GRAPH SEARCH ALGORITHMS

➢**BFS(Breadth First Search)**

Mashiwat Tabassum Waishy

Lecturer

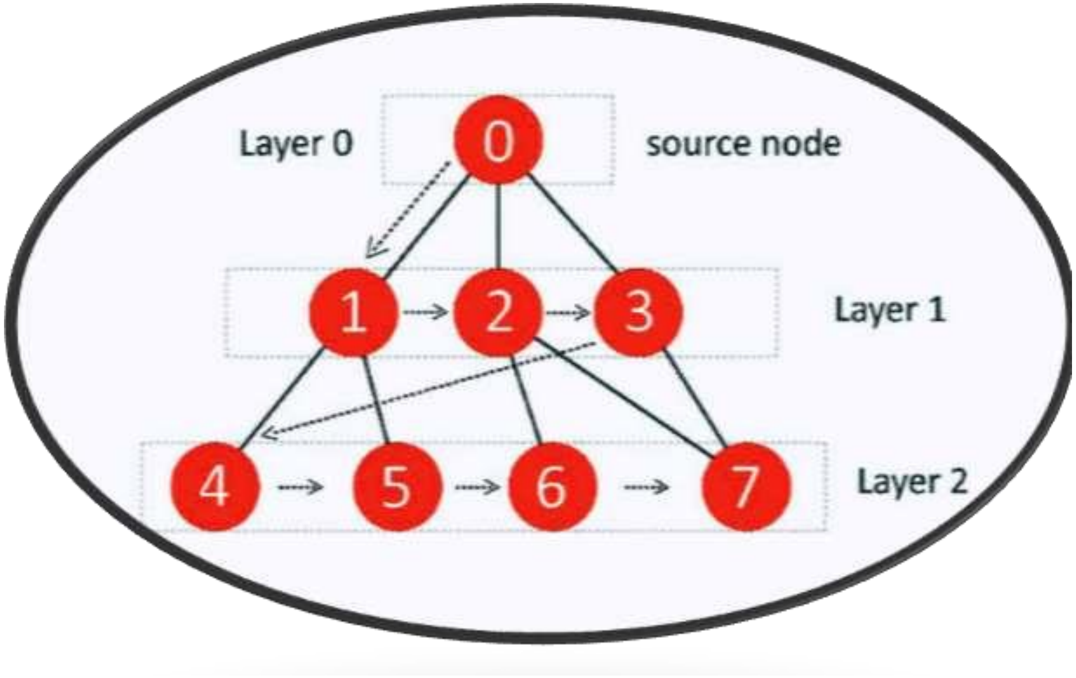
Department *of*

CSE

STAMFORD UNIVERSITY BANGLADESH

What we should know

- **Visiting a Vertex:** It means going on a particular vertex or checking a vertex's value.
- **Exploration of Vertex:** It means visiting all the adjacent vertices of a selected node.
- **Traversing**: Traversing means passing through nodes in a specific order
- **Level-Order:** It is a traversing method, where we have to visit every node on a level before going to a lower level.



Graph Search Algorithms

- ❑Systematic search of every edge and every vertex of a graph.
- ❑ Graph G = (V,E) is either directed or undirected.
- ❑ Applications
  - ❑ Compilers
  - ❑ Networks
    - ➢ Routing, Searching, Clustering, etc.

Graph Traversal

- ❑ Breadth First Search (BFS)
  - ➢Start several paths at a time, and advance in each one step at a time

- ❑ Depth First Search (DFS)
  - ➢Once a possible path is found, continue the search until the end of the path

Breadth-First Search

- Breadth-first search starts at a given vertex *s*, which is at level 0.
- In the first stage, we visit all the vertices that are at the distance of one edge away(adjacent vertices). When we visit there, we paint as "**visited**," the vertices adjacent to the start vertex *s* - these vertices are placed into level 1.
- In the second stage, we visit all the new vertices we can reach at the distance of two edges away from the source vertex s. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on.
- The BFS traversal terminates when every vertex has been visited.

Breadth-First Search

The algorithm maintains a queue Q to manage the set of vertices
and starts with s, the source vertex

Initially, all vertices except s are colored white, and s is gray.

BFS algorithm maintains the following information for each vertex u:

- color[u] (white, gray, or black) : indicates status

white = not discovered yet
gray = discovered, but not finished
black = finished

- **d[u]** : distance from s to u
- **π[u]** : predecessor of u in BF tree
- **Q :** FIFO(First In First Out) Queue

Each vertex is assigned a color.

In general, a vertex is **white** before we start processing it, it is **gray** during the period the vertex is being processed, and it is **black** after the processing of the vertex is completed.

BFS Algorithm

Inputs: Inputs are a graph(directed or undirected) G =(V, E) and a source vertex s, where s is an element of V. The adjacency list representation of a graph is used in this analysis.

Outputs: The outputs are a predecessor graph, which represents the paths travelled in the BFS traversal, and a collection of distances, which represent the distance of each of the vertices from the source vertex.

Example:

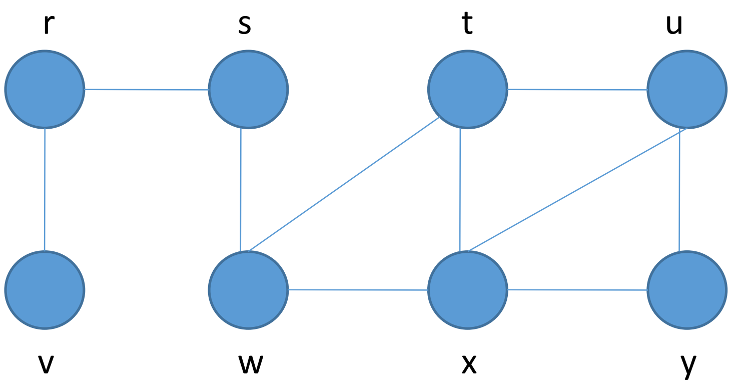


Figure: Graph G

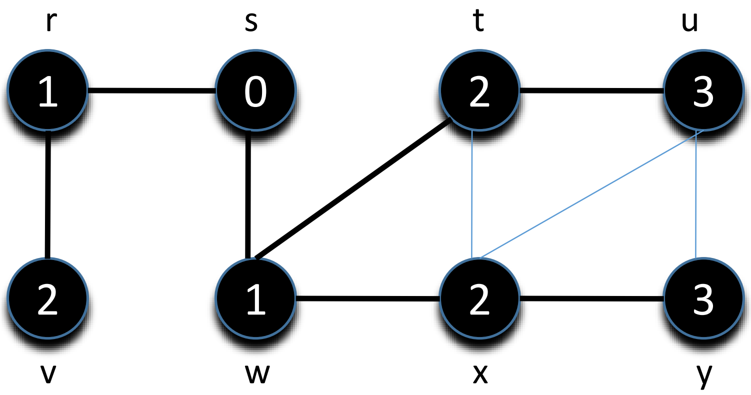
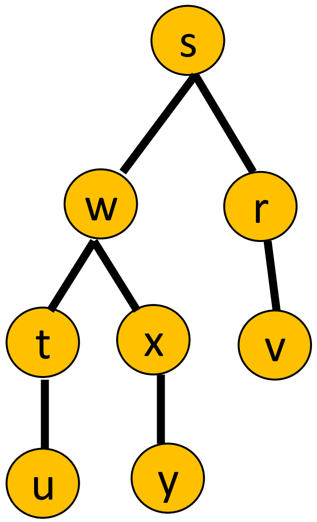
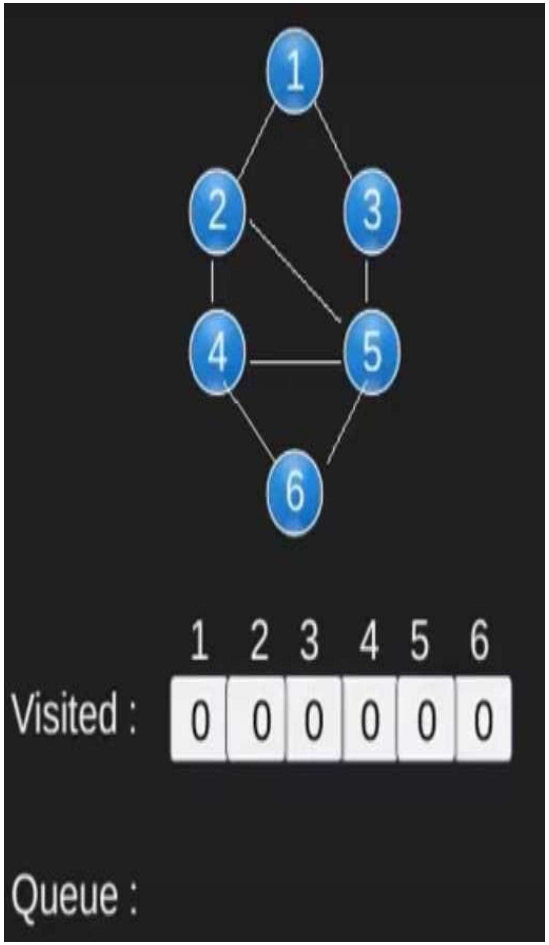


Figure: After BFS Algorithm on G, source vertex s

BFS Tree



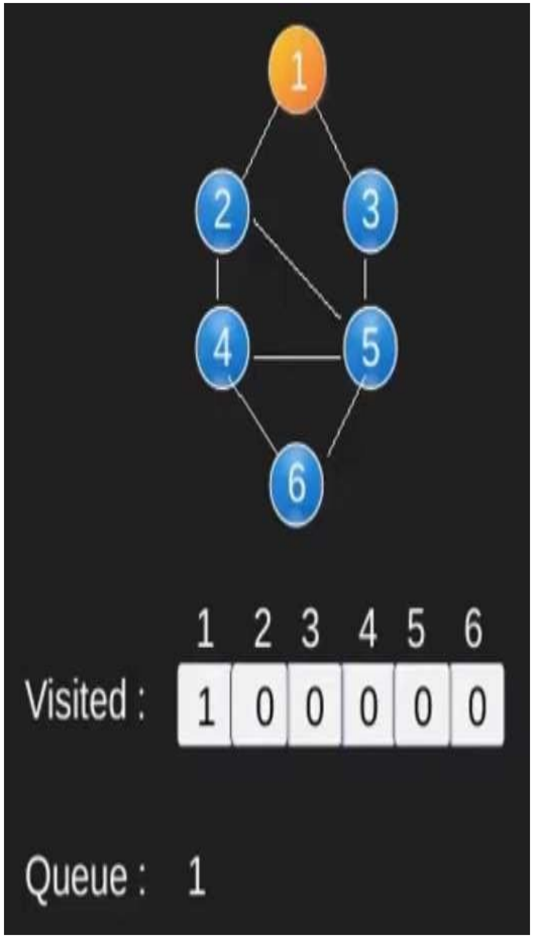
BFS Illustration



	1	2	3	4	5	6
Visited :	0	0	0	0	0	0

Queue :

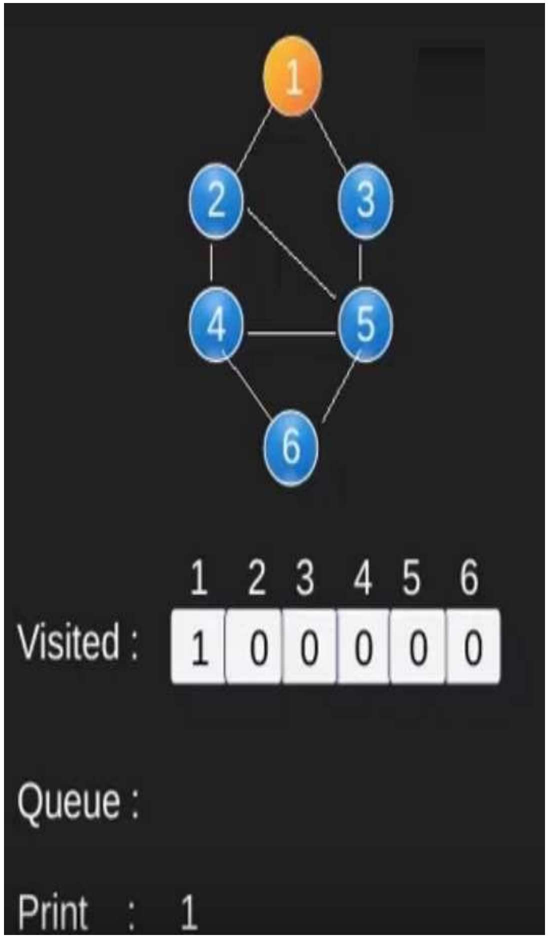
Fig: 01



	1	2	3	4	5	6
Visited :	1	0	0	0	0	0

Queue : 1

Fig: 02

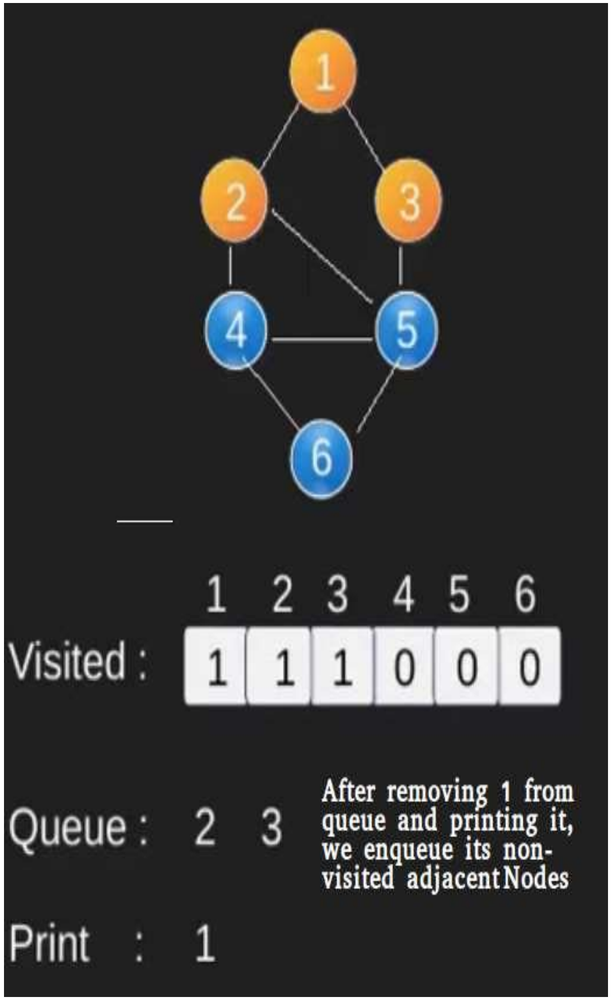


	1	2	3	4	5	6
Visited :	1	0	0	0	0	0

Queue :

Print : 1

Fig: 03



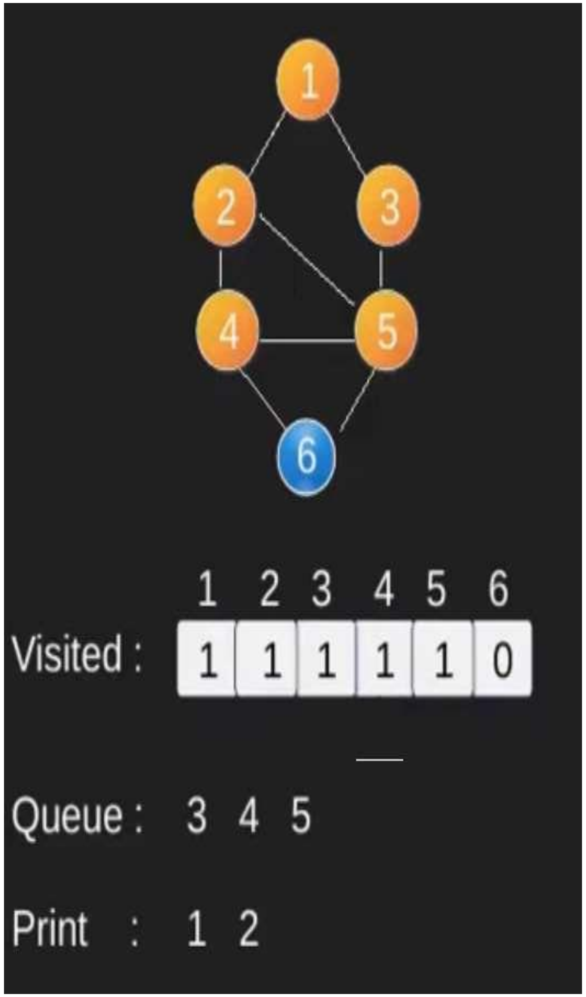
	1	2	3	4	5	6
Visited :	1	1	1	0	0	0

Queue : 2 3

After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes

Print : 1

Fig: 04

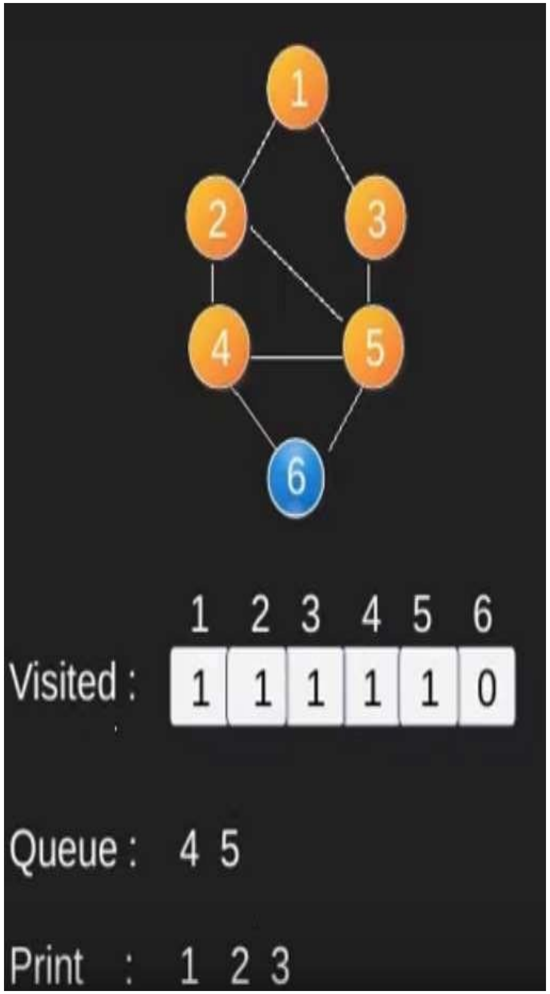


	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 3 4 5

Print : 1 2

Fig: 05

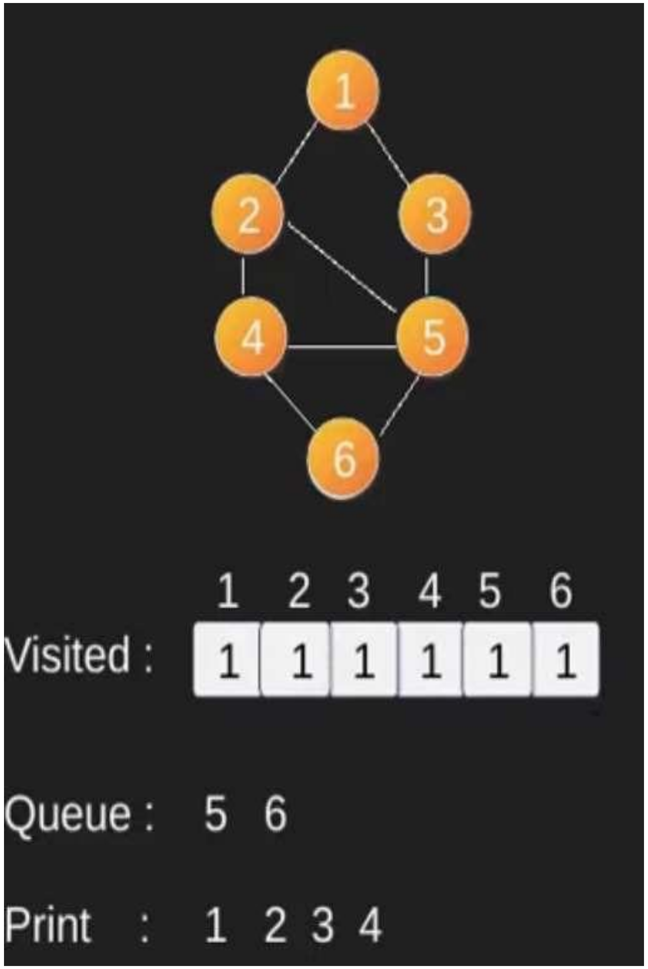


	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 4 5

Print : 1 2 3

Fig: 06

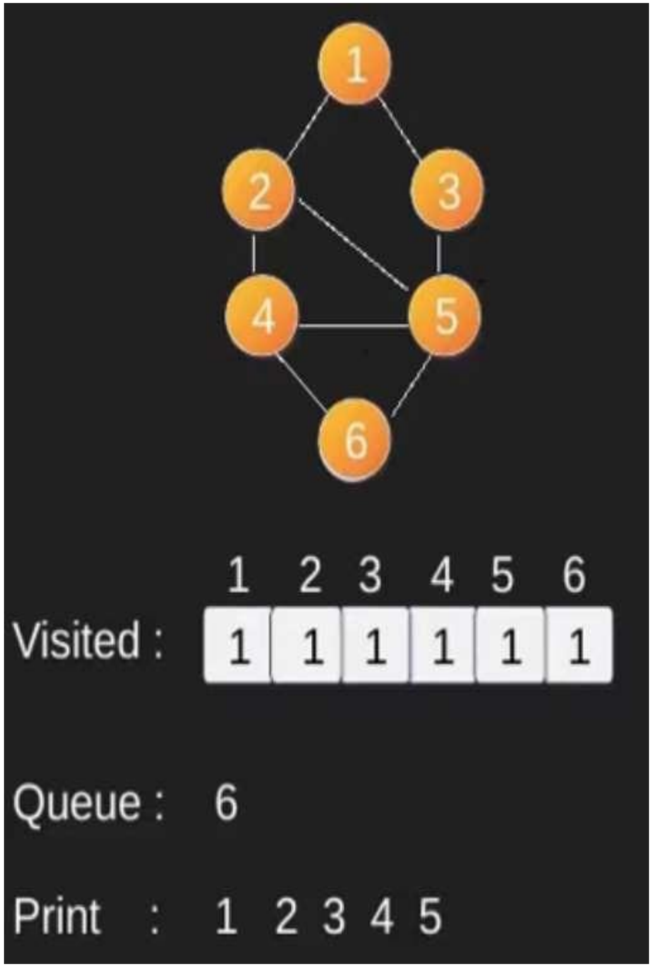


	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

Queue : 5 6

Print : 1 2 3 4

Fig: 07

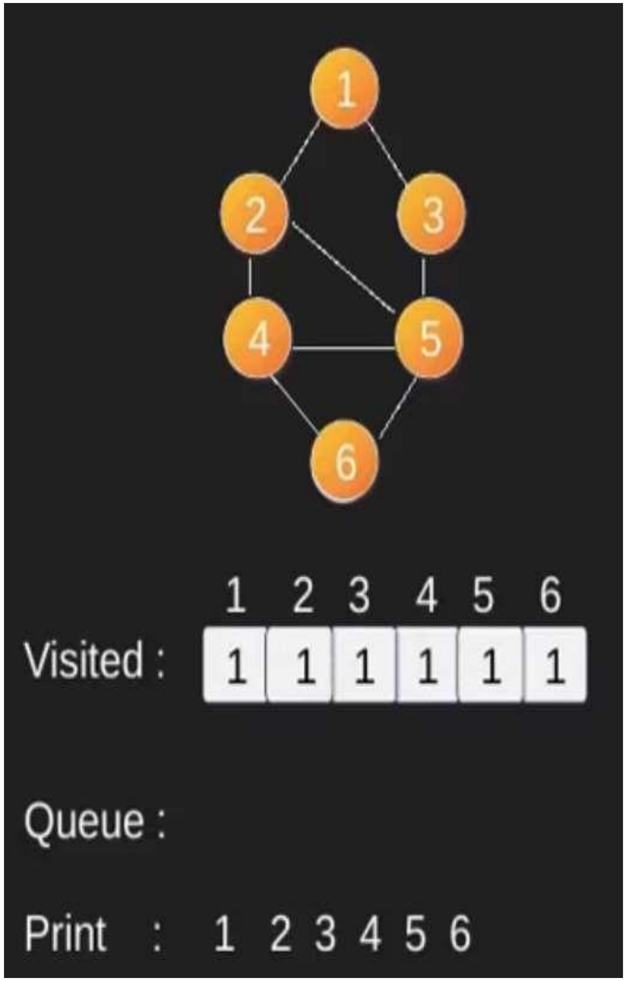


	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

Queue : 6

Print : 1 2 3 4 5

Fig: 08



	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

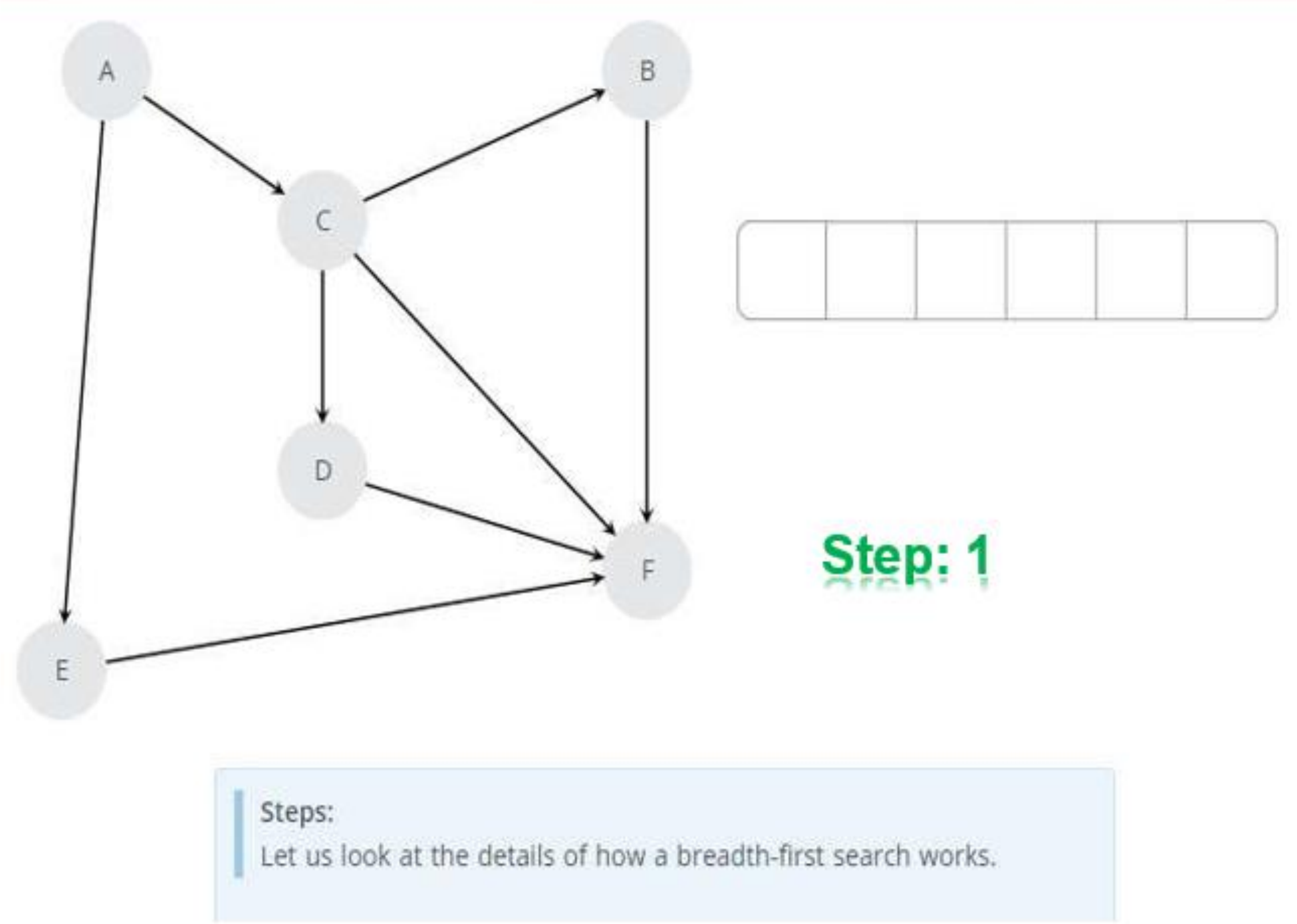
Queue :

Print : 1 2 3 4 5 6

Fig: 09

- BFS is just like Level Order & it follow 3 simple rules
- **Rule 1 :** Visit the adjacent unvisited vertex. Mark it as visited. Insert it in a queue & Display it.
- **Rule 2 :** If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3 :** Repeat Rule 1 and Rule 2 until the queue is empty.

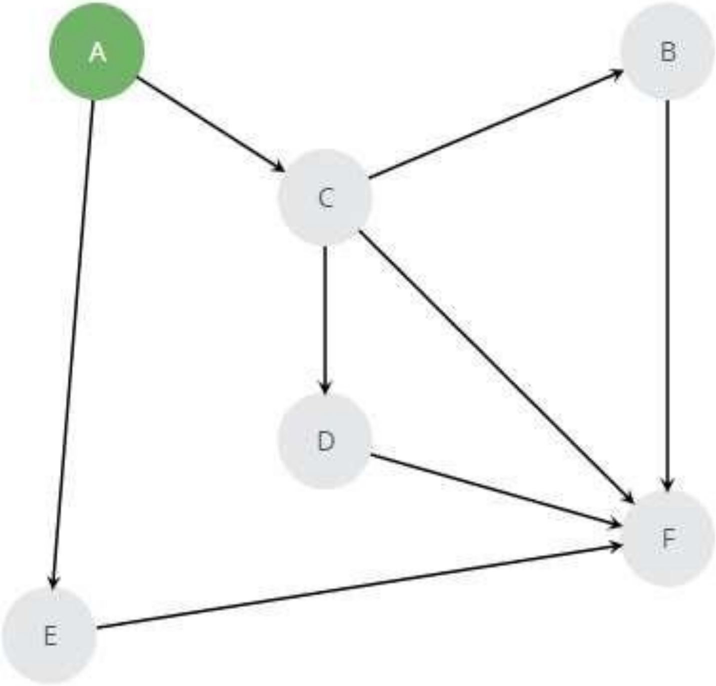
Breadth First Search Visualization



Step: 1

Steps:

Let us look at the details of how a breadth-first search works.

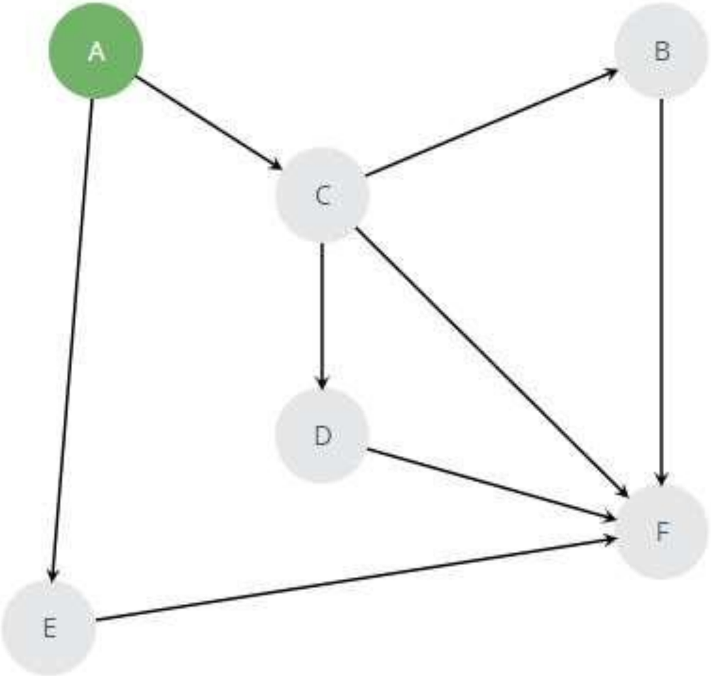


A					

Step: 2

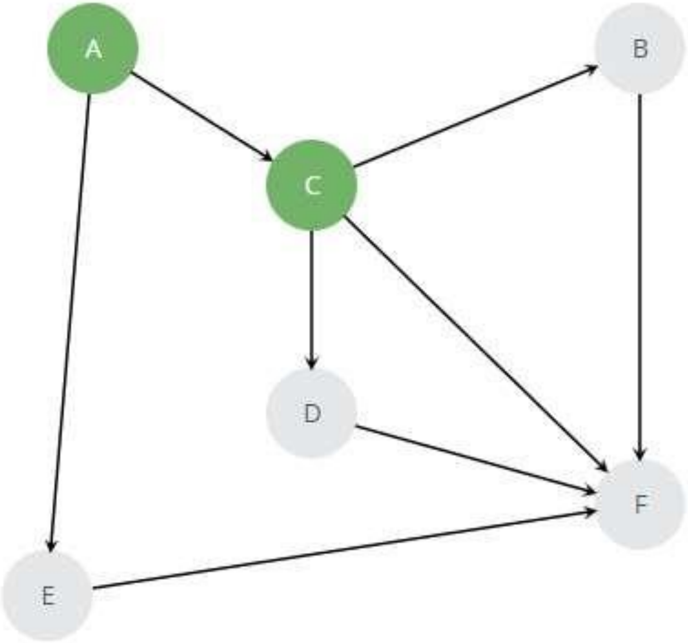
Steps:

Mark and enqueue A



Step: 3

Steps:
Dequeue A

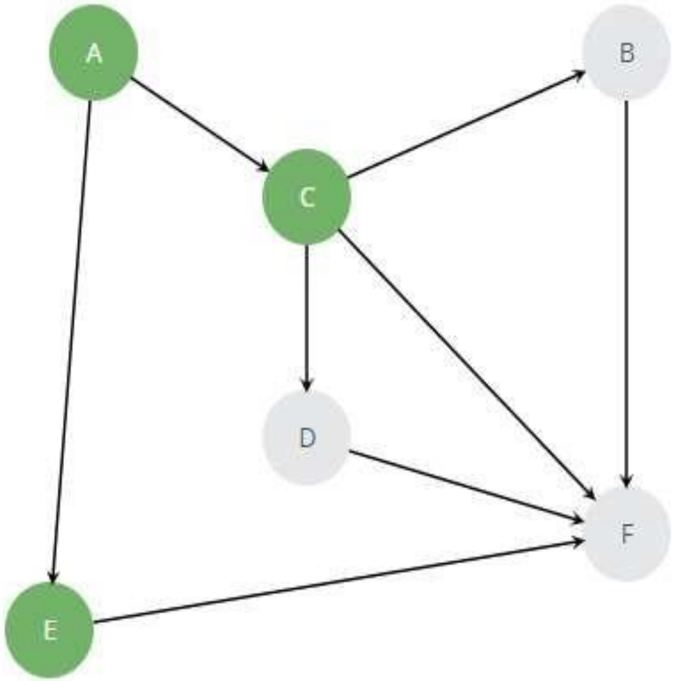


	C				

Step: 4

Steps:

Mark and enqueue C

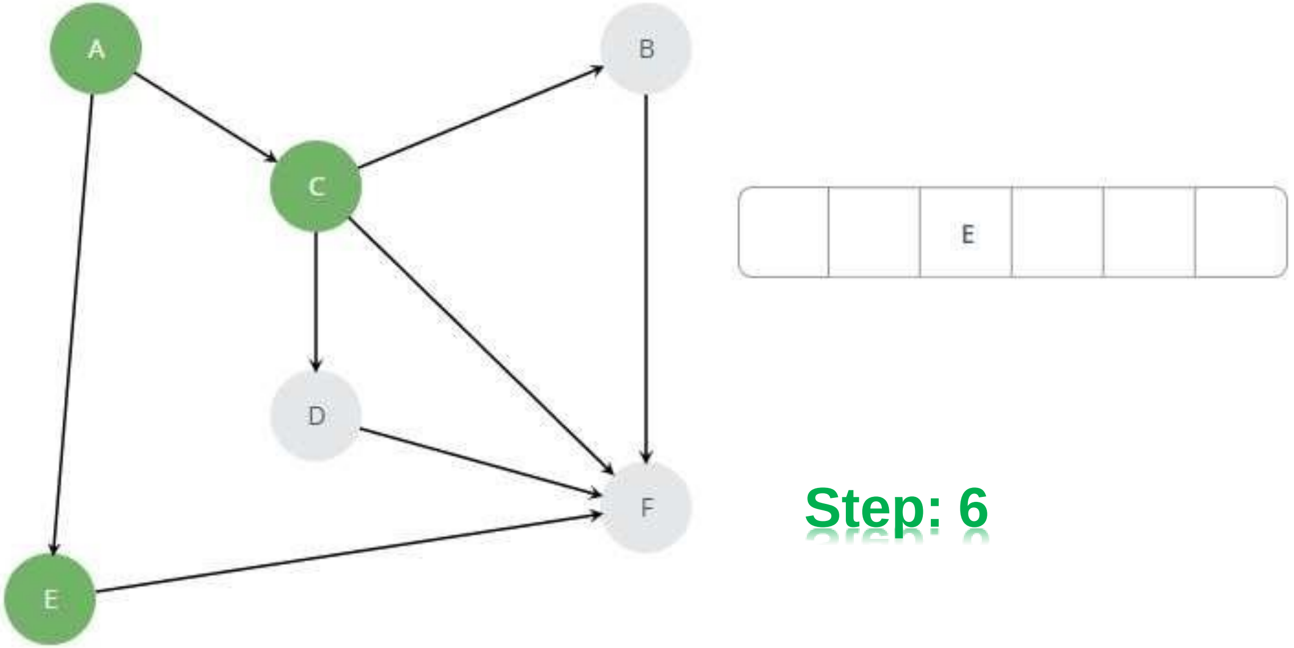


	C	E			

Step: 5

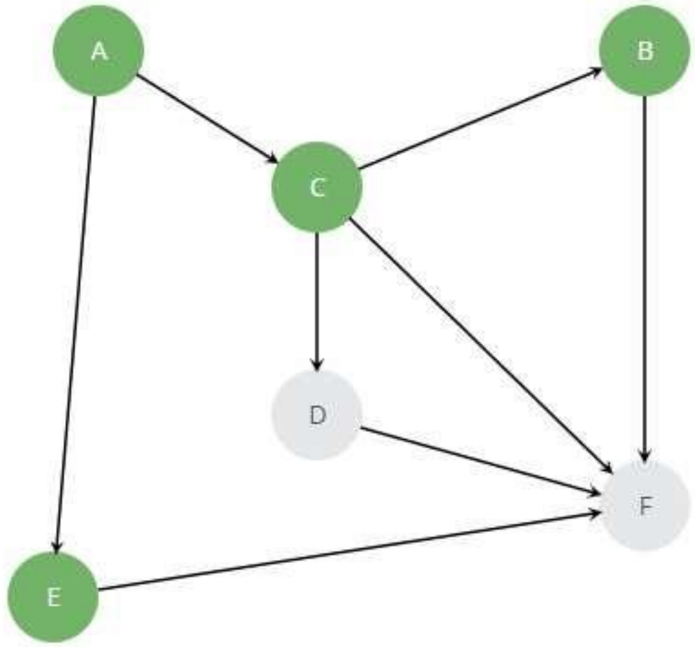
Steps:

Mark and enqueue E



Step: 6

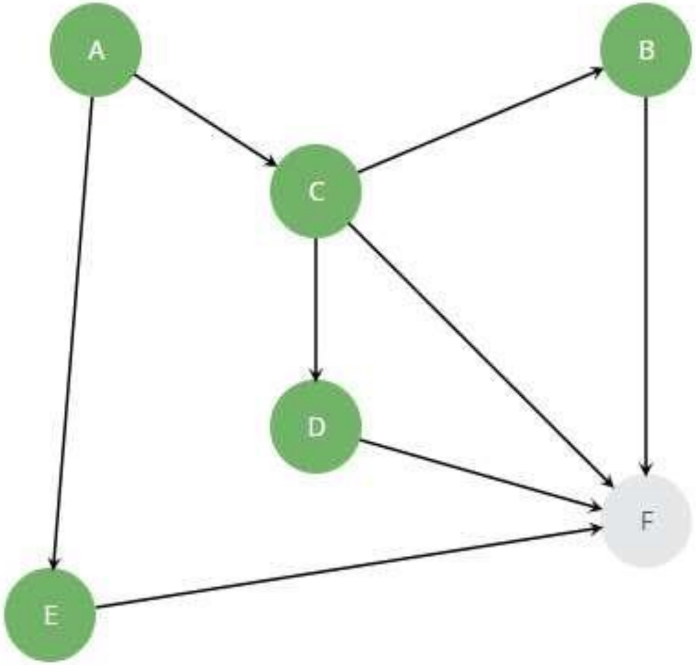
Steps:
Dequeue C



		E	B		

Step: 7

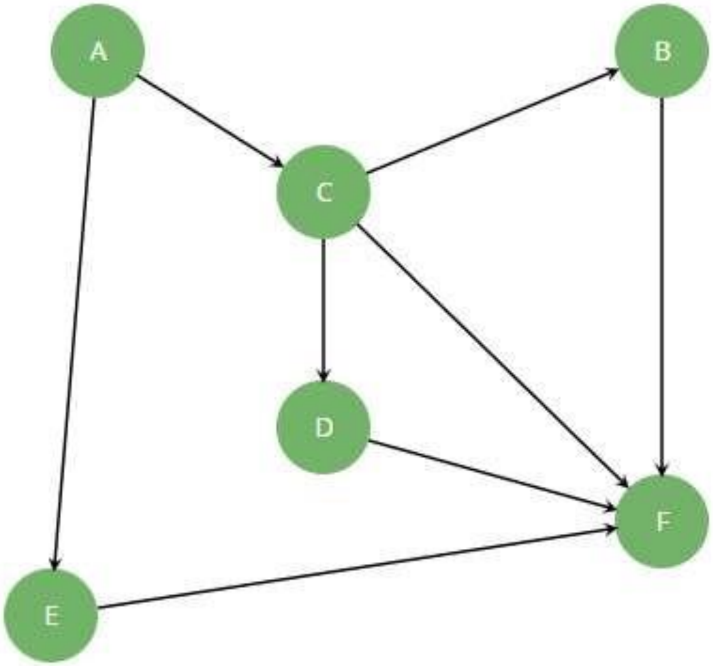
Steps:
Mark and enqueue B



		E	B	D	

Step: 8

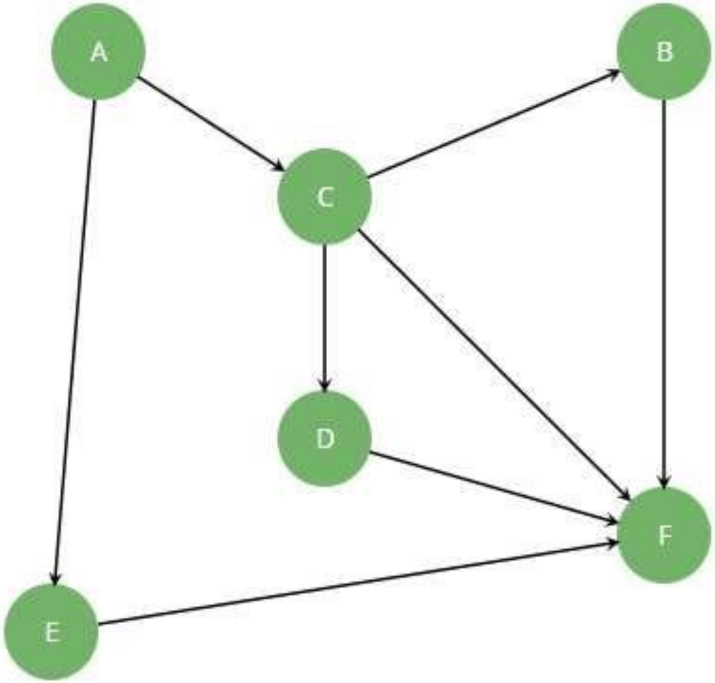
Steps:
Mark and enqueue D



		E	B	D	F

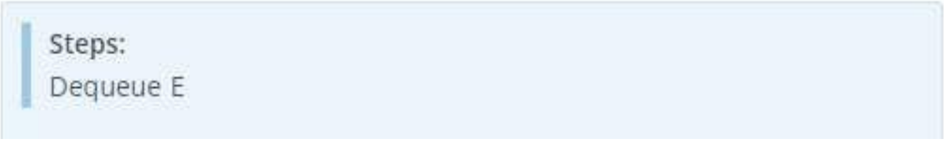
Step: 9

Steps:
Mark and enqueue F

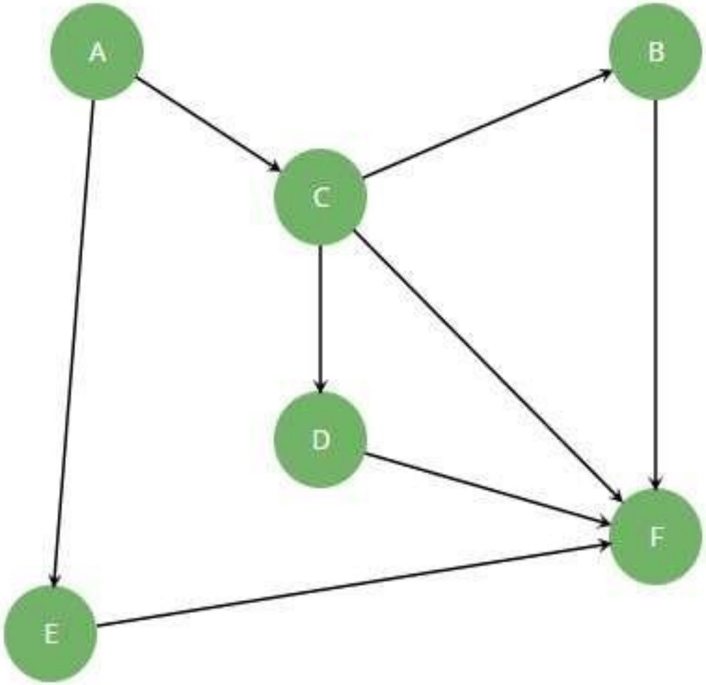


			B	D	F

Step: 10



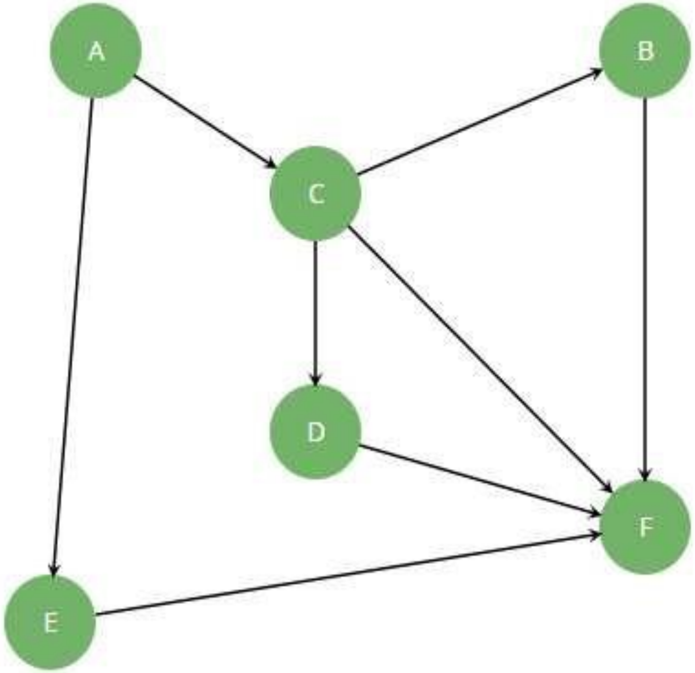
Steps:
Dequeue E



				D	F

Step: 11

Steps:
Dequeue B

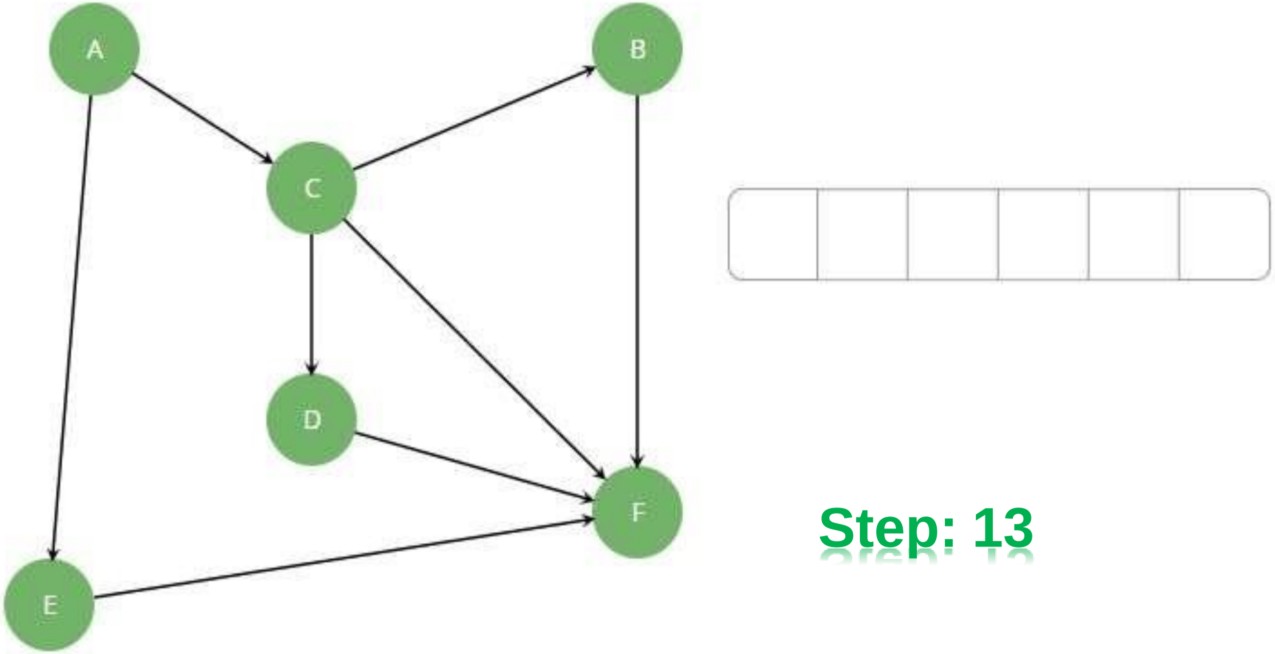


					F

Step: 12

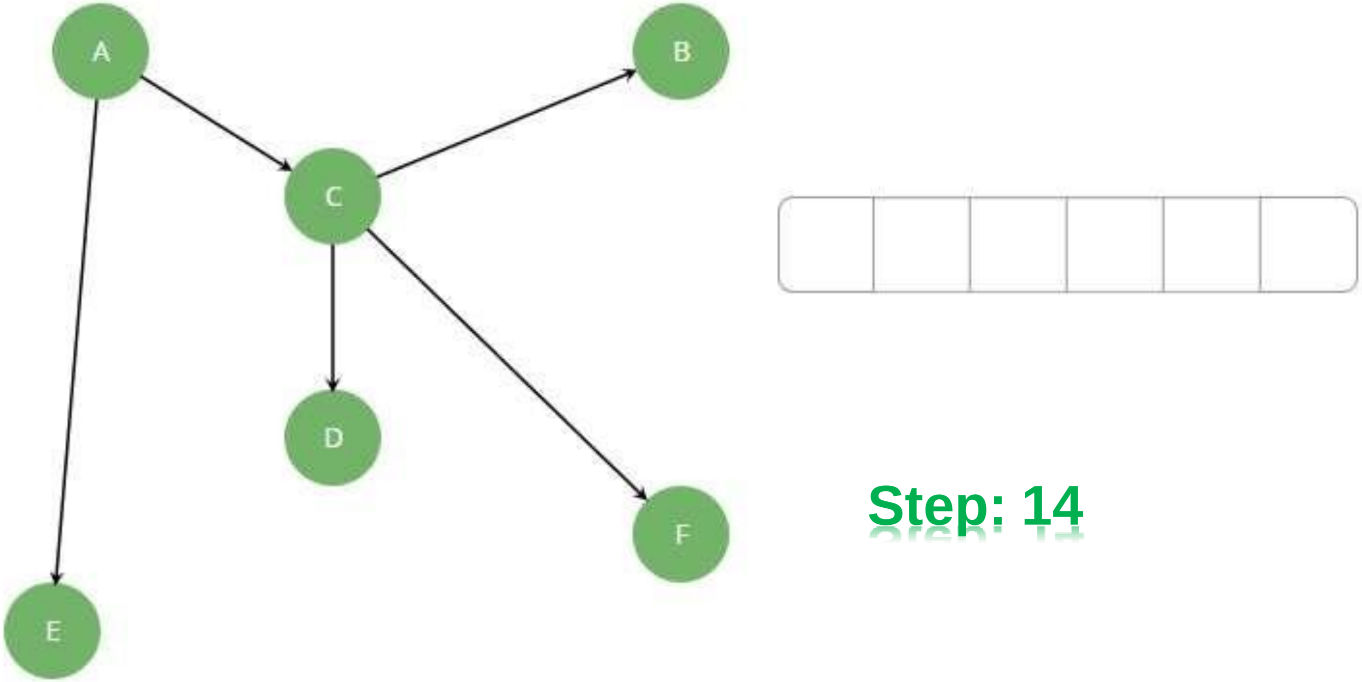
Steps:

Dequeue D



Step: 13

Steps:
Dequeue F



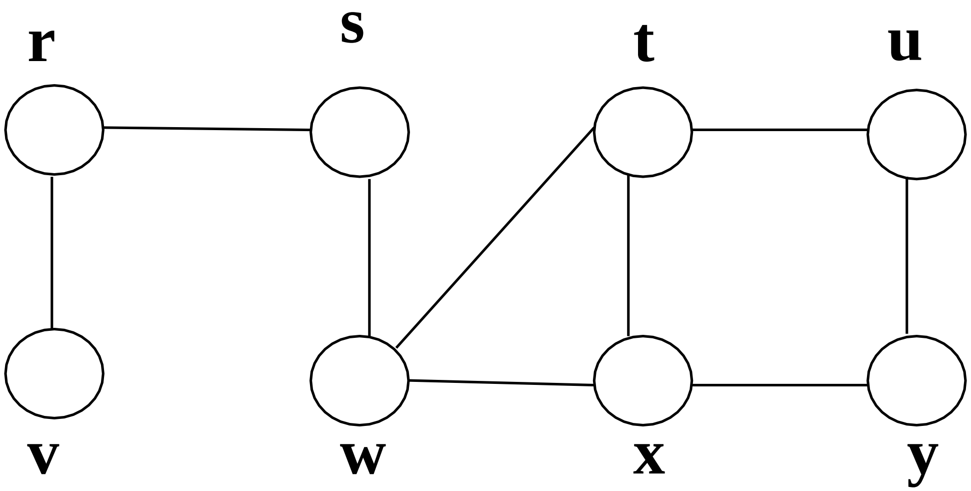
Step: 14

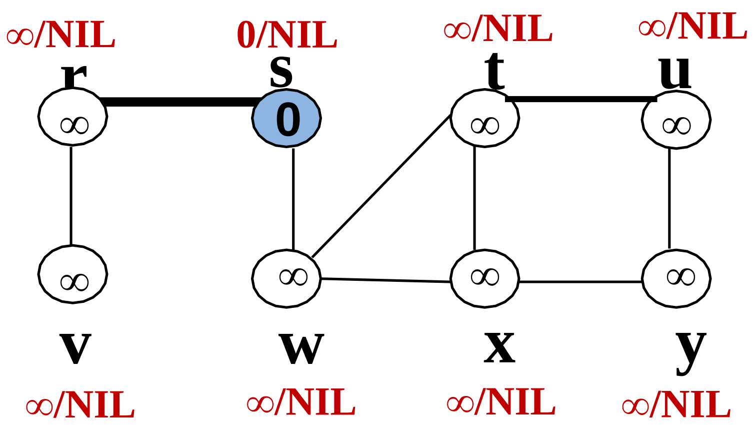
Steps:
Completed breadth first search graph

Algorithm: BFS(G, s)

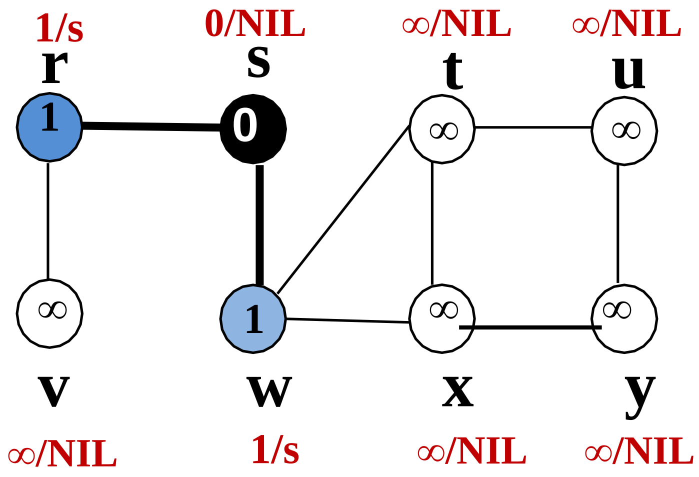
```
For each vertex u ε V[G] – { s }
    do color[u] := white  and d[u] := α and ∏[u] := NIL
color[s] := gray ,        d[s] := 0        and ∏[s] := NIL
Q := Empty
EnQueue(Q,s)                          ……….  Insert s into Q
While Q ≠ Empty
    do u := DeQueue(Q)                ……….  Delete from Q
        for each v ε Adj[u]
            do if color[v] = white
                then  color[v] := gray ,  d[v] := d[u] + 1   and ∏[v] := u
                      EnQueue(Q, v)   ………. Insert adjacent vertex v into Q
    color[u] := black
Exit
```

Graph G = (V, E)

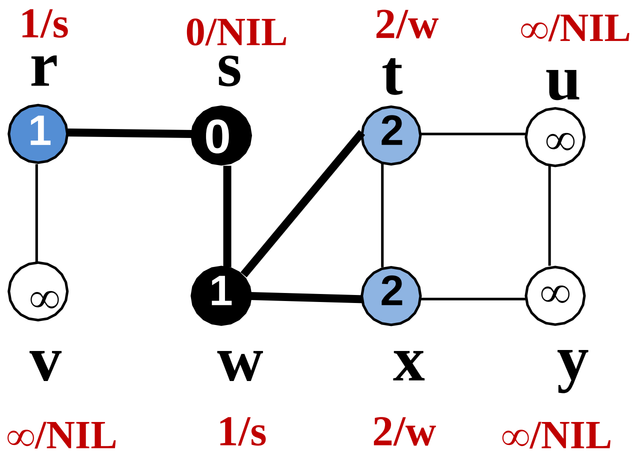



∞/NIL
0/NIL
∞/NIL
∞/NIL
r
s
t
u
∞
0
∞
∞
∞
∞
∞
∞
v
w
x
y
∞/NIL
∞/NIL
∞/NIL
∞/NIL

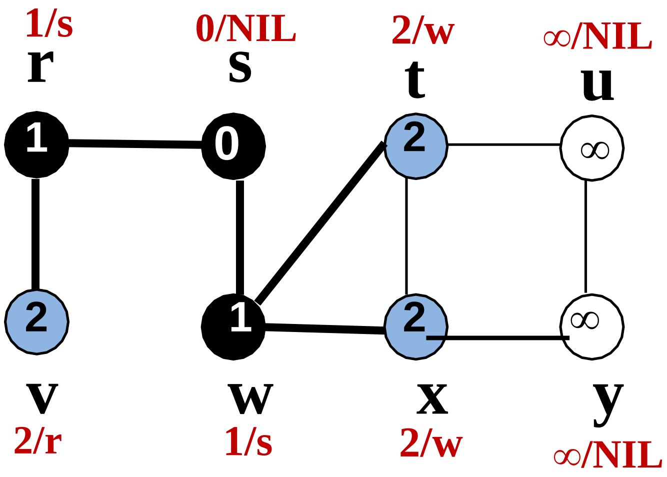
Q	s				
	0				


1/s
0/NIL
∞/NIL
∞/NIL
r
s
t
u
1
0
∞
∞
∞
1
∞
∞
v
w
x
y
∞/NIL
1/s
∞/NIL
∞/NIL

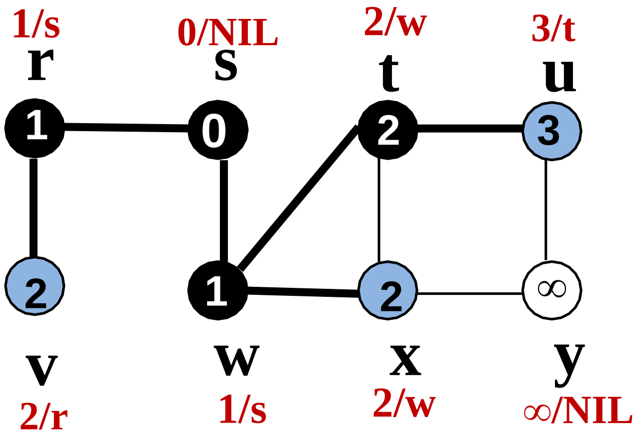
Q	w	r			
	1	1			



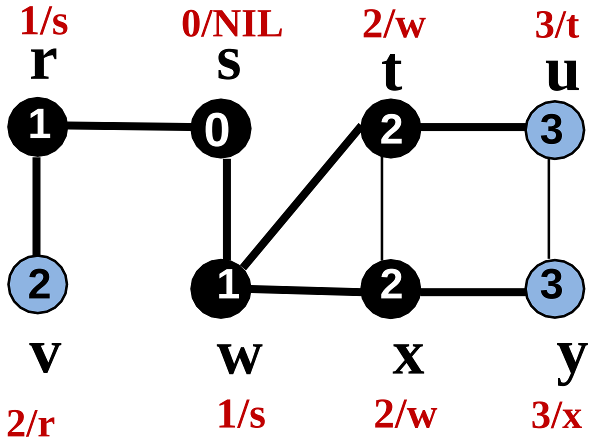
Q	r	t	x		
	1	2	2		



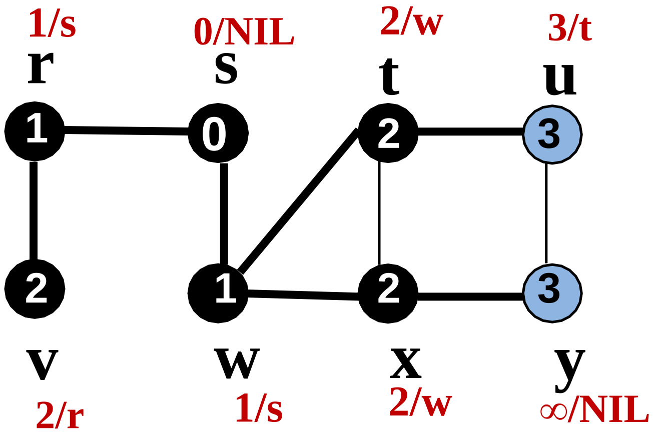
Q	t	x	v		
	1	2	2		



Q	x	v	u		
	2	2	3		

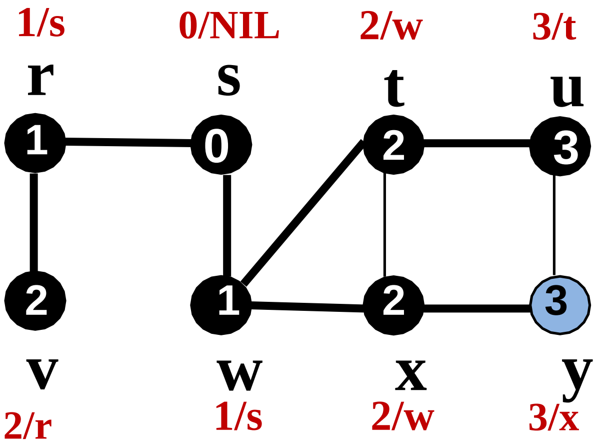


Q	v	u	y		
	2	3	3		


1/s
0/NIL
2/w
3/t
r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y
2/r
1/s
2/w
∞/NIL

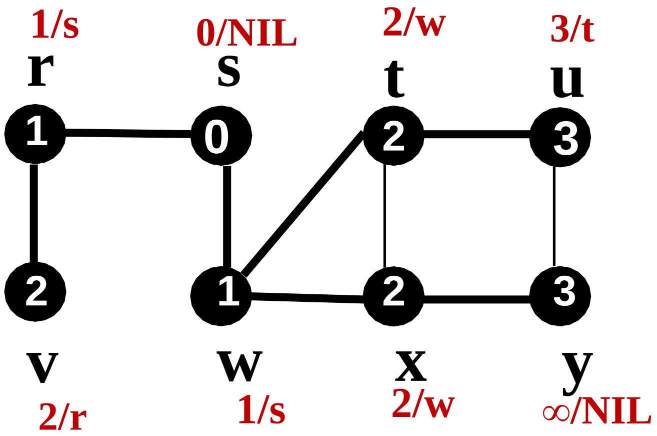
Q

u	y			
3	3			

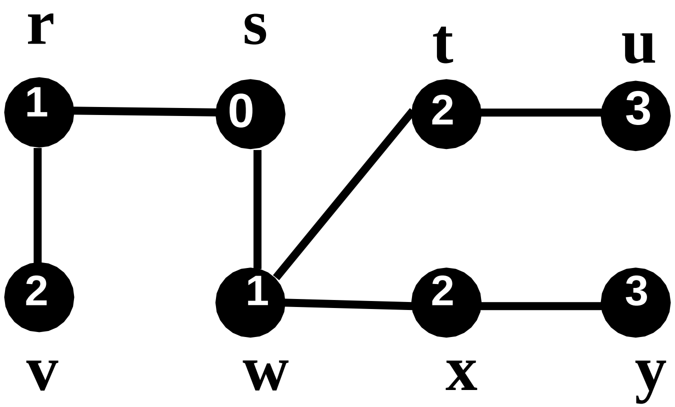

1/s
0/NIL
2/w
3/t
r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y
2/r
1/s
2/w
3/x

Q

y				
3				

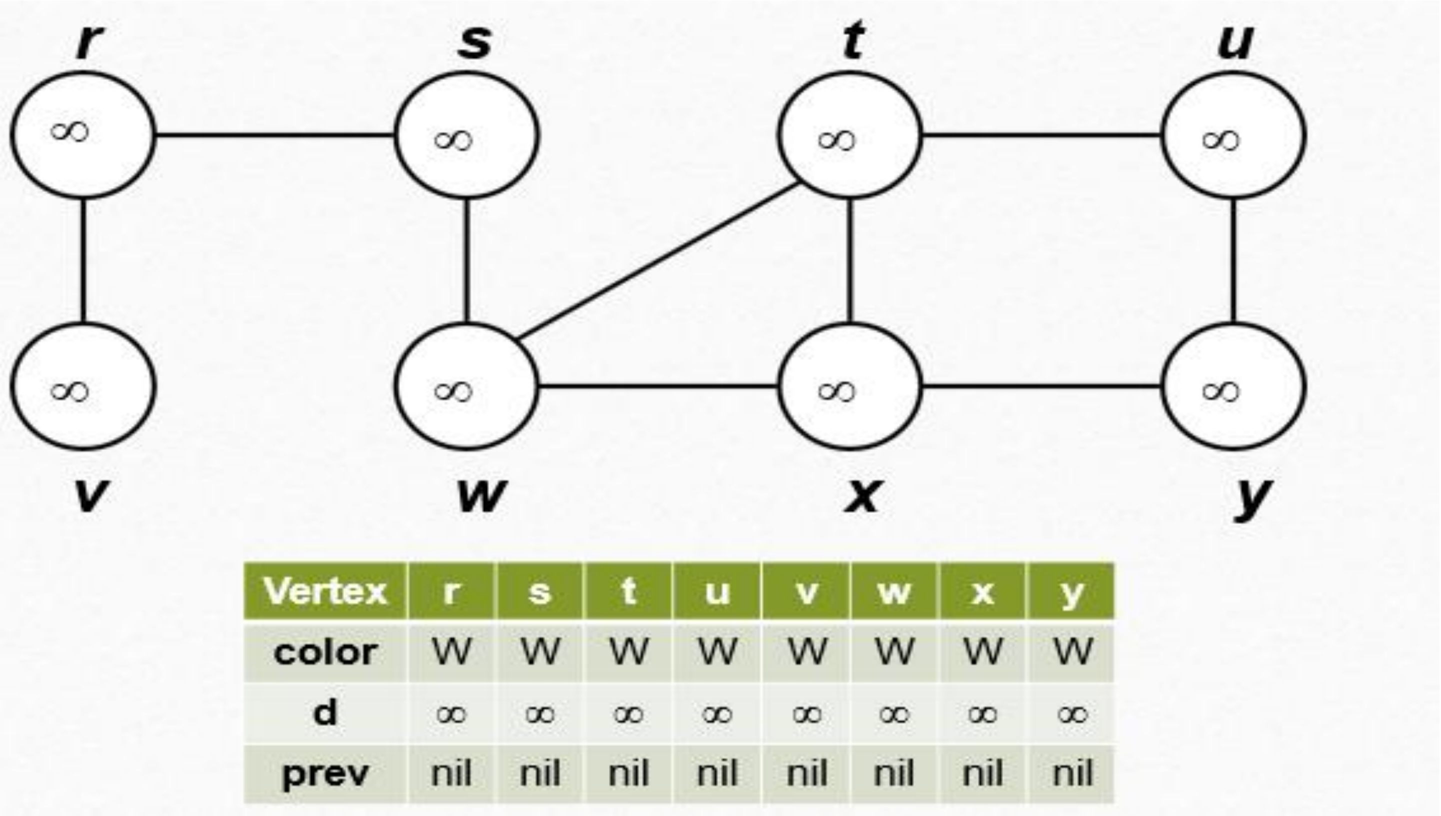


Empty



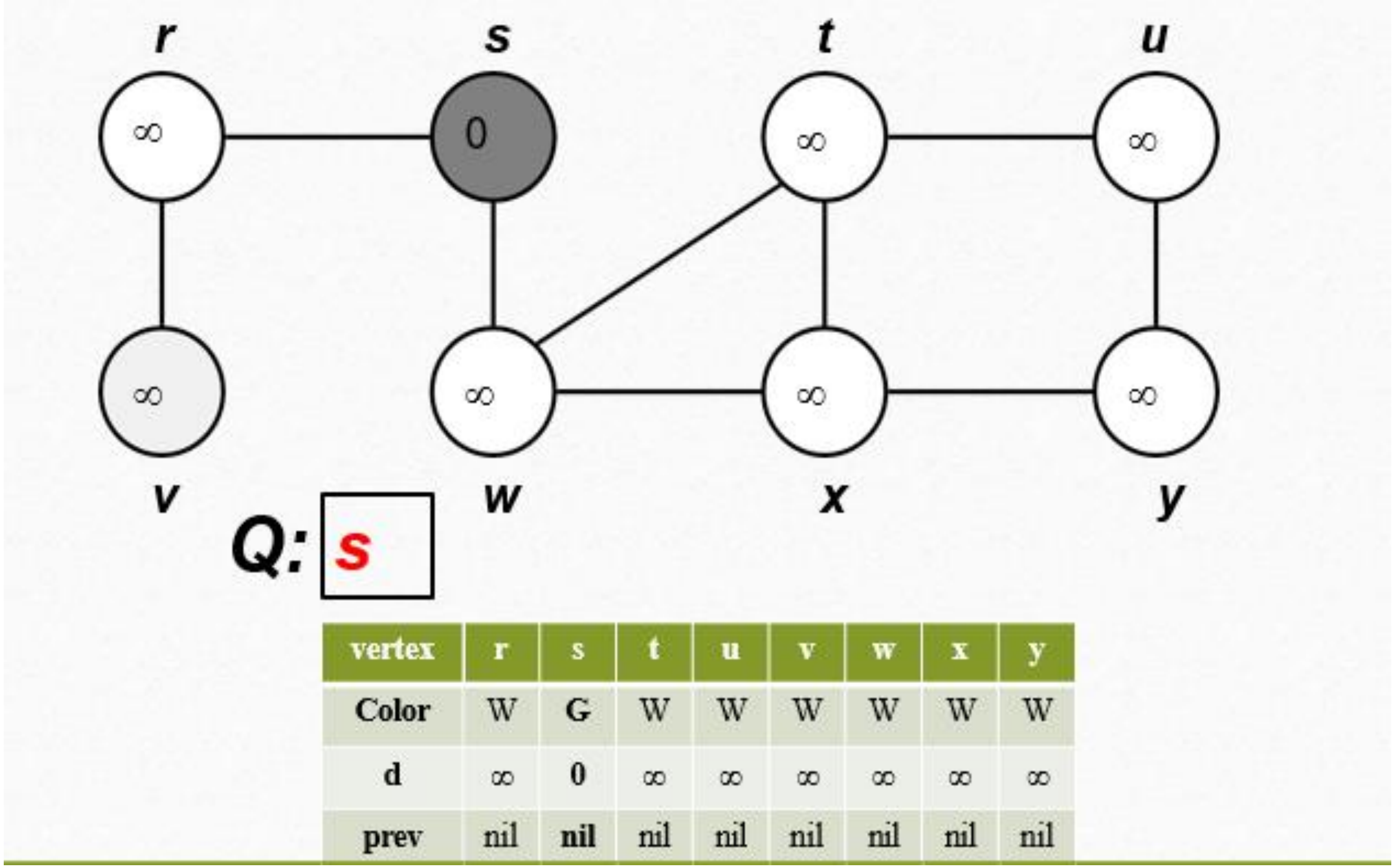
Spanning Tree

Breadth-First Search: Example



Vertex	r	s	t	u	v	w	x	y
color	W	W	W	W	W	W	W	W
d	∞	∞	∞	∞	∞	∞	∞	∞
prev	nil	nil	nil	nil	nil	nil	nil	nil

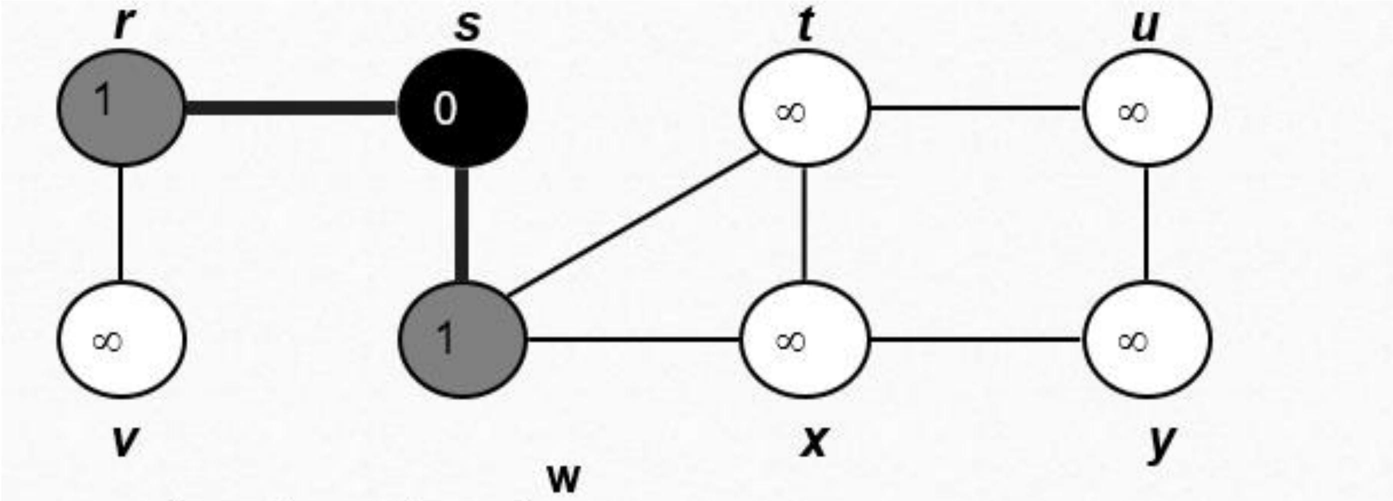
Breadth-First Search: Example



Q: s

vertex	r	s	t	u	v	w	x	y
Color	W	G	W	W	W	W	W	W
d	∞	0	∞	∞	∞	∞	∞	∞
prev	nil	nil	nil	nil	nil	nil	nil	nil

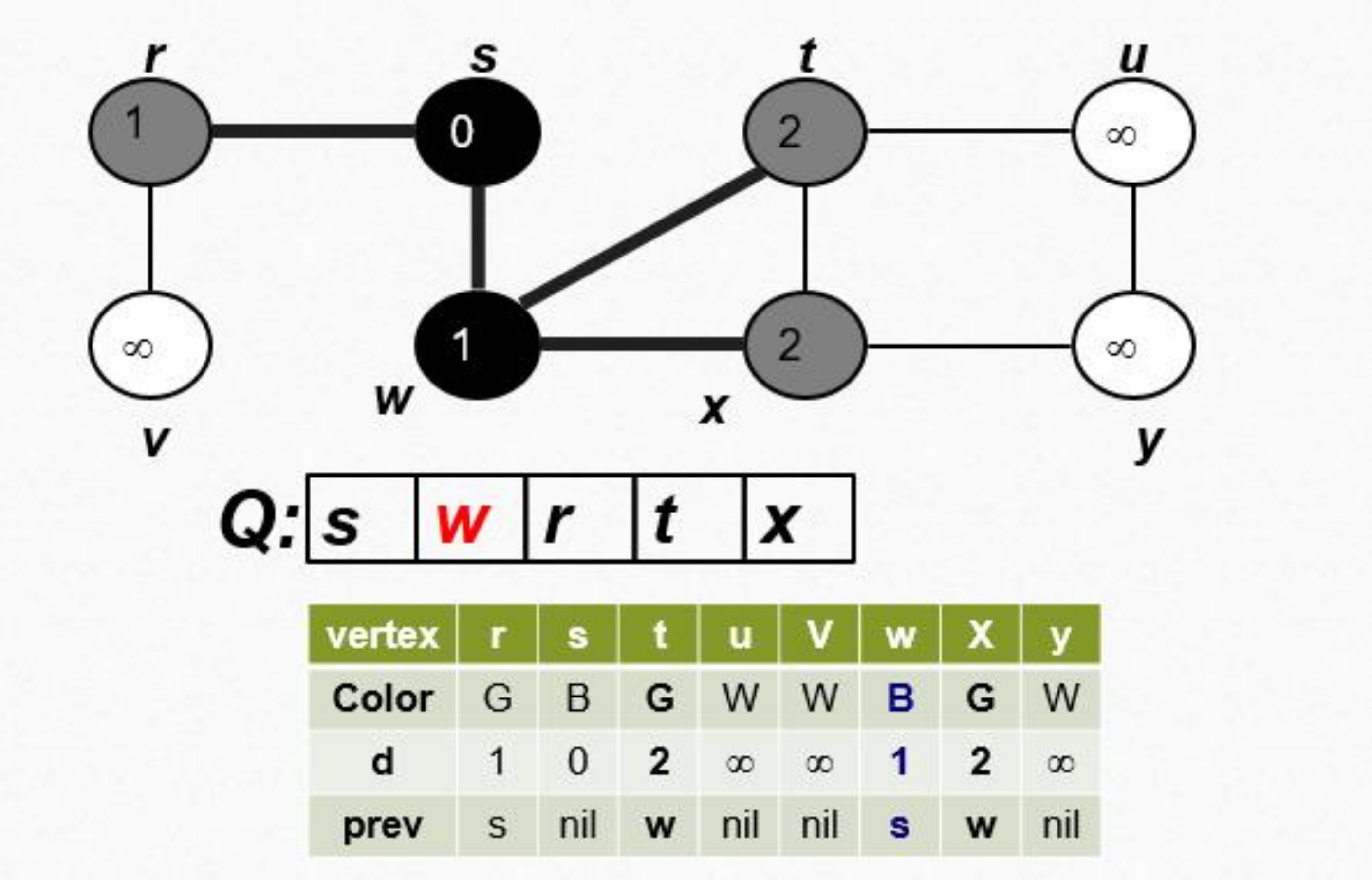
Breadth-First Search: Example



Q: | *s* | *w* | *r* |

vertex	r	s	t	u	v	w	x	y
Color	G	B	W	W	W	G	W	W
d	1	0	∞	∞	∞	1	∞	∞
prev	s	nil	nil	nil	nil	s	nil	nil

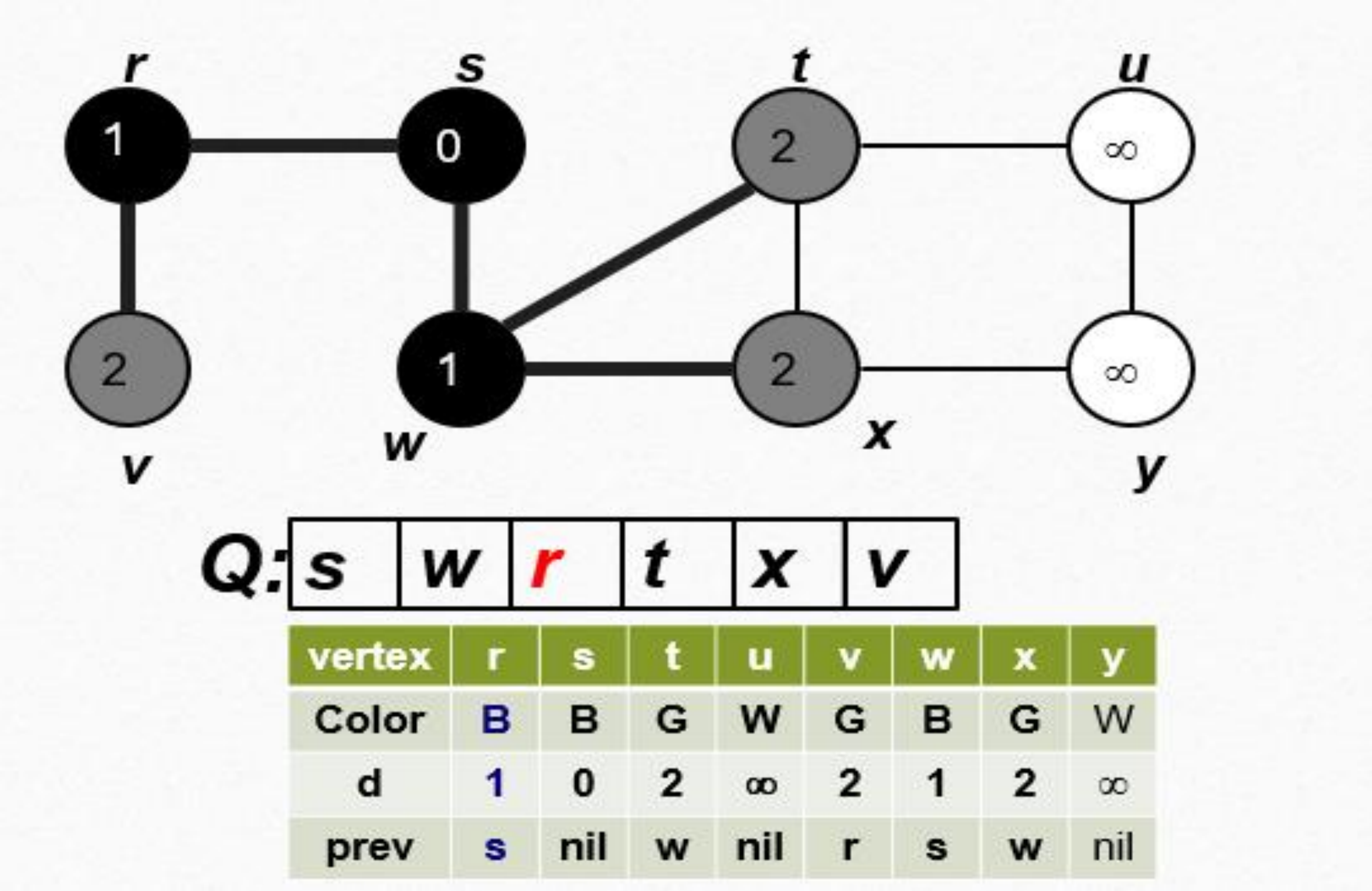
Breadth-First Search: Example



Q: *s* *w* *r* *t* *x*

vertex	r	s	t	u	V	w	X	y
Color	G	B	**G**	W	W	**B**	**G**	W
d	1	0	**2**	∞	∞	**1**	**2**	∞
prev	s	nil	**w**	nil	nil	**s**	**w**	nil

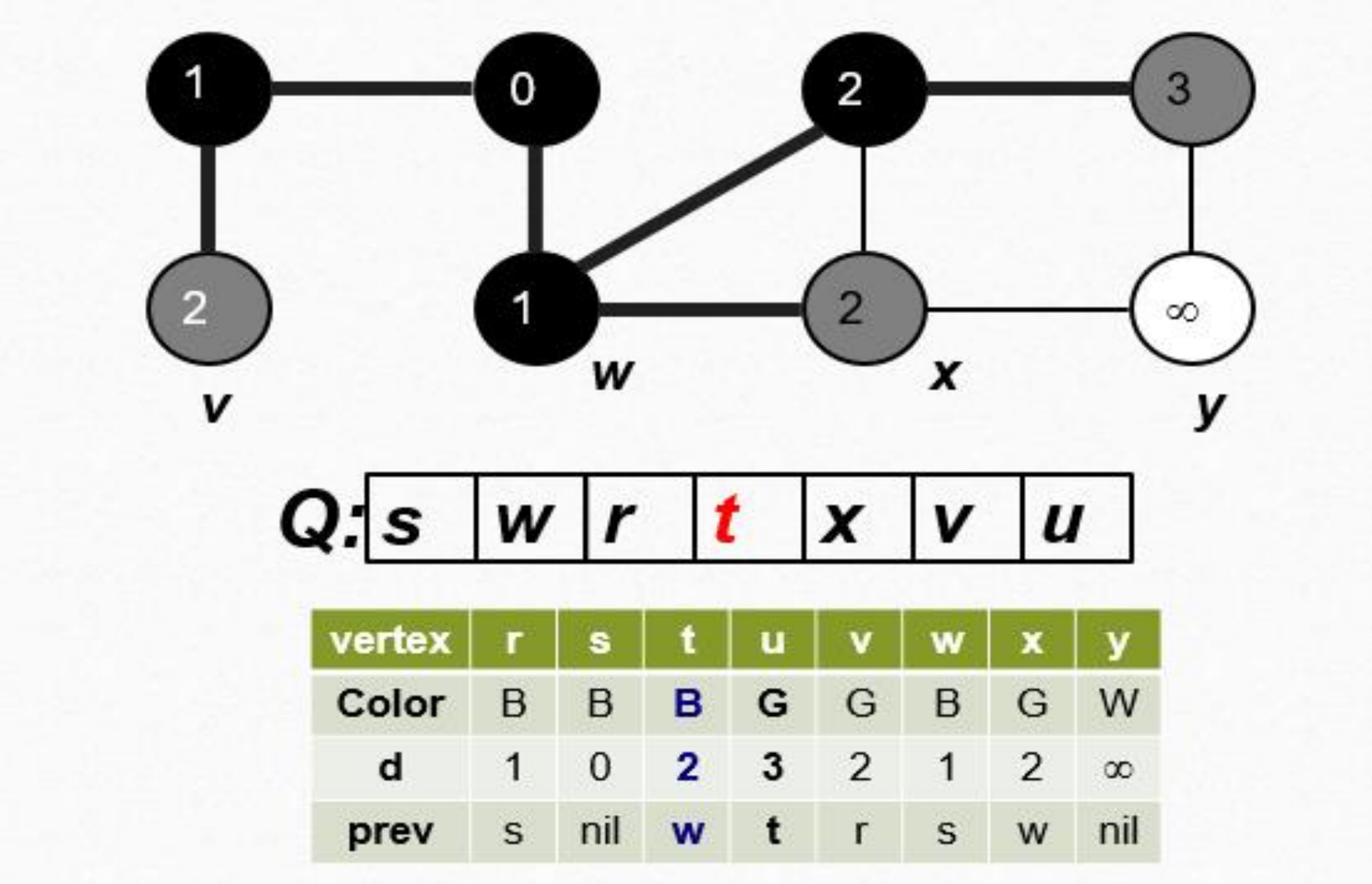
Breadth-First Search: Example



Q:					
s	w	r	t	x	v

vertex	r	s	t	u	v	w	x	y
Color	B	B	G	W	G	B	G	W
d	1	0	2	∞	2	1	2	∞
prev	s	nil	w	nil	r	s	w	nil

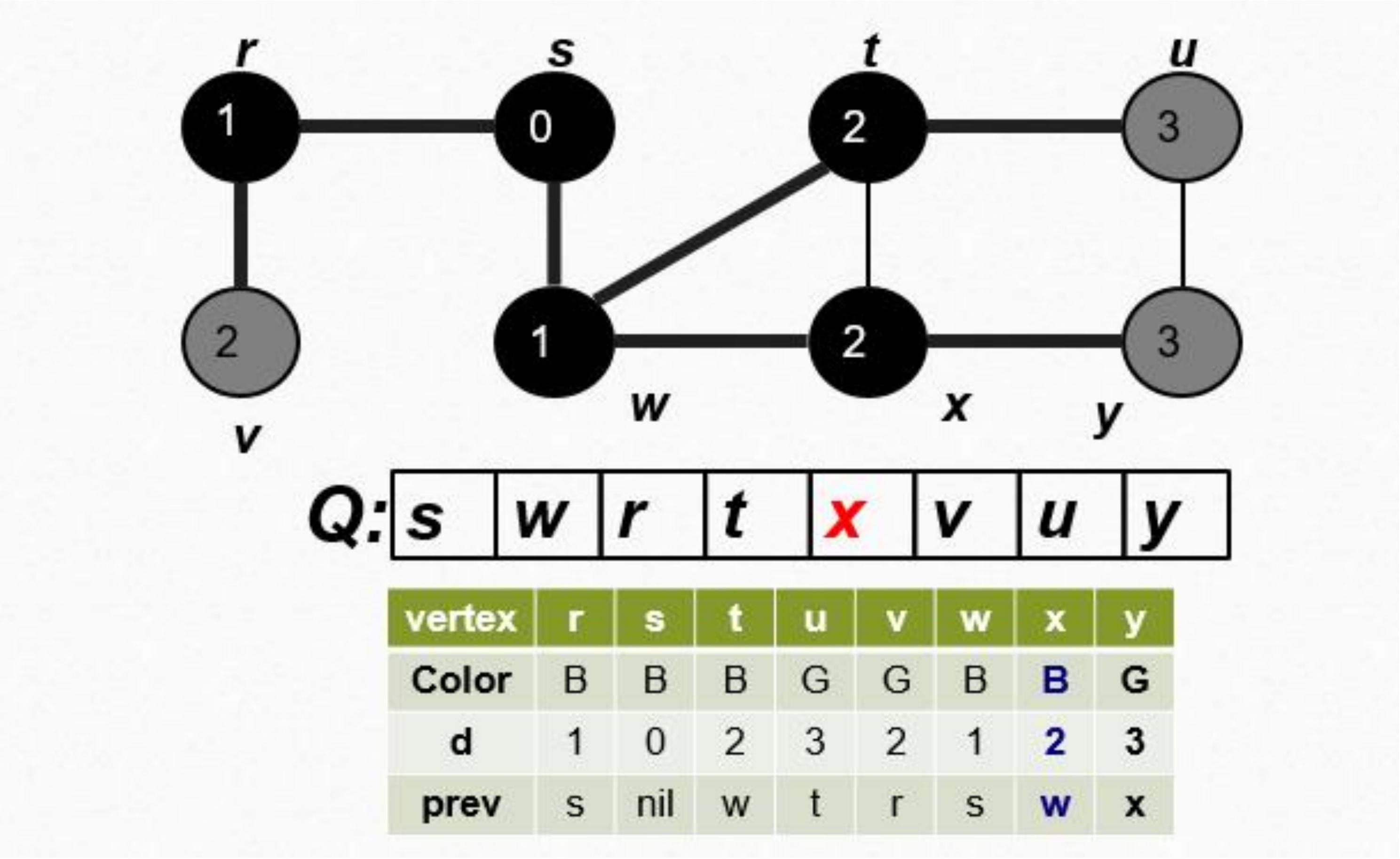
Breadth-First Search: Example



Q: *s* | *w* | *r* | *t* | *x* | *v* | *u*

vertex	r	s	t	u	v	w	x	y
Color	B	B	**B**	**G**	G	B	G	W
d	1	0	**2**	**3**	2	1	2	∞
prev	s	nil	**w**	**t**	r	s	w	nil

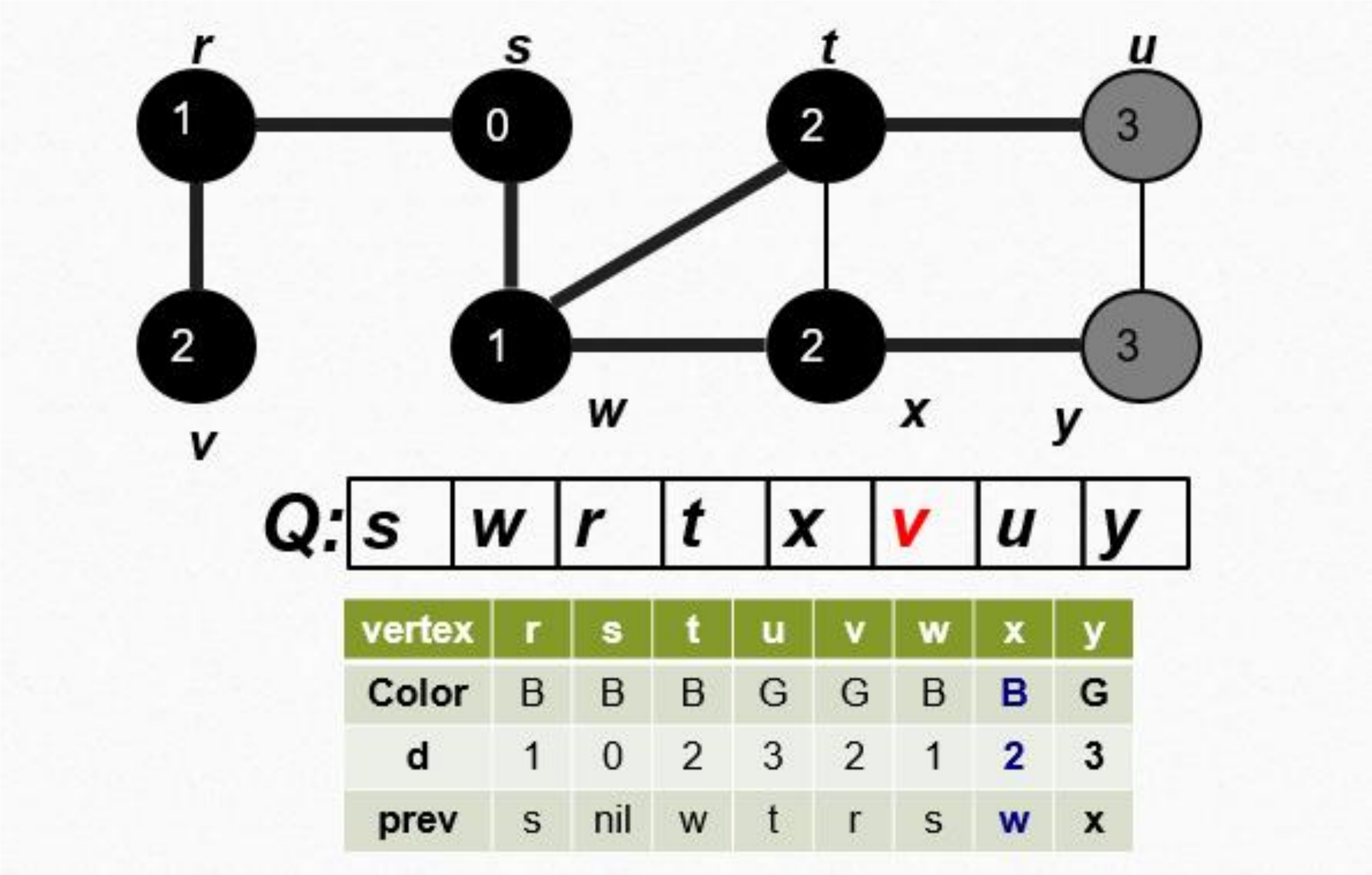
Breadth-First Search: Example



Q:	s	w	r	t	x	v	u	y

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	G	B	**B**	**G**
d	1	0	2	3	2	1	**2**	**3**
prev	s	nil	w	t	r	s	**w**	**x**

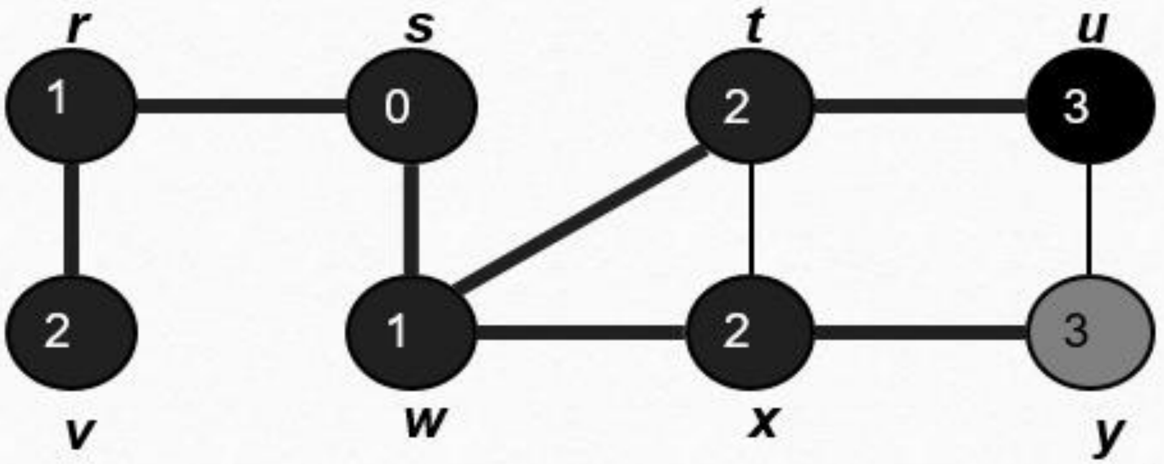
Breadth-First Search: Example



Q:	*s*	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	G	B	**B**	**G**
d	1	0	2	3	2	1	**2**	**3**
prev	s	nil	w	t	r	s	**w**	**x**

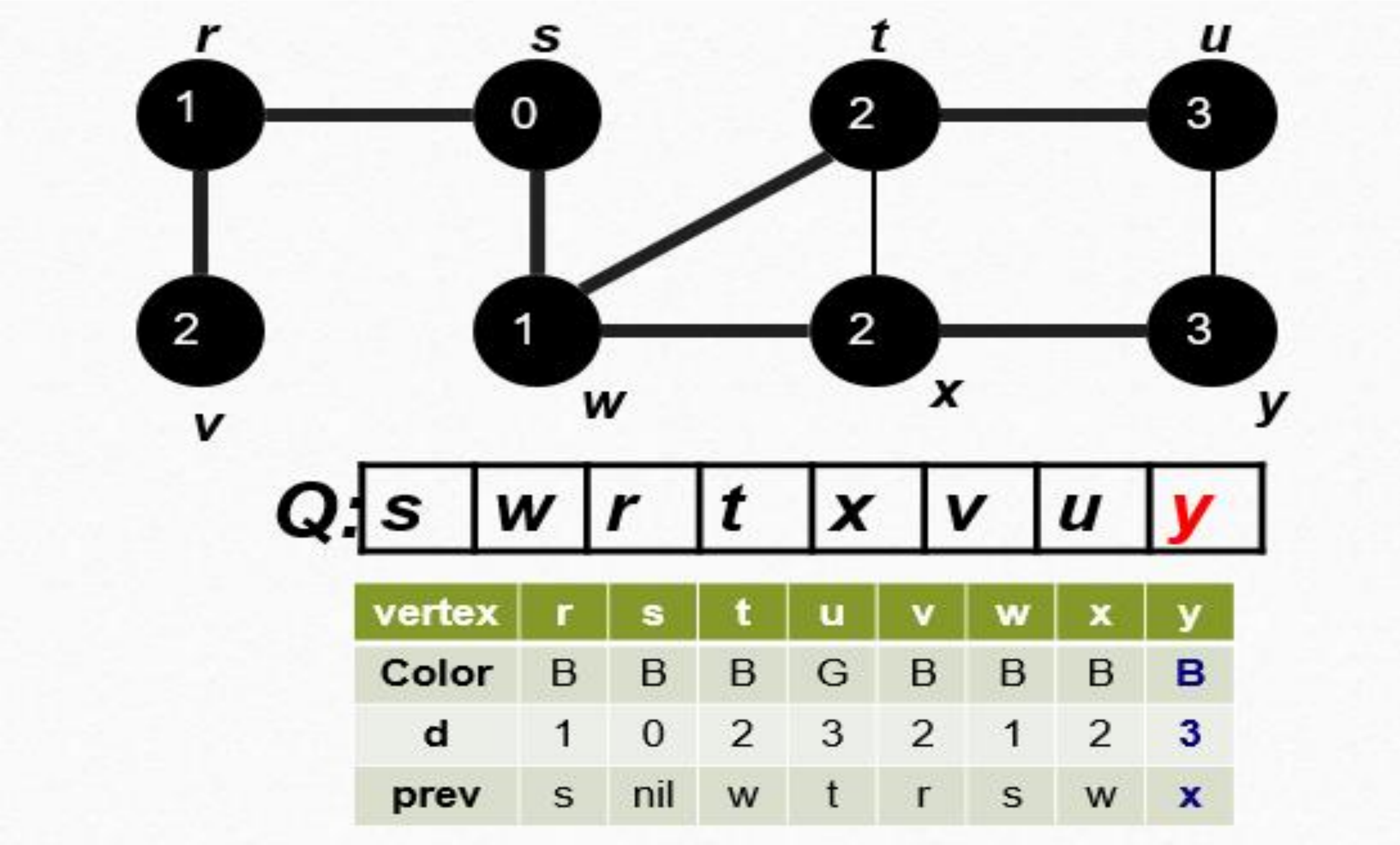
Breadth-First Search: Example



Q: *s* | *w* | *r* | *t* | *x* | *v* | *u* | *y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	**B**	B	B	B	G
d	1	0	2	**3**	2	1	2	3
prev	s	nil	w	**t**	r	s	w	x

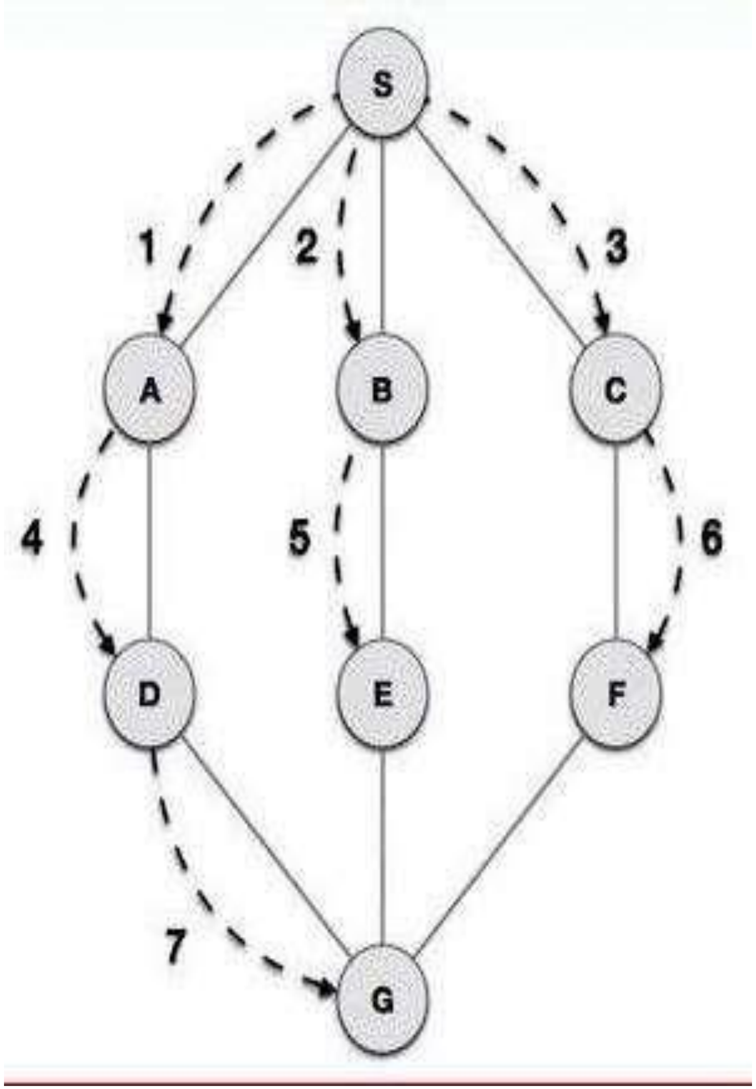
Breadth-First Search: Example



Q: s | w | r | t | x | v | u | y

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	B	B	B	**B**
d	1	0	2	3	2	1	2	**3**
prev	s	nil	w	t	r	s	w	**x**

Example



1. Visit S and take it to the queue.
2. Then visit adjacent node from S,lets take A and enqueue it.
3. Move to another node which is adjacent to S,which is B and enqueue it.
4. Then C and enqueue it.
5. We have ABC in queue.
6. Now there is no any adjacent node to Sso, we move to next node A.
7. Now we have BCD in queue.
8. Then for B, we have CDE.
9. At C node we have DEF in queue.
10. Now it turn for D and we have E F Gin queue.
11. Checking in Eand Fas there are no adjacent node so we need to dequeue them.
12. At last Gis remain in the queue.
13. As there is no adjacent node Gis dequeue from the queue.

Our final output will be S A B C D E F G.

Running Time of BFS Algorithm

After initialization (line 1-2), no vertex is ever whitened and line 9 ensures that each vertex is enqueued at most once and so dequeued at most once. The operations of enqueuing and dequeuing take 0(1) time, so total time devoted to queue operations is 0(V).

Since the adjacency list of each vertex is scanned only when the vertex is dequeued, the adjacency list of each vertex is scanned at most once. At most 0(E) time is spent in total scanning adjacency lists.

So the total running time of BFS is 0(V+E).

Thank you