Graph Algorithms I

Breadth First Search

Graphs

Graph G = (V, E)

V: Set of Nodes (Vertices)

E: Set of Edges

Graphs

Graph G = (V, E)

V: Set of Nodes (Vertices)

E: Set of Edges

Let |V| = n

Let |E| = m

Undirected graph:

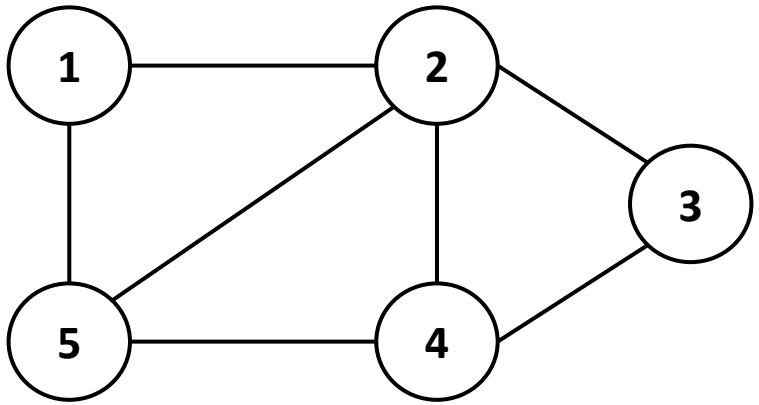
Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

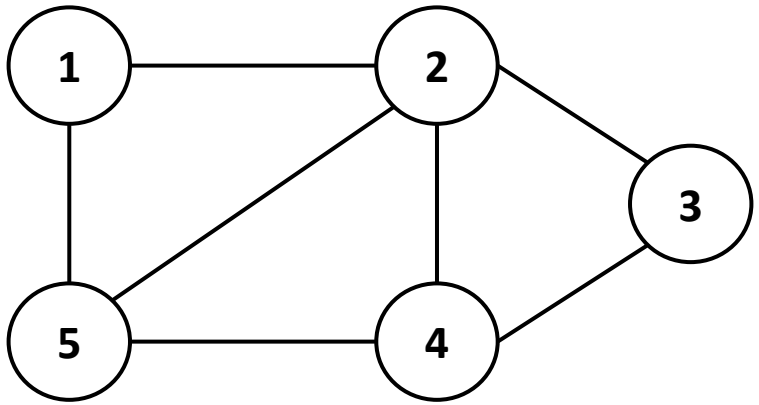
Example:



Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:

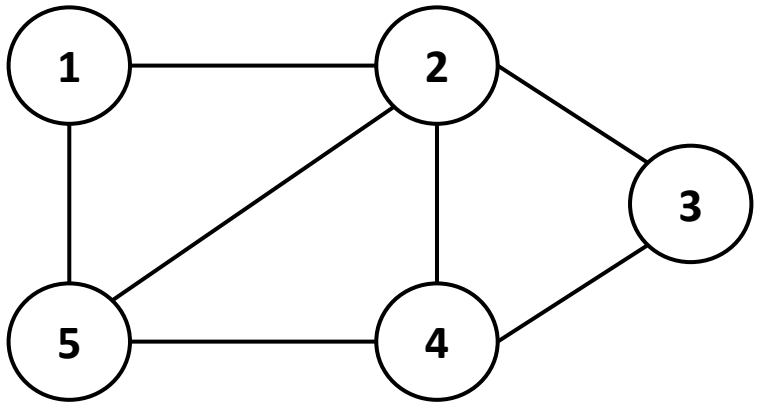


V = {1, 2, 3, 4, 5}

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



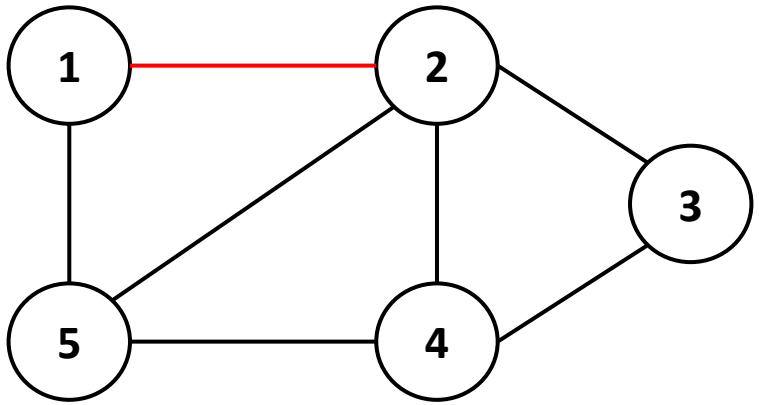
V = {1, 2, 3, 4, 5}

E =

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



V = {1, 2, 3, 4, 5}

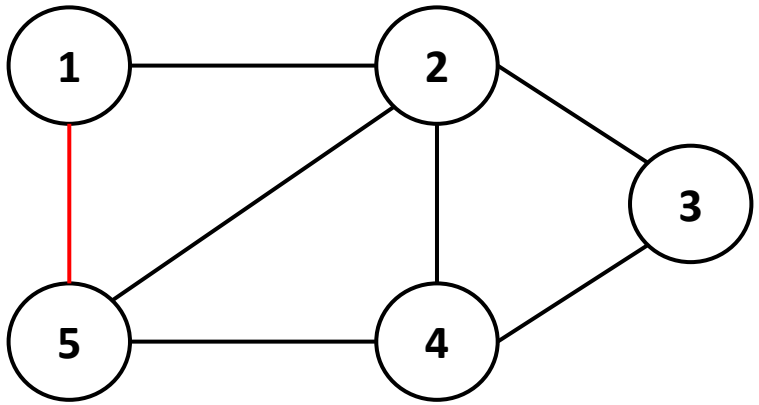
E = { (1, 2),

}

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



V = {1, 2, 3, 4, 5}

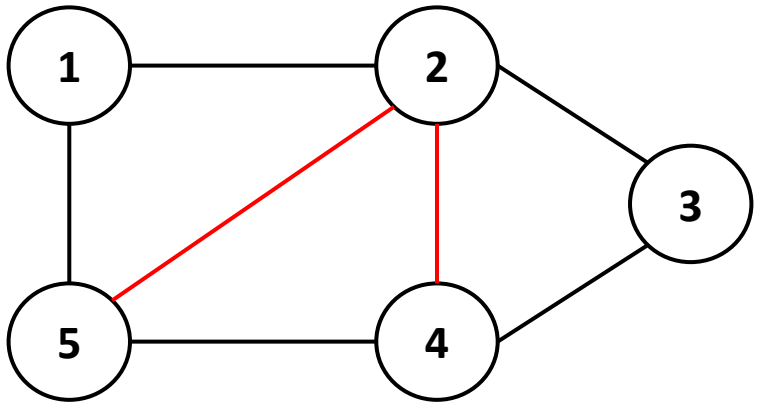
E = { (1, 2), (1, 5),

}

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



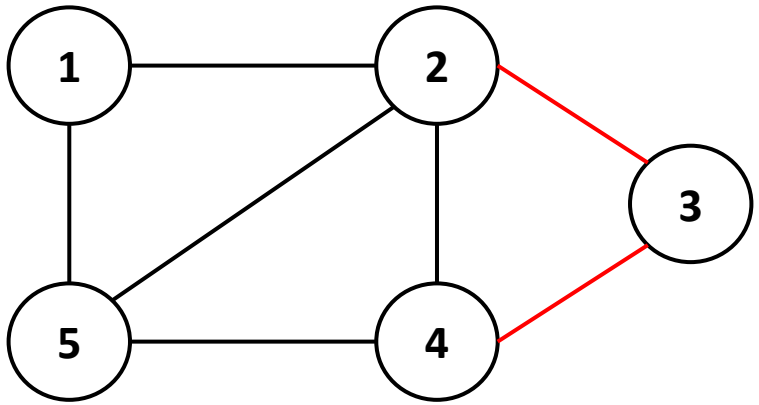
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
}

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



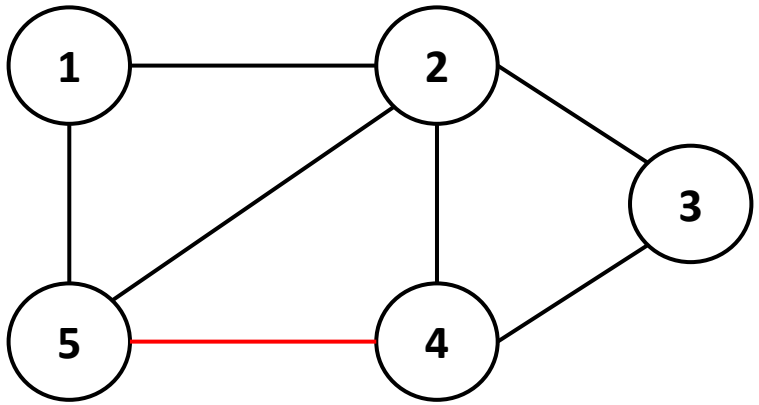
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), }

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



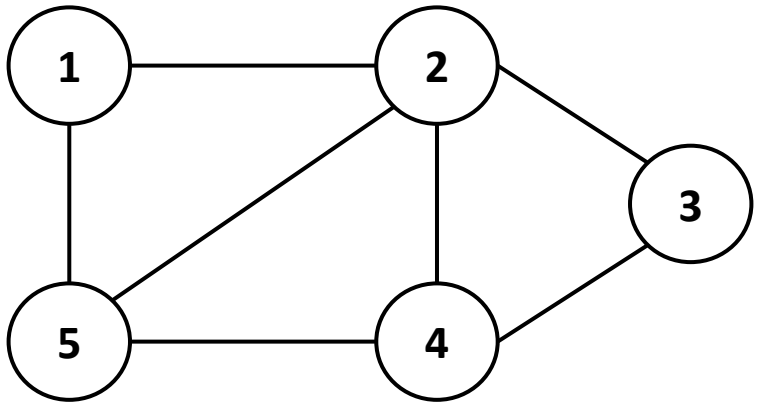
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



V = {1, 2, 3, 4, 5}

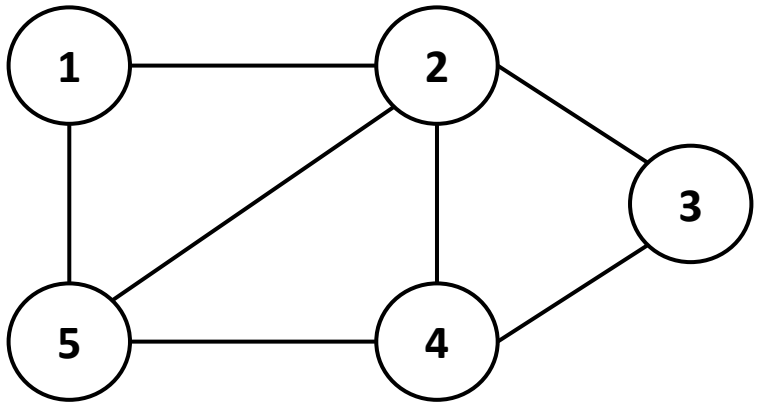
E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

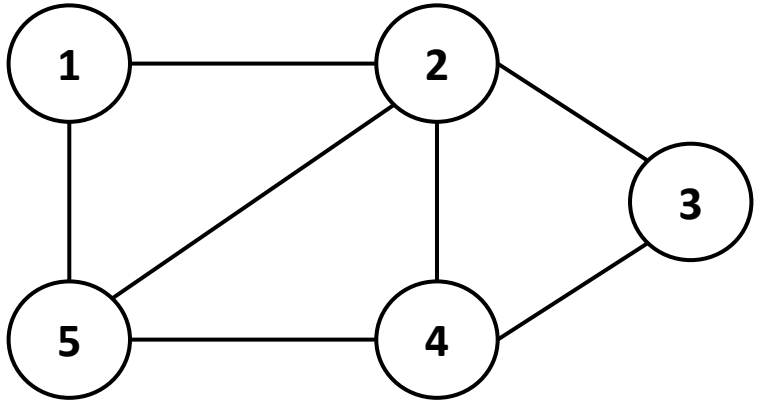
Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



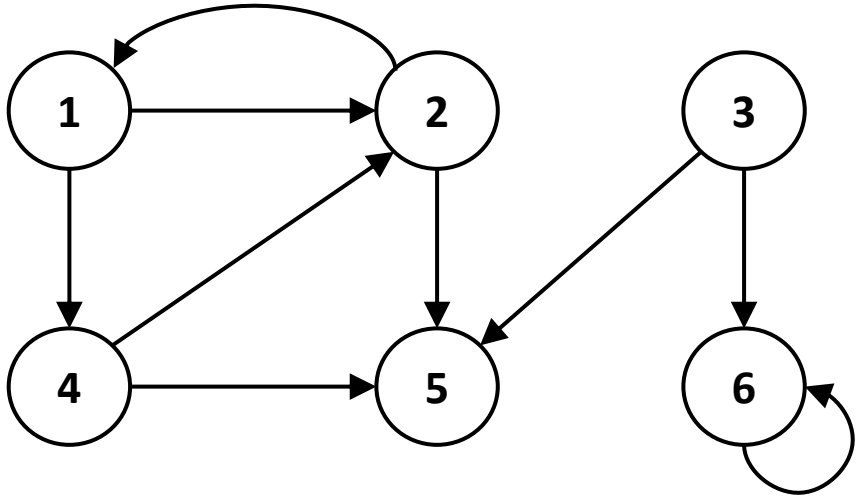
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

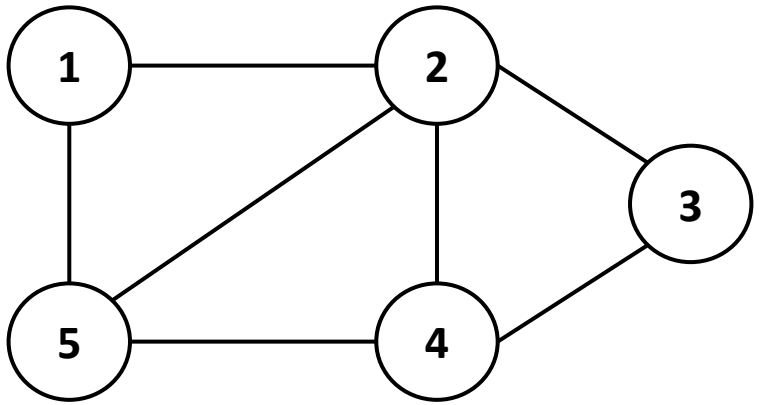
Example:



Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



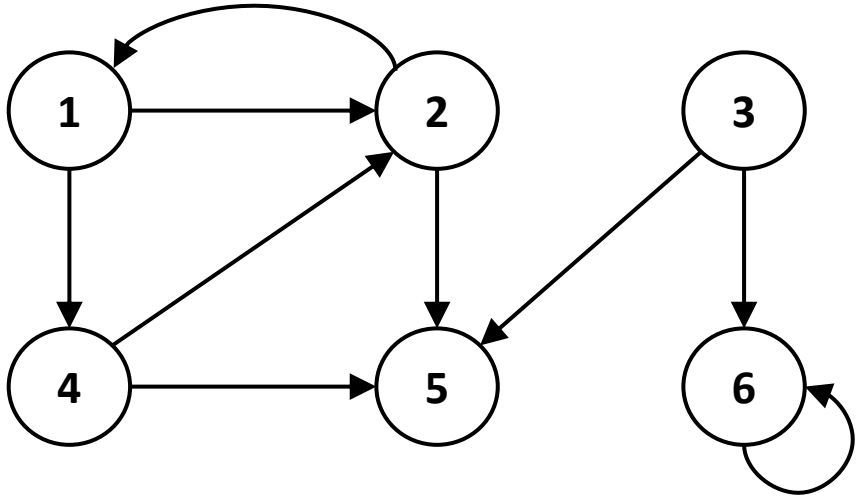
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Example:

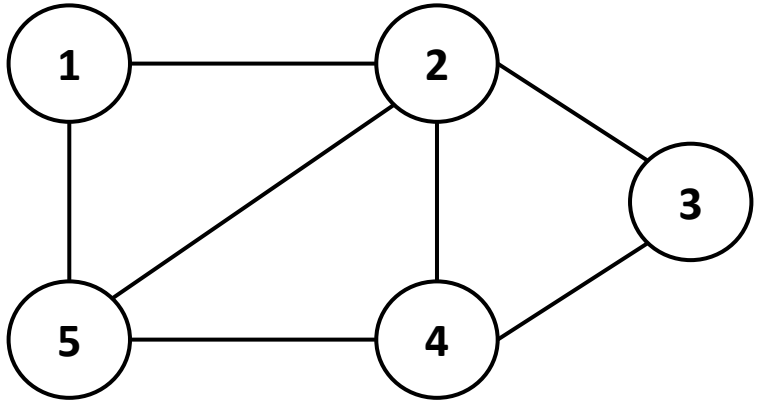


V = {1, 2, 3, 4, 5, 6}

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



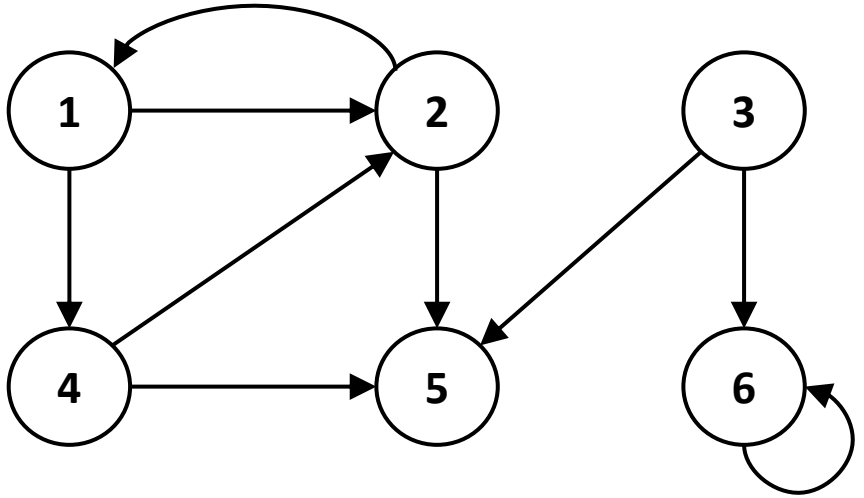
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Example:



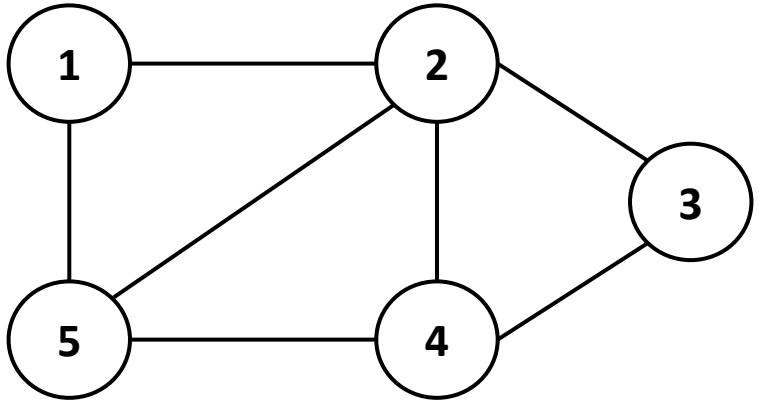
V = {1, 2, 3, 4, 5, 6}

E = { }

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



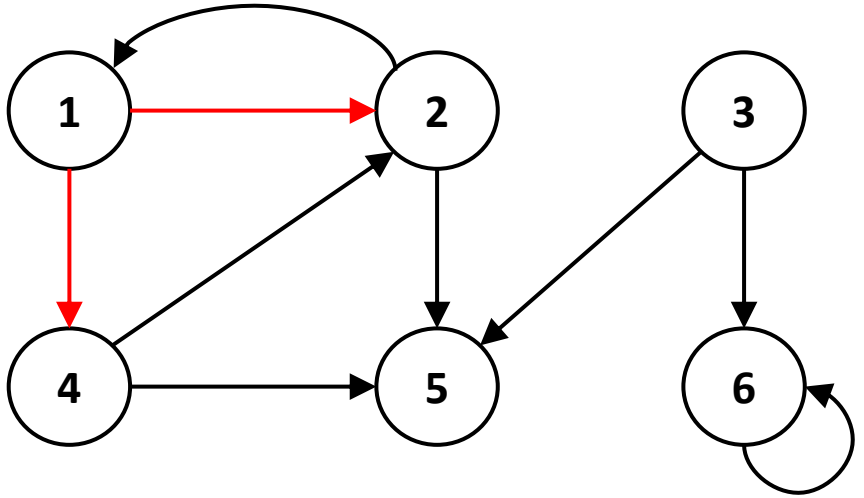
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Example:



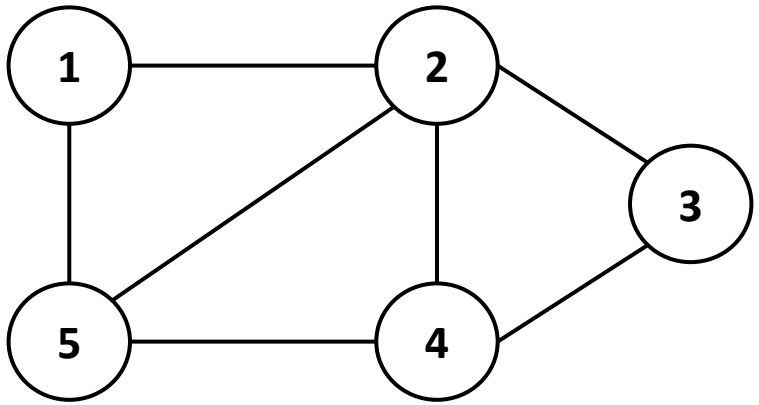
V = {1, 2, 3, 4, 5, 6}

E = { (1, 2), (1, 4),
}

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



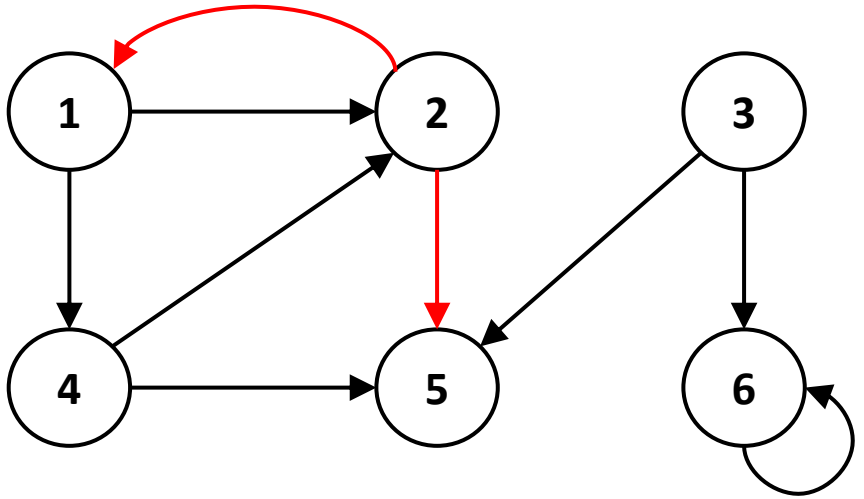
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Example:



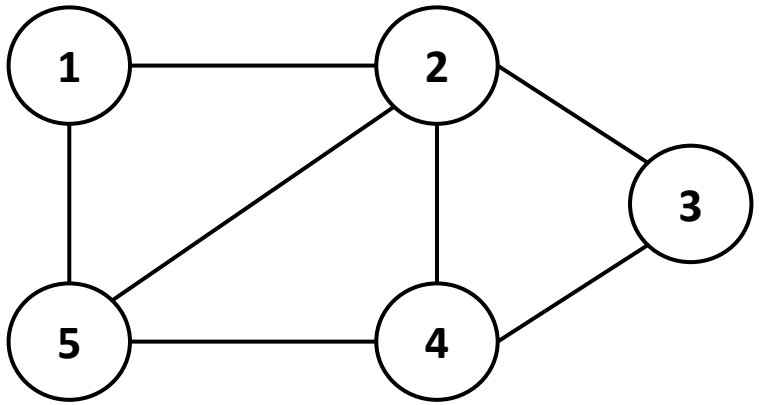
V = {1, 2, 3, 4, 5, 6}

E = { (1, 2), (1, 4), (2, 1), (2, 5),
}

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



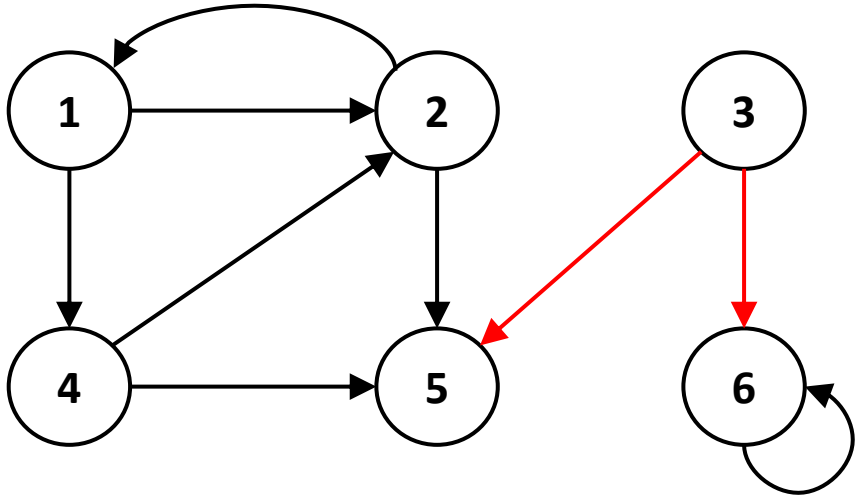
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Example:



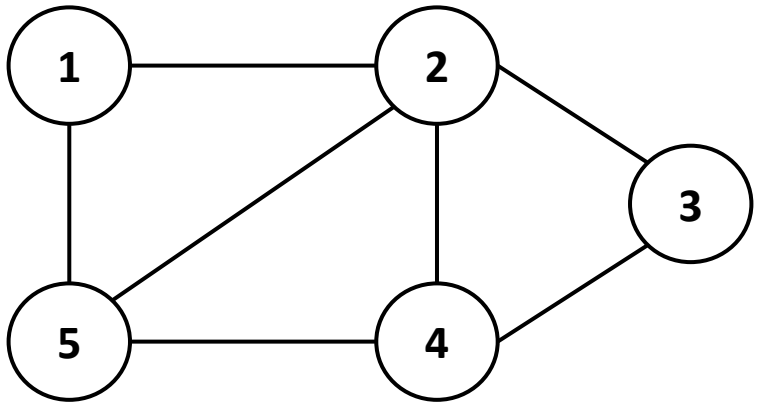
V = {1, 2, 3, 4, 5, 6}

E = { (1, 2), (1, 4), (2, 1), (2, 5),
(3, 5), (3, 6), }

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



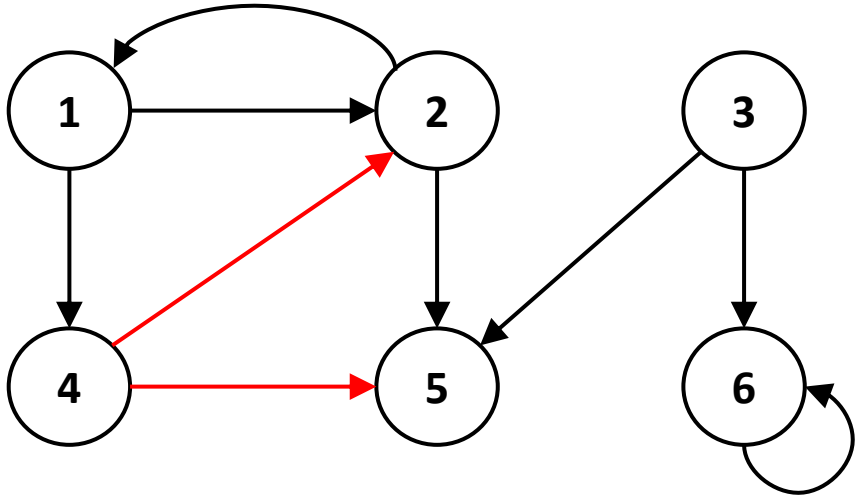
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Example:



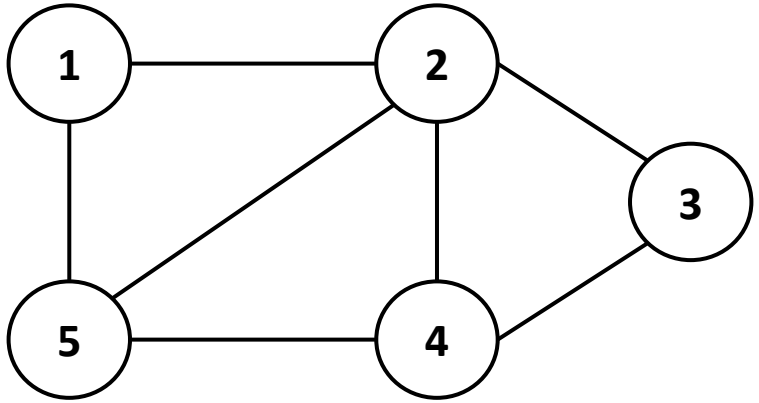
V = {1, 2, 3, 4, 5, 6}

E = { (1, 2), (1, 4), (2, 1), (2, 5),
(3, 5), (3, 6), (4, 2), (4, 5), }

Undirected graph:

Each edge is an **unordered** pair (u,v)
i.e. (u, v) = (v, u)

Example:



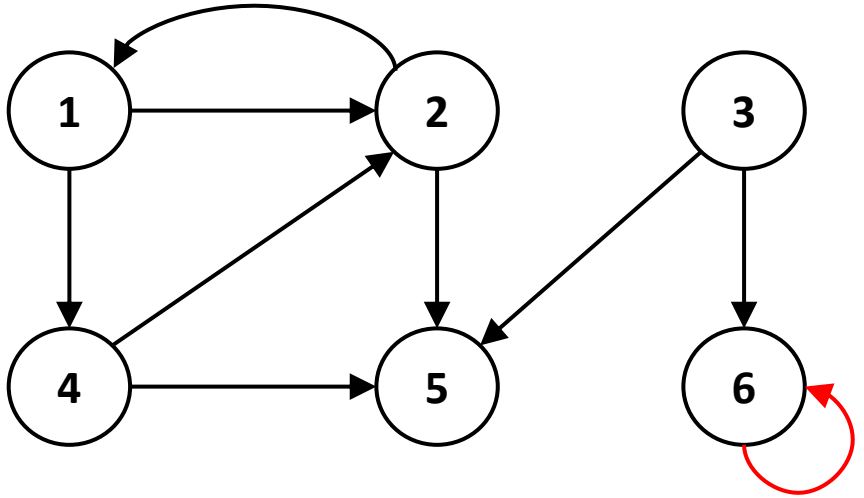
V = {1, 2, 3, 4, 5}

E = { (1, 2), (1, 5), (2, 5), (2, 4),
(3, 2), (3, 4), (4, 5) }

Directed graph:

Each edge is an **ordered** pair (u,v)
i.e. (u, v) ≠ (v, u)

Example:



V = {1, 2, 3, 4, 5, 6}

E = { (1, 2), (1, 4), (2, 1), (2, 5),
(3, 5), (3, 6), (4, 2), (4, 5), (6, 6) }

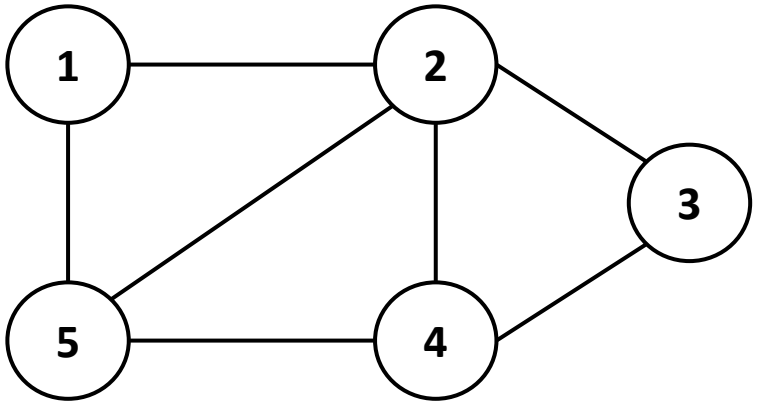
Representing graphs

Representing graphs

Adjacency List of G = (V, E):

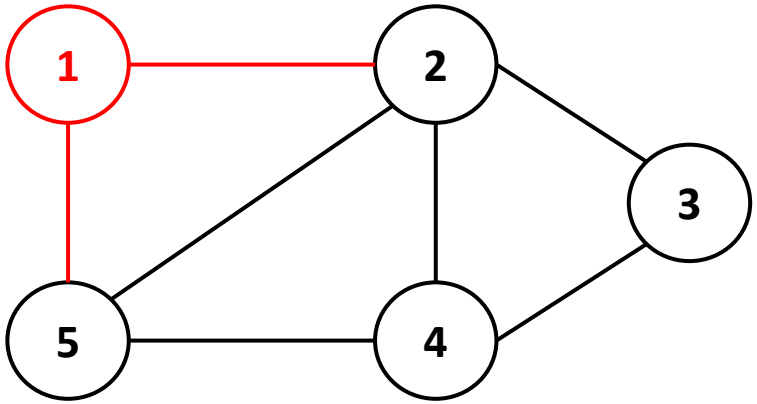
Array **Adj**[1…n] of n lists, one for each node u ∈ V,

Adj[u] : list of all nodes v such that (u,v) ∈ E

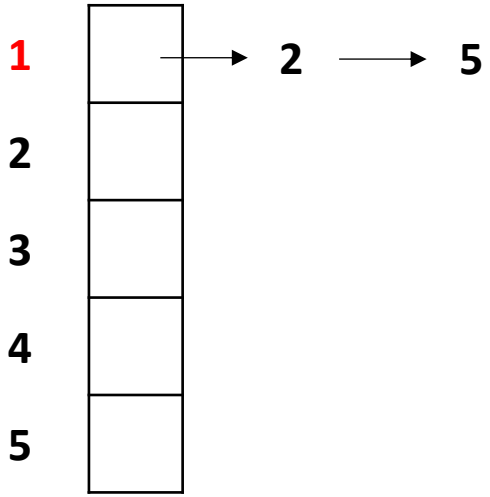


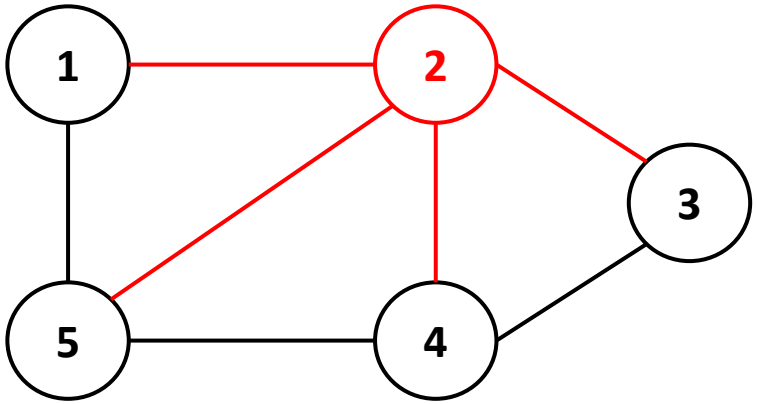
Adjacency list

1	
2	
3	
4	
5	

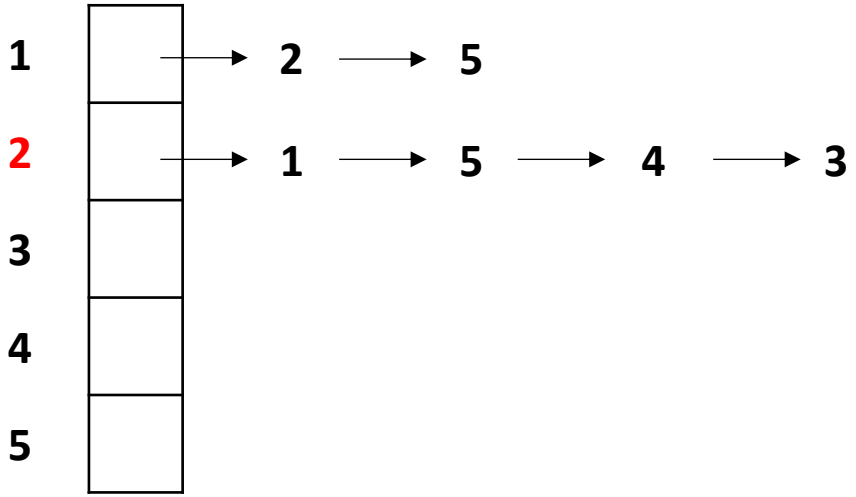


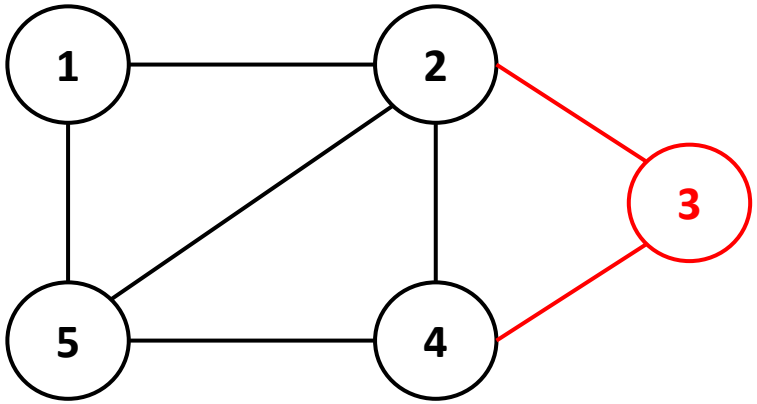
Adjacency list



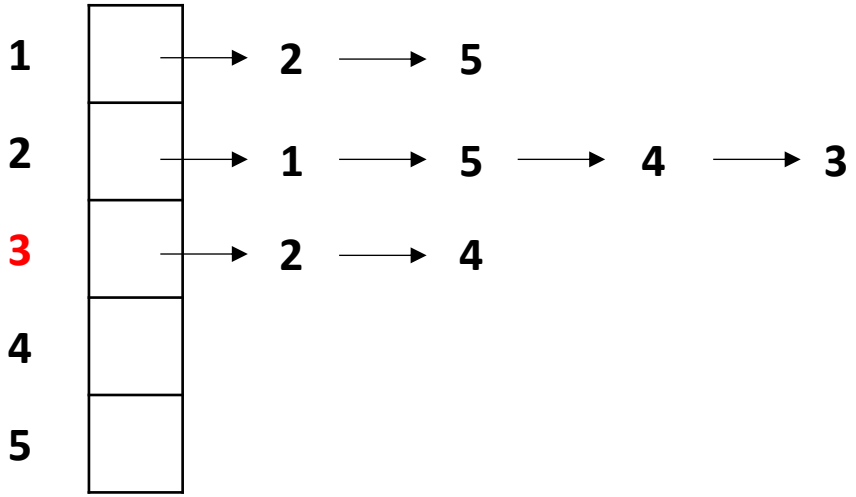


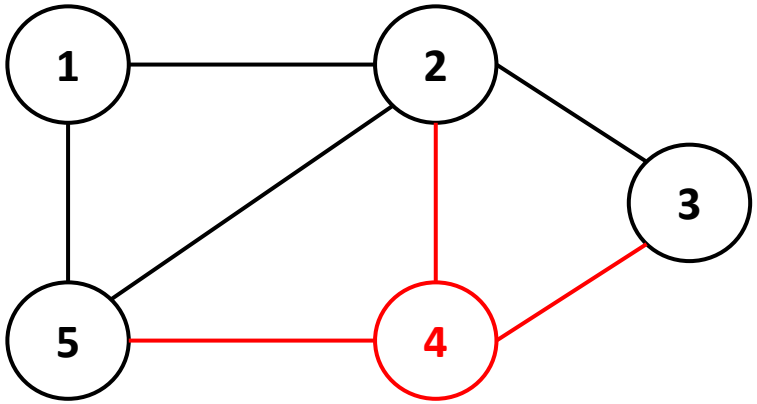
Adjacency list



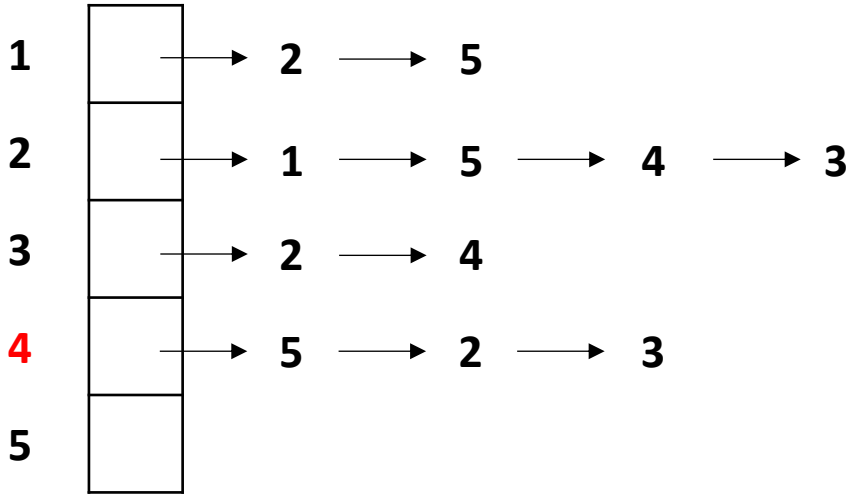


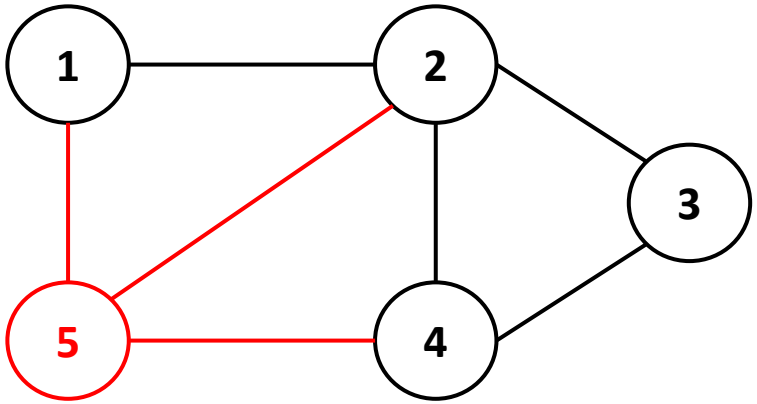
Adjacency list



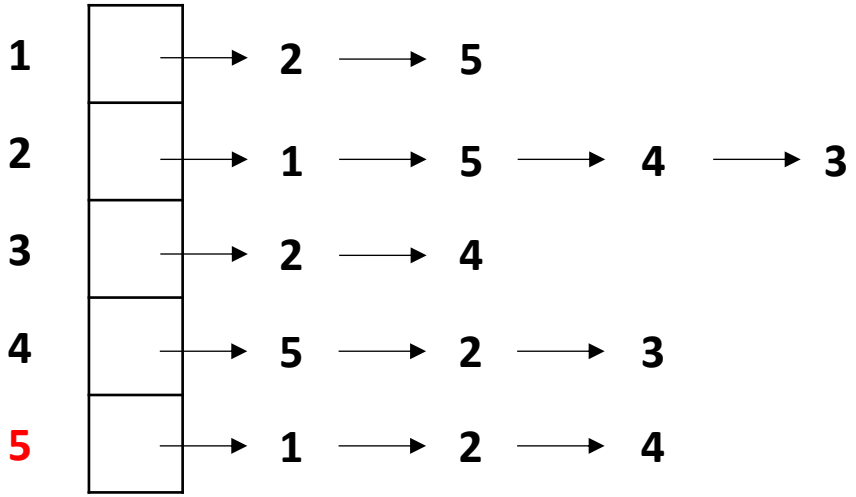


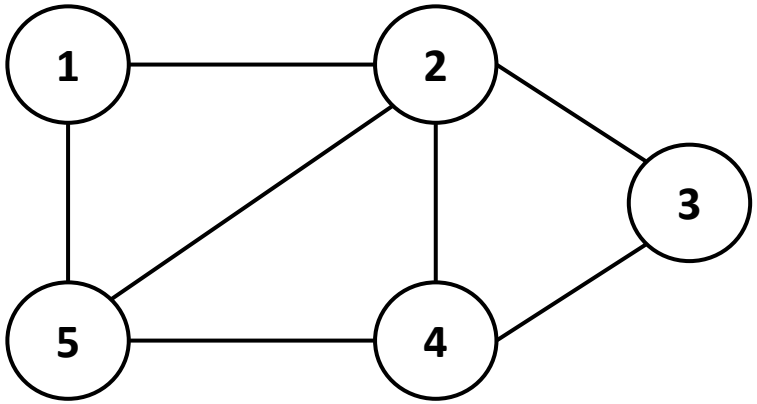
Adjacency list



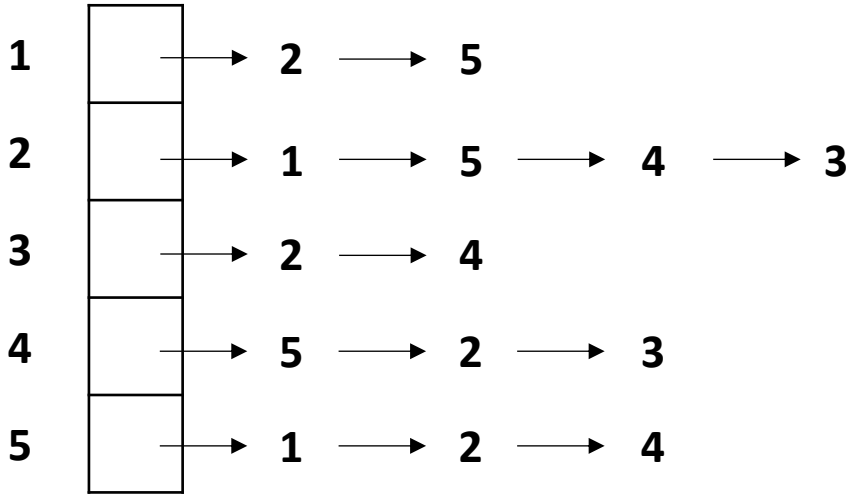


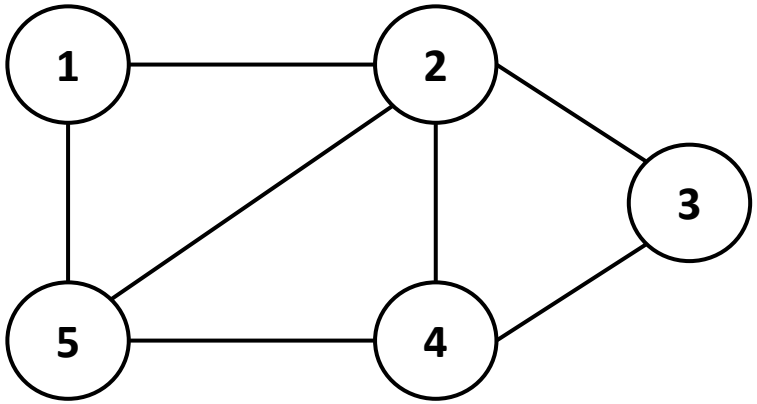
Adjacency list



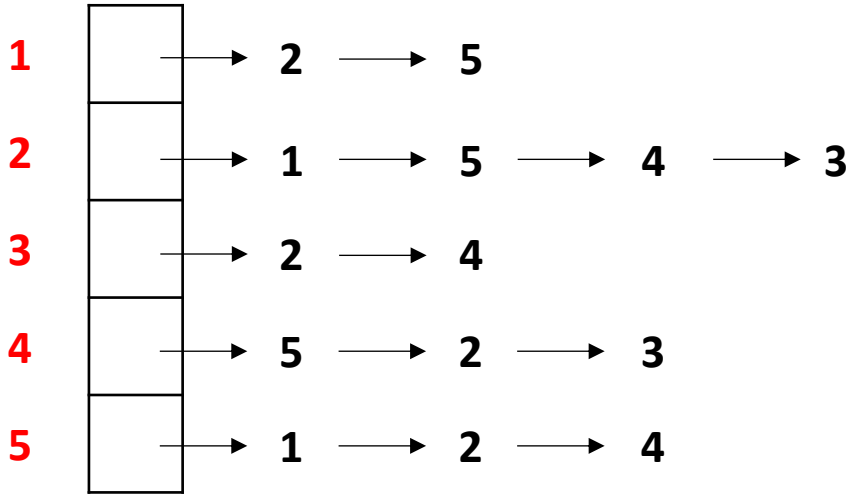


Adjacency list



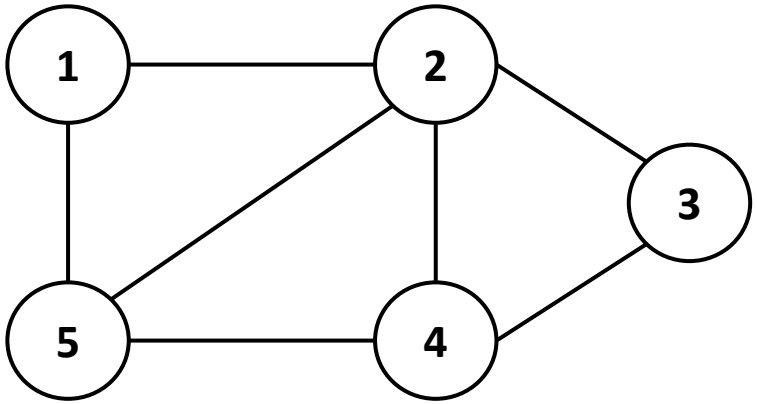


Adjacency list

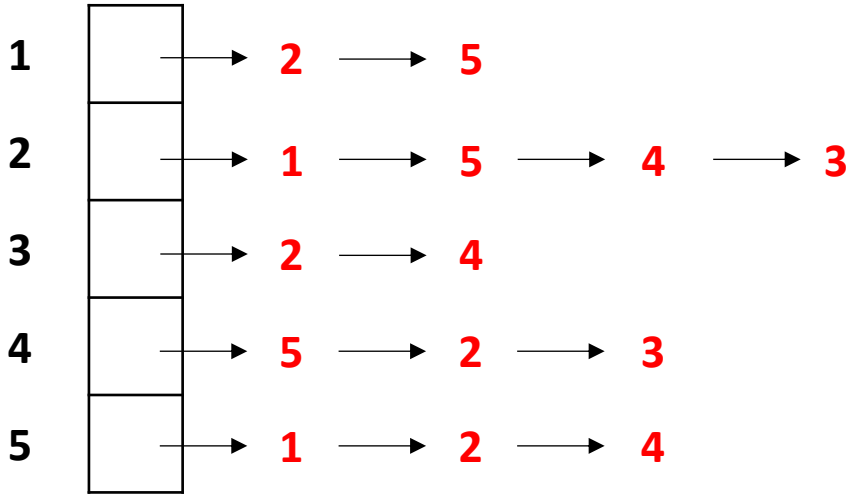


number of nodes

Size: Θ (n)

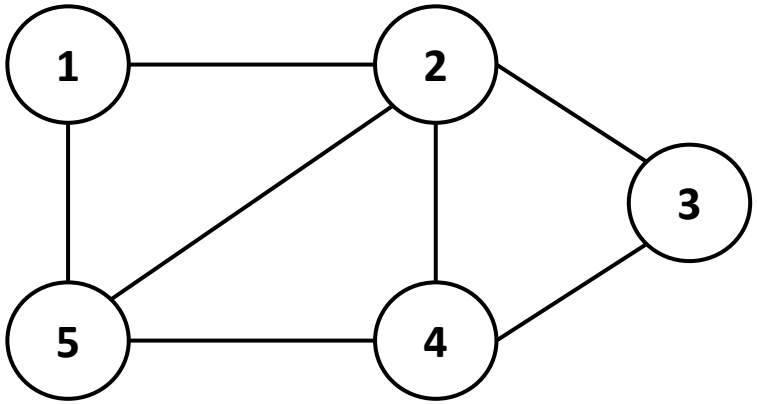


Adjacency list

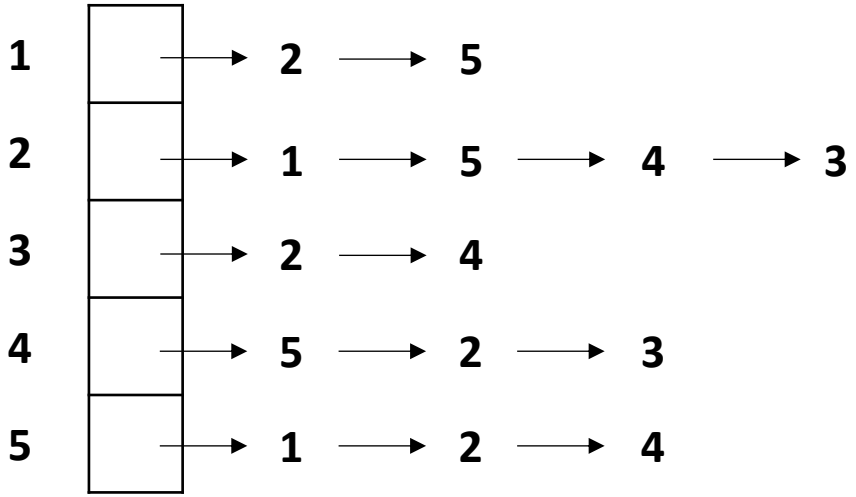


number of nodes number of edges

Size: Θ (n + m)

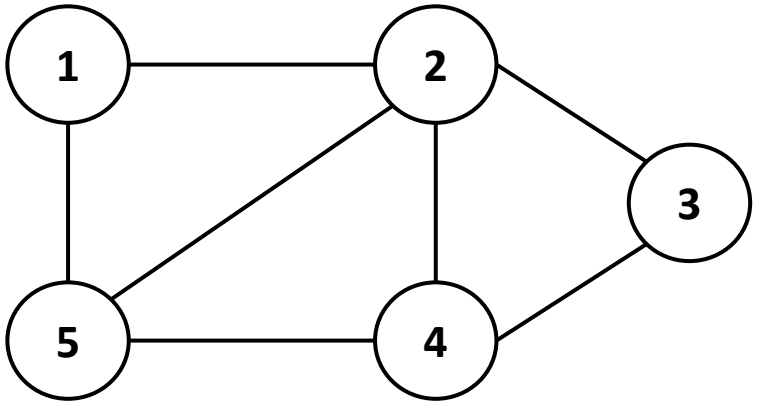


Adjacency list

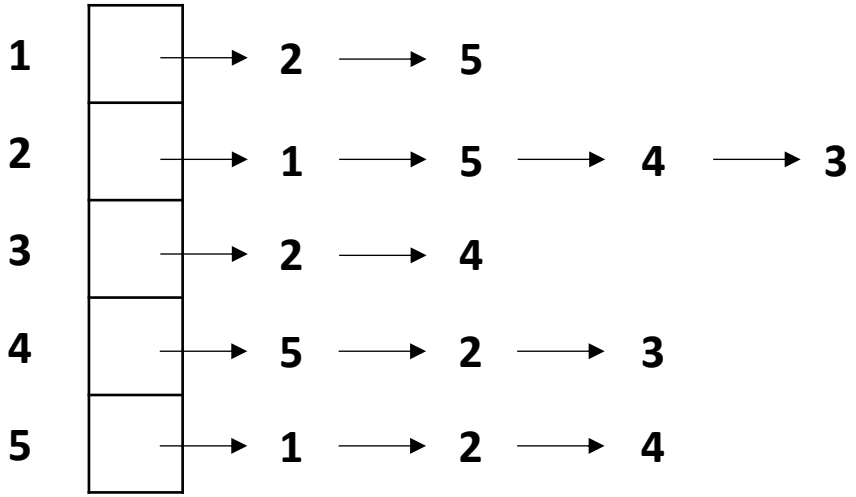


number of nodes number of edges

Size: $\Theta (n + m)$



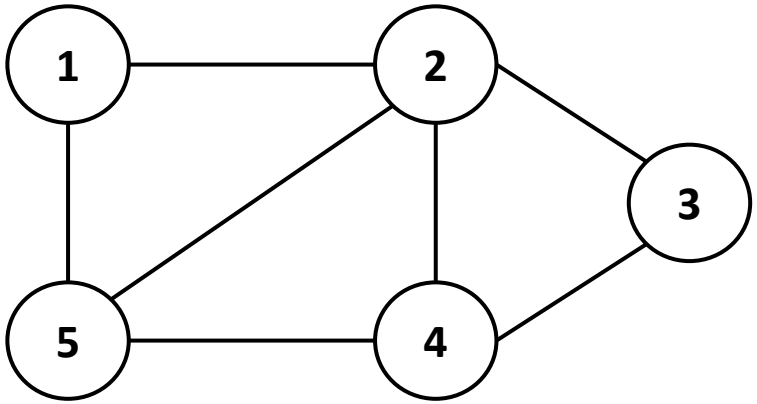
Adjacency list



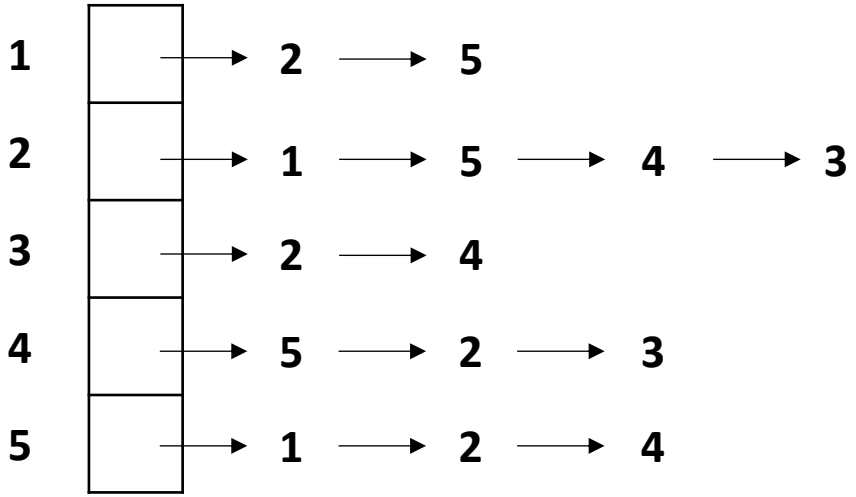
number of nodes number of edges

Size: $\Theta(n + m)$

Note that $m \leq$



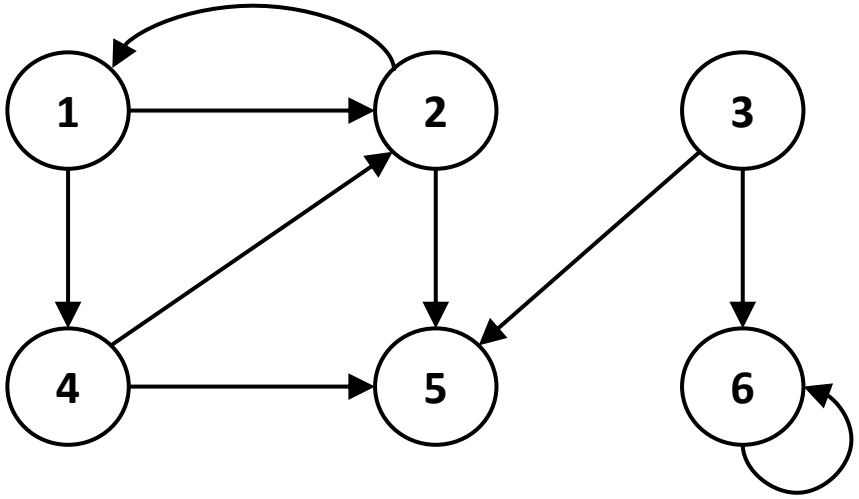
Adjacency list



number of nodes number of edges

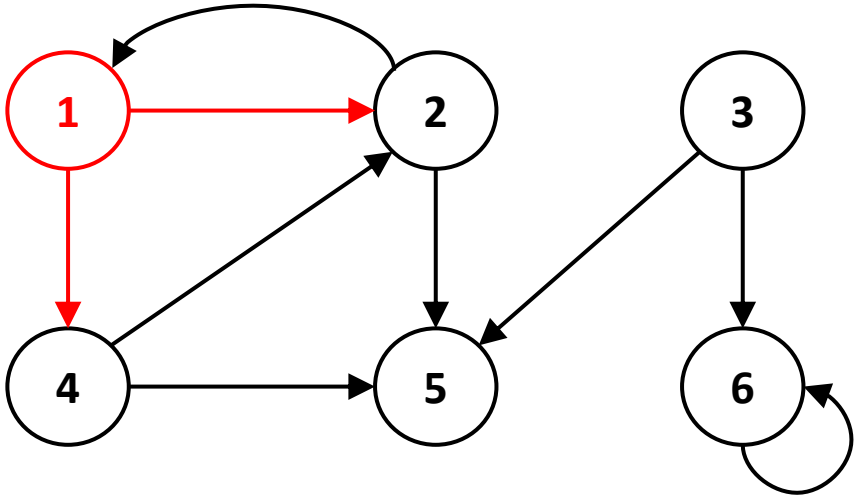
Size: $\Theta (n + m)$

Note that $m \leq n^2$

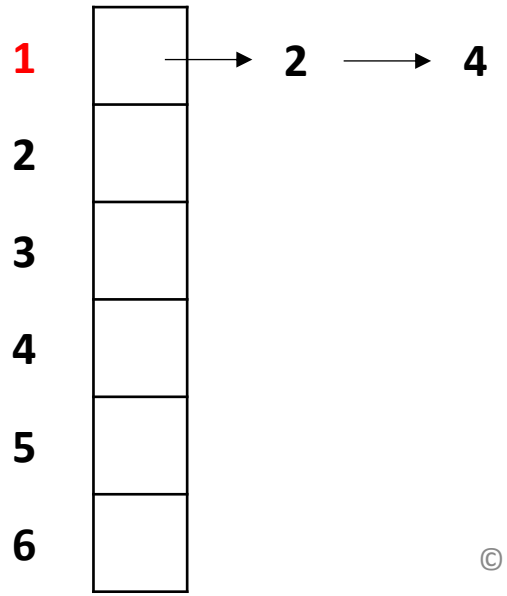


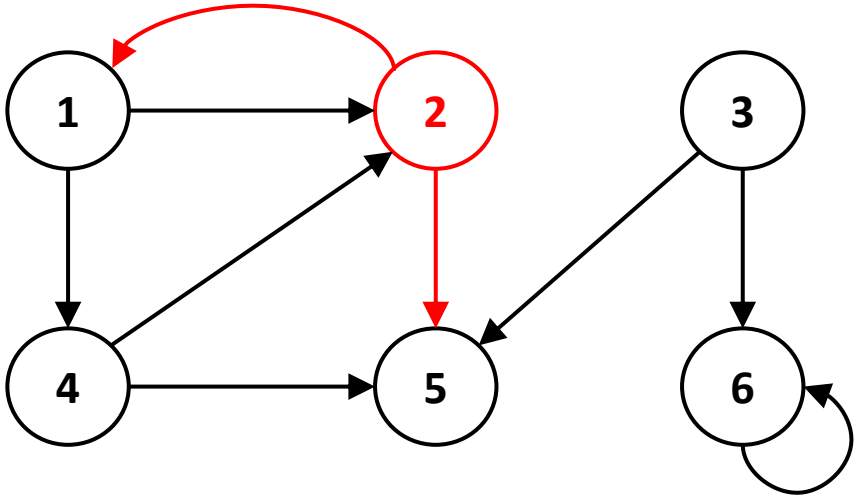
Adjacency list

1	
2	
3	
4	
5	
6	

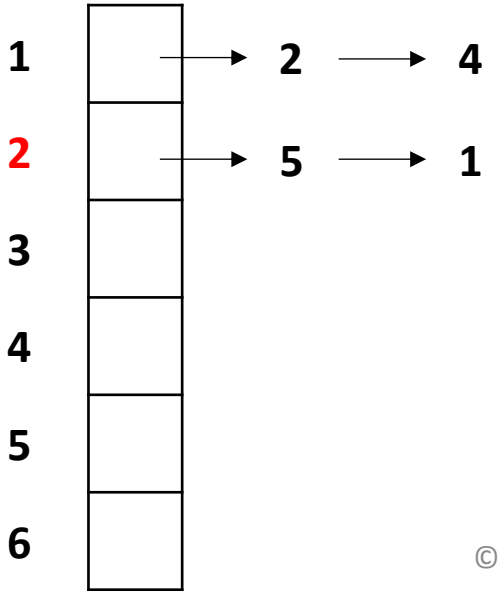


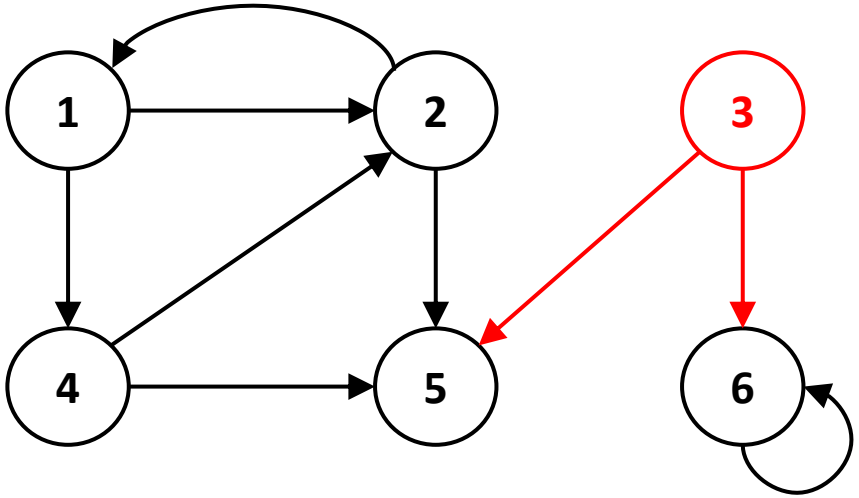
Adjacency list



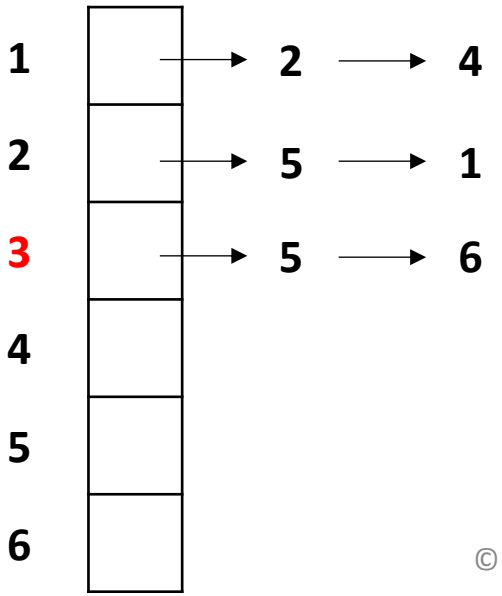


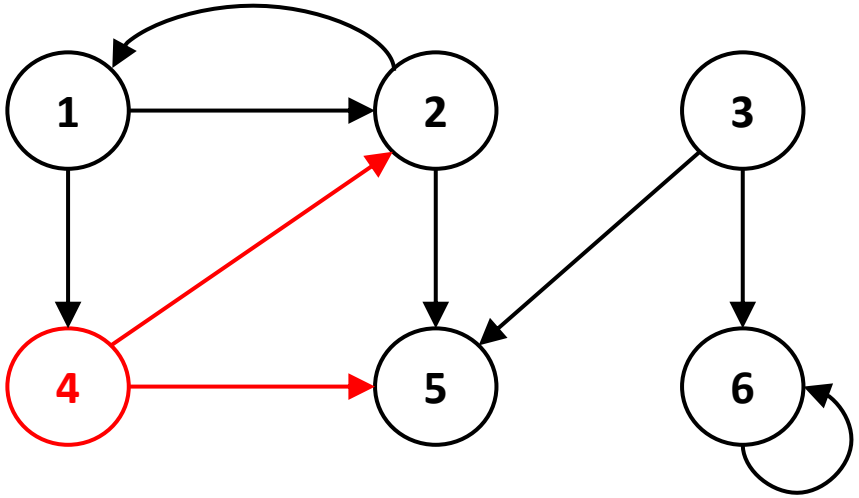
Adjacency list



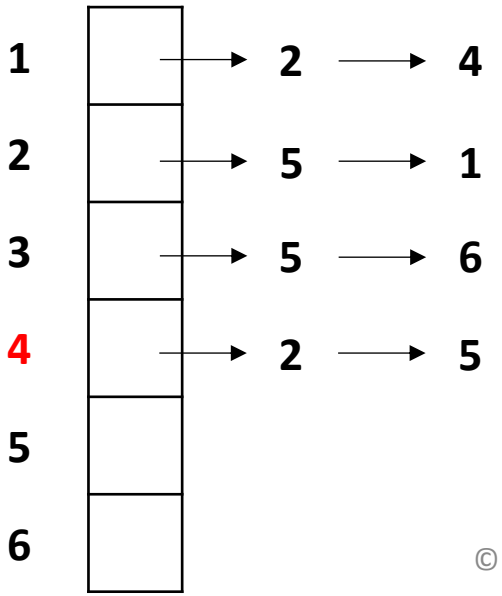


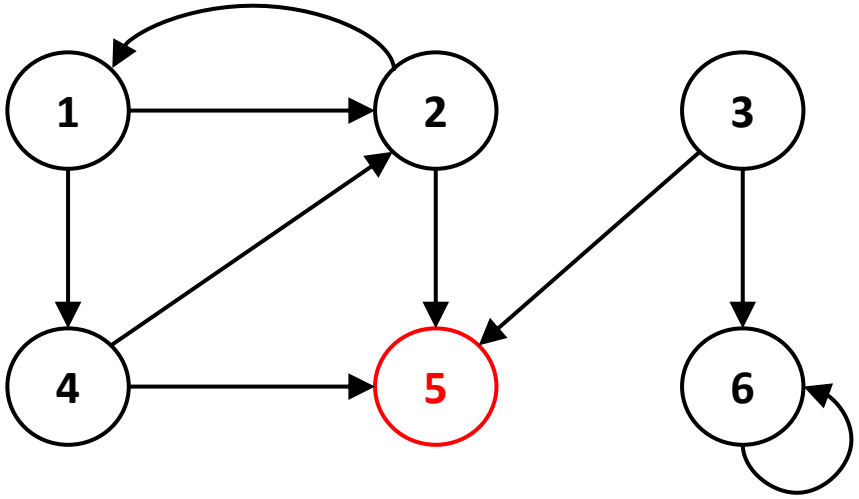
Adjacency list



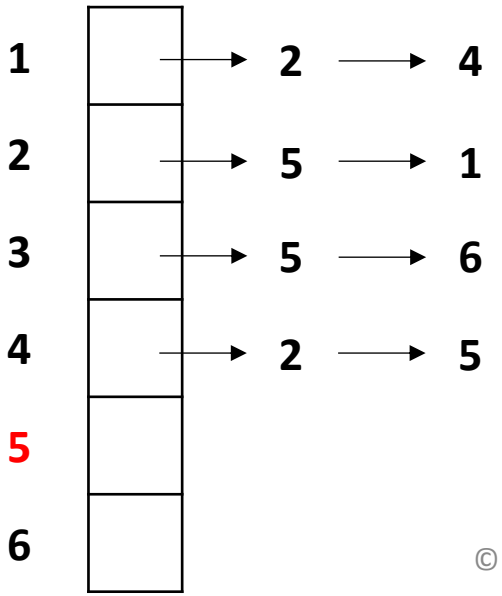


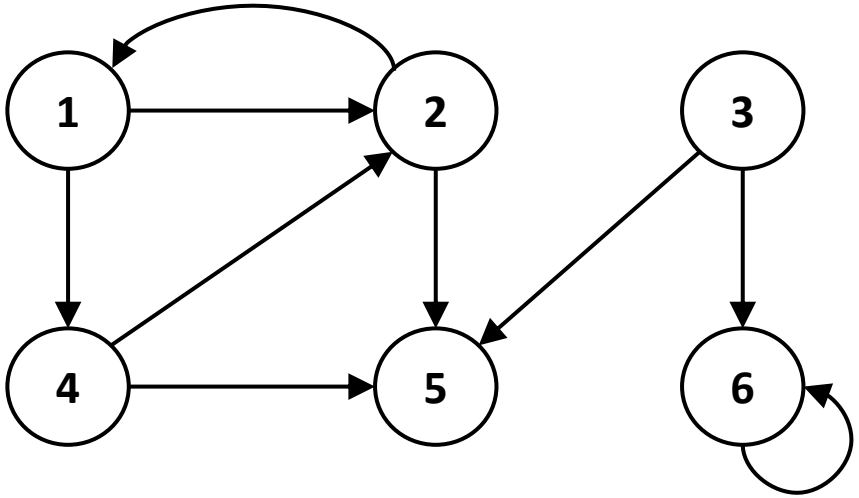
Adjacency list



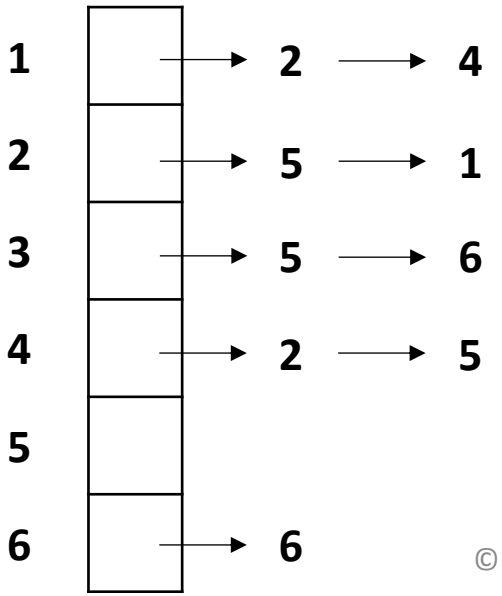


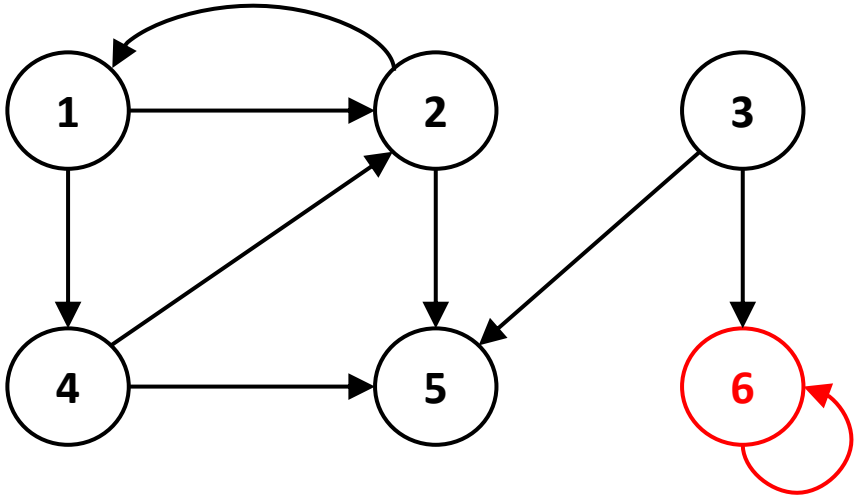
Adjacency list



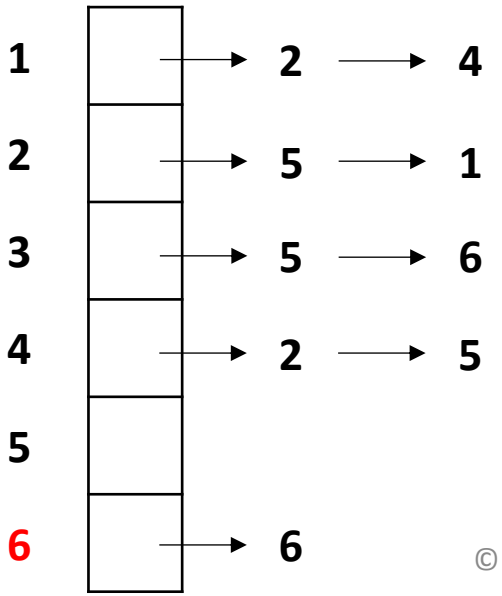


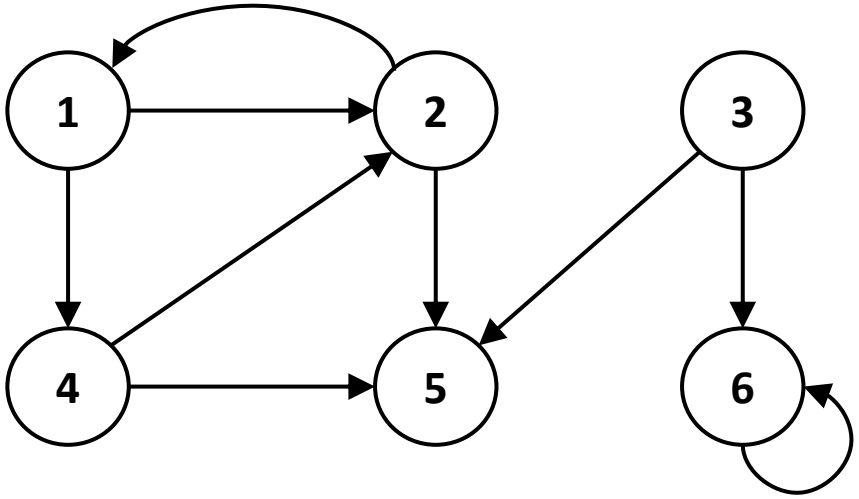
Adjacency list



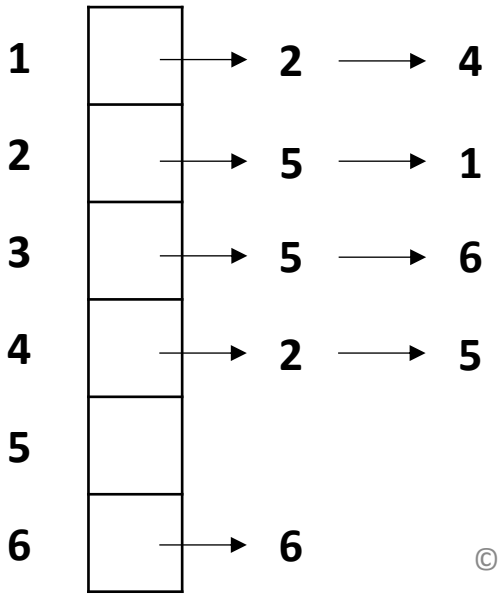


Adjacency list





Adjacency list



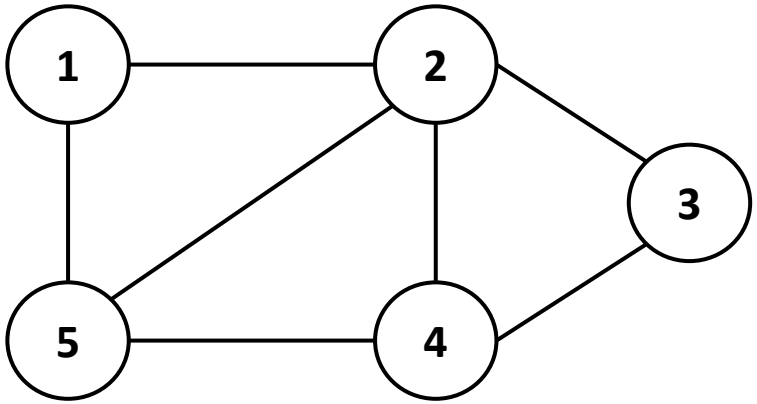
Size: $\Theta (n + m)$

Representing graphs

Adjacency Matrix of G = (V, E):

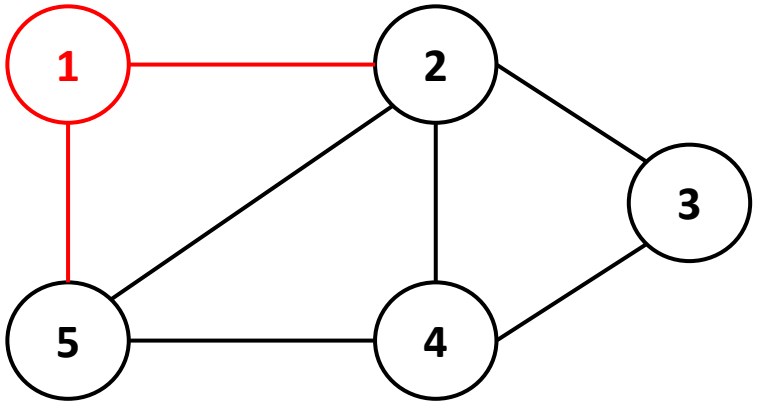
n x n matrix A such that for $1 \leq i,j \leq n$:

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$



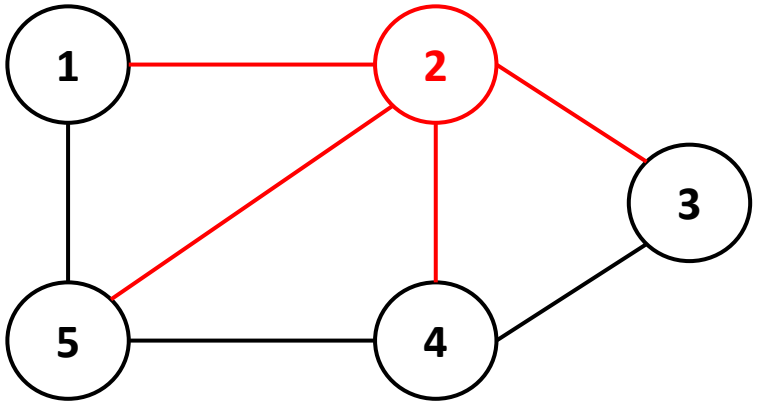
Adjacency matrix

	1	2	3	4	5
1					
2					
3					
4					
5					



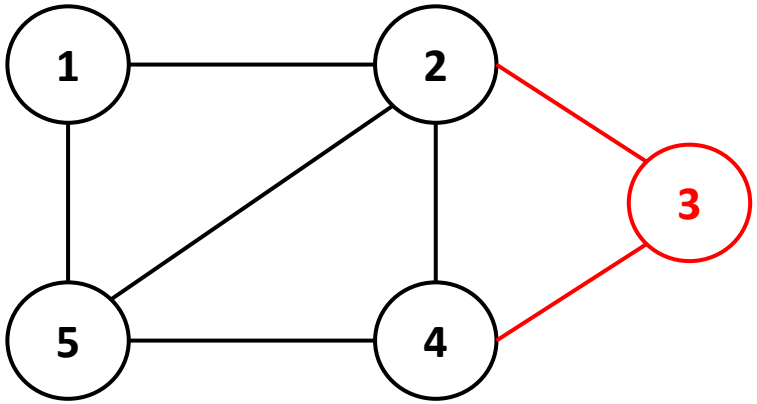
Adjacency matrix

	1	2	3	4	5
1	0	1	0	0	1
2					
3					
4					
5					



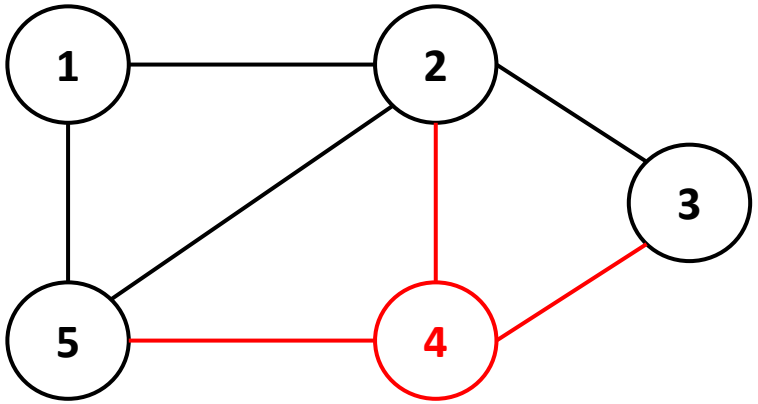
Adjacency matrix

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3					
4					
5					



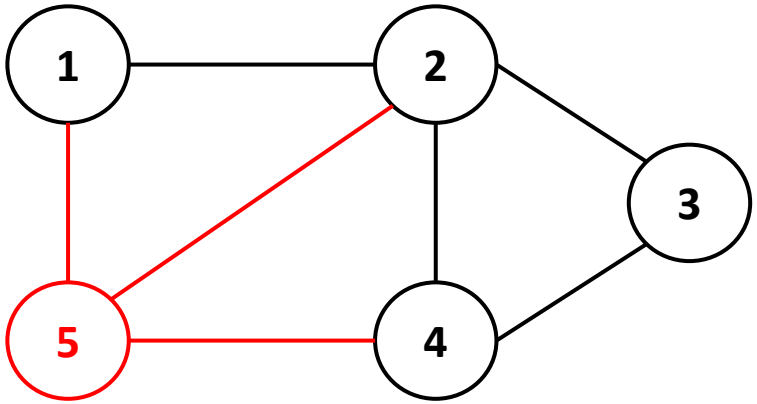
Adjacency matrix

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4					
5					



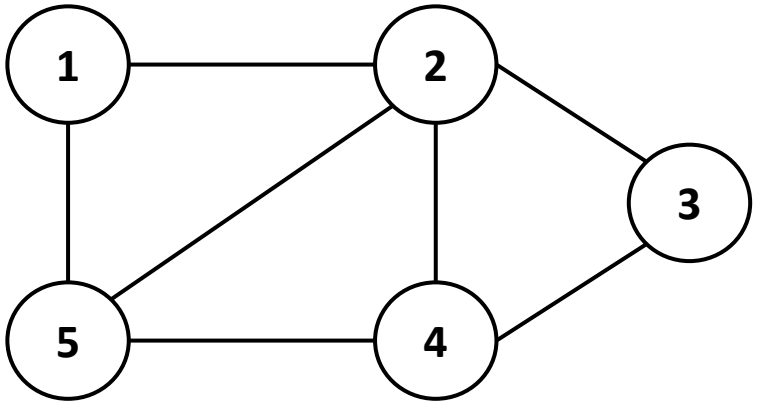
Adjacency matrix

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5					



Adjacency matrix

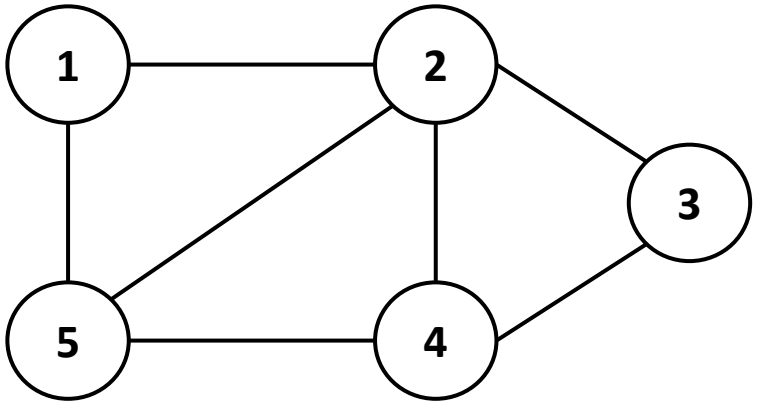
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



Adjacency matrix

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

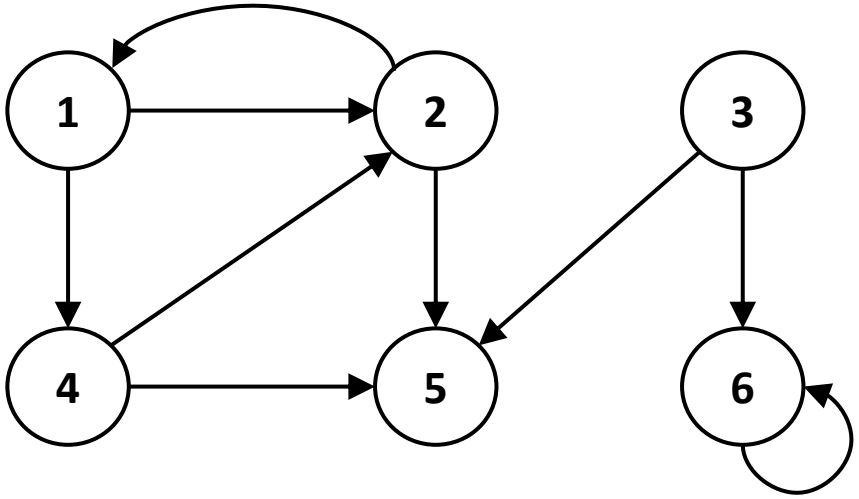
Adjacency matrix of an undirected graph is symmetric about its diagonal



Adjacency matrix

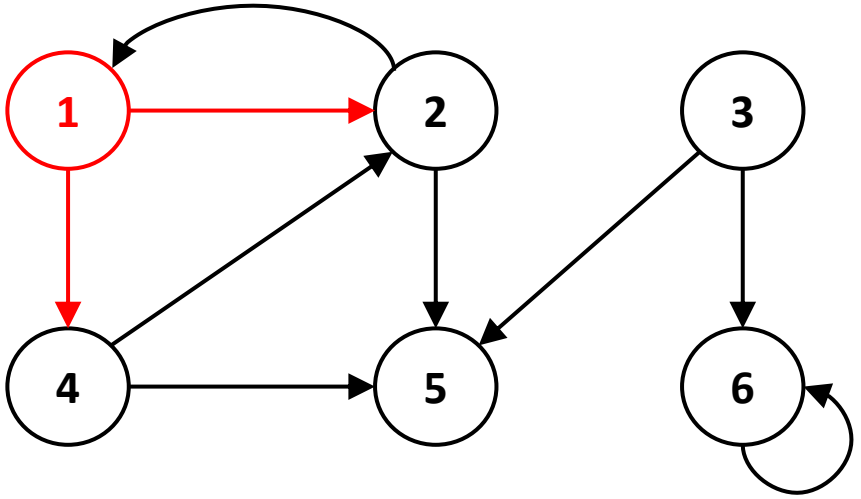
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Size: $O(n^2)$



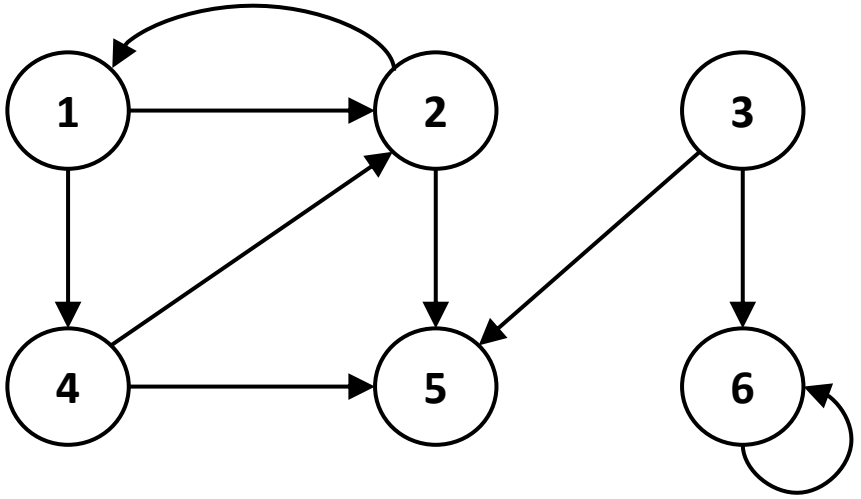
Adjacency matrix

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						



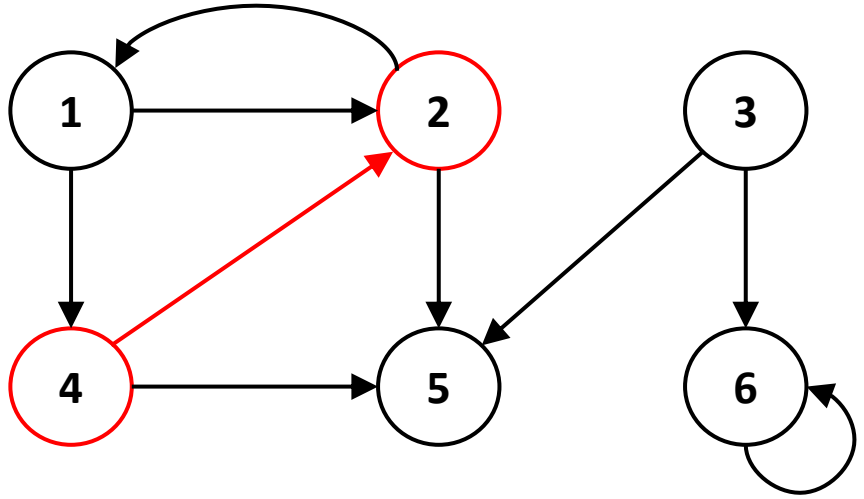
Adjacency matrix

	1	2	3	4	5	6
1	0	1	0	1	0	0
2						
3						
4						
5						
6						



Adjacency matrix

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	1	0
5	0	0	0	0	0	0
6	0	0	0	0	0	1

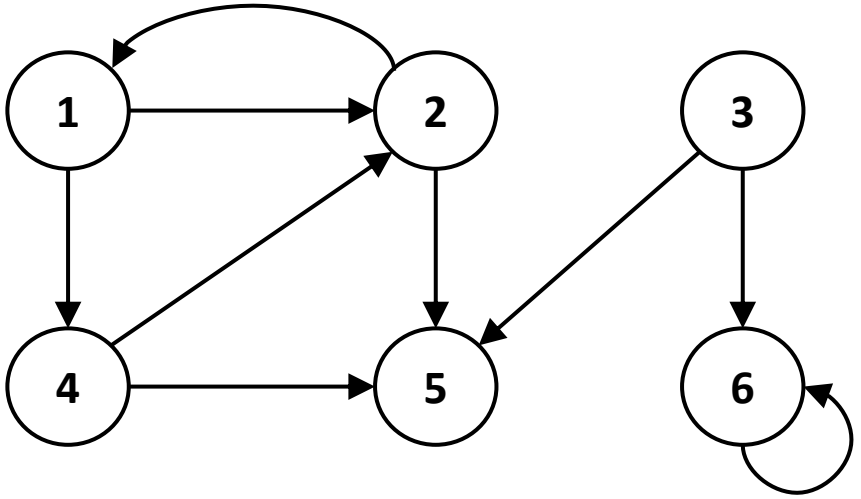


Adjacency matrix

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	**0**	1	0
3	0	0	0	0	1	1
4	0	**1**	0	0	1	0
5	0	0	0	0	0	0
6	0	0	0	0	0	1

A[2, 4] ≠ A[4, 2]

Adjacency matrix of a directed graph is not necessarily symmetric about its diagonal

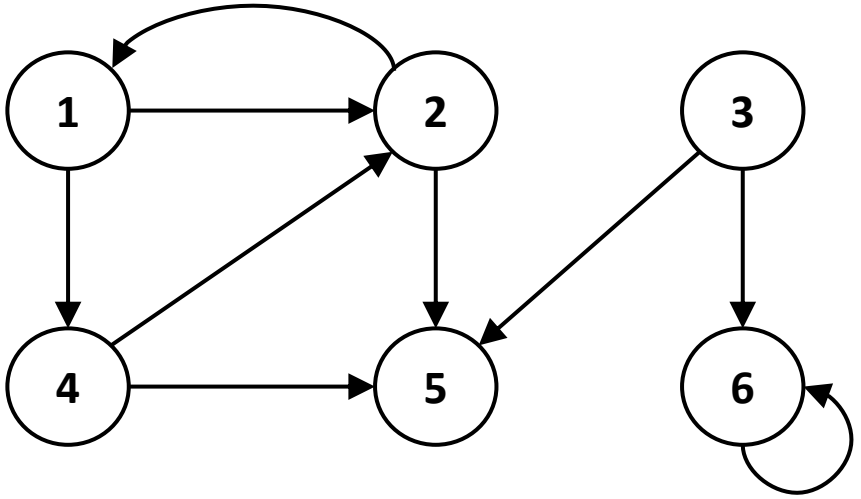


Adjacency matrix

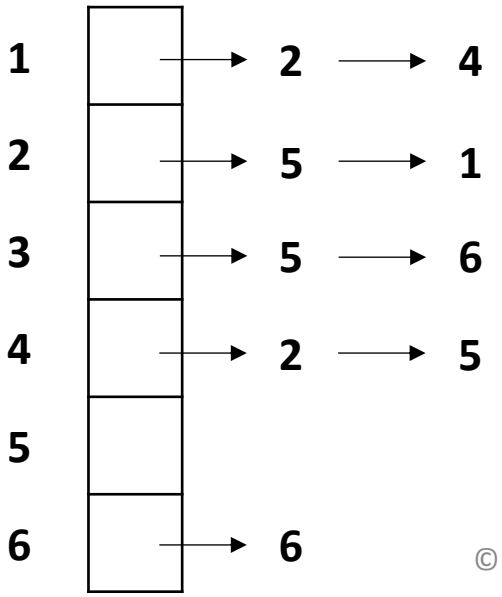
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	1	0
5	0	0	0	0	0	0
6	0	0	0	0	0	1

Size: $\Theta(n^2)$

When to use one representation over the other?

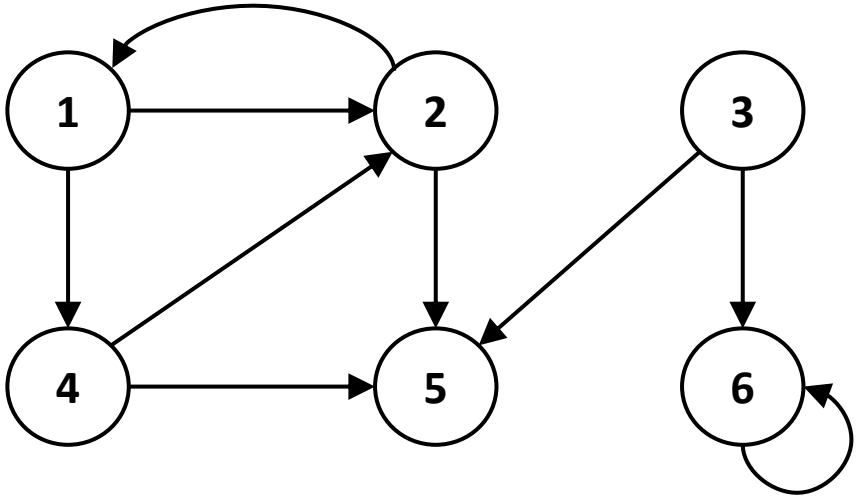


Adjacency list Size: $\Theta (n + m)$

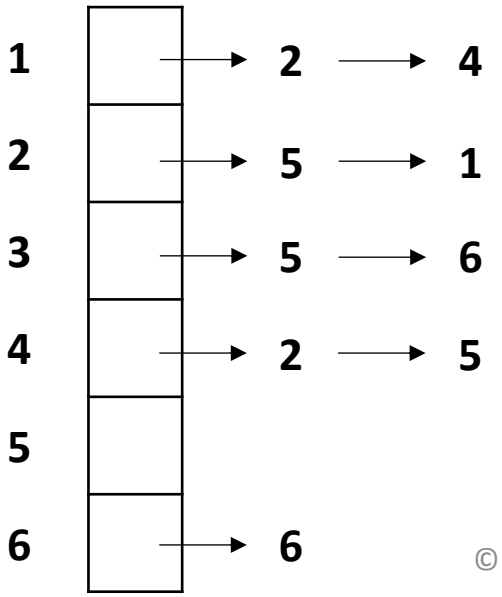


Adjacency matrix Size: $\Theta (n^2)$

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	1	0
5	0	0	0	0	0	0
6	0	0	0	0	0	1



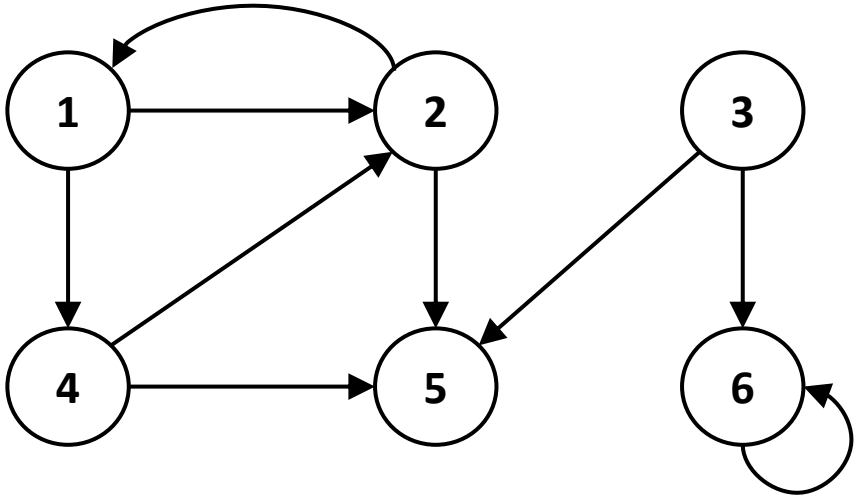
Adjacency list Size: $\Theta(n + m)$



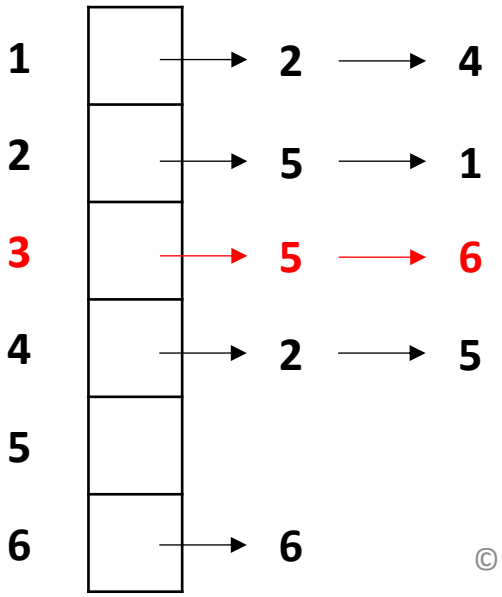
edge (3,6) is in G?

Adjacency matrix Size: $\Theta(n^2)$

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	1	0
5	0	0	0	0	0	0
6	0	0	0	0	0	1



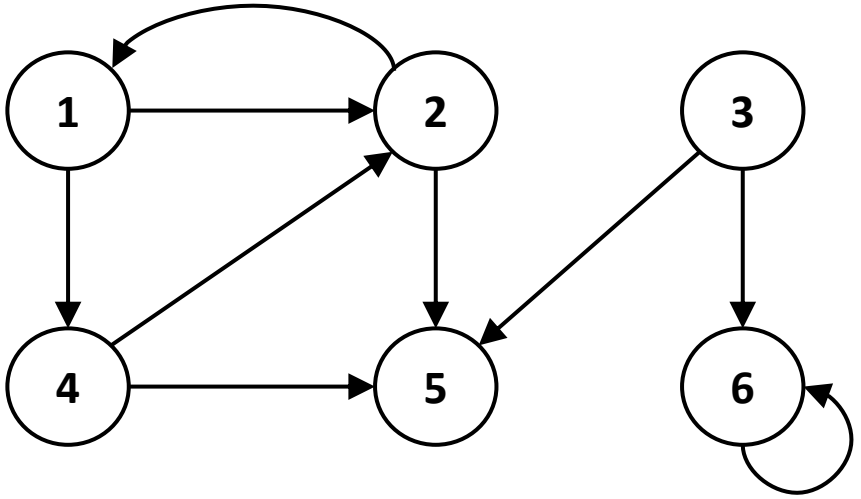
Adjacency list Size: $\Theta(n + m)$



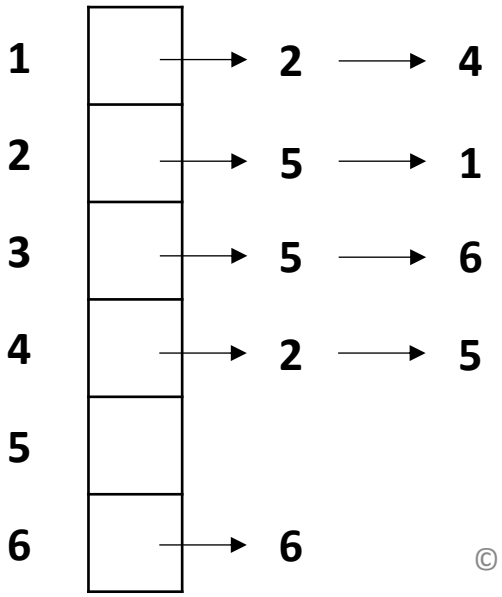
edge (3,6) is in G?

Adjacency matrix Size: $\Theta(n^2)$

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	1	0
5	0	0	0	0	0	0
6	0	0	0	0	0	1



Adjacency list Size: $\Theta(n + m)$



edge (3,6) is in G?

A[3,6] = 1 ?

Adjacency matrix Size: $\Theta(n^2)$

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	**1**
4	0	1	0	0	1	0
5	0	0	0	0	0	0
6	0	0	0	0	0	1

When to use one representation over the other?

- For sparse graphs ($m << n^2$), Adjacency List is more space-efficient.

When to use one representation over the other?

- For sparse graphs ($m << n^2$), Adjacency List is more space-efficient.

- Finding whether a certain $(u, v) \in E$:
  - Is O(1) with Adjacency Matrix
  - Is O(n) with Adjacency List

Graph Search

Systematic exploration of graph:

Graph Search

Systematic exploration of graph:

- Start at some node and explore it

Graph Search

Systematic exploration of graph:

- Start at some node and explore it

  i.e., follow its edges to discover new nodes

Graph Search

Systematic exploration of graph:

- Start at some node and explore it

  i.e., follow its edges to discover new nodes

- Explore newly discovered nodes, i.e., follow edges from those nodes etc…

Graph Search

Systematic exploration of graph:

- Start at some node and explore it

  i.e., follow its edges to discover new nodes

- Explore newly discovered nodes, i.e., follow edges from those nodes etc…

Graph searches reveal structural properties of G:

Graph Search

Systematic exploration of graph:

- Start at some node and explore it

  i.e., follow its edges to discover new nodes

- Explore newly discovered nodes, i.e., follow edges from those nodes etc…

Graph searches reveal structural properties of G:

Is G connected?

Graph Search

Systematic exploration of graph:

- Start at some node and explore it

  i.e., follow its edges to discover new nodes

- Explore newly discovered nodes, i.e., follow edges from those nodes etc…

Graph searches reveal structural properties of G:

Is G connected?

Does G have a cycle?

Graph Search

Systematic exploration of graph:

- Start at some node and explore it

  i.e., follow its edges to discover new nodes

- Explore newly discovered nodes, i.e., follow edges from those nodes etc…

Graph searches reveal structural properties of G:

Is G connected?

Does G have a cycle?

Shortest path info,

etc…

Graph Search

Two basic types of graph searches:

Graph Search

Two basic types of graph searches:

- Breadth First Search (BFS)
- Depth First Search (DFS)

Breadth First Search

Breadth First Search

BFS started at a node s

Breadth First Search

BFS started at a node s

(1) Explore s (i.e. follow all the edges out of s to discover new nodes)

Breadth First Search

BFS started at a node s

(1) Explore s (i.e. follow all the edges out of s to discover new nodes)

(2) Explore discovered nodes in the **order of their discovery**

Breadth First Search

BFS started at a node s

(1) Explore s (i.e. follow all the edges out of s to discover new nodes)

(2) Explore discovered nodes in the **order of their discovery**

["First Discovered-First Explored" policy]

Breadth First Search

BFS started at a node s

(1) Explore s (i.e. follow all the edges out of s to discover new nodes)

(2) Explore discovered nodes in the **order of their discovery**

["First Discovered-First Explored" policy]

To do so, put them in a Queue and explore them in FIFO order

Breadth First Search

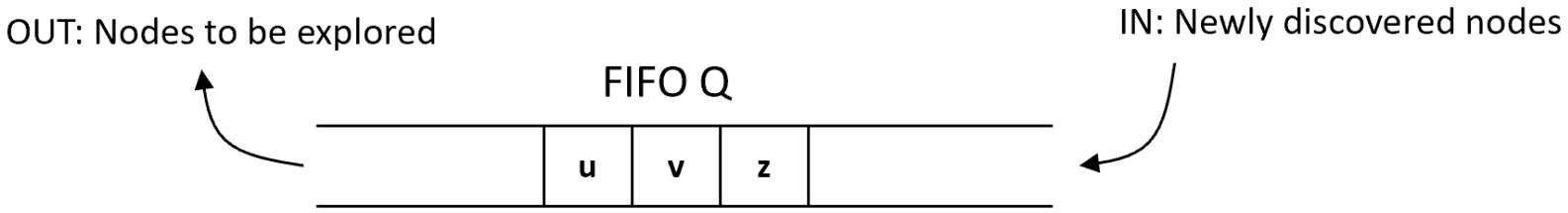
BFS started at a node s

(1) Explore s (i.e. follow all the edges out of s to discover new nodes)

(2) Explore discovered nodes in the **order of their discovery**

["First Discovered-First Explored" policy]

To do so, put them in a Queue and explore them in FIFO order



Breadth First Search

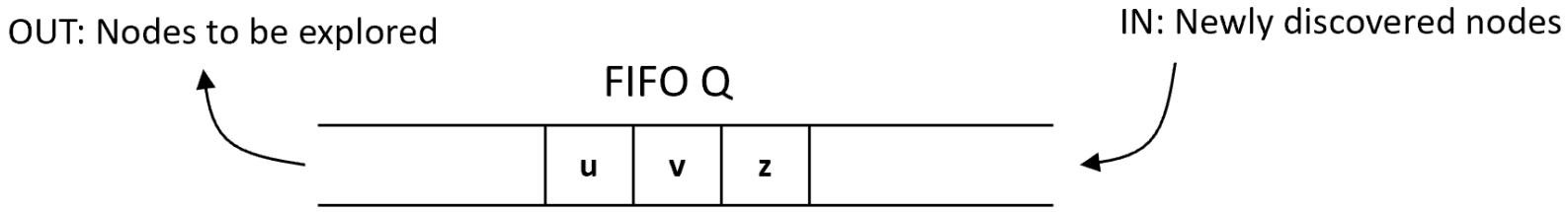
BFS started at a node s

(1) Explore s (i.e. follow all the edges out of s to discover new nodes)

(2) Explore discovered nodes in the **order of their discovery**

["First Discovered-First Explored" policy]

To do so, put them in a Queue and explore them in FIFO order



Do (2) until all discovered nodes are explored

BFS algorithm maintains the following information for each node v:

BFS algorithm maintains the following information for each node v:

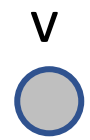
1. color[v] =

BFS algorithm maintains the following information for each node v:

1. color[v] = white : v is undiscovered

BFS algorithm maintains the following information for each node v:

1. color[v] = { white : v is undiscovered
grey : v was discovered but not yet explored



BFS algorithm maintains the following information for each node v:

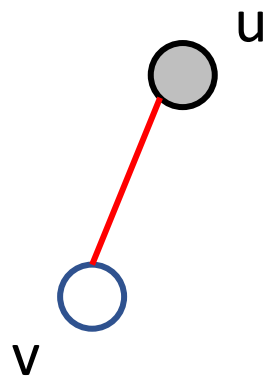
1. color[v] =
   - white : v is undiscovered
   - grey : v was discovered but not yet explored
   - black : v was discovered and explored

BFS algorithm maintains the following information for each node v:

2. p[v]

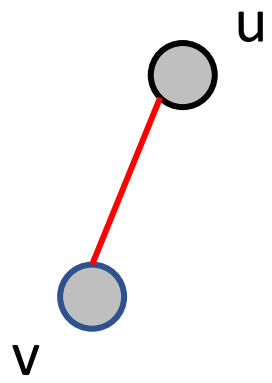
BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u



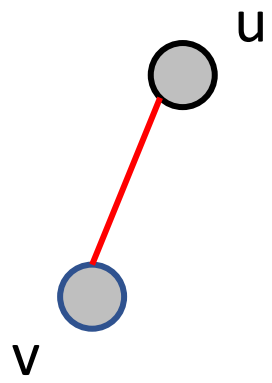
BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u



BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v



BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v]

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s

s

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s

$s \rightarrow u_1$

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s

$s \rightarrow u_1 \rightarrow u_2$

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s

$s \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots$

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s

$$s \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u$$

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s

$$s \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u \rightarrow v$$

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

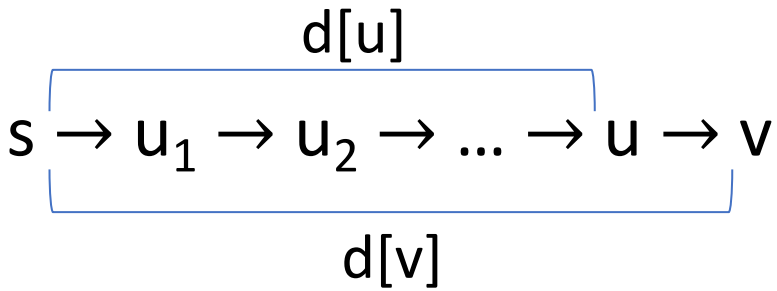
3. d[v] : Length of discovery path from s

$$\underbrace{s \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u \rightarrow v}_{d[v]}$$

BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

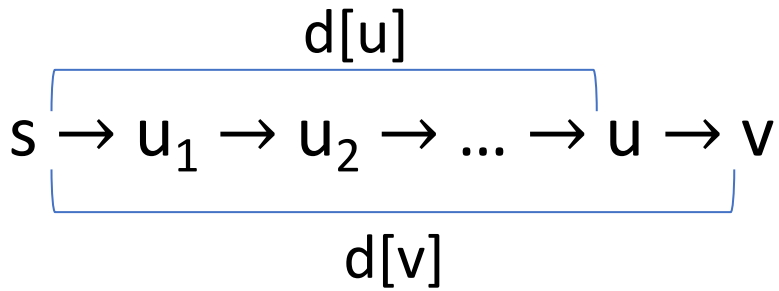
3. d[v] : Length of discovery path from s



BFS algorithm maintains the following information for each node v:

2. p[v] = u : node v was discovered while exploring u
: i.e. u discovered v

3. d[v] : Length of discovery path from s



Clearly, d[v] = d[u] + 1

BFS(G, s**)** /* G = (V, E) and s ∈ V */

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ←

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ←

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ←

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; /* Q: nodes that are discovered but not yet explored */

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

For each (u, v) ∈ E **do** /* Explore edge (u,v) */

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

For each (u, v) ∈ E **do** /* Explore edge (u,v) */

If color[v] = **then do** /* If v is first discovered */

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

For each (u, v) ∈ E **do** /* Explore edge (u,v) */

If color[v] = white **then do** /* If v is first discovered */

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

For each (u, v) ∈ E **do** /* Explore edge (u,v) */

If color[v] = white **then do** /* If v is first discovered */

color[v] ←

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

For each (u, v) ∈ E **do** /* Explore edge (u,v) */

If color[v] = white **then do** /* If v is first discovered */

color[v] ← grey

BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

For each (u, v) ∈ E **do** /* Explore edge (u,v) */

If color[v] = white **then do** /* If v is first discovered */

color[v] ← grey

d[v] ←

```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v]  ←  ∞
            p[v]  ←  NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                      /*  Explore u  */
            For each (u, v) ∈ E  do                         /*  Explore edge (u,v)  */
                    If color[v] = white  then do            /*  If v is first discovered  */
                            color[v] ←  grey
                            d[v] ←  d[u] + 1
```


BFS(G, s**)** /* G = (V, E) and s ∈ V */

color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL

For each v ∈ V – {s} **do**

color[v] ← white

d[v] ← ∞

p[v] ← NIL

Q ← empty ; ENQ(Q, s) /* Q: nodes that are discovered but not yet explored */

While Q is not empty **do**

u ← DEQ(Q) /* Explore u */

For each (u, v) ∈ E **do** /* Explore edge (u,v) */

If color[v] = white **then do** /* If v is first discovered */

color[v] ← grey

d[v] ← d[u] + 1

p[v] ←

```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ;  d[s] ← 0 ;  p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v]  ←  ∞
            p[v]  ←  NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                          /*  Explore u  */
            For each (u, v) ∈ E  do                             /*  Explore edge (u,v)  */
                    If color[v] = white  then do                /*  If v is first discovered  */
                            color[v] ← grey
                            d[v] ← d[u] + 1
                            p[v] ← u
```

```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ;  d[s] ← 0 ;  p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v]  ←  ∞
            p[v]  ←  NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                          /*  Explore u  */
            For each (u, v) ∈ E  do                             /*  Explore edge (u,v)  */
                    If color[v] = white  then do                /*  If v is first discovered  */
                            color[v] ← grey
                            d[v] ← d[u] + 1
                            p[v] ← u
                            ENQ(Q, v)
```

```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ;  d[s] ← 0 ;  p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v]  ←  ∞
            p[v]  ←  NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                          /*  Explore u  */
            For each (u, v) ∈ E  do                             /*  Explore edge (u,v)  */
                    If color[v] = white  then do                /*  If v is first discovered  */
                            color[v] ← grey
                            d[v] ← d[u] + 1
                            p[v] ← u
                            ENQ(Q, v)
                    End If
```

```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ;  d[s] ← 0 ;  p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v]  ←  ∞
            p[v]  ←  NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                          /*  Explore u  */
            For each (u, v) ∈ E  do                             /*  Explore edge (u,v)  */
                    If color[v] = white  then do                /*  If v is first discovered  */
                            color[v] ← grey
                            d[v] ← d[u] + 1
                            p[v] ← u
                            ENQ(Q, v)
                    End If
            End For
```

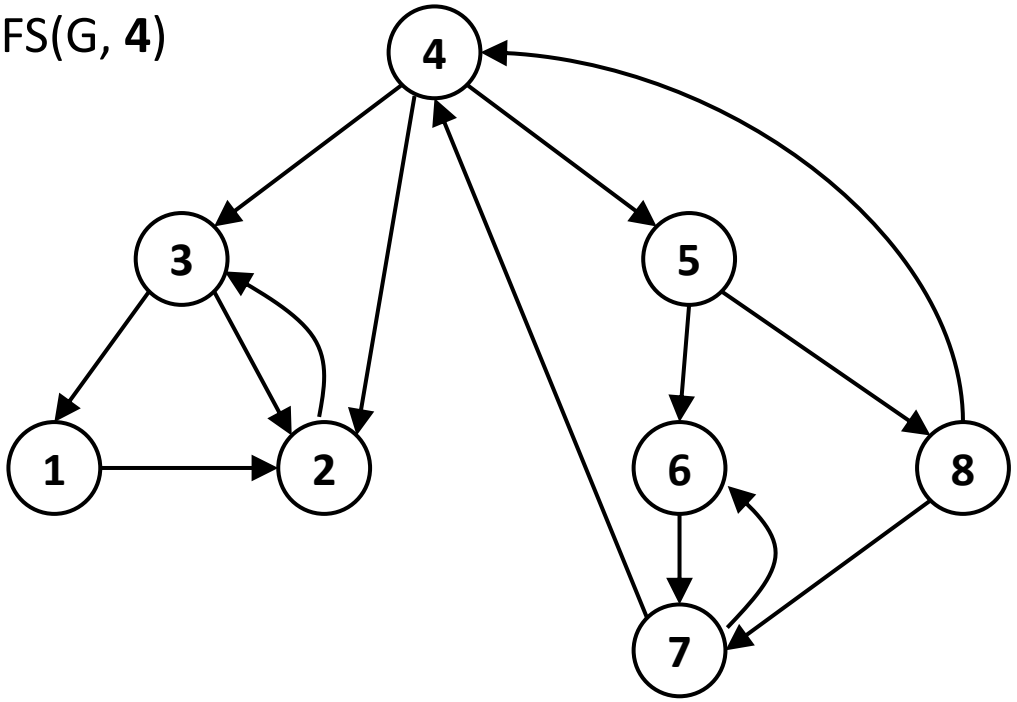
```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v] ← ∞
            p[v] ← NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                          /*  Explore u  */
            For each (u, v) ∈ E  do                             /*  Explore edge (u,v)  */
                    If color[v] = white  then do                /*  If v is first discovered  */
                            color[v] ← grey
                            d[v] ← d[u] + 1
                            p[v] ← u
                            ENQ(Q, v)
                    End If
            End For
            color[u] ←                                          /*  Done exploring u  */
```


```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v] ← ∞
            p[v] ← NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                          /*  Explore u  */
            For each (u, v) ∈ E  do                             /*  Explore edge (u,v)  */
                    If color[v] = white  then do                /*  If v is first discovered  */
                            color[v] ← grey
                            d[v] ← d[u] + 1
                            p[v] ← u
                            ENQ(Q, v)
                    End If
            End For
            color[u] ← black                                    /*  Done exploring u  */
```

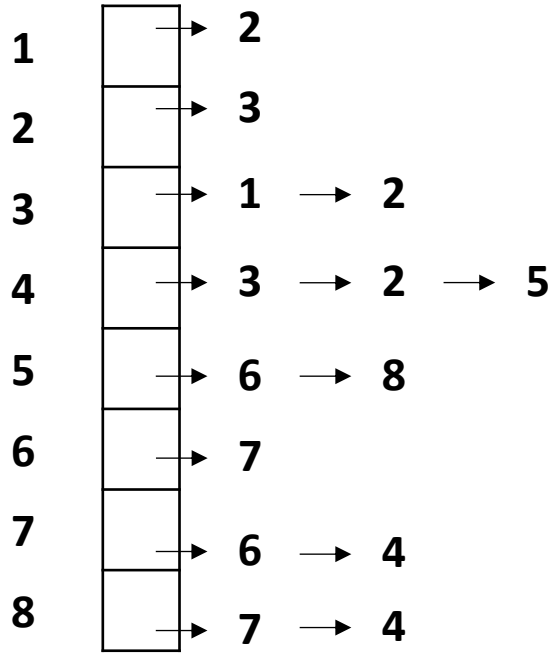

```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v] ← ∞
            p[v] ← NIL
    Q ← empty ; ENQ(Q, s)                          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                          /*  Explore u  */
            For each (u, v) ∈ E  do                             /*  Explore edge (u,v)  */
                    If color[v] = white  then do                /*  If v is first discovered  */
                            color[v] ← grey
                            d[v] ← d[u] + 1
                            p[v] ← u
                            ENQ(Q, v)
                    End If
            End For
            color[u] ← black                                    /*  Done exploring u  */
    End While
End BFS
```


```
BFS(G, s)                                          /*  G = (V, E) and s ∈ V  */
    color[s] ← grey ; d[s] ← 0 ; p[s] ← NIL
    For each v ∈ V – {s}  do
            color[v] ← white
            d[v] ← ∞
            p[v] ← NIL
    Q ← empty ; ENQ(Q, s)          /*  Q: nodes that are discovered but not yet explored  */
    While Q is not empty  do
            u ← DEQ(Q)                                 /*  Explore u  */
        For each (u, v) ∈ E  do                        /*  Explore edge (u,v)  */
            If color[v] = white  then do               /*  If v is first discovered  */
                color[v] ← grey
                d[v] ← d[u] + 1
                p[v] ← u
                ENQ(Q, v)
            End If
        End For
            color[u] ← black                           /*  Done exploring u  */
    End While
End BFS
```


BFS(G, **4**)

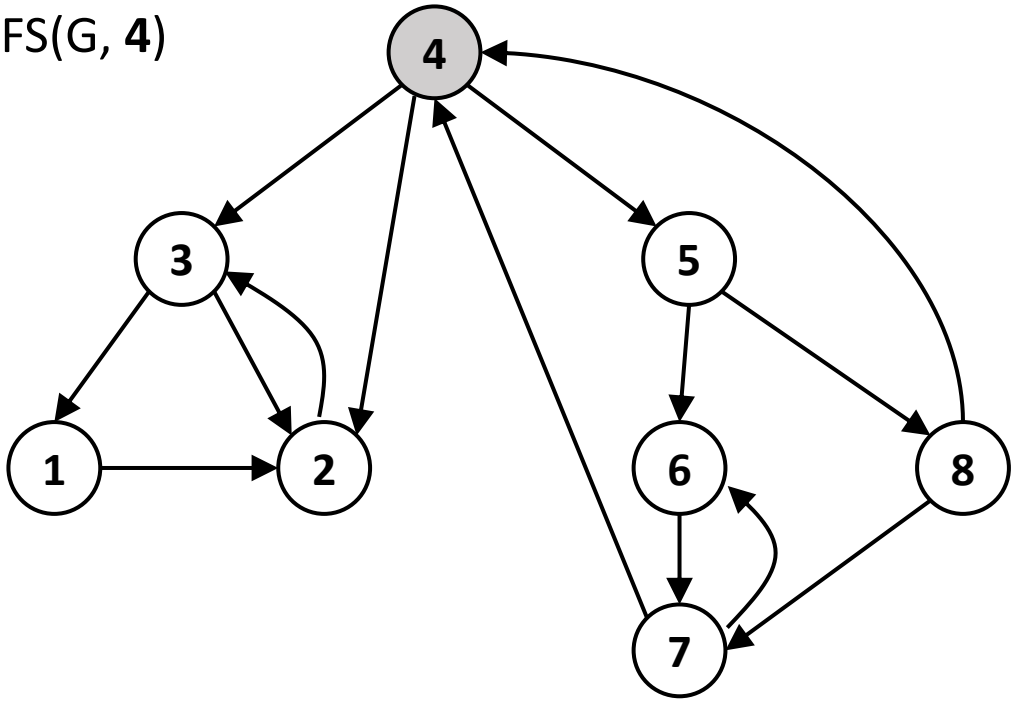


Adj List of G :



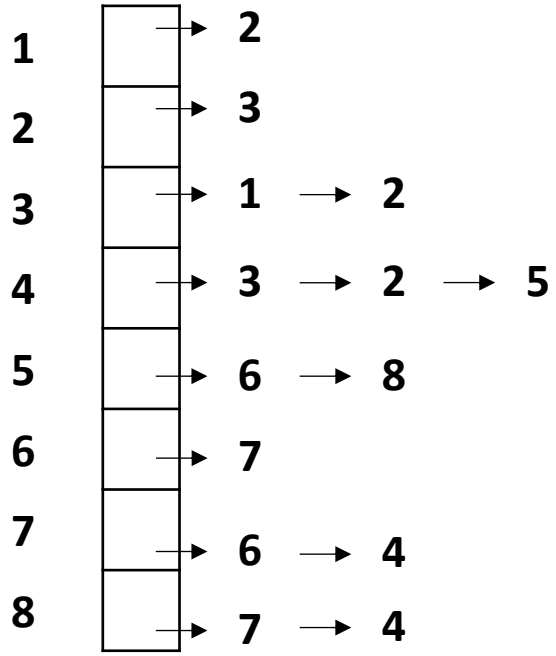
Contents of Q :

BFS(G, **4**)



4

Adj List of G :

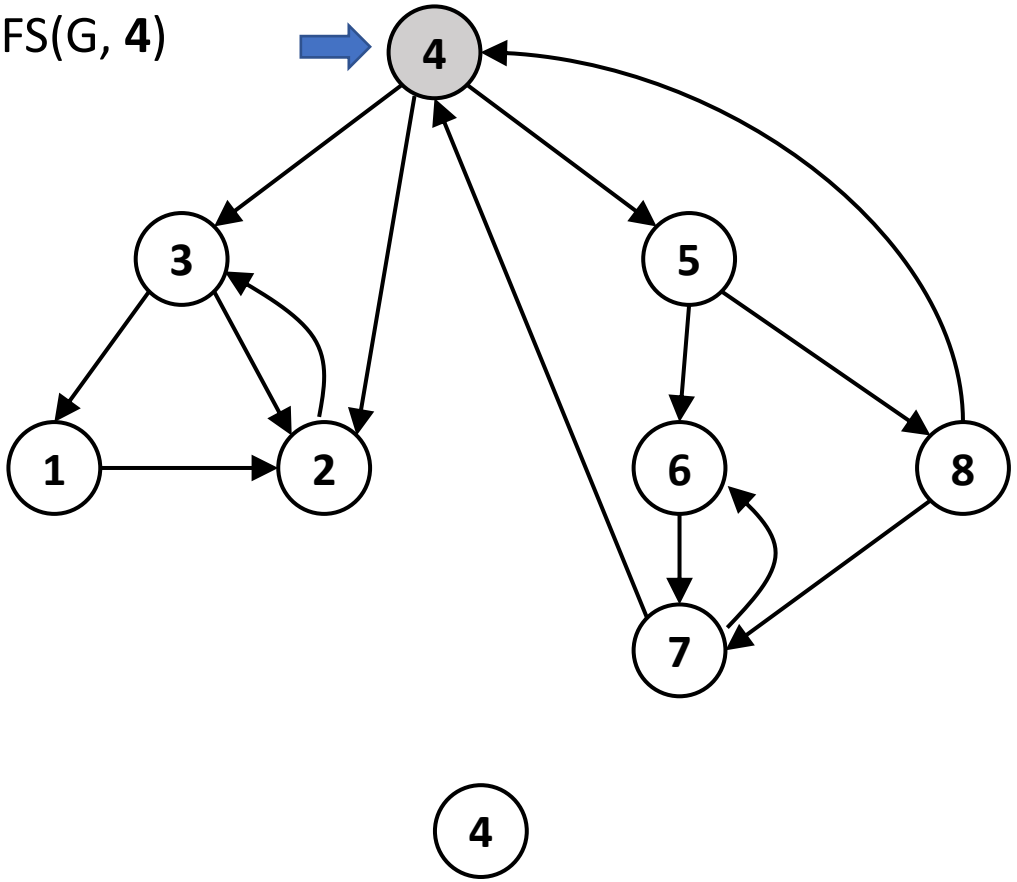


Contents of Q :

d = 0

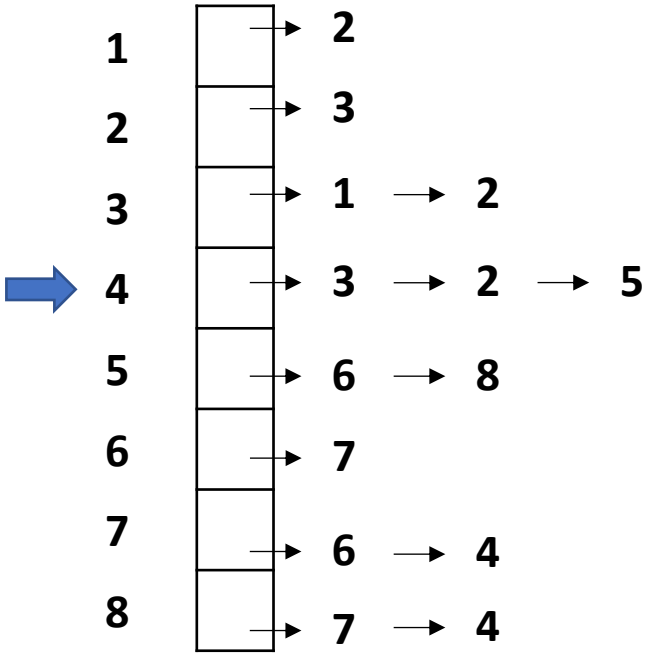
4

BFS(G, **4**)



4

Adj List of G :

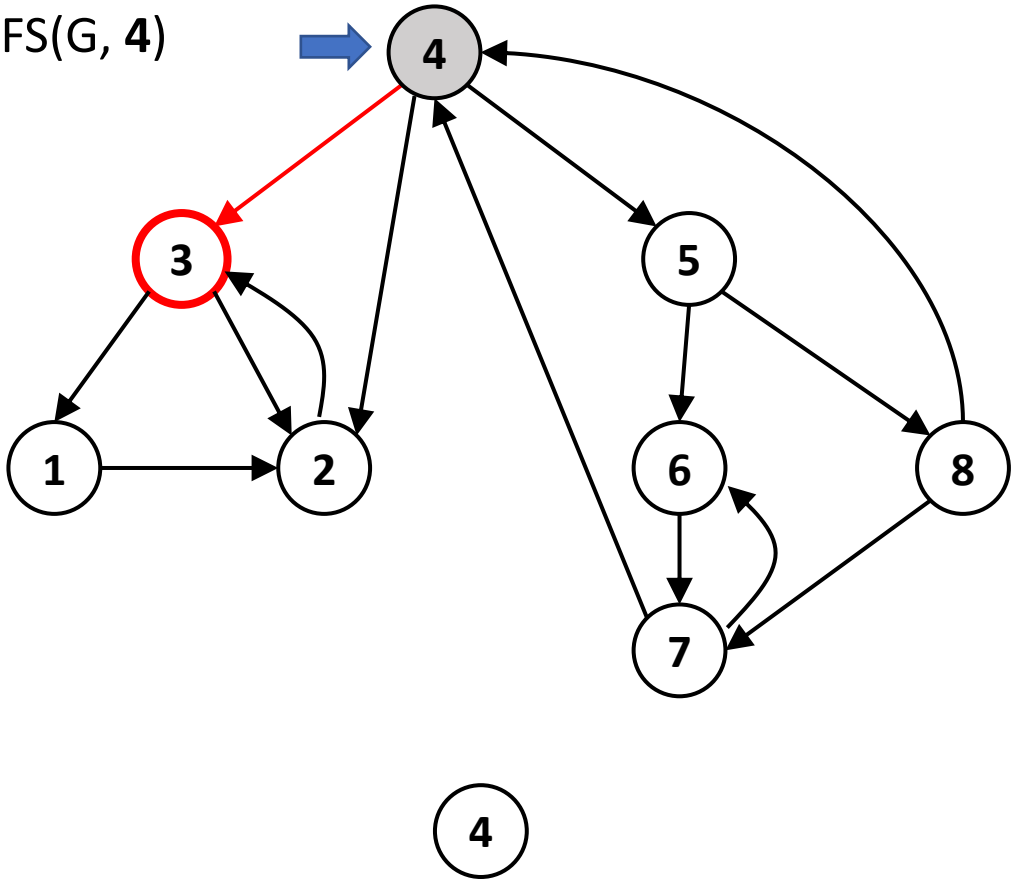


Contents of Q :

d = 0

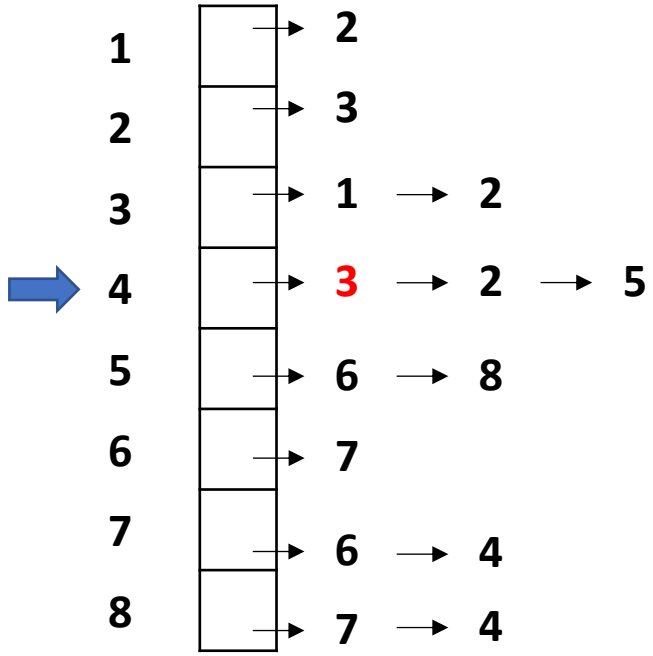
4

BFS(G, **4**)



4

Adj List of G :

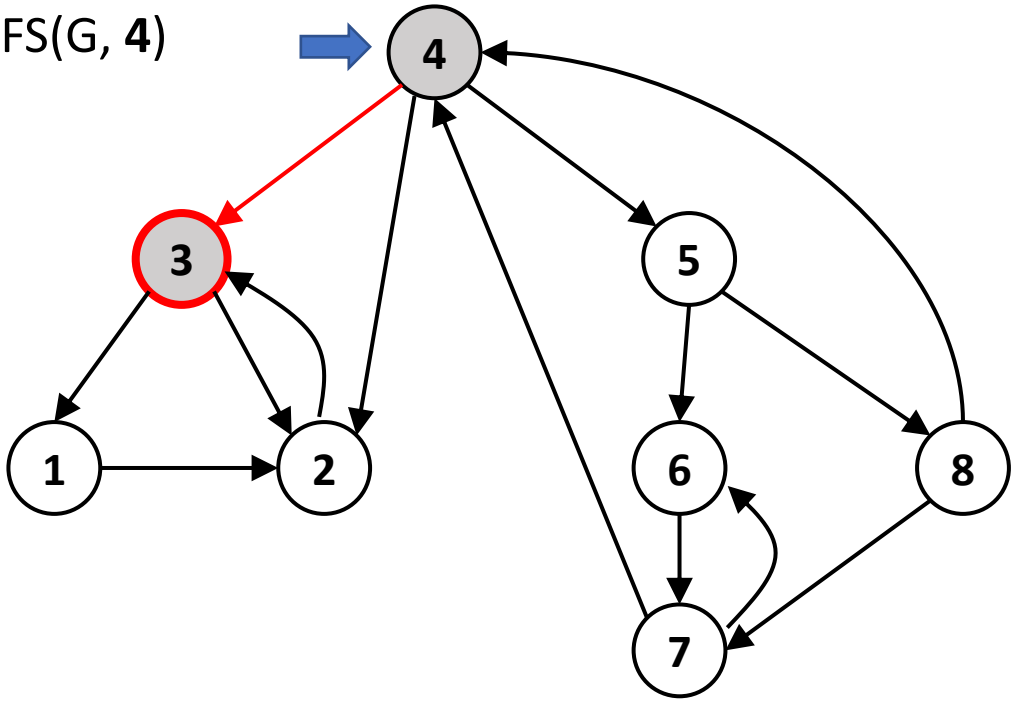


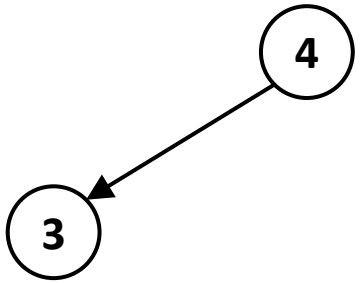
Contents of Q :

d = 0

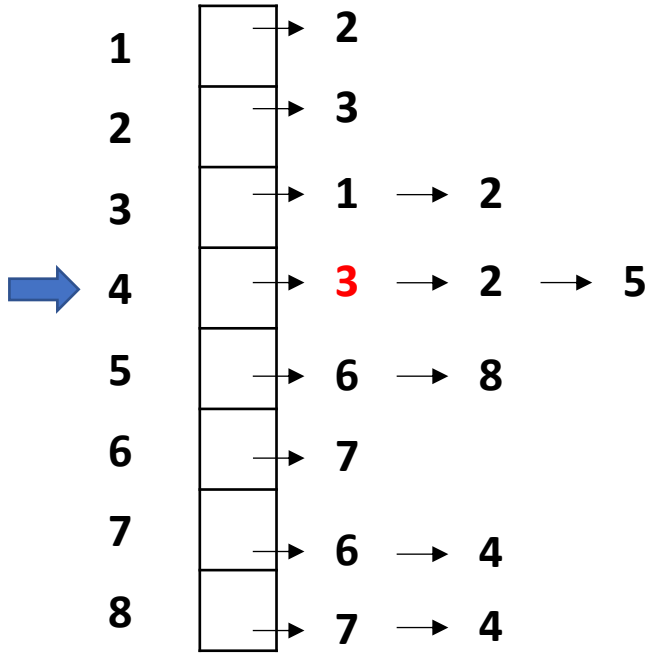
4

BFS(G, **4**)





Adj List of G :



Contents of Q :

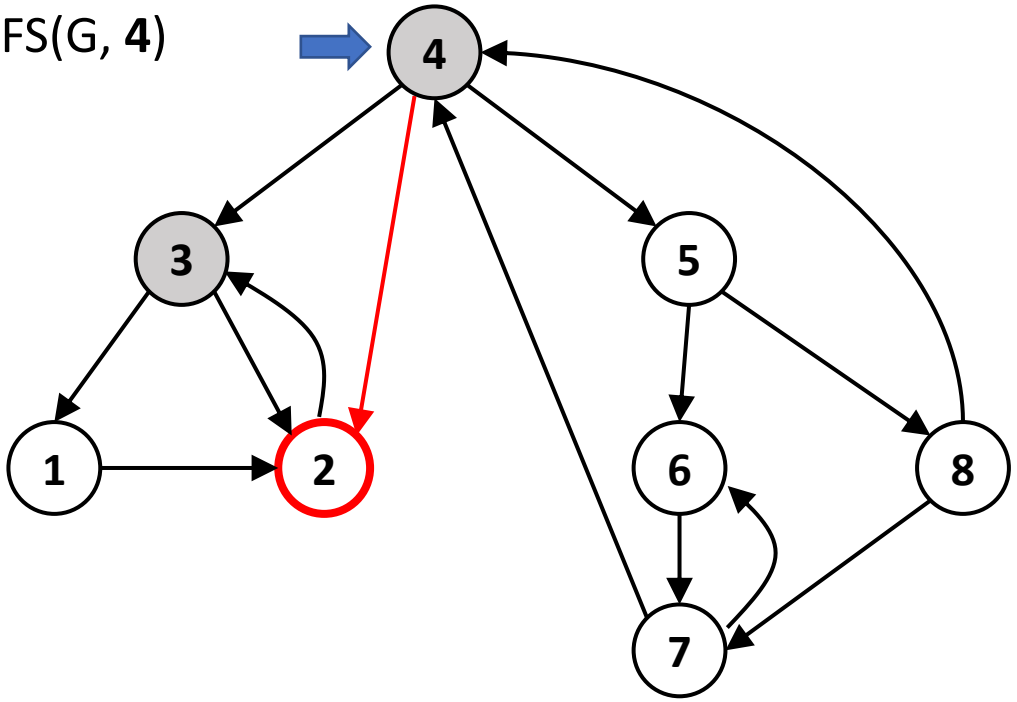
d = 0

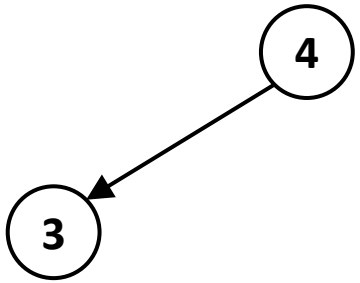
4

d = 1

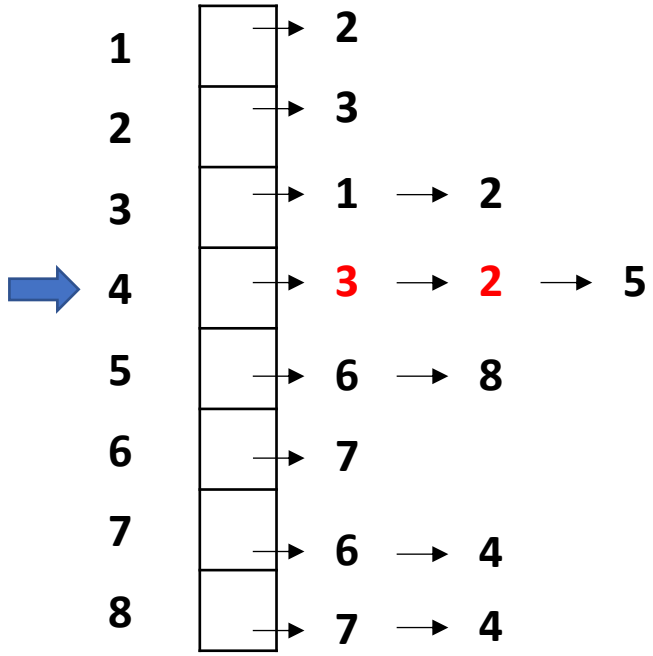
3

BFS(G, **4**)





Adj List of G :



Contents of Q :

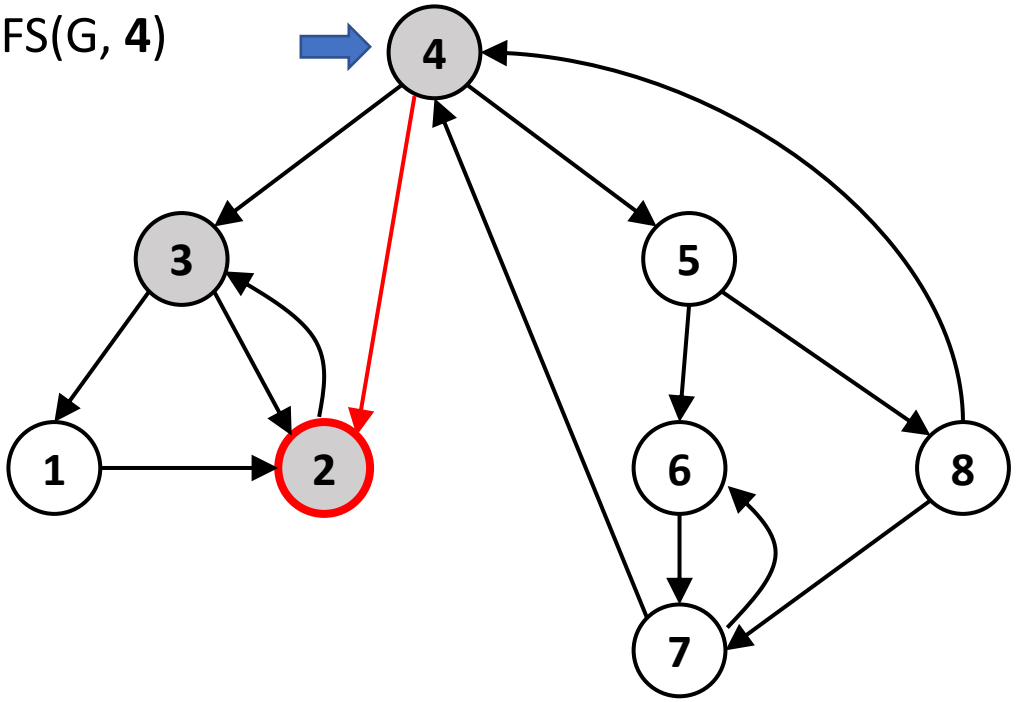
d = 0

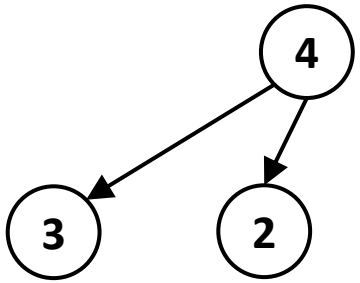
4

d = 1

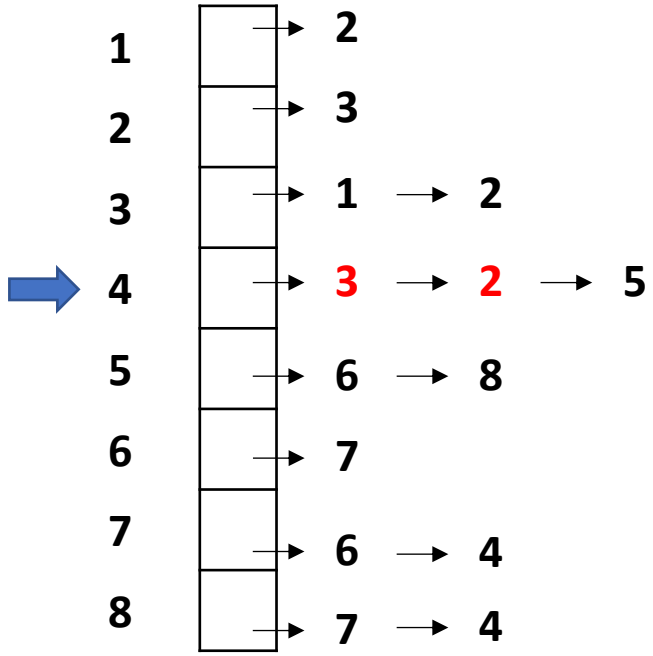
3

BFS(G, **4**)





Adj List of G :



Contents of Q :

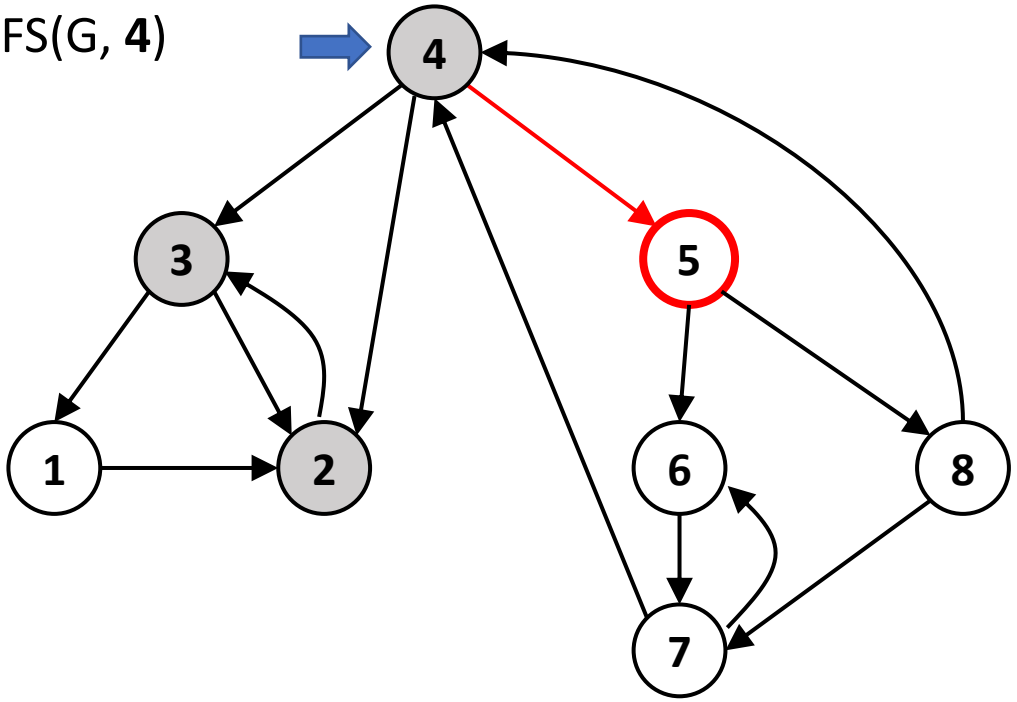
d = 0

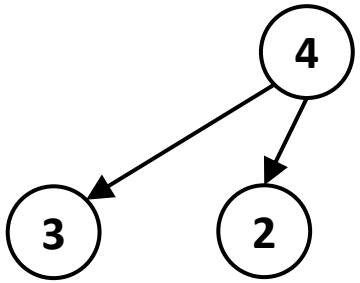
4

d = 1 1

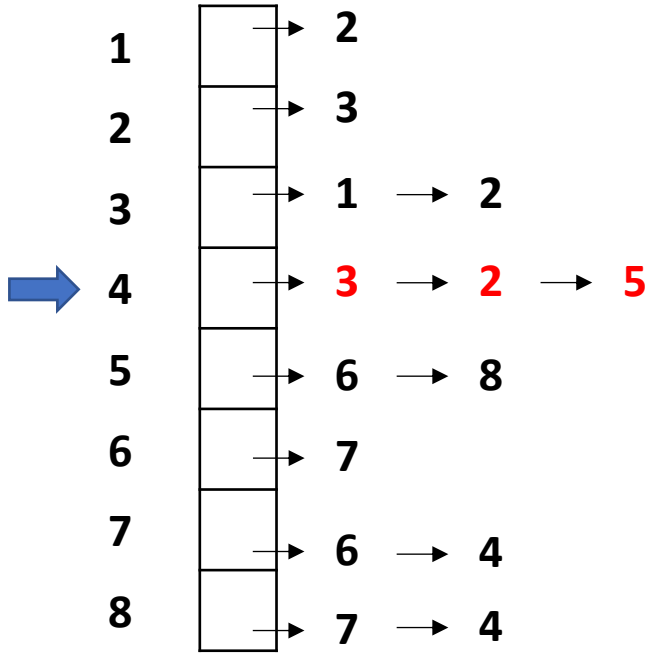
3	2

BFS(G, **4**)





Adj List of G :



Contents of Q :

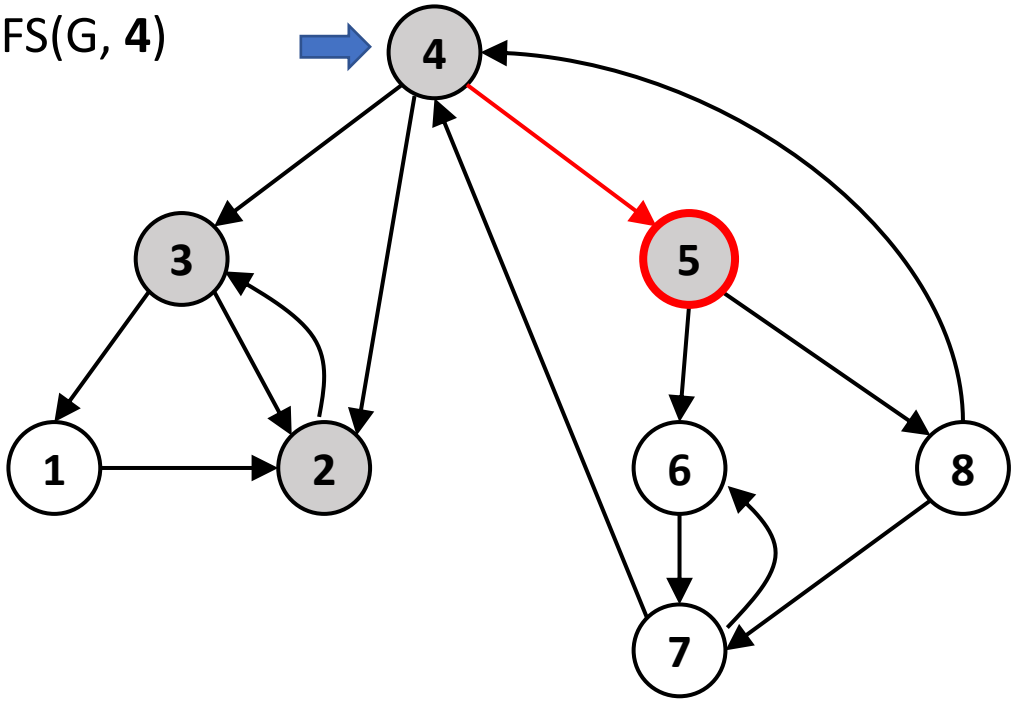
d = 0

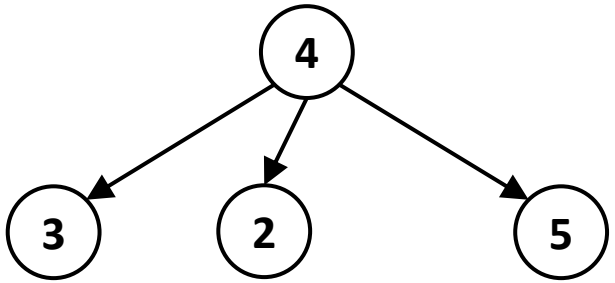
4

d = 1 1

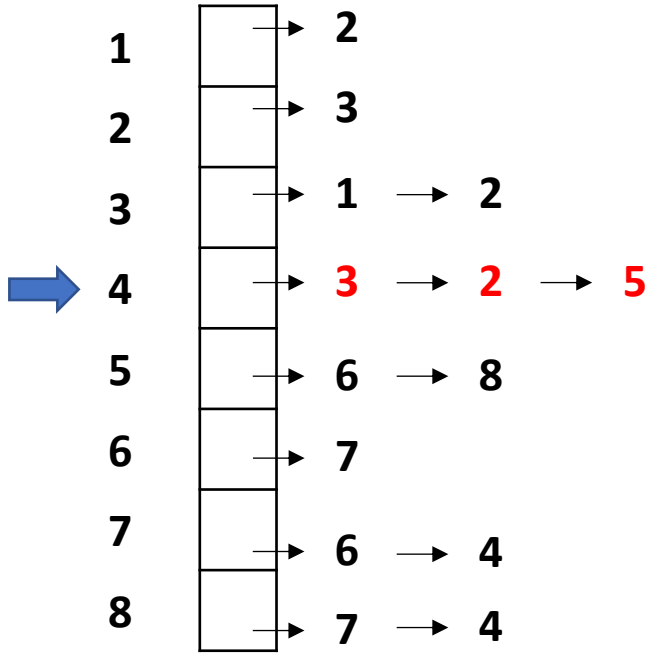
3	2

BFS(G, **4**)





Adj List of G :



Contents of Q :

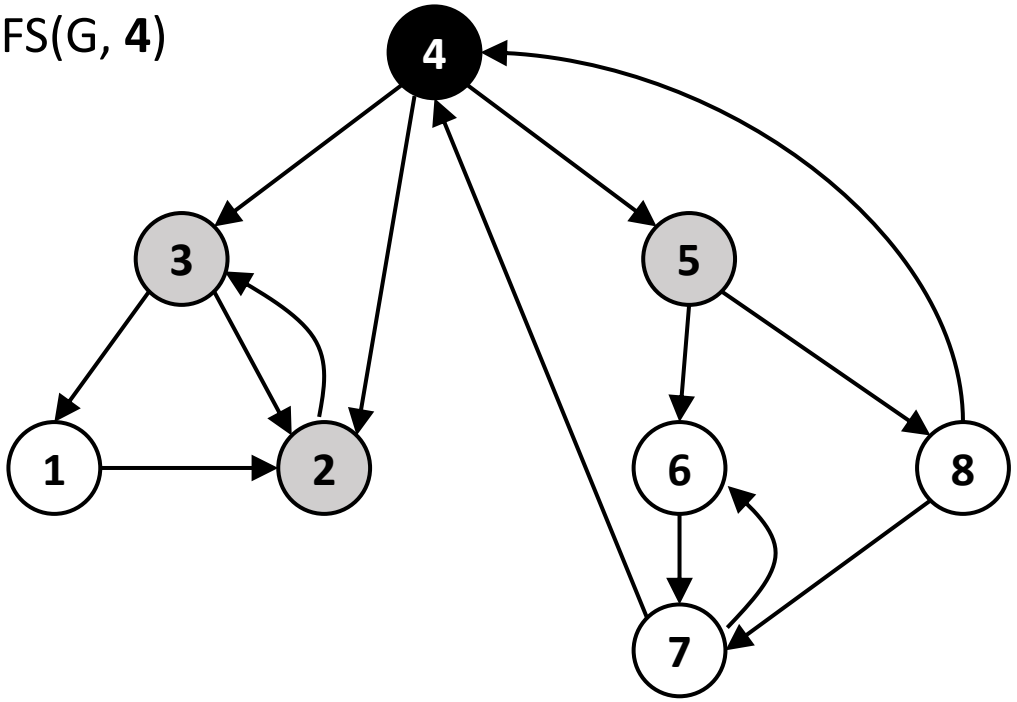
d = 0

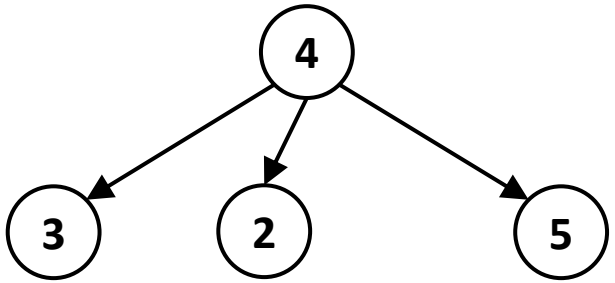
4

d = 1 1 1

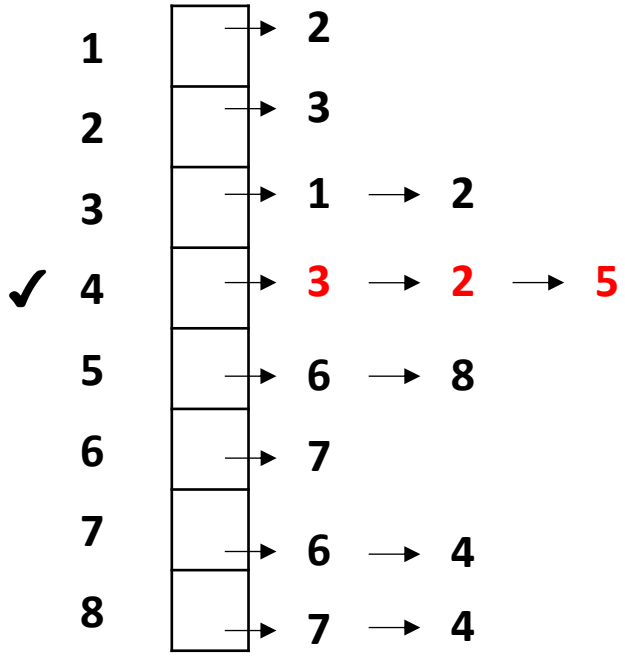
3	2	5

BFS(G, **4**)





Adj List of G :



Contents of Q :

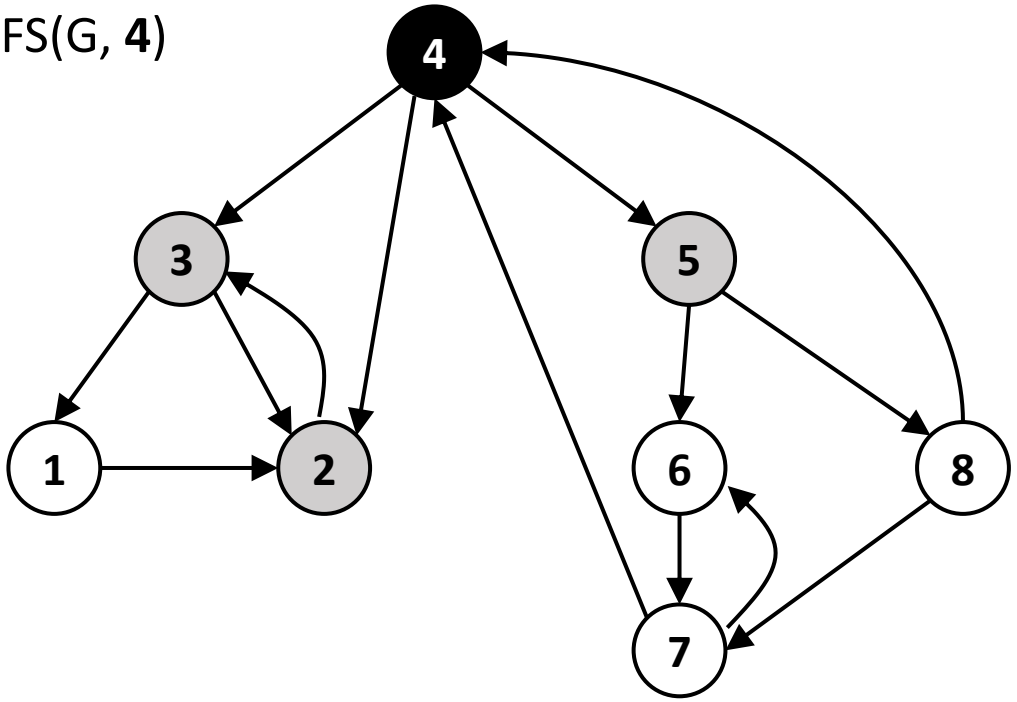
d = 0

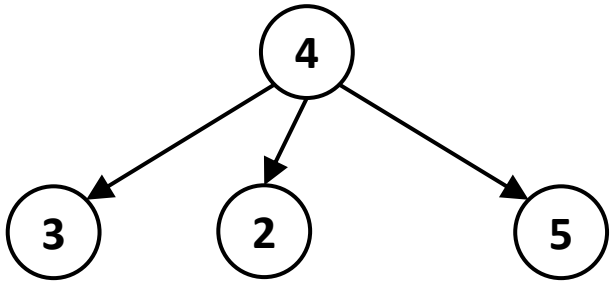
4

d = 1 1 1

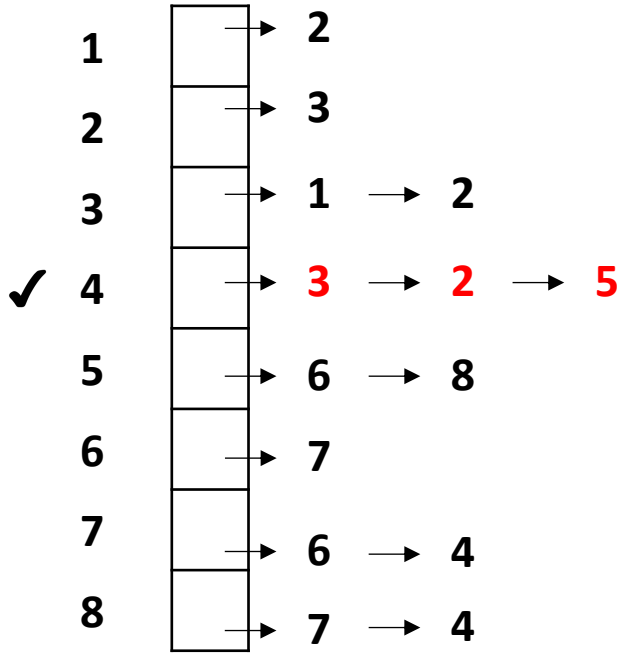
3	2	5

BFS(G, **4**)





Adj List of G :



Contents of Q :

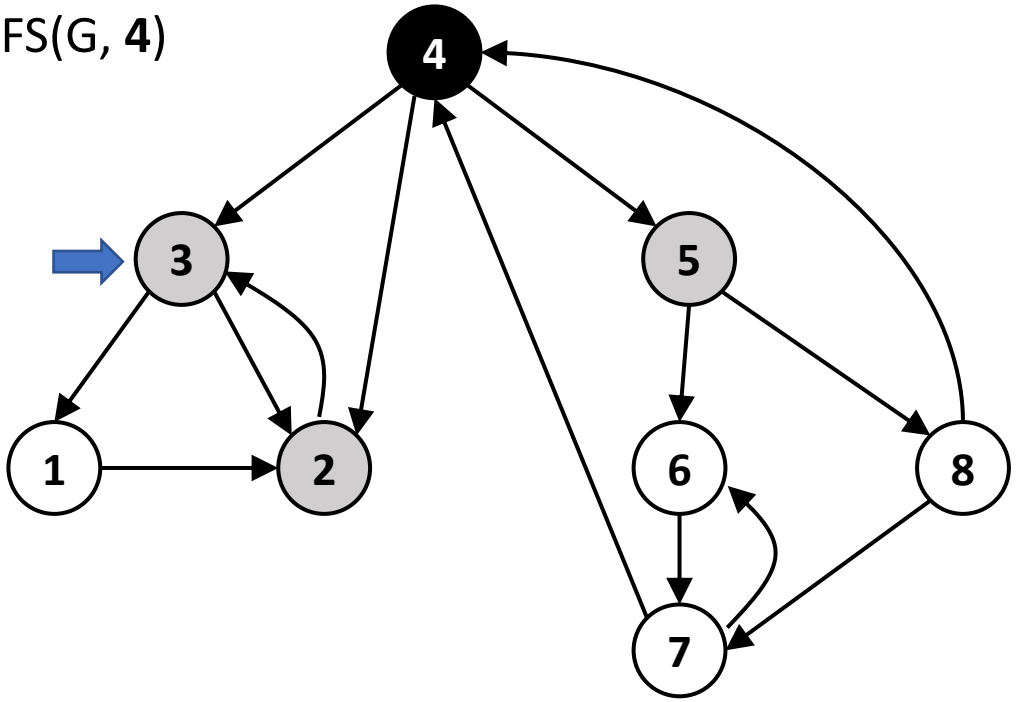
d = 0

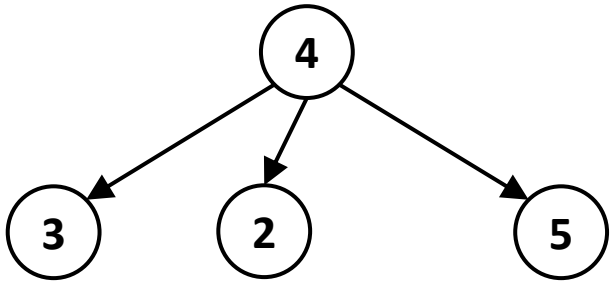
4

d = 1 1 1

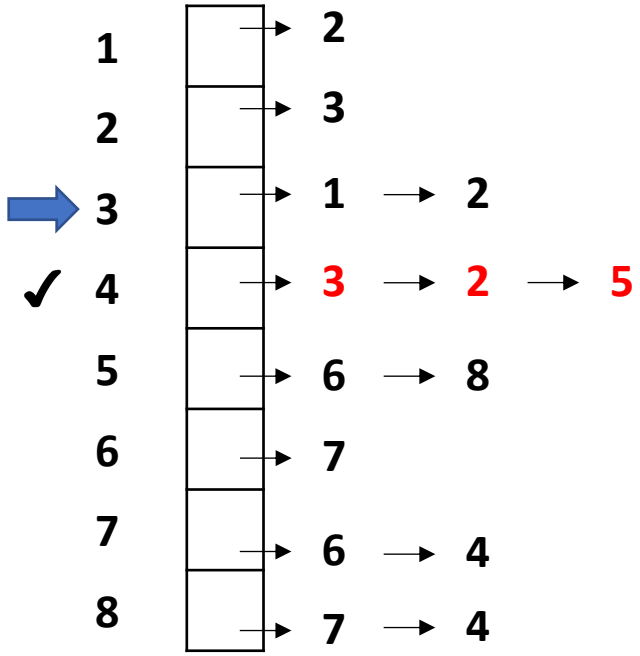
3	2	5

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

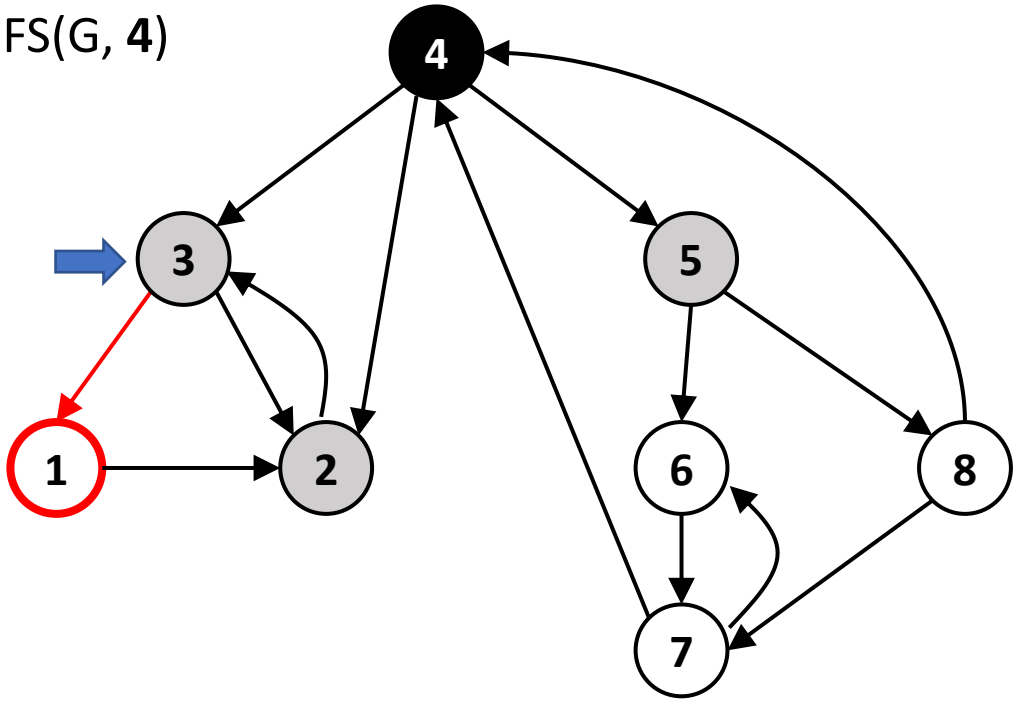
d = 1 1 1

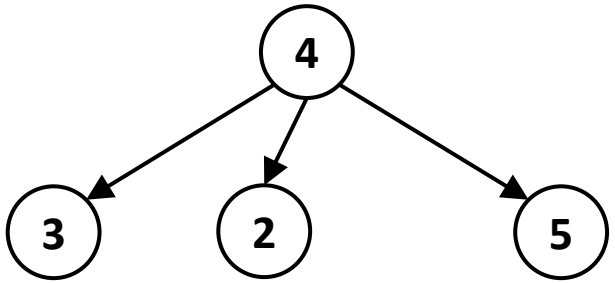
3	2	5

1 1

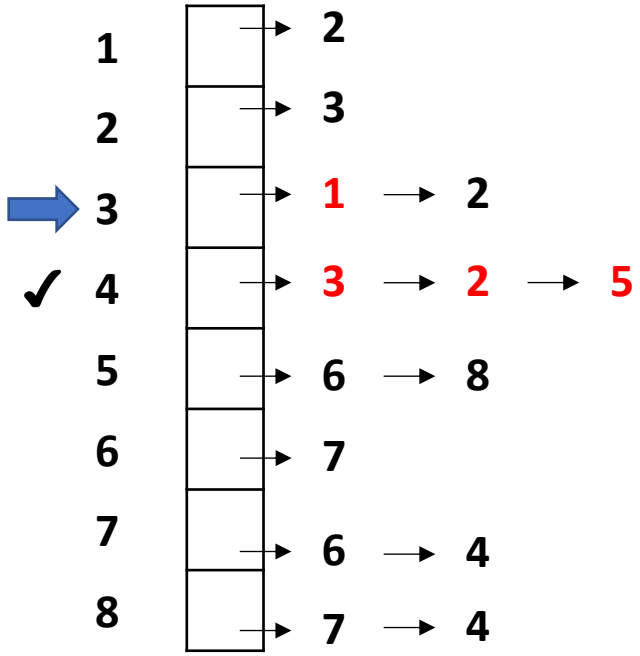
2	5

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

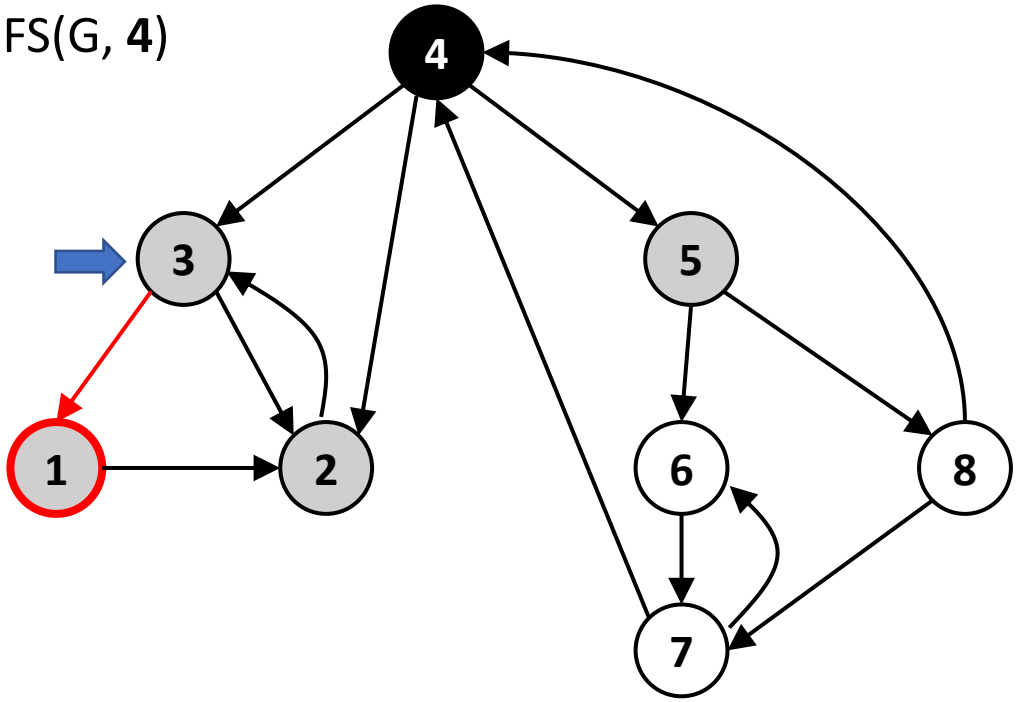
d = 1 1 1

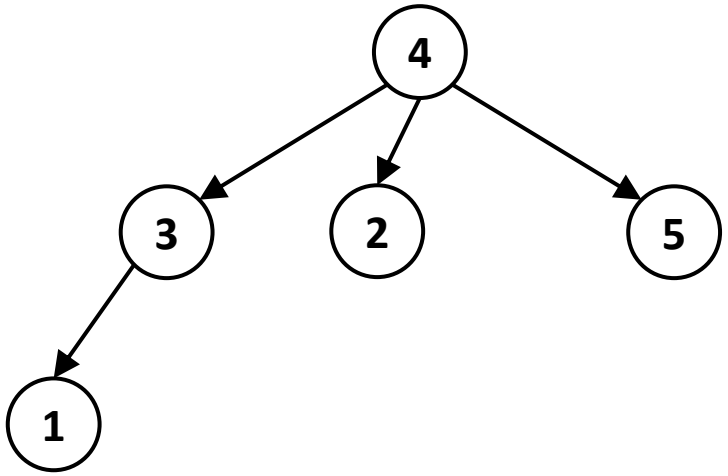
3	2	5

1 1

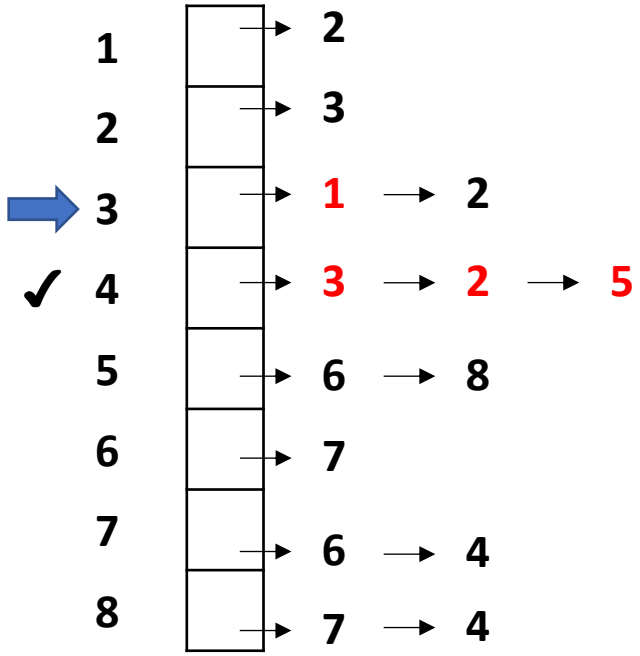
2	5

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

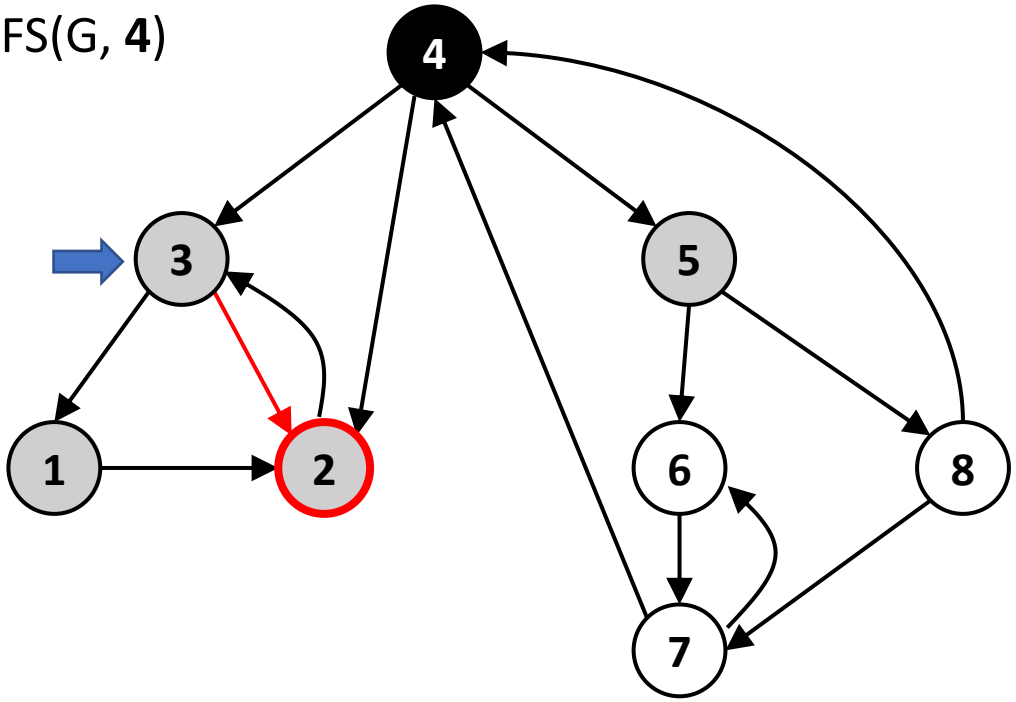
d = 1 1 1

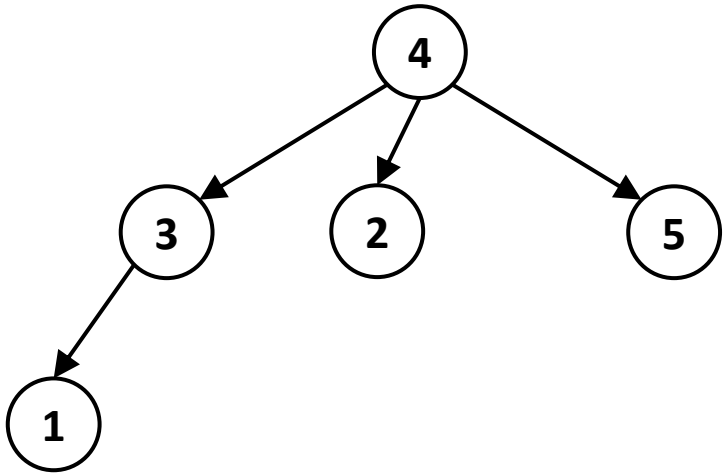
3	2	5

1 1 2

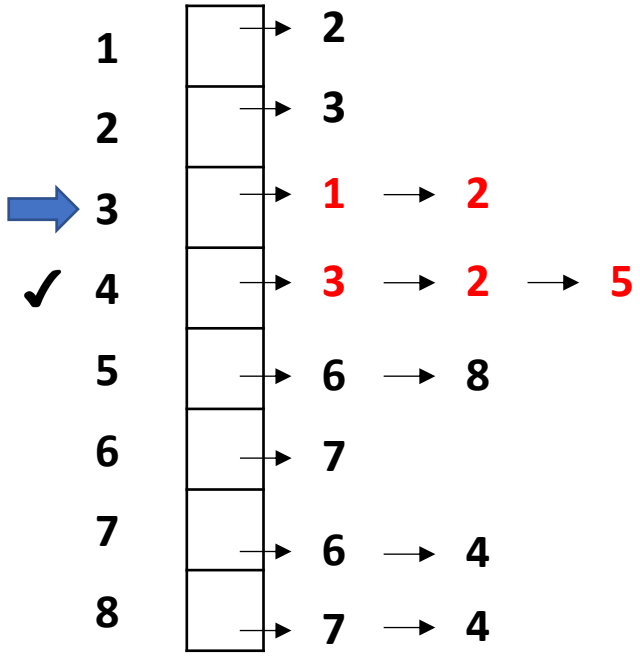
2	5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

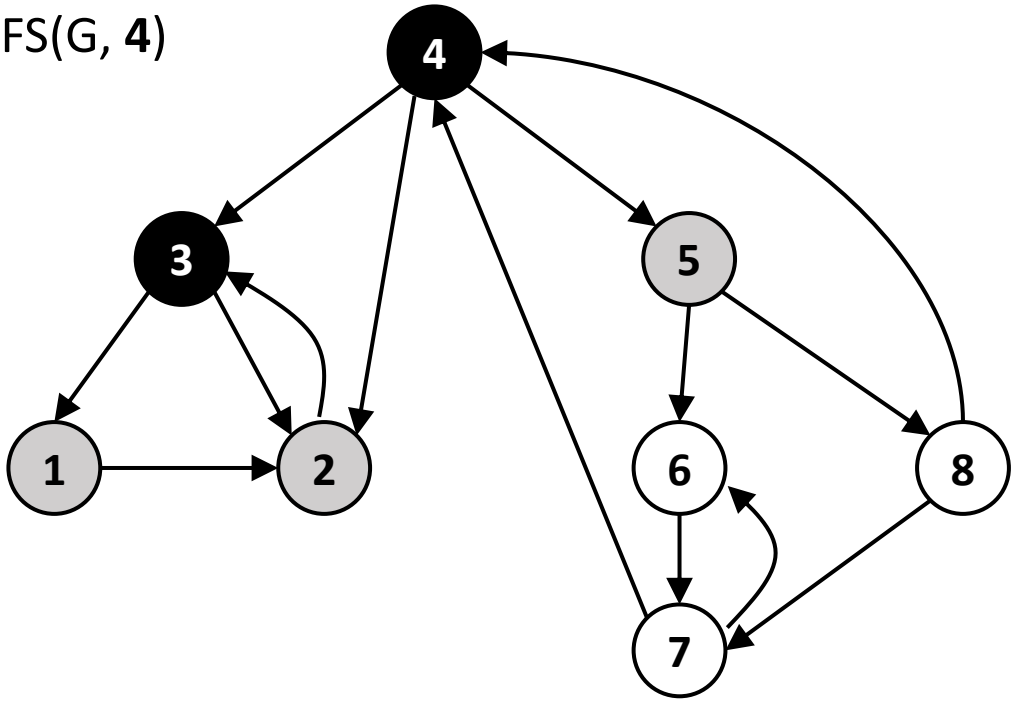
d = 1 1 1

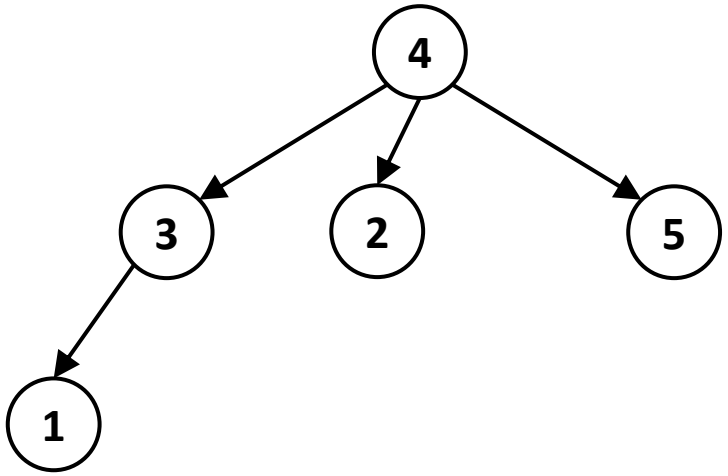
3	2	5

1 1 2

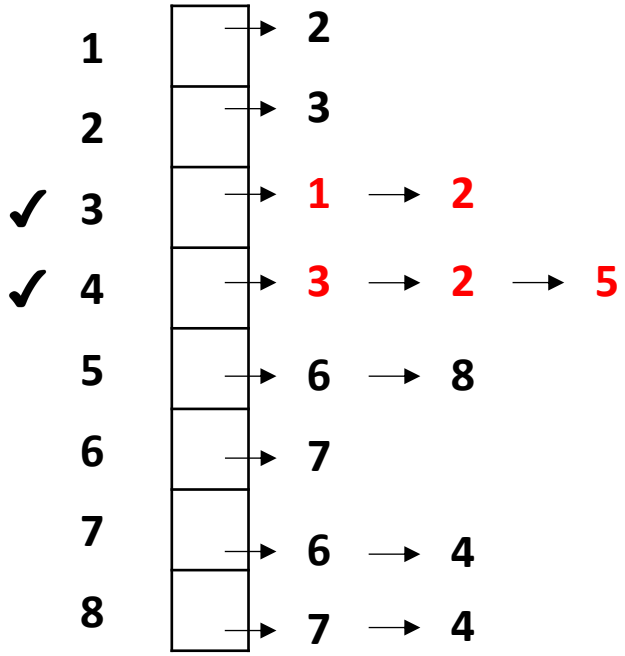
2	5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

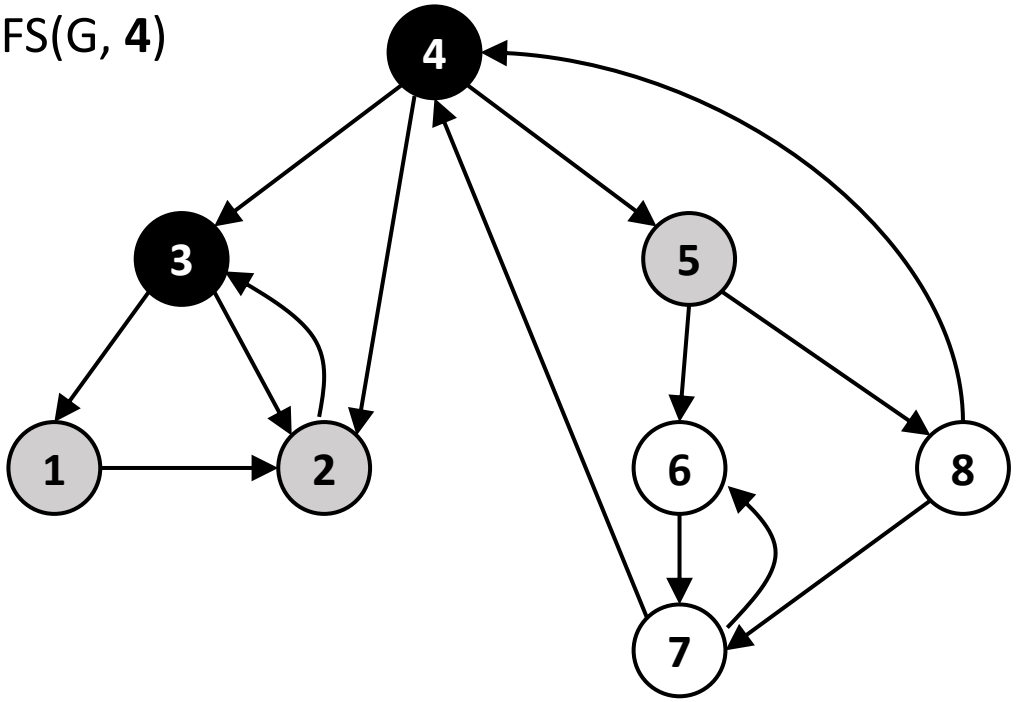
d = 1 1 1

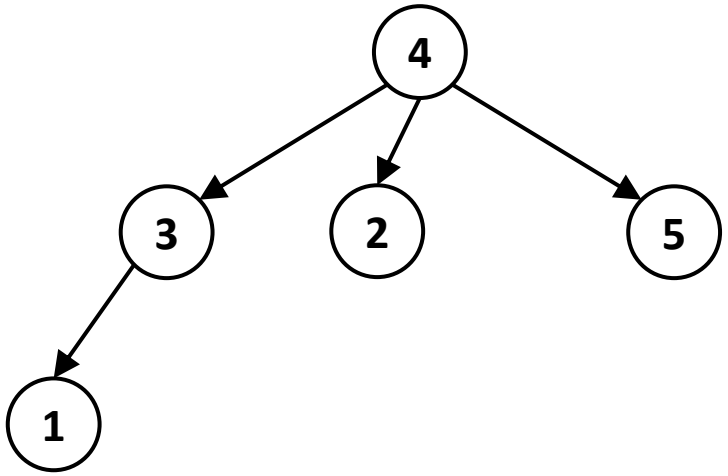
3	2	5

1 1 2

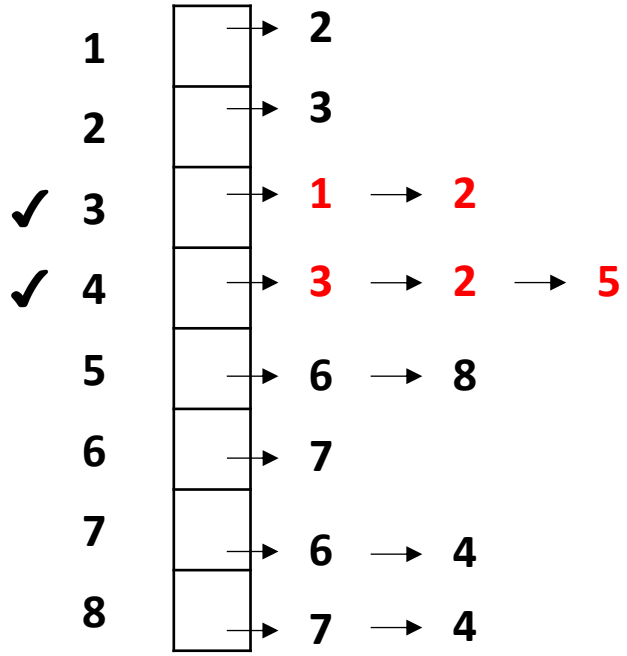
2	5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

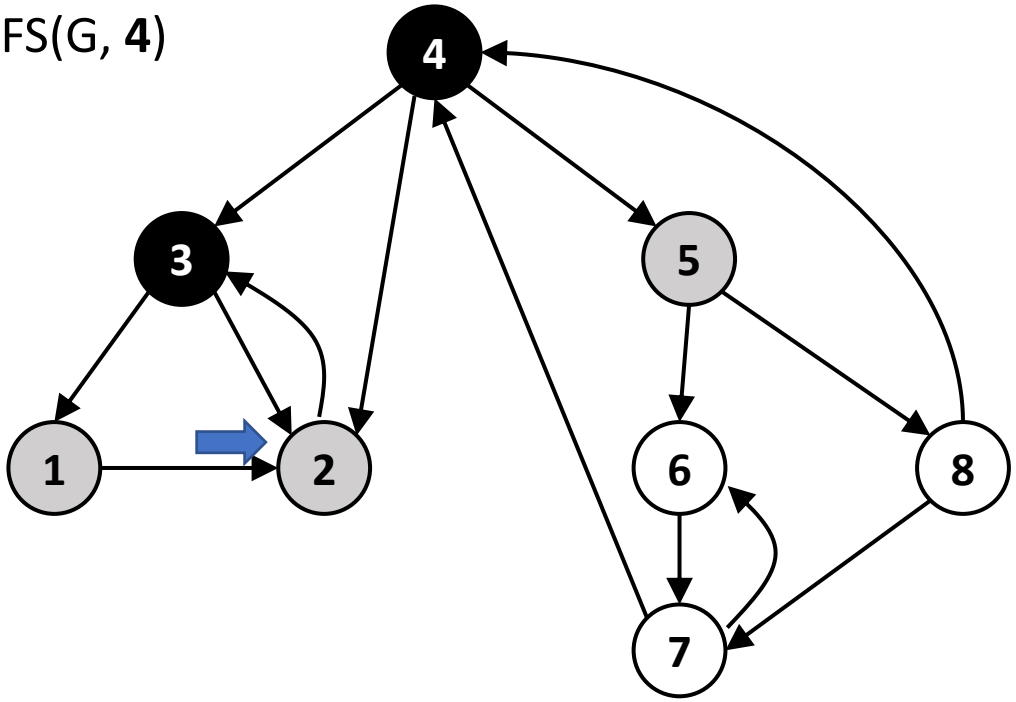
d = 1 1 1

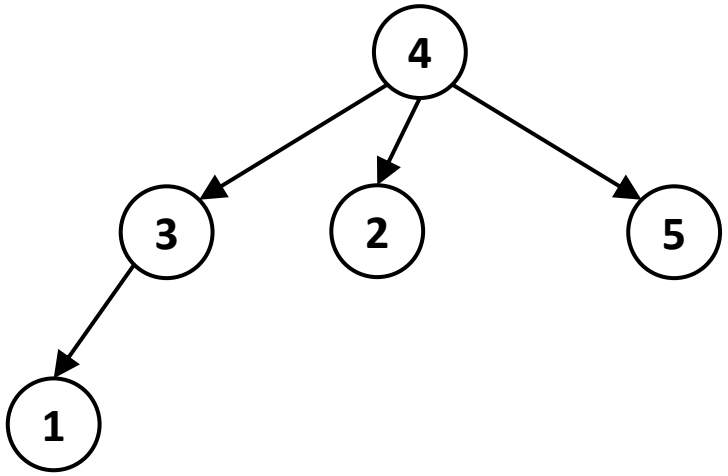
3	2	5

1 1 2

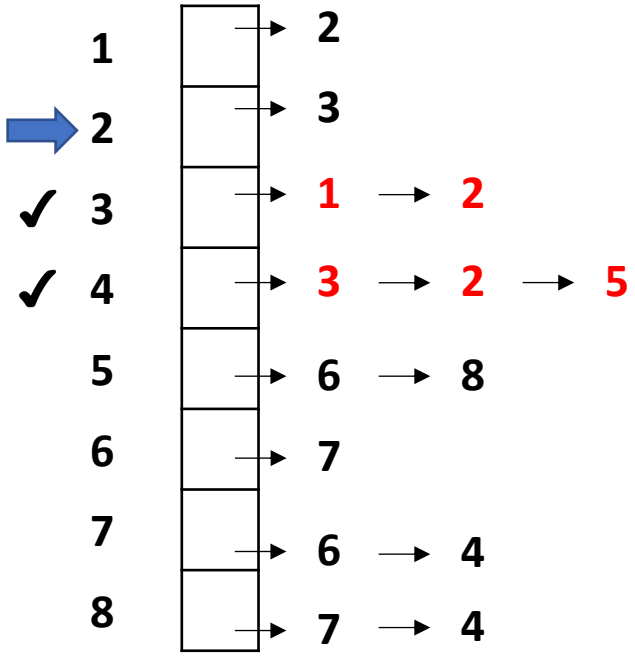
2	5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

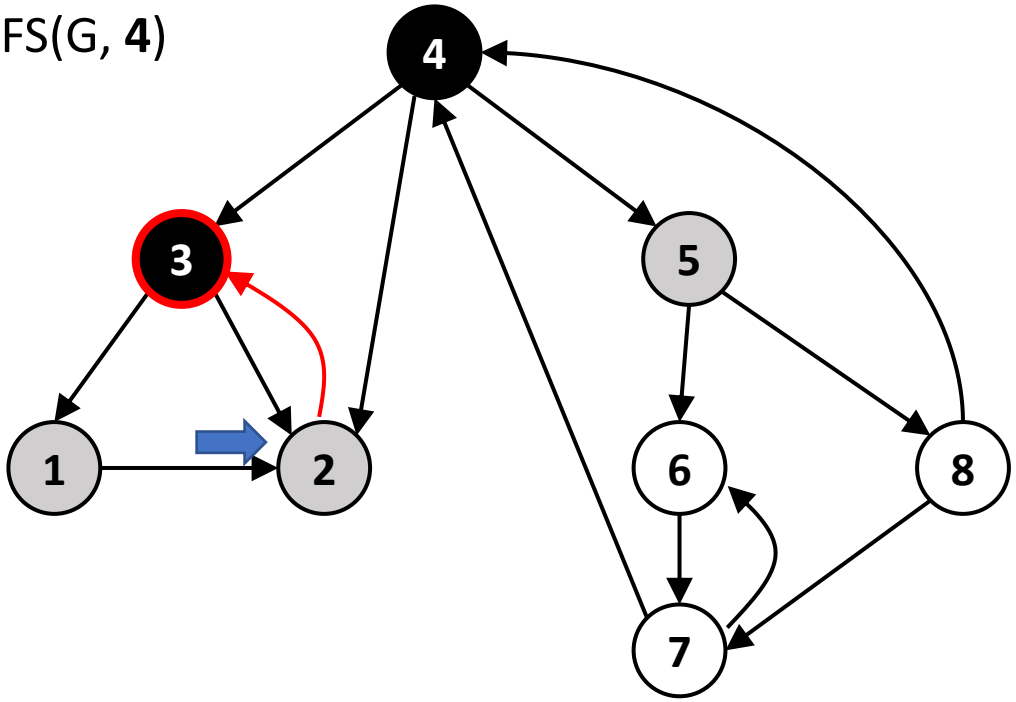
1 1 2

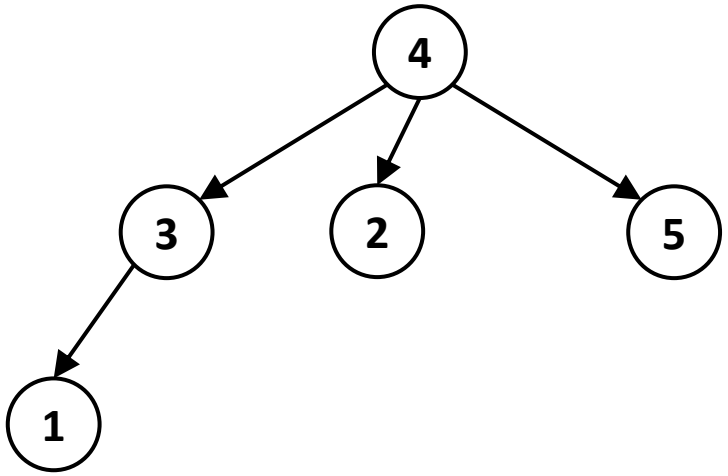
2	5	1

1 2

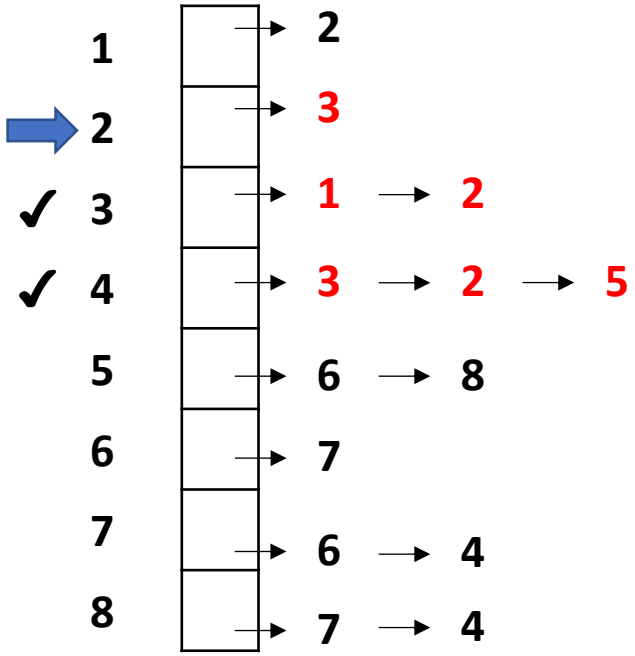
5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

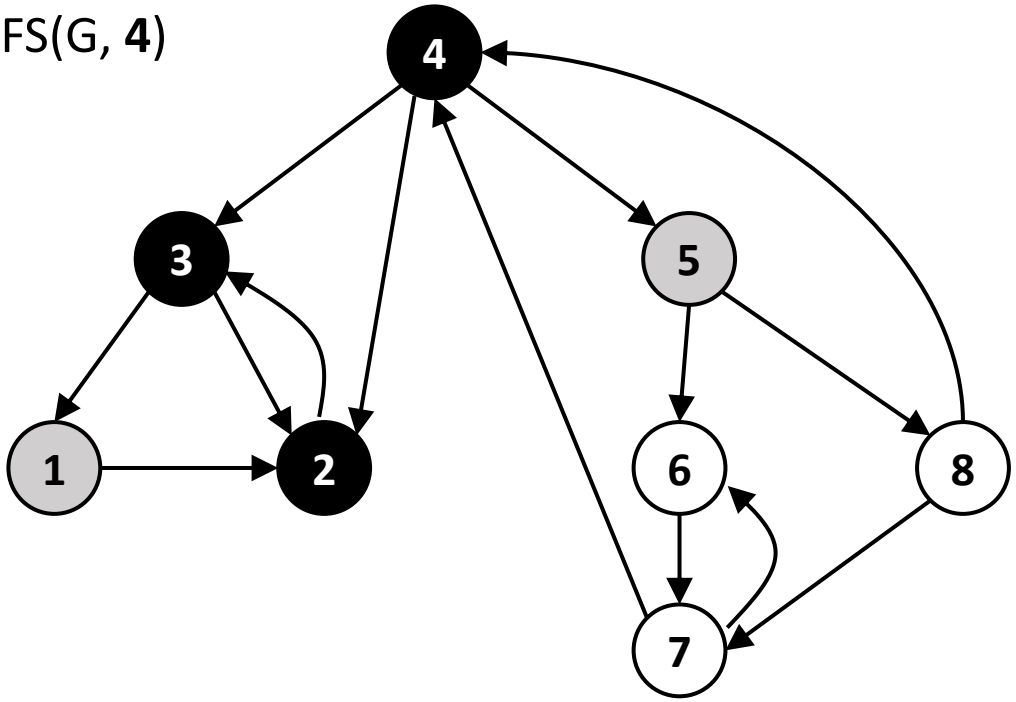
1 1 2

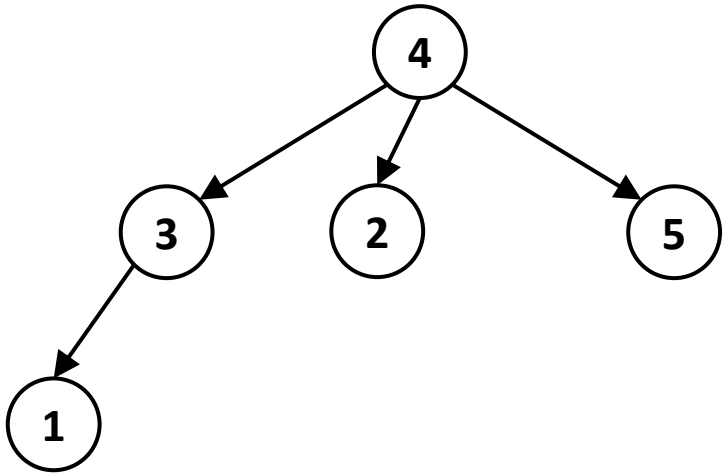
2	5	1

1 2

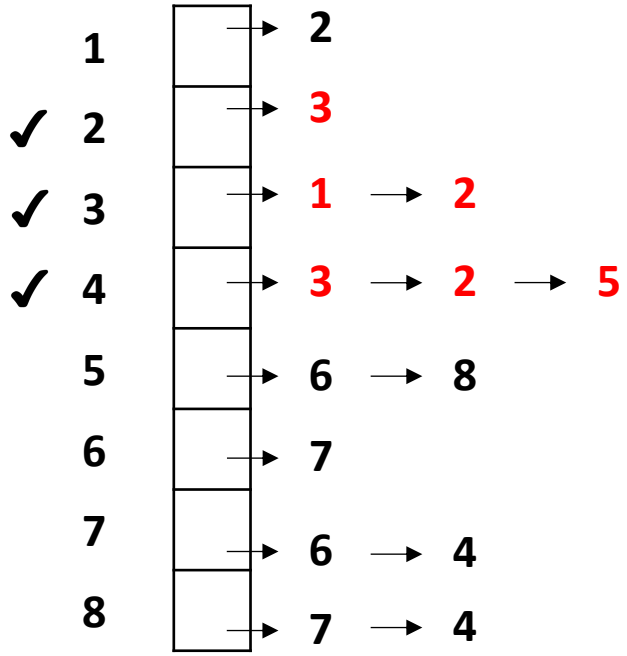
5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

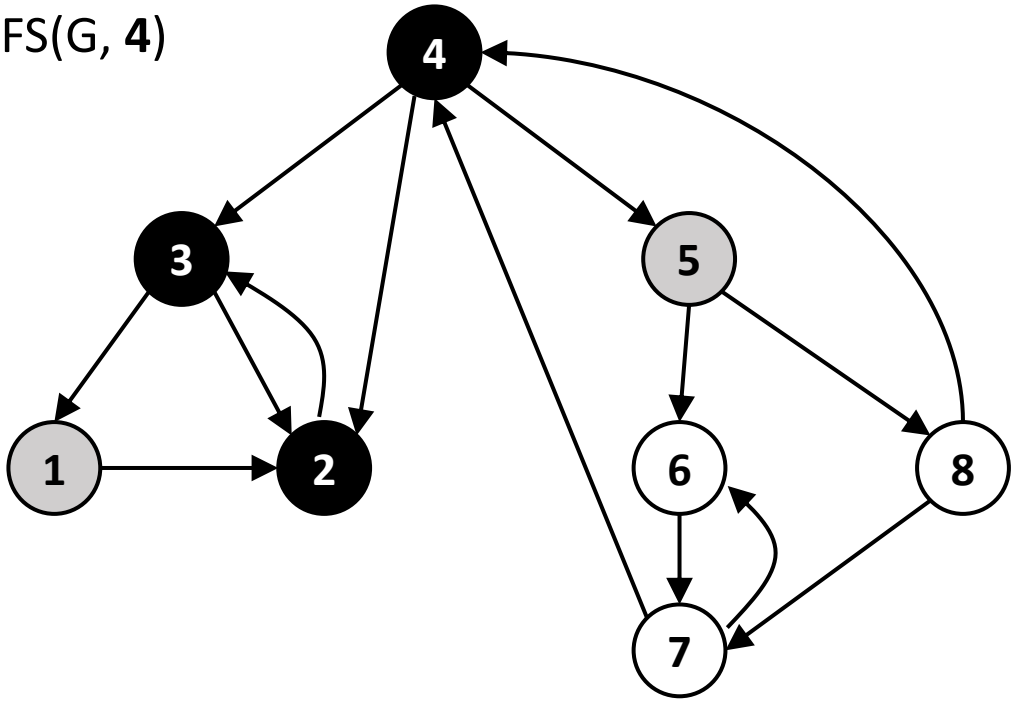
1 1 2

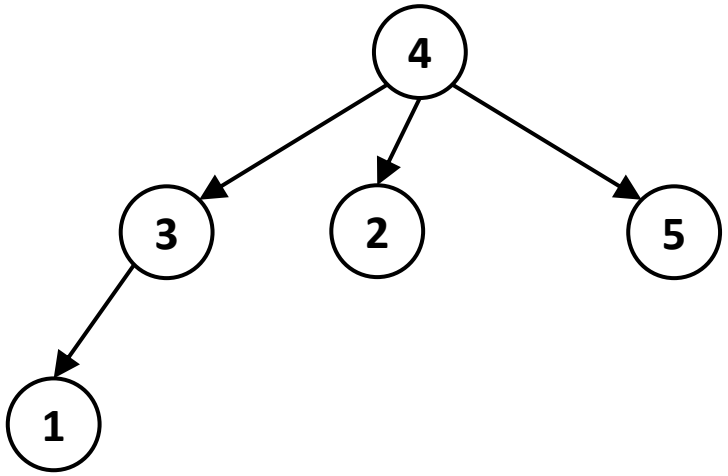
2	5	1

1 2

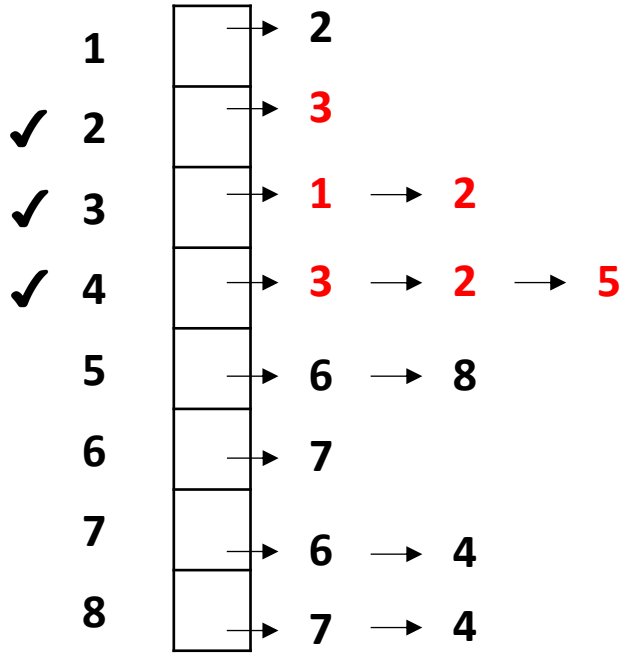
5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

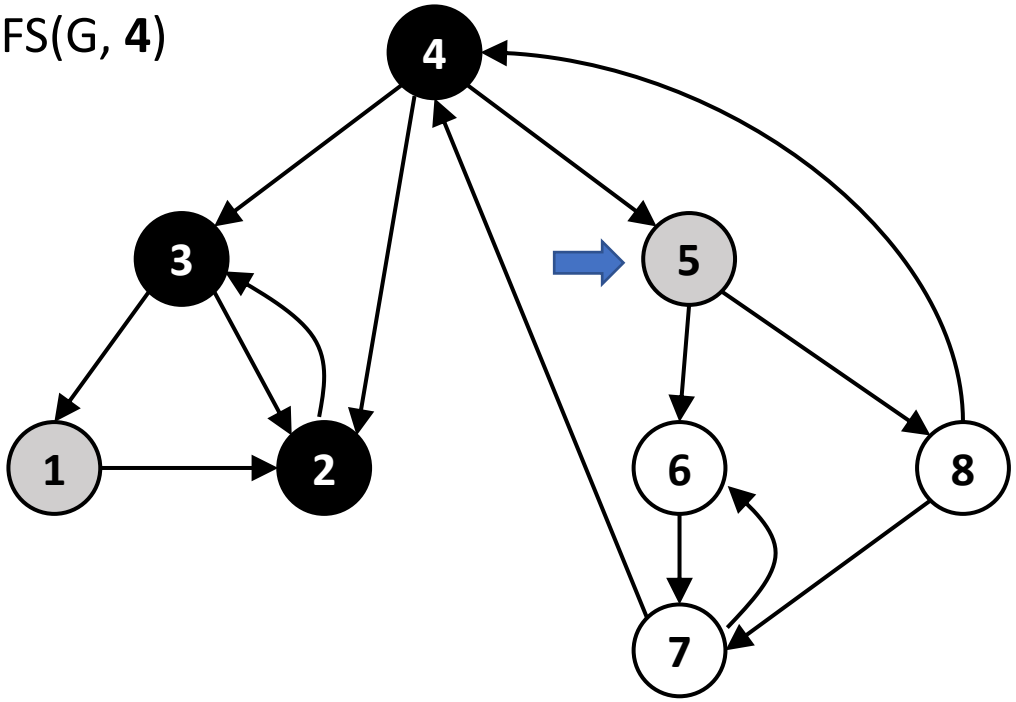
1 1 2

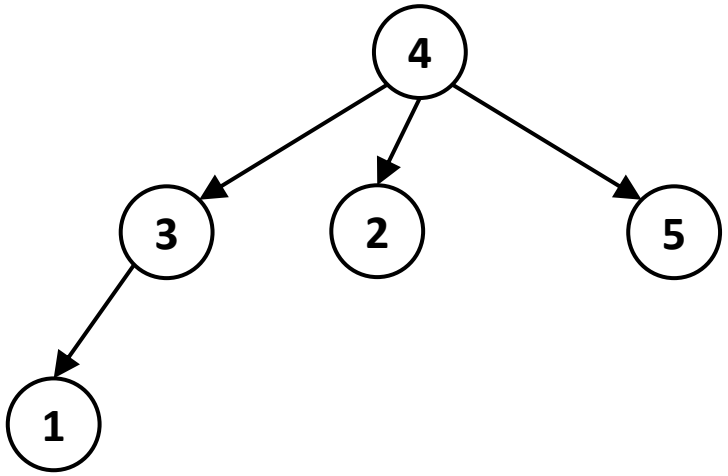
2	5	1

1 2

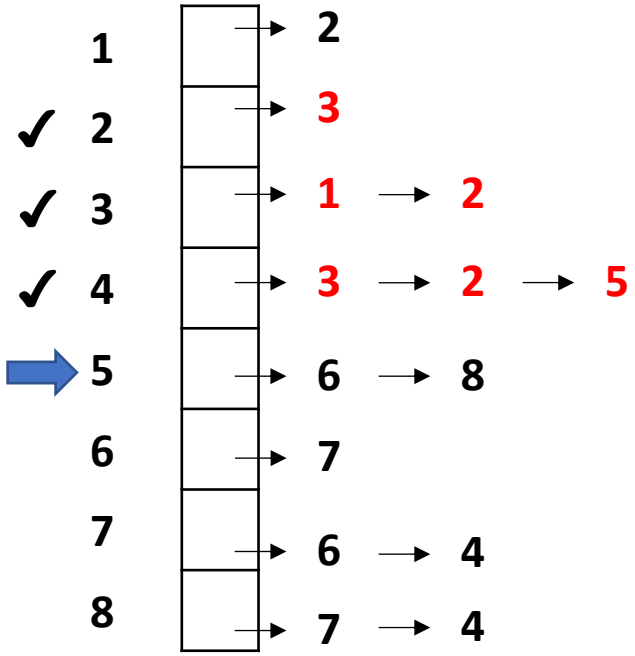
5	1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

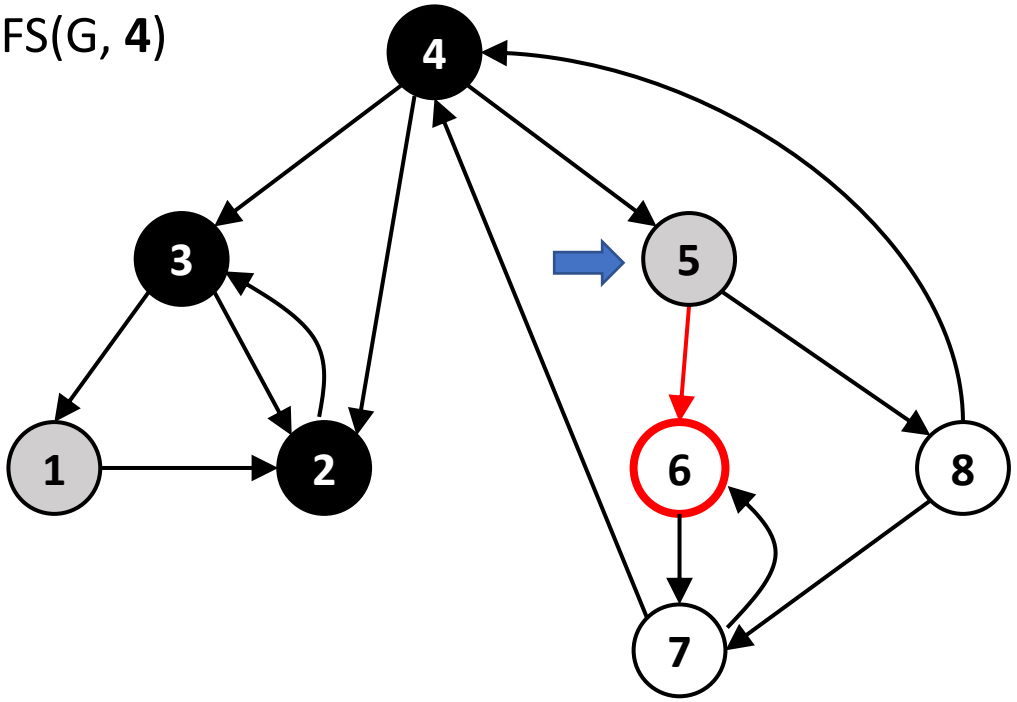
1 2

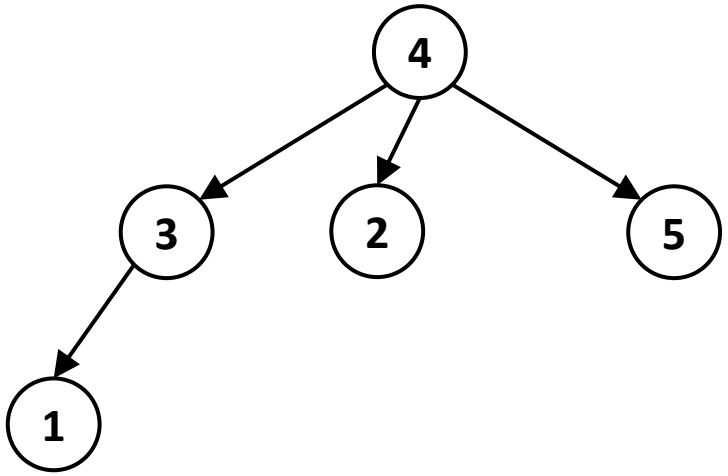
5	1

2

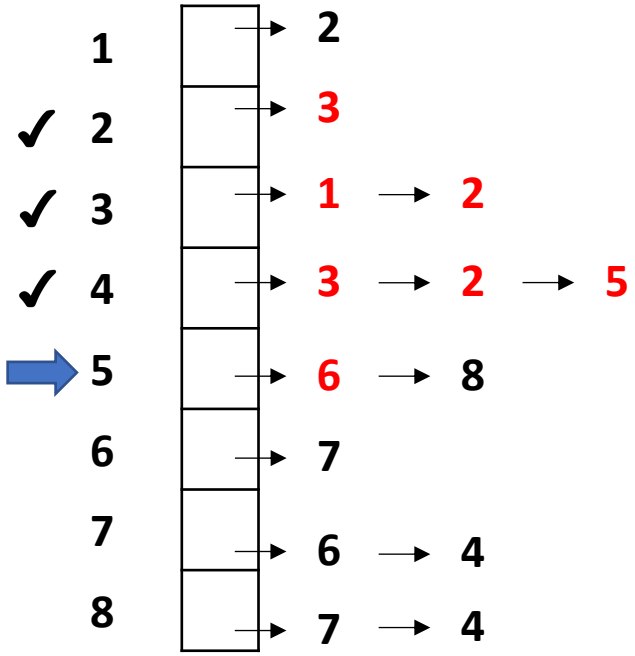
1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

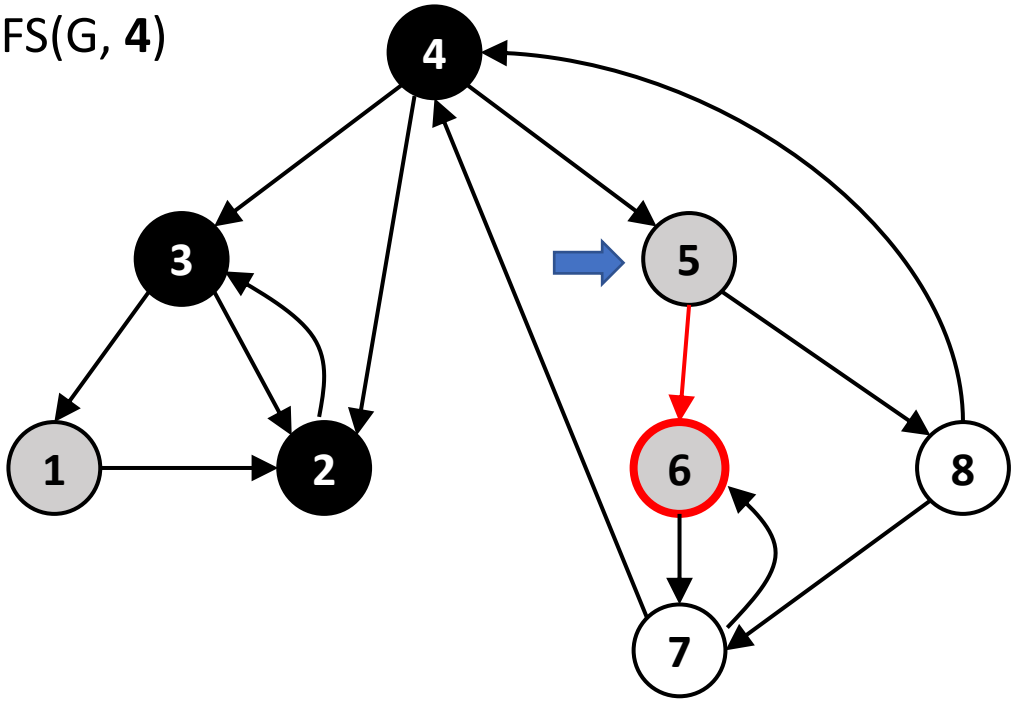
1 2

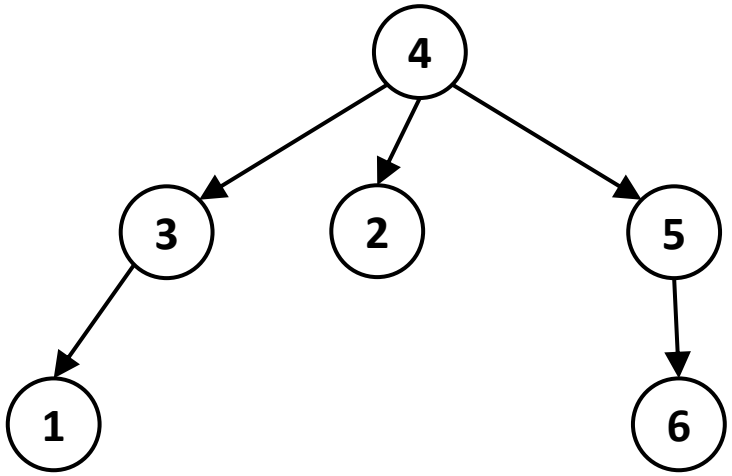
5	1

2

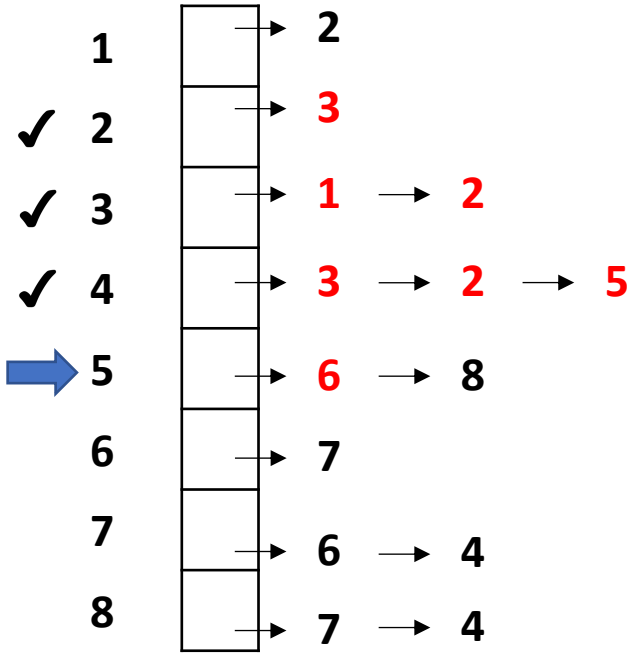
1

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d =

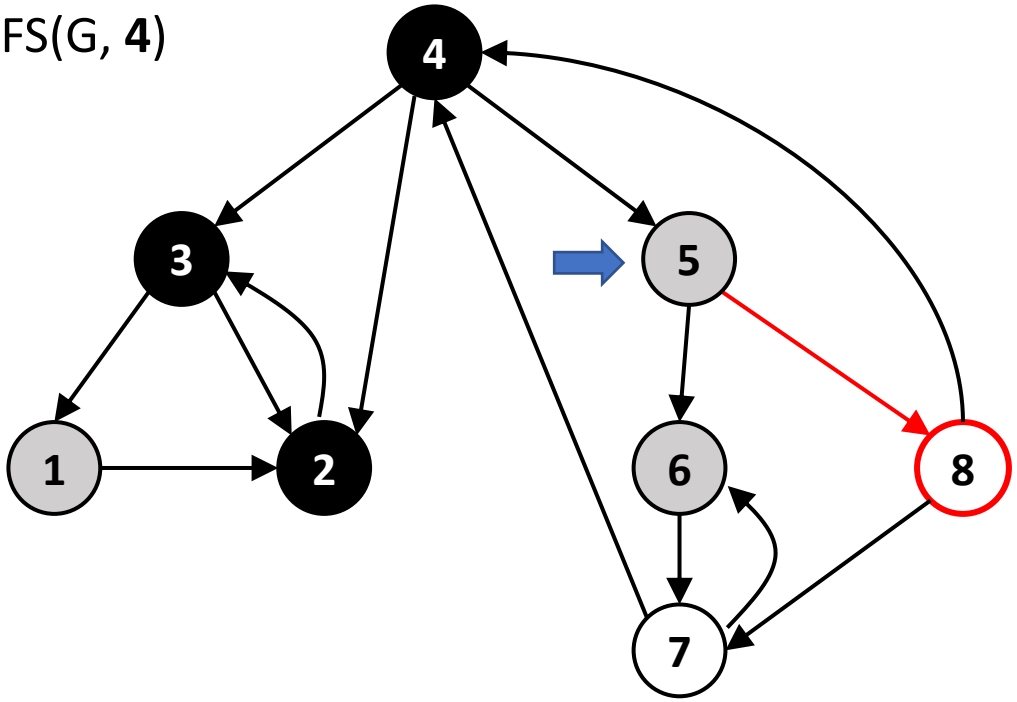
1	1	1
3	2	5

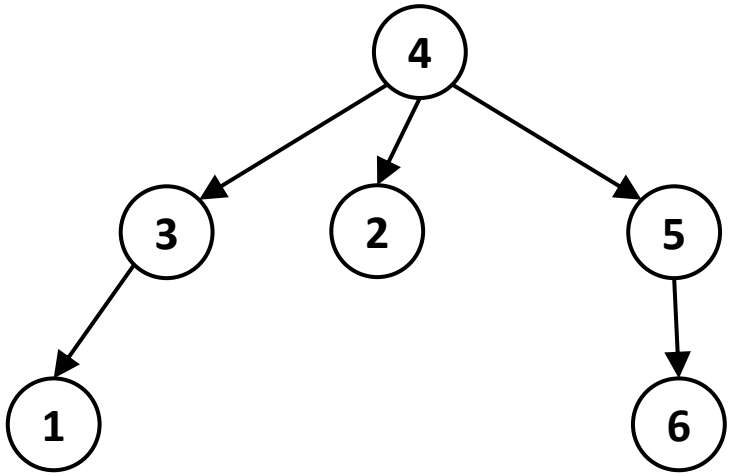
1	1	2
2	5	1

1	2
5	1

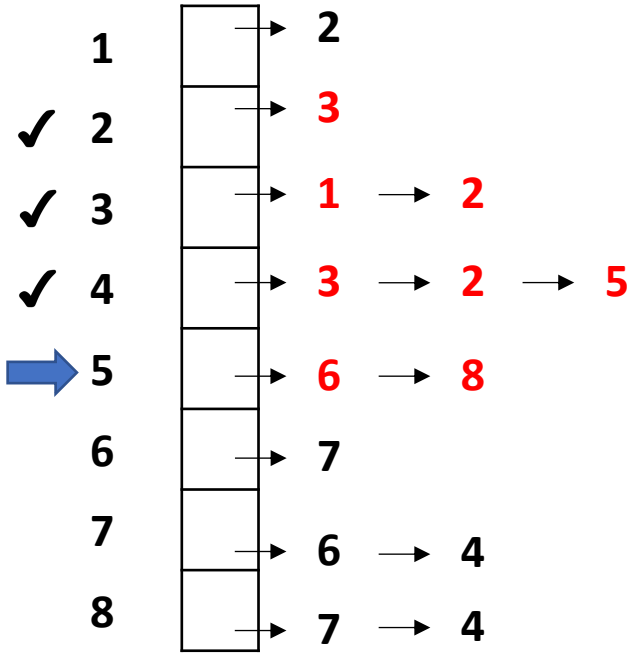
2	2
1	6

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

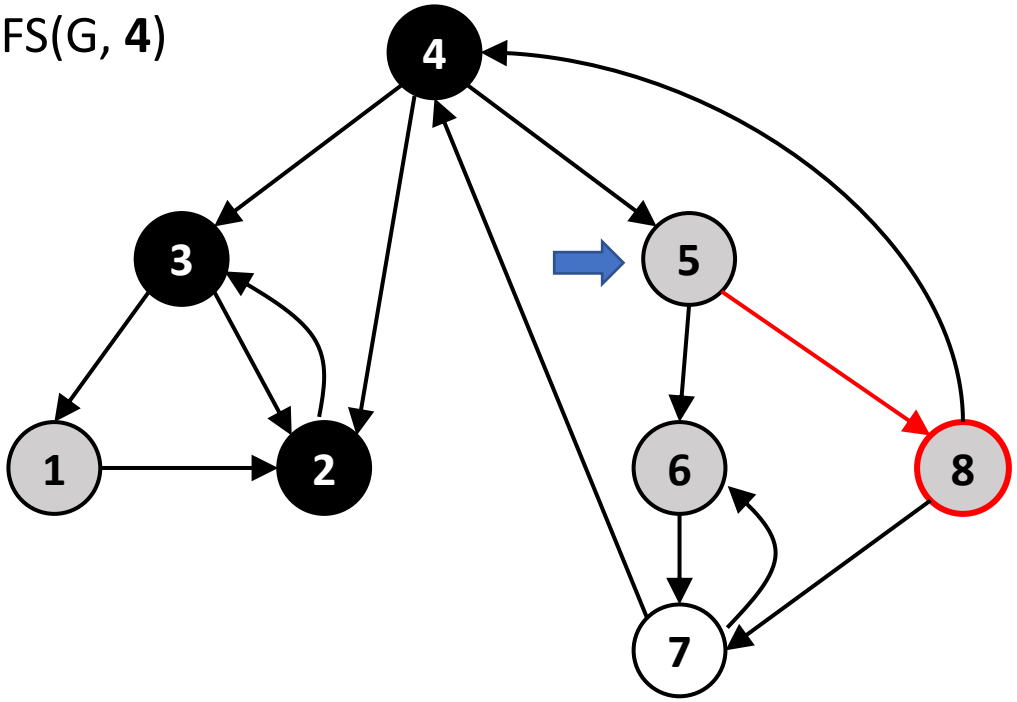
1 2

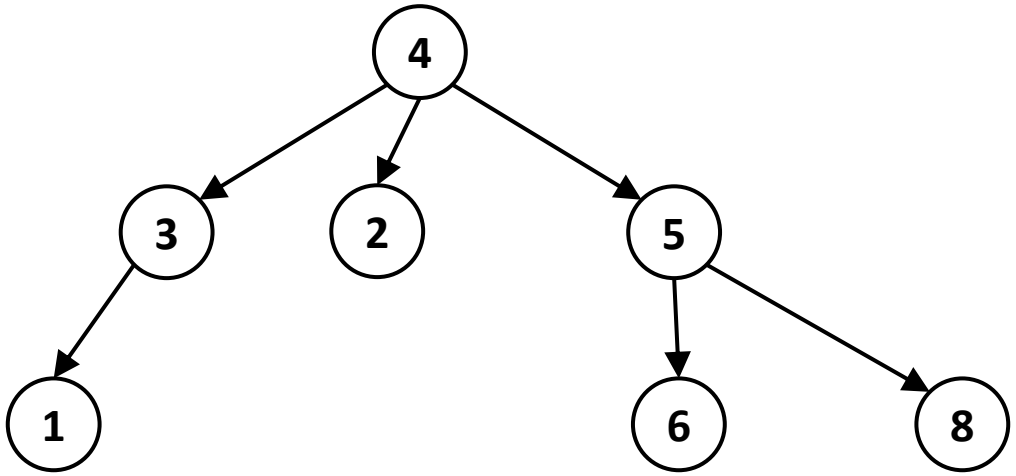
5	1

2 2

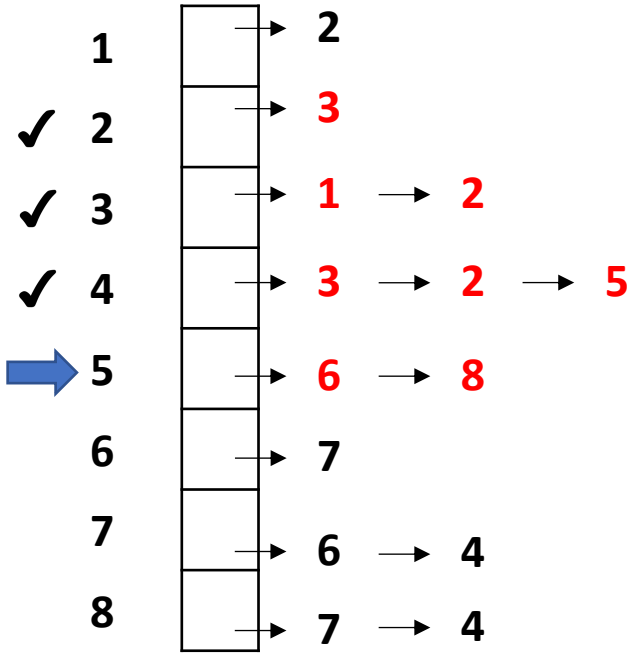
1	6

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

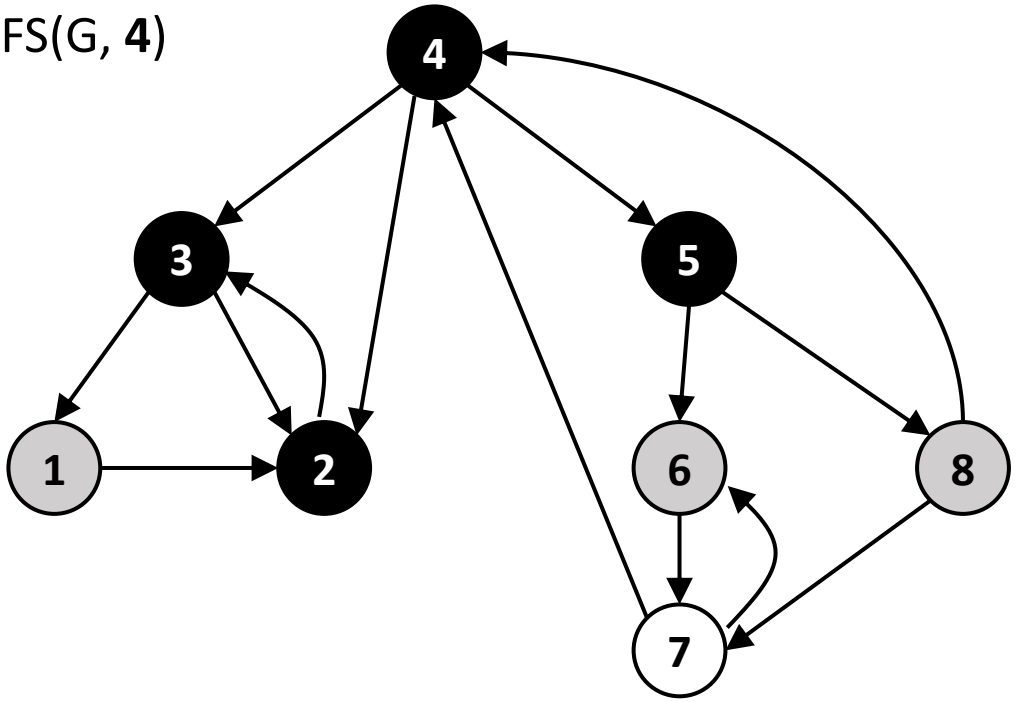
1 2

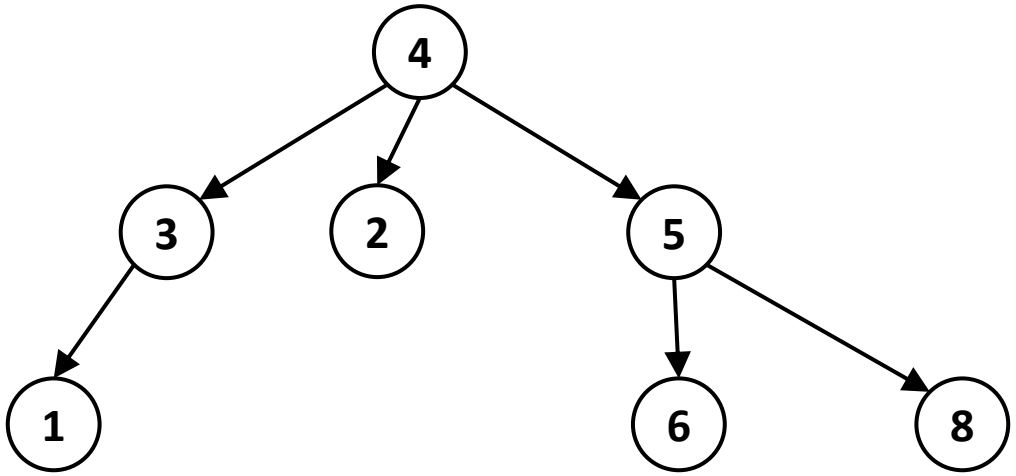
5	1

2 2 2

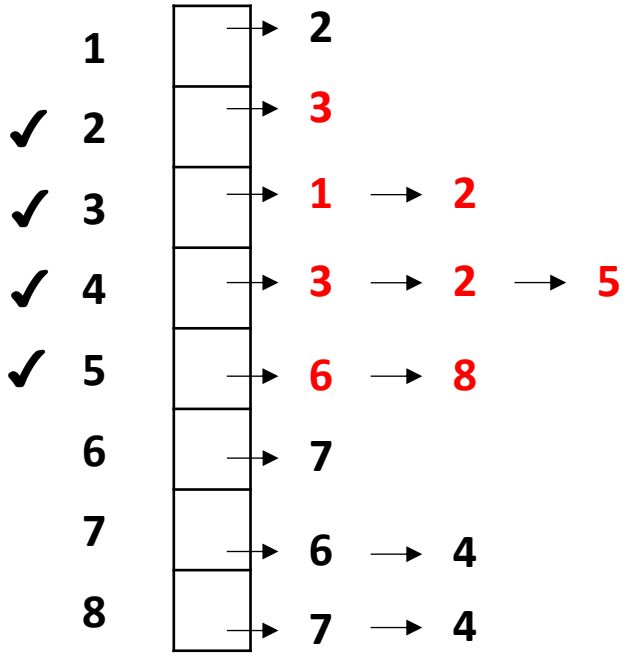
1	6	8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d =

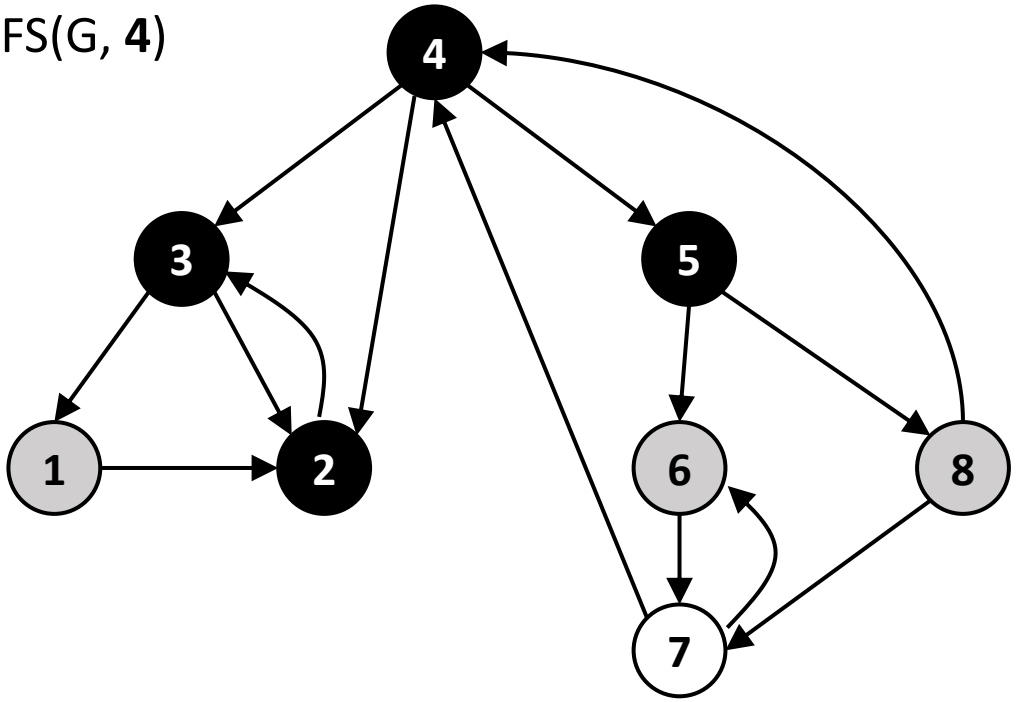
1	1	1
3	2	5

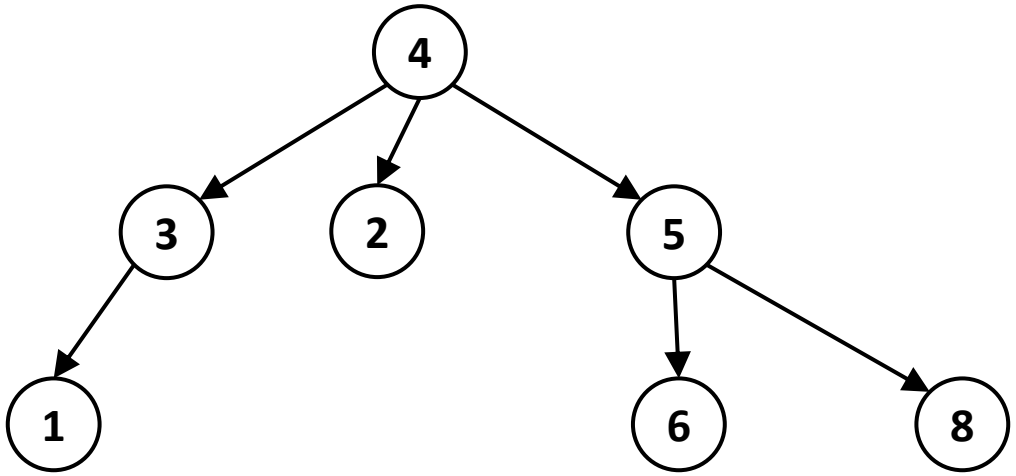
1	1	2
2	5	1

1	2
5	1

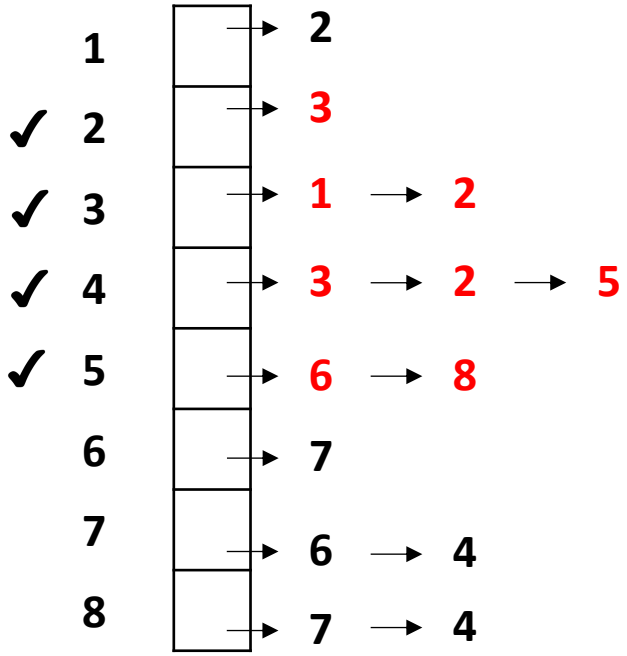
2	2	2
1	6	8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d =

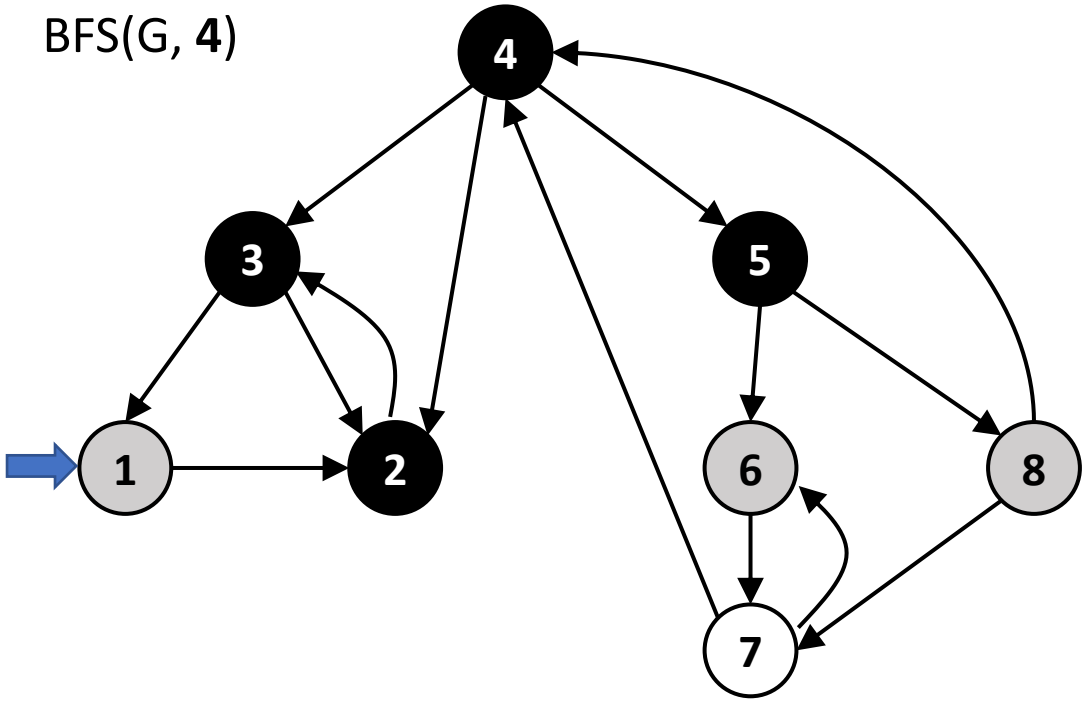
1	1	1
3	2	5

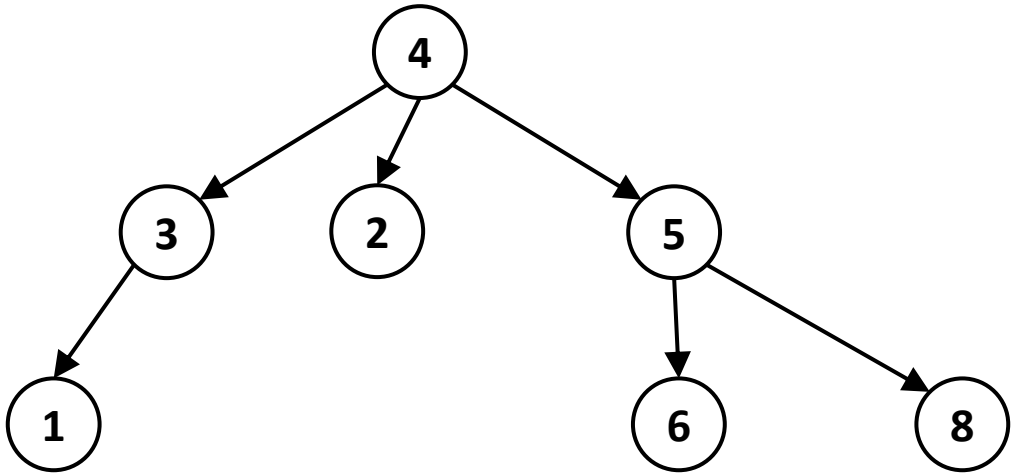
1	1	2
2	5	1

1	2
5	1

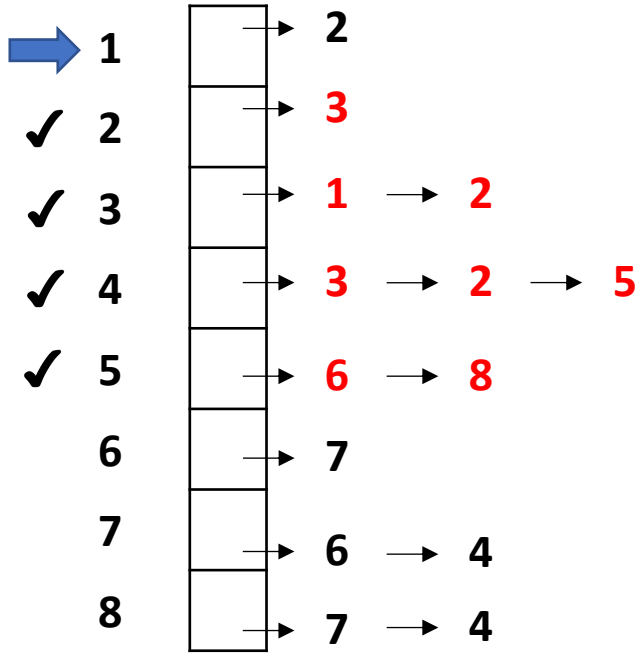
2	2	2
1	6	8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d =

1	1	1
3	2	5

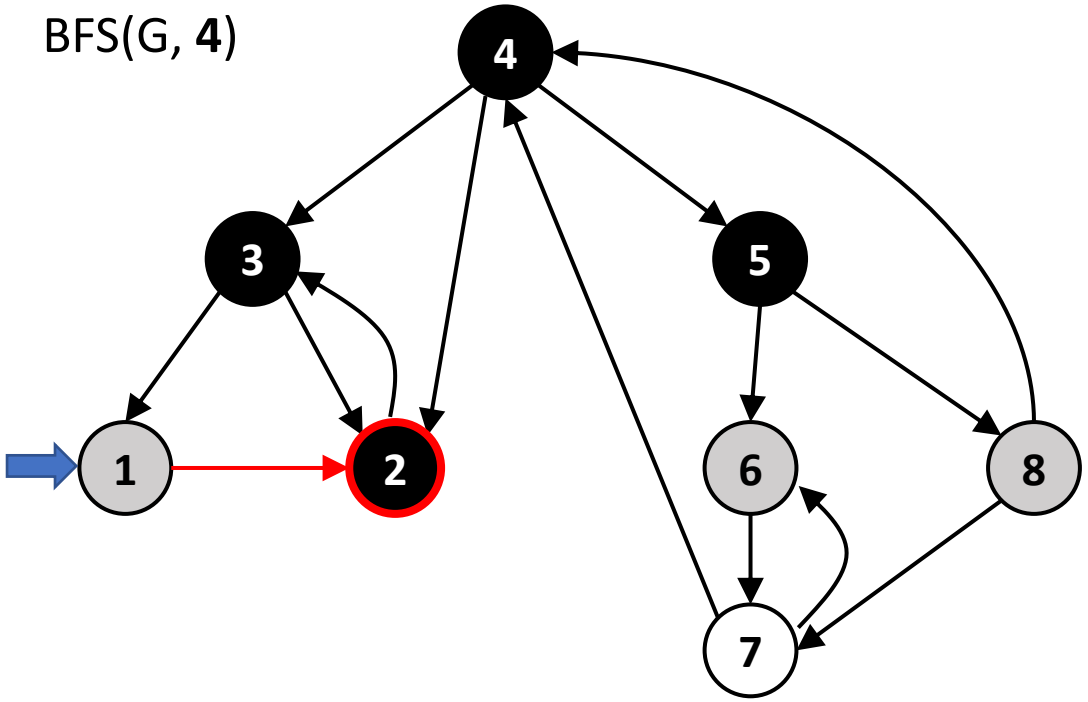
1	1	2
2	5	1

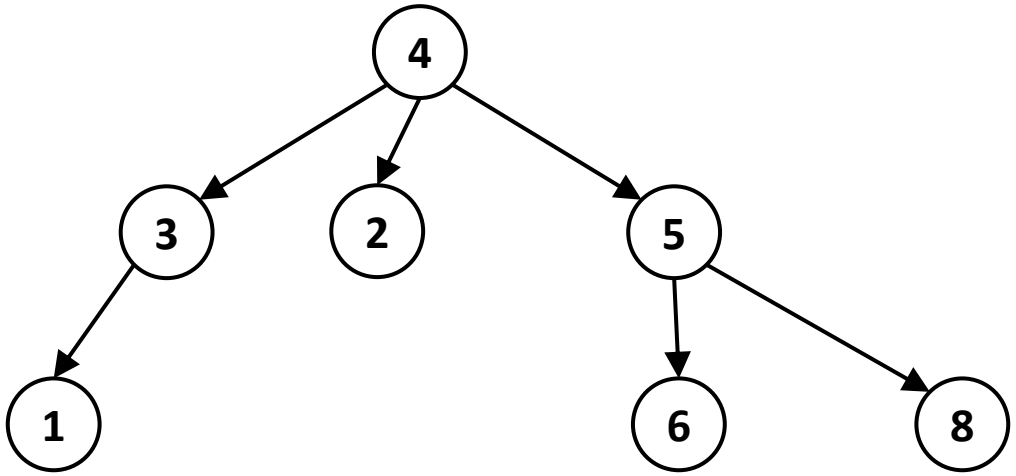
1	2
5	1

2	2	2
1	6	8

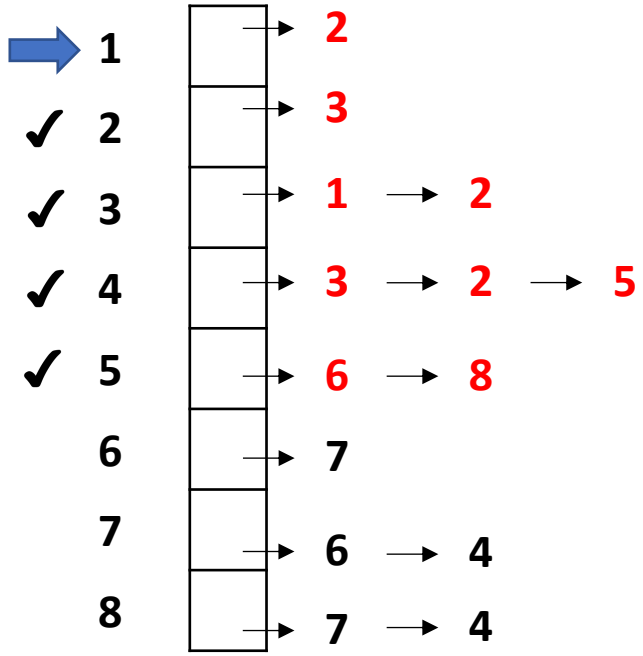
2	2
6	8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

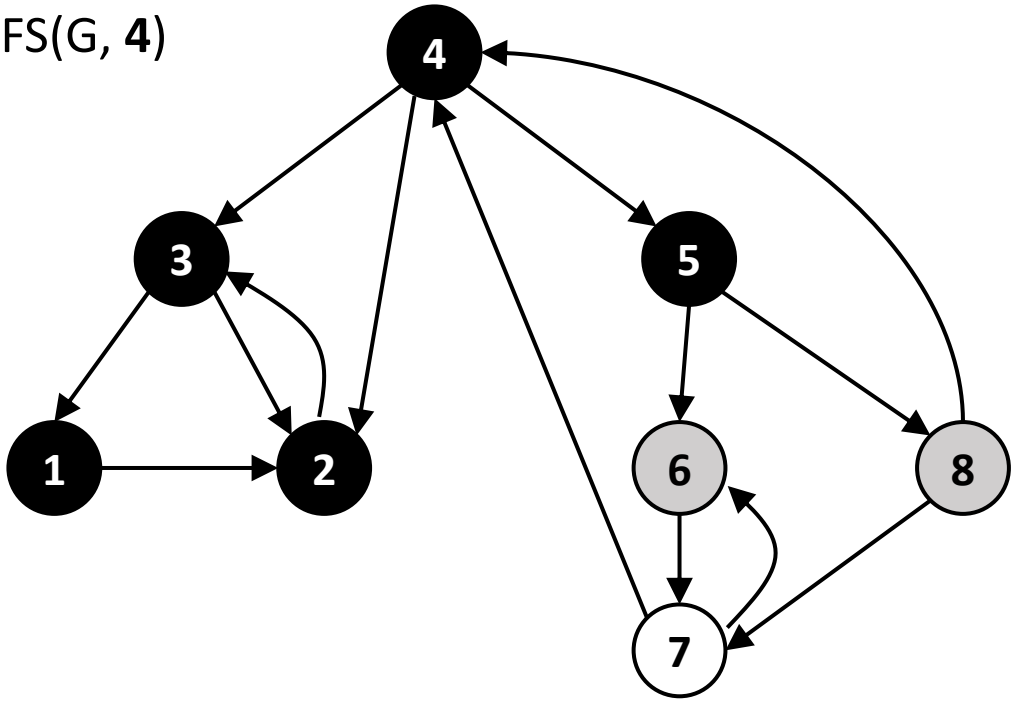
2 2 2

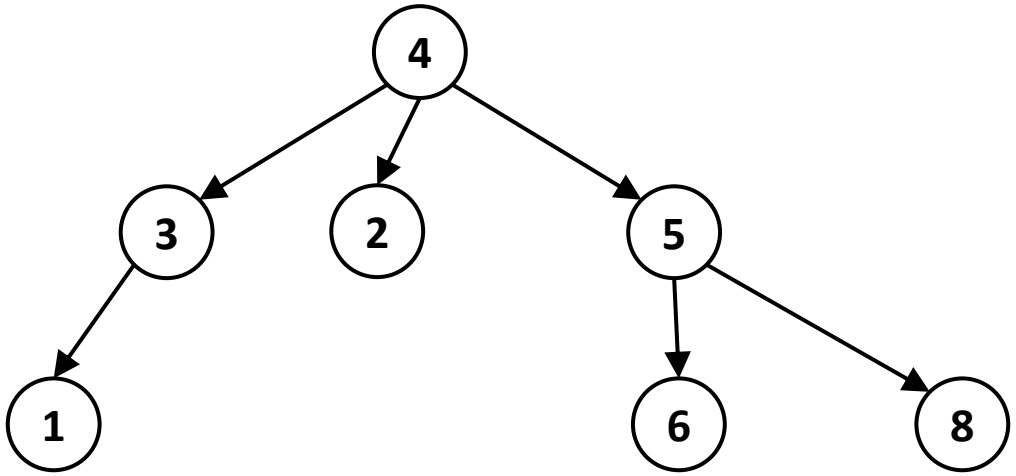
1	6	8

2 2

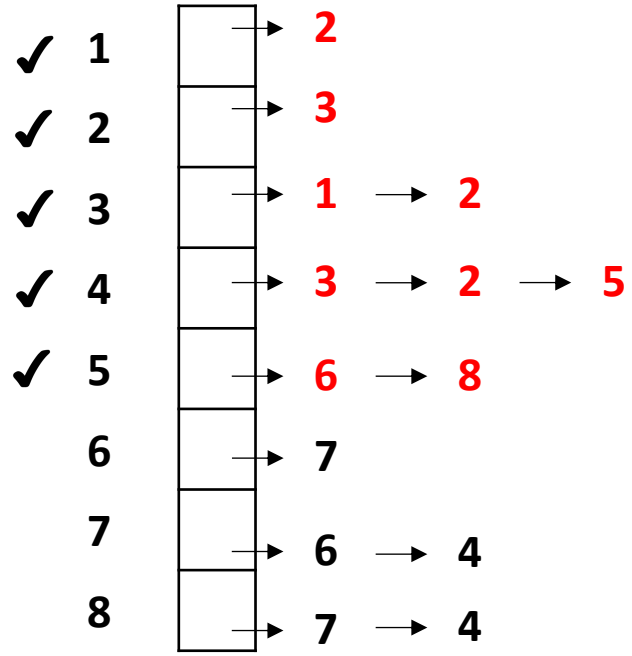
6	8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

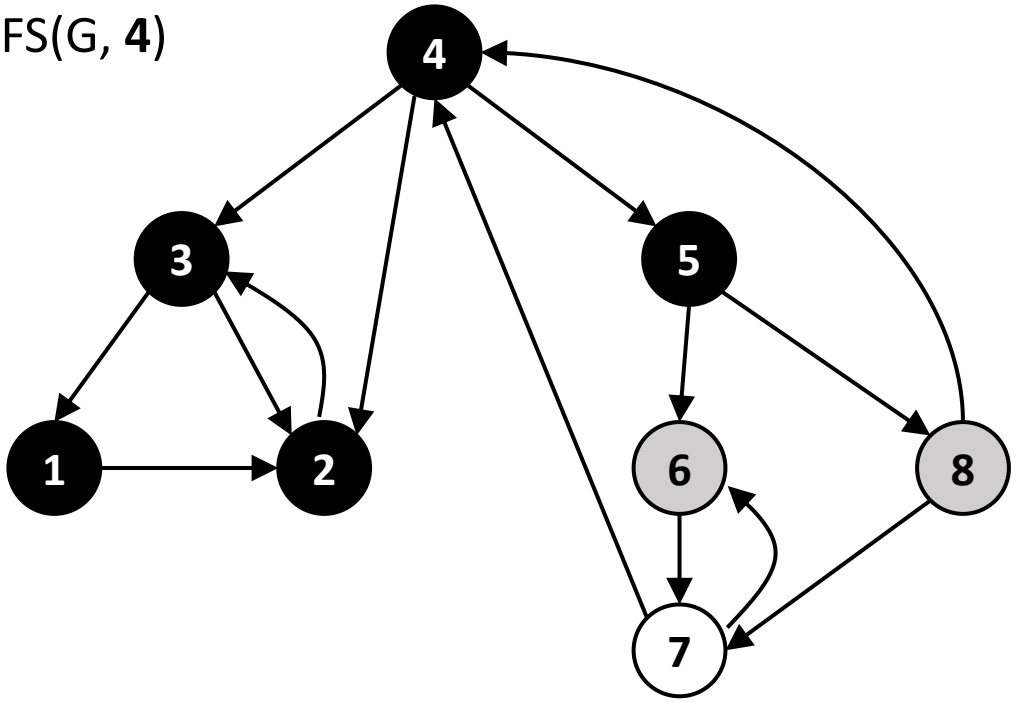
2	5	1

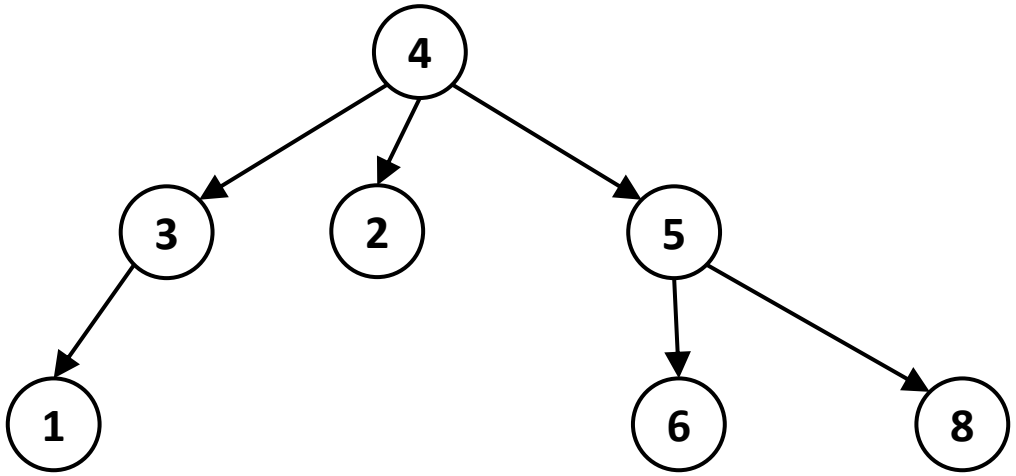
5	1

1	6	8

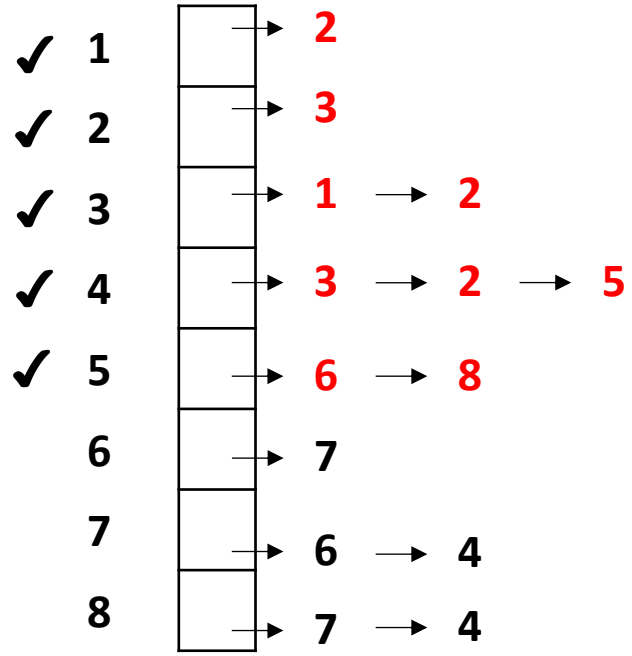
6	8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

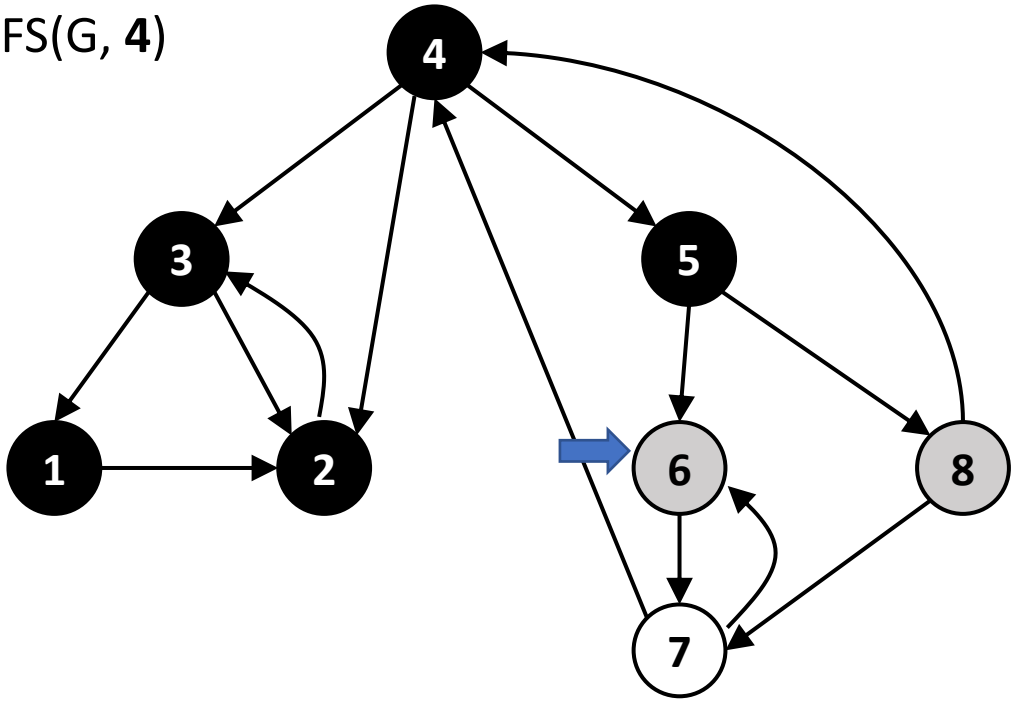
2 2 2

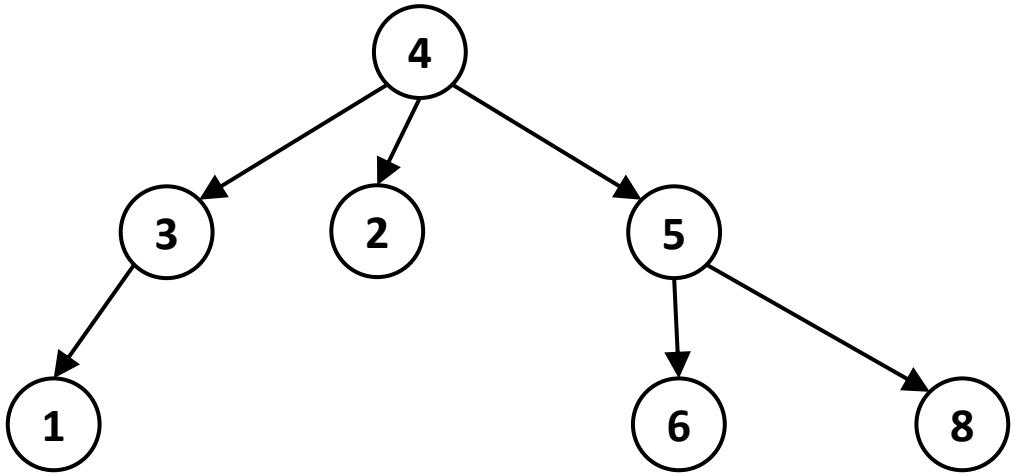
1	6	8

2 2

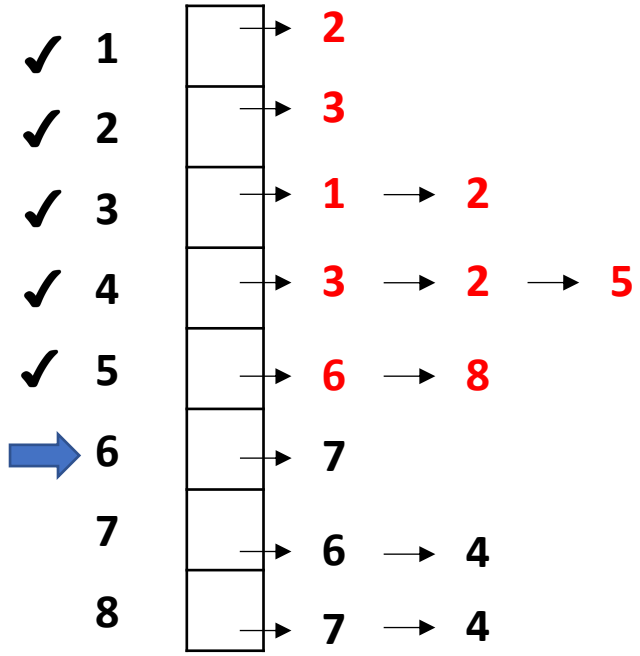
6	8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

2	5	1

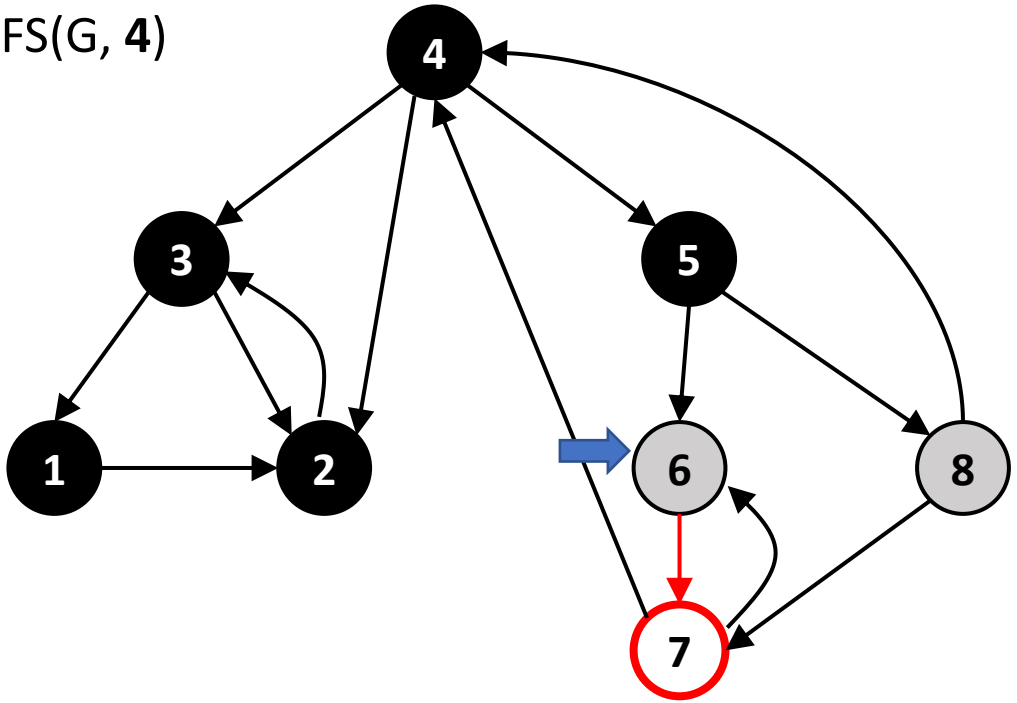
5	1

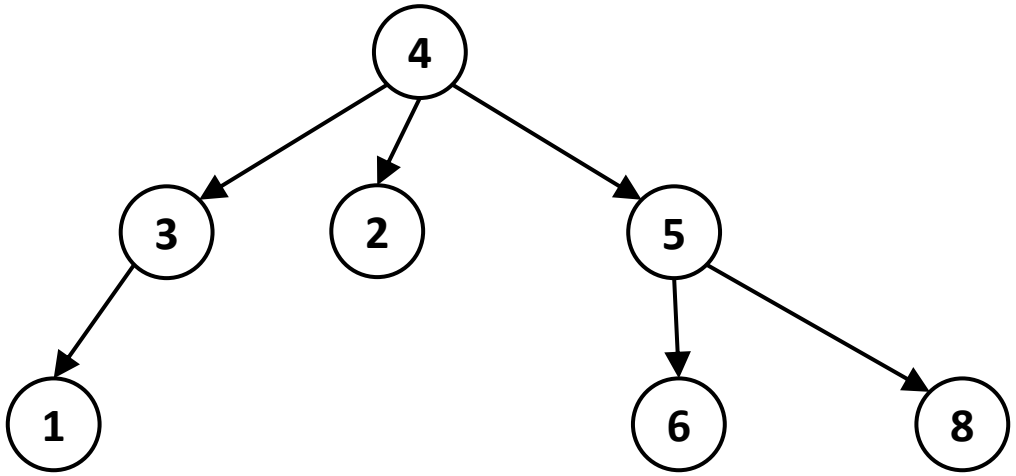
1	6	8

6	8

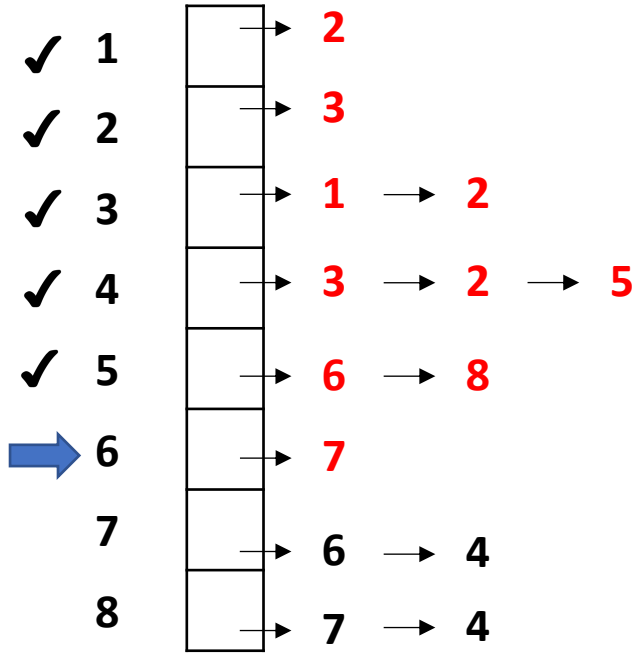
8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

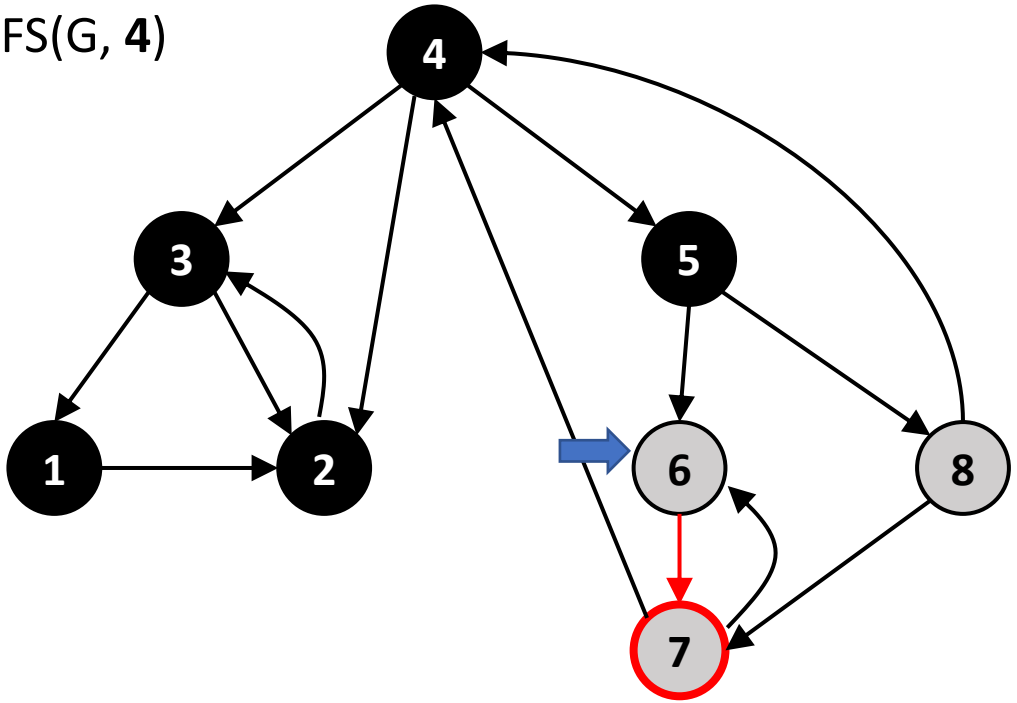
2 2

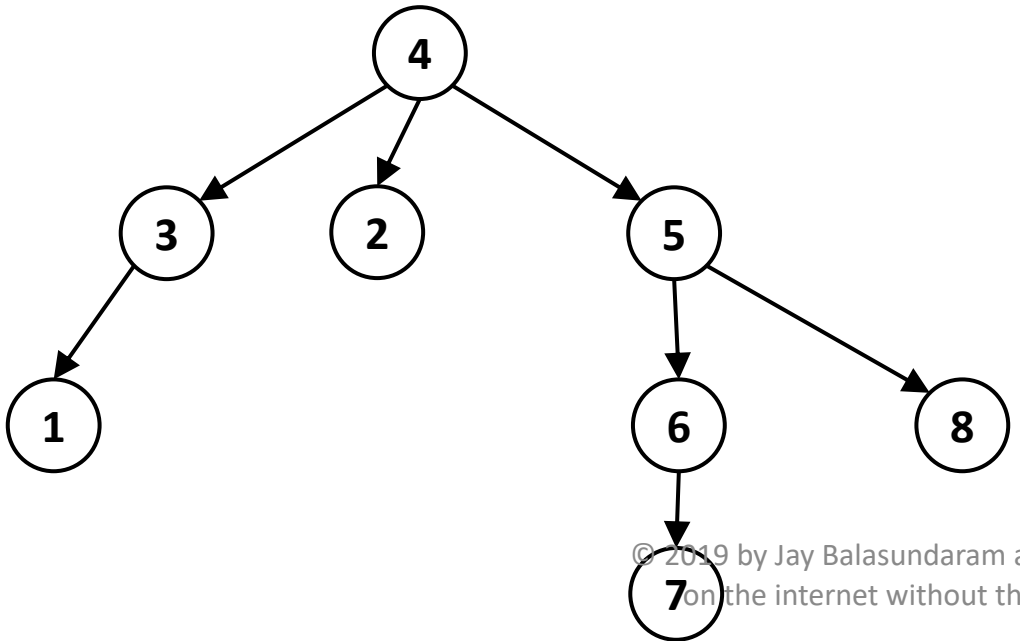
6	8

2

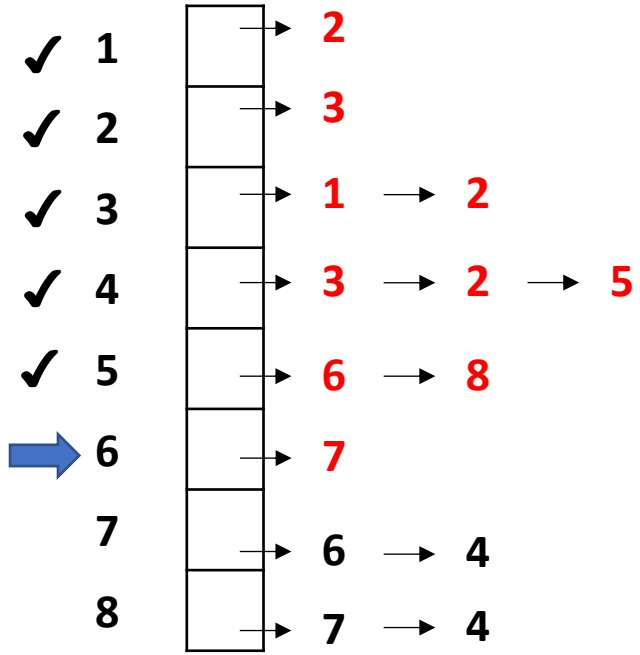
8

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

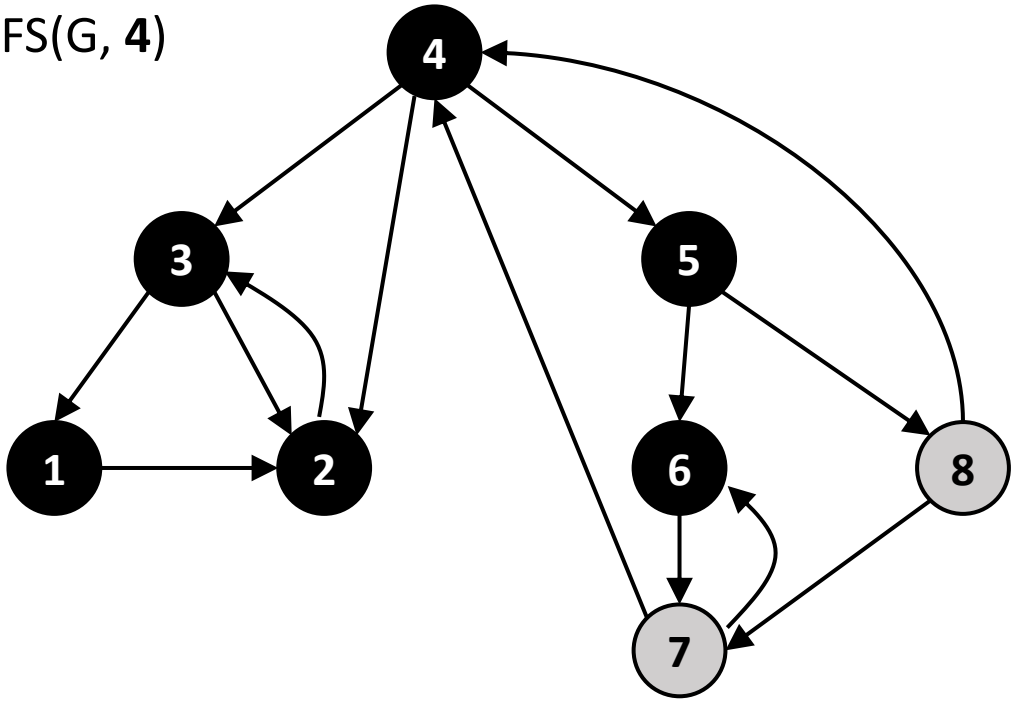
2 2

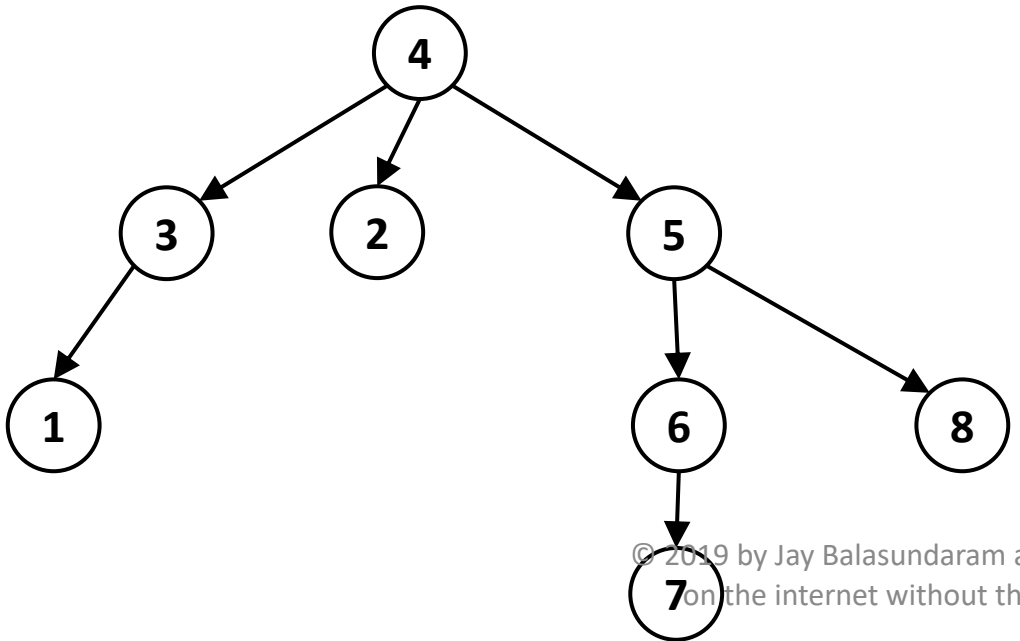
6	8

2 3

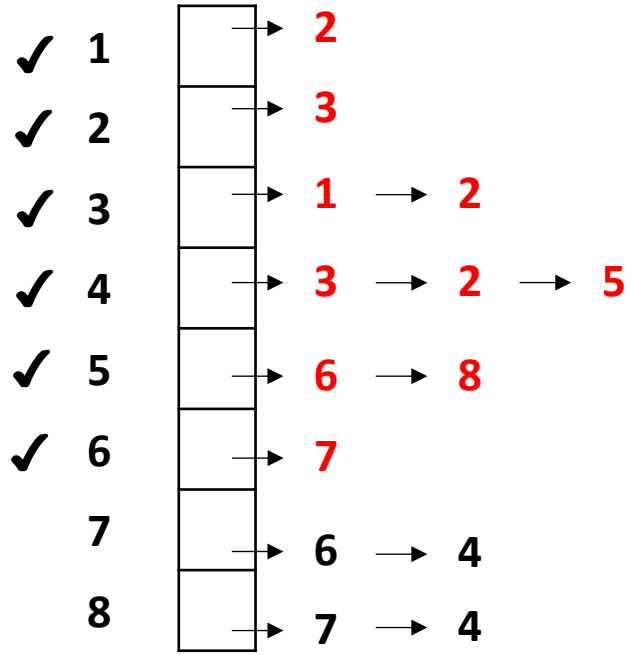
8	7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

2	5	1

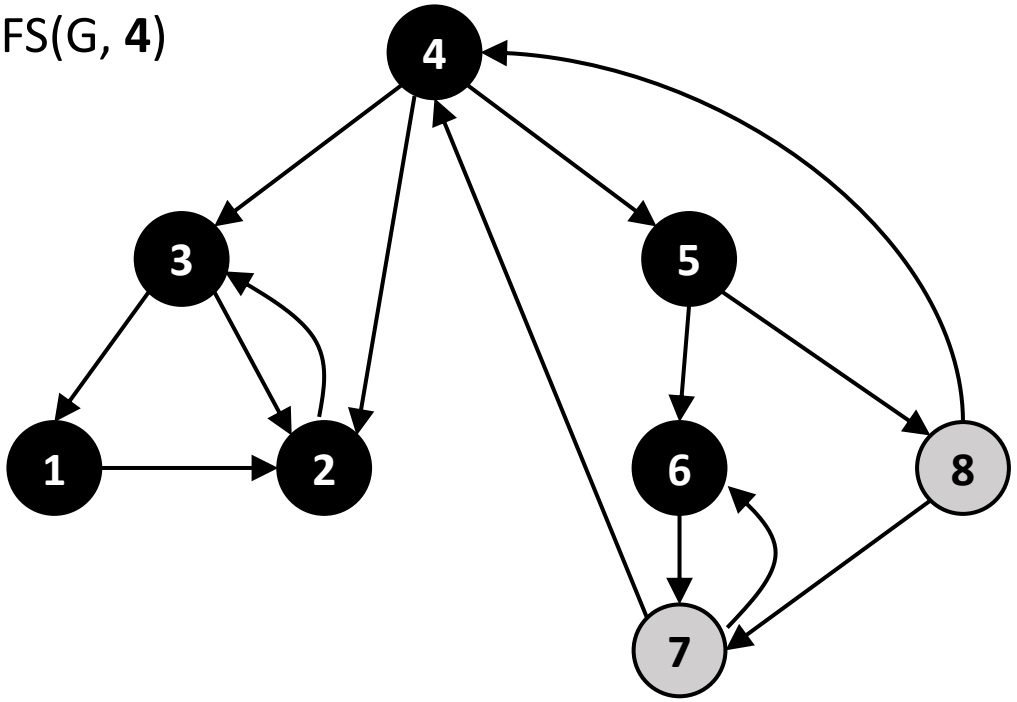
5	1

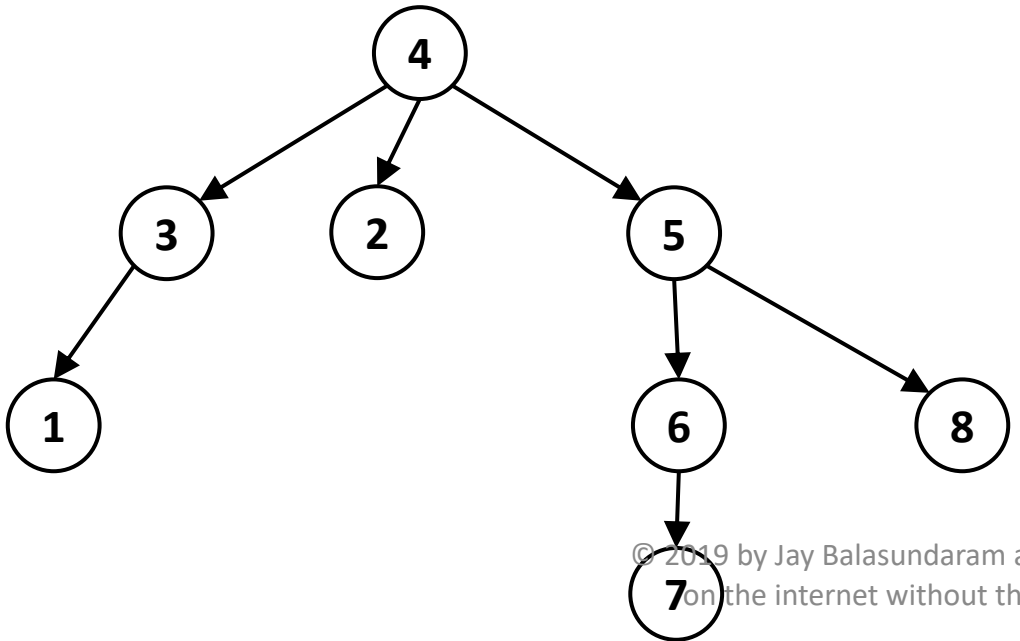
1	6	8

6	8

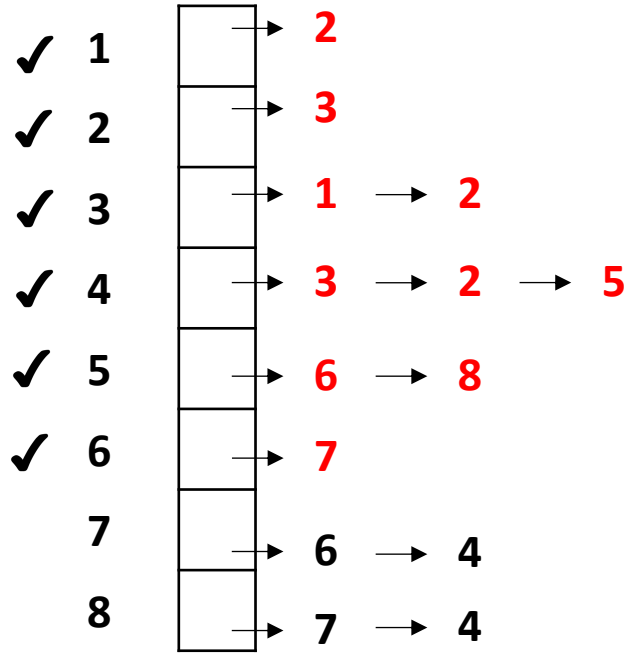
8	7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

2	5	1

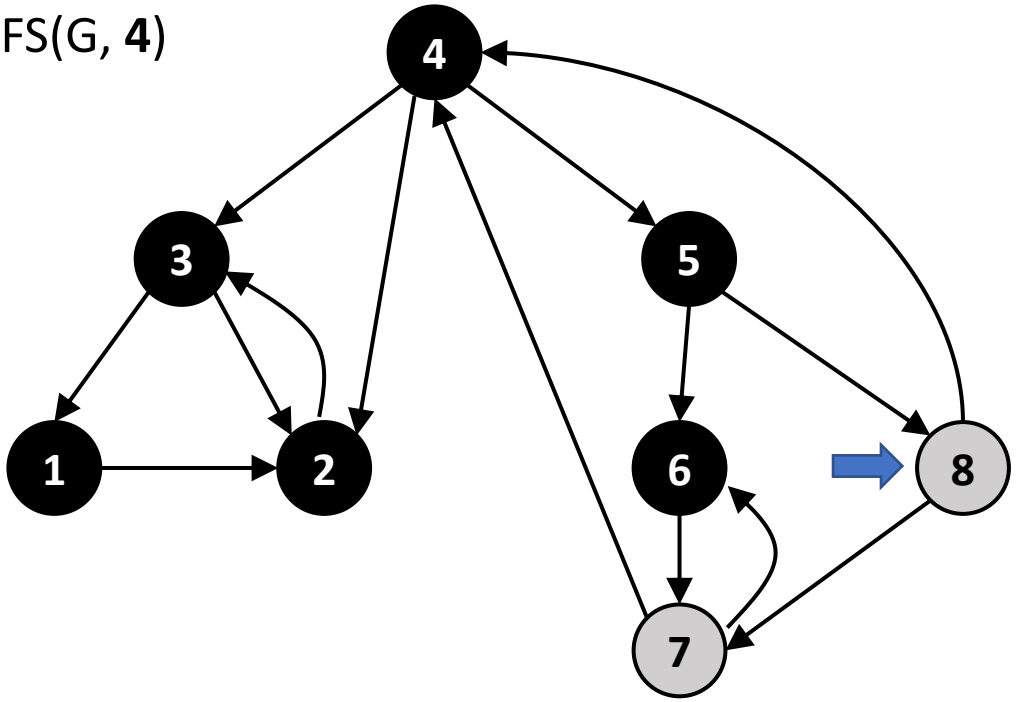
5	1

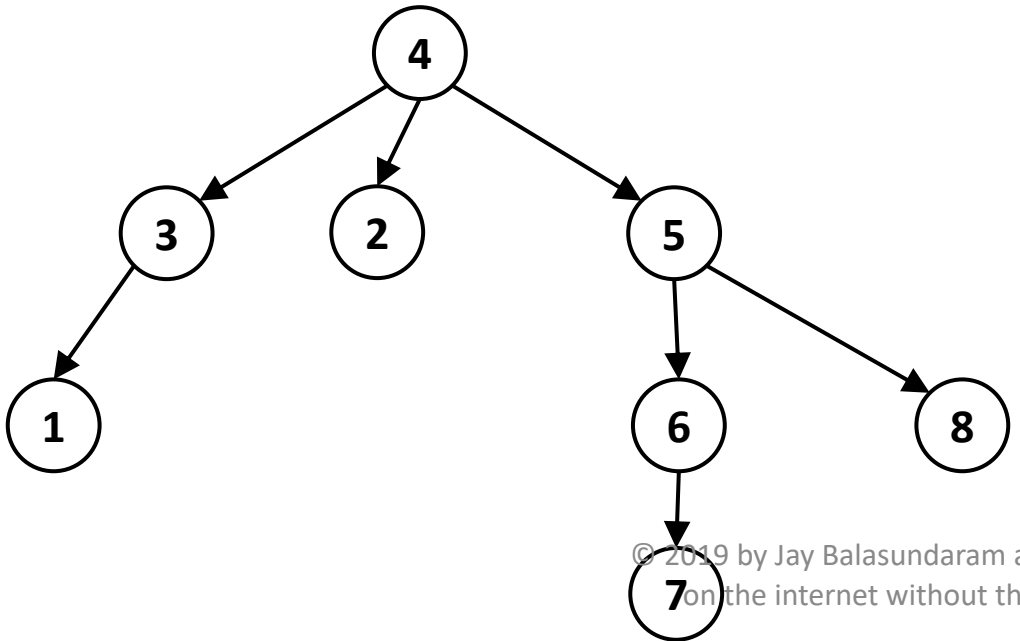
1	6	8

6	8

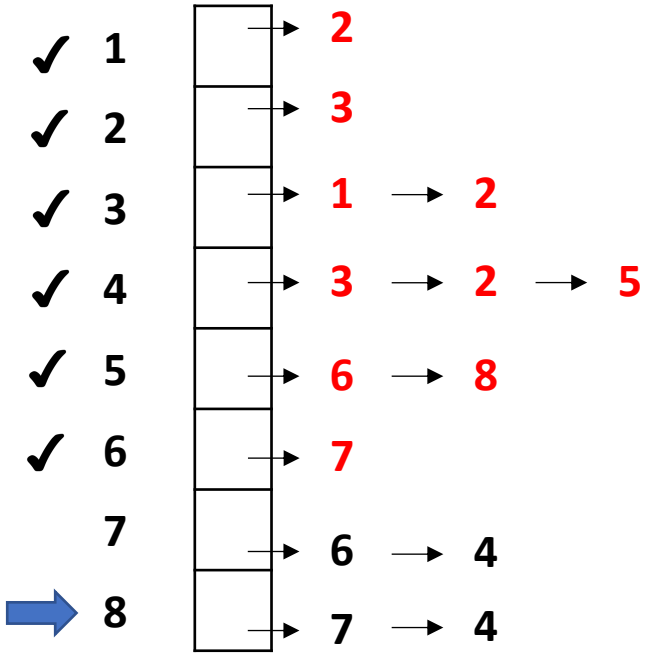
8	7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

2 2

6	8

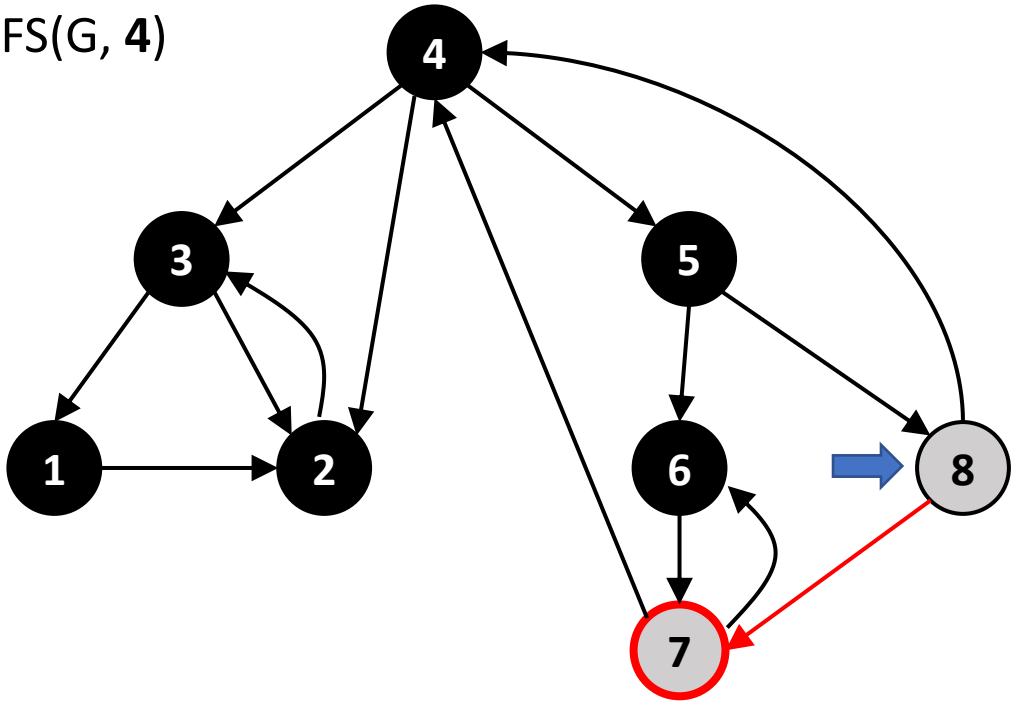
2 3

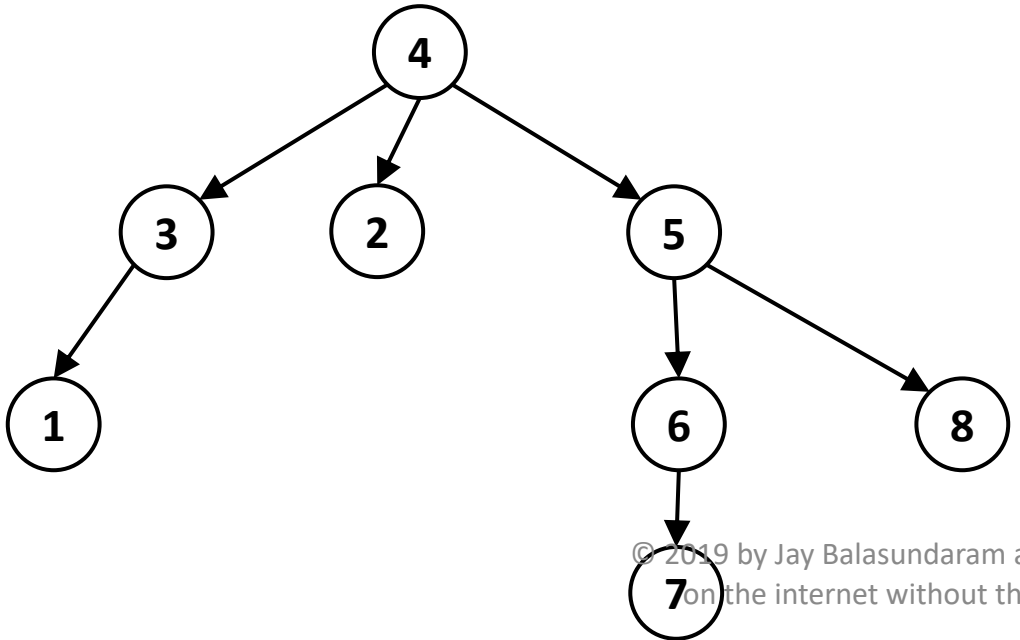
8	7

3

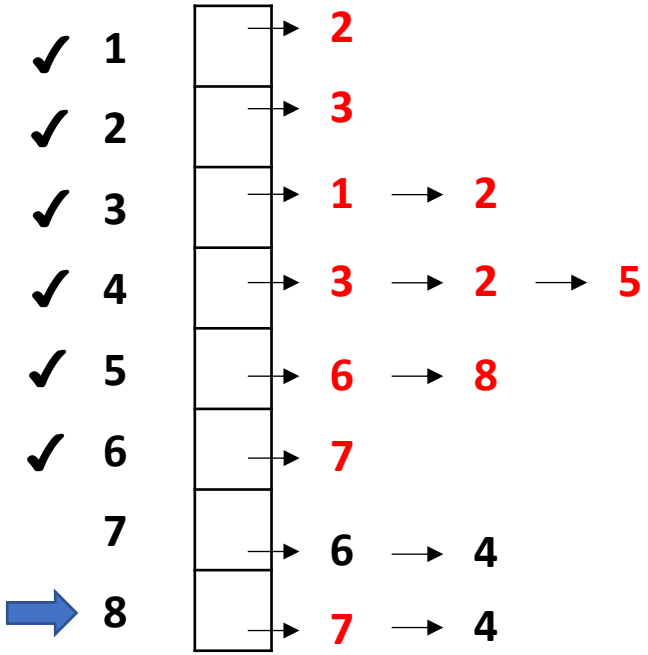
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d =

1	1	1
3	2	5

1	1	2
2	5	1

1	2
5	1

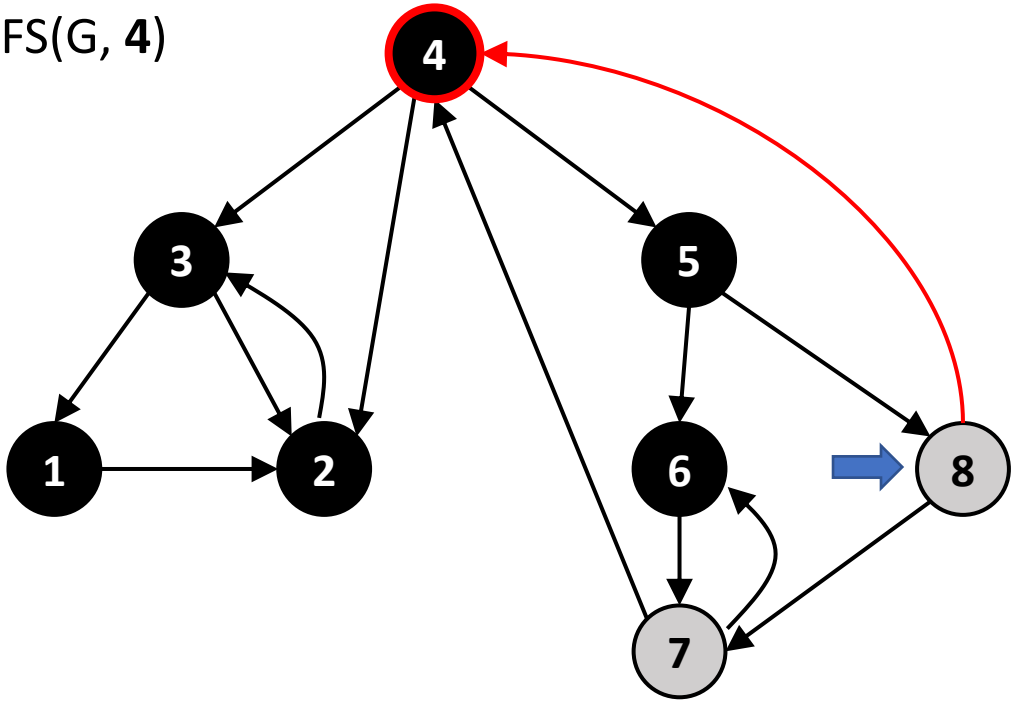
2	2	2
1	6	8

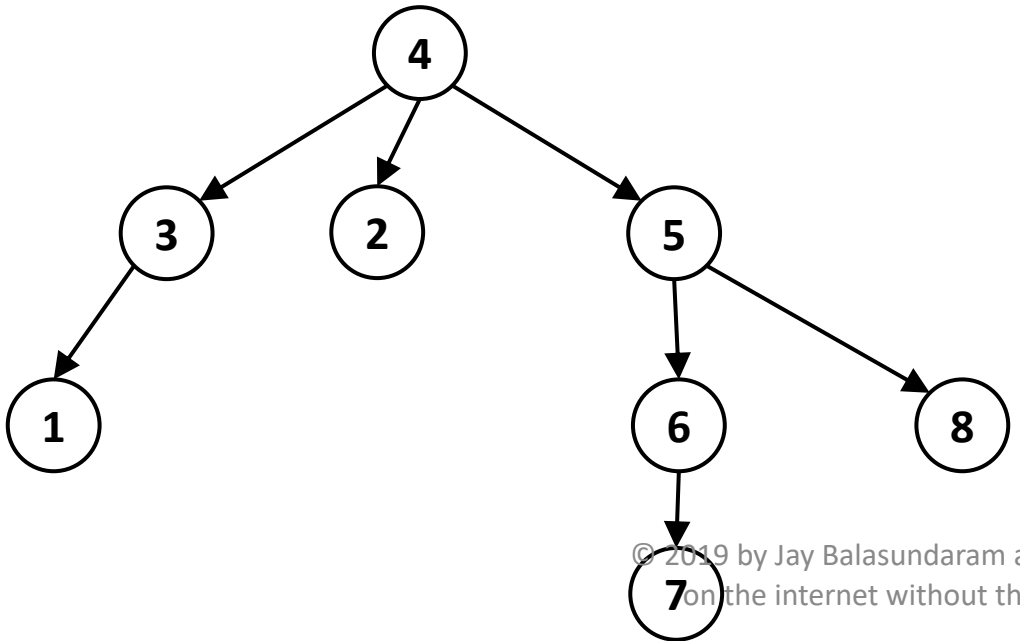
2	2
6	8

2	3
8	7

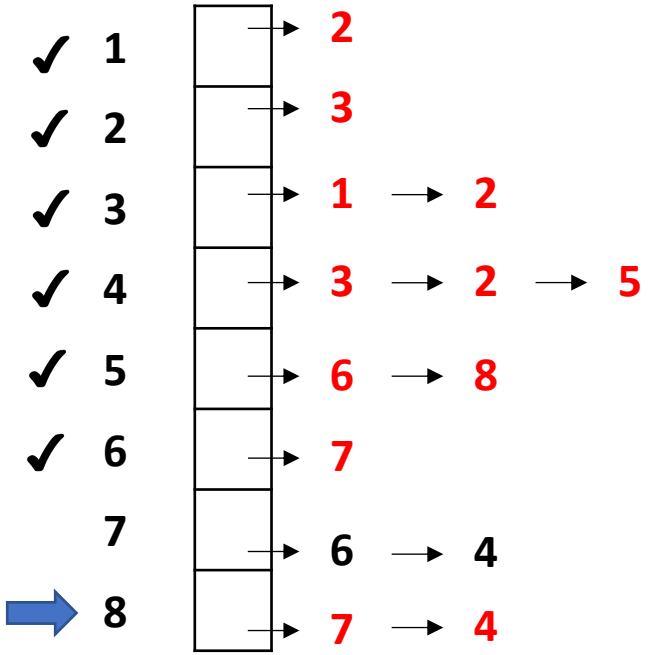
3
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d =

1	1	1
3	2	5

1	1	2
2	5	1

1	2
5	1

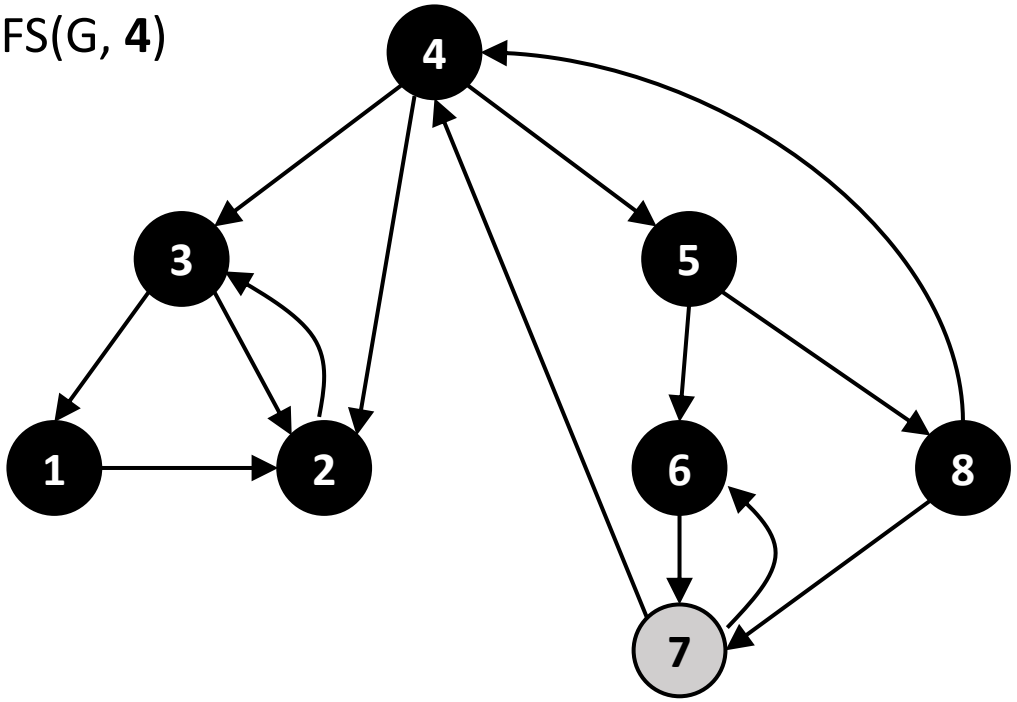
2	2	2
1	6	8

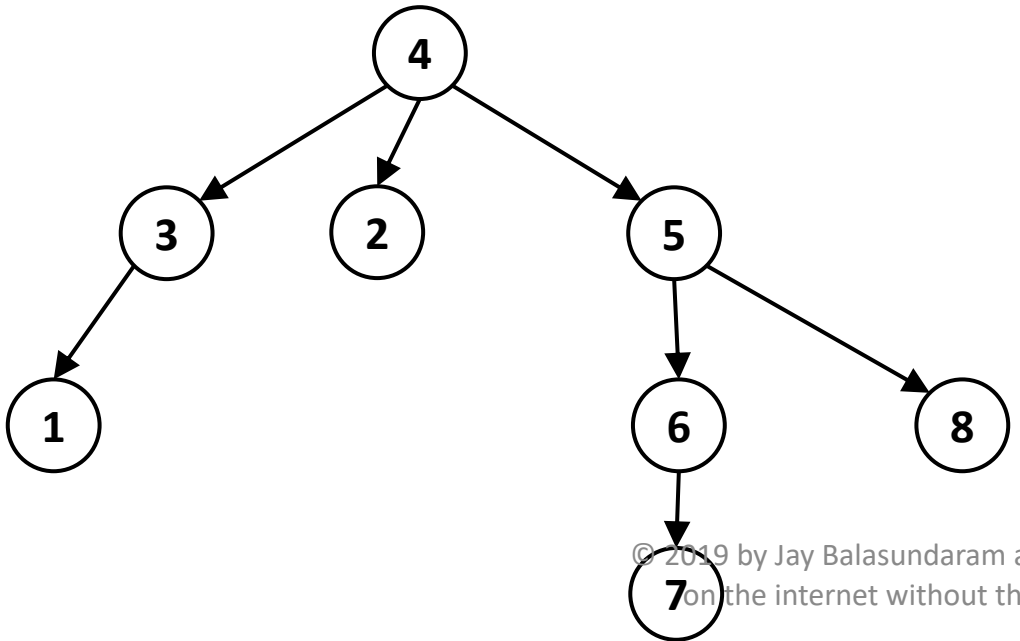
2	2
6	8

2	3
8	7

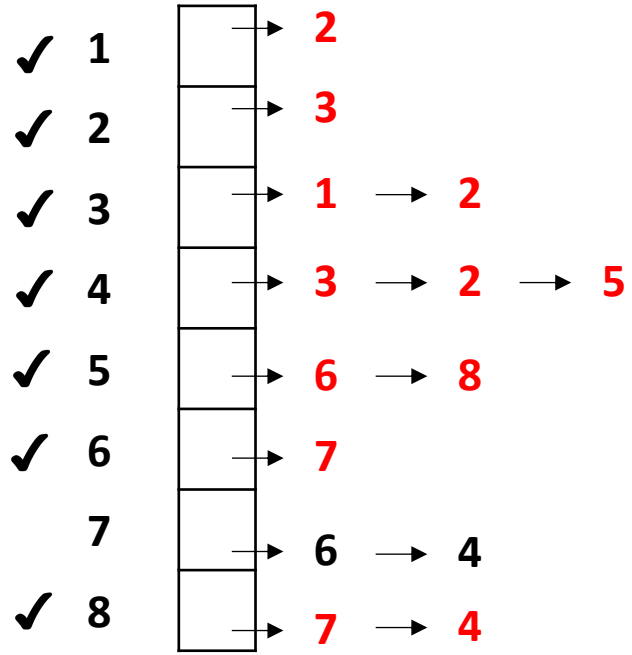
3
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

2	5	1

5	1

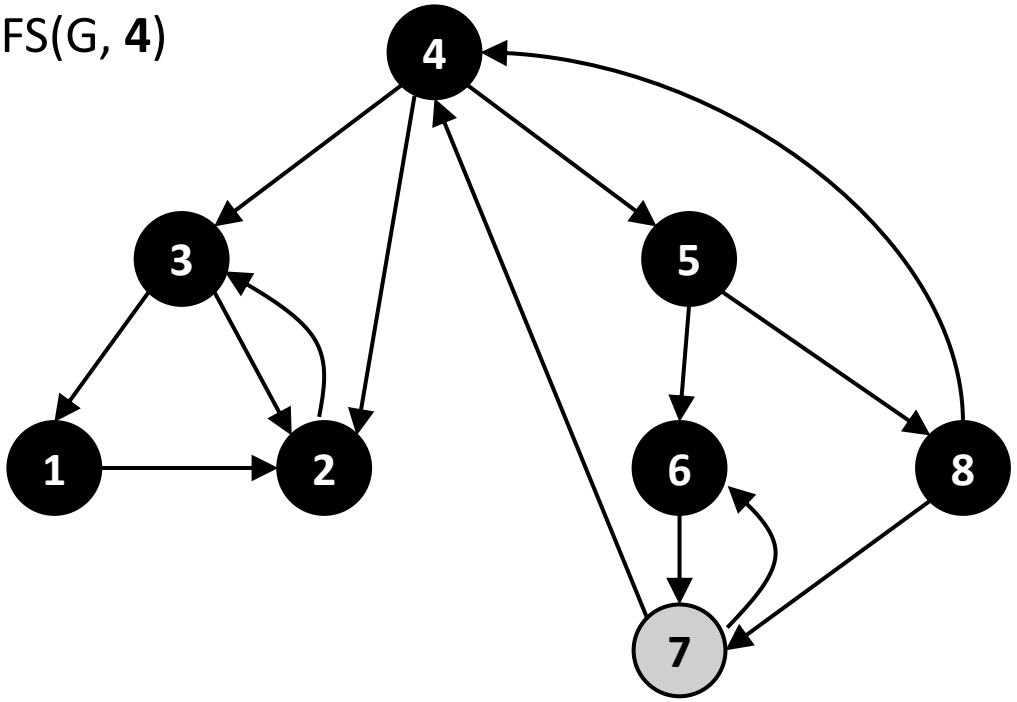
1	6	8

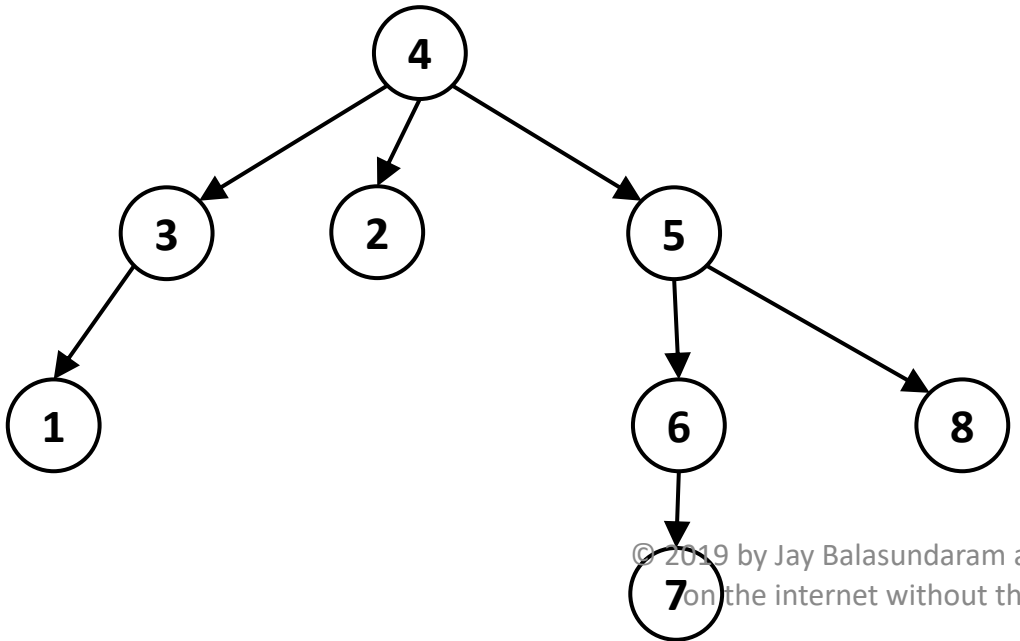
6	8

8	7

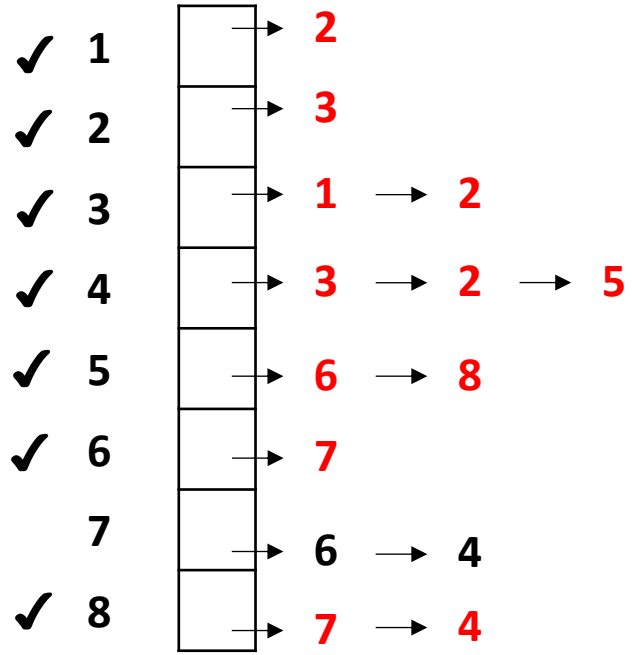
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

2	5	1

5	1

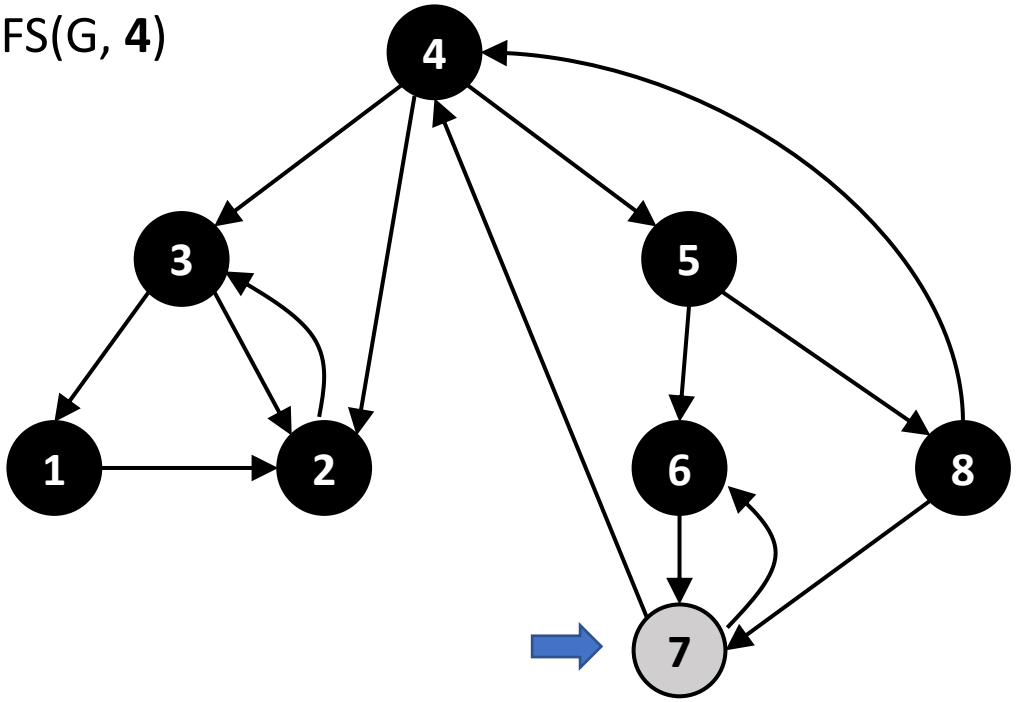
1	6	8

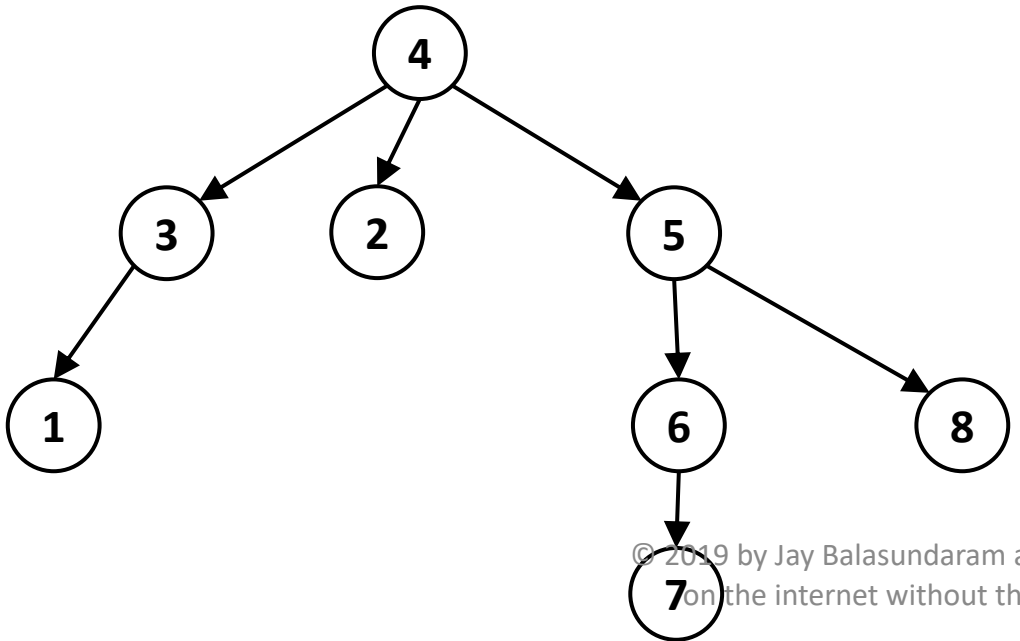
6	8

8	7

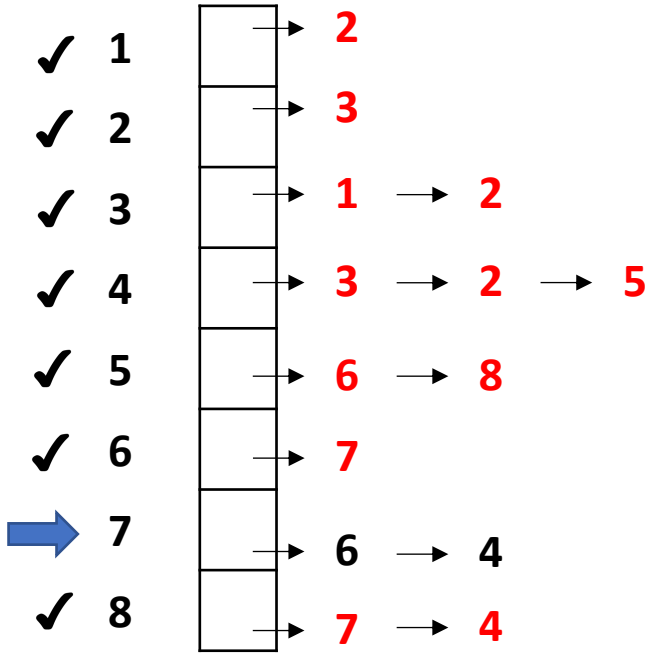
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

2 2

6	8

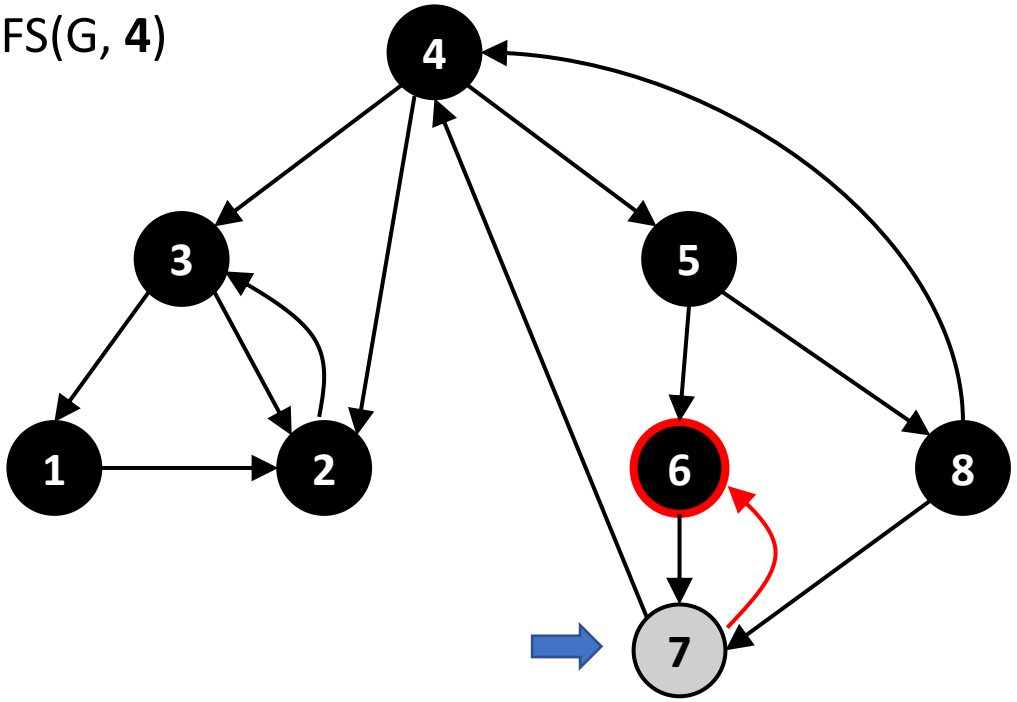
2 3

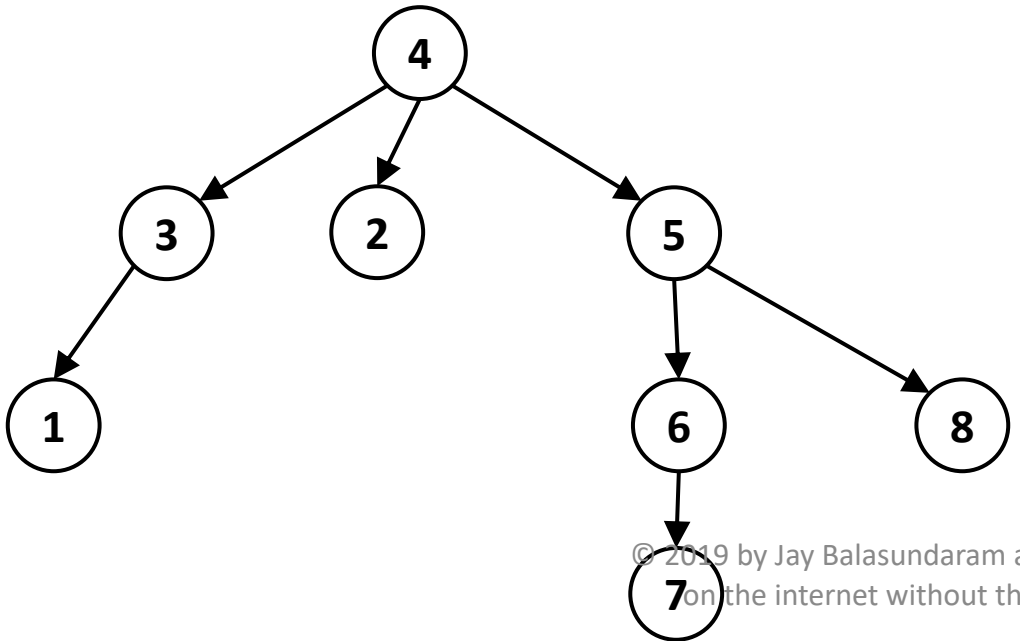
8	7

3

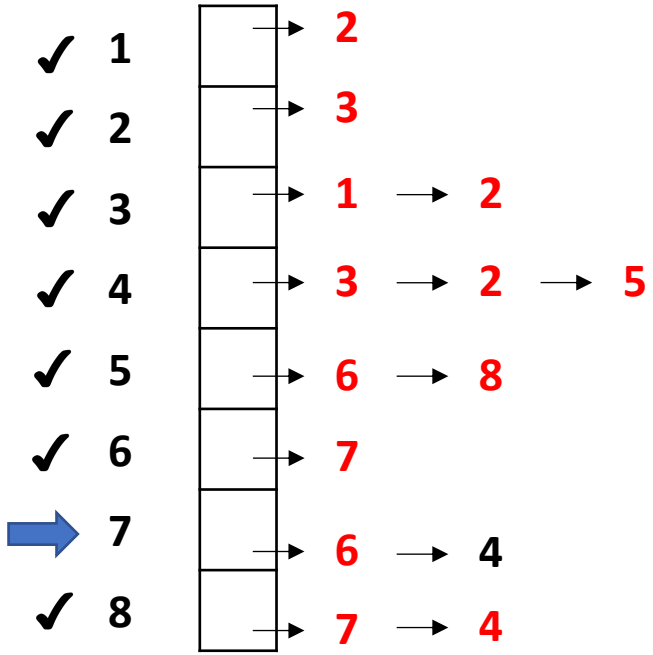
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

2 2

6	8

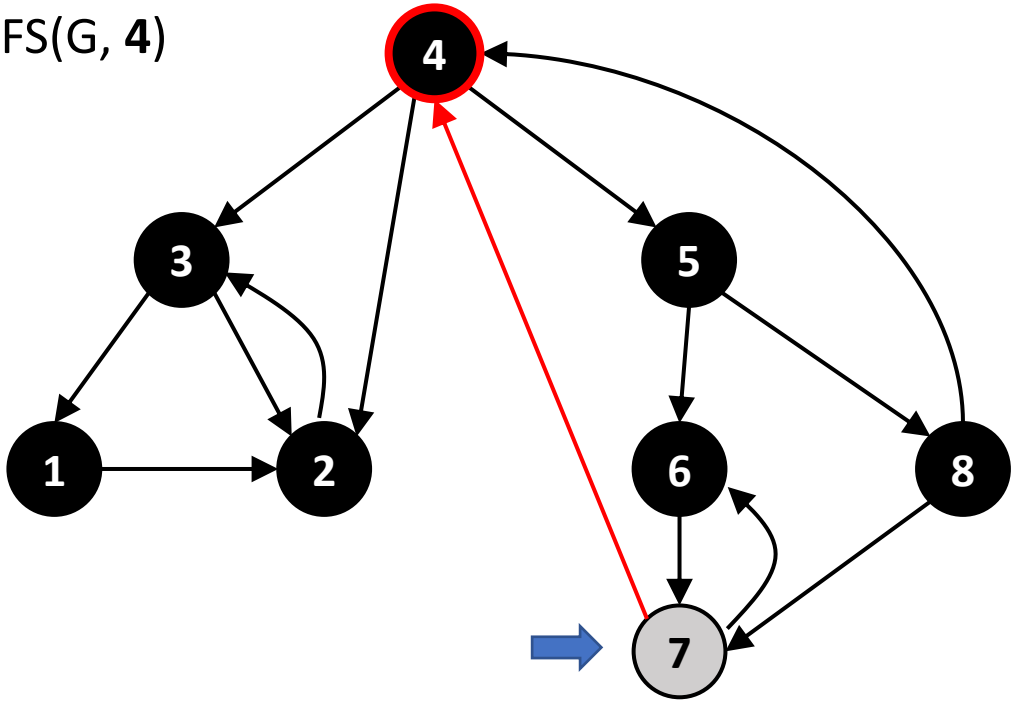
2 3

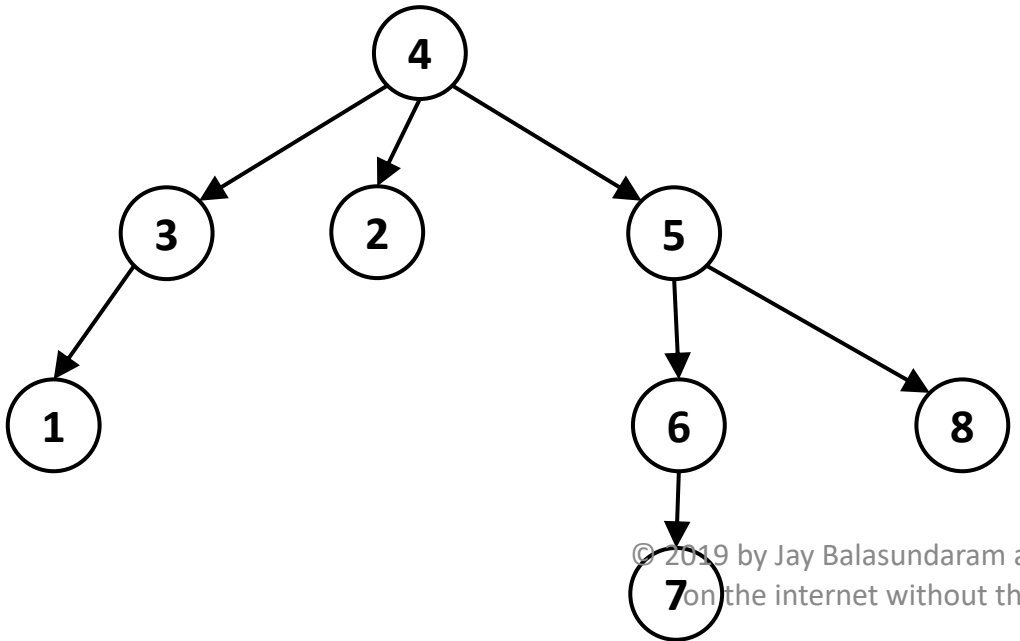
8	7

3

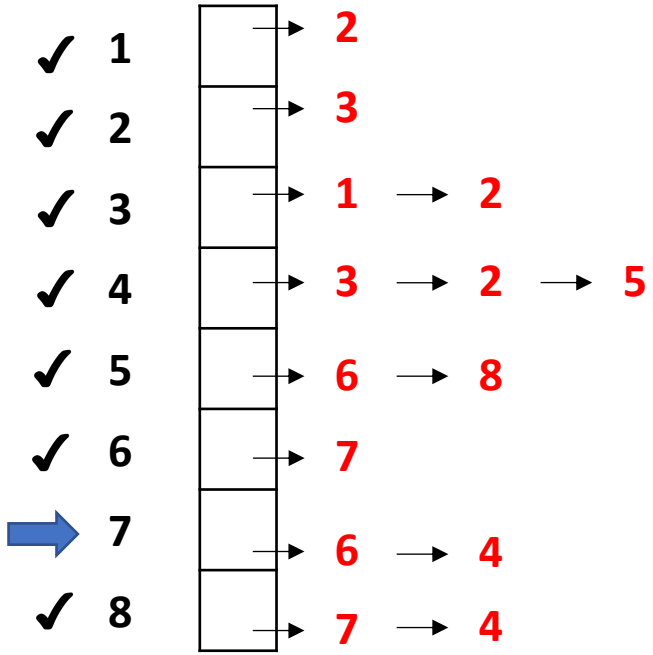
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

2 2

6	8

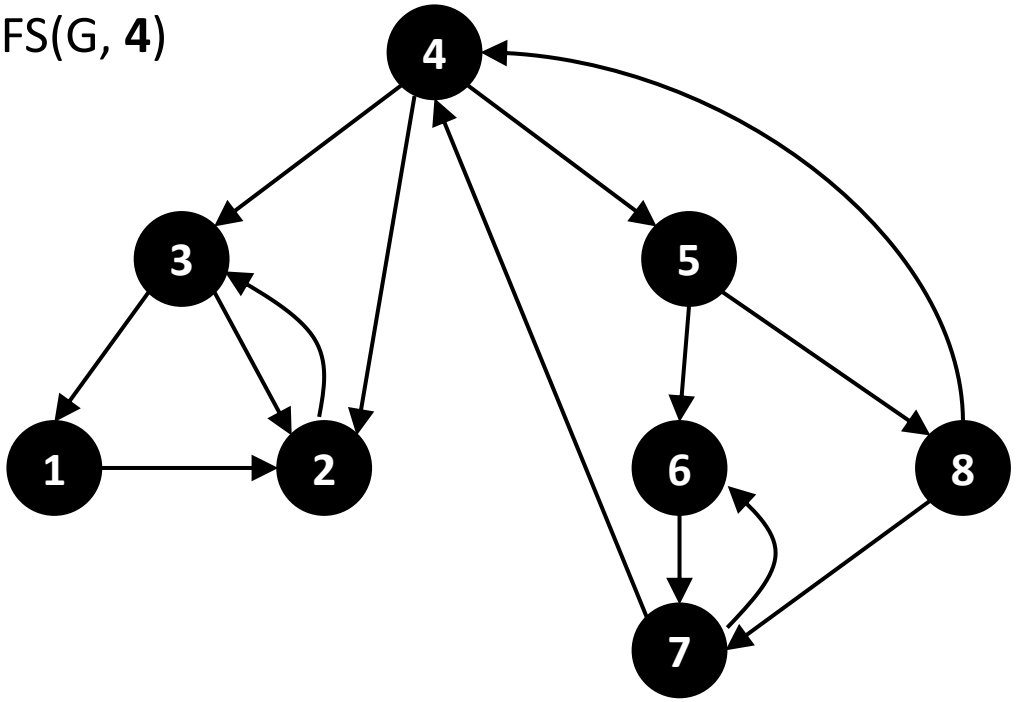
2 3

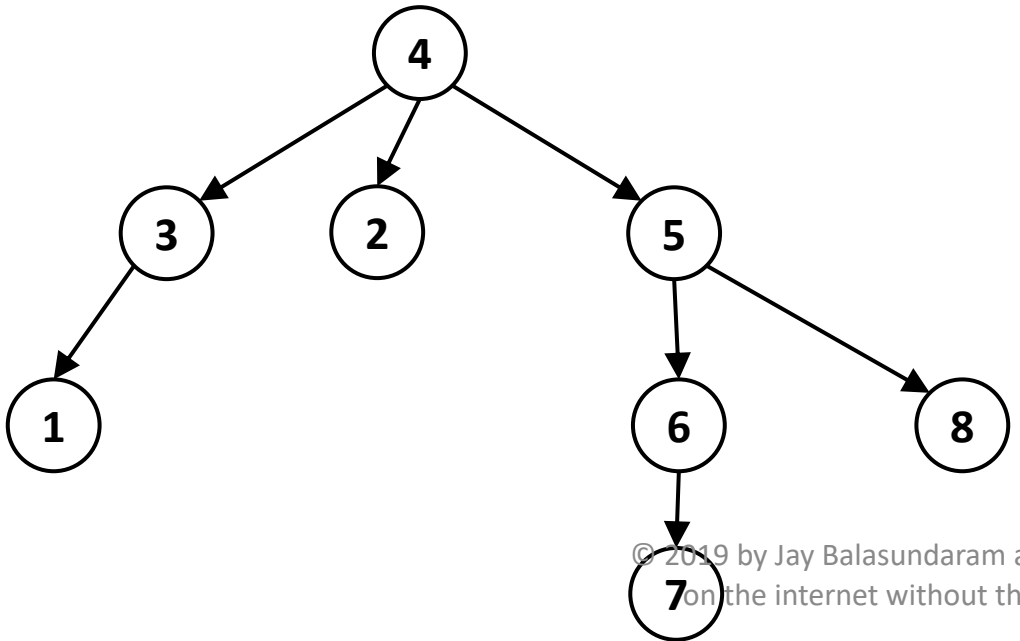
8	7

3

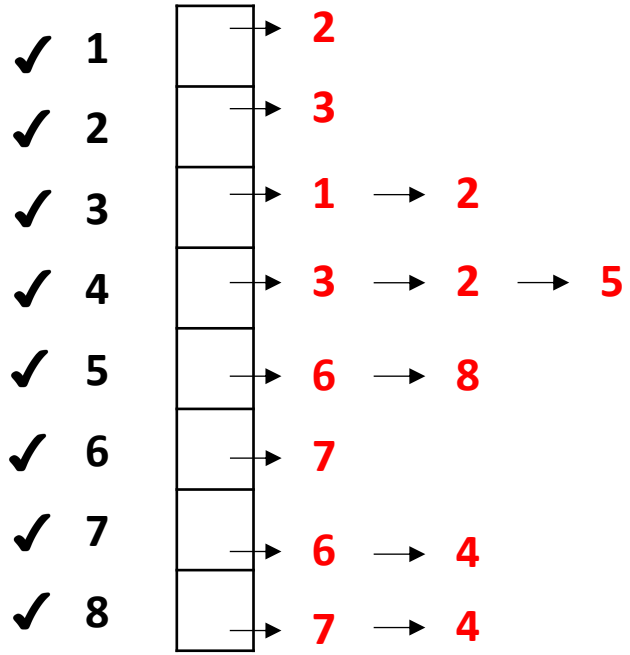
7

BFS(G, **4**)





Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

2	5	1

5	1

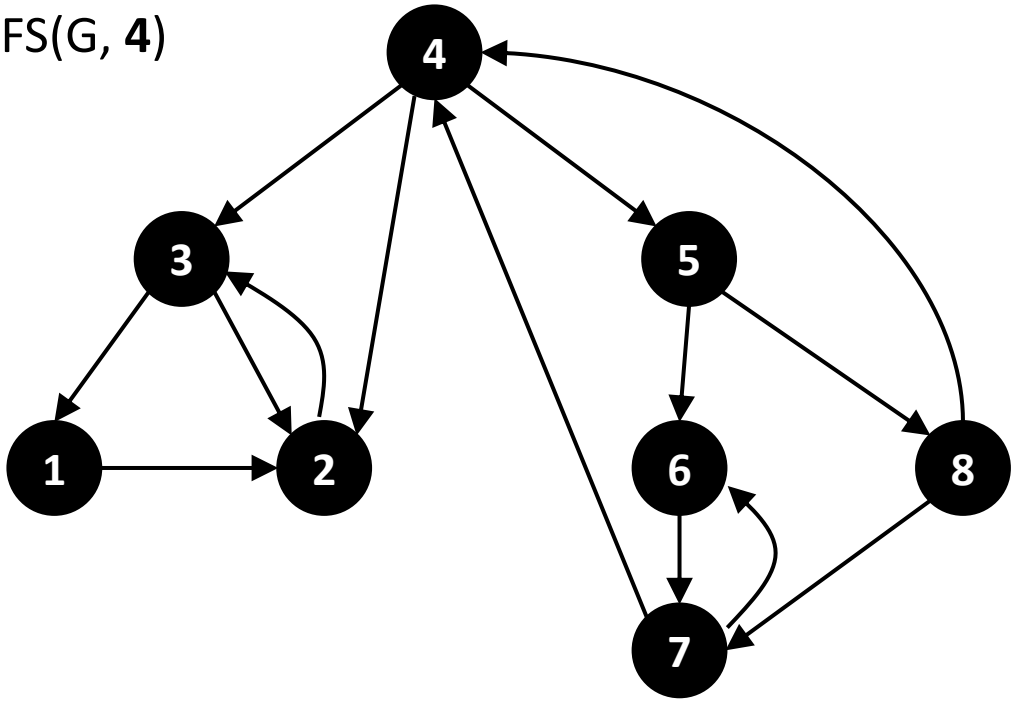
1	6	8

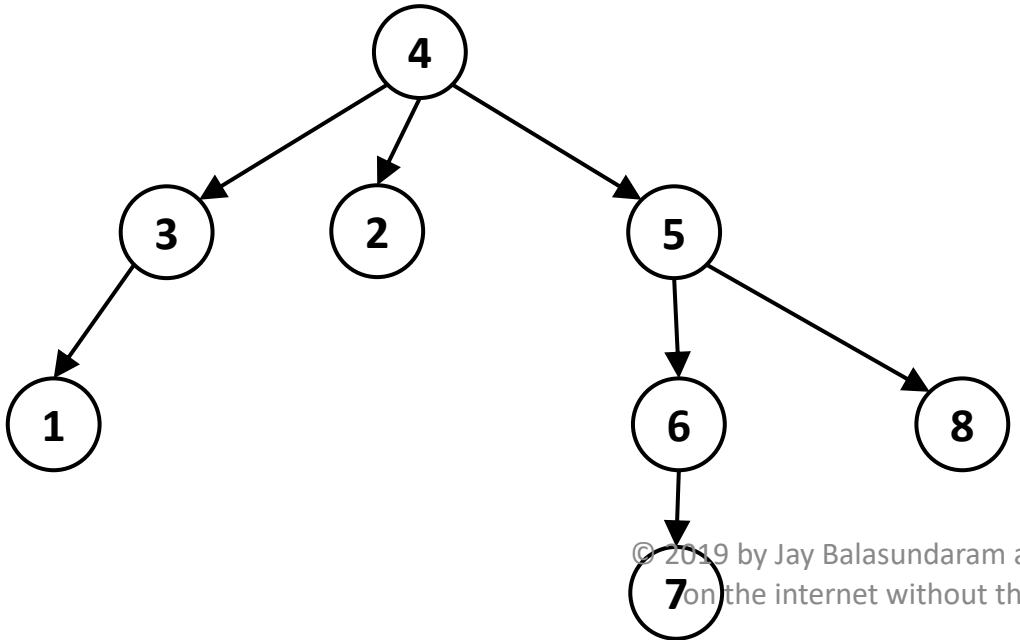
6	8

8	7

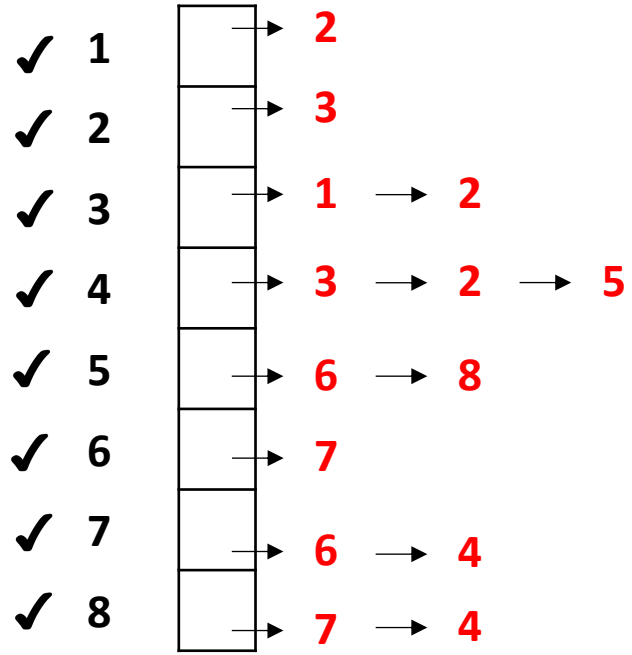
7

BFS(G, **4**)





Adj List of G :



Worst-Case Time Complexity of BFS:

$O(|V| + |E|)$

Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

2 2

6	8

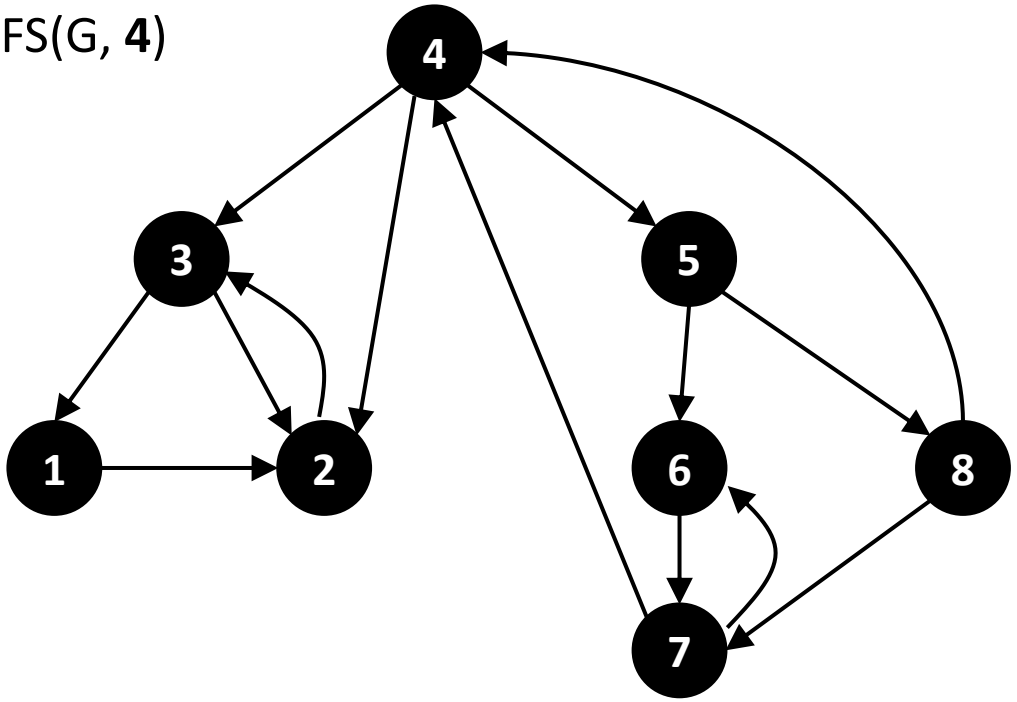
2 3

8	7

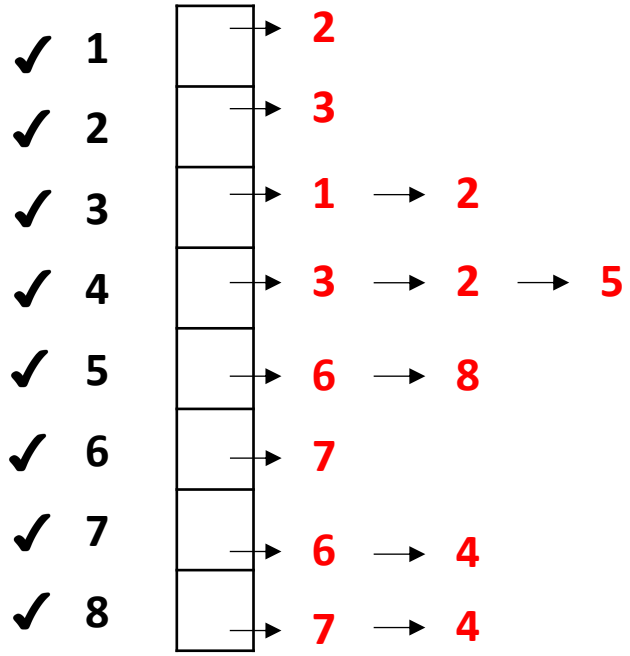
3

7

BFS(G, **4**)



Adj List of G :



Contents of Q :

d = 0

4

d = 1 1 1

3	2	5

1 1 2

2	5	1

1 2

5	1

2 2 2

1	6	8

2 2

6	8

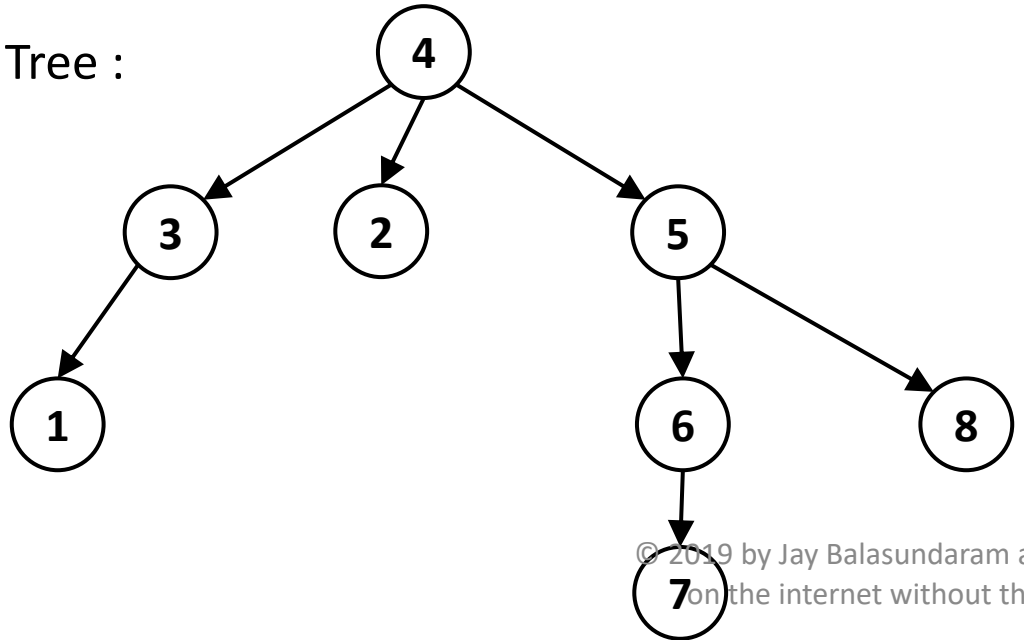
2 3

8	7

3

7

BFS Tree :



Worst-Case Time Complexity of BFS:

$O(|V| + |E|)$