Distributed Algorithms for Computer Networks

Chapter 5

Spanning Tree Construction

Dr.Ahmed Awad

Dr. Amjad Hawash

Introduction

- **Spanning Tree**: The maximal set of edges of a graph that contains no cycles.

➔ The minimal set of edges that connect all vertices.

- Why spanning trees in computer networks?
  - They provide a subtree with possibly less communication links, resulting in lowered communication costs.

- Providing a parent/child relationship among the nodes of the network eases the task of communication since the source and the destination of the communication is known beforehand.

The Flooding Algorithm

- **Broadcast:** Sending a message to all nodes in a computer network.
- **Flooding**: Forward any incoming message to all neighbors except the neighbor that has sent the message.
- If the same message arrives again, it should be discarded.
- The node that initiates the message to be broadcasted is called the **root**.

The Flooding Algorithm (2)

Algorithm 4.1 *Flood*

```
 1: int i, j
 2: boolean visited ← false
 3: message types msg
 4:
 5: if i = root then                          ▷ root initiates flooding
 6:     send msg to Γ(i)
 7:     visited ← true
 8: end if
 9:
10: receive flood(j)                          ▷ flood may be received many times
11: if visited = false then                   ▷ msg received first time
12:     send msg to Γ(i) \ {j}
13:     visited ← true
14: else                                      ▷ msg received before
15:     discard msg
16: end if
```

The Flooding Algorithm : Complexity Analysis

- **Message Complexity:**
  - Each edge connects two nodes.
  - Each edge is used to deliver a message at least once and at most twice when two nodes send messages concurrently.
  - Total number of messages =*2m* where *m* represents the number of edges in the graph.
  - ➔ Message complexity: ***O(2m)=O(m)***
- **Time Complexity:**
  - The longest time for the broadcast message to reach any node in the graph is the distance between the two farthest nodes of the graph.
  - ➔ Time complexity: $\Theta(d)$ *d:* The diameter of the graph.

Flooding-Based Asynchronous Spanning Tree Construction Algorithm (Flood_ST)

- **Objective:** Build spanning tree originating from the initiator root for broadcasting. This tree can be used for any further broadcast messages.

- **Assumption:** Each node in the tree except the leaf nodes should know the identifiers of its children and all nodes except the root should know their parents.

- Any node that wants to build a broadcast tree initiates the algorithm and becomes the root of the spanning tree to be formed.

Flood_ST

- **Types of messages:**
  - Probe.
  - Ack.
  - Reject

- Any node that wants to build a spanning tree starts the algorithm by sending **probe** message to its neighbors.

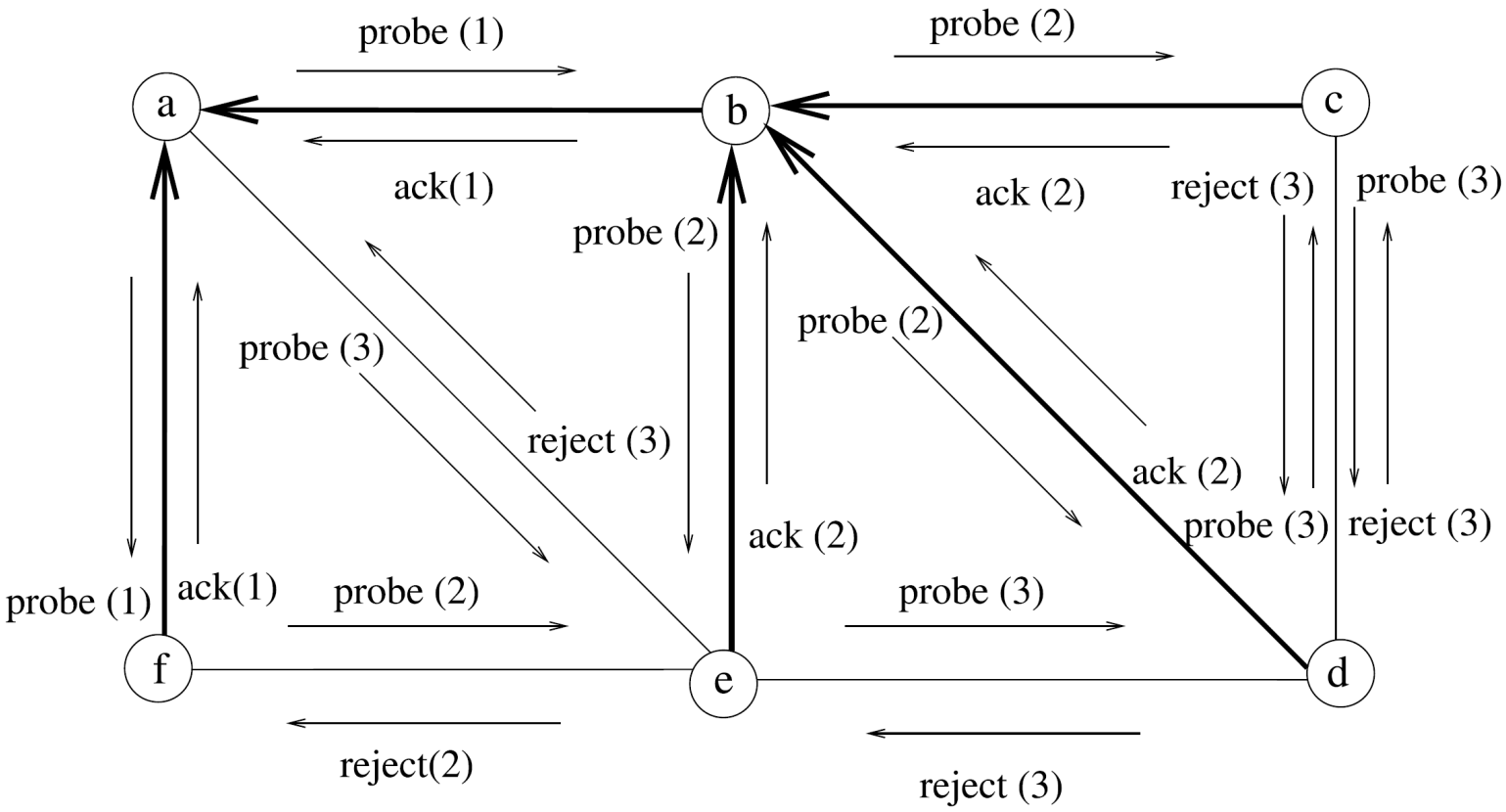
- When a node receives a probe message:
  - it sends a positive acknowledgement to the source node (which becomes a parent) in case it doesn’t have a parent.
  - It transfers the probe message to all of its neighbors except the parent.
  - If a node has a parent, it sends a reject message to source node of the probe message.

Flood_ST Algorithm

Algorithm 4.2 *Flood_ST*

```
 1: int parent ← ⊥
 2: set of int childs ← ∅, others ← ∅
 3: message types probe, ack, reject
 4:
 5: if i = root then                                   ▷ root initiates tree construction
 6:     send probe to Γ(i)
 7:     parent ← i
 8: end if
 9:
10: while (childs ∪ others) ≠ (Γ(i)\{parent}) do
11:     receive msg(j)
12:     case msg(j).type of
13:             probe:  if parent = ⊥ then            ▷ probe received first time
14:                         parent ← j
15:                         send ack to j
16:                         send probe to Γ(i)\{j}
17:                     else                          ▷ probe received before
18:                         send reject to j
19:             ack:    childs ← childs ∪ {j}         ▷ include j in children
20:             reject: others ← others ∪ {j}         ▷ include j in unrelated neighbors
21: end while
```

Flood_ST- Example



Message complexity= 2m+2=20
Time complexity=3

➔ Shortest path is not always followed due to communication link speeds.

Flood_ST: Complexity Analysis

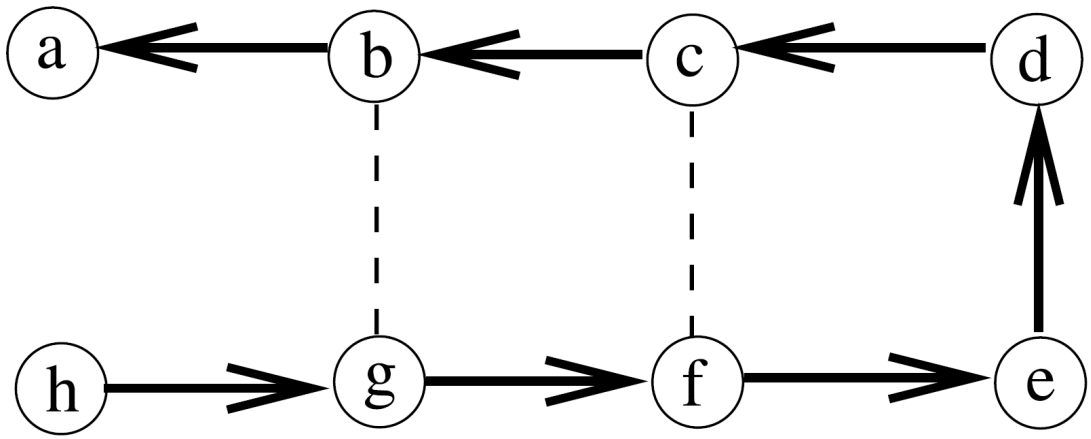
- **Message Complexity:**
  - Each edge will be traversed at least twice with probe and ack or probe and reject messages.
  - Each edge will be traversed at most 4 times in the case of two nodes attempting to send each other probe message concurrently.
  - In total 4m messages will be sent.

➔ Message complexity = ***O(4m)=O(m)***

Flood_ST: Complexity Analysis

- **Time Complexity:**
  - The message maybe transferred over the fast communication links of the longest path instead of the shortest paths with slow links.

  - ***➔ Time Complexity = O(n)***
    n: The number of vertices in the graph.

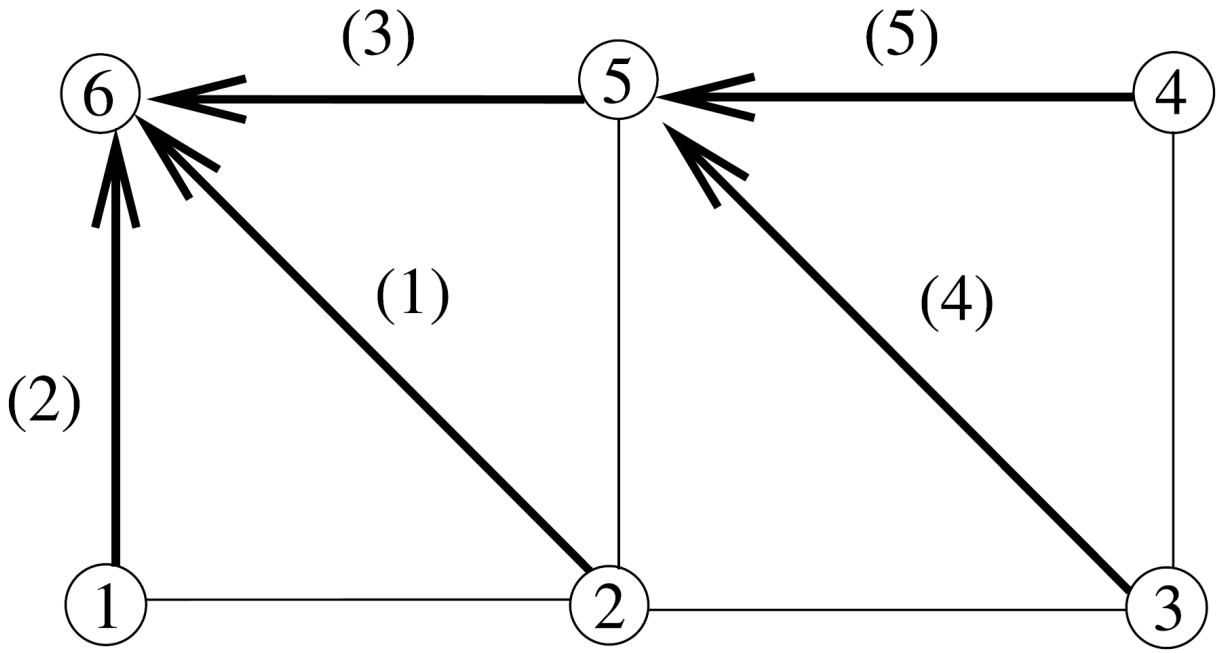


Breadth First Search (BFS)

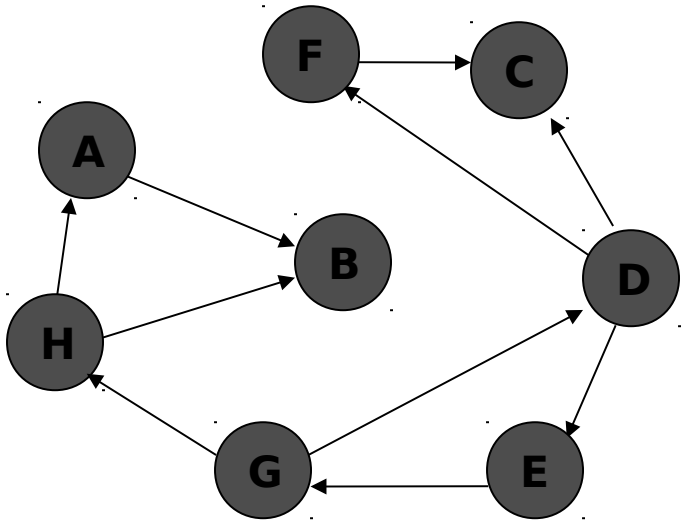
- **Graph Traversal**: Visiting all vertices of a graph in some predefined order.
- It maybe required to find the shortest distances of vertices from a source vertex in terms of the number of links (hops) to that source.
- ➔ Approach: Breadth-First-Search (BFS) tree.

Definition 5.1 (Breadth-First-Search Tree) A *breadth-first-search tree* T of a graph G is a spanning tree of G such that for every node of G, the tree path is a minimum-hop path to the root.

Breadth First Search (BFS)



Walk-Through



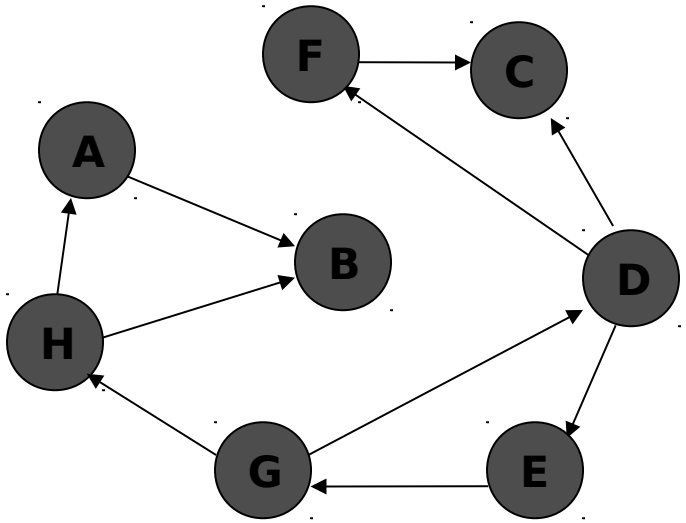
Enqueued Array

A	
B	
C	
D	
E	
F	
G	
H	

Q →

How is this accomplished? Simply replace the stack with a queue! Rules: (1) Maintain an *enqueued* array. (2) Visit node when *dequeued*.

Walk-Through



Enqueued Array

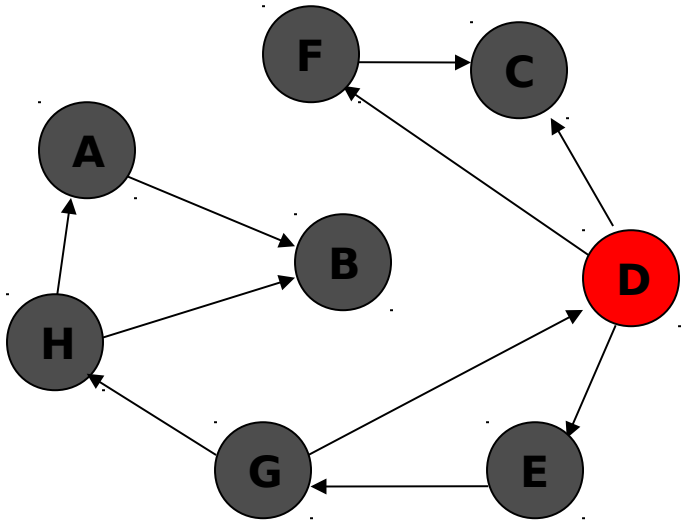
A	
B	
C	
D	√
E	
F	
G	
H	

Q → D

Nodes visited:

Enqueue D. Notice, D not yet visited.

Walk-Through



Enqueued Array

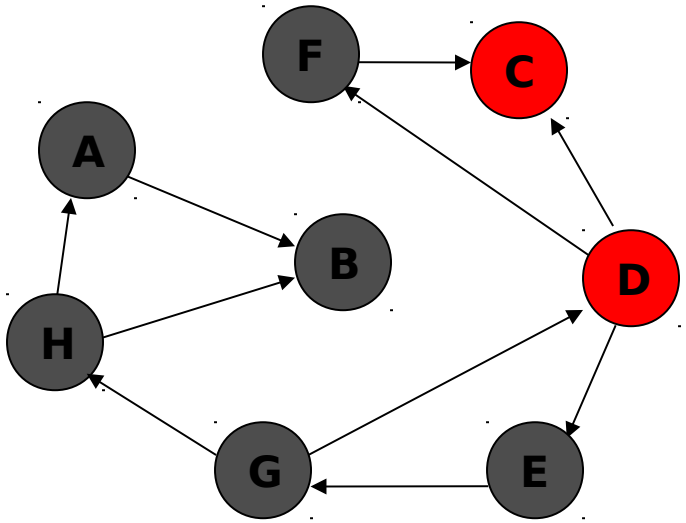
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → C → E → F

Nodes visited: D

Dequeue D. Visit D. Enqueue unenqueued nodes adjacent to D.

Walk-Through



Enqueued Array

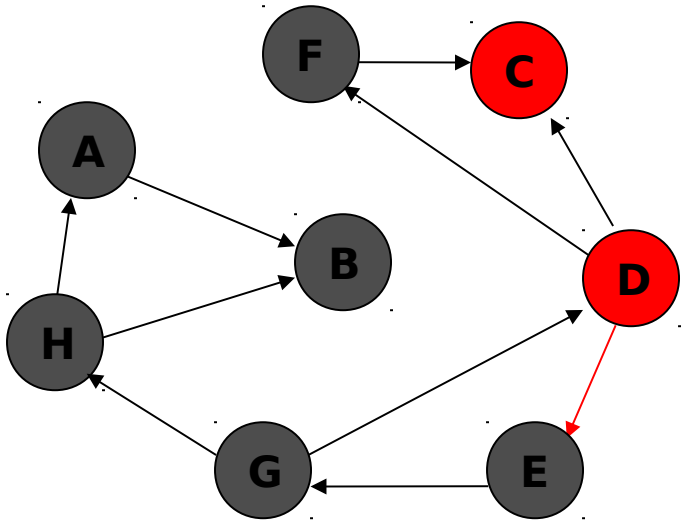
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → E → F

Nodes visited: D, C

Dequeue C. Visit C. Enqueue unenqueued nodes adjacent to C.

Walk-Through



Enqueued Array

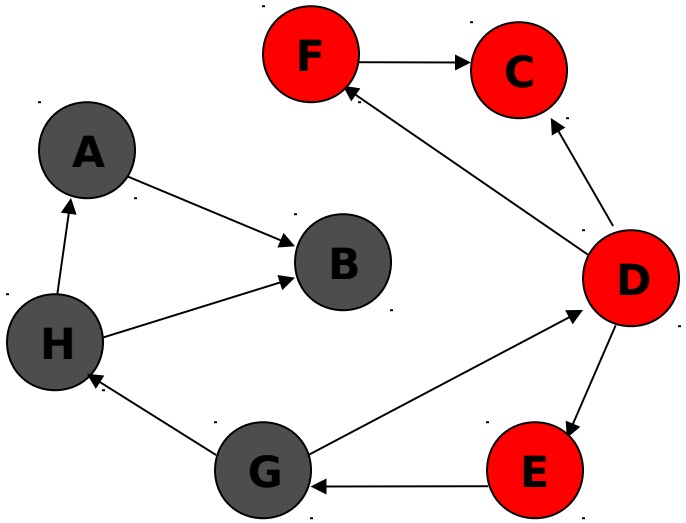
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → F → G

Nodes visited: D, C, E

Dequeue E. Visit E. Enqueue unenqueued nodes adjacent to E.

Walk-Through



Enqueued Array

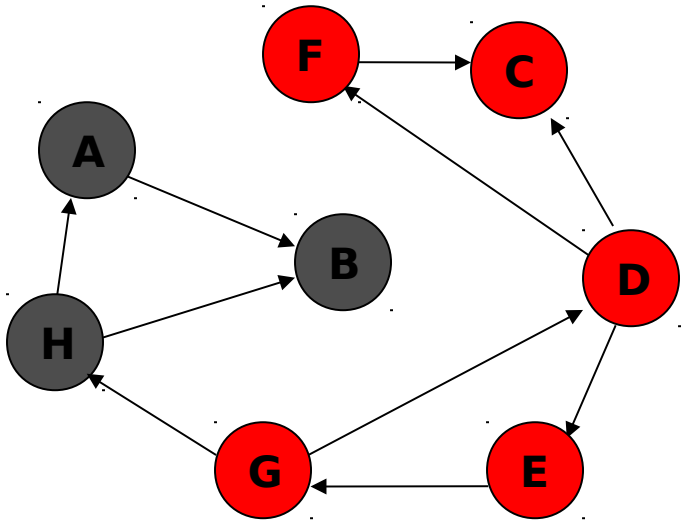
A	
B	
C	√
D	√
E	√
F	√
G	√
H	

Q → G

Nodes visited: D, C, E, F

Dequeue F. Visit F. Enqueue unenqueued nodes adjacent to F.

Walk-Through



Enqueued Array

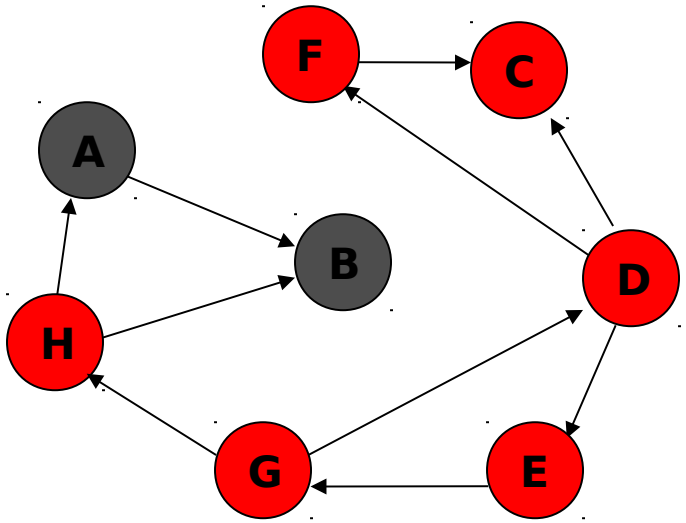
A	
B	
C	√
D	√
E	√
F	√
G	√
H	√

Q → H

Nodes visited: D, C, E, F, G

Dequeue G. Visit G. Enqueue unenqueued nodes adjacent to G.

Walk-Through



Enqueued Array

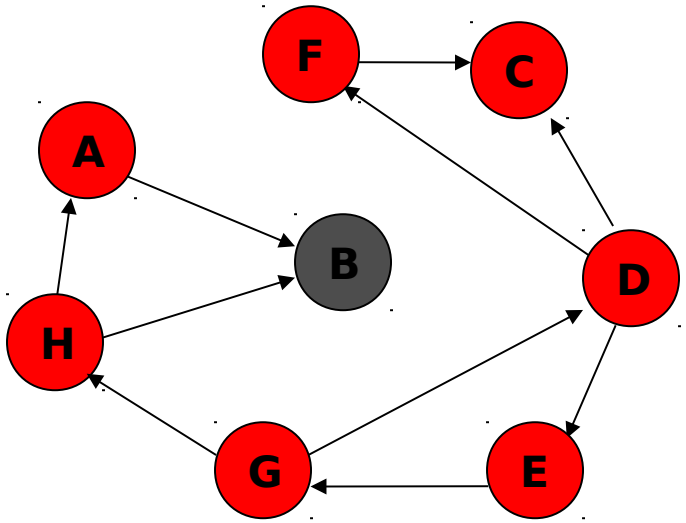
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q → A → B

Nodes visited: D, C, E, F, G, H

Dequeue H. Visit H. Enqueue unenqueued nodes adjacent to H.

Walk-Through



Enqueued Array

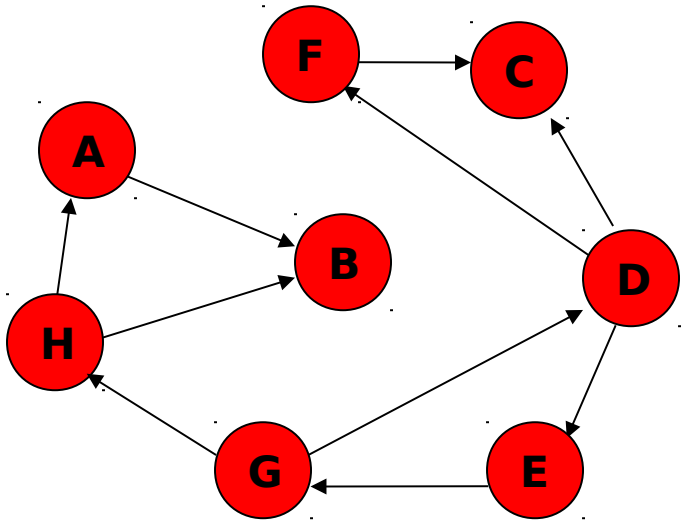
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q → B

Nodes visited: D, C, E, F, G, H, A

Dequeue A. Visit A. Enqueue unenqueued nodes adjacent to A.

Walk-Through



Enqueued Array

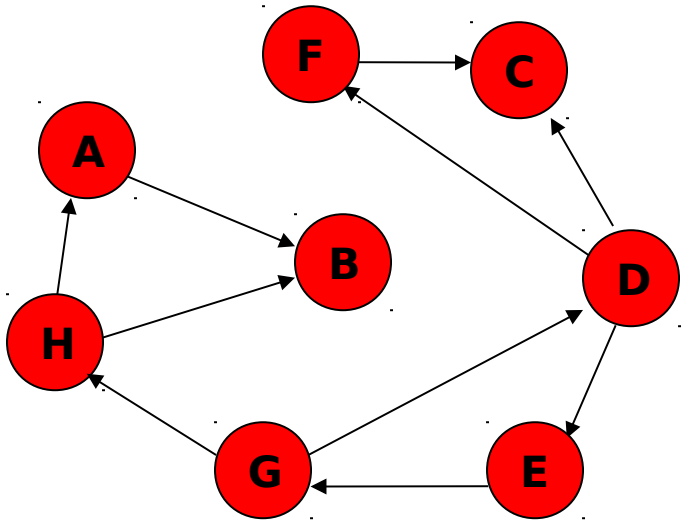
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q empty

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

Dequeue B. Visit B. Enqueue unenqueued nodes adjacent to B.

Walk-Through



Enqueued Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q empty

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

Q empty. Algorithm done.

Depth First Search (DFS)

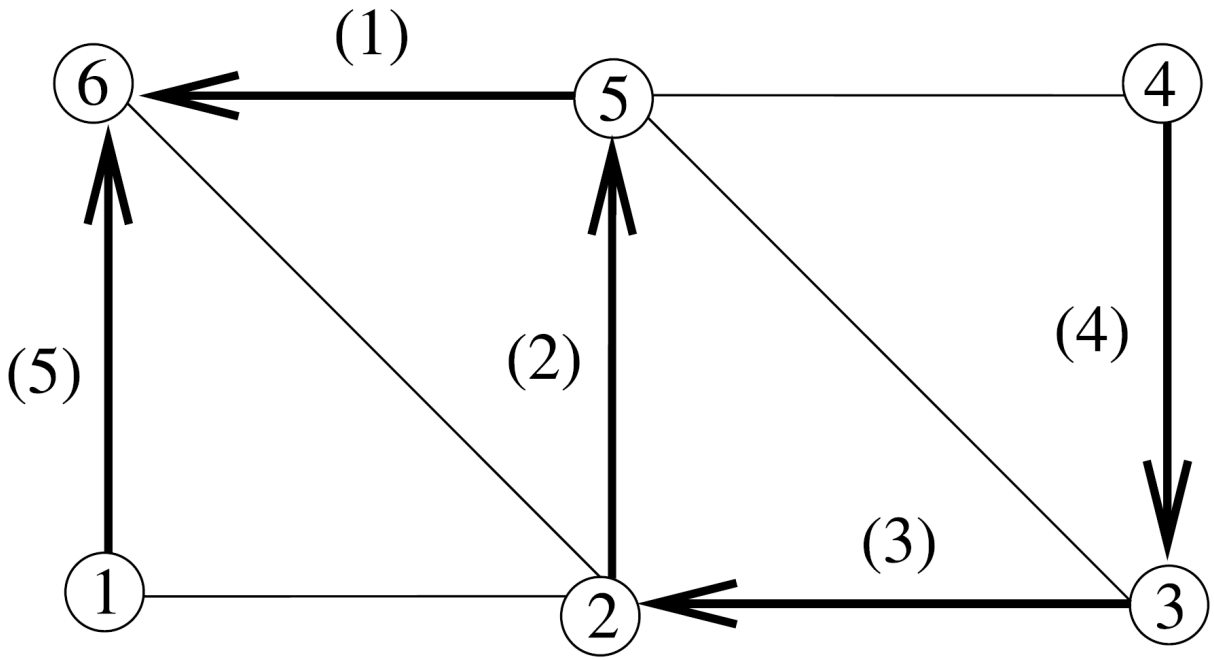
- **Objective**: Find all of the vertices reachable from a source vertex in the graph.

- It visits all possible vertices as far as it can reach and when all vertices are visited, it returns to the parent node.

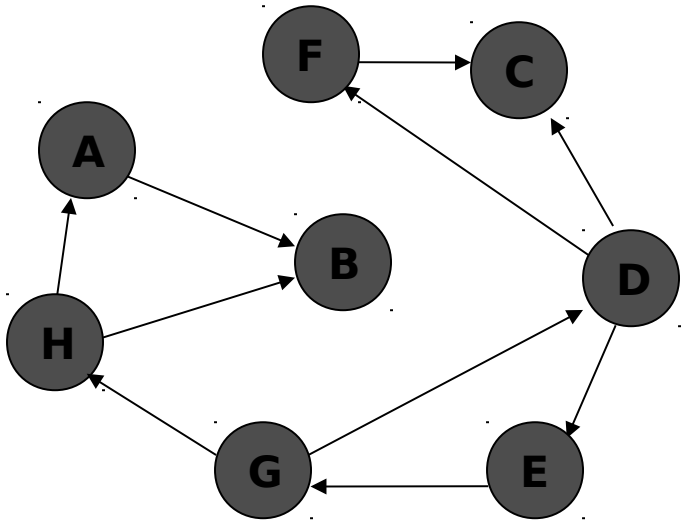
Definition 5.2 (Depth-First-Search Tree) A *frond* edge is an edge that does not belong to a spanning tree. For a rooted spanning tree T of a graph G, let us denote by $S(u)$ all the nodes in the subtree of u, and $P(u)$ denote all the vertices that exist between u and the root. A *depth-first-search tree* of a graph G is a spanning tree T of G such that for every frond edge $\{u, v\}$, $v \in S(u) \vee v \in P(u)$ [5].

- Applications:
  - Finding strongly connected components of a directed graph.
  - Cycles detection !!
  - ....

Depth First Search (DFS)

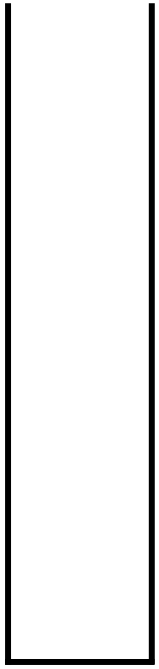


Walk-Through



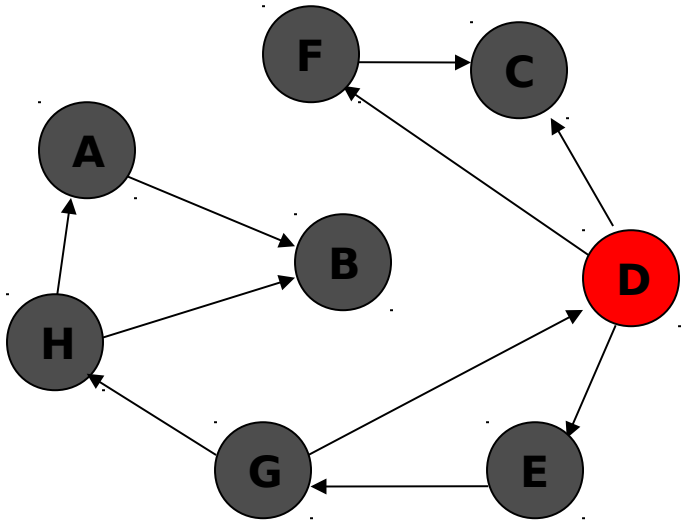
Visited Array

A	
B	
C	
D	
E	
F	
G	
H	



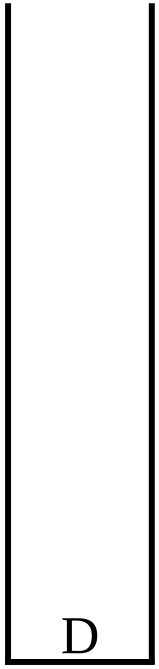
Task: Conduct a depth-first search of the graph starting with node D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

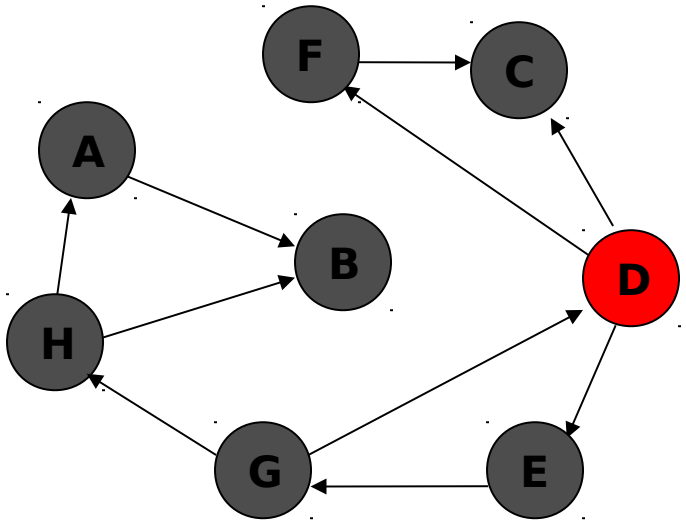


Visit D

The order nodes are visited:

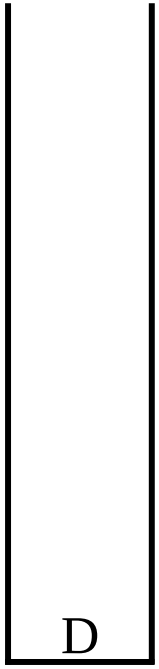
D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

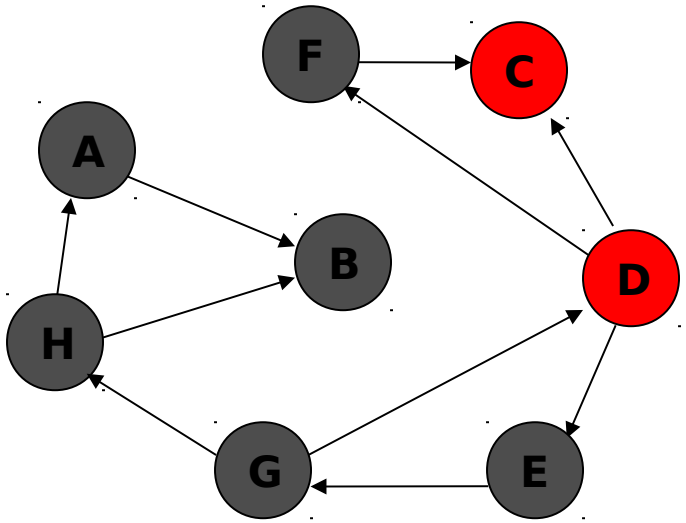


The order nodes are visited:

D

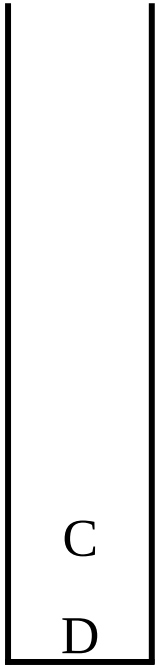
Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

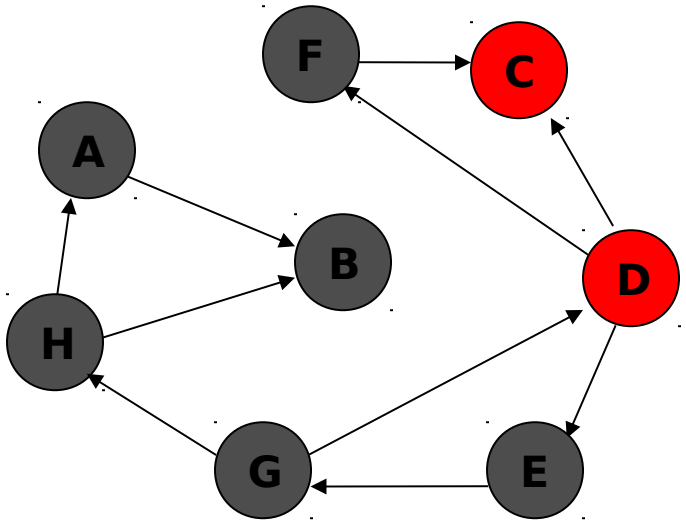


Visit C

The order nodes are visited:

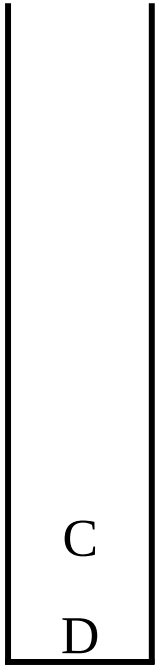
D, C

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

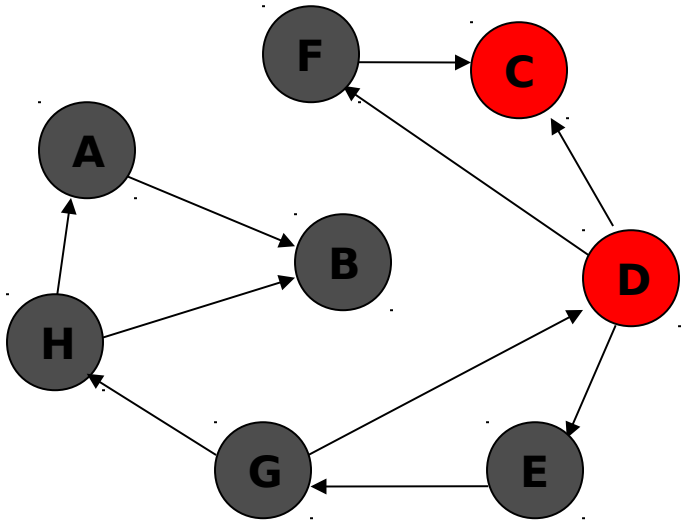


The order nodes are visited:

D, C

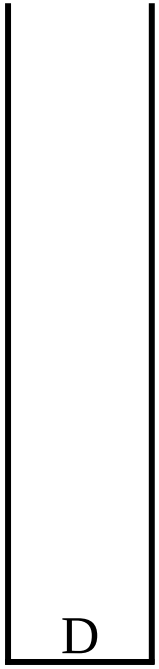
No nodes adjacent to C; cannot continue ➔ *backtrack*, i.e., pop stack and restore previous state

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

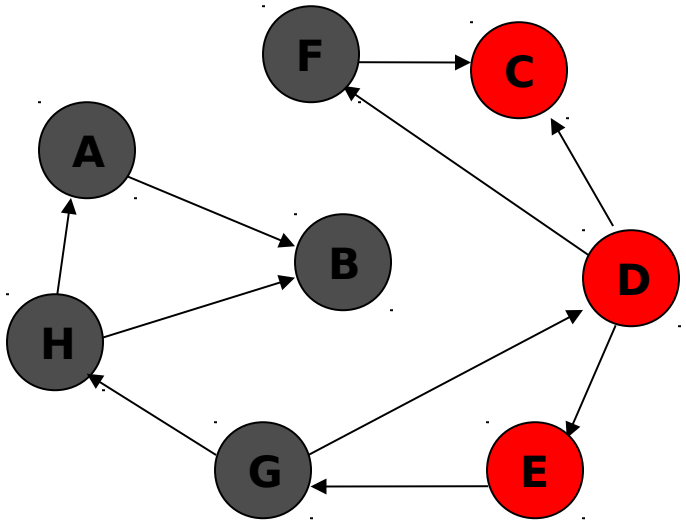


The order nodes are visited:

D, C

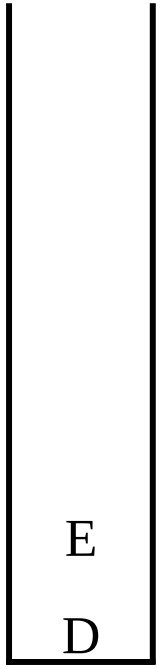
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

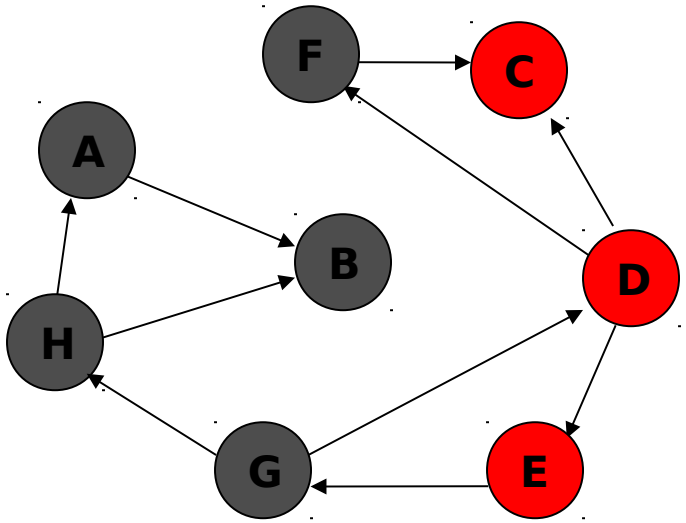


The order nodes are visited:

D, C, E

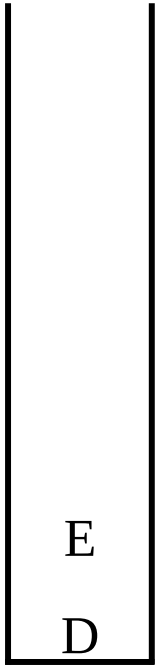
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

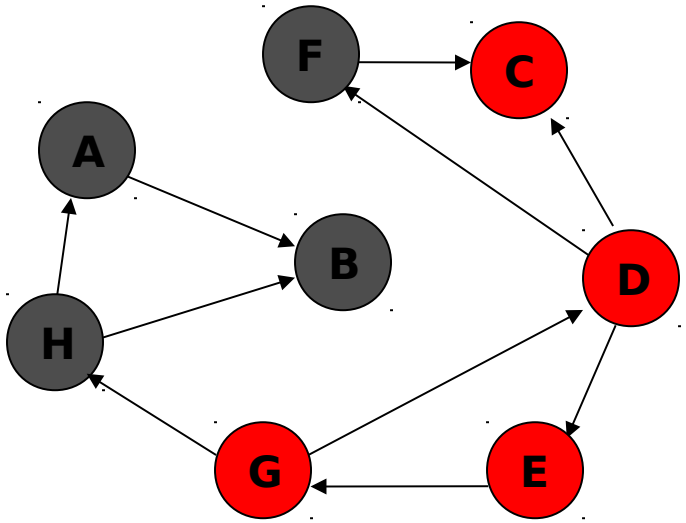


The order nodes are visited:

D, C, E

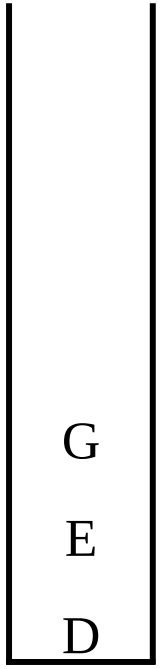
Only G is adjacent to E

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

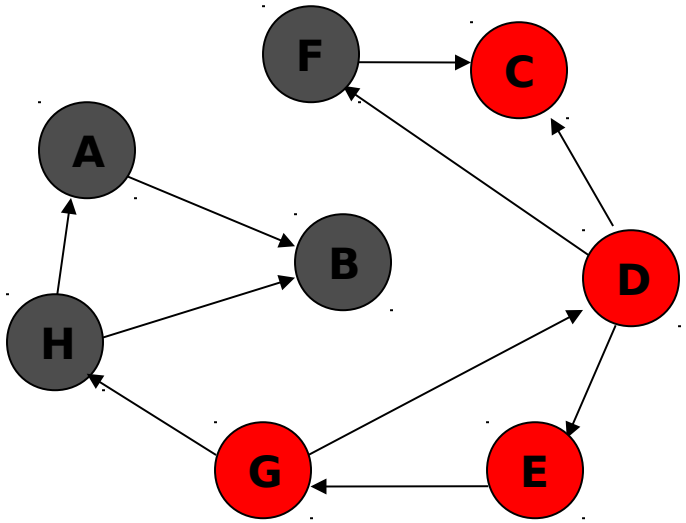


Visit G

The order nodes are visited:

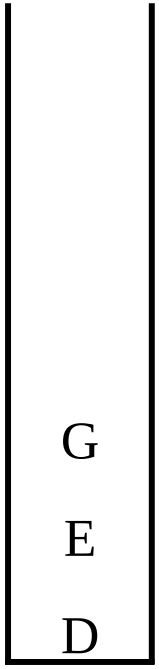
D, C, E, G

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

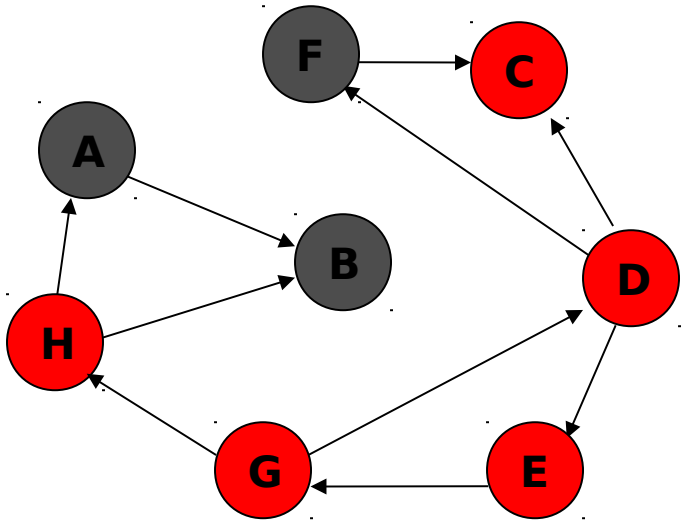


The order nodes are visited:

D, C, E, G

Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

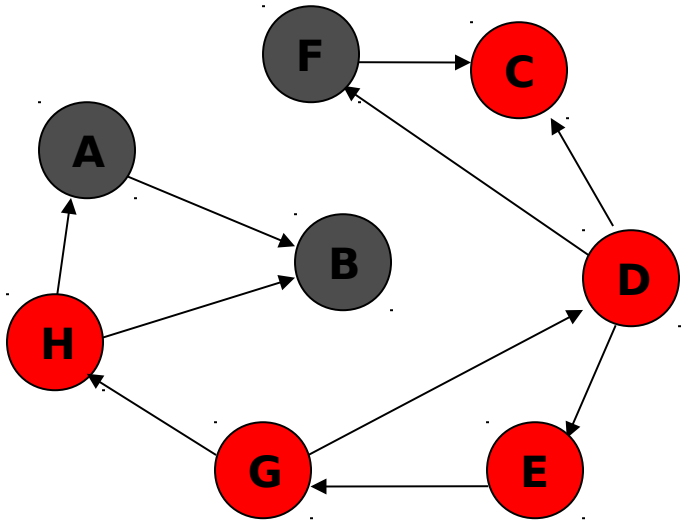
H
G
E
D

Visit H

The order nodes are visited:

D, C, E, G, H

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

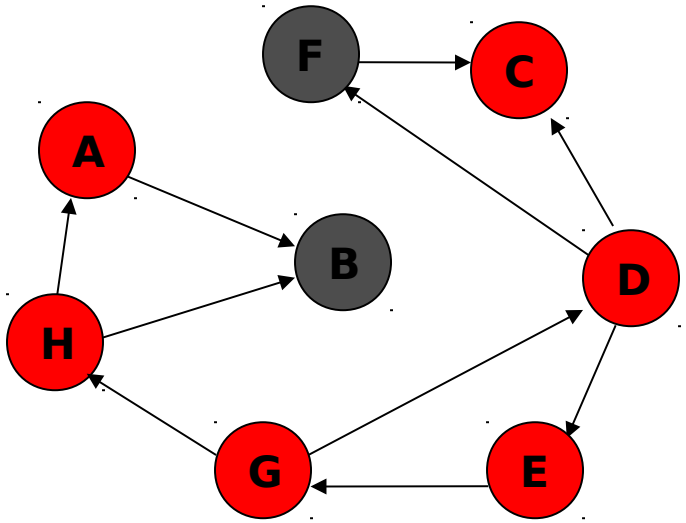
H
G
E
D

The order nodes are visited:

D, C, E, G, H

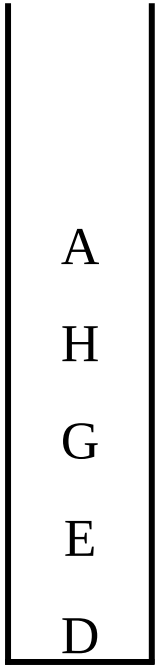
Nodes A and B are adjacent to F. Decide to visit A next.

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

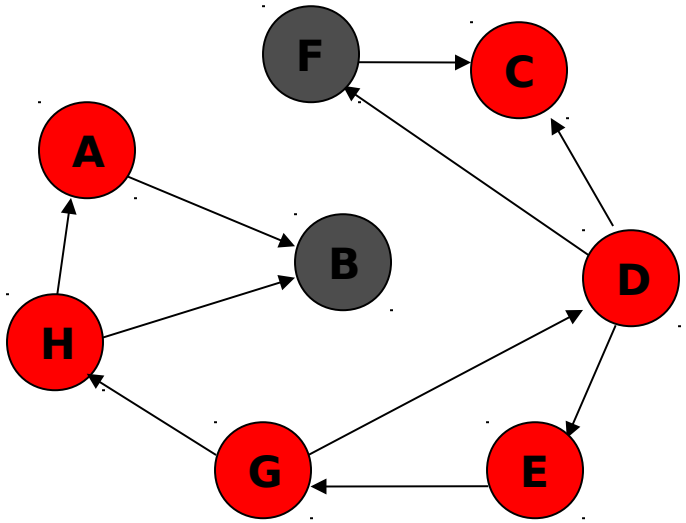


Visit A

The order nodes are visited:

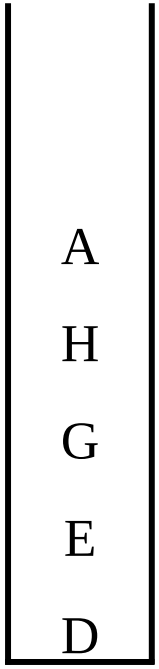
D, C, E, G, H, A

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

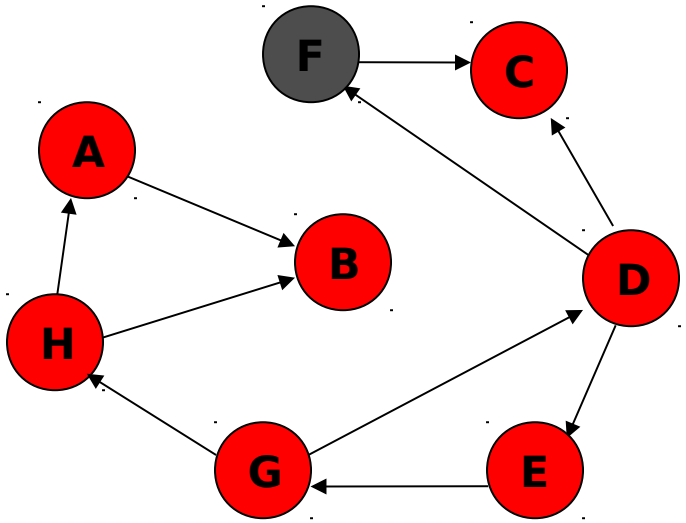


The order nodes are visited:

D, C, E, G, H, A

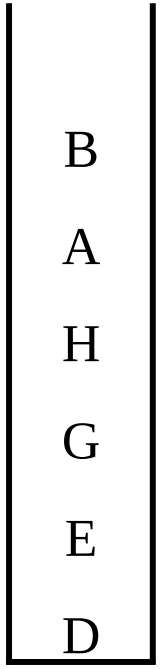
Only Node B is adjacent to A.
Decide to visit B next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

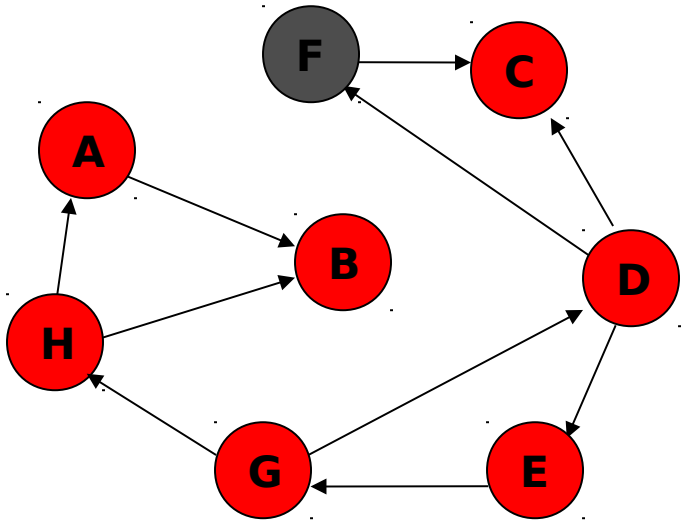


Visit B

The order nodes are visited:

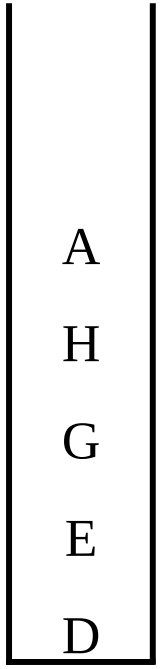
D, C, E, G, H, A, B

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

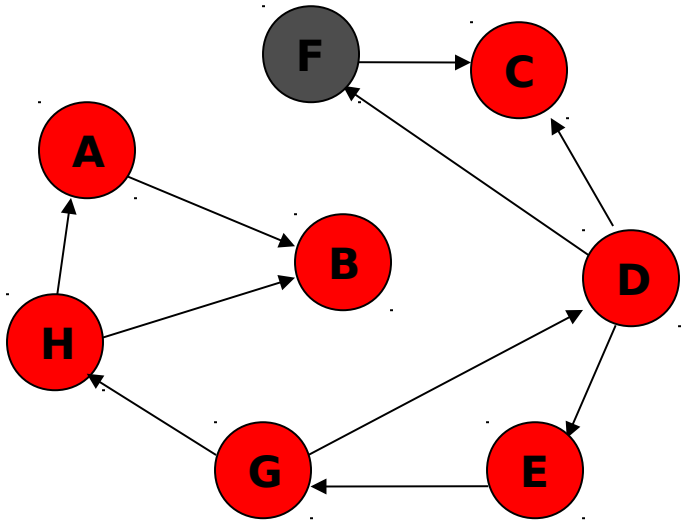


The order nodes are visited:

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

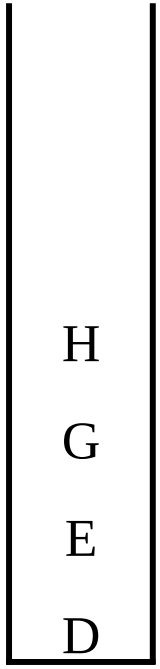
No unvisited nodes adjacent to B. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

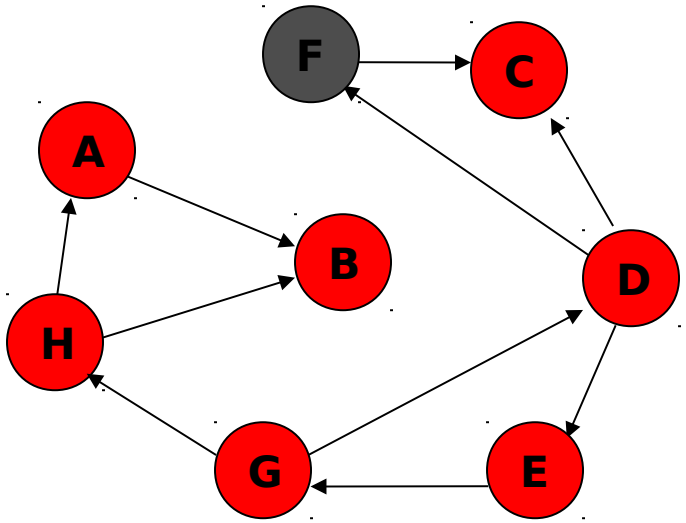


The order nodes are visited:

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

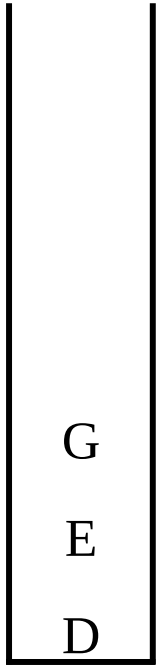
No unvisited nodes adjacent to A. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

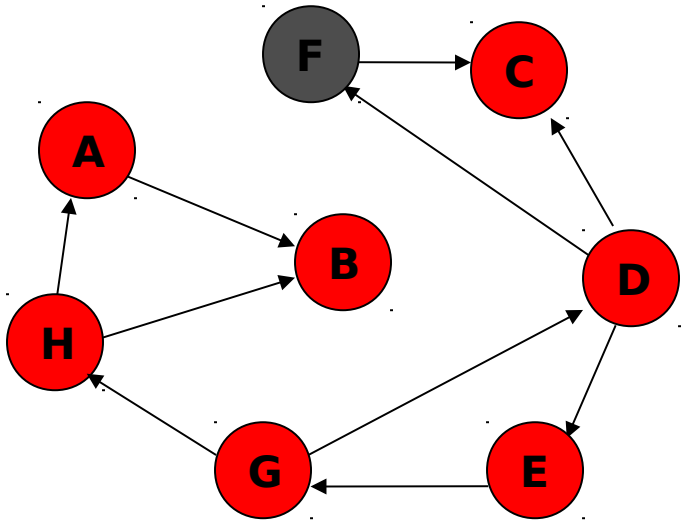


The order nodes are visited:

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

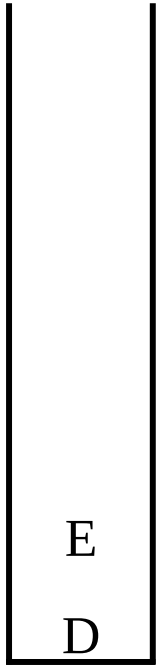
No unvisited nodes adjacent to H. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

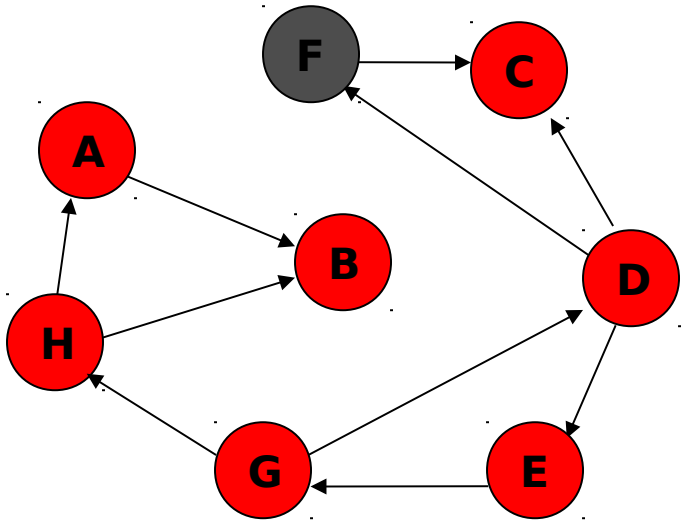


The order nodes are visited:

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

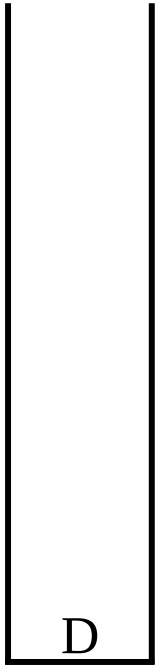
No unvisited nodes adjacent to G. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

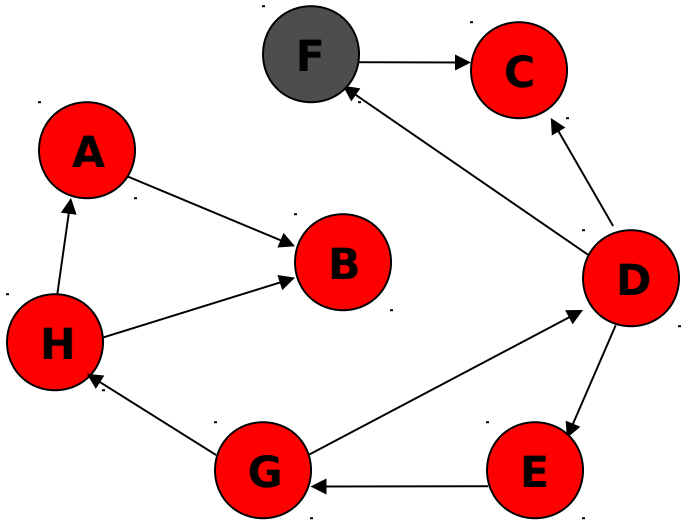


The order nodes are visited:

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

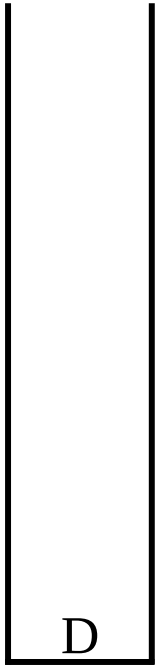
No unvisited nodes adjacent to E. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

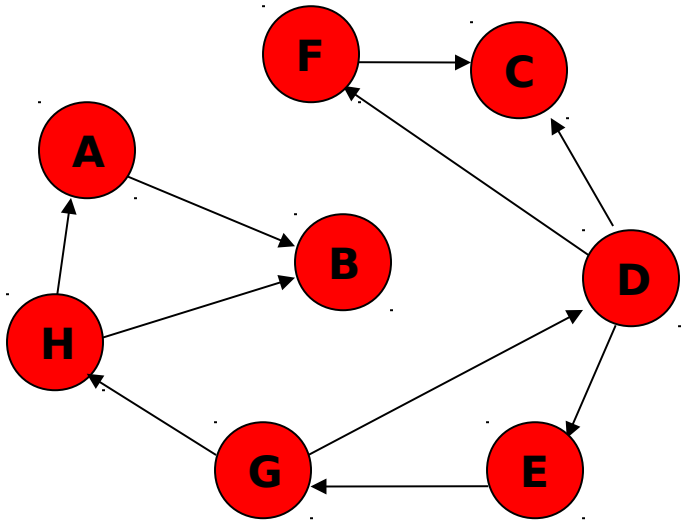


The order nodes are visited:

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

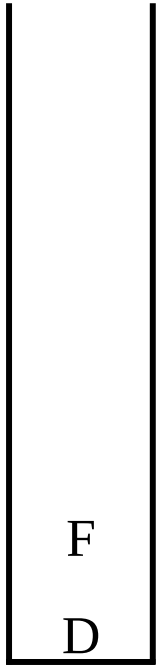
F is unvisited and is adjacent to D. Decide to visit F next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

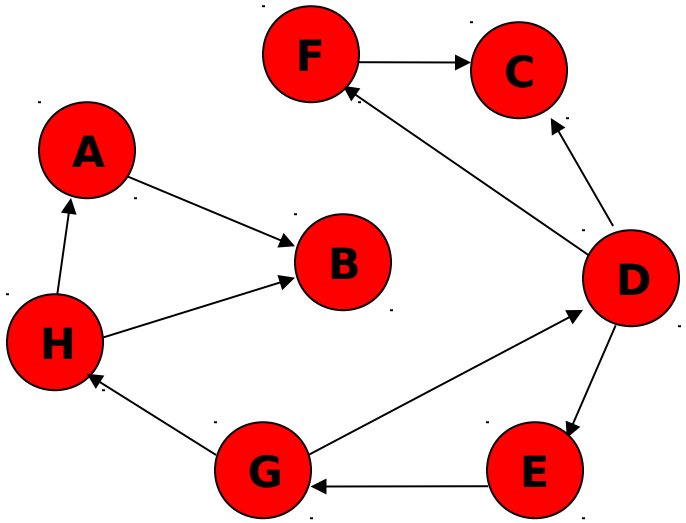


Visit F

The order nodes are visited:

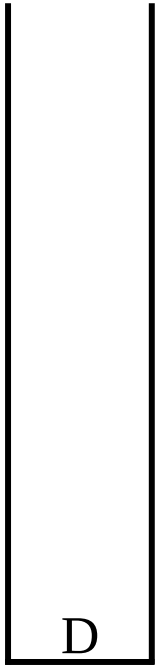
D, C, E, G, H, A, B, F

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

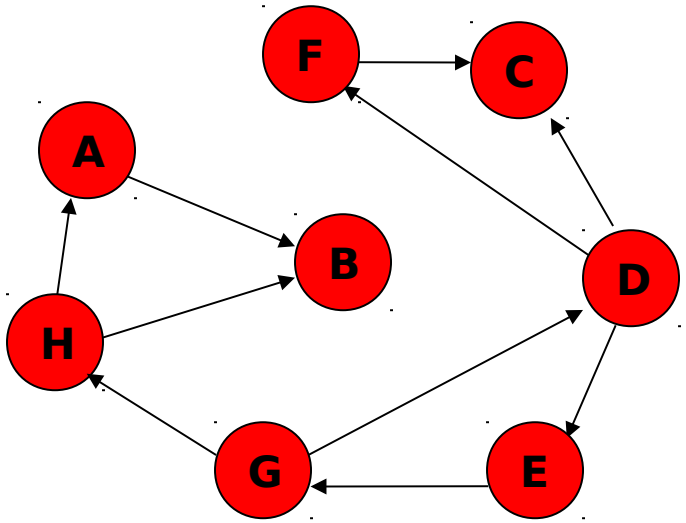


The order nodes are visited:

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

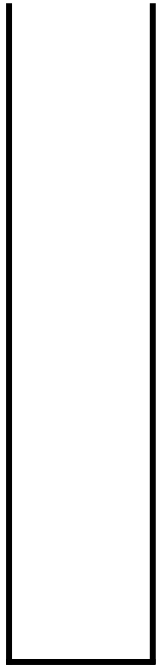
No unvisited nodes adjacent to F. Backtrack.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

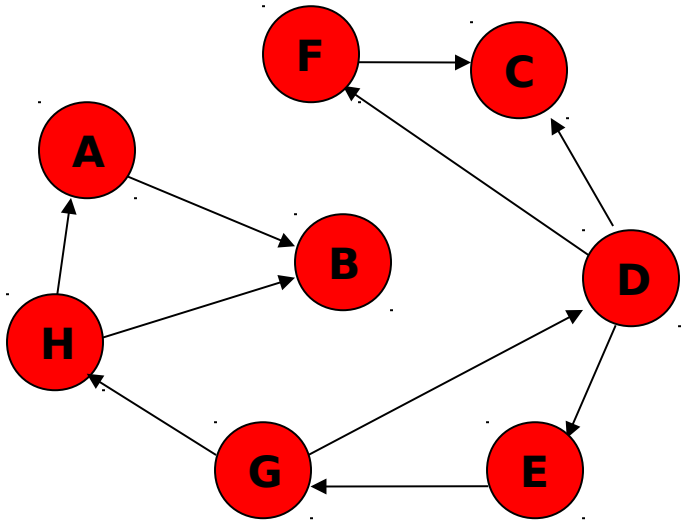


The order nodes are visited:

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

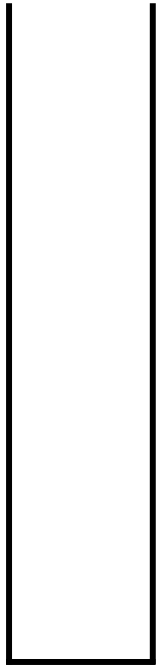
No unvisited nodes adjacent to D. Backtrack.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√



The order nodes are visited:

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

Stack is empty. Depth-first traversal is done.

Distributed Breadth-First-Search Algorithms

- **Synchronous BFS**: Synchronous algorithm working in rounds to construct the BFS tree.
- Asynchronous BFS Construction: Asynchronous algorithm to construct the BFS tree.

Synchronous BFS Construction Algorithm (Synch_BFS)

- The synchronous distributed version of Dijkstra's algorithm for single-source shortest path problem.
- A single initiator algorithm.
- In each synchronous round, a partial BFS tree is formed.
- The already formed branches of the tree are used to carry synchronization messages and the leaves search for new nodes to be added to the tree.
- The depth of the tree is incremented in each round until all the nodes are covered.

Synch_BFS Algorithm Features

- Full termination detection is provided so that all other nodes know when the BFS is constructed.
- Synchronization is performed using **special control messages**

➔ Eliminates the need of other synchronization techniques such as specific synchronizer node.

Synch_BFS Algorithm Message Types

- **Round**: Sent by the root at the beginning of each round and transferred by all the nodes of the partial BFS tree to their children.
- **Probe:** Sent by the leaves of the partial BFS tree to the unsearched neighbors (that have not yet been members of the tree).
- **Ack/Reject:** Sent by the searched node to accept/reject being a child of the sending leaf node.
- **Upcast:** Sent first by the leaf nodes of the partial BFS tree and then by the intermediate nodes to their parents to signal the end of neighbor search and the end of the round.
- **Finish:** Sent by the leaf nodes to their parents to signal that their part is done as they either have no neighbors other than the parent or they do not have any children.
- **Terminate:** Broadcasted by the root to all nodes to signal that the construction of the BFS tree is completed.

Synch_BFS Algorithm Flow

➔The root starts the algorithm by sending the first **round** message to its neighbors.

➔The neighbors respond with **ack** message.

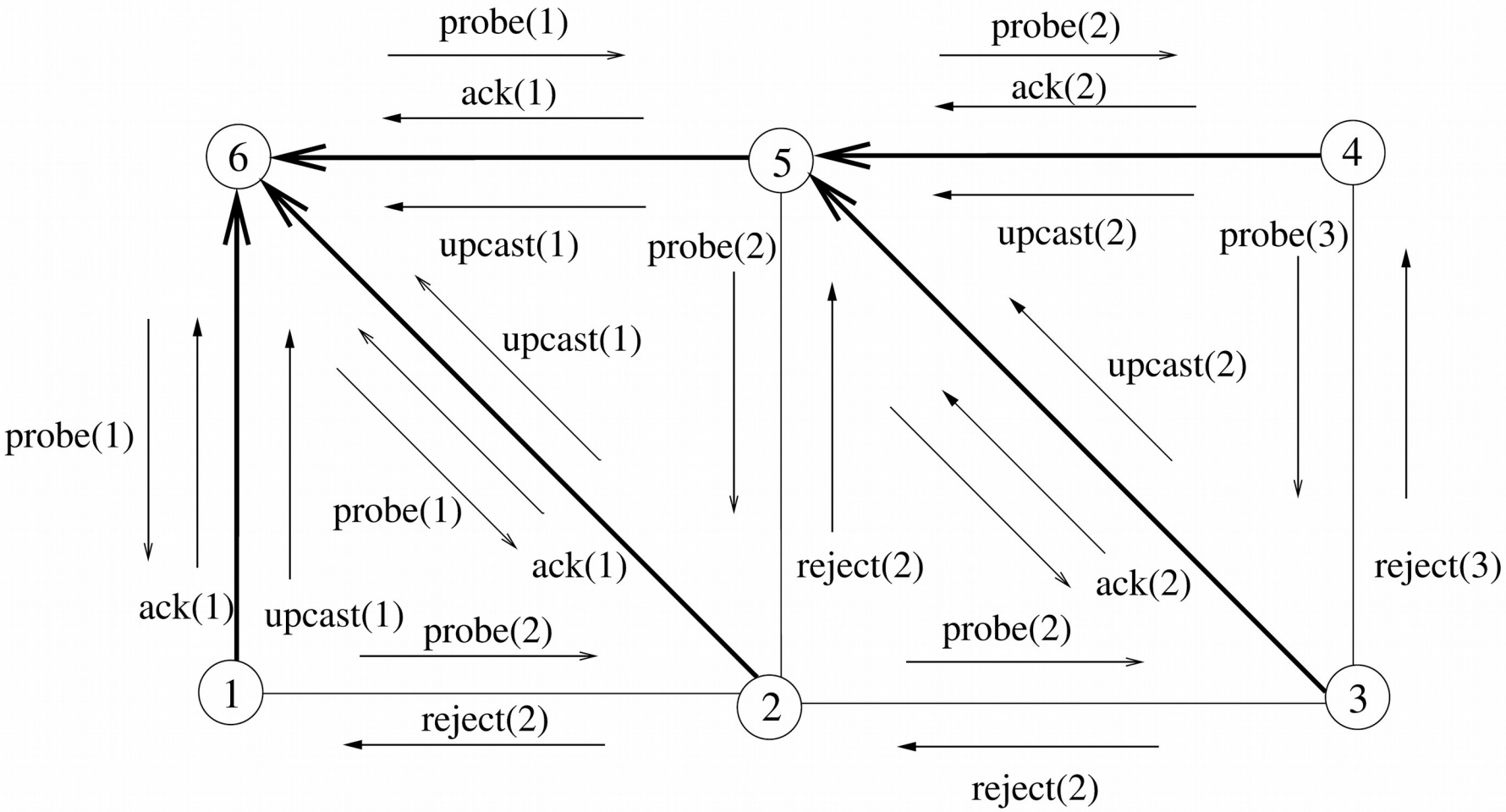
➔ The **round** message is transferred over the partial BFS tree to the leaf nodes.

➔ The leaf nodes search nodes for the next level the BFS tree by sending **probe** messages.

➔ Each leaf node that has received ack or reject message from all of its neighbors except the parent returns **upcast** message to its parent, which convergecasts the upcast message to its parent.

➔ When all upcast messages from the neighbors of the root are received, root starts the next round by issuing the next **round** message.

Synch_BFS Algorithm Flow- Example



Synch_BFS Algorithm Termination

- The root does not know beforehand the diameter of the graph.
- When all the neighbors of a node i except its parent have responded by a reject message or when a leaf node (in the constructed BFS tree) does not have any neighbors other than its parent, the part of node i is over.
- When a node reaches this situation, it sends a **finish** message to its parent which converecasts it to the root.
- When the root receives finish message from all of its children, it broadcasts a **terminate** message over the network to inform every node that the process is over.

Algorithm 5.1 *Synch_BFS*

```
 1: int parent ← ⊥, k ← 1, i, j
 2: set of int childs, others, phased, finished ← ∅
 3: message types round, probe, ack, reject, upcast
 4: boolean leaf_flag ← false
 5:
 6: if i = root then
 7:     send probe(k) to Γ(i)
 8:     while true do
 9:         receive msg(j)
10:         case msg.type of
11:             ack(k) ∨ upcast(k):     phased ← phased ∪ {j}
12:             finish:                 finished ← finished ∪ {j}
13:                                     if finished = childs then
14:                                         send terminate to childs
15:                                         exit                    ▷ root terminates
16:             if phased = childs then                             ▷ start next round
17:                 k ← k + 1
18:                 send round(k) to childs, phased ← 0
19:     end while
```

```
20: else
21:     while true do
22:         round_over ← false
23:         while ¬round_over do
24:             receive msg(j)
25:             case msg.type of
26:                 round(k):     if state = interm then                    ▷ intermediate node
27:                                   send round(k) to childs
28:                               else                                      ▷ leaf node
29:                                   send probe(k) to Γ(i) \ {parent}
30:                               round_recvd ← true
31:                 probe(k):     if parent = ⊥ then                        ▷ non-member node
32:                                   parent ← j; state ← new_leaf
33:                                   send ack(k) to j
34:                               else                                      ▷ a member node
35:                                   send reject(k) to j
36:                 ack(k):       childs ← childs ∪ {j}
37:                               if state ≠ interm then interm_flag ← true
38:                 reject(k):    others ← others ∪ {j}
39:                               if others = Γ(i) \ {parent} then          ▷ a leaf node finishes
40:                                   send finish to parent
41:                 upcast(k):    phased ← phased ∪ {j}
42:                 finish:       finished ← finished ∪ {j}
43:                               if (finished = childs) ∧ (state = interm) then
44:                                   send finish to parent
45:                 terminate:    if childs ≠ 0 then
46:                                   send terminate to childs
47:                               exit                                      ▷ all nodes terminate
```

```
48:             if round_recvd then
49:                 if ((state = leaf) ∧ ((childs ∪ others) = (Γ(i) \ {parent})) ∨ ((state =
    interm) ∧ (childs = phased ∪ finished)) then                    ▷ check end of round
50:                     send upcast(p) to parent
51:                     if state = new_leaf then                    ▷ update states
52:                         state ← leaf
53:                     else if interm_flag then
54:                         state ← interm
55:                     end if
56:                     phased, others ← ∅; round_recvd ← false; round_over ← true
57:                 end if
58:             end if
59:         end while
60:     end while
61: end if
```

Synch_BFS Algorithm Complexity Analysis

- **Lemma**: Synch_BFS algorithm correctly constructs a BFS tree.

Proof: By **mathematical induction**, let k denote the step at which the BFS tree is being constructed.

(1) At step k=0, no tree is constructed.

(2) Assume that at step k, T_k tree rooted at r is constructed.

(3) at step k+1, only the leaves of the tree will be active in sending the probe messages to their neighbors that are one hop away.

➔ Any added nodes will be k+1 hops away from r, forming $T`_k$

Synch_BFS Algorithm Complexity Analysis

- **Time Complexity:**
  - Broadcasting the round message to current T_k BFS tree (whose depth is k) is performed in k units.
  - Convergecast of the upcast messages is performed in k units as well.
  - Additional time is required for probe and ack/reject messages.
  - ➔ For each step, 2k+2 time is required.
  - The depth of the tree will be d in total (d represents the diameter of the graph).
  - Time Complexity= $= 2\sum_{k=1}^{d}(k+1) = (d+1)(d+2)/2 = O(d^2).$

➔$O(d^2)$

d: The diameter of the graph.

Synch_BFS Algorithm Complexity Analysis

- **Message Complexity:**
  - In round p, if the tree formed has k nodes, there will be k-1 round messages and k-1 upcast messages.
  - ➔ Total number of synchronization messages for a round is **O(n), n: the number of vertices in the graph.**
  - ➔ The number of synchronization messages in all rounds will be **O(nd)**

  - In each round, new edges of the BFS tree will be determined by probe and ack/reject messages.
  - ➔ The total number of discovery messages=2m

m: The number of edge

$$O(nd) + O(m) = O(nd + m)$$

➔ Message Complexity=