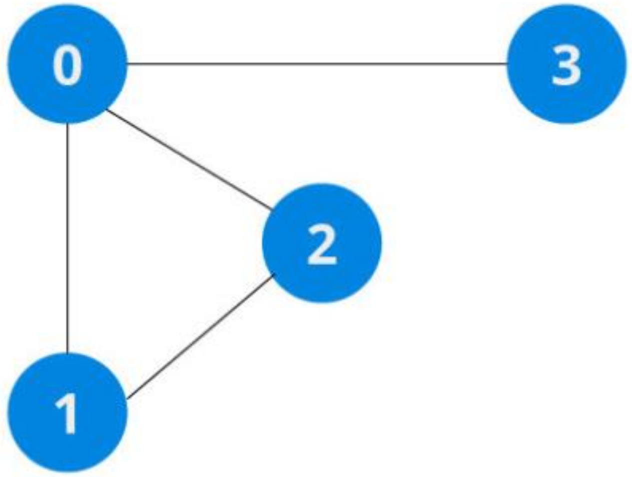
Graphs

CS223: Data Structures

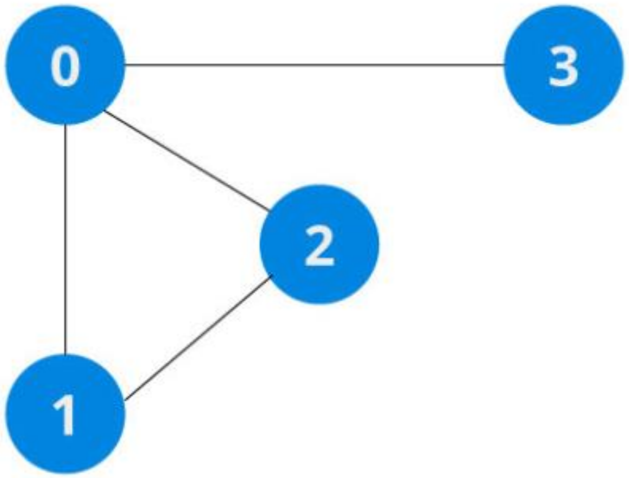
Definitions

- A graph is a non-linear data structure that enables representing relationships between different types of data.
- Real Applications:
  - Google maps, social media, web Page searching, City Planning, Traffic Control, Transportation & Navigation, Travelling Salesman Problems, GSM mobile phone networks
- There are two main parts of a graph G(V,E):
  - The vertices (nodes) where the data is stored (V).
  - The edges (connections) which connect the nodes (E).
    - represented as pairs of vertices (u,v), where $u,v \in V$.
- In the right graph:
  - V = {0, 1, 2, 3}
  - E = {(0,1), (0,2), (0,3), (1,2)}
  - G = {V, E}



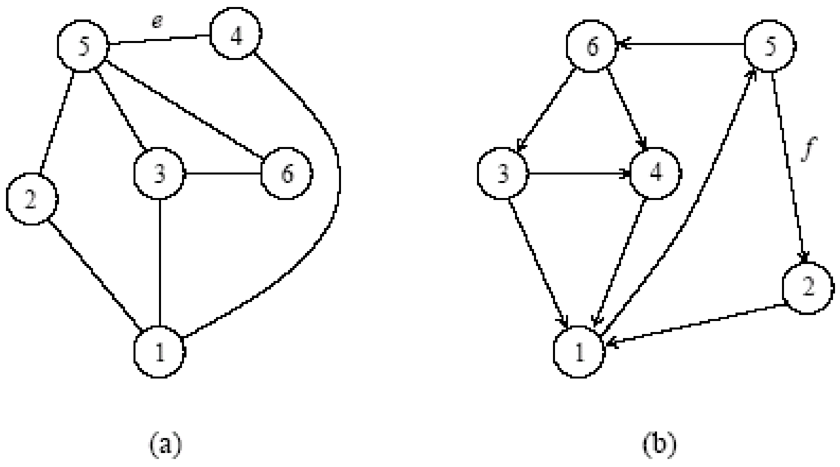
Definitions (Cont.)

- A **path** is a sequence of edges that allows you to go from a vertex, e.g., 0, to another vertex, e.g., 2.
  - Example: (0-1, 1-2) and (0-2) are paths from vertex 0 to vertex 2.
  - The length of a path is defined as the number of edges in the path.
- A vertex is said to be **adjacent** to another vertex if there is an edge connecting them. Vertices 2 and 3 are **not** adjacent because there is no edge between them.



<u>Definitions</u> (Cont.)

- Graphs can be **undirected** or **directed**
  - **Undirected graph:** The relationship exists in both directions (bi-directional) i.e. the edges do not point in any specific direction.
  - **Directed graph:** The relationships are based on the direction of the edges.
    - It can *be* a **one way relationship** *or* a **two-way relationship**, but it must be **explicitly** stated.
    - In a directed graph, the edge *e* is an ordered pair *(u,v)*.
    - An edge *(u,v)* is *incident from* vertex *u* and is *incident to* vertex *v*.



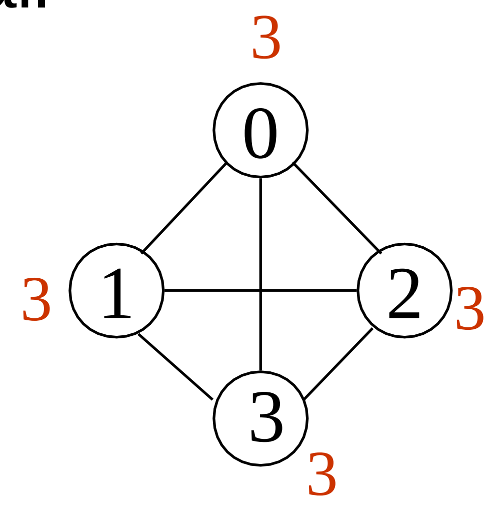
(a) An undirected graph and (b) a directed graph.

Definitions (Cont.)

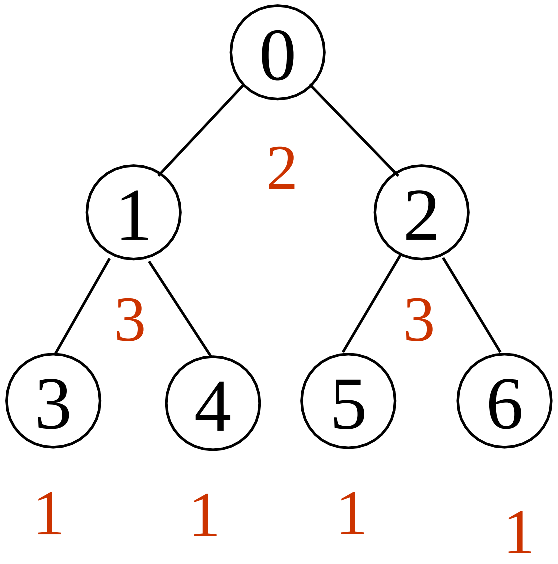
The degree of a vertex is the number of edges incident to that vertex

For directed graph,

- the in-degree of a vertex *v* is the number of edges that have *v* as the head
- the out-degree of a vertex *v* is the number of edges that have *v* as the tail

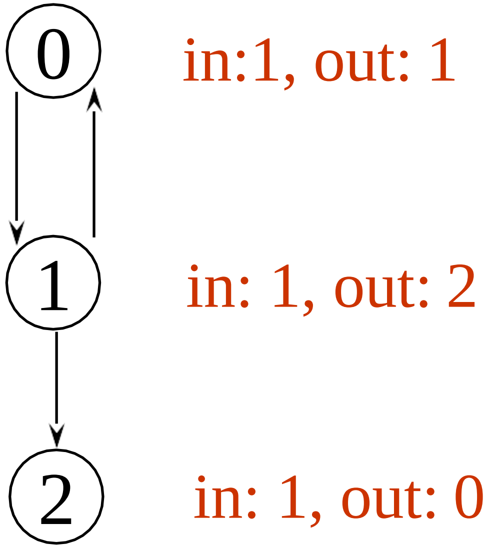


G1



G2

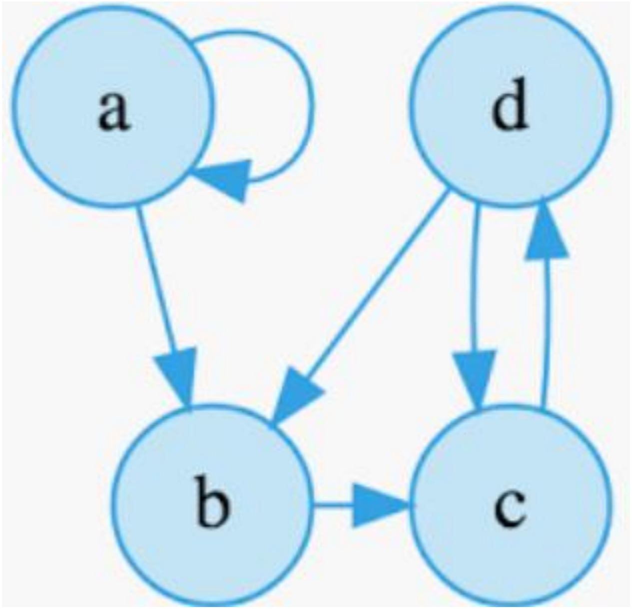
directed graph
in-degree
out-degree



G3

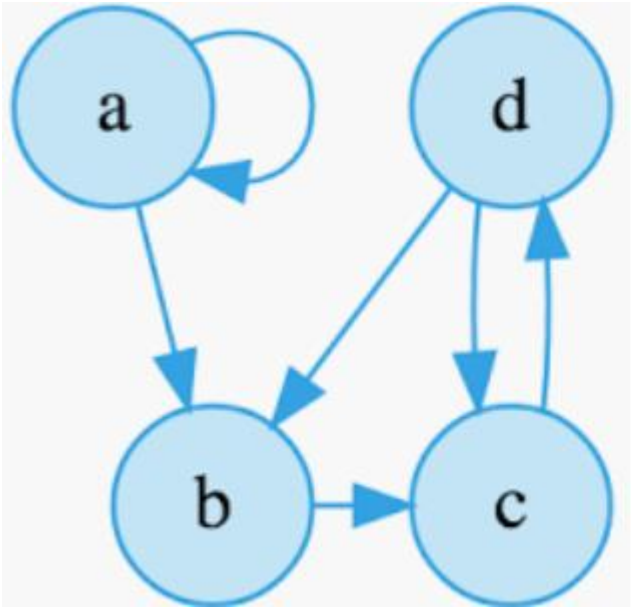
Definitions (Cont.)

- A **loop** is an edge from a vertex onto itself. It is denoted by (v, v) (e.g., (a,a)).
- A **simple path** is a path where no vertices are repeated along the path (dbc, abc).
- A **cycle** is a path with at least one edge such that the first and last vertices are the same (e.g., bcdb, cdbc, dbcd).
  - A cycle can have length one (i.e. a self loop).



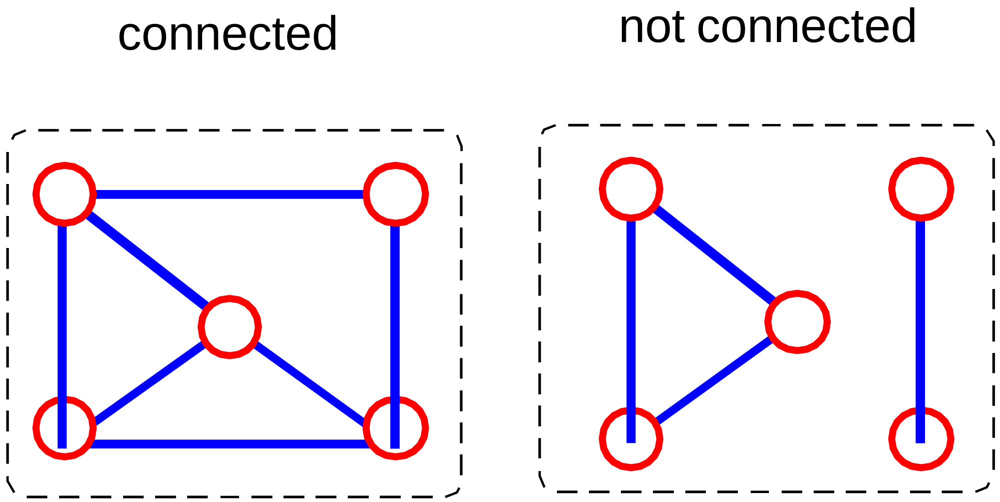
Definitions (Cont.)

- **Reachability** and **Connectivity**. A vertex v is reachable from a vertex u in G if there is a path starting at v and ending at u in G.
    - We use $R_G(v)$ to indicate the set of all vertices reachable from v in G.
    - e.g., $R_G(b) = \{c, d\}$



Definitions (Cont.)

- Connected graph: any two vertices are connected by some path
- An undirected graph is connected if all vertices are reachable from all other vertices.



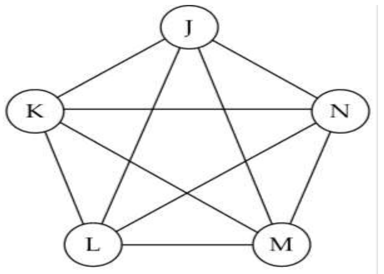
- A connected graph is strongly connected if it is a connected graph as well as a directed graph.
  - if all vertices are reachable from all other vertices.
- A connected graph is weakly connected if it is
  - a directed graph that is not strongly connected, but,
  - the underlying undirected graph is connected

Definitions (Cont.)

- Let **n** #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Undirected graph:
    - Each of the n vertices is incident to n-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = n(n -1)/2.
    - Therefore, if a graph is not complete, m < n(n -1)/2
  - Directed graph
    - m = n(n -1)
- Weighted graph: a graph in which each edge carries a value

$\mathbf{n} = 5$

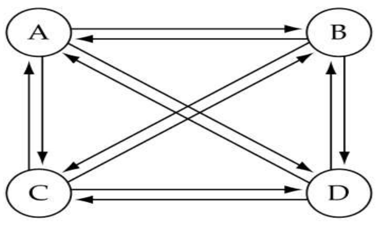
$\mathbf{m} = (5 * 4)/2 = 10$



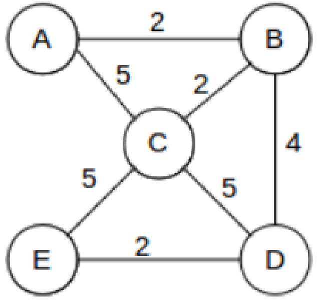
(b) Complete undirected graph.

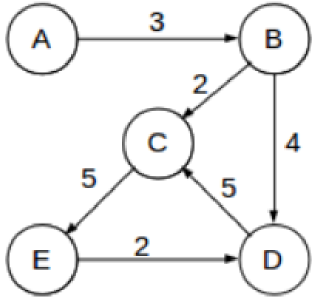
$\mathbf{n} = 4$

$\mathbf{m} = (4 * 3) = 12$



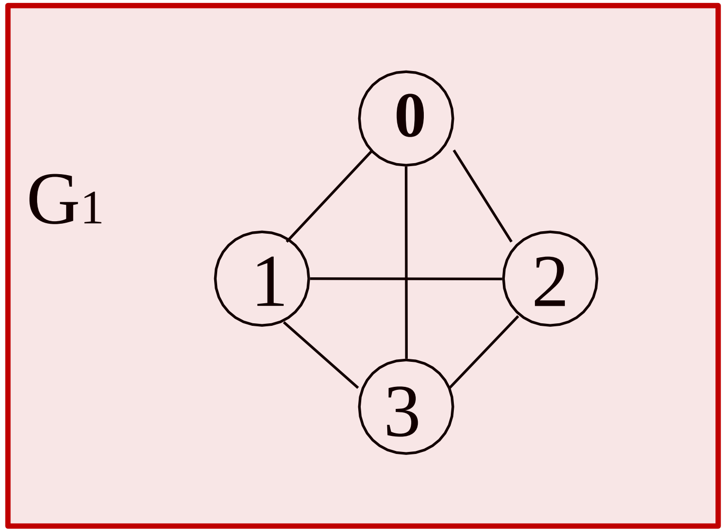
(a) Complete directed graph.

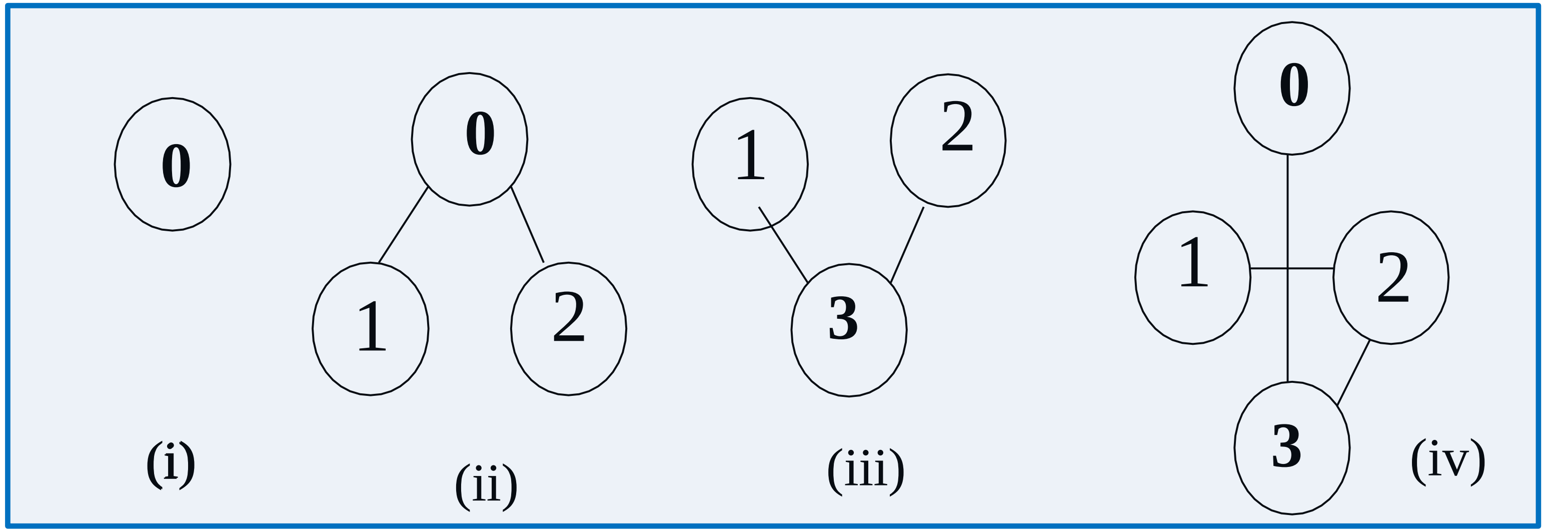




<u>Definitions</u> (Cont.)

- Subgraph: subset of vertices and edges forming a graph

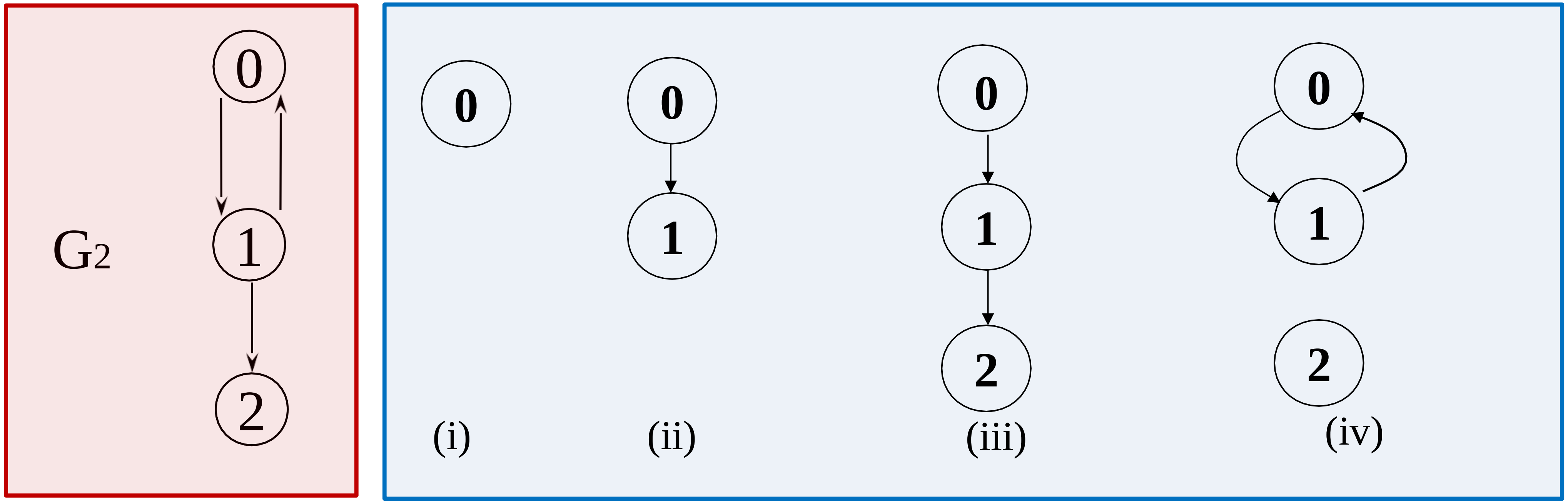




(i), (ii), (iii), and (iv) are subgraphs of G_1

Definitions (Cont.)

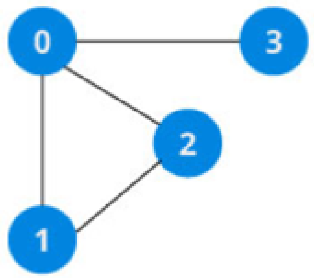
- Subgraph: subset of vertices and edges forming a graph



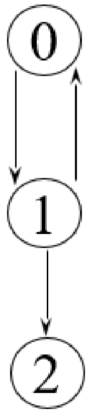
(i), (ii), (iii), and (iv) are subgraphs of G_2

Representation using a 2D Array

- A Graph can be represented by an adjacency matrix.
- An adjacency matrix is 2D array of V x V vertices. Each row and column represent a vertex.
- Adjacency matrices have a value $a_{i,j} = 1$ if nodes *i* and *j* share an edge; 0 otherwise.
  - If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a directed graph, the row sum is the out_degree, while the column sum is the in_degree.
  - In_d(v_i) = $\sum_{j=0}^{n-1} adj\_mat[j][i]$
  - Out_d(v_i) = $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- The adjacency matrix for an undirected graph is symmetric;
- The adjacency matrix for a digraph need not be symmetric.



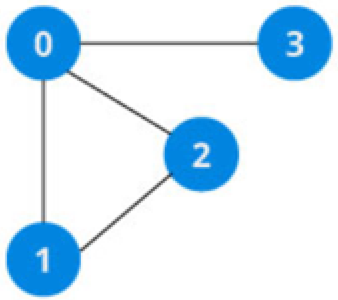
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

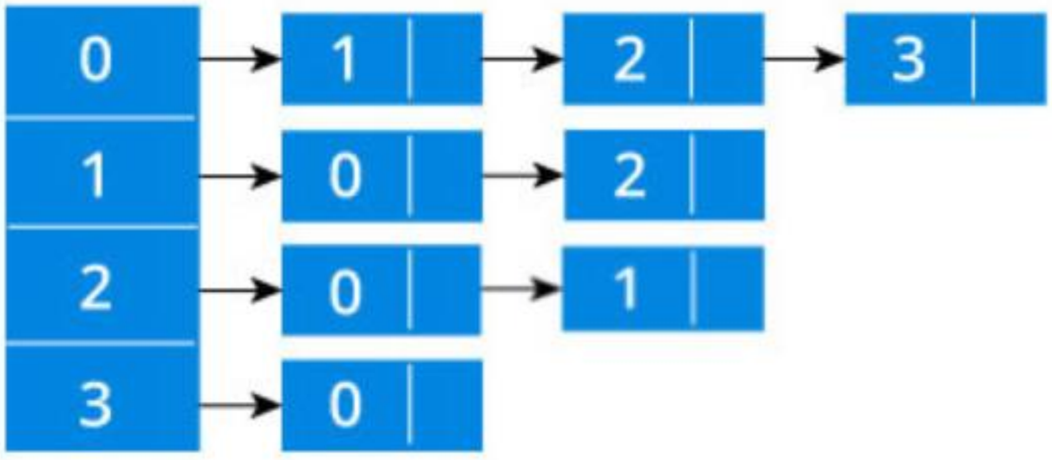


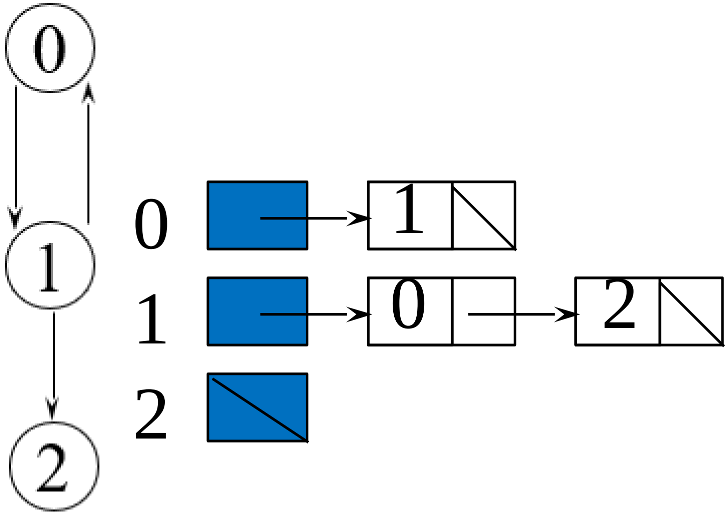
	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

Representation using a linked list

- An adjacency list represents a graph as an array of linked list.
- The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
- Degree of a vertex in an undirected graph is # of nodes in adjacency list
- If |*E*| is close to the number of vertices then G is considered **Sparse** and an adjacency list is preferred.
- If |*E*| is close to the maximum number of edges in G then it is considered **Dense** and the adjacency matrix is preferred.







GRAPH TRAVERSAL/ SEARCHING

- **How to traverse a graph or find a path between two nodes of the graph?**
  - Breadth-First-Search (BFS)
  - Depth-First-Search (DFS)

Breadth First Traversal

- Similar to traversing a tree **level-by-level**
  - Nodes at each level
    - Visited from left to right
  - All nodes at any level *i*
    - Visited before visiting nodes at level *i* + one
- Start several paths at a time, and advance in each, one step at a time
  - Expand shallowest unexpanded node

BREADTH-FIRST-TRAVERSAL

- Can be implemented efficiently using a queue

```
Input: s as the source node

BFS (G, s)
let Q be queue.
Q.enqueue( s )

mark s as visited
while ( Q is not empty)
v = Q.dequeue( )

for all neighbors w of v in Graph G
if w is not visited
Q.enqueue( w )
mark w as visited
```

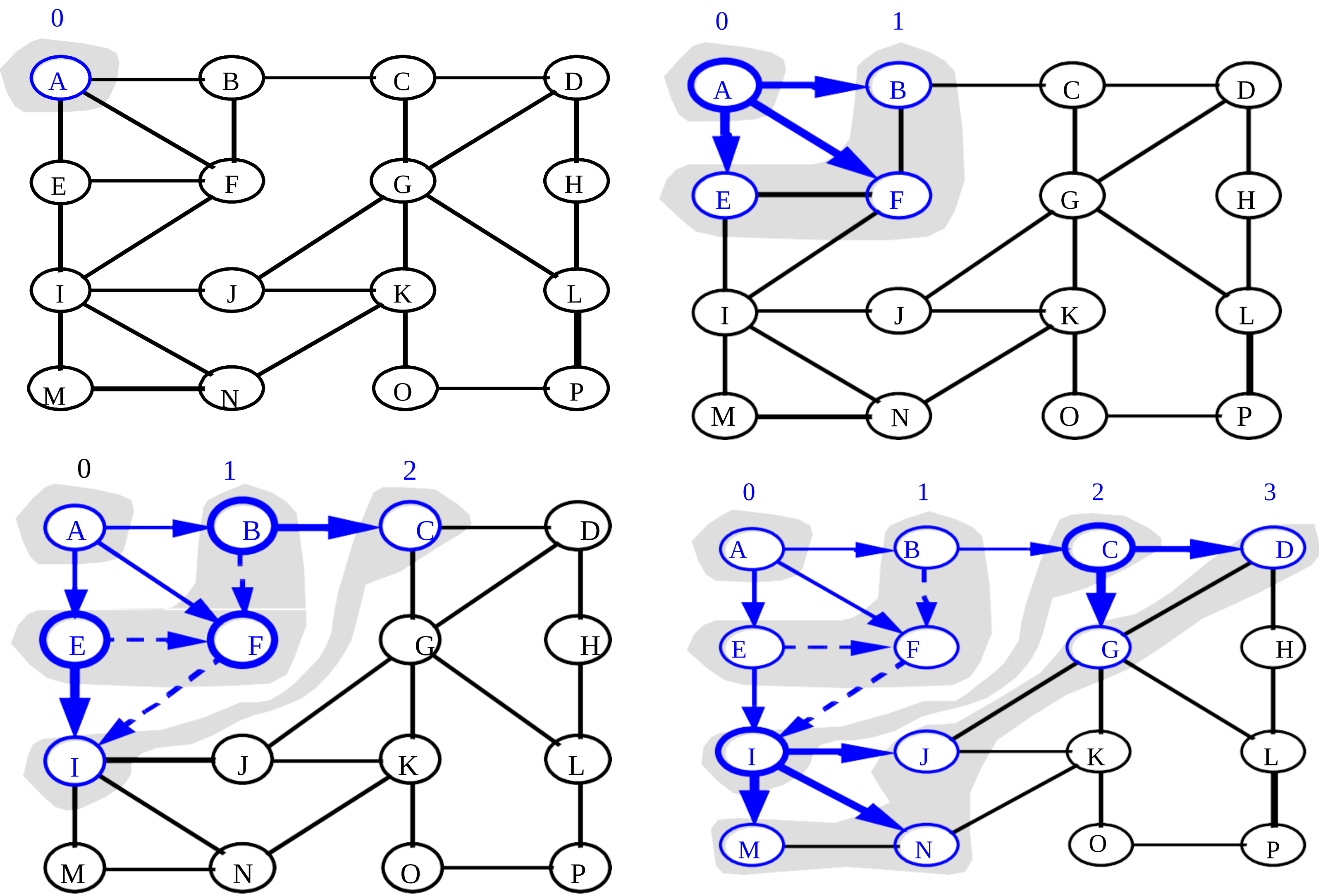
Breadth First Traversal

a)

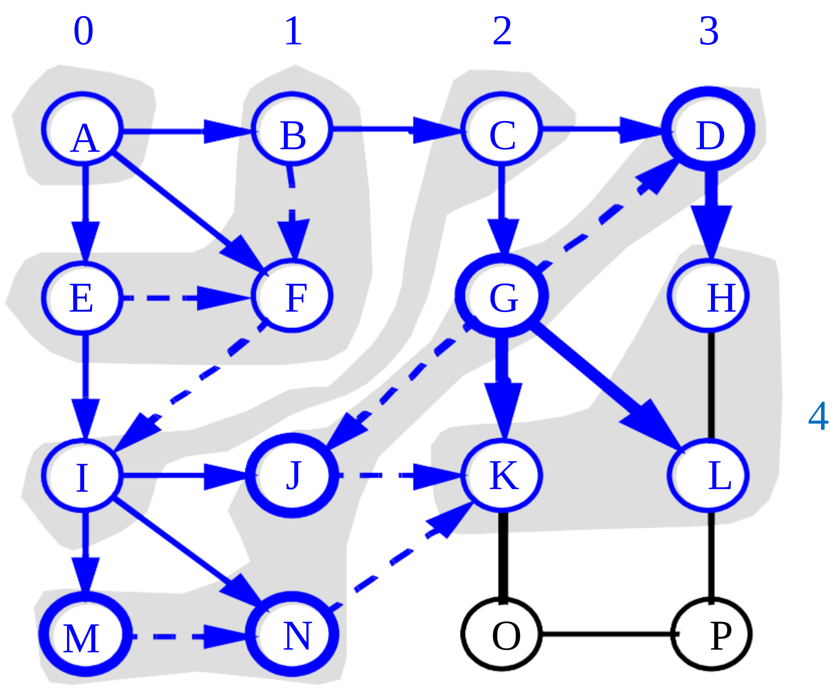
b)

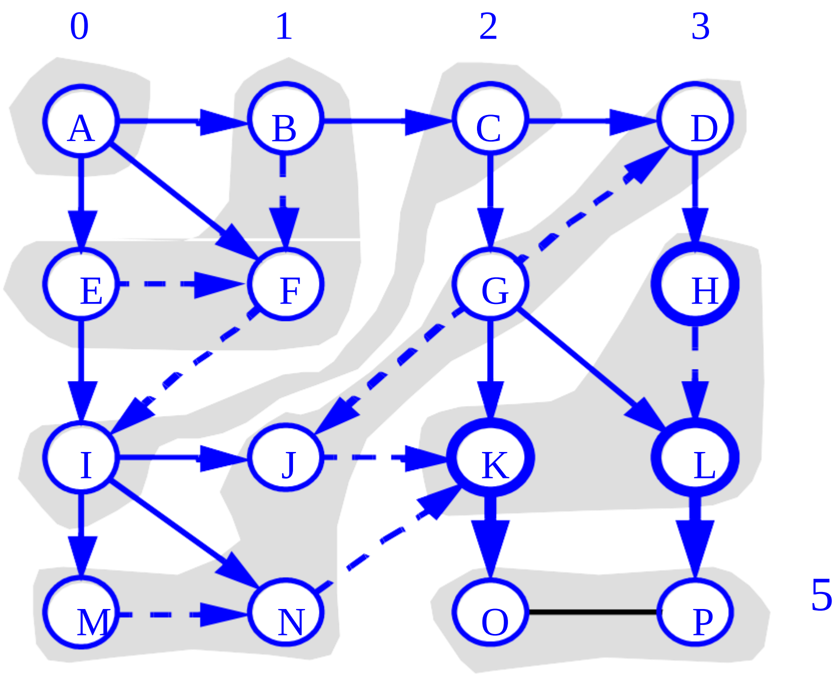
c)

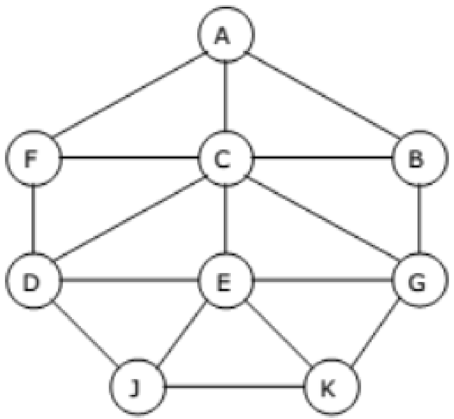
d)



Breadth First Traversal







A Graph G

Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

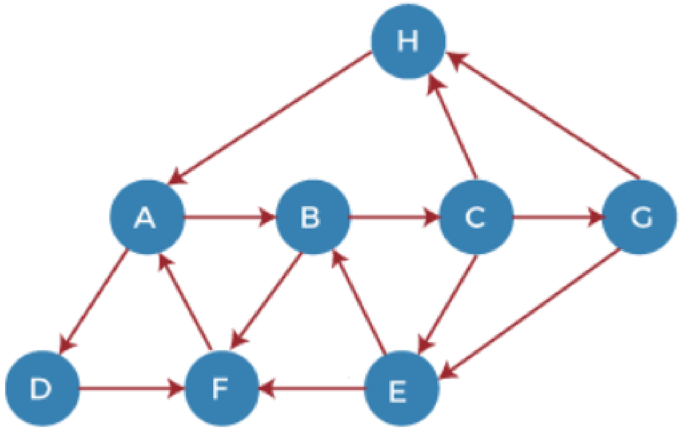
Adjacency list for graph G

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: ***A F C B D E G J K***.

Breadth First Traversal

- Example



Traversal starting from Node H.

HABDCFEG

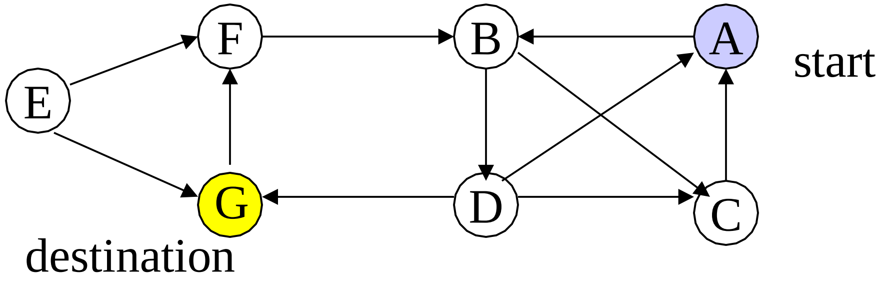
BREADTH-FIRST-SEARCH (BFS)

- BFS can be implemented efficiently using a queue

BFS-iterative

```
Set found to false
Enqueue(startVertex)
Do
     Dequeue(vertex)
     IF vertex == endVertex
             Set found to true
     ELSE
             Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
IF(!found)
      Write "Path does not exist"
```

BREADTH-FIRST-SEARCH (BFS)



rear front

A

Add A to queue

rear front

B

Dequeue A
Add B

rear front

D C

Dequeue B
Add C, D

rear front

D

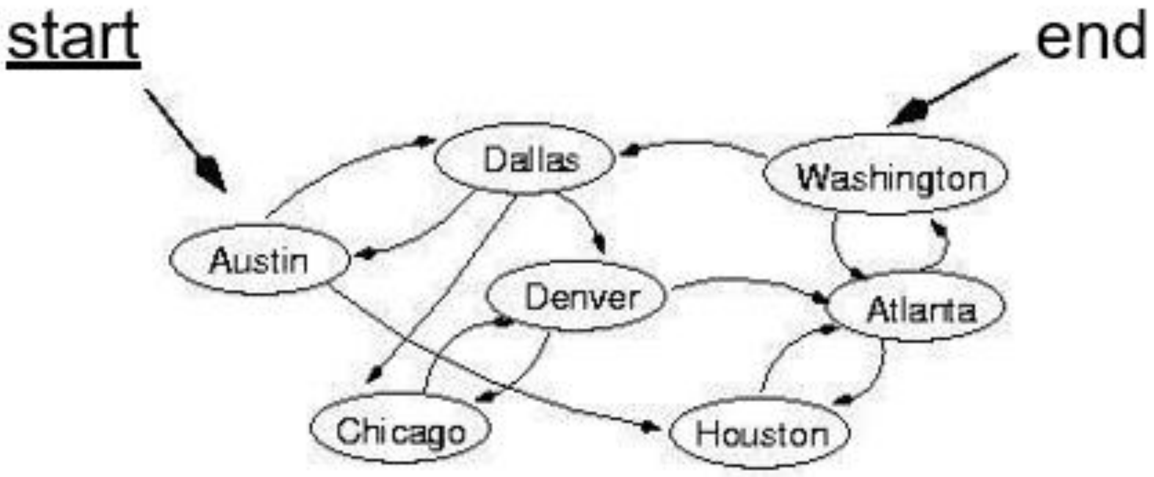
Dequeue C
Nothing to add

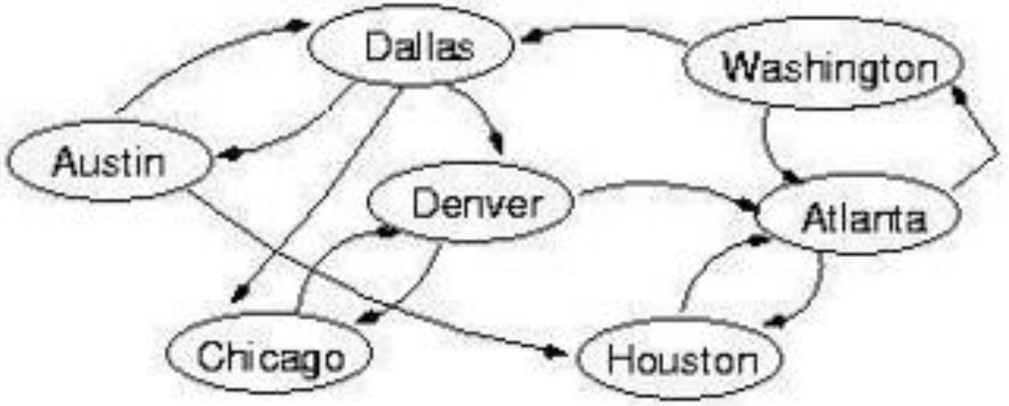
rear front

G

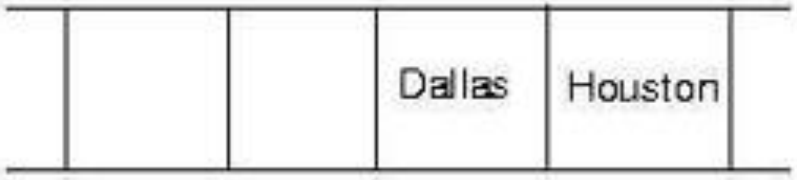
Dequeue D
Add G

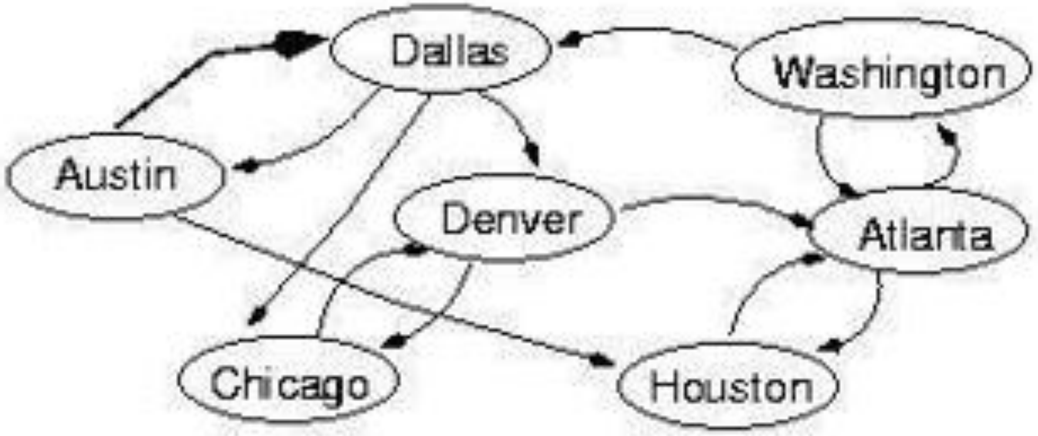
found destination - done!





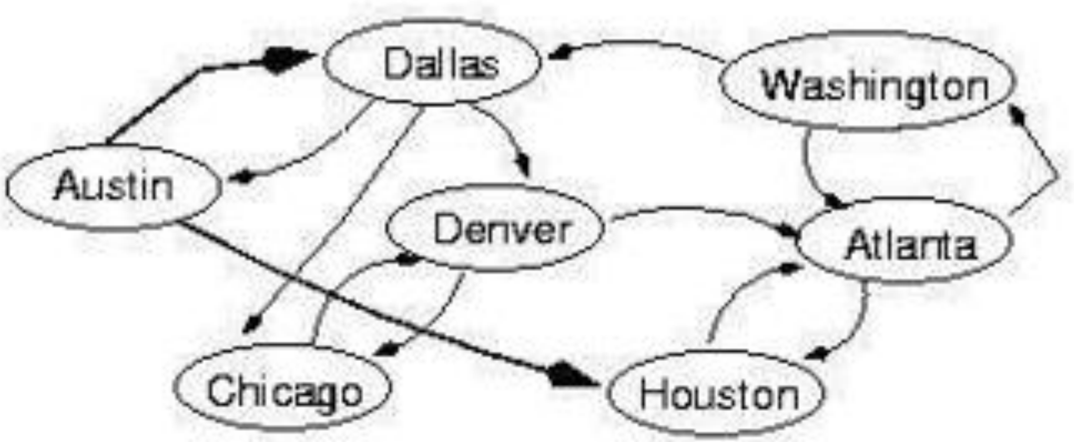
dequeue Austin





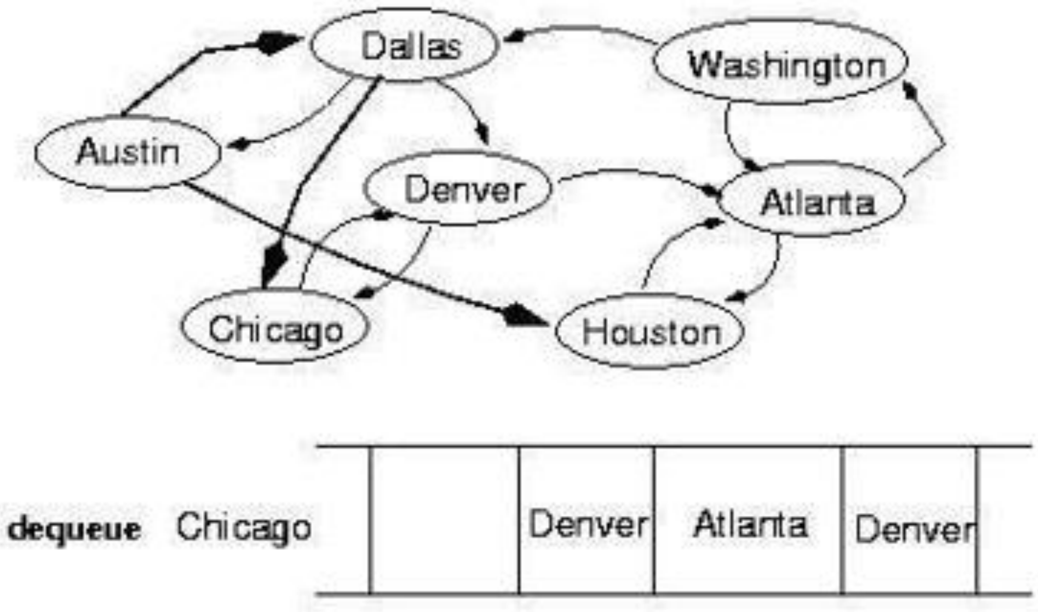
dequeue Dallas

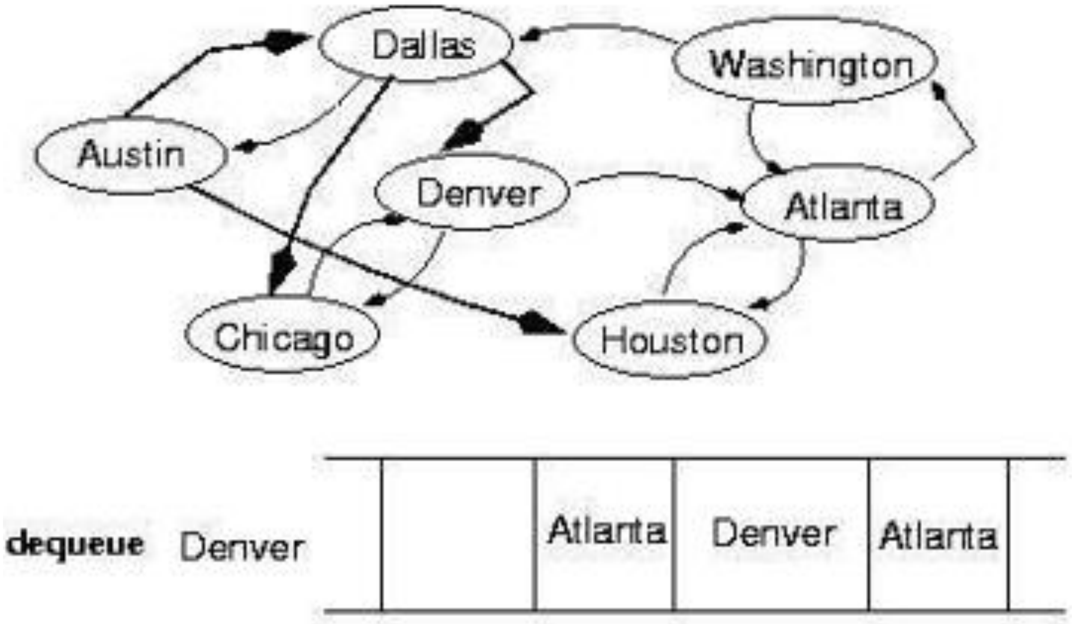
	Houston	Chicago	Denver

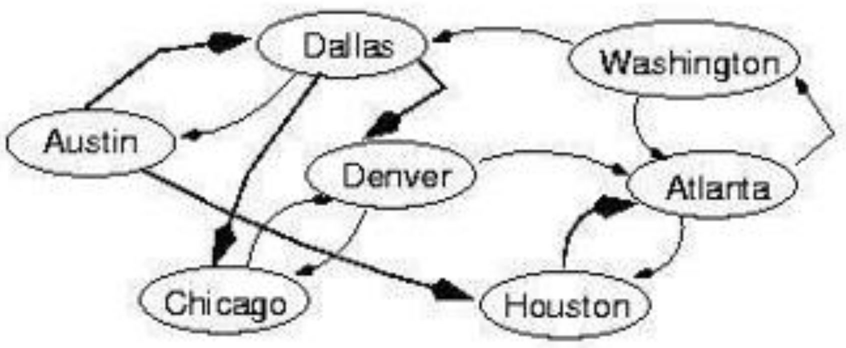


dequeue Houston

	Chicago	Denver	Atlanta

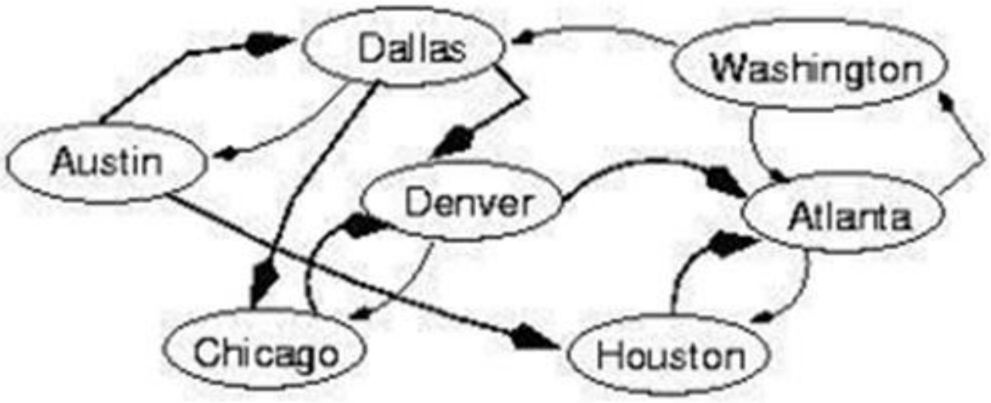

Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
dequeue
Chicago
Denver
Atlanta
Denver


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
dequeue
Denver
Atlanta
Denver
Atlanta



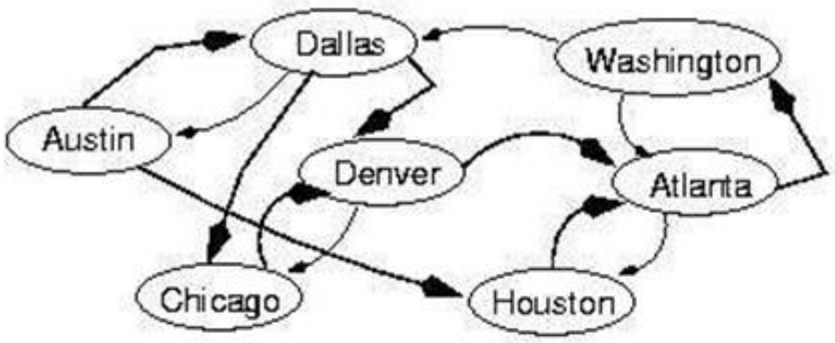
dequeue Atlanta

	Denver	Atlanta	Washington



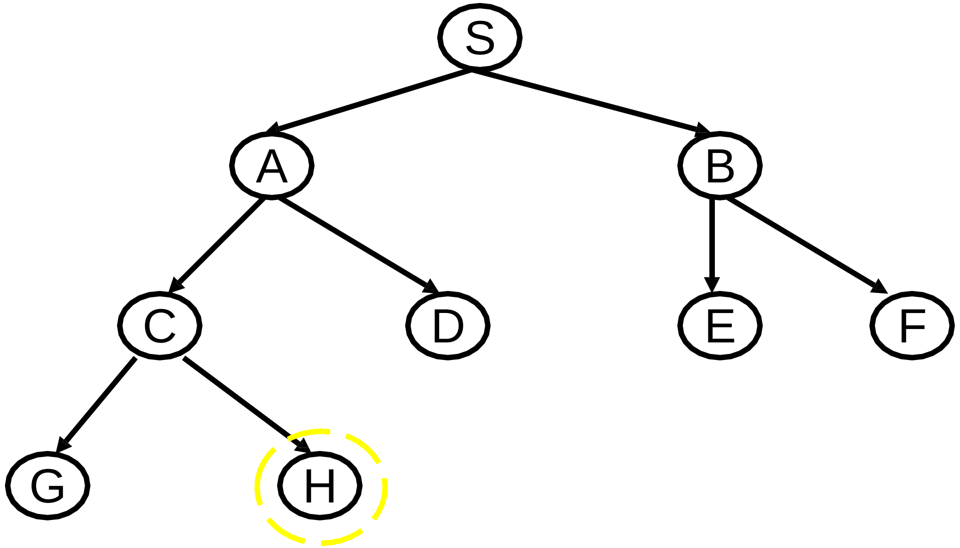
dequeue Denver, Atlanta

	Washington



dequeue Washington

BFS: EXAMPLE



Q

1

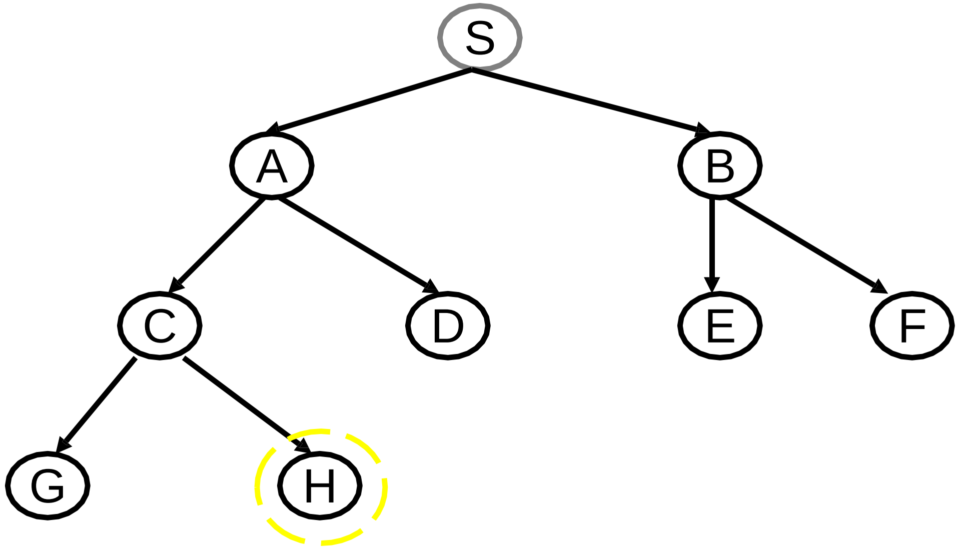
2

3

4

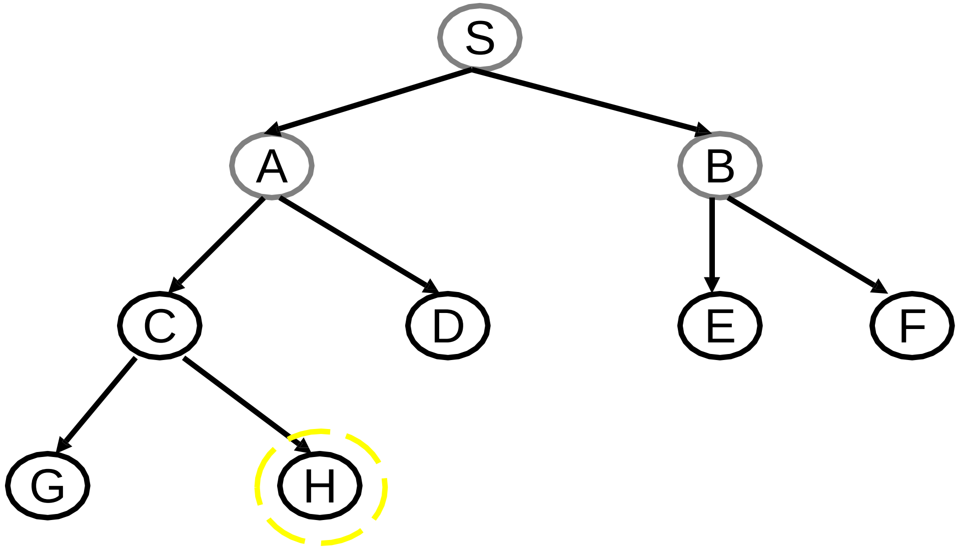
5

BFS: EXAMPLE



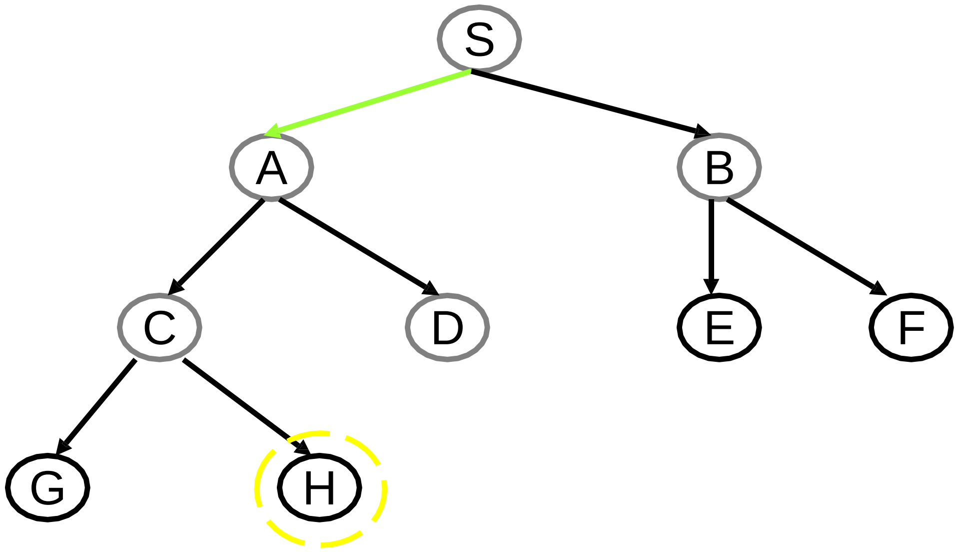
	Q
1	S
2	
3	
4	
5	

BFS: EXAMPLE



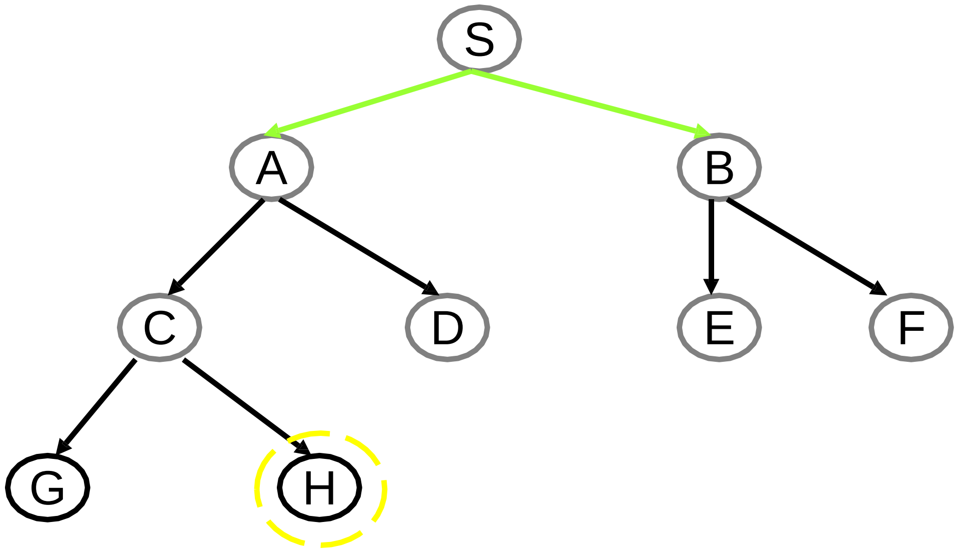
	Q
1	S
2	A,B
3	
4	
5	

BFS: EXAMPLE



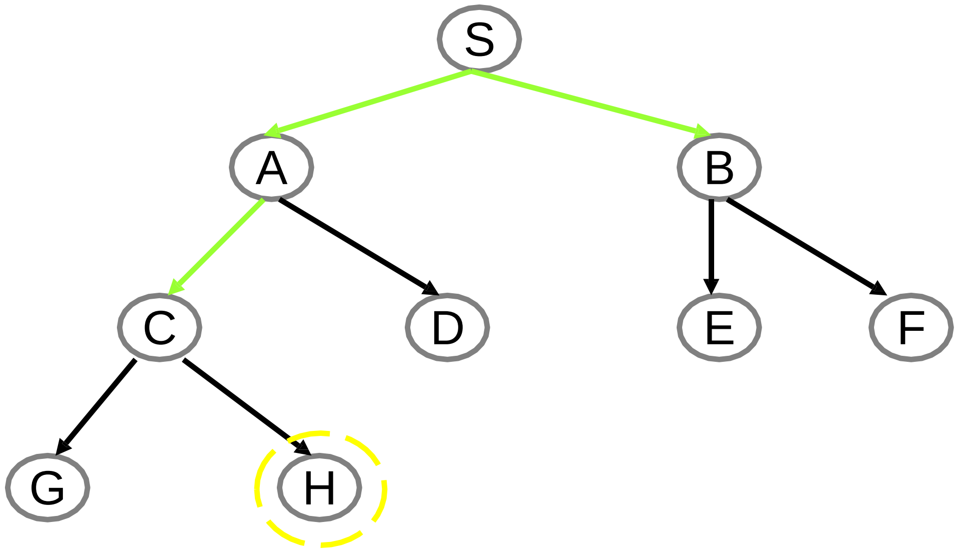
	Q
1	S
2	A,B
3	B,C,D
4	
5	

BFS: EXAMPLE



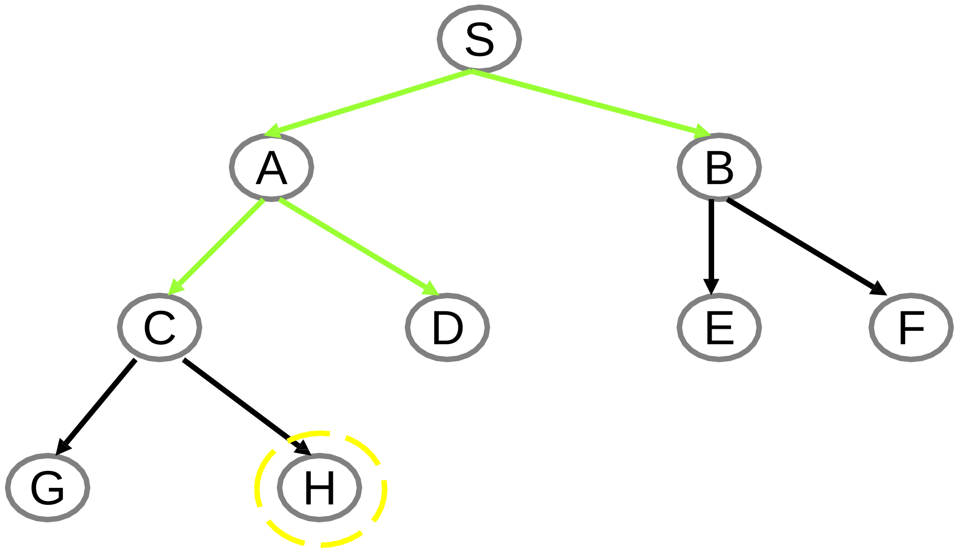
	Q
1	S
2	A,B
3	B,C,D
4	C,D,E,F
5	

BFS: EXAMPLE



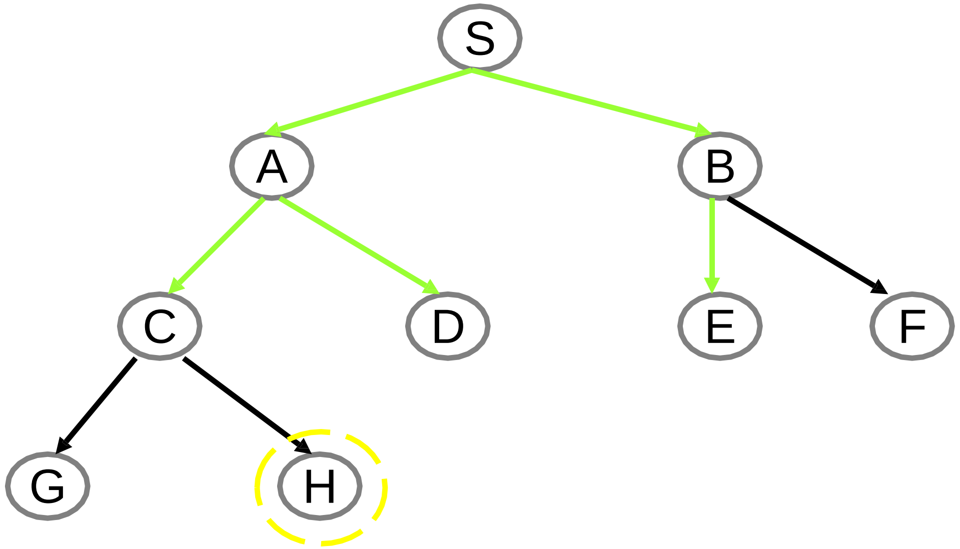
	Q
2	A,B
3	B,C,D
4	C,D,E,F
5	D,E,F,G,H
6	

BFS: EXAMPLE



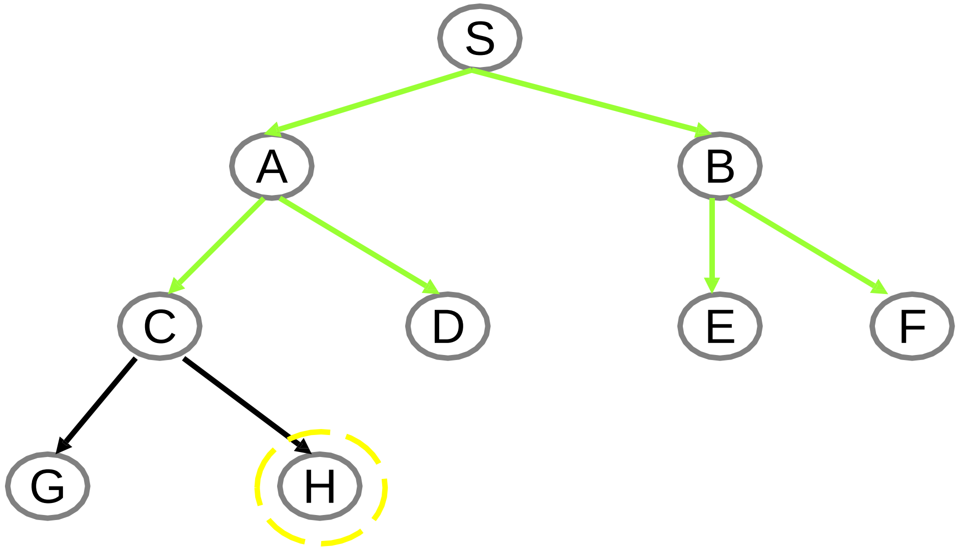
	Q
3	B,C,D
4	C,D,E,F
5	D,E,F,G,H
6	E,F,G,H
7	

BFS: EXAMPLE



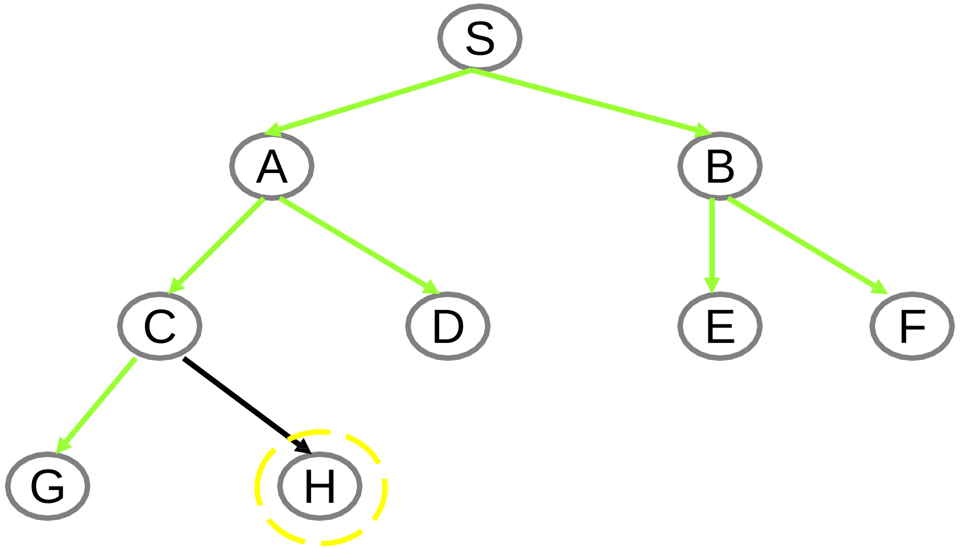
	Q
4	C,D,E,F
5	D,E,F,G,H
6	E,F,G,H
7	F,G,H
8	

BFS: EXAMPLE



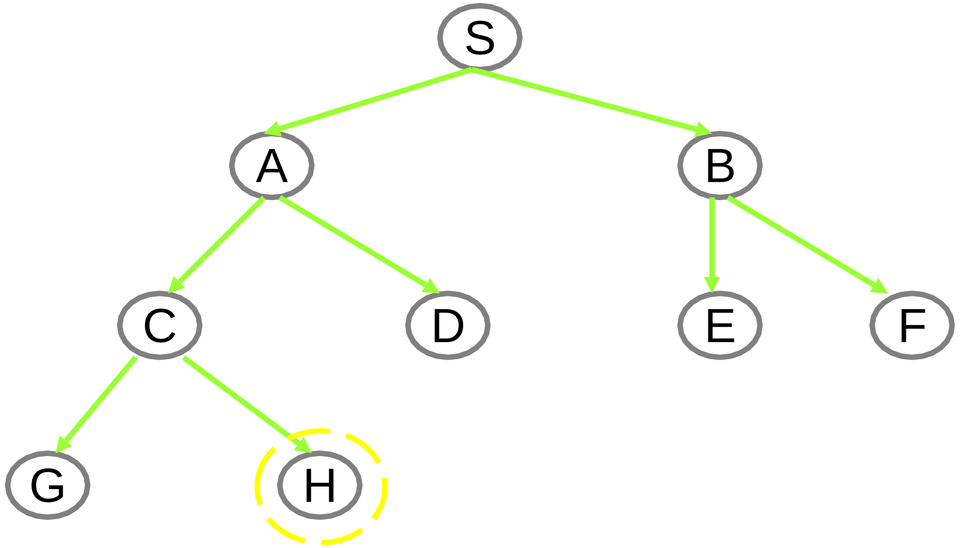
	Q
5	D,E,F,G,H
6	E,F,G,H
7	F,G,H
8	G,H
9	

BFS: EXAMPLE



	Q
6	E,F,G,H
7	F,G,H
8	G,H
9	H
10	

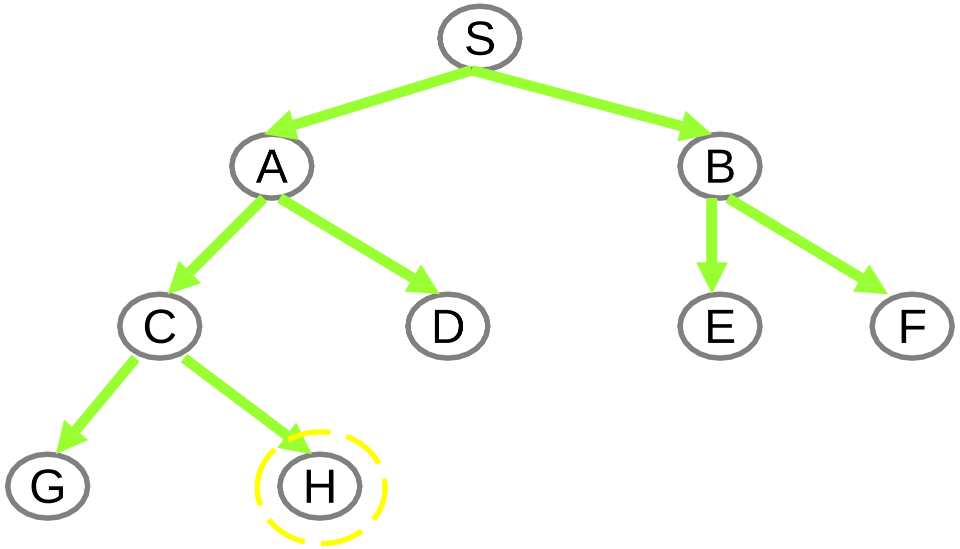
BFS: EXAMPLE



	Q
6	E,F,G,H
7	F,G,H
8	G,H
9	
10	

BFS: EXAMPLE

	Q
1	S
2	A,B
3	B,C,D
4	C,D,E,F
5	D,E,F,G,H
6	E,F,G,H
7	F,G,H
8	G,H
9	H
10	



Depth First Traversal

- Similar to binary tree preorder traversal
- Once a possible path is found, continue the search until the end of the path
- 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)
  - Expand deepest unexpanded node
- General algorithm for depth first traversal at a given node *v (recursive)*

```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

Depth First Traversal

- General algorithm for depth first traversal at a given node *v (recursive)*

```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

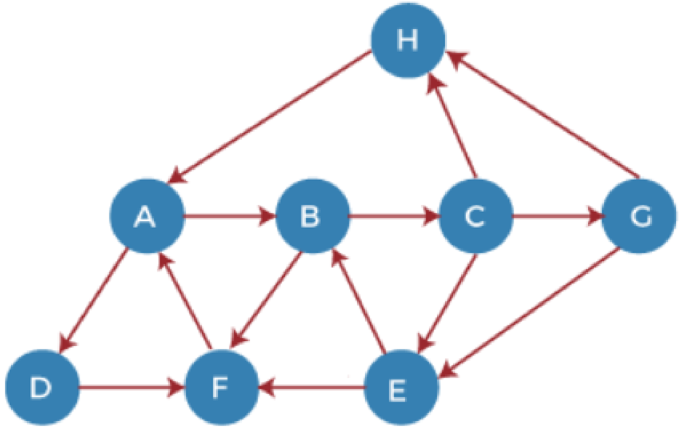
Depth First Traversal

- General algorithm for depth first traversal at a given node *v (non-recursive)*

```
DFS(G,v)   ( v is the vertex where the search starts )
       Stack S := {};   ( start with an empty stack )
       for each vertex u, set visited[u] := false;
       push S, v;
       while (S is not empty) do
          u := pop S;
          if (not visited[u]) then
             visited[u] := true;
             for each unvisited neighbour w of uu
                push S, w;
          end if
       end while
    END DFS()
```

Depth First Traversal

- Example

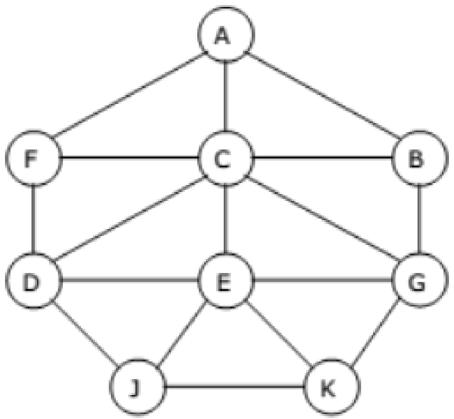


Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Traversal starting from Node H.

HADFBCGE



A Graph G

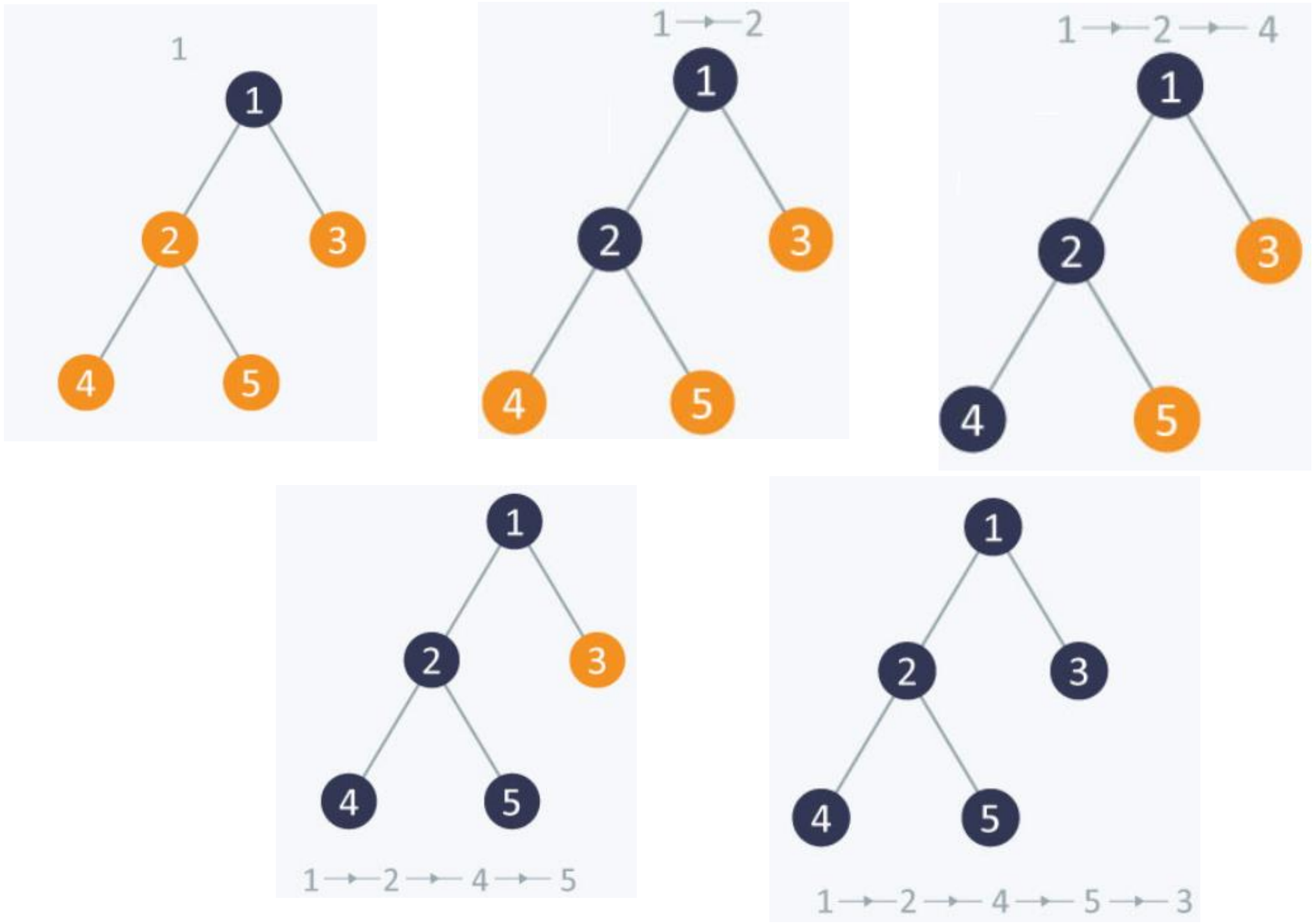
Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

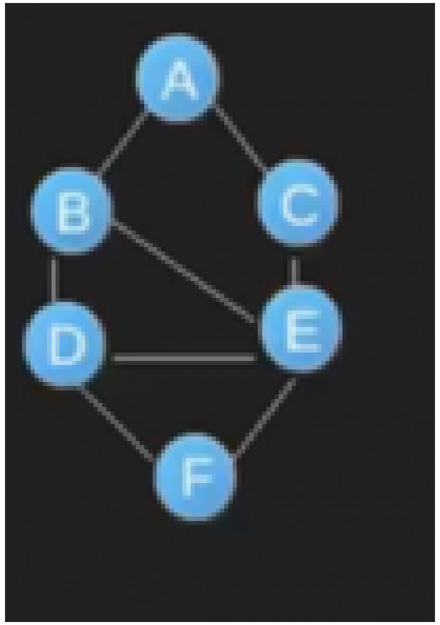
Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

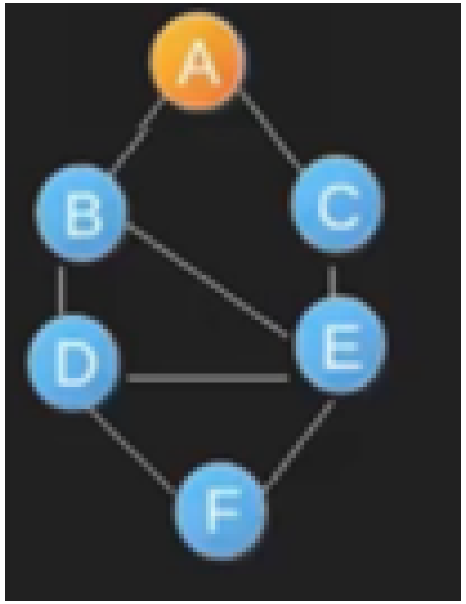
For the above graph the depth first traversal sequence is: ***A F D J K G E C B***.

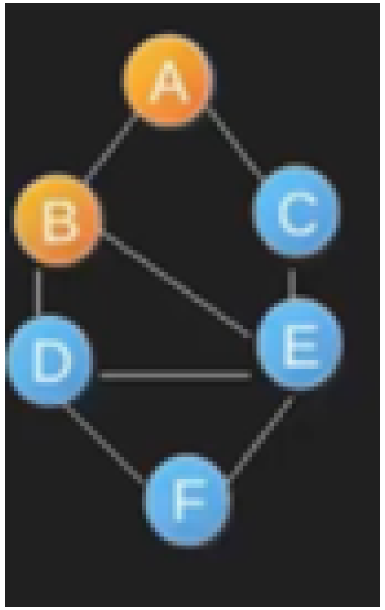
Depth First Traversal

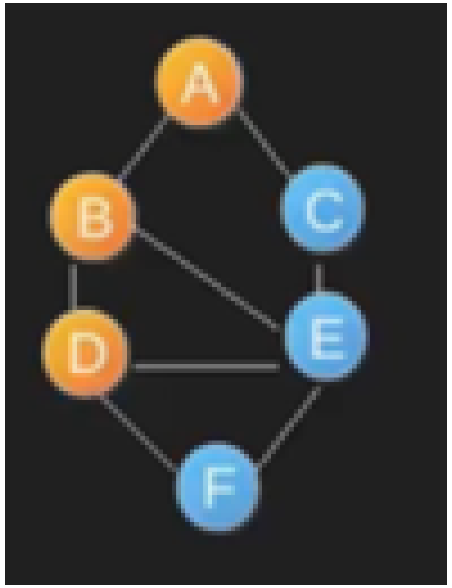


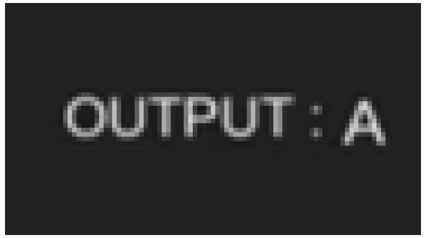
Depth First Traversal



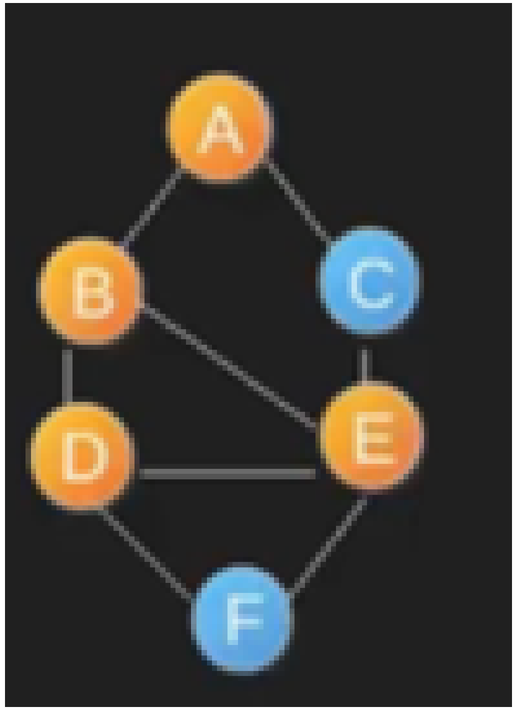




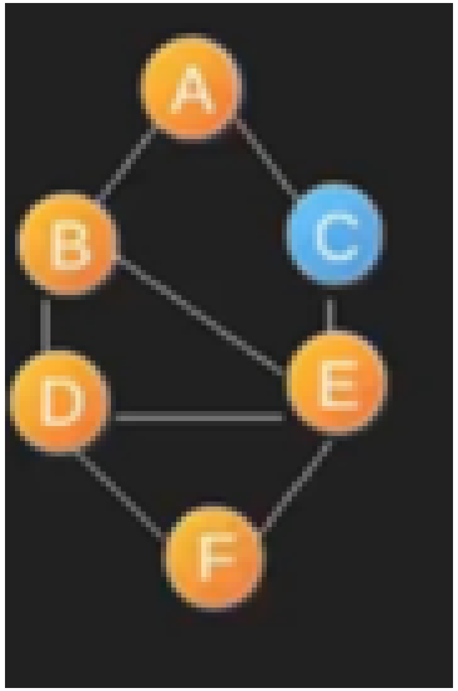




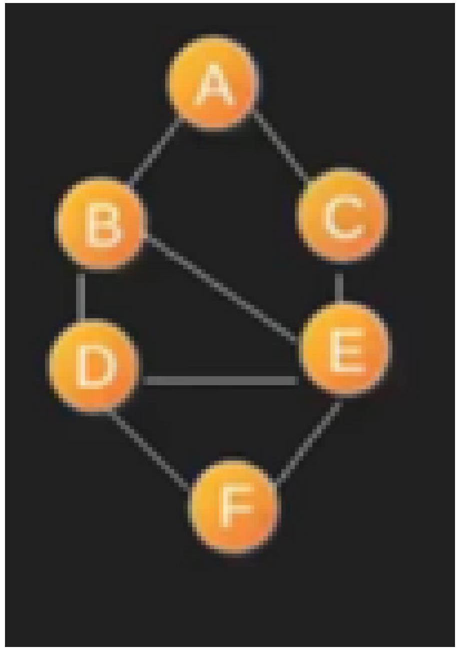
Depth First Traversal



OUTPUT : A B D E



OUTPUT : A B D E F



OUTPUT : A B D E F C

DEPTH-FIRST-SEARCH (DFS)

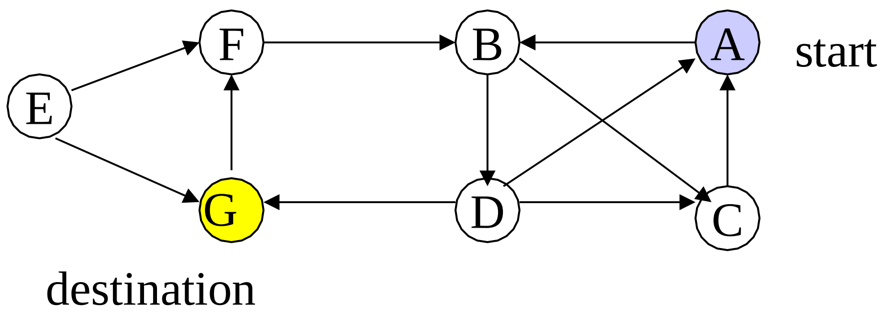
- DFS can be implemented efficiently using a *stack*

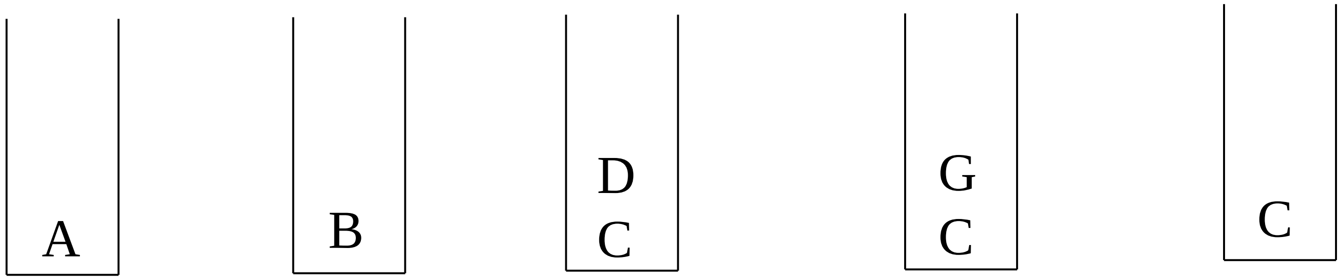
DFS-iterative

```
Set found to false
  Push(startVertex)
  DO
    Pop(vertex)
    IF vertex == endVertex
    Set found to true
    ELSE
      Push all adjacent vertices onto stack
  WHILE !stack.IsEmpty() AND !found

  IF(!found)
    Write "Path does not exist"
```

DEPTH-FIRST-SEARCH (DFS)





Push A

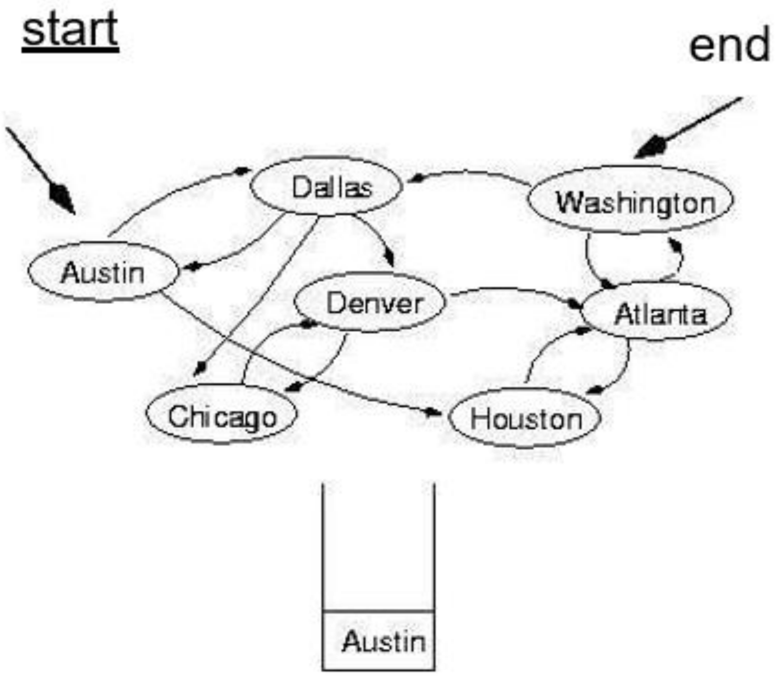
Pop A
Push B

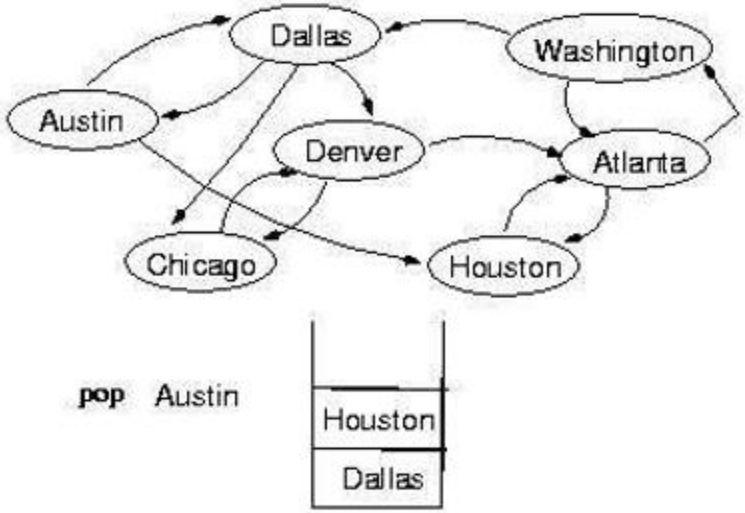
Pop B
Push C
Push D

Pop D
Push G

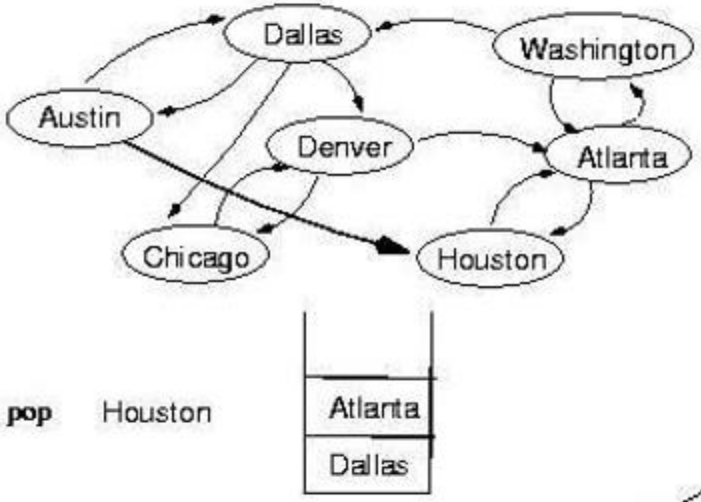
Pop G
found destination - done!

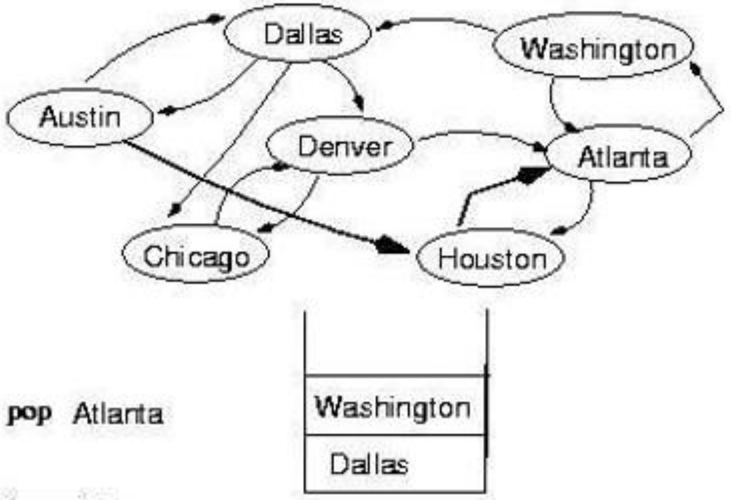
DEPTH-FIRST-SEARCH (DFS)

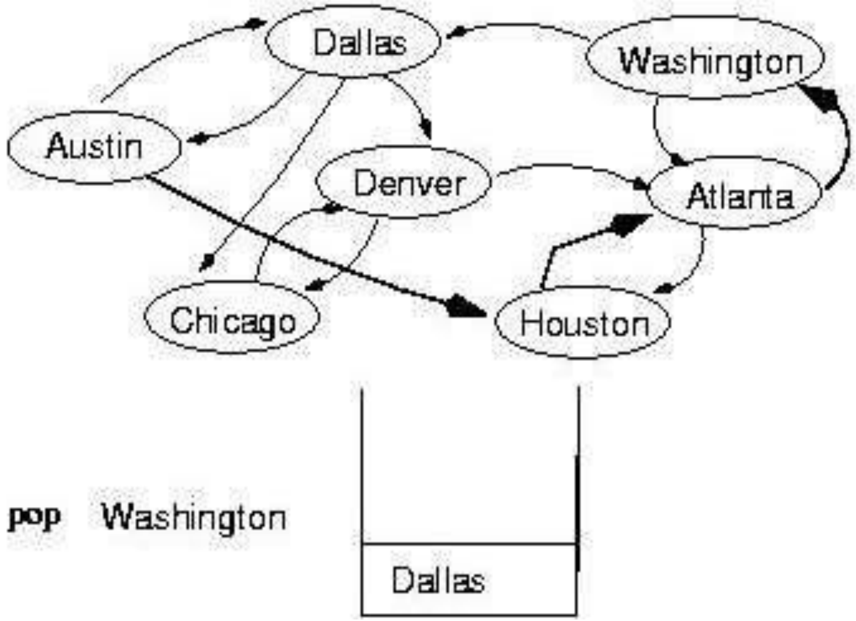




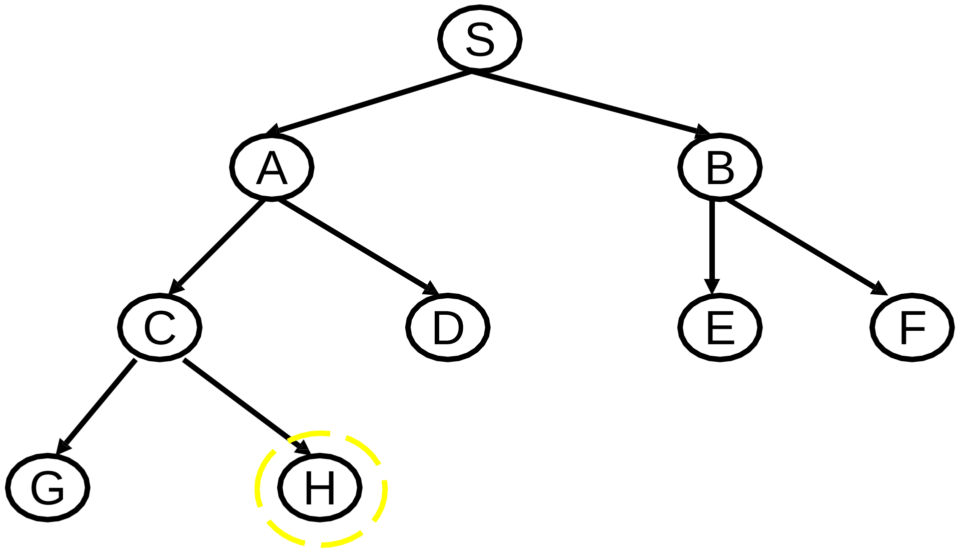
DEPTH-FIRST-SEARCH (DFS)







DFS: EXAMPLE



Q

1

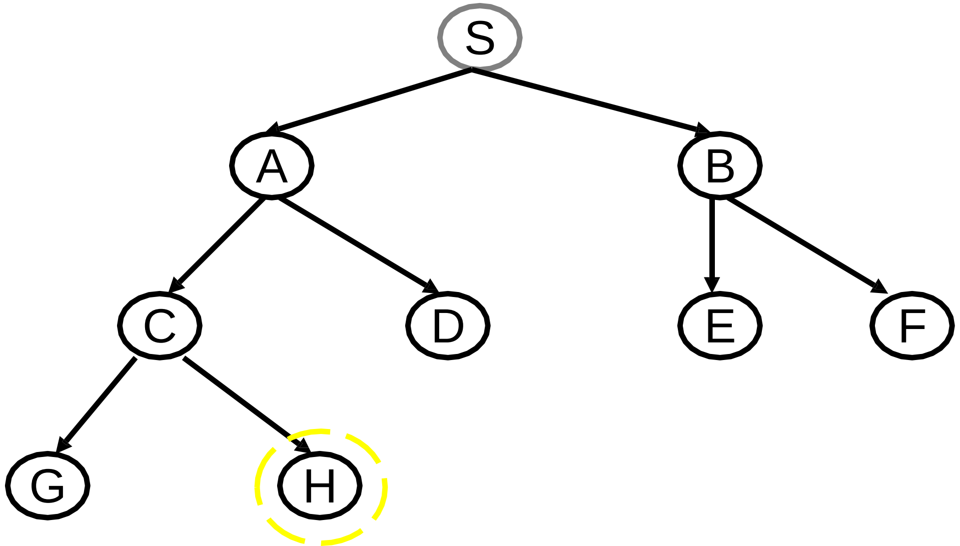
2

3

4

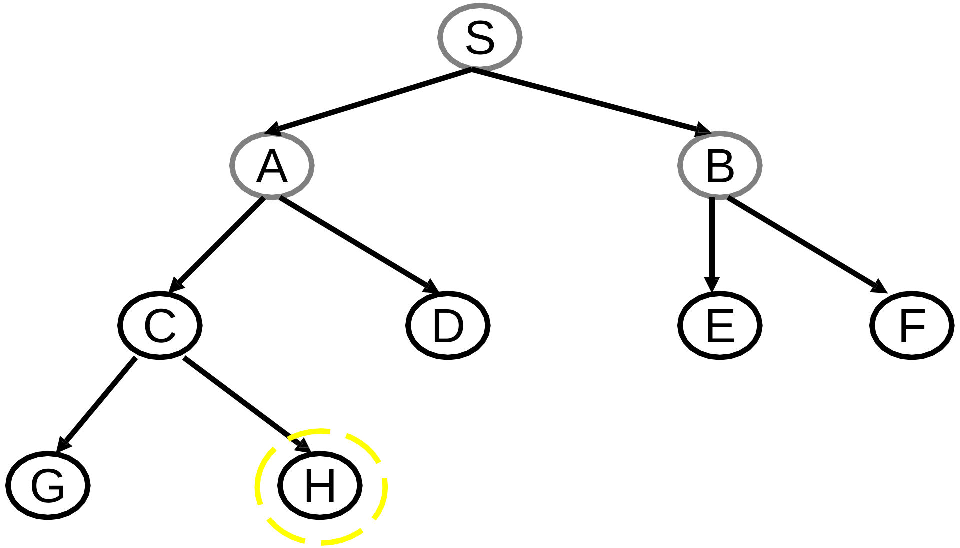
5

DFS: EXAMPLE



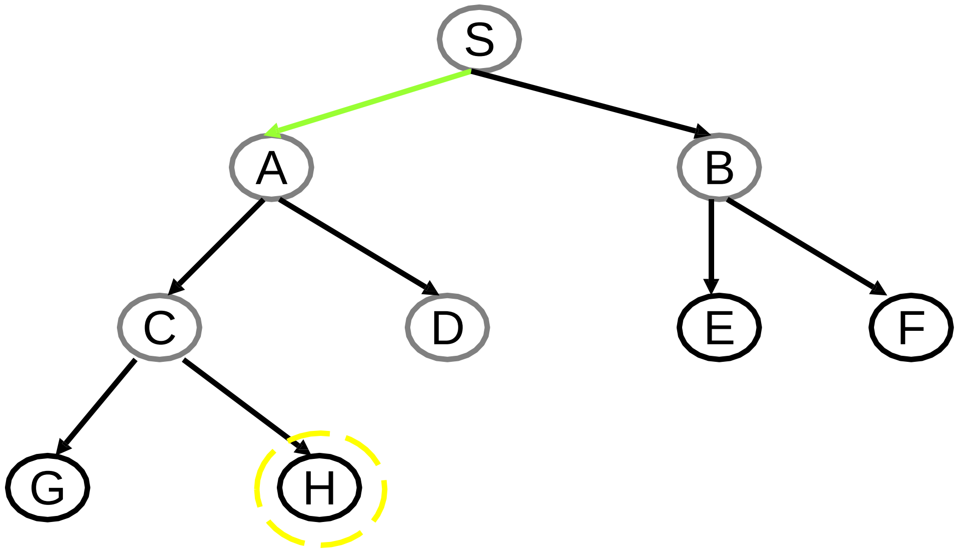
	Q
1	S
2	
3	
4	
5	

DFS: EXAMPLE



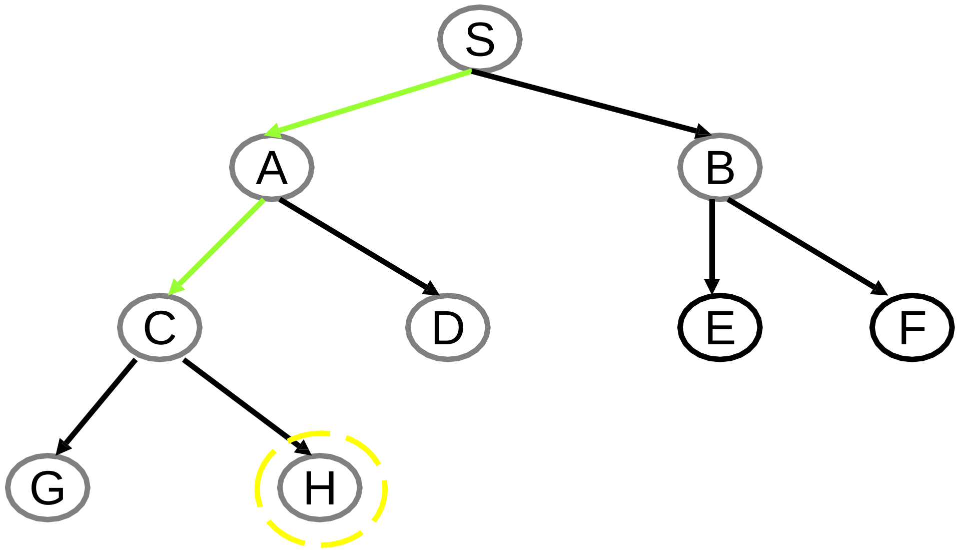
	Q
1	S
2	A,B
3	
4	
5	

DFS: EXAMPLE



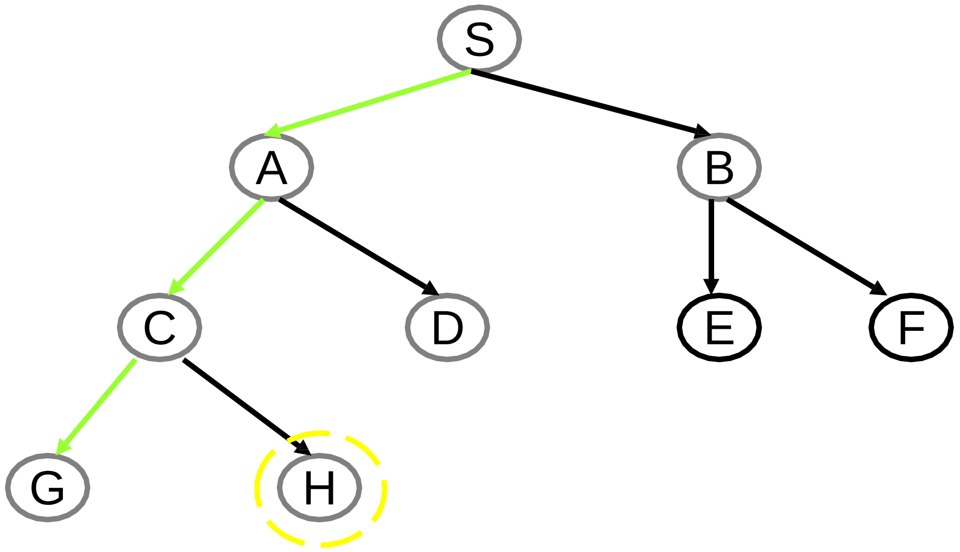
	Q
1	S
2	A,B
3	C,D,B
4	
5	

DFS: EXAMPLE



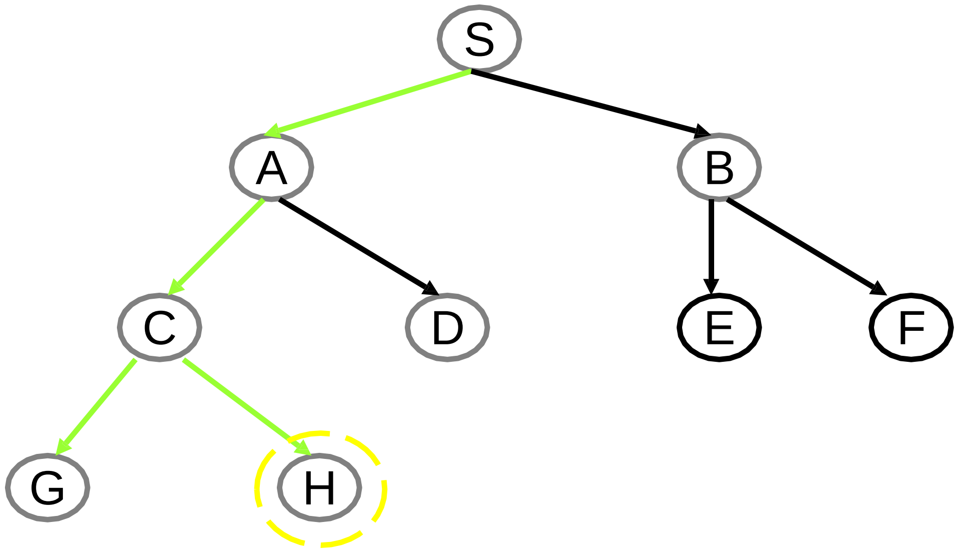
	Q
1	S
2	A,B
3	C,D,B
4	G,H,D,B
5	

DFS: EXAMPLE



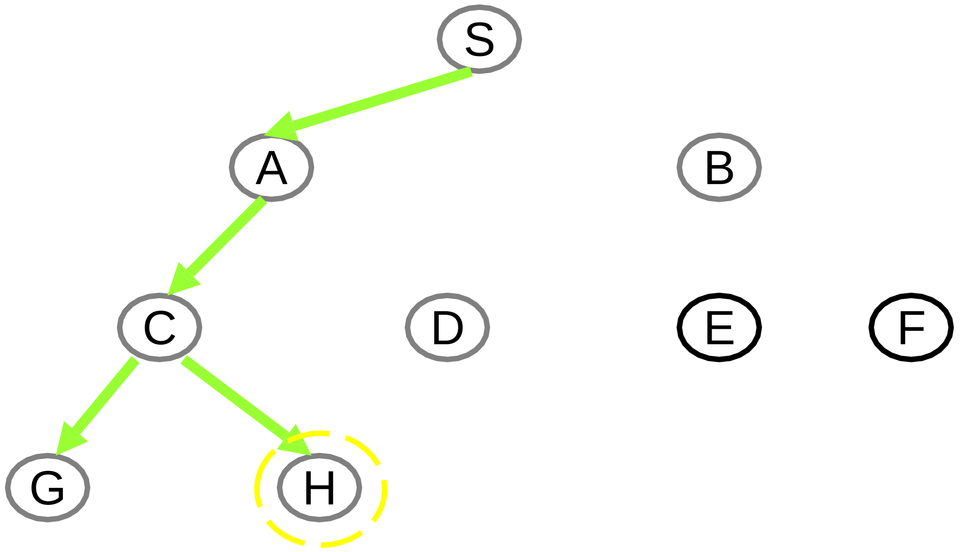
	Q
1	S
2	A,B
3	C,D,B
4	G,H,D,B
5	H,D,B

DFS: EXAMPLE



	Q
1	S
2	A,B
3	C,D,B
4	G,H,D,B
5	H,D,B
6	D,B

DFS: EXAMPLE



	Q
1	S
2	A,B
3	C,D,B
4	G,H,D,B
5	H,D,B
6	D,B