CSCI 2270 – CS 2: Data Structures

University of Colorado
Boulder

Reminders

Topics

- Graph Traversal Algorithms
  - Breadth-First Search
  - Depth-First Search

Graph Traversals

- Traversing a graph is similar to binary tree traversal
- Graph traversal can have cycles, whereas binary trees do not
  - May not be able to traverse the entire graph from a single vertex
  - Therefore, we must keep track of the vertices that have been visited
  - We must traverse the graph from each vertex (that hasn’t been visited) of the graph. This ensures that the entire graph is traversed.
- Two common graph traversal algorithms:
  - Breadth-first traversal
  - Depth-first traversal

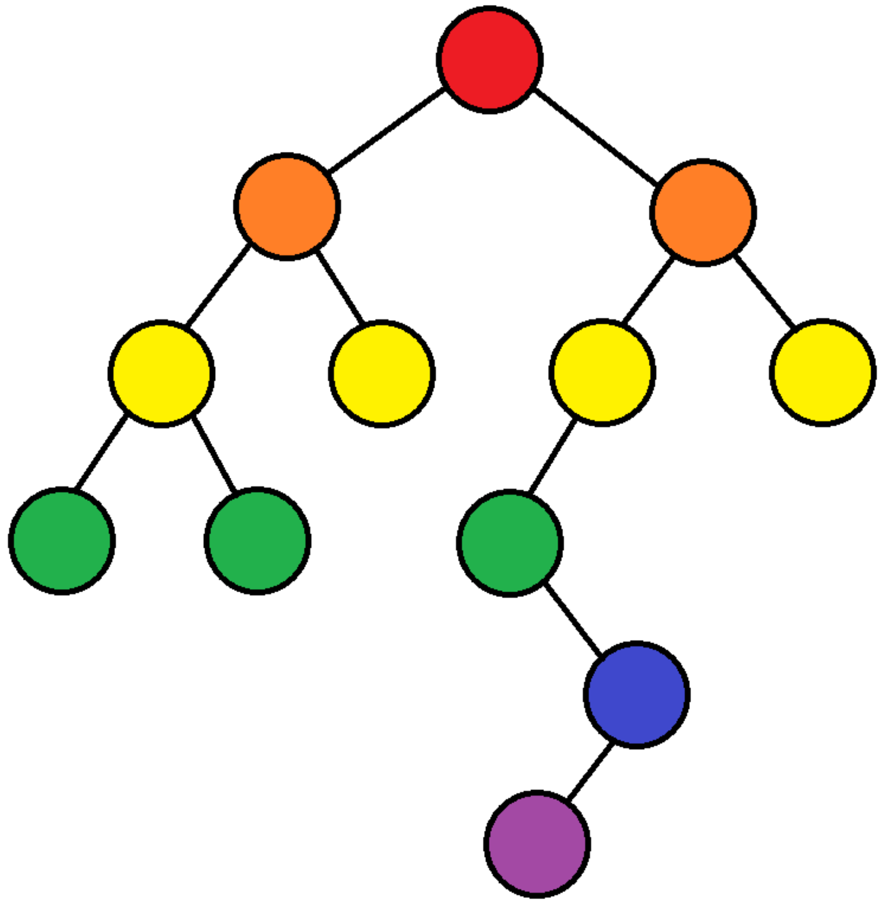
Graph Traversals

- For simplicity, we assume that when a vertex is visited, its index is output.
- Moreover, each vertex is visited only once.
- We can use a bool array to keep track of the visited vertices.
- The difference between BFS and DFS is the order in which we traverse the graph.
- Graph traversal can start with any node! There is no “root” node as we saw with trees.

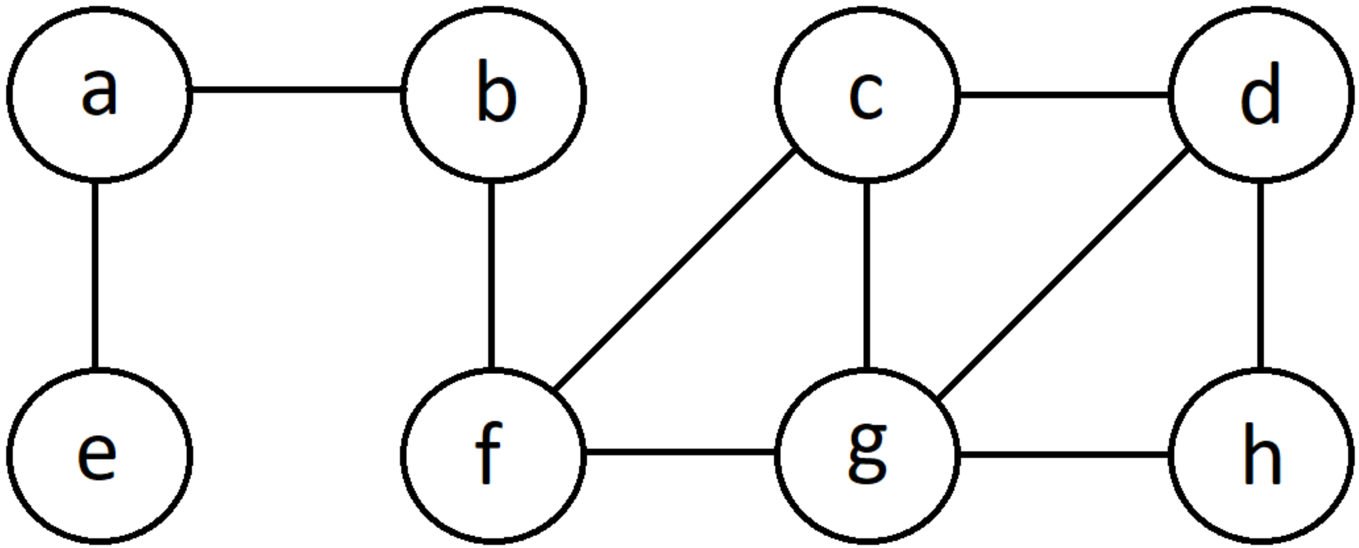
Breadth-First Traversal

- Similar to traversing a binary tree level-by-level (the nodes at each level are visited from left to right).
- All the nodes at any level, i, are visited before visiting the nodes at level $i + 1$.
- Use queues
- Note: we're not using BFS to search for the shortest path!

Breadth-First Traversal

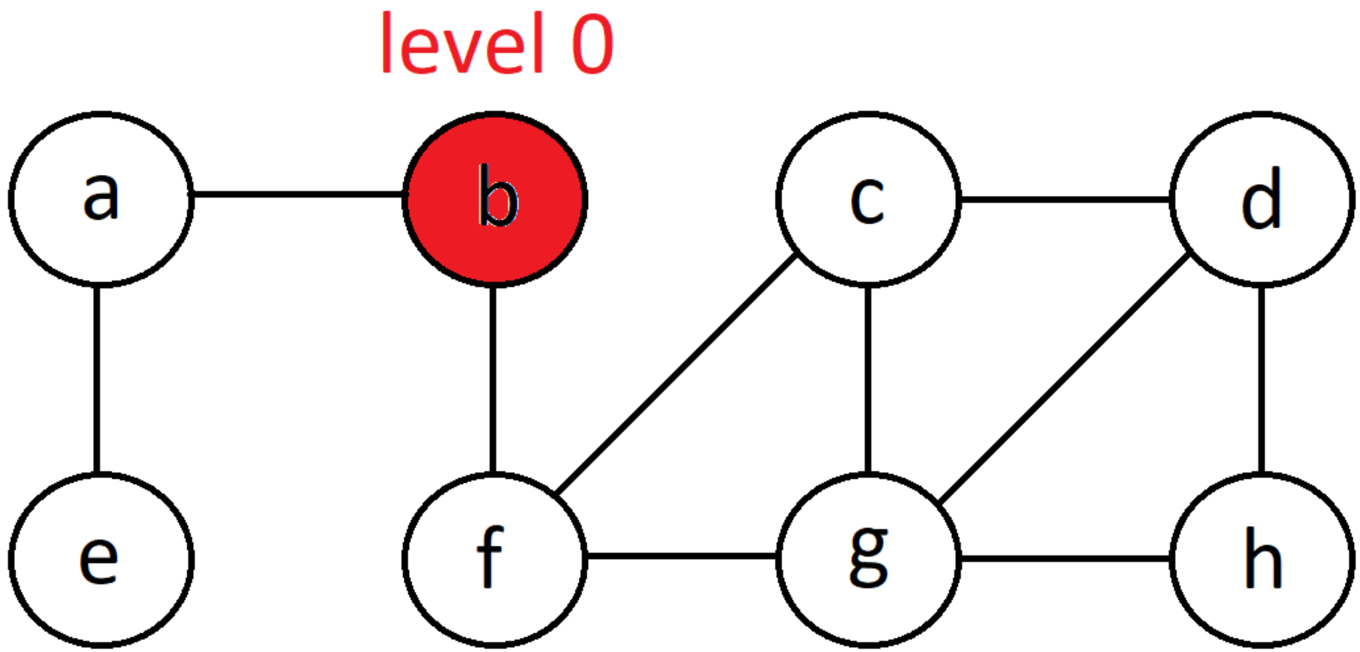


BFS Traversal - Undirected



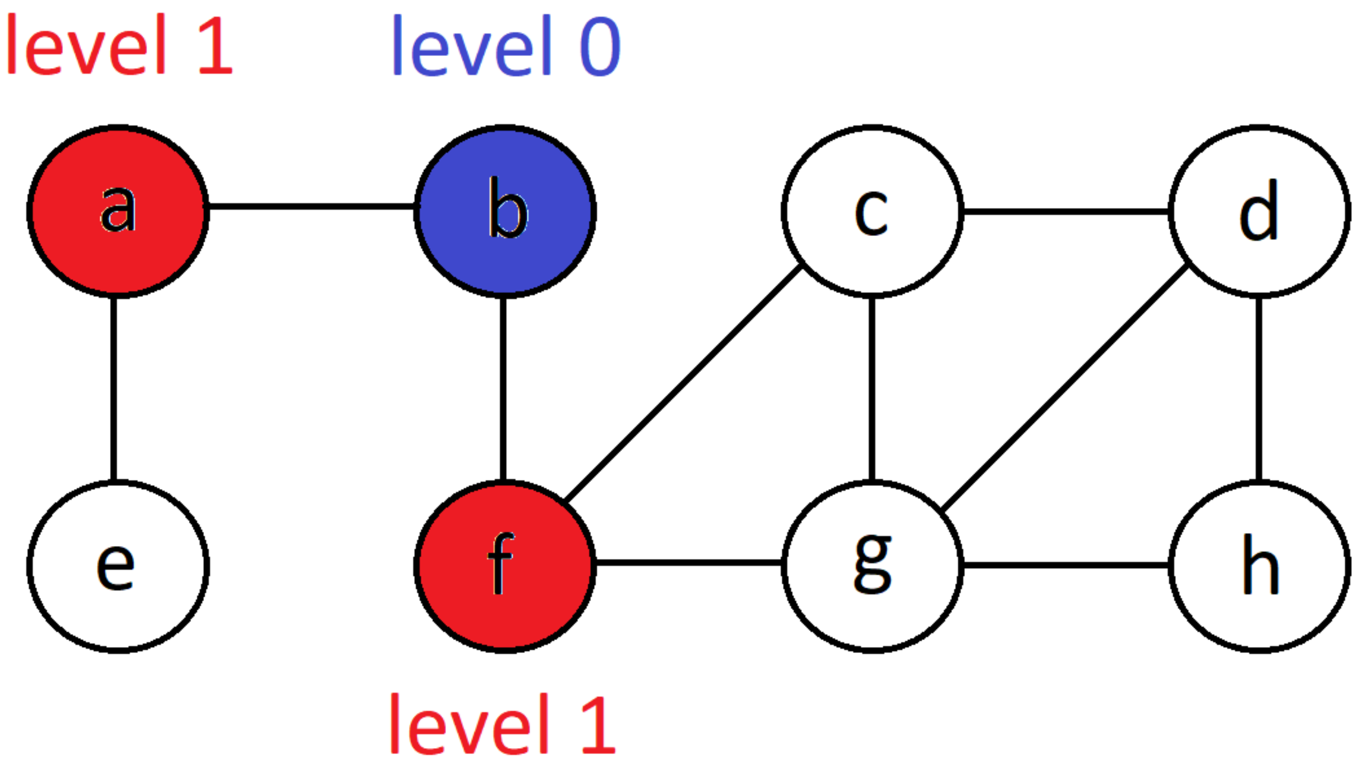
queue: -

BFS Traversal - Undirected



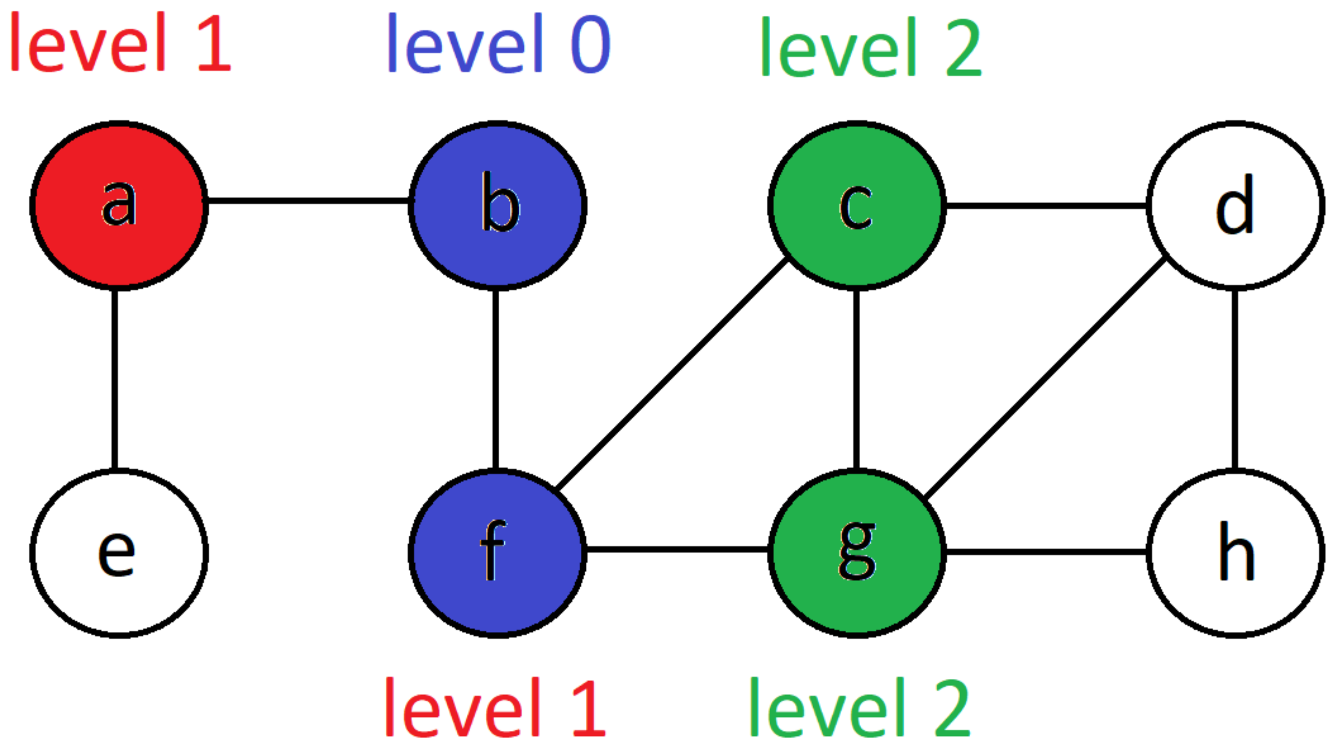
queue: **b**

BFS Traversal - Undirected



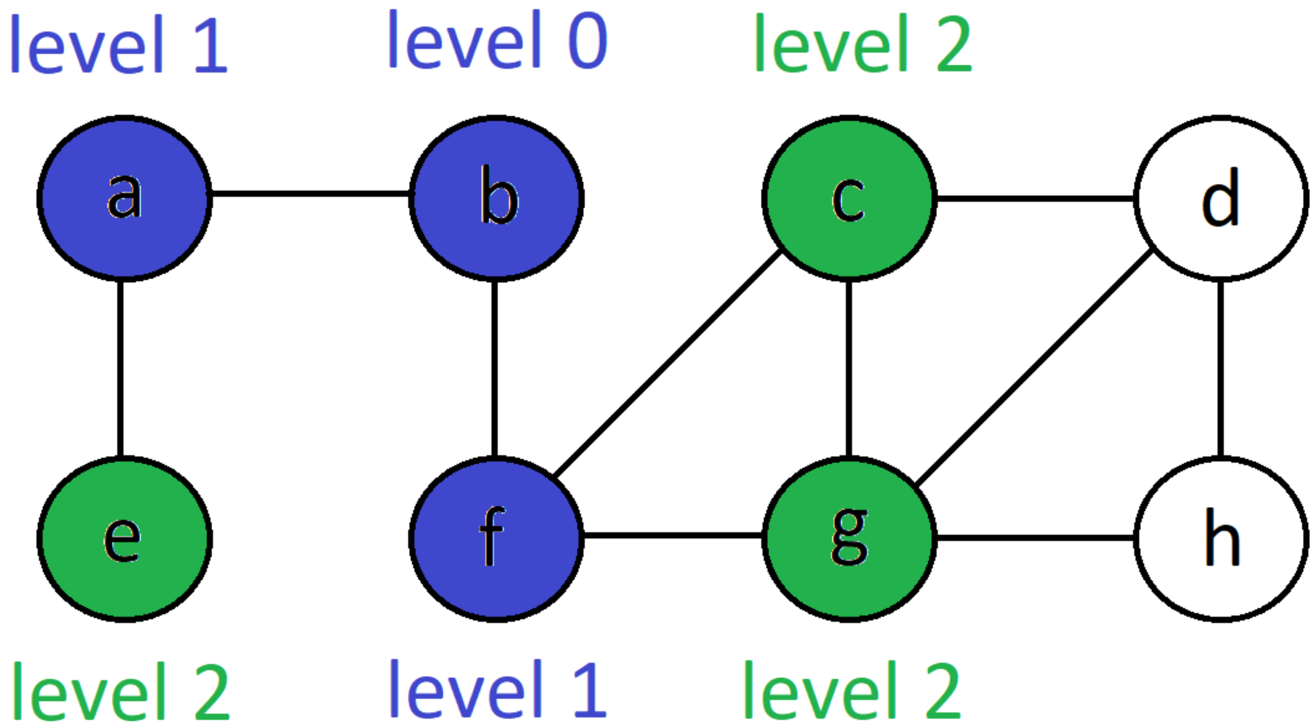
queue: **f a**
dequeue: **b**

BFS Traversal - Undirected



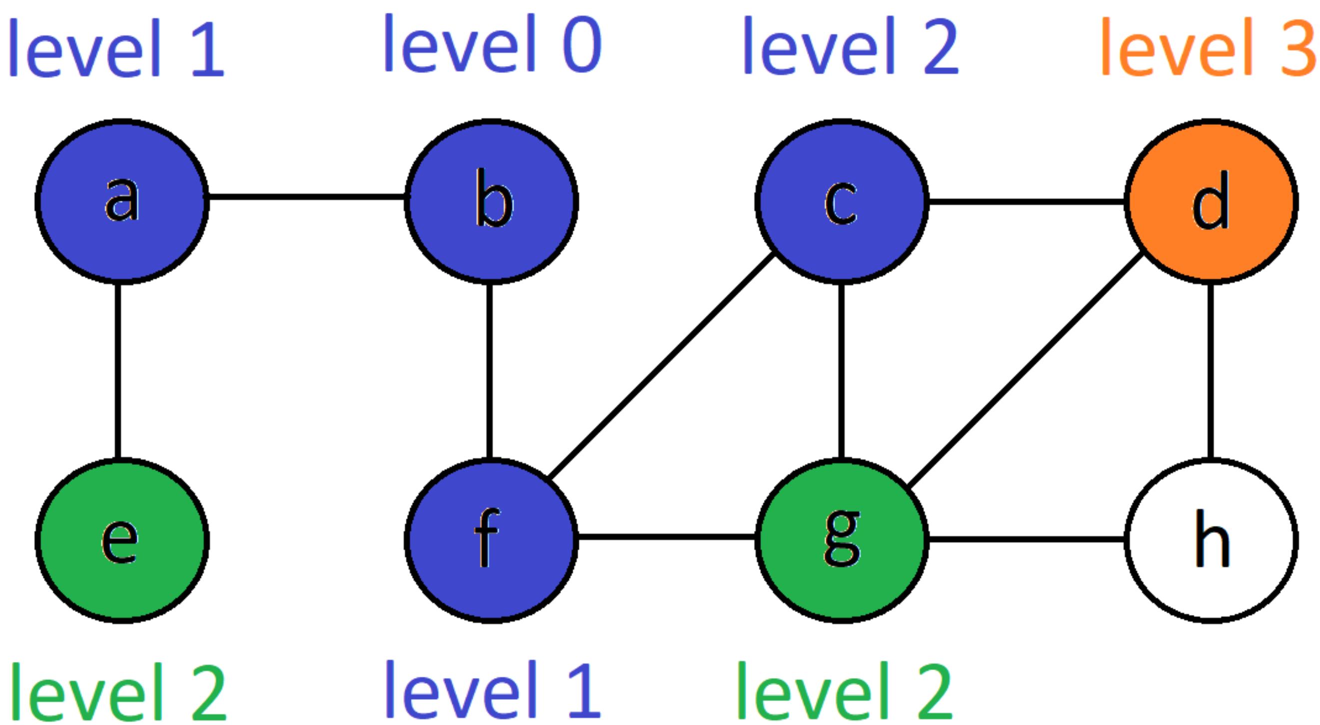
queue: **a c g**
dequeue: **f**

BFS Traversal - Undirected



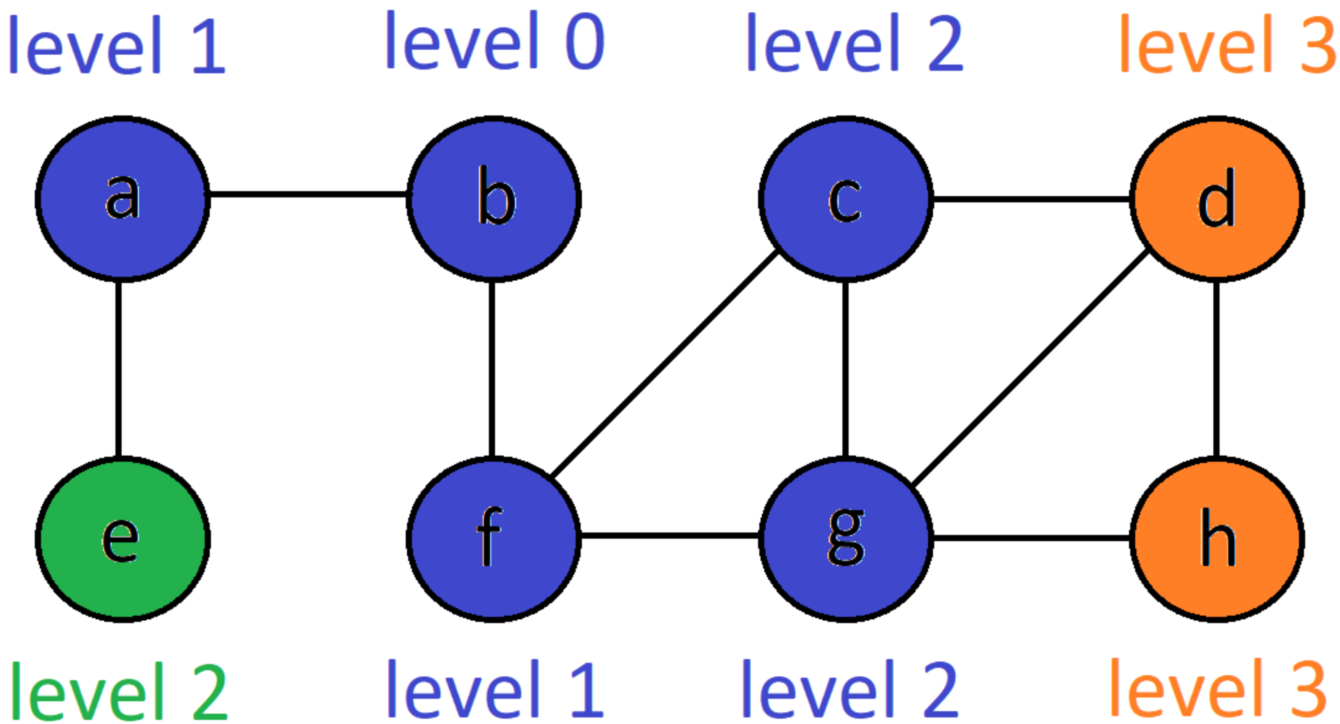
queue: **c g e**
dequeue: **a**

BFS Traversal - Undirected



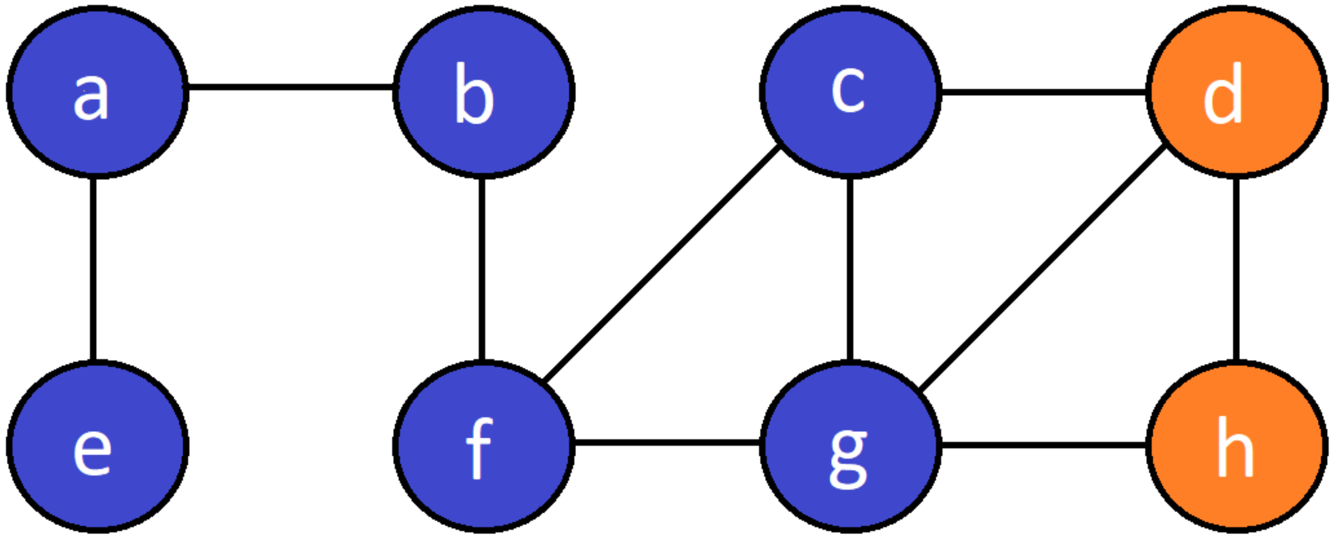
queue: **g e d**
dequeue: **c**

BFS Traversal - Undirected



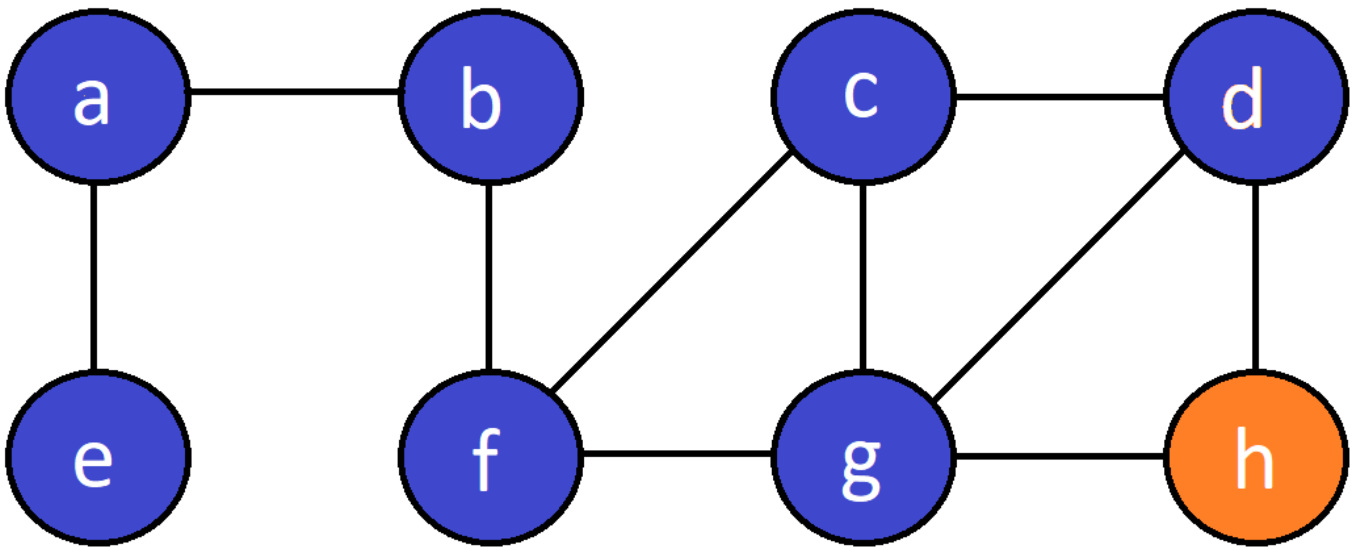
queue: **e d h**
dequeue: **g**

BFS Traversal - Undirected



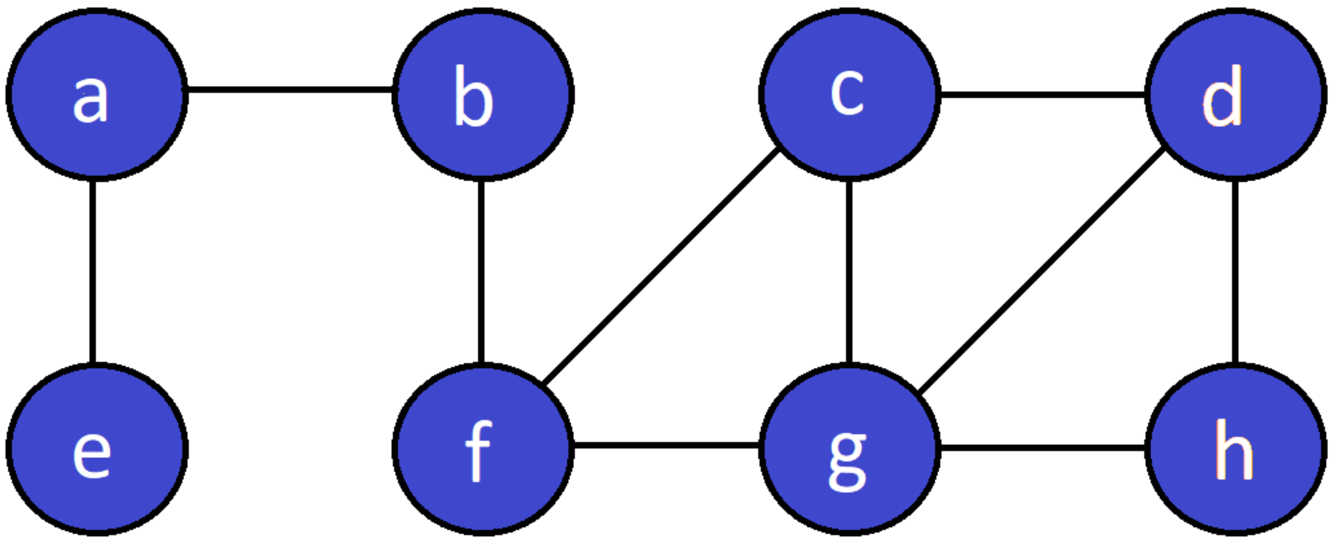
queue: **d h**
dequeue: **e**

BFS Traversal - Undirected



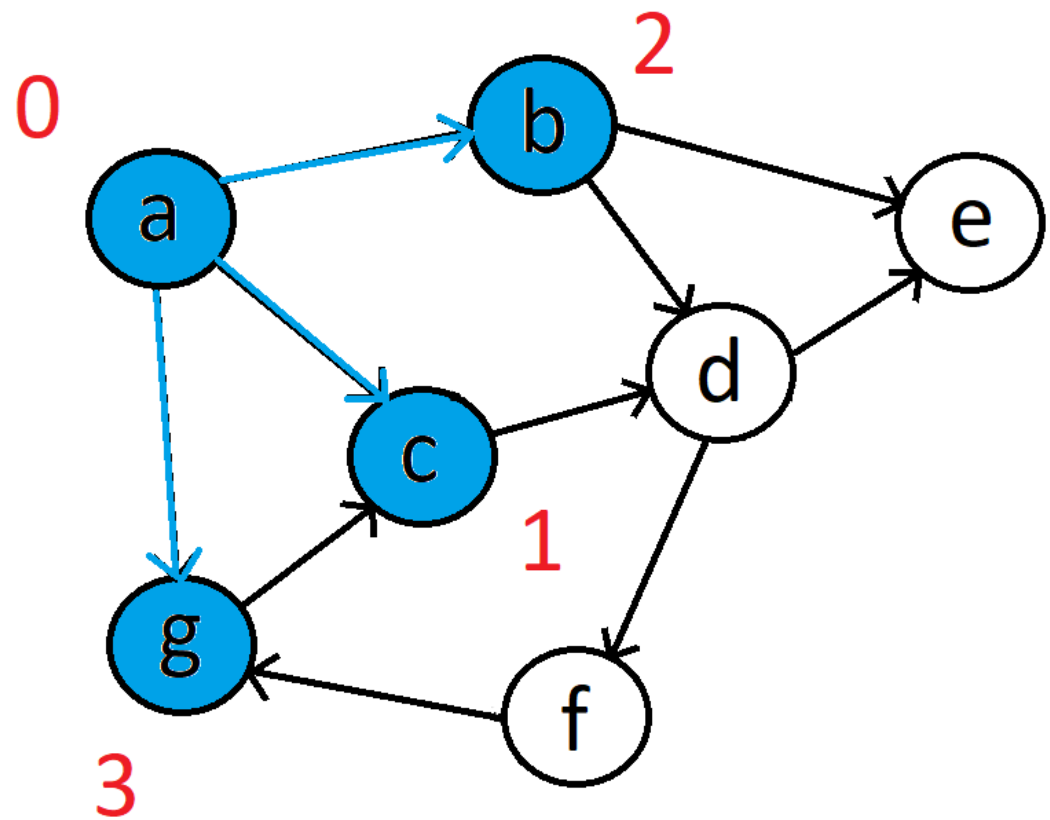
queue: **h**
dequeue: **d**

BFS Traversal - Undirected

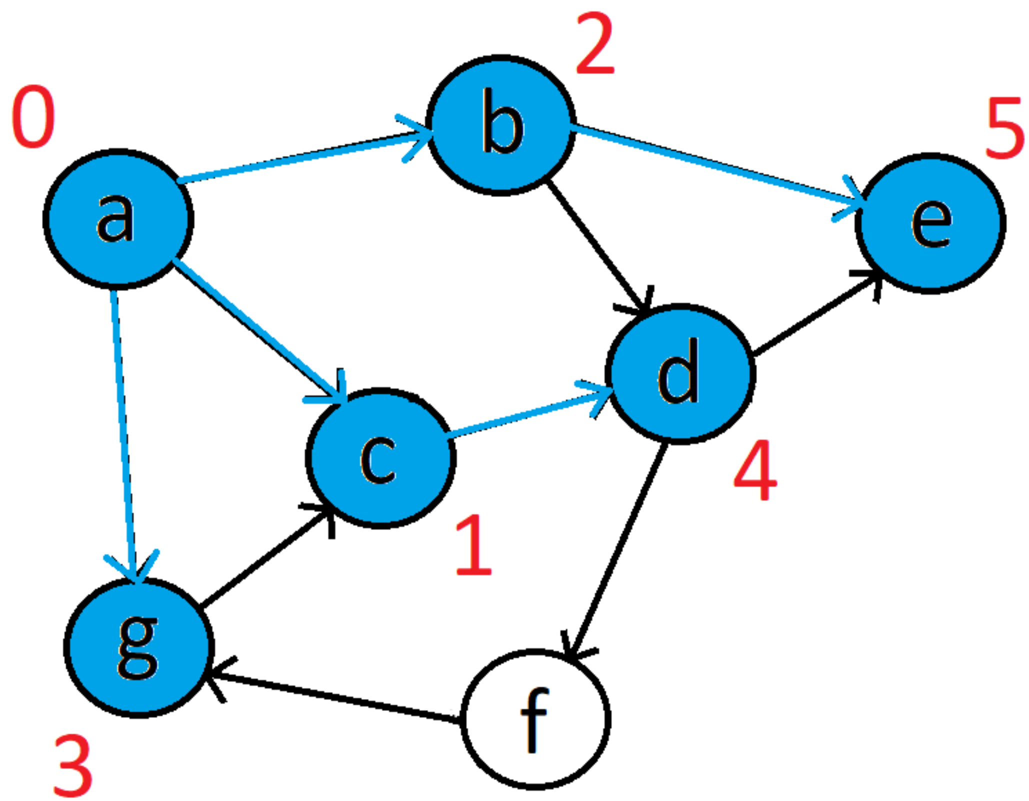


queue: -
dequeue: **h**
Traversal is done!

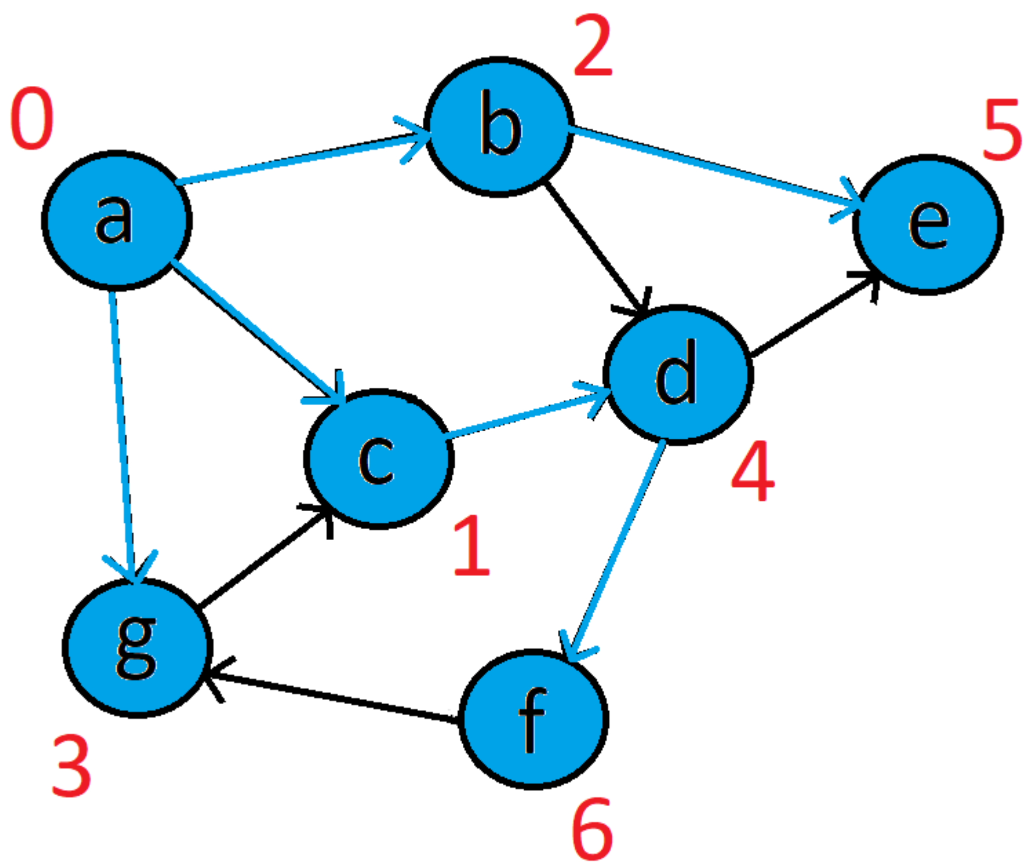
BFS Traversal - Directed



BFS Traversal - Directed



BFS Traversal - Directed



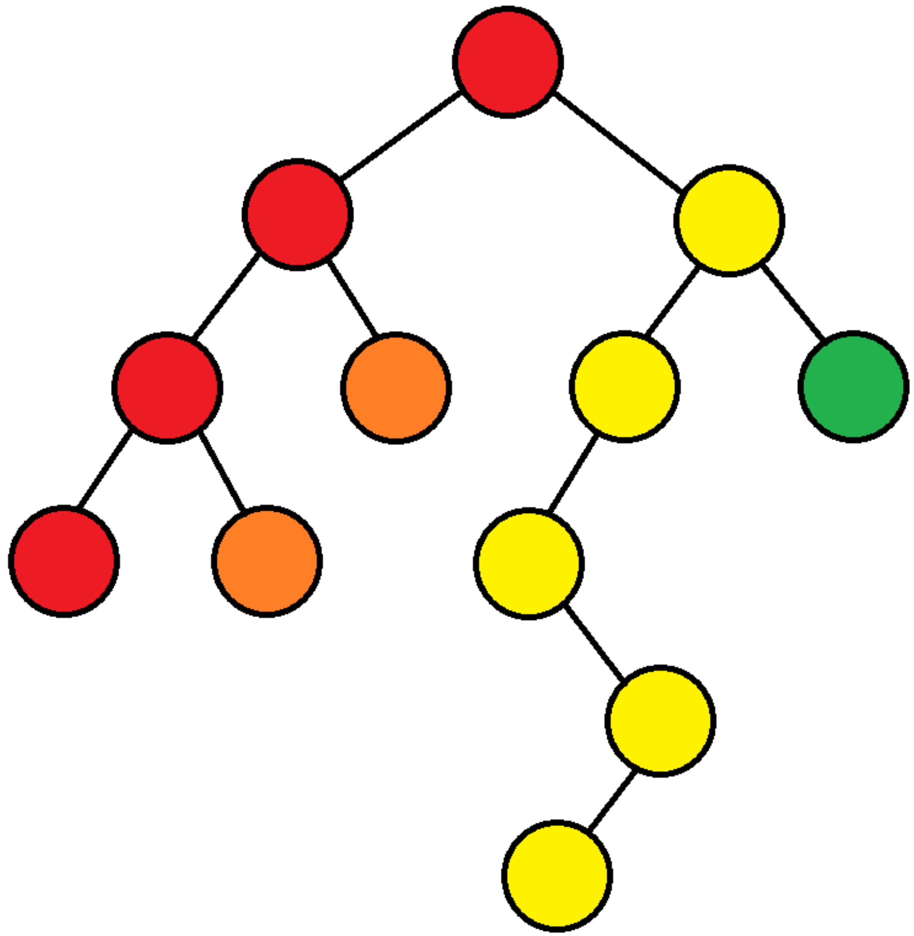
Depth-First Traversal

- Similar to pre-order traversal of a binary tree
  - Read the data at the node first, then move on to the left subtree, and then to the right subtree.
  - Could use post-order or in-order methods
- Once you start down a path, don't stop until you get to the end (a leaf). Make your way back up (backtrack) and then start on a new path.
- Commonly written using recursion
- Use stacks

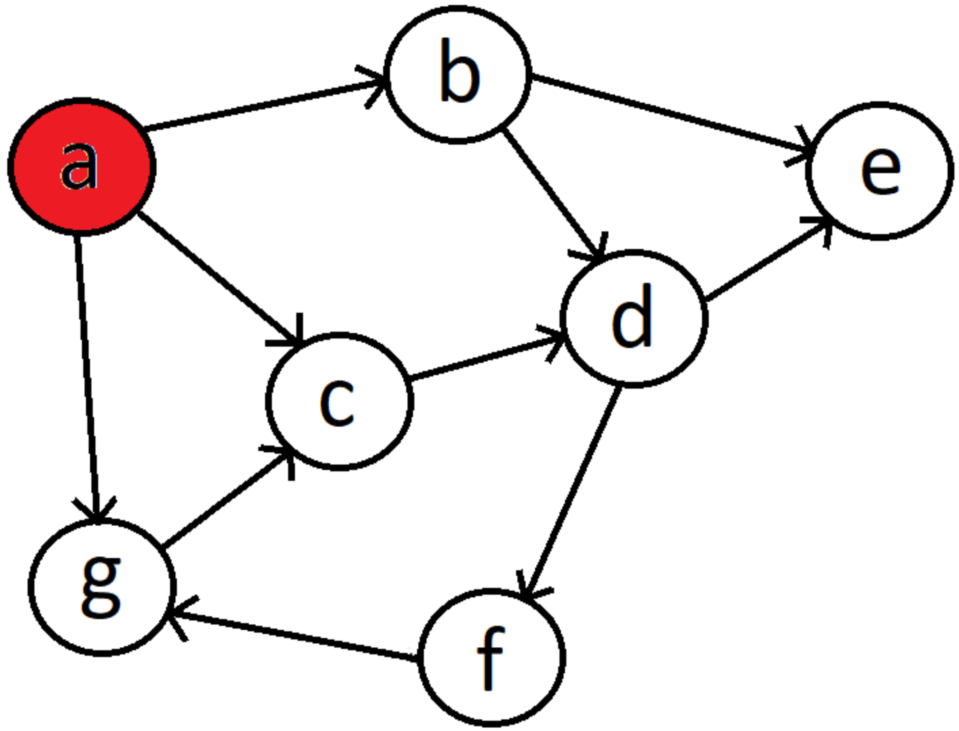
Depth-First Traversal

- Remember! The traversal is not done until all vertices are reached.

Depth-First Traversal

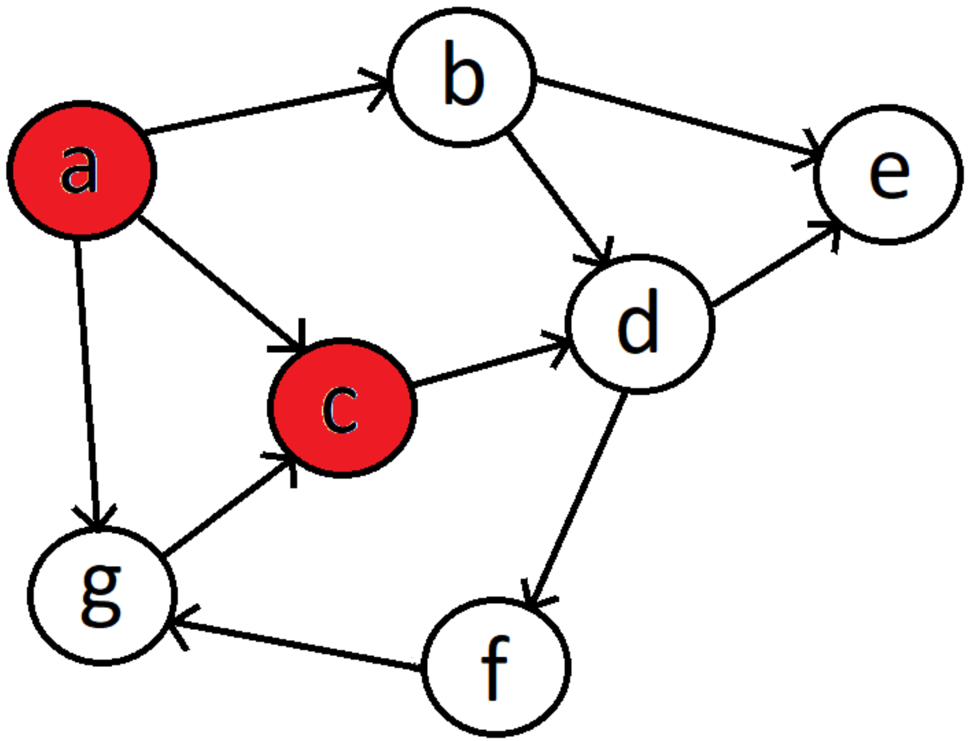


DFS Traversal - Directed



Stack: a
Output: a

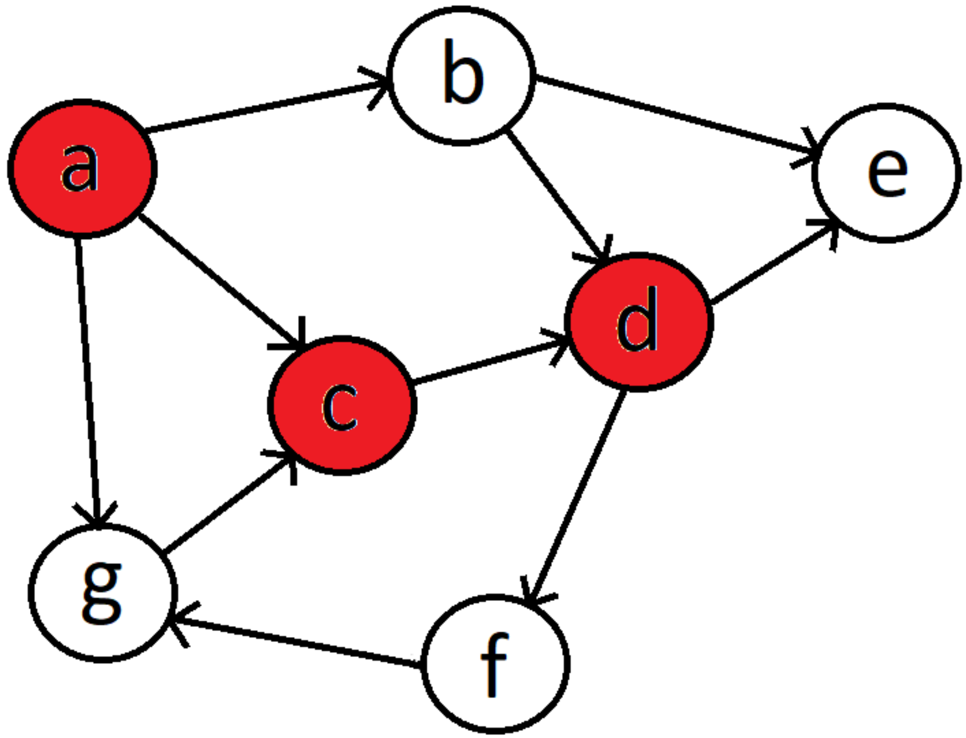
DFS Traversal - Directed



Stack: a c

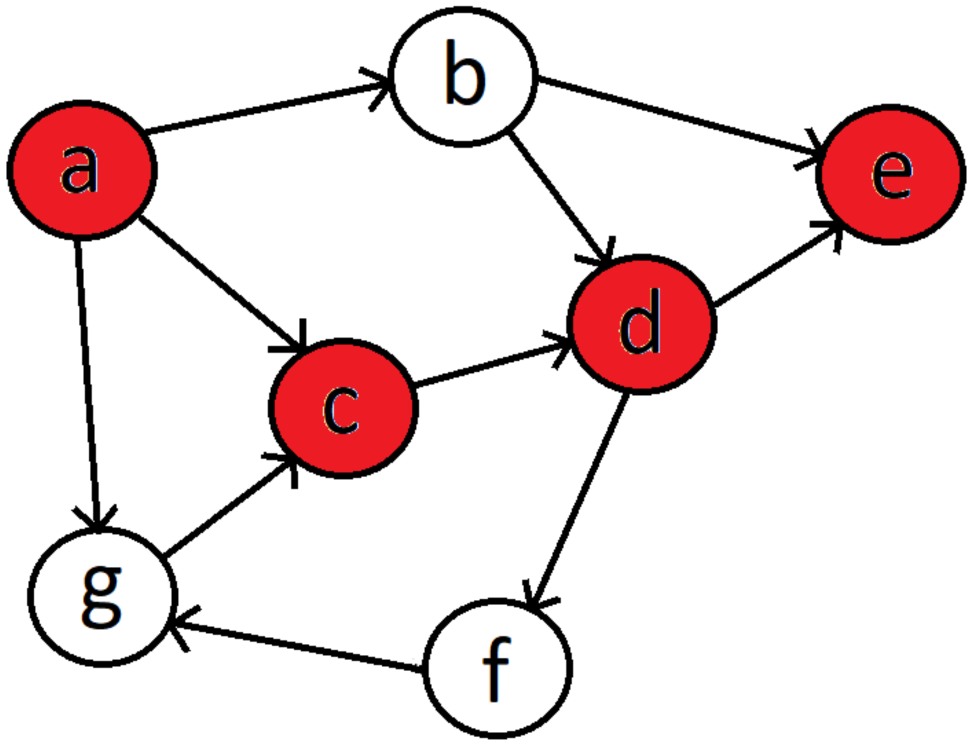
Output: a c

DFS Traversal - Directed



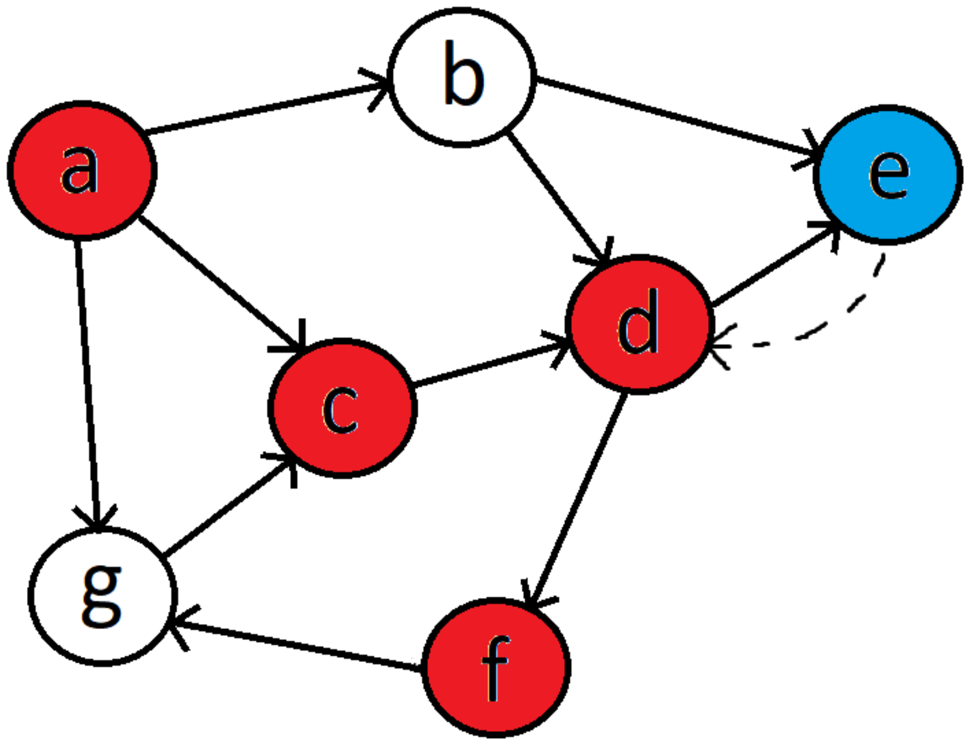
Stack: a c d
Output: a c d

DFS Traversal - Directed



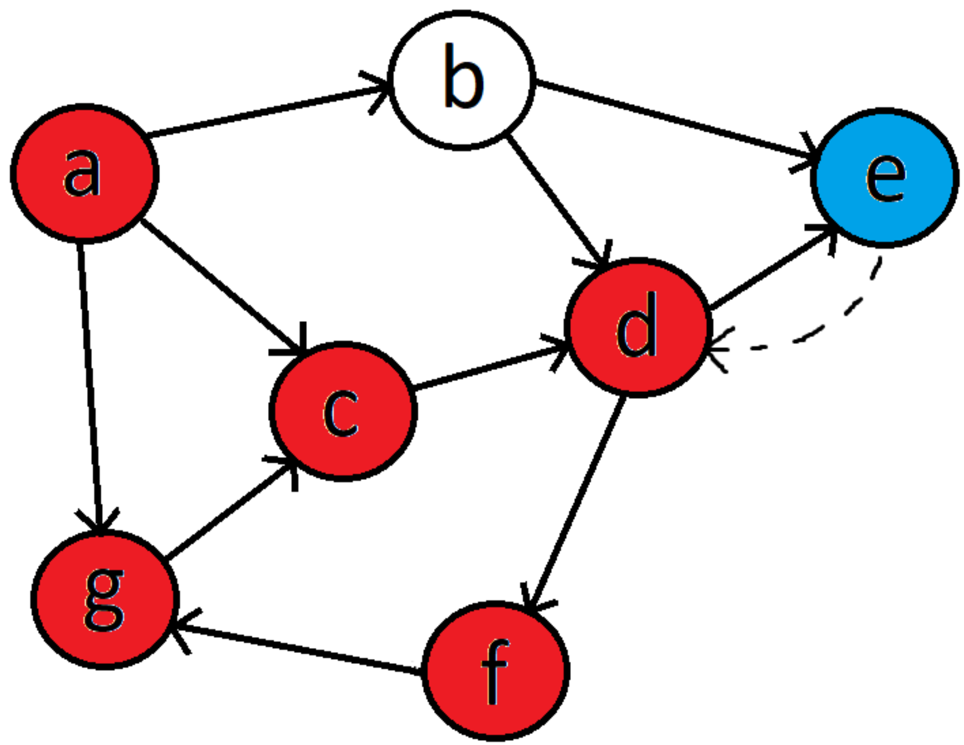
Stack: a c d ~~e~~
Output: a c d e

DFS Traversal - Directed



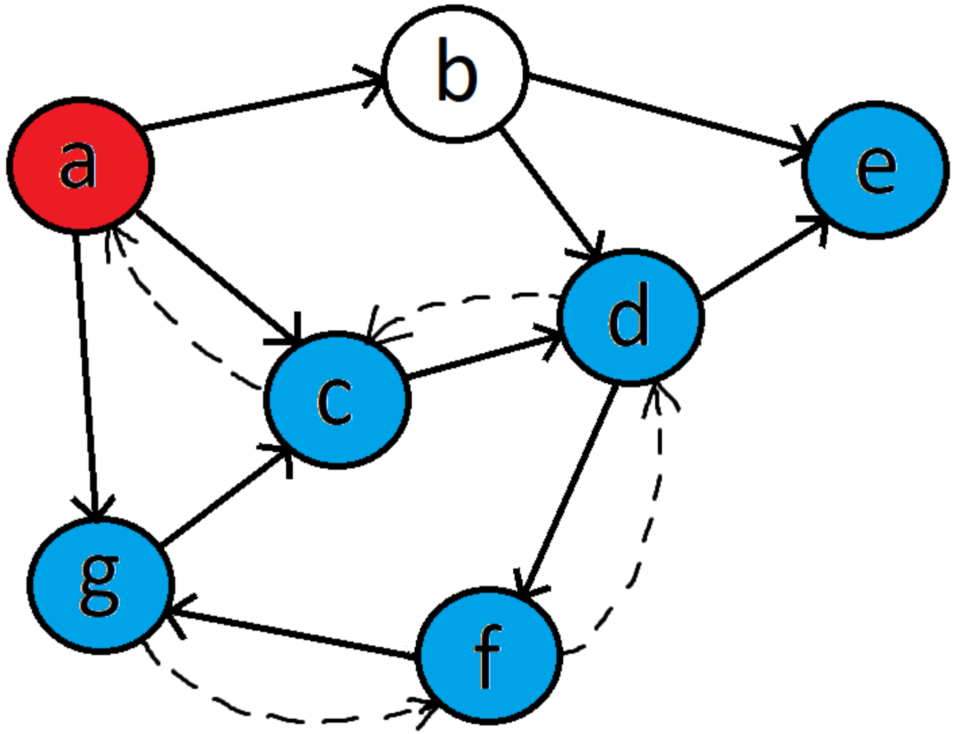
Stack: a c d f
Output: a c d e f

DFS Traversal - Directed



Stack: a c d f g
Output: a c d e f g

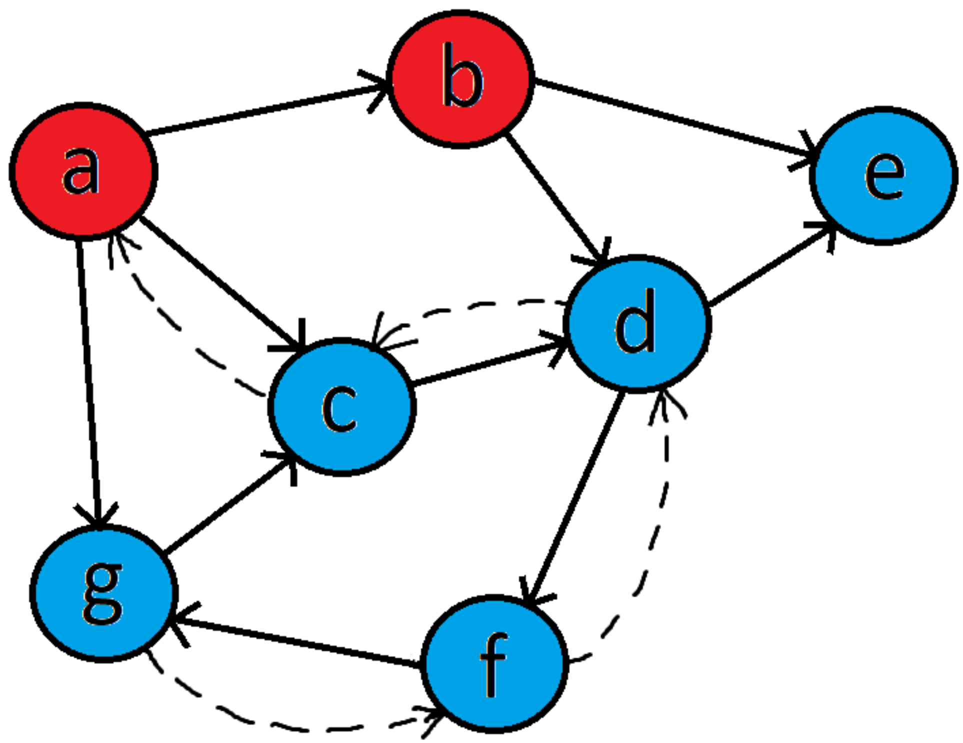
DFS Traversal - Directed



Stack: a

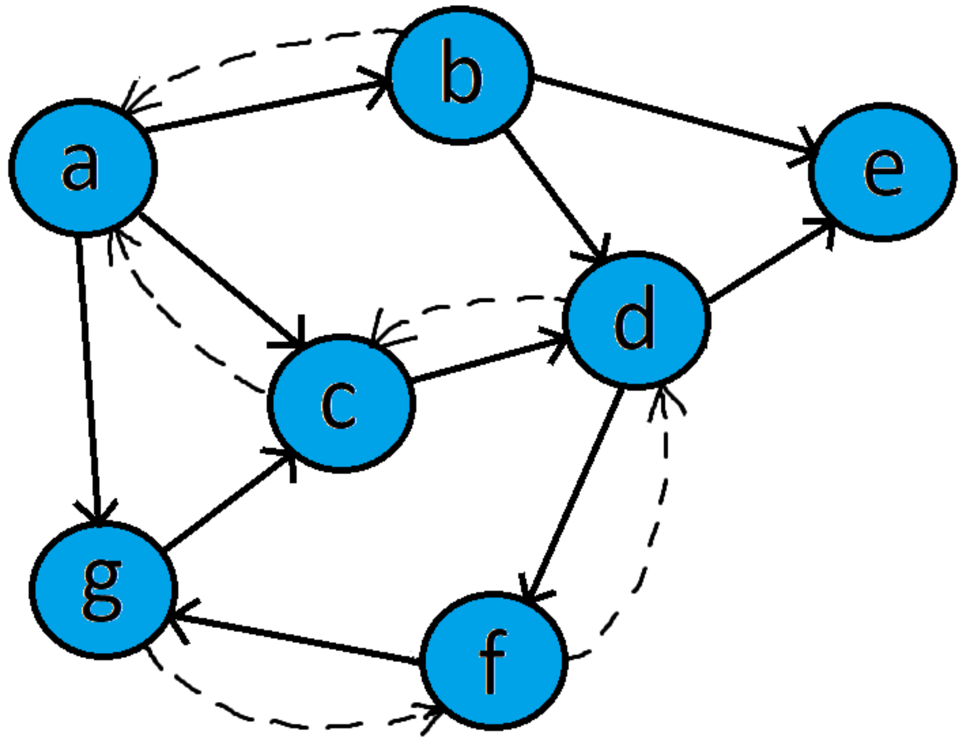
Output: a c d e f g

DFS Traversal - Directed



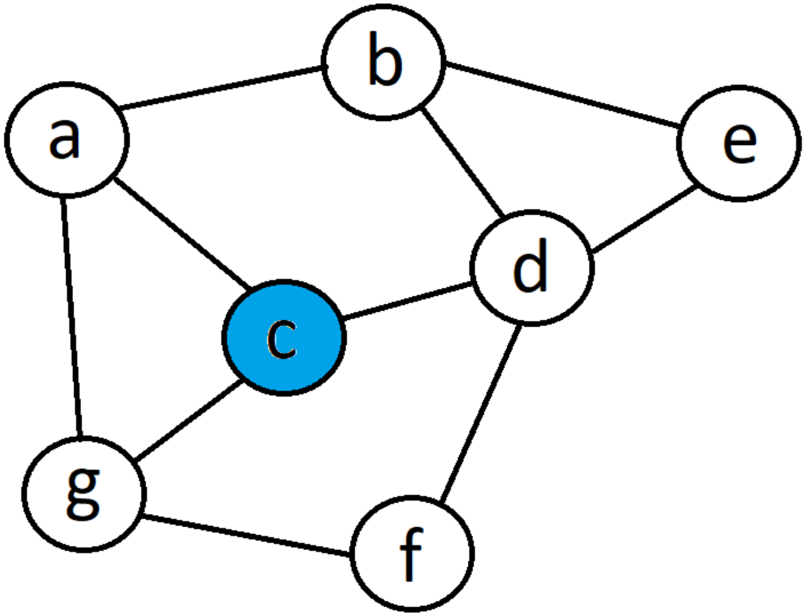
Stack: a b
Output: a c d e f g

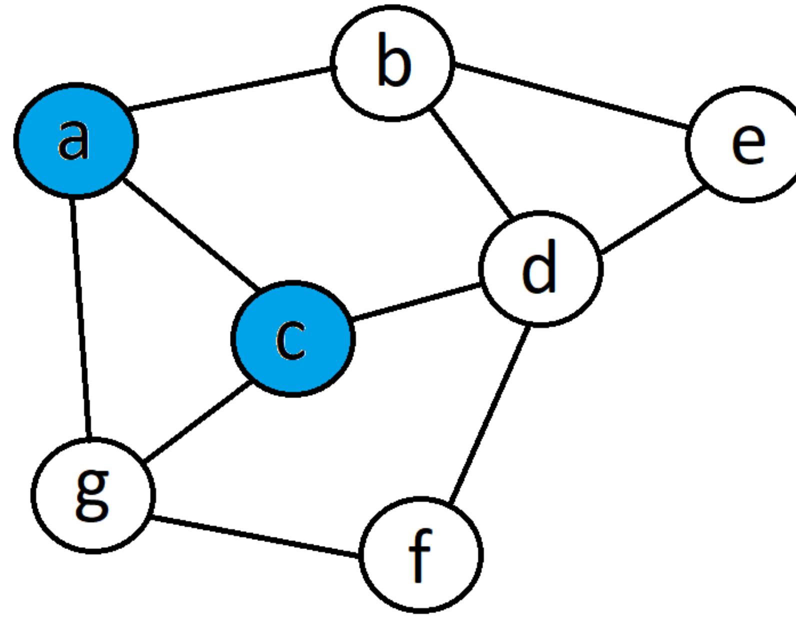
DFS Traversal - Directed



Stack: -
Output: a c d e f g b

DFS Traversal - Undirected

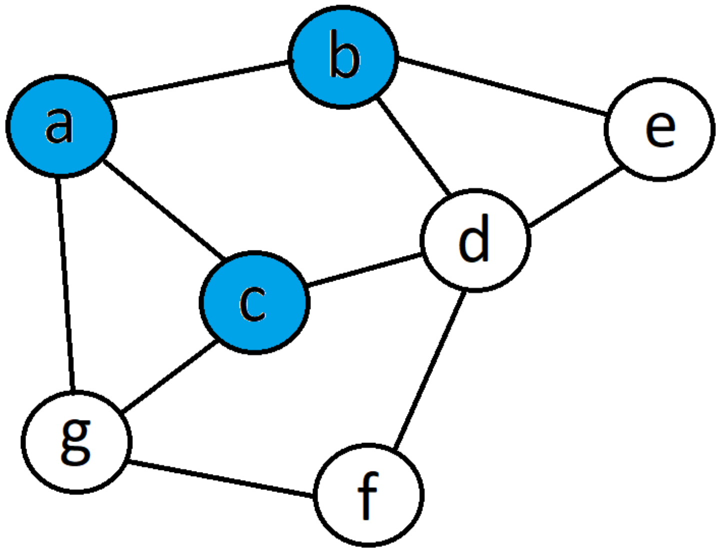


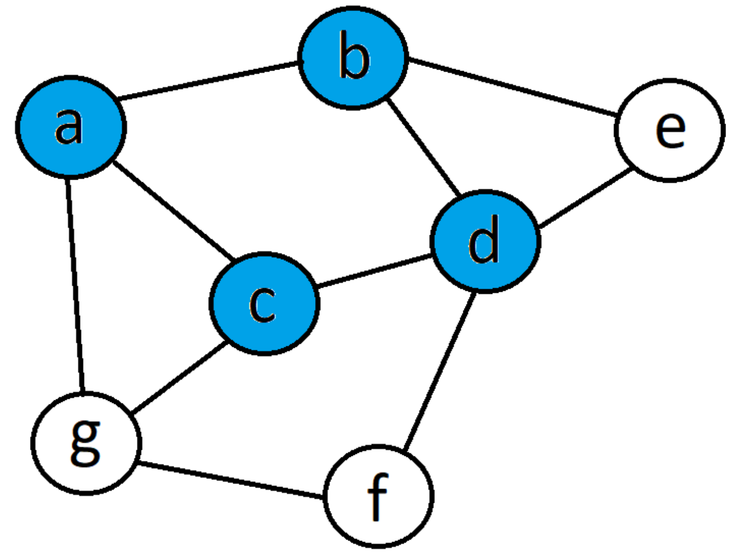


Output: c

Output: c a

DFS Traversal - Undirected





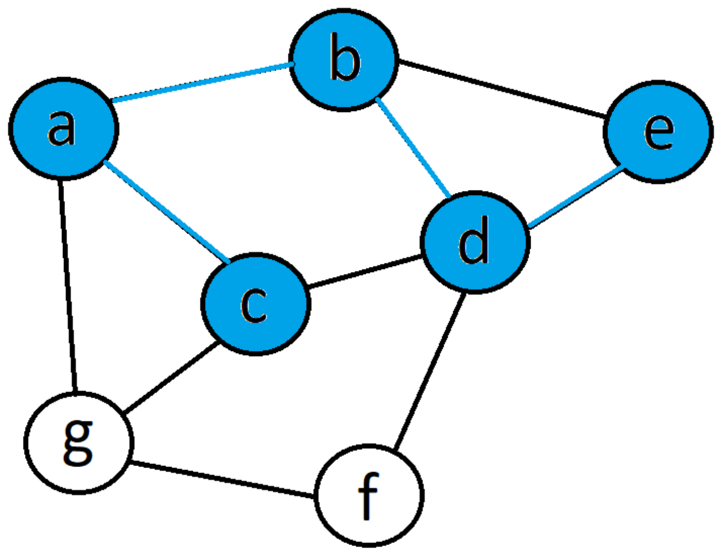
-b checks a, but a has been visited

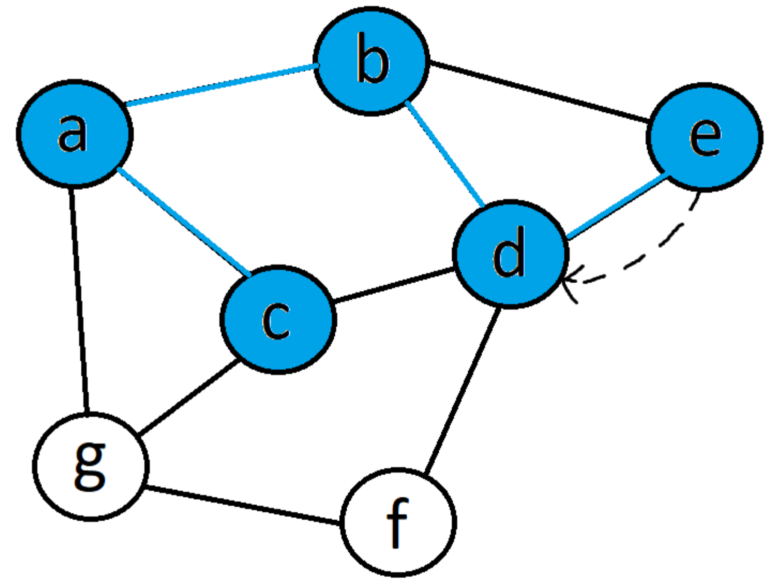
-It moves to d. (see diagram right)

Output: c a b

Output: c a b d

DFS Traversal - Undirected





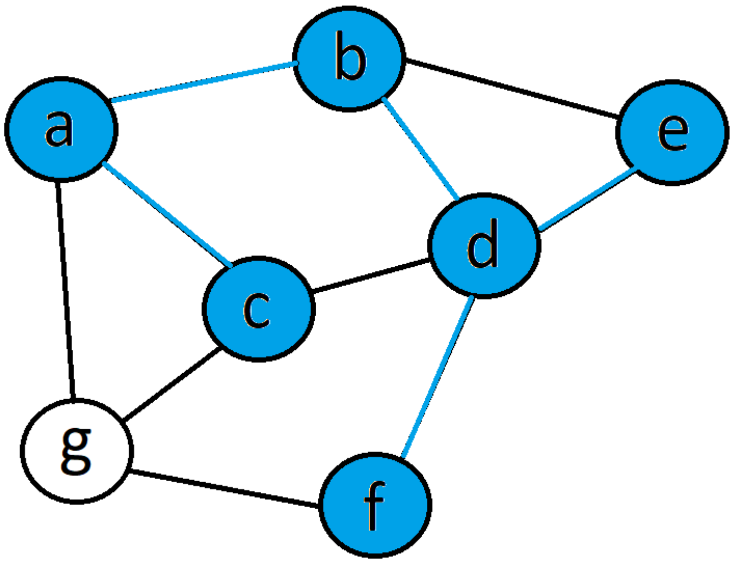
-d checks b, but b has been visited
-d checks c, but c has been visited
-d checks e, which it visits

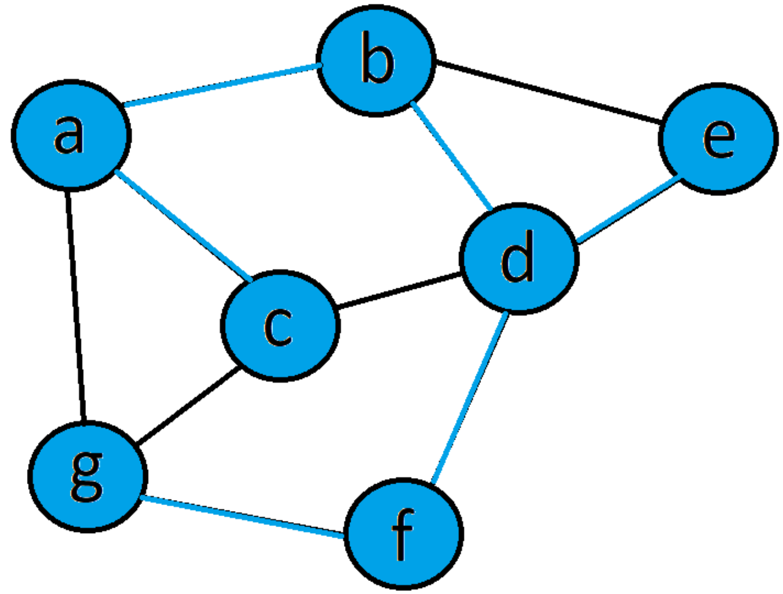
Output: c a b d e

-e checks b, which has been visited
-e checks d, which has been visited
-e can't go anywhere else, backtrack!

Output: c a b d e

DFS Traversal - Undirected





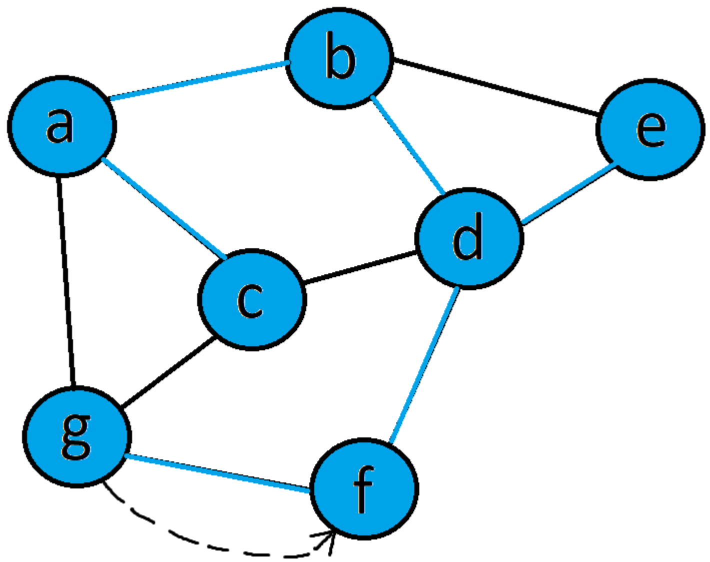
-d checks f, which it visits
-Go to diagram on right

-f checks d, which has been visited
-f checks g, which it visits

Output: c a b d e f

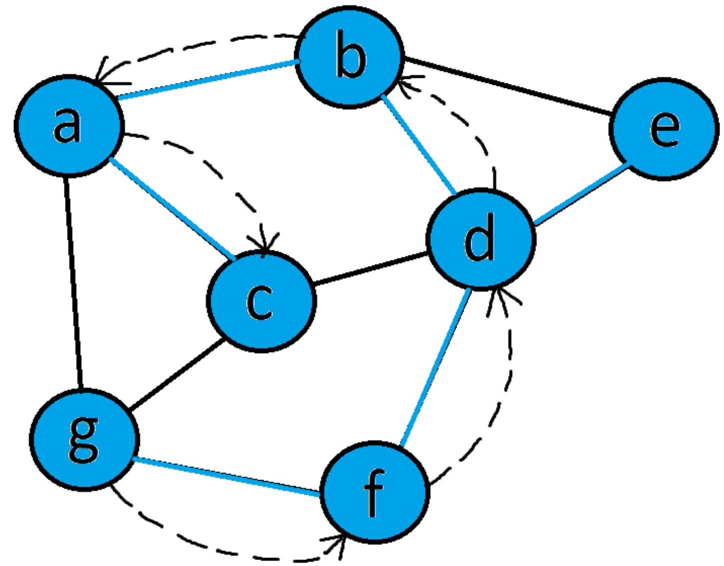
Output: c a b d e f g

DFS Traversal - Undirected



-g checks a, which has been visited
-g checks c, which has been visited
-g checks f, which has been visited
-g can't go anywhere, so backtrack!

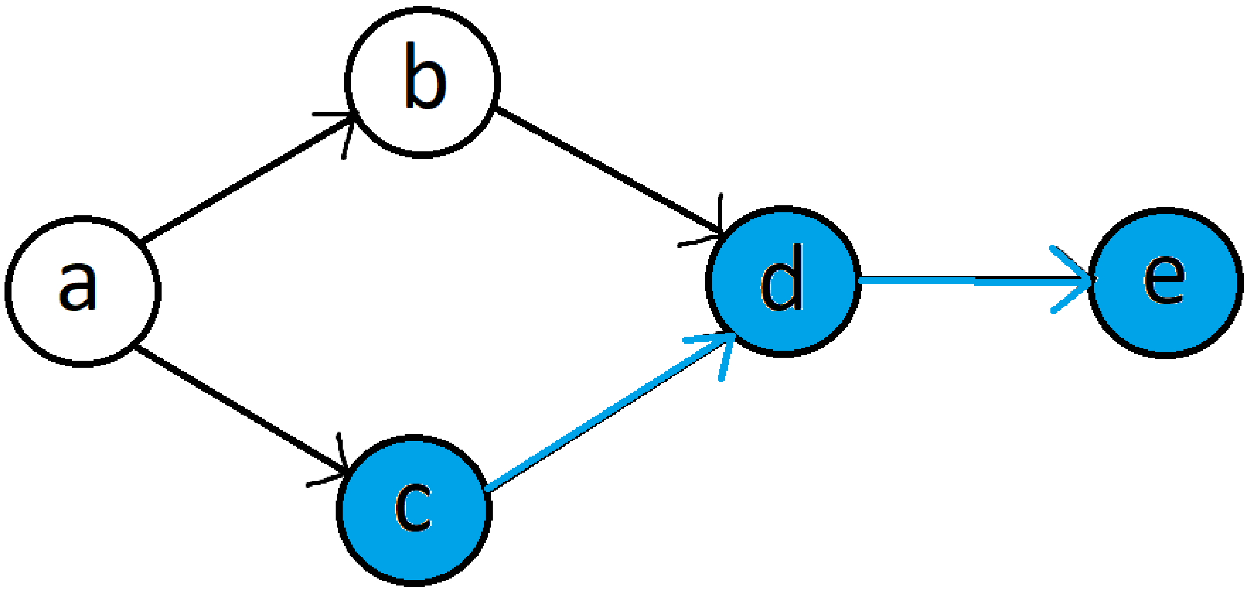
Output: c a b d e f g



-f backtracks to d
-d backtracks to b
-b checks e, which has been visited
-b backtracks to a
-a checks c, then g, which were visited
-a backtracks to c
-c checks d and g, which were visited

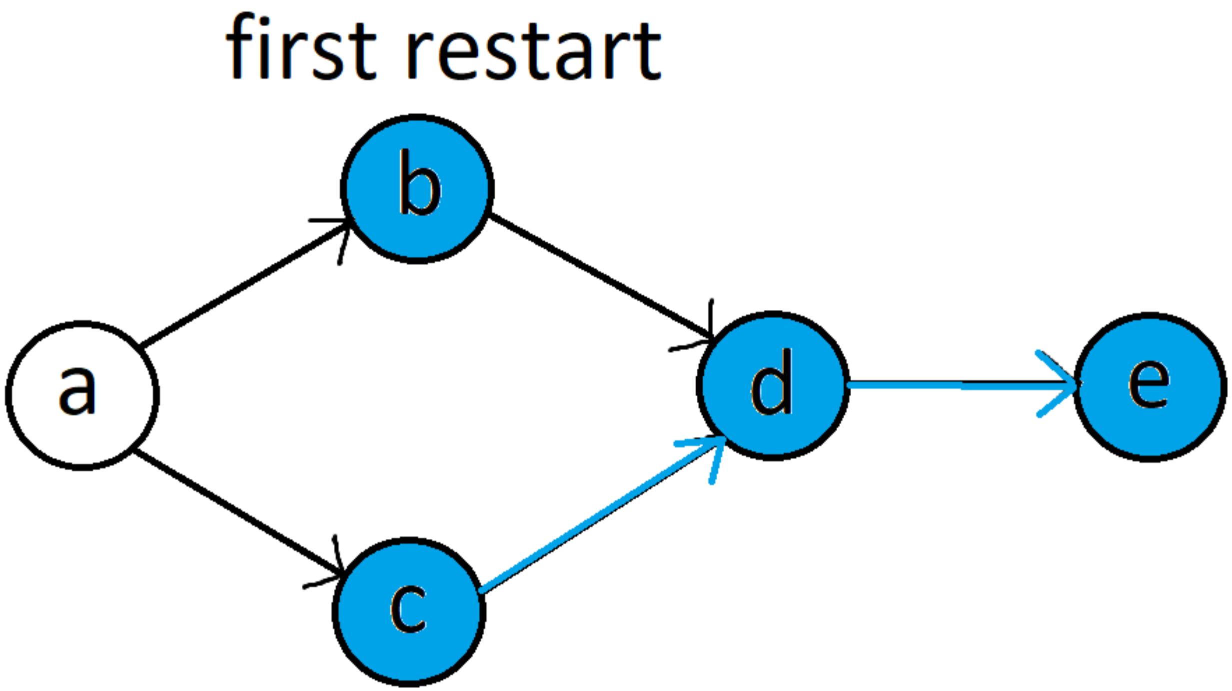
Output: c a b d e f g

DFS Traversal - Restart



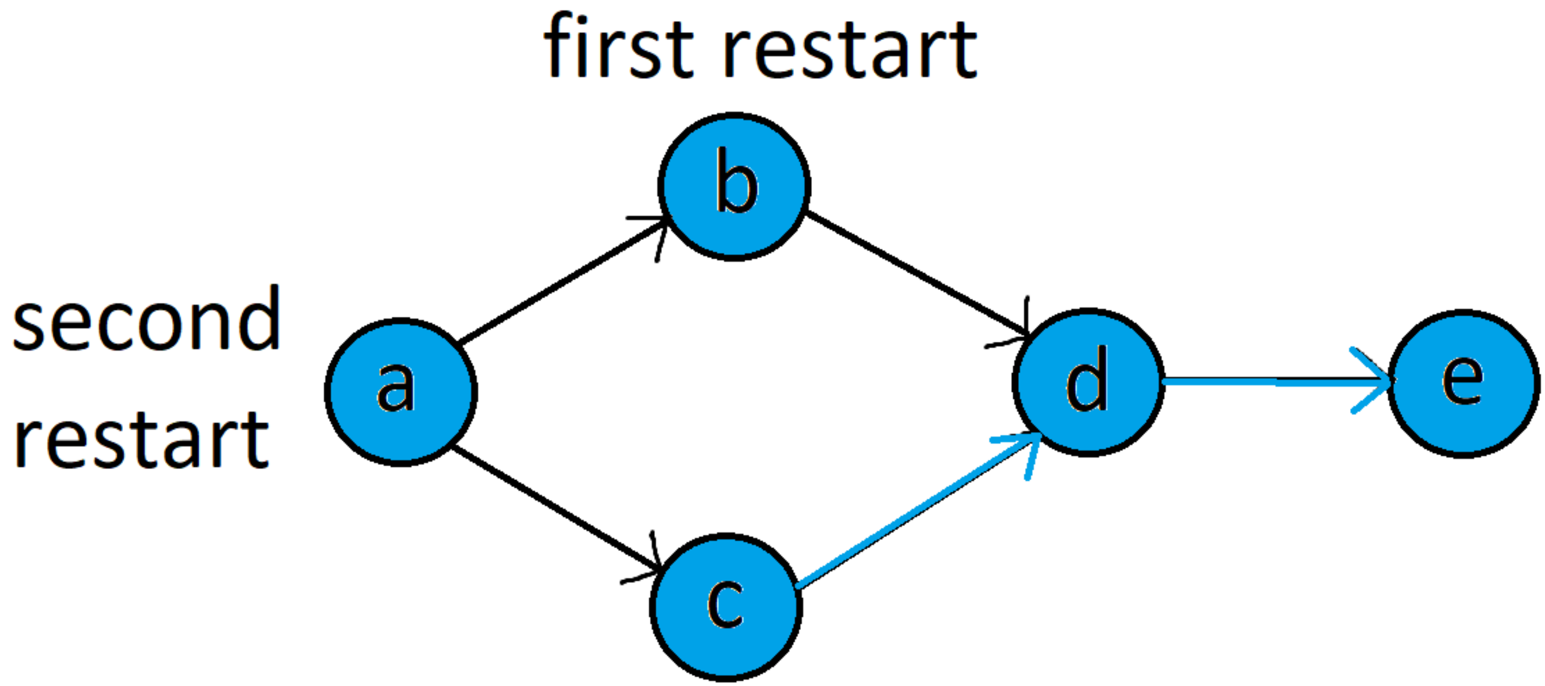
Output: c d e

DFS Traversal - Restart



Output: c d e b

DFS Traversal - Restart



Output: c d e b a

Time Complexity

- For BFS, we iterate through a vertex's adjacency list once (i.e. for each vertex, look at its edges/neighboring nodes once).
- If the graph is undirected, an edge appears twice, in two different adjacency lists. If the graph is directed, an edge appears only once.
- So, the time complexity is O(V + E):
  - Visit *n* nodes (i.e. read data and enqueue children) takes constant time. Each node is visited once so the time to use a BFS is O(n), where n = # of nodes.
  - Number of edges (e) in the adjacency list to be visited.
    - ***For undirected graph, 2|E|***
    - ***For directed graph, |E|***
  - Conclusion: the runtime complexity of BFS graph traversal is O(V+E)
- Time complexity for DFS is also O(V+E)
- Space complexity depends on size of the queue at its worst, which could be up to O(n).

Questions?