

# CSE214 – Analysis of Algorithms

PhD Furkan Gözükar, Toros University

<https://github.com/FurkanGozukara/CSE214> 2018

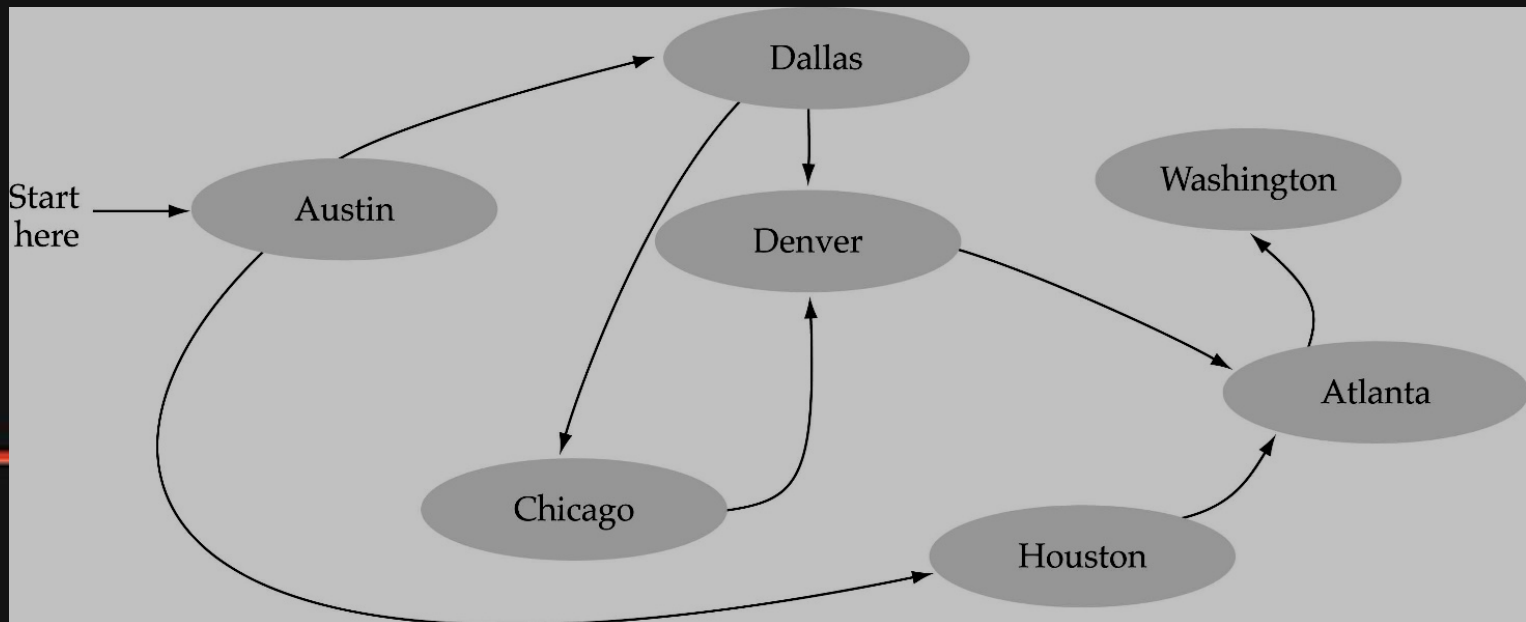
## Lecture 10

### Graphs

*Based George Bebis Lecture Notes - Reno Logo  
University of Nevada*

# What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



# Formal definition of graphs

- A graph  $G$  is defined as follows:

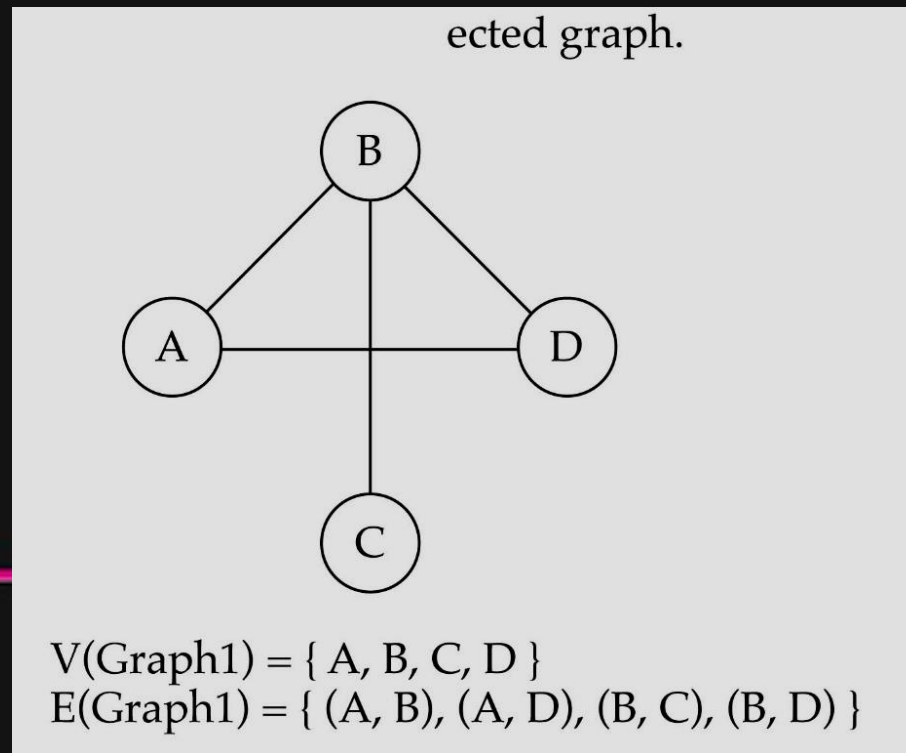
$$G=(V,E)$$

$V(G)$ : a finite, nonempty set of vertices

$E(G)$ : a set of edges (pairs of vertices)

# Directed vs. undirected graphs

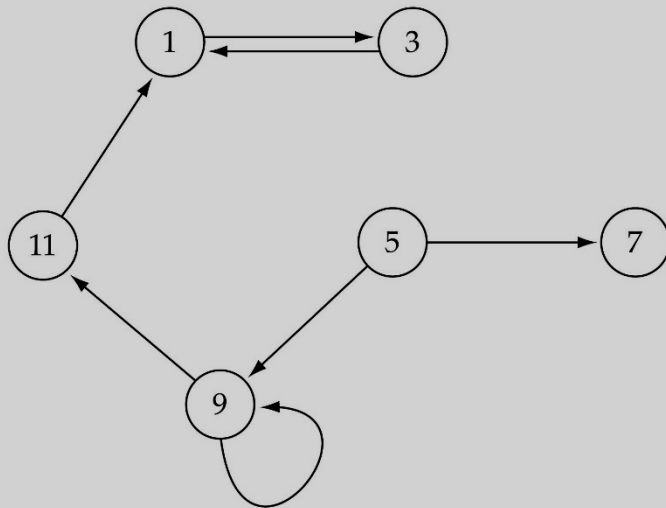
- When the edges in a graph have no direction, the graph is called *undirected*



# Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



$V(\text{Graph2}) = \{ 1, 3, 5, 7, 9, 11 \}$

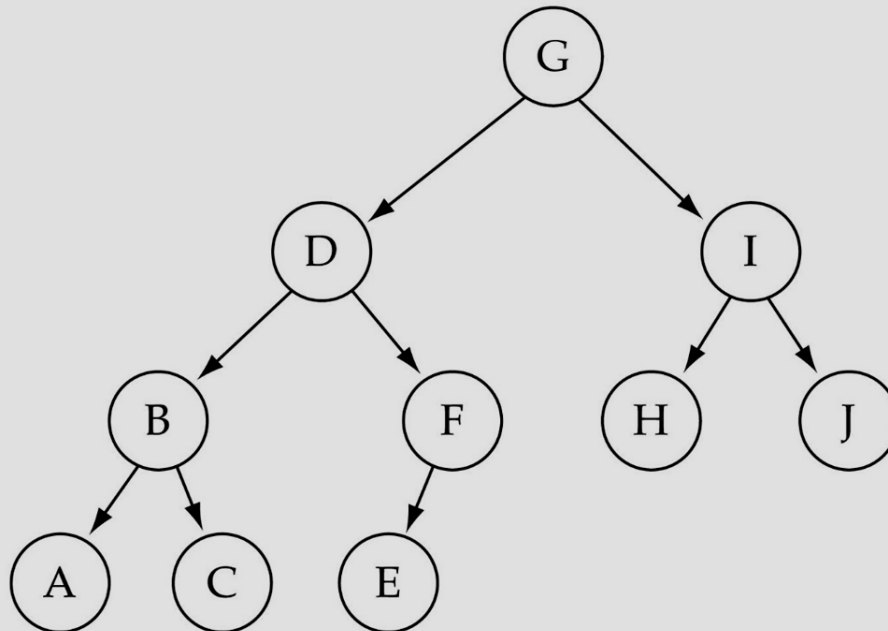
$E(\text{Graph2}) = \{(1,3) (3,1) (5,9) (9,11) (5,7) , (9, 9), (11, 1) \}$

*Warning:* if the graph is directed, the order of the vertices in each edge is important !!

# Trees vs graphs

- Trees are special cases of graphs!!

(c) Graph3 is a directed graph.

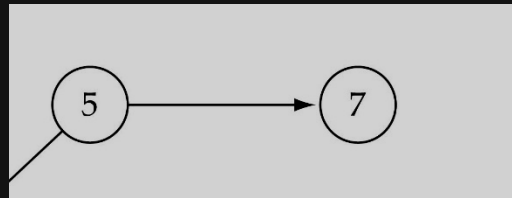


$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$

$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$

# Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



5 is adjacent **to** 7  
7 is adjacent **from** 5

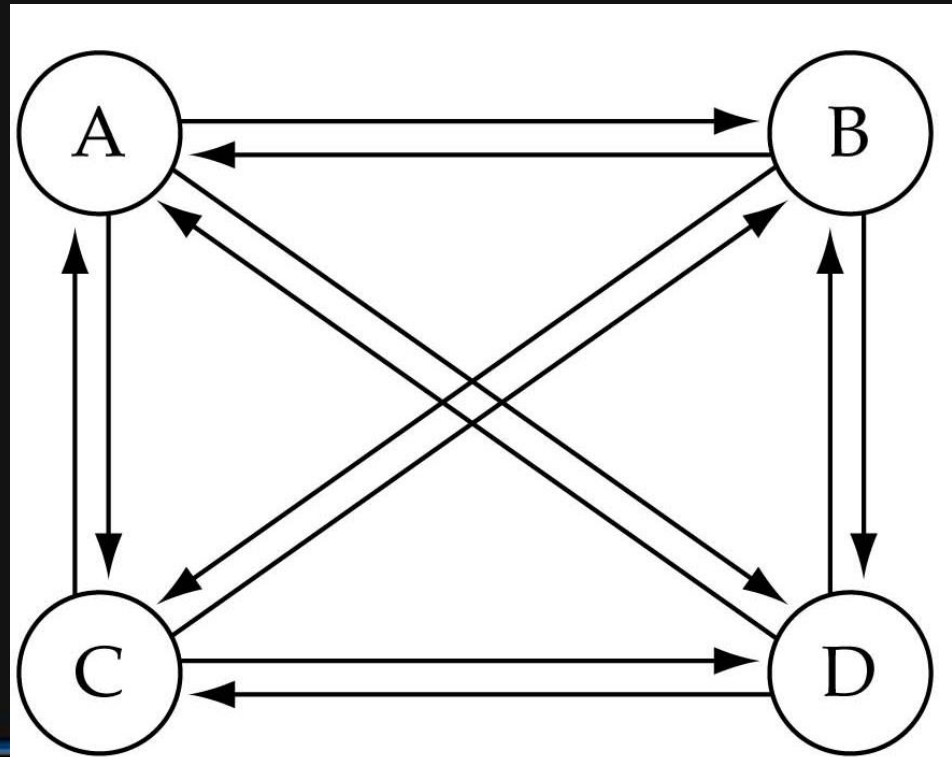
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

# Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

$$N * (N-1)$$

$$O(N^2)$$



(a) Complete directed graph.

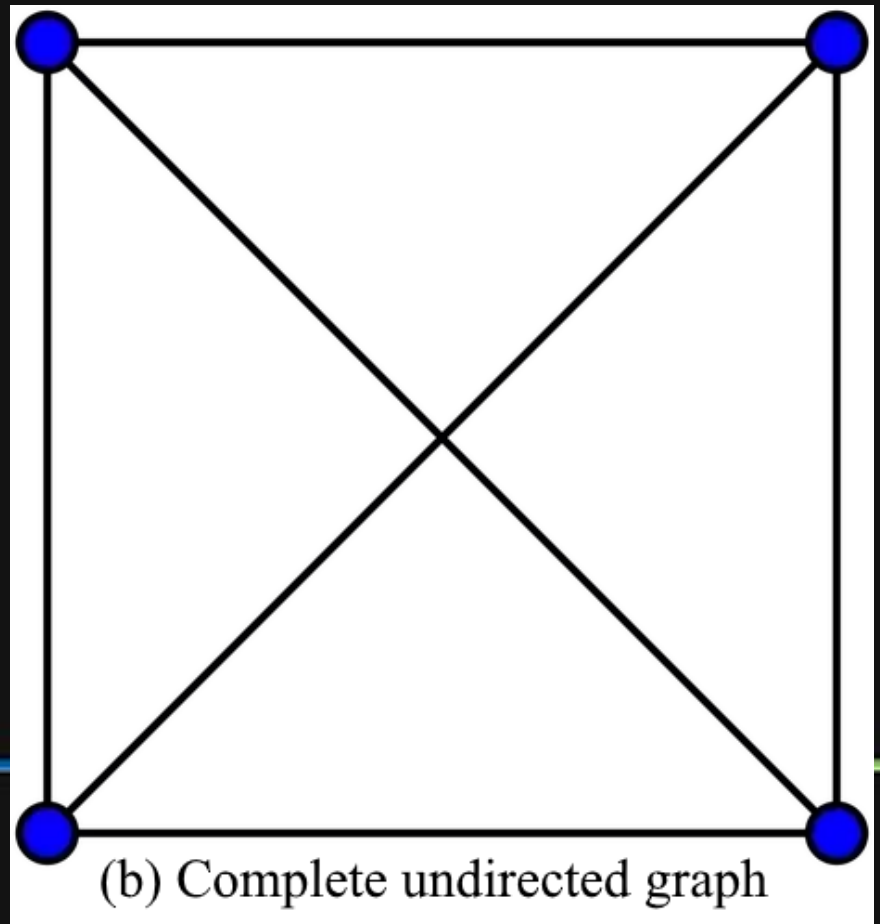


# Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

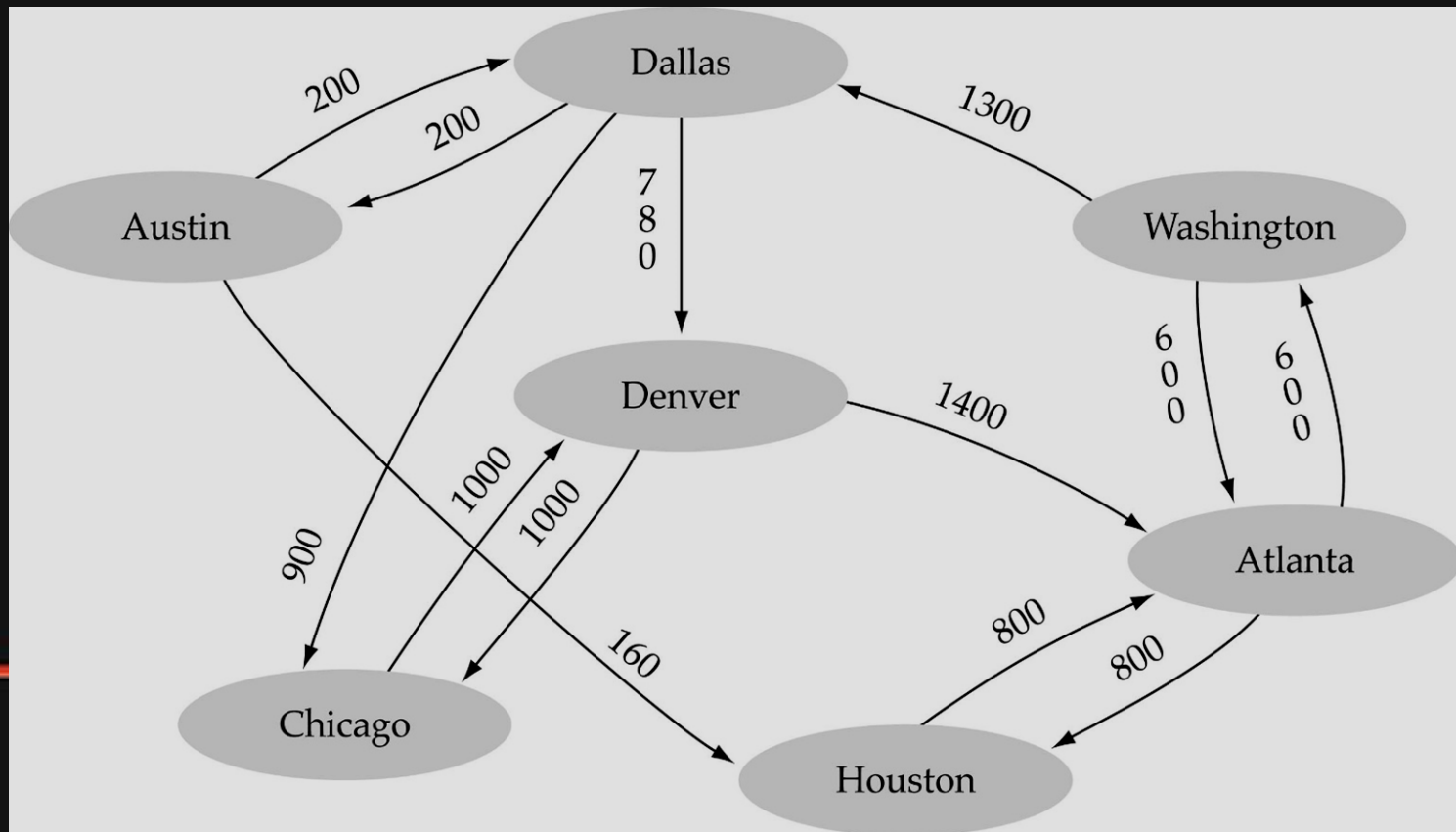
$$N * (N-1) / 2$$

$$O(N^2)$$



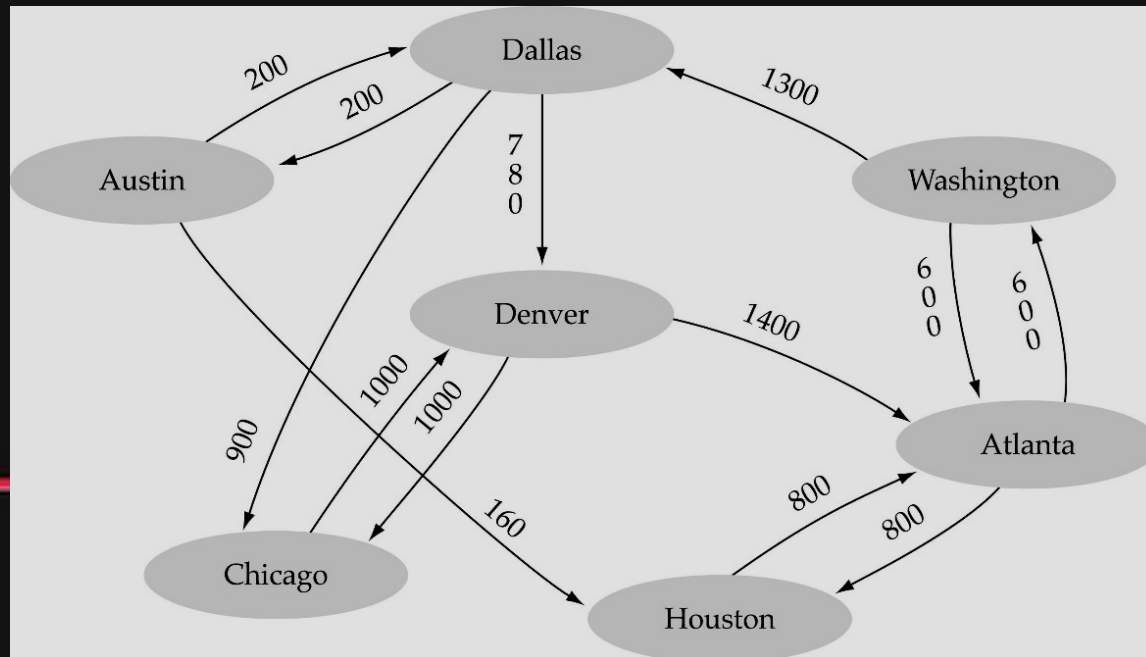
# Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



# Graph implementation

- Array-based implementation
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges



# Array-based implementation

graph

.numVertices 7

.vertices

|     |              |
|-----|--------------|
| [0] | "Atlanta"    |
| [1] | "Austin"     |
| [2] | "Chicago"    |
| [3] | "Dallas"     |
| [4] | "Denver"     |
| [5] | "Houston"    |
| [6] | "Washington" |
| [7] |              |
| [8] |              |
| [9] |              |

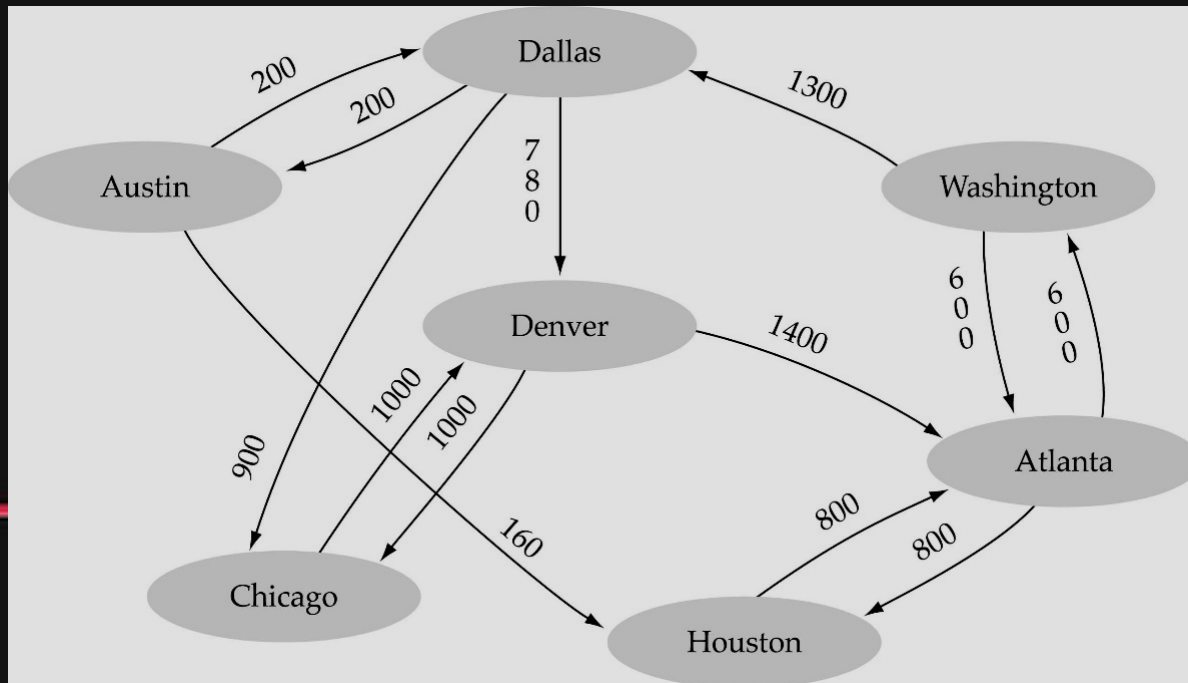
.edges

|     |      |     |      |      |      |     |     |     |     |     |
|-----|------|-----|------|------|------|-----|-----|-----|-----|-----|
| [0] | 0    | 0   | 0    | 0    | 0    | 800 | 600 | •   | •   | •   |
| [1] | 0    | 0   | 0    | 200  | 0    | 160 | 0   | •   | •   | •   |
| [2] | 0    | 0   | 0    | 0    | 1000 | 0   | 0   | •   | •   | •   |
| [3] | 0    | 200 | 900  | 0    | 780  | 0   | 0   | •   | •   | •   |
| [4] | 1400 | 0   | 1000 | 0    | 0    | 0   | 0   | •   | •   | •   |
| [5] | 800  | 0   | 0    | 0    | 0    | 0   | 0   | •   | •   | •   |
| [6] | 600  | 0   | 0    | 1300 | 0    | 0   | 0   | •   | •   | •   |
| [7] | •    | •   | •    | •    | •    | •   | •   | •   | •   | •   |
| [8] | •    | •   | •    | •    | •    | •   | •   | •   | •   | •   |
| [9] | •    | •   | •    | •    | •    | •   | •   | •   | •   | •   |
|     | [0]  | [1] | [2]  | [3]  | [4]  | [5] | [6] | [7] | [8] | [9] |

(Array positions marked '•' are undefined)

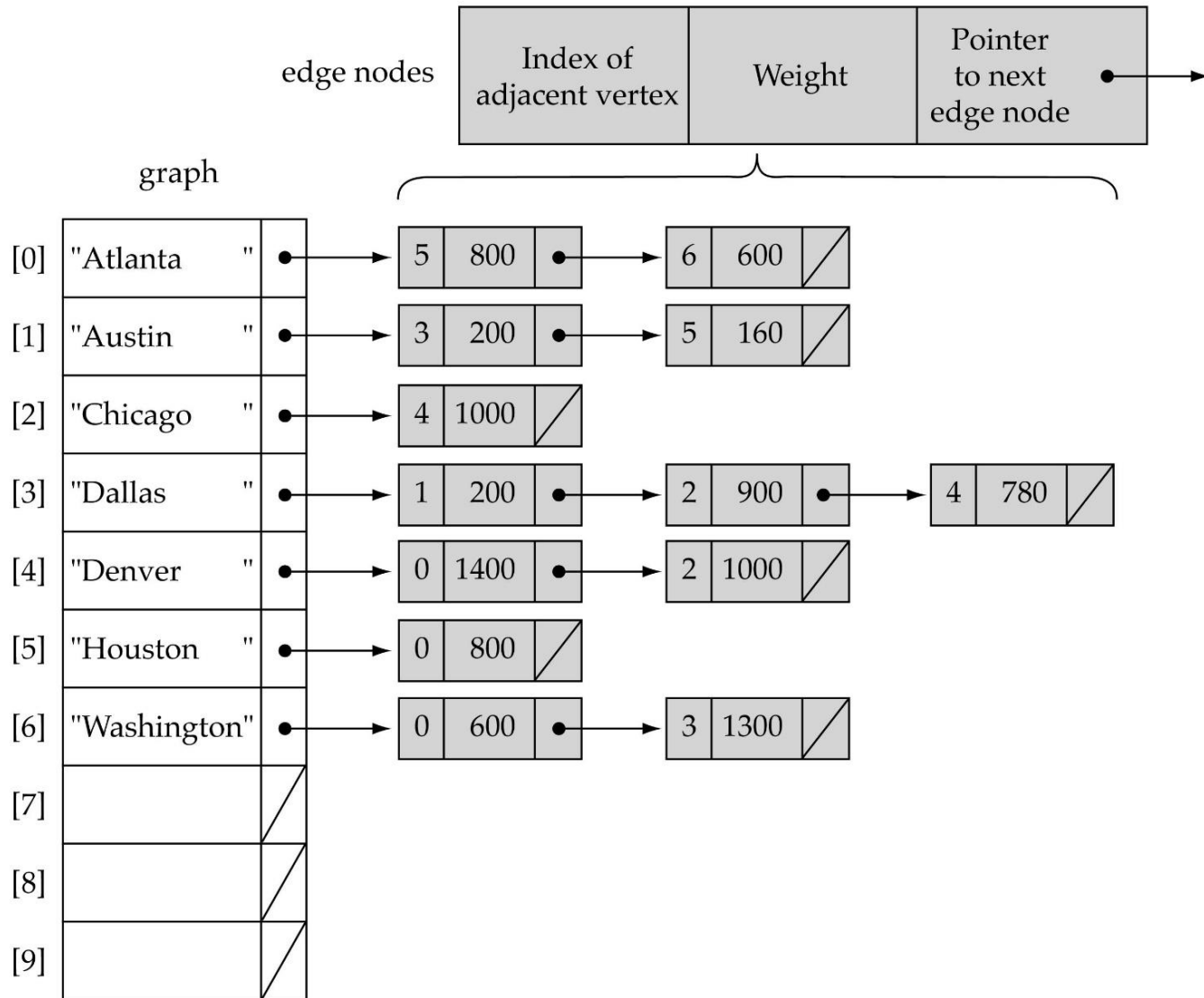
# Graph implementation (cont.)

- Linked-list implementation
  - A 1D array is used to represent the vertices
  - A list is used for each vertex  $v$  which contains the vertices which are adjacent from  $v$  (adjacency list)



# Linked-list implementation

(a)



# Adjacency matrix vs. adjacency list representation


- **Adjacency matrix**

- Good for dense graphs --  $|E| \sim O(|V|^2)$
- Memory requirements:  $O(|V| + |E|) = O(|V|^2)$
- Connectivity between two vertices can be tested quickly

- **Adjacency list**

- Good for sparse graphs --  $|E| \sim O(|V|)$
- Memory requirements:  $O(|V| + |E|) = O(|V|)$
- Vertices adjacent to another vertex can be found quickly

# Graph searching

- Problem: find a path between two nodes of the graph (e.g., Austin and Washington)
  - Methods: Depth-First-Search (DFS) or Breadth-First-Search (BFS)
- 



# Depth-First-Search (DFS)

- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a *stack*

# What is a Stack?

- A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the *top*.
- The operations: push (insert) and pop (delete)



# Breadth-First-Searching (BFS)

- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

# Depth-First Graph Traversal Algorithm

Alyce Brady  
Kalamazoo College

# Search vs Traversal

- Tree Search: Look for a given node
  - stop when node found, even if not all nodes were visited
- Tree Traversal: Always visit all nodes

# Depth-first Search

- Similar to Depth-first Traversal of a Binary Tree
- Choose a starting vertex
- Do a depth-first search on each adjacent vertex

# Pseudo-Code for Depth-First Search

## depth-first-search

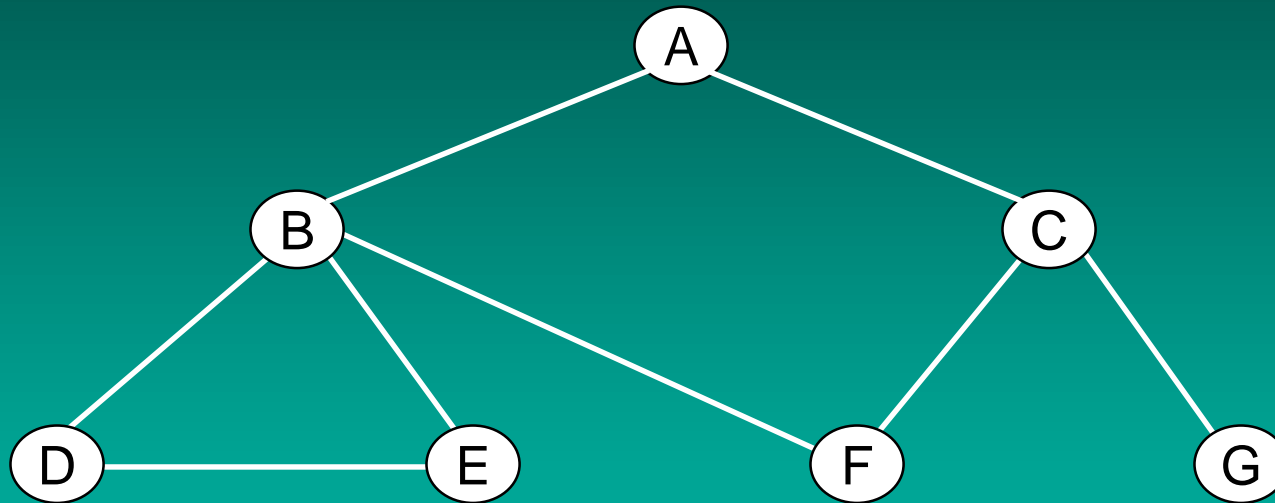
mark vertex as visited

for each adjacent vertex

if unvisited

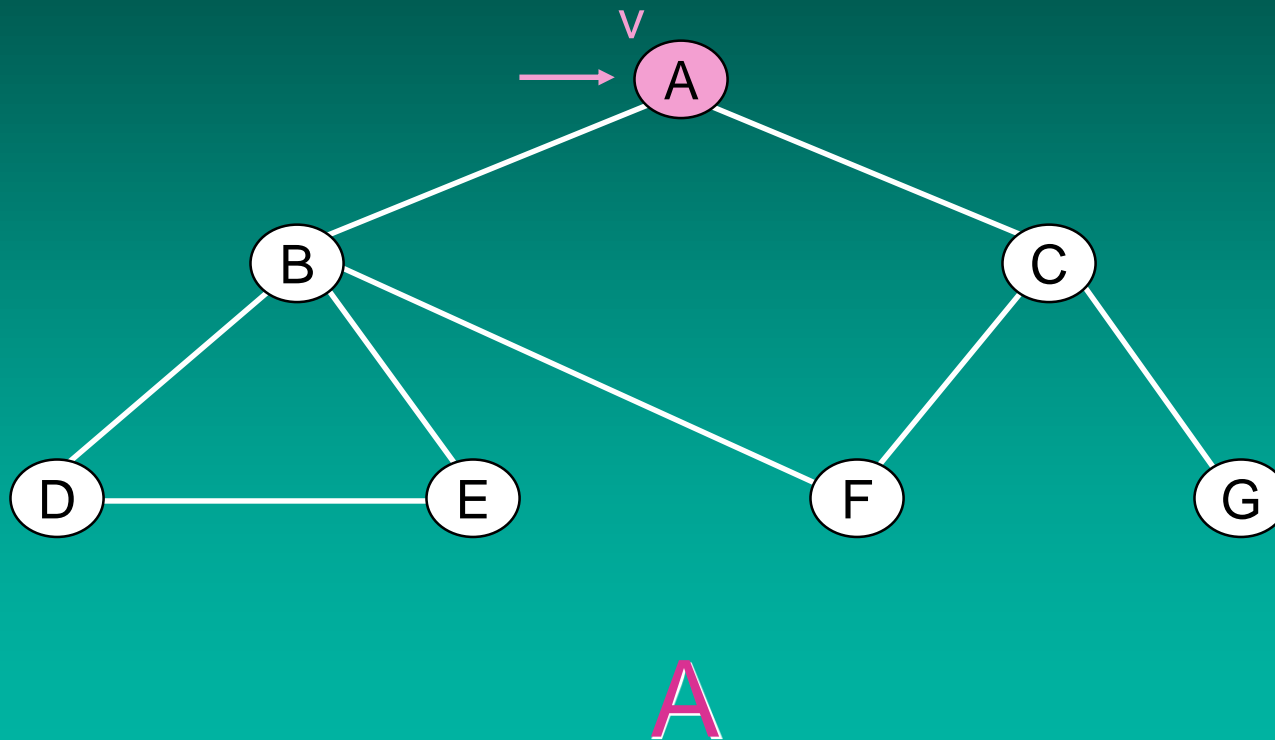
do a depth-first search on adjacent  
vertex

# Depth-First Search

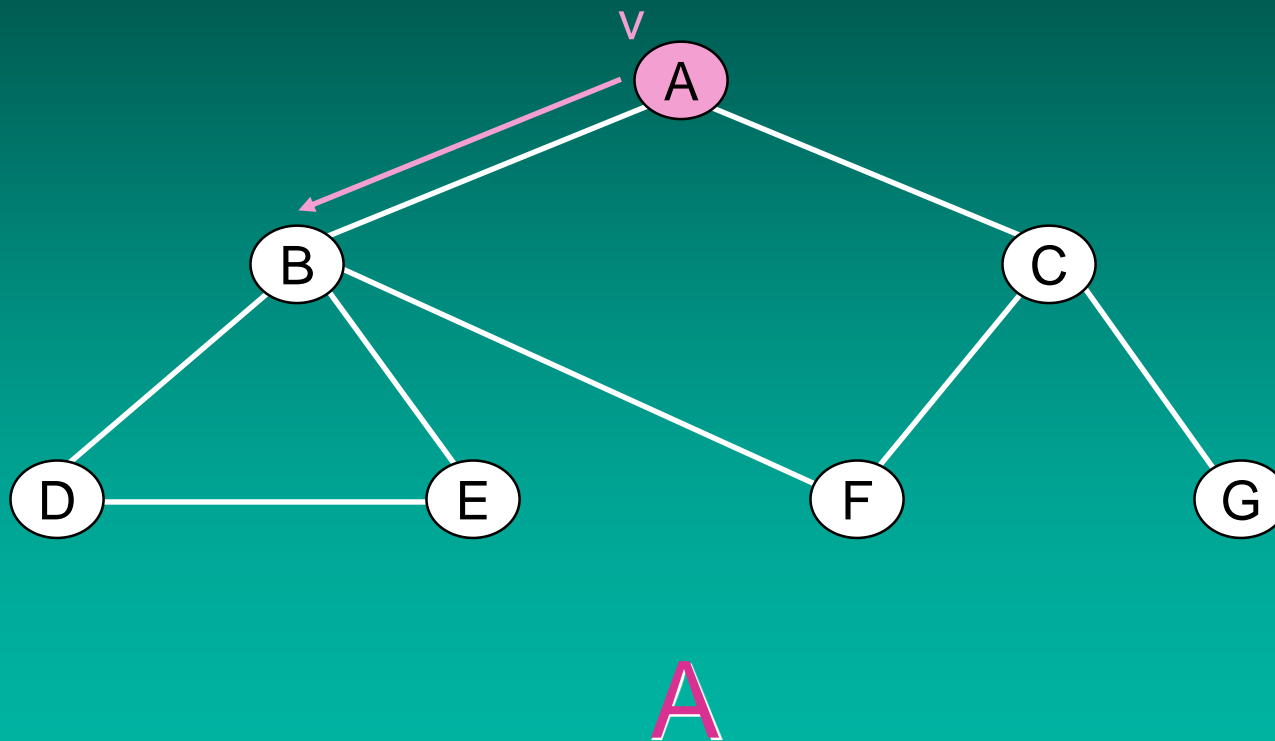




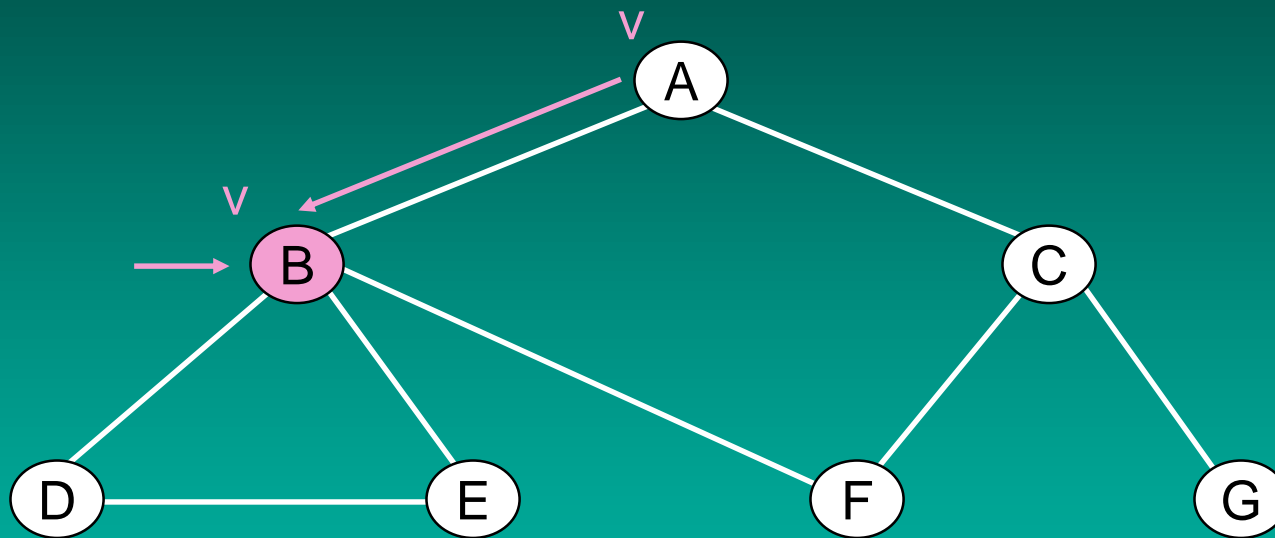
# Depth-First Search



# Depth-First Search

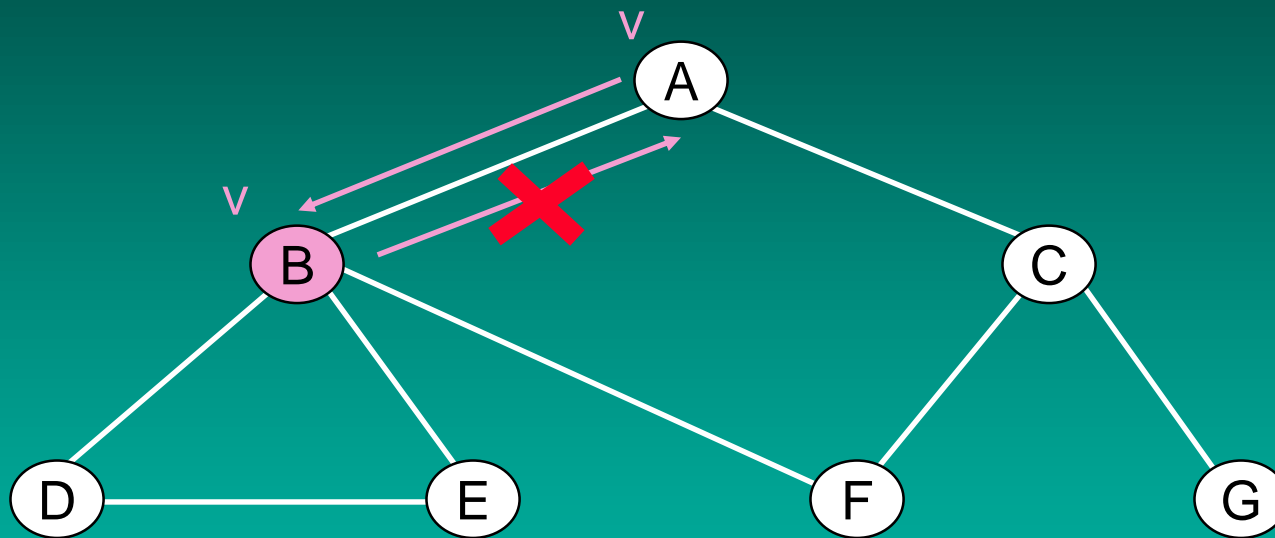


# Depth-First Search



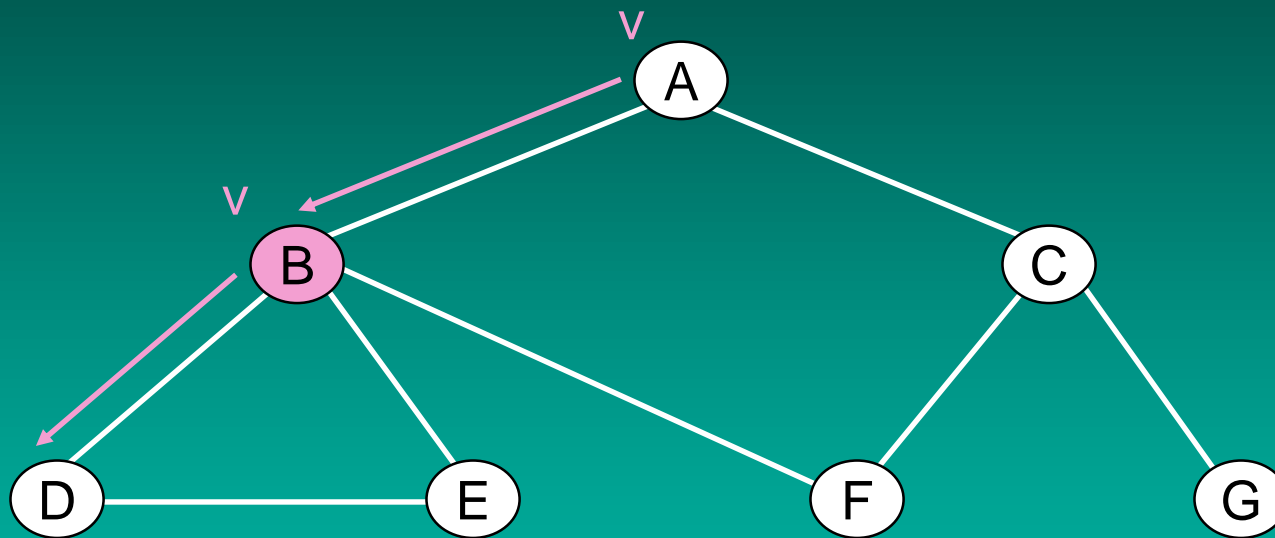
A B

# Depth-First Search



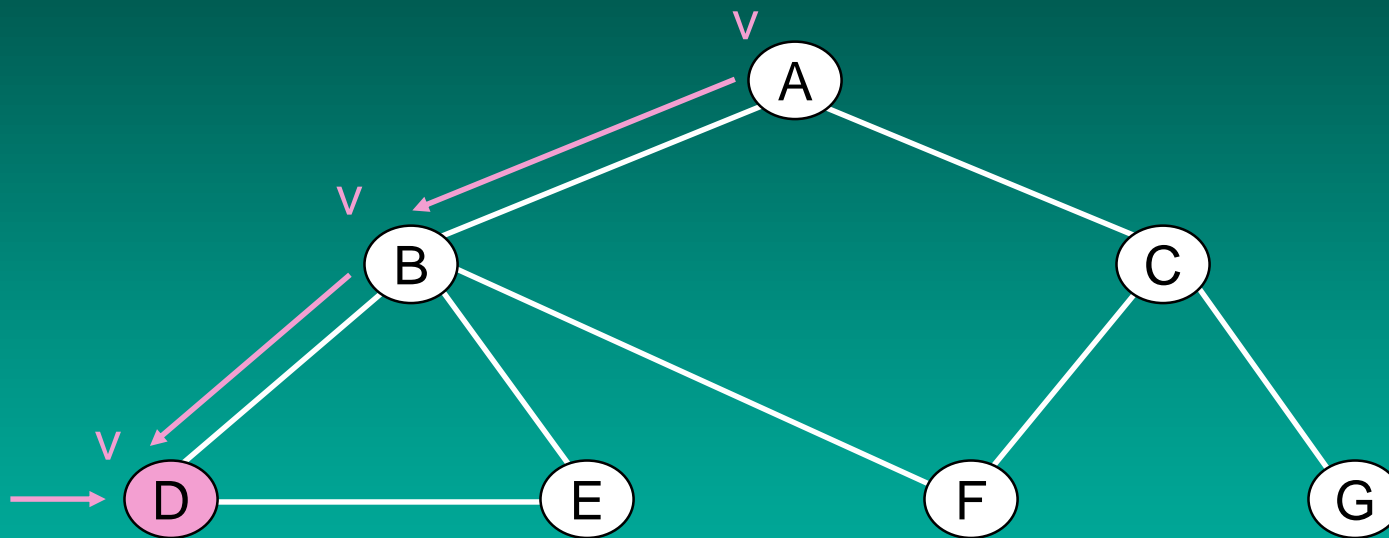
A B

# Depth-First Search



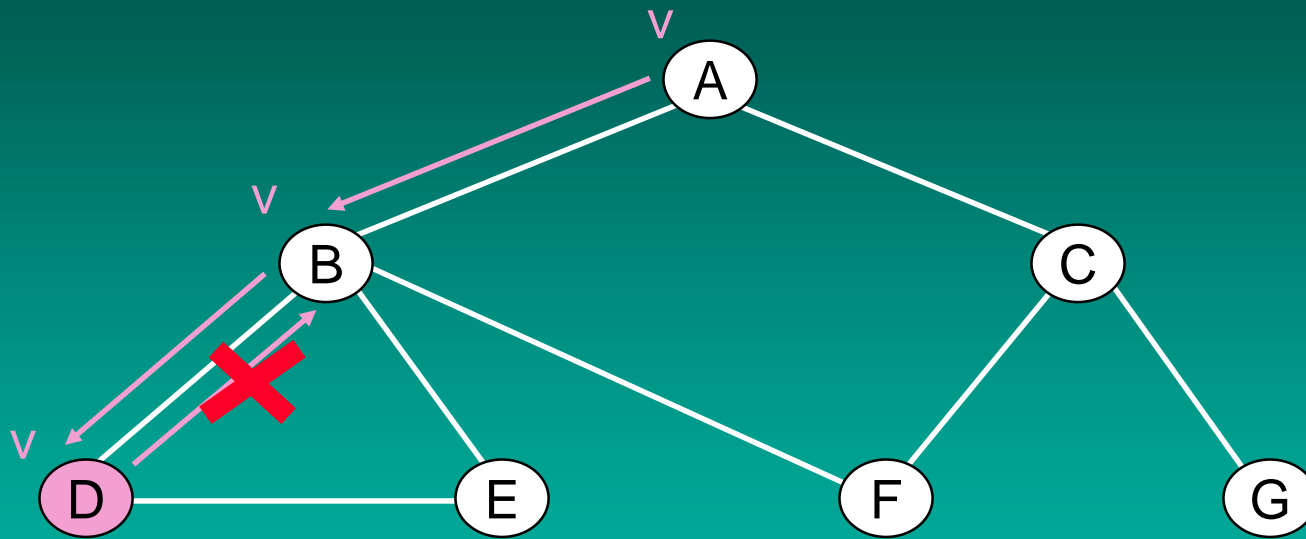
A B

# Depth-First Search



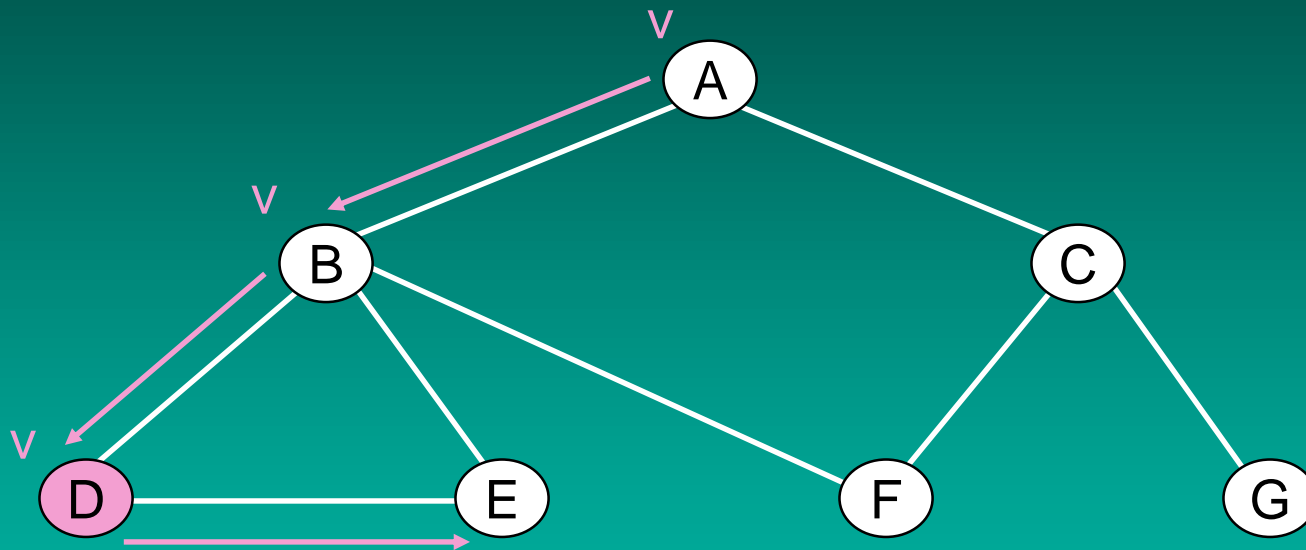
A B D

# Depth-First Search



A B D

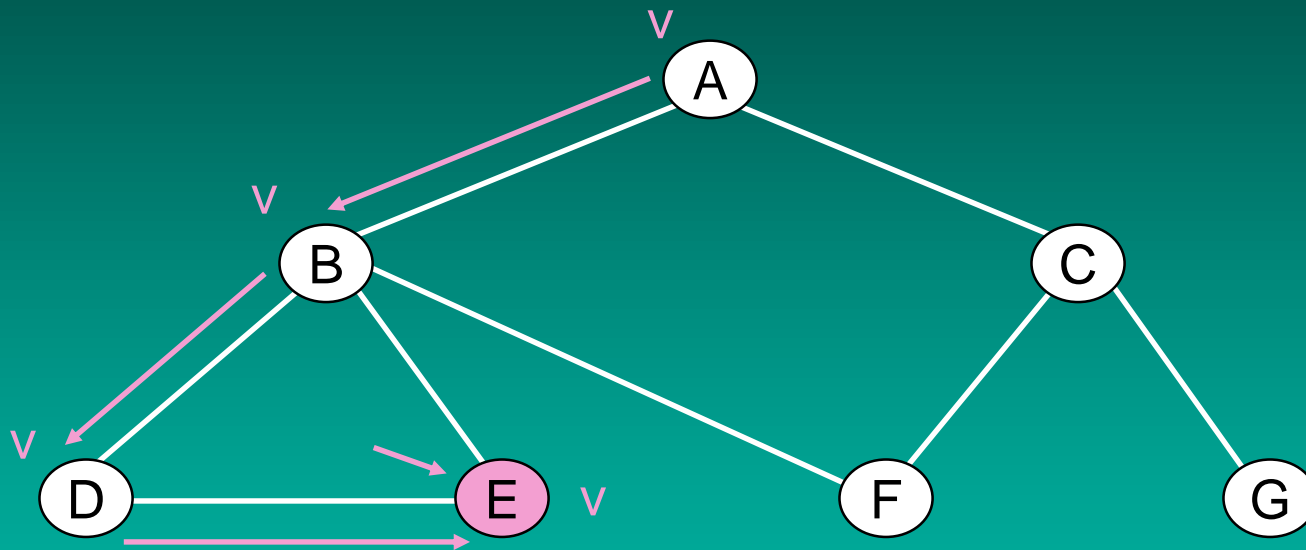
# Depth-First Search



A B D

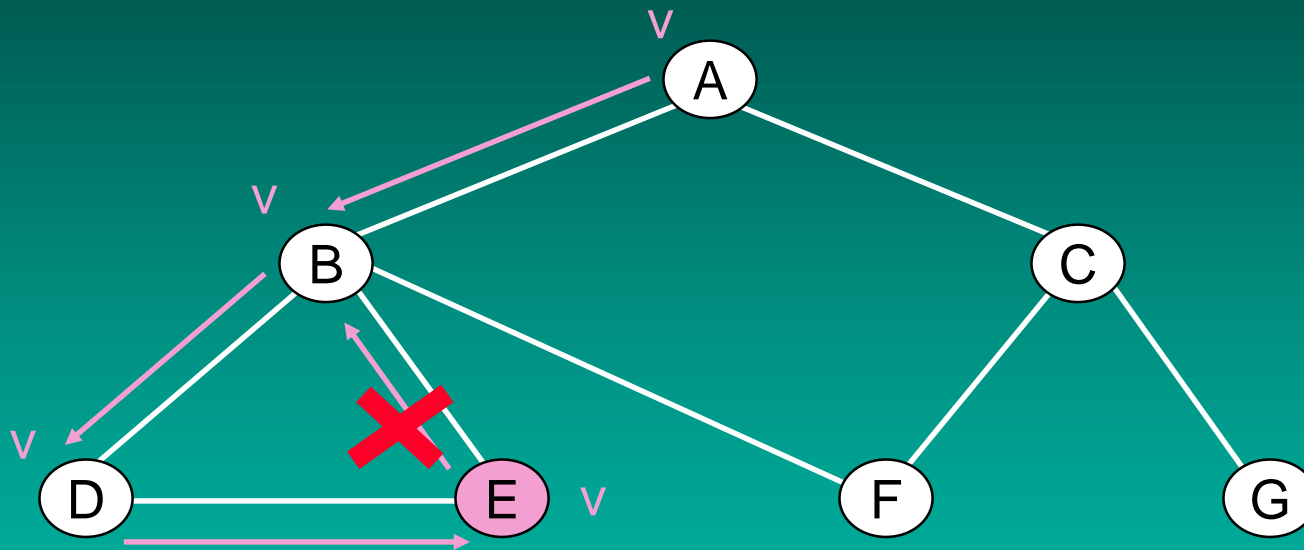


# Depth-First Search



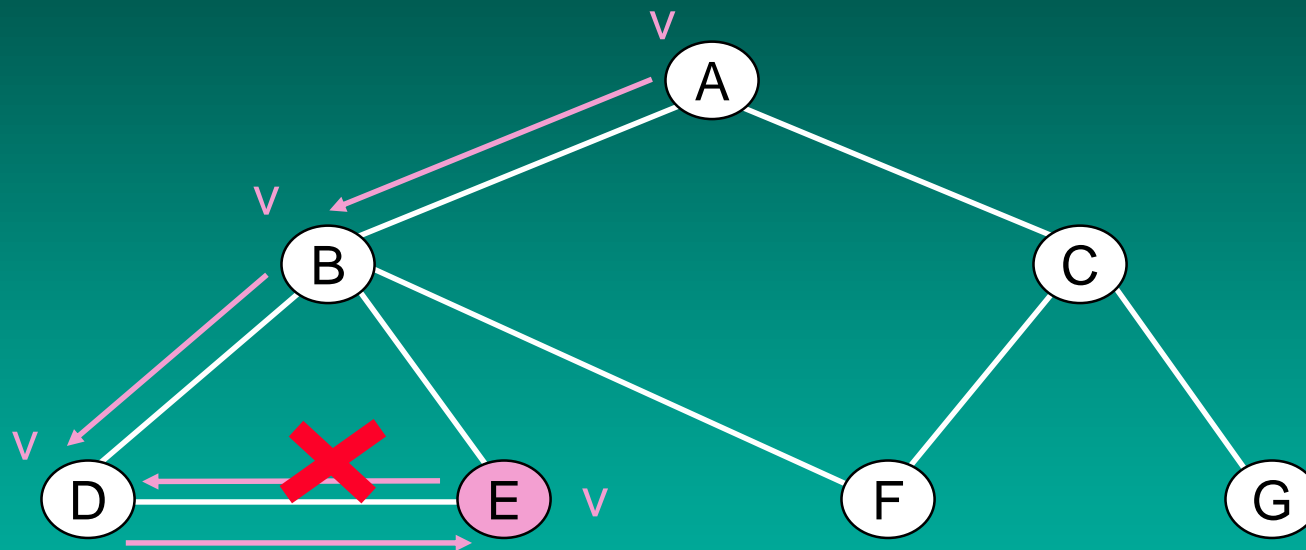
A B D E

# Depth-First Search



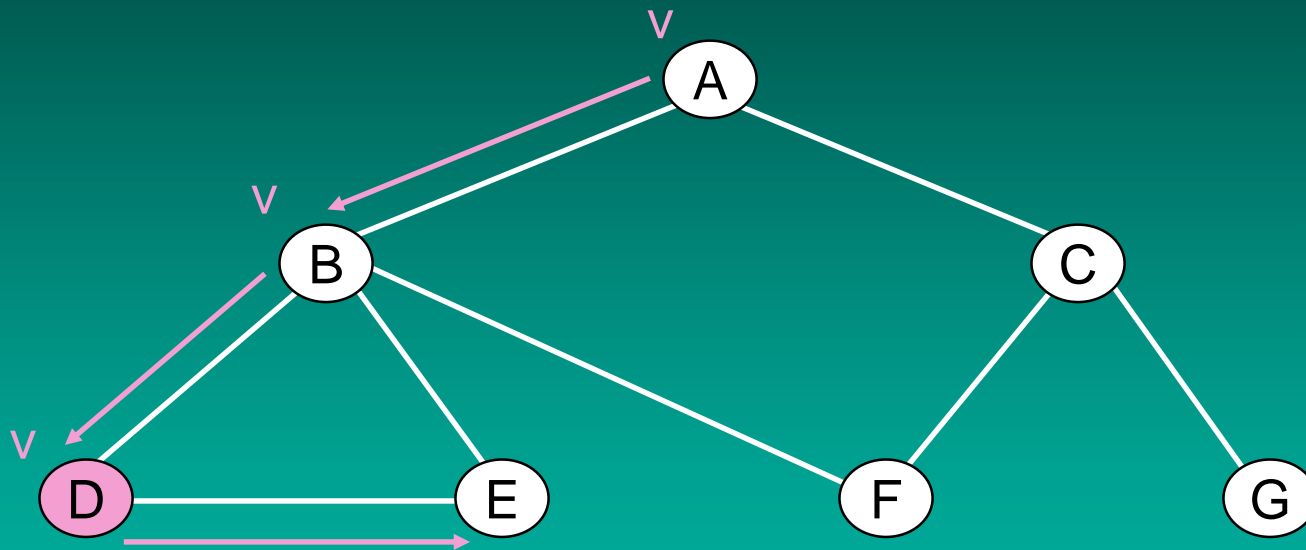
A B D E

# Depth-First Search



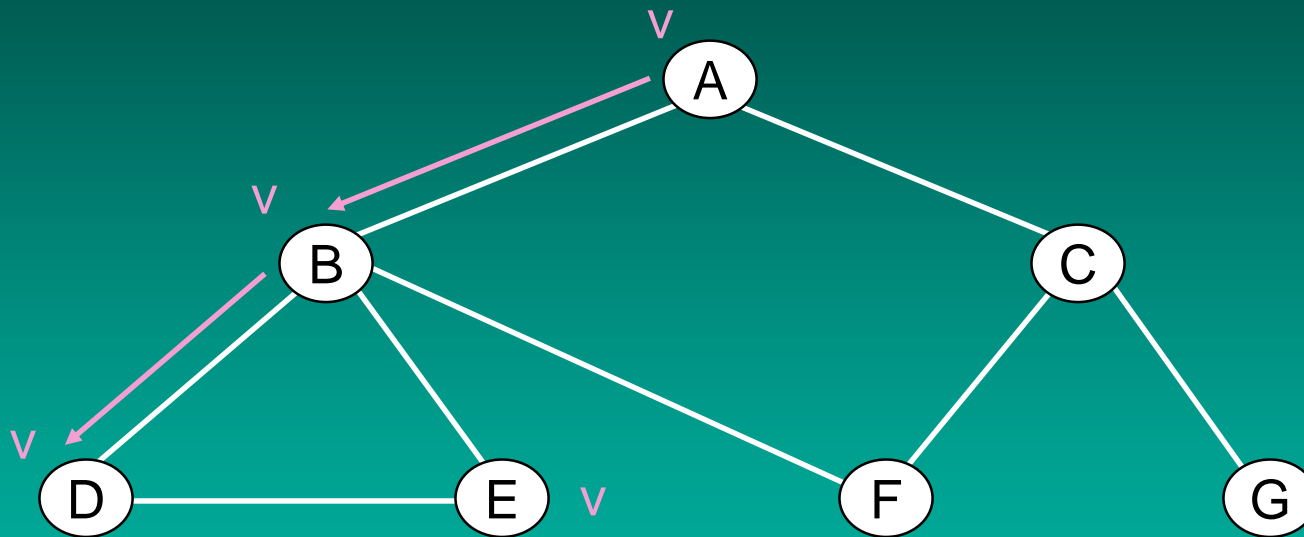
A B D E

# Depth-First Search



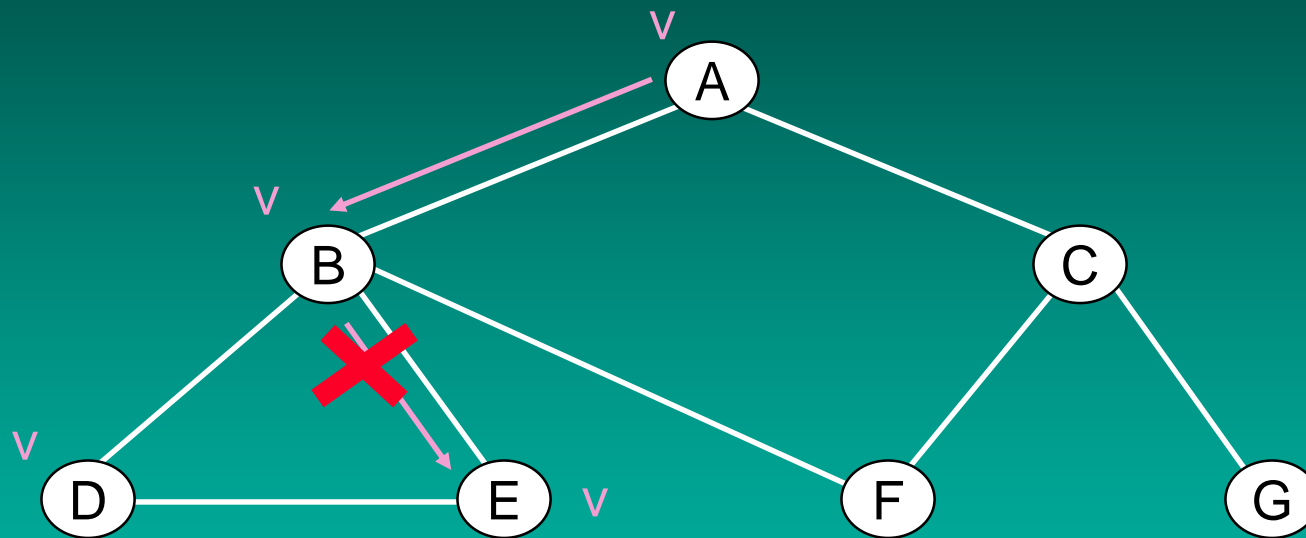
A B D E

# Depth-First Search



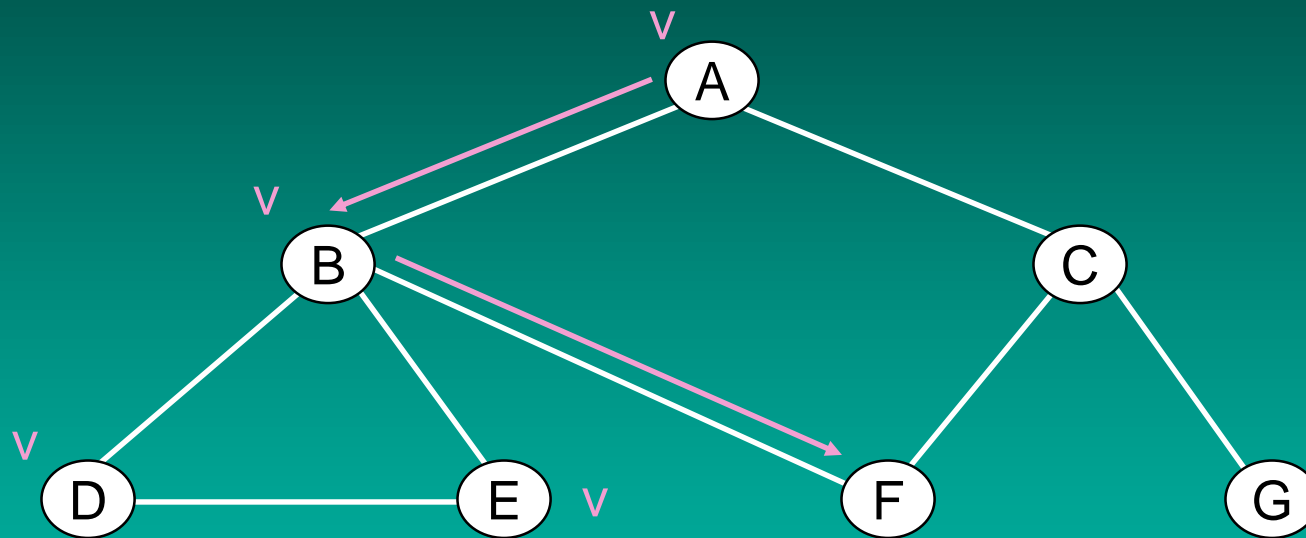
A B D E

# Depth-First Search



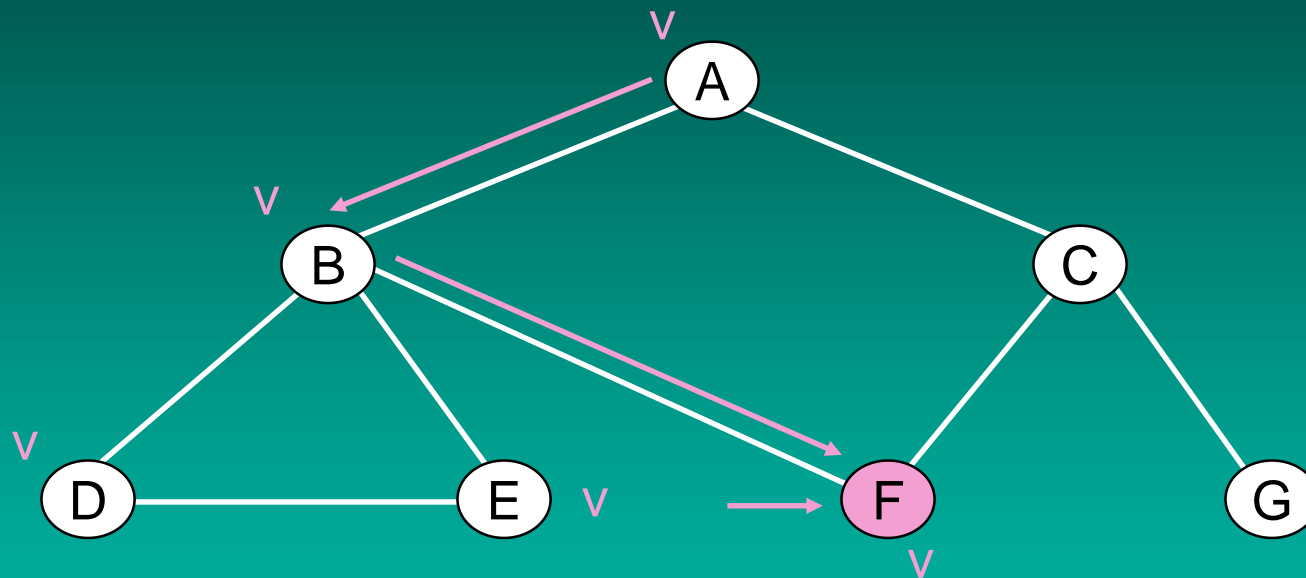
A B D E

# Depth-First Search



A B D E

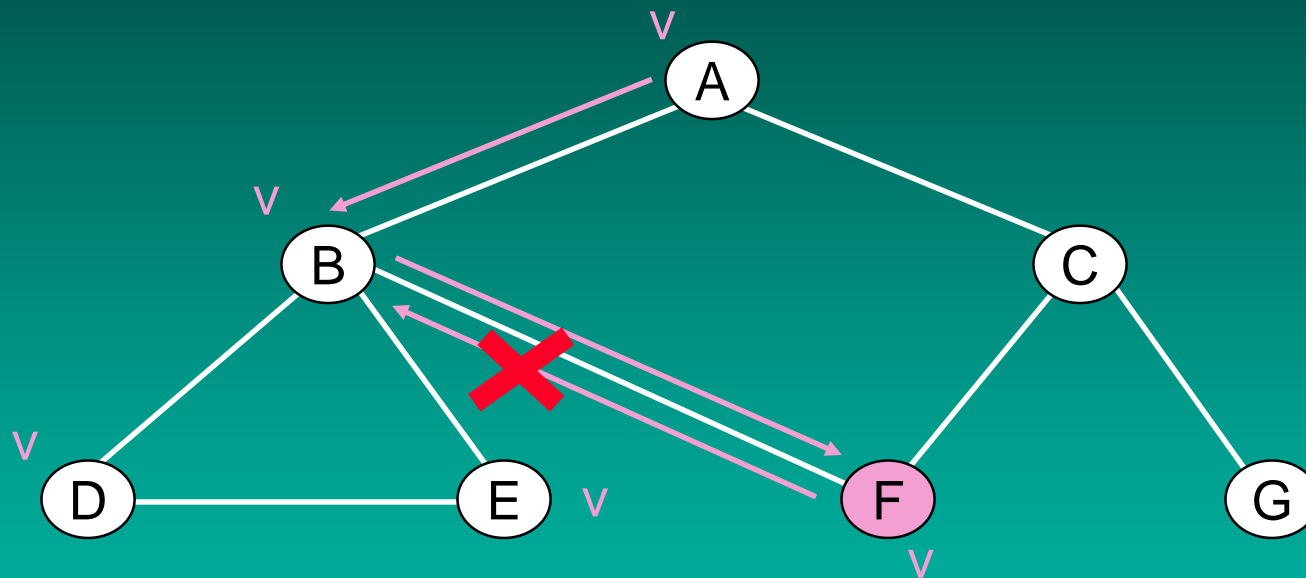
# Depth-First Search



A B D E F

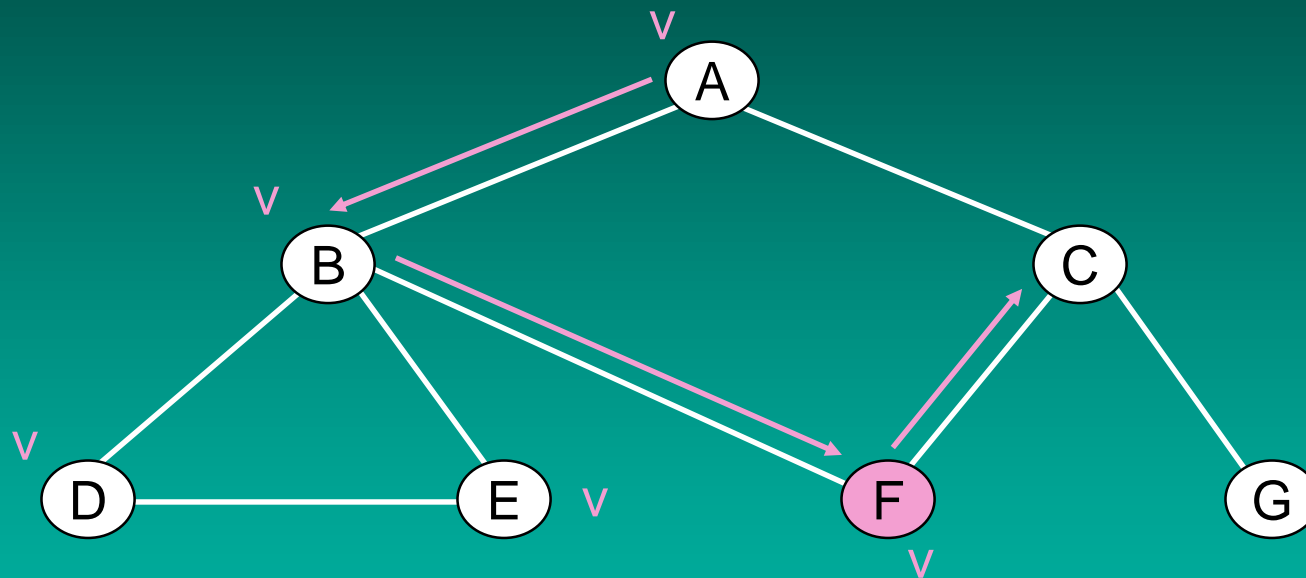


# Depth-First Search



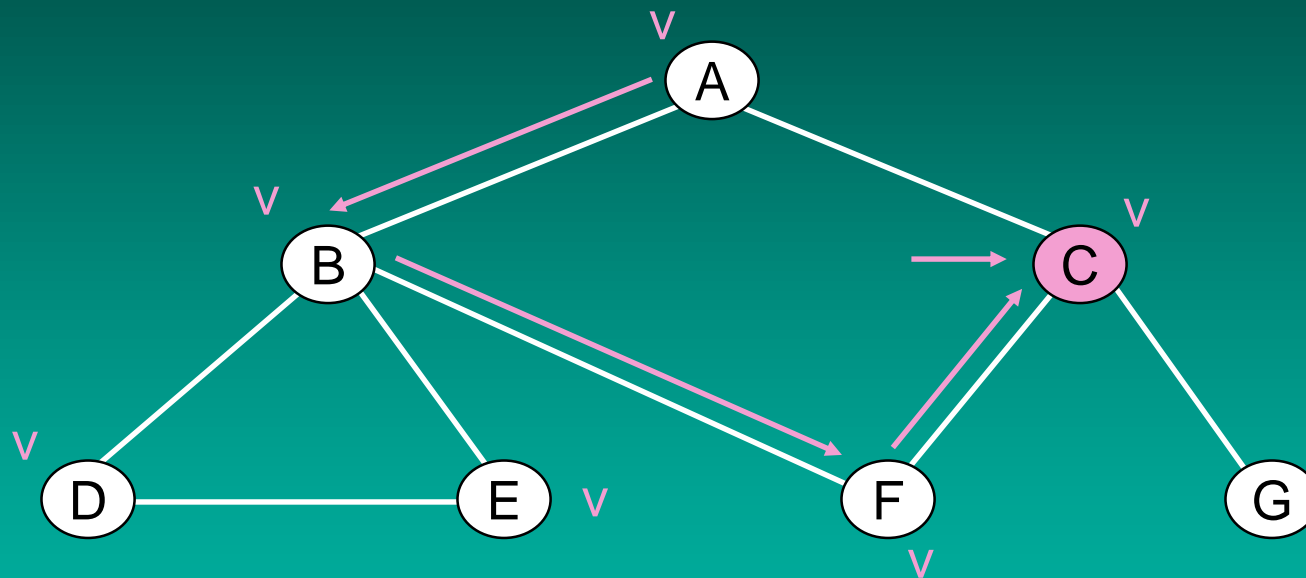
A B D E F

# Depth-First Search



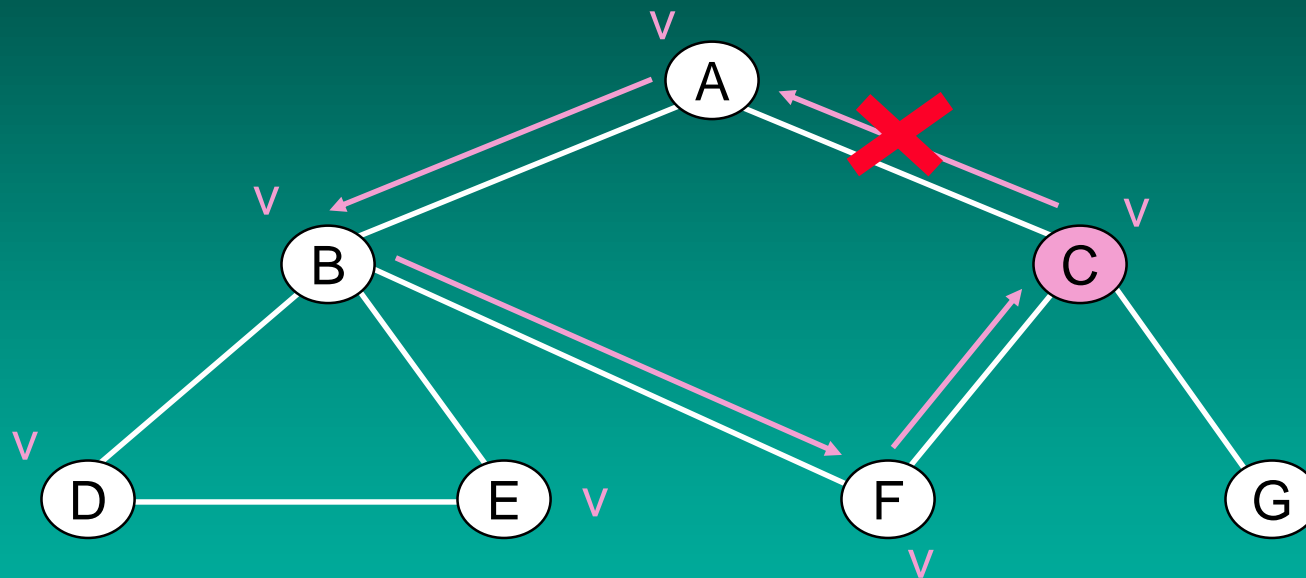
A B D E F

# Depth-First Search



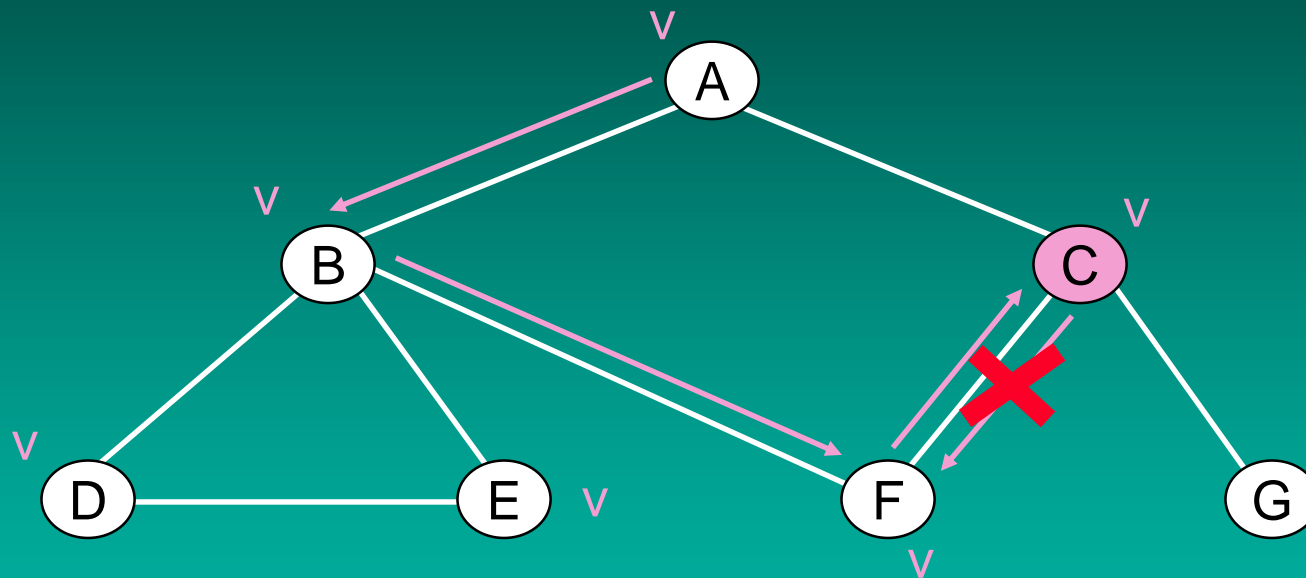
A B D E F C

# Depth-First Search



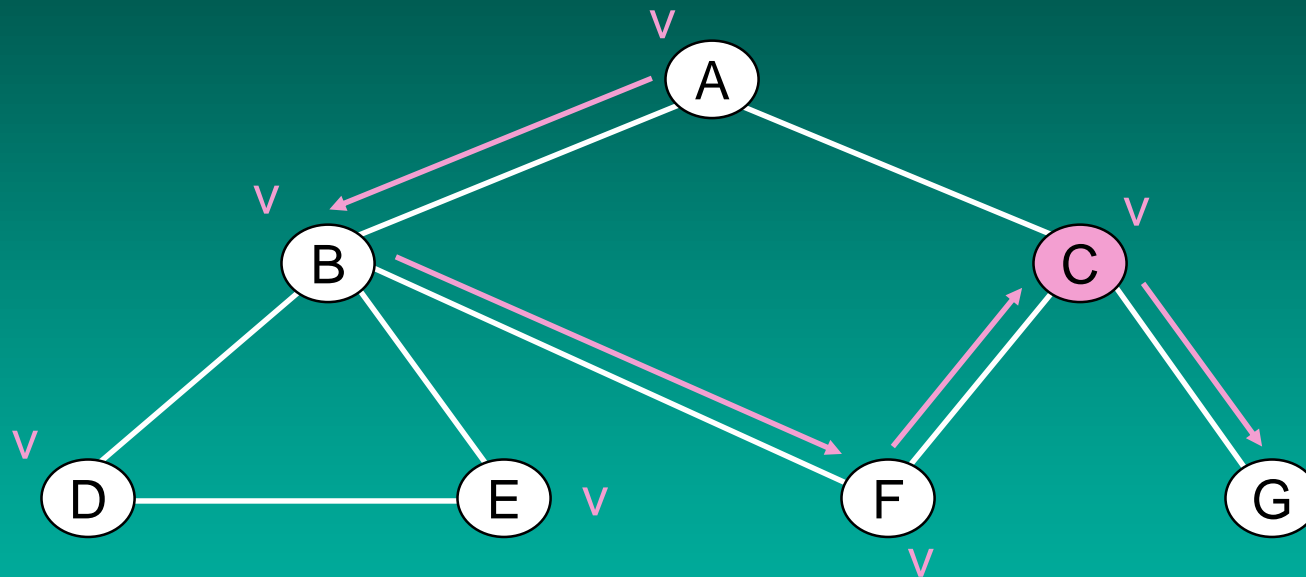
A B D E F C

# Depth-First Search



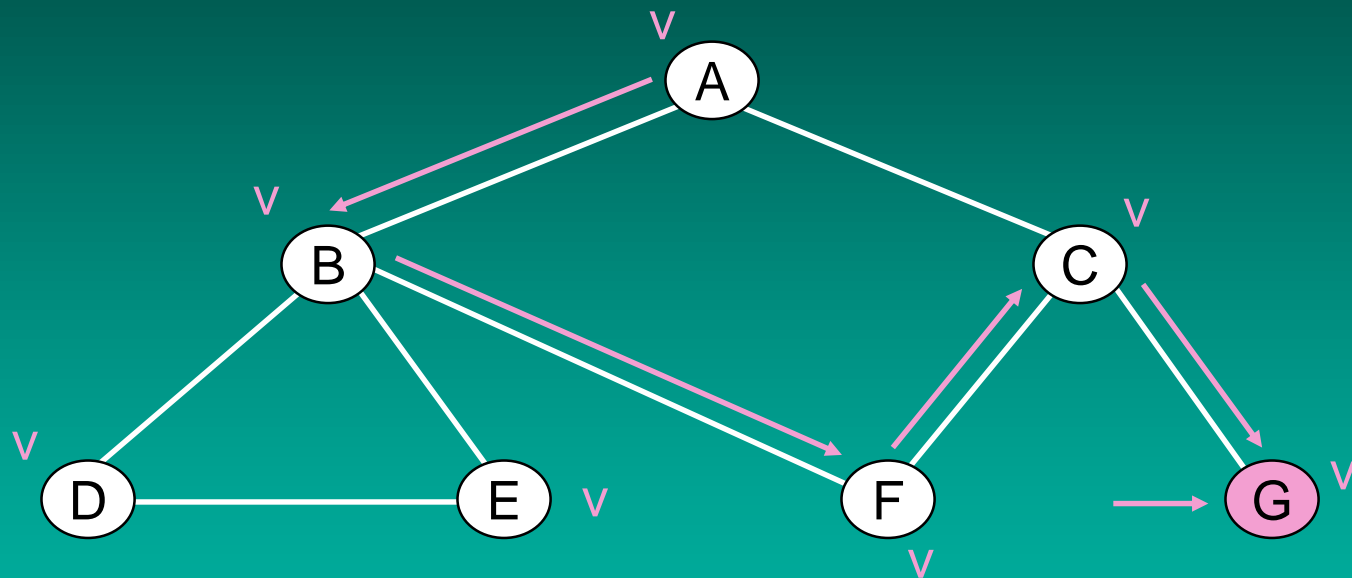
A B D E F C

# Depth-First Search



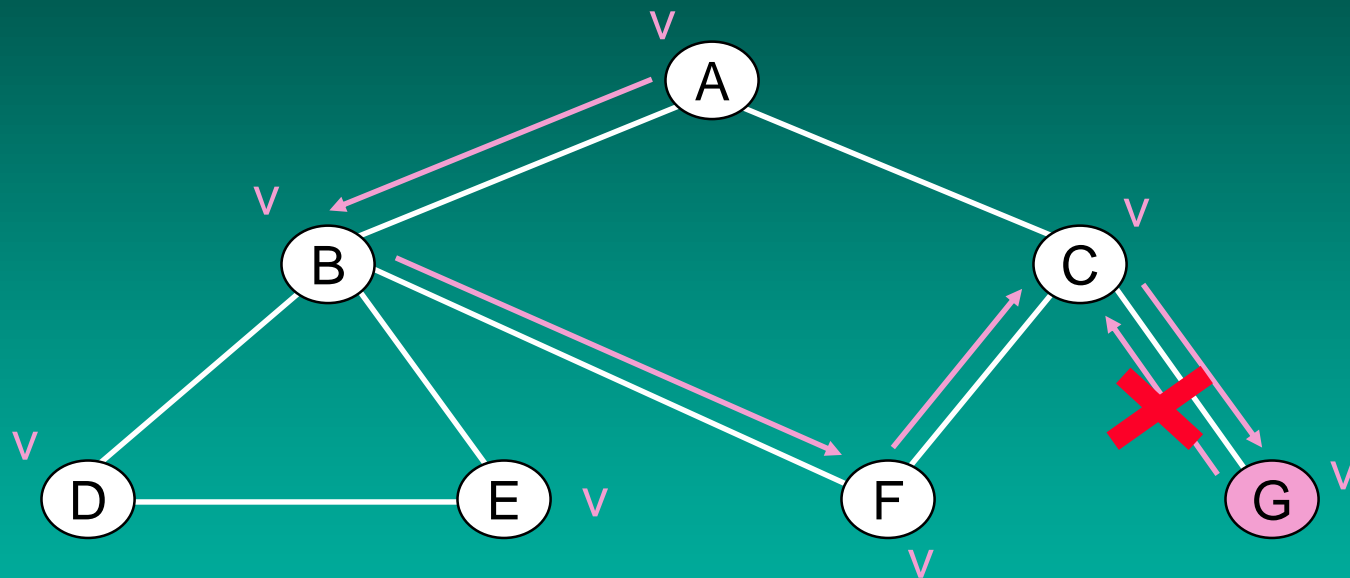
A B D E F C

# Depth-First Search



A B D E F C G

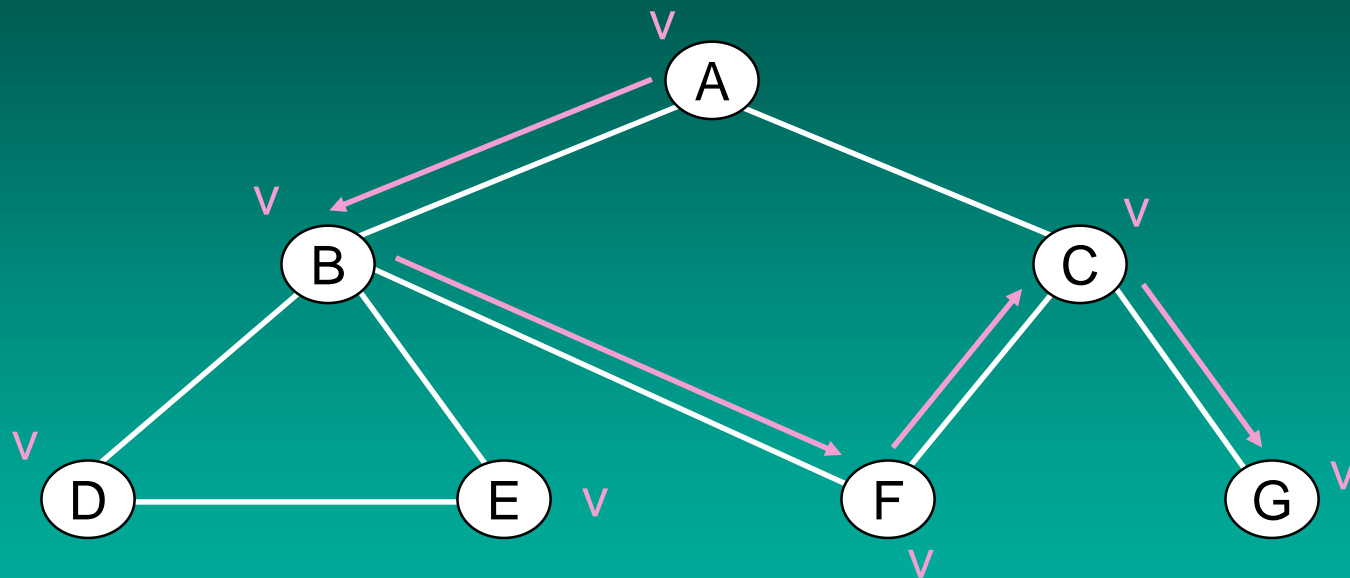
# Depth-First Search



A B D E F C G

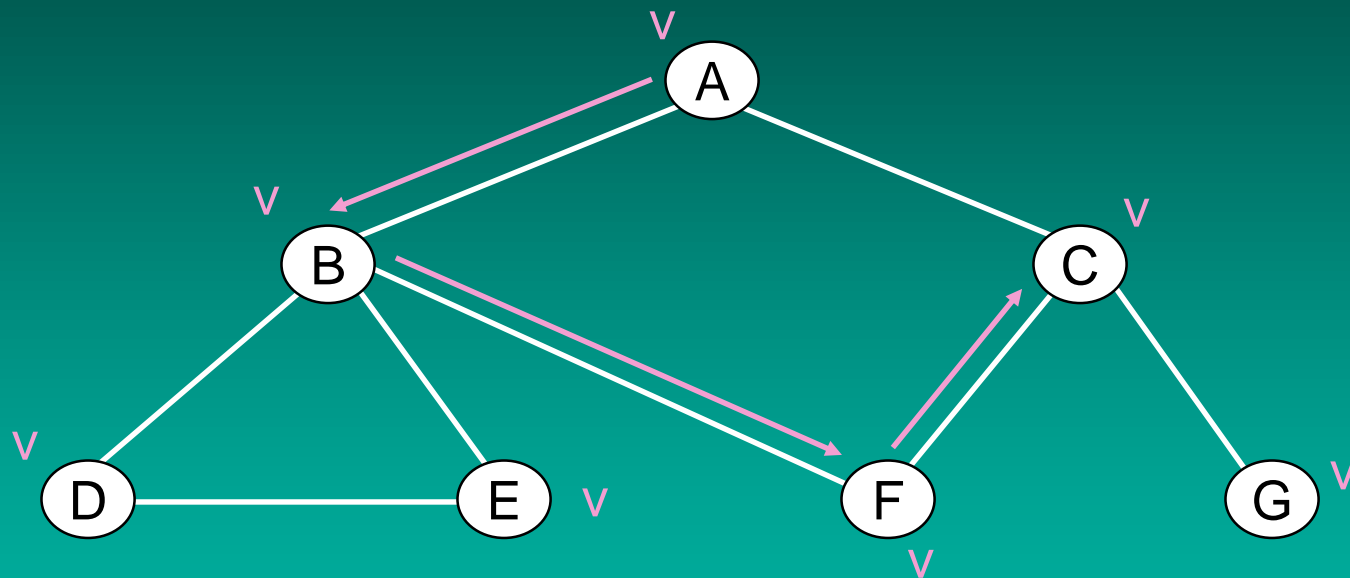


# Depth-First Search



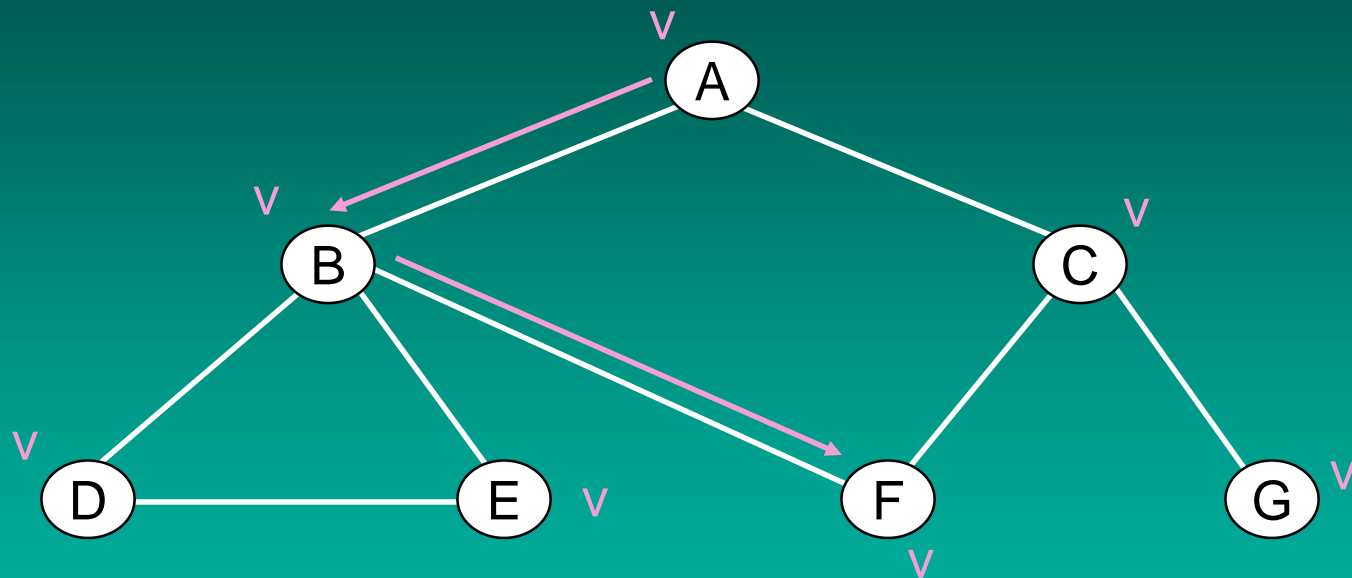
A B D E F C G

# Depth-First Search



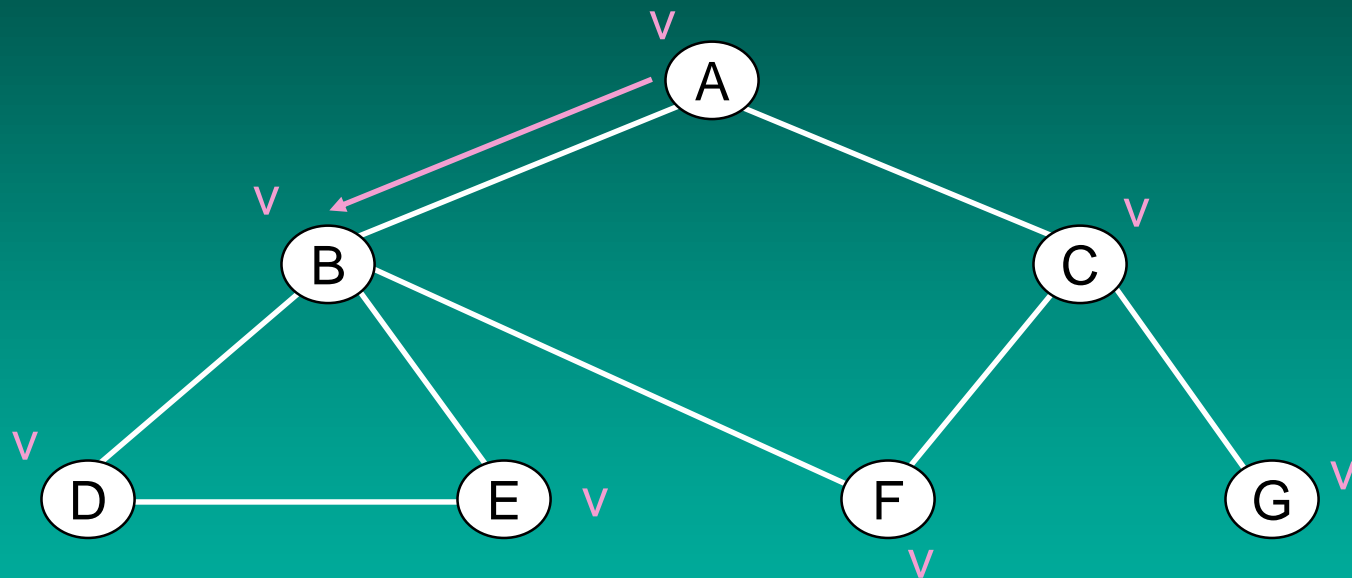
A B D E F C G

# Depth-First Search



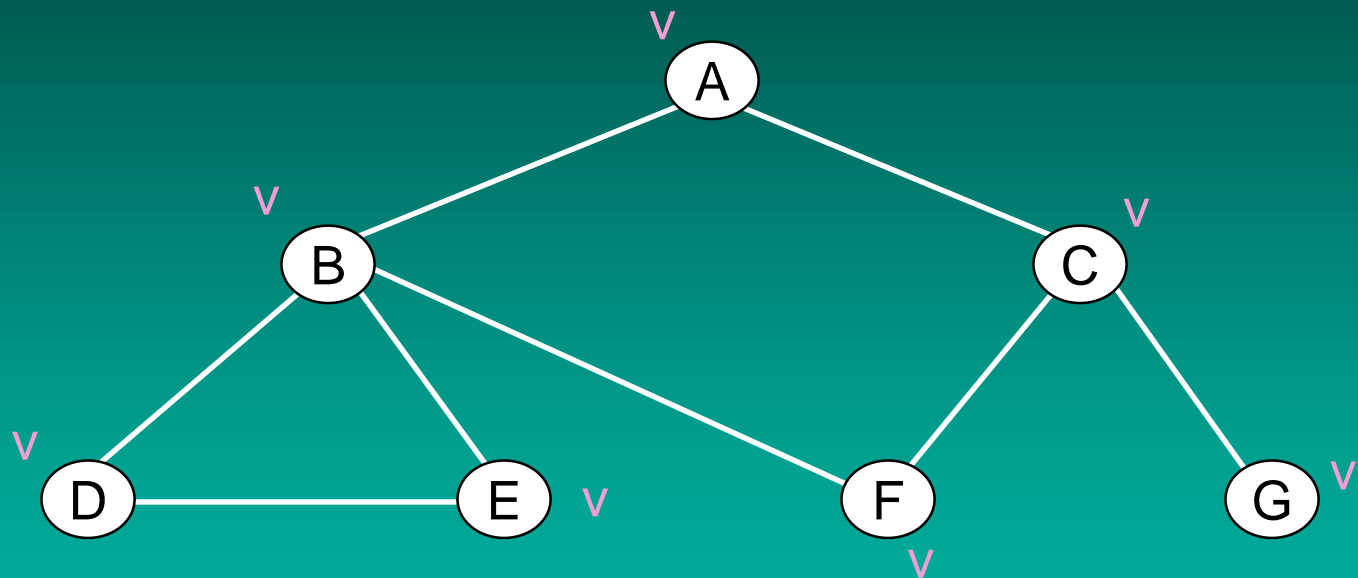
A B D E F C G

# Depth-First Search



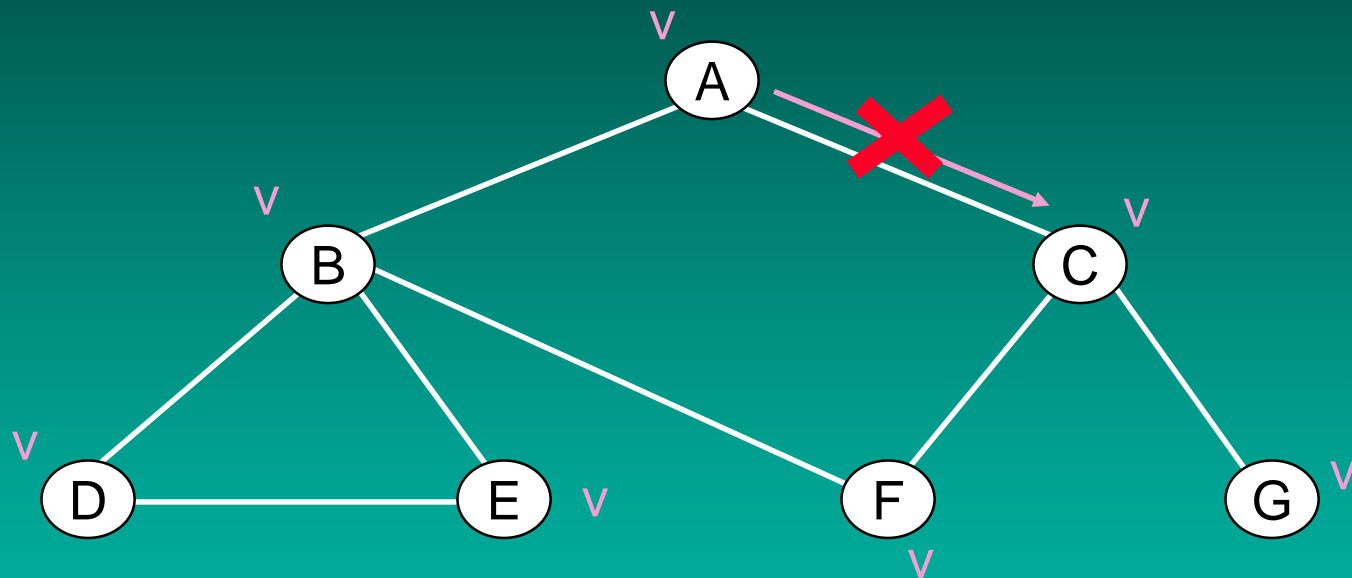
A B D E F C G

# Depth-First Search



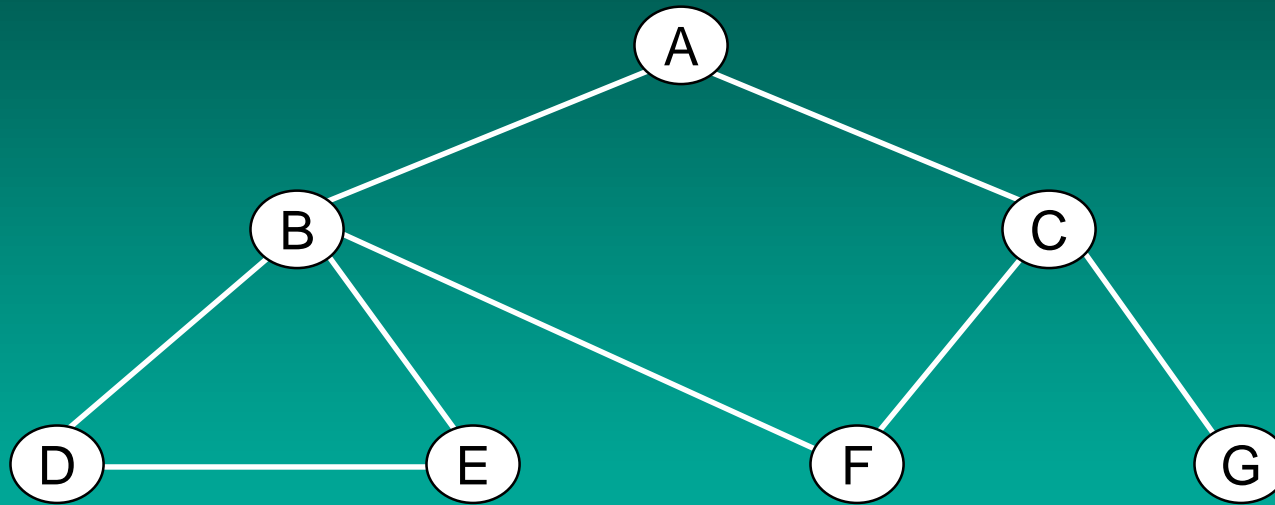
A B D E F C G

# Depth-First Search



A B D E F C G

# Depth-First Search



A B D E F C G

# Time and Space Complexity for Depth-First Search

## ■ Time Complexity

### – Adjacency Lists

- Each node is marked visited once
- Each node is checked for each incoming edge
- $O(v + e)$

### – Adjacency Matrix

- Have to check all entries in matrix:  $O(n^2)$



# Time and Space Complexity for Depth-First Search

## ■ Space Complexity

- Stack to handle nodes as they are explored

- Worst case: all nodes put on stack (if graph is linear)
- $O(n)$

# Breadth-First Graph Traversal Algorithm

Alyce Brady

# Breadth-first Search

- Similar to Breadth-first Traversal of a Binary Tree
- Choose a starting vertex
- Search all adjacent vertices
- Return to each adjacent vertex in turn and visit all of its adjacent vertices

# Pseudo-Code for Breadth-First Search

## breadth-first-search

mark starting vertex as visited; put on queue

while the queue is not empty

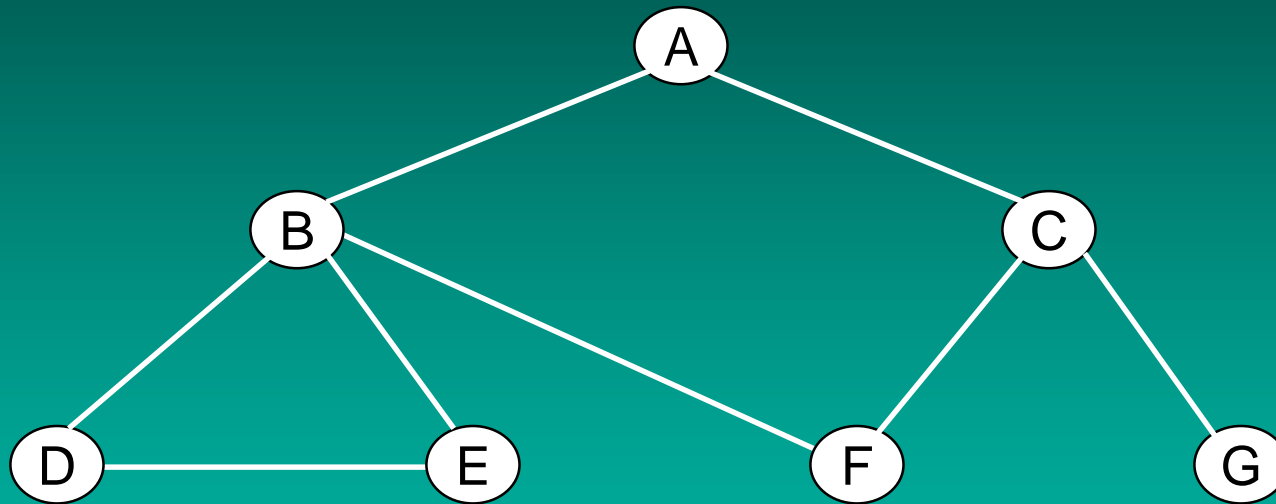
    dequeue the next node

    for all unvisited vertices adjacent to this  
    one

        –mark vertex as visited

        –add vertex to queue

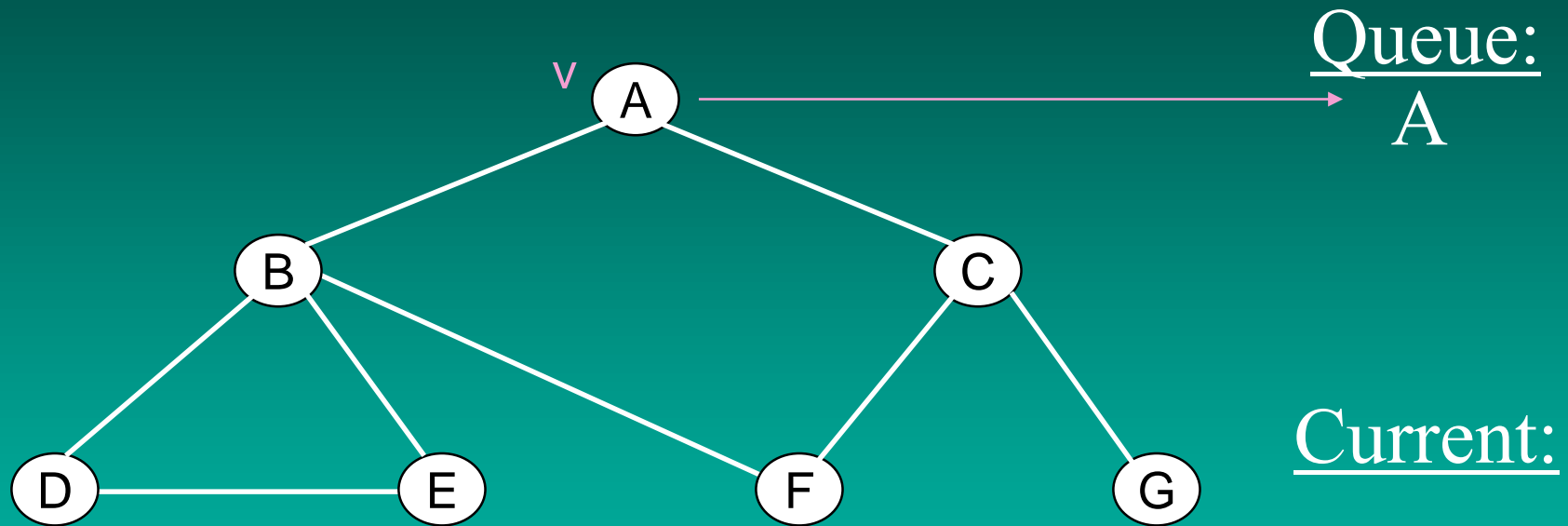
# Breadth-First Search



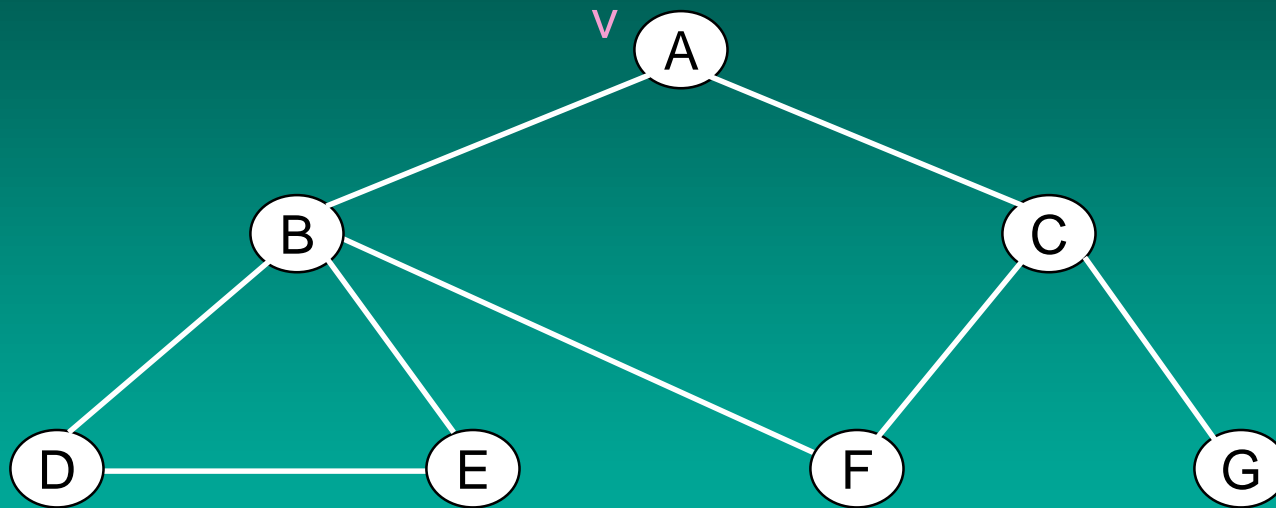
Queue:

Current:

# Breadth-First Search



# Breadth-First Search



Queue:

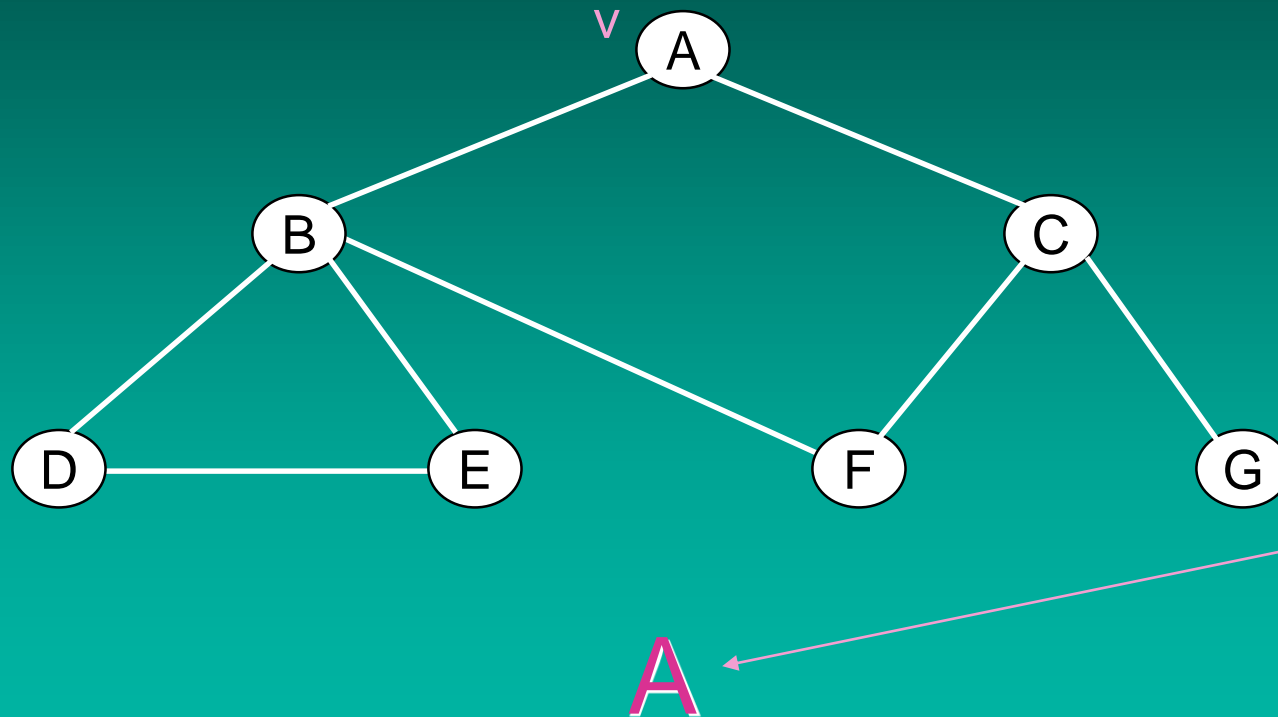
A



Current:

A

# Breadth-First Search



Queue:

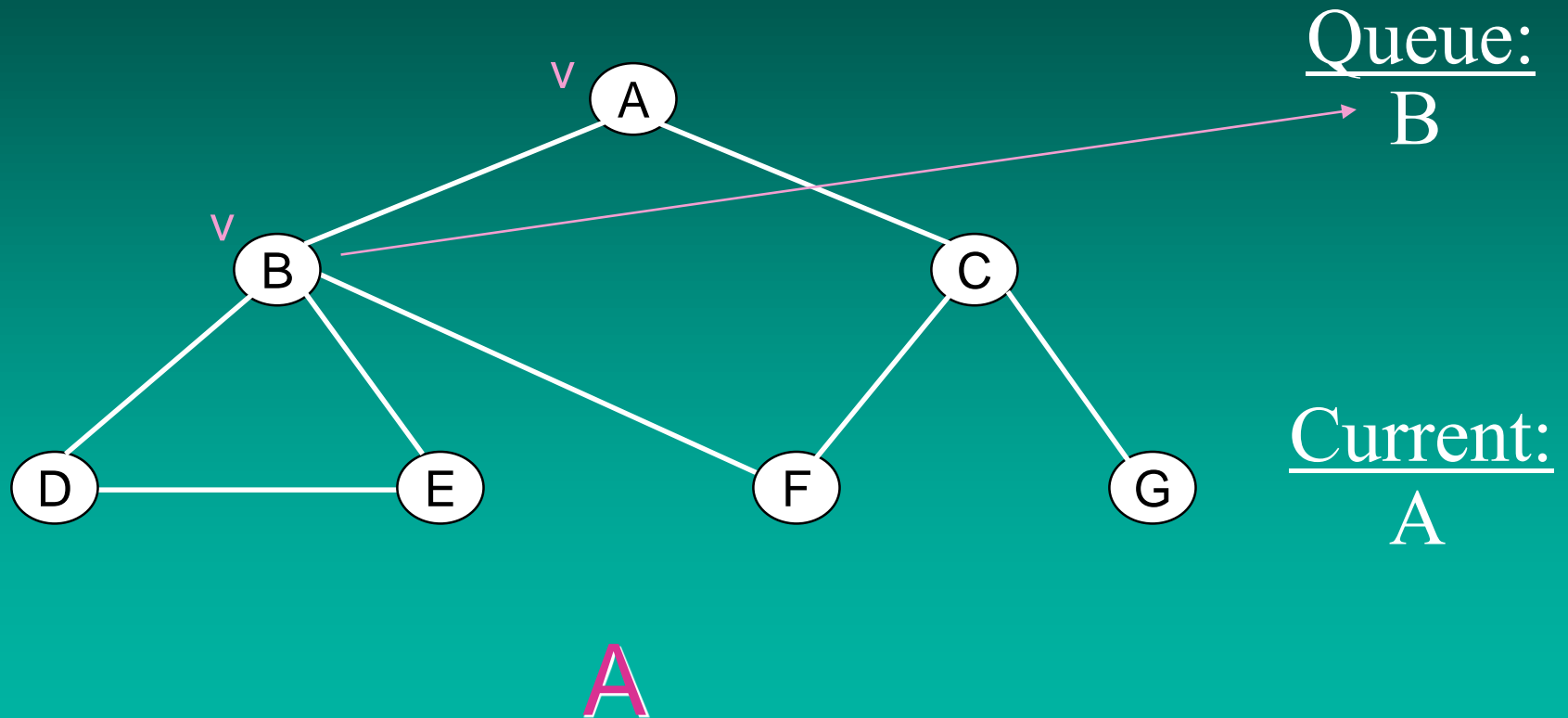
Current:

A

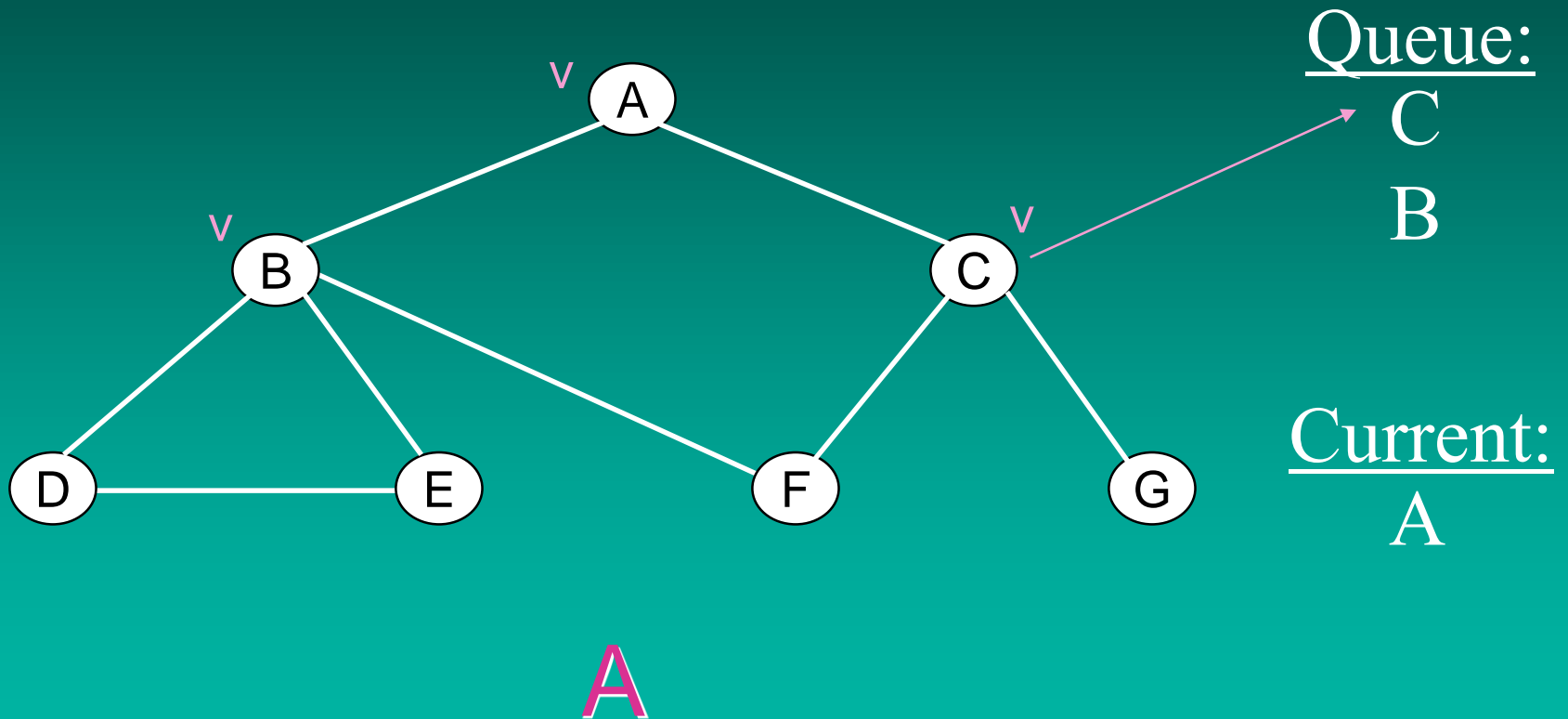
A



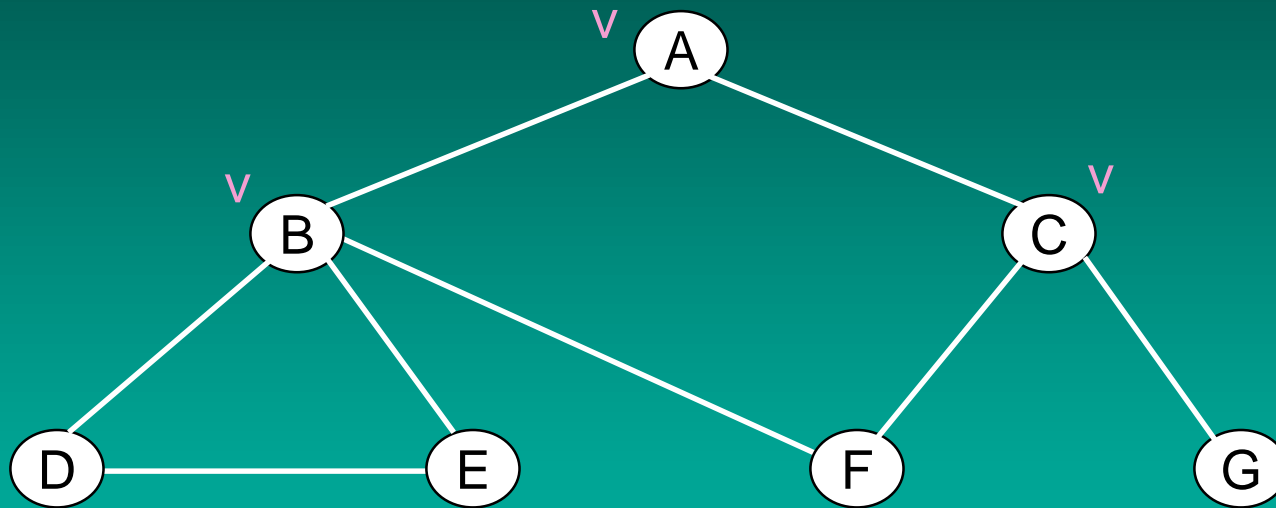
# Breadth-First Search



# Breadth-First Search



# Breadth-First Search



Queue:

C

B

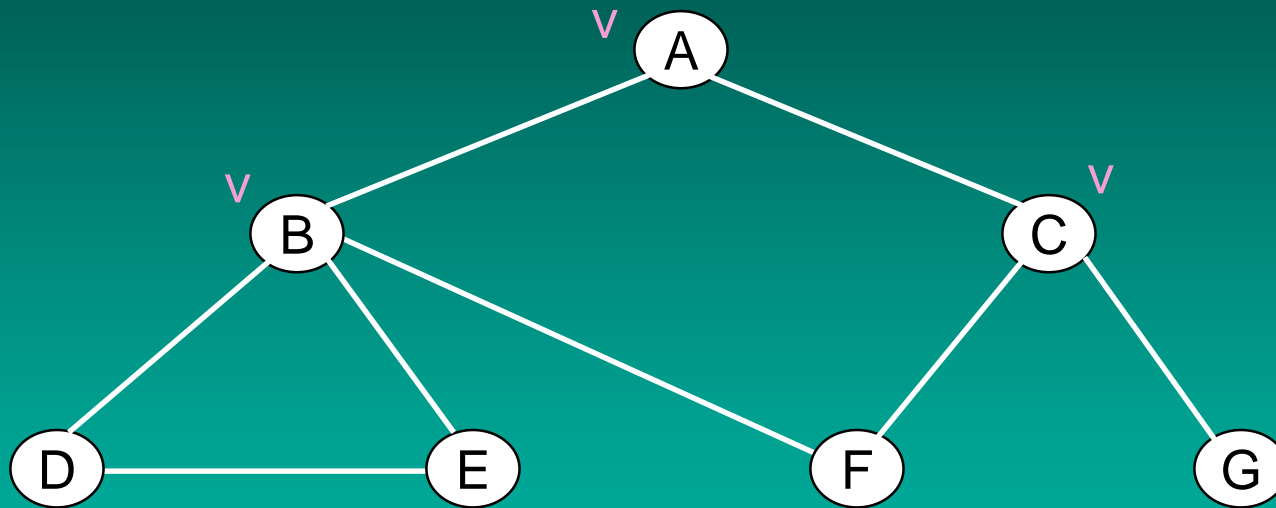


Current:

B

A

# Breadth-First Search

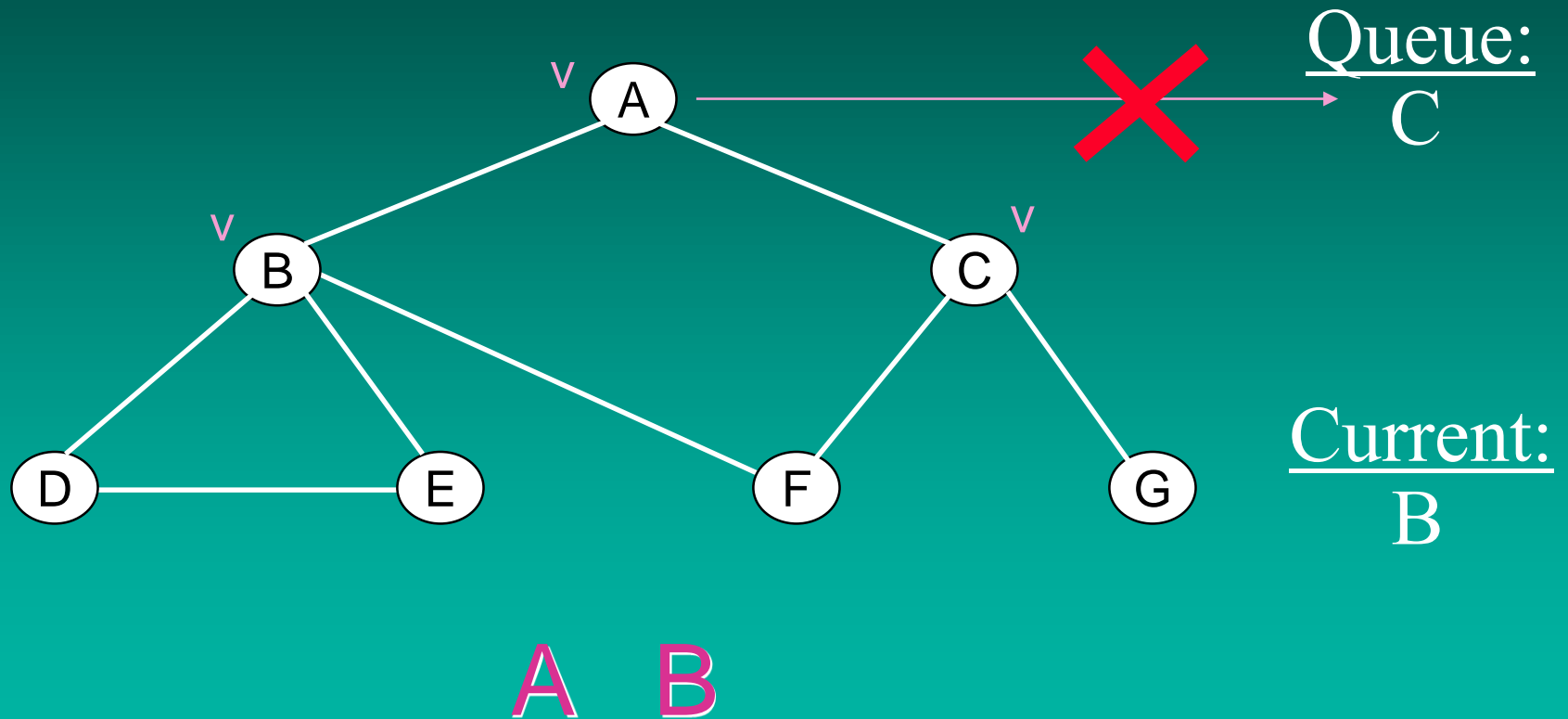


Queue:  
C

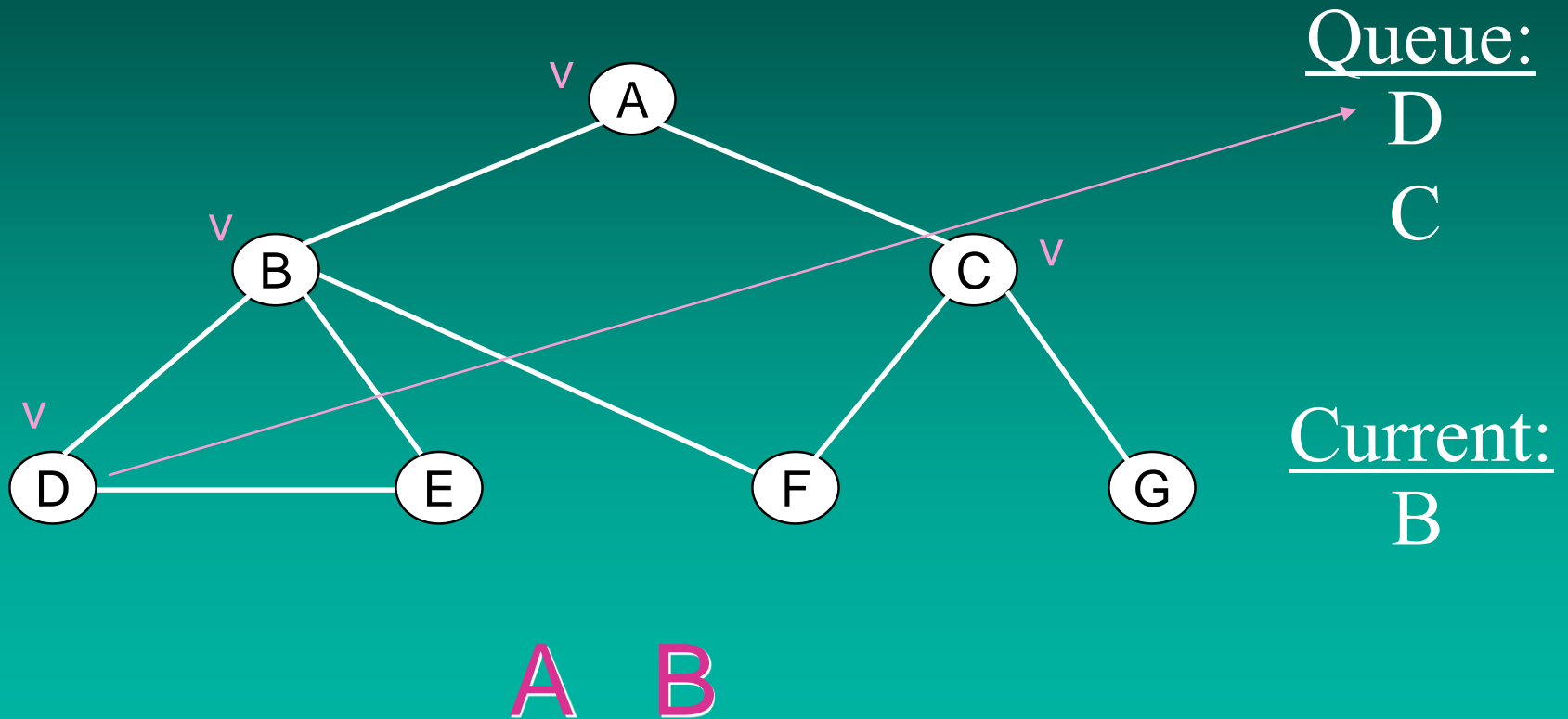
Current:  
B

A B

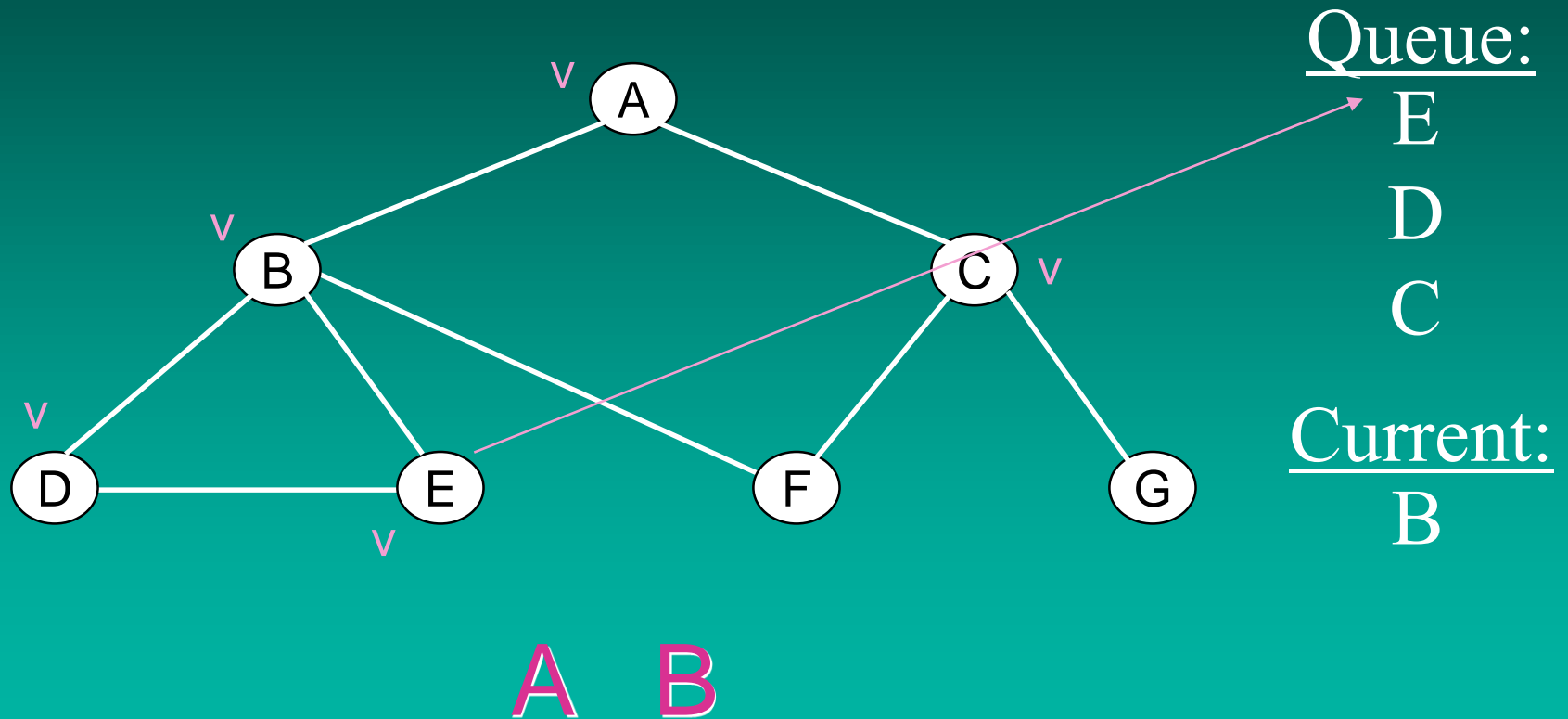
# Breadth-First Search



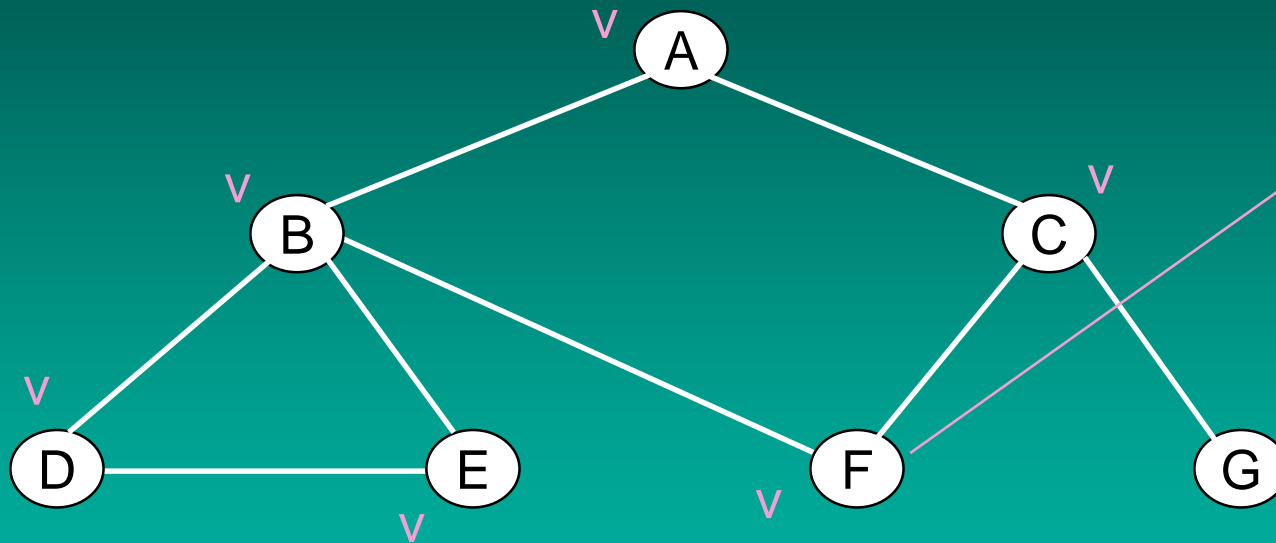
# Breadth-First Search



# Breadth-First Search



# Breadth-First Search



Queue:

F  
E  
D  
C

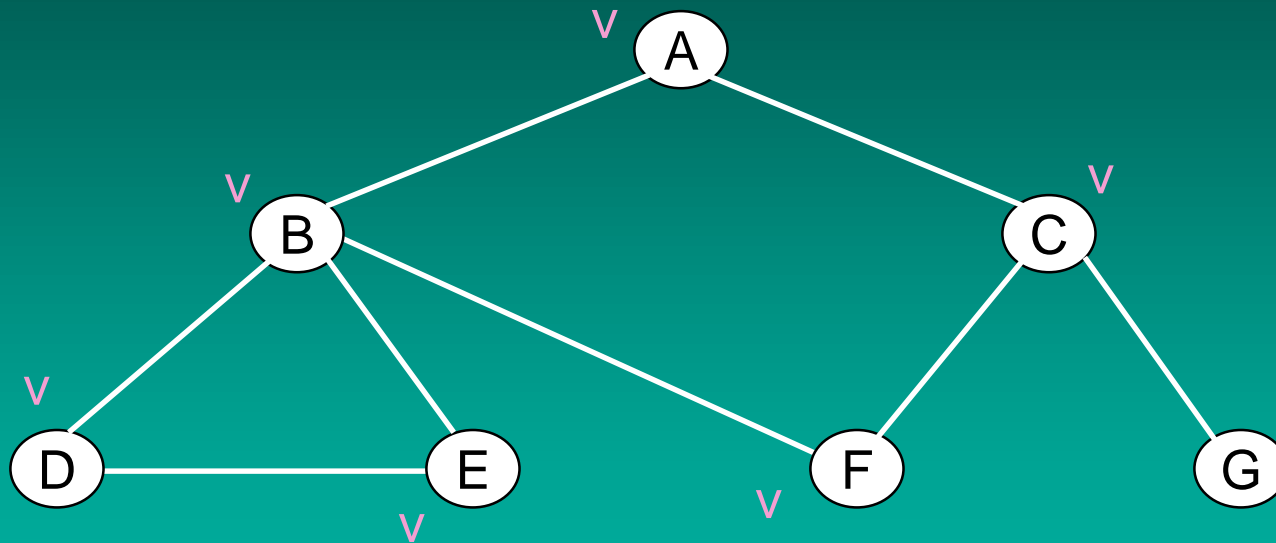
Current:

B

A B



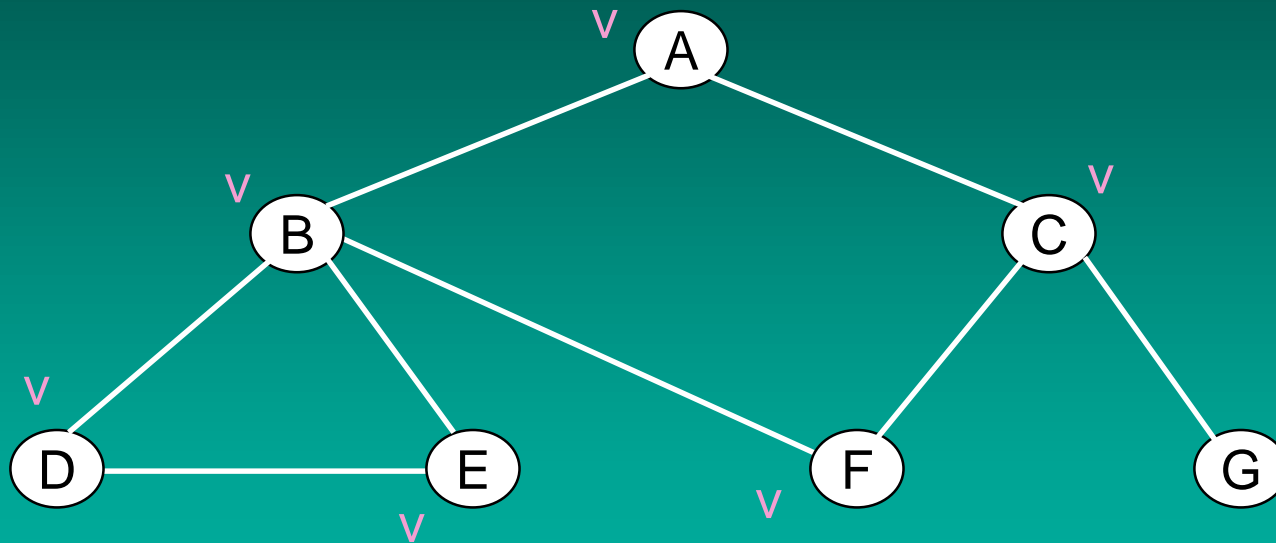
# Breadth-First Search



A B

Queue:  
F  
E  
D  
C  
↓  
Current:  
C

# Breadth-First Search



Queue:

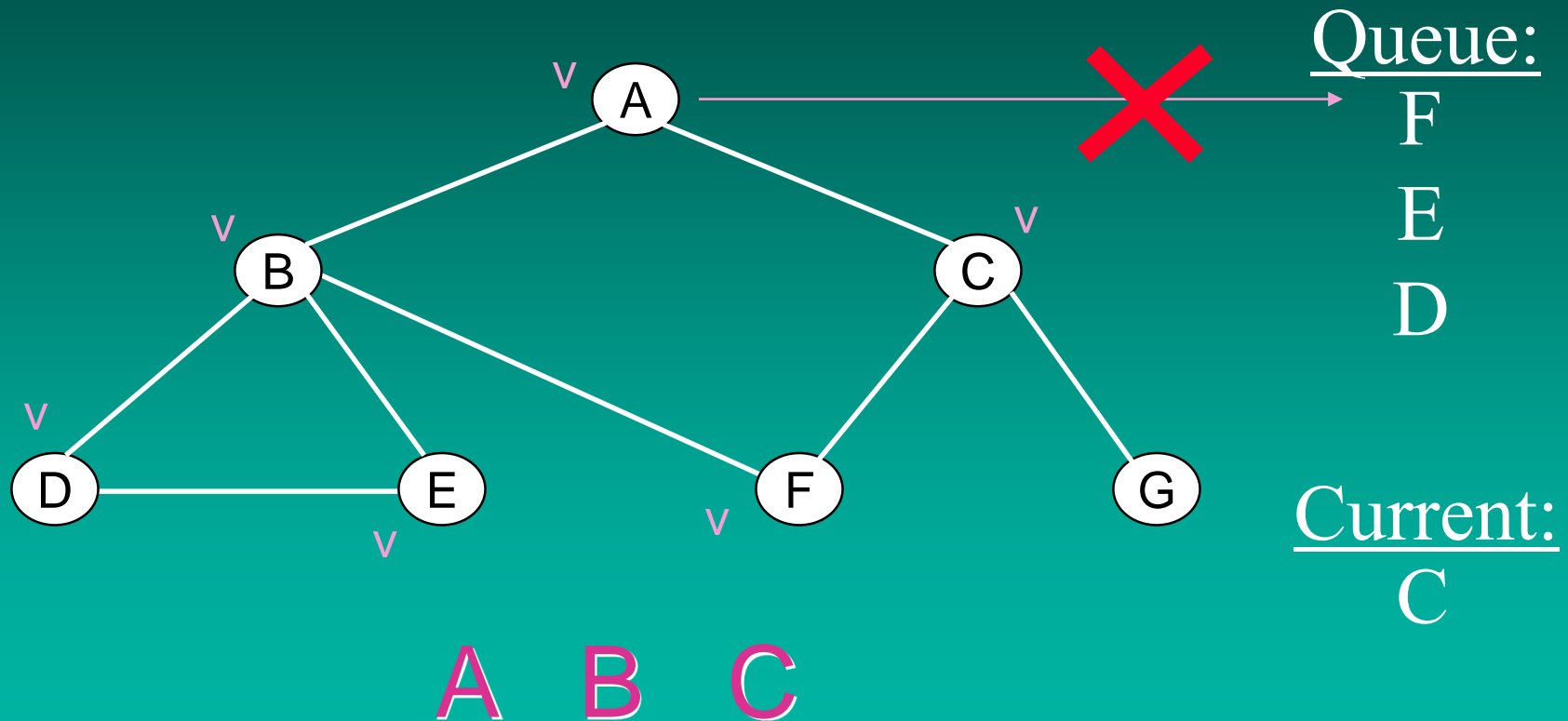
F  
E  
D

Current:

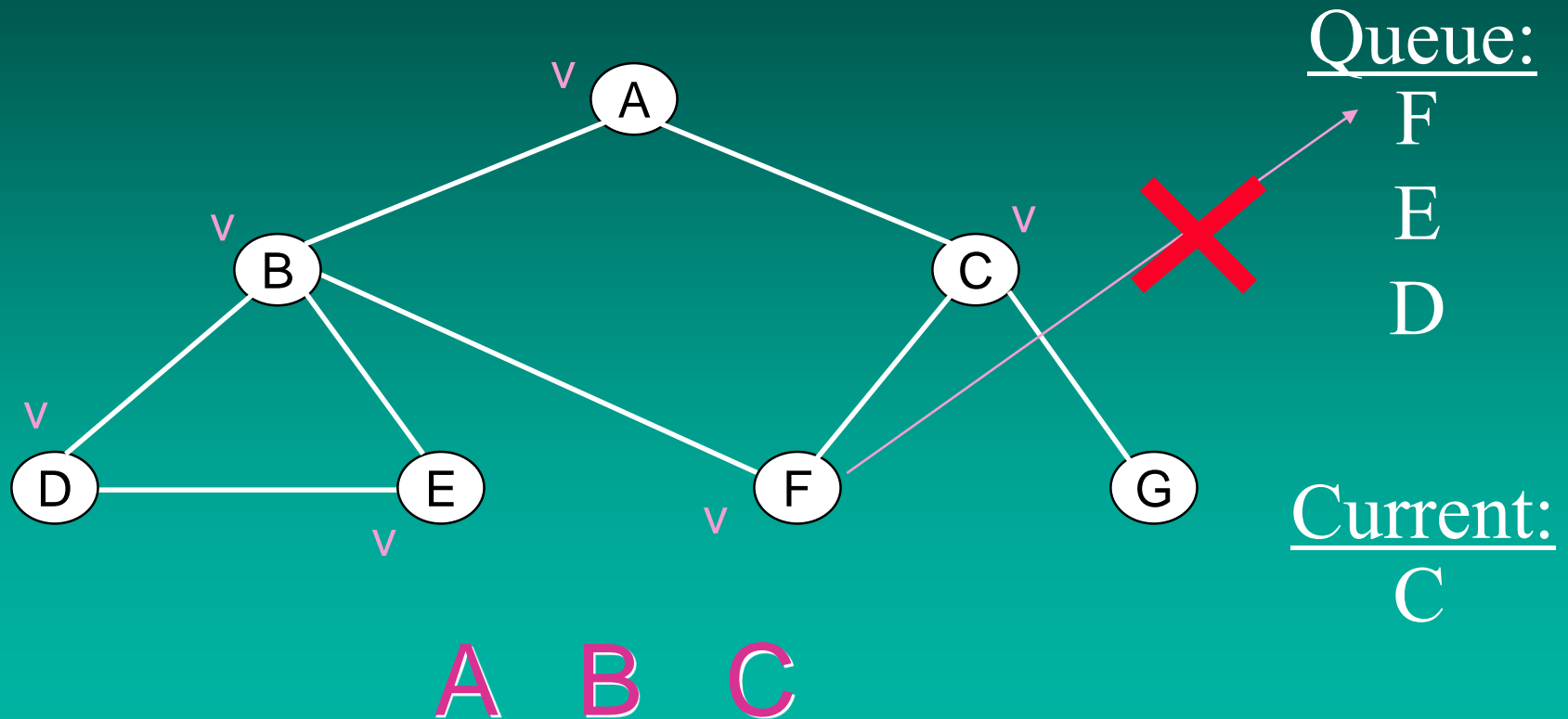
C

A B C

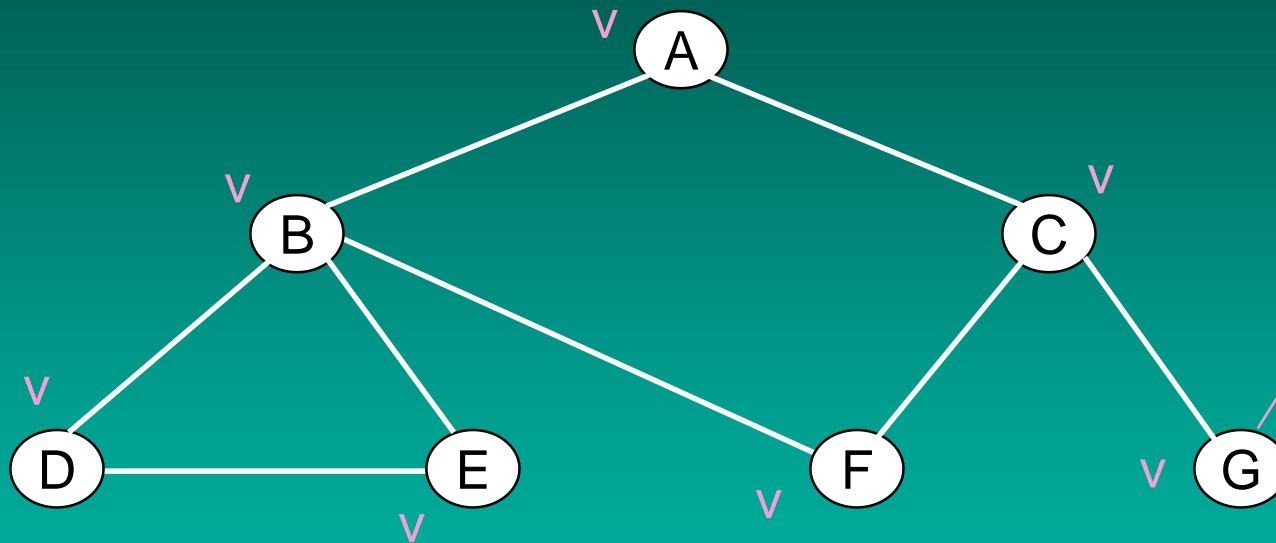
# Breadth-First Search



# Breadth-First Search



# Breadth-First Search



Queue:

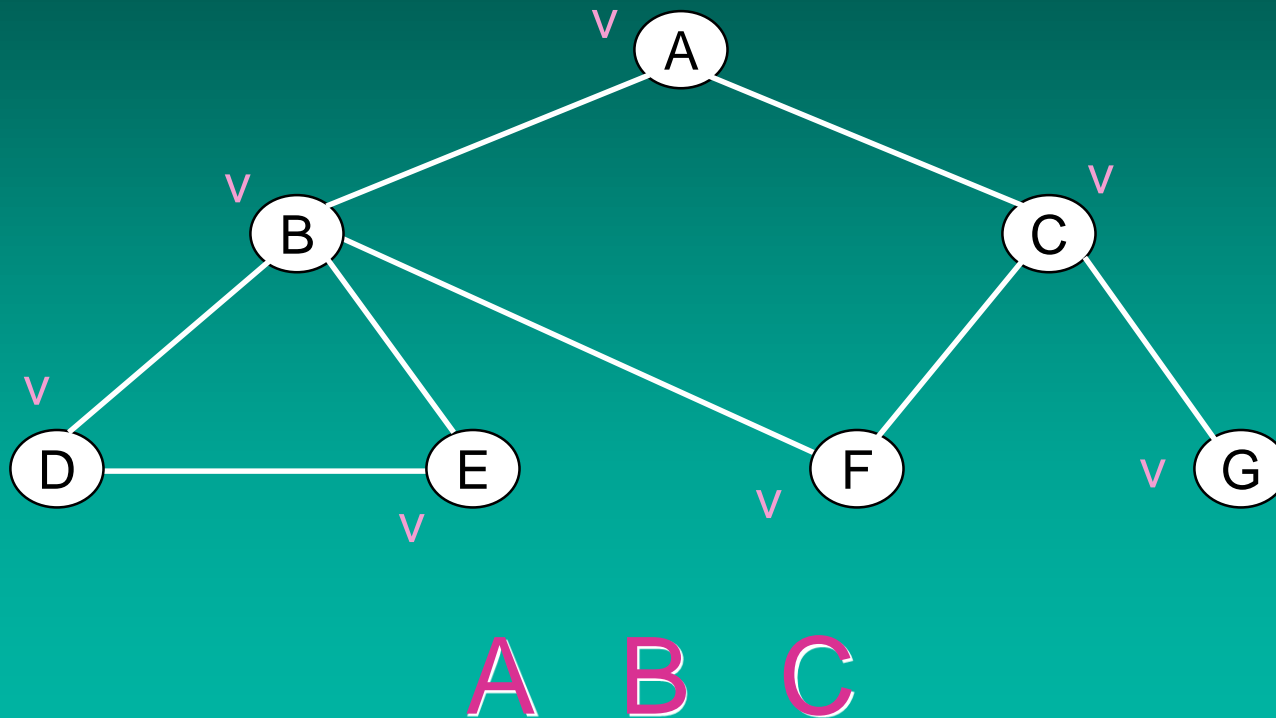
G  
F  
E  
D

Current:

C

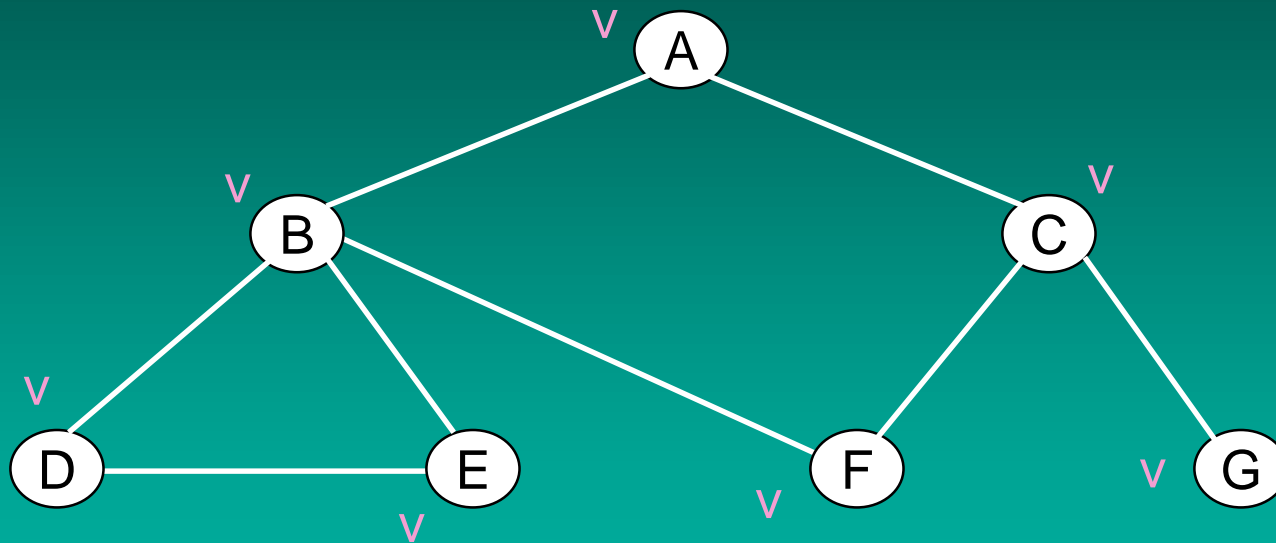
A B C

# Breadth-First Search



Queue:  
G  
F  
E  
D  
↓  
Current:  
D

# Breadth-First Search



Queue:

G

F

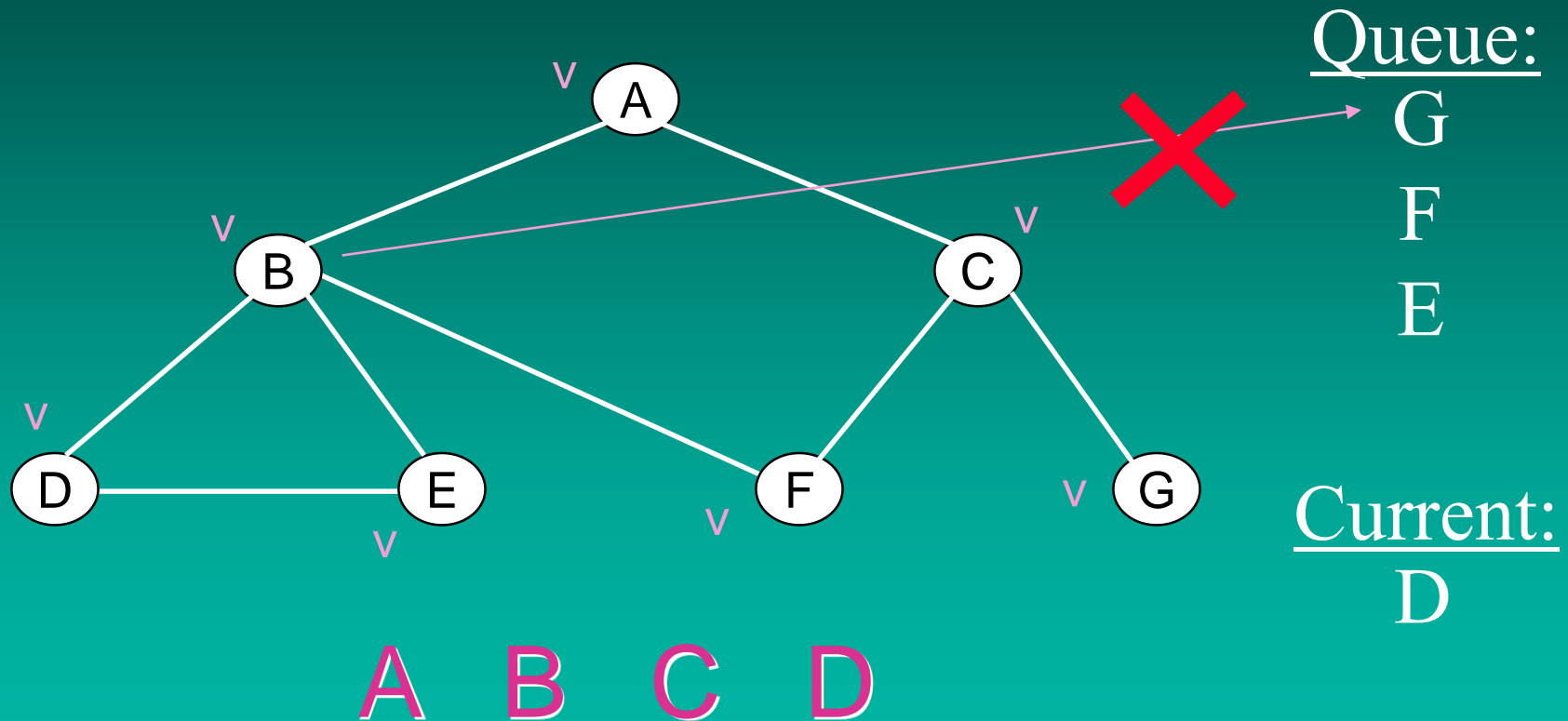
E

Current:

D

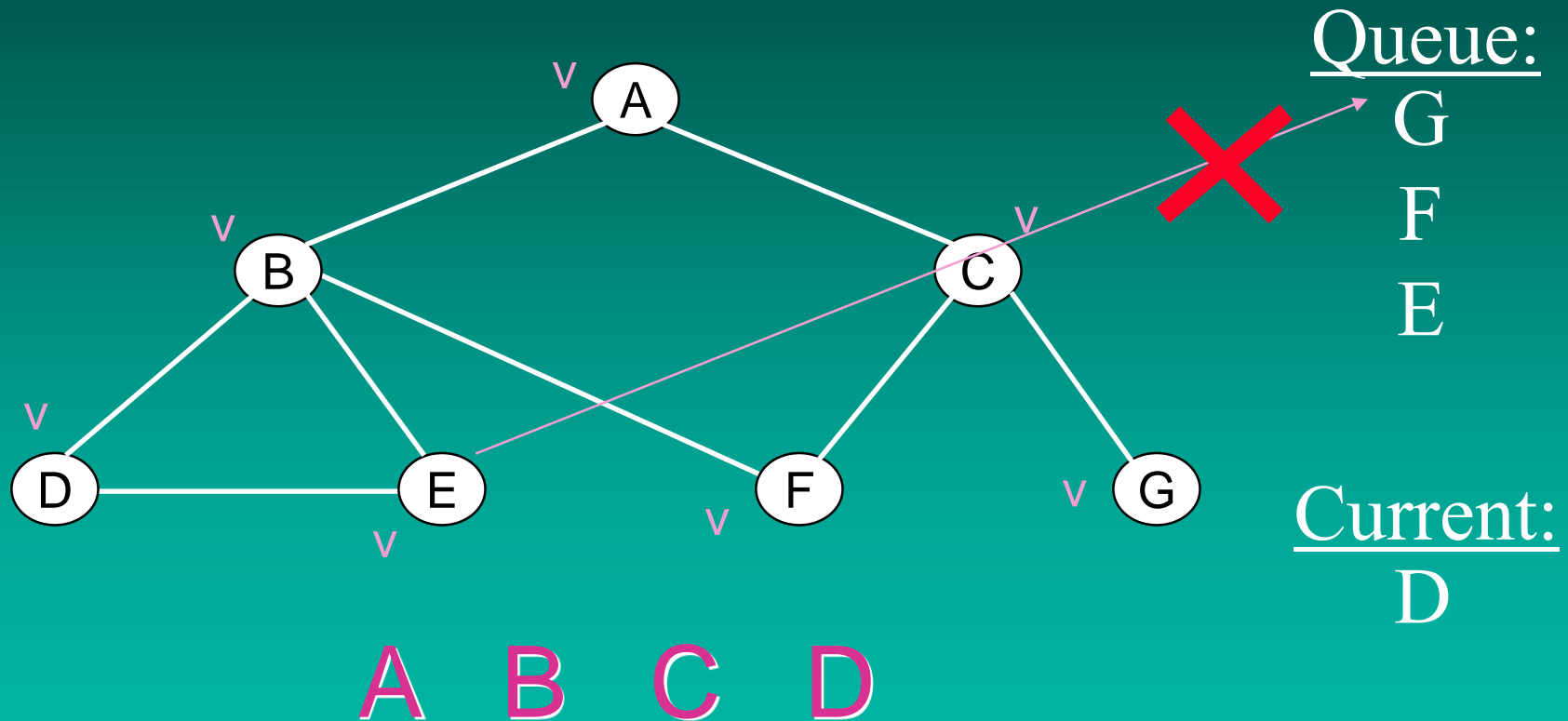
A B C D

# Breadth-First Search

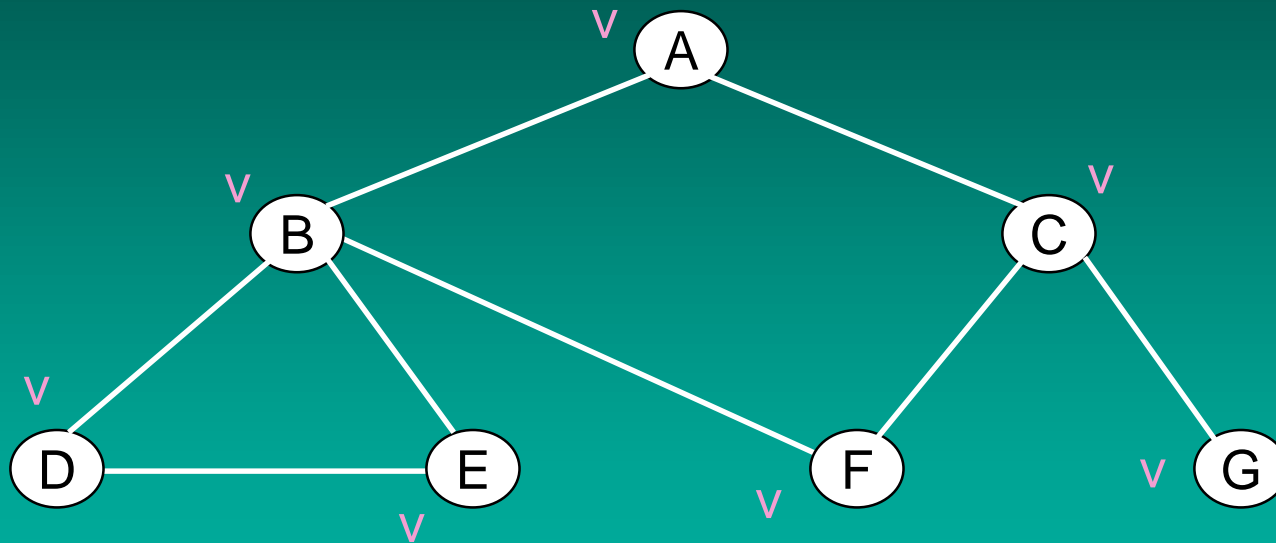




# Breadth-First Search



# Breadth-First Search



A B C D

Queue:

G

F

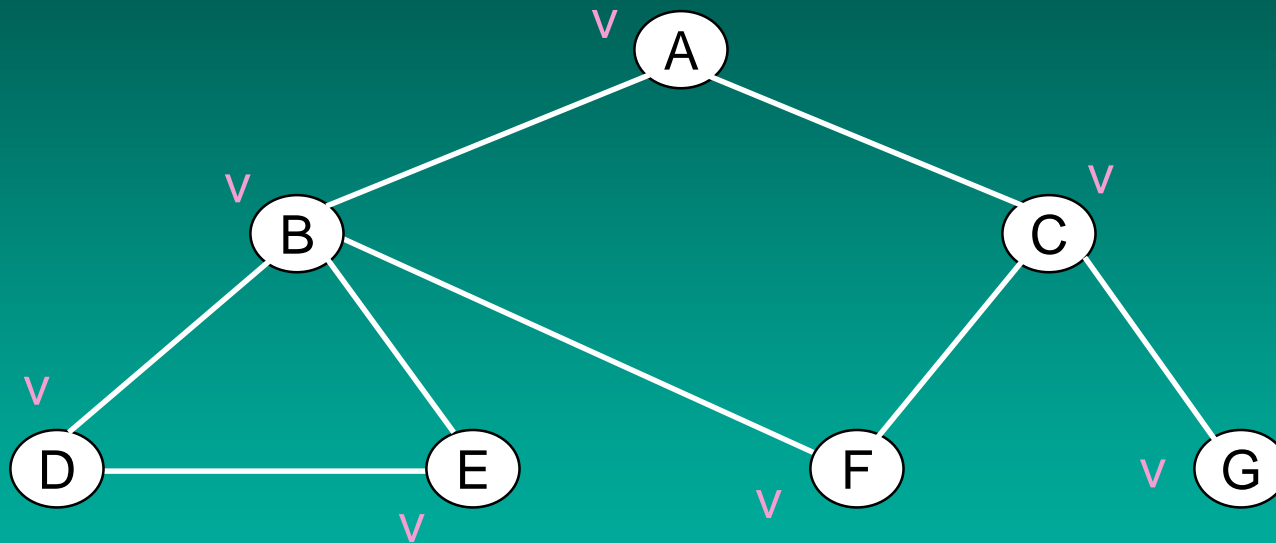
E



Current:

E

# Breadth-First Search

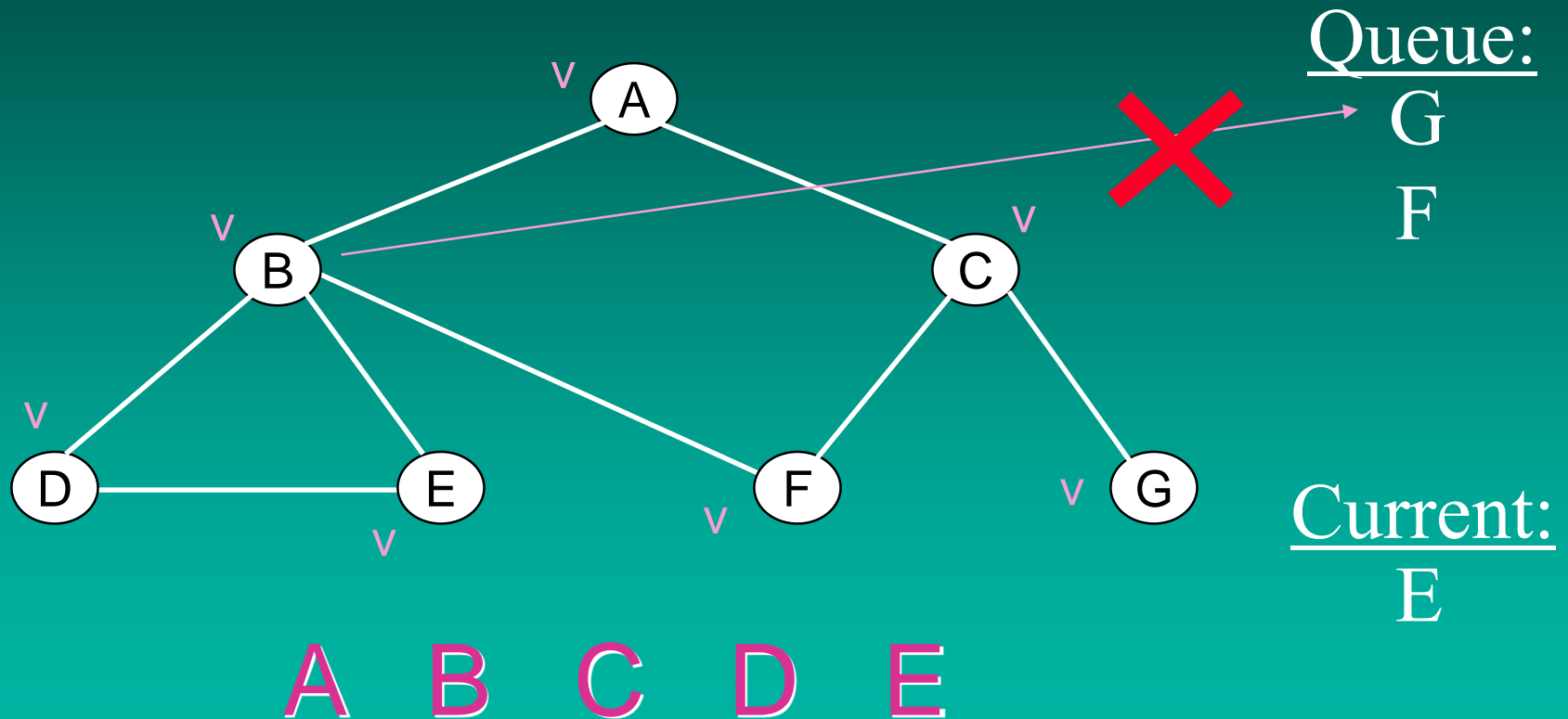


Queue:  
G  
F

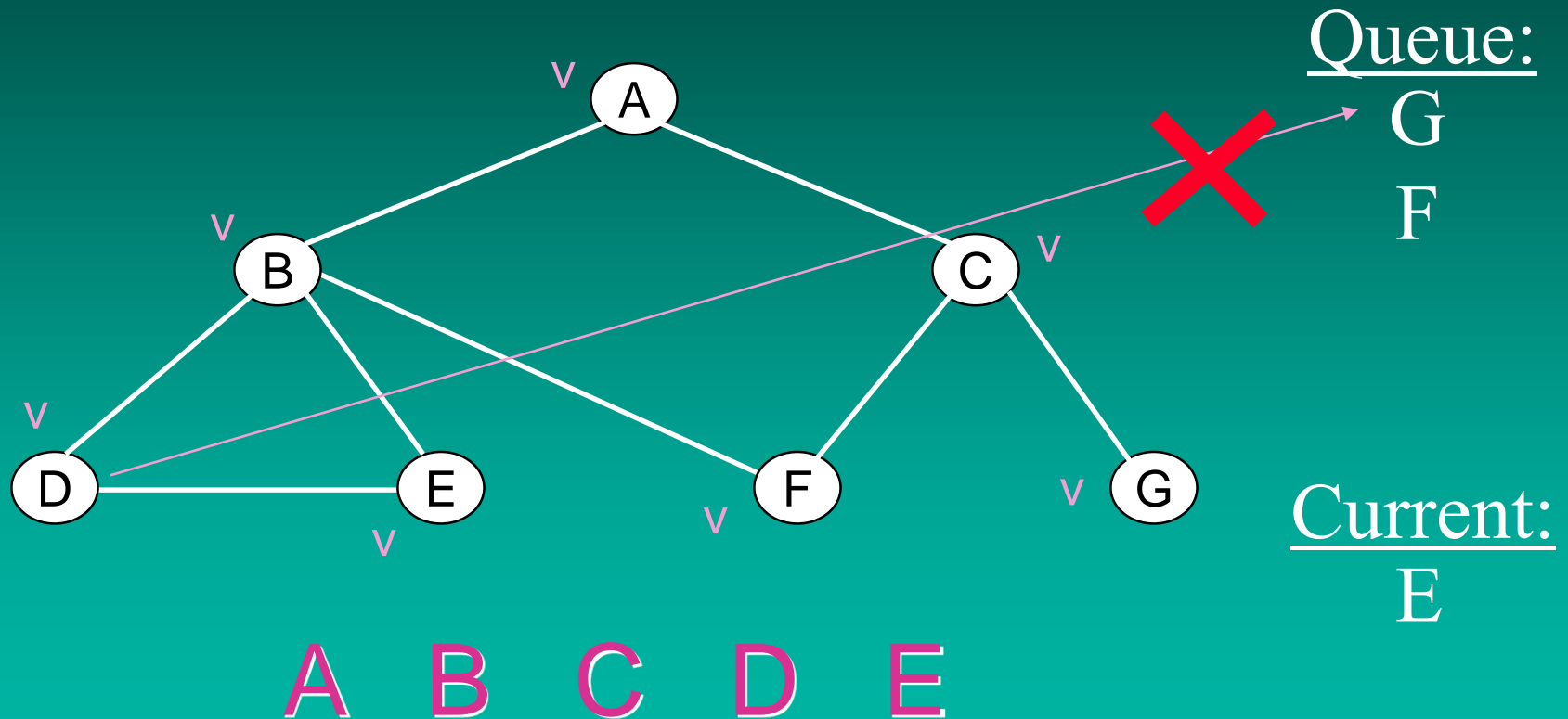
Current:  
E

A B C D E

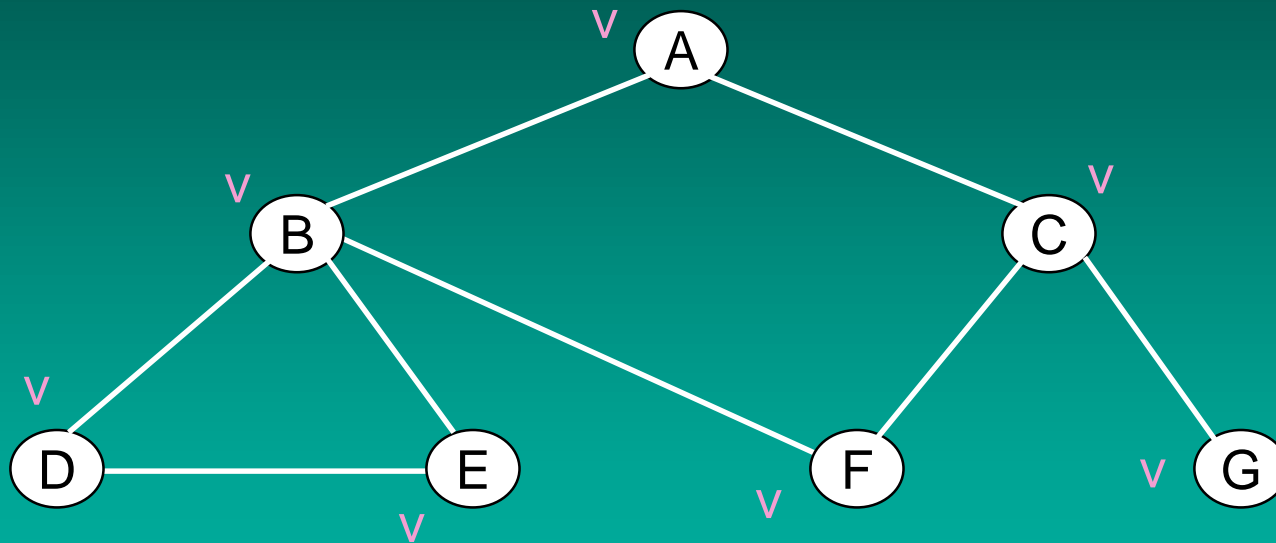
# Breadth-First Search



# Breadth-First Search



# Breadth-First Search



A B C D E

Queue:

G

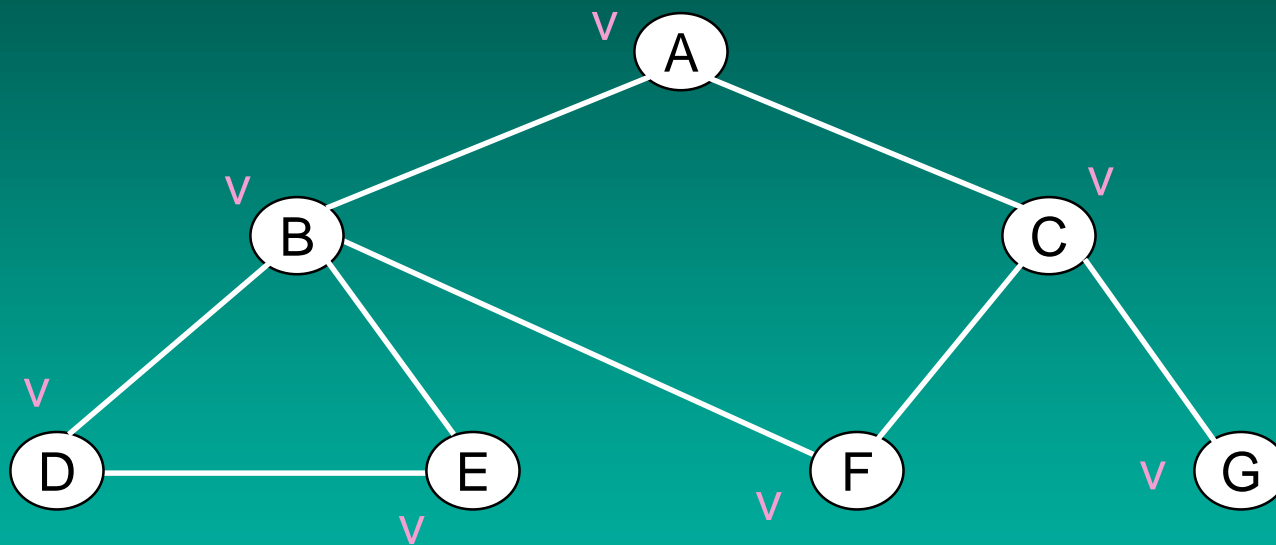
F



Current:

F

# Breadth-First Search

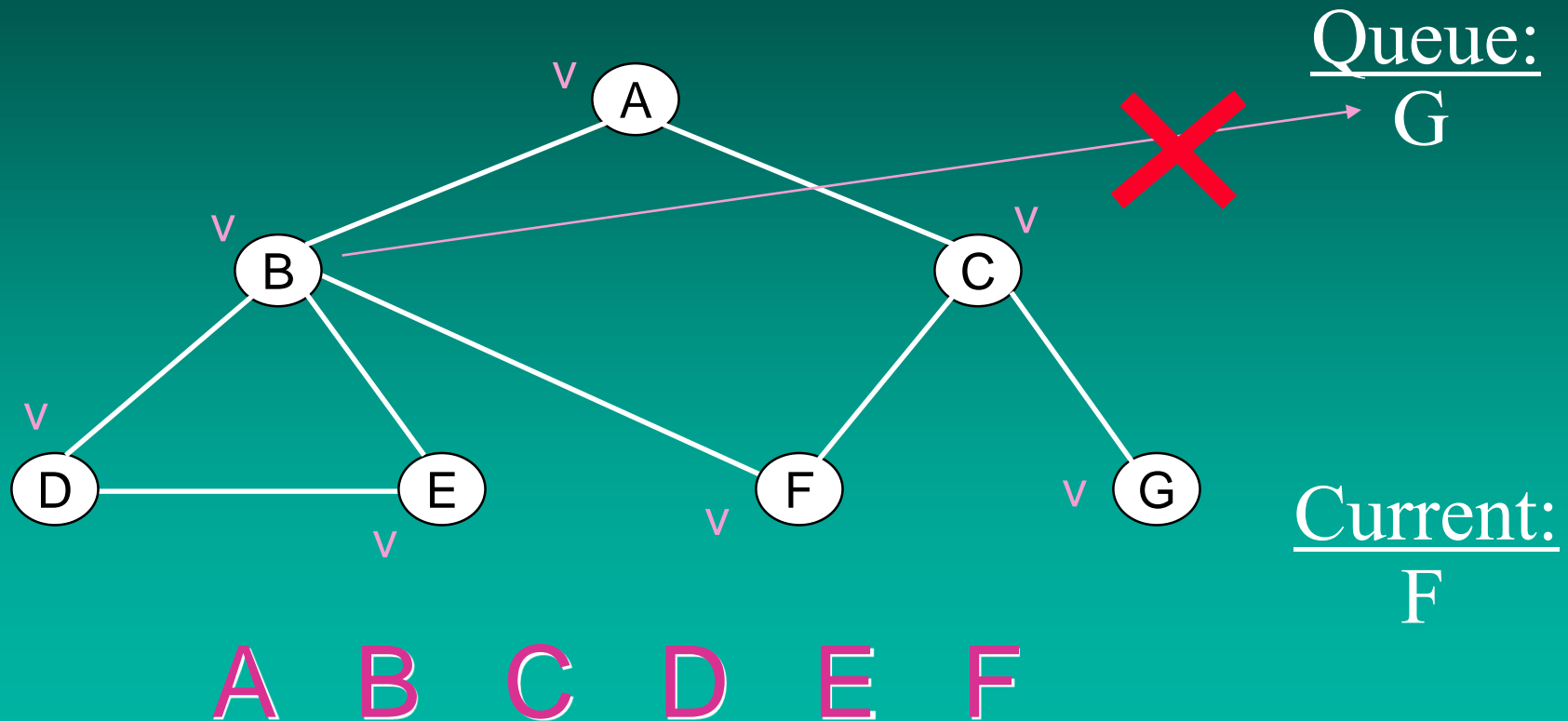


Queue:  
G

Current:  
F

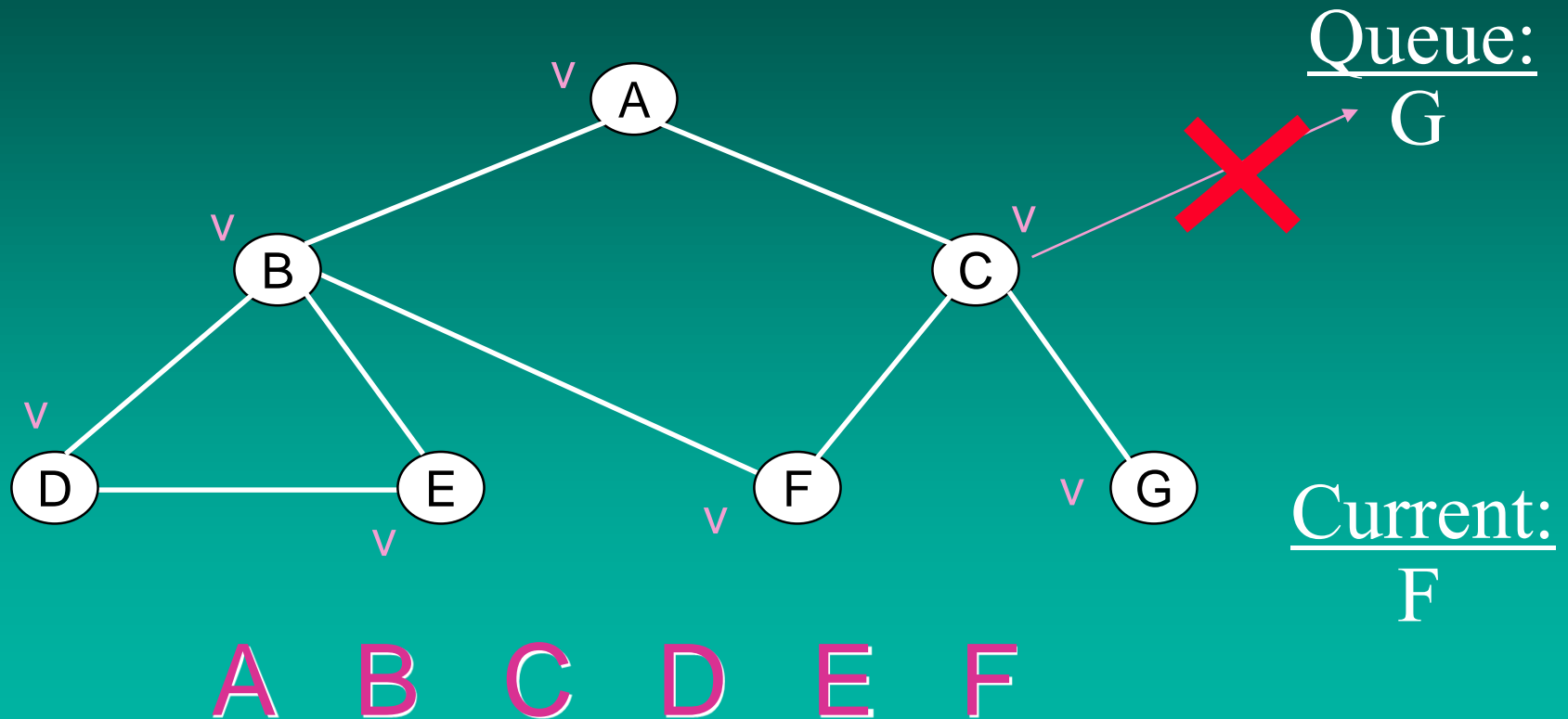
A B C D E F

# Breadth-First Search

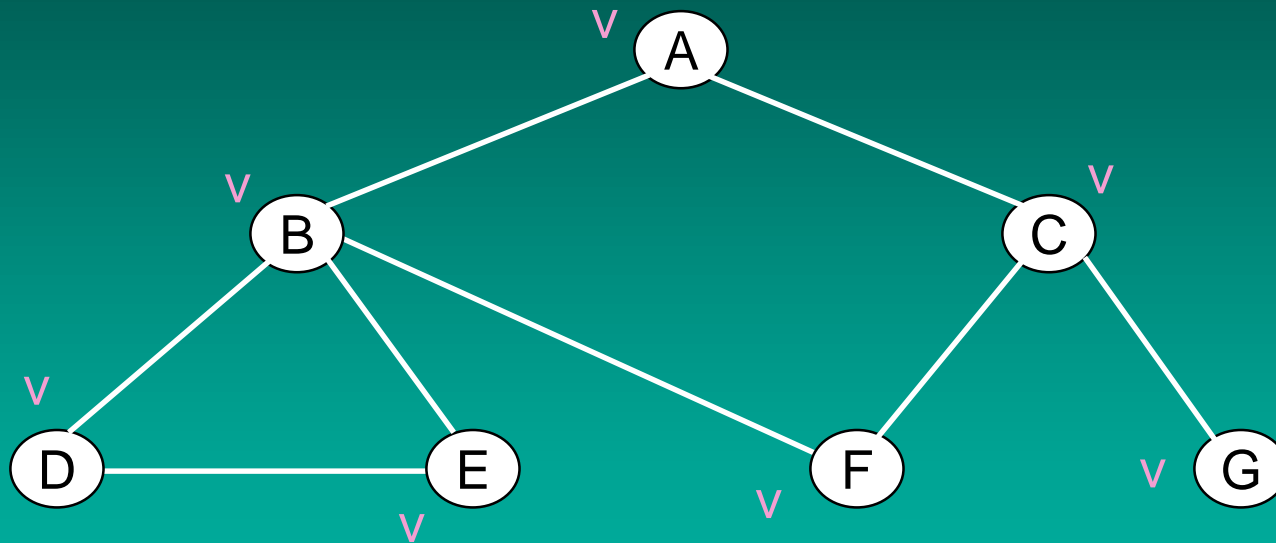




# Breadth-First Search



# Breadth-First Search



A B C D E F

Queue:

G

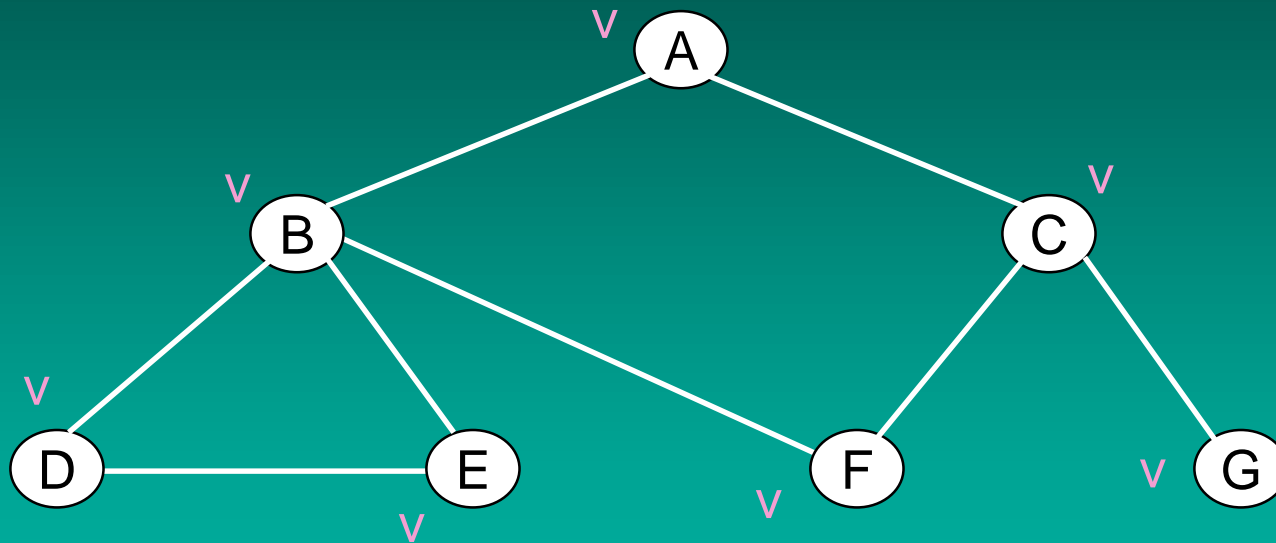


Current:

G

# Breadth-First Search

Queue:

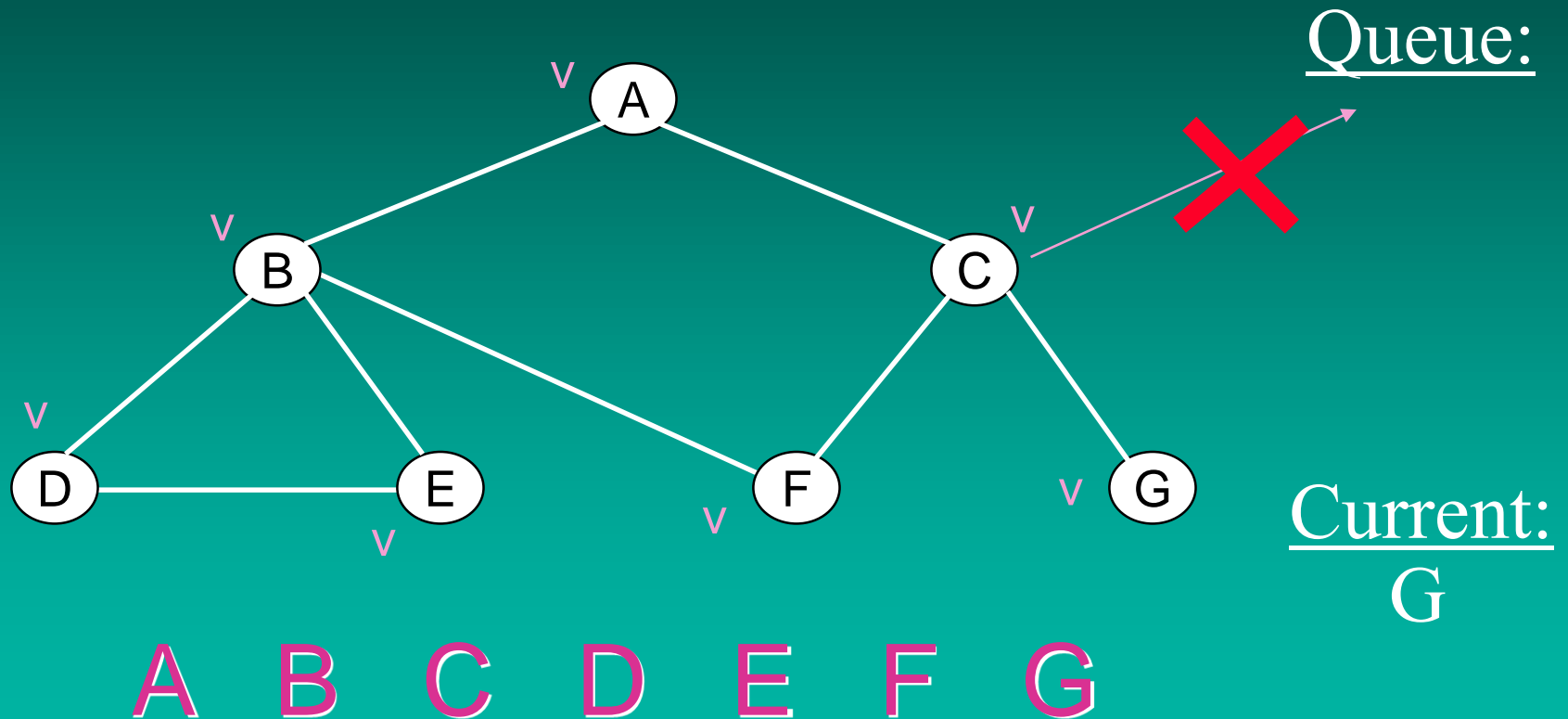


Current:

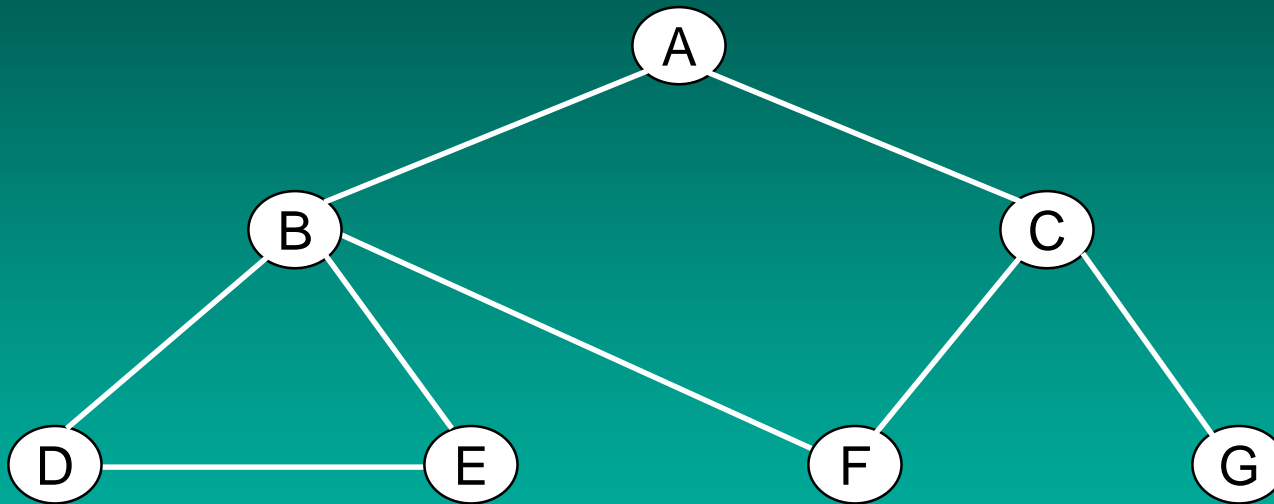
G

A B C D E F G

# Breadth-First Search



# Breadth-First Search



A B C D E F G

# Time and Space Complexity for Breadth-First Search

## ■ Time Complexity

### – Adjacency Lists

- Each node is added to queue once
- Each node is checked for each incoming edge
- $O(v + e)$

### – Adjacency Matrix

- Have to check all entries in matrix:  $O(n^2)$

# Time and Space Complexity for Breadth-First Search

## ■ Space Complexity

- Queue to handle unexplored nodes
  - Worst case: all nodes put on queue (if all are adjacent to first node)
  - $O(n)$

# Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- Shortest path: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
  - Austin->Houston->Atlanta->Washington:  
1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington:  
2980 miles



# Single-source shortest-path problem (cont.)

- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is weightless or all the weights are the same

(mark vertices before Enqueue)

