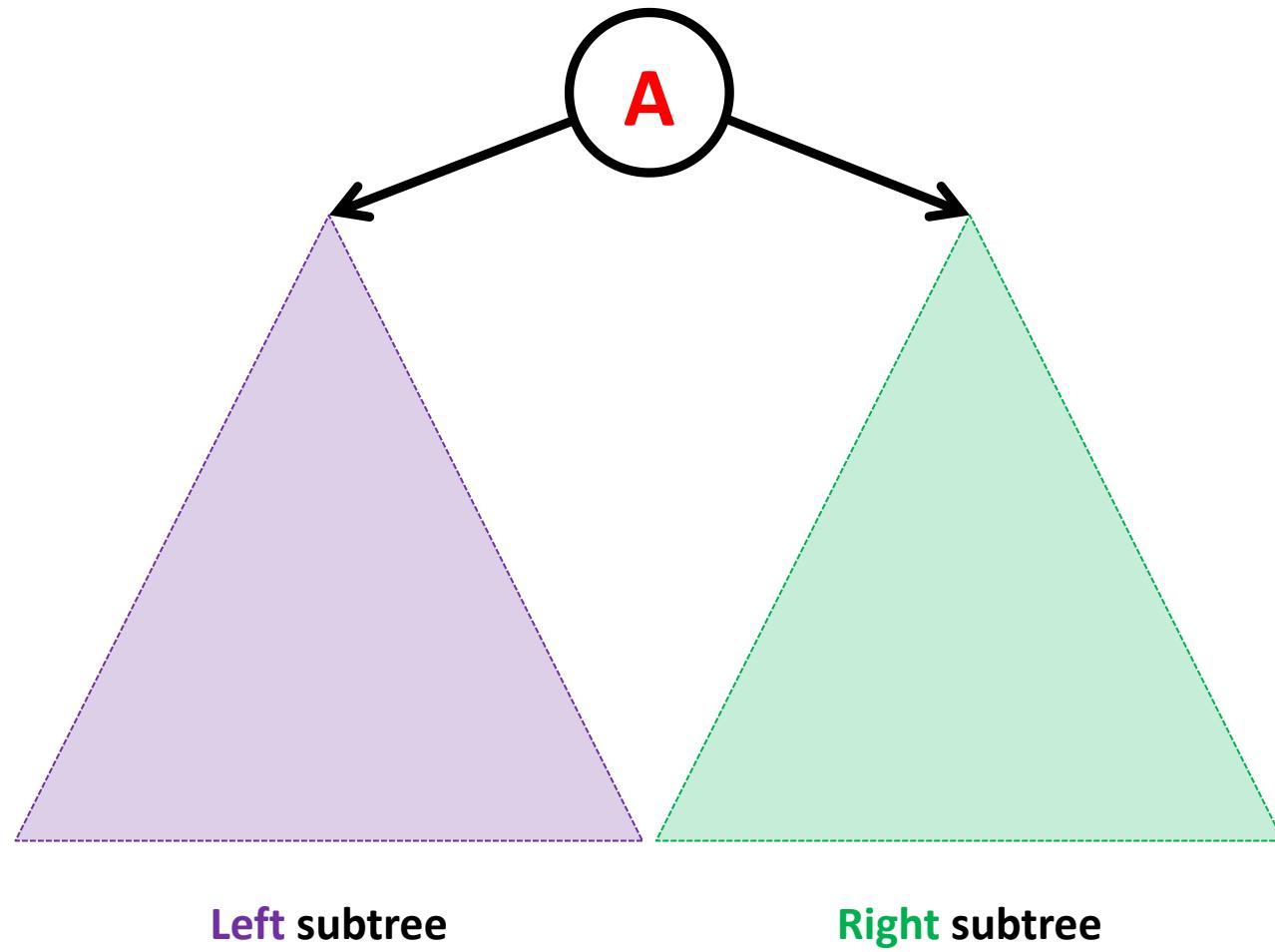
Depth First Traversals:

- Pre-order Traversal (**Node**, **Left Child**, **Right Child**)
- In-order Traversal (**Left Child**, **Node**, **Right Child**)
- Post-order Traversal (**Left Child**, **Right Child**, **Node**)

Breadth (Width) First

- Level-order (Level-by-level | left-to-right)

Binary Tree: Recursive Structure



Traversals: Algorithms

Pre-order Traversal (recursive)

```
void preOrder(Node node) {  
    if (node != null) {  
        System.out.print(node.getKey());  
        preOrder(node.getLeftChild());  
        preOrder(node.getRightChild());  
    } else {  
        return;  
    }  
}
```

In-Order Traversal (recursive)

```
void inOrder(Node node) {  
    if (node != null) {  
        inOrder(node.getLeftChild());  
        System.out.print(node.getKey());  
        inOrder(node.getRightChild());  
    } else {  
        return;  
    }  
}
```

Post-order Traversal (recursive)

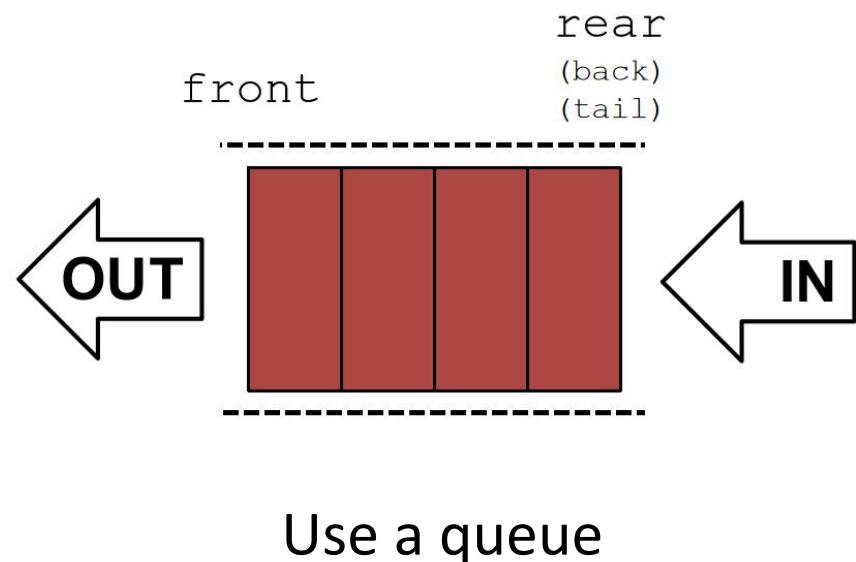
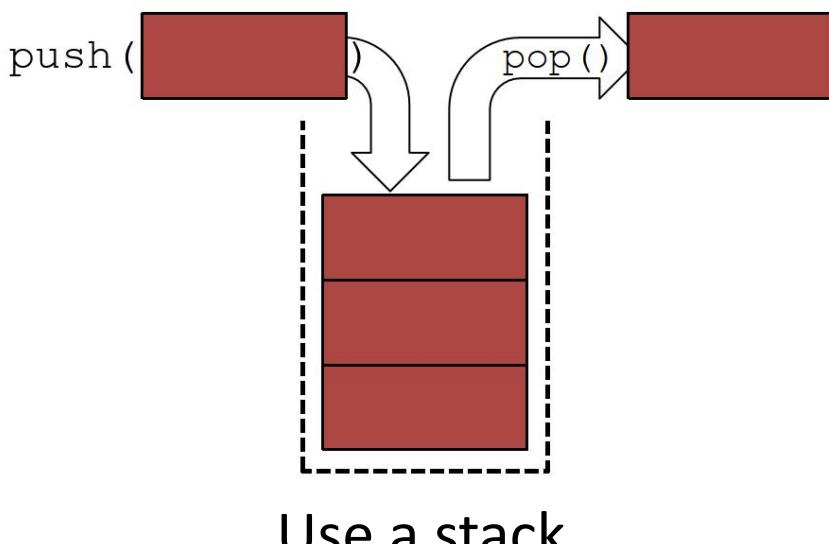
```
void postOrder(Node node) {  
    if (node != null) {  
        postOrder(node.getLeftChild());  
        postOrder(node.getRightChild());  
        System.out.print(node.getKey());  
    } else {  
        return;  
    }  
}
```

Level-order Traversal

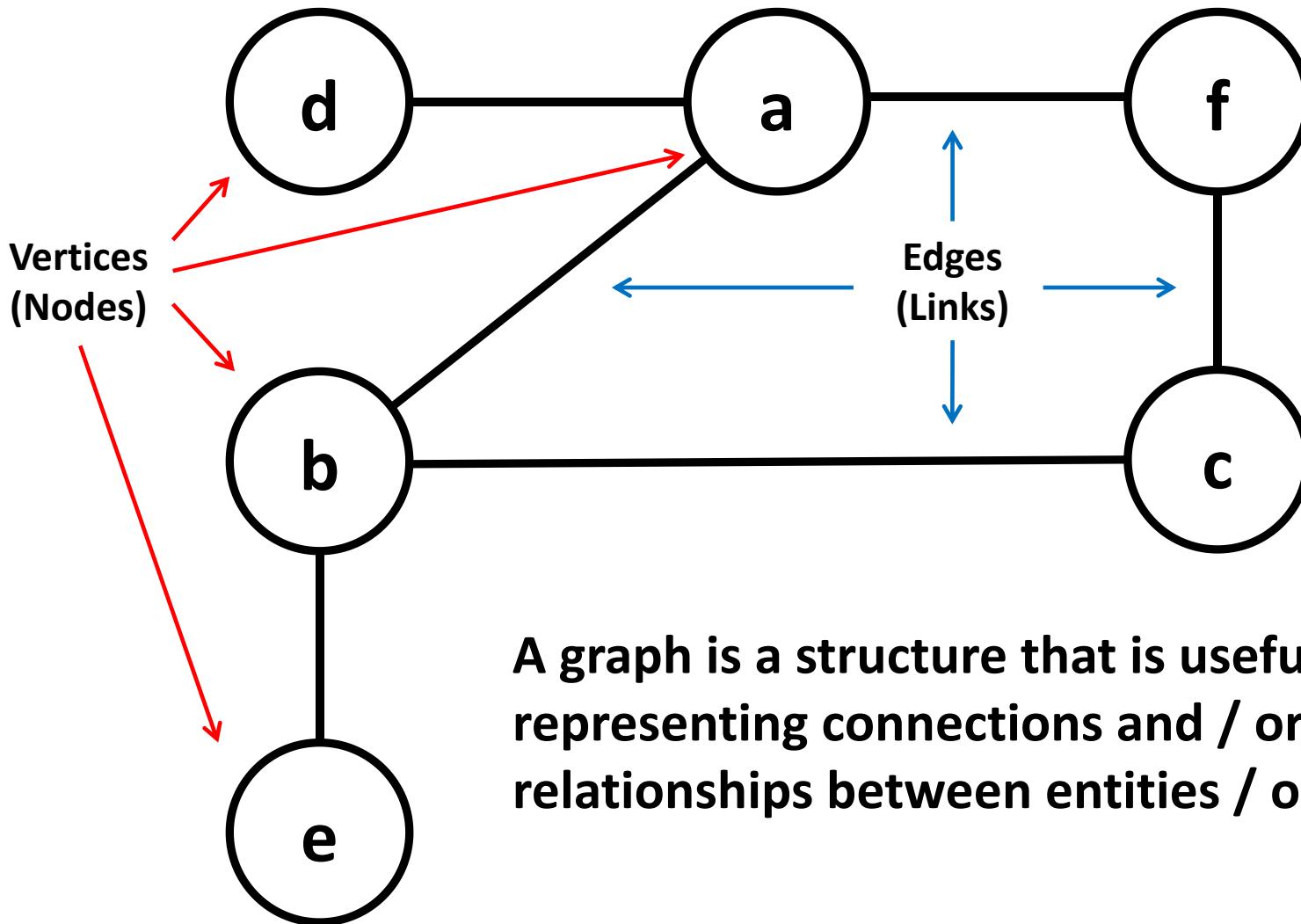
```
void levelOrder(Node node) {  
    if (node != null) {  
        queue.add(node);  
        while(queue.isEmpty() == null)  
            current = queue.dequeue();  
            System.out.println(current);  
            queue.enqueue(current.children)  
    }  
}
```

Traversals: Data Structure Use

- Pre-order Traversal
- In-order Traversal
- Post-order Traversal
- Level-order (Level-by-level | left-to-right)

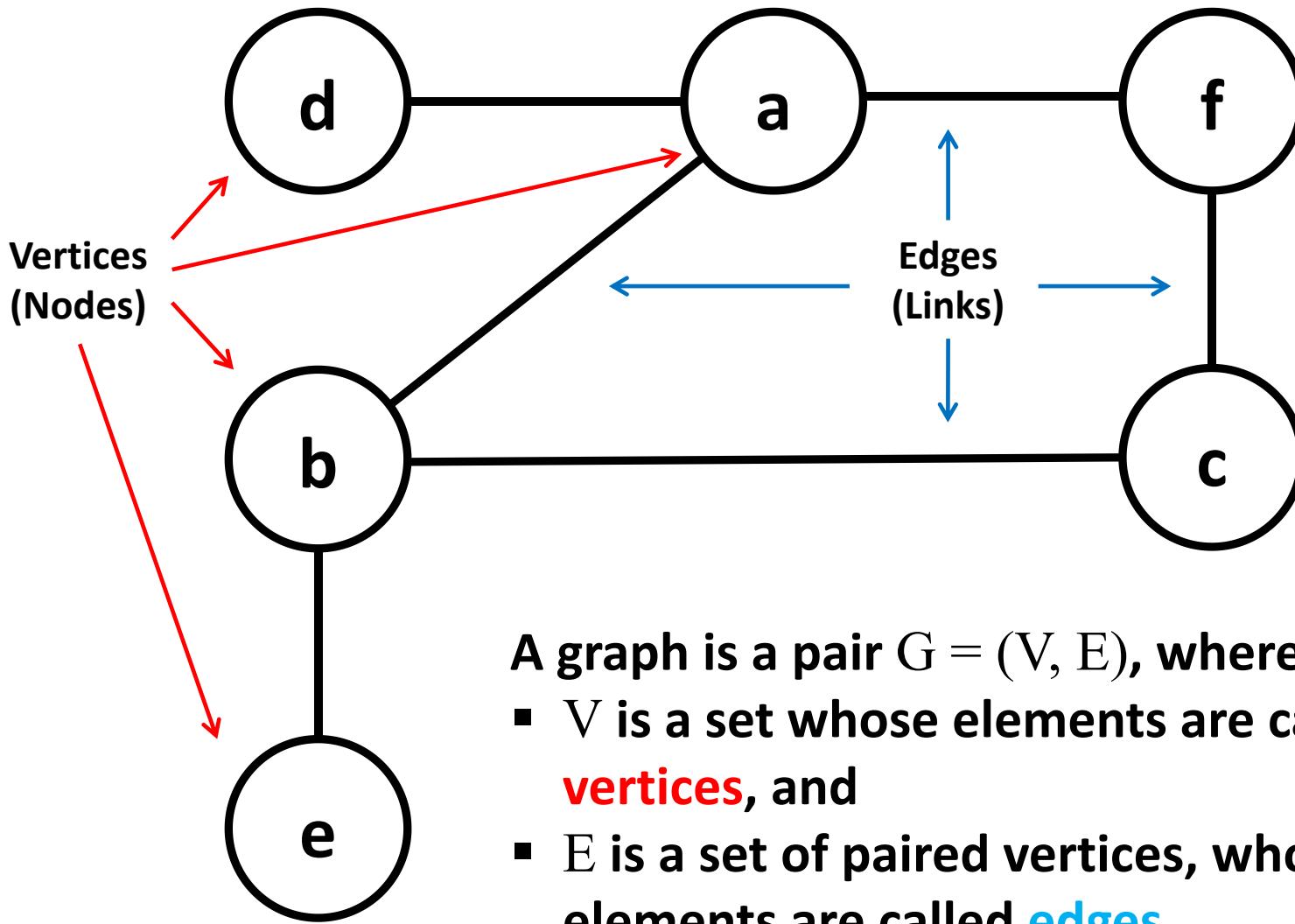


Graph = Vertices + Edges

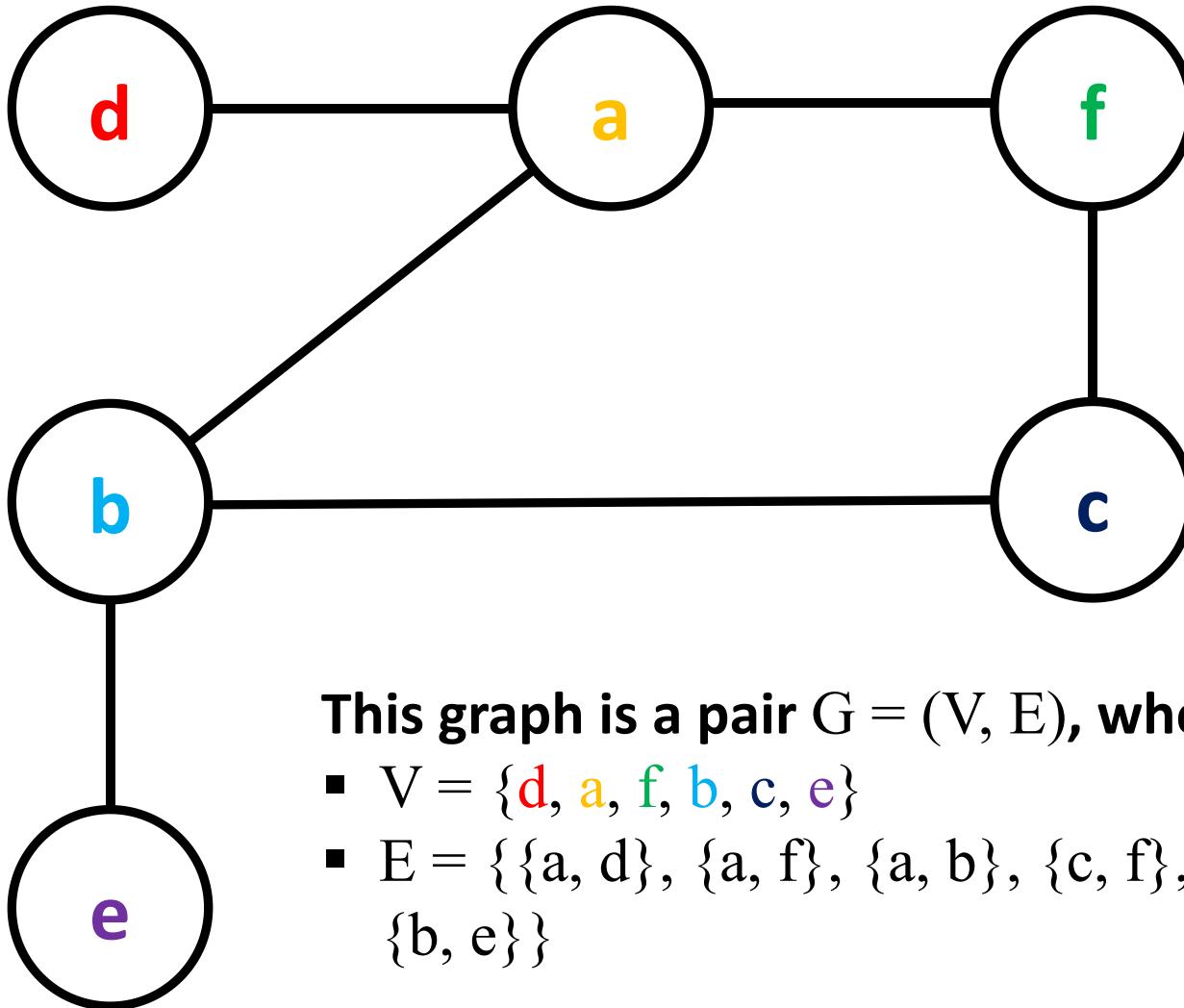


A graph is a structure that is useful in representing connections and / or relationships between entities / objects

Graph: $G = (V, E)$



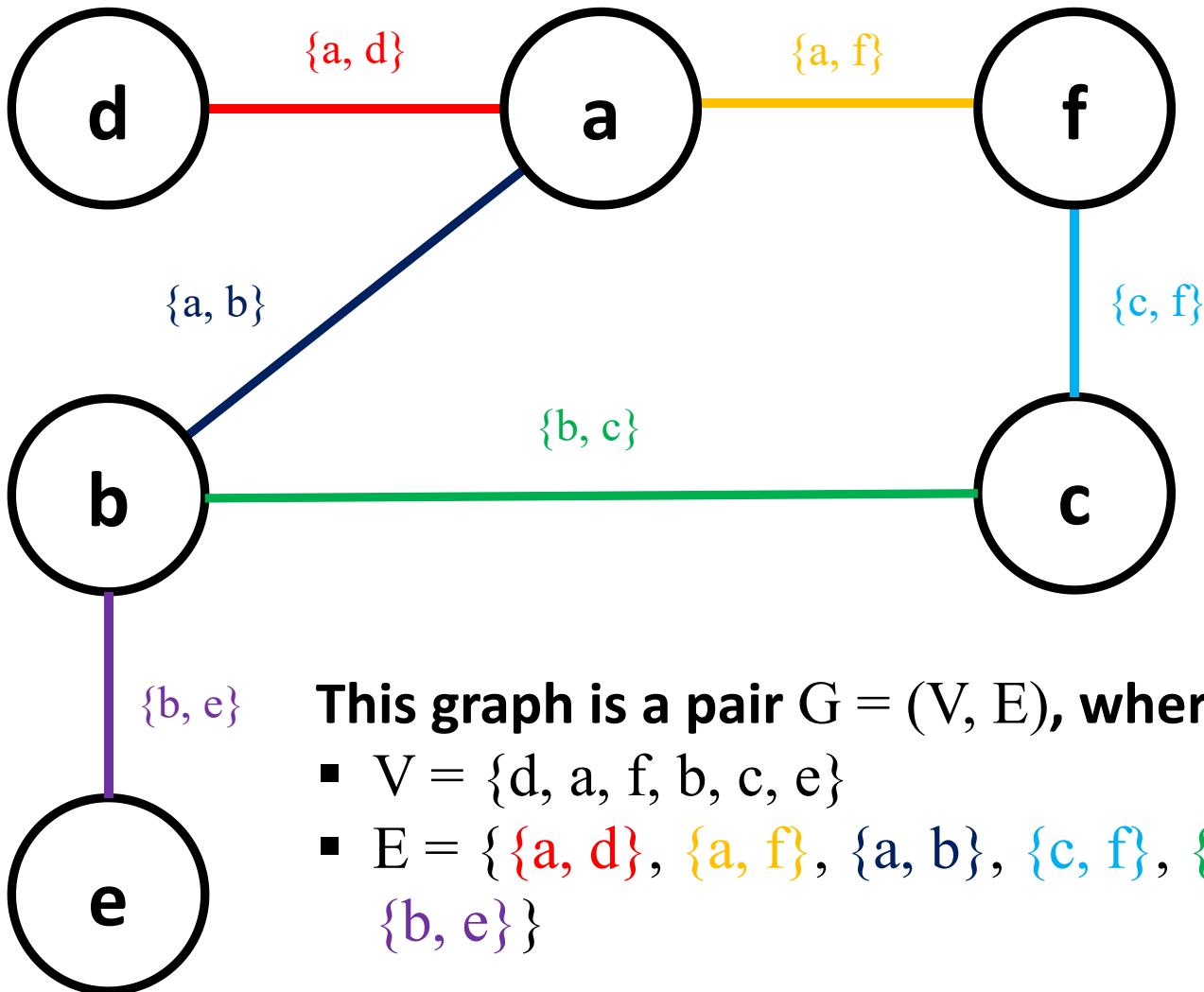
Graph: $G = (V, E)$



This graph is a pair $G = (V, E)$, where:

- $V = \{d, a, f, b, c, e\}$
- $E = \{\{a, d\}, \{a, f\}, \{a, b\}, \{c, f\}, \{b, c\}, \{b, e\}\}$

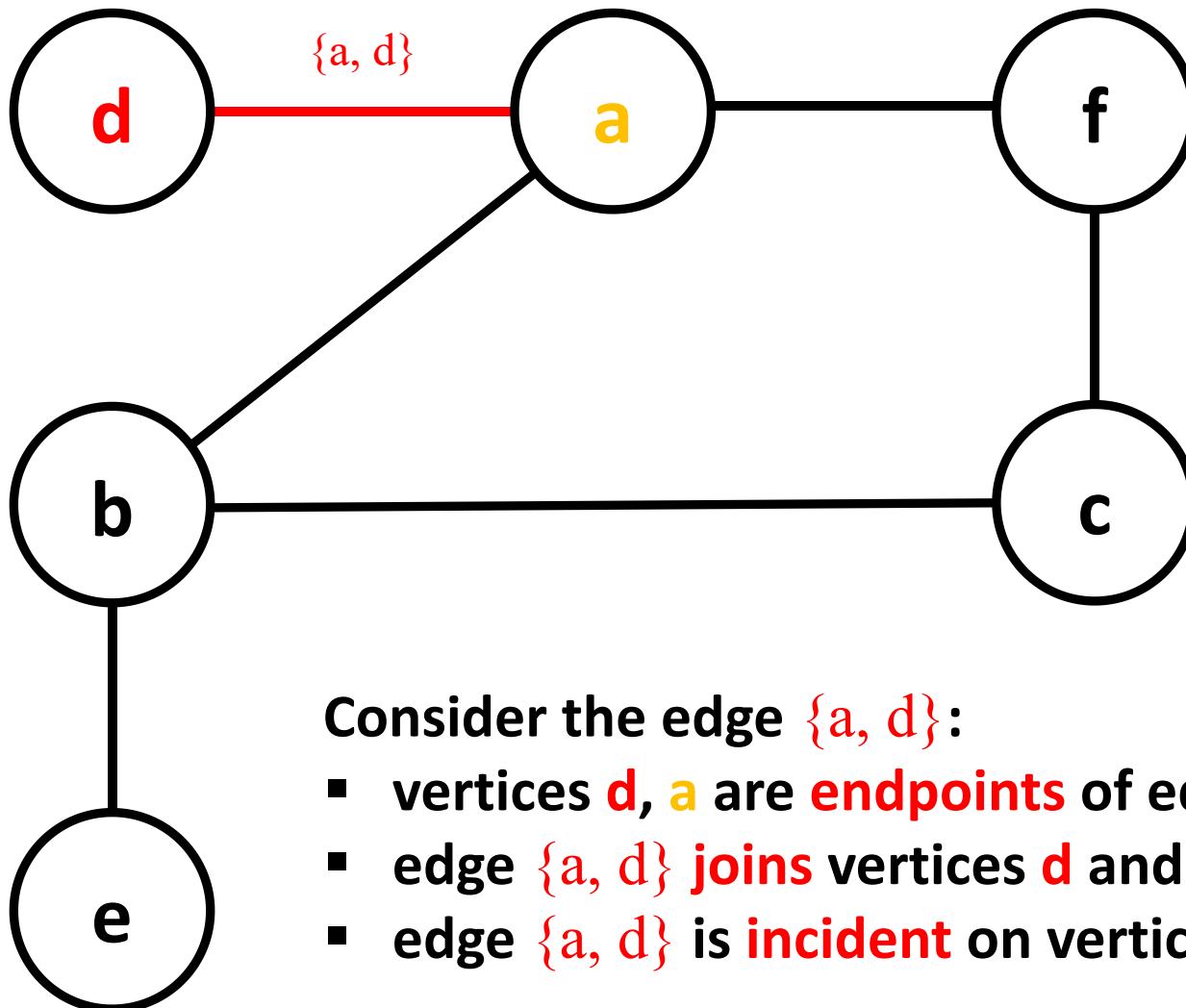
Graph: $G = (V, E)$



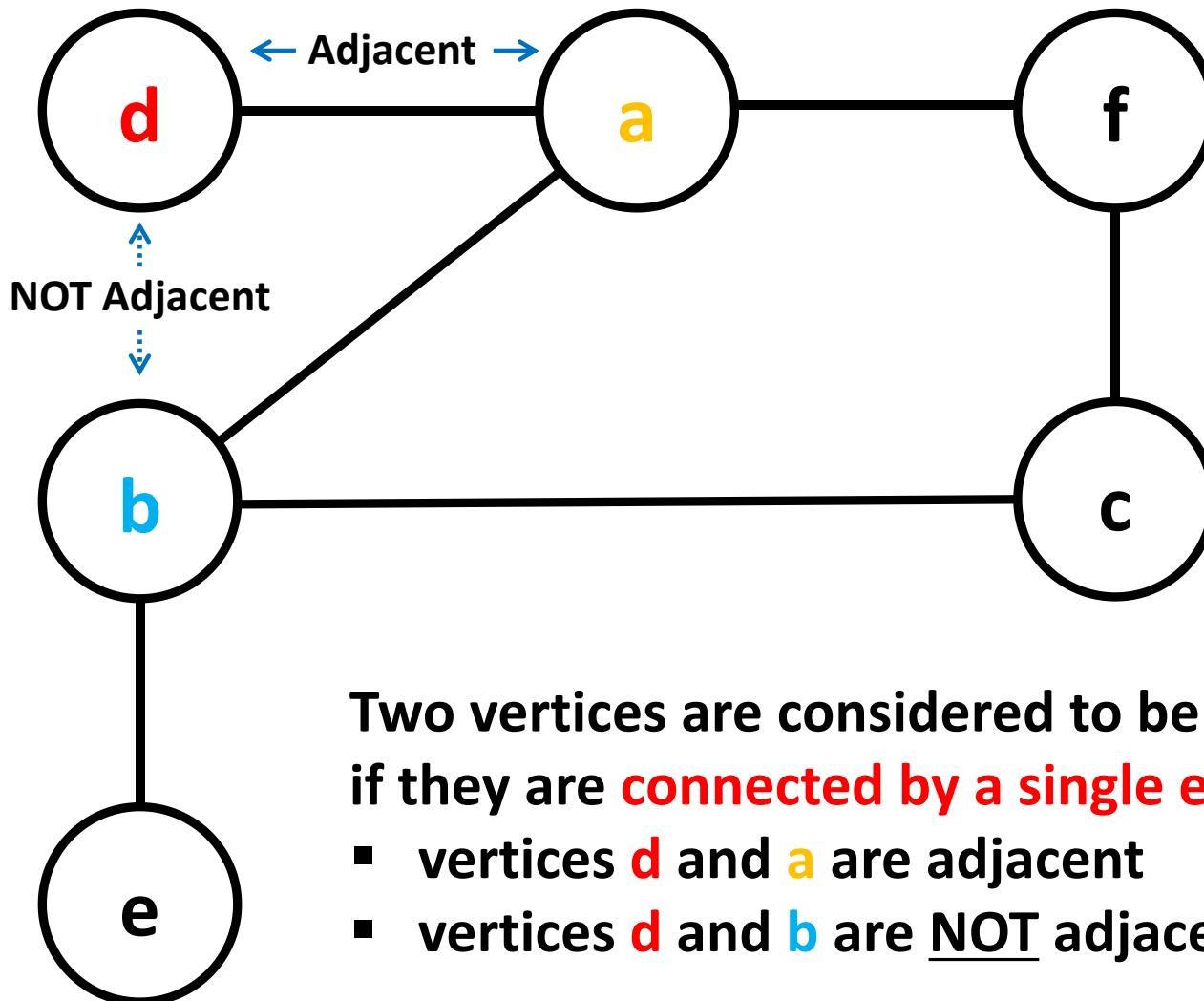
This graph is a pair $G = (V, E)$, where:

- $V = \{d, a, f, b, c, e\}$
- $E = \{\{a, d\}, \{a, f\}, \{a, b\}, \{c, f\}, \{b, c\}, \{b, e\}\}$

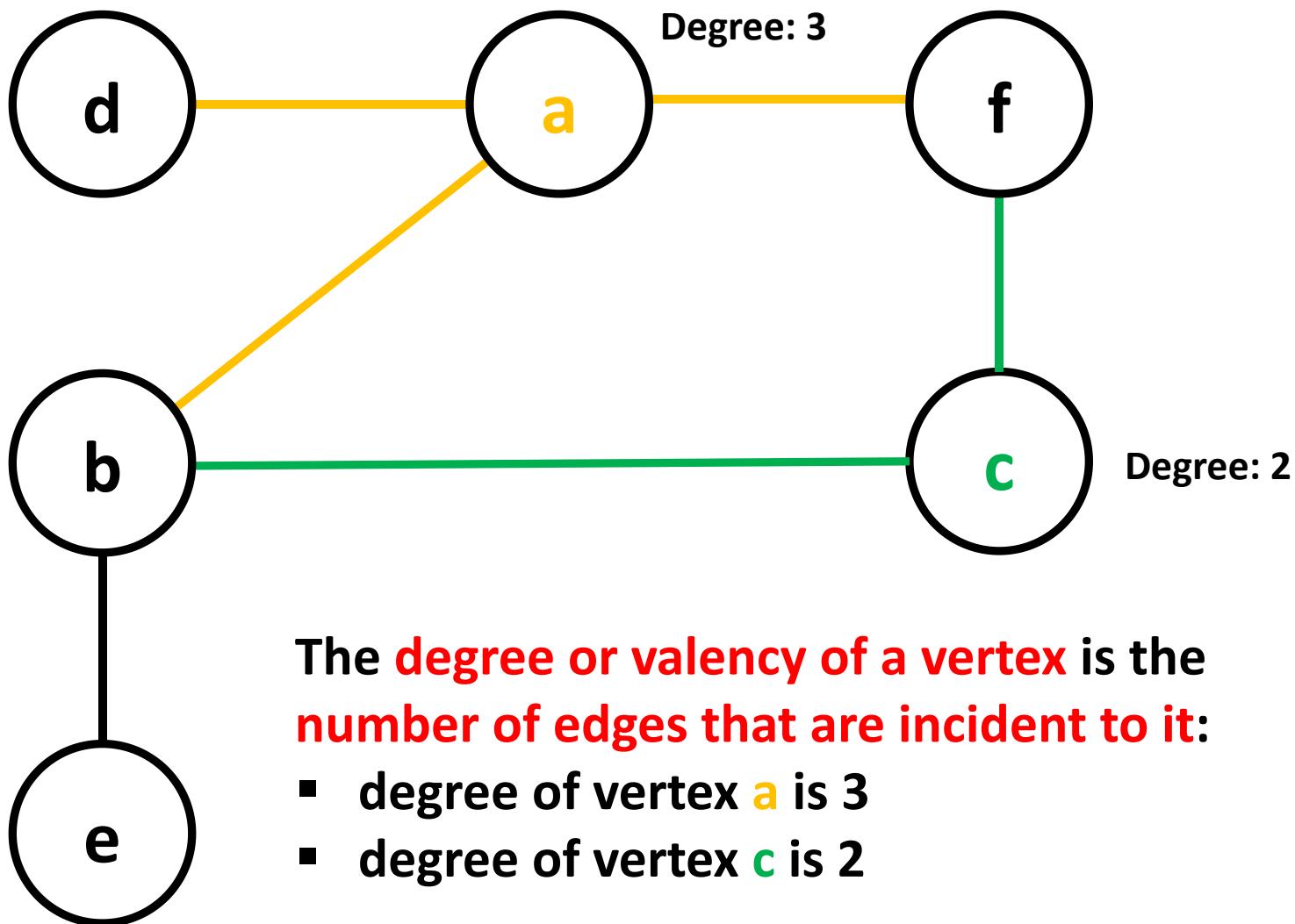
Graph: Edges



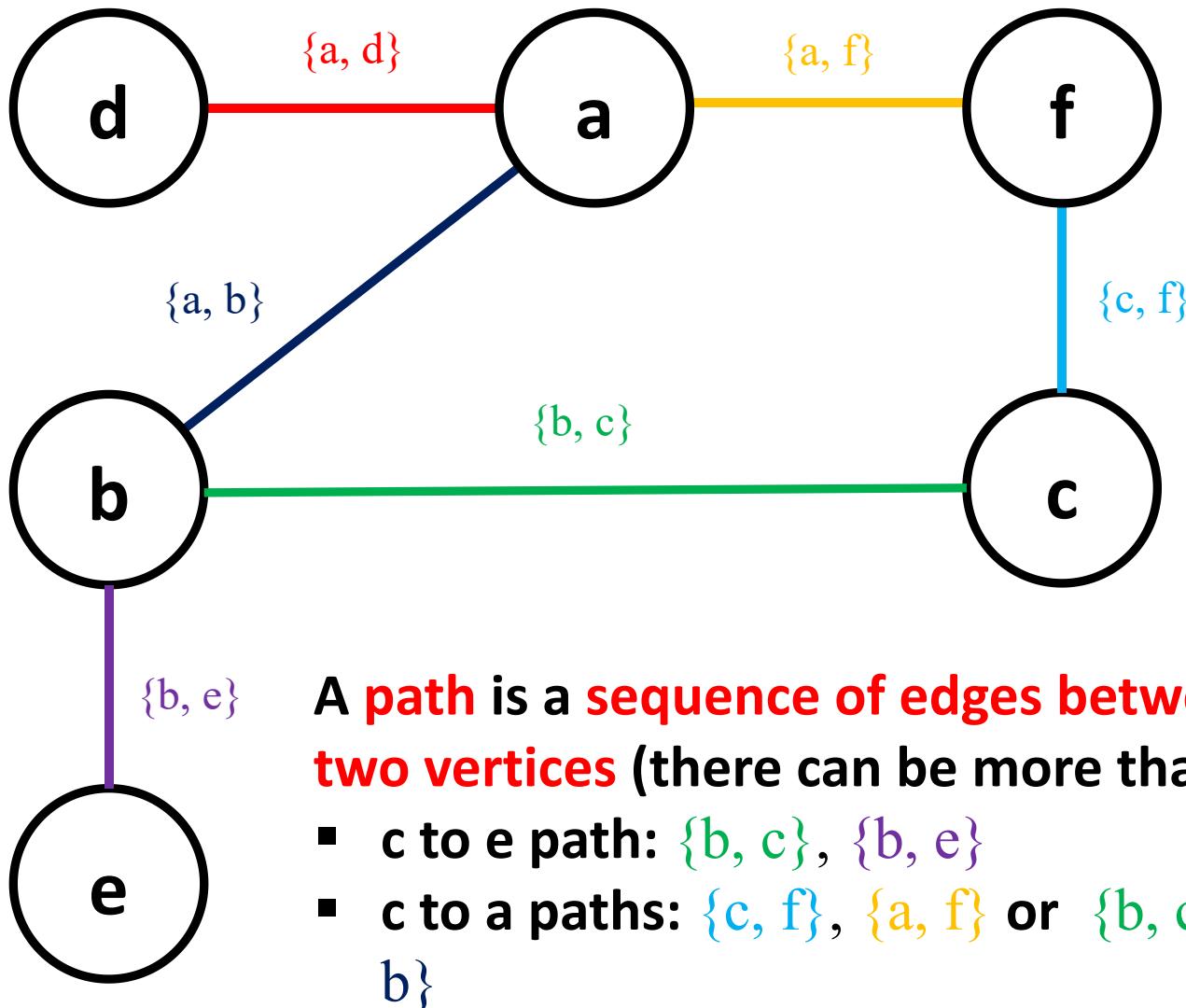
Graph: Vertex Adjacency



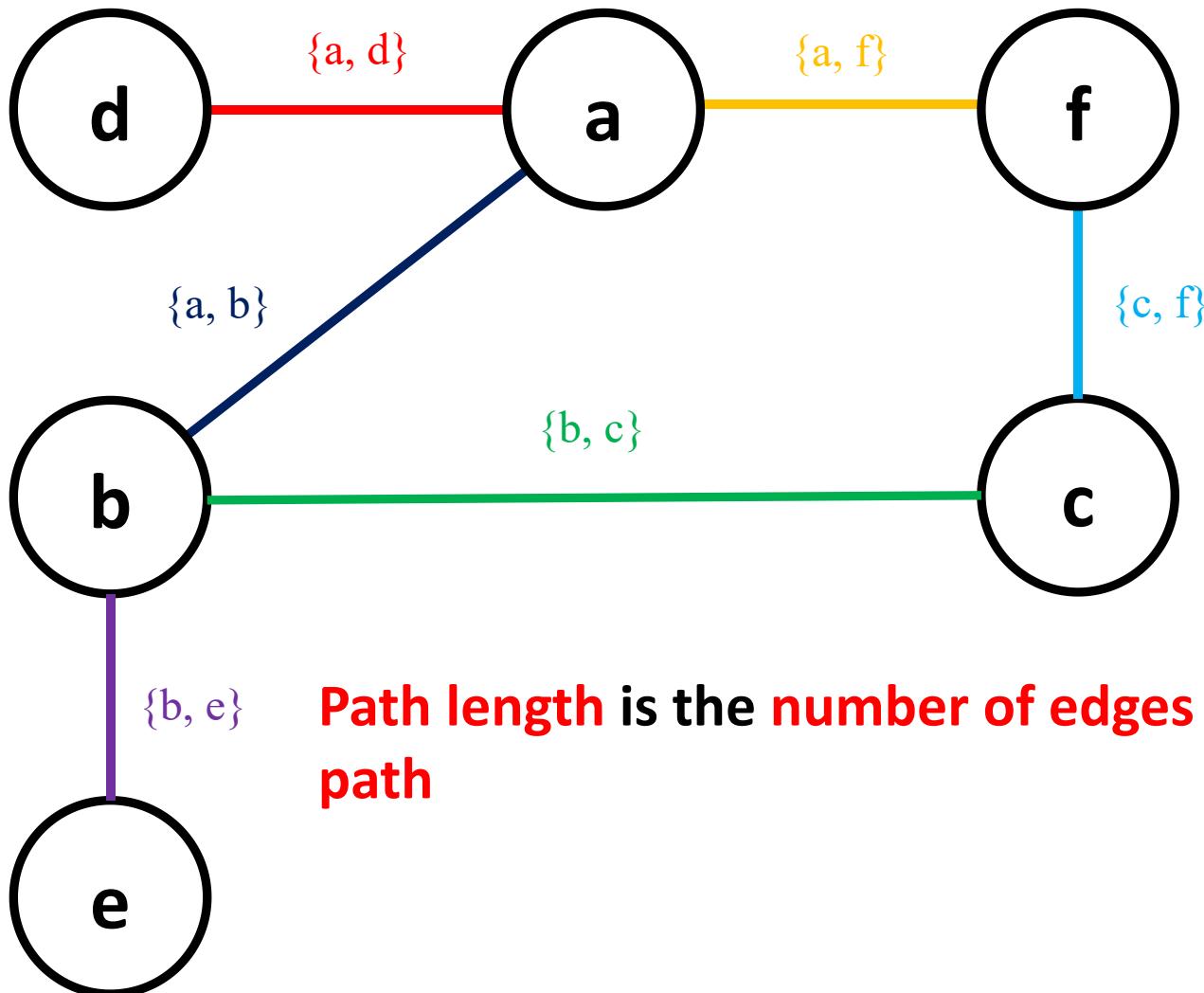
Graph: Vertex Degree / Valency



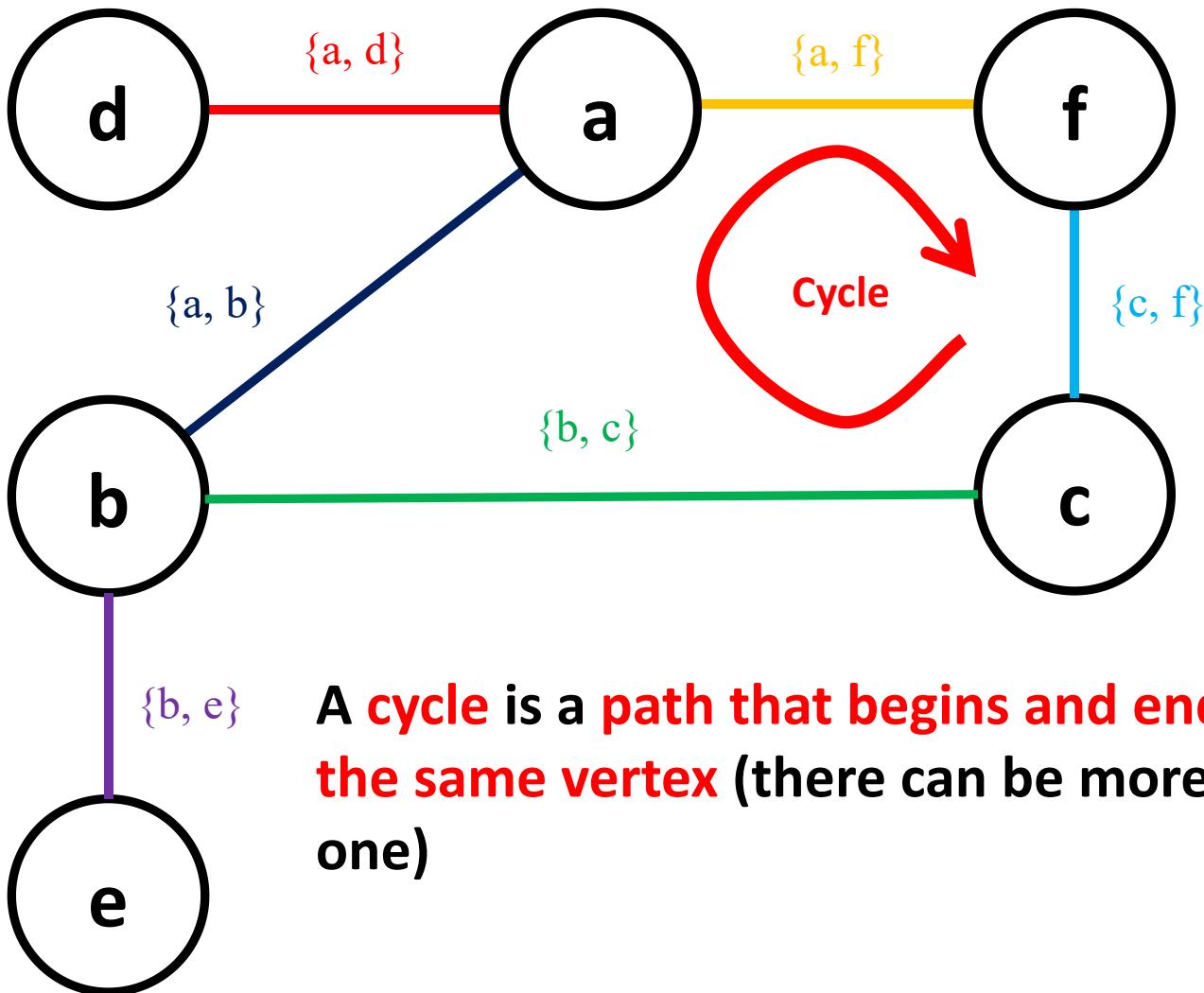
Graph: A Path



Graph: Path Length



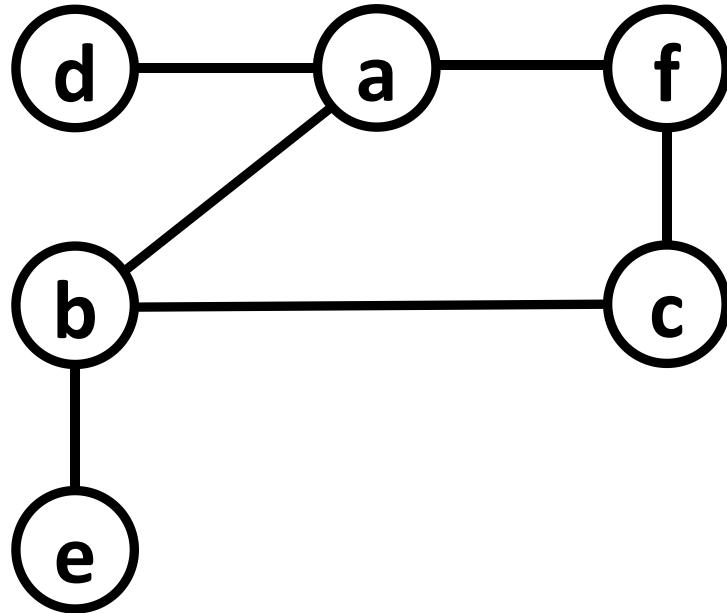
Graph: A Cycle



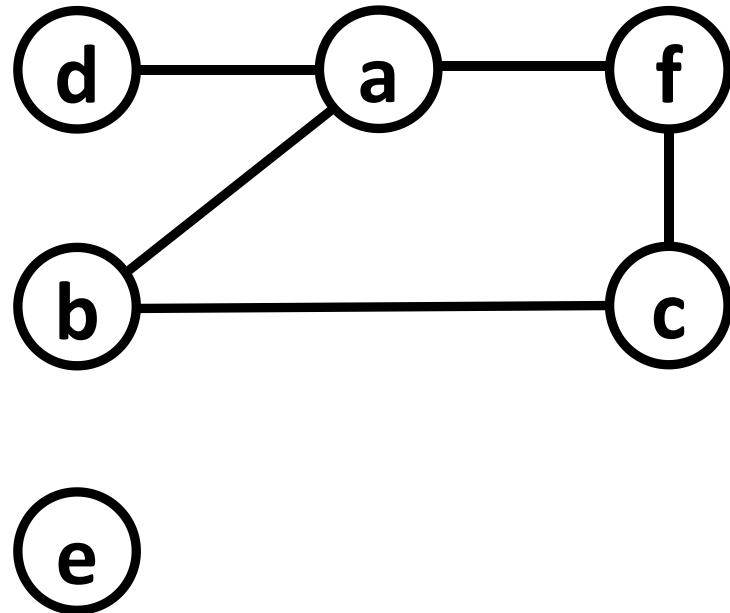
A **cycle** is a **path that begins and ends at the same vertex** (there can be more than one)

Connected vs. Non-Connected

Connected Graph



Non-Connected Graph



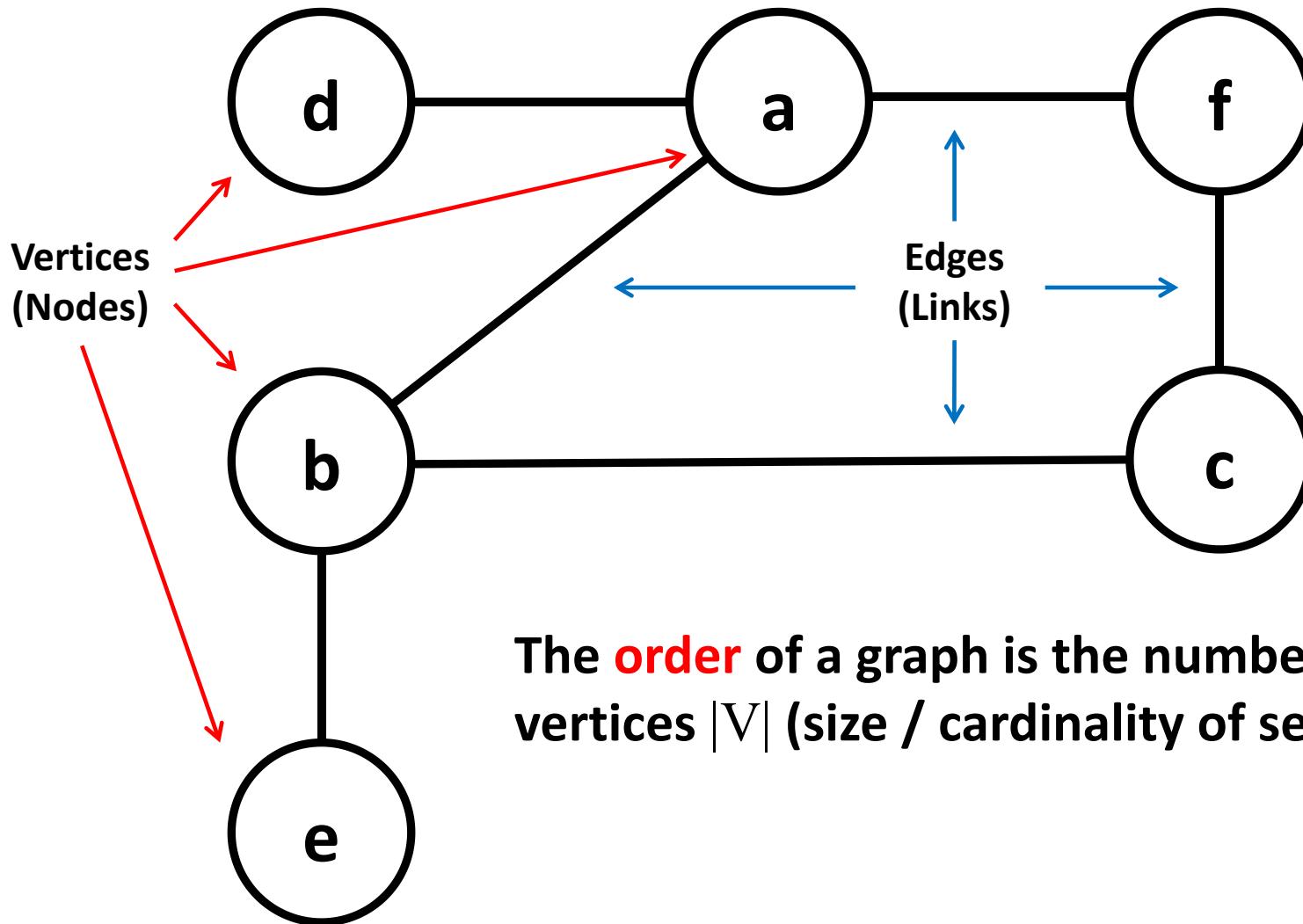
Connected graph:

- a graph is said to be **connected** if there exist at least one path from every vertex to every other vertex

Non-connected graph:

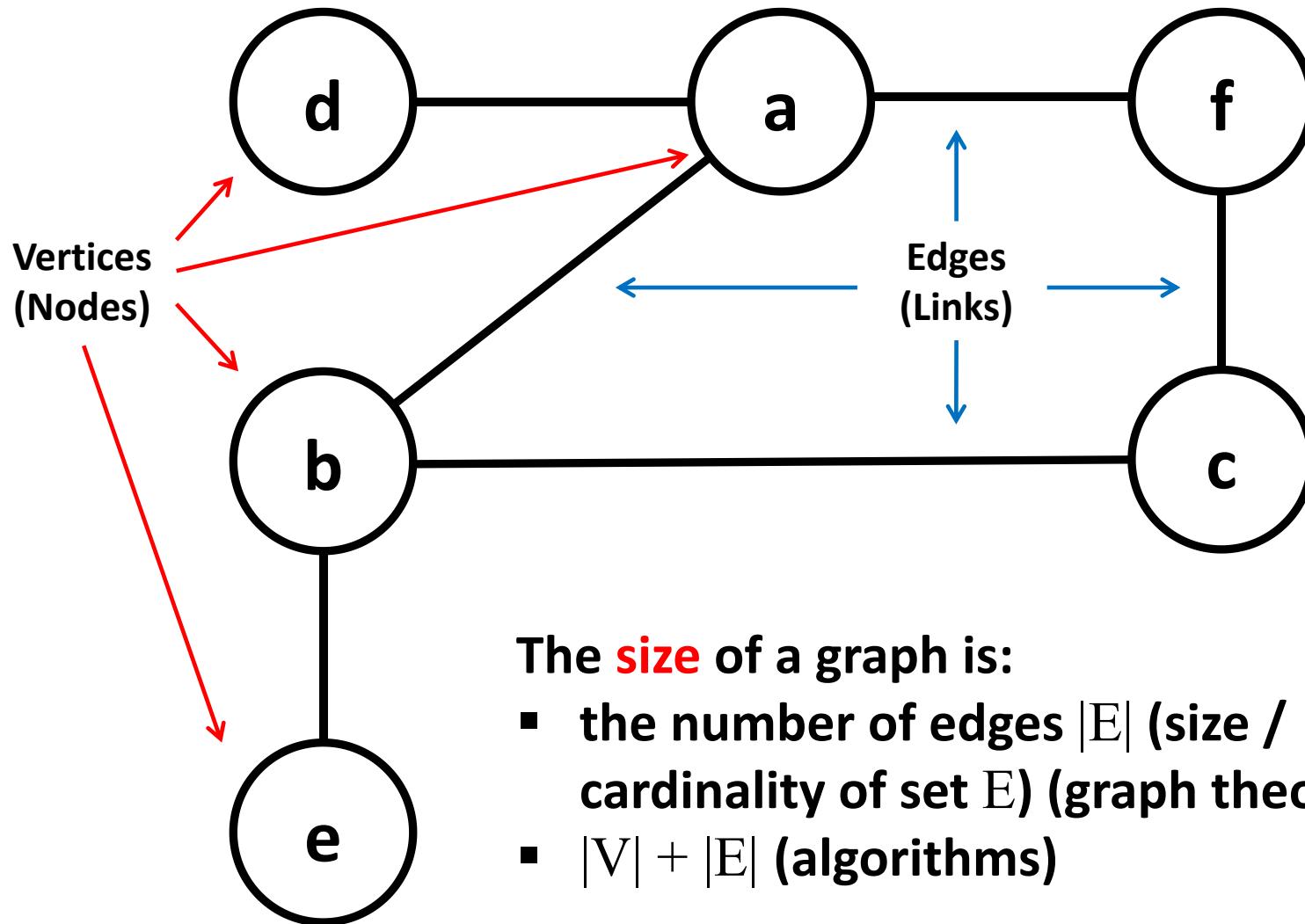
- a graph is **non-connected** if there exist at least one vertex such that there is no path to it from all the other vertices (vertex e in this example)
- vertex e is not “reachable”

Graph: Order



The **order** of a graph is the number of vertices $|V|$ (size / cardinality of set V)

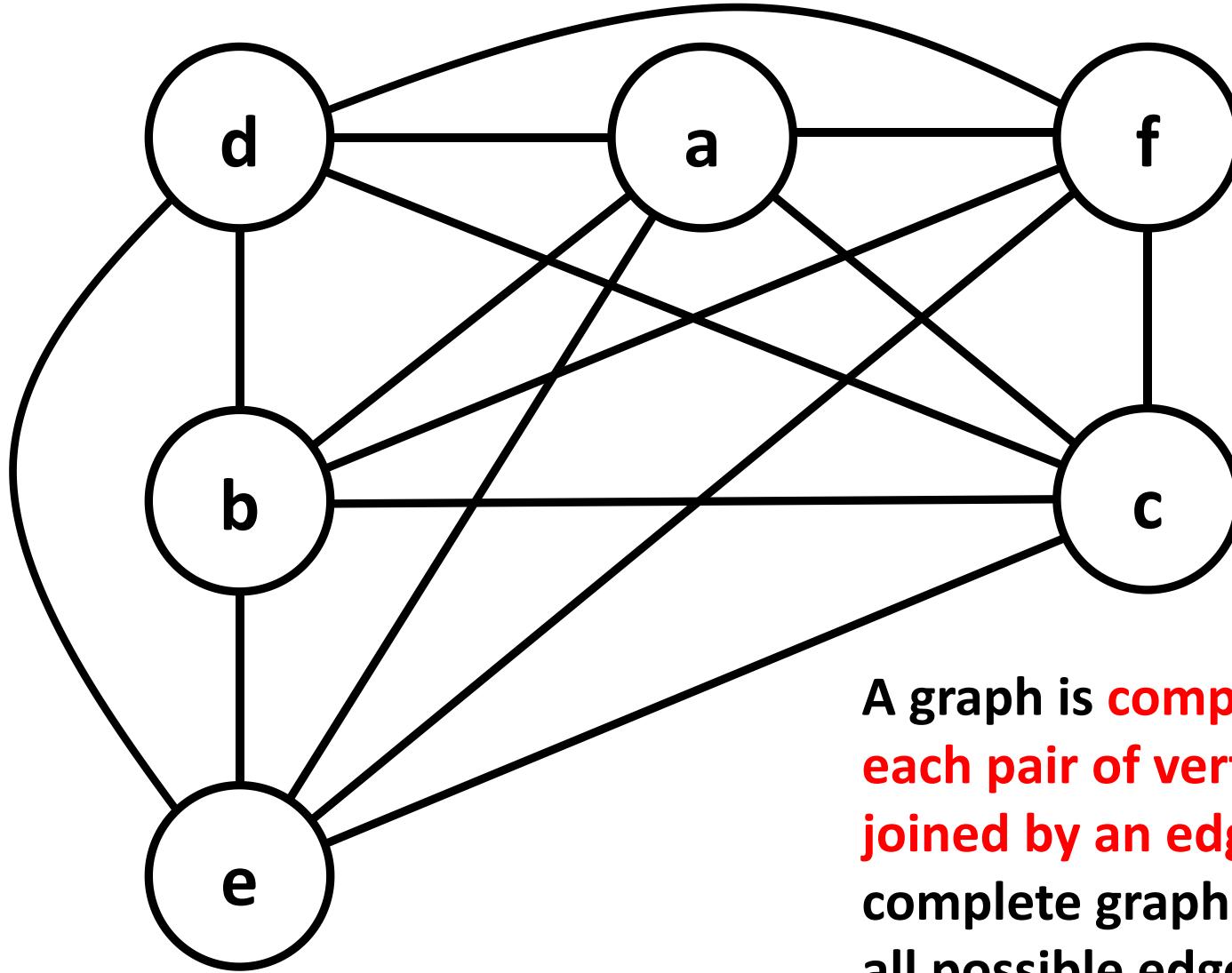
Graph: Size



The **size** of a graph is:

- the number of edges $|E|$ (size / cardinality of set E) (graph theory)
- $|V| + |E|$ (algorithms)

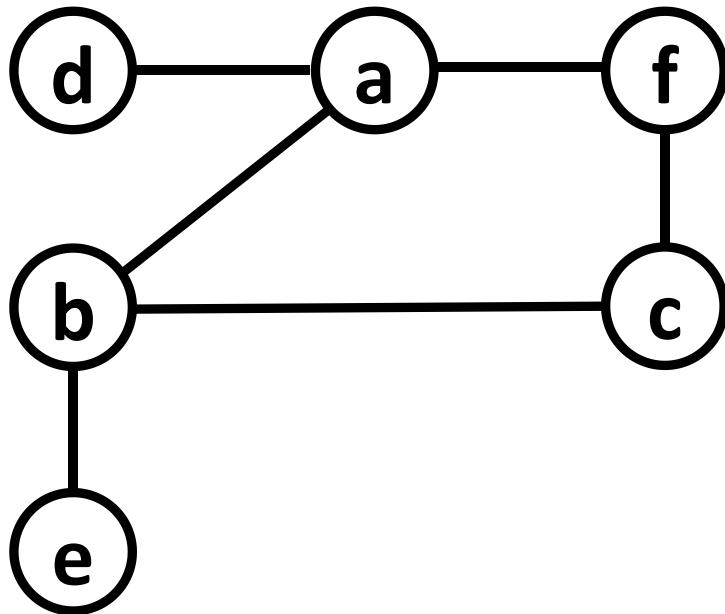
A Complete Graph



A graph is **complete** if each pair of vertices is joined by an edge. A complete graph contains all possible edges

Representation: Adjacency Matrix

Graph G



Adjacency Matrix for Graph G

	a	b	c	d	e	f
a	0	1	0	1	0	1
b	1	0	1	0	1	0
c	0	1	0	0	0	1
d	1	0	0	0	0	0
e	0	1	0	0	0	0
f	1	0	1	0	0	0

A graph $G = \{V, E\}$:

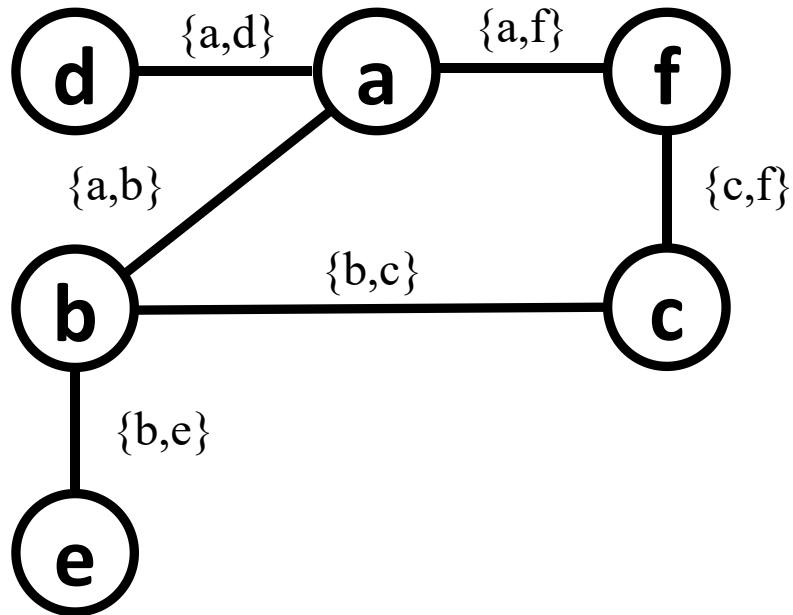
- $N = |V|$ vertices / nodes, and
- $|E|$ edges

Adjacency matrix for graph G:

- a 2D $N \times N$ array with:
 - 1s indicating an edge / connection between two vertices
 - 0s where there is no edge / connection between vertices

Representation: Incidence Matrix

Graph G



Incidence Matrix for Graph G

	{a,b}	{a,d}	{a,f}	{b,c}	{b,e}	{c,f}
a	1	1	1	0	0	0
b	1	0	0	1	1	0
c	0	0	0	1	0	1
d	0	1	0	0	0	0
e	0	0	0	0	1	0
f	0	0	1	0	0	1

A graph $G = \{V, E\}$:

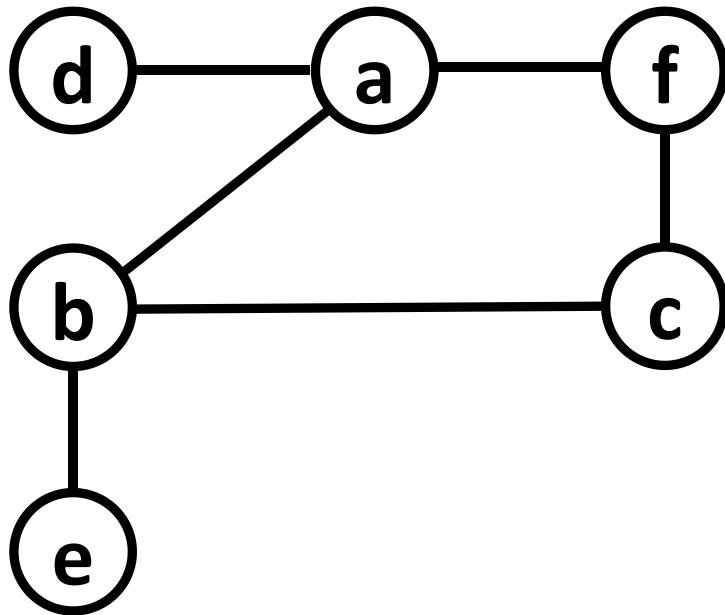
- $N = |V|$ vertices / nodes, and
- $|E|$ edges

Incidence matrix for graph G:

- a 2D $N \times N$ array with:
 - 1s indicating that an edge i is incident on vertex j
 - 0s otherwise

Representation: Adjacency List

Graph G



Adjacency List for Graph G

Vertex	List containing adjacent vertices
a	b → d → f
b	a → c → e
c	b → f
d	a
e	b
f	a → c

A graph $G = \{V, E\}$:

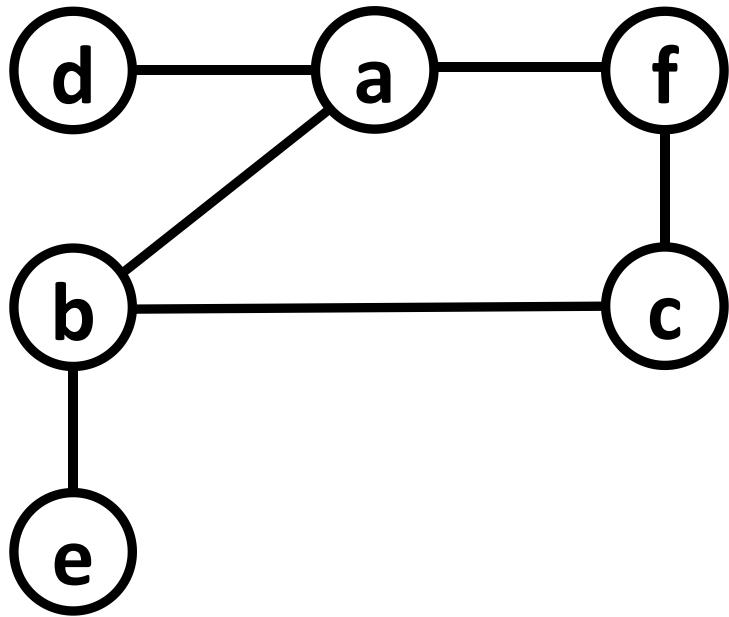
- $N = |V|$ vertices / nodes, and
- $|E|$ edges

Adjacency list for graph G:

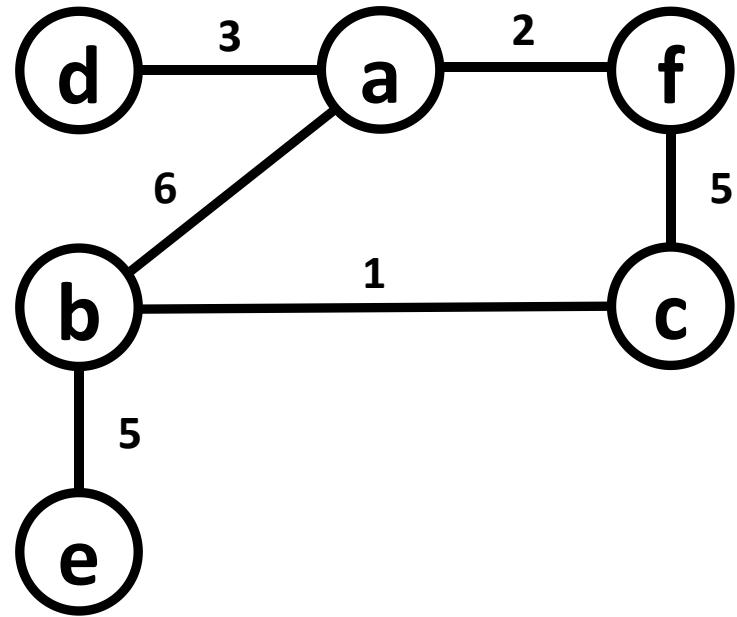
- an array (or a list) of lists, where:
 - each individual list corresponds to a single vertex i and includes all vertices adjacent to i
 - → indicates a link in a list

Non-Weighted vs. Weighted

Non-weighted Graph



Weighted Graph

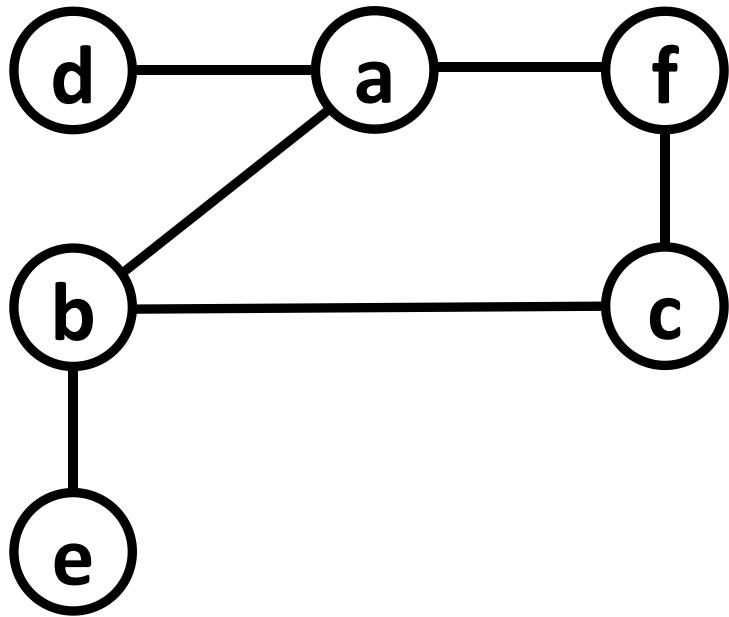


Weighted graph:

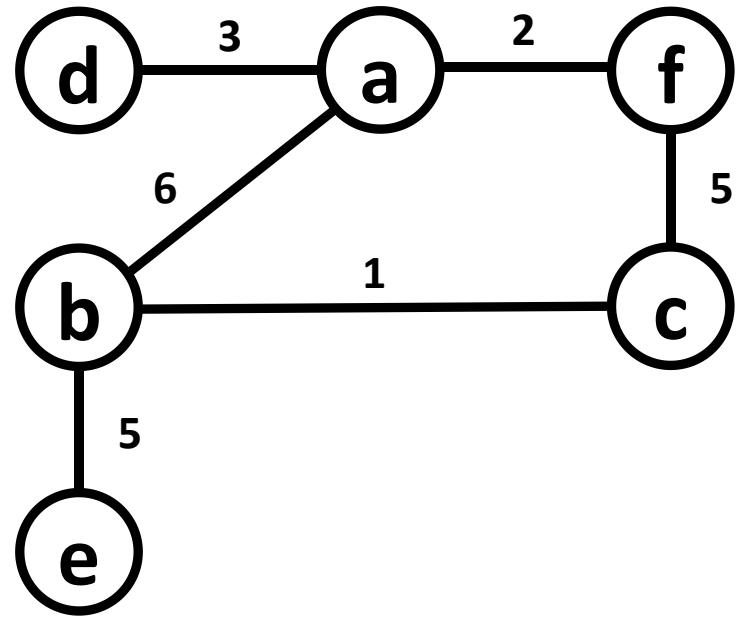
- a graph (sometimes called a network) is in which a number (weight) is assigned to each edge is considered to be weighted
- $G = \{V, E, w\}$

Non-Weighted vs. Weighted

Non-weighted Graph



Weighted Graph

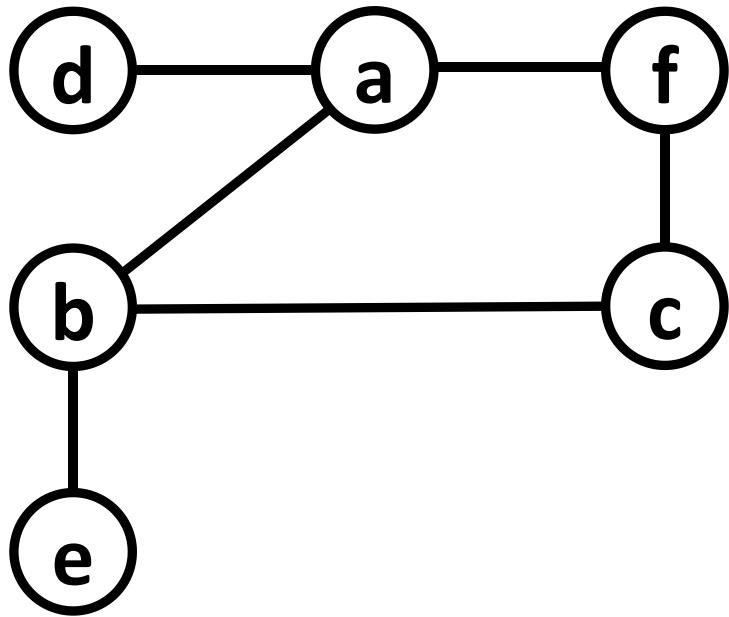


Weighted graph:

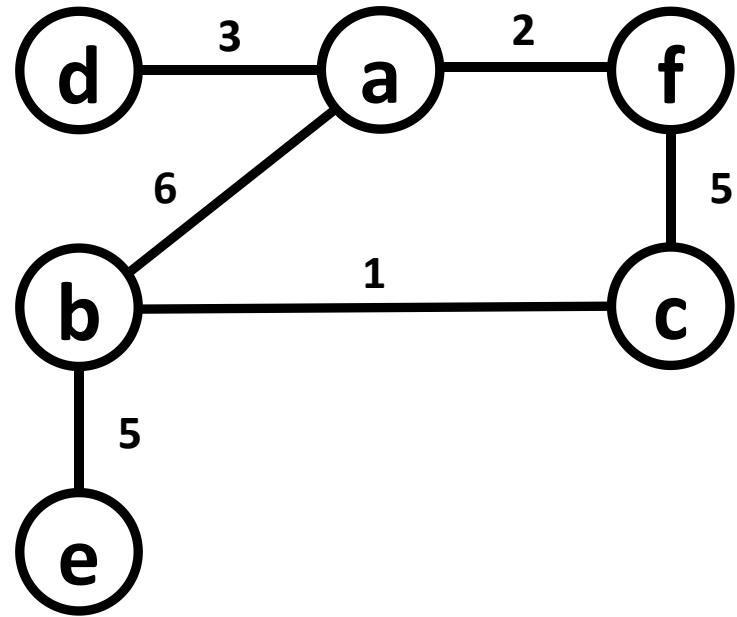
- weights can represent for costs, lengths or capacities, depending on the context / problem

Non-Weighted vs. Weighted

Non-weighted Graph



Weighted Graph

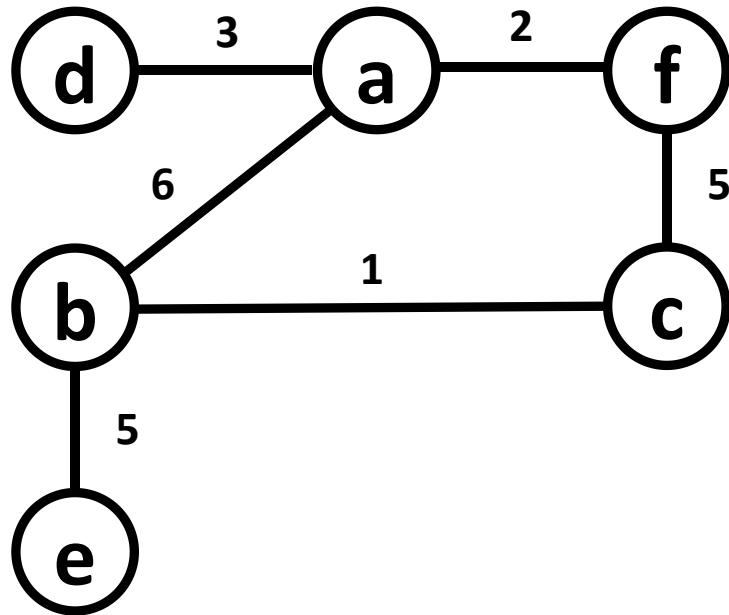


Weighted graph examples:

- maps (distances)
- sea shipping routes (cost)
- computer networks (throughput)
- airline route map (distances)
- Your CS MSc course plan (difficulty level)

Weighted Graph: Adjacency Matrix

Weighted Graph G



Adjacency Matrix for Graph G

	a	b	c	d	e	f
a	0	6	0	3	0	2
b	6	0	1	0	5	0
c	0	1	0	0	0	5
d	3	0	0	0	0	0
e	0	5	0	0	0	0
f	2	0	5	0	0	0

A graph $G = \{V, E, w\}$:

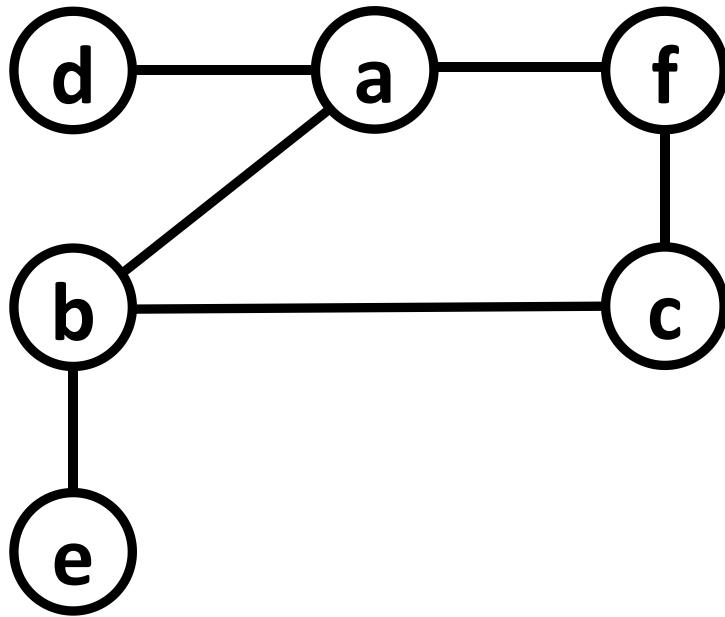
- $N = |V|$ vertices / nodes, and
- $|E|$ edges
- $|E|$ weights

Adjacency matrix for graph G:

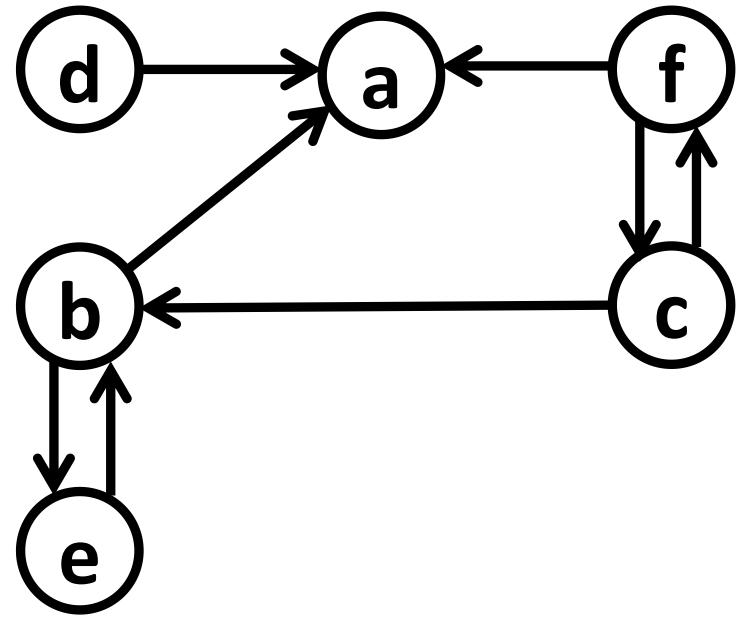
- a 2D $N \times N$ array with:
 - weights indicating an edge / connection between two vertices
 - 0s where there is no edge / connection between vertices

Undirected vs. Directed Graph

Undirected Graph



Directed Graph



Undirected graph:

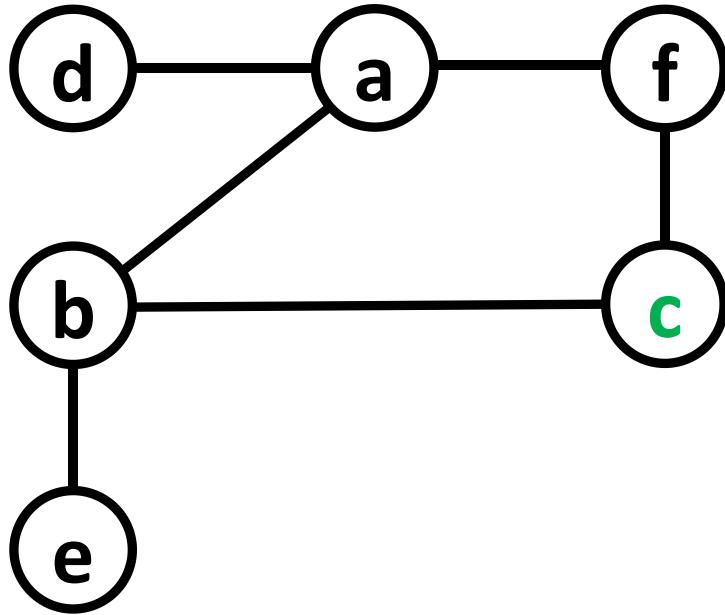
- a graph **in which edges DO NOT have orientations / directions** is called an **undirected graph**
- edge $\{b, e\}$ is the same as $\{e, b\}$

Directed graph:

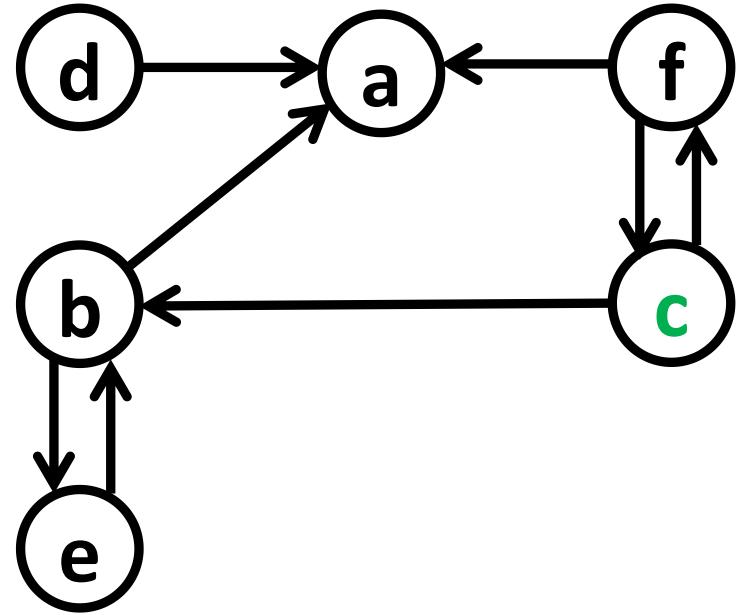
- a graph **in which edges have orientations / directions** is called a **directed graph (or digraph)**
- edge $\{b, e\}$ is **NOT** the same as $\{e, b\}$

Undirected vs. Directed Graph

Undirected Graph



Directed Graph



Undirected graph:

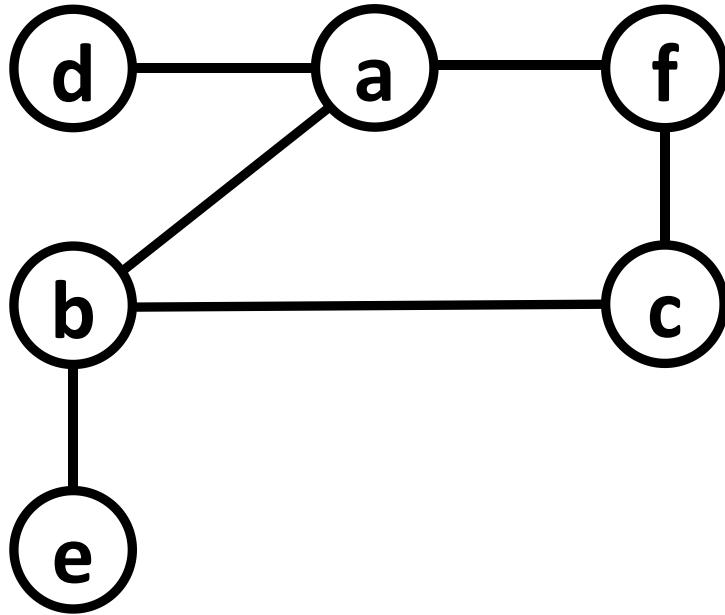
- the **degree of a vertex** is the **number of edges that are incident to it** (degree of vertex c: 2)

Directed graph:

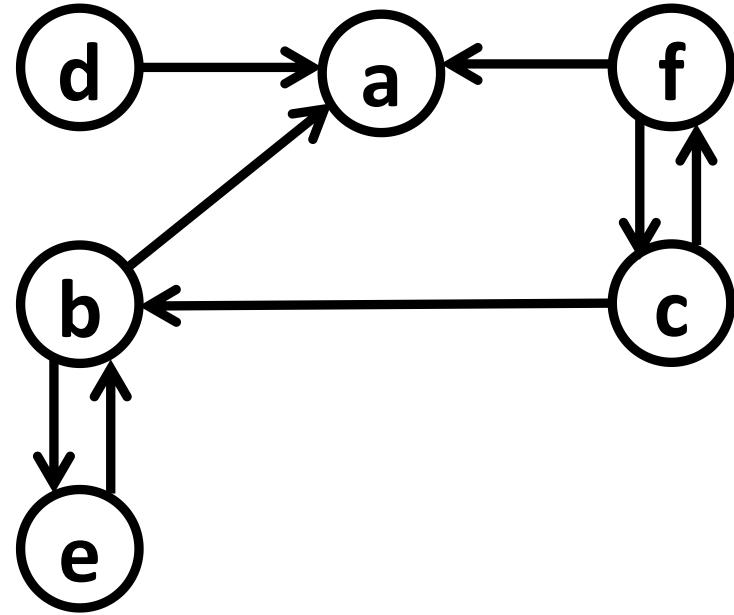
- the **in-degree of a vertex** is the **number of edges that are "entering" it** (in-degree of vertex c: 1)
- the **out-degree of a vertex** is the **number of edges that are "leaving" it** (out-degree of vertex c: 2)

Undirected vs. Directed Graph

Undirected Graph



Directed Graph



Undirected graph examples:

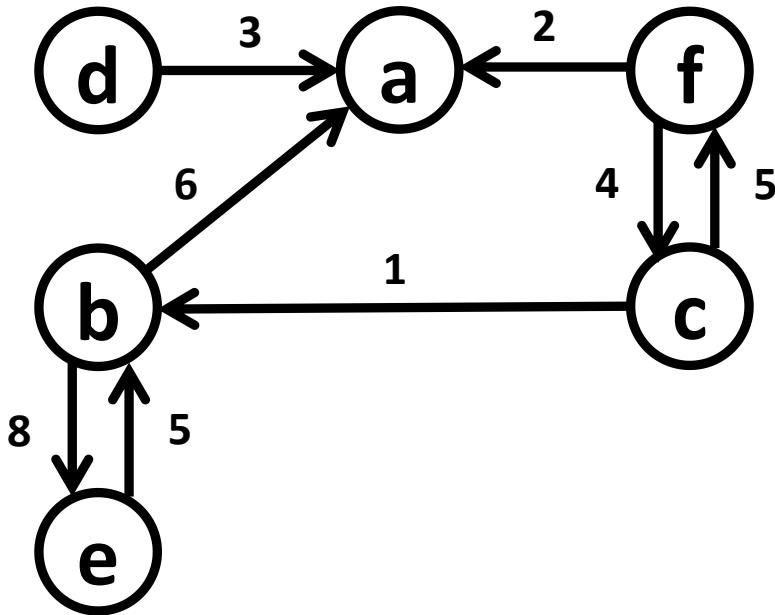
- two-way streets
- US interstate system
- computer networks
- Facebook friends
- academic paper collaborators

Directed graph examples:

- one-way streets
- social media followers
- computer networks / World Wide Web
- airline route map
- emergency exit routes

Directed Weighted Graph

Directed Weighted Graph G



Adjacency Matrix for Graph G

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	6	0	0	0	8	0
c	0	1	0	0	0	5
d	3	0	0	0	0	0
e	0	5	0	0	0	0
f	2	0	4	0	0	0

A graph $G = \{V, E, w\}$:

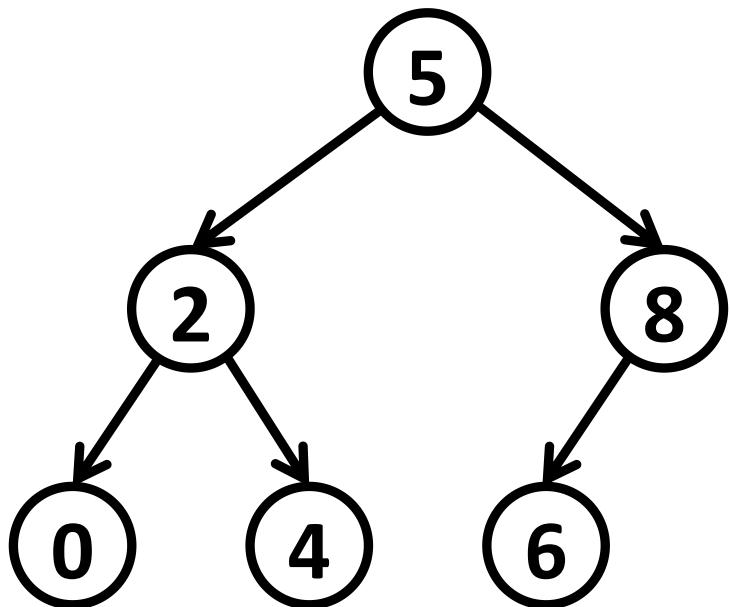
- $N = |V|$ vertices / nodes, and
- $|E|$ edges
- $|E|$ weights

Adjacency matrix for graph G:

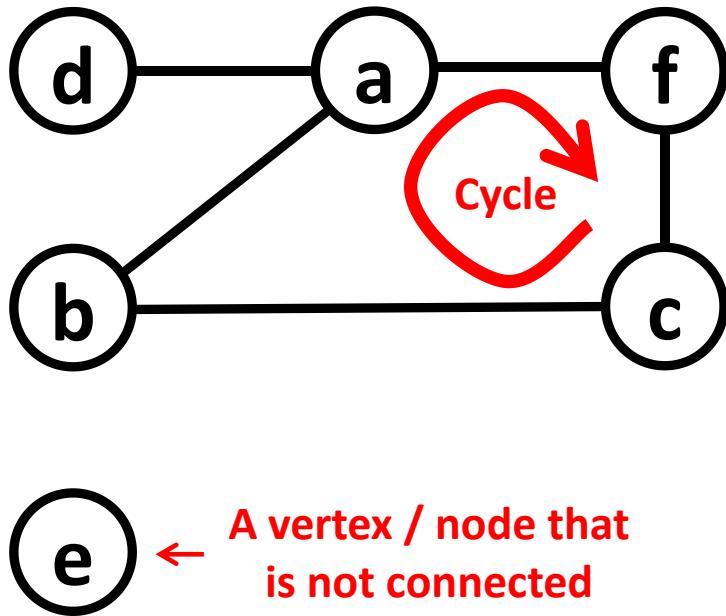
- a 2D $N \times N$ array with:
 - weights indicating an edge / connection between two vertices
 - 0s where there is no edge / connection between vertices

Graphs vs. Trees: Key Differences

Tree



Graph



A tree:

- is a graph where all vertices / nodes are connected to the root
- is a graph without any cycles (a tree is an acyclic graph)

A graph:

- a graph can be **non-connected** (some vertices are not “reachable”)
- a graph **can have one or more cycle**
- a graph **does not have to be directed**

Graphs: Traversals / Searches

Key difference between Tree and Graph Traversal:

- **Graphs: need to keep track of all visited vertices / nodes**

Purpose:

- **Finding a vertex / node**
- **Traversing the graph**

Types:

- **Depth First Traversal / Search**
- **Breadth First Traversal / Search**

Graph Traversals: Algorithms

General Depth-First Traversal (iterative)

```
dfs(Graph G, vertex v) //v start point
    initialize stack S
    S.push(v)
    mark v as visited
    while (S is not empty)
        m = S.pop()
        for all m's adjacent vertices k
            if k is not visited
                S.push(k)
                mark k as visited
        end
    end
end
```

General Breadth-First Traversal (iterative)

```
bfs(Graph G, vertex v) //v start point
    initialize queue Q
    S.enqueue(v)
    mark v as visited
    while (Q is not empty)
        m = Q.dequeue()
        for all m's adjacent vertices k
            if k is not visited
                Q.enqueue(k)
                mark k as visited
        end
    end
end
```

Applications:

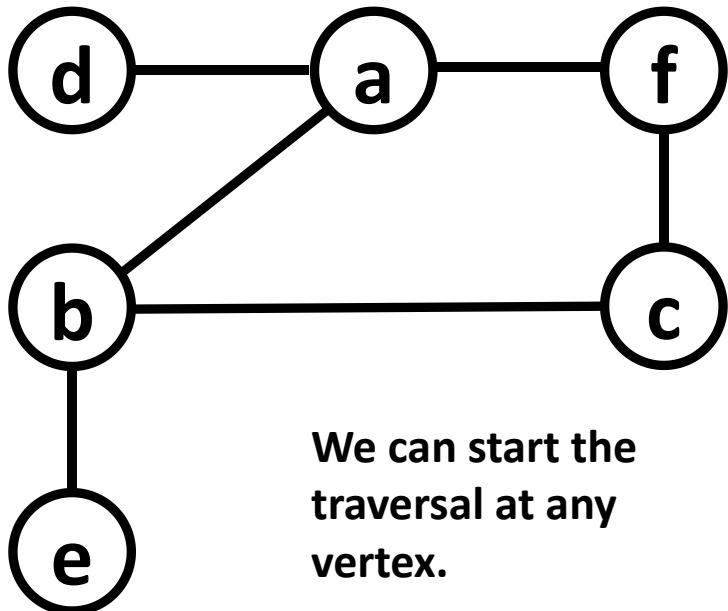
- Problems that require backtracking
- Path finding
- Solving puzzles with a single solution
- Detecting cycles in a graph

Applications:

- Nearest neighbor finding (for example in peer-to-peer networks or social networks)
- Web crawling / indexing
- Broadcasting information over a network

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

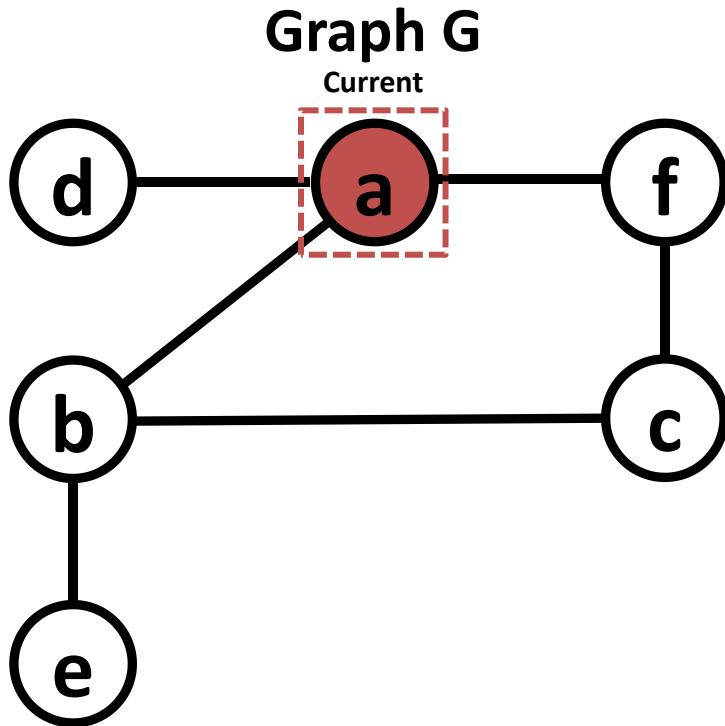
Current Vertex	
Already Visited Vertices	
None	
Adjacency List for Current Vertex	
Vertex Stack (Top to Bottom)	
Empty	

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	a
Already Visited Vertices	
a	
Adjacency List for Current Vertex	
a	b → d → f
Vertex Stack (Top to Bottom)	
a	

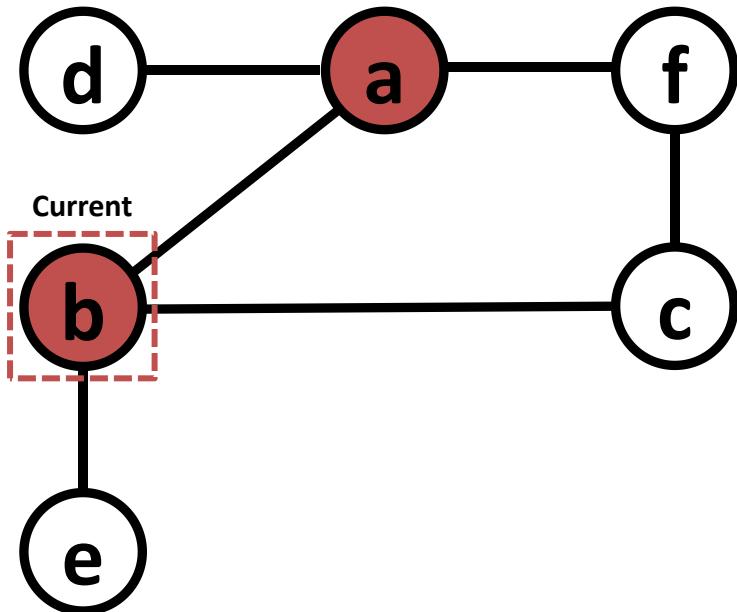
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	b
Already Visited Vertices	
a, b	
Adjacency List for Current Vertex	
b	a → c → e
Vertex Stack (Top to Bottom)	
b, a	

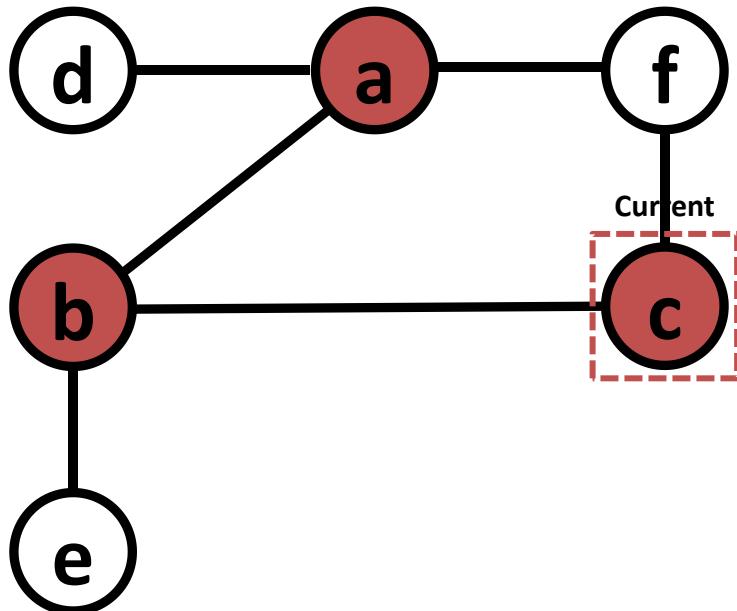
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	c
Already Visited Vertices	
a, b, c	
Adjacency List for Current Vertex	
c	b → f
Vertex Stack (Top to Bottom)	
c, b, a	

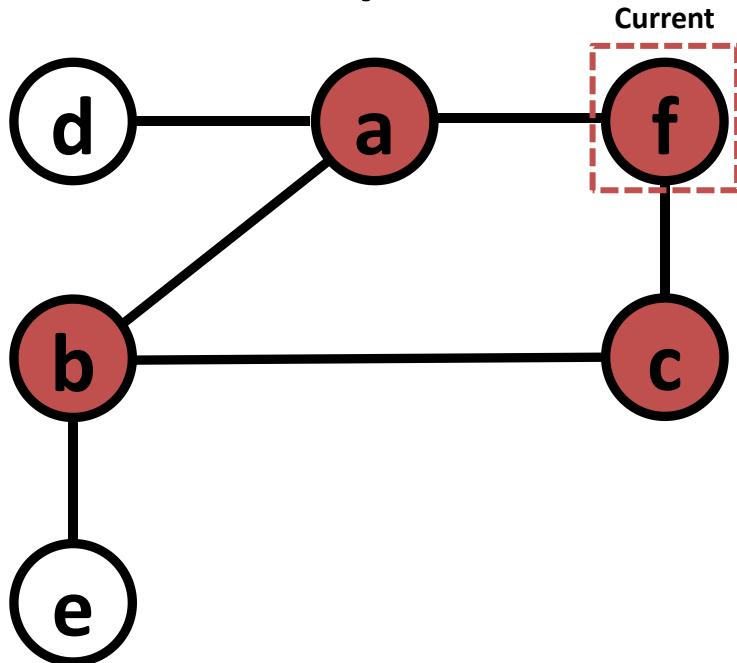
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Depth First Traversal For G

Current Vertex	f
Already Visited Vertices	
a, b, c, f	
Adjacency List for Current Vertex	
f	a → c [All adjacent visited]
Vertex Stack (Top to Bottom)	
f, c, b, a [pop() and backtrack]	



Already visited vertex

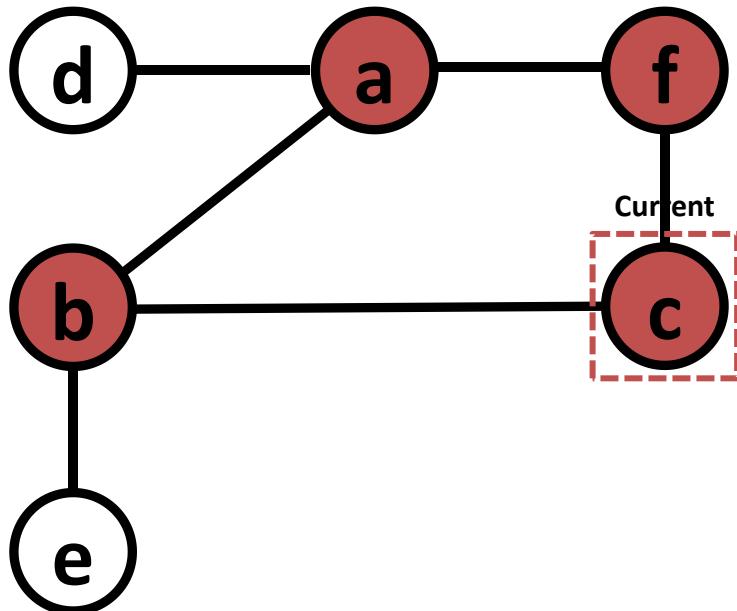


Not visited yet vertex

- Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack
- Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()
- Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	c
Already Visited Vertices	
a, b, c, f	
Adjacency List for Current Vertex	
c	b → f [All adjacent visited]
Vertex Stack (Top to Bottom)	
c, b, a [pop() and backtrack]	

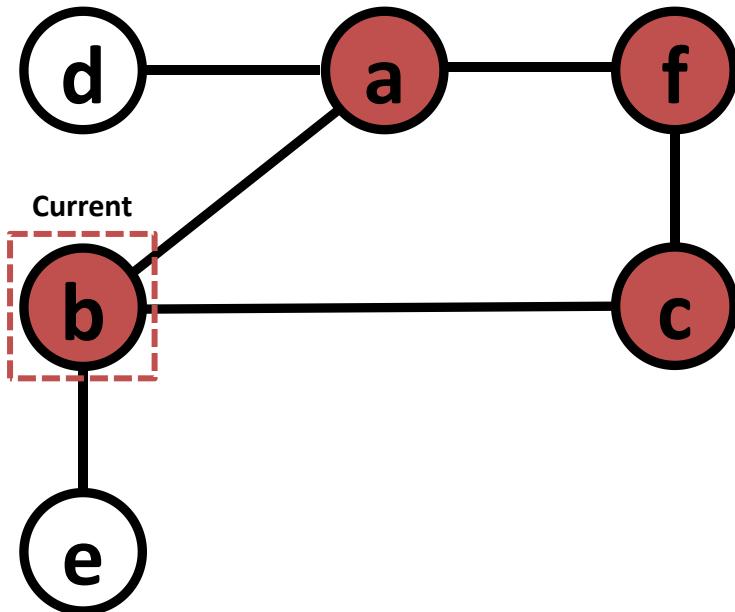
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	b
Already Visited Vertices	
a, b, c, f	
Adjacency List for Current Vertex	
b	a → c → e
Vertex Stack (Top to Bottom)	
b, a	

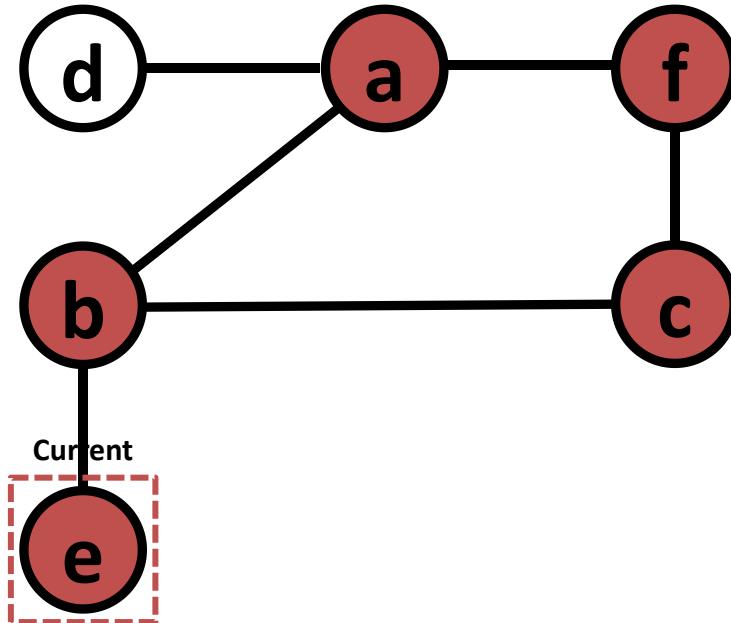
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	e
Already Visited Vertices	
a, b, c, f, e	
Adjacency List for Current Vertex	
e	b [All adjacent visited]
Vertex Stack (Top to Bottom)	
e, b, a [pop() and backtrack]	

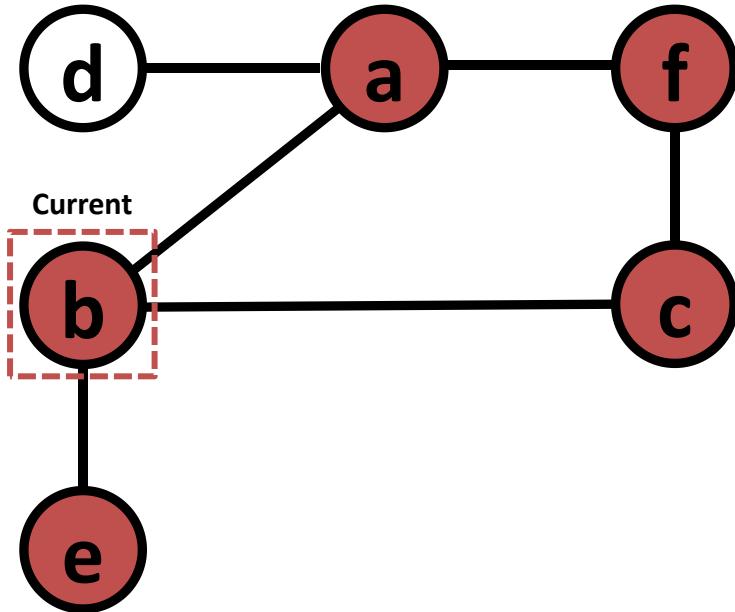
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

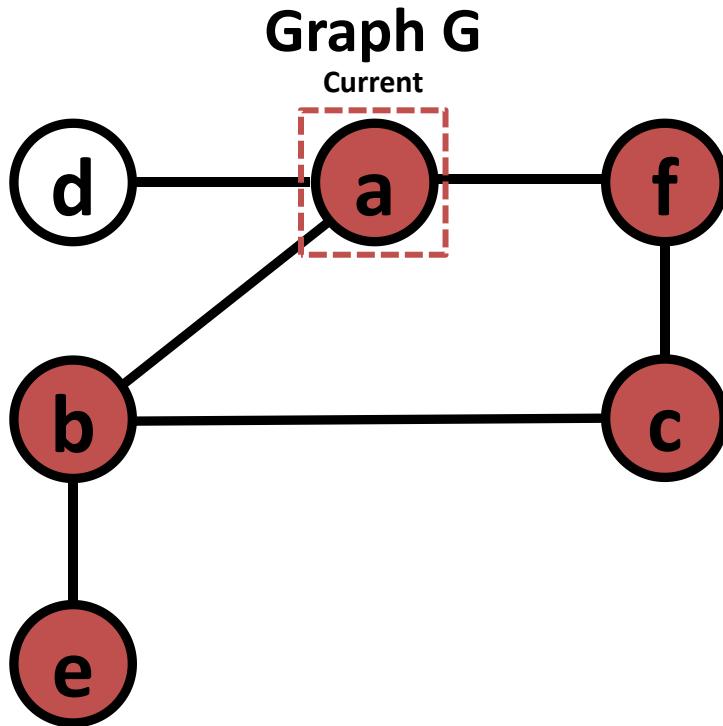
Current Vertex	b
Already Visited Vertices	
a, b, c, f, e	
Adjacency List for Current Vertex	
b	a → c → e [All adjacent visited]
Vertex Stack (Top to Bottom)	
b, a	[pop() and backtrack]

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	a
Already Visited Vertices	
a, b, c, f, e	
Adjacency List for Current Vertex	
a	b → d → f
Vertex Stack (Top to Bottom)	
a	

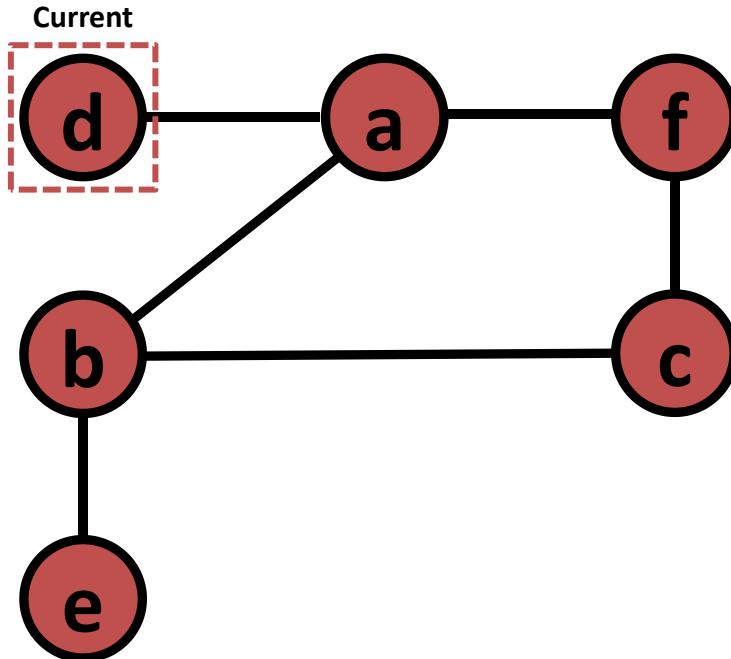
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

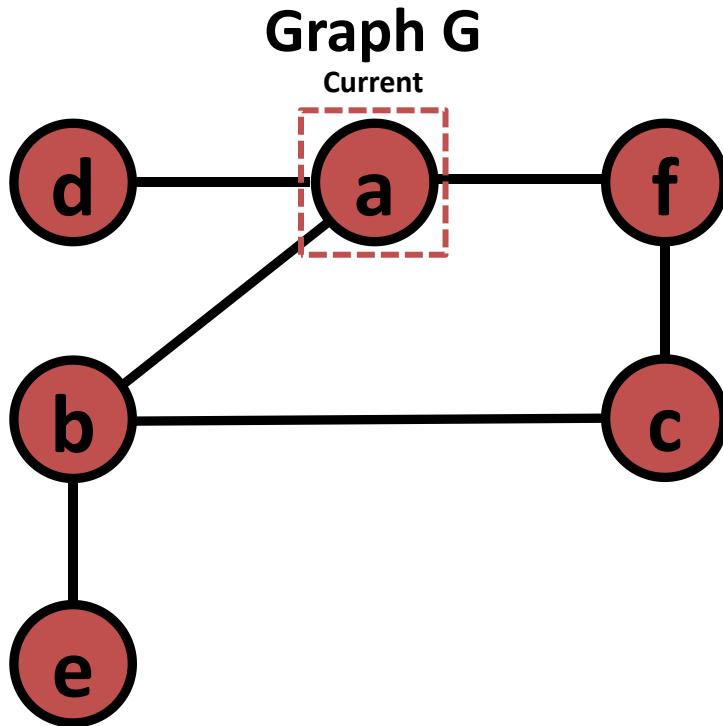
Current Vertex	d
Already Visited Vertices	
a, b, c, f, e, d	
Adjacency List for Current Vertex	
d	a [All adjacent visited]
Vertex Stack (Top to Bottom)	
d, a [pop() and backtrack]	

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	a
Already Visited Vertices	
a, b, c, f, b, e	
Adjacency List for Current Vertex	
a	b → d → f [All adjacent visited]
Vertex Stack (Top to Bottom)	
a [pop() and Done!]	

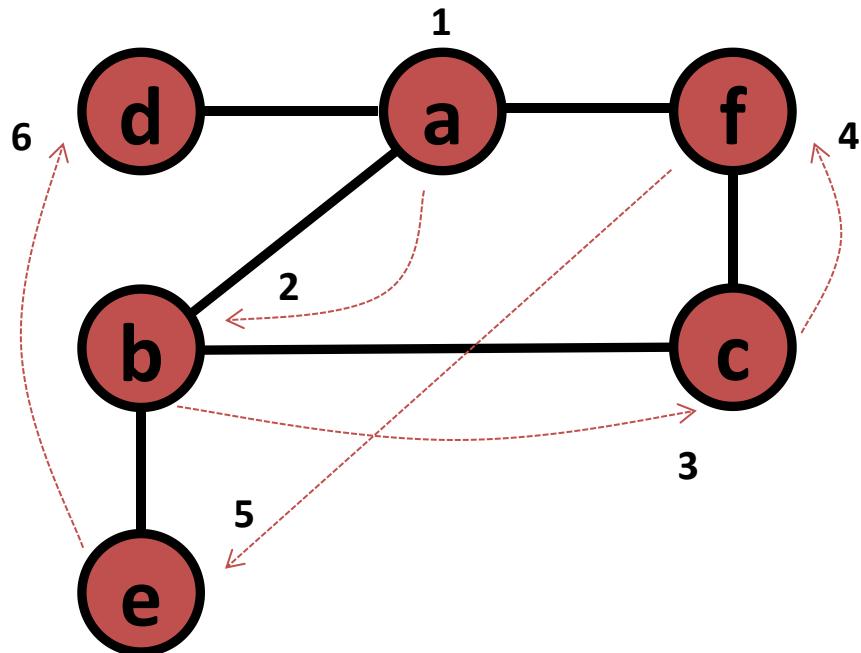
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Depth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Depth First Traversal For G

Current Vertex	
Already Visited Vertices	a, b, c, f, b, e
Adjacency List for Current Vertex	
Vertex Stack (Top to Bottom)	

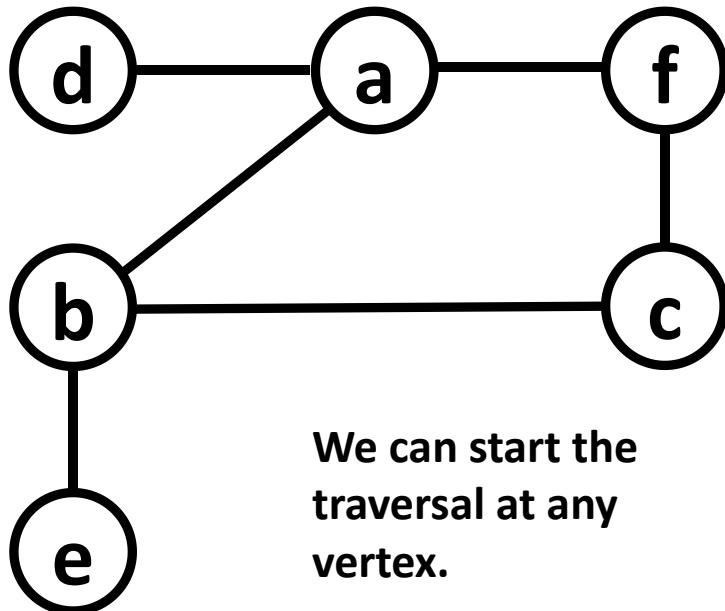
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, push it on the stack

Rule 2: If no unvisited vertices (adjacent to current), pop(), current vertex = top()/peek()

Rule 3: If Rules 1/2 impossible, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

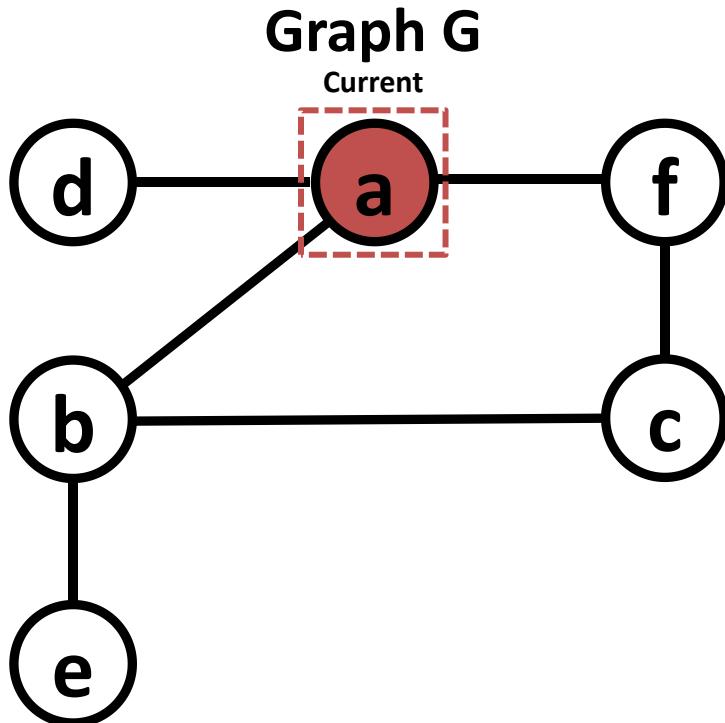
Current Vertex	
Already Visited Vertices	
None	
Adjacency List for Current Vertex	
Vertex Queue (Front to Rear)	
Empty	

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited vertices (adjacent to current), current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

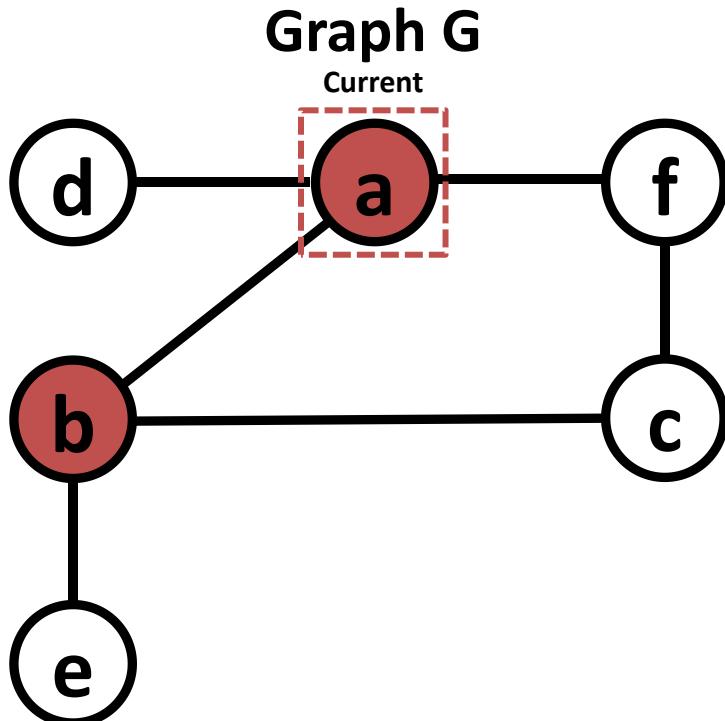
Current Vertex	a
Already Visited Vertices	
a	
Adjacency List for Current Vertex	
a	b → d → f
Vertex Queue (Front to Rear)	
Empty	

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

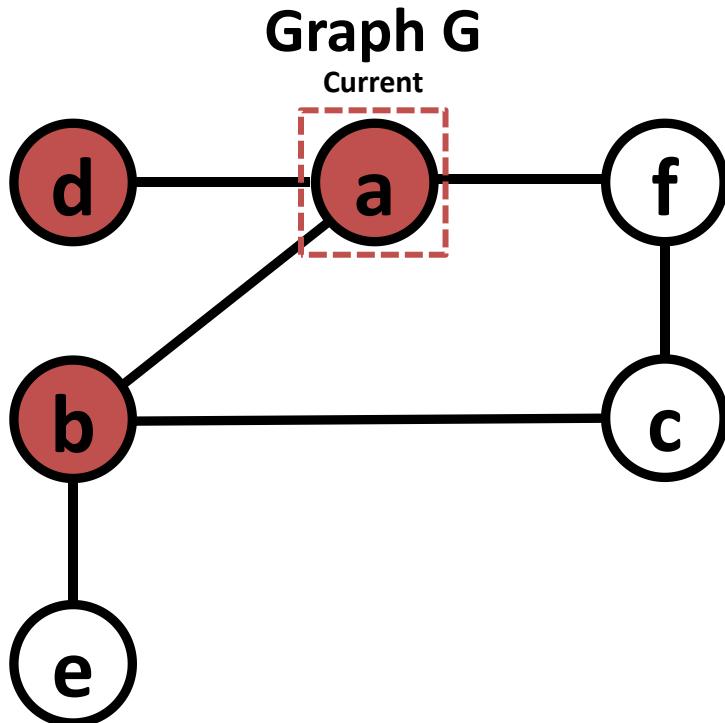
Current Vertex	a
Already Visited Vertices	
a, b	
Adjacency List for Current Vertex	
a	b → d → f
Vertex Queue (Front to Rear)	
b	

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal



Already visited vertex

Not visited yet vertex

Breadth First Traversal For G

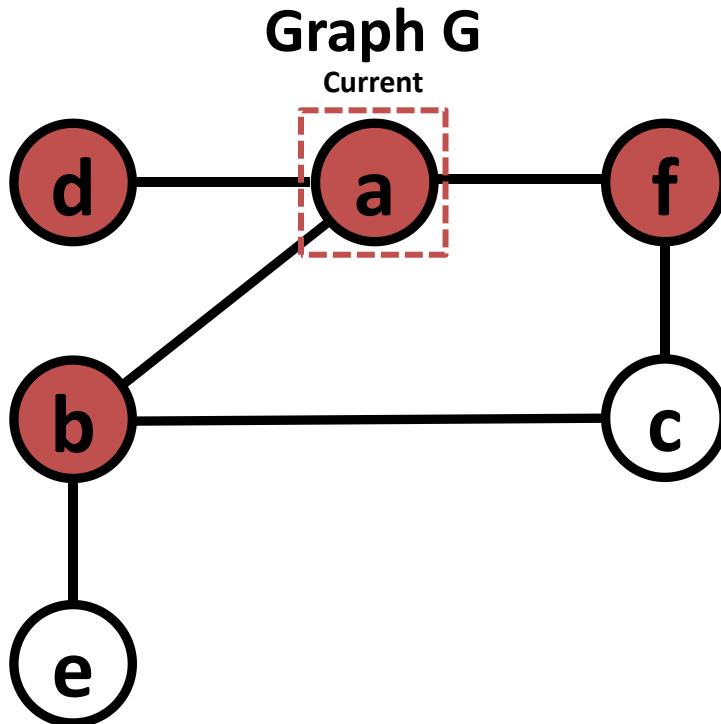
Current Vertex	a
Already Visited Vertices	
a, b, d	
Adjacency List for Current Vertex	
a	b → d → f
Vertex Queue (Front to Rear)	
b, d	

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

Current Vertex	a
Already Visited Vertices	
a, b, d, f	
Adjacency List for Current Vertex	
a	b → d → f [All visited now!]
Vertex Queue (Front to Rear)	
b, d, f [dequeue()]	

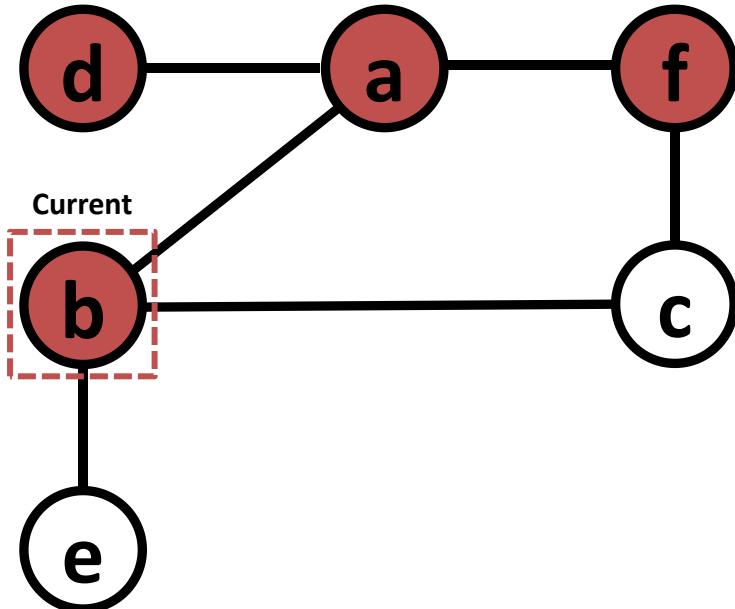
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

Current Vertex	b
Already Visited Vertices	
a, b, d, f	
Adjacency List for Current Vertex	
b	a → c → e
Vertex Queue (Front to Rear)	
d, f	

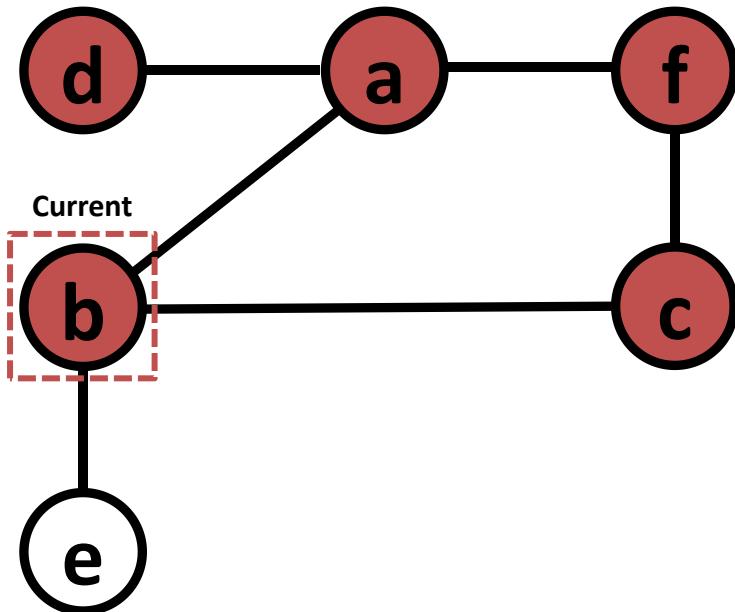
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

Current Vertex	b
Already Visited Vertices	
a, b, d, f, c	
Adjacency List for Current Vertex	
b	a → c → e
Vertex Queue (Front to Rear)	
d, f, c	

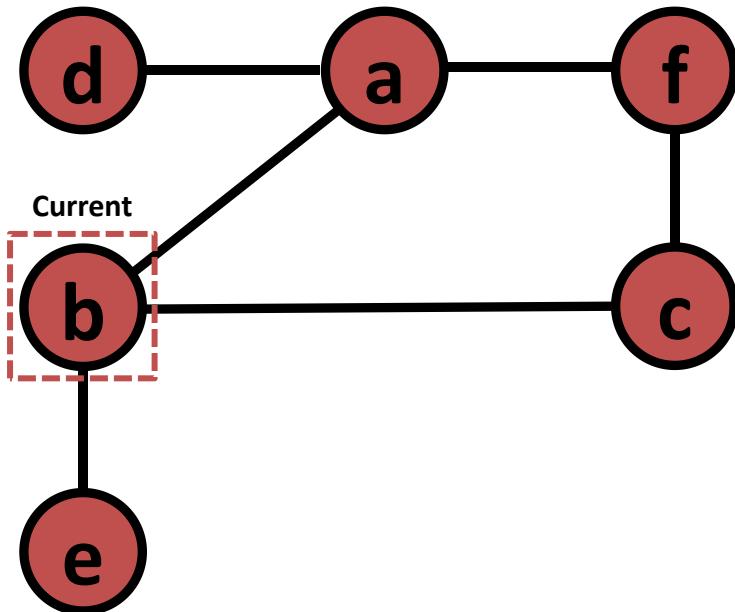
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

Current Vertex	b
Already Visited Vertices	
a, b, d, f, c, e	
Adjacency List for Current Vertex	
b	a → c → e [All visited now!]
Vertex Queue (Front to Rear)	
d, f, c, e [dequeue()]	

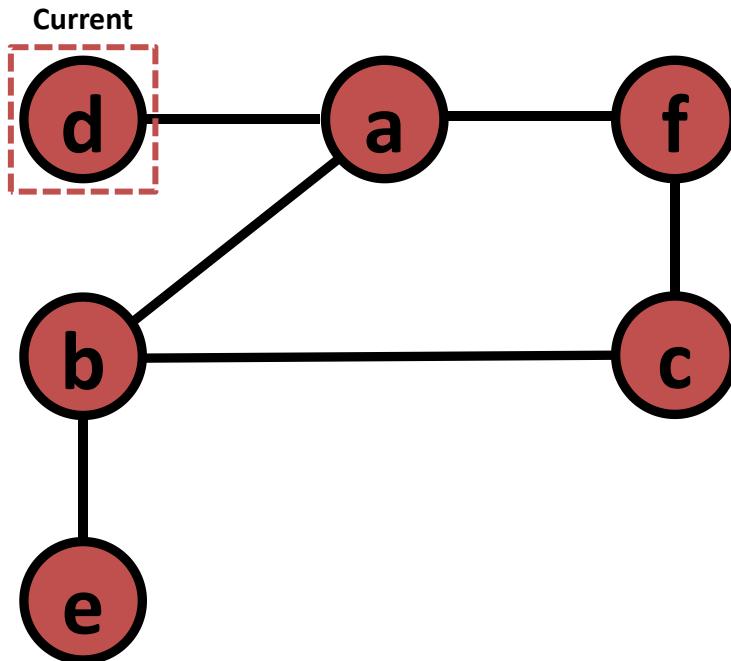
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

Current Vertex	d
Already Visited Vertices	
a, b, d, f, c, e	
Adjacency List for Current Vertex	
d	a [All visited now!]
Vertex Queue (Front to Rear)	
f, c, e [dequeue()]	

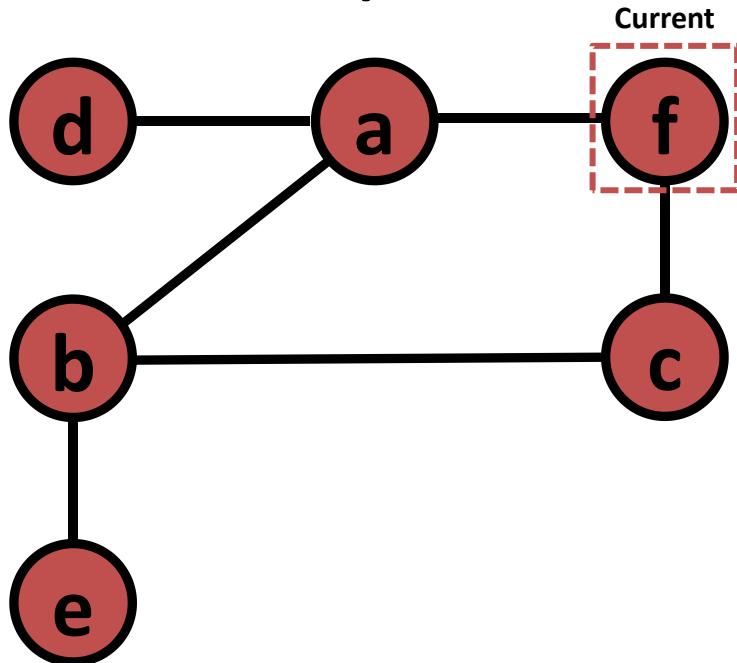
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex

Not visited yet vertex

Breadth First Traversal For G

Current Vertex	f
Already Visited Vertices	
a, b, d, f, c, e	
Adjacency List for Current Vertex	
f	a → c [All visited now!]
Vertex Queue (Front to Rear)	
c, e [dequeue()]	

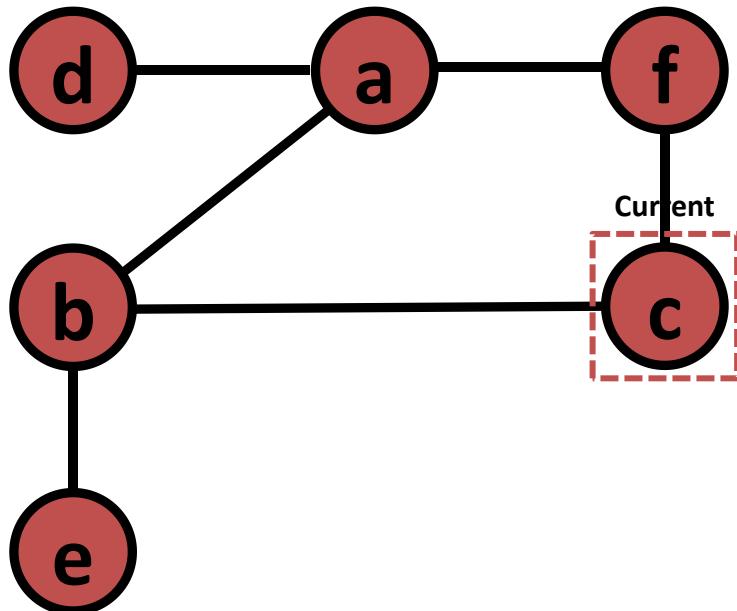
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Breadth-First Traversal For G

Current Vertex	c
Already Visited Vertices	
a, b, d, f, c, e	
Adjacency List for Current Vertex	
c	b → f [All visited now!]
Vertex Queue (Front to Rear)	
e	[dequeue()]

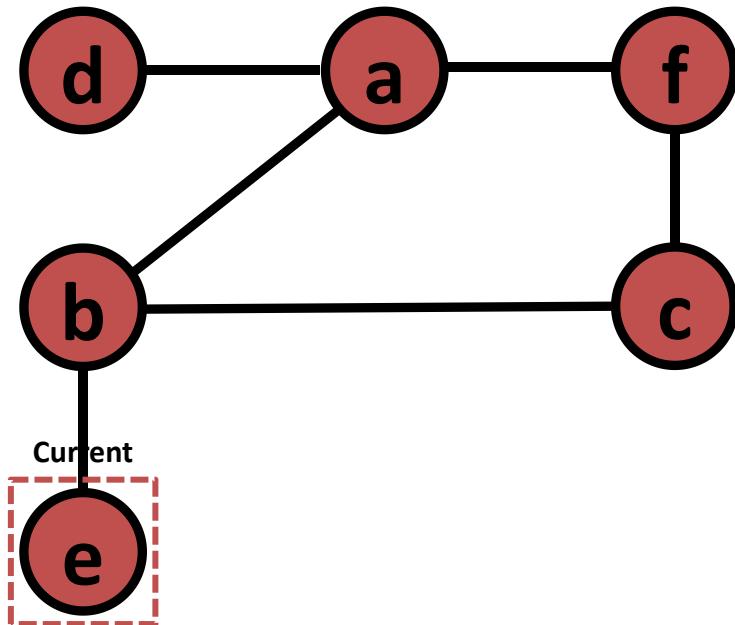
Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal

Graph G



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

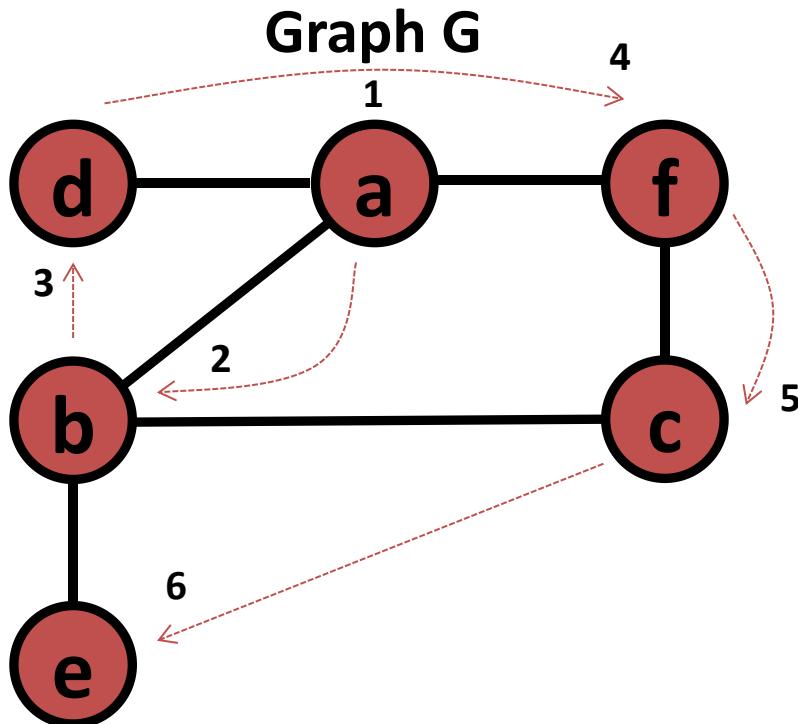
Current Vertex	e
Already Visited Vertices	
a, b, d, f, c, e	
Adjacency List for Current Vertex	
e	b [All visited now!]
Vertex Queue (Front to Rear)	
[empty - Done!]	

Rule 1: Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue

Rule 2: If no unvisited (adjacent to current) vertices, current vertex = dequeue()

Rule 3: If queue is empty, you are done

Graph: Breadth First Traversal



Already visited vertex



Not visited yet vertex

Breadth First Traversal For G

Current Vertex	
Already Visited Vertices	a, b, d, f, c, e
Adjacency List for Current Vertex	
Vertex Queue (Front to Rear)	

- Rule 1:** Visit next unvisited vertex adjacent to current vertex, mark it, add to the queue
- Rule 2:** If no unvisited (adjacent to current) vertices, current vertex = dequeue()
- Rule 3:** If queue is empty, you are done

Graph Traversals: Time Complexity

General Depth-First Traversal (iterative)

```
dfs(Graph G, vertex v) //v start point
    initialize stack S
    S.push(v)
    mark v as visited
    while (S is not empty)
        m = S.pop()
        for all (m, k) edges
            if k is not visited
                S.push(k)
                mark k as visited
        end
    end
end
```

General Breadth-First Traversal (iterative)

```
bfs(Graph G, vertex v) //v start point
    initialize queue Q
    S.enqueue(v)
    mark v as visited
    while (Q is not empty)
        m = Q.dequeue()
        for all (m, k) edges
            if k is not visited
                Q.enqueue(k)
                mark k as visited
        end
    end
end
```

Sequence of exploration (BFS):

$v_1 + (\text{edges incident on } v_1) + v_2 + (\text{edges incident on } v_2) + \dots + v_n + (\text{edges incident on } v_n)$

$[v_1 + v_2 + \dots + v_n] + [(\text{edges incident on } v_1) + (\text{edges incident on } v_2) + \dots + (\text{edges incident on } v_n)]$

which is:

$|V| + \text{at most } 2*|E|$ (each vertex is visited once, each edge is visited at most twice)

Big-O notation (assuming adjacency list representation):

$O(|V| + |E|)$