ADVANCED DATA STRUCTURES AND ALGORITHMS

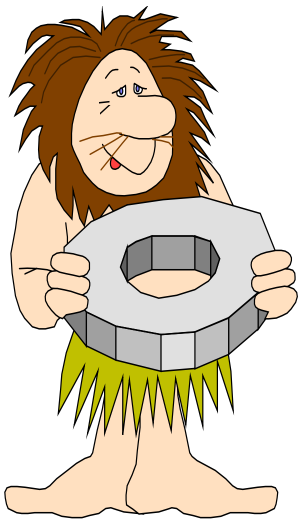
Associate Professor Dr. Raed Ibraheem Hamed

University of Human Development, College of Science and Technology
Computer Science Department

2015 – 2016

What this Lecture is about:

- Graph Traversals (Search)
- Breadth-first search
- BFS: Level-by-level traversal
- BFS for general graphs
- Handling vertices
- Interesting features of BFS
- Interesting features of BFS



Graph Traversals (Search)

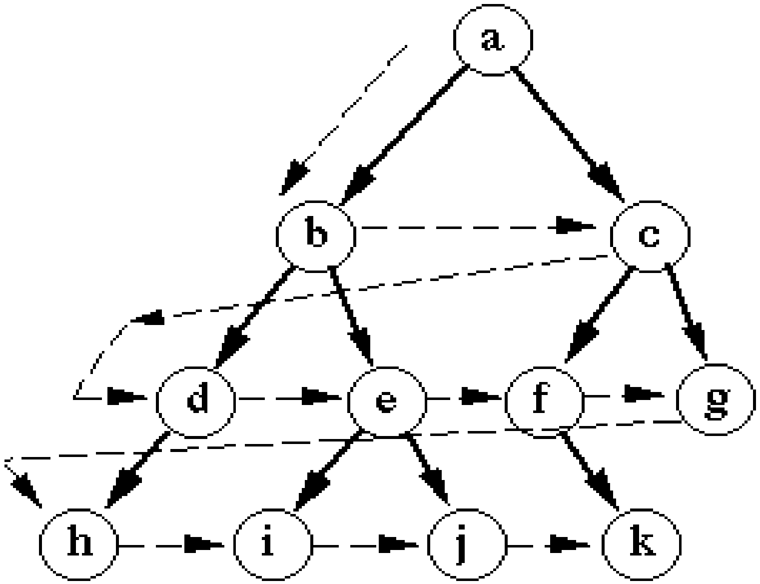
- We have covered some of these with binary trees
  - **Breadth-first search (BFS)**
  - **Depth-first search (DFS)**
- A traversal (search):
  - An algorithm for systematically exploring a graph
  - Visiting (all) vertices
  - Until finding a goal vertex or until no more vertices

Breadth-first search

- One of the simplest algorithms
- Also one of the most important
  - It forms the basis for MANY graph algorithms

BFS: Level-by-level traversal

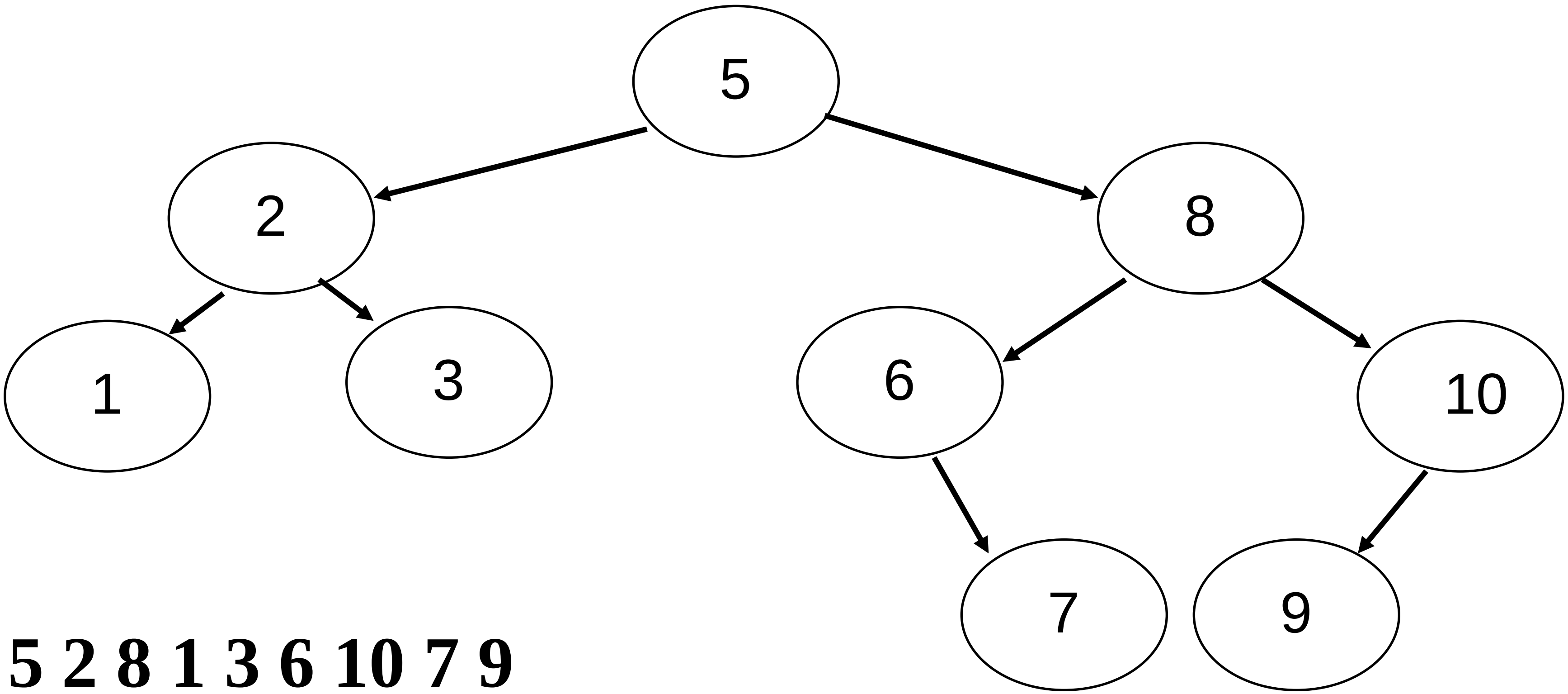
- Given a starting vertex s
- Visit all vertices at increasing distance from s
  - Visit all vertices at distance k from s
  - Then visit all vertices at distance k+1 from s
  - Then ….



Breadth-first search

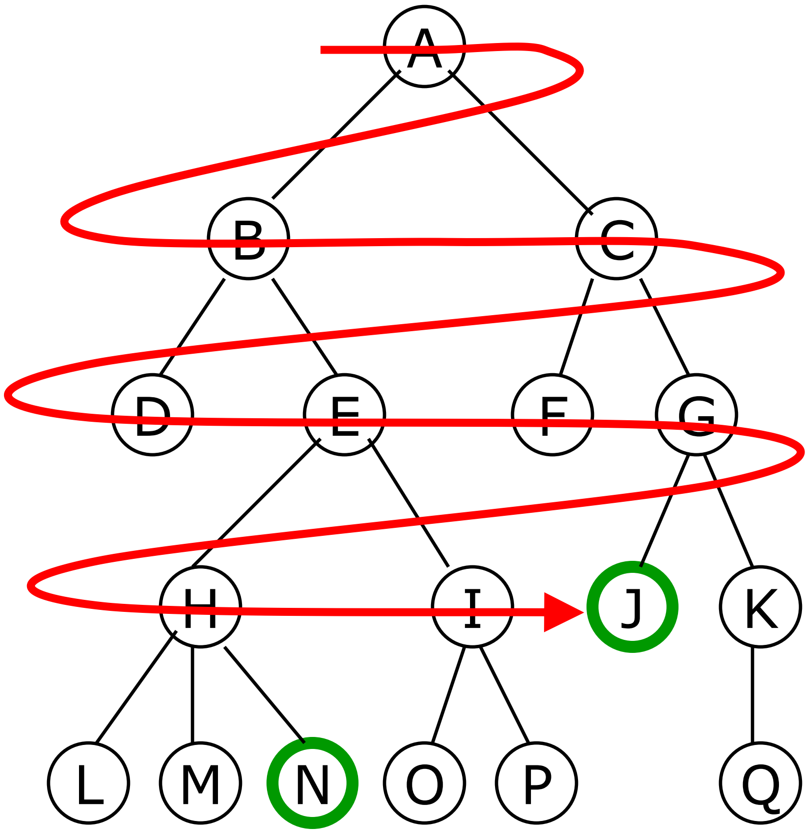
BFS in a binary tree (reminder)

BFS: visit all siblings before their descendents



5 2 8 1 3 6 10 7 9

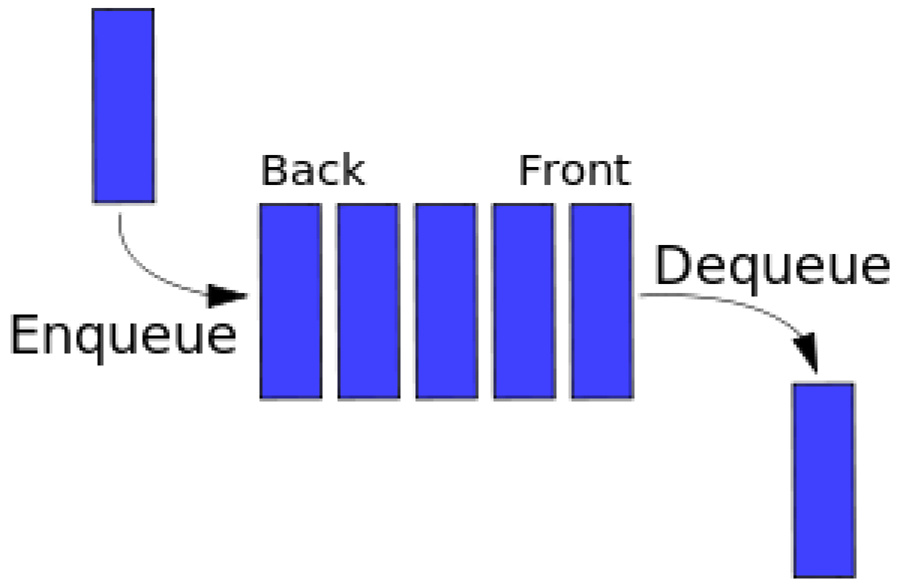
Breadth-first searching



- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

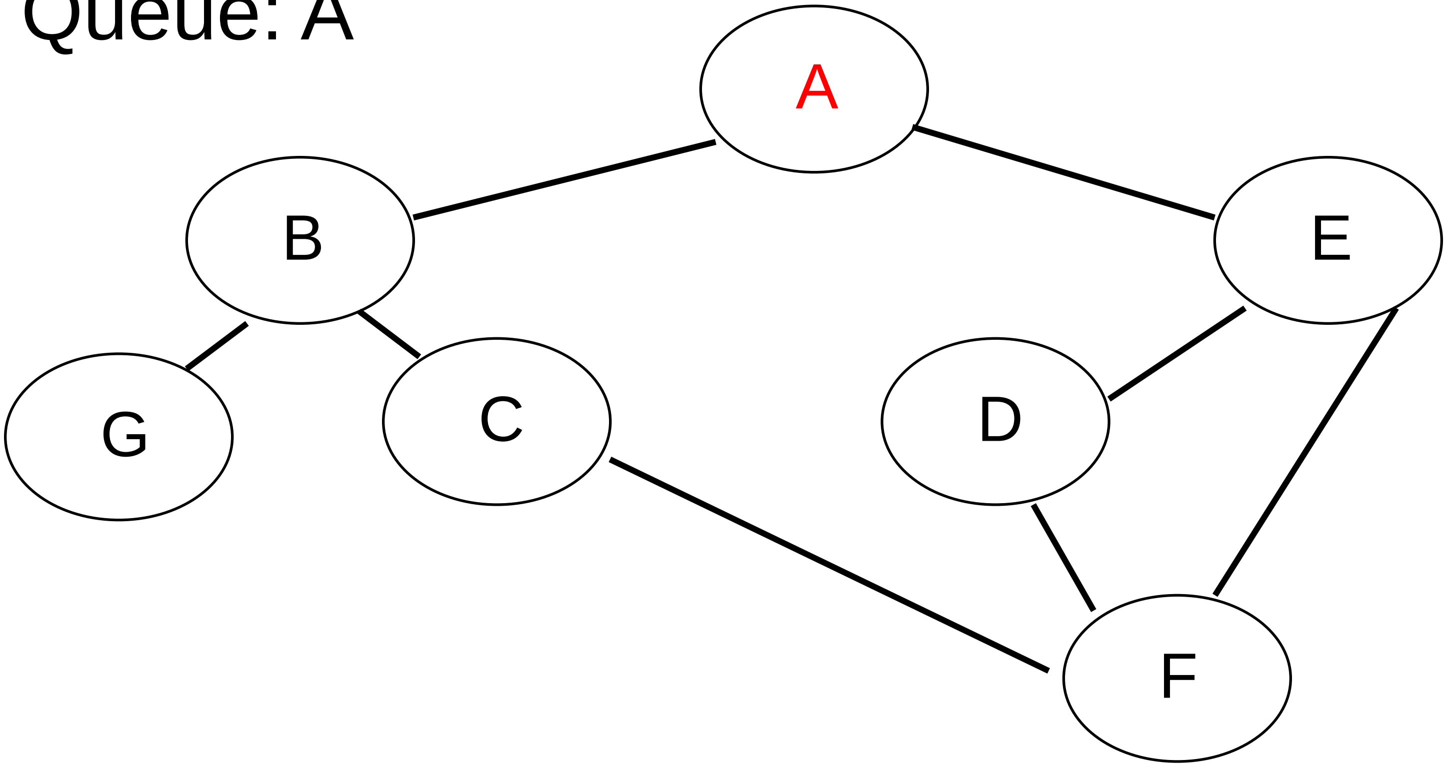
Queue

The queue is First-In-First-Out (FIFO) data structure.



BFS For General Graphs

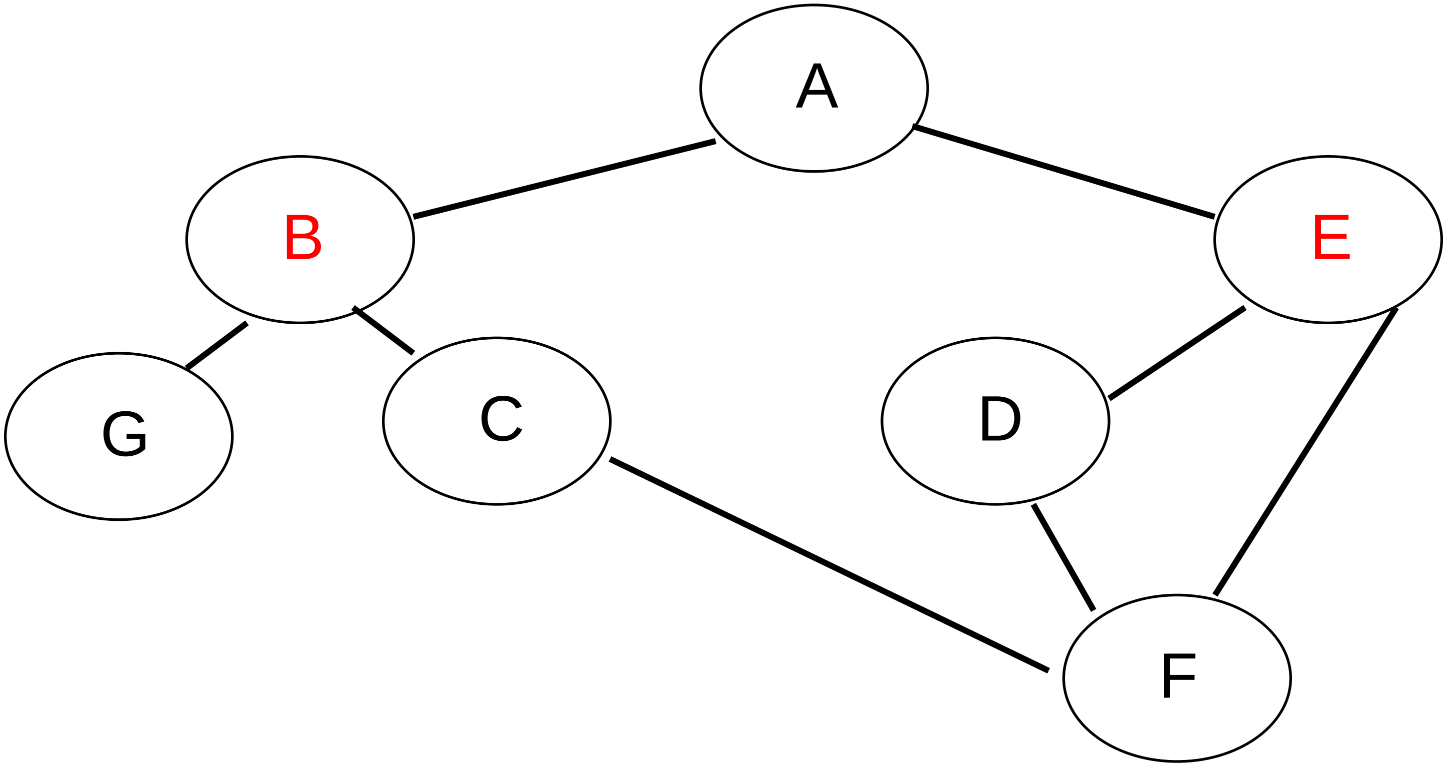
Queue: A



Start with A. Put in the queue (marked red)

Example.

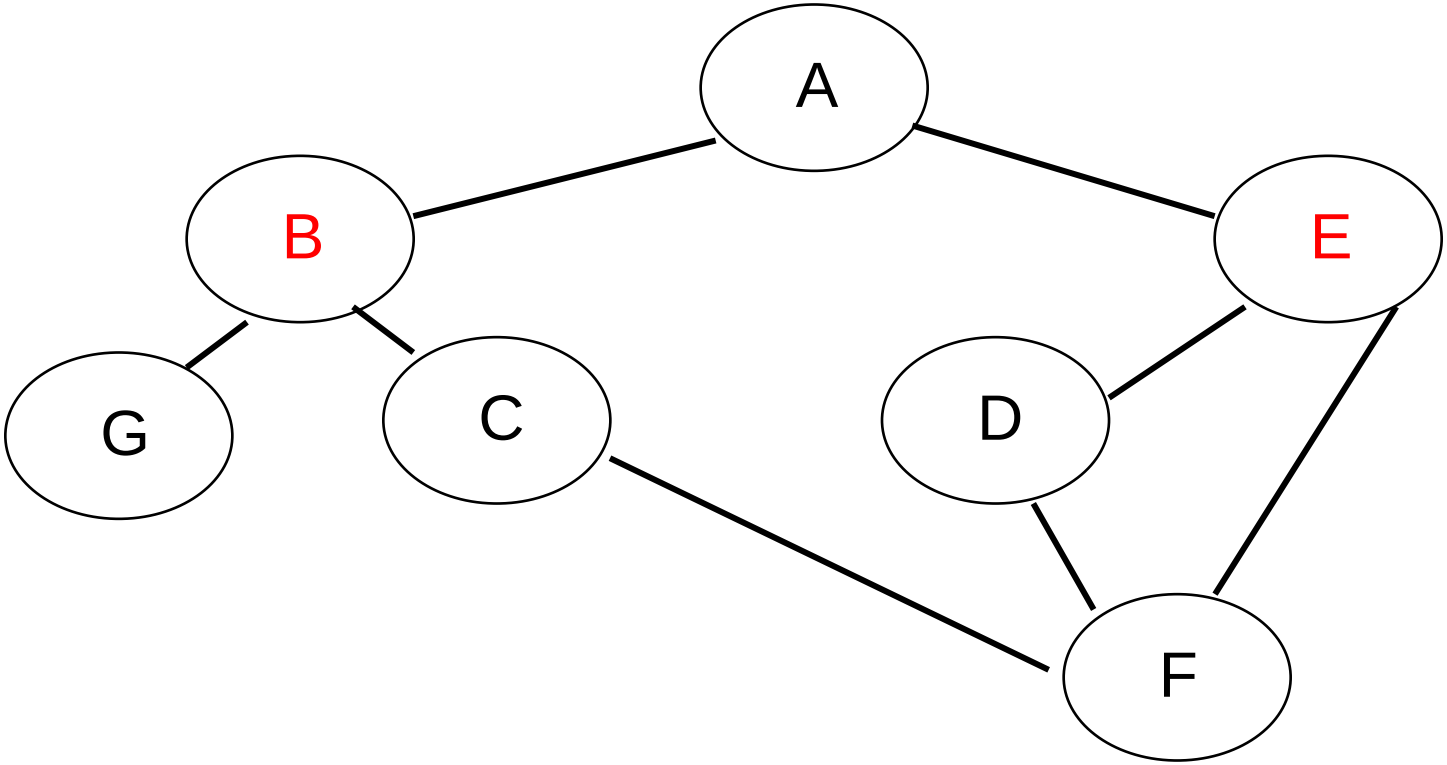
Queue: A B E



B and E are next

Example.

Queue: A B E C G D F

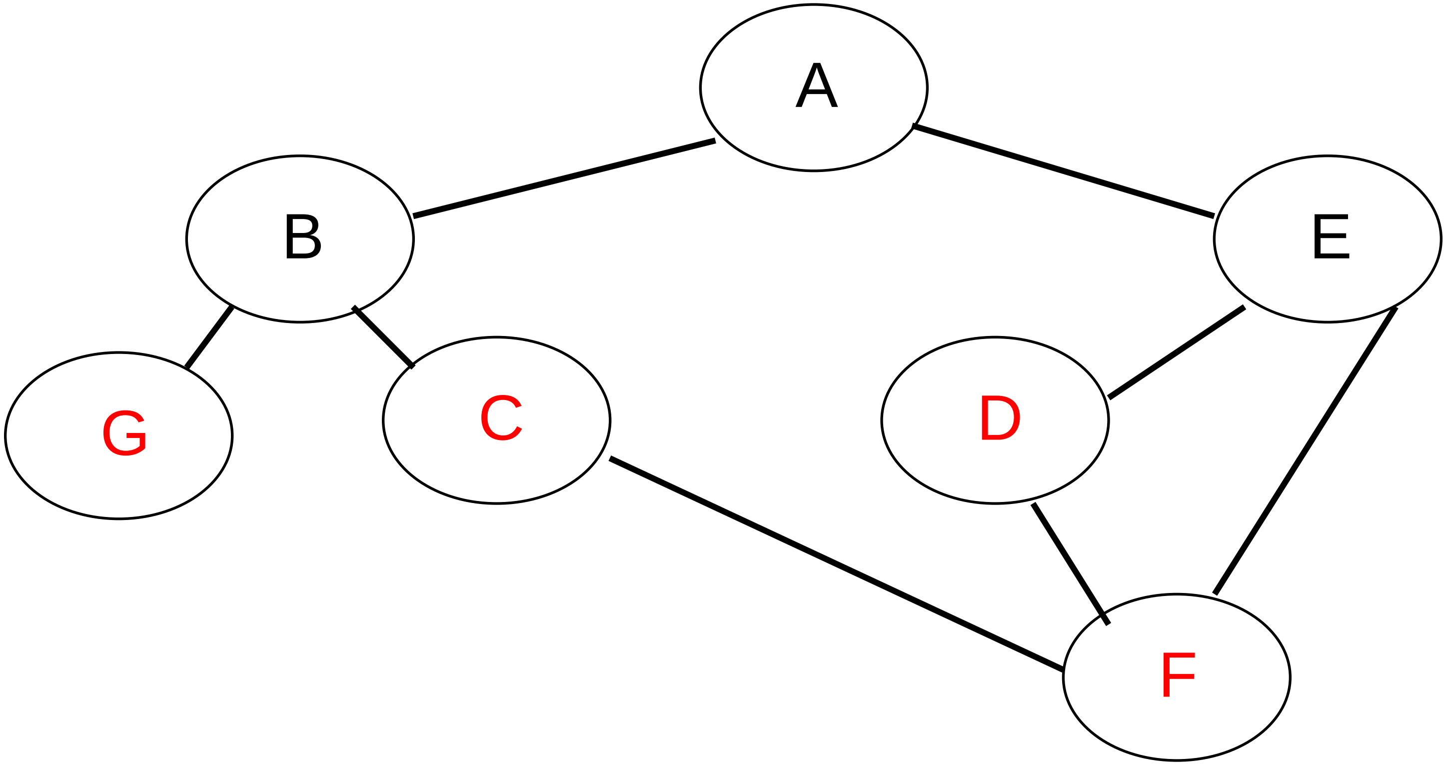


When we go to B, we put G and C in the queue

When we go to E, we put D and F in the queue

Example.

Queue: A B E C G D F

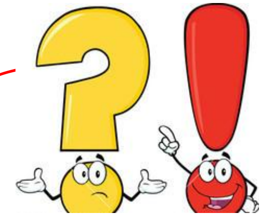


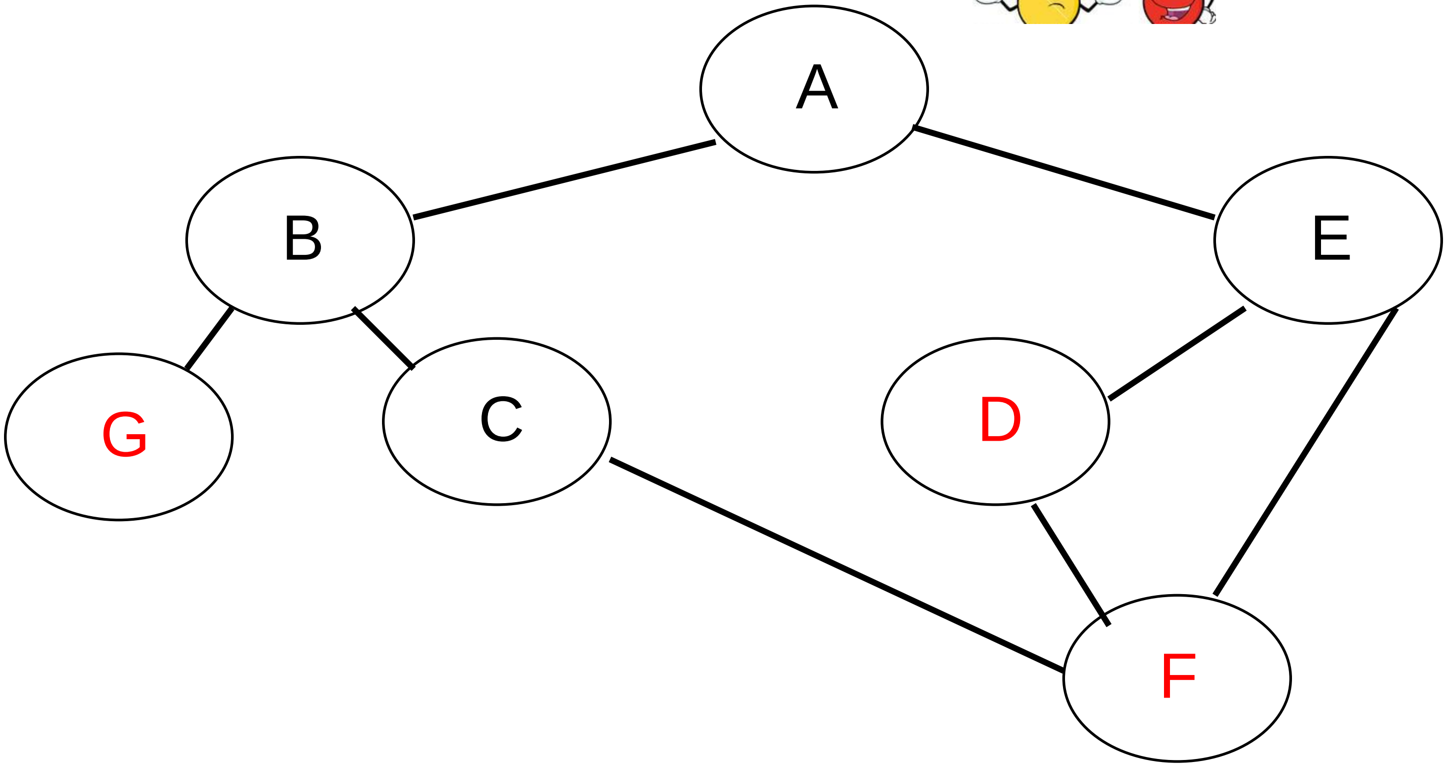
When we go to B, we put G and C in the queue

When we go to E, we put D and F in the queue

Example.

Queue: A B E C G D F F





Suppose we now want to expand C.

We put F in the queue again!

Generalizing BFS

- <u>Cycles:</u>
- We need to save auxiliary information
- Each node needs to be marked
  - Visited: No need to be put on queue
  - Not visited: Put on queue when found

<u>What about assuming only two children vertices?</u>

- Need to put all adjacent vertices in queue

The general BFS algorithm

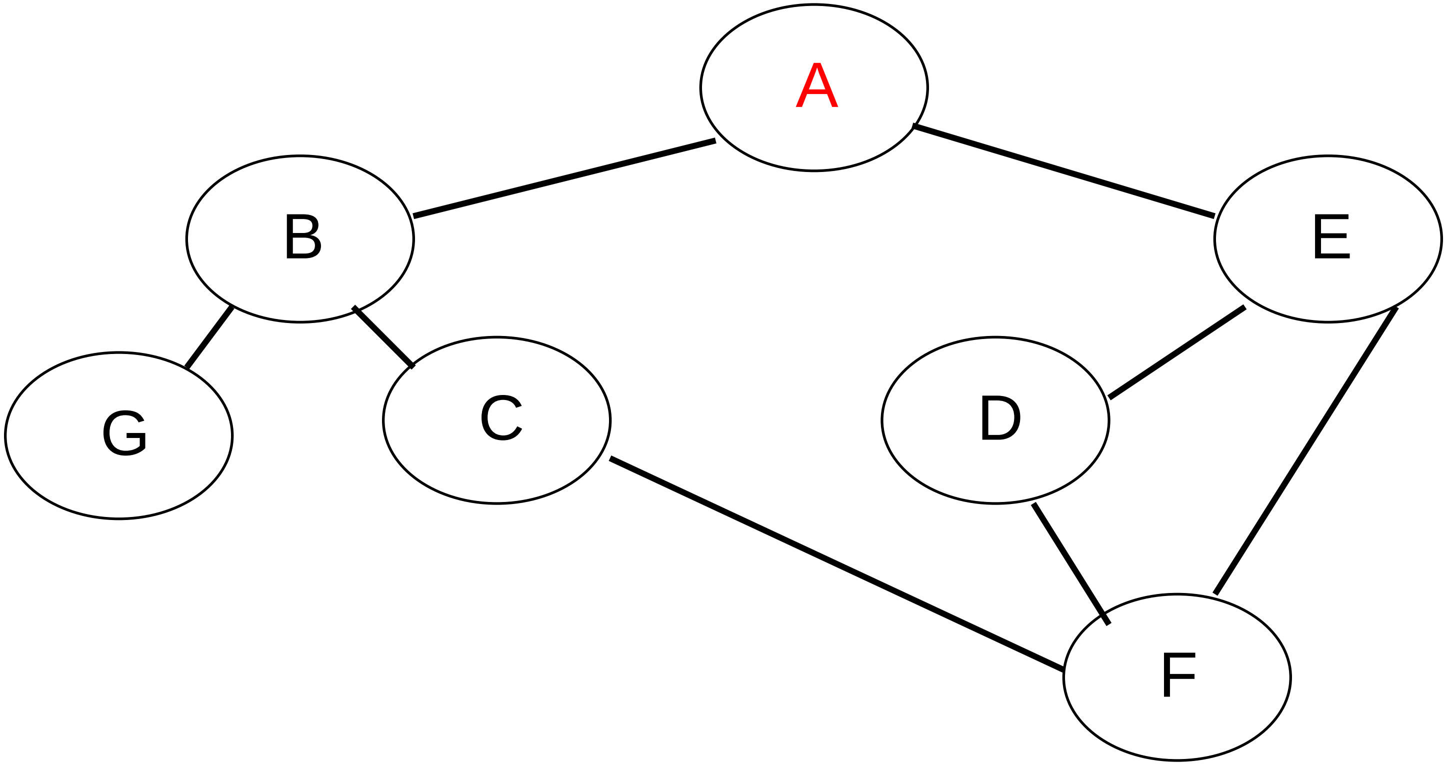
- Each vertex can be in one of three states:
  - **Unmarked and not on queue**
  - **Marked and on queue**
  - **Marked and off queue**
- The algorithm moves vertices between these states

Handling vertices

- Unmarked and not on queue:
  - Not reached yet
- Marked and on queue:
  - Known, but adjacent vertices not visited yet (possibly)
- Marked and off queue:
  - Known, all adjacent vertices on queue or done with

Example

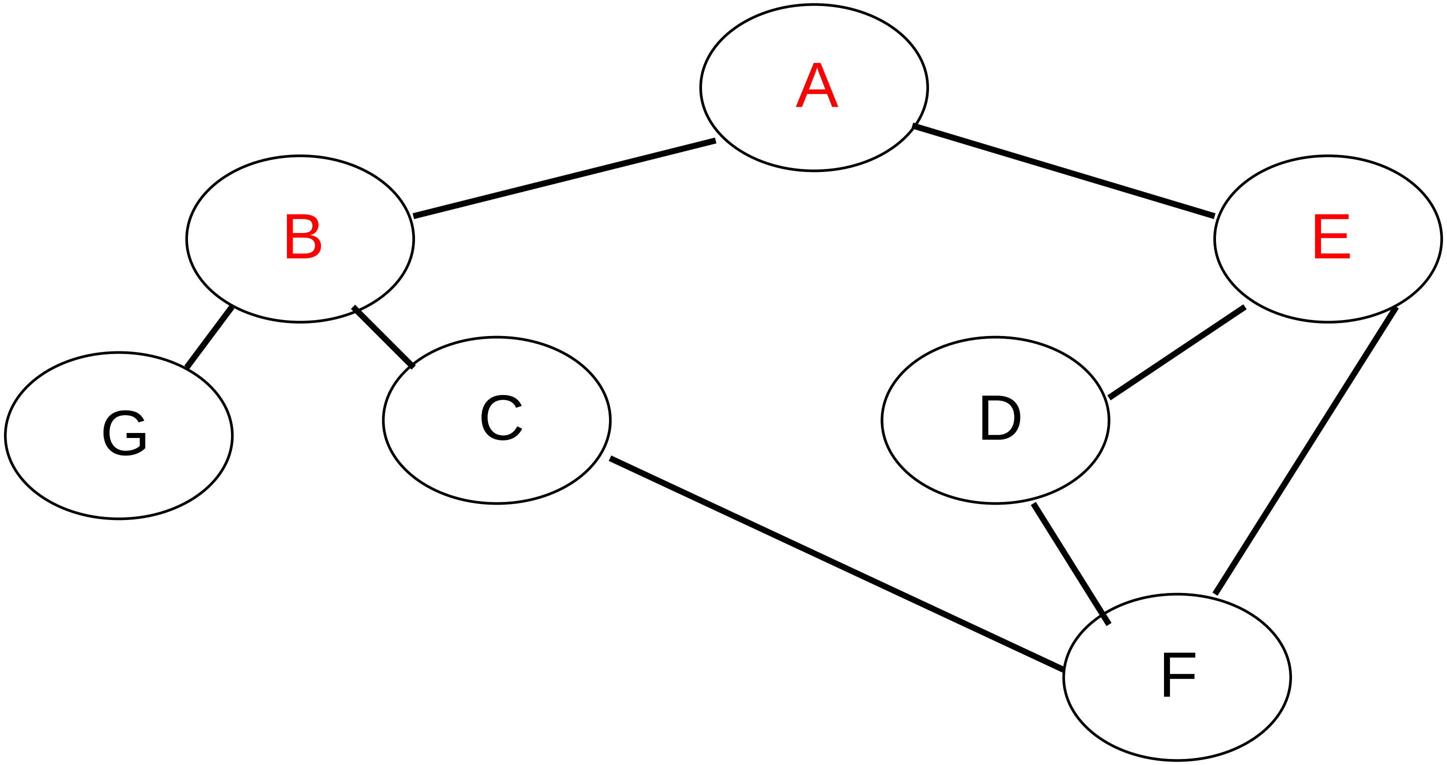
Queue: A



Start with A. Mark it.

Example

Queue: A B E

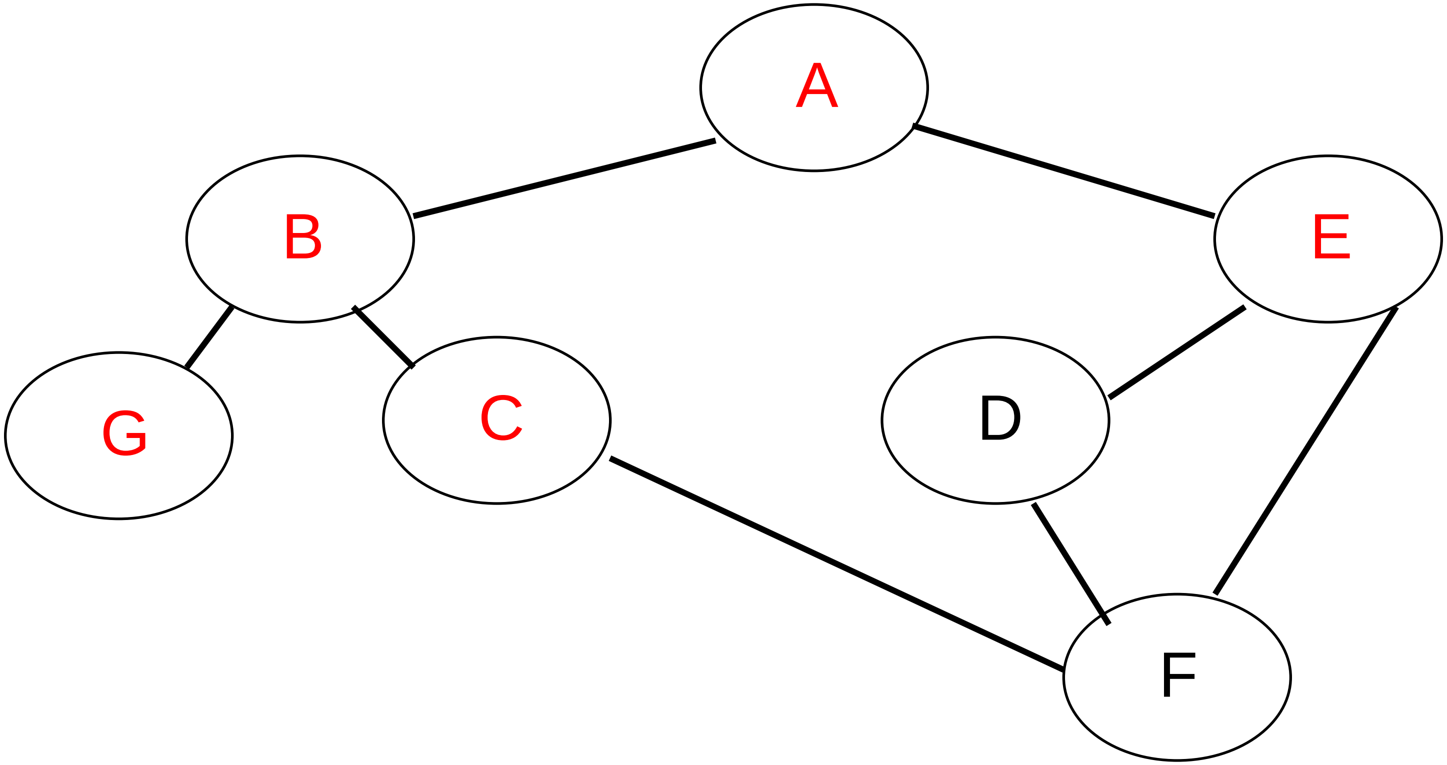


Expand A's adjacent vertices.

Mark them and put them in queue.

Example

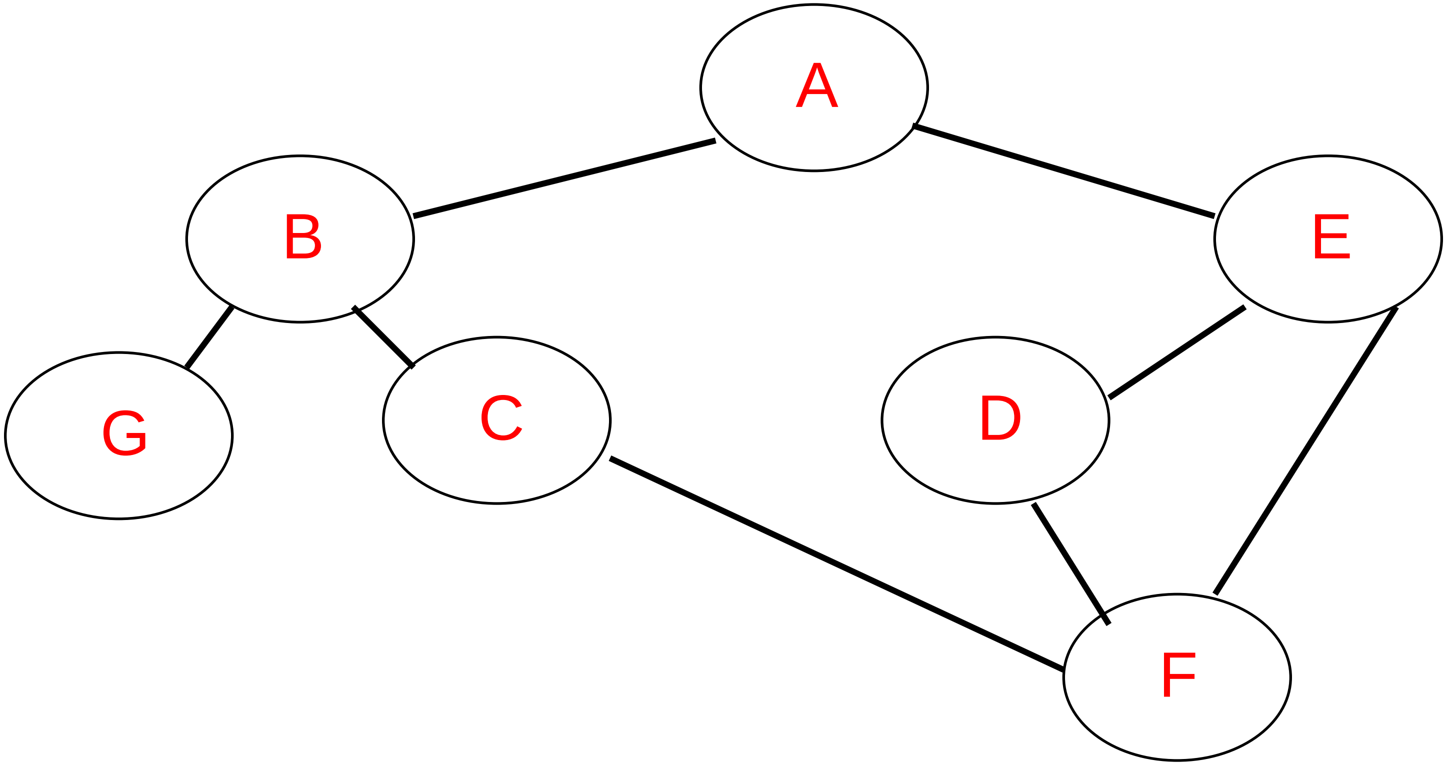
Queue: A B E C G



Now take B off queue, and queue its neighbors.

Example

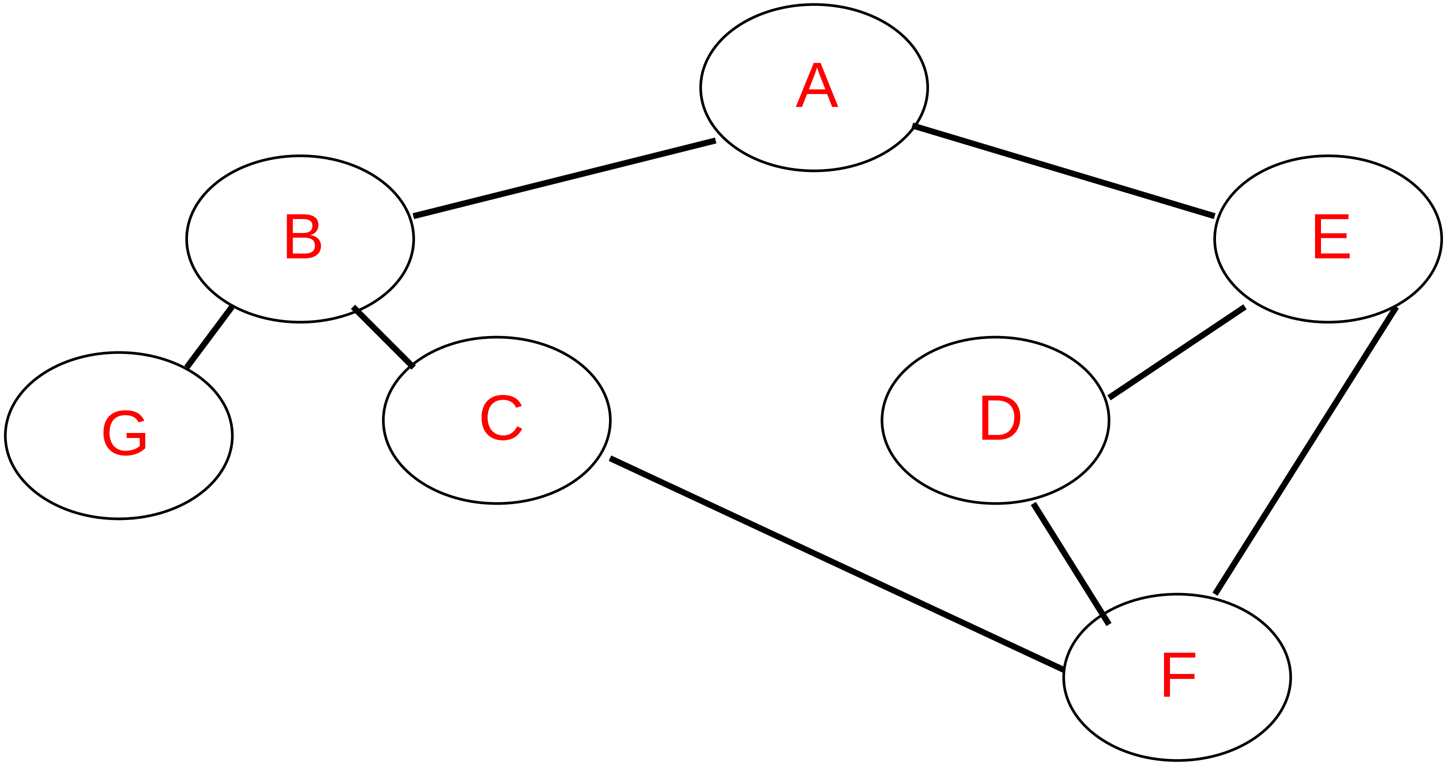
Queue: A B E C G D F



Do same with E.

Example

Queue: A B E C G D F

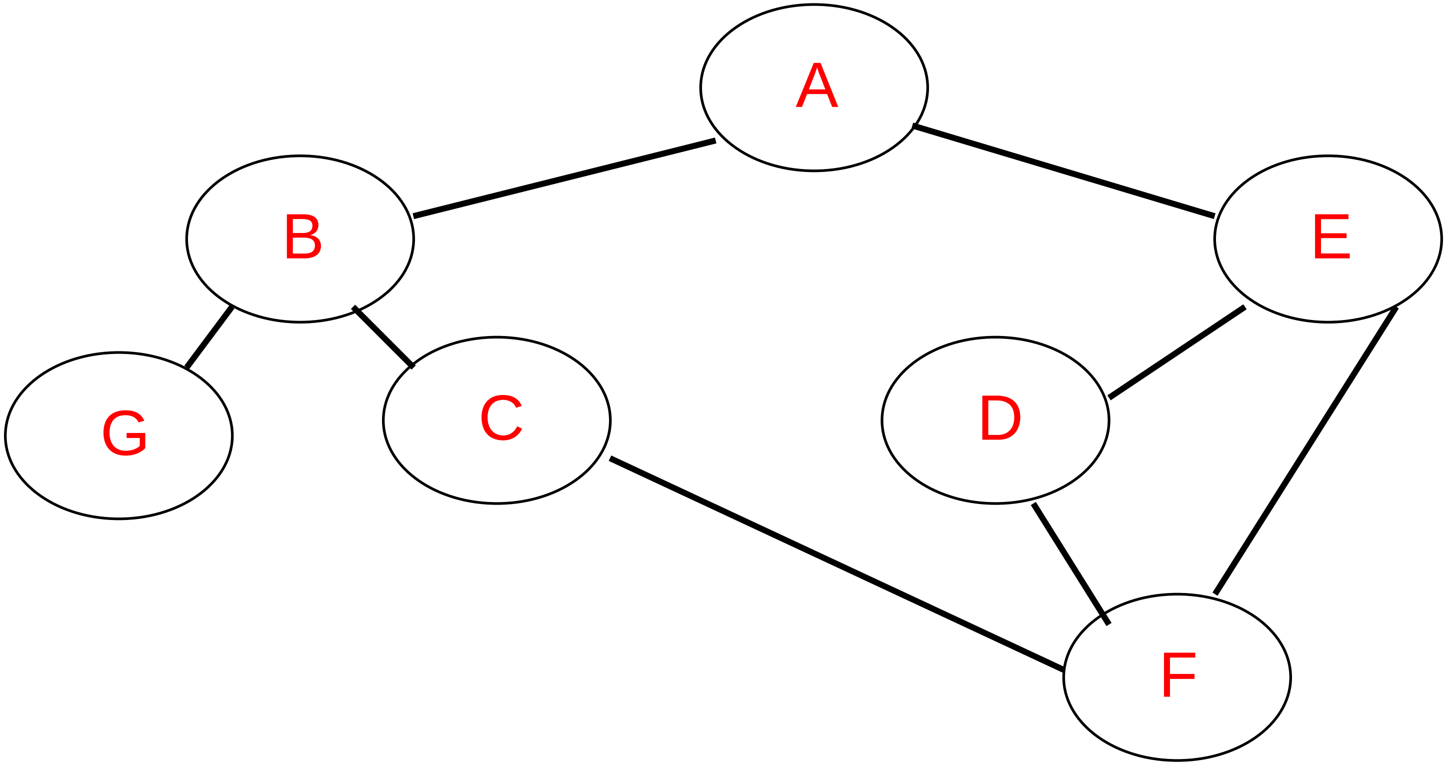


Visit C.

Its neighbor F is already marked, so not queued.

Example

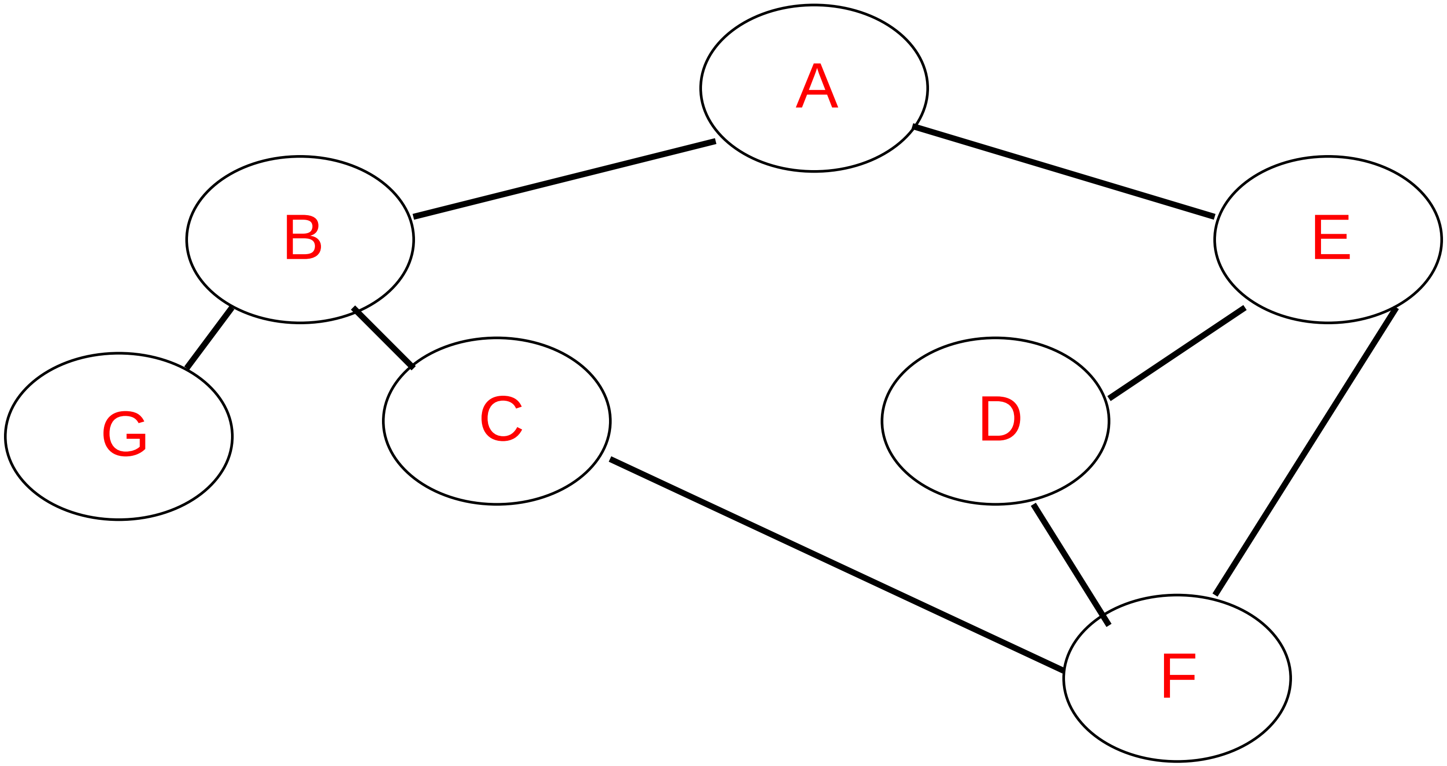
Queue: A B E C G D F



Visit G.

Example

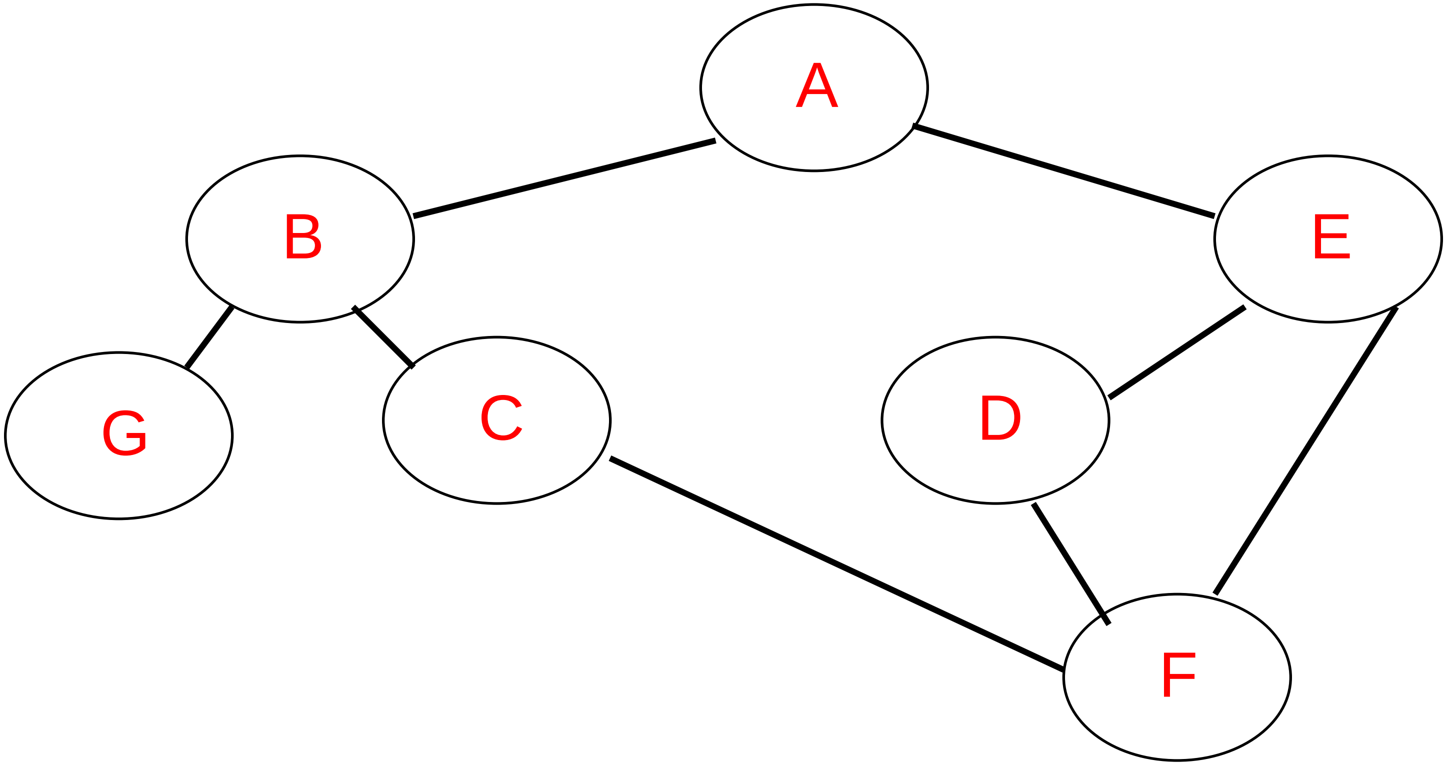
Queue: A B E C G D F



Visit D. F, E marked so not queued.

Example

Queue: A B E C G D F

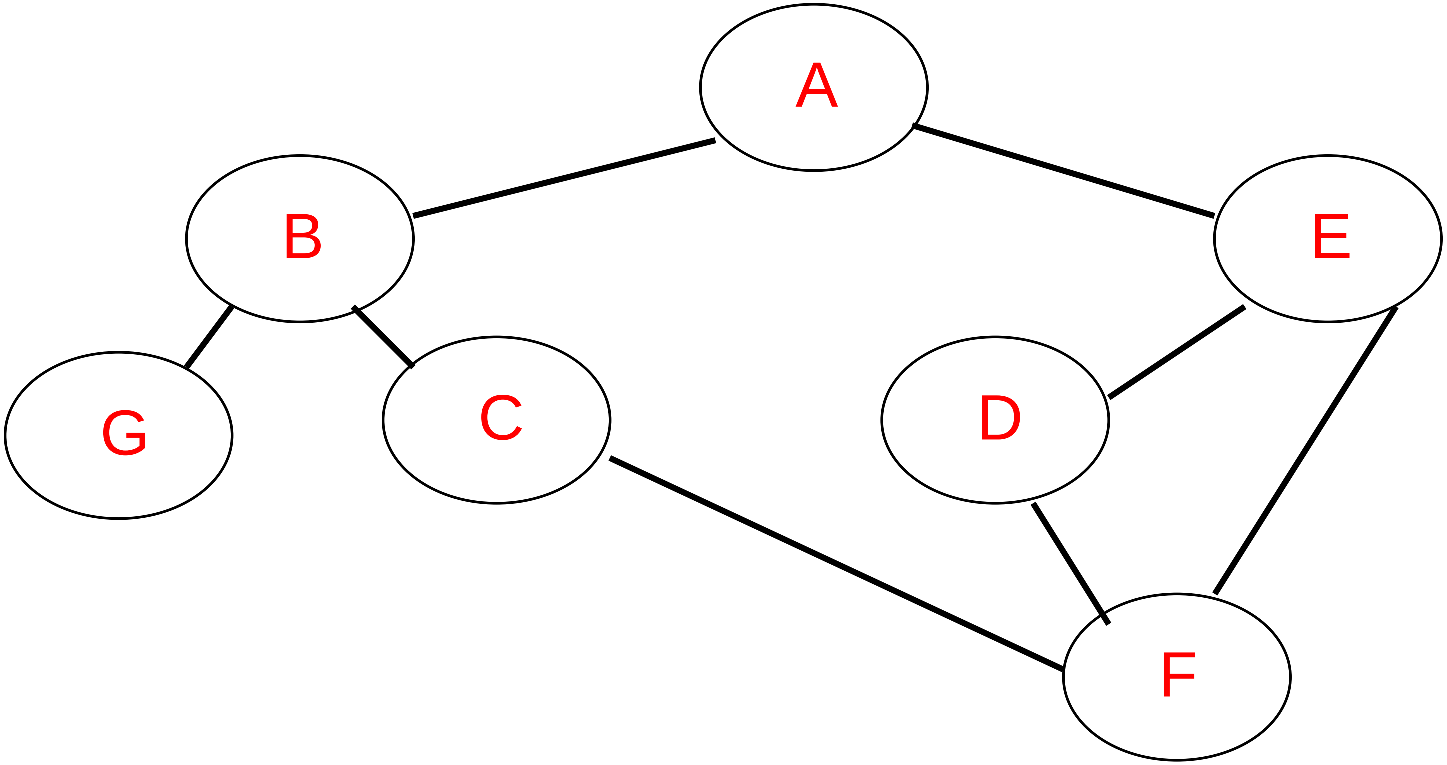


Visit F.

E, D, C marked, so not queued again.

Example

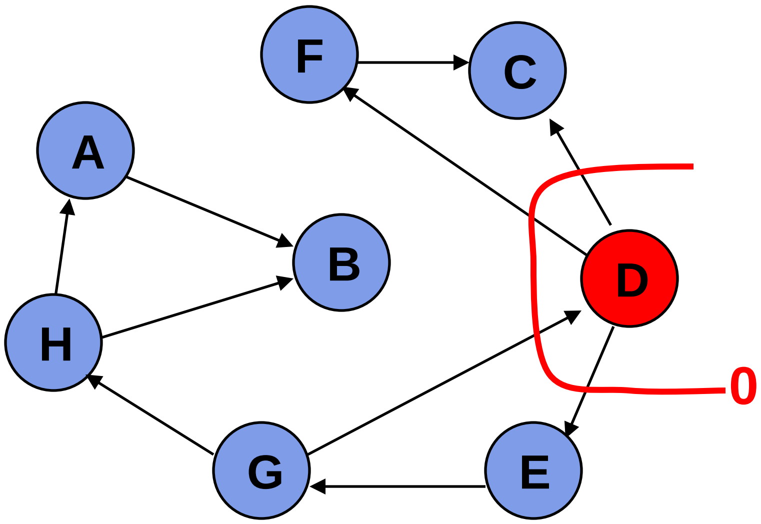
Queue: A B E C G D F



Done. We have explored the graph in order:

A B E C G D F

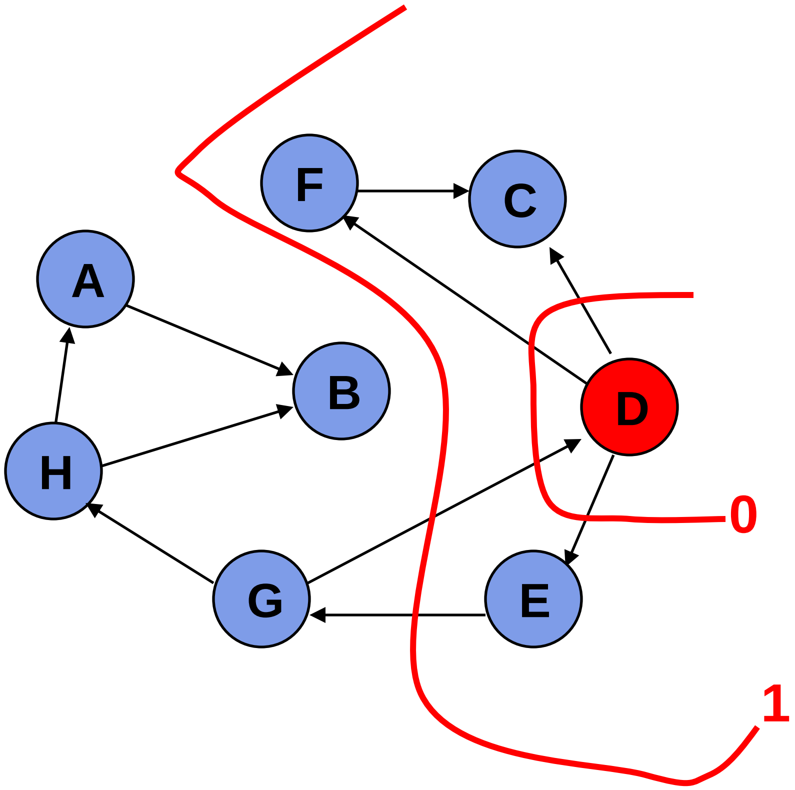
Overview



Breadth-first search starts with given node

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

Overview



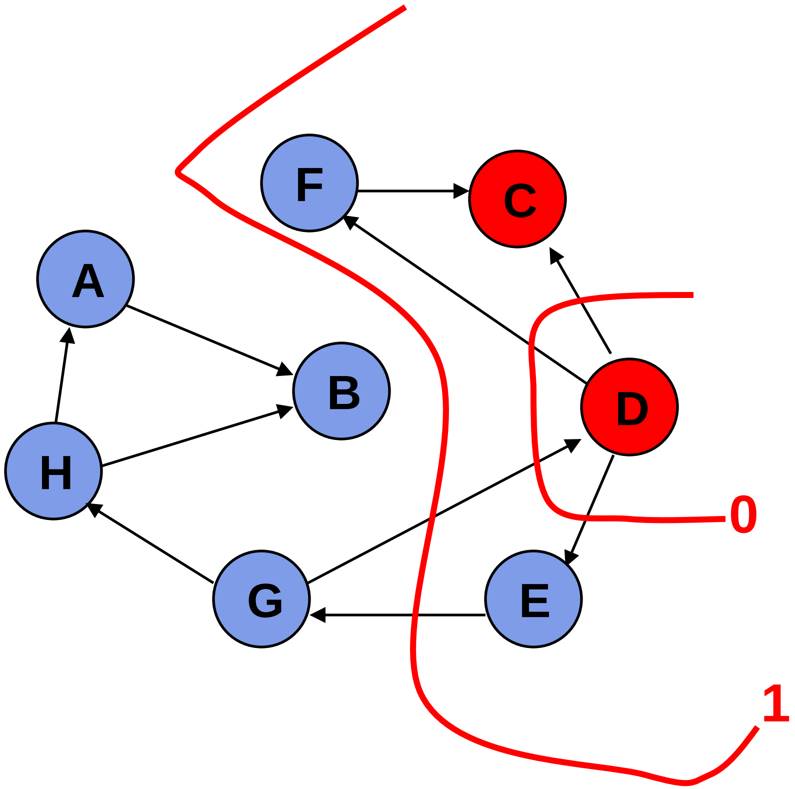
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., **alphabetical**)

Like ripples in a pond

Nodes visited: D

Overview



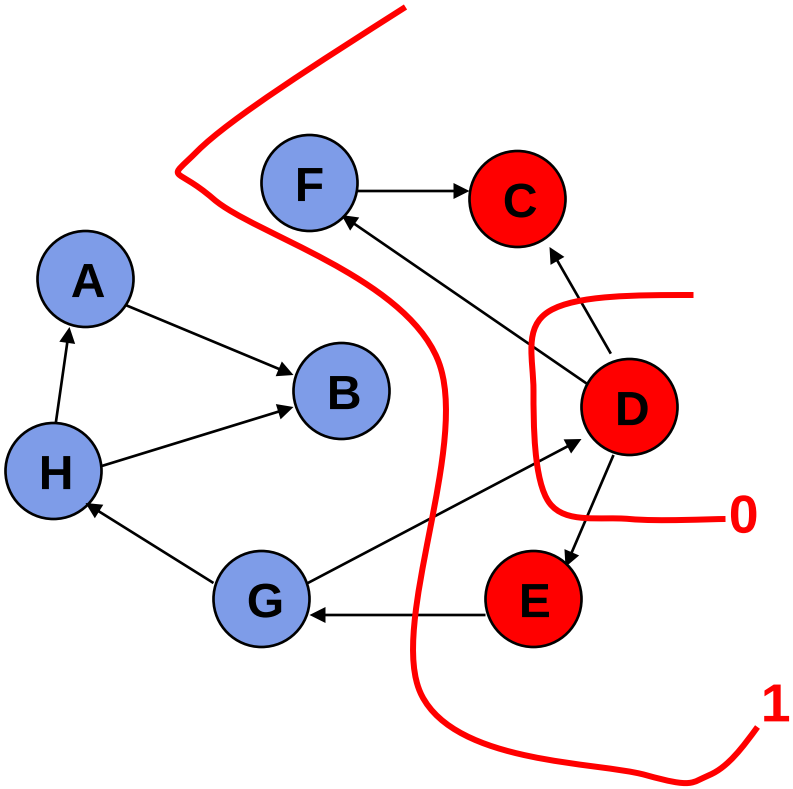
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., **alphabetical**)

Like ripples in a pond

Nodes visited: D, C

Overview



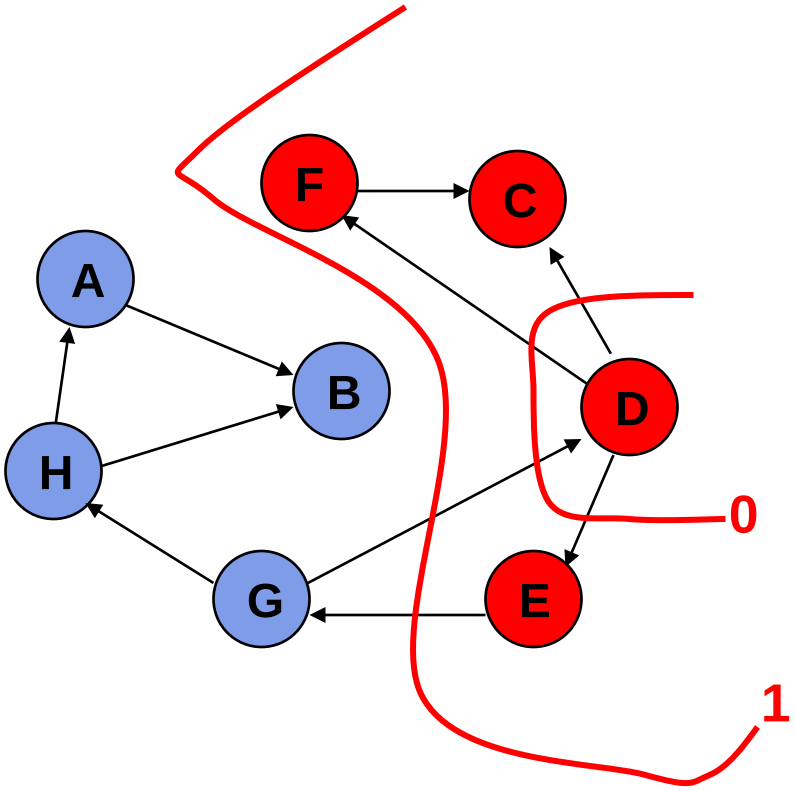
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D, C, E

Overview



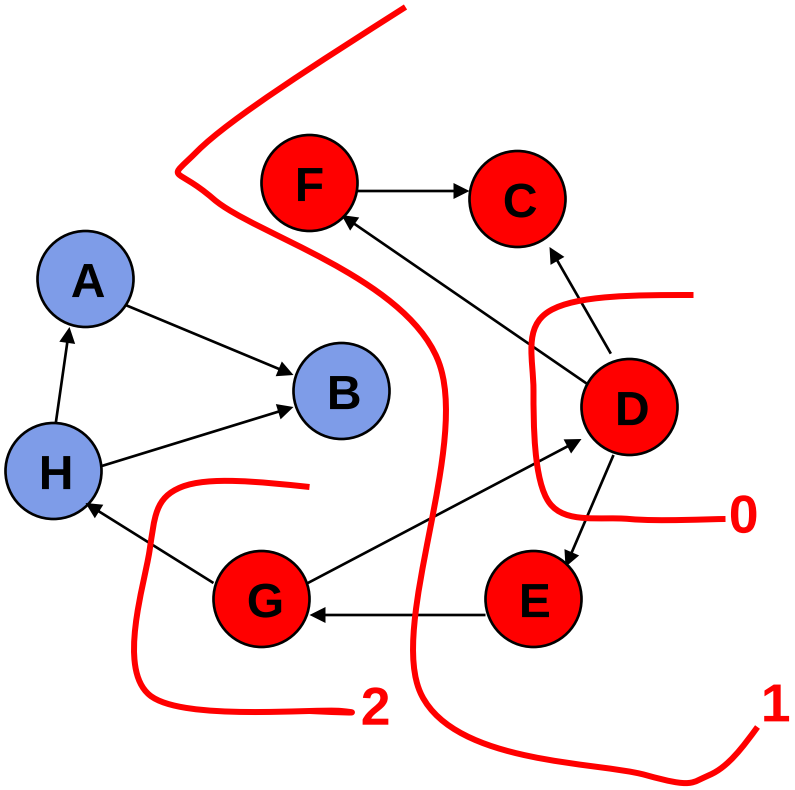
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D, C, E, F

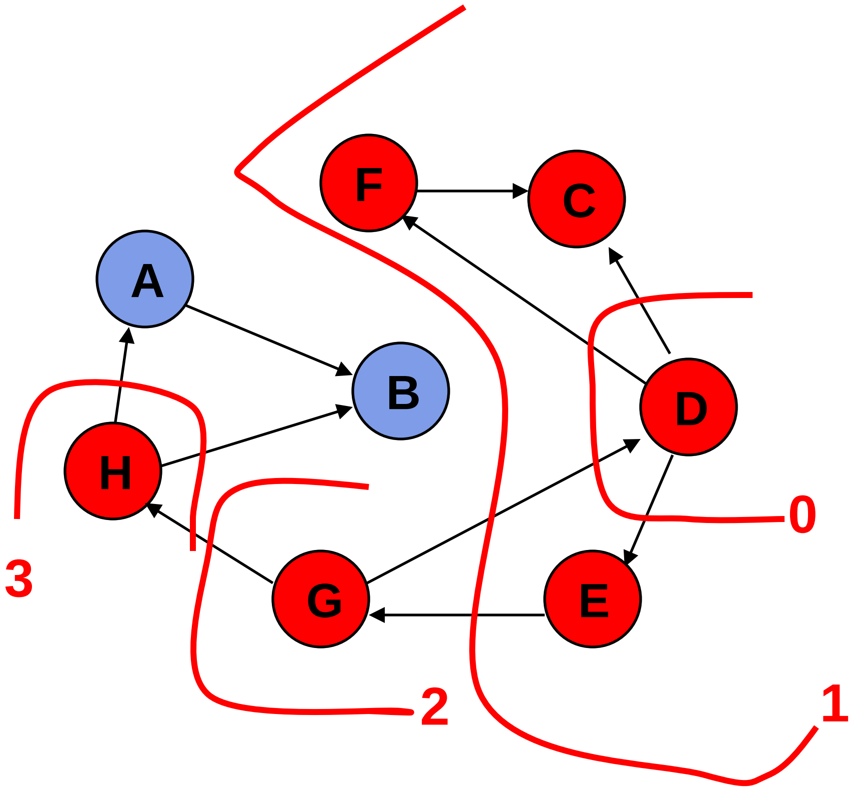
Overview



When all nodes in ripple are visited, visit nodes in next ripples

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

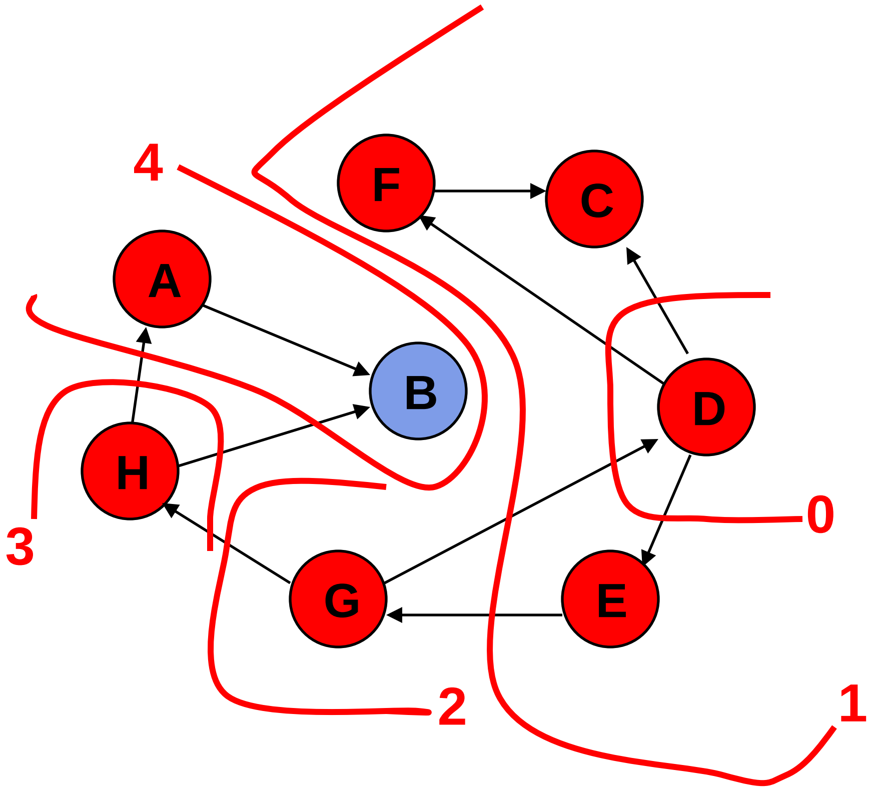
Overview



When all nodes in ripple are visited, visit nodes in next ripples

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

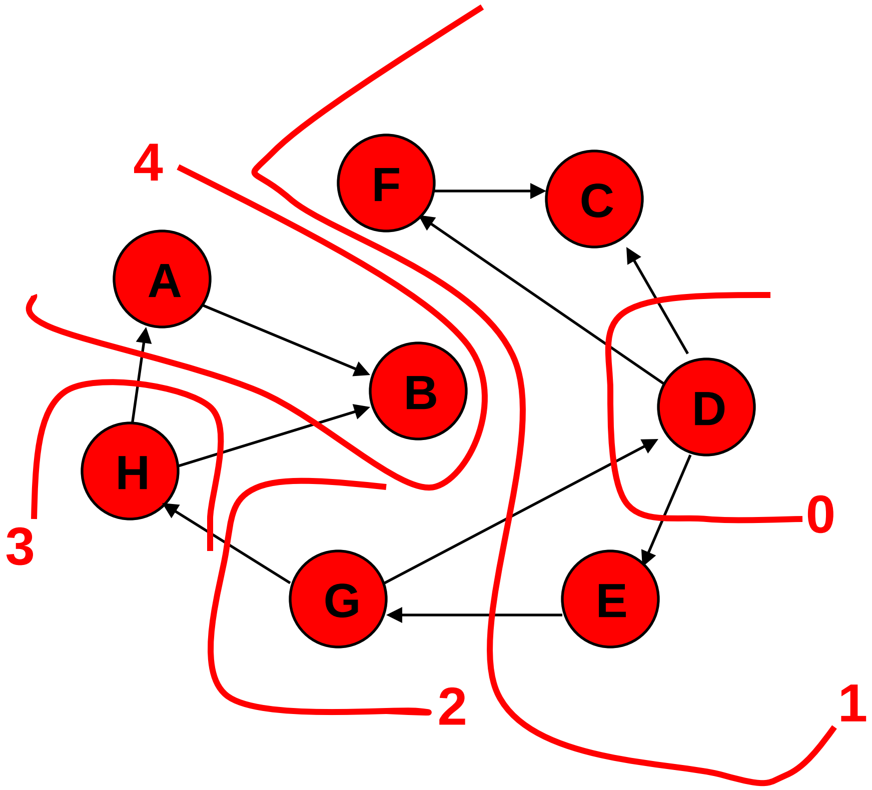
Overview



When all nodes in ripple are visited, visit nodes in next ripples

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

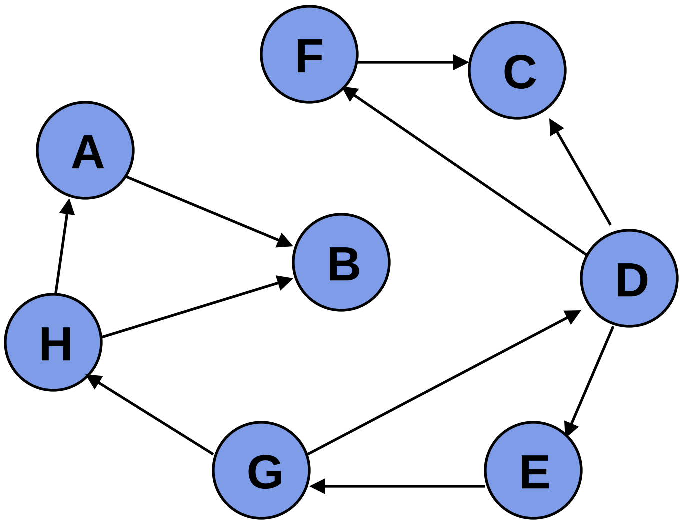
Overview



When all nodes in ripple are visited, visit nodes in next ripples

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

Walk-Through



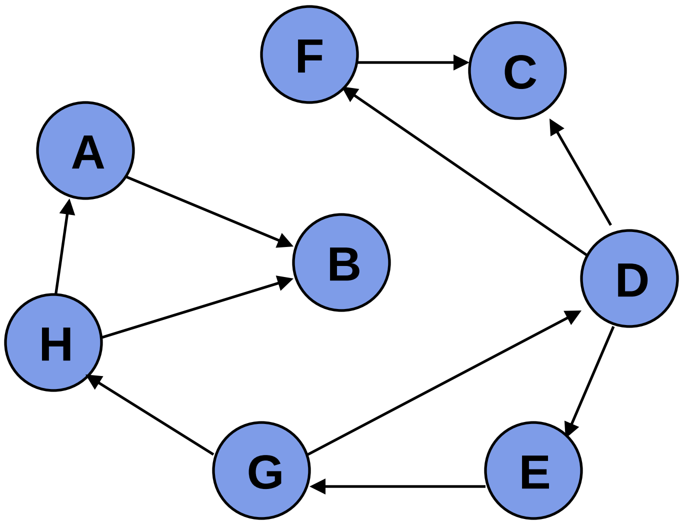
Enqueued Array

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

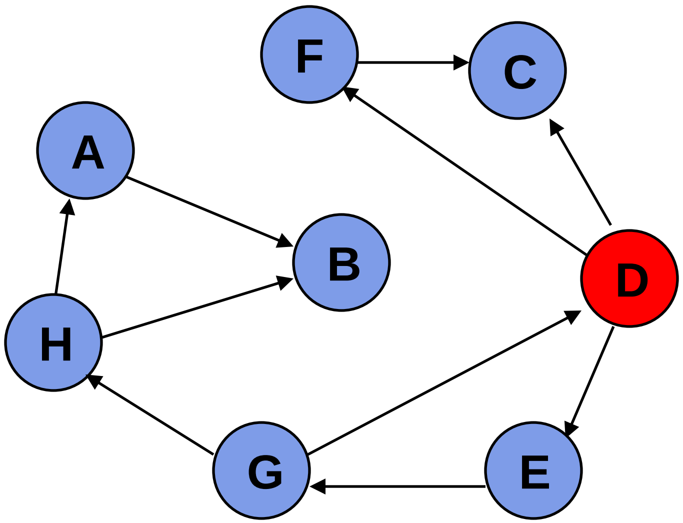
A	
B	
C	
D	√
E	
F	
G	
H	

Q → D

Nodes visited:

Enqueue D. Notice, D not yet visited.

Walk-Through



Enqueued Array

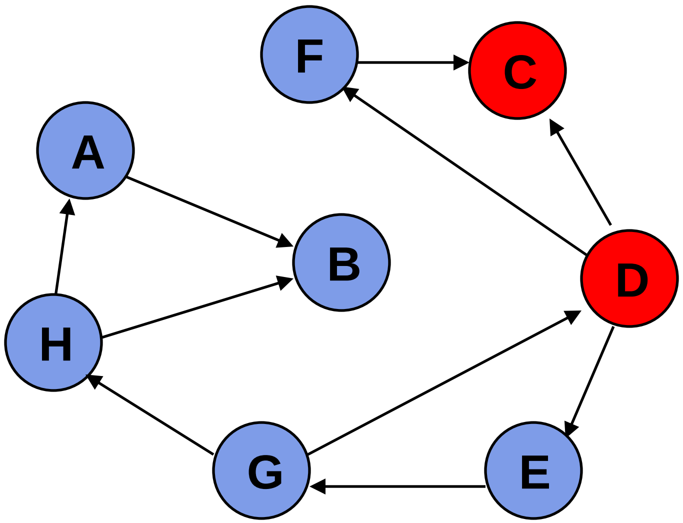
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → C → E → F

Nodes visited: D

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

Walk-Through



Enqueued Array

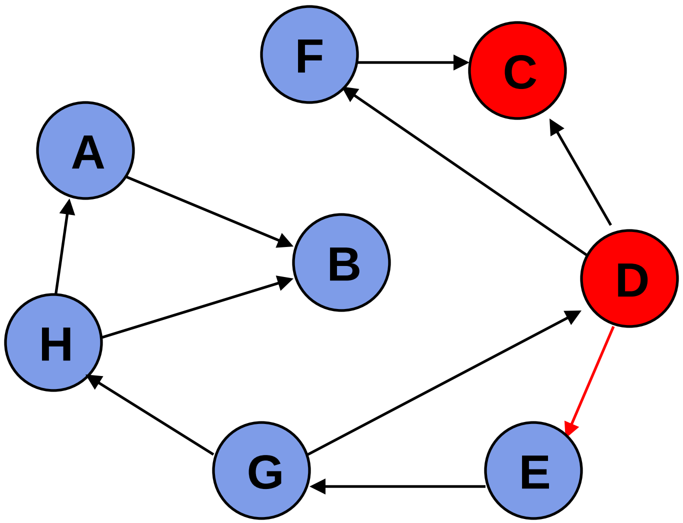
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → E → F

Nodes visited: D, C

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

Walk-Through



Enqueued Array

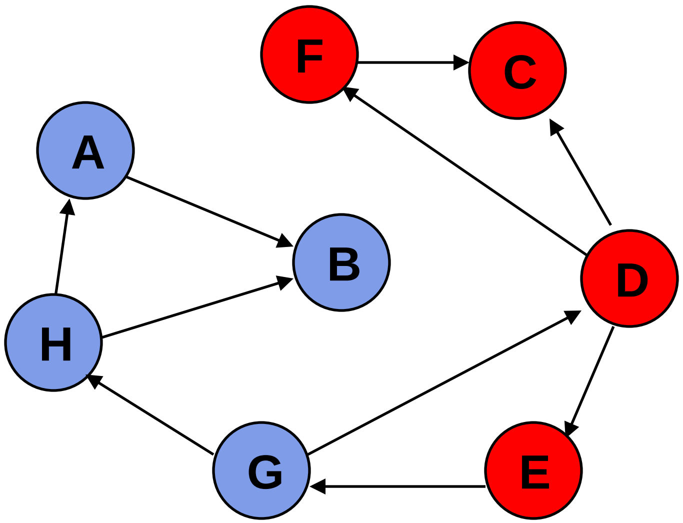
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → F → G

Nodes visited: D, C, E

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

Walk-Through



Enqueued Array

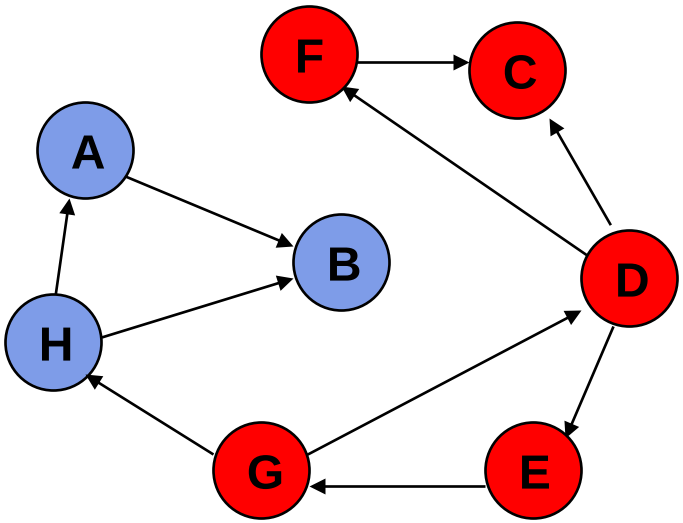
A	
B	
C	√
D	√
E	√
F	√
G	√
H	

Q → G

Nodes visited: D, C, E, F

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

Walk-Through



Enqueued Array

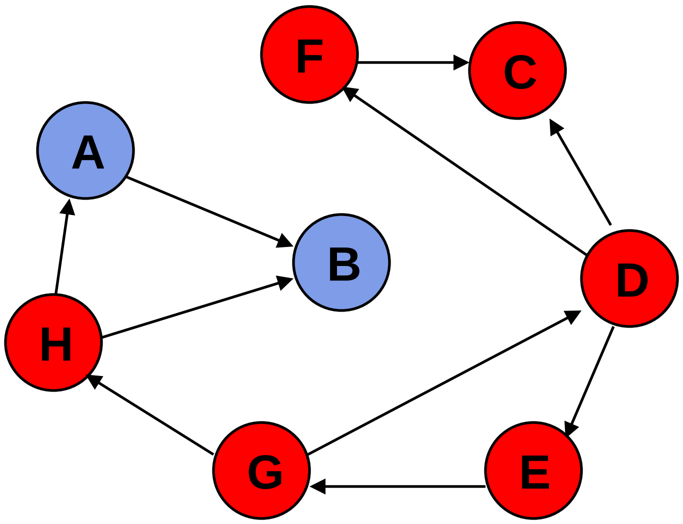
A	
B	
C	√
D	√
E	√
F	√
G	√
H	√

Q → H

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

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

Walk-Through



Enqueued Array

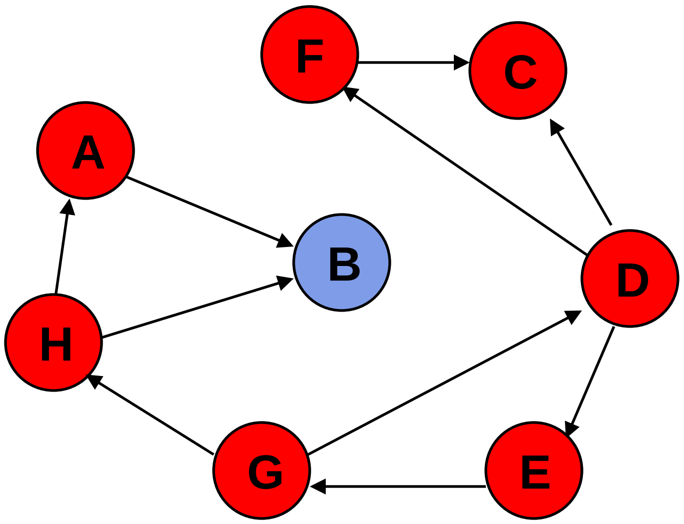
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

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

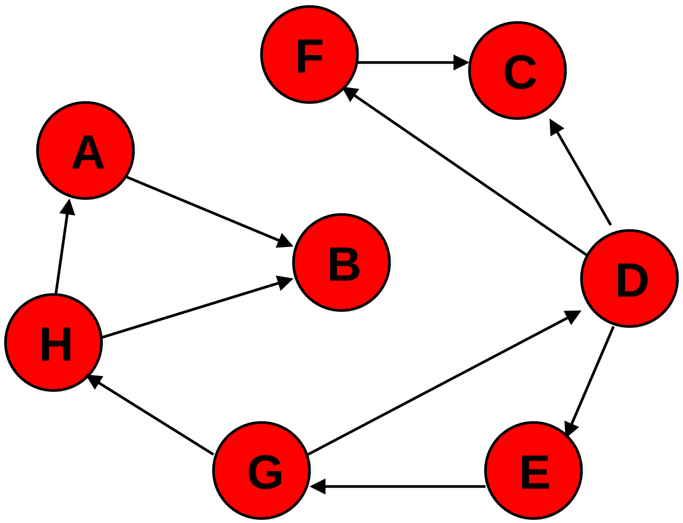
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

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

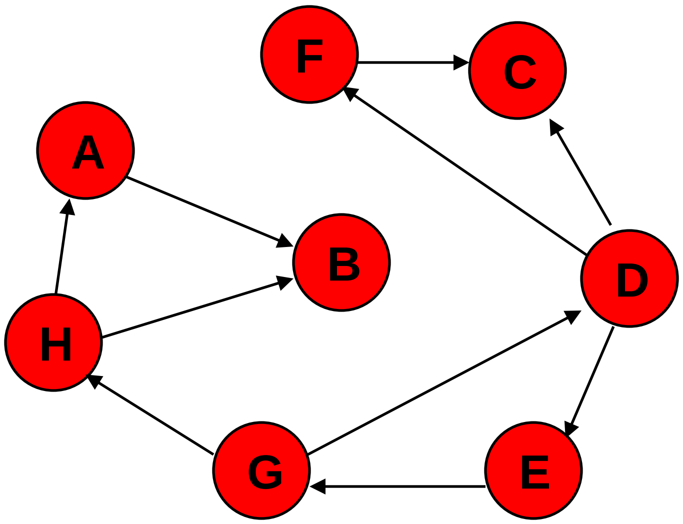
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

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.

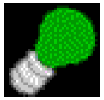
Breadth First Search Algorithm

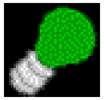
Given G = (V, E) and all v in V are marked unvisited,

Select one v in V and mark as visited;

Enqueue v in Q

```
While not is_empty(Q)
{
    x = front(Q); dequeue(Q);
        For each y in adjacent (x) if unvisited (y)
{
            Mark(y); enqueue y in Q;
                Process (x, y) ;
}
```





Thank you

???